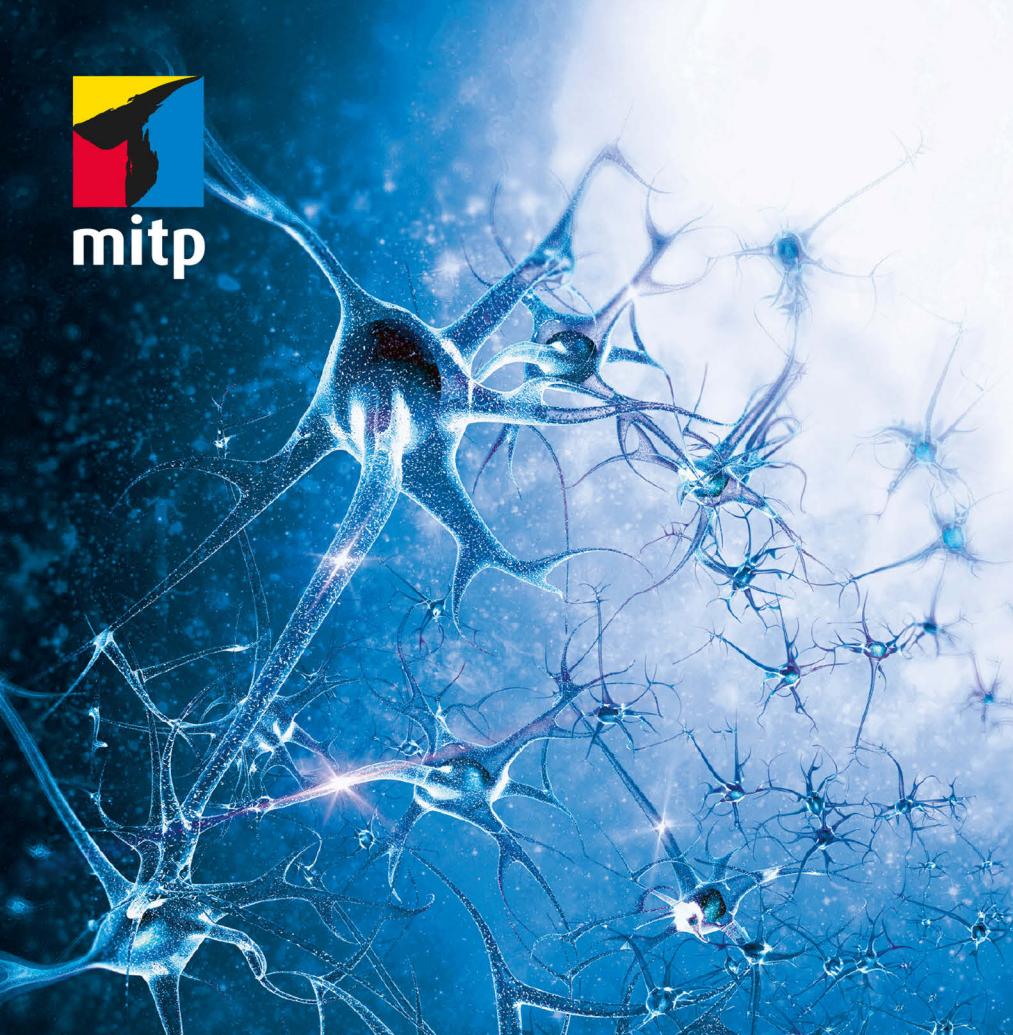




Sebastian  
Raschka

Vahid  
Mirjalili

2.,  
aktualisierte  
und erweiterte  
Auflage



# Machine Learning mit Python und Scikit-learn und TensorFlow

Das umfassende **Praxis-Handbuch**  
für Data Science, Deep Learning und Predictive Analytics



## **Hinweis des Verlages zum Urheberrecht und Digitalem Rechtemanagement (DRM)**

Der Verlag räumt Ihnen mit dem Kauf des ebooks das Recht ein, die Inhalte im Rahmen des geltenden Urheberrechts zu nutzen. Dieses Werk, einschließlich aller seiner Teile, ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Dies gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und Einspeicherung und Verarbeitung in elektronischen Systemen.

Der Verlag schützt seine ebooks vor Missbrauch des Urheberrechts durch ein digitales Rechtemanagement. Bei Kauf im Webshop des Verlages werden die ebooks mit einem nicht sichtbaren digitalen Wasserzeichen individuell pro Nutzer signiert.

Bei Kauf in anderen ebook-Webshops erfolgt die Signatur durch die Shopbetreiber. Angaben zu diesem DRM finden Sie auf den Seiten der jeweiligen Anbieter.

Sebastian Raschka  
Vahid Mirjalili

# Machine Learning mit Python und Scikit-learn und TensorFlow

Das umfassende Praxis-Handbuch für  
Data Science, Deep Learning und Predictive Analytics

2., aktualisierte und erweiterte Auflage

Übersetzung aus dem Amerikanischen  
von Knut Lorenzen



**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.

ISBN 978-3-95845-734-8

2. Auflage 2018

[www.mitp.de](http://www.mitp.de)

E-Mail: [mitp-verlag@sigloch.de](mailto:mitp-verlag@sigloch.de)

Telefon: +49 7953 / 7189 - 079

Telefax: +49 7953 / 7189 - 082

© 2018 mitp Verlags GmbH & Co. KG, Frechen

Dieses Werk, einschließlich aller seiner Teile, ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Dies gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürfen.

Übersetzung der amerikanischen Originalausgabe:

Raschka, Sebastian; Mirjalili, Vahid: Python Machine Learning – Second Edition  
Copyright © Packt Publishing 2017. First Published in the English language under the title ‚Python Machine Learning – Second Edition – (9781787125933)‘  
German language edition Copyright © by mitp Verlags GmbH und Co. KG.  
All rights reserved.

Lektorat: Sabine Schulz

Sprachkorrektorat: Maren Feilen

Coverbild: Shantanu N. Zagade, © Packt Publishing

Satz: III-satz, Husby, [www.drei-satz.de](http://www.drei-satz.de)

# Inhaltsverzeichnis

Über die Autoren .....	15
Über die Korrektoren .....	17
Einleitung .....	19
<b>1 Wie Computer aus Daten lernen können .....</b>	<b>25</b>
1.1 Intelligente Maschinen, die Daten in Wissen verwandeln .....	25
1.2 Die drei Arten des Machine Learnings .....	26
1.2.1 Mit überwachtem Lernen Vorhersagen treffen .....	26
1.2.2 Interaktive Aufgaben durch verstärkendes Lernen lösen ..	29
1.2.3 Durch unüberwachtes Lernen verborgene Strukturen erkennen .....	31
1.3 Grundlegende Terminologie und Notation .....	32
1.4 Entwicklung eines Systems für das Machine Learning .....	34
1.4.1 Vorverarbeitung: Daten in Form bringen .....	35
1.4.2 Trainieren und Auswählen eines Vorhersagmodells .....	36
1.4.3 Bewertung von Modellen und Vorhersage anhand unbekannter Dateninstanzen .....	37
1.5 Machine Learning mit Python .....	37
1.5.1 Python-Pakete installieren .....	37
1.5.2 Verwendung der Python-Distribution Anaconda .....	38
1.5.3 Pakete für wissenschaftliches Rechnen, Data Science und Machine Learning .....	38
1.6 Zusammenfassung .....	39
<b>2 Lernalgorithmen für die Klassifizierung trainieren .....</b>	<b>41</b>
2.1 Künstliche Neuronen: Ein kurzer Blick auf die Anfänge des Machine Learnings .....	41
2.1.1 Formale Definition eines künstlichen Neurons .....	42
2.1.2 Die Perzeptron-Lernregel .....	44
2.2 Implementierung eines Perzeptron-Lernalgorithmus in Python ..	47
2.2.1 Eine objektorientierte Perzeptron-API .....	47
2.2.2 Trainieren eines Perzeptron-Modells auf die Iris-Datensammlung .....	51

2.3	Adaptive lineare Neuronen und die Konvergenz des Lernens . . . . .	56
2.3.1	Straffunktionen mit dem Gradientenabstiegsverfahren minimieren . . . . .	57
2.3.2	Implementierung eines adaptiven linearen Neurons in Python . . . . .	59
2.3.3	Verbesserung des Gradientenabstiegsverfahrens durch Merkmalstandardisierung . . . . .	64
2.3.4	Großmaßstäbliches Machine Learning und stochastisches Gradientenabstiegsverfahren . . . . .	66
2.4	Zusammenfassung . . . . .	71
3	<b>Machine-Learning-Klassifizierer mit scikit-learn verwenden . . . . .</b>	73
3.1	Auswahl eines Klassifizierungsalgorithmus . . . . .	73
3.2	Erste Schritte mit scikit-learn: Trainieren eines Perzeptrons . . . . .	74
3.3	Klassenwahrscheinlichkeiten durch logistische Regression modellieren . . . . .	80
3.3.1	Logistische Regression und bedingte Wahrscheinlichkeiten	81
3.3.2	Gewichtungen der logistischen Straffunktion ermitteln . . . . .	84
3.3.3	Konvertieren einer Adaline-Implementierung in einen Algorithmus für eine logistische Regression . . . . .	87
3.3.4	Trainieren eines logistischen Regressionsmodells mit scikit-learn . . . . .	91
3.3.5	Überanpassung durch Regularisierung verhindern . . . . .	93
3.4	Maximum-Margin-Klassifizierung mit Support Vector Machines . . . . .	96
3.4.1	Maximierung des Randbereichs . . . . .	97
3.4.2	Handhabung des nicht linear trennbaren Falls mit Schlupfvariablen . . . . .	98
3.4.3	Alternative Implementierungen in scikit-learn . . . . .	100
3.5	Nichtlineare Aufgaben mit einer Kernel-SVM lösen . . . . .	101
3.5.1	Kernel-Methoden für linear nicht trennbare Daten . . . . .	101
3.5.2	Mit dem Kernel-Trick Hyperebenen in höherdimensionalen Räumen finden . . . . .	103
3.6	Lernen mit Entscheidungsbäumen . . . . .	107
3.6.1	Maximierung des Informationsgewinns: Daten ausreizen . . . . .	108
3.6.2	Konstruktion eines Entscheidungsbaums . . . . .	112
3.6.3	Mehrere Entscheidungsbäume zu einem Random Forest kombinieren . . . . .	116
3.7	k-Nearest-Neighbor: Ein Lazy-Learning-Algorithmus . . . . .	119
3.8	Zusammenfassung . . . . .	123

<b>4</b>	<b>Gut geeignete Trainingsdatenmengen: Datenvorverarbeitung . . . . .</b>	125
4.1	Umgang mit fehlenden Daten . . . . .	125
4.1.1	Fehlende Werte in Tabellendaten . . . . .	126
4.1.2	Exemplare oder Merkmale mit fehlenden Daten entfernen . . . . .	127
4.1.3	Fehlende Werte ergänzen . . . . .	128
4.1.4	Die Schätzer-API von scikit-learn . . . . .	129
4.2	Handhabung kategorialer Daten . . . . .	130
4.2.1	Nominale und ordinale Merkmale . . . . .	130
4.2.2	Erstellen einer Beispieldatenmenge . . . . .	130
4.2.3	Zuweisung von ordinalen Merkmalen . . . . .	131
4.2.4	Codierung der Klassenbezeichnungen . . . . .	132
4.2.5	One-hot-Codierung der nominalen Merkmale . . . . .	133
4.3	Aufteilung einer Datensammlung in Trainings- und Testdaten . . . . .	136
4.4	Anpassung der Merkmale . . . . .	138
4.5	Auswahl aussagekräftiger Merkmale . . . . .	140
4.5.1	L1- und L2-Regularisierung als Straffunktionen . . . . .	141
4.5.2	Geometrische Interpretation der L2-Regularisierung . . . . .	141
4.5.3	Dünnbesetzte Lösungen mit L1-Regularisierung . . . . .	143
4.5.4	Algorithmen zur sequenziellen Auswahl von Merkmälern . . . . .	147
4.6	Beurteilung der Bedeutung von Merkmalen mit Random Forests . . . . .	154
4.7	Zusammenfassung . . . . .	156
<b>5</b>	<b>Datenkomprimierung durch Dimensionsreduktion . . . . .</b>	159
5.1	Unüberwachte Dimensionsreduktion durch Hauptkomponentenanalyse . . . . .	159
5.1.1	Schritte bei der Hauptkomponentenanalyse . . . . .	160
5.1.2	Schrittweise Extraktion der Hauptkomponenten . . . . .	161
5.1.3	Totale Varianz und Varianzaufklärung . . . . .	164
5.1.4	Merkmalstransformation . . . . .	165
5.1.5	Hauptkomponentenanalyse mit scikit-learn . . . . .	168
5.2	Überwachte Datenkomprimierung durch lineare Diskriminanzanalyse . . . . .	171
5.2.1	Hauptkomponentenanalyse vs. lineare Diskriminanzanalyse . . . . .	172
5.2.2	Die interne Funktionsweise der linearen Diskriminanzanalyse . . . . .	173

5.2.3	Berechnung der Streumatrizen . . . . .	174
5.2.4	Auswahl linearer Diskriminanten für den neuen Merkmalsunerraum . . . . .	176
5.2.5	Projektion in den neuen Merkmalsraum . . . . .	179
5.2.6	LDA mit scikit-learn . . . . .	180
5.3	Kernel-Hauptkomponentenanalyse für nichtlineare Zuordnungen verwenden . . . . .	181
5.3.1	Kernel-Funktionen und der Kernel-Trick . . . . .	182
5.3.2	Implementierung einer Kernel-Hauptkomponentenanalyse in Python . . . . .	186
5.3.3	Projizieren neuer Datenpunkte . . . . .	193
5.3.4	Kernel-Hauptkomponentenanalyse mit scikit-learn . . . . .	197
5.4	Zusammenfassung . . . . .	198
<b>6</b>	<b>Best Practices zur Modellbewertung und Hyperparameter-Abstimmung . . . . .</b>	<b>201</b>
6.1	Arbeitsabläufe mit Pipelines optimieren . . . . .	201
6.1.1	Die Wisconsin-Brustkrebs-Datensammlung . . . . .	201
6.1.2	Transformer und Schätzer in einer Pipeline kombinieren . . . . .	203
6.2	Beurteilung des Modells durch k-fache Kreuzvalidierung . . . . .	205
6.2.1	2-fache Kreuzvalidierung . . . . .	205
6.2.2	k-fache Kreuzvalidierung . . . . .	206
6.3	Algorithmen mit Lern- und Validierungskurven debuggen . . . . .	211
6.3.1	Probleme mit Bias und Varianz anhand von Lernkurven erkennen . . . . .	211
6.3.2	Überanpassung und Unteranpassung anhand von Validierungskurven erkennen . . . . .	214
6.4	Feinabstimmung eines Lernmodells durch Rastersuche . . . . .	216
6.4.1	Hyperparameter-Abstimmung durch Rastersuche . . . . .	216
6.4.2	Algorithmenauswahl durch verschachtelte Kreuzvalidierung . . . . .	218
6.5	Verschiedene Kriterien zur Leistungsbewertung . . . . .	220
6.5.1	Interpretation einer Wahrheitsmatrix . . . . .	220
6.5.2	Optimierung der Genauigkeit und der Trefferquote eines Klassifizierungsmodells . . . . .	222
6.5.3	Receiver-Operating-Characteristic-Diagramme . . . . .	224
6.5.4	Bewertungskriterien für Mehrfachklassifizierungen . . . . .	227

6.6	Handhabung unausgewogener Klassenverteilung .....	228
6.7	Zusammenfassung .....	231
7	<b>Kombination verschiedener Modelle für das Ensemble Learning . . . . .</b>	233
7.1	Ensemble Learning .....	233
7.2	Klassifizierer durch Mehrheitsentscheidung kombinieren .....	237
7.2.1	Implementierung eines einfachen Mehrheitsentscheidungs-Klassifizierers.....	237
7.2.2	Vorhersagen nach dem Prinzip der Mehrheitsentscheidung treffen.....	244
7.3	Bewertung und Abstimmung des Klassifizierer-Ensembles .....	247
7.4	Bagging: Klassifizierer-Ensembles anhand von Bootstrap-Stichproben entwickeln .....	253
7.4.1	Bagging kurz zusammengefasst .....	254
7.4.2	Klassifizierung der Wein-Datensammlung durch Bagging .....	255
7.5	Schwache Klassifizierer durch adaptives Boosting verbessern .....	258
7.5.1	Funktionsweise des Boostings .....	259
7.5.2	AdaBoost mit scikit-learn anwenden .....	263
7.6	Zusammenfassung .....	266
8	<b>Machine Learning zur Analyse von Stimmungslagen nutzen . . . . .</b>	267
8.1	Die IMDb-Filmdatenbank. ....	267
8.1.1	Herunterladen der Datensammlung .....	268
8.1.2	Vorverarbeiten der Filmbewertungsdaten .....	268
8.2	Das Bag-of-words-Modell .....	270
8.2.1	Wörter in Merkmalsvektoren umwandeln .....	270
8.2.2	Beurteilung der Wortrelevanz durch das Tf-idf-Maß .....	272
8.2.3	Textdaten bereinigen .....	275
8.2.4	Dokumente in Token zerlegen .....	277
8.3	Ein logistisches Regressionsmodell für die Dokument-klassifizierung trainieren .....	279
8.4	Verarbeitung großer Datenmengen: Online-Algorithmen und Out-of-Core Learning .....	282
8.5	Topic Modeling mit latenter Dirichlet-Allokation.....	285
8.5.1	Aufteilung von Texten mit der LDA .....	286
8.5.2	LDA mit scikit-learn.....	286
8.6	Zusammenfassung .....	290

<b>9</b>	<b>Einbettung eines Machine-Learning-Modells in eine Webanwendung</b> .....	291
9.1	Serialisierung angepasster Schätzer mit scikit-learn. ....	291
9.2	Einrichtung einer SQLite-Datenbank zum Speichern von Daten ...	295
9.3	Entwicklung einer Webanwendung mit Flask. ....	297
9.3.1	Die erste Webanwendung mit Flask .....	298
9.3.2	Formularvalidierung und -ausgabe .....	300
9.4	Der Filmbewertungsklassifizierer als Webanwendung .....	304
9.4.1	Dateien und Ordner – die Verzeichnisstruktur .....	306
9.4.2	Implementierung der Hauptanwendung app.py .....	306
9.4.3	Einrichtung des Bewertungsformulars.....	309
9.4.4	Eine Vorlage für die Ergebnisseite erstellen.....	310
9.5	Einrichtung der Webanwendung auf einem öffentlich zugänglichen Webserver .....	312
9.5.1	Erstellen eines Benutzerkontos bei PythonAnywhere .....	312
9.5.2	Hochladen der Filmbewertungsanwendung .....	313
9.5.3	Updaten des Filmbewertungsklassifizierers.....	314
9.6	Zusammenfassung .....	316
<b>10</b>	<b>Vorhersage stetiger Zielvariablen durch Regressionsanalyse</b> .....	317
10.1	Lineare Regression. ....	317
10.1.1	Ein einfaches lineares Regressionsmodell .....	318
10.1.2	Multiple lineare Regression .....	319
10.2	Die Lebensbedingungen-Datensammlung .....	320
10.2.1	Einlesen der Datenmenge in einen DataFrame .....	320
10.2.2	Visualisierung der wichtigen Eigenschaften einer Datenmenge .....	321
10.2.3	Zusammenhänge anhand der Korrelationsmatrix erkennen .....	323
10.3	Implementierung eines linearen Regressionsmodells mit der Methode der kleinsten Quadrate .....	326
10.3.1	Berechnung der Regressionsparameter mit dem Gradientenabstiegsverfahren.....	326
10.3.2	Abschätzung der Koeffizienten eines Regressionsmodells mit scikit-learn .....	330
10.4	Anpassung eines robusten Regressionsmodells mit dem RANSAC-Algorithmus .....	332
10.5	Bewertung der Leistung linearer Regressionsmodelle .....	335
10.6	Regularisierungsverfahren für die Regression einsetzen.....	338

10.7	Polynomiale Regression: Umwandeln einer linearen Regression in eine Kurve . . . . .	340
10.7.1	Hinzufügen polynomialer Terme mit scikit-learn . . . . .	341
10.7.2	Modellierung nichtlinearer Zusammenhänge in der Lebensbedingungen-Datensammlung . . . . .	342
10.8	Handhabung nichtlinearer Beziehungen mit Random Forests . . . . .	346
10.8.1	Entscheidungsbaum-Regression . . . . .	346
10.8.2	Random-Forest-Regression . . . . .	348
10.9	Zusammenfassung . . . . .	351
11	<b>Verwendung nicht gekennzeichneter Daten: Clusteranalyse . . . . .</b>	353
11.1	Gruppierung von Objekten nach Ähnlichkeit mit dem k-Means-Algorithmus . . . . .	353
11.1.1	K-Means-Clustering mit scikit-learn . . . . .	354
11.1.2	Der k-Means++-Algorithmus . . . . .	358
11.1.3	»Harte« und »weiche« Clustering-Algorithmen . . . . .	359
11.1.4	Die optimale Anzahl der Cluster mit dem Ellenbogenkriterium ermitteln . . . . .	362
11.1.5	Quantifizierung der Clustering-Güte mit Silhouettendiagrammen . . . . .	363
11.2	Cluster als hierarchischen Baum organisieren . . . . .	368
11.2.1	Gruppierung von Clustern . . . . .	368
11.2.2	Hierarchisches Clustering einer Distanzmatrix . . . . .	370
11.2.3	Dendrogramme und Heatmaps verknüpfen . . . . .	373
11.2.4	Agglomeratives Clustering mit scikit-learn . . . . .	375
11.3	Bereiche hoher Dichte mit DBSCAN ermitteln . . . . .	376
11.4	Zusammenfassung . . . . .	382
12	<b>Implementierung eines künstlichen neuronalen Netzes . . . . .</b>	383
12.1	Modellierung komplexer Funktionen mit künstlichen neuronalen Netzen . . . . .	383
12.1.1	Einschichtige neuronale Netze . . . . .	385
12.1.2	Mehrschichtige neuronale Netzarchitektur . . . . .	387
12.1.3	Aktivierung eines neuronalen Netzes durch Vorwärtspropagation . . . . .	390
12.2	Klassifizierung handgeschriebener Ziffern . . . . .	392
12.2.1	Die MNIST-Datensammlung . . . . .	393
12.2.2	Implementierung eines mehrschichtigen Perzeptrons . . . . .	399
12.3	Trainieren eines künstlichen neuronalen Netzes . . . . .	410
12.3.1	Berechnung der logistischen Straffunktion . . . . .	410

12.3.2	Ein Gespür für die Backpropagation entwickeln . . . . .	413
12.3.3	Trainieren neuronaler Netze durch Backpropagation . . . . .	415
12.4	Konvergenz in neuronalen Netzen. . . . .	418
12.5	Abschließende Bemerkungen zur Implementierung neuronaler Netze . . . . .	420
12.6	Zusammenfassung . . . . .	420
<b>13</b>	<b>Parallelisierung des Trainings neuronaler Netze mit TensorFlow . . . . .</b>	<b>423</b>
13.1	TensorFlow und Trainingsleistung . . . . .	423
13.1.1	Was genau ist TensorFlow? . . . . .	425
13.1.2	TensorFlow erlernen . . . . .	425
13.1.3	Erste Schritte mit TensorFlow . . . . .	426
13.1.4	Mit Array-Strukturen arbeiten . . . . .	428
13.1.5	Entwicklung eines einfachen Modells mit TensorFlows Low-level-API . . . . .	430
13.2	Training neuronaler Netze mit TensorFlows High-level-APIs . . . . .	434
13.2.1	Entwicklung mehrschichtiger neuronaler Netze mit TensorFlows Layers-API . . . . .	435
13.2.2	Entwicklung eines mehrschichtigen neuronalen Netzes mit Keras . . . . .	439
13.3	Auswahl der Aktivierungsfunktionen mehrschichtiger neuronaler Netze . . . . .	444
13.3.1	Die logistische Funktion kurz zusammengefasst . . . . .	445
13.3.2	Wahrscheinlichkeiten bei der Mehrfachklassifizierung mit der softmax-Funktion abschätzen . . . . .	447
13.3.3	Verbreiterung des Ausgabespektrums mittels Tangens hyperbolicus . . . . .	448
13.3.4	Aktivierung durch rektifizierte Lineareinheiten . . . . .	449
13.4	Zusammenfassung . . . . .	451
<b>14</b>	<b>Die Funktionsweise von TensorFlow im Detail . . . . .</b>	<b>453</b>
14.1	Grundlegende Merkmale von TensorFlow . . . . .	453
14.2	TensorFlow-Tensoren und deren Rang . . . . .	454
14.2.1	Rang und Form eines Tensors ermitteln . . . . .	455
14.3	TensorFlow-Berechnungsgraphen . . . . .	456
14.4	Platzhalter in TensorFlow . . . . .	458
14.4.1	Platzhalter definieren . . . . .	459
14.4.2	Platzhaltern Daten zuführen . . . . .	459

14.4.3	Platzhalter für Datenarrays mit variierenden Stapelgrößen definieren .....	460
14.5	Variablen in TensorFlow.....	461
14.5.1	Variablen definieren.....	461
14.5.2	Variablen initialisieren.....	464
14.5.3	Geltungsbereich von Variablen.....	465
14.5.4	Wiederverwendung von Variablen .....	466
14.6	Erstellen eines Regressionsmodells .....	469
14.7	Ausführung von Objekten in einem TensorFlow-Graphen unter Verwendung ihres Namens .....	473
14.8	Speichern und wiederherstellen eines Modells in TensorFlow .....	474
14.9	Tensoren als mehrdimensionale Datenarrays transformieren .....	477
14.10	Mechanismen der Flusskontrolle beim Erstellen von Graphen verwenden .....	481
14.11	Graphen mit TensorBoard visualisieren .....	484
14.11.1	Erweitern Sie Ihre TensorBoard-Kenntnisse .....	487
14.12	Zusammenfassung .....	488
15	<b>Bildklassifizierung mit tiefen konvolutionalen neuronalen Netzen .....</b>	489
15.1	Bausteine konvolutionaler neuronaler Netze .....	489
15.1.1	CNNs und Merkmalshierarchie .....	490
15.1.2	Diskrete Faltungen.....	491
15.1.3	Subsampling.....	500
15.2	Implementierung eines CNNs .....	501
15.2.1	Verwendung mehrerer Eingabe- oder Farbkanäle .....	501
15.2.2	Regularisierung eines neuronalen Netzes mit Dropout .....	504
15.3	Implementierung eines tiefen konvolutionalen neuronalen Netzes mit TensorFlow .....	507
15.3.1	Die mehrschichtige CNN-Architektur .....	507
15.3.2	Einlesen und Vorverarbeiten der Daten .....	508
15.3.3	Implementierung eines CNNs mit TensorFlows Low-level-API .....	509
15.3.4	Implementierung eines CNNs mit TensorFlows Layers-API.....	521
15.4	Zusammenfassung .....	527

<b>16</b>	<b>Modellierung sequenzieller Daten durch rekurrente neuronale Netze . . . . .</b>	529
16.1	Sequenzielle Daten . . . . .	529
16.1.1	Modellierung sequenzieller Daten: Die Reihenfolge ist von Bedeutung . . . . .	530
16.1.2	Repräsentierung von Sequenzen . . . . .	530
16.1.3	Verschiedene Kategorien der Sequenzmodellierung . . . . .	531
16.2	Sequenzmodellierung mit RNNs . . . . .	532
16.2.1	Struktur und Ablauf eines RNNs . . . . .	532
16.2.2	Aktivierungen eines RNNs berechnen . . . . .	534
16.2.3	Probleme bei der Erkennung weitreichender Interaktionen . . . . .	537
16.2.4	LSTM-Einheiten . . . . .	538
16.3	Implementierung eines mehrschichtigen RNNs zur Sequenzmodellierung mit TensorFlow . . . . .	540
16.4	Projekt 1: Analyse der Stimmungslage in der IMDb-Filmbewertungsdatenbank mit mehrschichtigen RNNs . . . . .	541
16.4.1	Datenaufbereitung . . . . .	541
16.4.2	Einbettung . . . . .	545
16.4.3	Erstellen eines RNN-Modells . . . . .	547
16.4.4	Der Konstruktor der SentimentRNN-Klasse . . . . .	548
16.4.5	Die build-Methode . . . . .	548
16.4.6	Die train-Methode . . . . .	552
16.4.7	Die predict-Methode . . . . .	553
16.4.8	Instanziierung der SentimentRNN-Klasse . . . . .	554
16.4.9	Training und Optimierung des RNN-Modells zur Stimmungsanalyse . . . . .	554
16.5	Projekt 2: Implementierung eines RNNs zur Sprachmodellierung durch Zeichen mit TensorFlow . . . . .	555
16.5.1	Datenaufbereitung . . . . .	556
16.5.2	Erstellen eines RNNs zur Sprachmodellierung durch Zeichen . . . . .	560
16.5.3	Der Konstruktor . . . . .	560
16.5.4	Die build-Methode . . . . .	561
16.5.5	Die train-Methode . . . . .	564
16.5.6	Die sample-Methode . . . . .	565
16.5.7	Erstellen und Trainieren des CharRNN-Modells . . . . .	566
16.5.8	Das CharRNN-Modell im Sampling-Modus . . . . .	567
16.6	Zusammenfassung und Schlusswort . . . . .	568
	<b>Stichwortverzeichnis . . . . .</b>	571

# Über die Autoren

**Sebastian Raschka**, Autor des viel verkauften Buches *Python Machine Learning*, verfügt über jahrelange Erfahrung in der Python-Programmierung und leitete mehrere Seminare über praktische Data-Science-Anwendungen, Machine Learning und Deep Learning, unter anderem eine Einführung in Machine Learning auf der SciPy-Konferenz, der maßgeblichen Veranstaltung für wissenschaftliche Anwendungen in Python.

Seine Forschungsprojekte befassen sich zwar vornehmlich mit Fragen der Berechnung biologischer Phänomene, aber es bereitet ihm ebenfalls Freude, über Themen wie Data Science, Machine Learning und Python-Programmierung im Allgemeinen nachzudenken und zu schreiben, um es auch Leuten ohne Kenntnisse in Bezug auf maschinelle Lernverfahren zu ermöglichen, datengesteuerte Lösungen zu entwickeln.

Seine Arbeiten und Beiträge wurden kürzlich mit dem Preis für herausragende Leistungen von Hochschulabsolventen 2016 sowie dem ACM Computing Reviews Best of 2016/2017 ausgezeichnet. In seiner Freizeit leistet Sebastian aktiv Beiträge zu Open-Source-Projekten und die von ihm implementierten Verfahren werden erfolgreich in Mustererkennungswettbewerben wie z.B. Kaggle eingesetzt.

---

Ich möchte diese Gelegenheit nutzen, der großartigen Python-Community und den Entwicklern der Open-Source-Pakete meinen Dank auszusprechen, die mir dabei geholfen haben, die perfekte Umgebung für wissenschaftliche Forschung und Data Science einzurichten. Außerdem möchte ich meinen Eltern danken, die mich bei all meinen beruflichen Zielen, die ich so leidenschaftlich verfolgt habe, stets ermutigt und unterstützt haben.

Mein besonderer Dank gilt den Hauptentwicklern von scikit-learn. Als jemand, der selbst aktiv an diesem Projekt beteiligt war, hatte ich das Vergnügen, mit tollen Leuten zusammenarbeiten zu dürfen, die sich nicht nur mit Machine Learning auskennen, sondern auch hervorragende Programmierer sind. Und zum Schluss möchte ich mich bei Elie Kawerk bedanken, der anbot, das Buch in Augenschein zu nehmen und wertvolles Feedback zu den neuen Kapiteln lieferte.

---

**Vahid Mirjalili** erlangte seinen Doktortitel als Maschinenbauingenieur mit einer Arbeit über neue Verfahren für Computersimulationen molekularer Strukturen. Derzeit erforscht er Anwendungen des Machine Learnings in verschiedenen Computer-Vision-Projekten (»maschinelles Sehen«) am Fachbereich für Informatik und Ingenieurwesen an der Michigan State University.

Vahid hat sich für Python als bevorzugte Programmiersprache entschieden und während seiner akademischen Laufbahn enorme Erfahrung in der Python-Programmierung gesammelt. Er hat an der Michigan State University Programmierkurse in Python für angehende Ingenieure geleitet, was es ihm ermöglichte, den Studenten ein besseres Verständnis verschiedener Datenstrukturen und die effiziente Entwicklung von Python-Code zu vermitteln.

Neben der Erforschung von Deep Learning und Anwendungen des maschinellen Sehens ist er besonders daran interessiert, Lernverfahren zu entwickeln, die biometrische Daten wie Bilder von Gesichtern besser schützen, sodass ein Benutzer nicht mehr Informationen preisgibt als beabsichtigt. Darüber hinaus arbeitet er mit einem Team von Ingenieuren zusammen, das selbstfahrende Autos entwickelt. Er entwirft dabei Modelle neuronaler Netzwerke, die anhand multispektraler Bilddaten Fußgänger erkennen.

---

Ich möchte meinem Doktorvater Dr. Arun Ross dafür danken, dass er mir die Möglichkeit bietet, in seinem Forschungslabor an neuartigen Aufgaben zu arbeiten. Außerdem möchte ich Dr. Vishnu Boddeti dafür danken, dass er mein Interesse an Deep Learning entfacht und mir die grundlegenden Konzepte vor Augen geführt hat.

---

# Über die Korrektoren

**Jared Huffmann** ist Unternehmer, Spieler, Geschichtenerzähler, Machine-Learning-Fan und begeisterter Anhänger von Datenbanken. Die letzten zehn Jahre hat er mit der Entwicklung von Software und der Analyse von Daten verbracht. Seine früheren Tätigkeiten umfassen ein breites Spektrum, unter anderem Sicherheitsaspekte von Netzwerken, Finanzsysteme, systematische Unternehmensanalyse sowie Webdienste, Entwickler-Tools und Geschäftsstrategie. Erst kürzlich hat er das Data-Science-Team von Minecraft gegründet, das sich auf Big Data und Machine Learning konzentriert. Wenn er nicht arbeitet, ist er typischerweise beim Spielen oder genießt den wunderschönen pazifischen Nordwesten mit Freunden und der Familie.

---

Ich danke dem amerikanischen Originalverlag Packt für die Möglichkeit, an einem so tollen Buch mitwirken zu können, meiner Frau für die unermüdliche Ermutigung und meiner Tochter dafür, dass sie die meisten Abende durchgeschlafen hat, während ich mit der Durchsicht und dem Debuggen des Codes beschäftigt war.

---

**Huai-En, Sun (Ryan Sun)** verfügt über einen akademischen Grad der National Chiao Tung University in Statistik. Er ist derzeit als Data Scientist tätig und analysiert die Produktionsverfahren bei PEGATRON. Seine Forschungsschwerpunkte sind Machine Learning und Deep Learning.



# Einleitung

Aus den Nachrichten und den sozialen Medien ist Ihnen vermutlich bekannt, dass das Machine Learning zu einer der spannendsten Technologien der heutigen Zeit geworden ist. Große Unternehmen wie Google, Facebook, Apple, Amazon, IBM und viele andere investieren aus gutem Grund kräftig in die Erforschung des Machine Learnings und dessen Anwendung. Auch wenn man manchmal den Eindruck bekommt, dass »Machine Learning« als leeres Schlagwort gebraucht wird, handelt es sich doch zweifellos nicht um eine Modeerscheinung. Dieses spannende Fachgebiet eröffnet viele neue Möglichkeiten und ist aus dem Alltag schon nicht mehr wegzudenken. Denken Sie an die virtuellen Assistenten von Smartphones, Produktempfehlungen für Kunden in Onlineshops, das Verhindern von Kreditkartenbetrug, Spamfilter in E-Mail-Programmen, die Erkennung und Diagnose von Krankheitssymptomen – die Liste ließe sich beliebig lang fortsetzen.

Wenn Sie zu einem Praktiker des Machine Learnings und einem besseren Problemlöser werden möchten oder vielleicht sogar eine Laufbahn in der Erforschung des Machine Learnings anstreben, dann ist dies das richtige Buch für Sie. Für einen Neuling können die dem Machine Learning zugrunde liegenden theoretischen Konzepte zunächst einmal erdrückend wirken. In den vergangenen Jahren sind aber viele praxisorientierte Bücher mit leistungsfähigen Lernalgorithmen erschienen, die Ihnen den Start erleichtern.

Die Verwendung praxisorientierter Codebeispiele dient einem wichtigen Zweck: Konkrete Beispiele verdeutlichen die allgemeinen Konzepte, indem das Erlernte unmittelbar in die Tat umgesetzt wird. Allerdings darf man dabei nicht vergessen, dass mit großer Macht auch immer große Verantwortung einhergeht! Neben der unmittelbaren Erfahrung, Machine Learning mithilfe der Programmiersprache Python und auf Python beruhenden Lernbibliotheken in die Tat umzusetzen, stellt das Buch auch die den Machine-Learning-Algorithmen zugrunde liegenden mathematischen Konzepte vor, die für den erfolgreichen Einsatz von Machine Learning unverzichtbar sind. Das Buch ist also kein rein praktisch orientiertes Werk, sondern ein Buch, das die erforderlichen Details der Konzepte des Machine Learnings erörtert, die Funktionsweise von Lernalgorithmen und ihre Verwendung verständlich, aber dennoch informativ erklärt und – was noch wichtiger ist – das zeigt, wie man die häufigsten Fehler vermeidet.

Wenn Sie bei Google Scholar den Suchbegriff *machine learning* eingeben, erhalten Sie als Resultat eine riesige Zahl (ca. 1.800.000) von Treffern. Nun können wir in

diesem Buch natürlich nicht sämtliche Einzelheiten der in den letzten 60 Jahren entwickelten Algorithmen und Anwendungen erörtern. Wir werden uns jedoch auf eine spannende Tour begeben, die alle wichtigen Themen und Konzepte umfasst, damit Sie eine gründliche Einführung erhalten. Sollte Ihr Wissensdurst auch nach der Lektüre noch nicht gestillt sein, steht Ihnen eine Vielzahl weiterer hilfreicher Ressourcen zur Verfügung, die Sie nutzen können, um die entscheidenden Fortschritte auf diesem Fachgebiet zu verfolgen.

Falls Sie sich schon ausführlich mit der Theorie des Machine Learnings beschäftigt haben, zeigt Ihnen dieses Buch, wie Sie Ihre Kenntnisse in die Praxis umsetzen können. Wenn Sie bereits entsprechende Techniken eingesetzt haben, aber deren Funktionsweise besser verstehen möchten, kommen Sie hier ebenfalls auf Ihre Kosten. Und wenn Ihnen das Thema Machine Learning noch völlig neu ist, haben Sie umso mehr Grund, sich zu freuen, denn ich kann Ihnen versprechen, dass dieses Verfahren Ihre Denkweise über Ihre in Zukunft zu lösenden Aufgaben verändern wird – und ich möchte Ihnen zeigen, wie Sie Problemstellungen in Angriff nehmen können, indem Sie die den Daten innenwohnende Kraft freisetzen.

Bevor wir uns eingehender mit dem Machine Learning befassen, soll aber zunächst noch Ihre vermutlich vordringlichste Frage beantwortet werden: Warum Python? Die Antwort ist einfach: Es ist leistungsfähig und doch leicht zu erlernen. Python ist zur beliebtesten Programmiersprache im Bereich Data Science geworden, weil man sich die lästigen Teile bei der Programmierung erspart und eine Umgebung bereitsteht, in der sich Ideen und Konzepte sofort umsetzen lassen.

Wir, die Autoren, können aus eigener Erfahrung sagen, dass wir durch die Beschäftigung mit dem Machine Learning zu besseren Wissenschaftlern, Denkern und Problemlösern geworden sind. In diesem Buch möchten wir unsere diesbezüglichen Erkenntnisse mit Ihnen teilen. Wissen wird durch Lernen erworben, was wiederum einen gewissen Eifer erfordert, und erst Übung macht den sprichwörtlichen Meister. Der vor Ihnen liegende Weg ist manchmal nicht ganz einfach, und einige der Themenbereiche sind deutlich schwieriger als andere, aber wir hoffen dennoch, dass Sie die Gelegenheit nutzen und sich auf den Lohn der Mühe konzentrieren. Im weiteren Verlauf des Buches werden Sie Ihrem Repertoire eine ganze Reihe leistungsfähiger Techniken hinzufügen können, die dabei helfen, auch die schwierigsten Aufgaben auf datengesteuerte Weise zu bewältigen.

## Zum Inhalt des Buches

*Kapitel 1, Wie Computer aus Daten lernen können*, führt Sie in die wichtigsten Teilbereiche des Machine Learnings ein, mit denen sich verschiedene Probleme in Angriff nehmen lassen. Darüber hinaus werden die grundlegenden Schritte beim Entwurf eines typischen Machine-Learning-Modells erörtert, auf die wir in den nachfolgenden Kapiteln zurückgreifen.

*Kapitel 2, Lernalgorithmen für die Klassifizierung trainieren*, geht zurück zu den Anfängen des Machine Learnings und stellt binäre Perzepron-Klassifizierer und adaptive lineare Neuronen vor. Dieses Kapitel ist eine behutsame Einführung in die Grundlagen der Klassifizierung von Mustern und konzentriert sich auf das Zusammenspiel von Optimierungsalgorithmen und Machine Learning.

*Kapitel 3, Machine-Learning-Klassifizierer mit scikit-learn verwenden*, beschreibt die wichtigsten Klassifizierungsalgorithmen des Machine Learnings und stellt praktische Beispiele vor. Dabei kommt eine der beliebtesten und verständlichsten Open-Source-Bibliotheken für Machine Learning zum Einsatz: scikit-learn.

*Kapitel 4, Gut geeignete Trainingsdatenmengen: Datenvorverarbeitung*, erläutert die Handhabung der gängigsten Probleme unverarbeiteter Datenmengen, wie z.B. fehlende Daten. Außerdem werden verschiedene Ansätze zur Ermittlung der informativsten Merkmale einer Datenmenge vorgestellt. Des Weiteren erfahren Sie, wie sich Variablen unterschiedlichen Typs als geeignete Eingabe für Lernalgorithmen einsetzen lassen.

*Kapitel 5, Datenkomprimierung durch Dimensionsreduktion*, beschreibt ein wichtiges Verfahren zur Reduzierung der Merkmalsanzahl eines Datenbestands durch Aufteilung in kleinere Mengen unter Beibehaltung eines Großteils der nützlichsten und charakteristischsten Informationen. Hier wird der Standardansatz zur Dimensionsreduktion durch die Analyse der Hauptkomponenten erläutert und mit überwachten und nichtlinearen Transformationsverfahren verglichen.

*Kapitel 6, Bewährte Verfahren zur Modellbewertung und Hyperparameter-Abstimmung*, erörtert die Einschätzung der Aussagekraft von Vorhersagemodellen. Darüber hinaus kommen verschiedene Bewertungskriterien der Modelle sowie Verfahren zur Feinabstimmung der Lernalgorithmen zur Sprache.

*Kapitel 7, Kombination verschiedener Modelle für das Ensemble Learning*, führt Sie in die verschiedenen Konzepte zur effektiven Kombination diverser Lernalgorithmen ein. Sie erfahren, wie Sie Ensembles einrichten, um die Schwächen einzelner Klassifizierer zu überwinden, was genauere und verlässlichere Vorhersagen liefert.

*Kapitel 8, Machine Learning zur Analyse von Stimmungslagen nutzen*, erläutert die grundlegenden Schritte zur Transformierung von Textdaten in eine für Lernalgorithmen sinnvolle Form, um so die Meinung von Menschen anhand der von ihnen verfassten Texte vorherzusagen.

*Kapitel 9, Einbettung eines Machine-Learning-Modells in eine Webanwendung*, führt vor, wie Sie das Lernmodell des vorangehenden Kapitels Schritt für Schritt in eine Webanwendung einbetten können.

*Kapitel 10, Vorhersage stetiger Zielvariablen durch Regressionsanalyse*, erörtert grundlegende Verfahren zur Modellierung linearer Beziehungen zwischen Zielvariablen und Regressanden, um auch stetige Werte vorhersagen zu können. Nach der Vor-

stellung der linearen Modelle kommen auch Polynom-Regression und baumbasierte Ansätze zur Sprache.

*Kapitel 11, Verwendung nicht gekennzeichneter Daten: Clusteranalyse*, konzentriert sich auf einen anderen Teilbereich des Machine Learnings, nämlich auf das unüberwachte Lernen. Wir werden drei unterschiedlichen Familien von Clustering-Algorithmen zugehörige Verfahren anwenden, um Objektgruppen aufzuspüren, die einen gewissen Ähnlichkeitsgrad aufweisen.

*Kapitel 12, Implementierung eines künstlichen neuronalen Netzes*, erweitert das in Kapitel 2 vorgestellte Konzept der Gradient-basierten Optimierung, um leistungsfähige, mehrschichtige neuronale Netze zu erstellen, die auf dem verbreiteten Backpropagation-Algorithmus beruhen.

*Kapitel 13, Parallelisierung des Trainings neuronaler Netze mit TensorFlow*, baut auf den in den vorausgehenden Kapiteln erworbenen Kenntnissen auf, um Ihnen einen praxisorientierten Leitfaden für ein effizienteres Training neuronaler Netze an die Hand zu geben. Der Schwerpunkt dieses Kapitels liegt dabei auf TensorFlow, einer quelloffenen Python-Bibliothek, die die Verwendung mehrerer Kerne moderner Grafikprozessoren ermöglicht.

*Kapitel 14, Die Funktionsweise von TensorFlow im Detail*, behandelt TensorFlow ausführlicher und erläutert die grundlegenden Konzepte von Berechnungsgraphen und Sitzungen. Darüber hinaus kommen Themen wie das Abspeichern und Visualisieren der Graphen neuronaler Netze zur Sprache, was sich im verbleibenden Teil des Buches als sehr nützlich erweisen wird.

*Kapitel 15, Bildklassifizierung mit tiefen konvolutionalen neuronalen Netzen*, stellt neuronale Netzarchitekturen vor, die bei maschinellem Sehen und der Bilderkennung zu einem neuen Standard geworden sind, nämlich konvolutionale neuronale Netze. Dieses Kapitel erörtert die grundlegenden Konzepte konvolutionaler Schichten als Merkmalsextraktoren und zeigt die Anwendung einer konvolutionalen neuronalen Netzarchitektur zur Klassifizierung von Bildern, die eine nahezu perfekte Klassifizierung erzielt.

*Kapitel 16, Modellierung sequenzieller Daten durch rekurrente neuronale Netze*, stellt eine weitere verbreitete neuronale Netzarchitektur für Deep Learning vor, die besonders gut für die Verarbeitung von sequenziellen Daten und Zeitreihen geeignet ist. In diesem Kapitel werden wir verschiedene rekurrente neuronale Netzarchitekturen auf Textdaten anwenden. Als Aufwärmübung betrachten wir zunächst eine Stimmungsanalyse und erzeugen anschließend völlig neue Texte.

## Was Sie benötigen

Zum Ausführen der Codebeispiele ist die Python-Version 3.6.0 oder neuer auf macOS, Linux oder Microsoft Windows erforderlich. Wir werden häufig von für

wissenschaftliche Berechnungen unverzichtbaren Python-Bibliotheken Gebrauch machen, z.B. von SciPy, NumPy, scikit-learn, Matplotlib und pandas.

Im ersten Kapitel finden Sie Hinweise und Tipps zur Einrichtung Ihrer Python-Umgebung und dieser elementaren Bibliotheken. In den verschiedenen Kapiteln werden wir dann der Python-Umgebung weitere Bibliotheken hinzufügen: die NLTK-Bibliothek für die Verarbeitung natürlicher Sprache (Kapitel 8), das Web-Framework Flask (Kapitel 9), die Seaborn-Bibliothek zur Visualisierung statistischer Daten (Kapitel 10) und schließlich TensorFlow, um neuronale Netze effizient mit grafischen Symbolen zu trainieren.

## Für wen ist das Buch gedacht?

Wenn Sie wissen möchten, wie Sie Python einsetzen können, um wichtige Fragen über Ihre Daten zu beantworten, sind Sie hier genau richtig. Ob Sie nun Anfänger sind oder Ihre Kenntnisse auf dem Gebiet der Data Science vertiefen möchten: Dieses Buch ist eine unentbehrliche Informationsquelle und unbedingt lesenswert.

## Konventionen im Buch

In diesem Buch werden verschiedene Textarten verwendet, um zwischen Informationen unterschiedlicher Art zu unterscheiden. Nachstehend finden Sie einige Beispiele und deren Bedeutungen.

Schlüsselwörter, Datenbanktabellen-, Twitter-, Datei-, Ordner-, Datei- und Pfadnamen sowie URLs und Usereingaben werden im Fließtext wie folgt dargestellt:

»Durch die Einstellung `out_file=None` weisen wir die Daten direkt der Variablen `doc_data` zu, ohne sie erst in eine temporäre Datei `tree.dot` auf die Festplatte zu schreiben.«

Codeblöcke sehen so aus:

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> knn = KNeighborsClassifier(n_neighbors=5, p=2,
...                               metric='minkowski')
>>> knn.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std, y_combined,
...                         classifier=knn, test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.show()
```

Usereingaben oder Ausgaben auf der Kommandozeile werden in nicht proportionaler Schrift fett gedruckt:

```
pip3 install graphviz
```

*Neue Ausdrücke* und *wichtige Begriffe* werden kursiv gedruckt. Auf dem Bildschirm auswählbare oder anklickbare Bezeichnungen, wie z.B. Menüpunkte oder Schaltflächen, werden in der Schriftart Kapitälchen gedruckt: »Nach einem Klick auf die Schaltfläche **ABBRECHEN** in der unteren rechten Ecke wird der Vorgang abgebrochen.«

### Vorsicht

Warnungen oder wichtige Hinweise erscheinen in einem Kasten wie diesem.

### Tipp

Und so werden Tipps und Tricks dargestellt.

## Codebeispiele herunterladen

Die Codebeispiele zum Buch finden Sie auf GitHub unter <https://github.com/rasbt/python-machine-learning-book-2nd-edition>.

### Farbige Abbildungen

Alle in diesem Buch verwendeten Diagramme und Screenshots stehen unter [www.mitp.de/733](http://www.mitp.de/733) zusätzlich in einer farbigen Variante zum Download zur Verfügung.

# Wie Computer aus Daten lernen können

Unserer Ansicht nach ist *das Machine Learning (maschinelles Lernen)*, die Anwendung und Wissenschaft von Algorithmen, die den Sinn von Daten erkennen können, das spannendste Forschungsfeld der Informatik! Wir leben in einem Zeitalter, in dem Daten im Überfluss vorhanden sind – und mit den selbstlernenden Algorithmen des Machine Learnings können wir diese Daten in Wissen verwandeln. Dank der vielen in den letzten Jahren entwickelten Open-Source-Bibliotheken ist jetzt der richtige Zeitpunkt gekommen, um sich eingehend mit dem Thema Machine Learning zu befassen und zu erfahren, wie leistungsfähige Algorithmen dafür eingesetzt werden können, Muster in den Daten zu erkennen und Vorhersagen über zukünftige Ereignisse zu treffen.

In diesem Kapitel werden wir die grundlegenden Konzepte und verschiedene Arten des Machine Learnings erörtern. Mit einer Einführung in die relevante Terminologie schaffen wir die Grundlage dafür, Verfahren des Machine Learnings erfolgreich zum Lösen von in der Praxis auftretenden Problemen einzusetzen.

Dieses Kapitel hat folgende Themen zum Inhalt:

- Allgemeine Konzepte des Machine Learnings
- Die drei Arten des Machine Learnings und grundlegende Begriffe
- Die Bausteine des erfolgreichen Designs von Lernsystemen
- Installation von Python und Einrichtung einer für die Analyse von Daten und Machine Learning geeigneten Umgebung

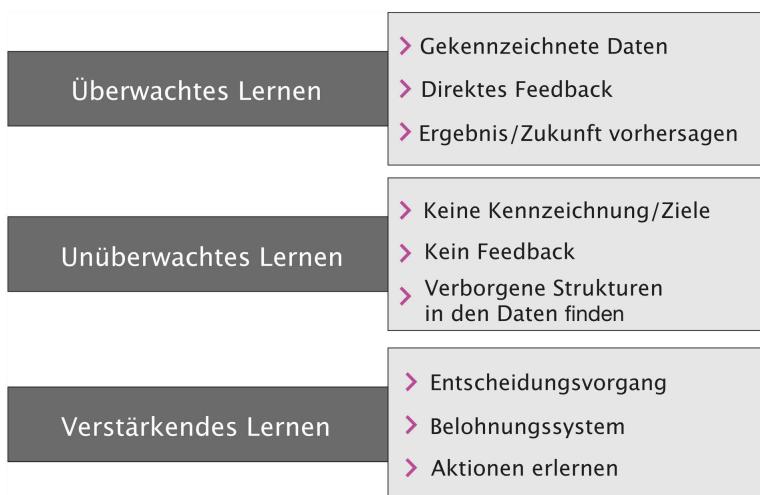
## 1.1 Intelligente Maschinen, die Daten in Wissen verwandeln

In diesem Zeitalter der modernen Technologie steht eine Ressource im Überfluss zur Verfügung: Unmengen von strukturierten und unstrukturierten Daten. In der zweiten Hälfte des 20. Jahrhunderts hat sich das Machine Learning als eine Teildisziplin der *Artificial Intelligence* (künstliche Intelligenz) herausgebildet, bei der es um die Entwicklung selbstlernender Algorithmen geht, die Erkenntnisse aus Daten extrahieren, um bestimmte Vorhersagen treffen zu können. Das Erfordernis menschlichen Eingreifens zur manuellen Ableitung von Regeln und der Entwicklung von Modellen anhand der Analyse großer Datenmengen erübriggt sich damit

mehr und mehr, denn das Machine-Learning-Verfahren bietet eine effiziente Alternative zur Erfassung des in den Daten enthaltenen Wissens – die zudem die auf diesen Daten basierende Entscheidungsfindung sowie die Aussagekraft von Vorhersagemodellen zusehends verbessert. Dieses Verfahren wird nicht nur in der Forschung immer wichtiger, es spielt auch im Alltag eine zunehmend größere Rolle: Dank des Machine Learnings erfreuen wir uns stabiler E-Mail-Spamfilter, praktischer Text- und Spracherkennungssoftware, verlässlicher Suchmaschinen, kaum zu schlagender Schachcomputer und hoffentlich bald auch sicherer selbstfahrender Autos.

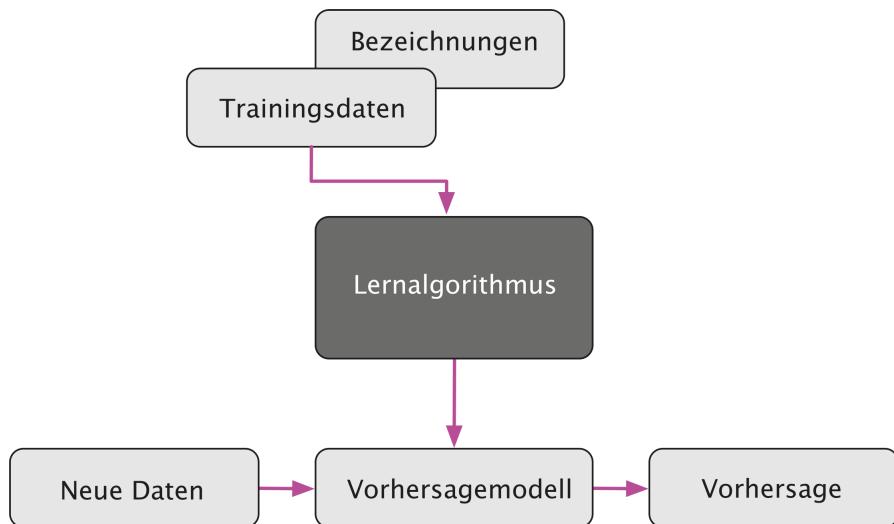
## 1.2 Die drei Arten des Machine Learnings

In diesem Abschnitt werden wir die drei verschiedenen Gattungen des Machine Learnings betrachten: *überwachtes Lernen*, *unüberwachtes Lernen* und *verstärkendes Lernen*. Sie werden erfahren, welche grundlegenden Unterschiede es zwischen diesen drei Varianten gibt und anhand von Beispielen allmählich ein Gespür dafür entwickeln, auf welche praktischen Aufgabenstellungen sie sich anwenden lassen:



### 1.2.1 Mit überwachtem Lernen Vorhersagen treffen

Das Hauptziel des überwachten Lernens ist, ein Modell anhand gekennzeichneter *Trainingsdaten* zu erlernen, um so Voraussagen über unbekannte oder zukünftige Daten treffen zu können. Der Begriff »*überwacht*« bezieht sich hier auf Trainingsdaten, die bereits mit den bekannten erwünschten Ausgabewerten (Bezeichnungen/Labels) gekennzeichnet sind.



Betrachten wir als Beispiel das Filtern von E-Mail-Spam. Wir können einen überwachten Lernalgorithmus mit einer Sammlung von als Spam oder Nicht-Spam gekennzeichneten E-Mails »trainieren«, um dann vorherzusagen, zu welcher dieser Klassen eine neue E-Mail gehört. Eine solche Einteilung in bestimmte Klassen wird als *Klassifizierung* bezeichnet. Eine weitere Unterkategorie des überwachten Lernens ist die *Regression*, bei der die Ausgabewerte im Gegensatz zur Klassifizierung stetig sind.

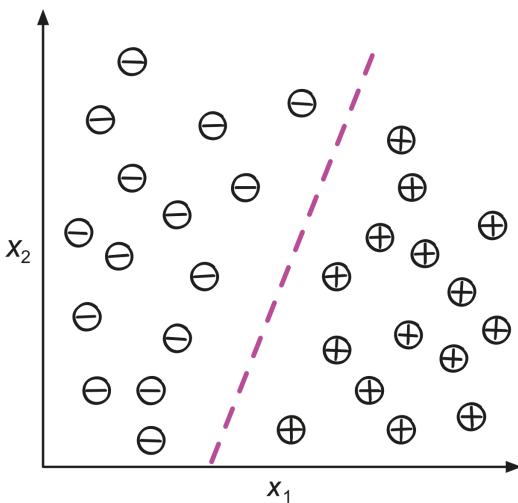
### **Klassifizierung: Vorhersage der Klassenbezeichnungen**

Die Klassifizierung ist eine Unterkategorie des überwachten Lernens, die es zum Ziel hat, anhand vorhergehender Beobachtungen die kategorialen Klassen neuer Instanzen vorherzusagen. Die Bezeichnungen dieser Klassen sind eindeutige, ungeordnete Werte, die als Gruppenzugehörigkeit der Instanzen aufgefasst werden können. Die soeben erwähnte E-Mail-Spamerkennung stellt ein typisches Beispiel für eine *binäre Klassifizierung* dar, denn der Algorithmus erlernt Regeln, um zwischen zwei möglichen Klassen zu unterscheiden: Spam oder Nicht-Spam.

Die Anzahl der Klassenbezeichnungen muss allerdings nicht auf zwei beschränkt sein. Das von einem überwachten Lernalgorithmus erlernte Vorhersagemodell kann einer neuen, noch nicht gekennzeichneten Instanz jede Bezeichnung zuordnen, die in den Trainingsdaten vorkommt. Ein typisches Beispiel für solch eine *Mehrfachklassifizierung* ist die Handschrifterkennung. Hier könnten wir eine Trainingsdatenmenge zusammenstellen, die aus mehreren handgeschriebenen Beispielen aller Buchstaben des Alphabets besteht. Wenn dann ein User über ein Eingabegerät einen neuen Buchstaben angibt, wäre unser Vorhersagemodell in der Lage, diesen mit einer gewissen Zuverlässigkeit zu erkennen. Das System

wäre allerdings nicht imstande, irgendeine der Zahlen von null bis neun zu erkennen, sofern diese nicht ebenfalls Bestandteil der Trainingsdaten waren.

Die folgende Abbildung illustriert das Konzept einer binären Klassifizierung, die mit 30 Beispielen trainiert wird, von denen 15 als *negative Klasse* (Minuszeichen) und weitere 15 als *positive Klasse* (Pluszeichen) gekennzeichnet sind. Die Datenmenge ist in diesem Szenario zweidimensional: Jedem Beispiel sind die beiden Werte  $x_1$  und  $x_2$  zugeordnet. Nun können wir dem überwachten Lernalgorithmus eine Regel beibringen: Die durch eine gestrichelte Linie dargestellte Grenze trennt die beiden Klassen voneinander und ermöglicht es, neue Daten anhand der Werte von  $x_1$  und  $x_2$  einer der beiden Klassen zuzuordnen.



## Regression: Vorhersage stetiger Ergebnisse

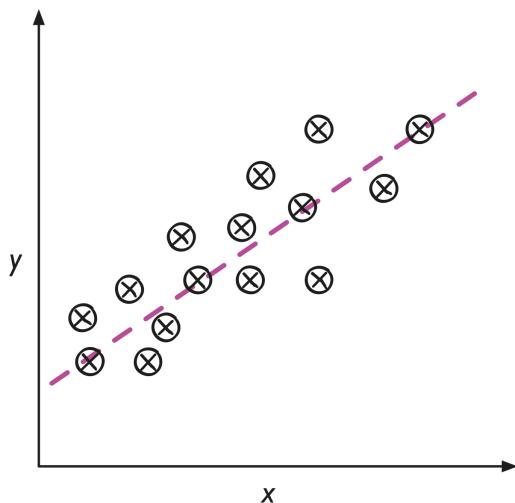
Im vorangegangenen Abschnitt haben wir festgestellt, dass es die Aufgabe einer Klassifizierung ist, Instanzen kategoriale, ungeordnete Klassenbezeichnungen zuzuordnen. Ein zweiter Typ des überwachten Lernens ist die Vorhersage stetiger Ergebnisse, die auch als *Regressionsanalyse* bezeichnet wird. Hierbei sind verschiedene *Regressor-Variablen* (unabhängige oder *erklärende* Variablen) sowie eine stetige Zielvariable (Ergebnis) vorgegeben und wir versuchen, eine Beziehung zwischen diesen Variablen zu finden, um Ergebnisse vorhersagen zu können.

Nehmen wir beispielsweise an, dass wir die von Schülern bei einer Matheprüfung erreichten Punktzahlen prognostizieren möchten. Sofern es einen Zusammenhang zwischen der mit dem Üben für die Prüfung verbrachten Zeit und den erzielten Punktzahlen gibt, könnten wir daraus Trainingsdaten für ein Modell herleiten, das anhand der aufgewendeten Übungszeit die Punktzahlen von Schülern voraussagt, die die Prüfung in Zukunft ebenfalls abzulegen beabsichtigen.

**Tipp**

Der Begriff *Regression* wurde schon 1886 von Francis Galton in einem Artikel mit dem Titel *Regression Towards Mediocrity in Hereditary Stature* (Regression zur Mitte in Bezug auf die ererbte Körpergröße) geprägt. Galton beschrieb darin das Phänomen, dass sich bei der Bevölkerung die mittlere Abweichung von der durchschnittlichen Körpergröße im Laufe der Zeit nicht vergrößert. Er beobachtete, dass die Körpergröße der Eltern nicht an die Kinder vererbt wird, vielmehr nähert sich die Größe der Kinder dem Durchschnittswert an.

Die folgende Abbildung illustriert das Konzept der *linearen Regression*. Bei vorgegebener unabhängiger Variablen  $x$  und abhängiger Variablen  $y$  passen wir eine Gerade so an die Daten an, dass ein Maß für den Abstand der Geraden von den Beispielwerten (üblicherweise der Mittelwert der quadrierten Differenzen) minimal wird. Nun können wir den aus den Daten ermittelten Schnittpunkt mit der  $y$ -Achse sowie die Steigung der Geraden verwenden, um das Ergebnis für neue Werte vorherzusagen.

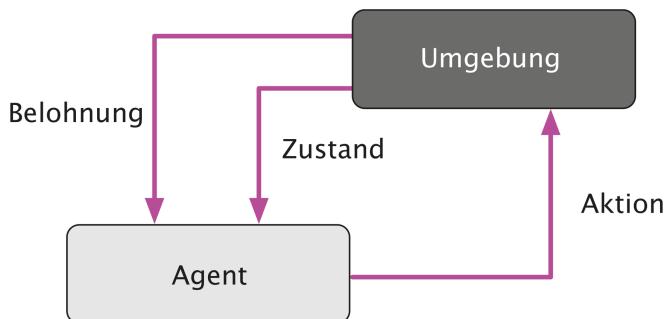


### 1.2.2 Interaktive Aufgaben durch verstärkendes Lernen lösen

Die dritte Variante des Machine Learnings ist das verstärkende Lernen. Hier besteht die Zielsetzung darin, ein System zu entwickeln (den *Agenten*), das seine Leistung durch Interaktionen mit seiner *Umgebung* verbessert. Zu den Informationen über den aktuellen Zustand der Umgebung gehört typischerweise ein sogenanntes *Belohnungssignal*, daher ist das verstärkende Lernen in gewisser Weise mit dem *überwachten Lernen* verwandt. Allerdings handelt es sich bei diesem Feedback

nicht um die korrekte Klassenbezeichnung oder den richtigen Wert, sondern um ein Maß dafür, wie gut die Aktion war, das durch eine *Belohnungsfunktion* beschrieben wird. Der Agent kann über Interaktionen mit seiner Umgebung durch verstärkendes Lernen erkennen, welche Aktionen besonders gut belohnt werden. Das kann durch schlichtes Ausprobieren (Versuch und Irrtum) oder durch bewusste Planung geschehen.

Ein schönes Beispiel für verstärkendes Lernen ist ein Schachcomputer. Hier bewertet der Agent nach einer Reihe von Zügen die Stellung auf dem Schachbrett (die Umgebung), und die Belohnung kann am Ende des Spiels als *Sieg* oder *Niederlage* definiert werden.



Es gibt eine Vielzahl verschiedener Unterarten des verstärkenden Lernens. Im Allgemeinen versucht der Agent beim verstärkenden Lernen jedoch, die Belohnung durch eine Reihe von Interaktionen mit der Umgebung zu maximieren. Jedem Zustand kann eine positive (oder negative) Bewertung zugeordnet werden, und die Belohnung kann dadurch definiert werden, dass ein Gesamtziel erreicht wird, wie z.B. das Gewinnen oder das Verlieren einer Schachpartie. Beim Schachspiel kann etwa das Ergebnis eines jeden Spielzugs als ein anderer Zustand der Umgebung aufgefasst werden.

Um beim Schach zu bleiben: Stellen Sie sich das Erreichen bestimmter Positionen auf dem Schachbrett als positives Ereignis vor – beispielsweise das Schlagen einer gegnerischen Spielfigur oder das bedrohen der Dame. Andere Positionen wiederum werden als negativ erachtet, beispielsweise wenn eine der eigenen Spielfiguren beim nächsten Zug geschlagen werden kann. Nun führt natürlich nicht jeder Zug zum Schlagen einer Spielfigur, daher versucht das verstärkende Lernen eine Reihe von Schritten zu erlernen, indem eine Belohnung aufgrund sofortigen oder verzögerten Feedbacks maximiert wird.

Dieser Abschnitt gibt zwar einen grundlegenden Überblick über das verstärkende Lernen, das Thema geht jedoch über den Rahmen dieses Buches hinaus, das sich vornehmlich mit Klassifizierung, Regressionsanalyse und Clustering befasst.

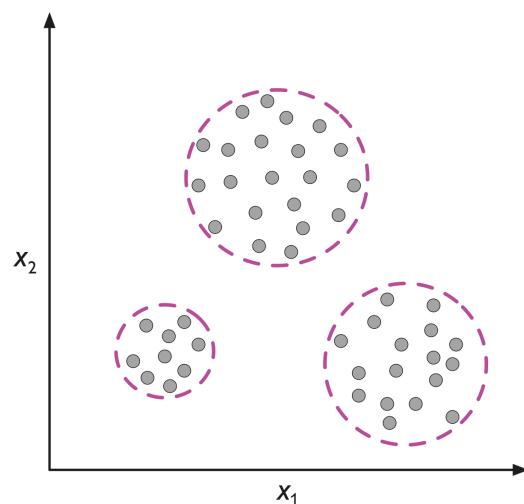
### 1.2.3 Durch unüberwachtes Lernen verborgene Strukturen erkennen

Beim überwachten Lernen ist die richtige Antwort beim Trainieren des Modells bereits im Vorhinein bekannt, und beim verstärkenden Lernen definieren wir eine Bewertung oder *Belohnung* für bestimmte Aktionen des Agenten. Beim unüberwachten Lernen hingegen haben wir es mit nicht gekennzeichneten Daten oder mit Daten *unbekannter Struktur* zu tun. Durch die beim unüberwachten Lernen eingesetzten Verfahren sind wir in der Lage, die Struktur der Daten zu erkunden, um sinnvolle Informationen daraus zu extrahieren, ohne dass es Hinweise auf eine Zielvariable oder eine Belohnungsfunktion gibt.

#### Aufspüren von Untergruppen durch Clustering

*Clustering* ist ein exploratives Datenanalyseverfahren, das es uns gestattet, Informationen in sinnvolle Untergruppen (*Cluster*) aufzuteilen, ohne vorherige Kenntnisse über die Gruppenzugehörigkeit dieser Informationen zu besitzen. Jeder bei der Analyse auftretende Cluster definiert eine Gruppe von Objekten, die bestimmte Eigenschaften gemeinsam haben, sich aber von Objekten in anderen Gruppen hinreichend unterscheiden. Deshalb wird das Clustering manchmal auch als »unüberwachte Klassifizierung« bezeichnet. Es ist ausgezeichnet geeignet, um Informationen zu strukturieren und sinnvolle Beziehungen zwischen den Daten abzuleiten. Beispielsweise ermöglicht es Marketingfachleuten, Kunden anhand ihrer Interessen in Gruppen einzurichten, um gezielte Kampagnen zu entwickeln.

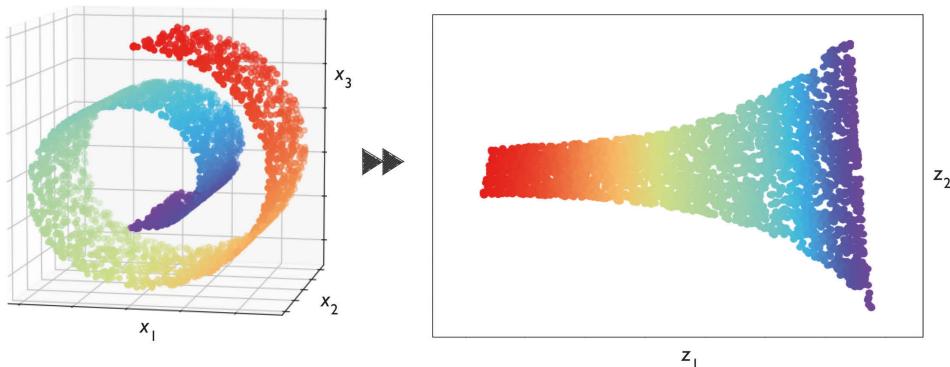
Die folgende Abbildung illustriert, wie man das Clustering-Verfahren zur Organisation nicht gekennzeichneter Daten in drei verschiedene Gruppen verwenden kann, die jeweils ähnliche Merkmale wie  $x_1$  und  $x_2$  aufweisen.



## Datenkomprimierung durch Dimensionsreduktion

Die *Dimensionsreduktion* ist eine weitere Teildisziplin des unüberwachten Lernens. Wir haben es des Öfteren mit Daten hoher Dimensionalität zu tun (jede Beobachtung besteht aus einer Vielzahl von Messwerten), was aufgrund der für die Lernalgorithmen geltenden Beschränkungen von Speicherplatz und Rechenleistung eine Herausforderung darstellen kann. Bei der Vorverarbeitung von Merkmalen wird häufig eine unüberwachte Dimensionsreduktion eingesetzt, um die Daten von sogenanntem »Rauschen« zu befreien. Dies kann allerdings zu einer Abschwächung der Aussagekraft bestimmter Vorhersagealgorithmen führen. Die Daten werden in kleinere Unterräume geringerer Dimensionalität aufgeteilt, wobei der Großteil der relevanten Informationen erhalten bleibt.

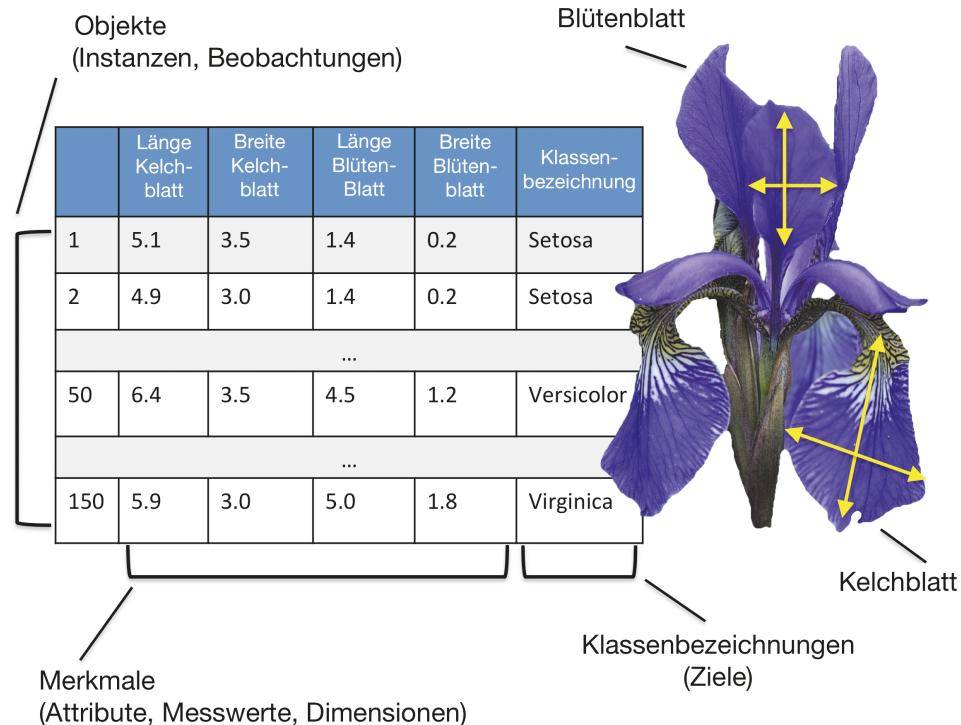
In manchen Fällen ist die Dimensionsreduktion auch für die Visualisierung der Daten nützlich. Beispielsweise können hochdimensionale Merkmalsmengen auf ein-, zwei- oder dreidimensionale Merkmalsräume projiziert werden, um sie als 3-D- oder 2-D-Streudiagramme bzw. -Histogramme darzustellen. Die Abbildung zeigt ein Beispiel, in dem eine nichtlineare Dimensionsreduktion auf eine 3-D-Punktmenge in Form einer Biskuitrolle angewendet wurde, um sie in einen zweidimensionalen Merkmalsraum zu transformieren.



## 1.3 Grundlegende Terminologie und Notation

Nachdem wir nun die drei Arten des Machine Learnings – überwachtes, unüberwachtes und verstärkendes Lernen – erörtert haben, werden wir als Nächstes die grundlegenden Begriffe klären, die in den folgenden Kapiteln Verwendung finden. Die Abbildung zeigt einen Auszug der *Iris-Datensammlung*, einem klassischen Beispiel für den Bereich des Machine Learnings. Dabei handelt es sich um Messdaten von 150 Schwertlilien dreier verschiedener Arten: *Iris setosa*, *Iris versicolor* und *Iris virginica*. Jedes der Blumenexemplare wird in dieser Datensammlung durch eine

Zeile repräsentiert. In den einzelnen Spalten stehen die in Zentimetern angegebenen Messdaten, die wir auch als *Merkmale* der Datenmenge bezeichnen.



Um die Notation und Implementierung einfach aber dennoch effizient zu halten, nutzen wir die Grundlagen der *linearen Algebra*. In den nachfolgenden Kapiteln verwenden wir die Matrizen- und Vektornotation zur Beschreibung der Daten. Wir folgen der üblichen Konvention, dass jedes Objekt durch eine Zeile in der Merkmalsmatrix  $X$  repräsentiert und jedes Merkmal als eigene Spalte gespeichert wird.

Die Iris-Datensammlung besteht aus 150 Datensätzen mit jeweils vier Merkmalen und kann somit als  $150 \times 4$ -Matrix  $X \in \mathbb{R}^{150 \times 4}$  geschrieben werden:

$$\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & x_4^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & x_4^{(2)} \\ \vdots & \vdots & \vdots & \vdots \\ x_1^{(150)} & x_2^{(150)} & x_3^{(150)} & x_4^{(150)} \end{bmatrix}$$

Ab jetzt verwenden wir ein hochgestelltes  $i$  und ein tiefgestelltes  $j$ , um auf das  $i$ -te Trainingsobjekt bzw. die  $j$ -te Dimension der Trainingsdatenmenge zu verweisen.

## Kapitel 1

Wie Computer aus Daten lernen können

Wir notieren Vektoren ( $x \in \mathbb{R}^{n \times 1}$ ) als fettgedruckte Kleinbuchstaben und Matrizen ( $X \in \mathbb{R}^{n \times m}$ ) als fettgedruckte Großbuchstaben. Um auf einzelne Elemente eines Vektors oder einer Matrix zu verweisen, werden kursive Buchstaben benutzt ( $x^{(n)}$  bzw.  $x_{(m)}^{(n)}$ ).

Beispielsweise verweist  $x_1^{150}$  auf die erste Dimension des Blumenexemplars 150, die Länge des Kelchblatts. Jede Zeile der Merkmalsmatrix repräsentiert ein Blumenexemplar und kann als vierdimensionaler Zeilenvektor  $x^{(i)} \in \mathbb{R}^{1 \times 4}$  geschrieben werden, z.B.:

$$\mathbf{x}^{(i)} = \begin{bmatrix} x_1^{(i)} & x_2^{(i)} & x_3^{(i)} & x_4^{(i)} \end{bmatrix}$$

Jede Merkmalsdimension ist ein 150-dimensionaler Spaltenvektor  $\mathbf{x}_j \in \mathbb{R}^{150 \times 1}$ :

$$\mathbf{x}_j = \begin{bmatrix} x_j^{(1)} \\ x_j^{(2)} \\ \vdots \\ x_j^{(150)} \end{bmatrix}$$

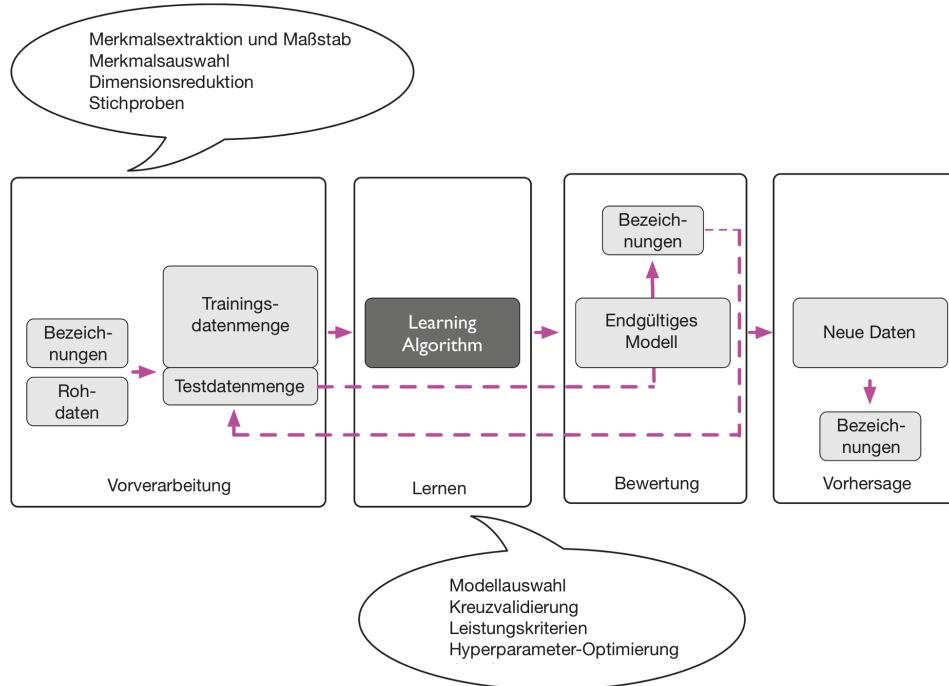
Die Zielvariablen (hier die Klassenbezeichnungen) werden ebenfalls als 150-dimensionale Spaltenvektoren notiert:

$$\mathbf{y} = \begin{bmatrix} y^{(1)} \\ \dots \\ y^{(150)} \end{bmatrix} (y \in \{\text{Setosa, Versicolor, Virginica}\})$$

## 1.4 Entwicklung eines Systems für das Machine Learning

In den vorangegangenen Abschnitten haben wir die grundlegenden Konzepte des Machine Learnings und die drei verschiedenen Arten des Lernens erörtert. In diesem Abschnitt befassen wir uns mit weiteren wichtigen Bestandteilen eines Systems für dieses Verfahren, die den Lernalgorithmus begleiten.

Das folgende Diagramm zeigt den typischen Ablauf, der beim Machine Learning in *Vorhersagemodellen* zum Einsatz kommt, die wir in den folgenden Abschnitten betrachten werden.



### 1.4.1 Vorverarbeitung: Daten in Form bringen

Rohdaten liegen nur selten in einer für die optimale Leistung eines Lernalgorithmus erforderlichen Form vor, deshalb ist die **Vorverarbeitung** der Daten bei jedem Lernalgorithmus von entscheidender Bedeutung. Im Fall der Iris-Datensammlung aus dem vorangegangenen Abschnitt könnten die Rohdaten beispielsweise als eine Reihe von Fotos der Blumenexemplare vorliegen, denen wir sinnvolle Merkmale entnehmen möchten. Das könnten etwa Grundfarbe und Tönung sowie Höhe, Länge und Breite der Pflanzen sein. Bei vielen Lernalgorithmen ist es außerdem erforderlich, dass die ausgewählten Merkmale irgendwie normiert sind (hier müssten die Pflanzen im selben Maßstab dargestellt sein), um ein optimales Ergebnis zu erzielen. Dies wird oftmals dadurch erreicht, dass die ausgewählten Merkmale auf ein Intervall  $[0, 1]$  oder eine Standardnormalverteilung (Mittelwert 0 und Standardabweichung 1) abgebildet werden, wie Sie in den nachfolgenden Kapiteln noch sehen werden.

Manche der ausgewählten Merkmale könnten hochgradig korreliert und daher in gewissem Maße redundant sein. In diesen Fällen sind Verfahren zur Dimensionsreduktion nützlich, um die Merkmale auf einen Merkmalsraum geringerer Dimensionalität abzubilden. Die Dimensionsreduktion des Merkmalsraums hat die Vorteile, dass weniger Speicherplatz benötigt wird und der Lernalgorithmus erheblich schneller arbeitet. In manchen Fällen kann eine Dimensionsreduktion auch die

Vorhersagekraft eines Modells verbessern, nämlich wenn die Datenmenge eine große Anzahl irrelevanter Merkmale (Rauschen) aufweist, das heißt, dass sie ein niedriges Signal-Rauschen-Verhältnis besitzt.

Um festzustellen, ob ein Lernalgorithmus nicht nur die Trainingsdaten ordentlich verarbeitet, sondern auch mit neuen Daten gut zurechtkommt, ist es sinnvoll, den Datenbestand nach dem Zufallsprinzip in separate Trainings- und Testdatenmengen aufzuteilen: Zum Trainieren und Optimieren des Lernmodells verwenden wir die Trainingsdatenmenge, während wir die Testdatenmenge bis zum Schluss zurückhalten, um das endgültige Modell bewerten zu können.

### 1.4.2 Trainieren und Auswählen eines Vorhersagemodells

Wie Sie in den nachfolgenden Kapiteln noch sehen werden, sind viele verschiedene Lernalgorithmen entwickelt worden, mit denen die unterschiedlichsten Aufgabenstellungen erledigt werden können. An dieser Stelle ist es allerdings wichtig festzuhalten, dass das Lernen nicht umsonst zu haben ist – so in etwa könnte man David Wolperts berühmte »No Free Lunch«-Theoreme zusammenfassen (*The Lack of A Priori Distinctions Between Learning Algorithms*, D.H. Wolpert 1996; *No Free Lunch Theorems for Optimization*, D.H. Wolpert und W.G. Macready, 1997). Noch besser lässt sich dieses Konzept anhand eines berühmten Zitats veranschaulichen: »Wenn das einzige verfügbare Werkzeug ein Hammer ist, dürfte es verlockend sein, alles wie einen Nagel zu behandeln.« (Abraham Maslow, 1966). Beispielsweise sind alle Klassifizierungsalgorithmen in irgendeiner Weise voreingenommen und kein Klassifizierungsmodell ist anderen überlegen, wenn man nicht bestimmte Annahmen über die Aufgabenstellung macht. In der Praxis ist es daher von entscheidender Bedeutung, wenigstens eine Handvoll verschiedener Algorithmen zu vergleichen, um das am besten funktionierende Modell zu trainieren und auszuwählen. Aber um Vergleiche zwischen verschiedenen Modellen anstellen zu können, müssen zunächst einmal Bewertungskriterien festgelegt werden. Ein gebräuchliches Kriterium ist die *Korrektklassifizierungsrate* (Vertrauenswahrscheinlichkeit) des Modells, die als Quote der korrekten Klassifizierungen definiert ist.

Nun stellt sich natürlich die Frage: *Wie kann man wissen, welches Modell mit den Testdaten und den »echten« Daten gut funktioniert, wenn man sie nicht bei der Auswahl des Modells verwendet, sondern bis zur Bewertung des endgültigen Modells zurückhält?* Um das mit dieser Frage verbundene Problem zu lösen, können verschiedene Kreuzvalidierungsverfahren eingesetzt werden, bei denen die Trainingsdatenmenge weiter in Trainings- und Validierungsteilmengen aufgeteilt wird, um die *Allgemeinertüchtigkeit* des Modells abzuschätzen. Und schließlich dürfen wir auch nicht erwarten, dass die Standardparameter der Lernalgorithmen verschiedener Softwarebibliotheken für unsere spezielle Aufgabenstellung optimiert sind. Daher werden wir in den noch folgenden Kapiteln häufig Gebrauch von Verfahren zur *Hyperparameter-Optimierung* machen, um die Leistung unseres Modells feiner

abzustimmen. Man kann sich diese Hyperparameter als Parameter vorstellen, die nicht anhand der Daten ermittelt werden, sondern als Einstellungsmöglichkeiten zur Verbesserung der Leistung, was sehr viel klarer werden wird, wenn wir uns in den folgenden Kapiteln einige dazu passende Beispiele ansehen.

### 1.4.3 Bewertung von Modellen und Vorhersage anhand unbekannter Dateninstanzen

Nach der Auswahl eines an die Trainingsdaten angepassten Modells können wir die Testdatenmenge verwenden, um zu ermitteln, wie gut es mit diesen unbekannten Daten bei der Einschätzung des Verallgemeinerungsfehlers zurechtkommt. Sofern die Leistung des Modells zufriedenstellend ausfällt, können wir es verwenden, um anhand neuer, zukünftiger Daten Vorhersagen zu treffen. Hier muss angemerkt werden, dass die Parameter der vorhin erwähnten Verfahren (wie der Maßstab der Merkmalsdarstellungen oder die Dimensionsreduktion) ausschließlich anhand der Trainingsdatenmenge ermittelt werden. Dieselben Parameter werden später auch auf die Testdatenmenge und neue Daten angewendet – ansonsten könnte die bei den Testdaten gemessene Leistung zu optimistisch sein.

## 1.5 Machine Learning mit Python

Im Bereich Data Science ist Python eine der beliebtesten Programmiersprachen, daher gibt es eine Vielzahl nützlicher Bibliotheken, die von der Python-Community entwickelt wurden.

Die Performance von Interpretersprachen wie Python ist derjenigen von komplizierten Programmiersprachen zwar unterlegen, es gibt allerdings Erweiterungsbibliotheken wie *NumPy* und *SciPy*, die auf maschinennahen Fortran- und C-Implementierungen beruhen, um schnelle Berechnungen mit mehrdimensionalen Arrays auszuführen.

Bei Aufgabenstellungen des Machine Learnings werden wir zumeist auf *scikit-learn* zurückgreifen, eine weit verbreitete und leicht verständliche Open-Source-Bibliothek für Machine Learning.

### 1.5.1 Python-Pakete installieren

Python ist für die drei wichtigsten Betriebssysteme Microsoft Windows, macOS und Linux verfügbar. Das Installationsprogramm und die Dokumentation stehen unter <https://www.python.org> zum Herunterladen bereit.

Dieses Buch setzt mindestens die Python-Version 3.6.0 voraus, es empfiehlt sich jedoch, immer die neueste verfügbare Python-3-Version zu verwenden. Die meisten Codebeispiele sind möglicherweise auch mit Python-Versionen ab 2.7.13 kompatibel. Wenn Sie Python 2.7 verwenden, sollten Sie sich über die wichtigsten

Abweichungen der beiden Python-Versionen im Klaren sein. Eine gute Zusammenfassung der Unterschiede zwischen Python 3.5 und 2.7 finden Sie unter <https://wiki.python.org/moin/Python2orPython3>.

Die zusätzlichen Pakete, die wir im Buch benutzen werden, können mit `pip` installiert werden. Dieses Installationsprogramm gehört seit der Python-Version 3.3 zur Standardbibliothek. Weitere Informationen über `pip` finden Sie unter <https://docs.python.org/3/installing/index.html>.

Nach erfolgreicher Python-Installation können Sie mit `pip` wie folgt weitere Python-Pakete installieren:

```
pip install Paketname
```

Bereits installierte Pakete können mit dem `--upgrade`-Flag aktualisiert werden:

```
pip install Paketname --upgrade
```

### 1.5.2 Verwendung der Python-Distribution Anaconda

Von Continuum Analytics gibt es eine sehr empfehlenswerte alternative Python-Distribution für wissenschaftliches Rechnen namens *Anaconda*. Hierbei handelt es sich um eine – auch für den kommerziellen Gebrauch – kostenlose Python-Distribution, die alle wichtigen Python-Pakete für Data Science, Mathematik und Engineering in einem einzigen, benutzerfreundlichen und plattformunabhängigen Paket bündelt. Das Installationsprogramm können Sie unter <http://continuum.io/downloads> herunterladen. Eine Kurzanleitung ist unter <https://conda.io/docs/test-drive.html> verfügbar.

Nach der Installation von Anaconda können Python-Pakete mit dem folgenden Befehl installiert werden:

```
conda install Paketname
```

Bereits vorhandene Pakete werden so aktualisiert:

```
conda update Paketname
```

### 1.5.3 Pakete für wissenschaftliches Rechnen, Data Science und Machine Learning

Im weiteren Verlauf des Buches werden wir vornehmlich *NumPys* mehrdimensionale Arrays verwenden, um Daten zu speichern und zu verarbeiten. Gelegentlich kommt auch *pandas* zum Einsatz, eine auf NumPy beruhende Bibliothek, die erweiterte Funktionen für die noch komfortablere Verarbeitung von Tabellendaten bereitstellt. Zur Ergänzung des Lernerlebnisses werden wir darüber hinaus die

sehr anpassungsfähige `matplotlib`-Bibliothek einsetzen, die für die Visualisierung und das intuitive Verständnis quantitativer Daten oft äußerst nützlich ist.

Die Versionsnummern der im Buch verwendeten Python-Pakete sind nachstehend aufgeführt. Vergewissern Sie sich, dass Ihre installierten Pakete mindestens diesen Versionsnummern entsprechen, damit gewährleistet ist, dass die Codebeispiele korrekt ausgeführt werden.

- NumPy 1.12.1
- SciPy 0.19.0
- scikit-learn 0.18.1
- matplotlib 2.0.2
- pandas 0.20.1

## 1.6 Zusammenfassung

In diesem Kapitel haben wir einen ganz allgemeinen Blick auf das Thema Machine Learning geworfen und uns mit dem Gesamtbild sowie den grundlegenden Konzepten vertraut gemacht, die wir in den folgenden Kapiteln eingehender betrachten werden. Wir haben erfahren, dass überwachtes Lernen aus zwei wichtigen Teilgebieten besteht: Klassifizierung und Regression. Klassifizierungsmodelle ermöglichen es, Objekte bekannten Klassen zuzuordnen und wir können die Regressionsanalyse nutzen, um stetige Werte einer Zielvariablen vorherzusagen. Das unüberwachte Lernen bietet nicht nur praktische Verfahren zum Auffinden von Strukturen in nicht gekennzeichneten Daten, es kann darüber hinaus bei der Vorverarbeitung auch zur Datenkomprimierung eingesetzt werden. Wir haben uns kurz die typische Vorgehensweise bei der Anwendung des Machine Learnings auf Problemstellungen angesehen, die bei der weiteren Erörterung und für praktische Beispiele in den folgenden Kapiteln als Grundlage dient. Darüber hinaus haben wir unsere Python-Umgebung eingerichtet und die erforderlichen Pakete aktualisiert und sind nun bereit, uns Machine Learning in Aktion anzusehen.

Im weiteren Verlauf des Buches werden wir neben dem Machine Learning selbst verschiedene Verfahren zur Vorverarbeitung von Daten vorstellen, die dabei helfen, mit verschiedenen Lernalgorithmen die beste Leistung zu erzielen. Wir werden uns im gesamten Buch ziemlich ausführlich mit Klassifizierungsalgorithmen befassen, aber auch einige Verfahren der Regressionsanalyse und des Clusterings betrachten.

Vor uns liegt eine interessante Tour, auf der viele leistungsfähige Verfahren des weiten Felds Machine Learning zur Sprache kommen. Wir gehen jedoch schrittweise vor und bauen auf das in den einzelnen Kapiteln allmählich erworbene Wissen auf.

## Kapitel 1

Wie Computer aus Daten lernen können

Im nächsten Kapitel beginnt diese Tour mit der Implementierung einer der ersten Lernalgorithmen zum Zweck der Klassifizierung, die uns auf das Kapitel 3 (*Machine-Learning-Klassifizierer mit scikit-learn verwenden*) vorbereitet, in dem wir die scikit-learn-Bibliothek nutzen werden, um erweiterte Lernalgorithmen zu erörtern.

# Lernalgorithmen für die Klassifizierung trainieren

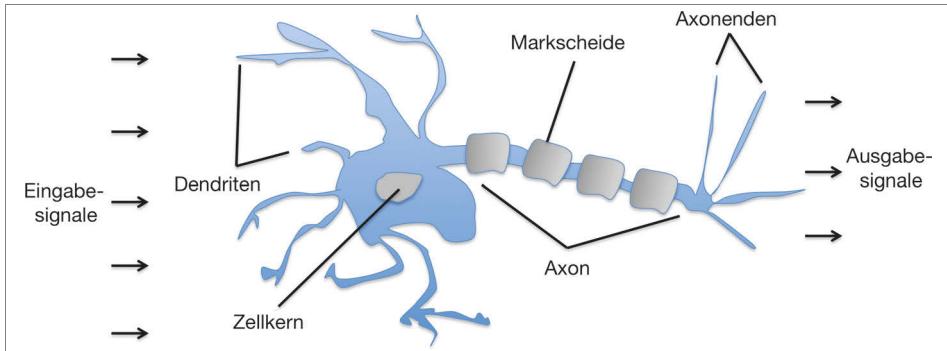
In diesem Kapitel werden wir einen der ersten in der Literatur beschriebenen Klassifizierungsalgorithmen verwenden: das *Perzeptron* und die damit einhergehenden *adaptiven linearen Neuronen*. Zunächst werden wir Schritt für Schritt ein Perzeptron in Python implementieren und darauf trainieren, die verschiedenen Blumenarten in der Iris-Datensammlung zu klassifizieren. Das wird uns dabei helfen, das Konzept eines Lernalgorithmus zur Klassifizierung und die effiziente Implementierung solch eines Algorithmus in Python zu verstehen. Dann werden wir die Grundlagen der Optimierung unter Verwendung adaptiver linearer Neuronen erörtern, die ihrerseits als Vorbereitung dazu dient, in Kapitel 3 leistungsfähigere Klassifizierer der scikit-learn-Bibliothek einzusetzen.

Die Themen in diesem Kapitel sind:

- Ein Gespür für Machine-Learning-Algorithmen entwickeln
- Die Verwendung von Pandas, NumPy und Matplotlib zum Einlesen, Verarbeiten und Visualisieren von Daten
- Implementierung linearer Klassifizierungsalgorithmen in Python

## 2.1 Künstliche Neuronen: Ein kurzer Blick auf die Anfänge des Machine Learnings

Bevor wir uns das Perzeptron und ähnliche Algorithmen genauer ansehen, wollen wir einen kurzen Ausflug zu den Anfängen des Machine Learnings unternehmen. Warren McCulloch und Walter Pitts hatten sich zum Ziel gesetzt, die Funktionsweise des Gehirns zu ergründen, um eine künstliche Intelligenz zu entwickeln, und veröffentlichten 1943 das erste Konzept einer vereinfachten Hirnzelle, das sogenannte *MCP-Neuron* (*McCulloch-Pitts-Neuron*, W.S. McCulloch und W. Pitts, *A Logical Calculus of the Ideas Immanent in Nervous Activity*, *The bulletin of mathematical biophysics*, 5(4):115-133, 1943). Bei Neuronen handelt es sich um mit einander verknüpfte Nervenzellen des Gehirns, die an der Verarbeitung und Weiterleitung chemischer und elektrischer Signale beteiligt sind, wie die folgende Abbildung illustriert.



McCulloch und Pitts beschrieben diese Nervenzelle als ein einfaches logisches Gatter mit binärer Ausgabe. Mehrere Eingabesignale erreichen die Dendriten und laufen im Zellkörper zusammen. Wenn ein bestimmter Schwellenwert erreicht ist, erzeugt die Zelle ein Ausgabesignal, das an das Axon weitergeleitet wird.

Nur wenige Jahre später veröffentlichte Frank Rosenblatt das erste Konzept einer Perzepron-Lernregel, die auf dem MCP-Neuronenmodell beruhte (F. Rosenblatt, *The Perceptron, a Perceiving and Recognizing Automaton*, Cornell Aeronautical Laboratory, 1957). Hiermit schlug er einen Algorithmus vor, der automatisch die optimalen Gewichtungskoeffizienten ermittelte, die mit den Eingabemerkmalsen multipliziert werden, um zu entscheiden, ob ein Neuron feuert oder nicht. Im Zusammenspiel mit dem überwachten Lernen und der Klassifizierung könnte solch ein Algorithmus eingesetzt werden, um zu prognostizieren, zu welcher von zwei Klassen ein Objekt gehört.

### 2.1.1 Formale Definition eines künstlichen Neurons

Formal kann diese Aufgabe als *binäre* oder *dichotome Klassifizierung* betrachtet werden, in der wir die beiden Klassen der Einfachheit halber als 1 (positive Klasse) und -1 (negative Klasse) bezeichnen. Wir können dann eine *Aktivierungsfunktion*  $\phi(z)$  definieren, die eine Linearkombination bestimmter Eingabewerte  $x$  und einen entsprechenden Gewichtungsvektor  $w$  entgegennimmt, wobei  $z$  die sogenannte *Nettoeingabe* ( $z = w_1x_1 + \dots + w_mx_m$ ) ist:

$$w = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

Wenn nun die Aktivierungsfunktion eines bestimmten Objekts  $x^{(i)}$ , also die Ausgabe von  $\phi(z)$ , einen bestimmten Schwellenwert  $\theta$  übertrifft, wird die Klasse 1 vorhergesagt, anderenfalls die Klasse -1. Beim Perzepron-Algorithmus ist die

Aktivierungsfunktion  $\phi(z)$  eine einfache *Sprungfunktion*, die auch als *Heaviside-Funktion* bezeichnet wird:

$$\phi(z) = \begin{cases} 1 & \text{wenn } z \geq \theta \\ -1 & \text{anderenfalls} \end{cases}$$

Der Einfachheit halber bringen wir den Schwellenwert  $\theta$  auf die linke Seite der Gleichung und definieren eine Nullgewichtung als  $w_0 = -\theta$  sowie  $x_0 = 1$ , sodass wir  $\mathbf{z}$  kompakter schreiben können:

$$z = w_0 x_0 + w_1 x_1 + \dots + w_m x_m = \mathbf{w}^T \mathbf{x} \text{ und } \phi(z) = \begin{cases} 1 & \text{wenn } z \geq \theta \\ -1 & \text{anderenfalls} \end{cases}.$$

In der Literatur wird der negative Schwellenwert oder die Nullgewichtung  $w_0 = -\theta$  für gewöhnlich als Bias-Einheit bezeichnet.

In den folgenden Abschnitten werden wir des Öfteren von der grundlegenden Notation der linearen Algebra Gebrauch machen. Beispielsweise kürzen wir die Summe der Produkte der Werte von  $\mathbf{x}$  und  $\mathbf{w}$  als Skalarprodukt zweier Vektoren ab, wobei das hochgestellte T für die *Transponierte* steht. Beim Transponieren werden Spaltenvektoren in Zeilenvektoren (und umgekehrt) transformiert:

$$z = w_0 x_0 + w_1 x_1 + \dots + w_m x_m = \sum_{j=0}^m \mathbf{x}_j \mathbf{w}_j = \mathbf{w}^T \mathbf{x}$$

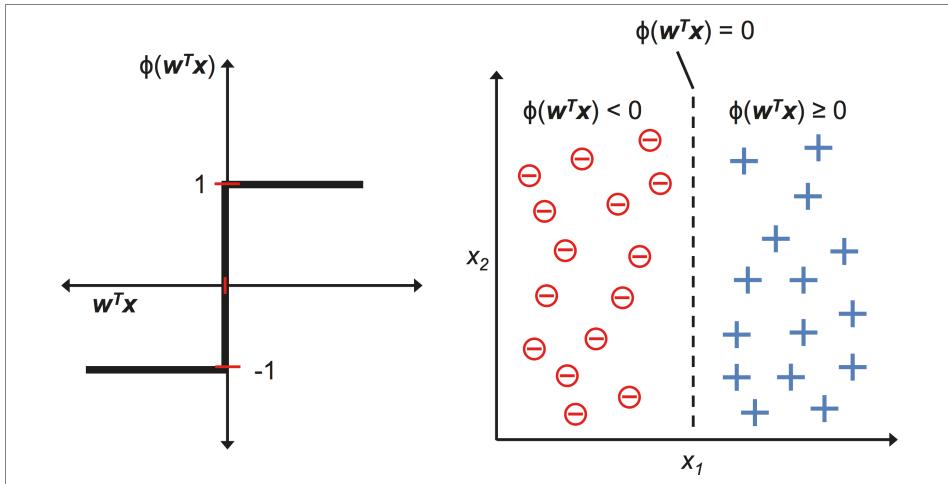
Zum Beispiel:  $[1 \ 2 \ 3] \times \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = 1 \times 4 + 2 \times 5 + 3 \times 6 = 32$

Darüber hinaus kann die Transponierung auch auf eine Matrix angewendet werden, beispielsweise um die Elemente an der Diagonalen zu spiegeln:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

In diesem Buch werden wir nur grundlegende Konzepte der linearen Algebra verwenden. Wenn Sie Ihre Kenntnisse auffrischen möchten, sollten Sie sich Zico Kolters ausgezeichnete und kostenlose Zusammenfassung der linearen Algebra ansehen, die unter [http://www.cs.cmu.edu/~zko/ter/course/linalg/linalg\\_notes.pdf](http://www.cs.cmu.edu/~zko/ter/course/linalg/linalg_notes.pdf) verfügbar ist.

Die folgende Abbildung veranschaulicht, wie die Nettoeingabe  $z = \mathbf{w}^T \mathbf{x}$  durch die Aktivierungsfunktion des Perzeptrons (in der Abbildung links) in eine binäre Ausgabe (-1 oder 1) umgewandelt wird und wie sie dazu verwendet werden kann, zwei linear trennbare Klassen zu unterscheiden (in der Abbildung rechts).



## 2.1.2 Die Perzeptron-Lernregel

Die dem MCP-Neuron und Rosenblatts mit einem Schwellenwert versehenen Perzeptron-Modell zugrunde liegende Idee besteht darin, einen reduzierten Ansatz zu verwenden, um zu simulieren, wie ein einzelnes Neuron im Gehirn funktioniert – entweder es feuert oder es feuert nicht. Rosenblatts ursprüngliche Perzeptron-Regel ist daher ziemlich einfach und kann durch die folgenden Schritte zusammengefasst werden:

1. Die Gewichtungen werden mit 0 oder kleinen zufälligen Werten initialisiert.
2. Mit jedem zum Training eingesetzten Objekt  $x^{(i)}$  werden folgende Schritte durchgeführt:
  1. Berechnung des Ausgabewertes  $\hat{y}$
  2. Aktualisierung der Gewichtungen

Der Ausgabewert ist hier die von der vorhin definierten Sprungfunktion vorhergesagte Klassenbezeichnung. Die gleichzeitige Aktualisierung der Gewichtungen  $w_j$  im Gewichtungsvektor  $w$  kann formal folgendermaßen geschrieben werden:

$$w_j := w_j + \Delta w_j$$

Der Wert von  $\Delta w_j$ , der zur Aktualisierung der Gewichtung  $w_j$  verwendet wird, lässt sich anhand der Perzeptron-Lernregel berechnen:

$$\Delta w_j = \eta (y^{(i)} - \hat{y}^{(i)}) x_j^{(i)}$$

Dabei ist  $\eta$  die *Lernrate* (typischerweise eine Konstante zwischen 0,0 und 1,0),  $y^{(i)}$  ist die *tatsächliche Klassenbezeichnung* des  $i$ -ten Trainingsobjekts und  $\hat{y}^{(i)}$  ist die *vorhergesagte Klassenbezeichnung*. Hier ist es wichtig anzumerken, dass alle Gewich-

tungen im Gewichtungsvektor gleichzeitig aktualisiert werden, was bedeutet, dass  $\hat{y}^{(i)}$  nicht erneut berechnet werden muss, bevor alle Gewichtungen  $\Delta w_j$  aktualisiert wurden. Im konkreten Fall einer 2-D-Datensammlung würden wir die Aktualisierung wie folgt schreiben:

$$\Delta w_0 = \eta \left( y^{(i)} - \text{Ausgabe}^{(i)} \right)$$

$$\Delta w_1 = \eta \left( y^{(i)} - \text{Ausgabe}^{(i)} \right) x_1^{(i)}$$

$$\Delta w_2 = \eta \left( y^{(i)} - \text{Ausgabe}^{(i)} \right) x_2^{(i)}$$

Bevor wir die Perzeptron-Regel in Python implementieren, wollen wir ein einfaches Gedankenexperiment anstellen, um zu illustrieren, wie wunderbar einfach diese Lernregel tatsächlich ist. In den beiden Szenarien, in denen das Perzeptron die Klassenbezeichnung korrekt vorhersagt, bleiben die Gewichtungen unverändert:

$$\Delta w_j = \eta(-1 - -1) x_j^{(i)} = 0$$

$$\Delta w_j = \eta(1 - 1) x_j^{(i)} = 0$$

Im Fall einer falschen Vorhersage werden die Gewichtungen in Richtung der positiven bzw. negativen Zielklasse verschoben:

$$\Delta w_j = \eta(1 - -1) x_j^{(i)} = \eta(2) x_j^{(i)}$$

$$\Delta w_j = \eta(-1 - 1) x_j^{(i)} = \eta(-2) x_j^{(i)}$$

Um ein besseres Gespür für den multiplikativen Faktor  $x_j^{(i)}$  zu bekommen, sehen wir uns ein weiteres einfaches Beispiel an, für das gilt:

$$y^{(i)} = +1, \hat{y}^{(i)} = -1, \eta = 1$$

Nehmen wir an, dass  $x_j^{(i)} = 0.5$  ist und wir dieses Objekt irrtümlich als -1 klassifizieren. In diesem Fall würden wir die zugehörige Gewichtung um 1 erhöhen, damit die Nettoeingabe  $x_j^{(i)} \times w_j$  positiver ist, wenn wir das nächste Mal auf dieses Objekt treffen – denn dadurch wird die Wahrscheinlichkeit erhöht, dass der Schwellenwert der Sprungfunktion überschritten und das Objekt somit als +1 klassifiziert wird:

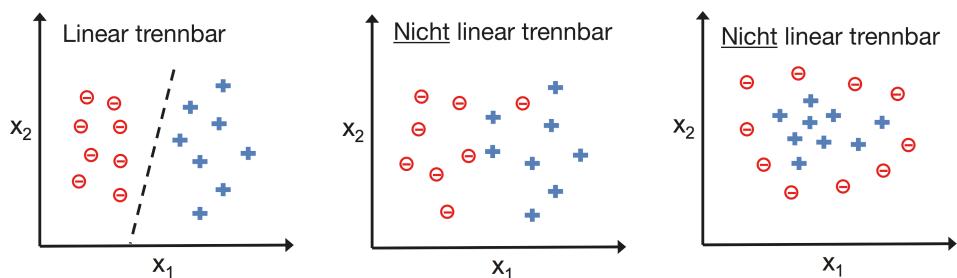
$$\Delta w_j = (1 - -1) 0.5 = (2) 0.5 = 1$$

Die Aktualisierung der Gewichtung ist proportional zum Wert von  $x_j^{(i)}$ . Wenn wir beispielsweise ein weiteres Objekt mit  $x_j^{(i)} = 2$  irrtümlich als -1 klassifizieren, wür-

den wir die Entscheidungsgrenze zur korrekten Klassifizierung dieses Objekts beim nächsten Versuch sogar noch weiter verschieben:

$$\Delta w_j = (1 - -1)2 = (2)2 = 4$$

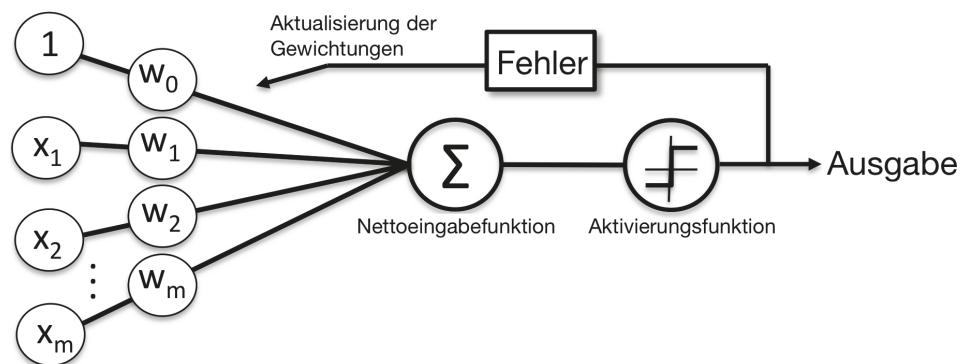
Beachten Sie hier, dass die Konvergenz des Perzeptrons nur dann garantiert ist, wenn die beiden Klassen linear trennbar sind (siehe Abbildung) und die Lernrate hinreichend klein ist. Lassen sich die beiden Klassen nicht durch eine lineare Entscheidungsgrenze trennen, können wir eine maximale Anzahl von Durchläufen der Trainingsdaten (*Epochen*) und/oder einen Schwellenwert für die Anzahl der tolerierbaren Fehlklassifizierungen festlegen – das Perzeptron würde anderenfalls endlos mit Aktualisierungen fortfahren.



### Tipp

Die Codebeispiele können Sie unter <http://www.mitp.de/733> herunterladen.

Lassen Sie uns das Ganze in einer einfachen Abbildung zusammenfassen, die das allgemeine Konzept des Perzeptrons veranschaulicht, bevor wir uns im nächsten Abschnitt der Implementierung zuwenden.



Die Abbildung illustriert, wie das Perzeptron die Eingabe  $x$  entgegennimmt und sie mit den Gewichtungen  $w$  kombiniert, um die Nettoeingabefunktion zu berechnen. Die Nettoeingabe wird dann der Aktivierungsfunktion (hier der Heaviside-Funktion) übergeben, die eine binäre Ausgabe erzeugt, nämlich -1 oder +1 – die Vorhersage für die Klassenbezeichnung des Objekts. Während der Lernphase wird diese Ausgabe genutzt, um Fehler festzustellen und die Gewichtungen zu aktualisieren.

## 2.2 Implementierung eines Perzeptron-Lernalgorithmus in Python

Im vorangegangenen Abschnitt haben Sie erfahren, wie Rosenblatts Perzeptron-Regel funktioniert. Diese wollen wir nun in Python implementieren und auf die Iris-Datensammlung anwenden, die in Kapitel 1 vorgestellt wurde.

### 2.2.1 Eine objektorientierte Perzeptron-API

Wir werden einen objektorientierten Ansatz verfolgen und die Perzeptron-Schnittstelle als eine Python-Klasse definieren, die es erlaubt, neue Perzeptron-Objekte zu initialisieren, die anhand einer `fit`-Methode aus Daten lernen und mittels einer `predict`-Methode Vorhersagen treffen können. Hierbei verwenden wir die Konvention, einen Unterstrich an Attribute anzuhängen, die nicht bei der Objektinitialisierung, sondern durch den Aufruf anderer Methoden des Objekts erzeugt werden, z.B. in der Form `self.w_`.

#### Tipp

Wenn Ihnen die wissenschaftlichen Python-Bibliotheken noch nicht vertraut sind oder Sie Ihre Kenntnisse auffrischen möchten, sollten Sie sich die folgenden Informationsquellen näher ansehen:

NumPy: [https://sebastianraschka.com/pdf/books/dlb/appendix\\_f\\_numpy-intro.pdf](https://sebastianraschka.com/pdf/books/dlb/appendix_f_numpy-intro.pdf)

Pandas: <https://pandas.pydata.org/pandas-docs/stable/10min.html>

Matplotlib: <http://matplotlib.org/users/beginner.html>

Nachstehend die Implementierung eines Perzeptrons:

```
import numpy as np
class Perceptron(object):
    """Perzeptron-Klassifizierer
```

## Kapitel 2

### Lernalgorithmen für die Klassifizierung trainieren

```
Parameter
-----
eta : float
    Lernrate (zwischen 0.0 und 1.0)
n_iter : int
    Durchläufe der Trainingsdatenmenge
random_state : int
    Zufallszahlengenerator für zufällige Gewichtung
    initialisieren.

Attribute
-----
w_ : 1d-array
    Gewichtungen nach Anpassung
errors_ : list
    Anzahl der Fehlklassifizierungen (Updates) pro Epoche

"""
def __init__(self, eta=0.01, n_iter=50, random_state=1):
    self.eta = eta
    self.n_iter = n_iter
    self.random_state = random_state

def fit(self, X, y):
    """Anpassen an die Trainingsdaten

Parameter
-----
X : {array-like}, shape = [n_samples, n_features]
    Trainingvektoren, n_samples ist
    die Anzahl der Objekte und
    n_features ist die Anzahl der Merkmale
y : array-like, shape = [n_samples]
    Zielwerte

Rückgabewert
-----
self : object

"""
rgen = np.random.RandomState(self.random_state)
self.w_ = rgen.normal(loc=0.0, scale=0.01,
                      size=1 + X.shape[1])
self.errors_ = []

for _ in range(self.n_iter):
```

```

        errors = 0
        for xi, target in zip(X, y):
            update = self.eta * (target - \
                                  self.predict(xi))
            self.w_[1:] += update * xi
            self.w_[0] += update
            errors += int(update != 0.0)
        self.errors_.append(errors)
    return self

    def net_input(self, X):
        """Nettoeingabe berechnen"""
        return np.dot(X, self.w_[1:]) + self.w_[0]
    def predict(self, X):
        """Klassenbezeichnung zurückgeben"""
        return np.where(self.net_input(X) >= 0.0, 1, -1)

```

Mit dieser Implementierung können wir jetzt neue Perceptron-Objekte mit einer gegebenen Lernrate `eta` und der Anzahl der Epochen `n_iter` (Durchläufe der Trainingsdaten) initialisieren. In der `fit`-Methode werden die Gewichtungen in `self.w_` mit einem Vektor  $\mathbb{R}^{m+1}$  initialisiert, wobei  $m$  die Anzahl der Dimensionen (Merkmale) in der Datensammlung angibt und dem ersten Element dieses Vektors, das die Bias-Einheit repräsentiert, 1 hinzufügt wird. Wie Sie wissen, repräsentiert das erste Element dieses Vektors, `self.w[0]`, die vorhin erwähnte sogenannte Bias-Einheit.

Beachten Sie außerdem, dass dieser Vektor kleine Zufallszahlen enthält, die via `rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])` einer Normalverteilung mit der Standardabweichung 0,01 entnommen werden. `rgen` ist hier ein NumPy-Zufallszahlengenerator, der mit einem benutzerdefinierten Wert initialisiert wird, sodass bei Bedarf vorhergehende Ergebnisse reproduzierbar sind.

Die Gewichtungen werden nicht mit null initialisiert, weil sich die Lernrate  $\eta$  (`eta`) nur dann auf das Ergebnis der Klassifizierung auswirkt, wenn die Gewichtungen von null verschiedene Werte besitzen. Wären alle Gewichtungen null, würde die Lernrate `eta` nur die Größe des Gewichtungsvektors beeinflussen, nicht aber die Richtung. Wenn Ihnen die Trigonometrie geläufig ist, betrachten Sie die Vektoren `v1=[1 2 3]` und `v2= 0.5*xv1`. Der Winkel zwischen `v1` und `v2` wäre genau null, wie der folgende Code zeigt:

```

>>> v1 = np.array([1, 2, 3])
>>> v2 = 0.5 * v1
>>> np.arccos(v1.dot(v2) / (np.linalg.norm(v1) *
...                           np.linalg.norm(v2)))
0.0

```

Hier ist `np.arccos` die Umkehrfunktion der Cosinusfunktion und `np.linalg.norm` berechnet die Länge eines Vektors. (Dass wir die Zufallszahlen einer Normalverteilung – und nicht etwa einer gleichmäßigen Verteilung – entnehmen und eine Standardabweichung von 0.01 verwenden, ist willkürlich; wir sind lediglich an kleinen zufälligen Werten interessiert, um die eben erwähnten Eigenschaften von Vektoren zu umgehen, deren Komponenten alle null sind.)

### Tipp

Die Indizierung eindimensionaler NumPy-Arrays erfolgt auf dieselbe Weise wie bei Python-Listen – mit eckigen Klammern (`[]`). Bei zweidimensionalen Arrays verweist der erste Index auf die Zeilennummer und der zweite auf die Spaltennummer. Um das Element in der dritten Zeile der vierten Spalte eines 2-D-Arrays `X` auszuwählen, würde man `X[2, 3]` verwenden.

Nach der Initialisierung der Gewichtungen durchläuft die `fit`-Methode alle in der Trainingsdatenmenge enthaltenen Objekte und aktualisiert die Gewichtungen gemäß der im vorangegangenen Abschnitt erörterten Perzeptron-Lernregel. Die Klassenbezeichnungen werden von der `predict`-Methode vorausgesagt, die auch in der `fit`-Methode aufgerufen wird, um die Klassenbezeichnung für die Aktualisierung einer Gewichtung vorherzusagen. Die `predict`-Methode kann aber auch zur Prognostizierung der Klassenbezeichnungen neuer Daten verwendet werden, nachdem unser Modell daran angepasst wurde. Außerdem sammeln wir in der Liste `self.errors_` die in jeder Epoche auftretenden Fehlklassifizierungen, um später analysieren zu können, wie gut das Perzeptron während des Trainings funktioniert hat. Die in der `net_input`-Methode aufgerufene Funktion `np.dot` berechnet einfach nur das Skalarprodukt  $w^T x$ .

### Tipp

Anstatt NumPy zur Berechnung des Skalarprodukts zweier Arrays via `a.dot(b)` oder `np.dot(a,b)` zu verwenden, könnten wir die Berechnung auch in reinem Python als `sum([i*j for i,j in zip(a,b)])` ausführen. NumPy hat gegenüber normalem Python jedoch den Vorteil, dass arithmetische Operationen vektorisiert sind. *Vektorisierung* bedeutet, dass einfache arithmetische Operationen automatisch auf alle Elemente eines Arrays angewendet werden. Indem wir arithmetische Operationen als eine Reihe von Array-Rechenanweisungen formulieren, statt die Berechnungen mit den einzelnen Elementen auszuführen, können wir die Fähigkeiten moderner CPUs mit SIMD-Architektur (*Single Instruction, Multiple Data*) besser nutzen.

Darüber hinaus verwendet NumPy hochoptimierte Bibliotheken für lineare Algebra, wie BLAS (*Basic Linear Algebra Subprograms*) und LAPACK (*Linear Algebra Package*), die in C oder Fortran programmiert sind. Und schließlich ermöglicht NumPy es außerdem, unseren Code kompakter und intuitiver mit der in der linearen Algebra üblichen Notation zu formulieren, wie etwa Skalarprodukte von Vektoren und Matrizen.

### 2.2.2 Trainieren eines Perzeptron-Modells auf die Iris-Datensammlung

Zum Testen unserer Perzeptron-Implementierung werden wir die beiden Klassen *Setosa* und *Versicolor* aus der Iris-Datensammlung einlesen. Die Perzeptron-Regel ist zwar nicht auf zwei Dimensionen beschränkt, wir werden bei der Visualisierung jedoch nur die beiden Merkmale »Länge des Kelchblatts« und »Länge des Blütenblatts« betrachten. Aus praktischen Gründen haben wir auch lediglich die beiden Klassen *Setosa* und *Versicolor* ausgewählt. Der Perzeptron-Algorithmus kann allerdings erweitert werden, um eine Mehrfachklassifizierung vorzunehmen, beispielsweise durch ein OvA-Verfahren (siehe Kasten).

#### Tipp

*One-vs.-All* (OvA), oder auch *One-vs.-Rest* (OvR), ist ein Verfahren, um binäre Klassifizierungen zu Mehrfachklassifizierung zu erweitern. Dabei wird pro Klasse ein Klassifizierer trainiert und die jeweilige Klasse wird als positive Klasse behandelt, während alle übrigen Klassen als negative Klassen betrachtet werden. Bei der Klassifizierung einer neuen Datensammlung würden wir unsere  $n$  Klassifizierer ( $n$  gibt die Anzahl der Klassenbezeichnungen an) verwenden und der betreffenden Klasse die Bezeichnung zuordnen, die am verlässlichsten klassifiziert wird. Im Fall des Perzeptrons würden wir OvA verwenden, um die Klassenbezeichnung auszuwählen, dessen Nettoeingabe den größten Absolutwert besitzt.

Zunächst importieren wir die Pandas-Bibliothek, lesen die Iris-Datensammlung vom *UCI Machine Learning Repository* direkt in ein `DataFrame`-Objekt ein und geben mit der `tail`-Methode die letzten fünf Zeilen aus, um zu überprüfen, ob die Daten korrekt geladen wurden:

```
>>> import pandas as pd  
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/'  
... 'machine-learning-databases/iris/iris.data', header=None)  
>>> df.tail()
```

	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>
<b>145</b>	6.7	3.0	5.2	2.3	Iris virginica
<b>146</b>	6.3	2.5	5.0	1.9	Iris virginica
<b>147</b>	6.5	3.0	5.2	2.0	Iris virginica
<b>148</b>	6.2	3.4	5.4	2.3	Iris virginica
<b>149</b>	5.9	3.0	5.1	1.8	Iris virginica

Nun lesen wir die ersten 100 Klassenbezeichnungen aus, die zu den 50 Iris setosa und den 50 Iris versicolor gehören und konvertieren sie in die beiden ganzzahligen Klassenbezeichnungen 1 (*Versicolor*) und -1 (*Setosa*), die wir einem Vektor  $y$  zuweisen. Die `values`-Methode eines panda-DataFrame-Objekts liefert die entsprechende NumPy-Repräsentation zurück. Auf ähnliche Weise extrahieren wir die erste (*Länge des Kelchblatts*) und die dritte (*Länge des Blütenblatts*) Merkmalspalte der 100 Trainingsdatensätze und weisen sie einer Merkmalsmatrix  $X$  zu, die wir als zweidimensionales Streudiagramm visualisieren:

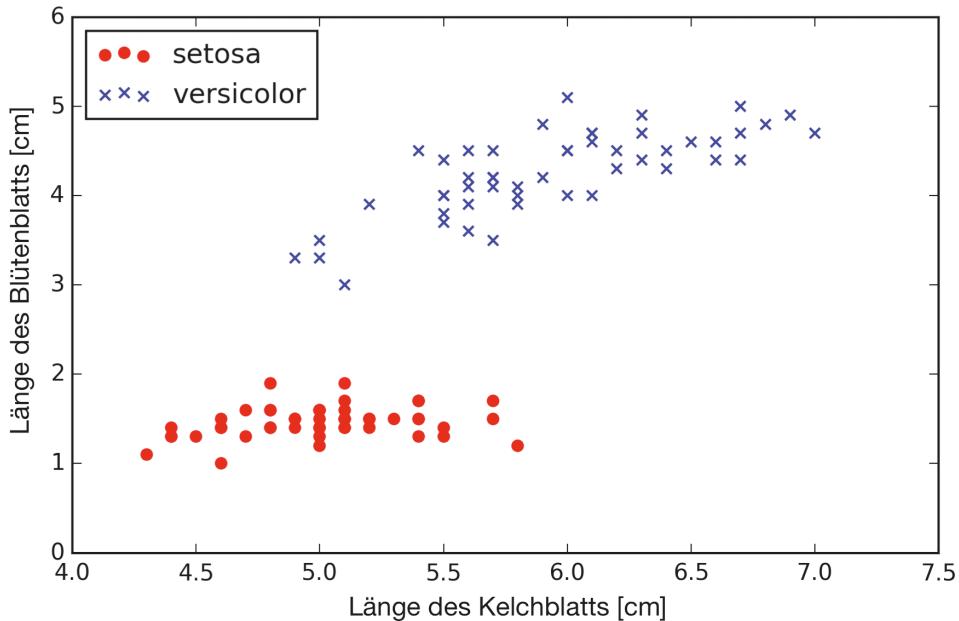
```
>>> import matplotlib.pyplot as plt
>>> import numpy as np

>>> # Auswahl von setosa und versicolor
>>> y = df.iloc[0:100, 4].values
>>> y = np.where(y == 'Iris-setosa', -1, 1)

>>> Auswahl von Kelch- und Blütenblattlänge
>>> X = df.iloc[0:100, [0, 2]].values

>>> # Diagramm ausgeben
>>> plt.scatter(X[:50, 0], X[:50, 1],
...                 color='red', marker='o', label='setosa')
>>> plt.scatter(X[50:100, 0], X[50:100, 1],
...                 color='blue', marker='x', label='versicolor')
>>> plt.xlabel('Länge des Kelchblatts [cm]')
>>> plt.ylabel('Länge des Blütenblatts [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Nach Ausführung des obigen Codes wird folgendes Diagramm angezeigt:

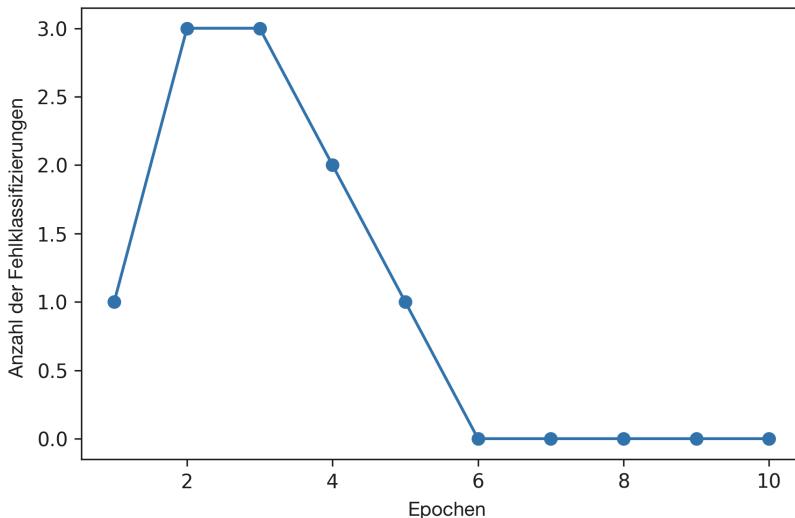


Das Diagramm zeigt die Verteilung der Blumenexemplare der Iris-Datensammlung entlang der beiden Merkmalsachsen Länge des Blütenblatts und Länge des Kelchblatts. Hier ist erkennbar, dass in diesem zweidimensionalen Merkmalsraum eine lineare Entscheidungsgrenze ausreichen sollte, um Setosa und Versicolor vollständig zu klassifizieren.

Nun ist es an der Zeit, den Perzeptron-Algorithmus mit der Teilmenge der Iris-Datensammlung zu trainieren, die wir soeben ausgelesen haben. Außerdem geben wir die Anzahl der Fehlklassifizierungen jeder Epoche aus, um zu prüfen, ob der Algorithmus konvergiert und eine Entscheidungsgrenze findet, die beide Blumenarten voneinander trennt.

```
>>> ppn = Perceptron(eta=0.1, n_iter=10)
>>> ppn.fit(X, y)
>>> plt.plot(range(1, len(ppn.errors_) + 1), ppn.errors_,
...           marker='o')
>>> plt.xlabel('Epochen')
>>> plt.ylabel('Anzahl der Fehlklassifizierungen')
>>> plt.show()
```

Nach der Ausführung des Codes wird ein Diagramm angezeigt, in dem die Anzahl der Fehlklassifizierungen gegen die Anzahl der Epochen aufgetragen ist.



Wie Sie dem Diagramm entnehmen können, konvergiert das Perzeptron bereits nach der sechsten Epoche und sollte nun in der Lage sein, die Trainingsdatenmenge vollständig korrekt zu klassifizieren. Wir implementieren noch eine kleine Hilfsfunktion zur komfortablen Visualisierung der Entscheidungsgrenzen zweidimensionaler Datenmengen.

```
from matplotlib.colors import ListedColormap

def plot_decision_regions(X, y, classifier, resolution=0.02):

    # Markierungen und Farben einstellen
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # Plotten der Entscheidungsgrenze
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, \
                                      resolution), np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), \
                                    xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # Plotten aller Objekte
```

```

for idx, cl in enumerate(np.unique(y)):
    plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                alpha=0.8, c=colors(idx),
                marker=markers[idx], label=cl,
                edgecolor='black')

```

Zunächst definieren wir eine Reihe von Farben und Markierungen und via `ListedColormap` eine Farbtabelle. Dann ermitteln wir die Minimal- und Maximalwerte der beiden Merkmale und verwenden die Merkmalsvektoren, um mit der NumPy-Funktion `meshgrid` die beiden Gitternetz-Arrays `xx1` und `xx2` zu erzeugen. Da der Perzeptron-Klassifizierer nur mit zwei Merkmalen trainiert wurde, müssen wir die Arrays anpassen und eine Matrix mit derselben Spaltenzahl wie in der Iris-Trainingsteilmenge erzeugen, damit wird die `predict`-Methode verwenden können, um die Klassenbezeichnungen `z` den jeweiligen Gitternetzpunkten zuzuordnen.

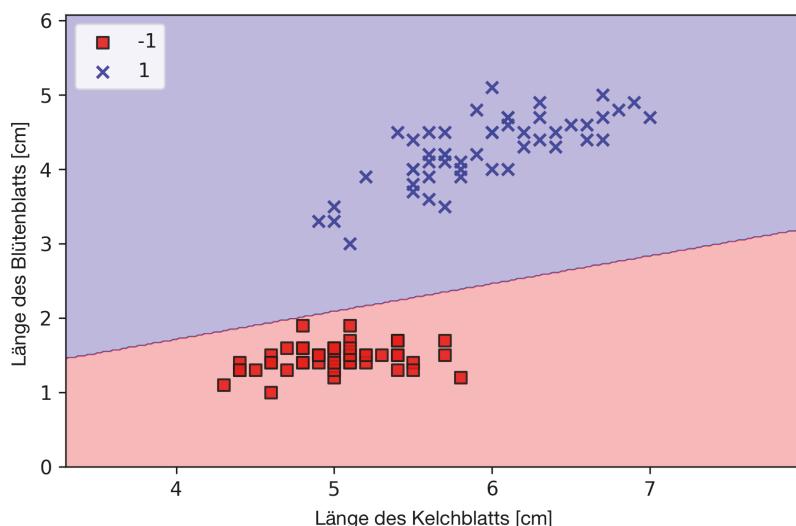
Wenn die vorhergesagten Klassenbezeichnungen in einem Gitternetz mit denselben Dimensionen wie `xx1` und `xx2` gespeichert sind, können wir mit der `matplotlib`-Funktion `contourf` ein Diagramm ausgeben, in dem den Entscheidungsbereichen der in den Gitternetz-Arrays gespeicherten Klassenbezeichnungen verschiedene Farben zugeordnet sind:

```

>>> plot_decision_regions(X, y, classifier=ppn)
>>> plt.xlabel('Länge des Kelchblatts [cm]')
>>> plt.ylabel('Länge des Blütenblatts [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()

```

Die Ausführung des Codes liefert folgendes Ergebnis:



Wie Sie sehen, hat das Perzeptron eine Entscheidungsgrenze ermittelt, durch die alle Objekte in der Iris-Trainingsteilmenge richtig klassifiziert werden.

### Tipp

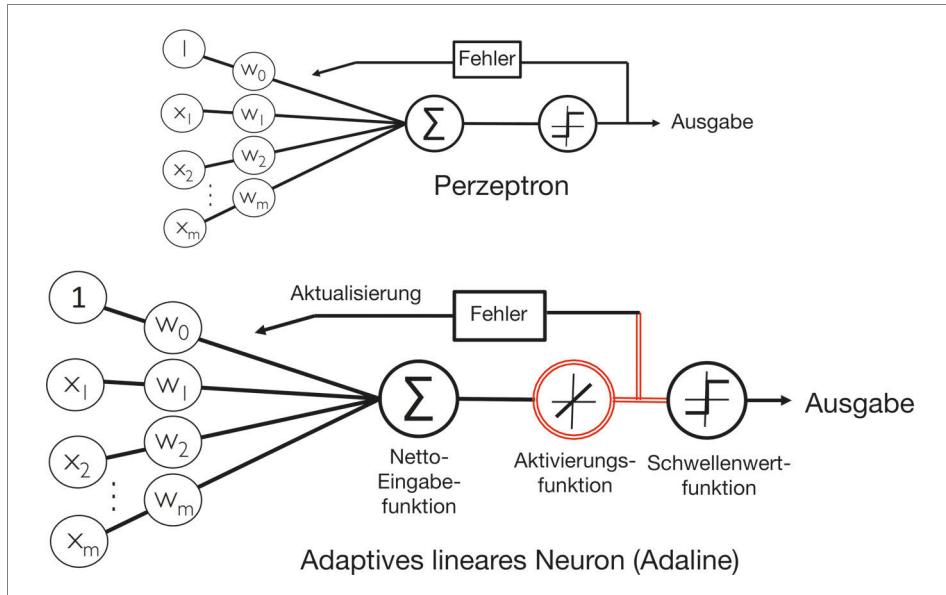
Hier wurden zwar alle Blumen richtig klassifiziert, dennoch stellt die Konvergenz eins der größten Probleme des Perzeptrons dar. Frank Rosenblatt hat den mathematischen Beweis geführt, dass die Perzeptron-Lernregel konvergiert, wenn die beiden Klassen durch eine lineare Hyperebene trennbar sind. Wenn sich die Klassen allerdings nicht perfekt durch solch eine lineare Entscheidungsgrenze trennen lassen, fährt das Perzeptron, sofern man die Anzahl der Epochen nicht begrenzt, endlos mit Aktualisierungen der Gewichtungen fort.

## 2.3 Adaptive lineare Neuronen und die Konvergenz des Lernens

In diesem Abschnitt werden wir uns mit einem weiteren neuronalen Netzwerk befassen, das lediglich aus einer einzigen Schicht besteht: *ADaptive LInear NEuron (Adaline)*. Adaline wurde nur wenige Jahre nach Frank Rosenblatts Perzeptron-Algorithmus von Bernard Widrow und seinem Doktoranden Tedd Hoff veröffentlicht und kann als Verbesserung des Letzteren aufgefasst werden (B. Widrow et al. *Adaptive "Adaline" neuron using chemical "memistors"*. Number Technical Report 1553-2. Stanford Electron. Labs. Stanford, CA, Oktober 1960). Der Adaline-Algorithmus ist besonders interessant, weil er das entscheidende Konzept der Definition und Minimierung von Straffunktionen illustriert, die ihrerseits die Grundlage dafür schaffen, fortgeschrittenere Lernalgorithmen für die Klassifizierung zu verstehen, wie etwa die logistische Regression, *Support Vector Machines (SVM*, Stützvektormethode) und Regressionsmodelle, zu denen wir in nachfolgenden Kapiteln noch kommen werden.

Der wesentliche Unterschied zwischen der Adaline-Regel (die auch als *Widrow-Hoff-Regel* bezeichnet wird) und Rosenblatts Perzeptron ist, dass die Aktualisierung der Gewichtungen nicht wie beim Perzeptron auf einer einfachen Sprungfunktion, sondern auf einer linearen Aktivierungsfunktion beruht. Bei Adaline ist diese Funktion  $\phi(z)$  einfach die identische Abbildung der Nettoeingabefunktion, sodass  $\phi(w^T x) = w^T x$  gilt.

Die lineare Aktivierungsfunktion wird dazu benutzt, die Gewichtungen zu erlernen. Anschließend kann man eine *Schwellenwertfunktion* verwenden, die Ähnlichkeit mit der bekannten Sprungfunktion besitzt, um die Klassenbezeichnungen vorherzusagen. Die folgende Abbildung soll die entscheidenden Unterschiede zwischen dem Perzeptron und dem Adaline-Algorithmus verdeutlichen.



Die Abbildung zeigt, dass der Adaline-Algorithmus die tatsächlichen Klassenbezeichnungen mit den stetigen Ausgaben der linearen Aktivierungsfunktion vergleicht, um Fehler des Modells zu berechnen und die Gewichtungen zu aktualisieren. Im Gegensatz dazu vergleicht das Perzepron die tatsächlichen mit den vorhergesagten Klassenbezeichnungen.

### 2.3.1 Straffunktionen mit dem Gradientenabstiegsverfahren minimieren

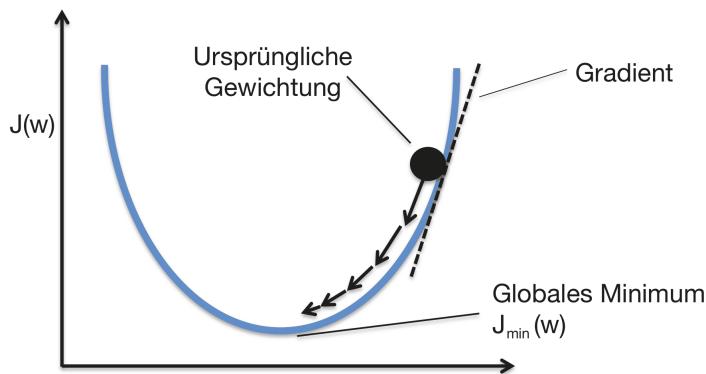
Bei der Entwicklung eines überwachten Lernalgorithmus ist es notwendig, eine Zielfunktion zu definieren, die während des Lernvorgangs optimiert werden muss. Oftmals handelt es sich bei dieser Zielfunktion um eine *Straffunktion*, die minimiert werden soll. Im Fall von Adaline definieren wir die Straffunktion  $J$  als *Summe der quadrierten Abweichungen* zwischen den berechneten Ergebnissen und den tatsächlichen Klassenbezeichnungen:

$$J(\mathbf{w}) = \frac{1}{2} \sum_i (y^{(i)} - \phi(z^{(i)}))^2$$

Der Faktor  $\frac{1}{2}$  ist hier nur der Bequemlichkeit halber vorhanden – wie Sie in Kürze sehen werden, vereinfacht er die Berechnung des Gradienten. Der Hauptvorteil dieser stetigen linearen Aktivierungsfunktion besteht darin, dass die Straffunktion im Gegensatz zur Sprungfunktion differenzierbar ist. Die Straffunktion hat außerdem die angenehme Eigenschaft, konvex zu sein, daher können wir einen einfachen, aber dennoch leistungsfähigen Optimierungsalgorithmus einsetzen,

der als *Gradientenabstiegsverfahren* bezeichnet wird, um die Gewichtungen zu berechnen, die unsere Straffunktion zur Klassifizierung der Objekte in der Iris-Datensammlung minimiert.

Wie die nächste Abbildung zeigt, kann man sich das dem Gradientenabstiegsverfahren zugrunde liegende Prinzip folgendermaßen vorstellen: Man steigt einen Hügel hinab, bis man auf ein lokales oder globales Minimum der Straffunktion stößt. Bei jedem Durchgang entfernt man sich einen Schritt vom Gradienten, wobei die Größe dieses Schrittes sowohl vom Wert der Lernrate als auch von der Steigung des Gradienten abhängt:



Mit dem Gradientenabstiegsverfahren können wir nun die Gewichtungen aktualisieren, indem wir uns einen Schritt vom Gradienten  $\nabla J(\mathbf{w})$  der Straffunktion  $J(\mathbf{w})$  fortbewegen:

$$\mathbf{w} := \mathbf{w} + \Delta\mathbf{w}$$

Die Änderung der Gewichtung  $\Delta\mathbf{w}$  ist definiert als der negative Gradient multipliziert mit der Lernrate  $\eta$ :

$$\Delta\mathbf{w} = -\eta \nabla J(\mathbf{w})$$

Zur Berechnung des Gradienten der Straffunktion benötigen wir die partiellen Ableitungen der Straffunktion nach den Gewichtungen

$$\frac{\partial J}{\partial w_j} = -\sum_i (y^{(i)} - \phi(z^{(i)})) x_j^{(i)}, \text{ sodass wir die Aktualisierung der Gewichtung } w_j \text{ als}$$

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i (y^{(i)} - \phi(z^{(i)})) x_j^{(i)}$$

schreiben können. Da wir alle Gewichtungen gleichzeitig aktualisieren, wird die Adaline-Lernregel zu  $\mathbf{w} := \mathbf{w} + \Delta\mathbf{w}$ .

**Tipp**

Die partiellen Ableitungen der Straffunktion nach der j-ten Gewichtung können wie folgt berechnet werden:

$$\begin{aligned}
 \frac{\partial J}{\partial w_j} &= \frac{\partial}{\partial w_j} \frac{1}{2} \sum_i \left( y^{(i)} - \phi(z^{(i)}) \right)^2 \\
 &= \frac{1}{2} \frac{\partial}{\partial w_j} \sum_i \left( y^{(i)} - \phi(z^{(i)}) \right)^2 \\
 &= \frac{1}{2} \sum_i 2 \left( y^{(i)} - \phi(z^{(i)}) \right) \frac{\partial}{\partial w_j} \left( y^{(i)} - \phi(z^{(i)}) \right) \\
 &= \sum_i \left( y^{(i)} - \phi(z^{(i)}) \right) \frac{\partial}{\partial w_j} \left( y^{(i)} - \sum_k (w_k x_k^{(i)}) \right) \\
 &= \sum_i \left( y^{(i)} - \phi(z^{(i)}) \right) (-x_j^{(i)}) \\
 &= -\sum_i \left( y^{(i)} - \phi(z^{(i)}) \right) x_j^{(i)}
 \end{aligned}$$

Die Adaline-Lernregel sieht zwar genauso aus wie die Perzepron-Lernregel, allerdings ist  $\phi(z^{(i)})$  mit  $z^{(i)} = \mathbf{w}^T \mathbf{x}^{(i)}$  eine reelle Zahl und keine ganzzahlige Klassenbezeichnung. Darüber hinaus beruht die Berechnung der Gewichtungsaktualisierung auf *allen* in der Trainingsdatenmenge enthaltenen Objekten, anstatt die Gewichtungen nach jedem Objekt inkrementell zu aktualisieren. Man spricht daher auch von einem Gradientenabstiegsverfahren als *Stapelverarbeitung*.

### 2.3.2 Implementierung eines adaptiven linearen Neurons in Python

Da die Perzepron- und Adaline-Regeln einander sehr ähneln, verwenden wir die eingangs dieses Kapitels definierte Perzepron-Implementierung und ändern hier die `fit`-Methode, sodass die Aktualisierung der Gewichtungen stattfindet, indem die Straffunktion mit dem Gradientenabstiegsverfahren minimiert wird:

```

class AdalineGD(object):
    """Adaline-Klassifizierer

    Parameter
    -----
    """

```

## Kapitel 2

### Lernalgorithmen für die Klassifizierung trainieren

```
eta : float
    Lernrate (zwischen 0.0 und 1.0)
n_iter : int
    Durchläufe der Trainingsdatenmenge
random_state : int
    Zufallszahlgenerator für zufällige Gewichtungen
    initialisieren

Attribute
-----
w_ : 1d-array
    Gewichtung nach Anpassung
cost_ : list
    Summe der quadrierten Werte der
    Straffunktion pro Epoche

"""
def __init__(self, eta=0.01, n_iter=50, random_state=1):
    self.eta = eta
    self.n_iter = n_iter
    self.random_state = random_state

def fit(self, X, y):
    """ Fit-Trainingsdaten

Parameter
-----
X : {array-like}, shape = [n_samples, n_features]
    Trainingsvektoren, n_samples ist
    die Anzahl der Exemplare und
    n_features ist die Anzahl der Merkmale
y : array-like, shape = [n_samples]
    Zielwerte

Rückgabewert
-----
self : object

"""
rgen = np.random.RandomState(self.random_state)
self.w_ = rgen.normal(loc=0.0, scale=0.01,
                      size=1 + X.shape[1])
self.cost_ = []
```

```

for i in range(self.n_iter):
    net_input = self.net_input(X)
    output = self.activation(net_input)
    errors = (y - output)
    self.w_[1:] += self.eta * X.T.dot(errors)
    self.w_[0] += self.eta * errors.sum()
    cost = (errors**2).sum() / 2.0
    self.cost_.append(cost)
return self

def net_input(self, X):
    """Nettoeingabe berechnen"""
    return np.dot(X, self.w_[1:]) + self.w_[0]

def activation(self, X):
    """Lineare Aktivierungsfunktion berechnen"""
    return X

def predict(self, X):
    """Rückgabe der Klassenbezeichnung"""
    return np.where(self.activation(self.net_input(X))
                    >= 0.0, 1, -1)

```

Anstatt wie beim Perzepron die Gewichtungen nach der Bewertung jedes einzelnen Trainingsexemplars zu aktualisieren, berechnen wir den Gradienten anhand der gesamten Trainingsdatenmenge mittels `self.eta * errors.sum()` für die Bias-Einheit (Gewichtung 0) und `self.eta * X.T.dot(errors)` für die Gewichtungen 1 bis  $m$ . Dabei ist `X.T.dot(errors)` eine *Matrix-Vektor-Multiplikation* von Merkmalsmatrix und Fehlervektor.

Beachten Sie hier, dass die `activation`-Methode im Code nichts bewirkt, denn es handelt sich einfach nur um eine Identitätsfunktion. Wir addieren die (durch die `activation`-Methode berechnete) Aktivierungsfunktion, um zu demonstrieren, wie Informationen durch ein einschichtiges neuronales Netzwerk fließen: Merkmale der Eingabedaten, Nettoeingabe, Aktivierungsfunktion und Ausgabe. Im nächsten Kapitel werden wir uns mit auf logistischer Regression beruhenden Klassifizierern befassen und feststellen, dass diese eng mit Adaline verwandt sind, denn sie unterscheiden sich nur durch die Aktivierungs- und Straffunktion.

Wie beim Perzepron sammeln wir in einer Liste `self.cost_` die auftretenden Werte der Straffunktion, um zu prüfen, ob der Algorithmus nach dem Training konvergiert.

**Tipp**

Die Multiplikation einer Matrix mit einem Vektor funktioniert auf ganz ähnliche Weise wie die Berechnung eines Skalarprodukts, bei der mit jeder Zeile der Matrix wie mit einem einzelnen Zeilenvektor verfahren wird. Dieser vektorisierte Ansatz ermöglicht eine kompaktere Schreibweise und führt bei der Verwendung von NumPy zu effizienteren Berechnungen. Ein Beispiel:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix} = \begin{bmatrix} 1 \times 7 + 2 \times 8 + 3 \times 9 \\ 4 \times 7 + 5 \times 8 + 6 \times 9 \end{bmatrix} = \begin{bmatrix} 50 \\ 122 \end{bmatrix}$$

In der Praxis ist oft ein wenig Herumexperimentieren notwendig, um einen guten Wert für die Lernrate  $\eta$  zu finden, der zu optimaler Konvergenz führt. Wir wählen daher für den Anfang zwei verschiedene Lernraten  $\eta = 0.01$  und  $\eta = 0.0001$  aus und tragen die Werte der Straffunktion gegen die Anzahl der Epochen auf, um zu sehen, wie gut die Adaline-Implementierung aus den Trainingsdaten lernt.

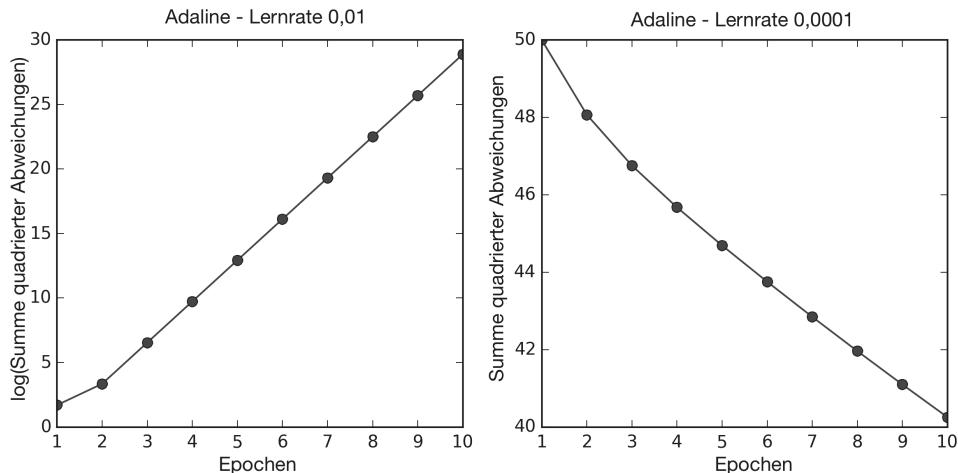
**Tipp**

Die Lernrate  $\eta$  und die Anzahl der Epochen `n_iter` sind sogenannte *Hyperparameter* der Perzepron- und Adaline-Lernalgorithmen. In Kapitel 6, *Bewährte Verfahren zur Modellbewertung und Hyperparameter-Abstimmung*, werden wir uns einige Verfahren ansehen, um automatisch die Werte verschiedener Hyperparameter zu finden, die zu einer optimalen Leistung des Klassifizierungsmodells führen.

Nun tragen wir für beide Lernraten die Werte der Straffunktion gegen die Anzahl der Epochen auf:

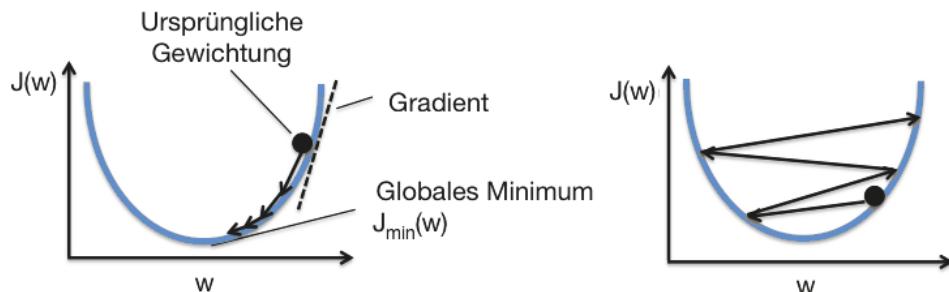
```
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))
>>> ada1 = AdalineGD(n_iter=10, eta=0.01).fit(X, y)
>>> ax[0].plot(range(1, len(ada1.cost_) + 1),
...             np.log10(ada1.cost_), marker='o')
>>> ax[0].set_xlabel('Epochen')
>>> ax[0].set_ylabel('log(Summe quadrierter Abweichungen)')
>>> ax[0].set_title('Adaline - Lernrate 0.01')
>>> ada2 = AdalineGD(n_iter=10, eta=0.0001).fit(X, y)
>>> ax[1].plot(range(1, len(ada2.cost_) + 1),
...             ada2.cost_, marker='o')
>>> ax[1].set_xlabel('Epochen')
>>> ax[1].set_ylabel('Summe quadrierter Abweichungen')
>>> ax[1].set_title('Adaline - Lernrate 0.0001')
>>> plt.show()
```

Wie Sie den Diagrammen entnehmen können, werden wir mit zwei verschiedenen Arten von Problemen konfrontiert. Das linke Diagramm zeigt, was passieren kann, wenn die Lernrate zu groß ist – die Werte der Straffunktion werden nicht minimiert, sondern wachsen sogar mit jeder Epoche weiter an, weil wir über das globale Minimum hinausgeschossen sind.



Beim rechten Diagramm ist zwar erkennbar, dass die Werte der Straffunktion sinken, allerdings ist die ausgewählte Lernrate  $\eta = 0.0001$  so klein, dass der Algorithmus sehr viele Epochen benötigen würde, um zu konvergieren.

Die nachstehende Abbildung illustriert, wie man den Wert eines bestimmten Gewichtungsparameters variieren kann, um die Straffunktion  $J$  zu minimieren. Auf der linken Seite der Abbildung wurde eine geeignete Lernrate ausgewählt: Die Straffunktion wird allmählich kleiner und bewegt sich in Richtung des globalen Minimums. Die rechte Seite der Abbildung demonstriert, was geschieht, wenn man die Lernrate zu groß wählt und über das Minimum hinausschießt.



### 2.3.3 Verbesserung des Gradientenabstiegsverfahrens durch Merkmalstandardisierung

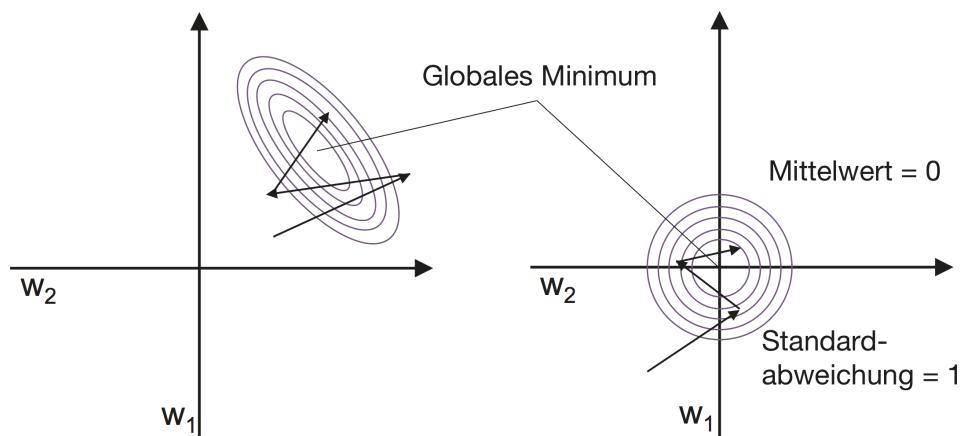
Viele der Lernalgorithmen, denen Sie im weiteren Verlauf des Buches noch begegnen werden, machen es erforderlich, die Merkmale irgendwie zu standardisieren, um eine optimale Leistung zu erzielen. Mehr dazu in Kapitel 3, *Machine-Learning-Klassifizierer mit scikit-learn verwenden* und Kapitel 4, *Gut geeignete Trainingsdatensammlungen: Datenvorverarbeitung*.

Das Gradientenabstiegsverfahren gehört zu den vielen Algorithmen, die von solch einer Standardisierung profitieren. Wir werden nun eine als *Standardisierung* bezeichnete Methode verwenden, die den Daten die Eigenschaften einer Standardnormalverteilung verleiht. Der Mittelwert jedes Merkmals liegt bei 0 und die Standardabweichung jeder Spalte beträgt 1. Um beispielsweise das Merkmal  $j$  zu standardisieren, müssen wir nur den Mittelwert  $\mu_j$  von der jeweiligen Stichprobe abziehen und das Ergebnis durch die Standardabweichung  $\sigma_j$  teilen:

$$\mathbf{x}'_j = \frac{\mathbf{x}_j - \mu_j}{\sigma_j}$$

$\mathbf{x}_j$  ist hier ein Vektor, dessen Elemente die  $j$ -ten Merkmalswerte aller Trainingsobjekte  $n$  sind. Dieses Standardisierungsverfahren wird auf alle Merkmale der Datenmenge angewendet.

Die Standardisierung verbessert das Gradientenabstiegsverfahren unter anderem, weil zum Auffinden einer guten oder optimalen Lösung (das globale Minimum der Straffunktion) weniger Schritte erforderlich sind, wie die nachstehende Abbildung verdeutlicht. Beide Diagramme stellen die Straf-»Fläche« einer zweidimensionalen Klassifizierungsaufgabe als Funktion der Gewichtung dar.



Mit den NumPy-Methoden `mean` und `std` kann diese Standardisierung leicht erzielt werden:

```
>>> X_std = np.copy(X)
>>> X_std[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
>>> X_std[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
```

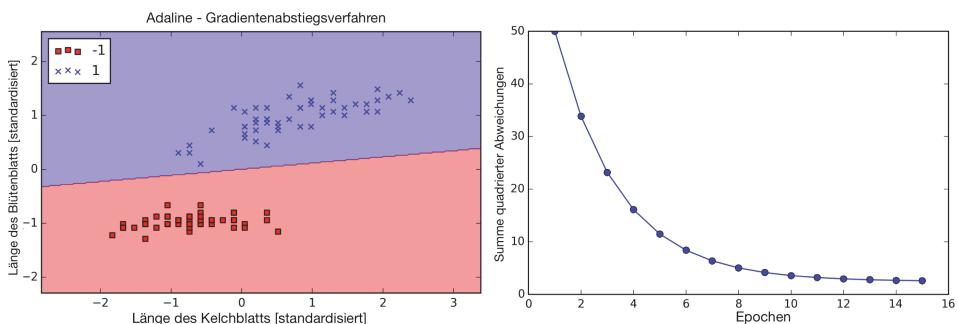
Nach der Standardisierung trainieren wir den Adaline-Algorithmus erneut und stellen fest, dass er nun mit einer Lernrate von  $\eta = 0.01$  konvergiert:

```
>>> ada = AdalineGD(n_iter=15, eta=0.01)
>>> ada.fit(X_std, y)

>>> plot_decision_regions(X_std, y, classifier=ada)
>>> plt.title('Adaline - Gradientenverfahren')
>>> plt.xlabel('Länge des Kelchblatts [standardisiert]')
>>> plt.ylabel('Länge des Blütenblatts [standardisiert]')
>>> plt.legend(loc='upper left')
>>> plt.show()

>>> plt.plot(range(1, len(ada.cost_) + 1),
...           ada.cost_, marker='o')
>>> plt.xlabel('Epochen')
>>> plt.ylabel('Summe quadrierter Abweichungen')
>>> plt.show()
```

Nach der Ausführung des Codes sollten die Entscheidungsbereiche sowie die sinkenden Werte der Straffunktion in zwei Diagrammen angezeigt werden:



Den Diagrammen ist zu entnehmen, dass der Adaline-Algorithmus nach dem Training mit den standardisierten Merkmalen und einer Lernrate von  $\eta = 0.01$  konvergiert. Beachten Sie jedoch, dass die Summe der quadrierten Abweichungen von null verschieden ist, obwohl alle Objekte korrekt klassifiziert wurden.

### 2.3.4 Großmaßstäbliches Machine Learning und stochastisches Gradientenabstiegsverfahren

Im vorangegangenen Abschnitt haben Sie erfahren, wie man die Straffunktion minimiert, indem man sich einen Schritt in die entgegengesetzte Richtung des Gradienten der gesamten Trainingsdatenmenge begibt (Gradientenabstiegsverfahren als Stapelverarbeitung). Nehmen wir nun an, wir hätten es mit einer sehr großen Datenmenge mit Millionen von Datensätzen zu tun – bei vielen Anwendungen des Machine Learnings ist das nicht ungewöhnlich. In solchen Fällen kann das Gradientenabstiegsverfahren als Stapelverarbeitung ziemlich rechenaufwendig werden, weil die *gesamte* Trainingsdatenmenge bei jedem Schritt in Richtung des globalen Minimums erneut ausgewertet werden muss.

Als Alternative bietet sich ein *stochastisches Gradientenabstiegsverfahren* an, das manchmal auch als *iteratives* oder *Online-Gradientenabstiegsverfahren* bezeichnet wird. Normalerweise werden die Gewichtungen anhand der Summe aller Abweichungen sämtlicher Objekte  $x^{(i)}$  aktualisiert:

$$\Delta w = \eta \sum_i (y^{(i)} - \phi(z^{(i)})) x^{(i)}$$

Stattdessen aktualisieren wir die Gewichtungen hier inkrementell für jedes einzelne Trainingsobjekt:

$$\eta (y^{(i)} - \phi(z^{(i)})) x^{(i)}$$

Das stochastische Gradientenabstiegsverfahren muss zwar als Näherung des normalen Gradientenabstiegsverfahrens aufgefasst werden, konvergiert allerdings aufgrund der häufigeren Aktualisierungen der Gewichtungen sehr viel schneller. Da der Gradient für jedes Trainingsexemplar einzeln berechnet wird, kommt es zu einem höheren statistischen Rauschen, was den Vorteil haben kann, dass sich das stochastische Gradientenabstiegsverfahren leichter von lokalen Minima lösen kann, wenn man es mit nichtlinearen Straffunktionen zu tun hat, wie wir in Kapitel 12 (*Implementierung eines künstlichen neuronalen Netzes*) noch sehen werden. Es ist wichtig, dem stochastischen Gradientenabstiegsverfahren die Daten in zufälliger Reihenfolge bereitzustellen, um brauchbare Ergebnisse zu erzielen, daher werden die Trainingsdaten für jede Epoche durchgemischt, um zu verhindern, dass es zu Wiederholungen kommt und sich das Verfahren im Kreis dreht.

#### Tipp

Bei der Implementierung des stochastischen Gradientenabstiegsverfahrens wird die konstante Lernrate  $\eta$  häufig durch eine adaptive Lernrate ersetzt, die im Laufe der Zeit sinkt, beispielsweise durch  $\frac{c_1}{[Anzahl\ der\ Iterationen] + c_2}$  mit den Konstanten  $c_1$  und  $c_2$ .

Beachten Sie, dass das stochastische Gradientenabstiegsverfahren nicht das globale Minimum erreicht, ihm aber sehr nahe kommt. Durch die adaptive Lernrate können wir uns dem globalen Minimum noch weiter annähern.

Ein weiterer Vorteil des stochastischen Gradientenabstiegsverfahrens besteht darin, dass es für das *Online Learning* geeignet ist. Hierbei wird das Modell in Echtzeit trainiert, wenn neue Daten eintreffen. Das erweist sich als besonders nützlich, wenn man es mit großen Datenmengen zu tun hat, beispielsweise mit Kundendaten in einer typischen Webanwendung. Beim Online Learning kann das System sofort auf Änderungen reagieren, und die Trainingsdaten können nach der Aktualisierung des Modells entfernt werden, sofern der Speicherplatz knapp wird.

### Tipp

Es gibt einen Mittelweg zwischen dem Gradientenabstiegsverfahren als Stapelverarbeitung und dem stochastischen Gradientenabstiegsverfahren: das sogenannte *Mini-Batch Learning* (Lernen als Mini-Stapelverarbeitung). Dabei wird das Gradientenabstiegsverfahren auf kleinere Teilmengen der Trainingsdaten angewendet, beispielsweise auf jeweils 32 Objekte. Der Vorteil gegenüber der reinen Stapelverarbeitung besteht darin, dass die Konvergenz mit kleineren Teilmengen aufgrund der häufiger stattfindenden Aktualisierung der Gewichtungen schneller erzielt wird. Darüber hinaus kann die `for`-Schleife zum Durchlaufen der Trainingsdatenmenge beim *stochastischen Gradientenabstiegsverfahren (SGD, Stochastic Gradient Descent)* durch vektorisierte Operationen ersetzt werden, was die Effizienz der Berechnungen der Lernalgorithmen verbessert.

Da wir die auf dem Gradientenabstiegsverfahren beruhende Adaline-Lernregel bereits implementiert haben, müssen nur einige wenige Anpassungen vorgenommen werden, um die Gewichtungen stattdessen durch ein stochastisches Gradientenabstiegsverfahren zu aktualisieren. In der `fit`-Methode werden nun die Gewichtungen nach jedem Trainingsexemplar aktualisiert. Für das Online Learning wird außerdem eine zusätzliche `partial_fit`-Methode ohne Neuinitialisierung der Gewichtungen implementiert.

Um zu prüfen, ob der Algorithmus nach dem Training konvergiert, berechnen wir die Straffunktion als Mittelwert der Straffunktionen der Trainingsdatenmenge in den einzelnen Epochen. Schließlich fügen wir mit der `shuffle`-Methode noch eine Option zum Durchmischen der Trainingsdaten hinzu, um bei der Optimierung der Straffunktion Wiederholungen zu verhindern. Mit dem Parameter `random_state` kann ein Anfangswert für den Zufallsgenerator festgelegt werden.

## Kapitel 2

### Lernalgorithmen für die Klassifizierung trainieren

```
class AdalineSGD(object):
    """Adaline-Klassifizierer
    Parameter
    -----
    eta : float
        Lernrate (zwischen 0.0 und 1.0)
    n_iter : int
        Durchläufe der Trainingsdatenmenge
    shuffle : bool (default: True)
        Durchmischt die Trainingsdaten nach jeder
        Epoche, falls True, um Wiederholungen zu verhindern.
    random_state : int
        Zufallszahlgenerator für zufällige Gewichtungen
        initialisieren

    Attribute
    -----
    w_ : 1d-array
        Gewichtung nach Anpassung
    cost_ : list
        Mittelwert der Summe der quadrierten Werte der
        Straffunktion aller Trainingsdaten pro Epoche
    shuffle : bool (default: True)
        Falls True, werden die Trainingsdaten vor jeder neuen
        Epoche durchgemischt, um Wiederholungen zu verhindern
    random_state : int (default: None)
        Anfangswert für den Zufallsgenerator setzen
        (Durchmischen/Initialisierung der Gewichtungen)

    """
    def __init__(self, eta=0.01, n_iter=10,
                 shuffle=True, random_state=None):
        self.eta = eta
        self.n_iter = n_iter
        self.w_initialized = False
        self.shuffle = shuffle
        self.random_state = random_state

    def fit(self, X, y):
        """ Fit-Trainingsdaten

        Parameter
        -----
        X : {array-like}, shape = [n_samples, n_features]
            Trainingsvektoren, n_samples ist
```

```
        die Anzahl der Exemplare und
        n_features ist die Anzahl der Merkmale
y : array-like, shape = [n_samples]
    Zielwerte

Rückgabewert
-----
self : object

"""
self._initialize_weights(X.shape[1])
self.cost_ = []
for i in range(self.n_iter):
    if self.shuffle:
        X, y = self._shuffle(X, y)
    cost = []
    for xi, target in zip(X, y):
        cost.append(self._update_weights(xi, target))
    avg_cost = sum(cost)/len(y)
    self.cost_.append(avg_cost)
return self

def partial_fit(self, X, y):
    """Anpassung an die Trainingsdaten ohne \
       Reinitialisierung der Gewichtungen"""
    if not self.w_initialized:
        self._initialize_weights(X.shape[1])
    if y.ravel().shape[0] > 1:
        for xi, target in zip(X, y):
            self._update_weights(xi, target)
    else:
        self._update_weights(X, y)
    return self

def _shuffle(self, X, y):
    """Trainingsdaten durchmischen"""
    r = np.random.permutation(len(y))
    return X[r], y[r]

def _initialize_weights(self, m):
    """Gewichtungen mit kleinen Zufallszahlen
       initialisieren"""
    self.w_ = np.random.RandomState(self.random_state)
```

## Kapitel 2

### Lernalgorithmen für die Klassifizierung trainieren

```
        self.w_ = self.rgen.normal(loc=0.0, scale=0.01,
                                size=1 + m)
        self.w_initialized = True

    def _update_weights(self, xi, target):
        """Adaline-Lernregel zur Aktualisierung \
            der Gewichtungen"""
        output = self.activation(self.net_input(xi))
        error = (target - output)
        self.w_[1:] += self.eta * xi.dot(error)
        self.w_[0] += self.eta * error
        cost = 0.5 * error**2
        return cost

    def net_input(self, X):
        """Nettoeingabe berechnen"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, X):
        """Lineare Aktivierung berechnen"""
        return X

    def predict(self, X):
        """Rückgabe der Klassenbezeichnung"""
        return np.where(self.activation(self.net_input(X))
                       >= 0.0, 1, -1)
```

Die `shuffle`-Methode, die wir nun im `AdalineSGD`-Klassifizierer verwenden, funktioniert folgendermaßen: Mit der `permutation`-Funktion von `np.random` erzeugen wir eine zufällige Sequenz eindeutiger Zufallszahlen zwischen 0 und 100. Diese Zahlen werden dann als Indizes verwendet, um die Merkmalsmatrix und den Vektor der Klassenbezeichnungen durchzumischen.

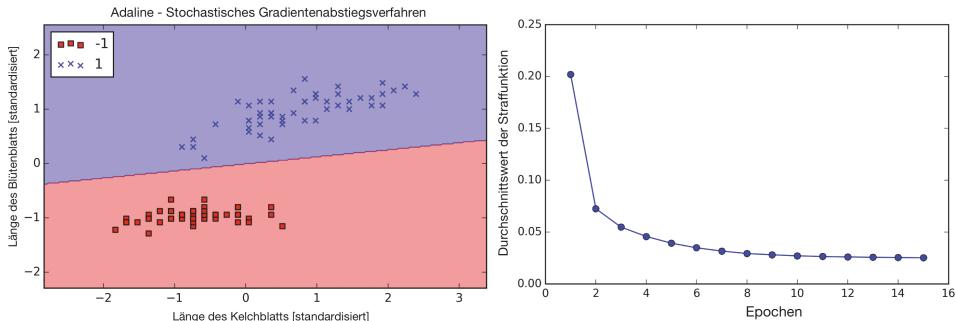
Nun können wir die `fit`-Methode benutzen, um den `AdalineSGD`-Klassifizierer zu trainieren, und die `plot_decision_region`-Methode verwenden, um das Trainingsergebnis darzustellen:

```
>>> ada = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
>>> ada.fit(X_std, y)

>>> plot_decision_regions(X_std, y, classifier=ada)
>>> plt.title('Adaline – Stochastisches
...          Gradientenabstiegsverfahren')
>>> plt.xlabel('Länge des Kelchblatts [standardisiert]')
>>> plt.ylabel('Länge des Blütenblatts [standardisiert]')
```

```
>>> plt.legend(loc='upper left')
>>> plt.show()
>>> plt.plot(range(1, len(ada.cost_) + 1),
...           ada.cost_, marker='o')
>>> plt.xlabel('Epochen')
>>> plt.ylabel('Durchschnittswert der Straffunktion')
>>> plt.show()
```

Die Ausführung dieses Codes liefert die beiden nachfolgenden Diagramme:



Wie Sie sehen, sinkt der Wert der Straffunktion ziemlich schnell, und die Entscheidungsgrenze nach 15 Epochen ist derjenigen des Gradientenabstiegsverfahrens als Stapelverarbeitung sehr ähnlich. Wenn wir dieses Modell aktualisieren möchten, beispielsweise um das Online Learning mit Streamingdaten zu ermöglichen, könnten wir die `partial_fit`-Methode einfach mit ausgewählten Datensätzen aufrufen, z.B. `ada.partial_fit(X_std[0, :], y[0])`.

## 2.4 Zusammenfassung

In diesem Kapitel haben Sie die grundlegenden Konzepte linearer Klassifizierer bei überwachtem Lernen kennengelernt. Nach der Implementierung eines Perzeptrons haben Sie erfahren, wie man adaptive lineare Neuronen durch eine vektorisierte Implementierung des Gradientenabstiegsverfahrens trainiert und wie sich Online Learning durch ein stochastisches Gradientenabstiegsverfahren realisieren lässt.

Nun wissen Sie, wie einfache Klassifizierer in Python implementiert werden. Im nächsten Kapitel werden wir die Python-Bibliothek scikit-learn verwenden, um fortgeschrittenere und leistungsfähigere Klassifizierer einzusetzen, die sowohl in der Forschung als auch in der Industrie gebräuchlich sind. Der objektorientierte Ansatz, den wir bei der Implementierung von Perzeptron- und Adaline-Algorithmus verfolgt haben, hilft beim Verständnis der scikit-learn-API, deren Implementierung auf einigen der in diesem Kapitel verwendeten grundlegenden Konzepte

## Kapitel 2

### Lernalgorithmen für die Klassifizierung trainieren

beruht: den Methoden `fit` und `predict`. Anhand dieser Konzepte werden wir uns mit der logistischen Regression zum Modellieren von Klassenwahrscheinlichkeiten und Support Vector Machines zur Handhabung nichtlinearer Entscheidungsgrenzen befassen. Darüber hinaus wird eine weitere Art überwachter Lernalgorithmen vorgestellt, nämlich baumbasierte Algorithmen, die häufig zu stabilen Klassifizierer-Ensembles miteinander kombiniert werden.

# Machine-Learning-Klassifizierer mit scikit-learn verwenden

In diesem Kapitel werden wir uns eine Auswahl verbreiteter und leistungsfähiger Lernalgorithmen ansehen, die sowohl in der Forschung als auch in der Industrie gebräuchlich sind. Beim Kennenlernen der Unterschiede zwischen den überwachten Klassifizierungsalgorithmen werden Sie auch ein gewisses Gespür für deren individuelle Stärken und Schwächen entwickeln. Darüber hinaus werden wir die ersten Schritte mit der scikit-learn-Bibliothek unternehmen, die eine benutzerfreundliche Schnittstelle für die effiziente und produktive Verwendung dieser Algorithmen bietet.

Die Themen in diesem Kapitel:

- Einführung in die Konzepte stabiler und verbreiteter Klassifizierungsalgorithmen, wie logistische Regression, Support Vector Machines und Entscheidungsbäume.
- Beispiele und Erläuterungen zur Verwendung der scikit-learn-Bibliothek für das Machine Learning, die ein breites Spektrum an Machine-Learning-Algorithmen mit benutzerfreundlicher Python-API bietet.
- Stärken und Schwächen von Klassifizierern mit linearen und nichtlinearen Entscheidungsgrenzen.

## 3.1 Auswahl eines Klassifizierungsalgorithmus

Man braucht Erfahrung, um einen für eine bestimmte Aufgabe geeigneten Algorithmus auszuwählen. Jeder Algorithmus besitzt gewisse Eigenheiten und geht von bestimmten Annahmen aus. Oder um das »No Free Lunch«-Theorem von David H. Wolpert umzuformulieren: Es gibt keinen Klassifizierer, der für alle denkbaren Szenarien geeignet wäre. (*The Lack of A Priori Distinctions Between Learning Algorithms*, Wolpert, David H., *Neural Computation* 8.7 (1996): 1341–1390). In der Praxis ist es immer empfehlenswert, die Leistung zumindest einer Handvoll verschiedener Lernalgorithmen miteinander zu vergleichen, um das beste Modell für eine gegebene Aufgabe auszuwählen. Welche Algorithmen infrage kommen, hängt davon ab, wie viele Merkmale es gibt, wie stark das »Rauschen« in der Datensammlung ist und ob die Klassen linear trennbar sind oder nicht.

Letzten Endes ist sowohl der Bedarf an Rechenleistung als auch die Vorhersagekraft eines Klassifizierers stark von den zum Lernen verfügbaren Daten abhängig. Die fünf zum Trainieren eines Lernalgorithmus erforderlichen Schritte lassen sich folgendermaßen zusammenfassen:

1. Auswahl der Merkmale und Sammeln von Trainingsdaten
2. Kriterien für die Leistungsbewertung festlegen
3. Auswahl eines Klassifizierers und eines Optimierungsalgorithmus
4. Bewertung der Leistung des Modells
5. Feinabstimmung des Algorithmus

Der Ansatz dieses Buches ist ein schrittweiser Aufbau der Kenntnisse über das Machine Learning, daher konzentrieren wir uns in diesem Kapitel auf die wesentlichen Konzepte der verschiedenen Algorithmen und werden uns mit einigen der gestreiften Themen wie Merkmalsauswahl, Vorverarbeitung, Leistungskriterien und Feinabstimmung der Hyperparameter erst im weiteren Verlauf des Buches befassen.

## 3.2 Erste Schritte mit scikit-learn: Trainieren eines Perzeptrons

In Kapitel 2, *Lernalgorithmen für die Klassifizierung trainieren*, haben Sie zwei miteinander verwandte Klassifizierungsalgorithmen kennengelernt, die wir selbst in Python implementiert haben: Perzepron und Adaline. Nun wollen wir uns die *scikit-learn-API (Application Programming Interface)* ansehen, die eine benutzerfreundliche Schnittstelle mit hochoptimierten Implementierungen verschiedener Klassifizierungsalgorithmen vereint. Allerdings bietet die scikit-learn-Bibliothek nicht nur eine große Vielfalt von verschiedenen Lernalgorithmen, sondern darüber hinaus auch viele komfortable Funktionen zur Datenvorverarbeitung sowie zur Feinabstimmung und Bewertung von Modellen. In den Kapiteln 4 (*Gut geeignete Trainingsdatenmengen: Datenvorverarbeitung*) und 5 (*Datenkomprimierung durch Dimensionsreduktion*) werden wir noch genauer darauf eingehen.

Als Erstes werden wir die scikit-learn-Bibliothek dazu benutzen, ein Perzepron-Modell zu trainieren, das dem in Kapitel 2 implementierten ähnelt. Der Einfachheit halber verwenden wir in den kommenden Abschnitten wieder die nun schon vertraute Iris-Datensammlung. Sie ist per scikit-learn bereits verfügbar, denn es handelt sich dabei um eine zwar einfache, aber weit verbreitete Datensammlung, die häufig dazu eingesetzt wird, Algorithmen zu testen und mit ihnen zu experimentieren. Zwecks Visualisierung werden wir nur zwei Merkmale der Iris-Datensammlung verwenden.

Der Merkmalsmatrix X weisen wir die Länge und Breite der Blütenblätter der 150 Blumenexemplare zu. Die zugehörigen Klassenbezeichnungen der Blumenarten werden im Vektor y gespeichert:

```
>>> from sklearn import datasets  
>>> import numpy as np  
  
>>> iris = datasets.load_iris()  
>>> X = iris.data[:, [2, 3]]  
>>> y = iris.target  
>>> print('Klassenbezeichnungen:', np.unique(y))  
Klassenbezeichnungen: [0 1 2]
```

Wenn man `np.unique(y)` ausführt, um die in `iris.target` gespeicherten Klassenbezeichnungen auszugeben, stellt man fest, dass die Bezeichnungen *Iris setosa*, *Iris versicolor* und *Iris virginica* bereits als Ganzzahlen (0, 1, 2) gespeichert sind. Viele scikit-learn-Funktionen und Klassenmethoden kommen zwar auch mit Klassenbezeichnungen im Stringformat zurecht, es empfiehlt sich jedoch, Ganzzahlen als Bezeichner zu verwenden, um technische Probleme zu vermeiden und aufgrund des geringeren Speicherbedarfs eine höhere Verarbeitungsgeschwindigkeit zu erzielen. Zudem ist es eine gängige Konvention der meisten Machine-Learning-Bibliotheken, Ganzzahlen als Klassenbezeichnungen zu verwenden.

Um beurteilen zu können, wie gut ein trainiertes Modell mit unbekannten Daten zurechtkommt, teilen wir die Datensammlung in eine Trainingsdatenmenge und eine Testdatenmenge auf. In Kapitel 6 (*Best Practices zur Modellbewertung und Hyperparameter-Abstimmung*) werden wir Best Practices zur Bewertung von Modellen eingehender betrachten.

```
>>> from sklearn.model_selection import train_test_split  
>>> X_train, X_test, y_train, y_test = train_test_split(  
...           X, y, test_size=0.3, random_state=0)
```

Mit der `train_test_split`-Funktion des zu scikit-learn gehörenden `model_selection`-Moduls teilen wir den Datenbestand in 30 Prozent Testdaten (45 Objekte) und 70 Prozent Trainingsdaten (105 Objekte) auf.

Beachten Sie hier, dass die `train_test_split`-Funktion die Trainingsdatenmengen bereits vor der Aufteilung durchmischt – anderenfalls würden alle zur Klasse 0 und 1 gehörenden Objekte in der Trainingsdatenmenge landen und die Testdatenmenge bestünde aus 45 Objekten der Klasse 2. Bislang hatten wir mit dem Parameter `random_state=1` eine Initialisierung des Pseudozufallszahlengenera-

tors vorgenommen, der zum Durchmischen der Datenmenge vor der Aufteilung verwendet wird. Auf diese Weise sind die Ergebnisse reproduzierbar.

Letzten Endes nutzen wir die integrierte Unterstützung für geschichtete Zufallsstichproben mit `stratify=y` zu unserem Vorteil. Das bedeutet, dass die `train_test_split`-Methode Trainings- und Testdatenmengen liefert, in denen die Verhältnisse des Vorkommens der verschiedenen Klassenbezeichnungen dem Verhältnis in der Eingabedatenmenge entsprechen. Wir verwenden NumPys `bincount`-Funktion, die zählt, wie oft die verschiedenen Werte in einem Array vorkommen, um zu überprüfen, ob das tatsächlich der Fall ist:

```
>>> print('Bezeichner in y:', np.bincount(y))
Bezeichner in y: [50 50 50]
>>> print('Bezeichner in y_train:', np.bincount(y_train))
Bezeichner in y_train: [35 35 35]
>>> print('Bezeichner in y_test:', np.bincount(y_test))
Bezeichner in y_test: [15 15 15]
```

Wie Sie am Beispiel des Gradientenabstiegsverfahrens in Kapitel 2 sehen konnten, ist für viele Lern- und Optimierungsalgorithmen eine Standardisierung der Merkmale erforderlich, um eine optimale Leistung zu erzielen. Hier standardisieren wir die Merkmale mit der `StandardScaler`-Klasse, die Teil des `preprocessing`-Moduls von scikit-learn ist:

```
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> sc.fit(X_train)
>>> X_train_std = sc.transform(X_train)
>>> X_test_std = sc.transform(X_test)
```

Der Code lädt die `StandardScaler`-Klasse des `preprocessing`-Moduls und initialisiert ein neues `StandardScaler`-Objekt, das den Variablen `sc` zugewiesen wird. Mit der `fit`-Methode schätzt das `StandardScaler`-Objekt den Mittelwert  $\mu$  und die Standardabweichung  $\sigma$  für jedes Merkmal der Trainingsdaten ab. Durch den Aufruf der `transform`-Methode werden die Trainingsdaten unter Verwendung der abgeschätzten Parameter  $\mu$  und  $\sigma$  standardisiert. Beachten Sie, dass wir dieselben Parameter wie bei der Standardisierung der Testdaten verwenden, damit die Werte in den Trainings- und Testdatenmengen vergleichbar bleiben.

Nach der Standardisierung der Trainingsdaten können wir das Perzepron-Modell trainieren. Die meisten scikit-learn-Algorithmen unterstützen mittels der *One-vs.-Rest*-Methode (OvR-Methode) standardmäßig bereits die Mehrfachklassifizierungen. Das erlaubt es uns, das Perzepron mit den Daten aller drei Blumenklassen gleichzeitig zu füttern. Der Code dafür sieht wie folgt aus:

```
>>> from sklearn.linear_model import Perceptron  
>>> ppn = Perceptron(n_iter=40, eta0=0.1, random_state=1)  
>>> ppn.fit(X_train_std, y_train)
```

Die scikit-learn-Schnittstelle ähnelt der Perzeptron-Implementierung in Kapitel 2: Nach dem Laden der `Perceptron`-Klasse des `linear_model`-Moduls initialisieren wir ein neues `Perceptron`-Objekt und trainieren das Modell mit der `fit`-Methode. Der Modellparameter `eta0` ist hier das Pendant zur Lernrate `eta`, die wir in der eigenen Perzeptron-Implementierung benutzt haben. Und der Parameter `n_iter` gibt die Anzahl der Epochen an (die Anzahl der Durchläufe der Trainingsdaten).

Wie Sie aus Kapitel 2 wissen, muss etwas herumexperimentiert werden, um eine geeignete Lernrate zu finden: Ist sie zu groß, wird der Algorithmus das globale Minimum überspringen. Ist sie hingegen zu klein, benötigt der Algorithmus mehr Epochen, bis er konvergiert, was das Lernen verlangsamen kann – insbesondere bei großen Datensätzen. Wir verwenden auch hier wieder den Parameter `random_state`, um den Zustand der Trainingsdatenmenge nach dem Ende der einzelnen Epochen reproduzieren zu können.

Wenn das scikit-Modell trainiert wurde, können wir mit der `predict`-Methode wie bei unserer Perzeptron-Implementierung aus Kapitel 2 Vorhersagen treffen. Der Code dafür sieht folgendermaßen aus:

```
>>> y_pred = ppn.predict(X_test_std)  
>>> print('Fehlklassifizierte Exemplare: %d' \  
      % (y_test != y_pred).sum())  
Fehlklassifizierte Exemplare: 3
```

Wie die Ausführung des Codes zeigt, hat das Perzepron 3 der 45 Exemplare fehlklassifiziert. Die Fehlklassifizierungsrate beträgt somit 0,067 oder 6,7 Prozent ( $3/45 \approx 0,067$ ).

### Tipp

In der Praxis wird statt einer Fehlklassifizierungsrate die *Korrektklassifizierungsrate* eines Modells angegeben, die sich so errechnet:

$1 - \text{Fehlklassifizierungsrate} \approx 0,933$  oder 93,3 Prozent.

In scikit-learn gibt es eine Vielzahl von verschiedenen Bewertungskriterien, die über das `metrics`-Modul zur Verfügung stehen. Wir können beispielsweise die Korrektklassifizierungsrate des Perzeptrons bei der Klassifizierung der Testdatensetze wie folgt berechnen:

### Kapitel 3

#### Machine-Learning-Klassifizierer mit scikit-learn verwenden

```
>>> from sklearn.metrics import accuracy_score  
>>> print(Korrektklassifizierungsrate: %.2f' % accuracy_score(y_test, y_pred))  
Korrektklassifizierungsrate: 0.93
```

Hier enthält `y_test` die tatsächlichen Klassenbezeichnungen und `y_pred` die vorhergesagten. Alle scikit-learn-Klassifizierer besitzen eine `score`-Methode, die Sie alternativ wie folgt zum Berechnen der Korrektklassifizierungsrate verwenden können:

```
>>> print('Korrektklassifizierungsrate: %.2f'  
      % ppn.score(X_test_std, y_test))  
Korrektklassifizierungsrate: 0.93
```

### Tipp

Beachten Sie, dass die Bewertung der Leistung des Modells anhand der in diesem Kapitel verwendeten Testdatenmenge erfolgt. In Kapitel 5, *Datenkomprimierung durch Dimensionsreduktion*, werden Sie praktische Verfahren kennenlernen, um eine *Überanpassung* zu erkennen und zu verhindern – unter anderem grafische Analysemethoden wie Lernkurven. Überanpassung bedeutet hier, dass ein Modell zwar die Muster in den Trainingsdaten gut erfasst, die Erkennung aber nicht auf unbekannte Daten verallgemeinern kann.

Nun können wir die Funktion `plot_decision_region` aus Kapitel 2 verwenden, um die *Entscheidungsbereiche* des neu trainierten Perzepron-Modells auszugeben und so zu visualisieren, wie gut es die verschiedenen Blumenexemplare auseinanderhalten kann. Wir nehmen allerdings eine kleine Änderung vor, um die Objekte der Testdatenmenge durch kleine Kreise hervorzuheben. Die entsprechenden Anpassungen sind fett gedruckt:

```
from matplotlib.colors import ListedColormap  
import matplotlib.pyplot as plt  
  
def plot_decision_regions(X, y, classifier, test_idx=None,  
                          resolution=0.02):  
    # Markierungen und Farben einstellen  
    markers = ('s', 'x', 'o', '^', 'v')  
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')  
    cmap = ListedColormap(colors[:len(np.unique(y))])  
  
    # Plotten der Entscheidungsgrenze
```

```

x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, \
                                 resolution), np.arange(x2_min, x2_max, resolution))
Z = classifier.predict(np.array([xx1.ravel(), \
                                 xx2.ravel()]).T)
Z = Z.reshape(xx1.shape)
plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
plt.xlim(xx1.min(), xx1.max())
plt.ylim(xx2.min(), xx2.max())

# Plotten aller Exemplare
for idx, cl in enumerate(np.unique(y)):
    plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                alpha=0.8, c=colors[idx],
                marker=markers[idx], label=cl,
                edgecolor='black')

# Exemplare der Testdatenmenge hervorheben
if test_idx:
    X_test, y_test = X[test_idx, :], y[test_idx]
    plt.scatter(X_test[:, 0], X_test[:, 1], c='',
                edgecolor='black',
                alpha=1.0, linewidths=1, marker='o',
                s=100, label='Testdaten')

```

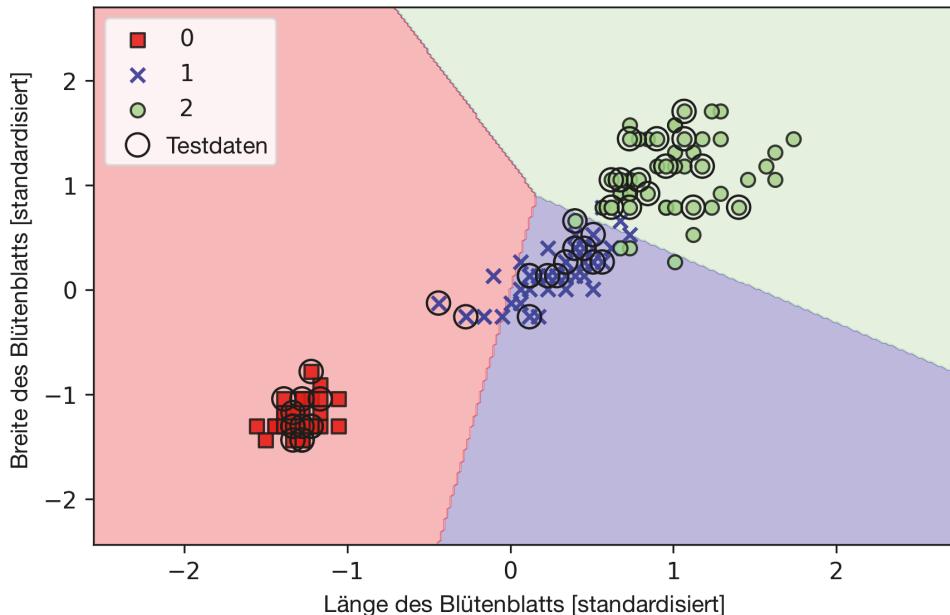
Mit den vorgenommenen Änderungen an der `plot_decision_regions`-Funktion können wir nun die Indizes der Objekte angeben, die in der Ausgabe markiert werden sollen. Hier der Code:

```

>>> X_combined_std = np.vstack((X_train_std, X_test_std))
>>> y_combined = np.hstack((y_train, y_test))
>>> plot_decision_regions(X=X_combined_std,
...                         y=y_combined,
...                         classifier=ppn,
...                         test_idx=range(105,150))
>>> plt.xlabel('Länge des Blütenblatts [standardisiert]')
>>> plt.ylabel('Breite des Blütenblatts [standardisiert]')
>>> plt.legend(loc='upper left')
>>> plt.show()

```

In der Ausgabe ist erkennbar, dass sich die drei Blumenarten nicht vollständig durch eine lineare Entscheidungsgrenze voneinander trennen lassen:



In Kapitel 2 hatten wir festgestellt, dass der Perzepron-Algorithmus nicht konvergiert, wenn sich die Datenmengen nicht vollkommen linear trennen lassen, daher ist von der Verwendung des Perzepron-Algorithmus in der Praxis üblicherweise abzuraten. In den nächsten Abschnitten sehen wir uns leistungsfähigere lineare Klassifizierer an, die auch dann gegen ein Minimum der Straffunktion konvergieren, wenn sich die Klassen nicht vollständig linear voneinander trennen.

### Tipp

Sowohl Perceptron als auch andere Funktionen und Klassen von scikit-learn verfügen über zusätzliche Parameter, die wir hier der Klarheit halber weglassen. Weitere Informationen über diese Parameter finden Sie in der Python-Hilfe (geben Sie dazu z.B. `help(Perceptron)` ein) ebenso wie in der ausgezeichneten Onlinedokumentation, die unter <http://scikit-learn.org/stable/> nachzulesen ist.

### 3.3 Klassenwahrscheinlichkeiten durch logistische Regression modellieren

Die Perzepron-Regel stellt zwar eine schöne und verständliche Einführung in selbstlernende Klassifizierungsalgorithmen dar, hat jedoch den großen Nachteil, dass sie nicht konvergiert, wenn die Klassen nicht vollständig linear getrennt wer-

den können. Die Klassifizierungsaufgabe des vorangegangenen Abschnitts ist ein Beispiel für solch einen Fall. Den Grund dafür können wir uns anschaulich vorstellen: Die Gewichtungen werden kontinuierlich aktualisiert, weil es in jeder Epoche mindestens ein fehlklassifiziertes Exemplar gibt. Man kann natürlich die Lernrate ändern und die Anzahl der Epochen erhöhen, aber seien Sie gewarnt, dass das Perzeptron bei dieser Datensammlung niemals konvergieren wird. Statt dessen sehen wir uns lieber einen weiteren einfachen, aber trotzdem leistungsfähigen Algorithmus für lineare und binäre Klassifizierungsaufgaben an: die *logistische Regression*. Trotz des Namens handelt es sich hier um ein Modell zur Klassifizierung, nicht zur Regression.

### 3.3.1 Logistische Regression und bedingte Wahrscheinlichkeiten

Die logistische Regression ist ein Klassifizierungsmodell, das sich sehr leicht implementieren lässt und mit linear trennbaren Klassen sehr gut funktioniert. Es ist daher einer der meistgebrauchten Klassifizierungsalgorithmen. Ebenso wie Perzeptron und Adaline ist die in diesem Kapitel vorgestellte logistische Regression ein lineares Modell zur binären Klassifizierung, das mit dem OvR-Verfahren zu einer Mehrfachklassifizierung erweitert werden kann.

Um die der logistischen Regression zugrunde liegende Idee als Wahrscheinlichkeitsmodell zu erläutern, müssen wir uns zunächst mit dem *Chancenverhältnis* befassen, das die Wahrscheinlichkeit für das Eintreten eines bestimmten Ereignisses beschreibt. Das Chancenverhältnis kann als  $\frac{p}{(1-p)}$  ausgedrückt werden, wobei  $p$  für die Wahrscheinlichkeit des Positivereignisses steht. Der Begriff »Positivereignis« bedeutet hier nicht unbedingt »gut«, sondern bezieht sich auf das Ereignis, das wir vorhersagen möchten, beispielsweise die Wahrscheinlichkeit, dass ein Patient an einer bestimmten Krankheit leidet. Stellen Sie sich das Positivereignis als ein Ereignis vor, dem die Klassenbezeichnung  $y = 1$  zugeordnet ist. Des Weiteren definieren wir die `logit`-Funktion, die einfach nur der Logarithmus des Chancenverhältnisses ist:

$$\text{logit}(p) = \log \frac{p}{(1-p)}$$

Beachten Sie, dass es sich hier um den natürlichen Logarithmus handelt, wie es in der Informatik gängige Konvention ist. Die `logit`-Funktion nimmt Werte zwischen 0 und 1 entgegen und bildet sie auf Werte aus dem gesamten Bereich der reellen Zahlen ab. Auf diese Weise kann eine lineare Beziehung zwischen Merkmalswerten und dem Logarithmus des Chancenverhältnisses formuliert werden:

$$\text{logit}(p(y=1 | \mathbf{x})) = w_0 x_0 + w_1 x_1 + \dots + w_m x_m = \sum_{i=0}^m w_i x_i = \mathbf{w}^T \mathbf{x}$$

Hier gibt  $p(y=1|x)$  die bedingte Wahrscheinlichkeit dafür an, dass ein bestimmtes Objekt bei gegebenem Merkmal  $x$  zur Klasse 1 gehört.

Wir sind ja eigentlich daran interessiert, die Wahrscheinlichkeit dafür vorherzusagen, dass ein konkretes Objekt zu einer bestimmten Klasse gehört. Diese Wahrscheinlichkeit ergibt sich aus dem Kehrwert der `logit`-Funktion. Sie wird auch als *logistische Funktion* oder aufgrund der charakteristischen S-Form einfach als *Sigmoidfunktion* bezeichnet:

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

Hier ist  $z$  die Nettoeingabe, also die Linearkombination aus Gewichtungen und Merkmalen, die folgendermaßen berechnet wird:

$$z = \mathbf{w}^T \mathbf{x} = w_0 + w_1 x_1 + \dots + w_m x_m$$

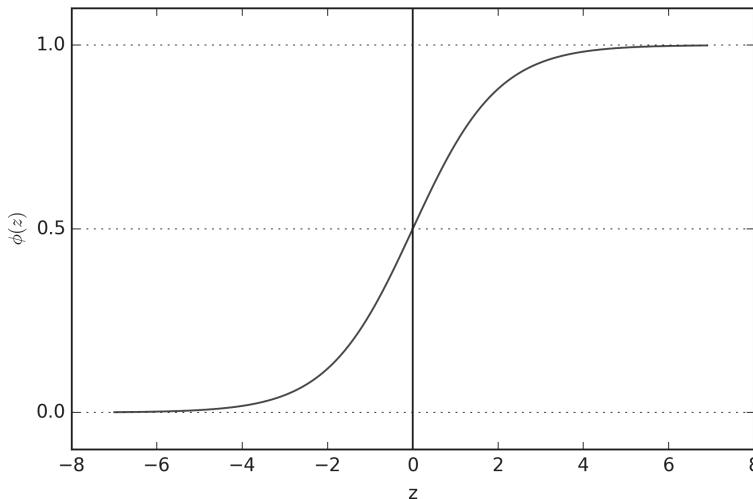
### Tipp

Beachten Sie, dass sich  $w_0$  (wie bei der in Kapitel 2 verwendeten Konvention) auf die Bias-Einheit bezieht und ein zusätzlicher Eingabewert für  $x_0$  ist, dem der Wert 1 zugewiesen ist.

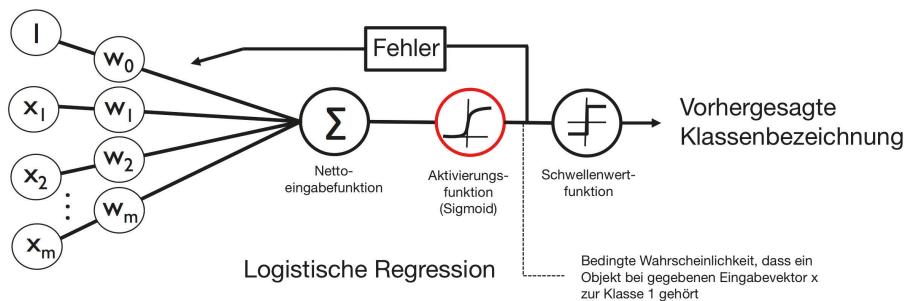
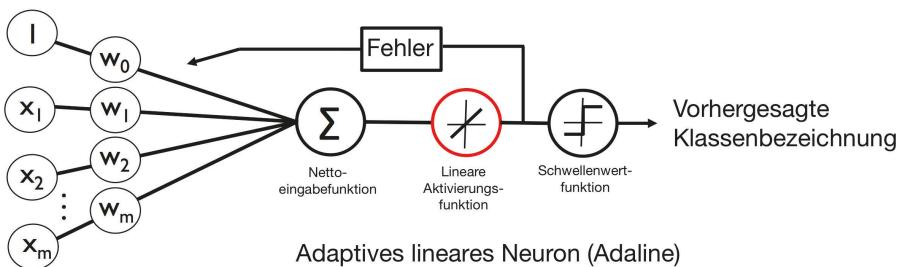
Nun geben wir die Sigmoidfunktion für einige Werte zwischen -7 und 7 aus, um einmal ihre Form zu sehen:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> def sigmoid(z):
...     return 1.0 / (1.0 + np.exp(-z))
>>> z = np.arange(-7, 7, 0.1)
>>> phi_z = sigmoid(z)
>>> plt.plot(z, phi_z)
>>> plt.axvline(0.0, color='k')
>>> plt.ylim(-0.1, 1.1)
>>> plt.xlabel('z')
>>> plt.ylabel('$\phi(z)$')
>>> # y-Achsenmarkierungen und Raster
>>> plt.yticks([0.0, 0.5, 1.0])
>>> ax = plt.gca()
>>> ax.yaxis.grid(True)
>>> plt.show()
```

Nun sollte eine Sigmoidkurve (S-förmige Kurve) angezeigt werden:



$\phi(z)$  geht gegen 1, wenn  $z$  gegen unendlich geht ( $z \rightarrow \infty$ ), denn  $e^{-z}$  wird bei großem  $z$  sehr klein. Umgekehrt geht  $\phi(z)$  gegen 0, wenn  $z \rightarrow -\infty$ , da der Nenner sehr groß wird. Die Sigmoidfunktion nimmt also reelle Zahlen als Eingabe entgegen und bildet sie auf das Intervall  $[0, 1]$  ab. Außerdem schneidet die Kurve die y-Achse, wenn  $\phi(z) = 0.5$  ist.



Um ein Gespür für das logistische Regressionsmodell zu entwickeln, können wir es der Adaline-Implementierung aus Kapitel 2 gegenüberstellen. Bei Adaline haben wir die identische Abbildung  $\phi(z) = z$  als Aktivierungsfunktion verwendet. Bei der logistischen Regression wird als Aktivierungsfunktion die soeben definierte Sigmoidfunktion benutzt, wie Sie der nachstehenden Abbildung entnehmen können.

Die Ausgabe der Sigmoidfunktion wird dann als die Wahrscheinlichkeit  $\phi(z) = P(y=1 | \mathbf{x}; \mathbf{w})$  interpretiert, dass ein bestimmtes Exemplar bei gegebenen, durch die Gewichtungen  $\mathbf{w}$  parametrisierten Merkmalen  $\mathbf{x}$  zur Klasse 1 gehört. Wenn sich beispielsweise für ein bestimmtes Blumenexemplar  $\phi(z) = 0.8$  ergibt, heißt das: Die Chance, dass es sich um ein Exemplar einer Iris versicolor handelt, beträgt 80 Prozent. Auf ähnliche Weise kann die Wahrscheinlichkeit, dass es sich um eine Iris setosa handelt, als  $P(y=0 | \mathbf{x}; \mathbf{w}) = 1 - P(y=1 | \mathbf{x}; \mathbf{w}) = 0.2$  oder 20 Prozent berechnet werden. Die vorhergesagte Wahrscheinlichkeit kann dann mittels einer Schwellenwertfunktion in ein binäres Ergebnis konvertiert werden:

$$\hat{y} = \begin{cases} 1, & \text{wenn } \phi(z) \geq 0.5 \\ 0 & \text{anderenfalls} \end{cases}$$

Wenn Sie sich das Diagramm der Sigmoidfunktion ansehen, ist das äquivalent zu

$$\hat{y} = \begin{cases} 1, & \text{wenn } z \geq 0.0 \\ 0 & \text{anderenfalls} \end{cases}$$

Tatsächlich gibt es viele Anwendungsfälle, bei denen man nicht nur an der vorhergesagten Klassenbezeichnung interessiert ist, sondern sich auch die Abschätzung der Wahrscheinlichkeit einer Klassenzugehörigkeit als besonders nützlich erweist. Beispielsweise wird bei der Wettervorhersage eine logistische Regression verwendet, um nicht nur zu prognostizieren, ob es an einem bestimmten Tag regnet, sondern auch, um eine Regenwahrscheinlichkeit anzugeben. In gleicher Weise kann eine logistische Regression genutzt werden, um die Wahrscheinlichkeit vorherzusagen, dass ein Patient bei gegebenen Symptomen an einer bestimmten Krankheit leidet. Daher sind logistische Regressionen insbesondere im medizinischen Bereich weit verbreitet.

### 3.3.2 Gewichtungen der logistischen Straffunktion ermitteln

Sie wissen nun, wie man das logistische Regressionsmodell dazu verwenden kann, Wahrscheinlichkeiten und Klassenbezeichnungen vorherzusagen. Sehen wir uns als Nächstes kurz die Parameter des Modells an, zum Beispiel die Gewichtungen  $\mathbf{w}$ . Im vorangegangenen Kapitel hatten wir die Straffunktion als Summe der quadrierten Abweichungen definiert:

$$J(\mathbf{w}) = \sum_i \frac{1}{2} (\phi(z^{(i)}) - y^{(i)})^2$$

Diese haben wir minimiert, um die Gewichtungen  $\mathbf{w}$  für das Adaline-Klassifizierungsmodell zu ermitteln. Um zu erklären, wie man die Straffunktion einer logistischen Regression herleitet, müssen wir zunächst die Wahrscheinlichkeit  $L$  (*Likelihood*) definieren, die beim logistischen Regressionsmodell maximiert werden soll, und zwar unter der Annahme, dass die einzelnen Objekte in unserer Datensammlung voneinander unabhängig sind. Hier die dazugehörige Formel:

$$L(\mathbf{w}) = P(\mathbf{y} | \mathbf{x}; \mathbf{w}) = \prod_{i=1}^n P(y^{(i)} | x^{(i)}; \mathbf{w}) = \prod_{i=1}^n \left( \phi(z^{(i)}) \right)^{y^{(i)}} \left( 1 - \phi(z^{(i)}) \right)^{1-y^{(i)}}$$

In der Praxis erweist es sich als einfacher, den (natürlichen) Logarithmus dieser Gleichung zu maximieren, die als *Log-Likelihood-Funktion* bezeichnet wird:

$$l(\mathbf{w}) = \log L(\mathbf{w}) = \sum_{i=1}^n y^{(i)} \log(\phi(z^{(i)})) + (1 - y^{(i)}) \log(1 - \phi(z^{(i)}))$$

Zum einen senkt das Logarithmieren das Potenzial für einen arithmetischen Unterlauf, der auftreten kann, wenn die Wahrscheinlichkeiten sehr klein sind. Und zum anderen können wir das Produkt der Faktoren als eine Summe der Faktoren schreiben, was es erleichtert, die Ableitung dieser Funktion durch den Additionstrick zu erhalten.

Nun könnten wir zur Maximierung dieser Log-Likelihood-Funktion einen Optimierungsalgorithmus wie das Gradientenabstiegsverfahren aus Kapitel 2 verwenden. Alternativ kann man die Log-Likelihood-Funktion auch als eine Straffunktion  $J$  formulieren, die sich wie in Kapitel 2 mit dem Gradientenabstiegsverfahren *minimieren* lässt:

$$J(\mathbf{w}) = \sum_{i=1}^n -y^{(i)} \log(\phi(z^{(i)})) - (1 - y^{(i)}) \log(1 - \phi(z^{(i)}))$$

Um eine bessere Vorstellung von dieser Straffunktion zu bekommen, betrachten wir ihren Wert für ein einzelnes Objekt:

$$J(\phi(z), y; \mathbf{w}) = -y \log(\phi(z)) - (1 - y) \log(1 - \phi(z))$$

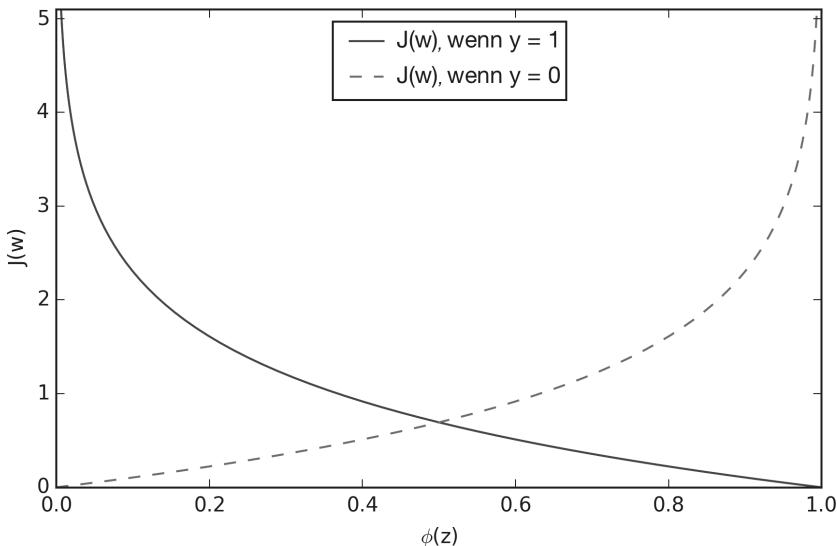
Wenn man sich diese Gleichung ansieht, wird deutlich, dass der erste oder der zweite Term null wird, wenn  $y = 0$  bzw.  $y = 1$  ist:

$$J(\phi(z), y; \mathbf{w}) = \begin{cases} -\log(\phi(z)), & \text{wenn } y = 1 \\ -\log(1 - \phi(z)), & \text{wenn } y = 0 \end{cases}$$

Hier ist der Code zum Erstellen eines Diagramms, das den Wert der Straffunktion bei der Klassifizierung eines einzelnen Objekts für verschiedene Werte von  $\phi(z)$  zeigt:

```
>>> def cost_1(z):
...     return - np.log(sigmoid(z))
>>> def cost_0(z):
...     return - np.log(1 - sigmoid(z))
>>> z = np.arange(-10, 10, 0.1)
>>> phi_z = sigmoid(z)
>>> c1 = [cost_1(x) for x in z]
>>> plt.plot(phi_z, c1, label='J(w), wenn y=1')
>>> c0 = [cost_0(x) for x in z]
>>> plt.plot(phi_z, c0, linestyle='--',
...             label='J(w), wenn y=0')
>>> plt.ylim(0.0, 5.1)
>>> plt.xlim([0, 1])
>>> plt.xlabel('$\phi(z)$')
>>> plt.ylabel('J(w)')
>>> plt.legend(loc='best')
>>> plt.show()
```

Das resultierende Diagramm zeigt die Kurve im x-Bereich von 0 bis 1 (die Eingabewerte für  $z$  lagen zwischen -10 und 10). Die y-Achse zeigt den Wert der Straffunktion an.



Die Straffunktion geht gegen null (durchgezogene Kurve), wenn wir korrekt vorhersagen, dass ein Exemplar zur Klasse 1 gehört. Sie geht ebenfalls gegen null,

wenn  $y = 0$  richtig vorhergesagt wird (gestrichelte Kurve). Ist die Vorhersage hingegen falsch, geht der Wert gegen unendlich. Entscheidend ist hier, dass falsche Vorhersagen zunehmend härter bestraft werden.

### 3.3.3 Konvertieren einer Adaline-Implementierung in einen Algorithmus für eine logistische Regression

Wenn wir selbst ein logistisches Regressionsmodell implementieren möchten, könnten wir einfach die Straffunktion  $J$  der Adaline-Implementierung aus Kapitel 2 durch die neue Straffunktion ersetzen:

$$J(\mathbf{w}) = -\sum_i y^{(i)} \log(\phi(z^{(i)})) + (1 - y^{(i)}) \log(1 - \phi(z^{(i)}))$$

Sie ermittelt den Aufwand der Klassifizierung aller Trainingsobjekte je Epoche. Außerdem müssen wir die lineare Aktivierungsfunktion durch die Sigmoid ersetzen und die Schwellenwertfunktion dahin gehend ändern, dass sie statt  $-1$  und  $+1$  die Klassenbezeichnungen  $0$  und  $1$  zurückgibt. Wenn wir diese drei Änderungen vornehmen, hätten wir ein funktionierendes logistisches Regressionsmodell wie dieses:

```
class LogisticRegressionGD(object):
    """Logistic Regression Classifier using gradient descent.

    Parameters
    -----
    eta : float
        Lernrate (zwischen 0.0 and 1.0)
    n_iter : int
        Durchläufe der Trainingsdatenmenge
    random_state : int
        Zufallszahlgenerator für zufällige Gewichtung
        initialisieren

    Attributes
    -----
    w_ : 1d-array
        Gewichtungen nach Anpassung
    cost_ : list
        Summe der quadrierten Werte der Straffunktion
        pro Epoche
    """

    def __init__(self, eta=0.05, n_iter=100, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state
```

```
def fit(self, X, y):
    """ Fit-Trainingsdaten
    Parameters
    -----
    X : {array-like}, shape = [n_samples, n_features]
        Trainingsvektoren, n_samples ist die
        Anzahl der Exemplare und n_features
        ist die Anzahl der Merkmale.
    y : array-like, shape = [n_samples]
        Zielwerte

    Rückgabewert
    -----
    self : object
    """
    rgen = np.random.RandomState(self.random_state)
    self.w_ = rgen.normal(loc=0.0, scale=0.01,
                          size=1 + X.shape[1])
    self.cost_ = []
    for i in range(self.n_iter):
        net_input = self.net_input(X)
        output = self.activation(net_input)
        errors = (y - output)
        self.w_[1:] += self.eta * X.T.dot(errors)
        self.w_[0] += self.eta * errors.sum()
        # Wir berechnen nun den Wert der Strafffunktion
        # der logistischen Regression, nicht mehr die
        # Summe der quadrierten Werte der Strafffunktion.
        cost = (-y.dot(np.log(output)) -
                ((1 - y).dot(np.log(1 - output))))
        self.cost_.append(cost)
    return self

def net_input(self, X):
    """Nettoeingabe berechnen"""
    return np.dot(X, self.w_[1:]) + self.w_[0]

def activation(self, z):
    """Logistische Aktivierungsfunktion berechnen"""
    return 1. / (1. + np.exp(-np.clip(z, -250, 250)))

def predict(self, X):
    """Klassenbezeichnung zurückgeben"""

```

```

    return np.where(self.net_input(X) >= 0.0, 1, 0)
    # Entspricht:
    # return np.where(self.activation(self.net_input(X))
    #                  >= 0.5, 1, 0)

```

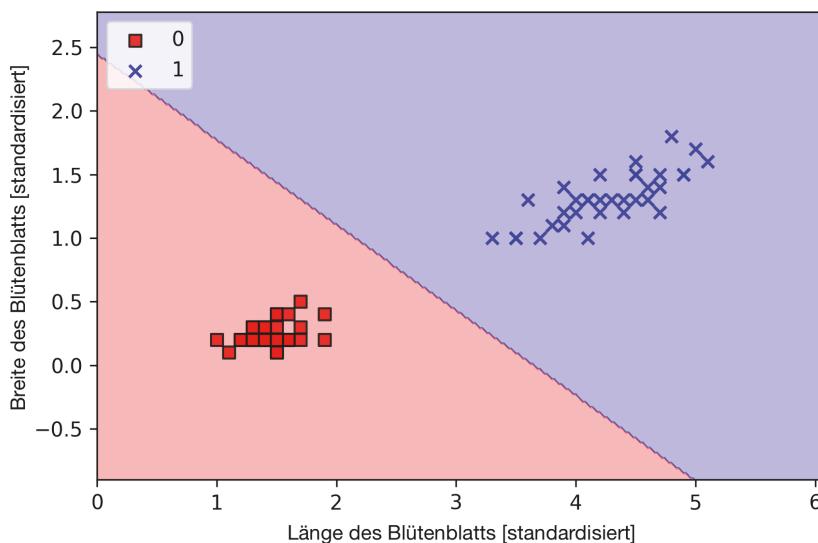
Bei der Anpassung eines logistischen Regressionsmodells müssen wir daran denken, dass es nur für binäre Klassifizierungsaufgaben geeignet ist. Wir berücksichtigen also nur Iris setosa und Iris versicolor (die Klassen 0 und 1) und überprüfen, ob die Implementierung der logistischen Regression funktioniert:

```

>>> X_train_01_subset =
...                 X_train[(y_train == 0) | (y_train == 1)]
>>> y_train_01_subset =
...                 y_train[(y_train == 0) | (y_train == 1)]
>>> lrgd = LogisticRegressionGD(eta=0.05,
...                               n_iter=1000,
...                               random_state=1)
>>> lrgd.fit(X_train_01_subset,
...             y_train_01_subset)
>>> plot_decision_regions(X=X_train_01_subset,
...                         y=y_train_01_subset,
...                         classifier=lrgd)
>>> plt.xlabel('Länge des Blütenblatts [standardisiert]')
>>> plt.ylabel('Breite des Blütenblatts [standardisiert]')
>>> plt.legend(loc='upper left')
>>> plt.show()

```

Das resultierende Diagramm sieht so aus:



## Das Gradientenabstiegsverfahren für eine logistische Regression

Mithilfe der Infinitesimalrechnung lässt sich zeigen, dass die Aktualisierung der Gewichtungen bei einer logistischen Regression mittels Gradientenabstiegsverfahren tatsächlich der Gleichung entspricht, die wir in Kapitel 2 beim Adaline-Algorithmus verwendet haben. Zunächst einmal berechnen wir dazu die partielle Ableitung der Log-Likelihood-Funktion nach der  $j$ -ten Gewichtung:

$$\frac{\partial}{\partial w_j} l(\mathbf{w}) = \left( y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \frac{\partial}{\partial w_j} \phi(z)$$

Bevor wir fortfahren, berechnen wir außerdem noch die partielle Ableitung der Sigmoidfunktion:

$$\begin{aligned} \frac{\partial}{\partial z} \phi(z) &= \frac{\partial}{\partial z} \frac{1}{1+e^{-z}} = \frac{1}{(1+e^{-z})^2} e^{-z} = \frac{1}{1+e^{-z}} \left( 1 - \frac{1}{1+e^{-z}} \right) \\ &= \phi(z)(1-\phi(z)) \end{aligned}$$

Nun können wir  $\frac{\partial}{\partial z} \phi(z) = \phi(z)(1-\phi(z))$  in der Ausgangsgleichung substituieren und erhalten:

$$\begin{aligned} &\left( y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \frac{\partial}{\partial w_j} \phi(z) \\ &= \left( y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \phi(z)(1-\phi(z)) \frac{\partial}{\partial w_j} z \\ &= (y(1-\phi(z)) - (1-y)\phi(z)) x_j \\ &= (y - \phi(z)) x_j \end{aligned}$$

Unser eigentliches Ziel besteht ja darin, die Gewichtungen zu finden, die die Log-Likelihood-Funktion maximieren, damit wir die Aktualisierungen der einzelnen Gewichtungen wie folgt durchführen können:

$$w_j := w_j + \eta \sum_{i=1}^n (y^{(i)} - \phi(z^{(i)})) x_j^{(i)}$$

Da wir alle Gewichtungen gleichzeitig aktualisieren, können wir die allgemeine Aktualisierungsregel folgendermaßen formulieren:

$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}$$

$\Delta w$  definieren wir so:

$$\Delta w = \eta \nabla l(w)$$

Weil die Maximierung der Log-Likelihood-Funktion gleichwertig zur Minimierung der vorhin definierten Straffunktion  $J$  ist, können wir die Aktualisierungsregel für das Gradientenabstiegsverfahren wie folgt schreiben:

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_{i=1}^n \left( y^{(i)} - \phi(z^{(i)}) \right) x^{(i)}$$

$$w := w + \Delta w, \Delta w = -\eta \nabla J(w)$$

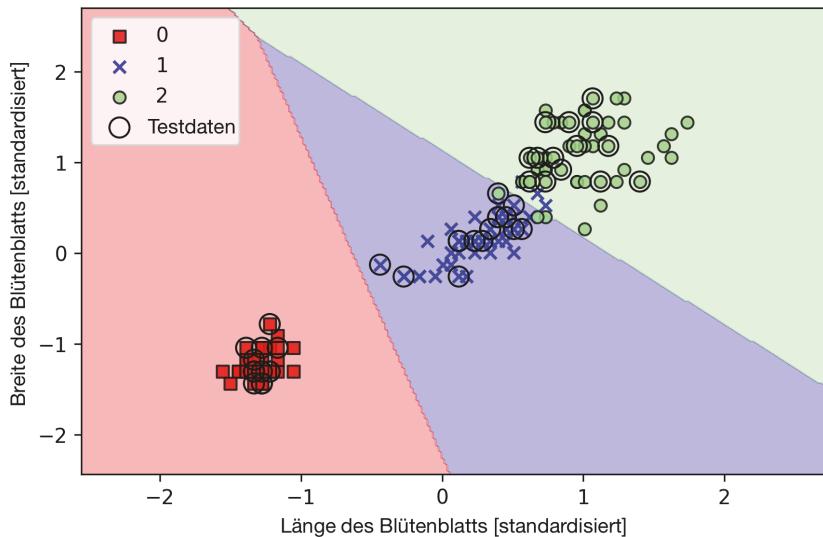
Dies entspricht der Regel des Gradientenabstiegsverfahrens des Adaline-Algorithmus aus Kapitel 2.

### 3.3.4 Trainieren eines logistischen Regressionsmodells mit scikit-learn

Im vorangehenden Abschnitt haben wir eine nützliche Übung absolviert, die Programmierung und mathematische Hintergründe zum Inhalt hatte, die die konzeptuellen Unterschiede zwischen Adaline und logistischer Regression illustrieren. Nun wollen wir scikit-learns hochoptimierte Version einer logistischen Regression betrachten, die auch mehrere Klassen unterstützt (standardmäßig via OvR). Im folgenden Codebeispiel verwenden wir die Klasse `sklearn.linear_model.LogisticRegression` sowie die bereits bekannte `fit`-Methode, um das Modell mit den drei Klassen der standardisierten Trainingsdatenmenge der Blumen zu trainieren.

```
>>> from sklearn.linear_model import LogisticRegression
>>> lr = LogisticRegression(C=100.0, random_state=1)
>>> lr.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                         y_combined, classifier=lr,
...                         test_idx=range(105,150))
>>> plt.xlabel('Länge des Blütenblatts [standardisiert]')
>>> plt.ylabel('Breite des Blütenblatts [standardisiert]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Nach der Anpassung des Modells an die Trainingsdaten geben wir die Entscheidungsbereiche sowie die Trainings- und Testexemplare aus:



Wenn Sie sich den Code zum Trainieren des **Logistic-Regression**-Modells ansehen, werden Sie sich vermutlich fragen, was der Parameter C zu bedeuten hat. Dazu kommen wir in Kürze, aber zunächst sehen wir uns im nächsten Abschnitt kurz das Konzept der Überanpassung und der Regularisierung an. Aber bevor wir mit diesen Themen fortfahren, soll die Erörterung der Wahrscheinlichkeiten der Klassenzugehörigkeit abgeschlossen werden.

Mit der `predict_proba`-Methode können wir die Wahrscheinlichkeiten der Klassenzugehörigkeit der Objekte vorhersagen, beispielsweise für die ersten drei Exemplare der Testdatenmenge:

```
>>> lr.predict_proba(X_test_std[3,:])
```

Der Code gibt ein Array zurück:

```
array([[ 3.20136878e-08,   1.46953648e-01,   8.53046320e-01],
       [ 8.34428069e-01,   1.65571931e-01,   4.57896429e-12],
       [ 8.49182775e-01,   1.50817225e-01,   4.65678779e-13]])
```

Die erste Zeile gibt die Wahrscheinlichkeiten der Klassenzugehörigkeit der ersten Blume an, die zweite Zeile die Wahrscheinlichkeiten der zweiten Blume usw. Beachten Sie, dass die Summe der drei Spalten jeder Zeile erwartungsgemäß 1 ergibt. Sie können das durch die Ausführung von

```
lr.predict_proba(X_test_std[:3, :]).sum(axis=1)
```

überprüfen. Der höchste Wert in der ersten Zeile ist rund 0,853, also gehört das erste Exemplar mit einer vorhergesagten Wahrscheinlichkeit von 85,3% zur dritten Klasse (Iris virginica). Wir können also die vorhergesagte Klasse anhand des höchsten Wertes einer Zeile ermitteln, z.B. mittels NumPys `argmax`-Funktion:

```
>>> lr.predict_proba(X_test_std[:3, :]).argmax(axis=1)
```

Die zurückgegebenen Indizes entsprechen den Klassen Iris virginica, Iris setosa und Iris versicolor:

```
array([2, 0, 0])
```

Die Klassenbezeichnungen, die wir anhand der bedingten Wahrscheinlichkeiten erhalten haben, sind natürlich nur ein manueller Ansatz. Wir können die `predict`-Methode auch direkt aufrufen:

```
>>> lr.predict(X_test_std[:3, :])
array([2, 0, 0])
```

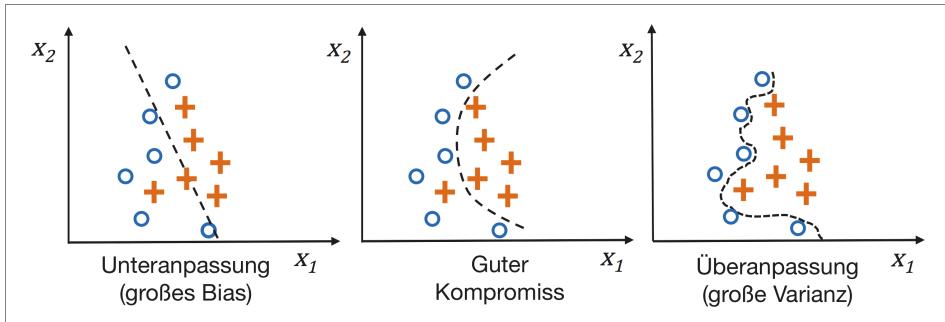
Zum Abschluss noch eine Warnung, für den Fall, dass Sie die Klassenbezeichnung einer einzelnen Blume vorhersagen möchten: scikit-learn erwartet als Eingabe ein zweidimensionales Array. Ein einzelner Slice muss also zunächst in dieses Format konvertiert werden. Dazu können Sie NumPys `reshape`-Methode verwenden:

```
>>> lr.predict(X_test_std[0, :].reshape(1, -1))
array([2])
```

### 3.3.5 Überanpassung durch Regularisierung verhindern

Beim Machine Learning tritt häufig ein Problem auf, das als *Überanpassung* bezeichnet wird: Das Modell funktioniert gut, solange Trainingsdaten verwendet werden – mit unbekannten Daten (Testdaten) kommt es hingegen kaum zurecht, weil die Verallgemeinerung nicht gelingt. Leidet ein Modell an solch einer Überanpassung, spricht man auch von *großer Varianz*, die häufig dadurch verursacht wird, dass es zu viele Parameter gibt, die zu einem für die zugrunde liegenden Daten zu komplexen Modell führen. In ähnlicher Weise kann ein Modell auch unter einer *Unteranpassung* leiden (*großes Bias*, hohe Tendenz zu bestimmten Entscheidungen), wenn es nicht komplex genug ist, um die Muster in den Trainingsdaten zu erkennen und daher ebenfalls bei unbekannten Daten schlecht funktioniert.

Wir haben uns zwar bislang nur mit linearen Klassifizierungsmodellen befasst, allerdings lässt sich die Über- bzw. Unteranpassung am besten anhand einer komplexeren, nichtlinearen Entscheidungsgrenze illustrieren (siehe Abbildung).

**Tipp**

Die **Varianz** ist ein Maß für die Konsistenz (oder die Veränderlichkeit) der Modellvorhersage für ein bestimmtes Objekt, wenn wir das Modell mehrmals trainieren, beispielsweise mit verschiedenen Untermengen der Trainingsdatensetze. So kann man beurteilen, ob das Modell auf die Zufälligkeit der Trainingsdaten empfindlich reagiert. Das **Bias** hingegen gibt an, wie stark die Vorhersagen von den korrekten Werten abweichen, wenn wir das Modell mehrfach auf verschiedene Trainingsdatensetze anwenden. Das Bias ist ein Maß für den systematischen Fehler, der nicht der Zufälligkeit geschuldet ist.

Eine Möglichkeit, einen guten Bias-Varianz-Kompromiss zu finden, besteht darin, die Komplexität des Modells durch *Regularisierung* abzustimmen. Hierbei handelt es sich um eine nützliche Methode, um Kollinearität (hohe Korrelation zwischen den Merkmalen) zu handhaben, Rauschen aus den Daten herauszufiltern und letztlich eine Überanpassung zu verhindern. Der Grundgedanke dabei ist, zusätzliche Informationen (Bias) einzuführen, die extreme Parameter (Gewichtungen) bestrafen. Die gebräuchlichste Art ist die sogenannte *L2-Regularisierung* (manchmal auch als *L2-Schrumpfung* oder *Verringerung der Gewichtungen* bezeichnet), die wie folgt formuliert werden kann:

$$\frac{\lambda}{2} \|\boldsymbol{w}\|^2 = \frac{\lambda}{2} \sum_{j=1}^m w_j^2$$

Hier ist  $\lambda$  der sogenannte *Regularisierungsparameter*.

**Tipp**

Die Regularisierung ist ein weiterer Grund dafür, dass eine Anpassung der Merkmale, wie etwa in Form der Standardisierung, von großer Bedeutung ist. Damit die Regularisierung funktioniert, ist es erforderlich, dass alle Merkmale von vergleichbarer Größenordnung sind.

Um die Regularisierung anzuwenden, müssen wir lediglich der für die logistische Regression definierten Straffunktion einen Regularisierungsterm hinzufügen, der die Gewichtungen während des Trainings verringert:

$$J(\mathbf{w}) = \left[ \sum_{i=1}^n y^{(i)} \left( -\log(\phi(z^{(i)})) \right) + (1-y^{(i)}) \left( -\log(1-\phi(z^{(i)})) \right) \right] + \frac{\lambda}{2} \|\mathbf{w}\|^2$$

Nun können wir über den Regularisierungsparameter  $\lambda$  steuern, wie genau das Modell an die Trainingsdaten angepasst ist, und gleichzeitig die Gewichtungen klein halten. Ein höherer Wert von  $\lambda$  bedeutet dabei eine stärkere Regularisierung.

Der in der **Logistic-Regression**-Klasse von scikit-learn implementierte Parameter  $C$  entstammt einer für **Support Vector Machines** (siehe nächster Abschnitt) geltenden Konvention.  $C$  ist der Kehrwert des Regularisierungsparameters  $\lambda$ :

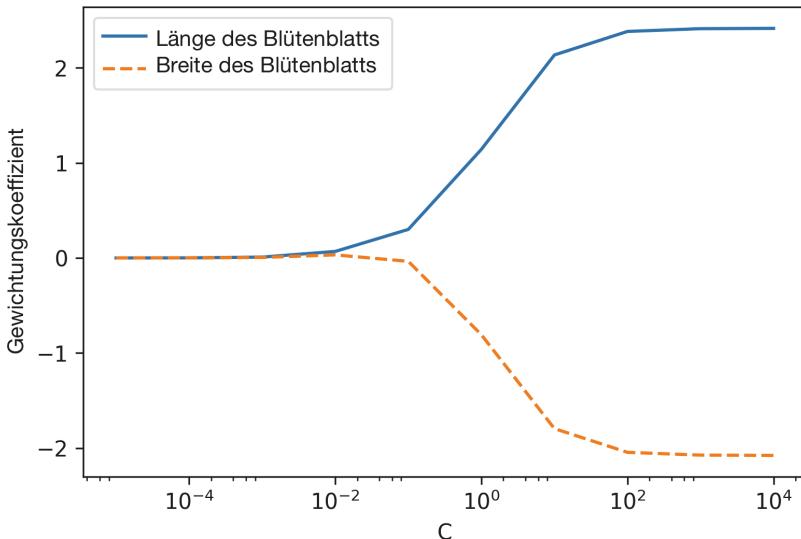
$$C = \frac{1}{\lambda}$$

Demzufolge bedeutet eine Verringerung des Kehrwertes des Regularisierungsparameters  $C$  eine Erhöhung der Regularisierungsstärke. Das können wir durch Ausgabe der L2-Regularisierungspfade für die beiden Gewichtungskoeffizienten visualisieren:

```
>>> weights, params = [], []
>>> for c in np.arange(-5, 5):
...     lr = LogisticRegression(C=10.**c, random_state=1)
...     lr.fit(X_train_std, y_train)
...     weights.append(lr.coef_[1])
...     params.append(10.**c)
>>> weights = np.array(weights)
>>> plt.plot(params, weights[:, 0],
...            label='Länge des Blütenblatts')
>>> plt.plot(params, weights[:, 1], linestyle='--',
...            label='Breite des Blütenblatts')
>>> plt.ylabel('Gewichtungskoeffizient')
>>> plt.xlabel('C')
>>> plt.legend(loc='upper left')
>>> plt.xscale('log')
>>> plt.show()
```

Der Code führt zehn logistische Regressionsmodelle mit verschiedenen Werten des inversen Regularisierungsparameters  $C$  durch. Zur Verdeutlichung zeigen wir nur die Gewichtungskoeffizienten der Klasse 1 (hier die zweite Klasse in der Datenmenge, Iris versicolor) und diejenigen aller Klassifizierer. Beachten Sie, dass wir für die Mehrfachklassifizierung das OvR-Verfahren einsetzen.

Wie das Diagramm zeigt, schrumpfen die Gewichtungskoeffizienten, wenn wir den Parameter  $C$  verkleinern, also die Regularisierungsstärke erhöhen.

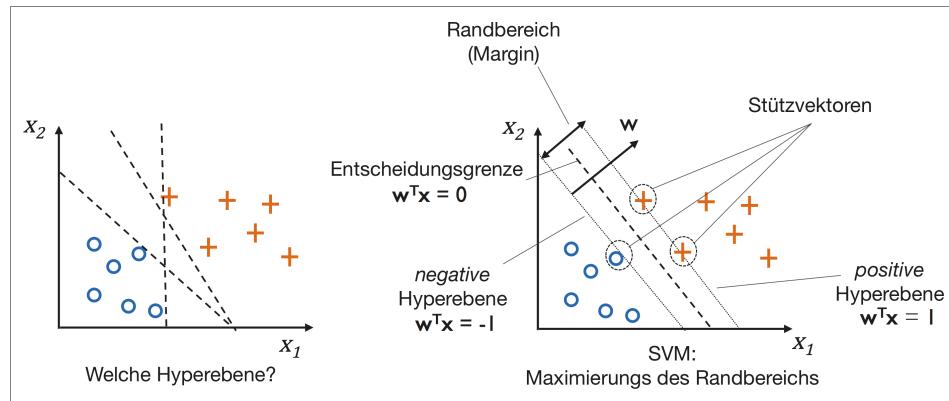


### Tipp

Da eine ausführliche Erläuterung der einzelnen Klassifizierungsalgorithmen über den Rahmen dieses Buches weit hinausgeht, kann ich Lesern, die mehr über logistische Regression erfahren möchten, Dr. Scott Menards *Logistic Regression: From Introductory to Advanced Concepts and Applications*, Sage Publications 2009, nur wärmstens empfehlen.

## 3.4 Maximum-Margin-Klassifizierung mit Support Vector Machines

Eine *Support Vector Machine (SVM, Stützvektormethode)* ist ein weiterer leistungsfähiger und häufig eingesetzter Klassifizierer, der als eine Erweiterung des Perzeptrons aufgefasst werden kann. Während mit dem Perzeptron-Algorithmus die Anzahl von Fehlklassifizierungen minimiert wird, hat die SVM zum Ziel, dass um die Klassengrenzen herum ein möglichst breiter Rand (»Margin«) frei von Objekten bleibt. Dieser Rand ist als der Abstand zwischen der trennenden Hyperebene (Entscheidungsgrenze) und den Objekten der Trainingsdatenmenge definiert, die der Ebene am nächsten sind. Hierbei handelt es sich um die sogenannten *Stützvektoren (Support Vectors)*, die für dieses Verfahren namensgebend waren (siehe Abbildung).



### 3.4.1 Maximierung des Randbereichs

Modelle mit Entscheidungsgrenzen, die einen großen Randbereich besitzen, weisen tendenziell geringere Fehler beim Verallgemeinern auf. Modelle mit schmalem Randbereich sind dagegen eher anfällig für eine Überanpassung. Um ein Gespür für die Maximierung des Randbereichs zu bekommen, sehen wir uns die parallel zur Entscheidungsgrenze verlaufenden Hyperebenen (*positive* und *negative*), die folgendermaßen beschrieben werden können, einmal genauer an:

$$w_0 + \mathbf{w}^T \mathbf{x}_{pos} = 1 \quad (1)$$

$$w_0 + \mathbf{w}^T \mathbf{x}_{neg} = -1 \quad (2)$$

Wenn wir die linearen Gleichungen (1) und (2) voneinander abziehen, erhalten wir:

$$\Rightarrow \mathbf{w}^T (\mathbf{x}_{pos} - \mathbf{x}_{neg}) = 2$$

Wir können diesen Ausdruck mit der Länge des Vektors  $\mathbf{w}$  normieren, die wie folgt definiert ist:

$$\|\mathbf{w}\| = \sqrt{\sum_{j=1}^m w_j^2}$$

Damit gelangen wir zu folgender Gleichung:

$$\frac{\mathbf{w}^T (\mathbf{x}_{pos} - \mathbf{x}_{neg})}{\|\mathbf{w}\|} = \frac{2}{\|\mathbf{w}\|}$$

Die linke Seite dieser Gleichung kann als der Abstand zwischen der positiven und der negativen Hyperebene interpretiert werden, der sogenannte *Margin*, der maximiert werden soll.

Ziel der SVM ist es, diesen Margin zu maximieren. Um das zu erreichen, müssen wir  $\frac{2}{\|\mathbf{w}\|}$  maximieren, mit der Einschränkung, dass die fraglichen Objekte korrekt klassifiziert werden, was sich wie folgt ausdrücken lässt (für  $i = 1 \dots N$ ; dabei ist  $N$  die Anzahl der Objekte in der Datenmenge):

$$w_0 + \mathbf{w}^T \mathbf{x}^{(i)} \geq 1, \text{ wenn } y^{(i)} = 1$$

$$w_0 + \mathbf{w}^T \mathbf{x}^{(i)} < -1, \text{ wenn } y^{(i)} = -1$$

Diese beiden Gleichungen besagen im Wesentlichen, dass sich alle zur negativen Klasse gehörenden Objekte jenseits der negativen Hyperebene und alle zur positiven Klasse gehörenden Objekte jenseits der positiven Hyperebene befinden sollten. Das lässt sich noch kompakter formulieren:

$$y^{(i)} \left( w_0 + \mathbf{w}^T \mathbf{x}^{(i)} \right) \geq 1 \quad \forall_i$$

In der Praxis erweist es sich als einfacher, den reziproken Term  $\frac{1}{2} \|\mathbf{w}\|^2$  zu minimieren, was sich mit quadratischen Programmen lösen lässt, deren eingehende Erläuterung allerdings über den Rahmen dieses Buches hinausgeht. An Support Vector Machines interessierten Lesern sei zu diesem Thema Vladimir Vapniks *The Nature of Statistical Learning Theory*, Springer Science & Business Media, oder Chris J.C. Burges' ausgezeichnete Erklärung in *A Tutorial on Support Vector Machines for Pattern Recognition* (Data Mining and Knowledge Discovery, 2(2):121-167, 1998) empfohlen.

### 3.4.2 Handhabung des nicht linear trennbaren Falls mit Schlupfvariablen

Wir wollen die der Margin-Klassifizierung zugrunde liegenden schwierigeren mathematischen Konzepte an dieser Stelle nicht weiter vertiefen, dennoch soll hier kurz die *Schlupfvariable*  $\xi$  Erwähnung finden. Sie wurde 1995 von Vladimir Vapnik eingeführt und begründete die sogenannte *Soft-Margin-Klassifizierung*. Die Motivation für die Einführung der Schlupfvariablen  $\xi$  war, dass die linearen Beschränkungen für nicht linear trennbare Daten gelockert werden mussten, um die Konvergenz der Optimierung beim Vorhandensein von Fehlklassifizierungen mit entsprechend großer Straffunktion zu ermöglichen.

Die Schlupfvariable wird einfach zu den linearen Einschränkungen addiert:

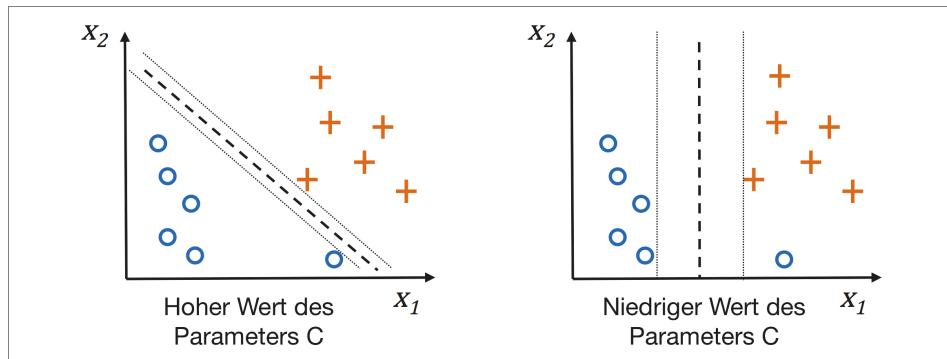
$$\mathbf{w}^T \mathbf{x}^{(i)} \geq 1 - \xi^{(i)}, \quad \text{wenn } y^{(i)} = 1$$

$$\mathbf{w}^T \mathbf{x}^{(i)} < -1 + \xi^{(i)}, \quad \text{wenn } y^{(i)} = -1$$

Hier gilt wieder für  $i = 1 \dots N$  und  $N$  ist die Anzahl der Objekte in der Datenmenge. Anschließend muss der folgende Ausdruck (der den genannten Einschränkungen unterliegt) minimiert werden:

$$\frac{1}{2} \|\mathbf{w}\|^2 + C \left( \sum_i \xi^{(i)} \right)$$

Mit der Variablen  $C$  können wir die Bestrafung für Fehlklassifizierungen steuern: Hohe Werte von  $C$  führen zu einer umfassenderen Straffunktion, sollen die Fehlklassifizierungen hingegen weniger streng gehandhabt werden, wählen wir kleinere Werte für  $C$ . Wir können diesen Parameter nun verwenden, um die Margin-Breite zu steuern und dadurch den Bias-Varianz-Kompromiss fein abstimmen, wie die nachstehende Abbildung illustriert.

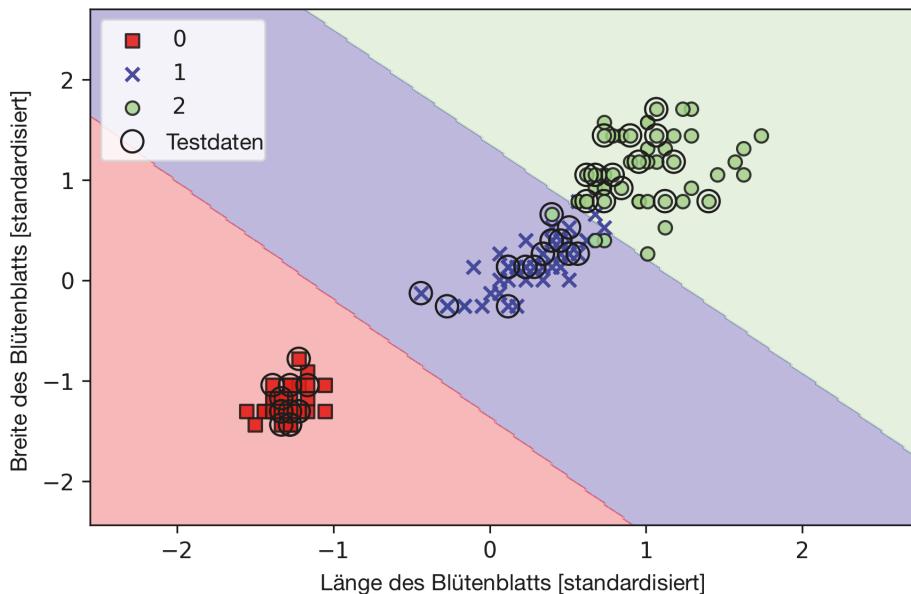


Das Konzept ähnelt der in Zusammenhang mit der logistischen Regression erläuterten Regularisierung, bei der das Verringern des Wertes von  $C$  für ein erhöhtes Bias und eine niedrigere Varianz des Modells sorgt.

Damit haben wir die grundlegenden Konzepte einer linearen SVM kennengelernt und wollen als Nächstes ein SVM-Modell darauf trainieren, die Blumen in der Iris-Datensammlung zu klassifizieren:

```
>>> from sklearn.svm import SVC
>>> svm = SVC(kernel='linear', C=1.0, random_state=1)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                         y_combined, classifier=svm,
...                         test_idx=range(105,150))
>>> plt.xlabel('Länge des Blütenblatts [standardisiert]')
>>> plt.ylabel('Breite des Blütenblatts [standardisiert]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Im nachstehenden Diagramm sind die Entscheidungsbereiche der SVM dargestellt, nachdem der Klassifizierer durch Ausführung des Codes mit der Iris-Datensammlung trainiert wurde.



### Tipp

#### Logistische Regression kontra SVM

Bei praktischen Klassifizierungsaufgaben liefern die lineare logistische Regression und die lineare SVM häufig sehr ähnliche Ergebnisse. Die logistische Regression versucht, die bedingten Wahrscheinlichkeiten der Trainingsdaten zu maximieren und ist daher anfälliger für Ausreißer in den Daten als eine SVM. Bei der SVM sind vor allem die Punkte in der Nähe der Entscheidungsgrenze (die Stützvektoren) ausschlaggebend. Andererseits besitzt die logistische Regression den Vorteil, dass es sich um ein einfacheres Modell handelt, das leichter zu implementieren ist. Darüber hinaus lassen sich logistische Regressionsmodelle leicht aktualisieren, was sie reizvoll macht, wenn es um die Verarbeitung von Datenströmen geht.

### 3.4.3 Alternative Implementierungen in scikit-learn

Die Perceptron- und LogisticRegression-Klassen, die wir in den vorangegangenen Abschnitten mit scikit-learn verwendet haben, machen Gebrauch von der LIBLINEAR-Bibliothek, einer hochoptimierten C/C++-Bibliothek, die an der National

Taiwan University (<http://www.csie.ntu.edu.tw/~cjlin/liblinear/>) entwickelt wurde. Und die SVC-Klasse, die wir beim Trainieren einer SVM benutzt haben, verwendet LIBSVM, eine weitere C/C++-Bibliothek speziell für SVMs (<http://www.csie.ntu.edu.tw/~cjlin/libsvm/>).

Die Verwendung von LIBLINEAR und LIBSVM anstelle von reinem Python-Code hat den Vorteil, dass sich große Mengen von linearen Klassifizierern äußerst schnell trainieren lassen. Allerdings sind die Datenmengen oft zu groß, um in den Arbeitsspeicher eines Computers zu passen. Daher bietet scikit-learn mit der SGDClassifier-Klasse eine alternative Implementierung an, die über die `partial_fit`-Methode auch das Online Learning unterstützt. Das der SGDClassifier-Klasse zugrunde liegende Konzept ähnelt demjenigen des stochastischen Gradientenabstiegsverfahrens, das wir in Kapitel 2 beim Adaline-Algorithmus implementiert haben. Auf dem stochastischen Gradientenabstiegsverfahren basierte Versionen des Perzeptrons, der logistischen Regression und der Support Vector Machine können wie folgt mit Standardparametern initialisiert werden:

```
>>> from sklearn.linear_model import SGDClassifier
>>> ppn = SGDClassifier(loss='perceptron')
>>> lr = SGDClassifier(loss='log')
>>> svm = SGDClassifier(loss='hinge')
```

## 3.5 Nichtlineare Aufgaben mit einer Kernel-SVM lösen

Dass sich mit den sogenannten Kernel-Methoden nichtlineare Klassifizierungsaufgaben lösen lassen, ist ein weiterer Grund dafür, dass sich SVMs beim Machine Learning großer Beliebtheit erfreuen. Bevor wir uns eingehender mit dem Konzept einer *Kernel-SVM* befassen, definieren und legen wir im Folgenden zunächst eine einfache Datensammlung an, um zu betrachten, wie solch eine nichtlineare Klassifizierungsaufgabe aussehen könnte.

### 3.5.1 Kernel-Methoden für linear nicht trennbare Daten

Mit dem folgenden Code erstellen wir unter Verwendung der `logical_xor`-Funktion von NumPy eine einfache Datensammlung nach Art eines XOR-Gatters, in der jeweils 100 Objekten entweder die Klassenbezeichnung 1 oder -1 zugeordnet wird:

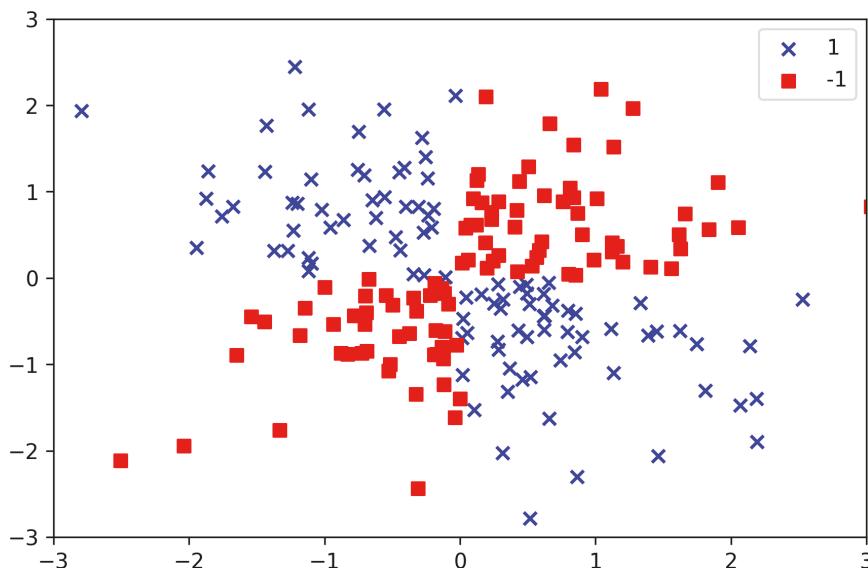
```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> np.random.seed(1)
>>> X_xor = np.random.randn(200, 2)
>>> y_xor = np.logical_xor(X_xor[:, 0] > 0,
```

```

...                         X_xor[:, 1] > 0)
>>> y_xor = np.where(y_xor, 1, -1)
>>> plt.scatter(X_xor[y_xor == 1, 0],
...                  X_xor[y_xor == 1, 1],
...                  c='b', marker='x',
...                  label='1')
>>> plt.scatter(X_xor[y_xor == -1, 0],
...                  X_xor[y_xor == -1, 1],
...                  c='r',
...                  marker='s',
...                  label=-1)
>>> plt.xlim([-3, 3])
>>> plt.ylim([-3, 3])
>>> plt.legend(loc='best')
>>> plt.show()

```

Nach Ausführung des Codes liegt eine XOR-Datensammlung mit zufälligem Rauschen wie in der folgenden Abbildung vor:



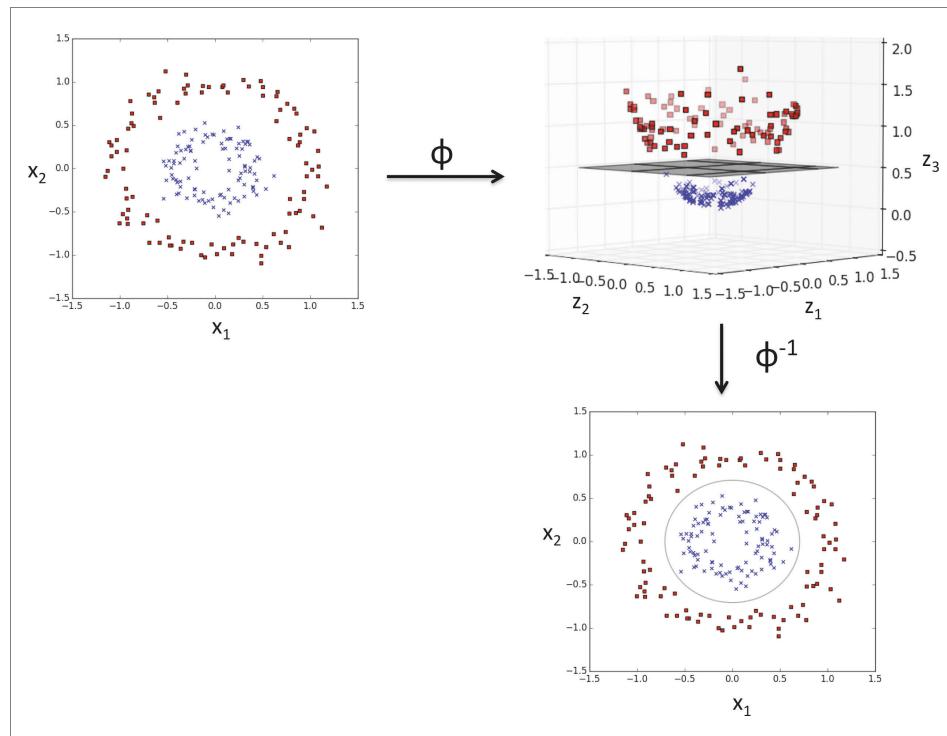
Mit der linearen logistischen Regression oder dem linearen SVM-Modell wären wir offensichtlich kaum in der Lage, die Objekte der positiven und der negativen Klasse durch eine lineare Hyperebene als Entscheidungsgrenze voneinander zu trennen.

Die den Kernel-Methoden zugrunde liegende Idee, um solche linear nicht trennbaren Daten zu handhaben, ist Folgende: Die ursprünglichen Merkmale werden durch eine Zuordnungsfunktion  $\phi$  auf einen höherdimensionalen Raum abgebil-

det, in dem sie linear trennbar sind. In der nächsten Abbildung wird eine zweidimensionale Datenmenge auf einen dreidimensionalen Merkmalsraum abgebildet, in dem sie linear trennbar ist, und zwar durch folgende Abbildung:

$$\phi(x_1, x_2) = (z_1, z_2, z_3) = (x_1, x_2, x_1^2 + x_2^2)$$

Auf diese Weise wird es möglich, die in der Abbildung gezeigten Klassen durch eine lineare Hyperebene zu trennen, die zu einer nichtlinearen Entscheidungsgrenze wird, wenn sie wieder auf den ursprünglichen Merkmalsraum projiziert wird.



### 3.5.2 Mit dem Kernel-Trick Hyperebenen in höherdimensionalen Räumen finden

Zum Lösen einer nichtlinearen Aufgabenstellung mit einer SVM bilden wir die Trainingsdaten mit einer Zuordnungsfunktion  $\phi$  in einen höherdimensionalen Merkmalsraum ab und trainieren ein lineares SVM-Modell für die Klassifizierung in diesem neuen Merkmalsraum. Dann können wir dieselbe Zuordnungsfunktion  $\phi$  verwenden, um neue, unbekannte Daten zu transformieren und mit dem linearen SVM-Modell zu klassifizieren.

Dabei tritt allerdings das Problem auf, dass die Erstellung der neuen Merkmale sehr rechenaufwendig ist, insbesondere wenn wir es mit hochdimensionalen Daten zu tun haben. Hier kommt der sogenannte *Kernel-Trick* ins Spiel. Wir wollen die Lösung mit quadratischen Programmen zum Trainieren einer SVM nicht allzu sehr vertiefen, denn in der Praxis müssen wir lediglich das Skalarprodukt  $\mathbf{x}^{(i)T} \mathbf{x}^{(j)}$  durch  $\phi(\mathbf{x}^{(i)})^T \phi(\mathbf{x}^{(j)})$  ersetzen. Um uns den aufwendigen Schritt der Berechnung dieses Skalarprodukts zu ersparen, definieren wir eine sogenannte Kernel-Funktion:

$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \phi(\mathbf{x}^{(i)})^T \phi(\mathbf{x}^{(j)})$$

Der *RBF-Kernel* (*Radiale Basisfunktion*, deren Wert nur vom Abstand zum Ursprung abhängt; auch *Gaußscher Kernel*) gehört zu den am häufigsten verwendeten:

$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\frac{\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2}{2\sigma^2}\right)$$

Diese Gleichung wird vereinfacht zu:

$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\gamma \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2\right)$$

Hier ist  $\gamma = \frac{1}{2\sigma^2}$  ein freier Parameter, der optimiert werden muss.

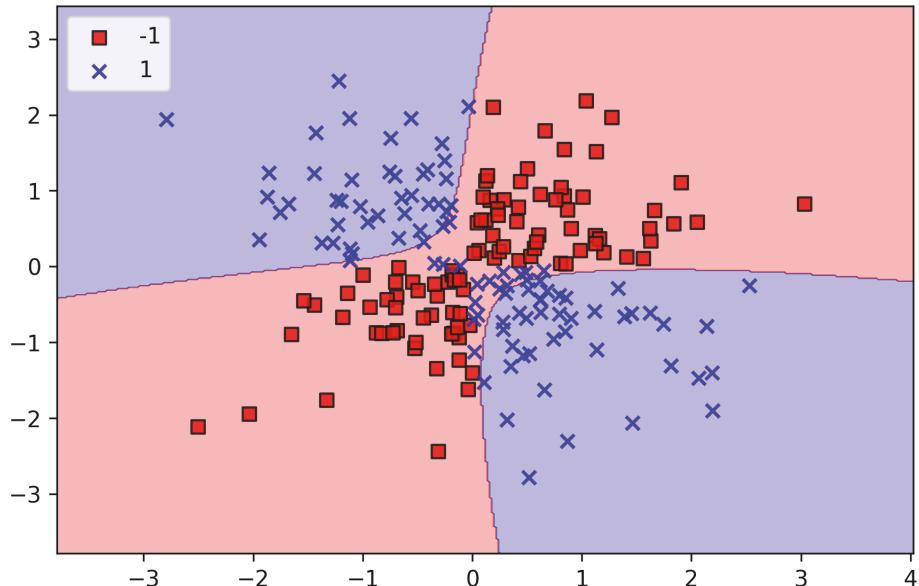
Einfacher ausgedrückt kann der *Kernel* als eine *Ähnlichkeitsfunktion* zwischen Objektpaaren aufgefasst werden. Das Minuszeichen wandelt den Abstand in ein Ähnlichkeitsmaß um und durch den exponentiellen Term liegt der resultierende Ähnlichkeitswert zwischen 1 (bei exakt gleichen Exemplaren) und 0 (bei sehr unterschiedlichen Exemplaren).

Nachdem wir nun den Kernel-Trick in groben Zügen kennengelernt haben, wollen wir prüfen, ob wir eine Kernel-SVM trainieren können, die in der Lage ist, eine vernünftige nichtlineare Entscheidungsgrenze zur Trennung der XOR-Daten zu ziehen. Dazu verwenden wir einfach die SVC-Klasse von scikit-learn, die wir vorher importiert hatten, und ersetzen den Parameter `kernel='linear'` durch `kernel='rbf'`:

```
>>> svm = SVC(kernel='rbf', random_state=1,
...             gamma=0.10, C=10.0)
>>> svm.fit(X_xor, y_xor)
>>> plot_decision_regions(X_xor, y_xor, classifier=svm)
```

```
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Wie das Diagramm zeigt, trennt die Kernel-SVM die XOR-Daten relativ gut:



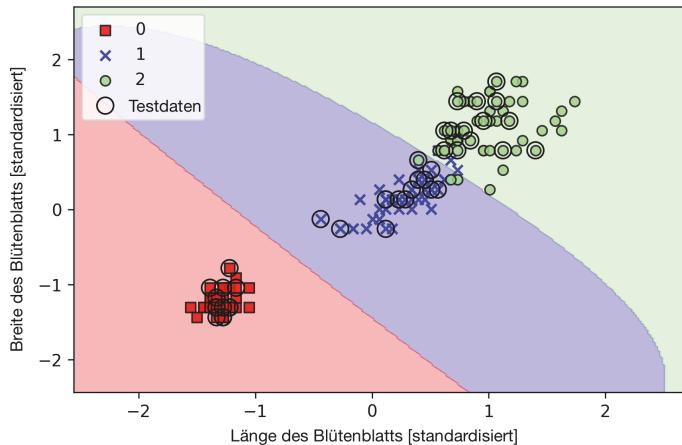
Der Parameter  $\gamma$ , den wir auf `gamma=0.1` gesetzt haben, kann als eine Art »Abschneideparameter« für die gaußsche Sphäre verstanden werden. Wenn wir den Wert von  $\gamma$  erhöhen, steigt der Einfluss oder die Reichweite der Trainingsobjekte, was auch Auswirkungen auf die Entscheidungsgrenze hat, die enger und weniger glatt verläuft. Um ein besseres Gespür für  $\gamma$  zu bekommen, wenden wir eine RBF-Kernel-SVM auf die Iris-Datensammlung an:

```
>>> svm = SVC(kernel='rbf', random_state=1, gamma=0.2, C=1.0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                         y_combined, classifier=svm,
...                         test_idx=range(105,150))
>>> plt.xlabel('Länge des Blütenblatts [standardisiert]')
>>> plt.ylabel('Breite des Blütenblatts [standardisiert]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Da wir einen vergleichsweise kleinen Wert für  $\gamma$  gewählt haben, ist die Entscheidungsgrenze des RBF-Kernel-SVM-Modells relativ weit gefasst, wie in der nachstehenden Abbildung erkennbar ist.

### Kapitel 3

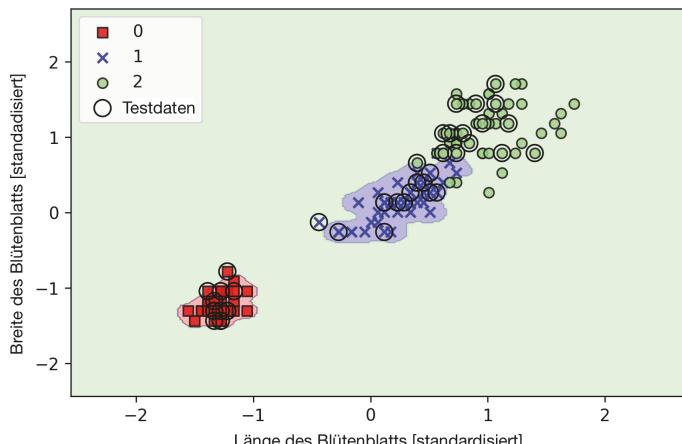
#### Machine-Learning-Klassifizierer mit scikit-learn verwenden



Nun erhöhen wir den Wert von  $\gamma$ , um die Auswirkung auf die Entscheidungsgrenze zu beobachten:

```
>>> svm = SVC(kernel='rbf', random_state=1,
...             gamma=100.0, C=1.0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                         y_combined, classifier=svm,
...                         test_idx=range(105,150))
>>> plt.xlabel('Länges des Blütenblatts [standardisiert]')
>>> plt.ylabel('Breite des Blütenblatts [standardisiert]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Im Diagramm ist erkennbar, dass die Entscheidungsgrenzen der Klassen 0 und 1 bei einem relativ großen Wert von  $\gamma$  nun sehr viel enger sind:

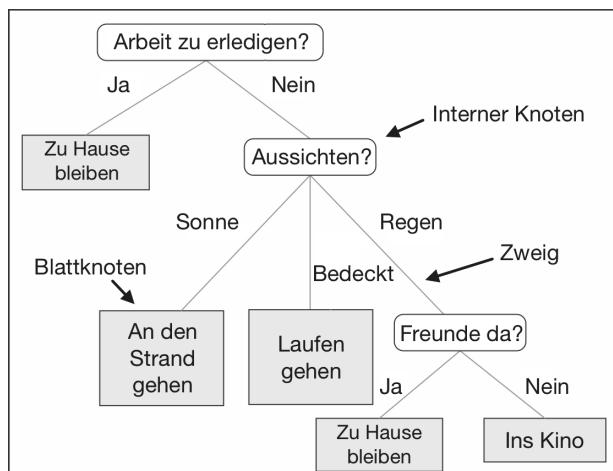


Das Modell ist zwar sehr gut an die Trainingsdaten angepasst, allerdings werden solch einem Klassifizierer bei unbekannten Daten beim Verallgemeinern vermutlich viele Fehler unterlaufen, was zeigt, dass die Optimierung von  $\gamma$  auch beim Verhindern einer Überanpassung eine wichtige Rolle spielt.

## 3.6 Lernen mit Entscheidungsbäumen

Auf *Entscheidungsbäumen* (*Decision Trees*) beruhende Klassifizierer sind attraktive Modelle, wenn die Interpretierbarkeit von Bedeutung ist. Wie die Bezeichnung *Entscheidungsbaum* bereits vermuten lässt, kann man sich dieses Modells so vorstellen, dass bei der Analyse der Daten anhand einer Reihe von Fragen Entscheidungen getroffen werden.

Betrachten wir folgendes Beispiel, bei dem es um einen Entscheidungsbaum geht, der festlegen soll, welche Aktion an einem bestimmten Tag durchgeführt werden soll:



Das Entscheidungsbaummodell erlernt anhand einer Reihe von Fragen die Klassenbezeichnungen der Objekte in der Trainingsdatenmenge. In der Abbildung werden zur Illustration eines Entscheidungsbäums zwar kategoriale Merkmale verwendet, aber das Ganze funktioniert auch, wenn die Merkmale reelle Zahlen sind, wie etwa in der Iris-Datensammlung. Wir könnten beispielsweise einen Grenzwert für die Merkmalsachse *Breite des Blütenblatts* festlegen und Ja/Nein-Fragen wie »Länge des Blütenblatts  $\geq 2,8 \text{ cm}$ ?« stellen.

Beim Entscheidungsalgorithmus fangen wir an der Wurzel des Baums an und teilen die Daten so auf, dass sich der größte *Informationsgewinn* (*IG*, *Information Gain*, siehe nächster Abschnitt) ergibt. Bei jedem Kindknoten des Baums wiederholen wir diese Aufteilungsprozedur, bis wir die Blattknoten erreichen. Das bedeutet,

tet, dass die Exemplare an jedem Knoten jeweils zur selben Klasse gehören. In der Praxis können daraus sehr tief verschachtelte Baumstrukturen mit vielen Knoten resultieren, was wiederum leicht zu einer Überanpassung führt. Daher werden wir den Baum typischerweise »zurechtstutzen«, indem wir seine maximale Tiefe begrenzen (sogenanntes *Pruning*).

### 3.6.1 Maximierung des Informationsgewinns: Daten ausreizen

Um die Knoten an den informativsten Merkmalen aufteilen zu können, müssen wir eine Zielfunktion definieren, die durch den Entscheidungsbaumalgorithmus optimiert werden soll. Hier wollen wir den Informationsgewinn (IG) bei jeder Aufteilung maximieren, was wir folgendermaßen definieren:

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^m \frac{N_j}{N_p} I(D_j)$$

In diesem Fall gibt  $f$  das Merkmal an, an dem die Aufteilung stattfinden soll,  $D_p$  und  $D_j$  sind die Datenmengen des Elternknotens und des  $j$ -ten Kindknotens,  $I$  ist ein Maß für die *Unreinheit (Impurity)*,  $N_p$  repräsentiert die Gesamtzahl der Exemplare des Elternknotens und  $N_j$  ist die Anzahl der Exemplare im  $j$ -ten Kindknoten. Wie man sieht, ergibt sich der Informationsgewinn als Differenz der Unreinheit des Elternknotens und der Summe der Unreinheiten der Kindknoten: Je geringer die Unreinheit der Kindknoten, desto größer ist der Informationsgewinn. Der Einfachheit halber und um den kombinatorischen Suchraum zu verkleinern, implementieren die meisten Bibliotheken (scikit-learn eingeschlossen) allerdings binäre Entscheidungsbäume. Jeder Elternknoten wird also in zwei Kindknoten  $D_{links}$  und  $D_{rechts}$  aufgeteilt:

$$IG(D_p, f) = I(D_p) - \frac{N_{links}}{N_p} I(D_{links}) - \frac{N_{rechts}}{N_p} I(D_{rechts})$$

Bei binären Entscheidungsbäumen finden vornehmlich drei Maße für die Unreinheit Verwendung: der *Gini-Koeffizient* ( $I_G$ ), die *Entropie* ( $I_H$ ) und der *Klassifizierungsfehler* ( $I_E$ ). Fangen wir mit der Definition der Entropie für alle *nicht-leeren* Klassen  $p(i|t) \neq 0$  an:

$$I_H(t) = -\sum_{i=1}^c p(i|t) \log_2 p(i|t)$$

Hier gibt  $p(i|t)$  den Anteil der zur Klasse  $c$  zugehörigen Exemplare für den Knoten  $t$  an. Die Entropie ist daher 0, wenn alle Objekte eines Knotens zur selben Klasse gehören. Sind die Objekte hingegen gleichmäßig auf alle Klassen verteilt, ist die Entropie maximal. Wenn es beispielsweise nur zwei Klassen gibt, ist die Entropie 0, wenn  $p(i=1|t)=1$  oder  $p(i=0|t)=0$ . Sind die Klassen jedoch gleichmäßig

verteilt und es gilt  $p(i=1|t)=0.5$  und  $p(i=0|t)=0.5$ , dann beträgt die Entropie 1. Man könnte daher sagen, dass die Entropie bestrebt ist, die wechselseitig im Baum vorhandenen Informationen zu maximieren.

Der Gini-Koeffizient kann als Maß dafür aufgefasst werden, inwieweit die Wahrscheinlichkeit einer Fehlklassifizierung minimiert ist:

$$I_G(t) = \sum_{i=1}^c p(i|t)(1-p(i|t)) = 1 - \sum_{i=1}^c p(i|t)^2$$

Wie die Entropie nimmt der Gini-Koeffizient sein Maximum an, wenn die Klassen perfekt durchgemischt sind, beispielsweise bei nur zwei Klassen ( $c = 2$ ):

$$I_G(t) = 1 - \sum_{i=1}^2 0.5^2 = 0.5$$

In der Praxis liefern Gini-Koeffizient und Entropie allerdings typischerweise sehr ähnliche Ergebnisse und es lohnt oft nicht, viel Zeit in die Auswertung von Bäumen mit verschiedenen Unreinheitskriterien zu stecken.

Ein weiteres Maß für die Unreinheit ist der Klassifizierungsfehler:

$$I_E(t) = 1 - \max \{p(i|t)\}$$

Hierbei handelt es sich um ein nützliches Kriterium zum »Stutzen« (Pruning) eines Entscheidungsbaums, nicht aber für dessen Konstruktion, da es weniger empfindlich für Änderungen der Klassenwahrscheinlichkeiten der Knoten ist. In der Abbildung sind die beiden möglichen Aufteilungen dargestellt:



Wir fangen beim Elternknoten der Datenmenge  $D_p$  an, die aus jeweils 40 Exemplaren der Klassen 1 und 2 besteht und in zwei Datenmengen  $D_{links}$  und  $D_{rechts}$  aufgeteilt werden soll. Der Informationsgewinn wäre in den beiden Fällen A und B gleich ( $IG_E = 0.25$ ), wenn man den Klassifizierungsfehler als Aufteilungskriterium heranzieht:

$$I_E(D_p) = 1 - 0.5 = 0.5$$

### Kapitel 3

#### Machine-Learning-Klassifizierer mit scikit-learn verwenden

$$A : I_E(D_{links}) = 1 - \frac{3}{4} = 0.25$$

$$A : I_E(D_{rechts}) = 1 - \frac{3}{4} = 0.25$$

$$A : IG_E = 0.5 - \frac{4}{8} 0.25 - \frac{4}{8} 0.25 = 0.25$$

$$B : I_E(D_{links}) = 1 - \frac{4}{6} = \frac{1}{3}$$

$$B : I_E(D_{rechts}) = 1 - 1 = 0$$

$$B : IG_E = 0.5 - \frac{6}{8} \times \frac{1}{3} - 0 = 0.25$$

Verwendet man als Kriterium hingegen den Gini-Koeffizienten, wäre das Szenario  $B (IG_G = 0.1\bar{6})$  dem Szenario  $A (IG_G = 0.125)$  vorzuziehen, denn es ist tatsächlich das »reinere«:

$$I_G(D_p) = 1 - (0.5^2 + 0.5^2) = 0.5$$

$$A : I_G(D_{links}) = 1 - \left( \left( \frac{3}{4} \right)^2 + \left( \frac{1}{4} \right)^2 \right) = \frac{3}{8} = 0.375$$

$$A : I_G(D_{rechts}) = 1 - \left( \left( \frac{1}{4} \right)^2 + \left( \frac{3}{4} \right)^2 \right) = \frac{3}{8} = 0.375$$

$$A : IG_G = 0.5 - \frac{4}{8} 0.375 - \frac{4}{8} 0.375 = 0.125$$

$$B : I_G(D_{links}) = 1 - \left( \left( \frac{2}{6} \right)^2 + \left( \frac{4}{6} \right)^2 \right) = \frac{4}{9} = 0.\bar{4}$$

$$B : I_G(D_{rechts}) = 1 - (1^2 + 0^2) = 0$$

$$B : IG_G = 0.5 - \frac{6}{8} 0.\bar{4} - 0 = 0.1\bar{6}$$

Und auch beim Kriterium der Entropie wäre das Szenario  $B (IG_H = 0.31)$  dem Szenario  $A (IG_H = 0.19)$  vorzuziehen:

$$I_H(D_p) = - (0.5 \log_2(0.5) + 0.5 \log_2(0.5)) = 1$$

$$A : I_H(D_{links}) = - \left( \frac{3}{4} \log_2\left(\frac{3}{4}\right) + \frac{1}{4} \log_2\left(\frac{1}{4}\right) \right) = 0.81$$

$$A : I_H(D_{rechts}) = - \left( \frac{1}{4} \log_2\left(\frac{1}{4}\right) + \frac{3}{4} \log_2\left(\frac{3}{4}\right) \right) = 0.81$$

$$A : IG_H = 1 - \frac{4}{8} 0.81 - \frac{4}{8} 0.81 = 0.19$$

$$B : I_H(D_{links}) = - \left( \frac{2}{6} \log_2\left(\frac{2}{6}\right) + \frac{4}{6} \log_2\left(\frac{4}{6}\right) \right) = 0.92$$

$$B : I_H(D_{rechts}) = 0$$

$$B : IG_H = 1 - \frac{6}{8} 0.92 - 0 = 0.31$$

Um die drei verschiedenen Unreinheitskriterien besser visuell vergleichen zu können, plotten wir nun die Unreinheitsindizes der Klasse 1 für das Wahrscheinlichkeitsintervall  $[0, 1]$ . Beachten Sie, dass wir zusätzlich eine skalierte Version der Entropie (`entropy/2`) darstellen, damit erkennbar wird, dass der Gini-Koeffizient ein Maß ist, das zwischen Entropie und Klassifizierungsfehler liegt. Hier der Code:

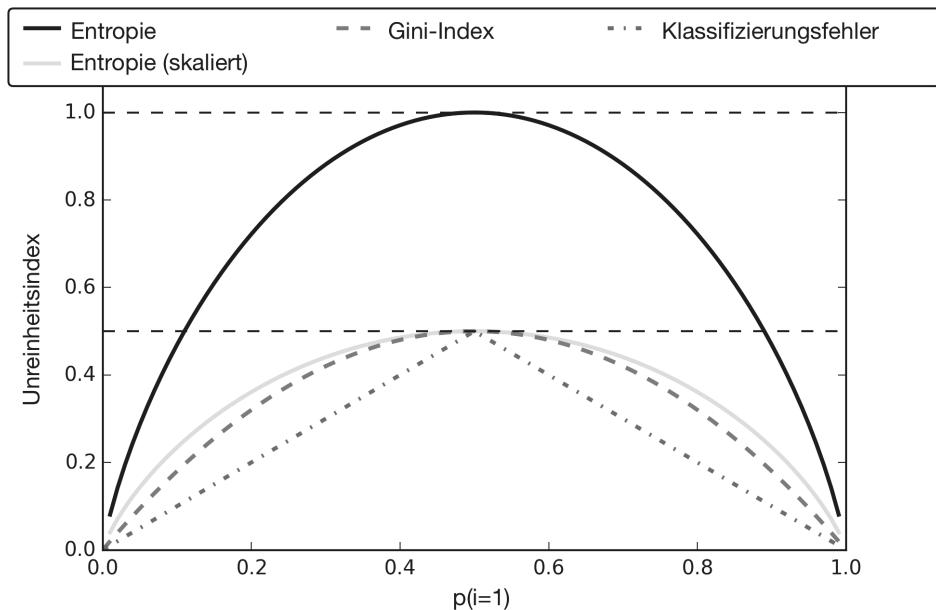
```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> def gini(p):
...     return (p)*(1 - (p)) + (1 - p)*(1 - (1-p))
>>> def entropy(p):
...     return - p*np.log2(p) - (1 - p)*np.log2((1 - p))
>>> def error(p):
...     return 1 - np.max([p, 1 - p])
>>> x = np.arange(0.0, 1.0, 0.01)
>>> ent = [entropy(p) if p != 0 else None for p in x]
>>> sc_ent = [e*0.5 if e else None for e in ent]
>>> err = [error(i) for i in x]
>>> fig = plt.figure()
>>> ax = plt.subplot(111)
```

```

>>> for i, lab, ls, c, in zip([ent, sc_ent, gini(x), err],
...                           ['Entropie', 'Entropie (skaliert)',
...                            'Gini-Koeffizient',
...                            'Klassifizierungsfehler'],
...                           [('-', '--', '---', '-.')],
...                           ['black', 'lightgray',
...                            'red', 'green', 'cyan']):
...     line = ax.plot(x, i, label=lab,
...                     linestyle=ls, lw=2, color=c)
>>> ax.legend(loc='upper center', bbox_to_anchor=(0.5, 1.15),
...             ncol=5, fancybox=True, shadow=False)
>>> ax.axhline(y=0.5, linewidth=1, color='k', linestyle='--')
>>> ax.axhline(y=1.0, linewidth=1, color='k', linestyle='--')
>>> plt.ylim([0, 1.1])
>>> plt.xlabel('p(i=1)')
>>> plt.ylabel('Unreinheitsindex')
>>> plt.show()

```

Und hier die Ausgabe:



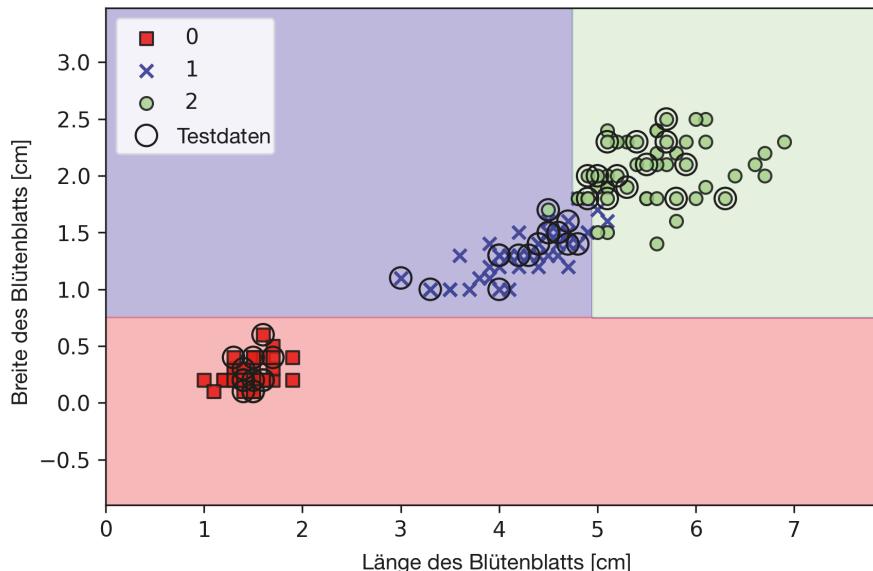
### 3.6.2 Konstruktion eines Entscheidungsbaums

Mit Entscheidungsbäumen lassen sich komplexe Entscheidungsgrenzen realisieren, indem der Merkmalsraum in Rechtecke aufgeteilt wird. Hier ist jedoch Vorsicht geboten, denn je tiefer der Entscheidungsbaum ist, desto komplizierter

werden die Entscheidungsgrenzen, was leicht zu einer Überanpassung führen kann. Wir werden nun mit scikit-learn einen Entscheidungsbaum der maximalen Tiefe 4 trainieren und dabei den Gini-Koeffizienten als Unreinheitskriterium verwenden. Zwecks Visualisierung kann es erwünscht sein, eine Standardisierung der Merkmale durchzuführen, erforderlich ist das für einen Entscheidungsbaumalgorithmus aber nicht. Hier der Code:

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> tree = DecisionTreeClassifier(criterion='gini',
...                                 max_depth=4, random_state=1)
>>> tree.fit(X_train, y_train)
>>> X_combined = np.vstack((X_train, X_test))
>>> y_combined = np.hstack((y_train, y_test))
>>> plot_decision_regions(X_combined, y_combined,
...                         classifier=tree, test_idx=range(105,150))
>>> plt.xlabel('Länge des Blütenblatts [cm]')
>>> plt.ylabel('Breite des Blütenblatts [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Nach Ausführung des Codes zeigen sich die typischen, parallel zu den Koordinatenachsen verlaufenden Entscheidungsgrenzen des Entscheidungsbaums:



In scikit-learn gibt es die Möglichkeit, einen Entscheidungsbaum nach dem Trainieren als .dot-Datei zu exportieren, die dann mit dem Programm *GraphViz* dargestellt werden kann. Dieses Programm steht unter <http://www.graphviz.org>

kostenlos zum Herunterladen bereit und unterstützt Linux, Windows und macOS. Neben GraphViz werden wir eine Python-Bibliothek namens `pydotplus` verwenden, die Ähnliches wie GraphViz leistet und es ermöglicht, .dot-Dateien in Bilddateien von Entscheidungsbäumen zu konvertieren. Nach der Installation von GraphViz (folgen Sie hierzu der Anleitung unter <http://www.graphviz.org/Download.php>) können Sie `pydotplus` direkt mit dem Programm `pip` installieren. Geben Sie auf der Kommandozeile Folgendes ein:

```
> pip3 install pydotplus
```

### Hinweis

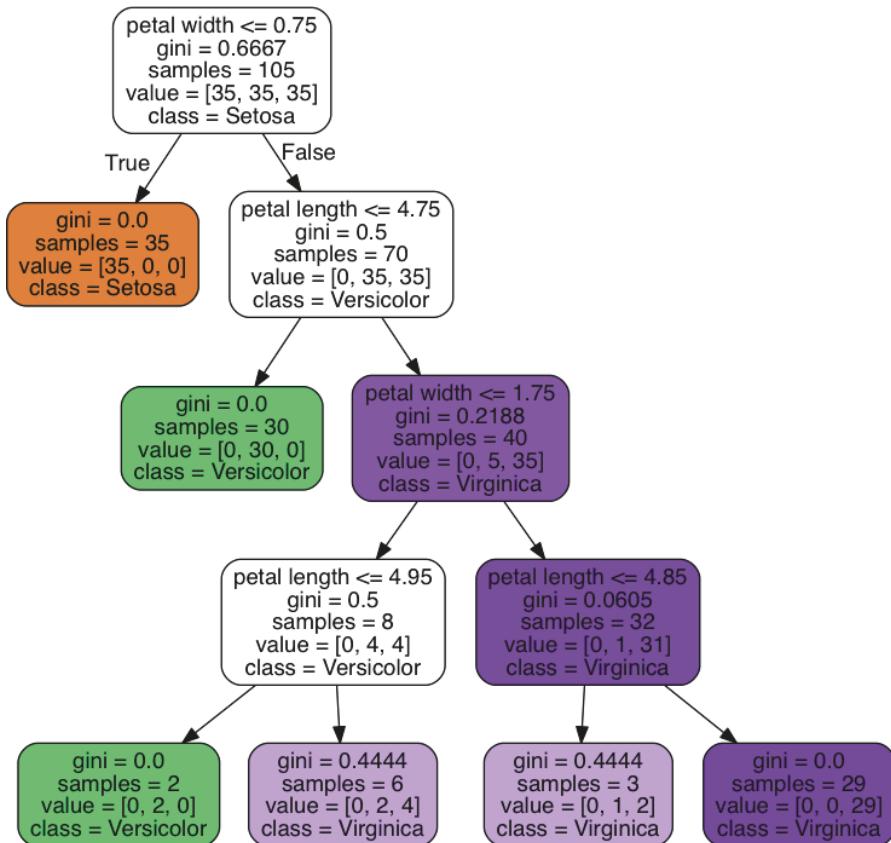
Auf manchen Systemen müssen Sie eventuell die für `pydotplus` erforderlichen Hilfsprogramme von Hand einrichten, indem Sie die folgenden Kommandos ausführen:

```
pip3 install graphviz  
pip3 install pyparsing
```

Der folgende Code erzeugt im lokalen Arbeitsverzeichnis ein Bild des Entscheidungsbäums im PNG-Format:

```
>>> from pydotplus import graph_from_dot_data
>>> from sklearn.tree import export_graphviz
>>> dot_data = export_graphviz(tree,
...                             filled=True,
...                             rounded=True,
...                             class_names=['Setosa',
...                                         'Versicolor',
...                                         'Virginica'],
...                             feature_names=['petal length',
...                                            'petal width'],
...                             out_file=None)
>>> graph = graph_from_dot_data(dot_data)
>>> graph.write_png('tree.png')
```

Durch die Einstellung `out_file=None` weisen wir die Daten direkt der Variablen `dot_data` zu, ohne sie in einer temporären Datei auf der Festplatte abzuspeichern. Die Argumente für `filled` (gefüllt), `rounded` (gerundet), `class_names` (Klassenbezeichnungen) und `feature_names` (Merkmalsbezeichnungen) sind optional, machen das resultierende Bild jedoch durch Farbe, runde Formen und Anzeige der Klassenbezeichnungen sowie der Merkmalsbezeichnungen an den Knoten visuell ansprechender. Diese Einstellungen liefern folgendes Bild des Entscheidungsbäums:



In der Darstellung des erstellten Entscheidungsbaums kann man nun genau zurückverfolgen, wo die Aufteilungen der Trainingsdaten vorgenommen werden. An der Wurzel gibt es 105 Exemplare, die anhand des Kriteriums »Länge des Blütenblatts  $\leq 0.75 \text{ cm}$ « in zwei Kindknoten mit 35 und 70 Exemplaren aufgeteilt werden. Nach der ersten Aufteilung ist der linke Kindknoten bereits unvermischt (Gini-Koeffizient = 0) und enthält nur noch Exemplare der Klasse *Iris setosa*. Die weiteren Aufteilungen auf der rechten Seite trennen dann die Exemplare der Klassen *Iris versicolor* und *Iris virginica* voneinander.

Wenn man diesen Baum und die Entscheidungsbereiche betrachtet, wird deutlich, dass der Entscheidungsbaum die Klassifizierung der Blumen sehr gut erledigt. Leider gibt es in scikit-learn derzeit noch keine Möglichkeit, einen Entscheidungsbau im Nachhinein zu stutzen. Wir könnten jedoch den schon vorhandenen Beispielcode verwenden, den Parameter `max_depth` auf den Wert 3 ändern und das Ergebnis mit dem jetzigen Modell vergleichen – probieren Sie es aus, wenn Sie daran interessiert sind.

### 3.6.3 Mehrere Entscheidungsbäume zu einem Random Forest kombinieren

*Random Forests* haben im vergangenen Jahrzehnt bei Anwendungen des Machine Learnings aufgrund ihrer guten Klassifizierungsfähigkeit, ihrer Skalierbarkeit und ihrer einfachen Handhabung sehr an Popularität gewonnen. Einen Random Forest kann man sich wie ein *Ensemble* von Entscheidungsbäumen vorstellen. Einem Random Forest liegt die Idee zugrunde, mehrere Entscheidungsbäume, die jeweils eine große Varianz aufweisen, zu einem stabileren Modell zu kombinieren, das beim Verallgemeinern besser arbeitet und weniger anfällig für eine Überanpassung ist. Der Random-Forest-Algorithmus kann in vier einfachen Schritten zusammengefasst werden:

1. Wählen Sie eine zufällige Stichprobe der Größe  $n$  aus der Trainingsdatenmenge aus (mit Ersetzung).
2. Konstruieren Sie anhand der Stichprobe einen Entscheidungsbaum. Führen Sie bei jedem Knoten folgende Schritte aus:
  1. Wählen Sie zufällig  $d$  Merkmale aus (ohne Ersetzung).
  2. Teilen Sie den Knoten an dem Merkmal auf, das hinsichtlich der Zielfunktion am besten geeignet ist (z.B. Maximierung des Informationsgewinns).
3. Wiederholen Sie die Schritte 1 und 2  $k$ -mal.
4. Fassen Sie die Vorhersagen der einzelnen Bäume durch eine *Mehrheitsentscheidung* (mehr dazu in Kapitel 7) zusammen, um die Klassenbezeichnung zuzuweisen.

Beachten Sie, dass wir im zweiten Schritt beim Training der einzelnen Entscheidungsbäume nicht alle Merkmale, sondern nur eine zufällige Untermenge berücksichtigen, um die beste Aufteilung zu finden.

#### Tipp

Falls Sie sich fragen, was mit »ohne« bzw. »mit Ersetzung« gemeint ist, hier ein einfaches Beispiel: Nehmen wir an, wir spielen eine Art Lotto und ziehen zufällige Zahlen aus einer Urne. Wir fangen mit einer Urne an, die fünf verschiedene Zahlen 0, 1, 2, 3 und 4 enthält, und ziehen in jeder Runde jeweils genau eine Zahl. In der ersten Runde beträgt die Wahrscheinlichkeit, eine bestimmte Zahl zu ziehen,  $1/5$ . Beim Ziehen ohne Ersetzung wird die gezogene Zahl der Urne nicht wieder hinzugefügt. Die Wahrscheinlichkeit, in der nächsten Runde eine bestimmte der verbliebenen Zahlen zu ziehen, hängt somit von der vorhergehenden Runde ab. Wenn beispielsweise die Zahlen 0, 1, 2 und 4 verblieben sind, beträgt die Wahrscheinlichkeit, in der nächsten Runde die Zahl 0 zu ziehen,  $1/4$ .

Beim Ziehen mit Ersetzung wird die gezogene Zahl dagegen nach jeder Runde wieder in die Urne zurückgelegt. Die Wahrscheinlichkeit, eine bestimmte Zahl zu ziehen, ändert sich in diesem Fall also nicht von Runde zu Runde – man kann sogar dieselbe Zahl mehrmals ziehen. Anders formuliert: Beim Ziehen mit Ersetzung sind die Stichproben (die Zahlen) voneinander unabhängig und die Kovarianz beträgt null. Die Ergebnisse von fünf Runden könnten beispielsweise folgendermaßen aussehen:

Ohne Ersetzung: 2, 1, 3, 4, 0

Mit Ersetzung: 1, 3, 3, 4, 1

Random Forests bieten zwar nicht das gleiche Maß an Interpretierbarkeit wie Entscheidungsbäume, haben aber den großen Vorteil, dass wir uns kaum Gedanken um die Auswahl geeigneter Hyperparameter machen müssen. Typischerweise brauchen wir den Random Forest nicht zurechtzustützen, denn das Ensemblemodell ist für das »Rauschen« einzelner Entscheidungsbäume kaum anfällig. Der einzige Parameter, um den wir uns in der Praxis kümmern müssen, ist die Anzahl  $k$  (Schritt 3) der Entscheidungsbäume, die wir für den Random Forest auswählen. Typischerweise arbeitet der Random-Forest-Klassifizierer umso zuverlässiger, aus je mehr Entscheidungsbäumen er besteht – allerdings zulasten einer erhöhten erforderlichen Rechenleistung.

In der Praxis ist es zwar weniger üblich, es gibt jedoch noch weitere Hyperparameter des Random Forests, die sich (mit Verfahren, die wir in Kapitel 5 erörtern werden) optimieren lassen: die Größe  $n$  der anfänglichen Stichprobe (Schritt 1) und die Anzahl  $d$  der Merkmale, die bei der Aufteilung (Schritt 2.1) zufällig ausgewählt werden. Über die Größe  $n$  der anfänglichen Stichprobe können wir das Bias-Varianz-Dilemma des Random Forests handhaben.

Durch die Wahl eines niedrigeren Werts für  $n$  können wir die Verschiedenartigkeit der einzelnen Bäume erhöhen, weil die Wahrscheinlichkeit, dass ein bestimmtes Objekt zur Stichprobe gehört, geringer ist. Ein kleinerer Wert für  $n$  kann die Zufälligkeit des Random Forests erhöhen und die Auswirkungen einer Überanpassung verringern. Allerdings führen geringere Werte für  $n$  typischerweise zu einer schlechteren Leistung des Random Forests. Der Unterschied zwischen Trainings- und Testdaten ist zwar gering, aber die Leistungsfähigkeit des Modells insgesamt ist schlechter. Umgekehrt kann ein größerer Wert für  $n$  das Ausmaß der Überanpassung erhöhen. Die in der Stichprobe vorhandenen Objekte sind einander ähnlicher und dementsprechend sind sich auch die Entscheidungsbäume ähnlicher und passen sich stärker an die ursprünglichen Trainingsdaten an.

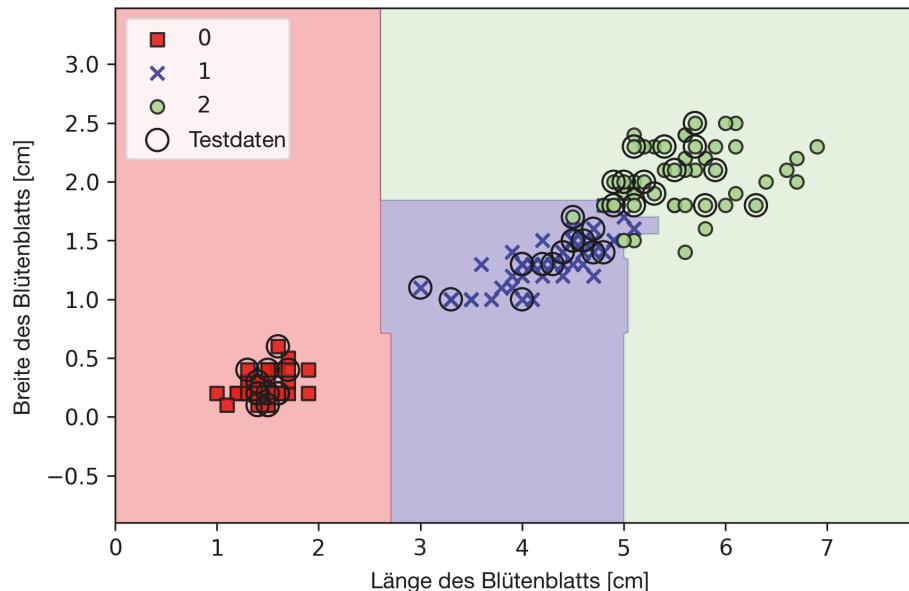
Bei den meisten Implementierungen, die `RandomForestClassifier`-Implementierung in scikit-learn eingeschlossen, wird als Größe der Stichprobe die Anzahl der Objekte in der ursprünglichen Trainingsdatenmenge gewählt, was für ge-

wöhnlich zu einem guten Bias-Varianz-Kompromiss führt. Für die Merkmalsanzahl  $d$  der Aufteilungen sollte ein Wert gewählt werden, der kleiner ist als die Gesamtzahl der Merkmale in der Trainingsdatenmenge. Ein vernünftiger Standardwert, der in scikit-learn und anderen Implementierungen verwendet wird, ist  $d = \sqrt{m}$ , wobei  $m$  die Anzahl der Merkmale in der Trainingsdatenmenge angibt.

Den Random-Forest-Klassifizierer müssen wir nicht selbst aus einzelnen Entscheidungsbäumen konstruieren, denn bequemerweise gibt es in scikit-learn bereits eine Implementierung, die wir nutzen können:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> forest = RandomForestClassifier(criterion='gini',
...                                 n_estimators=25,
...                                 random_state=1,
...                                 n_jobs=2)
>>> forest.fit(X_train, y_train)
>>> plot_decision_regions(X_combined, y_combined,
...                        classifier=forest, test_idx=range(105,150))
>>> plt.xlabel('Länge des Blütenblatts')
>>> plt.ylabel('Breite des Blütenblatts')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Nach der Ausführung des Codes sollten wie in der Abbildung die Entscheidungsbereiche des Entscheidungsbaum-Ensembles des Random Forests dargestellt werden:



Der Code trainiert einen aus 25 Entscheidungsbäumen bestehenden Random Forest (Parameter `n_estimators`) und verwendet zur Aufteilung der Knoten den Gini-Koeffizienten als Unreinheitskriterium. Wir konstruieren zwar einen sehr kleinen Random Forest anhand einer ebenso kleinen Trainingsdatenmenge, verwenden aber zu Demonstrationszwecken den Parameter `n_jobs`, der es ermöglicht, das Trainieren des Modells zu parallelisieren und mehrere Prozessorkerne des Computers (in diesem Fall zwei) zu verwenden.

### 3.7 k-Nearest-Neighbor: Ein Lazy-Learning-Algorithmus

Der letzte überwachte Lernalgorithmus, den wir in diesem Kapitel betrachten, ist der *k-Nearest-Neighbor-Algorithmus* (*KNN, k-Nächste-Nachbarn-Algorithmus*), der besonders interessant ist, weil er sich grundlegend von den bislang erörterten Lernalgorithmen unterscheidet.

KNN ist ein typisches Beispiel für einen *Lazy-Learning-Algorithmus* (»träges Lernen«). Er wird nicht wegen seiner offensichtlichen Einfachheit als »träger« bezeichnet, sondern weil die Modellbildung nicht durch das Lernen aus den Trainingsdaten erfolgt – stattdessen werden die Trainingsbeispiele einfach gespeichert.

#### Tipp

##### Parametrisierte kontra nichtparametrisierte Modelle

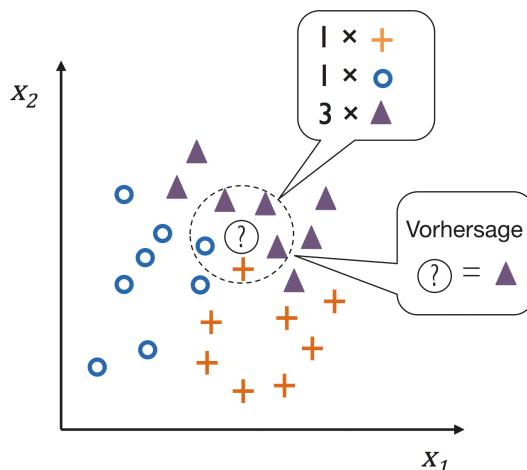
Lernalgorithmen können in parametrisierte und nichtparametrisierte Modelle unterteilt werden. Bei parametrisierten Modellen schätzen wir anhand der Trainingsdaten Parameter ab, um eine Diskriminanzfunktion herzuleiten, die neue Daten klassifizieren kann, ohne erneut auf die ursprünglichen Trainingsdaten zurückzugreifen. Typische Beispiele hierfür sind das Perzepron, die logistische Regression und die lineare SVM. Im Gegensatz dazu lassen sich nichtparametrisierte Modelle nicht durch einen festen Satz von Parametern beschreiben – die Anzahl der Parameter nimmt mit den Trainingsdaten zu. Der Entscheidungsbaum/Random Forest und die Kernel-SVM sind zwei Beispiele für nichtparametrisierte Modelle, denen wir schon begegnet sind.

Der KNN gehört zu einer Unterkategorie nichtparametrisierter Modelle, bei denen man von *instanzbasiertem Lernen* spricht. Darauf beruhende Modelle zeichnen sich dadurch aus, dass sie die Trainingsdaten speichern. Das Lazy Learning ist ein Spezialfall des instanzbasierten Lernens, bei dem mit dem Lernvorgang keinerlei Aufwand verbunden ist.

Der KNN-Algorithmus ist schnörkellos und kann folgendermaßen zusammengefasst werden:

1. Auswahl der Merkmale und Sammeln von Trainingsdaten
2. Auswahl der  $k$  nächsten Nachbarn des zu klassifizierenden Exemplars
3. Zuweisung der Klassenbezeichnung durch eine Mehrheitsentscheidung

Die nachstehende Abbildung illustriert, wie einem neuen Datenpunkt (?) anhand seiner fünf nächsten Nachbarn durch eine Mehrheitsentscheidung die Klassenbezeichnung »Dreieck« zugewiesen wird.



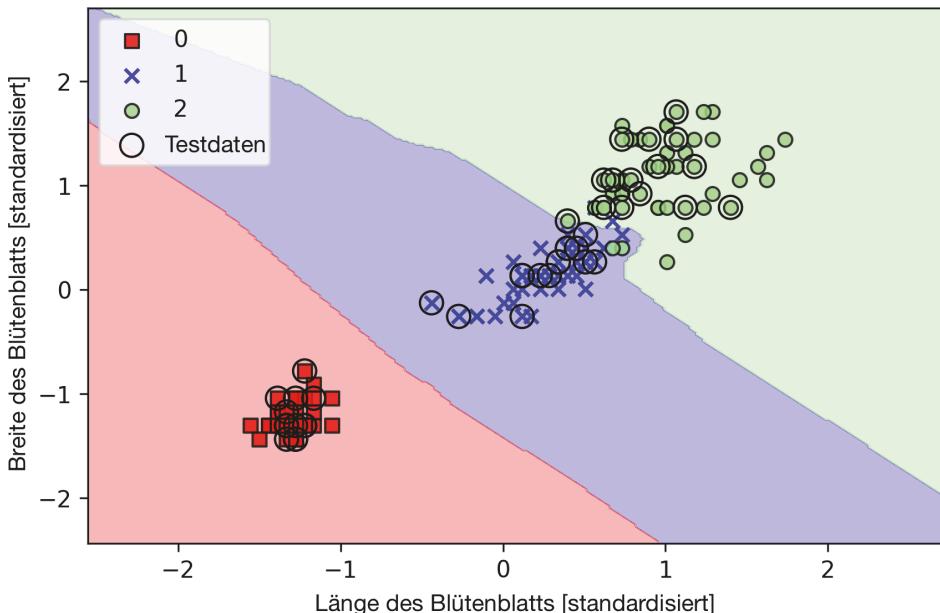
Anhand des ausgewählten Abstandsmaßes sucht der KNN-Algorithmus nach den  $k$  Exemplaren in der Trainingsdatenmenge, die dem zu klassifizierenden Punkt am nächsten bzw. am ähnlichsten sind. Die Klassenbezeichnung des neuen Wertes wird dann durch eine Mehrheitsentscheidung der  $k$  nächsten Nachbarn festgelegt.

Der Hauptvorteil solch eines speicherbasierten Ansatzes besteht darin, dass der Klassifizierer sich sofort anpasst, wenn neue Trainingsdaten hinzukommen. Es gibt jedoch auch den Nachteil, dass der für die Klassifizierung neuer Exemplare erforderliche Rechenaufwand schlimmstenfalls linear mit der Anzahl der in der Trainingsdatenmenge vorhandenen Exemplare ansteigt – es sei denn, dass die Datenmenge nur einige wenige Dimensionen (Merkmale) besitzt und der Algorithmus mittels effizienter Datenstrukturen wie k-d-Bäumen implementiert wird (J.H. Friedman, J.L. Bentley und R.A. Finkel: *An algorithm for finding best matches in logarithmic expected time*. ACM Transactions on Mathematical Software (TOMS), 3(3):209-226, 1977). Darüber hinaus können wir keine Trainingsdaten verwerfen, weil es kein Training gibt, daher kann der Speicherplatz zu einem Problem werden, wenn wir es mit großen Datenmengen zu tun haben.

Mit dem folgenden Code werden wir nun mit scikit-learn ein KNN-Modell implementieren, das einen euklidischen Abstand verwendet:

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> knn = KNeighborsClassifier(n_neighbors=5, p=2,
...                             metric='minkowski')
>>> knn.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std, y_combined,
...                         classifier=knn, test_idx=range(105,150))
>>> plt.xlabel('Länge des Blütenblatts [standardisiert]')
>>> plt.ylabel('Breite des Blütenblatts [standardisiert]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Da wir im KNN-Modell für diese Datensammlung fünf Nachbarn angeben, erhalten wir eine relativ glatt verlaufende Entscheidungsgrenze, wie die Abbildung zeigt:



### Tipp

Sollte es bei der Mehrheitsentscheidung zu einem Gleichstand kommen, bevorzugt die Implementierung des KNN-Algorithmus in scikit-learn diejenigen Nachbarn, deren Abstand zum zu klassifizierenden Objekt geringer ist. Sind die Nachbarn ebenfalls gleich weit entfernt, wählt der Algorithmus diejenige Klassenbezeichnung aus, die in den Trainingsdaten zuerst erscheint.

Die richtige Wahl von  $k$  ist entscheidend, um ein vernünftiges Gleichgewicht zwischen Über- und Unteranpassung zu finden. Wir müssen außerdem gewährleisten, dass wir eine für die in der Datensammlung enthaltenen Merkmale geeignete Metrik verwenden. Für reellwertige Daten, wie beispielsweise die in Zentimetern angegebenen Merkmale der Iris-Datensammlung, wird häufig ein einfacher euklidischer Abstand verwendet. Wenn wir eine euklidische Metrik benutzen, müssen wir die Daten allerdings auch standardisieren, damit jedes Merkmal in gleichem Maße zum Abstand beiträgt. Die im Code verwendete Minkowski-Metrik ist eine Verallgemeinerung der euklidischen und der Manhattan-Metrik, die wie folgt formuliert werden kann:

$$d(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \sqrt[p]{\sum_k |x_k^{(i)} - x_k^{(j)}|^p}$$

Daraus wird der euklidische Abstand oder die Manhattan-Metrik, wenn wir den Parameter  $p=2$  bzw.  $p=1$  setzen. In scikit-learn ist eine ganze Reihe weiterer Metriken verfügbar, die mit dem Parameter `metric` übergeben werden können. Eine vollständige Liste finden Sie unter <http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.DistanceMetric.html>.

### Tipp

#### Der Fluch der Dimensionalität

An dieser Stelle muss erwähnt werden, dass KNN aufgrund des *Fluchs der Dimensionalität* sehr anfällig für eine Überanpassung ist. Dieses Phänomen beschreibt, dass der Merkmalsraum einer Trainingsdatenmenge fester Größe zunehmend dünner besetzt ist, wenn die Anzahl der Dimensionen ansteigt. Zur besseren Veranschaulichung kann man sich vorstellen, dass selbst die engsten Nachbarn in einem hochdimensionalen Raum zu weit voneinander entfernt sind, um eine gute Abschätzung vornehmen zu können.

Im Abschnitt über die logistische Regression haben wir die Regularisierung als eine Möglichkeit erörtert, eine Überanpassung zu verhindern. Bei Modellen, in denen eine Regularisierung nicht anwendbar ist, wie etwa Entscheidungsbäumen oder KNN-Algorithmen, können wir Verfahren zur Merkmalsauswahl und Dimensionsreduktion einsetzen, die uns dabei helfen, dem Fluch der Dimensionalität zu entgehen. Mehr dazu erfahren Sie im nächsten Kapitel.

## 3.8 Zusammenfassung

In diesem Kapitel haben Sie viele verschiedene Algorithmen kennengelernt, mit denen Sie lineare und nichtlineare Aufgabenstellungen in Angriff nehmen können. Entscheidungsbäume sind besonders interessant, wenn die Interpretierbarkeit von Bedeutung ist. Die logistische Regression ist nicht nur eine nützliche Methode, um mit dem stochastischen Gradientenabstiegsverfahren Online Learning zu realisieren, vielmehr ermöglicht es auch, die Wahrscheinlichkeit eines bestimmten Ereignisses vorherzusagen. Support Vector Machines sind leistungsfähige lineare Modelle, die sich mit dem Kernel-Trick auch auf nichtlineare Aufgabenstellungen anwenden lassen, sie greifen jedoch auf sehr viele Parameter zurück, die fein abgestimmt werden müssen, um gute Vorhersagen treffen zu können. Im Gegensatz dazu ist bei Ensemblemethoden wie Random Forests weniger Feinabstimmung der Parameter erforderlich und es kommt nicht so schnell zu einer Überanpassung wie bei Entscheidungsbäumen, wodurch sie für viele praktische Anwendungsfälle interessant sind. Der k-Nearest-Neighbor-Klassifizierer bietet einen alternativen Ansatz zur Klassifizierung mittels Lazy Learning, der es uns zwar ermöglicht, ohne vorheriges Trainieren des Modells Vorhersagen zu treffen, aber auch sehr rechenaufwendig ist.

Noch wichtiger als die Auswahl eines geeigneten Lernalgorithmus ist die Qualität der Trainingsdatenmenge, denn kein Algorithmus ist in der Lage, ohne informative und aussagekräftige Merkmale gute Vorhersagen zu treffen.

Im nächsten Kapitel werden wir wichtige Themen hinsichtlich der Vorverarbeitung von Daten, der Merkmalsauswahl sowie der Dimensionsreduktion erörtern, auf die wir bei der Entwicklung leistungsfähiger Lernmodelle zurückgreifen. Und später, in Kapitel 6, werden wir uns damit befassen, wie wir die Leistung unserer Modelle bewerten und vergleichen können sowie nützliche Tricks zur Feinabstimmung verschiedener Algorithmen kennenlernen.



# Gut geeignete Trainingsdatenmengen: Datenvorverarbeitung

Die Qualität der Daten sowie der Umfang der darin enthaltenen Informationen sind entscheidende Faktoren, die festlegen, wie gut ein Lernalgorithmus daraus lernen kann. Daher ist es absolut unverzichtbar, eine Datensammlung zu untersuchen und vorzuverarbeiten, bevor wir einen Lernalgorithmus damit füttern. In diesem Kapitel werden wir wichtige Verfahren zur Datenvorverarbeitung erörtern, die es ermöglichen sollen, gut funktionierende Lernmodelle zu entwickeln.

Die Themen in diesem Kapitel:

- Unvollständige Werte aus der Datensammlung entfernen und fehlende Werte zuweisen
- Aufbereitung kategorialer Daten für den Lernalgorithmus
- Auswahl der für die Modellentwicklung maßgeblichen Merkmale

## 4.1 Umgang mit fehlenden Daten

Bei praktischen Anwendungen kommt es nicht selten vor, dass in der Stichprobe aus den verschiedensten Gründen einige Werte fehlen. Beispielsweise könnte es beim Sammeln der Daten zu Fehlern gekommen sein, vielleicht sind manche Messwerte nicht anwendbar, oder bestimmte Datenfelder wurden bei einer Befragung einfach nicht ausgefüllt (Antwortausfälle). Typischerweise erscheinen fehlende Werte in den Datentabellen als leere Stellen oder sind durch einen Platzhalter wie NaN (*Not a Number*, keine Zahl) oder NULL (ein in relationalen Datenbanken oft anzutreffender Wert) ersetzt.

Leider kommen die meisten Tools zur Datenauswertung mit solchen fehlenden Werten nicht zurecht oder liefern unvorhersehbare Ergebnisse, wenn man das Fehlen der Daten einfach ignoriert. Aus diesem Grund müssen wir uns um die fehlenden Werte kümmern, bevor wir mit der Datenanalyse fortfahren. Im Folgenden werden wir einige Verfahren zur Handhabung fehlender Daten erörtern. In diesem Kapitel werden wir verschiedene praktische Verfahren zur Handhabung fehlender Daten betrachten, die fehlende Werte entfernen oder durch vermutete Werte ersetzen.

### 4.1.1 Fehlende Werte in Tabellendaten

Um die Problematik besser verstehen zu können, erstellen wir zunächst einmal ein paar einfache Beispieldaten in Form einer *CSV-Datei* (*Comma Separated Values*, durch Kommas getrennte Werte):

```
>>> import pandas as pd
>>> from io import StringIO
>>> csv_data = \
...     '''A,B,C,D
... 1.0,2.0,3.0,4.0
... 5.0,6.0,,8.0
... 10.0,11.0,12.0,''' 
>>> # Falls Sie Python 2.7 verwenden, müssen
>>> # Sie den String in Unicode konvertieren:
>>> # csv_data = unicode(csv_data)
>>> df = pd.read_csv(StringIO(csv_data))
>>> df
   A      B      C      D
0  1.0    2.0    3.0    4.0
1  5.0    6.0    NaN    8.0
2 10.0   11.0   12.0   NaN
```

Der Code liest die CSV-formatierten Daten mit der `read_csv`-Funktion in einen pandas-DataFrame ein. Beachten Sie, dass die beiden fehlenden Werte durch `NaN` ersetzt wurden. Die Funktion `StringIO` soll lediglich demonstrieren, wie man den `csv_data` zugewiesenen String so in einen pandas-DataFrame einlesen kann, als ob es sich um eine auf der Festplatte gespeicherte CSV-Datei handeln würde.

Bei einem größeren DataFrame ist es aufwendig, nach fehlenden Werten zu suchen. In diesem Fall können wir die `isnull`-Methode verwenden, die einen DataFrame mit booleschen Werten zurückgibt, an denen sich ablesen lässt, ob eine Zelle einen numerischen Wert (`False`) enthält oder ob Daten fehlen (`True`). Mit der `sum`-Methode können wir dann wie folgt die Anzahl der fehlenden Werte pro Spalte anzeigen:

```
>>> df.isnull().sum()
A    0
B    0
C    1
D    1
dtype: int64
```

Auf diese Weise können wir also die Anzahl der fehlenden Werte pro Spalte ermitteln. In den nächsten Abschnitten werden wir verschiedene Strategien zum Umgang mit fehlenden Daten betrachten.

**Tipp**

Zwar ist scikit-learn dafür ausgelegt, NumPy-Arrays zu bearbeiten, dennoch kann es gelegentlich komfortabler sein, Daten mit pandas `DataFrame` vorzuverarbeiten. Auf das NumPy-Array kann über das `values`-Attribut des `DataFrames` zugegriffen werden, wenn es einem scikit-learn-Schätzer übergeben werden soll:

```
>>> df.values
array([[ 1.,   2.,   3.,   4.],
       [ 5.,   6.,   nan,   8.],
       [10.,  11.,  12.,   nan]])
```

### 4.1.2 Exemplare oder Merkmale mit fehlenden Daten entfernen

Am einfachsten ist es, Merkmale (Spalten) oder Exemplare (Zeilen) mit fehlenden Daten komplett aus der Datensammlung zu entfernen. Zeilen mit fehlenden Daten können ganz einfach mit der `dropna`-Methode aussortiert werden:

```
>>> df.dropna(axis=0)
      A      B      C      D
0  1.0  2.0  3.0  4.0
```

Spalten, in denen sich mindestens ein `NaN` befindet, lassen sich hingegen entfernen, indem man das `axis`-Argument auf 1 setzt:

```
>>> df.dropna(axis=1)
      A      B
0  1.0  2.0
1  5.0  6.0
2 10.0 11.0
```

Die `dropna`-Methode unterstützt überdies noch einige weitere praktische Parameter:

```
# Nur Zeilen löschen, in denen alle Spalten NaN enthalten
# (Gibt das gesamte Array zurück, da es keine Zeilen gibt,
# die nur NaN-Werte enthalten)
>>> df.dropna(how='all')
      A      B      C      D
0  1.0  2.0  3.0  4.0
1  5.0  6.0  NaN  8.0
2 10.0 11.0 12.0  NaN
# Zeilen entfernen, die weniger als 4 reelle Zahlen enthalten
>>> df.dropna(thresh=4)
```

```

      A      B      C      D
0  1.0   2.0   3.0   4.0
# Nur Zeilen löschen, in denen NaN in einer bestimmten
# Spalte (hier: 'C') enthalten ist
>>> df.dropna(subset=['C'])
      A      B      C      D
0  1.0   2.0   3.0   4.0
2 10.0  11.0  12.0   NaN

```

Das Entfernen von Spalten oder Zeilen mit fehlenden Daten ist zwar ein sehr bequemer Ansatz, allerdings hat er auch gewisse Nachteile – beispielsweise könnten unter Umständen so viele Objekte entfernt werden, dass eine verlässliche Analyse unmöglich wird. Oder wenn wir zu viele Merkmalsspalten entfernen, laufen wir Gefahr, wertvolle Informationen zu verlieren, die der Klassifizierer benötigt, um die Klassen voneinander zu unterscheiden. Im nächsten Abschnitt werden wir daher eine der gängigsten Möglichkeiten zur Handhabung fehlender Werte betrachten: Interpolationsverfahren.

### 4.1.3 Fehlende Werte ergänzen

Häufig ist das Entfernen einzelner Objekte oder sogar ganzer Merkmalsspalten nicht praktikabel, weil wir dadurch zu viele wertvolle Daten verlieren würden. In diesem Fall können wir verschiedene Interpolationsverfahren einsetzen, um die fehlenden Werte anhand der anderen in den Trainingsdaten vorhandenen Werte abzuschätzen. Eines der gebräuchlichsten Interpolationsverfahren ist die *Mittelwert-imputation*, bei der wir den fehlenden Wert einfach durch den Mittelwert der gesamten Merkmalsspalte ersetzen. Die `Imputer`-Klasse von scikit-learn bietet eine komfortable Möglichkeit, dies zu erreichen:

```

>>> from sklearn.preprocessing import Imputer
>>> imr = Imputer(missing_values='NaN',
...                  strategy='mean', axis=0)
>>> imr = imr.fit(df.values)
>>> imputed_data = imr.transform(df.values)
>>> imputed_data
array([[ 1.,   2.,   3.,   4.],
       [ 5.,   6.,   7.5,   8.],
       [10.,  11.,  12.,   6.]])

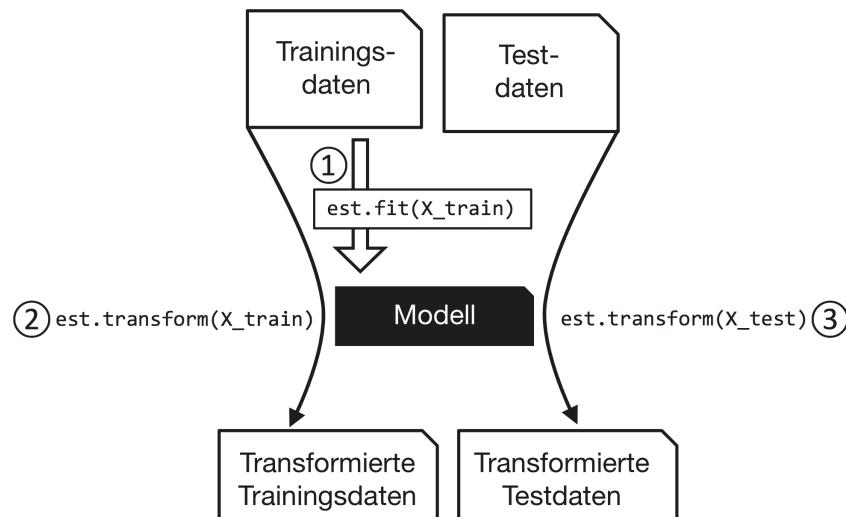
```

Hier wird jeder `NaN`-Wert durch den zugehörigen Mittelwert ersetzt, der für jede Merkmalsspalte separat berechnet wird. Wenn wir die Einstellung `axis=0` auf `axis=1` ändern, werden die Mittelwerte der Zeilen berechnet. Weitere Optionen für den `strategy`-Parameter sind `median` (Medianwert) oder `most_frequent` (häufigster Wert). Bei Letzterem werden die fehlenden Werte durch den häufigs-

ten Wert ersetzt, was sich bei der Ergänzung von kategorialen Merkmalswerten als nützlich erweist, beispielsweise einer Merkmalsspalte, die Farbbezeichnungen wie Rot, Grün oder Blau speichert. Wir werden später in diesem Kapitel noch ein Beispiel kennenlernen.

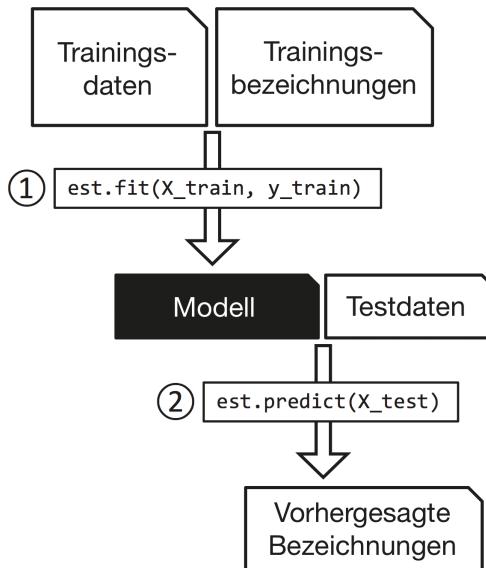
#### 4.1.4 Die Schätzer-API von scikit-learn

Im vorangegangenen Abschnitt haben wir die `Imputer`-Klasse von scikit-learn genutzt, um in der Datensammlung fehlende Werte zu ergänzen. Die `Imputer`-Klasse gehört zu den sogenannten *Transformer-Klassen* von scikit-learn, die zur Transformation von Daten dienen. Die beiden wichtigsten Methoden dieser Schätzer sind `fit` und `transform`: Die `fit`-Methode dient dazu, die Parameter aus den Trainingsdaten zu erlernen, und die `transform`-Methode nutzt diese Parameter, um die Daten zu transformieren. Ein zu transformierendes Datenarray muss die gleiche Anzahl an Merkmalen aufweisen wie dasjenige, das zur Anpassung an das Modell verwendet wurde. Die folgende Abbildung zeigt, wie ein an die Trainingsdaten angepasster Transformer eingesetzt wird, um sowohl eine Trainingsdatenmenge als auch eine neue Testdatenmenge zu transformieren:



Die in Kapitel 3 benutzten Klassifizierer gehören zu den sogenannten Schätzern von scikit-learn, die eine API besitzen, die konzeptionell derjenigen einer Transformer-Klasse sehr ähnlich ist. Schätzer verfügen über eine `predict`-Methode, unter Umständen aber auch über eine `transform`-Methode, wie Sie später noch sehen werden. Wir hatten beim Trainieren der Schätzer für die Klassifizierung bereits die `fit`-Methode verwendet, um die Parameter eines Modells zu erlernen. Beim überwachten Lernen stellen wir allerdings zusätzlich die Klassenbezeichnungen zur Verfügung, wenn das Modell angepasst wird. Diese können dann mit-

tels der `predict`-Methode genutzt werden, um Vorhersagen über neue Daten zu treffen (siehe Abbildung).



## 4.2 Handhabung kategorialer Daten

Bislang haben wir nur numerische Werte verwendet. Allerdings gibt es in echten Datensammlungen des Öfteren kategoriale Merkmalsspalten. In diesem Abschnitt werden wir einige einfache, aber aussagekräftige Beispiele betrachten, wie man diese Art von Daten mit numerischen Bibliotheken handhabt.

### 4.2.1 Nominale und ordinale Merkmale

Wir müssen bei kategorialen Daten zwischen *nominalen* (qualitativen oder bezeichnenden) und *ordinalen* (ordnenden) Merkmalen unterscheiden. *Ordinal* Merkmale sind kategoriale Werte, die sortiert oder geordnet werden können – beispielsweise wäre *T-Shirt-Größe (size)* ein Ordnungsmerkmal, denn wir können die Reihenfolge XL > L > M festlegen. Im Gegensatz dazu weisen *nominale Merkmale* keine Ordnung auf. Um bei dem Beispiel zu bleiben: Die *T-Shirt-Farbe (color)* wäre ein ordinales Merkmal, denn es ergibt normalerweise keinen Sinn zu behaupten, dass beispielsweise Rot (*red*) größer sei als Blau (*blue*) oder kleiner als Grün (*green*).

### 4.2.2 Erstellen einer Beispieldatenmenge

Bevor wir uns mit den verschiedenen Verfahren zur Handhabung solcher kategorialen Daten befassen, erstellen wir wieder ein paar Beispieldaten, um die Proble-

matik zu veranschaulichen. Neben Farbe und Größe kommt jetzt noch der Preis (**price**) hinzu:

```
>>> import pandas as pd
>>> df = pd.DataFrame([
...     ['green', 'M', 10.1, 'class1'],
...     ['red', 'L', 13.5, 'class2'],
...     ['blue', 'XL', 15.3, 'class1']])
>>> df.columns = ['color', 'size', 'price',
...                 'classlabel']
>>> df
   color  size  price  classlabel
0  green     M    10.1      class1
1    red     L    13.5      class2
2   blue    XL    15.3      class1
```

Wie Sie der Ausgabe entnehmen können, enthält der neu erstellte `DataFrame` ein nominales Merkmal (`color`), ein ordinales Merkmal (`size`) und ein numerisches Merkmal (`price`). Die Klassenbezeichnungen (`classlabel`) werden in der letzten Spalte gespeichert – vorausgesetzt, wir erstellen eine Datenmenge für überwachtes Lernen. Die in diesem Buch erörterten Klassifizierungsalgorithmen verwenden keine ordinalen Merkmale als Klassenbezeichnungen.

### 4.2.3 Zuweisung von ordinalen Merkmalen

Um zu gewährleisten, dass der Lernalgorithmus die ordinalen Merkmale korrekt interpretiert, müssen wir die kategorialen Bezeichnungen in Ganzzahlen konvertieren. Da es leider keine Funktion gibt, um die Bezeichnungen des Merkmals `size` automatisch in die richtige Reihenfolge zu bringen, erledigen wir das von Hand. In dem folgenden einfachen Beispiel gehen wir davon aus, dass wir die Unterschiede zwischen den Merkmalen kennen, beispielsweise dass `XL` = `L+1` = `M+2` gilt.

```
>>> size_mapping = {
...             'XL': 3,
...             'L': 2,
...             'M': 1}
>>> df['size'] = df['size'].map(size_mapping)
>>> df
   color  size  price  classlabel
0  green     1    10.1      class1
1    red     2    13.5      class2
2   blue     3    15.3      class1
```

Wenn wir die Ganzzahlen später wieder in die ursprünglichen Bezeichnungen umwandeln möchten, können wir einfach ein Dictionary

```
inv_size_mapping = {v: k for k, v in size_mapping.items()}
```

definieren, das dann mittels pandas `map`-Methode die transformierte Merkmalspalte auf dieselbe Art und Weise umwandelt, wie wir es mit dem vorher verwendeten `size_mapping`-Dictionary getan haben. Wir können es wie folgt verwenden:

```
>>> inv_size_mapping = {v: k for k,
...                      v in size_mapping.items()}
>>> df['size'].map(inv_size_mapping)
0      M
1      L
2     XL
Name: size, dtype: object
```

#### 4.2.4 Codierung der Klassenbezeichnungen

Viele Bibliotheken für Machine Learning machen es erforderlich, dass die Klassenbezeichnungen als Ganzzahlen codiert sind. Die meisten scikit-learn-Schätzer zur Klassifizierung konvertieren sie zwar intern in Ganzzahlen, dennoch gilt es als guter Stil, sie als Integer-Arrays bereitzustellen, um technischen Pannen vorzubeugen. Zur Codierung der Klassenbezeichnungen können wir einen ähnlichen Ansatz wie bei der vorhin erörterten Zuweisung von ordinalen Merkmalen verwenden. Klassenbezeichnungen sind *nicht* geordnet, daher spielt es keine Rolle, welche Zahl wir einer bestimmten Bezeichnung zuweisen. Wir können sie also einfach, angefangen bei 0, durchnummerieren:

```
>>> import numpy as np
>>> class_mapping = {label:idx for idx,label in
...                     enumerate(np.unique(df['classlabel']))}
>>> class_mapping
{'class1': 0, 'class2': 1}
```

Nun können wir das Zuweisungs-Dictionary verwenden, um die Klassenbezeichnungen in Ganzzahlen zu transformieren:

```
>>> df['classlabel'] = df['classlabel'].map(class_mapping)
>>> df
   color  size  price  classlabel
0  green     1   10.1          0
1    red     2   13.5          1
2   blue     3   15.3          0
```

Wir können die Schlüssel-Werte-Paare im Zuweisungs-Dictionary wie folgt umkehren, um den konvertierten Klassenbezeichnungen wieder die Repräsentationen als Zeichenketten zuzuordnen:

```
>>> inv_class_mapping = {v: k for k, v in
...                         class_mapping.items()}
>>> df['classlabel'] =
...                 df['classlabel'].map(inv_class_mapping)
>>> df
   color  size  price classlabel
0  green     1    10.1    class1
1    red     2    13.5    class2
2   blue     3    15.3    class1
```

Eine weitere Möglichkeit besteht darin, die komfortable `LabelEncoder`-Klasse von scikit-learn zu verwenden, um dasselbe Resultat zu erzielen:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> class_le = LabelEncoder()
>>> y = class_le.fit_transform(df['classlabel'].values)
>>> y
array([0, 1, 0])
```

Beachten Sie hier, dass die `fit_transform`-Methode lediglich eine Abkürzung für den getrennten Aufruf von `fit` und `transform` ist. Mit der `inverse_transform`-Methode können die ganzzahligen Klassenbezeichnungen wieder in die ursprünglichen Zeichenketten umgewandelt werden.

```
>>> class_le.inverse_transform(y)
array(['class1', 'class2', 'class1'], dtype=object)
```

#### 4.2.5 One-hot-Codierung der nominalen Merkmale

Im letzten Abschnitt haben wir ein einfaches Zuweisungs-Dictionary verwendet, um das ordinale Merkmal `size` in Ganzzahlen umzuwandeln. Da die Schätzer von scikit-learn die Klassenbezeichnungen als kategoriale Daten behandeln, die ungeordnet sind, haben wir die `LabelEncoder`-Klasse benutzt, um die Bezeichnungen in Ganzzahlen umzuwandeln. Nun könnte man auf den Gedanken kommen, einen ähnlichen Ansatz zu verfolgen, um die Spalte mit dem nominalen Merkmal `color` umzuwandeln:

```
>>> X = df[['color', 'size', 'price']].values
>>> color_le = LabelEncoder()
>>> X[:, 0] = color_le.fit_transform(X[:, 0])
```

```
>>> X
array([[1, 1, 10.1],
       [2, 2, 13.5],
       [0, 3, 15.3]], dtype=object)
```

Nach der Ausführung des Codes enthält die erste Spalte des NumPy-Arrays `X` die neuen `color`-Werte, die folgendermaßen codiert sind:

- `blue = 0`
- `green = 1`
- `red = 2`

Wenn wir es hierbei belassen und dem Klassifizierer dieses Array übergeben würden, würden wir einen der häufigsten Fehler beim Umgang mit kategorialen Daten begehen. Erkennen Sie das Problem? Obwohl die `color`-Werte überhaupt nicht geordnet sind, würde ein Lernalgorithmus nun annehmen, dass `green` größer ist als `blue` und dass `red` größer ist als `green`. Diese Annahme ist zwar falsch, dennoch könnte der Algorithmus nützliche Ergebnisse liefern, die jedoch nicht optimal wären.

Um dieses Problem zu umgehen, wird häufig ein als *One-hot-Codierung* bezeichnetes Verfahren eingesetzt. Diesem Verfahren liegt die Idee zugrunde, für jeden der verschiedenen möglichen Werte der Merkmalsspalte ein neues *Dummy-Merkmal* zu erstellen. Hier würden wir für `color` die drei neuen Merkmale `blue`, `green` und `red` erstellen, um die Farbe eines bestimmten Exemplars binär zu beschreiben – ein blaues Exemplar könnte beispielsweise als `blue=1, green=0, red=0` codiert werden. Zur Durchführung dieser Transformation können wir die `OneHotEncoder`-Klasse verwenden, die im `sklearn.preprocessing`-Modul implementiert ist:

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> ohe = OneHotEncoder(categorical_features=[0])
>>> ohe.fit_transform(X).toarray()
array([[ 0.,  1.,  0.,  1., 10.1],
       [ 0.,  0.,  1.,  2., 13.5],
       [ 1.,  0.,  0.,  3., 15.3]])
```

Bei der Initialisierung des `OneHotEncoder`s geben wir über den Parameter `categorical_feature` die Position der Spalte an, die wir transformieren möchten (`color` ist die erste Spalte in der Merkmalsmatrix `X`). Standardmäßig liefert der Aufruf der `transform`-Methode des `OneHotEncoder`s eine dünnbesetzte Matrix zurück, die mittels der `toarray`-Methode zum Zweck der Visualisierung in ein reguläres (vollbesetztes) NumPy-Array umgewandelt wird. Dünnbesetzte Matrizen stellen eine effektivere Methode zum Speichern großer Datensammlungen

bereit und werden von vielen scikit-learn-Funktionen unterstützt. Das erweist sich als besonders nützlich, wenn ein Array sehr viele Nullen enthält. Wir hätten die Umwandlung mit `toarray` auch weglassen und die Initialisierung stattdessen mit `OneHotEncoder(..., sparse=False)` vornehmen können, um ein normales NumPy-Array zu erhalten.

Eine noch komfortablere Möglichkeit zum Erstellen der Dummy-Merkmale per One-hot-Codierung bietet die in pandas implementierte `get_dummies`-Methode. Wendet man sie auf einen `DataFrame` an, konvertiert sie ausschließlich Spalten, die Zeichenketten enthalten – die übrigen Spalten bleiben unangetastet:

```
>>> pd.get_dummies(df[['price', 'color', 'size']])
   price  size  color_blue  color_green  color_red
0    10.1     1          0           1           0
1    13.5     2          0           0           1
2    15.3     3          1           0           0
```

Bei der Verwendung One-hot-codierter Datenmengen muss berücksichtigt werden, dass damit ein großes Maß an Kollinearität (hohe Korrelation der Merkmale) verbunden ist, was bei manchen Methoden problematisch sein kann (z.B. Methoden, die eine invertierte Matrix benötigen). Wenn Merkmale hochgradig korrelieren, ist die Berechnung der invertierten Matrix schwierig, was zu numerisch instabilen Abschätzungen führen kann. Um die Korrelation zwischen den Variablen zu verringern, können wir einfach eine Merkmalsspalte aus dem One-hot-codierten Array entfernen. Dadurch verlieren wir keinerlei wichtige Informationen, denn wenn wir z.B. die Spalte `color_blue` entfernen, sind ja immer noch die Spalten `color_green=0` und `color_red=0` vorhanden, was zwangsläufig `color_blue=1` bedeutet.

Bei der Verwendung der `get_dummies`-Funktion kann die erste Spalte durch Übergabe des Parameters `drop_first=True` entfernt werden, wie das Codebeispiel zeigt:

```
>>> pd.get_dummies(df[['price', 'color', 'size']],
...                  drop_first=True)
   price  size  color_green  color_red
0    10.1     1          1           0
1    13.5     2          0           1
2    15.3     3          0           0
```

Der `OneHotEncoder` bietet keinen Parameter zum Spaltenlöschen, aber wir können einen Slice des One-hot-codierten NumPy-Arrays verwenden, so wie hier:

```
ohe = OneHotEncoder(categorical_features=[0])
ohe.fit_transform(X).toarray()[:, 1:]
array([[ 1. ,  0. ,  1. , 10.1],
       [ 0. ,  1. ,  2. , 13.5],
       [ 0. ,  0. ,  3. , 15.3]])
```

## 4.3 Aufteilung einer Datensammlung in Trainings- und Testdaten

In den Kapiteln 1 und 3 wurde das Konzept der Aufteilung einer Datensammlung in Trainings- und Testdatenmengen bereits kurz erwähnt. Bedenken Sie, dass die Testdatenmenge dem endgültigen und letzten Test des Modells dient, bevor wir es in der Praxis einsetzen. In diesem Abschnitt werden wir eine neue Datensammlung aufbereiten: die *Wein-Datensammlung*. Nach der Vorverarbeitung der Daten werden wir verschiedene Verfahren zur Auswahl der Merkmale erkunden, um eine Dimensionsreduktion einer Datenmenge durchzuführen.

Die Wein-Datensammlung ist eine weitere öffentlich zugängliche Datensammlung, die vom UCI Machine Learning Repository bereitgestellt wird (<https://archive.ics.uci.edu/ml/datasets/Wine>). Sie besteht aus 178 Wein-Datensätzen mit jeweils 13 Merkmalen, die verschiedene chemische Eigenschaften beschreiben.

Mithilfe der pandas-Bibliothek lesen wir die Wein-Datensammlung direkt vom UCI Machine Learning Repository ein:

```
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/
... machine-learning-databases/wine/wine.data', header=None)
>>> df_wine.columns = ['Class label', 'Alkohol',
...                     'Apfelsäure', 'Asche',
...                     'Aschealkalität', 'Magnesium',
...                     'Phenole insgesamt', 'Flavanoide',
...                     'Nicht flavanoide Phenole',
...                     'Tannin',
...                     'Farbintensität', 'Farbe',
...                     'OD280/OD315 des verdünnten Weins',
...                     'Prolin']
>>> print('Class labels', np.unique(df_wine['Class label']))
Class labels [1 2 3]
>>> df_wine.head()
```

Die 13 verschiedenen Merkmale zur Beschreibung der chemischen Eigenschaften der 178 Datenmengen der Wein-Datensammlung finden Sie in nachstehender Tabelle.

Klas-sen-bez.	Alkohol	Apfel-säure	Asche	Asche-alkalität	Magnesium	Pheno-le insgesamt	Flavanoide	Nicht flavanoide Phenole	Tannin	Farb-intensität	Farbe	OD280/OD315 des verdünnten Weins	Prolin
0 1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	2.29	5.64	1.04	3.92	1065
1 1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.05	3.40	1050
2 1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185
3 1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480
4 1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735

Die Weine gehören zu einer der drei Klassen 1, 2 oder 3, die für verschiedene Traubensorten stehen, die in den Weinanbaugebieten Italiens kultiviert werden. Eine zusammenfassende Beschreibung der Datenmenge finden Sie unter <https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.names>.

Die `train_test_split`-Funktion des scikit-learn-Moduls `model_selection` bietet eine komfortable Möglichkeit, die Datensammlung nach dem Zufallsprinzip in eine Test- und eine Trainingsdatenmenge aufzuteilen:

```
>>> from sklearn.model_selection import train_test_split
>>> X, y = df_wine.iloc[:, 1:].values,
...                     df_wine.iloc[:, 0].values
>>> X_train, X_test, y_train, y_test = \
...     train_test_split(X, y, test_size=0.3,
...                      random_state=0, stratify=y)
```

Zunächst weisen wir dem NumPy-Array `X` die Merkmalsspalten 1 bis 13 und die Klassenbezeichnungen der ersten Spalte der Variablen `y` zu. Dann rufen wir die Funktion `train_test_split` auf, um `X` und `y` zufällig in Trainings- und Testdaten aufzuteilen. Durch die Zuweisung von `test_size=0.3` ordnen wir 30 Prozent der Weine `X_test` und `y_test` sowie die verbleibenden 70 Prozent `X_train` und `y_train` zu. Der Parameter `stratify=y` sorgt dafür, dass die Verhältnisse der Häufigkeit der Klassen in der Trainings- und der Testdatenmenge übereinstimmen.

### Tipp

Wenn wir eine Datensammlung in Trainings- und Testdaten aufteilen, ist zu bedenken, dass wir dadurch wertvolle Informationen zurückhalten, von denen der Lernalgorithmus profitieren könnte. Daher sollten nicht zu viele Informationen Testzwecken dienen. Allerdings wird die Abschätzung des Fehlers umso ungenauer, je kleiner die Testdatenmenge ist. Bei der Aufteilung in Trainings- und Testdaten geht es darum, ein geeignetes Gleichgewicht zu finden. In der Praxis ist es üblich, ein Aufteilungsverhältnis von 60:40, 70:30 oder 80:20 zu verwenden – je nach Größe der ursprünglichen Datensammlung.

Bei sehr großen Datenmengen sind aber auch Aufteilungen in Trainings- und Testdaten von 90:10 oder sogar 99:1 gebräuchlich und ebenso angemessen. Anstatt die Testdaten nach dem Training und der Bewertung des Modells zu verworfen, ist es gängige Praxis, den Klassifizierer mit der gesamten Datenmenge erneut zu trainieren, um die Vorhersagekraft des Modells zu verbessern. Dieser Ansatz ist im Allgemeinen zwar durchaus empfehlenswert, er kann jedoch zu einer Herabsetzung der Verallgemeinerungsfähigkeit des Modells führen, z.B. wenn die Datensammlung klein ist und die Testdatenmenge Ausreißer enthält. Außerdem sind nach einer erneuten Anpassung des Modells an die gesamte Datenmenge keine unabhängigen Daten mehr übrig, um die Leistung zu beurteilen.

## 4.4 Anpassung der Merkmale

Bei der Vorverarbeitung der Daten ist die Standardisierung der Merkmale ein entscheidender Schritt, der leicht übersehen wird. Entscheidungsbäume und Random Forests gehören zu den wenigen Lernalgorithmen, bei denen wir uns keine Gedanken über die Merkmalsstandardisierung machen müssen. Diese Algorithmen sind skaleninvariant. Allerdings zeigen die meisten Lern- und Optimierungsalgorithmen ein erheblich besseres Verhalten, wenn die Merkmale von gleicher Größenordnung sind, wie wir in Kapitel 2 bei der Implementierung des Gradientenabstiegsverfahrens gesehen haben.

Wie wichtig die Merkmalsstandardisierung ist, lässt sich an einem einfachen Beispiel demonstrieren. Nehmen wir an, es gibt zwei Merkmale, von denen das eine Werte zwischen 1 und 10 und das andere Werte zwischen 1 und 100.000 annimmt. Wenn man an die Funktion der quadrierten Fehler beim Adaline (ADAptive LInear NEuron)-Algorithmus aus Kapitel 2 denkt, würde man vermuten, dass der Algorithmus vornehmlich damit beschäftigt ist, die Gewichtungen aufgrund der größeren Fehler des zweiten Merkmals zu optimieren. Ein weiteres Beispiel ist der KNN-Algorithmus mit einem euklidischen Abstandsmaß – die errechneten Abstände zwischen den Objekten werden von der zweiten Merkmalsachse dominiert.

Es gibt zwei gebräuchliche Ansätze, um verschiedene Merkmale so anzupassen, dass sie von gleicher Größenordnung sind: die *Normierung* und die *Standardisierung*. Diese beiden Begriffe werden häufig ziemlich zwanglos gebraucht, deshalb muss ihre genaue Bedeutung in der Regel dem Kontext entnommen werden. Meistens bedeutet eine Normierung, dass ein Merkmal auf das Intervall [0, 1] abgebildet wird. Dabei handelt es sich um einen Spezialfall einer Min-Max-Skalierung. Zur Normierung der Daten wenden wir einfach eine Min-Max-Skalierung auf alle Merkmalsspalten an. Der neue Wert  $x_{norm}^{(i)}$  eines Objekts  $x^{(i)}$  kann wie folgt berechnet werden:

$$x_{norm}^{(i)} = \frac{x^{(i)} - x_{\min}}{x_{\max} - x_{\min}}$$

Hier bezeichnen  $x^{(i)}$  ein bestimmtes Objekt und  $x_{\min}$  sowie  $x_{\max}$  den kleinsten bzw. größten Wert der Merkmalsspalte.

Die Min-Max-Skalierung ist in scikit-learn bereits implementiert und kann wie folgt verwendet werden:

```
>>> from sklearn.preprocessing import MinMaxScaler
>>> mms = MinMaxScaler()
>>> X_train_norm = mms.fit_transform(X_train)
>>> X_test_norm = mms.transform(X_test)
```

Die Normierung per Min-Max-Skalierung ist zwar ein gebräuchliches Verfahren, das sich als nützlich erweist, wenn wir Werte innerhalb eines begrenzten Intervalls benötigen, allerdings ist eine Standardisierung für viele Lernalgorithmen praktischer, besonders für Optimierungsalgorithmen wie das Gradientenabstiegsverfahren, weil viele lineare Modelle, wie z.B. die logistische Regression oder die SVM, die Sie aus Kapitel 3 kennen, die Gewichtungen mit 0 oder kleinen, kaum von 0 verschiedenen Zufallswerten initialisieren. Bei der Standardisierung zentrieren wir die Werte der Merkmalsspalten beim Mittelwert 0 und einer Standardabweichung von 1, sodass sich eine Normalverteilung ergibt, die es erleichtert, die Gewichtungen zu ermitteln. Darüber hinaus erhält die Standardisierung nützliche Informationen über Ausreißer und sorgt dafür, dass der Algorithmus weniger empfindlich auf ungewöhnliche Werte reagiert – im Gegensatz zur Min-Max-Skalierung, die alle Daten skaliert auf ein begrenztes Intervall abbildet.

Der Standardisierungsvorgang wird durch folgende Gleichung beschrieben:

$$x_{std}^{(i)} = \frac{x^{(i)} - \mu_x}{\sigma_x}$$

Hier geben  $\mu_x$  den Mittelwert und  $\sigma_x$  die Standardabweichung einer bestimmten Merkmalsspalte an.

Die folgende Tabelle illustriert die Unterschiede zwischen den beiden gebräuchlichsten Verfahren zur Anpassung der Merkmale, der Standardisierung und der Normierung, die auf eine einfache, aus den Zahlen von 0 bis 5 bestehende Stichprobe angewendet wurden:

Eingabe	Standardisierung	Normierung
0.0	-1.46385	0.0
1.0	-0.87831	0.2
2.0	-0.29277	0.4

Eingabe	Standardisierung	Normierung
3.0	0.29277	0.6
4.0	0.87831	0.8
5.0	1.46385	1.0

Sie können die in der Tabelle aufgeführte Standardisierung und Normierung durch folgenden Code von Hand ausführen:

```
>>> ex = np.array([0, 1, 2, 3, 4, 5])
>>> print('Standardisierung:', (ex - ex.mean()) / ex.std())
Standardisierung: [-1.46385011 -0.87831007 -0.29277002
                    0.29277002  0.87831007  1.46385011]
>>> print('Normierung:', (ex - ex.min()) /
                  (ex.max() - ex.min()))
Normierung: [ 0.  0.2  0.4  0.6  0.8  1. ]
```

Neben dem `MinMaxScaler` implementiert scikit-learn auch eine Klasse zur Standardisierung:

```
>>> from sklearn.preprocessing import StandardScaler
>>> stdsc = StandardScaler()
>>> X_train_std = stdsc.fit_transform(X_train)
>>> X_test_std = stdsc.transform(X_test)
```

Auch hier ist es wichtig zu betonen, dass wir den `StandardScaler` nur einmal an die Trainingsdaten anpassen und die so gewonnenen Parameter verwenden, um die Testdatenmenge oder neue Daten zu transformieren.

## 4.5 Auswahl aussagekräftiger Merkmale

Wenn wir feststellen, dass ein Modell mit den Trainingsdaten viel besser funktioniert als mit den Testdaten, ist das ein deutlicher Hinweis auf eine *Überanpassung*, das heißt: Das Modell passt die Parameter zu genau an die Werte in den Trainingsdaten an, ohne dass es gut auf echte Daten übertragbar wäre. Wir sprechen in diesem Fall von einer *hohen Varianz* des Modells. Einer der Gründe für eine Überanpassung ist, dass sich unser Modell für die gegebenen Trainingsdaten als zu komplex erweist. Um diesem Fehler bei der Übertragung des Modells auf die Grundgesamtheit entgegenzuwirken, sind folgende Lösungen gebräuchlich:

- Mehr Trainingsdaten sammeln
- Einführung einer Straffunktion zwecks Ahndung von Komplexität per Regularisierung

- Auswahl eines einfacheren Modells mit weniger Parametern
- Dimensionsreduktion der Daten

Mehr Trainingsdaten zu sammeln, ist in der Praxis allerdings oft nicht machbar. In Kapitel 6 werden Sie ein nützliches Verfahren kennenlernen, mit dem festgestellt werden kann, ob zusätzliche Trainingsdaten die Situation überhaupt verbessern würden. In den folgenden Abschnitten werden wir gängige Methoden zur Verringerung der Überanpassung per Regularisierung und Dimensionsreduktion durch Merkmalsauswahl betrachten, sodass weniger Parameter zur Anpassung an die Daten benötigt werden.

### 4.5.1 L1- und L2-Regularisierung als Straffunktionen

Aus Kapitel 3 wissen Sie, dass die *L2-Regularisierung* ein Ansatz ist, um die Komplexität eines Modells zu verringern, indem große individuelle Gewichtungen »bestraft« werden. Die L2-Norm des Gewichtungsvektors  $w$  ist folgendermaßen definiert:

$$L2: \|w\|_2^2 = \sum_{j=1}^m w_j^2$$

Ein weiterer Ansatz zur Reduzierung der Komplexität des Modells ist die verwandte *L1-Regularisierung*:

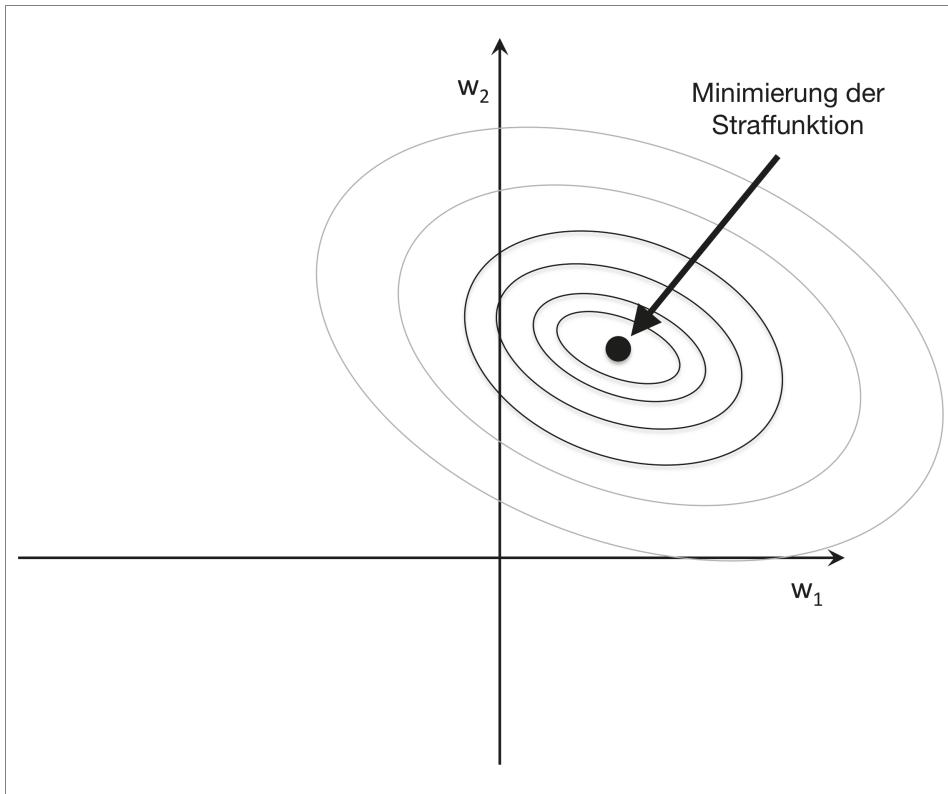
$$L1: \|w\|_1 = \sum_{j=1}^m |w_j|$$

Wir ersetzen hier einfach die Quadrate der Gewichtungen durch die Summe der Absolutwerte der Gewichtungen. Im Gegensatz zur L2-Regularisierung liefert die L1-Regularisierung dünnbesetzte Merkmalsvektoren – die meisten Merkmalsgewichtungen sind null. Das erweist sich in der Praxis als nützlich, wenn wir es mit hochdimensionalen Datenmengen zu tun haben, bei denen viele Merkmale nicht von Bedeutung sind, insbesondere in Fällen, in denen es mehr irrelevante Dimensionen als Objekte gibt. So gesehen kann man die L1-Regularisierung als ein Verfahren zur Auswahl von Merkmalen auffassen.

### 4.5.2 Geometrische Interpretation der L2-Regularisierung

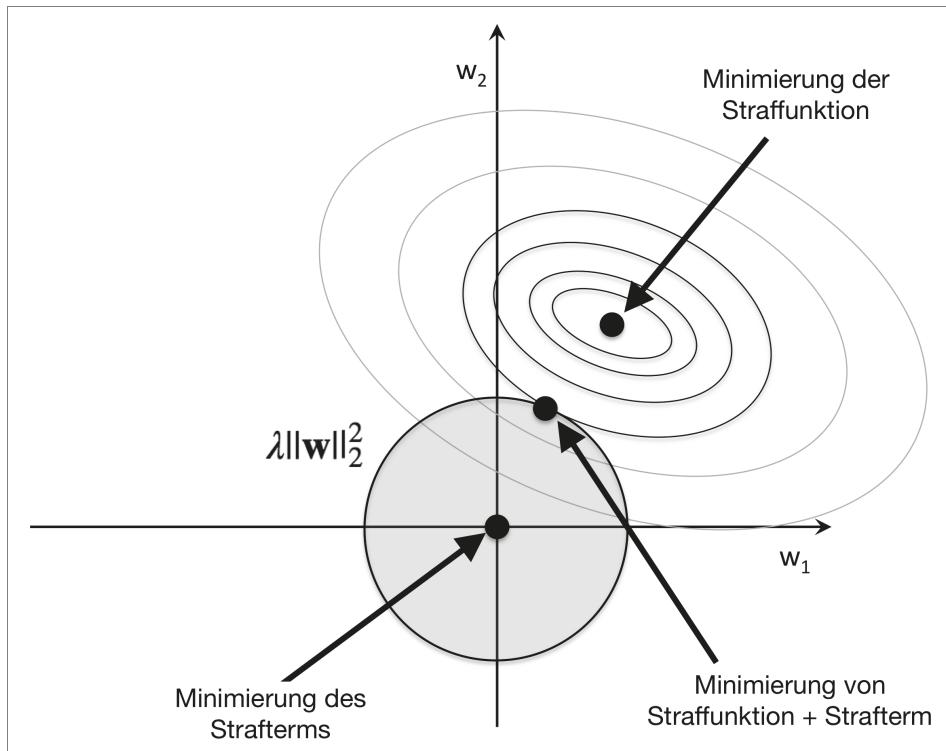
Wie Sie aus den letzten Abschnitten wissen, wird bei der L2-Regularisierung ein zusätzlicher Term zur Straffunktion hinzugefügt, der letztlich, im Vergleich zu einem Modell mit unregularisierter Straffunktion, zu weniger extremen Gewichtungen führt. Um besser zu verstehen, wie die L1-Regularisierung dünnbesetzte Vektoren begünstigt, betrachten wir eine geometrische Interpretation der Regula-

risierung. Wir plotten die Höhenlinien einer konvexen Straffunktion für zwei Gewichtungskoeffizienten  $w_1$  und  $w_2$ . Wir verwenden hier die in Kapitel 2 beim Adaline-Algorithmus eingeführte *Summe der quadrierten Abweichungen* als Straffunktion, da sie symmetrisch ist und sich besser darstellen lässt als die Straffunktion der logistischen Regression – für Letztere gelten jedoch dieselben Konzepte. Unser Ziel ist es, diejenige Kombination der Gewichtungskoeffizienten zu finden, bei der die Straffunktion für die Trainingsdaten wie in der Abbildung minimal wird (der Punkt inmitten der Ellipsen).



Man kann sich die Regularisierung als das Hinzufügen eines Strafterms zur Straffunktion vorstellen, der kleinere Gewichtungen begünstigt oder, anders ausgedrückt, große Gewichtungen bestraft.

Indem wir über den Regularisierungsparameter  $\lambda$  die Regularisierung verstärken, schrumpfen wir die Gewichtungen in Richtung null und verringern so die Abhängigkeit unseres Modells von den Trainingsdaten. Die folgende Abbildung illustriert dieses Konzept für den L2-Strafterm.

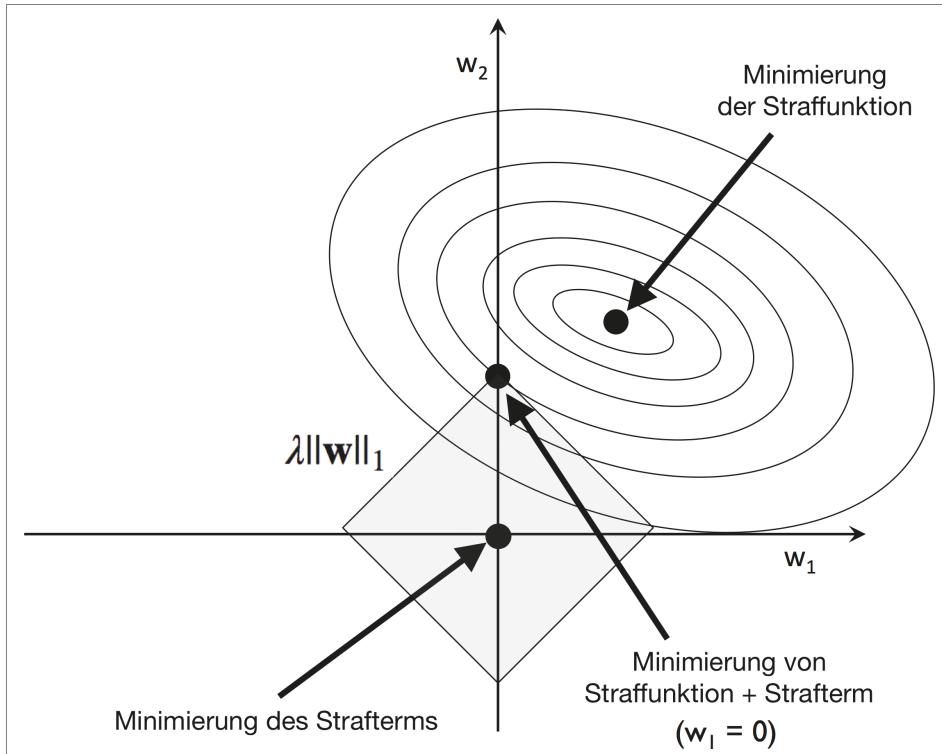


Der quadratische L<sub>2</sub>-Regularisierungsterm wird durch den grau gefärbten Kreis repräsentiert. Die Gewichtungskoeffizienten dürfen die vorgesehene Regularisierung nicht überschreiten – die Kombination der Koeffizienten kann nicht außerhalb des grauen Bereichs gelangen. Andererseits möchten wir ja auch die Straffunktion minimieren. Unter Berücksichtigung des Strafterms ist der Punkt, an dem sich die L<sub>2</sub>-Kugel und die Höhenlinien der Straffunktion ohne Strafterm schneiden, die beste Wahl. Je größer der Wert des Regularisierungsparameters  $\lambda$  wird, desto schneller wächst der Wert der Straffunktion mit Strafterm, was zu einer kleineren L<sub>2</sub>-Kugel führt. Wenn wir beispielsweise den Regularisierungspараметer gegen unendlich gehen lassen, gehen die Gewichtungskoeffizienten gegen null – das entspricht dem Mittelpunkt der L<sub>2</sub>-Kugel. Zusammengefasst heißt das: Unser Ziel ist es, die Summe aus Straffunktion und Strafterm zu minimieren, was wiederum als Hinzufügen eines Bias und Bevorzugung eines einfacheren Modells aufgefasst werden kann, um die Varianz zu verringern, sofern nicht genügend Trainingsdaten zur Anpassung des Modells vorliegen.

#### 4.5.3 Dünnbesetzte Lösungen mit L<sub>1</sub>-Regularisierung

Sehen wir uns nun die L<sub>1</sub>-Regularisierung an und wie sie sich auf dünnbesetzte Lösungen auswirkt. Das der L<sub>1</sub>-Regularisierung zugrunde liegende Hauptkonzept

ähnelt dem soeben vorgestellten. Da die L1-Strafffunktion jedoch als die Summe der Absolutwerte der Gewichtungskoeffizienten definiert ist (beachten Sie, dass der L2-Term quadratisch ist), können wir ihn wie in der folgenden Abbildung als rautenförmigen Bereich repräsentieren.



In der Abbildung ist erkennbar, dass die Höhenlinie der Straffunktion die L1-Raute bei  $w_1 = 0$  berührt. Die Höhenlinien eines Systems mit L1-Regularisierung sind klar definiert, somit ist es wahrscheinlicher, dass sich das Optimum (also der Schnittpunkt zwischen den Ellipsen der Straffunktion und dem Rand der L1-Raute) auf den Koordinatenachsen befindet, was dünnbesetzte Vektoren begünstigt.

### Hinweis

Eine Erläuterung der mathematischen Gründe dafür, dass die L1-Regularisierung zu dünnbesetzten Vektoren führt, geht über den Rahmen dieses Buches hinaus. Sollten Sie daran interessiert sein, finden Sie in dem Buch *The Elements of Statistical Learning* von Trevor Hastie, Robert Tibshirani und Jerome Friedman (Springer Science + Business Media, 2009) im Abschnitt 3.4 eine ausgezeichnete Gegenüberstellung von L1- und L2-Regularisierung.

Bei scikit-learn-Modellen, die eine L1-Regularisierung unterstützen, können wir einfach den Parameter `penalty` auf 'l1' setzen, um eine dünnbesetzte Lösung zu erhalten:

```
>>> from sklearn.linear_model import LogisticRegression
>>> LogisticRegression(penalty='l1')
```

Wendet man die logistische Regression mit der L1-Regularisierung auf die standardisierten Wein-Daten an, ergibt sich folgende dünnbesetzte Lösung:

```
>>> lr = LogisticRegression(penalty='l1', C=1.0)
>>> lr.fit(X_train_std, y_train)
>>> print('Korrektklassifizierungsrate Training:',
...       lr.score(X_train_std, y_train))
Korrektklassifizierungsrate Training: 1.0
>>> print('Korrektklassifizierungsrate Test:',
...       lr.score(X_test_std, y_test))
Korrektklassifizierungsrate Test: 1.0
```

Die Korrektklassifizierungsraten beim Training und Testen (jeweils 100 Prozent) zeigen, dass unser Modell beide Datenmengen vollständig richtig klassifiziert. Wenn wir anhand des Attributs `lr.intercept_` auf die Achsenabschnitte zugreifen, wird deutlich, dass das Array drei Werte enthält:

```
>>> lr.intercept_
array([-1.26338637, -1.21582071, -2.3701035])
```

Da wir das `LogisticRegression`-Objekt an eine Datenmenge mit mehreren Klassen anpassen, wird standardmäßig der *OvR-Ansatz* (One-vs.-Rest) verwendet. Dabei gehört der erste Achsenabschnitt zu dem Modell, das die Klasse 1 als positive Klasse behandelt (mit den Klassen 2 und 3 als negativen Klassen), der zweite Wert ist der Achsenabschnitt des Modells, das Klasse 2 als positive Klasse behandelt (Klasse 1 und 3 als negative Klassen), und der dritte Wert gehört schließlich zu dem Modell, in dem die Klasse 3 die positive Klasse ist (mit den Klassen 1 und 2 als negativen Klassen):

```
>>> lr.coef_
array([[ 1.24559337,  0.18041967,  0.74328894, -1.16046277,
        0. ,  0. ,  1.1678711,  0. ,  0. ,  0. ,  0.54941931,
        2.51017406], [-1.53720749, -0.38727002,
       -0.99539203,  0.3651479, -0.0596352 ,  0. ,
       0.66833149,  0. ,  0. , -1.9346134,  1.23297955,  0. ,
      -2.23135027], [ 0.13579227,  0.16837686,  0.35723831,
```

```
0., 0., 0., -2.43809275, 0., 0., 1.56391408,
-0.81933286, -0.49187817, 0.]])
```

Das mit dem Attribut `lr.coef_` angezeigte Array enthält drei Zeilen mit Gewichtungskoeffizienten, jeweils einen Gewichtungsvektor pro Klasse. Jede Zeile besteht aus 13 Gewichtungen, die mit den entsprechenden Merkmalen der 13-dimensionalen Wein-Datensammlung multipliziert werden, um die Nettoeingabe zu berechnen:

$$z = w_0 x_0 + \dots + w_m x_m = \sum_{j=0}^m x_j w_j = \mathbf{w}^T \mathbf{x}$$

### Tipp

In scikit-learn entsprechen  $w_0$  und  $w_j$  (mit  $j > 0$ ) `intercept_` und den Werten in `coef_`.

Als Ergebnis der L1-Regularisierung, die als Methode zur Merkmalsauswahl dient, haben wir nun ein Modell trainiert, das gegenüber den potenziell irrelevanten Merkmalen in dieser Datensammlung robust ist.

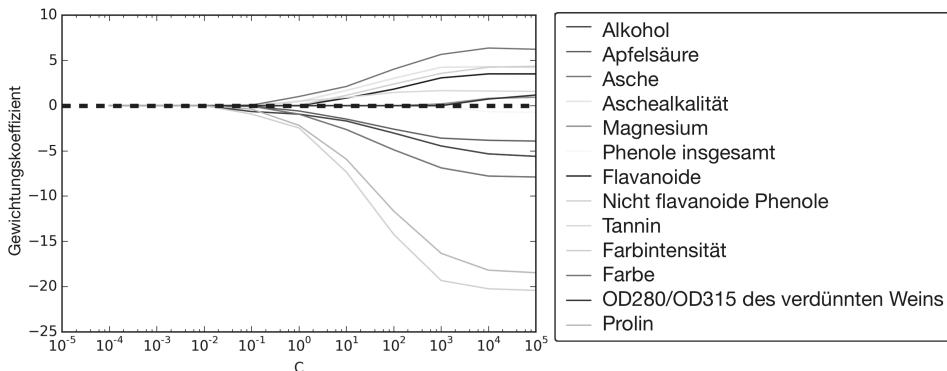
Genau genommen sind die Gewichtungsvektoren dieses Beispiels gar nicht dünnbesetzt, weil sie mehr von null verschiedene Einträge als Nulleinträge enthalten. Wir könnten dies aber erzwingen, indem wir die Regularisierungsstärke weiter erhöhen, also einen geringeren Wert für den Parameter `C` wählen.

Abschließend geben wir den Regularisierungspfad aus, bei dem es sich um die Gewichtungskoeffizienten der verschiedenen Merkmale bei unterschiedlicher Regularisierungsstärke handelt:

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> ax = plt.subplot(111)
>>> colors = ['blue', 'green', 'red', 'cyan',
...             'magenta', 'yellow', 'black',
...             'pink', 'lightgreen', 'lightblue',
...             'gray', 'indigo', 'orange']
>>> weights, params = [], []
>>> for c in np.arange(-4., 6.):
...     lr = LogisticRegression(penalty='l1',
...                             C=10.**c,
...                             random_state=0)
...     lr.fit(X_train_std, y_train)
...     weights.append(lr.coef_[1])
...     params.append(10**c)
```

```
>>> weights = np.array(weights)
>>> for column, color in zip(range(weights.shape[1]),colors):
...     plt.plot(params, weights[:, column],
...               label=df._wine.columns[column+1],
...               color=color)
>>> plt.axhline(0, color='black', linestyle='--', linewidth=3)
>>> plt.xlim([10**(-5), 10**5])
>>> plt.ylabel('Gewichtungskoeffizient')
>>> plt.xlabel('C')
>>> plt.xscale('log')
>>> plt.legend(loc='upper left')
>>> ax.legend(loc='upper center',
...            bbox_to_anchor=(1.38, 1.03),
...            ncol=1, fancybox=True)
>>> plt.show()
```

Die resultierende Ausgabe gewährt uns weitere Einsichten in das Verhalten der L1-Regularisierung. Wie man sieht, sind die Gewichtungen aller Merkmale null, wenn man eine hohe Regularisierungsstärke ( $C < 0.1$ ) verwendet.  $C$  ist der Kehrwert des Regularisierungsparameters  $\lambda$ .



#### 4.5.4 Algorithmen zur sequenziellen Auswahl von Merkmalen

Eine weitere Möglichkeit, die Komplexität des Modells zu verringern und eine Überanpassung zu vermeiden, ist die *Dimensionsreduktion* durch die Auswahl von Merkmalen, die sich insbesondere bei nicht regularisierten Modellen als nützlich erweist. Bei den Verfahren zur Dimensionsreduktion gibt es zwei Hauptkategorien: die Auswahl und die Extrahierung von Merkmalen. Bei der Auswahl von Merkmalen verwenden wir eine Untermenge der ursprünglichen Merkmale. Bei der Extrahierung von Merkmalen hingegen leiten wir aus der Menge der Merkmale Informationen her, um einen neuen Merkmalsraum zu konstruieren.

In diesem Abschnitt werden wir eine klassische Familie von Algorithmen zur Auswahl von Merkmalen betrachten. Im nächsten Kapitel werden wir uns dann mit verschiedenen Verfahren zur Extrahierung von Merkmalen befassen, um die Daten zu komprimieren, indem sie auf einen Merkmalsraum mit weniger Dimensionen abgebildet werden.

Bei den Algorithmen zur sequenziellen Auswahl von Merkmalen handelt es sich um eine Familie »habgieriger« sogenannter *Greedy-Suchalgorithmen* (siehe Kasten), die dazu verwendet werden, den anfänglich  $d$ -dimensionalen Merkmalsraum auf einen  $k$ -dimensionalen Unterraum zu reduzieren, wobei  $k < d$  gilt. Sie sollen es ermöglichen, automatisch eine Teilmenge der Merkmale auszuwählen, die für eine gegebene Aufgabenstellung die größte Bedeutung haben, um die Berechnungen effizienter ausführen zu können oder den Verallgemeinerungsfehler des Modells zu verringern, indem irrelevante Merkmale und Rauschen entfernt werden, was sich insbesondere für Algorithmen als nützlich erweist, die keine Regularisierung unterstützen.

Die *sequenzielle Rückwärtsauswahl* (*SBS*, *Sequential Backwards Selection*) ist ein klassischer Algorithmus zur Auswahl von Merkmalen, der versucht, eine Dimensionsreduktion des anfänglichen Merkmalsraums mit nur minimalen Leistungseinbußen des Klassifizierers vorzunehmen, um die Effizienz der Berechnungen zu erhöhen. Wenn ein Modell unter einer Überanpassung leidet, kann SBS unter bestimmten Umständen sogar die Vorhersagekraft des Modells verbessern.

### Tipp

Greedy-Algorithmen treffen bei jeder Stufe einer kombinatorischen Suchaufgabe eine lokal optimale Auswahl und liefern im Allgemeinen nicht die optimale Gesamtlösung. Im Gegensatz dazu bewerten vollständige Suchalgorithmen sämtliche möglichen Kombinationen und finden garantiert die optimale Gesamtlösung. In der Praxis ist eine vollständige Suche allerdings aufgrund des damit verbundenen Rechenaufwands oft nicht machbar, während Greedy-Algorithmen eine weniger umfassende, aber vom Rechenaufwand her effizientere Lösung ermöglichen.

Die dem SBS-Algorithmus zugrunde liegende Idee ist ganz einfach: SBS entfernt der Reihe nach Merkmale aus der vollständigen Merkmalsmenge, bis der verbleibende Merkmalsraum nur noch die erwünschte Anzahl von Merkmalen enthält. Um festzustellen, welches Merkmal bei jeder der Stufen entfernt werden soll, müssen wir eine Entscheidungsfunktion  $J$  definieren, die es zu minimieren gilt. Das von dieser Entscheidungsfunktion berechnete Kriterium könnte etwa die Differenz der Leistung des Klassifizierers vor und nach dem Entfernen eines bestimmten Merkmals sein. Das zu entfernende Merkmal einer jeden Stufe wäre

dann dasjenige, das den günstigsten Wert für das Entscheidungskriterium liefert oder etwas verständlicher formuliert: Wir entfernen jeweils das Merkmal, dessen Abwesenheit zu der geringsten Leistungseinbuße führt. Gemäß dieser SBS-Definition können wir den Algorithmus mit folgenden vier Schritten beschreiben:

1. Initialisieren des Algorithmus mit  $k = d$ . Dabei repräsentiert  $d$  die Dimensionalität des vollständigen Merkmalsraums  $\mathbf{X}_d$ .
2. Festlegen des Merkmals  $x^-$  zur Maximierung des Kriteriums:  

$$x^- = \operatorname{argmax} J(X_k - x), \text{ wobei } x \in X_k.$$
3. Entfernen des Merkmals  $x^-$  aus der Merkmalsmenge:  

$$X_{k-1} := X_k - x^-, k := k - 1.$$
4. Beenden, wenn  $k$  gleich der Anzahl der erwünschten Merkmale ist, ansonsten mit Schritt 2 fortfahren.

### Tipp

Eine ausführliche Untersuchung der verschiedenen sequenziellen Algorithmen zur Merkmalsauswahl finden Sie in *Comparative Study of Techniques for Large Scale Feature Selection*, F. Ferri, P. Pudil, M. Hatef und J. Kittler, *Pattern Recognition in Practice IV*, Seiten 403-413, 1994.

Leider ist der SBS-Algorithmus in scikit-learn noch nicht implementiert, da er jedoch so einfach ist, implementieren wir ihn einfach selbst in Python:

```
from sklearn.base import clone
from itertools import combinations
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

class SBS():
    def __init__(self, estimator, k_features,
                 scoring=accuracy_score,
                 test_size=0.25, random_state=1):
        self.scoring = scoring
        self.estimator = clone(estimator)
        self.k_features = k_features
        self.test_size = test_size
        self.random_state = random_state

    def fit(self, X, y):
        X_train, X_test, y_train, y_test = \
            train_test_split(X, y, test_size=self.test_size,
                             random_state=self.random_state)
```

```

        dim = X_train.shape[1]
        self.indices_ = tuple(range(dim))
        self.subsets_ = [self.indices_]
        score = self._calc_score(X_train, y_train,
                                X_test, y_test, self.indices_)
        self.scores_ = [score]

        while dim > self.k_features:
            scores = []
            subsets = []

            for p in combinations(self.indices_, r=dim-1):
                score = self._calc_score(X_train, y_train,
                                         X_test, y_test, p)
                scores.append(score)
                subsets.append(p)

            best = np.argmax(scores)
            self.indices_ = subsets[best]
            self.subsets_.append(self.indices_)
            dim -= 1

            self.scores_.append(scores[best])
            self.k_score_ = self.scores_[-1]

        return self
    def transform(self, X):
        return X[:, self.indices_]

    def _calc_score(self, X_train, y_train,
                   X_test, y_test, indices):
        self.estimator.fit(X_train[:, indices], y_train)
        y_pred = self.estimator.predict(X_test[:, indices])
        score = self.scoring(y_test, y_pred)
        return score

```

In dieser Implementierung legt der Parameter `k_features` die erwünschte Anzahl der Merkmale fest, die zurückgegeben werden sollen. Wir verwenden bei den Merkmalsuntermengen standardmäßig scikit-learns `accuracy_score`, um die Leistungsfähigkeit eines Modells und eines Schätzers zur Klassifizierung zu beurteilen. Innerhalb der `while`-Schleife der `fit`-Methode werden die von der `itertools.combinations`-Funktion erstellten Merkmalsuntermengen bewertet und verkleinert, bis die erwünschte Dimensionalität erreicht ist. Bei jeder Iteration wird der anhand der intern erzeugten Testdatenmenge `X_test` ermittelte Korrektklassifizierungsraten-Score der besten Untermenge in der Liste `self.scores_`

gespeichert. Diesen Score werden wir später zur Beurteilung der Ergebnisse verwenden. Die Spaltenindizes der endgültigen Merkmalsuntermenge werden `self.indices_` zugewiesen. Diese können wir nutzen, um über die `transform`-Methode ein neues Array mit den ausgewählten Merkmalsspalten zurückzugeben. Beachten Sie, dass in der `fit`-Methode keine Berechnung des Kriteriums erfolgt – stattdessen entfernen wir einfach dasjenige Merkmal, das nicht in der Merkmalsuntermenge mit den besten Scores enthalten ist.

Nun wollen wir unsere SBS-Implementierung in Aktion sehen und den KNN-Klassifizierer von scikit-learn verwenden:

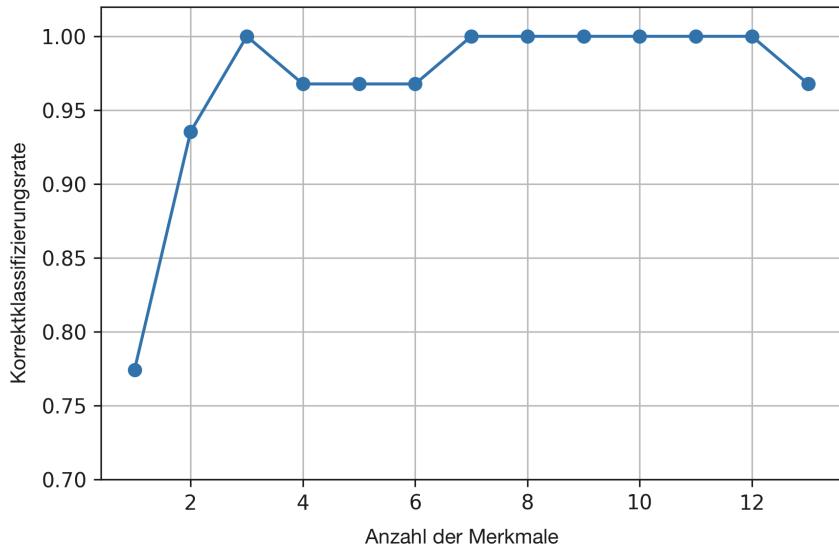
```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> import matplotlib.pyplot as plt
>>> knn = KNeighborsClassifier(n_neighbors=5)
>>> sbs = SBS(knn, k_features=1)
>>> sbs.fit(X_train_std, y_train)
```

Zwar teilt unsere SBS-Implementierung die Datensammlung innerhalb der `fit`-Methode bereits in eine Test- und eine Trainingsdatenmenge auf, wir übergeben dem Algorithmus aber dennoch die Trainingsdatenmenge `X_train`. Die `fit`-Methode erzeugt dann neue Trainingsuntermengen zum Testen (Validierung) und Trainieren, daher wird diese Testdatenmenge auch als *Validierungsmenge* bezeichnet. Diese Vorgehensweise ist erforderlich, um zu verhindern, dass unsere *ursprünglichen* Testdaten Teil der Trainingsdaten werden.

Der SBS-Algorithmus speichert bei jeder Stufe die Scores der besten Merkmalsuntermengen – wenden wir uns also den interessanteren Teilen unserer Implementierung zu. Wir plotten die Korrektklassifizierungsrate des KNN-Klassifizierers, die sich für die Validierungsdatenmenge ergibt. Hier der Code:

```
>>> k_feat = [len(k) for k in sbs.subsets_]
>>> plt.plot(k_feat, sbs.scores_, marker='o')
>>> plt.ylim([0.7, 1.02])
>>> plt.ylabel('Korrektklassifizierungsrate')
>>> plt.xlabel('Anzahl der Merkmale')
>>> plt.grid()
>>> plt.show()
```

In der nachfolgenden Abbildung ist erkennbar, dass sich die Korrektklassifizierungsrate des KNN-Klassifizierers bei der Validierungsdatenmenge verbessert, wenn die Anzahl der Merkmale verringert wird. Das liegt vermutlich daran, dass der in Zusammenhang mit dem KNN-Algorithmus in Kapitel 3 erörterte *Fluch der Dimensionalität* weniger stark zum Tragen kommt. Für  $k=[3, 7, 8, 9, 10, 11, 12]$  erreicht der Klassifizierer sogar eine Korrektklassifizierungsrate von 100 Prozent:



Um unsere Neugier zu befriedigen, sehen wir nun nach, welche drei Merkmale ( $k=3$ ) eine so gute Leistung bei der Validierungsmenge ermöglichen:

```
>>> k3 = list(sbs.subsets_[10])
>>> print(df_wine.columns[1:][k3])
Index(['Alkohol', 'Apfelsäure', 'OD280/OD315 des verdünnten Weins',
       dtype='object')
```

Der Code fragt die Indizes einer Untermenge mit drei Elementen ab der zehnten Position des Attributs `sbs.subsets` ab und gibt über den Spaltenindex des pandas-DataFrame die dazugehörigen Merkmalsbezeichnungen aus.

Sehen wir uns als Nächstes die Leistung des KNN-Klassifizierers bei der ursprünglichen Testdatenmenge an:

```
>>> knn.fit(X_train_std, y_train)
>>> print('Korrektklassifizierungsrate Training:', 
...       knn.score(X_train_std, y_train))
Korrektklassifizierungsrate Training: 0.967741935484
>>> print('Korrektklassifizierungsrate Test:', 
...       knn.score(X_test_std, y_test))
Korrektklassifizierungsrate Test: 0.962962962963
```

Hier haben wir alle Merkmale verwendet und erhalten bei den Trainingsdaten eine Korrektklassifizierungsrate von etwa 97 Prozent – bei den Testdaten ist sie etwas geringer ( $\approx 93$  Prozent), was darauf hindeutet, dass unser Modell schon ganz gut mit neuen Daten zurechtkommt. Nun verwenden wir die Untermenge der drei

ausgewählten Merkmale und überprüfen damit die Leistung des KNN-Algorithmus:

```
>>> knn.fit(X_train_std[:, k3], y_train)
>>> print('Korrektklassifizierungsrate Training:', 
...       knn.score(X_train_std[:, k3], y_train))
Korrektklassifizierungsrate Training: 0.951612903226
>>> print('Korrektklassifizierungsrate Test:', 
...       knn.score(X_test_std[:, k5], y_test))
Korrektklassifizierungsrate Test: 0.925925925926
```

Die Nutzung von weniger als einem Viertel der in der Wein-Datensammlung ursprünglich vorhandenen Merkmale führt bei der Testdatenmenge zu einer leichten Verschlechterung der Vorhersagegenauigkeit. Das könnte ein Hinweis darauf sein, dass diese drei Merkmale weniger zur Unterscheidung beitragende Informationen liefern als die ursprüngliche Datenmenge. Wir müssen jedoch auch berücksichtigen, dass die Wein-Datensammlung eine sehr kleine Datenmenge ist, die sehr anfällig für zufällige Schwankungen ist, also die Art und Weise, wie die Datenmenge in Trainings- und Testuntermengen und die Trainingsdatenmenge weiter in Trainings- und Validierungsmengen aufgeteilt werden.

Wir haben die Leistung des KNN-Modells zwar nicht gesteigert, aber die Größe der Datenmenge reduziert, was in der Praxis durchaus nützlich sein kann, wenn die Datenbeschaffung mit hohen Kosten verbunden ist. Außerdem erhalten wir durch die Verringerung der Anzahl der Merkmale einfachere Modelle, die besser interpretierbar sind.

## Tipp

### Algorithmen zur Auswahl von Merkmalen in scikit-learn

In scikit-learn steht eine ganze Reihe weiterer Algorithmen für die Auswahl von Merkmalen zur Verfügung. Dazu gehören unter anderem die rekursive Rückwärtseliminierung anhand der Merkmalsgewichtungen, baumbasierte Methoden zur Auswahl von Merkmalen aufgrund ihrer Bedeutung und univariate (eindimensionale) statistische Tests. Eine ausführliche Erläuterung der verschiedenen Methoden zur Auswahl von Merkmalen geht über den Rahmen dieses Buches hinaus, eine gute Zusammenfassung mit anschaulichen Beispielen finden Sie jedoch unter [http://scikit-learn.org/stable/modules/feature\\_selection.html](http://scikit-learn.org/stable/modules/feature_selection.html). Darüber hinaus habe ich verschiedene Algorithmen zur sequentiellen Merkmalsauswahl implementiert, die mit dem einfachen vorhin implementierten SBS-Algorithmus verwandt sind. Sie sind Bestandteil des Python-Pakets `mlxtend` und unter [http://rasbt.github.io/mlxtend/user\\_guide/feature\\_selection/SequentialFeatureSelector/](http://rasbt.github.io/mlxtend/user_guide/feature_selection/SequentialFeatureSelector/) zu finden.

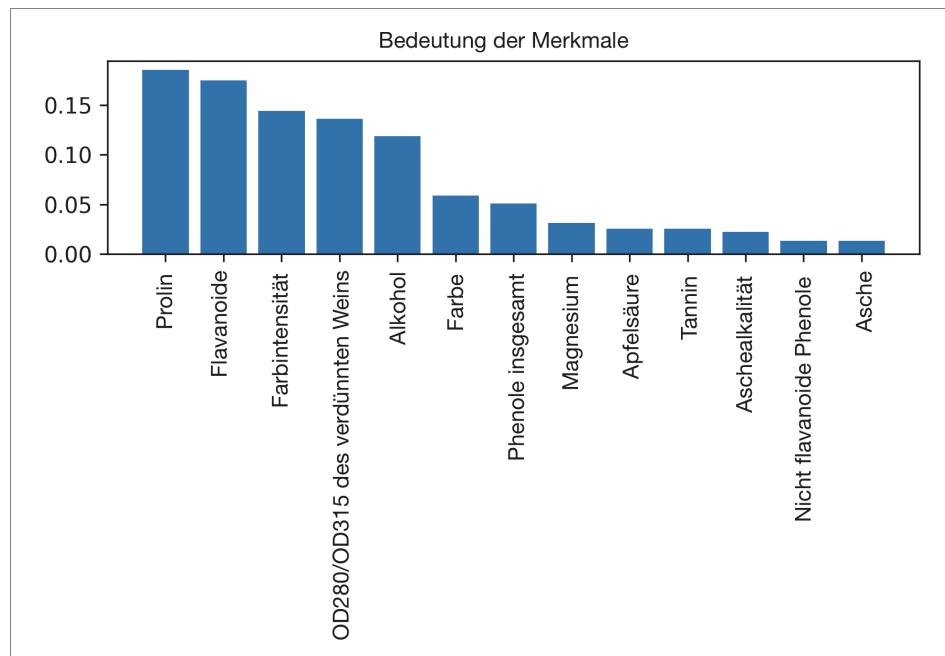
## 4.6 Beurteilung der Bedeutung von Merkmalen mit Random Forests

In den vorangegangenen Abschnitten haben Sie erfahren, wie man die L1-Regularisierung einsetzt, um irrelevante Merkmale mittels logistischer Regression zu entfernen, und Sie haben den SBS-Algorithmus zur Auswahl von Merkmalen kennengelernt und in einem KNN-Algorithmus angewendet. Ein weiterer nützlicher Ansatz zur Auswahl relevanter Merkmale einer Datensammlung ist der Einsatz eines Random Forests – einer Ensemblemethode, die in Kapitel 3 vorgestellt wurde. Mit einem Random Forest können wir die Bedeutung von Merkmalen als die durchschnittliche Absenkung der Unreinheit bemessen, die sich aus den Entscheidungsbäumen im Random Forest ergibt, ohne irgendwelche Annahmen darüber anzustellen, ob die vorliegenden Daten linear trennbar sind oder nicht. Die Implementierung des Random Forests in scikit-learn ermittelt bereits die Bedeutung von Merkmalen, sodass wir nach der Anpassung eines `RandomForestClassifiers` über das Attribut `feature_importance` darauf zugreifen können. Durch das Ausführen des nachstehenden Codes werden wir nun einen aus 500 Bäumen bestehenden Random Forest mit der Wein-Datensammlung trainieren und eine nach ihrer Bedeutung sortierte Rangliste der 13 Merkmale erstellen. Wie Sie aus Kapitel 3 wissen, ist es bei baumbasierten Modellen nicht erforderlich, standardisierte oder normierte Merkmale zu verwenden. Hier der Code:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> feat_labels = df_wine.columns[1:]
>>> forest = RandomForestClassifier(n_estimators=500,
...                                 random_state=1)
>>> forest.fit(X_train, y_train)
>>> importances = forest.feature_importances_
>>> indices = np.argsort(importances)[::-1]
>>> for f in range(X_train.shape[1]):
...     print("%2d) %-*s %f" % (f + 1, 30,
...                            feat_labels[indices[f]],
...                            importances[indices[f]]))
>>> plt.title('Bedeutung der Merkmale')
>>> plt.bar(range(X_train.shape[1]),
...          importances[indices],
...          color='lightblue',
...          align='center')
>>> plt.xticks(range(X_train.shape[1]),
...             feat_labels[indices], rotation=90)
>>> plt.xlim([-1, X_train.shape[1]])
>>> plt.tight_layout()
>>> plt.show()
```

1) Prolin	0.185453
2) Flavanoide	0.174751
3) Farbintensität	0.143920
4) OD280/OD315 des verdünnten Weins	0.136162
5) Alkohol	0.118529
6) Farbe	0.058739
7) Phenole insgesamt	0.050872
8) Magnesium	0.031357
9) Apfelsäure	0.025648
10) Tannin	0.025570
11) Aschealkalität	0.022366
12) Nicht flavanoide Phenole	0.013354
13) Asche	0.013279

Nach der Ausführung des Codes können wir ein Diagramm erstellen, in dem die verschiedenen Merkmale nach ihrer relativen Bedeutung sortiert sind. Beachten Sie, dass die Werte normiert sind und in der Summe 1.0 ergeben.



Wir können daraus schließen, dass Prolin- und Flavanoidgehalt, die Farbintensität des Weins, die OD280/OD315-Beugung und die Alkoholkonzentration die herausstechendsten Merkmale der Datensammlung sind, wenn man die durchschnittliche Absenkung der Unreinheit in den 500 Entscheidungsbäumen zugrunde legt. Interessanterweise gehören die ersten beiden Merkmale auch zu den drei Merkmalen, die der im vorangegangenen Abschnitt implementierte SBS-Algorithmus

ausgewählt hat (Alkoholkonzentration und OD280/OD315 des verdünnten Weins). Was die Interpretierbarkeit betrifft, hat das Random-Forest-Verfahren allerdings einen Haken, der hier erwähnt werden muss: Wenn beispielsweise zwei oder mehr Merkmale in hohem Maße korreliert sind, könnte eins davon in der Rangliste weit oben stehen, während die Informationen der anderen Merkmale womöglich gar nicht vollständig erfasst werden. Andererseits brauchen wir uns darüber keine Gedanken zu machen, wenn wir nicht an der Interpretierbarkeit der Bedeutung von Merkmalen, sondern lediglich an der Vorhersagekraft eines Modells interessiert sind.

Zum Abschluss dieses Abschnitts über die Bedeutung von Merkmalen und Random Forests soll noch erwähnt werden, dass scikit-learn auch ein `SelectFromModel`-Objekt implementiert, das nach der Anpassung eines Modells Merkmale anhand eines benutzerdefinierten Grenzwertes auswählt. Das erweist sich als nützlich, wenn man als Zwischenschritt einer scikit-learn-Pipeline zur Auswahl von Merkmalen einen `RandomForestClassifier` benutzen möchte, der es ermöglicht, verschiedene Vorverarbeitungsschritte mit einem Schätzer zu verknüpfen, wie Sie in Kapitel 6 noch sehen werden. Wir könnten beispielsweise mit dem nachstehenden Code einen Grenzwert von 0.1 festlegen, um die Datensammlung auf die fünf bedeutsamsten Merkmale zu beschränken:

```
>>> from sklearn.feature_selection import SelectFromModel
>>> sfm = SelectFromModel(forest, threshold=0.1, prefit=True)
>>> X_selected = sfm.transform(X_train)
>>> print('Anzahl der Objekte, für die das Kriterium gilt:', 
...       X_selected.shape[0])
Anzahl der Objekte, für die das Kriterium gilt: 124
>>> for f in range(X_selected.shape[1]):
...     print("%2d %-*s %f" % (f + 1, 30,
...                           feat_labels[indices[f]],
...                           importances[indices[f]]))
1) Prolin                  0.185453
2) Flavanoide              0.174751
3) Farbintensität          0.143920
4) OD280/OD315 des verdünnten Weins 0.136162
5) Alkohol                 0.118529
```

## 4.7 Zusammenfassung

Am Anfang dieses Kapitels haben wir nützliche Verfahren betrachtet, die gewährleisten, dass wir korrekt mit fehlenden Daten umgehen. Bevor wir einen Lernalgorithmus mit Daten füttern, müssen wir außerdem sicherstellen, dass die kategorialen Variablen richtig codiert sind. Sie haben außerdem erfahren, wie den Werten von ordinalen und nominalen Merkmalen Ganzzahlen zugeordnet werden.

Darüber hinaus haben wir uns kurz mit der L1-Regularisierung befasst, die dabei helfen kann, eine Überanpassung zu verhindern, indem sie die Komplexität des Modells reduziert. Als alternativen Ansatz zum Entfernen irrelevanter Merkmale haben wir einen sequenziellen Algorithmus zur Auswahl aussagekräftiger Merkmale einer Datensammlung eingesetzt.

Im nächsten Kapitel werden Sie einen weiteren nützlichen Ansatz zur Dimensionsreduktion kennenlernen: die Extrahierung von Merkmalen. Sie gestattet es, Merkmale in einen Merkmalsraum geringerer Dimensionalität abzubilden, anstatt sie wie bei der Auswahl von Merkmalen komplett zu entfernen.



# Datenkomprimierung durch Dimensionsreduktion

In Kapitel 4 haben Sie verschiedene Ansätze für die Dimensionsreduktion einer Datensammlung mithilfe unterschiedlicher Verfahren zur Auswahl von Merkmalen kennengelernt. Die *Merkmalsextraktion* ist ein alternativer Ansatz zur Auswahl von Merkmalen zwecks Dimensionsreduktion. In diesem Kapitel werden wir drei grundlegende Verfahren betrachten, die uns beim Zusammenfassen des Informationsgehalts einer Datensammlung helfen, indem sie in einen neuen Merkmalsraum transformiert wird, der eine niedrigere Dimensionalität besitzt als der ursprüngliche. Beim Machine Learning ist die Datenkomprimierung ein wichtiger Aspekt, der es ermöglicht, die stetig wachsenden Datens Mengen, die in unserem Technologiezeitalter erzeugt und gesammelt werden, zu speichern und zu analysieren.

Die Themen in diesem Kapitel sind:

- *Hauptkomponentenanalyse* (*PCA*, *Principal Component Analysis*) zur unüberwachten Datenkomprimierung
- *Lineare Diskriminanzanalyse* (*LDA*, *Linear Discriminant Analysis*) als ein überwachtes Verfahren zur Dimensionsreduktion, das die Trennbarkeit von Klassen maximiert
- Nichtlineare Dimensionsreduktion per Kernel-PCA (*KPCA*)

## 5.1 Unüberwachte Dimensionsreduktion durch Hauptkomponentenanalyse

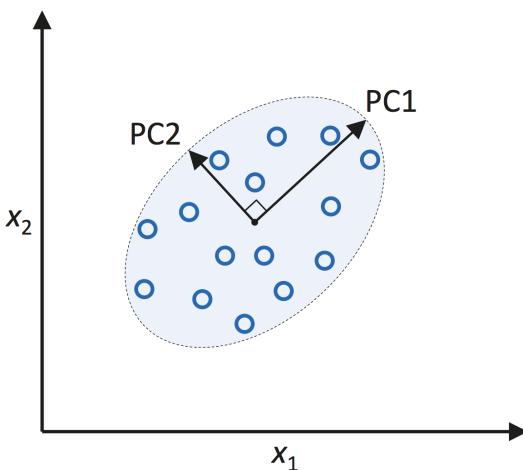
Ähnlich wie die Auswahl von Merkmalen können wir die Merkmalsextraktion zur Dimensionsreduktion einer Datensammlung nutzen. Beim Einsatz von Algorithmen zur Auswahl von Merkmalen (wie z.B. der sequenziellen Rückwärtsauswahl) haben wir die ursprünglichen Merkmale allerdings beibehalten, wohingegen die Daten bei der Merkmalsextraktion in einen neuen Merkmalsraum transformiert oder projiziert werden. Im Kontext der Dimensionsreduktion kann die Merkmalsextraktion als ein Ansatz zur Datenkomprimierung aufgefasst werden, der darauf abzielt, den Großteil der relevanten Informationen zu erhalten. Dieses Verfahren dient in der Praxis typischerweise dazu, die Effizienz der Berechnungen zu

erhöhen, kann aber auch die Vorhersagekraft verbessern, weil der *Fluch der Dimensionalität* weniger stark zum Tragen kommt – insbesondere dann, wenn wir es mit nicht regularisierten Modellen zu tun haben.

### 5.1.1 Schritte bei der Hauptkomponentenanalyse

In diesem Abschnitt betrachten wir die *Hauptkomponentenanalyse (PCA, Principal Component Analysis)*, ein unüberwachtes lineares Transformationsverfahren, das in vielen verschiedenen Bereichen Anwendung findet, vornehmlich zur Merkmalsextraktion und Dimensionsreduktion. Zu anderen Anwendungsbereichen der PCA gehören explorative Datenanalysen, das Entfernen des Rauschens aus Aktienhandelsdaten oder im Bereich Bioinformatik die Analyse des Genoms und die Genexpressionsanalyse.

Die PCA hilft uns dabei, anhand der Korrelationen zwischen verschiedenen Merkmalen Muster in den Daten aufzuspüren. Kurz und bündig ausgedrückt, wird hier versucht, die Richtungen maximaler Varianz in hochdimensionalen Daten zu finden und sie auf einen neuen Unterraum zu projizieren, der höchstens die gleiche Anzahl von Dimensionen besitzt wie der ursprüngliche Raum. Unter der Bedingung, dass die neuen Merkmalsachsen wie in der folgenden Abbildung orthogonal zueinander verlaufen, können die orthogonalen Achsen der Hauptkomponenten (*Principal Components, PC*) als die Richtung maximaler Varianz interpretiert werden. Hier sind  $x_1$  und  $x_2$  die ursprünglichen Merkmalsachsen und PC1 bzw. PC2 die Hauptkomponenten.



Wenn wir die PCA zur Dimensionsreduktion einsetzen, konstruieren wir eine  $d \times k$ -dimensionale Transformationsmatrix  $\mathbf{W}$ , die es uns ermöglicht, einen Vektor  $\mathbf{x}$  auf einen neuen  $k$ -dimensionalen Merkmalsunterraum abzubilden, der weniger Dimensionen als der ursprüngliche  $d$ -dimensionale Merkmalsraum besitzt:

$$\mathbf{x} = [x_1, x_2, \dots, x_d], \quad \mathbf{x} \in \mathbb{R}^d$$

$$\downarrow \mathbf{x}\mathbf{W}, \quad \mathbf{W} \in \mathbb{R}^{d \times k}$$

$$\mathbf{z} = [z_1, z_2, \dots, z_k], \quad \mathbf{z} \in \mathbb{R}^k$$

Die Transformation der ursprünglich  $d$ -dimensionalen Daten in den neuen  $k$ -dimensionalen Unterraum (typischerweise ist  $k << d$ ) hat zum Ergebnis, dass die erste Hauptkomponente die größtmögliche Varianz besitzt – und alle nachfolgenden Hauptkomponenten besitzen ebenfalls die größtmögliche Varianz, da sie mit den anderen Hauptkomponenten unkorreliert (orthogonal) sind – selbst wenn die Eingabemerkmale korrelieren, sind die resultierenden Hauptachsen orthogonal (unkorreliert). Beachten Sie, dass die Richtungen sehr empfindlich auf skalierte Daten reagieren. Wenn wir allen Merkmalen die gleiche Gewichtung beimessen möchten, müssen wir sie standardisieren, *befor* wir die PCA durchführen, sofern die Merkmale unterschiedliche Größenordnungen aufweisen.

Wir werden uns den PCA-Algorithmus zur Dimensionsreduktion gleich noch etwas genauer ansehen, fassen den Ansatz an dieser Stelle aber zunächst in einigen einfachen Schritten zusammen:

1. Standardisierung der  $d$ -dimensionalen Datenmenge
2. Konstruieren der Kovarianzmatrix
3. Zerlegung der Kovarianzmatrix in Eigenvektoren und Eigenwerte
4. Sortieren der Eigenwerte in absteigender Reihenfolge, um eine Rangliste der Eigenvektoren zu erhalten
5. Auswahl der  $k$  Eigenvektoren, die zu den  $k$  größten Eigenwerten gehören, wobei  $k$  die Dimensionalität des neuen Merkmalsunterraums angibt ( $k \leq d$ )
6. Konstruieren einer Projektionsmatrix  $\mathbf{W}$  aus den  $k$  in Schritt 5 ausgewählten Eigenvektoren
7. Transformation der  $d$ -dimensionalen Eingabemenge  $\mathbf{X}$  mit der Projektionsmatrix  $\mathbf{W}$ , um den neuen  $k$ -dimensionalen Merkmalsunterraum zu erhalten

In den folgenden Abschnitten werden wir schrittweise eine Hauptkomponentenanalyse mit Python vornehmen. Anschließend werden wir betrachten, wie sich eine Hauptkomponentenanalyse komfortabel mit scikit-learn durchführen lässt.

### 5.1.2 Schrittweise Extraktion der Hauptkomponenten

In diesem Abschnitt nehmen wir die ersten vier Schritte einer Hauptkomponentenanalyse in Angriff:

1. Standardisierung der Daten
2. Konstruktion der Kovarianzmatrix

3. Ermittlung der Eigenwerte und Eigenvektoren der Kovarianzmatrix
4. Sortieren der Eigenwerte in absteigender Reihenfolge, um eine Rangliste der Eigenvektoren zu erhalten.

Zunächst laden wir die in Kapitel 4 verwendete Wein-Datensammlung:

```
>>> import pandas as pd
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/
... machine-learning-databases/wine/wine.data', header=None)
```

Nun teilen wir die Wein-Daten in 70 Prozent Trainings- und 30 Prozent Testdaten auf und standardisieren sie:

```
>>> from sklearn.model_selection import train_test_split
>>> X, y = df_wine.iloc[:, 1:].values,
...         df_wine.iloc[:, 0].values
>>> X_train, X_test, y_train, y_test = \
...             train_test_split(X, y,
...                             test_size=0.3,
...                             stratify=y,
...                             random_state=0)
# Merkmale standardisieren
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> X_train_std = sc.fit_transform(X_train)
>>> X_test_std = sc.transform(X_test)
```

Nach Abschluss der obligatorischen Vorverarbeitungsschritte durch Ausführung des obigen Codes fahren wir mit dem zweiten Schritt fort: dem Konstruieren der Kovarianzmatrix. Die symmetrische  $d \times d$ -dimensionale Kovarianzmatrix, bei der  $d$  die Anzahl der Dimensionen der Datenmenge angibt, enthält paarweise die Kovarianzen der verschiedenen Merkmale. Die Kovarianz der Merkmale  $x_j$  und  $x_k$  kann beispielsweise durch folgende Gleichung berechnet werden:

$$\sigma_{jk} = \frac{1}{n} \sum_{i=1}^n (x_j^{(i)} - \mu_j)(x_k^{(i)} - \mu_k)$$

Hier sind  $\mu_j$  bzw.  $\mu_k$  die Mittelwerte der Merkmale  $j$  und  $k$ . Beachten Sie, dass die Mittelwerte null sind, wenn wir die Datenmenge standardisieren. Eine positive Kovarianz zweier Merkmale bedeutet, dass die Werte der Merkmale zusammen ansteigen bzw. sinken. Bei negativer Kovarianz gehen hingegen hohe Werte des einen Merkmals mit niedrigen Werten des anderen Merkmals einher und umgekehrt. Beispielsweise kann die Kovarianzmatrix dreier Merkmale folgendermaßen notiert werden:

$$\Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_2^2 & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_3^2 \end{bmatrix}$$

( $\Sigma$  steht hier für den griechischen Buchstaben *Sigma* und ist nicht mit dem Summenzeichen zu verwechseln.) Die Eigenvektoren der Kovarianzmatrix stellen die Hauptkomponenten dar, also die Richtungen der maximalen Varianz, während die dazugehörigen Eigenwerte deren Größe angeben. Bei der Wein-Datensammlung würde die  $13 \times 13$ -dimensionale Kovarianzmatrix jeweils 13 Eigenvektoren und Eigenwerte liefern.

Nun kommen wir zum dritten Schritt und berechnen die Paare der Eigenvektoren und Eigenwerte der Kovarianzmatrix. Von der linearen Algebra ist bekannt, dass ein Eigenvektor  $v$  folgende Bedingung erfüllt:

$$\Sigma v = \lambda v$$

Hier ist  $\lambda$  ein Skalar: der Eigenwert. Da die Berechnung von Eigenvektoren und Eigenwerten von Hand doch ziemlich mühsam ist, verwenden wir die NumPy-Funktion `linalg.eig`, um die Eigenvektor/Eigenwert-Paare der Kovarianzmatrix der Wein-Datensammlung zu erhalten:

```
>>> import numpy as np
>>> cov_mat = np.cov(X_train_std.T)
>>> eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
>>> print('\nEigenwerte \n%s' % eigen_vals)
Eigenwerte
[ 4.84274532  2.41602459  1.54845825  0.96120438  0.84166161
 0.6620634   0.51828472  0.34650377  0.3131368   0.10754642
 0.21357215  0.15362835  0.1808613 ]
```

Mit der NumPy-Funktion `cov` berechnen wir die Kovarianzmatrix der standardisierten Trainingsdatenmenge. Die Zerlegung in Eigenwerte und Eigenvektoren erfolgt mit der Funktion `linalg.eig`. Sie liefert einen Vektor (`eigen_vals`) mit 13 Eigenwerten und eine  $13 \times 13$ -dimensionale Matrix (`eigen_vecs`) zurück, in der die dazugehörigen Eigenvektoren als Spalten enthalten sind.

### Tipp

Die NumPy-Funktion `linalg.eig` ist zwar eigentlich dazu gedacht, symmetrische und nicht symmetrische quadratische Matrizen zu zerlegen, Sie werden jedoch feststellen, dass sie unter bestimmten Umständen komplexe Eigenwerte liefert.

Zur Zerlegung hermitescher Matrizen wurde die verwandte Funktion `linalg.eigh` implementiert, die für numerische Anwendungen mit symmetrischen Matrizen wie der Kovarianzmatrix besser geeignet ist. `linalg.eigh` liefert immer reelle Eigenwerte zurück.

### 5.1.3 Totale Varianz und Varianzaufklärung

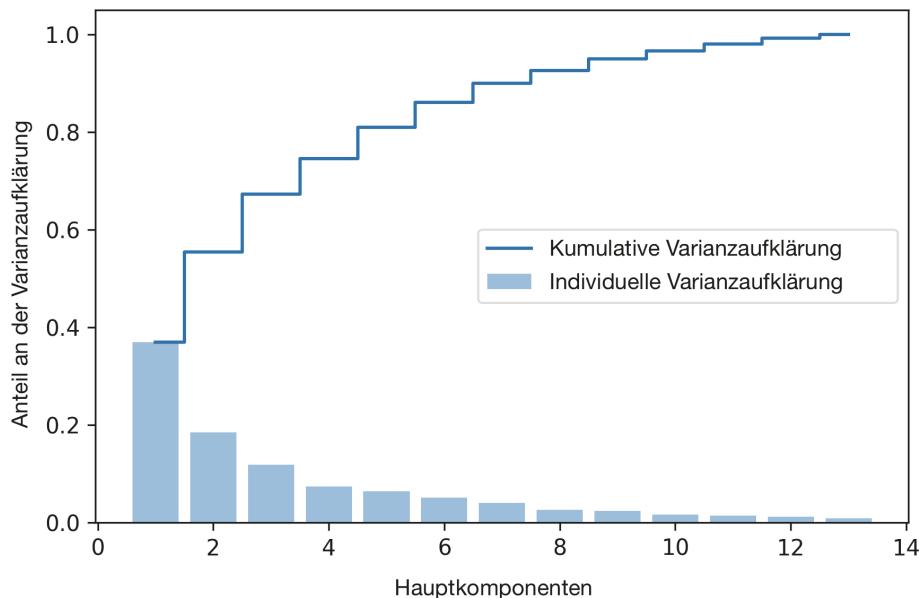
Da wir die Dimensionalität unserer Datensammlung reduzieren möchten, indem wir sie auf einen neuen Merkmalsumerraum abbilden, wählen wir diejenige Untermenge der Eigenvektoren (Hauptkomponenten) aus, die die meisten Informationen (Varianz) enthält. Die Eigenwerte legen die Größe der Eigenvektoren fest, daher müssen wir die Eigenwerte der Größe nach sortieren. Wir sind an den zu den  $k$  größten Eigenwerten zugehörigen Eigenvektoren interessiert. Aber bevor wir diese  $k$  informativsten Eigenvektoren ermitteln, plotten wir die Anteile der Eigenwerte an der Varianzaufklärung. Der Anteil eines Eigenwertes  $\lambda_j$  an der Varianzaufklärung ergibt sich einfach aus dem Quotienten dieses Wertes und der Summe aller Eigenwerte:

$$\frac{\lambda_j}{\sum_{j=1}^d \lambda_j}$$

Mit der NumPy-Funktion `cumsum` können wir die kumulierte Summe der Varianzaufklärungen berechnen und mit der `matplotlib`-Funktion `step` ausgeben:

```
>>> tot = sum(eigen_vals)
>>> var_exp = [(i / tot) for i in
...             sorted(eigen_vals, reverse=True)]
>>> cum_var_exp = np.cumsum(var_exp)
>>> import matplotlib.pyplot as plt
>>> plt.bar(range(1,14), var_exp, alpha=0.5, align='center',
...           label='Individuelle Varianzaufklärung')
>>> plt.step(range(1,14), cum_var_exp, where='mid',
...           label='Kumulative Varianzaufklärung')
>>> plt.ylabel('Anteil an der Varianzaufklärung')
>>> plt.xlabel('Hauptkomponenten')
>>> plt.legend(loc='best')
>>> plt.show()
```

Dem resultierenden Diagramm ist zu entnehmen, dass allein die erste Hauptkomponente für rund 40 Prozent der Varianz verantwortlich ist. Außerdem ist erkennbar, dass die beiden ersten Hauptkomponenten zusammen fast 60 Prozent der Varianz der Daten verursachen:



Das Diagramm erinnert zwar an die in Kapitel 4 mithilfe von Random Forests berechnete Bedeutung der Merkmale, wir dürfen aber nicht vergessen, dass es sich bei der PCA um eine unüberwachte Methode handelt und die Informationen über die Klassenbezeichnungen daher ignoriert werden. Ein Random Forest verwendet die Informationen über die Klassenzugehörigkeit, um die Unreinheit der Knoten zu berechnen. Die Varianz hingegen ist ein Maß für die Verteilung der Werte entlang einer Merkmalsachse.

#### 5.1.4 Merkmalstransformation

Nachdem wir die Kovarianzmatrix erfolgreich in Paare aus Eigenvektoren und Eigenwerten zerlegt haben, fahren wir nun mit den verbleibenden drei Schritten der Hauptachsentransformation der Wein-Datensammlung fort:

- Auswahl der  $k$  Eigenvektoren, die zu den  $k$  größten Eigenwerten gehören, wobei  $k$  die Dimensionalität des neuen Merkmalsunterraums angibt ( $k \leq d$ )
- Konstruieren einer Projektionsmatrix  $\mathbf{W}$  aus den  $k$  im vorhergehenden Schritt ausgewählten Eigenvektoren
- Transformation der  $d$ -dimensionalen Eingabemenge  $\mathbf{X}$  mit der Projektionsmatrix  $\mathbf{W}$ , um den neuen  $k$ -dimensionalen Merkmalsunterraum zu erhalten

Oder weniger technisch formuliert: Wir sortieren die Paare aus Eigenvektoren und Eigenwerten nach der Größe der Eigenwerte, konstruieren mit den ausgewählten Eigenvektoren eine Projektionsmatrix und verwenden diese, um die Daten in einen Unterraum niedrigerer Dimension zu transformieren.

Zunächst sortieren wir die Paare aus Eigenvektoren und Eigenwerten nach der Größe der Eigenwerte:

```
>>> # Liste von (eigenvalues, eigenvector)-Tupel anlegen
>>> eigen_pairs =[(np.abs(eigen_vals[i]),eigen_vecs[:,i])
...                 for i in range(len(eigen_vals))]
>>> # Die Tupel in absteigender Reihenfolge sortieren
>>> eigen_pairs.sort(reverse=True)
```

Nun ermitteln wir die beiden Eigenvektoren, die zu den beiden größten Eigenwerten gehören und zusammen für ungefähr 60 Prozent der Varianz der Datensammlung verantwortlich sind. Beachten Sie hier, dass wir zu Demonstrationszwecken nur zwei Eigenvektoren auswählen, denn wir werden die Daten später in einem zweidimensionalen Korrelationsdiagramm darstellen. In der Praxis muss man bei der Festlegung der Anzahl der Hauptkomponenten einen Kompromiss zwischen der Effizienz der Berechnung und der Leistung des Klassifizierers eingehen.

```
>>> w= np.hstack((eigen_pairs[0][1][:, np.newaxis],
...                  eigen_pairs[1][1][:, np.newaxis]))
>>> print('Matrix W:\n',w)
Matrix W:
[[-0.13724218  0.50303478]
 [ 0.24724326  0.16487119]
 [-0.02545159  0.24456476]
 [ 0.20694508 -0.11352904]
 [-0.15436582  0.28974518]
 [-0.39376952  0.05080104]
 [-0.41735106 -0.02287338]
 [ 0.30572896  0.09048885]
 [-0.30668347  0.00835233]
 [ 0.07554066  0.54977581]
 [-0.32613263 -0.20716433]
 [-0.36861022 -0.24902536]
 [-0.29669651  0.38022942]]
```

Durch das Ausführen des Codes haben wir anhand der beiden bedeutendsten Eigenvektoren eine  $13 \times 2$ -dimensionale Projektionsmatrix  $W$  erstellt.

### Hinweis

Je nachdem, welche Version von NumPy und LAPACK Sie verwenden, kann es sein, dass die Matrix  $W$  ein umgekehrtes Vorzeichen besitzt. Das stellt jedoch kein Problem dar. Wenn  $v$  ein Eigenvektor der Matrix  $\Sigma$  ist, gilt:

$$\Sigma v = \lambda v$$

Hier ist  $\lambda$  der Eigenwert und zu  $\lambda$  gehört ebenfalls ein Eigenvektor mit demselben Eigenwert, denn es gilt:

$$\Sigma \cdot (-v) = -v \Sigma = -\lambda v = \lambda \cdot (-v)$$

Mit dieser Projektionsmatrix können wir jetzt eine Stichprobe  $x$  (repräsentiert durch einen  $1 \times 13$ -dimensionalen Zeilenvektor) in den PCA-Unterraum transformieren und erhalten  $x'$  – einen nunmehr zweidimensionalen Vektor, der zwei neue Merkmale besitzt:

$$x' = xW$$

```
>>> X_train_std[0].dot(w)
array([2.38299011,  0.45458499])
```

Auf ähnliche Weise können wir durch die Berechnung des Skalarprodukts die gesamte  $124 \times 13$ -dimensionale Trainingsdatensammlung auf die beiden Hauptkomponenten abbilden:

$$X' = XW$$

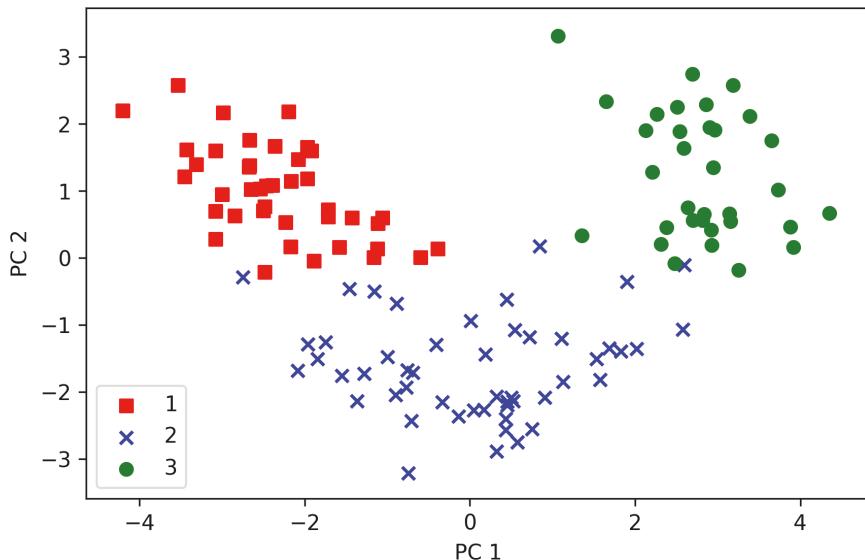
```
>>> X_train_pca = X_train_std.dot(w)
```

Zum Abschluss visualisieren wir die transformierte Wein-Datensammlung, die nun als  $124 \times 2$ -dimensionale Matrix gespeichert ist, als zweidimensionales Streudiagramm:

```
>>> colors = ['r', 'b', 'g']
>>> markers = ['s', 'x', 'o']
>>> for l, c, m in zip(np.unique(y_train), colors, markers):
...     plt.scatter(X_train_pca[y_train==l, 0],
...                 X_train_pca[y_train==l, 1],
...                 c=c, label=l, marker=m)
>>> plt.xlabel('PC 1')
>>> plt.ylabel('PC 2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

Wie der nachstehenden Abbildung zu entnehmen ist, sind die Daten auf der X-Achse (der ersten Hauptkomponente) weiter verteilt als auf der Y-Achse (der zweiten Hauptkomponente), was mit dem vorhin erstellten Plot der Varianzauf-

klärung übereinstimmt. Es ist jedoch intuitiv erkennbar, dass ein linearer Klassifizierer die Klassen vermutlich gut voneinander trennen kann.



In der Abbildung sind zwar die Klassenbezeichnungen angegeben, allerdings dürfen wir hierbei nicht vergessen, dass die PCA ein unüberwachtes Verfahren ist, das von den Klassenbezeichnungen keinen Gebrauch macht.

### 5.1.5 Hauptkomponentenanalyse mit scikit-learn

Der langwierige Ansatz des vorangegangenen Abschnitts hilft zwar, die interne Funktionsweise einer PCA besser zu verstehen, dennoch wollen wir an dieser Stelle einmal etwas genauer betrachten, wie man die in scikit-learn implementierte PCA-Klasse verwendet. Die PCA-Klasse gehört zu den Transformer-Klassen von scikit-learn, bei denen wir das Modell an die Trainingsdaten anpassen, bevor wir sowohl die Trainings- als auch die Testdaten anhand derselben Modellparameter transformieren.

Wir verwenden die PCA von scikit-learn in diesem Beispiel mit den Wein-Trainingsdaten, klassifizieren die transformierte Stichprobe mittels logistischer Regression und visualisieren schließlich die Entscheidungsbereiche mit der in Kapitel 2 definierten `plot_decision_region`-Funktion:

```
from matplotlib.colors import ListedColormap

def plot_decision_regions(X, y, classifier, resolution=0.02):
```

```

# Marker und Farben einstellen
markers = ('s', 'x', 'o', '^', 'v')
colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
cmap = ListedColormap(colors[:len(np.unique(y))])

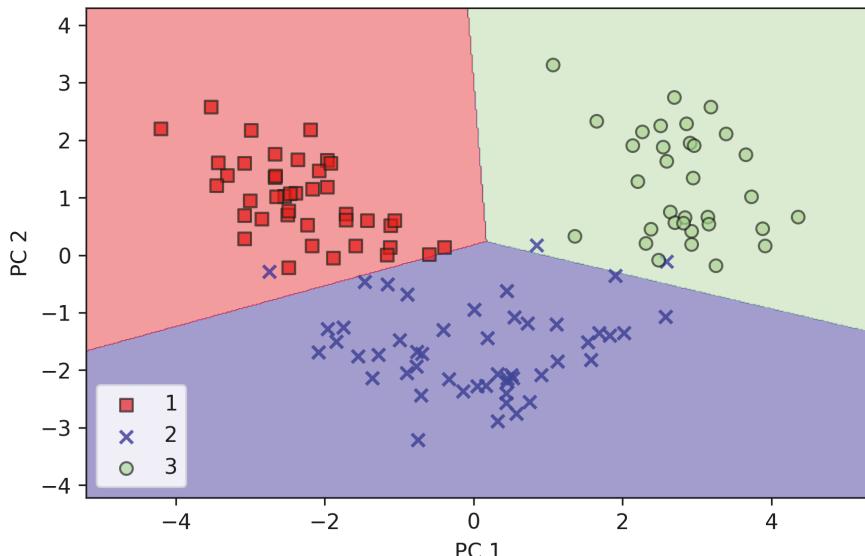
# Plotten der Entscheidungsbereiche
x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max,
                                 resolution), np.arange(x2_min, x2_max, resolution))
Z = classifier.predict(np.array([xx1.ravel(),
                                 xx2.ravel()]).T)
Z = Z.reshape(xx1.shape)
plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
plt.xlim(xx1.min(), xx1.max())
plt.ylim(xx2.min(), xx2.max())

# Plotten der Klassenobjekte
for idx, cl in enumerate(np.unique(y)):
    plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                alpha=0.8, c=cmap(idx),
                edgecolor='black',
                marker=markers[idx], label=cl)

>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=2)
>>> lr = LogisticRegression()
>>> X_train_pca = pca.fit_transform(X_train_std)
>>> X_test_pca = pca.transform(X_test_std)
>>> lr.fit(X_train_pca, y_train)
>>> plot_decision_regions(X_train_pca, y_train,
...                         classifier=lr)
>>> plt.xlabel('PC1')
>>> plt.ylabel('PC2')
>>> plt.legend(loc='lower left')
>>> plt.show()

```

Nach der Ausführung des Codes sollten nun die auf die beiden Hauptkomponenten reduzierten Entscheidungsbereiche für das Trainingsmodell angezeigt werden.

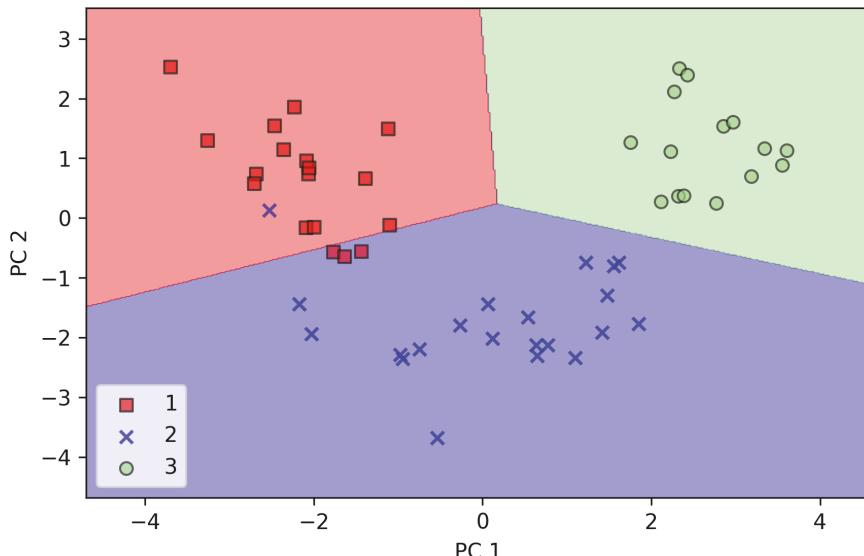


Wenn wir die PCA-Projektion von scikit-learn mit unserer eigenen PCA-Implementierung vergleichen, fällt auf, dass obiger Plot ein Spiegelbild der vorhergehenden PCA mit unserem schrittweisen Ansatz ist. Das liegt nicht etwa daran, dass eine der beiden Implementierungen fehlerhaft wäre.

Der Grund hierfür ist, dass Eigenvektoren je nach verwendetem Lösungsweg ein positives oder negatives Vorzeichen besitzen können. Es spielt zwar keine Rolle, aber wir könnten die Spiegelung rückgängig machen, indem wir die Daten mit -1 multiplizieren. Beachten Sie, dass Eigenvektoren typischerweise auf die Länge 1 skaliert werden. Der Vollständigkeit halber geben wir auch noch die Entscheidungsbereiche der logistischen Regression für die transformierte Testdatenmenge aus, um zu überprüfen, ob sie die Klassen gut voneinander trennen kann:

```
>>> plot_decision_regions(X_test_pca, y_test, classifier=lr)
>>> plt.xlabel('PC1')
>>> plt.ylabel('PC2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

Nach der Ausgabe der Entscheidungsbereiche durch die Ausführung des obigen Codes wird deutlich, dass die logistische Regression mit diesem kleinen zweidimensionalen Merkmalsunterraum ziemlich gut funktioniert und nur einige wenige Objekte der Testdatenmenge fehlklassifiziert.



Wenn wir nun an den Anteilen an der Varianzaufklärung der verschiedenen Hauptkomponenten interessiert wären, könnten wir die PCA-Klasse einfach mit dem Parameter `n_components=None` initialisieren, damit alle Hauptkomponenten erhalten bleiben, und über das Attribut `explained_variance_ratio_` auf die Anteile an der Varianzaufklärung zugreifen:

```
>>> pca = PCA(n_components=None)
>>> X_train_pca = pca.fit_transform(X_train_std)
>>> pca.explained_variance_ratio_
array([0.36951469, 0.18434927, 0.11815159, 0.07334252,
       0.06422108, 0.05051724, 0.03954654, 0.02643918,
       0.02389319, 0.01629614, 0.01380021, 0.01172226,
       0.00820609])
```

Beachten Sie, dass wir `n_components` bei der Initialisierung der PCA-Klasse auf `None` gesetzt haben, damit alle Hauptkomponenten in sortierter Reihenfolge zurückgegeben werden, anstatt eine Dimensionsreduktion durchzuführen.

## 5.2 Überwachte Datenkomprimierung durch lineare Diskriminanzanalyse

Die *lineare Diskriminanzanalyse* (*LDA*, *Linear Discriminant Analysis*) kann zur Merkmalsextraktion genutzt werden, um die Effizienz der Berechnungen zu erhöhen und bei nicht regularisierten Modellen das Ausmaß der Überanpassung aufgrund des Fluchs der Dimensionalität zu verringern.

Das der LDA zugrunde liegende Konzept ist dem der PCA sehr ähnlich: Die PCA versucht, die orthogonalen Komponentenachsen der maximalen Varianz einer Datenmenge zu finden, und die LDA versucht, denjenigen Merkmalsraum zu finden, der die Trennbarkeit der Klassen optimiert. In den folgenden Abschnitten werden wir die Gemeinsamkeiten von LDA und PCA eingehender betrachten und den LDA-Ansatz schrittweise nachvollziehen.

### 5.2.1 Hauptkomponentenanalyse vs. lineare Diskriminanzanalyse

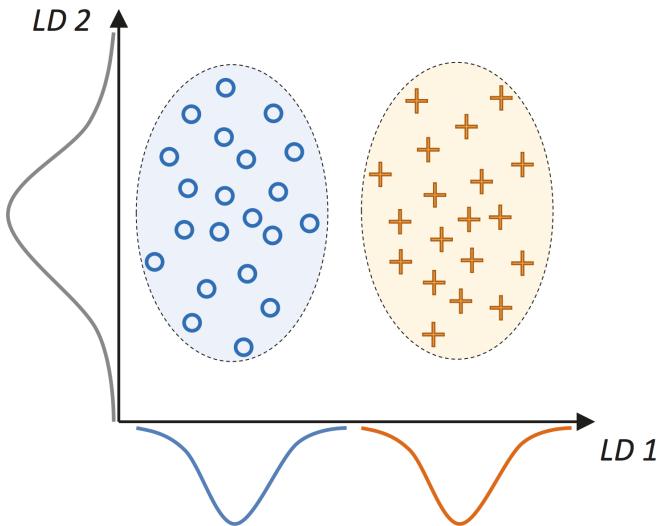
Sowohl LDA als auch PCA sind lineare Transformationsverfahren, mit denen sich die Anzahl der Dimensionen einer Datenmenge reduzieren lässt. Bei Ersterem handelt es sich um einen überwachten Algorithmus, Letzterer hingegen ist unüberwacht.

Man könnte daher meinen, dass die LDA der PCA bei Klassifizierungsaufgaben als Merkmalsextraktionsverfahren überlegen ist, allerdings berichtet A.M. Martinez, dass die Vorverarbeitung per PCA bei Bilderkennungsaufgaben in manchen Fällen tendenziell bessere Klassifizierungsergebnisse liefert, beispielsweise wenn die Klassen nur aus einigen wenigen Exemplaren bestehen (A.M. Martinez und A.C. Kak, *PCA Versus LDA, IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(2):228-233, 2001).

#### Tipp

Die LDA wird zwar manchmal auch als *Fisher-LDA* bezeichnet, tatsächlich formulierte Ronald A. Fisher 1936 aber ursprünglich »nur« *Fishers lineare Diskriminante* für Klassifizierungsaufgaben mit zwei Klassen (R.A. Fisher, *The Use of Multiple Measurements in Taxonomic Problems*, Annals of Eugenics, 7(2):179-188, 1936). Dieses Verfahren wurde dann 1948 von C. Radhakrishna Rao unter Annahme gleicher Klassenkovarianzen und normalverteilter Klassen für Klassifizierungsaufgaben mit mehreren Klassen verallgemeinert – und das bezeichnen wir heutzutage als LDA (C.R. Rao, *The Utilization of Multiple Measurements in Problems of Biological Classification*, Journal of the Royal Statistical Society, Series B (Methodological), 10(2):159-203, 1948).

Die folgende Abbildung fasst das Konzept einer LDA für eine Aufgabe mit zwei Klassen zusammen. Die zur Klasse 1 und 2 zugehörigen Objekte sind als Kreise bzw. Kreuze dargestellt.



Bei der LDA wird vorausgesetzt, dass die Daten normalverteilt sind. Außerdem nehmen wir an, dass die Kovarianzmatrizen der Klassen identisch und die Merkmale statistisch voneinander unabhängig sind. Aber selbst wenn eine oder mehrere dieser Annahmen nicht exakt zutreffen, kann die LDA zur Dimensionsreduktion noch einigermaßen gut funktionieren (R.O. Duda, P.E. Hart und D.G. Stork, *Pattern Classification*, 2nd. Edition. New York, 2001).

### 5.2.2 Die interne Funktionsweise der linearen Diskriminanzanalyse

Bevor wir die Implementierung des Codes eingehender betrachten, fassen wir die entscheidenden Schritte des LDA-Ansatzes noch einmal kurz zusammen:

1. Standardisierung der  $d$ -dimensionalen Datenmenge ( $d$  ist die Anzahl der Merkmale)
2. Berechnung des  $d$ -dimensionalen Mittelwertvektors jeder Klasse
3. Konstruieren der Streumatrix  $\mathbf{S}_B$  (Streuung zwischen den Klassen) und der Streumatrix  $\mathbf{S}_w$  (Streuung innerhalb einer Klasse)
4. Berechnung der Eigenvektoren und der zugehörigen Eigenwerte der Matrix  $\mathbf{S}_w^{-1} \mathbf{S}_B$
5. Sortieren der Eigenwerte in absteigender Reihenfolge, um eine Rangliste der Eigenvektoren zu erstellen.
6. Auswahl der  $k$  Eigenvektoren, die zu den  $k$  größten Eigenwerten gehören und Konstruktion einer  $d \times k$ -dimensionalen Transformationsmatrix  $\mathbf{W}$ ; die Eigenvektoren sind die Spalten dieser Matrix
7. Projektion der Daten auf einen neuen Merkmalsunterraum durch Anwendung der Transformationsmatrix  $\mathbf{W}$

Wie man sieht, ist die LDA der PCA ziemlich ähnlich, und zwar in dem Sinn, dass eine Matrix in Eigenwerte und Eigenvektoren zerlegt werden kann, die einen neuen, niedriger dimensionalen Merkmalsraum bilden. Allerdings berücksichtigt die LDA, wie bereits erwähnt, die Klassenbezeichnungen, die in Form der in Schritt 2 berechneten Mittelwertvektoren vorliegen. In den folgenden Abschnitten werden wir diese sieben Schritte ausführlicher betrachten und durch anschaulichen Code implementieren.

### 5.2.3 Berechnung der Streumatrizen

Da wir die Merkmale der Wein-Datensammlung bereits im Abschnitt über PCA standardisiert haben, können wir den ersten Schritt überspringen und mit der Berechnung der Mittelwertvektoren fortfahren, die wir dann zur Konstruktion der Streumatrizen für die Streuung innerhalb einer Klasse bzw. die Streuung zwischen den Klassen verwenden. Die Vektoren  $\mathbf{m}_i$  speichern den jeweiligen Mittelwert  $\mu_m$  der Merkmale der Klasse  $i$ :

$$\mathbf{m}_i = \frac{1}{n_i} \sum_{x \in D_i}^c \mathbf{x}_m$$

Damit ergeben sich drei Mittelwertvektoren:

$$\mathbf{m}_i = \begin{bmatrix} \mu_{i, \text{Alkohol}} \\ \mu_{i, \text{Apfelsäure}} \\ \vdots \\ \mu_{i, \text{Prolin}} \end{bmatrix} \quad i \in \{1, 2, 3\}$$

```
>>> np.set_printoptions(precision=4)
>>> mean_vecs = []
>>> for label in range(1,4):
...     mean_vecs.append(np.mean(
...         X_train_std[y_train==label], axis=0))
...     print('MV %s: %s\n' %(label, mean_vecs[label-1]))
MV 1: [ 0.9066 -0.3497  0.3201 -0.7189  0.5056  0.8807
0.9589 -0.5516  0.5416  0.2338  0.5897  0.6563  1.2075]
MV 2: [-0.8749 -0.2848 -0.3735  0.3157 -0.3848 -0.0433
0.0635 -0.0946  0.0703 -0.8286  0.3144  0.3608 -0.7253]
MV 3: [ 0.1992  0.866   0.1682  0.4148 -0.0451 -1.0286
-1.2876  0.8287 -0.7795  0.9649 -1.209  -1.3622 -0.4013]
```

Mit den Mittelwertvektoren können wir nun die Streumatrix  $\mathbf{S}_W$  berechnen:

$$\mathbf{S}_W = \sum_{i=1}^c \mathbf{S}_i$$

Die Berechnung erfolgt als die Summierung der einzelnen Streumatrizen  $\mathbf{S}_i$  der verschiedenen Klassen  $i$ :

$$\mathbf{S}_i = \sum_{x \in D_i}^c (\mathbf{x} - \mathbf{m}_i)(\mathbf{x} - \mathbf{m}_i)^T$$

```
>>> d = 13 # Anzahl der Merkmale
>>> S_W = np.zeros((d, d))
>>> for label,mv in zip(range(1,4), mean_vecs):
...     class_scatter = np.zeros((d, d))
...     for row in X_train_std[y_train == label]:
...         row, mv = row.reshape(d, 1), mv.reshape(d, 1)
...         class_scatter += (row-mv).dot((row-mv).T)
...     S_W += class_scatter
>> print('Streumatrix innerhalb der Klasse: %sx%s'
...       % (S_W.shape[0], S_W.shape[1]))
Streumatrix innerhalb der Klasse: 13x13
```

Bei der Berechnung der Streumatrizen gehen wir eigentlich davon aus, dass die Klassenbezeichnungen in der Trainingsdatenmenge gleichmäßig verteilt sind. Die Ausgabe der Anzahlen zeigt jedoch, dass diese Annahme gar nicht zutrifft:

```
>>> print('Verteilung der Klassenbezeichnungen: %s'
...       % np.bincount(y_train)[1:])
Verteilung der Klassenbezeichnungen: [41 50 33]
```

Wir sollten daher die einzelnen Streumatrizen  $\mathbf{S}_i$  skalieren, bevor wir sie zur Streumatrix  $\mathbf{S}_W$  summieren. Wenn wir die Streumatrix durch die Anzahl der in den Klassen vorhandenen Exemplare  $N_i$  teilen, stellen wir fest, dass die Berechnung derjenigen der Kovarianzmatrix  $\Sigma_i$  entspricht. Tatsächlich ist die Kovarianzmatrix eine normierte Version der Streumatrix:

$$\Sigma_i = \frac{1}{N_i} \mathbf{S}_W = \frac{1}{N_i} \sum_{x \in D_i}^c (\mathbf{x} - \mathbf{m}_i)(\mathbf{x} - \mathbf{m}_i)^T$$

```
>>> d = 13 # Anzahl der Merkmale
>>> S_W = np.zeros((d, d))
>>> for label,mv in zip(range(1, 4), mean_vecs):
...     class_scatter = np.cov(X_train_std[y_train==label].T)
...     S_W += class_scatter
>> print('Skalierte Streumatrix innerhalb der Klasse: %sx%s'
...       % (S_W.shape[0], S_W.shape[1]))
Skalierte Streumatrix innerhalb der Klasse: 13x13
```

Nachdem wir die skalierte Streumatrix innerhalb der Klasse (bzw. die Kovarianzmatrix) berechnet haben, können wir mit dem nächsten Schritt fortfahren und die Streumatrix  $S_B$  für die Streuung zwischen den Klassen berechnen:

$$S_B = \sum_{i=1}^c N_i (\mathbf{m}_i - \mathbf{m})(\mathbf{m}_i - \mathbf{m})^T$$

Hier ist  $\mathbf{m}$  der Mittelwert insgesamt, der die Objekte aller Klassen berücksichtigt.

```
>>> mean_overall = np.mean(X_train_std, axis=0)
>>> d = 13 # Anzahl der Merkmale
>>> S_B = np.zeros((d, d))
>>> for i,mean_vec in enumerate(mean_vecs):
...     n = X_train[y_train==i+1, :].shape[0]
...     mean_vec = mean_vec.reshape(d, 1) # Als Spaltenvektor
...     mean_overall = mean_overall.reshape(d, 1)
...     S_B += n * (mean_vec - mean_overall).dot(
...                 (mean_vec - mean_overall).T)
print('Streumatrix für die Streuung zwischen Klassen: %sx%s'
...      % (S_B.shape[0], S_B.shape[1]))
Streumatrix für die Streuung zwischen Klassen: 13x13
```

## 5.2.4 Auswahl linearer Diskriminanten für den neuen Merkmalsunterraum

Die noch verbleibenden Schritte bei der LDA sind mit denen der PCA vergleichbar. Statt eine Eigenwertzerlegung der Kovarianzmatrix durchzuführen, müssen wir hier allerdings das allgemeine Eigenwertproblem der Matrix  $S_w^{-1} S_B$  lösen.

```
>>> eigen_vals, eigen_vecs =
...             np.linalg.eig(np.linalg.inv(S_W).dot(S_B))
```

Nach der Berechnung der Paare aus Eigenvektoren und Eigenwerten können wir die Eigenwerte in absteigender Reihenfolge sortieren:

```
>>> eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:,i])
...                  for i in range(len(eigen_vals))]
>>> eigen_pairs = sorted(eigen_pairs,
...                       key=lambda k: k[0], reverse=True)
>>> print('Eigenwerte in absteigender Reihenfolge:\n')
>>> for eigen_val in eigen_pairs:
...     print(eigen_val[0])

Eigenwerte in absteigender Reihenfolge:
```

```
349.617808906
172.76152219
3.78531345125e-14
2.11739844822e-14
1.51646188942e-14
1.51646188942e-14
1.35795671405e-14
1.35795671405e-14
7.58776037165e-15
5.90603998447e-15
5.90603998447e-15
2.25644197857e-15
0.0
```

Bei der LDA beträgt die Anzahl der linearen Diskriminanten höchstens  $c - 1$ , wobei  $c$  die Anzahl der Klassenbezeichnungen angibt, denn die Streumatrix  $S_B$  für die Streuung zwischen den Klassen ist die Summe von  $c$  (oder weniger) Matrizen des Rangs 1. Tatsächlich gibt es nur zwei von null verschiedene Eigenwerte (die übrigen Eigenwerte sind nicht exakt null, aber das liegt an der Fließkommazahlenarithmetik in NumPy).

### Tipp

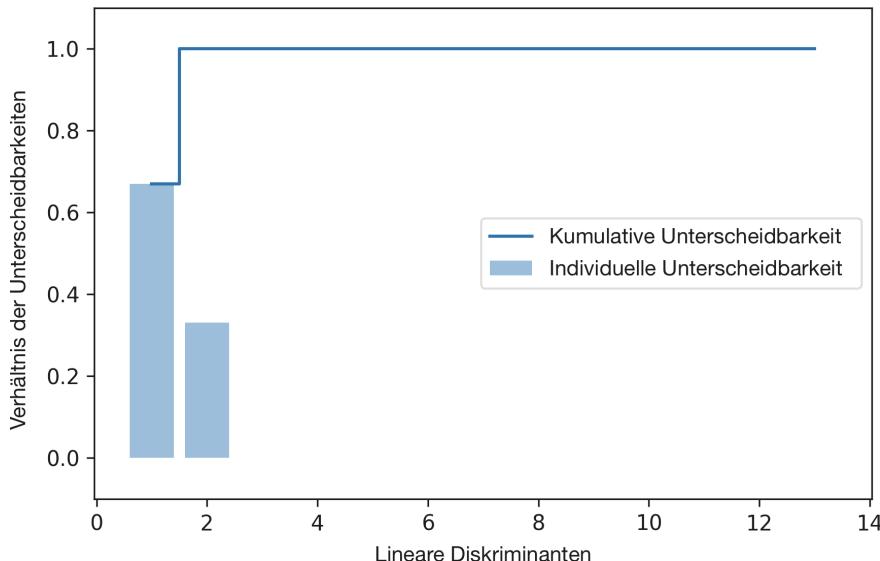
Beachten Sie, dass die Kovarianzmatrix im Fall perfekter Kollinearität (alle Punkte liegen auf einer geraden Linie) vom Rang eins ist, was zu nur einem Eigenvektor mit nicht von null verschiedenem Eigenwert führen würde.

Um zu ermitteln, wie viele der Informationen, durch die sich die Klassen unterscheiden, von den linearen Diskriminanten (Eigenvektoren) erfasst werden, plotten wir die Diskriminanten in absteigender Reihenfolge der Eigenwerte – ganz ähnlich wie beim Plot der Varianzaufklärung im Abschnitt über die PCA. Der Einfachheit halber bezeichnen wir die Informationen, durch die sich die Klassen von einander unterscheiden, als »Unterscheidbarkeit«.

```
>>> tot = sum(eigen_vals.real)
>>> discr = [(i / tot) for i in sorted(eigen_vals.real,
...                                         reverse=True)]
>>> cum_discr = np.cumsum(discr)
>>> plt.bar(range(1, 14), discr, alpha=0.5, align='center',
...           label='Individuelle Unterscheidbarkeit')
>>> plt.step(range(1, 14), cum_discr, where='mid',
...           label='Kumulative Unterscheidbarkeit')
>>> plt.ylabel('Verhältnis der Unterscheidbarkeiten')
>>> plt.xlabel('Lineare Diskriminanten')
```

```
>>> plt.ylim([-0.1, 1.1])
>>> plt.legend(loc='best')
>>> plt.show()
```

Wie Sie dem Diagramm entnehmen können, erfassen allein die beiden ersten linearen Diskriminanten 100 Prozent der nützlichen Informationen der Wein-Datensammlung.



Nun verknüpfen wir die beiden informativsten Eigenvektorspalten, um die Transformationsmatrix  $W$  zu erstellen:

```
>>> w = np.hstack((eigen_pairs[0][1][:, np.newaxis].real,
...                  eigen_pairs[1][1][:, np.newaxis].real))
>>> print('Matrix W:\n', w)
Matrix W:
[[ -0.1481 -0.4092]
 [  0.0908 -0.1577]
 [ -0.0168 -0.3537]
 [  0.1484  0.3223]
 [ -0.0163 -0.0817]
 [  0.1913  0.0842]
 [ -0.7338  0.2823]
 [ -0.075   -0.0102]
 [  0.0018  0.0907]
 [  0.294   -0.2152]
 [ -0.0328  0.2747]]
```

```
[[-0.3547 -0.0124]
 [-0.3915 -0.5958]]
```

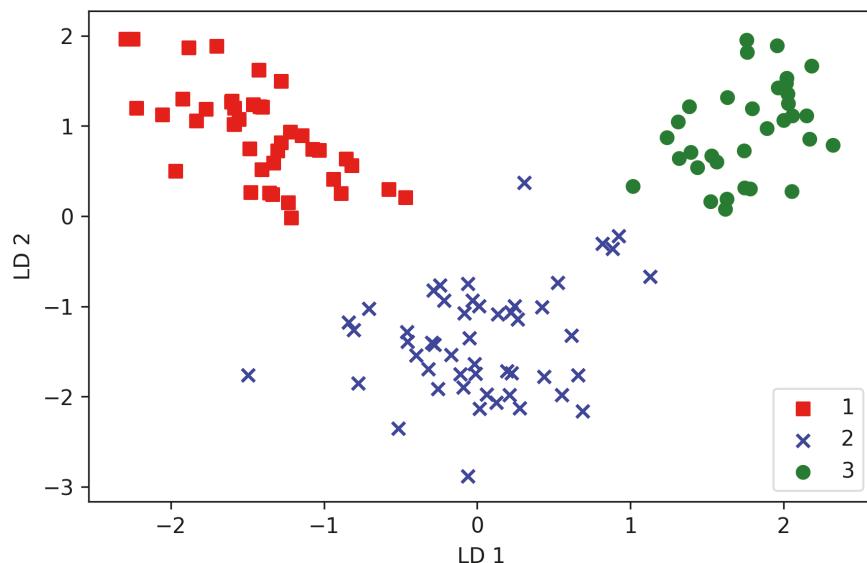
### 5.2.5 Projektion in den neuen Merkmalsraum

Mit der im vorangegangenen Abschnitt erstellten Transformationsmatrix  $W$  können wir die Trainingsdatenmenge transformieren, indem wir die Matrizen miteinander multiplizieren:

$$X' = XW$$

```
>>> X_train_lda = X_train_std.dot(w)
>>> colors = ['r', 'b', 'g']
>>> markers = ['s', 'x', 'o']
>>> for l, c, m in zip(np.unique(y_train), colors, markers):
...     plt.scatter(X_train_lda[y_train==l, 0],
...                 X_train_lda[y_train==l, 1] * (-1),
...                 c=c, label=l, marker=m)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='upper right')
>>> plt.show()
```

Wie Sie der Abbildung entnehmen können, sind die drei Wein-Klassen im neuen Merkmalsunterraum nun vollständig linear trennbar.



### 5.2.6 LDA mit scikit-learn

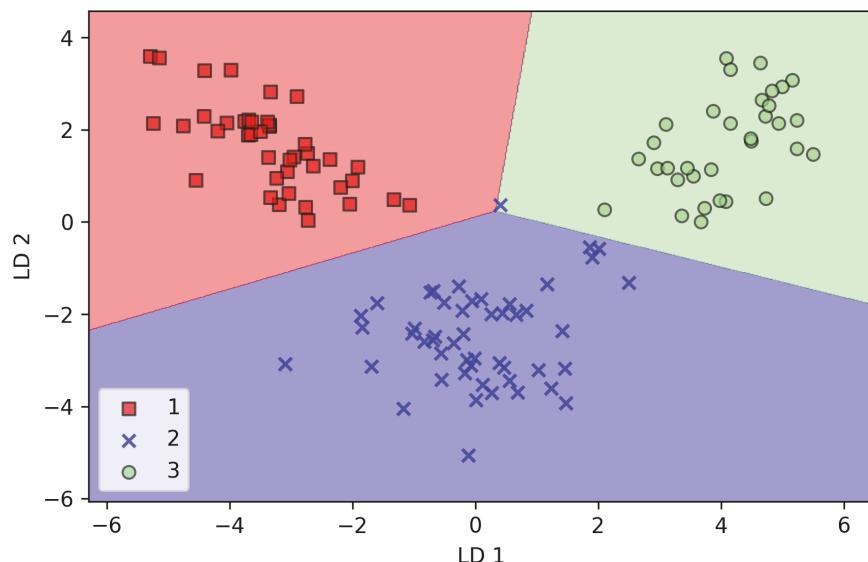
Die schrittweise Implementierung war eine gute Übung, um die Funktionsweise der LDA und die Unterschiede zwischen LDA und PCA zu verstehen. Sehen wir uns nun die in scikit-learn implementierte LDA-Klasse an:

```
>>> from sklearn.discriminant_analysis import
...     LinearDiscriminantAnalysis as LDA
>>> lda = LDA(n_components=2)
>>> X_train_lda = lda.fit_transform(X_train_std, y_train)
```

Im Folgenden überprüfen wir, wie die logistische Regression die Trainingsdatenmenge, die nach der LDA-Transformation von verringriger Dimensionalität ist, verarbeitet:

```
>>> lr = LogisticRegression()
>>> lr = lr.fit(X_train_lda, y_train)
>>> plot_decision_regions(X_train_lda,y_train, classifier=lr)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

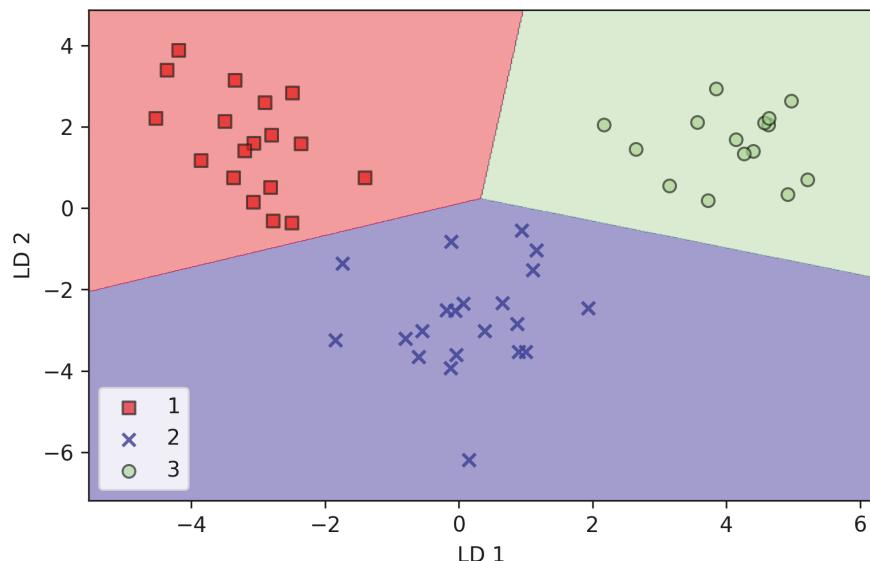
Im resultierenden Diagramm ist erkennbar, dass eins der zur Klasse 2 zugehörigen Objekte fehlklassifiziert wird:



Wir könnten die Entscheidungsgrenzen vermutlich verschieben, indem wir die Regularisierungsstärke vermindern, damit das logistische Regressionsmodell alle Objekte der Trainingsdatenmenge korrekt klassifiziert. Sehen wir uns aber zunächst einmal das Ergebnis für die Testdatenmenge an:

```
>>> X_test_lda = lda.transform(X_test_std)
>>> plot_decision_regions(X_test_lda, y_test, classifier=lr)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

Wie Sie sehen, ist der Klassifizierer für die logistische Regression nun in der Lage, sämtliche Objekte der Testdatenmenge anhand eines nur zweidimensionalen Merkmalsraums korrekt einzuordnen, anstatt auf die 13 ursprünglichen Wein-Merkmale zurückgreifen zu müssen.

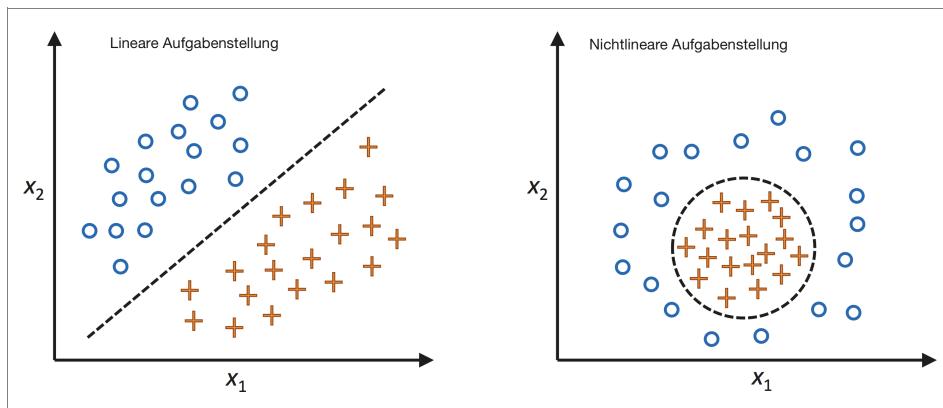


## 5.3 Kernel-Hauptkomponentenanalyse für nichtlineare Zuordnungen verwenden

Viele Lernalgorithmen stellen Annahmen über die lineare Trennbarkeit der Eingabedaten an. Beim Perzeptron ist es sogar erforderlich, dass die Trainingsdaten vollständig linear trennbar sind, damit es konvergiert. Andere der bislang vorgestellten Algorithmen setzen voraus, dass die Daten aufgrund von Rauschen nicht

vollständig linear trennbar sind – darunter auch Adaline (ADAptive LInear NEuron), die logistische Regression sowie (typische) Support Vector Machines (SVMs), um nur einige zu nennen.

Wenn wir es allerdings mit nichtlinearen Aufgabenstellungen zu tun haben, denen man in der Praxis recht häufig begegnet, sind lineare Transformationsverfahren zur Dimensionsreduktion wie PCA oder LDA oft nicht die beste Wahl. In diesem Abschnitt betrachten wir eine Kernel-PCA, die mit dem in Kapitel 3 vorgestellten Konzept einer Kernel-SVM verwandt ist. Beim Einsatz einer Kernel-PCA werden Sie erfahren, wie sich nicht linear trennbare Daten in einen neuen, niedrigdimensionalen Unterraum transformieren lassen, der für lineare Klassifizierer geeignet ist.



### 5.3.1 Kernel-Funktionen und der Kernel-Trick

Bei der Erörterung der Kernel-SVMs in Kapitel 3 haben Sie erfahren, dass wir nichtlineare Aufgabenstellungen in Angriff nehmen können, indem wir die Daten in einen höherdimensionalen Merkmalsraum projizieren, in dem die Klassen linear separierbar sind. Um die Elemente  $x \in \mathbb{R}^d$  in den  $k$ -dimensionalen Unterraum zu transformieren, haben wir eine nichtlineare Zuordnungsfunktion  $\phi$  definiert:

$$\phi: \mathbb{R}^d \rightarrow \mathbb{R}^k \quad (k > d)$$

Wir können uns  $\phi$  als eine Funktion vorstellen, die nichtlineare Kombinationen der ursprünglichen Merkmale erstellt, um die ursprünglich  $d$ -dimensionale Datenmenge auf einen größeren,  $k$ -dimensionalen Merkmalsraum abzubilden. Wenn wir beispielsweise einen Merkmalsvektor  $x \in \mathbb{R}^d$  ( $x$  ist ein aus  $d$  Merkmalen bestehender Spaltenvektor) mit zwei Dimensionen ( $d = 2$ ) betrachten, könnte die Zuordnung zu einem 3-D-Raum folgendermaßen aussehen:

$$\mathbf{x} = [x_1, x_2]^T$$

$$\downarrow \phi$$

$$\mathbf{z} = [x_1^2, \sqrt{2x_1x_2}, x_2^2]^T$$

Mit anderen Worten: Per Kernel-PCA führen wir eine nichtlineare Zuordnung durch, bei der die Daten unter Verwendung der Standard-PCA in einen höherdimensionalen Raum transformiert werden, um sie auf einen niedrigerdimensionalen Raum zurückzuprojizieren, in dem sie durch einen linearen Klassifizierer trennbar sind (vorausgesetzt, die Objekte lassen sich im Eingaberaum nach Dichte separieren). Der Nachteil dieses Ansatzes besteht darin, dass er sehr rechenaufwendig ist – und hier kommt der *Kernel-Trick* ins Spiel: Mit seiner Hilfe können wir die Ähnlichkeiten zwischen zwei hochdimensionalen Merkmalsvektoren im ursprünglichen Merkmalsraum berechnen.

Bevor wir uns näher mit dem Kernel-Trick befassen, um diesem rechenintensiven Problem zu begegnen, rekapitulieren wir kurz noch einmal den normalen PCA-Ansatz, den wir am Anfang des Kapitels implementiert haben. Wir haben die Kovarianz zweier Merkmale  $k$  und  $j$  folgendermaßen berechnet:

$$\sigma_{jk} = \frac{1}{n} \sum_{i=1}^n (x_j^{(i)} - \mu_j)(x_k^{(i)} - \mu_k)$$

Da die Standardisierung der Merkmale diese beim Mittelwert null zentriert, wie z.B.  $\mu_j = 0$  und  $\mu_k = 0$ , können wir die Gleichung folgendermaßen vereinfachen:

$$\sigma_{jk} = \frac{1}{n} \sum_{i=1}^n x_j^{(i)} x_k^{(i)}$$

Beachten Sie, dass sich diese Gleichung auf die Kovarianz zweier Merkmale bezieht. Die allgemeine Gleichung zur Berechnung der Kovarianzmatrix  $\Sigma$  lautet:

$$\Sigma = \frac{1}{n} \sum_{i=1}^n \mathbf{x}^{(i)} \mathbf{x}^{(i)T}$$

Bernhard Scholkopf hat diesen Ansatz verallgemeinert (B. Scholkopf, A. Smola und K.-R. Müller, *Kernel Principal Component Analysis*, Seiten 583-588, 1997), sodass wir das Skalarprodukt der Objekte aus dem ursprünglichen Merkmalsraum mittels  $\phi$  durch die nichtlinearen Merkmalskombinationen ersetzen können:

$$\Sigma = \frac{1}{n} \sum_{i=1}^n \phi(\mathbf{x}^{(i)}) \phi(\mathbf{x}^{(i)})^T$$

Um aus dieser Kovarianzmatrix die Eigenvektoren (die Hauptkomponenten) zu erhalten, müssen wir folgende Gleichung lösen:

$$\Sigma v = \lambda v$$

$$\begin{aligned} &\Rightarrow \frac{1}{n} \sum_{i=1}^n \phi(x^{(i)}) \phi(x^{(i)})^T v = \lambda v \\ &\Rightarrow v = \frac{1}{n\lambda} \sum_{i=1}^n \phi(x^{(i)}) \phi(x^{(i)})^T v = \frac{1}{n} \sum_{i=1}^n a^{(i)} \phi(x^{(i)}) \end{aligned}$$

Hier sind  $\lambda$  und  $v$  Eigenwerte und Eigenvektoren der Kovarianzmatrix  $\Sigma$ .  $a$  erhalten wir, wie Sie im Folgenden sehen werden, durch Extraktion des Eigenvektors der Kernel-Matrix (Ähnlichkeitsmatrix)  $K$ .

Die Herleitung der Kernel-Matrix gelingt folgendermaßen:

Zunächst verwenden wir für die Kovarianzmatrix die Matrixnotation, wobei  $\phi(X)$  eine  $n \times k$ -dimensionale Matrix ist:

$$\Sigma = \frac{1}{n} \sum_{i=1}^n \phi(x^{(i)}) \phi(x^{(i)})^T = \frac{1}{n} \phi(X)^T \phi(X)$$

Nun können wir die Eigenvektorgleichung so formulieren:

$$v = \frac{1}{n} \sum_{i=1}^n a^{(i)} \phi(x^{(i)}) = \lambda \phi(X)^T a$$

Da  $\Sigma v = \lambda v$  gilt, erhalten wir:

$$\frac{1}{n} \phi(X)^T \phi(X) \phi(X)^T a = \lambda \phi(X)^T a$$

Nun multiplizieren wir beide Seiten der Gleichung mit  $\phi(X)$  und es ergibt sich:

$$\frac{1}{n} \phi(X) \phi(X)^T \phi(X) \phi(X)^T a = \lambda \phi(X) \phi(X)^T a$$

$$\Rightarrow \frac{1}{n} \phi(X) \phi(X)^T a = \lambda a$$

$$\Rightarrow \frac{1}{n} K a = \lambda a$$

Hier ist  $K$  die Ähnlichkeitsmatrix (Kernel-Matrix):

$$K = \phi(X) \phi(X)^T$$

Wie wir aus dem Abschnitt über SVM in Kapitel 3 wissen, können wir den Kernel-Trick anwenden, um uns die Berechnung der paarweisen Skalarprodukte der Objekte  $\mathbf{x}$  bei  $\phi$  zu ersparen, indem wir eine Kernel-Funktion  $k$  nutzen, sodass wir die Eigenvektoren gar nicht explizit zu berechnen brauchen:

$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \phi(\mathbf{x}^{(i)})^T \phi(\mathbf{x}^{(j)})$$

Mit anderen Worten: Was wir nach der Kernel-PCA erhalten, sind die bereits auf die jeweiligen Komponenten projizierten Objekte – und damit erübrigt sich die Konstruktion einer Transformationsmatrix wie beim normalen PCA-Ansatz. Die Kernel-Funktion (oder einfach nur der *Kernel*) kann im Wesentlichen als eine Funktion aufgefasst werden, die das Skalarprodukt zweier Vektoren berechnet – ein Ähnlichkeitsmaß.

Die gebräuchlichsten Kernel sind die folgenden:

- Der polynomiale Kernel:

$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = (\mathbf{x}^{(i)T} \mathbf{x}^{(j)} + \theta)^p$$

Hier sind  $\theta$  und  $p$  Schwellenwert und Potenz, die vom User festgelegt werden müssen.

- Der Sigmoid-Kernel (Tangens-hyperbolicus-Kernel):

$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \tanh(\eta \mathbf{x}^{(i)T} \mathbf{x}^{(j)} + \theta)$$

- Der RBF- (Radiale Basisfunktion) oder *Gaußsche Kernel*, den wir in den nachfolgenden Beispielen verwenden werden:

$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\frac{\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2}{2\sigma^2}\right)$$

Folgende Schreibweise, mit der Variablen  $\gamma = \frac{1}{2\sigma}$  ist ebenfalls gebräuchlich:

$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\gamma \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2\right)$$

Wir können das Ganze zusammenfassen und die folgenden drei Schritte zur Implementierung einer RBF-Kernel-PCA definieren:

1. Wir berechnen die Kernel-Matrix (Ähnlichkeitsmatrix)  $k$  wie folgt:

$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\gamma \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2\right)$$

Dies wird für alle Wertepaarungen durchgeführt:

$$\mathbf{K} = \begin{bmatrix} \kappa(\mathbf{x}^{(1)}, \mathbf{x}^{(1)}) & \kappa(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}) & \dots & \kappa(\mathbf{x}^{(1)}, \mathbf{x}^{(n)}) \\ \kappa(\mathbf{x}^{(2)}, \mathbf{x}^{(1)}) & \kappa(\mathbf{x}^{(2)}, \mathbf{x}^{(2)}) & \dots & \kappa(\mathbf{x}^{(2)}, \mathbf{x}^{(n)}) \\ \vdots & \vdots & \ddots & \vdots \\ \kappa(\mathbf{x}^{(n)}, \mathbf{x}^{(1)}) & \kappa(\mathbf{x}^{(n)}, \mathbf{x}^{(2)}) & \dots & \kappa(\mathbf{x}^{(n)}, \mathbf{x}^{(n)}) \end{bmatrix}$$

Wenn unsere Datensammlung 100 Trainingsdatenmengen enthielte, wäre die symmetrische Kernel-Matrix der paarweisen Ähnlichkeiten  $100 \times 100$ -dimensional.

2. Wir zentrieren die Kernel-Matrix  $k$  unter Verwendung der folgenden Gleichung:

$$\mathbf{K}' = \mathbf{K} - \mathbf{1}_n \mathbf{K} \mathbf{1}_n + \mathbf{1}_n \mathbf{K} \mathbf{1}_n$$

Hier ist  $\mathbf{1}_n$  eine  $n \times n$ -dimensionale Matrix (dieselbe Dimension wie die Kernel-Matrix), deren Werte alle gleich  $\frac{1}{n}$  sind.

3. Wir ermitteln anhand der zugehörigen Eigenwerte die  $k$  wichtigsten Eigenvektoren der zentrierten Kernel-Matrix, die in absteigender Reihenfolge nach ihrer Größe sortiert werden. Im Gegensatz zur normalen PCA sind die Eigenvektoren nicht die Achsen der Hauptkomponenten, sondern die auf diese Achsen projizierten Objekte der Datensammlung.

Sie werden sich an dieser Stelle vielleicht fragen, warum wir die Kernel-Matrix im zweiten Schritt zentrieren müssen. Als wir die Kovarianzmatrix notiert und die Skalarprodukte mittels  $\phi$  durch die nichtlinearen Merkmalskombinationen ersetzt haben, sind wir davon ausgegangen, dass wir standardisierte Daten verwenden, bei denen alle Merkmale einen Mittelwert von null besitzen. Die Zentrierung der Kernel-Matrix ist also erforderlich, weil wir den neuen Merkmalsraum nicht explizit neu berechnen und somit nicht garantieren können, dass er ebenfalls auf null zentriert ist.

Im nächsten Abschnitt werden wir diese drei Schritte in die Tat umsetzen und eine Kernel-PCA in Python implementieren.

### 5.3.2 Implementierung einer Kernel-Hauptkomponentenanalyse in Python

Im vorangegangenen Abschnitt haben wir die Kernkonzepte einer Kernel-PCA erörtert. Nun werden wir in Python eine RBF-Kernel-PCA anhand der drei Schritte implementieren, die diesen Ansatz zusammenfassen. Dank der Hilfsfunktionen von SciPy und NumPy erweist sich das tatsächlich als ganz einfach:

```
from scipy.spatial.distance import pdist, squareform
from scipy import exp
```

```
from scipy.linalg import eigh
import numpy as np

def rbf_kernel_pca(X, gamma, n_components):
    """
    Implementierung der RBF-Kernel-PCA

    Parameter
    -----
    X: {NumPy ndarray}, shape = [n_samples, n_features]

    gamma: float
        Parameter zum Abstimmen des RBF-Kernels

    n_components: int
        Anzahl der zurückgelieferten Hauptkomponenten

    Rückgabewert
    -----
    X_pc: {NumPy ndarray}, shape = [n_samples, k_features]
        Projizierte Datenmenge
    """

    # Paarweise quadratische euklidische Abstände in der
    # MxN-dimensionalen Datenmenge berechnen
    sq_dists = pdist(X, 'sqeuclidean')

    # Abstände paarweise in quadratische Matrix umwandeln
    mat_sq_dists = squareform(sq_dists)

    # Symmetrische Kernel-Matrix berechnen
    K = exp(-gamma * mat_sq_dists)

    # Kernel-Matrix zentrieren
    N = K.shape[0]
    one_n = np.ones((N,N)) / N
    K = K - one_n.dot(K) - K.dot(one_n)
                + one_n.dot(K).dot(one_n)

    # Paare aus Eigenwerten und Eigenvektoren der
    # zentrierten Kernel-Matrix ermitteln; numpy.eigh gibt
    # diese in aufsteigender Reihenfolge sortiert zurück
    eigvals, eigvecs = eigh(K)

    # Die k wichtigsten Eigenvektoren ermitteln
    X_pc = np.column_stack((eigvecs[:, -i]
                            for i in range(1, n_components + 1)))

    return X_pc
```

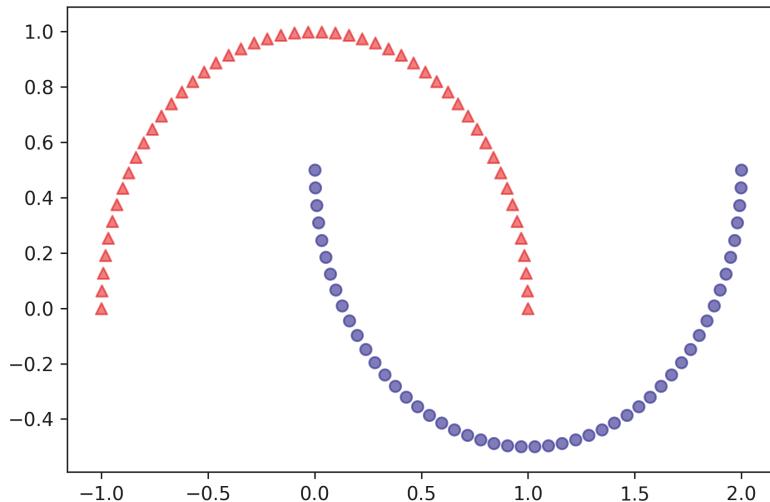
Der Nachteil der Dimensionsreduktion durch eine RBF-Kernel-PCA besteht darin, dass der Parameter  $\gamma$  a priori festgelegt werden muss. Um einen geeigneten Wert für  $\gamma$  zu finden, muss man herumexperimentieren. Am besten verwendet man hierfür Algorithmen zur Parameterabstimmung, die wir in Kapitel 6 eingehender betrachten werden, beispielsweise eine *Rastersuche (Grid Search)*.

### Beispiel 1: Separieren von Halbmondformen

Nun werden wir die `rbf_kernel_pca` auf einige nichtlineare Beispieldatenmengen anwenden. Zunächst erstellen wir eine zweidimensionale Datenmenge mit 100 Datenbeispielen, die zwei Halbmondformen repräsentieren:

```
>>> from sklearn.datasets import make_moons
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> plt.scatter(X[y==0, 0], X[y==0, 1],
...                 color='red', marker='^', alpha=0.5)
>>> plt.scatter(X[y==1, 0], X[y==1, 1],
...                 color='blue', marker='o', alpha=0.5)
>>> plt.show()
```

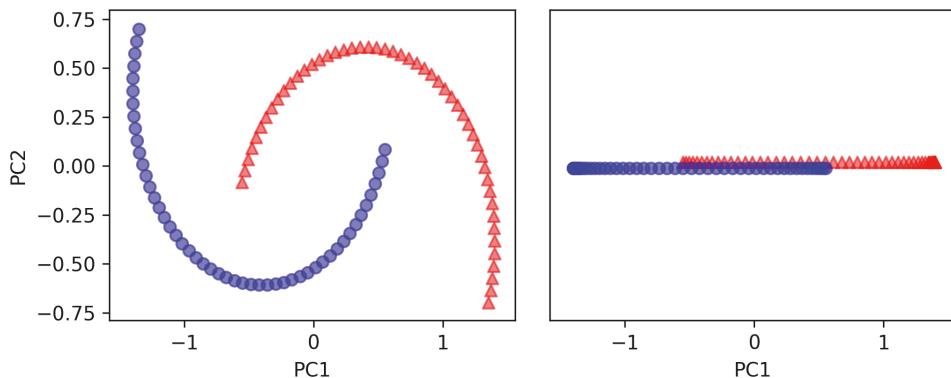
Der Halbmond aus dreieckigen Symbolen soll hier die eine Klasse darstellen, während der Halbmond aus Kreisen eine zweite Klasse repräsentiert.



Diese beiden Kurven sind offensichtlich nicht linear trennbar und unser Ziel ist es, die beiden Halbmonde via Kernel-PCA »auseinanderzufalten«, damit die Datenmenge als Eingabe für einen linearen Klassifizierer geeignet ist. Aber zunächst betrachten wir die Datenmenge, nachdem sie mit einer normalen PCA auf die Hauptkomponenten projiziert wurde:

```
>>> from sklearn.decomposition import PCA
>>> scikit_pca = PCA(n_components=2)
>>> X_spca = scikit_pca.fit_transform(X)
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
...                 color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
...                 color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_spca[y==0, 0], np.zeros((50,1))+0.02,
...                 color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_spca[y==1, 0], np.zeros((50,1))-0.02,
...                 color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.show()
```

Anhand der Abbildung wird deutlich, dass ein linearer Klassifizierer mit der per Standard-PCA transformierten Datenmenge nicht gut funktionieren würde.



Beachten Sie, dass die dreieckigen und kreisförmigen Symbole in dem Diagramm auf der rechten Seite der Abbildung, in dem nur die erste Hauptkomponente dargestellt ist, zur besseren Veranschaulichung der Klassenüberlappung leicht nach oben bzw. unten verschoben sind. Der linke Teil der Abbildung zeigt, dass die beiden ursprünglichen Halbmondformen nur geringfügig zum vertikalen Schwerpunkt verschoben werden – diese Transformation würde es einem linearen Klassifizierer nicht erleichtern, Kreise und Dreiecke zu unterscheiden. Und im rechten Teil der Abbildung ist erkennbar, dass die Kreise und Dreiecke der Halbmondformen nach einer Projektion der Datenmenge auf eine eindimensionale Merkmalsachse ebenfalls nicht linear trennbar sind.

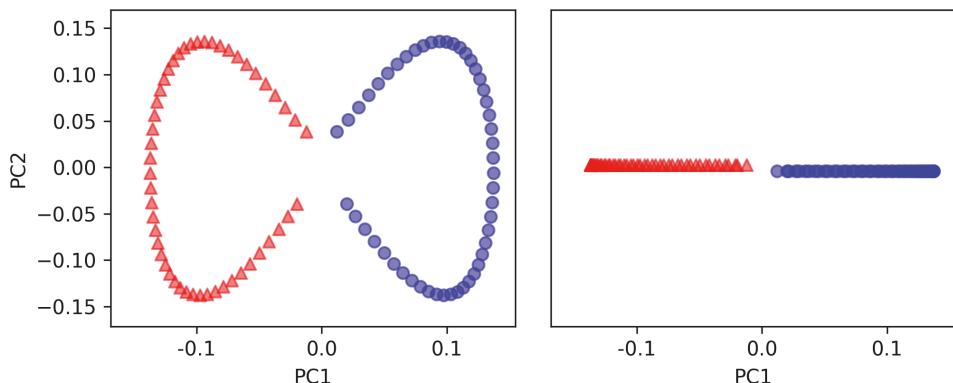
**Tipp**

Bitte denken Sie daran, dass die PCA ein unüberwachtes Verfahren ist, das im Gegensatz zur LDA keinen Gebrauch von den Informationen über die Klassenbezeichnungen macht, um die Varianz zu maximieren. Die dreieckigen und kreisförmigen Symbole wurden hier nur zu Illustrationszwecken hinzugefügt, um das Ausmaß der Trennung darstellen zu können.

Nun probieren wir die im vorangegangenen Abschnitt implementierte Kernel-PCA-Funktion `rbf_kernel_pca` aus:

```
>>> X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
...                 color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
...                 color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==0, 0], np.zeros((50,1))+0.02,
...                 color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==1, 0], np.zeros((50,1))-0.02,
...                 color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.show()
```

Hier ist erkennbar, dass die beiden Klassen (Dreiecke und Kreise) linear gut voneinander getrennt sind und somit eine geeignete Trainingsdatenmenge für lineare Klassifizierer darstellen.



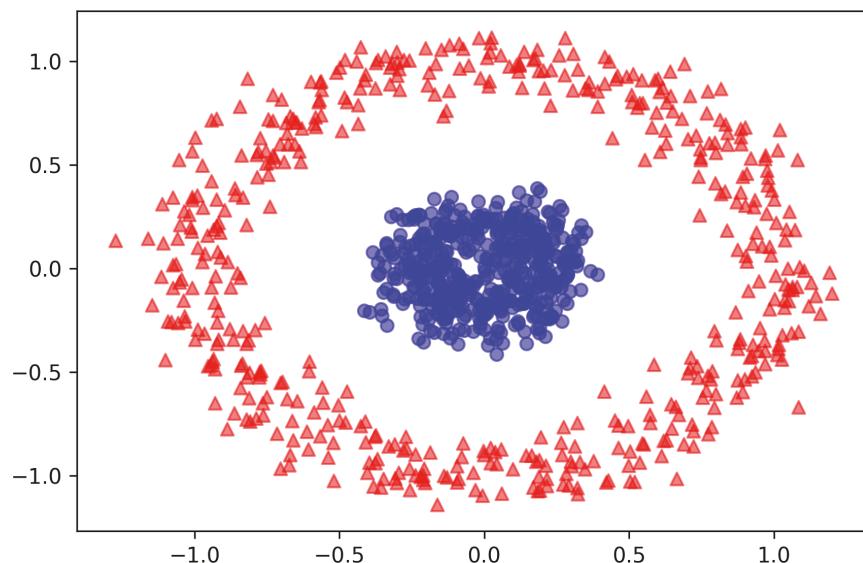
Leider gibt es keinen universellen Wert für den Abstimmungsparameter  $\gamma$ , der für verschiedene Datenmengen geeignet wäre. Um einen für eine gegebene Aufgabenstellung passenden  $\gamma$ -Wert zu finden, muss man experimentieren. In Kapitel 6 werden wir Verfahren erörtern, die uns dabei helfen können, die Optimierung solcher Abstimmungsparameter zu automatisieren. Hier verwende ich  $\gamma$ -Werte, die meiner Erfahrung nach gute Ergebnisse erzielen.

## Beispiel 2: Separieren konzentrischer Kreise

Im vorangegangenen Abschnitt haben Sie erfahren, wie Sie Halbmondformen via Kernel-PCA trennen können. Nachdem wir so großen Aufwand betrieben haben, um das Konzept der Kernel-PCA zu verstehen, sollten wir noch ein weiteres interessantes Beispiel für eine nichtlineare Aufgabenstellung betrachten, nämlich konzentrische Kreise:

```
>>> from sklearn.datasets import make_circles
>>> X, y = make_circles(n_samples=1000,
...                      random_state=123, noise=0.1, factor=0.2)
>>> plt.scatter(X[y==0, 0], X[y==0, 1],
...               color='red', marker='^', alpha=0.5)
>>> plt.scatter(X[y==1, 0], X[y==1, 1],
...               color='blue', marker='o', alpha=0.5)
>>> plt.show()
```

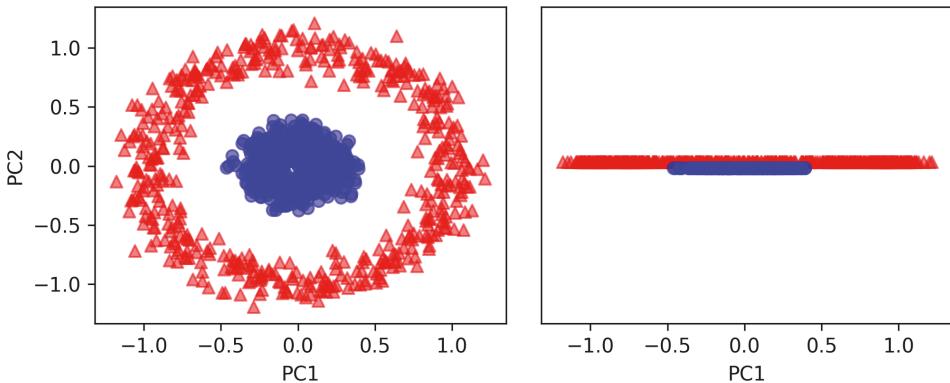
Wir gehen wieder davon aus, dass es sich um zwei Klassen handelt, die durch Dreiecke bzw. Kreise repräsentiert werden:



Zunächst probieren wir wieder den normalen PCA-Ansatz aus, um ihn mit dem Ergebnis der RBF-Kernel-PCA vergleichen zu können:

```
>>> scikit_pca = PCA(n_components=2)
>>> X_spca = scikit_pca.fit_transform(X)
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
...                 color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
...                 color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_spca[y==0, 0], np.zeros((500,1))+0.02,
...                 color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_spca[y==1, 0], np.zeros((500,1))-0.02,
...                 color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.show()
```

Erneut stellen wir fest, dass die Standard-PCA nicht in der Lage ist, ein Ergebnis zu liefern, das zum Trainieren eines linearen Klassifizierers geeignet wäre:



Nun probieren wir mit einem passenden  $\gamma$ -Wert aus, ob uns mit der Implementierung der RBF-Kernel-PCA mehr Erfolg beschieden ist:

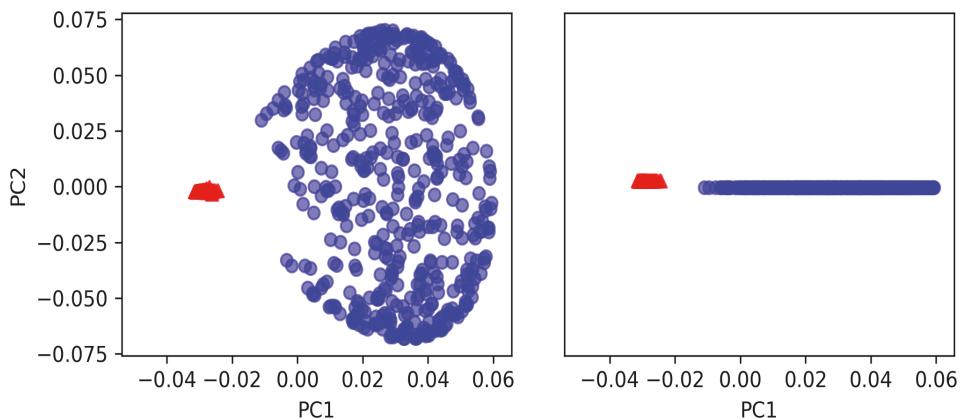
```
>>> X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
...                 color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
```

```

...
    color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==0, 0], np.zeros((500,1))+0.02,
...
...                 color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==1, 0], np.zeros((500,1))-0.02,
...
...                 color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.show()

```

Auch dieses Mal projiziert die RBF-Kernel-PCA die Daten auf einen neuen Unterraum, in dem sich die beiden Klassen linear trennen lassen:



### 5.3.3 Projizieren neuer Datenpunkte

In den beiden vorangegangenen Anwendungsbeispielen für eine Kernel-PCA, sprich bei den Halbmondformen und den konzentrischen Kreisen, haben wir jeweils eine einzelne Datenmenge auf ein neues Merkmal projiziert. In echten Anwendungsfällen kann es allerdings mehr als nur eine zu transformierende Datenmenge geben, beispielsweise Trainings- und Testdaten, und typischerweise kommen bei der Entwicklung des Modells und dessen Bewertung weitere Datenpunkte hinzu. In diesem Abschnitt erfahren Sie, wie Sie Datenpunkte projizieren, die nicht Teil der Trainingsdatenmenge waren.

Wie wir vom normalen PCA-Ansatz vom Anfang des Kapitels wissen, projizieren wir die Daten, indem wir das Skalarprodukt einer Transformationsmatrix und der Eingabedaten berechnen. Die Spalten der Transformationsmatrix sind die  $k$  wichtigsten Eigenvektoren ( $v$ ), die wir anhand der Kovarianzmatrix erhalten haben.

Nun stellt sich die Frage, wie wir dieses Konzept auf die Kernel-PCA übertragen können. Bei der Kernel-PCA berechnen wir einen Eigenvektor ( $\mathbf{a}$ ) der zentrierten Kernel-Matrix (nicht der Kovarianzmatrix), was bedeutet, dass es sich um die bereits auf die Hauptkomponentenachse  $\mathbf{v}$  projizierten Daten handelt. Wenn wir nun ein neues Objekt  $\mathbf{x}'$  auf diese Hauptkomponentenachse projizieren möchten, müssen wir Folgendes berechnen:

$$\phi(\mathbf{x}')^T \mathbf{v}$$

Glücklicherweise können wir hier den Kernel-Trick anwenden und müssen die Projektion  $\phi(\mathbf{x}')^T \mathbf{v}$  nicht explizit berechnen. Es ist außerdem anzumerken, dass es sich bei der Kernel-PCA im Gegensatz zur normalen PCA um ein speicherbares Verfahren handelt, was bedeutet, dass wir die ursprüngliche Trainingsdatenmenge bei jeder Projektion neuer Datenpunkte wiederverwenden müssen. Wir müssen den RBF-Kernel bzw. die Ähnlichkeit zwischen jedem  $i$ -ten Datenbeispiel in der Trainingsdatenmenge und dem neuen Objekt  $\mathbf{x}'$  paarweise berechnen:

$$\begin{aligned} \phi(\mathbf{x}')^T \mathbf{v} &= \sum_i a^{(i)} \phi(\mathbf{x}')^T \phi(\mathbf{x}^{(i)}) \\ &= \sum_i a^{(i)} k(\mathbf{x}', \mathbf{x}^{(i)}) \end{aligned}$$

Hier erfüllen die Eigenvektoren  $\mathbf{a}$  und die Eigenwerte  $\lambda$  der Kernel-Matrix  $\mathbf{K}$  die folgende Bedingung:

$$\mathbf{K}\mathbf{a} = \lambda\mathbf{a}$$

Nach der Berechnung der Ähnlichkeit zwischen dem neuen Objekt und den Objekten in der Trainingsdatenmenge müssen wir den Eigenvektor  $\mathbf{a}$  mit seinem Eigenwert normieren. Daher modifizieren wir die bereits implementierte `rbf_kernel_pca`-Funktion, damit sie auch die Eigenwerte der Kernel-Matrix zurückgibt:

```
from scipy.spatial.distance import pdist, squareform
from scipy import exp
from scipy.linalg import eigh
import numpy as np

def rbf_kernel_pca(X, gamma, n_components):
    """
    Implementierung der RBF-Kernel-PCA

    Parameter
    -----
    
```

```
X: {NumPy ndarray}, shape = [n_samples, n_features]

gamma: float
    Parameter zum Abstimmen des RBF-Kernels

n_components: int
    Anzahl der zurückgelieferten Hauptkomponenten

Rückgabewerte
-----
alphas: {NumPy ndarray}, shape = [n_samples, k_features]
    Projizierte Datenmenge

lambdas: list
    Eigenwerte
    ....
# Paarweise quadratische euklidische Abstände in der
# MxN-dimensionalen Datenmenge berechnen
sq_dists = pdist(X, 'squared')

# Abstände paarweise in quadratische Matrix umwandeln
mat_sq_dists = squareform(sq_dists)

# Symmetrische Kernel-Matrix berechnen
K = exp(-gamma * mat_sq_dists)

# Kernel-Matrix zentrieren
N = K.shape[0]
one_n = np.ones((N,N)) / N
K = K - one_n.dot(K) - K.dot(one_n)
        + one_n.dot(K).dot(one_n)

# Paare aus Eigenwerten und Eigenvektoren der
# zentrierten Kernel-Matrix ermitteln; numpy.eigh gibt
# diese in aufsteigender Reihenfolge sortiert zurück
eigvals, eigvecs = eigh(K)

# Die k wichtigsten Eigenvektoren ermitteln
alphas = np.column_stack((eigvecs[:, -i]
                           for i in range(1, n_components)))

# Korrespondierende Eigenwerte ermitteln
lambdas = [eigvals[-i] for i in range(1, n_components)]

return alphas, lambdas
```

Als Nächstes erstellen wir eine Halbmond-Datenmenge und projizieren sie mit der aktualisierten Implementierung der RBF-Kernel-PCA auf einen eindimensionalen Unterraum:

```
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> alphas, lambdas =
...                 rbf_kernel_pca(X, gamma=15, n_components=1)
```

Um den Code für die Projektion neuer Datenpunkte zu überprüfen, nehmen wir an, dass es sich beim 26. Punkt der Halbmond-Datenmenge um einen neuen Datenpunkt  $x'$  handelt und dass wir ihn auf diesen neuen Unterraum projizieren möchten:

```
>>> x_new = X[25]
>>> x_new
array([ 1.8713187 ,  0.00928245])
>>> x_proj = alphas[25] # ursprüngliche Projektion
>>> x_proj
array([ 0.07877284])
>>> def project_x(x_new, X, gamma, alphas, lambdas):
...     pair_dist = np.array([np.sum(
...         (x_new-row)**2) for row in X])
...     k = np.exp(-gamma * pair_dist)
...     return k.dot(alphas / lambdas)
```

Durch die Ausführung des folgenden Codes können wir die ursprüngliche Projektion reproduzieren. Mit der `project_x`-Funktion können wir außerdem neue Datenpunkte projizieren. Hier der Code:

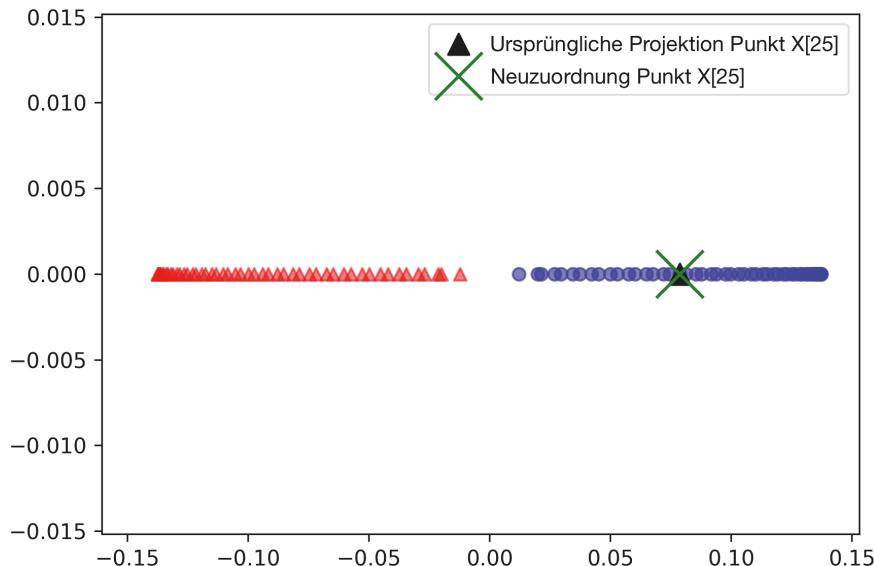
```
>>> x_reproj = project_x(x_new, X,
...                         gamma=15, alphas=alphas, lambdas=lambdas)
>>> x_reproj
array([ 0.07877284])
```

Zum Abschluss visualisieren wir die Projektion auf die erste Hauptkomponente:

```
>>> plt.scatter(alphas[y==0, 0], np.zeros((50)),
...               color='red', marker='^', alpha=0.5)
>>> plt.scatter(alphas[y==1, 0], np.zeros((50)),
...               color='blue', marker='o', alpha=0.5)
>>> plt.scatter(x_proj, 0, color='black',
...               label='Ursprüngliche Projektion Punkt X[25]',
...               marker='^', s=100)
>>> plt.scatter(x_reproj, 0, color='green',
...               label='Neuzuordnung Punkt X[25]',
```

```
... marker='x', s=500)
>>> plt.legend(scatterpoints=1)
>>> plt.show()
```

Dem Streudiagramm ist zu entnehmen, dass wir das Objekt  $x'$  korrekt auf die erste Hauptkomponente abgebildet haben:



### 5.3.4 Kernel-Hauptkomponentenanalyse mit scikit-learn

Im Submodul `sklearn.decomposition` stellt scikit-learn komfortablerweise die Klasse `KernelPCA` zur Verfügung, die wie die normale PCA-Klasse verwendet wird. Der Kernel-Typ wird über den Parameter `kernel` festgelegt:

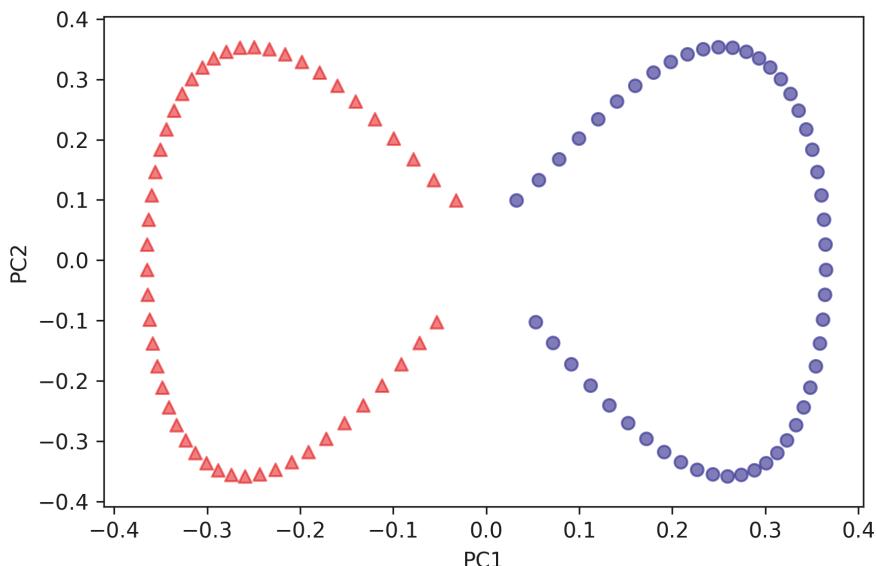
```
>>> from sklearn.decomposition import KernelPCA
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> scikit_kpca = KernelPCA(n_components=2,
...                           kernel='rbf', gamma=15)
>>> X_skernpca = scikit_kpca.fit_transform(X)
```

Um zu überprüfen, ob wir mit unserer eigenen Kernel-PCA-Implementierung übereinstimmende Ergebnisse erhalten, geben wir die transformierten Daten der Halbmondformen auf den beiden ersten Hauptkomponenten aus:

```
>>> plt.scatter(X_skernpca[y==0, 0], X_skernpca[y==0, 1],
...                 color='red', marker='^', alpha=0.5)
>>> plt.scatter(X_skernpca[y==1, 0], X_skernpca[y==1, 1],
...                 color='blue', marker='o', alpha=0.5)
```

```
>>> plt.xlabel('PC1')
>>> plt.ylabel('PC2')
>>> plt.show()
```

Wie Sie sehen, steht das Resultat der scikit-learn-KernelPCA mit dem unserer eigenen Implementierung im Einklang:



### Tipp

Scikit-learn implementiert außerdem einige erweiterte Verfahren zur Dimensionsreduktion, die über den Rahmen dieses Buches hinausgehen. Einen Überblick über die aktuellen Implementierungen mit anschaulichen Beispielen finden Sie unter <http://scikit-learn.org/stable/modules/manifold.html>.

## 5.4 Zusammenfassung

In diesem Kapitel haben Sie drei verschiedene grundlegende Dimensionsreduktionsverfahren zur Merkmalsextraktion kennengelernt: Standard-PCA, LDA und Kernel-PCA. Bei der Standard-PCA projizieren wir die Daten auf einen niedrigerdimensionalen Unterraum, um die Varianz entlang der orthogonalen Merkmalsachsen zu maximieren, und ignorieren dabei die Klassenbezeichnungen. Im Gegensatz zur PCA handelt es sich bei der LDA um ein Verfahren zur überwachten Dimensionsreduktion, was bedeutet, dass bei dem Versuch, die Separierbarkeit der Klassen in einem linearen Merkmalsraum zu maximieren, die Klasseninformationen in der Trainingsdatenmenge berücksichtigt werden.

Darüber hinaus haben Sie die Kernel-PCA kennengelernt, die es mithilfe des Kernel-Tricks und einer temporären Projektion auf einen höherdimensionalen Merkmalsraum gestattet, eine aus nichtlinearen Merkmalen bestehende Datenmenge auf einen niedrigerdimensionalen Unterraum abzubilden, in dem die Klassen linear trennbar sind.

Mit dem Wissen über diese Techniken zur Vorverarbeitung sind Sie nun gut dafür gerüstet, die bewährten Verfahren zur effizienten Vorverarbeitung und zur Bewertung verschiedener Modelle kennenzulernen, um die es im nächsten Kapitel geht.



# Best Practices zur Modellbewertung und Hyperparameter-Abstimmung

In den vorangegangenen Kapiteln haben Sie die grundlegenden Lernalgorithmen für die Klassifizierung kennengelernt und erfahren, wie Sie die Daten aufbereiten können, bevor die Algorithmen damit gefüttert werden. Nun ist es an der Zeit, sich mit den bewährten Verfahren zur Entwicklung guter Lernmodelle durch die Feinabstimmung der Algorithmen und die Bewertung der Leistung zu befassen.

Die Themen in diesem Kapitel sind:

- Verfahrensfehlerfreie Bewertung der Leistung eines Modells
- Typische bei Lernalgorithmen auftretende Schwierigkeiten
- Feinabstimmung von Lernmodellen
- Bewertung von Vorhersagemodellen anhand verschiedener Leistungskriterien

## 6.1 Arbeitsabläufe mit Pipelines optimieren

Bei der Anwendung verschiedener Verfahren zur Datenvorverarbeitung, wie etwa der in Kapitel 4 vorgestellten *Merkmalsstandardisierung* oder der *Hauptkomponentenanalyse* zwecks Datenkomprimierung aus Kapitel 5, haben Sie festgestellt, dass wir die bei der Anpassung an die Trainingsdaten erhaltenen Parameter wiederverwenden müssen, um neue Daten zu skalieren und zu komprimieren, beispielsweise die Objekte in der Testdatenmenge. In diesem Abschnitt lernen Sie ein äußerst praktisches Tool kennen, nämlich scikit-learns `Pipeline`-Klasse. Sie ermöglicht es, ein Modell inklusive einer beliebigen Anzahl von Transformationen an die Daten anzupassen und dann anhand neuer Daten Vorhersagen zu treffen.

### 6.1.1 Die Wisconsin-Brustkrebs-Datensammlung

In diesem Kapitel verwenden wir die *Wisconsin-Brustkrebs-Datensammlung*, die Daten zu 569 bösartigen und gutartigen Tumorzellen umfasst. Die ersten beiden Spalten der Datensammlung enthalten eine eindeutige ID der Objekte sowie die dazugehörige Diagnose M (=*malignant*, bösartig) bzw. B (=*benign*, gutartig). Die Spalten 3 bis 32 enthalten 30 anhand der digitalisierten Bilder der Zellkerne errechnete

reellwertige Merkmale, die für die Entwicklung eines Modells genutzt werden können, das prognostiziert, ob ein Tumor gut- oder bösartig ist. Die Wisconsin-Brustkrebs-Datensammlung ist Bestandteil des *UCI Machine Learning Repositorys*. Weitere Informationen hierzu finden Sie unter [https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+\(Diagnostic\)](https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)).

Als Erstes lesen wir nun die Datensammlung ein und teilen Sie in drei einfachen Schritten in Trainings- und Testdaten auf:

1. Zunächst rufen wir die Daten mit pandas direkt von der UCI-Website ab:

```
>>> import pandas as pd
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/'
...                   'machine-learning-databases'
...                   '/breast-cancer-wisconsin/wdbc.data',
...                   header=None)
```

2. Dann weisen wir die 30 Merkmale einem NumPy-Array namens X zu. Mit dem `LabelEncoder`-Objekt wandeln wir die ursprünglichen Klassenbezeichnungen (M und B) in Ganzzahlen um:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> X = df.loc[:, 2: ].values
>>> y = df.loc[:, 1].values
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
>>> le.classes_
array(['B', 'M'], dtype=object)
```

Nach der Zuweisung der Klassenbezeichnungen (bzw. der Diagnosen) zum Array y werden bösartige Tumoren als Klasse 1 und gutartige als Klasse 0 repräsentiert, wie man beim Aufruf der `transform`-Methode von `LabelEncoder` mit zwei Testklassenbezeichnungen sieht:

```
>>> le.transform(['M', 'B'])
array([1, 0])
```

3. Bevor wir im nächsten Abschnitt die erste Modell-Pipeline konstruieren, teilen wir die Datensammlung in eine Trainings- und eine Testdatenmenge auf (80 bzw. 20 Prozent der Daten):

```
>>> from sklearn.cross_validation import train_test_split
>>> X_train, X_test, y_train, y_test =
...     train_test_split(X, y, test_size=0.20, random_state=1)
```

## 6.1.2 Transformer und Schätzer in einer Pipeline kombinieren

Im vorangegangenen Kapitel haben Sie erfahren, dass viele Lernalgorithmen Eingabemerkmale gleicher Größenordnung benötigen, um optimal zu funktionieren. Wir müssen daher die Spalten der Wisconsin-Brustkrebs-Datensammlung standardisieren, bevor wir sie einem linearen Klassifizierer wie einer logistischen Regression übergeben. Darüber hinaus sollen die Daten der ursprünglich 30 Dimensionen auf einen zweidimensionalen Unterraum abgebildet werden, und zwar mittels *Hauptkomponentenanalyse (PCA, Principal Component Analysis)* – einem Merkmalsextraktionsverfahren zur Dimensionsreduktion, das in Kapitel 5 eingeführt wurde.

Anstatt die Schritte zur Anpassung und Transformation der Trainings- und Testdatenmengen getrennt auszuführen, verknüpfen wir die `StandardScaler`-, `PCA`- und `LogisticRegression`-Objekte in einer Pipeline:

```
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.decomposition import PCA
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.pipeline import make_pipeline
>>> pipe_lr = make_pipeline(StandardScaler(),
...                         PCA(n_components=2),
...                         LogisticRegression(random_state=1))
>>> pipe_lr.fit(X_train, y_train)
>>> y_pred = pipe_lr.predict(X_test)
>>> print('Korrektklassifizierungsrate Test: %.3f' %
...                   pipe_lr.score(X_test, y_test))
Korrektklassifizierungsrate Test: 0.956
```

Die `make_pipeline`-Funktion nimmt eine beliebige Anzahl von scikit-learn-Transformern (Objekte, die `fit`- und `transform`-Methoden als Eingabe unterstützen), gefolgt von einem scikit-learn-Schätzer, der `fit`- und `predict`-Methoden implementiert, als Eingabe entgegen. In unserem Codebeispiel haben wir zwei Transformer (`StandardScaler` und `PCA`) und einen `LogisticRegression`-Schätzer als Eingabe für die `make_pipeline`-Funktion verwendet, die anhand dieser Objekte ein `Pipeline`-Objekt erstellt.

Man kann sich eine scikit-learn-Pipeline als einen Meta-Schätzer oder als einen Wrapper für die verschiedenen Transformer und Schätzer vorstellen. Wenn man die `fit`-Methode der Pipeline aufruft, werden die Daten durch Aufrufe der `fit`- und `transform`-Methoden als Zwischenschritte einer Reihe von Transformern übergeben, bis sie schließlich beim Schätzer-Objekt ankommen (dem letzten Element einer Pipeline), der dann an die transformierten Trainingsdaten angepasst wird.

In dem vorangegangenen Beispielcode haben wir eine Pipeline eingerichtet, die aus zwei Zwischenschritten besteht, einem `StandardScaler` und einem `PCA`-

Transformer, und als abschließender Schätzer folgt ein Klassifizierer einer logistischen Regression.

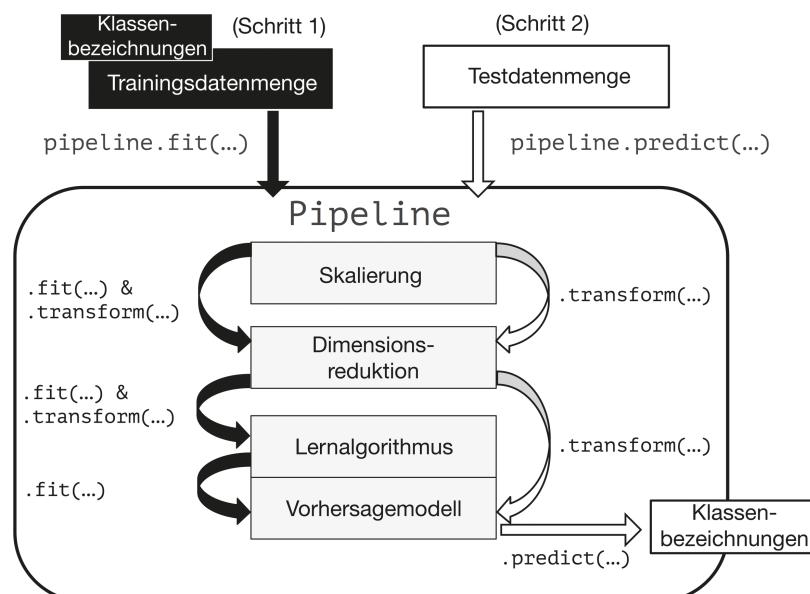
Wenn die `fit`-Methode der Pipeline `pipe_lr` aufgerufen wird, führt der `StandardScaler` die `fit`- und `transform`-Funktionen mit den Trainingsdaten aus und übergibt die transformierten Daten an das nächste Objekt der Pipeline, den `PCA`-Transformer. Wie beim vorhergehenden Schritt führt die `PCA` daraufhin die `fit`- und `transform`-Funktionen mit den skalierten Daten aus und übergibt das Ergebnis dem letzten Element der Pipeline, dem Schätzer.

Schließlich wird der `LogisticRegression`-Schätzer an die durch `StandardScaler` und `PCA` transformierten Daten angepasst. Beachten Sie, dass die Anzahl der Zwischenschritte in einer Pipeline nicht beschränkt ist, allerdings muss das letzte Element ein Schätzer sein.

Pipelines implementieren neben der `fit`- auch eine `predict`-Methode. Wenn man der `predict`-Methode einer Pipeline-Instanz eine Datenmenge übergibt, wird sie durch Aufrufe der `transform`-Methode an die verschiedenen Zwischenschritte übergeben. Der letzte Schritt besteht schließlich darin, dass ein Schätzer-Objekt eine Vorhersage anhand der transformierten Daten zurückliefert.

Pipelines sind ein äußerst nützliches Wrapper-Tool, und wir werden sie im verbleibenden Teil des Buches häufig verwenden.

Vergewissern Sie sich, dass Sie die Funktionsweise einer Pipeline gründlich verstanden haben, und sehen Sie sich die folgende Abbildung genau an, in der die vorangegangenen Erläuterungen zusammengefasst sind.



## 6.2 Beurteilung des Modells durch k-fache Kreuzvalidierung

Einer der entscheidenden Schritte bei der Entwicklung eines Lernmodells ist die Abschätzung der Leistung bei unbekannten Daten. Nehmen wir an, wir passen unser Modell an eine Trainingsdatenmenge an und verwenden dieselben Daten, um zu prüfen, wie gut es in der Praxis funktioniert. Aus dem Abschnitt *Überanpassung durch Regularisierung verhindern* in Kapitel 3 wissen Sie, dass ein zu einfaches Modell zu einer Unteranpassung (großes Bias) führen kann bzw. dass es zu einer Überanpassung (hohe Varianz) kommt, wenn das Modell für die zugrunde liegenden Trainingsdaten zu komplex ist.

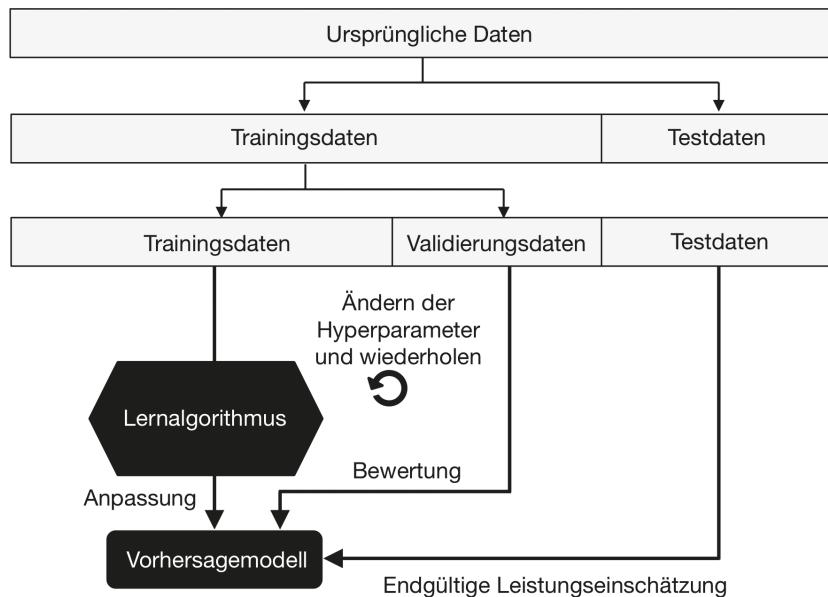
Um hier einen akzeptablen Bias-Varianz-Kompromiss zu finden, müssen wir das Modell sorgfältig bewerten. In diesem Abschnitt lernen Sie zwei nützliche Verfahren für verlässliche Abschätzungen des Verallgemeinerungsfehlers kennen, die Aufschluss darüber geben, wie gut das Modell mit unbekannten Daten funktioniert: die *2-fache* und die *k-fache Kreuzvalidierung*.

### 6.2.1 2-fache Kreuzvalidierung

Ein klassischer und verbreiteter Ansatz zur Einschätzung der Verallgemeinerungsfähigkeit eines Lernmodells ist die 2-fache Kreuzvalidierung, die auch als *Holdout-Methode* bezeichnet wird. Dabei wird die ursprüngliche Datensammlung in eine Trainings- und eine Testdatenmenge aufgeteilt. Erstere wird zum Trainieren des Modells verwendet, Letztere dient zur Bewertung der Leistung. Allerdings sind wir in typischen Anwendungsfällen auch an der Abstimmung und dem Vergleich verschiedener Parameter interessiert, um die Vorhersagekraft bei unbekannten Daten zu verbessern. Dieser Vorgang wird *Modellauswahl* genannt und beschreibt die Auswahl der optimalen Werte für die abstimmbaren Parameter (die sogenannten *Hyperparameter*) einer gegebenen Klassifizierungsaufgabe. Wenn wir bei der Modellauswahl jedoch immer wieder dieselben Testdaten verwenden, werden sie Teil der Trainingsdaten – und eine Überanpassung des Modells wird wahrscheinlicher. Trotz dieser Problematik wird hier häufig noch immer die Testdatenmenge benutzt, obwohl diese Vorgehensweise ungeeignet ist.

Eine bessere Möglichkeit, die 2-fache Kreuzvalidierung für die Modellauswahl einzusetzen, ist die Aufteilung der Daten in drei Teile: eine Trainings-, eine Validierungs- und eine Testdatenmenge. Die Trainingsdatenmenge wird zur Anpassung der verschiedenen Modelle eingesetzt – und die Leistung eines Modells unter Verwendung der Validierungsdaten wird dann zur Modellauswahl genutzt. Eine Testdatenmenge zu verwenden, die dem Modell während des Trainings und der Modellauswahl vorenthalten wird, hat den Vorteil, dass wir eine weniger verzerrte Einschätzung der Verallgemeinerungsfähigkeit bei neuen Daten erzielen. Die folgende Abbildung illustriert das Konzept der 2-fachen Kreuzvalidierung, bei der wir eine Validierungsdatenmenge verwenden, um nach dem Training mit verschiedenen Parameterwerten wiederholt die Leistung des Modells zu beurteilen.

Wenn wir mit der Abstimmung der Parameterwerte zufrieden sind, können wir den Verallgemeinerungsfehler des Modells mittels Anwendung auf die Testdaten abschätzen.



Ein Nachteil dieser Methode besteht darin, dass die Einschätzung der Leistung stark davon abhängen kann, wie die Trainingsdaten in die Untermengen Trainings- und Validierungsdaten aufgeteilt werden – die diesbezüglichen Ergebnisse werden bei unterschiedlichen Aufteilungen variieren. Im nächsten Abschnitt werden wir ein robusteres Verfahren zur Einschätzung der Leistung betrachten, bei dem wir die Kreuzvalidierung  $k$ -mal mit  $k$  Untermengen der Trainingsdaten wiederholen.

### 6.2.2 k-fache Kreuzvalidierung

Bei der  $k$ -fachen Kreuzvalidierung teilen wir die Trainingsdatenmenge in  $k$  Teilmengen (ohne Ersetzung) auf. Dann werden  $k - 1$  Teilmengen zum Trainieren des Modells benutzt und eine Teilmenge dient zum Testen. Diese Prozedur wird  $k$ -mal wiederholt, sodass wir  $k$  Modelle und  $k$  Einschätzungen der Leistung erhalten.

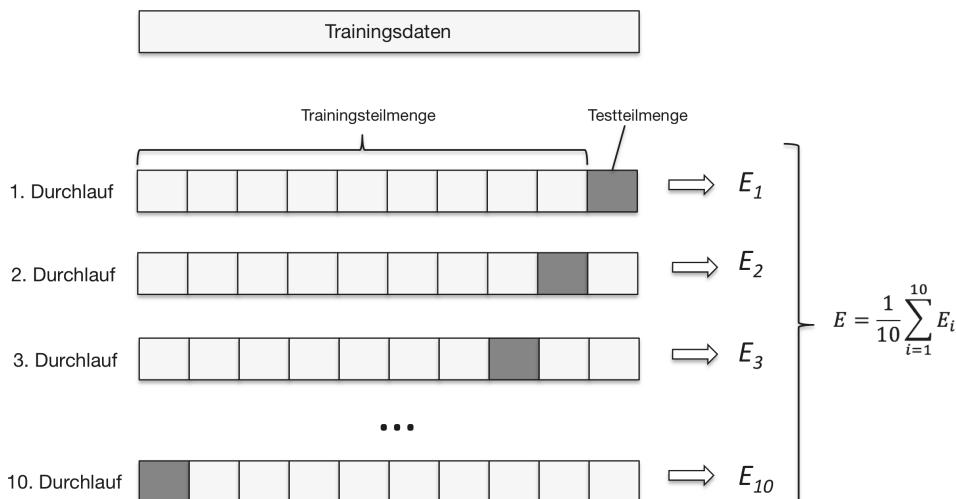
#### Tipp

In Abschnitt 3.6.3 (*Mehrere Entscheidungsbäume zu einem Random Forest kombinieren*) haben wir ein Beispiel für die Auswahl von Teilmengen *mit* und *ohne Ersetzung* kennengelernt. Wenn Sie das Kapitel noch nicht gelesen haben oder darüber nachlesen möchten, finden Sie dort einen Kasten mit Erläuterungen.

Die durchschnittliche Leistung des Modells berechnen wir dann anhand der verschiedenen unabhängigen Teilmengen und erhalten so eine Einschätzung, die im Vergleich zur 2-fachen Kreuzvalidierung weniger empfindlich auf die Aufteilung der Trainingsdaten reagiert. Die k-fache Kreuzvalidierung wird typischerweise bei der Modellabstimmung verwendet, um die optimalen Werte der Hyperparameter zu finden, die eine annehmbare Verallgemeinerungsfähigkeit liefern.

Sobald wir akzeptable Werte für die Hyperparameter gefunden haben, können wir das Modell erneut mit der gesamten Trainingsdatenmenge trainieren und erhalten mit der unabhängigen Testdatenmenge die endgültige Einschätzung der Leistung. Das Modell nach der k-fachen Kreuzvalidierung erneut an die gesamte Trainingsdatenmenge anzupassen, stellt dem Lernalgorithmus mehr Trainingsdaten bereit und führt im Allgemeinen zu einem genaueren und stabileren Modell.

Da es sich bei der k-fachen Kreuzvalidierung um eine wiederholte Entnahme ohne Ersetzung handelt, hat dieser Ansatz den Vorteil, dass alle Datenpunkte genau einmal Teil einer Trainings- und Testdatenmenge sind, was zu einer geringeren Abschätzung der Varianz der Modellleistung führt als bei der 2-fachen Kreuzvalidierung. In der nachstehenden Abbildung ist das Konzept der k-fachen Kreuzvalidierung für  $k = 10$  zusammengefasst. Die Trainingsdaten werden in 10 Teilmengen aufgeteilt. Während der 10 Durchläufe werden jeweils 9 Teilmengen für das Training eingesetzt und eine Teilmenge dient als Testdatenmenge zur Modellbewertung. Die Leistungsabschätzungen  $E_i$  (beispielsweise Korrektklassifizierungsrate oder Fehlerquote) der Teilmengen werden dann benutzt, um die durchschnittliche Abschätzung der Modellleistung  $E$  zu berechnen.



Bei der k-fachen Kreuzvalidierung wird als Standardwert für  $k$  10 benutzt, ein Wert, der erfahrungsgemäß für die meisten Anwendungen eine gute Wahl ist. So

weisen beispielsweise von Ron Kohavi mit aus der Praxis stammenden Datenmengen durchgeführte Experimente darauf hin, dass eine 10-fache Kreuzvalidierung in den meisten Fällen den besten Kompromiss zwischen Bias und Varianz darstellt (R. Kohavi et al., *A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection*, International Joint Conference on Artificial Intelligence (IJCAI), 14(12), Seiten 1137–1145, 1995).

Wenn wir allerdings relativ kleine Trainingsdatenmengen verwenden, kann es sinnvoll sein, die Anzahl der Teilmengen zu erhöhen. Erhöhen wir den Wert von  $k$ , werden bei jedem Durchlauf mehr Trainingsdaten verwendet, was zu einem niedrigeren Bias bei der Einschätzung der Verallgemeinerungsfähigkeit durch die Mittelwertbildung der einzelnen Schätzwerte führt. Allerdings verlängert ein großer Wert von  $k$  auch die zur Ausführung des Kreuzvalidierungsalgorithmus erforderliche Zeit und liefert Abschätzungen mit höherer Varianz, weil die Trainingsteilmengen einander ähnlicher sind. Wenn wir es andererseits mit großen Datenmengen zu tun haben, können wir einen kleineren Wert für  $k$  wählen, beispielsweise  $k = 5$ , und erhalten dennoch eine ziemlich präzise Einschätzung der durchschnittlichen Leistungsfähigkeit des Modells, während wir den Rechenaufwand für die wiederholte Anpassung und Bewertung der verschiedenen Teilmengen verringern.

### Tipp

Die sogenannte *Leave-One-Out-Kreuzvalidierung (LOO)* ist ein Spezialfall der  $k$ -fachen Kreuzvalidierung. Bei der LOO ist die Anzahl der Teilmengen gleich der Anzahl der Trainingsdatenmengen ( $k=n$ ), sodass bei jedem Durchlauf nur ein Trainingsobjekt zum Testen verwendet wird. Dieser Ansatz ist empfehlenswert, wenn man es mit sehr kleinen Datenmengen zu tun hat.

Im Vergleich zur  $k$ -fachen Kreuzvalidierung stellt die sogenannte *stratifizierte  $k$ -fache Kreuzvalidierung* eine leichte Verbesserung dar, weil das Bias verbessert und die Varianz der Einschätzung verringert werden – insbesondere in Fällen, in denen die Klassen ungleich verteilt sind, wie Ron Kohavi und andere in ihren oben genannten Untersuchungen gezeigt haben. Bei der stratifizierten Kreuzvalidierung wird darauf geachtet, dass die Klassen in den Teilmengen annähernd gleich verteilt sind. Dadurch ist gewährleistet, dass die einzelnen Teilmengen repräsentativ für die Klassenverteilung in der Trainingsdatenmenge sind. Das lässt sich mit dem `StratifiedKFold`-Iterator von scikit-learn veranschaulichen:

```
>>> import numpy as np
>>> from sklearn.model_selection import StratifiedKFold
>>> kfold = StratifiedKFold(n_splits=10,
...                         random_state=1).split(X_train,
...                         y_train)
```

```
>>> scores = []
>>> for k, (train, test) in enumerate(kfold):
...     pipe_lr.fit(X_train[train], y_train[train])
...     score = pipe_lr.score(X_train[test], y_train[test])
...     scores.append(score)
...     print('Fold: %2d, Class dist.: %s, Acc: %.3f' % (k+1,
...             np.bincount(y_train[train]), score))
Fold: 1, Class dist.: [256 153], Acc: 0.935
Fold: 2, Class dist.: [256 153], Acc: 0.935
Fold: 3, Class dist.: [256 153], Acc: 0.957
Fold: 4, Class dist.: [256 153], Acc: 0.957
Fold: 5, Class dist.: [256 153], Acc: 0.935
Fold: 6, Class dist.: [257 153], Acc: 0.956
Fold: 7, Class dist.: [257 153], Acc: 0.978
Fold: 8, Class dist.: [257 153], Acc: 0.933
Fold: 9, Class dist.: [257 153], Acc: 0.956
Fold: 10, Class dist.: [257 153], Acc: 0.956
>>> print('KV-Korrektklassifizierungsrate: %.3f +/- %.3f' % (
...             np.mean(scores), np.std(scores)))
KV-Korrektklassifizierungsrate: 0.950 +/- 0.014
```

Zunächst initialisieren wir den `StratifiedKFold`-Iterator des `sklearn.model_selection`-Moduls mit den Klassenbezeichnungen `y_train` der Trainingsdatenmenge und legen mit dem Parameter `n_splits` die Anzahl der Teilmengen fest. Beim Durchlaufen der  $k$  Teilmengen mit dem `kfold`-Iterator verwenden wir die von `train` zurückgegebenen Indizes, um die zu Beginn dieses Kapitels eingerichtete logistische Regression vorzunehmen. Durch den Einsatz der `pipe_lr`-Pipeline gewährleisten wir, dass die Daten bei jedem Durchlauf korrekt skaliert (z.B. standardisiert) sind. Dann verwenden wir die `test`-Indizes zur Berechnung des Korrektklassifizierungsraten-Scores des Modells, der in der Liste `scores` gespeichert wird, um zum Schluss die durchschnittliche Korrektklassifizierungsrate und die Standardabweichung der Abschätzung zu ermitteln.

Das vorangehende Codebeispiel ist nützlich, um die Funktionsweise der k-fachen Kreuzvalidierung zu demonstrieren, scikit-learn implementiert aber auch eine Bewertungsfunktion, die es erlaubt, ein Modell anhand der stratifizierten k-fachen Kreuzvalidierung effizienter zu beurteilen:

```
>>> from sklearn.model_selection import cross_val_score
>>> scores = cross_val_score(estimator=pipe_lr,
...                         X=X_train,
...                         y=y_train,
...                         cv=10,
...                         n_jobs=1)
```

```
>>> print('KV-Korrektklassifizierungsrate-Score:  
...           %s' % scores)  
KV-Korrektklassifizierungsrate-Score:  
[ 0.93478261  0.93478261  0.95652174  
  0.95652174  0.93478261  0.95555556  
  0.97777778  0.93333333  0.95555556  
  0.95555556]  
>>> print('KV-Korrektklassifizierungsrate:  
...           %.3f +/- %.3f' % (np.mean(scores),  
...           np.std(scores)))  
KV-Korrektklassifizierungsrate: 0.950 +/- 0.014
```

Ein besonders nützliches Feature dieses `cross_val_score`-Ansatzes ist, dass wir die Bewertung der verschiedenen Teilmengen auf mehrere CPUs des Rechners verteilen können. Wenn wir den Parameter `n_jobs` auf 1 setzen, wird wie beim vorangegangenen `StratifiedKFold`-Beispiel nur eine CPU verwendet, um die Berechnung der Leistung durchzuführen. Setzen wir jedoch `n_jobs=2`, wird die Berechnung der 10 Kreuzvalidierungsdurchläufe auf zwei CPUs verteilt (sofern verfügbar). Und wenn wir `n_jobs=-1` setzen, wird die Berechnung auf allen verfügbaren CPUs des Rechners parallel ausgeführt.

### Tipp

Eine ausführliche Erläuterung dazu, wie die Varianz der Verallgemeinerungsfähigkeit eines Kreuzvalidierungsverfahrens abgeschätzt wird, geht über den Rahmen dieses Buches hinaus, aber ich habe eine Reihe von Artikeln über Modellbewertung und Kreuzvalidierung verfasst, die diese Themen eingehender behandeln. Sie sind hier verfügbar:

<https://sebastianraschka.com/blog/2016/model-evaluation-selection-part1.html>

<https://sebastianraschka.com/blog/2016/model-evaluation-selection-part2.html>

<https://sebastianraschka.com/blog/2016/model-evaluation-selection-part3.html>

Darüber hinaus finden Sie eine umfangreiche Erklärung in einem exzellenten Artikel von M. Markatou et al. (M. Markatou, H. Tian, S. Biswas und G. M. Hripcak, *Analysis of Variance of Cross-validation Estimators of the Generalization Error*, Journal of Machine Learning Research, 6:1127-1168, 2005).

Weitere Kreuzvalidierungsverfahren, wie z.B. die .632-Bootstrap-Kreuzvalidierung, sind in einem Artikel von B. Efron und R. Tibshirani beschrieben (B. Efron und R. Tibshirani, *Improvements on Cross-validation: The .632+ Bootstrap Method*, Journal of the American Statistical Association, 92(438):548-560, 1997).

## 6.3 Algorithmen mit Lern- und Validierungskurven debuggen

In diesem Abschnitt werden wir zwei einfache, aber dennoch leistungsfähige Diagnosetools betrachten, die uns dabei helfen können, die Leistung eines Lernalgorithmus zu verbessern: *Lernkurven* und *Validierungskurven*. Zunächst werden wir erörtern, wie man mithilfe von Lernkurven feststellen kann, ob bei einem Lernalgorithmus Probleme mit einer Überanpassung (hohe Varianz) oder einer Unteranpassung (großes Bias) vorliegen. Darüber hinaus werden wir Validierungskurven betrachten, die uns dabei helfen können, die typischen bei Lernalgorithmen auftretenden Schwierigkeiten in den Griff zu bekommen.

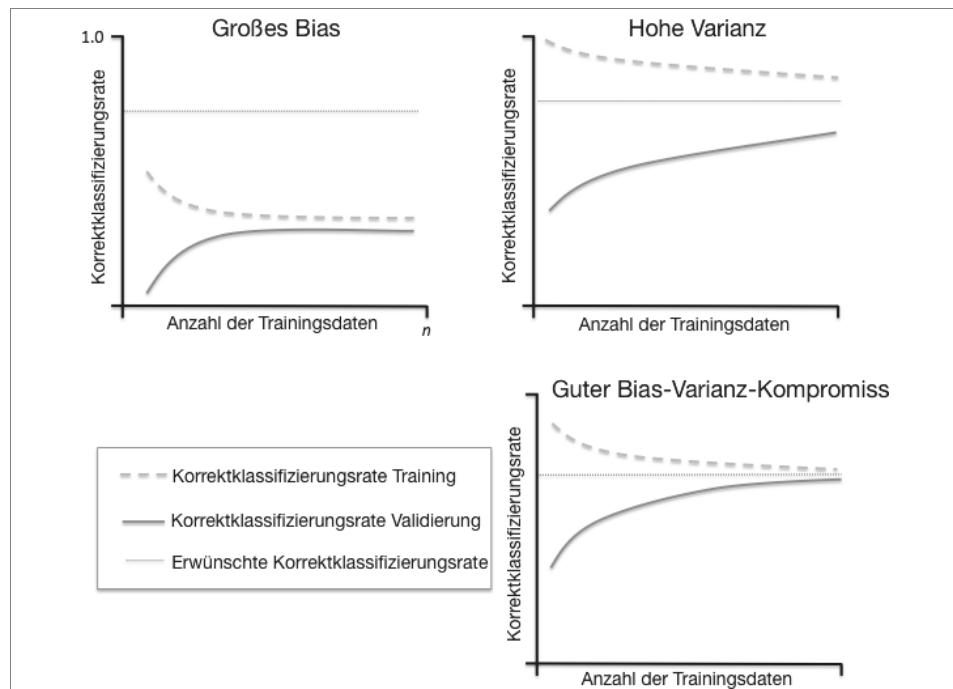
### 6.3.1 Probleme mit Bias und Varianz anhand von Lernkurven erkennen

Wenn ein Modell für eine gegebene Trainingsdatenmenge zu komplex ist – d.h., es besitzt zu viele Freiheitsgrade oder Parameter –, neigt es zu einer Überpassung und lässt sich schlecht auf unbekannte Daten verallgemeinern. In vielen Fällen hilft es, weitere Trainingsdaten zu sammeln, um das Ausmaß der Überanpassung zu verringern. In der Praxis kann es allerdings oft sehr schwierig oder schlicht und einfach undurchführbar sein, weitere Daten zu beschaffen. Wenn man die Korrektklassifizierungsrate des Modells bei Trainings- und Validierungsdaten als Funktion der Größe der Trainingsdatenmenge plottet, ist leicht erkennbar, ob das Modell eine hohe Varianz oder ein großes Bias aufweist und ob das Sammeln weiterer Daten hilfreich wäre, um dieses Problem in Angriff zu nehmen. Bevor wir jedoch die Ausgabe von Lernkurven mit scikit-learn betrachten, wollen wir uns anhand der folgenden Abbildung mit zwei gängigen Problemen befassen, die in Modellen auftreten.

Der Graph oben links zeigt ein Modell mit großem Bias. Die Korrektklassifizierungsrate ist sowohl bei den Trainings- als auch bei den Validierungsdaten gering, was darauf hinweist, dass eine Unteranpassung an die Trainingsdaten stattfindet. Um dieses Problem zu lösen, erhöht man üblicherweise die Anzahl der Modellparameter, indem man weitere Merkmale sammelt oder hinzufügt, oder man verringert das Ausmaß der Regularisierung, beispielsweise bei einer SVM oder einer Klassifizierung durch logistische Regression.

Der Graph oben rechts zeigt ein Modell mit hoher Varianz, was an der großen Lücke zwischen den Korrektklassifizierungsgraden bei Trainings- und Validierungsdaten erkennbar ist. Um diese Überanpassung zu vermindern, können wir weitere Trainingsdaten sammeln oder die Komplexität des Modells verringern, beispielsweise indem wir den Regularisierungsparameter erhöhen. Bei unregularisierten Modellen kann es auch helfen, die Anzahl der Merkmale mittels Merkmalsauswahl (siehe Kapitel 4) oder Merkmalsextraktion (siehe Kapitel 5) zu senken. Das Sammeln weiterer Trainingsdaten verringert für gewöhnlich die Wahrscheinlich-

keit einer Überanpassung – allerdings ist das nicht immer hilfreich, beispielsweise wenn die Trainingsdaten extrem verrauscht sind oder wenn das Modell schon sehr nah am Optimum ist.

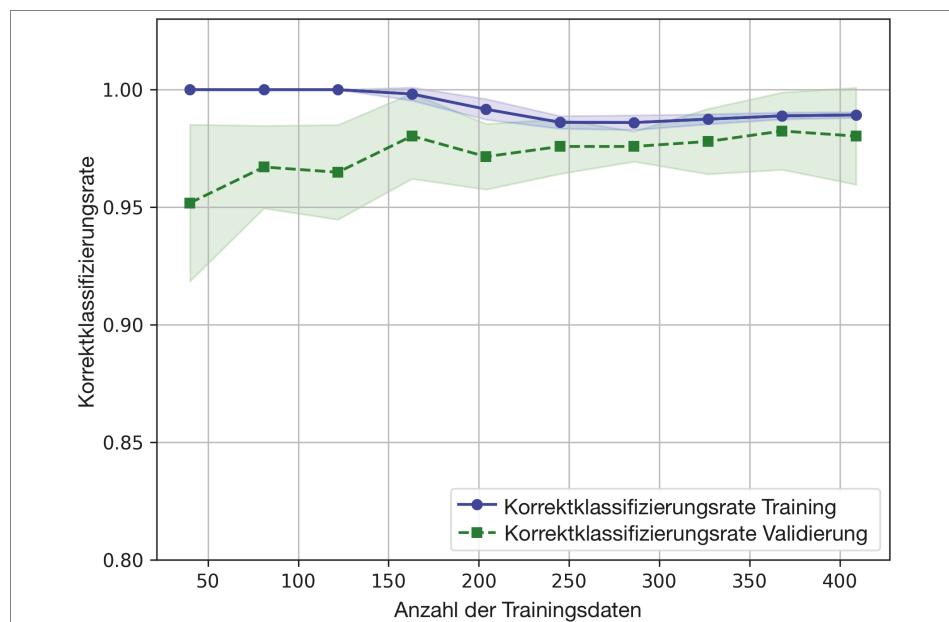


Im nächsten Abschnitt werden wir uns damit befassen, wie sich solche Probleme mit Validierungskurven lösen lassen, aber zunächst einmal sehen wir uns an, wie die Lernkurvenfunktion von scikit-learn zur Beurteilung des Modells genutzt werden kann:

```
>>> import matplotlib.pyplot as plt
>>> from sklearn.model_selection import learning_curve
>>> pipe_lr = make_pipeline(StandardScaler(),
...                         LogisticRegression(penalty='l2',
...                                             random_state=1))
>>> train_sizes, train_scores, test_scores = \
...         learning_curve(estimator=pipe_lr,
...                        X=X_train,
...                        y=y_train,
...                        train_sizes=np.linspace(0.1, 1.0, 10),
...                        cv=10,
...                        n_jobs=1)
>>> train_mean = np.mean(train_scores, axis=1)
>>> train_std = np.std(train_scores, axis=1)
>>> test_mean = np.mean(test_scores, axis=1)
```

```
>>> test_std = np.std(test_scores, axis=1)
>>> plt.plot(train_sizes, train_mean,
...             color='blue', marker='o',
...             markersize=5,
...             label='Korrektklassifizierungsrate Training')
>>> plt.fill_between(train_sizes,
...                     train_mean + train_std,
...                     train_mean - train_std,
...                     alpha=0.15, color='blue')
>>> plt.plot(train_sizes, test_mean,
...             color='green', linestyle='--',
...             marker='s', markersize=5,
...             label='Korrektklassifizierungsrate Validierung')
>>> plt.fill_between(train_sizes,
...                     test_mean + test_std,
...                     test_mean - test_std,
...                     alpha=0.15, color='green')
>>> plt.grid()
>>> plt.xlabel('Anzahl der Trainingsdaten')
>>> plt.ylabel('Korrektklassifizierungsrate')
>>> plt.legend(loc='lower right')
>>> plt.ylim([0.8, 1.0])
>>> plt.show()
```

Nach der erfolgreichen Ausführung des Codes erhalten wir die folgenden Lernkurven:



Über den `train_sizes`-Parameter der `learning_curve`-Funktion können wir die absolute oder relative Anzahl der Trainingsdaten steuern, die zur Erzeugung der Lernkurven verwendet werden. Hier setzen wir `train_sizes=np.linspace(0.1, 1.0, 10)`, um für die Größe der Trainingsdatenmengen 10 gleich große Intervalle zu erhalten. Die `learning_curve`-Funktion nutzt standardmäßig eine stratifizierte k-fache Kreuzvalidierung zur Berechnung der Korrektklassifizierungsraten, und über den Parameter `cv` setzen wir  $k = 10$ . Dann berechnen wir anhand der zurückgegebenen kreuzvalidierten Trainings- und Testscores für die verschiedenen Größen der Trainingsdatenmengen die durchschnittlichen Korrektklassifizierungsraten, die wir schließlich mit `matplotlibs plot`-Funktion ausgeben. Darüber hinaus fügen wir dem Diagramm mithilfe der `fill_between`-Funktion noch die Standardabweichungen der durchschnittlichen Korrektklassifizierungsraten hinzu, um die Varianz der Abschätzung anzuzeigen.

Wie Sie der vorangegangenen Abbildung entnehmen können, funktioniert unser Modell sowohl mit den Validierungsdaten als auch mit den Trainingsdaten ziemlich gut, nachdem das Modell mit mehr als 250 Objekten trainiert wurde. Außerdem ist erkennbar, dass die Korrektklassifizierungsraten bei Trainingsmengen mit weniger als 250 Objekten ansteigt und die Lücke zwischen den beiden Kurven größer wird – ein Hinweis auf eine zunehmende Überanpassung.

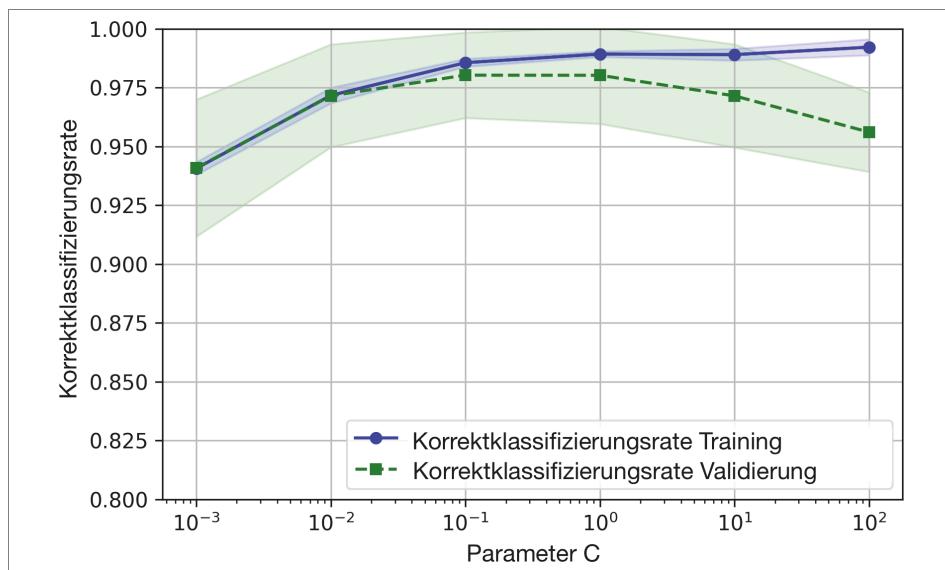
### 6.3.2 Überanpassung und Unteranpassung anhand von Validierungskurven erkennen

Validierungskurven sind ein nützliches Tool, um die Leistungsfähigkeit eines Modells zu verbessern, wenn es Probleme mit einer Über- oder Unteranpassung gibt. Validierungskurven ähneln Lernkurven, allerdings werden die Korrektklassifizierungsraten bei Trainings- und Testdaten nicht als Funktionen der Größe der Datenmenge ausgegeben – stattdessen variieren wir die Werte der Modellparameter, wie beispielsweise den Kehrwert des Regularisierungsparameters C einer logistischen Regression. Mit scikit-learn kann man eine Validierungskurve wie folgt erzeugen:

```
>>> from sklearn.model_selection import validation_curve
>>> param_range = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]
>>> train_scores, test_scores = validation_curve(
...                 estimator=pipe_lr,
...                 X=X_train,
...                 y=y_train,
...                 param_name='logisticregression__C',
...                 param_range=param_range,
...                 cv=10)
>>> train_mean = np.mean(train_scores, axis=1)
>>> train_std = np.std(train_scores, axis=1)
```

```
>>> test_mean = np.mean(test_scores, axis=1)
>>> test_std = np.std(test_scores, axis=1)
>>> plt.plot(param_range, train_mean,
...             color='blue', marker='o',
...             markersize=5,
...             label='Korrektklassifizierungsrate Training')
>>> plt.fill_between(param_range, train_mean + train_std,
...                     train_mean - train_std, alpha=0.15,
...                     color='blue')
>>> plt.plot(param_range, test_mean,
...             color='green', linestyle='--',
...             marker='s', markersize=5,
...             label='Korrektklassifizierungsrate Validierung')
>>> plt.fill_between(param_range,
...                     test_mean + test_std,
...                     test_mean - test_std,
...                     alpha=0.15, color='green')
>>> plt.grid()
>>> plt.xscale('log')
>>> plt.legend(loc='lower right')
>>> plt.xlabel('Parameter C')
>>> plt.ylabel('Korrektklassifizierungsrate')
>>> plt.ylim([0.8, 1.03])
>>> plt.show()
```

Der Code liefert die folgende Validierungskurve für den Parameter C:



Wie die `learning_curve`-Funktion nutzt auch die `validation_curve`-Funktion standardmäßig die stratifizierte k-fache Kreuzvalidierung zur Einschätzung der Leistung des Klassifizierers. In der `validation_curve`-Funktion legen wir den zu bewertenden Parameter fest – in diesem Fall `C`, den Kehrwert des Regularisierungsparameters des `LogisticRegression`-Klassifizierers, den wir als '`logisticregression_C`' angeben, um auf den mit dem `param_range`-Parameter festgelegten Wertebereich des `LogisticRegression`-Objekts in der scikit-learn-Pipeline zuzugreifen. Wie bei dem Lernkurvenbeispiel im vorangegangenen Abschnitt geben wir die durchschnittlichen Korrektklassifizierungsraten für Trainings- und Validierungsdaten sowie die dazugehörigen Standardabweichungen aus.

Die Unterschiede zwischen den Korrektklassifizierungsraten bei wechselnden Werten des Parameters `C` sind zwar nur gering, dennoch ist erkennbar, dass es bei einer Erhöhung der Regularisierungsstärke (kleinere Werte für `C`) zu einer leichten Unteranpassung kommt. Bei großen Werten für `C`, also einer geringeren Regularisierungsstärke, kommt es hingegen zu einer geringfügigen Überanpassung des Modells an die Daten. In diesem Fall scheint der geeignete Wert für `C` zwischen 0.01 und 0.1 zu liegen.

## 6.4 Feinabstimmung eines Lernmodells durch Rastersuche

Beim Machine Learning gibt es zwei Arten von Parametern: die aus den Trainingsdaten erlernten Parameter, beispielsweise die Gewichtungen einer logistischen Regression, und die Parameter eines Lernalgorithmus, die separat optimiert werden. Letztere sind die Abstimmungsparameter oder *Hyperparameter* eines Modells, beispielsweise der *Regularisierungsparameter* einer logistischen Regression oder die *Tiefe* eines Entscheidungsbaums.

Im vorangegangenen Abschnitt haben wir Validierungskurven zur Verbesserung der Leistung eines Modells genutzt, indem wir einen der Hyperparameter variierten. In diesem Abschnitt werden wir ein leistungsfähiges Verfahren zur Hyperparameteroptimierung betrachten, das als *Rastersuche* (engl. *Grid Search*) bezeichnet wird und die Leistung eines Modells durch das Aufspüren der *optimalen* Kombination der Hyperparameterwerte weiter verbessern kann.

### 6.4.1 Hyperparameter-Abstimmung durch Rastersuche

Der Ansatz der Rastersuche ist ganz einfach: Es handelt sich um eine Brute-Force-Methode, bei der wir eine Liste der Werte für die verschiedenen Hyperparameter angeben und der Computer die Leistung des Modells für alle möglichen Kombinationen bewertet, um die optimale zu ermitteln:

```
>>> from sklearn.model_selection import GridSearchCV
>>> from sklearn.svm import SVC
>>> pipe_svc = make_pipeline(StandardScaler(),
...                           SVC(random_state=1))
>>> param_range = [0.0001, 0.001, 0.01, 0.1,
...                  1.0, 10.0, 100.0, 1000.0]
>>> param_grid = [{ 'svc__C': param_range,
...                  'svc__kernel': ['linear']},
...                 { 'svc__C': param_range,
...                  'svc__gamma': param_range,
...                  'svc__kernel': ['rbf']}]
>>> gs = GridSearchCV(estimator=pipe_svc,
...                     param_grid=param_grid,
...                     scoring='accuracy',
...                     cv=10,
...                     n_jobs=-1)
>>> gs = gs.fit(X_train, y_train)
>>> print(gs.best_score_)
0.984615384615
>>> print(gs.best_params_)
{'svc__C': 100.0, 'svc__gamma': 0.001, 'svc__kernel': 'rbf'}
```

Der Code initialisiert ein `GridSearchCV`-Objekt des `sklearn.model_selection`-Moduls, damit wir eine Support Vector Machine Pipeline (SVM-Pipeline) trainieren und abstimmen können. Dem `param_grid`-Parameter von `GridSearchCV` weisen wir eine Liste mit Dictionaries zu, um die Parameter zu definieren, die wir abstimmen möchten. Bei der linearen SVM bewerten wir nur den Kehrwert des Regularisierungsparameters `svc__C`, bei der RBF-Kernel-SVM stimmen wir die Parameter `svc__C` und `svc_gamma` ab. Beachten Sie, dass es den Parameter `svc_gamma` nur bei Kernel-SVMs gibt.

Nachdem wir die Rastersuche mit den Trainingsdaten durchgeführt haben, erhalten wir den Score des am besten funktionierenden Modells über das Attribut `best_score_` und geben die entsprechenden Parameter aus, auf die wir über das Attribut `best_params_` zugreifen können. In diesem speziellen Fall liefert das RBF-Kernel-SVM-Modell mit '`svc__C' = 100.0` die beste Korrektklassifizierungsrate der k-fachen Kreuzvalidierung, nämlich 98,5 Prozent.

Zum Abschluss verwenden wir die unabhängige Testdatenmenge, um die Leistung des besten ausgewählten Modells abzuschätzen, das über das `best_estimator_`-Attribut des `GridSearchCV`-Objekts zur Verfügung steht:

```
>>> clf = gs.best_estimator_
>>> clf.fit(X_train, y_train)
>>> print('Korrektklassifizierungsrate:
...     %.3f' % clf.score(X_test, y_test))
Korrektklassifizierungsrate: 0.974
```

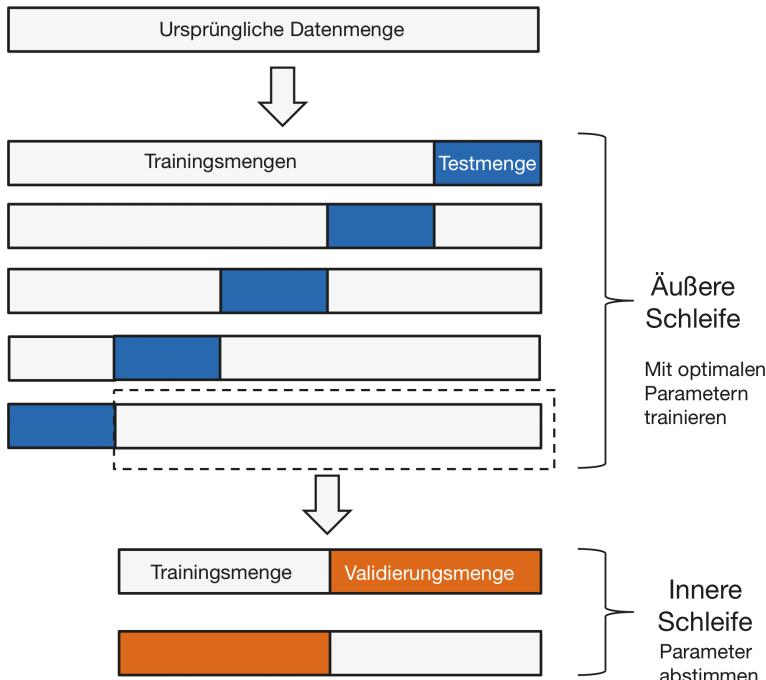
### Tipp

Die Rastersuche ist zwar ein leistungsfähiger Ansatz zum Aufspüren der optimalen Parameterkombination, die Bewertung aller möglichen Parameterkombinationen ist allerdings sehr rechenaufwendig. Ein alternativer Ansatz zur Überprüfung verschiedener Parameterkombinationen mit scikit-learn ist die randomisierte Suche. Mit der `RandomizedSearchCV`-Klasse von scikit-learn können wir zufällige Parameterkombinationen überprüfen und den erlaubten Rechenaufwand vorgeben. Weitere Informationen und Beispiele finden Sie unter [http://scikit-learn.org/stable/modules/grid\\_search.html#randomized-parameter-optimization](http://scikit-learn.org/stable/modules/grid_search.html#randomized-parameter-optimization).

## 6.4.2 Algorithmenauswahl durch verschachtelte Kreuzvalidierung

Die k-fache Kreuzvalidierung in Kombination mit einer Rastersuche ist ein nützlicher Ansatz zur Feinabstimmung der Leistung eines Lernmodells, bei dem die Werte der Hyperparameter wie im vorangegangenen Abschnitt variiert werden. Wenn wir allerdings eine Auswahl zwischen verschiedenen Lernalgorithmen treffen möchten, empfiehlt sich eine verschachtelte Kreuzvalidierung. In einer interessanten Untersuchung des Bias der Fehlerabschätzung kamen Varma und Simon zu dem Ergebnis, dass der wahre Fehler der Abschätzung bei einer verschachtelten Kreuzvalidierung im Vergleich zur Testdatenmenge nahezu unverzerrt ist (S. Varma und R. Simon, *Bias in Error Estimation When Using Cross-validation for Model Selection*, BMC Bioinformatics, 7(1):91, 2006).

Bei einer verschachtelten Kreuzvalidierung gibt es eine äußere k-fache Kreuzvalidierungsschleife zur Aufteilung der Daten in Trainings- und Testteilmengen sowie eine innere Schleife zur Auswahl des Modells anhand einer k-fachen Kreuzvalidierung der Trainingsteilmenge. Nach der Auswahl eines Modells wird die Testteilmenge zur Bewertung des Modells verwendet. In der Abbildung ist das Konzept einer verschachtelten Kreuzvalidierung mit nur fünf äußeren und zwei inneren Teilmengen dargestellt, das sich bei großen Datenmengen als nützlich erweist, bei denen der benötigte Rechenaufwand von Bedeutung ist. Diese spezielle Art der verschachtelten Kreuzvalidierung wird auch als *5x2-Kreuzvalidierung* bezeichnet.



Mit scikit-learn können wir eine verschachtelte Kreuzvalidierung folgendermaßen ausführen:

```
>>> gs = GridSearchCV(estimator=pipe_svc,
...                     param_grid=param_grid,
...                     scoring='accuracy',
...                     cv=2)
>>> scores = cross_val_score(gs, X_train, y_train,
...                           scoring='accuracy', cv=5)
>>> print('KV-Korrektklassifizierungsrate: %.3f +/- %.3f' % (
...     np.mean(scores), np.std(scores)))
KV-Korrektklassifizierungsrate: 0.974 +/- 0.015
```

Die zurückgegebene Korrektklassifizierungsrate der Kreuzvalidierung stellt eine gute Einschätzung dessen dar, was wir erwarten dürfen, wenn wir die Hyperparameter eines Modells abstimmen und es dann auf unbekannte Daten anwenden. Wir können die verschachtelte Kreuzvalidierung beispielsweise benutzen, um ein SVM-Modell mit einem einfachen Entscheidungsbaum-Klassifizierer zu vergleichen. Der Einfachheit halber variieren wir hier nur den `depth`-Parameter:

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> gs = GridSearchCV(
...     estimator=DecisionTreeClassifier(random_state=0),
```

```

...     param_grid=[{'max_depth': [1, 2, 3,
...                               4, 5, 6, 7, None]}],
...     scoring='accuracy',
...     cv=2)
>>> scores = cross_val_score(gs, X_train, y_train,
...                           scoring='accuracy', cv=5)
>>> print('KV-Korrektklassifizierungsrate: %.3f +/- %.3f' % (
...                           np.mean(scores), np.std(scores)))
KV-Korrektklassifizierungsrate: 0.934 +/- 0.016

```

Wie Sie sehen, ist die Leistung des SVM-Modells mit verschachtelter Kreuzvalidierung (97,4 Prozent) deutlich besser als diejenige des Entscheidungsbaums (93,4 Prozent). Wir würden daher erwarten, dass es zur Klassifizierung neuer Daten, die derselben Grundgesamtheit wie diese Datenmenge entstammen, die bessere Wahl ist.

## 6.5 Verschiedene Kriterien zur Leistungsbewertung

In den vorangegangenen Kapiteln und Abschnitten haben wir die Güte unserer Modelle anhand der Korrektklassifizierungsrate (Vertrauenswahrscheinlichkeit) beurteilt, einem im Allgemeinen brauchbaren Kriterium zur Bewertung der Leistung eines Modells. Es gibt jedoch noch einige weitere Kriterien, die zur Beurteilung der Klassifizierungsgüte eines Modells verwendet werden können, etwa die *Genauigkeit* (engl. *precision*), die *Trefferquote* (engl. *recall*) oder kombinierte Maße wie das sogenannte *F1-Maß*.

### 6.5.1 Interpretation einer Wahrheitsmatrix

Bevor wir uns mit den Details der verschiedenen Bewertungskriterien befassen, erstellen wir eine *Wahrheitsmatrix* (die auch als *Konfusionsmatrix* bezeichnet wird) zur Darstellung der Leistung eines Lernalgorithmus. Dabei handelt es sich um eine quadratische Matrix, in der die Anzahlen *richtig positiver*, *richtig negativer*, *falsch positiver* und *falsch negativer* Vorhersagen eines Klassifizierers angegeben sind (siehe Abbildung).

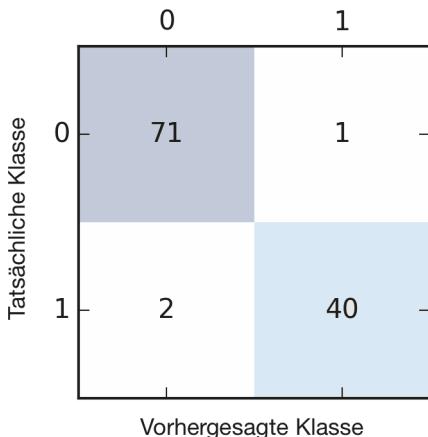
		Vorhergesagte Klasse	
		P	N
Tatsächliche Klasse	P	Richtig Positive (RP)	Falsch Negative (FN)
	N	Falsch Positive (FP)	Richtig Negative (RN)

Diese Kriterien lassen sich zwar leicht von Hand ermitteln, indem man die tatsächlichen und die vorhergesagten Klassenbezeichnungen vergleicht, scikit-learn stellt aber eine komfortable `confusion_matrix`-Funktion bereit, die wir folgendermaßen nutzen können:

```
>>> from sklearn.metrics import confusion_matrix
>>> pipe_svc.fit(X_train, y_train)
>>> y_pred = pipe_svc.predict(X_test)
>>> confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
>>> print(confmat)
[[71  1]
 [ 2 40]]
```

Das nach der Ausführung des Codes zurückgegebene Array liefert uns Informationen über die verschiedenen Fehlertypen, die dem Klassifizierer bei der Testdatenmenge unterlaufen sind und die wir dann mithilfe der `matshow`-Funktion von Matplotlib entsprechend der vorangegangenen Abbildung als Wahrheitstmatrix ausgeben können:

```
>>> fig, ax = plt.subplots(figsize=(2.5, 2.5))
>>> ax.matshow(confmat, cmap=plt.cm.Blues, alpha=0.3)
>>> for i in range(confmat.shape[0]):
...     for j in range(confmat.shape[1]):
...         ax.text(x=j, y=i,
...                 s=confmat[i, j],
...                 va='center', ha='center')
>>> plt.xlabel('Vorhergesagte Klasse')
>>> plt.ylabel('Tatsächliche Klasse')
>>> plt.show()
```



Wenn wir annehmen, dass in diesem Beispiel Klasse 1 (bösartig) die positive Klasse ist, klassifiziert das Modell 71 der zur Klasse 0 (richtig Negative) und 40 der zur Klasse 1 (richtig Positive) zugehörigen Datenmengen korrekt. Allerdings ordnet es auch eine der zur Klasse 0 zugehörigen Datenmengen der Klasse 1 zu (falsch Positive) und sagt voraus, dass zwei Stichproben gutartig sind, obwohl es sich tatsächlich um bösartige Tumoren handelt (falsch Negative). Im nächsten Abschnitt werden Sie erfahren, wie wir diese Informationen verwenden können, um verschiedene Fehlerkriterien zu berechnen.

### 6.5.2 Optimierung der Genauigkeit und der Trefferquote eines Klassifizierungsmodells

Sowohl die *Fehlerquote* (FQ) als auch die *Korrektklassifizierungsrate* (KKR) liefern Informationen darüber, wie viele Objekte fehlklassifiziert werden. Die Fehlerquote ergibt sich aus der Anzahl der falschen Vorhersagen geteilt durch die Gesamtzahl der Vorhersagen, und die Korrektklassifizierungsrate ist die Anzahl der richtigen Vorhersagen geteilt durch die Gesamtzahl der Vorhersagen:

$$FQ = \frac{FP + FN}{FP + FN + RP + RN}$$

Die Korrektklassifizierungsrate kann dann anhand der Fehlerquote berechnet werden:

$$KKR = \frac{RP + RN}{FP + FN + RP + RN} = 1 - FQ$$

Die *Richtig-Positiv-Rate* (RPR) und die *Falsch-Positiv-Rate* (FPR) sind Leistungskriterien, die besonders bei unausgewogenen Klassifizierungsaufgaben nützlich sind:

$$FPR = \frac{FP}{N} = \frac{FP}{FP + RN}$$

$$RPR = \frac{RP}{P} = \frac{RP}{FN + RP}$$

Bei der Tumordiagnose sind wir beispielsweise stärker daran interessiert, bösartige Tumoren zu erkennen, um einen Patienten entsprechend behandeln zu können. Es ist allerdings auch wichtig, die Anzahl der gutartigen Tumoren, die irrtümlich als bösartig eingestuft werden (falsche Positive), möglichst gering zu halten, um einen Patienten nicht unnötig zu beunruhigen. Im Gegensatz zur FPR liefert die Richtig-Positiv-Rate auch Informationen über den Anteil der positiven (oder relevanten) Objekte, die in der Gesamtmenge der Positiven P korrekt identifiziert wurden.

Die *Genauigkeit* (GEN) und die *Trefferquote* (TQ) stehen in enger Beziehung zu den Richtig-Positiv- und Richtig-Negativ-Raten – tatsächlich ist die Trefferquote ein Synonym für die Richtig-Positiv-Rate:

$$GEN = \frac{RP}{RP + FP}$$

$$TQ = RPR = \frac{RP}{P} = \frac{RP}{FN + RP}$$

In der Praxis wird häufig eine Kombination aus Genauigkeit und Trefferquote benutzt, das sogenannte *F1-Maß*:

$$F1 = 2 \frac{GEN \times TQ}{GEN + TQ}$$

Die genannten Kriterien sind alle in scikit-learn implementiert und können wie im folgenden Codefragment vom `sklearn.metrics`-Modul importiert werden:

```
>>> from sklearn.metrics import precision_score
>>> from sklearn.metrics import recall_score, f1_score
>>> print('Genauigkeit: %.3f' % precision_score(
...         y_true=y_test, y_pred=y_pred))
Genauigkeit: 0.976
>>> print('Trefferquote: %.3f' % recall_score(
...         y_true=y_test, y_pred=y_pred))
Trefferquote: 0.952
>>> print('F1: %.3f' % f1_score(
...         y_true=y_test, y_pred=y_pred))
F1: 0.964
```

Darüber hinaus können wir neben der Korrektklassifizierungsrate (*accuracy*) bei einer Rastersuche über den `scoring`-Parameter weitere Bewertungskriterien zugrunde legen. Eine vollständige Liste der möglichen Werte für den `scoring`-Parameter finden Sie unter [http://scikit-learn.org/stable/modules/model\\_evaluation.html](http://scikit-learn.org/stable/modules/model_evaluation.html).

Denken Sie daran, dass in scikit-learn die positive Klasse die Klassenbezeichnung 1 erhält. Wenn Sie eine andere positive Klassenbezeichnung vergeben möchten, können Sie mithilfe der `make_scorer`-Funktion eine eigene Bewertungsfunktion erstellen, die dann unmittelbar als Argument für den `scoring`-Parameter von `GridSearchCV` übergeben wird (in diesem Beispiel dient das F1-Maß `f1_score` als Kriterium):

```
>>> from sklearn.metrics import make_scorer, f1_score
>>> scorer = make_scorer(f1_score, pos_label=0)
```

```
>>> gs = GridSearchCV(estimator=pipe_svc,
...                     param_grid=param_grid,
...                     scoring=scorer,
...                     cv=10)
>>> gs.fit(X_train, y_train)
>>> print(gs.best_score_)
0.986202145696
>>> print(gs.best_params_)
{'svc__C': 10.0, 'svc__gamma': 0.01, 'svc__kernel': 'rbf'}
```

### 6.5.3 Receiver-Operating-Characteristic-Diagramme

*Receiver-Operating-Characteristic-Diagramme* (*Grenzwertoptimierungskurven* oder kurz *ROC-Kurven*) sind ein nützliches Tool, um Klassifizierungsmodelle anhand ihrer Leistung hinsichtlich der Falsch-Positiv- und Richtig-Positiv-Rate auszuwählen. Die Raten werden berechnet, indem die Entscheidungsgrenze des Klassifizierers verschoben wird. Die Diagonale eines ROC-Diagramms kann als zufälliges Raten aufgefasst werden, und Klassifizierungsmodelle, die sich unterhalb der Diagonalen befinden, gelten als unzureichender als zufälliges Raten. Ein perfekter Klassifizierer würde sich in der linken oberen Ecke des Diagramms befinden und besäße eine Richtig-Positiv-Rate von 1 sowie eine Falsch-Positiv-Rate von 0. Die Fläche unter der ROC-Kurve, die sogenannte *ROC AUC* (*Receiver Operator Characteristic Area Under Curve*) dient zur Beschreibung der Leistung eines Klassifizierungsmodells.

#### Tipp

Ähnlich den ROC-Kurven kann man auch Genauigkeit-Trefferquote-Kurven für die verschiedenen Wahrscheinlichkeitsschwellenwerte eines Klassifizierers berechnen. Eine Funktion zum Plotten von Kurven dieser Art ist in scikit-learn implementiert und unter [http://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision\\_recall\\_curve.html](http://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_curve.html) dokumentiert.

Die Ausführung des nachstehenden Codes erzeugt die ROC-Kurve eines Klassifizierers, der nur zwei Merkmale der Wisconsin-Brustkrebs-Datensammlung verwendet, um vorherzusagen, ob ein Tumor gut- oder bösartig ist. Wir benutzen zwar dieselbe logistische Regressions-Pipeline wie zuvor definiert, machen dem Klassifizierer die Aufgabe allerdings etwas schwerer, damit die resultierende ROC-Kurve etwas interessanter wird. Aus diesem Grund beschränken wir auch die Anzahl der Teilmengen in der `StratifiedKFold`-Validierung auf drei. Hier der Code:

```
>>> from sklearn.metrics import roc_curve, auc
>>> from scipy import interp
>>> pipe_lr = make_pipeline(StandardScaler(),
...                         PCA(n_components=2)),
...                         LogisticRegression(penalty='l2',
...                                         random_state=1,
...                                         C=100.0))
>>> X_train2 = X_train[:, [4, 14]]
>>> cv = StratifiedKFold(n_splits=3,
...                         random_state=1).split(X_train,
...                                               y_train))
>>> fig = plt.figure(figsize=(7, 5))
>>> mean_tpr = 0.0
>>> mean_fpr = np.linspace(0, 1, 100)
>>> all_tpr = []

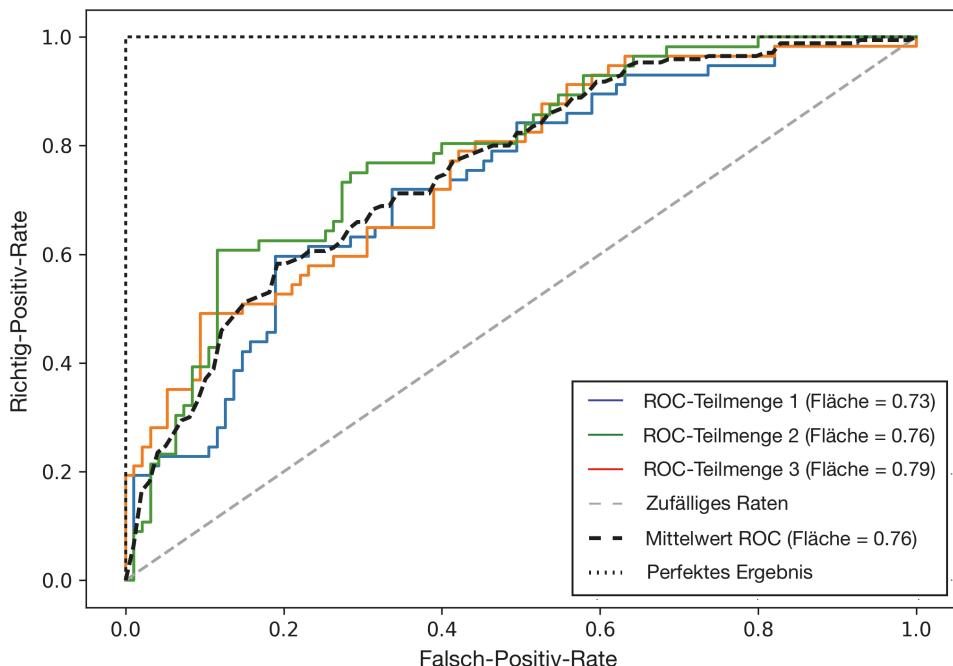
>>> for i, (train, test) in enumerate(cv):
...     probas = pipe_lr.fit(X_train2[train],
...                           y_train[train]).predict_proba(X_train2[test])
...     fpr, tpr, thresholds = roc_curve(y_train[test],
...                                      probas[:, 1],
...                                      pos_label=1)
>>>     mean_tpr += interp(mean_fpr, fpr, tpr)
>>>     mean_tpr[0] = 0.0
>>>     roc_auc = auc(fpr, tpr)
>>>     plt.plot(fpr,
...               tpr,
...               lw=1,
...               label='ROC-Teilmenge %d (Fläche = %0.2f)' %
...                     (i+1, roc_auc))
>>> plt.plot([0, 1],
...           [0, 1],
...           linestyle='--',
...           color=(0.6, 0.6, 0.6),
...           label='Zufälliges Raten')
>>> mean_tpr /= len(cv)
>>> mean_tpr[-1] = 1.0
>>> mean_auc = auc(mean_fpr, mean_tpr)
>>> plt.plot(mean_fpr, mean_tpr, 'k--',
...           label='Mittelwert ROC (Fläche = %0.2f)' %
...                 mean_auc, lw=2)
>>> plt.plot([0, 0, 1],
...           [0, 1, 1],
...           linestyle=':',
...           color='black',
```

```

...           label='Perfektes Ergebnis')
>>> plt.xlim([-0.05, 1.05])
>>> plt.ylim([-0.05, 1.05])
>>> plt.xlabel('Falsch-Positiv-Rate')
>>> plt.ylabel('Richtig-Positiv-Rate')
>>> plt.legend(loc="lower right")
>>> plt.show()

```

Im Beispielcode verwenden wir die bereits bekannte `StratifiedKFold`-Klasse von scikit-learn und berechnen in der `pipe_lr`-Pipeline mit der `roc_curve`-Funktion des `sklearn.metrics`-Moduls die ROC-Leistung des `LogisticRegression`-Klassifizierers bei jedem Durchlauf getrennt. Außerdem interpolieren wir mit der von SciPy importierten `interp`-Funktion anhand der drei Teilmengen die durchschnittliche ROC-Kurve und berechnen mit der `auc`-Funktion die Fläche unter der Kurve. Die resultierende ROC-Kurve weist darauf hin, dass es eine gewisse Varianz zwischen den verschiedenen Teilmengen gibt. Die durchschnittliche ROC-Fläche (0.76) liegt zwischen einem perfekten Ergebnis (1.0) und zufälligem Raten (0.5).



Wenn wir nur an der ROC-Fläche interessiert wären, könnten wir die `roc_auc_score`-Funktion auch direkt aus dem `sklearn.metrics`-Submodul importieren.

Die Leistung eines Klassifizierers durch die ROC AUC zu beschreiben, kann weitere Einsichten in das Verhalten des Klassifizierers bei unausgewogenen Daten-

mengen gewähren. Die Korrektklassifizierungsrate kann als einzelner Punkt auf einer ROC-Kurve interpretiert werden, allerdings hat A.P. Bradley gezeigt, dass die Bewertungen durch die ROC AUC und die Korrektklassifizierungsrate weitgehend übereinstimmen (A.P. Bradley, *The Use of the Area Under the ROC Curve in the Evaluation of Machine Learning Algorithms*, Pattern Recognition, 30(7):1145-1159, 1997).

#### 6.5.4 Bewertungskriterien für Mehrfachklassifizierungen

Die bislang erörterten Bewertungskriterien bezogen sich stets auf binäre Klassifizierungssysteme. Allerdings implementiert scikit-learn auch *Makro-* und *Mikro-*Methoden zur Mittelwertbildung, um diese Bewertungskriterien via OvA-Klassifizierung (One-vs.-All) auf Problemstellungen mit mehreren Klassen zu erweitern. Der *Mikro-Durchschnitt* wird anhand der Anzahl der richtig Positiven, richtig Negativen, falsch Positiven und falsch Negativen des Systems berechnet. Der *Mikro-Durchschnitt* der Genauigkeit eines Systems mit  $k$  Klassen errechnet sich folgendermaßen:

$$GEN_{\text{mikro}} = \frac{RP_1 + \dots + RP_k}{RP_1 + \dots + RP_k + FP_1 + \dots + FP_k}$$

Der *Makro-Durchschnitt* ergibt sich einfach als Durchschnittswert der verschiedenen Systeme:

$$GEN_{\text{makro}} = \frac{GEN_1 + \dots + GEN_k}{k}$$

Die *Mikro-Mittelwertbildung* erweist sich als nützlich, wenn wir die einzelnen Instanzen oder Vorhersagen gleich gewichten möchten, während die *Makro-Mittelwertbildung* bei der Bewertung der Gesamtleistung eines Klassifizierers hinsichtlich der häufigsten Klassenbezeichnungen alle Klassen gleich gewichtet.

Wenn wir für die Bewertung von Modellen für die Mehrfachklassifizierung binäre Leistungskriterien benutzen, verwendet scikit-learn standardmäßig eine normierte oder gewichtete Variante der *Makro-Mittelwertbildung*. Der gewichtete *Makro-Mittelwert* wird gebildet, indem bei der Berechnung der einzelnen Mittelwerte die Scores der jeweiligen Klassenbezeichnungen mit der Anzahl der tatsächlich vorhandenen Instanzen gewichtet werden. Der gewichtete *Makro-Mittelwert* ist nützlich, wenn wir es mit einer unausgewogenen Klassenverteilung zu tun haben, also wenn sich die Anzahl der Instanzen der verschiedenen Klassenbezeichnungen unterscheiden.

Standardmäßig verwendet scikit-learn für Aufgabenstellungen mit mehreren Klassen den gewichteten *Makro-Mittelwert*, wir können aber über den *average*-Parameter der verschiedenen aus dem `sklearn-metrics`-Modul importierten

Bewertungsfunktionen die Methode der Mittelwertbildung angeben, beispielsweise bei der `precision_score`- oder `make_scorer`-Funktion:

```
>>> pre_scorer = make_scorer(score_func=precision_score,
...                             pos_label=1,
...                             greater_is_better=True,
...                             average='micro')
```

## 6.6 Handhabung unausgewogener Klassenverteilung

Wir haben in diesem Kapitel mehrmals unausgewogene Klassenverteilungen erwähnt, aber bislang versäumt, zu erörtern, wie man diese in angemessener Weise handhabt. In der Praxis sind unausgewogene Klassenverteilungen ein ziemlich häufig auftretendes Problem – die zu einer Klasse oder zu mehreren Klassen zugehörigen Objekte sind in der Datenmenge überrepräsentiert. Dieses Phänomen tritt in verschiedenen Bereichen auf, etwa beim Spamfiltern, bei der Betrugserkennung oder bei medizinischen Reihenuntersuchungen.

Nehmen wir beispielsweise an, die in diesem Kapitel verwendete Brustkrebs-Datensammlung besteht zu 90 Prozent aus gesunden Patienten. In diesem Fall könnten wir eine Korrektklassifizierungsrate vom 90% erzielen, indem wir einfach für alle Objekte die häufigste Klasse (gutartiger Tumor) vorhersagen. Wenn ein mit dieser Datenmenge trainiertes Modell eine Korrektklassifizierungsrate von ca. 90% erzielt, würde das bedeuten, dass unser Modell überhaupt nichts Nützliches aus den in der Datensammlung vorhandenen Merkmalen gelernt hat.

In diesem Abschnitt werden wir uns kurz mit einigen Verfahren befassen, die zur Handhabung unausgewogener Klassenverteilungen geeignet sind. Aber bevor wir das in Angriff nehmen, erstellen wir zunächst einmal eine unausgewogene Datenmenge anhand der Brustkrebs-Datensammlung, die ursprünglich aus 357 gutartigen (Klasse 0) und 212 bösartigen Tumoren (Klasse 1) besteht:

```
>>> X_imb = np.vstack((X[y == 0], X[y == 1][:40]))
>>> y_imb = np.hstack((y[y == 0], y[y == 1][:40]))
```

Der Codeabschnitt bildet aus allen 357 gutartigen und 40 bösartigen Tumoren eine ausgeprägt unausgewogene Datenmenge. Wenn wir die Korrektklassifizierungsrate eines Modells berechnen, das wie eben erwähnt immer nur die häufigste Klasse (gutartig, Klasse 0) vorhersagt, ergibt sich ein Wert von rund 90%:

```
>>> y_pred = np.zeros(y_imb.shape[0])
>>> np.mean(y_pred == y_imb) * 100
89.924433249370267
```

Bei der Anpassung eines Klassifizierers an eine solche Datenmenge ist es daher sinnvoll, sich beim Vergleich verschiedener Modelle auf andere Kriterien als die Korrektklassifizierungsrate zu konzentrieren, wie Genauigkeit, Trefferquote oder die ROC-Kurve – was immer für den vorliegenden Anwendungsfall am bedeutsamsten ist. Wenn wir beispielsweise zum Ziel haben, den Großteil der Patienten mit bösartigem Tumor zu identifizieren, um ihnen eine weitere Untersuchung nahezulegen, sollten wir die Trefferquote als Kriterium anlegen. Und beim Spamfiltern sollen E-Mails nicht als Spam gekennzeichnet werden, wenn sich das System nicht sehr sicher ist, dass es sich tatsächlich um Spam handelt. Hier wäre womöglich die Genauigkeit ein geeigneteres Kriterium.

Eine unausgewogene Klassenverteilung wirkt sich jedoch nicht nur auf die Bewertung eines Lernmodells aus, sondern beeinflusst auch die eigentliche Anpassung des Modells. Da Lernalgorithmen typischerweise eine Belohnungs- oder Straffunktion optimieren, die sich als Summe aller bei der Anpassung des Modells beobachteten Objekte ergibt, wird die Entscheidungsregel wahrscheinlich zugunsten der häufigsten Klasse verzerrt sein. Oder anders ausgedrückt: Der Algorithmus bildet implizit ein Modell, das die Vorhersage anhand der häufigsten Klasse in der Datenmenge optimiert, um während des Trainings die Straffunktion zu minimieren oder die Belohnungsfunktion zu maximieren.

Eine Möglichkeit, die unausgewogene Klassenverteilung während der Anpassung des Modells zu handhaben, besteht darin, falsche Vorhersagen der Klassenzugehörigkeit der weniger häufigen Klasse härter zu bestrafen. In scikit-learn kann eine solche Bestrafung komfortabel durch die Zuweisung des Parameters `class_weight='balanced'` erreicht werden, der bei den meisten Klassifizierern implementiert ist.

Zu den weiteren Methoden der Handhabung von unausgewogenen Klassenverteilungen gehört die Erhöhung des Anteils der weniger häufigen Klasse (Upsampling) oder die Verringerung des Anteils der häufigsten Klasse (Downsampling) sowie das Erzeugen synthetischer Trainingsdaten. Leider gibt es für dieses Problem keine allgemeingültige beste Lösung und auch kein Verfahren, das für alle Anwendungsbereiche bestens geeignet wäre. In der Praxis empfiehlt es sich daher, diejenige Methode zu wählen, die am angemessensten erscheint.

In der scikit-learn-Bibliothek ist eine einfache `resample`-Funktion implementiert, die dabei hilft, den Anteil der weniger häufigen Klasse zu erhöhen, indem sie der Datenmenge neue Objekte entnimmt (mit Ersetzung). Der nachstehende Code entnimmt wiederholt Stichproben der in der unausgewogenen Brustkrebs-Datenmenge weniger häufigen Klasse (hier Klasse 1) und wiederholt das, bis sie genauso häufig vorkommt wie Klasse 0:

```
>>> from sklearn.utils import resample
>>> print('Anzahl Klasse 1 vorher:',
...      X_imb[y_imb == 1].shape[0])
Anzahl Klasse 1 vorher: 40
>>> X_upsampled, y_upsampled = resample(X_imb[y_imb == 1],
...           y_imb[y_imb == 1],
...           replace=True,
...           n_samples=X_imb[y_imb == 0].shape[0],
...           random_state=123)
>>> print('Anzahl Klasse 1 nachher:',
...       X_upsampled.shape[0])
Anzahl Klasse 1 nachher: 357
```

Nach dem Resampling können wir die ursprünglich zur Klasse 0 gehörenden Objekte mit der Teilmenge der zur Klasse 1 zugehörigen Objekten zusammenlegen und erhalten so eine ausgewogene Datenmenge:

```
>>> X_bal = np.vstack((X[y == 0], X_upsampled))
>>> y_bal = np.hstack((y[y == 0], y_upsampled))
```

Dementsprechend würde eine auf einer Mehrheitsentscheidung beruhende Entscheidungsregel eine Korrektklassifizierungsrate von nur 50% erzielen:

```
>>> y_pred = np.zeros(y_bal.shape[0])
>>> np.mean(y_pred == y_bal) * 100
```

Auf ähnliche Weise könnten wir auch den Anteil der häufigsten Klasse verringern, indem wir Trainingsdaten aus der Datenmenge entfernen. Um das mithilfe der `resample`-Funktion zu erledigen, müssten im Codebeispiel lediglich die Klassenbezeichnungen 0 und 1 vertauscht werden.

### Tipp

Das Erzeugen synthetischer Trainingsdaten ist ein weiteres Verfahren zur Handhabung unausgewogener Klassenverteilungen, das allerdings über den Rahmen dieses Buches hinausgeht. Der wohl gebräuchlichste dieser Algorithmen heißt SMOTE (Synthetic Minority Over-sampling Technique). Mehr zu diesem Thema finden Sie in der Forschungsarbeit von Nitesh Chawla et al.: *SMOTE: Synthetic Minority Over-sampling Technique, Journal of Artificial Intelligence Research*, 16: 321-357, 2002. Sie sollten sich außerdem die Python-Bibliothek `imbalanced-learn` ansehen, die sich komplett auf unausgewogene Klassenverteilungen konzentriert (<https://github.com/scikit-learn-contrib/imbalanced-learn>).

## 6.7 Zusammenfassung

Am Anfang des Kapitels haben wir erörtert, wie sich die verschiedenen Transformationsverfahren und Klassifizierer zu komfortablen Modell-Pipelines verknüpfen lassen, die uns dabei helfen, Lernmodelle effizienter zu trainieren und zu bewerten. Wir haben diese Pipelines dann dazu verwendet, eine k-fache Kreuzvalidierung durchzuführen, eine der grundlegenden Techniken zur Auswahl und Bewertung von Modellen. Mithilfe dieses Validierungsverfahrens haben wir Lern- und Validierungskurven benutzt, um gängige Probleme von Lernalgorithmen zu diagnostizieren, wie etwa Überanpassung und Unteranpassung. Und mit einer Rastersuche haben wir unser Modell noch feiner abgestimmt. Zum Abschluss dieses Kapitels haben wir schließlich die Wahrheitsmatrix sowie verschiedene Bewertungskriterien untersucht, die sich zur weiteren Optimierung der Modellleistung bei gegebener Aufgabenstellung nutzen lassen. Nun sollten Sie mit den grundlegenden Verfahren gut gerüstet sein, um erfolgreich überwachte Lernmodelle zur Klassifizierung entwickeln zu können.

Im nächsten Kapitel werden wir Ensemblemethoden betrachten, die es uns ermöglichen, mehrere Modelle und Klassifizierungsalgorithmen miteinander zu kombinieren, um so die Vorhersagekraft eines Systems für das Machine Learning weiter zu steigern.



# Kombination verschiedener Modelle für das Ensemble Learning

Im vorangegangenen Kapitel haben wir uns auf die bewährten Verfahren zur Anpassung und Bewertung verschiedener Klassifizierungsmodelle konzentriert. In diesem Kapitel bauen wir auf diesen Vorgehensweisen auf und erkunden verschiedene Methoden zum Erstellen eines Satzes von Klassifizierern, der in seiner Gesamtheit oft eine höhere Vorhersagekraft besitzt als die einzelnen darin enthaltenen Klassifizierer selbst.

Die Themen in diesem Kapitel sind:

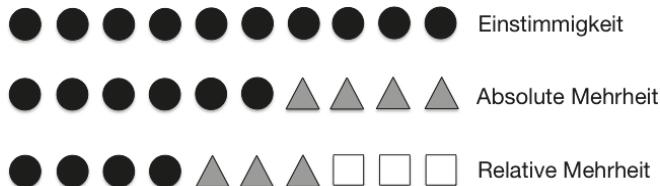
- Vorhersagen anhand von Mehrheitsentscheidungen treffen
- Verringern der Überanpassung durch die wiederholte Auswahl zufälliger Teilmengen der Trainingsdatenmenge
- Mit schwachen Klassifizierern leistungsfähige Modelle entwickeln, die aus ihren Fehlern lernen

## 7.1 Ensemble Learning

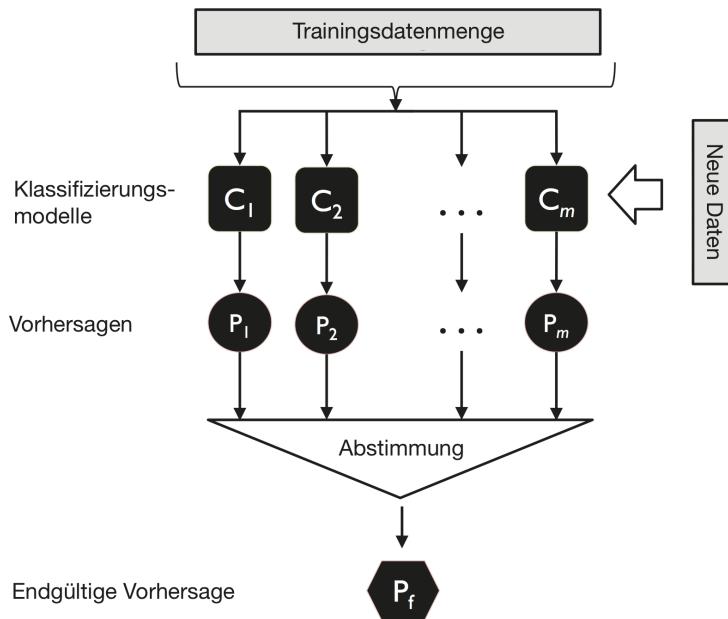
*Ensemblemethoden* haben zum Ziel, mehrere Klassifizierer zu einem Meta-Klassifizierer zu kombinieren, der über eine bessere Verallgemeinerungsfähigkeit verfügt als jeder einzelne Klassifizierer. Nehmen wir beispielsweise an, dass uns die Vorhersagen von 10 Experten vorliegen. Dann ermöglichen es uns die Ensemblemethoden, diese Vorhersagen strategisch so miteinander zu kombinieren, dass wir zu einer Vorhersage gelangen, die genauer und robuster ist als die Vorhersagen der einzelnen Experten. Wie Sie später in diesem Kapitel noch sehen werden, gibt es verschiedene Ansätze, solche Klassifizierer-Ensembles zu erzeugen. Als Erstes befassen wir uns im Folgenden mit der grundsätzlichen Arbeitsweise von Ensembles und erörtern, warum sie typischerweise eine gute Verallgemeinerungsfähigkeit besitzen.

In diesem Kapitel konzentrieren wir uns auf die verbreitetste Ensemblemethode, die auf dem Prinzip der *Mehrheitsentscheidung* beruht. Mit Mehrheitsentscheidung ist in diesem Zusammenhang schlicht und einfach gemeint, dass wir die Klassenbezeichnung auswählen, die von den meisten Klassifizierern vorhergesagt wird, sprich mehr als 50 Prozent der Stimmen erhält (*absolute Mehrheit*). Genau genom-

men bezieht sich die Mehrheitsentscheidung nur auf binäre Klassifizierungen. Das zugrunde liegende Prinzip kann jedoch auch problemlos für mehrere Klassen verallgemeinert werden – im Sinne einer *relativen* Mehrheit. In diesem Fall wählen wir dann die Klassenbezeichnung mit den meisten Stimmen aus. Die folgende Abbildung illustriert das Prinzip der absoluten und der relativen Mehrheit anhand eines 10 Klassifizierer umfassenden Ensembles, wobei die verschiedenen Symbole (Dreieck, Quadrat und Kreis) jeweils eine Klassenbezeichnung repräsentieren:



Wir fangen damit an,  $m$  verschiedene Klassifizierer  $C_1, \dots, C_m$  zu trainieren. Dabei kann das Ensemble aus verschiedenen Klassifizierungsalgorithmen bestehen, beispielsweise aus Entscheidungsbäumen, Support Vector Machines, logistischen Regressionen usw. Alternativ könnten wir auch denselben Klassifizierungsalgorithmus mit verschiedenen Teilmengen der Trainingsdatenmenge trainieren. Das bekannteste Beispiel für diesen Ansatz ist der Random-Forest-Algorithmus, der unterschiedliche Entscheidungsbaumklassifizierer miteinander kombiniert. Die folgende Abbildung illustriert das Konzept eines allgemeinen Ensembles mit Mehrheitsentscheidung:



Um die Klassenbezeichnung via absoluter oder relativer Mehrheit vorherzusagen, kombinieren wir die vorhergesagten Klassenbezeichnungen der einzelnen Klassifizierer  $C_j$  und wählen die Klassenbezeichnung  $\hat{y}$  mit den meisten Stimmen aus:

$$\hat{y} = \text{Modalwert} \{C_1(\mathbf{x}), C_2(\mathbf{x}), \dots, C_m(\mathbf{x})\}$$

Bei einer binären Klassifizierungsaufgabe, für die  $Klasse1 = -1$  und  $Klasse2 = +1$  gilt, können wir die Mehrheitsentscheidung folgendermaßen vorhersagen ( $\text{sign}$  ist die Vorzeichenfunktion):

$$C(\mathbf{x}) = \text{sign} \left[ \sum_j^m C_j(\mathbf{x}) \right] = \begin{cases} 1, & \text{wenn } \sum_i C_i(\mathbf{x}) \geq 0 \\ -1, & \text{andernfalls} \end{cases}$$

Um zu veranschaulichen, warum Ensemblemethoden besser funktionieren als einzelne Klassifizierer, wenden wir die einfachen Konzepte der Kombinatorik an. Im folgenden Beispiel nehmen wir an, dass alle  $n$  einfachen Klassifizierer einer binären Klassifizierungsaufgabe dieselbe Fehlerquote  $\varepsilon$  besitzen. Darüber hinaus nehmen wir an, dass die Klassifizierer unabhängig und die Fehlerquoten nicht korreliert sind. Unter diesen Voraussetzungen können wir die Fehlerwahrscheinlichkeit eines Ensembles einfacher Klassifizierer als Wahrscheinlichkeitsfunktion einer Binomialverteilung formulieren:

$$P(y \geq k) = \sum_k^n \binom{n}{k} \varepsilon^k (1-\varepsilon)^{n-k} = \varepsilon_{\text{Ensemble}}$$

Hier ist  $\binom{n}{k}$  der Binomialkoeffizient  $n$  über  $k$ . Anders formuliert: Wir berechnen die Wahrscheinlichkeit dafür, dass die Vorhersage des Ensembles falsch ist. Betrachten wir ein konkretes Beispiel mit 11 einfachen Klassifizierern ( $n = 11$ ) und einer Fehlerquote von 0.25 ( $\varepsilon = 0.25$ ).

$$P(y \geq k) = \sum_{k=6}^{11} \binom{11}{k} 0.25^k (1-\varepsilon)^{11-k} = 0.034$$

## Tipp

### Der Binomialkoeffizient

Der Binomialkoeffizient gibt die Anzahl der Möglichkeiten an, ungeordnete Teilmengen mit  $k$  Elementen aus einer Menge mit  $n$  Elementen auszuwählen. Aufgrund der Schreibweise spricht man auch von »n über k«. Da die Reihenfolge der Elemente keine Rolle spielt, ist der Binomialkoeffizient die Anzahl der Kombinationen ohne Wiederholung von  $k$  Elementen aus  $n$  Elementen.

Ausgeschrieben lautet die Formel  $\frac{n!}{(n-k)!k!}$ , wobei das Ausrufezeichen (!) für die Fakultätsfunktion steht, z.B.  $3! = 1 \times 2 \times 3 = 6$ .

Die Fehlerquote des Ensembles (0.034) ist viel niedriger als die der einzelnen Klassifizierer (0.25), sofern die genannten Voraussetzungen gegeben sind. Beachten Sie, dass in diesem vereinfachten Beispiel eine 50:50-Aufteilung durch eine gerade Anzahl Klassifizierer  $n$  als Fehler betrachtet wird, obwohl das nur in der Hälfte der Fälle zutrifft. Um solch ein idealisiertes Klassifizierer-Ensemble und einen einfachen Klassifizierer bei verschiedenen Grundfehlerquoten zu vergleichen, implementieren wir die Wahrscheinlichkeitsfunktion in Python:

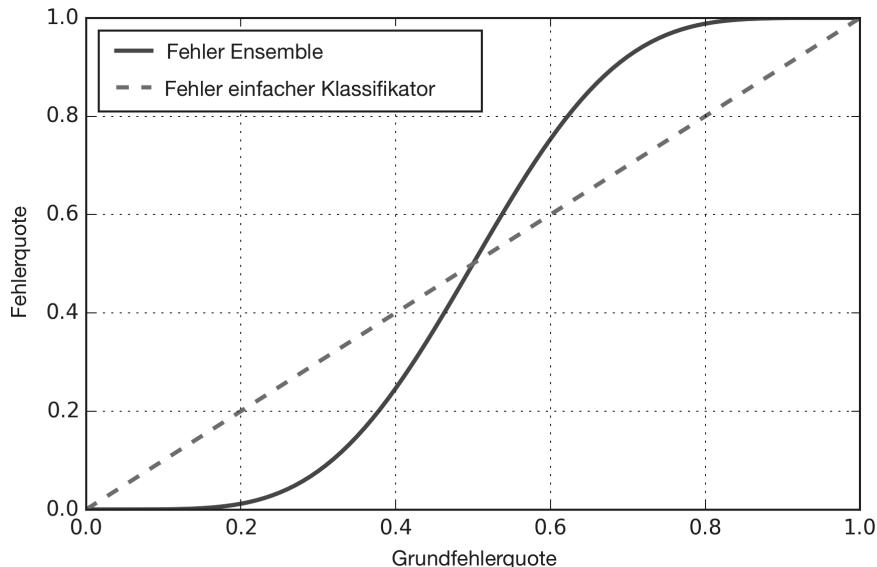
```
>>> from scipy.misc import comb
>>> import math
>>> def ensemble_error(n_classifier, error):
...     k_start = math.ceil(n_classifier / 2.0)
...     probs = [comb(n_classifier, k) *
...               error**k *
...               (1-error)**(n_classifier - k)
...     return sum(probs)
>>> ensemble_error(n_classifier=11, error=0.25)
0.034327507019042969
```

Nach der Implementierung der `ensemble_error`-Funktion können wir die Fehlerquoten für einen Bereich der Grundfehlerquoten von 0.0 bis 1.0 berechnen, um die Beziehung zwischen dem Fehler des einfachen Klassifizierers und dem Fehler des Ensembles als Diagramm zu visualisieren:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> error_range = np.arange(0.0, 1.01, 0.01)
>>> ens_errors= [ensemble_error(n_classifier=11, error=error)
...                 for error in error_range]
>>> plt.plot(error_range, ens_errors,
...             label='Fehler Ensemble',
...             linewidth=2)
>>> plt.plot(error_range, error_range,
...             linestyle='--',
...             label='Fehler einfacher Klassifizierer',
...             linewidth=2)
>>> plt.xlabel('Grundfehlerquote')
>>> plt.ylabel('Fehlerquote')
>>> plt.legend(loc='upper left')
```

```
>>> plt.grid()
>>> plt.show()
```

Dem Diagramm ist zu entnehmen, dass die Fehlerwahrscheinlichkeit eines Ensembles immer besser als die eines einfachen Klassifizierers ist, sofern der einfache Klassifizierer besser funktioniert als zufälliges Raten ( $\varepsilon < 0.5$ ). Beachten Sie hier, dass auf der Y-Achse sowohl die Fehlerquote des einfachen Klassifizierers (gestrichelte Linie) als auch die des Ensembles (durchgezogene Linie) aufgetragen ist.



## 7.2 Klassifizierer durch Mehrheitsentscheidung kombinieren

Nach dieser kurzen Einführung in das Ensemble Learning im vorangegangenen Abschnitt kommen wir nun zu einer kleinen Aufwärmübung und implementieren einen einfachen Mehrheitsentscheidungs-Klassifizierer in Python.

### 7.2.1 Implementierung eines einfachen Mehrheitsentscheidungs-Klassifizierers

Der zu implementierende Algorithmus soll es uns ermöglichen, verschiedene Klassifizierungsalgorithmen, denen verschiedene Gewichtungen der Zuverlässigkeit zugeordnet sind, miteinander zu kombinieren. Ziel ist es, einen Meta-Klassifizierer zu entwickeln, der die Schwächen der einzelnen Klassifizierer bei einer bestimmten Datenmenge ausgleicht. Als präzisere mathematische Beschreibung dafür formulieren wir die gewichtete Mehrheitsentscheidung folgendermaßen:

$$\hat{y} = \arg \max_i \sum_{j=1}^m w_j \chi_A(C_j(\mathbf{x}) = i)$$

Hier bezeichnet  $w_j$  die einem einfachen Klassifizierer  $C_j$  zugeordnete Gewichtung,  $\hat{y}$  die vom Ensemble vorhergesagte Klassenbezeichnung,  $\chi_A$  (der griechische Buchstabe Chi) die charakteristische Funktion  $[C_j(\mathbf{x}) = i \in A]$ , und  $A$  ist die Menge der eindeutigen Klassenbezeichnungen. Bei gleichen Gewichtungen kann diese Gleichung vereinfacht werden und wir formulieren sie so:

$$\hat{y} = \text{Modalwert}\{C_1(\mathbf{x}), C_2(\mathbf{x}), \dots, C_m(\mathbf{x})\}$$

### Tipp

In der Statistik bezeichnet der Modalwert das Element, das in einer Menge oder einer Stichprobe am häufigsten vorkommt, z.B.  $\text{Modalwert}\{1, 2, 1, 1, 2, 4, 5, 4\} = 1$ .

Um das Konzept der *Gewichtung* besser zu verstehen, sehen wir uns nun ein konkretes Beispiel an. Wir betrachten ein Ensemble mit drei einfachen Klassifizierern  $C_j (j \in \{1, 2, 3\})$  und möchten die Klassenbezeichnung einer gegebenen Instanz  $\mathbf{x}$  vorhersagen. Zwei der drei einfachen Klassifizierer prognostizieren die Klassenbezeichnung 0 und einer,  $C_3$ , sagt die Klassenbezeichnung 1 voraus. Wenn wir die einfachen Klassifizierer gleich gewichten, wird durch eine Mehrheitsentscheidung die Klassenbezeichnung 0 vorhergesagt:

$$C_1(\mathbf{x}) \rightarrow 0, C_2(\mathbf{x}) \rightarrow 0, C_3(\mathbf{x}) \rightarrow 1$$

$$\hat{y} = \text{Modalwert}\{0, 0, 1\} = 0$$

Nun weisen wir  $C_3$  eine Gewichtung von 0.6 und  $C_1$  sowie  $C_2$  eine Gewichtung von jeweils 0.2 zu.

$$\hat{y} = \arg \max_i \sum_{j=1}^m w_j \chi_A(C_j(\mathbf{x}) = i)$$

$$= \arg \max_i [0.2 \times i_0 + 0.2 \times i_0 + 0.6 \times i_1] = 1$$

Da  $3 \times 0.2 = 0.6$ , könnte man auch sagen, dass  $C_3$  drei Mal mehr Gewicht als  $C_1$  oder  $C_2$  besitzt. Das können wir folgendermaßen formulieren:

$$\hat{y} = \text{Modalwert}\{0, 0, 1, 1, 1\} = 1$$

Um das Konzept der gewichteten Mehrheitsentscheidung in Python-Code zu übersetzen, verwenden wir die `argmax`- und `bincount`-Funktionen von NumPy:

```
>>> import numpy as np
>>> np.argmax(np.bincount([0, 0, 1],
...                      weights=[0.2, 0.2, 0.6]))
1
```

Wie in Kapitel 3 erwähnt, können bestimmte Klassifizierer von scikit-learn mittels der `predict_proba`-Methode auch die Wahrscheinlichkeit der vorhergesagten Klassenbezeichnung zurückliefern. Bei der Mehrheitsentscheidung statt der Klassenbezeichnungen die vorhergesagten Klassenwahrscheinlichkeiten zu verwenden, erweist sich als nützlich, wenn die Klassifizierer des Ensembles gut kalibriert sind. Die modifizierte Version der Mehrheitsentscheidung zur Vorhersage der Klassenbezeichnungen anhand der Wahrscheinlichkeiten kann folgendermaßen formuliert werden:

$$\hat{y} = \arg \max_i \sum_{j=1}^m w_j p_{ij}$$

Hier ist  $p_{ij}$  die vorhergesagte Wahrscheinlichkeit des  $j$ -ten Klassifizierers für die Klassenbezeichnung  $i$ .

Wir fahren mit dem Beispiel fort und nehmen an, dass es sich um eine binäre Klassifizierungsaufgabe mit den Klassenbezeichnungen  $i \in \{0,1\}$  und ein Ensemble aus drei Klassifizierern  $C_j$  ( $j \in \{1,2,3\}$ ) handelt. Außerdem nehmen wir an, dass der Klassifizierer  $C_j$  die folgenden Wahrscheinlichkeiten für die Klassenzugehörigkeit einer bestimmten Instanz  $\mathbf{x}$  zurückgibt:

$$C_1(\mathbf{x}) \rightarrow [0.9, 0.1], C_2(\mathbf{x}) \rightarrow [0.8, 0.2], C_3(\mathbf{x}) \rightarrow [0.4, 0.6]$$

Die einzelnen Wahrscheinlichkeiten errechnen sich dann folgendermaßen:

$$p(i_0 | \mathbf{x}) = 0.2 \times 0.9 + 0.2 \times 0.8 + 0.6 \times 0.4 = 0.58$$

$$p(i_1 | \mathbf{x}) = 0.2 \times 0.1 + 0.2 \times 0.2 + 0.6 \times 0.06 = 0.42$$

$$\hat{y} = \arg \max_i [p(i_0 | \mathbf{x}), p(i_1 | \mathbf{x})] = 0$$

Zur Implementierung der auf Wahrscheinlichkeiten beruhenden gewichteten Mehrheitsentscheidung verwenden wir die NumPy-Funktionen `np.average` und `np.argmax`:

## Kapitel 7

### Kombination verschiedener Modelle für das Ensemble Learning

```
>>> ex = np.array([[0.9, 0.1],  
...                 [0.8, 0.2],  
...                 [0.4, 0.6]])  
>>> p = np.average(ex, axis=0, weights=[0.2, 0.2, 0.6])  
>>> p  
array([ 0.58, 0.42])  
>>> np.argmax(p)  
0
```

Nun können wir einen gewichteten Mehrheitsentscheidungs-Klassifizierer `MajorityVoteClassifier` in Python implementieren:

```
from sklearn.base import BaseEstimator  
from sklearn.base import ClassifierMixin  
from sklearn.preprocessing import LabelEncoder  
from sklearn.externals import six  
from sklearn.base import clone  
from sklearn.pipeline import _name_estimators  
import numpy as np  
import operator  
  
class MajorityVoteClassifier(BaseEstimator,  
                           ClassifierMixin):  
    """ Ein Mehrheitsentscheidungs-Klassifizierer  
        als Ensemble  
  
    Parameter  
    -----  
    classifiers : array-like, shape = [n_classifiers]  
        Verschiedene Klassifizierer des Ensembles  
  
    vote : str, {'classlabel', 'probability'}  
        Standard: 'classlabel'  
        Falls 'classlabel', beruht die Vorhersage auf dem  
        argmax-Wert der Klassenbezeichnungen.  
        Falls 'probability', wird der argmax-Wert der Summe der  
        Wahrscheinlichkeiten zur Vorhersage der Klassen-  
        bezeichnung verwendet (bei kalibrierten Klassifizierern  
        empfehlenswert).  
  
    weights : array-like, shape = [n_classifiers]  
        Optional, Standard: Keine  
        Falls eine Liste mit 'int'- oder 'float'-Werten  
        übergeben wird, werden die Klassifizierer ihrer  
        Bedeutung nach gewichtet. Falls `weights=None` ,
```

```

werden alle gleich gewichtet.

"""

def __init__(self, classifiers,
             vote='classlabel', weights=None):
    self.classifiers = classifiers
    self.named_classifiers = {key: value for
                               key, value in
                               _name_estimators(classifiers)}
    self.vote = vote
    self.weights = weights

def fit(self, X, y):
    """Klassifizierer anpassen

    Parameter
    -----
    X : {Array-artige, dünnbesetzte Matrix},
        shape = [n_samples, n_features]
        Matrix der Trainingsobjekte
    y : Array-artig, shape = [n_samples]
        Vektor der Zielklassenbezeichnungen

    Rückgabewert
    -----
    self : object

    """
    # LabelEncoder verwenden, um zu gewährleisten, dass
    # die Klassenbezeichnungen bei 0 beginnen; wichtig
    # für den np.argmax-Aufruf in self.predict
    self.lablenc_ = LabelEncoder()
    self.lablenc_.fit(y)
    self.classes_ = self.lablenc_.classes_
    self.classifiers_ = []
    for clf in self.classifiers:
        fitted_clf = clone(clf).fit(X,
                                    self.lablenc_.transform(y))
        self.classifiers_.append(fitted_clf)
    return self

```

Ich habe dem Code viele Kommentare hinzugefügt, um die einzelnen Teile zu erläutern. Bevor wir die verbleibenden Methoden implementieren, sollten wir an dieser Stelle kurz darauf eingehen, weil er auf den ersten Blick verwirrend erscheint. Die Elternklassen `BaseEstimator` und `ClassifierMixin` stellen bereits grundlegende Funktionalität bereit, unter anderem die Methoden `get_params` und `set_params` zum Einstellen und Abrufen der Parameter des Klassifi-

zierers sowie die `score`-Methode zur Berechnung der Korrektklassifizierungsrate. Beachten Sie auch, dass `six` importiert wird, damit der Mehrheitsentscheidungs-Klassifizierer `MajorityVoteClassifier` mit Python 2.6 kompatibel ist.

Nun fügen wir die `predict`-Methode zur Vorhersage der Klassenbezeichnung anhand der Mehrheitsentscheidung hinzu und initialisieren ein neues `MajorityVoteClassifier`-Objekt mit `vote='classlabel'`. Alternativ können wir das Klassifizierer-Ensemble auch mit `vote='probability'` initialisieren, um die Klassenbezeichnung anhand der Klassenzugehörigkeitswahrscheinlichkeiten zu prognostizieren. Außerdem fügen wir eine `predict_proba`-Methode hinzu, die die durchschnittlichen Wahrscheinlichkeiten zurückgibt, was sich bei der Berechnung der Fläche unter der ROC-Kurve (*ROC AUC, Receiver Operator Characteristic Area Under Curve*) als nützlich erweist.

```
def predict(self, X):
    """ Klassenbezeichnung für X vorhersagen.

    Parameter
    -----
    X : {Array-artige, dünnbesetzte Matrix},
        Shape = [n_samples, n_features]
        Matrix der Trainingsobjekte

    Rückgabewert
    -----
    maj_vote : Array-artig, shape = [n_samples]
        Vorhergesagte Klassenbezeichnungen
    """

    if self.vote == 'probability':
        maj_vote = np.argmax(self.predict_proba(X),
                             axis=1)
    else: # vote ist 'classlabel'

        # Resultate der clf.predict-Aufrufe ermitteln
        predictions = np.asarray([clf.predict(X)
                                  for clf in
                                  self.classifiers_]).T

        maj_vote = np.apply_along_axis(
            lambda x:
            np.argmax(np.bincount(x,
                                  weights=self.weights)),
            axis=1,
            arr=predictions)
    maj_vote = self.lablenc_.inverse_transform(maj_vote)
```

```

    return maj_vote

def predict_proba(self, X):
    """ Klassenwahrscheinlichkeiten für X vorhersagen

Parameter
-----
X : {Array-artige, dünnbesetzte Matrix},
    shape = [n_samples, n_features]
    Trainingsvektoren; n_samples ist die Anzahl der
    Objekte und n_features die Anzahl der Merkmale

Rückgabewert
-----
avg_proba : Array-artig,
    shape = [n_samples, n_classes]
    Gewichtete durchschnittliche Wahrscheinlichkeit
    für jede Klasse pro Objekt
"""
probas = np.asarray([clf.predict_proba(X)
                     for clf in self.classifiers_])
avg_proba = np.average(probas,
                       axis=0, weights=self.weights)
return avg_proba

def get_params(self, deep=True):
    """ Klassifizierer-Parameternamen für Rastersuche"""
    if not deep:
        return super(MajorityVoteClassifier,
                    self).get_params(deep=False)
    else:
        out = self.named_classifiers_.copy()
        for name, step in \
            six.iteritems(self.named_classifiers_):
            for key, value in six.iteritems(
                step.get_params(deep=True)):
                out['%s__%s' % (name, key)] = value
        return out

```

Beachten Sie außerdem, dass wir hier eine eigene, abgewandelte Version der `get_params`-Methode definieren, die für den Zugriff auf die Parameter der verschiedenen Klassifizierer im Ensemble die `_name_estimators`-Funktion verwendet. Auf den ersten Blick wirkt das Ganze ziemlich kompliziert, aber wenn wir in den nachfolgenden Abschnitten bei der Hyperparameter-Abstimmung eine Rastersuche verwenden, ergibt alles einen Sinn.

**Tipp**

Die `MajorityVoteClassifier`-Implementierung ist zwar für Demonstrationszwecke durchaus nützlich, wir haben allerdings eine erweiterte Version des Mehrheitsentscheidungs-Klassifizierers in scikit-learn implementiert, die auf der Implementierung in der ersten Auflage dieses Buches beruht. Sie ist seit der scikit-learn-Version 0.17 als `sklearn.ensemble.VotingClassifier` verfügbar.

### 7.2.2 Vorhersagen nach dem Prinzip der Mehrheitsentscheidung treffen

Nun ist es an der Zeit, den im vorangegangenen Abschnitt implementierten `MajorityVoteClassifier` auszuprobieren. Zunächst bereiten wir jedoch eine Datenmenge vor, mit der wir ihn testen können. Da wir mit dem Laden von Datensätzen aus CSV-Dateien bereits vertraut sind, nehmen wir eine Abkürzung und verwenden die Iris-Datensammlung von scikit-learns `datasets`-Modul. Wir werden nur zwei der Merkmale benutzen, nämlich die Breite des Kelchblatts und die Länge des Blütenblatts, um die Klassifizierung etwas zu erschweren. Der `MajorityVoteClassifier` lässt sich zwar für Aufgabenstellungen mit mehr als zwei Klassen verallgemeinern, wir werden jedoch zur späteren Berechnung der ROC AUC nur zwei der Arten klassifizieren, *Iris versicolor* und *Iris virginica*. Hier der Code:

```
>>> from sklearn import datasets
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.preprocessing import LabelEncoder
>>> iris = datasets.load_iris()
>>> X, y = iris.data[50:, [1, 2]], iris.target[50:]
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
```

**Tipp**

Beachten Sie, dass scikit-learn für die Berechnung des ROC-AUC-Scores die `predict_proba`-Methode nutzt (sofern anwendbar). In Kapitel 3 haben Sie erfahren, wie die Klassenwahrscheinlichkeiten mit logistischen Regressionsmodellen berechnet werden. Bei Entscheidungsbäumen werden die Wahrscheinlichkeiten der verschiedenen Knoten anhand eines beim Training erstellten Häufigkeitsvektors berechnet. Dieser Vektor speichert die Häufigkeitswerte der Klassenbezeichnungen, die wiederum anhand der Verteilung der Klassenbezeichnungen beim jeweiligen Knoten errechnet werden.

Dann werden die Häufigkeiten so normiert, dass die Summe 1 ergibt. Auf ähnliche Weise sind beim k-nächste-Nachbarn-Algorithmus die Klassenbezeichnungshäufigkeiten der k-nächsten Nachbarn normiert. Die vom Entscheidungsbaum und vom k-nächste-Nachbarn-Klassifizierer gelieferten normierten Wahrscheinlichkeiten ähneln zwar denjenigen eines logistischen Regressionsmodells, man darf jedoch nicht vergessen, dass sie tatsächlich nicht aus einer Wahrscheinlichkeitsfunktion hergeleitet wurden.

Nun teilen wir die Iris-Datensammlung in 50 Prozent Trainings- und 50 Prozent Testdaten auf:

```
>>> X_train, X_test, y_train, y_test =\
...     train_test_split(X, y,
...                     test_size=0.5,
...                     random_state=1,
...                     stratify=y)
```

Mit der Trainingsdatenmenge können wir nun die drei verschiedenen Klassifizierer trainieren:

- Klassifizierung durch eine logistische Regression
- Entscheidungsbaumklassifizierer
- k-nächste-Nachbarn-Klassifizierer

Anschließend betrachten wir die einzelnen Ergebnisse bei der Trainingsdatenmenge anhand einer 10-fachen Kreuzvalidierung, bevor wir die drei Klassifizierer zu einem Ensemble-Klassifizierer kombinieren.

```
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.tree import DecisionTreeClassifier
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.pipeline import Pipeline
>>> import numpy as np
>>> clf1 = LogisticRegression(penalty='l2',
...                             C=0.001,
...                             random_state=1)
>>> clf2 = DecisionTreeClassifier(max_depth=1,
...                                 criterion='entropy',
...                                 random_state=0)
>>> clf3 = KNeighborsClassifier(n_neighbors=1,
...                             p=2,
...                             metric='minkowski')
>>> pipe1 = Pipeline([('sc', StandardScaler()),
```

```

...
['clf', clf1]])
>>> pipe3 = Pipeline([('sc', StandardScaler()),
...                   ['clf', clf3]])
>>> clf_labels = ['Logistische Regression',
...                  'Entscheidungsbaum', 'KNN']
>>> print('10-fache Kreuzvalidierung:\n')
>>> for clf, label in zip([pipe1, clf2, pipe3], clf_labels):
...     scores = cross_val_score(estimator=clf,
...                               X=X_train,
...                               y=y_train,
...                               cv=10,
...                               scoring='roc_auc')
...     print("ROC AUC: %.2f (+/- %.2f) [%s]"
...           % (scores.mean(), scores.std(), label))

```

Die Ausgabe zeigt, dass die Vorhersagekraft der verschiedenen Klassifizierer nahezu gleich ist:

#### 10-fache Kreuzvalidierung:

```

ROC AUC: 0.87 (+/- 0.17) [Logistische Regression]
ROC AUC: 0.89 (+/- 0.16) [Entscheidungsbaum]
ROC AUC: 0.88 (+/- 0.15) [KNN]

```

Sie werden sich vielleicht fragen, warum wir die logistische Regression und den k-nächste-Nachbarn-Klassifizierer als Teil einer Pipeline trainieren. Der Grund dafür ist, wie in Kapitel 3 erörtert, dass die logistische Regression und der k-nächste-Nachbarn-Algorithmus (mit euklidischem Abstandsmaß) im Gegensatz zum Entscheidungsbaum nicht skaleninvariant sind. Die Merkmale der Blumen sind zwar alle in derselben Maßeinheit (Zentimeter) angegeben, dennoch ist es guter Stil, standardisierte Merkmale zu verwenden.

Sehen wir uns nun den interessanteren Teil an und kombinieren die verschiedenen Mehrheitsentscheidungs-Klassifizierer zum `MajorityVoteClassifier`:

```

>>> mv_clf = MajorityVoteClassifier(
...                 classifiers=[pipe1, clf2, pipe3])
>>> clf_labels += ['Mehrheitsentscheidung']
>>> all_clf = [pipe1, clf2, pipe3, mv_clf]
>>> for clf, label in zip(all_clf, clf_labels):
...     scores = cross_val_score(estimator=clf,
...                               X=X_train,
...                               y=y_train,
...                               cv=10,
...                               scoring='roc_auc')

```

```
....     print("Korrektklassifizierungsrate: %0.2f \
...             (+/- %0.2f) [%s]" % (scores.mean(),
...                                     scores.std(), label))
ROC AUC: 0.87 (+/- 0.17) [Logistische Regression]
ROC AUC: 0.89 (+/- 0.16) [Entscheidungsbaum]
ROC AUC: 0.88 (+/- 0.15) [KNN]
ROC AUC: 0.94 (+/- 0.13) [Mehrheitsentscheidung]
```

Wie Sie sehen, hat sich die Leistung des Mehrheitsentscheidungs-Klassifizierers gegenüber der 10-fachen Kreuzvalidierung der einzelnen Klassifizierer deutlich verbessert.

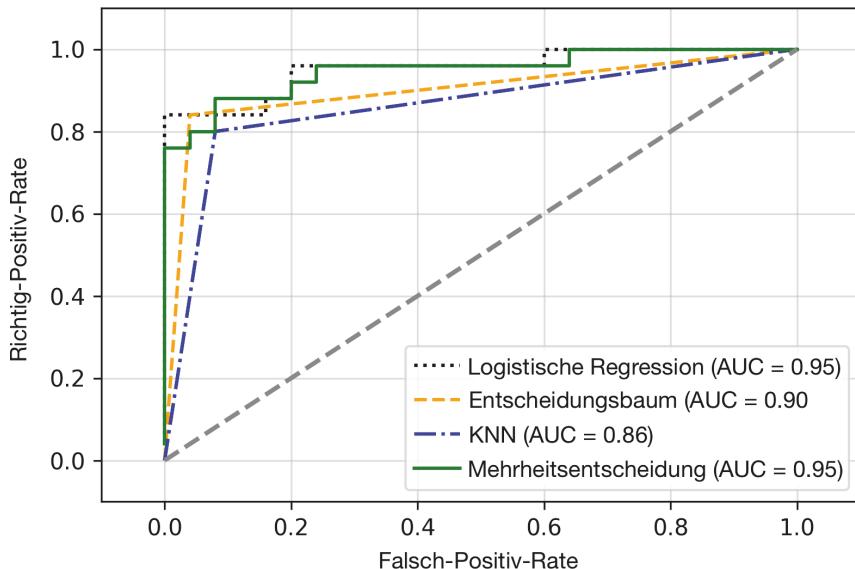
## 7.3 Bewertung und Abstimmung des Klassifizierer-Ensembles

In diesem Abschnitt werden wir die ROC-Kurven der Testdatenmenge berechnen, um zu überprüfen, ob der `MajorityVoteClassifier` gut mit unbekannten Daten zurechtkommt. Wir sollten aber daran denken, dass die Testdatenmenge nicht zur Modellauswahl verwendet wird – ihr einziger Zweck besteht darin, eine unverzerrte Einschätzung der Verallgemeinerungsfähigkeit eines Klassifizierungssystems zu ermöglichen. Hier der Code:

```
>>> from sklearn.metrics import roc_curve
>>> from sklearn.metrics import auc
>>> colors = ['black', 'orange', 'blue', 'green']
>>> linestyles = [':', '--', '-.', '-']
>>> for clf, label, clr, ls \
...         in zip(all_clf, clf_labels, colors, linestyles):
...     # Die Klassenbezeichnung der positiven Klasse sei 1
...     y_pred = clf.fit(X_train,
...                       y_train).predict_proba(X_test)[:, 1]
...     fpr, tpr, thresholds = roc_curve(y_true=y_test,
...                                      y_score=y_pred)
...     roc_auc = auc(x=fpr, y=tpr)
...     plt.plot(fpr, tpr,
...               color=clr,
...               linestyle=ls,
...               label='%s (auc = %0.2f)' % (label, roc_auc))
>>> plt.legend(loc='lower right')
>>> plt.plot([0, 1], [0, 1],
...           linestyle='--',
...           color='gray',
...           linewidth=2)
>>> plt.xlim([-0.1, 1.1])
```

```
>>> plt.ylim([-0.1, 1.1])
>>> plt.grid(alpha=0.5)
>>> plt.xlabel('Falsch-Positiv-Rate')
>>> plt.ylabel('Richtig-Positiv-Rate')
>>> plt.show()
```

Wie der ROC zu entnehmen ist, funktioniert das Klassifizierer-Ensemble auch bei der Testdatenmenge gut (ROC AUC = 0.95). Die Klassifizierung durch logistische Regression funktioniert bei derselben Datenmenge ähnlich gut, vermutlich aufgrund der in Anbetracht der kleinen Datenmenge hohen Varianz (die von der Art der Aufteilung der Datenmenge abhängt).



Da wir nur zwei Merkmale zur Klassifizierung ausgewählt haben, wäre es interessant zu wissen, wie die Entscheidungsbereiche eines Klassifizierer-Ensembles eigentlich aussehen. Es ist zwar nicht notwendig, die Trainingsmerkmale vor der Modellanpassung zu standardisieren, da die logistische Regression und der k-nächste-Nachbarn-Algorithmus das automatisch berücksichtigen, wir standardisieren die Trainingsdatenmenge aber dennoch, damit der Entscheidungsbereich des Entscheidungsbaums bei der Visualisierung von gleicher Größenordnung ist. Hier der Code:

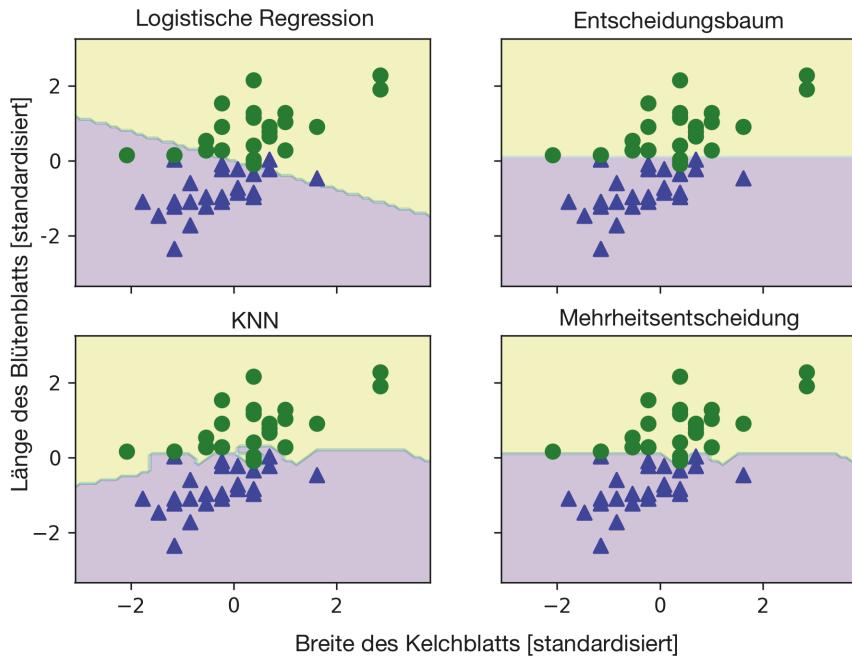
```
>>> sc = StandardScaler()
>>> X_train_std = sc.fit_transform(X_train)
>>> from itertools import product
>>> x_min = X_train_std[:, 0].min() - 1
>>> x_max = X_train_std[:, 0].max() + 1
```

```

>>> y_min = X_train_std[:, 1].min() - 1
>>> y_max = X_train_std[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                         np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(nrows=2, ncols=2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(7, 5))
>>> for idx, clf, tt in zip(product([0, 1], [0, 1]),
...                           all_clf, clf_labels):
...     clf.fit(X_train_std, y_train)
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx[0], idx[1]].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx[0], idx[1]].scatter(X_train_std[y_train==0,
...                                               0],
...                                   X_train_std[y_train==0, 1],
...                                   c='blue',
...                                   marker='^',
...                                   s=50)
...
...     axarr[idx[0], idx[1]].scatter(X_train_std[y_train==1,
...                                               0],
...                                   X_train_std[y_train==1, 1],
...                                   c='green',
...                                   marker='o',
...                                   s=50)
...     axarr[idx[0], idx[1]].set_title(tt)
>>> plt.text(-3.5, -4.5,
...             s='Breite des Kelchblatts[standardisiert]',
...             ha='center', va='center', fontsize=12)
>>> plt.text(-10.5, 4.5,
...             s='Länge des Blütenblatts [standardisiert]',
...             ha='center', va='center',
...             fontsize=12, rotation=90)
>>> plt.show()

```

Interessanterweise – aber auch erwartungsgemäß – scheint der Entscheidungsbereich des Klassifizierer-Ensembles ein Hybride aus den Entscheidungsbereichen der einzelnen Klassifizierer zu sein. Auf den ersten Blick hat die Entscheidungsgrenze der Mehrheitsentscheidung, die bei einer Kelchblattbreite von mehr als 1 cm senkrecht zur y-Achse verläuft, große Ähnlichkeit mit derjenigen des Entscheidungsbaums. Zudem ist auch erkennbar, dass die Nichtlinearität des k-nächste-Nachbarn-Klassifizierers eine Rolle spielt.



Bevor Sie erfahren, wie sich die einzelnen Klassifizierungsparameter eines Klassifizierer-Ensembles abstimmen lassen, soll die `get_params`-Methode aufgerufen werden, damit Sie einen Eindruck davon gewinnen, wie man auf die Parameter eines `GridSearch`-Objekts zugreifen kann:

```
>>> mv_clf.get_params()
{'decisiontreeclassifier': DecisionTreeClassifier(
    class_weight=None, criterion='entropy', max_depth=1,
    max_features=None, max_leaf_nodes=None,
    min_samples_leaf=1, min_samples_split=2,
    min_weight_fraction_leaf=0.0, random_state=0,
    splitter='best'), 'decisiontreeclassifier__class_weight':
None, 'decisiontreeclassifier__criterion': 'entropy',
[...]
'decisiontreeclassifier__random_state': 0,
'decisiontreeclassifier__splitter': 'best',
'pipeline-1': Pipeline(steps=[('sc',
    StandardScaler(copy=True, with_mean=True, with_std=True)),
    ('clf', LogisticRegression(C=0.001, class_weight=None,
    dual=False, fit_intercept=True, intercept_scaling=1,
    max_iter=100, multi_class='ovr', penalty='l2',
```

```

random_state=0, solver='liblinear', tol=0.0001,
verbose=0))]], 'pipeline-1_clf':
LogisticRegression(C=0.001, class_weight=None, dual=False,
fit_intercept=True, intercept_scaling=1, max_iter=100,
multi_class='ovr', penalty='l2', random_state=0,
solver='liblinear', tol=0.0001, verbose=0),
'pipeline-1_clf_C': 0.001,
'pipeline-1_clf_class_weight': None,
'pipeline-1_clf_dual': False,

[...]

'pipeline-1_sc_with_std': True, 'pipeline-2':
Pipeline(steps=[('sc', StandardScaler(copy=True,
with_mean=True, with_std=True)), ('clf',
KNeighborsClassifier(algorithm='auto', leaf_size=30,
metric='minkowski', metric_params=None, n_neighbors=1, p=2,
weights='uniform'))]), 'pipeline-2_clf':
KNeighborsClassifier(algorithm='auto', leaf_size=30,
metric='minkowski', metric_params=None, n_neighbors=1, p=2,
weights='uniform'), 'pipeline-2_clf_algorithm': 'auto',

[...]

'pipeline-2_sc_with_std': True}

```

Dank der von der `get_params`-Methode zurückgelieferten Werte wissen wir jetzt, wie auf die einzelnen Klassifizierer-Attribute zugegriffen werden kann. Nun sollen zur Illustration der inverse Regularisierungsparameter  $C$  einer logistischen Regression und mittels Rastersuche die Tiefe eines Entscheidungsbaums variiert werden. Hier der Code:

```

>>> from sklearn.model_selection import GridSearchCV
>>> params = {'decisiontreeclassifier__max_depth': [1, 2],
...             'pipeline-1_clf_C': [0.001, 0.1, 100.0]}
>>> grid = GridSearchCV(estimator=mv_clf,
...                       param_grid=params,
...                       cv=10,
...                       scoring='roc_auc')
>>> grid.fit(X_train, y_train)

```

Nach Abschluss der Rastersuche können wir die per 10-facher Kreuzvalidierung berechneten verschiedenen Wertekombinationen der Hyperparameter und die durchschnittlichen ROC-AUC-Scores ausgeben. Dazu dient folgender Code:

```
>>> for params, mean_score, scores in grid.grid_scores_:
...     print("%0.3f+/-%0.2f %r"
...           % (mean_score, scores.std() / 2, params))
0.933 +/- 0.07 {'pipeline-1__clf__C': 0.001,
'decisiontreeclassifier__max_depth': 1}
0.947 +/- 0.07 {'pipeline-1__clf__C': 0.1,
'decisiontreeclassifier__max_depth': 1}
0.973 +/- 0.04 {'pipeline-1__clf__C': 100.0,
'decisiontreeclassifier__max_depth': 1}
0.947 +/- 0.07 {'pipeline-1__clf__C': 0.001,
'decisiontreeclassifier__max_depth': 2}
0.947 +/- 0.07 {'pipeline-1__clf__C': 0.1,
'decisiontreeclassifier__max_depth': 2}
0.973 +/- 0.04 {'pipeline-1__clf__C': 100.0,
'decisiontreeclassifier__max_depth': 2}
>>> print('Beste Parameter: %s' % grid.best_params_)
Beste Parameter: {'pipeline-1__clf__C': 100.0, 'decisiontreeclassifier__':
max_depth': 1}
>>> print('Korrektklassifizierungsrate: %.2f' % grid.best_score_)
Korrektklassifizierungsrate: 0.97
```

Wie Sie sehen, erhalten wir bei der Kreuzvalidierung die besten Resultate, wenn wir eine geringere Regularisierungsstärke ( $C=100.0$ ) wählen. Die Tiefe des Baums hingegen scheint die Leistung gar nicht zu beeinflussen, was nahelegt, dass ein Entscheidungsbaum mit nur einer Ebene zur Klassifizierung der Daten ausreicht. Da man dieselben Testdaten nicht mehr als ein Mal zur Modellbewertung verwenden sollte, verzichten wir an dieser Stelle darauf, die Verallgemeinerungsfähigkeit der abgestimmten Hyperparameter abzuschätzen und kommen stattdessen gleich zu einem alternativen Ansatz für das Ensemble Learning, dem sogenannten *Bagging*.

### Tipp

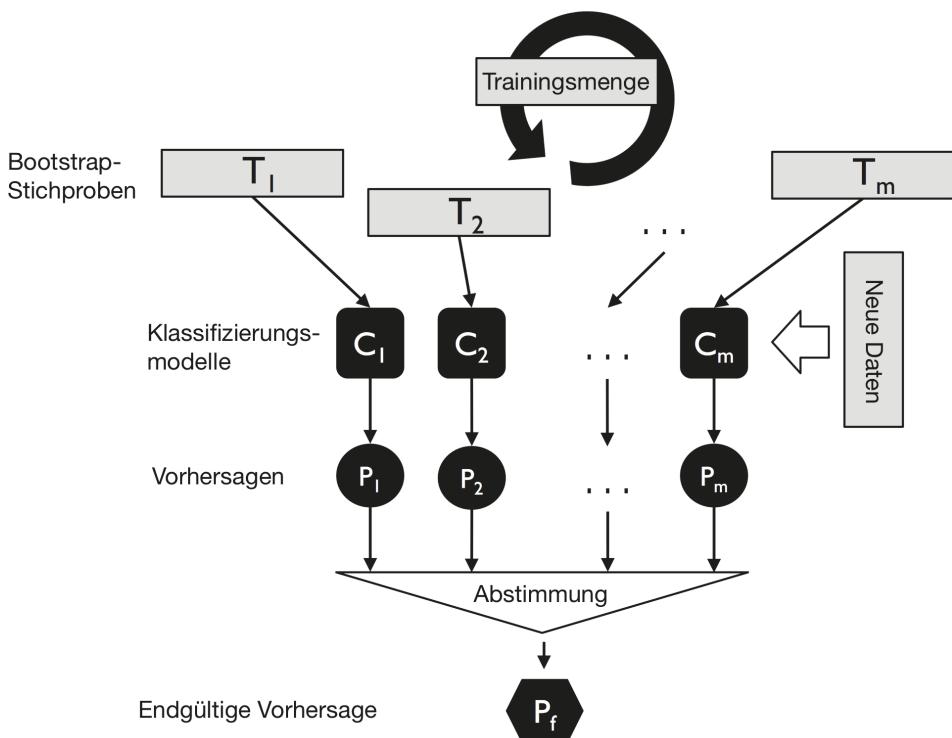
Die in diesem Abschnitt implementierte Methode einer Mehrheitsentscheidung sollte nicht mit dem als *Stacking* bezeichneten Verfahren verwechselt werden. Den Stacking-Algorithmus kann man sich als ein zweischichtiges Ensemble vorstellen, bei dem die erste Schicht aus verschiedenen Klassifizierern besteht, die ihre Vorhersagen an die zweite Schicht übergeben, in der ein weiterer Klassifizierer (typischerweise eine logistische Regression) die Vorhersagen der einzelnen Klassifizierer der ersten Schicht als Eingabe benutzt, um die endgültige Vorhersage zu treffen. Diese Vorgehensweise hat David H. Wolpert ausführlich beschrieben (D.H. Wolpert, *Stacked Generalization*, Neural Networks, 5(2):241-259, 1992).

Leider ist dieser Algorithmus derzeit noch nicht in scikit-learn implementiert; die Implementierung ist jedoch in Arbeit. Unter [http://rasbt.github.io/mlxtend/user\\_guide/classifier/StackingClassifier/](http://rasbt.github.io/mlxtend/user_guide/classifier/StackingClassifier/) und [http://rasbt.github.io/mlxtend/user\\_guide/classifier/StackingCVClassifier/](http://rasbt.github.io/mlxtend/user_guide/classifier/StackingCVClassifier/) sind scikit-learn-kompatible Stacking-Implementierung zu finden.

## 7.4 Bagging: Klassifizierer-Ensembles anhand von Bootstrap-Stichproben entwickeln

Das *Bagging* (von engl. *Bootstrap aggregating*) ist eine Methode des Ensemble Learnings, die eng mit dem im vorangegangenen Abschnitt implementierten Mehrheitsentscheidungs-Klassifizierer **MajorityVoteClassifier** verwandt ist. Anstatt dieselbe Trainingsdatenmenge für die Anpassung der einzelnen Klassifizierer zu verwenden, entnehmen wir der ursprünglichen Trainingsdatenmenge wiederholt Bootstrap-Stichproben (zufälliges Ziehen mit Ersetzung, daher auch die englische Bezeichnung).

Die folgende Abbildung fasst das Konzept des Baggings zusammen.



In den folgenden Abschnitten betrachten wir ein einfaches Beispiel für das Bagging von Hand und verwenden scikit-learn zur Klassifizierung der Wein-Datensammlung.

### 7.4.1 Bagging kurz zusammengefasst

Zum besseren Verständnis der Funktionsweise betrachten wir das konkrete Beispiel in der folgenden Abbildung. Hier gibt es sieben verschiedene Instanzen der Trainingsdaten (gekennzeichnet durch die Indizes 1 bis 7), die in jeder Bagging-Runde zufällig gezogen werden (Ziehen mit Ersetzung). Diese Bootstrap-Stichproben werden dann verwendet, um einen Klassifizierer  $C_j$  anzupassen, bei dem es sich typischerweise um einen (nicht gestützten) Entscheidungsbaum handelt.

Stichproben-index	Bagging-Runde 1	Bagging-Runde 2	...
1	2	7	...
2	2	3	...
3	1	2	...
4	3	1	...
5	7	1	...
6	2	7	...
7	4	7	...

The diagram shows a table with 7 rows (instances) and 4 columns (Bagging-Runde 1, Bagging-Runde 2, and two ellipsis columns). Below the table, three brackets group the columns for each bagging round. Arrows point from these groups to labels  $C_1$ ,  $C_2$ , and  $C_m$  respectively, indicating that each round's bootstrap sample is used to train a separate classifier.

Wie Sie der Abbildung entnehmen können, wird jedem Klassifizierer eine zufällige Teilmenge der Trainingsdatenmenge zugewiesen. Jede Teilmenge enthält einen bestimmten Anteil Duplikate, und einige der ursprünglichen Daten sind in keiner Teilmenge enthalten, weil in jeder Runde sämtliche Objekte zur Auswahl stehen (Ziehen mit Ersetzung). Nach der Anpassung der Klassifizierer an die Bootstrap-Stichproben werden die einzelnen Vorhersagen durch eine Mehrheitsentscheidung miteinander kombiniert.

Das Bagging ist mit dem in Kapitel 3 vorgestellten Random-Forest-Klassifizierer verwandt. Tatsächlich sind Random Forests ein Spezialfall des Baggings, bei dem wir ebenfalls zufällige Merkmalsteilmengen zur Anpassung der einzelnen Entscheidungsbäume verwenden.

**Tipp**

Bagging wurde erstmals 1994 von Leo Breiman in einem technischen Bericht vorgeschlagen. Er konnte außerdem zeigen, dass das Bagging-Verfahren die Korrektklassifizierungsrate instabiler Modelle verbessern und das Ausmaß der Überanpassung senken kann. Wenn Sie mehr zu diesem Thema erfahren möchten, kann ich die Lektüre seines Forschungsberichts nur wärmstens empfehlen (L. Breiman, *Bagging Predictors*, Machine Learning, 24(2):123-140, 1996, kostenlos online verfügbar).

### 7.4.2 Klassifizierung der Wein-Datensammlung durch Bagging

Um das Bagging in Aktion zu sehen, erstellen wir nun mit der in Kapitel 4 vorgestellten Wein-Datensammlung eine etwas komplexere Klassifizierungsaufgabe. Wir betrachten hier die Weinklassen 2 und 3 und wählen die beiden Merkmale *Alkohol* und *OD280/OD315 des verdünnten Weins* aus.

```
>>> import pandas as pd
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/'
    'machine-learning-databases/wine/wine.data',
    header=None)
>>> df_wine.columns = ['Klassenbezeichnung', 'Alkohol',
...                     'Apfelsäure', 'Asche',
...                     'Aschealkalität',
...                     'Magnesium', 'Phenole insgesamt',
...                     'Flavanoide',
...                     'Nichtflavanoide Phenols',
...                     'Tannin',
...                     'Farbintesität', 'Farbe',
...                     'OD280/OD315 des verdünnten Weins',
...                     'Prolin']
>>> # Eine Klasse entfernen
>>> df_wine = df_wine[df_wine['Klassenbezeichnung'] != 1]
>>> y = df_wine['Klassenbezeichnung'].values
>>> X = df_wine[['Alkohol',
...               'OD280/OD315 des verdünnten Weins']].values
```

Als Nächstes wandeln wir die Klassenbezeichnungen ins Binärformat und teilen die Daten in 80 Prozent Trainings- und 20 Prozent Testdaten auf:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> from sklearn.model_selection import train_test_split
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
```

```
>>> X_train, X_test, y_train, y_test = \
...         train_test_split(X, y,
...                         test_size=0.20,
...                         random_state=1,
...                         stratify=y)
```

In scikit-learn ist bereits ein `BaggingClassifier`-Algorithmus implementiert, den wir vom `ensemble`-Modul importieren können. Hier verwenden wir einen nicht gestützten Entscheidungsbaum als einfachen Klassifizierer und erstellen ein Ensemble von 500 Entscheidungsbäumen, die an die verschiedenen Bootstrap-Stichproben der Trainingsdatenmenge angepasst werden:

```
>>> from sklearn.ensemble import BaggingClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                                 max_depth=None,
...                                 random_state=1)
>>> bag = BaggingClassifier(base_estimator=tree,
...                           n_estimators=500,
...                           max_samples=1.0,
...                           max_features=1.0,
...                           bootstrap=True,
...                           bootstrap_features=False,
...                           n_jobs=1,
...                           random_state=1)
```

Nun berechnen wir den Korrektklassifizierungsraten-Score der Vorhersagen für die Trainings- und Testdatenmenge, um die Leistung des Bagging-Klassifizierers mit derjenigen eines einzelnen, nicht gestützten Entscheidungsbäums zu vergleichen.

```
>>> from sklearn.metrics import accuracy_score
>>> tree = tree.fit(X_train, y_train)
>>> y_train_pred = tree.predict(X_train)
>>> y_test_pred = tree.predict(X_test)
>>> tree_train = accuracy_score(y_train, y_train_pred)
>>> tree_test = accuracy_score(y_test, y_test_pred)
>>> print('Entscheidungsbaum Training/Test-KKR %.3f/%.3f'
...       % (tree_train, tree_test))
Entscheidungsbaum Training/Test-KKR 1.000/0.833
```

Die ausgegebene *Korrektklassifizierungsrate* (KKR) besagt, dass der nicht gestützte Entscheidungsbaum sämtliche Klassenbezeichnungen der Trainingsdatenmenge korrekt vorhersagt. Die deutlich geringere Korrektklassifizierungsrate bei der Test-

datenmenge weist allerdings auf eine hohe Varianz (Überanpassung) des Modells hin:

```
>>> bag = bag.fit(X_train, y_train)
>>> y_train_pred = bag.predict(X_train)
>>> y_test_pred = bag.predict(X_test)
>>> bag_train = accuracy_score(y_train, y_train_pred)
>>> bag_test = accuracy_score(y_test, y_test_pred)
>>> print('Bagging Training/Test-KKR %.3f/%.3f'
...      % (bag_train, bag_test))
Bagging Training/Test-KKR 1.000/0.917
```

Die Korrektklassifizierungsraten des Entscheidungsbaums und des Bagging-Klassifizierers stimmen bei den Trainingsdaten zwar überein (jeweils 100 Prozent), bei der Testdatenmenge weist der Bagging-Klassifizierer allerdings eine etwas bessere Verallgemeinerungsfähigkeit auf. Vergleichen wir also die Entscheidungsbereiche von Entscheidungsbaum und Bagging-Klassifizierer:

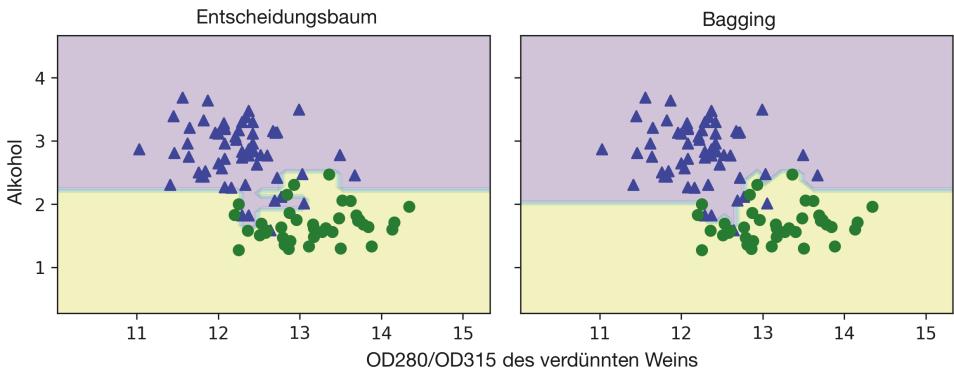
```
>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(nrows=1, ncols=2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(8, 3))
>>> for idx, clf, tt in zip([0, 1],
...                           [tree, bag],
...                           ['Entscheidungsbaum', 'Bagging']):
...     clf.fit(X_train, y_train)
...
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx].scatter(X_train[y_train==0, 0],
...                        X_train[y_train==0, 1],
...                        c='blue', marker='^')
...     axarr[idx].scatter(X_train[y_train==1, 0],
...                        X_train[y_train==1, 1],
...                        c='green', marker='o')
```

```

...     axarr[idx].set_title(tt)
>>> axarr[0].set_ylabel('Alkohol', fontsize=12)
>>> plt.text(10.2, -1.2,
...             s='OD280/OD315 des verdünnten Weins',
...             ha='center', va='center', fontsize=12)
>>> plt.show()

```

In dem resultierenden Diagramm wirkt die stückweise lineare Entscheidungsgrenze des drei Ebenen tiefen Entscheidungsbaums beim Bagging-Ensemble »glatter«:



Wir haben hier ein sehr einfaches Beispiel für das Bagging-Verfahren betrachtet. In der Praxis können kompliziertere Klassifizierungsaufgaben und hohe Dimensionalität der Datenmengen in einzelnen Entscheidungsbäumen leicht zu einer Überanpassung führen – dabei zeigt der Bagging-Algorithmus dann seine wahren Stärken. Abschließend sollte noch angemerkt werden, dass der Bagging-Algorithmus auch als ein effektiver Ansatz zur Verringerung der Varianz eines Modells dienen kann. Zur Reduzierung des Bias eines Modells ist er allerdings ungeeignet, das heißt, die Modelle sind zu einfach, um den Trend in den Daten vernünftig zu erfassen. Deshalb sollte man für das Bagging ein Klassifizierer-Ensemble mit geringem Bias wählen, wie beispielsweise nicht gestützte Entscheidungsbäume.

## 7.5 Schwache Klassifizierer durch adaptives Boosting verbessern

In diesem letzten Abschnitt zu den Ensemblemethoden werden wir das *Boosting* (engl. *Verstärken*) erörtern und dabei einen besonderen Schwerpunkt auf die gebräuchlichste Implementierung legen, die als *AdaBoost* (Abkürzung für *ADAptives BOOSTing*) bezeichnet wird.

## Tipp

Das AdaBoost zugrunde liegende Konzept wurde ursprünglich 1990 von Robert Schapire formuliert (R.E. Schapire, *The Strength of Weak Learnability*, Machine Learning, 5(2):197–227, 1990). Nachdem Robert Schapire und Yoav Freund 1996 auf der dreizehnten ICML (*International Conference on Machine Learning*) den AdaBoost-Algorithmus vorgestellt hatten, wurde er in den folgenden Jahren zur verbreitetsten Ensemblemethode (Y. Freund, R.E. Schapire et al., *Experiments with a New Boosting Algorithm*, ICML, Band 96, Seiten 148–156, 1996). 2003 erhielten Freund und Schapire für ihre bahnbrechende Arbeit den Gödel-Preis, einen renommierten Preis für herausragende Veröffentlichungen in der theoretischen Informatik.

Beim Boosting besteht das Ensemble aus sehr einfachen schwachen Klassifizierern, die manchmal auch *Weak Learners* (schwache Lerner) genannt werden. Ihre Leistung entspricht oft kaum mehr als zufälligem Raten. Ein typisches Beispiel für einen schwachen Klassifizierer ist ein einfacher Entscheidungsbaum mit nur einem Knoten (engl. *Decision Stump*). Das grundlegende Konzept beim Boosting ist, sich auf schwierig zu klassifizierende Trainingsdaten zu konzentrieren, damit die schwachen Klassifizierer allmählich aus fehlklassifizierten Objekten lernen, um so die Leistung des Ensembles zu verbessern.

In den folgenden Abschnitten wird das dem allgemeinen Konzept des Boostings zugrunde liegende algorithmische Verfahren und die verbreitete Variante namens AdaBoost vorgestellt. Anschließend erstellen wir mit scikit-learn ein Klassifizierungsbeispiel.

### 7.5.1 Funktionsweise des Boostings

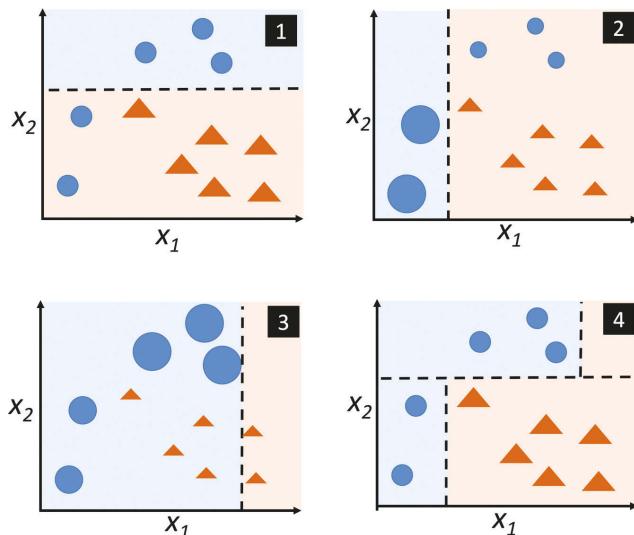
Im Gegensatz zum Bagging, der ursprünglichen Ausprägung des Boostings, verwendet dieser Algorithmus zufällig ausgewählte Stichproben der Trainingsdaten *ohne Ersetzung*. Das Boosting-Verfahren kann in vier Schritten zusammengefasst werden:

1. Auswahl einer zufälligen Stichprobe  $d_1$  aus der Trainingsdatenmenge  $D$  ohne Ersetzung, um einen schwachen Klassifizierer  $C_1$  zu trainieren.
2. Auswahl einer zweiten zufälligen Stichprobe  $d_2$  aus der Trainingsdatenmenge ohne Ersetzung und Hinzufügen von 50 Prozent der zuvor fehlklassifizierten Objekte, um einen schwachen Klassifizierer  $C_2$  zu trainieren.
3. Auswahl einer Stichprobe  $d_3$  aus der Trainingsdatenmenge  $D$ , die aus den Objekten besteht, die  $C_1$  und  $C_2$  unterschiedlich klassifiziert haben, um damit einen dritten schwachen Klassifizierer  $C_3$  zu trainieren.

4. Kombinieren der schwachen Klassifizierer  $C_1$ ,  $C_2$  und  $C_3$  durch Mehrheitsentscheidung.

Leo Breiman beschreibt (L. Breiman, *Bias, Variance, and Arcing Classifiers*, 1996), dass im Gegensatz zum Bagging beim Boosting sowohl Bias als auch Varianz sinken können. Aus der Praxis ist allerdings bekannt, dass Boosting-Algorithmen wie AdaBoost eine hohe Varianz aufweisen und zu einer Überanpassung an die Trainingsdaten neigen (G. Raetsch, T. Onoda und K.R. Mueller, *An Improvement of AdaBoost to Avoid Overfitting*, Proceedings of the International Conference on Neural Information Processing, CiteSeer, 1998).

Im Gegensatz zu dem hier beschriebenen ursprünglichen Boosting-Verfahren nutzt AdaBoost die gesamte Trainingsdatenmenge zum Trainieren der schwachen Klassifizierer, wobei die Trainingsobjekte bei jedem Durchgang erneut gewichtet werden, um einen besseren Klassifizierer zu erzeugen, der aus den Fehlern der im Ensemble vorhergehenden schwachen Klassifizierer lernt. Bevor wir uns eingehender mit den Details des AdaBoost-Algorithmus befassen, betrachten wir zum besseren Verständnis des zugrunde liegenden Konzepts zuerst noch die folgende Abbildung.



Wir gehen die Abbildung schrittweise durch und betrachten zunächst Teil 1. Hierbei handelt es sich um die Trainingsdatenmenge einer binären Klassifizierung, bei der allen Trainingsobjekten das gleiche Gewicht zugeordnet ist. Mit dieser Menge trainieren wir einen einfachen Entscheidungsbaum (als gestrichelte Linie dargestellt), der sowohl versucht, die Objekte zu klassifizieren (Dreiecke und Kreise), als auch die Straffunktion (oder im speziellen Fall eines Entscheidungsbaum-Ensembles die Unreinheit) zu minimieren.

In der nächsten Runde (Teil 2 der Abbildung) weisen wir den beiden zuvor fehlklassifizierten Objekten (den beiden Kreisen) eine höhere Gewichtung zu. Darüber hinaus verringern wir die Gewichtung der korrekt klassifizierten Objekte. Der nächste Entscheidungsbaum ist nun stärker auf die Trainingsobjekte mit den größten Gewichtungen fokussiert, also auf die vermutlich schwer klassifizierbaren Objekte. Der schwache Klassifizierer in Teil 2 der Abbildung fehlklassifizierte drei der zur Klasse »Kreis« zugehörigen Objekte, denen daraufhin eine höhere Gewichtung zugewiesen wird, wie in Teil 3 der Abbildung dargestellt.

Würde unser AdaBoost-Ensemble nur aus drei Boosting-Durchgängen bestehen, würden wir nun die drei durch eine gewichtete Mehrheitsentscheidung mit verschiedenen neu gewichteten Trainingsdatensätzen trainierten schwachen Klassifizierer miteinander kombinieren (Teil 4 der Abbildung).

Da Sie das dem AdaBoost-Algorithmus zugrunde liegende Konzept nun schon ein wenig besser kennen, können Sie ihn mithilfe von Pseudocode genauer betrachten. Der Deutlichkeit halber verwenden wir für die elementweise Multiplikation ein Kreuzsymbol ( $\times$ ) und für das Skalarprodukt zweier Vektoren ein Punktsymbol ( $\cdot$ ). Hier die einzelnen Schritte:

1. Dem Gewichtungsvektor  $w$  werden einheitliche Gewichtungen zugewiesen. Es gilt  $\sum_i w_i = 1$ .
2. In den  $m$  Boosting-Durchgängen müssen mit  $j$  die folgenden Schritte ausgeführt werden:
  - a) Trainieren eines schwachen Klassifizierers:  $C_j = \text{train}(X, y, w)$
  - b) Vorhersage der Klassenbezeichnungen:  $\hat{y} = \text{predict}(C_j, X)$
  - c) Berechnung der gewichteten Fehlerquote:  $\varepsilon = w \cdot (\hat{y} \neq y)$
  - d) Berechnung des Koeffizienten:  $\alpha_j = 0.5 \log \frac{1-\varepsilon}{\varepsilon}$
  - e) Aktualisierung der Gewichtungen:  $w := w \times \exp(-\alpha_j \times \hat{y} \times y)$
  - f) Normierung der Gewichtungen, sodass die Summe 1 ergibt:  $w := w / \sum_i w_i$
3. Berechnung der endgültigen Vorhersage:  $\hat{y} = \left( \sum_{j=1}^m (\alpha_j \times \text{predict}(C_j, X)) \right) > 0$

Beachten Sie hier, dass sich der Ausdruck  $(\hat{y} \neq y)$  in Schritt 2c) auf einen Vektor bezieht, dessen Elemente Einsen und Nullen sind. Wenn eine Vorhersage falsch ist, wird eine 1 zugewiesen, anderenfalls eine 0.

Der AdaBoost-Algorithmus ist eigentlich ziemlich schnörkellos, wir wollen aber dennoch ein konkreteres Beispiel mit einer aus 10 Objekten bestehenden Trainingsdatensetzung betrachten, die in der folgenden Abbildung dargestellt ist.

## Kapitel 7

### Kombination verschiedener Modelle für das Ensemble Learning

Stichprobenindex	x	y	Gewichtung	$\hat{y}(x \leq 3.0)$ ?	Korrekt?	Aktualisierte Gewichtung
1	1.0	1	0.1	1	Ja	0.072
2	2.0	1	0.1	1	Ja	0.072
3	3.0	1	0.1	1	Ja	0.072
4	4.0	-1	0.1	-1	Ja	0.072
5	5.0	-1	0.1	-1	Ja	0.072
6	6.0	-1	0.1	-1	Ja	0.072
7	7.0	1	0.1	-1	Nein	0.167
8	8.0	1	0.1	-1	Nein	0.167
9	9.0	1	0.1	-1	Nein	0.167
10	10.0	-1	0.1	-1	Ja	0.072

In der ersten Tabellenspalte befindet sich der Stichprobenindex der Trainingsobjekte 1 bis 10. In der zweiten Spalte sind Merkmalswerte der einzelnen Objekte aufgeführt, es soll sich also um eine eindimensionale Datenmenge handeln. In der dritten Spalte ist die tatsächliche Klassenbezeichnung  $y_i$  des Objekts  $x_i$  angegeben, wobei  $y_i \in \{1, -1\}$  gilt. Die vierte Spalte enthält die ursprünglichen Gewichtungen: Wir initialisieren sie mit einem einheitlichen Wert und normieren sie, sodass die Summe 1 ergibt – bei 10 Objekten weisen wir also jeder Gewichtung  $w_i$  im Gewichtungsvektor  $w$  den Wert 0.1 zu. Die vorhergesagte Klasse  $\hat{y}$  ist in der fünften Spalte unter Zugrundelegung des Trennkriteriums  $x \leq 3.0$  aufgeführt. In der sechsten Spalte steht, ob die Vorhersage korrekt ist, und in der letzten Spalte ist schließlich die anhand der im Pseudocode festgelegten Regeln berechnete aktualisierte Gewichtung angegeben.

Da die Berechnung der aktualisierten Gewichtungen auf den ersten Blick ein wenig kompliziert aussieht, führen wir sie schrittweise durch. Zunächst berechnen wir wie in Schritt 2c beschrieben die gewichtete Fehlerquote  $\varepsilon$ :

$$\begin{aligned}\varepsilon &= 0.1 \times 0 + 0.1 \times 1 + 0.1 \times 1 \\ &\quad + 0.1 \times 1 + 0.1 \times 0 = \frac{3}{10} = 0.3\end{aligned}$$

Dann wird der Koeffizient  $\alpha_j$  aus Schritt 2d) berechnet, der in Schritt 2e) zur Aktualisierung der Gewichtungen und in Schritt 3 zur Berechnung der endgültigen Vorhersage verwendet wird:

$$\alpha_j = 0.5 \log\left(\frac{1-\varepsilon}{\varepsilon}\right) \approx 0.424$$

Nach der Berechnung des Koeffizienten  $\alpha_j$  können wir den Gewichtungsvektor anhand folgender Gleichung aktualisieren:

$$\mathbf{w} := \mathbf{w} \times \exp(-\alpha_j \times \hat{\mathbf{y}} \times \mathbf{y})$$

Hier ist  $\hat{\mathbf{y}} \times \mathbf{y}$  eine elementweise Multiplikation der Vektoren der vorhergesagten und der tatsächlichen Klassenbezeichnung. Wenn also eine Vorhersage  $\hat{y}_i$  zutrifft, besitzt  $\hat{y}_i \times y_i$  ein positives Vorzeichen, wodurch die  $i$ -te Gewichtung verringert wird, denn  $\alpha$  ist ja ebenfalls eine positive Zahl:

$$0.1 \times \exp(-0.424 \times 1 \times 1) \approx 0.065$$

Auf ähnliche Weise wird die  $i$ -te Gewichtung erhöht, wenn  $\hat{y}_i$  die Klassenbezeichnung falsch vorhersagt, z.B. so

$$0.1 \times \exp(-0.424 \times 1 \times (-1)) \approx 0.153$$

oder so:

$$0.1 \times \exp(-0.424 \times (-1) \times (1)) \approx 0.153$$

Nach der Aktualisierung des Gewichtungsvektors normieren wir die Gewichtungen, sodass ihre Summe 1 ergibt (Schritt 2f)):

$$\mathbf{w} := \frac{\mathbf{w}}{\sum_i w_i}$$

Hier ist  $\sum_i w_i = 7 \times 0.065 + 3 \times 0.153 = 0.914$ .

Allen zu einem korrekt klassifizierten Objekt zugehörigen Gewichtungen wird also für den nächsten Boosting-Durchgang statt des anfänglichen Wertes 0.1 der Wert  $0.065 / 0.914 \approx 0.071$  zugewiesen. Und die Gewichtungen der fehlklassifizierten Objekte werden von 0.1 auf  $0.153 / 0.914 \approx 0.167$  erhöht.

## 7.5.2 AdaBoost mit scikit-learn anwenden

So viel, kurz und bündig, zum AdaBoost-Algorithmus. Im Folgenden wenden wir uns dem eher praktischen Teil zu und trainieren ein AdaBoost-Klassifizierer-Ensemble mit scikit-learn. Wir verwenden dieselbe Wein-Datensammlung wie im vorangegangenen Abschnitt beim Bagging-Klassifizierer. Mithilfe des `base_estimator`-Attributs trainieren wir einen `AdaBoostClassifier` mit 500 einfachen Entscheidungsbäumen:

```
>>> from sklearn.ensemble import AdaBoostClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                                 max_depth=1,
...                                 random_state=1)
>>> ada = AdaBoostClassifier(base_estimator=tree,
...                           n_estimators=500,
...                           learning_rate=0.1,
```

```

...                                         random_state=1)
>>> tree = tree.fit(X_train, y_train)
>>> y_train_pred = tree.predict(X_train)
>>> y_test_pred = tree.predict(X_test)
>>> tree_train = accuracy_score(y_train, y_train_pred)
>>> tree_test = accuracy_score(y_test, y_test_pred)
>>> print('Entscheidungsbaum Training/Test-KKR %.3f/%.3f'
...      % (tree_train, tree_test))
Entscheidungsbaum Training/Test-KKR 0.916/0.875

```

Wie man sieht, neigt der einfache Entscheidungsbaum im Gegensatz zum nicht gestützten Entscheidungsbaum im vorangegangenen Abschnitt zu einer Unteranpassung der Trainingsdaten.

```

>>> ada = ada.fit(X_train, y_train)
>>> y_train_pred = ada.predict(X_train)
>>> y_test_pred = ada.predict(X_test)
>>> ada_train = accuracy_score(y_train, y_train_pred)
>>> ada_test = accuracy_score(y_test, y_test_pred)
>>> print('AdaBoost Training/Test-KKR %.3f/%.3f'
...      % (ada_train, ada_test))
AdaBoost Training/Test-KKR 1.000/0.917

```

Das AdaBoost-Modell sagt alle Klassenbezeichnungen der Trainingsdatenmenge korrekt voraus und zeigt auch bei der Testdatenmenge eine etwas bessere Leistung als der einfache Entscheidungsbaum. Durch den Versuch, das Bias des Modells zu verringern, haben wir allerdings auch die Varianz erhöht – die Lücke zwischen den Leistungen bei Trainings- und Testdaten ist größer geworden.

In diesem Fall haben wir zu Demonstrationszwecken zwar wieder nur ein einfaches Beispiel betrachtet, dennoch wird deutlich, dass sich die Leistung des AdaBoost-Klassifizierers im Vergleich zum einfachen Entscheidungsbaum leicht verbessert hat und dass er eine mit dem im vorangegangenen Abschnitt trainierten Bagging-Klassifizierer vergleichbare Korrektklassifizierungsrate erzielt. Hier ist jedoch anzumerken, dass es als *Bad Practice*, sprich ungeeignete Verfahrensweise gilt, ein Modell auf der Grundlage der mehrmaligen Verwendung der Testdatenmenge auszuwählen. Die Einschätzung der Verallgemeinerungsfähigkeit könnte zu optimistisch sein, wie in Kapitel 6 bereits ausführlicher erläutert wurde. Zum Abschluss sehen wir uns wieder an, wie die Entscheidungsbereiche aussehen:

```

>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1

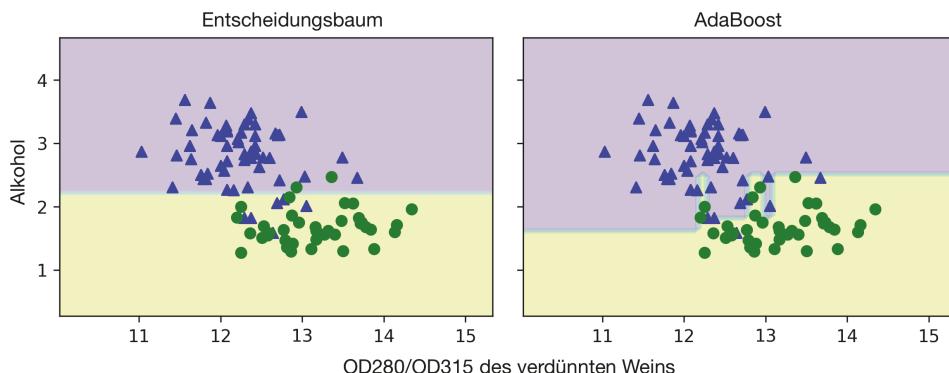
```

```

>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(1, 2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(8, 3))
>>> for idx, clf, tt in zip([0, 1],
...                           [tree, ada],
...                           ['Entscheidungsbaum', 'AdaBoost']):
...     clf.fit(X_train, y_train)
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx].scatter(X_train[y_train==0, 0],
...                        X_train[y_train==0, 1],
...                        c='blue',
...                        marker='^')
...     axarr[idx].scatter(X_train[y_train==1, 0],
...                        X_train[y_train==1, 1],
...                        c='green',
...                        marker='o')
...     axarr[idx].set_title(tt)
...     axarr[0].set_ylabel('Alkohol', fontsize=12)
>>> plt.text(10.2, -1.2,
...            s='OD280/OD315 des verdünnten Weins',
...            ha='center',
...            va='center',
...            fontsize=12)
>>> plt.show()

```

Wie man sieht, sind die Entscheidungsgrenzen des AdaBoost-Modells erheblich komplizierter als die des einfachen Entscheidungsbaums. Außerdem ist festzustellen, dass das AdaBoost-Modell den Merkmalsraum sehr ähnlich wie der im vorangegangenen Abschnitt trainierte Bagging-Klassifizierer unterteilt.



Als Schlussbemerkung zu den Ensemblemethoden können wir feststellen, dass das Ensemble Learning die Komplexität der Berechnungen im Vergleich zur Verwendung einzelner Klassifizierer erhöht. In der Praxis ist daher sorgfältig abzuwagen, ob es sich lohnt, für eine häufig nur relativ bescheidene Verbesserung der Vorhersagekraft den Preis des erhöhten Rechenaufwands zu zahlen.

Ein oft genanntes Beispiel für diese Abwägung ist der berühmte Millionen-Dollar-Netflix-Preis, der mithilfe von Ensemblemethoden gewonnen wurde. Die Details zu diesem Algorithmus wurden von A. Toescher, M. Jahrer und R.M. Bell in *The Bigchaos Solution to the Netflix Grand Prize*, Netflix prize documentation, 2009, unter [http://www.stat.osu.edu/~dms1/GrandPrize2009\\_BPC\\_BigChaos.pdf](http://www.stat.osu.edu/~dms1/GrandPrize2009_BPC_BigChaos.pdf) veröffentlicht. Das Gewinnerteam erhielt zwar das Preisgeld von einer Million Dollar, Netflix hat das Modell jedoch aufgrund der Komplexität, die es für eine praktische Anwendung untauglich macht, nie implementiert. Hier ein Zitat aus der Begründung (<http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html>):

»[...] die gemessenen Verbesserungen der Korrektklassifizierungsrate vermögen den für die praktische Umsetzung erforderlichen Aufwand offenbar nicht rechtfertigen.«

## 7.6 Zusammenfassung

In diesem Kapitel haben wir einige der verbreitetsten und am häufigsten eingesetzten Verfahren des Ensemble Learnings betrachtet. Ensemblemethoden kombinieren verschiedene Klassifizierungsmodelle, um die individuellen Schwächen wettzumachen – was oft zu stabilen und gut funktionierenden Modellen führt, die sowohl für industrielle Anwendungen als auch für Machine-Learning-Wettbewerbe sehr interessant sind.

Zu Beginn dieses Kapitels haben wir den Mehrheitsentscheidungs-Klassifizierer `MajorityVoteClassifier` in Python implementiert, der es uns ermöglicht, verschiedene Klassifizierungsalgorithmen zu kombinieren. Dann haben wir uns mit dem Bagging befasst, einem nützlichen Verfahren zur Verringerung der Varianz eines Modells, bei dem der Trainingsdatenmenge zufällige Bootstrap-Stichproben entnommen werden und die einzeln trainierten Klassifizierer eine Mehrheitsentscheidung treffen. Und schließlich haben wir AdaBoost betrachtet, einen Algorithmus, der auf schwachen Klassifizierern beruht, die allmählich aus Fehlern lernen.

In diesem Kapitel haben wir verschiedene Lernalgorithmen, Abstimmungs- und Bewertungsverfahren erörtert. Im nächsten Kapitel sehen wir uns eine spezielle Anwendung des Machine Learnings an, die Analyse von Stimmungslagen, die im Zeitalter des Internets und sozialer Medien zweifelsohne zu einem hochinteressanten Thema geworden ist.

# Machine Learning zur Analyse von Stimmungslagen nutzen

Im Zeitalter des Internets und Social Media sind die Meinungen, Bewertungen und Empfehlungen der Menschen für die Politik ebenso wie für die Geschäftswelt zu einer wertvollen Ressource geworden. Und dank moderner Technologien sind wir nun in der Lage, solche Daten äußerst effizient zu sammeln und zu analysieren. In diesem Kapitel werden wir uns mit einem Teilgebiet der *Verarbeitung natürlicher Sprache (NLP, Natural Language Processing)* befassen, der sogenannten *Stimmungsanalyse* (oder auch *Sentimentanalyse*). Sie werden erfahren, wie sich Lernalgorithmen dafür nutzen lassen, Texte anhand ihrer Tonalität, die den Standpunkt des Autors widerspiegelt, zu klassifizieren. Außerdem werden wir die Datensammlung der *Internet Movie Database (IMDb)* verwenden, die mehr als 50.000 Filmbewertungen enthält und ein Vorhersagesystem entwickeln, das zwischen positiven und negativen Bewertungen unterscheiden kann.

Die Themen in diesem Kapitel sind:

- Bereinigen und Aufbereiten von Texten
- Merkmalsvektoren für Textdokumente erstellen
- Trainieren eines Lernmodells für die Klassifizierung positiver und negativer Filmbewertungen
- Verarbeitung sehr langer Texte mit dem *Out-of-Core-Algorithmus*
- Aus einer Dokumentensammlung Themen zur Kategorisierung ableiten

## 8.1 Die IMDb-Filmdatenbank

Die Stimmungsanalyse, die manchmal auch als *Opinion Mining* bezeichnet wird, ist ein Teilgebiet der Verarbeitung natürlicher Sprache, das sich mit der Untersuchung der *Tonalität* von Texten befasst. Häufig besteht die Aufgabe einer Stimmungsanalyse darin, Texte anhand der von den Autoren zum Ausdruck gebrachten Ansichten und Empfindungen zu einem bestimmten Thema zu klassifizieren.

In diesem Kapitel werden wir eine umfangreiche Sammlung von Filmbewertungen der *Internet Movie Database (IMDb)* verwenden, die von Maas et al. zusammengestellt wurde (A.L. Maas, R.E. Daly, P.T. Pham, D. Huang, A.Y. Ng und C. Potts,

*Learning Word Vectors for Sentiment Analysis*, in: Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics; Human Language Technologies, Seiten 142-150, Portland, Oregon, USA, Juni 2011, Association for Computational Linguistics). Die Datensammlung besteht aus 50.000 gegensätzlichen Filmbewertungen, die entweder als *positiv* oder *negativ* gekennzeichnet sind. Positiv bedeutet hier, dass ein Film bei der IMDb mit mehr als sechs (von höchstens zehn) Sternen bewertet wurde. Negativ heißt hingegen, dass ein Film weniger als fünf Sterne erhielt. In den folgenden Abschnitten werden Sie erfahren, wie aus einer Teilmenge dieser Filmbewertungen aussagekräftige Informationen entnommen werden können, um ein Lernmodell zu entwickeln, das vorhersagt, ob ein Film einem bestimmten Reviewer (Bewerter) gefallen hat oder nicht.

### 8.1.1 Herunterladen der Datensammlung

Die Datensammlung (84,1 MB) kann unter <http://ai.stanford.edu/~amaas/data/sentiment/> als komprimierte gzip-Datei heruntergeladen werden:

- Wenn Sie Linux oder macOS verwenden, können Sie ein Terminalfenster öffnen, mit cd in das Verzeichnis Downloads wechseln und den Befehl tar -zxf aclImdb\_v1.tar.gz ausführen, um die Datei zu entkomprimieren.
- Wenn Sie Windows verwenden, können Sie sich ein kostenloses Programm zum Entkomprimieren herunterladen, z.B. 7-zip (<http://www.7-zip.org>), um die Dateien aus dem Archiv zu extrahieren.
- Alternativ können Sie die gzip-Datei wie folgt direkt in Python entpacken:

```
>>> import tarfile  
>>> with tarfile.open('aclImdb_v1.tar.gz', 'r:gz') as tar:  
...     tar.extractall()
```

### 8.1.2 Vorverarbeiten der Filmbewertungsdaten

Nach dem Entkomprimieren müssen die einzelnen Textdokumente des Downloads zu einer einzigen CSV-Datei vereinigt werden. Der folgende Codeabschnitt liest die Filmbewertungen in ein pandas-DataFrame-Objekt ein, was auf einem normalen Rechner bis zu 10 Minuten dauern kann. Zur Anzeige des Fortschritts und der verbleibenden Zeit verwenden wir das *PyPrind-Paket* (*Python Progress Indicator*, <https://pypi.python.org/pypi/PyPrind/>), das ich vor einigen Jahren für diesen Zweck entwickelt habe. PyPrind kann durch Eingabe des Befehls pip install pyprind installiert werden.

```
>>> import pyprind  
>>> import pandas as pd  
>>> import os  
>>> # Ändern Sie `basepath` auf das Verzeichnis, in dem
```

```
>>> # sich die komprimierte Datei befindet.
>>> basepath = 'aclImdb'
>>> pbar = pyprind.ProgBar(50000)
>>> labels = {'pos':1, 'neg':0}
>>> df = pd.DataFrame()
>>> for s in ('test', 'train'):
...     for l in ('pos', 'neg'):
...         path = os.path.join(basepath, s, l)
...         for file in os.listdir(path):
...             with open(os.path.join(path, file),
...                       'r', encoding='utf-8') as infile:
...                 txt = infile.read()
...                 df = df.append([[txt, labels[l]]],
...                               ignore_index=True)
...             pbar.update()
>>> df.columns = ['review', 'sentiment']
0%          100%
[#####] | ETA[sec]: 0.000
Total time elapsed: 00:03:37
```

Der Code initialisiert einen Fortschrittbalken `pbar` zur Anzeige von 50.000 Schritten, denn so viele Dokumente werden eingelesen. Mit den verschachtelten `for`-Schleifen durchlaufen wir die Unterverzeichnisse `train` und `test` im Hauptverzeichnis `aclImdb` und lesen die einzelnen Textdateien der Unterverzeichnisse `pos` und `neg` ein, die wir schließlich zusammen mit einer ganzzahligen Klassenbezeichnung (1 = positiv und 0 = negativ) dem `df`-Dataframe hinzufügen.

Da die Klassenbezeichnungen in der Datensammlung sortiert sind, werden wir den Dataframe nun mit der `permutation`-Funktion des `np.random`-Submoduls durchmischen. Das erweist sich als nützlich, wenn wir die die Datenmenge später beim direkten Einlesen vom lokalen Laufwerk in Trainings- und Testdaten aufteilen möchten. Der Bequemlichkeit halber speichern wir die durchgemischte Filmbewertungs-Datensammlung zusätzlich auch als CSV-Datei:

```
>>> import numpy as np
>>> np.random.seed(0)
>>> df = df.reindex(np.random.permutation(df.index))
>>> df.to_csv('movie_data.csv', index=False, encoding='utf-8')
```

Da wir diese Datensammlung später noch verwenden werden, sollten wir an dieser Stelle überprüfen, ob die Daten im richtigen Format gespeichert wurden, indem wir die ersten drei Datensätze einlesen und anzeigen:

```
>>> df = pd.read_csv('./movie_data.csv')
>>> df.head(3)
```

Wenn Sie den Code in einem Jupyter-Notebook ausführen, sollten Ihnen nun wie in der folgenden Tabelle die drei ersten Datensätze angezeigt werden:

	review	sentiment
0	In 1974, the teenager Martha Moxley (Maggie Gr...	1
1	OK... so... I really like Kris Kristofferson a...	0
2	***SPOILER*** Do not read this, if you think a...	0

## 8.2 Das Bag-of-words-Modell

Aus Kapitel 4 wissen Sie, dass wir kategoriale Daten wie Text oder Wörter in numerische Daten umwandeln müssen, bevor wir sie Lernalgorithmen zur Verarbeitung übergeben können. In diesem Abschnitt verwenden wir das *Bag-of-words-Modell* (»Beutel-voller-Wörter«-Modell), das es uns erlaubt, Text als numerischen Merkmalsvektor zu repräsentieren. Die diesem Modell zugrunde liegende Idee ist ganz einfach und kann folgendermaßen zusammengefasst werden:

1. Wir erstellen ein *Vokabular* eindeutiger *Tokens* – beispielsweise Wörter – für sämtliche Textdokumente.
2. Dann konstruieren wir für jedes Textdokument einen Merkmalsvektor, der die Anzahl der Vorkommen jedes einzelnen Wortes im jeweiligen Dokument enthält.

Da die verschiedenen Wörter in einem Textdokument nur eine kleine Teilmenge aller im Bag-of-words-Vokabular vorhandenen Begriffe darstellen, enthalten die Merkmalsvektoren hauptsächlich Nullen und werden deshalb als *dünnbesetzt* bezeichnet. Wenn Ihnen das nun ein wenig zu abstrakt erscheint, keine Sorge: In den folgenden Abschnitten werden wir die für die Erstellung eines Bag-of-words-Modells erforderlichen Schritte der Reihe nach durchgehen.

### 8.2.1 Wörter in Merkmalsvektoren umwandeln

Um anhand der Wörterzahlen des jeweiligen Textdokuments ein Bag-of-words-Modell zu erstellen, können wir die in scikit-learn implementierte *CountVectorizer*-Klasse nutzen. Wie Sie im folgenden Codeabschnitt sehen werden, nimmt die *CountVectorizer*-Klasse ein Array mit Textdaten entgegen, bei dem es sich um ganze Dokumente oder nur um einzelne Sätze handeln kann, und konstruiert daraus ein Bag-of-words-Modell.

```
>>> import numpy as np
>>> from sklearn.feature_extraction.text
...                                     import CountVectorizer
>>> count = CountVectorizer()
```

```
>>> docs = np.array([
...     'The sun is shining',
...     'The weather is sweet',
...     'The sun is shining and the weather is sweet'])
>>> bag = count.fit_transform(docs)
```

Durch den Aufruf der `fit_transform`-Methode der `CountVectorizer`-Klasse wird das Vokabular des Bag-of-words-Modells erstellt und die folgenden drei englischen Sätze/Dokumente werden in dünnbesetzte Merkmalsvektoren transformiert:

1. The sun is shining
2. The weather is sweet
3. The sun is shining, the weather is sweet, and on and one is two

Nun geben wir den Inhalt des Vokabulars aus, um das zugrunde liegende Konzept besser zu verstehen:

```
>>> print(count.vocabulary_)
{'and': 0,
 'two': 7,
 'shining': 3,
 'one': 2,
 'sun': 4,
 'weather': 8,
 'the': 6,
 'sweet': 5,
 'is': 1}
```

Wie die Ausgabe des Befehls zeigt, wird das Vokabular in einem Python-Dictionary gespeichert, in dem den einzelnen Wörtern ein ganzzahliger Index zugeordnet ist. Als Nächstes geben wir die soeben erstellten Merkmalsvektoren aus:

```
>>> print(bag.toarray())
[[0 1 0 1 1 0 1 0 0]
 [0 1 0 0 0 1 1 0 1]
 [2 3 2 1 1 1 2 1 1]]
```

Die Indexpositionen in den Merkmalsvektoren entsprechen den im `CountVectorizer`-Vokabular als Dictionary-Elemente gespeicherten ganzzahligen Werten. So gibt beispielsweise die Indexposition 0 die Anzahl der Vorkommen des Wortes »and« an, das nur im letzten Dokument enthalten ist. Das Wort »is« an der Indexposition 1 (dem zweiten Merkmal des Vektors) kommt hingegen in allen drei Sätzen vor. Die Werte in den Merkmalsvektoren werden als *Vorkommenshäufigkeiten (Raw Term Frequencies)  $tf(t,d)$*  bezeichnet: die Anzahl der Vorkommen des Wortes  $t$  in einem Dokument  $d$ .

**Tipp**

Die beim Bag-of-words-Modell durch die Zerlegung des Textes erzeugten Textfragmente werden auch als *Monogramme* bezeichnet, denn jedes Objekt oder Token des Vokabulars repräsentiert ein einziges Wort. Im Allgemeinen werden die zusammenhängenden Textfragmente – Wörter, Buchstaben oder Symbole – bei der Verarbeitung natürlicher Sprache *N-Gramme* genannt. Die Wahl der für ein N-Gramm-Modell verwendeten Zahl *N* hängt von der speziellen Anwendung ab. Eine Untersuchung von Kanaris et al. hat beispielsweise ergeben, dass N-Gramme der Größen 3 und 4 beim Filtern von E-Mails nach Spam gute Ergebnisse liefern (Ioannis Kanaris, Konstantinos Kanaris, Ioannis Houvardas und Efstathios Stamatatos, *Words versus Character N-Grams for Anti-Spam Filtering*, International Journal on Artificial Intelligence Tools, 16(06):1047-1067, 2007). Um das Konzept der Repräsentation als N-Gramme zusammenzufassen, würden die Monogramm- und Bigramm-Repräsentationen ( $N=1$  bzw.  $N=2$ ) des Satzes »the sun is shining« folgendermaßen aussehen:

Monogramm: »the«, »sun«, »is«, »shining«

Bigramm: »the sun«, »sun is«, »is shining«

Die `CountVectorizer`-Klasse von scikit-learn gestattet mittels des `ngram_range`-Parameters die Nutzung verschiedener N-Gramm-Modelle. Standardmäßig wird eine Monogramm-Repräsentation verwendet, wir könnten jedoch auch zu einer Bigramm-Repräsentation wechseln, indem wir eine neue `CountVectorizer`-Instanz mit `ngram_range(2, 2)` initialisieren.

### 8.2.2 Beurteilung der Wortelevanz durch das Tf-idf-Maß

Bei der Analyse von Texten stoßen wir des Öfteren auf Wörter, die in mehreren Dokumenten vorkommen, die zu beiden Klassen gehören. Diese häufig auftau chenden Wörter enthalten typischerweise keine nützlichen oder für die Unterscheidbarkeit brauchbaren Informationen. In diesem Abschnitt werden Sie ein Verfahren zur Beurteilung der Wortelevanz kennenlernen: das *Tf-idf-Maß* (von engl. *Term frequency/inverse document frequency*, Vorkommenshäufigkeit/inverse Dokumentenhäufigkeit), das zur Gewichtung dieser Wörter im Merkmalsvektor verwendet werden kann. Das Tf-idf-Maß kann als Produkt der Vorkommenshäufigkeit und der inversen Dokumentenhäufigkeit definiert werden:

$$\text{tf-idf}(t,d) = \text{tf}(t,d) \times \text{idf}(t,d)$$

Hier repräsentiert  $\text{tf}(t,d)$  die im vorangegangenen Abschnitt eingeführte Vorkommenshäufigkeit und die inverse Dokumentenhäufigkeit  $\text{idf}(t,d)$  kann folgendermaßen berechnet werden:

$$\text{idf}(t,d) = \log \frac{n_d}{1 + \text{df}(d,t)}$$

$n_d$  bezeichnet die Gesamtzahl der Dokumente und  $\text{df}(d,t)$  die Anzahl der Dokumente  $d$ , die das Wort  $t$  enthalten. Beachten Sie, dass das Hinzufügen der Konstanten 1 zum Nenner optional ist und dazu dient, allen in der Trainingsdatenmenge enthaltenen Objekten einen von null verschiedenen Wert zuzuordnen. Der Logarithmus soll gewährleisten, dass geringen Vorkommenshäufigkeiten nicht zu viel Gewicht beigemessen wird.

In scikit-learn ist noch ein weiterer Transformer implementiert, der die Vorkommenshäufigkeiten des `CountVectorizers` als Eingabe entgegennimmt und in Tf-idf-Maße transformiert:

```
>>> from sklearn.feature_extraction.text
...                               import TfidfTransformer
>>> tfidf = TfidfTransformer(use_idf=True,
...                           ...
...                           norm='l2',
...                           smooth_idf=True)
>>> np.set_printoptions(precision=2)
>>> print(tfidf.fit_transform(count.fit_transform(docs)))
...           .toarray())
[[ 0.      0.43   0.      0.56   0.      0.43   0.      0.      ]
 [ 0.      0.43   0.      0.      0.56   0.43   0.      0.56]
 [ 0.5     0.45   0.5     0.19   0.19   0.3     0.25   0.19]]
```

Im vorangegangenen Abschnitt besaß das Wort »is« die größte Vorkommenshäufigkeit im dritten Dokument, weil es am häufigsten enthalten ist. Nach der Transformation des Merkmalsvektors in ein Tf-idf-Maß besitzt das Wort »is« im dritten Dokument allerdings nur einen relativ geringen Wert (0.45), weil es auch in den beiden anderen Dokumenten vorkommt – und damit ist es unwahrscheinlich, dass es nützliche oder für die Unterscheidbarkeit brauchbare Informationen enthält.

Wenn wir die Tf-idf-Maße der Wörter im Merkmalsvektor von Hand berechneten, würden wir feststellen, dass der `TfidfTransformer` die Tf-idf-Maße im Vergleich zu den in Standardlehrbüchern angegebenen Gleichungen, die wir zuvor definiert haben, ein wenig anders errechnet. Die in scikit-learn implementierte Gleichung für die inverse Dokumentenhäufigkeit lautet:

$$\text{idf}(t,d) = \log \frac{1 + n_d}{1 + \text{df}(d,t)}$$

Auch das mit scikit-learn errechnete Tf-idf-Maß weicht ein wenig von der vorhin definierten Standardgleichung ab:

$$\text{tf-idf}(t,d) = tf(t,d) \times (\text{idf}(t,d) + 1)$$

Üblicherweise werden die Vorkommenshäufigkeiten normiert, bevor die Tf-idf-Maße berechnet werden, der `TfidfTransformer` hingegen normiert sie direkt. Standardmäßig (`norm='l2'`) verwendet der `TfidfTransformer` von scikit-learn die L2-Normierung, die einen Vektor der Länge 1 zurückliefert, indem ein nicht normierter Merkmalsvektor durch seine L2-Norm geteilt wird:

$$v_{\text{norm}} = \frac{v}{\|v\|_2} = \frac{v}{\sqrt{v_1^2 + v_2^2 + \dots + v_n^2}} = \frac{v}{\left(\sum_{i=1}^n v_i^2\right)^{1/2}}$$

Um zu gewährleisten, dass wir die `TfidfTransformer`-Funktionsweise richtig verstanden haben, betrachten wir ein Beispiel und berechnen das Tf-idf-Maß des Wortes »is« im dritten Dokument.

Das Wort »is« besitzt im dritten Dokument eine Vorkommenshäufigkeit von 3 ( $tf=3$ ), und die Dokumentenhäufigkeit des Wortes ist auch 3, denn es kommt in allen drei Dokumenten vor ( $df=3$ ). Die inverse Dokumentenhäufigkeit errechnet sich dann so:

$$\text{idf}("is", d3) = \log \frac{1+3}{1+3} = 0$$

Um nun das Tf-idf-Maß zu berechnen, brauchen wir nur 1 zur inversen Dokumentenhäufigkeit zu addieren und mit der Vorkommenshäufigkeit zu multiplizieren:

$$\text{tf-idf}("is", d3) = 3 \times (0 + 1) = 3$$

Wenn wir diese Berechnungen mit allen Wörtern des dritten Dokuments durchführen, erhalten wir den Tf-idf-Vektor [3.39, 3.0, 3.39, 1.29, 1.29, 1.29, 1.29, 2.0, 1.69, 1.29]. Allerdings fällt auf, dass sich die Werte in diesem Merkmalsvektor von den zuvor mit dem `TfidfTransformer` berechneten unterscheiden. Der noch fehlende abschließende Schritt ist die L2-Normierung, die wie folgt durchgeführt wird:

$$\begin{aligned} \text{tf-idf}(d3)_{\text{norm}} &= \frac{[3.39, 3.0, 3.39, 1.29, 1.29, 1.29, 1.29, 2.0, 1.69, 1.29]}{\sqrt{3.39^2 + 3.0^2 + 3.39^2 + 1.29^2 + 1.29^2 + 1.29^2 + 2.0^2 + 1.69^2 + 1.29^2}} \\ &= [0.5, 0.45, 0.5, 0.19, 0.19, 0.19, 0.3, 0.25, 0.19] \end{aligned}$$

$$\text{tf-idf}("is", d3) = 0.45$$

Nun stimmt das Ergebnis mit dem von scikit-learns `TfidfTransformer` zurückgegebenen überein. Da wir nun wissen, wie Tf-idf-Maße berechnet werden, können wir mit dem nächsten Abschnitt fortfahren und diese Konzepte auf die Filmbewertungsdatenbank anwenden.

### 8.2.3 Textdaten bereinigen

In den vorangegangenen Abschnitten haben Sie das Bag-of-words-Modell, die Vorkommenshäufigkeit und das Tf-idf-Maß kennengelernt. Bevor wir jedoch ein Bag-of-words-Modell erstellen, muss ein erster wichtiger Schritt erledigt werden: Wir müssen den Text bereinigen und alle unerwünschten Zeichen entfernen. Um zu veranschaulichen, warum das von Bedeutung ist, zeigen wir die letzten 50 Zeichen des ersten Dokuments der durchgemischten Filmbewertungs-Datensammlung an:

```
>>> df.loc[0, 'review'][-50:]
'is seven.<br /><br />Title (Brazil): Not Available'
```

Wie Sie sehen, enthält der Text sowohl HTML-Code als auch Satzzeichen sowie weitere Sonderzeichen, die keine Buchstaben sind. Der HTML-Code enthält keine verwertbaren Informationen, aber Satzzeichen können bei der Verarbeitung natürlicher Sprache unter bestimmten Umständen wertvolle zusätzliche Informationen liefern. Der Einfachheit halber entfernen wir sämtliche Satzzeichen – mit Ausnahme von Emoticons wie »:)«, da diese bei einer Analyse der Stimmungslage zweifelsohne nützlich sind. Um diese Aufgabe zu erledigen, bedienen wir uns regulärer Ausdrücke und verwenden Pythons Regex-Bibliothek `re`:

```
>>> import re
>>> def preprocessor(text):
...     text = re.sub('<[^>]*>', '', text)
...     emoticons =
...         re.findall('(?:.||=)(?:-)?(?:\)|\(|D|P)', text)
...     text = re.sub('[\W]+', ' ', text.lower()) +
...             ''.join(emoticons).replace('-', '')
...     return text
```

Mit dem ersten regulären Ausdruck `<[^>]*>` versuchen wir, den gesamten HTML-Code aus den Filmbewertungen zu entfernen. Zwar raten viele Programmierer im Allgemeinen davon ab, HTML mit regulären Ausdrücken zu *parsen*; um diese spezielle Datensammlung zu *bereinigen*, sollte der reguläre Ausdruck aber ausreichen. Nach dem Entfernen der HTML-Auszeichnungen verwenden wir einen etwas komplizierteren regulären Ausdruck für die Suche nach Emoticons, die wir temporär in der Variablen `emoticons` speichern. Dann entfernen wir mit dem Regex `[\W]+` alle Zeichen, die keine Wörter sind, und wandeln den Text in Kleinbuchstaben um.

**Tipp**

Im Rahmen dieser Analyse gehen wir davon aus, dass die Großschreibung eines Wortes, beispielsweise am Satzanfang, semantisch nicht relevant ist. Es gibt jedoch Ausnahmen, wie etwa die Schreibweise von Eigennamen. Aber wie erwähnt, gehen wir hier von der vereinfachenden Annahme aus, dass die Groß-/ Kleinschreibung für die Stimmungsanalyse nicht von Bedeutung ist.

Schließlich fügen wir die zwischengespeicherten Emoticons am Ende der verarbeiteten Zeichenkette wieder hinzu. Außerdem entfernen wir das Zeichen für die »Nase« der Emoticons (-), um sie zu vereinheitlichen.

**Tipp**

Reguläre Ausdrücke bieten zwar eine effiziente und bequeme Möglichkeit, um in Texten nach Zeichen zu suchen, ihre Anwendung ist aber auch nicht ganz einfach zu erlernen. Eine ausführliche Erläuterung dieses Themas geht über den Rahmen dieses Buches hinaus, daher verweise ich auf eine ausgezeichnete Einführung in die Welt der regulären Ausdrücke, die auf dem Entwicklerportal von Google unter <https://developers.google.com/edu/python/regular-expressions> verfügbar ist. Oder lesen sie die offizielle Dokumentation des Python-Moduls `re` unter <https://docs.python.org/3.6/library/re.html>.

Das Hinzufügen der Emoticons am Ende des bereinigten Dokuments mag vielleicht nicht der eleganteste Ansatz sein, allerdings spielt die Reihenfolge der Wörter im Bag-of-words-Modell keine Rolle, sofern das Vokabular nur aus einzelnen Wörtern besteht. Bevor wir uns mit der Aufteilung von Dokumenten in einzelne Ausdrücke, Wörter oder Tokens befassen, sollten wir uns jedoch zunächst davon überzeugen, dass der Präprozessor korrekt arbeitet:

```
>>> preprocessor(df.loc[0, 'review'][-50:])
'is seven title brazil not available'
>>> preprocessor("</a>This :) is :( a test :-)!")
'this is a test :) :( :)'
```

Da wir in den verbleibenden Abschnitten immer wieder die *bereinigten* Daten benutzen werden, wenden wir die `preprocessor`-Funktion auf sämtliche Filmbewertungen im DataFrame an:

```
>>> df['review'] = df['review'].apply(preprocessor)
```

### 8.2.4 Dokumente in Token zerlegen

Nach der erfolgreichen Aufbereitung der Filmbewertungsdatensammlung müssen wir uns nun darüber Gedanken machen, wie die Textkorpora in einzelne Elemente aufgeteilt werden. Eine Möglichkeit der *Tokenisierung* von Dokumenten besteht darin, den bereinigten Text an den Stellen in einzelne Wörter aufzuteilen, an denen sich *Whitespace* (Leerraum) befindet:

```
>>> def tokenizer(text):
...     return text.split()
>>> tokenizer('runners like running and thus they run')
['runners', 'like', 'running', 'and', 'thus', 'they', 'run']
```

Ein weiteres nützliches Verfahren ist das sogenannte *Stemming* (Stammformreduktion), bei dem die verschiedenen Varianten eines Wortes auf dessen Stammform zurückgeführt werden. Der ursprüngliche Stemming-Algorithmus wurde 1979 von Martin F. Porter entwickelt und ist unter der Bezeichnung *Porter-Stemmer-Algorithmus* bekannt (Martin F. Porter, *An algorithm for suffix stripping*, Program: electronic library and information systems, 14(3):130-137, 1980). Das *Natural Language Toolkit (NLTK)* für Python (<http://www.nltk.org>) implementiert den Porter-Stemmer-Algorithmus, den wir in den folgenden Abschnitten verwenden werden. Zur Installation des NLTK brauchen Sie lediglich den Befehl `conda install nltk` oder `pip install nltk` einzugeben.

Der folgende Code demonstriert die Verwendung des Porter-Stemmer-Algorithmus:

```
>>> from nltk.stem.porter import PorterStemmer
>>> porter = PorterStemmer()
>>> def tokenizer_porter(text):
...     return [porter.stem(word) for word in text.split()]
>>> tokenizer_porter('runners like running
...                     and thus they run')
['runner', 'like', 'run', 'and', 'thu', 'they', 'run']
```

#### Tipp

Das NLTK ist zwar nicht Thema dieses Kapitels, aber wenn Sie an erweiterten Anwendungen der Verarbeitung natürlicher Sprache interessiert sind, empfehle ich, die NLTK-Website aufzusuchen und einen Blick in das offizielle NLTK-Buch zu werfen, das unter <http://www.nltk.org/book/> kostenlos zur Verfügung steht.

Mit dem `PorterStemmer` des NLTK-Pakets haben wir die `tokenizer`-Funktion modifiziert, die nun die Stammform von Wörtern zurückgibt – wie das einfache Beispiel zeigt, in dem das Wort `running` auf die Stammform `run` zurückgeführt wird.

**Tipp**

Der Porter-Stemmer-Algorithmus dürfte wohl der älteste und einfachste Stemming-Algorithmus sein. Zu den weiteren verbreiteten Stemming-Algorithmen gehören unter anderem der jüngere *Snowball-Stemmer* (Porter2 oder »Englisch-Stemmer) und der *Lancaster-Stemmer* (Paice-Husk-Stemmer), der schneller, aber auch aggressiver als der Porter-Stemmer ist. Diese alternativen Stemming-Algorithmen sind ebenfalls über das NLTK-Paket verfügbar (<http://www.nltk.org/api/nltk.stem.html>).

Bei der Stammformreduktion kommt es manchmal zu nicht existierenden Wörtern, wie im vorangegangenen Beispiel etwa zu `thu` (für `thus`). Ein Verfahren, das als *Lemmatisierung* bezeichnet wird, zielt darauf ab, die kanonische (grammatikalisch korrekte) Form einzelner Wörter zu ermitteln, die sogenannten *Lemmata*. Allerdings ist die Lemmatisierung im Vergleich zum Stemming rechnerisch schwieriger und aufwendiger, und in der Praxis hat sich gezeigt, dass beide Verfahren nur geringen Einfluss auf die Leistungsstärke der Textklassifizierung besitzen (Michal Toman, Roman Tesar und Karel Jezek, *Influence of word normalization on text classification*, Proceedings of InSciT, Seiten 354-358, 2006).

Bevor wir mit dem nächsten Abschnitt fortfahren, in dem wir ein Bag-of-words-Lernmodell trainieren werden, müssen wir uns noch kurz dem Entfernen sogenannter *Stoppwörter* widmen. Stoppwörter sind einfach die in allen möglichen Textarten extrem häufig vorkommenden Wörter, die keinerlei (oder nur sehr wenige) nützliche oder zur Klassifizierung von Dokumenten brauchbare Informationen enthalten. Beispiele für englische Stoppwörter sind etwa »is«, »and«, »has« und »like«. Das Entfernen der Stoppwörter erweist sich als nützlich, wenn man es nicht mit Tf-idf-Maßen, die häufige Wörter bereits geringer gewichtet, sondern mit einfachen oder normierten Vorkommenshäufigkeiten zu tun hat.

Zum Entfernen der Stoppwörter aus den Filmbewertungen verwenden wir eine aus 127 englischen Wörtern bestehende Stoppwörterliste, die über die NLTK-Bibliothek zur Verfügung steht und mit der `nltk.download`-Funktion heruntergeladen werden kann:

```
>>> import nltk  
>>> nltk.download('stopwords')
```

Nach dem Herunterladen der Liste können wir sie folgendermaßen verwenden:

```
>>> from nltk.corpus import stopwords
>>> stop = stopwords.words('english')
>>> [w for w in tokenizer_porter('a runner likes running
... and runs a lot')[-10:] if w not in stop]
['runner', 'like', 'run', 'run', 'lot']
```

## 8.3 Ein logistisches Regressionsmodell für die Dokumentklassifizierung trainieren

In diesem Abschnitt werden wir ein logistisches Regressionsmodell dafür trainieren, die Filmbewertungen als positiv oder negativ zu klassifizieren. Zunächst einmal teilen wir die bereinigten Textdokumente in zwei Gruppen mit jeweils 25.000 Dokumenten zum Trainieren und Testen auf:

```
>>> X_train = df.loc[:25000, 'review'].values
>>> y_train = df.loc[:25000, 'sentiment'].values
>>> X_test = df.loc[25000:, 'review'].values
>>> y_test = df.loc[25000:, 'sentiment'].values
```

Nun verwenden wir ein `GridSearch`-Objekt, um mit einer 5-fachen stratifizierten Kreuzvalidierung die optimale Parameterkombination für das logistische Regressionsmodell zu finden:

```
>>> from sklearn.model_selection import GridSearchCV
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.feature_extraction.text
...                                import TfidfVectorizer
>>> tfidf = TfidfVectorizer(strip_accents=None,
...                         lowercase=False,
...                         preprocessor=None)
>>> param_grid = [{'vect__ngram_range': [(1,1)],
...                 'vect__stop_words': [stop, None],
...                 'vect__tokenizer': [tokenizer,
...                                    tokenizer_porter],
...                 'clf__penalty': ['l1', 'l2'],
...                 'clf__C': [1.0, 10.0, 100.0}],
...                {'vect__ngram_range': [(1,1)],
...                 'vect__stop_words': [stop, None],
...                 'vect__tokenizer': [tokenizer,
...                                    tokenizer_porter]},
```

```

...
    'vect__use_idf':[False],
...
    'vect__norm':[None],
...
    'clf__penalty': ['l1', 'l2'],
...
    'clf__C': [1.0, 10.0, 100.0]
...
}
>>> lr_tfidf = Pipeline([('vect', tfidf),
...
    ('clf',
     LogisticRegression(random_state=0))])
>>> gs_lr_tfidf = GridSearchCV(lr_tfidf, param_grid,
...
    scoring='accuracy',
...
    cv=5, verbose=1,
...
    n_jobs=1)
>>> gs_lr_tfidf.fit(X_train, y_train)

```

## Hinweis

Es empfiehlt sich, in diesem Codebeispiel `n_jobs=-1` (statt `n_jobs=1`) zu verwenden, um alle verfügbaren Prozessorkerne zu nutzen und so die Rastersuche zu beschleunigen. Einige Windows-Benutzer haben jedoch von Problemen mit dieser Einstellung berichtet, die im Zusammenhang mit dem `pickle`-Modul und den Funktionen `tokenizer` und `tokenizer_porter` auftreten. Man könnte das Problem umgehen, indem man diese Funktionen durch `str.split` ersetzt, allerdings wird in diesem Fall das Stemming nicht unterstützt.

Bei der Initialisierung des `GridSearch`-Objekts und der dazugehörigen Parameter beschränken wir uns auf eine begrenzte Zahl von Parameterkombinationen, denn sowohl die Anzahl der Merkmalsvektoren als auch das umfangreiche Vokabular kann die Rastersuche ziemlich rechenintensiv machen – auf einem normalen Computer kann es bis zu 40 Minuten dauern, bis der Rechenvorgang abgeschlossen ist.

Im Beispielcode wurden der `CountVectorizer` und der `TfidfTransformer` aus dem vorangegangenen Abschnitt durch den `TfidfVectorizer` ersetzt, der beide Funktionalitäten in sich vereint. Das `param_grid` besteht aus zwei Parameter-Dictionaries: Im ersten verwenden wir zur Berechnung der Tf-idf-Maße die Standardeinstellungen (`use_idf=True`, `smooth_idf=True` und `norm='l2'`), im zweiten setzen wir diese Parameter auf `use_idf=False`, `smooth_idf=False` und `norm=None`, um ein Modell anhand der einfachen Vorkommenshäufigkeiten zu trainieren. Für die logistische Regression selbst trainieren wir Modelle unter Verwendung von L1- und L2-Regularisierung mit dem Strafparameter und vergleichen verschiedene Regularisierungsstärken durch Angabe eines Wertebereichs für den Kehrwert des Regularisierungsparameters C.

Nach Abschluss der Rastersuche können wir die beste Parameterkombination ausgeben:

```
>>> print('Beste Parameterkombination:
... %s' % gs_lr_tfidf.best_params_)
Beste Parameterkombination: {'clf__C': 10.0,
'vect__stop_words': None, 'clf__penalty': 'l2',
'vect__tokenizer': <function tokenizer at 0x7f6c704948c8>,
'vect__ngram_range': (1, 1)}
```

Das beste Ergebnis der Rastersuche erzielen wir also mit dem regulären Tokenizer ohne Porter-Stemming, ohne Stopwörterliste und mit Tf-idf-Maßen in Kombination mit einer logistischen Regression, die eine L2-Regularisierung mit dem Regularisierungsparameter  $C=10.0$  nutzt.

Unter Verwendung des besten von der Rastersuche gefundenen Modells geben wir nun den durchschnittlichen Korrektklassifizierungsraten-Score der 5-fachen Kreuzvalidierung (KV) für die Trainingsdatenmenge und die Korrektklassifizierungsrate für die Testdatenmenge aus:

```
>>> print('KV-Korrektklassifizierungsrate: %.3f'
...     % gs_lr_tfidf.best_score_)
KV-Korrektklassifizierungsrate: 0.892
>>> clf = gs_lr_tfidf.best_estimator_
>>> print('Korrektklassifizierungsrate Test: %.3f'
...     % clf.score(X_test, y_test))
Korrektklassifizierungsrate Test: 0.899
```

Das Ergebnis zeigt, dass unser Lernmodell mit einer Korrektklassifizierungsrate von 90 Prozent vorhersagen kann, ob eine Filmbewertung positiv oder negativ ist.

## Tipp

Ein nach wie vor sehr beliebter Textklassifizierer ist der *naive Bayes-Klassifikator*, der als Spamfilter Bekanntheit erlangte. Ein naiver Bayes-Klassifikator ist leicht zu implementieren, schnell zu berechnen und funktioniert im Vergleich zu anderen Algorithmen tendenziell besonders gut, sofern es um relativ kleine Datensets geht. Auch wenn der naive Bayes-Klassifikator in diesem Buch keine Rolle spielt, steht mein Artikel über die naive Textklassifizierung dem interessierten Leser auf der arXiv-Website kostenlos zum Nachlesen zur Verfügung (S. Raschka, *Naive Bayes and Text Classification I – Introduction and Theory*. Computing Research Repository (CoRR), abs/1410.5329, 2014, <http://arxiv.org/pdf/1410.5329v3.pdf>).

## 8.4 Verarbeitung großer Datenmengen: Online-Algorithmen und Out-of-Core Learning

Wenn Sie die Codebeispiele in den vorangegangenen Abschnitten ausgeführt haben, wird Ihnen vermutlich aufgefallen sein, dass die Konstruktion der Merkmalsvektoren für 50.000 Filmbewertungen bei der Rastersuche ziemlich rechenintensiv ist. Bei vielen praktischen Anwendungen ist es nicht ungewöhnlich, dass sogar noch größere Datenmengen verarbeitet werden, die nicht vollständig in den Arbeitsspeicher des Rechners geladen werden können. Und da wir allerdings normalerweise keinen Zugriff auf einen Supercomputer haben, werden wir nun ein Verfahren namens *Out-of-Core Learning* anwenden, das es ermöglicht, mit derart großen Datenmengen zu arbeiten, indem der Klassifizierer schrittweise an kleinere Teilmengen der Daten angepasst wird.

In Kapitel 2 haben Sie das Konzept des stochastischen Gradientenabstiegsverfahrens kennengelernt, einem Optimierungsalgorithmus, der die Gewichtungen eines Modells ermittelt, indem die Objekte der Reihe nach verarbeitet werden. In diesem Abschnitt werden wir die `partial_fit`-Funktion des `SGDClassifier` von scikit-learn verwenden, um die Dokumente direkt vom lokalen Laufwerk einzulesen und ein logistisches Regressionsmodell mit kleinen Teilmengen der Dokumente zu trainieren.

Zunächst einmal definieren wir eine `tokenizer`-Funktion zur Bereinigung der unverarbeiteten Textdaten in der Datei `movie_data.csv`, die wir am Anfang des Kapitels erstellt haben. Sie entfernt die Stopwörter und erzeugt Wörter-Tokens:

```
>>> import numpy as np
>>> import re
>>> from nltk.corpus import stopwords
>>> stop = stopwords.words('english')
>>> def tokenizer(text):
...     text = re.sub('<[^>]*>', '', text)
...     emoticons =
...         re.findall('(?:[:|;|=](?:-)?(?:\:)|\\(|D|P)', text.lower())
...     text = re.sub('[\W]+', ' ', text.lower()) \
...         + ''.join(emoticons).replace('-', '')
...     tokenized =
...         [w for w in text.split() if w not in stop]
...     return tokenized
```

Nun definieren wir die Generatorfunktion `stream_docs`, die Dokumente der Reihe nach einliest und einzeln zurückliefert:

```
>>> def stream_docs(path):
...     with open(path, 'r', encoding='utf-8') as csv:
...         next(csv) # Überschrift überspringen
...         for line in csv:
...             text, label = line[:-3], int(line[-2])
...             yield text, label
```

Um zu überprüfen, ob die `stream_docs`-Funktion korrekt funktioniert, lesen wir das erste in der Datei `movie_data.csv` enthaltene Dokument ein, das wiederum ein Tupel umfassen sollte, das aus dem Text der Filmbewertung und der zugehörigen Klassenbezeichnung besteht:

```
>>> next(stream_docs(path='movie_data.csv'))
('In 1974, the teenager Martha Moxley ... ', 1)
```

Nun definieren wir die Funktion `get_minibatch`, die mithilfe der `stream_docs`-Funktion eine durch den Parameter `size` festgelegte Anzahl von Dokumenten zurückgibt:

```
>>> def get_minibatch(doc_stream, size):
...     docs, y = [], []
...     try:
...         for _ in range(size):
...             text, label = next(doc_stream)
...             docs.append(text)
...             y.append(label)
...     except StopIteration:
...         return None, None
...     return docs, y
```

Leider können wir für das Out-of-Core Learning nicht den `CountVectorizer` verwenden, denn dazu wäre es erforderlich, das gesamte Vokabular in den Arbeitsspeicher zu laden. Außerdem muss auch der `TfidfVectorizer` sämtliche Merkmalsvektoren der Trainingsdatenmenge in den Arbeitsspeicher laden, um die inverse Dokumenthäufigkeit zu berechnen. In scikit-learn ist jedoch eine weitere praktische Funktion zur Verarbeitung von Texten implementiert: der `HashingVectorizer`. Er ist von den Daten unabhängig und macht unter Einsatz des 32-Bit-MurmurHash3-Algorithmus von Austin Appleby (<https://sites.google.com/site/murmurhash/>) vom Hashing-Trick Gebrauch.

```
>>> from sklearn.feature_extraction.text
...                     import HashingVectorizer
>>> from sklearn.linear_model import SGDClassifier
>>> vect = HashingVectorizer(decode_error='ignore',
```

```

...
        n_features=2**21,
...
        preprocessor=None,
        tokenizer=tokenizer)
>>> clf = SGDClassifier(loss='log', random_state=1, n_iter=1)
>>> doc_stream = stream_docs(path='movie_data.csv')

```

Der Code initialisiert den `HashingVectorizer` mit der `tokenizer`-Funktion und setzt die Anzahl der Merkmale auf  $2^{21}$ . Darüber hinaus wird ein weiterer Klassifizierer (logistische Regression) initialisiert, indem der `loss`-Parameter des `SGDClassifier` den Wert `log` erhält. Durch die Auswahl einer sehr großen Merkmalsanzahl für den `HashingVectorizer` wird die Wahrscheinlichkeit einer Hash-Kollision verringert, allerdings erhöhen wir dadurch auch die Anzahl der Koeffizienten des logistischen Regressionsmodells. Nun kommen wir zum wirklich interessanten Teil. Nachdem alle ergänzenden Funktionen eingerichtet sind, können wir mit dem Out-of-Core Learning anfangen, und zwar mithilfe folgenden Codes:

```

>>> import pyprind
>>> pbar = pyprind.ProgBar(45)
>>> classes = np.array([0, 1])
>>> for _ in range(45):
...     X_train, y_train = get_minibatch(doc_stream, size=1000)
...     if not X_train:
...         break
...     X_train = vect.transform(X_train)
...     clf.partial_fit(X_train, y_train, classes=classes)
...     pbar.update()
0%                                100%
[########################################] | ETA[sec]: 0.000
Total time elapsed: 00:00:39

```

Hier machen wir wieder vom PyPrind-Paket Gebrauch, um den Fortschritt des Lernalgorithmus beobachten zu können: Wir initialisieren das Fortschrittsbalken-Objekt mit 45 Schritten und durchlaufen in der folgenden `for`-Schleife 45 kleine Teilmengen der Dokumente, die aus jeweils 1.000 Dokumenten bestehen.

Nach Abschluss des inkrementellen Lernens können wir die verbleibenden 5.000 Dokumente zur Performancebewertung des Modells verwenden:

```

>>> X_test, y_test = get_minibatch(doc_stream, size=5000)
>>> X_test = vect.transform(X_test)
>>> print('Korrektklassifizierungsrate: %.3f'
...                   % clf.score(X_test, y_test))
Korrektklassifizierungsrate: 0.878

```

Die Korrektklassifizierungsrate des Modells beträgt knapp 88 Prozent, etwas weniger als die Korrektklassifizierungsrate, die wir im vorangegangenen Abschnitt mit der Rastersuche zur Hyperparameter-Abstimmung erzielt haben. Allerdings ist das Out-of-Core Learning äußerst spechereffizient und benötigte weniger als eine Minute Rechenzeit. Zum Abschluss verwenden wir die letzten 5.000 Dokumente, um unser Modell zu aktualisieren:

```
>>> clf = clf.partial_fit(X_test, y_test)
```

Falls Sie beabsichtigen, sogleich mit Kapitel 9 fortzufahren, sollten Sie die aktuelle Python-Sitzung nicht beenden. Im nächsten Kapitel werden wir anhand des soeben trainierten Modells vorführen, wie man es zwecks späterer Wiederverwendung speichern und in eine Webanwendung einbetten kann.

### Tipp

Eine modernere Alternative zum Bag-of-words-Modell ist *word2vec*, ein 2013 von Google veröffentlichter Algorithmus (T. Mikolov, K. Chen, G. Corrado und J. Dean, *Efficient Estimation of Word Representations in Vector Space*, arXiv preprint arXiv:1301.3781, 2013). word2vec ist ein auf neuronalen Netzen beruhender unüberwachter Lernalgorithmus, der versucht, automatisch die Beziehung zwischen Wörtern zu erkennen. Zu diesem Zweck werden Wörter mit vergleichbarer Bedeutung zu Ähnlichkeits-Clustern zusammengefasst. Durch eine clevere Verwendung von Vektorräumen ist das Modell dann in der Lage, bestimmte Wörter durch einfache Vektorberechnungen nachzubilden, beispielsweise *Knecht – Mann + Frau = Magd*.

Die ursprüngliche Implementierung in C, die nützliche Hinweise auf die relevanten Veröffentlichungen und alternative Implementierungen enthält, ist unter <https://code.google.com/p/word2vec/> zu finden.

## 8.5 Topic Modeling mit latenter Dirichlet-Allokation

Der Begriff *Topic Modeling* beschreibt die umfassende Aufgabe, ungekennzeichnete Textdokumente einem Themenbereich zuzuordnen. Eine typische Anwendung wäre beispielsweise die Kategorisierung von Dokumenten eines großen Textkorpus von Zeitungsartikeln, von denen nicht bekannt ist, auf welcher Seite oder in welcher Rubrik sie erschienen sind. Das Topic Modeling hat zum Ziel, den Artikeln Kategorien zuzuweisen – z.B. Sport, Finanzen, Auslandsnachrichten, Politik, lokale Nachrichten und so weiter. In dem in Kapitel 1 erörterten Kontext der Kategorien des Machine Learnings ist das Topic Modeling als Clustering-Aufgabe einzurordnen; es handelt sich um eine Unterkategorie des unüberwachten Lernens.

In diesem Abschnitt betrachten wir ein verbreitetes Topic-Modeling-Verfahren, das *latente Dirichlet-Allokation* heißt, die zwar ebenfalls mit LDA abgekürzt wird, aber nicht mit der linearen Diskriminanzanalyse zu verwechseln ist, einem überwachten Verfahren zur Dimensionsreduktion, das in Kapitel 5 vorgestellt wurde.

**Tipp**

Die LDA unterscheidet sich von dem überwachten Lernansatz, den wir in diesem Kapitel bei der Klassifizierung von Filmbewertungen als positiv oder negativ verfolgt haben. Wenn Sie vor allem daran interessiert sind, scikit-learn-Modelle mit Flask in einer Webanwendung einzubetten, können Sie das anhand des Beispiels der Filmbewertung in Kapitel 9 nachlesen und später zu diesem davon unabhängigen Abschnitt zum Thema Topic Modeling zurückkehren.

### 8.5.1 Aufteilung von Texten mit der LDA

Die der LDA zugrunde liegende Mathematik ist relativ kompliziert und erfordert Kenntnisse Bayes'scher Verfahren, daher werden wir das Thema aus Sicht eines Praktikers angehen und die LDA durch vereinfachende Begriffe interpretieren. Der interessierte Leser findet weitere Informationen zur LDA in folgendem Artikel: D.M. Blei, A.Y. Ng und M.I. Jordan, *Latent Dirichlet Allocation*, The Journal of Machine Learning Research, 3:993-1022, 2003.

Die LDA ist ein generatives probabilistisches Modell, das versucht, Wortgruppen aufzuspüren, die in verschiedenen Dokumenten häufig zusammen vorkommen. Diese häufig auftretenden Wörter repräsentieren die Themen, sofern alle Dokumente aus einem Gemisch unterschiedlicher Wörter bestehen. Die Eingabe einer LDA ist das in diesem Kapitel bereits erörterte Bag-of-words-Modell. Die LDA unterteilt eine gegebene Bag-of-words-Matrix in zwei neue Matrizen:

- eine Dokument/Themen-Matrix und
- eine Wort/Themen-Matrix.

Die Unterteilung der Bag-of-words-Matrix durch die LDA findet auf eine Art und Weise statt, die es ermöglicht, durch Multiplikation der beiden Matrizen die ursprüngliche Bag-of-words-Matrix mit kleinstmöglichen Fehler zurückzugewinnen. In der Praxis sind wir an den Themen interessiert, die von der LDA in der Bag-of-words-Matrix aufgefunden werden. Nachteilig ist dabei nur, dass die Anzahl der Themen im Vorhinein festgelegt werden muss – die Themenanzahl ist ein Hyperparameter der LDA, der manuell anzugeben ist.

### 8.5.2 LDA mit scikit-learn

In diesem Abschnitt werden wir die in scikit-learn implementierte `LatentDirichletAllocation`-Klasse zur Aufteilung der Filmbewertungsdatenbank verwenden

und die Filme verschiedenen Themengebieten zuordnen. Im folgenden Beispiel beschränken wir die Analyse auf 10 verschiedene Themen, aber experimentieren Sie ruhig ein wenig mit den Hyperparametern des Algorithmus herum, um weitere Themen aufzuspüren, die sich in der Datensammlung finden lassen.

Als Erstes laden wir die Filmbewertungen aus der am Anfang des Kapitels erstellten lokalen Datei *movie\_data.csv* in einen pandas-DataFrame:

```
>>> import pandas as pd
>>> df = pd.read_csv('movie_data.csv', encoding='utf-8')
```

Als Nächstes verwenden wir den inzwischen wohlbekannten CountVectorizer, um eine Bag-of-words-Matrix als Eingabe für die LDA zu erzeugen. Der Bequemlichkeit halber verwenden wir die in scikit-learn integrierte Stopwörterliste:

```
>>> from sklearn.feature_extraction.text import
...                               CountVectorizer
>>> count = CountVectorizer(stop_words='english',
...                           max_df=.1,
...                           max_features=5000)
>>> X = count.fit_transform(df['review'].values)
```

Beachten Sie, dass die maximale Dokumenthäufigkeit von Wörtern auf 10 Prozent gesetzt wird (`max_df=.1`), um Wörter auszuschließen, die in den Dokumenten zu häufig vorkommen, weil es sich dabei vermutlich um gängige Wörter handelt, die in allen Dokumenten vorhanden sind und somit kaum einem bestimmten Themengebiet eines gegebenen Dokuments zuzuordnen sind. Außerdem beschränken wir die Anzahl der zu berücksichtigenden Wörter auf die 5.000 am häufigsten vorkommenden (`max_features=5000`), um die Dimensionalität der Datenmenge zu begrenzen, sodass die Leistung der LDA verbessert wird. Allerdings sind sowohl `max_df=.1` als auch `max_features=5000` willkürlich gewählte Werte der Hyperparameter, und Sie sollten diese ruhig ein wenig ändern und die Ergebnisse vergleichen.

Der folgende Beispielcode demonstriert die Anpassung des LatentDirichletAllocation-Schätzers an die Bag-of-words-Matrix und leitet aus den Dokumenten 10 verschiedene Themengebiete ab (der Vorgang kann auf einem normalen Laptop oder PC fünf Minuten oder länger in Anspruch nehmen):

```
>>> from sklearn.decomposition import
...                               LatentDirichletAllocation
>>> lda = LatentDirichletAllocation(n_topics=10,
...                                   random_state=123,
...                                   learning_method='batch')
>>> X_topics = lda.fit_transform(X)
```

Durch die Einstellung `learning_method='batch'` weisen wir den Schätzer an, seine Arbeit mit allen verfügbaren Daten (der Bag-of-words-Matrix) in einem Durchlauf zu erledigen, was langsamer ist als die alternative Lernmethode '`online`', aber zu genaueren Ergebnissen führen kann. (Die Einstellung `learning_method='online'` entspricht dem Online-Lernen bzw. dem Lernen als Mini-Stapelverarbeitung, das in Kapitel 2 und in diesem Kapitel erörtert wurde.)

**Tipp**

Die LDA-Implementierung der scikit-learn-Bibliothek verwendet den sogenannten Expectation-Maximization-Algorithmus (EM) zur Aktualisierung der iterativen Parameterabschätzung. Der EM-Algorithmus kam in diesem Kapitel nicht zur Sprache, aber wenn Sie mehr erfahren möchten, sollten Sie sich den ausgezeichneten Überblick der Wikipedia ansehen ([https://en.wikipedia.org/wiki/Expectation-maximization\\_algorithm](https://en.wikipedia.org/wiki/Expectation-maximization_algorithm)) und das ausführliche Tutorial der Verwendung des EM-Algorithmus bei der LDA von Colorado Reed lesen (*Latent Dirichlet Allocation: Towards a Deeper Understanding*), das kostenlos unter [http://obphio.us/pdfs/lda\\_tutorial.pdf](http://obphio.us/pdfs/lda_tutorial.pdf) verfügbar ist.

Nach der Anpassung der LDA können wir auf das `components`-Attribut der `lda`-Instanz zugreifen, in dem eine Matrix gespeichert ist, die die Bedeutung der Wörter (hier 5000) für die 10 Themengebiete in aufsteigender Reihenfolge enthält:

```
>>> lda.components_.shape  
(10, 5000)
```

Zwecks Analyse des Ergebnisses geben wir die fünf bedeutsamsten Wörter der 10 Themengebiete aus. Beachten Sie, dass die Bedeutungswerte der Wörter in aufsteigender Reihenfolge aufgeführt sind. Um die fünf wichtigsten Wörter anzugeben, müssen wir das `topic`-Array also in umgekehrter Reihenfolge sortieren:

```
>>> n_top_words = 5  
>>> feature_names = count.get_feature_names()  
>>> for topic_idx, topic in enumerate(lda.components_):  
...     print("Topic %d:" % (topic_idx + 1))  
...     print(" ".join([feature_names[i]  
...                     for i in topic.argsort()\br/>...                     [-n_top_words - 1:-1]]))  
  
Topic 1:  
worst minutes awful script stupid  
Topic 2:  
family mother father children girl  
Topic 3:
```

```

american war dvd music tv
Topic 4:
human audience cinema art sense
Topic 5:
police guy car dead murder
Topic 6:
horror house sex girl woman
Topic 7:
role performance comedy actor performances
Topic 8:
series episode war episodes tv
Topic 9:
book version original read novel
Topic 10:
action fight guy guys cool

```

Anhand der fünf bedeutsamsten Wörter der verschiedenen Themengebiete hat die LDA offenbar folgende Themen aufgefunden:

1. Allgemein schlechte Filme (genau genommen kein Themengebiet)
2. Filme über Familien
3. Kriegsfilme
4. Kunstmilme
5. Kriminalmilme
6. Horrorfilme
7. Komödien
8. Filme, die in irgendeinem Zusammenhang mit Fernsehshows stehen
9. Verfilmungen von Büchern
10. Actionmilme

Um zu überprüfen, ob die auf den Filmbewertungen beruhende Kategorisierung einen Sinn ergibt, zeigen wir drei Filme der Rubrik Horrorfilm an (Kategorie 6 an der Indexposition 5):

```

>>> horror = X_topics[:, 5].argsort()[:-1]
>>> for iter_idx, movie_idx in enumerate(horror[:3]):
...     print('\nHorrorfilm #%d: % (iter_idx + 1)')
...     print(df['review'][movie_idx][:300], '...')
Horrorfilm #1:
House of Dracula works from the same basic premise as House
of Frankenstein from the year before; namely that Universal's
three most famous monsters; Dracula, Frankenstein's Monster and The Wolf
Man are appearing in the movie together.

```

```
Naturally, the film is rather messy therefore, but the ...
```

Horrorfilm #2:

Okay, what the hell kind of TRASH have I been watching now?  
"The Witches' Mountain" has got to be one of the most  
incoherent and insane Spanish exploitation flicks ever and  
yet, at the same time, it's also strangely compelling.  
There's absolutely nothing that makes sense here  
and I even doubt there ...

Horrorfilm #3:

```
<br /><br />Horror movie time, Japanese style. Uzumaki/Spiral  
was a total freakfest from start to finish. A fun freakfest  
at that, but at times it was a tad too reliant on kitsch  
rather than the horror. The story is difficult to summarize  
succinctly: a carefree, normal teenage girl starts coming ...
```

Mit dem obigen Code haben wir die ersten 300 Zeichen der obersten drei Horrorfilme angezeigt und die Bewertungen klingen tatsächlich nach Horrorfilmen, wenngleich wir nicht wissen, zu welchem Film sie eigentlich genau gehören. Man könnte allerdings auch argumentieren, dass der zweite Horrorfilm besser zur Kategorie 1 passt, den allgemein schlechten Filmen.

## 8.6 Zusammenfassung

In diesem Kapitel haben Sie erfahren, wie man Lernalgorithmen für die Klassifizierung von Textdokumenten anhand ihrer Tonalität einsetzen kann – eine der grundlegenden Aufgaben in der Stimmungsanalyse im Bereich der Verarbeitung natürlicher Sprache. Sie haben nicht nur gelernt, wie man mithilfe des Bag-of-words-Modells Dokumente in Merkmalsvektoren umwandeln kann, sondern darüber hinaus auch erfahren, wie die Vorkommenshäufigkeiten durch Tf-idf-Maße anhand ihrer Relevanz gewichtet werden.

Die Verarbeitung von Textdaten kann aufgrund der bei diesem Vorgang erstellten großen Merkmalsvektoren sehr rechenintensiv sein. Im letzten Abschnitt haben Sie erfahren, wie man einen Lernalgorithmus durch inkrementelles oder Out-of-Core Learning trainieren kann, ohne die gesamte Datenmenge in den Arbeitsspeicher eines Computers zu laden.

Und schließlich haben wir das Konzept des Topic Modeling mit einer LDA zur unüberwachten Kategorisierung von Filmbewertungen vorgestellt.

Im nächsten Kapitel werden wir den Dokumentenklassifizierer einsetzen und ihn in eine Webanwendung einbetten.

# Einbettung eines Machine-Learning-Modells in eine Webanwendung

In den vorangegangenen Kapiteln haben Sie viele verschiedene Lernkonzepte und Algorithmen kennengelernt, die eine bessere und effizientere Entscheidungsfindung ermöglichen. Die Verfahren des Machine Learnings sind aber keineswegs auf Offlineanwendungen oder -analysen beschränkt und können auch als Webdienste zur Vorhersage genutzt werden. Zu den verbreitetsten und nützlichsten Anwendungen von Lernmodellen in Form von Webdiensten gehören beispielsweise die Spamerkennung in übermittelten Webformularen, Suchmaschinen, Empfehlungssysteme in Medien- oder Einkaufsportalen und viele andere mehr.

Auf den folgenden Seiten werden Sie erfahren, wie Sie ein Lernmodell in eine Webanwendung einbetten, die Daten nicht nur in Echtzeit klassifiziert, sondern gleichzeitig daraus lernt.

Die Themen in diesem Kapitel sind:

- Speicherung des aktuellen Zustands eines trainierten Lernmodells
- Einsatz von SQLite-Datenbanken zum Speichern von Daten
- Entwicklung einer Webanwendung mit dem beliebten Flask-Framework
- Bereitstellung der Anwendung auf einem öffentlich zugänglichen Webserver

## 9.1 Serialisierung angepasster Schätzer mit scikit-learn

Wie Sie aus Kapitel 8 wissen, kann das Trainieren eines Lernmodells sehr rechenaufwendig sein. Natürlich wollen wir unser Modell nicht jedes Mal, wenn wir den Python-Interpreter beendet haben oder die Webanwendung erneut laden, wieder neu trainieren müssen, um eine Vorhersage zu treffen. Eine Option, den Modellzustand dauerhaft zu speichern, stellt das integrierte `pickle`-Modul dar, das es ermöglicht, Python-Objektstrukturen zu serialisieren und als kompakten Bytecode zu speichern, der wieder eingelesen und deserialisiert werden kann (<https://docs.python.org/3.6/library/pickle.html>). Auf diese Weise können wir unseren Klassifizierer im aktuellen Zustand speichern und dann später wieder einlesen, wenn wir neue Objekte klassifizieren möchten, ohne das Modell erneut mit sämtlichen Daten trainieren zu müssen. Vergewissern Sie sich, dass Sie das

logistische Out-of-Core-Regressionsmodell aus dem letzten Abschnitt von Kapitel 8 trainiert haben, damit es in der aktuellen Python-Sitzung verfügbar ist, bevor Sie den nachstehenden Code ausführen.

```
>>> import pickle  
>>> import os  
>>> dest = os.path.join('movieclassifier', 'pkl_objects')  
>>> if not os.path.exists(dest):  
...     os.makedirs(dest)  
>>> pickle.dump(stop,  
...             open(os.path.join(dest, 'stopwords.pkl'), 'wb'),  
...             protocol=4)  
>>> pickle.dump(clf,  
...             open(os.path.join(dest, 'classifier.pkl'), 'wb'),  
...             protocol=4)
```

Der Code erstellt das Verzeichnis `movieclassifier`, in dem wir später Dateien und Daten der Webanwendung speichern werden. Innerhalb dieses Verzeichnisses legen wir ein Unterverzeichnis `pkl_objects` zum Speichern der serialisierten Python-Objekte auf dem lokalen Laufwerk an. Mit der `pickle`-Methode `dump` serialisieren wir das trainierte logistische Regressionsmodell sowie die Stopwörterliste der NLTK(Natural Language Toolkit)-Bibliothek, damit wir das NLTK-Vokabular nicht auf dem Server installieren müssen.

Die `dump`-Methode erwartet als erstes Argument das zu serialisierende Objekt. Als zweites Argument übergeben wir ein geöffnetes Dateiobjekt, in dem das Python-Objekt gespeichert wird. Mit dem `wb`-Argument in der `open`-Funktion öffnen wir die Datei im binären Modus und setzen `protocol=4`, um das neueste und effizienteste `pickle`-Protokoll auszuwählen, das in Python 3.4 neu dazugekommen ist. (Falls Sie beim Einsatz des Protokolls 4 auf Schwierigkeiten stoßen, sollten Sie überprüfen, ob Sie wirklich die neueste Version von Python 3 nutzen. Alternativ können Sie auch die Verwendung einer niedrigeren Protokollversion in Betracht ziehen.)

### Tipp

Das logistische Regressionsmodell enthält mehrere NumPy-Arrays, wie beispielsweise den Gewichtungsvektor, die sich mithilfe der `joblib`-Bibliothek noch effizienter serialisieren lassen. Um jedoch die Kompatibilität mit der Serverumgebung zu gewährleisten, die wir später verwenden werden, kommt hier lediglich die Standardmethode mit `pickle` zum Einsatz. Weitere Informationen zu `joblib` finden Sie unter <https://pypi.python.org/pypi/joblib>.

Den `HashingVectorizer` brauchen wir nicht zu serialisieren, da er nicht angepasst werden muss. Stattdessen erstellen wir ein Python-Skript, mit dem wir ihn in die aktuelle Python-Sitzung importieren können. Kopieren Sie den folgenden Code und speichern Sie ihn in einer Datei namens `vectorizer.py` im `movie-classifier`-Verzeichnis ab:

```
from sklearn.feature_extraction.text import HashingVectorizer
import re
import os
import pickle

cur_dir = os.path.dirname(__file__)
stop = pickle.load(open(
    os.path.join(cur_dir,
    'pkl_objects',
    'stopwords.pkl'), 'rb'))

def tokenizer(text):
    text = re.sub('<[^>]*>', '', text)
    emoticons = re.findall('(?::|;|=)(?:-)?(?:\\()|\\(|D|P)',
                           text.lower())
    text = re.sub('[\\W]+', ' ', text.lower()) \
        + ' '.join(emoticons).replace('-', '')
    tokenized = [w for w in text.split() if w not in stop]
    return tokenized

vect = HashingVectorizer(decode_error='ignore',
                        n_features=2**21,
                        preprocessor=None,
                        tokenizer=tokenizer)
```

Nach der Serialisierung der Python-Objekte und dem Erstellen der Datei `vectorizer.py` sollten wir den Python-Interpreter (oder den IPython-Notebook-Kernel) neu starten, um zu überprüfen, ob sich die Objekte problemlos deserialisieren lassen.

### Tipp

Beachten Sie jedoch, dass die Deserialisierung von Daten, die nicht vertrauenswürdigen Quellen entstammen, ein potenzielles Sicherheitsrisiko darstellt, denn das `pickle`-Modul schützt Sie nicht vor bösartigem Code. Da das `pickle`-Modul zur Serialisierung beliebiger Objekte gedacht ist, wird bei der Deserialisierung der in einer `pickle`-Datei gespeicherte Code ausgeführt.

Wenn Sie also `pickle`-Dateien aus einer nicht vertrauenswürdigen Quelle verwenden (beispielsweise aus dem Internet heruntergeladene Dateien), sollten Sie besonders vorsichtig sein und die Deserialisierung in einer virtuellen Maschine und/oder auf einem verzichtbaren Rechner durchführen, auf dem keine wichtigen Daten gespeichert sind, auf die außer Ihnen selbst niemand zugreifen sollte.

Navigieren Sie im Terminal zum `movieclassifier`-Verzeichnis, starten Sie eine neue Python-Sitzung und führen Sie den folgenden Code aus, um sicherzustellen, dass der Import des `HashingVectorizers` und die Deserialisierung des Klassifizierers funktionieren.

```
>>> import pickle
>>> import re
>>> import os
>>> from vectorizer import vect
>>> clf = pickle.load(open(
...         os.path.join('pkl_objects',
...                 'classifier.pkl'), 'rb'))
```

Verlief das Laden des `HashingVectorizers` und die Deserialisierung des Klassifizierers ohne Probleme, können wir diese Objekte nun verwenden, um die Dokumente vorzuverarbeiten und Vorhersagen über die Bewertungen zu treffen:

```
>>> import numpy as np
>>> label = {0:'negativ', 1:'positiv'}
>>> example = ['I love this movie']
>>> X = vect.transform(example)
>>> print('Vorhersage: %s\nWahrscheinlichkeit: %.2f%%' %\
...         (label[clf.predict(X)[0]],
...          np.max(clf.predict_proba(X))*100))
Vorhersage: positiv
Wahrscheinlichkeit: 91.56%
```

Da der Klassifizierer die Klassenbezeichnungen als Ganzzahlen zurück liefert, definieren wir ein einfaches Python-Dictionary, das den Zahlen die jeweilige Bewertung zuordnet. Dann nutzen wir den `HashingVectorizer`, um das einfache Beispiel in einen Wortvektor `X` zu transformieren. Und schließlich verwenden wir die `predict`-Methode des logistischen Regressionsklassifizierers zur Vorhersage der Klassenbezeichnung und die `predict_proba`-Methode zur Ausgabe der zugehörigen Wahrscheinlichkeit der Vorhersage. Beachten Sie, dass die `predict_proba`-Methode ein Array mit Wahrscheinlichkeitswerten der verschiedenen Klassenbezeichnungen zurückgibt. Da die Klassenbezeichnung mit dem höchsten

Wahrscheinlichkeitswert der vom `predict`-Aufruf zurückgegebenen Klassenbezeichnung entspricht, verwenden wir die `np.max`-Funktion zur Ausgabe der Wahrscheinlichkeit der vorhergesagten Klasse.

## 9.2 Einrichtung einer SQLite-Datenbank zum Speichern von Daten

In diesem Abschnitt werden wir eine einfache SQLite-Datenbank einrichten, die optional Feedback zu den Vorhersagen der Webanwendung speichern soll. Dieses Feedback können wir dann zur Aktualisierung des Klassifizierungsmodells verwenden. SQLite ist eine quelloffene SQL-Datenbank, die keinen eigenen Server benötigt und damit ideal für kleinere Projekte und einfache Webanwendungen geeignet ist. Sie kann im Wesentlichen als eine einzelne, eigenständige Datenbankdatei betrachtet werden, auf die man direkt zugreifen kann.

Außerdem erfordert SQLite keine systemspezifische Konfiguration und ist für alle gängigen Betriebssysteme verfügbar. SQLite hat sich den Ruf erworben, sehr zuverlässig zu sein und wird von bekannten Unternehmen wie Google, Mozilla, Adobe, Apple, Microsoft und vielen anderen eingesetzt. Wenn Sie mehr über dieses Datenbanksystem erfahren möchten, empfehle ich den Besuch der offiziellen Website unter <http://www.sqlite.org>.

Entsprechend der Python-Philosophie, »*Batteries included*«, gibt es in der Standardbibliothek bereits eine API namens `sqlite3`, die uns die Verwendung von SQLite-Datenbanken ermöglicht. Weitere Informationen zu `sqlite3` finden Sie unter <https://docs.python.org/3.6/library/sqlite3.html>.

Der folgende Code erzeugt im Verzeichnis `movieclassifier` eine neue SQLite-Datenbank und speichert zwei Beispiele für Filmbewertungen:

```
>>> import sqlite3
>>> import os

>>> if os.path.exists('reviews.sqlite'):
...     os.remove('reviews.sqlite')
>>> conn = sqlite3.connect('reviews.sqlite')
>>> c = conn.cursor()
>>> c.execute('CREATE TABLE review_db' \
...            ' (review TEXT, sentiment INTEGER, date TEXT)')
>>> example1 = 'I love this movie'
>>> c.execute("INSERT INTO review_db" \
...            " (review, sentiment, date) VALUES" \
...            " (?, ?, DATETIME('now'))", (example1, 1))
>>> example2 = 'I disliked this movie'
>>> c.execute("INSERT INTO review_db" \
```

```

...           " (review, sentiment, date) VALUES" \
...           "(?, ?, DATETIME('now'))", (example2, 0))
>>> conn.commit()
>>> conn.close()

```

Zunächst einmal erstellen wir eine Verbindung `conn` zur SQLite-Datenbank, indem wir die `connect`-Methode von `sqlite3` aufrufen, die im Verzeichnis `movie-classifier` eine neue Datenbankdatei namens `reviews.sqlite` erzeugt, sofern diese noch nicht vorhanden ist. Beachten Sie, dass es in SQLite keine Funktion gibt, um bereits vorhandene Datenbanken zu ersetzen – Sie müssen die Datenbankdatei also löschen, wenn Sie den Code ein zweites Mal ausführen möchten. Als Nächstes erstellen wir mit der `cursor`-Methode einen Datenbankcursor, mit dem wir dank der flexiblen SQL-Syntax die Datensätze durchlaufen können. Mit dem ersten `execute`-Aufruf legen wir eine neue Datenbanktabelle namens `review_db` an, die wir zum Speichern und Auslesen von Datensätzen verwenden. Dabei werden drei Spalten eingerichtet: `review`, `sentiment` und `date`. Sie dienen zum Speichern der beiden Filmbewertungen und der zugehörigen Klassenbezeichnungen (`sentiment`). Mit dem SQL-Befehl `DATETIME('now')` fügen wir den Datensätzen außerdem einen Zeitstempel hinzu. Neben dem Zeitstempel übergeben wir der `execute`-Methode Fragezeichen (?) als Platzhaltersymbole für die Bewertungstexte (`example1` und `example2`) und die zugehörigen Klassenbezeichnungen (1 und 0), die Elemente eines Tupels sind. Und zum Abschluss rufen wir die `commit`-Methode auf, um die vorgenommenen Änderungen zu speichern, und schließen die Verbindung mit der `close`-Methode.

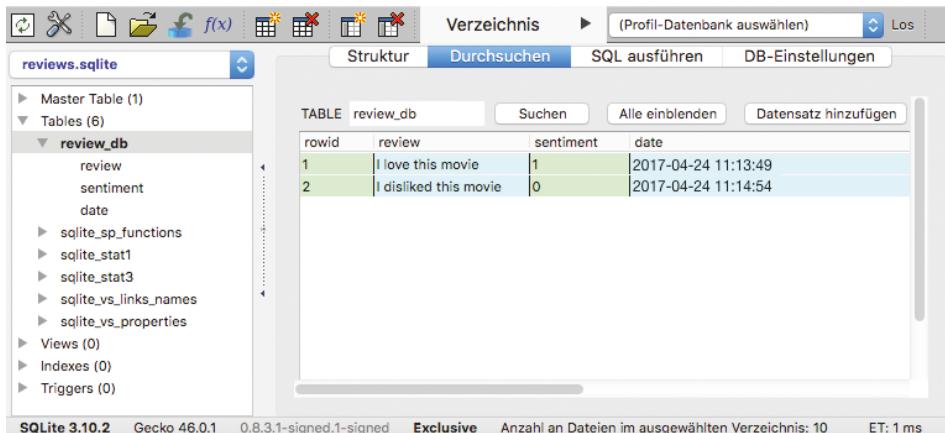
Um zu überprüfen, ob die Einträge in der Datenbanktabelle korrekt gespeichert wurden, öffnen wir erneut eine Verbindung zur Datenbank und rufen mit dem SQL-Befehl `select` alle zwischen Anfang 2017 und dem aktuellen Datum erstellten Zeilen der Datenbanktabelle ab:

```

>>> conn = sqlite3.connect('reviews.sqlite')
>>> c = conn.cursor()
>>> c.execute("SELECT * FROM review_db WHERE date" \
...           " BETWEEN '2017-01-01 00:00:00' AND DATETIME('now'))")
>>> results = c.fetchall()
>>> conn.close()
>>> print(results)
[('I love this movie', 1, '2017-04-24 00:14:38'),
 ('I disliked this movie', 0, '2017-04-24 00:14:38')]

```

Alternativ können Sie auch das kostenlose Plugin *SQLite Manager* für den Webbrowser Firefox verwenden (verfügbar unter <https://addons.mozilla.org/de/firefox/addon/sqlite-manager/>), das eine schicke grafische Benutzeroberfläche zum Bearbeiten von SQL-Datenbanken besitzt, wie das folgende Bildschirmfoto zeigt.



## 9.3 Entwicklung einer Webanwendung mit Flask

Nachdem wir in den vorangegangenen Abschnitten den Code zur Klassifizierung der Filmbewertungen vorbereitet haben, befassen wir uns als Nächstes mit den Grundlagen von *Flask*, einem Framework zum Entwickeln von Webanwendungen. Seit der Veröffentlichung der ersten Flask-Version von Armin Ronacher im Jahr 2010 hat diese API im Laufe der Zeit sehr an Popularität gewonnen. Zu den bekanntesten auf Flask beruhenden Anwendungen gehören z.B. LinkedIn und Pinterest. Da Flask in Python geschrieben wurde, gibt dieses Framework uns Python-Programmierern eine komfortable Schnittstelle an die Hand, um bereits vorhandenen Python-Code wie unseren Filmbewertungsklassifizierer einzubinden.

### Tipp

Flask wird auch als *Microframework* bezeichnet. Damit ist gemeint, dass der Kern des Frameworks schlank und einfach gehalten wird, sich jedoch leicht mit anderen Bibliotheken erweitern lässt. Der Einstieg in die leichtgewichtige Flask-API ist zwar nicht annähernd so schwierig wie dies bei manch anderem Python-Framework, z.B. Django, der Fall ist, ich empfehle Ihnen aber dennoch einen Blick in die offizielle Dokumentation, die Sie unter <http://flask.pocoo.org/docs/0.12/> finden, um die Funktionalität besser kennenzulernen.

Falls die Flask-Bibliothek in Ihrer Python-Umgebung noch nicht installiert ist, können Sie dies im Terminal mit `conda` oder `pip` leicht nachholen (die derzeit, Stand Oktober 2017, aktuelle stabile Flask-Version ist 0.12.1):

```
conda install flask
# oder pip install flask
```

### 9.3.1 Die erste Webanwendung mit Flask

In diesem Abschnitt werden wir eine sehr einfache Webanwendung entwickeln, um uns mit der Flask-API vertraut zu machen, bevor wir den Filmbewertungsklassifizierer implementieren. Zunächst erstellen wir den Verzeichnisbaum:

```
1st_flask_app_1/
    app.py
    templates/
        first_app.html
```

Die Datei `app.py` soll den eigentlichen Code enthalten, der vom Python-Interpreter ausgeführt wird, um die Flask-Webanwendung zu starten. Im Verzeichnis `templates` sucht Flask nach statischen HTML-Dateien, die im Webbroweser dargestellt werden sollen. Sehen wir uns den Inhalt der Datei `app.py` an:

```
from flask import Flask, render_template

app = Flask(__name__)
@app.route('/')
def index():
    return render_template('first_app.html')

if __name__ == '__main__':
    app.run()
```

Sehen Sie sich den Code genau an. Er besteht aus folgenden Schritten:

1. Wir starten unsere Anwendung als ein einzelnes Modul, daher initialisieren wir eine neue Flask-Instanz mit dem Argument `__name__`, um Flask mitzuteilen, dass sich der Ordner mit den HTML-Vorlagen (`templates`) im selben Verzeichnis befindet.
2. Als Nächstes setzen wir den Dekorierer `@app.route('/')` ein, um die URL anzugeben, deren Aufruf die Ausführung der `index`-Funktion auslösen soll.
3. Hier gibt die `index`-Funktion einfach nur die Datei `first_app.html` aus, die sich im `template`-Ordner befindet.
4. Zum Abschluss verwenden wir die `run`-Funktion, um die Anwendung nur dann auf dem Server zu starten, wenn dieses Skript direkt vom Python-Interpreter ausgeführt wird, was durch die Anweisung `if __name__ == '__main__'` gewährleistet ist.

Nun kommen wir zum Inhalt der Datei `first_app.html`.

```
<!doctype html>
<html>
  <head>
    <title>First app</title>
  </head>
  <body>
    <div>Die erste Webanwendung mit Flask!</div>
  </body>
</html>
```

### Tipp

Sollte Ihnen die HTML-Syntax noch nicht geläufig sein, empfehle ich den Besuch der Webseite <https://developer.mozilla.org/de/docs/Web/HTML>, auf der Sie hilfreiche Tutorials zu den Grundlagen von HTML finden.

Hier haben wir einfach eine leere HTML-Vorlage um ein `div`-Element (ein Element zur Textstrukturierung) erweitert, das den Text `Die erste Webanwendung mit Flask!` enthält.

Praktischerweise erlaubt Flask die lokale Ausführung der Webanwendungen, was deren Entwicklung sowie das Testen vor der Bereitstellung auf einem öffentlich zugänglichen Webserver erleichtert. Nun können wir die Webanwendung starten, indem wir im Arbeitsverzeichnis `1st_flask_app_1` den folgenden Befehl ausführen:

```
python3 app.py
```

Das Terminal sollte jetzt eine Meldung wie die folgende ausgeben:

```
* Running on http://127.0.0.1:5000/
```

Geben Sie die in dieser Zeile angezeigte Adresse des lokalen Servers im Browser an, um die Webanwendung in Aktion zu sehen. Wenn alles korrekt funktioniert, sollte nun eine schlichte Webseite mit dem Text `Die erste Webanwendung mit Flask!` angezeigt werden.



### 9.3.2 Formularvalidierung und -ausgabe

Im Folgenden werden wir unsere einfache Flask-Webanwendung um HTML-Formularelemente erweitern, damit wir vom User anzugebende Daten entgegennehmen können. Zu diesem Zweck installieren wir via `conda` oder `pip` die `WTForms`-Bibliothek (<https://wtforms.readthedocs.org/en/latest/>):

```
conda install wtforms  
# oder pip install wtforms
```

Die Webanwendung soll den User wie in der Abbildung dargestellt auffordern, einen Namen in ein Textfeld einzugeben.



Sobald die Schaltfläche zum Absenden (SAG HALLO) angeklickt wurde und die Formularvalidierung erfolgt ist, wird eine neue HTML-Seite ausgegeben, die den Username anzeigt:



### Einrichten der Verzeichnisstruktur

Die neue Verzeichnisstruktur, die wir für diese Anwendung einrichten müssen, sieht so aus:

```
1st_flask_app_2/  
    app.py  
    static/  
        style.css
```

```
templates/
    _formhelpers.html
    first_app.html
    hello.html
```

Die Datei `app.py` besitzt nun folgenden Inhalt:

```
from flask import Flask, render_template, request
from wtforms import Form, TextAreaField, validators

app = Flask(__name__)

class HelloForm(Form):
    sayhello = TextAreaField('', [validators.DataRequired()])

@app.route('/')
def index():
    form = HelloForm(request.form)
    return render_template('first_app.html', form=form)

@app.route('/hello', methods=['POST'])
def hello():
    form = HelloForm(request.form)
    if request.method == 'POST' and form.validate():
        name = request.form['sayhello']
        return render_template('hello.html', name=name)
    return render_template('first_app.html', form=form)

if __name__ == '__main__':
    app.run(debug=True)
```

Wir gehen den Code wieder schrittweise durch:

1. Wir erweitern die `index`-Funktion mit `wtforms` um ein Textfeld, das mithilfe der `TextAreaField`-Klasse auf der Startseite eingebettet wird. Diese Klasse prüft automatisch, ob der User einen gültigen Inhalt eingegeben hat.
2. Außerdem definieren wir eine neue Funktion namens `hello`, die eine HTML-Seite namens `hello.html` ausgibt, nachdem die Formularvalidierung erfolgt ist.
3. Wir verwenden hier die POST-Methode, um dem Server die Daten im Body der Nachricht zu übermitteln. Und schließlich aktivieren wir den Flask-Debugger, indem wir in der `app.run`-Methode das Argument `Debug` auf `True` setzen – bei der Entwicklung neuer Webanwendungen ist das ein nützliches Feature.

## Implementierung eines Makros mit Jinja2

Nun werden wir in der Datei `_formhelpers.html` mithilfe der *Jinja2-Templating-Engine*, die wir später in die Datei `firstapp.html` importieren, ein allgemeines Makro zur Ausgabe des Textfeldes implementieren:

```
{% macro render_field(field) %}
<dt>{{ field.label }}</dt>
<dd>{{ field(**kwargs)|safe }}</dd>
{% if field.errors %}
<ul class=errors>
{% for error in field.errors %}
<li>{{ error }}</li>
{% endfor %}
</ul>
{% endif %}
</dd>
</dt>
{% endmacro %}
```

Eine ausführliche Erläuterung der *Jinja2-Templating-Sprache* geht über den Rahmen dieses Buches hinaus, unter <http://jinja.pocoo.org> finden Sie jedoch eine umfassende Dokumentation der *Jinja2-Syntax*.

## Hinzufügen eines Stils per CSS

Als Nächstes erstellen wir eine einfache *CSS-Datei (Cascading Style Sheets)* namens `style.css`, um zu demonstrieren, wie sich das »Look & Feel« von HTML modifizieren lässt. Zu diesem Zweck muss die nachstehende CSS-Datei, die einfach nur die Schriftgröße der HTML-Elemente im Body verdoppelt, im Unterverzeichnis `static` (in dem Flask standardmäßig nach statischen Dateien sucht) gespeichert werden.

```
body {
    font-size: 2em;
}
```

Nachstehend der Inhalt der modifizierten Datei `first_app.html`, die nun ein Eingabeformular ausgibt, in das der User einen Namen eintragen kann:

```
<!doctype html>
<html>
<head>
<title>First app</title>
```

```
<link rel="stylesheet" href="{{ url_for('static',  
    filename='style.css') }}>  
</head>  
<body>  
    {% from "_formhelpers.html" import render_field %}  
    <div>Wie heißen Sie?</div>  
    <form method=post action="/hello">  
        <dl>  
            {{ render_field(form.sayhello) }}  
        </dl>  
        <input type=submit value='Sag Hallo' name='submit_btn'>  
    </form>  
</body>  
</html>
```

Im `<head>`-Abschnitt der Datei `first_app.html` laden wir die CSS-Datei. Sie sollte nun die Größe aller Textelemente im HTML-Body ändern. Im `<body>`-Abschnitt importieren wir das Makro aus der Datei `_formhelpers.html` und geben das Formular `sayhello` aus, das in der Datei `app.py` genannt wurde. Dann fügen wir dem Formularelement noch eine Schaltfläche hinzu, damit der User den eingegebenen Text übermitteln kann.

## Erstellen der Ergebnisseite

Abschließend erstellen wir die Datei `hello.html`. Diese wird in der `hello`-Funktion durch die Codezeile

```
return render_template('hello.html', name=name)
```

zurückgegeben, die wir im `app.py`-Skript zur Anzeige des vom User eingegebenen Textes definiert haben. Hier der Code:

```
<!doctype html>  
<html>  
    <head>  
        <title>First app</title>  
        <link rel="stylesheet" href="{{ url_for('static',  
            filename='style.css') }}>  
    </head>  
    <body>  
        <div>Hello {{ name }}</div>  
    </body>  
</html>
```

Nun ist die modifizierte Flask-Webanwendung bereit und wir können sie lokal ausführen, indem wir im Hauptverzeichnis der Anwendung den folgenden Befehl eingeben:

```
python3 app.py
```

Das Ergebnis kann durch die Eingabe von `http://127.0.0.1:5000/` in die Adresszeile des Browsers angezeigt werden.

### Tipp

Falls die Webentwicklung für Sie noch Neuland ist, dürfte Ihnen das Ganze auf den ersten Blick ziemlich kompliziert vorkommen. In diesem Fall empfehle ich Ihnen, einfach die obige Verzeichnisstruktur auf Ihrem Laufwerk einzurichten und sie genau zu erkunden. Sie werden feststellen, dass das Flask-Framework tatsächlich ziemlich unkompliziert und viel einfacher ist, als man zunächst denkt. Weitere Informationen und Beispiele finden Sie in der ausgezeichneten Flask-Dokumentation unter <http://flask.pocoo.org/docs/0.12/>.

## 9.4 Der Filmbewertungsklassifizierer als Webanwendung

Damit sind Sie nun mit den Grundlagen der Webentwicklung mit Flask vertraut und wir können damit fortfahren, den Filmbewertungsklassifizierer als Webanwendung zu implementieren. In diesem Abschnitt entwickeln wir eine Webanwendung, die den User wie in der folgenden Abbildung auffordert, eine Filmbewertung einzugeben:

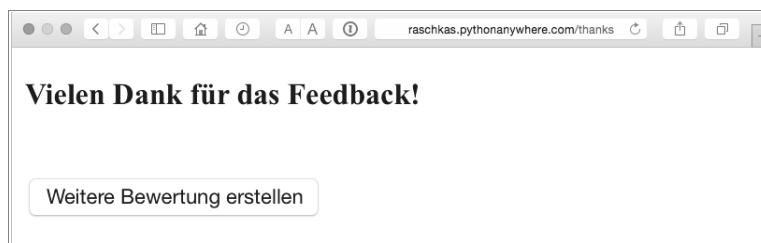


Nach dem Absenden der Bewertung wird dem User eine neue Seite mit der vorhergesagten Klassenbezeichnung und deren Wahrscheinlichkeit angezeigt.

Außerdem kann der User Feedback geben, indem er auf eine der Schaltflächen RICHTIG oder FALSCH klickt (siehe Abbildung).



Wenn der User auf eine der Schaltflächen RICHTIG oder FALSCH klickt, wird unser Klassifizierungsmodell unter Berücksichtigung des User-Feedbacks aktualisiert. Darüber hinaus speichern wir die vom User eingegebene Filmbewertung sowie die tatsächliche Klassenbezeichnung, die sich aus der angeklickten Schaltfläche ableiten lässt, zwecks künftiger Wiederverwendung in einer SQLite-Datenbank. Die dritte Seite, die dem User nach dem Anklicken einer der Feedback-Schaltflächen angezeigt wird, ist ein einfacher *Dankeschön*-Bildschirm mit einer Schaltfläche WEITERE BEWERTUNG ERSTELLEN, die wieder zur Startseite führt (siehe Abbildung).

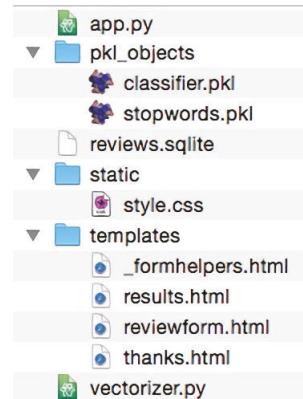


### Tipp

Bevor wir uns mit dem eigentlichen Code der Webanwendung befassen, sollten Sie einen Blick auf die Demoversion werfen, die ich auf meiner Website zur Verfügung gestellt habe, um besser zu veranschaulichen, was wir in diesem Abschnitt erreichen wollen (<http://raschka.pythonanywhere.com>).

### 9.4.1 Dateien und Ordner – die Verzeichnisstruktur

Werfen Sie zunächst einen Blick auf den Verzeichnisbaum, der für die Filmbewertungsanwendung erstellt werden muss:



In den vorangegangenen Abschnitten dieses Kapitels haben wir die Datei `vectorizer.py`, die SQLite-Datenbank `reviews.sqlite` und das `pkl_objects`-Unterverzeichnis für die serialisierten Objekte bereits erstellt.

Die im Hauptverzeichnis befindliche Datei `app.py` ist ein Python-Skript, das den Flask-Code enthält. Und in der zuvor angelegten Datenbankdatei `reviews.sqlite` speichern wir die der Webanwendung übermittelten Filmbewertungen. Das Unterverzeichnis `templates` enthält HTML-Vorlagen, die von Flask ausgegeben und im Browser dargestellt werden. Und das Unterverzeichnis `static` enthält eine einfache CSS-Datei, die das Aussehen des angezeigten HTML-Codes modifiziert.

#### Tipp

Die das Buch ergänzenden Codebeispiele enthalten ein weiteres Verzeichnis mit der Klassifiziereranwendung, die den hier erörterten Code verwendet. Sie können die Dateien bei GitHub herunterladen: <https://github.com/rasbt/python-machine-learning-book-2nd-edition/>. Den Code aus diesem Abschnitt finden Sie im Unterverzeichnis `.../code/ch09/movieclassifier`.

### 9.4.2 Implementierung der Hauptanwendung app.py

Da die Datei `app.py` ziemlich lang ist, erstellen wir sie in zwei Schritten. Das erste Fragment importiert die erforderlichen Python-Module und -Objekte und enthält den Code zum Deserialisieren und Einrichten des Klassifizierungsmodells:

```
from flask import Flask, render_template, request
from wtforms import Form, TextAreaField, validators
import pickle
import sqlite3
import os
import numpy as np
# HashingVectorizer aus lokalem Verzeichnis importieren
from vectorizer import vect

app = Flask(__name__)

##### Klassifizierer einrichten
cur_dir = os.path.dirname(__file__)
clf = pickle.load(open(os.path.join(cur_dir,
                                    'pkl_objects/classifier.pkl'), 'rb'))
db = os.path.join(cur_dir, 'reviews.sqlite')

def classify(document):
    label = {0: 'negativ', 1: 'positiv'}
    X = vect.transform([document])
    y = clf.predict(X)[0]
    proba = clf.predict_proba(X).max()
    return label[y], proba

def train(document, y):
    X = vect.transform([document])
    clf.partial_fit(X, [y])

def sqlite_entry(path, document, y):
    conn = sqlite3.connect(path)
    c = conn.cursor()
    c.execute("INSERT INTO review_db \
              (review, sentiment, date) " \
              "VALUES (?, ?, DATETIME('now'))", (document, y))
    conn.commit()
    conn.close()
```

Dieser erste Teil des `app.py`-Skripts sollte Ihnen inzwischen bekannt vorkommen: Wir importieren den `HashingVectorizer` und deserialisieren den logistischen Regressionsklassifizierer. Dann definieren wir die Funktion `classify`, die sowohl die vorhergesagte Klassenbezeichnung eines Textdokuments als auch die dazugehörige Wahrscheinlichkeit zurückgibt. Die Funktion `train` kann eine Aktualisierung des Klassifizierers vornehmen, wenn ihr ein Text und eine Klassenbezeichnung übergeben werden.

Mithilfe der Funktion `sqlite_entry` speichern wir die übermittelten Bewertungen zusammen mit Klassenbezeichnung und Zeitstempel in der SQLite-Datenbank. Beachten Sie, dass sich das `c1f`-Objekt nach einem Neustart der Webanwendung wieder im ursprünglichen serialisierten Zustand befindet. Am Ende dieses Kapitels werden Sie erfahren, wie man die in der SQLite-Datenbank gesammelten Daten für die dauerhafte Anpassung des Klassifizierers verwenden kann.

Die Vorgehensweise im zweiten Teil des `app.py`-Skripts erscheint ebenfalls vertraut:

```
##### Flask
class ReviewForm(Form):
    moviereview = TextAreaField('', [
        validators.DataRequired(),
        validators.length(min=15)])

@app.route('/')
def index():
    form = ReviewForm(request.form)
    return render_template('reviewform.html', form=form)

@app.route('/results', methods=['POST'])
def results():
    form = ReviewForm(request.form)
    if request.method == 'POST' and form.validate():
        review = request.form['moviereview']
        y, proba = classify(review)
        return render_template('results.html',
                               content=review,
                               prediction=y,
                               probability=round(proba*100, 2))
    return render_template('reviewform.html', form=form)

@app.route('/thanks', methods=['POST'])
def feedback():
    feedback = request.form['feedback_button']
    review = request.form['review']
    prediction = request.form['prediction']

    inv_label = {'negativ': 0, 'positiv': 1}
    y = inv_label[prediction]
    if feedback == 'Falsch':
        y = int(not(y))
    train(review, y)
```

```

    sqlite_entry(db, review, y)
    return render_template('thanks.html')

if __name__ == '__main__':
    app.run(debug=True)

```

Wir definieren die Klasse `ReviewForm`, die ein `TextAreaField` instanziert, das in der Vorlagendatei `reviewform.html` (die Startseite der Webanwendung) ausgegeben wird, die wiederum von der `index`-Funktion dargestellt wird. Mit dem Parameter `validators.length(min=15)` legen wir fest, dass der User eine Bewertung mit einer Länge von mindestens 15 Zeichen eingeben muss. In der `results`-Funktion rufen wir den Inhalt des übermittelten Formulars ab und übergeben ihn dem Klassifizierer, damit er eine Vorhersage trifft, die wir dann mit der Vorlage `results.html` anzeigen.

Die `feedback`-Funktion sieht auf den ersten Blick ein wenig kompliziert aus. Sie erledigt im Wesentlichen Folgendes: Wenn der User eine der Schaltflächen **RICHTIG** oder **FALSCH** anklickt, entnimmt sie der `results.html`-Vorlage die Vorhersage und wandelt sie in eine ganzzahlige Klassenbezeichnung um, mit der die im ersten Teil des `app.py`-Skripts implementierte `train`-Funktion den Klassifizierer aktualisiert. Außerdem wird der SQLite-Datenbank mithilfe der `sqlite_entry`-Funktion ein neuer Eintrag hinzugefügt, sofern der User Feedback gegeben hat. Schließlich wird die Vorlage `thanks.html` ausgegeben, um dem User für das Feedback zu danken.

### 9.4.3 Einrichtung des Bewertungsformulars

Sehen wir uns nun die Vorlage `reviewform.html` an, die als Startseite der Webanwendung dient:

```

<!doctype html>
<html>
  <head>
    <title>Filmbewertung</title>
  </head>
  <body>
    <h2>Bitte die Filmbewertung eingeben:</h2>
    {% from "_formhelpers.html" import render_field %}
    <form method=post action="/results">
      <dl>
        {{ render_field(form.movieReview, cols='30', rows='10') }}
      </dl>
      <div>
        <input type=submit value='Bewertung absenden'
               name='submit_btn'>
      </div>
    </form>
  </body>
</html>

```

```
</div>
</form>
</body>
</html>
```

Hier importieren wir die Vorlage `_formhelpers.html`, die wir im Abschnitt *Formularvalidierung und -ausgabe* definiert haben. Die `render_field`-Funktion dieses Makros dient zur Ausgabe eines `TextAreaFields`, in das der User seine Bewertung eingeben und dann mit der darunter angezeigten Schaltfläche BEWERTUNG ABSENDEN übermitteln kann. Dieses Eingabefeld besitzt 30 Spalten und 10 Zeilen.

#### 9.4.4 Eine Vorlage für die Ergebnisseite erstellen

Die nächste Vorlage `results.html` sieht schon um einiges interessanter aus:

```
<!doctype html>
<html>
  <head>
    <title>Filmbewertungsklassifizierung</title>
    <link rel="stylesheet" href="{{ url_for('static',
      filename='style.css') }}">
  </head>
  <body>
    <h3>Ihre Filmbewertung:</h3>
    <div>{{ content }}</div>

    <h3>Vorhersage:</h3>
    <div>Dieser Film ist <strong>{{ prediction }}</strong>
    (Wahrscheinlichkeit: {{ probability }}%).</div>

    <div id='button'>
      <form action="/thanks" method="post">
        <input type=submit value='Richtig'
        name='feedback_button'>
        <input type=submit value='Falsch'
        name='feedback_button'>
        <input type=hidden value='{{ prediction }}'
        name='prediction'>
        <input type=hidden value='{{ content }}'
        name='review'>
      </form>
    </div>

    <div id='button'>
      <form action="/">
```

```

<input type=submit
       value='Weitere Bewertung erstellen'>
</form>
</div>
</body>
</html>

```

Zunächst einmal fügen wir mit den Feldern {{ content }}, {{ prediction }} und {{ probability }} die übermittelte Bewertung und das Ergebnis der Vorhersage ein. Vielleicht ist Ihnen schon aufgefallen, dass wir die Platzhalter {{ content }} und {{ prediction }} in dem Formular, das die Schaltflächen RICHTIG und FALSCH enthält, ein weiteres Mal benutzen. Hierbei handelt es sich um einen kleinen Trick, um diese Werte mithilfe der POST-Methode dem Server zu übermitteln, der den Klassifizierer aktualisiert und die Bewertung speichert, wenn der User auf eine der beiden Schaltflächen klickt.

Außerdem importieren wir am Anfang der `results.html`-Datei eine CSS-Datei (`style.css`). Die Einstellungen in dieser Datei sind ziemlich einfach: Die Breite des Inhalts der Webanwendung wird auf 600 Pixel begrenzt und die mit der Div-ID `button` gekennzeichneten RICHTIG- und FALSCH-Schaltflächen werden um 20 Pixel nach unten verschoben:

```

body{
    width: 600px;
}
.button{
    padding-top: 20px;
}

```

Diese CSS-Datei ist lediglich ein Platzhalter – es steht Ihnen natürlich frei, das Aussehen der Webanwendung ganz nach Ihrem Geschmack zu gestalten.

Die letzte HTML-Datei, die wir für die Webanwendung noch implementieren müssen, ist die Vorlage `thanks.html`. Wie der Name schon ahnen lässt, stellt sie eine Dankeschön-Nachricht dar, wenn ein User mit einer der Schaltflächen RICHTIG oder FALSCH Feedback gegeben hat. Außerdem gibt es am unteren Ende dieser Seite eine Schaltfläche WEITERE BEWERTUNG ERSTELLEN, die zurück zur Startseite führt. Hier der Inhalt der Datei:

```

<!doctype html>
<html>
<head>
    <title>Filmbewertung</title>
</head>
<body>

```

```
<h3>Vielen Dank für Ihr Feedback!</h3>
<div id='button'>
    <form action="/">
        <input type=submit
            value='Weitere Bewertung erstellen'>
    </form>
</div>
</body>
</html>
```

Bevor wir fortfahren und die Webanwendung auf einem öffentlich zugänglichen Webserver zur Verfügung stellen, sollten wir sie testweise lokal starten. Geben Sie im Terminal folgenden Befehl ein:

```
python3 app.py
```

Vergessen Sie nach dem Testen der Webanwendung nicht, am Ende des `app.py`-Skripts beim Befehl `app.run()` das Argument `debug=True` zu entfernen.

## 9.5 Einrichtung der Webanwendung auf einem öffentlich zugänglichen Webserver

Nach dem lokalen Test der Webanwendung können wir sie auf einem öffentlich zugänglichen Webserver einrichten. Für dieses Beispiel werden wir den Webdienst *PythonAnywhere* nutzen, der auf die mühelose Einrichtung und das Hosten von Python-Webanwendungen spezialisiert ist. PythonAnywhere bietet unter anderem auch ein einfaches Userkonto für Einsteiger an, das den kostenlosen Betrieb einer einzelnen Webanwendung gestattet.

### 9.5.1 Erstellen eines Benutzerkontos bei PythonAnywhere

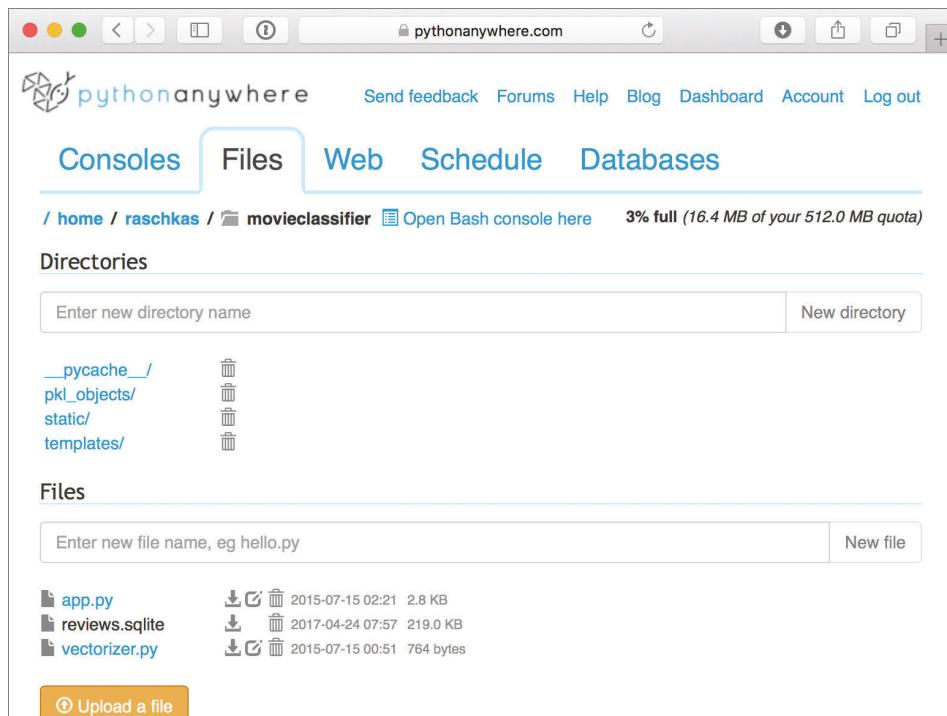
Besuchen Sie die Website <https://www.pythonanywhere.com>, um ein Benutzerkonto anzulegen. Folgen Sie dem Link PRICING & SIGNUP in der oberen rechten Ecke und klicken Sie dann auf CREATE A BEGINNER ACCOUNT. Hier müssen Sie einen Benutzernamen, ein Kennwort und eine gültige E-Mail-Adresse angeben. Klicken Sie nach dem Lesen und Akzeptieren der Nutzungsbedingungen auf REGISTER – nun verfügen Sie über ein Benutzerkonto.

Leider ist es mit dem kostenlosen Einsteigerkonto nicht möglich, vom Terminal aus per *SSH (Secure Shell)* auf den Server zuzugreifen, daher müssen wir die Bedienoberfläche der Webseite zur Verwaltung der Webanwendung nutzen. Um sie hochladen zu können, müssen wir zunächst eine neue Webanwendung für das

PythonAnywhere-Benutzerkonto anlegen. Nachdem Sie oben rechts auf die Schaltfläche DASHBOARD geklickt haben, steht ein Bedienfeld mit verschiedenen Registerkarten zur Verfügung. Öffnen Sie die nun angezeigte Registerkarte WEB und klicken Sie auf der linken Seite auf die Schaltfläche ADD A NEW WEB APP. Damit wird eine neue Python-3.5-Flask-Webanwendung erstellt. Geben Sie ihr den Namen movieclassifier.

### 9.5.2 Hochladen der Filmbewertungsanwendung

Wählen Sie anschließend die Registerkarte FILES aus, um das lokale Verzeichnis movieclassifier mithilfe der PythonAnywhere-Webschnittstelle hochzuladen. Wenn das erledigt ist, sollte im PythonAnywhere-Userkonto ein movieclassifier-Verzeichnis vorhanden sein, das dieselben Unterverzeichnisse und Dateien enthält wie der lokale movieclassifier-Ordner (siehe Abbildung).



Begeben Sie sich anschließend wieder zur Registerkarte WEB und klicken Sie auf die Schaltfläche RELOAD <USERNAME>.PYTHONANYWHERE.COM, um die vorgenommenen Änderungen zu übernehmen und die Webanwendung zu aktualisieren. Jetzt sollte die Webanwendung laufen und unter der Adresse <http://<Username>.pythonanywhere.com> öffentlich zugänglich sein.

**Tipp**

Leider sind Webserver ziemlich empfindlich, wenn eine Webanwendung auch nur den kleinsten Fehler aufweist. Für den Fall, dass die Ausführung der Webanwendung auf PythonAnywhere Schwierigkeiten bereitet und Ihr Webbrowsert Fehlermeldungen anzeigt, sollten Sie die über die Registerkarte WEB zugänglichen Logdateien des Servers überprüfen, um dem Problem auf den Grund zu gehen.

### 9.5.3 Update des Filmbewertungsklassifizierers

Das Vorhersagemodell wird zwar in Echtzeit aktualisiert, sobald ein User Feedback zur Klassifizierung gibt, bei einem Neustart oder einem Absturz des Servers wird das `c1f`-Objekt jedoch zurückgesetzt. Wenn wir die Webanwendung erneut starten, wird das `c1f`-Objekt in der Datei `classifier.pkl` reinitialisiert. Um Aktualisierungen dauerhaft zu übernehmen, könnten wir das `c1f`-Objekt nach jedem Update erneut serialisieren, allerdings wäre diese Vorgehensweise bei wachsender Userzahl sehr ineffizient und könnte womöglich zu einer defekten serialisierten Datei führen, wenn mehrere User gleichzeitig Feedback geben.

Eine bessere Lösung wäre es, das Vorhersagemodell anhand der in einer SQLite-Datenbank gespeicherten Feedbackdaten zu aktualisieren. Zu diesem Zweck müssen wir die SQLite-Datenbank vom PythonAnywhere-Server herunterladen, das `c1f`-Objekt auf dem lokalen Computer aktualisieren und die neue serialisierte Datei wieder bei PythonAnywhere hochladen. Zur lokalen Aktualisierung des Klassifizierers erstellen wir im `movieclassifier`-Verzeichnis eine Skriptdatei namens `update.py` mit folgendem Inhalt:

```
import pickle
import sqlite3
import numpy as np
import os

# HashingVectorizer aus lokalem Verzeichnis importieren
from vectorizer import vect

def update_model(db_path, model, batch_size=10000):
    conn = sqlite3.connect(db_path)
    c = conn.cursor()
    c.execute('SELECT * from review_db')

    results = c.fetchmany(batch_size)
    while results:
```

```

data = np.array(results)
X = data[:, 0]
y = data[:, 1].astype(int)

classes = np.array([0, 1])
X_train = vect.transform(X)
model.partial_fit(X_train, y, classes=classes)
results = c.fetchmany(batch_size)
conn.close()
return model

cur_dir = os.path.dirname(__file__)
clf = pickle.load(open(os.path.join(cur_dir,
                                    'pkl_objects',
                                    'classifier.pkl'), 'rb'))
db = os.path.join(cur_dir, 'reviews.sqlite')

clf = update_model(db_path=db, model=clf, batch_size=10000)

# Entfernen Sie die Kommentarzeichen vor den folgenden
# Zeilen, wenn Sie sicher sind, dass die Datei classifier.pkl
# dauerhaft aktualisiert werden soll.

# pickle.dump(clf, open(os.path.join(cur_dir,
#                                     'pkl_objects', 'classifier.pkl'), 'wb')
#             , protocol=4)

```

Die Funktion `update_model` ruft so lange jeweils 10.000 Datensätze der SQLite-Datenbank ab, bis weniger Einträge vorhanden sind. Wir hätten statt `fetchmany` auch `fetchone` verwenden können, um die Datensätze einzeln abzurufen, das wäre allerdings hinsichtlich des Rechenaufwands äußerst ineffizient. Die ebenfalls verfügbare `fetchall`-Methode zum Abruf aller Datensätze könnte Probleme bereiten, wenn wir es mit sehr großen Datenmengen zu tun haben, die zu groß sind, um sie in den Arbeitsspeicher des Computers oder des Servers zu laden.

Nachdem das `update.py`-Skript erstellt ist, können wir es ebenfalls in das `movie-classifier`-Verzeichnis bei PythonAnywhere hochladen und die `update_model`-Funktion im `app.py`-Skript importieren, um den Klassifizierer bei jedem Neustart der Webanwendung anhand der SQLite-Datenbank zu aktualisieren. Zu diesem Zweck müssen wir lediglich am Anfang der `app.py`-Datei eine Codezeile zum Importieren der `update_model`-Funktion des `update.py`-Skripts hinzufügen:

```
# update-Funktion aus lokalem Verzeichnis importieren
from update import update_model
```

Die `update_model`-Funktion muss jetzt im `__main__`-Teil der Anwendung aufgerufen werden:

```
...  
if __name__ == '__main__':  
    clf=update_model(db_path=db, model=c1f, batch_size=10000)  
...
```

Die Änderungen an den Codeabschnitten sorgen dafür, dass die `pickle`-Datei bei PythonAnywhere aktualisiert wird. In der Praxis ist es jedoch nur selten erforderlich, die Webanwendung neu zu starten, und es wäre sinnvoll, das in der SQLite-Datenbank gespeicherte Feedback der Benutzer vor der Aktualisierung zu überprüfen, um sicherzustellen, dass die Daten für den Klassifizierer wertvolle Informationen enthalten.

## 9.6 Zusammenfassung

In diesem Kapitel haben Sie eine Vielzahl nützlicher und praktischer Themengebiete kennengelernt, die Ihre Kenntnisse der Theorie des Machine Learnings erweitern. Sie haben erfahren, wie ein Modell nach dem Trainieren serialisiert und zwecks späteren Gebrauchs wieder eingelesen werden kann. Außerdem haben Sie eine SQLite-Datenbank zum effizienten Speichern von Daten erzeugt sowie eine Webanwendung erstellt, die es ermöglicht, den Filmbewertungsklassifizierer öffentlich zugänglich zu machen.

Bislang haben wir im Zusammenhang mit dem Machine Learning, bewährten Verfahrensweisen und überwachten Lernmodellen zur Klassifizierung eine Vielzahl von Themen erörtert. Im nächsten Kapitel werden wir uns mit einer weiteren Unterkategorie des überwachten Lernens befassen: der Regressionsanalyse. Im Gegensatz zu den kategorialen Klassenbezeichnungen einer Klassifizierung, die wir bisher betrachtet haben, ermöglicht sie die Vorhersage stetiger Variablen.

# Vorhersage stetiger Zielvariablen durch Regressionsanalyse

In den vorangegangenen Kapiteln haben Sie die grundlegenden Konzepte des *überwachten Lernens* kennengelernt und eine Vielzahl verschiedener Modelle für Klassifizierungsaufgaben trainiert, um Gruppenzugehörigkeiten oder kategoriale Variablen vorherzusagen. In diesem Kapitel werden wir uns mit einer weiteren Unterkategorie des überwachten Lernens befassen: mit der Regressionsanalyse.

Regressionsmodelle dienen dazu, *stetige* Zielvariablen vorherzusagen und sind daher sowohl für viele wissenschaftliche Fragestellungen als auch für industrielle Anwendungen interessant, wie etwa das Verständnis des Zusammenhangs zwischen Variablen, das Erkennen von Trends oder das Treffen von Vorhersagen. Ein typisches Beispiel ist eine Prognose der Verkäufe eines Unternehmens in den kommenden Monaten.

In diesem Kapitel werden wir die grundlegenden Konzepte von Regressionsmodellen erörtern und dabei folgende Themen betrachten:

- Erkundung und Visualisierung von Datenmengen
- Verschiedene Ansätze zur Implementierung linearer Regressionsmodelle
- Trainieren von nicht für statistische Ausreißer anfälligen Regressionsmodellen
- Bewertung von Regressionsmodellen und Erkennung gängiger Probleme
- Anpassung von Regressionsmodellen an nichtlineare Daten

## 10.1 Lineare Regression

Eine lineare Regression soll den Zusammenhang zwischen einem Merkmal oder mehreren Merkmalen und einer stetigen Zielvariable modellieren. Wie Sie aus Kapitel 1 wissen, ist die Regressionsanalyse eine Unterkategorie des überwachten Machine Learnings. Im Gegensatz zur Klassifizierung, einer weiteren Unterkategorie des überwachten Machine Learnings, versucht die Regressionsanalyse jedoch, statt kategorialer Klassenbezeichnungen stetige Werte vorherzusagen.

In den folgenden Abschnitten werden die grundlegenden Typen linearer Regressions vorgestellt: die einfache lineare Regression und der multivariate Fall, die lineare Regression mit mehreren Merkmalen.

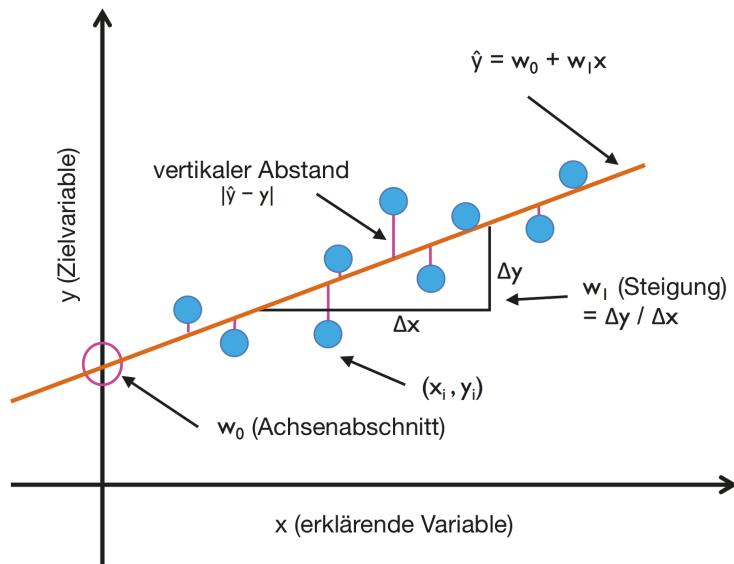
### 10.1.1 Ein einfaches lineares Regressionsmodell

Eine einfache (*univariate*) lineare Regression soll die Beziehung zwischen einem einzelnen Merkmal (der erklärenden Variablen  $x$  oder dem *Regressor*) und einer stetigen Zielvariable (der erklärten Variablen  $y$  oder dem *Regressanden*) modellieren. Die Gleichung eines linearen Modells mit einer erklärenden Variablen ist folgendermaßen definiert:

$$y = w_0 + w_1 x$$

Die Gewichtung  $w_0$  bezeichnet hier den y-Achsenabschnitt und  $w_1$  ist der Koeffizient der erklärenden Variablen. Unser Ziel ist es, die Gewichtungen der linearen Gleichung zu ermitteln, um die Beziehung zwischen der erklärenden und der Zielvariablen zu beschreiben, die dann wiederum dazu verwendet werden kann, den Wert der Zielvariablen für andere Werte der erklärenden Variablen vorherzusagen, die nicht Teil der Trainingsdatenmenge waren.

Gemäß der soeben definierten linearen Gleichung kann die lineare Regression als die Suche nach der Geraden aufgefasst werden, die am besten zu einer gegebenen Punktwolke passt (siehe Abbildung).



Diese Gerade wird *Regressionsgerade* genannt, und die vertikalen Abstände der einzelnen Punkte zur Regressionsgeraden heißen *Residuen* (oder unerklärte Anteile) – die Fehler der Vorhersage.

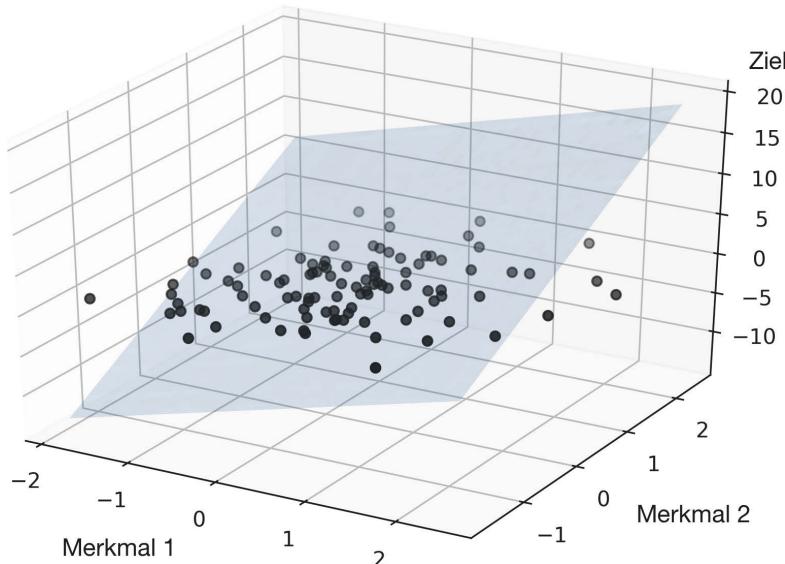
### 10.1.2 Multiple lineare Regression

Der spezielle Fall mit nur einer erklärenden Variablen wird als *einfache lineare Regression* bezeichnet. Es ist jedoch problemlos möglich, das lineare Regressionsmodell für mehrere erklärende Variablen zu verallgemeinern. Man spricht dann von *multipler (linearer) Regression*:

$$y = w_0 x_0 + w_1 x_1 + \dots + w_m x_m = \sum_{i=0}^m w_i x_i = w^T x$$

Hier ist  $w_0$  der y-Achsenabschnitt mit  $x_0 = 1$ .

Die folgende Abbildung zeigt, wie eine zweidimensionale angepasste Hyperebene einer multiplen linearen Regression mit zwei Merkmalen aussehen könnte:



Wie man sieht, ist die Visualisierung einer multiplen linearen Regression als dreidimensionales statisches Streudiagramm schon schwierig zu interpretieren. Da es keine Möglichkeit gibt, Hyperebenen mit zwei Dimensionen als Streudiagramm (an Datenmengen mit drei oder mehr Merkmalen angepasste multiple lineare Regressionsmodelle) darzustellen, konzentrieren sich die Beispiele und Visualisierungen dieses Kapitels vornehmlich auf den univariaten Fall und verwenden eine einfache lineare Regression. Allerdings beruhen uni- und multivariate lineare Regression auf denselben Konzepten und verwenden die gleichen Bewertungsverfahren. Der in diesem Kapitel implementierte Code ist also mit beiden Arten von Regressionsmodellen kompatibel.

## 10.2 Die Lebensbedingungen-Datensammlung

Bevor wir anfangen, das erste lineare Regressionsmodell zu implementieren, möchte ich noch eine weitere Datensammlung vorstellen: die *Lebensbedingungen-Datensammlung*. Sie enthält Informationen über die Lebensbedingungen in den Außenbezirken der Stadt Boston, die D. Harrison und D.L. Rubinfeld 1978 erhoben haben. Diese Datensammlung ist kürzlich aus dem UCI Machine Learning Repository entfernt worden, sie ist jedoch online verfügbar unter <https://raw.githubusercontent.com/rasbt/python-machine-learning-book-2nd-edition/master/code/ch10/housing.data.txt>. Wie bei jeder neuen Datenmenge ist es sinnvoll, sich durch eine einfache Visualisierung ein Bild von den Daten zu machen.

### 10.2.1 Einlesen der Datenmenge in einen DataFrame

In diesem Abschnitt werden wir die Lebensbedingungen-Datensammlung mit Pandas `read_csv`-Funktion einlesen. Sie ist schnell und vielseitig und empfiehlt sich als Tool zur Verarbeitung von in Tabellen gespeicherten einfachen Textdaten.

Die Merkmale der 506 Einträge können wie in dem Auszug aus der Beschreibung der Datensammlung folgendermaßen zusammengefasst werden:

- CRIM: Die Pro-Kopf-Kriminalitätsrate der jeweiligen Vorstadt
- ZN: Der Anteil der Landparzellen, die größer als 25.000 Quadratfuß sind (ca. 2.320 Quadratmeter, das entspricht einem Quadrat mit einer Seitenlänge von rund 48 Metern)
- INDUS: Der Flächenanteil der Gebiete, die nicht vom Einzelhandel genutzt werden
- CHAS: Die Charles-River-Dummy-Variable (der »Charles River« ist ein Fluss in Boston), die den Wert 1 besitzt, wenn der Fluss Teil des Bezirks ist, anderenfalls 0
- NOX: Die Stickstoffmonoxid-Konzentration (Teile pro 10 Millionen)
- RM: Die durchschnittliche Anzahl der Zimmer pro Wohnung/Gebäude
- AGE: Der Anteil der vor 1940 gebauten und von den Eigentümern selbst genutzten Wohneinheiten
- DIS: Die gewichteten Entfernungen zu fünf Arbeitsvermittlungsagenturen in Boston
- RAD: Ein Index, der die Zugänglichkeit von Einfallstraßen erfasst
- TAX: Die Höhe der unverminderten Grundsteuer pro 10.000 Dollar
- PTRATIO: Das Verhältnis von Schülern zu Lehrern in einem Bezirk
- B: Dieser Wert errechnet sich nach der Formel  $1000(Bk - 0.63)^2$ , wobei  $Bk$  den Anteil der Bevölkerung mit afroamerikanischer Abstammung angibt

- LSTAT: Der Prozentsatz der Bevölkerung mit niedrigem Sozialstatus
- MEDV: Der Medianwert des Preises der von Eigentümern selbst genutzten Wohnungen/Wohnhäuser in Einheiten von 1.000 Dollar

Im Folgenden soll der Preis der Wohnungen (MEDV) als Zielvariable dienen – also die Variable, die wir anhand einer oder mehrerer der anderen 13 erklärenden Variablen vorhersagen möchten. Zunächst einmal laden wir die Datensammlung herunter und speichern sie in einem pandas-DataFrame:

```
>>> import pandas as pd
>>> df = pd.read_csv('https://raw.githubusercontent.com/'
...                   'rasbt/python-machine-learning-book-'
...                   '2nd-edition/master/code/ch10//'
...                   'housing.data.txt',
...                   header=None, sep='\s+')
>>> df.columns = ['CRIM', 'ZN', 'INDUS', 'CHAS',
...                 'NOX', 'RM', 'AGE', 'DIS', 'RAD',
...                 'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV']
>>> df.head()
```

Um zu überprüfen, ob die Daten korrekt gespeichert wurden, geben wir die ersten fünf Zeilen der Datensammlung aus (siehe Abbildung).

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2

## 10.2.2 Visualisierung der wichtigen Eigenschaften einer Datenmenge

Die *explorative Datenanalyse (EDA, Exploratory Data Analysis)* ist ein wichtiger und empfehlenswerter erster Schritt zur Vorbereitung des Trainings eines Lernmodells. In diesem Abschnitt werden wir einige einfache, aber dennoch nützliche Verfahren der grafischen EDA-Toolbox einsetzen, die es ermöglichen, das Vorhandensein von sogenannten *Ausreißern (Outliers)*, die Datenverteilung und die Beziehungen zwischen den Merkmalen visuell zu erkennen.

Als Erstes erstellen wir eine *Streudiagrammmatrix* zur Visualisierung der paarweisen Korrelationen der Merkmale in dieser Datensammlung. Für die Ausgabe dieser Matrix verwenden wir die *pairplot*-Funktion der Seaborn-Bibliothek (<http://stanford.edu/~mwaskom/software/seaborn/>), einer auf Matplotlib basierenden Python-Bibliothek für die Darstellung statistischer Daten.

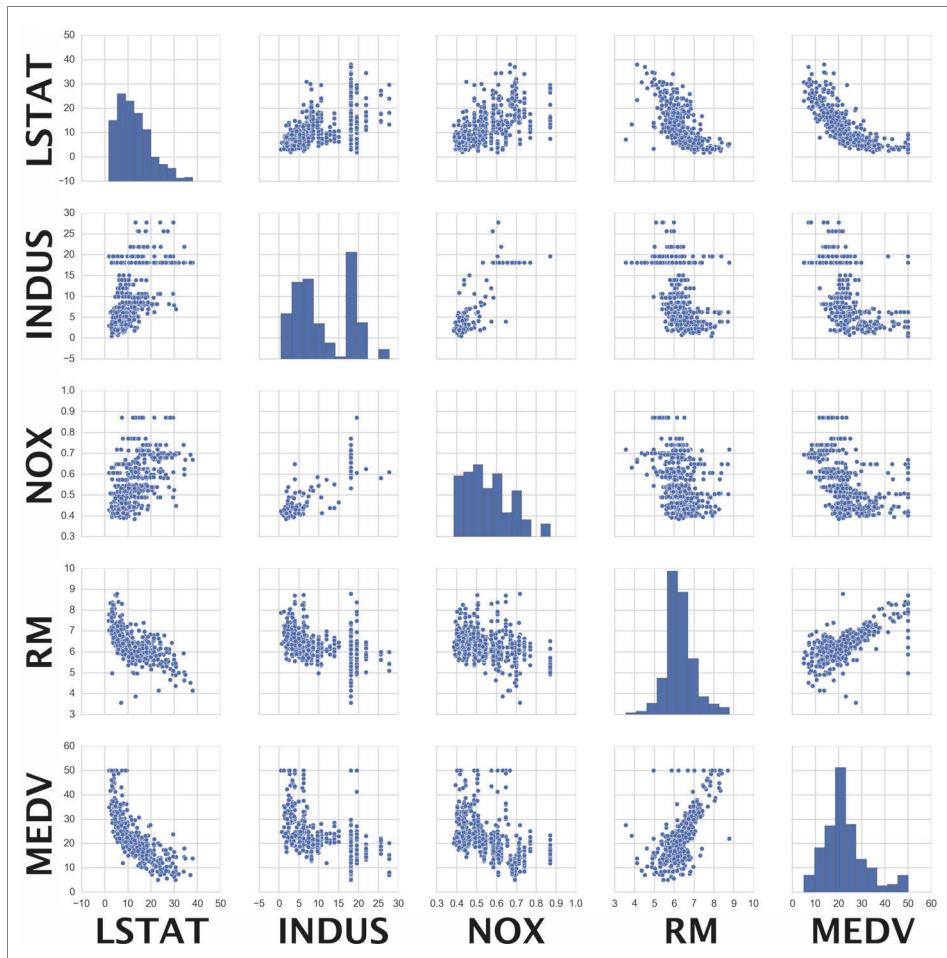
## Kapitel 10

### Vorhersage stetiger Zielvariablen durch Regressionsanalyse

Sie können das `seaborn`-Paket via `conda install seaborn` oder `pip install seaborn` installieren. Nach der Installation können Sie die Streudiagrammmatrix wie folgt erstellen:

```
>>> import matplotlib.pyplot as plt  
>>> import seaborn as sns  
>>> cols = ['LSTAT', 'INDUS', 'NOX', 'RM', 'MEDV']  
>>> sns.pairplot(df[cols], size=2.5)  
>>> plt.tight_layout()  
>>> plt.show()
```

Wie Sie der Abbildung entnehmen können, liefert die Streudiagrammmatrix eine Art grafische Zusammenfassung der verschiedenen Beziehungen in einer Datensetze.



Aus Platzgründen und der besseren Lesbarkeit halber haben wir nur fünf Spalten der Datensammlung ausgegeben: LSTAT, INDUS, NOX, RM und MEDV. Experimentieren Sie ruhig ein wenig herum und erstellen Sie eine Streudiagrammmatrix des gesamten `DataFrames`, um die Daten weiter zu erkunden. Wählen Sie z.B. andere Spaltennamen im `sns.pairplot`-Aufruf oder zeigen Sie alle Variablen der Streudiagrammmatrix an, indem Sie den Spaltenselektor weglassen (`sns.pairplot(df)`).

Anhand der Matrix können wir auf einen Blick erkennen, wie die Daten verteilt sind und ob es Ausreißer gibt. Beispielsweise ist in der fünften Spalte der vierten Zeile erkennbar, dass ein linearer Zusammenhang zwischen RM (Anzahl der Zimmer) und dem Wohnungspreis MEDV besteht. Außerdem deutet das Histogramm (in der Matrix ganz unten rechts) darauf hin, dass die MEDV-Werte in etwa normalverteilt sind, jedoch einige Ausreißer enthalten.

### Tipp

Anders als oftmals angenommen ist es zum Trainieren eines linearen Regressionsmodells nicht erforderlich, dass die erklärende Variable oder die Zielvariable normalverteilt sind. Die Annahme einer Normalverteilung ist lediglich bei bestimmten statistischen Tests und Hypothesentests notwendig, die über den Rahmen dieses Buches hinausgehen (Montgomery, D.C., Peck, E.A. und Vining, G.G. *Introduction to linear regression analysis*, John Wiley and Sons, 2012, Seiten 318-319).

### 10.2.3 Zusammenhänge anhand der Korrelationsmatrix erkennen

Im vorangegangenen Abschnitt haben wir die Datenverteilung der Variablen in der Lebensbedingungen-Datensammlung in Form von Streudiagrammen und Histogrammen visualisiert. Um den linearen Zusammenhang zwischen den Merkmalen quantifizieren und zusammenfassen zu können, erstellen wir nun eine Korrelationsmatrix. Diese ist eng mit der Kovarianzmatrix verwandt, der wir in Kapitel 5 im Abschnitt über die *Hauptkomponentenanalyse (PCA, Principal Component Analysis)* begegnet sind. Man kann sich die Korrelationsmatrix als eine neu skalierte Version der Kovarianzmatrix vorstellen – wenn man sie anhand standardisierter Daten berechnet, sind Korrelations- und Kovarianzmatrix tatsächlich sogar identisch.

Die Korrelationsmatrix ist eine quadratische Matrix der *Korrelationskoeffizienten* (manchmal auch *Produkt-Moment-* oder *Pearson-Korrelationskoeffizienten* genannt), die ein Maß für die linearen Abhängigkeiten zwischen den verschiedenen Merkmalspaaren sind. Der Korrelationskoeffizient  $r$  kann Werte zwischen -1 und +1 annehmen. Falls  $r = 1$  (bzw.  $r = -1$ ) ist, besteht ein vollständig positiver (bzw. negativer) linearer Zusammenhang zwischen zwei Merkmalen. Ist hingegen  $r = 0$ , besteht überhaupt kein Zusammenhang. Wie bereits erwähnt, kann der Korrela-

tionskoeffizient berechnet werden, indem man die Kovarianz zweier Merkmale  $x$  und  $y$  (Zähler) durch das Produkt ihrer Standardabweichungen (Nenner) teilt:

$$r = \frac{\sum_{i=1}^n (x^{(i)} - \mu_x)(y^{(i)} - \mu_y)}{\sqrt{\sum_{i=1}^n (x^{(i)} - \mu_x)^2} \sqrt{\sum_{i=1}^n (y^{(i)} - \mu_y)^2}} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

Hier bezeichnet  $\mu$  den Mittelwert der Stichprobe,  $\sigma_{xy}$  ist die Kovarianz der Merkmale  $x$  und  $y$  und  $\sigma_x$  sowie  $\sigma_y$  sind deren Standardabweichungen.

### Tipp

Man kann zeigen, dass die Kovarianz zweier standardisierter Merkmale dem linearen Korrelationskoeffizienten entspricht.

Zunächst standardisieren wir die Merkmale  $x$  und  $y$ , um die z-Werte zu erhalten, die wir als  $x'$  bzw.  $y'$  bezeichnen:

$$x' = \frac{x - \mu_x}{\sigma_x}, y' = \frac{y - \mu_y}{\sigma_y}$$

Die Kovarianz (der Grundgesamtheit) zweier Merkmale wird folgendermaßen berechnet:

$$\sigma_{xy} = \frac{1}{n} \sum_i^n (x^{(i)} - \mu_x)(y^{(i)} - \mu_y)$$

Da die Standardisierung eine Merkmalsvariable um den Mittelwert 0 zentriert, können wir nun die Kovarianz der skalierten Merkmale wie folgt berechnen:

$$\sigma'_{xy} = \frac{1}{n} \sum_i^n (x' - 0)(y' - 0)$$

Durch Wiedereinsetzen ergibt sich:

$$\begin{aligned} & \frac{1}{n} \sum_i^n \left( \frac{x - \mu_x}{\sigma_x} \right) \left( \frac{y - \mu_y}{\sigma_y} \right) \\ & \frac{1}{n \cdot \sigma_x \sigma_y} \sum_i^n (x^{(i)} - \mu_x)(y^{(i)} - \mu_y) \end{aligned}$$

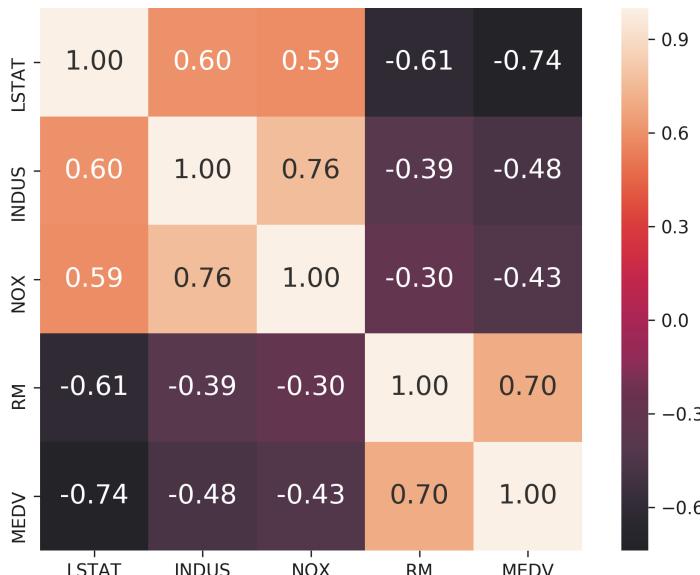
Das kann vereinfacht werden zu:

$$\sigma'_{xy} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

Im folgenden Codebeispiel wenden wir die `corrcoef`-Funktion von NumPy auf die fünf Merkmalsspalten an, die wir zuvor mithilfe der Streudiagrammmatrix visualisiert haben. Außerdem nutzen wir die `heatmap`-Funktion der `seaborn`-Bibliothek, um das Array der Korrelationsmatrix als Heatmap auszugeben:

```
>>> import numpy as np
>>> cm = np.corrcoef(df[cols].values.T)
>>> sns.set(font_scale=1.5)
>>> hm = sns.heatmap(cm,
...                   cbar=True,
...                   annot=True,
...                   square=True,
...                   fmt='.2f',
...                   annot_kws={'size': 15},
...                   yticklabels=cols,
...                   xticklabels=cols)
>>> plt.show()
```

Die Abbildung macht deutlich, dass die Korrelationsmatrix eine weitere nützliche zusammenfassende Grafik liefert, die dabei helfen kann, Merkmale anhand der jeweiligen linearen Abhängigkeiten auszuwählen:



Bei der Anpassung eines linearen Regressionsmodells sind wir an Merkmalen interessiert, die eine hohe Korrelation mit der Zielvariablen MEDV aufweisen. Beim Betrachten der Korrelationsmatrix wird deutlich, dass die Zielvariable MEDV am stärksten mit der Variablen LSTAT korreliert (-0,74). Allerdings zeigt die zuvor

erstellte Streudiagrammmatrix, dass eine deutliche nichtlineare Beziehung zwischen LSTAT und MEDV vorhanden ist. Andererseits ist die Korrelation zwischen RM und MEDV ebenfalls relativ hoch (0,70) – und in Anbetracht der linearen Beziehung zwischen den beiden Variablen, die wir im Streudiagramm feststellen konnten, scheint RM als erklärende Variable eine gute Wahl zu sein, um im nächsten Abschnitt die Konzepte eines einfachen linearen Regressionsmodells vorzustellen.

## 10.3 Implementierung eines linearen Regressionsmodells mit der Methode der kleinsten Quadrate

Am Anfang dieses Kapitels haben wir festgestellt, dass die lineare Regression als die Suche nach denjenigen Geraden aufgefasst werden kann, die am besten zu einer durch die Trainingsdaten gegebene Punktwolke passt. Wir haben jedoch weder den Begriff »am besten« definiert noch die verschiedenen Verfahren der Anpassung solch eines Modells erörtert. In den folgenden Abschnitten werden wir diese Lücken schließen und die *Methode der kleinsten Quadrate* verwenden, um die Parameter der Regressionsgeraden abzuschätzen, die die Summe der quadrierten vertikalen Abstände (die Residuen oder Fehler) zwischen Regressionsgeraden und Datenpunkten minimiert.

### 10.3.1 Berechnung der Regressionsparameter mit dem Gradientenabstiegsverfahren

Wie Sie von der Implementierung des *Adaline (ADaptive LInear NEuron)* in Kapitel 2 wissen, verwendet das künstliche Neuron eine lineare Aktivierungsfunktion, und wir haben eine Straffunktion  $J(\cdot)$  definiert, die minimiert wird, um durch Optimierungsalgorithmen, z.B. *Gradientenabstiegsverfahren (GD, Gradient Descent)* oder *stochastische Gradientenabstiegsverfahren (SGD, Stochastic Gradient Descent)*, die Gewichtungen zu ermitteln. Im Fall von Adaline ist diese Straffunktion die *Summe der quadrierten Abweichungen*. Sie ist identisch mit der Straffunktion, die wir für die Methode der kleinsten Quadrate wie folgt definieren:

$$J(w) = \frac{1}{2} \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2$$

Hier bezeichnet  $\hat{y}$  den vorhergesagten Wert  $\hat{y} = w^T x$ . (Beachten Sie, dass der Faktor  $\frac{1}{2}$  nur der bequemeren Herleitung der Aktualisierungsregel des Gradientenabstiegsverfahrens dient.) Die lineare Regression mit der Methode der kleinsten Quadrate kann als Adaline-Verfahren ohne Sprungfunktion aufgefasst werden, sodass wir statt der Klassenbezeichnungen -1 und +1 stetige Zielwerte erhalten. Um die Ähnlichkeit zu demonstrieren, verwenden wir die Implementierung des

Gradientenabstiegsverfahrens für Adaline aus Kapitel 2, entfernen die Sprungfunktion und implementieren damit unser erstes lineares Regressionsmodell:

```
class LinearRegressionGD(object):
    def __init__(self, eta=0.001, n_iter=20):
        self.eta = eta
        self.n_iter = n_iter

    def fit(self, X, y):
        self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []

        for i in range(self.n_iter):
            output = self.net_input(X)
            errors = (y - output)
            self.w_[1:] += self.eta * X.T.dot(errors)
            self.w_[0] += self.eta * errors.sum()
            cost = (errors**2).sum() / 2.0
            self.cost_.append(cost)
        return self

    def net_input(self, X):
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def predict(self, X):
        return self.net_input(X)
```

### Tipp

Falls Sie Ihre Erinnerung in Bezug darauf, wie die Gewichtungen aktualisiert werden (sich einen Schritt in die entgegengesetzte Richtung des Gradienten bewegen), auffrischen möchten, können Sie dies in Kapitel 2 nachlesen.

Um den `LinearRegressionGD`-Regressor in Aktion zu sehen, verwenden wir die `RM`-Variable (Anzahl der Zimmer) der `Lebensbedingungen`-Datensammlung als erklärende Variable, um ein Modell zu trainieren, das den `MEDV`-Wert (Preis der Wohnung/des Hauses) vorhersagen kann. Außerdem standardisieren wir die Variablen, damit das Gradientenabstiegsverfahren besser konvergiert. Hier der Code:

```
>>> X = df[['RM']].values
>>> y = df['MEDV'].values
>>> from sklearn.preprocessing import StandardScaler
>>> sc_x = StandardScaler()
>>> sc_y = StandardScaler()
```

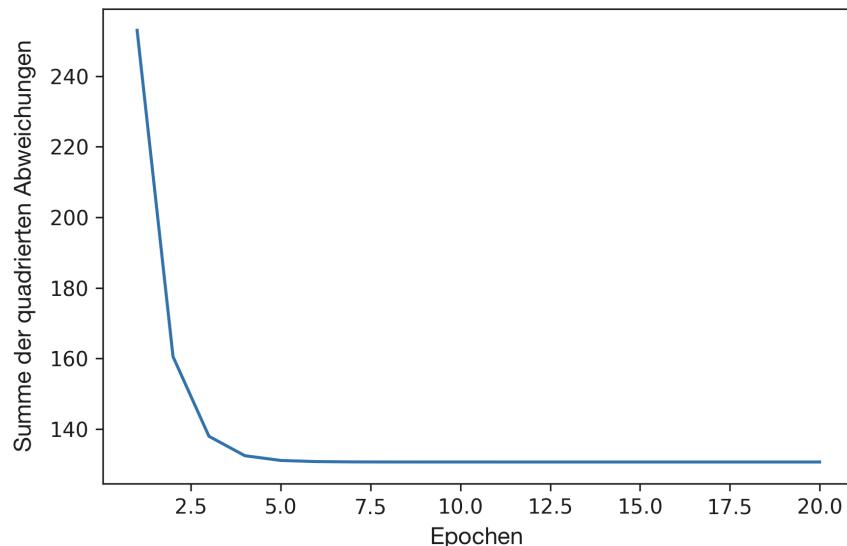
```
>>> X_std = sc_x.fit_transform(X)
>>> y_std = sc_y.fit_transform(y[:, np.newaxis]).flatten()
>>> lr = LinearRegressionGD()
>>> lr.fit(X_std, y_std)
```

Beachten Sie die Verwendung von `np.newaxis` und `flatten` bei der Zuweisung zu `y_std`. Die meisten Transformer in scikit-learn erwarten, dass die Daten in zweidimensionalen Arrays gespeichert sind. Im Codebeispiel wird dem Array durch die Verwendung von `np.newaxis` im Ausdruck `y[:, np.newaxis]` eine Dimension hinzugefügt. Nachdem der `StandardScaler` die skalierte Variable zurückgegeben hat, wird die ursprüngliche eindimensionale Form des Arrays durch den Aufruf von `flatten` wiederhergestellt.

In Kapitel 2 haben wir festgestellt, dass es immer lohnt, die Werte der Straffunktion gegen die der Anzahl der Epochen (Durchläufe der Trainingsdaten) aufzutragen, wenn wir einen Optimierungsalgorithmus wie das Gradientenabstiegsverfahren verwenden, um so zu überprüfen, ob der Algorithmus gegen ein Minimum (in diesem Fall ein *globales* Minimum) konvergiert.

```
>>> sns.reset_orig() # Matplotlib-Stil
>>> plt.plot(range(1, lr.n_iter+1), lr.cost_)
>>> plt.ylabel('Summe quadrierter Abweichungen')
>>> plt.xlabel('Epochen')
>>> plt.show()
```

Wie Sie in der Abbildung sehen, konvergiert das Gradientenabstiegsverfahren nach der fünften Epoche.



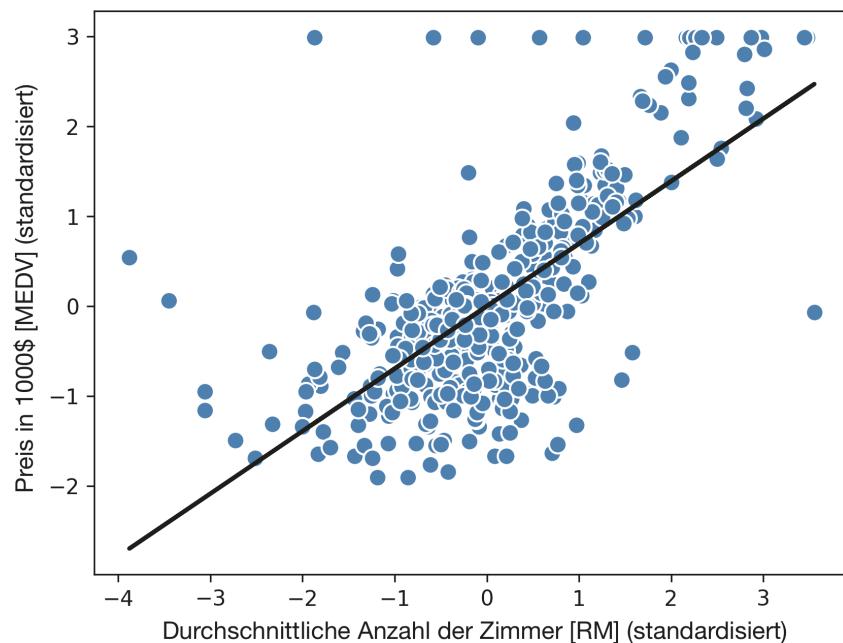
Als Nächstes wollen wir visualisieren, wie gut die lineare Regression an die Trainingsdaten angepasst ist. Zu diesem Zweck erstellen wir eine einfache Hilfsfunktion, die ein Streudiagramm der Trainingsdaten und die Regressionsgerade ausgibt:

```
>>> def lin_regplot(X, y, model):
...     plt.scatter(X, y, c='steelblue', edgecolor='white',
...                 s=70)
...     plt.plot(X, model.predict(X), color='black', lw=2)
...     return None
```

Nun nutzen wir diese `lin_regplot`-Funktion, um den Preis der Wohnimmobilien gegen die Anzahl der Zimmer aufzutragen:

```
>>> lin_regplot(X_std, y_std, lr)
>>> plt.xlabel('Durchschnittliche Anzahl der Zimmer [RM] (standardisiert)')
>>> plt.ylabel('Preis in 1000$ [MEDV] (standardisiert)')
>>> plt.show()
```

Wie Sie der Abbildung entnehmen können, spiegelt die lineare Regression die allgemeine Tendenz wider, dass Immobilien mit einer größeren Anzahl von Zimmern teurer sind:



Auf den ersten Blick erscheint das sinnvoll zu sein, allerdings sagen die Daten uns auch, dass die Anzahl der Zimmer die Immobilienpreise in vielen Fällen nicht besonders gut erklären kann. Später in diesem Kapitel werden wir erörtern, wie sich die Leistung eines Regressionsmodells quantifizieren lässt. Interessanterweise findet sich bei  $y = 3$  eine merkwürdige Häufung von Datenpunkten, was nahelegt, dass die Preise womöglich gedeckelt sind. Bei bestimmten Anwendungen kann es auch wichtig sein, die Vorhersagewerte im ursprünglichen Maßstab anzugeben. Um die vorhergesagten Preise wieder auf die Angaben in Einheiten von 1.000 Dollar zu skalieren, können wir einfach die `inverse_transform`-Methode des `StandardScalers` anwenden:

```
>>> num_rooms_std = sc_x.transform([5.0])
>>> price_std = lr.predict(num_rooms_std)
>>> print("Preis in 1000$: %.3f" % \
...       sc_y.inverse_transform(price_std))
Preis in 1000$: 10.840
```

Im vorstehenden Beispielcode haben wir das zuvor trainierte lineare Regressionsmodell verwendet, um den Preis von Häusern mit fünf Zimmern zu prognostizieren. Unserem Modell nach zu urteilen, ist solch ein Haus 10.840 Dollar wert.

Nebenbei bemerkt ist es erwähnenswert, dass wir rein technisch betrachtet die Gewichtung der y-Achsenabschnitte nicht aktualisieren müssen, sofern wir standariserte Variablen benutzen, da die y-Achsenabschnitte in diesen Fällen stets 0 sind. Das kann problemlos überprüft werden, indem wir die Gewichtungen ausgeben:

```
>>> print('Steigung: %.3f' % lr.w_[1])
Steigung: 0.695
>>> print('Achsenabschnitt: %.3f' % lr.w_[0])
Achsenabschitt: -0.000
```

### 10.3.2 Abschätzung der Koeffizienten eines Regressionsmodells mit scikit-learn

Im vorangegangenen Abschnitt haben wir ein funktionierendes Modell für die Regressionsanalyse implementiert. Allerdings sind wir im Falle praxisnaher Anwendungen womöglich an einer effizienteren Implementierung interessiert. Beispielsweise nutzen viele der scikit-learn-Schätzer für Regressionen die LIBLINEAR-Bibliothek, hochentwickelte Optimierungsalgorithmen und andere Code-optimierungen, die für nicht standardisierte Variablen besser geeignet sind – in bestimmten Anwendungsfällen ist das wünschenswert.

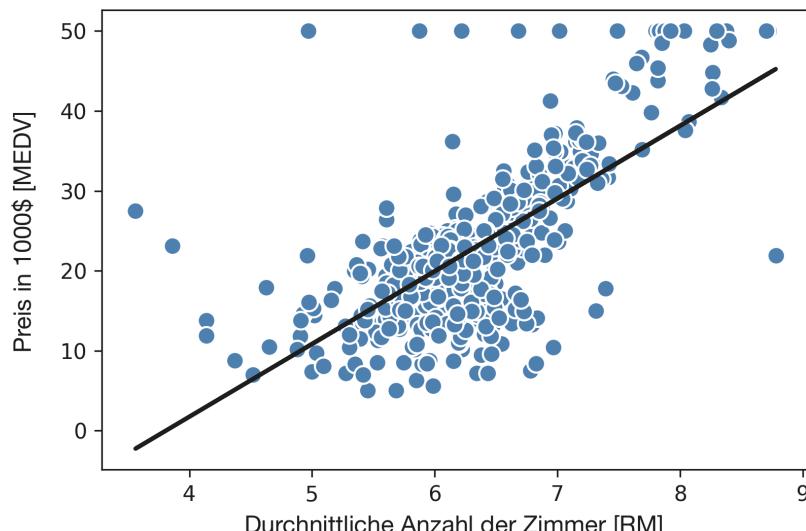
```
>>> from sklearn.linear_model import LinearRegression
>>> slr = LinearRegression()
```

```
>>> slr.fit(X, y)
>>> print('Steigung: %.3f' % slr.coef_[0])
Steigung: 9.102
>>> print('Achsenabschnitt: %.3f' % slr.intercept_)
Achsenabschnitt: -34.671
```

Beim Ausführen des Codes stellt man fest, dass das `LinearRegression`-Modell von scikit-learn für die nicht standardisierten Variablen RM und MEDV andere Koeffizienten liefert. Das soll nun mit unserer eigenen Implementierung des Gradientenabstiegsverfahrens verglichen werden, indem wir MEDV gegen RM auftragen:

```
>>> lin_regplot(X, y, slr)
>>> plt.xlabel('Durchschnittliche Anzahl der Zimmer [RM]')
>>> plt.ylabel('Preis in 1000$ [MEDV]')
>>> plt.show()
```

Wenn wir nun die Trainingsdaten und unser angepasstes Modell vergleichen, indem wir den obigen Code ausführen, wird deutlich, dass das Ergebnis im Großen und Ganzen mit dem unserer eigenen Implementierung übereinstimmt:



### Tipp

Als Alternative zur Verwendung von Bibliotheken für das Machine Learning bieten sich geschlossene analytische Lösungen in Form von Systemen linearer Gleichungen an, die in den meisten einführenden Statistiklehrbüchern zu finden sind:

$$w = (X^T X)^{-1} X^T y$$

In Python kann das folgendermaßen implementiert werden:

```
# Hinzufügen eines Spaltenvektors mit Einsen
>>> Xb = np.hstack((np.ones((X.shape[0], 1)), X))
>>> w = np.zeros(X.shape[1])
>>> z = np.linalg.inv(np.dot(Xb.T, Xb))
>>> w = np.dot(z, np.dot(Xb.T, y))
>>> print('Steigung: %.3f' % w[1])
Steigung: 9.102
>>> print('Achsenabschnitt: %.3f' % w[0])
Achsenabschnitt: -34.671
```

Diese Methode hat den Vorteil, dass sie die optimale Lösung garantiert analytisch findet. Wenn wir es allerdings mit sehr großen Datenmengen zu tun haben, kann es zu rechenaufwendig sein, die Matrix in dieser Formel (die auch als *Normalgleichung* bezeichnet wird) zu invertieren. Sie könnte auch singulär (nicht invertierbar) sein, daher ist in manchen Fällen die iterative Methode vorzuziehen.

Wenn Sie an weiteren Informationen zur Berechnung der Normalgleichungen interessiert sind, empfehle ich die Lektüre des Kapitels *The Classical Linear Regression Model* aus der Vorlesungsreihe von Dr. Stephen Pollock an der University of Leicester, die unter <http://www.le.ac.uk/users/dsgp1/COURSES/MESOMET/ECMETXT/06mesmet.pdf> zum kostenlosen Download zur Verfügung steht.

## 10.4 Anpassung eines robusten Regressionsmodells mit dem RANSAC-Algorithmus

Lineare Regressionsmodelle sind nicht robust gegenüber *Ausreißern* (*Outliers*) und können durch deren Vorhandensein stark beeinflusst werden. Unter bestimmten Umständen kann schon eine sehr kleine Teilmenge der Daten große Auswirkungen auf die Modellkoeffizienten haben. Es gibt eine Vielzahl statistischer Tests, um Ausreißer aufzuspüren, die jedoch über den Rahmen dieses Buches hinausgehen. Ob das Entfernen von Ausreißern angemessen ist, müssen Sie als Data Scientist auf der Grundlage Ihrer Fachkenntnis selbst beurteilen.

Als Alternative zum Entfernen der Ausreißer werden wir ein robustes Regressionsverfahren betrachten, das den *RANSAC-Algorithmus* (*RANdom SAmple Consensus*, etwa »Übereinstimmung mit einer zufälligen Stichprobe«) einsetzt, um ein Modell an eine Teilmenge der Daten anzupassen, die idealerweise keine Ausreißer mehr enthält, die sogenannten *Inliers*.

Der iterative RANSAC-Algorithmus besteht aus folgenden Schritten:

1. Wählen Sie eine zufällige Stichprobe aus, die als Inliers betrachtet werden kann, und berechnen Sie die Modellparameter.
2. Ermitteln Sie mit den berechneten Modellparametern die Abweichungen aller anderen Datenpunkte und fügen Sie diejenigen zu den Inliers hinzu, deren Abweichung unterhalb eines benutzerdefinierten Schwellenwertes liegt.
3. Berechnen Sie die Modellparameter für alle Inliers.
4. Schätzen Sie die Fehler des zuerst berechneten Modells und des Modells für die Inliers ab und vergleichen Sie diese.
5. Beenden Sie den Algorithmus, sobald die Leistung einen vorher festgelegten Schwellenwert erreicht oder eine vorgegebene Anzahl von Iterationen durchlaufen wurde. Fahren Sie anderenfalls mit Schritt 1 fort.

Nun soll unser lineares Modell mithilfe von scikit-learns `RANSACRegressor`-Klasse den RANSAC-Algorithmus einsetzen:

```
>>> from sklearn.linear_model import RANSACRegressor
>>> ransac = RANSACRegressor(LinearRegression(),
...                         max_trials=100,
...                         min_samples=50,
...                         loss='absolute_loss',
...                         residual_threshold=5.0,
...                         random_state=0)
>>> ransac.fit(X, y)
```

Wir setzen die maximale Anzahl der Iterationen des `RANSACRegressor` auf 100 und legen mit `min_samples=50` fest, dass die Größe der zufälligen Stichprobe mindestens 50 beträgt. Durch Angabe des Parameters `absolute_loss` berechnet der Algorithmus den Absolutbetrag der vertikalen Abstände zwischen den Datenpunkten und der Regressionsgeraden. Indem wir den Parameter `residual_threshold` auf 5.0 setzen, erlauben wir in der Menge der Inliers nur Datenpunkte, bei denen der vertikale Abstand zur Regressionsgeraden höchstens 5 Längeneinheiten beträgt – ein für diese spezielle Datensammlung gut geeigneter Wert.

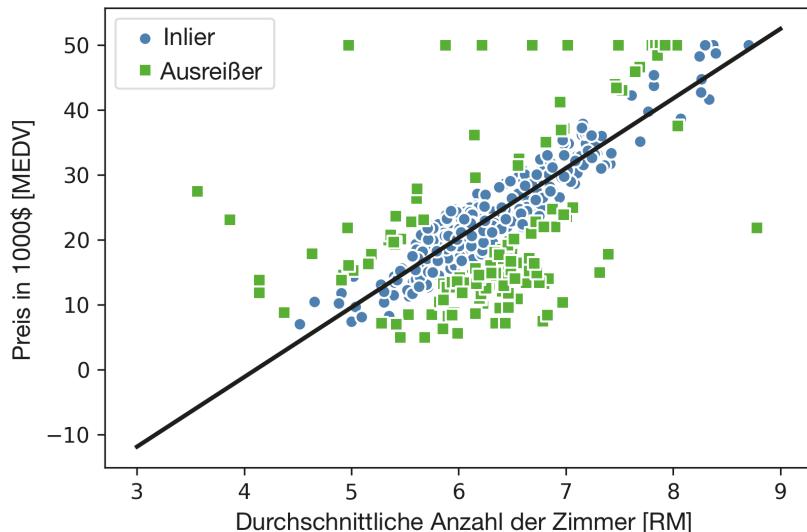
Standardmäßig verwendet scikit-learn als Schwellenwert zur Auswahl der Inliers die *MAD-Abschätzung*, wobei MAD den *Median der absoluten Abweichungen (Median Absolute Deviation)* der Zielwerte  $y$  bezeichnet. Die Auswahl eines geeigneten Inlier-Schwellenwertes ist allerdings von der jeweiligen Aufgabenstellung abhängig – ein Nachteil des RANSAC-Algorithmus. In den letzten Jahren wurden viele Ansätze entwickelt, den Schwellenwert zur Auswahl der Inliers automatisch zu bestimmen. R. Toldo und A. Fusiello haben das ausführlich erörtert (R. Toldo und A. Fusiello, *Automatic Estimation of the Inlier Threshold in Robust Multiple*

*Structures Fitting, Image Analysis and Processing–ICIAP 2009, Seiten 123-131. Springer, 2009).*

Nun ist das RANSAC-Modell bereit und wir können die Inliers und die Ausreißer, die das angepasste lineare Regressionsmodell mit dem RANSAC-Algorithmus liefert, zusammen mit der linearen Anpassung ausgeben.

```
>>> inlier_mask = ransac.inlier_mask_
>>> outlier_mask = np.logical_not(inlier_mask)
>>> line_X = np.arange(3, 10, 1)
>>> line_y_ransac = ransac.predict(line_X[:, np.newaxis])
>>> plt.scatter(X[inlier_mask], y[inlier_mask],
...                 c='steelblue', edgecolor='white',
...                 marker='o', label='Inlier')
>>> plt.scatter(X[outlier_mask], y[outlier_mask],
...                 c='limegreen', edgecolor='white',
...                 marker='s', label='Ausreißer')
>>> plt.plot(line_X, line_y_ransac, color='black')
>>> plt.xlabel('Durchschnittliche Anzahl der Zimmer [RM]')
>>> plt.ylabel('Preis in 1000$ [MEDV]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Wie man in dem Streudiagramm sieht, wurde das lineare Regressionsmodell an die als Inliers erkannten Datenpunkte (als Kreise dargestellt) angepasst.



Wenn wir durch Ausführung des folgenden Codes die Steigung und den Achsenabschnitt des Modells ausgeben, wird deutlich, dass sich die Regressionsgerade

ein wenig von derjenigen unterscheidet, die wir im vorangegangenen Abschnitt ohne RANSAC berechnet haben.

```
>>> print('Steigung: %.3f' % ransac.estimator_.coef_[0])
Steigung: 10.735
>>> print('Achsenabschnitt: %.3f'
...                      % ransac.estimator_.intercept_)
Achsenabschnitt: -44.089
```

Durch den Einsatz von RANSAC reduzieren wir die potenziellen Auswirkungen der Ausreißer in der Datenmenge, wir wissen jedoch nicht, ob dieser Ansatz einen positiven Effekt auf die Vorhersagekraft bei unbekannten Daten hat. Im nächsten Abschnitt werden wir uns daher mit verschiedenen Ansätzen zur Bewertung von Regressionsmodellen befassen, einem der entscheidenden Schritte bei der Entwicklung von Vorhersagemodellen.

## 10.5 Bewertung der Leistung linearer Regressionsmodelle

Im vorangegangenen Abschnitt haben wir erörtert, wie ein lineares Regressionsmodell an die Trainingsdatenmenge angepasst wird. Aus den vorangegangenen Kapiteln wissen Sie, dass es von entscheidender Bedeutung ist, das Modell mit Daten zu testen, die nicht Teil der Trainingsdaten sind, um eine weniger verzerrte Einschätzung der Leistung zu erhalten.

Wie in Kapitel 6 beschrieben, möchten wir unsere Datenmenge in separate Trainings- und Testdatenmengen aufteilen, wobei wir Erstere zum Anpassen des Modells und Letztere zur Beurteilung der Verallgemeinerungsfähigkeit bei unbekannten Daten verwenden. Anstatt mit dem einfachen Regressionsmodell fortzufahren, werden wir nun sämtliche Variablen der Datenmenge nutzen und ein multiples Regressionsmodell trainieren:

```
>>> from sklearn.model_selection import train_test_split
>>> X = df.iloc[:, :-1].values
>>> y = df['MEDV'].values
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.3, random_state=0)
>>> slr = LinearRegression()
>>> slr.fit(X_train, y_train)
>>> y_train_pred = slr.predict(X_train)
>>> y_test_pred = slr.predict(X_test)
```

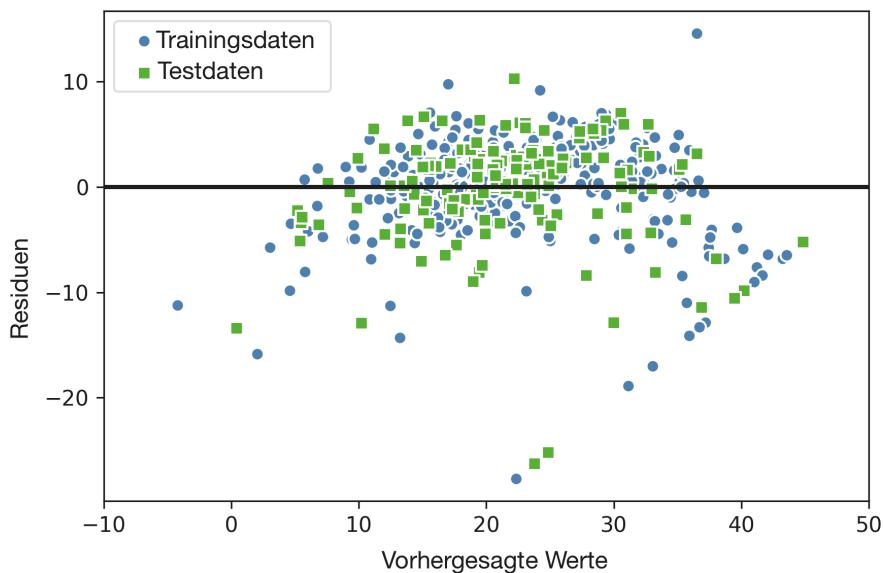
Da unser Modell mehrere erklärende Variablen besitzt, können wir die Regressionsgerade (oder genauer: die Regressions-Hyperebene) nicht in zweidimensionalen Diagrammen visualisieren – wir können jedoch die Residuen (die Abweichun-

gen oder vertikalen Abstände der tatsächlichen Werte von den vorhergesagten) gegen die vorhergesagten Werte auftragen, um unser Modell zu analysieren. Solche *Residualdiagramme* sind zur grafischen Analyse von Regressionsmodellen gebräuchlich, um Nichtlinearitäten und Ausreißer zu entdecken und um zu überprüfen, ob Fehler zufällig verteilt sind.

Mit dem folgenden Code stellen wir ein Residualdiagramm dar, bei dem wir einfach den tatsächlichen Wert der Zielvariablen von dem vorhergesagten Wert subtrahieren:

```
>>> plt.scatter(y_train_pred, y_train_pred - y_train,
...                 c='steelblue', marker='o',
...                 edgecolor='white', label='Trainingsdaten')
>>> plt.scatter(y_test_pred, y_test_pred - y_test,
...                 c='limegreen', marker='s',
...                 edgecolor='white', label='Testdaten')
>>> plt.xlabel('Vorhergesagte Werte')
>>> plt.ylabel('Residuen')
>>> plt.legend(loc='upper left')
>>> plt.hlines(y=0, xmin=-10, xmax=50, lw=2, color='black')
>>> plt.xlim([-10, 50])
>>> plt.show()
```

Nach der Ausführung des Codes sollte wie in der Abbildung ein Residualdiagramm mit einer durchgezogenen Linie durch den Ursprung entlang der x-Achse angezeigt werden.



Bei einer perfekten Vorhersage wären sämtliche Residuen exakt null – in der Praxis und bei echten Anwendungen werden wir so etwas jedoch wohl kaum zu Gesicht bekommen. Bei einem vernünftigen Regressionsmodell würden wir allerdings erwarten, dass die Fehler zufällig verteilt und die Residuen gleichmäßig um die Mittellinie gestreut sind. Wenn in einem Residualdiagramm Muster erkennbar sind, ist das ein Hinweis darauf, dass unser Modell nicht in der Lage ist, einige der erklärenden Informationen zu erfassen – und das zeigt sich in den verbleibenden Residuen, wie in obigem Residualdiagramm ebenfalls in gewissem Maße erkennbar ist. Darüber hinaus sind Residualdiagramme dafür geeignet, Ausreißer zu identifizieren, die durch die Datenpunkte mit großer Abweichung von der Mittellinie repräsentiert werden.

Die sogenannte *mittlere quadratische Abweichung (MSE, Mean Squared Error)* ist ein weiteres nützliches quantitatives Maß für die Leistung eines Modells. Sie repräsentiert den Durchschnittswert der Straffunktion (die Summe quadrierter Abweichungen), die wir beim linearen Regressionsmodell minimieren. Die mittlere quadratische Abweichung erweist sich beim Vergleichen verschiedener Regressionsmodelle oder bei der Parameterabstimmung per Rastersuche/Kreuzvalidierung als nützlich.

$$MSE = \frac{1}{n} \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2$$

Berechnen wir also die mittlere quadratische Abweichung der Vorhersagen für Trainings- und Testdaten:

```
>>> from sklearn.metrics import mean_squared_error
>>> print('MSE-Trainingsdaten: %.3f, Test: %.3f' % (
    mean_squared_error(y_train, y_train_pred),
    mean_squared_error(y_test, y_test_pred)))
MSE-Trainingsdaten: 19.958 Test: 27.196
```

Die mittlere quadratische Abweichung der Trainingsdaten beträgt 19.96. Diejenige der Testdaten ist mit 27.20 bedeutend größer – ein Hinweis darauf, dass es bei unserem Modell im Fall der Trainingsdaten zu einer Überanpassung kommt.

Manchmal ist es sinnvoller, das *Bestimmtheitsmaß ( $R^2$ )* anzugeben, das als standariserte Version der mittleren quadratischen Abweichung aufgefasst werden kann, die es erleichtert, die Leistung des Modells zu interpretieren. Oder anders ausgedrückt:  $R^2$  ist der Anteil der Varianz der Zielvariablen, die das Modell erfasst. Der Wert für  $R^2$  ist folgendermaßen definiert:

$$R^2 = 1 - \frac{SSE}{SST}$$

Hier bezeichnet  $SSE$  die *Summe der quadrierten Abweichungen (Sum of Squared Errors)* und  $SST$  die *Gesamtsumme der Quadrate (Sum of Squares Total)*

$SST = \sum_{i=1}^n (y^{(i)} - \mu_y)^2$ , oder anders formuliert: die Varianz der Zielvariablen.

Wir zeigen kurz, dass  $R^2$  tatsächlich nur eine neu skalierte Version der mittleren quadratischen Abweichung ist:

$$R^2 = 1 - \frac{SSE}{SST}$$

$$1 - \frac{\frac{1}{n} \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2}{\frac{1}{n} \sum_{i=1}^n (y^{(i)} - \mu_y)^2}$$

$$1 - \frac{MSE}{Var(y)}$$

Bei der Trainingsdatenmenge nimmt  $R^2$  Werte zwischen 0 und 1 an, kann jedoch bei der Testdatenmenge negativ werden. Falls  $R^2 = 1$  ist, gibt das Modell die Daten perfekt wieder, dementsprechend ist dann  $MSE = 0$ .

Für die Trainingsdaten ergibt sich bei unserem Modell für  $R^2$  ein Wert von 0.765 – gar nicht mal so schlecht. Allerdings beträgt der Wert von  $R^2$  bei den Testdaten nur 0.673, wie wir durch die Ausführung des folgenden Codes errechnen können:

```
>>> from sklearn.metrics import r2_score
>>> print('R^2 Training: %.3f, Test: %.3f' %
...      (r2_score(y_train, y_train_pred),
...       r2_score(y_test, y_test_pred)))
R^2 Training: 0.765 Test: 0.673
```

## 10.6 Regularisierungsverfahren für die Regression einsetzen

Wie Sie aus Kapitel 3 wissen, ist die Regularisierung ein Ansatz, um das Problem der Überanpassung in Angriff zu nehmen, indem weitere Informationen hinzugefügt und die Parameterwerte des Modells gesenkt werden, sodass Komplexität höhere Werte der Straffunktion zur Folge hat. Die verbreitetsten Ansätze zur Regularisierung linearer Regressionen sind die sogenannte *Ridge-Regression*, *LASSO (Least Absolute Shrinkage and Selection Operator*, ein der Methode der kleinsten Quadrate verwandtes Verfahren) und das *Elastic-Net-Verfahren*.

Die Ridge-Regression ist eine L2-Regularisierung, bei der wir einfach die quadrierte Summe der Gewichtungen zur Straffunktion hinzufügen:

$$J(w)_{\text{Ridge}} = \sum_{i=1}^n \left( y^{(i)} - \hat{y}^{(i)} \right)^2 + \lambda \|w\|_2^2$$

Hier gilt:

$$L2: \lambda \|w\|_2^2 = \lambda \sum_{j=1}^m w_j^2$$

Indem wir dem Hyperparameter  $\lambda$  größere Werte zuweisen, erhöhen wir die Regularisierungsstärke und verringern die Gewichtungen unseres Modells. Beachten Sie, dass der Achsenabschnitts-Term  $w_0$  nicht regularisiert wird.

LASSO ist ein alternativer Ansatz, der zu dünnbesetzten Modellen führen kann. Je nach Regularisierungsstärke können bestimmte Gewichtungen null werden, wodurch LASSO auch als überwachtes Verfahren zur Merkmalsauswahl fungieren kann:

$$J(w)_{\text{LASSO}} = \sum_{i=1}^n \left( y^{(i)} - \hat{y}^{(i)} \right)^2 + \lambda \|w\|_1$$

Hier gilt:

$$L1: \lambda \|w\|_1 = \lambda \sum_{j=1}^m |w_j|$$

Bei LASSO gibt es allerdings die Einschränkung, dass höchstens  $n$  Variablen ausgewählt werden, wenn  $m > n$  ist. Das *Elastic-Net-Verfahren* stellt einen Kompromiss zwischen Ridge-Regression und LASSO dar, bei dem es einen L1-Strafterm gibt, der die dünne Besetzung erzeugt, und einen L2-Strafterm, der einige der für LASSO geltenden Einschränkungen umgeht, wie etwa die Anzahl der ausgewählten Variablen.

$$J(w)_{\text{ElasticNet}} = \sum_{i=1}^n \left( y^{(i)} - \hat{y}^{(i)} \right)^2 + \lambda_1 \sum_{j=1}^m w_j^2 + \lambda_2 \sum_{j=1}^m |w_j|$$

All diese regularisierten Regressionsmodelle sind in scikit-learn verfügbar und sie werden auch sehr ähnlich wie die regulären Regressionsmodelle eingesetzt, mit der Ausnahme, dass wir in diesem Fall die Regularisierungsstärke über den Parameter  $\lambda$  angeben müssen, der beispielsweise durch k-fache Kreuzvalidierung optimiert wurde.

Eine Ridge-Regression kann folgendermaßen initialisiert werden:

```
>>> from sklearn.linear_model import Ridge
>>> ridge = Ridge(alpha=1.0)
```

Beachten Sie, dass die Regularisierungsstärke auf ähnliche Weise wie mit dem Parameter  $\lambda$  durch `alpha` gesteuert wird. Ein LASSO-Regressor kann über das `linear_model`-Submodul initialisiert werden:

```
>>> from sklearn.linear_model import Lasso
>>> lasso = Lasso(alpha=1.0)
```

Und die `ElasticNet`-Implementierung erlaubt es uns, das Verhältnis von L1 zu L2 zu variieren:

```
>>> from sklearn.linear_model import ElasticNet
>>> lasso = ElasticNet(alpha=1.0, l1_ratio=0.5)
```

Wenn wir `l1_ratio` beispielsweise auf 1.0 setzen, würde der `ElasticNet`-Regressor einer LASSO-Regression entsprechen. Weitere Informationen über die verschiedenen Implementierungen linearer Regressionen finden Sie in der Dokumentation unter [http://scikit-learn.org/stable/modules/linear\\_model.html](http://scikit-learn.org/stable/modules/linear_model.html).

## 10.7 Polynomiale Regression: Umwandeln einer linearen Regression in eine Kurve

In den vorangegangenen Abschnitten haben wir eine lineare Beziehung zwischen den erklärenden Variablen und der Zielvariable vorausgesetzt. Wenn diese Annahme nicht stimmt, kann dem durch ein polynomiales Regressionsmodell Rechnung getragen werden, das zusätzliche polynomiale Terme besitzt:

$$y = w_0 + w_1 x + w_2 x^2 + \dots + w_d x^d$$

Hier gibt  $d$  den Grad des Polynoms an. Wir könnten zwar eine polynomiale Regression einsetzen, um eine nichtlineare Beziehung zu modellieren, die Gleichung wird jedoch dessen ungeachtet wegen der linearen Regressionskoeffizienten  $w$  als multiples lineares Regressionsmodell betrachtet. In den folgenden Abschnitten werden wir uns ansehen, wie man solche polynomialen Terme zu einer vorhandenen Datenmenge komfortabel hinzufügen kann, und ein polynomiales Regressionsmodell anpassen.

### 10.7.1 Hinzufügen polynomialer Terme mit scikit-learn

Wir betrachten nun, wie sich die Transformerklasse `PolynomialFeatures` von scikit-learn dazu verwenden lässt, einer einfachen Regression mit einer erklärenden Variablen einen quadratischen Term ( $d = 2$ ) hinzuzufügen und vergleichen die polynomiale und die lineare Anpassung. Die Schritte im Einzelnen:

1. Fügen Sie einen polynomialen Term zweiten Grades hinzu:

```
from sklearn.preprocessing import PolynomialFeatures
>>> X = np.array([258.0, 270.0, 294.0, 320.0, 342.0,
...                 368.0, 396.0, 446.0, 480.0, 586.0])
...
...             [:, np.newaxis]
>>> y = np.array([236.4, 234.4, 252.8, 298.6, 314.2,
...                 342.2, 360.8, 368.0, 391.2, 390.8])
>>> lr = LinearRegression()
>>> pr = LinearRegression()
>>> quadratic = PolynomialFeatures(degree=2)
>>> X_quad = quadratic.fit_transform(X)
```

2. Passen Sie zu Vergleichszwecken ein einfaches Regressionsmodell an:

```
>>> lr.fit(X, y)
>>> X_fit = np.arange(250, 600, 10)[:, np.newaxis]
>>> y_lin_fit = lr.predict(X_fit)
```

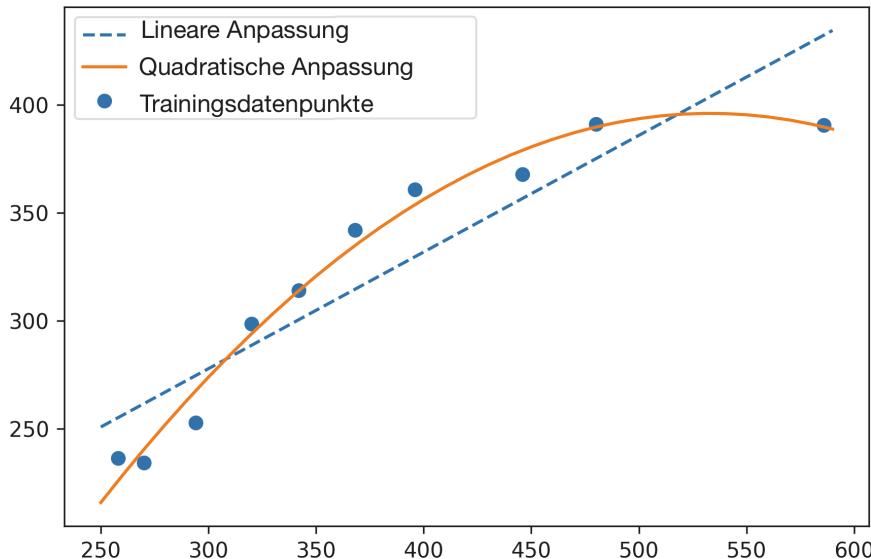
3. Passen Sie ein multiples Regressionsmodell mit polynomialer Regression an die transformierten Merkmale an:

```
>>> pr.fit(X_quad, y)
>>> y_quad_fit = pr.predict(quadratic.fit_transform(X_fit))
```

4. Ausgabe der Ergebnisse:

```
>>> plt.scatter(X, y, label='Trainingsdatenpunkte')
>>> plt.plot(X_fit, y_lin_fit,
...             label='Lineare Anpassung', linestyle='--')
>>> plt.plot(X_fit, y_quad_fit,
...             label='Quadratische Anpassung')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

In dem resultierenden Diagramm ist erkennbar, dass die polynomiale Anpassung die Beziehung zwischen der Zielvariablen und den erklärenden Variablen erheblich besser erfasst als die lineare Anpassung.



```
>>> y_lin_pred = lr.predict(X)
>>> y_quad_pred = pr.predict(X_quad)
>>> print('MSE-Training linear: %.3f, quadratisch: %.3f'
...      % (mean_squared_error(y, y_lin_pred),
...           mean_squared_error(y, y_quad_pred)))
MSE-Training linear: 569.780, quadratisch: 61.330
>>> print('R^2-Training linear: %.3f, quadratisch: %.3f'
...      % (r2_score(y, y_lin_pred),
...           r2_score(y, y_quad_pred)))
R^2-Training linear: 0.832, quadratisch: 0.982
```

Nach der Ausführung des obigen Codes wird deutlich, dass die mittlere quadratische Abweichung MSE von 570 (lineare Anpassung) auf 61 (quadratische Anpassung) sinkt – und das Bestimmtheitsmaß des quadratischen Modells ( $R^2 = 0.982$ ) spiegelt in diesem speziellen Beispiel die bessere Anpassung im Vergleich zum linearen Modell ( $R^2 = 0.832$ ) wider.

### 10.7.2 Modellierung nichtlinearer Zusammenhänge in der Lebensbedingungen-Datensammlung

Nachdem wir anhand eines einfachen Beispiels erörtert haben, wie nichtlineare Beziehungen durch ein polynomiales Modell angepasst werden können, wollen wir im Folgenden ein konkreteres Beispiel betrachten und dieses Konzept auf die Lebensbedingungen-Datensammlung anwenden. Durch die Ausführung des folgenden Codes wird der Zusammenhang zwischen Immobilienpreisen und dem Merkmal LSTAT (Prozentsatz der Bevölkerung mit niedrigem Sozialstatus) durch

Polynome zweiten und dritten Grades (quadratische bzw. kubische Polynome) modelliert und mit einem linearen Modell verglichen:

```
>>> X = df[['LSTAT']].values
>>> y = df['MEDV'].values
>>> regr = LinearRegression()

# Polynomiale Merkmale einrichten
>>> quadratic = PolynomialFeatures(degree=2)
>>> cubic = PolynomialFeatures(degree=3)
>>> X_quad = quadratic.fit_transform(X)
>>> X_cubic = cubic.fit_transform(X)

# Anpassungen vornehmen
>>> X_fit = np.arange(X.min(), X.max(), 1)[:, np.newaxis]

>>> regr = regr.fit(X, y)
>>> y_lin_fit = regr.predict(X_fit)
>>> linear_r2 = r2_score(y, regr.predict(X))

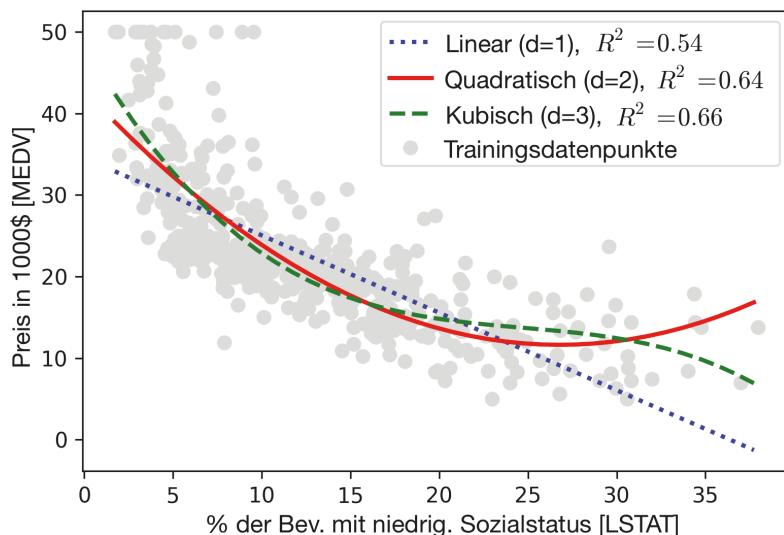
>>> regr = regr.fit(X_quad, y)
>>> y_quad_fit = regr.predict(quadratic.fit_transform(X_fit))
>>> quadratic_r2 = r2_score(y, regr.predict(X_quad))

>>> regr = regr.fit(X_cubic, y)
>>> y_cubic_fit = regr.predict(cubic.fit_transform(X_fit))
>>> cubic_r2 = r2_score(y, regr.predict(X_cubic))

# Ergebnisse ausgeben
>>> plt.scatter(X, y,
...                 label='Trainingdatenpunkte',
...                 color='lightgray')
>>> plt.plot(X_fit, y_lin_fit,
...             label='Linear (d=1), $R^2=% .2f$' % linear_r2,
...             color='blue',
...             lw=2,
...             linestyle=':')
>>> plt.plot(X_fit, y_quad_fit,
...             label='Quadratisch (d=2), $R^2=% .2f$'
...             % quadratic_r2,
...             color='red',
...             lw=2,
...             linestyle='--')
```

```
>>> plt.plot(X_fit, y_cubic_fit,
...             label='Kubisch (d=3), $R^2=%.2f$' % cubic_r2,
...             color='green',
...             lw=2,
...             linestyle='--')
>>> plt.xlabel('% der Bev. mit niedr. Sozialstatus [LSTAT]')
>>> plt.ylabel('Preis in 1000$ [MEDV]')
>>> plt.legend(loc='upper right')
>>> plt.show()
```

Das resultierende Diagramm sieht so aus:



In dem Diagramm ist erkennbar, dass das kubische Modell den Zusammenhang zwischen Immobilienpreisen und LSTAT besser erfasst als das quadratische und das lineare. Wir müssen uns allerdings bewusst sein, dass das Hinzufügen weiterer Polynome die Komplexität des Modells erhöht und somit auch die Wahrscheinlichkeit einer Überanpassung steigt. In der Praxis sollte man daher die Leistung eines Modells immer auch anhand einer separaten Testdatenmenge beurteilen, um die Verallgemeinerungsfähigkeit einzuschätzen zu können.

Darüber hinaus sind Polynome nicht immer unbedingt die beste Wahl für die Modellierung nichtlinearer Zusammenhänge. Wenn man das MEDV-LSTAT-Streudiagramm betrachtet, könnte man beispielsweise auf den Gedanken kommen, dass eine logarithmische Transformation der LSTAT-Merkalsvariablen und die Quadratwurzel der MEDV-Werte einen linearen Merkmalsraum ergeben würden, der für ein lineares Regressionsmodell geeignet wäre. So habe ich beispielsweise

den Eindruck, dass der Zusammenhang zwischen den beiden Variablen sehr nach einer Exponentialfunktion aussieht:

$$f(x) = 2^{-x}$$

Und da der natürliche Logarithmus einer Exponentialfunktion im Diagramm eine Gerade ergibt, vermute ich, dass hier eine logarithmische Transformation anwendbar ist:

$$\log(f(x)) = -x$$

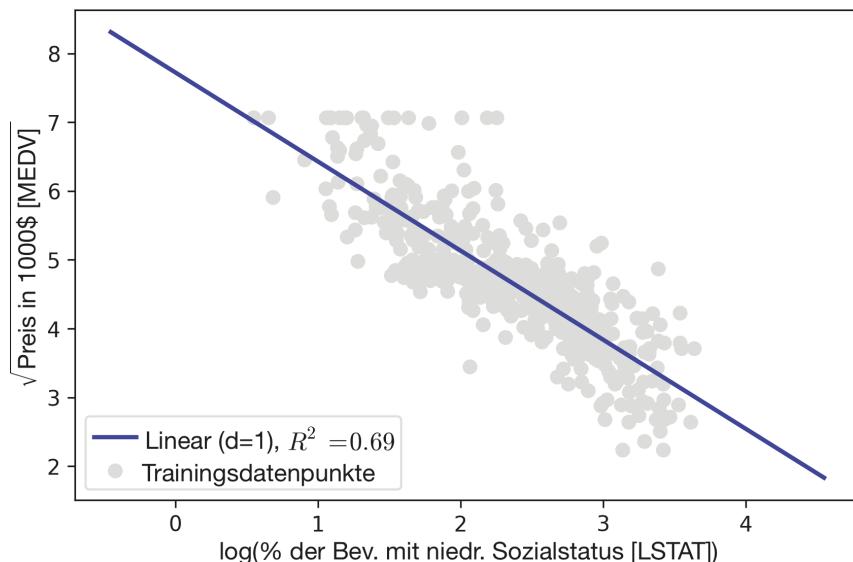
Diese Hypothese wollen wir nun mit folgendem Code überprüfen:

```
# Merkmale transformieren
>>> X_log = np.log(X)
>>> y_sqrt = np.sqrt(y)

# Merkmale anpassen
>>> X_fit = np.arange(X_log.min()-1,
...                     X_log.max()+1, 1)[:, np.newaxis]
>>> regr = regr.fit(X_log, y_sqrt)
>>> y_lin_fit = regr.predict(X_fit)
>>> linear_r2 = r2_score(y_sqrt, regr.predict(X_log))

# Ergebnisse ausgeben
>>> plt.scatter(X_log, y_sqrt,
...               label='Trainingsdatenpunkte',
...               color='lightgray')
>>> plt.plot(X_fit, y_lin_fit,
...            label='Linear (d=1), $R^2=% .2f$' % linear_r2,
...            color='blue',
...            lw=2)
>>> plt.xlabel('log(% der Bev. mit niedr. Sozialstatus [LSTAT])')
>>> plt.ylabel('$\sqrt{\text{Preis}} \text{ in } \$1000 \text{ [MEDV]}}$')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

Nach der Transformation der erklärenden Variablen auf eine logarithmische Skala und dem Ziehen der Quadratwurzel der Zielvariablen können wir die Beziehung zwischen den beiden Variablen mit einer linearen Regression beschreiben, die die Daten besser wiedergibt ( $R^2 = 0.69$ ) als die zuvor durchgeführten polynomiauen Merkmalstransformationen.



## 10.8 Handhabung nichtlinearer Beziehungen mit Random Forests

In diesem Abschnitt werden wir die *Random-Forest-Regression* betrachten, die sich konzeptionell von den bisher in diesem Kapitel verwendeten Regressionsmodellen unterscheidet. Ein Random Forest stellt ein Ensemble mehrerer Entscheidungsbäume dar, das im Gegensatz zu den bislang betrachteten globalen linearen und polynomialem Regressionsmodellen als die Gesamtheit stückweise linearer Funktionen aufgefasst werden kann. Oder anders ausgedrückt: Beim Entscheidungsbau-Algorithmus unterteilen wir den Eingaberaum in kleinere, besser zu handhabende Bereiche.

### 10.8.1 Entscheidungsbaum-Regression

Wenn wir es mit nichtlinearen Daten zu tun haben, hat der Entscheidungsbaum-Algorithmus den Vorteil, dass keine Merkmalstransformationen erforderlich sind. Aus Kapitel 3 wissen Sie, dass der Entscheidungsbaum dadurch wächst, dass wir die Baumknoten iterativ aufteilen, bis die Blätter keine Unreinheiten mehr aufweisen oder eine Abbruchbedingung erfüllt ist. Bei der Klassifizierung mit Entscheidungsbäumen haben wir die Entropie als ein Maß für die Unreinheit definiert, um zu ermitteln, welche Aufteilung den *Informationsgewinn (IG)* maximiert. Dieser kann wie folgt als binäre Aufteilung definiert werden:

$$IG(D_p, x_i) = I(D_p) - \frac{N_{links}}{N_p} I(D_{links}) - \frac{N_{rechts}}{N_p} I(D_{rechts})$$

Hier gibt  $x$  das Merkmal an, an dem die Aufteilung stattfinden soll,  $N_p$  ist die Anzahl der Objekte im Elternknoten,  $I$  ist ein Maß für die *Unreinheit (Impurity)*,  $D_p$  ist die Teilmenge der Trainingsdaten im Elternknoten, und  $D_{links}$  bzw.  $D_{rechts}$  sind die Teilmengen im linken und rechten Kindknoten nach der Aufteilung. Beachten Sie, dass wir zum Ziel haben, diejenige Aufteilung zu finden, die den Informationsgewinn maximiert, das heißt, wir möchten die Aufteilung finden, die Unreinheiten in den Kindknoten am stärksten verringert. In Kapitel 3 wurden Entropie und Gini-Koeffizient als Maße der Unreinheit erörtert, beides nützliche Klassifizierungskriterien. Um jedoch einen Entscheidungsbaum für die Regression verwenden zu können, benötigen wir ein Maß, das für stetige Werte geeignet ist, deshalb definieren wir als Maß für die Unreinheit eines Knotens  $t$  die mittlere quadratische Abweichung MSE:

$$I(t) = MSE(t) = \frac{1}{N_t} \sum_{i \in D_t} \left( y^{(i)} - \hat{y}_t \right)^2$$

Hier bezeichnet  $N_t$  die Anzahl der Trainingsobjekte des Knotens  $t$ ,  $D_t$  die Trainingsteilmenge des Knotens  $t$ ,  $y^{(i)}$  den tatsächlichen Zielwert und  $\hat{y}_t$  den vorhergesagten Zielwert (Mittelwert der Stichprobe):

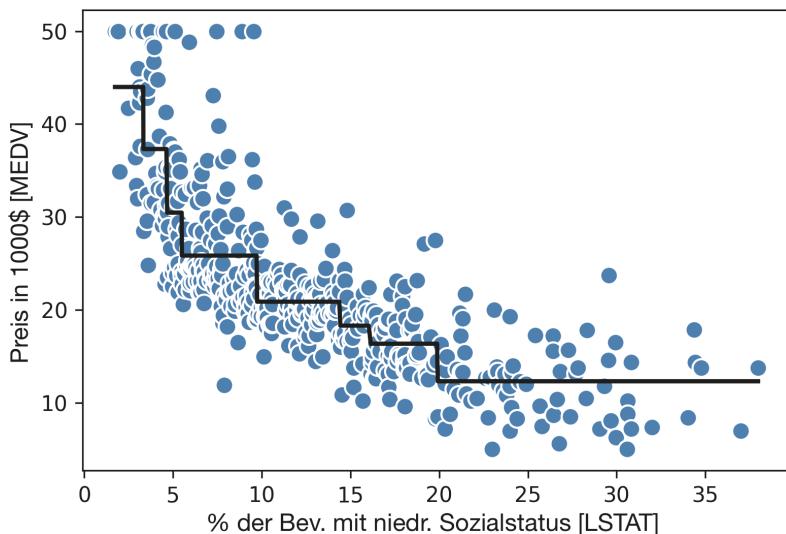
$$\hat{y}_t = \frac{1}{N} \sum_{i \in D_t} y^{(i)}$$

Im Kontext von Entscheidungsbaum-Regressionen wird die mittlere quadratische Abweichung manchmal auch als »Varianz innerhalb des Knotens« bezeichnet, deshalb heißt das Aufteilungsverfahren auch *Varianzreduktion*. Um uns die Anpassung durch einen Entscheidungsbaum zu vergegenwärtigen, verwenden wir den in scikit-learn implementierten **DecisionTreeRegressor** zur Modellierung des nichtlinearen Zusammenhangs zwischen den Variablen MEDV und LSTAT:

```
>>> from sklearn.tree import DecisionTreeRegressor
>>> X = df[['LSTAT']].values
>>> y = df['MEDV'].values
>>> tree = DecisionTreeRegressor(max_depth=3)
>>> tree.fit(X, y)
>>> sort_idx = X.flatten().argsort()
>>> lin_regplot(X[sort_idx], y[sort_idx], tree)
>>> plt.xlabel('% der Bev. mit niedr. Sozialstatus [LSTAT]')
>>> plt.ylabel('Preis in 1000$ [MEDV]')
>>> plt.show()
```

Das resultierende Diagramm macht deutlich, dass der Entscheidungsbaum den allgemeinen Trend in den Daten erkennt. Dieses Modell unterliegt allerdings der

Einschränkung, dass es die Stetigkeit und Differenzierbarkeit der gewünschten Vorhersage nicht widerspiegelt. Außerdem müssen wir für die Tiefe des Baums mit Bedacht einen geeigneten Wert wählen, damit es nicht zu einer Über- oder Unteranpassung an die Daten kommt. Hier scheint der Wert 3 eine gute Wahl zu sein:



Im nächsten Abschnitt betrachten wir eine robustere Methode zur Anpassung von Regressionsbäumen, nämlich Random Forests.

### 10.8.2 Random-Forest-Regression

Wie Sie aus Kapitel 3 wissen, ist der Random-Forest-Algorithmus eine Ensemblemethode, die mehrere Entscheidungsbäume miteinander kombiniert. Random Forests besitzen für gewöhnlich eine bessere Verallgemeinerungsfähigkeit als einzelne Entscheidungsbäume, da die Zufälligkeit die Varianz des Modells verringert. Sie haben außerdem den Vorteil, dass sie weniger empfindlich auf Ausreißer reagieren und kaum eine Abstimmung der Parameter erforderlich ist. Bei einem Random Forest müssen wir typischerweise nur mit einem Parameter herumexperimentieren, nämlich mit der Anzahl der Entscheidungsbäume im Ensemble. Der Algorithmus einer einfachen Random-Forest-Regression ist nahezu identisch mit dem in Kapitel 3 vorgestellten Algorithmus einer Random-Forest-Klassifizierung. Der einzige Unterschied besteht darin, dass wir beim Konstruieren der einzelnen Entscheidungsbäume die mittlere quadratische Abweichung als Kriterium heranziehen und dass die vorhergesagte Variable als Durchschnittswert aller Entscheidungsbäume berechnet wird.

Wir verwenden nun sämtliche Merkmale der Lebensbedingungen-Datensammlung, um ein Random-Forest-Regressionsmodell an 60 Prozent der Daten anzupassen und dessen Leistung anhand der verbleibenden 40 Prozent der Daten zu beurteilen. Hier der Code:

```
>>> X = df.iloc[:, :-1].values
>>> y = df['MEDV'].values
>>> X_train, X_test, y_train, y_test =
...     train_test_split(X, y,
...                       test_size=0.4,
...                       random_state=1)
>>> from sklearn.ensemble import RandomForestRegressor
>>> forest = RandomForestRegressor(n_estimators=1000,
...                                 criterion='mse',
...                                 random_state=1,
...                                 n_jobs=-1)
>>> forest.fit(X_train, y_train)
>>> y_train_pred = forest.predict(X_train)
>>> y_test_pred = forest.predict(X_test)
>>> print('MSE-Training: %.3f, Test: %.3f' %
...       mean_squared_error(y_train, y_train_pred),
...       mean_squared_error(y_test, y_test_pred)))
MSE-Training: 1.642, Test: 11.052
>>> print('R^2-Training: %.3f, Test: %.3f' %
...       r2_score(y_train, y_train_pred),
...       r2_score(y_test, y_test_pred)))

R^2-Training: 0.979, Test: 0.871
```

Leider müssen wir feststellen, dass es beim Random Forest tendenziell zu einer Überanpassung der Trainingsdaten kommt. Der Zusammenhang zwischen Zielvariable und erklärenden Variablen wird jedoch trotzdem relativ deutlich erkannt ( $R^2 = 0.871$  bei der Testdatenmenge).

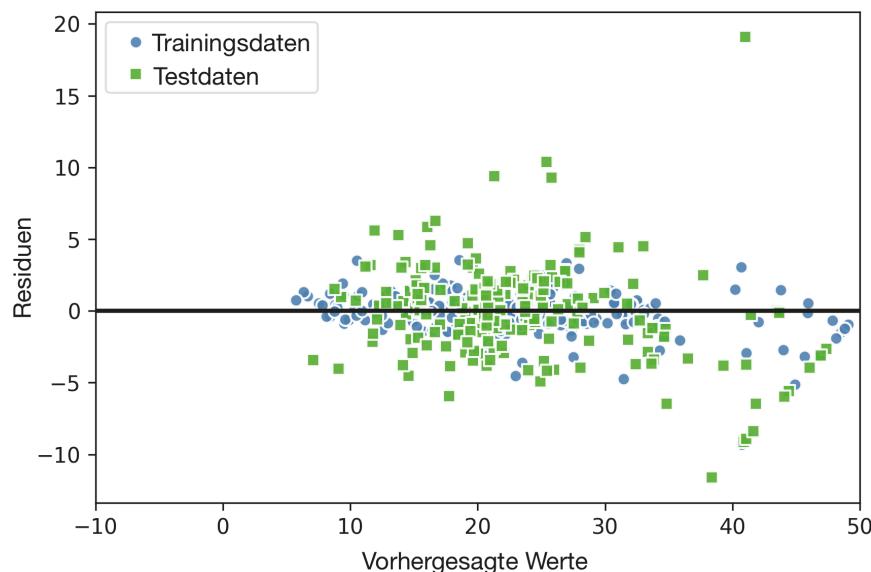
Zum Abschluss werfen wir noch einen Blick auf die Residuen der Vorhersage:

```
>>> plt.scatter(y_train_pred,
...               y_train_pred - y_train,
...               c='black',
...               edgecolor='white',
...               marker='o',
...               s=35,
...               alpha=0.9,
...               label='Trainingsdaten')
```

```
>>> plt.scatter(y_test_pred,
...                 y_test_pred - y_test,
...                 c='limegreen',
...                 marker='s',
...                 s=35,
...                 alpha=0.9,
...                 label='Testdaten')
>>> plt.xlabel('Vorhergesagte Werte')
>>> plt.ylabel('Residuen')
>>> plt.legend(loc='upper left')
>>> plt.hlines(y=0, xmin=-10, xmax=50, lw=2, color='black')
>>> plt.xlim([-10, 50])
>>> plt.show()
```

Dem  $R^2$ -Koeffizienten war schon anzusehen, dass das Modell die Trainingsdaten besser reproduziert als die Testdaten, wie die Ausreißer in Richtung der y-Achse zeigen. Außerdem scheint auch die Verteilung der Residuen um die Grundlinie bei 0 nicht völlig zufällig zu sein, was darauf hinweist, dass dieses Modell nicht in der Lage ist, alle erklärenden Informationen zu erfassen.

Allerdings zeigt dieses Residualdiagramm eine erhebliche Verbesserung im Vergleich zu dem früher in diesem Kapitel erstellten Residualdiagramm des linearen Modells.



Im Idealfall sollten die Fehler des Modells zufällig oder unvorhersagbar sein. Mit anderen Worten: Die Fehler der Vorhersagen sollten in keinem Zusammenhang mit den in den erklärenden Variablen enthaltenen Informationen stehen, sondern

vielmehr die in der Praxis auftretende Zufälligkeit der Verteilungen oder Muster widerspiegeln. Wenn bei den Vorhersagefehlern Muster beobachtbar sind, die beispielsweise durch das Erstellen eines Residualdiagramms sichtbar werden, stecken in dem Diagramm noch Informationen, die zur Vorhersage nutzbar sind. Dass erklärende Informationen in die Residuen einfließen, ist ein häufiger Grund hierfür.

Irgendwie muss man die Nicht-Zufälligkeit in Residualdiagrammen handhaben. Je nachdem, welche Daten verfügbar sind, könnte das Modell durch Variablentransformation, Abstimmung der Hyperparameter des Lernalgorithmus, Wahl eines einfacheren oder eines komplexeren Verfahrens, Entfernen von Ausreißern oder Hinzunahme weiterer Variablen verbessert werden.

### Tipp

In Kapitel 3 haben wir den Kernel-Trick erörtert, der in Kombination mit *Support Vector Machines (SVMs)* zur Klassifizierung genutzt werden kann, was sich als nützlich erweist, wenn wir es mit nichtlinearen Aufgabenstellungen zu tun haben. Eine ausführliche Betrachtung würde über den Rahmen dieses Buches hinausgehen, aber SVMs können auch für nichtlineare Regressionen verwendet werden. Interessierten Lesern sei für weitere Informationen der ausgezeichnete Artikel von S.R. Gunn (S.R. Gunn et al., *Support Vector Machines for Classification and Regression*, ISIS technical report, 14, 1998) zur Lektüre empfohlen. In scikit-learn ist ebenfalls ein SVM-Regressor implementiert. Weitere Informationen zu dessen Verwendung finden Sie unter <http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html#sklearn.svm.SVR>.

## 10.9 Zusammenfassung

Am Anfang dieses Kapitels haben Sie erfahren, wie sich die Beziehung zwischen einer einzelnen erklärenden Variablen und einer stetigen Zielvariablen durch eine einfache lineare Regressionsanalyse modellieren lässt. Danach haben wir ein nützliches Verfahren zur Datenanalyse erörtert, um Muster oder Anomalien in den Daten aufzuspüren – ein wichtiger erster Schritt bei der Entwicklung von Vorhersagemodellen.

Für die Entwicklung des ersten Modells wurde zunächst anhand eines auf dem Gradientenabstiegsverfahren beruhenden Optimierungsansatzes eine lineare Regression implementiert. Dann haben Sie erfahren, wie sich die linearen scikit-learn-Modelle für Regressionen einsetzen lassen. Darüber hinaus wurde als Ansatz zur Handhabung von Ausreißern ein robustes Regressionsverfahren (RANSAC) implementiert. Zwecks Beurteilung der Vorhersagekraft von Regressionsmodellen haben wir die mittlere quadratische Abweichung und das damit

verwandte Bestimmtheitsmaß  $R^2$  berechnet. Außerdem haben wir einen Blick auf die Residualdiagramme geworfen, einen praktischen grafischen Ansatz zur Diagnose der Schwierigkeiten, die Regressionsmodelle bereiten.

Nach der Erörterung, wie die Regularisierung auf Regressionsmodelle angewendet werden kann, um deren Komplexität zu verringern und eine Überanpassung zu verhindern, haben wir zudem verschiedene Ansätze zur Modellierung nichtlinearer Zusammenhänge betrachtet, unter anderem die polynomiale Merkmalstransformation und Random-Forest-Regressoren.

Wir haben uns in den vorangegangenen Kapiteln ausführlich mit den Themen überwachtes Lernen, Klassifizierung und Regressionsanalyse befasst. Im nächsten Kapitel werden wir uns nun mit einer weiteren Unterkategorie des Machine Learnings befassen, dem unüberwachten Lernen. In diesem Zusammenhang werden Sie auch erfahren, wie sich die Clusteranalyse einsetzen lässt, um auch ohne Zielvariablen verborgene Muster in den Daten zu erkennen.

# Verwendung nicht gekennzeichneter Daten: Clusteranalyse

In den vorangegangenen Kapiteln haben wir Verfahren des überwachten Lernens angewendet, um Lernmodelle zu entwickeln, bei denen die Antworten bereits bekannt waren – die Klassenbezeichnungen waren in den Trainingsdaten schon verfügbar. In diesem Kapitel gehen wir einen Schritt weiter und erkunden die *Clusteranalyse*, ein Verfahren zum *unüberwachten Lernen*, das es ermöglicht, verborgene Strukturen in Daten aufzuspüren, auch wenn uns die richtigen Antworten nicht bereits vorab bekannt sind. Das Ziel des Clusterings ist es, neue natürliche Gruppen zu finden, sodass die Objekte im selben Cluster eine größere Ähnlichkeit miteinander aufweisen als mit den Objekten in anderen Clustern.

In Anbetracht seiner explorativen Natur ist das Clustering ein spannendes Thema. In diesem Kapitel werden Sie die folgenden Konzepte kennenlernen, die Ihnen dabei helfen sollen, Ihre Daten sinnvoll zu strukturieren:

- Das Aufspüren der Zentren von Clustern ähnlicher Objekte mithilfe des verbreiteten k-Means-Algorithmus
- Einsatz einer Bottom-up-Methode zum Aufbau hierarchischer Clusterbäume
- Erkennung beliebig gestalteter Objektstrukturen mittels dichtebasiertem Clustering

## 11.1 Gruppierung von Objekten nach Ähnlichkeit mit dem k-Means-Algorithmus

In diesem Abschnitt werden wir eine der in Wissenschaft und Industrie am häufigsten verwendeten Methoden zur Gruppierung von Objekten betrachten, den *k-Means-Algorithmus*. Das Clustering, oder genauer gesagt die Clusteranalyse, ist ein Verfahren, das es uns ermöglicht, Gruppen ähnlicher Objekte zu bilden, also Objekte, die einander ähnlicher sind als den zu anderen Gruppen zugehörigen Objekten. Beispiele für geschäftlich orientierte Anwendungen des Clusterings sind unter anderem die Gruppierung von Dokumenten, Musikstücken und Filmen nach verschiedenen Themen bzw. Inhalten oder auch das Auffinden von Kundengruppen mit ähnlichen Interessen anhand des gemeinsamen Kaufverhaltens als Grundlage für Empfehlungssysteme.

### 11.1.1 K-Means-Clustering mit scikit-learn

Wie Sie in Kürze sehen werden, ist der k-Means-Algorithmus nicht nur besonders einfach zu implementieren, er ist im Vergleich mit anderen Clustering-Algorithmen auch äußerst effizient – was wohl seine große Beliebtheit erklären dürfte. Der k-Means-Algorithmus gehört zur Kategorie der *partitionierenden prototypbasierten* Clustering-Verfahren. Die beiden anderen Kategorien, *hierarchisches* und *dichtebasiertes* Clustering, kommen später in diesem Kapitel noch zur Sprache.

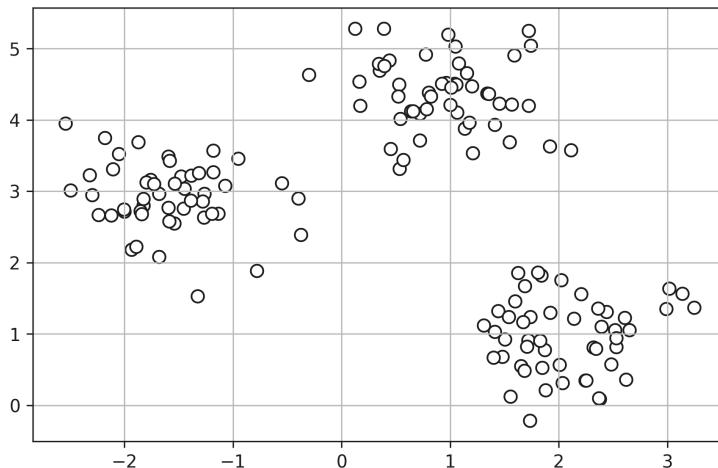
Prototypbasiertes Clustering bedeutet, dass ein Cluster durch einen Prototyp repräsentiert wird, bei dem es sich entweder um den *Zentroiden* (den Durchschnitt) ähnlicher Punkte mit stetigen Merkmalen oder aber um den *Medoiden* (den repräsentativsten oder am häufigsten vorkommenden Punkt) handelt, sofern wir es mit kategorialen Merkmalen zu tun haben. Der k-Means-Algorithmus ist zwar sehr gut zur Identifizierung sphärischer Cluster geeignet, hat aber den Nachteil, dass wir die Anzahl der Cluster  $k$  vorab festlegen müssen. Ein ungeeigneter Wert von  $k$  kann die Performance des Clusterings herabsetzen. In Kürze werden wir das sogenannte *Ellenbogenkriterium* sowie *Silhouettendiagramme* betrachten, die sich bei der Beurteilung der Güte eines Clusterings als nützlich erweisen und helfen, die optimale Anzahl der Cluster  $k$  zu ermitteln.

Der k-Means-Algorithmus ist zwar auch auf höherdimensionale Daten anwendbar, wir verwenden im folgenden Beispiel jedoch lediglich eine einfache zweidimensionale Datenmenge, um das Ergebnis visualisieren zu können.

```
>>> from sklearn.datasets import make_blobs
>>> X, y = make_blobs(n_samples=150,
...                     n_features=2,
...                     centers=3,
...                     cluster_std=0.5,
...                     shuffle=True,
...                     random_state=0)

>>> import matplotlib.pyplot as plt
>>> plt.scatter(X[:,0],
...               X[:,1],
...               c='white',
...               marker='o',
...               edgecolor='black'
...               s=50)
>>> plt.grid()
>>> plt.show()
```

Die soeben erstellte Datenmenge besteht aus 150 zufällig erzeugten Punkten, die im Wesentlichen in drei Bereichen höherer Dichte konzentriert sind, wie das zweidimensionale Streudiagramm in der folgenden Abbildung zeigt.



Bei der praktischen Anwendung der Clusteranalyse liegen keine Informationen über die möglichen Kategorien der Objekte vor – anderenfalls würde es sich um überwachtes Lernen handeln. Unser Ziel ist es also, die Objekte anhand ihrer ähnlichen Merkmale in Gruppen einzuteilen. Das können wir mit dem k-Means-Algorithmus und der Ausführung der folgenden vier Arbeitsschritte erreichen:

1. Wählen Sie aus den Objekten zufällig  $k$  Zentroide als anfängliche Cluster-Zentren aus.
2. Weisen Sie alle Objekte dem jeweils nächsten Zentroiden  $\mu^{(j)}$  ( $j \in \{1, \dots, k\}$ ) zu.
3. Berechnen Sie die Zentroide mit den in Schritt 2 zugewiesenen Objekten neu.
4. Wiederholen Sie die Schritte 2 und 3, bis sich die Zuordnungen nicht mehr ändern oder bis ein benutzerdefinierter Schwellenwert bzw. eine maximale Anzahl von Iterationen erreicht ist.

Nun stellt sich natürlich die Frage, wie die »Ähnlichkeit« von Objekten überhaupt beurteilt wird. Zunächst einmal können wir sie als das Gegenteil der Distanz definieren – und ein gebräuchliches Maß für die Distanz ist beim Clustering von Objekten mit stetigen Merkmalen die *quadratierte euklidische Distanz* zweier Punkte  $x$  und  $y$  im  $m$ -dimensionalen Raum:

$$d(x, y)^2 = \sum_{j=1}^m (x_j - y_j)^2 = \|x - y\|_2^2$$

Beachten Sie, dass der Index  $j$  in dieser Gleichung die  $j$ -te Dimension (Merkmalsspalte) der Punkte  $x$  und  $y$  bezeichnet. Im verbleibenden Teil dieses Abschnitts verwenden wir die hochgestellten Buchstaben  $i$  und  $j$ , um auf den Objektindex bzw. den Clusterindex zu verweisen.

Mit dem euklidischen Distanzmaß können wir den k-Means-Algorithmus als einfache Optimierungsaufgabe formulieren, also als iterativen Ansatz zur Minimie-

rung der *Summe der quadrierten Abweichungen innerhalb des Clusters (SSE, Sum of Squared Errors)*, die mitunter auch als Trägheit des Clusters bezeichnet wird:

$$SSE = \sum_{i=1}^n \sum_{j=1}^k w^{(i,j)} \left\| \mathbf{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2^2$$

Hier bezeichnet  $\boldsymbol{\mu}^{(j)}$  den Schwerpunkt (den Zentroiden) des Clusters  $j$ : Wenn  $\mathbf{x}^{(i)}$  zum Cluster  $j$  gehört, gilt  $w^{(i,j)} = 1$ , anderenfalls  $w^{(i,j)} = 0$ .

Nun wissen Sie, wie der k-Means-Algorithmus funktioniert und wir können ihn mithilfe der KMeans-Klasse von scikit-learns Cluster-Modul auf unsere Datenmenge anwenden:

```
>>> from sklearn.cluster import KMeans
>>> km = KMeans(n_clusters=3,
...                 init='random',
...                 n_init=10,
...                 max_iter=300,
...                 tol=1e-04,
...                 random_state=0)
>>> y_km = km.fit_predict(X)
```

Mit diesem Code setzen wir die Anzahl der Cluster auf 3. Dass die Anzahl der Cluster von vornherein festgelegt werden muss, gehört zu den Nachteilen des k-Means-Algorithmus. Wir weisen `n_init` den Wert 10 zu, um den k-Means-Algorithmus 10 Mal mit jeweils unterschiedlichen zufällig ausgewählten Zentroiden auszuführen, damit wir als endgültiges Modell dasjenige mit dem geringsten SSE-Wert auswählen können. Über den Parameter `max_iter` legen wir die maximale Anzahl der Iterationen (hier 300) bei den einzelnen Durchgängen fest. Beachten Sie, dass die Implementierung des k-Means-Algorithmus von scikit-learn abbricht, wenn der Algorithmus konvergiert, bevor die maximale Anzahl der Iterationen erreicht ist. Es ist allerdings ebenfalls möglich, dass der k-Means-Algorithmus bei einem der Durchgänge keine Konvergenz erzielt – was problematisch sein kann (hoher Rechenaufwand), wenn wir relativ große Werte für `max_iter` wählen. Eine Möglichkeit, solcher Konvergenzprobleme Herr zu werden, ist die Verwendung größerer Werte für `tol`, einem Parameter, der die Fehlertoleranz in Bezug auf die Änderungen der Summe der quadrierten Abweichungen innerhalb des Clusters steuert, um ein Konvergieren zu erkennen. Im vorliegenden Fall haben wir eine Fehlergrenze von `1e-04` (=0,0001) gewählt.

Eine weitere Schwierigkeit, die der k-Means-Algorithmus mit sich bringt, besteht darin, dass einer oder mehrere der Cluster leer sein können. Beachten Sie, dass dies beim k-Medoids- oder beim Fuzzy-C-Means-Algorithmus, auf den wir im nächsten Abschnitt noch eingehen werden, nicht der Fall ist. Diesem Problem wird in der aktuellen k-Means-Implementierung von scikit-learn jedoch Rechnung

getragen: Wenn ein Cluster leer ist, sucht der Algorithmus nach dem Objekt, das vom Zentroiden des leeren Clusters am weitesten entfernt ist. Dieser Punkt wird dann zum neuen Zentroiden des Clusters.

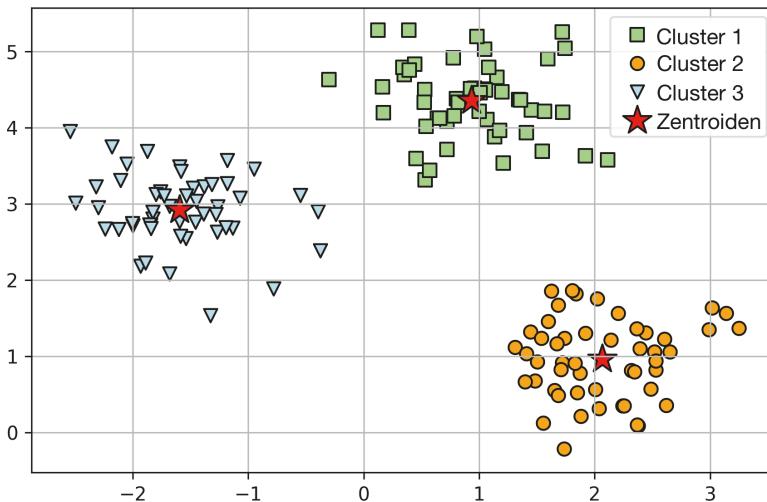
### Tipp

Wenn der k-Means-Algorithmus auf echte Daten angewendet wird und dabei ein euklidisches Distanzmaß zum Einsatz kommt, sollte gewährleistet sein, dass die Merkmale mit demselben Maßstab gemessen werden. Falls erforderlich, wird dann eine z-Wert-Standardisierung oder eine Min-Max-Skalierung durchgeführt.

Wir haben die Clusterbezeichnungen `y_km` vorhergesagt und die beim k-Means-Algorithmus auftretenden Schwierigkeiten erörtert. Nun sollen die vom k-Means-Algorithmus in der Datenmenge entdeckten Cluster sowie deren Zentroiden visualisiert werden. Letztere sind im `cluster_centers_-`-Attribut des `KMeans`-Objekts gespeichert:

```
>>> plt.scatter(X[y_km==0,0],
...                 X[y_km==0,1],
...                 s=50,
...                 c='lightgreen',
...                 marker='s', edgecolor='black',
...                 label='Cluster 1')
>>> plt.scatter(X[y_km==1,0],
...                 X[y_km==1,1],
...                 s=50,
...                 c='orange',
...                 marker='o', edgecolor='black',
...                 label='Cluster 2')
>>> plt.scatter(X[y_km==2,0],
...                 X[y_km==2,1],
...                 s=50,
...                 c='lightblue',
...                 marker='v', edgecolor='black',
...                 label='Cluster 3')
>>> plt.scatter(km.cluster_centers_[:,0],
...                 km.cluster_centers_[:,1],
...                 s=250,
...                 marker='*',
...                 c='red', edgecolor='black',
...                 label='Zentroiden')
>>> plt.legend(scatterpoints=1)
>>> plt.grid()
>>> plt.show()
```

Das folgende Streudiagramm zeigt, dass der k-Means-Algorithmus die drei Zentroiden im Zentrum der jeweiligen Sphäre platziert – eine bei dieser Datenmenge durchaus sinnvolle Gruppierung.



Bei dieser Beispieldatenmenge hat der k-Means-Algorithmus zwar gut funktioniert, wir sollten aber die Schwierigkeiten, die der Algorithmus mit sich bringt, nicht außer Acht lassen. Nachteilig ist, dass wir die Anzahl der Cluster  $k$  von vornherein festlegen müssen, was bei praktischen Anwendungen manchmal gar nicht so einfach ist – insbesondere dann nicht, wenn wir es mit höherdimensionalen Datenmengen zu tun haben, die nicht visualisiert werden können. Zu den weiteren Eigenschaften gehört auch, dass die Cluster einander nicht überlappen und nicht hierarchisch sind. Außerdem müssen wir annehmen, dass jeder Cluster mindestens ein Objekt enthält. Später in diesem Kapitel werden wir anderen Arten von Clustering-Algorithmen begegnen, nämlich hierarchischem und dichtebasiertem Clustering. Bei diesen Algorithmen ist es nicht erforderlich, die Anzahl der Cluster im Vorhinein festzulegen oder sphärische Strukturen in der Datenmenge vorauszusetzen.

In den nächsten Abschnitten werden wir eine verbreitete Variante des klassischen k-Means-Algorithmus namens k-Means++ vorstellen. Er setzt zwar dieselben Annahmen voraus und hat auch dieselben Nachteile wie der im letzten Abschnitt erläuterte k-Means-Algorithmus, kann die Clustering-Ergebnisse jedoch durch eine clevere Auswahl der anfänglichen Cluster-Zentren deutlich verbessern.

### 11.1.2 Der k-Means++-Algorithmus

Bislang haben wir den klassischen k-Means-Algorithmus betrachtet, der eine zufällige Auswahl der anfänglichen Zentroiden trifft. Ist diese Auswahl ungünstig, kann es zu einer mangelhaften Gruppenzuordnung oder langsamem Konvergi-

ren kommen. Dieses Problem kann man dadurch in den Griff bekommen, dass man den k-Means-Algorithmus mehrmals auf eine Datenmenge anwendet und dann das Modell auswählt, dessen Summe der quadrierten Abweichungen innerhalb des Clusters am geringsten ist. Die anfänglichen Zentroiden mit dem *k-Means++-Algorithmus* weit voneinander entfernt zu platzieren, ist eine weitere Strategie, die zu besseren und einheitlicheren Ergebnissen führt als der klassische k-Means-Algorithmus (D. Arthur und S. Vassilvitskii, *k-means++: The Advantages of Careful Seeding*, Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms, Seiten 1027-1035, Society for Industrial and Applied Mathematics, 2007).

Die Initialisierung des k-Means++-Algorithmus kann wie folgt zusammengefasst werden:

1. Initialisieren Sie eine leere Menge  $\mathbf{M}$  zum Speichern der  $k$  auszuwählenden Zentroiden.
2. Wählen Sie aus den Eingabeobjekten zufällig den ersten Zentroiden  $\mu^{(j)}$  aus und fügen Sie ihn der Menge  $\mathbf{M}$  hinzu.
3. Ermitteln Sie für alle nicht zu  $\mathbf{M}$  gehörenden Objekte  $x^{(i)}$  die minimale quadrierte Distanz  $d(x^{(i)}, \mathbf{M})^2$  zu den Zentroiden in der Menge  $\mathbf{M}$ .
4. Verwenden Sie die gewichtete Wahrscheinlichkeitsverteilung  $\frac{d(\mu^{(p)}, \mathbf{M})^2}{\sum_i d(x^{(i)}, \mathbf{M})^2}$ , um den nächsten Zentroiden  $\mu^{(p)}$  zufällig auszuwählen.
5. Wiederholen Sie die Schritte 2 und 3, bis  $k$  Zentroiden ausgewählt sind.
6. Fahren Sie mit dem klassischen k-Means-Algorithmus fort.

Um den k-Means++-Algorithmus mit dem scikit-learns KMeans-Objekt zu verwenden, muss lediglich der `init`-Parameter statt auf `random` auf `k-means++` (die Standardeinstellung) gesetzt werden, was in der Praxis auch dringend zu empfehlen ist. Wir haben ihn im letzten Beispiel nur deshalb nicht verwendet, um nicht zu viele Konzepte auf einmal einzuführen. Im verbleibenden Teil dieses Abschnitts kommt der k-Means++-Algorithmus zum Einsatz, aber experimentieren Sie ruhig ein wenig mit den beiden Ansätzen (dem klassischen k-Means mit `init='random'` und k-Means++ mit `init='k-Means++'`) zum Platzieren der Cluster-Zentroiden herum.

### 11.1.3 »Harte« und »weiche« Clustering-Algorithmen

»Harte« Clustering-Algorithmen, wie z.B. der im vorangegangenen Abschnitt erörterte k-Means-Algorithmus, ordnen jedes Objekt einer Datenmenge genau einem Cluster zu. Im Gegensatz dazu weisen »weiche« Clustering-Algorithmen (die manchmal auch als *Fuzzy-Clustering-Algorithmen* bezeichnet werden) Objekte einem oder mehreren Cluster(n) zu. Ein verbreitetes Beispiel für einen weichen Clustering-Algorithmus ist der *Fuzzy-C-Mean-Algorithmus* (*FCM-Algorithmus*, gelegentlich auch *Fuzzy-k-Means-Algorithmus* genannt). Die ursprüngliche Idee dazu entstand

schon in den 1970er-Jahren, als Joseph C. Dunn erstmals eine frühe Version des Fuzzy-Clusterings vorschlug, um den k-Means-Algorithmus zu verbessern (J.C. Dunn, *A Fuzzy Relative of the Isodata Process and its Use in Detecting Compact Well-separated Clusters*, 1973). Fast ein Jahrzehnt später veröffentlichte James C. Bezdek eine Arbeit mit Verbesserungen des Fuzzy-Clustering-Algorithmus, der heute als *FCM-Algorithmus* bezeichnet wird (J.C. Bezdek, *Pattern Recognition with Fuzzy Objective Function Algorithms*, Springer Science+Business Media, 2013).

Das FCM-Verfahren hat große Ähnlichkeit mit dem k-Means-Algorithmus. Wir ersetzen die »harte« Clusterzuordnung allerdings durch Werte, die für jeden Punkt die Wahrscheinlichkeit angeben, dass er zu einem bestimmten Cluster gehört. Beim k-Means-Algorithmus könnten wir die Clusterzugehörigkeit eines Objekts  $x$  durch einen dünnbesetzten Vektor binärer Elemente ausdrücken:

$$\begin{bmatrix} \mu^{(1)} \rightarrow 0 \\ \mu^{(2)} \rightarrow 1 \\ \mu^{(3)} \rightarrow 0 \end{bmatrix}$$

Hier bezeichnet die Indexposition mit dem Wert 1 den Cluster-Zentroiden  $\mu^{(j)}$ , dem das Objekt zugeordnet ist (mit der Annahme, dass  $k = 3$ ,  $j \in \{1, 2, 3\}$ ). Beim FCM-Algorithmus würde der Zugehörigkeitsvektor hingegen folgendermaßen aussehen:

$$\begin{bmatrix} \mu^{(1)} \rightarrow 0.1 \\ \mu^{(2)} \rightarrow 0.85 \\ \mu^{(3)} \rightarrow 0.05 \end{bmatrix}$$

Die Werte liegen hier jeweils im Intervall  $[0, 1]$  und geben die Wahrscheinlichkeiten für die Zugehörigkeit zum entsprechenden Zentroiden an. Die Summe der Zugehörigkeitswahrscheinlichkeiten eines Objekts ist also 1. Ebenso wie den k-Means-Algorithmus können wir auch den FCM-Algorithmus in vier Schritten zusammenfassen:

1. Legen Sie die Anzahl der Zentroiden  $k$  fest und weisen Sie allen Punkten eine zufällige Clusterzugehörigkeit zu.
2. Ermitteln Sie die Cluster-Zentroiden  $\mu^{(j)}, j \in \{1, \dots, k\}$ .
3. Aktualisieren Sie die Clusterzugehörigkeiten aller Punkte.
4. Wiederholen Sie die Schritte 2 und 3, bis sich die Zugehörigkeitskoeffizienten nicht mehr ändern oder bis ein benutzerdefinierter Schwellenwert bzw. eine maximale Anzahl von Iterationen erreicht ist.

Die Zielfunktion des FCM-Algorithmus, die wir als  $J_m$  bezeichnen, hat große Ähnlichkeit mit der Summe der quadrierten Abweichungen innerhalb des Clusters, die beim k-Means-Algorithmus minimiert wird:

$$J_m = \sum_{i=1}^n \sum_{j=1}^k w^{m(i,j)} \left\| \mathbf{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2^2, \quad m \in [1, \infty)$$

Beachten Sie jedoch, dass der Zugehörigkeitsindikator  $w^{(i,j)}$  anders als beim k-Means-Algorithmus kein binärer Wert ( $w^{(i,j)} \in \{0, 1\}$ ) ist, sondern eine reellwertige Wahrscheinlichkeit der Clusterzugehörigkeit ( $w^{(i,j)} \in [0, 1]$ ). Sie werden vermutlich auch bemerkt haben, dass dem Term  $w^{(i,j)}$  ein zusätzlicher Exponent hinzugefügt wurde. Dieser Exponent  $m$ , eine Zahl größer oder gleich 1 (typischerweise ist  $m=2$ ), ist der sogenannte *Fuzziness-Koeffizient* (oder einfach *Fuzzifier*), der den Grad der Fuzziness, sprich der graduellen Klassenzugehörigkeit festlegt. Je größer der Wert von  $m$  ist, desto geringer ist der Zugehörigkeitsgrad  $w^{(i,j)}$ , was zu »unschärferen« Clustern führt. Der Zugehörigkeitsgrad selbst wird folgendermaßen berechnet:

$$w^{(i,j)} = \left[ \sum_{p=1}^k \left( \frac{\left\| \mathbf{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \mathbf{x}^{(i)} - \boldsymbol{\mu}^{(p)} \right\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}$$

Wenn wir beispielsweise wie im vorangegangenen Beispiel drei Cluster-Zentren auswählen, können wir die Wahrscheinlichkeit, dass das Objekt  $\mathbf{x}^{(i)}$  zum Cluster  $\boldsymbol{\mu}^{(j)}$  gehört, so berechnen:

$$w^{(i,j)} = \left[ \left( \frac{\left\| \mathbf{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \mathbf{x}^{(i)} - \boldsymbol{\mu}^{(1)} \right\|_2} \right)^{\frac{2}{m-1}} + \left( \frac{\left\| \mathbf{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \mathbf{x}^{(i)} - \boldsymbol{\mu}^{(2)} \right\|_2} \right)^{\frac{2}{m-1}} + \left( \frac{\left\| \mathbf{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \mathbf{x}^{(i)} - \boldsymbol{\mu}^{(3)} \right\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}$$

Das Zentrum  $\boldsymbol{\mu}^{(j)}$  des Clusters selbst ergibt sich aus dem Mittelwert aller zum Cluster zugehörigen Objekte, die mit der Wahrscheinlichkeit der Zugehörigkeit zum eigenen Cluster gewichtet werden:

$$\boldsymbol{\mu}^{(j)} = \frac{\sum_{i=1}^n w^{m(i,j)} \mathbf{x}^{(i)}}{\sum_{i=1}^n w^{m(i,j)}}$$

Schon ein Blick auf die Gleichung zur Berechnung des Zugehörigkeitsgrades genügt, um festzustellen, dass jede FCM-Iteration rechenaufwendiger ist als eine Iteration des k-Means-Algorithmus. Allerdings benötigt der FCM-Algorithmus typischerweise insgesamt weniger Iterationen, um zu konvergieren. Leider ist der FCM-Algorithmus derzeit nicht in scikit-learn implementiert. In der Praxis hat

sich jedoch herausgestellt, dass der k-Means- und der FCM-Algorithmus zu sehr ähnlichen Ergebnissen kommen, wie in einer Studie festgestellt wurde (S. Ghosh und S.K. Dubey, *Comparative Analysis of k-means and Fuzzy c-means Algorithms*, IJACSA, 4: 35-38, 2013).

#### 11.1.4 Die optimale Anzahl der Cluster mit dem Ellenbogenkriterium ermitteln

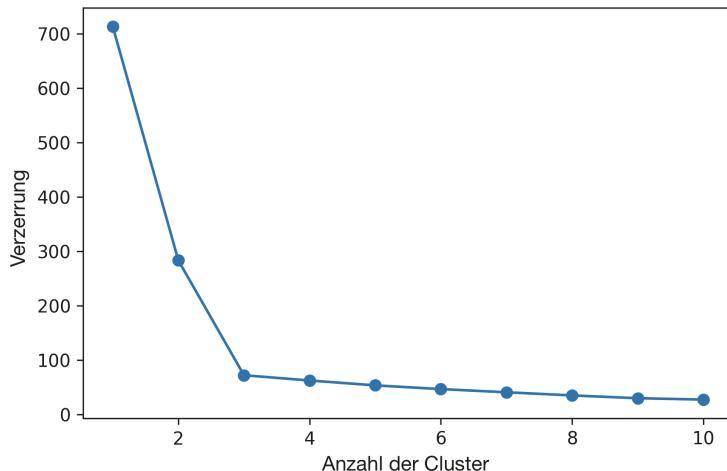
Beim unüberwachten Lernen stellt die Tatsache, dass wir keine eindeutige Antwort kennen, eine der größten Herausforderungen dar. In der Datenmenge sind keine Klassenbezeichnungen vorhanden, die es uns gestatten würden, die in Kapitel 6 zur Beurteilung der Leistung eines überwachten Modells verwendeten Verfahren einzusetzen. Um die Güte des Clusterings zu quantifizieren, müssen wir daher interne Bewertungskriterien einsetzen, wie beispielsweise die am Anfang dieses Kapitels erörterte Summe der quadrierten Abweichungen innerhalb des Clusters (die Verzerrung), um damit die Performance verschiedener k-Means-Clusterings zu vergleichen. Wir müssen die Summe der quadrierten Abweichungen innerhalb des Clusters noch nicht einmal selbst berechnen, da sie nach der Anpassung eines KMeans-Modells bereits als `inertia_-Attribut` zur Verfügung steht:

```
>>> print('Verzerrung: %.2f' % km.inertia_)
Verzerrung: 72.48
```

Wir können eine auf der Summe der quadrierten Abweichungen innerhalb des Clusters beruhende grafische Methode anwenden, um die optimale Anzahl der Cluster  $k$  für eine gegebene Aufgabenstellung zu ermitteln. Intuitiv würde man erwarten, dass die Verzerrung mit steigendem  $k$  sinkt, weil sich die Objekte näher an den Zentroiden befinden, denen sie zugeordnet sind. Dem *Ellenbogenkriterium* liegt die Idee zugrunde, denjenigen Wert für  $k$  zu finden, bei dem die Verzerrung am heftigsten anzusteigen beginnt. Durch die Darstellung der Verzerrung für verschiedene  $k$ -Werte als Diagramm wird dies sehr viel deutlicher werden:

```
>>> distortions = []
>>> for i in range(1, 11):
...     km = KMeans(n_clusters=i,
...                  init='k-means++',
...                  n_init=10,
...                  max_iter=300,
...                  random_state=0)
...     km.fit(X)
...     distortions.append(km.inertia_)
>>> plt.plot(range(1,11), distortions, marker='o')
>>> plt.xlabel('Anzahl der Cluster')
>>> plt.ylabel('Verzerrung')
>>> plt.show()
```

Wie Sie der Abbildung entnehmen können, ähnelt das Diagramm einem Ellenbogen, der sich bei  $k = 3$  befindet, was zeigt, dass  $k = 3$  für diese Datenmenge tatsächlich eine gute Wahl ist:



### 11.1.5 Quantifizierung der Clustering-Güte mit Silhouettendiagrammen

Die *Silhouettenanalyse* ist ein weiteres internes Bewertungskriterium der Clustering-Güte, das auch auf andere Algorithmen als den k-Means-Algorithmus anwendbar ist, auf die wir später in diesem Kapitel noch zu sprechen kommen. Die Silhouettenanalyse dient als grafische Methode zur Darstellung der Qualität der Zuordnungen der Objekte zu den Clustern. Für die Berechnung des *Silhouettenkoeffizienten* eines einzelnen Objekts der Datenmenge sind drei Schritte erforderlich:

1. Berechnen Sie die Geschlossenheit der Cluster  $a^{(i)}$  als den Mittelwert der Distanzen zwischen dem Objekt  $x^{(i)}$  und allen anderen Objekten im selben Cluster.
2. Berechnen Sie die Distanz  $b^{(i)}$  zum nächstgelegenen Cluster als mittlere Distanz zwischen dem Objekt  $x^{(i)}$  und allen anderen Objekten des nächstgelegenen Clusters.
3. Berechnen Sie die Silhouette  $s^{(i)}$ , indem Sie die Differenz von Distanz  $b^{(i)}$  und Geschlossenheit  $a^{(i)}$  durch den größeren der beiden Werte teilen:

$$s^{(i)} = \frac{b^{(i)} - a^{(i)}}{\max\{b^{(i)}, a^{(i)}\}}$$

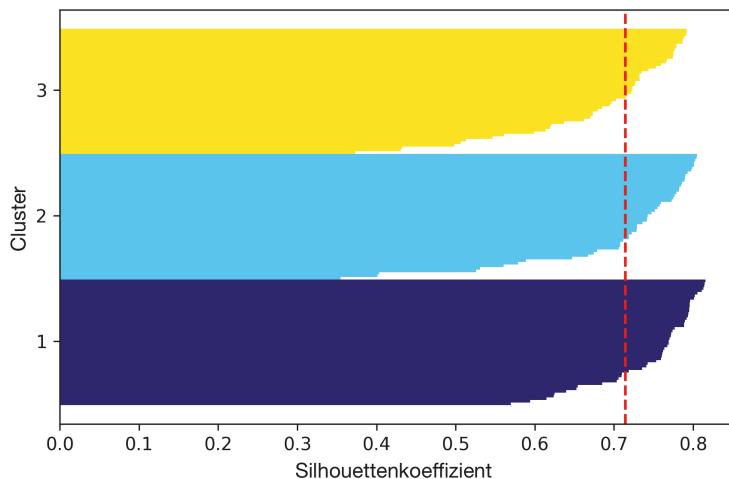
Der Silhouettenkoeffizient nimmt Werte zwischen -1 und +1 an. Gemäß der obigen Gleichung besitzt er den Wert 0, wenn Distanz und Geschlossenheit des Clusters gleich groß sind ( $b^{(i)} = a^{(i)}$ ). Außerdem kommen wir dem idealen Silhouettenkoeffizienten von 1 ziemlich nahe, sofern  $b^{(i)} >> a^{(i)}$  gilt, denn  $b^{(i)}$  quantifiziert, wie unähnlich ein Objekt den Objekten in anderen Clustern ist, und  $a^{(i)}$  ist ein Maß für die Ähnlichkeit zu den anderen Objekten im eigenen Cluster.

Der Silhouettenkoeffizient ist als Attribut `silhouette_samples` des scikit-learn-Moduls `metric` verfügbar. Bei Bedarf kann zusätzlich `silhouette_scores` importiert werden, womit der Mittelwert der Silhouettenkoeffizienten aller Objekte berechnet werden kann, was dem Ausdruck `numpy.mean(silhouette_samples(...))` entspricht. Mit dem nachstehenden Code erstellen wir ein Diagramm der Silhouettenkoeffizienten für ein k-Means-Clustering mit  $k = 3$ :

```
>>> km = KMeans(n_clusters=3,
...                 init='k-means++',
...                 n_init=10,
...                 max_iter=300,
...                 tol=1e-04,
...                 random_state=0)
>>> y_km = km.fit_predict(X)

>>> import numpy as np
>>> from matplotlib import cm
>>> from sklearn.metrics import silhouette_samples
>>> cluster_labels = np.unique(y_km)
>>> n_clusters = cluster_labels.shape[0]
>>> silhouette_vals = silhouette_samples(X,
...                                         y_km,
...                                         metric='euclidean')
>>> y_ax_lower, y_ax_upper = 0, 0
>>> yticks = []
>>> for i, c in enumerate(cluster_labels):
...     c_silhouette_vals = silhouette_vals[y_km == c]
...     c_silhouette_vals.sort()
...     y_ax_upper += len(c_silhouette_vals)
...     color = cm.jet(float(i) / n_clusters)
...     plt.barh(range(y_ax_lower, y_ax_upper),
...              c_silhouette_vals,
...              height=1.0,
...              edgecolor='none',
...              color=color)
...     yticks.append((y_ax_lower + y_ax_upper) / 2.)
...     y_ax_lower += len(c_silhouette_vals)
>>> silhouette_avg = np.mean(silhouette_vals)
>>> plt.axvline(silhouette_avg,
...               color="red",
...               linestyle="--")
>>> plt.yticks(yticks, cluster_labels + 1)
>>> plt.ylabel('Cluster')
>>> plt.xlabel('Silhouettenkoeffizient')
>>> plt.show()
```

Anhand des Silhouettendiagramms können wir mit einem Blick die Größe der verschiedenen Cluster überprüfen und erkennen, in welchen Clustern Ausreißer enthalten sind:



Aus dem Silhouettendiagramm geht hervor, dass die Silhouettenkoeffizienten nicht annähernd 0 sind, was ein Hinweis auf gelungenes Clustering sein kann. Außerdem ist der Mittelwert der Silhouettenkoeffizienten zwecks Zusammenfassung der Qualität des Clusterings durch eine gestrichelte Linie markiert.

Um in Erfahrung zu bringen, wie das Silhouettendiagramm eines vergleichsweise schlechten Clusterings aussieht, wenden wir den k-Means-Algorithmus nun mit nur zwei Zentroiden an:

```
>>> km = KMeans(n_clusters=2,
...                 init='k-means++',
...                 n_init=10,
...                 max_iter=300,
...                 tol=1e-04,
...                 random_state=0)
>>> y_km = km.fit_predict(X)

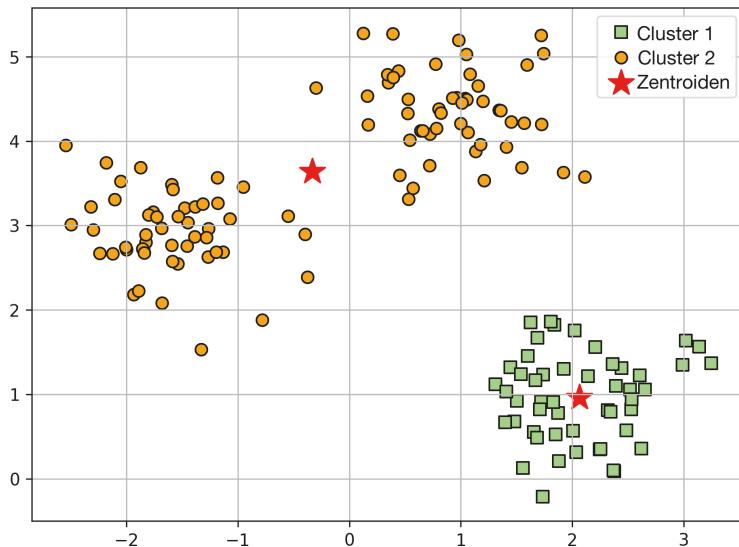
>>> plt.scatter(X[y_km==0,0],
...               X[y_km==0,1],
...               s=50, c='lightgreen',
...               edgecolor='black',
...               marker='s',
...               label='Cluster 1')
>>> plt.scatter(X[y_km==1,0],
...               X[y_km==1,1],
...               s=50,
...               c='orange',
```

```

...
    edgecolor='black',
...
    marker='o',
...
    label='Cluster 2')
>>> plt.scatter(km.cluster_centers_[:,0],
...                 km.cluster_centers_[:,1],
...                 s=250,
...                 marker='*',
...                 c='red',
...                 label='Zentroiden')
>>> plt.legend()
>>> plt.grid()
>>> plt.show()

```

Aus dem nachstehenden Streudiagramm ist ersichtlich, dass sich einer der Zentroiden mitten zwischen zwei der drei sphärischen Punktgruppierungen befindet. Das Clustering sieht zwar nicht vollkommen fürchterlich aus, es ist jedoch auch keineswegs optimal.



Beachten Sie hier bitte, dass wir uns den Luxus, Datenmengen als zweidimensionale Streudiagramme zu visualisieren, bei praktischen Anwendungen normalerweise nicht leisten können, da wir es in der Regel mit höherdimensionalen Daten zu tun haben. Als Nächstes erstellen wir ein Silhouettendiagramm, um das Ergebnis zu beurteilen:

```

>>> cluster_labels = np.unique(y_km)
>>> n_clusters = cluster_labels.shape[0]
>>> silhouette_vals = silhouette_samples(X,

```

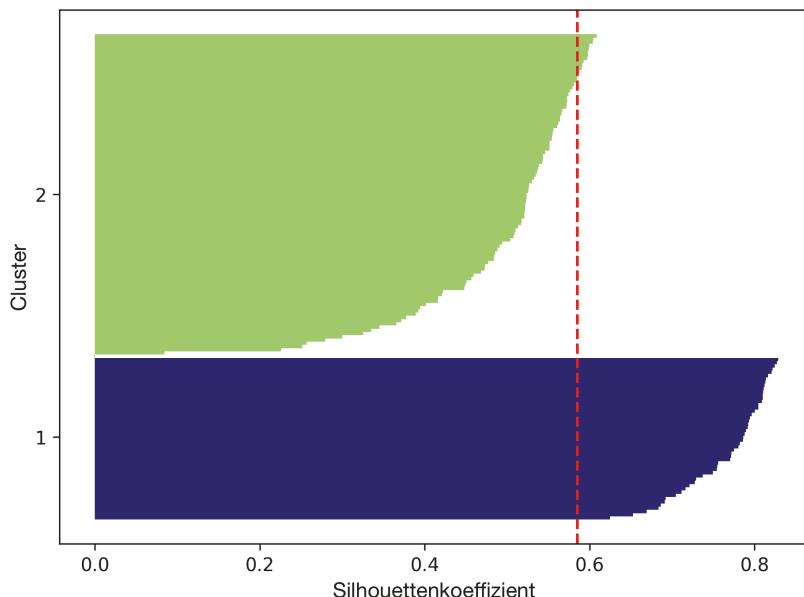
```

...
...                               y_km,
...                               metric='euclidean')

>>> y_ax_lower, y_ax_upper = 0, 0
yticks = []
>>> for i, c in enumerate(cluster_labels):
...     c_silhouette_vals = silhouette_vals[y_km == c]
...     c_silhouette_vals.sort()
...     y_ax_upper += len(c_silhouette_vals)
...     color = cm.jet(i / n_clusters)
...     plt.barh(range(y_ax_lower, y_ax_upper),
...              c_silhouette_vals,
...              height=1.0,
...              edgecolor='none',
...              color=color)
...     yticks.append((y_ax_lower + y_ax_upper) / 2)
...     y_ax_lower += len(c_silhouette_vals)
>>> silhouette_avg = np.mean(silhouette_vals)
>>> plt.axvline(silhouette_avg, color="red", linestyle="--")
>>> plt.yticks(yticks, cluster_labels + 1)
>>> plt.ylabel('Cluster')
>>> plt.xlabel('Silhouettenkoeffizient')
>>> plt.show()

```

Wie das Diagramm zeigt, sind die Silhouetten nun von deutlich unterschiedlicher Länge und Breite – das sind weitere Hinweise auf ein relativ schlechtes oder suboptimales Clustering.



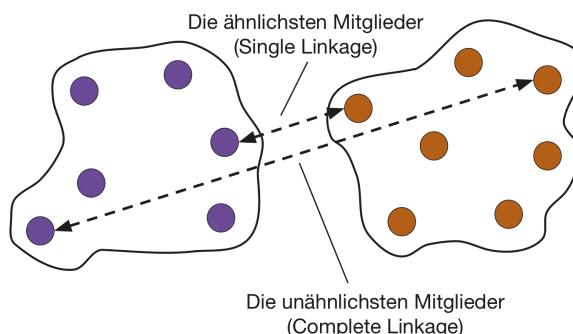
## 11.2 Cluster als hierarchischen Baum organisieren

In diesem Abschnitt werden wir einen alternativen Ansatz des prototypbasierten Clusterings betrachten: das *hierarchische Clustering*. Zu den Vorteilen hierarchischer Clustering-Algorithmen gehört, dass sie es uns ermöglichen, *Dendrogramme* (Visualisierungen binärer hierarchischer Clusterings) zu erstellen, die bei der Interpretation der Resultate hilfreich sind, indem sie sinnvolle Klassifizierungsschemata liefern. Ein weiterer Vorteil besteht darin, dass wir die Anzahl der Cluster nicht von vornherein festlegen müssen.

Bei der hierarchischen Clusteranalyse unterscheidet man zwei Methoden: *agglomeratives* und *divisives* hierarchisches Clustering. Beim divisiven hierarchischen Clustering gibt es zunächst nur einen einzigen Cluster, der alle Objekte umfasst. Dann wird dieser Cluster schrittweise in kleinere Cluster aufgeteilt, bis schließlich jeder Cluster nur noch ein Objekt enthält. In diesem Abschnitt konzentrieren wir uns auf das agglomerative Clustering, das den umgekehrten Ansatz verfolgt: Anfangs sind alle Objekte eigene Cluster und wir führen die nächstgelegenen Clusterpaare Schritt für Schritt zusammen, bis es nur noch einen einzigen Cluster gibt.

### 11.2.1 Gruppierung von Clustern

Die beiden Standardalgorithmen des agglomerativen hierarchischen Clusterings heißen *Single Linkage* (einzelne Kopplung) und *Complete Linkage* (vollständige Kopplung). Beim Single-Linkage-Verfahren berechnen wir die Distanzen der ähnlichsten Mitglieder für jedes Clusterpaar und führen die beiden Cluster zusammen, bei denen die Distanz der ähnlichsten Clustermitglieder am kleinsten ist. Das Complete-Linkage-Verfahren funktioniert auf ähnliche Weise, allerdings werden hier bei der Zusammenführung nicht die beiden ähnlichsten, sondern die beiden unähnlichsten Mitglieder miteinander verglichen (siehe Abbildung).



#### Tipp

Zu den weiteren gängigen agglomerativen hierarchischen Clustering-Algorithmen gehören *Average Linkage* und *Ward Linkage*. Beim Average-Linkage-Verfahren

werden Clusterpaare zusammengeführt, bei denen die mittlere Distanz aller Clustermitglieder minimal ist. Beim Ward-Verfahren werden die beiden Cluster zusammengeführt, die den geringsten Anstieg der Summe der quadrierten Abweichungen innerhalb des Clusters verursachen.

In diesem Abschnitt konzentrieren wir uns auf das agglomerative Clustering und wenden das Complete-Linkage-Verfahren an. Hierbei handelt es sich um ein iteratives Verfahren, das durch die folgenden Schritte zusammengefasst werden kann:

1. Berechnen Sie die Distanzmatrix aller Objekte.
2. Repräsentieren Sie alle Datenpunkte als einzelne Cluster.
3. Führen Sie die beiden gemäß der Distanz zwischen den unähnlichsten (entferntesten) Mitgliedern nächstgelegenen Cluster zusammen.
4. Aktualisieren Sie die Distanzmatrix.
5. Wiederholen Sie die Schritte 2 bis 4, bis nur noch ein Cluster verbleibt.

Nun werden wir uns damit befassen, wie man die in Schritt 1 erwähnte Distanzmatrix berechnet. Zunächst einmal erzeugen wir aber einige zufällige Beispieldaten, mit denen wir arbeiten können. Die Zeilen repräsentieren verschiedene Beobachtungen (IDs 0 bis 4) und die Spalten die verschiedenen Merkmale (X, Y, Z) dieser Daten.

```
>>> import pandas as pd
>>> import numpy as np
>>> np.random.seed(123)
>>> variables = ['X', 'Y', 'Z']
>>> labels = ['ID_0', 'ID_1', 'ID_2', 'ID_3', 'ID_4']
>>> X = np.random.random([5,3])*10
>>> df = pd.DataFrame(X, columns=variables, index=labels)
>>> df
```

Nach dem Ausführen des Codes sollte folgender `DataFrame` angezeigt werden, der die zufällig erzeugten Objekte enthält:

	<b>X</b>	<b>Y</b>	<b>Z</b>
<b>ID_0</b>	6.964692	2.861393	2.268515
<b>ID_1</b>	5.513148	7.194690	4.231065
<b>ID_2</b>	9.807642	6.848297	4.809319
<b>ID_3</b>	3.921175	3.431780	7.290497
<b>ID_4</b>	4.385722	0.596779	3.980443

### 11.2.2 Hierarchisches Clustering einer Distanzmatrix

Zur Berechnung der Distanzmatrix, die als Eingabe des hierarchischen Clustering-Algorithmus dienen soll, nutzen wir die `pdist`-Funktion des `spatial.distance`-Submoduls von SciPy:

```
>>> from scipy.spatial.distance import pdist, squareform
>>> row_dist = pd.DataFrame(squareform(
...     pdist(df, metric='euclidean')),
...     columns=labels, index=labels)
>>> row_dist
```

Der Code berechnet anhand der Merkmale X, Y und Z die euklidische Distanz zwischen allen Datenpunktpaarungen der Datenmenge. Wir stellen die von `pdist` zurückgelieferte Distanzmatrix (eine obere Dreiecksmatrix) der `squareform`-Funktion als Eingabe bereit, um eine symmetrische Matrix der paarweisen Distanzen zu erhalten:

	ID_0	ID_1	ID_2	ID_3	ID_4
ID_0	0.000000	4.973534	5.516653	5.899885	3.835396
ID_1	4.973534	0.000000	4.347073	5.104311	6.698233
ID_2	5.516653	4.347073	0.000000	7.244262	8.316594
ID_3	5.899885	5.104311	7.244262	0.000000	4.382864
ID_4	3.835396	6.698233	8.316594	4.382864	0.000000

Als Nächstes wenden wir eine Complete-Linkage-Agglomeration auf die Cluster an. Dazu nutzen wir die `linkage`-Funktion des `cluster.hierarchy`-Submoduls von SciPy, die eine sogenannte *Kopplungsmatrix* zurückliefert.

Bevor wir die `linkage`-Funktion aufrufen, sollten wir die Dokumentation sorgfältig durchlesen:

```
>>> from scipy.cluster.hierarchy import linkage
>>> help(linkage)
[...]
Parameters:
y : ndarray
A condensed or redundant distance matrix. A condensed
distance matrix is a flat array containing the upper
triangular of the distance matrix. This is the form
that pdist returns. Alternatively, a collection of m
observation vectors in n dimensions may be passed as
```

```

an m by n array.

method : str, optional
    The linkage algorithm to use. See the Linkage Methods
    section below for full descriptions.

metric : str, optional
    The distance metric to use. See the distance.pdist
    function for a list of valid distance metrics.

Returns:
Z : ndarray
    The hierarchical clustering encoded as a linkage matrix.
[...]

```

Dieser Beschreibung zufolge können wir die von der `pdist`-Funktion zurückgelieferte Distanzmatrix (eine obere Dreiecksmatrix) als Eingabe verwenden. Alternativ könnten wir auch das ursprüngliche Datenarray nutzen und beim Aufruf der `Linkage`-Funktion den Parameter `metric='euclidean'` setzen. Wir sollten allerdings nicht die vorhin definierte quadratische Distanzmatrix einsetzen, da diese andere Distanzwerte als erwartet liefern würde. Zusammengefasst gibt es hier drei Möglichkeiten:

- *Fehlerhafter Ansatz*: Bei diesem Ansatz verwenden wir die quadratische Distanzmatrix, was zu fehlerhaften Ergebnissen führt. Hier der Code:

```

>>> from scipy.cluster.hierarchy import linkage
>>> row_clusters = linkage(row_dist,
...                         method='complete',
...                         metric='euclidean')

```

- *Richtiger Ansatz*: Bei diesem Ansatz verwenden wir mit folgendem Code die Distanzmatrix in Form einer oberen Dreiecksmatrix:

```

>>> row_clusters = linkage(pdist(df, metric='euclidean'),
...                         method='complete')

```

- *Richtiger Ansatz*: Bei diesem Ansatz verwenden wir die vollständige Matrix der Eingabeobjekte, was wie beim vorhergehenden Ansatz die richtige Distanzmatrix ergibt:

```

>>> row_clusters = linkage(df.values,
...                         method='complete',
...                         metric='euclidean')

```

Um einen genaueren Blick auf die Clustering-Ergebnisse zu werfen, wandeln wir sie wie folgt in ein pandas-`DataFrame`-Objekt um (die Anzeige erfolgt am besten in einem Jupyter-Notebook):

```
>>> pd.DataFrame(row_clusters,
...                 columns=['Zeile 1',
...                            'Zeile 2',
...                            'Distanz',
...                            '# Objekte im Cluster'],
...                 index=['Cluster %d' %(i+1) for i in
...                         range(row_clusters.shape[0])])
```

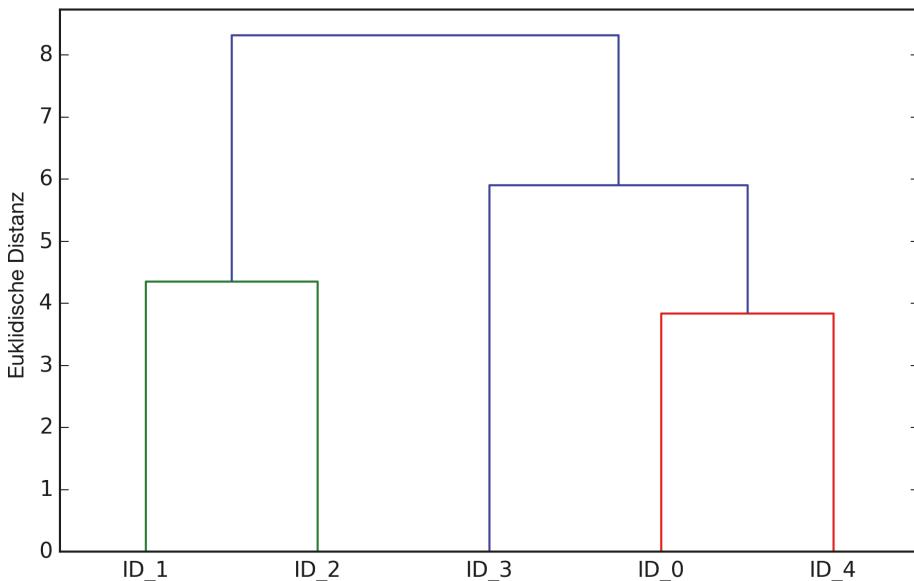
Die folgende Abbildung zeigt die Kopplungsmatrix, die aus mehreren Zeilen besteht, die jeweils eine Zusammenlegung repräsentieren. Die erste und die zweite Spalte bezeichnen die unähnlichsten Mitglieder der Cluster, und in der dritten Spalte ist die jeweilige Distanz angegeben. Die letzte Spalte enthält die Anzahl der Objekte des Clusters.

	<b>Zeile 1</b>	<b>Zeile 2</b>	<b>Distanz</b>	<b># Objekte im Cluster</b>
<b>Cluster 1</b>	0	4	3.835396	2
<b>Cluster 2</b>	1	2	4.347073	2
<b>Cluster 3</b>	3	5	5.899885	3
<b>Cluster 4</b>	6	7	8.316594	5

Nun haben wir die Kopplungsmatrix berechnet und können das Ergebnis in Form eines Dendrogramms visualisieren:

```
>>> from scipy.cluster.hierarchy import dendrogram
# Dendrogrammfarbe Schwarz (Teil 1/2)
# from scipy.cluster.hierarchy import set_link_color_palette
# set_link_color_palette(['black'])
>>> row_dendr = dendrogram(row_clusters,
...                           labels=labels,
...                           # Dendrogrammfarbe Schwarz (Teil 2/2)
...                           # color_threshold=np.inf
...                           )
>>> plt.tight_layout()
>>> plt.ylabel('Euclidean distance')
>>> plt.show()
```

Wenn Sie den obigen Code ausführen oder die E-Book-Version dieses Buches lesen, werden Sie feststellen, dass die Äste des Dendrogramms unterschiedliche Farben besitzen. Das Farbschema wird aus einer Liste von `matplotlib`-Farben abgeleitet, die zur Anzeige der Distanzschwellenwerte dienen. Wenn Sie das Dendrogramm beispielsweise in Schwarz darstellen möchten, brauchen Sie nur die Kommentarzeichen zu entfernen, die ich dem Code hinzugefügt habe.



Ein solches Dendrogramm stellt die beim agglomerativen hierarchischen Clustering gebildeten verschiedenen Cluster dar. So ist beispielsweise ersichtlich, dass die Objekte *ID\_0* und *ID\_4*, gefolgt von *ID\_1* und *ID\_2* einander gemäß dem Kriterium euklidische Distanz am ähnlichsten sind.

### 11.2.3 Dendrogramme und Heatmaps verknüpfen

In der Praxis werden hierarchische Clustering-Dendrogramme häufig in Kombination mit Heatmaps verwendet, die es ermöglichen, die verschiedenen Werte in der Stichprobenmatrix durch einen Farbcode zu repräsentieren. In diesem Abschnitt werden wir anhand eines Dendrogramms eine Heatmap erstellen und die Anordnung der darin enthaltenen Zeilen entsprechend arranjieren.

Das Verknüpfen eines Dendrogramms mit einer Heatmap kann allerdings etwas knifflig sein, gehen wir den Vorgang also schrittweise durch:

1. Wir erstellen ein `figure`-Objekt und definieren mit dem `add_axes`-Attribut die Position von x- und y-Achse sowie die Breite und die Höhe des Dendro-

gramms. Außerdem drehen wir das Dendrogramm um 90 Grad gegen den Uhrzeigersinn:

```
>>> fig = plt.figure(figsize=(8,8), facecolor='white')
>>> axd = fig.add_axes([0.09,0.1,0.2,0.6])
>>> row_dendr = dendrogram(row_clusters, orientation='left')
# Für matplotlib < v1.5.1 orientation='right' verwenden
```

2. Als Nächstes ordnen wir die Daten in dem ursprünglichen DataFrame entsprechend der Clustering-Bezeichnungen um, auf die über den Schlüssel `leaves` des Dendrogramm-Objekts zugegriffen werden kann, bei dem es sich im Wesentlichen um ein Python-Dictionary handelt:

```
>>> df_rowclust = df.iloc[row_dendr['leaves'][::-1]]
```

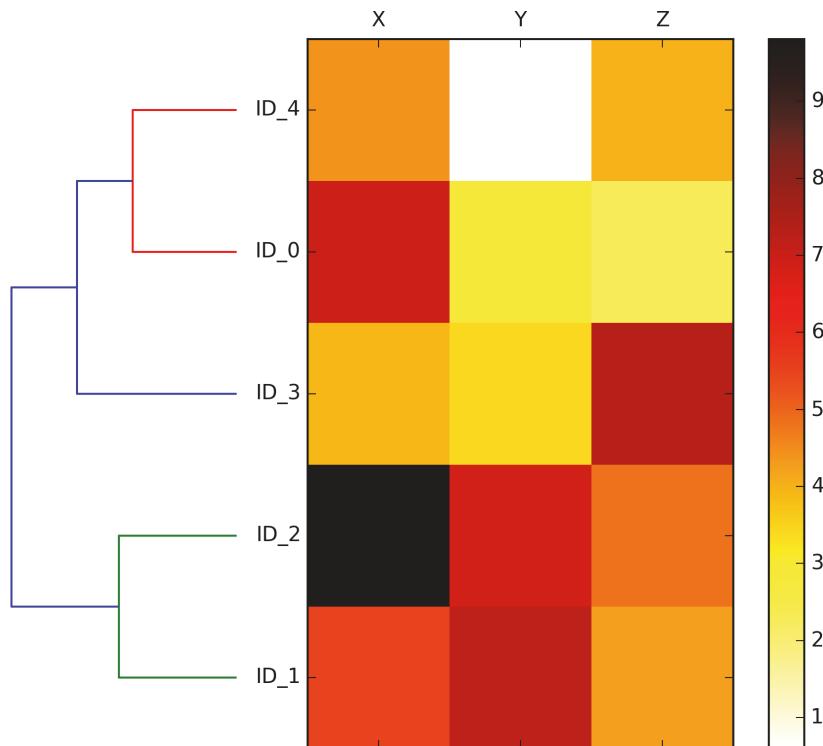
3. Nun erzeugen wir anhand des umgeordneten DataFrames eine Heatmap und platzieren sie unmittelbar neben dem Dendrogramm:

```
>>> axm = fig.add_axes([0.23,0.1,0.6,0.6])
>>> cax = axm.matshow(df_rowclust,
...                     interpolation='nearest', cmap='hot_r')
```

4. Anschließend modifizieren wir das Aussehen der Heatmap, indem wir die Markierungen auf den Achsen entfernen und die Anzeige der Achsen selbst unterdrücken. Darüber hinaus fügen wir eine Farbskala hinzu und weisen den Bezeichnungen der x- und y-Achsen die Namen der Merkmale und Objekte zu. Hier der Code:

```
>>> axd.set_xticks([])
>>> axd.set_yticks([])
>>> for i in axd.spines.values():
...     i.set_visible(False)
>>> fig.colorbar(cax)
>>> axm.set_xticklabels([''] + list(df_rowclust.columns))
>>> axm.set_yticklabels([''] + list(df_rowclust.index))
>>> plt.show()
```

Nach dem Ausführen dieser Schritte sollte die Heatmap zusammen mit dem Dendrogramm angezeigt werden.



Wie man sieht, spiegelt die Reihenfolge der Zeilen in der Heatmap das Clustering der Objekte im Dendrogramm wider. Zusammen mit dem einfachen Dendrogramm bieten die farbcodierten Objekte und Merkmale in der Heatmap eine praktische Übersicht über die Datenmenge.

#### 11.2.4 Agglomeratives Clustering mit scikit-learn

Im letzten Abschnitt haben Sie erfahren, wie man mit SciPy ein agglomeratives hierarchisches Clustering durchführt. Allerdings stellt auch scikit-learn eine `AgglomerativeClustering`-Implementierung zur Verfügung, die es uns ermöglicht, die Anzahl der Cluster anzugeben, die zurückgeliefert werden sollen. Das erweist sich als nützlich, wenn wir den hierarchischen Clusterbaum »zurechtstuzen« möchten. Wir setzen den Parameter `n_clusters` auf den Wert 3, um die Stichprobe mit dem Complete-Linkage-Verfahren und wie zuvor gemäß eines euklidischen Distanzmaßes in drei Gruppen aufzuteilen:

```
>>> from sklearn.cluster import AgglomerativeClustering
>>> ac = AgglomerativeClustering(n_clusters=3,
...                               affinity='euclidean',
...                               linkage='complete')
```

```
>>> labels = ac.fit_predict(X)
>>> print('Clusterbezeichnungen: %s' % labels)
Clusterbezeichnungen: [1 0 0 2 1]
```

Aus den Clusterbezeichnungen ist ersichtlich, dass das erste und das fünfte Objekt (ID\_0 und ID\_4) einem Cluster (Bezeichnung 1) und die Objekte ID\_1 und ID\_2 einem zweiten Cluster (Bezeichnung 0) zugeordnet wurden. Das Objekt ID\_3 wurde einem eigenen Cluster (Bezeichnung 2) zugeordnet, was mit den im Dendrogramm beobachtbaren Resultaten übereinstimmt. Wir sollten noch zur Kenntnis nehmen, dass das Objekt ID\_3 den Objekten ID\_0 und ID\_4 ähnlicher ist als den Objekten ID\_1 und ID\_2, wie dem Dendrogramm zu entnehmen ist. Aus scikit-learns Clustering-Ergebnis ist das nicht ersichtlich. Nun führen wir das AgglomerativeClustering erneut mit n\_clusters= 2 aus:

```
>>> ac = AgglomerativeClustering(n_clusters=2,
...                                 affinity='euclidean',
...                                 linkage='complete')
>>> labels = ac.fit_predict(X)
>>> print('Clusterbezeichnungen: %s' % labels)
Clusterbezeichnungen: [0 1 1 0 0]
```

Wie man sieht, wurde das Objekt ID\_3 in dieser zurechtgestutzten Clustering-Hierarchie wie erwartet demselben Cluster zugeordnet wie die Objekte ID\_0 und ID\_4.

### 11.3 Bereiche hoher Dichte mit DBSCAN ermitteln

Wir können in diesem Kapitel zwar nicht die große Vielfalt verschiedener Clustering-Algorithmen behandeln, aber ein weiterer Ansatz soll noch vorgestellt werden: *DBSCAN (Density-based Spatial Clustering of Applications with Noise)*, dichtebasierter räumliche Clusteranalyse mit Rauschen). Im Gegensatz zum k-Means-Algorithmus sind hier keine Annahmen bezüglich sphärischer Cluster erforderlich und es ist auch nicht notwendig, die Datenmenge anhand eines benutzerdefinierten Kriteriums hierarchisch aufzuteilen. Der Begriff *Dichte* ist hier als die Anzahl der Punkte innerhalb einer Umgebung mit dem Radius  $\varepsilon$  definiert.

Beim DBSCAN-Algorithmus wird jedem Objekt (Punkt) anhand folgender Kriterien eine spezielle Bezeichnung zugeordnet:

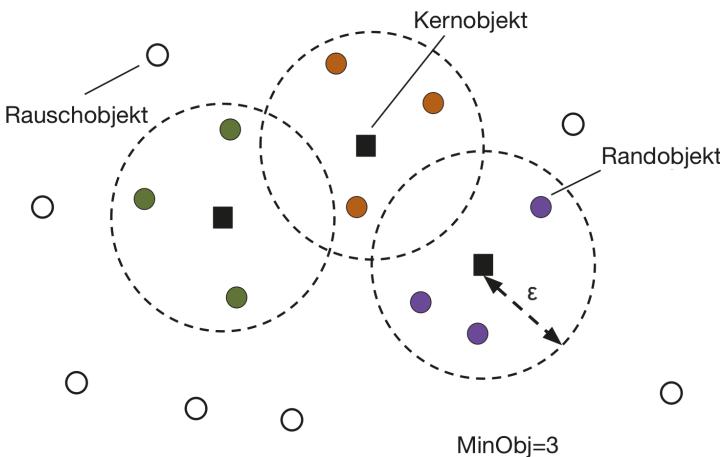
- Ein Punkt ist ein *Kernobjekt*, wenn sich innerhalb des Radius  $\varepsilon$  mindestens eine vorgegebene Anzahl (*MinObj*) weiterer Punkte befinden.
- Ein Punkt ist ein *Randobjekt*, wenn sich innerhalb des Radius  $\varepsilon$  weniger als *MinObj* Punkte befinden, der Punkt selbst sich jedoch innerhalb des Radius  $\varepsilon$  eines anderen Kernobjekts befindet.

- Alle übrigen Punkte, die weder Kernobjekte noch Randobjekte sind, werden als *Rauschobjekte* bezeichnet.

Wenn alle Punkte als Kern-, Rand- oder Rauschobjekte gekennzeichnet sind, besteht der DBSCAN-Algorithmus aus zwei Schritten:

- Bilden Sie für jedes Kernobjekt oder jede Gruppe verbundener Kernobjekte einen eigenen Cluster. (Kernobjekte sind verbunden, wenn sie nicht weiter als  $\epsilon$  voneinander entfernt sind.)
- Weisen Sie alle Randobjekte dem Cluster des zugehörigen Kernobjekts zu.

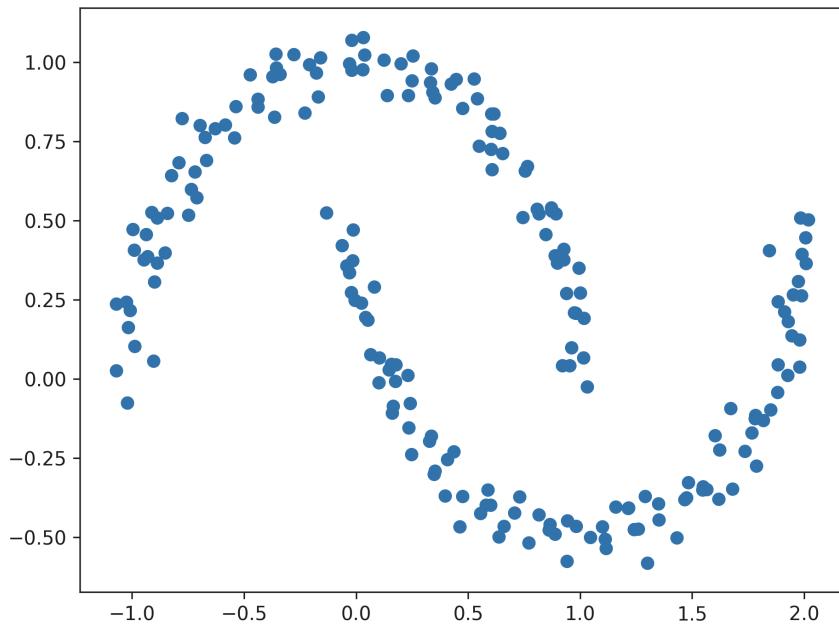
Bevor wir die Implementierung in Angriff nehmen, fassen wir das Konzept von Kern-, Rand- und Rauschobjekten in der folgenden Abbildung noch einmal zusammen, um so zu einem besseren Verständnis zu gelangen, wie das DBSCAN-Ergebnis aussehen könnte.



Einer der Hauptvorteile des DBSCAN-Algorithmus ist, dass anders als beim k-Means-Algorithmus nicht vorausgesetzt wird, dass die Cluster von sphärischer Form sind. Außerdem unterscheidet sich der DBSCAN-Algorithmus vom k-Means-Algorithmus und vom hierarchischen Clustering dadurch, dass er nicht unbedingt alle Objekte einem Cluster zuordnet, sondern in der Lage ist, Rauschobjekte zu entfernen. Betrachten wir ein etwas anschaulicheres Beispiel und erstellen eine neue Datenmenge mit halbmondförmigen Strukturen, um k-Means-Clustering, hierarchisches Clustering und DBSCAN miteinander zu vergleichen:

```
>>> from sklearn.datasets import make_moons
>>> X, y = make_moons(n_samples=200,
...                     noise=0.05,
...                     random_state=0)
>>> plt.scatter(X[:,0], X[:,1])
>>> plt.show()
```

Das Diagramm zeigt zwei aus jeweils 100 Objekten bestehende halbmondförmige Gruppen:

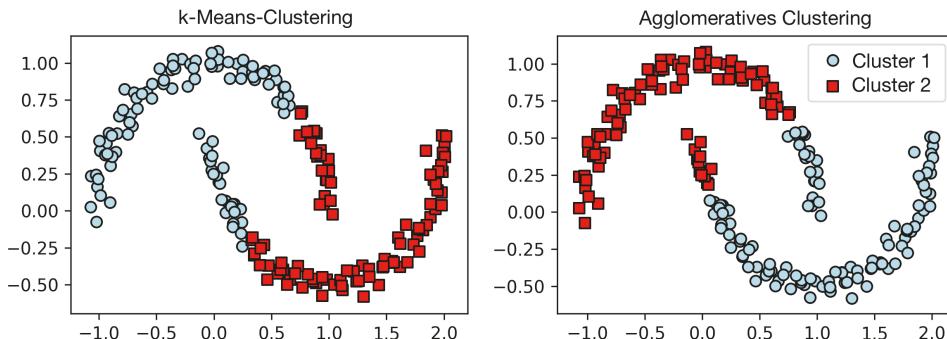


Wir verwenden zunächst den k-Means-Algorithmus und das Complete-Linkage-Verfahren, um zu überprüfen, ob einer dieser Clustering-Algorithmen in der Lage ist, die Halbmondformen als separate Cluster zu identifizieren. Hier der Code:

```
>>> f, (ax1, ax2) = plt.subplots(1, 2, figsize=(8,3))
>>> km = KMeans(n_clusters=2,
...                 random_state=0)
>>> y_km = km.fit_predict(X)
>>> ax1.scatter(X[y_km==0,0],
...               X[y_km==0,1],
...               c='lightblue',
...               edgecolor='black',
...               marker='o',
...               s=40,
...               label='Cluster 1')
>>> ax1.scatter(X[y_km==1,0],
...               X[y_km==1,1],
...               c='red',
...               edgecolor='black',
...               marker='s',
...               s=40,
...               label='Cluster 2')
```

```
>>> ax1.set_title('k-Means-Clustering')
>>> ac = AgglomerativeClustering(n_clusters=2,
...                                affinity='euclidean',
...                                linkage='complete')
>>> y_ac = ac.fit_predict(X)
>>> ax2.scatter(X[y_ac==0,0],
...               X[y_ac==0,1],
...               c='lightblue',
...               edgecolor='black',
...               marker='o',
...               s=40,
...               label='Cluster 1')
>>> ax2.scatter(X[y_ac==1,0],
...               X[y_ac==1,1],
...               c='red',
...               edgecolor='black',
...               marker='s',
...               s=40,
...               label='Cluster 2')
>>> ax2.set_title('Agglomeratives Clustering')
>>> plt.legend()
>>> plt.show()
```

Die Diagramme zeigen, dass der k-Means-Algorithmus nicht in der Lage ist, die beiden Cluster zu unterscheiden und dass auch der hierarchische Clustering-Algorithmus nicht mit diesen komplexen Formen zureckkommt.



Nun probieren wir den DBSCAN-Algorithmus mit dieser Datenmenge aus, um festzustellen, ob er die halbmondförmigen Cluster mit dem dichtebasierten Ansatz erkennt:

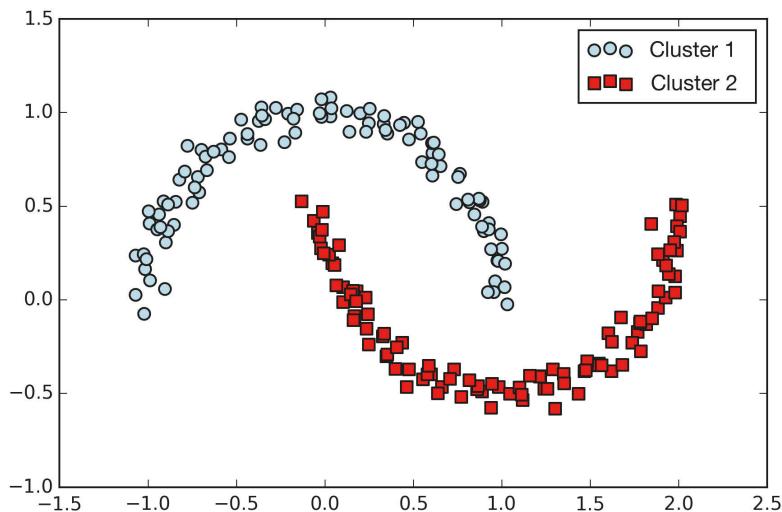
```
>>> from sklearn.cluster import DBSCAN
>>> db = DBSCAN(eps=0.2,
...              min_samples=5,
```

```

...                 metric='euclidean')
>>> y_db = db.fit_predict(X)
>>> plt.scatter(X[y_db==0,0],
...                  X[y_db==0,1],
...                  c='lightblue',
...                  edgecolor='black',
...                  marker='o',
...                  s=40,
...                  label='Cluster 1')
>>> plt.scatter(X[y_db==1,0],
...                  X[y_db==1,1],
...                  c='red',
...                  edgecolor='black',
...                  marker='s',
...                  s=40,
...                  label='Cluster 2')
>>> plt.legend()
>>> plt.show()

```

Tatsächlich kann der DBSCAN-Algorithmus die Halbmondformen erkennen. Hier zeigt sich eine der Stärken dieses Algorithmus: das Clustering von Daten beliebiger Formen.



An dieser Stelle müssen jedoch auch einige der Nachteile des DBSCAN-Algorithmus erwähnt werden. Bei einer Trainingsdatenmenge fester Größe macht sich bei wachsender Anzahl der Merkmale der Datenmenge der negative Effekt des *Fluchs der Dimensionalität* zunehmend bemerkbar. Dies ist insbesondere bei Verwendung eines euklidischen Distanzmaßes problematisch. Der Fluch der Dimensionalität

betrifft allerdings nicht nur den DBSCAN-Algorithmus, sondern auch andere Clustering-Algorithmen, die ein euklidisches Distanzmaß verwenden, beispielsweise den k-Means-Algorithmus und das hierarchische Clustering. Darüber hinaus müssen beim DBSCAN-Algorithmus zwei Hyperparameter (`MinObj` und  $\varepsilon$ ) optimiert werden, um brauchbare Ergebnisse zu erzielen. Eine geeignete Kombination von `MinObj` und  $\varepsilon$  zu finden, kann Schwierigkeiten bereiten, wenn die Dichteunterschiede in der Datenmenge relativ groß sind.

### Tipp

Sie haben nun die drei grundlegenden Kategorien von Clustering-Algorithmen kennengelernt: prototypbasiertes Clustering mit dem k-Means-Algorithmus, agglomeratives hierarchisches Clustering und dichtebasierter Clustering mit dem DBSCAN-Algorithmus. Ich möchte jedoch noch eine vierte Art fortgeschrittenen Clustering-Algorithmen erwähnen, die in diesem Kapitel noch nicht behandelt wurde: das *graphenbasierte Clustering*. Zu den bekanntesten Mitgliedern dieser Kategorie zählt das *spektrale Clustering*. Es gibt zwar eine Vielzahl verschiedener Implementierungen des spektralen Clusterings, die allerdings alle gemeinsam haben, dass sie die Eigenvektoren einer Ähnlichkeits- oder Distanzmatrix verwenden, um die Beziehungen der Cluster untereinander herzuleiten. Da eine genauere Betrachtung des spektralen Clusterings über den Rahmen dieses Buches hinausgeht, empfehle ich Ulrike von Luxburgs exzellente Einführung in dieses Thema (U. von Luxburg, *A Tutorial on Spectral Clustering*, Statistics and Computing, 17(4):395-416, 2007). Sie ist auf der arXiv-Website unter <http://arxiv.org/pdf/0711.0189v1.pdf> kostenlos verfügbar.

Beachten Sie, dass in der Praxis nicht immer auf der Hand liegt, welcher Algorithmus mit einer gegebenen Datenmenge am besten funktioniert – insbesondere dann nicht, wenn es sich um mehrdimensionale Daten handelt, die Visualisierungen schwierig oder gar unmöglich machen. Außerdem muss betont werden, dass ein erfolgreiches Clustering nicht nur vom Algorithmus und dessen Hyperparametern abhängt. Die Wahl eines geeigneten Distanzmaßes und Kenntnisse auf dem jeweiligen Fachgebiet, die hilfreich dabei sind, den Ablauf der Untersuchung zu strukturieren, können sogar noch wichtiger sein.

Aufgrund des Fluchs der Dimensionalität ist es daher gängige Praxis, vor dem Clustering Verfahren zur Dimensionsreduktion einzusetzen. Bei unüberwachten Verfahren kommen dafür die in Kapitel 5 erläuterten Methoden Hauptkomponentenanalyse (PCA) und RBF-Kernel-PCA infrage. Darüber hinaus ist es üblich, Datenmengen in zweidimensionale Unterräume zu transformieren, die es ermöglichen, Cluster und Klassenbezeichnungen durch zweidimensionale Streudiagramme zu visualisieren, die für die Beurteilung der Ergebnisse besonders nützlich sind.

## 11.4 Zusammenfassung

In diesem Kapitel haben Sie drei verschiedene Clustering-Algorithmen kennengelernt, die hilfreich sind, um verborgene Strukturen oder Informationen in den Daten aufzuspüren. Am Anfang des Kapitels haben wir mit dem k-Means-Algorithmus einen prototypbasierten Ansatz erörtert, der Objekte anhand einer vorgegebenen Anzahl von Zentroiden in sphärische Cluster gruppiert. Da es sich beim Clustering um eine unüberwachte Form des Machine Learnings handelt, steht uns der Luxus bekannter Klassenbezeichnungen bei der Beurteilung der Leistung eines Modells nicht zur Verfügung. Deshalb haben wir interne Bewertungskriterien betrachtet, wie z.B. das Ellenbogenkriterium oder die Silhouettenanalyse, um die Güte des Clusterings quantifizieren zu können.

Dann haben wir einen anderen Ansatz erörtert: das agglomerative hierarchische Clustering. Beim hierarchischen Clustering ist es nicht erforderlich, die Anzahl der Cluster im Vorhinein festzulegen, und das Ergebnis kann mit Dendrogrammen visualisiert werden, die bei der Interpretation der Ergebnisse hilfreich sind. Das letzte in diesem Kapitel betrachtete Clustering-Verfahren war der DBSCAN-Algorithmus, der die Objekte anhand der lokalen Dichte gruppiert und in der Lage ist, mit Ausreißern zurechtzukommen und nicht sphärische Formen zu erkennen.

Nach diesem Ausflug in das Fachgebiet des unüberwachten Machine Learnings ist es nun an der Zeit, den spannendsten Algorithmus des überwachten Machine Learnings vorzustellen: mehrschichtige künstliche neuronale Netze. Nach ihrem jüngsten Wiederaufleben gehören neuronale Netze wieder zu den aktuellsten Themen bei der Erforschung des Machine Learnings. Dank der erst kürzlich entwickelten Deep-Learning-Algorithmen gelten neuronale Netze bei vielen komplexen Aufgaben wie der Klassifizierung von Bildern und der Spracherkennung als aktueller Stand der Technik. Im nächsten Kapitel werden wir von Grund auf ein eigenes neuronales Netz entwickeln. Und in Kapitel 13, *Parallelisierung des Trainings neuronaler Netze mit TensorFlow*, werden Sie leistungsfähige Bibliotheken kennenlernen, die hilfreich sind, um komplexe Netzarchitekturen effizienter zu trainieren.

# Implementierung eines künstlichen neuronalen Netzes

Wie Ihnen vermutlich bekannt ist, findet das *Deep Learning* ein großes Medienecho und gehört zweifellos zu den aktuellsten Themen bei der Erforschung des Machine Learnings. Das Deep Learning kann als eine Algorithmenmenge aufgefasst werden, die dafür entwickelt wurde, künstliche *neuronale Netze* mit vielen Schichten effizienter zu trainieren. In diesem Kapitel lernen Sie die Grundlagen künstlicher neuronaler Netze kennen, damit Sie für die folgenden Kapitel gut gerüstet sind, um die auf Python basierenden fortgeschrittenen Deep-Learning-Bibliotheken und Deep-Neural-Network-Architekturen (DNN-Architekturen) zu erkunden, die insbesondere für die Analyse von Bildern und Texten geeignet sind.

Die Themen in diesem Kapitel sind:

- Konzeptuelles Verständnis mehrschichtiger neuronaler Netze
- Implementierung des grundlegenden Backpropagation-Verfahrens zum Trainieren neuronaler Netze
- Trainieren eines elementaren mehrschichtigen neuronalen Netzes für die Bildklassifizierung

## 12.1 Modellierung komplexer Funktionen mit künstlichen neuronalen Netzen

Am Anfang des Buches sind Ihnen in Kapitel 2, *Lernalgorithmen für die Klassifizierung trainieren*, künstliche Neuronen schon einmal begegnet. Sie sind die Bausteine mehrschichtiger künstlicher neuronaler Netze, die wir in diesem Kapitel eingehender betrachten werden. Das grundlegende Konzept neuronaler Netze beruht auf Hypothesen und Modellen der Funktionsweise des menschlichen Gehirns beim Lösen komplexer Aufgabenstellungen. Künstliche neuronale Netze erfreuen sich zwar seit einigen Jahren zunehmender Beliebtheit, die ersten Untersuchungen fanden aber schon in den 1940er-Jahren statt. Damals beschrieben Warren McCulloch und Walter Pitts erstmals die mögliche Funktionsweise von Neuronen.

Nach der ersten Implementierung des McCulloch-Pitts-Neuronenmodells durch Rosenblatts Perzeptron in den 1950ern verloren viele Forschungspioniere auf dem Gebiet des Machine Learnings in den folgenden Jahrzehnten allmählich das Inter-

resse an neuronalen Netzen, weil niemand eine gute Lösung für das Trainieren eines neuronalen Netzes mit mehreren Schichten vorweisen konnte. Erst 1986 lebte das Interesse an neuronalen Netzen wieder auf, als D.E. Rumelhart, G.E. Hinton und R.J. Williams das *Backpropagation*-Verfahren zum effizienteren Trainieren neuronaler Netze, das wir später in diesem Kapitel noch näher betrachten werden, (wieder-)entdeckten und ihm zu einer größeren Verbreitung verhalfen. (David E. Rumelhart, Geoffrey E. Hinton und Ronald J. Williams (1986), *Learning Representations by Back-propagating Errors*, Nature 323 (6088):533-536, 1986). An der Geschichte von künstlicher Intelligenz, Machine Learning und neuronalen Netzen interessierte Leser sollten den Wikipedia-Artikel über Phasen des geringen Interesses und mangelnder Förderung der Erforschung künstlicher Intelligenz lesen ([https://en.wikipedia.org/wiki/AI\\_winter](https://en.wikipedia.org/wiki/AI_winter)).

Im vergangenen Jahrzehnt gab es einige bahnbrechende Fortschritte, die zu den Konzepten führten, die wir heutzutage *Deep-Learning-Algorithmen* und Deep-Learning-Architekturen nennen – aus vielen Schichten bestehende neuronale Netze. Neuronale Netze sind nicht nur in der reinen Forschung von Interesse – auch große Technologieunternehmen wie Facebook, Microsoft und Google investieren kräftig in die Erforschung von künstlichen neuronalen Netzen und Deep-Learning-Verfahren. Auf Deep-Learning-Algorithmen beruhende komplexe neuronale Netze sind heutiger Stand der Technik, wenn es um komplexe Aufgabenstellungen wie Bild- oder Spracherkennung geht. Bekannte Beispiele für Produkte aus dem Alltag, die auf Deep Learning beruhen, sind Googles Bildersuche und Google Translate. Letzteres ist eine Smartphone-App, die in Echtzeit automatisch Text in Bildern erkennt und in mehr als 20 verschiedene Sprachen übersetzt.

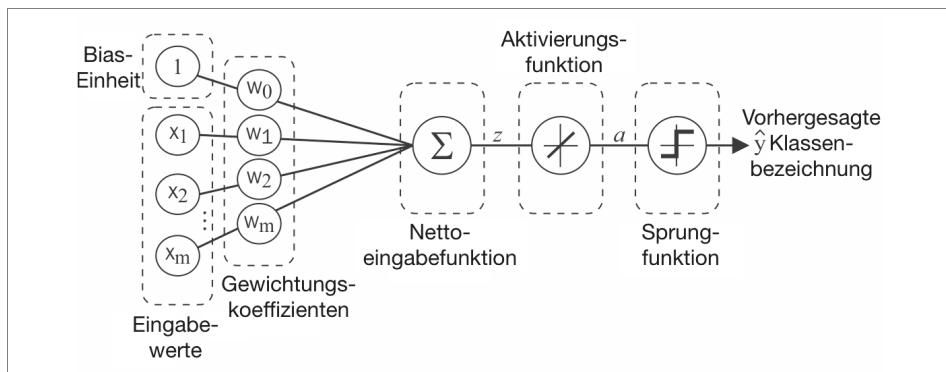
Die bedeutenden Technologieunternehmen und die Pharmaindustrie haben viele spannende, auf komplexen neuronalen Netzen basierende Anwendungen entwickelt, wie die folgende, natürlich unvollständige Liste von Beispielen zeigt:

- Facebooks Gesichtserkennung *DeepFace* (Y. Taigman, M. Yang, M. Ranzato und L. Wolf, *DeepFace: Closing the gap to human-level performance in face verification*, Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference, Seiten 1701-1708, 2014)
- Baidus *DeepSpeech*, das Anfragen in der Sprache Mandarin verarbeiten kann (A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates et al., *DeepSpeech: Scaling up end-to-end speech recognition*, arXiv preprint arXiv:1412.5567, 2014)
- Googles neuer Übersetzungsdiensst (*Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*, arXiv preprint arXiv:1412.5567, 2016)
- Neuartige Verfahren für die Entwicklung von Medikamenten sowie für die Vorhersage der Giftigkeit von Substanzen (T. Unterthiner, A. Mayr, G. Klambauer und S. Hochreiter, *Toxicity prediction using deep learning*, arXiv preprint arXiv:1503.01445, 2015)

- Eine mobile Anwendung, die in der Lage ist, Hautkrebs mit derselben Erfolgsquote wie ausgebildete Dermatologen zu erkennen (A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau und S. Thrun. *Dermatologist-level classification of skin cancer with deep neural networks*, in Nature 542, no. 7639, 2017, Seiten 115-118)

### 12.1.1 Einschichtige neuronale Netze

In diesem Kapitel dreht sich alles um mehrschichtige neuronale Netze, ihre Funktionsweise und die Frage, wie sie für das Lösen komplexer Aufgabenstellungen trainiert werden. Bevor wir uns eingehender mit bestimmten mehrschichtigen neuronalen Netzarchitekturen befassen, möchte ich kurz noch einmal einige der Konzepte des in Kapitel 2 vorgestellten einschichtigen neuronalen Netzes zusammenfassen – und damit meine ich den *Adaline-Algorithmus (ADAptive LInear NEuron)*, der in der folgenden Abbildung dargestellt ist.



In Kapitel 2 haben wir den Adaline-Algorithmus implementiert, um binäre Klassifizierungen vorzunehmen, und das Gradientenabstiegsverfahren als Optimierungsalgorithmus eingesetzt, um die Gewichtungskoeffizienten des Modells zu ermitteln. Bei jeder Epoche (jedem Durchlauf der Trainingsdatenmenge) haben wir den Gewichtungsvektor  $w$  anhand folgender Regel aktualisiert:

$$w := w + \Delta w, \text{ wobei } \Delta w = -\eta \nabla J(w)$$

Oder anders formuliert: Wir berechnen den Gradienten anhand der gesamten Trainingsdatenmenge und aktualisieren die Gewichtungen des Modells, indem wir uns einen Schritt in die entgegengesetzte Richtung des Gradienten  $\nabla J(w)$  bewegen. Um die optimalen Gewichtungen des Modells zu finden, minimieren wir eine Zielfunktion, die als Straffunktion  $J(w)$  (die *Summe der quadrierten Abweichung*) definiert ist. Außerdem multiplizieren wir den Gradienten mit der *Lernrate*  $\eta$ , einem sorgfältig gewählten Faktor, der einen Kompromiss zwischen Lerngeschwindigkeit und dem Hinausschießen über das globale Minimum der Straffunktion bieten soll.

Bei der Optimierung mit dem Gradientenabstiegsverfahren aktualisieren wir nach jeder Epoche alle Gewichtungen gleichzeitig. Die partiellen Ableitungen der Gewichtungen  $w_j$  im Gewichtungsvektor  $\mathbf{w}$  sind folgendermaßen definiert:

$$\frac{\partial}{\partial w_j} J(\mathbf{w}) = \sum_i (y^{(i)} - a^{(i)}) x_j^{(i)}$$

Hier repräsentiert  $y^{(i)}$  die Zielklassenbezeichnung des Objekts  $x^{(i)}$  und  $a^{(i)}$  die *Aktivierung* des Neurons, die im Spezialfall des Adaline-Algorithmus eine lineare Funktion ist. Wir definieren die *Aktivierungsfunktion*  $\phi(\cdot)$  wie folgt:

$$\phi(z) = z = a$$

In diesem Fall ist die Nettoeingabe  $z$  eine Linearkombination aus den die Eingabeschicht mit der Ausgabeschicht verknüpfenden Gewichtungen:

$$z = \sum_j w_j x_j = \mathbf{w}^T \mathbf{x}$$

Wir verwenden die Aktivierungsfunktion  $\phi(z)$  zur Berechnung der Aktualisierung des Gradienten und implementieren eine *Schwellenwertfunktion* (Heaviside-Funktion), um die stetige Ausgabe zum Zweck einer Vorhersage binären Klassenbezeichnungen zuordnen zu können:

$$\hat{y} = \begin{cases} 1, & \text{wenn } g(z) \geq 0 \\ -1 & \text{anderenfalls} \end{cases}$$

### Tipp

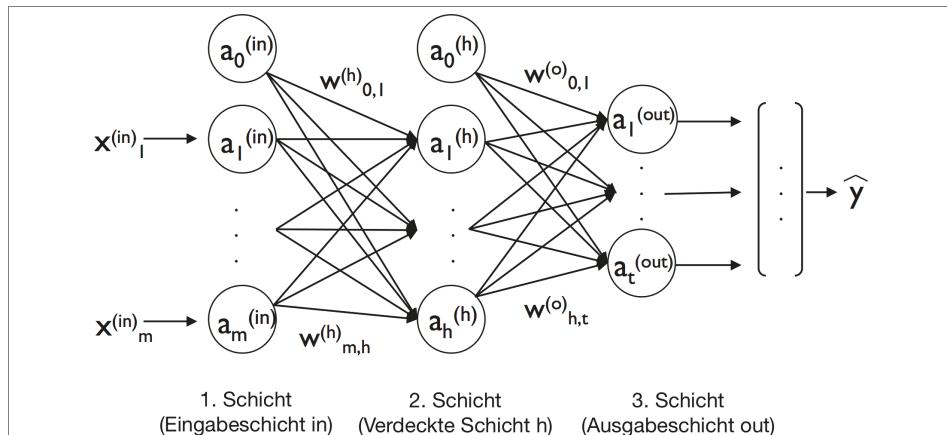
Beachten Sie, dass Adaline zwar aus zwei Schichten besteht, einer Eingabe- und einer Ausgabeschicht, aber dennoch als einschichtiges Netz bezeichnet wird, weil es nur eine Verbindung zwischen Ein- und Ausgabeschicht gibt.

Wir haben außerdem einen Trick zur Beschleunigung des Lernverfahrens verwendet, das sogenannte stochastische Gradientenabstiegsverfahren, das die Straffunktion anhand eines einzelnen Trainingsobjekts (Online-Lernen) oder einer kleinen Teilmenge der Trainingsobjekte (Lernen als Mini-Stapelverarbeitung) abschätzt. Wir werden später in diesem Kapitel bei der Implementierung und dem Trainieren eines mehrschichtigen Perzeptrons von diesem Konzept Gebrauch machen. Neben dem schnelleren Lernen durch die gegenüber dem Gradientenabstiegsverfahren häufigere Aktualisierung der Gewichtungen ist auch das verrauschte Verhalten beim Trainieren eines mehrschichtigen neuronalen Netzes mit nichtlinearen Aktivierungsfunktionen, die keine konvexe Straffunktion besitzen, als vorteilhaft zu betrachten. Das zusätzliche Rauschen kann in diesem Fall dabei helfen, lokale

Minima der Straffunktion zu überspringen, aber das werden wir später noch ausführlicher erörtern.

### 12.1.2 Mehrschichtige neuronale Netzarchitektur

In diesem Abschnitt werden wir mehrere einzelne Neuronen zu einem *mehrschichtigen Feedforward-Netz* zusammenfassen. Diese spezielle Form eines vollständig verknüpften Netzes wird auch als *mehrschichtiges Perzeptron (MLP, Multi-Layer Perceptron)* bezeichnet. In der nachstehenden Abbildung ist das Konzept eines MLP dargestellt, das aus drei Schichten besteht:



Das abgebildete MLP besteht aus einer Eingabeschicht, einer verdeckten Schicht sowie einer Ausgabeschicht. Die Einheiten in der verdeckten Schicht sind vollständig mit der Eingabeschicht verknüpft, und die Ausgabeschicht ist vollständig mit der verdeckten Schicht verknüpft. Besitzt ein Netz mehr als eine verdeckte Schicht, spricht man von einem *tiefen* künstlichen neuronalen Netz.

#### Tipp

Wir könnten dem MLP eine beliebige Anzahl verdeckter Schichten hinzufügen, um ein tieferes neuronales Netz zu erzeugen. Man kann sich die Anzahl der Schichten und Einheiten in einem neuronalen Netz als zusätzliche *Hyperparameter* vorstellen, die wir mit der in Kapitel 6 erörterten Kreuzvalidierung für eine gegebene Aufgabenstellung optimieren möchten.

Durch das Hinzufügen weiterer Schichten zu dem neuronalen Netz würden die Fehlergradienten, die wir in Kürze mittels des Backpropagation-Verfahrens berechnen werden, zunehmend kleiner werden. Dieses *Problem des verschwindenden Gradienten* erschwert das Lernen des Modells. Aus diesem Grund wurden spezielle Algorithmen zum Trainieren solcher tiefen neuronalen Netze entwickelt, was als *Deep Learning* bezeichnet wird.

In der vorangegangenen Abbildung wird die  $i$ -te Aktivierungseinheit in der  $l$ -ten Schicht als  $a_i^{(l)}$  bezeichnet. Um die Mathematik und den Code etwas verständlicher zu gestalten, verwenden wir keine numerischen Indizes für die verschiedenen Schichten, sondern ein hochgestelltes *in* für die Eingabeschicht, ein hochgestelltes *h* (hidden) für die verdeckte Schicht und ein hochgestelltes *out* für die Ausgabeschicht. Beispielsweise steht  $a_i^{(in)}$  für die  $i$ -te Einheit der Eingabeschicht,  $a_i^{(h)}$  für die  $i$ -te Einheit in der verdeckten Schicht und  $a_i^{(out)}$  für die  $i$ -te Einheit in der Ausgabeschicht. Die Aktivierungseinheiten  $a_0^{(in)}$  und  $a_0^{(h)}$  sind *Bias-Einheiten*, denen wir den Wert 1 zuweisen. Die Aktivierung der Einheiten in der Eingabeschicht ist einfach nur die Eingabe selbst und die Bias-Einheit:

$$a^{(in)} = \begin{bmatrix} a_0^{(in)} \\ a_1^{(in)} \\ \vdots \\ a_m^{(in)} \end{bmatrix} = \begin{bmatrix} 1 \\ x_1^{(in)} \\ \vdots \\ x_m^{(in)} \end{bmatrix}$$

### Tipp

Später in diesem Kapitel werden wir ein mehrschichtiges Perzeptron implementieren, das für die Bias-Einheit separate Vektoren verwendet. Die Implementierung wird dadurch effizienter und besser verständlich. Die Deep-Learning-Bibliothek TensorFlow, die in Kapitel 13 vorgestellt wird, verwendet dieses Konzept ebenfalls. Die nachfolgenden mathematischen Gleichungen wären allerdings komplizierter, wenn wir zusätzliche Variablen für das Bias verwenden würden. Die Berechnung durch das Anhängen von Einsen an den Eingabevektor (wie es schon vorkam) und die Verwendung einer Gewichtungsvariablen als Bias entspricht exakt der Vorgehensweise mit separaten Bias-Vektoren; es handelt sich hierbei lediglich um eine andere Konvention.

Alle Einheiten in Schicht  $l$  sind mit allen Einheiten der Schicht  $l + 1$  über Gewichtungskoeffizienten verknüpft. Die Verknüpfung zwischen der  $k$ -ten Einheit in Schicht  $l$  mit der  $j$ -ten Einheit in Schicht  $l + 1$  wird beispielsweise als  $w_{k,j}^{(l)}$  notiert. Entsprechend der vorangegangenen Abbildung notieren wir die Gewichtsmatrix, die Eingabeschicht und verdeckte Schicht verknüpft, als  $W^{(h)}$  und die Matrix, die verdeckte Schicht und Ausgabeschicht verknüpft, als  $W^{(out)}$ .

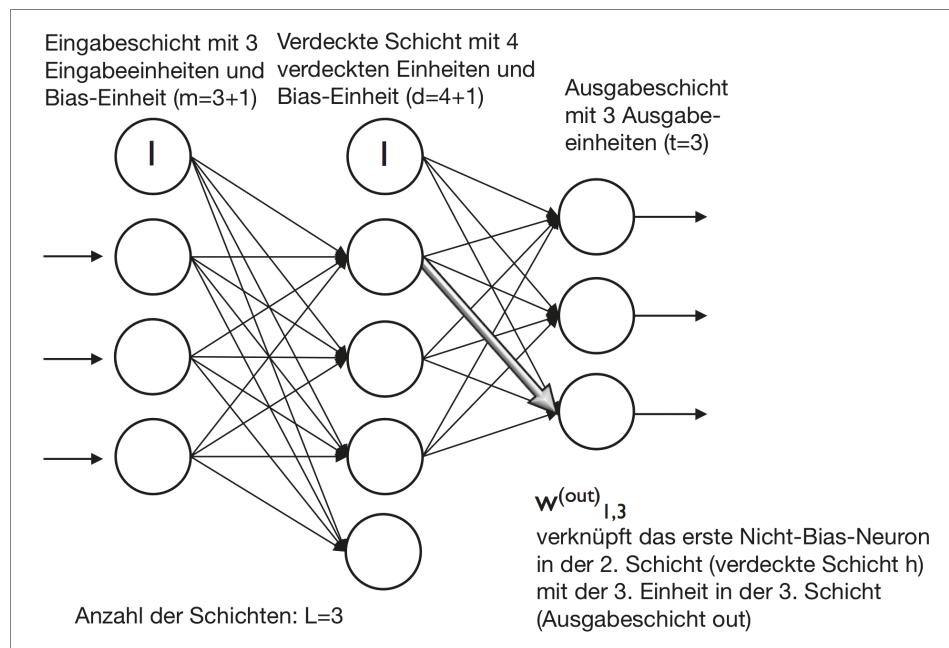
Für eine binäre Klassifizierung wäre zwar eine Einheit in der Ausgabeschicht ausreichend, in der vorangegangenen Abbildung ist jedoch eine allgemeinere Form eines neuronalen Netzes dargestellt, die es ermöglicht, mittels des *One-vs-All-Verfahrens* (*OvA*) auch Mehrfachklassifizierungen vorzunehmen. Denken Sie an die in Kapitel 4 vorgestellte *One-hot-Codierung* kategorialer Variablen, um die Funktions-

weise besser zu verstehen. So würden wir beispielsweise die drei Klassenbezeichnungen in der Iris-Datensammlung (0=Setosa, 1=Versicolor, 2=Virginica) folgendermaßen codieren:

$$0 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, 1 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, 2 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Diese Codierung ermöglicht es, Klassifizierungsaufgaben mit einer beliebigen Anzahl eindeutiger Klassenbezeichnungen in der Trainingsdatenmenge in Angriff zu nehmen.

Falls Ihnen die Notation neuronaler Netze noch nicht vertraut ist, erscheint die Notation bezüglich der hoch- und tiefgestellten Indizes ein wenig verwirrend. Was hier noch übermäßig kompliziert erscheint, ergibt einen Sinn, sobald wir die Repräsentation neuronaler Netze vektorisieren. Beispielsweise fassen wir die Gewichtungen, die die Eingabeschicht und die verdeckte Schicht miteinander verknüpfen, in einer Matrix  $W^{(h)} \in \mathbb{R}^{m \times d}$  zusammen, wobei  $d$  die Anzahl der verdeckten Einheiten und  $m$  die Anzahl der Eingabeeinheiten nebst Bias-Einheit bezeichnen. Da es für das Verständnis der in diesem Kapitel noch folgenden Konzepte wichtig ist, diese Schreibweise zu verinnerlichen, ist sie hier noch einmal anhand eines vereinfachten mehrschichtigen 3-4-3-Perzeptrons in einer beschreibenden Abbildung dargestellt:



### 12.1.3 Aktivierung eines neuronalen Netzes durch Vorwärtspropagation

In diesem Abschnitt betrachten wir die Vorgänge bei einer Vorwärtspropagation, um die Ausgabe eines MLP-Modells zu berechnen. Zu diesem Zweck fassen wir das MLP-Lernverfahren in drei Schritten zusammen:

1. Von der Eingabeschicht aus werden die Muster der Trainingsdaten durch das Netz vorwärts propagierte, um eine Ausgabe zu erzeugen.
2. Anhand der Ausgabe wird der Fehler berechnet, den wir durch eine Straffunktion minimieren möchten, auf die wir noch zu sprechen kommen.
3. Der Fehler wird durch das Netz zurückpropagiert. Dann berechnen wir dessen Ableitung bezüglich der Gewichtungen im Netz und aktualisieren das Modell.

Nachdem wir diese Schritte für mehrere Epochen durchgeführt und die Gewichtungen des MLP ermittelt haben, nutzen wir die Vorwärtspropagation, um die Ausgabe des Netzes zu berechnen und wenden eine Schwellenwertfunktion an, um die vorhergesagten Klassenbezeichnungen in Form der im vorangegangenen Abschnitt beschriebenen One-hot-Codierung zu erhalten.

Nun gehen wir die Schritte der Vorwärtspropagation der Reihe nach durch, um mit den Mustern der Trainingsdaten Ausgaben zu erzeugen. Da alle Einheiten in der verdeckten Schicht mit allen Einheiten der Eingabeschicht verknüpft sind, berechnen wir zunächst die Aktivierung  $a_1^{(h)}$  wie folgt:

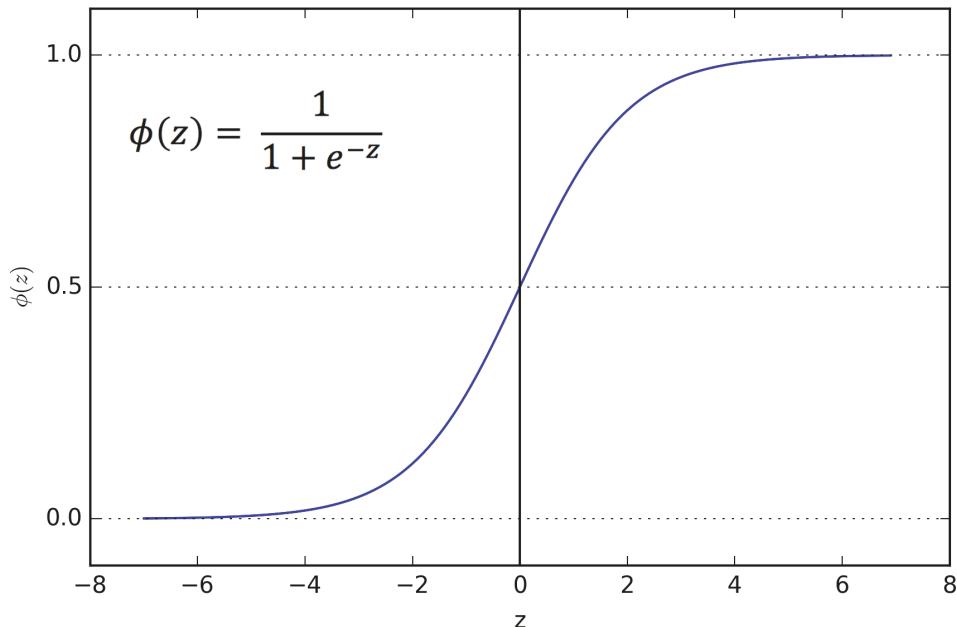
$$z_1^{(h)} = a_0^{(in)} w_{0,1}^{(h)} + a_1^{(in)} w_{1,1}^{(h)} + \dots + a_m^{(in)} w_{m,1}^{(h)}$$

$$a_1^{(h)} = \phi(z_1^{(h)})$$

Hier ist  $z_1^{(h)}$  die Nettoeingabe und  $\phi(\cdot)$  die Aktivierungsfunktion, die differenzierbar sein muss, um die Neuronen verknüpfenden Gewichtungen mit einem gradientenbasierten Ansatz berechnen zu können. Zur Lösung komplexer Aufgaben wie der Bildklassifizierung sind nichtlineare Aktivierungsfunktionen für das MLP-Modell erforderlich, wie beispielsweise die *sigmoide* (logistische) Aktivierungsfunktion, die wir bei der *logistischen Regression* in Kapitel 3 verwendet haben:

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

Wie Sie wissen, ist die Sigmoidfunktion eine S-förmige Kurve, mit der die Nettoeingabe  $z$  auf eine logistische Verteilung im Bereich zwischen 0 und 1 abgebildet wird. Sie schneidet die y-Achse wie in der folgenden Abbildung bei  $z = 0$ .



Das MLP ist ein typisches Beispiel für ein künstliches neuronales *Feedforward*-Netz. Der Begriff *feedforward* (*vorwärtsgerichtet*) bezieht sich darauf, dass jede Schicht jeweils als Eingabe für die nachfolgende Schicht dient, und zwar ohne Schleifen – im Gegensatz zu *rekurrenten* (*rückwärtsgerichteten*) neuronalen Netzen, die später in diesem Kapitel und ausführlicher in Kapitel 16 (*Modellierung sequenzieller Daten durch rekurrente neuronale Netze*) noch zur Sprache kommen. Der Ausdruck »mehrschichtiges Perzeptron« klingt vielleicht etwas irreführend, weil die künstlichen Neuronen dieses Netzes typischerweise sigmoide Einheiten sind und keine Perzeptrons. Man kann sich die Neuronen in einem MLP vielmehr als Einheiten vorstellen, die eine logistische Regression durchführen und stetige Werte zwischen 0 und 1 zurückliefern.

Zwecks effizienterer Codes und besserer Lesbarkeit notieren wir die Aktivierung ab jetzt in einer kompakteren Form und verwenden dafür die Konzepte der grundlegenden linearen Algebra. Dies ermöglicht es uns, die Implementierung mit NumPy zu *vektorisieren*, anstatt mehrfach verschachtelte und rechenaufwendige *for*-Schleifen zu verwenden:

$$z^{(h)} = \mathbf{a}^{(in)} \mathbf{W}^{(h)}$$

$$\mathbf{a}^{(h)} = \phi(z^{(h)})$$

Hier ist  $\mathbf{a}^{(in)}$  der  $1 \times m$ -dimensionale Merkmalsvektor eines Objekts  $\mathbf{x}^{(in)}$  nebst Bias-Einheit.  $\mathbf{W}^{(h)}$  ist eine  $m \times d$ -dimensionale Gewichtungsmatrix, bei der  $d$  die Anzahl der Einheiten in der verdeckten Schicht angibt. Nach der Matrizenmultiplikation erhalten wir den  $1 \times d$ -dimensionalen Nettoeingabevektor  $\mathbf{z}^{(h)}$  zur Berechnung der Aktivierung  $\mathbf{a}^{(h)}$  (wobei  $\mathbf{a}^{(h)} \in \mathbb{R}^{1 \times d}$ ). Außerdem können wir diese Berechnung für alle  $n$  Objekte in der Trainingsdatenmenge verallgemeinern:

$$\mathbf{Z}^{(h)} = \mathbf{A}^{(in)} \mathbf{W}^{(h)}$$

$\mathbf{A}^{(in)}$  ist nun eine  $n \times m$ -Matrix und die Matrizenmultiplikation ergibt eine  $n \times d$ -dimensionale Nettoeingabematrix  $\mathbf{Z}^{(h)}$ . Anschließend wenden wir die Aktivierungsfunktion  $\phi(\cdot)$  auf alle Werte in der Nettoeingabematrix an und erhalten so die  $n \times d$ -Aktivierungsmatrix  $\mathbf{A}^{(h)}$  für die nächste Schicht (in diesem Fall die Ausgabeschicht):

$$\mathbf{A}^{(h)} = \phi(\mathbf{Z}^{(h)})$$

Auf ähnliche Weise können wir auch die Aktivierung der Ausgabeschicht in vektorisierter Form schreiben:

$$\mathbf{Z}^{(out)} = \mathbf{A}^{(h)} \mathbf{W}^{(out)}$$

Hier multiplizieren wir die  $d \times t$ -Matrix  $\mathbf{W}^{(out)}$  ( $t$  ist die Anzahl der Ausgabeeinheiten) mit der  $n \times d$ -dimensionalen Matrix  $\mathbf{A}^{(h)}$  und erhalten die  $n \times t$ -dimensionale Matrix  $\mathbf{Z}^{(out)}$  (deren Spalten die Ausgaben für die Objekte der Stichprobe repräsentieren).

Abschließend wenden wir die sigmoide Aktivierungsfunktion an, um die stetigen Ausgabewerte des Netzes zu erhalten:

$$\mathbf{A}^{(out)} = \phi(\mathbf{Z}^{(out)}), \quad \mathbf{A}^{(out)} \in \mathbb{R}^{n \times t}$$

## 12.2 Klassifizierung handgeschriebener Ziffern

Im vorangegangenen Abschnitt haben wir eine Menge theoretische Grundlagen zu den neuronalen Netzen erörtert, die ein wenig erdrückend erscheinen mögen, wenn dieses Thema Neuland für Sie ist. Bevor wir mit der Betrachtung des Back-propagation-Verfahrens zur Ermittlung der Gewichtungen eines MLP-Modells fortfahren, lassen wir die Theorie an dieser Stelle einmal kurz hinter uns und sehen uns ein neuronales Netz in Aktion an.

## Tipp

Die Theorie der neuronalen Netze kann ziemlich kompliziert sein, daher möchte ich Ihnen zwei weitere Ressourcen empfehlen, die sich einigen der in diesem Kapitel erörterten Konzepte ausführlicher widmen:

I. Goodfellow, Y. Bengio und A. Courville, Kapitel 6, *Deep Feedforward Networks, Deep Learning*, MIT Press, 2016. (Unter <http://www.deeplearningbook.org> kostenlos verfügbar.)

C.M. Bishop et al., *Pattern Recognition and Machine Learning*, Band 1, Springer New York, 2006.

In diesem Abschnitt werden wir ein erstes mehrschichtiges neuronales Netz für die Klassifizierung handgeschriebener Ziffern der bekannten *MNIST-Datensammlung* trainieren. Die Abkürzung steht für *Mixed National Institute of Standard and Technology* (Nationales Institut für Standards und Technologie). Diese von Yann LeCun et al. zusammengestellte Datensammlung wird häufig als Prüfmenge für maschinelle Lernalgorithmen eingesetzt (Y. LeCun, L. Bottou, Y. Bengio und P. Haffner, *Gradient-based Learning Applied to Document Recognition*, Proceedings of the IEEE, 86(11):2278-2324, November 1998).

### 12.2.1 Die MNIST-Datensammlung

Die MNIST-Datensammlung ist unter <http://yann.lecun.com/exdb/mnist/> öffentlich verfügbar und besteht aus den folgenden vier Teilen:

- *Trainingsdatenmenge Bilder*: `train-images-idx3-ubyte.gz` (9.9 MB, 47 MB entpackt, 60.000 Objekte)
- *Trainingsdatenmenge Bezeichnungen*: `train-labels-idx1-ubyte.gz` (29 KB, 60 KB entpackt, 60.000 Bezeichnungen)
- *Testdatenmenge Bilder*: `t10k-images-idx3-ubyte.gz` (1.6 MB, 7.8 MB entpackt, 10.000 Objekte)
- *Testdatenmenge Bezeichnungen*: `t10k-labels-idx1-ubyte.gz` (5 KB, 10 KB entpackt, 10.000 Bezeichnungen)

Die MNIST-Daten wurden aus zwei Datensammlungen des amerikanischen *National Institute of Standards and Technology (NIST)* zusammengestellt. Die Trainingsdatenmenge enthält von 250 verschiedenen Personen mit der Hand geschriebene Ziffern. 50 Prozent davon waren Studenten und die anderen 50 Prozent Mitarbeiter des Statistikamtes. Beachten Sie, dass die zur Testdatenmenge gehörenden handgeschriebenen Ziffern im gleichen Verhältnis von Studenten und Mitarbeitern stammen.

Nach dem Herunterladen können Sie die Dateien im Terminal mit dem Kommandozeilentool `gzip` wie folgt in Ihr lokales `mnist`-Verzeichnis entpacken:

```
gzip *ubyte.gz -d
```

Sollten Sie Microsoft Windows verwenden, können Sie alternativ auch ein Entkomprimierungsprogramm Ihrer Wahl verwenden. Die Bilder sind in einer Byte-Reihenfolge gespeichert und werden in NumPy-Arrays eingelesen, die wir zum Trainieren und Testen der MLP-Implementierung verwenden. Dazu verwenden wir die folgende Hilfsfunktion:

```
import os
import struct
import numpy as np

def load_mnist(path, kind='train'):
    """MNIST-Daten von `path` laden"""
    labels_path = os.path.join(path,
                               '%s-labels-idx1-ubyte' % kind)
    images_path = os.path.join(path,
                               '%s-images-idx3-ubyte' % kind)

    with open(labels_path, 'rb') as lbpath:
        magic, n = struct.unpack('>II',
                                 lbpath.read(8))
        labels = np.fromfile(lbpath,
                             dtype=np.uint8)

    with open(images_path, 'rb') as imgpath:
        magic, num, rows, cols = struct.unpack(">IIII",
                                                imgpath.read(16))
        images = np.fromfile(imgpath,
                            dtype=np.uint8).reshape(
                                len(labels), 784)

    return images, labels
```

Die `load_mnist`-Funktion gibt zwei Arrays zurück. Das erste ist ein  $n \times m$ -dimensionales NumPy-Array (`images`), das  $n$  Objekte mit jeweils  $m$  Merkmalen (hier die Pixel) enthält. Die Trainingsdatenmenge besteht aus 60.000 und die Testdatenmenge aus 10.000 Ziffern. Die Bilder sind  $28 \times 28$  Pixel groß und alle Pixel besitzen einen Wert auf einer Grauskala. Wir speichern die  $28 \times 28$  Pixel in einem eindimensionalen Zeilenvektor, der die Ziffern in unserem `images`-Array repräsentiert (784 Werte pro Zeile oder Bild). Das zweite von der `load_mnist`-Funktion

zurückgegebene Array (`labels`) enthält die zugehörigen Zielvariablen, die Klassenbezeichnungen (Ganzzahlen von 0 bis 9) der handgeschriebenen Ziffern.

Die Art und Weise, wie die Bilder eingelesen werden, sieht auf den ersten Blick etwas sonderbar aus:

```
>>> magic, n = struct.unpack('>II', lbpath.read(8))
>>> labels = np.fromfile(lbpath, dtype=np.int8)
```

Um die Funktionsweise dieser beiden Codezeilen zu verstehen, muss man sich die Beschreibung der Datensammlung auf der MNIST-Website ansehen:

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000801(2049)	magic number (MSB first)
0004	32 bit integer	60000	number of items
0008	unsigned byte	??	label
0009	unsigned byte	??	label
.....			
xxxx	unsigned byte	??	label

Mit den beiden Codezeilen lesen wir zunächst die `magic number` (die eine Beschreibung des Dateiprotokolls enthält) und die Anzahl der Objekte (`n`) über den Filebuffer ein. Anschließend werden die nachfolgenden Bytes mithilfe der `fromfile`-Methode in ein NumPy-Array eingelesen. Der Wert `>II`, den wir als Argument an `struct.unpack` übergeben, besteht aus zwei Teilen:

- `>`: Legt *Big-Endian* als Byte-Reihenfolge (*Endianness*) fest. Sollten Ihnen die Begriffe *Big-Endian* und *Little-Endian* nicht geläufig sein, finden Sie bei Wikipedia einen lesenswerten Artikel zu diesem Thema:  
<https://de.wikipedia.org/wiki/Byte-Reihenfolge>
- `I`: Eine vorzeichenlose Ganzzahl.

Abschließend normieren wir die Pixelwerte der MNIST-Daten, die ursprünglich Werte zwischen 0 und 255 besitzen, auf das Intervall von -1 bis 1:

```
images = ((images / 255.) - .5) * 2
```

Der Grund hierfür ist, dass die gradientenbasierte Optimierung unter diesen Bedingungen, wie in Kapitel 2 erörtert, deutlich stabiler ist. Beachten Sie, dass die Bilder pixelweise skaliert werden – ein Ansatz, der sich von den in den vorangegangenen Kapiteln verwendeten Merkmalsskalierungen unterscheidet. Bislang haben wir die Skalierungsparameter aus der Trainingsdatenmenge abgeleitet und zur Skalierung der Spalten in der Trainings- und Testdatenmenge verwendet. Wenn man es mit Pixeln von Bildern zu tun hat, ist es üblich, sie um den Wert 0

zu zentrieren und auf das Intervall von -1 bis 1 abzubilden, was in der Praxis für gewöhnlich gut funktioniert.

### Tipp

Die Stapelverarbeitung normierter Eingabedaten ist ein kürzlich entwickelter Trick zur Verbesserung der Konvergenz von gradientenbasierten Optimierungen. Hierbei handelt es sich um ein fortgeschrittenes Thema, das in diesem Buch nicht erörtert wird. Wenn Sie an Deep-Learning-Anwendungen interessiert sind, kann ich die Lektüre des Artikels *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift* von Sergey Ioffe und Christian Szegedy (2015, <https://arxiv.org/abs/1502.03167>) nur wärmstens empfehlen.

Der nachstehende Code liest die 60.000 Trainingsinstanzen sowie die 10.000 Testdatenobjekte aus dem `mnist`-Verzeichnis ein, in dem die MNIST-Datensammlung entpackt wurde. Hier wird davon ausgegangen, dass sich die MNIST-Dateien bei der Ausführung des Codes im aktuellen Arbeitsverzeichnis befinden:

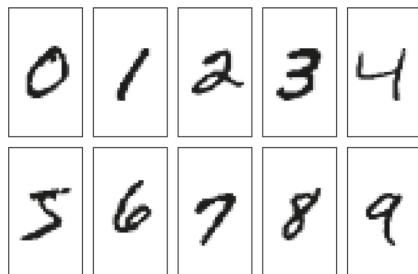
```
>>> X_train, y_train = load_mnist('mnist', kind='train')
>>> print('Zeilen: %d, Spalten: %d'
...      % (X_train.shape[0], X_train.shape[1]))
Zeilen: 60000, Spalten: 784
>>> X_test, y_test = load_mnist('mnist', kind='t10k')
>>> print('Zeilen: %d, Spalten: %d'
...      % (X_test.shape[0], X_test.shape[1]))
Zeilen: 10000, Spalten: 784
```

Um eine Vorstellung davon zu bekommen, wie die MNIST-Bilder aussehen, geben wir einige Beispiele der Ziffern 0 bis 9 aus, nachdem wir die 784-Pixel-Vektoren wieder in das ursprüngliche  $28 \times 28$ -Pixel-Format umgewandelt haben, das mit der `imshow`-Funktion von Matplotlib darstellbar ist:

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(nrows=2, ncols=5,
...                         sharex=True, sharey=True)
>>> ax = ax.flatten()
>>> for i in range(10):
...     img = X_train[y_train == i][0].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys')

>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

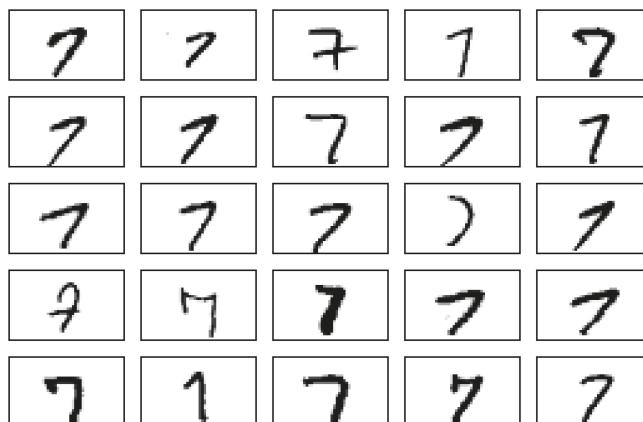
Nun sollte eine Grafik mit  $2 \times 5$  Darstellungen der verschiedenen Ziffern angezeigt werden:



Darüber hinaus geben wir jetzt mehrere Beispiele derselben Ziffer aus, um zu sehen, wie unterschiedlich die handgeschriebenen Exemplare tatsächlich sind:

```
>>> fig, ax = plt.subplots(nrows=5,
...                         ncols=5,
...                         sharex=True,
...                         sharey=True)
>>> ax = ax.flatten()
>>> for i in range(25):
...     img = X_train[y_train == 7][i].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys')
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

Nach der Ausführung des Codes sollten die ersten 25 Varianten der Ziffer 7 angezeigt werden:



Nach der Durchführung der genannten Schritte sollten die skalierten Bilder in einem Format gespeichert werden, das es nicht erforderlich macht, die Daten immer wieder verarbeiten zu müssen, und sich in einer neuen Python-Sitzung schneller einlesen lässt. Bei der Verwendung von NumPy-Arrays bietet NumPys `savez`-Funktion eine effiziente und dennoch komfortable Methode zum Speichern mehrdimensionaler Arrays. Die offizielle Dokumentation finden Sie unter <https://docs.scipy.org/doc/numpy/reference/generated/numpy.savez.html>.

Die `savez`-Funktion ist kurz ausgedrückt ein Pendant zu Pythons `pickle`-Modul, das wir in Kapitel 9 verwendet haben, allerdings ist sie für das Speichern von NumPy-Arrays optimiert. Sie erstellt zip-Archive der Daten mit der Dateiendung `.npz`, die Dateien im `.npy`-Format enthalten. Wenn Sie mehr über dieses Format erfahren möchten, finden Sie in der NumPy-Dokumentation unter <https://docs.scipy.org/doc/numpy/neps/npy-format.html> eine schöne Erklärung inklusive einer Erörterung der Vor- und Nachteile. Statt `savez` werden wir jedoch `savez_compressed` verwenden, das zwar dieselbe Syntax wie `savez` verwendet, aber deutlich kleinere Dateien erzeugt (in diesem Fall ca. 22 MB statt 400 MB). Der folgende Code speichert die Trainings- und Testdatenmengen in einer Archivdatei namens `mnist_scaled.npz`:

```
>>> import numpy as np
>>> np.savez_compressed('mnist_scaled.npz',
...                      X_train=X_train,
...                      y_train=y_train,
...                      X_test=X_test,
...                      y_test=y_test)
```

Nach dem Erstellen der `.npz`-Datei können wir die vorverarbeiteten Arrays der MNIST-Bilder mit NumPys `load`-Funktion wie folgt einlesen:

```
>>> mnist = np.load('mnist_scaled.npz')
```

Die Variable `mnist` verweist nun auf ein Objekt, das auf die vier Datenarrays zugreifen kann, die wir als Schlüsselwortargumente an die `savez_compressed`-Funktion übergeben haben. Diese werden als Dateiattribute des `mnist`-Objekts aufgeführt:

```
>>> mnist.files
['X_train', 'y_train', 'X_test', 'y_test']
```

Um beispielsweise die Trainingsdaten in die aktuelle Python-Sitzung einzulesen, greifen wir auf das `X_train`-Array ähnlich wie auf ein Python-Dictionary zu:

```
>>> X_train = mnist['X_train']
```

Mit einer Listenabstraktion können wir alle vier Arrays abrufen:

```
>>> X_train, y_train, X_test, y_test = [mnist[f] for
...                                     f in mnist.files]
```

Die beiden Beispiele für `np.savez_compressed` und `np.load` sind für die Ausführung des in diesem Kapitel vorgestellten Codes nicht unbedingt nötig, sie sollen vielmehr demonstrieren, wie sich NumPy-Arrays komfortabel und effizient speichern und laden lassen.

### 12.2.2 Implementierung eines mehrschichtigen Perzeptrons

In diesem Abschnitt werden wir zur Klassifizierung der Bilder in der MNIST-Datensammlung den Code eines MLP mit einer Eingabeschicht, einer verdeckten Schicht und einer Ausgabeschicht implementieren. Ich habe zwar versucht, den Code so einfach wie möglich zu halten, allerdings ist er dennoch ziemlich komplex. Insofern empfiehlt es sich, den Beispielcode dieses Kapitels herunterzuladen (<https://github.com/rasbt/python-machine-learning-book-2nd-edition>), der zusätzlich kommentiert und der besseren Lesbarkeit halber mit Syntaxhervorhebungen versehen ist. Wenn Sie den Code nicht mit einem IPython-Notebook ausführen, sollten Sie ihn in eine Skriptdatei in Ihrem aktuellen Arbeitsverzeichnis kopieren, die Sie beispielsweise `neuronalesnetz.py` nennen könnten, und dann mit folgendem Befehl in Ihre aktuelle Python-Sitzung importieren:

```
from neuronalesnetz import NeuralNetMLP
```

Einige Teile dieses Codes, wie etwa das Backpropagation-Verfahren, haben wir bislang zwar noch nicht erörtert, das meiste sollte Ihnen jedoch vertraut sein, denn der Code beruht auf der Adaline-Implementierung aus Kapitel 2 und der in den vorangegangenen Abschnitten erläuterten Vorwärtspropagation.

Machen Sie sich keine Gedanken, wenn nicht alle Teile des Codes auf Anhieb einen Sinn für Sie ergeben – bestimmte Abschnitte kommen erst im weiteren Verlauf dieses Kapitels zur Sprache. Den Code schon jetzt zu betrachten, kann jedoch das Verständnis der nachfolgenden theoretischen Ausführungen erleichtern.

```
import numpy as np
from scipy.special import expit
import sys

class NeuralNetMLP(object):
    """ Neuronales Feedforward-Netz / mehrschichtiges
        Perzepron als Klassifizierer
```

```

Parameter
-----
n_hidden : int (default: 30)
    Anzahl der verdeckten Einheiten.
l2 : float (default: 0.)
    Lambda-Wert der L2-Regularisierung.
    Keine Regularisierung, wenn l2=0 (Vorgabe).
epochs : int (default: 100)
    Anzahl der Durchläufe der Trainingsdatenmenge.
eta : float (default: 0.001)
    Lernrate.
shuffle : bool (default: True)
    Durchmischen der Trainingsdaten nach jeder Epoche,
    falls True, um Wiederholungen zu vermeiden.
minibatch_size : int (default: 1)
    Anzahl der Trainingsobjekte pro
    Mini-Stapelverarbeitung.
seed : int (default: None)
    Initialisierung des Zufallszahlengenerators für
    Gewichtungen und Durchmischen.

Attribute
-----
eval_ : dict
    Dictionary für Werte der Straffunktion, Korrekt-
    klassifizierungsrate der Trainings und
    Validierungsmenge der Epochen während des Trainings.
"""
def __init__(self, n_hidden=30,
             l2=0.0, epochs=100, eta=0.001,
             shuffle=True, minibatch_size=1, seed=None):
    np.random.seed(random_state)
    self.n_hidden = n_hidden
    self.l2 = l2
    self.epochs = epochs
    self.eta = eta
    self.shuffle = shuffle
    self.minibatch_size = minibatch_size

    def _onehot(self, y, n_classes):
        """ One-hot-Codierung der Klassenbezeichnungen
Parameter
-----
y : array, shape = [n_samples]
    Zielwerte.

```

```
Rückgabewerte
-----
onehot : array, shape = (n_samples, n_labels)
"""
onehot = np.zeros((n_classes, y.shape[0]))
for idx, val in enumerate(y.astype(int)):
    onehot[val, idx] = 1.
return onehot.T

def _sigmoid(self, z):
    """Logistische Funktion (Sigmoid) berechnen."""
    return 1. / (1. + np.exp(-np.clip(z, -250, 250)))

def _forward(self, X):
    """Vorwärtspropagation berechnen"""
    # Schritt 1: Nettoeingabe der verdeckten Schicht
    # [n_samples, n_features] dot [n_features, n_hidden]
    # -> [n_samples, n_hidden]
    z_h = np.dot(X, self.w_h) + self.b_h

    # Schritt 2: Aktivierung der verdeckten Schicht
    a_h = self._sigmoid(z_h)

    # Schritt 3: Nettoeingabe der Ausgabeschicht
    # [n_samples, n_hidden] dot [n_hidden, n_classlabels]
    # -> [n_samples, n_classlabels]
    z_out = np.dot(a_h, self.w_out) + self.b_out

    # Schritt 4: Aktivierung der Ausgabeschicht
    a_out = self._sigmoid(z_out)
    return z_h, a_h, z_out, a_out

def _compute_cost(self, y_enc, output):
    """Straffunktion berechnen.

Parameter
-----
y_enc : array, shape = (n_samples, n_labels)
    One-hot-Codierung der Klassenbezeichnungen.
output : array, shape = [n_samples, n_output_units]
    Aktivierung der Ausgabeschicht (Vorwärtspropagation)
Rückgabewerte
-----
cost : float
    Wert der regularisierten Straffunktion
```

```

"""
L2_term = (self.l2 *
            (np.sum(self.w_h ** 2.) +
             np.sum(self.w_out ** 2.)))
term1 = -y_enc * (np.log(output))
term2 = (1. - y_enc) * np.log(1. - output)
cost = np.sum(term1 - term2) + L2_term
return cost

def predict(self, X):
    """Vorhersage der Klassenbezeichnungen
    Parameter
    -----
    X : array, shape = [n_samples, n_features]
        Eingabeschicht mit ursprünglichen Merkmalen.
    Rückgabewerte
    -----
    y_pred : array, shape = [n_samples]
        Vorhergesagte Klassenbezeichnungen.
"""
z_h, a_h, z_out, a_out = self._forward(X)
y_pred = np.argmax(z_out, axis=1)
return y_pred

def fit(self, X_train, y_train, X_valid, y_valid):
    """ Gewichtungen anhand der Trainingsdaten ermitteln.
    Parameter
    -----
    X_train : array, shape = [n_samples, n_features]
        Eingabeschicht mit ursprünglichen Merkmalen.
    y_train : array, shape = [n_samples]
        Zielklassenbezeichnungen.
    X_valid : array, shape = [n_samples, n_features]
        Merkmale zur Validierung während des Trainings.
    y_valid : array, shape = [n_samples]
        Klassenbezeichnungen zur Validierung während des
        Trainings.
    Rückgabewert
    -----
    self
"""
n_output = np.unique(y_train).shape[0]
# Anzahl der Klassenbezeichnungen

n_features = X_train.shape[1]

```

```
#####
# Gewichtungen initialisieren
#####

# Gewichtungen Eingabeschicht -> verdeckte Schicht
self.b_h = np.zeros(self.n_hidden)
self.w_h = self.random.normal(loc=0.0, scale=0.1,
                               size=(n_features,
                                      self.n_hidden))
# Gewichtungen verdeckte Schicht -> Ausgabeschicht
self.b_out = np.zeros(n_output)
self.w_out = self.random.normal(loc=0.0, scale=0.1,
                                 size=(self.n_hidden,
                                       n_output))

epoch_strlen = len(str(self.epochs))
self.eval_ = {'cost': [], 'train_acc': [], \
             'valid_acc': []}
y_train_enc = self._onehot(y_train, n_output)

# Trainingsepochen durchlaufen
for i in range(self.epochs):
    # Mini-Stapel durchlaufen
    indices = np.arange(X_train.shape[0])
    if self.shuffle:
        self.random.shuffle(indices)
    for start_idx in range(0, indices.shape[0] - \
                           self.minibatch_size + \
                           1, self.minibatch_size):
        batch_idx = indices[start_idx:start_idx + \
                            self.minibatch_size]

        # Vorwärtspropagation
        z_h, a_h, z_out, a_out = \
            self._forward(X_train[batch_idx])

    #####
    # Backpropagation
    #####
    # [n_samples, n_classlabels]
    sigma_out = a_out - y_train_enc[batch_idx]

    # [n_samples, n_hidden]
    sigmoid_derivative_h = a_h * (1. - a_h)
```

```

# [n_samples, n_classlabels] dot
#      [n_classlabels, n_hidden]
# -> [n_samples, n_hidden]
sigma_h = (np.dot(sigma_out, self.w_out.T) *
            sigmoid_derivative_h)

# [n_features, n_samples] dot [n_samples,
#                               n_hidden]
# -> [n_features, n_hidden]
grad_w_h=np.dot(X_train[batch_idx].T,sigma_h)
grad_b_h = np.sum(sigma_h, axis=0)

# [n_hidden, n_samples] dot [n_samples,
#                           n_classlabels]
# -> [n_hidden, n_classlabels]
grad_w_out = np.dot(a_h.T, sigma_out)
grad_b_out = np.sum(sigma_out, axis=0)

# Aktualisierung von Regularisierung und
# Gewichtungen
delta_w_h = (grad_w_h + self.l2*self.w_h)
delta_b_h = grad_b_h
# Bias ist nicht regularisiert
self.w_h -= self.eta * delta_w_h
self.b_h -= self.eta * delta_b_h
delta_w_out = (grad_w_out +
               self.l2*self.w_out)
delta_b_out = grad_b_out
# Bias ist nicht regularisiert
self.w_out -= self.eta * delta_w_out
self.b_out -= self.eta * delta_b_out

#####
# Bewertung
#####

# Bewertung nach jeder Epoche des Trainings
z_h, a_h, z_out, a_out = self._forward(X_train)
cost = self._compute_cost(y_enc=y_train_enc,
                           output=a_out)
y_train_pred = self.predict(X_train)
y_valid_pred = self.predict(X_valid)

train_acc = ((np.sum(y_train ==

```

```

        y_train_pred)).astype(np.float) \
X_train.shape[0])
valid_acc = ((np.sum(y_valid ==
        y_valid_pred)).astype(np.float) \
X_valid.shape[0])
sys.stderr.write('\r%0*d/%d | Wert SF: %.2f '
    '| Train/Valid KKR.: '
    '%.2f%%/.2f%% ' %
    (epoch_strlen, i+1, self.epochs,
     cost, train_acc*100,
     valid_acc*100))
sys.stderr.flush()

self.eval_['cost'].append(cost)
self.eval_['train_acc'].append(train_acc)
self.eval_['valid_acc'].append(valid_acc)

return self

```

Nun initialisieren wir ein **784-100-10-MLP** – ein neuronales Netz mit 784 Eingabe-einheiten (**n\_features**), 100 verdeckten Einheiten (**n\_hidden**) und 10 Ausgabe-einheiten (**n\_output**):

```

>>> nn = NeuralNetMLP(n_hidden=100,
...                      l2=0.01,
...                      epochs=200,
...                      eta=0.0005,
...                      minibatch_size=100,
...                      shuffle=True,
...                      seed=1)

```

Nach aufmerksamem Lesen des **NeuralNetMLP**-Codes haben Sie vermutlich schon erraten, welche Aufgaben diese Parameter haben, die nachstehend kurz zusammengefasst sind:

- **l2**: Der Parameter  $\lambda$  der L2-Regularisierung zur Verringerung der Überanpassung.
- **epochs**: Die Anzahl der Durchläufe der Trainingsdatenmenge.
- **eta**: Die Lernrate  $\eta$ .
- **shuffle**: Das Durchmischen der Trainingsdatenmenge vor jeder neuen Epoche verhindert, dass der Algorithmus in einer Schleife hängen bleibt.
- **seed**: Dieser Wert dient zur Initialisierung des Zufallszahlengenerators für das Durchmischen und die Gewichtungen.

- **minibatch\_size:** Die Anzahl der Objekte in den Teilmengen, in die die Trainingsdatenmenge in jeder Epoche des stochastischen Gradientenverfahrens aufgeteilt wird. Der Gradient wird nicht für die gesamte Trainingsdatenmenge, sondern für jede Teilmenge separat berechnet, um das Lernen zu beschleunigen (*Mini-Batch Learning*).

Als Nächstes trainieren wir das MLP mit 55.000 Objekten aus der bereits durchgemischten MNIST-Trainingsdatenmenge und verwenden die verbleibenden 5.000 während des Trainings zur Validierung. Beachten Sie, dass dieses Training des neuronalen Netzes auf normaler Hardware etwa 5 Minuten Zeit in Anspruch nimmt.

Vielleicht ist Ihnen in der Implementierung aufgefallen, dass die `fit`-Methode vier Parameter entgegennimmt: Bilder und Klassenbezeichnungen der Trainingsdaten und der Validierungsdaten. Beim Trainieren eines neuronalen Netzes ist es außerordentlich nützlich, die Korrektklassifizierungsraten von Trainings- und Validierungsmenge zu vergleichen, weil wir für gegebene Architektur und Hyperparameter beurteilen können, ob das Modell gut funktioniert.

Im Vergleich zu den übrigen bislang erörterten Modellen ist das Trainieren (tiefer) neuronaler Netze im Allgemeinen relativ rechenintensiv, deshalb ist es sinnvoll, das Training unter bestimmten Umständen vorzeitig abzubrechen, um andere Hyperparameter ausprobieren zu können. Wenn wir feststellen, dass es bei den Trainingsdaten zunehmend zu einer Überanpassung kommt (bemerkbar durch eine größer werdende Lücke zwischen der Leistung bei Trainings- und Validierungsmengen), ist es ebenfalls sinnvoll, das Training abzubrechen.

Führen Sie folgenden Code aus, um das Training zu starten (Wert SF bedeutet Wert der Straffunktion; Train-Valid-KKR gibt die Korrektklassifizierungsraten für die Trainings- bzw. Validierungsmenge an):

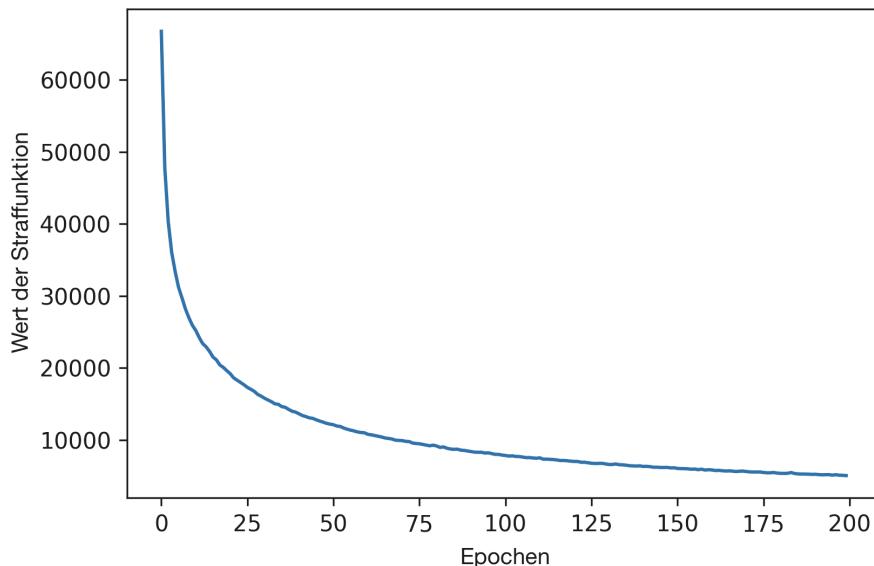
```
>>> nn.fit(X_train=X_train[:55000],
...           y_train=y_train[:55000],
...           X_valid=X_train[55000:],
...           y_valid=y_train[55000:])
200/200 | Wert SF: 5065.78 | Train/Valid-KKR.: 99.28%/97.98%
```

In der `NeuralNetMLP`-Implementierung haben wir ein Attribut `eval_` definiert, das die Werte der Straffunktion und die Korrektklassifizierungsraten für Trainings- bzw. Validierungsmenge aller Epochen speichert, damit wir die Ergebnisse mit Matplotlib visualisieren können:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(range(nn.epochs), nn.eval_['cost'])
>>> plt.ylabel('Wert der Straffunktion')
```

```
>>> plt.xlabel('Epochen')
>>> plt.show()
```

Der Code trägt die Werte der Straffunktion gegen die 200 Epochen auf (siehe Abbildung).

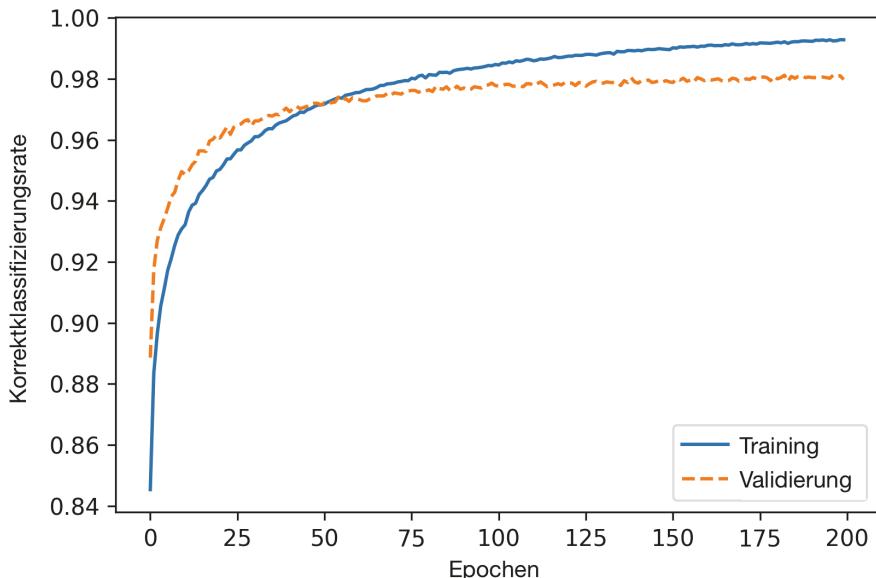


Im Diagramm ist erkennbar, dass der Wert der Straffunktion während der ersten 100 Epochen erheblich sinkt und in den letzten 100 Epochen offenbar allmählich konvergiert. Das leichte Absinken zwischen den Epochen 175 und 200 deutet darauf hin, dass der Wert der Straffunktion weiter sinken würde, wenn man das Modell mit weiteren Epochen trainiert.

Sehen wir uns nun die Korrektklassifizierungsraten der Trainings- und Validierungsdatenmengen an:

```
>>> plt.plot(range(nn.epochs), nn.eval_['train_acc'],
...           label='Training')
>>> plt.plot(range(nn.epochs), nn.eval_['valid_acc'],
...           label='Validierung', linestyle='--')
>>> plt.ylabel('Korrektklassifizierungsrate')
>>> plt.xlabel('Epochen')
>>> plt.legend()
>>> plt.show()
```

Im folgenden Diagramm sind die Korrektklassifizierungsraten als Funktionen der 200 Epochen dargestellt.



Das Diagramm zeigt, dass sich die Lücke zwischen den Korrektklassifizierungsraten der Trainings- und Validierungsdaten mit zunehmender Zahl der Epochen vergrößert. Nach ca. 50 Epochen sind die Werte gleich groß, aber danach kommt es beim neuronalen Netz zu einer Überanpassung der Trainingsdaten.

Beachten Sie hier, dass dieses Beispiel bewusst so gewählt wurde, dass es den Effekt der Überanpassung zeigt und demonstriert, warum es sinnvoll ist, die Korrektklassifizierungsraten von Trainings- und Validierungsdaten während des Trainings zu vergleichen.

Eine Möglichkeit, die Auswirkungen der Überanpassung zu verringern, ist die Erhöhung der Regularisierungsstärke, z.B. indem 12 der Wert 0.1 zugewiesen wird. Ein weiteres nützliches Verfahren, um die Überanpassung neuronaler Netze in den Griff zu bekommen, heißt Dropout und wird in Kapitel 15 (*Bildklassifizierung mit tiefen konvolutionalen neuronalen Netzen*) betrachtet.

Zum Abschluss soll die Verallgemeinerungsfähigkeit des Modells beurteilt werden, indem wir die Korrektklassifizierungsraten für die Testdatensetze berechnen:

Wie man sieht, klassifiziert das Modell die meisten Ziffern der Trainingsdatensetze korrekt. Aber wie steht es um die Verallgemeinerungsfähigkeit auf unbekannte Daten? Berechnen wir also die Korrektklassifizierungsraten für die Testdatensetze:

```
>>> y_test_pred = nn.predict(X_test)
>>> acc = (np.sum(y_test == y_test_pred)
...         .astype(np.float) / X_test.shape[0])
```

```
>>> print('Korrektklassifizierungsrate Test: %.2f%%'  
...  
     '% (acc * 100))  
Korrektklassifizierungsrate Test: 97.54%
```

Trotz der leichten Überanpassung der Trainingsdaten erzielt das einfache neuronale Netz mit einer verdeckten Schicht bei der Testdatenmenge eine relativ gute Leistung, die mit derjenigen für die Validierungsmenge (97.98 Prozent) vergleichbar ist.

Um das Modell noch feiner abzustimmen, könnten wir die Anzahl der verdeckten Einheiten, die Werte für den Regularisierungsparameter und die Lernrate variieren, oder eine Vielzahl weiterer Verfahren einsetzen, die im Laufe der Jahre entwickelt wurden, aber über den Rahmen dieses Buches hinausgehen. In Kapitel 14 (*Die Funktionsweise von TensorFlow im Detail*) werden Sie eine andere neuronale Netzarchitektur kennenlernen, die für ihre gute Leistung bei der Verarbeitung von Bilddatenmengen bekannt ist. In Kapitel 14 werden zudem einige Tricks zur Leistungsverbesserung vorgestellt, wie adaptive Lernraten, »träges« Lernen und Dropout.

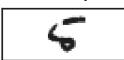
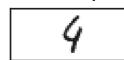
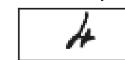
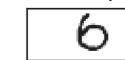
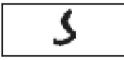
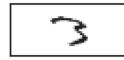
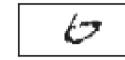
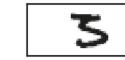
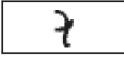
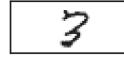
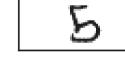
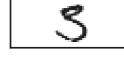
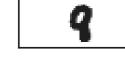
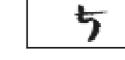
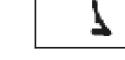
Sehen wir uns nun einige der Ziffern an, die dem MLP Schwierigkeiten bereiten:

```
>>> miscl_img = X_test[y_test != y_test_pred][:25]  
>>> correct_lab = y_test[y_test != y_test_pred][:25]  
>>> miscl_lab= y_test_pred[y_test != y_test_pred][:25]  
>>> fig, ax = plt.subplots(nrows=5,  
...                         ncols=5,  
...                         sharex=True,  
...                         sharey=True,)  
>>> ax = ax.flatten()  
>>> for i in range(25):  
...     img = miscl_img[i].reshape(28, 28)  
...     ax[i].imshow(img,  
...                  cmap='Greys',  
...                  interpolation='nearest')  
...     ax[i].set_title('%d t: %d p: %d'  
...                   % (i+1, correct_lab[i], miscl_lab[i]))  
>>> ax[0].set_xticks([])  
>>> ax[0].set_yticks([])  
>>> plt.tight_layout()  
>>> plt.show()
```

Nun sollte eine  $5 \times 5$ -Matrix mit Darstellungen der Ziffern angezeigt werden. Die erste Zahl in den Beschriftungen gibt den Index an, die zweite die tatsächliche (t) und die dritte die prognostizierte (p) Klassenbezeichnung.

## Kapitel 12

### Implementierung eines künstlichen neuronalen Netzes

1) t: 5 p: 6	2) t: 4 p: 9	3) t: 4 p: 2	4) t: 6 p: 0	5) t: 2 p: 7
				
6) t: 5 p: 3	7) t: 3 p: 7	8) t: 6 p: 0	9) t: 3 p: 5	10) t: 8 p: 0
				
11) t: 7 p: 1	12) t: 3 p: 7	13) t: 1 p: 8	14) t: 2 p: 6	15) t: 2 p: 8
				
16) t: 7 p: 3	17) t: 8 p: 4	18) t: 5 p: 8	19) t: 4 p: 9	20) t: 9 p: 7
				
21) t: 2 p: 7	22) t: 3 p: 5	23) t: 8 p: 9	24) t: 5 p: 4	25) t: 1 p: 2
				

In der Abbildung gibt es einige Ziffern, deren Klassifizierung selbst für einen menschlichen Betrachter eine Herausforderung darstellt. Beispielsweise sieht die Ziffer 6 (Index 8) tatsächlich wie eine flüchtig geschriebene 0 aus, und die 8 (Index 23) könnte aufgrund der schmalen unteren Hälfte und der breiten Strichstärke auch eine 9 sein.

## 12.3 Trainieren eines künstlichen neuronalen Netzes

Damit haben Sie nun ein neuronales Netz in Aktion gesehen und durch die Begutachtung des Codes ein grundlegendes Verständnis von dessen Funktionsweise erlangt. Gehen wir also noch einen Schritt weiter und betrachten die logistische Straffunktion und das Backpropagation-Verfahren, das wir zum Ermitteln der Gewichtungen implementiert haben.

### 12.3.1 Berechnung der logistischen Straffunktion

Die logistische Straffunktion, die wir in Form der `_compute_cost`-Methode implementiert haben, ist tatsächlich ganz einfach zu verstehen, denn es handelt sich um dieselbe Straffunktion, die in Kapitel 3 in dem Abschnitt über die logistische Regression beschrieben wird:

$$J(\mathbf{w}) = -\sum_{i=1}^n y^{[i]} \log(a^{[i]}) + (1 - y^{[i]}) \log(1 - a^{[i]})$$

Hier ist  $a^{[i]}$  die sigmoide Aktivierung der  $i$ -ten Einheit in der Datenmenge, die wir bei der Vorwärtspropagation berechnen:

$$a^{[i]} = \phi(z^{[i]})$$

Beachten Sie hier, dass das hochgestellte  $[i]$  ein Index für die Trainingsobjekte ist, nicht für die Schichten.

Nun fügen wir einen *Regularisierungsterm* hinzu, der es ermöglicht, den Grad der Überanpassung zu verringern. Wie Sie aus den vorangegangenen Kapiteln wissen, ist der L2-Regulierungsterm folgendermaßen definiert (denken Sie daran, dass die Bias-Einheiten nicht regularisiert werden):

$$L2 = \lambda \|\mathbf{w}\|_2^2 = \lambda \sum_{j=1}^m w_j^2$$

Durch das Hinzufügen des L2-Regularisierungsterms zur logistischen Straffunktion erhalten wir folgende Gleichung:

$$J(\mathbf{w}) = - \left[ \sum_{i=1}^n y^{[i]} \log(a^{[i]}) + (1 - y^{[i]}) \log(1 - a^{[i]}) \right] + \frac{\lambda}{2} \|\mathbf{w}\|_2^2$$

Da es sich um ein MLP zur Mehrfachklassifizierung handelt, der einen Ausgabevektor mit  $t$  Elementen zurückgibt, den wir mit dem in Form der One-hot-Codierung vorliegenden  $t \times 1$ -dimensionalen Zielvektor vergleichen müssen. Beispielsweise könnten die Aktivierung der dritten Schicht und die Zielklasse (hier Klasse 2) eines bestimmten Objekts folgendermaßen aussehen:

$$a^{(out)} = \begin{bmatrix} 0.1 \\ 0.9 \\ \vdots \\ 0.3 \end{bmatrix}, \quad y = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

Wir müssen also die logistische Straffunktion für alle Aktivierungseinheiten  $j$  im Netz verallgemeinern. Die Straffunktion (ohne Regularisierungsterm) sieht dann wie folgt aus:

$$J(\mathbf{W}) = - \sum_{i=1}^n \sum_{j=1}^t y_j^{[i]} \log(a_j^{[i]}) + (1 - y_j^{[i]}) \log(1 - a_j^{[i]})$$

Das hochgestellte  $i$  ist hier wieder der Index eines bestimmten Objekts in der Trainingsdatenmenge.

Der folgende verallgemeinerte Regularisierungsterm sieht auf den ersten Blick ziemlich kompliziert aus, aber wir berechnen damit lediglich die Summe aller Gewichtungen einer Schicht  $l$  (ohne den Bias-Term), die wir der ersten Spalte hinzugefügt haben:

$$J(\mathbf{W}) = - \left[ \sum_{i=1}^n \sum_{j=1}^l y_j^{[i]} \log \left( a_{-j}^{[i]} \right) + \left( 1 - y_j^{[i]} \right) \log \left( 1 - a_{-j}^{[i]} \right) \right] + \frac{\lambda}{2} \sum_{l=1}^{L-1} \sum_{i=1}^{u_l} \sum_{j=1}^{u_{l+1}} \left( w_{j,i}^{(l)} \right)^2$$

Dabei bezieht sich  $u_l$  auf die Anzahl der Einheiten in einer gegebenen Schicht  $l$ . Der folgende Ausdruck stellt den Strafterm dar:

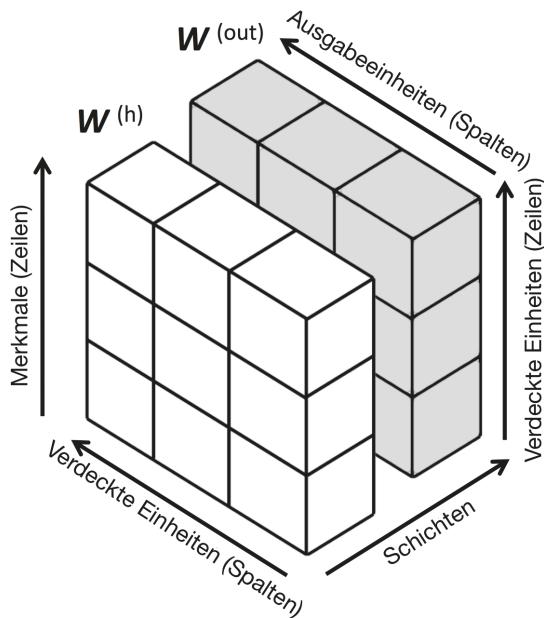
$$\frac{\lambda}{2} \sum_{l=1}^{L-1} \sum_{i=1}^{u_l} \sum_{j=1}^{u_{l+1}} \left( w_{j,i}^{(l)} \right)^2$$

Unser Ziel ist es hier ja, die Straffunktion  $J(\mathbf{W})$  zu minimieren, daher müssen wir die partiellen Ableitungen der Matrix  $\mathbf{W}$  bezüglich aller Gewichtungen in jeder Schicht des Netzes berechnen:

$$\frac{\partial}{\partial w_{j,i}^{(l)}} J(\mathbf{W})$$

Im nächsten Abschnitt kommt das Backpropagation-Verfahren zum Einsatz, mit dem wir diese partiellen Ableitungen zwecks Minimierung der Straffunktion berechnen können.

Beachten Sie, dass  $\mathbf{W}$  aus mehreren Matrizen besteht. Bei einem mehrschichtigen Perzepron mit einer verdeckten Schicht verknüpfen die Gewichtungsmatrizen  $\mathbf{W}^{(h)}$  und  $\mathbf{W}^{(out)}$  die Eingabeschicht mit der verdeckten Schicht bzw. die verdeckte Schicht mit der Ausgabeschicht. Die Visualisierung der Matrix  $\mathbf{W}$  in der folgenden Abbildung soll dies veranschaulichen.



Anhand der vereinfachenden Abbildung könnte man den Eindruck gewinnen, dass  $\mathbf{W}^{(h)}$  und  $\mathbf{W}^{(out)}$  gleich viele Zeilen und Spalten besitzen – das ist im Allgemeinen jedoch nicht der Fall, es sei denn, wir initialisieren ein MLP mit derselben Anzahl von verdeckten Einheiten, Ausgabeeinheiten und Eingabemerkmalen.

Wenn das zunächst verwirrend erscheint, dann lesen Sie den nächsten Abschnitt, in dem die Dimensionalität von  $\mathbf{W}^{(h)}$  und  $\mathbf{W}^{(out)}$  im Kontext des Backpropagation-Verfahrens ausführlicher erläutert wird. Ich empfehle Ihnen außerdem, ein weiteres Mal den Code der NeuralNetMLP-Implementierung zu lesen, den ich mit hilfreichen Kommentaren zur Dimensionalität bezüglich der verschiedenen Matrizen und Vektortransformationen versehen habe. Sie können den kommentierten Code hier herunterladen: <https://github.com/rasbt/python-machine-learning-book-2nd-edition>.

### 12.3.2 Ein Gespür für die Backpropagation entwickeln

Obwohl es schon rund 30 Jahre her ist, dass die Backpropagation wiederentdeckt und populär wurde (D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning representations by back-propagating errors*, Nature, 323: 6088, Seiten 533–536, 1986), repräsentiert dieses Verfahren nach wie vor einen der am häufigsten zum effizienten Trainieren neuronaler Netze eingesetzten Algorithmen. Wenn Sie an weiteren Referenzen zur Geschichte des Backpropagation-Verfahrens interessiert sind, sollten Sie sich den von Juergen Schmidhuber verfassten Übersichtsartikel *Who invented Backpropagation?* ansehen, den Sie online unter <http://people.idsia.ch/~juergen/who-invented-backpropagation.html> finden.

In diesem Abschnitt geht es um eine anschauliche Zusammenfassung und das Gesamtbild der Funktionsweise dieses faszinierenden Algorithmus. Im Wesentlichen stellt das Backpropagation-Verfahren lediglich einen vom Rechenaufwand her äußerst effizienten Ansatz zur Berechnung der partiellen Ableitungen komplizierter Straffunktionen dar. Unser Ziel ist es, diese Ableitungen zum Ermitteln der Gewichtungskoeffizienten zu verwenden, um ein mehrschichtiges künstliches neuronales Netz zu parametrisieren. Bei der Parametrisierung neuronaler Netze gibt es die Schwierigkeit, dass wir es typischerweise mit einer sehr großen Anzahl von Gewichtungskoeffizienten in hochdimensionalen Merkmalsräumen zu tun haben. Im Gegensatz zu anderen Straffunktionen einschichtiger neuronaler Netze wie Adaline oder logistische Regression, die Sie in den vorangegangenen Kapiteln kennengelernt haben, ist die Fehleroberfläche der Straffunktion eines neuronalen Netzes bezüglich der Parameter nicht konvex oder eben. Vielmehr gibt es in dieser hochdimensionalen Fehleroberfläche viele Dellen (lokale Minima), die wir überwinden müssen, um das globale Minimum der Straffunktion zu finden.

Vielleicht erinnern Sie sich aus dem Matheunterricht in Ihrer Schulzeit noch an die Kettenregel, einen Ansatz, um verschachtelte Funktionen wie z.B.  $f(g(x))$  folgendermaßen zu differenzieren:

$$\frac{d}{dx} [f(g(x))] = \frac{df}{dg} \cdot \frac{dg}{dx}$$

Auf ähnliche Weise ist die Kettenregel auf beliebig lange Verknüpfungen von Funktionen anwendbar. Nehmen wir beispielsweise an, es liegen fünf verschiedene Funktionen vor,  $f(x)$ ,  $g(x)$ ,  $h(x)$ ,  $u(x)$  und  $v(x)$ .  $F$  sei die Verknüpfung dieser Funktionen:  $F(x) = f(g(h(u(v(x)))))$ . Durch Anwendung der Kettenregel können wir die Ableitung dieser Funktion folgendermaßen berechnen:

$$\frac{dF}{dx} = \frac{d}{dx} F(x) = \frac{d}{dx} f\left(g\left(h\left(u\left(v(x)\right)\right)\right)\right) = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{du} \cdot \frac{du}{dv} \cdot \frac{dv}{dx}$$

Im Rahmen der Computeralgebra sind verschiedene sehr effiziente Verfahren zum Lösen solcher Aufgabenstellungen entwickelt worden, die als *automatisches Differenzieren* bezeichnet werden. Sollten Sie am Einsatz dieser Verfahren in Machine-Learnings-Anwendungen interessiert sein, empfehle ich die Lektüre des Artikels *Automatic Differentiation of Algorithms for Machine Learning*, arXiv preprint arXiv:1404.7456, 2014, von A.G. Baydin und B.A. Pearlmutter, der unter <http://arxiv.org/pdf/1404.7456.pdf> kostenlos zur Verfügung steht.

Beim automatischen Differenzieren werden zwei Modi unterschieden: *Vorwärts-* und *Rückwärtsmodus*. Das Backpropagation-Verfahren ist lediglich ein Spezialfall des Rückwärtsmodus. Der entscheidende Punkt ist hier, dass die Anwendung der Kettenregel im Vorwärtsmodus sehr rechenintensiv sein kann, da es bei jeder Schicht erforderlich ist, große Matrizen (Jacobi-Matrizen) miteinander zu multiplizieren, die schließlich mit einem Vektor multipliziert werden müssen, um eine Ausgabe zu erhalten. Der Trick beim Rückwärtsmodus besteht darin, dass man von rechts nach links vorgeht: Wir multiplizieren eine Matrix mit einem Vektor, was einen Vektor ergibt, der wiederum mit der nächsten Matrix multipliziert wird usw. Eine Matrix-Vektor-Multiplikation erfordert allerdings einen beträchtlich geringeren Rechenaufwand als eine Matrix-Matrix-Multiplikation – aus diesem Grund stellt das Backpropagation-Verfahren beim Trainieren neuronaler Netze einen der beliebtesten Algorithmen dar.

### Tipp

Zum vollständigen Verständnis des Backpropagation-Verfahrens sind einige Konzepte der Differentialrechnung erforderlich, die über den Rahmen dieses Buches hinausgehen. Ich habe ein Probekapitel über die grundlegenden Konzepte verfasst, das Sie in diesem Zusammenhang vielleicht nützlich finden. Es hat Ableitungen, partielle Ableitungen, Gradienten und die Jacobi-Matrix zum Inhalt. Der Text ist kostenlos verfügbar unter [https://sebastianraschka.com/pdf/books/dlb/appendix\\_d\\_calculus.pdf](https://sebastianraschka.com/pdf/books/dlb/appendix_d_calculus.pdf). Sollte Ihnen die Differentialrechnung nicht vertraut sein oder wenn Sie meinen, eine kleine Auffrischung Ihrer Kenntnisse zu benötigen, sollten Sie dieses Kapitel als ergänzende Ressource lesen, bevor Sie mit der Lektüre des nächsten Abschnitts fortfahren.

### 12.3.3 Trainieren neuronaler Netze durch Backpropagation

In diesem Abschnitt werden wir uns mit der Mathematik der Backpropagation befassen, um zu verstehen, wie sich die Gewichtungen in einem neuronalen Netz äußerst effizient ermitteln lassen. Je nachdem, wie vertraut Ihnen die mathematische Repräsentation ist, können die folgenden Gleichungen auf den ersten Blick relativ kompliziert erscheinen.

In einem vorangegangenen Abschnitt haben wir die Straffunktion als Differenz der Aktivierung der letzten Schicht und der Zielklasse berechnet. Nun werden wir aus mathematischer Perspektive betrachten, wie der im Abschnitt # Backpropagation der `fit`-Methode implementierte Backpropagation-Algorithmus vorgeht, um die Gewichtungen im MLP-Modell zu aktualisieren. Wie Sie vom Anfang dieses Kapitels wissen, müssen wir zunächst eine Vorwärtspropagation einsetzen, um die Aktivierung der Ausgabeschicht zu erhalten. Wir notieren das folgendermaßen:

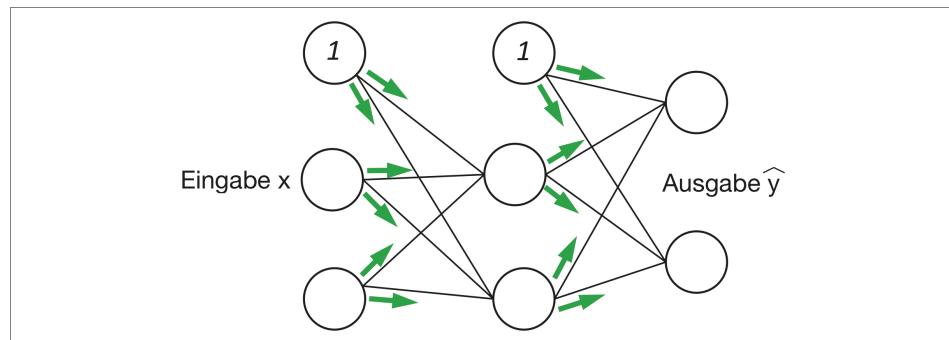
$$\mathbf{Z}^{(h)} = \mathbf{A}^{(in)} \mathbf{W}^{(h)} \quad (\text{Nettoeingabe der verdeckten Schicht})$$

$$\mathbf{A}^{(h)} = \phi(\mathbf{Z}^{(h)}) \quad (\text{Aktivierung der verdeckten Schicht})$$

$$\mathbf{Z}^{(out)} = \mathbf{A}^{(h)} \mathbf{W}^{(out)} \quad (\text{Nettoeingabe der Ausgabeschicht})$$

$$\mathbf{A}^{(out)} = \phi(\mathbf{Z}^{(out)}) \quad (\text{Aktivierung der Ausgabeschicht})$$

Kurz und bündig: Wir propagieren die Eingabemerkmale über die Verknüpfungen des Netzes vorwärtsgerichtet, wie in der Abbildung gezeigt:



Bei der Backpropagation wird der Fehler von rechts nach links propagiert. Zunächst berechnen wir den Fehlervektor der Ausgabeschicht:

$$\delta^{(out)} = \mathbf{a}^{(out)} - \mathbf{y}$$

Hier bezeichnet  $y$  den Vektor der tatsächlichen Klassenbezeichnungen (die entsprechende Variable im `NeuralNetMLP`-Code heißt `sigma_out`).

Als Nächstes erfolgt die Berechnung des Fehlerterms der verdeckten Schicht:

$$\delta^{(h)} = \delta^{(out)} \left( W^{(out)} \right)^T \odot \frac{\partial \phi(z^{(h)})}{\partial z^{(h)}}$$

Der Ausdruck  $\frac{\partial \phi(z^{(h)})}{\partial z^{(h)}}$  ist die Ableitung der sigmoiden Aktivierungsfunktion, die in der `fit`-Methode als `_sigmoid_derivative_h = a_h * (1. - a_h)` implementiert ist:

$$\frac{\partial \phi(z)}{\partial z} = \left( a^{(h)} \odot (1 - a^{(h)}) \right)$$

Beachten Sie hier, dass mit dem Symbol  $\odot$  in diesem Kontext eine elementweise Multiplikation gemeint ist.

### Tipp

Ein umfassendes Verständnis der folgenden Gleichungen ist zwar nicht unbedingt erforderlich, aber für den Fall, dass Sie wissen möchten, wie die Ableitung der Aktivierungsfunktion berechnet wurde, finden Sie die entsprechende Herleitung nachstehend schrittweise beschrieben:

$$\begin{aligned}\phi'(z) &= \frac{\partial}{\partial z} \left( \frac{1}{1 + e^{-z}} \right) \\ &= \frac{e^{-z}}{(1 + e^{-z})^2} \\ &= \frac{1 + e^{-z}}{(1 + e^{-z})^2} - \left( \frac{1}{1 + e^{-z}} \right)^2 \\ &= \frac{1}{(1 + e^{-z})} - \left( \frac{1}{1 + e^{-z}} \right)^2 \\ &= \phi(z) - (\phi(z))^2 \\ &= \phi(z)(1 - \phi(z)) \\ &= a(1 - a)\end{aligned}$$

Als Nächstes berechnen wir die Fehlermatrix  $\delta^{(h)}$  (`sigma_h`) wie folgt:

$$\delta^{(h)} = \delta^{(out)} \left( \mathbf{W}^{(out)} \right)^T \odot \left( a^{(h)} \odot (1 - a^{(h)}) \right)$$

Um die Berechnung des  $\delta^{(h)}$ -Terms besser zu verstehen, betrachten wir sie einmal etwas genauer. In der vorstehenden Gleichung haben wir die Transponierte  $\left( \mathbf{W}^{(out)} \right)^T$  der  $h \times t$ -dimensionalen Matrix  $\mathbf{W}^{(out)}$  verwendet.  $t$  ist die Anzahl der Klassenbezeichnungen der Ausgabe und  $h$  die Anzahl der verdeckten Einheiten. Die Matrizenmultiplikation der  $n \times t$ -dimensionalen Matrix  $\delta^{(out)}$  mit der  $t \times h$ -dimensionalen Matrix  $\left( \mathbf{W}^{(out)} \right)^T$  ergibt eine  $n \times h$ -dimensionale Matrix, die elementweise mit der sigmoiden Ableitung gleicher Dimension multipliziert wird, was eine  $n \times h$ -dimensionale Matrix  $\delta^{(h)}$  liefert.

Schließlich erhalten wir die  $\delta$ -Terme und können die Ableitung der Straffunktion so formulieren:

$$\frac{\partial}{\partial w_{i,j}^{(out)}} J(\mathbf{W}) = a_j^{(h)} \delta_i^{(out)}$$

$$\frac{\partial}{\partial w_{i,j}^{(h)}} J(\mathbf{W}) = a_j^{(in)} \delta_i^{(h)}$$

Als Nächstes müssen wir die partiellen Ableitungen aller Knoten in allen Schichten  $l$  und den Fehler des Knotens in der nächsten Schicht kumulieren. Wir müssen jedoch bedenken, dass wir  $\Delta_{i,j}^{(l)}$  für alle in der Trainingsdatenmenge enthaltenen Objekte berechnen müssen.

$$\Delta_{i,j}^{(l)} := \Delta_{i,j}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$$

Denken Sie daran, dass wir  $\Delta_{i,j}^{(l)}$  für alle Objekte in der Trainingsdatenmenge berechnen müssen. Es ist daher einfacher, die Berechnung wie im `NeuralNetMLP`-Code als vektorisierte Version zu implementieren:

$$\Delta^{(h)} = \Delta^{(h)} + \left( \mathbf{A}^{(in)} \right)^T \delta^{(h)}$$

$$\Delta^{(out)} = \Delta^{(out)} + \left( \mathbf{A}^{(h)} \right)^T \delta^{(out)}$$

Nach dem Kumulieren der partiellen Ableitungen können wir den Regularisierungsterm folgendermaßen hinzufügen:

$$\Delta^{(l)} := \Delta^{(l)} + \lambda^{(l)} \text{ (außer für den Bias-Term)}$$

Die beiden letzten Gleichungen entsprechen im `NeuralNetMLP`-Code den Variablen `delta_w_h`, `delta_b_h`, `delta_w_out` und `delta_b_out`.

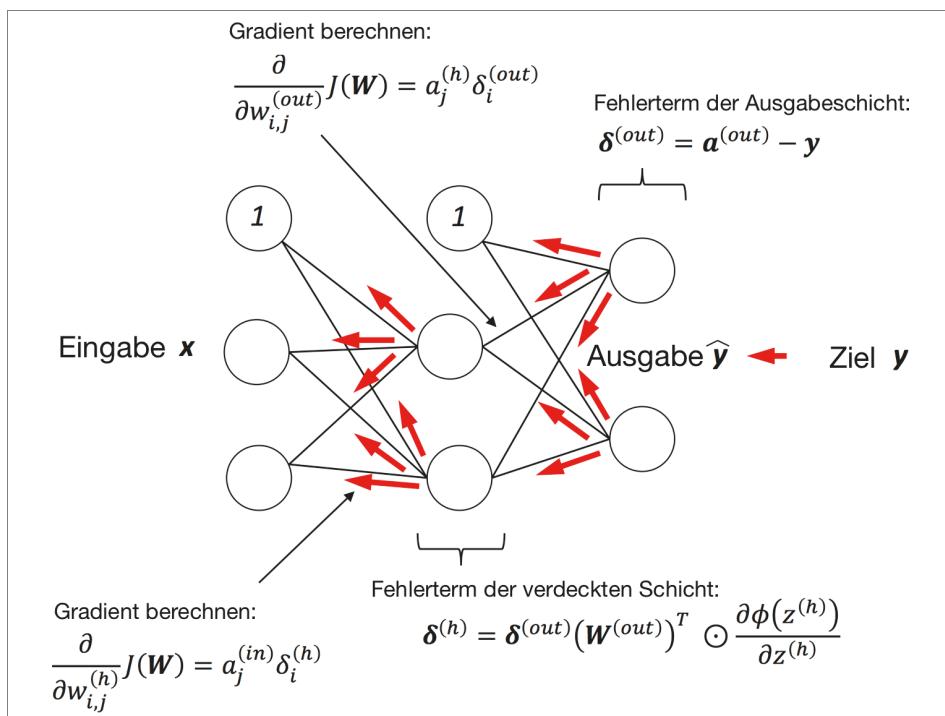
Im Anschluss an die Berechnung der Gradienten können wir schließlich die Gewichtungen aktualisieren, indem wir uns in jeder Schicht einen Schritt in die entgegengesetzte Richtung des Gradienten begeben:

$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} - \eta \Delta^{(l)}$$

Dies ist folgendermaßen implementiert:

```
self.w_h -= self.eta * delta_w_h
self.b_h -= self.eta * delta_b_h
self.w_out -= self.eta * delta_w_out
self.b_out -= self.eta * delta_b_out
```

Das Backpropagation-Verfahren ist in der nachfolgenden Abbildung als Ganzes zusammengefasst.

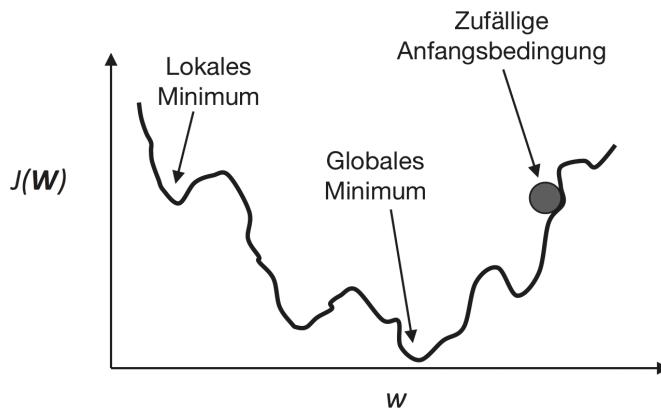


## 12.4 Konvergenz in neuronalen Netzen

Vielleicht haben Sie sich schon gefragt, warum wir keinen normalen Gradientenabstieg, sondern das Mini-Batch Learning zum Trainieren des neuronalen Netzes für die Klassifizierung der handgeschriebenen Ziffern einsetzen. Als es um den

stochastischen Gradientenabstieg ging, hatten wir ein Online Learning implementiert. Dabei wird der Gradient bei der Aktualisierung der Gewichtungen jeweils anhand eines einzelnen Trainingsobjekts ( $k = 1$ ) berechnet. Hierbei handelt es sich zwar um einen stochastischen Ansatz, dieser führt allerdings oft zu sehr exakten Lösungen und konvergiert viel schneller als ein normaler Gradientenabstieg. Das Mini-Batch Learning ist eine besondere Form eines stochastischen Gradientenabstiegs, bei der der Gradient anhand einer Teilmenge  $k$  der  $n$  Trainingsobjekte ( $1 < k < n$ ) berechnet wird. Gegenüber dem Online Learning hat das Mini-Batch Learning den Vorteil, dass wir die vektorisierten Implementierungen verwenden können, um die Effizienz der Berechnungen zu verbessern – können die Gewichtungen also viel schneller aktualisieren als mit einem normalen Gradientenabstieg. Sie können sich das Mini-Batch Learning in etwa so vorstellen wie die Vorhersage eines Wahlergebnisses durch die Befragung einer repräsentativen Wählergruppe statt der gesamten Bevölkerung.

Außerdem sind mehrschichtige neuronale Netze viel schwieriger trainierbar als einfache Algorithmen wie Adaline, die logistische Regression oder Support Vector Machines. Bei mehrschichtigen neuronalen Netzen müssen wir typischerweise Hunderte, Tausende oder sogar Milliarden von Gewichtungen optimieren. Leider ist die Oberfläche der Ausgabefunktion uneben – der Optimierungsalgorithmus kann leicht in lokalen Minima stecken bleiben (siehe Abbildung).



Beachten Sie hier, dass diese Darstellung extrem vereinfacht ist, denn das neuronale Netz besitzt eine Vielzahl von Dimensionen, sie ermöglicht es jedoch, die Oberfläche der Straffunktion zu visualisieren. Die Abbildung zeigt lediglich die Form der Straffunktion einer einzelnen Gewichtung entlang der x-Achse. Entscheidend ist hier, dass der Algorithmus nicht in lokalen Minima gefangen bleibt. Durch eine Erhöhung der Lernrate gelingt es zwar leichter, diesen lokalen Minima zu entkommen, andererseits erhöhen wir jedoch die Wahrscheinlichkeit, über das globale Minimum hinauszuschießen, wenn die Lernrate zu groß ist. Da wir die Gewich-

tungen zufällig initialisieren, ist die anfängliche Lösung der Optimierungsaufgabe normalerweise völlig verkehrt.

## 12.5 Abschließende Bemerkungen zur Implementierung neuronaler Netze

Sie stellen sich nun vielleicht die Frage, warum man sich mit all dieser Theorie herumplagen sollte, nur um ein einfaches mehrschichtiges neuronales Netz zur Erkennung handgeschriebener Ziffern zu implementieren. Stattdessen könnte man doch auch eine der quelloffenen Python-Bibliotheken für Machine Learning nutzen. Tatsächlich werden wir in den folgenden Kapiteln komplexere Modelle neuronaler Netze vorstellen, die wir mithilfe der TensorFlow-Bibliothek (<https://www.tensorflow.org>) trainieren werden. Auch wenn die vollständige Implementierung in diesem Kapitel auf den ersten Blick etwas mühsam erscheint, war sie doch eine gute Übung, um die Grundlagen des Backpropagation-Verfahrens und neuronaler Netze zu verstehen. Ein elementares Verständnis von Algorithmen ist für die korrekte und erfolgreiche Anwendung von Verfahren des Machine Learnings unverzichtbar.

Nun kennen Sie die Funktionsweise neuronaler Feedforward-Netze und sind dafür gewappnet, ausgeklügeltere, auf NumPy beruhende Python-Bibliotheken wie TensorFlow und Keras (<http://keras.io>) zu erkunden, die ein effizienteres Entwickeln neuronaler Netze ermöglichen, wie Sie in Kapitel 13 sehen werden. TensorFlow hat seit der Veröffentlichung im November 2015 bei auf dem Gebiet des Machine Learnings tätigen Forschern sehr an Popularität gewonnen. Sie setzen die Bibliothek zur Entwicklung tiefer neuronaler Netze ein, weil sie es ermöglicht, mathematische Ausdrücke für die Berechnung mehrdimensionaler Arrays auf *Grafikprozessoren (GPUs, Graphical Processing Units)* zu optimieren. TensorFlow kann eher als maschinennahe Deep-Learning-Bibliothek betrachtet werden, eine vereinfachende API wie Keras hingegen wurde entwickelt, um das Erstellen gängiger Deep-Learning-Modelle komfortabler zu gestalten, wie Sie in Kapitel 13 sehen werden.

## 12.6 Zusammenfassung

In diesem Kapitel haben Sie die wichtigsten Konzepte mehrschichtiger künstlicher neuronaler Netze kennengelernt, die derzeit zu den am intensivsten erforschten Themen im Bereich des Machine Learnings gehören. In Kapitel 2 haben wir mit einer einfachen einschichtigen neuronalen Netzstruktur angefangen, und jetzt haben wir mehrere Neuronen zu einer leistungsfähigen neuronalen Netzarchitektur verknüpft, die komplexe Aufgaben wie die Erkennung handgeschriebener Ziffern bewerkstelligen kann. Wir haben uns mit dem Backpropagation-Verfahren

befasst, einem der Bausteine vieler neuronaler Netzmodelle, die beim Deep Learning Verwendung finden. Durch die in diesem Kapitel erworbenen Kenntnisse des Backpropagation-Algorithmus sind wir gut gerüstet, komplexere neuronale Netze zu erkunden. In den verbleibenden Kapiteln werden wir TensorFlow vorstellen, eine Open-Source-Bibliothek für Deep Learning, die ein effizienteres Implementieren und Trainieren mehrschichtiger neuronaler Netze ermöglicht.



# Parallelisierung des Trainings neuronaler Netze mit TensorFlow

In diesem Kapitel lassen wir die mathematischen Grundlagen des Machine Learnings und des Deep Learnings hinter uns und wenden uns *TensorFlow* zu. Dabei handelt es sich um die wohl verbreitetste verfügbare Deep-Learning-Bibliothek, die es ermöglicht, neuronale Netze erheblich effizienter als die bisher vorgestellten NumPy-Beispiele zu implementieren. In diesem Kapitel werden wir TensorFlow einsetzen und zeigen, wie sich damit signifikante Verbesserungen der Trainingsleistung erzielen lassen.

Dieses Kapitel ist der Anfang einer neuen Etappe auf der Tour durch das Trainieren von Aufgabenstellungen des Machine Learnings und des Deep Learnings, und wir werden folgende Themen betrachten:

- Wie TensorFlow die Trainingsleistung verbessert
- Für das Machine Learning optimierten Code mit TensorFlow entwickeln
- Verwenden der TensorFlow-API zur Entwicklung künstlicher neuronaler Netze
- Auswahl der Aktivierungsfunktionen künstlicher neuronaler Netze
- Verwenden der Deep-Learning-Bibliothek Keras für schnelle und einfache Experimente

## 13.1 TensorFlow und Trainingsleistung

TensorFlow kann das Trainieren von Lernsystemen beträchtlich beschleunigen. Um besser zu verstehen, wie das funktioniert, wollen wir zunächst einige der Schwierigkeiten im Zusammenhang mit der Ausführung aufwendiger Berechnungen auf der Hardware betrachten.

Natürlich wurden im Bereich der Prozessorperformance im Laufe der vergangenen Jahre stetige und weitreichende Fortschritte erzielt, was es uns ermöglicht, leistungsfähigere und komplexere Lernsysteme zu trainieren, um so die Vorhersagekraft der Lernmodelle zu verbessern. Heutzutage verfügen selbst die preiswertesten Rechner über Prozessoren mit mehreren Kernen.

In den vorangegangenen Kapiteln haben Sie erfahren, dass viele Funktionen von scikit-learn die Möglichkeit bieten, Berechnungen auf mehrere Prozessorkerne zu verteilen. Allerdings ist Python durch den *Global Interpreter Lock (GIL)* standard-

mäßig auf den Einsatz nur eines Kerns beschränkt. Wir machen uns zwar die **multiprocessing**-Bibliothek zunutze, um Berechnungen auf mehrere Kerne zu verteilen, müssen dabei aber berücksichtigen, dass selbst bessere Desktoprechner selten über mehr als 8 oder 16 Prozessorkerne verfügen.

Im letzten Kapitel mussten wir bei der Implementierung des denkbar einfachen mehrschichtigen Perzeptrons mit einer aus nur 100 Einheiten bestehenden verdeckten Schicht bereits rund 80.000 Gewichtungsparameter optimieren ( $[784 * 100 + 100] + [100 * 10] + 10 = 79.510$ ), um ein Modell für eine sehr einfache Bilderkennung zu trainieren. Die Bilder der MNIST-Datensammlung sind ziemlich klein (28 x 28 Pixel) – und nun können wir uns ausmalen, inwieweit die Anzahl der Parameter explosionsartig wächst, sobald wir weitere verdeckte Schichten hinzufügen oder Bilder mit höherer Auflösung verwenden möchten.

Derartige Aufgabenstellungen sind mit nur einem Kern schnell nicht mehr machbar. Wie also können wir solche Aufgaben effektiver in Angriff nehmen?

Die naheliegende Antwort lautet, GPUs (Grafikprozessoren) zu nutzen, denn GPUs sind echte Arbeitspferde. Stellen Sie sich die Grafikkarte einfach als einen kleinen Rechnerverbund innerhalb des Computers vor. Ein weiterer Vorteil besteht darin, dass Grafikkarten im Vergleich zu aktuellen CPUs relativ preiswert sind, wie die folgende Übersicht zeigt:

Spezifikation	Intel® Core™ i7-6900K Processor Extreme Ed.	NVIDIA GeForce® GTX™ 1080 Ti
Taktfrequenz	3,2 GHz	< 1,5 GHz
Kerne	8	3.584
Datendurchsatz	64 GB/s	484 GB/s
Fließkommazahlenoperationen	409 GFLOPS	11.300 GFLOPS
Preis	ca. 1.000 USD	ca. 700 USD

Die Quellen für diese Angaben (Stand August 2017) sind im Internet zu finden:

- <https://www.intel.com/content/www/us/en/products/processors/core/x-series/i7-6900k.html>
- <https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080-ti/>

Für 70 Prozent des Preises einer modernen CPU ist eine GPU erhältlich, die fast 450 Mal so viele Kerne besitzt und mehr als 27 Mal mehr Fließkommaoperationen pro Sekunde ausführen kann. Was sollte uns also davon abhalten, GPUs für unsere Aufgabenstellungen zu verwenden?

Das Problem besteht darin, dass das Schreiben von Code für die GPU nicht so trivial ist wie das Ausführen von Python-Code im Interpreter. Nun gibt es zwar spe-

zielle Pakete wie CUDA oder OpenCL, die die Codeerstellung für die GPU unterstützen, allerdings bieten sie nicht gerade die komfortabelsten Entwicklungs-umgebungen für die Implementierung und das Ausführen von Lernalgorithmen. Die gute Nachricht lautet: Genau diesen Zweck erfüllt TensorFlow!

### 13.1.1 Was genau ist TensorFlow?

TensorFlow ist eine skalierbare und plattformübergreifende Programmierschnittstelle zum Implementieren und Ausführen von Algorithmen des Machine Learnings, die zudem komfortable Wrapper-Funktionen für Deep Learning mitbringt.

TensorFlow wurde von einem Team aus Forschern und Ingenieuren bei Google Brain entwickelt. Die Hauptentwicklung liegt zwar in den Händen dieses Teams, in die Entwicklung fließen jedoch auch viele Beiträge der Open-Source-Community ein. TensorFlow wurde ursprünglich nur für den internen Gebrauch bei Google entwickelt, wurde letztendlich aber im November 2015 unter einer großzügigen Open-Source-Lizenz veröffentlicht.

Zur Verbesserung der Trainingsleistung von Lernmodellen gestattet TensorFlow die Ausführung sowohl auf CPUs als auch auf GPUs. Den größten Leistungsschub erzielt man allerdings mit der Verwendung von GPUs. TensorFlow unterstützt offiziell bislang nur CUDA-fähige GPUs; die Unterstützung für OpenCL-fähige Geräte ist noch experimentell. Allerdings wird OpenCL wahrscheinlich schon in naher Zukunft ebenfalls offiziell unterstützt.

TensorFlow bietet Schnittstellen für eine ganze Reihe verschiedener Programmiersprachen. Wir Python-User haben das Glück, das Tensorflows Python-API derzeit die vollständigste ist, wodurch viele auf dem Gebiet des Machine Learnings und Deep Learnings tätige Forscher und Entwickler angezogen werden. TensorFlow besitzt zudem eine offizielle API für C++.

Die APIs für andere Programmiersprachen, wie Java, Haskell, Node.js und Go sind noch nicht ausgereift, aber die Open-Source-Community und die TensorFlow-Entwickler verbessern sie fortlaufend. TensorFlow-Berechnungen beruhen auf sogenannten gerichteten Graphen, die den Datenfluss repräsentieren. Das Konstruieren eines gerichteten Graphen hört sich vielleicht kompliziert an, aber Tensorflows High-level-API hat diesen Vorgang stark vereinfacht.

### 13.1.2 TensorFlow erlernen

Zunächst einmal befassen wir uns mit der Low-level-API von TensorFlow. Die Low-level-Implementierung eines Modells mag anfangs ein wenig mühsam erscheinen, bietet uns Programmierern bei der Entwicklung komplexer Lernmodelle beim Kombinieren der grundlegenden Operationen jedoch größere Flexibilität. Seit der TensorFlow-Version 1.1.0 gibt es eine High-level-API, die auf der

Low-level-API (wie z.B. den Schätzer-APIs) aufbaut und es ermöglicht, Prototypen eines Modells sehr viel schneller zu entwickeln.

Nach dem Kennenlernen der Low-level-API betrachten wir zwei High-level-APIs, nämlich TensorFlow-Layers und Keras. Zunächst einmal unternehmen wir jedoch die ersten Schritte mit Tensorflows Low-level-API und machen uns mit der Funktionsweise vertraut.

### 13.1.3 Erste Schritte mit TensorFlow

In diesem Abschnitt werden wir die ersten Schritte mit TensorFlow unternehmen. Je nachdem, wie Ihr System eingerichtet ist, können Sie normalerweise einfach den `pip`-Befehl verwenden, um TensorFlow über PyPI zu installieren. Geben Sie im Terminal folgenden Befehl ein:

```
pip install tensorflow
```

Falls Sie GPUs verwenden möchten, müssen auch das CUDA-Toolkit sowie die cuDNN-Bibliothek von NVIDIA installiert werden. Sie können TensorFlow mit GPU-Unterstützung folgendermaßen installieren:

```
pip install tensorflow-gpu
```

TensorFlow wird fortlaufend weiterentwickelt, daher wird alle paar Monate eine neue Version mit wichtigen Änderungen veröffentlicht. Derzeit ist 1.3.0 die aktuelle Version von TensorFlow. Sie können die Versionsnummer im Terminal wie folgt überprüfen:

```
python -c 'import tensorflow as tf; print(tf.__version__)'
```

#### Tipp

Falls Sie bei der Installation auf Schwierigkeiten stoßen, sollten Sie die system- und plattformabhängigen Hinweise beachten, die auf der TensorFlow Website unter <http://www.tensorflow.org/install> angegeben sind. Sämtlicher Code in diesem Kapitel ist auch auf der CPU ausführbar – die Verwendung der GPU ist hier optional, aber durchaus empfehlenswert, wenn Sie sich die Vorzüge von TensorFlow zunutze machen möchten. Wenn Sie eine Grafikkarte verwenden, sollten Sie die Dokumentation lesen, um alles korrekt einzurichten. Vielleicht finden Sie auch die unter [https://sebastianraschka.com/pdf/books/dlb/appendix\\_h\\_cloud-computing.pdf](https://sebastianraschka.com/pdf/books/dlb/appendix_h_cloud-computing.pdf) verfügbare Anleitung nützlich, die erläutert, wie die Grafiktreiber von NVIDIA, CUDA und cuDNN unter Ubuntu installiert werden. Das ist zwar zum Ausführen von TensorFlow auf einer GPU nicht zwingend erforderlich, aber durchaus empfehlenswert.

TensorFlow beruht auf einem aus mehreren Knoten bestehenden Berechnungsgraphen. Dabei repräsentiert jeder Knoten eine Operation, die keine oder aber mehrere Ein- und Ausgaben verwenden. Die Werte, die entlang der Kanten des Berechnungsgraphen fließen, werden als *Tensoren* bezeichnet. Diese können Sie sich als eine Verallgemeinerung von Skalaren, Vektoren, Matrizen usw. vorstellen. Oder genauer: Ein Skalar ist ein Tensor des Rangs 0, ein Vektor ist ein Tensor des Rangs 1, eine Matrix ist ein Tensor des Rangs 2, und in der dritten Dimension »aufgestapelte« Matrizen stellen einen Tensor des Rangs 3 dar.

Sobald der Berechnungsgraph konstruiert ist, kann er in einer TensorFlow-Sitzung gestartet werden, um die verschiedenen Knoten des Graphen auszuführen. In Kapitel 14 (*Die Funktionsweise von TensorFlow im Detail*) werden wir die zur Konstruktion eines Berechnungsgraphen erforderlichen Schritte und den Start in einer TensorFlow-Sitzung eingehender betrachten.

Als Aufwärmübung fangen wir mit der Verwendung einfacher Skalare in TensorFlow an und berechnen die Nettoeingabe  $z$  eines Punkts  $x$  mit der Gewichtung  $w$  und dem Bias  $b$  in einer eindimensionalen Datenmenge:

$$z = w \times x + b$$

Der folgende Code zeigt die Implementierung dieser Gleichung mit der Low-level API von TensorFlow:

```
import tensorflow as tf
## Konstruktion eines Graphen
g = tf.Graph()
with g.as_default():
    x = tf.placeholder(dtype=tf.float32,
                       shape=(None), name='x')
    w = tf.Variable(2.0, name='weight')
    b = tf.Variable(0.7, name='bias')
    z = w*x + b

    init = tf.global_variables_initializer()
## Sitzung eröffnen und den Graphen g übergeben
with tf.Session(graph=g) as sess:
    ## w und b initialisieren:
    sess.run(init)
    ## Auswertung von z:
    for t in [1.0, 0.6, -1.8]:
        print('x=%4.1f --> z=%4.1f'%(t, sess.run(z, feed_dict={x:t})))
```

Nach der Ausführung des Codes sollte die folgende Ausgabe angezeigt werden:

```
x= 1.0 --> z= 2.7
x= 0.6 --> z= 1.9
x=-1.8 --> z=-2.9
```

Das war doch ziemlich schnörkellos, oder? Bei der Entwicklung eines Modells mit der Low-level-API von TensorFlow müssen wir Platzhalter für die Eingabedaten ( $x$  und  $y$ , in manchen Fällen auch weitere abstimmbare Parameter) und anschließend die Gewichtungsmatrizen definieren sowie das Modell zur Verknüpfung von Eingabe und Ausgabe einrichten. Falls es sich um eine Optimierungsaufgabe handelt, sollten wir die Straffunktion definieren und herausfinden, welcher Algorithmus verwendet werden sollte. TensorFlow erstellt dann einen Graphen, der alle von uns definierten Symbole als Knoten enthält.

Nun erstellen wir einen Platzhalter für  $x$  mit `shape=(None)`. Auf diese Weise können wir die Werte elementweise und in Form einer Stapelverarbeitung übergeben:

```
>>> with tf.Session(graph=g) as sess:
...     sess.run(init)
...     print(sess.run(z, feed_dict={x:[1., 2., 3.]}))
[ 2.70000005  4.69999981  6.69999981]
```

## Hinweis

In diesem Kapitel lassen wir Pythons Eingabeaufforderung hin und wieder weg, damit bei längeren Codebeispielen keine überflüssigen Zeilenumbrüche erfolgen, denn die Bezeichnungen der Funktionen und Methoden in TensorFlow sind zum Teil *sehr* weitschweifig.

Beachten Sie außerdem, dass die offiziellen Stilrichtlinien für TensorFlow ([https://www.tensorflow.org/community/style\\_guide](https://www.tensorflow.org/community/style_guide)) empfehlen, für die Einrückung von Code zwei Leerzeichen zu verwenden. Wir haben uns allerdings dafür entschieden, vier Leerzeichen zu verwenden, weil dies den offiziellen Stilrichtlinien für Python entspricht und in vielen Texteditoren und in den das Buch ergänzenden Jupyter-Notebooks (<https://github.com/rasbt/python-machine-learning-book-2nd-edition>) bei der Syntaxhervorhebung besser erkannt wird.

### 13.1.4 Mit Array-Strukturen arbeiten

In diesem Abschnitt erörtern wir die Verwendung der Array-Strukturen von TensorFlow. Der folgende Code erzeugt einen einfachen Tensor des Rangs 3 und der Größe  $Stapelgröße \times 2 \times 3$ , formt ihn um und berechnet mithilfe der in TensorFlow

zur Verfügung stehenden optimierten Tensorausdrücke die Spaltensummen. Da der Wert der Stapelgröße vorab nicht bekannt ist, übergeben wir dem Platzhalter  $x$  als Argument für den Parameter `shape` für die Stapelgröße den Wert `None`:

```
import tensorflow as tf
import numpy as np

g = tf.Graph()
with g.as_default():
    x = tf.placeholder(dtype=tf.float32,
                       shape=(None, 2, 3),
                       name='input_x')
    x2 = tf.reshape(x, shape=(-1, 6),
                    name='x2')
    ## Summe der Spalten berechnen
    xsum = tf.reduce_sum(x2, axis=0, name='col_sum')

    ## Mittelwerte der Spalten berechnen
    xmean = tf.reduce_mean(x2, axis=0, name='col_mean')

with tf.Session(graph=g) as sess:
    x_array = np.arange(18).reshape(3, 2, 3)
    print('Eingabe-Shape: ', x_array.shape)
    print('Umgeformt:\n',
          sess.run(x2, feed_dict={x:x_array}))
    print('Spaltensummen:\n',
          sess.run(xsum, feed_dict={x:x_array}))
    print('Spaltenmittelwerte:\n',
          sess.run(xmean, feed_dict={x:x_array}))
```

Die Ausführung des vorstehenden Codes sollte folgende Ausgabe erzeugen:

```
Eingabe-Shape: (3, 2, 3)
Umgeformt:
[[ 0.  1.  2.  3.  4.  5.]
 [ 6.  7.  8.  9. 10. 11.]
 [12. 13. 14. 15. 16. 17.]]
Spaltensummen:
[ 18.  21.  24.  27.  30.  33.]
Spaltenmittelwerte:
[ 6.  7.  8.  9. 10. 11.]
```

In diesem Beispiel haben wir drei Funktionen verwendet: `tf.reshape`, `tf.reduce_sum` und `tf.reduce_mean`. Beachten Sie, dass wir bei der Umformung mit `reshape` für die erste Dimension den Wert `-1` verwenden, weil der Wert für die

Stapelgröße unbekannt ist. Wenn man beim Umformen eines Tensors für eine bestimmte Dimension den Wert -1 angibt, wird die Größe dieser Dimension entsprechend der Gesamtgröße des Tensors und den übrigen Dimensionen berechnet, deshalb kann `tf.reshape(tensor, shape=(-1))` zum Umformen eines Tensors verwendet werden.

Probieren Sie ruhig einige der anderen TensorFlow-Funktionen aus, die in der offiziellen Dokumentation unter [https://www.TensorFlow.org/api\\_docs/python/tf](https://www.TensorFlow.org/api_docs/python/tf) beschrieben sind.

### 13.1.5 Entwicklung eines einfachen Modells mit Tensorflows Low-level-API

Nachdem wir uns nun ein wenig mit TensorFlow vertraut gemacht haben, wollen wir ein echtes Beispiel aus der Praxis betrachten und eine Regression mit der Methode der kleinsten Quadrate (KQ-Methode) implementieren. Die Grundlagen der Regressionsanalyse können Sie in Kapitel 10 (*Vorhersage stetiger Zielvariablen durch Regressionsanalyse*) nachlesen.

Als Erstes erstellen wir eine kleine eindimensionale Beispieldatenmenge, die aus 10 Trainingsobjekten besteht:

```
>>> import tensorflow as tf
>>> import numpy as np
>>>
>>> X_train = np.arange(10).reshape((10, 1))
>>> y_train = np.array([1.0, 1.3, 3.1,
...                     2.0, 5.0, 6.3,
...                     6.6, 7.4, 8.0,
...                     9.0])
```

Mit dieser Datenmenge soll ein lineares Regressionsmodell darauf trainiert werden, anhand der Eingabe  $x$  eine Vorhersage für die Ausgabe  $y$  zu treffen. Wir implementieren dieses Modell in Form einer `TfLinreg` genannten Klasse. Dafür benötigen wir zwei Platzhalter – einen für die Eingabe  $x$  und einen zweiten  $y$ , um das Modell mit Daten zu füttern. Außerdem müssen wir die trainierbaren Variablen definieren, die Gewichtungen  $w$  und das Bias  $b$ .

Anschließend können wir das lineare Regressionsmodell als  $z = w \times x + b$  definieren. Als Straffunktion wird die mittlere quadratische Abweichung (MSE, Mean Squared Error) verwendet. Die Gewichtungen werden durch das Gradientenabstiegsverfahren ermittelt. Hier der Code:

```
class TfLinreg(object):
    def __init__(self, x_dim, learning_rate=0.01,
                 random_seed=None):
```

```
self.x_dim = x_dim
self.learning_rate = learning_rate
self.g = tf.Graph()
## Modell einrichten
with self.g.as_default():
    ## Zufallszahlengenerator initialisieren
    tf.set_random_seed(random_seed)
    self.build()
    ## Initialisierer erstellen
    self.init_op = tf.global_variables_initializer()
def build(self):
    ## Platzhalter für Eingaben definieren
    self.X = tf.placeholder(dtype=tf.float32,
                           shape=(None, self.x_dim),
                           name='x_input')
    self.y = tf.placeholder(dtype=tf.float32,
                           shape=(None),
                           name='y_input')
    print(self.X)
    print(self.y)
    ## Gewichtungsmatrix und Bias-Vektor definieren
    w = tf.Variable(tf.zeros(shape=(1)),
                   name='weight')
    b = tf.Variable(tf.zeros(shape=(1)),
                   name="bias")
    print(w)
    print(b)
    self.z_net = tf.squeeze(w*self.X + b,
                           name='z_net')
    print(self.z_net)
    sqr_errors = tf.square(self.y - self.z_net,
                           name='sqr_errors')
    print(sqr_errors)
    self.mean_cost = tf.reduce_mean(sqr_errors,
                                   name='mean_cost')
    optimizer = tf.train.GradientDescentOptimizer(
        learning_rate=self.learning_rate,
        name='GradientDescent')
    self.optimizer = optimizer.minimize(self.mean_cost)
```

Nun ist die Klasse zum Erstellen des Modells definiert. Wir erzeugen eine Instanz des Modells und nennen sie `lrm`:

```
>>> lrm = TfLinreg(x_dim=X_train.shape[1],
...                  learning_rate=0.01)
```

Die `print`-Funktionen in der `build`-Methode zeigen die Namen und die Formen (`shapes`) der sechs Knoten `X`, `y`, `w`, `b`, `z_net` und `sqr_errors` des Graphen an.

Diese `print`-Anweisungen sind optional, aber die Überprüfung der Formen von Variablen kann beim Debuggen komplexer Modelle sehr hilfreich sein. Beim Erstellen des Modells werden die folgenden Zeilen ausgegeben:

```
Tensor("x_input:0", shape=(?, 1), dtype=float32)
Tensor("y_input:0", dtype=float32)
<tf.Variable 'weight:0' shape=(1,) dtype=float32_ref>
<tf.Variable 'bias:0' shape=(1,) dtype=float32_ref>
Tensor("z_net:0", dtype=float32)
Tensor("sqr_errors:0", dtype=float32)
```

Der nächste Schritt ist die Implementierung einer Trainingsfunktion zur Ermittlung der Gewichtungen des linearen Regressionsmodells. Beachten Sie hier, das  $b$  die Bias-Einheit (der  $y$ -Achsenabschnitt bei  $x=0$ ) ist.

Für das Training verwenden wir eine separate Funktion, die eine TensorFlow-Sitzung, eine Instanz des Modells, Trainingsdaten und die Anzahl der Epochen als Argumente entgegennimmt. In dieser Funktion initialisieren wir mit der im Modell definierten Operation `init_op` zunächst die Variablen. Anschließend startet die Iteration, in der die `optimizer`-Operation des Modells aufgerufen wird, während es mit Trainingsdaten gefüttert wird. Diese Funktion liefert als Nebenprodukt eine Liste mit Werten der Straffunktion zurück.

```
def train_linreg(sess, model, X_train, y_train,
                 num_epochs=10):
    ## Initialisierung aller Variablen: W und b
    sess.run(model.init_op)

    training_costs = []
    for i in range(num_epochs):
        _, cost = sess.run([model.optimizer, model.mean_cost],
                           feed_dict={model.X:X_train,
                                      model.y:y_train})
        training_costs.append(cost)
    return training_costs
```

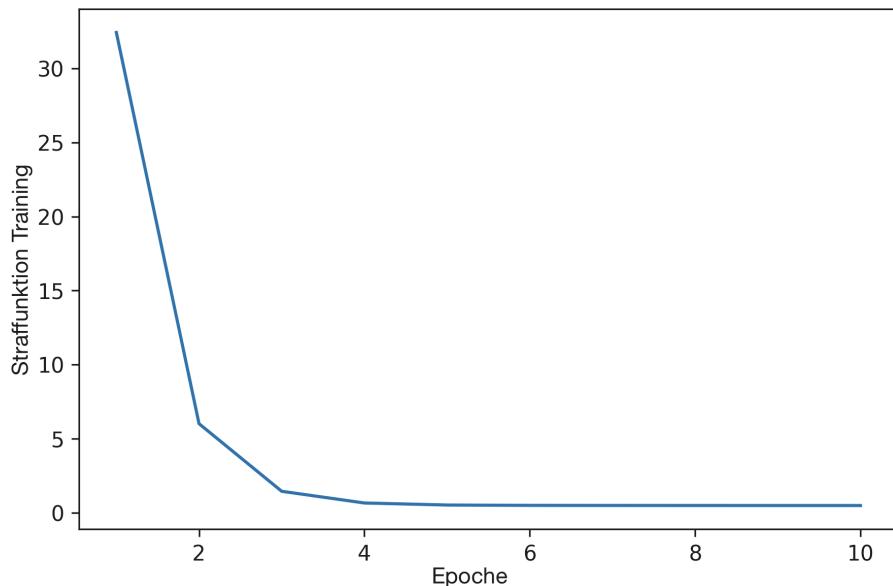
Nun können wir eine neue TensorFlow-Sitzung eröffnen, die den `lrmodel.g`-Graphen startet und alle erforderlichen Parameter an die `train_linreg`-Funktion übergibt, um das Training durchzuführen:

```
>>> sess = tf.Session(graph=lrmodel.g)
>>> training_costs = train_linreg(sess, lrmodel,
                                  X_train, y_train)
```

Jetzt visualisieren wir die Werte der Straffunktion nach diesen 10 Epochen, um zu überprüfen, ob das Modell konvergiert:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(range(1,len(training_costs) + 1),
             training_costs)
>>> plt.tight_layout()
>>> plt.xlabel('Epoche')
>>> plt.ylabel('Strafffunktion Training')
>>> plt.show()
```

Wie in der folgenden Abbildung zu erkennen ist, konvergiert dieses einfache Modell schon nach einigen wenigen Epochen.



So weit, so gut. In Anbetracht der Strafffunktion haben wir offenbar ein funktionierendes Regressionsmodell für diese spezielle Datenmenge erstellt. Jetzt kompilieren wir eine neue Funktion, um anhand der Eingabemerkmale Vorhersagen zu treffen. Dafür benötigen wir die TensorFlow-Sitzung, das Modell und die Testdatenmenge:

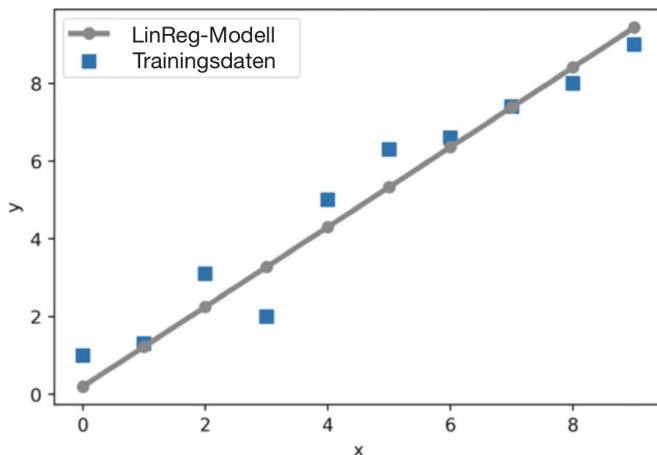
```
def predict_linreg(sess, model, X_test):
    y_pred = sess.run(model.z_net,
                      feed_dict={model.X:X_test})
    return y_pred
```

Die Implementierung einer Vorhersagefunktion ist ziemlich unkompliziert: Es genügt, die im Graphen definierte `z_net`-Funktion auszuführen, um die Vorher-

sagewerte zu berechnen. Als Nächstes geben wir die Anpassung der linearen Regression an die Trainingsdaten aus:

```
>>> plt.scatter(X_train, y_train,
...                 marker='s', s=50,
...                 label='Trainingsdaten')
>>> plt.plot(range(X_train.shape[0]),
...             predict_linreg(sess, lrm, X_train),
...             color='gray', marker='o',
...             markersize=6, linewidth=3,
...             label='LinReg-Modell')
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.legend()
>>> plt.tight_layout()
>>> plt.show()
```

Das resultierende Diagramm zeigt, dass unser Modell gut an die Trainingsdatenpunkte angepasst ist:



## 13.2 Training neuronaler Netze mit TensorFlows High-level-APIs

In diesem Abschnitt befassen wir uns mit TensorFlows High-level-APIs, der Layers-API (`tensorflow.layers` oder `tf.layers`) und der Keras-API (`tensorflow.contrib.keras`).

Keras kann auch als separates Paket installiert werden und unterstützt sowohl Theano als auch TensorFlow als Backend. Weitere Informationen finden Sie auf der offiziellen Keras-Website (<https://keras.io/>).

Nach der Veröffentlichung von TensorFlow 1.1.0 wurde Keras dem TensorFlow-Submodul `contrib` hinzugefügt. Sehr wahrscheinlich wird Keras schon bald aus dem experimentellen `contrib`-Submodul entfernt und zu einem der regulären TensorFlow-Submodule werden.

### 13.2.1 Entwicklung mehrschichtiger neuronaler Netze mit TensorFlows Layers-API

Nun implementieren wir ein mehrschichtiges Perzeptron zum Klassifizieren der handgeschriebenen Ziffern der im letzten Kapitel vorgestellten MNIST-Datensammlung, um zu betrachten, wie das Training eines neuronalen Netzes mit TensorFlows High-level-API `tensorflow.layers` (`tf.layers`) funktioniert. Die MNIST-Datensammlung ist unter <http://yann.lecun.com/exdb/mnist/> öffentlich verfügbar und besteht aus den folgenden vier Teilen:

- *Trainingsdatenmenge Bilder: train-images-idx3-ubyte.gz (9,5 MB)*
- *Trainingsdatenmenge Bezeichnungen: train-labels-idx1-ubyte.gz (32 KB)*
- *Testdatenmenge Bilder: t10k-images-idx3-ubyte.gz (1,6 MB)*
- *Testdatenmenge Bezeichnungen: t10k-labels-idx1-ubyte.gz (8 KB)*

#### Tipp

TensorFlow stellt diese Datenmenge ebenfalls zur Verfügung:

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist \
import input_data
```

Wir verwenden die MNIST-Datensammlung jedoch als externe Datenmenge, um die bei der Vorverarbeitung von Daten erforderlichen Schritte zu zeigen. Auf diese Weise erfahren Sie, wie Sie bei Ihren eigenen Daten vorgehen müssen.

Nach dem Herunterladen und Entpacken des Archivs sollten die Daten im Ordner `mnist` im aktuellen Arbeitsverzeichnis gespeichert werden, damit wir sowohl die Trainings- als auch die Testdatenmenge mit der in Kapitel 12 implementierten Funktion `load_mnist(path, kind)` einlesen können.

Die Datenmenge wird folgendermaßen geladen:

```
>>> ## Daten einlesen
>>> X_train, y_train = load_mnist('./mnist/', kind='train')
>>> print('Zeilen: %d, Spalten: %d' %(X_train.shape[0],
...                                         X_train.shape[1]))
Zeilen: 60000, Spalten: 784
>>> X_test, y_test = load_mnist('./mnist/', kind='t10k')
```

```
>>> print('Zeilen: %d, Spalten: %d' %(X_test.shape[0],
...                                         X_test.shape[1]))
Rows: 10000, Columns: 784
>>> ## Zentrierung um Mittelwert und Normierung:
>>> mean_vals = np.mean(X_train, axis=0)
>>> std_val = np.std(X_train)
>>>
>>> X_train_centered = (X_train - mean_vals)/std_val
>>> X_test_centered = (X_test - mean_vals)/std_val
>>>
>>> del X_train, X_test
>>>
>>> print(X_train_centered.shape, y_train.shape)
(60000, 784) (60000,)
>>> print(X_test_centered.shape, y_test.shape)
(10000, 784) (10000,)
```

Nun können wir das Modell einrichten. Als Erstes erstellen wir zwei Platzhalter namens `tf_x` und `tf_y`. Anschließend verwenden wir wie in Kapitel 12 ein mehrschichtiges Perzepron, das jedoch aus drei vollständig verknüpften Schichten besteht.

Allerdings ersetzen wir die logistischen Einheiten der verdeckten Schicht durch Tangens-hyperbolicus-Aktivierungsfunktionen (`tanh`), die logistische Funktion in der Ausgabeschicht durch `softmax` und fügen eine weitere verdeckte Schicht hinzu.

### Hinweis

`tanh` und `softmax` sind neue Aktivierungsfunktionen. Mehr dazu in einem der nächsten Abschnitte (*Auswahl der Aktivierungsfunktionen mehrschichtiger neuronaler Netze*).

```
import tensorflow as tf
n_features = X_train_centered.shape[1]
n_classes = 10
random_seed = 123
np.random.seed(random_seed)
g = tf.Graph()
with g.as_default():
    tf.set_random_seed(random_seed)
    tf_x = tf.placeholder(dtype=tf.float32,
                          shape=(None, n_features),
                          name='tf_x')
    tf_y = tf.placeholder(dtype=tf.int32,
```

```

        shape=None, name='tf_y')
y_onehot = tf.one_hot(indices=tf_y, depth=n_classes)
h1 = tf.layers.dense(inputs=tf_x, units=50,
                     activation=tf.tanh,
                     name='layer1')
h2 = tf.layers.dense(inputs=h1, units=50,
                     activation=tf.tanh,
                     name='layer2')
logits = tf.layers.dense(inputs=h2,
                         units=10,
                         activation=None,
                         name='layer3')
predictions = {
    'classes' : tf.argmax(logits, axis=1,
                          name='predicted_classes'),
    'probabilities' : tf.nn.softmax(logits,
                                    name='softmax_tensor')
}

```

Als Nächstes definieren wir die Straffunktion und fügen einen Operator zum Initialisieren der Modellvariablen sowie einen Optimierungsoperator hinzu:

```

## Straffunktion und Optimierer definieren:
with g.as_default():
    cost = tf.losses.softmax_cross_entropy(
        onehot_labels=y_onehot, logits=logits)
    optimizer = tf.train.GradientDescentOptimizer(
        learning_rate=0.001)
    train_op = optimizer.minimize(loss=cost)
    init_op = tf.global_variables_initializer()

```

Bevor wir mit dem Training anfangen, benötigen wir eine Möglichkeit, Daten für die Stapelverarbeitung zu erzeugen. Zu diesem Zweck implementieren wir die folgende Funktion, die einen Generator zurückgibt.

```

def create_batch_generator(X, y, batch_size=128,
                           shuffle=False):
    X_copy = np.array(X)
    y_copy = np.array(y)

    if shuffle:
        data = np.column_stack((X_copy, y_copy))
        np.random.shuffle(data)
        X_copy = data[:, :-1]
        y_copy = data[:, -1].astype(int)

```

```

for i in range(0, X.shape[0], batch_size):
    yield (X_copy[i:i+batch_size, :],
           y_copy[i:i+batch_size])

```

Nun eröffnen wir eine neue TensorFlow-Sitzung, initialisieren alle Variablen des Netzes und trainieren es. Außerdem zeigen wir nach jeder Epoche den Durchschnittswert der Straffunktion an, um den Lernvorgang beurteilen zu können.

```

>>> ## Sitzung zum Starten des Graphen erzeugen
>>> sess = tf.Session(graph=g)
>>> ## Operator zur Variableninitialisierung ausführen
>>> sess.run(init_op)
>>>
>>> ## 50 Trainingsepochen:
>>> for epoch in range(50):
...     training_costs = []
...     batch_generator = create_batch_generator(
...         X_train_centered, y_train,
...         batch_size=64)
...     for batch_X, batch_y in batch_generator:
...         ## Dictionary für die Datenübergabe vorbereiten:
...         feed = {tf_x:batch_X, tf_y:batch_y}
...         _, batch_cost = sess.run([train_op, cost],
...                                 feed_dict=feed)
...         training_costs.append(batch_cost)
...     print(' -- Epoche %2d   '
...           'Durchschnittswert der Straffunktion: %.4f' % (
...               epoch+1, np.mean(training_costs)))
... )

-- Epoche  1 Durchschnittswert der Straffunktion: 1.5573
-- Epoche  2 Durchschnittswert der Straffunktion: 1.2532
-- Epoche  3 Durchschnittswert der Straffunktion: 1.0854
-- Epoche  4 Durchschnittswert der Straffunktion: 0.9738
...
-- Epoche 49 Durchschnittswert der Straffunktion: 0.3527
-- Epoche 50 Durchschnittswert der Straffunktion: 0.3498

```

Das Training kann einige Minuten Zeit in Anspruch nehmen. Schließlich können wir das trainierte Modell verwenden, um Vorhersagen für die Testdatenmenge zu treffen:

```

>>> ## Vorhersagen für die Testdatenmenge treffen:
>>> feed = {tf_x : X_test_centered}
>>> y_pred = sess.run(predictions['classes'],

```

```
...           feed_dict=feed)
>>> print('Korrektklassifizierungsrate Test: %.2f%%' % (
...     100*np.sum(y_pred == y_test)/y_test.shape[0]))
Korrektklassifizierungsrate Test: 93.89%
```

Wie Sie sehen, kann man mit der High-level-API schnell und einfach ein Modell entwickeln und testen. Deshalb ist die High-level-API bestens zum Entwickeln von Prototypen und zur schnellen Überprüfung der Ergebnisse geeignet.

Als Nächstes werden wir ein ähnliches Klassifizierungsmodell für die MNIST-Datensammlung mit Keras entwickeln, einer weiteren High-level-API für TensorFlow.

### 13.2.2 Entwicklung eines mehrschichtigen neuronalen Netzes mit Keras

Die Entwicklung von Keras begann Anfang 2015. Inzwischen hat es sich zu einer der verbreitetsten und am meisten genutzten auf Theano und TensorFlow aufbauenden Bibliotheken entwickelt.

Ähnlich wie TensorFlow ermöglicht Keras die Nutzung der GPUs zum Beschleunigen des Trainings neuronaler Netze. Zu den bekanntesten Features zählt die sehr intuitive und benutzerfreundliche API, die es ermöglicht, neuronale Netze mit nur einigen wenigen Zeilen Code zu implementieren.

Keras wurde zunächst als eigenständige API veröffentlicht, die auf Theano aufbaute. Später wurde die Unterstützung für TensorFlow nachgereicht. Seit Version 1.1.0 ist Keras in TensorFlow integriert. Wenn Sie also die Version 1.1.0 von TensorFlow verwenden, muss Keras nicht mehr installiert werden. Weitere Informationen über Keras finden Sie auf der offiziellen Website unter <http://keras.io>.

Derzeit gehört Keras zum `contrib`-Modul, das Pakete von Dritten enthält, die als experimenteller Code betrachtet werden. In künftigen TensorFlow-Versionen könnte Keras aber sehr wohl ein eigenständiges Modul der TensorFlow-API werden. Weitere Informationen finden Sie in der Dokumentation der TensorFlow-Website unter [https://www.tensorflow.org/api\\_docs/python/tf/contrib/keras](https://www.tensorflow.org/api_docs/python/tf/contrib/keras).

#### Tipp

Möglicherweise müssen Sie den Code in den folgenden Beispielen von `import tensorflow.contrib.keras as keras` auf `import tensorflow.keras as keras` ändern, damit sie mit zukünftigen TensorFlow-Versionen funktionieren.

Auf den folgenden Seiten werden die Codebeispiele zur Verwendung von Keras schrittweise erläutert. Die Daten müssen mit den im letzten Abschnitt verwendeten Funktionen wie folgt geladen werden:

```
>>> X_train, y_train = load_mnist('./mnist/', kind='train')
>>> print('Zeilen: %d, Spalten: %d' %(X_train.shape[0],
...                                         X_train.shape[1]))
Zeilen: 60000, Spalten: 784
>>> X_test, y_test = load_mnist('./mnist/', kind='t10k')
>>> print('Zeilen: %d, Spalten: %d' %(X_test.shape[0],
...                                         X_test.shape[1]))
...                                         X_test.shape[1]))
Rows: 10000, Columns: 784
>>> ## Zentrierung um Mittelwert und Normierung:
>>> mean_vals = np.mean(X_train, axis=0)
>>> std_val = np.std(X_train)
>>>
>>> X_train_centered = (X_train - mean_vals)/std_val
>>> X_test_centered = (X_test - mean_vals)/std_val
>>>
>>> del X_train, X_test
>>>
>>> print(X_train_centered.shape, y_train.shape)
(60000, 784) (60000,)
>>> print(X_test_centered.shape, y_test.shape)
(10000, 784) (10000,)
```

Als Erstes initialisieren wir den Zufallszahlengenerator für NumPy und TensorFlow, damit wir reproduzierbare Ergebnisse erhalten:

```
>>> import tensorflow as tf
>>> import tensorflow.contrib.keras as keras
>>> np.random.seed(123)
>>> tf.set_random_seed(123)
```

Um die Trainingsdaten weiter vorzubereiten, müssen wir die Klassenbezeichnungen (Ganzzahlen von 0 bis 9) in das One-hot-Format konvertieren. Erfreulicherweise stellt Keras hierfür ein komfortables Tool bereit:

```
>>> y_train_onehot = keras.utils.to_categorical(y_train)
>>>
>>> print('Die ersten 3 Klassenbezeichnungen: ', y_train[:3])
Die ersten 3 Klassenbezeichnungen: [5 0 4]
>>> print('\nDie ersten 3 Klassenbezeichnungen (One-hot):\n',
...       y_train_onehot[:3])
Die ersten 3 Klassenbezeichnungen (One-hot):
[[ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]
 [ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.  0.  0.  0.]]
```

Nun kommen wir zum interessanten Teil und implementieren das neuronale Netz. Kurz, wir verwenden drei Schichten, von denen die beiden ersten jeweils 50 verdeckte Einheiten mit `tanh`-Aktivierungsfunktion besitzen. Die letzte Schicht enthält 10 Einheiten und verwendet `softmax`, um die Wahrscheinlichkeiten der 10 Klassen anzugeben. Keras vereinfacht diese Aufgaben sehr, wie Sie der folgenden Implementierung entnehmen können:

```
model = keras.models.Sequential()

model.add(
    keras.layers.Dense(
        units=50,
        input_dim=X_train_centered.shape[1],
        kernel_initializer='glorot_uniform',
        bias_initializer='zeros',
        activation='tanh'))

model.add(
    keras.layers.Dense(
        units=50,
        input_dim=50,
        kernel_initializer='glorot_uniform',
        bias_initializer='zeros',
        activation='tanh'))

model.add(
    keras.layers.Dense(
        units=y_train_onehot.shape[1],
        input_dim=50,
        kernel_initializer='glorot_uniform',
        bias_initializer='zeros',
        activation='softmax'))

sgd_optimizer = keras.optimizers.SGD(
    lr=0.001, decay=1e-7, momentum=.9)

model.compile(optimizer=sgd_optimizer,
              loss='categorical_crossentropy')
```

Als Erstes initialisieren wir mit der Klasse `Sequential` ein neues Modell, das ein neuronales Feedforward-Netz implementiert, dem wir beliebig viele Schichten hinzufügen können. Da wir die Eingabeschicht als erste hinzufügen, müssen wir sicherstellen, dass das Attribut `input_dim` mit der Anzahl der Merkmale (Spalten) in der Trainingsdatenmenge übereinstimmt (784 Merkmale oder Pixel in der Implementierung des neuronalen Netzes).

Außerdem müssen wir uns vergewissern, dass die Anzahl der Ausgabeeinheiten (`units`) mit der Anzahl der Eingabeeinheiten (`input_dim`) zweier aufeinanderfol-

gender Schichten übereinstimmt. Im letzten Beispiel haben wir zwei verdeckte Schichten mit jeweils 50 verdeckten Einheiten nebst jeweils einer Bias-Einheit hinzugefügt. Die Anzahl der Einheiten in der Ausgabeschicht sollte der Anzahl eindeutiger Klassenbezeichnungen entsprechen – der Anzahl der Spalten im Array der One-hot-Codierung der Klassenbezeichnungen.

### Hinweis

Beachten Sie, dass wir hier mit der Einstellung `kernel_initializer='glorot_uniform'` einen neuen Initialisierungsalgorithmus für die Gewichtungsmatrizen verwenden. Glorot-Initialisierung (die auch als Xavier-Initialisierung bezeichnet wird) ist eine stabilere Methode zur Initialisierung von tiefen neuronalen Netzen (Xavier Glorot und Yoshua Bengio, *Understanding the difficulty of training deep feedforward neural networks*, in Artificial Intelligence and Statistics, Band 9, Seiten 249–256. 2010). Die Bias-Werte werden mit null initialisiert, was gebräuchlicher ist und in Keras sogar die Voreinstellung. In Kapitel 14 werden wir ausführlicher auf dieses Verfahren zur Initialisierung der Gewichtungen eingehen.

Wir müssen zunächst einen Optimizer definieren, damit wir das Modell kompilieren können. Im letzten Beispiel hatten wir das aus dem letzten Kapitel bereits vertraute Gradientenabstiegsverfahren gewählt. Darüber hinaus können wir, wie in Kapitel 12 erörtert, den Konstanten für die Verringerung der Gewichtung und dem Trägheitsterm Werte zuweisen und Momentum Learning verwenden, um die Lernrate an die einzelnen Epochen anzupassen. Abschließend setzen wir die Straffunktion auf `categorical_crossentropy`.

Die binäre Kreuzentropie ist lediglich ein technischer Begriff für die Straffunktion der logistischen Regression, und die kategoriale Kreuzentropie (`categorical_crossentropy`) ist die Verallgemeinerung auf die Vorhersage bei der Mehrfachklassifizierung via `softmax`-Funktion, einem Verfahren, das später in diesem Kapitel noch vorgestellt wird.

Nach dem Kompilieren des Modells können wir es durch Aufruf der `fit`-Methode trainieren. Hier verwenden wir das Mini-Batch Learning mit stochastischem Gradientenabstieg mit einer Stapelgröße von jeweils 64 Trainingsobjekten. Wir trainieren das *MLP* (*Multi-Layer Perceptron*) 50 Epochen lang und können durch die Einstellung `verbose=1` die Optimierung der Straffunktion während des Trainings mitverfolgen.

Als besonders praktisch erweist sich der Parameter `validation_split`, der 10 Prozent der Trainingsdaten (in diesem Fall 6.000 Objekte) für die Validierung am Ende der einzelnen Epochen reserviert, sodass wir beobachten können, wenn es während des Trainings zu einer Überanpassung kommt:

```
>>> history = model.fit(X_train_centered, y_train_onehot,
...                      batch_size=64, epochs=50,
...                      verbose=1,
...                      validation_split=0.1)
Train on 54000 samples, validate on 6000 samples
Epoch 1/50
54000/54000 [=====] - 3s -
loss: 0.7247 - val_loss: 0.3616
Epoch 2/50
54000/54000 [=====] - 3s -
loss: 0.3718 - val_loss: 0.2815
Epoch 3/50
54000/54000 [=====] - 3s -
loss: 0.3087 - val_loss: 0.2447

[...]
Epoch 50/50
54000/54000 [=====] - 3s -
loss: 0.0485 - val_loss: 0.1174
```

Den Wert der Straffunktion während des Trainings auszugeben, ist außerordentlich nützlich, denn auf diese Weise können wir sofort feststellen, dass die Werte kleiner werden, und den Algorithmus früher beenden oder anderenfalls die Werte der Hyperparameter abstimmen.

Zur Vorhersage der Klassenbezeichnungen können wir die `predict_classes`-Methode verwenden, die die Bezeichnungen direkt als Ganzzahlen zurückliefert:

```
>>> y_train_pred = model.predict_classes(X_train_centered,
...                                         verbose=0)
>>> print('Die ersten 3 Vorhersagen: ', y_train_pred[:3])
Die ersten 3 Vorhersagen: [5 0 4]
```

Zum Abschluss geben wir die Korrektklassifizierungsraten des Modells für Trainings- und Testdatenmenge aus:

```
>>> y_train_pred = model.predict_classes(X_train_centered,
...                                         verbose=0)
>>> correct_preds = np.sum(y_train == y_train_pred, axis=0)
>>> train_acc = correct_preds / y_train.shape[0]
>>>
>>> print('Die ersten 3 Vorhersagen: ', y_train_pred[:3])
Die ersten 3 Vorhersagen: [5 0 4]
>>>
>>> print('Korrektklassifizierungsrate Training:')
```

```

...      %.2f%%' % (train_acc * 100))
Korrektklassifizierungsrate Training: 98.88%
>>>
>>> y_test_pred = model.predict_classes(X_test_centered,
...                                         verbose=0)
>>> correct_preds = np.sum(y_test == y_test_pred, axis=0)
>>> test_acc = correct_preds / y_test.shape[0]
>>> print('Korrektklassifizierungsrate Test: %.2f%%' % (test_acc * 100))
Korrektklassifizierungsrate Test: 96.04%

```

Beachten Sie, dass es sich hier lediglich um ein sehr einfaches neuronales Netz ohne Optimierung der Parameter handelt. Wenn Sie noch ein wenig mehr mit Keras herumexperimentieren möchten, steht es Ihnen selbstverständlich frei, die Stellschrauben wie Lernrate, Trägheit (Momentum), Verringerungsrate und Anzahl der verdeckten Einheiten zu variieren.

### 13.3 Auswahl der Aktivierungsfunktionen mehrschichtiger neuronaler Netze

Der Einfachheit halber haben wir im Zusammenhang mit mehrschichtigen Feed-forward-Netzen bisher nur die sigmoide Aktivierungsfunktion betrachtet. In Kapitel 12 haben wir sie bei der Implementierung des mehrschichtigen Perzeptrons sowohl in der verdeckten Schicht als auch in der Ausgabeschicht verwendet.

Wir haben diese Aktivierungsfunktion bislang mit dem allgemein gebräuchlichen Begriff *Sigmoidfunktion* bezeichnet, allerdings wären *logistische Funktion* oder *negative Log-likelihood-Funktion* präzisere Definitionen. In den folgenden Abschnitten werden Sie einige alternative Sigmoidfunktionen kennenlernen, die sich bei der Implementierung mehrschichtiger neuronaler Netze als nützlich erweisen.

Rein technisch betrachtet könnten wir jede beliebige Funktion als Aktivierungsfunktion eines mehrschichtigen neuronalen Netzes verwenden, sofern sie nur differenzierbar ist. Wir könnten sogar wie in Kapitel 2 beim Adaline (ADAptive LInear NEuron)-Algorithmus lineare Aktivierungsfunktionen verwenden. In der Praxis wären diese allerdings weder für die verdeckte Schicht noch für die Ausgabeschicht besonders sinnvoll, denn in einem typischen neuronalen Netz wollen wir ja gerade Nichtlinearität einbringen, um komplexe Aufgabenstellungen in Angriff nehmen zu können, denn die Summe linearer Funktionen ist schließlich ebenfalls eine lineare Funktion.

Die in den vorangegangenen Kapiteln eingesetzte logistische Aktivierungsfunktion ahmt die Funktionsweise der Neuronen im Gehirn vermutlich am besten nach. Wir können sie uns als die Wahrscheinlichkeit vorstellen, dass ein Neuron feuert.

Logistische Aktivierungsfunktionen sind jedoch nicht ganz unproblematisch, wenn es große negative Eingaben gibt, weil die Ausgabe der Sigmoidfunktion in diesem Fall fast null ist. Das neuronale Netz lernt dann nur sehr langsam und die Wahrscheinlichkeit steigt, dass der Algorithmus während des Trainings in einem der lokalen Minima stecken bleibt. Aus diesem Grund wird in verdeckten Schichten häufig der *Tangens hyperbolicus* als Aktivierungsfunktion verwendet.

Bevor wir näher darauf eingehen, rekapitulieren wir aber kurz noch einmal die Grundlagen der logistischen Funktion und betrachten eine Verallgemeinerung, durch die sie sich besser für Mehrfachklassifizierungen eignet.

### 13.3.1 Die logistische Funktion kurz zusammengefasst

Wie in der Einleitung dieses Abschnitts erwähnt, ist die generell einfach als »*Sigmoidfunktion*« bezeichnete logistische Funktion tatsächlich ein Spezialfall einer Sigmoidfunktion. Aus dem Abschnitt über die logistische Regression in Kapitel 3 wissen Sie bereits, dass die logistische Funktion genutzt werden kann, um die Wahrscheinlichkeit zu modellieren, dass ein Objekt  $x$  bei einer binären Klassifizierung zur positiven Klasse (Klasse 1) gehört. Die folgende Gleichung zeigt die Nettoeingabe:

$$z = w_0x_0 + w_1x_1 + \dots + w_mx_m = \sum_{i=0}^m w_i x_i = w^T x$$

Die logistische Funktion berechnet Folgendes:

$$\phi_{\text{logistisch}}(z) = \frac{1}{1 + e^{-z}}$$

Beachten Sie, dass  $w_0$  in diesem Fall die Bias-Einheit bezeichnet (y-Achsenabschnitt, d.h.  $x_0 = 1$ ). Um ein konkreteres Beispiel zu betrachten, gehen wir von einem Modell für einen zweidimensionalen Datenpunkt  $x$  aus. Bei einem weiteren Modell sind dem Vektor  $w$  die folgenden Gewichtungskoeffizienten zugewiesen:

```
>>> import numpy as np
>>> X = np.array([1, 1.4, 2.5]) ## first value must be 1
>>> w = np.array([0.4, 0.3, 0.5])
>>> def net_input(X, w):
...     return np.dot(X, w)
...
>>> def logistic(z):
...     return 1.0 / (1.0 + np.exp(-z))
...
>>> def logistic_activation(X, w):
...     z = net_input(X, w)
...     return logistic(z)
```

```
...
>>> print('P(y=1|x) = %.3f' % logistic_activation(X, w))
P(y=1|x) = 0.888
```

Wenn wir die Nettoeingabe berechnen und sie zur Aktivierung eines logistischen Neurons mit diesen speziellen Merkmalswerten und Gewichtungskoeffizienten verwenden, erhalten wir den Wert 0.888, den wir als die 88,8-prozentige Wahrscheinlichkeit interpretieren können, dass Objekt  $x$  zur positiven Klasse gehört.

In Kapitel 12 haben wir die One-hot-Codierung verwendet, um die aus mehreren logistischen Aktivierungseinheiten bestehenden Werte der Ausgabeschicht zu berechnen. Allerdings liefert eine aus mehreren logistischen Aktivierungseinheiten bestehende Ausgabeschicht keine sinnvollen, als Wahrscheinlichkeiten interpretierbaren Werte, wie das folgende Beispiel zeigt:

```
>>> # W: Array mit Shape = (n_output_units, n_hidden_units+1)
... # Die erste Spalte enthält die Bias-Einheiten
...
>>> W = np.array([[1.1, 1.2, 0.8, 0.4],
...                 [0.2, 0.4, 1.0, 0.2],
...                 [0.6, 1.5, 1.2, 0.7]])
>>>
>>> # A: Datenarray mit Shape = (n_hidden_units + 1,
...         n_samples)
... # Die erste Spalte dieses Arrays muss 1 enthalten.
...
>>> A = np.array([[1, 0.1, 0.4, 0.6]])
>>>
>>> Z = np.dot(W, A[0])
>>> y_probas = logistic(Z)
>>> print('Nettoeingabe: \n', Z)
Nettoeingabe:
 [ 1.78  0.76  1.65]
>>> print('Ausgabeeinheiten:\n', y_probas)
Ausgabeeinheiten:
 [ 0.85569687  0.68135373  0.83889105]
```

Wie die Ausgabe zeigt, können die resultierenden Werte nicht als Wahrscheinlichkeiten für eine Klassifizierungsaufgabe mit drei Klassen interpretiert werden, denn ihre Summe ergibt nicht 1. Tatsächlich spielt dieser Umstand jedoch kaum eine Rolle, wenn wir unser Modell nur zur Vorhersage der Klassenbezeichnungen, nicht aber der Zugehörigkeitswahrscheinlichkeiten nutzen. Zur Vorhersage der Klassenbezeichnung kann z.B. der Maximalwert verwendet werden:

```
>>> y_class = np.argmax(Z, axis=0)
>>> print('Vorhergesagte Klasse: %d' % y_class[0])
Vorhergesagte Klasse: 0
```

Unter bestimmten Umständen kann es jedoch von Nutzen sein, bei Mehrfachklassifizierungen aussagekräftige Klassenzugehörigkeitswahrscheinlichkeiten zurückzugeben. Im nächsten Abschnitt betrachten wir eine Verallgemeinerung der logistischen Funktion, die softmax-Funktion, die bei dieser Aufgabe hilfreich ist.

### 13.3.2 Wahrscheinlichkeiten bei der Mehrfachklassifizierung mit der softmax-Funktion abschätzen

Im letzten Abschnitt haben wir die Klassenbezeichnung mithilfe der argmax-Funktion ermittelt. Die softmax-Funktion ist de facto eine Variante der argmax-Funktion, die keinen einzelnen Index einer Klasse, sondern die Wahrscheinlichkeiten für die verschiedenen Klassen zurückgibt. Sie ermöglicht es somit, bei der Mehrfachklassifizierung (multinomiale logistische Regression) aussagekräftige Klassenwahrscheinlichkeiten zu berechnen.

Bei diesem Verfahren kann die Wahrscheinlichkeit, dass ein bestimmtes Objekt mit der Nettoeingabe  $z$  zur  $i$ -ten Klasse gehört, mit einem Normierungsterm im Nenner berechnet werden, der die Summe aller  $M$  linearen Funktionen enthält:

$$p(y=i|z) = \phi(z) = \frac{e^{z_i}}{\sum_{j=1}^M e^{z_j}}$$

Um die softmax-Funktion in Aktion zu sehen, schreiben wir folgenden Python-Code:

```
>>> def softmax(z):
...     return np.exp(z) / np.sum(np.exp(z))
...
>>> y_probas = softmax(Z)
>>> print('Wahrscheinlichkeiten:\n', y_probas)
Wahrscheinlichkeiten:
 [ 0.44668973  0.16107406  0.39223621]
>>> np.sum(y_probas)
1.0
```

Wie man sieht, summieren sich die vorhergesagten Wahrscheinlichkeiten nun erwartungsgemäß zu 1. Außerdem ist hier bemerkenswert, dass die vorhergesagte Klassenbezeichnung dieselbe ist wie bei Anwendung der argmax-Funktion auf die logistische Funktion. Man kann sich die softmax-Funktion als eine *normierte* logistische Funktion vorstellen, die sich als nützlich erweist, um bei Mehrfachklassifizierungen aussagekräftige Vorhersagen zur Klassenzugehörigkeit zu erhalten.

### 13.3.3 Verbreiterung des Ausgabespektrums mittels Tangens hyperbolicus

Der *Tangens hyperbolicus* (üblicherweise abgekürzt als `tanh`) ist eine weitere Sigmoidfunktion, die häufig in den verdeckten Schichten künstlicher neuronaler Netze Verwendung findet. Sie kann als eine neu skalierte Version der logistischen

Funktion interpretiert werden:  $\phi_{\text{logistisch}}(z) = \frac{1}{1+e^{-z}}$

$$\phi_{\tanh}(z) = 2 \times \phi_{\text{logistisch}}(2 \times z) - 1 = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Der Vorteil des Tangens hyperbolicus gegenüber der logistischen Funktion besteht darin, dass er mit dem offenen Intervall  $(-1, 1)$  ein breiteres Spektrum von Ausgabewerten besitzt. Dadurch kann die Konvergenz des Backpropagation-Algorithmus verbessert werden (C.M. Bishop, *Neural networks for pattern recognition*, Oxford University Press, 1995, Seiten 500–501).

Die logistische Funktion hingegen liefert Ausgabewerte im offenen Intervall  $(0, 1)$ . Zum anschaulichen Vergleich von logistischer Funktion und Tangens hyperbolicus geben wir zwei Sigmoidfunktionen aus:

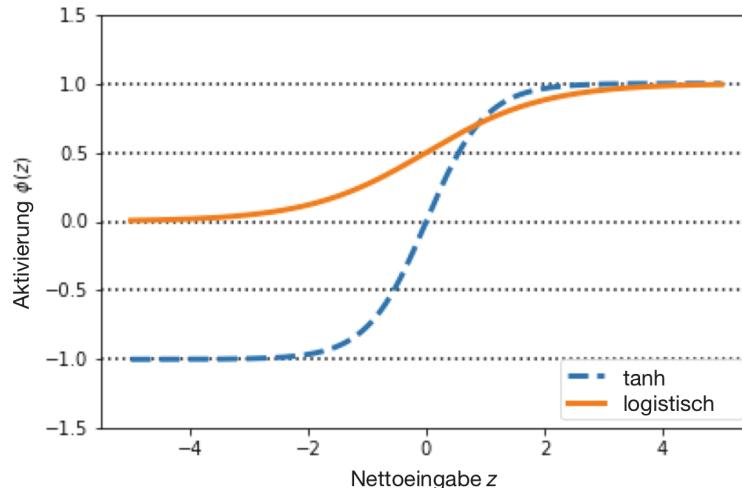
```
>>> import matplotlib.pyplot as plt

>>> def tanh(z):
...     e_p = np.exp(z)
...     e_m = np.exp(-z)
...     return (e_p - e_m) / (e_p + e_m)

>>> z = np.arange(-5, 5, 0.005)
>>> log_act = logistic(z)
>>> tanh_act = tanh(z)
>>> plt.ylim([-1.5, 1.5])
>>> plt.xlabel('Nettoeingabe $z$')
>>> plt.ylabel('Aktivierung $\phi(z)$')
>>> plt.axhline(1, color='black', linestyle=':')
>>> plt.axhline(0.5, color='black', linestyle=':')
>>> plt.axhline(0, color='black', linestyle=':')
>>> plt.axhline(-0.5, color='black', linestyle=':')
>>> plt.axhline(-1, color='black', linestyle=':')
>>> plt.plot(z, tanh_act,
...            linewidth=3, linestyle='--',
...            label='tanh')
>>> plt.plot(z, log_act,
...            linewidth=3,
...            label='logistisch')
>>> plt.legend(loc='lower right')
```

```
>>> plt.tight_layout()
>>> plt.show()
```

Wie Sie sehen, sind die Formen der beiden Sigmoidfunktionen sehr ähnlich, allerdings ist das Ausgabespektrum der `tanh`-Funktion doppelt so groß wie das der logistischen Funktion.



Beachten Sie, dass die Funktionen `logistic` und `tanh` hier zur Veranschaulichung ausführlich implementiert sind. In der Praxis lässt sich allerdings mit der `tanh`-Funktion von NumPy dasselbe Resultat erzielen:

```
>>> tanh_act = np.tanh(z)
```

Und auch die logistische Funktion ist über das SciPy-Modul `special` verfügbar:

```
>>> from scipy.special import expit
>>> log_act = expit(z)
```

### 13.3.4 Aktivierung durch rektifizierte Lineareinheiten

In tiefen neuronalen Netzen werden häufig *rektifizierte Lineareinheiten* (Rectified Linear Units, kurz ReLUs) verwendet.

Zum besseren Verständnis der ReLu müssen wir zunächst das Problem des verschwindenden Gradienten betrachten, das bei `tanh`- und logistischer Aktivierungsfunktion auftreten kann.

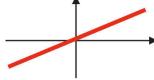
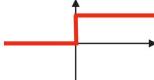
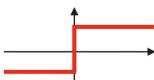
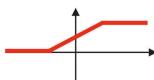
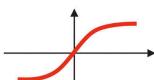
Nehmen wir an, es liegt eine Nettoeingabe  $z_1 = 20$  vor, die sich auf  $z_1 = 25$  ändert. Die Berechnung der Aktivierung ergibt  $\phi(z_1) \approx 1.0$  und  $\phi(z_1) \approx 1.0$ , zeigt also keine Änderung der Ausgabe.

Das bedeutet, dass die Ableitung der Aktivierung bezüglich der Nettoeingabe für große Werte von  $z$  verschwindet. Das führt zu einer sehr langsamem Ermittlung der Gewichtungen in der Trainingsphase, weil die Gradiententerme sehr nah bei null liegen können. Die ReLU-Aktivierung behebt dieses Problem. ReLU ist wie folgt definiert:

$$\phi(z) = \max(0, z)$$

ReLU ist eine lineare Funktion, die gut für das Erlernen komplexer Funktionen neuronaler Netze geeignet ist. Die Ableitung der ReLU nach der Eingabe ist für positive Eingabewerte immer 1, wodurch das Problem des verschwindenden Gradienten gelöst ist. Im nächsten Kapitel werden wir ReLU als Aktivierungsfunktion eines konvolutionalen neuronalen Netzes verwenden.

Nachdem Sie damit etwas mehr über die verschiedenen Aktivierungsfunktionen wissen, die üblicherweise bei künstlichen neuronalen Netzen zum Einsatz kommen, schließen wir diesen Abschnitt mit einer Übersicht über die verschiedenen Aktivierungsfunktionen ab, denen Sie in diesem Buch begegnet sind.

Aktivierungsfunktion	Gleichung	Beispiel	1-D-Graph
Linear	$\phi(z) = z$	Adaline, lineare Regression	
Sprungfunktion (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Varianten des Perzeptrons	
Vorzeichen (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Varianten des Perzeptrons	
Stückweise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support Vector Machine	
Logistisch (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistische Regression, mehrschichtige neuronale Netze	
Tangens hyperbolicus (tanh)	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Mehrschichtige neuronale Netze, rekurrente neuronale Netze	
ReLU	$\phi(z) = \begin{cases} 0, & z < 0, \\ z, & z > 0, \end{cases}$	Mehrschichtige neuronale Netze, konvolutionale neuronale Netze	

## 13.4 Zusammenfassung

In diesem Kapitel haben Sie TensorFlow kennengelernt, eine quelloffene Bibliothek für numerische Berechnungen, die insbesondere für das Deep Learning ausgelegt ist. Die Verwendung von TensorFlow ist gegenüber NumPy aufgrund der mit der Unterstützung von GPUs verbundenen Komplexität zwar etwas weniger komfortabel, ermöglicht jedoch das sehr effiziente Erstellen und Trainieren großer, mehrschichtiger neuronaler Netze.

Außerdem haben Sie erfahren, wie sich die TensorFlow-API zur Entwicklung und Ausführung komplexer Modelle des Machine Learnings und neuronaler Netze einsetzen lässt. Zunächst haben wir die Programmierung der Low-level-API von TensorFlow erkundet. Die Programmierung auf dieser Ebene kann ein wenig mühsam sein, wenn man Matrix-Vektor-Multiplikationen ausführen und jedes Detail einer Operation definieren muss, hat jedoch den Vorteil, dass sie uns als Entwicklern erlaubt, diese grundlegenden Operationen miteinander zu kombinieren und komplexere Modelle zu entwickeln. Darüber hinaus haben wir erörtert, dass TensorFlow beim Trainieren und Testen großer neuronaler Netze die Verwendung von GPUs gestattet. Ohne den Einsatz von GPUs würde das Trainieren mancher Netze typischerweise mehrere Monate Rechenzeit in Anspruch nehmen!

Anschließend haben wir uns mit zwei der High-level-APIs befasst, die das Entwickeln neuronaler Netze im Vergleich zur Low-Level-API stark vereinfacht. Wir haben Tensorflows Layers und Keras zur Entwicklung mehrschichtiger neuronaler Netze verwendet und erfahren, wie man die APIs zu diesem Zweck einsetzt.

Und schließlich haben Sie eine Reihe verschiedener Aktivierungsfunktionen, deren Verhalten und Anwendungsfälle kennengelernt. In diesem Kapitel ging es insbesondere um `tanh`, `softmax` und `ReLU`. In Kapitel 12 haben wir damit angefangen, ein einfaches *MLP* (Multi-Layer Perceptron) zur Klassifizierung handgeschriebener Ziffern der MNIST-Datensammlung zu implementieren. Die von Grund auf durchgeführte Low-level-Implementierung war eine gute Übung für das Verständnis der grundlegenden Konzepte mehrschichtiger neuronaler Netze, wie z.B. Vorrwärtspropagation und das Backpropagation-Verfahren, denn das Trainieren großer neuronaler Netze mit NumPy ist ineffizient und praktisch kaum durchführbar.

Im nächsten Kapitel setzen wir unsere Tour fort und werden uns eingehender mit TensorFlow befassen und dabei Graphen und Sitzungsobjekten begegnen. Unterdessen werden wir viele neue Konzepte wie Platzhalter, Variablen sowie das Speichern und Wiederherstellen von Modellen in TensorFlow kennenlernen.



# Die Funktionsweise von TensorFlow im Detail

In Kapitel 13 haben wir ein mehrschichtiges neuronales Netz zur Klassifizierung der MNIST-Ziffern trainiert und dabei verschiedene Teile von Tensorflows Python-API verwendet. Auf diese Weise konnten wir unmittelbar praktische Erfahrung mit dem Einsatz von TensorFlow beim Trainieren neuronaler Netze und dem Machine Learning sammeln.

In diesem Kapitel werden wir uns voll und ganz auf TensorFlow selbst konzentrieren und die beeindruckenden Mechanismen und Features ausführlich betrachten:

- Grundlegende Merkmale und Vorteile von TensorFlow
- TensorFlow-Tensoren und deren Rang
- Verwendung von TensorFlow-Graphen
- TensorFlow-Operationen mit verschiedenen Geltungsbereichen
- Tensor-Transformationen: Rang, Shape und Typ
- Tensoren als mehrdimensionale Arrays transformieren
- Speichern und Wiederherstellen von Modellen
- Visualisierung neuronaler Netzgraphen mit TensorBoard

Wir werden uns in diesem Kapitel vornehmlich mit der Praxis befassen und beim Erkunden der wesentlichen Features und Konzepte von TensorFlow verschiedene Graphen implementieren. Wir werden auf ein schon bekanntes Regressionsmodell zurückkommen, Graphen neuronaler Netze mit TensorBoard visualisieren und einige Vorschläge erörtern, wie Sie die im Verlaufe dieses Kapiteln erstellten Graphen bei der Visualisierung genauer untersuchen können.

## 14.1 Grundlegende Merkmale von TensorFlow

TensorFlow stellt eine skalierbare, plattformübergreifende Programmierschnittstelle für die Implementierung und Ausführung von Machine-Learning-Algorithmen bereit. Die TensorFlow-API ist seit der Veröffentlichung der Version 1.0 im Jahr 2017 relativ stabil und ausgereift. Die übrigen verfügbaren Deep-Learning-Bibliotheken sind im Vergleich zu TensorFlow noch als experimentell zu betrachten.

Eines der herausragenden Merkmale von TensorFlow, das in Kapitel 13 bereits erwähnt wurde, ist die Fähigkeit, GPUs verwenden zu können. Das ermöglicht es dem Benutzer, Lernmodelle auf großen Systemen effizient zu trainieren.

Die Entwicklung von TensorFlow wird kräftig vorangetrieben, denn sie wird von Google finanziert und unterstützt. Ein großes Programmiererteam ist fortlaufend damit beschäftigt, die Software zu verbessern. Außerdem wird TensorFlow auch von Open-Source-Entwicklern unterstützt, die eifrig Beiträge leisten und Feedback liefern. Auf diese Weise ist die Bibliothek sowohl für die Forschung als auch für die Entwickler in der Branche von größerem Wert. Ein weiterer Vorteil dieses Sachverhalts ist die umfangreiche Dokumentation und die Vielzahl der Tutorials, die neuen Anwendern zur Verfügung stehen.

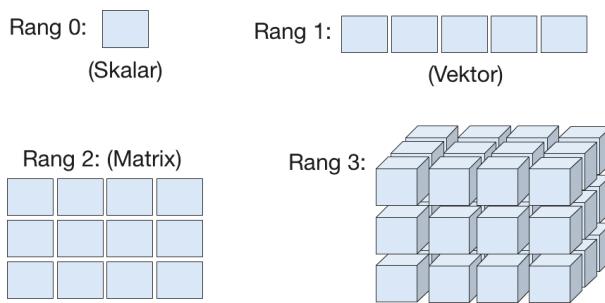
Und nicht zuletzt unterstützt TensorFlow auch die Anwendung auf mobilen Geräten, wodurch es sehr gut für den produktiven Einsatz geeignet ist.

## 14.2 TensorFlow-Tensoren und deren Rang

Die TensorFlow-Bibliothek ermöglicht es dem Benutzer, Operationen und Funktionen mithilfe von Tensoren als Berechnungsgraphen zu definieren. Tensoren sind eine verallgemeinerte mathematische Notation für mehrdimensionale Arrays, die Daten speichern. Die Dimensionalität eines Tensors wird als sein *Rang* (oder manchmal auch als *Stufe*) bezeichnet.

Bislang haben wir hauptsächlich Tensoren des Rangs 0 bis 2 verwendet. Ein Skalar, also eine einzelne Zahl wie eine Ganzzahl oder eine Fließkommazahl, ist ein Tensor des Rangs 0. Ein Vektor besitzt den Rang 1 und eine Matrix ist ein Tensor des Rangs 2. Es geht jedoch noch weiter. Die Tensornotation lässt sich auf höhere Dimensionen verallgemeinern – wie Sie im nächsten Kapitel bei der Verwendung eines Eingabetensors des Rangs 3 und eines Gewichtungstensors des Rangs 4 zur Unterstützung von Bildern mit mehreren Farbkanälen noch sehen werden.

Sehen Sie sich die folgende Abbildung an, die das Konzept eines Tensors veranschaulichen soll. Im oberen Bereich sind Tensoren des Rangs 0 und 1 dargestellt, im unteren Tensoren des Rangs 2 und 3.



### 14.2.1 Rang und Form eines Tensors ermitteln

Mit der Funktion `tf.rank` kann der Rang eines Tensors ermittelt werden. An dieser Stelle ist es wichtig anzumerken, dass `tf.rank` als Ausgabe einen Tensor zurückgibt. Um den eigentlichen Wert zu erhalten, muss dieser Tensor ausgewertet werden.

Neben dem Rang kann auch die Form eines TensorFlow-Tensors (ähnlich wie bei einem NumPy-Array) abgefragt werden. Wenn beispielsweise `X` ein Tensor ist, können wir direkt `X.get_shape()` aufrufen und erhalten ein Objekt zurück, das zu der Klasse `TensorShape` gehört.

Beim Erzeugen anderer Tensoren kann dieses Objekt direkt dem `shape`-Argument übergeben werden. Es ist allerdings *nicht* möglich, eine direkte Indizierung oder Slicing auf dieses Objekt anzuwenden. In diesem Fall muss das Objekt zunächst mit der `as_list`-Methode der Tensor-Klasse in eine Python-Liste konvertiert werden.

Die folgenden Beispiele zeigen, wie die `tf.rank`-Funktion und die `get_shape`-Methode eines Tensors verwendet werden. Der folgende Code demonstriert, wie der Rang und die Form eines Tensors in einer TensorFlow-Sitzung ermittelt werden:

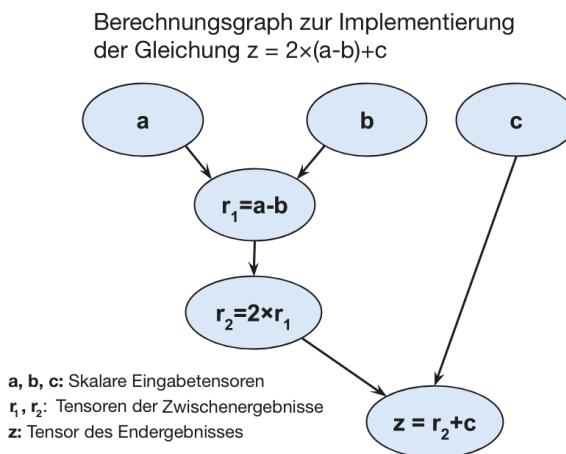
```
>>> import tensorflow as tf
>>> import numpy as np
>>>
>>> g = tf.Graph()
>>>
>>> ## Berechnungsgraph definieren
>>> with g.as_default():
...     ## Tensoren t1, t2, t3 definieren
...     t1 = tf.constant(np.pi)
...     t2 = tf.constant([1, 2, 3, 4])
...     t3 = tf.constant([[1, 2], [3, 4]])
...
...
...     ## Rang abfragen
...     r1 = tf.rank(t1)
...     r2 = tf.rank(t2)
...     r3 = tf.rank(t3)
...
...
...     ## Form abfragen
...     s1 = t1.get_shape()
...     s2 = t2.get_shape()
...     s3 = t3.get_shape()
...     print('Form:', s1, s2, s3)
Form: [] (4,) (2, 2)
```

```
>>> with tf.Session(graph=g) as sess:
...     print('Rang:', 
...           r1.eval(),
...           r2.eval(),
...           r3.eval())
Rang: 0 1 2
```

Der Tensor  $t_1$  hat den Rang 0, denn es handelt sich um ein Skalar (entsprechend der Form  $[]$ ). Der Rang des Vektors  $t_2$  ist 1, und da er vier Elemente besitzt, ist die Form das aus nur einem Element bestehende Tupel  $(4,)$ . Für den Rang der  $2 \times 2$ -Matrix  $t_3$  ergibt sich 2 und die entsprechende Form ist das Tupel  $(2, 2)$ .

## 14.3 TensorFlow-Berechnungsgraphen

TensorFlow beruht grundsätzlich auf einem Berechnungsgraphen und verwendet diesen, um zwischen Eingabe und Ausgabe Beziehungen zwischen den Tensoren herzuleiten. Nehmen wir an, es liegen die skalaren (Rang 0) Tensoren  $a, b$  sowie  $c$  vor und wir wollen den Ausdruck  $z = 2 \times (a - b) + c$  auswerten. Diese Auswertung kann durch einen Berechnungsgraphen repräsentiert werden (siehe Abbildung).



Der Berechnungsgraph ist also einfach nur ein aus verschiedenen Knoten bestehendes Netz. Jeder Knoten entspricht einer Operation, die eine Funktion auf den Eingabetensor oder die Eingabetensoren anwendet und null oder mehr Tensoren ausgibt.

TensorFlow konstruiert diesen Berechnungsgraphen und verwendet ihn zur Berechnung der Gradienten. Das Konstruieren und Kompilieren eines solchen Berechnungsgraphen besteht aus den folgenden Schritten:

1. Instanziieren eines neuen Berechnungsgraphen
2. Hinzufügen von Knoten (Tensoren und Operationen)
3. Ausführung des Graphen:
  - a) Eröffnen einer neuen Sitzung
  - b) Initialisierung der Variablen des Graphen
  - c) Ausführung des Berechnungsgraphen in dieser Sitzung

Erstellen wir also den Graphen für die Auswertung von  $z=2 \times (a-b)+c$  wie in der Abbildung, in der  $a$ ,  $b$  und  $c$  Skalare (einzelne Zahlen) sind. Wir definieren sie hier als TensorFlow-Konstanten. Der Aufruf von `tf.Graph()` erstellt einen neuen Graphen, dem wie folgt Knoten hinzugefügt werden können:

```
>>> g = tf.Graph()
>>>
>>> with g.as_default():
...     a = tf.constant(1, name='a')
...     b = tf.constant(2, name='b')
...     c = tf.constant(3, name='c')
...
...     z = 2*(a-b) + c
```

Der Code fügt dem Graphen `g` per `with g.as_default()` neue Knoten hinzu. Wenn wir nicht ausdrücklich einen neuen Graphen erzeugen, wird ein standardmäßig vorhandener Graph verwendet, dem dann die Knoten hinzugefügt werden. Der Deutlichkeit halber werden wir in diesem Buch die Verwendung des Standardgraphen vermeiden. Dieser Ansatz erweist sich bei der Entwicklung von Code in einem Jupyter-Notebook als besonders nützlich, weil dadurch verhindert wird, dass dem Standardgraphen versehentlich unerwünschte Knoten hinzugefügt werden.

Eine TensorFlow-Sitzung stellt eine Umgebung dar, in der Operationen mit Tensoren eines Graphen ausgeführt werden können. Mit dem Aufruf von `tf.Session` wird ein neues Sitzungsobjekt erzeugt, dem ein bereits vorhandener Graph (hier `g`) mit `tf.Session(graph=g)` übergeben werden kann. Andernfalls wird der Standardgraph gestartet, der womöglich leer ist.

Nach dem Start eines Graphen in einer TensorFlow-Sitzung können die Knoten ausgeführt werden, das heißt, dass die Tensoren ausgewertet oder die Operationen durchgeführt werden. Um einen bestimmten Knoten des Graphen auszuführen, muss TensorFlow zunächst alle vorhergehenden Knoten im Graphen ausführen, bis schließlich der fragliche Knoten erreicht ist. Falls Platzhalter vorhanden sind, müssen ihnen Daten zugeführt werden, wie Sie im nächsten Abschnitt sehen werden.

Die Ausführung von Operationen kann auch mit der `run`-Methode einer Sitzung gestartet werden. Im letzten Beispiel ist `train_op` ein Operator, der keinen Tensor zurückliefert. Dieser Operator kann mit `train_op.run()` ausgeführt werden. Darüber hinaus gibt es eine universelle Methode, sowohl Tensoren auszuwerten als auch Operatoren auszuführen: `tf.Session().run()`. Wie Sie später noch sehen werden, können mit dieser Methode auch mehrere Tensoren und Operatoren in einer Liste oder einem Tupel platziert werden. `tf.Session().run()` gibt in diesem Fall eine Liste oder ein Tupel gleicher Größe zurück.

Hier starten wir den Graphen in einer TensorFlow-Sitzung und werten den Tensor `z` folgendermaßen aus:

```
>>> with tf.Session(graph=g) as sess:
...     print('2*(a-b)+c => ', sess.run(z))
2*(a-b)+c => 1
```

Tensoren und Operationen werden in TensorFlow stets im Kontext eines Berechnungsgraphen definiert. Anschließend wird eine TensorFlow-Sitzung eröffnet, die die Operationen des Graphen ausführt und die Ergebnisse auswertet und zurückgibt.

In diesem Abschnitt haben Sie erfahren, wie ein Berechnungsgraph definiert wird, wie ihm Knoten hinzugefügt werden und wie die Tensoren eines Graphen im Rahmen einer TensorFlow-Sitzung ausgewertet werden. Nun werden wir uns die verschiedenen Knotentypen, einschließlich Platzhaltern und Variablen, die in einem Berechnungsgraphen vorkommen können, etwas genauer ansehen. Dabei werden wir einigen weiteren Operatoren begegnen, die keine Tensoren als Ausgabe zurückliefern.

## 14.4 Platzhalter in TensorFlow

TensorFlow verfügt über spezielle Mechanismen zum Zuführen von Daten, zu denen auch Platzhalter gehören. Dabei handelt es sich um vordefinierte Tensoren bestimmten Typs mit vorgegebener Form.

Diese Tensoren werden dem Berechnungsgraphen mithilfe der Funktion `tf.placeholder` hinzugefügt. Sie enthalten keine Daten, aber bei der Ausführung bestimmter Knoten des Graphen müssen ihnen Datenarrays zugeführt werden.

In den folgenden Abschnitten werden Sie sehen, wie die Platzhalter in einem Graphen definiert werden und wie man ihnen bei der Ausführung Daten zuführt.

#### 14.4.1 Platzhalter definieren

Wie Sie wissen, werden Platzhalter mit der Funktion `tf.placeholder` definiert. Dabei müssen wir gemäß der Daten, die einem Platzhalter bei der Ausführung zugeführt werden sollen, entscheiden, von welcher Form und von welchem Typ er sein soll.

Fangen wir mit einem einfachen Beispiel an. Der folgende Code definiert denselben Graphen zur Auswertung von  $z=2 \times (a-b)+c$ , der im letzten Abschnitt abgebildet ist. Jetzt verwenden wir jedoch Platzhalter für die Skalare  $a$ ,  $b$  und  $c$ . Außerdem speichern wir die Zwischenergebnisse für  $r_1$  und  $r_2$  wie folgt:

```
>>> import tensorflow as tf
>>>
>>> g = tf.Graph()
>>> with g.as_default():
...     tf_a = tf.placeholder(tf.int32, shape=[],
...                           name='tf_a')
...     tf_b = tf.placeholder(tf.int32, shape=[],
...                           name='tf_b')
...     tf_c = tf.placeholder(tf.int32, shape=[],
...                           name='tf_c')
...     r1 = tf_a-tf_b
...     r2 = 2*r1
...     z = r2 + tf_c
```

Der Code definiert drei Platzhalter namens `tf_a`, `tf_b` und `tf_c` des Typs `tf.int32` (32-Bit-Ganzzahlen) und weist ihnen per `shape=[]` ihre Form zu, weil es sich um Skalare (Tensoren des Rangs 0) handelt. In diesem Buch stellen wir Platzhalterobjekten der Deutlichkeit halber und damit man sie von anderen Tensoren unterscheiden kann, immer ein `tf_` voran.

Im letzten Codebeispiel hatten wir es mit Skalaren zu tun, deshalb wurde ihnen `shape=[]` zugewiesen. Es ist jedoch sehr leicht möglich, Platzhalter höherer Dimension zu definieren. Ein Platzhalter des Rangs 3 und des Typs `float` mit der Form  $3 \times 4 \times 5$  könnte beispielsweise durch `tf.placeholder(dtype=tf.float32, shape=[2, 3, 4])` definiert werden.

#### 14.4.2 Platzhaltern Daten zuführen

Bei der Ausführung eines Knotens in einem Graphen müssen wir ein Python-Dictionary erstellen, um einem Platzhalter gemäß seiner Form und seines Typs Datenarrays zuzuführen. Dieses Dictionary wird als Eingabeargument `feed_dict` an die `run`-Methode einer Sitzung übergeben.

Dem letzten Graphen haben wir drei Platzhalter des Typs `tf.int32` hinzugefügt, um die Skalare zur Berechnung von `z` zuzuführen. Um nun den resultierenden Tensor `z` auszuwerten, können wir den Platzhaltern beliebige Ganzzahlen (hier 1, 2 und 3) zuführen:

```
>>> with tf.Session(graph=g) as sess:
...     feed = {tf_a: 1,
...             tf_b: 2,
...             tf_c: 3}
...     print('z:', sess.run(z, feed_dict=feed))
z: 1
```

Das bedeutet, dass ein zusätzliches Array für Platzhalter keinen Fehler verursacht; es ist lediglich redundant. Wenn für die Ausführung eines bestimmten Knotens ein Platzhalter benötigt wird, dem über das `feed_dict`-Argument kein Dictionary bereitgestellt wird, kommt es zu einem Laufzeitfehler.

#### 14.4.3 Platzhalter für Datenarrays mit variierenden Stapelgrößen definieren

Bei der Entwicklung von Modellen neuronaler Netze hat man es gelegentlich mit Mini-Batch-Teilmengen unterschiedlicher Größe zu tun. Wir könnten beispielsweise ein neuronales Netz mit einer bestimmten Stapelgröße trainieren, aber mit dem neuronalen Netz Vorhersagen für eine Eingabe oder mehrere Eingaben treffen wollen.

Platzhalter besitzen die angenehme Eigenschaft, dass man für die Dimension variierender Größe den Wert `None` angeben kann. Wir können beispielsweise einen Platzhalter des Rangs 2 erzeugen, dessen erste Dimension unbekannt ist (oder variieren darf), und zwar so:

```
>>> import tensorflow as tf
>>>
>>> g = tf.Graph()
>>>
>>> with g.as_default():
...     tf_x = tf.placeholder(tf.float32,
...                           shape=[None, 2],
...                           name='tf_x')
...     x_mean = tf.reduce_mean(tf_x,
...                            axis=0,
...                            name='mean')
```

Nun können wir `x_mean` mit zwei verschiedenen Eingabewerten `x1` und `x2` auswerten, bei denen es sich um NumPy-Arrays der Form `(5, 2)` und `(10, 2)` handelt:

```
>>> import numpy as np
>>> np.random.seed(123)
>>> np.set_printoptions(precision=2)
>>> with tf.Session(graph=g) as sess:
...     x1 = np.random.uniform(low=0, high=1,
...                           size=(5, 2))
...     print('Daten der Form ', x1.shape, 'zuführen')
...     print('Ergebnis:', sess.run(x_mean,
...                                 feed_dict={tf_x: x1}))
...     x2 = np.random.uniform(low=0, high=1,
...                           size=(10,2))
...     print('Daten der Form', x2.shape, 'zuführen')
...     print('Ergebnis:', sess.run(x_mean,
...                                 feed_dict={tf_x: x2}))
```

Hier die Ausgabe:

```
Daten der Form (5, 2) zuführen
Ergebnis: [ 0.62  0.47]
Daten der Form (10, 2) zuführen
Ergebnis: [ 0.46  0.49]
```

Und wenn wir zum Abschluss das Objekt `tf_x` ausgeben, erhalten wir `Tensor("tf_x:0", shape=(?, 2), dtype=float32)`, was zeigt, dass dieser Tensor von der Form `(?, 2)` ist.

## 14.5 Variablen in TensorFlow

Variablen sind in TensorFlow spezielle Tensorobjekte, die es ermöglichen, die Modellparameter während des Trainings in einer TensorFlow-Sitzung zu speichern und zu aktualisieren. Im folgenden Abschnitt wird erläutert, wie man Variablen in einem Graphen definiert, sie in einer Sitzung initialisiert, mithilfe des Geltungsbereichs organisiert und wie man schon vorhandene Variablen wiederverwendet.

### 14.5.1 Variablen definieren

TensorFlow-Variablen speichern die Parameter eines Modells, das während des Trainings aktualisiert werden kann. Dazu gehören beispielsweise die Gewichtungen der Eingabeschicht, der verdeckten Schicht und der Ausgabeschicht eines neuronalen Netzes. Bei der Definition einer Variablen muss diese mit einem Tensor initialisiert werden, der die Werte enthält. Weitere Informationen zum Thema TensorFlow-Variablen finden Sie unter [https://www.tensorflow.org/programmers\\_guide/variables](https://www.tensorflow.org/programmers_guide/variables).

Es gibt zwei Möglichkeiten, in TensorFlow Variablen zu definieren:

- `tf.Variable(<Initialisierungswert>, name="Variablename")`
- `tf.get_variable(name, ...)`

Die Klasse `tf.Variable` erstellt für eine neue Variable ein Objekt und fügt es dem Graphen hinzu. Beachten Sie, dass `tf.Variable` keine Möglichkeit bietet, `shape` und `dtype` explizit festzulegen; die beiden Werte werden vom Initialisierungswert übernommen.

Die zweite Option `tf.get_variable` kann zur Wiederverwendung einer schon vorhandenen Variablen mit dem angegebenen Namen verwendet werden, sofern dieser Name im Graphen existiert. Gibt es den Namen noch nicht, wird eine neue Variable dieses Namens erzeugt. In diesem Fall ist der Name von entscheidender Bedeutung und muss deshalb wohl auch als erstes Argument an diese Funktion übergeben werden. Darüber hinaus ermöglicht `tf.get_variable` die explizite Angabe von `shape` und `dtype`. Diese Parameter sind nur beim Erzeugen einer neuen Variablen erforderlich und können bei der Wiederverwendung vorhandener Variablen weggelassen werden.

`tf.get_variable` hat gegenüber `tf.Variable` zwei Vorteile: Zum einen ermöglicht es die Wiederverwendung bereits vorhandener Variablen, zum anderen verwendet es standardmäßig die verbreitete Xavier/Glorot-Initialisierung.

Neben den Initialisierungsoptionen stellt die `get_variable`-Funktion weitere Parameter zur Handhabung des Tensors bereit, so kann beispielsweise ein Regularisierungsterm für die Variable hinzugefügt werden. Die Dokumentation der weiteren Parameter von `tf.get_variable` finden Sie unter [https://www.tensorflow.org/api\\_docs/python/tf/get\\_variable](https://www.tensorflow.org/api_docs/python/tf/get_variable).

### Tipp

#### Xavier- oder Glorot-Initialisierung

In der Anfangsphase der Entwicklung des Deep Learnings hat man festgestellt, dass eine gleichmäßige zufällige oder normalverteilte Initialisierung der Gewichtungen häufig zu einer schlechten Trainingsleistung der Modelle führt.

2010 untersuchten Xavier Glorot und Yoshua Bengio die Auswirkungen der Initialisierung und schlugen ein neues, robusteres Initialisierungsverfahren vor, um das Trainieren tiefer neuronaler Netze zu vereinfachen.

Die grundlegende Idee der Xavier-Initialisierung besteht darin, die Varianz der Gradienten ungefähr gleichmäßig auf die verschiedenen Schichten zu verteilen. Ansonsten wird beim Training womöglich einer der Schichten zu viel Bedeutung beigemessen, während die anderen vernachlässigt werden.

Der Forschungsarbeit von Glorot und Bengio zufolge sollten wir das Intervall für eine gleichmäßige Verteilung der Initialisierung der Gewichtungen wie folgt wählen:

$$W \sim \text{Gleichmäßig} \left( -\frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}} \right)$$

Hier ist  $n_{in}$  die Anzahl der Eingabeneuronen, die mit den Gewichtungen multipliziert werden, und  $n_{out}$  ist die Anzahl der Neuronen, die der nächsten Schicht zugeführt werden. Für die Initialisierung der Gewichtungen mit einer Normalverteilung empfehlen die Autoren, die Gauß'sche Standardabweichung zu wählen:

$$\sigma = \frac{\sqrt{2}}{\sqrt{n_{in} + n_{out}}}$$

TensorFlow unterstützt sowohl die gleichmäßige als auch die normalverteilte Initialisierung der Gewichtungen. Die Dokumentation stellt detaillierte Informationen zur Xavier-Initialisierung mit TensorFlow bereit: [https://www.tensorflow.org/api\\_docs/python/tf/contrib/layers/xavier\\_initializer](https://www.tensorflow.org/api_docs/python/tf/contrib/layers/xavier_initializer).

Weitere Informationen zu Glorots und Bengios Initialisierungsverfahren, inklusive mathematischer Herleitung und Beweis, finden Sie in der Forschungsarbeit (Xavier Glorot und Yoshua Bengio, 2010, *Understanding the difficulty of deep feed-forward neural networks*), die unter <http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf> kostenlos verfügbar ist.

Bei beiden Initialisierungsverfahren ist es wichtig zu beachten, dass die Initialisierungswerte erst dann übermittelt werden, wenn wir den Graphen in `tf.Session` starten und den Initialisierungsoperator in dieser Sitzung ausführen. Tatsächlich wird sogar der benötigte Arbeitsspeicher erst bei der Initialisierung der Variablen der TensorFlow-Sitzung reserviert.

Nun folgt ein Beispiel für das Erstellen eines Variablenobjekts, bei dem die Initialisierungswerte aus einem NumPy-Array stammen. Der Datentyp `dtype` dieses Tensors ist `tf.int64` und wird automatisch aus dem NumPy-Eingabearray abgeleitet:

```
>>> import tensorflow as tf
>>> import numpy as np
>>>
>>> g1 = tf.Graph()
>>>
>>> with g1.as_default():
...     w = tf.Variable(np.array([[1, 2, 3, 4],
...                             [5, 6, 7, 8]]), name='w')
```

```
...     print(w)
<tf.Variable 'w:0' shape=(2, 4) dtype=int64_ref>
```

### 14.5.2 Variablen initialisieren

Hier ist es wichtig, sich klarzumachen, dass der Speicherplatz für als Variablen definierte Tensoren erst bei der Initialisierung reserviert wird – sie enthalten vorher keine Werte. Vor der Ausführung irgendeines Knotens des Berechnungsgraphen müssen die Variablen initialisiert werden, die zu dem Pfad gehören, der zum fraglichen Knoten führt.

Die Initialisierung reserviert Speicherplatz für die beteiligten Tensoren und weist ihnen ihre Anfangswerte zu. TensorFlow stellt eine Funktion namens `tf.global_variables_initializer()` zur Verfügung, die einen Operator zur Initialisierung aller im Berechnungsgraphen vorhandenen Variablen zurückgibt. Durch die Ausführung dieses Operators werden die Variablen folgendermaßen initialisiert:

```
>>> with tf.Session(graph=g1) as sess:
...     sess.run(tf.global_variables_initializer())
...     print(sess.run(w))
[[1 2 3 4]
 [5 6 7 8]]
```

Wir können diesen Operator auch als Objekt speichern, wie z.B. `init_op = tf.global_variables_initializer()`, und ihn später mit `sess.run(init_op)` oder `init_op.run()` ausführen. Wir müssen uns allerdings vergewissern, dass dieser Operator erst dann erstellt wird, nachdem alle Variablen definiert worden sind.

Im folgenden Codebeispiel definieren wir eine Variable `w1`, dann einen Operator `init_op`, gefolgt von der Variablen `w2`:

```
>>> import tensorflow as tf
>>>
>>> g2 = tf.Graph()
>>>
>>> with g2.as_default():
...     w1 = tf.Variable(1, name='w1')
...     init_op = tf.global_variables_initializer()
...     w2 = tf.Variable(2, name='w2')
```

Nun werten wir `w1` aus:

```
>>> with tf.Session(graph=g2) as sess:
...     sess.run(init_op)
```

```
...     print('w1:', sess.run(w1))
w1: 1
```

Das funktioniert tadellos. Und nun w2:

```
>>> with tf.Session(graph=g2) as sess:
...     sess.run(init_op)
...     print('w2:', sess.run(w2))
FailedPreconditionError
Attempting to use uninitialized value w2
[[Node: _retval_w2_0_0 = _Retval[T=DT_INT32,
index=0, _device="/ job:localhost(replica:0/task:0/cpu:0"] (w2)]]
```

Wie das Beispiel zeigt, verursacht die Ausführung des Graphen einen Fehler, weil w2 nicht via `sess.run(init_op)` initialisiert wurde und daher auch nicht ausgewertet werden konnte. Der Operator `init_op` wurde definiert, bevor w2 dem Graphen hinzugefügt wurde, deshalb wird w2 durch die Ausführung von `init_op` nicht initialisiert.

### 14.5.3 Geltungsbereich von Variablen

In diesem Abschnitt erörtern wir den Geltungsbereich von Variablen, einem wichtigen Konzept in TensorFlow, das sich insbesondere bei der Konstruktion der Berechnungsgraphen großer neuronaler Netze als nützlich erweist.

Mithilfe eines Geltungsbereichs (Scope) können wir Variablen in Untergruppen organisieren. Nach dem Erstellen eines Geltungsbereichs wird den darin erzeugten Operationen und Tensoren der Name dieses Geltungsbereichs vorangestellt. Geltungsbereiche können verschachtelt werden. Wenn beispielsweise zwei Subnetze vorhanden sind, die jeweils aus mehreren Schichten bestehen, können wir die beiden Geltungsbereiche 'net\_A' und 'net\_B' definieren. Die verschiedenen Schichten werden dann jeweils innerhalb eines dieser beiden Geltungsbereiche definiert.

Sehen wir uns also an, wie die Variablennamen im folgenden Codebeispiel aussehen:

```
>>> import tensorflow as tf
>>>
>>> g = tf.Graph()
>>>
>>> with g.as_default():
...     with tf.variable_scope('net_A'):
...         with tf.variable_scope('layer-1'):
...             w1 = tf.Variable(tf.random_normal(
```

```

...
        shape=(10,4)), name='weights')
...
    with tf.variable_scope('layer-2'):
...
        w2 = tf.Variable(tf.random_normal(
            shape=(20,10)), name='weights')
...
    with tf.variable_scope('net_B'):
...
        with tf.variable_scope('layer-1'):
...
            w3 = tf.Variable(tf.random_normal(
                shape=(10,4)), name='weights')
...
...
    print(w1)
...
    print(w2)
...
    print(w3)
<tf.Variable 'net_A/layer-1/weights:0' shape=(10, 4)
dtype=float32_ref>
<tf.Variable 'net_A/layer-2/weights:0' shape=(20, 10)
dtype=float32_ref>
<tf.Variable 'net_B/layer-1/weights:0' shape=(10, 4)
dtype=float32_ref>

```

Beachten Sie, dass den Variablenamen nun durch Schrägstriche (/) getrennt ihre verschachtelten Geltungsbereiche vorangestellt sind.

### Hinweis

Weitere Information über Geltungsbereiche von Variablen finden Sie in der Dokumentation unter [https://www.tensorflow.org/programmers\\_guide/variable\\_scope](https://www.tensorflow.org/programmers_guide/variable_scope) und [https://www.tensorflow.org/api\\_docs/python/tf/variable\\_scope](https://www.tensorflow.org/api_docs/python/tf/variable_scope).

#### 14.5.4 Wiederverwendung von Variablen

Nehmen wir an, dass wir ein ziemlich komplexes Modell eines neuronalen Netzes entwickeln, dessen Klassifizierer Eingabedaten aus mehreren Quellen erhält. Weiter gehen wir davon aus, dass die Daten  $(X_A, y_A)$  der Quelle A und die Daten  $(X_B, y_B)$  der Quelle B entstammen. In diesem Beispiel soll der Graph so ausgelegt sein, dass er die Daten von nur einer Quelle als Eingabetensor verwendet, um das Netz aufzubauen. Auf diese Weise können wir die Daten der anderen Quelle demselben Klassifizierer zuführen.

Im folgenden Beispiel nehmen wir an, dass die Daten der Quelle A einem Platzhalter zugeführt werden, und Quelle B ist die Ausgabe eines generierenden Netzes. Dieses Generatornetz wird durch den Aufruf der Funktion `build_generator` im Geltungsbereich `generator` realisiert. Dann fügen wir noch einen Klassifizierer hinzu, indem wir `build_classifier` im Geltungsbereich `classifier` aufrufen:

```
import tensorflow as tf
>>>
>>> #####
... ## Hilfsfunktionen ##
... #####
>>>
>>> def build_classifier(data, labels, n_classes=2):
...     data_shape = data.get_shape().as_list()
...     weights = tf.get_variable(name='weights',
...                               shape=(data_shape[1],
...                                      n_classes),
...                               dtype=tf.float32)
...     bias = tf.get_variable(name='bias',
...                           initializer=tf.zeros(shape=n_classes))
...     logits = tf.add(tf.matmul(data, weights),
...                    bias, name='logits')
...     return logits, tf.nn.softmax(logits)
>>>
>>> def build_generator(data, n_hidden):
...     data_shape = data.get_shape().as_list()
...     w1 = tf.Variable(
...         tf.random_normal(shape=(data_shape[1],
...                            n_hidden),
...                          name='w1'))
...     b1 = tf.Variable(tf.zeros(shape=n_hidden),
...                     name='b1')
...     hidden = tf.add(tf.matmul(data, w1), b1,
...                    name='hidden_pre-activation')
...     hidden = tf.nn.relu(hidden, 'hidden_activation')
...
...     w2 = tf.Variable(
...         tf.random_normal(shape=(n_hidden,
...                               data_shape[1]),
...                          name='w2'))
...     b2 = tf.Variable(tf.zeros(shape=data_shape[1]),
...                     name='b2')
...     output = tf.add(tf.matmul(hidden, w2), b2,
...                    name='output')
...     return output, tf.nn.sigmoid(output)
>>>
>>> #####
... ## Graphen erstellen ##
>>> #####
>>>
>>> batch_size=64
>>> g = tf.Graph()
>>>
```

```
>>> with g.as_default():
...     tf_X = tf.placeholder(shape=(batch_size, 100),
...                           dtype=tf.float32,
...                           name='tf_X')
...
...     ## Generator erstellen
...     with tf.variable_scope('generator'):
...         gen_out1 = build_generator(data=tf_X,
...                                    n_hidden=50)
...
...
...     ## Klassifizierer erstellen
...     with tf.variable_scope('classifier') as scope:
...         ## Klassifizierer für die ursprünglichen Daten:
...         cls_out1 = build_classifier(data=tf_X,
...                                     labels=tf.ones(
...                                         shape=batch_size))
...
...
...         ## Wiederverwenden des Klassifizierers für
...         ## die generierten Daten
...         scope.reuse_variables()
...         cls_out2 = build_classifier(data=gen_out1[1],
...                                     labels=tf.zeros(
...                                         shape=batch_size))
```

Beachten Sie hier, dass wir die Funktion `build_classifier` zwei Mal aufrufen. Der erste Aufruf sorgt für den Aufbau des Netzes. Anschließend wird `scope.reuse_variables()` und dann erneut `build_classifier` aufgerufen. Das hat zum Ergebnis, dass der zweite Aufruf keine neuen Variablen erzeugt, sondern dieselben Variablen wiederverwendet. Alternativ könnten wir die Variablen auch durch Angabe des Parameters `reuse=True` wiederverwenden:

```
>>> g = tf.Graph()
>>>
>>> with g.as_default():
...     tf_X = tf.placeholder(shape=(batch_size, 100),
...                           dtype=tf.float32,
...                           name='tf_X')
...
...     ## Generator erstellen
...     with tf.variable_scope('generator'):
...         gen_out1 = build_generator(data=tf_X,
...                                    n_hidden=50)
...
...
...     ## Klassifizierer erstellen
...     with tf.variable_scope('classifier'):
...         ## Klassifizierer für die ursprünglichen Daten:
...         cls_out1 = build_classifier(data=tf_X,
...                                     labels=tf.ones(
...                                         shape=batch_size))
```

```
...
...     with tf.variable_scope('classifier', reuse=True):
...         ## Wiederverwenden des Klassifizierers für
...         ## die generierten Daten
...         cls_out2 = build_classifier(data=gen_out1[1],
...                                     labels=tf.zeros(
...                                         shape=batch_size))
```

**Tipp**

Wir haben zwar erörtert, wie Berechnungsgraphen und Variablen in TensorFlow definiert werden, aber eine ausführliche Darlegung der Berechnung von Gradienten in einem Berechnungsgraphen geht über den Rahmen des Buches hinaus, denn hier verwenden wir TensorFlows komfortable Optimierer-Klassen, die das Backpropagation-Verfahren automatisch erledigen. Wenn Sie mehr über die Berechnung von Gradienten in Graphen und die verschiedenen Methoden, sie in TensorFlow zu berechnen, wissen möchten, sollten Sie sich den PyData-Vortrag von Sebastian Raschka ansehen: <https://github.com/rasbt/pydata-ann Arbor2017-dl-tutorial>.

## 14.6 Erstellen eines Regressionsmodells

Nun haben wir Platzhalter und Variablen erkundet und können als Beispiel ein Modell für eine Regressionsanalyse erstellen, das dem aus Kapitel 13 ähnelt. Ziel ist es, ein lineares Regressionsmodell zu implementieren:  $\hat{y} = wx + b$ .

Bei diesem einfachen Regressionsmodell sind  $w$  und  $b$  die beiden Parameter, die als Variablen definiert werden müssen.  $x$  ist die Eingabe für das Modell und kann als Platzhalter definiert werden. Außerdem benötigen wir für das Trainieren dieses Modells eine Straffunktion. Wir verwenden hier die in Kapitel 10 definierte *mittlere quadratische Abweichung (MSE, Mean Squared Error)* als Straffunktion:

$$MSE = \frac{1}{n} \sum_{i=1}^n \left( y^{(i)} - \hat{y}^{(i)} \right)^2$$

Hier bezeichnet  $y$  den tatsächlichen Wert, der dem Modell beim Training als Eingabe übergeben wird, daher müssen wir  $y$  ebenfalls als Platzhalter definieren.  $\hat{y}$  ist die Vorhersage, die mit den TensorFlow-Operationen `tf.matmul` und `tf.add` berechnet wird. Wie Sie wissen, liefern TensorFlow-Operationen null oder mehr Tensoren zurück. `tf.matmul` und `tf.add` geben jeweils einen Tensor zurück.

Wir könnten auch den überladenen »+«-Operator zum Addieren zweier Tensoren verwenden, `tf.add` hat jedoch den Vorteil, dass wir über den Parameter `name` eine zusätzliche Bezeichnung für den resultierenden Tensor angeben können.

Fassen wir also alle beteiligten Tensoren mit ihren mathematischen Schreibweisen und den Bezeichnungen im Code noch einmal zusammen:

- Eingabe  $x$ : `tf_x` wird als Platzhalter definiert.
- Eingabe  $y$ : `tf_y` wird als Platzhalter definiert.
- Modellparameter  $w$ : `weight` wird als Variable definiert.
- Modellparameter  $b$ : `bias` wird als Variable definiert.
- Modellausgabe  $\hat{y}$ : `y_hat` wird von den TensorFlow-Operationen zur Berechnung der Vorhersage anhand des Regressionsmodells zurückgegeben.

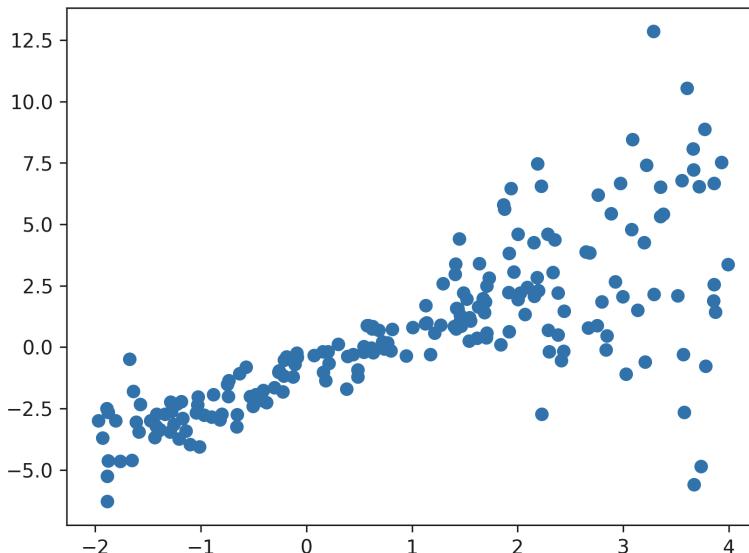
Hier die Implementierung des einfachen Regressionsmodells:

```
>>> import tensorflow as tf
>>> import numpy as np
>>>
>>> g = tf.Graph()
>>>
>>> with g.as_default():
...     tf.set_random_seed(123)
...     ## Platzhalter
...     tf_x = tf.placeholder(shape=(None),
...                           dtype=tf.float32,
...                           name='tf_x')
...     tf_y = tf.placeholder(shape=(None),
...                           dtype=tf.float32,
...                           name='tf_y')
...
...     ## Variablen definieren (Modellparameter)
...     weight = tf.Variable(
...         tf.random_normal(
...             shape=(1, 1),
...             stddev=0.25),
...         name='weight')
...     bias = tf.Variable(0.0, name='bias')
...
...     ## Modell erstellen
...     y_hat = tf.add(weight * tf_x, bias, name='y_hat')
...
...     ## Straffunktion berechnen
...     cost = tf.reduce_mean(tf.square(tf_y - y_hat),
...                          name='cost')
...
...     ## Modell trainieren
...     optim = tf.train.GradientDescentOptimizer(
...             learning_rate=0.001)
...     train_op = optim.minimize(cost, name='train_op')
```

Nachdem der Graph nun erstellt ist, besteht der nächste Schritt darin, eine Sitzung zum Starten des Graphen zu eröffnen und das Modell zu trainieren. Bevor wir fortfahren, betrachten wir, wie Tensoren ausgewertet und Operationen ausgeführt werden. Mit der Funktion `make_random_data` werden zufällige Regressionsdaten mit einem Merkmal erzeugt und anschließend visualisiert:

```
>>> ## Erzeugen zufälliger Beispieldaten für die Regression
>>>
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> np.random.seed(0)
>>>
>>> def make_random_data():
...     x = np.random.uniform(low=-2, high=4, size=200)
...     y = []
...     for t in x:
...         r = np.random.normal(loc=0.0,
...                             scale=(0.5 + t*t/3),
...                             size=None)
...         y.append(r)
...     return x, 1.726*x -0.84 + np.array(y)
>>>
>>> x, y = make_random_data()
>>> plt.plot(x, y, 'o')
>>> plt.show()
```

Die folgende Abbildung zeigt die zufällig generierten Regressionsdaten:



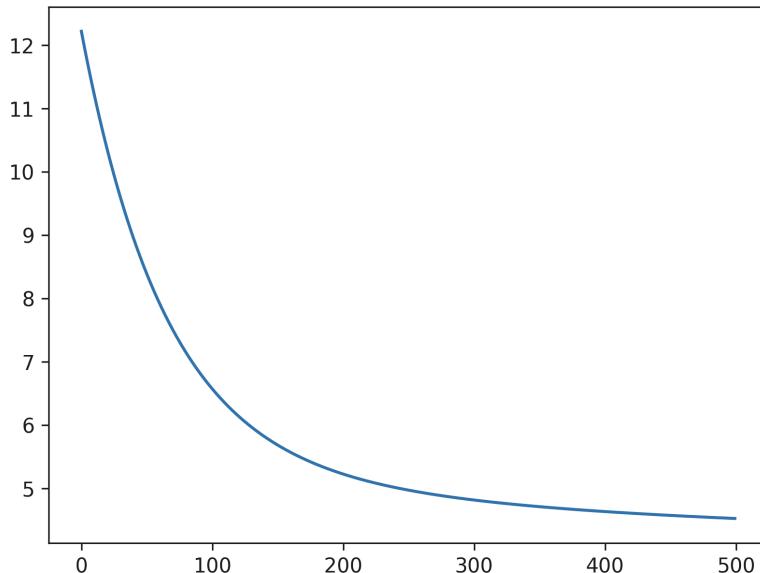
Jetzt ist alles bereit, um das Modell zu trainieren. Zunächst erstellen wir ein Sitzungsobjekt namens `sess`. Anschließend sollen die Variablen initialisiert werden, was mit `sess.run(tf.global_variables_initializer())` erfolgt. Nun können wir eine `for`-Schleife erstellen, die den Trainings-Operator ausführt und gleichzeitig die Straffunktion berechnet.

Die Ausführung eines Operators und die Auswertung eines Tensors wird in einem Aufruf der Methode `sess.run` kombiniert:

```
>>> ## Training/Test-Aufteilung
>>> x_train, y_train = x[:100], y[:100]
>>> x_test, y_test = x[100:], y[100:]
>>> n_epochs = 500
>>> training_costs = []
>>> n_epochs = 500
>>> training_costs = []
>>> with tf.Session(graph=g) as sess:
...     ## Variablen initialisieren
...     sess.run(tf.global_variables_initializer())
...
...     ## Modell n_epochs lang trainieren
...     for e in range(n_epochs):
...         c, _ = sess.run([cost, train_op],
...                       feed_dict={tf_x: x_train,
...                                  tf_y: y_train})
...         training_costs.append(c)
...         if not e % 50:
...             print('Epoche %d: %.4f' % (e, c))
Epoche 0: 12.2230
Epoche 50: 8.3876
Epoche 100: 6.5721
Epoche 150: 5.6844
Epoche 200: 5.2269
Epoche 250: 4.9725
Epoche 300: 4.8169
Epoche 350: 4.7119
Epoche 400: 4.6347
Epoche 450: 4.5742

plt.plot(training_costs)
plt.show()
```

Der Code erzeugt das folgende Diagramm, das den Wert der Straffunktion nach jeder Epoche darstellt:



## 14.7 Ausführung von Objekten in einem TensorFlow-Graphen unter Verwendung ihres Namens

Variablen und Operatoren anhand ihres Namens ausführen zu können, ist unter vielen Umständen äußerst nützlich. Wenn wir beispielsweise ein Modell als separates Modul entwickeln, sind die Variablen gemäß Pythons Regeln für Geltungsbereiche in den übrigen Geltungsbereichen nicht verfügbar. Bei einem Graphen können wir jedoch die anderen Knoten des Graphen unter Verwendung ihres Namens ausführen.

Im letzten Codebeispiel kann das leicht erreicht werden, indem man die `sess.run`-Methode dahin gehend ändert, dass statt der Python-Variablen `cost` der Variablenname `cost` des Graphen verwendet wird, indem man `sess.run([cost, train_op], ...)` zu `sess.run(['cost:0', 'train_op'], ...)` ändert.

```
>>> n_epochs = 500
>>> training_costs = []
>>> with tf.Session(graph=g) as sess:
...     ## Variablen initialisieren
...     sess.run(tf.global_variables_initializer())
...     ## Modell n_epochs lang trainieren
...     for e in range(n_epochs):
...         c, _ = sess.run(['cost:0', 'train_op'],
...                       feed_dict={'tf_x:0':x_train,
...                                 'tf_y:0':y_train})
```

```

...
    training_costs.append(c)
...
    if e%50 == 0:
        print('Epoch {:_d} : {:.4f}'
              .format(e, c))
...

```

Die Auswertung von `cost` erfolgt anhand des Namens, der '`cost:0`' lautet, und auch der Trainings-Operator wird unter Verwendung seines Namens '`train_op`' ausgeführt. Außerdem wird im `feed_dict` statt `tf_x: x_train` jetzt '`tf_x:0: x_train`' verwendet.

### Hinweis

Achten Sie auf die Namen der Tensoren: TensorFlow fügt den Namen der Tensoren das Suffix '`:0`' hinzu.

Die Namen der Operatoren besitzen ein solches Suffix allerdings nicht. Wenn ein Tensor mit einem bestimmten Namen, wie z.B. '`my_tensor`', erzeugt wird, hängt TensorFlow das Suffix '`:0`' an und der Tensor heißt dann '`my_tensor:0`'.

Wenn wir nun versuchen, im selben Graphen weitere Tensoren mit diesem Namen zu erzeugen, hängt TensorFlow '`_1:0`', '`_2:0`' usw. an den Namen an, sodass die neuen Tensoren die Namen '`my_tensor_1:0`', '`my_tensor_2:0`' usw. tragen. Bei dieser Namensgebung wird davon ausgegangen, dass wir nicht versuchen, bereits vorhandene Tensoren wiederzuverwenden.

## 14.8 Speichern und wiederherstellen eines Modells in TensorFlow

Im letzten Abschnitt haben wir einen Graphen erzeugt und ein Modell trainiert. Wie wäre es damit, die Vorhersage anhand der zurückgehaltenen Testdatenmenge durchzuführen? Das Problem dabei ist, dass wir die Modellparameter nicht gespeichert haben – sobald die Ausführung der Anweisungen abgeschlossen ist und wir die `tf.Session`-Umgebung verlassen, gehen alle Variablen verloren und ihr reservierter Speicherplatz wird freigegeben.

Eine mögliche Lösung wäre es, das Modell zu trainieren und ihm nach dem Abschluss des Trainings sofort die Testdatenmenge zuzuführen. Dieser Ansatz ist allerdings nicht besonders sinnvoll, denn das Trainieren tiefer neuronaler Netze dauert typischerweise mehrere Stunden, Tage oder sogar Wochen.

Der beste Ansatz ist es, das trainierte Modell zwecks zukünftiger Verwendung zu speichern. Dazu müssen wir dem Graphen einen weiteren Knoten hinzufügen, und zwar eine Instanz der Klasse `tf.train.Saver`, die wir kurz `saver` nennen. Mit der folgenden Anweisung können einem bestimmten Graphen weitere Kno-

ten hinzugefügt werden. In diesem Fall wird dem Graphen `g` der Knoten `saver` hinzugefügt:

```
>>> with g.as_default():
...     saver = tf.train.Saver()
```

Als Nächstes trainieren wir das Modell erneut und rufen zusätzlich die Funktion `saver.save()` auf, um das Modell zu speichern:

```
>>> n_epochs = 500
>>> training_costs = []
>>> with tf.Session(graph=g) as sess:
...     sess.run(tf.global_variables_initializer())
...
...     ## Modell n_epochs lang trainieren
...     for e in range(n_epochs):
...         c, _ = sess.run([cost, train_op],
...                       feed_dict={tf_x: x_train,
...                                 tf_y: y_train})
...         training_costs.append(c)
...         if not e % 50:
...             print('Epoch %4d: %.4f' % (e, c))
...     saver.save(sess, './trainiertes-Modell')
```

Die Ausführung der neuen Anweisung erzeugt drei Dateien mit den Dateiendungen `.data`, `.index` und `.meta`. TensorFlow verwendet Protocol Buffers (<https://developers.google.com/protocol-buffers/>), eine sprachunabhängige Methode zur Serialisierung strukturierter Daten.

Zum Wiederherstellen eines trainierten Modells sind zwei Schritte erforderlich:

1. Neuerstellung eines Graphen, der dieselben Knoten und Namen beinhaltet wie das gespeicherte Modell
2. Wiederherstellung der gespeicherten Variablen in einer neuen `tf.Session`-Umgebung

Im ersten Schritt könnten wir die ursprünglichen Anweisungen erneut ausführen und so den Graphen `g` erzeugen. Es gibt jedoch eine viel einfachere Möglichkeit, denn alle den Graphen beschreibenden Informationen sind ja als Metadaten in der Datei mit der Endung `.meta` gespeichert. Der folgende Code rekonstruiert den Graphen durch den Import der Metadatendatei:

```
>>> with tf.Session() as sess:
...     new_saver = tf.train.import_meta_graph(
...                 './trainiertes-Modell.meta')
```

Die Funktion `tf.train.import_meta_graph` rekonstruiert nun den in der Datei '`./trainiertes-Modell.meta`' gespeicherten Graphen. Nach der Rekonstruktion des Graphen können wir das `new_saver`-Objekt zur Wiederherstellung der Modellparameter in der letzten Sitzung verwenden und das Modell ausführen. Hier ist der vollständige Code zur Ausführung des Modells mit einer Testdatenmenge:

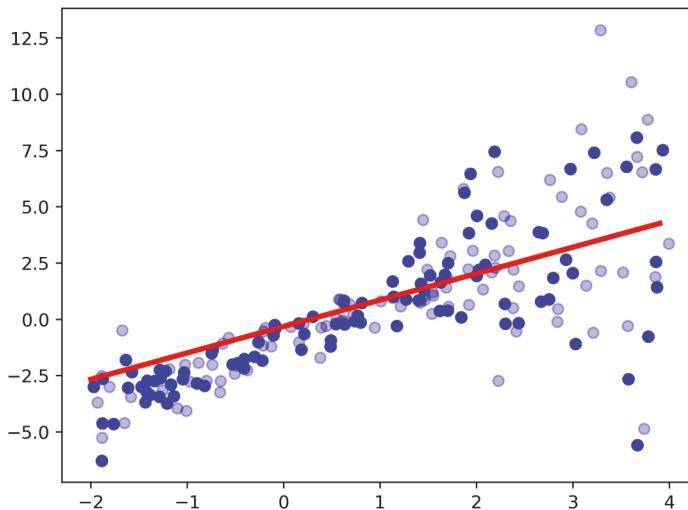
```
>>> import tensorflow as tf
>>> import numpy as np
>>>
>>> g2 = tf.Graph()
>>> with tf.Session(graph=g2) as sess:
...     new_saver = tf.train.import_meta_graph(
...             './trainiertes-Modell.meta')
...     new_saver.restore(sess, './trainiertes-Modell')
...     y_pred = sess.run('y_hat:0',
...                       feed_dict={'tf_x:0': x_test})
```

Beachten Sie, dass die Auswertung des Tensors  $\hat{y}$  anhand des vorher zugewiesenen Namens '`y_hat:0`' erfolgt. Zudem müssen wir dem Platzhalter `tf_x` Werte zuführen, wobei auch hier der Name verwendet wird, nämlich '`tf_x:0`'. In diesem Fall ist es nicht nötig, den tatsächlichen Klassenbezeichnungen  $y$  Werte zuzuführen, denn die Ausführung des `y_hat`-Knotens ist in dem erstellten Berechnungsgraphen unabhängig von `tf_y`.

Nun visualisieren wir die Vorhersage wie folgt:

```
>>> import matplotlib.pyplot as plt
>>>
>>> x_arr = np.arange(-2, 4, 0.1)
>>>
>>> g2 = tf.Graph()
>>> with tf.Session(graph=g2) as sess:
...     new_saver = tf.train.import_meta_graph(
...             './trainiertes-Modell.meta')
...     new_saver.restore(sess, './trainiertes-Modell')
...
...     y_arr = sess.run('y_hat:0',
...                     feed_dict={'tf_x:0' : x_arr})
>>>
>>> plt.figure()
>>> plt.plot(x_train, y_train, 'bo')
>>> plt.plot(x_test, y_test, 'bo', alpha=0.3)
>>> plt.plot(x_arr, y_arr.T[:, 0], '-r', lw=3)
>>> plt.show()
```

Die Abbildung zeigt das resultierende Diagramm, in dem sowohl Trainings- als auch Testdaten dargestellt sind:



In der Trainingsphase großer Modelle wird das Speichern und Wiederherstellen von Modellen ziemlich häufig eingesetzt, denn das Trainieren solcher Modelle kann mehrere Stunden oder Tage dauern, und auf diese Weise kann die Trainingsphase in mehrere kleinere Abschnitte aufgeteilt werden. Wenn beispielsweise 100 Epochen trainiert werden sollen, kann das Training in 25 Teilaufgaben aufgeteilt werden, in denen vier Epochen verarbeitet werden. Das trainierte Modell wird jeweils gespeichert und vor der Ausführung der nächsten Teilaufgabe wiederhergestellt.

## 14.9 Tensoren als mehrdimensionale Datenarrays transformieren

In diesem Abschnitt erkunden wir eine Reihe von Operatoren, die zum Transformieren von Tensoren verwendet werden können. Einige davon sind den NumPy-Transformationen sehr ähnlich. Da wir es jedoch mit Tensoren zu tun haben, deren Rang größer als 2 ist, müssen wir bei der Verwendung solcher Transformationen, wie z.B. dem transponierten Tensor, große Sorgfalt walten lassen.

In NumPy wird das Attribut `arr.shape` verwendet, um die Form eines NumPy-Arrays abzufragen. In TensorFlow gibt es stattdessen die Funktion `tf.get_shape`:

```
>>> import tensorflow as tf
>>> import numpy as np
>>>
>>> g = tf.Graph()
>>> with g.as_default():
```

```

...     arr = np.array([[1., 2., 3., 3.5],
...                     [4., 5., 6., 6.5],
...                     [7., 8., 9., 9.5]])
...     T1 = tf.constant(arr, name='T1')
...     print(T1)
...     s = T1.get_shape()
...     print('T1 hat die Form', s)
...     T2 = tf.Variable(tf.random_normal(shape=s))
...     print(T2)
...     T3 = tf.Variable(tf.random_normal(
...                         shape=(s.as_list()[0],)))
...     print(T3)

```

Der Code erzeugt folgende Ausgabe:

```

Tensor("T1:0", shape=(3, 4), dtype=float64)
T1 hat die Form (3, 4)
<tf.Variable 'Variable:0' shape=(3, 4) dtype=float32_ref>
<tf.Variable 'Variable_1:0' shape=(3,) dtype=float32_ref>

```

Beachten Sie, dass T2 anhand von s erstellt wird, es ist aber nicht möglich, Slicing oder einen Index zu verwenden, um T3 zu erzeugen. Deshalb muss s zunächst durch s.as\_list() in eine reguläre Python-Liste konvertiert werden, bevor die normalen Konventionen der Indizierung anwendbar sind.

Nun betrachten wir die Umformung von Tensoren. In NumPy gibt es zu diesem Zweck `np.reshape` oder `arr.reshape`. In TensorFlow wird zum Umformen eines Tensors die Funktion `tf.reshape` verwendet. Wie in NumPy kann einer der Dimensionen der Wert -1 zugewiesen werden, sodass die Größe dieser Dimension aus der Gesamtgröße des Arrays und den übrigen angegebenen Dimensionen abgeleitet wird.

Der folgende Code formt den Tensor T1 in die Tensoren T4 und T5 um, die beide vom Rang 3 sind:

```

>>> with g.as_default():
...     T4 = tf.reshape(T1, shape=[1, 1, -1], name='T4')
...     print(T4)
...     T5 = tf.reshape(T1, shape=[1, 3, -1], name='T5')
...     print(T5)

```

Hier die Ausgabe:

```

Tensor("T4:0", shape=(1, 1, 12), dtype=float64)
Tensor("T5:0", shape=(1, 3, 4), dtype=float64)

```

Als Nächstes geben wir die Elemente von T4 und T5 aus:

```
>>> with tf.Session(graph = g) as sess:
...     print(sess.run(T4))
...     print()
...     print(sess.run(T5))
[[[ 1.  2.  3.  3.5  4.  5.  6.  6.5  7.  8.  9.  9.5]]]

[[[ 1.  2.  3.  3.5]
 [ 4.  5.  6.  6.5]
 [ 7.  8.  9.  9.5]]]
```

Wie Sie wissen, gibt es in NumPy drei Möglichkeiten, ein Array zu transponieren: `arr.T`, `arr.transpose()` und `np.transpose(arr)`. In TensorFlow verwenden wir stattdessen die Funktion `tf.transpose`, die neben der regulären Transponierung noch die Möglichkeit bietet, mit `perm=[...]` eine beliebige Reihenfolge der Dimensionen anzugeben. Hier ein Beispiel:

```
>>> with g.as_default():
...     T6 = tf.transpose(T5, perm=[2, 1, 0], name='T6')
...     print(T6)
...     T7 = tf.transpose(T5, perm=[0, 2, 1], name='T7')
...     print(T7)
Tensor("T6:0", shape=(4, 3, 1), dtype=float64)
Tensor("T7:0", shape=(1, 4, 3), dtype=float64)
```

Als Nächstes teilen wir mit der `tf.split`-Funktion einen Tensor in eine Liste von Subtensoren auf:

```
>>> with g.as_default():
...     t5_splt = tf.split(T5, num_or_size_splits=2,
...                         axis=2, name='T8')
...     print(t5_splt)
[<tf.Tensor 'T8:0' shape=(1, 3, 2) dtype=float64>,
 <tf.Tensor 'T8:1' shape=(1, 3, 2) dtype=float64>]
```

Hier ist es wichtig zu beachten, dass die Ausgabe kein Tensorobjekt ist, sondern eine Liste von Tensoren mit den Namen '`T8:0`' und '`T8:1`'.

Die Verkettung mehrerer Tensoren ist eine weitere nützliche Transformation. Eine Liste von Tensoren gleicher Form und gleichen Datentyps (`dtype`) kann mit der Funktion `tf.concat` zu einem großen Tensor kombiniert werden. Der folgende Code zeigt ein Beispiel:

```
>>> g = tf.Graph()
>>> with g.as_default():
...     t1 = tf.ones(shape=(10, 1),
...                  dtype=tf.float32, name='t1')
...     t2 = tf.zeros(shape=(10, 1),
...                  dtype=tf.float32, name='t2')
...     print(t1)
...     print(t2)
>>> with g.as_default():
...     t3 = tf.concat([t1, t2], axis=0, name='t3')
...     print(t3)
...     t4 = tf.concat([t1, t2], axis=1, name='t4')
...     print(t4)

Tensor("t1:0", shape=(5, 1), dtype=float32)
Tensor("t2:0", shape=(5, 1), dtype=float32)
Tensor("t3:0", shape=(10, 1), dtype=float32)
Tensor("t4:0", shape=(5, 2), dtype=float32)
```

Nun geben wir die verketteten Tensoren aus:

```
>>> with tf.Session(graph=g) as sess:
...     print(t3.eval())
...     print()
...     print(t4.eval())
[[ 1.]
 [ 1.]
 [ 1.]
 [ 1.]
 [ 1.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]]]

[[ 1.  0.]
 [ 1.  0.]
 [ 1.  0.]
 [ 1.  0.]
 [ 1.  0.]]]
```

## 14.10 Mechanismen der Flusskontrolle beim Erstellen von Graphen verwenden

TensorFlow bietet einen sehr interessanten Mechanismus zur Entscheidungsfindung beim Erstellen eines Graphen. Allerdings zeigen sich beim Vergleich von Python's Anweisungen zur Flusskontrolle und TensorFlows Funktionen der Flusskontrolle bei der Erstellung eines Graphen einige subtile Unterschiede.

Zur Veranschaulichung dieser Unterschiede durch einige einfache Codebeispiele betrachten wir die Implementierung der folgenden Gleichung in TensorFlow:

$$res = \begin{cases} x + y & \text{wenn } x < y \\ x - y & \text{anderenfalls} \end{cases}$$

Der folgende Code verwendet Python's `if`-Anweisung auf naive Weise zum Erstellen eines Graphen, der diese Gleichung abbildet:

```
>>> import tensorflow as tf
>>>
>>> x, y = 1.0, 2.0
>>>
>>> g = tf.Graph()
>>> with g.as_default():
...     tf_x = tf.placeholder(dtype=tf.float32,
...                           shape=None, name='tf_x')
...     tf_y = tf.placeholder(dtype=tf.float32,
...                           shape=None, name='tf_y')
...
...     if x < y:
...         res = tf.add(tf_x, tf_y, name='result_add')
...     else:
...         res = tf.subtract(tf_x, tf_y, name='result_sub')
...
...     print('Objekt: ', res)
>>>
>>> with tf.Session(graph=g) as sess:
...     print('x < y: %s -> Resultat:' % (x < y),
...           res.eval(feed_dict={'tf_x:0': x, 'tf_y:0': y}))
...     x, y = 2.0, 1.0
...     print('x < y: %s -> Resultat:' % (x < y),
...           res.eval(feed_dict={'tf_x:0': x, 'tf_y:0': y}))
```

Der Code liefert folgendes Resultat:

```
Objekt: Tensor("result_add:0", dtype=float32)
x < y: True -> Resultat: 3.0
x < y: False -> Resultat: 3.0
```

Wie Sie sehen, ist das `res`-Objekt ein Tensor namens '`result_add:0`'. Es ist sehr wichtig, zu verstehen, dass der Berechnungsgraph bei diesem Mechanismus nur einen Zweig besitzt, der zum Additionsoperator gehört. Der Subtraktionsoperator wird gar nicht aufgerufen.

Der TensorFlow-Berechnungsgraph ist statisch, und das bedeutet, dass er, wenn er einmal erstellt ist, während der Ausführung unverändert bleibt. Wir könnten sogar die Werte von `x` und `y` ändern und die neuen Werte an den Graphen übergeben, die Tensoren würden trotzdem denselben Pfad im Graphen durchlaufen. Aus diesem Grund erfolgt in beiden Fällen,  $x=2$ ,  $y=1$  und  $x=1$ ,  $y=2$  die Ausgabe 3.0.

Nun betrachten wir den Mechanismus der Flusskontrolle in TensorFlow. Wir implementieren die Gleichung im folgenden Code nicht mit Python's `if`-Anweisung, sondern mit der Funktion `tf.cond`:

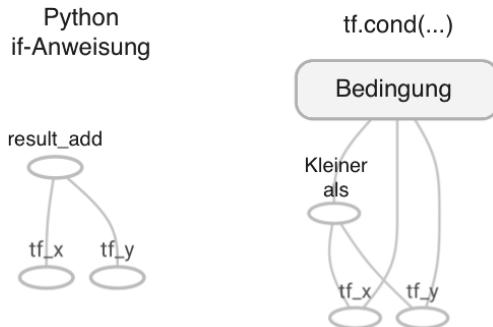
```
>>> import tensorflow as tf
>>>
>>> x, y = 1.0, 2.0
>>>
>>> g = tf.Graph()
>>> with g.as_default():
...     tf_x = tf.placeholder(dtype=tf.float32,
...                           shape=None, name='tf_x')
...     tf_y = tf.placeholder(dtype=tf.float32,
...                           shape=None, name='tf_y')
...     res = tf.cond(tf_x < tf_y,
...                   lambda: tf.add(tf_x, tf_y,
...                                 name='result_add'),
...                   lambda: tf.subtract(tf_x, tf_y,
...                                     name='result_sub'))
...     print('Objekt:', res)
...
...     with tf.Session(graph=g) as sess:
...         print('x < y: %s -> Resultat:' % (x < y),
...               res.eval(feed_dict={'tf_x:0': x, 'tf_y:0': y}))
...         x, y = 2.0, 1.0
...         print('x < y: %s -> Resultat:' % (x < y),
...               res.eval(feed_dict={'tf_x:0': x, 'tf_y:0': y}))
```

Hier ergibt sich Folgendes:

```
Objekt: Tensor("cond/Merge:0", dtype=float32)
x < y: True -> Result: 3.0
x < y: False -> Result: 1.0
```

Das `res`-Objekt heißt jetzt "cond/Merge:0". In diesem Fall besitzt der Berechnungsgraph zwei Zweige und verfügt über einen Mechanismus, um zu entscheiden, welchem Pfad zur Laufzeit zu folgen ist. Dementsprechend wird im Fall  $x=1$ ,  $y=2$  der Additionszweig mit dem Resultat 3.0 durchlaufen und der Subtraktionszweig mit der Ausgabe 1.0, wenn  $x=2$ ,  $y=1$  gilt.

In der folgenden Abbildung sind die unterschiedlichen Berechnungsgraphen der Implementierungen mit Python's `if`-Anweisung und TensorFlows `tf.cond`-Funktion dargestellt.



Neben `tf.cond` sind in TensorFlow eine Reihe weiterer Operatoren zur Flusskontrolle verfügbar, wie etwa `tf.case`, und `tf.loop`. `tf.case` ist beispielsweise das TensorFlow-Pendant zu Python's `if...else`-Anweisung. Betrachten Sie folgenden Python-Ausdruck:

```
if (x < y):
    result = 1
else:
    result = 0
```

Das `tf.case`-Äquivalent dieser Anweisung zur bedingten Ausführung kann in einem TensorFlow-Graphen folgendermaßen implementiert werden:

```
f1 = lambda: tf.constant(1)
f2 = lambda: tf.constant(0)
result = tf.case([(tf.less(x, y), f1)], default=f2)
```

Auf ähnliche Weise kann einem TensorFlow-Graphen eine `while`-Schleife hinzugefügt werden, die eine Variable `i` um 1 erhöht, bis ein Grenzwert (`threshold`) erreicht ist:

```
i = tf.constant(0)
threshold = 100
c = lambda i: tf.less(i, 100)
b = lambda i: tf.add(i, 1)
r = tf.while_loop(cond=c, body=b, loop_vars=[i])
```

Weitere Informationen über die verschiedenen Operatoren zur Flusskontrolle finden Sie in der offiziellen Dokumentation unter [https://www.tensorflow.org/api\\_guides/python/control\\_flow\\_ops](https://www.tensorflow.org/api_guides/python/control_flow_ops).

Vielleicht ist Ihnen aufgefallen, dass die Berechnungsgraphen von TensorBoard erzeugt werden. Damit ist der Zeitpunkt gekommen, im nächsten Abschnitt einen genaueren Blick darauf zu werfen.

## 14.11 Graphen mit TensorBoard visualisieren

TensorBoard ist ein TensorFlow-Modul zur Visualisierung sowohl des Graphen als auch des Lernens eines Modells. Die Visualisierung des Graphen zeigt die Verknüpfungen zwischen den Knoten, lässt die Abhängigkeiten der Knoten untereinander erkennen und dient bei Bedarf auch zum Debuggen des Modells.

Wir visualisieren nun ein bereits bekanntes Netz, das aus einem Generator und einem Klassifizierer besteht. Ein Teil des Codes wird Ihnen bekannt vorkommen, weil wir ihn schon in Hilfsfunktionen verwendet haben. Die Funktionsdefinitionen von `build_generator` und `build_classifier` können Sie in Abschnitt 14.5.4 (*Wiederverwendung von Variablen*) nachlesen. Zusammen mit diesen beiden Hilfsfunktionen erstellen wir einen Graphen:

```
>>> batch_size=64
>>> g = tf.Graph()
>>>
>>> with g.as_default():
...     tf_X = tf.placeholder(shape=(batch_size, 100),
...                           dtype=tf.float32,
...                           name='tf_X')
...
...     ## Generator erstellen
...     with tf.variable_scope('generator'):
...         gen_out1 = build_generator(data=tf_X,
...                                   n_hidden=50)
```

```

...
    ## Klassifizierer erstellen
    with tf.variable_scope('classifier') as scope:
        ...
        ## Klassifizierer der ursprünglichen Daten:
        cls_out1 = build_classifier(data=tf_X,
                                     labels=tf.ones(
                                         shape=batch_size))

        ...
        ## Wiederverwenden des Klassifizierers für
        ## die generierten Daten
        scope.reuse_variables()
        cls_out2 = build_classifier(data=gen_out1[1],
                                     labels=tf.zeros(
                                         shape=batch_size))

```

Beachten Sie, dass zum Erstellen des Graphen bislang keinerlei Änderungen erforderlich waren. Nachdem der Graph erstellt ist, kann er ganz unkompliziert visualisiert werden. Der folgende Code exportiert den Graphen zwecks Visualisierung:

```

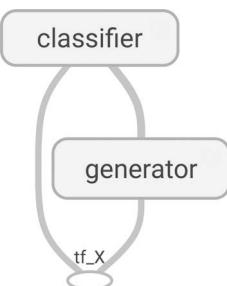
>>> with tf.Session(graph=g) as sess:
...     sess.run(tf.global_variables_initializer())
...
...     file_writer = tf.summary.FileWriter(
...         logdir='./logs/', graph=g)

```

Der Code erstellt ein neues Verzeichnis namens `logs/`. Nun muss nur noch in einem Linux- oder macOS-Terminal folgender Befehl ausgeführt werden:

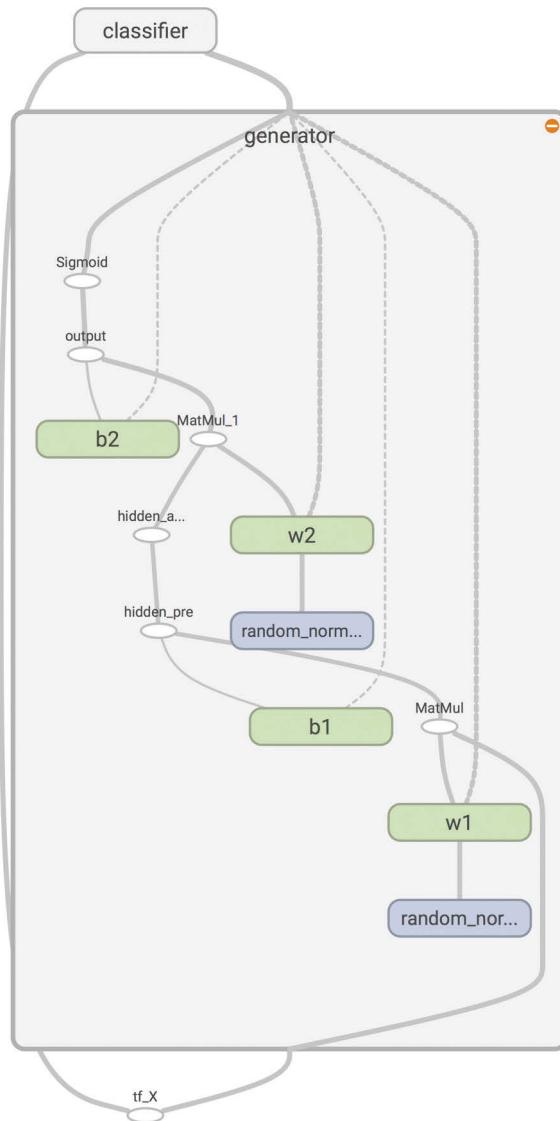
```
tensorboard -logdir logs/
```

Der Befehl gibt eine Meldung aus, bei der es sich um eine URL handelt. Kopieren Sie den Link, wie z.B. `http://localhost:6006/#graphs`, und fügen Sie ihn in der Adresszeile Ihres Webbrowsers ein. Nun sollte Ihnen der zu diesem Modell gehörende Graph angezeigt werden, wie in der folgenden Abbildung.

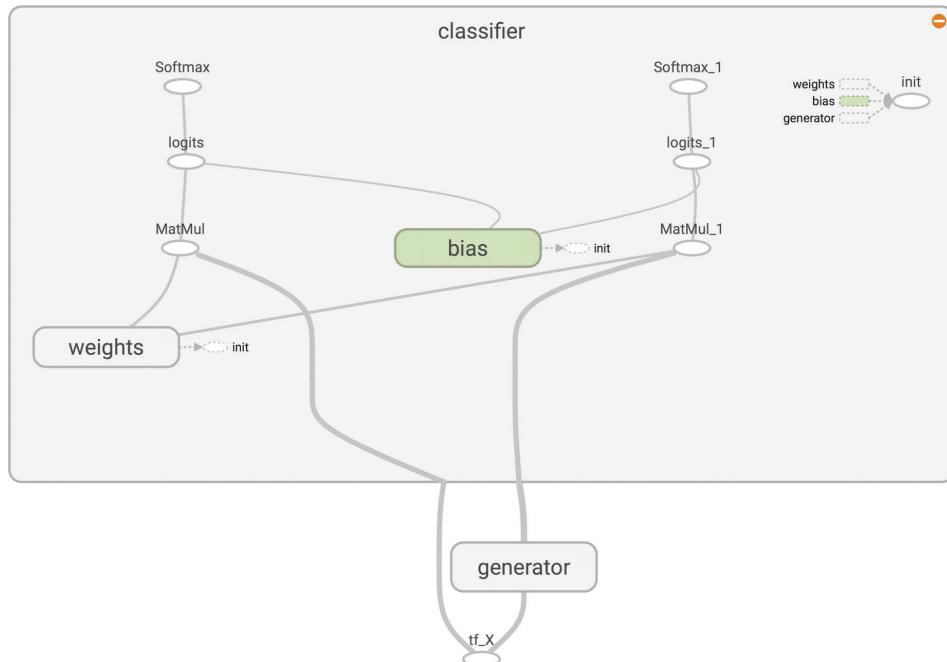


Die beiden großen Kästen mit abgerundeten Ecken stellen die beiden Subnetze dar, die wir erstellt haben: `generator` und `classifier`. Da wir beim Erstellen des Graphen die Funktion `tf.variable_scope` verwendet haben, gehören alle Komponenten dieser beiden Subnetze wie in der Abbildung zu diesen beiden Kästen.

Die Ansicht dieser Kästen lässt sich erweitern: Klicken Sie mit der Maus auf das Pluszeichen in der oberen rechten Ecke des `generator`-Kastens, um ihn zu erweitern. Nun werden die Details des `generator`-Subnetzes dargestellt (siehe Abbildung).



Bei genauerer Betrachtung dieses Graphen wird deutlich, dass er zwei Gewichtstensoren namens `w1` und `w2` besitzt. Erweitern Sie nun wie in der folgenden Abbildung das `classifier`-Subnetz.



Der Abbildung ist zu entnehmen, dass der Klassifizierer über zwei Eingabequellen verfügt. Die eine ist der Platzhalter `tf_X`, die andere die Ausgabe des `generator`-Subnetzes.

### 14.11.1 Erweitern Sie Ihre TensorBoard-Kenntnisse

Als interessante Übung sollten Sie die verschiedenen in diesem Kapitel implementierten Graphen mit TensorBoard visualisieren. Sie könnten beispielsweise ähnliche Schritte wie die vorgestellten zum Erstellen der Graphen verwenden und zusätzliche Linien zur Visualisierung hinzufügen. Sie können auch Visualisierungen für den Abschnitt über die Flusskontrolle erzeugen, die den Unterschied zwischen mit Python's `if`-Anweisung und der `tf.cond`-Funktion erstellten Graphen veranschaulichen.

Weitere Informationen und Beispiele für die Visualisierung von Graphen finden Sie auf der offiziellen TensorFlow-Website unter [https://www.tensorflow.org/get\\_started/graph\\_viz](https://www.tensorflow.org/get_started/graph_viz).

## 14.12 Zusammenfassung

In diesem Kapitel haben wir die wichtigsten Features und Konzepte von TensorFlow behandelt. Zunächst haben wir die bedeutendsten Merkmale und Vorteile sowie entscheidende TensorFlow-Konzepte wie Tensoren und deren Rang erörtert. Wir haben TensorFlows Berechnungsgraphen betrachtet und uns damit befasst, wie man einen Graphen in einer Sitzungsumgebung startet. Sie haben Platzhalter und Variablen kennengelernt und erfahren, welche Möglichkeiten zur Auswertung von Tensoren und zur Ausführung von Operatoren mit Python-Variablen oder anhand des Namens im Graphen zur Verfügung stehen.

Wir haben einige der wichtigsten Operatoren und Funktionen von TensorFlow zum Transformieren von Tensoren erkundet, wie z.B. `tf.transpose`, `tf.reshape`, `tf.split` und `tf.concat`. Und schließlich haben wir TensorFlow-Berechnungsgraphen mit TensorBoard visualisiert, einem Modul, das sich insbesondere beim Debuggen komplexer Modelle als sehr nützlich erweisen kann.

Im nächsten Kapitel werden wir von dieser Bibliothek Gebrauch machen, um einen erweiterten Bildklassifizierer zu implementieren: ein *konvolutionales neuronales Netz* (engl. *Convolutional Neural Network*, CNN). CNNs sind leistungsfähige Modelle, die besonders gut für die Bildklassifizierung und für maschinelles Sehen (Computer Vision) geeignet sind. Wir werden die elementaren Operationen in CNNs erörtern und mit TensorFlow ein tiefes konvolutionales Netz zur Bildklassifizierung implementieren.

# Bildklassifizierung mit tiefen konvolutionalen neuronalen Netzen

Im letzten Kapitel haben wir uns ausführlich mit verschiedenen Aspekten der TensorFlow-API befasst, uns mit Tensoren, Benennung von Variablen und Operatoren vertraut gemacht sowie gelernt, wie man Geltungsbereiche von Variablen verwendet. Dieses Kapitel hat *konvolutionale neuronale Netze* (*Convolutional Neural Networks*, CNNs) und deren Implementierung mit TensorFlow zum Thema. Wir werden eine solche tiefe neuronale Netzarchitektur zur Bildklassifizierung einsetzen.

Zunächst geht es um die grundlegenden Bausteine von CNNs, wir verwenden also einen Bottom-up-Ansatz. Anschließend werden wir uns eingehender mit der CNN-Architektur befassen und erörtern, wie man diese mit TensorFlow implementieren kann. Dabei kommen die folgenden Themen zur Sprache:

- Ein- und zweidimensionale konvolutionale Operationen
- Bausteine von CNN-Architekturen
- Implementierung von tiefen neuronalen Netzen mit TensorFlow

## 15.1 Bausteine konvolutionaler neuronaler Netze

Konvolutionale neuronale Netze, oder kurz CNNs, gehören zu einer Modellfamilie, die sich an der Funktionsweise der Sehrinde (visueller Kortex) des menschlichen Gehirns bei der Erkennung von Objekten orientiert.

Die Entwicklung von CNNs reicht zurück bis in die 1990er-Jahre. Damals schlugen Yann LeCun und seine Mitarbeiter eine neuartige neuronale Netzarchitektur zur Klassifizierung von handgeschriebenen Ziffern vor (Y. LeCun et al., *Handwritten Digit Recognition with a Back-Propagation Network*, 1989, veröffentlicht auf der Konferenz *Neural Information Processing Systems*).

Dank der bei Bildklassifizierungsaufgaben erzielten herausragenden Leistungen haben CNNs viel Aufmerksamkeit auf sich gezogen, was zu enormen Verbesserungen bei Anwendungen des Machine Learnings und des maschinellen Sehens geführt hat.

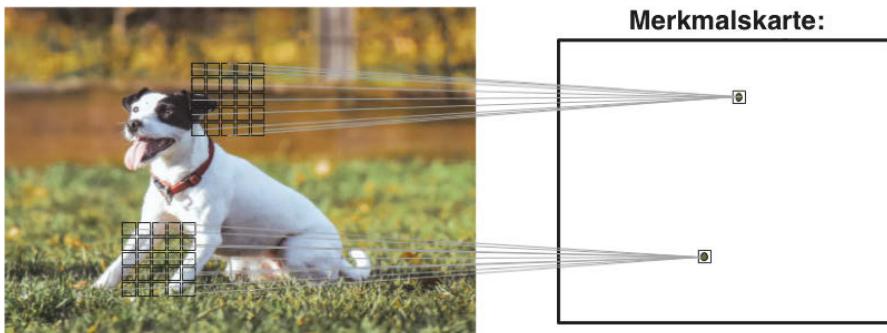
In den folgenden Abschnitten werden wir betrachten, wie sich CNNs zur Merkmalsextraktion einsetzen lassen, und die theoretische Definition der Faltung sowie die Berechnung der ein- und zweidimensionalen Faltung erörtern.

### 15.1.1 CNNs und Merkmalshierarchie

Das Extrahieren *auffälliger* (relevanter) *Merkmale* ist für die Leistung eines Lernalgorithmus entscheidend. Herkömmliche Lernalgorithmen beruhen auf Eingabemerkmalen, die anhand von Fachkenntnissen vorgegeben werden, oder verwenden Verfahren zur berechneten Merkmalsextraktion. Neuronale Netze sind in der Lage, die für eine bestimmte Aufgabe wesentlichen Merkmale anhand der Rohdaten automatisch zu erkennen. Aus diesem Grund werden neuronale Netze als ein Mechanismus zur Merkmalsextraktion betrachtet: Die ersten Schichten (die nach der Eingabeschicht folgenden) extrahieren die *Low-level-Merkmale*.

Mehrschichtige neuronale Netze und insbesondere tiefe neuronale Netze konstruieren eine sogenannte *Merkmalshierarchie*, indem die Low-level-Merkmale schichtweise zu High-level-Merkmalen kombiniert werden. Wenn es beispielsweise um Bilder geht, werden in den ersten Schichten Low-level-Merkmale wie Kanten oder Pixelanhäufungen extrahiert und zu High-level-Merkmalen kombiniert, die Objektformen darstellen, wie z.B. ein Gebäude, ein Auto oder einen Hund.

In der folgenden Abbildung ist dargestellt, wie ein CNN anhand eines Eingabebilds eine Merkmalskarte berechnet, in der die Elemente einer bestimmten Fläche von Pixeln entsprechen.



(Foto: Alexander Dummer/Unsplash)

Diese lokalen Pixelfenster werden als *rezeptive Felder* bezeichnet. Für gewöhnlich sind CNNs für Bildverarbeitungsaufgaben aufgrund zweier wichtiger Eigenschaften sehr gut geeignet:

- **Gezielte Verknüpfungen:** Die Elemente der Merkmalskarte sind nur mit einigen wenigen Pixeln verknüpft. (Dies ist ein großer Unterschied zur vollständigen Verknüpfung mit dem gesamten Bild wie beim Perzeptron. Sie können in Kapitel 12 nachlesen und vergleichen, wie ein vollständig mit dem Bild verknüpftes Netz implementiert wurde.)
- **Gemeinsame Parameter:** Für die verschiedenen Pixelfenster des Eingabebilds werden dieselben Gewichtungen verwendet.

Diese beiden Eigenschaften sind dafür verantwortlich, dass sich die Anzahl der Gewichtungen (Parameter) des Netzes drastisch reduziert. Zudem ist es einfacher, *auffällige* Merkmale zu erfassen. Intuitiv erscheint es sinnvoll, dass näher beieinanderliegende Pixel für einander vermutlich von größerer Bedeutung sind als weit voneinander entfernte.

CNNs bestehen typischerweise aus mehreren Konvolutionsschichten und Pooling-Schichten (Unterstichproben), denen vollständig verknüpfte Schichten folgen. Die vollständig verknüpften Schichten entsprechen im Wesentlichen einem mehrschichtigen Perzeptron, bei dem alle Eingabeeinheiten  $i$  mit allen Ausgabeeinheiten  $j$  und den Gewichtungen  $w_{ij}$  verknüpft sind (siehe Kapitel 12).

Beachten Sie, dass die Pooling-Schichten keine erlernbaren Parameter besitzen. So gibt es in Pooling-Schichten beispielsweise keine Gewichtungen und keine Bias-Einheiten, die in Konvolutionsschichten und vollständig verknüpften Schichten jedoch sehr wohl vorhanden sind.

In den folgenden Abschnitten werden wir Konvolutionsschichten und Pooling-Schichten sowie ihre Funktionsweise eingehender untersuchen. Zunächst betrachten wir eine eindimensionale Faltung und anschließend erörtern wir den typischen Anwendungsfall zweidimensionaler Bilder.

### 15.1.2 Diskrete Faltungen

Eine *diskrete Faltung* (oder einfach Faltung) ist eine grundlegende Operation in einem CNN, deshalb ist es wichtig, die Funktionsweise zu verstehen. In diesem Abschnitt geht es um die mathematische Definition und einige naive Algorithmen zur Berechnung der Faltungen zweier eindimensionaler Vektoren oder zweidimensionaler Matrizen.

Die Beschreibung an dieser Stelle dient ausschließlich dem Verständnis der Funktionsweise einer Faltung. Tatsächlich gibt es in Paketen wie TensorFlow sehr viel effizientere Implementierungen von Faltungsoperationen, wie Sie später in diesem Kapitel noch sehen werden.

#### Tipp

##### Mathematische Notation

In diesem Kapitel werden tiefgestellte Buchstaben für die Größe eines mehrdimensionalen Arrays verwendet; beispielsweise ist  $A_{n_1 \times n_2}$  ein zweidimensionales Array der Größe  $n_1 \times n_2$ . Zur Indizierung mehrdimensionaler Arrays verwenden wir eckige Klammern [ ].  $A[i, j]$  kennzeichnet also das Element mit dem Index  $i, j$  in der Matrix  $A$ . Für die Faltungsoperation zweier Vektoren oder Matrizen verwenden wir das Symbol »\*«, das nicht mit Python's Multiplikationsoperator \* zu verwechseln ist.

## Eindimensionale diskrete Faltung

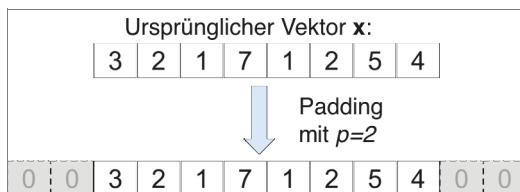
Zunächst einmal sind einige elementare Definitionen und die Festlegung der Schreibweise erforderlich. Die diskrete Faltung zweier eindimensionaler Vektoren  $\mathbf{x}$  und  $\mathbf{w}$  wird als  $\mathbf{y} = \mathbf{x} * \mathbf{w}$  notiert. Hier ist  $\mathbf{x}$  die Eingabe (die manchmal auch als *Signal* bezeichnet wird) und  $\mathbf{w}$  der *Filter* oder *Kernel*. Die diskrete Faltung ist mathematisch folgendermaßen definiert:

$$\mathbf{y} = \mathbf{x} * \mathbf{w} \rightarrow y[i] = \sum_{k=-\infty}^{+\infty} x[i-k]w[k]$$

Die eckigen Klammern  $[ ]$  kennzeichnen hier die Indizierung der Vektorelemente. Der Index  $i$  durchläuft alle Elemente des Ausgabevektors  $\mathbf{y}$ . In der Gleichung gibt es noch zwei Besonderheiten, die geklärt werden müssen: Die von  $-\infty$  bis  $+\infty$  laufenden Indizes und negative Indizes von  $\mathbf{x}$ .

Es erscheint merkwürdig, dass die Indizes der Summe von  $-\infty$  bis  $+\infty$  laufen, denn bei Anwendungen des Machine Learnings hat man es immer mit endlichen Merkmalsvektoren zu tun. Wenn  $\mathbf{x}$  beispielsweise 10 Merkmale mit den Indizes 0, 1, 2, ..., 8, 9 besitzt, sind die Indizes von  $-\infty$  bis -1 und von 10 bis  $+\infty$  gar nicht vorhanden. Um die Summe trotzdem korrekt berechnen zu können, nimmt man einfach an, dass  $\mathbf{x}$  und  $\mathbf{w}$  an den fehlenden Positionen lauter Nullen enthalten. Damit ergibt sich ein Ausgabevektor  $\mathbf{y}$  von unendlicher Größe, der ebenfalls sehr viele Nullen enthält. In der Praxis ist das allerdings nicht machbar, daher wird  $\mathbf{x}$  nur mit einer begrenzten Zahl von Nullen aufgefüllt.

Dieses Verfahren wird als *Zero-Padding* (mit Nullen auffüllen) oder einfach *Padding* bezeichnet. Die Anzahl der auf beiden Seiten aufgefüllten Nullen wird mit  $p$  bezeichnet. Die Abbildung zeigt als Beispiel das Padding eines eindimensionalen Vektors  $\mathbf{x}$ , mit  $p=2$ .



Nehmen wir an, die Eingabe  $\mathbf{x}$  und der Filter  $\mathbf{w}$  besitzen ursprünglich  $n$  bzw.  $m$  Elemente und es gilt  $m \leq n$ . Dann ist der mit Nullen aufgefüllte Vektor  $\mathbf{x}^p$  von der Größe  $n+2p$ . Als Gleichung für die diskrete Faltung ergibt sich dann:

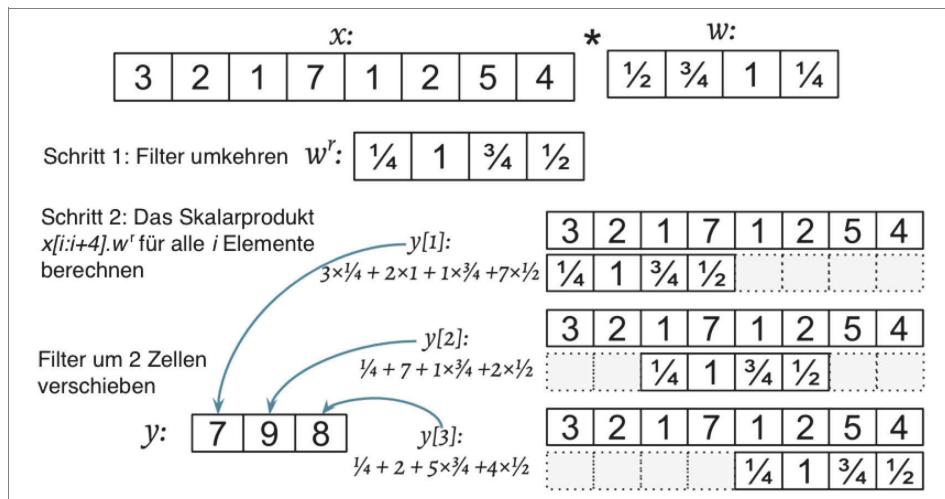
$$\mathbf{y} = \mathbf{x} * \mathbf{w} \rightarrow y[i] = \sum_{k=0}^{k=m-1} x^p[i+m-k]w[k]$$

Das Problem der unendlichen Indizes ist damit gelöst. Nun kommen wir zur zweiten Besonderheit, der Indizierung von  $\mathbf{x}$  mit  $i+m-k$ . Hier ist der entscheidende

Punkt, dass  $\mathbf{x}$  und  $\mathbf{w}$  bei der Summierung in gegensätzlicher Richtung indiziert werden. Aus diesem Grund kann einer der Vektoren  $\mathbf{x}$  oder  $\mathbf{w}$  nach dem Padding umgekehrt werden. Anschließend können wir einfach ihr Skalarprodukt berechnen.

Nehmen wir an, wir kehren den Filter  $\mathbf{w}$  um. Das ergibt den umgekehrten Filter  $\mathbf{w}^r$ . Nun berechnen wir das Skalarprodukt  $\mathbf{x}[i:i+m] \cdot \mathbf{w}^r$  und erhalten das Element  $\mathbf{y}[i]$ , wobei  $\mathbf{x}[i:i+m]$  ein Stück von  $\mathbf{x}$  der Größe  $m$  ist.

Diese Operation wird stückweise wiederholt, um alle Ausgabeelemente zu erhalten. Die folgende Abbildung zeigt ein Beispiel mit  $\mathbf{x}=(3, 2, 1, 7, 1, 2, 5, 4)$  und  $\mathbf{w}=\left(\frac{1}{2}, \frac{3}{4}, 1, \frac{1}{4}\right)$ , das die ersten drei Elemente der Ausgabe berechnet.



Im letzten Beispiel wurden keine Nullen aufgefüllt ( $p=0$ ). Der umgekehrte Filter  $\mathbf{w}^r$  wird jedes Mal um zwei Zellen verschoben. Diese Schrittweite  $s$  ist ein Hyperparameter der Faltung. Im Beispiel ist also  $s=2$ . Beachten Sie, dass die Schrittweite eine positive ganze Zahl und kleiner als die Größe des Eingabevektors sein muss. Im nächsten Abschnitt werden wir uns weiter mit Padding und Schrittweite befassen.

## Tipp

### Kreuzkorrelation

Die Kreuzkorrelation zwischen einem Eingabevektor und einem Filter wird als  $\mathbf{y} = \mathbf{x} * \mathbf{w}$  notiert und ist eine Art Geschwister der Faltung mit einem kleinen Unterschied. Dieser Unterschied besteht darin, dass die Multiplikation bei der Kreuzkorrelation in derselben Richtung ausgeführt wird. Daher ist es nicht erforderlich, die Filtermatrix  $\mathbf{w}$  in jeder Dimension zu rotieren. Die Kreuzkorrelation ist wie folgt definiert:

$$\mathbf{y} = \mathbf{x} * \mathbf{w} \rightarrow y[i] = \sum_{k=-\infty}^{+\infty} x[i+k]w[k]$$

Die Regeln des Paddings sind bei der Kreuzkorrelation ebenfalls anwendbar.

## Auswirkung des Zero-Paddings bei der Faltung

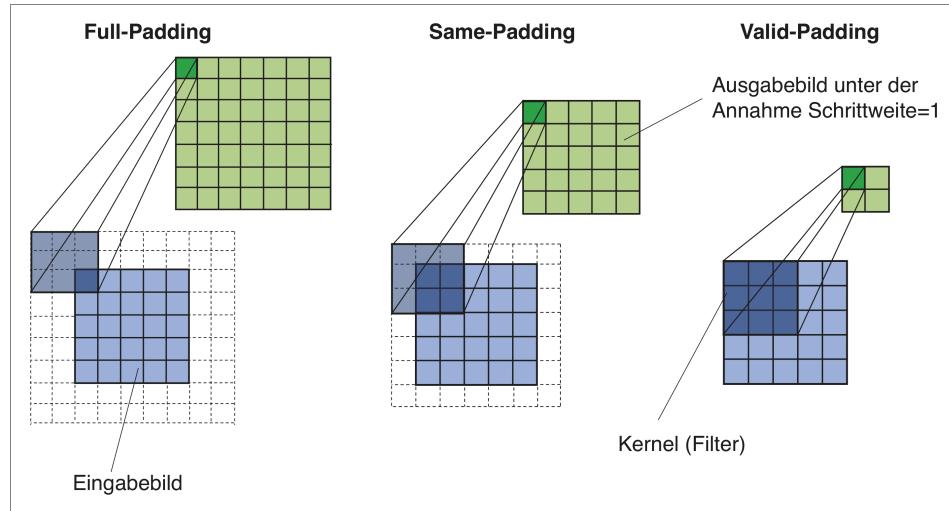
Bislang haben wir das Zero-Padding zur Berechnung von Ausgabevektoren endlicher Größe verwendet. Rein technisch betrachtet ist das Padding jedoch mit beliebigem  $p \geq 0$  anwendbar. Je nach gewähltem Wert von  $p$  werden Zellen am Rand möglicherweise anders verarbeitet als in der Mitte von  $\mathbf{x}$  befindliche Zellen.

Betrachten Sie ein Beispiel mit den Werten  $n = 5$  und  $m = 3$ . Wenn  $p = 0$  ist, wird  $\mathbf{x}[0]$  bei der Berechnung nur eines Ausgabeelements ( $\mathbf{y}[0]$ ) berücksichtigt,  $\mathbf{x}[1]$  hingegen geht zwei Mal in die Berechnung ein ( $\mathbf{y}[0]$  und  $\mathbf{y}[1]$ ). Sie sehen also, dass diese unterschiedliche Handhabung der Elemente von  $\mathbf{x}$  dem mittleren Element  $\mathbf{x}[2]$  größere Bedeutung beimisst, weil es in den meisten Berechnungen vor kommt. Durch die Wahl  $p = 2$  kann dieses Problem umgangen werden, denn in diesem Fall werden alle Elemente von  $\mathbf{x}$  bei der Berechnung von drei Elementen von  $\mathbf{y}$  berücksichtigt.

Die Größe der Ausgabe  $\mathbf{y}$  hängt außerdem von der verwendeten Padding-Strategie ab. In der Praxis sind drei verschiedene Modi gebräuchlich, nämlich Full-Padding, Same-Padding und Valid-Padding:

- Beim Full-Padding wird der Parameter  $p = m - l$  verwendet. Full-Padding erhöht die Anzahl der Dimensionen der Ausgabe, daher kommt es bei konvolutionalen neuronalen Netzen nur selten zum Einsatz.
- Same-Padding wird für gewöhnlich verwendet, wenn die Größe der Ausgabe mit der Größe des Eingabevektors  $\mathbf{x}$  übereinstimmen soll. Der Padding-Parameter  $p$  wird in diesem Fall entsprechend der Größe des Filters berechnet. Außerdem muss die Größe von Ein- und Ausgabe übereinstimmen.
- Valid-Padding schließlich beschreibt den Fall, dass bei der Berechnung der Faltung  $p = 0$  gesetzt wird (kein Padding).

Die folgende Abbildung veranschaulicht die drei verschiedenen Padding-Modi anhand einer einfachen  $5 \times 5$  Pixel großen Eingabe mit einer Kernelgröße von  $3 \times 3$  und der Schrittweite 1.



Bei konvolutionalen neuronalen Netzen ist Same-Padding am gebräuchlichsten. Gegenüber den anderen Padding-Modi hat es den Vorteil, Höhe und Breite des Eingabebilds oder des Eingabetensors zu erhalten, wodurch sich der Entwurf einer Netzarchitektur komfortabler gestaltet.

Valid-Padding hat im Vergleich mit Full- und Same-Padding den großen Nachteil, dass der Umfang der Tensoren bei neuronalen Netzen mit vielen Schichten beträchtlich zunimmt – und das kann für die Leistung des Netzes abträglich sein.

In der Praxis ist es empfehlenswert, die Größe bei den konvolutionalen Schichten beizubehalten und sie stattdessen in den Pooling-Schichten zu verringern. Beim Full-Padding nimmt die Größe der Ausgabe im Vergleich zur Eingabe zu. Full-Padding kommt für gewöhnlich bei Anwendungen zur Signalverarbeitung zum Einsatz, bei denen es wichtig ist, Randeffekte zu minimieren. Im Zusammenhang mit Deep Learning spielen derartige Effekte normalerweise keine Rolle, deshalb kommt Full-Padding nur selten vor.

## Größe der Faltungsausgabe ermitteln

Die Größe der Ausgabe einer Faltung wird durch die Anzahl der Verschiebungen des Filters  $w$  entlang des Eingabevektors festgelegt. Nehmen wir an, der Eingabevektor besitzt die Größe  $n$  und der Filter die Größe  $m$ . Die Größe der resultierenden Ausgabe von  $x * w$  mit Padding  $p$  und Schrittweite  $s$  ergibt sich so:

$$o = \left\lfloor \frac{n + 2p - m}{s} \right\rfloor + 1$$

Hier ist  $\lfloor \cdot \rfloor$  die Abrundungsfunktion (Gaußklammer oder floor-Funktion).

**Tipp**

Die Abrundungsfunktion (Gaußklammer) oder `floor`-Funktion liefert die größte ganze Zahl, die kleiner oder gleich der Eingabe ist, z.B.:

$$\text{floor}(1.77) = \lfloor 1.77 \rfloor = 1$$

Betrachten Sie die beiden folgenden Fälle:

- Berechnung der Ausgabegröße eines Eingabevektors der Größe 10 mit einer Kernelgröße von 5, Padding 2 und Schrittweite 1:

$$n=10, m=5, p=2, s=1 \rightarrow o = \left\lfloor \frac{10 + 2 \times 2 - 5}{1} \right\rfloor + 1 = 10$$

(In diesem Fall stimmt die Größe der Ausgabe mit der Größe der Eingabe überein, daher lässt sich folgern, dass es sich um Same-Padding handelt.)

- Wie ändert sich die Größe der Ausgabe bei gleichem Eingabevektor, aber einer Kernelgröße von 3 und der Schrittweite 2?

$$n=10, m=3, p=2, s=2 \rightarrow o = \left\lfloor \frac{10 + 2 \times 2 - 3}{2} \right\rfloor + 1 = 6$$

Wenn Sie mehr über die Größe der Ausgabe von Faltungen erfahren möchten, empfiehlt sich die Lektüre des Artikels *A guide to convolution arithmetic for deep learning* von Vincent Dumoulin und Francesco Visin (2016), der unter <https://arxiv.org/abs/1603.07285> kostenlos verfügbar ist.

Der nachstehende Code zeigt eine naive Implementierung der Berechnung einer eindimensionalen Faltung, und das Ergebnis wird mit dem der Funktion `numpy.convolve` verglichen:

```
>>> import numpy as np
>>> def conv1d(x, w, p=0, s=1):
...     w_rot = np.array(w[::-1])
...     x_padded = np.array(x)
...     if p > 0:
...         zero_pad = np.zeros(shape=p)
...         x_padded = np.concatenate([zero_pad,
...                                     x_padded,
...                                     zero_pad])
...     res = []
...     for i in range(0, int(len(x)/s), s):
...         res.append(np.sum(x_padded[i:i+w_rot.shape[0]] *
...                           w_rot))
```

```

...     return np.array(res)
>>> ## Testen:
>>> x = [1, 3, 2, 4, 5, 6, 1, 3]
>>> w = [1, 0, 3, 1, 2]
>>> print('Conv1d-Implementierung:', 
... conv1d(x, w, p=2, s=1))
Conv1d-Implementierung: [ 5. 14. 16. 26. 24. 34. 19. 22.]
>>> print('Ergebnis Numpy:', np.convolve(x, w, mode='same'))
Ergebnis Numpy: [ 5 14 16 26 24 34 19 22]

```

Wir haben bislang eine eindimensionale Faltung erörtert, um die grundlegenden Konzepte leichter verständlich zu machen. Im nächsten Abschnitt betrachten wir den zweidimensionalen Fall.

## Zweidimensionale diskrete Faltung

Die im letzten Abschnitt vorgestellten Konzepte lassen sich leicht auf zwei Dimensionen übertragen. Wenn wir es mit zweidimensionalen Eingaben zu tun haben (einer Matrix  $X_{n_1 \times n_2}$  und einer Filtermatrix  $W_{m_1 \times m_2}$ , mit  $m_1 \leq n_1$  und  $m_2 \leq n_2$ ), ist das Ergebnis der zweidimensionalen Faltung von  $\mathbf{X}$  und  $\mathbf{W}$  die Matrix  $\mathbf{Y} = \mathbf{X} * \mathbf{W}$ . Die Definition sieht folgendermaßen aus:

$$\mathbf{Y} = \mathbf{X} * \mathbf{W} \rightarrow Y[i, j] = \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} X[i - k_1, j - k_2] W[k_1, k_2]$$

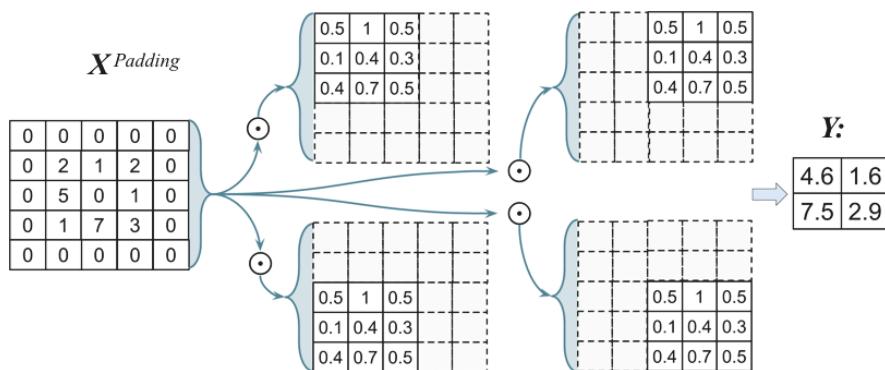
Wenn man eine der Dimensionen weglässt, ist dies genau dieselbe Formel, die wir zur Berechnung der eindimensionalen Faltung verwendet haben. Tatsächlich sind die bisher eingesetzten Verfahren wie Zero-Padding, Umkehren der Filtermatrix und die Schrittweite auch für zweidimensionale Faltungen anwendbar, sofern sie unabhängig voneinander auf beide Dimensionen erweitert werden. Das folgende Beispiel veranschaulicht die Berechnung einer zweidimensionalen Faltung einer Eingabematrix  $X_{3 \times 3}$  und einer Kernelmatrix  $W_{3 \times 3}$  mit Padding  $p = (1, 1)$  und Schrittweite  $s = (2, 2)$ . Durch das angegebene Padding werden alle vier Seiten der Matrix  $X_{5 \times 5}^{Padding}$  wie in der Abbildung mit Nullen aufgefüllt:

<b><math>X</math></b>	<b><math>W</math></b>							
0	0	0	0	0	*	0.5	0.7	0.4
0	2	1	2	0	*	0.3	0.4	0.1
0	5	0	1	0		0.5	1	0.5
0	1	7	3	0				
0	0	0	0	0				

Der dazugehörige umgekehrte Filter sieht folgendermaßen aus:

$$W^r = \begin{bmatrix} 0.5 & 1 & 0.5 \\ 0.1 & 0.4 & 0.3 \\ 0.4 & 0.7 & 0.5 \end{bmatrix}$$

Beachten Sie hier, dass die umgekehrte oder rotierte Matrix *nicht* der transponierten Matrix entspricht. Mit NumPy kann man den rotierten Filter als  $W_{\text{rot}}=W[::-1, ::-1]$  schreiben. Anschließend verschieben wir den rotierten Filter in der mit Nullen aufgefüllten Matrix  $X^{Padding}$  wie ein Fenster und berechnen die Summe der elementweisen Produkte. In der folgenden Abbildung ist das durch den Operator  $\odot$  gekennzeichnet.



Das Ergebnis ist eine  $2 \times 2$ -Matrix  $\mathbf{Y}$ .

Nun implementieren wir eine zweidimensionale Faltung entsprechend des beschriebenen naiven Algorithmus. Das Paket `scipy.signal` bietet die Möglichkeit, eine zweidimensionale Faltung mit der Funktion `scipy.signal.convolve2d` zu berechnen:

```
>>> import numpy as np
>>> import scipy.signal
>>> def conv2d(X, W, p=(0, 0), s=(1, 1)):
...     W_rot = np.array(W)[::-1, ::-1]
...     X_orig = np.array(X)
...     n1 = X_orig.shape[0] + 2*p[0]
...     n2 = X_orig.shape[1] + 2*p[1]
...     X_padded = np.zeros(shape=(n1, n2))
...     X_padded[p[0]:p[0]+X_orig.shape[0],
...               p[1]:p[1]+X_orig.shape[1]] = X_orig
...
...     res = []
...     for i in range(0, int((X_padded.shape[0] - \

```

```

...
        W_rot.shape[0])/s[0])+1, s[0]):
...
    res.append([])
...
    for j in range(0, int((X_padded.shape[1] - \
...
        W_rot.shape[1])/s[1])+1, s[1]):
...
        X_sub = X_padded[i:i+W_rot.shape[0],
...
            j:j+W_rot.shape[1]]
...
        res[-1].append(np.sum(X_sub * W_rot))
...
    return(np.array(res))

>>> X = [[1, 3, 2, 4], [5, 6, 1, 3], [1, 2, 0, 2], [3, 4, 3, 2]]
>>> W = [[1, 0, 3], [1, 2, 1], [0, 1, 1]]
>>> print('Conv2d-Implementierung:\n',
...       conv2d(X, W, p=(1, 1), s=(1, 1)))
Conv2d-Implementierung:
[[ 11.  25.  32.  13.]
 [ 19.  25.  24.  13.]
 [ 13.  28.  25.  17.]
 [ 11.  17.  14.   9.]]
>>> print('Ergebnis SciPy:\n',
...       scipy.signal.convolve2d(X, W, mode='same'))
Ergebnis SciPy:
[[11 25 32 13]
 [19 25 24 13]
 [13 28 25 17]
 [11 17 14  9]]

```

## Hinweis

Wir haben hier eine naive Implementierung der Berechnung einer zweidimensionalen Faltung vorgestellt, um die Konzepte zu verdeutlichen. Diese Implementierung ist hinsichtlich des Speicherbedarfs und der Komplexität der Berechnungen sehr ineffizient und sollte daher in der Praxis nicht verwendet werden.

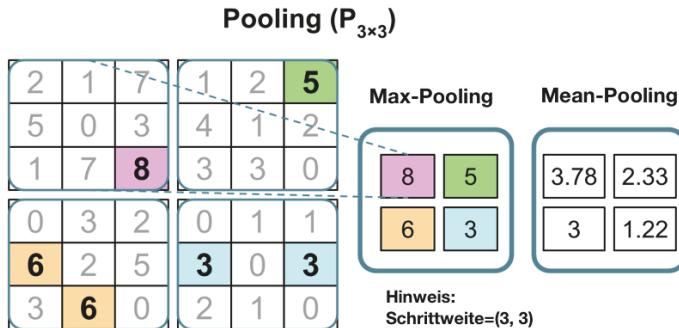
In den letzten Jahren sind sehr viel effizientere Algorithmen entwickelt worden, die zur Berechnung von Faltungen Fourier-Transformationen verwenden. An dieser Stelle muss außerdem erwähnt werden, dass die Größe des Faltungskernels im Kontext neuronaler Netze für gewöhnlich sehr viel kleiner ist als die des Eingabebilds. Moderne CNNs verwenden beispielsweise Kernelgrößen wie  $1 \times 1$ ,  $3 \times 3$  oder  $5 \times 5$ , für die es Algorithmen gibt, die Faltungsoperationen sehr viel effizienter ausführen können, wie etwa der Winograd-Algorithmus (minimale Filter). Solche Algorithmen gehen über den Rahmen des Buches hinaus, aber wenn Sie mehr darüber erfahren möchten, sollten Sie den Artikel *Fast Algorithms for Convolutional Neural Networks* von Andrew Lavin und Scott Gray (2015) lesen, der unter <https://arxiv.org/abs/1509.09308> kostenlos verfügbar ist.

Im nächsten Abschnitt erörtern wie das Subsampling, eine weitere Operation, die in CNNs häufig eingesetzt wird.

### 15.1.3 Subsampling

Typischerweise finden in konvolutionalen neuronalen Netzen zwei Arten von Pooling-Operationen Anwendung: *Max-Pooling* und *Mean-Pooling* (das mitunter auch als *Average-Pooling* bezeichnet wird). Die Pooling-Schicht wird für gewöhnlich mit  $P_{n_1 \times n_2}$  bezeichnet. Der tiefgestellte Index gibt hier die Größe der Nachbarschaft an (die Anzahl der benachbarten Pixel in jeder Dimension), mit der die Max- oder Mean-Operation ausgeführt wird. Wir bezeichnen diese Nachbarschaft als *Pooling-Größe*.

Die folgende Abbildung beschreibt diese Operationen. Beim Max-Pooling wird der maximale Wert der benachbarten Pixel entnommen und beim Mean-Pooling wird der Mittelwert gebildet.



Das Pooling hat zwei Vorteile:

- Durch das Pooling (Max-Pooling) wird eine gewisse lokale Invarianz eingebracht. Damit ist gemeint, dass kleine Veränderungen in einer lokalen Nachbarschaft nicht zu einem anderen Ergebnis des Max-Poolings führen. Deshalb ist es sinnvoll, Merkmale zu generieren, die weniger anfällig für das Rauschen in den Eingabedaten sind. Das folgende Beispiel zeigt das Max-Pooling zweier verschiedener Matrizen  $X_1$  und  $X_2$ , das jedoch zum selben Ergebnis führt:

$$X_1 = \begin{bmatrix} 10 & 255 & 125 & 0 & 170 & 100 \\ 70 & 255 & 105 & 25 & 25 & 70 \\ 255 & 0 & 150 & 0 & 10 & 10 \\ 0 & 255 & 10 & 10 & 150 & 20 \\ 70 & 15 & 200 & 100 & 95 & 0 \\ 35 & 25 & 100 & 20 & 0 & 60 \end{bmatrix}$$

$$X_2 = \begin{bmatrix} 100 & 100 & 100 & 50 & 100 & 50 \\ 95 & 255 & 100 & 125 & 125 & 170 \\ 80 & 40 & 10 & 10 & 125 & 150 \\ 255 & 30 & 150 & 20 & 120 & 125 \\ 30 & 30 & 150 & 100 & 70 & 70 \\ 70 & 30 & 100 & 200 & 70 & 95 \end{bmatrix}$$

Max-Pooling  $P_{2 \times 2} \rightarrow \begin{bmatrix} 255 & 125 & 170 \\ 255 & 150 & 150 \\ 70 & 200 & 95 \end{bmatrix}$

- Das Pooling reduziert die Anzahl der Merkmale, und das ermöglicht effizientere Berechnungen. Darüber hinaus kann die verringerte Anzahl von Merkmalen auch das Ausmaß einer Überanpassung reduzieren.

### Hinweis

Normalerweise geht man davon aus, dass es beim Pooling keine Überschneidungen gibt. Das Pooling wird typischerweise mit sich nicht überlappenden Nachbarschaften durchgeführt. Dazu setzt man den Parameter für die Schrittweite gleich der Pooling-Größe. Für eine Pooling-Schicht ohne Überschneidungen  $P_{n_1 \times n_2}$  ist beispielsweise eine Schrittweite  $s=(n_1, n_2)$  erforderlich.

Sollte die Schrittweite hingegen kleiner als die Pooling-Größe sein, tritt beim Pooling eine Überschneidung auf. Ein Beispiel für ein Pooling mit Überschneidungen in einem konvolutionalen neuronalen Netz ist in dem Artikel *ImageNet Classification with Deep Convolutional Neural Networks* von A. Krizhevsky, I. Sutskever und G. Hinton (2012) beschrieben. Er ist unter <https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks> kostenlos online verfügbar.

## 15.2 Implementierung eines CNNs

Sie haben nun die elementaren Bausteine eines konvolutionalen neuronalen Netzes kennengelernt. Die in diesem Kapitel vorgestellten Konzepte sind eigentlich nicht schwieriger zu verstehen als herkömmliche mehrschichtige neuronale Netze. Oder anschaulich ausgedrückt: Bei einem herkömmlichen neuronalen Netz ist die Matrix-Vektor-Multiplikation die wichtigste Operation.

Wir verwenden Matrix-Vektor-Multiplikationen wie  $\mathbf{a} = \mathbf{Wx} + \mathbf{b}$  beispielsweise bei der Berechnung von Voraktivierungen (oder der Nettoeingabe). Hier ist  $\mathbf{x}$  ein Spaltenvektor zur Repräsentierung von Pixeln und  $\mathbf{W}$  ist die Gewichtungsmatrix, die Pixeleingaben mit den verdeckten Einheiten verknüpft. Bei einem konvolutionalen neuronalen Netz wird diese Operation durch eine Faltung wie  $\mathbf{A} = \mathbf{W} * \mathbf{X} + \mathbf{b}$  ersetzt.  $\mathbf{X}$  ist hier eine Matrix, in der die Pixel in Form von Höhe und Breite repräsentiert werden. In beiden Fällen werden die Voraktivierungen an eine Aktivierungsfunktion übergeben, um die Aktivierung einer verdeckten Einheit  $\mathbf{H} = \phi(\mathbf{A})$  zu erhalten, wobei  $\phi$  die Aktivierungsfunktion ist. Wie Sie wissen, ist Subsampling ein weiterer Baustein eines konvolutionalen neuronalen Netzes und kann, wie im letzten Abschnitt beschrieben, in Form von Pooling auftreten.

### 15.2.1 Verwendung mehrerer Eingabe- oder Farbkanäle

Die Eingabe für eine Konvolutionsschicht kann mehrere zweidimensionale Arrays oder Matrizen der Dimension  $N_1 \times N_2$  (beispielsweise Höhe und Breite des Bildes

in Pixeln) enthalten. Diese  $N_1 \times N_2$ -Matrizen werden als Kanäle bezeichnet. Die Nutzung mehrerer Kanäle als Eingabe für eine Konvolutionsschicht macht es somit erforderlich, einen Tensor des Rangs 3 oder ein dreidimensionales Array zu verwenden:  $X_{N_1 \times N_2 \times C_{in}}$ , wobei  $C_{in}$  die Anzahl der Eingabekanäle angibt.

Betrachten wir den Fall, dass Bilder als Eingabe für die erste Schicht eines CNNs verwendet werden. Wenn das Bild farbig ist und den RGB-Farbraum verwendet, dann gilt  $C_{in} = 3$  (für die RGB-Farbkanäle Rot, Grün und Blau). Liegt das Bild hingegen in Graustufen vor, gilt  $C_{in} = 1$ , denn dann gibt es nur einen Graustufenkanal, der die Intensitätswerte der Pixel enthält.

### Tipp

Bilder können mithilfe des Datentyps '`uint8`' (vorzeichenlose 8-Bit-Ganzzahl) in NumPy-Arrays eingelesen werden, um gegenüber der Verwendung von 16-, 32- oder 64-Bit-Datentypen Speicherplatz einzusparen. Vorzeichenlose 8-Bit-Ganzzahlen können Werte zwischen 0 und 255 annehmen, was ausreicht, um die Pixelinformationen von RGB-Bildern zu speichern, die Werte aus demselben Bereich besitzen. Sehen Sie sich ein Beispiel an, wie Bilder mit SciPy in eine Python-Sitzung eingelesen werden. Dazu muss das Paket PIL (Python Image Library) installiert sein. Sie können Pillow (<https://python-pillow.org>), eine benutzerfreundlichere Variante von PIL, wie folgt installieren:

```
pip install pillow
```

Nach der Installation kann die `imread`-Funktion des `scipy.misc`-Moduls verwendet werden, um RGB-Bilder zu lesen (das Beispielbild befindet sich im Codeordner zu diesem Kapitel unter <https://github.com/rasbt/python-machine-learning-book-2nd-edition/tree/master/code/ch15>):

```
>>> import scipy.misc
>>> img = scipy.misc.imread(
...             './example-image.png',
...             mode='RGB')
>>> print('Form des Bilds:', img.shape)
Form des Bilds: (252, 221, 3)
>>> print('Anzahl der Kanäle:', img.shape[2])
Anzahl der Kanäle: 3
>>> print('Datentyp des Bilds:', img.dtype)
Datentyp des Bilds: uint8
>>> print(img[100:102, 100:102, :])
[[[179 134 110]
 [182 136 112]]
 [[180 135 111]
 [182 137 113]]]
```

Nun haben wir uns mit der Struktur der Eingabedaten vertraut gemacht und es stellt sich die Frage, wie man die in den vorangegangenen Abschnitten erörterte Faltungsoperation mit mehreren Eingabekanälen verwendet.

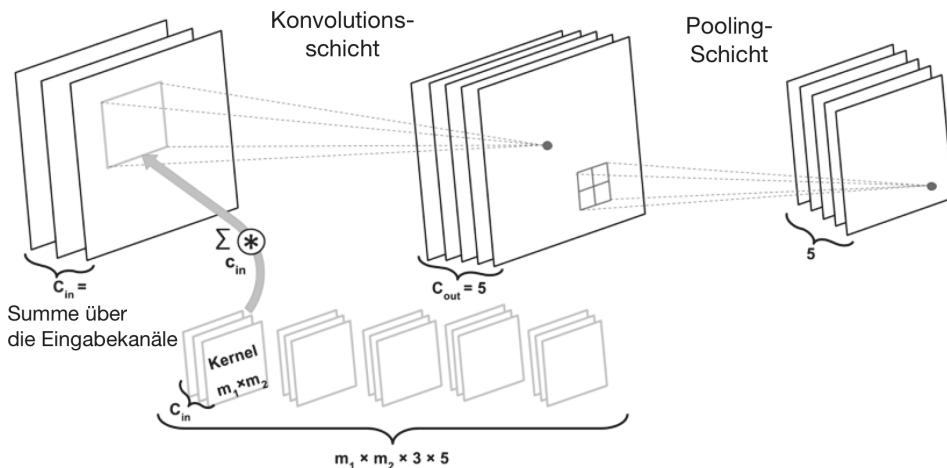
Die Antwort ist ganz einfach: Wir führen die Faltungsoperation für jeden Kanal einzeln durch und addieren die Ergebnisse durch Matrixsummierung. Jede zu einem Kanal ( $c$ ) zugehörige Faltung besitzt eine eigene Kernelmatrix  $\mathbf{W}[:, :, c]$ . Die gesamte Voraktivierung wird mit folgender Gleichung berechnet:

$$\begin{array}{ll} \text{Gegeben: } & \mathbf{X}_{n_1 \times n_2 \times c_{in}} \\ \text{Kernelmatrix } & \mathbf{W}_{m_1 \times m_2 \times c_{in}} \Rightarrow \left\{ \begin{array}{l} \mathbf{Y}^{Faltung} = \sum_{c=1}^{C_{in}} \mathbf{W}[:, :, c] * \mathbf{X}[:, :, c] \\ \text{Voraktivierung: } \mathbf{A} = \mathbf{Y}^{Faltung} + \mathbf{b} \\ \text{Merkmalskarte: } \mathbf{H} = \phi(\mathbf{A}) \end{array} \right. \\ \text{und das Bias } & b \end{array}$$

Das Endergebnis  $\mathbf{H}$  wird als Merkmalskarte bezeichnet. Für gewöhnlich besitzt die Konvolutionsschicht eines CNNs mehr als eine Merkmalskarte. Wenn wir mehr als eine Merkmalskarte verwenden, wird der Kernaltensor vierdimensional:  $Breite \times Höhe \times C_{in} \times C_{out}$ .  $Breite \times Höhe$  ist hier die Kernelgröße,  $C_{in}$  die Anzahl der Eingabekanäle und  $C_{out}$  die Anzahl der ausgegebenen Merkmalskarten. Nun übernehmen wir in der letzten Gleichung die ausgegebenen Merkmalskarten und erhalten:

$$\begin{array}{ll} \text{Gegeben: } & \mathbf{X}_{n_1 \times n_2 \times C_{in}} \\ \text{Kernelmatrix } & \mathbf{W}_{m_1 \times m_2 \times C_{in} \times C_{out}} \Rightarrow \left\{ \begin{array}{l} \mathbf{Y}^{Faltung}[:, :, :, k] = \sum_{c=1}^{C_{in}} \mathbf{W}[:, :, c, k] * \mathbf{X}[:, :, c] \\ \mathbf{A}[:, :, :, k] = \mathbf{Y}^{Faltung}[:, :, :, k] + \mathbf{b}[k] \\ \mathbf{H}[:, :, :, k] = \phi(\mathbf{A}[:, :, :, k]) \end{array} \right. \\ \text{und Biasvektor } & \mathbf{b}_{C_{out}} \end{array}$$

Wir fahren mit der Erörterung der Berechnung von Faltungen im Kontext neuronaler Netze fort und betrachten das Beispiel in der folgenden Abbildung, das eine Konvolutionsschicht zeigt, der eine Pooling-Schicht folgt.



In diesem Beispiel gibt es drei Eingabekanäle. Der Kerneltensor ist vierdimensional. Die Kernelmatrizen sind mit  $m_1 \times m_2$  bezeichnet. Es gibt drei Stück – eine für jeden Eingabekanal. Außerdem gibt es fünf solche Kernel, die für fünf ausgegebene Merkmalskarten verantwortlich sind. Und schließlich ist da noch die Pooling-Schicht für das Subsampling der Merkmalskarten (siehe Abbildung).

### Tipp

Um die Vorteile von Faltung, gemeinsamen Parametern und gezielten Verknüpfungen zu veranschaulichen, betrachten wir ein Beispiel. Die Konvolutionsschicht des Netzes in der letzten Abbildung ist ein vierdimensionaler Tensor. Dem Kernel sind also  $m_1 \times m_2 \times 3 \times 5 + 5$  Parameter zugeordnet. Zudem gibt es für jede Merkmalskarte der Konvolutionsschicht einen Biasvektor. Die Größe des Biasvektors ist also 5. Pooling-Schichten besitzen keine (trainierbaren) Parameter, also können wir schreiben:

$$m_1 \times m_2 \times 3 \times 5 + 5.$$

Wenn wir annehmen, dass die Faltung mit Same-Padding ausgeführt wird und der Eingabetensor die Größe  $n_1 \times n_2 \times 3$  besitzt, hätten die ausgegebenen Merkmalskarten die Größe  $n_1 \times n_2 \times 5$ .

Beachten Sie, dass diese Zahl erheblich kleiner ist als für den Fall, dass wir statt der Konvolutionsschicht eine vollständig verknüpfte Schicht verwenden. Dann ergibt sich nämlich für die Anzahl der Parameter für die Gewichtungsmatrix, die genauso viele Ausgabeeinheiten verknüpft, folgender Wert:

$$(n_1 \times n_2 \times 3) \times (n_1 \times n_2 \times 5) = (n_1 \times n_2)^2 \times 3 \times 5.$$

In Anbetracht der Tatsache, dass  $m_1 < n_1$  und  $m_2 < n_2$  ist, wird klar, dass sich die Anzahlen der trainierbarer Parameter enorm unterscheiden.

Der nächste Abschnitt hat zum Thema, wie man ein neuronales Netz regularisieren kann.

### 15.2.2 Regularisierung eines neuronalen Netzes mit Dropout

Ob man es mit herkömmlichen (vollständig verknüpften) oder neuronalen Netzen zu tun hat, spielt keine Rolle: Es war schon immer eine Herausforderung, die Größe eines Netzes zu wählen. So müssen beispielsweise die Größe der Gewichtungsmatrix und die Anzahl der Schichten abgestimmt werden, um eine einigermaßen vernünftige Leistung zu erzielen.

Die Aufnahmefähigkeit eines Netzes bezieht sich auf den Grad der Komplexität der Funktion, die es erlernen kann. Kleine Netze mit relativ wenigen Parametern

weisen ein geringes Aufnahmevermögen auf, sodass es eher zu einer Unteranpassung kommen kann, was wiederum zu mangelhafter Leistung führt, weil die komplexen Datenmengen zugrunde liegende Struktur nicht erlernt werden kann.

Sehr große Netze neigen hingegen zu einer Überanpassung, wenn das Netz die Trainingsdaten gespeichert hat und mit den Trainingsdaten außerordentlich gute Leistungen erzielt, aber bei der zurückgehaltenen Testdatenmenge nur schlecht funktioniert. In der Praxis steht *a priori* nicht fest, wie groß ein Netz sein sollte, um eine bestimmte Machine-Learning-Aufgabe zu lösen.

Eine Möglichkeit, dieses Problem in den Griff zu bekommen, ist, ein Netz mit relativ großer Aufnahmefähigkeit zu verwenden (in der Praxis sollte die Aufnahmefähigkeit etwas größer sein als nötig), das gut mit den Trainingsdaten zurechtkommt. Dann verwenden wir ein Regularisierungsverfahren (oder mehrere), um eine Überanpassung zu verhindern und eine gute Verallgemeinerungsfähigkeit bei neuen Daten zu erzielen, etwa eine zurückgehaltene Testdatenmenge. Die L2-Regularisierung, die in diesem Buch bereits erörtert wurde, ist ein gebräuchliches Verfahren.

In den letzten Jahren hat sich ein weiteres Regularisierungsverfahren namens *Dropout* etabliert, das bemerkenswert gut zur Regularisierung (tiefer) neuronaler Netze geeignet ist (Nitish Srivastava et al., *Dropout: a simple way to prevent neural networks from overfitting*, Journal of Machine Learning Research 15.1, Seiten 1929–1958, 2014, <http://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf>).

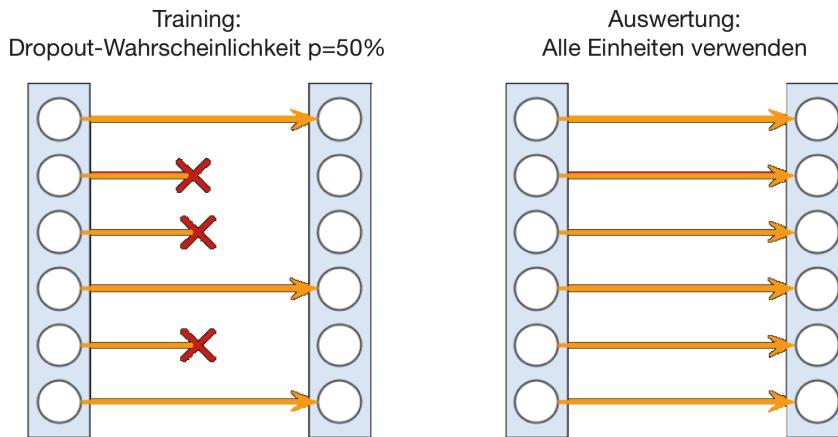
Anschaulich kann man Dropout als den Konsens (durch Durchschnittsbildung) eines Ensembles von Modellen beschreiben. Beim Ensemble Learning werden mehrere Modelle voneinander unabhängig trainiert. Zur Vorhersage dient der Konsens aller trainierten Modelle. Allerdings ist sowohl das Trainieren mehrerer Modelle als auch die Durchschnittsbildung der Ausgaben mehrerer Modelle rechenaufwendig. Hier bietet Dropout eine Lösung in Form eines effizienten Verfahrens zum gleichzeitigen Trainieren vieler Modelle und der Durchschnittsbildung ihrer Ergebnisse zum Zeitpunkt des Tests oder der Vorhersage.

Das Dropout-Verfahren wird für gewöhnlich auf die verdeckten Einheiten höherer Schichten angewendet. Während der Trainingsphase des neuronalen Netzes wird bei jeder Iteration ein bestimmter Anteil der verdeckten Einheiten mit der Wahrscheinlichkeit  $p_{drop}$  zufällig entfernt (oder mit der Wahrscheinlichkeit  $p_{keep} = 1 - p_{drop}$  nicht entfernt).

Die Dropout-Wahrscheinlichkeit wird vom Benutzer festgelegt. Ein gebräuchlicher Wert ist  $p = 0.5$ , wie in dem eben erwähnten Artikel von Nitish Srivastava et al. (2014) erläutert wird. Nach dem Entfernen dieses Teils der Eingabeneuronen werden die Gewichtungen der verbleibenden Neuronen neu berechnet, um den fehlenden (entfernten) Neuronen Rechnung zu tragen.

Dieses zufällige Entfernen bewirkt, dass das Netz gezwungen ist, eine redundante Repräsentierung der Daten zu erlernen. Das Netz kann nicht auf die Aktivierung durch irgendeine Menge verdeckter Einheiten zurückgreifen, denn diese könnten während des Trainings zu jedem beliebigen Zeitpunkt abgeschaltet werden, und ist somit gezwungen, aus allgemeineren und robusteren Mustern in den Daten zu lernen.

Das zufällige Entfernen kann eine Überanpassung sehr effektiv verhindern. Die folgende Abbildung zeigt ein Beispiel eines Dropouts mit der Wahrscheinlichkeit  $p = 0.5$ , in dem während der Trainingsphase jeweils die Hälfte der Neuronen zufällig deaktiviert wurde. Bei der Vorhersage tragen allerdings alle Neuronen zur Voraktivierung der nächsten Schicht bei.



Hier ist es wichtig, daran zu denken, dass die Einheiten nur während des Trainings zufällig deaktiviert werden, denn in der Auswertungsphase müssen alle verdeckten Einheiten aktiv sein (also  $p_{drop} = 0$  oder  $p_{keep} = 1$ ). Um sicherzustellen, dass die Aktivierungen insgesamt während des Trainings und bei der Vorhersage von gleicher Größenordnung sind, müssen die Aktivierungen der aktiven Neuronen entsprechend skaliert werden (z.B. durch Halbierung der Aktivierungen, wenn die Dropout-Wahrscheinlichkeit auf  $p = 0.5$  gesetzt war).

Da es in der Praxis allerdings lästig ist, beim Treffen von Vorhersagen immer die Aktivierungen skalieren zu müssen, erledigen TensorFlow und andere Tools diese Aufgabe während des Trainings (z.B. durch Verdopplung der Aktivierungen, wenn die Dropout-Wahrscheinlichkeit auf  $p = 0.5$  gesetzt war).

Und in welchem Verhältnis stehen Dropout und Ensemble Learning? Da bei jeder Iteration andere verdeckte Neuronen deaktiviert werden, trainieren wir de facto verschiedene Modelle. Wenn all diese Modelle endgültig trainiert werden, setzen wir  $p_{keep} = 1$  und verwenden alle verdeckten Einheiten. Wir verwenden also die durchschnittliche Aktivierung aller verdeckten Einheiten.

## 15.3 Implementierung eines tiefen konvolutionalen neuronalen Netzes mit TensorFlow

In Kapitel 13 haben wir ein mehrschichtiges neuronales Netz zur Erkennung handgeschriebener Ziffern implementiert und dafür Tensorflows High- und Low-level-API verwendet. Damit konnten wir eine Korrektklassifizierungsrate von 97 Prozent erzielen.

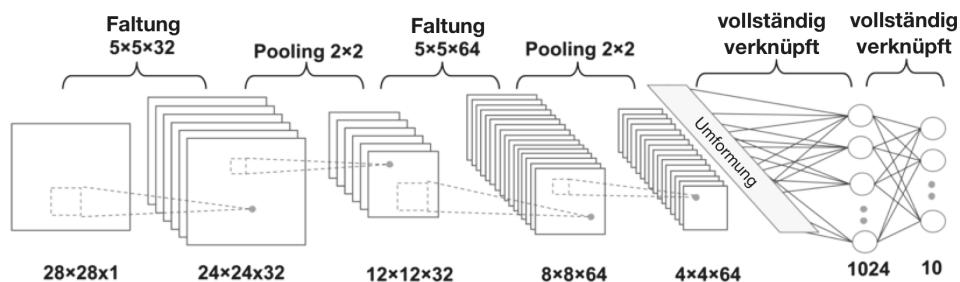
Nun wollen wir ein CNN zur Lösung derselben Aufgabe implementieren und prüfen, wie gut es die handgeschriebenen Ziffern klassifizieren kann. Die in Kapitel 13 verwendeten vollständig verknüpften Schichten konnten diese Aufgabe gut lösen. Bei manchen Anwendungsfällen, wie z.B. dem Einlesen handgeschriebener Nummern von Bankkonten, können schon kleinste Fehler schwerwiegende Konsequenzen haben, daher ist es besonders wichtig, die Fehlerquote so weit wie möglich zu minimieren.

### 15.3.1 Die mehrschichtige CNN-Architektur

In der folgenden Abbildung ist die Netzarchitektur dargestellt, die wir implementieren werden. Als Eingabe dienen  $28 \times 28$  Pixel große Graustufenbilder. Bei Graustufenbildern gibt es nur einen Eingabekanal, also ist die Größe des Eingabetensors bei einem Stapel von Bildern  $\text{Stapelgröße} \times 28 \times 28 \times 1$ .

Die Eingabedaten durchlaufen zwei Konvolutionsschichten mit einer Kernelgröße von  $5 \times 5$ . Die erste Faltung ergibt 32 Merkmalskarten und die zweite 64. Diesen beiden Schichten folgt jeweils eine Subsampling-Schicht, in der eine Max-Pooling-Operation stattfindet.

Anschließend übergibt eine vollständig verknüpfte Schicht die Ausgabe an eine zweite vollständig verknüpfte Schicht, die als endgültige softmax-Ausgabeschicht fungiert (siehe Abbildung).



Die Tensoren haben in den verschiedenen Schichten folgende Dimensionen:

- Eingabe: [Stapelgröße $\times 28 \times 28 \times 1$ ]
- Faltung\_1: [Stapelgröße $\times 24 \times 24 \times 32$ ]

- Pooling\_1: [Stapelgröße x 12 x 12 x 32]
- Faltung\_2: [Stapelgröße x 8 x 8 x 64]
- Pooling\_2: [Stapelgröße x 4 x 4 x 64]
- Erste vollständig verknüpfte Schicht: [Stapelgröße x 1024]
- Zweite vollständig verknüpfte softmax-Schicht: [Stapelgröße x 10]

Wir implementieren das Netz mit zwei APIs, nämlich mit Tensorflows Low-level- und High-level-API. Im nächsten Abschnitt müssen jedoch zunächst einige Hilfsfunktionen definiert werden.

### 15.3.2 Einlesen und Vorverarbeiten der Daten

In Kapitel 13 haben wir zum Einlesen der handgeschriebenen MNIST-Datensammlung eine Funktion namens `load_mnist` verwendet. Hier benötigen wir diese Funktionalität ebenfalls:

```
>>> ##### Daten einlesen
>>> X_data, y_data = load_mnist('./mnist/', kind='train')
>>> print('Zeilen: {}, Spalten: {}'.format(
...         X_data.shape[0], X_data.shape[1]))
>>> X_test, y_test = load_mnist('./mnist/', kind='t10k')
>>> print('Zeilen: {}, Spalten: {}'.format(
...         X_test.shape[0], X_test.shape[1]))
>>> X_train, y_train = X_data[:50000,:], y_data[:50000]
>>> X_valid, y_valid = X_data[50000:,:], y_data[50000:]
>>> print('Training:    ', X_train.shape, y_train.shape)
>>> print('Validierung: ', X_valid.shape, y_valid.shape)
>>> print('Testdaten:   ', X_test.shape, y_test.shape)
```

Wir teilen die Daten in eine Trainings-, eine Validierungs- und eine Testdatenmenge auf. Die Ausgabe zeigt die Form dieser Datenmengen:

```
Zeilen: 60000, Spalten: 784
Zeilen: 10000, Spalten: 784
Training:    (50000, 784) (50000,)
Validierung: (10000, 784) (10000,)
Testdaten:   (10000, 784) (10000,)
```

Nach dem Einlesen der Daten benötigen wir eine Funktion, um die Mini-Batches zu durchlaufen:

```
>>> def batch_generator(X, y, batch_size=64,
...                      shuffle=False, random_seed=None):
...     ...
```

```

...     idx = np.arange(y.shape[0])
...
...     if shuffle:
...         rng = np.random.RandomState(random_seed)
...         rng.shuffle(idx)
...         X = X[idx]
...         y = y[idx]
...     for i in range(0, X.shape[0], batch_size):
...         yield (X[i:i+batch_size, :], y[i:i+batch_size])

```

Die Funktion gibt einen Generator für Tupel von Objekten zurück, z.B. die Daten  $X$  und die Klassenbezeichnung  $y$ . Anschließend müssen die Daten normiert werden (Zentrierung um den Mittelwert und Division durch die Standardabweichung), um eine bessere Trainingsleistung und Konvergenz zu erzielen.

Wir berechnen den Mittelwert der Merkmale anhand der Trainingsdaten ( $X_{train}$ ) und berechnen die Standardabweichung aller Merkmale. Die Standardabweichungen der Merkmale werden nicht einzeln berechnet, weil es in Bilddaten wie der MNIST-Datenmenge Merkmale (Pixelpositionen) gibt, die in allen Bildern einen konstanten Wert von 255 enthalten (weiße Pixel in einem Graustufenbild).

Ein konstanter Wert in allen Bildern bedeutet keine Varianz, daher wäre die Standardabweichung dieser Merkmale null und es käme zu einer Fehlermeldung (Division durch null). Deshalb berechnen wir die Standardabweichung des  $X_{train}$ -Arrays mit `np.std` ohne Angabe eines `axis`-Arguments:

```

>>> mean_vals = np.mean(X_train, axis=0)
>>> std_val = np.std(X_train)
>>> X_train_centered = (X_train - mean_vals)/std_val
>>> X_valid_centered = (X_valid - mean_vals)/std_val
>>> X_test_centered = (X_test - mean_vals)/std_val

```

Nun ist alles vorbereitet, um das eben beschriebene CNN mit TensorFlow zu implementieren.

### 15.3.3 Implementierung eines CNNs mit Tensorflows Low-level-API

Zwecks Implementierung eines CNNs mit TensorFlow definieren wir zunächst zwei Wrapper-Funktionen, die das Erstellen des Netzes vereinfachen: eine Funktion zum Erstellen einer Konvolutionsschicht sowie eine Funktion zum Erzeugen einer vollständig verknüpften Schicht.

Hier ist der Code für das Erstellen einer Konvolutionsschicht:

```

import tensorflow as tf
import numpy as np

```

```

def conv_layer(input_tensor, name,
               kernel_size, n_output_channels,
               padding_mode='SAME', strides=(1, 1, 1, 1)):
    with tf.variable_scope(name):
        ## get n_input_channels:
        ##   input tensor shape:
        ##   [batch x width x height x channels_in]
        input_shape = input_tensor.get_shape().as_list()
        n_input_channels = input_shape[-1]
        weights_shape = list(kernel_size) + \
            [n_input_channels, n_output_channels]
        weights = tf.get_variable(name='_weights',
                                  shape=weights_shape)
        print(weights)
        biases = tf.get_variable(name='_biases',
                                 initializer=tf.zeros(
                                     shape=[n_output_channels]))
        print(biases)
        conv = tf.nn.conv2d(input=input_tensor,
                            filter=weights,
                            strides=strides,
                            padding=padding_mode)
        print(conv)
        conv = tf.nn.bias_add(conv, biases,
                              name='net_pre-activation')
        print(conv)
        conv = tf.nn.relu(conv, name='activation')
        print(conv)

    return conv

```

Diese Wrapper-Funktion erledigt alle für das Erstellen einer Konvolutionsschicht erforderlichen Aufgaben, inklusive der Definition von Gewichtungen und Bias, deren Initialisierung und der Faltungsoperation mit der `tf.nn.conv2d`-Funktion. Dafür sind vier Argumente notwendig:

- `input_tensor`: Der an die Konvolutionsschicht als Eingabe übergebene Tensor
- `name`: Die Bezeichnung der Schicht, die als Geltungsbereich verwendet wird
- `kernel_size`: Die Dimensionen des als Tupel oder Liste übergebenen Kernel-tensors
- `n_output_channels`: Die Anzahl der auszugebenden Merkmalskarten

Bei Verwendung von `tf.get_variable` kommt für die Gewichtungen standardmäßig die Xavier-(oder Gorot-)Initialisierung zum Einsatz (die Xavier/Gorot-Initi-

alisierung wurde in Kapitel 14 erörtert). Die Bias werden hingegen mit der `tf.zeros`-Funktion initialisiert. Die Nettovoraktivierungen werden an die ReLU-Aktivierungsfunktion übergeben. Um Formen und Typen der Tensoren anzusehen, können wir die Knoten und Operationen des TensorFlow-Graphen ausgeben. Nun testen wir die Funktion mit einer einfachen Eingabe durch die Definition eines Platzhalters:

```
>>> g = tf.Graph()
>>> with g.as_default():
...     x = tf.placeholder(tf.float32,
...                        shape=[None, 28, 28, 1])
...     conv_layer(x, name='convtest',
...                kernel_size=(3, 3),
...                n_output_channels=32)
>>>
>>> del g, x

<tf.Variable 'convtest/_weights:0' shape=(3, 3, 1, 32)
dtype=float32_ref>
<tf.Variable 'convtest/_biases:0' shape=(32,)
dtype=float32_ref>
Tensor("convtest/Conv2D:0", shape=(?, 28, 28, 32),
dtype=float32)
Tensor("convtest/net_pre-activaiton:0",
shape=(?, 28, 28, 32), dtype=float32)
Tensor("convtest/activation:0", shape=(?, 28, 28, 32),
dtype=float32)
```

Die nächste Wrapper-Funktion dient zur Definition der vollständig verknüpften Schicht:

```
def fc_layer(input_tensor, name,
             n_output_units, activation_fn=None):
    with tf.variable_scope(name):
        input_shape = input_tensor.get_shape().as_list()[1:]
        n_input_units = np.prod(input_shape)
        if len(input_shape) > 1:
            input_tensor = tf.reshape(input_tensor,
                                     shape=(-1, n_input_units))
        weights_shape = [n_input_units, n_output_units]
        weights = tf.get_variable(name='_weights',
                                 shape=weights_shape)
        print(weights)
        biases = tf.get_variable(name='_biases',
                               initializer=tf.zeros(
```

```

                shape=[n_output_units])))

print(biases)
layer = tf.matmul(input_tensor, weights)
print(layer)
layer = tf.nn.bias_add(layer, biases,
                       name='net_pre-activaiton')
print(layer)
if activation_fn is None:
    return layer
layer = activation_fn(layer, name='activation')
print(layer)
return layer

```

Die Wrapper-Funktion `fc_layer` initialisiert die Gewichtungen und Bias auf ähnliche Weise wie die `conv_layer`-Funktion und führt anschließend mit der Funktion `tf.matmul` eine Matrixmultiplikation durch. Die `fc_layer`-Funktion benötigt drei Argumente:

- `input_tensor`: Der Eingabetensor
- `name`: Die Bezeichnung der Schicht, die als Geltungsbereich verwendet wird
- `n_output_units`: Die Anzahl der Ausgabeeinheiten

Wir können die Funktion mit einem einfachen Eingabetensor wie folgt testen:

```

>>> g = tf.Graph()
>>> with g.as_default():
...     x = tf.placeholder(tf.float32,
...                         shape=[None, 28, 28, 1])
...     fc_layer(x, name='fctest', n_output_units=32,
...               activation_fn=tf.nn.relu)
>>>
>>> del g, x
<tf.Variable 'fctest/_weights:0' shape=(784, 32)
dtype=float32_ref>
<tf.Variable 'fctest/_biases:0' shape=(32,)>
dtype=float32_ref>
Tensor("fctest/MatMul:0", shape=(?, 32),
dtype=float32)
Tensor("fctest/net_pre-activaiton:0", shape=(?, 32),
dtype=float32)
Tensor("fctest/activation:0", shape=(?, 32), dtype=float32)

```

Diese Funktion verhält sich bei den beiden vollständig verknüpften Schichten in unserem Modell ein wenig anders. Die erste vollständig verknüpfte Schicht erhält ihre Eingabe direkt von einer Konvolutionsschicht, deshalb handelt es sich

noch immer um einen vierdimensionalen Tensor. Bei der zweiten vollständig verknüpften Schicht müssen wir den Eingabetensor mit der `tf.reshape`-Funktion umformen. Außerdem werden die Nettovoraktivierungen der ersten vollständig verknüpften Schicht an die ReLU-Aktivierungsfunktion übergeben, bei der zweiten jedoch entspricht dies den `logits`, daher muss eine lineare Aktivierungsfunktion verwendet werden.

Nun können wir diese Wrapper-Funktionen verwenden, um das komplette konvolutionale Netz zu erstellen. Mit dem folgenden Code definieren wir eine Funktion namens `build_cnn`, die das Erstellen des CNN-Modells handhabt:

```
def build_cnn():
    ## Platzhalter für X und y:
    tf_x = tf.placeholder(tf.float32, shape=[None, 784],
                          name='tf_x')
    tf_y = tf.placeholder(tf.int32, shape=[None],
                          name='tf_y')
    # x in einen 4-D-Tensor umformen:
    # [batchsize, width, height, 1]
    tf_x_image = tf.reshape(tf_x, shape=[-1, 28, 28, 1],
                           name='tf_x_reshaped')

    ## One-hot-Codierung:
    tf_y_onehot = tf.one_hot(indices=tf_y, depth=10,
                            dtype=tf.float32,
                            name='tf_y_onehot')

    ## 1. Schicht: Faltung_1
    print('\nErste Schicht erstellen:')
    h1 = conv_layer(tf_x_image, name='conv_1',
                    kernel_size=(5, 5),
                    padding_mode='VALID',
                    n_output_channels=32)

    ## Max-Pooling
    h1_pool = tf.nn.max_pool(h1,
                            ksize=[1, 2, 2, 1],
                            strides=[1, 2, 2, 1],
                            padding='SAME')

    ## 2. Schicht: Faltung_2
    print('\nZweite Schicht erstellen:')
    h2 = conv_layer(h1_pool, name='conv_2',
                    kernel_size=(5, 5),
                    padding_mode='VALID',
                    n_output_channels=64)

    ## MaxPooling
    h2_pool = tf.nn.max_pool(h2,
                            ksize=[1, 2, 2, 1],
```

```

        strides=[1, 2, 2, 1],
        padding='SAME')

## 3. Schicht: Vollständig verknüpft
print('\nDritte Schicht erstellen:')
h3 = fc_layer(h2_pool, name='fc_3',
               n_output_units=1024,
               activation_fn=tf.nn.relu)

## Dropout
keep_prob = tf.placeholder(tf.float32,
                           name='fc_keep_prob')
h3_drop = tf.nn.dropout(h3, keep_prob=keep_prob,
                        name='dropout_layer')

## 4. Schicht: Vollständig verknüpft (lineare Aktivierung)
print('\nVierte Schicht erstellen:')
h4 = fc_layer(h3_drop, name='fc_4',
               n_output_units=10,
               activation_fn=None)

## Vorhersage
predictions = {
    'probabilities': tf.nn.softmax(h4,
                                    name='probabilities'),
    'labels': tf.cast(tf.argmax(h4, axis=1), tf.int32,
                     name='labels')
}

## Visualisierung des Graphen mit TensorBoard:
## Verlustfunktion und Optimierung
cross_entropy_loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(
        logits=h4, labels=tf_y_onehot),
        name='cross_entropy_loss')

## Optimierung:
optimizer = tf.train.AdamOptimizer(learning_rate)
optimizer = optimizer.minimize(cross_entropy_loss,
                               name='train_op')

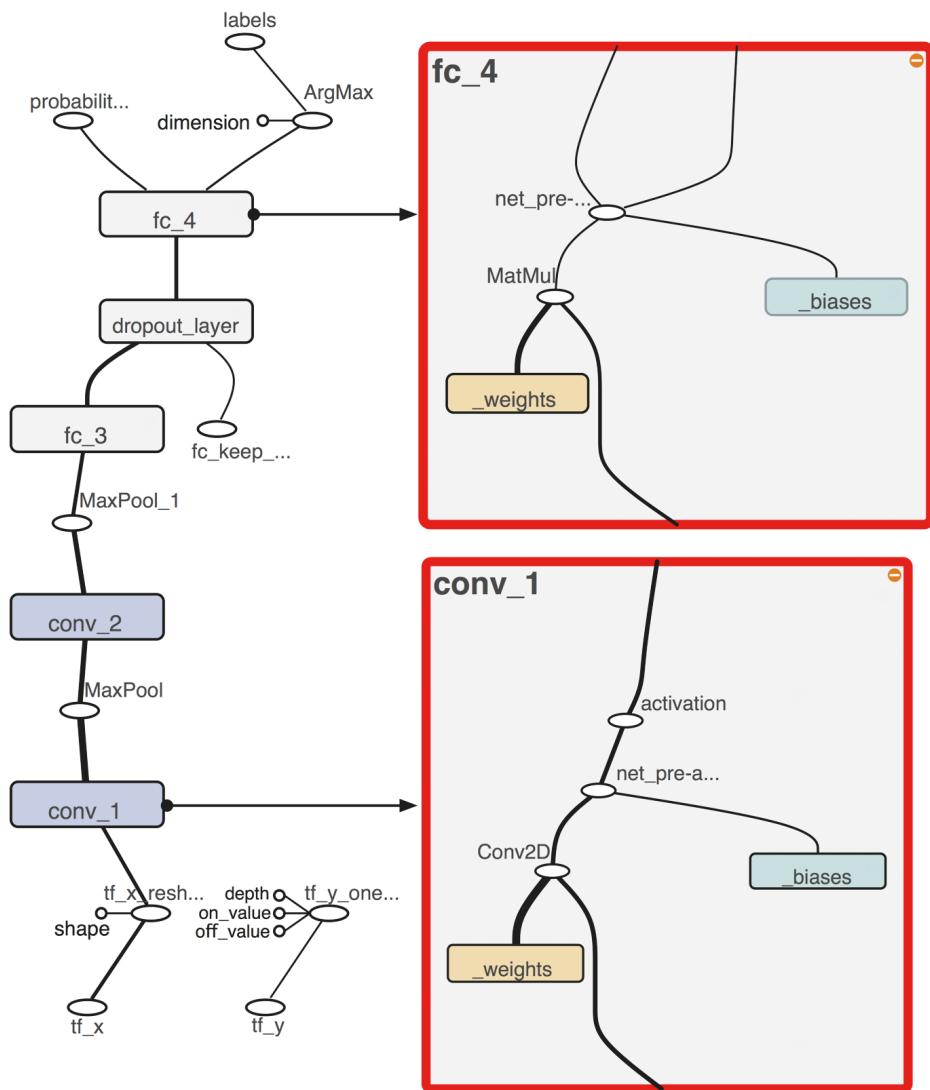
## Berechnung der Korrektklassifizierungsrate
correct_predictions = tf.equal(predictions['labels'],
                                tf_y, name='correct_preds')

accuracy = tf.reduce_mean(
    tf.cast(correct_predictions, tf.float32),
    name='accuracy')

```

Um reproduzierbare Ergebnisse zu erhalten, müssen wir die Zufallszahlgeneratoren von NumPy und TensorFlow initialisieren. In TensorFlow kann das im Graphen erledigt werden, indem man `tf.set_random_seed` im Geltungsbereich des Graphen aufruft, wie Sie später noch sehen werden. Die folgende Abbildung zeigt den mit TensorBoard visualisierten TensorFlow-Graphen des mehrschichtigen CNNs.

## Graph des mehrschichtigen CNNs



### Hinweis

In dieser Implementierung wird zum Trainieren des CNN-Modells die Funktion `tf.train.AdamOptimizer` verwendet. Der Adam-Optimierer ist ein robustes gradientenbasiertes Optimierungsverfahren für Aufgabenstellungen des Machine Learnings und nicht konvexe Optimierungen. Die Entwicklung dieses Verfahrens wurde durch zwei verbreitete Optimierungsmethoden angeregt: RMSProp und AdaGrad.

Die Adam-Optimierung hat den entscheidenden Vorteil, dass die Schrittweite bei Aktualisierungen laufend anhand des gleitenden Mittelwerts der Trägheitsterme der Gradienten abgeleitet wird. Weitere Informationen finden Sie in dem Artikel *Adam: A Method for Stochastic Optimization* von Diederik P. Kingma und Jimmy Lei Ba (2014), der unter <https://arxiv.org/abs/1412.6980> kostenlos verfügbar ist.

Wir definieren außerdem vier weitere Funktionen: `save` und `load` zum Speichern und Einlesen von Fixpunkten des trainierten Modells, `train` zum Trainieren des Modells mit der Datenmenge `training_set` sowie `predict` zur Abfrage von Wahrscheinlichkeiten oder vorhergesagter Klassenbezeichnung für die Testdatenmenge. Hier ist der entsprechende Code (die Abkürzungen DWV und KKR bedeuten Durchschnittswert der Verlustfunktion bzw. Korrektklassifizierungsrate):

```
def save(saver, sess, epoch, path='./model/'):
    if not os.path.isdir(path):
        os.makedirs(path)
    print('Modell speichern in %s' % path)
    saver.save(sess, os.path.join(path, 'cnn-model.ckpt'),
               global_step=epoch)

def load(saver, sess, path, epoch):
    print('Modell laden aus %s' % path)
    saver.restore(sess, os.path.join(
        path, 'cnn-model.ckpt-%d' % epoch))

def train(sess, training_set, validation_set=None,
          initialize=True, epochs=20, shuffle=True,
          dropout=0.5, random_seed=None):

    X_data = np.array(training_set[0])
    y_data = np.array(training_set[1])
    training_loss = []

    ## Variablen initialisieren
    if initialize:
        sess.run(tf.global_variables_initializer())

    np.random.seed(random_seed) # Mischen in batch_generator
    for epoch in range(1, epochs+1):
        batch_gen = batch_generator(
            X_data, y_data,
            shuffle=shuffle)
        avg_loss = 0.0
```

```

for i,(batch_x,batch_y) in enumerate(batch_gen):
    feed = {'tf_x:0': batch_x,
            'tf_y:0': batch_y,
            'fc_keep_prob:0': dropout}
    loss, _ = sess.run(
        ['cross_entropy_loss:0', 'train_op'],
        feed_dict=feed)
    avg_loss += loss

    training_loss.append(avg_loss / (i+1))
    print('Epoch %02d DWV Training: %7.3f' % (
        epoch, avg_loss), end=' ')
if validation_set is not None:
    feed = {'tf_x:0': validation_set[0],
            'tf_y:0': validation_set[1],
            'fc_keep_prob:0':1.0}
    valid_acc = sess.run('accuracy:0',
                         feed_dict=feed)
    print('KKR Validierung: %7.3f' % valid_acc)
else:
    print()

def predict(sess, X_test, return_proba=False):
    feed = {'tf_x:0': X_test,
            'fc_keep_prob:0': 1.0}
    if return_proba:
        return sess.run('probabilities:0', feed_dict=feed)
    else:
        return sess.run('labels:0', feed_dict=feed)

```

Nun können wir ein Graphenobjekt erzeugen, den Zufallszahlgenerator initialisieren und das CNN-Modell in diesem Graphen erstellen:

```

>>> ## Hyperparameter definieren
>>> learning_rate = 1e-4
>>> random_seed = 123
>>>
>>> ## Graphen erzeugen
>>> g = tf.Graph()
>>> with g.as_default():
...     tf.set_random_seed(random_seed)
...     ## Modell erstellen
...     build_cnn()
...

```

```
...    ## Speichern:  
...    saver = tf.train.Saver()
```

Nach dem Erstellen des Modells durch den Aufruf der `build_cnn`-Funktion erzeugt der obige Code anhand der Klasse `tf.train.Saver` ein `saver`-Objekt zum Speichern und Wiederherstellen trainierter Modelle, wie Sie es aus Kapitel 14 kennen.

Der nächste Schritt ist das Trainieren des CNN-Modells. Zu diesem Zweck müssen wir eine TensorFlow-Sitzung eröffnen und den Graphen starten. Anschließend rufen wir die Funktion `train` auf. Beim ersten Trainieren des Modells müssen alle Variablen des Netzes initialisiert werden.

Für diese Aufgabe haben wir ein Argument namens `initialize` definiert, das für die Initialisierung verantwortlich ist. Wenn `initialize` den Wert `True` besitzt, wird via `session.run` die Funktion `tf.global_variables_initializer` ausgeführt. Diese Initialisierung sollte man unterlassen, falls das Modell mit weiteren Epochen trainiert werden soll. Sie können beispielsweise ein bereits trainiertes Modell wiederherstellen und mit zusätzlichen 10 Epochen trainieren. Der Code für das erstmalige Trainieren des Modells sieht folgendermaßen aus:

```
>>> ## TF-Sitzung eröffnen und das CNN-Modell trainieren  
>>>  
>>> with tf.Session(graph=g) as sess:  
...     train(sess,  
...             training_set=(X_train_centered, y_train),  
...             validation_set=(X_valid_centered, y_valid),  
...             initialize=True,  
...             random_seed=123)  
...     save(saver, sess, epoch=20)  
Epoch 01 DWV Training: 272.772 KKR Validierung: 0.973  
Epoch 02 DWV Training: 76.053 KKR Validierung: 0.981  
Epoch 03 DWV Training: 51.309 KKR Validierung: 0.984  
Epoch 04 DWV Training: 39.740 KKR Validierung: 0.986  
Epoch 05 DWV Training: 31.508 KKR Validierung: 0.987  
...  
Epoch 19 DWV Training: 5.386 KKR Validierung: 0.991  
Epoch 20 DWV Training: 3.965 KKR Validierung: 0.992  
Modell speichern in ./model/
```

Nachdem die 20 Epochen durchgelaufen sind, wird das trainierte Modell zwecks zukünftiger Verwendung gespeichert, sodass wir es nicht jedes Mal neu trainieren müssen und Rechenzeit einsparen. Wir löschen den Graphen `g`, erzeugen einen neuen Graphen `g2` und stellen das trainierte Modell wieder her, um Vorhersagen für die Testdatenmenge zu treffen:

```

>>> ### Korrektklassifizierungsrate für die Testdaten
>>> ### berechnen; gespeichertes Modell wiederherstellen
>>>
>>> del g
>>>
>>> ## Neuen Graphen erzeugen und Modell erstellen
>>> g2 = tf.Graph()
>>> with g2.as_default():
...     tf.set_random_seed(random_seed)
...     ## Modell erstellen
...     build_cnn()
...
...     ## saver:
...     saver = tf.train.Saver()
>>>
>>> ## Neue Sitzung eröffnen und Modell wiederherstellen
>>> with tf.Session(graph=g2) as sess:
...     load(saver, sess, epoch=20, path='./model/')
...     preds = predict(sess, X_test_centered,
...                     return_proba=False)
...     print('KKR Test: %.3f%%' % (100*
...                                 np.sum(preds == y_test)/len(y_test)))
Erste Schicht erstellen:
..
Zweite Schicht erstellen:
..
Dritte Schicht erstellen:
..
Vierte Schicht erstellen:
..
KKR Test: 99.310%

```

Die Ausgabe enthält aufgrund der `print`-Anweisungen in der Funktion `build_cnn` einige zusätzliche Zeilen, die aus Platzgründen weggelassen sind. Wie Sie sehen, ist die Korrektklassifizierungsrate für die Testdatenmenge schon besser als die mit dem mehrschichtigen Perzeptron in Kapitel 13 erzielte.

Vergewissern Sie sich bitte, dass Sie `X_test_centered` verwenden, die vorverarbeitete Version der Testdaten. Wenn Sie `X_test` verwenden, fällt die Korrektklassifizierungsrate niedriger aus.

Nun wollen wir uns die vorhergesagten Klassenbezeichnungen und die Wahrscheinlichkeiten der ersten zehn Objekte ansehen. Die Vorhersagen sind bereits in `preds` gespeichert, aber da es von Vorteil ist, Erfahrung bei der Verwendung der Sitzung und dem Starten des Graphen zu sammeln, wiederholen wir die Schritte hier noch einmal:

```
>>> ## Vorhersagen für die Testdatenmenge treffen
>>> np.set_printoptions(precision=2, suppress=True)
>>>
>>> with tf.Session(graph=g2) as sess:
...     load(saver, sess, epoch=20, path='./model/')
...     print(predict(sess, X_test_centered[:10], return_proba=False))
...     print(predict(sess, X_test_centered[:10], return_proba=True))
Modell laden aus ./model/
INFO:tensorflow:Restoring parameters from ./model/cnn-model.ckpt-20
[7 2 1 0 4 1 4 9 5 9]
[[ 0.    0.    0.    0.    0.    0.    0.    1.    0.    0.   ]
 [ 0.    0.    1.    0.    0.    0.    0.    0.    0.    0.   ]
 [ 0.    1.    0.    0.    0.    0.    0.    0.    0.    0.   ]
 [ 1.    0.    0.    0.    0.    0.    0.    0.    0.    0.   ]
 [ 0.    0.    0.    0.    1.    0.    0.    0.    0.    0.   ]
 [ 0.    1.    0.    0.    0.    0.    0.    0.    0.    0.   ]
 [ 0.    0.    0.    0.    1.    0.    0.    0.    0.    0.   ]
 [ 0.    0.    0.    0.    0.    0.    0.    0.    0.    1.   ]
 [ 0.    0.    0.    0.    0.    0.99  0.01  0.    0.    0.   ]
 [ 0.    0.    0.    0.    0.    0.    0.    0.    0.    1.   ]]
```

Zum Abschluss trainieren wir das Modell weiter, bis es 40 Epochen erreicht. Da wir bereits 20 Epochen mit den initialisierten Gewichtungen und Bias trainiert haben, können wir durch die Wiederherstellung des trainierten Modells Zeit einsparen und das Modell mit weiteren 20 Epochen trainieren. Das ist ganz einfach: Wir brauchen nur die `train`-Funktion erneut aufzurufen, geben dann aber `initialize=False` an, um den Initialisierungsschritt zu überspringen. Hier der Code:

```
## Training mit weiteren 20 Epochen fortsetzen
## Keine erneute Initialisierung: initialize=False
## Neue Sitzung eröffnen und Modell wiederherstellen
with tf.Session(graph=g2) as sess:
    load(saver, sess, epoch=20, path='./model/')
    train(sess,
          training_set=(X_train_centered, y_train),
          validation_set=(X_valid_centered, y_valid),
          initialize=False,
          epochs=20,
          random_seed=123)
    save(saver, sess, epoch=40, path='./model/')
    preds = predict(sess, X_test_centered,
                    return_proba=False)
    print('Korrektklassifizierungsrate Test: %.3f%%' % (100*
                                                       np.sum(preds == y_test)/len(y_test)))
```

Das Ergebnis zeigt, dass das Trainieren von 20 weiteren Epochen bei der Testdatenmenge eine leichte Leistungsverbesserung auf 99,37 Prozent bringt.

In diesem Abschnitt haben wir ein mehrschichtiges konvolutionales neuronales Netz mit TensorFlows Low-level-API implementiert. Im nächsten Abschnitt werden wir dasselbe Netz implementieren, aber dafür TensorFlows Layers-API verwenden.

#### 15.3.4 Implementierung eines CNNs mit TensorFlows Layers-API

Zwecks Implementierung des CNNs mit TensorFlows Layers-API müssen wir die selben Schritte zum Einlesen und Vorverarbeiten der Daten wie im letzten Abschnitt durchführen, um `X_train_centered`, `X_valid_centered` und `X_test_centered` zu erhalten. Anschließend können wir das Modell in einer neuen Klasse implementieren:

```
import tensorflow as tf
import numpy as np

class ConvNN(object):
    def __init__(self, batchsize=64,
                 epochs=20, learning_rate=1e-4,
                 dropout_rate=0.5,
                 shuffle=True, random_seed=None):
        np.random.seed(random_seed)
        self.batchsize = batchsize
        self.epochs = epochs
        self.learning_rate = learning_rate
        self.dropout_rate = dropout_rate
        self.shuffle = shuffle

        g = tf.Graph()
        with g.as_default():
            ## Zufallszahlgenerator initialisieren:
            tf.set_random_seed(random_seed)

            ## Netz erstellen:
            self.build()

            ## Initialisierer
            self.init_op = tf.global_variables_initializer()

            ## Saver
            self.saver = tf.train.Saver()
```

```

## Sitzung eröffnen
self.sess = tf.Session(graph=g)

def build(self):
    ## Platzhalter für X und y:
    tf_x = tf.placeholder(tf.float32,
                          shape=[None, 784],
                          name='tf_x')
    tf_y = tf.placeholder(tf.int32,
                          shape=[None],
                          name='tf_y')
    is_train = tf.placeholder(tf.bool,
                             shape=(),
                             name='is_train')

    ## x in einen 4D-Tensor umformen:
    ## [batchsize, width, height, 1]
    tf_x_image = tf.reshape(tf_x, shape=[-1, 28, 28, 1],
                           name='input_x_2dimages')
    ## One-hot-Codierung:
    tf_y_onehot = tf.one_hot(indices=tf_y, depth=10,
                            dtype=tf.float32,
                            name='input_y_onehot')

    ## 1. Schicht: Faltung_1
    h1 = tf.layers.conv2d(tf_x_image,
                         kernel_size=(5, 5),
                         filters=32,
                         activation=tf.nn.relu)

    ## Max-Pooling
    h1_pool = tf.layers.max_pooling2d(h1,
                                      pool_size=(2, 2),
                                      strides=(2, 2))

    ## 2. Schicht: Faltung_2
    h2 = tf.layers.conv2d(h1_pool, kernel_size=(5, 5),
                         filters=64,
                         activation=tf.nn.relu)

    ## Max-Pooling
    h2_pool = tf.layers.max_pooling2d(h2,
                                      pool_size=(2, 2),
                                      strides=(2, 2))

    ## 3. Schicht: Vollständig verknüpft
    input_shape = h2_pool.get_shape().as_list()
    n_input_units = np.prod(input_shape[1:])

```

```
h2_pool_flat = tf.reshape(h2_pool,
                           shape=[-1, n_input_units])
h3 = tf.layers.dense(h2_pool_flat, 1024,
                     activation=tf.nn.relu)

## Dropout
h3_drop = tf.layers.dropout(h3,
                            rate=self.dropout_rate,
                            training=is_train)

## 4. Schicht: Vollständig verknüpft
##           (lineare Aktivierung)
h4 = tf.layers.dense(h3_drop, 10,
                     activation=None)

## Vorhersage
predictions = {
    'probabilities': tf.nn.softmax(h4,
                                    name='probabilities'),
    'labels': tf.cast(tf.argmax(h4, axis=1),
                      tf.int32, name='labels')
}

## Verlustfunktion und Optimierung
cross_entropy_loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(
        logits=h4, labels=tf_y_onehot),
    name='cross_entropy_loss')

## Optimierer:
optimizer = tf.train.AdamOptimizer(self.learning_rate)
optimizer = optimizer.minimize(cross_entropy_loss,
                               name='train_op')

## Korrektklassifizierungsrate ermitteln
correct_predictions = tf.equal(predictions['labels'],
                                tf_y,
                                name='correct_preds')

accuracy = tf.reduce_mean(
    tf.cast(correct_predictions, tf.float32),
    name='accuracy')

def save(self, epoch, path='./tflayers-model/'):
    if not os.path.isdir(path):
```

```

        os.makedirs(path)
        print('Modell speichern in %s' % path)
        self.saver.save(self.sess,
                        os.path.join(path, 'model.ckpt'),
                        global_step=epoch)

    def load(self, epoch, path):
        print('Modell laden aus %s' % path)
        self.saver.restore(self.sess,
                           os.path.join(path, 'model.ckpt-%d' % epoch))

    def train(self, training_set,
              validation_set=None,
              initialize=True):
        ## Variablen initialisieren
        if initialize:
            self.sess.run(self.init_op)

        self.train_cost_ = []
        X_data = np.array(training_set[0])
        y_data = np.array(training_set[1])

        for epoch in range(1, self.epochs + 1):
            batch_gen = batch_generator(X_data, y_data,
                                         shuffle=self.shuffle)
            avg_loss = 0.0
            for i, (batch_x,batch_y) in enumerate(batch_gen):
                feed = {'tf_x:0': batch_x,
                        'tf_y:0': batch_y,
                        'is_train:0': True} ## für Dropout
                loss, _ = self.sess.run(
                    ['cross_entropy_loss:0', 'train_op'],
                    feed_dict=feed)
                avg_loss += loss

            print('Epoche %02d: DWV Training: '
                  '%7.3f' % (epoch, avg_loss), end=' ')
            if validation_set is not None:
                feed = {'tf_x:0': batch_x,
                        'tf_y:0': batch_y,
                        'is_train:0': False} ## für Dropout
                valid_acc = self.sess.run('accuracy:0',
                                         feed_dict=feed)
                print('KKR Validierung: %7.3f' % valid_acc)
            else:

```

```

print()

def predict(self, X_test, return_proba = False):
    feed = {'tf_x:0': X_test,
            'is_train:0': False} ## für Dropout
    if return_proba:
        return self.sess.run('probabilities:0',
                             feed_dict=feed)
    else:
        return self.sess.run('labels:0',
                             feed_dict=feed)

```

Die Struktur dieser Klasse ist dem letzten Abschnitt mit TensorFlows Low-level-API sehr ähnlich. Die Klasse besitzt einen Konstruktor, der die Trainingsparameter einstellt, einen Graphen `g` erzeugt und das Modell erstellt. Abgesehen vom Konstruktor gibt es fünf bedeutsame Methoden:

- `.build`: Erstellt das Modell
- `.save`: Speichert das trainierte Modell
- `.load`: Stellt ein gespeichertes Modell wieder her
- `.train`: Trainiert das Modell
- `.predict`: Trifft Vorhersagen für die Testdatenmenge

Wie in der Implementierung im letzten Abschnitt haben wir nach der ersten vollständig verknüpften Schicht eine Dropout-Schicht verwendet. In der Implementierung mit TensorFlows Low-level-API haben wir die Funktion `tf.nn.dropout` benutzt, hier hingegen verwenden wir `tf.layers.dropout`, eine Wrapper-Funktion für `tf.nn.dropout`. Zwischen diesen beiden Funktionen gibt es zwei wesentliche Unterschiede, die wir berücksichtigen müssen:

- `tf.nn.dropout` nimmt ein Argument namens `keep_prob` entgegen, das die Wahrscheinlichkeit angibt, die Einheiten zu behalten, `tf.layers.dropout` besitzt dagegen einen Parameter `rate`, der die Quote angibt, mit der Einheiten entfernt werden – deshalb gilt `rate = 1 - keep_prob`.
- Bei der `tf.nn.dropout`-Funktion führen wir dem `keep_prob`-Parameter über einen Platzhalter Werte zu. Während des Trainings werden wir `keep_prob=0.5` verwenden, während des anschließenden Auswertungsmodus (beim Treffen von Vorhersagen) verwenden wir `keep_prob=1`. Bei der Funktion `tf.layers.dropout` wird der Wert des Parameters `rate` allerdings schon beim Erstellen der Dropout-Schicht im Graphen bereitgestellt und lässt sich während des Trainings oder im Auswertungsmodus nicht mehr ändern. Stattdessen müssen wir ein boolesches Argument namens `training` übergeben, um festzulegen, ob ein Dropout stattfinden soll oder nicht. Diese Aufgabe kann mit einem Platzhalter des Typs `tf.bool` erledigt werden, dem wir im Trainingsmodus den Wert `True` und im Auswertungsmodus den Wert `False` zuführen.

Nun erstellen wir eine Instanz der Klasse ConvNN, trainieren sie mit 20 Epochen und speichern das Modell. Hier ist der dazugehörige Code:

```
>>> cnn = ConvNN(random_seed=123)
>>>
>>> ## Modell trainieren
>>> cnn.train(training_set=(X_train_centered, y_train),
...             validation_set=(X_valid_centered, y_valid),
...             initialize=True)
>>> cnn.save(epoch=20)
```

Nach dem Abschluss des Trainings kann das Modell wie folgt verwendet werden, um Vorhersagen für die Testdatenmenge zu treffen:

```
>>> del cnn
>>>
>>> cnn2 = ConvNN(random_seed=123)
>>> cnn2.load(epoch=20, path='./tflayers-model/')
>>>
>>> print(cnn2.predict(X_test_centered[:10, :]))
Modell laden aus ./tflayers-model/
INFO:tensorflow:Restoring parameters from
./tflayers-model/model.ckpt-20
[7 2 1 0 4 1 4 9 5 9]
```

Die Korrektklassifizierungsrate für die Testdatenmenge können wir folgendermaßen ermitteln:

```
>>> preds = cnn2.predict(X_test_centered)
>>>
>>> print('KKR Test: %.2f%%' % (100*
...           np.sum(y_test == preds)/len(y_test)))
KKR Test: 99.32%
```

Die erzielte Korrektklassifizierungsrate beträgt 99,32 Prozent. Es werden also nur 68 Objekte fehlklassifiziert!

Damit sind die Implementierungen konvolutionaler neuronaler Netze mit TensorFlows Low-level- und Layers-API abgeschlossen. Bei der ersten Implementierung haben wir mit der Low-level-API Wrapper-Funktionen definiert. Die zweite Implementierung erwies sich als unkomplizierter, weil wir zum Erstellen der Konvolutionsschicht und der vollständig verknüpften Schicht die Funktionen `tf.layers.conv2d` und `tf.layers.dense` verwenden konnten.

## 15.4 Zusammenfassung

In diesem Kapitel haben wir uns mit CNNs (konvolutionalen neuronalen Netzen) befasst und die Bausteine verschiedener CNN-Architekturen erkundet. Wir haben zunächst Faltungsoperationen definiert und anhand der Erörterung ein- und zweidimensionaler Implementierungen die Grundlagen kennengelernt.

Außerdem haben wir zwei Arten von Pooling-Operationen kennengelernt, nämlich Max-Pooling und Mean-Pooling. Dann haben wir mithilfe dieser Bausteine unter Verwendung von TensorFlows Low-level- und Layers-API ein tiefes konvolutionales neuronales Netz zur Bilderkennung implementiert.

Im nächsten Kapitel fahren wir mit rekurrenten (rückgekoppelten) neuronalen Netzen (RNNs) fort, die zur Erkennung der Struktur sequenzieller Daten eingesetzt werden. Dafür gibt es einige faszinierende Anwendungen wie Sprachübersetzung und das automatische Betiteln von Bildern.



# Modellierung sequenzieller Daten durch rekurrente neuronale Netze

Im letzten Kapitel ging es vor allem um konvolutionale neuronale Netze zur Bildklassifizierung. In diesem Kapitel werden wir *rekurrente neuronale Netze* (RNNs) erkunden und ihre Anwendung bei der Modellierung sequenzieller Daten sowie eine bestimmte Teilmenge solcher Daten betrachten, nämlich Zeitreihen. In diesem Kapitel behandeln wir die folgenden Themen:

- Sequenzielle Daten
- RNNs für die Modellierung von Sequenzen
- Long Short-Term Memory (LSTM, zu deutsch etwa »Langes Kurzzeitgedächtnis«)
- Truncated Backpropagation-Through-Time (TBPTT)
- Implementierung eines mehrschichtigen RNNs zur Modellierung von Sequenzen mit TensorFlow
- Projekt 1: Analyse der Stimmungslage in der IMDb-Filmbewertungsdatenbank mit einem RNN
- Projekt 2: Sprachmodellierung durch Zeichen mit LSTM-Zellen unter Verwendung des Texts von Shakespeares Hamlet
- Gradienten-Clipping zur Umgehung des Problems eines explosionsartig wachsenden Gradienten

Auf unserer Tour durch das Machine Learning mit Python ist dieses Kapitel das abschließende, daher soll noch einmal zusammengefasst werden, was wir im Verlauf des Buches über Machine Learning und Deep Learning in Erfahrung gebracht haben und uns schließlich zu RNNs führte. Zum Abschluss möchten die Autoren einige Links mit Ihnen teilen, bei denen es um die Initiativen und führenden Köpfe auf diesem wunderbaren Forschungsgebiet geht, damit Sie Ihre Tour durch das Machine Learning und Deep Learning fortsetzen können.

## 16.1 Sequenzielle Daten

Betrachten wir zunächst einmal die Eigenheiten sequenzieller Daten bzw. einer *Sequenz*. Wir werden uns die speziellen Eigenschaften ansehen, durch die sich Sequenzen von anderen Daten unterscheiden. Anschließend werden wir untersu-

chen, wie sich sequenzielle Daten repräsentieren lassen, und die verschiedenen auf der Ein- und Ausgabe beruhenden Modellkategorien für sequenzielle Daten erkunden. Das wird sich als hilfreich erweisen, wenn wir später in diesem Kapitel die Beziehungen zwischen RNNs und Sequenzen eingehender betrachten.

### 16.1.1 Modellierung sequenzieller Daten: Die Reihenfolge ist von Bedeutung

Sequenzen unterscheiden sich von anderen Daten dadurch, dass ihre Elemente in einer bestimmten Reihenfolge angeordnet und somit nicht voneinander unabhängig sind.

Wie Sie aus Kapitel 6 wissen, erwarten typische überwachte Machine-Learning-Algorithmen, dass die Eingabedaten voneinander *unabhängig* und *gleichmäßig verteilt* sind. Wenn beispielsweise  $n$  Datenobjekte  $x^{(1)}, x^{(2)}, \dots, x^{(n)}$  vorliegen, dann spielt die Reihenfolge, in der wir die Daten zum Trainieren des Lernalgorithmus verwenden, keine Rolle.

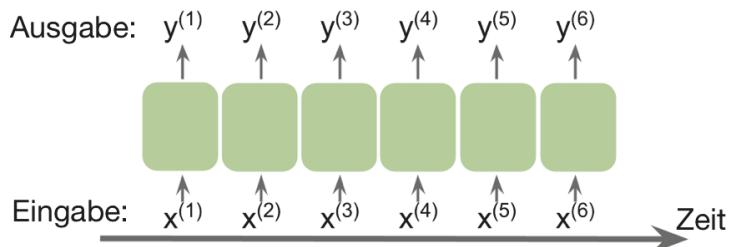
Wenn wir es jedoch mit Sequenzen zu tun haben, sind diese Annahmen nicht mehr gültig – die Reihenfolge ist *per definitionem* von Bedeutung.

### 16.1.2 Repräsentierung von Sequenzen

Sequenzen sind also Eingabedaten, deren Reihenfolge nicht voneinander unabhängig ist. Wir müssen nach Möglichkeiten suchen, diese wertvolle Information für unser Lernmodell nutzbar zu machen.

In diesem Kapitel werden Sequenzen als  $(x^{(1)}, x^{(2)}, \dots, x^{(T)})$  dargestellt. Die hochgestellten Indizes bezeichnen die Positionen der Instanzen und die Länge der Sequenz ist  $T$ . Stellen Sie sich als Beispiel für eine Sequenz eine Zeitreihe vor, bei der die Datenpunkte  $x^{(t)}$  jeweils zu einem bestimmten Zeitpunkt  $t$  gehören.

Die folgende Abbildung zeigt ein Beispiel für eine Zeitreihe, in der die  $x$ - und  $y$ -Werte auf natürliche Weise gemäß ihrer Zeitachse aufeinanderfolgen. Sowohl die  $x$ - als auch die  $y$ -Werte sind also Sequenzen:



Die bislang erläuterten Standardmodelle neuronaler Netze, wie MLPs und CNNs, sind nicht in der Lage, die *Reihenfolge* der Eingabeobjekte zu berücksichtigen.

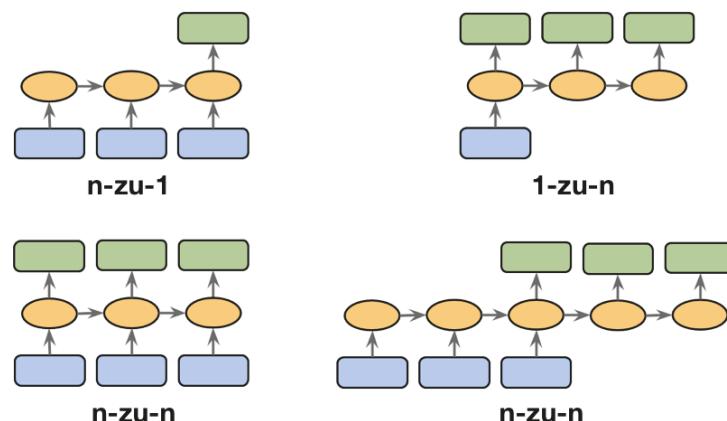
Anschaulich könnte man sagen, dass solche Modelle sich nicht an die bereits verarbeiteten Objekte *erinnern*. Beispielsweise durchlaufen die Objekte die Feedforward- und Backpropagation-Schritte und die Gewichtungen werden unabhängig von der Verarbeitungsreihenfolge aktualisiert.

Im Gegensatz dazu sind RNNs dafür ausgelegt, Sequenzen zu modellieren, und dazu fähig, früher erhaltene Informationen zu speichern und neue Vorgänge dementsprechend zu verarbeiten.

### 16.1.3 Verschiedene Kategorien der Sequenzmodellierung

Für die Sequenzmodellierung gibt es viele faszinierende Anwendungen, wie etwa Sprachübersetzungen (z.B. vom Englischen ins Deutsche), automatisches Betiteln von Bildern oder Texterzeugung.

Wir müssen die verschiedenen Arten von Aufgaben, die sich durch Sequenzmodellierung lösen lassen, jedoch genau verstehen, um ein passendes Modell entwickeln zu können. Die folgende Abbildung beruht auf den Erklärungen in dem ausgezeichneten Artikel *The Unreasonable Effectiveness of Recurrent Neural Networks* von Andrej Karpathy (siehe <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>). Sie zeigt die verschiedenen Kategorien der Beziehungen zwischen Ein- und Ausgabedaten.



Sehen wir uns die Ein- und Ausgaben hier genauer an. Wenn weder die Eingabe noch die Ausgabedaten Sequenzen darstellen, haben wir es mit Standarddaten zu tun und wir können eine der bekannten Methoden zur Modellierung der Daten verwenden. Wenn jedoch Ein- oder Ausgabedaten eine Sequenz bilden, gehören die Daten zu einer der folgenden drei Kategorien:

- **n-zu-1:** Die Eingabedaten bilden eine Sequenz, die Ausgabe ist jedoch ein Vektor fester Größe, keine Sequenz. Bei der Stimmungsanalyse beispielsweise besteht die Eingabe aus Text und die Ausgabe ist eine Klassenbezeichnung.

- **1-zu-n:** Die Eingabe liegt als Standarddaten vor, nicht als Sequenz, die Ausgabe ist jedoch eine Sequenz. Ein Beispiel für diese Kategorie ist die automatische Betitelung von Bildern – die Eingabe ist ein Bild, die Ausgabe ein sprachlicher Ausdruck.
- **n-zu-n:** Sowohl bei der Eingabe als auch bei der Ausgabe handelt es sich um Sequenzen. Diese Kategorie kann noch weiter unterteilt werden, je nachdem ob die Ein- und Ausgabe *synchron* erfolgt oder nicht. Ein Beispiel für eine *synchrone n-zu-n*-Modellierungsaufgabe ist eine Klassifizierung von Videos, bei der alle Einzelbilder des Videos klassifiziert werden. Ein Beispiel für eine *verzögerte n-zu-n*-Beziehung ist die Übersetzung von einer Sprache in eine andere. Beispielsweise muss ein englischer Satz erst vollständig eingelesen und verarbeitet werden, bevor er ins Deutsche übersetzt werden kann.

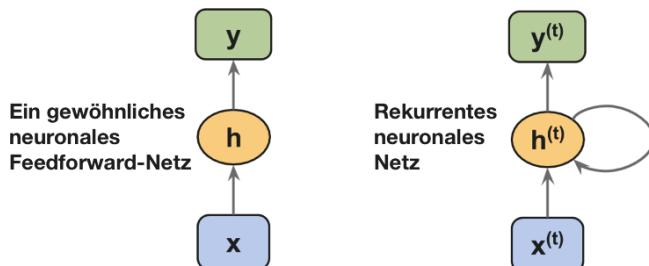
Nun kennen Sie die Kategorien der Sequenzmodellierung und wir können mit der Betrachtung der Struktur eines RNNs fortfahren.

## 16.2 Sequenzmodellierung mit RNNs

Mit dem Wissen über Sequenzen können wir in diesem Abschnitt die Grundlagen von RNNs erörtern. Zunächst einmal betrachten wir die typische Struktur eines RNNs und untersuchen, wie die Daten darin durch die verdeckten Schichten fließen. Anschließend befassen wir uns mit der Berechnung der Aktivierung der Neuronen eines typischen RNNs. In diesem Zusammenhang werden wir die üblicherweise auftretenden Probleme beim Trainieren von RNNs erörtern und die moderne Lösung für diese Probleme erkunden: LSTM.

### 16.2.1 Struktur und Ablauf eines RNNs

Wir werfen zunächst einen Blick auf die Architektur eines RNNs. In der folgenden Abbildung ist einem gewöhnlichen neuronalen Feedforward-Netz zum Vergleich ein RNN gegenüberstellt.



Beide Netze besitzen lediglich eine verdeckte Schicht. Hier sind die Einheiten nicht dargestellt, aber wir gehen davon aus, dass die Eingabeschicht ( $x$ ), die verdeckte Schicht ( $h$ ) und die Ausgabeschicht ( $y$ ) Vektoren sind, die viele Einheiten enthalten.

**Hinweis**

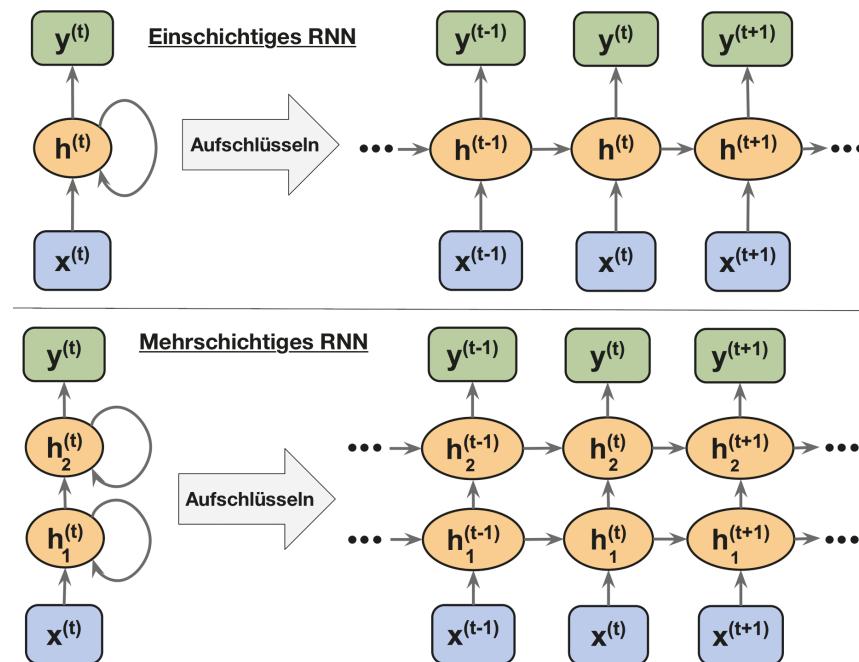
Diese allgemeine RNN-Architektur könnte zu den beiden Kategorien der Sequenzmodellierung gehören, in denen die Eingabe eine Sequenz bildet. Es könnte sich um eine n-zu-n-Beziehung handeln, wenn wir  $y^{(t)}$  als die endgültige Ausgabe betrachten, oder um eine n-zu-1-Beziehung, sofern nur das letzte Element von  $y^{(t)}$  als endgültiges Ergebnis erachtet wird.

Sie werden später noch sehen, dass die Ausgabesequenz  $y^{(t)}$  in eine normale, nicht sequenzielle Ausgabe umgewandelt werden kann.

In einem gewöhnlichen neuronalen Feedforward-Netz fließen Informationen von der Eingabeschicht in die verdeckte Schicht und anschließend weiter von der verdeckten Schicht zur Ausgabeschicht. In einem rekurrenten Netz erhält die verdeckte Schicht hingegen sowohl von der Eingabeschicht als auch von der verdeckten Schicht des vorangegangenen Zeitschritts Eingaben.

Der bei aufeinanderfolgenden Zeitschritten in der verdeckten Schicht stattfindende Informationsfluss ermöglicht es dem Netz, sich an vorhergehende Ereignisse zu »erinnern«. Dieser Informationsfluss wird für gewöhnlich als Schleife dargestellt und in einem Graphen auch als *rekurrente Kante* bezeichnet, die für die allgemeine Architektur namensgebend war.

In der folgenden Abbildung veranschaulichen ein einschichtiges und ein mehrschichtiges Netz die unterschiedlichen Architekturen.



Um die Architektur von RNNs und den Informationsfluss zu untersuchen, kann die kompakte Repräsentierung durch eine rekurrente Kante wie in der Abbildung aufgeschlüsselt werden.

Wie Sie wissen, erhält jede verdeckte Einheit in einem gewöhnlichen neuronalen Netz nur eine Eingabe, nämlich die der Eingabeschicht zugeordnete Nettovoraktivierung. Im Gegensatz dazu erhält jede verdeckte Einheit in einem RNN zwei *verschiedene* Eingaben – die Voraktivierung der Eingabeschicht und die Aktivierung derselben verdeckten Schicht des vorangegangenen Schritts zum Zeitpunkt  $t-1$ .

Beim ersten Zeitschritt zum Zeitpunkt  $t=0$  werden die verdeckten Einheiten mit Nullen oder kleinen zufälligen Werten initialisiert. Anschließend erhalten die verdeckten Einheiten bei einem Zeitschritt mit  $t > 0$  als Eingabe die Datenpunkte zum aktuellen Zeitpunkt  $\mathbf{x}^{(t)}$  und die letzten Werte der verdeckten Einheiten zum Zeitpunkt  $t-1$ , die als  $\mathbf{h}^{(t-1)}$  notiert sind.

Im Fall des mehrschichtigen RNNs können wir den Informationsfluss wie folgt zusammenfassen:

- *Schicht = 1*: Hier wird die verdeckte Schicht als  $\mathbf{h}_1^{(t)}$  notiert und erhält als Eingabe den Datenpunkt  $\mathbf{x}^{(t)}$  und den Wert  $\mathbf{h}_1^{(t-1)}$  derselben Schicht beim vorangegangenen Zeitschritt.
- *Schicht = 2*: Die zweite verdeckte Schicht  $\mathbf{h}_2^{(t)}$  erhält ihre Eingabe von den verdeckten Einheiten der darunter liegenden Schicht zum aktuellen Zeitpunkt ( $\mathbf{h}_1^{(t)}$ ) und den eigenen Werten  $\mathbf{h}_2^{(t-1)}$  beim vorangegangenen Zeitschritt.

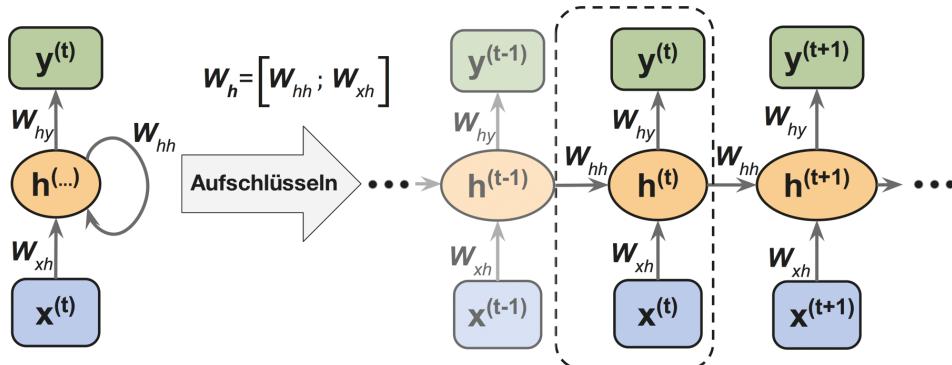
### 16.2.2 Aktivierungen eines RNNs berechnen

Nun kennen wir die Struktur und den Informationsfluss in einem RNN und können ins Detail gehen und die Aktivierungen der verdeckten Schicht sowie der Ausgabeschicht berechnen. Der Einfachheit halber betrachten wir nur eine einzelne verdeckte Schicht, dieselben Überlegungen treffen aber auch auf mehrschichtige RNNs zu.

Jeder gerichteten Kante (die Verbindungen zwischen den Kästen) in der soeben betrachteten Darstellung eines RNNs ist eine Gewichtungsmatrix zugeordnet. Diese Gewichtungen sind nicht von der Zeit  $t$  abhängig und bleiben deshalb entlang der Zeitachse unverändert. In einem einschichtigen RNN gibt es die folgenden Gewichtungsmatrizen:

- $\mathbf{W}_{\text{xh}}$ : Die Gewichtungsmatrix zur Verknüpfung der Eingabe  $\mathbf{x}^{(t)}$  und der verdeckten Schicht  $\mathbf{h}$
- $\mathbf{W}_{\text{hh}}$ : Die der rekurrenten Kante zugeordnete Gewichtungsmatrix
- $\mathbf{W}_{\text{hy}}$ : Die Gewichtungsmatrix zur Verknüpfung der verdeckten Schicht und der Ausgabeschicht

In der folgenden Abbildung sind diese Gewichtungsmatrizen dargestellt.



Manche Implementierungen verknüpfen die Gewichtungsmatrizen  $\mathbf{W}_{\text{xh}}$  und  $\mathbf{W}_{\text{hh}}$  zu einer kombinierten Matrix  $\mathbf{W}_h = [\mathbf{W}_{\text{xh}}; \mathbf{W}_{\text{hh}}]$ . Wir werden von dieser Notation später ebenfalls Gebrauch machen.

Die Berechnung der Aktivierungen erfolgt auf sehr ähnliche Weise wie beim mehrschichtigen Perzeptron und anderen Arten neuronaler Feedforward-Netze. Die Nettoeingabe der verdeckten Schicht  $\mathbf{z}_h$  (die Voraktivierung) wird mit einer Linearkombination berechnet. Das heißt, wir berechnen die Summe der Produkte der Gewichtungsmatrizen mit den dazugehörigen Vektoren und addieren die Bias-Einheit:  $\mathbf{z}_h^{(t)} = \mathbf{W}_{\text{xh}} \mathbf{x}^{(t)} + \mathbf{W}_{\text{hh}} \mathbf{h}^{(t-1)} + \mathbf{b}_h$ . Anschließend werden die Aktivierungen der verdeckten Einheiten für den aktuellen Zeitschritt  $t$  wie folgt berechnet:

$$\mathbf{h}^{(t)} = \phi_h(\mathbf{z}_h^{(t)}) = \phi_h(\mathbf{W}_{\text{xh}} \mathbf{x}^{(t)} + \mathbf{W}_{\text{hh}} \mathbf{h}^{(t-1)} + \mathbf{b}_h)$$

Hier ist  $\mathbf{b}_h$  der Bias-Vektor der verdeckten Einheiten und  $\phi_h(\cdot)$  ist die Aktivierungsfunktion der verdeckten Schicht.

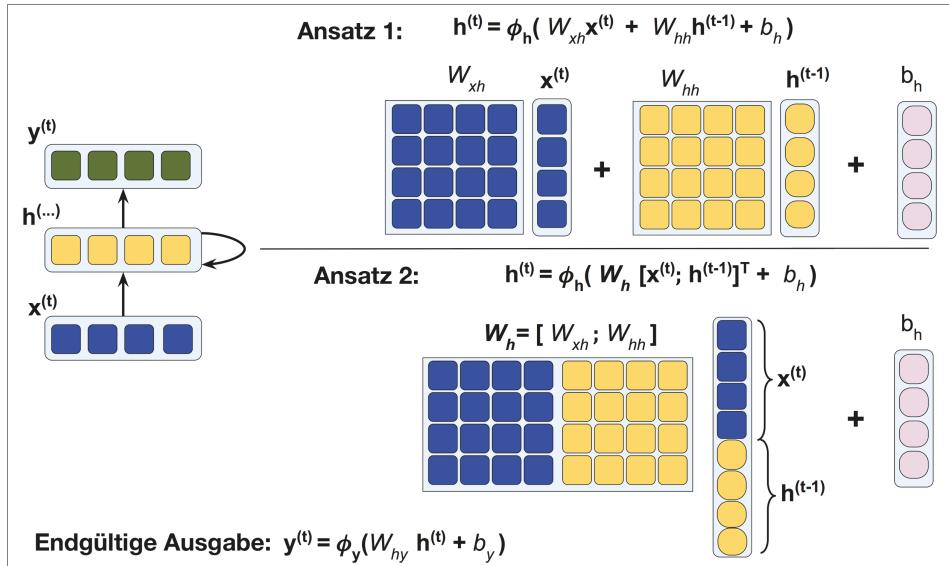
Falls Sie die verknüpfte Gewichtungsmatrix  $\mathbf{W}_h = [\mathbf{W}_{\text{xh}}; \mathbf{W}_{\text{hh}}]$  verwenden möchten, ändert sich die Gleichung zur Berechnung der verdeckten Einheiten folgendermaßen:

$$\mathbf{h}^{(t)} = \phi_h\left([\mathbf{W}_{\text{xh}}; \mathbf{W}_{\text{hh}}] \begin{bmatrix} \mathbf{x}^{(t)} \\ \mathbf{h}^{(t-1)} \end{bmatrix} + \mathbf{b}_h\right)$$

Sobald die Aktivierungen der verdeckten Einheiten zum aktuellen Zeitpunkt berechnet sind, ergeben sich die Aktivierungen der Ausgabeeinheiten wie folgt:

$$\mathbf{y}^{(t)} = \phi_y(\mathbf{W}_{\text{hy}} \mathbf{h}^{(t)} + \mathbf{b}_y)$$

Um das weiter zu verdeutlichen, zeigt die folgende Abbildung die Berechnung der Aktivierungen mit beiden Ansätzen:



## Hinweis

### Trainieren von RNNs mit BPTT

Dieser Lernalgorithmus für RNNs wurde 1990 in dem Artikel *Backpropagation Through Time: What It does and How to Do It* von Paul Werbos vorgestellt (Proceedings of IEEE, 78(10): Seiten 1550–1560, 1990).

Die Herleitung des Gradienten ist ein wenig kompliziert, aber die eigentliche Idee ist, dass sich der Gesamtverlust  $L$  als Summe der Verlustfunktionen zu den Zeitpunkten  $t=1$  bis  $t=T$  ergibt:

$$L = \sum_{t=1}^T L^{(t)}$$

Der Verlust zum Zeitpunkt  $t$  ist von den verdeckten Einheiten aller vorhergehenden Zeitschritte abhängig, somit ergibt für den Gradienten:

$$\frac{\partial L^{(t)}}{\partial \mathbf{W}_{hh}} = \frac{\partial L^{(t)}}{\partial \mathbf{y}^{(t)}} \times \frac{\partial \mathbf{y}^{(t)}}{\partial \mathbf{h}^{(t)}} \times \left( \sum_{k=1}^t \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} \times \frac{\partial \mathbf{h}^{(k)}}{\partial \mathbf{W}_{hh}} \right)$$

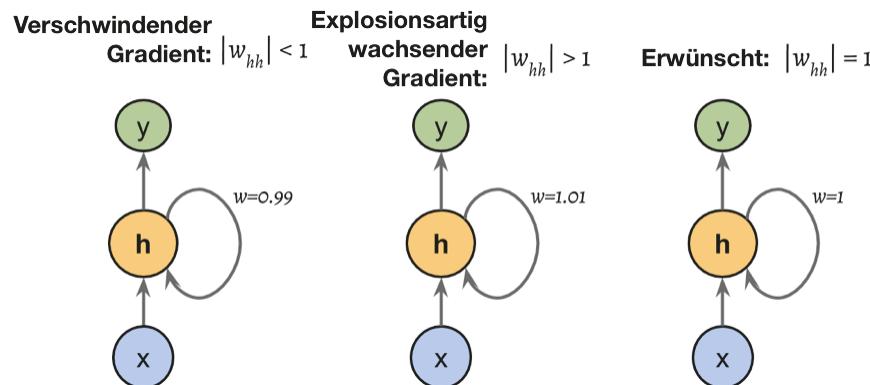
Hier wird  $\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}}$  als das Produkt der aufeinanderfolgenden Zeitschritte berechnet:

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} = \prod_{i=k+1}^t \frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{h}^{(i-1)}}$$

### 16.2.3 Probleme bei der Erkennung weitreichender Interaktionen

Die im letzten Kasten kurz erwähnte Backpropagation-Through-Time (BPTT) bringt einige neue Herausforderungen mit sich.

Der Faktor  $\frac{\partial h^{(t)}}{\partial h^{(k)}}$  in der Gleichung zur Berechnung des Gradienten der Verlustfunktion führt zum sogenannten *Problem des verschwindenden* oder *explosionsartig wachsenden Gradienten*. Die folgende Abbildung, die der Einfachheit halber ein RNN mit nur einer verdeckten Einheit zeigt, verdeutlicht dieses Problem.



Der Term  $\frac{\partial h^{(t)}}{\partial h^{(k)}}$  besteht aus  $t - k$  Faktoren; die  $(t - k)$ -fache Multiplikation mit der Gewichtung  $w$  führt also zu einem Faktor  $w^{t-k}$ , der sehr klein wird, wenn  $|w| < 1$  und  $t - k$  groß ist. Ist hingegen die Gewichtung der rekurrenten Kante  $|w| > 1$ , dann wird  $w^{t-k}$  sehr groß, wenn auch  $t - k$  groß ist. Beachten Sie hier, das sich  $t - k$  auf weitreichende Abhängigkeiten bezieht.

Das Verschwinden oder das explosionsartige Wachsen des Gradienten kann natürlich auf naive Weise verhindert werden, wenn man sicherstellt, dass  $|w| = 1$  ist. Wenn Sie daran interessiert sind und mehr über dieses Thema erfahren möchten, empfiehlt sich die Lektüre des Artikels *On the difficulty of training recurrent neural networks* von R. Pascanu, T. Mikolov und Y. Bengio (2012, <https://arxiv.org/pdf/1211.5063.pdf>).

In der Praxis bieten sich zwei Lösungen an:

- Truncated Backpropagation-Through-Time (TBPTT)
- Long Short-Term Memory (LSTM)

Beim TBPTT-Verfahren wird der Gradient oberhalb eines Grenzwerts abgeschnitten. Dadurch wird zwar das explosionsartige Anwachsen verhindert, allerdings wird auch die Anzahl der Schritte begrenzt, in denen der Gradient zur ordnungsgemäßen Aktualisierung der Gewichtungen verwendet werden kann.

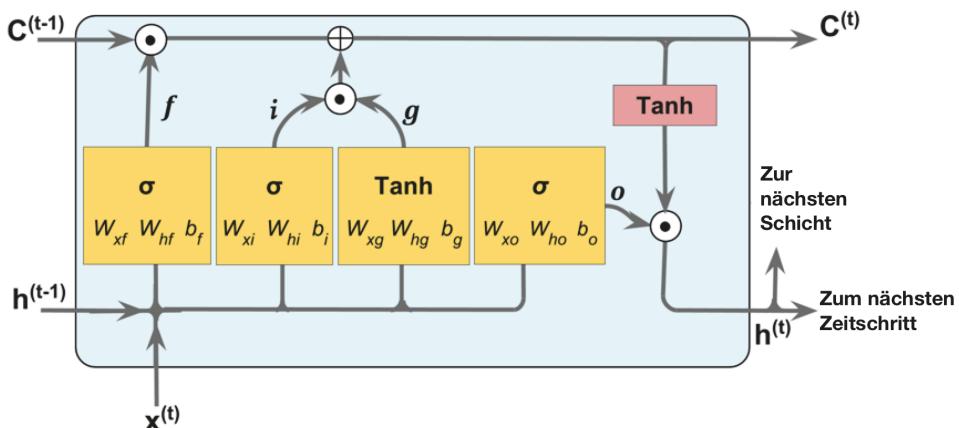
Das 1997 von Hochreiter und Schmidhuber entwickelte LSTM-Verfahren erwies sich bei der Modellierung umfangreicher Sequenzen bei der Lösung des Problems

des verschwindenden Gradienten als erfolgreicher. Dieses Verfahren sehen wir uns genauer an.

### 16.2.4 LSTM-Einheiten

LSTMs wurden ursprünglich eingeführt, um das Problem des verschwindenden Gradienten zu lösen (S. Hochreiter und J. Schmidhuber, *Long Short-Term Memory*, Neural Computation, 9(8): 1735-1780, 1997). Die Bausteine des LSTMs sind sogenannte *Speicherzellen*, die im Wesentlichen eine verdeckte Schicht darstellen.

Jede Speicherzelle besitzt eine rekurrente Kante mit der gewünschten Gewichtung  $w = 1$ , um die eben erörterten Probleme des verschwindenden oder des explosionsartig wachsenden Gradienten zu umgehen. Die diesen rekurrenten Kanten zugeordneten Werte werden als *Zellzustand* bezeichnet. Die folgende Abbildung zeigt die Struktur einer heutigen LSTM-Zelle:



Um den Zellzustand des aktuellen Zeitschritts  $C^{(t)}$  zu erhalten, wird auf den Zellzustand des vorhergehenden Zeitschritts  $C^{(t-1)}$  zugegriffen, ohne ihn direkt mit irgendeinem Gewichtungsfaktor zu multiplizieren.

Der Informationsfluss in der Speicherzelle wird durch folgende Berechnungen gesteuert: In der Abbildung kennzeichnet  $\odot$  das *elementweise Produkt* (durch elementweise Multiplikation) und  $\oplus$  die *elementweise Summierung* (durch elementweise Addition).  $x^{(t)}$  steht für die Eingabedaten zum Zeitpunkt  $t$  und  $h^{(t-1)}$  für die verdeckten Einheiten zum Zeitpunkt  $t - 1$ .

Die vier Kästen sind mit Aktivierungsfunktionen (der Sigmoidfunktion ( $\sigma$ ) oder dem Tangens hyperbolicus) und einem Satz von Gewichtungen gekennzeichnet. Diese Kästen wenden durch das Ausführen einer Matrix-Vektor-Multiplikation eine Linearkombination auf ihre Eingabe an. Diese Berechnungseinheiten mit sigmoiden Aktivierungsfunktionen, deren Ausgaben durch  $\odot$  verknüpft werden, bezeichnet man als *Gates*.

In einer LSTM-Zelle gibt es drei verschiedene Arten von Gates, die als Lösch-Gate, Eingabe-Gate und Ausgabe-Gate bezeichnet werden:

- Das Lösch-Gate ( $f_t$ , engl. *forget gate*) ermöglicht es der Speicherzelle, den Zellzustand zurückzusetzen, ohne unbegrenzt zu wachsen. Tatsächlich entscheidet das Lösch-Gate, welche Informationen durchgelassen und welche unterdrückt werden.  $f_t$  wird folgendermaßen berechnet:  $f_t = \sigma(W_{xf}x^{(t)} + W_{hf}h^{(t-1)} + b_f)$ . In der ursprünglichen LSTM-Zelle gab es noch kein Lösch-Gate – es wurde erst einige Jahre später zur Verbesserung des ursprünglichen Modells hinzugefügt (F. Gers, J. Schmidhuber und F. Cummins, *Learning to Forget: Continual Prediction with LSTM*, Neural Computation 12, Seiten 2451–2471, 2000).
- Das Eingabe-Gate ( $i_t$ ) und der Eingabeknoten ( $g_t$ ) sind für die Aktualisierung des Zellzustands verantwortlich. Sie werden wie folgt berechnet:

$$i_t = \sigma(W_{xi}x^{(t)} + W_{hi}h^{(t-1)} + b_i) \quad g_t = \tanh(W_{xg}x^{(t)} + W_{hg}h^{(t-1)} + b_g)$$

Der Zellzustand zum Zeitpunkt  $t$  wird so berechnet:

$$C^{(t)} = (C^{(t-1)} \odot f_t) \oplus (i_t \odot g_t)$$

- Das Ausgabe-Gate ( $o_t$ ) legt fest, wie die Werte der verdeckten Einheiten aktualisiert werden:

$$o_t = \sigma(W_{xo}x^{(t)} + W_{ho}h^{(t-1)} + b_o)$$

Damit ergibt sich für die verdeckten Einheiten des aktuellen Zeitschriffts:

$$h^{(t)} = o_t \odot \tanh(C^{(t)})$$

Der Aufbau einer LSTM-Zelle und die damit verbundenen Berechnungen erscheinen doch sehr kompliziert. Die gute Nachricht lautet jedoch, dass all dies in TensorFlow bereits in Form von Wrapper-Funktionen implementiert ist, die es ermöglichen, LSTM-Zellen auf einfache Weise zu definieren. Später in diesem Kapitel werden wir beim Einsatz von TensorFlow eine Anwendung von LSTMs in Aktion sehen.

### Hinweis

Die in diesem Abschnitt vorgestellten LSTMs stellen einen grundlegenden Ansatz für die Modellierung weitreichender Abhängigkeiten in Sequenzen dar. Es ist an dieser Stelle wichtig zu erwähnen, dass in der einschlägigen Literatur eine Vielzahl verschiedenen LSTM-Varianten beschrieben ist (Rafal Jozefowicz, Wojciech Zaremba und Ilya Sutskever, *An Empirical Exploration of Recurrent Network Architectures*, Proceedings of ICML, Seiten 2342–2350, 2015).

Ebenfalls erwähnenswert ist ein neuerer Ansatz namens *Gated Recurrent Unit* (GRU), der 2014 vorgeschlagen wurde. Die Architektur von GRUs ist einfacher als die von LSTMs, daher rechnen sie effizienter, obwohl die Leistung bei bestimmten Aufgaben, wie z.B. der Modellierung mehrstimmiger Musik, mit der von LSTMs vergleichbar ist. Mehr zu diesem Thema finden Sie in dem Artikel *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling* von Junyoung Chung et al. (2014, <https://arxiv.org/pdf/1412.3555v1.pdf>).

## 16.3 Implementierung eines mehrschichtigen RNNs zur Sequenzmodellierung mit TensorFlow

Nun haben Sie die RNNs zugrunde liegende Theorie kennengelernt und sind für den eher praktisch orientierten Teil gerüstet, der Implementierung eines RNNs mit TensorFlow. In den folgenden Abschnitten werden wir zwei gängige Aufgaben mit RNNs in Angriff nehmen:

1. Stimmungsanalyse
2. Sprachmodellierung

Diese beiden Projekte, die auf den folgenden Seiten erstellt werden, sind zwar überaus faszinierend, aber auch ziemlich komplex. Anstatt die vollständige Lösung zu präsentieren, werden wir die Implementierung schrittweise durchführen und den jeweiligen Code ausführlich erläutern. Wenn Sie sich ein Gesamtbild machen möchten und den vollständigen Code sehen wollen, bevor Sie sich eingehender damit befassen, können Sie das tun. Sie finden ihn unter <https://github.com/rasbt/python-machine-learning-book-2nd-edition/blob/master/code/ch16/ch16.ipynb>.

Bevor wir in diesem Kapitel mit dem Programmieren loslegen, zuerst noch ein wichtiger Hinweis. Wir nutzen eine sehr neue Version von TensorFlow (1.3.0, Stand August 2017) und verwenden Code aus dem **contrib**-Submodul von TensorFlows Python-API. Die dazugehörigen Funktionen und Klassen sowie die in diesem Kapitel genannten Verweise auf deren Dokumentation können sich in zukünftigen TensorFlow-Versionen ändern oder werden möglicherweise dem Submodul **tf.nn** hinzugefügt. Es ist daher ratsam, die Dokumentation der TensorFlow-API im Auge zu behalten, um von den letzten Änderungen zu erfahren, insbesondere dann, wenn Sie bei der Verwendung des in diesem Kapitel beschriebenen **tf.contrib**-Codes auf Schwierigkeiten stoßen.

## 16.4 Projekt 1: Analyse der Stimmungslage in der IMDb-Filmbewertungsdatenbank mit mehrschichtigen RNNs

Aus Kapitel 8 wissen Sie, dass sich die Stimmungsanalyse damit befasst, die in einem Satz oder in einem Textdokument geäußerte Meinung als positiv oder negativ zu beurteilen. Im Folgenden werden wir ein mehrschichtiges RNN zur Stimmungsanalyse mit einer n-zu-1-Architektur implementieren.

Zunächst implementieren wir ein n-zu-n-RNN für eine Anwendung als Sprachmodell. Die hier zur Einführung der Grundlagen von RNNs verwendeten Beispiele sind bewusst einfach gewählt worden, aber es gibt eine Vielzahl interessanter Anwendungen für die Sprachmodellierung, wie z.B. die Entwicklung eines Chatbots: Dem Computer wird die Fähigkeit verliehen, unmittelbar mit einem Menschen zu sprechen und zu interagieren.

### 16.4.1 Datenaufbereitung

Bei der Vorverarbeitung der Daten in Kapitel 8 haben wir eine bereinigte Datenmenge namens `movie_data.csv` erstellt, die jetzt wieder zum Einsatz kommt. Zunächst einmal importieren wir die erforderlichen Module und lesen die Dateien in einen Pandas-DataFrame ein:

```
>>> import pyprind
>>> import pandas as pd
>>> from string import punctuation
>>> import re
>>> import numpy as np
>>>
>>> df = pd.read_csv('movie_data.csv', encoding='utf-8')
```

Wie Sie wissen, besitzt der DataFrame `df` zwei Spalten namens '`review`' und '`sentiment`', die den Text der Filmbewertung und die Klassenbezeichnung 0 oder 1 enthalten. Die Textkomponenten dieser Filmbewertungen stellen Sequenzen aus Wörtern dar, deshalb soll das RNN-Modell die Wörter einer Sequenz verarbeiten und zum Schluss die gesamte Sequenz als zur Klasse 0 oder 1 zugehörig klassifizieren.

Damit die Daten als Eingabe für ein neuronales Netz dienen können, müssen wir sie in numerische Werte konvertieren. Zu diesem Zweck müssen wir zunächst die verschiedenen Wörter in der gesamten Datenmenge finden. Dafür könnten wir Python-Mengen verwenden. Wir haben allerdings festgestellt, dass Mengen für das Suchen nach Wörtern in einer so großen Datenmenge alles andere als effizient ist.

Hier verwendet man besser einen Zähler, wie den Counter aus dem collections-Paket (<https://docs.python.org/3/library/collections.html#collections.Counter>).

Im folgenden Code definieren wir ein counts-Objekt der Counter-Klasse, das die Vorkommen der verschiedenen Wörter im Text zählt. Bei dieser speziellen Anwendung sind wir (anders als beim Bag-of-words-Modell) ausschließlich an einer Menge der verschiedenen Wörter interessiert, nicht an deren Anzahl, die sozusagen als Nebenprodukt anfällt.

Anschließend erstellen wir ein Dictionary, das den verschiedenen Wörtern der Datenmenge eindeutige ganze Zahlen zuordnet. Dieses Dictionary wird word\_to\_int genannt und dient dazu, den gesamten Text einer Bewertung in eine Zahlenliste zu konvertieren. Die einzelnen Wörter sind nach ihrer Anzahl sortiert, aber die Reihenfolge spielt für das Endergebnis keine Rolle. Die Konvertierung des Texts in eine Zahlenliste erledigt der folgende Code:

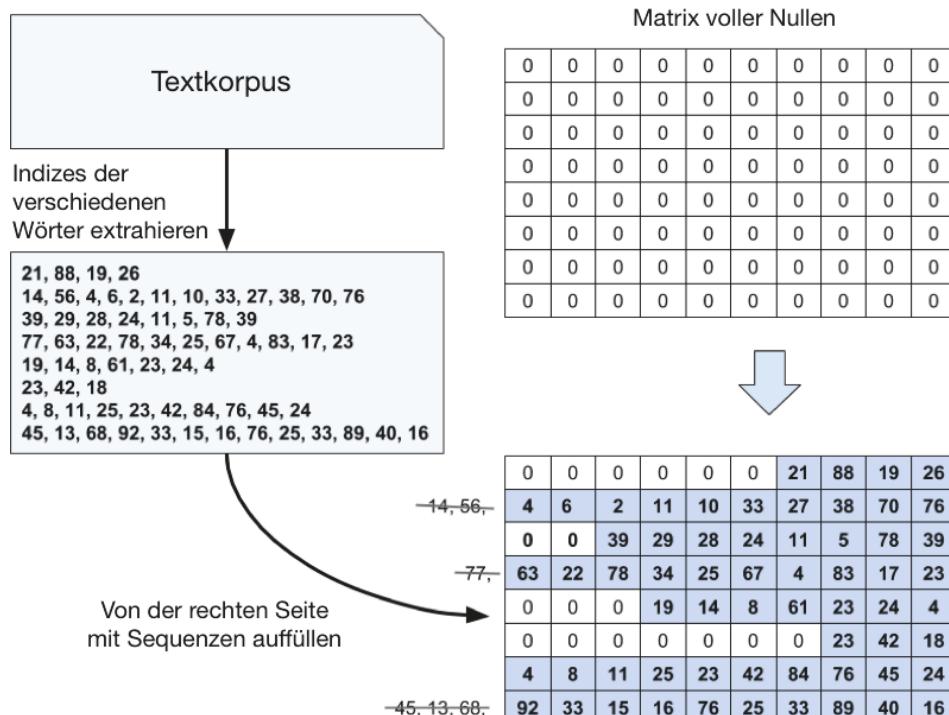
```
>>> ## Datenaufbereitung:
>>> ## In Wörter aufteilen und Vorkommen zählen
>>>
>>> from collections import Counter
>>> counts = Counter()
>>> pbar = pyprind.ProgBar(len(df['review']), \
...                         title='Vorkommen der Wörter zählen')
>>> for i, review in enumerate(df['review']):
...     text = ''.join([c if c not in punctuation else ' \
...                   '+c+' ' for c in review]).lower()
...     df.loc[i, 'review'] = text
...     pbar.update()
...     counts.update(text.split())
>>>
>>> ## Zuordnung erzeugen
>>> ## Den verschiedenen Wörtern eindeutige Zahlen zuordnen
>>> word_counts = sorted(counts, key=counts.get, \
...                         reverse=True)
>>> print(word_counts[:5])
>>> word_to_int = {word: ii for ii, word in \
...                         enumerate(word_counts, 1)}
>>>
>>> mapped_reviews = []
>>> pbar = pyprind.ProgBar(len(df['review']), \
...                         title='Bewertungen Zahlen zuordnen')
>>> for review in df['review']:
...     mapped_reviews.append([word_to_int[word] \
...                           for word in review.split()])
...     pbar.update()
```

Nun haben wir die Wortsequenzen in Sequenzen von Zahlen konvertiert. Es gibt aber noch ein weiteres Problem zu lösen: Die Sequenzen sind unterschiedlich lang. Um mit dem RNN kompatible Eingabedaten zu erhalten, müssen wir dafür sorgen, dass alle Sequenzen von gleicher Länge sind.

Zu diesem Zweck definieren wir einen Parameter namens `sequence_length` und weisen ihm den Wert 200 zu. Sequenzen mit weniger als 200 Wörtern werden von links mit Nullen aufgefüllt. Umgekehrt werden Sequenzen mit mehr als 200 Wörtern abgeschnitten, sodass nur die verbleibenden 200 Wörter verwendet werden. Wir können die Datenaufbereitung in zwei Schritten erledigen:

1. Erstellen einer Matrix voller Nullen, in der jede Zeile einer Sequenz der Größe 200 entspricht
2. Befüllen der Matrix von der rechten Seite mit den Wortsequenzen. Wenn eine Sequenz aus 150 Wörtern besteht, enthalten also die ersten 50 Elemente der dazugehörigen Zeile Nullen.

Diese beiden Schritte sind in der folgenden Abbildung dargestellt. Als Beispiel dienen acht Sequenzen der Größen 4, 12, 8, 11, 7, 3, 10 und 13.



Beachten Sie, dass es sich bei `sequence_length` um einen Hyperparameter handelt, der zum Optimieren der Leistung abgestimmt werden kann. Aus Platzgrün-

den verzichten wir hier darauf, diesen Hyperparameter zu optimieren, aber probieren Sie ruhig ein paar verschiedene Werte für `sequence_length` aus, wie z.B. 50, 100, 200, 250 und 300.

Der folgende Code implementiert diese Schritte zum Erstellen von Sequenzen gleicher Länge:

```
>>> ## Sequenzen gleicher Länge erzeugen
>>> ## Wenn sequence_length < 200, von links mit Nullen
>>> ## auffüllen, wenn sequence_length > 200, die letzten
>>> ## 200 Elemente verwenden.
>>>
>>> sequence_length = 200 ## (T in den RNN-Gleichungen)
>>> sequences = np.zeros((len(mapped_reviews),
...                         sequence_length),
...                         dtype=int)
>>>
>>> for i, row in enumerate(mapped_reviews):
...     review_arr = np.array(row)
...     sequences[i, -len(row):] = \
...         review_arr[-sequence_length:]
```

Nach der Vorverarbeitung der Daten können wir die Datenmenge in Trainings- und Testdatenmenge aufteilen. Da die Daten bereits durchmischt sind, können wir einfach die erste Hälfte für das Training und die zweite Hälfte zum Testen verwenden:

```
>>> X_train = sequences[:25000,:]
>>> y_train = df.loc[:25000, 'sentiment'].values
>>> X_test = sequences[25000:,:]
>>> y_test = df.loc[25000:, 'sentiment'].values
```

Wenn wir eine Kreuzvalidierung durchführen möchten, können wir die zweite Hälfte der Daten weiter aufteilen, um kleinere Test- und Validierungsdatenmengen für die Optimierung der Hyperparameter zu erzeugen.

Nun definieren wir noch eine Hilfsfunktion, die eine Datenmenge (bei der es sich um eine Trainings- oder Testdatenmenge handeln könnte) in Teilmengen aufteilt und einen Generator zum Durchlaufen dieser Teilmengen (den sogenannten *Mini-Batches*) zurückgibt:

```
>>> np.random.seed(123) # Reproduzierbarkeit
>>> ## Funktion zum Erzeugen von Mini-Batches definieren:
>>> def create_batch_generator(x, y=None, batch_size=64):
...     n_batches = len(x)//batch_size
```

```

...
...     x = x[:n_batches*batch_size]
...     if y is not None:
...         y = y[:n_batches*batch_size]
...     for ii in range(0, len(x), batch_size):
...         if y is not None:
...             yield x[ii:ii+batch_size], y[ii:ii+batch_size]
...         else:
...             yield x[ii:ii+batch_size]

```

Die Verwendung von Generatoren wie in diesem Code ist ein äußerst nützliches Verfahren, um mit knappem Arbeitsspeicher zurechtzukommen. Hierbei handelt es sich um den empfohlenen Ansatz zur Aufteilung von Datenmengen zum Trainieren eines neuronalen Netzes, anstatt alle Datenaufteilungen vorab vorzunehmen und sie beim Training im Arbeitsspeicher zu behalten.

## 16.4.2 Einbettung

Während der Datenaufbereitung im letzten Abschnitt haben wir Sequenzen gleicher Länge erstellt. Die Elemente dieser Sequenzen sind ganze Zahlen, die den Indizes der verschiedenen Wörter entsprechen.

Die Wortindizes können auf verschiedene Weise in Eingabemerkmale konvertiert werden. Man könnte die One-hot-Codierung anwenden, um die Indizes in aus Nullen und Einsen bestehende Vektoren zu konvertieren. In diesem Fall wird jedem Wort ein Vektor zugeordnet, dessen Größe der Anzahl unterschiedlicher Wörter der gesamten Datenmenge entspricht. In Anbetracht der Tatsache, dass diese Zahl (die Größe des Vokabulars) in der Größenordnung von 20.000 liegt (das entspricht der Zahl der Eingabemerkmale), würde ein mit diesen Merkmalen trainiertes Modell vermutlich unter dem Fluch der Dimensionalität leiden. Darüber hinaus sind diese Merkmalsvektoren extrem dünnbesetzt, da bis auf eins alle Elemente null sind.

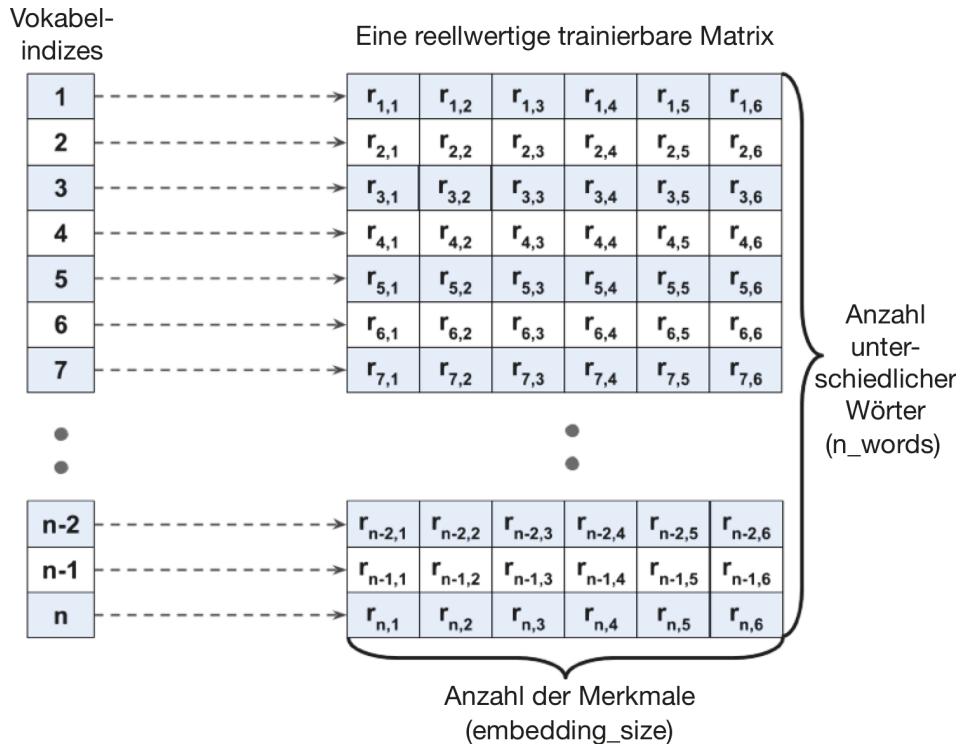
Jedem Wort einen Vektor fester Größe mit reellwertigen Elementen (nicht unbedingt Integer-Zahlen) zuzuordnen, ist ein eleganterer Ansatz. Im Gegensatz zu One-hot-codierten Vektoren können Vektoren fester Größe unendlich viele reelle Zahlen repräsentieren. (Theoretisch können wir jedem beliebigen Intervall wie z.B.  $[-1,1]$  unendlich viele reelle Zahlen entnehmen.)

Diese Idee liegt der sogenannten *Einbettung* zugrunde, ein Verfahren zum Erkennen von Merkmalen, das wir zum automatischen Erlernen der auffälligsten Merkmale zur Repräsentierung der Wörter in der Datenmenge einsetzen können. Bei einer gegebenen Anzahl unterschiedlicher Wörter können wir für die Größe des Einbettungsvektors einen Wert wählen, der sehr viel kleiner als diese Anzahl ist (Größe des Einbettungsvektors << Anzahl unterschiedlicher Wörter), um das gesamte Vokabular als Eingabemerkmale zu repräsentieren.

Die Einbettung weist gegenüber der One-hot-Codierung folgende Vorteile auf:

- Die Dimensionsreduktion des Merkmalsraums verringert die Auswirkungen des Fluchs der Dimensionalität.
- Die Extraktion auffälliger Merkmale, denn die Einbettungsschicht eines neuronalen Netzes kann trainiert werden.

Die folgende Abbildung zeigt die Funktionsweise der Einbettung: Die Vokabelindizes werden den Elementen einer trainierbaren Einbettungsmatrix zugeordnet.



TensorFlow implementiert eine effiziente Funktion namens `tf.nn.embedding_lookup`, die jeder Zahl, die einem bestimmten Wort entspricht, eine Zeile dieser trainierbaren Matrix zuordnet. Beispielsweise ist 1 der ersten Zeile zugeordnet, 2 der zweiten und so weiter. Liegt nun eine Zahlensequenz wie `<0, 5, 3, 4, 19, 2 ...>` vor, können wir die den Elementen der Sequenz entsprechende Zeile nachschlagen.

Sehen wir uns doch einmal an, wie man in der Praxis eine solche Einbettungsschicht erstellt. `tf_x` ist die Eingabeschicht, der die Vokabelindizes als Datentyp `tf.int32` übergeben werden. Das Erstellen der Einbettungsschicht erfolgt in zwei Schritten:

- Zunächst einmal erstellen wir eine Matrix der Größe `[n_words × embedding_size]` als Tensorvariable, die wir `embedding` nennen und mit zufälligen Fließkommazahlen aus dem Intervall  $[-1, 1]$  initialisieren:

```
embedding = tf.Variable(
    tf.random_uniform(
        shape=(n_words, embedding_size),
        minval=-1, maxval=1)
)
```

- Anschließend verwenden wir die Funktion `tf.nn.embedding_lookup`, um die zu den Elementen von `tf_x` zugehörigen Zeilen der Einbettungsmatrix zu finden:

```
embed_x = tf.nn.embedding_lookup(embedding, tf_x)
```

### Hinweis

Vielleicht ist Ihnen aufgefallen, dass die Funktion `tf.nn.embedding_lookup` zum Erstellen einer Einbettungsschicht zwei Argumente benötigt: den Einbettungstensor und die IDs.

An die Funktion `tf.nn.embedding_lookup` können einige optionale Argumente übergeben werden, die es Ihnen erlauben, das Verhalten der Einbettungsschicht zu ändern, beispielsweise eine L2-Normierung anzuwenden. Weitere Informationen über diese Funktion finden Sie in der offiziellen Dokumentation unter [https://www.tensorflow.org/api\\_docs/python/tf/nn/embedding\\_lookup](https://www.tensorflow.org/api_docs/python/tf/nn/embedding_lookup).

### 16.4.3 Erstellen eines RNN-Modells

Jetzt ist alles vorbereitet, um ein RNN-Modell zu erstellen. Wir implementieren die Klasse `SentimentRNN` mit den folgenden Methoden:

- Einen Konstruktor, der alle Modellparameter einstellt, den Berechnungsgraphen erzeugt und die Methode `self.build` aufruft, um das mehrschichtige RNN-Modell anzulegen.
- Eine `build`-Methode, die drei Platzhalter für Eingabedaten, Eingabeklassenbezeichnungen und die Wahrscheinlichkeit des Verbleibens für die Dropout-Konfiguration der verdeckten Schicht deklariert.
- Eine `train`-Methode, die eine TensorFlow-Sitzung zum Starten des Berechnungsgraphen eröffnet, die Mini-Batches der Daten durcharbeitet und eine festgelegte Anzahl von Epochen läuft, um die im Graphen definierte Straffunktion zu minimieren. Darüber hinaus speichert die Methode nach jeweils 10 Epochen einen Fixpunkt des Modells.

- Eine `predict`-Methode, die eine neue Sitzung eröffnet, den während des Trainings zuletzt gespeicherten Fixpunkt wiederherstellt und die Vorhersage für die Testdaten trifft.

Der Code für die Implementierung dieser Klasse und ihrer Methoden wird in den folgenden Abschnitten schrittweise vorgestellt und erläutert.

#### 16.4.4 Der Konstruktor der SentimentRNN-Klasse

Als Erstes erstellen wir wie folgt den Konstruktor der SentimentRNN-Klasse:

```
import tensorflow as tf

class SentimentRNN(object):
    def __init__(self, n_words, seq_len=200,
                 lstm_size=256, num_layers=1, batch_size=64,
                 learning_rate=0.0001, embed_size=200):
        self.n_words = n_words
        self.seq_len = seq_len
        self.lstm_size = lstm_size # Zahl verdeckter Einheiten
        self.num_layers = num_layers
        self.batch_size = batch_size
        self.learning_rate = learning_rate
        self.embed_size = embed_size

        self.g = tf.Graph()
        with self.g.as_default():
            tf.set_random_seed(123)
            self.build()
            self.saver = tf.train.Saver()
            self.init_op = tf.global_variables_initializer()
```

Der Parameter `n_words` muss der Anzahl unterschiedlicher Wörter entsprechen (zuzüglich 1, denn wir verwenden 0, um Sequenzen aufzufüllen, die kürzer als 200 sind). Er wird beim Erstellen der Einbettungsschicht zusammen mit dem Hyperparameter `embed_size` benutzt. Der Variablen `seq_len` muss ein Wert entsprechend der Länge der bei der Datenaufbereitung erstellten Sequenzen zugewiesen werden. Beachten Sie außerdem den zusätzlichen Hyperparameter `lstm_size`, der die Anzahl der verdeckten Einheiten in den RNN-Schichten festlegt.

#### 16.4.5 Die build-Methode

Nun kommen wir zur `build`-Methode der `SentimentRNN`-Klasse. Sie ist die längste und wichtigste der Methoden, daher werden wir sie sehr detailliert unter die Lupe nehmen. Sehen Sie sich den Code zuerst als Ganzes an. Anschließend werden wir die Hauptbestandteile der Reihe nach analysieren.

```

def build(self):
    ## Platzhalter definieren
    tf_x = tf.placeholder(tf.int32,
                          shape=(self.batch_size, self.seq_len),
                          name='tf_x')
    tf_y = tf.placeholder(tf.float32,
                          shape=(self.batch_size), name='tf_y')
    tf_keepprob = tf.placeholder(tf.float32,
                                name='tf_keepprob')

    ## Einbettungsschicht erstellen
    embedding = tf.Variable(
        tf.random_uniform(
            (self.n_words, self.embed_size),
            minval=-1, maxval=1, name='embedding'))
    embed_x = tf.nn.embedding_lookup(embedding, tf_x,
                                    name='embeded_x')

    ## LSTM-Zelle und Stack zusammen definieren
    cells = tf.contrib.rnn.MultiRNNCell(
        [tf.contrib.rnn.DropoutWrapper(
            tf.contrib.rnn.BasicLSTMCell(self.lstm_size),
            output_keep_prob=tf_keepprob)
         for i in range(self.num_layers)])

    ## Anfangszustand definieren:
    self.initial_state = cells.zero_state(
        self.batch_size, tf.float32)
    print(' <> Anfangszustand >> ', self.initial_state)

    lstm_outputs, self.final_state = tf.nn.dynamic_rnn(
        cells, embed_x, initial_state=self.initial_state)

    ## Hinweis: lstm_outputs shape:
    ## [batch_size, max_time, cells.output_size]
    print('\n <> lstm_output >> ', lstm_outputs)
    print('\n <> Endzustand >> ', self.final_state)

    logits = tf.layers.dense(
        inputs=lstm_outputs[:, -1],
        units=1, activation=None,
        name='logits')

    logits = tf.squeeze(logits, name='logits_squeezed')
    print ('\n <> Logits >> ', logits)

```

```

y_proba = tf.nn.sigmoid(logits, name='probabilities')
predictions = {
    'probabilities': y_proba,
    'labels' : tf.cast(tf.round(y_proba), tf.int32,
                      name='labels')
}
print('\n << Vorhersagen >> ', predictions)

## Straffunktion definieren
cost = tf.reduce_mean(
    tf.nn.sigmoid_cross_entropy_with_logits(
        labels=tf_y, logits=logits), name='cost')

## Optimierer definieren
optimizer = tf.train.AdamOptimizer(self.learning_rate)
train_op = optimizer.minimize(cost, name='train_op')

```

In der `build`-Methode erstellen wir zuallererst drei Platzhalter, nämlich `tf_x`, `tf_y` und `tf_keepprob`, die wir für die Zuführung der Eingabedaten benötigen. Anschließend fügen wir die Einbettungsschicht hinzu, die wie vorhin erörtert die Repräsentierung `embed_x` erzeugt.

Als Nächstes erzeugen wir in der `build`-Methode das RNN-Netz mit LSTM-Zellen. Dazu sind diese drei Schritte nötig:

1. Definition der mehrschichtigen RNN-Zellen
2. Definition der Anfangszustände dieser Zellen
3. Erzeugen des durch die RNN-Zellen und ihre Anfangszustände gegebenen RNNs

Diese drei Schritte werden wir in den nächsten drei Abschnitten detailliert aufschlüsseln, um ausführlich zu untersuchen, wie das RNN-Netz in der `build`-Methode erzeugt wird.

## Schritt 1: Definition der Zellen eines mehrschichtigen RNNs

Der erste Schritt beim Erstellen des RNN in der `build`-Methode ist die Definition der mehrschichtigen RNN-Zellen.

Erfreulicherweise stellt TensorFlow eine äußerst praktische Wrapper-Klasse zum Definieren von LSTM-Zellen bereit, nämlich die Klasse `BasicLSTMCell`, die zusammen mit der Wrapper-Klasse `MultiRNNCell` zum Aufbau eines mehrschichtigen RNN genutzt werden kann. Das Zusammenfügen von RNN-Zellen mit Dropout besteht aus drei verschachtelten Schritten, die sich folgendermaßen beschreiben lassen:

1. Erstellen Sie mit `tf.contrib.rnn.BasicLSTMCell` die RNN-Zellen.
2. Wenden Sie mithilfe von `tf.contrib.rnn.DropoutWrapper` das Dropout-Verfahren auf die RNN-Zellen an.
3. Erstellen Sie entsprechend der gewünschten Anzahl von RNN-Schichten eine Liste solcher Zellen und übergeben Sie diese an `tf.contrib.rnn.MultirNNCell`.

Im Code der `build`-Methode werden diese Listen mit Pythons Listenabstraktionen erstellt. Bei einer einzelnen Schicht enthält diese Liste nur eine Zelle.

### Hinweis

Weitere Informationen über diese Funktionen sind unter den folgenden Links verfügbar:

`tf.contrib.rnn.BasicLSTMCell`:

[https://www.tensorflow.org/api\\_docs/python/tf/contrib/rnn/BasicLSTMCell](https://www.tensorflow.org/api_docs/python/tf/contrib/rnn/BasicLSTMCell)

`tf.contrib.rnn.DropoutWrapper`:

[https://www.tensorflow.org/api\\_docs/python/tf/contrib/rnn/DropoutWrapper](https://www.tensorflow.org/api_docs/python/tf/contrib/rnn/DropoutWrapper)

`tf.contrib.rnn.MultirNNCell`:

[https://www.tensorflow.org/api\\_docs/python/tf/contrib/rnn/MultirNNCell](https://www.tensorflow.org/api_docs/python/tf/contrib/rnn/MultirNNCell)

## Schritt 2: Definition der Anfangszustände der RNN-Zellen

Der zweite Schritt zum Erstellen des RNN in der `build`-Methode ist die Definition der Anfangszustände der RNN-Zellen.

Wie Sie aus der Beschreibung des Aufbaus von LSTM-Zellen wissen, gibt es drei verschiedene Arten von Eingaben für eine LSTM-Zelle: die Eingabedaten  $x^{(t)}$ , die Aktivierungen der verdeckten Einheiten vom vorhergehenden Zeitschritt  $h^{(t-1)}$  und den Zellzustand des vorherigen Zeitschritts  $C^{(t-1)}$ .

In der Implementierung der `build`-Methode ist  $x^{(t)}$  also der eingebettete Daten-tensor `embed_x`. Bei der Auswertung der Zellen `cells` müssen wir den vorherigen Zustand der Zellen angeben. Wenn die Verarbeitung einer neuen Eingabesequenz beginnt, initialisieren wir die Zellen mit einem Nullzustand. Anschließend wird der aktualisierte Zustand der Zellen nach jedem Zeitschritt gespeichert, damit er beim nachfolgenden verwendet werden kann.

Nachdem das mehrschichtige RNN-Objekt (`cells` in unserer Implementierung) definiert ist, legen wir mit der `cells.zero_state`-Methode den Anfangszustand dieses Objekts fest.

### Schritt 3: Erstellen des RNNs unter Verwendung der RNN-Zellen und ihrer Zustände

Der dritte Schritt zum Erstellen des RNN in der `build`-Methode ist der Aufruf der Funktion `tf.nn.dynamic_rnn`, mit der alle Komponenten zusammengefügt werden.

Die `tf.nn.dynamic_rnn`-Funktion erstellt anhand der eingebetteten Daten, der RNN-Zellen und ihren Anfangszuständen entsprechend der aufgeschlüsselten Architektur der LSTM-Zellen eine Pipeline.

Die Funktion liefert ein Tupel zurück, das die Aktivierungen der RNN-Zellen (`outputs`) und ihren Endzustand (`state`) enthält. Die Ausgabe ist ein dreidimensionaler Tensor mit der Shape (`batch_size, num_steps, lstm_size`). Wir übergeben `outputs` an eine vollständig verknüpfte Schicht, um `logits` zu erhalten, und speichern den Endzustand, um ihn als Anfangszustand für den nächsten Mini-Batch der Daten verwenden zu können.

#### Hinweis

Weitere Informationen über die Funktion `tf.nn.dynamic_rnn` finden Sie in der offiziellen Dokumentation unter [https://www.tensorflow.org/api\\_docs/python/tf/nn/dynamic\\_rnn](https://www.tensorflow.org/api_docs/python/tf/nn/dynamic_rnn).

Nach der Einrichtung der RNN-Komponenten können am Ende der `build`-Methode, wie bei jedem anderen neuronalen Netz, die Straffunktion und die Optimierungsverfahren definiert werden.

#### 16.4.6 Die train-Methode

Die nächste Methode der `SentimentRNN`-Klasse heißt `train`. Sie ist den in Kapitel 14 und 15 erstellten `train`-Methoden ziemlich ähnlich, allerdings gibt es hier den zusätzlichen Tensor `state`, der in das neuronale Netz eingespeist wird.

Der folgende Code implementiert die `train`-Methode:

```
def train(self, X_train, y_train, num_epochs):
    with tf.Session(graph=self.g) as sess:
        sess.run(self.init_op)
        iteration = 1
        for epoch in range(num_epochs):
            state = sess.run(self.initial_state)

                for batch_x, batch_y in create_batch_generator(
                    X_train, y_train, self.batch_size):
```

```

        feed = {'tf_x:0': batch_x,
                 'tf_y:0': batch_y,
                 'tf_keepprob:0': 0.5,
                 self.initial_state : state}
        loss, _, state = sess.run(
            ['cost:0', 'train_op',
             self.final_state],
            feed_dict=feed)

        if iteration % 20 == 0:
            print("Epoche: %d/%d Iteration: %d "
                  "| Verlust Training: %.5f" % (
                      epoch + 1, num_epochs,
                      iteration, loss))
        iteration +=1
        if (epoch+1)%10 == 0:
            self.saver.save(sess,
                            "model/sentiment-%d.ckpt" % epoch)
    
```

In dieser Implementierung der `train`-Methode wird den RNN-Zellen am Anfang jeder Epoche als aktueller Zustand der Nullzustand zugewiesen. Zur Verarbeitung der Mini-Batches wird der aktuelle Zustand zusammen mit den Daten `batch_x` und den Klassenbezeichnungen `batch_y` eingespeist. Nach der Ausführung wird der aktuelle Zustand aktualisiert und damit zum Endzustand, den die Funktion `tf.nn.dynamic_rnn` zurückliefert. Dieser Zustand ist bei der Verarbeitung des nächsten Mini-Batch der Anfangszustand. Während der gesamten Epoche wird dieser Vorgang wiederholt und der aktuelle Zustand aktualisiert.

### 16.4.7 Die predict-Methode

Die letzte Methode der `SentimentRNN`-Klasse ist `predict`, die den aktuellen Zustand – ähnlich wie die `train`-Methode – laufend aktualisiert:

```

def predict(self, X_data, return_proba=False):
    preds = []
    with tf.Session(graph = self.g) as sess:
        self.saver.restore(
            sess, tf.train.latest_checkpoint('./model/'))
        test_state = sess.run(self.initial_state)
        for ii, batch_x in enumerate(
            create_batch_generator(
                X_data, None, batch_size=self.batch_size), 1):
            feed = {'tf_x:0' : batch_x,
                    'tf_keepprob:0' : 1.0,
                    self.initial_state : test_state}
            
```

```

        if return_proba:
            pred, test_state = sess.run(
                ['probabilities:0', self.final_state],
                feed_dict=feed)
        else:
            pred, test_state = sess.run(
                ['labels:0', self.final_state],
                feed_dict=feed)
        preds.append(pred)

    return np.concatenate(preds)

```

### 16.4.8 Instanziierung der SentimentRNN-Klasse

Wir haben nun die vier Teile der SentimentRNN-Klasse programmiert und untersucht: den Konstruktor sowie die `build`-, die `train`- und die `predict`-Methode.

Nun erzeugen wir eine Instanz der SentimentRNN-Klasse mit folgenden Parametern:

```

>>> n_words = max(list(word_to_int.values())) + 1
>>>
>>> rnn = SentimentRNN(n_words=n_words,
...                      seq_len=sequence_length,
...                      embed_size=256,
...                      lstm_size=128,
...                      num_layers=1,
...                      batch_size=100,
...                      learning_rate=0.001)

```

Beachten Sie die Einstellung `num_layers=1` für eine einzelne RNN-Schicht. Die Implementierung ermöglicht aber auch die Erstellung mehrschichtiger RNNs, indem `num_layers` ein größerer Wert als 1 zugewiesen wird. Hier sollten wir die geringe Größe der Datenmenge berücksichtigen und bedenken, dass eine einzelne RNN-Schicht vermutlich besser mit unbekannten Daten zurechtkommt, weil es weniger wahrscheinlich ist, dass es zu einer Überanpassung an die Trainingsdaten kommt.

### 16.4.9 Training und Optimierung des RNN-Modells zur Stimmungsanalyse

Jetzt rufen wir die Funktion `rnn.train` auf, um das RNN-Modell zu trainieren. Der folgende Code trainiert das Modell 40 Epochen lang mit den Eingaben `X_train` und den in `y_train` gespeicherten dazugehörigen Klassenbezeichnungen:

```
>>> rnn.train(X_train, y_train, num_epochs=40)
Epoch: 1/40 Iteration: 20 | Verlust Training: 0.70637
Epoch: 1/40 Iteration: 40 | Verlust Training: 0.60539
Epoch: 1/40 Iteration: 60 | Verlust Training: 0.66977
Epoch: 1/40 Iteration: 80 | Verlust Training: 0.51997
...
...
```

Das trainierte Modell wird mit TensorFlows Fixpunktsystem gespeichert, das in Kapitel 14 erläutert wurde. Nun können wir das Modell zur Vorhersage der Klassenbezeichnung der Testdatenmenge folgendermaßen verwenden:

```
>>> preds = rnn.predict(X_test)
>>> y_true = y_test[:len(preds)]
>>> print('Test Acc.: %.3f' % (
...     np.sum(preds == y_true) / len(y_true)))
```

Es ergibt sich eine Korrektklassifizierungsrate von 86 Prozent. In Anbetracht der geringen Größe dieser Datenmenge ist dieses Ergebnis mit der in Kapitel 8 erzielten Korrektklassifizierungsrate für die Testdatenmenge vergleichbar.

Wir können das Verfahren weiter optimieren, indem wir die Hyperparameter des Modells, wie `lstm_size`, `seq_len` und `embed_size`, ändern, um eine bessere Verallgemeinerungsfähigkeit zu erzielen. Es empfiehlt sich allerdings, für die Hyperparameter-Abstimmung eine separate Validierungsdatenmenge zu erstellen und die Testdatenmenge nicht mehrmals zur Beurteilung einzusetzen, um zu verhindern, dass es durch die in Kapitel 6 beschriebenen Datenlecks der Testdatenmenge zu einem Bias kommt.

Falls Sie statt der Klassenbezeichnungen der Testdatenmenge lieber die Wahrscheinlichkeiten für die Klassenzugehörigkeit vorhersagen möchten, können Sie die Einstellung `return_proba=True` verwenden:

```
>>> proba = rnn.predict(X_test, return_proba=True)
```

So viel zum ersten RNN-Modell zur Stimmungsanalyse. Wir fahren mit der Erstellung eines RNN-Modells zur zeichenweisen Sprachmodellierung mit TensorFlow fort, einer weiteren verbreiteten Anwendung der Sequenzmodellierung.

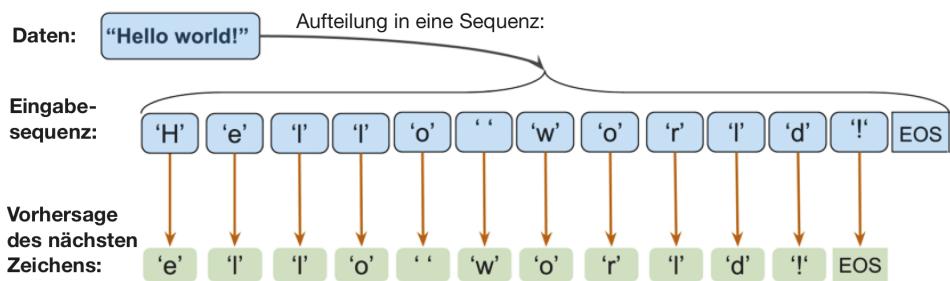
## 16.5 Projekt 2: Implementierung eines RNNs zur Sprachmodellierung durch Zeichen mit TensorFlow

Die Sprachmodellierung ist eine faszinierende Anwendung, die es Maschinen ermöglicht, Aufgaben zu erledigen, die von der menschlichen Sprache Gebrauch

machen, beispielsweise englische Sätze zu erzeugen. Zu den interessantesten Errungenschaften auf diesem Gebiet gehören die Arbeiten von Sutskever, Martens und Hinton (Ilya Sutskever, James Martens und Geoffrey E. Hinton, *Generating Text with Recurrent Neural Networks*, Proceedings of the 28th International Conference on Machine Learning (ICML-11), 2011, <https://pdfs.semanticscholar.org/93c2/0e38c85b69fc2d2eb314b3c1217913f7db11.pdf>).

Das Modell, das wir entwickeln werden, erhält als Eingabe ein Textdokument. Ziel ist es, dass das Modell neue Texte erzeugt, die dem eingegebenen Text vergleichbar sind. Als Eingabe kommt beispielsweise ein Buch oder ein Computerprogramm in einer bestimmten Programmiersprache infrage.

Bei der Sprachmodellierung durch Zeichen wird die Eingabe in eine Sequenz umgewandelt, die zeichenweise in das Modell eingespeist werden. Bei der Verarbeitung neuer Zeichen berücksichtigt das Modell die gespeicherten vorangegangenen Zeichen und versucht, das nächste Zeichen vorherzusagen. Die folgende Abbildung zeigt ein Beispiel für die zeichenweise Sprachmodellierung:



Wir teilen die Implementierung in drei einzelne Schritte auf: Datenaufbereitung, Erstellen des RNN-Modells und Entnahme von Zeichen zur Erzeugung neuer Texte bzw. zur Vorhersage des nächsten Zeichens.

Im letzten Abschnitt wurde das Problem des explosionsartig wachsenden Gradienten erwähnt. Bei dieser Anwendung bietet sich die Gelegenheit, Verfahren zur Beschneidung des Gradienten auszuprobieren, um dieses Problem zu umgehen.

### 16.5.1 Datenaufbereitung

In diesem Abschnitt bereinigen wir die Daten, um sie für die zeichenweise Sprachmodellierung verwenden zu können.

Die Eingabedaten finden Sie auf der Website des Gutenberg-Projekts, das Tausende kostenlose E-Books zur Verfügung stellt: <https://www.gutenberg.org/>. Für das Beispiel verwenden wir das Buch *The Tragedie of Hamlet* von William Shakespeare im Format reiner Text: <http://www.gutenberg.org/cache/epub/2265/pg2265.txt>.

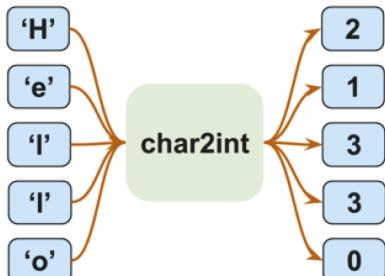
Der Link führt Sie direkt auf die Seite zum Herunterladen des Texts. Falls Sie macOS oder Linux verwenden, können Sie die Datei durch Eingabe des folgenden Befehls im Terminal herunterladen:

```
curl http://www.gutenberg.org/cache/epub/2265/pg2265.txt >
pg2265.txt
```

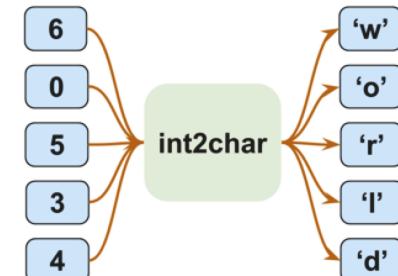
Für den Fall, dass die Datei nicht mehr verfügbar sein sollte, ist sie auch im Verzeichnis mit dem Code für dieses Kapitel enthalten: <https://github.com/rasbt/python-machine-learning-book-2nd-edition>.

Wenn die Daten vorliegen, lesen wir sie in einer Python-Sitzung als reinen Text ein. Im folgenden Code repräsentiert die Python-Variable `chars` die Menge der im Text vorhandenen Zeichen, in der jedes Zeichen nur einmalig enthalten ist. Wir erstellen ein Dictionary namens `char2int`, das jedem Zeichen eine Integer-Zahl zuordnet, sowie ein Dictionary `int2char` für die umgekehrte Zuordnung. Mithilfe des `char2int`-Dictionarys konvertieren wir den Text in ein NumPy-Integer-Array. Die folgende Abbildung zeigt Beispiele für die Konvertierung von Zeichen in Integer-Zahlen und umgekehrt anhand der Wörter »Hello« und »world«:

Zuordnung von Zeichen zu Integer-Zahlen



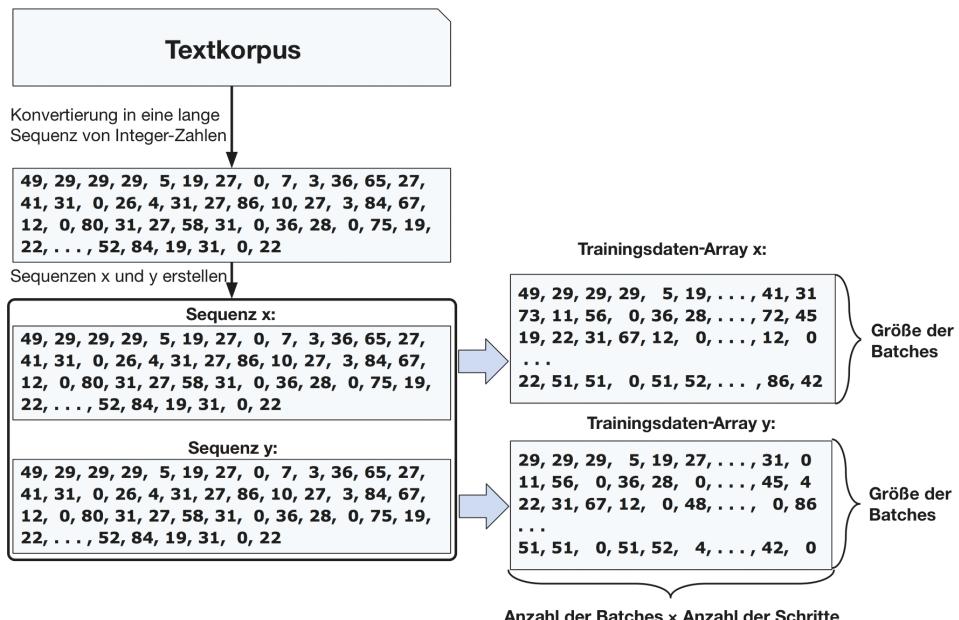
Zuordnung von Integer-Zahlen zu Zeichen



Der Code liest den heruntergeladenen Text ein, entfernt den am Anfang befindlichen Teil, der rechtliche Anmerkungen zum Gutenberg-Projekt enthält, und konstruiert schließlich anhand des verbleibenden Texts die beiden Dictionarys.

```
>>> import numpy as np
>>> ## Text einlesen und verarbeiten
>>> with open('pg2265.txt', 'r', encoding='utf-8') as f:
...     text=f.read()
>>> text = text[15858:]
>>> chars = set(text)
>>> char2int = {ch:i for i,ch in enumerate(chars)}
>>> int2char = dict(enumerate(chars))
>>> text_ints = np.array([char2int[ch] for ch in text],
...                         dtype=np.int32)
```

Nun kommt der wichtigste Schritt der Datenaufbereitung, die Umformung der Daten in Sequenzen für die Stapelverarbeitung. Wie Sie wissen, haben wir zum Ziel, das *nächste* Zeichen anhand der in der Sequenz bereits beobachteten Zeichen vorherzusagen. Deshalb verschieben wir die Eingabe ( $x$ ) und die Ausgabe ( $y$ ) des neuronalen Netzes um ein Zeichen gegeneinander. Die folgende Abbildung zeigt, ausgehend von einem Textkorpus, diesen Schritt der Datenaufbereitung zum Erzeugen von  $x$  und  $y$ :



Wie Sie der Abbildung entnehmen können, besitzen die Trainingsdaten-Arrays  $x$  und  $y$  die gleiche Anzahl Dimensionen. Die Anzahl der Zeilen entspricht der Stapelgröße und die Anzahl der Spalten dem Produkt aus Stapelgröße und Anzahl der Schritte.

Der folgenden Funktion wird das Array `sequence` übergeben, das die dem Textkorpus entsprechenden Integer-Zahlen enthält. Sie erzeugt  $x$  und  $y$ , die wie in der letzten Abbildung aufgebaut sind:

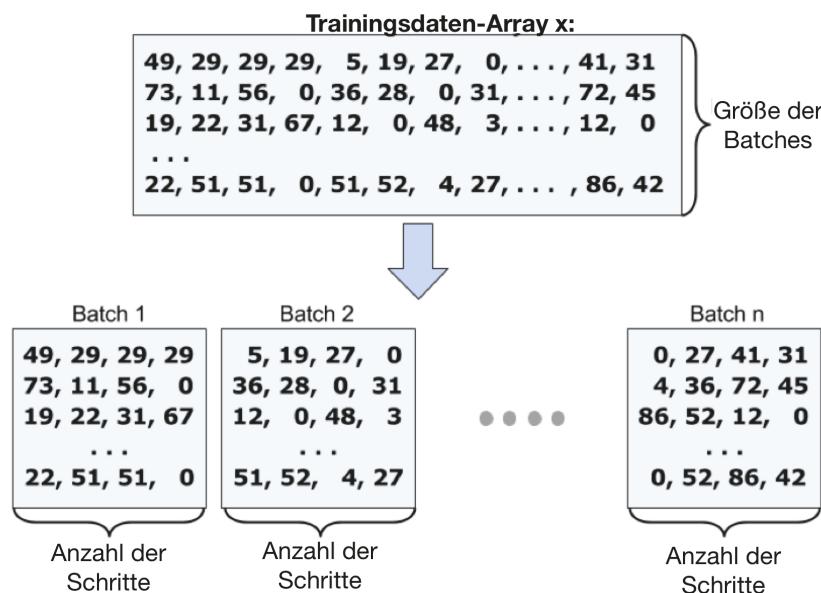
```
>>> def reshape_data(sequence, batch_size, num_steps):
...     tot_batch_length = batch_size * num_steps
...     num_batches = int(len(sequence) / tot_batch_length)
...     if num_batches*tot_batch_length + 1 > len(sequence):
...         num_batches = num_batches - 1
...     ## Sequenz beschneiden, um die übrigen Zeichen zu
...     ## entfernen, die keinen kompletten Batch bilden.
...     x = sequence[0: num_batches*tot_batch_length]
```

```

...
y = sequence[1: num_batches*tot_batch_length + 1]
...
## x & y in eine Liste von Sequenzen aufteilen:
...
x_batch_splits = np.split(x, batch_size)
y_batch_splits = np.split(y, batch_size)
...
## Batches zusammenfügen
...
## batch_size x tot_batch_length
x = np.stack(x_batch_splits)
y = np.stack(y_batch_splits)
...
...
return x, y

```

Der nächste Schritt ist die Aufteilung der Arrays **x** und **y** in Mini-Batches. Jede Zeile entspricht einer Sequenz, deren Länge so groß ist wie die Anzahl der Schritte. Die folgende Abbildung zeigt die Aufteilung des Arrays **x**:



Im folgenden Code definieren wir die Funktion **create\_batch\_generator**, die die Arrays **x** und **y** wie in der Abbildung aufteilt und einen Batch-Generator zurückgibt. Wir werden diesen Generator später verwenden, um während des Trainings des neuronalen Netzes die Mini-Batches zu durchlaufen.

```

>>> def create_batch_generator(data_x, data_y, num_steps):
...     batch_size, tot_batch_length = data_x.shape
...     num_batches = int(tot_batch_length/num_steps)
...     for b in range(num_batches):
...         yield (data_x[:, b*num_steps:(b+1)*num_steps],
...                data_y[:, b*num_steps:(b+1)*num_steps])

```

Damit ist die Datenvorverarbeitung abgeschlossen und die Daten liegen im richtigen Format vor. Im nächsten Abschnitt implementieren wir das RNN-Modell zur Sprachmodellierung.

### 16.5.2 Erstellen eines RNNs zur Sprachmodellierung durch Zeichen

Für das neuronale Netz zur Sprachmodellierung durch Zeichen implementieren wir die Klasse `CharRNN`, die einen Graphen des RNNs konstruiert, das nach der Beobachtung einer bestimmten Zeichensequenz das nächste Zeichen vorhersagt. Aus Sicht der Klassifizierung entspricht die Anzahl der Klassen der Anzahl der im Textkorpus enthaltenen unterschiedlichen Zeichen. Die Klasse `CharRNN` besitzt die folgenden vier Methoden:

- Einen Konstruktor, der die Lernparameter einstellt, einen Berechnungsgraphen erzeugt und die `build`-Methode entsprechend des aktuellen Modus (Training oder Sampling) aufruft
- Eine `build`-Methode, die Platzhalter für die Zuführung von Daten definiert, mithilfe von LSTM-Zellen das RNN erzeugt und Ausgabe, Straffunktion und Optimierer des neuronalen Netzes festlegt
- Eine `train`-Methode, die die Mini-Batches durchläuft, die Wahrscheinlichkeiten des jeweils nächsten Zeichens berechnet und das Netz die angegebene Anzahl von Epochen lang trainiert
- Eine `sample`-Methode, die mit einem angegebenen String startet, die Wahrscheinlichkeit für das nächste Zeichen berechnet und diesen Wahrscheinlichkeiten entsprechend zufällig ein Zeichen auswählt. Dieser Vorgang wird wiederholt, und die ausgewählten Zeichen werden aneinandergereiht, sodass sie einen String bilden. Sobald dieser String von der angegebenen Länge ist, gibt die Methode ihn zurück.

Die vier Methoden werden in den nächsten Abschnitten einzeln erörtert. Beachten Sie, dass die Implementierung des RNN-Teils dieses Modells derjenigen von Projekt 1 sehr ähnlich ist. Wir verzichten deshalb an dieser Stelle auf die Beschreibung des Erstellens der RNN-Komponenten.

### 16.5.3 Der Konstruktor

Bei der Implementierung der Stimmungsanalyse wurde derselbe Berechnungsgraph sowohl für den Trainings- als auch für den Vorhersagemodus verwendet. Im Gegensatz dazu unterscheiden sich die Berechnungsgraphen für das Training und die Vorhersage in diesem Fall.

Wir müssen dem Konstruktor also ein neues boolesches Argument hinzufügen, das festlegt, ob wir ein Trainings- oder ein Vorhersagemodell erzeugen möchten. Der folgende Code zeigt die Implementierung des Konstruktors in der Klassendefinition:

```

import tensorflow as tf
import os

class CharRNN(object):
    def __init__(self, num_classes, batch_size=64,
                 num_steps=100, lstm_size=128,
                 num_layers=1, learning_rate=0.001,
                 keep_prob=0.5, grad_clip=5,
                 sampling=False):
        self.num_classes = num_classes
        self.batch_size = batch_size
        self.num_steps = num_steps
        self.lstm_size = lstm_size
        self.num_layers = num_layers
        self.learning_rate = learning_rate
        self.keep_prob = keep_prob
        self.grad_clip = grad_clip

        self.g = tf.Graph()
        with self.g.as_default():
            tf.set_random_seed(123)
            self.build(sampling=sampling)
            self.saver = tf.train.Saver()
            self.init_op = tf.global_variables_initializer()

```

Das boolesche Argument `sampling` legt also wie geplant fest, ob die CharRNN-Instanz den Graphen im Trainingsmodus (`sampling=False`) oder im Sampling-Modus (`sampling=True`) erstellt.

Neben `sampling` gibt es ein weiteres Argument namens `grad_clip`, das zum Gradienten-Clipping dient, um so das vorhin erwähnte Problem des explosionsartig wachsenden Gradienten zu umgehen.

Ähnlich wie in der vorangegangenen Implementierung erstellt der Konstruktor einen Berechnungsgraphen, initialisiert zwecks reproduzierbarer Ergebnisse den Zufallszahlengenerator und ruft die `build`-Methode auf.

#### 16.5.4 Die build-Methode

Die nächste Methode der CharRNN-Klasse heißt `build` und ist derjenigen im Abschnitt *Projekt 1: Analyse der Stimmungslage in der IMDb-Filmbewertungsdatenbank mit mehrschichtigen RNNs* bis auf einige geringfügige Unterschiede sehr ähnlich. Zunächst werden in Abhängigkeit vom Modus die beiden lokalen Variablen `batch_size` und `num_steps` wie folgt definiert:

Im Sampling-Modus:

```
batch_size = 1 und num_steps = 1
```

Im Trainingsmodus:

```
batch_size = self.batch_size und num_steps = self.num_steps
```

Bei der Implementierung der Stimmungsanalyse hatten wir eine eingebettete Schicht verwendet, um eine Repräsentierung der verschiedenen auffälligen Wörter in der Datenmenge zu erstellen. Hier kommt stattdessen sowohl für  $x$  als auch für  $y$  die One-hot-Codierung zum Einsatz, und zwar mit `depth=num_classes`, wobei `num_classes` die Gesamtzahl der Zeichen im Textkorpus angibt.

Das Erstellen der mehrschichtigen Komponente des Modells erfolgt auf genau die gleiche Weise wie bei der Implementierung der Stimmungsanalyse mit der Funktion `tf.nn.dynamic_rnn`. In der Funktion `tf.nn.dynamic_rnn` ist `outputs` hier ein dreidimensionaler Tensor mit der Shape (`batch_size, num_steps, lstm_size`), der in einen zweidimensionalen Tensor mit der Shape (`batch_size*num_steps, lstm_size`) umgeformt wird, der an die Funktion `tf.layers.dense` übergeben wird, um eine vollständig verknüpfte Schicht zu erstellen und `logits` (Nettoeingaben) zu erhalten. Schließlich werden die Wahrscheinlichkeiten für den nächsten Stapel an Zeichen errechnet und die Straffunktion wird definiert. Darüber hinaus wenden wir hier mit der Funktion `tf.clip_by_global_norm` das Gradienten-Clipping an, um das Problem des explosionsartig wachsenden Gradienten zu umgehen.

Der nachstehende Code zeigt die Implementierung der soeben beschriebenen `build`-Methode:

```
def build(self, sampling):
    if sampling == True:
        batch_size, num_steps = 1, 1
    else:
        batch_size = self.batch_size
        num_steps = self.num_steps

    tf_x = tf.placeholder(tf.int32,
                          shape=[batch_size, num_steps],
                          name='tf_x')
    tf_y = tf.placeholder(tf.int32,
                          shape=[batch_size, num_steps],
                          name='tf_y')
    tf_kepprob = tf.placeholder(tf.float32,
                               name='tf_kepprob')

    # One-hot-Codierung:
    x_onehot = tf.one_hot(tf_x, depth=self.num_classes)
```

```
y_onehot = tf.one_hot(tf_y, depth=self.num_classes)

### Mehrschichtige RNN-Zellen erstellen
cells = tf.contrib.rnn.MultiRNNCell(
    [tf.contrib.rnn.DropoutWrapper(
        tf.contrib.rnn.BasicLSTMCell(self.lstm_size),
        output_keep_prob=tf_keepprob)
     for _ in range(self.num_layers)])

## Anfangszustand definieren
self.initial_state = cells.zero_state(
    batch_size, tf.float32)
## Sequenzen mit dem RNN verarbeiten
lstm_outputs, self.final_state = tf.nn.dynamic_rnn(
    cells, x_onehot,
    initial_state=self.initial_state)

print(' << lstm_outputs >>', lstm_outputs)

seq_output_reshaped = tf.reshape(
    lstm_outputs,
    shape=[-1, self.lstm_size],
    name='seq_output_reshaped')
logits = tf.layers.dense(
    inputs=seq_output_reshaped,
    units=self.num_classes,
    activation=None,
    name='logits')
proba = tf.nn.softmax(
    logits,
    name='probabilities')
y_reshaped = tf.reshape(
    y_onehot,
    shape=[-1, self.num_classes],
    name='y_reshaped')
cost = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(
        logits=logits,
        labels=y_reshaped),
    name='cost')

# Gradient-Clipping zur Umgehung des Problems des
# explosionsartig wachsenden Gradienten
tvars = tf.trainable_variables()
grads, _ = tf.clip_by_global_norm(
```

```

        tf.gradients(cost, tvars),
        self.grad_clip)
optimizer = tf.train.AdamOptimizer(self.learning_rate)
train_op = optimizer.apply_gradients(
    zip(grads, tvars),
    name='train_op')

```

### 16.5.5 Die train-Methode

Auch die nächste Methode der CharRNN-Klasse namens `train` ist der im Abschnitt *Projekt 1: Analyse der Stimmungslage in der IMDb-Filmbewertungsdatenbank mit mehrschichtigen RNNs* beschriebenen sehr ähnlich. Der nachfolgende Code weist eine vergleichbare Struktur zur Version der Stimmungsanalyse auf:

```

def train(self, train_x, train_y,
          num_epochs, ckpt_dir='./model/'):
    ## Verzeichnis erstellen, falls noch nicht vorhanden
    if not os.path.exists(ckpt_dir):
        os.mkdir(ckpt_dir)

    with tf.Session(graph=self.g) as sess:
        sess.run(self.init_op)

        n_batches = int(train_x.shape[1]/self.num_steps)
        iterations = n_batches * num_epochs
        for epoch in range(num_epochs):

            # Netz trainieren
            new_state = sess.run(self.initial_state)
            loss = 0
            ## Minibatch-Generator:
            bgen = create_batch_generator(
                train_x, train_y, self.num_steps)
            for b, (batch_x, batch_y) in enumerate(bgen, 1):
                iteration = epoch*n_batches + b

                feed = {'tf_x:0': batch_x,
                        'tf_y:0': batch_y,
                        'tf_keepprob:0': self.keep_prob,
                        self.initial_state : new_state}
                batch_cost, _, new_state = sess.run(
                    ['cost:0', 'train_op',
                     self.final_state],
                    feed_dict=feed)
                if iteration % 10 == 0:

```

```

        print('Epoch %d/%d Iteration %d'
              '| Verlust Training: %.4f' % (
                  epoch + 1, num_epochs,
                  iteration, batch_cost))

    ## Trainiertes Modell speichern
    self.saver.save(
        sess, os.path.join(
            ckpt_dir, 'language_modeling.ckpt'))

```

### 16.5.6 Die sample-Methode

Die letzte Methode der CharRNN-Klasse trägt den Namen `sample`. Sie zeigt ein ähnliches Verhalten wie die im Abschnitt *Projekt 1: Analyse der Stimmungslage in der IMDb-Filmbewertungsdatenbank mit mehrschichtigen RNNs* implementierte `predict`-Methode. Der Unterschied besteht darin, dass wir hier anhand der beobachteten Sequenz `observed_seq` die Wahrscheinlichkeiten für das nächste Zeichen berechnen. Diese Zeichen werden an eine Funktion namens `get_top_char` übergeben, die gemäß der Wahrscheinlichkeiten zufällig ein Zeichen auswählt.

Am Anfang besitzt die Sequenz den Wert `starter_seq`, der als Argument übergeben wird. Die gemäß der Wahrscheinlichkeiten ausgewählten Zeichen werden an die bisher beobachtete Sequenz angehängt, die dann zur Vorhersage des nächsten Zeichens herangezogen wird.

Hier ist die Implementierung der `sample`-Methode:

```

def sample(self, output_length,
          ckpt_dir, starter_seq="The "):
    observed_seq = [ch for ch in starter_seq]
    with tf.Session(graph=self.g) as sess:
        self.saver.restore(
            sess,
            tf.train.latest_checkpoint(ckpt_dir))
        ## 1: Modell auf die Startsequenz anwenden
        new_state = sess.run(self.initial_state)
        for ch in starter_seq:
            x = np.zeros((1, 1))
            x[0,0] = char2int[ch]
            feed = {'tf_x:0': x,
                    'tf_kepprob:0': 1.0,
                    self.initial_state: new_state}
            proba, new_state = sess.run(
                ['probabilities:0', self.final_state],
                feed_dict=feed)

```

```

ch_id = get_top_char(proba, len(chars))
observed_seq.append(int2char[ch_id])

## 2: Modell auf die aktualisierte Sequenz anwenden
for i in range(output_length):
    x[0,0] = ch_id
    feed = {'tf_x:0': x,
            'tf_keepprob:0': 1.0,
            self.initial_state: new_state}
    proba, new_state = sess.run(
        ['probabilities:0', self.final_state],
        feed_dict=feed)

    ch_id = get_top_char(proba, len(chars))
    observed_seq.append(int2char[ch_id])

return ''.join(observed_seq)

```

Die `sample`-Methode ruft also die Funktion `get_top_char` auf, die gemäß der Wahrscheinlichkeiten zufällig die ID eines Zeichens (`ch_id`) auswählt.

In der `get_top_char`-Funktion werden die Wahrscheinlichkeiten zunächst sortiert. Anschließend werden die ersten `top_n` Wahrscheinlichkeiten an die Funktion `numpy.random.choice` übergeben, die zufällig eine davon auswählt. Die Implementierung der `get_top_char` sieht folgendermaßen aus:

```

def get_top_char(probas, char_size, top_n=5):
    p = np.squeeze(probas)
    p[np.argsort(p)[-top_n]] = 0.0
    p = p / np.sum(p)
    ch_id = np.random.choice(char_size, 1, p=p)[0]
    return ch_id

```

Diese Funktion muss natürlich schon definiert sein, bevor die CharRNN-Klasse erstellt wird. Wir haben das Ganze in dieser Reihenfolge dargestellt, um die zugrunde liegenden Konzepte in der richtigen Reihenfolge erläutern zu können. Stöbern Sie in dem zu diesem Kapitel gehörigen Code-Notebook, um einen besseren Überblick über die Reihenfolge der Funktionsdefinitionen zu erlangen.

### 16.5.7 Erstellen und Trainieren des CharRNN-Modells

Nun sind wir so weit, dass wir die CharRNN-Klasse instanziiieren können, um das RNN-Modell zu erstellen und mit folgenden Einstellungen zu trainieren:

```
>>> batch_size = 64
>>> num_steps = 100
>>> train_x, train_y = reshape_data(text_ints,
...                                     batch_size,
...                                     num_steps)
>>>
>>> rnn = CharRNN(num_classes=len(chars),
...                  batch_size=batch_size)
>>> rnn.train(train_x, train_y,
...             num_epochs=100,
...             ckpt_dir='./model-100/')
```

Das trainierte Modell wird in einem Verzeichnis namens `./model-100/` abgespeichert, damit wir es später wiederherstellen können, um Vorhersagen zu treffen oder mit dem Training fortzufahren.

### 16.5.8 Das CharRNN-Modell im Sampling-Modus

Jetzt erstellen wir eine Instanz der CharRNN-Klasse im Sampling-Modus durch Angabe des Parameters `sampling=True`. Wir rufen die `sample`-Methode auf, die das im Verzeichnis `./model-100/` gespeicherte Modell lädt, und erzeugen eine 500 Zeichen lange Sequenz:

```
>>> del rnn
>>> np.random.seed(123)
>>> rnn = CharRNN(len(chars), sampling=True)
>>> print(rnn.sample(ckpt_dir='./model-100/',
...                   output_length=500))
```

Der erzeugte Text sieht folgendermaßen aus:

```
Ham. I woll thenke the Solde and as ither thes will, at a dis tantend
To maness of he and mering and the bus and,
The hiss a fit of ant ort time, and her wind of
A beart of his mine it a faulouthensers

Hal. Whe that so my Larger,
Thin we selfe to mat tean the hims but
With was sore to beene tive to ser is betit,
Was thin so a mangers and hill or and asthie

Hor. This mest the senges of hation thee to hos the herr,
The sacke a my Lort worke. That his she lete,
And whise howers
```

Die Ausgabe zeigt, dass manche englischen Wörter im Wesentlichen erhalten bleiben. Hier ist außerdem zu beachten, dass die Ausgabe auf einem altenglischen

Text beruht, sodass einige Wörter des ursprünglichen Texts fremdartig erscheinen. Um ein besseres Ergebnis zu erzielen, müssten wir das Modell mit einer höheren Anzahl von Epochen trainieren. Probieren Sie es doch einfach mal aus, ein deutlich größeres Textdokument zu verwenden und beim Trainieren die Anzahl der Epochen zu erhöhen.

## 16.6 Zusammenfassung und Schlusswort

Wir können uns nur wünschen, dass Ihnen dieses letzte Kapitel und die spannende Tour durch das Machine Learning und Deep Learning gefallen haben. Sie haben in diesem Buch alle wesentlichen Inhalte kennengelernt, die dieses Fachgebiet zu bieten hat. Damit sollten Sie gut gerüstet sein, um diese Verfahren in der Praxis umzusetzen und echte Aufgabenstellungen zu lösen.

Als Erstes haben Sie einen kurzen Überblick über die verschiedenen Arten des Machine Learnings erhalten: überwachtes, verstärkendes und unüberwachtes Lernen. Dann haben wir verschiedene Lernalgorithmen für die Klassifizierung erörtert. Kapitel 2, *Lernalgorithmen für die Klassifizierung trainieren*, hatte ein einfaches, einschichtiges neuronales Netz zum Thema.

Anschließend haben wir uns in Kapitel 3, *Machine-Learning-Klassifizierer mit scikit-learn verwenden*, mit fortgeschritteneren Klassifizierungsalgorithmen befasst, und Sie haben in Kapitel 4, *Gut geeignete Trainingsdatenmengen: Datenvorverarbeitung* und Kapitel 5, *Datenkomprimierung durch Dimensionsreduktion*, die wichtigsten Aspekte einer Pipeline zum Machine Learning kennengelernt.

Vergessen Sie nicht, dass selbst die ausgeklügeltesten Algorithmen auf die in den zum Lernen eingesetzten Trainingsdaten enthaltenen Informationen beschränkt sind. Deshalb haben wir in Kapitel 6, *Bewährte Verfahren zur Modellbewertung und Hyperparameter-Abstimmung*, erprobte Verfahren zur Entwicklung und Beurteilung von Vorhersagemodellen vorgestellt, ein wichtiger Aspekt bei der Anwendung des Machine Learnings.

Wenn ein einzelner Lernalgorithmus nicht die erwünschte Leistung erbringt, kann es manchmal hilfreich sein, ein Ensemble zusammenzustellen, um Vorhersagen zu treffen. Dieses Konzept haben wir in Kapitel 7, *Kombination verschiedener Modelle für das Ensemble Learning*, untersucht.

In Kapitel 8, *Machine Learning zur Analyse von Stimmungslagen nutzen*, haben wir das Machine Learning zur Analyse der wohl interessantesten Art von Daten im heutigen Zeitalter eingesetzt, das von den Social Media im Internet dominiert wird: Textdokumente.

Machine-Learning-Verfahren sind jedoch nicht auf Offline-Datenanalysen beschränkt, und in Kapitel 9, *Einbettung eines Machine-Learning-Modells in eine*

*Webanwendung*, haben Sie erfahren, wie sich ein Lernmodell in eine Webanwendung einbetten lässt, um es der Außenwelt zugänglich zu machen.

Wir haben uns größtenteils auf Klassifizierungsalgorithmen konzentriert, dem wohl verbreitetsten Anwendungsgebiet des Machine Learnings. Das war aber noch nicht alles! In Kapitel 10, *Vorhersage stetiger Zielvariablen durch Regressionsanalyse*, haben wir verschiedene Algorithmen für die Regressionsanalyse erkundet, um stetige Ausgabewerte vorherzusagen.

Die Clusteranalyse ist ein weiteres spannendes Teilgebiet des Machine Learnings, das uns helfen kann, verborgene Strukturen in den Daten aufzuspüren, selbst wenn unsere Trainingsdaten nicht die richtigen Antworten enthalten, aus denen ein Modell lernen könnte. Damit haben wir uns in Kapitel 11, *Verwendung nicht gekennzeichneter Daten: Clusteranalyse*, befasst.

Dann haben wir unsere Aufmerksamkeit einem der spannendsten Algorithmen des Machine Learnings gewidmet: künstlichen neuronalen Netzen. In Kapitel 12, *Implementierung eines künstlichen neuronalen Netzes*, haben wir zunächst von Grund auf mit NumPy ein mehrschichtiges Perzepton implementiert.

Die Leistungsfähigkeit von TensorFlow wurde in Kapitel 13, *Parallelisierung des Trainings neuronaler Netze mit TensorFlow*, deutlich, in dem wir TensorFlow verwendet haben, um das Erstellen von Modellen neuronaler Netze zu vereinfachen und von Grafikprozessoren Gebrauch zu machen, die ein effizienteres Training mehrschichtiger neuronaler Netze ermöglichen.

In Kapitel 14, *Die Funktionsweise von TensorFlow im Detail*, haben wir uns eingehender mit TensorFlow befasst und verschiedene Aspekte und Mechanismen dieser Bibliothek erörtert, wie etwa Variablen und Operatoren in Tensorflows Berechnungsgraphen, Gültigkeitsbereichen von Variablen, das Starten von Graphen und verschiedene Möglichkeiten, Knoten auszuführen.

Kapitel 15, *Bildklassifizierung mit tiefen konvolutionalen neuronalen Netzen*, hatte konvolutionale neuronale Netze zum Thema, die derzeit dank ihrer ausgezeichneten Leistung bei Bilderkennungsaufgaben umfassend für das maschinelle Sehen eingesetzt werden.

Und in diesem letzten Kapitel 16, *Modellierung sequenzieller Daten durch rekurrente neuronale Netze*, haben Sie die Sequenzmodellierung mit RNNs kennengelernt. Eine ausführliche Darstellung des Deep Learnings geht zwar weit über den Rahmen dieses Buches hinaus, wir hoffen aber, dass wir zumindest Ihr Interesse wecken könnten, die Fortschritte auf diesem Fachgebiet zu verfolgen.

Wenn Sie mit dem Gedanken spielen, das Machine Learning selbst zu erforschen, oder einfach nur auf dem neuesten Stand bleiben möchten, empfehlen wir, die Arbeiten der führenden Experten auf diesem Gebiet zu verfolgen. Um nur einige zu nennen:

- Geoffry Hinton (<http://www.cs.toronto.edu/~hinton/>)
- Andrew Ng (<http://www.andrewng.org>)
- Yann LeCun (<http://yann.lecun.com>)
- Jürgen Schmidhuber (<http://people.idsia.ch/~juergen/>)
- Yoshua Bengio ([http://www.iro.umontreal.ca/~bengioy/yoshua\\_en/](http://www.iro.umontreal.ca/~bengioy/yoshua_en/))

Zögern Sie nicht, den Mailinglisten von scikit-learn, TensorFlow und Keras beizutreten und an interessanten Diskussionen über diese Bibliotheken und Machine Learning im Allgemeinen teilzunehmen. Und zu guter Letzt können Sie sich auch darüber informieren, was wir, die Autoren, so treiben: <http://sebastianraschka.com> und <http://vahidmirjalili.com>.

Wir freuen uns darauf, von Ihnen zu hören! Sie können uns auch gern persönlich kontaktieren, wenn Sie Fragen zu diesem Buch haben oder allgemeine Ratschläge zum Thema Machine Learning suchen.

# Stichwortverzeichnis

- 1-zu-n-Beziehung 532  
\_Konvention (Unterstrich) 47
- A**
- Abrundungsfunktion 495
  - Abstimmungsparameter 191
  - AdaBoost 258
  - AdaBoost-Algorithmus 261
  - AdaBoost-Klassifizierer-Ensemble 263
  - Adaline 56
  - Adaline-Algorithmus 385
  - Adam-Optimierer 515
  - Agent 29
  - Agglomeratives Clustering 368, 375
  - Ähnlichkeitsfunktion 104
  - Aktivierungsfunktion 42, 386
    - sigmoide 444
    - Tangens hyperbolicus 445
  - Anaconda 38
  - Antwortausfälle 125
  - Ausdruck
    - regulärer 276
  - Ausgabespektrum 449
  - Ausreißer 332
  - Average Linkage 368
- B**
- Backpropagation-Through-Time 537
  - Bagging 253
  - Bag-of-words-Modell 270
  - Bayes-Klassifikator 281
  - Belohnungsfunktion 30
  - Belohnungssignal 29
  - Berechnungsgraph 427, 454
    - Flusskontrolle 481
  - Bestimmtheitsmaß 337
  - Bias 93, 94
  - Bias-Einheit 43, 388, 432
  - Bias-Varianz-Kompromiss 94
  - Bibliotheken 47
  - bincount-Funktion 76
  - Binomialkoeffizient 235
- Binomialverteilung 235  
Boosting 258
- C**
- Chancenverhältnis 81
  - close-Methode 296
  - Cluster
    - Trägheit 356
  - Clusteranalyse 353
  - Clustering 31
    - agglomeratives 368, 375
    - divisives 368
    - graphenbasiertes 381
    - hierarchisches 368
    - prototypbasiertes 354
    - spektrales 381
  - Clustering-Güte 362, 363
  - CNN
    - Bausteine 489
    - Implementierung 501
    - Implementierung mit Layers-API 521
    - Implementierung mit Low-level-API 509
    - mehrschichtige Architektur 507
  - Codierung der Klassenbezeichnung 132
  - commit-Methode 296
  - Complete Linkage 368
  - connect-Methode 296
  - contrib-Modul 439
  - corrcoef-Funktion 325
  - CountVectorizer-Klasse 270
  - CSV-Datei 126, 269
  - CUDA 425
  - cumsum-Funktion 164
  - cursor-Methode 296
- D**
- DataFrame-Objekt 51
  - Daten
    - fehlende 125
  - Datenanalyse
    - explorative 321
  - Datenbank 295

Datenprojektion 193  
Datenvorverarbeitung 35  
DBSCAN 376  
DecisionTreeRegressor 347  
Deep Learning 383  
DeepFace 384  
Deep-Neural-Network (DNN) 383  
DeepSpeech 384  
Dendrogramm 368, 373  
Deserialisierung 293  
Dichte 376  
Differenzierbarkeit 57  
Differenzieren  
  automatisches 414  
Dimensionalität 32  
Dimensionsreduktion 32, 147  
Distanzmaß  
  euklidisches 355  
Distanzmatrix 369  
Divisives Clustering 368  
Dokumenthäufigkeit  
  inverse 272  
dropna-Methode 127  
Dropout 505  
Dummy-Merkmal 134  
Dünnbesetzter Merkmalsvektor 271

## E

Eigenvektor 163  
Eigenwert 163  
Einsbettung 545  
Eingabekanal 502  
Elastic-Net-Verfahren 339  
Ellenbogenkriterium 362  
Emoticon 275  
Ensemble 116  
  Fehlerquote 236  
Ensemble Learning 233  
Ensemblemethoden 233  
Entropie 108, 346  
Entscheidungsbaum 107  
Entscheidungsbaum-Regression 346  
Entscheidungsbereiche 78  
Epoche 46  
Erklärende Variable 318  
Ersetzung (mit/ohne) 116  
Euklidische Distanz 355  
Euklidische Metrik 121  
execute-Methode 296  
Explorative Datenanalyse (EDA) 321  
Extraktion 159

## F

F1-Maß 223  
Falsch-Positiv-Rate 222  
Faltung  
  diskrete 491  
  eindimensionale 492  
  Zero-Padding 494  
  zweidimensionale 497  
Farbkanal 502  
FCM-Algorithmus 360  
Feedforward-Netz 387  
Fehlende Daten 125  
Fehlerquote 222, 235  
Fehlklassifizierung 53  
Fehlklassifizierungsrate 77  
Feld  
  rezeptives 490  
Filter 492  
fit\_transform-Methode 133  
fit-Methode 50, 129  
Flask 297, 298  
floor-Funktion 495  
Fluch der Dimensionalität 122, 151, 380  
Flusskontrolle 481  
Formularvalidierung 300, 301  
Full-Padding 494  
Funktion  
  logistische 82, 445  
Fuzziness-Koeffizient 361  
Fuzzy-C-Mean-Algorithmus (FCM) 359

## G

Galton, Francis 29  
Gaußklammer 495  
Gaußscher Kernel 104  
Generator 437  
get\_dummies-Methode 135  
Gewichtung 238  
Gewichtungsvektor 45  
Gini-Koeffizient 108  
Global Interpreter Lock (GIL) 423  
Glorot-Initialisierung 442, 462  
Google Translate 384  
Gradientenabstiegsverfahren 57  
  als Stapelverarbeitung 59  
  stochastisches 66  
Grafikprozessor (GPU) 424  
Graphenbasiertes Clustering 381  
GraphViz 113  
Graustufenkanal 502

**H**

- Halbmondform 378
- Hamlet 556
- HashingVectorizer 283
- Hauptkomponentenanalyse 159, 168
- Heatmap 325, 373
- Heaviside-Funktion 43, 386
- Hierarchisches Clustering 368
- Hoff, Tedd 56
- Holdout-Methode 205
- Hyperebene 103
- Hyperparameter 37, 62, 205, 216, 387

**I**

- IMDb-Datensammlung 268
- Imputer-Klasse 128
- Informationsgewinn 107, 346
- Inlier 332
- Interpolationsverfahren 128
- Inverse Dokumenthäufigkeit 272
- inverse\_transform-Methode 133
- Iris-Datensammlung 32
- isnull-Methode 126

**J**

- Jinja2 302

**K**

- Kante
  - rekurrente 533
- Keras 434, 439
- Kernel 104
- Kernel-Funktion 104, 182
- Kernel-Hauptkomponentenanalyse 181
- Kernel-Methode 102
- KernelPCA-Klasse 197
- Kernel-SVM 101
- Kernel-Trick 104, 183
- Kernobjekt 376
- Kettenregel (Ableitung) 413
- Klasse 28
- Klassenverteilung
  - unausgewogene 228
- Klassifizierer
  - Fehlertypen 220
  - schwacher 259
- Klassifizierer-Attribute 251
- Klassifizierung 27
  - binäre 27, 42
  - dichotome 42
  - unüberwachte 31

- Klassifizierungsfehler 108
- Klassifizierungsgüte 220
- k-Means++-Algorithmus 359
- k-Means-Algorithmus 353
- KMeans-Klasse 356
- k-Nearest-Neighbor-Algorithmus 119
- KNN Siehe k-Nearest-Neighbor-Algorithmus
- Konfusionsmatrix 220
- Konvention (Unterstrich) 47
- Konvergenz 418
- Konvex 57
- Konvolutionales neuronales Netz Siehe CNN
- Konvolutionsschicht 491
- Kopplungsmatrix 370
- Korrektklassifizierungsrate 36, 77, 222
- Korrelationskoeffizient 323
- Korrelationsmatrix 323, 325
- Kovarianz 162
- Kovarianzmatrix 162, 323
- Kreuzentropie 442
- Kreuzkorrelation 493
- Kreuzvalidierung
  - 2-fache 205
  - 5x2- 218
  - k-fache 206
  - stratifizierte k-fache 208
  - verschachtelte 218

**L**

- L1-Regularisierung 141, 143
- L2-Regularisierung 94, 141
- L2-Strafterm 142
- LabelEncoder-Klasse 133
- LabelEncoder-Objekt 202
- Lancaster-Stemmer 278
- LASSO 339
- LatentDirichletAllocation-Klasse 286
- Latente Dirichlet-Allokation 286
- Lazy-Learning-Algorithmus 119
- LDA Siehe Lineare Diskriminanzanalyse
- LDA-Klasse 180
- Leave-One-Out-Kreuzvalidierung (LOO) 208
- Lebensbedingungen-Datensammlung 320
- Leerraum 277
- Lemmat 278
- Lemmatisierung 278
- Lernen
  - instanzbasiertes 119
  - überwachtes 26
  - unüberwachtes 26, 31, 353
  - verstärkendes 26

- Lernkurve 212  
Lernrate 44, 62  
adaptive 66  
LIBLINEAR-Bibliothek 100  
LIBSVM-Bibliothek 101  
linalg.eig-Funktion 163  
Lineare Diskriminanzanalyse (LDA) 171  
Lineare Regression 29, 318  
Lineare Trennbarkeit 46, 56  
load-Funktion 398  
LogisticRegression-Klasse 91  
Logistische Funktion 82, 445  
Logistische Regression 81  
logit-Funktion 81  
Log-Likelihood-Funktion 85  
Löscher-Gate 539  
LSTM  
    Ausgabe-Gate 539  
    Eingabe-Gate 539  
    Löscher-Gate 539  
    Speicherzelle 538  
    Zellzustand 538  
LSTM (Long Short-Term Memory) 538
- M**
- Machine Learning  
    Anwendungen 26  
Makro-Mittelwertbildung 227  
Manhattan-Metrik 122  
map-Methode 132  
Margin 97  
matplotlib 39  
Matrix-Vektor-Multiplikation 61  
Max-Pooling 500  
McCulloch, Warren 41, 383  
mean-Methode 65  
Mean-Pooling 500  
Median der absoluten Abweichungen (MAD)  
    333  
Medianwert 128  
Medoid 354  
Mehrfachklassifizierung 27, 227  
Mehrheit  
    absolute 233  
    relative 234  
Mehrheitsentscheidung 120, 233  
Mehrheitsentscheidungs-Klassifizierer 237  
Merkmal 33  
    Auswahl 140, 147  
    Bedeutung 154  
Extrahierung 147  
Extraktion 159  
Interpretierbarkeit 156  
nominales 130  
ordinales 130  
Merkmashierarchie 490  
Merkmalskarte 503  
Merkmalstransformation 165  
Merkmalsvektor  
    dünnbesetzter 270  
Methode der kleinsten Quadrate 326  
metric-Parameter 122  
metrics-Modul 77, 223  
Microframework 297  
Mikro-Mittelwertbildung 227  
Mini-Batch Learning 67, 406, 419  
Minkowski-Metrik 122  
Min-Max-Skalierung 138  
Mit Ersetzung 116  
Mittelwertimputation 128  
Mittelwertvektor 174  
Mittlere quadratische Abweichung 337, 469  
MNIST-Datensammlung 393  
Modalwert 238  
model\_selection-Modul 75  
Modellauswahl 205  
Modellbewertung 207  
Modul  
    contrib 439  
    TensorBoard 484  
multiprocessing-Bibliothek 424
- N**
- NaN (Not a Number) 125  
Natural Language Processing (NLP) 267  
Natural Language Toolkit (NLTK) 277  
Negative Klasse 28  
Nettoeingabe 42  
Neuron 41  
    adaptives lineares 41  
Neuronales Netz 383  
    Konvergenz 418  
N-Gramm 272  
NLP Siehe Natural Language Processing  
Nominales Merkmal 130  
Normalgleichung 332  
Normierung 35, 138  
n-zu-1-Beziehung 531  
n-zu-n-Beziehung 532

**O**

Ohne Ersetzung 116  
 One-hot-Codierung 134, 388  
 OneHotEncoder-Klasse 134  
 One-vs.-All (OvA) 51  
 One-vs-all-Verfahren (OvA) 388  
 Online Learning 67, 101  
 OpenCL 425  
 Opinion Mining 267  
 Ordinales Merkmal 130  
 Out-of-Core Learning 282

**P**

Padding 492, 494  
 pairplot-Funktion 321  
 pandas 38  
 partial\_fit-Methode 67  
 PCA (Principal Component Analysis) *Siehe*  
     Hauptkomponentenanalyse  
 PCA-Klasse 168  
 permutation-Methode 70  
 Perzepron 41, 46  
     mehrschichtiges 387  
 Perzepron-Lernregel 44  
 pickle-Modul 291  
 pip (Installationsprogramm) 38  
 Pipeline 201, 203, 217  
 Pitts, Walter 41, 383  
 Platzhalter 430, 458  
     Daten zuführen 459  
 plot\_decision\_regions-Methode 79  
 Pooling-Größe 500  
 Pooling-Schicht 491  
 Porter-Stemmer-Algorithmus 277  
 Positive Klasse 28  
 predict\_proba-Methode 92  
 predict-Methode 50, 77, 129  
 preprocessing-Modul 76  
 Problem des explosionsartig wachsenden  
     Gradienten 537  
 Problem des verschwindenden Gradienten  
     387, 449, 537  
 Projektionsmatrix 165  
 Pruning 108  
 pydotplus 114  
 PythonAnywhere 312  
 Python-Pakete 37

**R**

Randobjekt 376  
 Random Forest 116, 348

RandomForestClassifier 154  
 Random-Forest-Klassifizierer 118  
 Random-Forest-Regression 346, 348  
 Randomisierte Suche 218  
 Rang eines Tensors 427, 454  
 RANSAC-Algorithmus 333  
 Rastersuche 216  
 Rauschobjekt 377  
 RBF-Kernel 104  
 read\_csv-Funktion 126  
 Receiver-Operating-Characteristic-Diagramme (ROC-Diagramme) 224  
 Regex-Bibliothek 275  
 Regression 27  
     Gradientenabstiegsverfahren 90  
     lineare 29, 318  
     logistische 81  
     multiple 319  
     polynomiale 340  
 Regressionsanalyse 28  
 Regressionsgerade 318  
 Regressionsmodell  
     Leistung 335  
     multiples 335  
     Random Forest 349  
 Regulärer Ausdruck 275, 276  
 Regularisierung 94, 338  
     Dropout 505  
 Regularisierungsparameter 94  
 Regularisierungsstärke 216  
 Regularisierungsterm 411  
 Rekurrente Kante 533  
 Rekurrentes neuronales Netz *Siehe* RNN  
 ReLU (Rektifizierte Lineareinheit) 449  
 Residualdiagramm 336  
 Residuum 318  
 Rezeptives Feld 490  
 Richtig-Positiv-Rate 222  
 Ridge-Regression 339  
 RNN  
     Aktivierung 534  
     Architektur 533  
     Datenaufbereitung 541  
     Datenbereinigung 556  
     Implementierung mit TensorFlow 540  
     Korrektklassifizierungsrate 555  
     Modell erstellen 547  
     Wahrscheinlichkeit 565  
     Zeitschritt 534  
 RNN (Rekurrentes neuronales Netz) 529  
 ROC AUC 224

ROC-Kurve 224  
 Rosenblatt, Frank 42

**S**

Same-Padding 494  
 savez-Funktion 398  
 SBS-Algorithmus 148  
 Schätzer 129, 204  
 Schlupfvariable 98  
 Schwellenwertfunktion 386  
 scikit-learn-API 74  
 score-Methode 78  
 Seaborn-Bibliothek 321  
 select-Befehl 296  
 SelectFromModel-Objekt 156  
 Sentimentanalyse 267  
 Sequential Backwards Selection *Siehe* SBS-Algorithmus  
 Sequenz  
     Eigenschaften 530  
     Repräsentierung 530  
 Sequenzmodellierung  
     Kategorien 531  
 Serialisierung 291, 475  
 SGDClassifier-Klasse 101  
 shuffle-Methode 67  
 Sigmoidfunktion 82, 444  
 Signal 492  
 Silhouettenanalyse 363  
 Silhouettendiagramm 365  
 Silhouettenkoeffizient 363  
 Single Linkage 368  
 Skalarprodukt 50  
 Soft-Margin-Klassifizierung 98  
 softmax-Funktion 447  
 Spektrales Clustering 381  
 Sprungfunktion 43  
 SQLite 295  
 SQLite Manager (Browser-Plugin) 296  
 sqlite3-Bibliothek 295  
 Stacking 252  
 Stammformreduktion 277  
 Standardisierung 64, 138  
 StandardScaler-Klasse 76  
 std-Methode 65  
 Stemming 277  
 Stetigkeit 317  
 Stimmungsanalyse 267, 541  
 Stopwort 278

Straffunktion 57, 84, 85, 385  
     Fehleroberfläche 413  
     logistische 410  
 Strafterm 142  
 stream\_docs-Funktion 283  
 Streudiagrammmatrix 321  
 Streumatrix 174  
 StringIO-Funktion 126  
 sum-Methode 126  
 Support Vector Machine (SVM) 96, 351  
 SVM *Siehe* Support Vector Machine

**T**

Tangens hyperbolicus 445, 448  
 Tensor 427  
     Rang 454  
 TensorBoard 484  
 TensorFlow  
     Berechnungsgraph 427, 456  
     Geltungsbereich von Variablen 465  
     Generator 466  
     GPU-Unterstützung 426  
     Grundlagen 425  
     High-level-API 434  
     Keras-API 434  
     Knoten 457  
     Layers-API 434  
     Low-level-API 430  
     Merkmale 453  
     Modell speichern und wiederherstellen 474  
     Objekt im Graphen ausführen 473  
     Operation 456  
     Platzhalter 430, 458  
     Regressionsmodell 469  
     run-Methode 458  
     Sitzung 457  
     Tensor 454  
     Tensor transformieren 477  
     Trainingsleistung 423  
     Variable 430, 461  
     Variableninitialisierung 464  
     Wiederverwendung von Variablen 466  
 TensorShape-Klasse 455  
 Tensor-Transformation 477  
 Testdatenmenge 205  
 Textbereinigung 275  
 Textfeld 301  
 tf.bool 525

tf.case 483  
 tf.rank 455  
 Tf-idf-Maß 272  
 toarray-Methode 134  
 Token 270  
 Tokenisierung 277  
 Tonalität 267  
 Topic Modeling 285  
 train\_test\_split-Funktion 137  
 Trainingsdatenmenge 205  
 Transformer 204  
 Transformer-Klasse 129  
 transform-Methode 129  
 Transponierte 43  
 Trefferquote 223  
 Trennbarkeit  
     lineare 56

**U**

Überanpassung 78, 93, 140, 505  
 Überwachtes Lernen 26  
 Unteranpassung 93, 505  
 Unüberwachtes Lernen 26, 353

**V**

Validierungsdatenmenge 205  
 Validierungskurve 214  
 Validierungsmenge 151  
 Variable  
     erklärende 28, 318  
     erklärte 318  
 Varianz 93, 94  
 Varianzreduktion 347  
 Vektorisierung 50  
 Verallgemeinerungsfehler 206  
 Verstärkendes Lernen 26

Vertrauenswahrscheinlichkeit 36  
 Vokabular 270  
 Vorkommenshäufigkeit 271, 272  
 Vorverarbeitung 35, 138  
 Vorwärtspropagation 390  
 Vorzeichenfunktion 235

**W**

Wahrheitsmatrix 220  
 Wahrscheinlichkeit  
     bedingte 82  
 Ward Linkage 368  
 Webanwendung 298  
 Wein-Datensammlung 136  
 while-Schleife 484  
 Whitespace 277  
 Widrow, Bernard 56  
 Wisconsin-Brustkrebs-Datensammlung 202  
 word2vec 285  
 Wortrelevanz 272  
 Wortsequenz 541  
 WTForms-Bibliothek 300

**X**

Xavier-Initialisierung 442, 462

**Z**

Zeitstempel 296  
 Zentroid 354  
 Zero-Padding 492  
 Zielfunktion 57  
 Zielvariable 28, 318  
 Zufallsgenerator 67  
 Zugehörigkeitsgrad 361  
 Zuordnungsfunktion 103



Nathan Marz  
mit James Warren

# Big Data

**Entwicklung und  
Programmierung von  
Systemen für große  
Datenmengen und Einsatz  
der Lambda-Architektur**

**Einführung in Big-Data-Systeme und  
-Technologien**

**Große Datenmengen speichern und ver-  
arbeiten**

**Einsatz zahlreicher Tools wie Hadoop,  
Apache Cassandra, Apache Storm uvm.**

Daten müssen mittlerweile von den meisten Unternehmen in irgendeiner Form verarbeitet werden. Dabei können sehr schnell so große Datenmengen entstehen, dass herkömmliche Datenbanksysteme nicht mehr ausreichen. Big-Data-Systeme erfordern Architekturen, die in der Lage sind, Datenmengen nahezu beliebigen Umfangs zu speichern und zu verarbeiten. Dies bringt grundlegende Anforderungen mit sich, mit denen viele Entwickler noch nicht vertraut sind.

Die Autoren erläutern die Einrichtung solcher Datenhaltungssysteme anhand eines speziell für große Datenmengen ausgelegten Frameworks: der Lambda-Architektur. Hierbei handelt



es sich um einen skalierbaren, leicht verständlichen Ansatz, der auch von kleinen Teams implementiert und langfristig betrieben werden kann.

Die Grundlagen von Big-Data-Systemen werden anhand eines realistischen Beispiels praktisch umgesetzt. In diesem Kontext lernen Sie neben einem allgemeinen Framework zur Verarbeitung großer Datenmengen auch Technologien wie Hadoop, Storm und NoSQL-Datenbanken kennen.

Dieses Buch setzt keinerlei Vorkenntnisse über Tools zur Datenanalyse oder NoSQL voraus, grundlegende Erfahrungen im Umgang mit herkömmlichen Datenbanken sind aber durchaus hilfreich.





FOSTER PROVOST | TOM FAWCETT

Foster Provost | Tom Fawcett

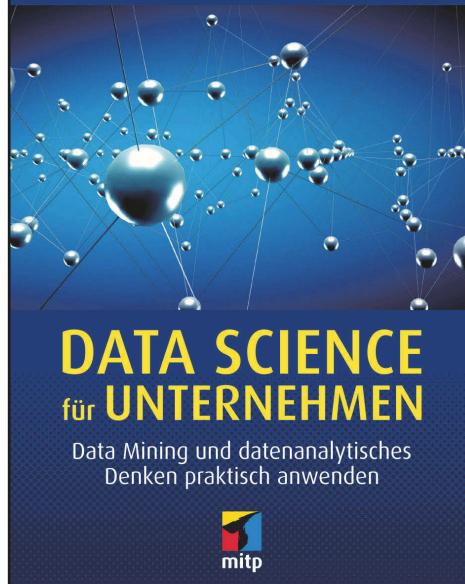
# Data Science für Unternehmen

**Data Mining und datenanalytisches Denken praktisch anwenden**

**Die grundlegenden Konzepte der Data Science verstehen, Wissen aus Daten ziehen und für Vorhersagen und Entscheidungen nutzen**

**Die wichtigsten Data-Mining-Verfahren gezielt und gewinnbringend einsetzen**

**Zahlreiche Praxisbeispiele zur Veranschaulichung**



Die anerkannten Data-Science-Experten Foster Provost und Tom Fawcett stellen in diesem Buch die grundlegenden Konzepte der Data Science vor, die für den effektiven Einsatz im Unternehmen von Bedeutung sind.

Sie erläutern das datenanalytische Denken, das erforderlich ist, damit Sie aus Ihren gesammelten Daten nützliches Wissen und geschäftlichen Nutzen ziehen können. Sie erfahren detailliert, welche Methoden der Data Science zu hilfreichen Erkenntnissen führen, so dass auf dieser Grundlage wichtige Entscheidungsfindungen unterstützt werden können.

Dieser Leitfaden hilft Ihnen dabei, die vielen zurzeit gebräuchlichen Data-Mining-Verfahren zu verstehen und gezielt und gewinnbringend anzuwenden. Sie lernen u.a., wie Sie:

- Data Science in Ihrem Unternehmen nutzen und damit Wettbewerbsvorteile erzielen
- Daten als ein strategisches Gut behandeln, in das investiert werden muss, um echten Nutzen daraus zu ziehen
- Geschäftliche Aufgaben datenanalytisch angehen und den Data-Mining-Prozess nutzen, um auf effiziente Weise sinnvolle Daten zu sammeln

Das Buch beruht auf einem Kurs für Betriebswirtschaftler, den Provost seit rund zehn Jahren an der New York University unterrichtet, und nutzt viele Beispiele aus der Praxis, um die Konzepte zu veranschaulichen.

Das Buch richtet sich an Führungskräfte und Projektmanager, die Data-Science-orientierte Projekte managen, an Entwickler, die Data-Science-Lösungen implementieren sowie an alle angehenden Data Scientists und Studenten.

**ISBN 978-3-95845-546-7**

Probekapitel und Infos erhalten Sie unter:  
**www.mitp.de/546**





Thomas Kaffka

# Neuronale Netze

## Grundlagen

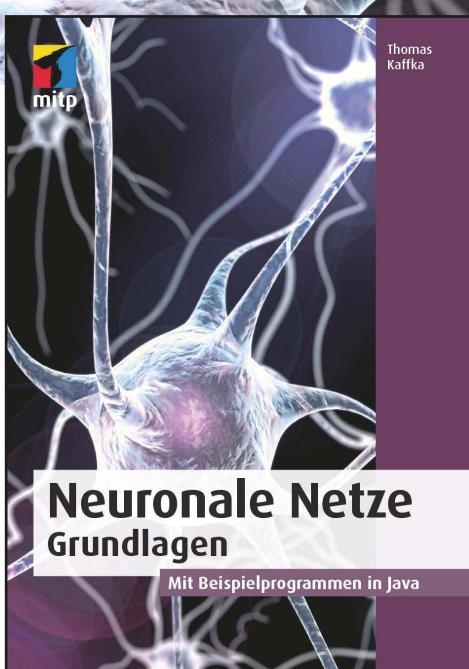
### Mit Beispielprogrammen in Java

Von den ersten Modellen bis zum Backpropagation-Netz

Allgemeinverständliche Erläuterungen mit vielen Praxis- und Anwendungsbeispielen

Zahlreiche Programme zum Ausprobieren, Ausführen und Trainieren Neuronaler Netze mit Beispieldaten

Für Programmierer: Vollständige Programmierung eines Backpropagation-Netzes zur Passworterkennung (in Java)



Dieses Buch ist eine grundlegende Einführung in die Entwicklung und Funktionsweise Neuronaler Netze. Sie lernen verschiedene Modelle kennen sowie alle Elemente, die für die Funktionalität Neuronaler Netze von Bedeutung sind. So werden Sie im Detail verstehen, wie diese arbeiten.

Praxisansatz des Buches:

- Alle vorgestellten Netze werden beispielhaft anschaulich durchgerechnet. So können Sie nachvollziehen, wie ein Neuronales Netz funktioniert und arbeitet.
- Außerdem liefert der Autor zusätzlich zum Buch selbst erstellte Programme, mit denen Sie am PC experimentieren können, indem Sie Beispieldaten eingeben und die jeweiligen Eigenschaften der unterschiedlichen Netze praktisch ausprobieren sowie diese trainieren und ausführen können.

Kaffka beschreibt zunächst die frühesten Modelle Neuronaler Netze sowie die Hebb'sche Formel und das von Rosenblatt entwickelte Modell des Perzeptrons. Daraufhin geht er auf die Mustererkennung mit einem Hopfield-Netz ein und erläutert die Grundlagen eines einfachen und eines bidirektionalen Assoziativspeichers.

Schließlich behandelt Kaffka das aktuelle Modell des Backpropagation-Netzes. Sie lernen im Detail, wie ein solches Neuronales Netz funktioniert – von der Netztopologie über die Transferfunktion bis zur Lernformel zum Trainieren eines Netzes.

Darauf aufbauend stellt der Autor verschiedene Beispiele und Anwendungen für Neuronale Netze vor. Hier diskutiert er zusätzlich, wie diese im Rahmen der Regressionsanalyse eingesetzt werden können. Zudem wird gezeigt, dass Neuronale Netze auch drei- oder mehrdimensionale Funktionen erlernen können.

Ein Ausblick zu Expertensystemen im Vergleich zu Neuronalen Netzen rundet die Einführung ab.

Zusatznutzen für Programmierer

- Programmierer, die selber ein neuronales Netz programmieren wollen, finden ein ausführliches Kapitel, in dem ein Backpropagation-Netz vollständig in Java programmiert wird.
- Für Programmierer wird der Java-Code aller im Buch verwendeten Programme erläutert.





Jerry Kaplan

# Künstliche Intelligenz

## Eine Einführung



**Die Künstliche Intelligenz verändert unsere Welt in vielen Bereichen: Lebensweise, Arbeit, Gesellschaft und sogar unser Platz im Universum werden neu definiert.**

Jerry Kaplan diskutiert in diesem Buch die wichtigsten gesellschaftlichen, rechtlichen und wirtschaftlichen Aspekte für die gegenwärtige und zukünftige Bedeutung der Künstlichen Intelligenz. Dabei behandelt er grundlegende Fragen wie u.a.:

*Werden Maschinen eines Tages klüger sein als der Mensch?*

*Wie wirken sich lernende, flexible Roboter auf den Arbeitsmarkt aus?*

*Kann man eine Maschine für ihre Handlungen verantwortlich machen?*

Der Autor macht deutlich, inwiefern Fortschritte im Hinblick auf die intellektuellen und physischen Fähigkeiten von Maschinen unsere Gesellschaft grundlegend verändern werden. Dabei zeigt er auf, dass diese kontinuierliche Weiterentwicklung von Maschinen eine immer wichtigere Rolle spielen und mit vielen Bereichen unseres täglichen Lebens untrennbar verbunden sein wird.

Dieses Buch ist eine kompakte und leicht zugängliche Einführung in das Thema. Jerry Kaplan veranschaulicht mögliche künftige Auswirkungen dieser bedeutenden Entwicklung und lässt dabei technologische Details außen vor. So ist das Buch gut geeignet für die Auseinandersetzung mit den grundlegenden Fragen zur Künstlichen Intelligenz.

ISBN 978-3-95845-632-7

Probekapitel und Infos erhalten Sie unter:  
[www.mitp.de/632](http://www.mitp.de/632)



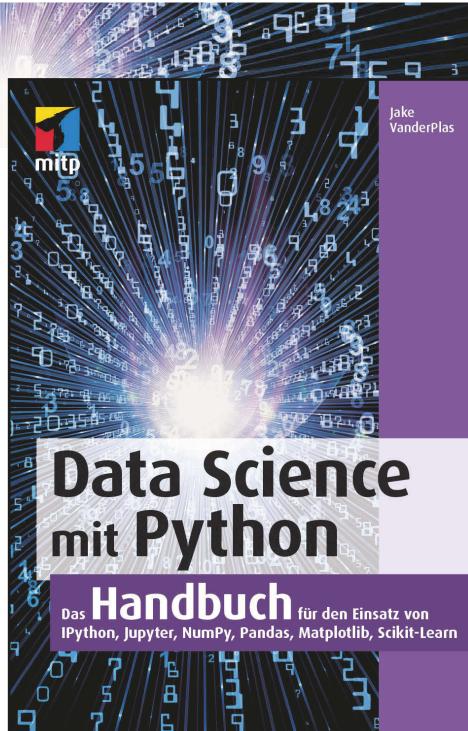


Jake VanderPlas

# Data Science mit Python

Das Handbuch für den Einsatz von IPython, Jupyter, NumPy, Pandas, Matplotlib, Scikit-Learn

Die wichtigsten Tools für die Datenanalyse und -bearbeitung im praktischen Einsatz  
Python effizient für datenintensive Berechnungen einsetzen mit IPython und Jupyter  
Laden, Speichern und Bearbeiten von Daten und numerischen Arrays mit NumPy und Pandas  
Visualisierung von Daten mit Matplotlib



Python ist für viele die erste Wahl für Data Science, weil eine Vielzahl von Ressourcen und Bibliotheken zum Speichern, Bearbeiten und Auswerten von Daten verfügbar ist. In diesem Buch erläutert der Autor den Einsatz der wichtigsten Tools.

Für Datenanalytiker und Wissenschaftler ist dieses umfassende Handbuch von unschätzbarem Wert für jede Art von Berechnung mit Python sowie bei der Erledigung alltäglicher Aufgaben. Dazu gehören das Bearbeiten, Umwandeln und Bereinigen von Daten, die Visualisierung verschiedener Datentypen und die Nutzung von Daten zum Erstellen von Statistiken oder Machine-Learning-Modellen.

**Dieses Handbuch erläutert die Verwendung der folgenden Tools:**

- IPython und Jupyter für datenintensive Berechnungen
- NumPy und Pandas zum effizienten Speichern und Bearbeiten von Daten und Datenarrays in Python
- Matplotlib für vielfältige Möglichkeiten der Visualisierung von Daten
- Scikit-Learn zur effizienten und sauberen Implementierung der wichtigsten und am meisten verbreiteten Algorithmen des Machine Learnings

Der Autor zeigt Ihnen, wie Sie die zum Betreiben von Data Science verfügbaren Pakete nutzen, um Daten effektiv zu speichern, zu handhaben und Einblick in diese Daten zu gewinnen. Grundlegende Kenntnisse in Python werden dabei vorausgesetzt.

ISBN 978-3-95845-695-2

Probekapitel und Infos erhalten Sie unter:  
[www.mitp.de/695](http://www.mitp.de/695)



```
get export objects  
obj_export_list = viewport_s  
f self use selection setting
```

Michael Weigend

# Python 3

Lernen und professionell  
anwenden  
Das umfassende Praxisbuch

7., erweiterte Auflage

Klassen, Objekte und Vererbung,  
Dictionaries, XML, Datenbanken und  
Internet-Programmierung

Benutzungsoberflächen und  
Multimediaanwendungen mit PyQt

Wissenschaftliches Rechnen mit NumPy  
und parallele Verarbeitung großer  
Datenmengen

Übungen mit Musterlösungen zu  
jedem Kapitel

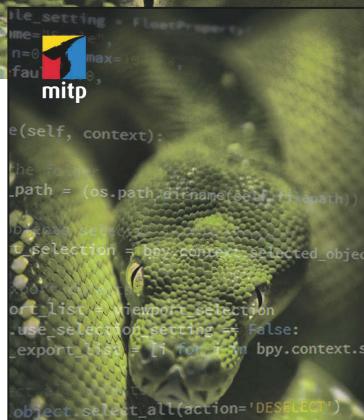
Die Skriptsprache Python mit ihrer einfachen Syntax ist hervorragend geeignet, um modernes Programmieren zu lernen. Mit diesem Buch erhalten Sie einen umfassenden Einblick in Python 3. Michael Weigend behandelt Python von Grund auf und erläutert die wesentlichen Sprachelemente. Er geht dabei besonders auf die Anwendung von Konzepten der objektorientierten Programmierung ein.

Insgesamt liegt der Schwerpunkt auf der praktischen Arbeit mit Python. Ziel ist es, die wesentlichen Techniken und dahinter stehenden Ideen anhand zahlreicher anschaulicher Beispiele verständlich zu machen. Zu typischen Problemstellungen werden Schritt für Schritt Lösungen erarbeitet. So erlernen Sie praxisorientiert die Programmentwicklung mit Python und die Anwendung von Konzepten der objektorientierten Programmierung.

Alle Kapitel enden mit einfachen und komplexen Übungsaufgaben mit vollständigen Musterlösungen.

Das Buch behandelt die Grundlagen von Python 3 (Version 3.5) und zusätzlich auch weiterführende Themen wie die Gestaltung grafischer Benutzungsoberflächen mit tkinter und PyQt, Threads und Multiprocessing, CGI- und Internetprogrammierung, automatisiertes Testen, Datenmodellierung mit XML, Datenbanken und wissenschaftliches Rechnen mit NumPy.

Der Autor wendet sich sowohl an ambitionierte Einsteiger als auch an Leser, die bereits mit einer höheren Programmiersprache vertraut sind. Zugleich bietet sich dieses Lehrbuch als Textgrundlage oder nützliche Ergänzung zu Universitätskursen an.



Michael  
Weigend  
7., erweiterte  
Auflage



ISBN 978-3-95845-791-1

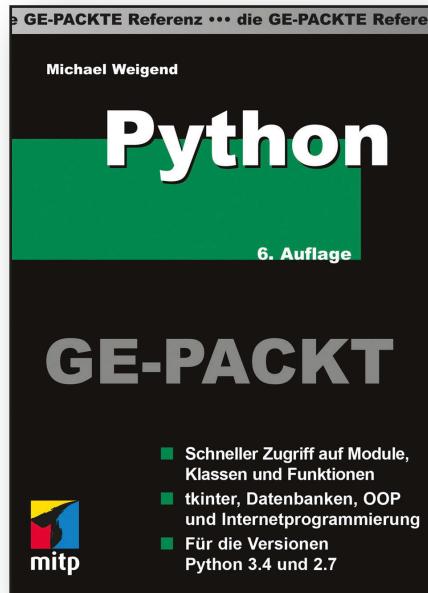
Probekapitel und Infos erhalten Sie unter:  
[www.mitp.de/791](http://www.mitp.de/791)



**Michael Weigend**

# Python GE-PACKT

- Schneller Zugriff auf Module, Klassen und Funktionen
- tkinter, Datenbanken, OOP und Internetprogrammierung
- Für die Versionen Python 3.4 und 2.7



6. Auflage

Mit dieser Referenz erhalten Sie effiziente Unterstützung bei der Programmierung mit Python 3.4 und Python 2.7 – klar strukturiert zum Nachschlagen. In 24 thematisch gegliederten Kapiteln werden die wichtigsten Module detailliert und praxisbezogen erläutert: angefangen bei grundlegenden Elementen wie Datentypen, Operatoren und Standardfunktionen bis hin zu Spezialthemen wie der Schnittstelle zum Laufzeit- und Betriebssystem, Generatoren, GUI-Programmierung mit tkinter, PIL, Logging, Mengenverarbeitung, XML und Dezimalarithmetik. Darüber hinaus finden Sie kompakte Darstellungen der Umsetzung von objektorientierter Programmierung, CGI- und Internetprogrammierung (E-Mail, FTP, Telnet, HTTP) sowie der Datenbankanbindung (MySQL, SQLite).

Die Erläuterungen werden ergänzt durch übersichtliche Tabellen, UML-Diagramme und zahlreiche leicht nachvollziehbare Beispiele, die Anregungen und Lösungen für eigene Programmieraufgaben liefern.

Probekapitel und Infos erhalten Sie unter:  
[www.mitp.de/8726](http://www.mitp.de/8726)

**ISBN 978-3-8266-8726-6**