



# Einführung in Machine Learning mit Python

---

PRAXISWISSEN DATA SCIENCE

Andreas C. Müller & Sarah Guido  
Übersetzung von Kristian Rother

klaus.engel@t-systems.com  
plus\_8ac69b7940c88abd1664-20482-12825-1

Papier  
plus<sup>+</sup>  
PDF.

Zu diesem Buch – sowie zu vielen weiteren O'Reilly-Büchern – können Sie auch das entsprechende E-Book im PDF-Format herunterladen. Werden Sie dazu einfach Mitglied bei oreilly.plus<sup>+</sup>: [www.oreilly.plus](http://www.oreilly.plus)

---

# Einführung in Machine Learning mit Python

*Praxiswissen Data Science*

*Andreas C. Müller und Sarah Guido*

*Deutsche Übersetzung  
von Kristian Rother*

O'REILLY®

Andreas C. Müller und Sarah Guido

Übersetzung: Kristian Rother

Lektorat: Alexandra Follenius

Korrektorat: Claudia Lötschert, [www.richtiger-text.de](http://www.richtiger-text.de)

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

Herstellung: Susanne Bröckelmann

Umschlaggestaltung: Michael Oréal, [www.oreal.de](http://www.oreal.de)

Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

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:

Print 978-3-96009-049-6

PDF 978-3-96010-111-6

ePub 978-3-96010-112-3

mobi 978-3-96010-113-0

Dieses Buch erscheint in Kooperation mit O'Reilly Media, Inc. unter dem Imprint »O'REILLY«.

O'REILLY ist ein Markenzeichen und eine eingetragene Marke von O'Reilly Media, Inc. und wird mit Einwilligung des Eigentümers verwendet.

1. Auflage 2017

Copyright © 2017 dpunkt.verlag GmbH

Wieblinger Weg 17

69123 Heidelberg

Authorized German translation of the English edition of *Introduction to Machine Learning with Python*, ISBN 978-1-4493-6941-5 © 2017 Sarah Guido, Andreas Müller. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.

Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.

Die Informationen in diesem Buch wurden mit größter Sorgfalt erarbeitet. Dennoch können Fehler nicht vollständig ausgeschlossen werden. Verlag, Autoren und Übersetzer übernehmen keine juristische Verantwortung oder irgendeine Haftung für eventuell verbliebene Fehler und deren Folgen.

5 4 3 2 1 0

---

# Inhalt

<b>Vorwort</b> .....	<b>IX</b>
<b>1 Einführung</b> .....	<b>1</b>
Warum Machine Learning? .....	1
Welche Probleme kann Machine Learning lösen? .....	2
Ihre Aufgabe und Ihre Daten kennen .....	5
Warum Python? .....	5
scikit-learn .....	6
Installieren von scikit-learn .....	6
Grundlegende Bibliotheken und Werkzeuge .....	7
Jupyter Notebook .....	8
NumPy .....	8
SciPy .....	8
matplotlib .....	10
pandas .....	10
mglearn .....	11
Python 2 versus Python 3 .....	12
In diesem Buch verwendete Versionen .....	13
Eine erste Anwendung: Klassifizieren von Iris-Spezies .....	14
Die Daten kennenlernen .....	15
Erfolg nachweisen: Trainings- und Testdaten .....	17
Das Wichtigste zuerst: Sichten Sie Ihre Daten .....	19
Ihr erstes Modell konstruieren: k-nächste-Nachbarn .....	21
Vorhersagen treffen .....	22
Evaluieren des Modells .....	23
Zusammenfassung und Ausblick .....	23

<b>2 Überwachtes Lernen . . . . .</b>	<b>27</b>
Klassifikation und Regression . . . . .	27
Verallgemeinerung, Overfitting und Underfitting . . . . .	28
Zusammenhang zwischen Modellkomplexität und Größe des Datensatzes . . . . .	31
Algorithmen zum überwachten Lernen . . . . .	32
Einige Beispieldatensätze . . . . .	32
k-nächste-Nachbarn . . . . .	36
Lineare Modelle . . . . .	45
Naive Bayes-Klassifikatoren . . . . .	66
Entscheidungsbäume . . . . .	68
Ensembles von Entscheidungsbäumen . . . . .	80
Support Vector Machines mit Kernel . . . . .	88
Neuronale Netze (Deep Learning) . . . . .	99
Schätzungen der Unsicherheit von Klassifikatoren . . . . .	112
Die Entscheidungsfunktion . . . . .	113
Vorhersagen von Wahrscheinlichkeiten . . . . .	116
Unsicherheit bei der Klassifikation mehrerer Kategorien . . . . .	118
Zusammenfassung und Ausblick . . . . .	120
<b>3 Unüberwachtes Lernen und Vorverarbeitung . . . . .</b>	<b>123</b>
Arten von unüberwachtem Lernen . . . . .	123
Herausforderungen beim unüberwachten Lernen . . . . .	124
Vorverarbeiten und Skalieren . . . . .	124
Unterschiedliche Möglichkeiten der Vorverarbeitung . . . . .	125
Anwenden von Datentransformationen . . . . .	126
Trainings- und Testdaten in gleicher Weise skalieren . . . . .	128
Die Auswirkungen der Vorverarbeitung auf überwachtes Lernen . . . . .	130
Dimensionsreduktion, Extraktion von Merkmalen und Manifold Learning . . . . .	132
Hauptkomponentenzerlegung (PCA) . . . . .	132
Nicht-negative-Matrix-Faktorisierung (NMF) . . . . .	147
Manifold Learning mit t-SNE . . . . .	154
Clusteranalyse . . . . .	158
k-Means-Clustering . . . . .	158
Agglomeratives Clustering . . . . .	169
DBSCAN . . . . .	174
Vergleichen und Auswerten von Clusteralgorithmen . . . . .	178
Zusammenfassung der Clustering-Methoden . . . . .	192
Zusammenfassung und Ausblick . . . . .	193

<b>4 Repräsentation von Daten und Merkmalsgenerierung</b>	<b>195</b>
Kategorische Variablen	196
One-Hot-Kodierung (Dummy-Variablen)	197
Zahlen können kategorische Daten kodieren	202
Binning, Diskretisierung, lineare Modelle und Bäume	204
Interaktionen und Polynome	208
Univariate nichtlineare Transformation	214
Automatische Auswahl von Merkmalen	218
Univariate Statistiken	218
Modellbasierte Auswahl von Merkmalen	221
Iterative Auswahl von Merkmalen	222
Berücksichtigen von Expertenwissen	224
Zusammenfassung und Ausblick	233
<b>5 Evaluierung und Verbesserung von Modellen</b>	<b>235</b>
Kreuzvalidierung	236
Kreuzvalidierung in scikit-learn	237
Vorteile der Kreuzvalidierung	238
Stratifizierte k-fache Kreuzvalidierung und andere Strategien	238
Gittersuche	244
Einfache Gittersuche	245
Die Gefahr des Overfittings von Parametern und Validierungsdaten	246
Gittersuche mit Kreuzvalidierung	248
Evaluationsmetriken	260
Das Ziel im Auge behalten	260
Metriken zur binären Klassifikation	261
Metriken zur Klassifikation mehrerer Kategorien	282
Regressionsmetriken	284
Verwenden von Metriken zur Modellauswahl	285
Zusammenfassung und Ausblick	287
<b>6 Verkettete Algorithmen und Pipelines</b>	<b>289</b>
Parameterauswahl mit Vorverarbeitung	290
Erstellen von Pipelines	292
Pipelines zur Gittersuche einsetzen	293
Die allgemeine Pipeline-Schnittstelle	296
Bequemes Erstellen von Pipelines mit <code>make_pipeline</code>	297
Zugriff auf Attribute von Schritten	298
Zugriff auf Attribute in einer Pipeline mit Gittersuche	299

Gittersuche für Vorverarbeitungsschritte und Modellparameter . . . . .	300
Gittersuche nach dem richtigen Modell . . . . .	303
Zusammenfassung und Ausblick . . . . .	304
<b>7 Verarbeiten von Textdaten . . . . .</b>	<b>307</b>
Arten von als Strings repräsentierter Daten . . . . .	307
Anwendungsbeispiel: Meinungsanalyse zu Filmbewertungen . . . . .	309
Repräsentation von Text als Bag-of-Words . . . . .	311
Anwenden von Bag-of-Words auf einen einfachen Datensatz . . . . .	313
Bag-of-Words der Filmbewertungen . . . . .	314
Stoppwörter . . . . .	318
Umskalieren der Daten mit tf-idf . . . . .	319
Untersuchen der Koeffizienten des Modells . . . . .	322
Bag-of-Words mit mehr als einem Wort (n-Gramme) . . . . .	323
Fortgeschrittene Tokenisierung, Stemming und Lemmatisierung . . . . .	327
Modellierung von Themen und Clustering von Dokumenten . . . . .	331
Latent Dirichlet Allocation . . . . .	331
Zusammenfassung und Ausblick . . . . .	338
<b>8 Zusammenfassung und weiterführende Ressourcen . . . . .</b>	<b>341</b>
Herangehensweise an eine Fragestellung beim maschinellen Lernen . . . . .	341
Der menschliche Faktor . . . . .	342
Vom Prototyp zum Produktivsystem . . . . .	343
Testen von Produktivsystemen . . . . .	344
Konstruieren eines eigenen Estimators . . . . .	344
Wie geht es von hier aus weiter? . . . . .	345
Theorie . . . . .	345
Andere Umgebungen und Programmpakete zum maschinellen Lernen . . . . .	346
Ranking, Empfehlungssysteme und andere Arten von Lernen . . . . .	347
Probabilistische Modellierung, Inferenz und probabilistische Programmierung . . . . .	347
Neuronale Netze . . . . .	348
Skalieren auf größere Datensätze . . . . .	349
Verfeinern Sie Ihre Fähigkeiten . . . . .	350
Schlussbemerkung . . . . .	351
<b>Index . . . . .</b>	<b>353</b>

---

# Vorwort

Machine Learning ist heutzutage ein integraler Bestandteil vieler kommerzieller Anwendungen und Forschungsprojekte in so unterschiedlichen Gebieten wie medizinischer Diagnose und Behandlung oder dem Finden von Freunden in einem sozialen Netzwerk. Viele glauben, dass maschinelles Lernen nur von großen Firmen mit großen Forschungsabteilungen umgesetzt werden kann. In diesem Buch möchten wir Ihnen zeigen, wie einfach es ist, selbst maschinelle Lernsysteme zu bauen, und wie Sie dabei am besten vorgehen. Mit dem Wissen aus diesem Buch können Sie sich Ihr eigenes System zusammenbauen, um herauszufinden, wie Menschen sich auf Twitter fühlen, oder um Vorhersagen über die globale Erwärmung zu treffen. Die Anwendungen von maschinellem Lernen sind schier endlos und mit den heutzutage zur Verfügung stehenden Daten hauptsächlich durch Ihre eigene Vorstellungskraft begrenzt.

## Wer sollte dieses Buch lesen?

Dieses Buch ist für gegenwärtige und aufstrebende Anwender maschinellen Lernens geeignet, die reale Fragestellungen mithilfe von maschinellem Lernen beantworten möchten. Dies ist ein Einführungsbuch, das keinerlei Vorwissen über maschinelles Lernen oder künstliche Intelligenz (KI) voraussetzt. Wir werden unser Hauptaugenmerk auf die Verwendung von Python und der Bibliothek scikit-learn richten und sämtliche Schritte beim Entwickeln einer erfolgreichen Anwendung zum maschinellen Lernen abhandeln. Die dabei vorgestellten Methoden sind für Wissenschaftler, Forscher und an kommerziellen Anwendungen arbeitende Data Scientists nützlich. Den größten Nutzen wird dieses Buch für Sie haben, wenn Sie bereits ein wenig mit Python und den Bibliotheken NumPy und matplotlib vertraut sind.

Wir haben uns redlich bemüht, uns weniger um die Mathematik zu kümmern, sondern das Augenmerk vielmehr auf die praktischen Aspekte der Nutzung von Algorithmen zum maschinellen Lernen zu legen. Auch wenn Mathematik (insbesondere Wahrscheinlichkeitstheorie) die Grundlage maschinellen Lernens darstellt, werden

wir die Algorithmen nicht in aller Ausführlichkeit analysieren. Wenn Sie sich für die den maschinellen Lernalgorithmen zugrunde liegende Mathematik interessieren, empfehlen wir Ihnen das Buch *The Elements of Statistical Learning* (Springer) von Trevor Hastie, Robert Tibshirani und Jerome Friedman, das kostenlos auf der Website der Autoren (<http://statweb.stanford.edu/~tibs/ElemStatLearn/>) verfügbar ist. Ebenso werden wir nicht beschreiben, wie man Algorithmen für maschinelles Lernen von null auf implementiert. Stattdessen möchten wir Ihnen vor allem zeigen, wie Sie die zahlreichen bereits in scikit-learn und anderen Bibliotheken implementierten Methoden nutzen können.

## Warum wir dieses Buch geschrieben haben

Es gibt zahlreiche Bücher über Machine Learning und KI. Allerdings sind diese alle für Masterstudenten und Doktoranden der Informatik geschrieben und stecken voller fortgeschrittenen Mathematik. Das steht in deutlichem Gegensatz dazu, wie maschinelles Lernen tatsächlich verwendet wird, nämlich als bequemes Werkzeug in der Forschung und in kommerziellen Anwendungen. Heutzutage braucht man für Machine Learning keinen Doktortitel. Es gibt aber einige Ressourcen, die die wichtigsten Aspekte der Implementierung von maschinellem Lernen in der Praxis vollständig wiedergeben, ohne dass Sie einen Mathekurs für Fortgeschrittene besuchen müssen. Wir hoffen, dass dieses Buch beim Anwenden maschineller Lernmethoden hilft, ohne dass dazu jahrelange Lektüre in Analysis, linearer Algebra und Wahrscheinlichkeitstheorie notwendig wäre.

## Wegweiser durch dieses Buch

Dieses Buch ist etwa folgendermaßen organisiert:

- Kapitel 1 stellt grundlegende Konzepte und Anwendungen maschinellen Lernens vor und beschreibt die im gesamten Buch verwendete Arbeitsumgebung.
- Die Kapitel 2 und 3 beschreiben die am häufigsten in der Praxis verwendeten Algorithmen zum maschinellen Lernen und besprechen deren Stärken und Schwachstellen.
- Kapitel 4 diskutiert, warum die Repräsentation der von maschinellem Lernen verarbeiteten Daten wichtig ist und welche Aspekte der Daten man beachten sollte.
- Kapitel 5 befasst sich mit fortgeschrittenen Methoden zum Auswerten von Modellen und zur Optimierung von Parametern. Dabei liegt besonderes Augenmerk auf Kreuzvalidierung und Gittersuche.
- Kapitel 6 erklärt die Verwendung von Pipelines zum Verketten von Modellen und zur Kapselung von Arbeitsabläufen.

- Kapitel 7 zeigt, wie Sie die in den ersten Kapiteln beschriebenen Methoden auf Textdaten anwenden können, und macht Sie mit einigen speziellen Techniken zur Verarbeitung von Textdaten bekannt.
- Kapitel 8 gibt einen allgemeinen Überblick und verweist auf fortgeschrittene Themen.

Die eigentlichen Algorithmen werden in den Kapiteln 2 und 3 vorgestellt, auch wenn das Verständnis all dieser Algorithmen für einen Anfänger nicht unbedingt erforderlich ist. Wenn Sie ein maschinelles Lernsystem so schnell wie möglich konstruieren müssen, empfehlen wir Ihnen, mit Kapitel 1 und den ersten Abschnitten von Kapitel 2 zu beginnen, in denen die wichtigsten Begriffe vorgestellt werden. Sie können dann direkt zu Abschnitt »Zusammenfassung und Ausblick« auf Seite 120 in Kapitel 2 springen, wo Sie eine Liste aller besprochenen überwachten Modelle finden. Wählen Sie das für Ihre Zwecke am besten geeignete Modell aus und blättern Sie dann zum entsprechenden Abschnitt, der sich mit den Details auseinandersetzt. Danach können Sie die Techniken in Kapitel 5 anwenden, um Ihr Modell zu evaluieren und zu optimieren.

## Ressourcen im Netz

Beim Studium dieses Buches sollten Sie unbedingt die Webseite von scikit-learn (<http://scikit-learn.org>) besuchen, um die detaillierte Dokumentation der Klassen und Funktionen zu sichten und weitere Beispiele zu finden. Es gibt auch einen Videokurs von Andreas Müller, »Advanced Machine Learning with scikit-learn,« der dieses Buch begleitet. Sie können diesen auf [http://bit.ly/advanced\\_machine\\_learning\\_scikit-learn](http://bit.ly/advanced_machine_learning_scikit-learn) finden.

## In diesem Buch verwendete Konventionen

Die folgenden typografischen Konventionen werden in diesem Buch verwendet:

### Kursiv

Kennzeichnet sprachliche Hervorhebungen, neue Begriffe, URLs, E-Mail-Adressen, Dateinamen und Dateiendungen.

### Feste Zeichenbreite

Wird für Programm listings und für Programmelemente wie Namen von Variablen, Funktionen, Datenbanken, Datentypen, Umgebungsvariablen, Anweisungen und Schlüsselwörter verwendet. Auch Befehle sowie Namen von Modulen und Paketen verwenden diese Notation.

### Konstante Zeichenbreite, fett

Kennzeichnet Befehle oder anderen Text, den der Nutzer wörtlich eingeben sollte.

### *Konstante Zeichenbreite, kursiv*

Kennzeichnet Text, den der Nutzer je nach Kontext durch entsprechende Werte ersetzen sollte.



Dieses Zeichen steht für einen Tipp oder eine Empfehlung.



Dieses Zeichen steht für einen allgemeinen Hinweis.



Dieses Piktogramm steht für eine Warnung oder erhöhte Aufmerksamkeit.

## Verwenden von Codebeispielen

Zusätzliche Materialien (Codebeispiele, IPython Notebooks usw.) stehen unter [https://github.com/amueller/introduction\\_to\\_ml\\_with\\_python](https://github.com/amueller/introduction_to_ml_with_python) zum Download bereit.

Dieses Buch ist dazu da, Ihnen bei Ihren Aufgaben zu helfen. Im Allgemeinen dürfen Sie die mit diesem Buch bereitgestellten Codebeispiele in Ihren Programmen und der dazugehörigen Dokumentation nutzen. Sie müssen uns dazu nicht um Erlaubnis fragen, es sei denn, Sie reproduzieren einen signifikanten Teil des Codes. Beispielsweise benötigen Sie keine Erlaubnis, um ein Programm zu schreiben, das mehrere Codeabschnitte aus diesem Buch enthält. Eine CD-ROM mit Beispielen aus Büchern von O'Reilly zu verkaufen oder zu verteilen, benötigt dagegen eine Erlaubnis. Eine Frage mit einem Zitat aus diesem Buch unter Angabe eines Codebeispiels zu beantworten, benötigt keine Erlaubnis. Einen wesentlichen Anteil der Codebeispiele aus diesem Buch in Ihr eigenes Buch zu integrieren, benötigt dagegen eine Erlaubnis.

Wir freuen uns über Zitate, verlangen diese aber nicht. Ein Zitat enthält Titel, Autor, Verlag und ISBN. Beispiel: »*Einführung in Machine Learning mit Python* von Andreas C. Müller and Sarah Guido, O'Reilly 2017, ISBN 978-3-96009-049-6.«

Wenn Sie glauben, dass Ihre Verwendung von Codebeispielen über gewöhnliche Nutzung hinausgeht oder außerhalb der oben vorgestellten Nutzungsbedingungen liegt, kontaktieren Sie uns bitte unter [kommentar@oreilly.de](mailto:kommentar@oreilly.de).

# Danksagungen

## Von Andreas

Ohne die Hilfe und Unterstützung einer großen Gruppe von Menschen wäre dieses Buch nie entstanden.

Ich möchte den Lektoren Meghan Blanchette, Brian MacDonald und besonders Dawn Schanafelt dafür danken, dass sie Sarah und mir bei der Verwirklichung dieses Buches geholfen haben.

Ich möchte meinen Reviewern Thomas Caswell, Olivier Grisel, Stefan van der Walt und John Myles White danken, die sich die Zeit zum Lesen früher Versionen des Buches genommen und mir wertvolles Feedback gegeben haben – und außerdem zu den Eckpfeilern im wissenschaftlichen Open Source-Ökosystem gehören.

Ich bin der herzlichen wissenschaftlichen Open Source Python-Community auf ewig dankbar, besonders den Mitwirkenden an scikit-learn. Ohne die Unterstützung und Hilfe dieser Gemeinde, insbesondere von Gael Varoquaux, Alex Gramfort und Olivier Grisel, wäre ich niemals ein Kernalentwickler von scikit-learn geworden oder hätte dieses Paket nie so durchgehend verstanden wie heute. Ich danke auch allen anderen Mitwirkenden, die ihre Zeit in die Verbesserung und Pflege dieses Pakets investieren.

Ich bin außerdem für die vielen Diskussionen mit zahlreichen meiner Kollegen dankbar, durch die ich die Herausforderungen beim maschinellen Lernen verstehen und Ideen für die Struktur eines Grundlagenbuches sammeln konnte. Unter den Leuten, mit denen ich mich über Machine Learning unterhalten habe, möchte ich Brian McFee, Daniela Huttenkopp, Joel Nothman, Gilles Louppe, Hugo Bowne-Anderson, Sven Kreis, Alice Zheng, Kyunghyun Cho, Pablo Baber und Dan Cervone besonders danken.

Ich danke auch Rachel Rakov, die eine unermüdliche Betatesterin und Korrekturleserin der frühen Versionen dieses Buches war und mir bei der Ausarbeitung auf vielerlei Weise geholfen hat.

Danken möchte ich meinen Eltern, Harald und Margot, und meiner Schwester Miriam für ihre anhaltende Unterstützung und Ermutigung. Ich möchte auch den vielen Menschen in meinem Leben danken, deren Liebe und Freundschaft mir die Kraft und den Beistand für diesesfordernde Unterfangen gaben.

## Von Sarah

Ich möchte Meg Blanchette danken, ohne deren Hilfe und Beistand dieses Projekt nicht einmal existieren würde. Ich danke Celia La und Brian Carlson für das Lesen der frühen Entwürfe. Ich danke den Leuten von O'Reilly für ihre unendliche Geduld. Schließlich danke ich DTS für deine ewige und endlose Unterstützung.



# KAPITEL 1

# Einführung

Beim Machine Learning geht es darum, Wissen aus Daten zu extrahieren. Es handelt sich dabei um ein Forschungsfeld in der Schnittmenge von Statistik, künstlicher Intelligenz und Informatik und ist ebenfalls als prädiktive Analytik oder statistisches Lernen bekannt. Die Anwendung von Methoden maschinellen Lernens sind in den letzten Jahren Teil unseres Alltags geworden. Von automatischen Empfehlungen für Filme, Nahrungsmittel oder andere Produkte über personalisiertes Online-Radio bis zur Erkennung von Freunden in Ihren Fotos enthalten viele moderne Webseiten und Geräte als Herzstück Algorithmen für maschinelles Lernen. Wenn Sie sich eine komplexe Webseite wie Facebook, Amazon oder Netflix ansehen, ist es sehr wahrscheinlich, dass jeder Teil der Seite mehrere maschinelle Lernmodelle enthält.

Außerhalb kommerzieller Anwendungen hat Machine Learning einen immensen Einfluss auf die heutige Methodik datengetriebener Forschung gehabt. Die in diesem Buch vorgestellten Werkzeuge sind auf so unterschiedliche wissenschaftliche Fragestellungen angewandt worden wie das Verstehen von Sternen, das Finden weit entfernter Planeten, das Entdecken neuer Elementarteilchen, die Analyse von DNA-Sequenzen und die personalisierte Krebsbehandlung.

Um von maschinellem Lernen zu profitieren, muss Ihre Anwendung dabei gar nicht so gewaltig oder weltverändernd sein wie diese Beispiele. In diesem Kapitel werden wir erklären, warum Machine Learning so beliebt geworden ist, und werden erörtern, was für Fragestellungen damit beantwortet werden können. Anschließend werden wir Ihnen zeigen, wie Sie Ihr erstes Modell mithilfe von maschinellem Lernen bauen können, und dabei wichtige Begriffe vorstellen.

## Warum Machine Learning?

In den ersten Tagen »intelligenter« Anwendungen verwendeten viele Systeme von Hand kodierte Regeln in Form von »if/else«-Entscheidungen, um Daten zu verarbeiten oder Benutzereingaben anzupassen. Stellen Sie sich einen Spam-Filter vor,

dessen Aufgabe es ist, eingehende Nachrichten in den Spam-Ordner zu verschieben. Sie könnten eine schwarze Liste von Wörtern erstellen, die zum Einstufen einer E-Mail als Spam führen. Dies ist ein Beispiel für ein von Experten entwickeltes Regelsystem als »intelligente« Anwendung. Bei manchen Anwendungen ist das Festlegen von Regeln von Hand praktikabel, besonders wenn Menschen ein gutes Verständnis für den zu modellierenden Prozess besitzen. Allerdings hat das Verwenden von Hand erstellter Regeln zwei große Nachteile:

- Die Entscheidungslogik ist für eine bestimmtes Fachgebiet und eine Aufgabe spezifisch. Selbst eine kleine Veränderung der Aufgabe kann dazu führen, dass das gesamte System neu geschrieben werden muss.
- Das Entwickeln von Regeln erfordert ein tiefes Verständnis davon, wie ein menschlicher Experte diese Entscheidung treffen würde.

Ein Beispiel, bei dem der händische Ansatz fehlschlägt, ist das Erkennen von Gesichtern in Bildern. Heutzutage kann jedes Smartphone ein Gesicht in einem Bild erkennen. Allerdings war Gesichtserkennung bis 2001 ein ungelöstes Problem. Das Hauptproblem dabei war, dass ein Computer die Pixel (aus denen ein Computerbild besteht) im Vergleich zu Menschen auf sehr unterschiedliche Weise »wahrnimmt«. Diese unterschiedliche Repräsentation macht es einem Menschen praktisch unmöglich, ein gutes Regelwerk zu entwickeln, mit dem sich umschreiben lässt, was ein Gesicht in einem digitalen Bild ausmacht. Mit maschinellem Lernen ist es dagegen ausreichend, einem Programm eine große Sammlung von Bildern mit Gesichtern vorzulegen, um die Charakteristiken zum Erkennen von Gesichtern auszuarbeiten.

## Welche Probleme kann Machine Learning lösen?

Die erfolgreichsten Arten maschineller Lernalgorithmen sind diejenigen, die den Entscheidungsprozess durch Verallgemeinerung aus bekannten Beispielen automatisieren. In diesem als *überwachtes Lernen* bekannten Szenario beliefert der Nutzer einen Algorithmus mit Paaren von Eingabewerten und erwünschten Ausgabewerten, und der Algorithmus findet heraus, wie sich die gewünschte Ausgabe erstellen lässt. Damit ist der Algorithmus ohne menschliche Hilfe in der Lage, aus zuvor unbekannten Eingaben eine Ausgabe zu berechnen. Bei unserem Beispiel der Spam-Klassifizierung würde der Nutzer dem Algorithmus eine große Anzahl E-Mails (die Eingaben) sowie die Angabe, welche dieser E-Mails Spam sind (die erwünschte Ausgabe), zur Verfügung stellen. Für eine neue E-Mail kann der Algorithmus dann vorhersagen, ob die neue E-Mail Spam ist.

Maschinelle Lernalgorithmen, die aus Eingabe-Ausgabe-Paaren lernen, bezeichnet man als überwachte Lernalgorithmen, weil ein »Lehrer« den Algorithmus in Form der erwünschten Ausgaben für jedes Lernbeispiel anleitet. Obwohl das Erstellen eines Datensatzes geeigneter Ein- und Ausgaben oft mühevolle Handarbeit bedeu-

tet, sind überwachte Lernalgorithmen gut verständlich, und ihre Leistung ist leicht messbar. Wenn Ihre Anwendung sich als überwachte Lernaufgabe formulieren lässt und Sie einen Datensatz mit den gewünschten Ergebnissen erstellen können, lässt sich Ihre Fragestellung vermutlich durch Machine Learning beantworten.

Beispiele für überwachtes maschinelles Lernen sind:

*Auf einem Briefumschlag die Postleitzahl aus handschriftlichen Ziffern zu erkennen*

Hier besteht die Eingabe aus der eingescannten Handschrift, und die gewünschte Ausgabe sind die Ziffern der Postleitzahl. Um einen Datensatz zum Erstellen eines maschinellen Lernmodells zu erzeugen, müssen Sie zuerst viele Umschläge sammeln. Dann können Sie die Postleitzahlen selbst lesen und die Ziffern als gewünschtes Ergebnis abspeichern.

*Anhand eines medizinischen Bildes entscheiden, ob ein Tumor gutartig ist*

Hierbei ist die Eingabe das Bild, und die Ausgabe, ob der Tumor gutartig ist.

Um einen Datensatz zum Erstellen eines Modells aufzubauen, benötigen Sie eine Datenbank mit medizinischen Bildern. Sie benötigen auch eine Expertenmeinung, es muss sich also ein Arzt sämtliche Bilder ansehen und entscheiden, welche Tumore gutartig sind und welche nicht. Es ist sogar möglich, dass zur Entscheidung, ob der Tumor im Bild krebsartig ist oder nicht, zusätzlich zum Bild weitere Diagnosen nötig sind.

*Erkennen betrügerischer Aktivitäten bei Kreditkartentransaktionen*

Hierbei sind die Eingaben Aufzeichnungen von Kreditkartentransaktionen, und die Ausgabe ist, ob diese vermutlich betrügerisch sind oder nicht. Wenn Ihre Organisation Kreditkarten ausgibt, müssen Sie sämtliche Transaktionen aufzeichnen und ob ein Nutzer betrügerische Transaktionen meldet.

Bei diesen Beispielen sollte man hervorheben, dass das Sammeln der Daten bei diesen Aufgaben grundsätzlich unterschiedlich ist, auch wenn die Ein- und Ausgabedaten sehr klar wirken. Das Lesen von Umschlägen ist zwar mühevoll, aber auch unkompliziert. Medizinische Bilder und Diagnosen zu sammeln, erfordert dagegen nicht nur teure Geräte, sondern auch seltenes und teures Expertenwissen, von den ethischen und datenschutztechnischen Bedenken einmal abgesehen. Beim Erkennen von Kreditkartentrug ist das Sammeln der Daten deutlich einfacher. Ihre Kunden werden Sie mit den nötigen Ausgabedaten versorgen. Um die Ein-/Ausgabedaten für betrügerische und ehrliche Aktivitäten zu erhalten, müssen Sie nichts anderes tun, als zu warten.

*Unüberwachte Algorithmen*

sind eine weitere Art in diesem Buch behandelter Algorithmen. Beim unüberwachten Lernen sind nur die Eingabedaten bekannt, und dem Algorithmus werden keine bekannten Ausgabedaten zur Verfügung gestellt. Es sind viele erfolgreiche Anwendungen dieser Methoden bekannt, sie sind aber in der Regel schwieriger zu verstehen und auszuwerten.

Beispiele für unüberwachtes Lernen sind:

#### *Themen in einer Serie von Blögeinträgen erkennen*

Sie haben eine große Menge Textdaten und möchten diese zusammenfassen und die darin vorherrschenden Themen herausfinden. Wenn Sie nicht im Voraus wissen, welches diese Themen sind oder wie viele Themen es gibt, so gibt es keine bekannte Ausgabe.

#### *Kunden in Gruppen mit ähnlichen Vorlieben einteilen*

Mit einem Satz Kundendaten könnten Sie ähnliche Kunden erkennen und herausfinden, ob es Kundengruppen mit ähnlichen Vorlieben gibt. Bei einem Online-Geschäft könnten diese »Eltern«, »Bücherwürmer« oder »Spieler« sein. Weil diese Gruppen nicht im Voraus bekannt sind, oft nicht einmal deren Anzahl, haben Sie keine bekannte Ausgabe.

#### *Erkennung außergewöhnlicher Zugriffsmuster auf eine Webseite*

Um unrechtmäßige Nutzung oder Fehler zu erkennen, ist es oft hilfreich, Zugriffe zu finden, die sich von der Durchschnittsnutzung abheben. Jedes außergewöhnliche Muster kann sehr unterschiedlich sein, und Sie haben womöglich keinerlei Aufzeichnungen über abnorme Nutzung. Weil Sie in diesem Fall die Zugriffe einer Webseite beobachten und nicht wissen, was normale Nutzung ist und was nicht, handelt es sich hier um eine unüberwachte Fragestellung.

Sowohl bei überwachten als auch bei unüberwachten Lernaufgaben ist es wichtig, eine für den Computer verständliche Repräsentation Ihrer Eingabedaten zu haben. Oft hilft es, sich die Daten als Tabelle vorzustellen. Jeder zu untersuchende Datenpunkt (jede E-Mail, jeder Kunde, jede Transaktion) ist eine Zeile, und jede Eigenschaft, die diesen Datenpunkt beschreibt (z. B. das Alter eines Kunden oder die Menge oder der Ort der Transaktion), ist eine Spalte. Sie können Nutzer durch Alter, Geschlecht, das Datum der Registrierung und wie oft sie in Ihrem Online-Geschäft eingekauft haben, charakterisieren. Das Bild eines Tumors lässt sich durch die Graustufenwerte jedes Pixels beschreiben oder aber durch Größe, Gestalt und Farbe des Tumors.

Jede Entität oder Zeile bezeichnet man beim maschinellen Lernen als *Datenpunkt* (oder Probe), die Spalten – also die Eigenschaften, die diese Entitäten beschreiben werden – nennt man *Merkmale*.

Im weiteren Verlauf dieses Buches werden wir uns genauer mit dem Aufbau einer guten Datenrepräsentation beschäftigen, was man als *Extrahieren von Merkmalen* oder *Merkmalsgenerierung* bezeichnet. Sie sollten auf jeden Fall bedenken, dass kein maschinelles Lernverfahren ohne entsprechende Information zu Vorhersagen in der Lage ist. Wenn zum Beispiel das einzige bekannte Merkmal eines Patienten der Nachname ist, wird kein Algorithmus in der Lage sein, das Geschlecht vorherzusagen. Diese Information ist schlicht nicht in Ihren Daten enthalten. Wenn Sie

ein weiteres Merkmal mit dem Vornamen des Patienten hinzufügen, haben Sie mehr Glück, da sich das Geschlecht häufig aus dem Vornamen vorhersagen lässt.

## Ihre Aufgabe und Ihre Daten kennen

Der möglicherweise wichtigste Teil beim maschinellen Lernen ist, dass Sie Ihre Daten verstehen und wie diese mit der zu lösenden Aufgabe zusammenhängen. Es ist nicht sinnvoll, zufällig einen Algorithmus auszuwählen und Ihre Daten hineinzuwerfen. Bevor Sie ein Modell konstruieren können, ist es notwendig, zu verstehen, was in Ihrem Datensatz vorgeht. Jeder Algorithmus unterscheidet sich darin, bei welchen Daten und welchen Aufgabenstellungen er am besten funktioniert. Wenn Sie eine Aufgabe mit maschinellem Lernen bearbeiten, sollten Sie folgende Fragen beantworten oder zumindest im Hinterkopf behalten:

- Welche Fragen versuche ich zu beantworten? Glaube ich, dass die erhobenen Daten diese Frage beantworten können?
- Wie lässt sich meine Frage am besten als maschinelle Lernaufgabe ausdrücken?
- Habe ich genug Daten gesammelt, um die zu beantwortende Fragestellung zu repräsentieren?
- Welche Merkmale der Daten habe ich extrahiert? Sind diese zu den richtigen Vorhersagen in der Lage?
- Wie messe ich, ob meine Anwendung erfolgreich ist?
- Wie interagiert mein maschinelles Lernmodell mit anderen Teilen meiner Forschungsarbeit oder meines Produkts?

In einem breiteren Kontext sind die Algorithmen und Methoden für maschinelles Lernen nur Teil eines größeren Prozesses zum Lösen einer bestimmten Aufgabe. Es ist sinnvoll, das große Ganze stets im Blick zu behalten. Viele Leute investieren eine Menge Zeit in das Entwickeln eines komplexen maschinellen Lernsystems, nur um hinterher herauszufinden, dass sie das falsche Problem gelöst haben.

Wenn man sich eingehend mit den technischen Aspekten maschinellen Lernens beschäftigt (wie wir es in diesem Buch tun werden), ist es leicht, die endgültigen Ziele aus den Augen zu verlieren. Wir werden die hier gestellten Fragen nicht im Detail diskutieren, halten Sie aber dazu an, sämtliche explizit oder implizit getroffenen Annahmen beim Aufbau maschiner Lernmodelle zu berücksichtigen.

## Warum Python?

Python ist für viele Anwendungen aus dem Bereich Data Science die lingua franca geworden. Python kombiniert die Ausdrucks Kraft allgemein nutzbarer Programmiersprachen mit der einfachen Benutzbarkeit einer domänenspezifischen Skript-

sprache wie MATLAB oder R. Für Python gibt es Bibliotheken zum Laden von Daten, Visualisieren, Berechnen von Statistiken, Sprachverarbeitung, Bildverarbeitung usw. Dies gibt Data Scientists einen sehr umfangreichen Werkzeugkasten mit Funktionalität für allgemeine und besondere Einsatzgebiete. Einer der Hauptvorteile von Python ist die Möglichkeit, direkt mit dem Code zu interagieren, sei es in einer Konsole oder einer anderen Umgebung wie dem Jupyter Notebook, das wir uns in Kürze ansehen werden. Machine Learning und Datenanalyse sind von Grund auf iterative Prozesse, bei denen die Daten die Analyse bestimmen. Es ist entscheidend, diese Prozesse mit Werkzeugen zu unterstützen, die schnelle Iterationen und leichte Benutzbarkeit ermöglichen.

Als allgemein einsetzbare Programmiersprache lassen sich mit Python auch komplexe grafische Benutzeroberflächen (GUIs) und Webdienste entwickeln und in bestehende Systeme integrieren.

## scikit-learn

scikit-learn ist ein Open Source-Projekt, Sie dürfen es also kostenlos verwenden und verbreiten. Jeder kommt leicht an die Quelltexte heran und kann sehen, was hinter den Kulissen passiert. Das scikit-learn-Projekt wird kontinuierlich weiterentwickelt und verbessert und besitzt eine große Nutzergemeinde. Es enthält eine Anzahl hochentwickelter maschineller Lernalgorithmen und eine umfangreiche Dokumentation (<http://scikit-learn.org/stable/documentation>) zu jedem Algorithmus. scikit-learn ist sehr beliebt, und die Nummer Eins der Python-Bibliotheken für Machine Learning. Es wird in Wirtschaft und Forschung eingesetzt, und im Netz existieren zahlreiche Tutorials und Codebeispiele. scikit-learn arbeitet eng mit einigen weiteren wissenschaftlichen Python-Werkzeugen zusammen, die wir im Verlauf dieses Kapitels kennenlernen werden.

Wir empfehlen, dass Sie beim Lesen dieses Buches auch den User Guide ([http://scikit-learn.org/stable/user\\_guide.html](http://scikit-learn.org/stable/user_guide.html)) und die Dokumentation der API von scikit-learn lesen, um zusätzliche Details und weitere Optionen zu jedem Algorithmus kennenzulernen. Die Online-Dokumentation ist sehr ausführlich, und dieses Buch liefert Ihnen die Grundlagen in maschinellem Lernen, um es im Detail zu verstehen.

## Installieren von scikit-learn

scikit-learn benötigt zwei weitere Python-Pakete, *NumPy* und *SciPy*. Zum Plotten und zur interaktiven Entwicklung sollten Sie außerdem *matplotlib*, *IPython* und Jupyter Notebook installieren. Wir empfehlen Ihnen, eine der folgenden Python-Distributionen zu verwenden, in denen die notwendigen Pakete bereits enthalten sind:

### *Anaconda (<https://store.continuum.io/cshop/anaconda/>)*

Eine Python-Distribution für Datenverarbeitung in großem Stil, vorherrschende Analyse und wissenschaftliche Berechnungen. Anaconda enthält NumPy, SciPy, matplotlib, pandas, IPython, Jupyter Notebook und scikit-learn. Dies ist eine sehr bequeme Lösung unter Mac OS, Windows und Linux, und wir empfehlen sie Anwendern ohne bestehende Installation einer wissenschaftlichen Python-Umgebung. Anaconda enthält inzwischen auch eine kostenlose Ausgabe der Bibliothek *Intel MKL*. Das Verwenden von MKL (was bei Installation von Anaconda automatisch geschieht) führt zu deutlichen Geschwindigkeitsverbesserungen bei vielen der Algorithmen in scikit-learn.

### *Enthought Canopy (<https://www.enthought.com/products/canopy/>)*

Eine weitere Python-Distribution für wissenschaftliches Arbeiten. Diese enthält NumPy, SciPy, matplotlib, pandas und IPython, aber in der kostenlosen Version ist scikit-learn nicht enthalten. Wenn Ihre Institution akademische Abschlüsse vergibt, können Sie eine akademische Lizenz beantragen und freien Zugang zu einem Abonnement von Enthought Canopy erhalten. Enthought Canopy ist für Python 2.7.x verfügbar und läuft auf Mac OS, Windows und Linux.

### *Python(x,y) (<http://python-xy.github.io/>)*

Eine freie Python-Distribution für wissenschaftliches Arbeiten, insbesondere unter Windows. Python(x,y) enthält NumPy, SciPy, matplotlib, pandas, IPython und scikit-learn.

Wenn Sie bereits eine Python-Installation haben, können Sie die folgenden Pakete mit pip installieren:

```
$ pip install numpy scipy matplotlib ipython scikit-learn pandas
```

## **Grundlegende Bibliotheken und Werkzeuge**

Es ist wichtig zu verstehen, was scikit-learn ist und wie es funktioniert. Es gibt jedoch einige weitere Bibliotheken, die Ihre Produktivität verbessern werden. scikit-learn basiert auf den Python-Bibliotheken NumPy und SciPy. Außer NumPy und SciPy werden wir auch pandas und matplotlib verwenden. Außerdem werden wir Jupyter Notebook, eine browsergestützte interaktive Programmierumgebung, kennenlernen. Kurz gesagt, sollten Sie etwas über folgende Hilfsmittel wissen, um das Beste aus scikit-learn herauszuholen.<sup>1</sup>

---

<sup>1</sup> Wenn Sie sich mit NumPy oder matplotlib gar nicht auskennen, empfehlen wir Ihnen die Lektüre des ersten Kapitels der SciPy Lecture Notes (<http://www.scipy-lectures.org/>).

## Jupyter Notebook

Das Jupyter Notebook ist eine interaktive Umgebung, um Code über den Browser auszuführen. Es ist ein großartiges Werkzeug zur erkundenden Datenanalyse und wird von Data Scientists in großem Stil eingesetzt. Obwohl das Jupyter Notebook viele Programmiersprachen unterstützt, benötigen wir nur die Python-Unterstützung. In einem Jupyter Notebook können Sie leicht Code, Texte und Bilder einbinden. Dieses gesamte Buch wurde als Jupyter Notebook geschrieben. Sie können sich sämtliche enthaltenen Codebeispiele von GitHub ([https://github.com/amueller/introduction\\_to\\_ml\\_with\\_python](https://github.com/amueller/introduction_to_ml_with_python)) herunterladen.

## NumPy

NumPy ist eines der grundlegenden Pakete für wissenschaftliche Berechnungen in Python. Es enthält die Funktionalität für mehrdimensionale Arrays, mathematische Funktionen wie lineare Algebra und Fourier-Transformationen sowie Generatoren für Pseudozufallszahlen.

In scikit-learn ist das NumPy-Array die fundamentale Datenstruktur. scikit-learn verarbeitet Daten in Form von NumPy-Arrays. Jegliche Daten, die Sie verwenden, müssen in ein NumPy-Array umgewandelt werden. Der wichtigste Bestandteil von NumPy ist die Klasse `ndarray`, ein mehrdimensionales ( $n$ -dimensionales) Array. Sämtliche Elemente eines Arrays müssen vom gleichen Typ sein. Ein NumPy-Array sieht folgendermaßen aus:

**In[1]:**

```
import numpy as np  
  
x = np.array([[1, 2, 3], [4, 5, 6]])  
print("x:\n{}".format(x))
```

**Out[1]:**

```
x:  
[[1 2 3]  
 [4 5 6]]
```

Wir werden NumPy in diesem Buch sehr viel verwenden und die Objekte der Klasse `ndarray` als »NumPy-Arrays« oder einfach »Arrays« bezeichnen.

## SciPy

SciPy ist eine Sammlung von Funktionen zum wissenschaftlichen Rechnen in Python. Sie enthält unter anderem Routinen für fortgeschrittene lineare Algebra, Optimierung mathematischer Funktionen, Signalverarbeitung, spezielle mathematische Funktionen und statistische Verteilungen. scikit-learn bedient sich aus dem Funktionspool in SciPy, um seine Algorithmen zu implementieren. Der für uns wichtigste Bestandteil von SciPy ist `scipy.sparse`: Dieser stellt *dünn besetzte Matrizen*

zen zur Verfügung, eine weitere Datenrepräsentation in scikit-learn. Dünn besetzte Matrizen werden immer dann eingesetzt, wenn ein 2-D-Array zum größten Teil aus Nullen besteht:

**In[2]:**

```
from scipy import sparse

# Erstelle ein 2-D NumPy Array mit einer Diagonale aus Einsen und sonst Nullen
eye = np.eye(4)
print("NumPy Array:\n{}".format(eye))
```

**Out[2]:**

```
NumPy Array:
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
```

**In[3]:**

```
# Wandle das NumPy Array in eine SciPy sparse matrix im CSR-Format um
# Nur die Einträge ungleich null werden gespeichert
sparse_matrix = sparse.csr_matrix(eye)
print("\nSciPy sparse CSR matrix:\n{}".format(sparse_matrix))
```

**Out[3]:**

```
SciPy sparse CSR matrix:
(0, 0)    1.0
(1, 1)    1.0
(2, 2)    1.0
(3, 3)    1.0
```

Normalerweise ist es nicht möglich, eine dichte Repräsentation einer dünn besetzten Matrix zu erzeugen (sie würde nämlich nicht in den Speicher passen), daher müssen wir die dünn besetzte Matrix direkt erzeugen. Hier ist eine Möglichkeit, die gleiche sparse matrix wie oben im COO-Format zu erstellen:

**In[4]:**

```
data = np.ones(4)
row_indices = np.arange(4)
col_indices = np.arange(4)
eye_coo = sparse.coo_matrix((data, (row_indices, col_indices)))
print("COO-Repräsentation:\n{}".format(eye_coo))
```

**Out[4]:**

```
COO-Repräsentation:
(0, 0)    1.0
(1, 1)    1.0
(2, 2)    1.0
(3, 3)    1.0
```

Weitere Details zu dünn besetzten Matrizen in SciPy finden Sie in den SciPy Lecture Notes (<http://www.scipy-lectures.org/>).

## matplotlib

matplotlib ist die wichtigste Python-Bibliothek zum wissenschaftlichen Plotten. Sie enthält Funktionen zum Erstellen von Diagrammen in Publikationsqualität, z. B. Liniendiagramme, Histogramme, Streudiagramme usw. Ihre Daten und unterschiedliche Aspekte Ihrer Daten zu visualisieren, liefert wichtige Erkenntnisse, und wir werden matplotlib für sämtliche Visualisierungsaufgaben einsetzen. Wenn Sie mit einem Jupyter Notebook arbeiten, können Sie Diagramme über die Befehle `%matplotlib notebook` und `%matplotlib inline` direkt im Browser darstellen. Wir empfehlen Ihnen `%matplotlib notebook`, wodurch Sie eine interaktive Umgebung erhalten (obwohl wir zur Produktion dieses Buches `%matplotlib inline` eingesetzt haben). Zum Beispiel erstellt der folgende Code das Diagramm in Abbildung 1-1:

In[5]:

```
%matplotlib inline
import matplotlib.pyplot as plt

# Erstelle eine Zahlenfolge von -10 bis 10 mit 100 Zwischenschritten
x = np.linspace(-10, 10, 100)
# Erstelle ein zweites Array mit einer Sinusfunktion
y = np.sin(x)
# Die Funktion plot zeichnet ein Liniendiagramm eines Arrays über dem anderen
plt.plot(x, y, marker="x")
```

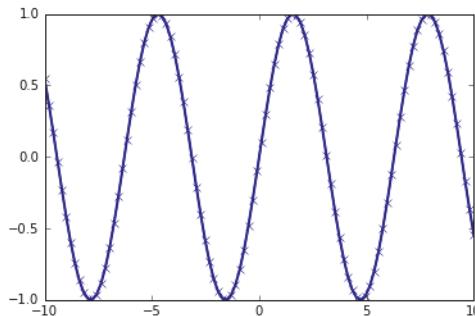


Abbildung 1-1: Mit matplotlib erstelltes Liniendiagramm einer Sinusfunktion

## pandas

pandas ist eine Python-Bibliothek zur Datenaufbereitung und Analyse. Sie ist um eine Datenstruktur namens DataFrame herum aufgebaut, die dem DataFrame in R nachempfunden ist. Einfach gesagt, ist ein pandas DataFrame eine Tabelle, einem Excel-Tabellenblatt nicht unähnlich. pandas enthält eine große Bandbreite an Methoden zum Modifizieren und Verarbeiten dieser Tabellen; insbesondere sind SQL-artige Suchanfragen und Verbindungsoperationen möglich. Im Gegensatz zu NumPy, bei dem sämtliche Einträge eines Arrays den gleichen Typ haben müssen, lässt pandas in jeder Spalte unterschiedliche Typen zu (z. B. Integer, Datum, Fließ-

kommazahl oder String). Ein weiteres wertvolles Werkzeug in pandas sind Einleseprozeduren für eine große Anzahl Dateiformate und Datenbanken wie SQL, Excel-Dateien und kommaseparierte Dateien (CSV). Wir werden in diesem Buch die Funktionalität von pandas nicht im Detail besprechen. Allerdings gibt das Buch *Datenanalyse mit Python* von Wes McKinney (O'Reilly 2015, ISBN 978-3-96009-000-7) eine großartige Einführung. Hier ist ein kleines Beispiel für das Erstellen eines DataFrames über ein Dictionary:

In[6]:

```
import pandas as pd

# erstelle einen einfachen Datensatz mit Personen
data = {'Name': ["John", "Anna", "Peter", "Linda"],
        'Location' : ["New York", "Paris", "Berlin", "London"],
        'Age' : [24, 13, 53, 33]
       }

data_pandas = pd.DataFrame(data)
# IPython.display erlaubt das "pretty printing" von Data Frames
# im Jupyter Notebook
display(data_pandas)
```

Dadurch erhalten wir folgende Ausgabe:

	Age	Location	Name
0	24	New York	John
1	13	Paris	Anna
2	53	Berlin	Peter
3	33	London	Linda

Es gibt mehrere Möglichkeiten, Anfragen an diese Tabelle zu senden. Beispielsweise:

In[7]:

```
# Wähle alle Zeilen mit einem Wert für age größer 30 aus
display(data_pandas[data_pandas.Age > 30])
```

Dadurch erhalten wir folgendes Ergebnis:

	Age	Location	Name
2	53	Berlin	Peter
3	33	London	Linda

## mlearn

Dieses Buch wird von Code begleitet, den Sie auf GitHub ([https://github.com/amueller/introduction\\_to\\_ml\\_with\\_python](https://github.com/amueller/introduction_to_ml_with_python)) finden. Der begleitende Code enthält

nicht nur sämtliche Beispiele aus diesem Buch, sondern auch die Bibliothek `mglearn`. Dies ist eine Bibliothek von Hilfsfunktionen, die wir für dieses Buch geschrieben haben, um die Codebeispiele nicht mit Details zum Plotten und Laden von Daten zu verunstalten. Bei Interesse können Sie die Details sämtlicher Funktionen im Repository nachschlagen, aber die Details des Moduls `mglearn` sind für das Material in diesem Buch nicht wirklich wichtig. Wenn Sie einen Aufruf von `mglearn` im Code sehen, ist es normalerweise eine Möglichkeit, schnell ein ansprechendes Bild zu erzeugen oder interessante Daten in die Finger zu bekommen.



Im Verlauf dieses Buches werden wir ständig NumPy, matplotlib und pandas einsetzen. Alle Codebeispiele setzen die folgenden import-Befehle voraus:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import mglearn
from IPython import display
```

Wir gehen außerdem davon aus, dass Sie den Code in einem Jupyter Notebook unter Verwendung der Funktionen `%matplotlib notebook` oder `%matplotlib inline` zum Darstellen von Diagrammen ausführen. Wenn Sie kein Notebook oder keinen dieser Befehle verwenden, müssen Sie `plt.show` aufrufen, um die Diagramme zu sehen.

## Python 2 versus Python 3

Es gibt zwei größere Versionen von Python, die im Moment weitverbreitet sind: Python 2 (genauer 2.7) und Python 3 (die gegenwärtig jüngste Version ist 3.5). Das sorgt bisweilen für Verwirrung. Python 2 wird nicht mehr weiterentwickelt, aber wegen tief greifender Änderungen in Python 3 läuft für Python 2 geschriebener Code meist nicht unter Python 3. Wenn Python für Sie neu ist oder Sie ein neues Projekt beginnen, empfehlen wir Ihnen wärmstens die neueste Version von Python 3. Wenn Sie von einer größeren Menge unter Python 2 geschriebenen Codes abhängig sind, sind Sie vorläufig von einem Upgrade entschuldigt. Sie sollten jedoch so bald wie möglich auf Python 3 umsteigen. Beim Schreiben neuer Programme ist es meist einfach, Code zu schreiben, der sowohl unter Python 2 als auch unter Python 3 läuft.<sup>2</sup> Wenn Sie sich nicht auf bestehende Software stützen müssen, sollten Sie definitiv Python 3 verwenden. Sämtliche Codebeispiele in diesem Buch funktionieren mit beiden Versionen. Allerdings kann sich die genaue Ausgabe unter Python 2 stellenweise von der unter Python 3 unterscheiden.

---

<sup>2</sup> Das Paket six (<https://pypi.python.org/pypi/six>) kann dabei sehr hilfreich sein.

# In diesem Buch verwendete Versionen

Wir verwenden in diesem Buch die folgenden Versionen der oben erwähnten Bibliotheken:

**In[8]:**

```
import sys
print("Python Version: {}".format(sys.version))

import pandas as pd
print("pandas Version: {}".format(pd.__version__))

import matplotlib
print("matplotlib Version: {}".format(matplotlib.__version__))

import numpy as np
print("NumPy Version: {}".format(np.__version__))

import scipy as sp
print("SciPy Version: {}".format(sp.__version__))

import IPython
print("IPython Version: {}".format(IPython.__version__))

import sklearn
print("scikit-learn Version: {}".format(sklearn.__version__))
```

**Out[8]:**

```
Python Version: 3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul 2 2016, 17:53:06)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)]
pandas Version: 0.18.1
matplotlib Version: 1.5.1
NumPy Version: 1.11.1
SciPy Version: 0.17.1
IPython Version: 5.1.0
scikit-learn Version: 0.18
```

Auch wenn es nicht entscheidend ist, genau diese Versionen zu verwenden, sollten Sie eine Version von scikit-learn haben, die mindestens so neu ist wie unsere.

Nun haben wir alles eingerichtet und können in unsere erste Anwendung maschinellen Lernens starten.



Dieses Buch geht davon aus, dass Sie mindestens Version 0.18 von scikit-learn installiert haben. Das Modul `model_selection` wurde in 0.18 hinzugefügt, und wenn Sie eine ältere Version von scikit-learn verwenden, werden Sie die `import`-Anweisungen für dieses Modul anpassen müssen.

# Eine erste Anwendung: Klassifizieren von Iris-Spezies

In diesem Abschnitt werden wir eine einfache Anwendung maschinellen Lernens durchgehen und unser erstes Modell erstellen. Dabei werden wir einige zentrale Konzepte und Begriffe kennenlernen.

Nehmen wir an, dass eine Hobbybotanikerin daran interessiert ist, die Spezies einiger gefundener Irisblüten zu unterscheiden. Sie hat einige Messdaten zu jeder Iris gesammelt: Länge und Breite der Kronblätter (petals) und Länge und Breite der Kelchblätter (sepals). Alle Längen sind in Zentimetern gemessen worden (siehe Abbildung 1-2).

Sie besitzt darüber hinaus die Messdaten einiger Irisblüten, die zuvor von einem professionellen Botaniker den Spezies *setosa*, *versicolor* und *virginica* zugeordnet wurden. Bei diesen Messungen kann sie sich sicher sein, welchen Spezies jede Iris zugeordnet wurde. Nehmen wir an, dies seien die einzigen Spezies, denen unsere Hobbybotanikerin in freier Wildbahn begegnet.

Unser Ziel ist es, ein maschinelles Lernmodell zu entwickeln, das anhand der Messdaten für Iris mit bekannter Spezies lernen kann, sodass wir die Spezies einer neuen Iris vorhersagen können.

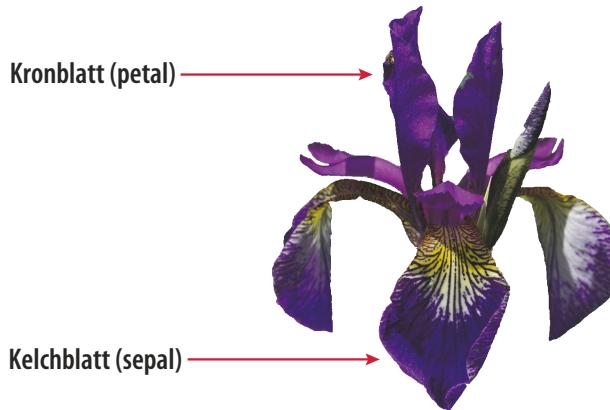


Abbildung 1-2: Teile der Irisblüte

Weil wir Messungen haben, für die wir die korrekte Iris-Spezies kennen, handelt es sich hier um eine überwachte Lernaufgabe. Bei dieser Aufgabe möchten wir eine von mehreren Möglichkeiten (die Iris-Spezies) vorhersagen. Dies ist ein Beispiel für eine *Klassifikationsaufgabe*. Die möglichen Ausgaben (unterschiedliche Iris-Spezies) nennt man *Kategorien*. Jede Iris im Datensatz gehört einer von drei Kategorien an, daher handelt es sich hier um eine Klassifikationsaufgabe mit drei Kategorien.

Für einen einzelnen Datenpunkt (eine Iris) ist die gewünschte Ausgabe die Spezies dieser Blüte. Man bezeichnet die einem bestimmten Datenpunkt zugehörige Spezies auch als *Label*.

## Die Daten kennenlernen

Die Daten, die wir in diesem Beispiel verwenden werden, sind der Datensatz Iris, ein klassischer Datensatz aus dem maschinellen Lernen und der Statistik. Er ist in scikit-learn im Modul datasets enthalten. Wir können ihn durch Aufruf der Funktion load\_iris laden:

In[9]:

```
from sklearn.datasets import load_iris  
iris_dataset = load_iris()
```

Das von load\_iris zurückgegebene Objekt iris ist ein Objekt vom Typ Bunch, das einem Dictionary sehr ähnlich ist. Es enthält Schlüssel und Werte:

In[10]:

```
print("Schlüssel von iris_dataset: \n{}".format(iris_dataset.keys()))
```

Out[10]:

```
Schlüssel von iris_dataset:  
dict_keys(['target_names', 'feature_names', 'DESCR', 'data', 'target'])
```

Der Wert des Schlüssels DESCR ist eine kurze Beschreibung des Datensatzes. Wir führen hier den Anfang der Beschreibung auf (Sie können gerne den Rest selbst nachschlagen):

In[11]:

```
print(iris_dataset['DESCR'][:193] + "\n...")
```

Out[11]:

```
Iris Plants Database  
=====  
Notes  
----  
Data Set Characteristics:  
 :Number of Instances: 150 (50 in each of three classes)  
 :Number of Attributes: 4 numeric, predictive att  
 ...  
 ----
```

Der Wert des Schlüssels target\_names ist ein Array aus Strings, die die Spezies der vorherzusagenden Blüten enthalten:

In[12]:

```
print("Zielbezeichnungen: {}".format(iris_dataset['target_names']))
```

Out[12]:

```
Zielbezeichnungen: ['setosa' 'versicolor' 'virginica']
```

Der Wert von feature\_names ist eine Liste von Strings mit den Beschreibungen jedes Merkmals:

**In[13]:**

```
print("Namen der Merkmale: \n{}".format(iris_dataset['feature_names']))
```

**Out[13]:**

```
Namen der Merkmale:  
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',  
 'petal width (cm)']
```

Die Daten selbst sind in den Feldern target und data enthalten. data enthält die Messwerte für Länge und Breite der Kelch- und Kronblätter als NumPy-Array:

**In[14]:**

```
print("Typ der Daten: {}".format(type(iris_dataset['data'])))
```

**Out[14]:**

```
Typ der Daten: <class 'numpy.ndarray'>
```

Die Zeilen im Array data entsprechen einzelnen Blüten, die Spalten entsprechen den vier für jede Blüte erhobenen Messwerten:

**In[15]:**

```
print("Abmessung der Daten: {}".format(iris_dataset['data'].shape))
```

**Out[15]:**

```
Abmessung der Daten: (150, 4)
```

Wir sehen, dass das Array Messungen von 150 unterschiedlichen Blüten enthält. Bedenken Sie, dass einzelne Datenpunkte beim maschinellen Lernen auch *Proben* genannt und ihre Eigenschaften als *Merkmale* bezeichnet werden. Die *Abmessungen* (shape) des Arrays data sind die Anzahl der Proben mit der Anzahl der Merkmale multipliziert. Dies ist eine Konvention in scikit-learn, und es wird stets angenommen, dass sich Ihre Daten in diesem Format befinden. Hier sind die Werte der Merkmale der ersten vier Datenpunkte:

**In[16]:**

```
print("Die ersten fünf Zeilen der Daten:\n{}".format(iris_dataset['data'][:5]))
```

**Out[16]:**

```
Die ersten fünf Zeilen der Daten:  
[[ 5.1  3.5  1.4  0.2]  
 [ 4.9  3.   1.4  0.2]  
 [ 4.7  3.2  1.3  0.2]  
 [ 4.6  3.1  1.5  0.2]  
 [ 5.   3.6  1.4  0.2]]
```

Aus diesen Daten können wir ersehen, dass die fünf ersten Blüten alle 0.2 cm breite Kronblätter haben und dass die erste Blüte mit 5.1 cm die längsten Kelchblätter hat.

Das Array target enthält die Spezies jeder vermessenen Blüte ebenfalls als NumPy-Array:

In[17]:

```
print("Typ der Zielgröße: {}".format(type(iris_dataset['target'])))
```

Out[17]:

Typ der Zielgröße: <class 'numpy.ndarray'>

target ist ein eindimensionales Array mit einem Eintrag pro Blüte:

In[18]:

```
print("Abmessungen der Zielgröße: {}".format(iris_dataset['target'].shape))
```

**Out[18]:**

Abmessungen der Zielgröße: (150,)

Die Spezies sind als ganze Zahlen von 0 bis 2 kodiert:

In[19]:

```
print("Zielwerte:\n{}".format(iris_dataset['target']))
```

Out[19]:

Die Bedeutung der Zahlen sind im Array `iris['target_names']` enthalten: 0 steht für `setosa`, 1 für `versicolor` und 2 für `virginica`.

## Erfolg nachweisen: Trainings- und Testdaten

Wir möchten anhand dieser Daten ein maschinelles Lernmodell konstruieren, mit dem wir die Iris-Spezies für neue Messdaten vorhersagen können. Bevor wir aber unser Modell auf neue Messdaten anwenden können, müssen wir erst einmal wissen, ob es überhaupt funktioniert – ob wir also den Vorhersagen trauen können.

Leider können wir das Modell nicht mit den Daten auswerten, mit denen wir es konstruiert haben. Dies liegt daran, dass unser Modell stets einfach den gesamten Trainingsdatensatz auswendig lernen könnte und damit immer für jeden Punkt in den Trainingsdaten die korrekte Bezeichnung vorhersagt. Dieses »auswendig Lernen« verrät uns nicht, ob unser Modell gut *verallgemeinert* (also ob es auch für neue Daten gut funktioniert).

Um die Leistung eines Modells zu bewerten, zeigen wir diesem neue Daten mit bekannten Labels (Daten, die es zuvor nicht gesehen hat). Normalerweise unterteilt man dazu die gesammelten und gelabelten Daten (hier unsere 150 vermessenen Blüten) in zwei Gruppen. Ein Teil der Daten wird zum Aufbau unseres maschinellen Lernmodells verwendet. Wir bezeichnen diesen als *Trainingsdaten* oder *Train-*

*ningsdatensatz*. Die übrigen Daten werden verwendet, um die Vorhersagequalität des Modells zu beurteilen; wir bezeichnen diese als *Testdaten*, *Testdatensatz* oder *zurückgehaltene Daten*.

scikit-learn enthält eine Funktion, die einen Datensatz durchmischt und für Sie aufteilt: die Funktion `train_test_split`. Diese Funktion verwendet 75 % der Einträge mit den entsprechenden Labels als Trainingsdatensatz. Die übrigen 25 % der Daten werden zusammen mit den übrigen Labels zum Testdatensatz erklärt. Die Entscheidung, wie viele Daten dem Trainings- und dem Testdatensatz zugeordnet werden, ist ein wenig willkürlich, aber einen Testdatensatz mit 25 % der Testdaten aufzubauen, ist eine gute Faustregel.

In scikit-learn werden die Daten für gewöhnlich mit einem großen X gekennzeichnet, die Labels dagegen mit einem kleingeschriebenen y. Dies leitet sich von der in der Mathematik üblichen Schreibweise  $f(x)=y$  ab, bei der x die Eingabe einer Funktion und y die Ausgabe ist. Wir verwenden eine weitere mathematische Konvention, indem wir für das zweidimensionale Array (eine Matrix) mit den Daten ein großes X verwenden, für das eindimensionale Array (einen Vektor) mit der Zielgröße ein kleingeschriebenes y.

Rufen wir nun `train_test_split` für unsere Daten auf und weisen wir die Ausgabe mit dieser Nomenklatur zu:

**In[20]:**

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    iris_dataset['data'], iris_dataset['target'], random_state=0)
```

Bevor wir die Teilung durchführen, durchmischt `train_test_split` den Datensatz über einen Pseudozufallszahlengenerator. Würden wir einfach nur die letzten 25 % der Daten als Testdatensatz verwenden, würden sämtliche Datenpunkte der Kategorie 2 angehören, da die Datenpunkte nach den Labels sortiert sind (dies ist in der oben gezeigten Ausgabe von `iris['target']` der Fall). Ein Testdatensatz mit nur einer der drei Kategorien würde uns nicht viel darüber verraten, wie gut unser Modell verallgemeinert. Deshalb mischen wir die Daten, um sicherzustellen, dass die Testdaten Einträge aus allen Kategorien enthalten.

Um zu garantieren, dass wir für den gleichen Funktionsaufruf mehrmals hintereinander die gleiche Ausgabe erhalten, übergeben wir dem Pseudozufallszahlengenerator über den Parameter `random_state` einen festen Seed-Wert. Dadurch wird das Ergebnis deterministisch, und diese Zeile liefert stets das gleich Ergebnis. Wir werden bei Zufallsprozeduren in diesem Buch `random_state` stets auf diese Weise setzen.

Die Ausgabe der Funktion `train_test_split` ist `X_train`, `X_test`, `y_train` und `y_test`, die allesamt NumPy-Arrays sind. `X_train` enthält 75 % der Zeilen im Datensatz, `X_test` enthält die restlichen 25 %:

**In[21]:**

```
print("Abmessungen von X_train: {}".format(X_train.shape))
print("Abmessungen von y_train: {}".format(y_train.shape))
```

**Out[21]:**

```
Abmessungen von X_train: (112, 4)
Abmessungen von y_train: (112,)
```

**In[22]:**

```
print("Abmessungen von X_test: {}".format(X_test.shape))
print("Abmessungen von y_test: {}".format(y_test.shape))
```

**Out[22]:**

```
Abmessungen von X_test: (38, 4)
Abmessungen von y_test: (38,)
```

## Das Wichtigste zuerst: Sichten Sie Ihre Daten

Bevor wir ein maschinelles Lernmodell konstruieren, lohnt sich meist eine genaue Inspektion der Daten, um herauszufinden, ob die Aufgabe ohne maschinelles Lernen lösbar und die gewünschte Information überhaupt in den Daten enthalten ist.

Außerdem ist die Inspektion von Daten ein sinnvoller Weg, Abnormitäten und Besonderheiten Ihrer Daten zu entdecken. Vielleicht wurden z. B. einige der Irisblüten in Zoll statt in Zentimetern vermessen. Im wirklichen Leben sind Inkonsistenz der Daten und unerwartete Messwerte sehr häufig.

Eine der besten Möglichkeiten zur Dateninspektion besteht darin, die Daten zu visualisieren. Dies lässt sich beispielsweise mit einem *Streudiagramm* realisieren. In einem Streudiagramm wird ein Merkmal auf der x-Achse und ein weiteres auf der y-Achse aufgetragen, und für jeden Datenpunkt wird ein Symbol gezeichnet. Leider haben Computerbildschirme nur zwei Dimensionen, wodurch wir nur zwei (vielleicht auch drei) Merkmale zeitgleich darstellen können. Datensätze mit mehr als drei Merkmalen lassen sich auf diese Weise nur schwer darstellen. Eine Möglichkeit, dieses Problem zu umgehen, ist ein *Paarplot*, bei dem wir uns alle möglichen Merkmalspaare ansehen. Wenn Ihnen eine kleine Anzahl Merkmale vorliegt, wie die vier in unserem Beispiel, ist dies sehr sinnvoll. Sie sollten aber bedenken, dass ein Paarplot nicht die Wechselwirkungen aller Merkmale gleichzeitig zeigt. Daher sind nicht unbedingt alle interessanten Aspekte der Daten bei dieser Darstellungsart erkennbar.

Abbildung 1-3 zeigt einen Paarplot der Merkmale im Trainingsdatensatz. Die Datenpunkte sind entsprechend der zugehörigen Iris-Spezies eingefärbt. Um dieses Diagramm zu erstellen, wandeln wir das NumPy-Array zunächst in ein pandas DataFrame um. pandas enthält eine Funktion zum Zeichnen von Paarplots namens `scatter_matrix`. Die Diagonale dieser Matrix ist mit Histogrammen für jedes Merkmal belegt:

In[23]:

```
# erstelle aus den Daten in X_train ein DataFrame  
# verwende die Strings aus iris_dataset.feature_names als Spaltenüberschriften  
iris_dataframe = pd.DataFrame(X_train, columns=iris_dataset.feature_names)  
# erstelle eine Matrix von Streudiagrammen aus dem DataFrame  
# färbe nach y_train ein  
grr = pd.scatter_matrix(iris_dataframe, c=y_train, figsize=(15, 15), marker='o',  
hist_kwds={'bins': 20}, s=60, alpha=.8, cmap=mglearn.cm3)
```

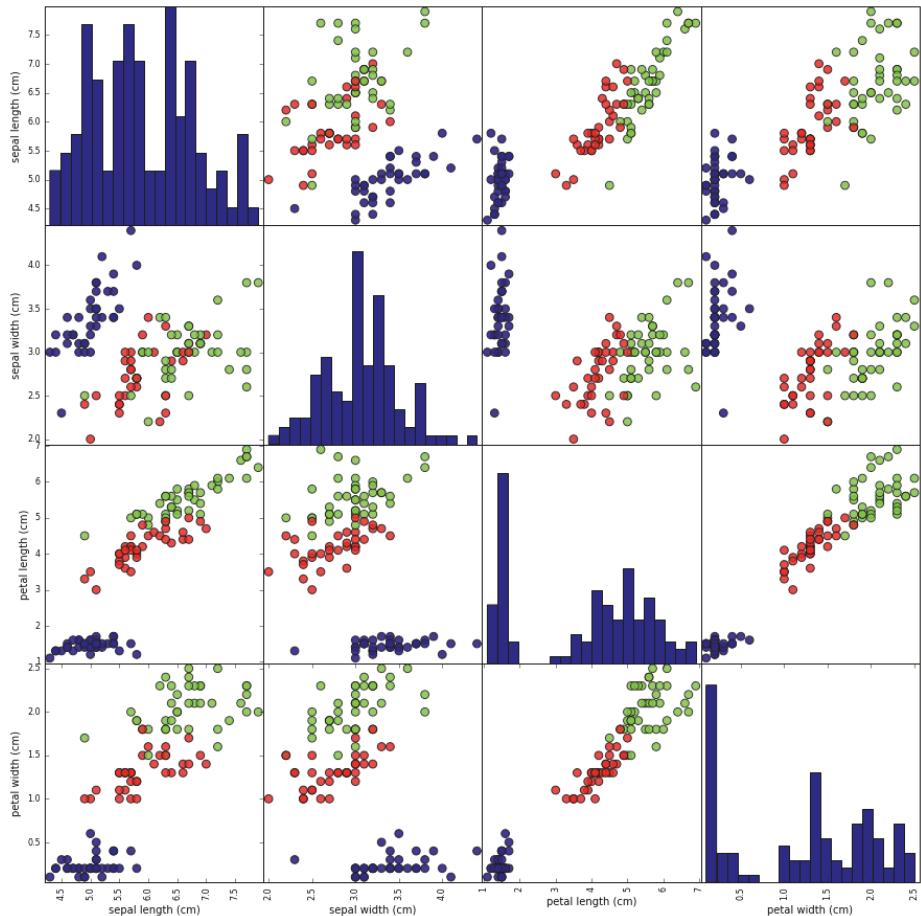


Abbildung 1-3: Paarplot für den Iris-Datensatz, nach Kategoriebezeichnung eingefärbt

Aus den Plots sehen wir, dass die drei Kategorien recht gut durch die Messungen von Kelch- und Kronblättern voneinander abgegrenzt sind. Das bedeutet, dass ein maschinelles Lernmodell vermutlich in der Lage ist, diese zu unterscheiden.

## Ihr erstes Modell konstruieren: k-nächste-Nachbarn

Nun können wir beginnen, das eigentliche maschinelle Lernmodell zu konstruieren. In scikit-learn gibt es viele Klassifikationsalgorithmen, die wir einsetzen könnten. Hier verwenden wir den leicht verständlichen *k*-nächste-Nachbarn-Klassifikator. Der Aufbau des Modells erfordert nur das Speichern der Trainingsdaten. Um für einen neuen Datenpunkt eine Vorhersage zu treffen, sucht der Algorithmus den Punkt im Trainingsdatensatz, der dem neuen Punkt am nächsten ist. Dann wird das Label dieses Trainingsdatenpunktes dem neuen Datenpunkt zugewiesen.

Das *k* im Namen *k*-nächste-Nachbarn deutet darauf hin, dass wir statt nur den nächsten Nachbarn zu berücksichtigen, auch eine festgelegte Anzahl von *k* Nachbarn aus den Trainingsdaten verwenden könnten (beispielsweise die nächsten drei oder fünf Nachbarn). Dann können wir mit der mehrheitlich vertretenen Kategorie unter diesen Nachbarn eine Vorhersage treffen. Wir werden uns hiermit in Kapitel 2 eingehender beschäftigen; im Moment verwenden wir nur einen einzelnen Nachbarn.

Sämtliche maschinellen Lernmodelle in scikit-learn sind als eigene Klassen implementiert, die wir als Estimator-Klassen bezeichnen. Der *k*-nächste-Nachbarn-Algorithmus zur Klassifikation ist in der Klasse `KNeighborsClassifier` im Modul `neighbors` implementiert. Bevor wir das Modell verwenden können, müssen wir die Klasse zu einem Objekt instanziiieren. An dieser Stelle können wir die Parameter des Modells festlegen. Der wichtigste Parameter des `KNeighborsClassifier` ist die Anzahl der Nachbarn, die wir auf 1 festlegen:

**In[24]:**

```
from sklearn.neighbors import KNeighborsClassifier  
knn = KNeighborsClassifier(n_neighbors=1)
```

Das Objekt `knn` enthält den Algorithmus zum Konstruieren des Modells aus Trainingsdaten und den Algorithmus für die Vorhersage mit neuen Datenpunkten. Das Objekt enthält außerdem die aus den Trainingsdaten abgeleiteten Informationen. Im Falle des `KNeighborsClassifier` wird einfach nur der Trainingsdatensatz gespeichert.

Um auf dem Trainingsdatensatz ein Modell aufzubauen, rufen wir die Methode `fit` des Objekts `knn` auf. Als Argumente übergeben wir das NumPy-Array `X_train` mit den Trainingsdaten und das NumPy-Array `y_train` mit den entsprechenden Labels aus dem Trainingsdatensatz:

**In[25]:**

```
knn.fit(X_train, y_train)
```

**Out[25]:**

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',  
metric_params=None, n_jobs=1, n_neighbors=1, p=2,  
weights='uniform')
```

Die Methode `fit` gibt das Objekt `knn` selbst zurück (und verändert dieses intern), sodass wir eine Repräsentation unseres Klassifikators als String erhalten. Die Repräsentation zeigt uns die im Modell eingestellten Parameter. Beinahe alle sind Standardwerte, Sie sehen aber auch den von uns übergebenen Parameter `n_neighbors=1`. Die meisten Modelle in scikit-learn enthalten viele Parameter, die meisten davon sind aber entweder zur Optimierung der Laufzeit oder für besondere Anwendungsfälle gedacht. Sie müssen sich über die übrigen gezeigten Parameter keine großen Gedanken machen. Die Ausgabe eines scikit-learn-Modells kann zu sehr langen Strings führen, aber lassen Sie sich davon nicht verunsichern. Wir werden uns in Kapitel 2 mit allen wichtigen Parametern beschäftigen. Im weiteren Verlauf dieses Buches werden wir die Ausgabe von `fit` nicht abdrucken, da sie keinerlei neue Information enthält.

## Vorhersagen treffen

Wir können mit diesem Modell nun Vorhersagen für neue Daten treffen, für die wir die korrekte Kategorie nicht kennen. Nehmen wir an, wir haben eine Iris mit 5 cm langen und 2.9 cm breiten Kelchblättern sowie 1 cm langen und 0.2 cm breiten Kronblättern in freier Wildbahn angetroffen. Welche Iris-Spezies ist dies? Wir können diese Daten in einem NumPy-Array ablegen, dessen Abmessungen sich aus der Anzahl der Datenpunkte (1) und der Anzahl der Merkmale (4) ergeben:

**In[26]:**

```
X_new = np.array([[5, 2.9, 1, 0.2]])
print("X_new.shape: {}".format(X_new.shape))
```

**Out[26]:**

```
X_new.shape: (1, 4)
```

Wir haben die Messungen dieser einzelnen Blüte in einem zweidimensionalem NumPy-Array abgelegt, da scikit-learn für Daten stets ein zweidimensionales Array erwartet.

Um eine Vorhersage zu treffen, rufen wir die Methode `predict` des Objekts `knn` auf:

**In[27]:**

```
prediction = knn.predict(X_new)
print("Vorhersage: {}".format(prediction))
print("Vorhergesagter Name: {}".format(
    iris_dataset['target_names'][prediction]))
```

**Out[27]:**

```
Vorhersage: [0]
Vorhergesagter Name: ['setosa']
```

Unser Modell sagt vorher, dass diese neue Iris zur Kategorie 0 und damit zur Spezies *setosa* gehört. Aber woher wissen wir, dass wir unserem Modell trauen können? Wir kennen die korrekte Spezies für diesen Datenpunkt nicht, weswegen wir das Modell ja überhaupt gebaut haben!

## Evaluieren des Modells

An dieser Stelle kommt der weiter oben erstellte Testdatensatz wieder ins Spiel. Diese Daten wurden beim Erstellen des Modells nicht verwendet, wir kennen aber für jede Iris im Testdatensatz die korrekte Spezies.

Deshalb können wir für jede Iris im Testdatensatz eine Vorhersage treffen und mit dem korrekten Label (der bekannten Spezies) vergleichen. Wir können messen, wie gut das Modell funktioniert, indem wir die *Genauigkeit* berechnen, also den Anteil der Blüten mit korrekt vorhergesagter Spezies:

**In[28]:**

```
y_pred = knn.predict(X_test)
print("Vorhersagen für den Testdatensatz:\n {}".format(y_pred))
```

**Out[28]:**

```
Vorhersagen für den Testdatensatz:
[2 1 0 2 0 2 0 1 1 1 2 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1 0 2 2 1 0 2]
```

**In[29]:**

```
print("Genauigkeit auf den Testdaten: {:.2f}".format(np.mean(y_pred == y_test)))
```

**Out[29]:**

```
Genauigkeit auf den Testdaten: 0.97
```

Wir können auch die Methode `score` des `knn`-Objekts verwenden, um die Genauigkeit auf den Testdaten zu berechnen:

**In[30]:**

```
print("Genauigkeit auf den Testdaten: {:.2f}".format(knn.score(X_test, y_test)))
```

**Out[30]:**

```
Genauigkeit auf den Testdaten: 0.97
```

Bei diesem Modell beträgt die Genauigkeit auf den Testdaten etwa 0.97. Das bedeutet, dass wir für 97 % der Irisblüten im Testdatensatz die richtige Vorhersage treffen. Mit einigen mathematischen Annahmen können wir davon ausgehen, dass unser Modell bei neuen Irisblüten in 97 % der Fälle richtig liegt. Für die Anwendung unserer Hobbybotanikerin bedeutet diese hohe Genauigkeit, dass das Modell vertrauenswürdig genug für eine praktische Anwendung ist. In späteren Kapiteln werden wir diskutieren, wie wir die Leistung eines Modells verbessern können und welche Fallstricke es dabei gibt.

## Zusammenfassung und Ausblick

Fassen wir zusammen, was wir in diesem Kapitel gelernt haben. Wir haben mit einer kurzen Einführung in maschinelles Lernen und mögliche Anwendungen begonnen, anschließend den Unterschied zwischen überwachtem und unüber-

wachtem Lernen erläutert und uns einen Überblick über die in diesem Buch verwendeten Werkzeuge verschafft. Dann haben wir die Vorhersage der Spezies einer Iris aus den physikalischen Abmessungen ihrer Blüten als Aufgabe formuliert. Wir haben einen von Experten mit der korrekten Spezies annotierten Satz von Messdaten zum Aufbau eines Modells verwendet, wodurch dies eine überwachte Lernaufgabe wurde. Es gab drei mögliche Spezies, *setosa*, *versicolor* und *virginica*, wodurch dies zu einer Klassifikationsaufgabe mit drei Kategorien wurde. Bei dieser Klassifikationsaufgabe nennt man die möglichen Spezies *Kategorien*, die Spezies einer einzelnen Iris nennt man *Label*.

Der Iris-Datensatz besteht aus zwei NumPy-Arrays: Eines enthält die Daten, die wir in scikit-learn als *X* bezeichnen, das andere enthält die korrekten oder erwünschten Ausgabewerte, die wir als *y* bezeichnen. Das Array *X* ist ein zweidimensionales Array mit den Merkmalen, wobei jede Zeile einem Datenpunkt und jede Spalte einem Merkmal entspricht. Das Array *y* ist ein eindimensionales Array, das hier die Kategoriebezeichnung für jede Blüte als Integerzahl zwischen 0 und 2 enthält.

Wir haben unseren Datensatz in *Trainingsdaten* unterteilt, mit denen wir unser Modell aufbauen, und in *Testdaten*, mit denen wir die Fähigkeit unseres Modells zur Verallgemeinerung auf neue, vorher nicht bekannte Daten überprüfen.

Wir entschieden uns zur Klassifikation für den *k*-nächste-Nachbarn-Algorithmus, bei dem wir für einen neuen Datenpunkt eine Vorhersage auf Grundlage seines nächsten Nachbarn im Trainingsdatensatz treffen. Dieser ist in der Klasse *KNeighborsClassifier* implementiert, die sowohl den Algorithmus zum Aufbau des Modells als auch den Algorithmus zur Vorhersage mithilfe des Modells enthält. Wir haben eine Instanz dieser Klasse gebildet und dabei Parameter festgelegt. Anschließend haben wir das Modell durch Aufruf der Methode *fit* konstruiert und dazu die Trainingsdaten (*X\_train*) und Labels (*y\_train*) als Parameter übergeben. Zum Evaluieren des Modells haben wir über die Methode *score* die Genauigkeit des Modells berechnet. Die Methode *score* haben wir auch auf den Testdatensatz und die zugehörigen Labels angewendet und dabei herausgefunden, dass unser Modell zu 97 % genau ist. Unser Modell liegt also in 97 % der Fälle im Testdatensatz richtig.

Dies hat uns zuversichtlich genug gemacht, das Modell auf neue Daten anzuwenden (in diesem Beispiel Messungen neuer Blüten) und darauf zu vertrauen, dass unser Modell auch hier in 97 % der Fälle eine korrekte Vorhersage trifft.

Hier ist der nötige Code für die gesamte Prozedur zum Trainieren und Auswerten des Modells zusammengefasst:

**In[31]:**

```
X_train, X_test, y_train, y_test = train_test_split(  
    iris_dataset['data'], iris_dataset['target'], random_state=0)  
  
knn = KNeighborsClassifier(n_neighbors=1)  
knn.fit(X_train, y_train)  
  
print("Genauigkeit auf den Testdaten: {:.2f}".format(knn.score(X_test, y_test)))
```

**Out[31]:**

Genauigkeit auf den Testdaten: 0.97

Dieser Codeschnipsel enthält den wesentlichen Code zum Anwenden eines beliebigen maschinellen Lernalgorithmus mit scikit-learn. Die Methoden `fit`, `predict` und `score` bieten eine gemeinsame Schnittstelle für die überwachten Modelle in scikit-learn. Mit den in diesem Kapitel eingeführten Begriffen können Sie diese Modelle auf viele maschinelle Lernaufgaben anwenden. Im nächsten Kapitel werden wir tiefer ins Detail gehen und die unterschiedlichen Arten überwachter Modelle in scikit-learn sowie ihre erfolgreiche Anwendung kennenlernen.



## KAPITEL 2

# Überwachtes Lernen

Wie bereits erwähnt, ist überwachtes maschinelles Lernen eine der am häufigsten eingesetzten und erfolgreichsten Arten maschinellen Lernens. In diesem Kapitel werden wir überwachtes Lernen im Detail beschreiben und mehrere beliebte Algorithmen für überwachtes Lernen erklären. Wir haben in Kapitel 1 bereits einen Anwendungsfall für maschinelles Lernen kennengelernt: mehrere Spezies der Gattung Iris anhand der Abmessungen ihrer Blütenblätter zu klassifizieren.

Rufen wir uns in Erinnerung, dass überwachtes Lernen immer dann einsetzbar ist, wenn wir aus der Eingabe ein bestimmtes Ergebnis vorhersagen möchten und Paare für Ein- und Ausgabewerte haben. Anhand dieser Eingabe-Ausgabe-Paare erstellen wir ein maschinelles Lernmodell, das unsere Trainingsdaten repräsentiert. Unser Ziel ist, Vorhersagen für neue, im Voraus unbekannte Daten zu treffen. Überwachtes Lernen erfordert oft Handarbeit beim Zusammenstellen des Trainingsdatensatzes. Die folgenden Arbeitsschritte sind aber automatisierbar und beschleunigen dadurch ansonsten mühevolle oder undurchführbare Aufgaben.

## Klassifikation und Regression

Es gibt zwei wesentliche Arten von Aufgaben für überwachtes Lernen, nämlich *Klassifikation* und *Regression*.

Das Ziel bei der Klassifikation ist es, eine *Klassenbezeichnung* vorherzusagen, also eine Auswahl aus einer vorgegebenen Liste von Möglichkeiten zu treffen. In Kapitel 1 haben wir als Beispiel Irisblüten einer von drei möglichen Spezies zugeordnet. Klassifikationsverfahren unterteilt man gelegentlich in *binäre Klassifikation*, den Spezialfall der Unterscheidung zwischen genau zwei Klassen, und *Klassifikation mehrerer Klassen*, bei der es mehr als zwei Klassen gibt. Sie können sich binäre Klassifizierung als den Versuch vorstellen, eine Ja/Nein-Frage zu beantworten. E-Mails als Spam oder Nicht-Spam einzustufen, ist ein Beispiel für ein binäres Klassifikationsproblem. Die in diesem binären Klassifikationsproblem beantwortete Ja/Nein-Frage wäre: »Ist diese E-Mail Spam?«

Bei der binären Klassifikation bezeichnen wir die eine Klasse häufig als *positive* Klasse und die andere als *negative* Klasse. Dabei ist mit positiv nicht der Nutzen oder Wert gemeint, sondern das Ziel der Untersuchung. Wenn wir also nach Spam suchen, könnte mit »positiv« auch Spam als Klasse gemeint sein. Welche der beiden Klassen positiv genannt wird, ist oft eine subjektive Entscheidung und hängt vom Forschungsgegenstand ab.

Das Iris-Beispiel hingegen ist ein Beispiel für eine Klassifikationsaufgabe mit mehreren Klassen. Ein weiteres Beispiel wäre, die Sprache einer Webseite aus dem Text auf der Seite vorherzusagen. Die Klassen wären dabei eine vorgegebene Liste möglicher Sprachen.

Bei Regressionsproblemen ist das Ziel, eine kontinuierliche Größe oder *Fließkommazahl* im Programmierjargon (oder *Realzahl* in der Mathematik) vorherzusagen. Das jährliche Einkommen einer Person aus Bildungsgrad, Alter und Wohnort vorherzusagen, ist ein Beispiel für ein Regressionsproblem. Beim Vorhersagen des Einkommens ist die vorhergesagte Größe eine *Anzahl*. Dabei ist jede Zahl in einem bestimmten Bereich möglich. Ein weiteres Beispiel ist, den Ernteertrag einer Farm aus gegebenen Eigenschaften wie vorangegangenen Ernten, dem Wetter und der Anzahl Arbeiter auf dem Hof vorherzusagen. Auch hier ist der Ernteertrag eine beliebige Zahl.

Klassifikations- und Regressionsaufgaben lassen sich leicht anhand der Frage unterscheiden, ob eine Art von Kontinuität bei der Ausgabe vorliegt. Wenn es Kontinuität beim Ausgabewert gibt, handelt es sich um eine Regressionsaufgabe. Betrachten wir die Vorhersage des jährlichen Einkommens. Es gibt ganz klar Kontinuität bei den Ausgabewerten. Ob eine Person 40000 Euro oder 40001 Euro im Jahr verdient, macht keinen bemerkenswerten Unterschied, auch wenn dies unterschiedliche Beträge sind; es kümmert uns nicht besonders, ob unser Algorithmus 39999 Euro oder 40001 Euro vorhersagt, wenn das korrekte Ergebnis 40000 Euro wäre.

Dagegen gibt es bei der Aufgabe, die Sprache einer Webseite zu erkennen (einer Klassifikationsaufgabe), keine Abstufungen. Eine Webseite ist entweder in der einen Sprache oder in einer anderen. Es gibt keine Kontinuität zwischen Sprachen. Es gibt keine Sprache, die *zwischen* Englisch und Französisch liegt.<sup>1</sup>

## Verallgemeinerung, Overfitting und Underfitting

Beim überwachten Lernen möchte man mithilfe der Trainingsdaten ein Modell entwickeln und dann genaue Vorhersagen für neue, zuvor unbekannte Daten treffen, die ähnliche Charakteristiken haben wie die Trainingsdaten. Wenn ein Modell zu

---

<sup>1</sup> Wir bitten die Sprachwissenschaftler diese vereinfachte Darstellung von Sprachen als klar abgegrenzte und feste Entitäten zu entschuldigen.

genauen Vorhersagen auf unbekannte Daten in der Lage ist, sagen wir, dass das Modell von den Trainingsdaten auf die Testdaten *verallgemeinern* kann. Wir möchten also ein Modell konstruieren, dass so genau wie möglich verallgemeinert.

Normalerweise entwickelt man ein Modell, um auf den Trainingsdaten genaue Vorhersagen treffen zu können. Wenn die Trainings- und Testdaten zueinander ausreichend ähnlich sind, können wir erwarten, dass es auch auf den Testdaten genau funktioniert. Es gibt allerdings einige Fälle, in denen wir damit scheitern. Wenn wir beispielsweise sehr komplexe Modelle zulassen, können wir praktisch beliebig genaue Vorhersagen auf den Trainingsdaten treffen.

Betrachten wir ein Gedankenbeispiel, um diesen Punkt zu verdeutlichen. Nehmen wir an, ein unerfahrener Data Scientist möchte vorhersagen, ob ein Kunde ein Boot kaufen wird, und verfügt dazu über Daten vergangener Bootskäufer und nicht interessierter Kunden.<sup>2</sup> Das Ziel ist, Werbe-E-Mails an Kunden mit guten Verkaufsaussichten zu verschicken, die weniger vielversprechenden aber nicht zu belästigen.

Nehmen wir an, wir hätten die in Tabelle 2-1 gezeigten Kundendaten.

*Tabelle 2-1: Beispieldaten über Kunden*

Alter	Anzahl Autos	Hausbesitzer	Anzahl Kinder	Lebensstand	Hundebesitzer	hat Boot gekauft
66	1	ja	2	verwitwet	nein	ja
52	2	ja	3	verheiratet	nein	ja
22	0	nein	0	verheiratet	ja	nein
25	1	nein	1	ledig	nein	nein
44	0	nein	2	geschieden	ja	nein
39	1	ja	2	verheiratet	ja	nein
26	1	nein	2	ledig	nein	nein
40	3	ja	1	verheiratet	ja	nein
53	2	ja	2	geschieden	nein	ja
64	2	ja	3	geschieden	nein	nein
58	2	ja	2	verheiratet	ja	ja
33	1	nein	1	ledig	nein	nein

Nachdem er sich diese Daten eine Weile angesehen hat, entwickelt unser unerfahrener Data Scientist folgende Regel: »Wenn der Kunde älter als 45 Jahre ist, weniger als 3 Kinder hat oder nicht geschieden ist, dann möchte er ein Boot kaufen.« Auf die Frage, wie genau seine Regel ist, antwortet der Data Scientist: »Sie ist zu 100 %

---

<sup>2</sup> In der Realität ist dies eine schwierige Aufgabe. Obwohl wir wissen, dass die anderen Kunden bei uns kein Boot gekauft haben, könnten sie woanders eines gekauft haben oder auf einen Bootskauf in der Zukunft sparen.

genau!« Und in der Tat funktioniert diese Regel für die Daten in der Tabelle perfekt. Es gibt viele mögliche Regeln, mit denen wir perfekt erklären könnten, ob jemand in diesem Datensatz ein Boot kaufen möchte. Kein Alter kommt doppelt vor, wir könnten also sagen, dass Leute, die 66, 52, 53 oder 58 Jahre alt sind, ein Boot kaufen möchten, alle anderen dagegen nicht. Auch wenn wir für diese Daten gut funktionierende Regeln entwickeln können, müssen wir bedenken, dass wir die Vorhersagen nicht für diesen Datensatz treffen möchten; wir kennen die Antworten dieser Kunden bereits. Wir möchten wissen, ob *neue Kunden* voraussichtlich an einem Boot interessiert sind. Daher müssen wir eine Regel finden, die auch für neue Kunden gut funktioniert, und eine Genauigkeit von 100 % auf den Trainingsdaten hilft uns hier nicht weiter. Wir erwarten nicht, dass die von unserem Data Scientist entwickelte Regel besonders gut bei neuen Kunden funktioniert. Sie erscheint zu komplex und wird nicht von besonders vielen Daten gestützt. Beispielsweise hängt der Teil »oder nicht geschieden ist« von einem einzigen Kunden ab.

Das einzige Maß dafür, ob ein Algorithmus bei neuen Daten gut funktioniert, ist eine Auswertung auf den Testdaten. Allerdings erwarten wir intuitiv<sup>3</sup>, dass einfache Modelle besser verallgemeinern können. Wenn die Regel »Leute über 50 möchten ein Boot kaufen« hieße und diese Regel das Verhalten aller Kunden beschreiben würde, wäre diese Regel weitaus vertrauenswürdiger als die Regel, die zusätzlich zum Alter auch Kinder und den Lebensstand berücksichtigt. Aus diesem Grunde sind wir bestrebt, stets das einfachste Modell zu finden. Das Bauen eines Modells wie das des unerfahrenen Data Scientist, das für die Menge vorhandener Information zu komplex ist, nennt man *Overfitting*. Overfitting entsteht, wenn Sie ein Modell zu eng an die Eigenheiten der Trainingsdaten anpassen und dabei ein Modell erhalten, das zwar gut für die Trainingsdaten funktioniert, aber auf neue Daten nicht verallgemeinern kann. Wenn dagegen das Modell zu einfach ist – beispielsweise »Jeder, der ein Haus besitzt, kauft ein Boot« –, dann erfassen wir nicht alle Aspekte und die Variabilität der Daten, sodass das Modell selbst auf den Trainingsdaten schlecht abschneidet. Ein zu stark vereinfachendes Modell auszuwählen, nennt man *Underfitting*.

Je mehr Komplexität wir in unserem Modell zulassen, desto bessere Vorhersagen können wir für die Trainingsdaten treffen. Wenn allerdings unser Modell zu komplex wird, konzentriert es sich zu sehr auf jeden einzelnen Datenpunkt in den Trainingsdaten und verallgemeinert daher nicht so gut.

Es gibt eine goldene Mitte, die eine optimale Verallgemeinerung ergibt. Dieses Modell möchten wir finden.

In Abbildung 2-1 ist die Wechselbeziehung zwischen Overfitting und Underfitting dargestellt.

---

<sup>3</sup> und mit den richtigen Formeln auch beweisbar

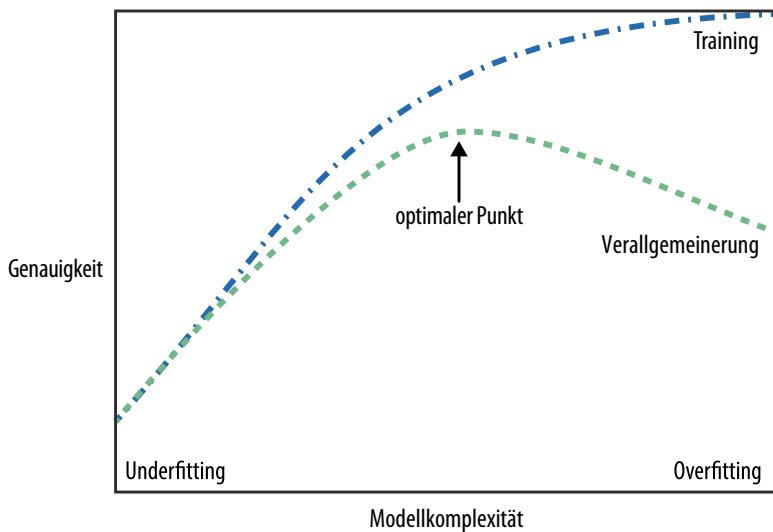


Abbildung 2-1: Wechselbeziehung zwischen Modellkomplexität gegenüber der Genauigkeit auf den Trainings- und Testdaten

## Zusammenhang zwischen Modellkomplexität und Größe des Datensatzes

Es ist wichtig zu betonen, dass die Komplexität des Modells eng mit der Variabilität der Eingabedaten im Trainingsdatensatz zusammenhängt: Je mehr Varianten die Datenpunkte im Datensatz enthalten, umso komplexer darf das Modell sein, ohne zu overfitten. Normalerweise führt das Sammeln von weiteren Datenpunkten zu mehr Varianten, sodass man mit größeren Datensätzen komplexere Modelle konstruieren kann. Allerdings hilft es nicht, die gleichen Daten einfach zu duplizieren oder sehr ähnliche Daten zu sammeln.

Bleiben wir beim Beispiel der Bootsverkäufe: Wenn wir 10000 weitere Kundendatensätze betrachten würden und auf alle die Regel zutrifft »Wenn der Kunde älter als 45 ist, weniger als 3 Kinder hat oder nicht geschieden ist, möchte er ein Boot kaufen«, dann könnten wir dieser Regel viel eher Glauben schenken, als wenn sie nur mit den 12 Einträgen in Tabelle 2-1 entwickelt wurde.

Mehr Daten zu haben und dementsprechend komplexere Modelle zu konstruieren, kann beim überwachten Lernen mitunter Wunder wirken. In diesem Buch konzentrieren wir uns auf Datensätze mit fester Größe. Im wirklichen Leben haben Sie oft Einfluss darauf, wie viele Daten Sie sammeln möchten. Dies kann nützlicher sein, als ausführlich an Ihrem Modell herumzudoktern. Unterschätzen Sie niemals die Kraft zusätzlicher Daten.

# Algorithmen zum überwachten Lernen

Wir werden im Folgenden die beliebtesten Algorithmen zum maschinellen Lernen betrachten und erklären, wie diese aus Daten lernen und wie sie Vorhersagen treffen. Wir werden ebenfalls besprechen, wie sich die Komplexität in den unterschiedlichen Modellen manifestiert, und geben einen Überblick über die Modellkonstruktion jedes Algorithmus. Wir werden die Stärken und Schwächen der Algorithmen untersuchen und auf welche Daten sie sich am besten einsetzen lassen. Wir werden auch die Bedeutung der wichtigsten Parameter und Optionen erklären.<sup>4</sup> Viele Algorithmen gibt es in einer Klassifikations- und einer Regressionsvariante, und wir werden uns mit beiden befassen.

Es ist nicht notwendig, sich die Beschreibung jedes Algorithmus im Detail durchzulesen, aber ein Verständnis der Modelle gibt Ihnen ein besseres Gefühl dafür, auf welch unterschiedliche Art und Weise maschinelle Lernverfahren funktionieren können. Dieses Kapitel soll als Nachschlagwerk dienen, und Sie können später hierher zurückkehren, wenn Sie sich über die Funktionsweise eines der Algorithmen nicht im Klaren sind.

## Einige Beispieldatensätze

Wir werden zum Veranschaulichen der Algorithmen mehrere Datensätze verwenden. Einige der Datensätze werden klein und synthetisch (also ausgedacht) sein, um bestimmte Aspekte der Algorithmen hervorzuheben. Andere größere Datensätze enthalten Beispiele aus der Wirklichkeit.

Ein Beispiel für einen synthetischen Datensatz zur Klassifikation mit zwei Kategorien ist der Datensatz `forge`, der zwei Merkmale aufweist. Der folgende Code generiert ein Streudiagramm (Abbildung 2-2), in dem alle Punkte aus diesem Datensatz dargestellt sind. Das Diagramm trägt das erste Merkmal auf der x-Achse, das zweite auf der y-Achse auf. Wie bei Streudiagrammen üblich, ist jeder Datenpunkt als ein Symbol dargestellt. Farbe und Form der Punkte zeigen ihre Kategorie an:

**In[1]:**

```
# Datensatz generieren
X, y = mglearn.datasets.make_forge()
# Datensatz zeichnen
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.legend(["Kategorie 0", "Kategorie 1"], loc=4)
plt.xlabel("Erstes Merkmal")
plt.ylabel("Zweites Merkmal")
print("X.shape: {}".format(X.shape))
```

---

<sup>4</sup> Sie alle zu diskutieren, sprengt den Rahmen dieses Buches, und wir verweisen Sie zu Details auf die Dokumentation von scikit-learn (<http://scikit-learn.org/stable/documentation>).

**Out[1]:**

X.shape: (26, 2)

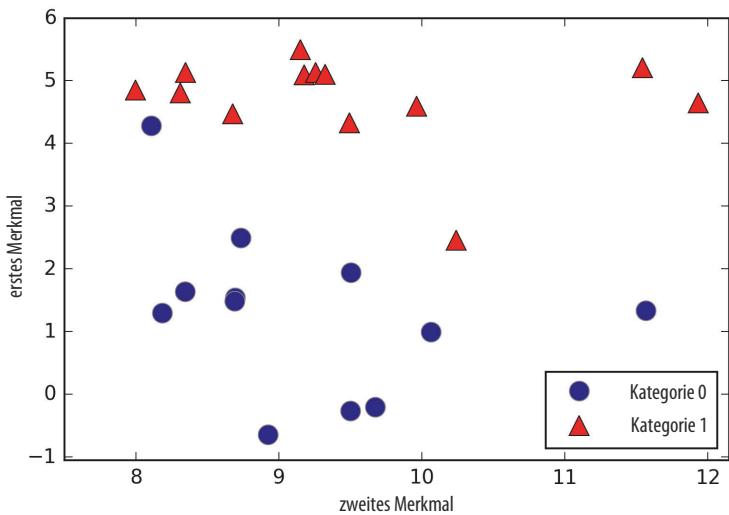


Abbildung 2-2: Streudiagramm für den Datensatz forge

Wie aus X.shape ersichtlich ist, besteht dieser Datensatz aus 26 Datenpunkten mit 2 Merkmalen.

Um Algorithmen zur Regression zu veranschaulichen, verwenden wir den künstlichen Datensatz wave. Der Datensatz wave besitzt ein einziges Merkmal als Eingabe und eine kontinuierliche Variable (oder *response*), die wir modellieren möchten. Das hier erzeigte Diagramm (Abbildung 2-3) stellt das eine Merkmal auf der x-Achse und die Zielgröße der Regression (die Ausgabe) auf der y-Achse dar:

**In[2]:**

```
X, y = mglearn.datasets.make_wave(n_samples=40)
plt.plot(X, y, 'o')
plt.ylim(-3, 3)
plt.xlabel("Merkmals")
plt.ylabel("Zielgröße")
```

Wir verwenden diese sehr einfachen Datensätze mit wenigen Dimensionen, weil sie sich leicht visualisieren lassen – eine gedruckte Seite hat nur zwei Dimensionen, daher sind Daten mit mehr als zwei Merkmalen schwer darzustellen. Schlussfolgerungen aus Datensätzen mit wenigen Merkmalen (oder *wenigen Dimensionen*) gelten nicht unbedingt für Datensätze mit vielen Merkmalen (oder *vielen Dimensionen*). Solange wir dies im Hinterkopf behalten, ist es sehr lehrreich, Algorithmen anhand von Datensätzen mit wenigen Dimensionen zu untersuchen.

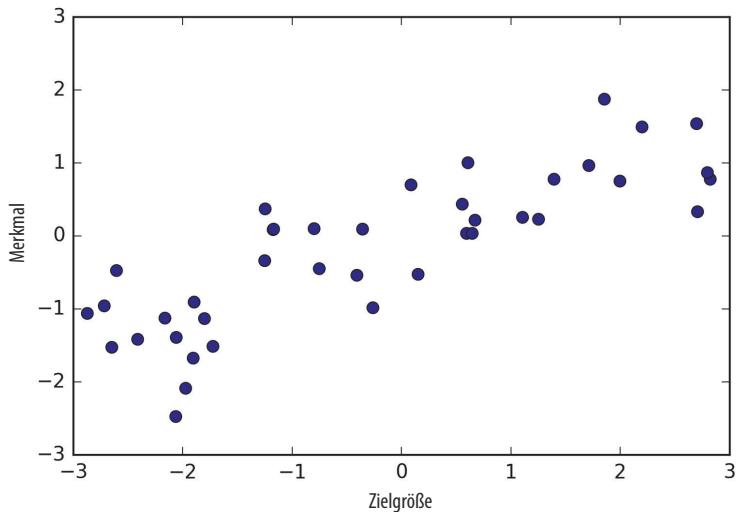


Abbildung 2-3: Diagramm für den Datensatz wave. Das Merkmal ist auf der x-Achse dargestellt, die Zielgröße der Regression auf der y-Achse

Wir werden diese kleinen, synthetischen Datensätze durch zwei echte Datensätze ergänzen, die bereits in scikit-learn enthalten sind. Einer davon ist der Wisconsin Brustkrebs-Datensatz (kurz cancer). In diesem sind klinische Vermessungen von Tumoren in Brustgewebe abgelegt. Jeder Tumor ist als »benigner« (harmloser Tumor) oder »maligner« (krebsartiger Tumor) beschriftet. Die Aufgabe besteht darin, aus der Vermessung des Gewebes vorherzusagen, ob ein Tumor bösartig ist.

Diese Daten lassen sich mit der Funktion `load_breast_cancer` aus scikit-learn laden:

**In[3]:**

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
print("cancer.keys(): \n{}".format(cancer.keys()))
```

**Out[3]:**

```
cancer.keys():
dict_keys(['feature_names', 'data', 'DESCR', 'target', 'target_names'])
```



Datensätze in scikit-learn sind für gewöhnlich als Objekte vom Typ `Bunch` abgelegt, die sowohl Informationen über den Datensatz als auch die Daten selbst enthalten. Alles, was Sie über `Bunch`-Objekte wissen müssen, ist, dass sich diese wie Dictionaries verhalten und Sie die Werte zusätzlich über einen Punkt ansprechen können (z. B. als `bunch.key` anstelle von `bunch['key']`).

Der Datensatz enthält 569 Datenpunkte mit jeweils 30 Merkmalen:

**In[4]:**

```
print("Abmessungen des Datensatzes cancer: {}".format(cancer.data.shape))
```

**Out[4]:**

```
Abmessungen des Datensatzes cancer: (569, 30)
```

Von diesen 569 Datenpunkten sind 212 als maligne und 357 als benigne markiert:

**In[5]:**

```
print("Anzahl Datenpunkte pro Kategorie:\n{}".format(
    {n: v for n, v in zip(cancer.target_names, np.bincount(cancer.target))}))
```

**Out[5]:**

```
Anzahl Datenpunkte pro Kategorie:
{'benign': 357, 'malignant': 212}
```

Um eine semantische Beschreibung der Merkmale zu erhalten, müssen wir uns das Attribut `feature_names` ansehen:

**In[6]:**

```
print("Namen der Merkmale:\n{}".format(cancer.feature_names))
```

**Out[6]:**

```
Namen der Merkmale:
['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']
```

Wenn Sie mehr über die Daten erfahren möchten, lesen Sie `cancer.DESCR`.

Wir werden auch einen echten Datensatz zur Regression verwenden, den Boston Immobilien-Datensatz. Die mit diesem Datensatz verbundene Aufgabe ist, den Median des Wertes von Häusern in einigen Gegenden von Boston in den 1970ern vorherzusagen und dazu Informationen wie die Kriminalitätsrate, die Nähe zum Charles River, zu Highways usw. zu verwenden. Der Datensatz enthält 506 Datenpunkte mit 13 Merkmalen:

**In[7]:**

```
from sklearn.datasets import load_boston
boston = load_boston()
print("Abmessungen der Daten: {}".format(boston.data.shape))
```

**Out[7]:**

```
Abmessungen der Daten: (506, 13)
```

Auch hier können Sie mehr über den Datensatz erfahren, indem Sie sich das Attribut `boston.DESCR` durchlesen. Wir werden diesen Datensatz für unsere Zwecke erweitern, indem wir nicht nur diese 13 Messwerte als Eingabe betrachten, sondern auch sämtliche Produkte zweier Merkmale (auch *Interaktionen* genannt). Anders gesagt, berücksichtigen wir nicht nur die Kriminalitätsrate und Entfernung zum Highway als Merkmale, sondern auch das Produkt von Kriminalitätsrate und Entfernung zum Highway. Solche abgeleiteten Merkmale ebenfalls zu berücksichtigen, nennt man *Merkmalsgenerierung*, mit dem wir uns ausführlicher in Kapitel 4 beschäftigen werden. Wir können diesen Datensatz mit der Funktion `load_extended_boston` laden:

**In[8]:**

```
X, y = mglearn.datasets.load_extended_boston()  
print("X.shape: {}".format(X.shape))
```

**Out[8]:**

```
X.shape: (506, 104)
```

Die resultierenden 104 Merkmale enthalten die 13 ursprünglichen Merkmale und die 91 möglichen Kombinationen zweier Merkmale aus diesen 13 (mit Zurücklegen).<sup>5</sup>

Wir werden diese Datensätze verwenden, um die Eigenschaften verschiedener maschineller Lernalgorithmen zu erklären und zu veranschaulichen. Aber stellen wir erst einmal die Algorithmen selbst vor. Schauen wir uns zunächst noch einmal den *k*-nächste-Nachbarn-Algorithmus (*k*-NN) an, den wir im vorigen Kapitel kennengelernt haben.

## k-nächste-Nachbarn

Der *k*-NN-Algorithmus ist der vermutlich einfachste Algorithmus für maschinelles Lernen. Die Konstruktion des Modells besteht lediglich aus dem Speichern des Trainingsdatensatzes. Um für einen neuen Datenpunkt eine Vorhersage zu treffen, sucht der Algorithmus den nächsten Punkt im Trainingsdatensatz, nämlich den »nächsten Nachbarn«.

### k-nächste-Nachbarn-Klassifikation

In der einfachsten Ausfertigung betrachtet der *k*-NN-Algorithmus genau einen nächsten Nachbarn, der der zum vorherzusagenden Punkt dichteste Datenpunkt aus den Trainingsdaten ist. Die Vorhersage ist dann einfach die für diesen Trainingsdatenpunkt bekannte Kategorie. In Abbildung 2-4 ist dies für eine Klassifikation auf dem Datensatz `forge` dargestellt:

---

<sup>5</sup> Das bedeutet 13 Interaktionen für das erste Merkmal plus 12 für das zweite (ohne das erste) plus 11 für das dritte usw.:  $13 + 12 + 11 + \dots + 1 = 91$

In[9]:

```
mglearn.plots.plot_knn_classification(n_neighbors=1)
```

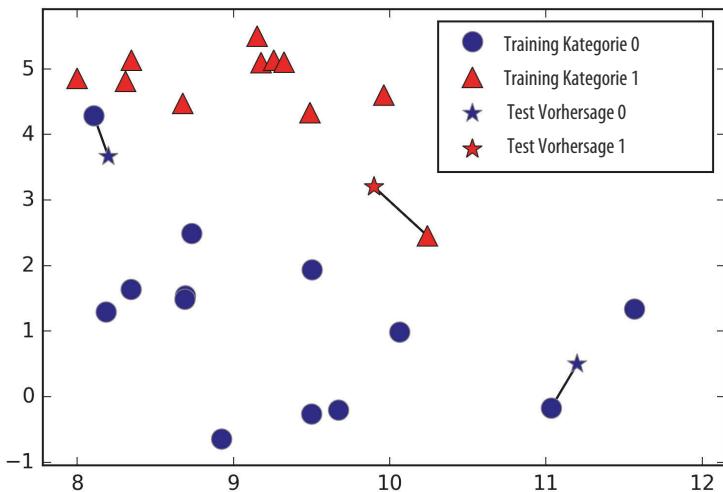


Abbildung 2-4: Vorhersagen mit einem 1-nächste-Nachbarn-Modell auf dem Datensatz forge

Wir haben hier drei neue Datenpunkte hinzufügt und als Sterne dargestellt. Zu jedem ist der nächste Punkt aus den Trainingsdaten markiert. Die Vorhersage des 1-nächste-Nachbarn-Algorithmus ist die Kategorie dieses Punktes (als Farbe des Kreuzes dargestellt).

Anstatt nur den nächsten Nachbarn zu berücksichtigen, können wir auch eine beliebige Anzahl von  $k$  Nachbarn betrachten. So entstand der Name des  $k$ -nächste-Nachbarn-Algorithmus. Wenn wir mehr als einen Nachbarn aufnehmen, findet eine *Abstimmung* statt, um den Ausgabewert festzulegen. Wir zählen für jeden Testdatenpunkt, wie viele Nachbarn der Kategorie 0 und wie viele der Kategorie 1 angehören. Dann ordnen wir die häufigere Kategorie als Ergebnis zu: Anders gesagt, entscheidet die unter den  $k$ -nächsten Nachbarn vorherrschende Kategorie. Das folgende Beispiel (Abbildung 2-5) verwendet die drei nächsten Nachbarn:

In[10]:

```
mglearn.plots.plot_knn_classification(n_neighbors=3)
```

Auch hier ist die Vorhersage als Farbe der Kreuze dargestellt. Sie sehen, dass die Vorhersage für den neuen Datenpunkt links oben nicht die gleiche ist wie die Vorhersage mit nur einem Nachbarpunkt.

Diese Veranschaulichung nimmt zwar eine binäre Klassifikation vor, die Methode lässt sich aber auf Datensätze mit einer beliebigen Anzahl Kategorien anwenden. Bei mehreren Kategorien zählen wir, wie viele Nachbarn zu jeder Kategorie gehören, und sagen wieder die dominierende Kategorie vorher.

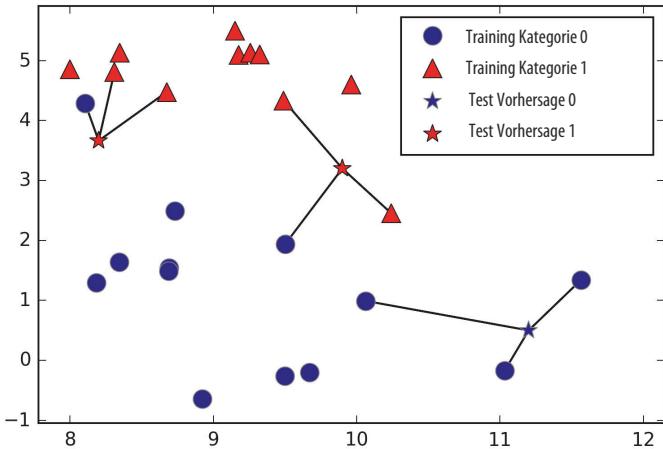


Abbildung 2-5: Vorhersage durch ein 3-nächste-Nachbarn-Modell auf dem Datensatz forge

Betrachten wir, wie wir den  $k$ -nächste-Nachbarn-Algorithmus mit scikit-learn verwenden können. Wir teilen zunächst unsere Daten in einen Trainings- und einen Testdatensatz auf, sodass wir die Fähigkeit zur Verallgemeinerung bewerten können, wie in Kapitel 1 besprochen:

**In[11]:**

```
from sklearn.model_selection import train_test_split
X, y = mglearn.datasets.make_forge()

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Als Nächstes importieren wir die Klasse und instanziiieren diese. An dieser Stelle können wir Parameter wie die Anzahl der Nachbarn festlegen. Hier setzen wir sie auf 3:

**In[12]:**

```
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(n_neighbors=3)
```

Nun passen wir den Klassifikator mit den Trainingsdaten an. Beim KNeighborsClassifier wird dabei der Datensatz intern abgespeichert, sodass sich bei der Vorhersage Nachbarn berechnen lassen:

**In[13]:**

```
clf.fit(X_train, y_train)
```

Um auf den Testdaten Vorhersagen vorzunehmen, rufen wir die Methode predict auf. Diese berechnet für jeden Datenpunkt im Testdatensatz die nächsten Nachbarn und findet die darunter dominierende Kategorie:

**In[14]:**

```
print("Vorhersage für die Testdaten: {}".format(clf.predict(X_test)))
```

**Out[14]:**

Vorhersage für die Testdaten: [1 0 1 0 1 0 0]

Um die Qualität der Verallgemeinerung unseres Modells auszuwerten, rufen wir die Methode score mit den Testdaten und deren Beschriftung auf:

**In[15]:**

```
print("Genauigkeit auf dem Testdatensatz: {:.2f}".format(clf.score(X_test, y_test)))
```

**Out[15]:**

Genauigkeit auf dem Testdatensatz: 0.86

Wir sehen, dass unser Modell etwa zu 86 % genau ist, das Modell konnte also für 86 % der Punkte im Testdatensatz die korrekte Kategorie vorhersagen.

### Analyse des KNeighborsClassifier

Bei zweidimensionalen Datensätzen können wir die Vorhersage aller Testdatenpunkte in der x-/y-Ebene darstellen. Wir färben die Ebene entsprechend der Kategorie ein, die einem Punkt an dieser Stelle zugeordnet werden würde. Damit können wir die *Entscheidungsgrenzen* sehen, die Region, an der der Algorithmus zwischen der Zuordnung von Kategorie 0 und Kategorie 1 umschaltet. Der folgende Code visualisiert die Entscheidungsgrenzen für einen, drei und neun Nachbarn in Abbildung 2-6:

**In[16]:**

```
fig, axes = plt.subplots(1, 3, figsize=(10, 3))

for n_neighbors, ax in zip([1, 3, 9], axes):
    # die Methode fit liefert das self-Objekt, sodass wir
    # in einer Zeile instanzieren und anpassen können
    clf = KNeighborsClassifier(n_neighbors=n_neighbors).fit(X, y)
    mglearn.plots.plot_2d_separator(clf, X, fill=True, eps=0.5, ax=ax, alpha=.4)
    mglearn.discrete_scatter(X[:, 0], X[:, 1], y, ax=ax)
    ax.set_title("{} Nachbar(n)".format(n_neighbors))
    ax.set_xlabel("Merkmal 0")
    ax.set_ylabel("Merkmal 1")
    axes[0].legend(loc=3)
```

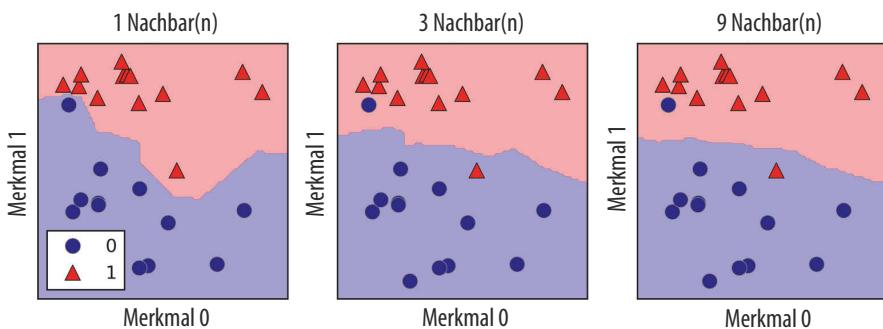


Abbildung 2-6: Vom nächsten-Nachbarn-Modell berechnete Entscheidungsgrenzen mit unterschiedlichen Werten für  $n_{neighbors}$

Wie Sie auf der linken Seite des Diagramms sehen können, folgt die mit einem einzelnen Nachbarn berechnete Entscheidungsgrenze dicht den Trainingsdaten. Mit mehr und mehr Nachbarn wird die Entscheidungsgrenze weicher. Eine weichere Grenze entspricht einem einfacheren Modell. Anders gesagt, ist ein Modell mit wenigen Nachbarn komplexer (wie auf der rechten Seite von Abbildung 2-1 gezeigt), und ein Modell mit vielen Nachbarn ist weniger komplex (wie auf der linken Seite von Abbildung 2-1 gezeigt). Im Extremfall, bei dem die Anzahl Nachbarn der Anzahl der Punkte im Trainingsdatensatz entspricht, hätte jeder Testpunkt die exakt gleichen Nachbarn (alle Trainingsdatenpunkte), und sämtliche Vorhersagen würden gleich ausfallen: die im Trainingsdatensatz häufigste Kategorie.

Untersuchen wir einmal, ob wir die weiter oben vorgestellte Beziehung zwischen der Komplexität eines Modells und dessen Fähigkeit zur Verallgemeinerung bestätigen können. Wir tun dies anhand des Brustkrebs-Datensatzes. Wir teilen zunächst den Datensatz in Trainings- und Testdaten auf. Dann werten wir die Vorhersagegüte für Trainings- und Testdaten mit unterschiedlichen Anzahlen von Nachbarn aus. Die Ergebnisse finden Sie in Abbildung 2-7:

In[17]:

```
from sklearn.datasets import load_breast_cancer

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=66)

training_accuracy = []
test_accuracy = []
# probiere Werte für n_neighbors von 1 bis 10 aus
neighbors_settings = range(1, 11)

for n_neighbors in neighbors_settings:
    # konstruiere das Modell
    clf = KNeighborsClassifier(n_neighbors=n_neighbors)
    clf.fit(X_train, y_train)
    # registriere die Genauigkeit auf den Trainingsdaten
    training_accuracy.append(clf.score(X_train, y_train))
    # registriere die Genauigkeit der Verallgemeinerung
    test_accuracy.append(clf.score(X_test, y_test))

plt.plot(neighbors_settings, training_accuracy, label="Genauigkeit Trainingsdaten")
plt.plot(neighbors_settings, test_accuracy, label="Genauigkeit Testdaten")
plt.ylabel("Genauigkeit")
plt.xlabel("n_neighbors")
plt.legend()
```

Das Diagramm zeigt auf der y-Achse die Genauigkeit der Vorhersage für Trainings- und Testdaten über dem Parameter `n_neighbors` auf der x-Achse. Auch wenn realistische Diagramme selten sehr weich verlaufen, können wir einige Charakteristiken von Overfitting und Underfitting beobachten (weil weniger Nachbarn einem kom-

plexeren Modell entsprechen, ist diese Darstellung im Vergleich zu Abbildung 2-1 seitenverkehrt).

Mit einem einzelnen nächsten Nachbarn ist die Vorhersage auf den Trainingsdaten perfekt. Wenn wir aber mehr Nachbarn berücksichtigen, wird das Modell einfacher, und die Genauigkeit bei den Trainingsdaten sinkt. Die Genauigkeit bei den Testdaten ist bei einem Nachbarn geringer als bei mehreren, was darauf hinweist, dass das Modell mit einem nächsten Nachbarn zu komplex ist. Berücksichtigen wir hingegen zehn Nachbarn, wird das Modell zu einfach, und die Leistung verschlechtert sich noch mehr. Die beste Vorhersagequalität erhalten wir in der Mitte bei etwa sechs Nachbarn. Es lohnt sich jedoch, die Skalierung des Diagramms zu berücksichtigen. Die schlechteste Vorhersage liegt bei einer Genauigkeit von etwa 88 %, was immer noch in Ordnung sein kann.

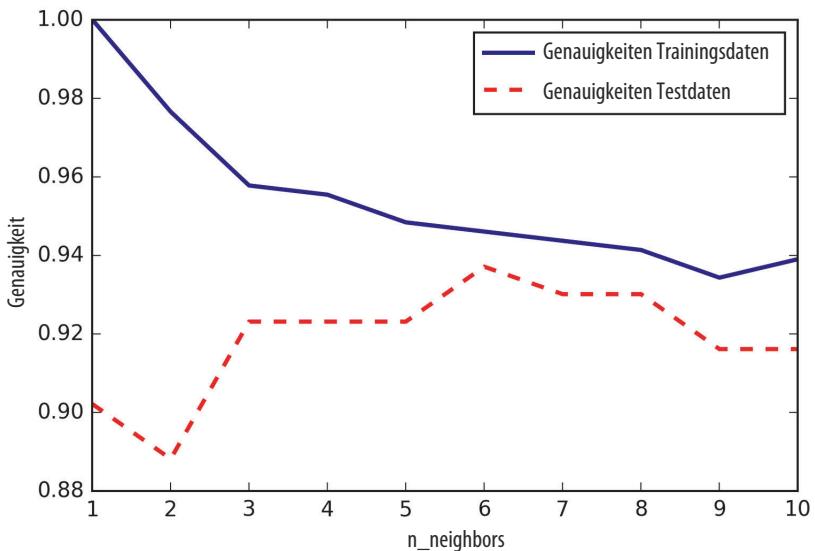


Abbildung 2-7: Vergleich der Genauigkeit von Trainings- und Testdatensatz in Abhängigkeit von  $n_{neighbors}$

### k-nächste-Nachbarn-Regression

Es gibt auch eine Regressions-Variante des  $k$ -nächste-Nachbarn-Algorithmus. Wir beginnen wieder mit einem nächsten Nachbarn, diesmal mit dem Datensatz `wave`. Wir haben auf der x-Achse drei Testdatenpunkte hinzugefügt. Die Vorhersage mit einem einzelnen Nachbarn ist einfach der Zielwert des nächsten Nachbarn. Diese sind in Abbildung 2-8 als Sterne dargestellt:

In[18]:

```
mglearn.plots.plot_knn_regression(n_neighbors=1)
```

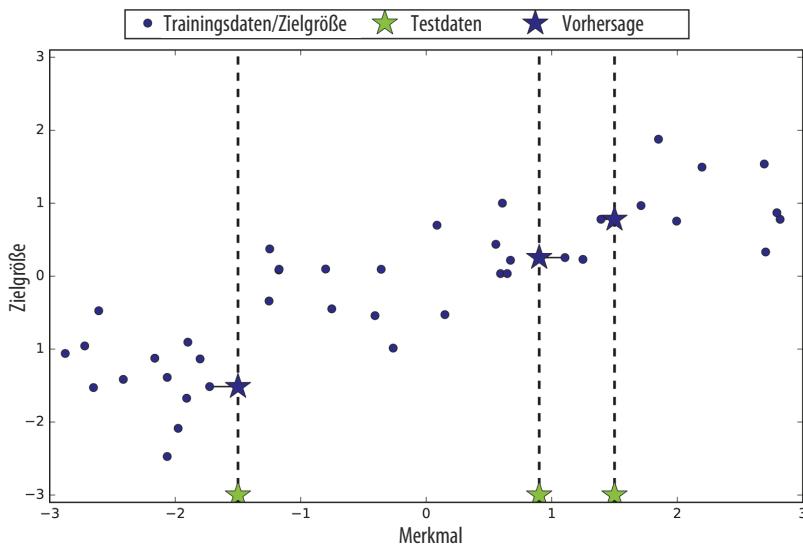


Abbildung 2-8: Vorhersage mit 1-nächster-Nachbarn-Regression auf dem Datensatz wave

Wir können wieder mehr als einen nächsten Nachbarn zur Regression einsetzen. Bei mehreren nächsten Nachbarn entspricht die Vorhersage dem Durchschnitt oder Mittelwert der relevanten Nachbarn (Abbildung 2-9):

In[19]:

```
mglearn.plots.plot_knn_regression(n_neighbors=3)
```

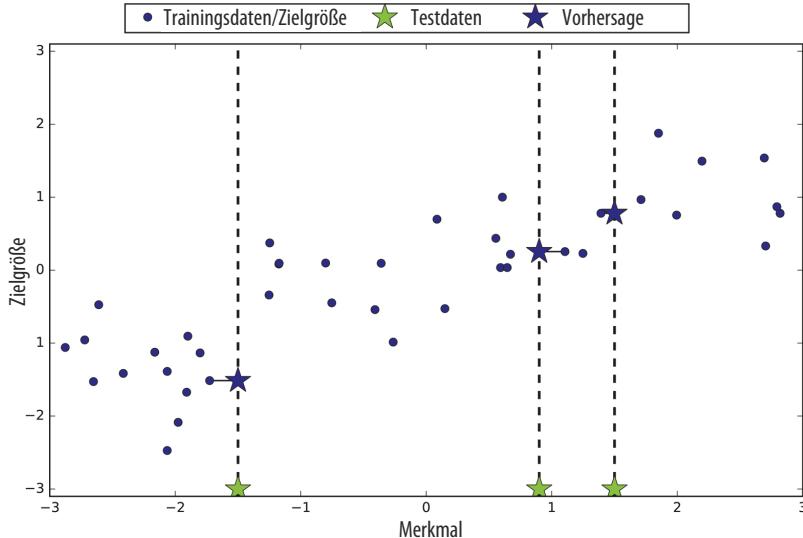


Abbildung 2-9: Vorhersagen mit 3-nächste-Nachbarn-Regression auf dem Datensatz wave

Der  $k$ -nächste-Nachbarn-Algorithmus ist in der Klasse `KNeighborsRegressor` in `scikit-learn` implementiert. Er lässt sich ähnlich zum `KNeighborsClassifier` einsetzen:

**In[20]:**

```
from sklearn.neighbors import KNeighborsRegressor

X, y = mglearn.datasets.make_wave(n_samples=40)

# teile den Datensatz wave in Trainings- und Testdaten auf
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

# instanziere das Modell und setze die Anzahl Nachbarn auf 3
reg = KNeighborsRegressor(n_neighbors=3)
# passe das Modell mit Trainingsdaten und Zielwerten an
reg.fit(X_train, y_train)
```

Nun können wir mit den Testdaten Vorhersagen treffen:

**In[21]:**

```
print("Vorhersagen auf dem Testdatensatz:\n{}".format(reg.predict(X_test)))
```

**Out[21]:**

```
Vorhersagen auf dem Testdatensatz:  
[-0.054  0.357  1.137 -1.894 -1.139 -1.631  0.357  0.912 -0.447 -1.139]
```

Wir können das Modell auch mithilfe der Methode `score` auswerten, die für Regressoren den  $R^2$ -Wert liefert. Der  $R^2$ -Wert, auch Determinationskoeffizient genannt, ist ein Bestimmtheitsmaß für die Vorhersagequalität eines Regressionsmodells und beträgt zwischen 0 und 1. Ein Wert von 1 entspricht einer perfekten Vorhersage, ein Wert von 0 entspricht einem konstanten Modell, das einfach nur den Mittelwert der Trainingsdaten `y_train` vorhersagt:

**In[22]:**

```
print("Testdatensatz R^2: {:.2f}".format(reg.score(X_test, y_test)))
```

**Out[22]:**

```
Testdatensatz R^2: 0.83
```

In unserem Beispiel beträgt der Wert 0.83, was einem recht gut angepassten Modell entspricht.

## Analyse des `KNeighborsRegressor`

Bei unserem eindimensionalen Datensatz können wir sehen, wie sich die Vorhersagen für alle möglichen Merkmalswerte verhalten (Abbildung 2-10). Dazu erstellen wir uns einen Testdatensatz aus vielen Punkten entlang einer Linie:

**In[23]:**

```
fig, axes = plt.subplots(1, 3, figsize=(15, 4))
# erstelle 1000 Datenpunkte in gleichem Abstand zwischen -3 und 3
```

```

line = np.linspace(-3, 3, 1000).reshape(-1, 1)
for n_neighbors, ax in zip([1, 3, 9], axes):
    # trifft Vorhersagen mit 1, 3 oder 9 Nachbarn
    reg = KNeighborsRegressor(n_neighbors=n_neighbors)
    reg.fit(X_train, y_train)
    ax.plot(line, reg.predict(line))
    ax.plot(X_train, y_train, '^', c=mlearn.cm2(0), markersize=8)
    ax.plot(X_test, y_test, 'v', c=mlearn.cm2(1), markersize=8)

    ax.set_title(
        "{} Nachbar(n)\n Score Training: {:.2f} Score Test: {:.2f}".format(
            n_neighbors, reg.score(X_train, y_train),
            reg.score(X_test, y_test)))
    ax.set_xlabel("Merkmale")
    ax.set_ylabel("Zielgröße")
    axes[0].legend(["Vorhersagen des Modells", "Trainingsdaten/Ziel",
                    "Testdaten/Ziel"], loc="best")

```

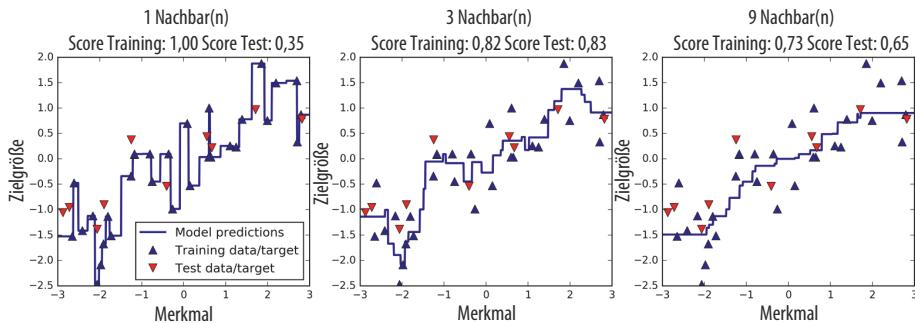


Abbildung 2-10: Vergleich der Vorhersagen der nächste-Nachbarn-Regression mit unterschiedlichen Werten von `n_neighbors`

Wie wir aus dem Diagramm sehen können, hat bei nur einem Nachbarn jeder Punkt in den Trainingsdaten einen offensichtlichen Einfluss auf die Vorhersage. Die vorhergesagten Werte durchlaufen sämtliche Datenpunkte. Dadurch ergibt sich eine sehr unstete Vorhersage. Das Berücksichtigen weiterer Nachbarn führt zu weicheren Vorhersagen, diese passen jedoch nicht so gut zu den Trainingsdaten.

## Stärken, Schwächen und Parameter

Im Prinzip besitzt der Klassifikator `KNeighbors` zwei wichtige Parameter: die Anzahl der Nachbarn und das Maß für die Distanz zwischen Datenpunkten. In der Praxis funktioniert eine geringe Anzahl Nachbarn wie drei oder fünf oft gut, aber Sie sollten diesen Parameter unbedingt genau einstellen. Die Wahl des richtigen Distanzmaßes liegt etwas außerhalb des Fokus dieses Buches. Standardmäßig wird der euklidische Abstand verwendet, der in vielen Fällen gut funktioniert.

Eine der Stärken des *k*-NN-Verfahrens ist, dass das Modell sehr leicht zu verstehen ist und oft ohne große Anpassungen eine recht anständige Vorhersagequalität liefert. Mit diesem Algorithmus lässt sich eine Grundlinie etablieren, bevor man fortgeschrit-

tenere Techniken ausprobiert. Ein nächste-Nachbarn-Modell lässt sich für gewöhnlich sehr schnell erstellen, aber wenn Ihr Trainingsdatensatz sehr groß ist (viele Merkmale oder viele Datenpunkte), kann die Vorhersage langsam sein. Beim  $k$ -NN-Algorithmus ist eine Vorverarbeitung der Daten wichtig (siehe Kapitel 3). Dieses Verfahren schneidet bei Datensätzen mit vielen Merkmalen (Hunderte oder mehr) meist nicht so gut ab und ist besonders schlecht für Datensätze geeignet, bei denen viele Merkmale oft den Wert 0 enthalten (sogenannte *dünn besetzte Datensätze*).

Obwohl der  $k$ -nächste-Nachbarn-Algorithmus leicht verständlich ist, wird er in der Praxis nicht so häufig eingesetzt, da die Vorhersage langsam ist und er nicht gut mit vielen Merkmalen umgehen kann. Die Methode, die wir als Nächstes vorstellen werden, hat keinen dieser Nachteile.

## Lineare Modelle

Lineare Modelle sind eine in der Praxis weitverbreitete Familie von Modellen, die in den letzten Jahrzehnten intensiv untersucht wurden. Ihre Wurzeln reichen über 100 Jahre zurück. Lineare Modelle treffen mithilfe einer *linearen Funktion* der Eingabemerkmale eine Vorhersage, wie wir gleich sehen werden.

### Lineare Modelle zur Regression

Die allgemeine Vorhersageformel zur Regression mit einem linearen Modell lautet:

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \dots + w[p] * x[p] + b$$

Dabei sind  $x[0]$  bis  $x[p]$  die Merkmale (in diesem Beispiel ist  $p+1$  die Anzahl der Merkmale) eines einzelnen Datenpunktes,  $w$  und  $b$  sind erlernte Parameter des Modells, und  $\hat{y}$  ist die Vorhersage durch das Modell. Bei einem Datensatz mit einem einzelnen Merkmal lautet die Formel:

$$\hat{y} = w[0] * x[0] + b$$

Aus der Schule erinnern Sie sich vielleicht daran, dass dies die Gleichung für eine Gerade ist. Dabei ist  $w[0]$  die Steigung, und  $b$  ist der y-Achsenabschnitt. Bei weiteren Merkmalen enthält  $w$  die Steigungen entlang jeder Merkmalsachse. Alternativ können Sie sich den vorhergesagten Wert als gewichtete Summe der Eingabemerkmale vorstellen, wobei die Gewichte (die auch negativ sein dürfen) durch die Einträge in  $w$  gegeben sind.

Trainieren wir die Parameter  $w[0]$  und  $b$  auf unserem eindimensionalen Datensatz `wave`, ergibt sich die folgende Gerade (siehe Abbildung 2-11):

In[24]:

```
mglearn.plots.plot_linear_regression_wave()
```

Out[24]:

```
w[0]: 0.393906 b: -0.031804
```

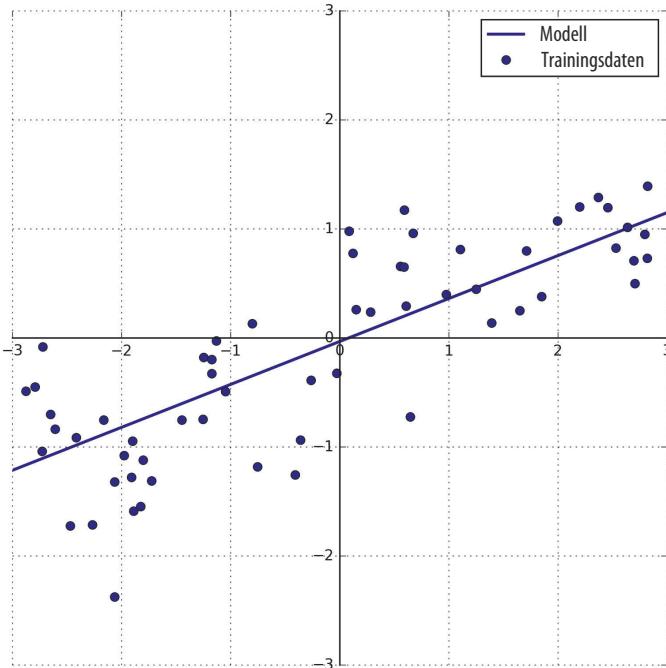


Abbildung 2-11: Vorhersagen eines linearen Modells auf dem Datensatz *wave*

Wir haben dem Diagramm ein Koordinatensystem hinzugefügt, um die Gerade leichter lesbar zu machen. Beim Betrachten von  $w[0]$  sehen wir, dass die Steigung etwa 0.4 betragen sollte, was wir im Diagramm per Augenmaß bestätigen können. Der Achsenabschnitt ist dort, wo die vorhergesagte Gerade die y-Achse schneidet: Dieser liegt etwas unter null, was wir ebenfalls im Diagramm bestätigt sehen.

Lineare Modelle zur Regression lassen sich als Regressionsmodelle beschreiben, bei denen die Vorhersage bei einem Merkmal eine Gerade ist, bei zwei Merkmalen ist sie eine Ebene und bei mehr Dimensionen eine Hyperebene.

Wenn Sie die durch unsere Gerade getroffenen Vorhersagen mit denen des KNeighborsRegressor in Abbildung 2-10 vergleichen, erscheint es zunächst sehr einen-gend, eine Gerade zur Vorhersage zu verwenden. Es wirkt, als würden sämtliche Details der Daten verloren gehen. Dies ist gewissermaßen korrekt. Die Annahme, dass unsere Zielgröße  $y$  eine lineare Kombination der Merkmale ist, ist mutig (und ein wenig unrealistisch). Eindimensionale Daten bieten allerdings eine etwas verzerrte Betrachtungsweise. Bei Datensätzen mit vielen Merkmalen können lineare Modelle außerordentlich mächtig sein. Insbesondere wenn Sie mehr Merkmale als Trainingsdatenpunkte haben, lässt sich jede Zielgröße  $y$  ausgezeichnet als lineare Funktion modellieren (auf den Trainingsdaten).<sup>6</sup>

---

<sup>6</sup> Dies ist einfacher zu verstehen, wenn Sie ein wenig lineare Algebra kennen.

Es gibt viele verschiedene lineare Modelle zur Regression. Diese Modelle unterscheiden sich darin, wie die Parameter des Modells  $w$  und  $b$  anhand der Trainingsdaten ermittelt werden und wie sich die Komplexität des Modells kontrollieren lässt. Wir werden uns nun die beliebtesten linearen Regressionsmodelle ansehen.

### Lineare Regression (gewöhnliche kleinste Quadrate)

Lineare Regression, oder die *gewöhnliche kleinste-Quadrat-Methode* (ordinary least squares, OLS), ist die einfachste und die klassische lineare Regressionsmethode. Die lineare Regression ermittelt die Parameter  $w$  und  $b$ , die die *mittlere quadratische Abweichung* zwischen Vorhersage und den tatsächlichen Werten der Zielgröße  $y$  im Trainingsdatensatz minimieren. Der mittlere quadratische Abstand ist die Summe der quadrierten Differenzen zwischen den Vorhersagen und den tatsächlichen Werten geteilt durch die Anzahl der Datenpunkte. Bei der linearen Regression gibt es keine Parameter, was ein Vorteil ist. Es gibt allerdings auch keine Möglichkeit, die Komplexität des Modells zu beeinflussen.

Hier ist der Code, der das in Abbildung 2-11 sichtbare Modell berechnet:

In[25]:

```
from sklearn.linear_model import LinearRegression
X, y = mglearn.datasets.make_wave(n_samples=60)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

lr = LinearRegression().fit(X_train, y_train)
```

Die Parameter für die »Steigung« ( $w$ ) werden auch *Gewichte* oder *Koeffizienten* genannt und sind im Attribut `coef_` gespeichert, wobei der *Achsenabschnitt* ( $b$ ) im Attribut `intercept_` gespeichert ist:

In[26]:

```
print("lr.coef_: {}".format(lr.coef_))
print("lr.intercept_: {}".format(lr.intercept_))
```

Out[26]:

```
lr.coef_: [ 0.394]
lr.intercept_: -0.031804343026759746
```



Sie haben vielleicht die seltsamen Unterstriche am Ende von `coef_` und `intercept_` bemerkt. scikit-learn speichert alle von den Trainingsdaten abgeleiteten Werte in Attributen, die mit einem Unterstrich enden. Damit lassen sie sich leichter von den durch Nutzer angegebenen Parametern unterscheiden.

Das Attribut `intercept_` ist immer eine einzelne Fließkommazahl, während das Attribut `coef_` ein NumPy-Array mit einem Eintrag pro Eingabemerkmals ist. Da wir im Datensatz `wave` nur ein einziges Eingabemerkmals haben, enthält `lr.coef_` nur einen einzelnen Wert.

Betrachten wir die Vorhersagequalität auf den Trainings- und Testdaten:

**In[27]:**

```
print("Score für den Trainingsdatensatz: {:.2f}".format(lr.score(X_train, y_train)))
print("Score für den Testdatensatz: {:.2f}".format(lr.score(X_test, y_test)))
```

**Out[27]:**

```
Score für den Trainingsdatensatz: 0.67
Score für den Testdatensatz: 0.66
```

Ein  $R^2$ -Wert von 0.66 ist nicht besonders gut, aber zumindest liegen die Werte für Trainings- und Testdaten sehr dicht beieinander. Das bedeutet, dass wir vermutlich underfitten anstatt zu overfitten. Bei diesem eindimensionalen Datensatz ist die Gefahr, zu overfitten, gering, da das Modell sehr einfach (oder eingeschränkt) ist. Bei Datensätzen mit mehr Dimensionen (also einer großen Anzahl Merkmale) werden lineare Modelle mächtiger, und das Risiko, zu overfitten, wächst. Sehen wir uns an, wie LinearRegression auf einem komplexeren Datensatz wie dem Boston Immobilien-Datensatz abschneidet. Denken Sie daran, dass dieser Datensatz 506 Datenpunkte und 105 abgeleitete Merkmale enthält. Wir laden zunächst den Datensatz und teilen ihn in Trainings- und Testdaten auf. Anschließend erstellen wir wie zuvor ein lineares Regressionsmodell:

**In[28]:**

```
X, y = mglearn.datasets.load_extended_boston()
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
lr = LinearRegression().fit(X_train, y_train)
```

Beim Vergleich der Scores für Trainings- und Testdaten bemerken wir, dass die Vorhersage auf den Trainingsdaten sehr genau, aber der  $R^2$ -Wert auf den Testdaten viel schlechter ist:

**In[29]:**

```
print("Score auf den Trainingsdaten: {:.2f}".format(lr.score(X_train, y_train)))
print("Score auf den Testdaten: {:.2f}".format(lr.score(X_test, y_test)))
```

**Out[29]:**

```
Score auf den Trainingsdaten: 0.95
Score auf den Testdaten: 0.61
```

Diese Diskrepanz zwischen der Vorhersagegüte auf Trainings- und Testdatensatz ist ein deutliches Zeichen für Overfitting. Deshalb sollten wir ein Modell finden, bei dem wir die Komplexität steuern können. Eine der am häufigsten eingesetzten Alternativen zur gewöhnlichen linearen Regression ist die *Ridge-Regression*, die wir uns im nächsten Abschnitt ansehen.

## Ridge-Regression

Die Ridge-Regression ist ebenfalls ein lineares Regressionsmodell, daher ist die Formel zur Vorhersage die gleiche wie bei der kleinsten-Quadrat-Methode. Bei der Ridge-Regression werden allerdings die Koeffizienten ( $w$ ) nicht nur der optimalen Vorhersage der Trainingsdaten angepasst, sondern auch einer zusätzlichen Nebenbedingung. Diese ist, dass wir die Beträge der Koeffizienten so klein wie möglich halten möchten; anders gesagt, sollten sämtliche Einträge von  $w$  nah bei null sein. Dies bedeutet, dass jedes einzelne Merkmal so wenig wie möglich zum Ergebnis beitragen sollte (was einer geringen Steigung entspricht), aber dennoch gute Vorhersagen getroffen werden. Diese Nebenbedingung ist ein Beispiel für *Regularisierung*. Regularisierung bedeutet, ein Modell ausdrücklich einzuschränken, um Overfitting zu vermeiden. Die bei der Ridge-Regression verwendete Art nennt man L2-Regularisierung.<sup>7</sup>

Die Ridge-Regression ist in der Klasse `linear_model.Ridge` implementiert. Schauen wir uns einmal an, wie sie auf dem erweiterten Boston Immobilien-Datensatz abschneidet:

**In[30]:**

```
from sklearn.linear_model import Ridge

ridge = Ridge().fit(X_train, y_train)
print("Score auf den Trainingsdaten: {:.2f}".format(ridge.score(X_train, y_train)))
print("Score auf den Testdaten: {:.2f}".format(ridge.score(X_test, y_test)))
```

**Out[30]:**

```
Score auf den Trainingsdaten: 0.89
Score auf den Testdaten: 0.75
```

Wie Sie sehen, ist der Score auf den Trainingsdaten bei Ridge *niedriger* als bei `LinearRegression`, während der Score auf den Testdaten *höher* ist. Dies entspricht unserer Erwartung. Bei der linearen Regression haben wir unsere Daten overfittet. Ridge ist ein stärker eingeschränktes Modell, bei dem Overfitting weniger wahrscheinlich ist. Ein weniger komplexes Modell führt zu einer schlechteren Vorhersage auf den Trainingsdaten, kann aber besser verallgemeinern. Da wir lediglich an der Verallgemeinerungsleistung interessiert sind, sollten wir das Ridge-Modell dem mit `LinearRegression` erstellten Modell vorziehen.

Das Ridge-Modell geht einen Kompromiss zwischen einem einfacheren Modell (Koeffizienten nahe null) und der Vorhersagegüte auf den Trainingsdaten ein. Der Benutzer kann über den Parameter `alpha` einstellen, wie viel Bedeutung das Modell der Einfachheit im Gegensatz zur Vorhersage auf den Trainingsdaten beimessen soll. Im vorigen Beispiel haben wir den Standardwert `alpha=1.0` verwendet. Es gibt

---

<sup>7</sup> Mathematisch wird bei Ridge ein Strafterm auf die quadrierte L2-Norm der Koeffizienten oder die euklidische Länge von  $w$  gesetzt.

aber keinen Grund anzunehmen, dass dies der bestmögliche Wert sei. Der optimale Wert für alpha hängt vom verwendeten Datensatz ab. Erhöhen von alpha drückt die Koeffizienten in Richtung null, was die Vorhersagegüte für die Trainingsdaten verschlechtert, aber bei der Verallgemeinerung helfen kann. Beispielsweise:

**In[31]:**

```
ridge10 = Ridge(alpha=10).fit(X_train, y_train)
print("Score auf den Trainingsdaten: {:.2f}".format(ridge10.score(X_train, y_train)))
print("Score auf den Testdaten: {:.2f}".format(ridge10.score(X_test, y_test)))
```

**Out[31]:**

```
Score auf den Trainingsdaten: 0.79
Score auf den Testdaten: 0.64
```

Das Verringern von alpha schränkt die Koeffizienten weniger stark ein, womit wir uns in Abbildung 2-1 nach rechts bewegen. Mit sehr kleinen Werten für alpha sind die Koeffizienten so gut wie gar nicht eingeschränkt, unser Modell entspricht dann praktisch dem mit LinearRegression erstellten:

**In[32]:**

```
ridge01 = Ridge(alpha=0.1).fit(X_train, y_train)
print("Score auf den Trainingsdaten: {:.2f}".format(ridge01.score(X_train, y_train)))
print("Score auf den Testdaten: {:.2f}".format(ridge01.score(X_test, y_test)))
```

**Out[32]:**

```
Score auf den Trainingsdaten: 0.93
Score auf den Testdaten: 0.77
```

Der Wert alpha=0.1 scheint gut zu funktionieren. Wir könnten alpha noch weiter senken, um die Verallgemeinerung zu verbessern. Es genügt fürs Erste, festzustellen, dass der Parameter alpha der Komplexität des Modells in Abbildung 2-1 entspricht. Wir werden in Kapitel 5 Methoden besprechen, um die richtigen Parameter auszuwählen.

Wir können uns auch einen qualitativen Einblick verschaffen, wie der Parameter alpha das Modell verändert. Dazu inspizieren wir das Attribut `coef_` von mit verschiedenen Werten für alpha berechneten Modellen. Ein höheres alpha steht für ein stärker eingeschränktes Modell, und daher erwarten wir mit einem größeren Wert für alpha kleinere Beträge in `coef_` als mit geringem alpha. Das Diagramm in Abbildung 2-12 bestätigt uns dies:

**In[33]:**

```
plt.plot(ridge.coef_, 's', label="Ridge alpha=1")
plt.plot(ridge10.coef_, '^', label="Ridge alpha=10")
plt.plot(ridge01.coef_, 'v', label="Ridge alpha=0.1")

plt.plot(lr.coef_, 'o', label="LinearRegression")
plt.xlabel("Index des Koeffizienten")
```

```

plt.ylabel("Betrag des Koeffizienten")
plt.hlines(0, 0, len(lr.coef_))
plt.ylim(-25, 25)
plt.legend()

```

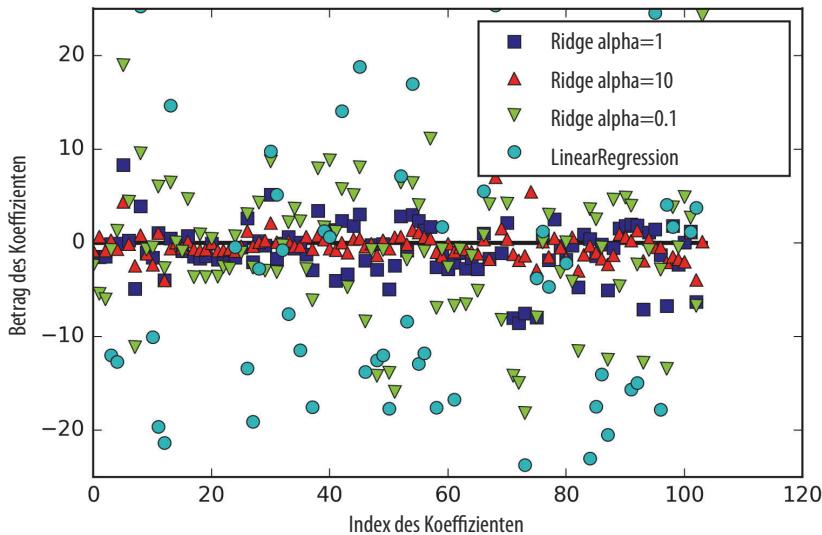


Abbildung 2-12: Vergleich der Beträge von Koeffizienten bei der Ridge-Regression mit unterschiedlichen Werten für  $\alpha$  und der linearen Regression

Hier werden die Einträge von `coef_` entlang der x-Achse aufgezählt:  $x=0$  zeigt den Koeffizienten für das erste Merkmal,  $x=1$  den Koeffizienten für das zweite Merkmal usw. bis  $x=100$ . Auf der y-Achse sind die Beträge der entsprechenden Koeffizienten dargestellt. Aus dem Diagramm können wir entnehmen, dass die Koeffizienten mit  $\alpha=10$  meist zwischen  $-3$  und  $3$  liegen. Beim Ridge-Modell mit  $\alpha=1$  sind die Koeffizienten etwas größer. Bei  $\alpha=0.1$  steigen die Beträge noch weiter an, und bei einer linearen Regression ohne Regularisierung (was  $\alpha=0$  entspricht) liegen viele der Punkte bereits außerhalb des im Diagramm gezeigten Ausschnitts.

Der Einfluss von Regularisierung lässt sich auch veranschaulichen, indem ein Wert für  $\alpha$  festgelegt, aber die Anzahl verfügbarer Trainingsdaten verändert wird. In Abbildung 2-13 haben wir Teilmengen des Boston Immobilien-Datensatzes gebildet und sowohl LinearRegression als auch Ridge( $\alpha=1$ ) auf Teilmengen wachsender Größe berechnet (Diagramme, die die Vorhersageleistung in Abhängigkeit von der Datensetanzahl darstellen, nennt man *Lernkurven*):

In[34]:

```
mglearn.plots.plot_ridge_n_samples()
```

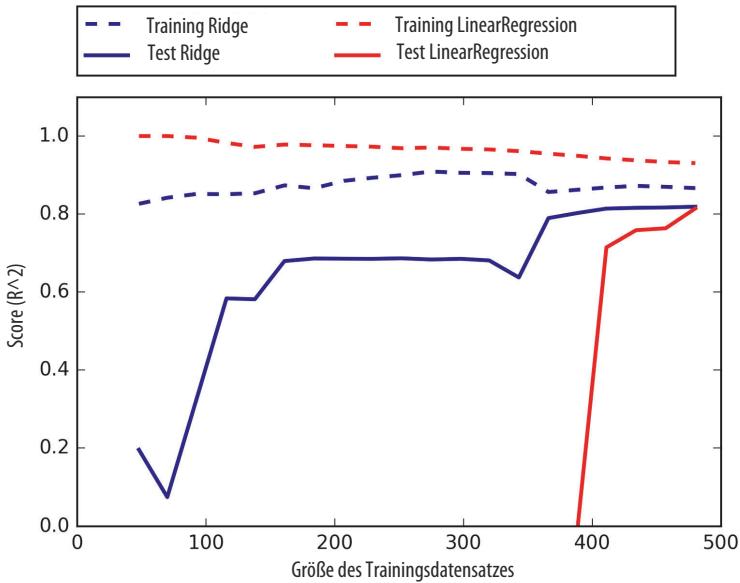


Abbildung 2-13: Learnkurven für Ridge-Regression und lineare Regression auf dem Boston Immobilien-Datensatz

Wie erwartet, ist der Score für die Trainingsdaten bei allen Teilmengen sowohl bei Ridge als auch bei der linearen Regression höher als für die Testdaten. Weil Ridge regularisiert ist, ist der Trainings-Score bei Ridge durchgängig geringer. Allerdings ist der Test-Score bei Ridge besonders bei kleinen Datensätzen besser. Bei weniger als 400 Datenpunkten ist die lineare Regression nicht imstande, irgendetwas zu lernen. Sobald dem Modell mehr und mehr Daten zur Verfügung stehen, verbessern sich beide Modelle, bis die lineare Regression Ridge am Ende einholt. Mit genug Trainingsdaten verliert die Regularisierung also an Wichtigkeit, und die lineare und Ridge-Regression schneiden gleich gut ab (es ist reiner Zufall, dass dies in unserem Beispiel beim vollständigen Datensatz der Fall ist). Ein weiterer interessanter Aspekt in Abbildung 2-13 ist, dass die Vorhersagegüte bei der linearen Regression sinkt. Werden mehr Daten hinzugefügt, wird es schwieriger, zu overfitten oder die Daten einfach auswendig zu lernen.

## Lasso

Lasso ist eine Alternative zu Ridge beim Regularisieren einer linearen Regression. Wie bei der Ridge-Regression drängt Lasso die Koeffizienten ebenfalls in Richtung null, aber auf eine andere, L1-Regression genannte Art und Weise.<sup>8</sup> Die Folge der L1-Regularisierung mit Lasso ist, dass einige Koeffizienten *genau null* betragen. Es

<sup>8</sup> Lasso ist ein Strafterm auf der L1-Norm des Koeffizientenvektors – anders gesagt, ein Strafterm auf der Summe der Beträge der Koeffizienten.

werden also einige Merkmale durch das Modell vollständig ignoriert. Wir können das als eine Art automatische Auswahl von Merkmalen ansehen. Sind einige Koeffizienten genau null, ist das Modell häufig einfacher interpretierbar und hebt die wichtigsten Merkmale Ihres Modells hervor.

Wenden wir Lasso auf den erweiterten Boston Immobilien-Datensatz an:

**In[35]:**

```
from sklearn.linear_model import Lasso

lasso = Lasso().fit(X_train, y_train)
print("Score auf den Trainingsdaten: {:.2f}".format(lasso.score(X_train, y_train)))
print("Score auf den Testdaten: {:.2f}".format(lasso.score(X_test, y_test)))
print("Anzahl verwendeter Merkmale: {}".format(np.sum(lasso.coef_ != 0)))
```

**Out[35]:**

```
Score auf den Trainingsdaten: 0.29
Score auf den Testdaten: 0.21
Anzahl verwendeter Merkmale: 4
```

Wie Sie sehen können, schneidet Lasso sowohl auf den Trainings- als auch den Testdaten sehr schlecht ab. Dies weist auf Underfitting hin. Wir sehen, dass nur 4 der 105 Merkmale verwendet wurden. Ähnlich wie Ridge gibt es auch bei Lasso einen Regularisierungsparameter alpha. Dieser beeinflusst, wie stark die Koeffizienten in Richtung null gedrängt werden. Im obigen Beispiel haben wir den Standardwert alpha=1.0 verwendet. Um das Underfitting zu verringern, probieren wir einen niedrigeren Wert für alpha aus. Dabei müssen wir auch den Standardwert für max\_iter neu einstellen (die Höchstzahl auszuführender Iterationen):

**In[36]:**

```
# wir erhöhen den Standardwert für "max_iter",
# andernfalls würde uns das Modell ermahnen, max_iter zu erhöhen.
lasso001 = Lasso(alpha=0.01, max_iter=100000).fit(X_train, y_train)
print("Score auf den Trainingsdaten: {:.2f}".format(lasso001.score(X_train, y_train)))
print("Score auf den Testdaten: {:.2f}".format(lasso001.score(X_test, y_test)))
print("Anzahl verwendeter Merkmale: {}".format(np.sum(lasso001.coef_ != 0)))
```

**Out[36]:**

```
Score auf den Trainingsdaten: 0.90
Score auf den Testdaten: 0.77
Anzahl verwendeter Merkmale: 33
```

Durch das niedrigere alpha konnten wir ein komplexeres Modell entwickeln, das auf Trainings- und Testdaten besser funktioniert. Die Vorhersagegüte ist etwas besser als bei Ridge, und wir verwenden nur 33 der 105 Merkmale. Damit ist das Modell möglicherweise einfacher zu verstehen.

Wenn wir allerdings alpha zu niedrig ansetzen, verschwindet der Effekt der Regularisierung wieder, sodass wir am Ende overfitten und das Ergebnis ähnlich zu dem von LinearRegression wird:

In[37]:

```
lasso00001 = Lasso(alpha=0.0001, max_iter=100000).fit(X_train, y_train)
print("Score auf den Trainingsdaten: {:.2f}".format(lasso00001.score(X_train, y_train)))
print("Score auf den Testdaten: {:.2f}".format(lasso00001.score(X_test, y_test)))
print("Anzahl verwendeter Merkmale: {}".format(np.sum(lasso00001.coef_ != 0)))
```

Out[37]:

```
Training set score: 0.95
Score auf den Testdaten: 0.64
Anzahl verwendeter Merkmale: 94
```

Wir können auch diesmal die Koeffizienten der verschiedenen Modelle ähnlich zu Abbildung 2-12 plotten. Das Ergebnis sehen Sie in Abbildung 2-14:

In[38]:

```
plt.plot(lasso.coef_, 's', label="Lasso alpha=1")
plt.plot(lasso001.coef_, '^', label="Lasso alpha=0.01")
plt.plot(lasso00001.coef_, 'v', label="Lasso alpha=0.0001")

plt.plot(ridge01.coef_, 'o', label="Ridge alpha=0.1")
plt.legend(ncol=2, loc=(0, 1.05))
plt.ylim(-25, 25)
plt.xlabel("Index des Koeffizienten")
plt.ylabel("Betrag des Koeffizienten")
```

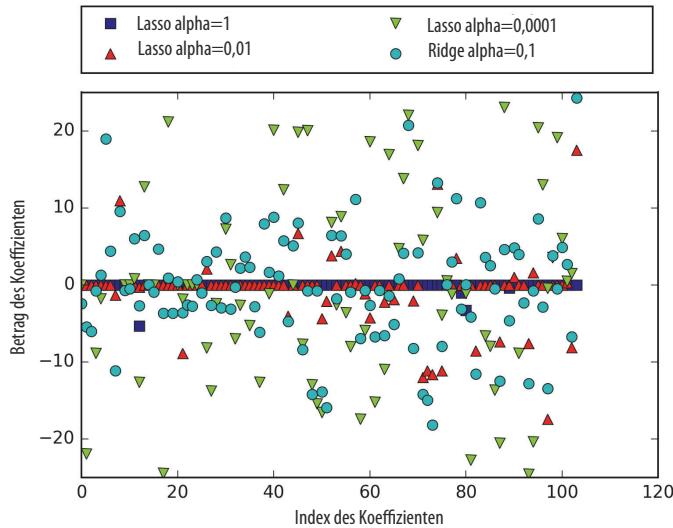


Abbildung 2-14: Vergleich der Beträge von Koeffizienten bei der Lasso-Regression mit unterschiedlichen Werten für  $\alpha$  und der Ridge-Regression

Bei  $\alpha=1$  sehen wir nicht nur, dass die meisten Koeffizienten null betragen (was wir bereits wussten), sondern auch, dass die übrigen Koeffizienten geringe Beträge haben. Verringern von  $\alpha$  auf 0.01 führt zu der durch aufrechte Dreiecke dargestellten Lösung, bei der viele Koeffizienten genau null betragen. Mit  $\alpha=0.0001$

erhalten wir ein weitgehend unregularisiertes Modell, bei dem die meisten Koeffizienten ungleich null sind und große Beträge haben. Zum Vergleich ist die beste mit Ridge ermittelte Lösung durch Kreise dargestellt. Das Ridge-Modell mit  $\alpha=0.1$  weist eine ähnliche Vorhersageleistung auf wie das Lasso-Modell mit  $\alpha=0.01$ , aber bei Ridge sind sämtliche Koeffizienten ungleich null.

In der Praxis zieht man unter diesen beiden Modellen meist die Ridge-Regression vor. Wenn Sie jedoch eine große Anzahl Merkmale haben und erwarten, dass nur wenige davon bedeutsam sind, so könnte Lasso die geeignete Wahl sein. Auch wenn Sie ein leicht interpretierbares Modell erhalten möchten, liefert Lasso ein leichter verständliches Modell, bei dem nur ein Teil der Eingabemerkmale ausgewählt wird. Außerdem enthält scikit-learn noch die Klasse ElasticNet, die die Strafsterme aus Lasso und Ridge in sich vereint. In der Praxis funktioniert diese Kombination am besten, auch wenn Sie dafür zwei Parameter anpassen müssen: einen für die L1-Regularisierung und einen für die L2-Regularisierung.

## Lineare Modelle zur Klassifizierung

Lineare Modelle sind auch zur Klassifizierung weitverbreitet. Betrachten wir zunächst die binäre Klassifizierung. In diesem Fall soll anhand folgender Formel eine Vorhersage getroffen werden:

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \dots + w[p] * x[p] + b > 0$$

Diese Formel ist der zur linearen Regression sehr ähnlich. Anstatt jedoch eine gewichtete Summe der Merkmale zu berechnen, setzen wir einen Schwellenwert von null für die vorhergesagte Größe an. Ist die Funktion kleiner als null, sagen wir die Kategorie  $-1$  vorher; ist sie größer als null, sagen wir die Kategorie  $+1$  vorher. Diese Regel zur Vorhersage trifft auf alle linearen Klassifikationsmodelle zu. Auch hier gibt es viele Möglichkeiten, die Koeffizienten ( $w$ ) und den Achsenabschnitt ( $b$ ) zu ermitteln.

Bei linearen Regressionsmodellen ist die Ausgabe  $\hat{y}$  eine lineare Funktion der Merkmale: Gerade, Ebene oder Hyperebene (bei mehr Dimensionen). Bei linearen Klassifikationsmodellen ist die *Entscheidungsgrenze* eine lineare Funktion der Eingabe. Anders gesagt, ist ein (binärer) linearer Klassifikator ein Klassifikator, der zwei Klassen durch eine Gerade, eine Ebene oder eine Hyperebene voneinander trennt. In diesem Abschnitt werden wir Beispiele hierfür kennenlernen.

Es gibt viele Algorithmen zum Trainieren linearer Modelle. Diese Algorithmen unterscheiden sich in den folgenden zwei Punkten:

- die Methode, nach der berechnet wird, wie gut eine bestimmte Kombination von Koeffizienten und Achsenabschnitt zu den Trainingsdaten passt
- ob und was für eine Regularisierung verwendet wird

Unterschiedliche Algorithmen wenden unterschiedliche Kriterien an, um die Anpassung an die Trainingsdaten zu messen. Aus technisch-mathematischen Gründen ist es leider nicht möglich,  $w$  und  $b$  so einzustellen, dass die Anzahl falscher Zuordnungen minimiert wird. Für unsere Zwecke sind die unterschiedlichen

Möglichkeiten für den ersten Punkt in der obigen Liste (genannt *Verlustfunktionen*) von geringer Bedeutung.

Die zwei beliebtesten linearen Algorithmen zur Klassifikation sind *logistische Regression*, die in der Klasse `linear_model.LogisticRegression` implementiert ist, sowie *lineare Support Vector Machines* (lineare SVMs), implementiert als `svm.LinearSVC` (`SVC` steht für Support Vector Classifier). Trotz seines Namens handelt es sich bei `LogisticRegression` um ein Klassifikationsverfahren und keinen Regressionsalgorithmus, und wir sollten ihn keinesfalls mit `LinearRegression` verwechseln.

Wir können die Modelle `LogisticRegression` und `LinearSVC` auf den Datensatz `forge` anwenden und die von diesen linearen Modellen ermittelte Entscheidungsgrenze visualisieren (Abbildung 2-15):

In[39]:

```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC

X, y = mglearn.datasets.make_forge()

fig, axes = plt.subplots(1, 2, figsize=(10, 3))

for model, ax in zip([LinearSVC(), LogisticRegression()], axes):
    clf = model.fit(X, y)
    mglearn.plots.plot_2d_separator(clf, X, fill=False, eps=0.5,
                                    ax=ax, alpha=.7)
    mglearn.discrete_scatter(X[:, 0], X[:, 1], y, ax=ax)
    ax.set_title("{}".format(clf.__class__.__name__))
    ax.set_xlabel("Merkmal 0")
    ax.set_ylabel("Merkmal 1")
    axes[0].legend()
```

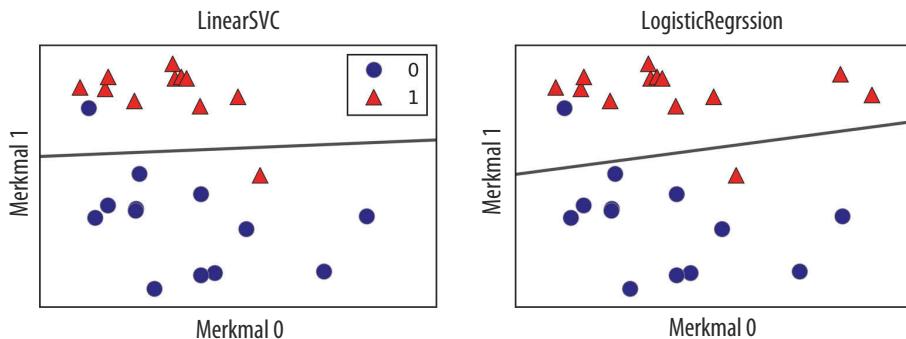


Abbildung 2-15: Entscheidungsgrenzen eines linearen SVM und der logistischen Regression auf dem Datensatz `forge` mit Standardeinstellungen

In dieser Abbildung ist wie zuvor das erste Merkmal des `forge`-Datensatzes auf der x-Achse aufgetragen, und das zweite Merkmal steht auf der y-Achse. Wir stellen die jeweils von `LinearSVC` und `LogisticRegression` gefundenen Entscheidungsgrenzen als Linien dar, die den oberen als Kategorie 1 zugeordneten Bereich von dem unte-

ren als Kategorie 0 zugeordneten trennen. Jeder zusätzliche Datenpunkt über der schwarzen Linie wird also vom Klassifikator der Kategorie 1 und jeder Punkt unter der schwarzen Linie der Kategorie 0 zugewiesen.

Beide Modelle ermitteln ähnliche Entscheidungsgrenzen. Beachten Sie, dass beide Modelle zwei der Punkte falsch zuordnen. Standardmäßig verwenden beide Modelle eine L2-Regularisierung, wie Ridge es bei der Regression tut.

Bei den Modellen LogisticRegression und LinearSVC heißt der Parameter zum Einstellen der Stärke der Regularisierung  $C$ . Höhere Werte für  $C$  entsprechen *weniger* Regularisierung. Wenn Sie also einen hohen Wert für  $C$  verwenden, versuchen LogisticRegression und LinearSVC die Trainingsdaten so gut wie möglich anzupassen, wohingegen geringe Werte für den Parameter  $C$  das Modell einen Koeffizientenvektor ( $w$ ) finden lassen, der nahe null liegt.

Es gibt einen weiteren interessanten Aspekt der Funktionsweise von  $C$ . Bei niedrigen Werten für  $C$  versucht der Algorithmus, sich an die »Mehrheit« der Datenpunkte anzupassen. Ein höherer Wert für  $C$  betont dagegen die Wichtigkeit einer korrekten Klassifikation jedes einzelnen Datenpunktes. Hier sehen Sie eine Visualisierung mit LinearSVC (Abbildung 2-16):

In[40]:

```
mglearn.plots.plot_linear_svc_regularization()
```

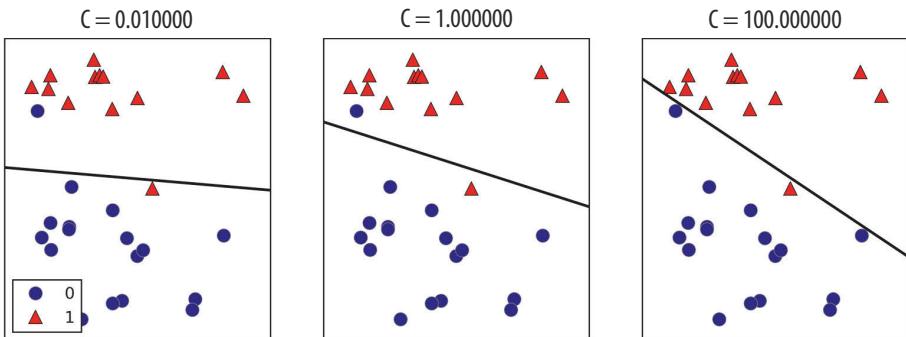


Abbildung 2-16: Entscheidungsgrenzen einer linearen SVM auf dem Datensatz forge mit unterschiedlichen Werten für  $C$

Auf der linken Seite verwenden wir ein sehr kleines  $C$ , das einer sehr starken Regularisierung entspricht. Die meisten der Punkte in Kategorie 0 liegen unten und die meisten Punkte in Kategorie 1 liegen oben. Das stark regularisierte Modell wählt eine mehr oder weniger horizontale Linie, bei der zwei Punkte falsch zugeordnet werden. Im mittleren Diagramm ist  $C$  etwas höher, und das Modell konzentriert sich stärker auf die zwei falsch zugeordneten Punkte, sodass sich die Entscheidungsgrenze verschiebt. Auf der rechten Seite schließlich kippt der hohe Wert für  $C$  die Entscheidungsgrenze deutlich, sodass nun sämtliche Punkte korrekt der Kategorie 0 zugeordnet werden. Einer der Punkte in Kategorie 1 ist noch immer falsch,

da es in diesem Datensatz nicht möglich ist, alle Punkte durch eine Gerade korrekt zu klassifizieren. Das Modell auf der rechten Seite bemüht sich sehr um eine korrekte Klassifizierung, gibt aber das Gesamtbild der beiden Kategorien nicht ungedingt gut wieder. Anders gesagt, führt es leichter zu Overfitting.

Ähnlich wie bei der Regression erscheinen lineare Klassifikationsmodelle bei wenigen Dimensionen sehr restriktiv, da sie nur Linien und Ebenen als Entscheidungsgrenzen zulassen. Auch bei der Klassifikation werden lineare Modelle bei vielen Dimensionen sehr mächtig, und es wird zunehmend wichtiger, sich bei mehr Merkmalen vor Overfitting zu schützen.

Analysieren wir LogisticRegression etwas genauer anhand des Brustkrebs-Datensatzes:

**In[41]:**

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=42)
logreg = LogisticRegression().fit(X_train, y_train)
print("Score auf den Trainingsdaten: {:.3f}".format(logreg.score(X_train, y_train)))
print("Score auf den Testdaten: {:.3f}".format(logreg.score(X_test, y_test)))
```

**Out[41]:**

```
Score auf den Trainingsdaten: 0.953
Score auf den Testdaten: 0.958
```

Der Standardwert von  $C=1$  liefert uns mit 95 % sowohl auf den Trainings- als auch den Testdaten bereits eine gute Vorhersagegenauigkeit. Da jedoch die Scores für den Trainings- und den Testdatensatz sehr nah beieinanderliegen, underfitten wir vermutlich. Erhöhen wir also  $C$ , um ein flexibleres Modell zu erhalten:

**In[42]:**

```
logreg100 = LogisticRegression(C=100).fit(X_train, y_train)
print("Score auf den Trainingsdaten: {:.3f}".format(logreg100.score(X_train, y_train)))
print("Score auf den Testdaten: {:.3f}".format(logreg100.score(X_test, y_test)))
```

**Out[42]:**

```
Score auf den Trainingsdaten: 0.972
Score auf den Testdaten: 0.965
```

Der Wert  $C=100$  erhöht die Genauigkeit auf den Trainingsdaten, und auch die Genauigkeit auf den Testdaten erhöht sich etwas. Dies bestätigt unsere Vermutung, dass ein komplexeres Modell hier angebracht ist.

Wir können auch untersuchen, wie sich ein stärker regularisiertes Modell auswirkt, indem wir statt dem Standardwert  $C=1$  den Parameter  $C=0.01$  verwenden:

**In[43]:**

```
logreg001 = LogisticRegression(C=0.01).fit(X_train, y_train)
print("Score auf den Trainingsdaten: {:.3f}".format(logreg001.score(X_train, y_train)))
print("Score auf den Testdaten: {:.3f}".format(logreg001.score(X_test, y_test)))
```

**Out[43]:**

```
Score auf den Trainingsdaten: 0.934
Score auf den Testdaten: 0.930
```

Wie erwartet, sinkt die Genauigkeit sowohl für die Trainings- als auch die Testdaten, wenn wir uns auf der in Abbildung 2-1 gezeigten Skala nach links bewegen, obwohl bereits Underfitting vorliegt.

Schließlich betrachten wir noch die vom Modell mit den drei unterschiedlichen Werten für den Regularisierungsparameter  $C$  berechneten Koeffizienten (Abbildung 2-17):

**In[44]:**

```
plt.plot(logreg.coef_.T, 'o', label="C=1")
plt.plot(logreg100.coef_.T, '^', label="C=100")
plt.plot(logreg001.coef_.T, 'v', label="C=0.001")
plt.xticks(range(cancer.data.shape[1]), cancer.feature_names, rotation=90)
plt.hlines(0, 0, cancer.data.shape[1])
plt.ylim(-5, 5)
plt.xlabel("Index des Koeffizienten")
plt.ylabel("Betrag des Koeffizienten")
plt.legend()
```

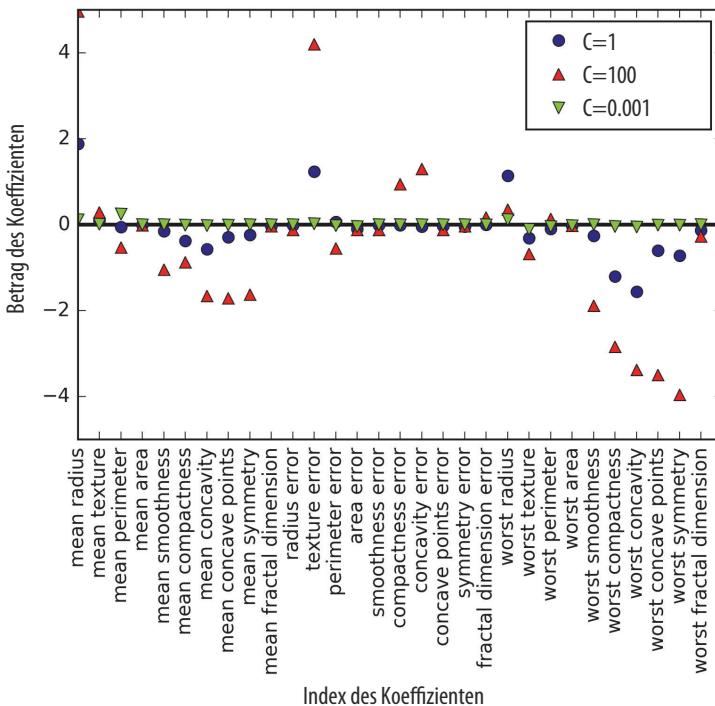


Abbildung 2-17: Durch logistische Regression auf dem Brustkrebs-Datensatz ermittelte Koeffizienten für unterschiedliche Werte von  $C$



Weil LogisticRegression standardmäßig eine L2-Regularisierung vornimmt, sieht das Ergebnis ähnlich wie das von Ridge in Abbildung 2-12 produzierte aus. Stärkere Regularisierung drängt die Koeffizienten mehr und mehr in Richtung null, auch wenn sie nie genau null werden. Betrachten wir das Diagramm genauer, beobachten wir beim dritten Koeffizienten, »mean perimeter«, einen interessanten Effekt: Bei  $C=100$  und  $C=1$  ist der Koeffizient negativ, bei  $C=0.001$  dagegen positiv und hat einen höheren Betrag als für  $C=1$ . Wenn wir ein Modell auf diese Weise interpretieren, könnte man annehmen, dass uns die Koeffizienten etwas darüber verraten, zu welcher Kategorie ein Merkmal gehört. Beispielsweise könnte man meinen, dass ein hoher »texture error« mit der Kategorie »malignant« assoziiert ist. Allerdings bedeutet der Vorzeichenwechsel bei »mean perimeter«, dass ein hoher »mean perimeter« je nach Modell ein Anzeichen für die »benigne« oder »maligne« Kategorie ist. Dies zeigt uns, dass man die Koeffizienten eines linearen Modells immer mit etwas Vorsicht betrachten sollte.

Wenn wir uns ein leichter interpretierbares Modell wünschen, kann uns L1-Regularisierung helfen, da diese das Modell auf wenige Merkmale begrenzt. Hier sehen Sie das Diagramm der Koeffizienten und die Genauigkeiten der Klassifikation mit L1-Regularisierung (Abbildung 2-18):

**In[45]:**

```
for C, marker in zip([0.001, 1, 100], ['o', '^', 'v']):
    lr_l1 = LogisticRegression(C=C, penalty="l1").fit(X_train, y_train)
    print("Genauigkeit auf den Trainingsdaten mit l1 logreg und C={:.3f}: {:.2f}".
          format(C, lr_l1.score(X_train, y_train)))
    print("Genauigkeit auf den Testdaten mit l1 logreg und C={:.3f}: {:.2f}".
          format(C, lr_l1.score(X_test, y_test)))
    plt.plot(lr_l1.coef_.T, marker, label="C={:.3f}".format(C))

plt.xticks(range(cancer.data.shape[1]), cancer.feature_names, rotation=90)
plt.hlines(0, 0, cancer.data.shape[1])
plt.xlabel("Name des Merkmals")
plt.ylabel("Betrag des Koeffizienten")

plt.ylim(-5, 5)
plt.legend(loc=3)
```

**Out[45]:**

```
Genauigkeit auf den Trainingsdaten mit l1 logreg und C=0.001: 0.91
Genauigkeit auf den Testdaten mit l1 logreg und C=0.001: 0.92
Genauigkeit auf den Trainingsdaten mit l1 logreg und C=1.000: 0.96
Genauigkeit auf den Testdaten mit l1 logreg und C=1.000: 0.96
Genauigkeit auf den Trainingsdaten mit l1 logreg und C=100.000: 0.99
Genauigkeit auf den Testdaten mit l1 logreg und C=100.000: 0.98
```

Wie Sie sehen, gibt es viele Parallelen zwischen linearen Modellen zur binären Klassifikation und linearen Regressionsmodellen. Wie bei der Regression ist der Hauptunterschied zwischen den Modellen der Strafterm `penalty`, der die Regularisierung

beeinflusst, und ob das Modell sämtliche verfügbaren Merkmale oder nur eine Teilmenge berücksichtigt.

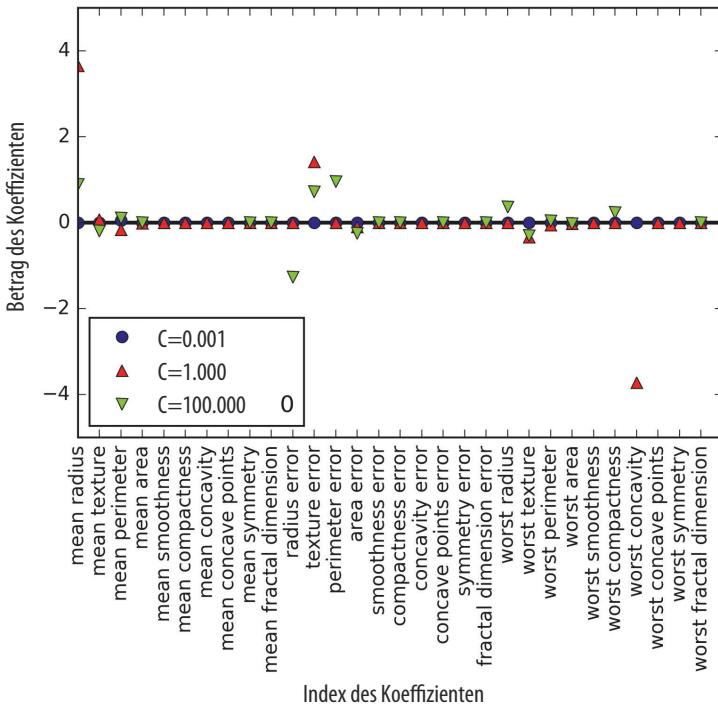


Abbildung 2-18: Mit logistischer Regression mit L1-Strafterm auf dem Brustkrebs-Datensatz ermittelte Koeffizienten bei unterschiedlichen Werten für  $C$

### Lineare Klassifikationsmodelle mit mehreren Kategorien

Viele lineare Klassifikationsmodelle eignen sich nur für binäre Klassifikationsaufgaben. Sie lassen sich nicht ohne Weiteres für Fälle mit mehreren Kategorien erweitern (mit der Ausnahme der logistischen Regression). Eine verbreitete Technik zum Erweitern eines binären Klassifikationsalgorithmus zu einem für viele Kategorien ist der *One-vs.-Rest*-Ansatz. Beim One-vs.-Rest-Ansatz wird für jede Kategorie ein binäres Modell trainiert, das diese Kategorie von allen übrigen separiert. So ergeben sich genau so viele binäre Modelle, wie es Kategorien gibt. Um eine Vorhersage zu treffen, werden sämtliche binären Klassifikatoren auf einem Testdatenpunkt ausgeführt. Der Klassifikator mit dem höchsten Score auf seiner einzelnen Kategorie »gewinnt«, und die Bezeichnung seiner Kategorie wird als Vorhersage ausgegeben.

Bei einem binären Klassifikator pro Kategorie erhalten wir für jede Kategorie einen Koeffizientenvektor ( $w$ ) und einen Achsenabschnitt ( $b$ ). Die Kategorie, bei der das Ergebnis der hier angegebenen Konfidenzformel zur Klassifikation am höchsten ist, wird vorhergesagt:

$$w[0] * x[0] + w[1] * x[1] + \dots + w[p] * x[p] + b$$

Die der logistischen Regression bei mehreren Klassen zugrunde liegende Mathematik unterscheidet sich etwas vom One-vs.-Rest-Ansatz, aber auch sie resultiert in einem Koeffizientenvektor und einem Achsenabschnitt pro Kategorie, und es kommt die gleiche Methode zur Vorhersage zur Anwendung.

Wenden wir die One-vs.-Rest-Methode auf einen einfachen Klassifikationsdatensatz mit drei Kategorien an. Wir verwenden einen zweidimensionalen Datensatz, bei dem jede Kategorie aus einer durch eine Gaußverteilung generierten Stichprobe besteht (siehe Abbildung 2-19):

**In[46]:**

```
from sklearn.datasets import make_blobs

X, y = make_blobs(random_state=42)
mplt.scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Merkmal 0")
plt.ylabel("Merkmal 1")
plt.legend(["Kategorie 0", "Kategorie 1", "Kategorie 2"])
```

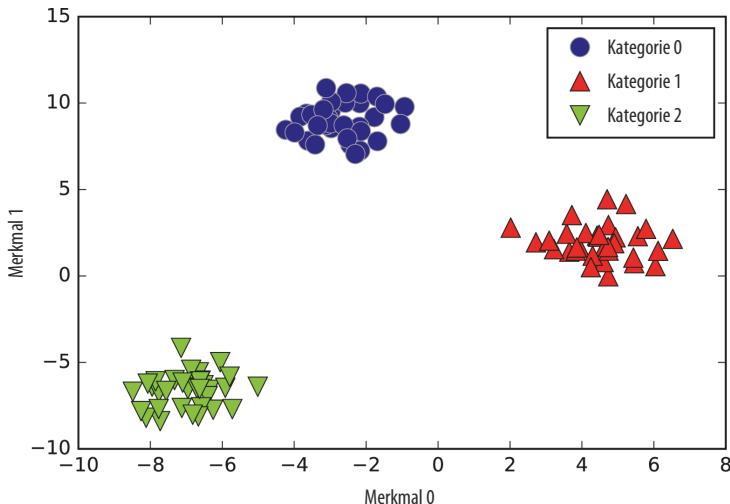


Abbildung 2-19: Zweidimensionaler Beispieldatensatz mit drei Kategorien

Nun trainieren wir den Klassifikator LinearSVC auf diesem Datensatz:

**In[47]:**

```
linear_svm = LinearSVC().fit(X, y)
print("Abmessungen der Koeffizienten: ", linear_svm.coef_.shape)
print("Abmessungen des Achsenabschnitts: ", linear_svm.intercept_.shape)
```

**Out[47]:**

```
Abmessungen der Koeffizienten: (3, 2)
Abmessungen des Achsenabschnitts: (3,)
```

Wir sehen, dass die Abmessungen von `coef_` bei (3, 2) liegen, also jede Zeile in `coef_` dem Koeffizientenvektor einer der drei Kategorien entspricht und jede Spalte den Wert des Koeffizienten für ein Merkmal enthält (in diesem Datensatz gibt es zwei Merkmale). Die Achsenabschnitte für jede Kategorie sind nun in `intercept_` als eindimensionales Array gespeichert. Die durch diese drei binären Klassifikatoren definierten Linien lassen sich visualisieren (Abbildung 2-20):

**In[48]:**

```
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
line = np.linspace(-15, 15)
for coef, intercept, color in zip(linear_svm.coef_, linear_svm.intercept_,
                                  ['b', 'r', 'g']):
    plt.plot(line, -(line * coef[0] + intercept) / coef[1], c=color)
plt.ylim(-10, 15)
plt.xlim(-10, 8)
plt.xlabel("Merkmal 0")
plt.ylabel("Merkmal 1")
plt.legend(['Kategorie 0', 'Kategorie 1', 'Kategorie 2', 'Gerade Kategorie 0',
           'Gerade Kategorie 1', 'Gerade Kategorie 2'], loc=(1.01, 0.3))
```

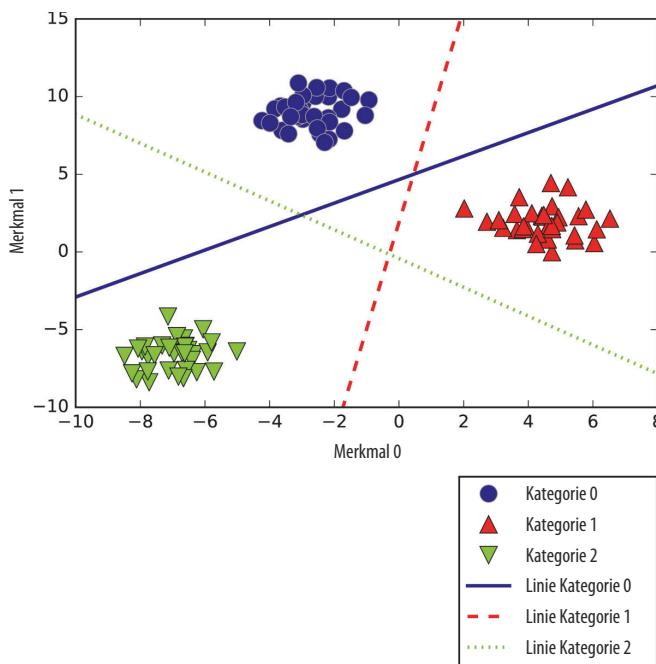


Abbildung 2-20: Entscheidungsgrenzen der drei One-vs.-Rest-Klassifikatoren

Wie Sie sehen, liegen alle Trainingsdatenpunkte aus Kategorie 0 über der Geraden für Kategorie 0. Sie liegen also für diesen binären Klassifikator auf der Seite »Kategorie 0«. Die Punkte in Kategorie 0 liegen über der Geraden für Kategorie 2, sie werden vom binären Klassifikator für Kategorie 2 als »Rest« klassifiziert. Die Punkte in Kategorie 0 liegen links der Geraden für Kategorie 1, daher stuft sie auch der binäre

Klassifikator für Kategorie 1 als »Rest« ein. Deshalb werden alle Punkte in diesem Gebiet vom gesamten Klassifikationsverfahren als Kategorie 0 klassifiziert (das Ergebnis der Konfidenzformel ist beim Klassifikator 0 größer null und kleiner als null bei den beiden übrigen Klassifikatoren).

Was aber ist mit dem Dreieck in der Mitte des Diagramms? Alle drei binären Klassifikatoren stufen die dortigen Punkte als »Rest« ein. Welcher Kategorie würde ein Punkt in diesem Bereich zugeordnet? Die Antwort ist diejenige Kategorie mit dem höchsten Wert in der Klassifikationsformel: die Kategorie mit der nächstgelegenen Geraden. Das folgende Beispiel (Abbildung 2-21) zeigt die Vorhersagen für sämtliche Bereiche des 2-D-Raumes:

In[49]:

```
mglearn.plots.plot_2d_classification(linear_svm, X, fill=True, alpha=.7)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
line = np.linspace(-15, 15)
for coef, intercept, color in zip(linear_svm.coef_, linear_svm.intercept_,
['b', 'r', 'g']):
    plt.plot(line, -(line * coef[0] + intercept) / coef[1], c=color)
plt.legend(['Kategorie 0', 'Kategorie 1', 'Kategorie 2', 'Gerade Kategorie 0',
'Gerade Kategorie 1', 'Gerade Kategorie 2'], loc=(1.01, 0.3))
plt.xlabel("Merkmals 0")
plt.ylabel("Merkmals 1")
```

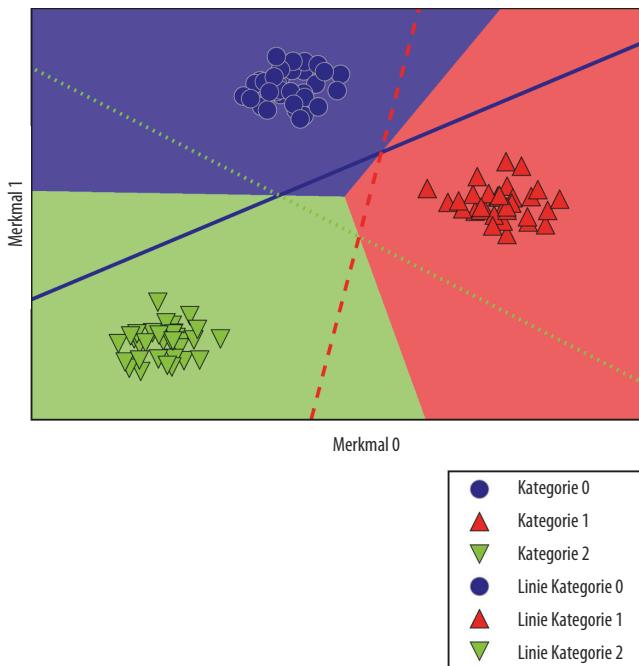


Abbildung 2-21: Von den drei One-vs.-Rest-Klassifikatoren abgeleitete Entscheidungsgrenzen für mehrere Klassen

## Stärken, Schwächen und Parameter

Der wichtigste Parameter bei linearen Modellen ist der Regularisierungsparameter. Bei Regressionsmodellen wird dieser alpha genannt und C bei LinearSVC und LogisticRegression. Große Werte für alpha und kleine Werte für C stehen für einfache Modelle. Insbesondere bei Regressionsmodellen ist das Einstellen dieser Parameter recht wichtig. Normalerweise werden C und alpha entlang einer logarithmischen Skala durchsucht. Eine zweite Entscheidung ist, ob Sie L1-Regularisierung oder L2-Regularisierung verwenden möchten. Wenn Sie annehmen, dass nur einige Ihrer Merkmale wichtig sind, sollten Sie sich für L1 entscheiden. Andernfalls empfehlen wir L2 als Standardeinstellung. L1 ist auch nützlich, wenn die Interpretierbarkeit des Modells wichtig ist. Da L1 nur wenige Merkmale verwendet, ist es einfacher zu deuten, welche Merkmale für das Modell wichtig sind und welche Auswirkungen diese Merkmale haben.

Lineare Modelle lassen sich sehr schnell trainieren und sind auch in der Vorhersage schnell. Sie skalieren für sehr große Datensätze und funktionieren auch mit dünn besetzten Daten gut. Wenn Ihre Daten aus Hunderttausenden oder Millionen Datenpunkten bestehen, sollten Sie die Option solver='sag' bei LogisticRegression und Ridge erwägen, die bei großen Datensätzen schneller arbeitet als die Standardeinstellung. Andere Möglichkeiten sind die Klassen SGDClassifier und SGDRegressor, die besser skalierbare Versionen der hier beschriebenen linearen Modelle implementieren.

Eine weitere Stärke linearer Modelle ist, dass durch die oben angegebenen Formeln zur Regression und Klassifikation relativ leicht nachvollziehbar wird, wie eine Vorhersage zustande kommt. Leider ist nicht immer klar, warum die Koeffizienten so sind, wie sie sind. Dies trifft besonders auf Datensätze mit stark korrelierenden Merkmalen zu; in diesen Fällen können die Koeffizienten schwer zu interpretieren sein.

Lineare Modelle sind oft besonders leistungsfähig, wenn die Anzahl der Merkmale im Vergleich zur Anzahl der Datenpunkte hoch ist. Sie werden auch häufig bei sehr großen Datensätzen eingesetzt, weil das Trainieren anderer Modelle schlichtweg nicht praktikabel ist. In Räumen mit weniger Dimensionen jedoch können andere Modelle leistungsfähiger verallgemeinern. Wir werden uns in Abschnitt »Support Vector Machines mit Kernel« auf Seite 88 einige Beispiele ansehen, bei denen lineare Modelle scheitern.

## Method Chaining

Die Methode fit sämtlicher Modelle in scikit-learn gibt self zurück. Damit können wir Code so schreiben, wie wir es in diesem Kapitel bereits oft getan haben:

In[50]:

```
# instanziere das Modell und passe es in der gleichen Zeile an
logreg = LogisticRegression().fit(X_train, y_train)
```

Wir haben hier den Rückgabewert von `fit` (also `self`) verwendet, um das trainierte Modell der Variablen `logreg` zuzuweisen. Diese Verbindung von Methodenaufrufen (hier `init` und anschließend `fit`) nennt man auch *Method Chaining*. Eine weitere Anwendung von Method Chaining in `scikit-learn` ist, `fit` und `predict` in einer Zeile aufzurufen:

**In[51]:**

```
logreg = LogisticRegression()
y_pred = logreg.fit(X_train, y_train).predict(X_test)
```

Sie können sogar die Instanziierung des Modells, Anpassen und Vorhersage in einer Zeile durchführen:

**In[52]:**

```
y_pred = LogisticRegression().fit(X_train, y_train).predict(X_test)
```

Diese sehr kurze Schreibweise ist allerdings nicht ideal. In dieser einzelnen Zeile passiert eine Menge, wodurch der Code schwerer zu lesen ist. Außerdem speichern wir das angepasste logistische Regressionsmodell in keiner Variablen, sodass wir es nicht inspizieren oder zur Vorhersage weiterer Daten verwenden können.

## Naive Bayes-Klassifikatoren

Naive Bayes-Klassifikatoren sind eine Familie von Klassifikatoren, die zu den linearen Modellen aus dem vorigen Abschnitt recht ähnlich sind. Allerdings sind sie tendenziell noch schneller zu trainieren. Der Preis für diese Effizienz ist, dass naive Bayes-Modelle oft eine etwas schlechtere Fähigkeit zur Verallgemeinerung im Vergleich mit linearen Klassifikatoren wie `LogisticRegression` und `LinearSVC` aufweisen.

Der Grund für die Effizienz naiver Bayes-Modelle ist, dass sie Parameter trainieren, indem sie jedes Merkmal einzeln betrachten und aus jedem Merkmal einfache Statistiken für jede Kategorie ableiten. In `scikit-learn` sind drei Arten naiver Bayes-Klassifikatoren implementiert: `GaussianNB`, `BernoulliNB`, und `MultinomialNB`. `GaussianNB` lässt sich auf jegliche kontinuierliche Daten anwenden, während `BernoulliNB` von binären Daten ausgeht und `MultinomialNB` Zähldaten erwartet (also jedes Merkmal eine ganzzahlige Anzahl von etwas ist, z. B. wie oft ein Wort in einem Satz vorkommt). `BernoulliNB` und `MultinomialNB` werden vor allem bei der Klassifikation von Textdaten eingesetzt.

Der Klassifikator `BernoulliNB` zählt, wie oft jedes Merkmal in jeder Kategorie ungleich null ist. Dies lässt sich am leichtesten mit einem Beispiel nachvollziehen:

**In[53]:**

```
X = np.array([[0, 1, 0, 1],
              [1, 0, 1, 1],
              [0, 0, 0, 1],
              [1, 0, 1, 0]])
y = np.array([0, 1, 0, 1])
```

Hier haben wir vier Datenpunkte mit jeweils vier binären Merkmalen. Es gibt zwei Kategorien, 0 und 1. Bei Kategorie 0 (der erste und dritte Datenpunkt) ist das erste Merkmal zweimal null und nie ungleich null, das zweite Merkmal ist einmal null und einmal ungleich null usw. Die gleichen Anzahlen werden auch für die Datenpunkte der zweiten Kategorie ermittelt. Im Wesentlichen besteht das Zählen der Einträge ungleich null für jede Kategorie:

**In[54]:**

```
counts = {}
for label in np.unique(y):
    # iteriere über jede Kategorie
    # Zähle (summiere) Einträge von 1 für jedes Merkmal zusammen
    counts[label] = X[y == label].sum(axis=0)
print("Anzahl Merkmale:\n{}".format(counts))
```

**Out[54]:**

```
Anzahl Merkmale:
0: array([0, 1, 0, 2]), 1: array([2, 0, 2, 1])}
```

Die anderen beiden naiven Bayes-Modelle, `MultinomialNB` und `GaussianNB`, unterscheiden sich ein wenig in der Art der berechneten Statistiken. `MultinomialNB` berücksichtigt den Durchschnittswert jedes Merkmals bei jeder Kategorie. Dagegen speichert `GaussianNB` sowohl den Durchschnittswert als auch die Standardabweichung für jedes Merkmal und für jede Kategorie ab.

Um eine Vorhersage zu treffen, wird ein Datenpunkt mit den für jede Kategorie erhobenen Statistiken verglichen, und die am besten passende Kategorie wird vorhergesagt. Interessanterweise führt diese sowohl bei `MultinomialNB` als auch bei `BernoulliNB` zu einer Vorhersageformel der gleichen Form wie bei linearen Modellen (siehe Abschnitt »Lineare Modelle zur Klassifizierung« auf Seite 55). Leider ist die Bedeutung von `coef_` bei naiven Bayes-Modellen eine andere als bei linearen Modellen, sodass `coef_` nicht identisch mit `w` ist.

## Stärken, Schwächen und Parameter

`MultinomialNB` und `BernoulliNB` besitzen einen einzigen Parameter, `alpha`, der die Komplexität des Modells einstellt. Die Funktionsweise von `alpha` ist, dass der Algorithmus virtuelle Datenpunkte mit positiven Werten für alle Merkmale zu den Daten hinzufügt. Dies führt zu einer »Glättung« der Statistiken. Ein großer Wert für `alpha` führt zu mehr Glättung und damit zu weniger komplexen Modellen. Die Leistung des Algorithmus ist relativ unempfindlich gegenüber dem Wert für `alpha`. Für eine gute Leistung ist ein Setzen von `alpha` nicht entscheidend. Trotzdem lässt sich die Genauigkeit für gewöhnlich durch Feineinstellung dieses Wertes etwas verbessern.

`GaussianNB` wird vor allem bei Daten mit sehr vielen Dimensionen eingesetzt, während die anderen beiden Varianten des naiven Bayes-Verfahrens bei dünn besetzten Zähldaten wie Text weitverbreitet sind. `MultinomialNB` schneidet dabei normalerweise besser als `BernoulliNB` ab, vor allem bei Datensätzen mit einer relativ großen Zahl von Nullwerten (z. B. großen Dokumenten).

Die naiven Bayes-Modelle teilen viele der Stärken und Schwächen linearer Modelle. Sie lassen sich schnell trainieren und für Vorhersagen einsetzen, und die Prozedur beim Trainieren ist leicht nachvollziehbar. Die Modelle funktionieren für hochdimensionale dünn besetzte Daten gut und sind Änderungen der Parameter gegenüber robust. Naive Bayes-Modelle eignen sich hervorragend, um eine Grundlinie zu etablieren, und werden häufig bei sehr großen Datensätzen eingesetzt, bei denen sogar das Trainieren eines linearen Modells zu lange dauern würde.

## Entscheidungsbäume

Entscheidungsbäume sind weitverbreitete Modelle für Klassifikations- und Regressionsaufgaben. Im Wesentlichen erlernen sie eine hierarchische Folge von Ja/Nein-Fragen, die zu einer Entscheidung führen.

Diese Fragen sind ähnlich zu denen im Spiel »20 Fragen«. Nehmen wir an, Sie möchten zwischen den folgenden vier Tieren unterscheiden: Bären, Falken, Pinguinen und Delfinen. Ihr Ziel ist es, die richtige Antwort mit so wenigen Ja/Nein-Fragen wie möglich zu finden. Sie könnten zuerst die Frage stellen, ob das Tier Federn hat, eine Frage, die die möglichen Tiere auf zwei einschränkt. Wenn die Antwort »ja« ist, können Sie eine zweite Frage stellen, die zwischen Falken und Pinguinen unterscheidet. Zum Beispiel könnten Sie fragen, ob das Tier fliegen kann. Wenn das Tier keine Federn hat, kommen noch Delfine und Bären infrage. Dann benötigen Sie eine Frage, die zwischen diesen beiden Tieren unterscheidet – zum Beispiel, ob das Tier Flossen hat.

Diese Fragen lassen sich als Entscheidungsbaum ausdrücken, wie in Abbildung 2-22 gezeigt.

In[55]:

```
mglearn.plots.plot_animal_tree()
```

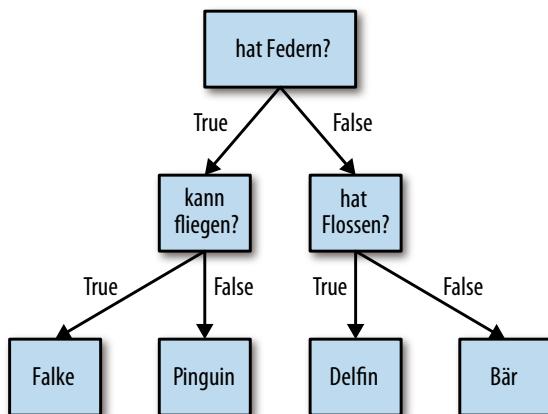


Abbildung 2-22: Ein Entscheidungsbaum zum Unterscheiden einiger Tierarten

In dieser Darstellungsart entspricht jeder Knoten im Baum entweder einer Frage oder einem äußeren Knoten (auch *Blatt* genannt) mit einer Antwort. Die Kanten verbinden die Antworten zu einer Frage mit der jeweils nächsten zu stellenden Frage. In der Terminologie maschinellen Lernens haben wir ein Modell konstruiert, um zwischen vier Kategorien von Tieren zu unterscheiden (Falken, Pinguine, Delfine und Bären). Dazu verwenden wir die drei Merkmale »hat Federn«, »kann fliegen« und »hat Flossen.« Anstatt dieses Modell von Hand zu bauen, können wir es aus den Daten durch überwachtes Lernen konstruieren.

### Aufbauen von Entscheidungsbäumen

Gehen wir einmal den Prozess zum Aufbauen eines Entscheidungsbaumes aus dem zweidimensionalen Klassifikationsdatensatz in Abbildung 2-23 durch. Der Datensatz besteht aus zwei Halbmonden, wobei jede Kategorie aus 75 Datenpunkten besteht. Wir nennen diesen Datensatz im folgenden `two_moons`.

Einen Entscheidungsbau zu trainieren, bedeutet, eine Abfolge von Ja/Nein-Fragen zu erlernen, die so schnell wie möglich zur richtigen Antwort führt. Bei maschinellem Lernen werden diese Fragen *Tests* genannt (nicht zu verwechseln mit dem Testdatensatz, den wir zum Überprüfen der Verallgemeinerbarkeit unseres Modells verwenden). Normalerweise liegen Daten nicht als binäre Ja/Nein-Eigenschaften wie im Tierbeispiel vor, sondern als kontinuierliche Merkmale wie im 2-D-Datensatz in Abbildung 2-23. Die auf kontinuierlichen Daten verwendeten Tests entsprechen der Form »ist das Merkmal  $i$  größer als der Wert  $a$ ?«

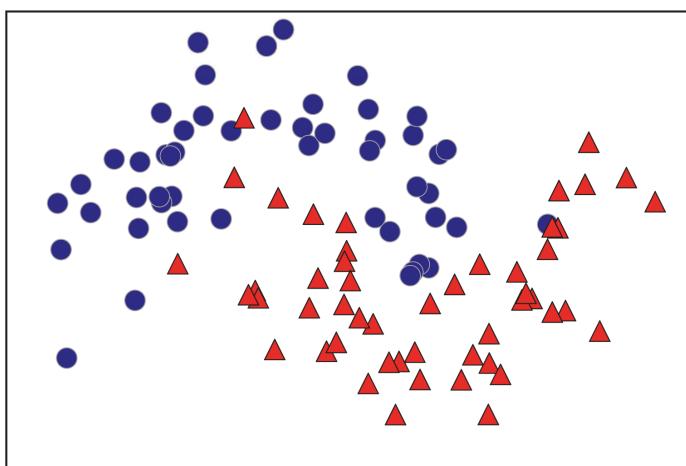


Abbildung 2-23: Two-moons-Datensatz, für den der Entscheidungsbau konstruiert wird

Um einen Baum aufzubauen, durchsucht der Algorithmus alle möglichen Tests und ermittelt denjenigen, der am meisten Information über die Zielgröße enthält. Abbildung 2-24 zeigt den ersten ausgewählten Test. Die vertikale Teilung des Datensatzes bei  $x[1]=0.0596$  liefert die meiste Information; sie trennt die Punkte in Kategorie 0

von den Punkten in Kategorie 1. Der erste Knoten, auch die *Wurzel* genannt, repräsentiert den gesamten Datensatz mit 75 Punkten in Kategorie 0 und 75 Punkten in Kategorie 1. Die Teilung wird durch den Test  $x[1] \leq 0.0596$  vorgenommen und ist als schwarze Linie dargestellt. Ist der Test wahr, wird der Punkt dem linken Knoten zugeordnet, der 2 Punkte aus Kategorie 0 und 32 Punkte aus Kategorie 1 enthält. Andernfalls wird der Punkt dem rechten Knoten mit 48 Punkten aus Kategorie 0 und 18 Punkten aus Kategorie 1 zugeordnet. Diese zwei Knoten entsprechen den oberen und unteren Bereichen in Abbildung 2-24. Auch wenn die erste Teilung die zwei Kategorien sehr gut separiert hat, enthält die untere Region noch immer Punkte aus Kategorie 0, und die obere Region enthält noch immer Punkte aus Kategorie 1. Wir können ein genaueres Modell bauen, indem wir die Suche nach dem besten Test in beiden Regionen wiederholen. Abbildung 2-25 zeigt, dass die ergiebigste Teilung der linken und der rechten Region auf  $x[0]$  beruht.

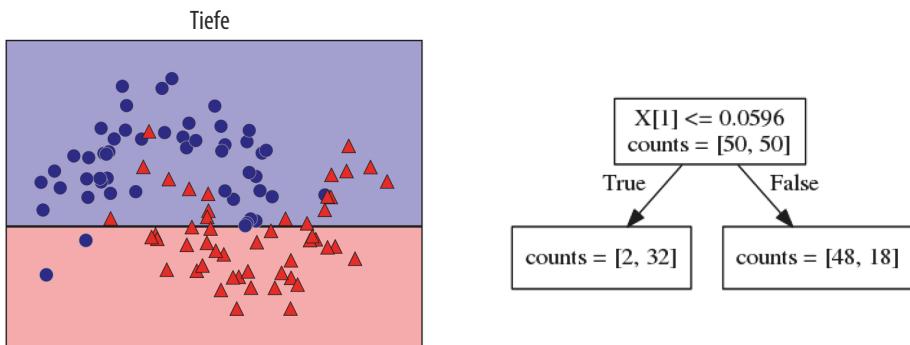


Abbildung 2-24: Entscheidungsgrenze eines Baumes der Tiefe 1 (links) und der dazugehörige Entscheidungsbaum (rechts)

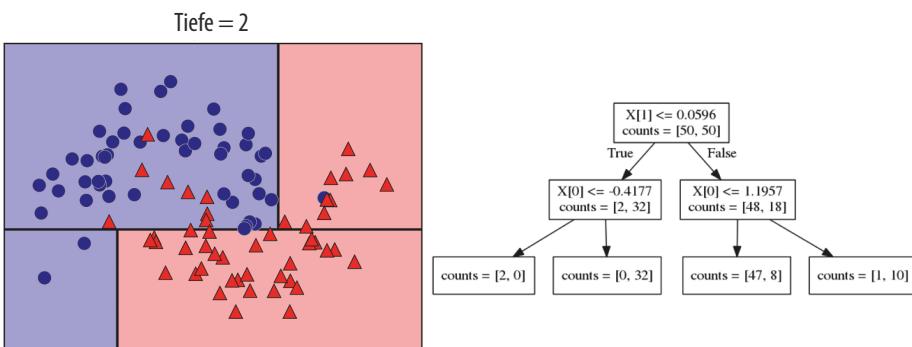


Abbildung 2-25: Entscheidungsgrenze eines Baumes der Tiefe 2 (links) und der dazugehörige Entscheidungsbaum (rechts)

Bei diesem rekursiven Prozess entsteht ein Binärbaum aus Entscheidungen, wobei jeder Knoten einen Test enthält. Alternativ können Sie sich jeden Test als Auftei-

lung der aktuell betrachteten Daten entlang einer Achse vorstellen. Dabei erhalten wir eine Sichtweise auf den Algorithmus als Aufbau einer hierarchischen Teilung. Da jeder Test nur ein einzelnes Merkmal betrifft, haben die Regionen in der resultierenden Aufteilung stets zu den Achsen parallele Begrenzungen.

Die rekursive Aufteilung der Daten wird wiederholt, bis jede Region in der Aufteilung (jedes Blatt im Entscheidungsbaum) nur einen einzigen Zielwert hat (eine einzelne Kategorie oder einen Regressionswert). Ein Blatt im Baum, das nur Datenpunkte mit dem gleichen Zielwert enthält, nennt man *rein*. Die endgültige Aufteilung dieses Datensatzes ist in Abbildung 2-26 dargestellt.

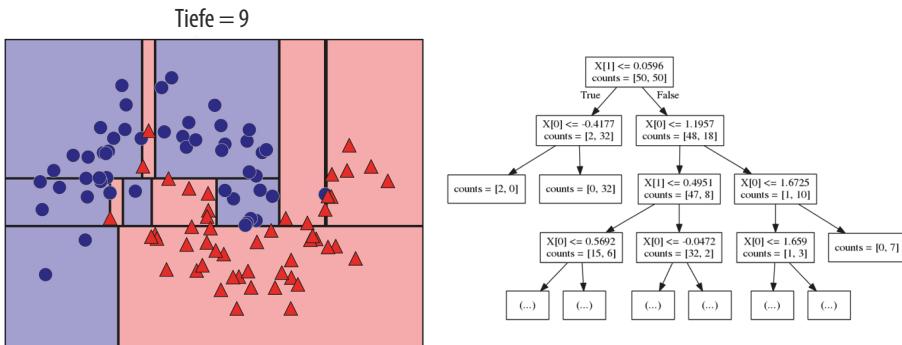


Abbildung 2-26: Entscheidungsgrenze eines Baumes der Tiefe 9 (links) und ein Teil des entsprechenden Baumes (rechts); der vollständige Baum ist recht umfangreich und nur schwer darzustellen.

Für einen neuen Datenpunkt lässt sich eine Vorhersage durchführen, indem überprüft wird, in welcher Region der Aufteilung des Merkmalsraumes der Punkt liegt, um dann den Zielwert der Mehrheit in dieser Region vorherzusagen (oder im Falle von reinen Blättern den einen Zielwert). Diese Region lässt sich finden, indem man den Baum von der Wurzel an durchschreitet und je nach Ergebnis der einzelnen Tests mit dem Knoten links oder rechts fortfährt.

Es ist ebenfalls möglich, Bäume mit genau der gleichen Vorgehensweise für Regressionsaufgaben einzusetzen. Um eine Vorhersage zu treffen, durchschreiten wir den Baum entsprechend den Tests in jedem Knoten und finden das Blatt, zu dem der neue Datenpunkt gehört. Die Ausgabe für diesen Datenpunkt ist dann der mittlere Zielwert aller Trainingspunkte in diesem Blatt.

### Kontrollieren der Komplexität von Entscheidungsbäumen

Wenn man einen Baum wie hier beschrieben aufbaut und so lange fortfährt, bis alle Blätter rein sind, erhält man typischerweise sehr komplexe Modelle, die hochgradig an die Trainingsdaten overfittet sind. Wenn es nur reine Blätter gibt, beträgt die Genauigkeit eines Baumes auf den Trainingsdaten 100 %; jeder Datenpunkt im Trainingsdatensatz befindet sich in einem Blatt mit der korrekten Klasse als Mehrheit. Das Overfitting lässt sich auf der linken Seite in Abbildung 2-26 beobachten.

Sie sehen die der Kategorie 1 zugeordneten Regionen inmitten der Kategorie 0 zugehörigen Punkte. Andererseits gibt es einen schmalen der Kategorie 0 zugeordneten Streifen um den Punkt rechts außen. Dies entspricht nicht dem, wie wir uns die Entscheidungsgrenze vorstellen, und die Entscheidungsgrenze misst hier einzelnen Ausreißern eine hohe Bedeutung zu, obwohl diese weit von anderen Punkten dieser Kategorie entfernt sind.

Es gibt zwei verbreitete Strategien zum Verhindern von Overfitting: den Aufbau des Baumes früh einzustellen (genannt *Prä-Pruning*) oder den Baum aufzubauen, aber anschließend Knoten mit wenig Information zu entfernen oder zusammenzulegen (genannt *Post-Pruning* oder einfach *Pruning*). Mögliche Methoden für das Prä-Pruning sind, die maximale Tiefe des Baumes zu begrenzen, die maximale Anzahl Blätter zu begrenzen oder eine Mindestanzahl von Punkten in einem Knoten zu verlangen, bevor dieser geteilt werden kann.

In scikit-learn sind Entscheidungsbäume in den Klassen `DecisionTreeRegressor` und `DecisionTreeClassifier` implementiert. scikit-learn enthält lediglich Prä-Pruning, aber kein Post-Pruning.

Betrachten wir die Auswirkungen von Prä-Pruning etwas genauer anhand des Brustkrebs-Datensatzes. Wie immer importieren wir den Datensatz und teilen diesen in Trainings- und Testdaten auf. Anschließend konstruieren wir das Modell mit der Standardeinstellung, die einen vollständigen Baum aufbaut (bis sämtliche Blätter rein sind). Wir legen auch einen Wert für `random_state` fest, der im Baum intern zum Auflösen von Pattsituationen verwendet wird:

**In[56]:**

```
from sklearn.tree import DecisionTreeClassifier

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=42)
tree = DecisionTreeClassifier(random_state=0)
tree.fit(X_train, y_train)
print("Genauigkeit auf den Trainingsdaten: {:.3f}".format(
    tree.score(X_train, y_train)))
print("Genauigkeit auf den Testdaten: {:.3f}".format(tree.score(X_test, y_test)))
```

**Out[56]:**

```
Genauigkeit auf den Trainingsdaten: 1.000
Genauigkeit auf den Testdaten: 0.937
```

Wie erwartet, beträgt die Genauigkeit auf dem Trainingsdatensatz 100 % – weil sämtliche Blätter rein sind. Der Baum wurde tief genug entwickelt, sodass er sämtliche Zielwerte in den Trainingsdaten auswendig lernen konnte. Die Genauigkeit auf den Testdaten ist etwas schlechter als bei den oben betrachteten linearen Modellen, für die wir eine Genauigkeit um 95 % erreicht hatten.

Wenn wir die Tiefe eines Entscheidungsbaumes nicht begrenzen, kann der Baum beliebig tief und komplex werden. Daher sind Bäume ohne Pruning sehr anfällig für

Overfitting und können nicht gut auf neue Daten verallgemeinern. Wenden wir nun Prä-Pruning auf den Baum an. Dadurch beenden wir den Aufbau des Baumes, bevor er perfekt an die Trainingsdaten angepasst ist. Eine Möglichkeit ist, den Aufbau anzuhalten, sobald eine bestimmte Tiefe erreicht ist. Hier setzen wir `max_depth=4`, was für maximal vier aufeinanderfolgende Fragen steht (siehe Abbildungen 2-24 und 2-26). Das Begrenzen der Tiefe des Baumes senkt die Anfälligkeit für Overfitting. Damit sinkt auch die Genauigkeit auf den Trainingsdaten, führt bei den Testdaten aber zu einer Verbesserung:

**In[57]:**

```
tree = DecisionTreeClassifier(max_depth=4, random_state=0)
tree.fit(X_train, y_train)

print("Genauigkeit auf den Trainingsdaten: {:.3f}".format(
    tree.score(X_train, y_train)))
print("Genauigkeit auf den Testdaten: {:.3f}".format(tree.score(X_test, y_test)))
```

**Out[57]:**

```
Genauigkeit auf den Trainingsdaten: 0.988
Genauigkeit auf den Testdaten: 0.951
```

## Analysieren von Entscheidungsbäumen

Bäume lassen sich über die Funktion `export_graphviz` aus dem Modul `tree` visualisieren. Diese schreibt eine Datei im Format `.dot`, einem Textformat zum Speichern von Graphen. Wir setzen einen Parameter zum Einfärben der Knoten, um die mehrheitliche Kategorie in jedem Knoten zu kennzeichnen, und übergeben die Namen von Kategorien und Merkmalen, sodass der Baum dementsprechend beschriftet werden kann:

**In[58]:**

```
from sklearn.tree import export_graphviz
export_graphviz(tree, out_file="tree.dot", class_names=["malignant", "benign"],
                feature_names=cancer.feature_names, impurity=False, filled=True)
```

Wir können diese Datei lesen und mit dem Modul `graphviz` wie in Abbildung 2-27 gezeigt visualisieren (Sie können aber auch ein beliebiges Programm zum Lesen von `.dot`-Dateien verwenden):

**In[59]:**

```
import graphviz

with open("tree.dot") as f:
    dot_graph = f.read()
graphviz.Source(dot_graph)
```

Die Visualisierung des Baumes gibt uns im Detail Aufschluss darüber, wie der Algorithmus Vorhersagen trifft, und ist ein hervorragendes Beispiel für einen maschinellen Lernalgorithmus, der sich auch Laien leicht erklären lässt. Allerdings kann schon ein Baum der Tiefe vier wie der hier gezeigte ein wenig überwältigend wir-

ken. Tiefere Bäume (eine Tiefe von 10 ist keine Seltenheit) sind noch schwieriger zu erfassen. Eine hilfreiche Methode zum Inspizieren eines Baumes ist, herauszufinden, welchen Pfad die Mehrheit der Daten beschreitet. Die in Abbildung 2-27 gezeigten Werte für samples in jedem Knoten geben die Anzahl Datenpunkte in diesem Knoten an, während value die Anzahl Datenpunkte pro Kategorie angibt. Folgen wir den Verzweigungen auf der rechten Seite, so sehen wir, dass `worst radius > 16.795` einen Knoten mit nur 8 benignen aber 134 malignen Datenpunkten erstellt. Die übrigen Knoten auf dieser Seite des Baumes verwenden anschließend feinere Unterscheidungen, um auch die restlichen 8 benignen Datenpunkte von den übrigen zu trennen. Von den 142 Punkten in der ursprünglichen Teilung finden sich fast alle (132) im Blatt ganz rechts wieder.

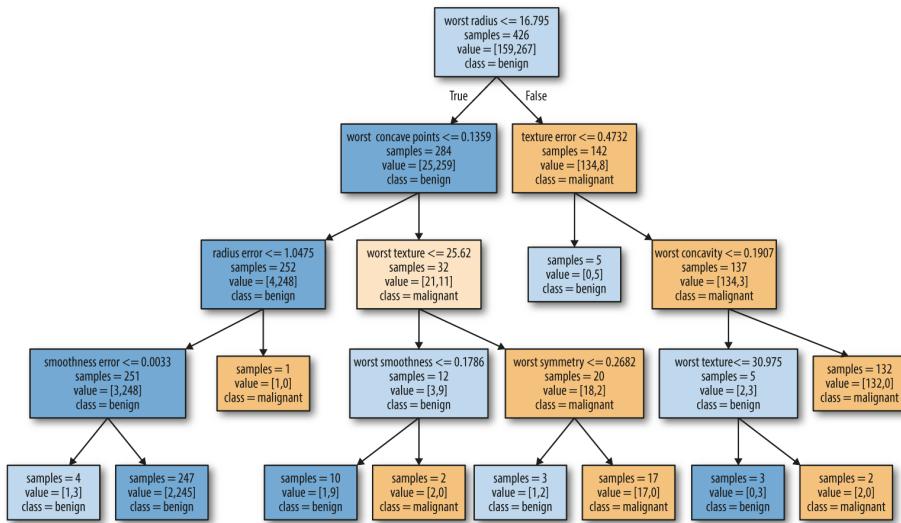


Abbildung 2-27: Visualisierung des auf dem Brustkrebs-Datensatz entwickelten Entscheidungsbaumes

Wenden wir uns von der Wurzel aus nach links, erhalten wir mit `worst radius <= 16.795` noch 25 maligne und 259 benigne Datenpunkte. Fast alle der benigenen Punkte finden sich im zweiten Blatt von links wieder, und die meisten übrigen Blätter enthalten nur wenige Punkte.

## Wichtigkeit von Merkmalen in Bäumen

Anstatt den gesamten Baum zu betrachten, was beschwerlich sein kann, gibt es einige nützliche Eigenschaften, mit denen wir das Innere eines Baumes zusammenfassen können. Das am häufigsten verwendete Maß ist die *Wichtigkeit von Merkmalen*, die die Bedeutung eines Merkmals für die Entscheidung eines Baumes quantifiziert. Sie ist eine Zahl zwischen 0 und 1, wobei 0 für »überhaupt nicht verwendet« steht und 1 für »sagt die Zielgröße perfekt vorher.« Die Summe der Wichtigkeiten aller Merkmale ist immer 1:

In[60]:

```
print("Wichtigkeit der Merkmale:\n{}".format(tree.feature_importances_))
```

Out[60]:

Wichtigkeit der Merkmale:

```
[ 0.         0.         0.         0.         0.         0.         0.         0.         0.         0.         0.         0.01
  0.048     0.         0.         0.002     0.         0.         0.         0.         0.         0.         0.727     0.046
  0.         0.         0.014     0.         0.018     0.122     0.012     0.         ]
```

Wir können die Wichtigkeit der Merkmale in ähnlicher Weise visualisieren wie die Koeffizienten eines linearen Modells (Abbildung 2-28):

In[61]:

```
def plot_feature_importances_cancer(model):
    n_features = cancer.data.shape[1]
    plt.barh(range(n_features), model.feature_importances_, align='center')
    plt.yticks(np.arange(n_features), cancer.feature_names)
    plt.xlabel("Wichtigkeit des Merkmals")
    plt.ylabel("Merkmale")
```

```
plot_feature_importances_cancer(tree)
```

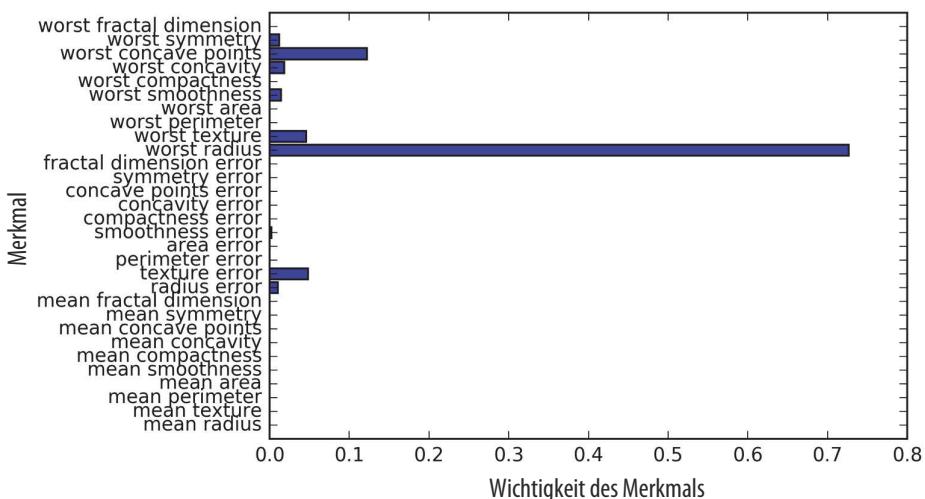


Abbildung 2-28: Wichtigkeiten von Merkmalen, die aus einem auf dem Brustkrebs-Datensatz trainierten Entscheidungsbaum abgeleitet wurden

Wir sehen deutlich, dass das für die erste Teilung verwendete Merkmal (»worst radius«) das bei Weitem wichtigste Merkmal ist. Dies bestätigt unsere Vermutung beim Analysieren des Baumes, dass die erste Ebene die beiden Kategorien bereits recht gut voneinander trennt.

Wenn allerdings ein Merkmal einen niedrigen Wert für `feature_importance_` besitzt, bedeutet das nicht, dass dieses Merkmal uninformativ ist. Es bedeutet nur, dass dieses Merkmal nicht vom Baum ausgewählt wurde, vermutlich weil die gleiche Information auch in einem anderen Merkmal enthalten ist.

Im Gegensatz zu den Koeffizienten in linearen Modellen sind die Wichtigkeiten der Merkmale immer positiv und kodieren nicht, welche Kategorie ein Merkmal bevorzugt. Die Wichtigkeiten der Merkmale sagen uns, dass das Merkmal »worst radius« wichtig ist, aber nicht, ob es ein Indikator für einen benignen oder malignen Datenpunkt ist. Eine derart einfache Beziehung zwischen Merkmal und Kategorie muss nicht einmal existieren, wie das folgende Beispiel zeigt (Abbildungen 2-29 und 2-30):

**In[62]:**

```
tree = mglearn.plots.plot_tree_not_monotone()
display(tree)
```

**Out[62]:**

Wichtigkeit der Merkmale: [ 0. 1.]

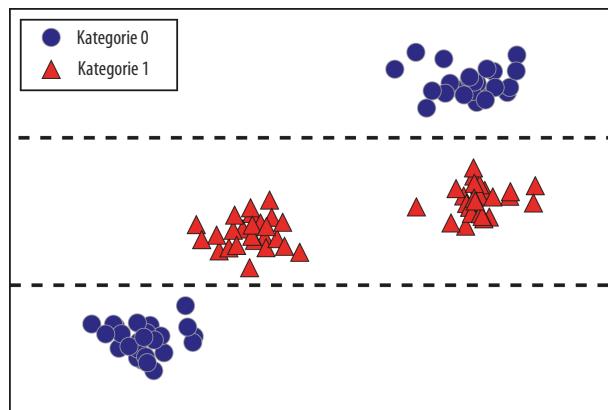


Abbildung 2-29: Ein zweidimensionaler Datensatz, bei dem das Merkmal auf der y-Achse in einer nicht monotonen Beziehung zur Kategoriebezeichnung steht, sowie die vom Entscheidungsbaum ermittelten Entscheidungsgrenzen

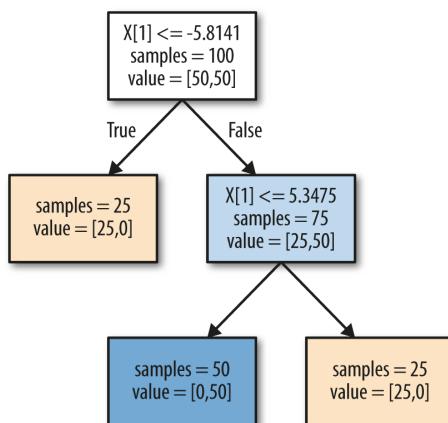


Abbildung 2-30: Mit den Daten in Abbildung 2-29 trainierter Entscheidungsbaum

Das Diagramm zeigt einen Datensatz mit zwei Merkmalen und zwei Klassen. Hierbei sind sämtliche Informationen in  $X[1]$  enthalten, und  $X[0]$  wird überhaupt nicht verwendet. Die Beziehung zwischen  $X[1]$  und der Zielkategorie ist jedoch nicht monoton. Daher können wir nicht sagen: »Ein hoher Wert für  $X[1]$  steht für Kategorie 0, und ein niedriger Wert steht für Klasse 1« (oder umgekehrt).

Während sich unsere Betrachtung auf Entscheidungsbäume zur Klassifikation konzentriert hat, trifft alles Gesagte in ähnlicher Weise auch für Entscheidungsbäume zur Regression zu, die im `DecisionTreeRegressor` implementiert sind. Die Verwendung und Analyse von Regressionsbäumen ist denen zu Klassifikationsbäumen sehr ähnlich. Wir möchten jedoch an dieser Stelle eine Eigenschaft von Regressionsbäumen hervorheben. Der `DecisionTreeRegressor` (und alle anderen baumbasierten Regressionsmodelle) ist nicht in der Lage, zu *extrapolieren* oder Vorhersagen außerhalb des Wertebereichs der Trainingsdaten zu treffen.

Sehen wir uns diese Eigenschaft bei einem Datensatz historischer Preise für Computerspeicher (RAM) genauer an. Abbildung 2-31 stellt den Datensatz mit dem Datum auf der x-Achse und dem Preis für ein Megabyte RAM im jeweiligen Jahr auf der y-Achse dar:

**In[63]:**

```
import pandas as pd
ram_prices = pd.read_csv("data/ram_price.csv")

plt.semilogy(ram_prices.date, ram_prices.price)
plt.xlabel("Jahr")
plt.ylabel("Preis in $/Mbyte")
```

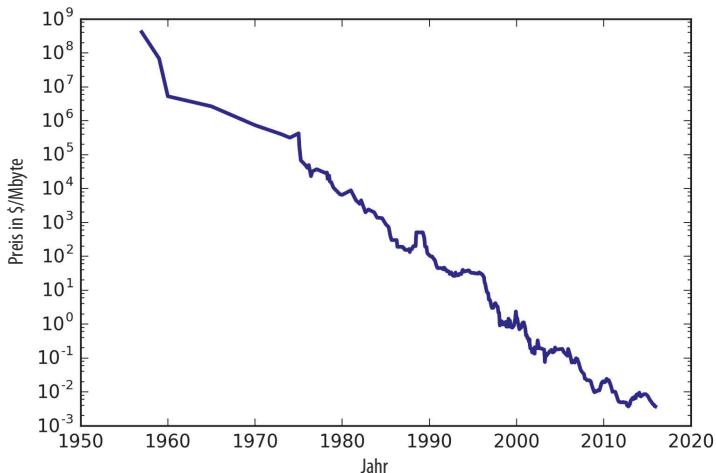


Abbildung 2-31: Historische Entwicklung der Preise für RAM auf einer logarithmischen Skala

Beachten Sie die logarithmische Skala der y-Achse. Wenn wir diese Daten logarithmisch auftragen, wirkt die Beziehung linear und sollte daher, von einigen Hügeln einmal abgesehen, recht einfach vorherzusagen sein.

Wir werden mit den historischen Daten bis zum Jahr 2000 eine Vorhersage für die Jahre danach treffen, wobei das Datum unser einziges Merkmal ist. Wir werden zwei einfache Modelle vergleichen: `DecisionTreeRegressor` und `LinearRegression`. Wir logarithmieren die Preise, sodass sich eine einigermaßen lineare Beziehung ergibt. Für den `DecisionTreeRegressor` macht dies keinen Unterschied, für `LinearRegression` hingegen einen großen (wir werden diesen Umstand ausführlicher in Kapitel 4 erörtern). Nach dem Trainieren der Modelle und der Vorhersage wenden wir eine Exponentialfunktion an, um die logarithmische Transformierung rückgängig zu machen. Wir treffen zur Veranschaulichung Vorhersagen für den gesamten Datensatz, zur quantitativen Auswertung würden wir aber nur die Testdaten heranziehen:

**In[64]:**

```
from sklearn.tree import DecisionTreeRegressor
# verwende historische Daten, um Preise nach 2000 vorherzusagen
data_train = ram_prices[ram_prices.date < 2000]
data_test = ram_prices[ram_prices.date >= 2000]

# sage Preise anhand des Datums vorher
X_train = data_train.date[:, np.newaxis]
# verwende eine log-Transformierung, um die Zielgröße zu vereinfachen
y_train = np.log(data_train.price)

tree = DecisionTreeRegressor().fit(X_train, y_train)
linear_reg = LinearRegression().fit(X_train, y_train)

# sage sämtliche Daten vorher
X_all = ram_prices.date[:, np.newaxis]

pred_tree = tree.predict(X_all)
pred_lr = linear_reg.predict(X_all)

# mache die log-Transformierung rückgängig
price_tree = np.exp(pred_tree)
price_lr = np.exp(pred_lr)
```

Abbildung 2-32 vergleicht die Vorhersagen des Entscheidungsbaumes mit denen des linearen Regressionsmodells und den tatsächlichen Werten:

**In[65]:**

```
plt.semilogy(data_train.date, data_train.price, label="Trainingsdaten")
plt.semilogy(data_test.date, data_test.price, label="Testdaten")
plt.semilogy(ram_prices.date, price_tree, label="Tree prediction")
plt.semilogy(ram_prices.date, price_lr, label="Linear prediction")
plt.legend()
```

Der Unterschied zwischen den Modellen ist recht erstaunlich. Das lineare Modell approximiert die Daten wie erwartet durch eine Gerade. Diese Gerade liefert eine recht gute Vorhersage für die Testdaten (die Jahre nach 2000) und sieht über die kleineren Schwankungen in Trainings- und Testdaten hinweg. Der Entscheidungsbaum dagegen trifft für die Trainingsdaten perfekte Vorhersagen; wir haben die Komplexität des Baumes nicht eingeschränkt, daher hat das Modell den gesamten

Datensatz auswendig gelernt. Wenn wir allerdings den Datumsbereich verlassen, für den dem Modell Daten vorliegen, sagt das Modell einfach den letzten bekannten Punkt vorher. Der Baum ist nicht in der Lage, »neue« Antworten außerhalb des in den Trainingsdaten gesehenen Bereichs zu generieren. Diese Einschränkung gilt für alle auf Bäumen basierenden Modelle.<sup>9</sup>

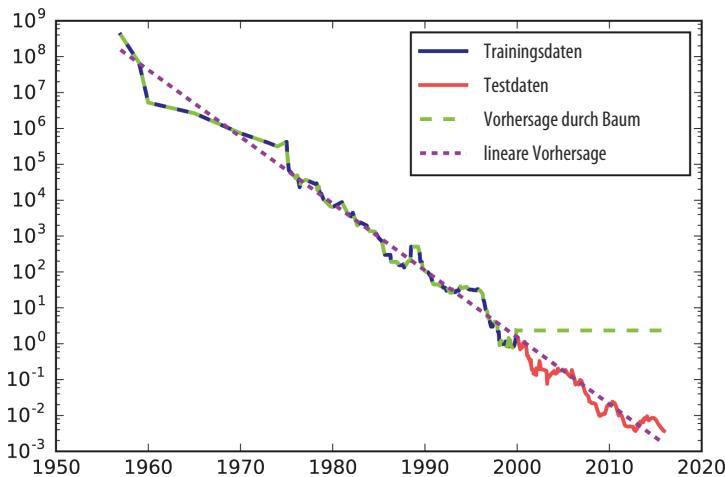


Abbildung 2-32: Vergleich der Vorhersagen durch ein lineares Modell mit denen eines Regressionsbaumes auf dem Datensatz von RAM-Preisen

### Stärken, Schwächen und Parameter

Wie weiter oben erwähnt, dienen die Prä-Pruning-Parameter zum Kontrollieren der Modellkomplexität von Entscheidungsbäumen, indem sie den Aufbau des Baumes stoppen, bevor dieser vollständig entwickelt ist. Üblicherweise genügt es zum Prä-Pruning, einen der Parameter `max_depth`, `max_leaf_nodes` oder `min_samples_leaf` zu setzen, um Overfitting zu verhindern.

Entscheidungsbäume haben gegenüber den meisten bisher betrachteten Algorithmen zwei Vorteile vorzuweisen: Das fertige Modell lässt sich einfach visualisieren und ist auch von Nicht-Experten zu verstehen (zumindest bei kleineren Bäumen), und die Algorithmen sind vollständig von der Skalierung der Daten unabhängig. Da jedes Merkmal einzeln abgearbeitet wird und die einzelnen Teilungen der Daten nicht von der Skalierung abhängen, ist bei Entscheidungsbäumen keinerlei Vorverarbeitung wie Normalisierung oder Standardisierung von Merkmalen notwendig. Entscheidungsbäume funktionieren insbesondere auch dann gut, wenn Ihnen

---

<sup>9</sup> Es ist möglich, mit baumbasierten Modellen sehr gute Vorhersagen zu treffen (z. B. vorherzusagen, ob ein Preis sich nach oben oder unten entwickeln wird). Das Entscheidende bei diesem Beispiel ist nicht, zu zeigen, dass Bäume für Zeitreihen ungeeignet sind, sondern zu veranschaulichen, dass Bäume Vorhersagen auf eine bestimmte Weise treffen.

Merkmale auf vollkommen unterschiedlichen Skalen vorliegen oder binäre und kontinuierliche Merkmale gemischt auftreten.

Der Hauptnachteil von Entscheidungsbäumen ist, dass sie sogar mit Prä-Pruning zum Overfitting neigen und dann kaum verallgemeinern. Daher kommen bei den meisten Anwendungen die als Nächstes besprochenen Ensemble-Methoden anstelle einzelner Entscheidungsbäume zum Einsatz.

## Ensembles von Entscheidungsbäumen

*Ensembles* sind Methoden, die mehrere maschinelle Lernmodelle miteinander kombinieren, um mächtigere Modelle zu erstellen. In der Literatur zu maschinellem Lernen sind viele Modelle aus dieser Kategorie beschrieben, es haben sich aber nur zwei Ensemble-Modelle bei einer großen Bandbreite von Datensätzen als effektiv erwiesen. Beide verwenden Entscheidungsbäume als Grundkomponente: Random Forests und Entscheidungsbäume mit Gradient Boosting.

### Random Forests

Wie wir soeben beobachten konnten, ist die Tendenz zum Overfitting auf den Trainingsdaten der Hauptnachteil von Entscheidungsbäumen. Random Forests sind eine Möglichkeit, dieses Problem zu umgehen. Ein Random Forest ist im Wesentlichen eine Menge von Entscheidungsbäumen, wobei sich jeder Baum ein wenig von den übrigen unterscheidet. Die Idee bei Random Forests ist, dass jeder Baum eine recht gute Vorhersage treffen kann, aber voraussichtlich einen Teil der Daten overfittet. Wenn wir viele Bäume konstruieren, die alle gut funktionieren und auf unterschiedliche Weise overfitten, können wir durch Mitteln der Ergebnisse das Overfitting reduzieren. Dieses Herausrechnen des Overfittings beim gleichzeitigen Bewahren der Vorhersagekraft der Bäume lässt sich mathematisch exakt beweisen.

Um diese Strategie umzusetzen, benötigen wir viele Entscheidungsbäume. Jeder Baum sollte bei der Vorhersage der Zielgröße eine akzeptable Leistung erbringen und sich gleichzeitig von den übrigen Bäumen unterscheiden. Random Forests haben ihren Namen daher, dass beim Aufbauen der Bäume ein Zufallselement dafür sorgt, dass sich die Bäume unterscheiden. Es gibt zwei Arten von Zufall in Random Forests: durch Auswahl der Datenpunkte beim Aufbau eines Baumes und durch Auswahl der zu testenden Merkmale bei jeder Teilung. Gehen wir diesen Prozess im Einzelnen durch.

**Erstellen von Random Forests.** Um ein Random Forest-Modell zu konstruieren, müssen Sie die Anzahl der zu erstellenden Bäume festlegen (über den Parameter `n_estimators` in `RandomForestRegressor` oder `RandomForestClassifier`). Sagen wir, Sie möchten zehn Bäume erstellen. Diese Bäume werden vollständig unabhängig voneinander erstellt, und der Algorithmus trifft für jeden Baum unterschiedliche Zufallsentscheidungen, um sicherzustellen, dass die Bäume unterschiedlich sind.

Um einen Baum zu konstruieren, verwenden wir zunächst eine sogenannte *Bootstrap-Stichprobe* unserer Daten. Das heißt, wir wählen wiederholt aus unseren `n_samples` Datenpunkten `n_samples` Mal zufällig und mit Zurücklegen aus (das heißt, der gleiche Datenpunkt kann mehrfach ausgewählt werden). Dadurch erhalten wir einen Datensatz, der genau so groß ist wie der ursprüngliche, aber einige Datenpunkte (ungefähr ein Drittel) fehlen, während andere mehrfach auftreten.

Um dies zu veranschaulichen, erstellen wir eine Bootstrap-Stichprobe der Liste `['a', 'b', 'c', 'd']`. Eine mögliche Bootstrap-Stichprobe wäre `['b', 'd', 'd', 'c']`. Eine weitere mögliche Stichprobe wäre `['d', 'a', 'd', 'a']`.

Als Nächstes wird auf diesem neuen Datensatz ein Entscheidungsbaum konstruiert. Allerdings ist der Algorithmus zu der obigen Beschreibung von Entscheidungsbäumen leicht verändert. Anstatt den besten Test für einen Knoten zu ermitteln, wählt der Algorithmus zufällig eine Teilmenge der Merkmale aus und ermittelt den bestmöglichen Test für eines der ausgewählten Merkmale. Die Anzahl ausgewählter Merkmale lässt sich über den Parameter `max_features` festlegen. Diese Auswahl von Merkmalen wird für jeden Knoten separat wiederholt, sodass jeder Knoten im Baum anhand einer anderen Auswahl von Merkmalen eine Entscheidung treffen kann.

Durch die Bootstrap-Stichproben wird jeder Entscheidungsbaum im Random Forest auf einem leicht veränderten Datensatz aufgebaut. Wegen der Auswahl von Merkmalen in jedem Knoten wird jede Teilung an einer anderen Untermenge von Merkmalen vorgenommen. Zusammen stellen diese zwei Mechanismen sicher, dass sich sämtliche Bäume im Random Forest voneinander unterscheiden.

Ein entscheidender Parameter in diesem Prozess ist `max_features`. Wenn wir `max_features` auf `n_features` setzen, werden bei jeder Teilung sämtliche Merkmale des Datensatzes berücksichtigt, und in die Auswahl von Merkmalen fließt kein Zufall ein (es bleibt nur das Bootstrapping als Zufallselement). Wenn wir dagegen `max_features` auf 1 setzen, gibt es bei den Teilungen gar keine Auswahlmöglichkeit unter den Merkmalen, und es können nur noch unterschiedliche Schwellenwerte für das zufällig ausgewählte Merkmal ausprobiert werden. Daher führt eine hoher Wert für `max_features` dazu, dass die Bäume im Random Forest einander recht ähnlich sind und dass sie die Daten leicht anhand herausragender Merkmale erlernen können. Ein niedriger Wert für `max_features` führt dazu, dass die Bäume im Random Forest sehr unterschiedlich sind und dass jeder einzelne Baum unter Umständen sehr tief sein muss, um die Daten gut abzubilden.

Um mithilfe des Random Forest eine Vorhersage zu treffen, trifft der Algorithmus zunächst eine Vorhersage für jeden einzelnen der Bäume. Bei einer Regression können wir die endgültige Vorhersage durch Mittelwertbildung erhalten. Bei einer Klassifikation wird die »soft voting«-Strategie verwendet. Dabei liefert jeder Baum eine »weiche« Vorhersage ab, bei der es für jeden möglichen Ausgabewert eine Wahrscheinlichkeit gibt. Die Wahrscheinlichkeiten aller Bäume werden gemittelt, und die Kategorie mit der höchsten Wahrscheinlichkeit wird vorhergesagt.

**Analysieren von Random Forests.** Wenden wir einen aus fünf Bäumen bestehenden Random Forest auf den bereits betrachteten Datensatz `two_moons` an:

In[66]:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=100, noise=0.25, random_state=3)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
                                                    random_state=42)

forest = RandomForestClassifier(n_estimators=5, random_state=2)
forest.fit(X_train, y_train)
```

Die als Teil des Random Forest konstruierten Bäume sind im Attribut `estimators_` gespeichert. Visualisieren wir die von jedem der Bäume erlernten Entscheidungsgrenzen sowie die vom gesamten Random Forest getroffene Vorhersage (Abbildung 2-33):

In[67]:

```
fig, axes = plt.subplots(2, 3, figsize=(20, 10))
for i, (ax, tree) in enumerate(zip(axes.ravel(), forest.estimators_)):
    ax.set_title("Baum {}".format(i))
    mglearn.plots.plot_tree_partition(X_train, y_train, tree, ax=ax)

mglearn.plots.plot_2d_separator(forest, X_train, fill=True, ax=axes[-1, -1],
                                alpha=.4)
axes[-1, -1].set_title("Random Forest")
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
```

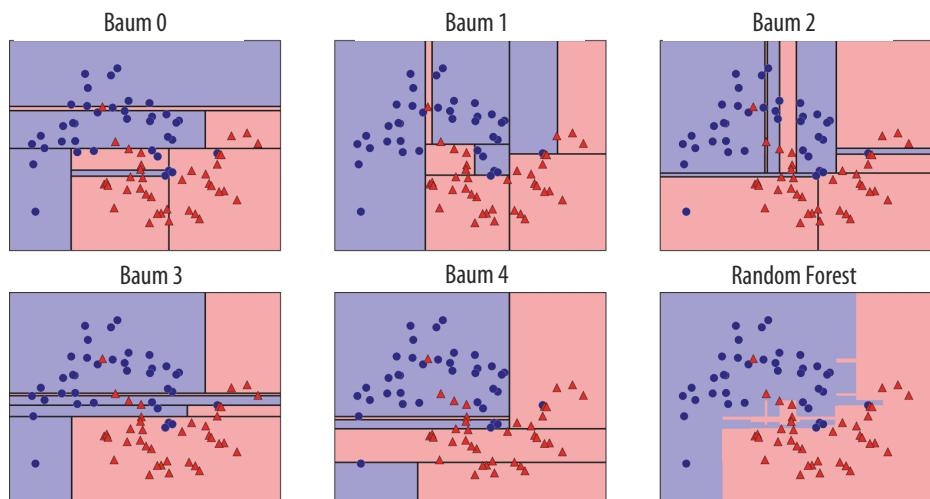


Abbildung 2-33: Von fünf randomisierten Entscheidungsbäumen ermittelte Entscheidungsgrenzen und die durch Mitteln der vorhergesagten Wahrscheinlichkeiten erhaltenen Entscheidungsgrenzen

Sie sehen deutlich, dass sich die Entscheidungsgrenzen der fünf Bäume recht stark unterscheiden. Jeder Baum begeht einige Fehler, da durch das Bootstrapping manche der hier dargestellten Trainingsdatenpunkte gar nicht in den Trainingspunkten der Bäume enthalten waren. Das Overfitting ist beim Random Forest geringer als bei jedem der Bäume für sich allein, und die Entscheidungsgrenze ist sehr viel einprägsamer. In einer echten Anwendung würde man sehr viel mehr Bäume verwenden (oft Hunderte oder Tausende), wodurch die Grenzen noch weicher werden.

Als weiteres Beispiel wenden wir einen Random Forest aus 100 Bäumen auf den Brustkrebs-Datensatz an:

**In[68]:**

```
X_train, X_test, y_train, y_test = train_test_split(  
    cancer.data, cancer.target, random_state=0)  
forest = RandomForestClassifier(n_estimators=100, random_state=0)  
forest.fit(X_train, y_train)  
  
print("Genauigkeit auf den Trainingsdaten: {:.3f}".format(forest.score(X_train,  
    y_train)))  
print("Genauigkeit auf den Testdaten: {:.3f}".format(forest.score(X_test, y_test)))
```

**Out[68]:**

```
Genauigkeit auf den Trainingsdaten: 1.000  
Genauigkeit auf den Testdaten: 0.972
```

Die Genauigkeit des Random Forest beträgt 97 % und ist damit besser als die linearen Modelle oder ein einzelner Entscheidungsbaum, ohne dass wir Parameter eingestellt hätten. Wir könnten den Wert für `max_features` modifizieren oder Prä-Pruning wie bei einem einfachen Entscheidungsbaum vornehmen. Die Standardparameter eines Random Forest funktionieren aber bereits recht gut.

Ähnlich wie ein Entscheidungsbaum gibt auch ein Random Forest die Wichtigkeit von Merkmalen an. Diese werden durch Aggregieren der Merkmalswichtigkeiten aller Bäume im Forest berechnet. Üblicherweise sind die vom Random Forest berechneten Wichtigkeiten zuverlässiger als die von einem einzelnen Baum angegebenen. Betrachten Sie dazu Abbildung 2-34.

**In[69]:**

```
plot_feature_importances_cancer(forest)
```

Wie Sie sehen, verleiht der Random Forest viel mehr Merkmalen eine Wichtigkeit ungleich null als ein einzelner Baum. Wie beim einzelnen Entscheidungsbaum misst auch der Random Forest dem Merkmal »worst radius« eine Menge Bedeutung zu, aber als insgesamt informativstes Merkmal wird »worst perimeter« ausgewählt. Der Zufall beim Aufbau des Random Forest zwingt den Algorithmus dazu, viele mögliche Erklärungen zu berücksichtigen. Dies führt dazu, dass der Random Forest ein wesentlich weiteres Spektrum der Daten erfasst als ein einzelner Baum.

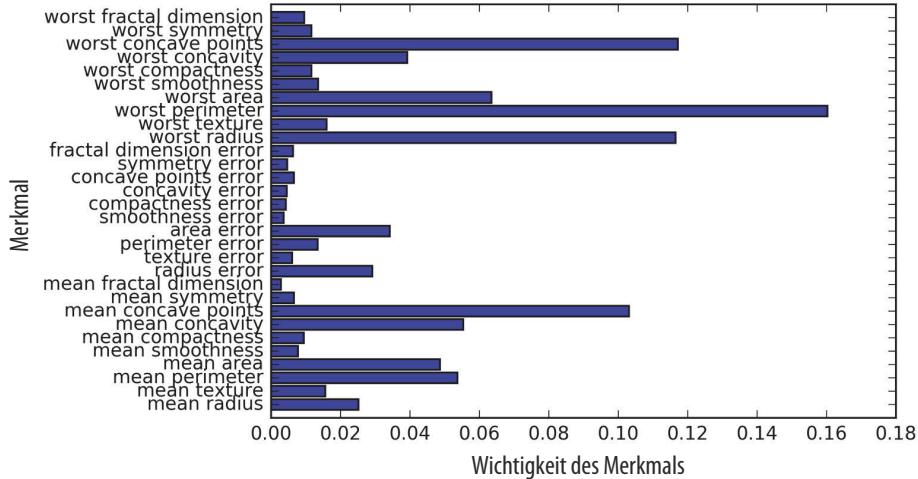


Abbildung 2-34: Aus einem Random Forest berechnete Wichtigkeit von Merkmalen für den Brustkrebs-Datensatz

**Stärken, Schwächen und Parameter.** Random Forests gehören gegenwärtig zu den verbreitetsten maschinellen Lernmethoden zur Regression und Klassifikation. Sie sind sehr mächtig, funktionieren häufig, ohne viel an den Parametern einstellen zu müssen, und erfordern kein Skalieren der Daten.

Im Wesentlichen teilen Random Forests sämtliche Vorteile von Entscheidungsbäumen und umgehen einige ihrer Nachteile. Wenn Sie jedoch eine kompakte Repräsentation des Entscheidungsprozesses benötigen, wäre dies ein Grund, trotzdem Entscheidungsbäume anstelle von Random Forests zu verwenden. Es ist praktisch unmöglich, Dutzende oder Hunderte von Bäumen im Detail zu interpretieren, und Bäume in einem Random Forest sind eher tiefer als einzelne Entscheidungsbäume (da erstere Teilmengen von Merkmalen verwenden). Wenn Sie also die Entstehung einer Vorhersage für Laien visuell zusammenfassen möchten, kann ein einzelner Entscheidungsbau die bessere Wahl sein. Auch wenn die Konstruktion von Random Forests auf großen Datensätzen recht zeitaufwendig werden kann, lässt sie sich leicht auf mehrere CPU Cores in einem Computer aufteilen. Wenn Sie einen Multi-Core-Prozessor verwenden (wie die meisten modernen Computer), können Sie die Anzahl verwendeter Cores über den Parameter `n_jobs` einstellen. Mehr CPU Cores führen zu einer linearen Beschleunigung (mit zwei Cores wird das Trainieren des Random Forest doppelt so schnell), aber ein Wert für `n_jobs` über der Anzahl verfügbarer Cores hilft nicht. Sie können auch `n_jobs=-1` angeben, um alle Cores in Ihrem Computer zu verwenden.

Sie sollten bedenken, dass Random Forests auf Zufall basieren, und unterschiedliche Zustände für den Zufallsgenerator anzugeben (oder gar keinen Wert für `random_state`), kann das erstellte Modell drastisch verändern. Je mehr Bäume der

Forest enthält, umso robuster wird dieser gegenüber Änderungen im Zufallsgenerator. Wenn Sie reproduzierbare Ergebnisse erhalten möchten, ist es wichtig, `random_state` festzulegen.

Auf dünn besetzten Daten mit sehr vielen Dimensionen, etwa Textdaten, schneiden Random Forests nicht gut ab. Für diese Art Daten sind lineare Modelle besser geeignet. Random Forests funktionieren normalerweise auch für sehr große Datensätze gut, und mit einem leistungsstarken Rechner kann das Trainieren auf viele CPU Cores verteilt werden. Allerdings benötigen Random Forests mehr Speicher und mehr Zeit beim Trainieren und bei der Vorhersage als lineare Modelle. Falls Zeit und Speicherverbrauch in Ihrer Anwendung wichtig sind, ist möglicherweise ein lineares Modell angebrachter.

Die wichtigen einzustellenden Parameter sind `n_estimators`, `max_features` und eventuell Optionen zum Prä-Pruning wie `max_depth`. Bei `n_estimators` ist ein höherer Wert stets besser. Der Durchschnitt vieler Bäume liefert ein robusteres Ensemble und reduziert Overfitting. Andererseits benötigen mehr Bäume auch mehr Speicher und Rechenzeit. Eine verbreitete Faustregel ist, so viele Bäume zu konstruieren, »wie Sie Zeit und Speicher übrig haben«.

Wie oben angegeben, legt `max_features` fest, wie stark zufällig die Bäume sind, wobei ein kleiner Wert für `max_features` das Overfitting verringert. Im Allgemeinen ist es eine gute Faustregel, den Standardwert zu verwenden: `max_features=sqrt(n_features)` bei der Klassifikation und `max_features=n_features` bei der Regression. Das Setzen von `max_features` oder `max_leaf_nodes` kann bisweilen die Genauigkeit verbessern. Es kann auch drastisch den erforderlichen Speicher oder die Zeit beim Trainieren und Vorhersagen senken.

## **Régressionsbäume mit Gradient Boosting (Gradient Boosting Machines)**

Régressionsbäume mit Gradient Boosting sind eine weitere Ensemble-Methode, bei der mehrere Entscheidungsbäume zu einem mächtigeren Modell kombiniert werden. Trotz des Wortes »Regression« im Namen lassen sich diese Modelle sowohl zur Regression als auch zur Klassifikation einsetzen. Im Gegensatz zu Random Forests werden beim Gradient Boosting die Bäume nacheinander erstellt, wobei jeder Baum versucht, die Fehler des vorigen nachzubessern. Es gibt bei Régressionsbäumen mit Gradient Boosting standardmäßig kein Zufallselement; stattdessen kommt starkes Prä-Pruning zum Einsatz. Gradient Boosting verwendet oft sehr flache Bäume mit Tiefen von eins bis fünf, wodurch das Modell im Speicher weniger Platz einnimmt und schnelle Vorhersagen möglich sind. Die Grundidee beim Gradient Boosting ist, viele einfache Modelle (in diesem Zusammenhang als *schwache Lerner*) miteinander zu kombinieren, beispielsweise flache Bäume. Jeder Baum kann nur auf einem Teil der Daten gute Vorhersagen liefern, und daher werden buchstäblich mehr und mehr Bäume hinzugefügt, um die Leistung nach und nach zu verbessern.

Bäume mit Gradient Boosting finden sich häufig auf den Gewinnerplätzen von Wettbewerben in maschinellem Lernen und kommen in der Wirtschaft großflächig zum Einsatz. Sie sind im Allgemeinen gegenüber den Parametern etwas empfindlicher als Random Forests, aber mit korrekt eingestellten Parametern liefern sie eine höhere Genauigkeit.

Außer Prä-Pruning und der Anzahl Bäume im Ensemble ist auch der learning\_rate-Parameter wichtig. Dieser kontrolliert, wie stark jeder Baum die Fehler seiner Vorgänger zu korrigieren versucht. Eine höhere Lernrate bedeutet, dass jeder Baum stärkere Korrekturen vornehmen darf, wodurch komplexere Modelle entstehen. Das Hinzufügen weiterer Bäume zum Ensemble durch Erhöhen von n\_estimators steigert die Komplexität ebenfalls, da das Modell mehr Gelegenheiten erhält, Fehler auf den Trainingsdaten zu korrigieren.

Als Beispiel verwenden wir GradientBoostingClassifier auf dem Brustkrebs-Datensatz. Standardmäßig werden 100 Bäume mit einer maximalen Tiefe von 3 und einer Lernrate von 0.1 verwendet:

**In[70]:**

```
from sklearn.ensemble import GradientBoostingClassifier

X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train)

print("Genauigkeit auf den Trainingsdaten: {:.3f}".format(gbret.score(X_train, y_train)))
print("Genauigkeit auf den Testdaten: {:.3f}".format(gbret.score(X_test, y_test)))
```

**Out[70]:**

```
Genauigkeit auf den Trainingsdaten: 1.000
Genauigkeit auf den Testdaten: 0.958
```

Da die Genauigkeit auf den Trainingsdaten 100 % beträgt, liegt vermutlich Overfitting vor. Um das Overfitting zu verringern, können wir das Prä-Pruning verstärken, indem wir die maximale Tiefe begrenzen oder die Lernrate verringern:

**In[71]:**

```
gbret = GradientBoostingClassifier(random_state=0, max_depth=1)
gbret.fit(X_train, y_train)

print("Genauigkeit auf den Trainingsdaten: {:.3f}".format(gbret.score(X_train, y_train)))
print("Genauigkeit auf den Testdaten: {:.3f}".format(gbret.score(X_test, y_test)))
```

**Out[71]:**

```
Genauigkeit auf den Trainingsdaten: 0.991
Genauigkeit auf den Testdaten: 0.972
```

In[72]:

```
gbrt = GradientBoostingClassifier(random_state=0, learning_rate=0.01)
gbrt.fit(X_train, y_train)

print("Genauigkeit auf den Trainingsdaten: {:.3f}".format(gbdt.score(X_train, y_train)))
print("Genauigkeit auf den Testdaten: {:.3f}".format(gbdt.score(X_test, y_test)))
```

Out[72]:

```
Genauigkeit auf den Trainingsdaten: 0.988
Genauigkeit auf den Testdaten: 0.965
```

Beide Ansätze zum Verringern der Modellkomplexität haben die Genauigkeit auf den Trainingsdaten wie erwartet verringert. In diesem Fall führt das Verringern der maximalen Tiefe zu einer deutlichen Verbesserung des Modells, während das Verringern der Lernrate die Vorhersageleistung nur leicht erhöht.

Wie bei den anderen auf Entscheidungsbäumen basierenden Modellen können wir auch hier die Wichtigkeiten der Merkmale visualisieren, um mehr über unser Modell zu erfahren (Abbildung 2-35). Da wir 100 Bäume verwendet haben, ist es nicht praktikabel, alle zu inspizieren, selbst wenn sie eine Tiefe von 1 haben:

In[73]:

```
gbdt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbdt.fit(X_train, y_train)

plot_feature_importances_cancer(gbdt)
```

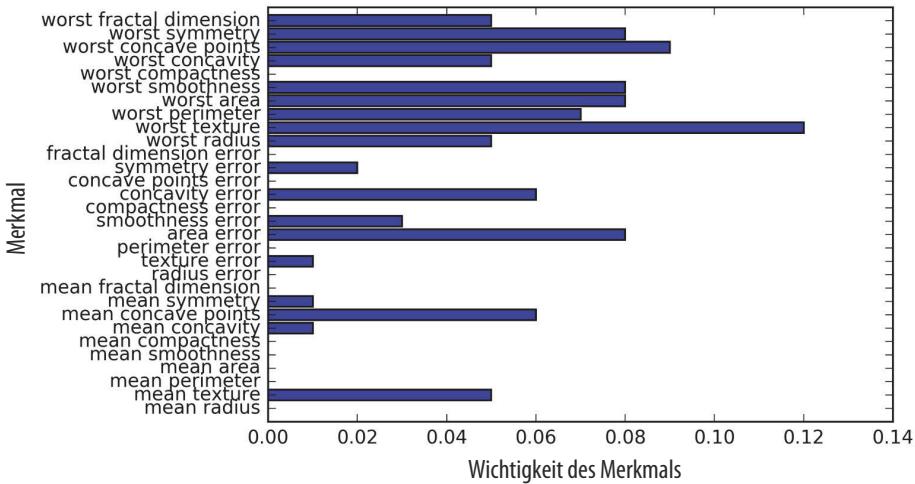


Abbildung 2-35: Aus einem Klassifikator mit Gradient Boosting auf dem Brustkrebs-Datensatz berechnete Wichtigkeiten von Merkmalen

Wir beobachten, dass die Wichtigkeiten der Merkmale von Bäumen mit Gradient Boosting ähnlich zu denen von Random Forests sind, auch wenn das Gradient Boosting einige Merkmale vollkommen ignoriert.

Da sowohl Gradient Boosting als auch Random Forests auf verschiedenartigen Daten gut funktionieren, ist ein verbreiteter Ansatz, zuerst den robusteren Random Forest auszuprobieren. Wenn dieser gut funktioniert, die Vorhersage jedoch lange dauert oder man das letzte Quäntchen Genauigkeit aus dem maschinellen Lernmodell herausquetschen möchte, hilft meist ein Wechsel auf Gradient Boosting.

Wenn Sie Gradient Boosting bei einer größeren Aufgabe verwenden möchten, lohnt ein Blick auf das Paket xgboost und dessen Python-Schnittstelle. Dieses ist bei vielen Datensätzen gegenwärtig schneller (und manchmal leichter zu optimieren) als die Implementierung in scikit-learn.

**Stärken, Schwächen und Parameter.** Entscheidungsbäume mit Gradient Boosting gehören zu den mächtigsten verbreiteten Modellen für überwachtes Lernen. Ihr Hauptnachteil ist, dass man deren Parameter umsichtig einstellen muss und dass Sie lange zum Trainieren benötigen. Ähnlich wie bei den anderen auf Bäumen basierenden Modellen funktioniert der Algorithmus auch ohne Skalierung gut und kann mit einem Mix aus binären und kontinuierlichen Merkmalen umgehen. Wie bei anderen auf Bäumen basierenden Modellen kann er oft nicht so gut mit hochdimensionalen dünn besetzten Daten umgehen.

Die wichtigsten Parameter von Modellen mit Gradient Boosting sind die Anzahl der Bäume, `n_estimators` und `learning_rate`, mit dem sich die Stärke der Korrektur von Fehlern vorhergehender Bäume einstellen lässt. Diese zwei Parameter hängen stark voneinander ab, da ein niedrigerer Wert für `learning_rate` mehr Bäume erfordert, um ein ähnlich komplexes Modell zu konstruieren. Im Gegensatz zu Random Forests, bei denen ein höherer Wert für `n_estimators` immer besser ist, führt das Erhöhen von `n_estimators` beim Gradient Boosting zu einem komplexeren Modell und begünstigt damit Overfitting. Es ist gängige Praxis, den Wert für `n_estimators` der verfügbaren Zeit und dem Speicher anzupassen und dann unterschiedliche Werte für `learning_rate` zu durchsuchen.

Ein weiterer wichtiger Parameter ist `max_depth` (oder alternativ `max_leaf_nodes`), der die Komplexität jedes Baumes verringert. Normalerweise ist `max_depth` beim Gradient Boosting auf einen sehr niedrigen Wert gesetzt, meist nicht tiefer als fünf Verzweigungen.

## Support Vector Machines mit Kernel

Die nächste Art überwachter Modelle, die wir besprechen werden, sind Support Vector Machines mit Kernel. Wir haben lineare Support Vector Machines zur Klassifikation bereits in Abschnitt »Lineare Modelle zur Klassifizierung« auf Seite 55 ausprobiert. Support Vector Machines mit Kernel (oft einfach SVMs genannt) sind eine Erweiterung, bei der komplexere, nicht auf Hyperebenen im Eingaberaum basierende Modelle möglich sind. Obwohl Support Vector Machines für Klassifikation und Regression existieren, werden wir uns hier auf die als SVC implementierten Modelle konzentrieren.

tierte Variante zur Klassifikation beschränken. Bei der als SVR implementierten Support Vector Regression findet sich ein ähnliches Funktionsprinzip.

Die mathematische Grundlage von Support Vector Machines mit Kernel ist etwas aufwendig und würde den Rahmen dieses Buches sprengen. Sie finden die Details dazu in Kapitel 12 in »The Elements of Statistical Learning« (<http://statweb.stanford.edu/~tibs/ElemStatLearn/>) von Hastie, Tibshirani und Friedman. Wir werden dennoch versuchen, Ihnen ein Gefühl für die Idee zu vermitteln.

### Lineare Modelle und nichtlineare Merkmale

Wie Sie in Abbildung 2-15 sehen konnten, sind lineare Modelle in niedrig dimensionalen Räumen eher begrenzt, da Linien und Hyperebenen nur wenig flexibel sind. Ein lineares Modell lässt sich durch das Hinzufügen weiterer Merkmale flexibler machen, beispielsweise durch zusätzliche Interaktionen oder Polynome der Eingabemerkmale.

Betrachten wir dazu den in Abschnitt »Wichtigkeit von Merkmalen in Bäumen« auf Seite 74 verwendeten synthetischen Datensatz (siehe Abbildung 2-36):

In[74]:

```
X, y = make_blobs(centers=4, random_state=8)
y = y % 2

mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Merkmal 0")
plt.ylabel("Merkmal 1")
```

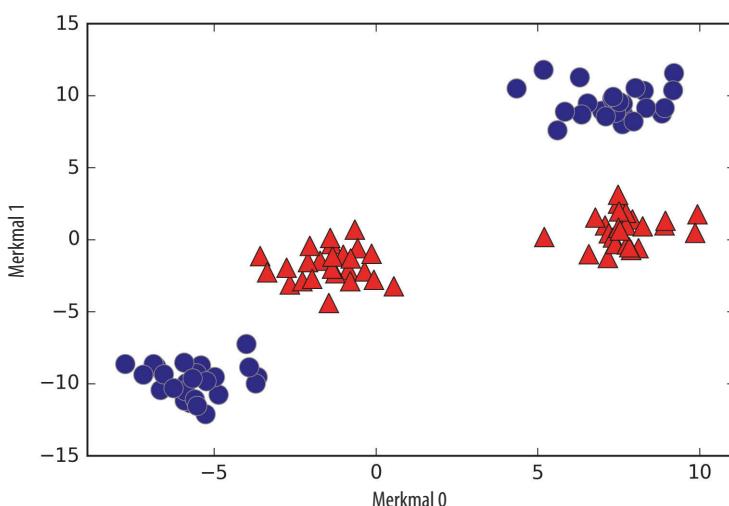


Abbildung 2-36: Klassifikationsdatensatz mit zwei Kategorien, die nichtlinear separierbar sind

Ein lineares Klassifikationsmodell kann Punkte nur über eine Linie voneinander trennen, und bei diesem Datensatz wird es nicht sehr gut abschneiden (siehe Abbildung 2-37):

In[75]:

```
from sklearn.svm import LinearSVC
linear_svm = LinearSVC().fit(X, y)

mglearn.plots.plot_2d_separator(linear_svm, X)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Merkmals 0")
plt.ylabel("Merkmals 1")
```

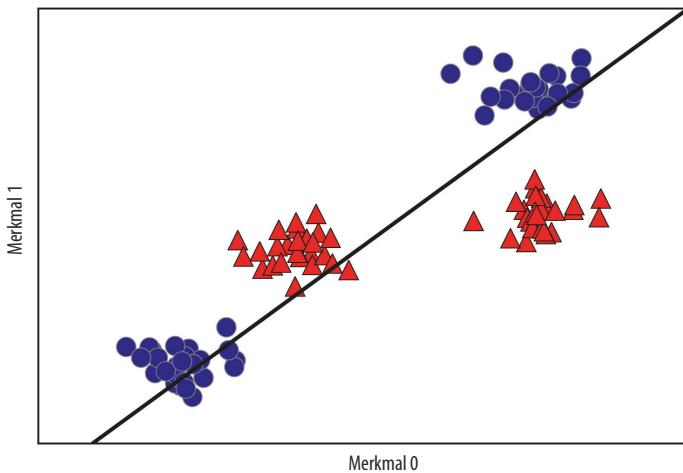


Abbildung 2-37: Von einem linearen SVM gefundene Entscheidungsgrenze

Erweitern wir nun die Menge der Merkmale, indem wir auch `merkmals1 ** 2`, das Quadrat des zweiten Merkmals, als zusätzliches Merkmal hinzufügen. Anstatt jeden Datenpunkt als zweidimensionalen Punkt (`merkmals0, merkmals1`) anzugeben, repräsentieren wir diese nun als dreidimensionalen Punkt, (`merkmals0, merkmals1, merkmals1 ** 2`).<sup>10</sup> Diese neue Repräsentation ist in Abbildung 2-38 als dreidimensionales Streudiagramm dargestellt:

In[76]:

```
# füge das quadrierte zweite Merkmal hinzu
X_new = np.hstack([X, X[:, 1:] ** 2])

from mpl_toolkits.mplot3d import Axes3D, axes3d
figure = plt.figure()
# visualisiere in 3-D
ax = Axes3D(figure, elev=-152, azim=-26)
```

<sup>10</sup> Wir haben dieses Merkmal nur zur Veranschaulichung ausgewählt. Die Auswahl selbst ist in keiner Weise wichtig.

```
# plotte zuerst alle Punkte mit y == 0, dann alle mit y == 1
mask = y == 0
ax.scatter(X_new[mask, 0], X_new[mask, 1], X_new[mask, 2], c='b',
           cmap=mglearn.cm2, s=60)
ax.scatter(X_new[~mask, 0], X_new[~mask, 1], X_new[~mask, 2], c='r', marker='^',
           cmap=mglearn.cm2, s=60)
ax.set_xlabel("merkmal0")
ax.set_ylabel("merkmal1")
ax.set_zlabel("merkmal1 ** 2")
```

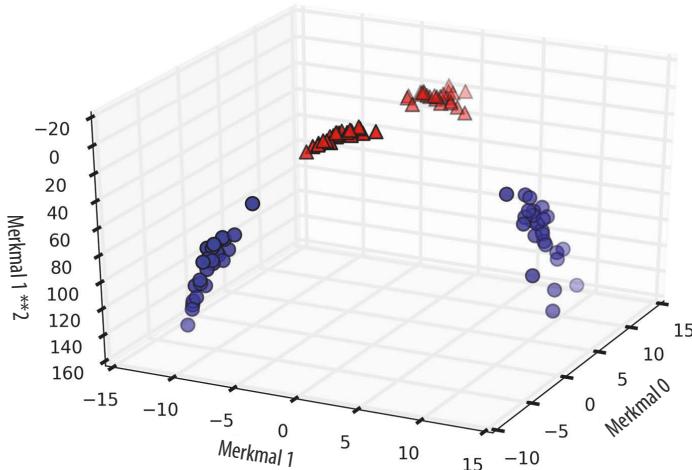


Abbildung 2-38: Erweiterung des Datensatzes aus Abbildung 2-37 durch ein zusätzliches von merkmal1 abgeleitetes drittes Merkmal

In der neuen Repräsentation der Daten ist es nun tatsächlich möglich, die beiden Kategorien über ein lineares Modell zu trennen, einer Ebene in drei Dimensionen. Wir können dies durch Anpassen eines linearen Modells an die erweiterten Daten nachweisen (siehe Abbildung 2-39):

In[77]:

```
linear_svm_3d = LinearSVC().fit(X_new, y)
coef, intercept = linear_svm_3d.coef_.ravel(), linear_svm_3d.intercept_

# stelle die lineare Entscheidungsgrenze dar
figure = plt.figure()
ax = Axes3D(figure, elev=-152, azim=-26)
xx = np.linspace(X_new[:, 0].min() - 2, X_new[:, 0].max() + 2, 50)
yy = np.linspace(X_new[:, 1].min() - 2, X_new[:, 1].max() + 2, 50)

XX, YY = np.meshgrid(xx, yy)
ZZ = (coef[0] * XX + coef[1] * YY + intercept) / -coef[2]
ax.plot_surface(XX, YY, ZZ, rstride=8, cstride=8, alpha=0.3)
ax.scatter(X_new[mask, 0], X_new[mask, 1], X_new[mask, 2], c='b',
           cmap=mglearn.cm2, s=60)
ax.scatter(X_new[~mask, 0], X_new[~mask, 1], X_new[~mask, 2], c='r', marker='^',
           cmap=mglearn.cm2, s=60)
```

```

ax.set_xlabel("merkmal0")
ax.set_ylabel("merkmal1")
ax.set_zlabel("merkmal0 ** 2")

```

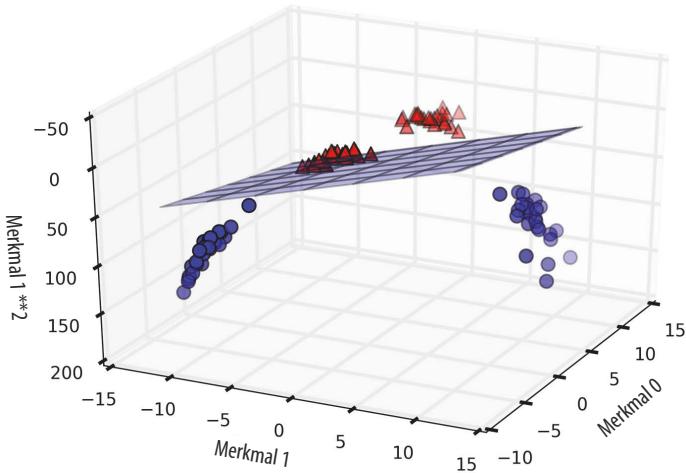


Abbildung 2-39: Von einem linearen SVM auf dem erweiterten dreidimensionalen Datensatz ermittelte Entscheidungsgrenze

Als Funktion der ursprünglichen Merkmale ist das lineare SVM-Modell streng genommen nicht mehr linear. Es ist keine Linie, sondern eher eine Ellipse, wie Sie im folgenden Diagramm sehen (Abbildung 2-40):

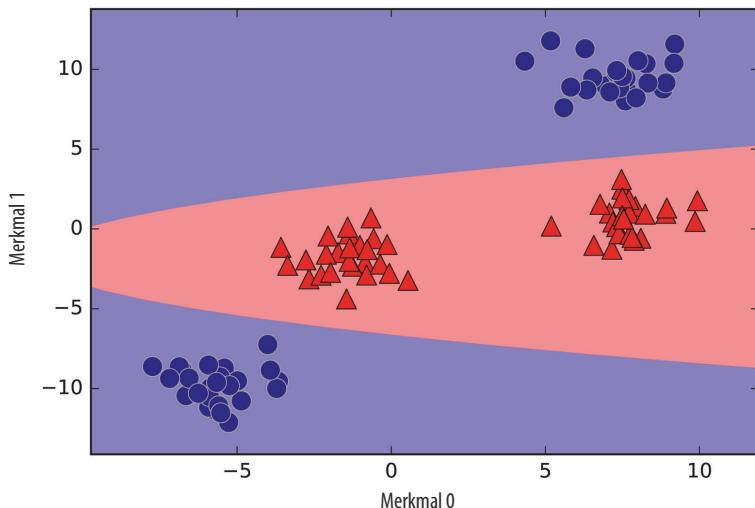


Abbildung 2-40: Die Entscheidungsgrenze aus Abbildung 2-39 als Funktion der ursprünglichen zwei Merkmale

In[78]:

```
ZZ = YY ** 2
dec = linear_svm_3d.decision_function(np.c_[XX.ravel(), YY.ravel(), ZZ.ravel()])
plt.contourf(XX, YY, dec.reshape(XX.shape), levels=[dec.min(), 0, dec.max()],
             cmap=mglearn.cm2, alpha=0.5)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Merkmakl 0")
plt.ylabel("Merkmakl 1")
```

## Der Kernel-Trick

Wir können daraus schlussfolgern, dass das Hinzufügen nichtlinearer Merkmale zur Repräsentation unserer Daten lineare Modelle sehr viel mächtiger macht. Allerdings weiß man meist nicht, welche Merkmale man hinzufügen soll. Einfach sehr viele Merkmale hinzuzufügen (z. B. alle möglichen Interaktionen in einem 100-dimensionalen Merkmalsraum), bedeutet einen sehr hohen Rechenaufwand. Glücklicherweise gibt es einen mathematischen Trick, mit dem wir einen Klassifikator in einem höher dimensionalen Raum trainieren können, ohne die neuen, möglicherweise sehr große Repräsentation explizit zu berechnen. Dies nennt man den *Kernel-Trick*, bei dem man die Entfernung der Datenpunkte (genauer, deren Skalarprodukte) im erweiterten Merkmalsraum berechnen kann, ohne diese Erweiterung explizit zu berechnen.

Es gibt zwei gebräuchliche Methoden, Daten für eine Support Vector Machine in einen höher dimensionalen Raum zu projizieren: der polynomische Kernel, der sämtliche möglichen Polynome der ursprünglichen Merkmale bis zu einem bestimmten Grad berechnet (z. B. `merkmakl1 ** 2 * merkmakl2 ** 5`), und der Kernel mit radialer Basisfunktion (RBF), der auch als Gaußscher Kernel bezeichnet wird. Der Gaußsche Kernel ist etwas schwieriger zu erklären, da er sich auf einen Merkmalsraum mit unendlich vielen Dimensionen bezieht. Eine Erklärungsmöglichkeit ist, dass dieser alle möglichen Polynome aller Grade berücksichtigt, aber die Wichtigkeit der Merkmale mit steigendem Grad abnimmt.<sup>11</sup>

In der Praxis sind die mathematischen Details der Kernel-SVMs jedoch gar nicht so entscheidend. Im nächsten Abschnitt sehen wir eine einfache Zusammenfassung dessen, wie eine SVM mit RBF-Kernel Entscheidungen trifft.

## SVMs verstehen

Beim Training lernt eine SVM, wie wichtig jeder der Trainingsdatenpunkte für die Repräsentation der Entscheidungsgrenze zwischen den zwei Kategorien ist. Typischerweise ist nur ein Teil der Trainingsdaten zum Definieren der Entscheidungsgrenze entscheidend: diejenigen, die an der Grenze zwischen den Kategorien liegen. Diese nennt man *Support-Vektoren*, und Support Vector Machines sind nach ihnen benannt.

---

<sup>11</sup> Dies ergibt sich aus der Schreibweise einer Exponentialfunktion als Taylor-Reihe.

Um für einen Punkt eine Vorhersage vorzunehmen, bestimmt man die Entfernung zu jedem der Support-Vektoren. Aufgrund der Entfernungen und der beim Training ermittelten Wichtigkeiten der Support-Vektoren (im Attribut `dual_coef_` in SVC) wird eine Entscheidung getroffen.

Die Entfernung zwischen den Datenpunkten wird vom Gaußschen Kernel folgendermaßen berechnet:

$$k_{\text{rbf}}(x_1, x_2) = \exp(-\gamma \|x_1 - x_2\|^2)$$

Dabei sind  $x_1$  und  $x_2$  Datenpunkte,  $\|x_1 - x_2\|$  bezeichnet den euklidischen Abstand, und  $\gamma$  (gamma) ist ein Parameter zum Kontrollieren der Breite des Gaußschen Kernels.

Abbildung 2-41 zeigt die Ergebnisse des Trainings einer Support Vector Machine auf einem zweidimensionalen Datensatz mit zwei Kategorien. Die Entscheidungsgrenze ist schwarz dargestellt, und die Support-Vektoren sind die größeren Punkte mit breitem Rand. Der folgende Code erstellt dieses Diagramm, indem er eine SVM auf dem Datensatz `forge` trainiert:

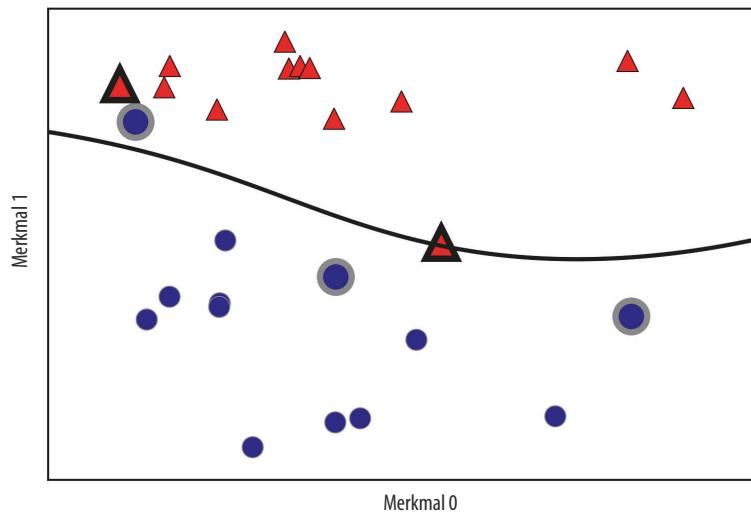


Abbildung 2-41: Von einer SVM mit RBF-Kernel ermittelte Entscheidungsgrenze und Support-Vektoren

In[79]:

```
from sklearn.svm import SVC
X, y = mglearn.tools.make_handcrafted_dataset()
svm = SVC(kernel='rbf', C=10, gamma=0.1).fit(X, y)
mglearn.plots.plot_2d_separator(svm, X, eps=.5)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
# zeichne die Support-Vektoren
sv = svm.support_vectors_
# Kategoriebezeichner der Support-Vektoren sind durch das Vorzeichen
# der dualen Koeffizienten angegeben
sv_labels = svm.dual_coef_.ravel() > 0
```

```
mglearn.discrete_scatter(sv[:, 0], sv[:, 1], sv_labels, s=15, markeredgewidth=3)
plt.xlabel("Merkmals 0")
plt.ylabel("Merkmals 1")
```

In diesem Fall ermittelt die SVM eine sehr weiche, nichtlineare Grenze (keine gerade Linie). Wir haben hier zwei Parameter angegeben: den Parameter  $C$  und den Parameter  $\gamma$ , die wir als Nächstes besprechen werden.

## Optimieren von SVM-Parametern

Der Parameter  $\gamma$  ist der in der Formel im vorigen Abschnitt angegebene, der die Breite des Gaußschen Kernels festlegt. Er bestimmt die Größenordnung, bei der Punkte als nahe beieinanderliegend gelten. Der Parameter  $C$  ist ein Regularisierungsparameter, ähnlich denen in linearen Modellen. Er begrenzt die Wichtigkeit jedes Punktes (genauer, ihres Wertes in `dual_coef_`).

Sehen wir uns an, wie sich Änderungen dieser Parameter auswirken (Abbildung 2-42):

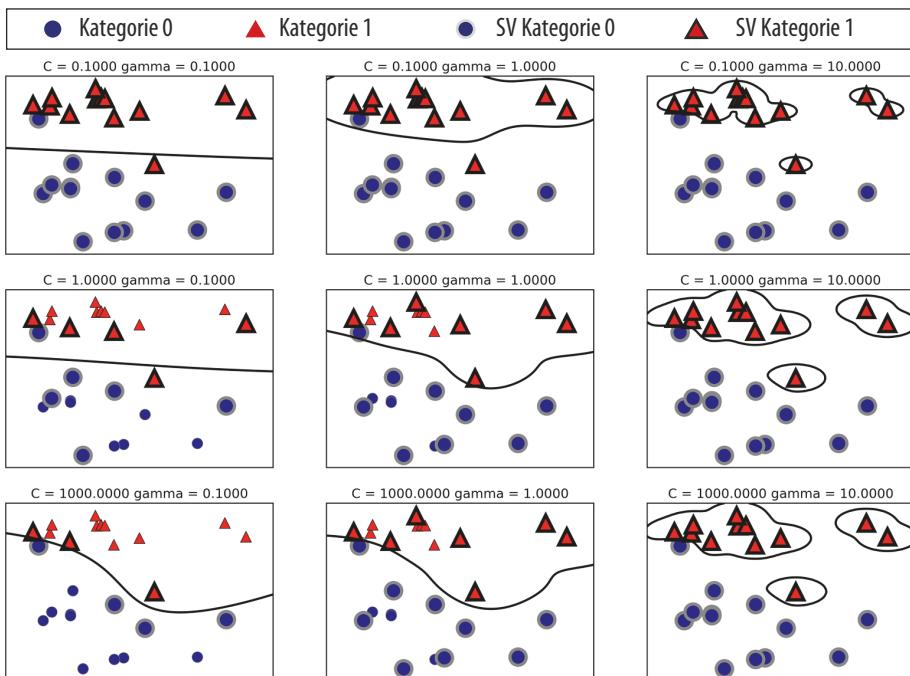


Abbildung 2-42: Entscheidungsgrenzen und Support-Vektoren bei unterschiedlich eingestellten Werten für die Parameter  $C$  und  $\gamma$

In[80]:

```
fig, axes = plt.subplots(3, 3, figsize=(15, 10))

for ax, C in zip(axes, [-1, 0, 3]):
    for a, gamma in zip(ax, range(-1, 2)):
        mglearn.plots.plot_svm(log_C=C, log_gamma=gamma, ax=a)
```

```
axes[0, 0].legend(["Kategorie 0", "Kategorie 1", "sv Kategorie 0", "sv Kategorie 1"],  
                  ncol=4, loc=(.9, 1.2))
```

Von links nach rechts erhöhen wir den Wert des Parameters `gamma` von 0.1 bis 10. Bei kleinem `gamma` ist der Radius des Gaußschen Kernels groß, wodurch viele Punkte als nahe gelegen betrachtet werden. Dies führt zu sehr weichen Entscheidungsgrenzen auf der linken Seite und Grenzen, die sich mehr auf einzelne Punkte konzentrieren, auf der rechten. Ein niedriger Wert für `gamma` führt zu einer langsam variierten Entscheidungsgrenze, wodurch ein wenig komplexes Modell entsteht. Ein hoher Wert für `gamma` ergibt dagegen ein komplexeres Modell.

Wir erhöhen den Parameter `C` von 0.1 oben bis auf 1000 unten. Wie bei linearen Modellen steht ein kleiner Wert für `C` für ein sehr restriktives Modell, bei dem jeder Datenpunkt nur einen sehr begrenzten Einfluss hat. Sie können sehen, dass die Entscheidungsgrenze oben links beinahe linear ist und die falsch zugeordneten Punkte kaum einen Einfluss auf die Linie ausüben. Erhöhen von `C` verstärkt den Einfluss dieser Punkte auf das Modell und biegt die Entscheidungsgrenze so, dass die Punkte korrekt klassifiziert werden. Dies ist unten links zu sehen.

Wenden wir nun eine SVM mit RBF-Kernel auf den Brustkrebs-Datensatz an. Die Standardeinstellungen sind `C=1` und `gamma=1/n_merkmale`:

**In[81]:**

```
X_train, X_test, y_train, y_test = train_test_split(  
    cancer.data, cancer.target, random_state=0)  
  
svc = SVC()  
svc.fit(X_train, y_train)  
  
print("Genauigkeit auf den Trainingsdaten: {:.2f}".format(svc.score(X_train, y_train)))  
print("Genauigkeit auf den Testdaten: {:.2f}".format(svc.score(X_test, y_test)))
```

**Out[81]:**

```
Genauigkeit auf den Trainingsdaten: 1.00  
Genauigkeit auf den Testdaten: 0.63
```

Im Modell liegt ein beträchtliches Overfitting vor, da die Genauigkeit auf den Trainingsdaten perfekt ist, auf den Testdaten aber nur 63 % beträgt. Auch wenn SVMs oft eine recht gute Leistung erbringen, sind sie sehr empfindlich gegenüber den Parametereinstellungen und der Skalierung der Daten. Insbesondere setzen sie voraus, dass sich sämtliche Merkmale auf einer ähnlichen Skala befinden. Betrachten wir die minimalen und maximalen Werte jedes Merkmals auf einer logarithmischen Skala (Abbildung 2-43):

**In[82]:**

```
plt.plot(X_train.min(axis=0), 'o', label="min")  
plt.plot(X_train.max(axis=0), '^', label="max")  
plt.legend(loc=4)  
plt.xlabel("Index des Merkmals")  
plt.ylabel("Größe des Merkmals")  
plt.yscale("log")
```

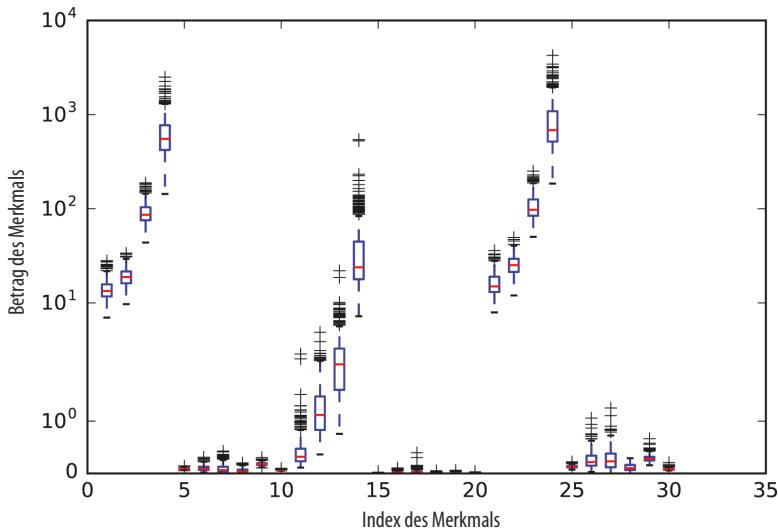


Abbildung 2-43: Größenordnung der Merkmale im Brustkrebs-Datensatz (die y-Achse hat eine logarithmische Skala)

Aus diesem Diagramm können wir schließen, dass die Merkmale im Brustkrebs-Datensatz völlig unterschiedliche Größenordnungen haben. Die ist für andere Modelle ein wenig problematisch (z. B. bei linearen Modellen), aber bei Kernel-SVMs hat es katastrophale Auswirkungen. Untersuchen wir einige Möglichkeiten, mit diesem Thema umzugehen.

### Vorverarbeitung von Daten für SVMs

Eine Möglichkeit zur Lösung dieses Problems ist, jedes Merkmal so umzuskalieren, dass sich alle auf der etwa gleichen Skala bewegen. Eine übliche Methode zum Umskalieren bei Kernel-SVMs ist, die Daten so zu skalieren, dass alle Merkmale zwischen 0 und 1 liegen. Wir werden dies mit der Methode `MinMaxScaler` in Kapitel 3 tun, wo wir hierzu weitere Details erfahren. Diesmal führen wir die Skalierung »von Hand« durch:

**In[83]:**

```
# berechne den kleinsten Wert pro Merkmal aus den Trainingsdaten
min_on_training = X_train.min(axis=0)
# berechne die Spannbreite (max - min) jedes Merkmals aus den Trainingsdaten
range_on_training = (X_train - min_on_training).max(axis=0)

# ziehe min ab und teile durch die Spannbreite
# anschließend ist bei jedem Merkmal min=0 und max=1
X_train_scaled = (X_train - min_on_training) / range_on_training
print("Minima aller Merkmale:\n{}".format(X_train_scaled.min(axis=0)))
print("Maxima aller Merkmale:\n{}".format(X_train_scaled.max(axis=0)))
```

**Out[83]:**

```
Minima aller Merkmale:  
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  
 0.  0.  0.  0.  0.  0.  0.  0.  0.  0. ]  
Maxima aller Merkmale:  
[ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  
 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

**In[84]:**

```
# verwende DIE GLEICHE Transformation auf den Testdaten  
# mit min und Spannbreite der Trainingsdaten (Details in Kapitel 3)  
X_test_scaled = (X_test - min_on_training) / range_on_training
```

**In[85]:**

```
svc = SVC()  
svc.fit(X_train_scaled, y_train)  
  
print("Genauigkeit auf den Trainingsdaten: {:.3f}".format(  
    svc.score(X_train_scaled, y_train)))  
print("Genauigkeit auf den Testdaten: {:.3f}".format(svc.score(X_test_scaled, y_test)))
```

**Out[85]:**

```
Genauigkeit auf den Trainingsdaten: 0.948  
Genauigkeit auf den Testdaten: 0.951
```

Das Skalieren der Daten macht einen entscheidenden Unterschied! Nun sind wir schon dabei zu unterfitten, wobei die Genauigkeiten auf Trainings- und Testdaten ähnlich zueinander, aber weniger nah an 100 % sind. Wir können nun entweder C oder gamma erhöhen, um ein etwas komplexeres Modell anzupassen. Beispielsweise:

**In[86]:**

```
svc = SVC(C=1000)  
svc.fit(X_train_scaled, y_train)  
  
print("Genauigkeit auf den Trainingsdaten: {:.3f}".format(  
    svc.score(X_train_scaled, y_train)))  
print("Genauigkeit auf den Testdaten: {:.3f}".format(svc.score(X_test_scaled, y_test)))
```

**Out[86]:**

```
Genauigkeit auf den Trainingsdaten: 0.988  
Genauigkeit auf den Testdaten: 0.972
```

Durch das Erhöhen von C verbessert sich die Genauigkeit des Modells deutlich auf 97.2 %.

## Stärken, Schwächen und Parameter

Support Vector Machines mit Kernel (SVMs) sind mächtige Modelle, die auf einer Vielzahl von Datensätzen gut funktionieren. SVMs können komplexe Entscheidungsgrenzen abbilden, auch wenn die Daten nur wenige Merkmale beinhalten. Sie funktionieren mit niedrig dimensionalen und hoch dimensionalen Daten (weni-

gen und vielen Merkmalen) gut, skalieren aber nicht sehr gut mit der Anzahl der Datenpunkte. Eine SVM mit bis zu 10000 Datenpunkten kann gut funktionieren, aber mit 100000 Punkten und mehr zu arbeiten, kann die Laufzeit und den Speicher übermäßig beanspruchen.

Ein weiterer Nachteil von SVMs ist, dass sie eine umsichtige Vorverarbeitung der Daten und Einstellung der Parameter erfordern. Deshalb werden heutzutage bei vielen Aufgabenstellungen meist auf Bäumen basierende Modelle wie Random Forests oder Gradient Boosting bevorzugt (welche wenig oder keine Vorverarbeitung erfordern). Außerdem sind SVM-Modelle schwer zu untersuchen; es kann schwierig sein, zu verstehen, warum eine bestimmte Vorhersage vorgenommen wurde, oder das Modell einem Laien zu erklären.

Dennoch kann sich das Ausprobieren von SVMs lohnen, besonders wenn alle Ihre Merkmale Messungen in einer ähnlichen Einheit (z. B. Intensitäten von Pixeln) in einer ähnlichen Größenordnung sind.

Die wichtigen Parameter bei Kernel-SVMs sind der Regularisierungsparameter  $C$ , die Auswahl des Kernels und die für den Kernel spezifischen Parameter. Auch wenn wir uns vor allem mit dem RBF-Kernel beschäftigt haben, bietet uns scikit-learn noch weitere Auswahlmöglichkeiten. Der RBF-Kernel besitzt nur einen Parameter,  $\gamma$ , der die inverse Breite des Gaußschen Kernels enthält. Sowohl  $\gamma$  als auch  $C$  beeinflussen die Komplexität des Modells, wobei große Werte beider Parameter zu einem komplexeren Modell führen. Daher korrelieren gute Einstellungen beider Parameter für gewöhnlich miteinander. Man sollte  $C$  und  $\gamma$  gemeinsam optimieren.

## Neuronale Netze (Deep Learning)

Eine als neuronale Netze bekannte Familie von Algorithmen hat in jüngerer Zeit ein Comeback unter dem Namen »Deep Learning« erfahren. Deep Learning ist für viele Anwendungen maschinellen Lernens sehr vielversprechend. Deep Learning-Algorithmen sind oft sehr genau auf einen bestimmten Anwendungsfall zugeschnitten. Hier werden wir nur eine relativ einfache Methode namens *mehrschichtige Perzeptrons* zur Klassifikation und Regression besprechen. Diese soll als Ausgangspunkt für weitere Deep Learning-Methoden dienen. Mehrschichtige Perzeptrons (MLPs) sind auch als (vanilla) Feed-Forward-Netze bekannt oder einfach als neuronale Netze.

### Das Modell bei neuronalen Netzen

MLPs lassen sich als Verallgemeinerung von linearen Modellen ansehen, die in mehreren Verarbeitungsschritten zu einer Entscheidung gelangen. Erinnern wir uns daran, dass die Vorhersage durch einen linearen Regressor durch folgende Formel gegeben ist:

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \dots + w[p] * x[p] + b$$

Zu Deutsch ist  $\hat{y}$  eine gewichtete Summe der Eingabemerkmale  $x[0]$  bis  $x[p]$ , mit den Koeffizienten  $w[0]$  bis  $w[p]$  als Gewichte. Dies lässt sich grafisch wie in Abbildung 2-44 darstellen:

**In[87]:**

```
display(mglearn.plots.plot_logistic_regression_graph())
```

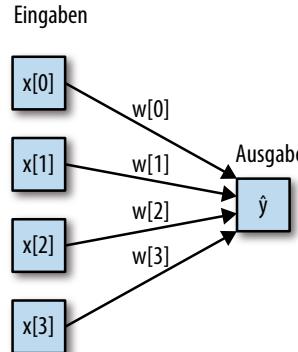


Abbildung 2-44: Visualisierung einer logistischen Regression, wobei die Eingabemerkmale und Vorhersagen als Knoten und die Koeffizienten als Verbindungen dieser Knoten dargestellt sind

Hierbei steht jeder Knoten auf der linken Seite für ein Merkmal der Eingabe, die Verbindungslienien stehen für die erlernten Koeffizienten, und der Knoten auf der rechten Seite steht für die Ausgabe als gewichtete Summe der Eingabewerte.

Bei einem MLP wird dieses Berechnen einer gewichteten Summe mehrfach wiederholt. Zuerst werden *verborgene Einheiten* berechnet, die einen Zwischenschritt darstellen. Diese wiederum werden durch eine gewichtete Summe miteinander kombiniert, um das Endergebnis zu erhalten (Abbildung 2-45):

**In[88]:**

```
display(mglearn.plots.plot_single_hidden_layer_graph())
```

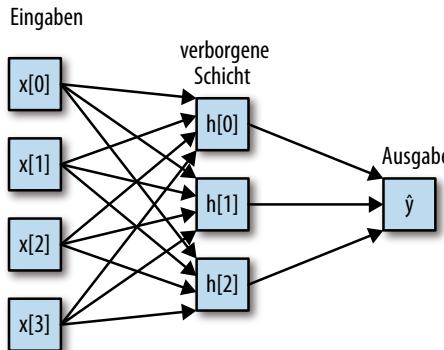


Abbildung 2-45: Darstellung eines mehrschichtigen Perzeptrons mit einer verborgenen Schicht

Bei diesem Modell gibt es wesentlich mehr Koeffizienten (Gewichte) zu erlernen: Es gibt einen zwischen jedem Eingabewert und jeder verborgenen Einheit (aus denen die *verborgene Schicht* besteht) sowie einen zwischen jeder Einheit der verborgenen Schicht und der Ausgabe.

Eine Serie gewichteter Summen zu berechnen, ist mathematisch das Gleiche wie die Berechnung genau einer gewichteten Summe. Um dieses Modell tatsächlich leistungsfähiger als ein lineares Modell zu machen, benötigen wir einen Trick. Nach dem Berechnen der gewichteten Summe für jede verborgene Einheit wendet man eine nichtlineare Funktion auf das Ergebnis an – normalerweise die *rectifying non-linearity* (auch *rectified linear unit* oder *relu* genannt) oder einen *Tangens hyperbolicus* (*tanh*). Das Ergebnis dieser Funktion wird dann in der gewichteten Summe zur Berechnung der Ausgabe  $\hat{y}$  verwendet. Die beiden Funktionen sind in Abbildung 2-46 dargestellt. Die *relu* schneidet Werte unter null ab, während *tanh* sich bei niedrigen Eingabewerten  $-1$  annähert, bei hohen Eingabewerten dagegen  $+1$ . Beide nichtlineare Funktionen ermöglichen es den neuronalen Netzen, wesentlich kompliziertere Funktionen zu erlernen, als es ein lineares Modell könnte:

**In[89]:**

```
line = np.linspace(-3, 3, 100)
plt.plot(line, np.tanh(line), label="tanh")
plt.plot(line, np.maximum(line, 0), label="relu")
plt.legend(loc="best")
plt.xlabel("x")
plt.ylabel("relu(x), tanh(x)")
```

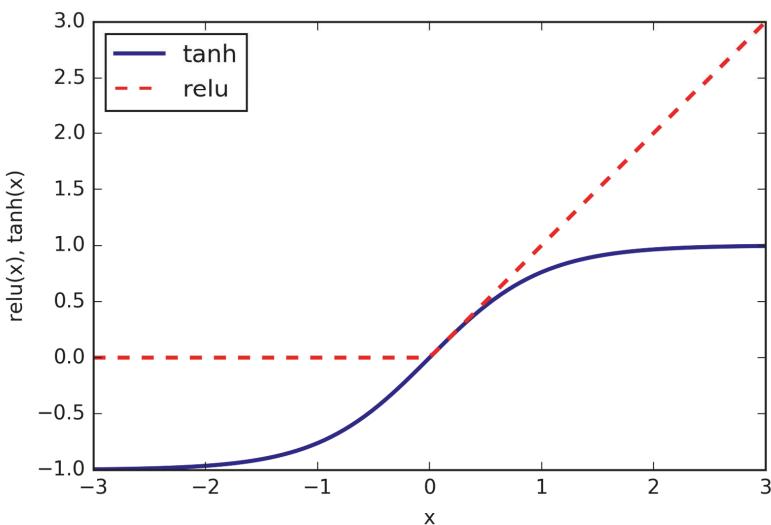


Abbildung 2-46: Die Aktivierungsfunktionen Tangens hyperbolicus und rectified linear

Bei dem kleinen neuronalen Netz in Abbildung 2-45 wäre die vollständige Formel zum Berechnen von  $\hat{y}$  im Falle einer Regression (mit tanh als nichtlinearer Funktion):

$$\begin{aligned} h[0] &= \tanh(w[0, 0] * x[0] + w[1, 0] * x[1] + w[2, 0] * x[2] + w[3, 0] * x[3] + b[0]) \\ h[1] &= \tanh(w[0, 0] * x[0] + w[1, 0] * x[1] + w[2, 0] * x[2] + w[3, 0] * x[3] + b[1]) \\ h[2] &= \tanh(w[0, 0] * x[0] + w[1, 0] * x[1] + w[2, 0] * x[2] + w[3, 0] * x[3] + b[2]) \\ \hat{y} &= v[0] * h[0] + v[1] * h[1] + v[2] * h[2] + b \end{aligned}$$

Dabei sind  $w$  die Gewichte zwischen den Eingabewerten  $x$  und der verborgenen Schicht  $h$ , und  $v$  sind die Gewichte zwischen der verborgenen Schicht  $h$  und der Ausgabe  $\hat{y}$ . Die Gewichte  $v$  und  $w$  lassen sich anhand der Daten trainieren,  $x$  sind die Merkmale der Eingabe,  $\hat{y}$  ist die berechnete Ausgabe, und  $h$  sind Zwischenergebnisse. Ein wichtiger Parameter, den der Nutzer einstellen muss, ist die Anzahl der Knoten in der verborgenen Schicht. Diese Zahl kann für sehr kleine oder einfache Datensätze 10 betragen und für sehr komplexe Daten 10000 erreichen. Man kann auch wie in Abbildung 2-47 zusätzliche verborgene Schichten hinzufügen:

**In[90]:**

```
mglearn.plots.plot_two_hidden_layer_graph()
```

Große neuronale Netze mit vielen dieser Berechnungsschichten haben zum Namen »Deep Learning« inspiriert.

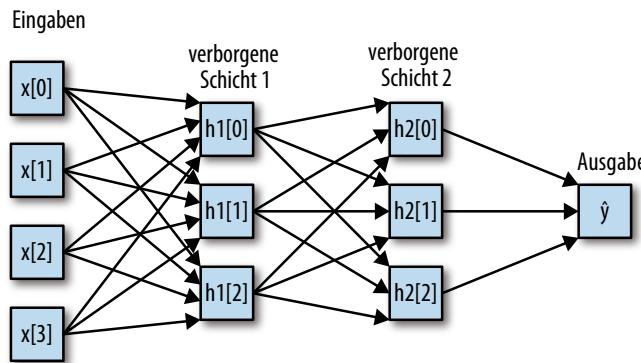


Abbildung 2-47: Ein mehrschichtiges Perzeptron mit zwei verborgenen Schichten

### Feineinstellung neuronaler Netze

Betrachten wir ein MLP in Aktion, indem wir den `MLPClassifier` auf den in diesem Kapitel bereits verwendeten Datensatz `two_moons` anwenden. Das Ergebnis sehen Sie in Abbildung 2-48:

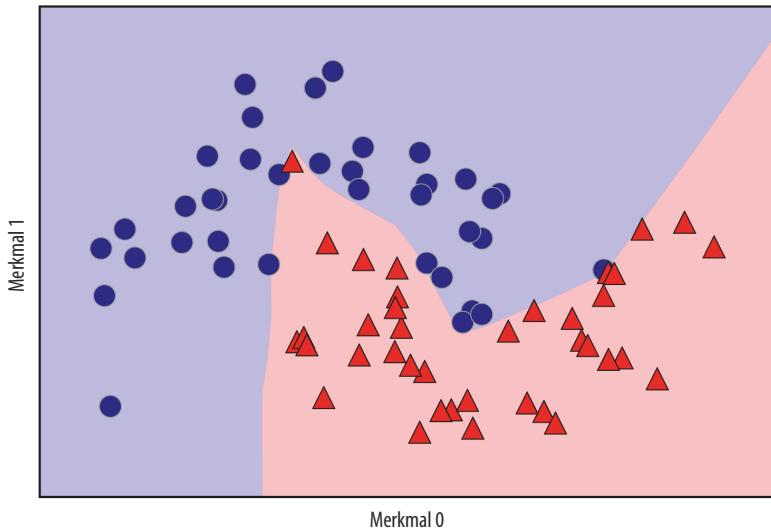
**In[91]:**

```
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import make_moons
```

```
X, y = make_moons(n_samples=100, noise=0.25, random_state=3)

X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
                                                random_state=42)

mlp = MLPClassifier(solver='lbfgs', random_state=0).fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Merkmal 0")
plt.ylabel("Merkmal 1")
```



*Abbildung 2-48: Von einem neuronalen Netz mit 100 verborgenen Einheiten auf dem Datensatz two\_moons erlernte Entscheidungsgrenzen*

Wie Sie sehen, hat das neuronale Netz eine stark nichtlineare, aber relativ weiche Entscheidungsgrenze erlernt. Wir haben die Option `solver='lbfgs'` verwendet, die wir später besprechen werden.

Standardmäßig verwendet das MLP 100 verborgene Knoten, für diesen kleinen Datensatz eine Menge. Wir können diese Zahl senken (und damit die Komplexität des Modells senken) und dennoch ein gutes Ergebnis erhalten (Abbildung 2-49):

**In[92]:**

```
mlp = MLPClassifier(solver='lbfgs', random_state=0, hidden_layer_sizes=[10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Merkmal 0")
plt.ylabel("Merkmal 1")
```

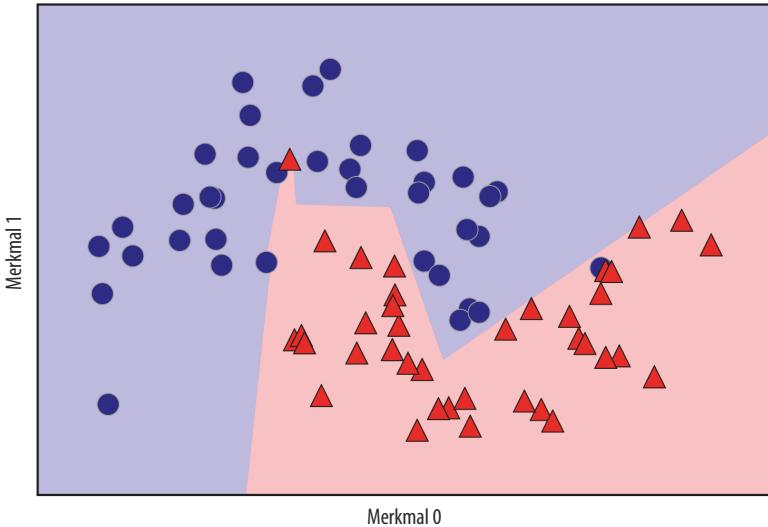


Abbildung 2-49: Von einem neuronalen Netz mit 10 verborgenen Einheiten auf dem Datensatz `two_moons` erlernte Entscheidungsgrenzen

Mit nur 10 verborgenen Einheiten wirkt die Entscheidungsgrenze deutlich zerzauster. Standardmäßig wird `relu` für die Nichtlinearität verwendet, wie in Abbildung 2-46 zu sehen ist. Mit einer verborgenen Ebene besteht die Entscheidungsfunktion aus 10 linearen Segmenten. Wenn wir eine weichere Entscheidungsgrenze erhalten möchten, könnten wir mehr verborgene Einheiten (wie in Abbildung 2-48) hinzufügen, eine zweite verborgene Ebene hinzufügen (Abbildung 2-50) oder `tanh` für die Nichtlinearität verwenden (Abbildung 2-51):

**In[93]:**

```
# verwenden zweier verborgener Ebenen mit je 10 Einheiten
mlp = MLPClassifier(solver='lbfgs', random_state=0,
                     hidden_layer_sizes=[10, 10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Merkmals 0")
plt.ylabel("Merkmals 1")
```

**In[94]:**

```
# Verwenden von zwei verborgenen Ebenen mit je 10 Einheiten
# und tanh für die Nichtlinearität
mlp = MLPClassifier(solver='lbfgs', activation='tanh',
                     random_state=0, hidden_layer_sizes=[10, 10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Merkmals 0")
plt.ylabel("Merkmals 1")
```

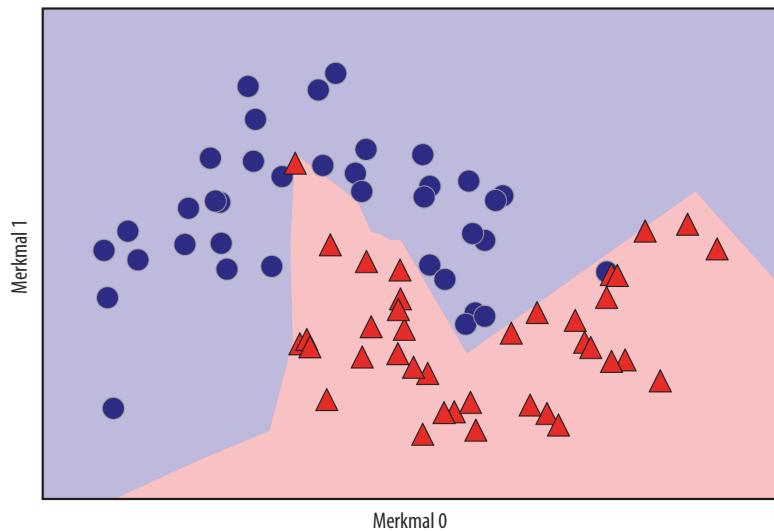


Abbildung 2-50: Mit zwei verborgenen Schichten mit je 10 verborgenen Einheiten erlernte Entscheidungsgrenze mit rect als Aktivierungsfunktion

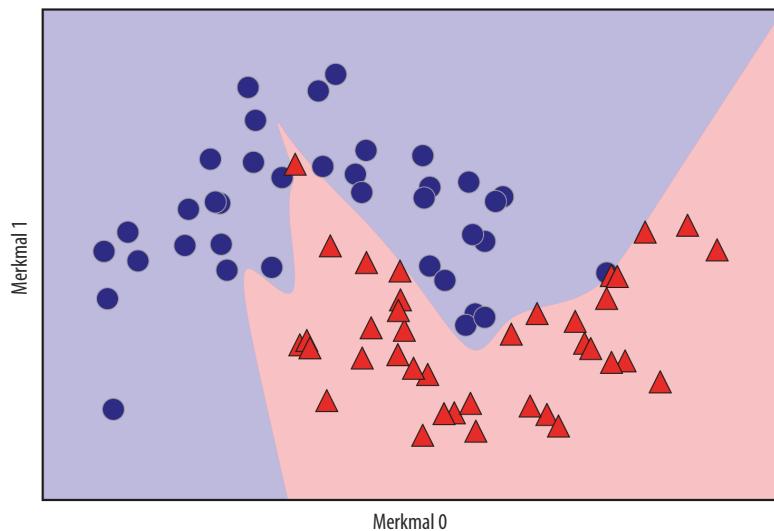


Abbildung 2-51: Mit zwei verborgenen Schichten mit je 10 verborgenen Einheiten erlernte Entscheidungsgrenze mit tanh als Aktivierungsfunktion

Schließlich können wir die Komplexität eines neuronalen Netzes auch kontrollieren, indem wir wie bei der Ridge-Regression und den linearen Klassifikatoren die Gewichte über einen L2-Strafterm in Richtung null drängen. Der Parameter beim MLPClassifier dazu heißt  $\alpha$  (wie bei den linearen Regressionsmodellen). Standardmäßig ist dieser auf einen sehr niedrigen Wert gesetzt (wenig Regularisie-

rung). In Abbildung 2-52 sind die Auswirkungen unterschiedlicher Werte für alpha auf den Datensatz two\_moons mit zwei verborgenen Schichten mit je 10 oder 100 Einheiten dargestellt:

In[95]:

```
fig, axes = plt.subplots(2, 4, figsize=(20, 8))
for axx, n_hidden_nodes in zip(axes, [10, 100]):
    for ax, alpha in zip(axx, [0.0001, 0.01, 0.1, 1]):
        mlp = MLPClassifier(solver='lbfgs', random_state=0,
                             hidden_layer_sizes=[n_hidden_nodes, n_hidden_nodes],
                             alpha=alpha)
        mlp.fit(X_train, y_train)
        mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3, ax=ax)
        mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train, ax=ax)
        ax.set_title("n_hidden=[{}, {}]\nalpha={:.4f}".format(
            n_hidden_nodes, n_hidden_nodes, alpha))
```

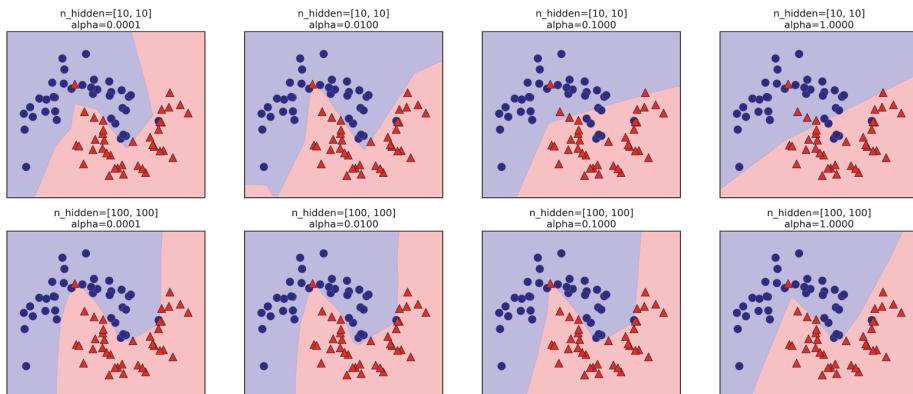


Abbildung 2-52: Entscheidungsfunktionen mit unterschiedlichen Anzahlen verborgener Einheiten und unterschiedlichen Werten für den Parameter alpha

Wie Sie inzwischen vermutlich bemerkt haben, gibt es mehrere Möglichkeiten, die Komplexität eines neuronalen Netzes zu beeinflussen: die Anzahl verborgener Schichten, die Anzahl Einheiten pro Schicht und die Regularisierung (alpha). Es gibt noch weitere, auf die wir hier nicht eingehen.

Eine wichtige Eigenschaft neuronaler Netze ist, dass ihre Gewichte vor dem Lernprozess zufällig gesetzt werden, und diese zufällige Initialisierung beeinflusst das trainierte Modell. Das bedeutet, dass wir auch mit identischen Parametern, aber anderem Random Seed sehr unterschiedliche Modelle erhalten können. Wenn die Netze groß sind und ihre Komplexität angemessen eingestellt ist, sollte das die Genauigkeit nicht sehr beeinflussen, aber es lohnt sich, dies im Hinterkopf zu behalten (besonders bei kleinen Netzen).

In Abbildung 2-53 sind mehrere mit den gleichen Parametern trainierte Modelle dargestellt:

In[96]:

```
fig, axes = plt.subplots(2, 4, figsize=(20, 8))
for i, ax in enumerate(axes.ravel()):
    mlp = MLPClassifier(solver='lbfgs', random_state=i,
                         hidden_layer_sizes=[100, 100])
    mlp.fit(X_train, y_train)
    mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3, ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train, ax=ax)
```

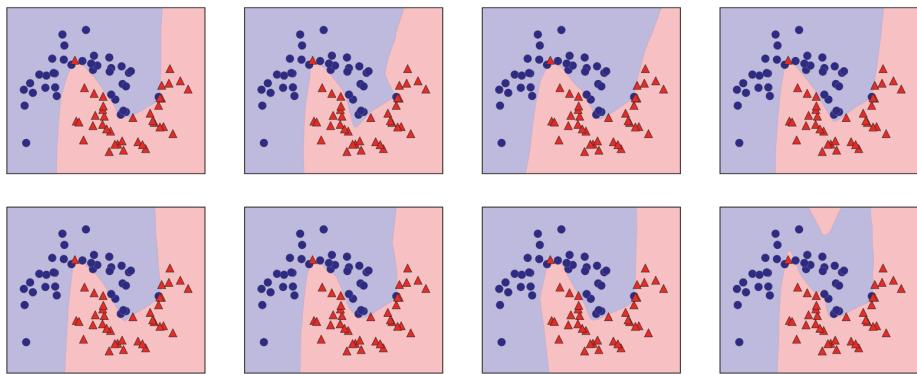


Abbildung 2-53: Mit den gleichen Paramtern aber unterschiedlichem Zufallswert trainierte Entscheidungsfunktionen

Für ein besseres Verständnis neuronaler Netze auf realen Daten wenden wir den `MLPClassifier` auf den Brustkrebs-Datensatz an. Wir beginnen mit den Standardparametern:

In[97]:

```
print("Maxima pro Merkmal für die Krebsdaten:\n{}".format(cancer.data.max(axis=0)))
```

Out[97]:

```
Maxima pro Merkmal für die Krebsdaten:
 [ 28.110   39.280   188.500  2501.000     0.163     0.345     0.427
   0.201     0.304     0.097     2.873     4.885    21.980  542.200
   0.031     0.135     0.396     0.053     0.079     0.030    36.040
  49.540   251.200  4254.000     0.223     1.058     1.252     0.291
   0.664     0.207 ]
```

In[98]:

```
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

mlp = MLPClassifier(random_state=42)
mlp.fit(X_train, y_train)

print("Genauigkeit auf den Trainingsdaten: {:.2f}".format(mlp.score(X_train, y_train)))
print("Genauigkeit auf den Testdaten: {:.2f}".format(mlp.score(X_test, y_test)))
```

**Out[98]:**

```
Genauigkeit auf den Trainingsdaten: 0.92
Genauigkeit auf den Testdaten: 0.90
```

Die Genauigkeit des MLP ist recht gut, reicht aber nicht an die anderen Modelle heran. Wie im Beispiel mit dem SVC-Modell liegt dies vermutlich an der Skalierung der Daten. Auch neuronale Netze erwarten, dass sämtliche Merkmale der Eingabe in ähnlicher Weise variieren und idealerweise den Mittelwert 0 und eine Varianz von 1 aufweisen. Wir müssen unsere Daten umskalieren, sodass diese Anforderungen erfüllt sind. Wir werden dies auch hier manuell tun, aber in Kapitel 3 den StandardScaler einführen, der diesen Schritt automatisiert:

**In[99]:**

```
# berechne den Mittelwert jedes Merkmals der Trainingsdaten
mean_on_train = X_train.mean(axis=0)
# berechne die Standardabweichung jedes Merkmals der Trainingsdaten
std_on_train = X_train.std(axis=0)

# subtrahiere den Mittelwert und skaliere mit der inversen
# Standardabweichung, sodass MW=0 und STABW=1 gilt
X_train_scaled = (X_train - mean_on_train) / std_on_train
# wende DIE GLEICHE Transformation (MW und STABW der Trainingsdaten) auf den
Testdatensatz an
X_test_scaled = (X_test - mean_on_train) / std_on_train

mlp = MLPClassifier(random_state=0)
mlp.fit(X_train_scaled, y_train)

print("Genauigkeit auf den Trainingsdaten: {:.3f}".format(
    mlp.score(X_train_scaled, y_train)))
print("Genauigkeit auf den Testdaten: {:.3f}".format(mlp.score(X_test_scaled, y_test)))
```

**Out[99]:**

```
Genauigkeit auf den Trainingsdaten: 0.991
Genauigkeit auf den Testdaten: 0.965
```

```
ConvergenceWarning:
  Stochastic Optimizer: Maximum iterations reached and the optimization
  hasn't converged yet.
```

Die Ergebnisse sind nach dem Skalieren sehr viel besser und können bereits mithalten. Wir haben allerdings eine Warnung vom Modell über das Erreichen der Maximalzahl von Iterationen erhalten. Dies ist Teil des Algorithmus `adam` zum Trainieren des Modells und sagt uns, dass wir die Anzahl der Iterationen erhöhen sollten:

**In[100]:**

```
mlp = MLPClassifier(max_iter=1000, random_state=0)
mlp.fit(X_train_scaled, y_train)

print("Genauigkeit auf den Trainingsdaten: {:.3f}".format(
    mlp.score(X_train_scaled, y_train)))
print("Genauigkeit auf den Testdaten: {:.3f}".format(mlp.score(X_test_scaled, y_test)))
```

**Out[100]:**

```
Genauigkeit auf den Trainingsdaten: 0.995
Genauigkeit auf den Testdaten: 0.965
```

Das Erhöhen der Anzahl Iterationen hat nur die Vorhersage auf den Trainingsdaten erhöht, aber nicht die Qualität der Verallgemeinerung. Das Modell ist noch immer recht gut. Da es aber eine Lücke zwischen der Genauigkeit auf Trainings- und Testdaten gibt, sollten wir die Komplexität des Modells verringern, um eine bessere Verallgemeinerung zu erzielen. Hier regularisieren wir die Wichtungsfaktoren stärker, indem wir den Parameter alpha erhöhen (sogar recht vehement von 0.0001 auf 1):

**In[101]:**

```
mlp = MLPClassifier(max_iter=1000, alpha=1, random_state=0)
mlp.fit(X_train_scaled, y_train)

print("Genauigkeit auf den Trainingsdaten: {:.3f}".format(
    mlp.score(X_train_scaled, y_train)))
print("Genauigkeit auf den Testdaten: {:.3f}".format(mlp.score(X_test_scaled, y_test)))
```

**Out[101]:**

```
Genauigkeit auf den Trainingsdaten: 0.988
Genauigkeit auf den Testdaten: 0.972
```

Die damit erhaltene Vorhersageleistung liegt mit den bisher besten Modellen auf Augenhöhe.<sup>12</sup>

Obwohl man analysieren kann, was ein neuronales Netz gelernt hat, ist es für gewöhnlich deutlich schwieriger, als ein lineares Modell oder einen Entscheidungsbaum zu analysieren. Eine Möglichkeit ist, die Gewichte des Modells zu inspizieren. Sie finden ein Beispiel hierzu in der scikit-learn example gallery ([http://scikit-learn.org/stable/auto\\_examples/neural\\_networks/plot\\_mnist\\_filters.html](http://scikit-learn.org/stable/auto_examples/neural_networks/plot_mnist_filters.html)). Beim Brustkrebs-Datensatz könnte dies etwas schwer zu verstehen sein. Das folgende Diagramm (Abbildung 2-54) zeigt die Gewichte, die die Eingabe mit der ersten verborgenen Schicht verbinden. Die Zeilen im Diagram entsprechen den 30 Eingabemerkmalen, die Spalten den 100 verborgenen Einheiten. Helle Farben stehen für große, positive Werte, dunkle Farben für negative Werte:

**In[102]:**

```
plt.figure(figsize=(20, 5))
plt.imshow(mlp.coefs_[0], interpolation='none', cmap='viridis')
plt.yticks(range(30), cancer.feature_names)
plt.xlabel("Spalten in der Gewichtungsmatrix")
plt.ylabel("Eingabemerkmal")
plt.colorbar()
```

---

<sup>12</sup> Sie haben vielleicht bemerkt, dass viele der besseren Modelle genau die gleiche Genauigkeit von 0.972 erzielen. Das bedeutet, dass diese Modelle genau die gleiche Anzahl Fehler begehen, nämlich vier. Wenn Sie die Vorhersagen genauer untersuchen, können Sie sogar herausfinden, dass sie alle die genau gleichen Fehler machen! Dies könnte daran liegen, dass der Datensatz sehr klein ist oder dass diese Punkte sich wirklich stark von den übrigen unterscheiden.

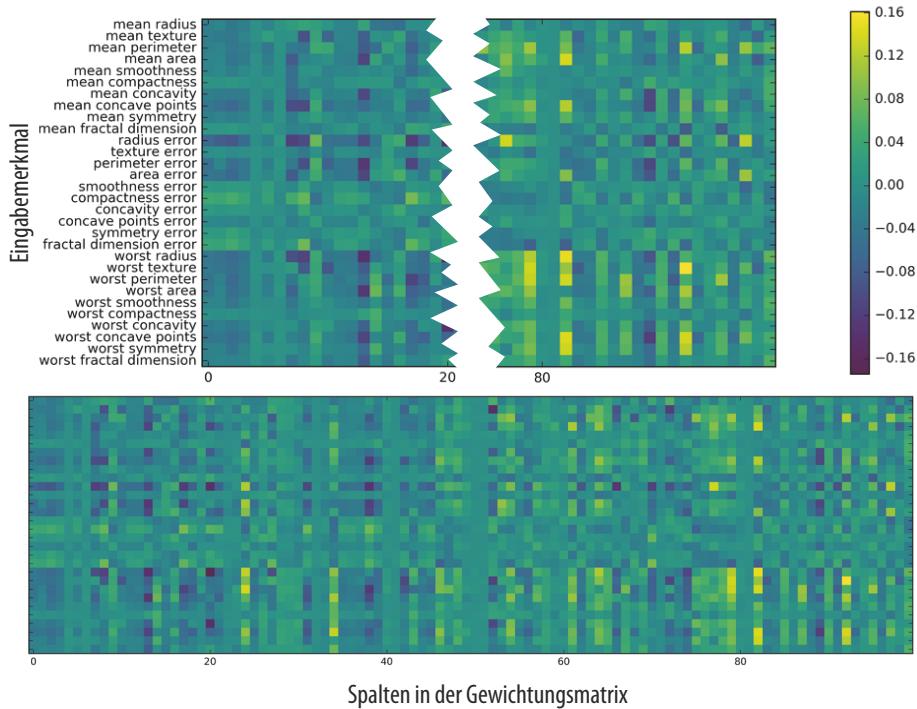


Abbildung 2-54: Heatmap der Gewichte der ersten Schicht in einem mit dem Brustkrebs-Datensatz trainierten neuronalen Netz

Eine mögliche Erkenntnis hieraus ist, dass Merkmale mit sehr kleinen Gewichten bei sämtlichen verborgenen Einheiten für das Modell »weniger wichtig« sind. Wir sehen, dass »mean smoothness« und »mean compactness« zusätzlich zu den Merkmalen zwischen »smoothness error« und »fractal dimension error« im Vergleich zu den übrigen Merkmalen relativ geringe Gewichte aufweisen. Das könnte bedeuten, dass diese Merkmale weniger wichtig sind oder dass wir sie nicht in einer für das neuronale Netz nutzbaren Weise repräsentiert haben.

Wir könnten auch die Gewichte, die die verborgene Schicht mit der Ausgabe verbinden, visualisieren. Allerdings sind diese noch schwieriger zu interpretieren.

Die Klassen `MLPClassifier` und `MLPRegressor` bieten eine leicht nutzbare Schnittstelle zu verbreiteten neuronalen Netzarchitekturen. Dies ist nur ein kleiner Teil dessen, was mit neuronalen Netzen möglich ist. Wenn Sie mit flexibleren oder größeren Modellen arbeiten möchten, empfehlen wir Ihnen, sich die fantastischen Bibliotheken außerhalb von scikit-learn anzuschauen. Etablierte Bibliotheken für Python-Nutzer sind keras, lasagna und tensor-flow. lasagna setzt auf der Bibliothek theano auf, während keras entweder tensor-flow oder theano nutzen kann. Diese Bibliotheken bieten eine wesentlich flexiblere Schnittstelle zum Erstellen neuronaler Netze und zeugen vom rasanten Fortschritte in der Deep-Learning-For-

schung. Sämtliche dieser beliebten Bibliotheken für Deep Learning können leistungsfähige Grafikchips (GPUs) nutzen, was scikit-learn nicht unterstützt. Mit GPUs lassen sich die Berechnungen um den Faktor 10 bis 100 beschleunigen. Sie sind für die Anwendung von Deep-Learning-Methoden auf großen Datensätze unentbehrlich.

## Stärken, Schwächen und Parameter

Neuronale Netze sind bei vielen Anwendungen maschinellen Lernens als Modelle erster Wahl erneut auf den Plan getreten. Einer ihrer vielen Vorteile ist, dass sie Informationen aus großen Datenmengen erfassen und daraus ungeheuer komplexe Modelle erstellen können. Mit genügend Rechenzeit, Daten und umsichtigem Einstellen der Parameter schlagen neuronale Netze oft die anderen maschinellen Lernverfahren (bei Klassifikations- und Regressionsaufgaben).

Damit kommen wir auch schon zu den Nachteilen. Neuronale Netze benötigen zum Trainieren oft lange. Sie erfordern, wie hier gezeigt, auch eine umsichtige Vorbereitung der Daten. Ähnlich wie bei SVMs funktionieren sie am besten mit »homogenen« Daten, bei denen sämtliche Merkmale eine ähnliche Bedeutung haben. Bei Daten mit sehr unterschiedlichen Arten von Merkmalen funktionieren auf Bäumen basierende Modelle besser. Das Optimieren neuronaler Netze ist eine Kunst für sich. In unseren Versuchen haben wir gerade einmal an der Oberfläche dessen gekratzt, was beim Einstellen und Trainieren neuronaler Netze möglich ist.

**Abschätzen der Komplexität neuronaler Netze.** Die wichtigsten Parameter sind die Anzahl der Schichten und die verborgener Einheiten pro Schicht. Sie sollten mit einer oder zwei verborgenen Schichten beginnen und sich von dort aus vortasten. Die Anzahl Knoten pro verborgener Schicht ist oft ähnlich zur Anzahl der Eingabemerkmale, aber selten höher als im niedrigen oder mittleren Tausenderbereich.

Ein hilfreiches Maß für die Komplexität eines neuronalen Netzes ist die Anzahl der erlernten Gewichte oder Koeffizienten. Wenn Sie einen binären Klassifikationsdatensatz mit 100 Merkmalen und 100 verborgene Einheiten haben, so gibt es  $100 * 100 = 10000$  Gewichte zwischen der Eingabe und der ersten verborgenen Schicht. Es gibt außerdem  $100 * 1 = 100$  Gewichte zwischen der verborgenen und der Ausgabeschicht, also insgesamt 10100 Gewichte. Wenn Sie eine zweite verborgene Schicht mit 100 Einheiten hinzufügen, so kommen noch einmal  $100 * 100 = 10,000$  Gewichte zwischen der ersten und zweiten verborgenen Schicht hinzu. Insgesamt wären es dann 20100 Gewichte. Falls Sie stattdessen eine Schicht mit 1000 verborgenen Einheiten verwenden, trainieren Sie  $100 * 1000 = 100000$  Gewichte zwischen der Eingabe und der verborgenen Schicht und  $1000 * 1 = 1000$  Gewichte zwischen der verborgenen Schicht und der Ausgabe, insgesamt also 101000. Wenn Sie nun eine zweite verborgene Schicht hinzufügen, sind dort  $1000 * 1000 = 1000000$  Gewichte, sodass die Gesamtzahl auf eindrucksvolle 1101000 ansteigt – 50 Mal höher als das Modell mit zwei verborgenen Schichten der Größe 100.

Eine übliche Strategie zum Anpassen der Parameter eines neuronalen Netzes ist, zuerst ein so großes Netz zu erstellen, dass es zu Overfitting kommt. Dies stellt sicher, dass das Netz die Aufgabe erlernen kann. Sobald Sie das wissen, können Sie das Netz verkleinern oder den Regularisierungsparameter alpha erhöhen, wodurch sich die Fähigkeit zum Verallgemeinern verbessert.

In unseren Versuchen haben wir uns vor allem auf die Definition des Modells konzentriert: die Anzahl der Schichten und die der Knoten pro Schicht, die Regularisierung und die Nichtlinearität. Diese definieren das zu trainierende Modell. Es gibt außerdem die Frage, wie das Modell trainiert werden soll, also den zum Erlernen der Parameter verwendeten Algorithmus. Dieser wird über den Parameter algorithm festgelegt. Es gibt zwei leicht einsetzbare Auswahlmöglichkeiten für algorithm. Die Voreinstellung ist adam, die in den meisten Situationen gut funktioniert, aber recht anfällig für die Skalierung der Daten ist (es ist also wichtig, Ihre Daten immer auf Mittelwert 0 und Varianz 1 zu skalieren). Der zweite Algorithmus ist 'lbfgs', der recht robust, aber bei größeren Modellen und Datensätzen rechenintensiver ist. Für Fortgeschrittene gibt es auch die Option 'sgd', die von vielen Deep-Learning-Forschern eingesetzt wird. Die 'sgd'-Option enthält viele zusätzliche Parameter, die für ein optimales Ergebnis eingestellt werden müssen. Sie können die Beschreibung dieser Parameter in der Nutzerdokumentation finden. Wenn Sie noch nicht lange mit MLPs arbeiten, empfehlen wir Ihnen, bei 'adam' und 'lbfgs' zu bleiben.



#### **fit setzt ein Modell zurück**

Eine wichtige Eigenschaft von Modellen in scikit-learn ist, dass ein Aufruf von fit ein zuvor trainiertes Modell zurücksetzt. Wenn Sie also ein Modell auf einem Datensatz aufbauen und anschließend fit für einen anderen Datensatz aufrufen, »vergisst« das Modell alles, was es über den ersten Datensatz weiß. Sie können fit so oft aufrufen, wie Sie möchten. Das Ergebnis ist das gleiche wie bei einem »neu erstellten« Modell.

## **Schätzungen der Unsicherheit von Klassifikatoren**

Ein noch nicht behandelter nützlicher Teil der Schnittstelle von scikit-learn ist die Fähigkeit von Klassifikatoren, Schätzungen für die Unsicherheit von Vorhersagen anzugeben. Oft ist man nicht nur daran interessiert, welche Kategorie für einen bestimmten Datenpunkt vorhergesagt wird, sondern auch, wie sicher es sich um die richtige Kategorie handelt. In der Praxis haben unterschiedliche Arten von Fehlern stark unterschiedliche Folgen in realen Anwendungen. Stellen Sie sich einmal eine Krebsdiagnose als medizinische Anwendung vor. Eine falsch positive Vorhersage könnte bei einem Patienten zu zusätzlichen Untersuchungen führen, eine falsch negative Vorhersage jedoch dazu, dass eine ernsthafte Erkrankung nicht behandelt wird. Wir werden uns mit diesem Thema in Kapitel 6 genauer beschäftigen.

Es gibt zwei Funktionen in scikit-learn, mit denen wir Schätzungen der Unsicherheit von Klassifikatoren erhalten können: `decision_function` und `predict_proba`. Die meisten (aber nicht alle) Klassifikatoren enthalten mindestens eine davon, und viele Klassifikatoren enthalten beide. Schauen wir uns an, was diese beiden Funktionen mit einem synthetischen zweidimensionalen Datensatz tun, wenn wir einen GradientBoostingClassifier-Klassifikator erstellen. Dieser enthält sowohl die Methode `decision_function` als auch `predict_proba`:

**In[103]:**

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.datasets import make_circles
X, y = make_circles(noise=0.25, factor=0.5, random_state=1)

# wir nennen die Kategorien zur Veranschaulichung "blue" und "red"
y_named = np.array(["blue", "red"])[y]

# wir können train_test_split mit beliebig vielen Arrays aufrufen;
# alle werden konsistent aufgeteilt
X_train, X_test, y_train_named, y_test_named, y_train, y_test = \
    train_test_split(X, y_named, y, random_state=0)

# erstelle ein Modell mit Gradient Boosting
gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train_named)
```

## Die Entscheidungsfunktion

Im Fall der binären Klassifikation ist der Rückgabewert von `decision_function` ein Array mit den Abmessungen (`n_samples`,), und er enthält eine Fließkommazahl pro Datenpunkt:

**In[104]:**

```
print("X_test.shape: {}".format(X_test.shape))
print("Abmessungen der Entscheidungsfunktion: {}".format(
    gbrt.decision_function(X_test).shape))
```

**Out[104]:**

```
X_test.shape: (25, 2)
Abmessungen der Entscheidungsfunktion: (25,)
```

Die Werte im Array zeigen an, wie stark das Modell daran glaubt, dass ein Datenpunkt der »positiven« Kategorie angehört, in diesem Fall Kategorie 1. Positive Zahlen stehen dabei für eine Bevorzugung der positiven Kategorie und negative Zahlen für eine Bevorzugung der »negativen« (anderen) Kategorie:

**In[105]:**

```
# zeige die ersten Einträge der Entscheidungsfunktion
print("Entscheidungsfunktion:\n{}".format(gbrt.decision_function(X_test)[:6]))
```

**Out[105]:**

```
Entscheidungsfunktion:  
[ 4.136 -1.683 -3.951 -3.626  4.29   3.662]
```

Wir können die Vorhersage reproduzieren, indem wir uns lediglich das Vorzeichen der Entscheidungsfunktion ansehen:

**In[106]:**

```
print("Entscheidungsfunktion mit Schwellenwert:\n{}".format(  
      gbrt.decision_function(X_test) > 0))  
print("Vorhersagen:\n{}".format(gbrt.predict(X_test)))
```

**Out[106]:**

```
Entscheidungsfunktion mit Schwellenwert:  
[ True False False False  True  True  True  True  True  True  
  True False  True False False  True  True  True  True  True  True  
  False]  
Vorhersagen:  
['red' 'blue' 'blue' 'blue' 'red' 'red' 'blue' 'red' 'red' 'red'  
 'red' 'red' 'blue' 'red' 'blue' 'blue' 'blue' 'red' 'red' 'red'  
 'red' 'red' 'blue' 'blue']
```

Bei einer binären Klassifikation ist die »negative« Kategorie stets der erste Eintrag des Attributes `classes_` und die »positive« Kategorie der zweite Eintrag von `classes_`. Wenn Sie also die vollständige Ausgabe von `predict` nachbauen möchten, können Sie sich das Attribut `classes_` zunutze machen:

**In[107]:**

```
# wandle das boolesche True/False in 0 und 1 um  
greater_zero = (gbrt.decision_function(X_test) > 0).astype(int)  
# verwende 0 und 1 als Indices für classes_  
pred = gbrt.classes_[greater_zero]  
# pred ist das Gleiche wie die Ausgabe von gbrt.predict  
print("pred ist mit den Vorhersagen identisch: {}".format(  
    np.all(pred == gbrt.predict(X_test))))
```

**Out[107]:**

```
pred ist mit den Vorhersagen identisch: True
```

Die Spannbreite von `decision_function` kann beliebig sein und hängt von den Daten und den Modellparametern ab:

**In[108]:**

```
decision_function = gbrt.decision_function(X_test)  
print("Minimum der Entscheidungsfunktion: {:.2f} Maximum: {:.2f}".format(  
    np.min(decision_function), np.max(decision_function)))
```

**Out[108]:**

```
Minimum der Entscheidungsfunktion: -7.69 Maximum: 4.29
```

Diese willkürliche Skalierung macht die Ausgabe von `decision_function` häufig schwer interpretierbar.

Im folgenden Beispiel plotten wir `decision_function` als Farbskale mit allen Punkten auf einer 2-D-Ebene neben einer Darstellung der bereits bekannten Entscheidungsgrenze. Wir stellen die Punkte aus dem Trainingsdatensatz als Kreise und die Testdaten als Dreiecke dar (Abbildung 2-55):

**In[109]:**

```
fig, axes = plt.subplots(1, 2, figsize=(13, 5))
mglearn.tools.plot_2d_separator(gbrt, X, ax=axes[0], alpha=.4,
                                fill=True, cm=mglearn.cm2)
scores_image = mglearn.tools.plot_2d_scores(gbrt, X, ax=axes[1],
                                            alpha=.4, cm=mglearn.ReBl)

for ax in axes:
    # plotte die Trainings- und Testdatenpunkte
    mglearn.discrete_scatter(X_test[:, 0], X_test[:, 1], y_test,
                              markers='^', ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train,
                              markers='o', ax=ax)
    ax.set_xlabel("Merkmal 0")
    ax.set_ylabel("Merkmal 1")
cbar = plt.colorbar(scores_image, ax=axes.tolist())
axes[0].legend(["Test Kategorie 0", "Test Kategorie 1", "Train Kategorie 0",
                "Train Kategorie 1"], ncol=4, loc=(.1, 1.1))
```

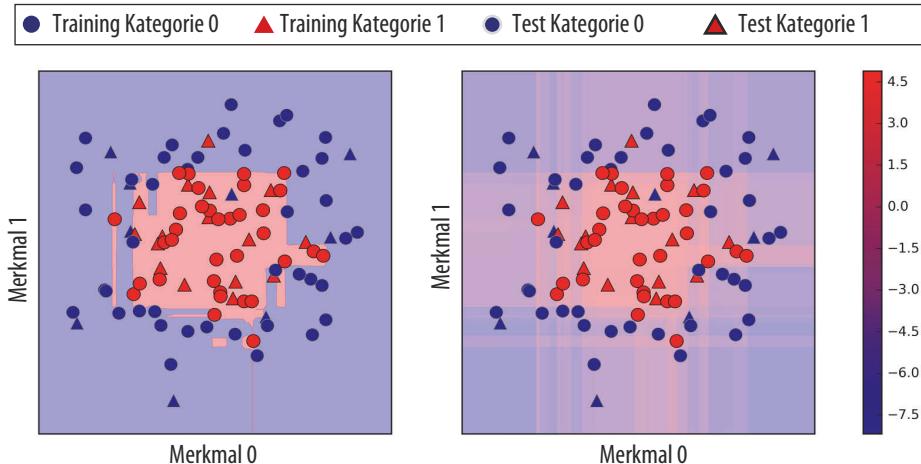


Abbildung 2-55: Entscheidungsgrenze (links) und Entscheidungsfunktion (rechts) für ein Gradient Boosting-Modell auf einem zweidimensionalen Beispieldatensatz

Zusätzlich zum vorhergesagten Ergebnis liefert uns die Angabe, wie sicher sich der Klassifikator über dieses Ergebnis ist, darüber hinausgehende Informationen. In dieser Darstellung ist es allerdings schwierig, die Grenze zwischen beiden Kategorien auszumachen.

## Vorhersagen von Wahrscheinlichkeiten

Die Ausgabe von `predict_proba` ist eine Wahrscheinlichkeit für jede Kategorie, die sich leichter verstehen lässt als die Ausgabe von `decision_function`. Das Array hat bei einem binären Klassifikator immer die Abmessungen (`n_samples, 2`):

**In[110]:**

```
print("Abmessungen der Wahrscheinlichkeiten: {}".format(
    gbdt.predict_proba(X_test).shape))
```

**Out[110]:**

```
Abmessungen der Wahrscheinlichkeiten: (25, 2)
```

Der erste Eintrag jeder Zeile ist die geschätzte Wahrscheinlichkeit für die erste Kategorie und der zweite Eintrag die geschätzte Wahrscheinlichkeit für die zweite Kategorie. Weil es sich um eine Wahrscheinlichkeit handelt, liegen die Ausgabewerte von `predict_proba` stets zwischen 0 und 1. Die Summe der Einträge beider Kategorien ist immer 1:

**In[111]:**

```
# zeige die ersten paar Einträge von predict_proba an
print("vorhergesagte Wahrscheinlichkeiten:\n{}".format(
    gbdt.predict_proba(X_test[:6])))
```

**Out[111]:**

```
vorhergesagte Wahrscheinlichkeiten:
[[ 0.016  0.984]
 [ 0.843  0.157]
 [ 0.981  0.019]
 [ 0.974  0.026]
 [ 0.014  0.986]
 [ 0.025  0.975]]
```

Weil die Wahrscheinlichkeiten der beiden Kategorien zusammen 1 ergeben, liegt die Konfidenz für genau eine der Kategorien über 50 %. Dies ist die vorhergesagte Kategorie.<sup>13</sup>

Sie können in der obigen Ausgabe sehen, dass der Klassifikator sich bei den meisten Punkten recht sicher ist. Wie gut die Unsicherheit des Modells die Unsicherheit in den Daten wiedergibt, hängt vom Modell und den Parametern ab. Mit Overfitting neigt ein Modell dazu, sicherere Vorhersagen abzugeben, auch wenn diese falsch sind. Ein weniger komplexes Modell enthält für gewöhnlich mehr Unsicherheit in seinen Vorhersagen. Man nennt ein Modell *kalibriert*, wenn die ausgegebene Unsicherheit der tatsächlichen Genauigkeit entspricht. In einem kalibrierten Modell wäre eine mit 70 % Konfidenz getroffene Vorhersage in 70 % der Fälle korrekt.

---

<sup>13</sup> Weil die Wahrscheinlichkeiten Fließkommazahlen sind, ist es unwahrscheinlich, dass beide Werte genau 0.500 betragen. Falls das doch passiert, wird die Vorhersage zufällig getroffen.

Im folgenden Beispiel (Abbildung 2-56) stellen wir noch einmal die Entscheidungsgrenze auf dem Datenatz dar und daneben die Wahrscheinlichkeiten für Kategorie 1:

In[112]:

```
fig, axes = plt.subplots(1, 2, figsize=(13, 5))

mglearn.tools.plot_2d_separator(
    gbrt, X, ax=axes[0], alpha=.4, fill=True, cm=mglearn.cm2)
scores_image = mglearn.tools.plot_2d_scores(
    gbrt, X, ax=axes[1], alpha=.5, cm=mglearn.ReBl, function='predict_proba')

for ax in axes:
    # plotte die Trainings- und Testdatenpunkte
    mglearn.discrete_scatter(X_test[:, 0], X_test[:, 1], y_test,
                              markers='^', ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train,
                              markers='o', ax=ax)
    ax.set_xlabel("Merkmal 0")
    ax.set_ylabel("Merkmal 1")
cbar = plt.colorbar(scores_image, ax=axes.tolist())
axes[0].legend(["Test Kategorie 0", "Test Kategorie 1", "Training Kategorie 0",
                "Training Kategorie 1"], ncol=4, loc=(.1, 1.1))
```

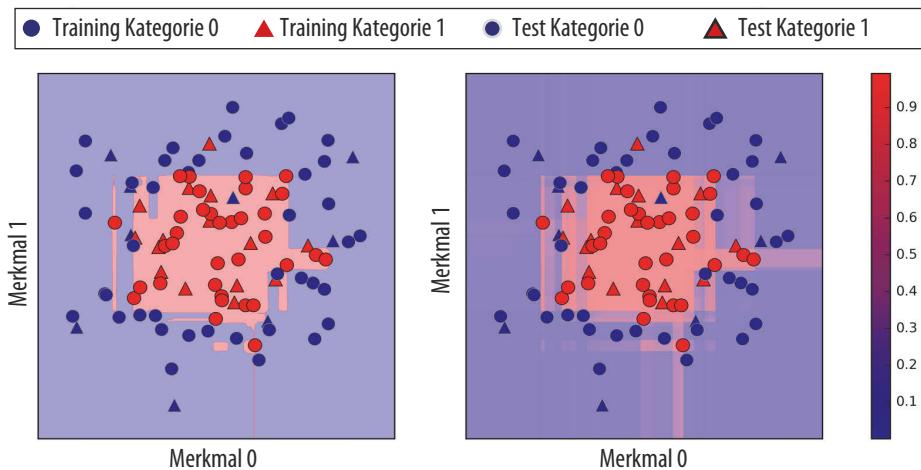


Abbildung 2-56: Entscheidungsgrenze (links) und vorhergesagte Wahrscheinlichkeiten für das Modell mit Gradient Boosting in Abbildung 2-55

Die Grenzen sind in diesem Diagramm viel schärfer definiert, und einige kleinere unsichere Regionen sind deutlich zu sehen.

Die scikit-learn-Website (<http://bit.ly/2cqCYx6>) zeigt einen großartigen Vergleich vieler Modelle und ihrer Schätzungen der Unsicherheit. Wir haben diese Darstellung für Abbildung 2-57 reproduziert und ermuntern Sie, die Beispiele auf der Seite durchzugehen.

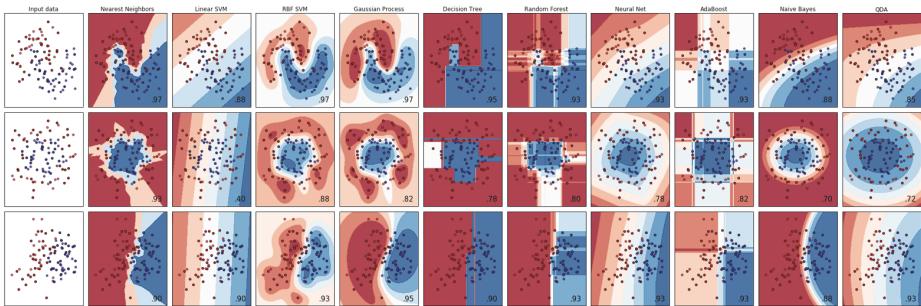


Abbildung 2-57: Vergleich mehrerer Klassifikationsverfahren in scikit-learn auf synthetischen Datensätzen (Bild von <http://scikit-learn.org> zur Verfügung gestellt)

## Unsicherheit bei der Klassifikation mehrerer Kategorien

Bisher haben wir nur über das Abschätzen der Unsicherheit bei der binären Klassifikation gesprochen. Die Methoden `decision_function` und `predict_proba` funktionieren aber auch bei mehreren Kategorien. Wenden wir sie einmal auf den Iris-Datensatz an, eine Klassifikationsaufgabe mit drei Kategorien:

**In[113]:**

```
from sklearn.datasets import load_iris

iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, random_state=42)

gbdt = GradientBoostingClassifier(learning_rate=0.01, random_state=0)
gbdt.fit(X_train, y_train)
```

**In[114]:**

```
print("Abmessungen der Entscheidungsfunktion: {}".format(
    gbdt.decision_function(X_test).shape))
# plotte die ersten Einträge der Entscheidungsfunktion
print("Entscheidungsfunktion:\n{}".format(gbdt.decision_function(X_test)[:6, :]))
```

**Out[114]:**

```
Abmessungen der Entscheidungsfunktion: (38, 3)
Entscheidungsfunktion:
[[ -0.529  1.466 -0.504]
 [ 1.512 -0.496 -0.503]
 [ -0.524 -0.468  1.52 ]
 [ -0.529  1.466 -0.504]
 [ -0.531  1.282  0.215]
 [ 1.512 -0.496 -0.503]]
```

Bei mehreren Kategorien hat das Array `decision_function` die Abmessungen (`n_datenpunkte, n_kategorien`). Jede Spalte enthält ein »Gewissheitsmaß« für jede Kategorie, wobei ein hoher Score bedeutet, dass diese Kategorie wahrscheinlicher

ist, und ein geringer Score bedeutet, dass diese Kategorie weniger wahrscheinlich ist. Sie können aus diesen Scores die Vorhersagen ableiten, indem Sie für jeden Datenpunkt den höchsten Eintrag heraussuchen:

**In[115]:**

```
print("Argmax der Entscheidungsfunktion:\n{}".format(
    np.argmax(gbrt.decision_function(X_test), axis=1)))
print("Vorhersagen:\n{}".format(gbrt.predict(X_test)))
```

**Out[115]:**

```
Argmax der Entscheidungsfunktion:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0]
Vorhersagen:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0]
```

Die Ausgabe von predict\_proba hat die gleichen Abmessungen (n\_datenpunkte, n\_kategorien). Auch hier ist die Summe der Wahrscheinlichkeiten der möglichen Kategorien für jeden Datenpunkt genau 1:

**In[116]:**

```
# gib die ersten Einträge von predict_proba aus
print("Vorhergesagte Wahrscheinlichkeiten:\n{}".format(
    gbrt.predict_proba(X_test)[:6]))
# weise nach, dass die Summe jeder Zeile eins ist
print("Summen: {}".format(gbrt.predict_proba(X_test)[:6].sum(axis=1)))
```

**Out[116]:**

```
Vorhergesagte Wahrscheinlichkeiten:
[[ 0.107  0.784  0.109]
 [ 0.789  0.106  0.105]
 [ 0.102  0.108  0.789]
 [ 0.107  0.784  0.109]
 [ 0.108  0.663  0.228]
 [ 0.789  0.106  0.105]]
Summen: [ 1.  1.  1.  1.  1.  1.]
```

Wir können die Vorhersagen auch hier ermitteln, indem wir die Funktion argmax auf predict\_proba anwenden:

**In[117]:**

```
print("Argmax der vorhergesagten Wahrscheinlichkeiten:\n{}".format(
    np.argmax(gbrt.predict_proba(X_test), axis=1)))
print("Vorhersagen:\n{}".format(gbrt.predict(X_test)))
```

**Out[117]:**

```
Argmax der vorhergesagten Wahrscheinlichkeiten:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0]
Vorhersagen:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0]
```

Zusammengefasst sind predict\_proba und decision\_function Arrays, die als Abmessungen immer (n\_datenpunkte, n\_kategorien) haben – außer bei decision\_

function im binären Sonderfall. Bei einer binären Klassifikation hat decision\_function nur eine Spalte, die der »positiven« Kategorie classes\_[1] entspricht. Dies hat vor allem historische Ursachen.

Wenn die Anzahl der Spalten n\_kategorien beträgt, können Sie die Vorhersagen durch Anwenden der argmax-Funktion über die Spalten reproduzieren. Seien Sie jedoch vorsichtig, falls Ihre Kategorien Strings oder keine bei 0 beginnende kontinuierliche Folge ganzer Zahlen sind. Wenn Sie mit predict erhaltene Ergebnisse mit den über decision\_function oder predict\_proba erhaltenen vergleichen möchten, sollten Sie das Attribut classes\_ verwenden, um die eigentlichen Namen der Kategorien zu bekommen:

In[118]:

```
logreg = LogisticRegression()

# stelle jeden Zielwert durch die Namen der Kategorien
# im Datensatz iris dar
named_target = iris.target_names[y_train]
logreg.fit(X_train, named_target)
print("eindeutige Kategorien im Iris-Datensatz: {}".format(logreg.classes_))
print("Vorhersagen: {}".format(logreg.predict(X_test)[:10]))
argmax_dec_func = np.argmax(logreg.decision_function(X_test), axis=1)
print("argmax der Entscheidungsfunktion: {}".format(argmax_dec_func[:10]))
print("argmax mit classes_ kombiniert: {}".format(
    logreg.classes_[argmax_dec_func][:10]))
```

Out[118]:

```
eindeutige Kategorien im Iris-Datensatz: ['setosa' 'versicolor' 'virginica']
Vorhersagen: ['versicolor' 'setosa' 'virginica' 'versicolor' 'versicolor'
              'setosa' 'versicolor' 'virginica' 'versicolor' 'versicolor']
argmax der Entscheidungsfunktion: [1 0 2 1 1 0 1 2 1 1]
argmax mit classes_ kombiniert: ['versicolor' 'setosa' 'virginica' 'versicolor'
                                   'versicolor' 'setosa' 'versicolor' 'virginica' 'versicolor' 'versicolor']
```

## Zusammenfassung und Ausblick

Wir haben dieses Kapitel mit einer Diskussion zur Komplexität von Modellen eingeleitet, anschließend *Verallgemeinerung* besprochen, das Trainieren eines Modells, das auf neuen, vorher nicht bekannten Daten eine gute Vorhersage leistet. Darüber haben wir die Begriffe Underfitting und Overfitting kennengelernt. Underfitting beschreibt ein Modell, das die Variationen in den Trainingsdaten nicht erfassen kann. Overfitting beschreibt ein Modell, welches sich zu stark auf die Trainingsdaten konzentriert und daher nicht sehr gut auf neue Daten verallgemeinern kann.

Wir haben anschließend eine große Bandbreite maschineller Lernmodelle zur Klassifikation und Regression besprochen und dabei deren Vor- und Nachteile sowie Möglichkeiten zum Kontrollieren der Modellkomplexität kennengelernt. Wir

haben gesehen, dass es bei vielen der Algorithmen für eine hohe Vorhersagegüte wichtig ist, die richtigen Parameter zu setzen. Einige der Daten reagieren empfindlich auf die Repräsentation der Eingabedaten, insbesondere auf die Skalierung der Merkmale. Daher erhält man selten ein genaues Modell, wenn man einen Algorithmus blindlings auf einen Datensatz anwendet, ohne die dem Modell zugrunde liegenden Annahmen und die Bedeutung der Parameter zu verstehen.

Dieses Kapitel enthält eine Menge Informationen über die Algorithmen. Für die folgenden Kapitel ist es nicht notwendig, dass Sie sich an sämtliche Details erinnern. Etwas Grundwissen zu den hier beschriebenen Modellen – und in welchen Situationen sich diese anwenden lassen – ist zur erfolgreichen praktischen Anwendung wichtig. Hier folgt eine kurze Zusammenfassung, wann Sie welches Modell anwenden können:

#### *Nächste-Nachbarn*

Bei kleinen Datensätzen, gut als Grundlinie geeignet, einfach zu erklären.

#### *Lineare Modelle*

Ausgezeichnet für den ersten Versuch, gut für sehr große Datensätze, gut für Daten mit sehr vielen Dimensionen.

#### *Naive Bayes*

Nur zur Klassifizierung. Noch schneller als lineare Modelle, gut für sehr große Datensätze und höherdimensionale Daten. Oft weniger genau als lineare Modelle.

#### *Entscheidungsbäume*

Sehr schnell, benötigen kein Skalieren der Daten, lassen sich visualisieren und schnell erklären.

#### *Random Forests*

Schneiden fast immer besser ab als ein einzelner Entscheidungsbaum, sehr robust und mächtig. Benötigen kein Skalieren der Daten. Nicht gut für sehr hochdimensionale oder dünn besetzte Daten geeignet.

#### *Entscheidungsbäume mit Gradient Boosting*

Oft etwas genauer als Random Forests. Langsamer zu trainieren, aber bei der Vorhersage schneller als Random Forests. Geringerer Speicherverbrauch. Benötigen mehr Feineinstellung von Parametern als Random Forests.

#### *Support Vector Machines*

Mächtig bei Datensätzen mittlerer Größe mit Merkmalen ähnlicher Bedeutung. Benötigen Skalieren der Daten und reagieren empfindlich auf Parametereinstellungen.

#### *Neuronale Netze*

Erlauben den Aufbau sehr komplexer Modelle, insbesondere für große Datensätze. Reagieren empfindlich auf die Skalierung der Daten und die Auswahl der Parameter. Große Modelle benötigen lange zum Trainieren.

Bei der Arbeit mit einem neuen Datensatz ist es für gewöhnlich eine gute Idee, mit einem einfachen Modell zu beginnen, z. B. einem linearen Modell, einem Naive Bayes- oder einem nächste-Nachbarn-Klassifikator, und zu schauen, wie weit man damit kommt. Sobald Sie Ihre Daten etwas besser verstehen, können Sie den Wechsel zu einem Algorithmus erwägen, der komplexere Modelle erstellen kann, wie etwa Random Forests, Entscheidungsbäume mit Gradient Boosting, SVMs oder neuronale Netze.

Sie sollten nun eine ungefähre Idee davon haben, wie sich die hier vorgestellten Modelle anwenden, einstellen und analysieren lassen. In diesem Kapitel haben wir uns auf binäre Klassifikationen konzentriert, da diese meist am einfachsten zu verstehen sind. Die meisten der hier vorgestellten Algorithmen gibt es in einer Klassifikations- und einer Regressionsvariante. Allerdings unterstützen sämtliche Algorithmen zur Klassifikation sowohl die binäre als auch die Klassifikation mit mehreren Kategorien. Versuchen Sie, einen dieser Algorithmen auf die in scikit-learn eingebauten Datensätze anzuwenden, wie etwa die Datensätze `boston_housing` oder `diabetes` zur Regression oder den Datensatz `digits` zur Klassifikation mit mehreren Kategorien. Durch Herumprobieren mit diesen Algorithmen auf unterschiedlichen Datensätzen werden Sie ein besseres Gefühl dafür entwickeln, wie lange diese zum Trainieren brauchen, wie leicht sich die Modelle analysieren lassen und wie sensibel sie auf die Repräsentation der Daten reagieren.

Auch wenn wir die Folgen unterschiedlicher Parameter bei den betrachteten Algorithmen untersucht haben, ist es in der Praxis doch etwas schwieriger, ein Modell zu konstruieren, das gut auf neue Daten verallgemeinert. Wir werden in Kapitel 6 noch sehen, wie sich Parameter besser einstellen lassen und wie Sie gute Parameter automatisiert ermitteln können.

Zunächst werden wir uns im nächsten Kapitel allerdings ausführlicher mit unüberwachtem Lernen und der Vorverarbeitung von Daten beschäftigen.

# Unüberwachtes Lernen und Vorverarbeitung

Die zweite Gruppe von Algorithmen für maschinelles Lernen, die wir betrachten werden, sind Algorithmen für unüberwachtes Lernen. Unter diesen Begriff fallen sämtliche Arten von maschinellem Lernen, bei denen das Ergebnis unbekannt ist und es keinen Lehrer zum Trainieren des Algorithmus gibt. Beim unüberwachten Lernen erhält der Lernalgorithmus lediglich die Eingabedaten und wird damit beauftragt, Wissen aus diesen Daten zu extrahieren.

## Arten von unüberwachtem Lernen

Wir werden uns in diesem Kapitel zwei Arten unüberwachten Lernens ansehen: Transformationen des Datensatzes und Clusterverfahren.

*Unüberwachte Transformationen* eines Datensatzes sind Algorithmen, die eine neue Repräsentation der Daten erzeugen, die für Menschen oder andere maschinelle Lernalgorithmen besser verständlich ist als deren ursprüngliche Darstellung. Eine häufige Anwendung der unüberwachten Transformation ist die Dimensionsreduktion, mit der sich aus einer höher dimensionierten Repräsentation der Daten mit vielen Merkmalen eine zusammengefasste Repräsentation weniger zentraler Merkmale ermitteln lässt. Ein verbreitetes Anwendungsbeispiel der Dimensionsreduktion ist die Projektion auf zwei Dimension, um Daten besser visualisieren zu können.

Eine weitere Anwendung für unüberwachte Transformationen ist das Finden von Teilen oder Komponenten, die »den Kern« der Daten darstellen. Ein Beispiel hierfür ist das Finden von Themen in einer Sammlung von Textdokumenten. Die Aufgabe besteht darin, unbekannte Themen zu finden, die in allen Dokumenten erwähnt werden, und zu erfahren, welche Themen in allen Dokumenten vorkommen. Das ist zum Verfolgen von Diskussionen zu Themen wie Wahlen, Waffengesetze oder Popstars in sozialen Medien nützlich.

*Clusterverfahren* dagegen teilen Datensätze in separate Gruppen mit ähnlichen Elementen ein. Betrachten wir als Beispiel das Hochladen von Bildern in ein soziales

Netzwerk. Um Ihre Bilder zu sortieren, könnte die Webseite versuchen, Bilder mit der gleichen Person nebeneinanderzustellen. Allerdings weiß die Webseite nicht, wer auf welchem Bild zu sehen ist und wie viele unterschiedliche Personen in Ihrer Fotosammlung vertreten sind. Ein sinnvoller Ansatz wäre, alle Gesichter zu extrahieren und Gruppen mit ähnlichen Gesichtern zu bilden. Diese entsprechen dann hoffentlich den gleichen Personen, sodass Sie die Bilder nicht selbst anordnen müssen.

## Herausforderungen beim unüberwachten Lernen

Das Hauptproblem beim unüberwachten Lernen ist, auszuwerten, ob der Algorithmus etwas Nützliches gelernt hat. Für gewöhnlich werden Algorithmen zum unüberwachten Lernen auf nicht markierte Daten angewendet, sodass wir nicht wissen, wie die korrekte Ausgabe aussehen soll. Deshalb ist es sehr schwer zu entscheiden, ob ein Modell »richtigliegt«. Beispielsweise könnte unser hypothetisches Clusterverfahren alle Personen im Profil und alle Frontalansichten gruppieren. Natürlich ist das eine Möglichkeit, Bilder von Gesichtern anzugeben, aber nicht die von uns gesuchte. Allerdings haben wir keine Möglichkeit, dem Algorithmus zu »sagen«, wonach wir suchen. Häufig ist die einzige Möglichkeit zur Evaluation eines unüberwachten Algorithmus, das Ergebnis von Hand zu prüfen.

Deshalb werden unüberwachte Algorithmen häufig in der Erkundungsphase eingesetzt, in der ein Data Scientist die Daten besser verstehen möchte, und weniger als Teil eines großen automatisierten Systems. Eine weitere häufige Anwendung von unüberwachten Algorithmen ist die Vorverarbeitung für überwachte Algorithmen. Eine neue Repräsentation der Daten erhöht bisweilen die Lerngenauigkeit des überwachten Algorithmus oder reduziert den Speicher- und Zeitaufwand.

Bevor wir uns den »echten« unüberwachten Algorithmen zuwenden, betrachten wir kurz einige einfache Verfahren zur Vorverarbeitung, die oft nützlich sind. Obwohl Vorverarbeiten und Skalieren oft mit überwachten Lernalgorithmen Hand in Hand gehen, verwenden Skalierungsverfahren keine der überwachten Informationen und sind damit unüberwacht.

## Vorverarbeiten und Skalieren

Im vorigen Kapitel haben wir beobachtet, dass einige Algorithmen wie neuronale Netze und SVMs sehr empfindlich auf die Skalierung der Daten reagieren. Aus diesem Grund ist es üblich, die Merkmale der Daten so anzupassen, dass sie für diese Algorithmen besser geeignet sind. Meistens genügt ein einfacher Schritt zum Umskalieren und Anpassen pro Merkmal. Der folgende Code (Abbildung 3-1) zeigt ein einfaches Beispiel:

In[1]:

```
mglearn.plots.plot_scaling()
```

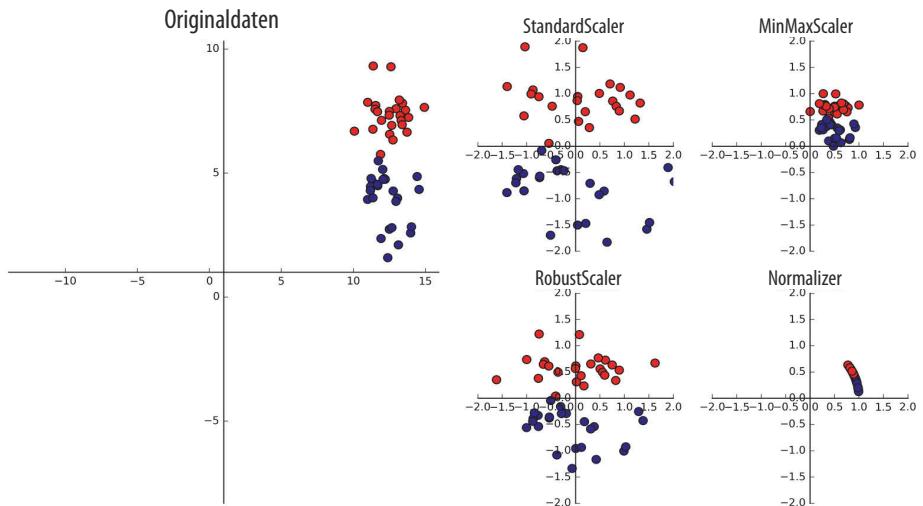


Abbildung 3-1: Unterschiedliche Möglichkeiten zum Umskalieren und Vorverarbeiten eines Datensatzes

## Unterschiedliche Möglichkeiten der Vorverarbeitung

Das erste Diagramm in Abbildung 3-1 zeigt einen synthetischen Datensatz mit zwei Kategorien und zwei Merkmalen. Das erste Merkmal (der Wert auf der x-Achse) liegt zwischen 10 und 15. Das zweite Merkmal (der Wert auf der y-Achse) liegt zwischen 1 und 9.

Die folgenden vier Diagramme zeigen unterschiedliche Möglichkeiten, die Daten zu transformieren, sodass sich standardisierte Wertebereiche ergeben. Der StandardScaler in scikit-learn stellt sicher, dass jedes Merkmal den Mittelwert 0 und die Varianz 1 aufweist, sodass alle Merkmale in der gleichen Größenordnung liegen. Allerdings legt dieses Skalierungsverfahren keine bestimmten Minimal- und Maximalwerte für die Merkmale fest. Der RobustScaler funktioniert so ähnlich wie der StandardScaler, indem er statistische Eigenschaften in der gleichen Größenordnung für jedes Merkmal garantiert. Allerdings verwendet der RobustScaler Median und Quartile,<sup>1</sup> anstelle von Mittelwert und Varianz. Damit ignoriert der RobustScaler Datenpunkte, die sich stark von den übrigen unterscheiden (z. B. Messfehler). Diese abweichenden Datenpunkte werden auch *Ausreißer* genannt und können bei anderen Skalierverfahren zu Problemen führen.

Der MinMaxScaler verschiebt die Daten dagegen so, dass alle Merkmale exakt zwischen 0 und 1 liegen. Bei einem zweidimensionalen Datensatz bedeutet das, dass

<sup>1</sup> Der Median einer Menge von Zahlen ist eine Zahl  $x$ , bei der die Hälfte der Zahlen kleiner als  $x$  und die Hälfte größer als  $x$  sind. Das untere Quartil ist eine Zahl  $x$ , bei der ein Viertel der Zahlen kleiner als  $x$  sind, und das obere Quartil ist eine Zahl  $x$ , bei der ein Viertel der Zahlen größer als  $x$  sind.

sich sämtliche Daten im von der x-Achse zwischen 0 und 1 und der y-Achse zwischen 0 und 1 aufgespannten Quadrat befinden.

Der Normalizer schließlich skaliert auf eine ganz andere Art und Weise um. Jeder Datenpunkt wird so umskaliert, dass der Merkmalsvektor eine euklidische Länge von 1 hat. Anders gesagt, wird der Datenpunkt auf einen Kreis (oder bei mehr Dimensionen auf eine Kugel) mit dem Radius 1 projiziert. Das bedeutet, dass jeder Datenpunkt mit einer anderen Zahl skaliert wird (dem Kehrwert seiner Länge). Diese Art von Normalisierung wird dann verwendet, wenn es nur auf die Richtung (den Winkel) der Daten ankommt, nicht aber auf die Länge des Merkmalsvektors.

## Anwenden von Datentransformationen

Nachdem wir gesehen haben, was die verschiedenen Transformationen tun, werden wir diese in scikit-learn verwenden. Wir verwenden dazu den Datensatz cancer, den wir bereits aus Kapitel 2 kennen. Methoden zur Vorverarbeitung wie die Skalierungsverfahren werden normalerweise vor einem überwachten Lernalgorithmus angewendet. Wir möchten zum Beispiel den SVM-Kernel (SVC) auf den Datensatz cancer anwenden und MinMaxScaler zum Vorverarbeiten der Daten einsetzen. Wir beginnen, indem wir unseren Datensatz laden und in Trainingsdaten und Testdaten unterteilen (wir benötigen separate Trainings- und Testdaten, um anschließend das überwachte Modell zu überprüfen):

**In[2]:**

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
cancer = load_breast_cancer()

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
                                                    random_state=1)
print(X_train.shape)
print(X_test.shape)
```

**Out[2]:**

```
(426, 30)
(143, 30)
```

Zur Erinnerung: Der Datensatz enthält 569 Datenpunkte, jeder mit 30 Messwerten. Wir unterteilen den Datensatz in 426 Datenpunkte als Trainingsdatensatz und 143 Datenpunkte als Testdatensatz.

Wie beim zuvor erstellten überwachten Modell importieren wir zunächst die Klasse für die Vorverarbeitung und erstellen daraus eine Instanz:

**In[3]:**

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
```

Wir passen den Skalierer nun mit der Methode `fit` an die Trainingsdaten an. Beim `MinMaxScaler` berechnet die Methode `fit` Minimal- und Maximalwert jedes Merkmals in den Trainingsdaten. Im Gegensatz zu den Klassifikatoren und Regressionsverfahren aus Kapitel 2 erhält der Skalierer beim Aufruf von `fit` lediglich die Daten aus (`X_train`). `y_train` wird nicht verwendet:

**In[4]:**

```
scaler.fit(X_train)
```

**Out[4]:**

```
MinMaxScaler(copy=True, feature_range=(0, 1))
```

Um die soeben erlernte Transformation anzuwenden – das heißt, die Werte im Trainingsdatensatz tatsächlich zu *skalieren* –, verwenden wir die Methode `transform` des Skalierers. Die Methode `transform` kommt in scikit-learn immer dann zur Anwendung, wenn ein Modell eine neue Repräsentation der Daten erzeugt:

**In[5]:**

```
# Transformieren der Daten
X_train_scaled = scaler.transform(X_train)
# Merkmale des Datensatzes vor und nach Skalieren ausgeben
print("transformierte Abmessungen: {}".format(X_train_scaled.shape))
print("Minimum pro Merkmal vor Skalieren:\n{}".format(X_train.min(axis=0)))
print("Maximum pro Merkmal vor Skalieren:\n{}".format(X_train.max(axis=0)))
print("Minimum pro Merkmal nach Skalieren:\n{}".format(
    X_train_scaled.min(axis=0)))
print("Maximum pro Merkmal nach Skalieren:\n{}".format(
    X_train_scaled.max(axis=0)))
```

**Out[5]:**

```
transformierte Abmessungen: (426, 30)
Minimum pro Merkmal vor Skalieren:
 [ 6.98   9.71  43.79 143.50    0.05    0.02    0.     0.     0.11
  0.05   0.12   0.36   0.76    6.80     0.     0.     0.     0.
  0.01   0.     7.93  12.02   50.41  185.20    0.07    0.03   0.
  0.     0.16   0.06]

Maximum pro Merkmal vor Skalieren:
 [ 28.11   39.28  188.5  2501.0    0.16    0.29    0.43    0.2
  0.300   0.100   2.87   4.88   21.98  542.20    0.03   0.14
  0.400   0.050   0.06   0.03   36.04  49.54   251.20  4254.00
  0.220   0.940   1.17   0.29    0.58   0.15]

Minimum pro Merkmal nach Skalieren:
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.

Maximum pro Merkmal nach Skalieren:
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

Der transformierte Datensatz hat die gleichen Abmessungen wie die ursprünglichen Daten – die Merkmale werden einfach nur verschoben und skaliert. Sie können sehen, dass nun alle Merkmale wie gewünscht zwischen 0 und 1 liegen.

Um das SVM auf die skalierten Daten anzuwenden, müssen wir auch den Testdatensatz transformieren. Auch das erreichen wir durch Aufrufen der Methode `transform`, diesmal auf `X_test`:

**In[6]:**

```
# transformieren der Testdaten
X_test_scaled = scaler.transform(X_test)
# Merkmale des Testdatensatzes nach Skalieren ausgeben
print("Minimum pro Merkmal nach Skalieren:\n{}".format(X_test_scaled.min(axis=0)))
print("Maximum pro Merkmal nach Skalieren:\n{}".format(X_test_scaled.max(axis=0)))
```

**Out[6]:**

```
Minimum pro Merkmal nach Skalieren:
 [ 0.034  0.023  0.031  0.011  0.141  0.044  0.       0.       0.154 -0.006
 -0.001  0.006  0.004  0.001  0.039  0.011  0.       0.       -0.032  0.007
  0.027  0.058  0.02    0.009  0.109  0.026  0.       0.       -0.       -0.002]
Maximum pro Merkmal nach Skalieren:
 [ 0.958  0.815  0.956  0.894  0.811  1.22   0.88   0.933  0.932  1.037
  0.427  0.498  0.441  0.284  0.487  0.739  0.767  0.629  1.337  0.391
  0.896  0.793  0.849  0.745  0.915  1.132  1.07   0.924  1.205  1.631]
```

Es überrascht Sie möglicherweise, dass nach dem Skalieren Minimum und Maximum des Testdatensatzes nicht bei 0 und 1 liegen. Einige der Merkmale liegen sogar außerhalb des Bereichs 0–1! Die Erklärung hierfür ist, dass der `MinMaxScaler` (und alle anderen Skalierer) stets die exakt gleiche Transformation auf Trainings- und Testdaten anwenden. Daher subtrahiert die Methode `transform` immer das Minimum des Trainingsdatensatzes und teilt durch die Spannbreite des Trainingsdatensatzes. Beide Werte können sich von denen des Testdatensatzes unterscheiden.

## Trainings- und Testdaten in gleicher Weise skalieren

Es ist wichtig, die exakt gleiche Transformation auf Trainings- und Testdaten anzuwenden, damit das überwachte Modell auch auf den Testdaten funktioniert. Das folgende Beispiel (Abbildung 3-2) verdeutlicht, was passieren würde, wenn wir stattdessen Minimum und Spannbreite der Testdaten zum Skalieren verwenden.

**In[7]:**

```
from sklearn.datasets import make_blobs
# synthetische Daten erzeugen
X, _ = make_blobs(n_samples=50, centers=5, random_state=4, cluster_std=2)
# Aufteilen in Trainings- und Testdatensatz
X_train, X_test = train_test_split(X, random_state=5, test_size=.1)

# Plotten der Trainings- und Testdaten
fig, axes = plt.subplots(1, 3, figsize=(13, 4))
axes[0].scatter(X_train[:, 0], X_train[:, 1],
                 c=mglearn.cm2(0), label="Trainingsdaten", s=60)
axes[0].scatter(X_test[:, 0], X_test[:, 1], marker='^',
                 c=mglearn.cm2(1), label="Testdaten", s=60)
axes[0].legend(loc='upper left')
```

```

axes[0].set_title("Ursprüngliche Daten")

# Skalieren der Daten mit MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Darstellen der korrekt skalierten Daten
axes[1].scatter(X_train_scaled[:, 0], X_train_scaled[:, 1],
                 c=mglearn.cm2(0), label="Trainingsdaten", s=60)
axes[1].scatter(X_test_scaled[:, 0], X_test_scaled[:, 1], marker='^',
                 c=mglearn.cm2(1), label="Testdaten", s=60)
axes[1].set_title("Skalierte Daten")

# Separates Umskalieren der Testdaten, sodass
# die Testdaten das Minimum 0 und Maximum 1 aufweisen
# TUN SIE DIES NICHT! Nur zur Veranschaulichung.
test_scaler = MinMaxScaler()
test_scaler.fit(X_test)
X_test_scaled_badly = test_scaler.transform(X_test)

# Darstellen der falsch skalierten Daten
axes[2].scatter(X_train_scaled[:, 0], X_train_scaled[:, 1],
                 c=mglearn.cm2(0), label="Trainingsdaten", s=60)
axes[2].scatter(X_test_scaled_badly[:, 0], X_test_scaled_badly[:, 1],
                 marker='^', c=mglearn.cm2(1), label="Testdaten", s=60)
axes[2].set_title("Inkorrekt skalierte Daten")

for ax in axes:
    ax.set_xlabel("Merkmal 0")
    ax.set_ylabel("Merkmal 1")

```

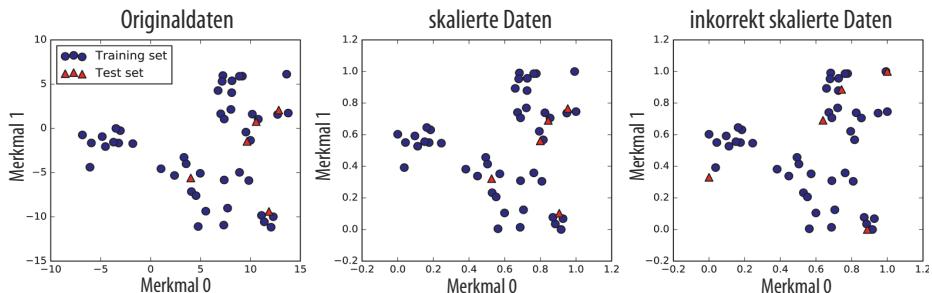


Abbildung 3-2: Auswirkungen des Skalierens des Trainings- und Testdatensatzes auf der linken Seite, einmal gemeinsam (Mitte) und einmal einzeln (rechts)

Das erste Diagramm zeigt einen unskalierten zweidimensionalen Datensatz, bei dem die Trainingsdaten als Kreise und der Testdatensatz als Dreiecke dargestellt sind. Das zweite Diagramm enthält die gleichen mit dem `MinMaxScaler` umskalierten Daten. Wir haben `fit` auf den Trainingsdaten aufgerufen und danach `transform` auf die Trainings- und Testdaten angewendet. Sie sehen, dass die Datensätze im

ersten und zweiten Diagramm identisch aussehen; nur die Skalenstriche auf den Achsen haben sich geändert. Nun liegen alle Merkmale zwischen 0 und 1. Sie können auch sehen, dass die Minimal- und Maximalwerte für den Testdatensatz (die Dreiecke) nicht 0 und 1 sind.

Das dritte Diagramm zeigt, was geschehen würde, wenn wir die Trainingsdaten und Testdaten unabhängig voneinander skalieren. In diesem Fall sind zwar die Minimal- und Maximalwerte für Trainings- und Testdatensatz jeweils 0 und 1, allerdings sieht der Datensatz nun anders aus. Die Datenpunkte zum Testen haben sich inkongruent zum Trainingsdatensatz verschoben, weil sie unterschiedlich skaliert wurden. Wir haben die Anordnung der Daten willkürlich verändert. Das ist natürlich nicht das, was wir erreichen möchten.

Sie können sich den Testdatensatz alternativ auch als einzelnen Punkt vorstellen. Es gibt keine Möglichkeit, einen einzelnen Punkt korrekt zu skalieren, sodass die Anforderungen an den Minimal- und Maximalwert von `MinMaxScaler` erfüllt sind. Aber die Größe des Testdatensatzes sollte unsere Vorgehensweise nicht beeinflussen.

## Abkürzungen und effiziente Alternativen

Meist möchte man ein Modell auf einen Datensatz mit `fit` einstellen und anschließend `transform` anwenden. Dies ist eine sehr häufige Aufgabe, die sich oft effizienter berechnen lässt, als es die Aufrufe von `fit` und `transform` zulassen. Für diese Art von Anwendung besitzen alle Modelle, die die Methode `transform` aufweisen, auch die Methode `fit_transform`. Hier folgt ein Beispiel unter Verwendung von `StandardScaler`:

**In[8]:**

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
# fit und transform nacheinander aufrufen (Aneinanderreihen von Methoden)
X_scaled = scaler.fit(X).transform(X)
# gleiches Ergebnis, aber effizientere Berechnung
X_scaled_d = scaler.fit_transform(X)
```

Obwohl `fit_transform` nicht notwendigerweise für alle Modelle effizienter ist, ist es dennoch eine gängige Praxis, diese Methode für das Transformieren von Trainingsdaten zu verwenden.

## Die Auswirkungen der Vorverarbeitung auf überwachtes Lernen

Kehren wir noch einmal zum Datensatz `cancer` zurück, um die Auswirkung des `MinMaxScaler` auf das Trainieren des SVC zu untersuchen (dies ist eine Alternative zur in Kapitel 2 durchgeführten Skalierung). Wir passen zum Vergleich erst einmal das SVC an die ursprünglichen Daten an:

**In[9]:**

```
from sklearn.svm import SVC

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
                                                    random_state=0)

svm = SVC(C=100)
svm.fit(X_train, y_train)
print("Genauigkeit des Testdatensatzes: {:.2f}".format(svm.score(X_test, y_test)))
```

**Out[9]:**

Genauigkeit des Testdatensatzes: 0.63

Nun skalieren wir die Daten mit dem `MinMaxScaler`, bevor wir das `SVC` trainieren:

**In[10]:**

```
# Vorverarbeitung mit 0-1-Skalierung
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Trainieren einer SVM auf den skalierten Testdaten
svm.fit(X_train_scaled, y_train)

# Auswerten auf dem skalierten Testdatensatz
print("Genauigkeit des skalierten Testdatensatzes: {:.2f}".format(
    svm.score(X_test_scaled, y_test)))
```

**Out[10]:**

Genauigkeit des skalierten Testdatensatzes: 0.97

Wie wir oben gesehen haben, sind die Auswirkungen der Skalierung recht deutlich. Obwohl das Skalieren der Daten keine komplizierte Mathematik ist, ist es gängige Praxis, die Skalierungsmechanismen von `scikit-learn` zu verwenden, anstatt sie selbst zu implementieren, weil sich auch in derart einfachen Berechnungen schnell Fehler einschleichen.

Sie können also einen Algorithmus zur Vorverarbeitung leicht durch einen anderen ersetzen, indem Sie die verwendete Klasse austauschen. Alle Klassen zur Vorverarbeitung stellen die gleiche Schnittstelle mit den Methoden `fit` und `transform` zur Verfügung:

**In[11]:**

```
# Vorverarbeitung mit Skalierung auf den Mittelwert Null und Varianz 1
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Trainieren einer SVM auf den skalierten Trainingsdaten
```

```
svm.fit(X_train_scaled, y_train)

# Auswerten auf dem skalierten Testdatensatz
print("SVM Testgenauigkeit: {:.2f}".format(svm.score(X_test_scaled, y_test)))
```

**Out[11]:**

```
SVM Testgenauigkeit: 0.96
```

Nachdem wir gesehen haben, wie einfache Datentransformationen zum Vorverarbeiten funktionieren, wenden wir uns interessanteren Transformationen mittels unüberwachtem Lernen zu.

## Dimensionsreduktion, Extraktion von Merkmalen und Manifold Learning

Wie zuvor besprochen, gibt es viele mögliche Gründe, Daten mit unüberwachtem Lernen transformieren zu wollen. Die häufigsten Beweggründe sind Daten zu visualisieren, zu komprimieren und eine für die weitere Verarbeitung aufschlussreichere Repräsentation zu finden.

Einer der einfachsten und verbreitetsten Algorithmen für all diese Anwendungen ist die Hauptkomponentenzerlegung. Wir werden außerdem zwei weitere Algorithmen kennenlernen: Nicht-negative-Matrizenfaktorisierung (NMF), ein gebräuchliches Verfahren zum Extrahieren von Merkmalen, sowie t-SNE, ein Verfahren zur Visualisierung mit zweidimensionalen Streudiagrammen.

### Hauptkomponentenzerlegung (PCA)

Hauptkomponentenzerlegung ist eine Methode, die den Datensatz so rotiert, dass die rotierten Merkmalen statistisch unkorreliert sind. Auf diese Rotation folgend wird meist nur eine Teilmenge der neuen Eigenschaften ausgewählt, je nachdem, wie aussagekräftig diese die Daten erklären. Das folgende Beispiel (Abbildung 3-3) verdeutlicht die Auswirkungen der Hauptkomponentenzerlegung anhand eines synthetischen zweidimensionalen Datensatzes:

**In[12]:**

```
mglearn.plots.plot_pca_illustration()
```

Das erste Diagramm (oben links) zeigt die ursprünglichen Datenpunkte zur besseren Unterscheidbarkeit eingefärbt. Der Algorithmus sucht zuerst nach der Richtung mit der größten Varianz, die mit »Component 1« beschriftet ist. Dies ist die Richtung (oder der Vektor), in der die Daten die meiste Information enthalten. Anders gesagt, in dieser Richtung korrelieren die Merkmale am stärksten miteinander. Anschließend sucht der Algorithmus die Richtung mit der meisten Information, die orthogonal (im rechten Winkel) zur ersten Richtung liegt. In zwei Dimensionen gibt es nur eine mögliche Orientierung im rechten Winkel, aber in höher dimensionierten Räumen kann es (unendlich) viele orthogonale Richtungen

geben. Obwohl die zwei Komponenten als Pfeile dargestellt sind, ist es nicht wirklich entscheidend, zu welcher Seite die Pfeilspitze zeigt; wir könnten die erste Komponente auch von der Mitte nach oben links anstatt nach unten rechts zeichnen. Die auf diese Weise gefundenen Richtungen werden *Hauptkomponenten* genannt, da sie die Hauptrichtungen der Varianz in den Daten enthalten. Im Allgemeinen gibt es so viele Hauptkomponenten wie Merkmale.

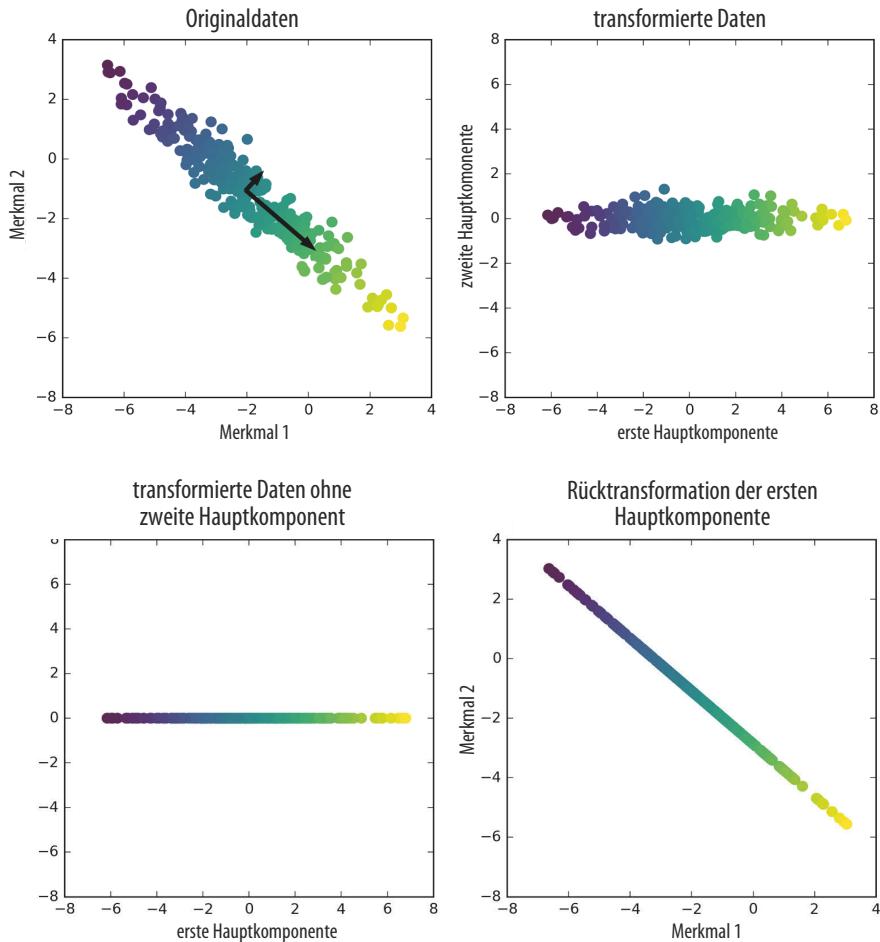


Abbildung 3-3: Datentransformation mittels Hauptkomponentenzerlegung

Das zweite Diagramm (oben rechts) zeigt die gleichen Daten, aber so gedreht, dass die erste Hauptkomponente mit der x-Achse zusammenfällt und die zweite Hauptkomponente mit der y-Achse. Vor der Rotation wurde der Mittelwert von den Daten abgezogen, sodass die rotierten Daten um den Nullpunkt herum liegen. In der von der Hauptkomponentenzerlegung gefundenen Repräsentation nach der Drehung sind die beiden Achsen unkorreliert. Die Korrelationsmatrix der Daten in dieser Darstellung ist also null, abgesehen von der Diagonalen.

Wir können die Hauptkomponentenzerlegung zur Dimensionsreduktion einsetzen und nur einige der Hauptkomponenten verwenden. In diesem Beispiel könnten wir nur die erste Hauptkomponente verwenden, wie im dritten Diagramm Abbildung 3-3 gezeigt (unten links). Damit reduzieren wir die Daten von einem zweidimensionalen zu einem eindimensionalen Datensatz. Es sollte allerdings betont werden, dass wir, anstatt nur eines der ursprünglichen Merkmale auszuwählen, die interessanteste Richtung, die erste Hauptkomponente (im ersten Diagramm von oben links nach unten rechts), gefunden und ausgewählt haben.

Schließlich können wir die Rotation auch wieder rückgängig machen und den Mittelwert wieder zu den Daten hinzufügen. Damit erhalten wir die im letzten Diagramm in Abbildung 3-3 gezeigten Daten. Diese Punkte befinden sich im ursprünglichen Merkmalsraum, aber wir haben nur die Information der ersten Hauptkomponente beibehalten. Diese Art von Transformation wird bisweilen verwendet, um die Auswirkungen von Rauschen zu entfernen oder zu visualisieren, welcher Teil der Information mit der ersten Hauptkomponente erhalten bleibt.

### Anwenden der Hauptkomponentenzerlegung auf den Datensatz cancer zur Visualisierung

Eine der verbreitetsten Anwendungen der Hauptkomponentenzerlegung ist, höher dimensionierte Datensätze zu visualisieren. Wie wir bereits in Kapitel 1 gesehen haben, ist das Erzeugen von Streudiagrammen für Daten mit mehr als zwei Merkmalen schwierig. Mit dem Iris-Datensatz könnten wir einen Paarplot (Abbildung 1-3 in Kapitel 1) generieren, in dem uns alle möglichen Kombinationen von je zwei Merkmalen ein grobes Bild der Daten vermittelten. Aber wenn wir uns den Brustkrebs-Datensatz ansehen möchten, ist sogar ein Paarplot schwierig. Dieser Datensatz enthält 30 Merkmale, was  $30 * 14 = 420$  Streudiagramme ergeben würde! Wir könnten alle diese Diagramme niemals im Detail betrachten, geschweige denn versuchen, sie zu verstehen.

Wir können aber eine weitaus einfachere Visualisierung verwenden – nämlich Histogramme aller Merkmale für die zwei Kategorien, benigne und maligne Tumore, berechnen (Abbildung 3-4):

In[13]:

```
fig, axes = plt.subplots(15, 2, figsize=(10, 20))
malignant = cancer.data[cancer.target == 0]
benign = cancer.data[cancer.target == 1]

ax = axes.ravel()

for i in range(30):
    _, bins = np.histogram(cancer.data[:, i], bins=50)
    ax[i].hist(malignant[:, i], bins=bins, color=mglearn.cm3(0), alpha=.5)
    ax[i].hist(benign[:, i], bins=bins, color=mglearn.cm3(2), alpha=.5)
    ax[i].set_title(cancer.feature_names[i])
    ax[i].set_yticks(())
ax[0].set_xlabel("Größe des Merkmals")
```

```

ax[0].set_ylabel("Häufigkeit")
ax[0].legend(["maligne", "benigne"], loc="best")
fig.tight_layout()

```

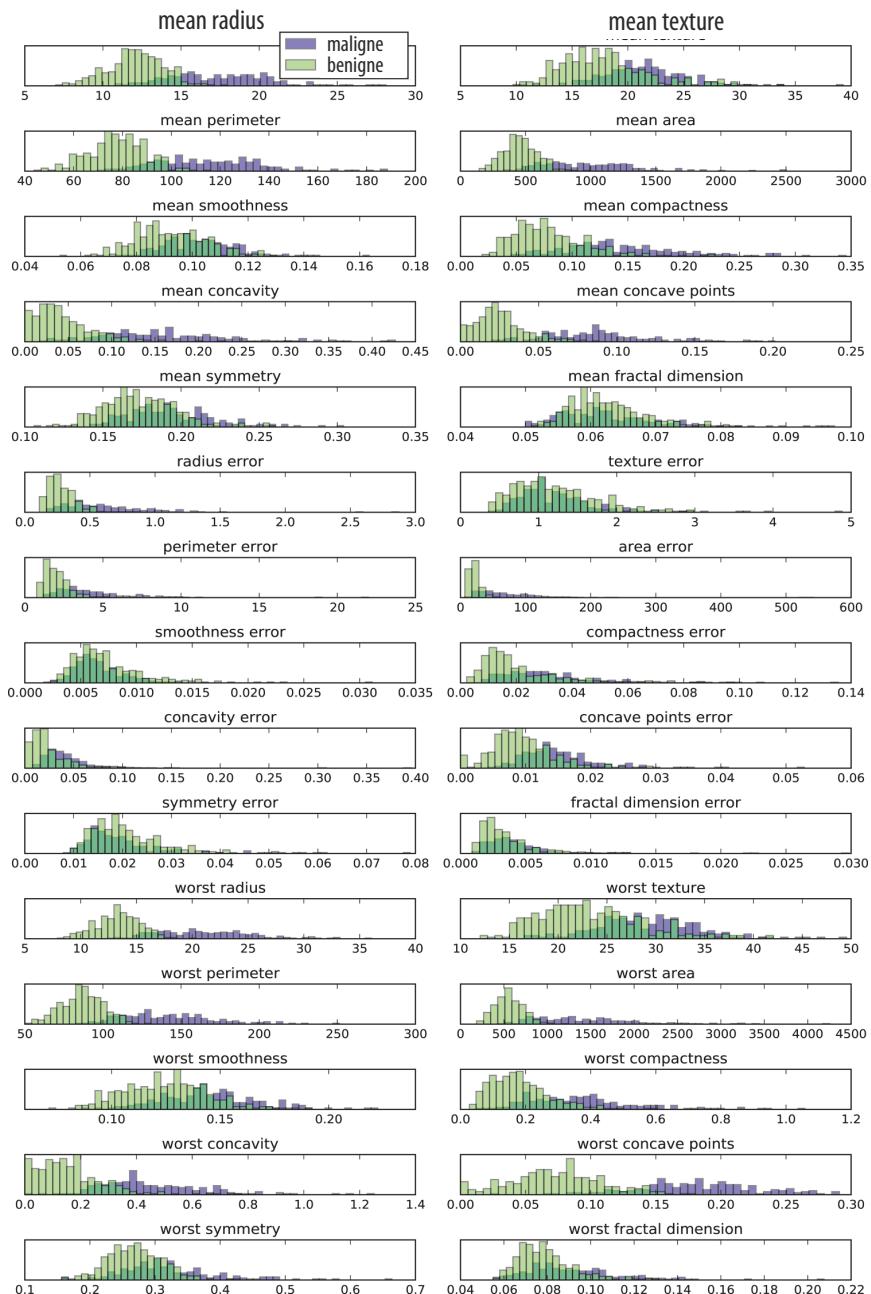


Abbildung 3-4: Histogramme der Merkmale nach Kategorien für den Brustkrebs-Datensatz

Wir erstellen hier ein Histogramm für jedes Merkmal und zählen, wie oft ein Merkmal eines Datenpunktes in einem bestimmten Bereich liegt (eine *Klasse*). In jedem Diagramm liegen zwei Histogramme aufeinander, eines für alle Punkte in der benignen Kategorie (blau) und eines für alle Punkte in der malignen Klasse (rot). Damit erhalten wir einen Eindruck, wie jedes Merkmal über die zwei Kategorien verteilt ist. Wir können so Vermutungen darüber anstellen, welche Merkmale besser zur Unterscheidung zwischen malignen und benignen Stichproben geeignet sind. Zum Beispiel scheint das Merkmal »smoothness error« wenig aussagekräftig, weil die zwei Histogramme weitgehend deckungsgleich sind. Das Merkmal »worst concave points« erscheint hingegen aussagekräftig, weil die Histogramme weiter auseinander liegen.

Dieses Diagramm verrät uns jedoch nichts über die Zusammenhänge zwischen Variablen und wie sie mit den Kategorien zusammenhängen. Mit einer Hauptkomponentenzerlegung können wir die wichtigsten Zusammenhänge erfassen und ein etwas vollständigeres Bild erhalten. Wir können die zwei ersten Hauptkomponenten berechnen und diese neuen zweidimensionalen Daten als ein einziges Streudiagramm visualisieren.

Bevor wir zur Hauptkomponentenzerlegung schreiten, skalieren wir unsere Daten mit dem `StandardScaler`, sodass jedes Merkmal eine einheitliche Varianz besitzt:

**In[14]:**

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()

scaler = StandardScaler()
scaler.fit(cancer.data)
X_scaled = scaler.transform(cancer.data)
```

Das Berechnen einer Hauptkomponentenzerlegung und Anwenden der Transformation ist ähnlich einfach wie eine Transformation zur Vorverarbeitung. Wir erstellen eine Instanz des PCA-Objekts, ermitteln die Hauptkomponenten durch Aufruf der Methode `fit` und wenden anschließend die Rotation und Dimensionsreduktion durch den Aufruf von `transform` an. Mit den Standardeinstellungen rotiert (und verschiebt) PCA die Daten, aber berücksichtigt sämtliche Hauptkomponenten. Um die Dimensionalität der Daten zu reduzieren, müssen wir beim Erstellen des PCA-Objekts angeben, wie viele Hauptkomponenten wir verwenden möchten:

**In[15]:**

```
from sklearn.decomposition import PCA
# verwende die ersten zwei Hauptkomponenten der Daten
pca = PCA(n_components=2)
# passe das PCA-Modell an den Brustkrebs-Datensatz an
pca.fit(X_scaled)

# transformiere die Daten auf die ersten zwei Hauptkomponenten
X_pca = pca.transform(X_scaled)
```

```
print("Ursprüngliche Abmessungen: {}".format(str(X_scaled.shape)))
print("Reduzierte Abmessungen: {}".format(str(X_pca.shape)))
```

Out[15]:

```
Ursprüngliche Abmessungen: (569, 30)
Reduzierte Abmessungen: (569, 2)
```

Wir können nun die ersten zwei Hauptkomponenten plotten (Abbildung 3-5):

In[16]:

```
# erste gegen zweite Hauptkomponente, nach Kategorie eingefärbt
plt.figure(figsize=(8, 8))
mglearn.discrete_scatter(X_pca[:, 0], X_pca[:, 1], cancer.target)
plt.legend(cancer.target_names, loc="best")
plt.gca().set_aspect("equal")
plt.xlabel("Erste Hauptkomponente")
plt.ylabel("Zweite Hauptkomponente")
```

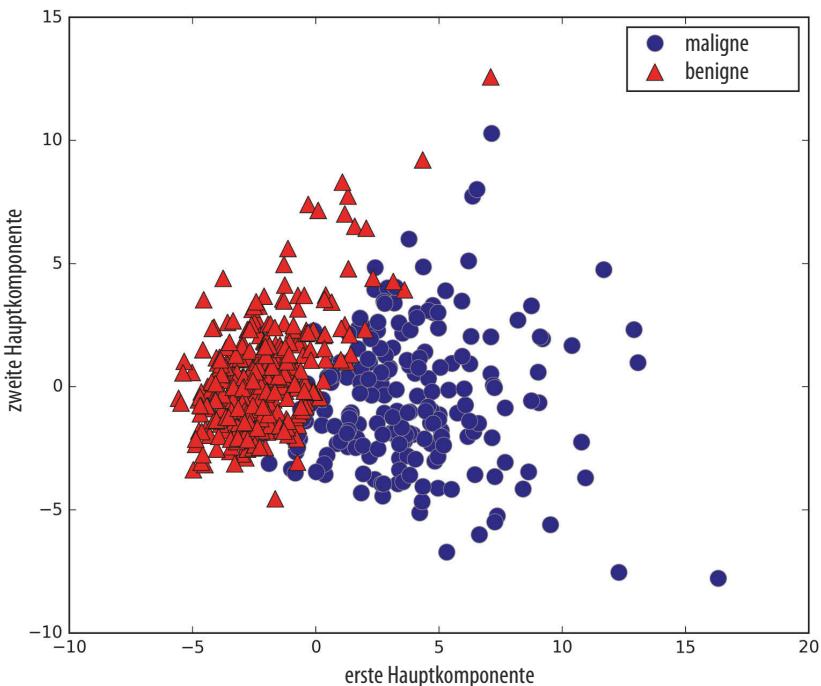


Abbildung 3-5: Zweidimensionales Streudiagramm der ersten zwei Hauptkomponenten für den Brustkrebs-Datensatz

Es sollte betont werden, dass die Hauptkomponentenzerlegung eine unüberwachte Methode ist und dass sie zum Finden der Rotation keinerlei Information aus den Kategorien verwendet. Sie betrachtet lediglich die Korrelationen innerhalb der Daten. Für das hier gezeigte Streudiagramm haben wir die erste gegen die zweite Hauptkomponente aufgetragen und anschließend die Punkte nach Kategorien ein-

gefärbt. Sie können sehen, dass die Kategorien in diesem zweidimensionalen Raum recht gut getrennt werden. Dies bringt uns zu der Annahme, dass sogar ein linearer Klassifikator (der eine Gerade in diesen Raum legt) eine recht gute Unterscheidung zwischen den zwei Kategorien vornehmen könnte. Wir sehen auch, dass die malignen Punkte breiter gestreut sind als die benignen – ein Umstand, den wir bereits aus den Histogrammen in Abbildung 3-4 erahnen konnten.

Ein Nachteil der Hauptkomponentenzerlegung ist, dass die zwei Achsen im Diagramm meist nicht leicht zu interpretieren sind. Die Hauptkomponenten entsprechen Richtungen in den Originaldaten und sind damit aus den ursprünglichen Merkmalen zusammengesetzt. Allerdings sind diese Kombinationen meist sehr komplex, wie wir in Kürze sehen werden. Die Hauptkomponenten selbst befinden sich im Attribut `components_` des berechneten PCA-Objekts:

**In[17]:**

```
print("Abmessungen der Hauptkomponenten: {}".format(pca.components_.shape))
```

**Out[17]:**

```
Abmessungen der Hauptkomponenten: (2, 30)
```

Jede Zeile in `components_` entspricht einer Hauptkomponente, sortiert nach ihrer Bedeutung (die erste Hauptkomponente zuerst usw.). Die Spalten entsprechen den ursprünglichen Merkmalen, in diesem Beispiel »mean radius«, »mean texture«, usw. Sehen wir uns den Inhalt von `components_` einmal an:

**In[18]:**

```
print("Hauptkomponenten:\n{}".format(pca.components_))
```

**Out[18]:**

```
Hauptkomponenten:  
[[ 0.219  0.104  0.228  0.221  0.143  0.239  0.258  0.261  0.138  0.064  
   0.206  0.017  0.211  0.203  0.015  0.17   0.154  0.183  0.042  0.103  
   0.228  0.104  0.237  0.225  0.128  0.21   0.229  0.251  0.123  0.132]  
 [-0.234 -0.06  -0.215 -0.231  0.186  0.152  0.06  -0.035  0.19   0.367  
  -0.106  0.09  -0.089 -0.152  0.204  0.233  0.197  0.13   0.184  0.28  
  -0.22  -0.045 -0.2   -0.219  0.172  0.144  0.098 -0.008  0.142  0.275]]
```

Wir können diese Koeffizienten auch als leichter verständliche Heatmap darstellen (Abbildung 3-6):

**In[19]:**

```
plt.matshow(pca.components_, cmap='viridis')  
plt.yticks([0, 1], ["Erste Komponente", "Zweite Komponente"])  
plt.colorbar()  
plt.xticks(range(len(cancer.feature_names)),  
           cancer.feature_names, rotation=60, ha='left')  
plt.xlabel("Merkmale")  
plt.ylabel("Hauptkomponenten")
```

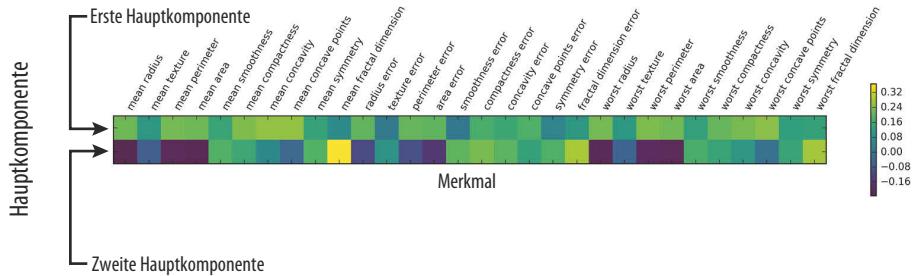


Abbildung 3-6: Heatmap der ersten zwei Hauptkomponenten des Brustkrebs-Datensatzes

Sie können sehen, dass alle Koeffizienten in der ersten Hauptkomponente das gleiche Vorzeichen haben (es ist positiv, aber wie oben erwähnt, spielt die Pfeilrichtung keine Rolle). Das bedeutet, dass grundsätzlich alle Merkmale miteinander korrelieren. Wenn ein Messwert hoch ist, sind die übrigen wahrscheinlich ebenfalls hoch. Die zweite Hauptkomponente weist unterschiedliche Vorzeichen auf, und alle 30 Merkmale fließen in beide Hauptkomponenten ein. Diese Mischung aller Merkmale ist es, die das Interpretieren der Achsen in Abbildung 3-6 so schwierig macht.

### Bildanalyse mit Eigengesichtern

Ein weiteres Anwendungsgebiet der Hauptkomponentenzerlegung ist, wie oben erwähnt, die Extraktion von Merkmalen. Die der Extraktion von Merkmalen zugrunde liegende Idee ist, eine besser zur Analyse geeignete Repräsentation der Daten als die Rohdaten selbst zu finden. Ein erstklassiges Beispiel hierfür sind Bilder. Bilder bestehen aus Pixeln, die meist als Intensitäten der Farben Rot, Grün und Blau (RGB) abgelegt sind. Objekte in Bildern bestehen für gewöhnlich aus Tausenden von Pixeln und ergeben nur gemeinsam einen Sinn.

Wir betrachten ein sehr einfaches Anwendungsbeispiel für die Extraktion von Merkmalen aus Bildern mittels Hauptkomponentenzerlegung, bei dem wir Bilder von Gesichtern aus dem Datensatz »Labeled Faces in the Wild« verwenden. Dieser Datensatz enthält Bilder der Gesichter von Prominenten aus dem Internet, darunter Gesichter von Politikern, Musikern, Schauspielern und Sportlern aus den frühen 2000er-Jahren. Wir verwenden Graustufenversionen dieser Bilder und skalieren sie zur schnelleren Verarbeitung auf eine geringere Größe. Sie können einige der Bilder in Abbildung 3-7 sehen:

In[20]:

```
from sklearn.datasets import fetch_lfw_people
people = fetch_lfw_people(min_faces_per_person=20, resize=0.7)
image_shape = people.images[0].shape

fix, axes = plt.subplots(2, 5, figsize=(15, 8),
                       subplot_kw={'xticks': (), 'yticks': ()})
for target, image, ax in zip(people.target, people.images, axes.ravel()):
    ax.imshow(image)
    ax.set_title(people.target_names[target])
```



Abbildung 3-7: Einige Bilder aus dem Datensatz "Labeled Faces in the Wild"

Es gibt insgesamt 3023 Bilder von 62 unterschiedlichen Personen, jedes davon  $87 \times 65$  Pixel groß.

**In[21]:**

```
print("people.images.shape: {}".format(people.images.shape))
print("Anzahl Kategorien: {}".format(len(people.target_names)))
```

**Out[21]:**

```
people.images.shape: (3023, 87, 65)
Anzahl Kategorien: 62
```

Der Datensatz ist ein wenig unbalanciert, da er eine Menge Bilder von George W. Bush und Colin Powell enthält, wie Sie hier sehen:

**In[22]:**

```
# zählt, wie oft jede Person vorkommt
counts = np.bincount(people.target)
# gibt die Anzahl neben den Namen aus
for i, (count, name) in enumerate(zip(counts, people.target_names)):
    print("{}:{:25} {:1:3}".format(name, count), end='   ')
    if (i + 1) % 3 == 0:
        print()
```

**Out[22]:**

Alejandro Toledo	39	Alvaro Uribe	35
Amelie Mauresmo	21	Andre Agassi	36
Angelina Jolie	20	Arnold Schwarzenegger	42
Atal Bihari Vajpayee	24	Bill Clinton	29
Carlos Menem	21	Colin Powell	236
David Beckham	31	Donald Rumsfeld	121
George W. Bush	530	George Robertson	22

Gerhard Schroeder	109	Gloria Macapagal Arroyo	44
Gray Davis	26	Guillermo Coria	30
Hamid Karzai	22	Hans Blix	39
Hugo Chavez	71	Igor Ivanov	20
[...]		[...]	
Laura Bush	41	Lindsay Davenport	22
Lleyton Hewitt	41	Luiz Inacio Lula da Silva	48
Mahmoud Abbas	29	Megawati Sukarnoputri	33
Michael Bloomberg	20	Naomi Watts	22
Nestor Kirchner	37	Paul Bremer	20
Pete Sampras	22	Recep Tayyip Erdogan	30
Ricardo Lagos	27	Roh Moo-hyun	32
Rudolph Giuliani	26	Saddam Hussein	23
Serena Williams	52	Silvio Berlusconi	33
Tiger Woods	23	Tom Daschle	25
Tom Ridge	33	Tony Blair	144
Vicente Fox	32	Vladimir Putin	49
Winona Ryder	24		

Um weniger verzerrte Daten zu erhalten, verwenden wir nur 50 Bilder jeder Person (ansonsten wäre die Merkmalsextraktion von der Auftrittswahrscheinlichkeit von George W. Bush überwältigt):

### In[23]:

```
mask = np.zeros(people.target.shape, dtype=np.bool)
for target in np.unique(people.target):
    mask[np.where(people.target == target)[0][:50]] = 1

X_people = people.data[mask]
y_people = people.target[mask]

# skaliere die Graustufen auf Werte zwischen 0 und 1
# statt 0 und 255 zugunsten der numerischen Stabilität
X_people = X_people / 255.
```

Eine häufige Aufgabe bei der Gesichtserkennung ist die Klärung der Frage, ob ein neues Gesicht zu einer der bekannten Personen aus einer Datenbank passt. Anwendungen hierfür umfassen Fotosammlungen, soziale Medien und Anwendungen zu Sicherheitszwecken. Eine mögliche Lösung bestünde darin, einen Klassifikator zu bauen, bei dem jede Person einer separaten Kategorie entspricht. Allerdings gibt es normalerweise sehr viele unterschiedliche Personen in Gesichterdatenbanken und sehr wenige Bilder der gleichen Person (also sehr wenige Trainingsbeispiele pro Kategorie). Das erschwert das Trainieren der meisten Klassifikatoren. Außerdem möchte man leicht neue Personen hinzufügen, ohne ein großes Modell erneut zu trainieren.

Eine einfache Lösung wäre ein einfacher nächste-Nachbarn-Klassifikator, der nach dem jeweils ähnlichsten Gesicht zu dem zu klassifizierenden Gesicht sucht. Dieser Klassifikator könnte theoretisch mit einem einzigen Trainingsbeispiel pro Kategorie arbeiten. Betrachten wir, wie gut KNeighborsClassifier bei dieser Aufgabe abschneidet:

**In[24]:**

```
from sklearn.neighbors import KNeighborsClassifier
# teile den Datensatz in Trainings- und Testdaten auf
X_train, X_test, y_train, y_test = train_test_split(
    X_people, y_people, stratify=y_people, random_state=0)
# konstruiere einen KNeighborsClassifier mit einem Nachbarn
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)
print("Testgenauigkeit für 1-NN: {:.2f}".format(knn.score(X_test, y_test)))
```

**Out[24]:**

```
Testgenauigkeit für 1-NN: 0.27
```

Wir erhalten eine Genauigkeit von 26.6 %, was für ein Klassifikationsproblem mit 62 Kategorien gar nicht so schlecht ist (zufälliges Raten würde uns eine Genauigkeit von etwa  $1/62 = 1.5\%$  liefern). Andererseits ist es auch kein großartiges Ergebnis. Wir identifizieren nur bei jedem vierten Versuch die richtige Person.

An dieser Stelle kommt die Hauptkomponentenzerlegung ins Spiel. Den Abstand im Pixelraum zu berechnen, ist ein denkbar schlechter Weg, um die Ähnlichkeit zwischen Gesichtern zu bestimmen. Beim Verwenden einer pixelbasierten Repräsentation zum Vergleich von Bildern vergleichen wir den Graustufenwert jedes einzelnen Pixels mit dem Wert des Pixels in der entsprechenden Position im anderen Bild. Diese Repräsentation unterscheidet sich stark davon, wie Menschen ein Bild mit einem Gesicht interpretieren. Es ist schwierig, die Eigenschaften des Gesichts mit einem derart detaillierten Ansatz zu erfassen. Zum Beispiel führt das Verwenden des Pixelabstandes dazu, dass das Verschieben eines Gesichts um einen Pixel eine drastische Veränderung darstellt. Wir hoffen, dass das Verwenden von Abständen entlang der Hauptkomponenten unsere Genauigkeit verbessert. In diesem Fall verwenden wir bei der Hauptkomponentenzerlegung die Option *Whitening*, die die Hauptkomponenten in die gleiche Größenordnung umskaliert. Dies entspricht dem Anwenden von StandardScaler nach der Transformation. Verwenden wir die Daten aus Abbildung 3-3 noch einmal, entspricht das Whitening nicht nur einer Rotation der Daten, sondern auch einer Umskalierung, sodass der mittlere Teil ein Kreis statt einer Ellipse wird (siehe Abbildung 3-8):

**In[25]:**

```
mglearn.plots.plot_pca_whitening()
```

Wir passen das Objekt PCA an die Trainingsdaten an und extrahieren die ersten 100 Hauptkomponenten. Anschließend transformieren wir die Trainings- und Testdaten:

**In[26]:**

```
pca = PCA(n_components=100, whiten=True, random_state=0).fit(X_train)
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)

print("X_train_pca.shape: {}".format(X_train_pca.shape))
```

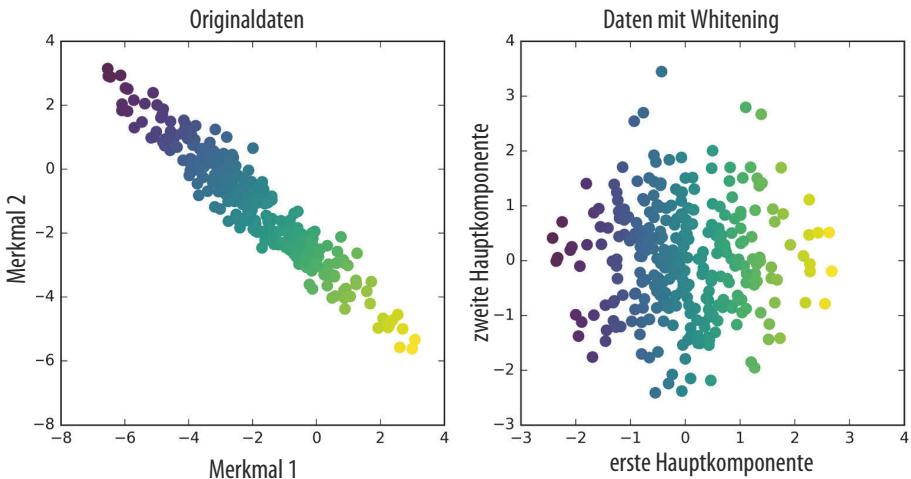


Abbildung 3-8: Transformation der Daten durch eine Hauptkomponentenzerlegung mit Whitening

**Out[26]:**

```
X_train_pca.shape: (1547, 100)
```

Die neuen Daten haben 100 Merkmale, die ersten 100 Hauptkomponenten. Diese neue Repräsentation können wir zum Klassifizieren unserer Bilder mit dem Nächsten-Nachbarn-Verfahren einsetzen.

**In[27]:**

```
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train_pca, y_train)
print("Genauigkeit auf den Testdaten: {:.2f}".format(knn.score(X_test_pca, y_test)))
```

**Out[27]:**

```
Genauigkeit auf den Testdaten: 0.36
```

Wir konnten die Genauigkeit erheblich steigern, von 26.6 % auf 35.7 %. Damit hat sich unsere Vermutung, dass die Hauptkomponenten eine bessere Repräsentation der Daten darstellen, bestätigt.

Bei Bilddaten lassen sich die gefundenen Hauptkomponenten leicht darstellen. Denken wir daran, dass den Komponenten Richtungen in den Eingabedaten entsprechen. Die Eingabedaten sind hier Graustufenbilder mit  $87 \times 65$  Pixeln, daher sind Richtungen innerhalb dieses Raumes ebenfalls Graustufenbilder mit  $87 \times 65$  Pixeln.

Betrachten wir einmal die ersten Hauptkomponenten (Abbildung 3-9):

**In[28]:**

```
print("pca.components_.shape: {}".format(pca.components_.shape))
```

**Out[28]:**

```
pca.components_.shape: (100, 5655)
```

**In[29]:**

```
fix, axes = plt.subplots(3, 5, figsize=(15, 12),
                        subplot_kw={'xticks': (), 'yticks': ()})
for i, (component, ax) in enumerate(zip(pca.components_, axes.ravel())):
    ax.imshow(component.reshape(image_shape),
               cmap='viridis')
    ax.set_title("{}.\u00d6. component".format((i + 1)))
```

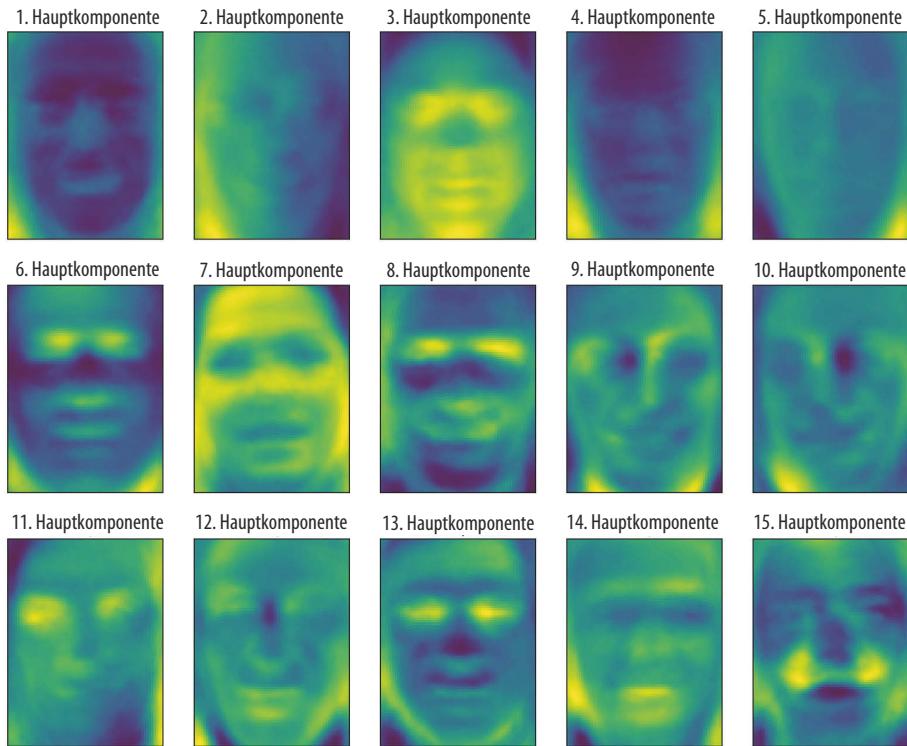


Abbildung 3-9: Komponentenvektoren der ersten 15 Hauptkomponenten auf dem Datensatz von Gesichtern

Obwohl wir mit Sicherheit nicht alle Aspekte dieser Komponenten verstehen können, lässt sich erahnen, welche Eigenschaften der Gesichter von einigen Hauptkomponenten erfasst werden. Die erste Komponente scheint vor allem den Kontrast zwischen Gesicht und Hintergrund abzubilden, die zweite Komponente kodiert Unterschiede in der Belichtung der linken und rechten Gesichtshälfte usw. Auch wenn diese Darstellung etwas bedeutungsvoller als die nackten Pixeldaten ist, sind wir noch immer recht weit davon entfernt, wie ein Mensch Gesichter wahrnimmt. Weil auch die Hauptkomponentenzerlegung auf Pixeln basiert, haben

sowohl die Ausrichtung des Gesichts (die Position von Augen, Kinn und Nase) als auch die Belichtung einen starken Einfluss auf die Ähnlichkeit der Bilder auf Pixel-ebene. Aber ein Mensch würde sicher nicht als Erstes Ausrichtung und Belichtung wahrnehmen. Bittet man Leute, Gesichter nach Ähnlichkeit zu ordnen, werden diese vermutlich Merkmale wie Alter, Geschlecht, Gesichtsausdruck und Frisur verwenden, Merkmale, die schwierig aus den Pixelintensitäten abzuleiten sind. Es ist wichtig, im Hinterkopf zu behalten, dass Algorithmen Daten ganz anders interpretieren, als Menschen es tun würden (insbesondere visuelle Daten, mit denen Menschen gut umgehen können).

Kehren wir noch einmal zu einem Sonderfall der Hauptkomponentenzerlegung zurück. Wir haben die Transformation bei der Hauptkomponentenzerlegung als Rotation der Daten und anschließendem Entfernen der Komponenten mit niedriger Varianz erklärt. Eine weitere nützliche Lesart ist, Koeffizienten zu finden (die neuen Merkmalswerte nach der Rotation), mit denen wir die Datenpunkte als gewichtete Summe der Hauptkomponenten ausdrücken können (siehe Abbildung 3-10).

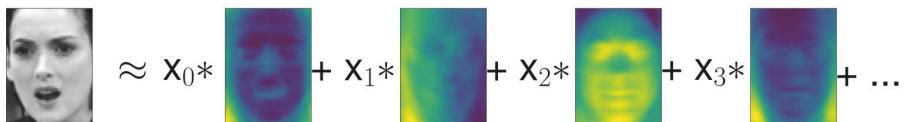


Abbildung 3-10: Schematische Darstellung der Hauptkomponentenzerlegung als Zerlegung eines Bildes in eine gewichtete Summe von Komponenten

Hierbei sind  $x_0$ ,  $x_1$  usw. die Koeffizienten der Hauptkomponenten für den jeweiligen Datenpunkt; anders gesagt, sie sind die Repräsentation des Bildes im rotierten Raum.

Wir können uns die Hauptkomponentenzerlegung auch veranschaulichen, indem wir die Ursprungsdaten mit nur einigen der Komponenten rekonstruieren. In Abbildung 3-3 haben wir im dritten Teil der Abbildung nach Entfernen der zweiten Komponente die Rotation rückgängig gemacht und den Mittelwert wieder hinzugefügt. So erhalten wir Punkte im ursprünglichen Raum, nur ohne die zweite Komponente, wie im letzten Bildteil gezeigt. Wir können eine ähnliche Transformation auch mit den Gesichtern durchführen, indem wir nur einige Hauptkomponenten verwenden und die Daten dann in den ursprünglichen Raum überführen. Diese Rückführung in den ursprünglichen Raum lässt sich mit der Methode `inverse_transform` erreichen. Hier stellen wir die Rekonstruktion einiger Gesichter mit 10, 50, 100 oder 500 Komponenten dar (Abbildung 3-11):

**In[30]:**

```
mglearn.plots.plot_pca_faces(X_train, X_test, image_shape)
```



*Abbildung 3-11: Rekonstruktion von drei Bildern von Gesichtern mit einer steigenden Anzahl von Hauptkomponenten*

Sie können sehen, dass wir mit den ersten zehn Hauptkomponenten lediglich die Grundeigenschaften des Bildes wie die Orientierung des Gesichts und die Belichtung erfassen. Mit mehr und mehr Hauptkomponenten werden zunehmend mehr Details in den Bildern sichtbar. Dabei nehmen wir dementsprechend mehr Bestandteile in die Summe in Abbildung 3-10 auf. Würden wir so viele Hauptkomponenten verwenden, wie es Pixel gibt, würden wir nach der Rotation keine Information verlieren und könnten die Bilder perfekt rekonstruieren.

Wir können mit der Hauptkomponentenzerlegung auch alle Gesichter im Datensatz als Streudiagramm der ersten zwei Hauptkomponenten darstellen (Abbildung 3-12), wobei die Kategorien ähnlich wie beim Datensatz cancer den Personen im Bild entsprechen:

**In[31]:**

```
mglearn.discrete_scatter(X_train_pca[:, 0], X_train_pca[:, 1], y_train)
plt.xlabel("Erste Hauptkomponente")
plt.ylabel("Zweite Hauptkomponente")
```

Wenn wir nur die ersten zwei Hauptkomponenten verwenden, ist der gesamte Datensatz einfach nur ein großer Haufen. Es ist kein Unterschied zwischen den

Kategorien sichtbar. Dies ist nicht besonders überraschend, da wir sogar bei zehn Komponenten, wie in Abbildung 3-11 gezeigt, nur sehr grobe Charakteristiken der Gesichter abbilden.

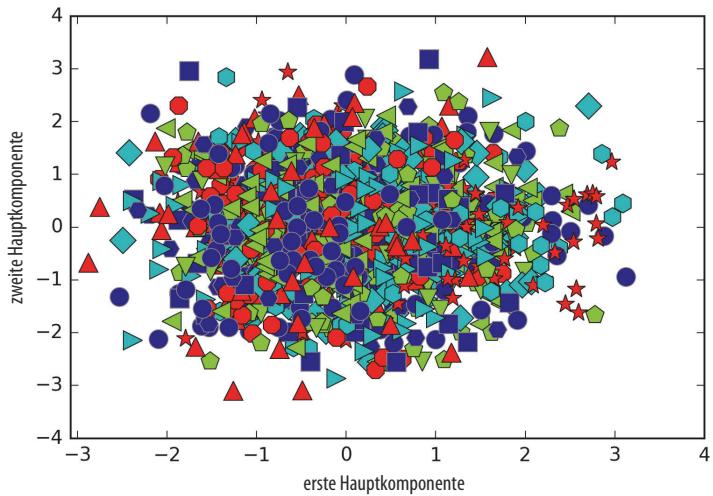


Abbildung 3-12: Streudiagramm der ersten zwei Hauptkomponenten für den Datensatz von Gesichtern (in Abbildung 3-5 befindet sich ein entsprechendes Diagramm für den Krebs-Datensatz)

## Nicht-negative-Matrix-Faktorisierung (NMF)

Nicht-negative-Matrix-Faktorisierung ist ein weiterer unüberwachter Lernalgorithmus, mit dem sich nützliche Merkmale ermitteln lassen. Er funktioniert so ähnlich wie die Hauptkomponentenzerlegung und lässt sich auch zur Dimensionsreduktion einsetzen. Wie bei der Hauptkomponentenzerlegung versuchen wir, jeden Datenpunkt als gewichtete Summe einiger Komponenten auszudrücken, wie in Abbildung 3-10 veranschaulicht. Während wir uns aber bei der Hauptkomponentenzerlegung für orthogonale Komponenten interessierten, die die Varianz der Daten möglichst genau erklärten, möchten wir bei der NMF nicht-negative Komponenten und Koeffizienten erhalten; das bedeutet, wir möchten sowohl Komponenten als auch Koeffizienten erhalten, die größer oder gleich null sind. Konsequenterweise lässt sich diese Methode nur anwenden, wenn keines der Merkmale negativ ist, weil eine nicht-negative Summe nicht-negativer Komponenten niemals negativ werden kann.

Der Prozess der Datenzerlegung in eine nicht-negative gewichtete Summe ist besonders hilfreich, falls Daten aus mehreren unabhängigen Quellen addiert (oder zusammengelegt) werden, wie bei einer Tonspur mit mehreren sprechenden Personen oder bei Musik mit mehreren Instrumenten. In solchen Fällen kann NMF die ursprünglichen Komponenten erkennen, aus denen die zusammengelegten Daten bestehen. Insgesamt ergeben sich bei der NMF leichter interpretierbare Kompo-

nenten als bei der PCA, da negative Komponenten und Koeffizienten schwer interpretierbare Auslöschungseffekte nach sich ziehen. Beispielsweise enthalten die Eigengesichter in Abbildung 3-9 sowohl positive als auch negative Teile, und das Vorzeichen ist, wie bereits bei der Beschreibung der Hauptkomponentenzerlegung erwähnt, willkürlich. Bevor wir NMF auf die Gesichter anwenden, kehren wir noch einmal zu den synthetischen Daten zurück.

### Anwenden von NMF auf synthetische Daten

Im Gegensatz zur Hauptkomponentenzerlegung müssen wir bei der NMF sicherstellen, dass unsere Daten positiv sind. Es kommt also bei der NMF darauf an, wie die Daten relativ zum Koordinatenursprung  $(0, 0)$  liegen. Sie können sich daher die bei der NMF gefundenen nicht-negativen Komponenten als Vektoren vom Ursprung  $(0, 0)$  zu den Daten vorstellen.

Das folgende Beispiel (Abbildung 3-13) zeigt das Ergebnis einer NMF auf den synthetischen zweidimensionalen Daten:

In[32]:

```
mglearn.plots.plot_nmf_illustration()
```

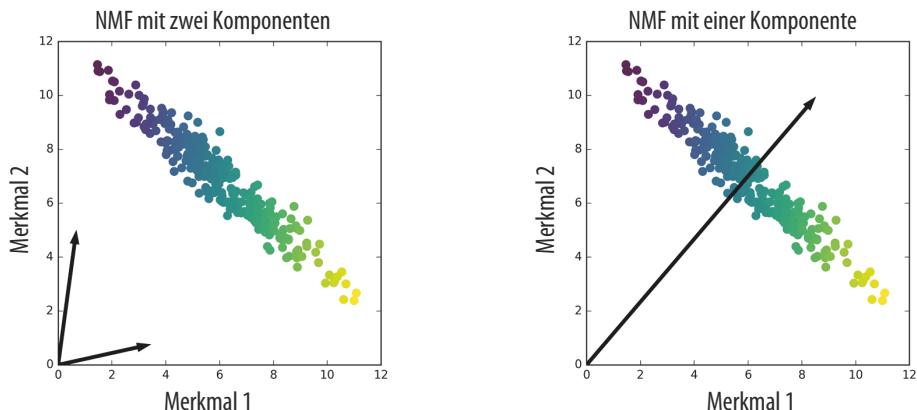


Abbildung 3-13: Durch Nicht-negative-Matrix-Faktorisierung ermittelte Komponenten, mit zwei Komponenten (links) und einer Komponente (rechts)

Bei der NMF mit zwei Komponenten auf der linken Seite wird klar, dass sich alle Punkte im Datensatz durch positive Kombination der zwei Komponenten ausdrücken lassen. Wenn es genug Komponenten gibt, um die Daten perfekt zu rekonstruieren (so viele, wie es Merkmale gibt), wählt der Algorithmus Richtungsvektoren, die auf die äußersten Datenpunkte zeigen.

Wenn wir aber nur eine einzelne Komponente verwenden, ermittelt die NMF eine Komponente, die auf den Mittelwert gerichtet ist, da diese die Daten am besten erklärt. Sie können erkennen, dass im Gegensatz zur Hauptkomponentenzerlegung eine Änderung der Komponentenzahl nicht nur einige Vektoren entfernt, son-

dern auch einen völlig anderen Satz von Komponenten erzeugt! Die Komponenten sind bei der NMF also nicht geordnet, es gibt keine »erste nicht-negative Komponente«: Alle Komponenten sind gleichberechtigt.

NMF verwendet zufällige Startwerte, was in Abhängigkeit vom Seed-Wert des Zufallsgenerators zu unterschiedlichen Ergebnissen führen kann. In einfachen Fällen wie dem synthetischen Datensatz mit zwei Komponenten, bei dem sich alle Datenpunkte perfekt abbilden lassen, sind die Auswirkungen des Zufalls gering (auch wenn sich die Reihenfolge oder Länge der Komponenten ändern könnte). In komplexeren Situationen sind auch weitreichende Änderungen möglich.

### Anwenden von NMF auf Bilder von Gesichtern

Lassen Sie uns NMF auf den bereits verwendeten Datensatz »Labeled Faces in the Wild« anwenden. Der wichtigste Parameter bei der NMF ist die Anzahl zu ermittelnder Komponenten. Normalerweise ist diese Zahl geringer als die Anzahl der Merkmale in den Eingabedaten (andernfalls könnte man die Daten auch als eine Komponente pro Pixel darstellen). Als Erstes untersuchen wir, wie sich die Anzahl der Komponenten auf die Rekonstruierbarkeit der Daten auswirkt (Abbildung 3-14):

In[33]:

```
mglearn.plots.plot_nmf_faces(X_train, X_test, image_shape)
```



Abbildung 3-14: Rekonstruktion dreier Bilder von Gesichtern mit einer steigenden Zahl von mittels NMF gefundener Komponenten

Die Qualität der rücktransformierten Daten ist vergleichbar mit der der Hauptkomponentenzerlegung oder ein wenig schlechter. Das ist zu erwarten, da die Hauptkomponentenzerlegung die zur Rekonstruktion optimalen Richtungsvektoren ermittelt. Man verwendet NMF für gewöhnlich nicht zur Rekonstruktion oder zum Abbilden von Daten, sondern zum Finden interessanter Muster.

Um einen ersten Blick auf die Daten zu werfen, ermitteln wir einige Komponenten (sagen wir 15). Das Ergebnis sehen Sie in Abbildung 3-15:

In[34]:

```
from sklearn.decomposition import NMF
nmf = NMF(n_components=15, random_state=0)
nmf.fit(X_train)
X_train_nmf = nmf.transform(X_train)
X_test_nmf = nmf.transform(X_test)

fix, axes = plt.subplots(3, 5, figsize=(15, 12),
                       subplot_kw={'xticks': (), 'yticks': ()})
for i, (component, ax) in enumerate(zip(nmf.components_, axes.ravel())):
    ax.imshow(component.reshape(image_shape))
    ax.set_title("{}.\ Komponente".format(i))
```



Abbildung 3-15: Die ersten 15 mit NMF gefundenen Komponenten aus dem Gesichter-Datensatz

Diese Komponenten sind alle positiv und stellen somit viel eher prototypische Gesichter dar als die in Abbildung 3-9 über die Hauptkomponentenzerlegung ermittelten Komponenten. Beispielsweise sieht man deutlich, dass Komponente 3 ein etwas nach rechts gedrehtes Gesicht darstellt, während Komponente 7 ein etwas nach links gedrehtes Gesicht zeigt. Betrachten wir die Bilder, bei denen diese Komponenten besonders ausgeprägt sind in Abbildung 3-16 und Abbildung 3-17:

**In[35]:**

```
compn = 3
# sortiere nach der 3. Komponente und zeichne die ersten 10 Bilder
inds = np.argsort(X_train_nmf[:, compn])[::-1]
fig, axes = plt.subplots(2, 5, figsize=(15, 8),
                       subplot_kw={'xticks': (), 'yticks': ()})
for i, (ind, ax) in enumerate(zip(inds, axes.ravel())):
    ax.imshow(X_train[ind].reshape(image_shape))

compn = 7
# sortiere nach der 7. Komponente und zeichne die ersten 10 Bilder
inds = np.argsort(X_train_nmf[:, compn])[::-1]
fig, axes = plt.subplots(2, 5, figsize=(15, 8),
                       subplot_kw={'xticks': (), 'yticks': ()})
for i, (ind, ax) in enumerate(zip(inds, axes.ravel())):
    ax.imshow(X_train[ind].reshape(image_shape))
```



Abbildung 3-16: Gesichter mit ausgeprägten Koeffizienten bei Komponente 3

Wie erwartet, blicken die Gesichter mit hohem Koeffizienten bei Komponente 3 nach rechts (Abbildung 3-16), und Gesichter mit hohem Koeffizienten bei Komponente 7 blicken nach links (Abbildung 3-17). Wie bereits erwähnt, funktioniert die Erkennung von Mustern wie diesen besonders gut bei additiven Daten wie Audiodaten, Genexpressionsdaten und Texten. Gehen wir noch ein fiktives Datenbeispiel durch, um zu sehen, wie das in der Praxis aussieht.



Abbildung 3-17: Gesichter mit ausgeprägten Koeffizienten bei Komponente 7

Interessieren wir uns beispielsweise für ein Signal, das eine Kombination von drei verschiedenen Quellen ist (Abbildung 3-18):

In[36]:

```
S = mglearn.datasets.make_signals()
plt.figure(figsize=(6, 1))
plt.plot(S, '-')
plt.xlabel("Zeit")
plt.ylabel("Signal")
```

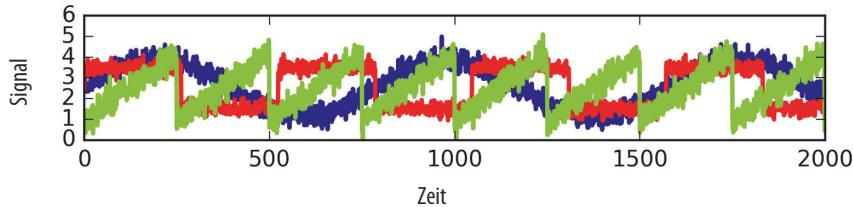


Abbildung 3-18: Ursprüngliche Signaldaten

Leider können wir die ursprünglichen Signale nicht sehen, sondern nur die additive Zusammensetzung der drei. Wir möchten das zusammengesetzte Signal in die ursprünglichen Komponenten zerlegen. Wir nehmen an, dass wir viele Möglichkeiten haben, das zusammengesetzte Signal zu beobachten (z. B. 100 Messgeräte). Jedes Messgerät liefert dabei eine Serie von Messwerten:

In[37]:

```
# vermische Daten zu einem 100-dimensionalen Zustand
A = np.random.RandomState(0).uniform(size=(100, 3))
X = np.dot(S, A.T)
print("Abmessungen der Messwerte: {}".format(X.shape))
```

**Out[37]:**

```
Abmessungen der Messwerte: (2000, 100)
```

Wir können die drei Signale mit NMF ermitteln:

**In[38]:**

```
nmf = NMF(n_components=3, random_state=42)
S_ = nmf.fit_transform(X)
print("Ermittelte Abmessungen des Signals: {}".format(S_.shape))
```

**Out[38]:**

```
Ermittelte Abmessungen des Signals: (2000, 3)
```

Zum Vergleich können wir auch noch die Hauptkomponentenzerlegung verwenden:

**In[39]:**

```
pca = PCA(n_components=3)
H = pca.fit_transform(X)
```

Abbildung 3-19 zeigt die mittels NMF und Hauptkomponentenzerlegung erhaltene Signalaktivität:

**In[40]:**

```
models = [X, S, S_, H]
names = ['Messwerte (erste drei Messungen)',
         'Originaldaten',
         'mit NMF ermittelte Signale',
         'mit PCA ermittelte Signale']

fig, axes = plt.subplots(4, figsize=(8, 4), gridspec_kw={'hspace': .5},
                       subplot_kw={'xticks': (), 'yticks': ()})

for model, name, ax in zip(models, names, axes):
    ax.set_title(name)
    ax.plot(model[:, :3], '-')
```

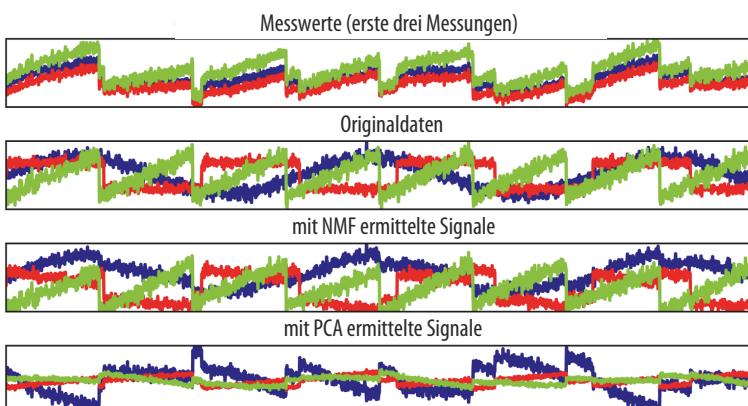


Abbildung 3-19: Zerlegen zusammengesetzter Signale mittels NMF und PCA

Die Abbildung enthält zum Vergleich drei der 100 zusammengesetzten Messungen für X. Wie Sie sehen, hat NMF die ursprünglichen Signale recht gut erkannt, wohingegen die Hauptkomponentenzerlegung gescheitert ist und stattdessen die Streuung der Daten über die erste Hauptkomponente abgebildet hat. Denken Sie daran, dass die mittels NMF gefundenen Komponenten ungeordnet sind. In diesem Beispiel ist die Reihenfolge der NMF-Komponenten die gleiche wie im ursprünglichen Signal (entsprechend den Schattierungen im Diagramm), aber das ist reiner Zufall.

Es gibt viele weitere Algorithmen, die sich wie PCA und NMF zum Zerlegen von Datenpunkten in eine gewichtete Summe festgelegter Komponenten eignen. Es sprengt den Rahmen, diese alle hier zu besprechen, und bei den Bedingungen zu den Komponenten und Koeffizienten spielt die Wahrscheinlichkeitstheorie eine gewisse Rolle. Falls Sie sich für diese Art der Mustererkennung interessieren, empfehlen wir Ihnen, sich mit den Abschnitten zu Unabhängigkeitsanalyse (ICA), Faktorenanalyse (FA) und Sparse Coding (Dictionary Learning) in der Dokumentation zu scikit\_learn zu beschäftigen, die sie alle auf der Seite decomposition methods (<http://scikit-learn.org/stable/modules/decomposition.html>) finden.

## Manifold Learning mit t-SNE

Auch wenn die Hauptkomponentenzerlegung oft ein guter Anfang bei der Datentransformation zum Visualisieren als Streudiagramm ist, begrenzen die Eigenschaften der Methode (Anwenden einer Rotation und anschließendes Weglassen von Richtungsvektoren) ihre Nützlichkeit, was wir bei Streudiagramm des Datensatzes »Labeled Faces in the Wild« beobachten konnten. Es gibt eine Klasse Algorithmen zur Visualisierung, genannt *Manifold Learning-Algorithmen*, die wesentlich komplexere Zuordnungen erlaubt und die Visualisierbarkeit verbessert. Der t-SNE-Algorithmus ist besonders nützlich.

Manifold Learning-Algorithmen zielen vor allem auf die Visualisierung ab und generieren daher selten mehr als zwei Merkmale. Einige wie t-SNE berechnen eine neue Repräsentation der Trainingsdaten, erlauben aber keine Transformation zusätzlicher Daten. Das bedeutet, dass diese Algorithmen sich nicht auf einen Testdatensatz anwenden lassen: Sie können nur die Daten transformieren, mit denen sie trainiert wurden. Manifold Learning ist bei der erkundenden Datenanalyse nützlich, wird aber nur selten eingesetzt, um auf ein überwachtes Lernmodell hinzuarbeiten. Die t-SNE zugrunde liegende Idee ist, eine zweidimensionale Repräsentation der Daten zu erhalten, bei der die Abstände zwischen den Punkten so gut wie möglich erhalten bleiben. t-SNE beginnt mit einer zufälligen zweidimensionalen Anordnung der Datenpunkte und versucht anschließend, im ursprünglichen Merkmalsraum nah beieinanderliegende Punkte näher aneinanderzurücken und im Merkmalsraum weiter auseinanderliegende Punkte weiter voneinander weg zurückzurücken. t-SNE legt mehr Gewicht auf nah beieinanderliegende Punkte, anstatt die Abstände zwischen weit entfernt liegenden Punkten zu erhalten. Anders gesagt, versucht das Verfahren, Informationen über Nachbarschaftsverhältnisse zwischen Punkten zu erhalten.

Wir werden den Manifold Learning-Algorithmus t-SNE auf einen in scikit-learn enthaltenen Datensatz handgeschriebener Ziffern anwenden.<sup>2</sup> Jeder Datenpunkt in diesem Datensatz ist ein  $8 \times 8$  Pixel großes Bild in Graustufen, das eine handgeschriebene Ziffer zwischen 0 und 9 darstellt. Abbildung 3-20 zeigt ein Beispielbild aus jeder Kategorie:

In[41]:

```
from sklearn.datasets import load_digits
digits = load_digits()

fig, axes = plt.subplots(2, 5, figsize=(10, 5),
                       subplot_kw={'xticks':(), 'yticks': ()})
for ax, img in zip(axes.ravel(), digits.images):
    ax.imshow(img)
```

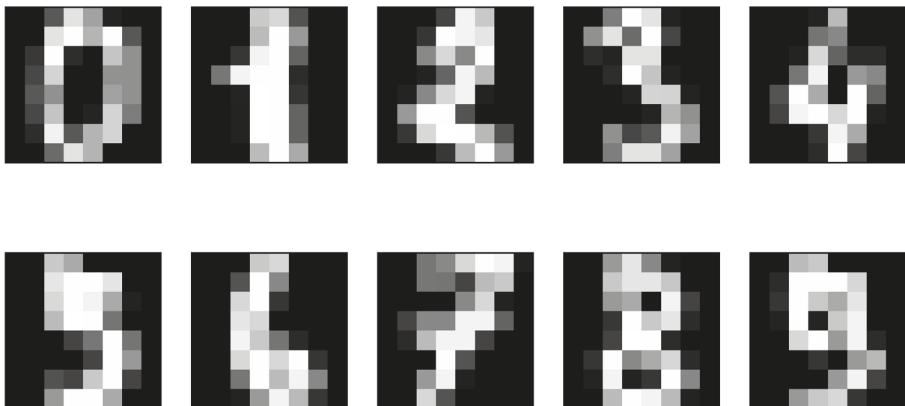


Abbildung 3-20: Beispielbilder aus dem Datensatz "digits"

Wir wenden die Hauptkomponentenzerlegung an, um diese Daten auf zwei Dimensionen reduziert darzustellen. Wir plotten die ersten zwei Hauptkomponenten und stellen jeden Punkt als Ziffer entsprechend seiner Kategorie dar (siehe Abbildung 3-21):

In[42]:

```
# erstelle ein Modell zur Hauptkomponentenzerlegung
pca = PCA(n_components=2)
pca.fit(digits.data)

# transformiere die Ziffern auf die ersten zwei Hauptkomponenten
digits_pca = pca.transform(digits.data)
colors = ["#476A2A", "#7851B8", "#BD3430", "#4A2D4E", "#875525",
          "#A83683", "#4E655E", "#853541", "#3A3120", "#535D8E"]
plt.figure(figsize=(10, 10))
plt.xlim(digits_pca[:, 0].min(), digits_pca[:, 0].max())
plt.ylim(digits_pca[:, 1].min(), digits_pca[:, 1].max())
for i in range(len(digits.data)):
```

---

<sup>2</sup> Dieser ist nicht mit dem weitaus größeren MNIST-Datensatz zu verwechseln.

```

# hier werden die Ziffern als Text anstatt mit scatter gezeichnet
plt.text(digits_pca[i, 0], digits_pca[i, 1], str(digits.target[i]),
         color = colors[digits.target[i]],
         fontdict={'weight': 'bold', 'size': 9})
plt.xlabel("Erste Hauptkomponente")
plt.ylabel("Zweite Hauptkomponente")

```

Wir haben hier die tatsächlichen Ziffernkategorien zur Beschriftung verwendet, um anzusehen, welche Kategorie sich wo befindet. Mit den ersten zwei Hauptkomponenten lassen sich die Ziffern Null, Sechs und Vier recht gut voneinander trennen, auch wenn sie sich noch ein wenig überlappen. Die meisten der übrigen Ziffern überlappen einander deutlich.

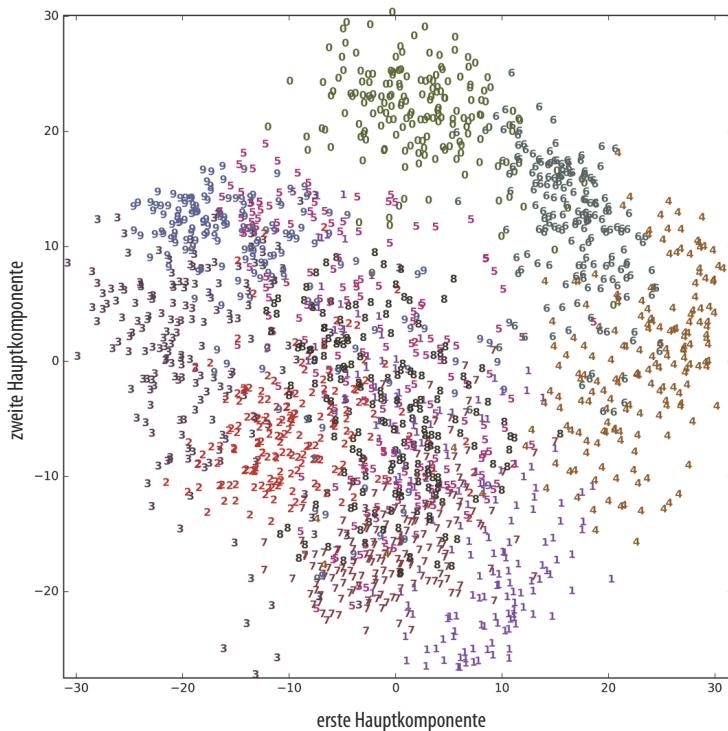


Abbildung 3-21: Streudiagramm für den Datensatz "digits" mit den ersten zwei Hauptkomponenten

Wenden wir nun t-SNE auf den gleichen Datensatz an und vergleichen wir das Ergebnis. Weil t-SNE keine Transformation zusätzlicher Daten unterstützt, besitzt die Klasse TSNE keine Methode transform. Stattdessen können wir die Methode fit\_transform aufrufen, mit der wir das Modell konstruieren und unmittelbar einen transformierten Datensatz erhalten (siehe Abbildung 3-22):

In[43]:

```

from sklearn.manifold import TSNE
tsne = TSNE(random_state=42)

```

```
# verwende fit_transform anstelle von fit, da TSNE keine Methode transform kennt
digits_tsne = tsne.fit_transform(digits.data)
```

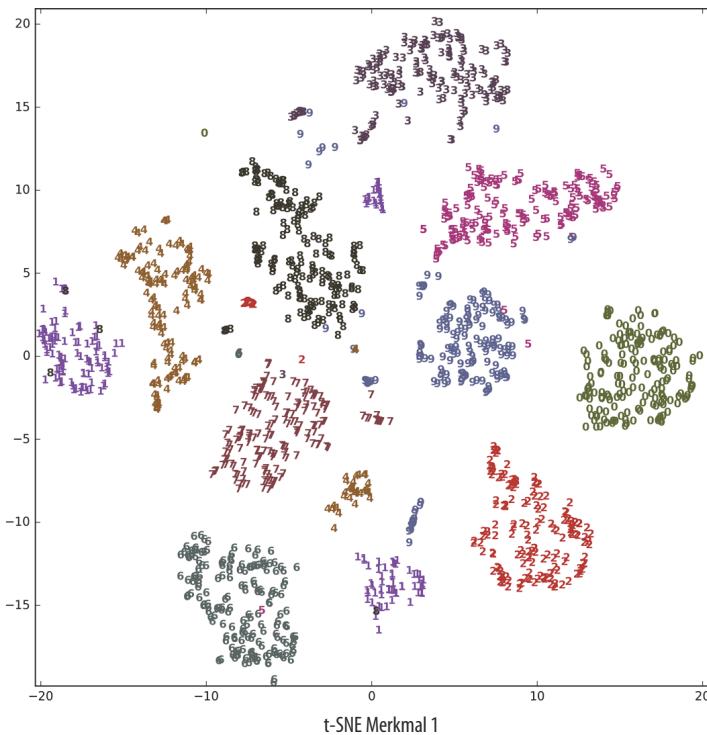


Abbildung 3-22: Streudiagramm für den Datensatz "digits" mit zwei von t-SNE gefundenen Komponenten

In[44]:

```
plt.figure(figsize=(10, 10))
plt.xlim(digits_tsne[:, 0].min(), digits_tsne[:, 0].max() + 1)
plt.ylim(digits_tsne[:, 1].min(), digits_tsne[:, 1].max() + 1)
for i in range(len(digits.data)):
    # hier werden die Ziffern als Text anstatt mit scatter gezeichnet
    plt.text(digits_tsne[i, 0], digits_tsne[i, 1], str(digits.target[i]),
            color = colors[digits.target[i]],
            fontdict={'weight': 'bold', 'size': 9})
plt.xlabel("t-SNE Merkmal 0")
plt.xlabel("t-SNE Merkmal 1")
```

Das Ergebnis von t-SNE ist recht beeindruckend. Alle Kategorien sind klar voneinander abgegrenzt. Die Einsen und Neunen sind ein wenig verstreut, aber die meisten Kategorien bilden eine einzelne dichte Gruppe. Denken Sie daran, dass diese Methode nichts über die Zuordnung der Kategorien weiß: Sie arbeitet völlig unüberwacht. Dennoch findet sie auf der Basis der Abstände im ursprünglichen Raum eine zweidimensionale Repräsentation der Daten, die die Kategorien klar voneinander abgrenzt.

Der t-SNE-Algorithmus besitzt einige Parameter zur Optimierung, auch wenn er häufig mit den Standardeinstellungen funktioniert. Sie können versuchen, mit den Werten für perplexity und early\_exaggeration herumzuspielen, aber die Auswirkungen sind üblicherweise gering.

## Clusteranalyse

Wie weiter oben besprochen, geht es bei der *Clusteranalyse* um das Einteilen eines Datensatzes in Cluster genannte Gruppen. Das Ziel ist, die Daten so aufzuteilen, dass die Datenpunkte innerhalb eines Clusters sehr ähnlich zueinander und Punkte in unterschiedlichen Clustern verschieden sind. Ähnlich wie bei Algorithmen zur Klassifikation ordnen Clustering-Algorithmen jedem Datenpunkt eine Zahl zu (oder treffen eine Vorhersage), die anzeigt, zu welchem Cluster ein bestimmter Punkt gehört.

### k-Means-Clustering

*k*-Means-Clustering ist einer der einfachsten und am häufigsten eingesetzten Clustering-Algorithmen. Er versucht *Mittelpunkte von Clustern* zu finden, die bestimmte Regionen innerhalb der Daten repräsentieren. Der Algorithmus wechselt zwischen zwei Schritten hin und her: jeden Datenpunkt dem nächstgelegenen Cluster zuzuordnen und dann jeden Clustermittelpunkt auf den Mittelwert der ihm zugeordneten Datenpunkte zu setzen. Der Algorithmus ist beendet, sobald sich die Zuordnung von Datenpunkten zu Clustern nicht mehr ändert.

Das folgende Beispiel (Abbildung 3-23) verdeutlicht den Algorithmus anhand eines synthetischen Datensatzes:

In[45]:

```
mglearn.plots.plot_kmeans_algorithm()
```

Die Cluster-Mittelpunkte sind als Dreiecke dargestellt, die Datenpunkte als Kreise. Die Farben zeigen die Zugehörigkeit zu den Clustern an. Wir haben drei zu findende Cluster vorgegeben, daher erklärt der Algorithmus drei zufällig ausgewählte Datenpunkte zu Clustermittelpunkten (siehe »Initialization«). Anschließend startet der iterative Algorithmus. Zuerst wird jeder Datenpunkt dem nächstgelegenen Clustermittelpunkt zugeordnet (siehe »Assign Points (1)«). Anschließend werden die Clustermittelpunkte entsprechend der Mittelwerte der ihnen zugeordneten Punkte aktualisiert (siehe »Recompute Centers (1)«). Dann wird der gesamte Prozess noch zweimal wiederholt. Nach der dritten Iteration hat sich die Zuordnung der Punkte zu Clustern nicht mehr verändert. An dieser Stelle wird der Algorithmus beendet.

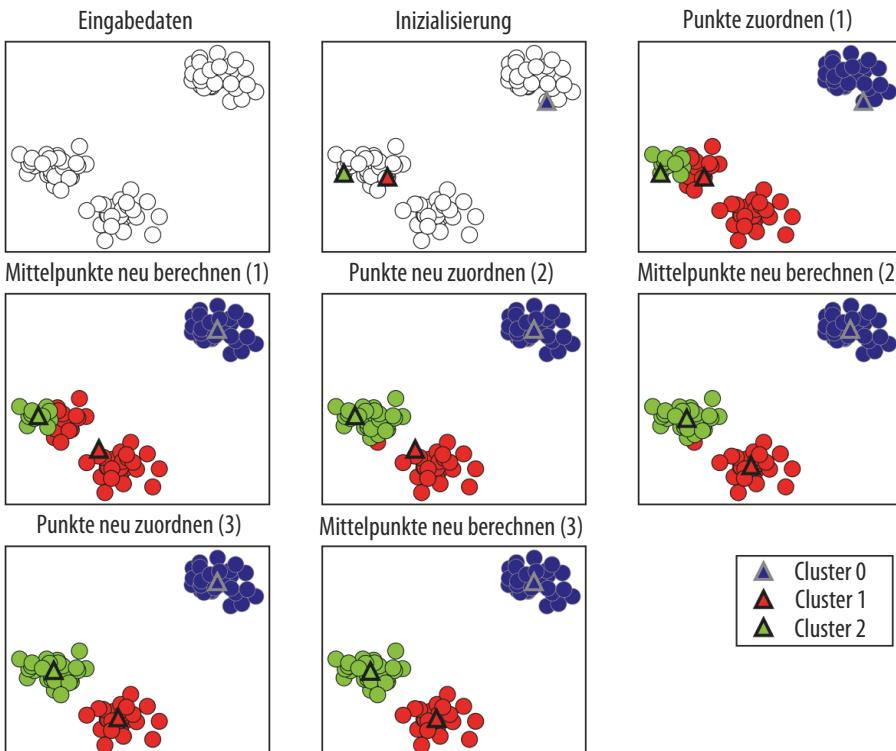


Abbildung 3-23: Eingabedaten und drei Schritte des k-Means-Algorithmus

Zusätzliche Datenpunkte werden von k-Means dem jeweils nächsten Clustermittel-  
punkt zugeordnet. Das nächste Beispiel (Abbildung 3-24) zeigt die Grenzen der  
erlernten Cluster in Abbildung 3-23:

**In[46]:**

```
mglearn.plots.plot_kmeans_boundaries()
```

Das k-Means-Verfahren lässt sich mit scikit-learn einigermaßen direkt umsetzen.  
Hier wenden wir es auf die synthetischen Daten an, die wir auch für die vorigen  
Plots verwendet hatten. Wir bilden dazu eine Instanz der Klasse KMeans und legen  
die Anzahl gewünschter Cluster fest.<sup>3</sup> Anschließend rufen wir die Methode fit mit  
den Daten auf:

**In[47]:**

```
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans

# generiere synthetische zweidimensionale Daten
```

<sup>3</sup> Falls Sie n\_clusters nicht angeben, wird automatisch der Wert 8 eingesetzt. Es gibt keinen besonderen Grund, diesen Wert zu verwenden.

```
X, y = make_blobs(random_state=1)

# konstruiere das Clustering-Modell
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
```

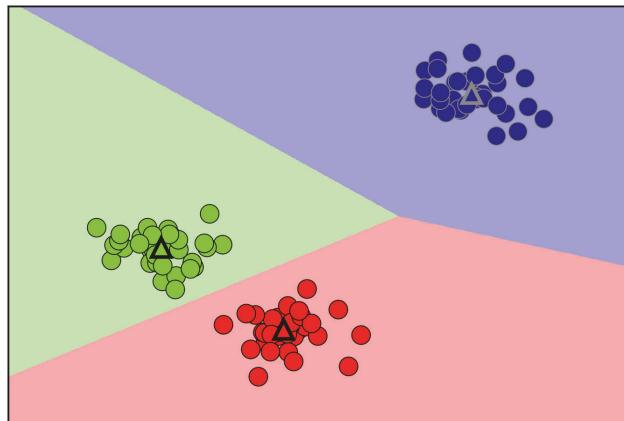


Abbildung 3-24: Durch den k-Means-Algorithmus gefundene Clustermittelpunkte und Grenzen von Clustern

Der Algorithmus weist jedem Datenpunkt in X eine Clusterbezeichnung zu. Sie finden diese Bezeichnungen im Attribut kmeans.labels\_:

**In[48]:**

```
print("Cluster-Zugehörigkeiten:\n{}".format(kmeans.labels_))
```

**Out[48]:**

```
Cluster-Zugehörigkeiten:
[1 2 2 2 0 0 0 2 1 1 2 2 0 1 0 0 0 1 2 2 0 2 0 1 2 0 0 1 1 0 1 1 0 1 2 0 2
 2 2 0 0 2 1 2 2 0 1 1 1 2 0 0 0 1 0 2 2 1 1 2 0 0 2 2 0 1 0 1 2 2 2 0 1
 1 2 0 0 1 2 1 2 2 0 1 1 1 2 1 0 1 1 2 2 0 0 1 0 1]
```

Da wir drei Cluster eingestellt hatten, sind die Cluster von 0 bis 2 durchnummieriert. Sie können über die Methode predict auch zusätzlichen Datenpunkten Clusterbezeichnungen zuweisen. Jeder Punkt wird bei der Vorhersage dem jeweils nächstgelegenen Clustermittelpunkt zugeordnet, aber das vorhandene Modell verändert sich nicht. Der Aufruf von predict auf den Trainingsdaten führt zum gleichen Ergebnis wie labels\_:

**In[49]:**

```
print(kmeans.predict(X))
```

**Out[49]:**

```
[1 2 2 2 0 0 0 2 1 1 2 2 0 1 0 0 0 1 2 2 0 2 0 1 2 0 0 1 1 0 1 1 0 1 2 0 2
 2 2 0 0 2 1 2 2 0 1 1 1 2 0 0 0 1 0 2 2 1 1 2 0 0 2 2 0 1 0 1 2 2 2 0 1
 1 2 0 0 1 2 1 2 2 0 1 1 1 2 1 0 1 1 2 2 0 0 1 0 1]
```

Sie sehen, dass Clustering Ähnlichkeiten mit Klassifizierung aufweist, in dem Sinne, dass jedem Element eine Bezeichnung zugeordnet wird. Allerdings gibt es keine im Voraus bekannte richtige Lösung, und die Bezeichner haben a priori keine Bedeutung. Kehren wir noch einmal zum oben besprochenen Beispiel des Clusters der Bilder von Gesichtern zurück. Es könnte sein, dass der vom Algorithmus gefundene Cluster 3 nur Gesichter Ihres Freundes Bela enthält. Sie können das allerdings nur herausfinden, indem Sie sich die Bilder ansehen. Die Zahl 3 ist dabei willkürlich. Die einzige Information, die Ihnen der Algorithmus liefert, ist, dass alle Gesichter im Cluster 3 einander ähnlich sind.

Für Clustering auf dem soeben berechneten künstlichen zweidimensionalen Datensatz bedeutet dies, dass es keine Bedeutung hat, ob eine Gruppe mit 0 und die andere mit 1 bezeichnet wurde. Eine wiederholte Ausführung des Algorithmus kann zu anderen Clusternummern führen, da die Initialisierung auf Zufall basiert.

Hier folgt noch einmal ein Diagramm der Daten (Abbildung 3-25). Die Mittelpunkte der Cluster sind im Attribut `cluster_centers_` gespeichert, und wir zeichnen diese als Dreiecke:

**In[50]:**

```
mglearn.discrete_scatter(X[:, 0], X[:, 1], kmeans.labels_, markers='o')
mglearn.discrete_scatter(
    kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], [0, 1, 2],
    markers='^', markeredgewidth=2)
```

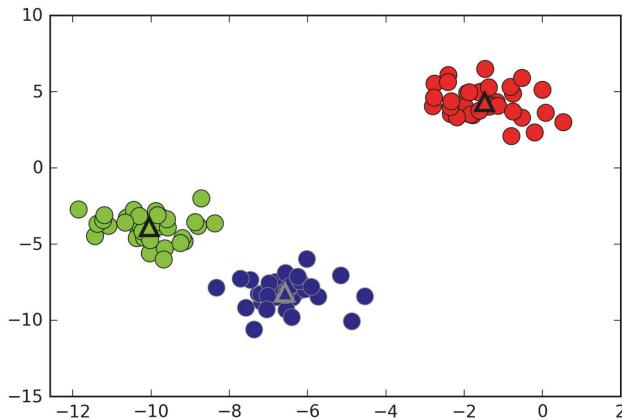


Abbildung 3-25: Vom k-Means-Algorithmus ermittelte Zuordnung zu Clustern und Clustermittelpunkte bei drei Clustern

Wir können auch mehr oder weniger Clustermittelpunkte einstellen (Abbildung 3-26):

**In[51]:**

```
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
```

```

# zwei Clustermittelpunkte:
kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
assignments = kmeans.labels_

mglearn.discrete_scatter(X[:, 0], X[:, 1], assignments, ax=axes[0])

# fünf Clustermittelpunkte:
kmeans = KMeans(n_clusters=5)
kmeans.fit(X)
assignments = kmeans.labels_

mglearn.discrete_scatter(X[:, 0], X[:, 1], assignments, ax=axes[1])

```

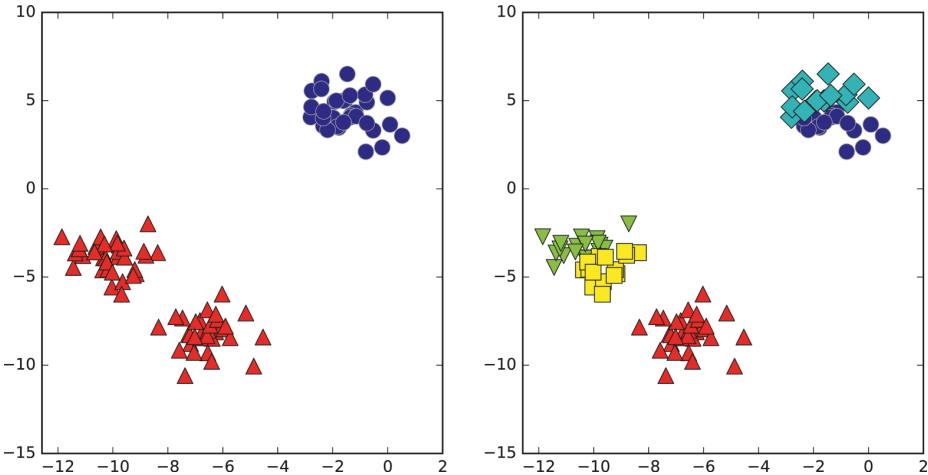


Abbildung 3-26: Vom  $k$ -Means-Algorithmus ermittelte Zuordnung zu Clustern mit zwei Clustern (links) und fünf Clustern (rechts)

### Wann versagt $k$ -Means?

Sogar wenn Sie die »richtige« Anzahl Cluster für einen gegebenen Datensatz kennen, findet  $k$ -Means diese nicht immer. Jeder Cluster ist allein durch seinen Mittelpunkt definiert. Daher hat jeder Cluster eine konvexe Gestalt. Eine Folge davon ist, dass  $k$ -Means nur recht einfache Formen erfassen kann.  $k$ -Means nimmt außerdem an, dass alle Cluster den gleichen »Durchmesser« haben; die Grenze zwischen Clustern wird immer genau in der Mitte zwischen den Mittelpunkten der Cluster gezogen. Das führt bisweilen zu überraschenden Ergebnissen wie in Abbildung 3-27:

In[52]:

```

X_varied, y_varied = make_blobs(n_samples=200,
                                 cluster_std=[1.0, 2.5, 0.5],
                                 random_state=170)
y_pred = KMeans(n_clusters=3, random_state=0).fit_predict(X_varied)

```

```
mglearn.discrete_scatter(X_varied[:, 0], X_varied[:, 1], y_pred)
plt.legend(["Cluster 0", "Cluster 1", "Cluster 2"], loc='best')
plt.xlabel("Merkmal 0")
plt.ylabel("Merkmal 1")
```

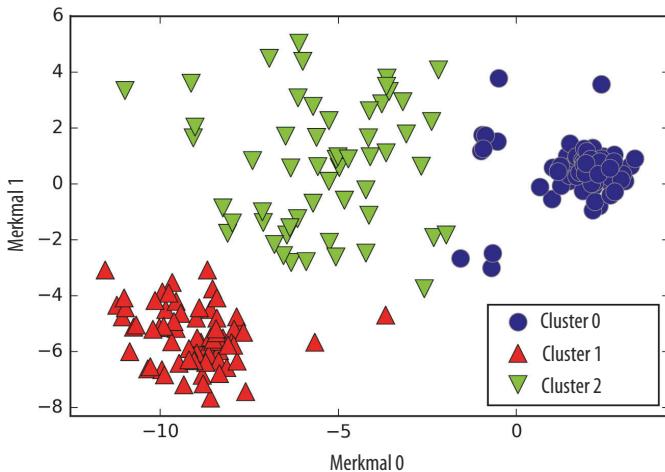


Abbildung 3-27: Von k-Means ermittelte Zuordnung bei Clustern mit unterschiedlicher Dichte

Man würde erwarten, dass die dichte Region links unten den ersten Cluster darstellt, die dichte Region oben rechts den zweiten und die weniger dichte Region in der Mitte den dritten. Stattdessen enthalten sowohl Cluster 0 als auch Cluster 1 einige Punkte in der mittleren Region, die weit von den übrigen in diesem Cluster entfernt sind.

k-Means nimmt außerdem an, dass alle Richtungen für jeden Cluster gleich wichtig sind. Das folgende Diagramm (Abbildung 3-28) zeigt einen zweidimensionalen Datensatz, bei dem es drei klar abgegrenzte Bereiche in den Daten gibt. Allerdings sind diese Bereiche zur Diagonale hin gestreckt. Weil k-Means lediglich den Abstand zum Clustermittelpunkt berücksichtigt, kann es mit dieser Art von Daten nichts anfangen:

In[53]:

```
# erzeuge einige zufällige Daten zum Clustern
X, y = make_blobs(random_state=170, n_samples=600)
rng = np.random.RandomState(74)

# transformiere die Daten, um sie zu strecken
transformation = rng.normal(size=(2, 2))
X = np.dot(X, transformation)

# Clustere die Daten zu drei Clustern
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
```

```

y_pred = kmeans.predict(X)

# zeichne die Zuordnung der Cluster und Clustermittelpunkte
plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap=mglearn.cm3)
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
           marker='^', c=[0, 1, 2], s=100, linewidth=2, cmap=mglearn.cm3)
plt.xlabel("Merkmal 0")
plt.ylabel("Merkmal 1")

```

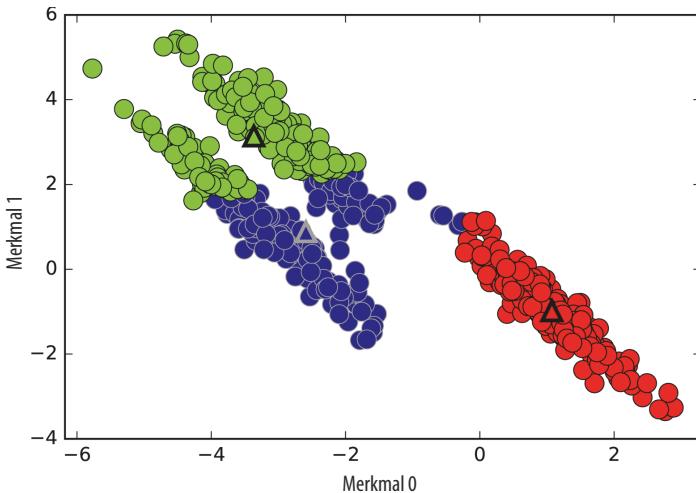


Abbildung 3-28: *k*-Means scheitert am Erkennen von nicht sphärischen Clustern

*k*-Means schneidet auch bei Clustern mit komplexeren Formen genauso schlecht ab wie der Datensatz `two_moons`, dem wir in Kapitel 2 begegnet sind (siehe Abbildung 3-29):

In[54]:

```

# erzeuge künstliche Daten für two_moons (diesmal mit weniger Rauschen)
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

# ordne die Daten zwei Clustern zu
kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
y_pred = kmeans.predict(X)

# plotte die Clusterzuordnung und Clustermittelpunkte
plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap=mglearn.cm2, s=60)
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
           marker='^', c=[mglearn.cm2(0), mglearn.cm2(1)], s=100, linewidth=2)
plt.xlabel("Merkmal 0")
plt.ylabel("Merkmal 1")

```

Man könnte hoffen, dass der Clustering-Algorithmus die zwei Halbmonde erfolgreich erkennt. Leider ist auch das mit dem *k*-Means-Algorithmus nicht möglich.

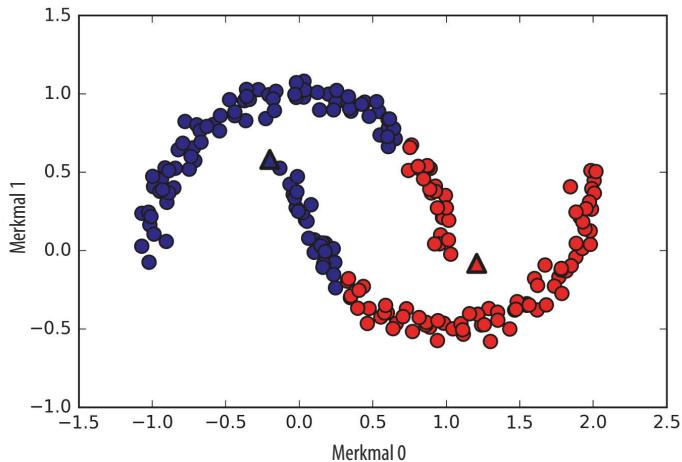


Abbildung 3-29: k-Means scheitert am Erkennen von Clustern mit komplexen Formen

### Vektorquantisierung oder k-Means als Dekomposition

Auch wenn  $k$ -Means ein Algorithmus zum Clustern ist, gibt es einige interessante Parallelen zwischen  $k$ -Means und den bereits besprochenen Dekompositionsmethoden wie der Hauptkomponentenzerlegung und NMF. Sie erinnern sich womöglich, dass die Hauptkomponentenzerlegung nach der Richtung mit der größten Varianz in den Daten sucht, während NMF additive Komponenten sucht, die oft »Extrema« oder »Teilen« der Daten entsprechen (siehe Abbildung 3-13). Beide Methoden versuchen, die Datenpunkte als Summe einiger Komponenten auszudrücken. Dagegen versucht  $k$ -Means, jeden Datenpunkt durch einen Clustermittelpunkt zu repräsentieren. Dies können Sie sich auch so vorstellen, dass jeder Punkt durch eine einzige Komponente repräsentiert wird, nämlich den Clustermittelpunkt. Diese Sichtweise auf das  $k$ -Means-Verfahren als Methode zur Dekomposition, bei der jeder Punkt durch eine Komponente dargestellt wird, nennt man *Vektorquantisierung*.

Vergleichen wir einmal die Hauptkomponentenzerlegung, NMF und  $k$ -Means, indem wir die extrahierten Komponenten (Abbildung 3-30) und die Rekonstruktion der Gesichter aus dem Testdatensatz mit 100 Komponenten (Abbildung 3-31) betrachten. Bei  $k$ -Means verwenden wir zur Rekonstruktion den nächsten Clustermittelpunkt im Trainingsdatensatz:

In[55]:

```
X_train, X_test, y_train, y_test = train_test_split(
    X_people, y_people, stratify=y_people, random_state=0)
nmf = NMF(n_components=100, random_state=0)
nmf.fit(X_train)
pca = PCA(n_components=100, random_state=0)
pca.fit(X_train)
kmeans = KMeans(n_clusters=100, random_state=0)
kmeans.fit(X_train)
```

```

X_reconstructed_pca = pca.inverse_transform(pca.transform(X_test))
X_reconstructed_kmeans = kmeans.cluster_centers_[kmeans.predict(X_test)]
X_reconstructed_nmf = np.dot(nmf.transform(X_test), nmf.components_)

```

Extrahierte Komponenten

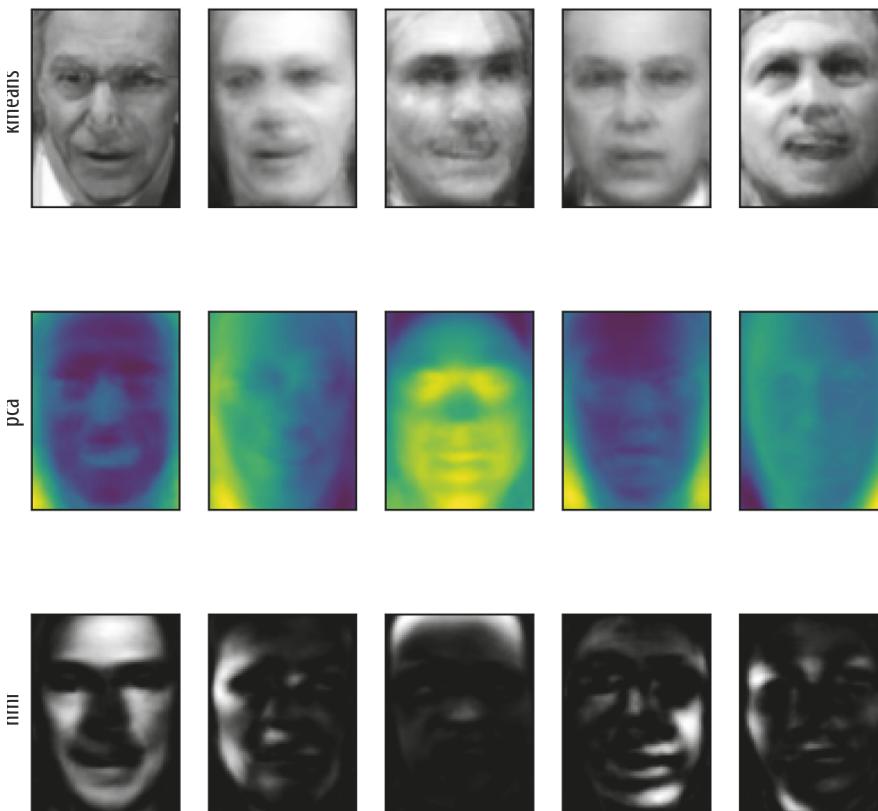


Abbildung 3-30: Vergleich der Clustermittelpunkte aus k-Means mit den durch Hauptkomponentenzerlegung (PCA) und NMF ermittelten Komponenten

In[56]:

```

fig, axes = plt.subplots(3, 5, figsize=(8, 8),
                       subplot_kw={'xticks': (), 'yticks': ()})
fig.suptitle("Extrahierte Komponenten")
for ax, comp_kmeans, comp_pca, comp_nmf in zip(
    axes.T, kmeans.cluster_centers_, pca.components_, nmf.components_):
    ax[0].imshow(comp_kmeans.reshape(image_shape))
    ax[1].imshow(comp_pca.reshape(image_shape), cmap='viridis')
    ax[2].imshow(comp_nmf.reshape(image_shape))

axes[0, 0].set_ylabel("kmeans")
axes[1, 0].set_ylabel("pca")
axes[2, 0].set_ylabel("nmf")

```

```

fig, axes = plt.subplots(4, 5, subplot_kw={'xticks': (), 'yticks': ()},
                       figsize=(8, 8))
fig.suptitle("Rekonstruktionen")
for ax, orig, rec_kmeans, rec_pca, rec_nmf in zip(
    axes.T, X_test, X_reconstructed_kmeans, X_reconstructed_pca,
    X_reconstructed_nmf):
    ax[0].imshow(orig.reshape(image_shape))
    ax[1].imshow(rec_kmeans.reshape(image_shape))
    ax[2].imshow(rec_pca.reshape(image_shape))
    ax[3].imshow(rec_nmf.reshape(image_shape))

axes[0, 0].set_ylabel("Original")
axes[1, 0].set_ylabel("kmeans")
axes[2, 0].set_ylabel("pca")
axes[3, 0].set_ylabel("nmf")

```

Rekonstruktionen



*Abbildung 3-31: Vergleich der Bildrekonstruktion mittels k-Means, Hauptkomponentenzerlegung (PCA) und NMF mit 100 Komponenten (oder Clustermittelpunkten) – k-Means verwendet nur einen einzelnen Clustermittelpunkt pro Bild.*

Ein interessanter Gesichtspunkt der Vektorquantisierung mit  $k$ -Means ist, dass wir sehr viel mehr Cluster als Eingabemerkmale zum Codieren unserer Daten verwenden können. Kehren wir noch einmal zum Datensatz `two_moons` zurück. Weder Hauptkomponentenzerlegung oder NMF können viel mit diesen Daten anfangen, da sie nur zwei Dimensionen besitzen. Diese mittels PCA oder NMF auf nur eine Dimension zu reduzieren, würde die Struktur der Daten komplett zerstören. Aber wir können mithilfe von  $k$ -Means die Daten weitaus deutlicher beschreiben, nämlich indem wir mehr Clustermittelpunkte verwenden (siehe Abbildung 3-32):

**In[57]:**

```
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

kmeans = KMeans(n_clusters=10, random_state=0)
kmeans.fit(X)
y_pred = kmeans.predict(X)

plt.scatter(X[:, 0], X[:, 1], c=y_pred, s=60, cmap='Paired')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s=60,
            marker='^', c=range(kmeans.n_clusters), linewidth=2, cmap='Paired')
plt.xlabel("Merkmals 0")
plt.ylabel("Merkmals 1")
print("Cluster-Zugehörigkeiten:\n{}".format(y_pred))
```

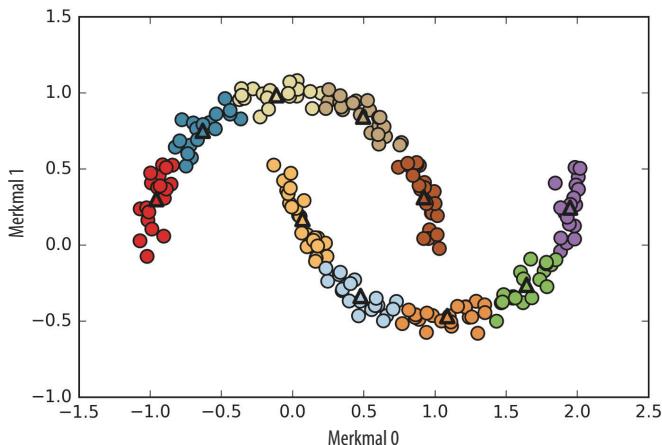


Abbildung 3-32: Verwenden von  $k$ -Means mit vielen Clustern, um die Variationen in einem komplexen Datensatz zu erfassen

**Out[57]:**

Cluster-Zugehörigkeiten:

```
[9 2 5 4 2 7 9 6 9 6 1 0 2 6 1 9 3 0 3 1 7 6 8 6 8 5 2 7 5 8 9 8 6 5 3 7 0
 9 4 5 0 1 3 5 2 8 9 1 5 6 1 0 7 4 6 3 3 6 3 8 0 4 2 9 6 4 8 2 8 4 0 4 0 5
 6 4 5 9 3 0 7 8 0 7 5 8 9 8 0 7 3 9 7 1 7 2 2 0 4 5 6 7 8 9 4 5 4 1 2 3 1
 8 8 4 9 2 3 7 0 9 9 1 5 8 5 1 9 5 6 7 9 1 4 0 6 2 6 4 7 9 5 5 3 8 1 9 5 6
 3 5 0 2 9 3 0 8 6 0 3 3 5 6 3 2 0 2 3 0 2 6 3 4 4 1 5 6 7 1 1 3 2 4 7 2 7
 3 8 6 4 1 4 3 9 9 5 1 7 5 8 2]
```

Wir haben 10 Clustermittelpunkte verwendet, sodass jedem Punkt eine Zahl zwischen 0 und 9 zugeordnet wird. Wir können das auch so betrachten, dass wir die Daten durch 10 Komponenten repräsentieren (wir haben also 10 neue Merkmale). Dabei sind alle Merkmale gleich 0, außer der Nummer des dem Datenpunkt zugeordneten Clustermittelpunktes. Mit dieser 10-dimensionalen Repräsentation wäre es nun möglich, die zwei Halbmonde mit einem linearen Modell zu separieren. Mit den zwei ursprünglichen Merkmalen war das nicht möglich. Es ist sogar eine noch explizitere Repräsentation der Daten möglich, bei der die Entferungen der Daten zu jedem der Clustermittelpunkte als Merkmal verwendet werden. Diese lässt sich über die Methode `transform` von `kmeans` berechnen:

**In[58]:**

```
distance_features = kmeans.transform(X)
print("Abmessungen der Distanzmerkmale: {}".format(distance_features.shape))
print("Distanzmerkmale:\n{}".format(distance_features))
```

**Out[58]:**

```
Abmessungen der Distanzmerkmale: (200, 10)
Distanzmerkmale:
[[ 0.922  1.466  1.14 ...,  1.166  1.039  0.233]
 [ 1.142  2.517  0.12 ...,  0.707  2.204  0.983]
 [ 0.788  0.774  1.749 ...,  1.971  0.716  0.944]
 ...,
 [ 0.446  1.106  1.49 ...,  1.791  1.032  0.812]
 [ 1.39   0.798  1.981 ...,  1.978  0.239  1.058]
 [ 1.149  2.454  0.045 ...,  0.572  2.113  0.882]]
```

$k$ -Means ist ein sehr beliebter Clustering-Algorithmus, nicht nur, weil er vergleichsweise leicht zu verstehen ist, sondern auch, weil er schnell ausgeführt wird.  $k$ -Means skaliert gut auf große Datensätze und scikit-learn enthält mit der Klasse `MiniBatchKMeans` sogar eine besser skalierbare Variante, die mit sehr großen Datensätzen umgehen kann.

Einer der Nachteile von  $k$ -Means ist die Abhängigkeit von zufälligen Startpunkten. Damit hängt das Ergebnis des Algorithmus vom Seed-Wert des Zufallsgenerators ab. In scikit-learn wird der Algorithmus standardmäßig zehnmal mit unterschiedlichen Startwerten ausgeführt und das beste Ergebnis ausgegeben.<sup>4</sup> Weitere Nachteile von  $k$ -Means sind die relativ strengen Annahmen über die Form von Clustern und die zwingende Angabe der Anzahl der zu findenden Cluster (welche in einer realen Anwendung nicht unbedingt bekannt ist).

Als Nächstes werden wir uns zwei weitere Cluster-Algorithmen ansehen, um diesen Nachteilen etwas entgegenzusetzen.

## Agglomeratives Clustering

*Agglomeratives Clustering* bezeichnet eine Familie von Clustering-Algorithmen, die alle nach dem gleichen Prinzip funktionieren: Der Algorithmus beginnt damit, dass

---

<sup>4</sup> In diesem Fall heißt »bestes Ergebnis«, dass die Summe der Varianzen der Cluster klein ist.

jeder Punkt als ein eigener Cluster aufgefasst wird, und vereinigt dann sukzessiv die zwei jeweils ähnlichsten Cluster, bis eine Abbruchbedingung erreicht wird. Die in scikit-learn implementierte Abbruchbedingung ist die Anzahl der Cluster, sodass Cluster miteinander fusioniert werden, bis nur noch eine vorgegebene Anzahl Cluster übrig ist. Es gibt verschiedene *Linkage*-Kriterien, ein Ähnlichkeitsmaß zur Bestimmung der jeweils »ähnlichsten Cluster«. Dieses Ähnlichkeitsmaß ist für zwei beliebige Cluster stets berechenbar.

Die folgenden drei Ähnlichkeitsmaße sind in scikit-learn implementiert:

`ward`

Das standardmäßig eingestellte `ward`-Verfahren wählt zwei zu fusionierende Cluster so aus, dass die Varianz innerhalb aller Cluster am wenigsten ansteigt. Dies führt meist zu etwa gleich großen Clustern.

`average`

Average-Linkage fusioniert die zwei Cluster mit dem geringsten durchschnittlichen Abstand ihrer Punkte untereinander.

`complete`

Complete-Linkage (auch als Maximum Linkage bekannt) fusioniert die zwei Cluster mit der geringsten Maximaldistanz zwischen zwei Punkten.

`ward` funktioniert für die meisten Datensätze. Wir werden dieses Verfahren in unseren Beispielen einsetzen. Falls die Cluster eine sehr unterschiedliche Anzahl Punkte aufweisen (z. B. ein Cluster sehr viel größer als die anderen ist), könnten auch `average` oder `complete` besser funktionieren.

Das folgende Diagramm (Abbildung 3-33) verdeutlicht die Schritte beim agglomerativen Clustering eines zweidimensionalen Datensatzes, bei dem wir nach drei Clustern suchen:

**In[59]:**

```
mglearn.plots.plot_agglomerative_algorithm()
```

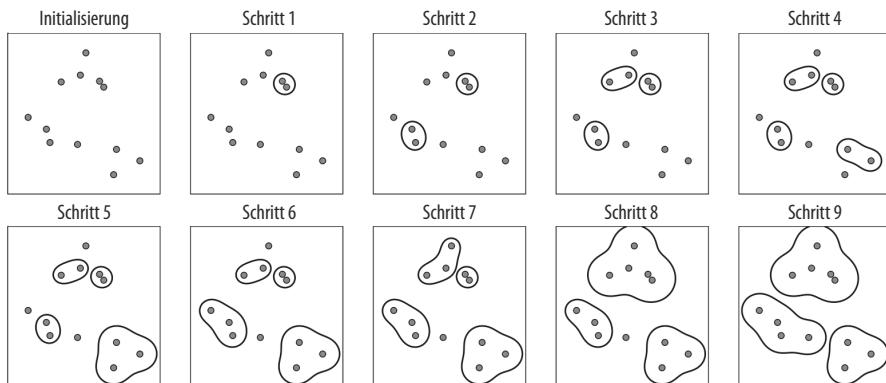


Abbildung 3-33: Agglomeratives Clustering fusioniert die zwei ähnlichsten Cluster iterativ miteinander

Anfänglich ist jeder Punkt ein einzelner Cluster. Dann werden in jedem Schritt die zwei ähnlichsten Cluster miteinander vereinigt. In den ersten vier Schritten werden zwei Cluster mit je einem einzelnen Punkt ausgewählt und zu Zwei-Punkte-Clustern fusioniert. In Schritt 5 wird ein Zwei-Punkte-Cluster auf drei Punkte erweitert usw. In Schritt 9 sind nur noch drei Cluster übrig. Wir hatten angegeben, dass wir nach drei Clustern suchen, und deshalb bricht der Algorithmus an dieser Stelle ab.

Betrachten wir einmal, wie das agglomerative Clustering bei diesem einfachen Drei-Cluster-Datenbeispiel abschneidet. Die Funktionsweise des Algorithmus bedingt, dass agglomeratives Clustering keinerlei Vorhersage für neue Datenpunkte treffen kann. Daher besitzt AgglomerativeClustering keine Methode predict. Um das Modell zu erstellen und die Clusterzugehörigkeiten für die Trainingsdaten zu berechnen, verwenden wir stattdessen die Methode fit\_predict.<sup>5</sup> Das Ergebnis ist in Abbildung 3-34 dargestellt:

In[60]:

```
from sklearn.cluster import AgglomerativeClustering
X, y = make_blobs(random_state=1)

agg = AgglomerativeClustering(n_clusters=3)
assignment = agg.fit_predict(X)

mglearn.discrete_scatter(X[:, 0], X[:, 1], assignment)
plt.xlabel("Merkmals 0")
plt.ylabel("Merkmals 1")
```

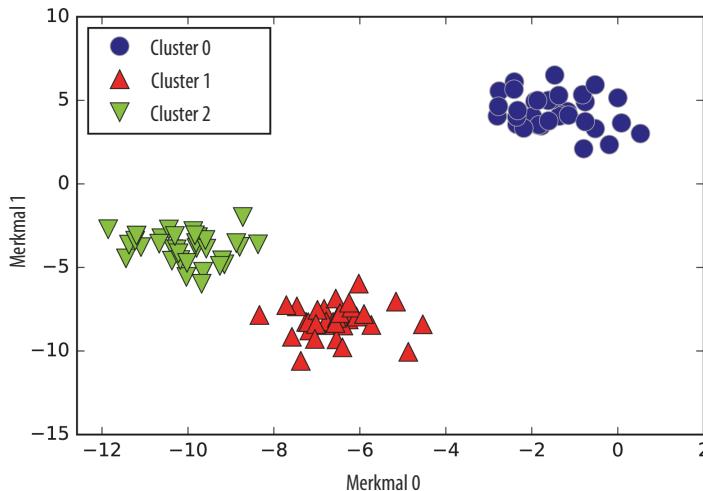


Abbildung 3-34: Zuordnung von Clustern über agglomeratives Clustering mit drei Clustern

Der Algorithmus ordnet die Cluster wie erwartet perfekt zu. Obwohl die Anzahl der zu findenden Cluster bei der Implementierung in scikit-learn angegeben wer-

---

<sup>5</sup> Wir könnten auch das Attribut labels\_ verwenden, wie wir es bei k-Means getan haben.

den muss, geben uns agglomerative Clusterverfahren eine Hilfestellung beim Herausfinden der richtigen Anzahl, wie wir im nächsten Abschnitt sehen werden.

## Hierarchisches Clustering und Dendrogramme

Das Ergebnis beim agglomerativen Clustering ist ein sogenanntes *hierarchisches Clustering*. Das Clusterverfahren arbeitet iterativ, sodass jeder Punkt eine Reise vom anfänglichen Ein-Punkt-Cluster bis zu seinem endgültigen Cluster durchlebt. Jeder Zwischenschritt ist eine gültige Zuordnung (einer anderen Anzahl) von Clustern. Manchmal ist es hilfreich, sich alle möglichen Cluster auf einmal anzusehen. Das nächste Beispiel (Abbildung 3-35) zeigt eine Überlagerung aller möglichen Clusterzuordnungen in Abbildung 3-33. Dies gibt uns Aufschluss darüber, wie sich jeder Cluster aus kleineren Clustern zusammensetzt:

In[61]:

```
mglearn.plots.plot_agglomerative()
```

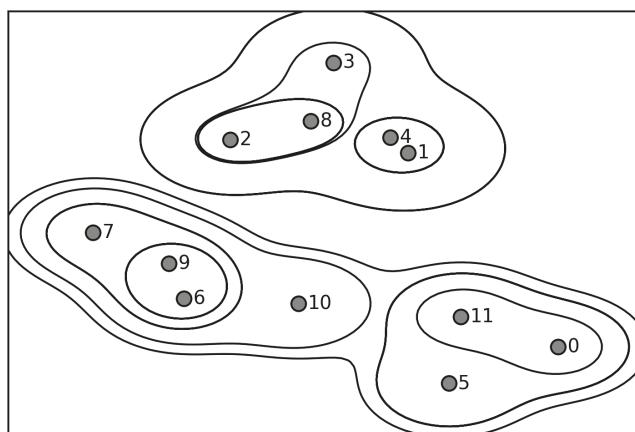


Abbildung 3-35: Über agglomeratives Clustering erzeugte hierarchische Zuordnung von Clustern (als Linien dargestellt). Die Datenpunkte sind nummeriert (siehe auch Abbildung 3-36).

Diese Art der Visualisierung des hierarchischen Clusterings ist sehr detailliert. Sie ist von den zwei Dimensionen der Daten abhängig und lässt sich deshalb nicht auf Datensätze mit mehr als zwei Merkmalen anwenden. Es gibt allerdings ein Werkzeug zum Darstellen von hierarchischen Clustern auch in mehreren Dimensionen, nämlich *Dendrogramme*.

Leider verfügt scikit-learn gegenwärtig über keine Funktionalität zum Zeichnen von Dendrogrammen. Sie können diese allerdings leicht mit SciPy generieren. Die Clustering-Algorithmen in SciPy haben eine etwas andere Schnittstelle als scikit-learn. SciPy enthält eine Funktion, die aus einem Array  $X$  mit Eingabedaten ein *Linkage Array* berechnet, in dem die hierarchischen Ähnlichkeiten von Clustern enthalten sind. Dieses Linkage Array lässt sich in die SciPy-Funktion `dendrogram` einspeisen, um das Dendrogramm zu zeichnen (Abbildung 3-36):

In[62]:

```
# Importiere die Funktionen dendrogram und ward Clustering aus SciPy
from scipy.cluster.hierarchy import dendrogram, ward

X, y = make_blobs(random_state=0, n_samples=12)
# Führe ein ward-Clustering auf den Daten im Array X durch
# Die Funktion ward in SciPy liefert ein Array mit den
# beim agglomerativen Clustering überbrückten Distanzen
linkage_array = ward(X)
# Nun zeichnen wir ein Dendrogramm mit den Distanzen
# zwischen Clustern in linkage_array
dendrogram(linkage_array)

# Markiere die Schnittpunkte im Baum für zwei und drei Cluster
ax = plt.gca()
bounds = ax.get_xbound()
ax.plot(bounds, [7.25, 7.25], '--', c='k')
ax.plot(bounds, [4, 4], '--', c='k')

ax.text(bounds[1], 7.25, 'zwei Cluster', va='center', fontdict={'size': 15})
ax.text(bounds[1], 4, 'drei Cluster', va='center', fontdict={'size': 15})
plt.xlabel("Index")
plt.ylabel("Cluster-Distanz")
```

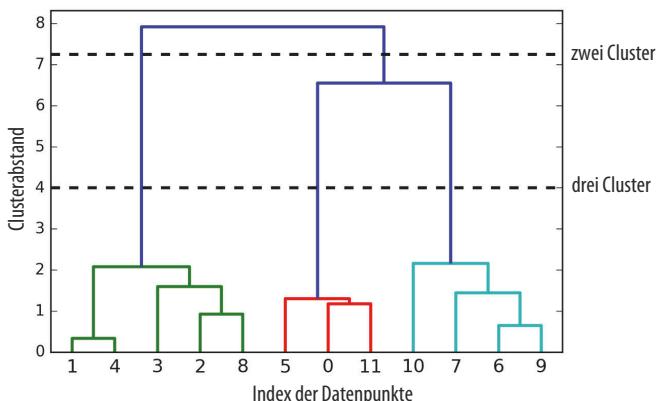


Abbildung 3-36: Dendrogramm des in Abbildung 3-35 berechneten Clusterings. Die Linien zeigen die Aufteilung in zwei bzw. drei Cluster an.

Das Dendrogramm zeigt die Datenpunkte unten als Punkte mit Nummern von 0 bis 11 an. Anschließend wird mit diesen Punkten ein Baum aufgebaut, dessen Blätter Ein-Punkte-Cluster darstellen. Für jeweils zwei fusionierte Cluster wird ein neuer Knoten im Baum hinzugefügt.

Liest man den Baum von unten nach oben, werden zunächst die Datenpunkte 1 und 4 fusioniert (wie in Abbildung 3-33 gezeigt). Anschließend werden die Punkte 6 und 9 zu einem Cluster vereint usw. Auf der obersten Ebene gibt es noch zwei Äste, einer bestehend aus den Punkten 11, 0, 5, 10, 7, 6 und 9, der andere aus den

Punkten 1, 4, 3, 2 und 8. Diese entsprechen den zwei größten Clustern auf der linken Seite des Diagramms.

Die y-Achse im Dendrogramm zeigt nicht nur, in welcher Reihenfolge die Cluster im Verlauf des agglomerativen Algorithmus fusioniert werden. Die Länge der Äste stellt außerdem den Abstand der vereinigten Cluster zueinander dar. Die längsten Äste im Dendrogramm sind an der gestrichelten Linie mit »three clusters« beschriftet. Die Länge der Äste weist darauf hin, dass beim Schritt von drei auf zwei Cluster einige sehr weit voneinander entfernte Punkte vereinigt werden mussten. Wir sehen das noch einmal am oberen Ende des Diagramms, wo zum Fusionieren der zwei verbliebenen Cluster zu einem einzigen wieder eine relativ große Distanz überbrückt werden muss.

Leider scheitert das agglomerative Clustering am Aufteilen komplexer Formen wie dem Datensatz `two_moons`. Bei DBSCAN, dem dritten betrachteten Algorithmus, ist das aber nicht so.

## DBSCAN

Ein weiterer sehr nützlicher Clustering-Algorithmus ist DBSCAN (»density-based spatial clustering of applications with noise«). Die wichtigsten Vorteile von DBSCAN sind, dass der Nutzer die Anzahl der Cluster nicht *a priori* einstellen muss, dass das Verfahren auch Cluster mit komplexen Formen erfassen kann und dass es auch Punkte identifiziert, die zu keinem der Cluster gehören. Das DBSCAN-Verfahren ist etwas langsamer als das agglomerative Clustering und *k*-Means, skaliert aber trotzdem auch für relativ große Datensätze.

DBSCAN identifiziert Punkte in »dicht besetzten« Regionen des Merkmalsraumes, in denen viele Datenpunkte dicht beieinanderliegen. Solche Regionen nennt man auch *dichte* Bereiche im Merkmalsraum. Die Grundidee bei DBSCAN ist, dass Cluster dichte Regionen in den Daten bilden und durch relativ leere Bereiche getrennt sind.

Punkte innerhalb einer dichten Region nennt man *Kernobjekte* (oder core points). Diese sind wie folgt definiert. In DBSCAN gibt es zwei Parameter: `min_samples` und `eps`. Wenn mindestens `min_samples` Datenpunkte innerhalb des Abstandes `eps` zu einem gegebenen Punkt liegen, wird dieser Datenpunkt als ein Kernobjekt eingestuft. Kernobjekte, die näher als der Abstand `eps` zueinander entfernt sind, ordnet DBSCAN dem gleichen Cluster zu.

Der Algorithmus wählt zu Beginn einen beliebigen Startpunkt aus. Dann findet er alle Punkte im Abstand `eps` oder näher zu diesem Punkt. Falls im Abstand `eps` zum Startpunkt weniger als `min_samples` Punkte gefunden werden, wird dieser Punkt als *Rauschen* eingestuft. Das bedeutet, dass er zu keinem Cluster gehört. Falls es mehr als `min_samples` Punkte im Abstand `eps` gibt, wird der Punkt als Kernobjekt beschriftet und erhält eine neue Clusterbezeichnung. Anschließend werden alle seine Nachbarn (im Abstand `eps`) abgearbeitet. Falls diese noch keinem Cluster

zugeordnet wurden, erhalten auch diese die soeben erstellte Clusterbezeichnung. Falls sie Kernobjekte sind, werden auch deren Nachbarn abgearbeitet usw. Der Cluster wächst an, bis es keine weiteren Kernobjekte im Abstand  $\text{eps}$  mehr gibt. Anschließend wird der nächste noch nicht abgearbeitete Punkt ausgewählt und die Prozedur für diesen wiederholt.

Am Ende bleiben drei Arten von Datenpunkten übrig: Kernobjekte, Punkte im Abstand  $\text{eps}$  von Kernobjekten (sogenannte *Dichte-erreichbare Punkte*) und Rauschen. Wird der DBSCAN-Algorithmus mehrmals auf dem gleichen Datensatz ausgeführt, ist das Clustering der Kernobjekte jedes Mal identisch, und es werden stets die gleichen Punkte als Rauschen eingeordnet. Allerdings können Dichte-erreichbare Punkte zu Kernobjekten von mehr als einem Cluster benachbart sein. Daher hängt die Clusterzugehörigkeit der Dichte-erreichbaren Punkte von der Reihenfolge ab, in der die Punkte abgearbeitet werden. Für gewöhnlich gibt es nur wenige Dichte-erreichbare Punkte, und die Abhängigkeit von der Reihenfolge ist nicht sehr wichtig.

Wenden wir DBSCAN nun auf den gleichen künstlichen Datensatz an, anhand dessen wir bereits das agglomerative Clustering veranschaulicht haben. Wie beim agglomerativen Clustering erlaubt auch DBSCAN keine Vorhersage anhand zusätzlicher Testdaten, daher führen wir das Clustering mit der Methode `fit_predict` durch und ermitteln im gleichen Schritt auch die Clusterbezeichnungen:

**In[63]:**

```
from sklearn.cluster import DBSCAN
X, y = make_blobs(random_state=0, n_samples=12)

dbSCAN = DBSCAN()
clusters = dbSCAN.fit_predict(X)
print("Clusterzugehörigkeiten:\n{}".format(clusters))
```

**Out[63]:**

```
Clusterzugehörigkeiten:
[-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
```

Wie Sie sehen können, wurde allen Datenpunkten die Bezeichnung `-1` für Rauschen zugewiesen. Das liegt daran, dass die Standardparameter für  $\text{eps}$  und `min_samples` nicht für kleine Datensätze geeignet sind. Die Clusterzuordnungen mit verschiedenen Werten für `min_samples` und  $\text{eps}$  sind unten gezeigt und in Abbildung 3-37 visualisiert:

**In[64]:**

```
mglearn.plots.plot_dbSCAN()
```

**Out[64]:**

```
min_samples: 2 eps: 1.000000 cluster: [-1  0  0 -1  0 -1  1  1  0  1 -1 -1]
min_samples: 2 eps: 1.500000 cluster: [0  1  1  1  1  0  2  2  1  2  2  0]
min_samples: 2 eps: 2.000000 cluster: [0  1  1  1  1  0  0  0  1  0  0  0]
min_samples: 2 eps: 3.000000 cluster: [0  0  0  0  0  0  0  0  0  0  0  0]
min_samples: 3 eps: 1.000000 cluster: [-1  0  0 -1  0 -1  1  1  0  1 -1 -1]
```

```

min_samples: 3 eps: 1.500000 cluster: [0 1 1 1 1 0 2 2 1 2 2 0]
min_samples: 3 eps: 2.000000 cluster: [0 1 1 1 1 0 0 0 1 0 0 0]
min_samples: 3 eps: 3.000000 cluster: [0 0 0 0 0 0 0 0 0 0 0 0]
min_samples: 5 eps: 1.000000 cluster: [-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
min_samples: 5 eps: 1.500000 cluster: [-1 0 0 0 0 -1 -1 -1 0 -1 -1 -1]
min_samples: 5 eps: 2.000000 cluster: [-1 0 0 0 0 -1 -1 -1 0 -1 -1 -1]
min_samples: 5 eps: 3.000000 cluster: [0 0 0 0 0 0 0 0 0 0 0 0]

```

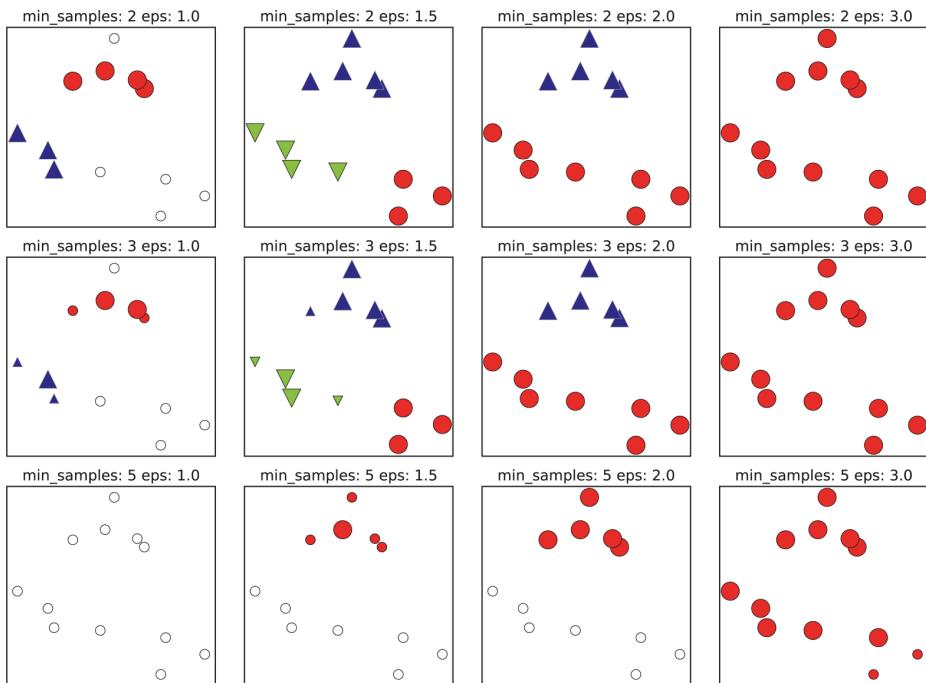


Abbildung 3-37: Von DBSCAN gefundene Clusterzuordnungen mit unterschiedlichen Werten für die Parameter `min_samples` und `eps`

In diesem Diagramm sind die zu Clustern gehörenden Punkte farbig ausgefüllt, während das Rauschen als transparente Kreise zu sehen ist. Kernobjekte sind durch große Markierungen, Dichte-erreichbare Punkte durch kleinere Markierungen gekennzeichnet. Vergrößern von `eps` (im Diagramm von links nach rechts) führt dazu, dass mehr Punkte in einem Cluster aufgenommen werden. Die Cluster wachsen dadurch an, es können aber auch mehrere Cluster zu einem verschmelzen. Erhöhen von `min_samples` (im Diagramm von oben nach unten) bedeutet, dass weniger Punkte Kernobjekte sind und mehr Punkte als Rauschen eingestuft werden.

Der Parameter `eps` ist insgesamt der wichtigere, weil er bestimmt, was »nah beieinanderliegende Punkte« sind. Ein sehr kleiner Wert für `eps` führt dazu, dass es keine Kernobjekte gibt und alle Punkte als Rauschen markiert werden. Ein sehr großer Wert für `eps` führt dazu, dass alle Punkte einen einzigen Cluster bilden.

Der Parameter `min_samples` bestimmt vor allem, ob Punkte in weniger dichten Bereichen als Ausreißer von Clustern oder als eigene Cluster klassifiziert werden.

Wenn Sie `min_samples` erhöhen, wird jeder Bereich mit weniger als `min_samples` Punkten als Rauschen eingestuft. Daher legt `min_samples` die minimale Clustergröße fest. Sie können das sehr deutlich in Abbildung 3-37 beim Übergang von `min_samples=3` zu `min_samples=5` mit `eps=1.5` sehen. Mit `min_samples=3` gibt es drei Cluster: einen mit vier Punkten, einen mit fünf Punkten und einen mit drei Punkten. Mit `min_samples=5` werden die zwei kleineren Cluster (mit drei und vier Punkten) nun als Rauschen eingestuft, sodass nur der Cluster mit fünf Punkten übrig bleibt.

Obwohl DBSCAN nicht erwartet, dass man die Clusteranzahl im Voraus explizit festlegt, bestimmt der Wert für `eps` implizit die Anzahl der gefundenen Cluster. Manchmal ist es nach Skalieren der Daten mit `StandardScaler` oder `MinMaxScaler` leichter, einen guten Wert für `eps` zu finden, da diese Skalierungstechniken sicherstellen, dass alle Merkmale eine ähnliche Spannbreite haben. Abbildung 3-38 zeigt das Ergebnis von DBSCAN auf dem Datensatz `two_moons`. Der Algorithmus erkennt die zwei Halbkreise und unterscheidet bereits mit den Standardparametern zwischen ihnen:

**In[65]:**

```
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

# skaliere die Daten auf Mittelwert Null und Varianz Eins
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)

dbSCAN = DBSCAN()
clusters = dbSCAN.fit_predict(X_scaled)
# plotte die Clusterzuordnung
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=clusters, cmap=mglearn.cm2, s=60)
plt.xlabel("Merkmals 0")
plt.ylabel("Merkmals 1")
```

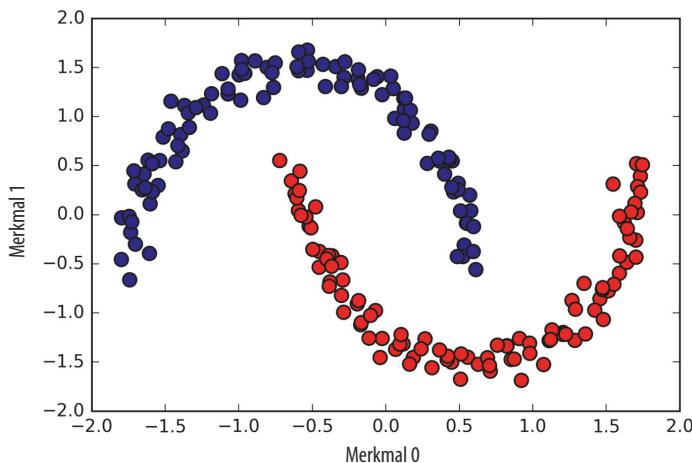


Abbildung 3-38: Von DBSCAN ermittelte Clusterzuordnung mit der Standardeinstellung `eps=0.5`

Da der Algorithmus die gewünschte Anzahl von Clustern (zwei) ausgibt, scheinen die Parameter gut zu funktionieren. Wenn wir `eps` auf 0.2 senken (vom Standardwert 0.5), erhalten wir acht Cluster, was natürlich zu viele sind. Erhöhen wir `eps` auf 0.7, erhalten wir einen einzigen Cluster.

Beim Verwenden von DBSCAN müssen Sie mit den ermittelten Clusterzuordnungen ein wenig aufpassen. Der Wert -1 steht für Rauschen und könnte zu unerwünschten Nebeneffekten führen, wenn Sie die Clusterzuordnungen zum Indizieren eines weiteren Arrays verwenden.

## Vergleichen und Auswerten von Clusteralgorithmen

Eine Herausforderung beim Verwenden von Clustering-Algorithmen ist, dass man die Qualität des Ergebnisses nur sehr schwer bewerten und Ergebnisse unterschiedlicher Algorithmen nur schwer vergleichen kann. Nachdem wir die Algorithmen beim *k*-Means-Verfahren, agglomerativen Clustering und DBSCAN besprochen haben, werden wir diese anhand einiger realer Datensätze miteinander vergleichen.

### Ein Clustering mithilfe eines Referenzdatensatzes auswerten

Das Ergebnis eines Clustering-Algorithmus lässt sich mit verschiedenen Metriken relativ zum Clustering eines Referenzdatensatzes auswerten. Die wichtigsten sind der *angepasste Rand-Index* (adjusted rand Index, ARI) und die *normalisierte gegenseitige Information* (normalized mutual information, NMI). Beide liefern eine Maßzahl zwischen mit dem optimalen Wert 1 und dem Wert 0 bei nicht verwandten Clusterungen (auch wenn der ARI-Wert negativ sein kann).

In diesem Abschnitt werden wir die Algorithmen *k*-Means, agglomeratives Clustering und DBSCAN mithilfe des ARI miteinander vergleichen. Wir ziehen auch eine zufällige Zuordnung von Punkten zu zwei Clustern in die Betrachtung mit ein (siehe Abbildung 3-39):

In[66]:

```
from sklearn.metrics.cluster import adjusted_rand_score
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

# skaliere die Daten auf Mittelwert 0 und Varianz 1
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)

fig, axes = plt.subplots(1, 4, figsize=(15, 3),
                      subplot_kw={'xticks': (), 'yticks': ()})

# erstelle eine Liste der zu verwendenden Algorithmen
algorithms = [KMeans(n_clusters=2), AgglomerativeClustering(n_clusters=2),
              DBSCAN()]

# erstelle eine zufällige Zuordnung von Clustern zum Vergleich
random_state = np.random.RandomState(seed=0)
random_clusters = random_state.randint(low=0, high=2, size=len(X))
```

```

# plotte die zufällige Zuordnung
axes[0].scatter(X_scaled[:, 0], X_scaled[:, 1], c=random_clusters,
                 cmap=mglearn.cm3, s=60)
axes[0].set_title("Zufällige Zuordnung - ARI: {:.2f}".format(
    adjusted_rand_score(y, random_clusters)))

for ax, algorithm in zip(axes[1:], algorithms):
    # plotte die Clusterzuordnung und Clustermittelpunkte
    clusters = algorithm.fit_predict(X_scaled)
    ax.scatter(X_scaled[:, 0], X_scaled[:, 1], c=clusters,
               cmap=mglearn.cm3, s=60)
    ax.set_title("{} - ARI: {:.2f}".format(algorithm.__class__.__name__,
                                           adjusted_rand_score(y, clusters)))

```

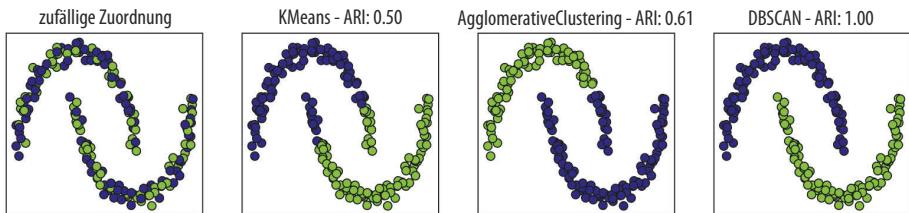


Abbildung 3-39: Vergleich von zufälliger Zuordnung, k-Means, agglomerativem Clustering und DBSCAN auf dem Datensatz two\_moons mithilfe des überwachten ARI Score

Der angepasste Rand-Index liefert ein intuitiv verständliches Ergebnis, bei dem eine zufällige Zuordnung den Score 0 erhält und DBSCAN (welches die gewünschten Cluster perfekt erkennt) den Wert 1.

Beim Auswerten von Clusterings mit dieser Methode ist es ein verbreiteter Fehler, die Funktion `accuracy_score` anstelle von `adjusted_rand_score`, `normalized_mutual_info_score` oder einer anderen Metrik für Clusterings zu verwenden. Das Problem beim Verwenden der Genauigkeit ist, dass hierbei die zugeordneten Clusterbezeichner denen im Referenzdatensatz exakt entsprechen müssen. Allerdings sind die Clusterbezeichner selbst ohne Bedeutung – es kommt nur darauf an, welche Punkte im gleichen Cluster sind:

**In[67]:**

```

from sklearn.metrics import accuracy_score

# diese zwei Zuordnungen von Punkten entsprechen
# dem gleichen Clustering
clusters1 = [0, 0, 1, 1, 0]
clusters2 = [1, 1, 0, 0, 1]
# die Genauigkeit ist null, da keine Bezeichner identisch sind
print("Genauigkeit: {:.2f}".format(accuracy_score(clusters1, clusters2)))
# der angepasste Rand-Index ist 1, da das Clustering identisch ist
print("ARI: {:.2f}".format(adjusted_rand_score(clusters1, clusters2)))

```

**Out[67]:**

```

Genauigkeit: 0.00
ARI: 1.00

```

## Auswerten von Clustern ohne Referenzdaten

Auch wenn wir soeben eine Möglichkeit zum Evaluieren von Clustering-Algorithmen kennengelernt haben, gibt es in der Praxis bei der Verwendung von Metriken wie ARI ein großes Problem. Bei Clustering-Algorithmen gibt es für gewöhnlich keine Referenzdaten, mit denen man die Ergebnisse vergleichen könnte. Würden wir die korrekte Clusterung der Daten kennen, könnten wir mit diesen Daten ein überwachtes Modell bauen, etwa einen Klassifikator. Daher helfen Metriken wie ARI und NMI normalerweise nur beim Entwickeln von Algorithmen, nicht beim Bewerten des Erfolges ihrer Anwendung.

Einige Metriken zum Bewerten von Clusterverfahren wie der *Silhouettenkoeffizient* erfordern keine Referenzdaten. Allerdings funktionieren diese in der Praxis meist nicht so gut. Der Silhouetten-Score berechnet die Kompaktheit eines Clusters, wobei ein höherer Wert besser ist, bis zum perfekten Ergebnis 1. Obwohl kompakte Cluster eine gute Sache sind, lässt Kompaktheit keine komplexen Formen zu.

Hier folgt ein beispielhafter Vergleich von  $k$ -Means, agglomerativem Clustering und DBSCAN mit dem Silhouette Score auf dem Datensatz two-moons (Abbildung 3-40):

In[68]:

```
from sklearn.metrics.cluster import silhouette_score

X, y = make_moons(n_samples=200, noise=0.05, random_state=0)
# skaliere die Daten auf Mittelwert 0 und Varianz 1
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)

fig, axes = plt.subplots(1, 4, figsize=(15, 3),
                      subplot_kw={'xticks': (), 'yticks': ()})

# erstelle zum Vergleich eine zufällige Clusterzuordnung
random_state = np.random.RandomState(seed=0)
random_clusters = random_state.randint(low=0, high=2, size=len(X))

# plotte die zufällige Zuordnung
axes[0].scatter(X_scaled[:, 0], X_scaled[:, 1], c=random_clusters,
                 cmap=mlearn.cm3, s=60)
axes[0].set_title("zufällige Zuordnung: {:.2f}\n".format(
    silhouette_score(X_scaled, random_clusters)))

algorithms = [KMeans(n_clusters=2), AgglomerativeClustering(n_clusters=2),
              DBSCAN()]

for ax, algorithm in zip(axes[1:], algorithms):
    clusters = algorithm.fit_predict(X_scaled)
    # plotte die Clusterzuordnung und Clustermittelpunkte
    ax.scatter(X_scaled[:, 0], X_scaled[:, 1], c=clusters, cmap=mlearn.cm3, s=60)
```

```
ax.set_title("{} : {:.2f}".format(algorithm.__class__.__name__,
silhouette_score(X_scaled, clusters)))
```

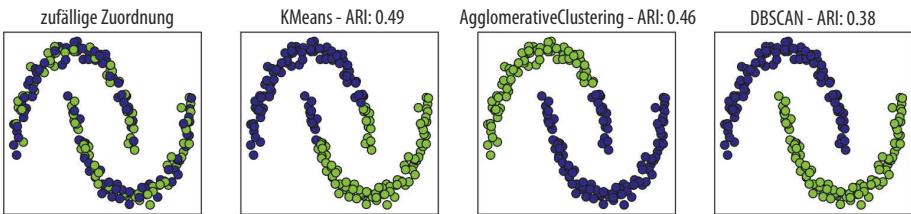


Abbildung 3-40: Vergleich von zufälliger Zuordnung, k-Means, agglomerativem Clustering und DBSCAN auf dem Datensatz two\_moons mit dem Silhouette Score. Das intuitivere Ergebnis von DBSCAN hat einen geringeren Silhouette Score als die Zuordnung durch k-Means.

Wie Sie sehen, erhält *k*-Means den höchsten Silhouette Score, obwohl uns das Ergebnis von DBSCAN vermutlich besser gefällt. Ein etwas besserer Ansatz zum Auswerten von Clustern sind *robustheitsbasierte* Clustering-Metriken. Bei diesen wird ein Algorithmus nach Hinzufügen von ein wenig Rauschen zu den Daten oder mit unterschiedlichen Parametern ausgeführt und das Ergebnis verglichen. Die Grundidee ist, dass es vermutlich vertrauenswürdig ist, wenn viele Parameter des Algorithmus und viele Störungen der Daten das gleiche Ergebnis erzielen. Leider ist dieser Ansatz derzeit in scikit-learn nicht implementiert.

Selbst mit einem sehr robusten Clustering oder einem sehr hohen Silhouette Score wissen wir immer noch nicht, ob mit den Clustern irgendeine semantische Bedeutung verbunden ist oder ob die Cluster einen für uns interessanten Aspekt der Daten widerspiegeln. Kehren wir noch einmal zu den Bildern von Gesichtern zurück. Wir hoffen, Gruppen von ähnlichen Gesichtern zu finden – sagen wir, Gruppen von Männern und Frauen, älteren und jüngeren Menschen oder Menschen mit und ohne Bärten.

Wenn wir die Daten in zwei Cluster einteilen und alle Algorithmen in der Zuordnung der Punkte übereinstimmen, wissen wir immer noch nicht, ob die Cluster mit den Begriffen, an denen wir interessiert sind, etwas zu tun haben. Es könnte sein, dass Seitenansichten von Profilbildern oder Nachtbilder von Tagbildern oder mit iPhone aufgenommene Bilder von denen, die mit Android-Geräten aufgenommen sind, unterschieden wurden. Die einzige Möglichkeit herauszufinden, ob die Cluster mit den für uns interessanten Dingen zu tun haben, ist, die Cluster von Hand zu untersuchen.

### Vergleich von Algorithmen auf dem Datensatz von Gesichtern

Wenden wir nun die Algorithmen *k*-Means, agglomeratives Clustering und DBSCAN auf den Datensatz »Labeled Faces in the Wild« an, um eventuell interessante Strukturen zu entdecken. Wir nehmen dazu die durch Hauptkomponentenzerlegung mit `PCA(whiten=True)` generierte Repräsentation als Eigengesichter mit 100 Komponenten:

**In[69]:**

```
# extrahiere Eigengesichter aus den LFW-Daten und transformiere
from sklearn.decomposition import PCA
pca = PCA(n_components=100, whiten=True, random_state=0)
pca.fit_transform(X_people)
X_pca = pca.transform(X_people)
```

Weiter oben haben wir bereits festgestellt, dass diese Repräsentation der Bilder deutlich inhaltstragender ist als die rohen Pixeldaten. Sie beschleunigt außerdem die Berechnung. Es wäre eine gute Übung, das folgende Experiment auf den Originaldaten ohne PCA auszuführen, um zu sehen, ob ähnliche Cluster dabei herauskommen.

**Analysieren des Gesichter-Datensatzes mit DBSCAN.** Wir beginnen mit dem zuletzt besprochenen Verfahren DBSCAN:

**In[70]:**

```
# wende DBSCAN mit Standardparametern an
dbscan = DBSCAN()
labels = dbscan.fit_predict(X_pca)
print("Eindeutige Bezeichner: {}".format(np.unique(labels)))
```

**Out[70]:**

```
Eindeutige Bezeichner: [-1]
```

Alle produzierten Bezeichner sind -1, DBSCAN hat also die gesamten Daten als »Rauschen« eingestuft. Es gibt zwei Dinge, die wir dagegen tun können: Wir können `eps` erhöhen, um die Nachbarschaft jedes Punktes zu vergrößern, und `min_samples` verringern, um auch kleinere Gruppen von Punkten als Cluster zu betrachten. Verändern wir zunächst `min_samples`:

**In[71]:**

```
dbscan = DBSCAN(min_samples=3)
labels = dbscan.fit_predict(X_pca)
print("Eindeutige Bezeichner: {}".format(np.unique(labels)))
```

**Out[71]:**

```
Eindeutige Bezeichner: [-1]
```

Sogar, wenn wir nur Gruppen mit drei Punkten in Betracht ziehen, wird alles als Rauschen eingestuft. Daher müssen wir `eps` erhöhen:

**In[72]:**

```
dbscan = DBSCAN(min_samples=3, eps=15)
labels = dbscan.fit_predict(X_pca)
print("Eindeutige Bezeichner: {}".format(np.unique(labels)))
```

**Out[72]:**

```
Eindeutige Bezeichner: [-1  0]
```

Auch mit dem viel größeren Wert von 15 für eps erhalten wir nur einen einzigen Cluster und ansonsten Rauschen. Wir können anhand dieses Ergebnisses untersuchen, was das Rauschen vom Rest der Daten unterscheidet. Um besser zu verstehen, was passiert, schauen wir nach, wie viele Punkte Rauschen sind und wie viele sich im Cluster befinden:

In[73]:

```
# zähle die Punkte in allen Clustern und im Rauschen.  
# bincount lässt keine negativen Zahlen zu, addiere daher 1.  
# Die erste Zahl im Ergebnis entspricht dem Rauschen.  
print("Anzahl Punkte pro Cluster: {}".format(np.bincount(labels + 1)))
```

Out[73]:

```
Anzahl Punkte pro Cluster: [ 27 2036]
```

Es gibt sehr wenige verrauschte Punkte (nur 27). Wir können uns diese alle anschauen (siehe Abbildung 3-41):

In[74]:

```
noise = X_people[labels== -1]  
  
fig, axes = plt.subplots(3, 9, subplot_kw={'xticks': (), 'yticks': ()},  
                      figsize=(12, 4))  
for image, ax in zip(noise, axes.ravel()):  
    ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
```



Abbildung 3-41: Von DBSCAN als Rauschen eingestufter Auszug aus dem Gesichter-Datensatz

Vergleichen wir diese Bilder mit der zufälligen Stichprobe von Gesichtern aus Abbildung 3-7, können wir vermuten, warum sie als Rauschen eingestuft wurden: Das fünfte Bild in der ersten Reihe zeigt eine Person, die aus einem Glas trinkt, es gibt Bilder von Leuten mit Hüten, und im letzten Bild ist vor dem Gesicht eine Hand zu sehen. Die übrigen Bilder sind aus schrägen Perspektiven aufgenommen, oder die Bildausschnitte sind zu nah oder weit.

Diese Art der Analyse – die Suche nach »Exoten« – nennt man *Erkennung von Ausreißern*. In einer echten Anwendung könnten wir versuchen, die Bildausschnitte besser zu wählen oder homogenere Daten zu bekommen. Wir können wenig dage-

gen tun, dass die Leute im Bild bisweilen Hüte tragen, trinken oder etwas vor das Gesicht halten, aber es ist gut zu wissen, dass jeder Algorithmus sich mit dieser Art von Daten wird auseinandersetzen müssen.

Wenn wir interessanter als nur einen großen Cluster erhalten möchten, müssen wir eps auf einen Wert zwischen 15 und 0.5 (den Standardwert) verringern. Betrachten wir einmal, was unterschiedliche Werte für eps ergeben:

**In[75]:**

```
for eps in [1, 3, 5, 7, 9, 11, 13]:
    print("\neps={}".format(eps))
    dbscan = DBSCAN(eps=eps, min_samples=3)
    labels = dbscan.fit_predict(X_pca)
    print("Anzahl Cluster: {}".format(np.unique(labels)))
    print("Größe der Cluster: {}".format(np.bincount(labels + 1)))
```

**Out[75]:**

```
eps=1
Anzahl Cluster: [-1]
Größe der Cluster: [2063]

eps=3
Anzahl Cluster: [-1]
Größe der Cluster: [2063]

eps=5
Anzahl Cluster: [-1]
Größe der Cluster: [2063]

eps=7
Anzahl Cluster: [-1  0  1  2  3  4  5  6  7  8  9 10 11 12]
Größe der Cluster: [2006  4  6  6  6  9  3  3  4  3  3  3  3  4]

eps=9
Anzahl Cluster: [-1  0  1  2]
Größe der Cluster: [1269  788     3     3]

eps=11
Anzahl Cluster: [-1  0]
Größe der Cluster: [ 430 1633]

eps=13
Anzahl Cluster: [-1  0]
Größe der Cluster: [ 112 1951]
```

Bei niedrigen Werten für eps werden alle Punkte als Rauschen eingestuft. Mit eps=7 erhalten wir viel Rauschen und mehrere kleine Cluster. Mit eps=9 erhalten wir noch immer viel Rauschen, aber auch einen großen und mehrere kleinere Cluster. Ab eps=11 erhalten wir nur noch einen großen Cluster und Rauschen.

Es ist interessant, hervorzuheben, dass es nie mehr als einen großen Cluster gibt. Es gibt höchstens einen großen Cluster mit einem Großteil der Punkte und einige kleinere Cluster. Das deutet darauf hin, dass es nicht zwei oder drei charakteristische

Arten von Gesichtern gibt, sondern dass alle Bilder mehr oder weniger ähnlich (oder unähnlich) zu den übrigen sind.

Die Ergebnisse mit  $\text{eps}=7$  sehen wegen der vielen kleinen Cluster am vielversprechendsten aus. Diese Clusterung können wir im Detail erkunden, indem wir alle Punkte in den 13 kleinen Clustern visualisieren (Abbildung 3-42):



Abbildung 3-42: Von DBSCAN mit  $\text{eps}=7$  gefundene Cluster

In[76]:

```
dbSCAN = DBSCAN(min_samples=3, eps=7)
labels = dbSCAN.fit_predict(X_pca)

for cluster in range(max(labels) + 1):
    mask = labels == cluster
    n_images = np.sum(mask)
    fig, axes = plt.subplots(1, n_images, figsize=(n_images * 1.5, 4),
                           subplot_kw={'xticks': (), 'yticks': ()})
    for image, label, ax in zip(X_people[mask], y_people[mask], axes):
        ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
        ax.set_title(people.target_names[label].split()[-1])
```

Einige der Cluster entsprechen Personen mit sehr charakteristischen Gesichtern (innerhalb dieses Datensatzes), z. B. Sharon oder Koizumi. Innerhalb eines Clusters sind die Blickrichtung und der Gesichtsausdruck in etwa gleich. Einige Cluster enthalten Bilder von mehreren Personen, aber mit ähnlicher Ausrichtung und Gesichtsausdruck.

Damit beenden wir unsere Analyse des DBSCAN-Algorithmus auf dem Gesichter-Datensatz. Wie Sie sehen können, führen wir eine Untersuchung von Hand durch, im Gegensatz zur automatischen Suche beim überwachten Lernen mit  $R^2$ -Score oder Genauigkeit.

Fahren wir mit der Anwendung von  $k$ -Means und agglomerativem Clustering fort.

**Analyse des Gesichter-Datensatzes mit  $k$ -Means.** Wir haben gesehen, dass DBSCAN nicht mehr als einen großen Cluster erzeugen konnte. Agglomeratives Clustering und  $k$ -Means sind eher prädestiniert, gleich große Cluster zu generieren. Allerdings müssen wir die gewünschte Anzahl Cluster festlegen. Wir könnten die Anzahl Personen im Datensatz als Clusteranzahl festlegen. Es ist aber unwahrscheinlich, dass ein unüberwachter Clustering-Algorithmus diese auch findet. Stattdessen beginnen wir mit einer geringen Anzahl Cluster, z. B. zehn, wodurch wir uns jeden der Cluster genau ansehen können:

In[77]:

```
# extrahiere Cluster mit k-Means
km = KMeans(n_clusters=10, random_state=0)
labels_km = km.fit_predict(X_pca)
print("Cluster-Größen mit k-Means: {}".format(np.bincount(labels_km)))
```

Out[77]:

```
Cluster-Größen mit k-Means: [269 128 170 186 386 222 237 64 253 148]
```

Wie Sie sehen, teilt  $k$ -Means die Daten in etwa ähnlich große Cluster mit 64 bis 386 Bildern ein. Dieses Ergebnis unterscheidet sich stark von DBSCAN.

Wir können das Ergebnis von  $k$ -Means weiter analysieren, indem wir die Clustermittelpunkte visualisieren (Abbildung 3-43). Da wir die über Hauptkomponentenzerlegung berechnete Repräsentation geclustert haben, müssen wir die Clustermittel-

punkte zur Visualisierung mit `pca.inverse_transform` wieder in den ursprünglichen Raum überführen:

In[78]:

```
fig, axes = plt.subplots(2, 5, subplot_kw={'xticks': (), 'yticks': ()},  
                        figsize=(12, 4))  
for center, ax in zip(km.cluster_centers_, axes.ravel()):  
    ax.imshow(pca.inverse_transform(center).reshape(image_shape),  
              vmin=0, vmax=1)
```



Abbildung 3-43: Von  $k$ -Means gefundene Clustermittelpunkte bei einer Clusteranzahl von 10

Die von  $k$ -Means gefundenen Clustermittelpunkte sind sehr weiche Varianten von Gesichtern. Dies ist nicht sehr überraschend, da jeder Mittelpunkt den Durchschnitt von 64 bis 386 Bildern darstellt. Eine durch PCA reduzierten Repräsentation erhöht den Weichheitsgrad der Bilder (im Vergleich zu den mit 100 Dimensionen rekonstruierten Bildern in Abbildung 3-11). Die Cluster scheinen unterschiedliche Orientierungen der Gesichter zu erfassen, unterschiedliche Gesichtsausdrücke (der dritte Cluster scheint ein Lächeln zu enthalten) oder das Vorhandensein von Hemdkragen (im vorletzten Clustermittelpunkt).

Um noch mehr ins Detail zu gehen, betrachten wir in Abbildung 3-44 fünf typische Bilder pro Cluster (die fünf Bilder, die in diesem Cluster dem Mittelpunkt am nächsten sind) sowie fünf untypische Bilder (die in diesem Cluster am weitesten vom Clustermittelpunkt entfernten Bilder):

In[79]:

```
mglearn.plots.plot_kmeans_faces(km, pca, X_pca, X_people,  
                                  y_people, people.target_names)
```

Abbildung 3-44 bestätigt unsere Vermutung zu den lächelnden Gesichtern im dritten Cluster und die Wichtigkeit der Ausrichtung bei den anderen Clustern. Die »untypischen« Punkte sehen den Clustermittelpunkten allerdings nicht besonders ähnlich, und deren Zuordnung scheint etwas willkürlich.

Dies könnte daran liegen, dass  $k$ -Means grundsätzlich alle Datenpunkte zuordnet und kein »Rauschen« wie DBSCAN kennt. Mit einer größeren Anzahl Cluster könnte der Algorithmus eine feinere Unterscheidung treffen. Allerdings macht eine größere Anzahl Cluster die manuelle Auswertung noch schwieriger.



Abbildung 3-44: Von k-Means gefundene Bilder für jeden Cluster – die Clustermittelpunkte befinden sich auf der linken Seite, daneben die fünf dem Mittelpunkt nächsten Bilder in diesem Cluster sowie die fünf Bilder in diesem Cluster, die am weitesten vom Mittelpunkt entfernt sind.

**Analyse des Gesichter-Datensatzes mit agglomerativem Clustering.** Betrachten wir nun die Ergebnisse für das agglomerative Clustering:

**In[80]:**

```
# extrahiere Cluster mit dem agglomerativen Clustering nach ward
agglomerative = AgglomerativeClustering(n_clusters=10)
labels_agg = agglomerative.fit_predict(X_pca)
print("Cluster-Größen für das agglomerative Clustering: {}".format(
    np.bincount(labels_agg)))
```

**Out[80]:**

```
Cluster-Größen für das agglomerative Clustering:
[255 623  86 102 122 199 265  26 230 155]
```

Auch das agglomerative Clustering berechnet ähnlich große Cluster mit Größen zwischen 26 und 623. Die Größe ist weniger gleichmäßig verteilt als bei  $k$ -Means, aber viel gleichmäßiger als bei DBSCAN.

Wir können den ARI berechnen, um zu bestimmen, inwieweit die Unterteilung der Daten durch agglomeratives Clustering und  $k$ -Means übereinstimmen:

**In[81]:**

```
print("ARI: {:.2f}".format(adjusted_rand_score(labels_agg, labels_km)))
```

**Out[81]:**

```
ARI: 0.13
```

Ein ARI von 0.13 bedeutet, dass die zwei Clusterungen `labels_agg` und `labels_km` wenig gemeinsam haben. Dies ist nicht besonders erstaunlich, da die vom Clustermittelpunkt weiter entfernten Punkte bei  $k$ -Means nicht sehr ähnlich sind.

Als Nächstes zeichnen wir ein Dendrogramm (Abbildung 3-45). Wir begrenzen die Länge des Baumes im Plot, da eine Verzweigung bis hin zu den 2063 einzelnen Datenpunkten einen unleserlich dichten Plot erzeugen würde:

**In[82]:**

```
linkage_array = ward(X_pca)
# wir zeichnen ein Dendrogramm für das linkage_array
# mit dem Abstand zwischen Clustern
plt.figure(figsize=(20, 5))
dendrogram(linkage_array, p=7, truncate_mode='level', no_labels=True)
plt.xlabel("Index")
plt.ylabel("Cluster-Abstand")
```

Zum Generieren von zehn Clustern durchschneiden wir den Baum weit oben, wo es zehn vertikale Linien gibt. Im Dendrogramm für den synthetischen Datensatz in Abbildung 3-36 konnten wir anhand der Länge der Verzweigungen sehen, dass zwei oder drei Cluster die Daten angemessen wiedergeben würden. Bei den Gesichtern scheint es solch einen natürlichen Schnittpunkt nicht zu geben. Einige Äste stellen klarer abgegrenzte Gruppen dar, aber es gibt keine bestimmte Gruppe von Clustern, die besonders gut passt. Dies ist angesichts der Ergebnisse von DBSCAN nicht überraschend, da dort alle Punkte geballt in einem Cluster auftraten.

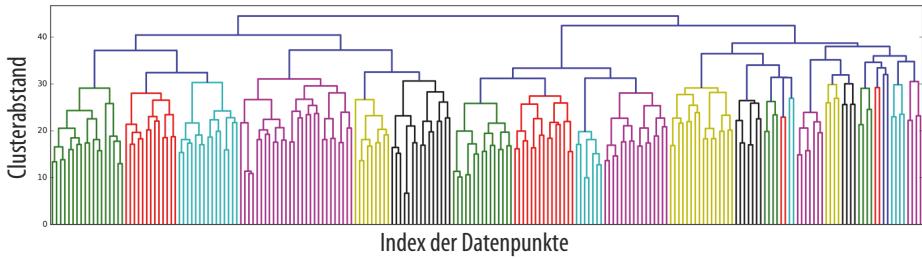


Abbildung 3-45: Dendrogramm für das agglomerative Clustering auf dem Gesichter-Datensatz

Wir visualisieren die zehn Cluster wie zuvor bei  $k$ -Means (Abbildung 3-46). Bedenken Sie, dass es beim agglomerativen Clustering keine Clustermittelpunkte gibt (obwohl wir den Mittelwert berechnen könnten) und wir einfach nur die ersten paar Bilder aus jedem Cluster ausgeben. Links vom ersten Bild ist die Anzahl Punkte pro Cluster angezeigt:

In[83]:

```
n_clusters = 10
for cluster in range(n_clusters):
    mask = labels_agg == cluster
    fig, axes = plt.subplots(1, 10, subplot_kw={'xticks': (), 'yticks': ()},
                           figsize=(15, 8))
    axes[0].set_ylabel(np.sum(mask))
    for image, label, asdf, ax in zip(X_people[mask], y_people[mask],
                                      labels_agg[mask], axes):
        ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
        ax.set_title(people.target_names[label].split()[-1], fontdict={'fontsize': 9})
```

Auch wenn einige der Cluster ein inhaltliches Thema zu haben scheinen, sind viele schlicht zu groß, um homogen zu sein. Um homogenere Cluster zu erhalten, führen wir den Algorithmus erneut aus, diesmal mit 40 Clustern. Wir wählen einige der interessanteren Cluster aus (Abbildung 3-47):

In[84]:

```
# extrahiere Cluster mit dem agglomerativen Clustering nach ward
agglomerative = AgglomerativeClustering(n_clusters=40)
labels_agg = agglomerative.fit_predict(X_pca)
print("Cluster-Größen für agglomeratives Clustering: {}".format(np.bincount(labels_agg)))

n_clusters = 40
for cluster in [10, 13, 19, 22, 36]: # von Hand ausgewählte Cluster
    mask = labels_agg == cluster
    fig, axes = plt.subplots(1, 15, subplot_kw={'xticks': (), 'yticks': ()},
                           figsize=(15, 8))
    cluster_size = np.sum(mask)
    axes[0].set_ylabel("#{}: {}".format(cluster, cluster_size))
    for image, label, asdf, ax in zip(X_people[mask], y_people[mask],
                                      labels_agg[mask], axes):
        ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
        ax.set_title(people.target_names[label].split()[-1], fontdict={'fontsize': 9})
    for i in range(cluster_size, 15):
        axes[i].set_visible(False)
```



Abbildung 3-46: Zufällig ausgewählte Bilder aus den mit In[82] erzeugten Clustern – jede Zeile entspricht einem Cluster; die Zahl links gibt die Anzahl Bilder im Cluster an.

**Out[84]:**

Cluster-Größen für agglomeratives Clustering:

```
[ 58  80  79  40 222  50  55  78 172  28  26  34  14  11  60  66 152  27
 47  31  54   5   8  56   3   5   8  18  22  82  37  89  28  24  41  40
 21  10 113  69]
```



Abbildung 3-47: Bilder ausgewählter Cluster, die mit agglomerativem Clustering mit auf 40 gesetzter Clusteranzahl generiert wurden. Der Text auf der linken Seite zeigt den Index des Clusters und die Gesamtzahl Punkte im Cluster an.

Das Clustering hat diesmal auf »dunkelhäutig und lächelnd«, »Hemd mit Kragen«, »lächelnde Frau«, »Hussein« und »hohe Stirn« angeschlagen. Wir könnten diese stark ähnlichen Cluster bei einer genaueren Untersuchung auch im Dendrogramm finden.

## Zusammenfassung der Clustering-Methoden

Dieser Abschnitt hat gezeigt, dass die Anwendung und Auswertung von Clusterverfahren ein in hohem Maße qualitativer Vorgang ist, der in der Erkundungsphase einer Datenanalyse besonders nützlich ist. Wir haben drei Clustering-Algorithmen genauer betrachtet:  $k$ -Means, DBSCAN und agglomeratives Clustering. Bei allen dreien lässt sich die Granularität der Clusterung einstellen. Bei  $k$ -Means und agglomerativem Clustering können wir die Anzahl gewünschter Cluster direkt angeben,

bei DBSCAN dagegen können wir Nähe über den Parameter `eps` einstellen, was die Clustergröße indirekt beeinflusst. Alle drei Methoden lassen sich auf große realistische Datensätze anwenden, sind relativ leicht zu verstehen und erlauben die Zuordnung vieler Cluster.

Jeder der Algorithmen hat etwas andere Stärken. *k*-Means erlaubt die Charakterisierung von Clustern über ihre Mittelpunkte. Dieser Algorithmus lässt sich auch als Verfahren zur Dekomposition auffassen, wobei jeder Datenpunkt durch seinen Clustermittelpunkt repräsentiert wird. Mit DBSCAN können wir »Rauschen« erkennen, Punkte, die keinem Cluster zugeordnet sind, oder die Anzahl Cluster ermitteln. Im Gegensatz zu den zwei übrigen Methoden erkennt dieser Algorithmus komplexe Formen von Clustern, wie wir beim Beispiel `two_moons` gesehen haben. DBSCAN erzeugt bisweilen Cluster mit sehr unterschiedlicher Größe, was sowohl eine Stärke als auch eine Schwäche sein kann. Agglomeratives Clustering schließlich liefert uns eine komplette Hierarchie möglicher Aufteilungen der Daten, die sich leicht als Dendrogramm inspizieren lässt.

## Zusammenfassung und Ausblick

In diesem Kapitel wurde eine Anzahl Algorithmen für unüberwachtes Lernen eingeführt, die sich für die erkundende Datenanalyse und Vorverarbeitung eignet. Für den Erfolg überwachten und unüberwachten Lernens ist oft die richtige Repräsentation der Daten entscheidend, und daher spielen Vorverarbeitung und Zerlegung eine wichtige Rolle bei der Datenaufbereitung.

Zerlegung, Manifold Learning und Clustering sind wichtige Werkzeuge, um das Verständnis Ihrer Daten voranzutreiben. Bei Abwesenheit von übergeordneter Information können dies sogar die einzigen Wege sein, Sinn in Ihren Daten zu finden. Sogar bei überwachtem Lernen sind erkundende Werkzeuge wichtig, um die Eigenschaften der Daten besser zu verstehen. Es ist häufig schwierig, den Nutzen eines unüberwachten Algorithmus in Zahlen zu fassen. Das sollte Sie aber nicht davon abhalten, diesen zum Sammeln von Erkenntnissen über Ihre Daten einzusetzen.

Mit diesen Methoden in Ihrer Werkzeugkiste sind Sie nun mit den wesentlichen Lernalgorithmen ausgestattet, die Praktiker täglich beim maschinellen Lernen einsetzen.

Wir ermuntern Sie, Methoden zum Clustern und Zerlegen sowohl auf zweidimensionale Beispieldaten als auch auf echten Datensätzen in `scikit-learn` auszuprobieren, wie den Datensätzen `digits`, `iris` und `cancer`.

## Zusammenfassung der Estimator-Schnittstelle

Fassen wir kurz die Erläuterungen zu der in den Kapiteln 2 und 3 eingeführten Schnittstelle zusammen. Alle Algorithmen in scikit-learn, ob zur Vorverarbeitung, überwachten oder unüberwachten Lernalgorithmen, sind als Klassen implementiert. Diese Klassen nennt man in scikit-learn Estimatoren. Um einen Algorithmus anzuwenden, müssen Sie zunächst ein Objekt dieser Klasse instanzieren:

In[85]:

```
from sklearn.linear_model import LogisticRegression  
logreg = LogisticRegression()
```

Die Estimator-Klasse enthält den Algorithmus und speichert auch das anhand der Daten mit diesem Algorithmus trainierte Modell.

Sie sollten jegliche Parameter des Modells festlegen, sobald Sie das Objekt erstellen. Diese Parameter umfassen Regularisierung, Kontrolle von Komplexität, Anzahl zu findender Cluster usw. Alle Estimatoren besitzen die Methode `fit`, mit der das Modell erstellt wird. Die Methode `fit` erfordert als ersten Parameter stets die Daten `X` als NumPy-Array oder als dünn besetzte Matrix aus SciPy, wobei jede Zeile einen Datenpunkt darstellt. Es wird davon ausgegangen, dass die Daten `X` kontinuierliche Werte (Fließkommazahlen) enthalten. Überwachte Algorithmen erfordern auch den Parameter `y`, ein eindimensionales NumPy-Array mit den Zielvorgaben für die Regression oder Klassifikation (also die bekannten Markierungen für die Ausgabe oder Antworten).

Es gibt zwei Möglichkeiten, ein mit scikit-learn trainiertes Modell anzuwenden. Um eine Vorhersage als neue Ausgabe wie `y` zu erstellen, setzen Sie die Methode `predict` ein. Um eine neue Repräsentation der Eingabedaten `X` zu erzeugen, verwenden Sie die Methode `transform`. In Tabelle 3-1 sind die Anwendungsfälle der Methoden `predict` und `transform` zusammengefasst.

Tabelle 3-1: Zusammenfassung der scikit-learn API

<code>estimator.fit(x_train, [y_train])</code>	
<code>estimator.predict(X_test)</code>	<code>estimator.transform(X_test)</code>
Klassifikation	Vorverarbeitung
Regression	Dimensionsreduktion
Clustering	Extrahieren von Eigenschaften
	Auswahl von Eigenschaften

Außerdem besitzen alle überwachten Modelle die Methode `score(X_test, y_test)`, die eine Bewertung des Modells erlaubt. In Tabelle 3-1 beziehen sich `X_train` und `y_train` auf die Trainingsdaten und deren Bezeichner, und `X_test` sowie `y_test` beziehen sich auf die Testdaten und deren Bezeichner (sofern anwendbar).

# Repräsentation von Daten und Merkmalsgenerierung

Bisher sind wir davon ausgegangen, dass unsere Daten als zweidimensionale Arrays von Fließkommazahlen vorliegen, wobei jede Spalte ein *kontinuierliches Merkmal* beschreibt. Bei vielen Anwendungen werden Daten aber nicht auf diese Weise gesammelt. *Kategorische Merkmale* kommen besonders häufig vor. Diese auch *discrete Merkmale* genannten Eigenschaften sind für gewöhnlich keine Zahlen. Der Unterschied zwischen kategorischen und kontinuierlichen Merkmalen ist analog zum Unterschied zwischen Klassifikation und Regression, nur aufseiten der Eingabedaten und nicht bei der Ausgabe. Bereits betrachtete Beispiele für kontinuierliche Merkmale sind die Helligkeit von Pixeln und Messungen der Größe von Pflanzenteilen. Beispiele für kategorische Merkmale sind die Marke eines Produkts, dessen Farbe oder die verkaufende Abteilung (Bücher, Bekleidung, Eisenwaren). Alle diese Eigenschaften beschreiben einen Gegenstand, aber variieren nicht kontinuierlich. Ein Produkt gehört entweder in die Bekleidungsabteilung oder in die Bücherabteilung. Es gibt keine Grauzone zwischen Büchern und Bekleidung und keine natürliche Reihenfolge der unterschiedlichen Kategorien (Bücher sind nicht größer als Bekleidung, Eisenwaren liegen nicht zwischen Büchern und Bekleidung usw.).

Unabhängig von der Art der Merkmale, aus denen Ihre Daten bestehen, kann es enorme Auswirkungen auf die Leistung von maschinellen Lernmodellen haben, wie Sie Ihre Daten repräsentieren. Wir haben in den Kapiteln 2 und 3 gesehen, dass die Skalierung der Daten wichtig ist. Anders gesagt, wenn Sie Ihre Daten nicht umskalieren (z. B. auf eine Varianz von 1), macht es einen Unterschied, ob Sie eine Messung in Zentimetern oder Zoll vornehmen. Wir haben außerdem in Kapitel 2 erlebt, dass das *Ergänzen* von Daten durch zusätzliche Merkmale wie Interaktionen (Produkte) von Merkmalen oder allgemein Polynomialtermen hilfreich sein kann.

Die Frage, wie Sie Ihre Daten am besten für eine bestimmte Anwendung repräsentieren sollen, nennt man *Merkmalsgenerierung (feature engineering)*, eine der Hauptbeschäftigungen von Data Scientists und Anwendern maschineller Lernmethoden beim Bearbeiten realer Aufgabenstellungen. Die richtige Repräsentation Ihrer Daten kann einen größeren Einfluss auf die Leistung eines überwachten Modells haben als die von Ihnen gewählten Modellparameter.

In diesem Kapitel werden wir uns zuerst mit den wichtigen und sehr häufigen kategorischen Merkmalen beschäftigen und anschließend einige Beispiele für hilfreiche Transformationen und spezielle Kombinationen von Merkmalen und Modellen betrachten.

## Kategoriale Variablen

Als Beispiel verwenden wir einen Datensatz der Einkommen von Erwachsenen in den Vereinigten Staaten, der aus der Volkszählung von 1994 stammt. Die Aufgabe zum Datensatz `adult` ist, vorherzusagen, ob ein Arbeitnehmer ein Einkommen über oder unter 50000 USD besitzt. Die Merkmale in diesem Datensatz enthalten das Alter der Personen, die Art der Beschäftigung (selbstständig, in der Wirtschaft angestellt, von der Regierung beschäftigt usw.), Bildungsstand, Geschlecht, Arbeitsstunden pro Woche und viele weitere. Tabelle 4-1 zeigt die ersten Einträge aus diesem Datensatz.

Tabelle 4-1: Die ersten Einträge im Datensatz `adult`

	<code>age</code>	<code>workclass</code>	<code>education</code>	<code>gender</code>	<code>hours-per-week</code>	<code>occupation</code>	<code>income</code>
0	39	State-gov	Bachelors	Male	40	Adm-clerical	<=50K
1	50	Self-emp-not-inc	Bachelors	Male	13	Exec-managerial	<=50K
2	38	Private	HS-grad	Male	40	Handlers-cleaners	<=50K
3	53	Private	11th	Male	40	Handlers-cleaners	<=50K
4	28	Private	Bachelors	Female	40	Prof-specialty	<=50K
5	37	Private	Masters	Female	40	Exec-managerial	<=50K
6	49	Private	9th	Female	16	Other-service	<=50K
7	52	Self-emp-not-inc	HS-grad	Male	45	Exec-managerial	>50K
8	31	Private	Masters	Female	50	Prof-specialty	>50K
9	42	Private	Bachelors	Male	40	Exec-managerial	>50K
10	37	Private	Some-college	Male	80	Exec-managerial	>50K

Die Aufgabe ist eine Klassifizierung mit den zwei Einkommensklassen  $\leq 50\text{k}$  und  $> 50\text{k}$ . Es wäre auch möglich, das genaue Einkommen vorherzusagen und somit hieraus eine Regressionsaufgabe zu machen. Allerdings wäre das weitaus schwieriger, und die Unterscheidung bei 50000 zu verstehen, ist bereits interessant genug.

In diesem Datensatz sind `age` und `hours-per-week` kontinuierliche Merkmale, mit denen wir bereits umgehen können. Dagegen sind `workclass`, `education`, `gender` und `occupation` kategoriale Merkmale. Für jedes existiert eine festgelegte Liste möglicher Werte anstatt eines Wertebereichs. Sie umschreiben eine qualitative Eigenschaft anstatt einer quantitativen.

Als Ausgangspunkt werden wir einen Klassifikator mit logistischer Regression auf diesen Daten trainieren. Wir wissen aus Kapitel 2 bereits, dass die logistische Regression Vorhersagen  $\hat{y}$  mit folgender Formel berechnet:

$$\hat{y} = w[0]*x[0] + w[1]*x[1] + \dots + w[p]*x[p] + b > 0$$

Dabei sind  $w[i]$  und  $b$  die anhand der Trainingsdaten ermittelten Koeffizienten, und  $x[i]$  sind die Eingabemerkmale. Diese Formel ist dann sinnvoll, wenn  $x[i]$  Zahlen beinhaltet, aber nicht, wenn  $x[2]$  Werte wie "Masters" oder "Bachelors" enthält. Um die logistische Regression einsetzen zu können, müssen wir unsere Daten anders repräsentieren. Der nächste Abschnitt erklärt, wie wir mit diesem Problem umgehen können.

## One-Hot-Kodierung (Dummy-Variablen)

Die verbreitetste Möglichkeit, kategorische Variablen zu repräsentieren, ist die *One-Hot-Kodierung* oder *1-aus-N-Kodierung*, auch bekannt als *Dummy-Variablen*. Die Idee bei Dummy-Variablen ist, eine kategorische Variable durch ein oder mehrere Merkmale mit den Werten 0 und 1 zu ersetzen. Die Werte 0 und 1 lassen sich sinnvoll in die Formel für lineare binäre Klassifikation einsetzen (und in sämtliche Modelle in scikit-learn). Wir beschreiben an dieser Stelle, wie sich eine beliebige Anzahl Kategorien durch Einführen eines neuen Merkmals pro Kategorie repräsentieren lässt.

Nehmen wir an, es gäbe für das Merkmal `workclass` als mögliche Werte "Government Employee", "Private Employee", "Self Employed" und "Self Employed Incorporated". Um diese vier Werte zu kodieren, müssen wir vier neue Merkmale namens "Government Employee", "Private Employee", "Self Employed" und "Self Employed Incorporated" erstellen. Falls `workclass` für eine Person den entsprechenden Wert hat, ist das Merkmal 1, andernfalls 0. Also ist genau eines der vier neuen Merkmale für einen Datensatz gleich 1. Aus diesem Grund wird diese Methode One-Hot- oder 1-aus-N-Kodierung genannt.

In Tabelle 4-2 ist das Grundprinzip dargestellt. Ein einzelnes Merkmal wird durch vier neue Merkmale kodiert. Beim Verwenden dieser Daten in einem maschinellen Lernverfahren könnten wir das ursprüngliche Merkmal `workclass` löschen und lediglich die binären Merkmale übrig lassen.

Tabelle 4-2: Kodieren des Merkmals 'workclass' mittels One-Hot-Kodierung

<code>workclass</code>	<code>Government Employee</code>	<code>Private Employee</code>	<code>Self Employed</code>	<code>Self Employed Incorporated</code>
Government Employee	1	0	0	0
Private Employee	0	1	0	0
Self Employed	0	0	1	0
Self Employed Incorporated	0	0	0	1



Die von uns verwendete One-Hot-Kodierung ist recht ähnlich zur in der Statistik verwendeten Dummy-Kodierung, aber nicht identisch. Der Einfachheit halber kodieren wir jede Kategorie als ein eigenes binäres Merkmal. In der Statistik ist es dagegen üblich, ein kategorisches Merkmal mit  $k$  möglichen Werten als  $k-1$  Merkmale zu kodieren (wobei das letzte durch lauter Nullen ausgedrückt wird). Dies tut man, um die Analyse zu vereinfachen (genauer, damit stellt man sicher, dass die Matrix vollen Rang besitzt).

Es gibt zwei Möglichkeiten, Daten mit kategorischen Variablen zu One-Hot-kodierten umzuwandeln. Eine verwendet `pandas`, die andere `scikit-learn`. Zum gegenwärtigen Zeitpunkt ist die Verwendung von `pandas` etwas einfacher, also schlagen wir diesen Weg ein. Zuerst lassen wir `pandas` die Daten aus einer Komma-separierten Datei (CSV) laden:

**In[1]:**

```
import pandas as pd
# Die Datei hat keine Kopfzeilen mit Spaltennamen, daher
# übergeben wir header=None und geben die Namen der Spalten
# explizit durch "names" an.
data = pd.read_csv(
    "data/adult.data", header=None, index_col=False,
    names=['age', 'workclass', 'fnlwgt', 'education', 'education-num',
           'marital-status', 'occupation', 'relationship', 'race', 'gender',
           'capital-gain', 'capital-loss', 'hours-per-week', 'native-country',
           'income'])
# Zur Veranschaulichung verwenden wir nur einige der Spalten
data = data[['age', 'workclass', 'education', 'gender', 'hours-per-week',
             'occupation', 'income']]
# IPython.display formatiert die Ausgabe im Jupyter notebook
display(data.head())
```

Tabelle 4-3 zeigt das Ergebnis.

*Tabelle 4-3: Die ersten fünf Zeilen des Datensatzes adult*

	age	workclass	education	gender	hours-per-week	occupation	income
0	39	State-gov	Bachelors	Male	40	Adm-clerical	<=50K
1	50	Self-emp-not-inc	Bachelors	Male	13	Exec-managerial	<=50K
2	38	Private	HS-grad	Male	40	Handlers-cleaners	<=50K
3	53	Private	11th	Male	40	Handlers-cleaners	<=50K
4	28	Private	Bachelors	Female	40	Prof-specialty	<=50K

## Prüfen von String-kodierten kategorischen Daten

Nach dem Einlesen eines Datensatzes mit dieser Methode ist es oft sinnvoll zu prüfen, ob eine Spalte überhaupt bedeutsame kategoriale Daten enthält. Falls Sie mit von Menschen eingegebenen Daten arbeiten (z. B. Nutzern einer Webseite), gibt es womöglich keinen festen Satz Kategorien, und Unterschiede in der Schreibweise

und Groß-/Kleinschreibung machen eine Vorverarbeitung nötig. Es ist zum Beispiel möglich, dass manche Leute ihr Geschlecht als »männlich«, andere als »Mann« angeben, obwohl wir beide Eingaben der gleichen Kategorie zuordnen möchten. Wir können den Inhalt einer Spalte mit der Funktion `value_counts` einer Series in pandas (dem Datentyp einer Spalte eines DataFrame) untersuchen, um die unterschiedlichen Werte und ihre Häufigkeiten anzusehen:

**In[2]:**

```
print(data.gender.value_counts())
```

**Out[2]:**

```
Male      21790  
Female    10771  
Name: gender, dtype: int64
```

Wir sehen, dass es in diesem Datensatz genau zwei Werte für 'gender' gibt, nämlich Male und Female. Das bedeutet, dass sich die Daten bereits in einem guten Format zur Repräsentation durch One-Hot-Kodierung befinden. In einer wirklichen Anwendung sollten Sie sich die Werte sämtlicher Spalten ansehen, was wir hier der Kürze halber überspringen.

Mit der Funktion `get_dummies` haben wir eine sehr einfache Möglichkeit, die Daten in pandas zu kodieren. Die Funktion `get_dummies` wandelt automatisch alle Spalten um, die als Typ ein Objekt (wie Strings) besitzen oder Kategorien sind (ein spezielles Konzept in pandas, über das wir noch nicht gesprochen haben):

**In[3]:**

```
print("Ursprüngliche Merkmale:\n", list(data.columns), "\n")  
data_dummies = pd.get_dummies(data)  
print("Merkmale nach get_dummies:\n", list(data_dummies.columns))
```

**Out[3]:**

```
Ursprüngliche Merkmale:  
['age', 'workclass', 'education', 'gender', 'hours-per-week', 'occupation',  
'income']  
  
Merkmale nach get_dummies:  
['age', 'hours-per-week', 'workclass_?', 'workclass_Federal-gov',  
'workclass_Local-gov', 'workclass_Never-worked', 'workclass_Private',  
'workclass_Self-emp-inc', 'workclass_Self-emp-not-inc',  
'workclass_State-gov', 'workclass_Without-pay', 'education_10th',  
'education_11th', 'education_12th', 'education_1st-4th',  
...  
'education_Preschool', 'education_Prof-school', 'education_Some-college',  
'gender_Female', 'gender_Male', 'occupation_?',  
'occupation_Adm-clerical', 'occupation_Armed-Forces',  
'occupation_Craft-repair', 'occupation_Exec-managerial',  
'occupation_Farming-fishing', 'occupation_Handlers-cleaners',  
...  
'occupation_Tech-support', 'occupation_Transport-moving',  
'income_<=50K', 'income_>50K']
```

Sie können sehen, dass die kontinuierlichen Merkmale `age` und `hours-per-week` nicht angerührt wurden, die kategorischen Merkmale aber um je ein neues Merkmal pro möglichem Wert erweitert wurden:

**In[4]:**

```
data_dummies.head()
```

**Out[4]:**

	hours-age	per-week	workclass_?	workclass_Federal-gov	workclass_Local-gov	...	occupation_Tech-support	occupation_Transport-moving	income_<=50K	income_>50K
0	39	40	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0
1	50	13	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0
2	38	40	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0
3	53	40	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0
4	28	40	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0

5 rows × 46 columns

Wir können nun das Attribut `values` verwenden, um das DataFrame `data_dummies` in ein NumPy-Array zu überführen und anschließend ein maschinelles Lernmodell darauf zu trainieren. Denken Sie daran, die Zielvariable (nun in den zwei Spalten `income` kodiert) von den Daten vor dem Trainieren eines Modells zu trennen. Eine AusgabevARIABLE oder ein davon abgeleitetes Merkmal in die repräsentierten Merkmale aufzunehmen, ist ein sehr verbreiteter Fehler beim Erstellen von überwachten maschinellen Lernmodellen.



Seien Sie vorsichtig: Die Spaltenindizierung in pandas nimmt das Ende des angegebenen Intervalls mit auf. Daher enthält 'age': 'occupation\_Transport-moving' auch `occupation_Transport-moving`. Diese Art des Slicing unterscheidet sich von Arrays in NumPy, bei denen der Endpunkt eines Intervalls nicht verwendet wird: Zum Beispiel enthält das Array `np.arange(11)[0:10]` den Eintrag mit dem Index 10 nicht.

In diesem Fall extrahieren wir lediglich die Spalten mit Merkmalen – also sämtliche Spalten von `age` bis `occupation_Transport-moving`. Dieses Intervall enthält alle Merkmale, nicht aber die Zielspalte:

**In[5]:**

```
features = data_dummies.ix[:, <'age': 'occupation_Transport-moving']
# Extrahiere NumPy arrays
X = features.values
y = data_dummies['income_>50K'].values
print("X.shape: {} y.shape: {}".format(X.shape, y.shape))
```

## Out[5]:

```
X.shape: (32561, 44) y.shape: (32561,)
```

Nun liegen die Daten in einer Repräsentation vor, mit der scikit-learn arbeiten kann, und wir gehen auf die bekannte Weise vor:

## In[6]:

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
logreg = LogisticRegression()
logreg.fit(X_train, y_train)
print("Test Score: {:.2f}".format(logreg.score(X_test, y_test)))
```

## Out[6]:

```
Test Score: 0.81
```



In diesem Beispiel haben wir die Funktion `get_dummies` auf einem DataFrame mit Trainings- und Testdaten aufgerufen. Es ist wichtig sicherzustellen, dass kategorische Daten sowohl im Trainings- als auch im Testdatensatz auf die gleiche Art und Weise repräsentiert sind.

Stellen Sie sich vor, wir hätten die Trainings- und die Testdaten in zwei unterschiedlichen DataFrames abgelegt. Falls der Wert "Private Employee" für das Merkmal `workclass` nicht im Testdatensatz auftritt, nimmt pandas an, dass es nur drei mögliche Werte für dieses Merkmal gibt, und erstellt dementsprechend nur drei neue Dummy-Merkmale. Dann hätten unser Trainings- und Testdatensatz unterschiedliche Merkmale, und wir könnten das auf den Trainingsdaten erstellte Modell nicht auf die Testdaten anwenden. Noch schlimmer wird es, falls das Merkmal `workclass` im Trainingsdatensatz die Werte "Government Employee" und "Private Employee" enthält, der Testdatensatz aber "Self Employed" und "Self Employed Incorporated". In beiden Fällen generiert pandas zwei Dummy-Merkmale, sodass die kodierten DataFrames die gleiche Anzahl Merkmale haben. Allerdings haben die zwei Dummy-Merkmale im Trainings- und im Testdatensatz eine komplett unterschiedliche Bedeutung. Die Spalte, die in den Trainingsdaten für "Government Employee" steht, würde im Testdatensatz plötzlich für "Self Employed" kodieren.

Wenn wir auf diesen Daten ein maschinelles Lernmodell konstruierten, würde es sehr schlecht funktionieren, da es fälschlicherweise annähme, dass die Spalten die gleichen Dinge beschreiben, da sie ja in der gleichen Position sind. Um dieses Problem zu vermeiden, rufen Sie entweder `get_dummies` mit einem DataFrame auf, das sowohl Trainings- als auch Testdaten enthält, oder Sie stellen sicher, dass die Spaltennamen nach dem Aufruf von `get_dummies` im Trainings- und Testdatensatz identisch sind.

## Zahlen können kategorische Daten kodieren

Die kategorischen Variablen im Datensatz `adult` waren als Strings kodiert. Auf der einen Seite werden dadurch Tippfehler möglich, auf der anderen Seite ist eine Variable dadurch klar als kategorisch gekennzeichnet. Häufig werden kategorische Variablen auch als Integerzahlen kodiert, teils der einfacheren Speicherung wegen, teils bedingt durch die Art der Datensammlung. Stellen Sie sich zum Beispiel vor, dass die Zensus-Daten im Datensatz `adult` über einen Fragebogen erhoben und die Antworten für `workclass` als 0 (erstes Kästchen angekreuzt), 1 (zweites Kästchen angekreuzt), 2 (drittes Kästchen angekreuzt) usw. abgelegt wurden. Dann würde die Spalte `Zahlen` von 0 bis 8 anstelle von Strings wie "Private" enthalten, und es wäre nicht unmittelbar ersichtlich, ob diese Tabellenspalte kontinuierlich oder kategorisch ist. Da wir wissen, dass die Zahlen die Art der Beschäftigung anzeigen, ist allerdings klar, dass es sich um klar abgegrenzte Zustände handelt, die nicht durch eine kontinuierliche Größe modelliert werden sollten.



Kategorische Merkmale werden häufig als Integerzahlen kodiert. Dass es sich dabei um Zahlen handelt, bedeutet nicht, dass man sie automatisch wie kontinuierliche Merkmale behandeln sollte. Es ist nicht immer klar, ob ein Integer-Merkmal als kontinuierlich oder diskret (und One-Hot-kodiert) anzusehen ist. Wenn es in der kodierten Bedeutung keine Reihenfolge gibt (wie im Beispiel der Spalte `workclass`), ist das Merkmal als diskret anzusehen. In anderen Fällen, wie Fünf-Sterne-Beurteilungen, hängt es von der konkreten Aufgabe, den Daten und dem verwendeten maschinellen Lernalgorithmus ab, welche Kodierung die bessere ist.

Die Funktion `get_dummies` in `pandas` sieht sämtliche Zahlen als kontinuierliche Größen an und generiert für diese keine Dummy-Variablen. Um dies zu umgehen, können Sie entweder den `OneHotEncoder` in `scikit-learn` verwenden, bei dem Sie angeben können, welche Variablen kontinuierlich und welche diskret sind, oder numerische Spalten im `DataFrame` in Strings umwandeln. Um Letzteres zu verdeutlichen, erstellen wir ein `DataFrame`-Object mit zwei Spalten, eine mit Strings und eine mit Integerzahlen:

**In[7]:**

```
# erstelle ein DataFrame mit einem Integer-Merkmal
# und einem kategorischen String-Merkmal
demo_df = pd.DataFrame({'Integer-Merkmal': [0, 1, 2, 1],
                        'Kategorisches Merkmal':
                        ['socks', 'fox', 'socks', 'box']})
display(demo_df)
```

Tabelle 4-4 zeigt das Ergebnis.

Tabelle 4-4: DataFrame mit kategorischen String-Merkmalen und Integer-Merkmalen

	Kategorisches Merkmal	Integer-Merkmal
0	socks	0
1	fox	1
2	socks	2
3	box	1

Mit `get_dummies` können wir nur das String-Merkmal kodieren, das Integer-Merkmal ändert sich nicht, wie in Tabelle 4-5 zu sehen ist:

In[8]:

```
pd.get_dummies(demo_df)
```

Tabelle 4-5: One-Hot-kodierte Version der Daten aus Tabelle 4-4, bei der das Integer-Merkmal unverändert ist

	Integer-Merkmal	Kategorisches Merkmal_box	Kategorisches Merkmal_fox	Kategorisches Merkmal_socks
0	0	0.0	0.0	1.0
1	1	0.0	1.0	0.0
2	2	0.0	0.0	1.0
3	1	1.0	0.0	0.0

Wenn Sie für die Spalte »Integer-Merkmal« Dummy-Variablen erzeugen möchten, können Sie die zu kodierenden Spalten explizit über den Parameter `columns` angeben. Dann werden beide Merkmale als kategorisch behandelt (siehe Tabelle 4-6):

In[9]:

```
demo_df['Integer-Merkmal'] = demo_df['Integer-Merkmal'].astype(str)
pd.get_dummies(demo_df, columns=['Integer-Merkmal', 'Kategorisches Merkmal'])
```

Tabelle 4-6: One-Hot-Kodierung der in Tabelle 4-4 kodierten Daten, sowohl für die Integer- als auch die String-Merkmale

	Integer-Merkmal_0	Integer-Merkmal_1	Integer-Merkmal_2	Kategorisches Merkmal_box	Kategorisches Merkmal_fox	Kategorisches Merkmal_socks
0	1.0	0.0	0.0	0.0	0.0	1.0
1	0.0	1.0	0.0	0.0	1.0	0.0
2	0.0	0.0	1.0	0.0	0.0	1.0
3	0.0	1.0	0.0	1.0	0.0	0.0

# Binning, Diskretisierung, lineare Modelle und Bäume

Die beste Möglichkeit, Daten zu repräsentieren, hängt nicht nur von der Semantik der Daten ab, sondern auch von der Art des verwendeten Modells. Lineare Modelle und auf Bäumen basierende Modelle (wie Entscheidungsbäume, Bäume mit Gradient Boosting und Random Forests), zwei große und häufig eingesetzte Familien, reagieren sehr verschieden auf unterschiedliche Repräsentation von Merkmalen. Kehren wir noch einmal zum in Kapitel 2 zur Regression verwendeten Datensatz `wave` zurück. Dieser enthält nur ein einziges Eingabemerkmals. Hier vergleichen wir ein lineares Regressionsmodell und einen Regressionsbaum auf diesem Datensatz (siehe Abbildung 4-1):

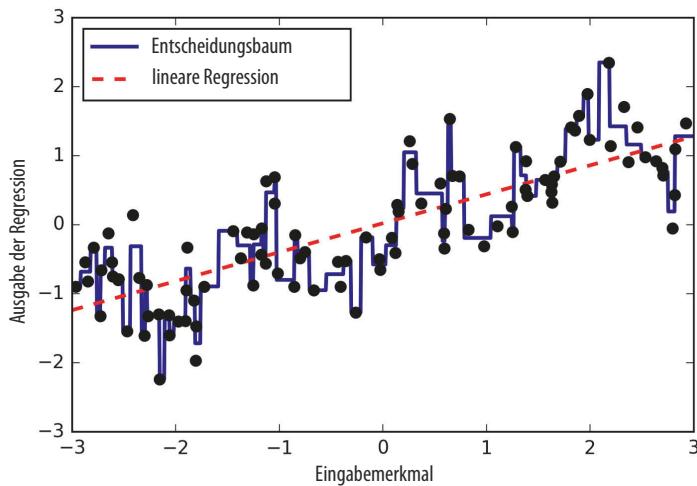


Abbildung 4-1: Vergleich von linearer Regression und einem Entscheidungsbau'm auf dem Datensatz 'wave'

In[10]:

```
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor

X, y = mglearn.datasets.make_wave(n_samples=100)
line = np.linspace(-3, 3, 1000, endpoint=False).reshape(-1, 1)

reg = DecisionTreeRegressor(min_samples_split=3).fit(X, y)
plt.plot(line, reg.predict(line), label="Entscheidungsbau'm")

reg = LinearRegression().fit(X, y)
plt.plot(line, reg.predict(line), label="lineare Regression")

plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("Ausgabe der Regression")
```

```
plt.xlabel("Eingabemerkmal")
plt.legend(loc="best")
```

Wie Sie wissen, können lineare Modelle nur lineare Beziehungen abbilden, im Falle eines einzelnen Merkmals also Linien. Der Entscheidungsbaum kann ein wesentlich komplexeres Modell der Daten erstellen. Allerdings hängt dieses stark von der Repräsentation der Daten ab. Lineare Modelle lassen sich bei kontinuierlichen Daten durch *Klassenbildung* verbessern (auch *Diskretisierung* genannt), bei der das Merkmal in mehrere Merkmale wie folgt aufgeteilt wird.

Stellen wir uns vor, wir unterteilen den Wertebereich eines Merkmals (in diesem Fall die Zahlen von  $-3$  bis  $3$ ) in eine festgelegte Anzahl von *Klassen* – sagen wir zehn. Ein Datenpunkt wird dann durch die Klasse repräsentiert, in die er passt. Um dies zu berechnen, müssen wir zuerst die Klassen festlegen. In diesem Fall, definieren wir zehn Klassen gleicher Breite zwischen  $-3$  und  $3$ . Wir verwenden dazu die Funktion `np.linspace`, um elf Werte als Klassengrenzen zu erstellen – die zehn Klassen sind die Intervalle zwischen zwei aufeinanderfolgenden Klassengrenzen:

**In[11]:**

```
bins = np.linspace(-3, 3, 11)
print("bins: {}".format(bins))
```

**Out[11]:**

```
bins: [-3. -2.4 -1.8 -1.2 -0.6 0. 0.6 1.2 1.8 2.4 3.]
```

Die erste Klasse im Beispiel enthält sämtliche Datenpunkte mit Werten von  $-3$  bis  $-2.4$ , die zweite Klasse enthält sämtliche Punkte mit Werten von  $-2.4$  bis  $-1.8$  usw.

Anschließend speichern wir die Zugehörigkeit der einzelnen Datenpunkte zu ihren Klassen. Dies lässt sich leicht mit der Funktion `np.digitize` berechnen:

**In[12]:**

```
which_bin = np.digitize(X, bins=bins)
print("\nDatenpunkte:\n", X[:5])
print("\nKlassenzugehörigkeit der Datenpunkte:\n", which_bin[:5])
```

**Out[12]:**

```
Datenpunkte:
[[ -0.753]
 [ 2.704]
 [ 1.392]
 [ 0.592]
 [-2.064]]
```

```
Klassenzugehörigkeit der Datenpunkte:
[[ 4]
 [10]
 [ 8]
 [ 6]
 [ 2]]
```

Wir haben also das einzelne kontinuierliche Eingabe-Merkmal aus dem Datensatz wave in ein kategorisches Merkmal mit der Klassenzugehörigkeit der einzelnen Punkte transformiert. Um ein Modell aus scikit-learn mit diesen Daten zu verwenden, transformieren wir dieses diskrete Merkmal mit dem OneHotEncoder aus dem Modul preprocessing. Der OneHotEncoder nimmt die gleiche Kodierung wie pandas.get\_dummies vor, allerdings funktioniert dies derzeit nur für durch Integerzahlen repräsentierte kategorische Variablen:

**In[13]:**

```
from sklearn.preprocessing import OneHotEncoder
# transformiere mit dem OneHotEncoder
encoder = OneHotEncoder(sparse=False)
# encoder.fit findet eindeutige Werte, die in which_bin auftreten
encoder.fit(which_bin)
# transform erstellt die One-Hot-Kodierung
X_binned = encoder.transform(which_bin)
print(X_binned[:5])
```

**Out[13]:**

```
[[ 0.  0.  0.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]
 [ 0.  0.  0.  0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]]
```

Weil wir zehn Klassen festgelegt haben, besteht der transformierte Datensatz X\_binned nun aus zehn Merkmalen:

**In[14]:**

```
print("X_binned.shape: {}".format(X_binned.shape))
```

**Out[14]:**

```
X_binned.shape: (100, 10)
```

Nun erstellen wir ein neues lineares Regressionsmodell und ein neues Modell mit einem Entscheidungsbaum aus den One-Hot-kodierten Daten. Das Ergebnis ist in Abbildung 4-2 dargestellt, wobei die Klassengrenzen als schwarz gepunktete Linien gezeichnet sind:

**In[15]:**

```
line_binned = encoder.transform(np.digitize(line, bins=bins))

reg = LinearRegression().fit(X_binned, y)
plt.plot(line, reg.predict(line_binned), label='lineare Regression mit Binning')

reg = DecisionTreeRegressor(min_samples_split=3).fit(X_binned, y)
plt.plot(line, reg.predict(line_binned), label='Entscheidungsbaum mit Binning')
plt.plot(X[:, 0], y, 'o', c='k')
plt.vlines(bins, -3, 3, linewidth=1, alpha=.2)
```

```

plt.legend(loc="best")
plt.ylabel("Ausgabe der Regression")
plt.xlabel("Eingabe-Merkmal")

```

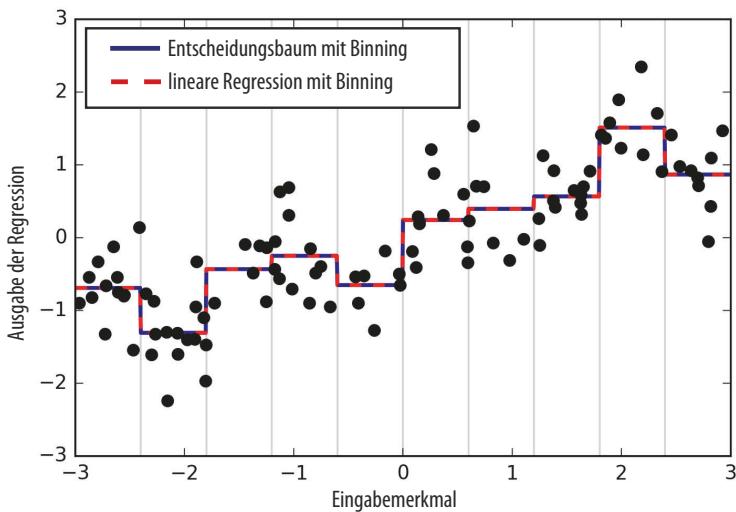


Abbildung 4-2: Vergleich von linearer Regression und Entscheidungsbaum auf in Klassen eingeteilten Merkmalen

Die gestrichelte und die durchgezogene Linie liegen genau aufeinander. Damit treffen das lineare Regressionsmodell und der Entscheidungsbaum die exakt gleichen Vorhersagen. Für jede Klasse sagen sie einen konstanten Wert vorher. Da die Merkmale innerhalb jeder Klasse konstant sind, muss ein Modell für alle Punkte einer Klasse den gleichen Wert vorhersagen. Vergleichen wir, was die Modelle vor und nach dem Binning erlernt haben, sehen wir, dass das lineare Modell deutlich flexibler geworden ist, weil es nun jeder Klasse einen anderen Wert zuordnen kann.

Gleichzeitig ist der Entscheidungsbaum deutlich weniger flexibel geworden. Merkmale in Klassen zu unterteilen, führt im Allgemeinen bei auf Bäumen basierenden Modellen zu keiner Verbesserung, weil diese Modelle selbst lernen, die Daten zu unterteilen. Entscheidungsbäume können also gewissermaßen das zur Vorhersage beste Binning ermitteln. Außerdem betrachten Entscheidungsbäume mehrere Merkmale auf einmal, wohingegen ein Binning normalerweise nur für einzelne Merkmale vorgenommen wird. Das lineare Modell hat aber durch die Transformation der Daten deutlich an Ausdrucksstärke hinzugewonnen.

Wenn es gute Gründe gibt, ein lineares Modell für einen bestimmten Datensatz einzusetzen – zum Beispiel weil dieser sehr groß und hochdimensioniert ist, einige Merkmale aber in nichtlinearer Beziehung zur Ausgabe stehen –, ist Binning eine ausgezeichnete Möglichkeit zur Verbesserung des Modells.

# Interaktionen und Polynome

Eine andere Möglichkeit, die Repräsentation eines Merkmals insbesondere bei der linearen Regression anzureichern, ist, aus den Originaldaten abgeleitete *interagierende Merkmale* und *polynomische Merkmale* hinzuzufügen. In der statistischen Modellierung wird diese Art der Merkmalsgenerierung häufig eingesetzt, aber sie findet sich auch oft in vielen praktischen Anwendungen maschinellen Lernens.

Als erstes Beispiel betrachten wir noch einmal Abbildung 4-2. Das lineare Modell hat für jede Klasse im Datensatz wave einen konstanten Wert erlernt. Wir wissen allerdings, dass lineare Modelle nicht nur den Achsenabschnitt, sondern auch die Steigung abbilden können. Eine Steigung können wir aus den in Klassen eingeteilten Daten ermitteln, indem wir das ursprüngliche Merkmal (die x-Achse im Diagramm) wieder hinzufügen. Damit erhalten wir den in Abbildung 4-3 gezeigten 11-dimensionalen Datensatz:

**In[16]:**

```
X_combined = np.hstack([X, X_binned])
print(X_combined.shape)
```

**Out[16]:**

```
(100, 11)
```

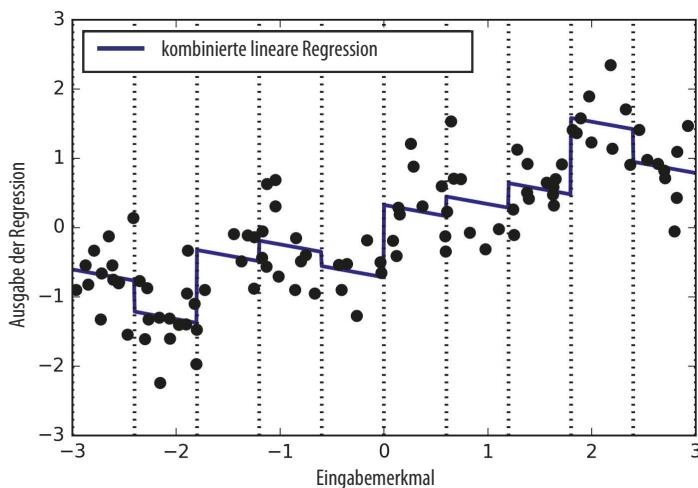


Abbildung 4-3: Lineare Regression mit in Klassen eingeteilten Merkmalen und einer einzelnen globalen Steigung

**In[17]:**

```
reg = LinearRegression().fit(X_combined, y)

line_combined = np.hstack([line, line_binned])
plt.plot(line, reg.predict(line_combined), label='kombinierte lineare Regression')

for bin in bins:
```

```

plt.plot([bin, bin], [-3, 3], ':', c='k')
plt.legend(loc="best")
plt.ylabel("Ausgabe der Regression")
plt.xlabel("Eingabe-Merkmal")
plt.plot(X[:, 0], y, 'o', c='k')

```

In diesem Beispiel hat das Modell für jede Klasse einen Achsenabschnitt und zusätzlich eine Steigung ermittelt. Die Steigung ist abfallend und gilt für alle Klassen. Es gibt nur ein einzelnes Merkmal für die x-Achse mit einer Steigung. Es erscheint nicht sehr sinnvoll, dass die Steigung für alle Klassen gleich ist. Wir würden eine separate Steigung für jede Klasse bevorzugen! Das erreichen wir, indem wir eine Interaktion oder ein zusammengesetztes Merkmal hinzufügen, das die Klasse eines Datenpunktes *und* dessen Position auf der x-Achse beinhaltet. Dieses Merkmal ist das Produkt aus dem Klassenbezeichner und dem ursprünglichen Merkmal. Wir erzeugen den entsprechenden Datensatz folgendermaßen:

**In[18]:**

```
X_product = np.hstack([X_binned, X * X_binned])
print(X_product.shape)
```

**Out[18]:**

```
(100, 20)
```

Der Datensatz hat nun 20 Merkmale: die Indikatoren für die Klassenzugehörigkeit der Datenpunkte und ein Produkt aus dem ursprünglichen Merkmal und der Klassenzugehörigkeit. Sie können sich das Produkt-Merkmal als eine zusätzliche Kopie des Merkmals auf der x-Achse pro Klasse vorstellen. Innerhalb der Klasse ist es das ursprüngliche Merkmal, und sonst ist es überall gleich null. Abbildung 4-4 zeigt das Ergebnis des linearen Modells auf dieser neuen Repräsentation:

**In[19]:**

```

reg = LinearRegression().fit(X_product, y)

line_product = np.hstack([line_binned, line * line_binned])
plt.plot(line, reg.predict(line_product), label='lineare Regression als Product')

for bin in bins:
    plt.plot([bin, bin], [-3, 3], ':', c='k')

plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("Ausgabe der Regression")
plt.xlabel("Eingabe-Merkmal")
plt.legend(loc="best")
```

Wie Sie sehen, hat in diesem Modell jede Klasse ihren eigenen Achsenabschnitt und ihre eigene Steigung.

Binning ist eine Möglichkeit, ein kontinuierliches Merkmal zu zerlegen. *Polynome* der ursprünglichen Merkmale sind eine weitere. Wir könnten für ein gegebenes Merkmal  $x$  auch die Terme  $x * \star 2$ ,  $x * \star 3$ ,  $x * \star 4$  usw. betrachten. Dies ist im Modul `preprocessing` als `PolynomialFeatures` implementiert:

In[20]:

```
from sklearn.preprocessing import PolynomialFeatures

# berücksichtige Polynome bis  $x^{** 10}$ :
# der Standardwert "include_bias=True" erzeugt ein Merkmal, das stets 1 ist
poly = PolynomialFeatures(degree=10, include_bias=False)
poly.fit(X)
X_poly = poly.transform(X)
```

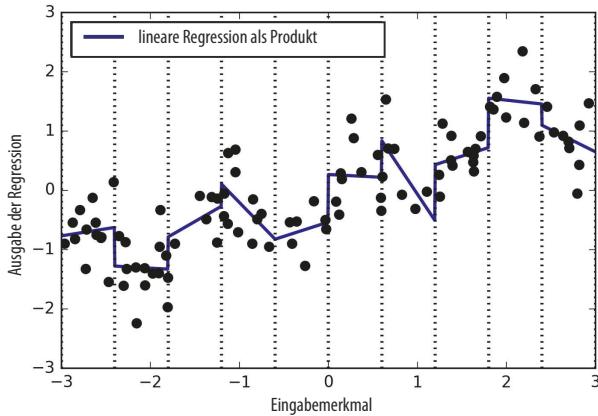


Abbildung 4-4: Lineare Regression mit einer separaten Steigung pro Klasse

Mit einem Polynom 10. Grades erhalten wir zehn Merkmale:

In[21]:

```
print("X_poly.shape: {}".format(X_poly.shape))
```

Out[21]:

```
X_poly.shape: (100, 10)
```

Vergleichen wir einmal die Einträge von X\_poly mit denen von X:

In[22]:

```
print("Einträge in X:\n{}".format(X[:5]))
print("Einträge in X_poly:\n{}".format(X_poly[:5]))
```

Out[22]:

```
Einträge in X:
[[-0.753]
 [ 2.704]
 [ 1.392]
 [ 0.592]
 [-2.064]]
Einträge in X_poly:
[[ -0.753      0.567     -0.427      0.321     -0.242      0.182
   -0.137      0.103     -0.078      0.058]
 [ 2.704      7.313    19.777    53.482   144.632    391.125
 1057.714   2860.360   7735.232  20918.278]]
```

```
[ 1.392    1.938    2.697    3.754    5.226    7.274
 [ 10.125   14.094   19.618   27.307]
 [ 0.592    0.350    0.207    0.123    0.073    0.043
 [ 0.025    0.015    0.009    0.005]
 [ -2.064   4.260   -8.791   18.144   -37.448   77.289
 [-159.516  329.222  -679.478  1402.367]]
```

Sie können über die Methode `get_feature_names` die Bedeutung der Merkmale als Exponenten aufschlüsseln:

**In[23]:**

```
print("Namen der polynomiellen Merkmale:\n{}".format(poly.get_feature_names()))
```

**Out[23]:**

```
Namen der polynomiellen Merkmale:
['x0', 'x0^2', 'x0^3', 'x0^4', 'x0^5', 'x0^6', 'x0^7', 'x0^8', 'x0^9', 'x0^10']
```

Sie sehen, dass die erste Spalte von `X_poly` genau `X` entspricht, während die übrigen Spalten Potenzen des ersten Eintrags sind. Es ist interessant zu sehen, wie groß einige der Werte werden. Die zweite Spalte enthält Werte bis über 20000, mehrere Größenordnungen von den übrigen entfernt.

Verwenden wir polynomielle Merkmale zusammen mit einem linearen Regressionsmodell, erhalten wir die klassische *polynomielle Regression* (siehe Abbildung 4-5):

**In[24]:**

```
reg = LinearRegression().fit(X_poly, y)

line_poly = poly.transform(line)
plt.plot(line, reg.predict(line_poly), label='polynomielle lineare Regression')
plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("Ausgabe der Regression")
plt.xlabel("Eingabemerkmal")
plt.legend(loc="best")
```

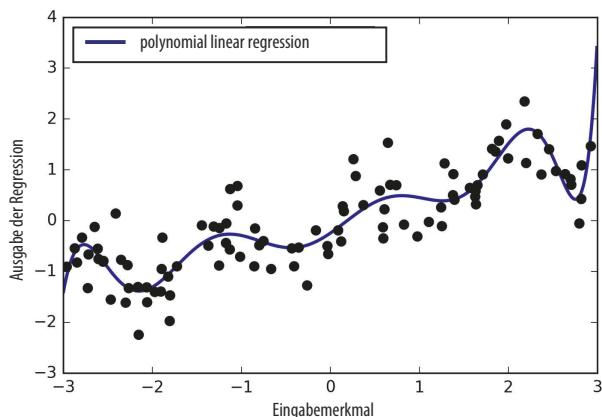


Abbildung 4-5: Lineare Regression mit polynomiellen Merkmalen 10. Grades

Wie Sie sehen, ermöglichen polynomiale Merkmale eine sehr geschmeidige Anpassung an die eindimensionalen Daten. Allerdings neigen Polynome höheren Grades dazu, sich an den Rändern oder in Bereichen mit wenig Daten extrem zu verhalten.

Zum Vergleich betrachten wir ein mit den Originaldaten ohne Transformation trainiertes Kernel-SVM-Modell (siehe Abbildung 4-6):

In[25]:

```
from sklearn.svm import SVR

for gamma in [1, 10]:
    svr = SVR(gamma=gamma).fit(X, y)
    plt.plot(line, svr.predict(line), label='SVR gamma={}'.format(gamma))

plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("Ausgabe der Regression")
plt.xlabel("Eingabemerkmal")
plt.legend(loc="best")
```

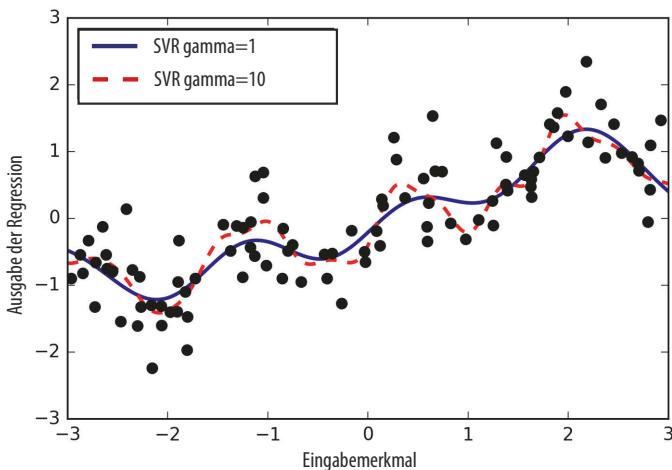


Abbildung 4-6: Vergleich unterschiedlicher Parameter für  $\gamma$  auf einem SVM mit RBF-Kernel

Mit einem komplexeren Modell wie einem Kernel SVM konnten wir eine ähnlich komplexe Vorhersage wie bei der polynomiellen Regression erzielen, ohne die Transformation der Merkmale explizit festzulegen.

Wir betrachten als realistischeres Beispiel für Interaktionen und Polynome noch einmal den Datensatz zu Immobilien in Boston. Wir haben polynomielle Merkmale auf diesem Datensatz bereits in Kapitel 2 eingesetzt. Diesmal kümmern wir uns darum, wie diese Merkmale aufgebaut wurden und wie sehr die polynomiellen Merkmale helfen. Zunächst laden wir die Daten und skalieren diese mit `MinMaxScaler`, sodass sie zwischen 0 und 1 liegen:

**In[26]:**

```
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

boston = load_boston()
X_train, X_test, y_train, y_test = train_test_split
    (boston.data, boston.target, random_state=0)

# Umskalieren der Daten
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Nun extrahieren wir polynomielle Merkmale und Interaktionen bis zum 2. Grad:

**In[27]:**

```
poly = PolynomialFeatures(degree=2).fit(X_train_scaled)
X_train_poly = poly.transform(X_train_scaled)
X_test_poly = poly.transform(X_test_scaled)
print("X_train.shape: {}".format(X_train.shape))
print("X_train_poly.shape: {}".format(X_train_poly.shape))
```

**Out[27]:**

```
X_train.shape: (379, 13)
X_train_poly.shape: (379, 105)
```

Ursprünglich bestanden die Daten aus 13 Merkmalen, die auf 105 Interaktions-Merkmale erweitert wurden. Diese neuen Merkmale enthalten alle möglichen Interaktionen zwischen zwei unterschiedlichen ursprünglichen Merkmalen sowie das Quadrat jedes einzelnen ursprünglichen Merkmals. Dabei bedeutet degree=2, dass wir alle Merkmale berücksichtigen, die aus bis zu zwei ursprünglichen Merkmalen zusammengesetzt sind. Die genauen Beziehungen zwischen den ein- und ausgegebenen Merkmalen lassen sich mit der Methode get\_feature\_names herausfinden:

**In[28]:**

```
print("Namen der polynomiellen Merkmale:\n{}".format(poly.get_feature_names()))
```

**Out[28]:**

```
Namen der polynomiellen Merkmale:
['1', 'x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'x9', 'x10',
 'x11', 'x12', 'x0^2', 'x0 x1', 'x0 x2', 'x0 x3', 'x0 x4', 'x0 x5', 'x0 x6',
 'x0 x7', 'x0 x8', 'x0 x9', 'x0 x10', 'x0 x11', 'x0 x12', 'x1^2', 'x1 x2',
 'x1 x3', 'x1 x4', 'x1 x5', 'x1 x6', 'x1 x7', 'x1 x8', 'x1 x9', 'x1 x10',
 'x1 x11', 'x1 x12', 'x2^2', 'x2 x3', 'x2 x4', 'x2 x5', 'x2 x6', 'x2 x7',
 'x2 x8', 'x2 x9', 'x2 x10', 'x2 x11', 'x2 x12', 'x3^2', 'x3 x4', 'x3 x5',
 'x3 x6', 'x3 x7', 'x3 x8', 'x3 x9', 'x3 x10', 'x3 x11', 'x3 x12', 'x4^2',
 'x4 x5', 'x4 x6', 'x4 x7', 'x4 x8', 'x4 x9', 'x4 x10', 'x4 x11', 'x4 x12',
 'x5^2', 'x5 x6', 'x5 x7', 'x5 x8', 'x5 x9', 'x5 x10', 'x5 x11', 'x5 x12',
 'x6^2', 'x6 x7', 'x6 x8', 'x6 x9', 'x6 x10', 'x6 x11', 'x6 x12', 'x7^2',
 'x7 x8', 'x7 x9', 'x7 x10', 'x7 x11', 'x7 x12', 'x8^2', 'x8 x9', 'x8 x10',
```

```
'x8 x11', 'x8 x12', 'x9^2', 'x9 x10', 'x9 x11', 'x9 x12', 'x10^2', 'x10 x11',
'x10 x12', 'x11^2', 'x11 x12', 'x12^2']
```

Das erste neue Merkmal ist ein konstantes Merkmal, hier als "1" bezeichnet. Die nächsten 13 sind die ursprünglichen Merkmale (mit "x0" bis "x12" bezeichnet). Es folgen das Quadrat des ersten Merkmals ("x0^2") und die Kombinationen des ersten mit allen übrigen Merkmalen.

Vergleichen wir die Aussagekraft der Daten, indem wir Ridge mit und ohne Interaktionen auf den Daten ausführen:

**In[29]:**

```
from sklearn.linear_model import Ridge
ridge = Ridge().fit(X_train_scaled, y_train)
print("Score ohne Interaktionen: {:.3f}".format(
    ridge.score(X_test_scaled, y_test)))
ridge = Ridge().fit(X_train_poly, y_train)
print("Score mit Interaktionen: {:.3f}".format(
    ridge.score(X_test_poly, y_test)))
```

**Out[29]:**

```
Score ohne Interaktionen: 0.621
Score mit Interaktionen: 0.753
```

Die Interaktionen und polynomiellen Merkmale haben ganz klar zu einem sprunghaften Anstieg der Vorhersagequalität beim Verwenden von Ridge geführt. Verwenden wir ein komplexeres Modell, wie etwa einen Random Forest, sieht es allerdings etwas anders aus:

**In[30]:**

```
from sklearn.ensemble import RandomForestRegressor
rf = RandomForestRegressor(n_estimators=100).fit(X_train_scaled, y_train)
print("Score ohne Interaktionen: {:.3f}".format(
    rf.score(X_test_scaled, y_test)))
rf = RandomForestRegressor(n_estimators=100).fit(X_train_poly, y_train)
print("Score mit Interaktionen: {:.3f}".format(rf.score(X_test_poly, y_test)))
```

**Out[30]:**

```
Score ohne Interaktionen: 0.799
Score mit Interaktionen: 0.763
```

Sie sehen, dass der Random Forest sogar ohne zusätzliche Merkmale im Hinblick auf die Vorhersagequalität dem Ridge-Verfahren überlegen ist. Das Hinzufügen von Interaktionen und Polynomen senkt die Vorhersagegüte sogar ein wenig.

## Univariate nichtlineare Transformation

Wir haben bereits gesehen, dass das Hinzufügen von quadrierten Merkmalen oder höheren Potenzen bei der Regression hilft. Es gibt noch weitere nützliche Transformationen: Insbesondere hilft die Anwendung von mathematischen Funktionen wie

`log`, `exp` oder `sin` bei bestimmten Merkmalen. Während auf Bäumen basierende Modelle sich nur für die Reihenfolge der Merkmale interessieren, sind lineare Modelle und neuronale Netze stark an die Skalierung und Verteilung jedes Merkmals gebunden. Wenn es eine nichtlineare Beziehung zwischen dem Merkmal und der Zielgröße gibt, lässt sich dies nur schwer modellieren – insbesondere bei einer Regression. Die Funktionen `log` und `exp` helfen, die relative Skalierung der Daten anzupassen, sodass sie durch ein lineares Modell oder ein neuronales Netz besser erfasst werden können. Wir haben hierfür mit den Speicherpreisen in Kapitel 2 bereits ein Anwendungsbeispiel gesehen. Die Funktionen `sin` und `cos` erweisen sich bei Daten mit periodischen Mustern als nützlich.

Die meisten Modelle funktionieren am besten, wenn jedes Merkmal (und im Falle einer Regression auch die Zielgröße) mehr oder weniger normalverteilt ist – das Histogramm jedes einzelnen Merkmals sollte also ähnlich zur bekannten »Gaußschen Glockenkurve« sein. Transformationen wie `log` und `exp` wirken etwas improvisiert, sind aber einfache und effiziente Möglichkeiten, dies zu erreichen. Eine solche Transformation ist vor allem beim Arbeiten mit Anzahlen in Form von Integerzahlen hilfreich. Mit Anzahlen meinen wir Merkmale wie »wie oft hat sich Nutzer A eingeloggt?«. Anzahlen sind niemals negativ und folgen häufig bestimmten statistischen Gesetzmäßigkeiten. Wir verwenden hier einen synthetischen Datensatz von Anzahlen mit Eigenschaften, die Sie auch in realen Anwendungen finden können. Alle Merkmale sind ganzzahlige Werte, die Ausgabegröße hingegen ist kontinuierlich:

**In[31]:**

```
rnd = np.random.RandomState(0)
X_org = rnd.normal(size=(1000, 3))
w = rnd.normal(size=3)

X = rnd.poisson(10 * np.exp(X_org))
y = np.dot(X_org, w)
```

Betrachten wir die ersten zehn Einträge des ersten Merkmals. Alle sind positive ganze Zahlen, aber abgesehen davon ist kein bestimmtes Muster erkennbar.

Wenn wir die Häufigkeit jedes einzelnen Wertes auszählen, wird die Verteilung der Werte deutlicher:

**In[32]:**

```
print("Häufigkeit von Merkmalen:\n{}".format(np.bincount(X[:, 0])))
```

**Out[32]:**

Häufigkeit von Merkmalen																								
[28	38	68	48	61	59	45	56	37	40	35	34	36	26	23	26	27	21	23	23	18	21	10	9	17
9	7	14	12	7	3	8	4	5	5	3	4	2	4	1	1	3	2	5	3	8	2	5	2	1
2	3	3	2	2	3	3	0	1	2	1	0	0	3	1	0	0	0	1	3	0	1	0	2	0
1	1	0	0	0	0	1	0	0	2	2	0	1	1	0	0	0	0	1	1	0	0	0	0	0
0	0	1	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Der Wert 2 scheint mit 68 Instanzen am häufigsten aufzutreten (bincount beginnt immer bei 0), und die Anzahl fällt für die höheren Werte schnell ab. Allerdings gibt es einige sehr hohe Werte wie 134, der zweimal vorkommt. Wir stellen die Häufigkeiten in Abbildung 4-7 dar:

In[33]:

```
bins = np.bincount(X[:, 0])
plt.bar(range(len(bins)), bins, color='w')
plt.ylabel("Häufigkeit")
plt.xlabel("Wert")
```

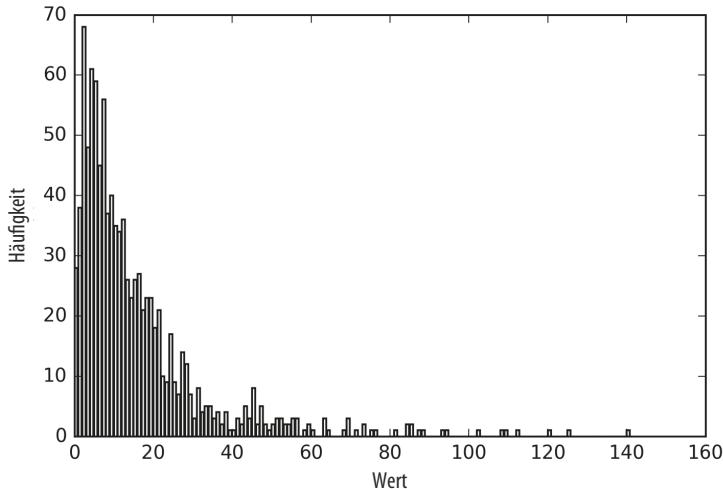


Abbildung 4-7: Histogramm der Werte von  $X[0]$

Die Merkmale  $X[:, 1]$  und  $X[:, 2]$  haben ähnliche Eigenschaften. Diese Art Verteilung von Werten (viele kleine und wenige sehr große) tritt in der Praxis sehr häufig auf.<sup>1</sup> Allerdings können lineare Modelle mit dieser Verteilung nicht besonders gut umgehen. Versuchen wir einmal, diese Daten mit einer Ridge-Regression zu modellieren:

In[34]:

```
from sklearn.linear_model import Ridge
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
score = Ridge().fit(X_train, y_train).score(X_test, y_test)
print("Test Score: {:.3f}".format(score))
```

Out[34]:

```
Test Score: 0.622
```

Wie Sie am relativ niedrigen  $R^2$ -Score sehen, konnte Ridge die Beziehung zwischen  $X$  und  $y$  nicht wirklich abbilden. Eine logarithmische Transformation könnte aller-

<sup>1</sup> Dies ist die beim Zählen von Daten elementare Poisson-Verteilung.

dings helfen. Weil der Wert 0 in den Daten vorkommt (und der Logarithmus von 0 nicht definiert ist), können wir nicht einfach  $\log$  berechnen, sondern müssen stattdessen  $\log(X + 1)$  verwenden:

**In[35]:**

```
X_train_log = np.log(X_train + 1)
X_test_log = np.log(X_test + 1)
```

Nach der Transformation ist die Verteilung der Daten weniger asymmetrisch und enthält keine großen Ausreißer mehr (siehe Abbildung 4-8):

**In[36]:**

```
plt.hist(X_train_log[:, 0], bins=25, color='gray')
plt.ylabel("Häufigkeit")
plt.xlabel("Wert")
```

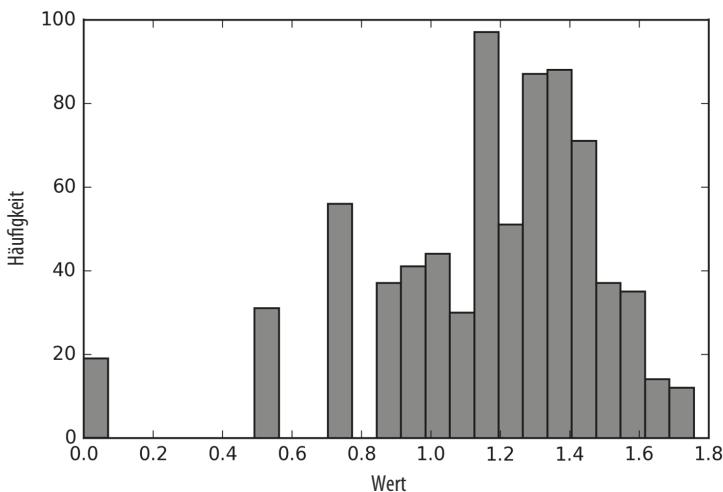


Abbildung 4-8: Histogramm der Werte von  $X[0]$  nach logarithmischer Transformation

Ein auf den neuen Daten konstruiertes Ridge-Modell passt deutlich besser:

**In[37]:**

```
score = Ridge().fit(X_train_log, y_train).score(X_test_log, y_test)
print("Test Score: {:.3f}".format(score))
```

**Out[37]:**

```
Test Score: 0.875
```

Für jede Kombination von Datensatz und Modell die richtige Transformation zu finden, ist so etwas wie ein Kunsthhandwerk. In diesem Beispiel hatten sämtliche Merkmale die gleichen Eigenschaften. In der Praxis ist das eher selten. Normalerweise sollte nur ein Teil der Merkmale transformiert werden, manchmal muss auch jedes Merkmal unterschiedlich transformiert werden. Wie weiter oben erwähnt, ist diese Art der Transformation für auf Bäumen basierende Modelle irrelevant, aber

für lineare Modelle ist sie essenziell. Bisweilen ist es bei einer Regression sinnvoll, die Zielgröße  $y$  zu transformieren. Die Vorhersage von Anzahlen (z. B. die Anzahl Bestellungen) ist eine wiederkehrende Aufgabe, und die Transformation  $\log(y + 1)$  hilft dabei häufig.<sup>2</sup>

Wie Sie in den vorigen Beispielen gesehen haben, können Klassenbildung, Polynome und Interaktionen einen riesigen Einfluss auf die Vorhersagegüte von Modellen auf einem bestimmten Datensatz haben. Dies trifft vor allem auf weniger komplexe Modelle wie lineare Modelle und naive Bayes-Modelle zu. Auf Bäumen basierende Modelle sind dagegen meist in der Lage, wichtige Zusammenhänge selbst zu entdecken, und erfordern daher in den meisten Fällen keine explizite Transformation der Daten. Andere Modelle wie SVMs, nächste-Nachbarn und neuronale Netze profitieren manchmal von Klassenbildung, Interaktionen oder Polynomen, aber die Implikationen sind in der Regel weniger klar als im Falle von linearen Modellen.

## Automatische Auswahl von Merkmalen

Mit so vielen Möglichkeiten, neue Merkmale zu erzeugen, ist die Versuchung groß, die Dimensionen der Daten weit über die Anzahl ursprünglicher Merkmale zu erhöhen. Allerdings werden alle Modelle durch das Hinzufügen von Merkmalen komplexer, was das Risiko von Overfitting erhöht. Wenn Sie neue Merkmale hinzufügen oder allgemein mit hochdimensionalen Datensätzen arbeiten, kann es sinnvoll sein, die Anzahl Merkmale auf die allernützlichsten zu reduzieren und den Rest zu verwerfen. Dadurch erhalten Sie einfachere Modelle, die besser verallgemeinern. Aber woher sollen wir wissen, wie gut die einzelnen Merkmale sind? Es gibt drei Grundstrategien: *univariate Statistiken*, *modellbasierte Auswahl* und *iterative Auswahl*. Wir werden alle drei im Detail besprechen. Alle drei Verfahren sind überwachte Methoden, was heißt, dass sie zum Anpassen des Modells eine Zielgröße benötigen. Daher müssen wir die Daten in Trainings- und Testdaten aufteilen und lediglich die Trainingsdaten zur Auswahl von Merkmalen verwenden.

### Univariate Statistiken

Bei univariaten Statistiken berechnen wir, ob es einen statistisch signifikanten Zusammenhang zwischen je einem der Merkmale und der Zielgröße gibt. Dann werden die mit der stärksten Konfidenz korrelierenden Merkmale ausgewählt. Bei einer Klassifikationsaufgabe nennt man dieses Vorgehen *Analyse der Varianz (ANOVA)*. Eine entscheidende Eigenschaft dieser Art von Test ist, dass sie *univariat* sind, also jedes Merkmal einzeln betrachtet wird. Deshalb wird ein Merkmal verworfen, wenn es nur in Verbindung mit einem zweiten Merkmal aussagekräftig

<sup>2</sup> Dies ist eine sehr ungenaue Näherung der Poisson-Regression, die vom probabilistischen Standpunkt die angemessene Lösung wäre.

ist. Univariate Tests lassen sich sehr schnell berechnen und erfordern keine Modellbildung. Sie sind völlig vom nach der Merkmalsauswahl zu entwickelnden Modell unabhängig.

Um eine Auswahl univariater Merkmale in scikit-learn durchzuführen, müssen Sie ein Testverfahren auswählen, für gewöhnlich entweder `f_classif` (die Standardeinstellung) für Klassifikationsprobleme oder `f_regression` für Regressionen, sowie eine Methode zum Verwerfen von Merkmalen aufgrund der im Test berechneten  $p$ -Werte. Alle Methoden zum Verwerfen von Parametern verwenden einen Schwellenwert, um Merkmale mit zu hohem  $p$ -Wert zu verwerfen (was bedeutet, dass ein Zusammenhang mit der Zielgröße unwahrscheinlich ist). Die Methoden unterscheiden sich darin, wie der Schwellenwert berechnet wird. Die einfachsten darunter sind `SelectKBest`, die eine als  $k$  festgelegte Anzahl Merkmale auswählt, und `SelectPercentile`, die einen festen prozentualen Anteil der Merkmale auswählt. Wenden wir die Auswahl von Merkmalen auf die Klassifikation des Datensatzes `cancer` an. Um uns die Aufgabe ein wenig zu erschweren, fügen wir den Daten einige nicht informative Merkmale mit Rauschen hinzu. Wir erwarten, dass die Auswahl von Merkmalen diese nicht informativen Merkmale erkennt und entfernt:

**In[38]:**

```
from sklearn.datasets import load_breast_cancer
from sklearn.feature_selection import SelectPercentile
from sklearn.model_selection import train_test_split

cancer = load_breast_cancer()

# erzeuge deterministische Zufallszahlen
rng = np.random.RandomState(42)
noise = rng.normal(size=(len(cancer.data), 50))
# füge den Daten verrauschte Merkmale hinzu
# die ersten 30 Merkmale sind aus dem Datensatz, die nächsten 50 Rauschen
X_w_noise = np.hstack([cancer.data, noise])

X_train, X_test, y_train, y_test = train_test_split(
    X_w_noise, cancer.target, random_state=0, test_size=.5)
# verwende f_classif (den Standard) und SelectPercentile
# um 50 % der Merkmale auszuwählen
select = SelectPercentile(percentile=50)
select.fit(X_train, y_train)
# transformiere die Trainingsdaten
X_train_selected = select.transform(X_train)

print("X_train.shape: {}".format(X_train.shape))
print("X_train_selected.shape: {}".format(X_train_selected.shape))
```

**Out[38]:**

```
X_train.shape: (284, 80)
X_train_selected.shape: (284, 40)
```

Wie Sie sehen, hat sich die Anzahl Merkmale von 80 auf 40 verringert (50 % der ursprünglichen Merkmale). Mit der Methode `get_support` können wir herausfinden, welche Merkmale ausgewählt wurden. Die Methode liefert uns die ausgewählten Merkmale als Maske mit booleschen Werten (dargestellt in Abbildung 4-9):

**In[39]:**

```
mask = select.get_support()  
print(mask)  
# stelle die Maske dar -- schwarz ist True, weiß ist False  
plt.matshow(mask.reshape(1, -1), cmap='gray_r')  
plt.xlabel("Index")
```

**Out[39]:**

```
[ True  True  True  True  True  True  True  True  True  False  True  False  
 True  True  True  True  True  True  False  False  True  True  True  True  
 True  True  True  True  True  True  False  False  False  True  False  True  
 False  False  True  False  False  False  False  True  False  False  True  False  
 False  True  False  True  False  False  False  False  False  False  True  False  
 True  False  False  False  False  True  False  True  False  False  False  False  
 True  True  False  True  False  False  False  False  False  False  False  False]
```



Abbildung 4-9: Von `SelectPercentile` ausgewählte Merkmale

Wie Sie anhand der Darstellung der Maske sehen können, gehören die meisten ausgewählten Merkmale zu den ursprünglichen Merkmalen, und die meisten verrauschten Merkmale wurden entfernt. Allerdings ist die Erkennung der ursprünglichen Merkmale nicht perfekt. Vergleichen wir die Qualität einer logistischen Regression auf allen Merkmalen mit der Qualität auf dem Satz ausgewählter Merkmale:

**In[40]:**

```
from sklearn.linear_model import LogisticRegression  
  
# transformiere die Testdaten  
X_test_selected = select.transform(X_test)  
  
lr = LogisticRegression()  
lr.fit(X_train, y_train)  
print("Score mit allen Merkmälern: {:.3f}".format(lr.score(X_test, y_test)))  
lr.fit(X_train_selected, y_train)  
print("Score mit ausgewählten Merkmälern: {:.3f}".format(  
    lr.score(X_test_selected, y_test)))
```

**Out[40]:**

```
Score mit allen Merkmälern: 0.930  
Score mit ausgewählten Merkmälern: 0.940
```

In diesem Fall hat das Entfernen des Rauschens die Vorhersagegüte verbessert, auch wenn einige der ursprünglichen Merkmale verloren gegangen sind. Dies war

ein sehr einfaches künstliches Beispiel, bei realen Daten sind die Ergebnisse in der Regel unterschiedlich. Univariate Auswahl von Merkmalen kann trotzdem sehr hilfreich sein, wenn es so viele Merkmale gibt, dass die Konstruktion eines Modells mit diesen nicht möglich ist, oder wenn Sie vermuten, dass viele Merkmale keinerlei Information enthalten.

## Modellbasierte Auswahl von Merkmalen

Die modellbasierte Auswahl von Merkmalen verwendet ein überwachtes maschinelles Lernverfahren, um die Aussagekraft jedes Merkmals zu beurteilen, und lässt nur die wichtigsten übrig. Das zur Auswahl von Merkmalen verwendete überwachte Modell muss dabei nicht das gleiche Modell sein wie das bei der endgültigen überwachten Modellierung verwendete. Das Modell zur Merkmalsauswahl muss ein Maß für die Wichtigkeit jedes Merkmals enthalten, sodass entsprechend diesem Maß eine Rangliste erstellt werden kann. Entscheidungsbäume und auf Entscheidungsbäumen basierende Modelle enthalten das Attribut `feature_importances_`, das die Wichtigkeit jedes Merkmals direkt kodiert. Lineare Modelle enthalten Koeffizienten, die sich ebenfalls zum Erfassen der Wichtigkeit von Merkmalen einsetzen lassen, indem man deren absolute Werte betrachtet. Wie wir in Kapitel 2 gesehen haben, erlernen lineare Modelle mit einem L1-Strafterm dünn besetzte Koeffizienten, die nur einen kleinen Teil der Merkmale verwenden. Dies lässt sich als eine Art Merkmalsauswahl für das Modell selbst ansehen, wir können das Verfahren aber auch zur Merkmalsauswahl in der Vorverarbeitung für ein anderes Modell verwenden. Im Gegensatz zu univariater Auswahl berücksichtigt die modellbasierte Auswahl alle Merkmale auf einmal und kann so Wechselwirkungen erfassen (sofern das Modell diese abbilden kann). Um eine modellbasierte Auswahl von Merkmalen durchzuführen, müssen wir den Transformer `SelectFromModel` verwenden:

In[41]:

```
from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import RandomForestClassifier
select = SelectFromModel(
    RandomForestClassifier(n_estimators=100, random_state=42),
    threshold="median")
```

Die Klasse `SelectFromModel` wählt alle Merkmale mit einem Wichtigkeitsmaß über dem angegebenen Schwellenwert aus. Dazu wird das vom überwachten Modell bereitgestellte Wichtigkeitsmaß verwendet. Um ein zur univariaten Merkmalsauswahl vergleichbares Ergebnis zu erhalten, haben wir den Median als Schwellenwert festgelegt, sodass die Hälfte der Merkmale ausgewählt wird. Wir verwenden zur Klassifikation einen Random Forest mit 100 Bäumen und berechnen die Wichtigkeiten der Merkmale. Dies ist ein recht komplexes Modell, das sehr viel mächtiger als die univariaten Tests ist. Damit ist es an der Zeit, das Modell anzupassen:

In[42]:

```
select.fit(X_train, y_train)
X_train_l1 = select.transform(X_train)
print("X_train.shape: {}".format(X_train.shape))
print("X_train_l1.shape: {}".format(X_train_l1.shape))
```

Out[42]:

```
X_train.shape: (284, 80)
X_train_l1.shape: (284, 40)
```

Wir schauen uns abermals die ausgewählten Merkmale an (Abbildung 4-10):

In[43]:

```
mask = select.get_support()
# visualisiere die Maske -- schwarz ist True, weiß ist False
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
plt.xlabel("Index")
```



Abbildung 4-10: Von `SelectFromModel` mit einem `RandomForestClassifier` ausgewählte Merkmale

Diesmal wurden alle der ursprünglichen Merkmale bis auf zwei ausgewählt. Weil wir die Auswahl von 40 Merkmalen angegeben haben, wurden auch einige der vertrauschten Merkmale ausgewählt. Betrachten wir die Vorhersagequalität:

In[44]:

```
X_test_l1 = select.transform(X_test)
score = LogisticRegression().fit(X_train_l1, y_train).score(X_test_l1, y_test)
print("Test Score: {:.3f}".format(score))
```

Out[44]:

```
Test Score: 0.951
```

Mit der verbesserten Auswahl von Merkmalen hat sich auch die Vorhersagequalität etwas verbessert.

## Iterative Auswahl von Merkmalen

Beim univariaten Test haben wir kein Modell verwendet und bei der modellbasierten Auswahl Merkmale mit einem einzelnen Modell ausgewählt. Bei der iterativen Merkmalsauswahl wird eine Folge von Modellen mit unterschiedlich ausgewählten Merkmalen konstruiert. Es gibt zwei grundlegende Vorgehensweisen: ohne Merkmale anzufangen und einzelne Merkmale hinzuzufügen, bis eine Abbruchbedingung erfüllt ist, oder mit allen Merkmalen anzufangen und sie bis zum Erreichen der Abbruchbedingung einzeln zu entfernen. Weil dabei viele Modelle erstellt werden, sind diese Methoden sehr viel rechenintensiver als die zuvor vorgestellten Methoden. Eine solche Methode ist die *rekursive Eliminierung von Merkmalen*, die mit

allen Merkmalen beginnt, ein Modell erstellt und das dem Modell nach unwichtigste Merkmal entfernt. Anschließend wird ein neues Modell mit allen außer dem entfernten Merkmal berechnet und der Vorgang wiederholt, bis nur noch eine vorher festgelegte Anzahl Merkmale übrig ist. Damit dieser Vorgang funktioniert, muss das zur Auswahl verwendete Modell wie bei der modellbasierten Merkmalsauswahl irgendeine Art Maß für die Wichtigkeit von Merkmalen bereitstellen. Hier verwenden wir den schon weiter oben eingesetzten Random Forest und erhalten die in Abbildung 4-11 gezeigten Ergebnisse:

**In[45]:**

```
from sklearn.feature_selection import RFE
select = RFE(RandomForestClassifier(n_estimators=100, random_state=42),
              n_features_to_select=40)

select.fit(X_train, y_train)
# visualisiere die ausgewählten Merkmale:
mask = select.get_support()
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
plt.xlabel("Index")
```



Abbildung 4-11: Durch rekursive Eliminierung mit einem Random Forest-Klassifikator ausgewählte Merkmale

Die Auswahl von Merkmalen hat sich im Vergleich zur univariaten und modellbasierten Auswahl verbessert, aber ein Merkmal wird nach wie vor außer Acht gelassen. Auch dauert es deutlich länger als bei der modellbasierten Auswahl, diesen Code auszuführen, weil das Random Forest-Modell insgesamt 40 Mal trainiert wird, einmal für jedes eliminierte Merkmal. Überprüfen wir die Genauigkeit des logistischen Regressionsmodells beim Einsatz von RFE für die Auswahl von Merkmalen:

**In[46]:**

```
X_train_rfe= select.transform(X_train)
X_test_rfe= select.transform(X_test)

score = LogisticRegression().fit(X_train_rfe, y_train).score(X_test_rfe, y_test)
print("Test Score: {:.3f}".format(score))
```

**Out[46]:**

```
Test Score: 0.951
```

Wir können das Modell innerhalb der RFE also verwenden, um Vorhersagen zu treffen. Dieses verwendet nur die ausgewählten Merkmale:

**In[47]:**

```
print("Test Score: {:.3f}".format(select.score(X_test, y_test)))
```

## Out[47]:

Test Score: 0.951

Die Vorhersagegüte des Random Forest innerhalb des RFE ist die gleiche wie die durch das Trainieren eines logistischen Regressionsmodells auf den ausgewählten Merkmalen. Anders gesagt, haben wir erst einmal die richtigen Merkmale ausgewählt, schneidet das lineare Modell genauso gut ab wie ein Random Forest.

Falls Sie sich nicht sicher sind, was Sie als Eingabe für Ihre maschinellen Lernalgorithmen verwenden sollen, ist die automatische Auswahl von Merkmalen ausgesprochen hilfreich. Sie ist auch ausgezeichnet zum Reduzieren der Anzahl Merkmale – zum Beispiel, um die Vorhersage zu beschleunigen oder leichter interpretierbare Modelle zu ermöglichen. In der Praxis führt die Auswahl von Merkmalen meist nicht zu großen Verbesserungen der Vorhersagequalität. Dennoch ist es ein wertvolles Instrument im Werkzeugkasten desjenigen, der Merkmale generiert.

## Berücksichtigen von Expertenwissen

Das Generieren von Merkmalen ist oft ein wichtiger Zeitpunkt, um *Expertенwissen* für eine bestimmte Anwendung einzubinden. Auch wenn der Zweck von maschinellem Lernen oft ist, das Erstellen eines Regelwerks durch Experten zu vermeiden, bedeutet das nicht, dass man bereits vorhandenes Wissen der Anwendung oder des Fachgebiets ignorieren sollte. Experten können oft dabei helfen, nützliche Merkmale zu identifizieren, die viel informativer als die ursprüngliche Repräsentation der Daten sind. Stellen Sie sich vor, Sie arbeiten für ein Reisebüro und möchten die Preise von Flügen vorhersagen. Nehmen wir an, Sie hätten bereits Aufzeichnungen zu Preisen, Terminen, Fluglinien, Abflug- und Zielflughäfen. Ein maschinelles Lernmodell könnte daraus eine anständige Vorhersage konstruieren. Allerdings lassen sich einige wichtige Faktoren von Flugpreisen nicht erlernen. Beispielsweise sind Flüge normalerweise in den Ferienmonaten und um Feiertage herum teurer. Einige Feiertage (z. B. Weihnachten) stehen fest, und deren Effekte lassen sich anhand des Datums erlernen, andere hängen von den Mondphasen ab (z. B. Hanukkah und Ostern) oder werden von Behörden festgelegt (z. B. Schulferien). Letztere Ereignisse lassen sich nicht aus den Daten ermitteln, falls zu jedem Flug lediglich das (gregorianische) Datum gespeichert wird. Es ist dagegen einfach, ein Merkmal hinzuzufügen, in dem kodiert ist, ob ein Flug während, vor oder nach einem öffentlichen Feiertag oder Ferientag stattfand. Auf diese Weise wird vorhandenes Wissen über die Hintergründe der Aufgabe in den Merkmalen eingebunden, um einen maschinellen Lernalgorithmus zu unterstützen. Das Hinzufügen eines Merkmals zwingt einen maschinellen Lernalgorithmus nicht dazu, dieses auch zu verwenden. Selbst wenn sich herausstellen sollte, dass die Angabe der Feiertage nichts über den Flugpreis aussagt, schadet es nicht, die Daten mit dieser Information anzureichern.

Wir werden uns nun den Einsatz von Expertenwissen an einem Fallbeispiel genauer ansehen – auch wenn dies mehr mit »gesundem Menschenverstand« als mit Exper-

tenwissen zu tun hat. Die Aufgabe ist, die vor Andreas' Haus gemieteten Fahrräder vorherzusagen.

In New York betreibt Citi Bike ein Netzwerk von Stationen zur Vermietung von Fahrrädern mit einem Abonnementsystem. Die Stationen befinden sich überall in der Stadt verteilt und stellen eine angenehme Möglichkeit der Fortbewegung dar. Die Daten zur Fahrradvermietung werden in anonymisierter Form (<https://www.citibikenyc.com/system-data>) zur Verfügung gestellt und wurden auf unterschiedliche Weise analysiert. Bei dieser Aufgabe möchten wir vorhersagen, wie viele Leute zu einer gegebenen Zeit ein Fahrrad vor Andreas' Haus mieten werden, sodass er weiß, wie viele Fahrräder für ihn übrig sind. Wir laden zunächst die Daten dieser Station für August 2015 in ein pandas DataFrame. Wir unterteilen die Daten in dreistündige Intervalle, um die wichtigsten Trends für jeden Tag zu erhalten:

**In[48]:**

```
citibike = mglearn.datasets.load_citibike()
```

**In[49]:**

```
print("Citi Bike Daten:\n{}".format(citibike.head()))
```

**Out[49]:**

```
Citi Bike Daten:  
starttime  
2015-08-01 00:00:00    3.0  
2015-08-01 03:00:00    0.0  
2015-08-01 06:00:00    9.0  
2015-08-01 09:00:00   41.0  
2015-08-01 12:00:00   39.0  
Freq: 3H, Name: one, dtype: float64
```

Das folgende Beispiel stellt die Mietfrequenzen für den gesamten Monat dar (Abbildung 4-12):

**In[50]:**

```
plt.figure(figsize=(10, 3))  
xticks = pd.date_range(start=citibike.index.min(), end=citibike.index.max(),  
                      freq='D')  
plt.xticks(xticks, xticks.strftime("%a %m-%d"), rotation=90, ha="left")  
plt.plot(citibike, linewidth=1)  
plt.xlabel("Datum")  
plt.ylabel("Vermietungen")
```

Beim Betrachten der Daten können wir für jedes 24-Stunden-Intervall Tag und Nacht deutlich voneinander unterscheiden. Auch die Muster für Wochentage und Wochenenden sind recht unterschiedlich. Beim Auswerten einer Zeitreihe wie dieser für eine Vorhersage möchten wir für gewöhnlich *aus der Vergangenheit lernen* und *die Zukunft vorhersagen*. Das bedeutet für die Aufteilung in Trainings- und Testdaten, dass wir alle Daten bis zu einem bestimmten Termin als Trainingsdaten und alle Daten nach diesem Termin als Testdaten verwenden. Dies entspricht dem typischen Einsatzgebiet von Vorhersagen auf Zeitreihen: Was können wir über morgen mit dem Wissen über heutige Vermietungen aussagen? Hier verwenden wir

die ersten 184 Datenpunkte zu den ersten 23 Tagen als Trainingsdatensatz und die übrigen 64 Datenpunkte zu den verbleibenden 8 Tagen als Testdatensatz.

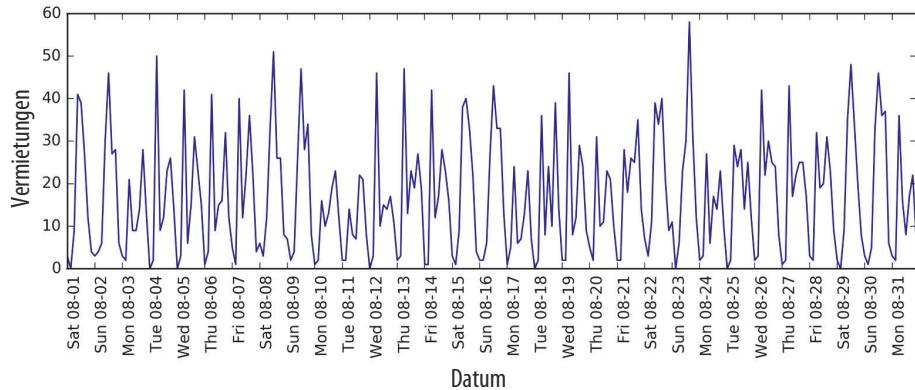


Abbildung 4-12: Anzahl von Fahrradvermietungen über die Zeit an einer bestimmten Citi Bike-Station

Die einzigen für unsere Vorhersage verwendeten Merkmale sind das Datum und die Uhrzeit, zu der eine bestimmte Anzahl Vermietungen stattfand. Also ist die Eingabe das Datum und die Uhrzeit – z. B. 2015-08-01 00:00:00 – und die Ausgabe ist die Anzahl Vermietungen in den folgenden drei Stunden (laut unserem DataFrame in diesem Falle drei). Die POSIX-Zeit, die seit Januar 1970 00:00:00 verflossene Anzahl Sekunden (der Beginn der UNIX-Zeitrechnung), ist eine (überraschend) verbreitete Art, Daten auf Computern zu speichern. Für einen ersten Versuch verwenden wir diese Integerzahl als Repräsentation unserer Daten:

In[51]:

```
# extrahiere die Zielwerte (Anzahl Vermietungen)
y = citibike.values
# wandle die Zeit mit "%s" in POSIX-Zeit um
X = citibike.index.strftime("%s").astype("int").reshape(-1, 1)
```

Wir definieren zunächst eine Funktion zum Aufteilen der Daten in Trainings- und Testdatensätze, konstruieren das Modell und plotten das Ergebnis:

In[52]:

```
# verwende die ersten 184 Datenpunkte zum Trainieren, den Rest zum Testen
n_train = 184

# Funktion zum Auswerten und Plotten eines Regressors auf einem gegebenen Satz von
# Merkmalen
def eval_on_features(features, target, regressor):
    # teile die gegebenen Merkmale in Trainings- und Testdaten auf
    X_train, X_test = features[:n_train], features[n_train:]
    # teile auch das Ziel-Array auf
    y_train, y_test = target[:n_train], target[n_train:]
    regressor.fit(X_train, y_train)
    print("Testdatensatz R^2: {:.2f}".format(regressor.score(X_test, y_test)))
```

```

y_pred = regressor.predict(X_test)
y_pred_train = regressor.predict(X_train)
plt.figure(figsize=(10, 3))

plt.xticks(range(0, len(X), 8), xticks.strftime("%a %m-%d"), rotation=90,
           ha="left")

plt.plot(range(n_train), y_train, label="Training")
plt.plot(range(n_train), len(y_test) + n_train, y_test, '-', label="Test")
plt.plot(range(n_train), y_pred_train, '--', label="Vorhersage Training")

plt.plot(range(n_train), len(y_test) + n_train), y_pred, '--',
         label="Vorhersage Test")
plt.legend(loc=(1.01, 0))
plt.xlabel("Datum")
plt.ylabel("Vermietungen")

```

Wir haben bereits festgestellt, dass Random Forests mit sehr wenig Vorverarbeitung der Daten auskommen. Damit ist dies für den Anfang ein gutes Modell. Wir verwenden die POSIX-Zeit als Merkmal X und übergeben unserer Funktion eval\_on\_features einen Random Forest-Regressor. Das Ergebnis sehen Sie in Abbildung 4-13:

**In[53]:**

```

from sklearn.ensemble import RandomForestRegressor
regressor = RandomForestRegressor(n_estimators=100, random_state=0)
eval_on_features(X, y, regressor)

```

**Out[53]:**

Testdatensatz R<sup>2</sup>: -0.04

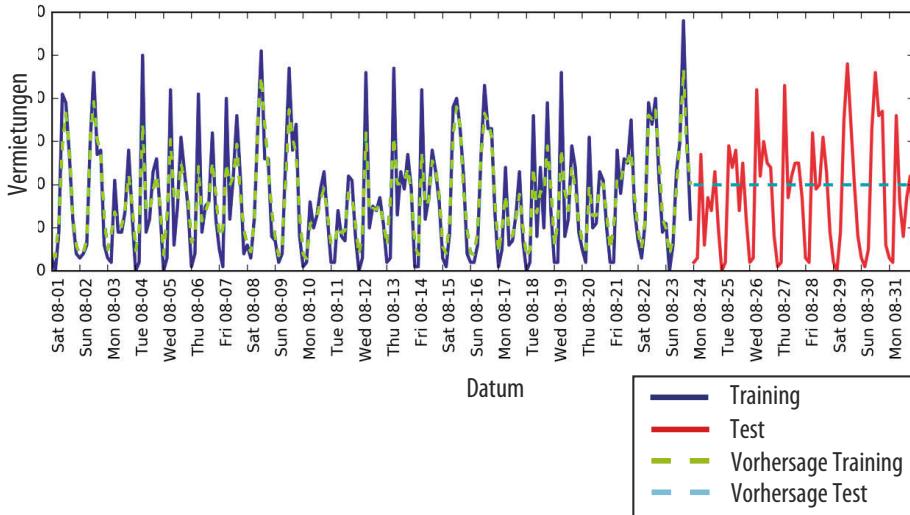


Abbildung 4-13: Vorhersagen durch einen Random Forest unter alleiniger Verwendung der POSIX-Zeit

Die Vorhersagen auf den Trainingsdaten sind recht gut, wie es für Random Forests typisch ist. Allerdings wird für die Testdaten eine konstante Linie vorhergesagt. Der  $R^2$ -Wert beträgt  $-0.04$ , was bedeutet, dass wir nichts gelernt haben. Was ist passiert?

Das Problem ist die Kombination unseres Merkmals mit dem Random Forest. Der Wert der POSIX-Zeit liegt für die Testdaten außerhalb des Bereichs der Werte des Trainingsdatensatzes: Sämtliche Punkte im Testdatensatz haben spätere Zeitstempel als sämtliche Punkte im Trainingsdatensatz. Bäume und damit auch Random Forests können nicht *extrapolieren*, um Werte für Merkmale außerhalb der Trainingsdaten abzubilden. Als Ergebnis sagt das Modell einfach den Wert für den nächsten Punkt im Trainingsdatensatz vorher – weil dies der letzte Zeitpunkt ist, zu dem überhaupt Daten vorliegen.

Natürlich können wir das Modell verbessern. An dieser Stelle kommt unser »Expertenwissen« ins Spiel. Beim Betrachten der Vermietungszahlen im Trainingsdatensatz sehen wir, dass zwei Faktoren ausschlaggebend sind: die Tageszeit und der Wochentag. Mit der POSIX-Zeit können wir nicht wirklich etwas anfangen, also verwerfen wir diese. Zunächst probieren wir nur die Tageszeit. Wie in Abbildung 4-14 zu sehen ist, folgen die Vorhersagen für jeden Wochentag dem gleichen Muster:

**In[54]:**

```
X_hour = citibike.index.hour.reshape(-1, 1)
eval_on_features(X_hour, y, regressor)
```

**Out[54]:**

Testdaten  $R^2$ : 0.60

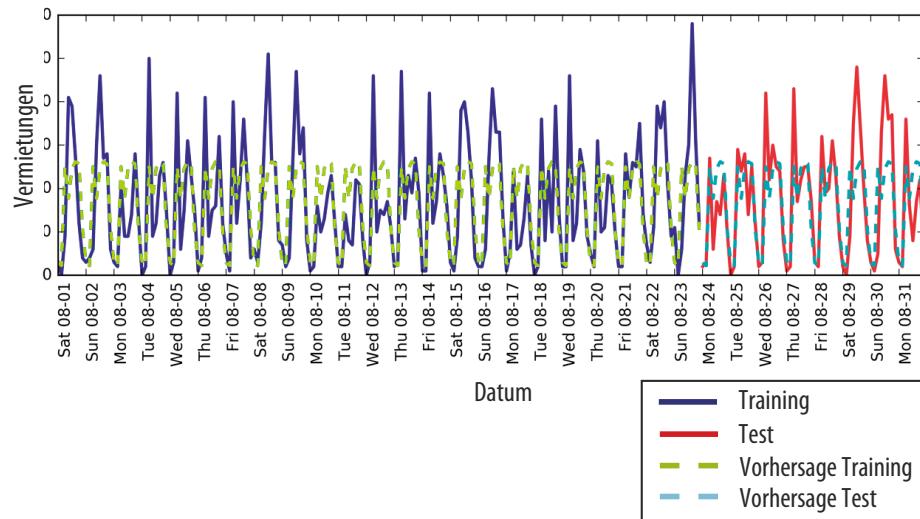


Abbildung 4-14: Vorhersagen durch einen Random Forest allein aus der Tageszeit

Der  $R^2$ -Wert ist schon sehr viel besser, aber die Vorhersagen bilden den Wochenrhythmus nicht ab. Also fügen wir auch noch den Wochentag hinzu (siehe Abbildung 4-15):

**In[55]:**

```
X_hour_week = np.hstack([citibike.index.dayofweek.reshape(-1, 1),
                         citibike.index.hour.reshape(-1, 1)])
eval_on_features(X_hour_week, y, regressor)
```

**Out[55]:**

Testdaten  $R^2$ : 0.84

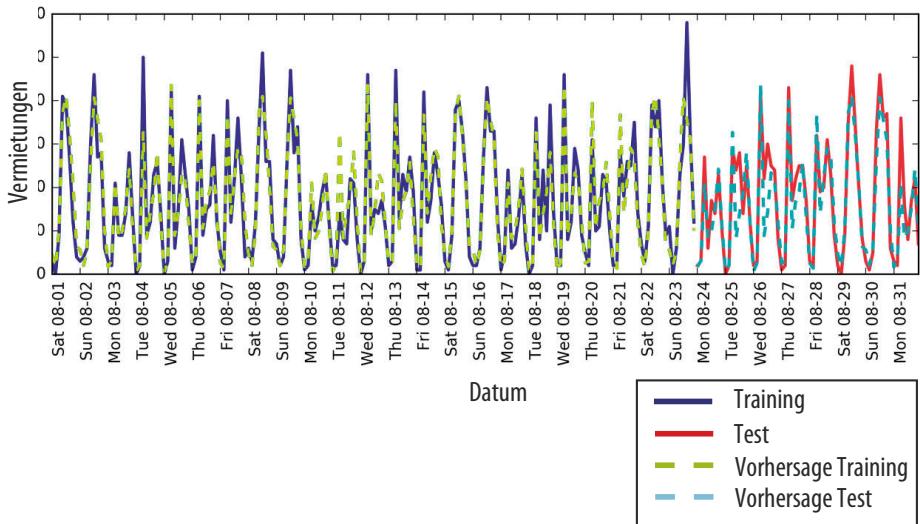


Abbildung 4-15: Vorhersage mit einem Random Forest unter Verwendung von Wochentag und Stunde

Nun haben wir ein Modell, das das periodische Verhalten durch Berücksichtigen von Wochentag und Tageszeit erfasst. Es besitzt einen  $R^2$ -Wert von 0.84 und hat eine recht gute Vorhersagequalität. Höchstwahrscheinlich lernt dieses Modell die mittlere Anzahl Vermietungen für jede Kombination von Wochentag und Tageszeit aus den ersten 23 Tagen im August. Dies erfordert eigentlich kein komplexes Modell wie einen Random Forest, daher probieren wir als einfacheres Modell LinearRegression aus (siehe Abbildung 4-16):

**In[56]:**

```
from sklearn.linear_model import LinearRegression
eval_on_features(X_hour_week, y, LinearRegression())
```

**Out[56]:**

Testdaten  $R^2$ : 0.13

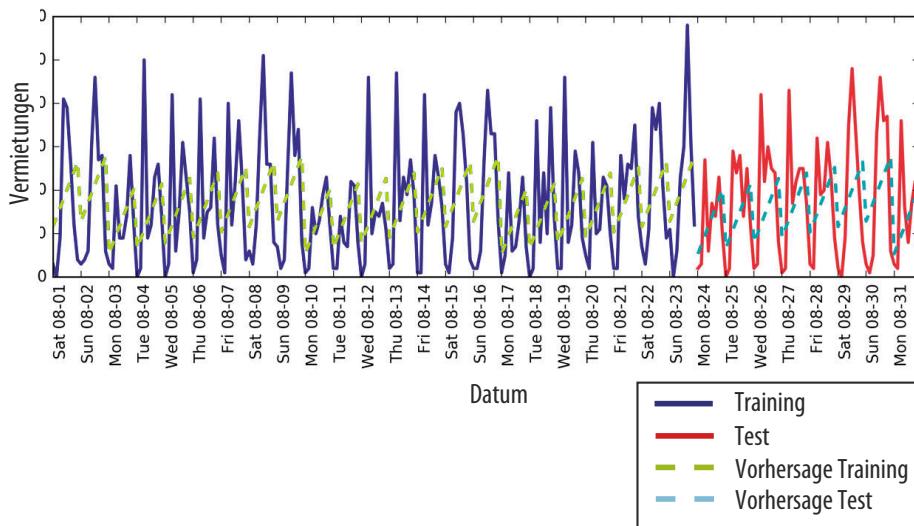


Abbildung 4-16: Vorhersage mittels linearer Regression mit Wochentag und Tageszeit als Merkmale

LinearRegression funktioniert viel schlechter, und die Periodizität sieht seltsam aus. Grund dafür ist, dass wir den Wochentag und die Uhrzeit als Integerzahlen kodiert haben, die als kontinuierliche Variablen interpretiert werden. Daher kann das lineare Modell nur eine lineare Funktion der Tageszeit lernen – und es hat gelernt, dass es zu späteren Tageszeiten mehr Vermietungen gibt. Allerdings sind die wirklichen Muster weitaus komplexer. Wir können dies erfassen, indem wir die Integerzahlen als kategorische Variablen auffassen und sie mit dem OneHotEncoder transformieren (siehe Abbildung 4-17):

**In[57]:**

```
enc = OneHotEncoder()
X_hour_week_onehot = enc.fit_transform(X_hour_week).toarray()
```

**In[58]:**

```
eval_on_features(X_hour_week_onehot, y, Ridge())
```

**Out[58]:**

```
Testdaten R^2: 0.62
```

Wir erhalten eine viel bessere Übereinstimmung als mit dem kontinuierlich kodierten Merkmal. Nun lernt das lineare Modell einen Koeffizienten pro Wochentag und einen pro Tageszeit. Das bedeutet allerdings, dass die »tageszeitliche Periodizität« über alle Wochentage verteilt ist.

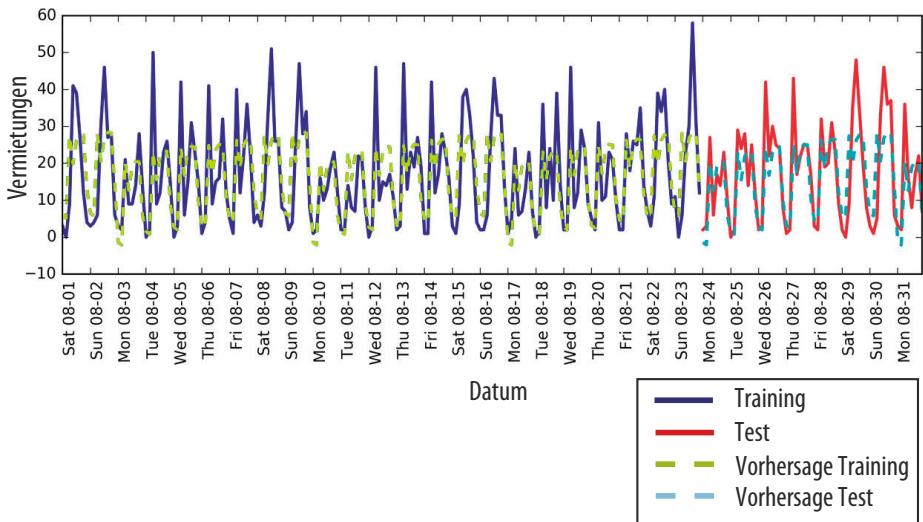


Abbildung 4-17: Vorhersage durch lineare Regression mit One-Hot-Kodierung der Tageszeit und des Wochentages

Mit Interaktions-Merkmalen können wir einen Koeffizienten für jede Kombination von Wochentag und Tageszeit im Modell zulassen (siehe Abbildung 4-18):

**In[59]:**

```
poly_transformer = PolynomialFeatures(degree=2, interaction_only=True,
                                     include_bias=False)
X_hour_week_onehot_poly = poly_transformer.fit_transform(X_hour_week_onehot)
lr = Ridge()
eval_on_features(X_hour_week_onehot_poly, y, lr)
```

**Out[59]:**

Testdaten R<sup>2</sup>: 0.85

Mit dieser Transformation erhalten wir endlich ein Modell, das ähnlich gut wie der Random Forest abschneidet. Ein großer Vorteil dieses Modells ist, dass sehr deutlich wird, was es gelernt hat: einen Koeffizienten für jeden Tag und jede Uhrzeit. Wir können die vom Modell ermittelten Koeffizienten einfach plotten, was mit einem Random Forest nicht möglich wäre.

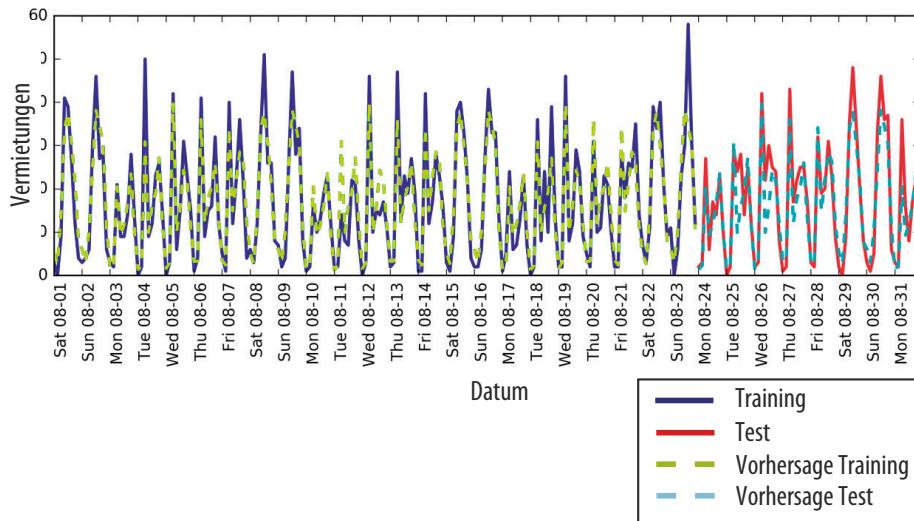


Abbildung 4-18: Vorhersage mit linearer Regression über das Produkt der Merkmale Wochentag und Tageszeit

Dazu erstellen wir zuerst die Namen der Merkmale für Stunde und Wochentag:

**In[60]:**

```
hour = ["%02d:00" % i for i in range(0, 24, 3)]
day = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
features = day + hour
```

Dann ermitteln wir mit der `get_feature_names` sämtliche von `PolyomialFeatures` extrahierten Interaktions-Merkmale und übernehmen nur die Merkmale mit Koeffizienten ungleich null:

**In[61]:**

```
features_poly = poly_transformer.get_feature_names(features)
features_nonzero = np.array(features_poly)[lr.coef_ != 0]
coef_nonzero = lr.coef_[lr.coef_ != 0]
```

Nun können wir die vom linearen Modell erlernten Koeffizienten wie in Abbildung 4-19 darstellen:

**In[62]:**

```
plt.figure(figsize=(15, 2))
plt.plot(coef_nonzero, 'o')
plt.xticks(np.arange(len(coef_nonzero)), features_nonzero, rotation=90)
plt.xlabel("Stärke des Merkmals")
plt.ylabel("Name des Merkmals")
```

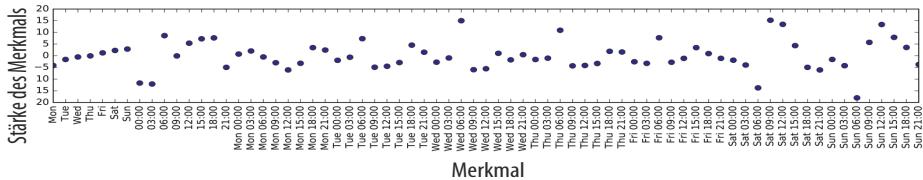


Abbildung 4-19: Koeffizienten des linearen Regressionsmodells mit einem Produkt von Stunde und Tag

## Zusammenfassung und Ausblick

In diesem Kapitel haben wir besprochen, wie man mit unterschiedlichen Arten von Daten umgeht (insbesondere kategorischen Variablen). Wir haben betont, wie wichtig es ist, dass die Daten in einer für den maschinellen Lernalgorithmus geeigneten Art und Weise repräsentiert werden – zum Beispiel als One-Hot-kodierte kategorische Variablen. Wir haben auch die Wichtigkeit des Generierens neuer Merkmale besprochen sowie die Möglichkeit, Expertenwissen zum Erstellen abgeleiteter Merkmale zu nutzen. Insbesondere können lineare Modelle stark vom Erstellen neuer Merkmale durch Klassenbildung, Polynome und Interaktionen profitieren. Komplexe nichtlineare Modelle wie Random Forests und SVMs sind dagegen in der Lage, komplexere Aufgaben ohne explizites Auffächern des Merkmalsraumes zu bewältigen. In der Praxis sind die verwendeten Merkmale (und wie Merkmale und Methoden zusammengefasst werden) oft die wichtigste Komponente, die zum guten Funktionieren eines maschinellen Lernprojekts beiträgt.

Da Sie nun eine Vorstellung davon haben, wie Sie Ihre Daten angemessen repräsentieren können und welcher Algorithmus für welche Aufgabe geeignet ist, konzentrieren wir uns im nächsten Kapitel darauf, die Qualität maschineller Lernmodelle zu bewerten und die richtigen Parameter auszuwählen.



# Evaluierung und Verbesserung von Modellen

Wir haben nun die Grundlagen von überwachtem und unüberwachtem Lernen besprochen und dabei viele maschinelle Lernalgorithmen ausprobiert. Wir werden uns jetzt intensiver mit der Evaluierung von Modellen und der Auswahl von Parametern auseinandersetzen.

Wir werden uns dabei auf überwachte Methoden zur Regression und Klassifikation konzentrieren, da die Evaluierung und Auswahl unüberwachter Lernmodelle oft ein eher qualitativer Prozess ist (wie in Kapitel 3 vorgestellt).

Um unsere überwachten Modelle zu evaluieren, haben wir bisher unsere Datensätze mit der Funktion `train_test_split` in einen Trainingsdatensatz und einen Testdatensatz unterteilt, auf den Trainingsdaten ein Modell über die Methode `fit` erstellt und es anschließend über die Methode `score` auf den Testdaten ausgewertet. Bei Klassifikationsaufgaben berechnet `score` den Anteil korrekt vorhergesagter Datenpunkte. Hier folgt ein Beispiel für diesen Vorgang:

**In[1]:**

```
from sklearn.datasets import make_blobs
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

# erstelle einen synthetischen Datensatz
X, y = make_blobs(random_state=0)
# unterteile Daten und Labels in Trainings- und Testdatensatz
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
# instanziiere ein Modell und passe es auf den Trainingsdaten an
logreg = LogisticRegression().fit(X_train, y_train)
# evaluiere das Modell auf dem Testdatensatz
print("Genauigkeit auf den Testdaten: {:.2f}".format(logreg.score(X_test, y_test)))
```

**Out[1]:**

Genauigkeit auf den Testdaten: 0.88

Der Grund, aus dem wir unsere Daten in Trainings- und Testdaten unterteilen, ist bekanntlich, dass wir an der Fähigkeit unseres Modells zur *Verallgemeinerung* auf neue, bisher unbekannte Daten interessiert sind. Wir interessieren uns nicht so sehr dafür, wie gut das Modell auf den Trainingsdaten abschneidet, sondern vielmehr dafür, wie gute Vorhersagen für im Training nicht beobachtete Daten getroffen werden.

In diesem Kapitel werden wir zwei Aspekte dieser Art von Evaluierung genauer ausführen. Wir werden zuerst den Begriff der Kreuzvalidierung einführen, eine robustere Möglichkeit zum Bewerten der Modellqualität. Wir werden ebenfalls Methoden zum Auswerten der Klassifikations- und Regressionsleistung besprechen, die über die von der Methode `score` berechneten Maße Genauigkeit und  $R^2$  hinausgehen. Wir werden anschließend die Gittersuche besprechen, eine effektive Methode zum Einstellen der Parameter in überwachten Modellen, um dessen Verallgemeinerungsleistung zu optimieren.

## Kreuzvalidierung

*Kreuzvalidierung* ist eine statistische Methode zum Auswerten der Verallgemeinerungsleistung, die im Vergleich zum Aufteilen in Trainings- und Testdaten stabiler und gründlicher ist. Bei der Kreuzvalidierung werden die Daten wiederholt aufgeteilt, und dabei werden mehrere Modelle trainiert. Die häufigste Variante der Kreuzvalidierung ist die *k-fache Kreuzvalidierung*, bei der  $k$  eine vom Nutzer angegebene Zahl ist, normalerweise 5 oder 10. Beim Durchführen einer fünffachen Kreuzvalidierung werden die Daten zuerst in fünf (etwa) gleich große Teile, genannt *Folds*, aufgeteilt. Anschließend wird eine Folge von Modellen trainiert. Beim ersten Modell wird der erste Fold als Testdatensatz beiseitegelegt, und die übrigen Folds (2–5) bilden den Trainingsdatensatz. Das Modell wird mit den Daten der Folds 2–5 konstruiert, und dann berechnen wir mit Fold 1 die Genauigkeit. Anschließend wird ein weiteres Modell gebaut, diesmal mit Fold 2 als Testdatensatz und den Daten der Folds 1, 3, 4 und 5 als Trainingsdatensatz. Dieser Vorgang wird mit den Folds 3, 4 und 5 als Testdatensatz wiederholt. Bei jeder dieser fünf *Teilungen* der Daten in Trainings- und Testdatensätze berechnen wir eine Genauigkeit. Am Ende haben wir fünf Werte für die Genauigkeit gesammelt. Dieser Prozess ist in Abbildung 5-1 dargestellt:

In[2]:

```
mglearn.plots.plot_cross_validation()
```

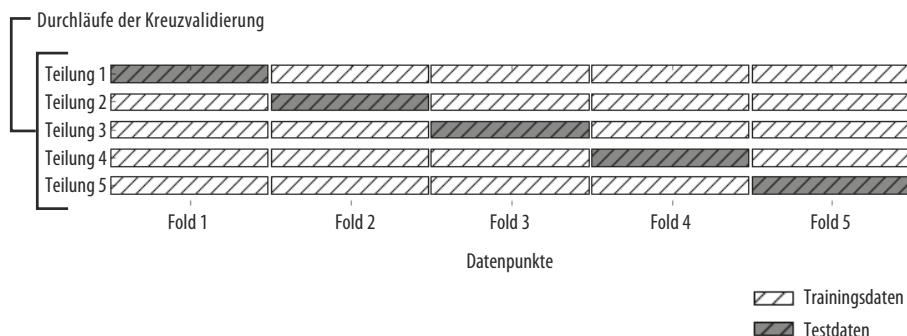


Abbildung 5-1: Aufteilung der Daten bei einer fünffachen Kreuzvalidierung

Für gewöhnlich bildet das erste Fünftel der Daten den ersten Fold, das zweite Fünftel der Daten den zweiten Fold usw.

## Kreuzvalidierung in scikit-learn

Kreuzvalidierung ist in scikit-learn als Funktion `cross_val_score` im Modul `model_selection` implementiert. Die Parameter der Funktion `cross_val_score` sind das zu evaluierende Modell, die Trainingsdaten und die bekannten Labels. Werten wir als Beispiel `LogisticRegression` auf dem Datensatz `iris` aus:

**In[3]:**

```
from sklearn.model_selection import cross_val_score
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression

iris = load_iris()
logreg = LogisticRegression()

scores = cross_val_score(logreg, iris.data, iris.target)
print("Scores der Kreuzvalidierung: {}".format(scores))
```

**Out[3]:**

```
Scores der Kreuzvalidierung: [ 0.961  0.922  0.958]
```

Standardmäßig nimmt `cross_val_score` eine dreifache Kreuzvalidierung vor, die zu drei Genauigkeitswerten führt. Wir können die Anzahl der Folds über den Parameter `cv` einstellen:

**In[4]:**

```
scores = cross_val_score(logreg, iris.data, iris.target, cv=5)
print("Scores der Kreuzvalidierung: {}".format(scores))
```

**Out[4]:**

```
Scores der Kreuzvalidierung: [ 1.      0.967  0.933  0.9     1.     ]
```

Die Genauigkeit aus der Kreuzvalidierung wird häufig als Mittelwert zusammengefasst:

**In[5]:**

```
print("durchschnittlicher Score der Kreuzvalidierung: {:.2f}".format(scores.mean()))
```

**Out[5]:**

```
durchschnittlicher Score der Kreuzvalidierung: 0.96
```

Mit dem Mittelwert der Kreuzvalidierung können wir den Rückschluss ziehen, dass unser Modell im Durchschnitt eine Genauigkeit von 96 % erzielt. Betrachten wir alle fünf von der Kreuzvalidierung zurückgegebenen Werte, bemerken wir außerdem, dass die Genauigkeit von Fold zu Fold stark zwischen 100 % und 90 % schwankt. Dies könnte bedeuten, dass das Modell deutlich von bestimmten zum

Trainieren verwendeten Folds abhängig ist. Es könnte aber auch einfach daran liegen, dass der Datensatz klein ist.

## Vorteile der Kreuzvalidierung

Eine Kreuzvalidierung anstelle einer einfachen Teilung in Trainings- und Testdaten zu verwenden, hat mehrere Vorteile. Erinnern wir uns daran, dass `train_test_split` den Datensatz zufällig aufteilt. Wenn wir dabei »Glück« haben und alle schwierig zu klassifizierenden Beispiele im Trainingsdatensatz landen, befinden sich in den Testdaten nur »leichte« Beispiele. Die Genauigkeit für den Testdatensatz wäre in diesem Fall unrealistisch hoch. Haben wir dagegen »Pech,« so landen alle schwierig zu klassifizierenden Beispiele in den Testdaten, und die Genauigkeit wird dadurch unrealistisch niedrig. Bei einer Kreuzvalidierung dagegen befindet sich jedes Beispiel genau einmal in den Testdaten: Jeder Datenpunkt befindet sich in einem der Folds, und jeder Fold wird einmal als Testdatensatz verwendet. Damit muss das Modell gut für alle Punkte im Datensatz verallgemeinern, damit die Scores der Kreuzvalidierung (und ihr Mittelwert) hoch sind.

Mehreren Teilungen des Datensatzes geben uns auch darüber Aufschluss, wie sensibel unser Modell auf die Auswahl des Trainingsdatensatzes reagiert. Beim Datensatz `iris` haben wir Genauigkeiten zwischen 90 % und 100 % beobachtet. Dies ist eine recht große Spannbreite und verrät uns, wie sich das Modell bei neuen Daten im bestmöglichen und im schlechtesten möglichen Fall verhalten könnte.

Ein weiterer Nutzen der Kreuzvalidierung gegenüber einer Teilung der Daten ist, dass wir unsere Daten effektiver nutzen. Beim Verwenden von `train_test_split` verwenden wir normalerweise 75 % der Daten zum Trainieren und 25 % der Daten zur Auswertung. Bei einer fünffachen Kreuzvalidierung können wir in jedem Durchgang vier Fünftel der Daten (80 %) zum Trainieren des Modells verwenden. Bei einer zehnfachen Kreuzvalidierung können wir neun Zehntel der Daten (90 %) verwenden. Mehr Daten führen üblicherweise zu einem genaueren Modell.

Der Hauptnachteil der Kreuzvalidierung ist der zusätzliche Rechenaufwand. Da wir nun  $k$  Modelle anstatt eines einzelnen Modells trainieren, dauert die Kreuzvalidierung etwa  $k$  Mal so lange wie eine einzelne Teilung der Daten.



Bei der Kreuzvalidierung ist es wichtig zu bedenken, dass dabei kein Modell entsteht, dass sich auf neue Daten anwenden lässt. Bei der Kreuzvalidierung wird kein Modell zurückgegeben. Beim Aufruf von `cross_val_score` werden intern mehrere Modell konstruiert, aber der Zweck einer Kreuzvalidierung ist, auszuwerten, wie gut ein Algorithmus auf einem gegebenen Datensatz verallgemeinern kann.

## Stratifizierte k-fache Kreuzvalidierung und andere Strategien

Das Aufteilen eines Datensatzes wie im letzten Abschnitt in  $k$  Folds vom ersten  $k$ -tel der Daten an ist nicht immer eine gute Idee. Betrachten wir als Beispiel den Datensatz `iris`:

In[6]:

```
from sklearn.datasets import load_iris  
iris = load_iris()  
print("Iris Labels:\n{}".format(iris.target))
```

Out[6]:

Wie Sie sehen, ist das erste Drittel der Daten der Kategorie 0 zugeordnet, das zweite Drittel der Kategorie 1 und das letzte Drittel der Kategorie 2. Stellen Sie sich eine dreifache Kreuzvalidierung auf diesem Datensatz vor. Der erste Fold würde ausschließlich Kategorie 0 enthalten, damit bestünde der erste Testdatensatz nur aus Kategorie 0, während die Trainingsdaten nur die Kategorien 1 und 2 enthalten. Da die Kategorien in allen drei Folds in Trainings- und Testdaten jeweils unterschiedlich sind, wäre die Genauigkeit der dreifachen Kreuzvalidierung für diesen Datensatz gleich null. Dies wäre nicht sonderlich hilfreich, da wir für *iris* eine deutlich bessere Genauigkeit als 0 % erzielen können.

Da die einfache  $k$ -fache Strategie hier scheitert, verwendet scikit-learn diese nicht zur Klassifizierung. Stattdessen wird die *stratifizierte  $k$ -fache Kreuzvalidierung* eingesetzt. Bei der stratifizierten Kreuzvalidierung teilen wir die Daten so auf, dass die Proportionen zwischen den Kategorien in jedem Fold die gleichen wie im gesamten Datensatz sind. Dies ist in Abbildung 5-2 dargestellt:

In[7]:

```
mglearn.plots.plot_stratified_cross_validation()
```

Wenn beispielsweise 90 % der Datenpunkte Kategorie A und 10 % der Datenpunkte Kategorie B angehören, stellt die stratifizierte Kreuzvalidierung sicher, dass in jedem Fold 90 % der Datenpunkte aus Kategorie A und 10 % aus Kategorie B ausgewählt werden.

Normalerweise ist es eine gute Idee, zum Auswerten eines Klassifikators die stratifizierte  $k$ -fache Kreuzvalidierung anstatt der einfachen  $k$ -fachen Kreuzvalidierung einzusetzen, weil sie zu zuverlässigeren Schätzungen der Verallgemeinerungsleistung führt. Falls nur 10 % der Datenpunkte zu Kategorie B gehören, geschieht es bei der einfachen  $k$ -fachen Kreuzvalidierung leicht, dass ein Fold nur Punkte aus Kategorie A enthält. Damit wäre dieser Fold als Testdatensatz für die Gesamtleistung des Klassifikators nicht besonders aussagekräftig.

Bei der Regression verwendet scikit-learn standardmäßig die  $k$ -fache Kreuzvalidierung. Man könnte auch hier versuchen, jeden Fold die unterschiedlichen Werte der Zielgröße repräsentieren zu lassen, aber dies ist kein allgemein gebräuchlicher Ansatz und würde die meisten Nutzer eher überraschen.

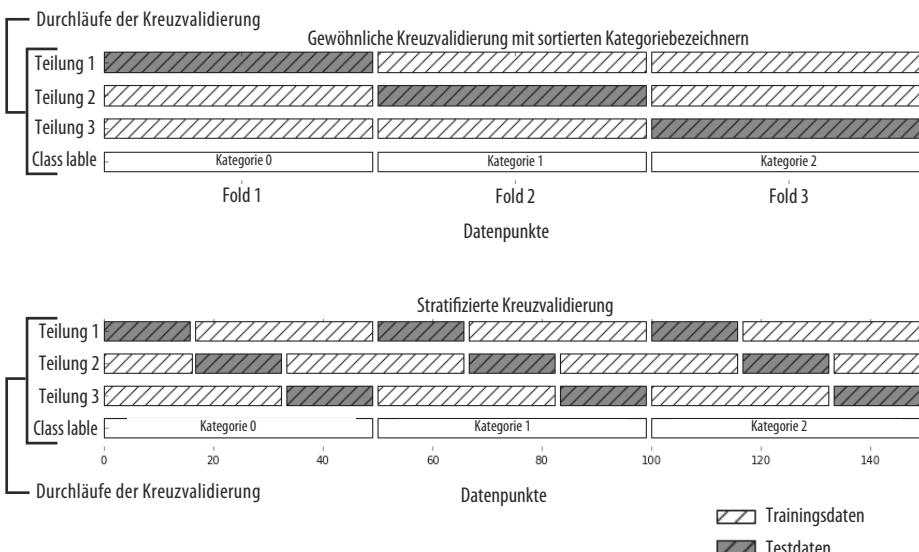


Abbildung 5-2: Vergleich zwischen der einfachen Kreuzvalidierung und der stratifizierten Kreuzvalidierung, wenn die Daten nach Kategorien sortiert sind

## Zusätzliche Kontrolle über die Kreuzvalidierung

Wir haben bereits gesehen, dass wir die Anzahl der in `cross_val_score` verwendeten Folds über den Parameter `cv` beeinflussen können. Wir können die Aufteilung der Daten in scikit-learn aber noch viel genauer steuern, indem wir einen *Kreuzvalidierungs-Splitter* als Parameter `cv` angeben. In den meisten Fällen funktioniert die voreingestellte  $k$ -fache Kreuzvalidierung bei der Regression und die stratifizierte  $k$ -fache Kreuzvalidierung bei der Klassifikation gut. Es gibt aber einige Fälle, in denen Sie eine andere Strategie bevorzugen sollten. Nehmen wir an, wir möchten die  $k$ -fache Kreuzvalidierung einsetzen, um die Ergebnisse von jemand anderem zu reproduzieren. Dazu müssen wir die Splitter-Klasse `KFold` aus dem Modul `model_selection` importieren und eine Instanz mit der Anzahl der gewünschten Folds bilden:

In[8]:

```
from sklearn.model_selection import KFold  
kfolds = KFold(n_splits=5)
```

Dann können wir das Splitter-Objekt `kfold` als Parameter `cv` an die Funktion `cross_val_score` übergeben:

In[9]:

```
print("Scores der Kreuzvalidierung:\n{}".format(  
    cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

**Out[9]:**

```
Scores der Kreuzvalidierung:  
[ 1. 0.933 0.433 0.967 0.433]
```

Auf diese Weise können wir verifizieren, dass eine dreifache (nicht stratifizierte) Kreuzvalidierung auf dem Datensatz `iris` eine wirklich sehr schlechte Idee ist:

**In[10]:**

```
kfold = KFold(n_splits=3)  
print("Scores der Kreuzvalidierung:\n{}".format(  
    cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

**Out[10]:**

```
Scores der Kreuzvalidierung:  
[ 0. 0. 0.]
```

Bedenken Sie, dass jeder Fold im Datensatz `iris` einer der Kategorien entspricht, es also nichts zu lernen gibt. Eine andere Möglichkeit zur Lösung dieses Problems wäre, anstatt der Stratifizierung den Datensatz einfach zu durchmischen, sodass die Anordnung nach Kategorien verschwindet. Dies erreichen wir, indem wir den Parameter `shuffle` bei `KFold` auf `True` setzen. Wenn wir die Daten durchmischen, müssen wir auch einen Wert für `random_state` festlegen, damit wir eine reproduzierbare Reihenfolge erhalten. Andernfalls würden wir bei jedem Aufruf von `cross_val_score` ein anderes Ergebnis erhalten, da ihr eine unterschiedliche Aufteilung zugrunde liegt (dies ist nicht unbedingt ein Problem, führt aber zu Überraschungen). Das Durchmischen vor der Aufteilung führt zu einem deutlich besseren Ergebnis:

**In[11]:**

```
kfold = KFold(n_splits=3, shuffle=True, random_state=0)  
print("Scores der Kreuzvalidierung:\n{}".format(  
    cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

**Out[11]:**

```
Scores der Kreuzvalidierung:  
[ 0.9 0.96 0.96]
```

## Leave-One-Out-Kreuzvalidierung

Eine weitere, oft genutzte Methode zur Kreuzvalidierung ist *Leave-One-Out*. Sie können sich die Leave-One-Out-Kreuzvalidierung als eine  $k$ -fache Kreuzvalidierung vorstellen, bei der jeder Fold ein einzelner Datenpunkt ist. Bei jeder Teilung wählen Sie einen einzelnen Datenpunkt als Testdatensatz aus. Dies ist besonders bei großen Datensätzen sehr zeitaufwendig, führt aber bei kleineren Datensätzen manchmal zu genaueren Schätzungen:

**In[12]:**

```
from sklearn.model_selection import LeaveOneOut  
loo = LeaveOneOut()
```

```

scores = cross_val_score(logreg, iris.data, iris.target, cv=loo)
print("Anzahl Iterationen bei der KV: ", len(scores))
print("mittlere Genauigkeit: {:.2f}".format(scores.mean()))

```

**Out[12]:**

```

Anzahl Iterationen bei der KV: 150
mittlere Genauigkeit: 0.95

```

## Shuffle-Split-Kreuzvalidierung

Ein weiterer, sehr flexibler Ansatz zur Kreuzvalidierung ist die *Shuffle-Split-Kreuzvalidierung*. Bei der Shuffle-Split-Kreuzvalidierung werden bei jeder Teilung `train_size` Punkte als Trainingsdaten und `test_size` (andere) Punkte als Testdaten ausgewählt. Diese Art der Aufteilung wird `n_iter` Mal wiederholt. In Abbildung 5-3 sind vier Iterationen dargestellt, in denen aus einem Datensatz mit zehn Punkten Trainingsdatensätze mit je fünf Punkten und Testdatensätze mit je zwei Punkten generiert werden (Sie können für `train_size` und `test_size` ganzzahlige Zahlen als Absolutwerte oder Dezimalbrüche als Anteile des gesamten Datensatzes angeben):

**In[13]:**

```
mglearn.plots.plot_shuffle_split()
```

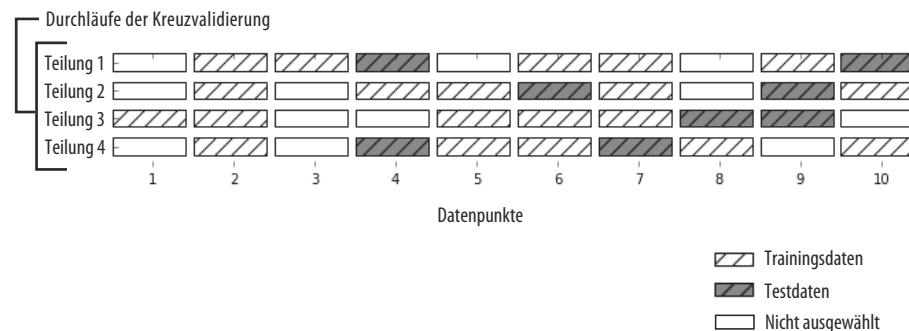


Abbildung 5-3: *ShuffleSplit* mit zehn Punkten, `train_size=5`, `test_size=2` und `n_iter=4`

Der folgende Code teilt den Datensatz über zehn Iterationen in 50 % Trainingsdatensatz und 50 % Testdaten auf:

**In[14]:**

```

from sklearn.model_selection import ShuffleSplit
shuffle_split = ShuffleSplit(test_size=.5, train_size=.5, n_splits=10)
scores = cross_val_score(logreg, iris.data, iris.target, cv=shuffle_split)
print("Scores der Kreuzvalidierung:\n{}\n".format(scores))

```

**Out[14]:**

```

Scores der Kreuzvalidierung:
[ 0.96   0.907   0.947   0.96   0.96   0.907   0.893   0.907   0.92   0.973]

```

Mit der Shuffle-Split-Kreuzvalidierung können wir die Anzahl der Iterationen unabhängig von der Größe der Trainings- und Testdatensätze steuern, was manchmal hilfreich ist. Wir können so auch in jeder Iteration lediglich einen Teil der Daten verwenden, indem wir für `train_size` und `test_size` Werte angeben, die zusammen weniger als 1 ergeben. Diese Art der Stichprobenbildung ist beim Experimentieren mit großen Datensätzen nützlich.

Es gibt auch eine stratifizierte Variante von `ShuffleSplit`, die passenderweise `StratifiedShuffleSplit` heißt. Mit dieser lassen sich bei Klassifikationsaufgaben zuverlässigere Ergebnisse erzielen.

### Kreuzvalidierung mit Gruppen

Ein weiterer Parameter bei der Kreuzvalidierung kommt zum Einsatz, falls es innerhalb der Daten zueinander ähnliche Gruppen gibt. Nehmen wir an, Sie möchten ein System zum Erkennen von Emotionen in Bildern von Gesichtern entwickeln. Dazu erstellen Sie einen Datensatz mit Bildern von 100 Personen, wobei jede Person mehrmals mit unterschiedlichen Gesichtsausdrücken aufgenommen wird. Es soll ein Klassifikator erstellt werden, der Personen, die nicht im Datensatz enthalten sind, die korrekte Emotion zuordnet. Sie könnten die Vorhersagekraft des Klassifikators über eine gewöhnliche stratifizierte Kreuzvalidierung bestimmen. Allerdings ist es so wahrscheinlich, dass sich Bilder der gleichen Person sowohl in den Trainings- als auch in den Testdaten befinden. Für einen Klassifikator ist es voraussichtlich deutlich einfacher, Emotionen in einem aus den Trainingsdaten bekannten Gesicht im Gegensatz zu einem komplett neuen Gesicht zu erkennen. Um die Verallgemeinerung auf neue Gesichter sinnvoll zu evaluieren, müssen wir also sicherstellen, dass Trainings- und Testdatensatz Gesichter von unterschiedlichen Personen enthalten.

Zu diesem Zweck können wir `GroupKFold` verwenden, das ein Array mit der Angabe der Personen in den Bildern als Argument `groups` annimmt. Das Array `groups` zeigt hierbei Gruppen in den Daten an, die beim Erstellen von Trainings- und Testdaten nicht aufgeteilt werden sollen. Man sollte diese Gruppen nicht mit den Kategorielabels verwechseln.

Diese Art von Gruppen in den Daten ist bei medizinischen Anwendungen häufig, bei denen es für einen Patienten mehrere Datenpunkte geben kann, Sie sich aber für eine Verallgemeinerung auf neue Patienten interessieren. In ähnlicher Weise könnten Sie bei Spracherkennung mehrere Aufnahmen des gleichen Sprechers in Ihrem Datensatz haben, aber an der Spracherkennung neuer Sprecher interessiert sein.

Das folgende Beispiel verwendet einen synthetischen Datensatz mit einer durch das Array `groups` angegebenen Gruppierung. Der Datensatz besteht aus zwölf Datenpunkten. `groups` gibt für jeden Datenpunkt an, zu welcher Gruppe (z. B. Patienten) dieser Punkt gehört. In diesem Fall geben die Gruppen an, dass es vier Gruppen gibt, die ersten drei Punkte zur ersten Gruppe gehören, die nächsten vier Punkte zur zweiten Gruppe usw.:

In[15]:

```
from sklearn.model_selection import GroupKFold
# synthetischen Datensatz erstellen
X, y = make_blobs(n_samples=12, random_state=0)
# nimm an, dass die ersten drei Punkte zur gleichen Gruppe
# gehören, dann die nächsten vier usw.
groups = [0, 0, 0, 1, 1, 1, 2, 2, 3, 3, 3]
scores = cross_val_score(logreg, X, y, groups, cv=GroupKFold(n_splits=3))
print("Scores der Kreuzvalidierung:\n{}".format(scores))
```

Out[15]:

```
Scores der Kreuzvalidierung:
[ 0.75    0.8     0.667]
```

Die Datenpunkte müssen nicht nach Gruppen sortiert sein; wir haben das nur zur Veranschaulichung getan. Die anhand dieser Zuordnung berechneten Teilungen sind in Abbildung 5-4 veranschaulicht. Wie Sie sehen, befindet sich jede Gruppe bei jeder Teilung entweder vollständig im Trainingsdatensatz oder vollständig im Testdatensatz:

In[16]:

```
mglearn.plots.plot_label_kfold()
```

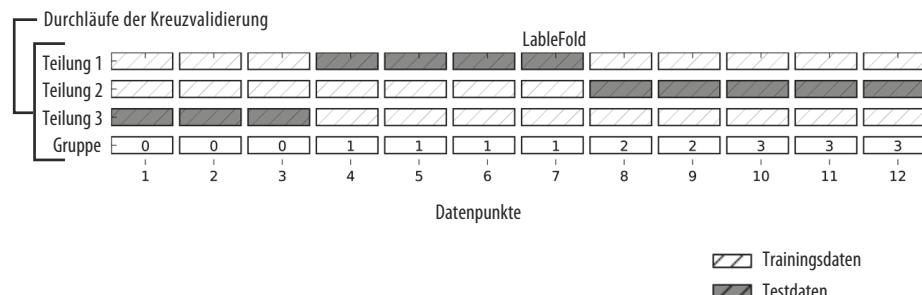


Abbildung 5-4: Von einer Gruppenzuordnung abhängige Aufteilung mit GroupKFold

Es gibt in scikit-learn noch weitere Strategien zur Aufteilung bei der Kreuzvalidierung, die eine noch größere Bandbreite von Anwendungsmöglichkeiten eröffnen (Sie finden diese im scikit-learn User Guide ([http://scikit-learn.org/stable/modules/cross\\_validation.html](http://scikit-learn.org/stable/modules/cross_validation.html))). Allerdings sind die Methoden KFold, StratifiedKFold und GroupKFold die bei Weitem gebräuchlichsten.

## Gittersuche

Weil wir nun wissen, wie wir die Leistung eines Modells zur Verallgemeinerung evaluieren können, werden wir als nächsten Schritt die Verallgemeinerungsleistung eines Modells verbessern, indem wir seine Parameter optimieren. Wir haben in den

Kapiteln 2 und 3 die Parameter vieler Algorithmen in scikit-learn besprochen, und es ist wichtig, vor der Feineinstellung die Bedeutung dieser Parameter zu kennen. Die Werte wichtiger Parameter eines Modells zu finden, mit denen sich die beste Vorhersage erzielen lässt, ist eine schwierige Aufgabe, jedoch bei fast allen Modellen und Datensätzen notwendig. Und weil dies eine derart häufige Aufgabe ist, gibt es in scikit-learn standardisierte Methoden, die Ihnen dabei unter die Arme greifen. Die am häufigsten eingesetzte Methode ist die *Gittersuche*, auch *Rastersuche* genannt, bei der wir praktisch alle möglichen Kombinationen der für uns interessanten Parameter ausprobieren.

Betrachten wir eine Kernel-SVM mit einem RBF-Kernel (Radial Basis Function), wie sie in der Klasse SVC implementiert ist. Wie in Kapitel 2 besprochen, gibt es zwei wichtige Parameter: die Bandbreite des Kernels, gamma, und den Regularisierungsparameter C. Nehmen wir an, wir möchten für die Parameter C und gamma jeweils die Werte 0.001, 0.01, 0.1, 1, 10 und 100 ausprobieren. Weil wir sechs unterschiedliche Werte für C und gamma ausprobieren möchten, gibt es insgesamt 36 Parameterkombinationen. Betrachten wir alle möglichen Kombinationen, ergibt sich die folgende Tabelle (ein Gitter) mit den Paramtern der SVM:

	C = 0.001	C = 0.01	...	C = 10
gamma=0.001	SVC (C=0.001, gamma=0.001)	SVC (C=0.01, gamma=0.001)	...	SVC (C=10, gamma=0.001)
gamma=0.01	SVC (C=0.001, gamma=0.01)	SVC (C=0.01, gamma=0.01)	...	SVC (C=10, gamma=0.01)
...	...	...	...	...
gamma=100	SVC (C=0.001, gamma=100)	SVC (C=0.01, gamma=100)	...	SVC (C=10, gamma=100)

## Einfache Gittersuche

Wir können eine einfache Gittersuche mit zwei for-Schleifen über die zwei Parameter implementieren, indem wir für jede Kombination einen Klassifikator erstellen, trainieren und auswerten:

In[17]:

```
# naive Implementierung der Gittersuche
from sklearn.svm import SVC
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, random_state=0)
print("Größe des Trainingsdatensatzes: {} Größe des Testdatensatzes: {}".format(
    X_train.shape[0], X_test.shape[0]))

best_score = 0

for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # trainiere einen SVC für jede Parameterkombination
```

```

svm = SVC(gamma=gamma, C=C)
svm.fit(X_train, y_train)
# evaluiere den SVC auf den Testdaten
score = svm.score(X_test, y_test)
# wenn der Score besser ist, speichere Score und Parameter
if score > best_score:
    best_score = score
best_parameters = {'C': C, 'gamma': gamma}

print("Bester Score: {:.2f}".format(best_score))
print("Beste Parameter: {}".format(best_parameters))

```

**Out[17]:**

```

Größe des Trainingsdatensatzes: 112   Größe des Testdatensatzes: 38
Bester Score: 0.97
Beste Parameter: {'C': 100, 'gamma': 0.001}

```

## Die Gefahr des Overfittings von Parametern und Validierungsdaten

Mit diesem Ergebnis sind wir versucht zu behaupten, dass Modell erziele auf unserem Datensatz eine Genauigkeit von 97 %. Allerdings könnte sich diese Behauptung aus folgendem Grund als zu optimistisch (oder schlicht falsch) erweisen: Wir haben viele unterschiedliche Parameter ausprobiert und diejenigen mit der besten Genauigkeit auf dem Testdatensatz ausgewählt. Allerdings lässt sich diese Genauigkeit nicht unbedingt auf neue Daten übertragen. Da wir die Testdaten zum Einstellen der Parameter verwendet haben, dürfen wir diese nicht zum Bewerten der Modellqualität verwenden. Aus dem gleichen Grund hatten wir ursprünglich die Daten in Trainings- und Testdaten aufgeteilt; wir benötigen zur Evaluierung einen unabhängigen Datensatz, der zum Erstellen des Modells nicht verwendet wurde.

Eine mögliche Lösung des Problems ist, die Daten abermals zu teilen, sodass wir drei Datensätze erhalten: die Trainingsdaten zum Erstellen des Modells, die Validierungsdaten zur Auswahl der Parameter für das Modell und die Testdaten, um die Leistung der ausgewählten Parameter zu evaluieren. Abbildung 5-5 zeigt, wie diese Teilung aussieht:

**In[18]:**

```
mglearn.plots.plot_threefold_split()
```

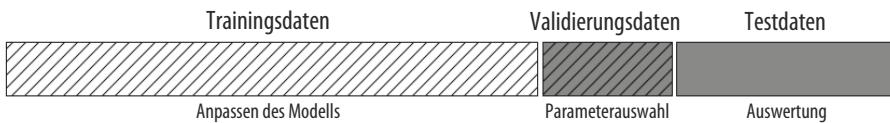


Abbildung 5-5: Dreifache Teilung eines Datensatzes in Trainingsdaten, Validierungsdaten und Testdaten

Nach Auswahl der besten Parameter mit den Validierungsdaten können wir das Modell mit den besten Parametern noch einmal erstellen, nur dass wir diesmal

sowohl die Trainingsdaten als auch die Validierungsdaten verwenden. Auf diese Weise verwenden wir so viele Daten wie möglich, um unser Modell zu erstellen. Damit erhalten wir folgende Implementierung:

**In[19]:**

```
from sklearn.svm import SVC
# teile die Daten in Training+Validierungsdaten und Testdaten
X_trainval, X_test, y_trainval, y_test = train_test_split(
    iris.data, iris.target, random_state=0)
# teile die Training+Validierungsdaten in Trainings- und Validierungsdaten
X_train, X_valid, y_train, y_valid = train_test_split(
    X_trainval, y_trainval, random_state=1)
print("Größe Trainingsdaten: {}  Größe Validierungsdaten: {}  Größe Testdaten: {}"
      " {}\n".format(X_train.shape[0], X_valid.shape[0], X_test.shape[0]))

best_score = 0

for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # trainiere einen SVC für jede Parameterkombination
        svm = SVC(gamma=gamma, C=C)
        svm.fit(X_train, y_train)
        # evaluiere den SVC auf den Validierungsdaten
        score = svm.score(X_valid, y_valid)
        # wenn der Score besser wird, speichere Score und Parameter
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}

# erstelle das Modell erneut mit den kombinierten Trainings-
# und Validierungsdaten und evaluiere es mit den Testdaten
svm = SVC(**best_parameters)
svm.fit(X_trainval, y_trainval)
test_score = svm.score(X_test, y_test)
print("Bester Score auf den Validierungsdaten: {:.2f}".format(best_score))
print("Beste Parameter: ", best_parameters)
print("Score auf den Testdaten mit besten Parametern: {:.2f}".format(test_score))
```

**Out[19]:**

```
Größe Trainingsdaten: 84  Größe Validierungsdaten: 28  Größe Testdaten: 38

Bester Score auf den Validierungsdaten: 0.96
Beste Parameter: {'C': 10, 'gamma': 0.001}
Score auf den Testdaten mit besten Parametern: 0.92
```

Der beste Score auf den Validierungsdaten ist mit 96 % etwas geringer als zuvor, vermutlich weil wir das Modell mit weniger Daten trainiert haben (`X_train` ist nun kleiner, da wir den Datensatz zweimal geteilt haben). Allerdings ist der Score auf den Testdaten – also die Genauigkeit, die uns etwas über die Fähigkeit zur Verallgemeinerung verrät – mit 92 % noch geringer. Damit können wir nur beanspruchen, 92 % der Daten korrekt zu klassifizieren, nicht wie ursprünglich angenommen 97 %!

Die Unterscheidung zwischen Trainingsdaten, Validierungsdaten und Testdaten ist grundlegend wichtig, um maschinelle Lernmethoden in der Praxis anzuwenden. Jegliche aufgrund der Genauigkeit des Testdatensatzes getroffenen Entscheidungen lassen Information aus dem Testdatensatz in das Modell »durchsickern«. Daher ist es wichtig, einen separaten Testdatensatz für die endgültige Auswertung beiseitezulegen. Eine gängige Praxis ist, sämtliche erkundenden Analysen und die Modellauswahl auf den kombinierten Trainings- und Validierungsdaten durchzuführen und die Testdaten für die endgültige Evaluierung aufzuheben – dies betrifft sogar die Visualisierung zum Erkunden der Daten. Streng genommen führt jede Auswertung von zwei oder mehr Modellen auf den Testdaten und die Auswahl des besseren Modells zu einer übermäßig zuversichtlichen Schätzung der Modellgenauigkeit.

## Gittersuche mit Kreuzvalidierung

Die Methode, Daten in Trainings-, Validierungs- und Testdaten aufzuteilen, ist durchführbar und relativ verbreitet. Sie ist aber auch recht störungsanfällig dafür, auf welche Weise die Daten aufgeteilt werden. Aus der Ausgabe des vorigen Codebeispiels sehen wir, dass die Gittersuche die Werte 'C': 10, 'gamma': 0.001 als beste Parameter auswählt, während der Code davor 'C': 100, 'gamma': 0.001 als beste Parameter bestimmt. Für eine bessere Einschätzung der Verallgemeinerungsleistung können wir anstelle einer einzelnen Teilung in Trainings- und Validierungsdaten die einzelnen Parameterkombinationen auch über eine Kreuzvalidierung evaluieren. Dieser Ansatz lässt sich folgendermaßen kodieren:

In[20]:

```
for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # trainiere einen SVC für jede
        # Parameterkombination
        svm = SVC(gamma=gamma, C=C)
        # führe eine Kreuzvalidierung durch
        scores = cross_val_score(svm, X_trainval, y_trainval, cv=5)
        # berechne die mittlere Genauigkeit der Kreuzvalidierung
        score = np.mean(scores)
        # wenn sich der Score verbessert, speichere Score und Parameter
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}
# erstelle ein neues Modell auf den kombinierten
# Trainings- und Validierungsdaten
svm = SVC(**best_parameters)
svm.fit(X_trainval, y_trainval)
```

Um die Genauigkeit der SVM mit optimierten Werten für C und gamma über fünffache Kreuzvalidierung zu bestimmen, müssen wir  $36 * 5 = 180$  Modelle trainieren. Wie Sie sich vorstellen können, ist der Hauptnachteil der Kreuzvalidierung die zum Bauen all dieser Modelle nötige Rechenzeit.

Die folgende Darstellung (Abbildung 5-6) verdeutlicht, wie die besten Parameter im obigen Code ausgewählt werden:

In[21]:

```
mglearn.plots.plot_cross_val_selection()
```

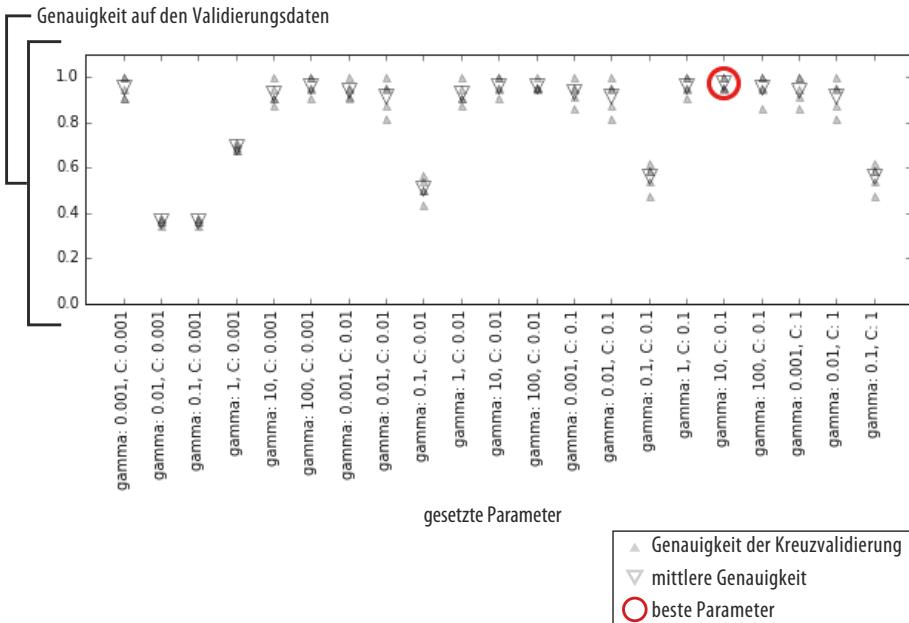


Abbildung 5-6: Ergebnisse der Gittersuche mit Kreuzvalidierung

Für jede Parametereinstellung (nur eine Teilmenge ist gezeigt) werden fünf Genauigkeiten berechnet, eine für jede Teilung während der Kreuzvalidierung. Anschließend wird die mittlere Genauigkeit für jede Parametereinstellung berechnet. Die Parameter mit der höchsten durchschnittlichen Genauigkeit werden ausgewählt und sind durch einen Kreis markiert.



Wie weiter oben gesagt, ist die Kreuzvalidierung eine Möglichkeit, einen bestimmten Algorithmus auf einem bestimmten Datensatz auszuwerten. Allerdings wird sie häufig gemeinsam mit Methoden zur Parametersuche wie der Gittersuche eingesetzt. Aus diesem Grund verwenden viele Leute den Begriff *Kreuzvalidierung* umgangssprachlich, wenn sie die Gittersuche mit Kreuzvalidierung meinen.

Der gesamte Prozess vom Aufteilen der Daten über die Gittersuche bis zum Evaluieren der endgültigen Parameter ist in Abbildung 5-7 dargestellt:

In[22]:

```
mglearn.plots.plot_grid_search_overview()
```

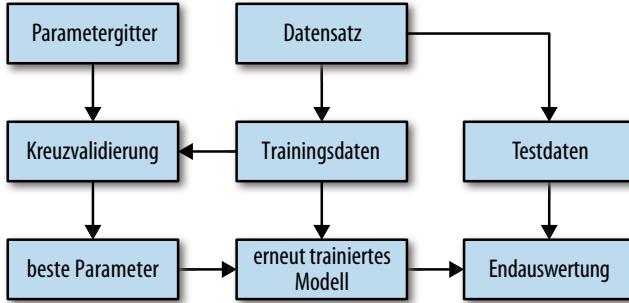


Abbildung 5-7: Überblick über die Parametersuche und Modellauswahl mit GridSearchCV

Weil die Gittersuche mit Kreuzvalidierung eine derart häufige Methode zur Anpassung von Parametern ist, gibt es in scikit-learn die Klasse `GridSearchCV`, die sie in Form eines Estimators implementiert. Um die Klasse `GridSearchCV` zu verwenden, müssen Sie zuerst die zu durchsuchenden Parameter als Dictionary angeben. `GridSearchCV` führt dann die notwendigen Anpassungen des Modells durch. Die Schlüssel des Dictionaries sind dabei die Namen der anzupassenden Parameter (wie beim Konstruieren des Modells – in diesem Fall, `C` und `gamma`), und die Werte sind die auszuprobierenden Werte für diese Parameter. Mit den Werten 0.001, 0.01, 0.1, 1, 10 und 100 für `C` und `gamma` erhalten wir das folgende Dictionary:

**In[23]:**

```
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
              'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
print("Parametergitter:\n{}".format(param_grid))
```

**Out[23]:**

```
Parametergitter:
{'C': [0.001, 0.01, 0.1, 1, 10, 100], 'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
```

Nun können wir eine Instanz der Klasse `GridSearchCV` bilden, wobei wir als Argumente das Modell (`SVC`), das zu durchsuchende Parametergitter (`param_grid`) und die zu verwendende Strategie zur Kreuzvalidierung (z. B. fünffache stratifizierte Kreuzvalidierung) angeben:

**In[24]:**

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
```

`GridSearchCV` verwendet die Kreuzvalidierung anstelle der zuvor verwendeten Teilung in Trainings- und Validierungsdaten. Allerdings müssen wir noch immer die Daten in einen Trainings- und Testdatensatz unterteilen, um ein Overfitting der Parameter zu vermeiden:

**In[25]:**

```
X_train, X_test, y_train, y_test = train_test_split(  
    iris.data, iris.target, random_state=0)
```

Das soeben erstellte Objekt `grid_search` verhält sich wie ein Klassifikator; wir rufen wie gewohnt die Methoden `fit`, `predict` und `score` auf.<sup>1</sup> Wenn wir allerdings `fit` aufrufen, wird die Kreuzvalidierung für jede in `param_grid` angegebene Kombination von Parametern durchgeführt:

**In[26]:**

```
grid_search.fit(X_train, y_train)
```

Anpassen des `GridSearchCV`-Objekts sucht nicht nur nach den besten Parametern, sondern erstellt auch automatisch ein neues, mit dem gesamten Trainingsdatensatz trainiertes Modell, wobei die Parameter mit dem besten Score in der Kreuzvalidierung verwendet werden. Damit ist die Funktionsweise von `fit` zum Code in In[21] zu Beginn dieses Abschnitts äquivalent. Die Klasse `GridSearchCV` gibt uns eine komfortable Schnittstelle, über die wir das neu trainierte Modell über die Methoden `predict` und `score` ansprechen können. Um zu evaluieren, wie gut die besten gefundenen Parameter verallgemeinern, rufen wir `score` mit den Testdaten auf:

**In[27]:**

```
print("Genauigkeit auf den Testdaten: {:.2f}".format(grid_search.score(X_test, y_test)))
```

**Out[27]:**

```
Genauigkeit auf den Testdaten: 0.97
```

Mit der Parameterauswahl über Kreuzvalidierung haben wir ein Modell entdeckt, das eine Genauigkeit von 97 % auf den Testdaten liefert. Dabei ist es wichtig, dass wir die Testdaten nicht verwendet haben, um die Parameter auszuwählen. Die gefundenen Parameter sind im Attribut `best_params_` gespeichert, die beste Genauigkeit aus der Kreuzvalidierung in `best_score_` (also der Mittelwert der Genauigkeiten aller Teilungen für diese Parameter):

**In[28]:**

```
print("Beste Parameter: {}".format(grid_search.best_params_))  
print("Beste Genauigkeit aus der Kreuzvalidierung: {:.2f}".format(grid_search.  
    best_score_))
```

**Out[28]:**

```
Beste Parameter: {'C': 100, 'gamma': 0.01}  
Beste Genauigkeit aus der Kreuzvalidierung: 0.97
```

---

<sup>1</sup> Einen scikit-learn-Estimator, der mit einem anderen Estimator erstellt wird, nennt man einen *Meta-Estimator*. `GridSearchCV` ist der am häufigsten verwendete Meta-Estimator, aber wir werden später noch weitere kennenlernen.



Denken Sie auch diesmal daran, `best_score_` nicht mit der durch `score` auf den Testdaten berechneten Verallgemeinerungsleistung des Modells zu verwechseln. Mit der Methode `score` (oder durch Auswerten der Ausgabe der Methode `predict`) verwenden Sie ein Modell, das *auf den gesamten Trainingsdaten* trainiert wurde. Das Attribut `best_score_` enthält die mittlere Genauigkeit der Kreuzvalidierung, wobei die *Kreuzvalidierung auf den Trainingsdaten* durchgeführt wurde.

Manchmal ist es hilfreich, Zugriff auf das eigentliche gefundene Modell zu erhalten – beispielsweise um Koeffizienten oder Wichtigkeiten von Merkmalen zu untersuchen. Sie können auf das mit den besten Parametern auf den gesamten Trainingsdaten trainierte Modell über das Attribut `best_estimator_` zugreifen:

**In[29]:**

```
print("Bester Estimator:\n{}".format(grid_search.best_estimator_))
```

**Out[29]:**

```
Bester Estimator:  
SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,  
     decision_function_shape=None, degree=3, gamma=0.01, kernel='rbf',  
     max_iter=-1, probability=False, random_state=None, shrinking=True,  
     tol=0.001, verbose=False)
```

Weil `grid_search` selbst die Methoden `predict` und `score` besitzt, muss man nicht auf `best_estimator_` zugreifen, um Vorhersagen zu treffen oder das Modell zu evaluieren.

## Analyse der Ergebnisse einer Kreuzvalidierung

Es ist oft hilfreich, die Ergebnisse einer Kreuzvalidierung zu visualisieren, um die Abhängigkeit der Verallgemeinerung durch das Modells von den durchsuchten Parametern nachzuvollziehen. Da Gittersuchen eher rechenintensiv sind, ist es oft besser, mit einem groben, kleinen Gitter zu beginnen. Wir können dann die Ergebnisse einer Gittersuche mit Kreuzvalidierung inspizieren und den Suchraum gegebenenfalls erweitern. Die Ergebnisse der Gittersuche lassen sich im Attribut `cv_results_` finden, das ein Dictionary mit sämtlichen Aspekten der Suche ist. Es enthält eine Menge Details, wie Sie in der folgenden Ausgabe sehen. Diese lassen sich am besten nach Umwandlung in ein pandas DataFrame betrachten:

**In[30]:**

```
import pandas as pd  
# Umwandlung in ein DataFrame  
results = pd.DataFrame(grid_search.cv_results_)  
# zeige die ersten 5 Zeilen an  
display(results.head())
```

**Out[30]:**

	param_C	param_gamma	params	mean_test_score
0	0.001	0.001	{'C': 0.001, 'gamma': 0.001}	0.366
1	0.001	0.01	{'C': 0.001, 'gamma': 0.01}	0.366
2	0.001	0.1	{'C': 0.001, 'gamma': 0.1}	0.366
3	0.001	1	{'C': 0.001, 'gamma': 1}	0.366
4	0.001	10	{'C': 0.001, 'gamma': 10}	0.366
	rank_test_score	split0_test_score	split1_test_score	split2_test_score
0	22	0.375	0.347	0.363
1	22	0.375	0.347	0.363
2	22	0.375	0.347	0.363
3	22	0.375	0.347	0.363
4	22	0.375	0.347	0.363
	split3_test_score	split4_test_score	std_test_score	
0	0.363	0.380	0.011	
1	0.363	0.380	0.011	
2	0.363	0.380	0.011	
3	0.363	0.380	0.011	
4	0.363	0.380	0.011	

Jede Zeile in results entspricht einer bestimmten Parametereinstellung. Die Ergebnisse aller Teilungen bei der Kreuzvalidierung werden für jede Einstellung aufgezeichnet, ebenso Mittelwert und Standardabweichung über alle Teilungen. Während wir das zweidimensionale Gitter der Parameter C und gamma durchsuchen, lassen sich diese am besten als Heatmap darstellen (Abbildung 5-9). Wir extrahieren zunächst die mittleren Scores der Validierung, anschließend wandeln wir die Scores mit reshape so um, dass die Achsen C und gamma entsprechen:

**In[31]:**

```
scores = np.array(results.mean_test_score).reshape(6, 6)

# Zeichne den mittleren Score der Kreuzvalidierung
mglearn.tools.heatmap(scores, xlabel='gamma', xticklabels=param_grid['gamma'],
                      ylabel='C', yticklabels=param_grid['C'], cmap="viridis")
```

Jeder Punkt in der Heatmap entspricht einer Kreuzvalidierung mit bestimmten Parametereinstellungen. Die Farbe zeigt die Genauigkeit bei der Kreuzvalidierung an, wobei helle Farben auf eine hohe und dunkle Farben auf eine geringe Genauigkeit hindeuten. Sie sehen, dass der SVC sehr anfällig auf die Parameter reagiert. Für viele Parametereinstellungen liegt die Genauigkeit bei etwa 40 %, was recht schlecht ist; bei anderen Einstellungen liegt die Genauigkeit bei ungefähr 96 %. Wir können aus der Analyse dieses Diagramms mehrere Dinge lernen. Beispielsweise ist das Einstellen der Parameter für eine hohe Genauigkeit *sehr wichtig*. Beide Parameter (C und gamma) haben einen hohen Einfluss, da Änderungen die Genauigkeit von 40 % auf 96 % verändern können. Außerdem ist der Wertebereich der Parameter so gewählt, dass wir deutliche Änderungen des Ergebnisses sehen. Es ist auch zu betonen, dass die Spannbreite beider Parameter groß genug ist: Die optimalen Werte beider Parameter liegen nicht am Rande des Diagramms.

Betrachten wir nun einige Diagramme (in Abbildung 5-8), bei denen das Ergebnis aufgrund schlecht gewählter Suchgrenzen weniger günstig ist:

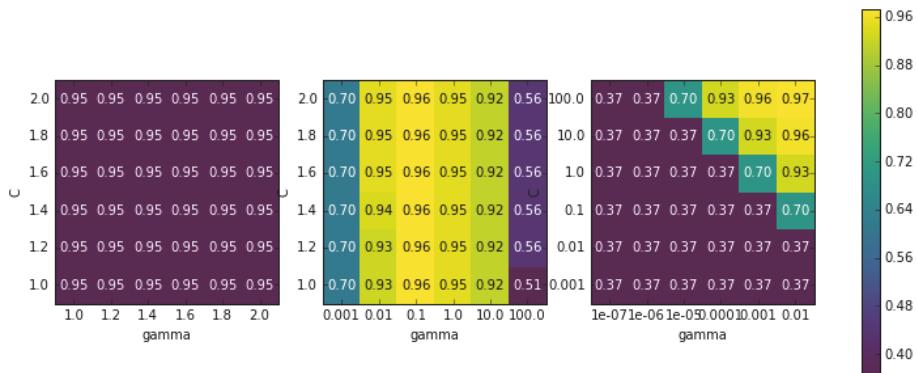


Abbildung 5-8: Darstellung schlecht eingestellter Suchgitter als Heatmap

In[32]:

```
fig, axes = plt.subplots(1, 3, figsize=(13, 5))

param_grid_linear = {'C': np.linspace(1, 2, 6),
                     'gamma': np.linspace(1, 2, 6)}

param_grid_one_log = {'C': np.linspace(1, 2, 6),
                      'gamma': np.logspace(-3, 2, 6)}

param_grid_range = {'C': np.logspace(-3, 2, 6),
                    'gamma': np.logspace(-7, -2, 6)}

for param_grid, ax in zip([param_grid_linear, param_grid_one_log,
                           param_grid_range], axes):
    grid_search = GridSearchCV(SVC(), param_grid, cv=5)
    grid_search.fit(X_train, y_train)
    scores = grid_search.cv_results_['mean_test_score'].reshape(6, 6)

    # Plotte die mittleren Scores der Kreuzvalidierung
    scores_image = mglearn.tools.heatmap(
        scores, xlabel='gamma', ylabel='C', xticklabels=param_grid['gamma'],
        yticklabels=param_grid['C'], cmap="viridis", ax=ax)

plt.colorbar(scores_image, ax=axes.tolist())
```

Das erste Diagramm zeigt keinerlei Änderungen. Das gesamte Parametergitter ist mit der gleichen Farbe gefüllt. In diesem Falle liegt das an einer ungeeigneten Skalierung und dem Wertebereich der Parameter C und gamma. Wenn wir bei unterschiedlichen Parametereinstellungen keine Änderung der Genauigkeit beobachten, kann dies aber auch daran liegen, dass der Parameter überhaupt nicht wichtig ist. Normalerweise sollten Sie zuerst extreme Werte auszuprobieren, um eventuelle Änderungen der Genauigkeit als Folge der Parameter zu beobachten.

Das zweite Diagramm zeigt ein vertikales Streifenmuster. Dies deutet darauf hin, dass nur der Parameter gamma einen Einfluss hat. Das könnte bedeuten, dass wir

interessante Werte für  $\gamma$  durchsuchen, für  $C$  aber nicht. Es könnte aber auch bedeuten, dass der Parameter  $C$  nicht wichtig ist.

Das dritte Diagramm zeigt Änderungen sowohl bei  $C$  als auch bei  $\gamma$ . Allerdings sehen wir, dass im gesamten unteren linken Bereich des Diagramms nichts Interessantes passiert. Wir können vermutlich bei zukünftigen Gittersuchen sehr kleine Werte ausschließen. Der optimale Parameter befindet sich oben rechts. Da das Optimum am Rand des Diagramms zu finden ist, ist zu erwarten, dass es jenseits dieses Randes noch bessere Werte gibt. Wir sollten daher den Suchbereich erweitern, um weitere Parametereinstellungen aus diesem Bereich aufzunehmen.

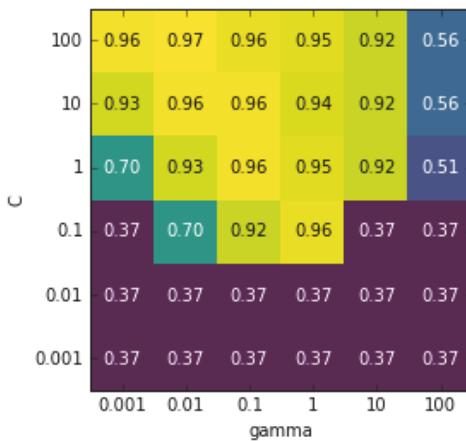


Abbildung 5-9: Heatmap der mittleren Genauigkeit bei der Kreuzvalidierung in Abhängigkeit von  $C$  und  $\gamma$

Das Optimieren des Parametergitters auf Grundlage des Scores aus der Kreuzvalidierung ist eine gute Möglichkeit, die Wichtigkeit unterschiedlicher Parameter zu untersuchen. Sie sollten aber unterschiedliche Parametereinstellungen nicht mit dem Testdatensatz auswerten – wie bereits erwähnt, sollte die Evaluation auf den Testdaten nur einmal stattfinden, nämlich sobald wir genau wissen, welches Modell wir verwenden möchten.

### Suche über Räume, die keine Gitter sind

In einigen Fällen ist das Ausprobieren aller möglichen Parameterkombinationen wie bei GridSearchCV keine gute Idee. Beispielsweise enthält SVC den Parameter `kernel`, und je nach gewähltem Kernel sind andere Parameter relevant. Bei `kernel='linear'` ist das Modell linear, und lediglich der Parameter `C` wird verwendet. Bei `kernel='rbf'` werden `C` und  $\gamma$  als Parameter verwendet (aber andere Parameter wie `degree` nicht). In diesem Fall würde die Suche über mögliche Werte für  $C$ ,  $\gamma$  und `kernel` keinen Sinn ergeben: Bei `kernel='linear'` wird  $\gamma$  nicht verwendet, und unterschiedliche Werte für  $\gamma$  wären reine Zeitverschwendungen. Um mit dieser Art »bedingter« Parameter umzugehen, lässt GridSearchCV für param\_

grid auch eine *Liste von Dictionaries* zu. Jedes Dictionary in der Liste wird als unabhängiges Gitter behandelt. Eine mögliche Gittersuche mit Kernel und den entsprechenden Parametern könnte folgendermaßen aussehen:

**In[33]:**

```
param_grid = [ {'kernel': ['rbf'],
   'C': [0.001, 0.01, 0.1, 1, 10, 100],
   'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}, 
  {'kernel': ['linear'],
   'C': [0.001, 0.01, 0.1, 1, 10, 100]}]
print("Liste von Gittern:\n{}".format(param_grid))
```

**Out[33]:**

```
Liste von Gittern:
[{'kernel': ['rbf'], 'C': [0.001, 0.01, 0.1, 1, 10, 100],
 'gamma': [0.001, 0.01, 0.1, 1, 10, 100]},
 {'kernel': ['linear'], 'C': [0.001, 0.01, 0.1, 1, 10, 100]}]
```

Im ersten Gitter ist der Parameter kernel stets auf 'rbf' gesetzt (beachten Sie, dass der Eintrag für kernel eine Liste der Länge 1 ist), und sowohl C als auch gamma werden variiert. Im zweiten Gitter ist der Parameter kernel stets auf linear gesetzt, und nur C wird verändert. Führen wir diese etwas komplexere Parametersuche einmal durch:

**In[34]:**

```
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
grid_search.fit(X_train, y_train)
print("Beste Parameter: {}".format(grid_search.best_params_))
print("Bester Score aus der Kreuzvalidierung: {:.2f}".format(grid_search.best_score_))
```

**Out[34]:**

```
Beste Parameter: {'C': 100, 'kernel': 'rbf', 'gamma': 0.01}
Bester Score aus der Kreuzvalidierung: 0.97
```

Betrachten wir noch einmal cv\_results\_. Wie erwartet, wird bei einem linearen kernel nur C verändert:

**In[35]:**

```
results = pd.DataFrame(grid_search.cv_results_)
# wir zeigen die Tabelle transponiert an, damit sie besser auf die Seite passt:
display(results.T)
```

**Out[35]:**

```
12 rows × 42 columns
```

	0	1	2	3	...	38	39	40	41
param_C	0.001	0.001	0.001	0.001	...	0.1	1	10	100
param_gamma	0.001	0.01	0.1	1	...	NaN	NaN	NaN	NaN
param_kernel	rbf	rbf	rbf	rbf	...	linear	linear	linear	linear

	0	1	2	3	...	38	39	40	41
params	{C: 0.001, kernel: rbf, gamma: 0.001}	{C: 0.001, kernel: rbf, gamma: 0.01}	{C: 0.001, kernel: rbf, gamma: 0.1}	{C: 0.001, kernel: rbf, gamma: 1}	...	{C: 0.1, kernel: linear}	{C: 1, kernel: linear}	{C: 10, kernel: linear}	{C: 100, kernel: linear}
mean_test_score	0.37	0.37	0.37	0.37	...	0.95	0.97	0.96	0.96
rank_test_score	27	27	27	27	...	11	1	3	3
split0_test_score	0.38	0.38	0.38	0.38	...	0.96	1	0.96	0.96
split1_test_score	0.35	0.35	0.35	0.35	...	0.91	0.96	1	1
split2_test_score	0.36	0.36	0.36	0.36	...	1	1	1	1
split3_test_score	0.36	0.36	0.36	0.36	...	0.91	0.95	0.91	0.91
split4_test_score	0.38	0.38	0.38	0.38	...	0.95	0.95	0.95	0.95
std_test_score	0.011	0.011	0.011	0.011	...	0.033	0.022	0.034	0.034

## Unterschiedliche Strategien zur Kreuzvalidierung mit Gittersuche

Ähnlich zu `cross_val_score` verwendet auch `GridSearchCV` standardmäßig die stratifizierte  $k$ -fache Kreuzvalidierung zur Klassifizierung und die  $k$ -fache Kreuzvalidierung zur Regression. Sie können aber über den Parameter `cv` auch einen beliebigen Splitter zur Kreuzvalidierung an `GridSearchCV` übergeben, wie unter Abschnitt »Zusätzliche Kontrolle über die Kreuzvalidierung« auf Seite 240 beschrieben. Insbesondere können Sie `ShuffleSplit` oder `StratifiedShuffleSplit` mit `n_iter=1` einsetzen, um nur eine Teilung in Trainings- und Testdaten durchzuführen. Dies kann bei sehr großen Datensätzen oder sehr langsamem Modellen helfen.

## Verschachtelte Kreuzvalidierung

In den vorigen Beispielen sind wir von einer einzelnen Teilung der Daten in Trainings-, Validations- und Testdatensätze ausgegangen. Anschließend haben wir die Daten in Trainingsdaten und Testdaten aufgeteilt und auf den Trainingsdaten eine Kreuzvalidierung durchgeführt. Bei der oben beschriebenen Verwendung von `GridSearchCV` haben wir ebenfalls eine einzelne Teilung in Trainings- und Testdaten vorgenommen, wodurch unsere Ergebnisse instabil und stark von dieser einen Teilung abhängig werden. Wir können einen Schritt weiter gehen und statt einer einzelnen Teilung der ursprünglichen Daten in Trainings- und Testdatensätze mehrere Teilungen zur Kreuzvalidierung verwenden. Damit erhalten wir eine sogenannte *verschachtelte Kreuzvalidierung*. Bei der verschachtelten Kreuzvalidierung gibt es eine äußere Schleife, die die Daten in Trainings- und Testdatensätze aufteilt. Für jeden Datensatz wird eine Gittersuche durchgeführt (wobei sich für jede Teilung in der äußeren Schleife andere beste Parameter ergeben können). Dann wird für jede Teilung der äußeren Schleife der Score mit den besten Parametereinstellungen ausgegeben.

Das Ergebnis dieses Vorgangs ist eine Liste von Genauigkeiten – kein Modell und keine Parameter. Die Genauigkeiten geben Aufschluss darüber, wie gut das Modell

mit den besten im Gitter gefundenen Parametern verallgemeinern kann. Da wir kein auf neue Daten anwendbares Modell erhalten, wird die verschachtelte Kreuzvalidierung selten bei der Suche nach einem für zukünftige Daten anwendbaren Vorhersagemodell eingesetzt. Sie ist aber nützlich, um auszuwerten, wie gut ein vorhandenes Modell auf einem bestimmten Datensatz abschneidet.

Die verschachtelte Kreuzvalidierung lässt sich in scikit-learn einfach umsetzen. Wir rufen `cross_val_score` mit einer Instanz von `GridSearchCV` als Modell auf:

**In[36]:**

```
scores = cross_val_score(GridSearchCV(SVC(), param_grid, cv=5),
                         iris.data, iris.target, cv=5)
print("Scores aus der Kreuzvalidierung: ", scores)
print("mittlerer Score aus der Kreuzvalidierung: ", scores.mean())
```

**Out[36]:**

```
Scores aus der Kreuzvalidierung: [ 0.967  1.      0.967  0.967  1.      ]
mittlerer Score aus der Kreuzvalidierung:  0.98
```

Das Ergebnis unserer verschachtelten Kreuzvalidierung lässt sich wie folgt zusammenfassen: »SVC ist in der Lage, auf dem Datensatz `iris` eine mittlere Genauigkeit von 98 % zu erzielen« – nicht mehr und nicht weniger.

Wir haben hier sowohl in der inneren als auch der äußeren Schleife die stratifizierte fünffache Kreuzvalidierung verwendet. Da `param_grid` 36 Parameterkombinationen enthält, erhalten wir die stolze Zahl von  $36 * 5 * 5 = 900$  zu konstruierenden Modellen, wodurch die verschachtelte Kreuzvalidierung ein sehr rechenintensiver Vorgang ist. Wir haben hier in der inneren und äußeren Schleife den gleichen Splitter zur Kreuzvalidierung verwendet. Dies ist aber nicht zwingend; Sie können beliebige Strategien zur Kreuzvalidierung miteinander kombinieren. Es kann ein wenig schwierig sein, die einzelne obige Programmzeile nachzuvollziehen. In der folgenden, vereinfachten Implementierung ist der Vorgang zur Veranschaulichung durch `for`-Schleifen dargestellt:

**In[37]:**

```
def nested_cv(X, y, inner_cv, outer_cv, Classifier, parameter_grid):
    outer_scores = []
    # für jede Teilung in der äußeren Kreuzvalidierung
    # (die Methode split liefert Indices)
    for training_samples, test_samples in outer_cv.split(X, y):
        # finde die besten Parameter durch innere Kreuzvalidierung
        best_params = {}
        best_score = -np.inf
        # iteriere über die Parameter
        for parameters in parameter_grid:
            # sammle die Scores für die inneren Teilungen
            cv_scores = []
            # iteriere über die innere Kreuzvalidierung
            for inner_train, inner_test in inner_cv.split(
                X[training_samples], y[training_samples]):

```

```

# erstelle einen Klassifikator mit Parametern
# und Trainingsdaten
clf = Classifier(**parameters)
clf.fit(X[inner_train], y[inner_train])
# evaluiere mit dem inneren Testdatensatz
score = clf.score(X[inner_test], y[inner_test])
cv_scores.append(score)

# berechne den mittleren Score über die inneren Folds
mean_score = np.mean(cv_scores)
if mean_score > best_score:
    # speichere die Parameter bei Verbesserung
    best_score = mean_score
    best_params = parameters

# erstelle einen Klassifikator mit den besten Parametern
# und den äußeren Trainingsdaten
clf = Classifier(**best_params)
clf.fit(X[training_samples], y[training_samples])
# evaluiere
outer_scores.append(clf.score(X[test_samples], y[test_samples]))
return np.array(outer_scores)

```

Wenden wir diese Funktion nun auf den Datensatz iris an:

**In[38]:**

```

from sklearn.model_selection import ParameterGrid, StratifiedKFold
scores = nested_cv(iris.data, iris.target, StratifiedKFold(5),
                    StratifiedKFold(5), SVC, ParameterGrid(param_grid))
print("Scores der Kreuzvalidierung: {}".format(scores))

```

**Out[38]:**

```
Scores der Kreuzvalidierung: [ 0.967  1.      0.967  0.967  1.      ]
```

## Parallelisieren von Kreuzvalidierung und Gittersuche

Das Ausführen einer Gittersuche über viele Parameter und große Datensätze ist zwar sehr rechenintensiv, es ist aber auch *verblüffend parallel*. Das bedeutet, dass sich der Modellbau mit bestimmten Parametern für eine bestimmte Teilung einer Kreuzvalidierung völlig unabhängig von den übrigen Parametereinstellungen und Modellen durchführen lässt. Damit eignen sich Gittersuche und Kreuzvalidierung ideal zum Parallelisieren mit mehreren CPU-Cores oder einem Rechnercluster. Sie können bei GridSearchCV und cross\_val\_score mehrere Cores verwenden, indem Sie über den Parameter n\_jobs die Anzahl zu verwendender CPU-Cores angeben. Sie können auch n\_jobs=-1 festlegen, um sämtliche verfügbaren Cores zu nutzen.

Ihnen sollte klar sein, dass Sie in scikit-learn *keine parallelen Operationen verschachteln dürfen*. Wenn Sie also die Option n\_jobs bereits in Ihrem Modell nutzen (z. B. einem Random Forest), können Sie diese nicht in GridSearchCV einsetzen, um über dieses Modell zu suchen. Wenn Ihr Datensatz und Ihr Modell sehr groß sind, kann es sein, dass die Nutzung mehrerer Cores zu viel Speicher verbraucht. Daher sollten Sie beim parallelen Aufbau großer Modelle die Speichernutzung im Auge behalten.

Es ist auch möglich, die Gittersuche und Kreuzvalidierung über mehrere Rechner in einem Cluster zu verteilen, auch wenn dies gegenwärtig von scikit-learn nicht unterstützt wird. Sie können dafür das in IPython vorhandene parallele Framework für parallele Gittersuchen einzusetzen, sofern es Ihnen nichts ausmacht, sich dazu eine for-Schleife über die Parameter zu schreiben, wie in Abschnitt »Einfache Gittersuche« auf Seite 245 beschrieben.

Für Nutzer von Spark gibt es das vor Kurzem entwickelte Paket spark-sklearn (<https://github.com/databricks/spark-sklearn>), mit dem wir eine Gittersuche auf einem bereits vorhandenen Spark-Cluster ausführen können.

## Evaluationsmetriken

Bisher haben wir die Qualität einer Klassifikation über die Genauigkeit ausgewertet (den Anteil korrekt klassifizierter Datenpunkte) und die Qualität einer Regression über den  $R^2$ -Wert. Dies sind jedoch nur zwei von vielen Möglichkeiten, die Vorhersagegüte eines überwachten Modells für einen bestimmten Datensatz zusammenzufassen. In der Praxis sind diese Maße für Ihren Datensatz nicht unbedingt geeignet, und die Auswahl der richtigen Metrik ist für die Auswahl eines Modells und die Einstellung von Parametern wichtig.

### Das Ziel im Auge behalten

Bei der Auswahl einer Metrik sollten Sie stets das Ziel der maschinellen Lernanwendung im Auge behalten. In der Praxis interessieren wir uns für gewöhnlich nicht einfach nur für gute Vorhersagen, sondern möchten diese Vorhersagen als Teil eines größeren Entscheidungsprozesses einsetzen. Bevor Sie eine Metrik zum maschinellen Lernen auswählen, sollten Sie über das übergeordnete Ziel Ihrer Anwendung nachdenken, das oft als *Business-Metrik* bezeichnet wird. Die Folgen der Auswahl eines bestimmten Algorithmus zum maschinellen Lernen nennt man auch *Business-Impact*.<sup>2</sup> Das übergeordnete Ziel könnte sein, Verkehrsunfälle zu vermeiden oder die Anzahl Einweisungen in eine Klinik zu verringern. Es könnte auch sein, mehr Nutzer auf Ihre Webseite zu führen oder Nutzer zu veranlassen, in Ihrem Geschäft mehr Geld auszugeben. Bei der Auswahl eines Modells oder beim Einstellen von Parametern sollten Sie sich für das Modell oder die Parametereinstellungen entscheiden, die den stärksten positiven Einfluss auf die Business-Metrik haben. Häufig ist dies schwierig, da es zum Bewerten des Business-Impact eines bestimmten Modells nötig sein kann, das Modell in einem Produktivsystem unterzubringen.

---

<sup>2</sup> Wir bitten die wissenschaftlich gesinnten Leser die Geschäftssprache in diesem Abschnitt zu entschuldigen. Das eigentliche Ziel im Auge zu behalten, ist in der Wissenschaft gleichermaßen wichtig, auch wenn den Autoren kein zu »Business-Impact« äquivalenter Ausdruck aus diesem Bereich bekannt ist.

In frühen Entwicklungsstadien und zum Einstellen von Parametern ist es oft nicht praktikabel, ein Modell einfach nur zum Testen in einem Produktivsystem zu platzieren, da dies mit hohen Geschäfts- oder persönlichen Risiken verbunden ist. Stellen Sie sich vor, Sie würden bei einem selbstfahrenden Auto die Fähigkeit, Fußgängern auszuweichen, testen, indem sie das Auto herumfahren lassen, ohne es zuvor zu verifizieren; falls Ihr Modell sich als schlecht herausstellt, sind die Fußgänger in Gefahr! Daher benötigen wir häufig ersatzweise eine Prozedur zur Auswertung, etwa eine leichter berechenbare Metrik. Wir könnten beispielsweise Bilder von Fußgängern mit Nicht-Fußgängern vergleichen und die Genauigkeit messen. Bedenken Sie, dass es sich dabei nur um einen Ersatz handelt, und es lohnt sich, eine dem ursprünglichen Geschäftsziel nahe und auswertbare Metrik zu finden. Diese Metrik sollte zur Evaluierung und Auswahl von Modellen so oft wie möglich eingesetzt werden. Das Ergebnis dieser Evaluierung ist nicht unbedingt eine einzelne Zahl – die Folge Ihres Algorithmus könnte sein, dass Sie 10 % mehr Kunden haben, aber jeder Kunde 15 % weniger ausgibt. Das Ergebnis sollte aber die zu erwartenden Auswirkungen der Entscheidung für das eine oder das andere Modell erfassen.

In diesem Abschnitt werden wir zunächst Metriken für den wichtigen Sonderfall der binären Klassifikation besprechen, uns anschließend der Klassifikation mehrerer Kategorien und schließlich der Regression zuwenden.

## **Metriken zur binären Klassifikation**

Die binäre Klassifikation ist vermutlich die häufigste und konzeptionell einfachste praktische Anwendung maschinellen Lernens. Aber sogar bei der Auswertung dieser einfachen Aufgabe gilt es einige Dinge zu beachten. Bevor wir uns mit den verschiedenen Metriken beschäftigen, schauen wir uns an, auf welche Weise das Bestimmen der Genauigkeit irreführend sein kann. Bei der binären Klassifikation sprechen wir oft von einer *positiven* und einer *negativen* Kategorie, wobei die positive Kategorie diejenige ist, nach der wir suchen.

### **Arten von Fehlern**

Die Genauigkeit ist of kein gutes Maß für die Vorhersageleistung, da die Anzahl der begangenen Fehler nicht alle für uns interessanten Informationen enthält. Stellen Sie sich eine Anwendung zur automatischen Früherkennung von Krebs vor. Falls der Test negativ ist, nimmt man an, dass der Patient gesund ist, ist der Test dagegen positiv, wird der Patient weiteren Untersuchungen unterzogen. In diesem Beispiel nennen wir einen positiven Test (als Indikation für Krebs) die positive Kategorie und einen negativen Test die negative Kategorie. Wir können nicht davon ausgehen, dass unser Modell immer perfekt funktioniert, es wird also Fehler machen. Bei jeder Anwendung müssen wir uns daher fragen, was die Konsequenzen dieser Fehler im wirklichen Leben bedeuten.

Ein möglicher Fehler ist, dass ein gesunder Patient als positiv klassifiziert wird, was weitere Untersuchungen nach sich zieht. Dadurch entstehen einige Kosten und Unannehmlichkeiten für den Patienten (und vermutlich eine Menge Beunruhigung). Eine inkorrekte positive Vorhersage nennt man *falsch positiv*. Die andere Möglichkeit ist, dass ein kranker Patient als negativ klassifiziert wird und daher keine weiteren Untersuchungen oder Behandlung erfährt. Der nicht diagnostizierte Krebs kann zu ernsten gesundheitlichen Problemen führen oder sogar tödlich sein. Einen Fehler dieser Art – eine inkorrekte negative Vorhersage – nennt man *falsch negativ*. In der Statistik bezeichnet man falsch positive Ergebnisse auch als *Fehler vom Typ I* und falsch negative auch als *Fehler vom Typ II*. Wir werden hier »falsch negativ« und »falsch positiv« verwenden, da sie klarer und leichter zu merken sind. Im Beispiel der Krebsdiagnose ist klar, dass wir so viele falsch negative Tests wie möglich vermeiden möchten, während die falsch positiven eher ein kleines Ärgernis sind.

Auch wenn dies ein besonders drastisches Beispiel ist, sind die Folgen von falsch positiven und falsch negativen Vorhersagen selten gleich. In kommerziellen Anwendungen könnte es möglich sein, beiden Arten von Fehlern einen monetären Wert zuzuweisen, wodurch man den Fehler einer bestimmten Vorhersage in Euro anstatt als Genauigkeit bestimmen könnte. Für Geschäftsentscheidungen kann es deutlich sinnvoller sein, diese Art von Metrik zur Modellauswahl zu verwenden.

## Nicht balancierte Datensätze

Die Art des Fehlers spielt eine wichtige Rolle, wenn eine der beiden Kategorien viel häufiger als die andere ist. Dies kommt in der Praxis sehr oft vor; ein gutes Beispiel ist die Vorhersage des Anklickens, wobei jeder Datenpunkt für einen »Eindruck«, einen dem Nutzer gezeigten Inhalt, steht. Dieser Inhalt könnte eine Werbeanzeige, ein verwandter Artikel oder eine ähnliche Person zum Folgen in einem sozialen Netzwerk sein. Das Ziel ist, vorherzusagen, ob ein Nutzer auf einen bestimmten gezeigten Inhalt klickt (wodurch dieser Interesse bekundet). Die meisten Nutzern im Internet vorgeführten Dinge (insbesondere Werbung) führen nicht zu Klicks. Sie müssen einem Nutzer unter Umständen 100 Werbebanner oder Artikel vorführen, bevor dieser etwas ausreichend Interessantes findet und draufklickt. Damit erhalten Sie einen Datensatz, bei dem es für je 99 Punkte »kein Klick« einen Datenpunkt »Klick« gibt. Anders gesagt, gehören 99 % der Datenpunkte zur Kategorie »kein Klick«. Datensätze, bei denen eine Kategorie sehr viel häufiger als die andere ist, nennt man auch *nicht balancierte Datensätze* oder *Datensätze mit nicht balancierten Kategorien*. In der Wirklichkeit sind nicht balancierte Daten der Normalfall, die interessanten Ereignisse treten selten gleich oft bzw. nicht einmal ähnlich häufig in den Daten auf.

Wenn Sie nun einen Klassifikator erstellen, der für das Klickbeispiel zu 99 % korrekte Vorhersagen liefert. Was sagt Ihnen das? Eine Genauigkeit von 99 % klingt erst einmal eindrucksvoll, berücksichtigt aber die fehlende Balance zwischen den Kategorien nicht. Sie können auch völlig ohne ein maschinelles Lernmodell 99 % Genauigkeit erreichen, indem Sie nämlich immer »kein Klick« vorhersagen. Ande-

rerseits könnte ein Modell mit 99%iger Genauigkeit auch wirklich gut sein. Leider können wir mit der Genauigkeit das konstante »kein Klick«-Modell nicht von einem möglicherweise guten Modell unterscheiden.

Um diesen Fall zu verdeutlichen, erstellen wir einen im Verhältnis 9:1 nicht balancierten Datensatz aus dem Datensatz digits, indem wir die Ziffer 9 gegen alle übrigen klassifizieren:

**In[39]:**

```
from sklearn.datasets import load_digits

digits = load_digits()
y = digits.target == 9

X_train, X_test, y_train, y_test = train_test_split(
    digits.data, y, random_state=0)
```

Wir können den `DummyClassifier` verwenden, um stets die dominierende Kategorie vorherzusagen (hier »nicht neun«) und zu sehen, wie wenig informativ die Genauigkeit sein kann:

**In[40]:**

```
from sklearn.dummy import DummyClassifier
dummy_majority = DummyClassifier(strategy='most_frequent').fit(X_train, y_train)
pred_most_frequent = dummy_majority.predict(X_test)
print("Eindeutige vorhergesagte Labels: {}".format(np.unique(pred_most_frequent)))
print("Genauigkeit auf den Testdaten: {:.2f}".format(dummy_majority.score(X_test, y_test)))
```

**Out[40]:**

```
Eindeutige vorhergesagte Labels: [False]
Genauigkeit auf den Testdaten: 0.90
```

Wir haben eine Genauigkeit von 90 % erreicht, ohne irgendetwas zu lernen. Dies mag erstaunlich erscheinen, aber denken wir einen Moment darüber nach. Stellen wir uns vor, jemand behauptet, sein Modell sei zu 90 % genau. Sie könnten denken, dass derjenige gute Arbeit geleistet hat, aber je nach Aufgabenstellung hat er vielleicht nur eine Kategorie vorhergesagt! Vergleichen wir dieses Ergebnis mit einem echten Klassifikator:

**In[41]:**

```
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier(max_depth=2).fit(X_train, y_train)
pred_tree = tree.predict(X_test)
print("Genauigkeit auf den Testdaten: {:.2f}".format(tree.score(X_test, y_test)))
```

**Out[41]:**

```
Genauigkeit auf den Testdaten: 0.92
```

Der Genauigkeit nach ist der `DecisionTreeClassifier` nur geringfügig besser als die konstante Vorhersage. Dies könnte darauf hinweisen, dass etwas mit dem von uns

verwendeten DecisionTreeClassifier nicht in Ordnung ist oder dass die Genauigkeit in diesem Fall kein gutes Maß ist.

Zu Vergleichszwecken werten wir noch zwei weitere Klassifikatoren aus, LogisticRegression und den DummyClassifier, der mit den Standardparametern zufällige Vorhersagen trifft, aber Kategorien mit den gleichen Proportionen wie im Trainingsdatensatz vorhersagt:

**In[42]:**

```
from sklearn.linear_model import LogisticRegression

dummy = DummyClassifier().fit(X_train, y_train)
pred_dummy = dummy.predict(X_test)
print("Dummy-Genauigkeit: {:.2f}".format(dummy.score(X_test, y_test)))

logreg = LogisticRegression(C=0.1).fit(X_train, y_train)
pred_logreg = logreg.predict(X_test)
print("Logreg-Genauigkeit: {:.2f}".format(logreg.score(X_test, y_test)))
```

**Out[42]:**

```
Dummy-Genauigkeit: 0.80
Logreg-Genauigkeit: 0.98
```

Der Dummy-Klassifikator mit der zufälligen Ausgabe ist mit Abstand der schlechteste in der Gruppe (der Genauigkeit nach), während LogisticRegression sehr gute Ergebnisse produziert. Allerdings erzielt auch der zufällige Klassifikator eine Genauigkeit von 80 %. Dadurch ist es sehr schwer zu beurteilen, welches dieser Ergebnisse überhaupt aufschlussreich ist. Das Problem liegt auch hier darin, dass bei diesem nicht balancierten Datensatz die Genauigkeit ein inadäquates Maß zum Bestimmen der Vorhersagequalität ist. Im weiteren Verlauf dieses Kapitels werden wir alternative Metriken kennenlernen, mit denen wir eine bessere Entscheidungsgrundlage zur Auswahl von Modellen erhalten. Insbesondere benötigen wir Metriken, die uns verraten, wie viel besser ein Modell gegenüber einer konstanten oder zufälligen Vorhersage ist, wie die in pred\_most\_frequent und pred\_dummy berechneten. Wenn wir unsere Modelle mit einer Metrik bewerten, sollte es auf jeden Fall möglich sein, diese Nonsense-Vorhersagen auszusortieren.

## Konfusionsmatrizen

Konfusionsmatrizen sind eine der übersichtlichsten Möglichkeiten zum Darstellen der Ergebnisse einer binären Klassifikation. Betrachten wir die von LogisticRegression im letzten Abschnitt getroffenen Vorhersagen mit der Funktion confusion\_matrix. Wir haben die Vorhersagen auf den Testdaten bereits in pred\_logreg gespeichert:

**In[43]:**

```
from sklearn.metrics import confusion_matrix
```

```
confusion = confusion_matrix(y_test, pred_logreg)
print("Konfusionsmatrix:\n{}".format(confusion))
```

**Out[43]:**

```
Konfusionsmatrix:
[[401  2]
 [ 8 39]]
```

Die Ausgabe von `confusion_matrix` ist ein Array der Größe  $2 \times 2$ , wobei die Zeilen den tatsächlichen Kategorien und die Spalten den vorhergesagten Kategorien entsprechen. Jeder Eintrag gibt an, wie oft ein der jeweiligen Zeile zugehöriger Datenpunkt (hier »nicht neun« und »neun«) zu einer der Spalte entsprechenden Kategorie zugeordnet wurde. Das folgende Diagramm verdeutlicht diesen Aufbau (Abbildung 5-10):

**In[44]:**

```
mglearn.plots.plot_confusion_matrix_illustration()
```

Einträge auf der Hauptdiagonalen<sup>3</sup> der Konfusionsmatrix entsprechen korrekten Klassifikationen, die übrigen Einträge dagegen verraten uns, wie viele Datenpunkte einer Kategorie versehentlich einer anderen Kategorie zugeordnet wurden.

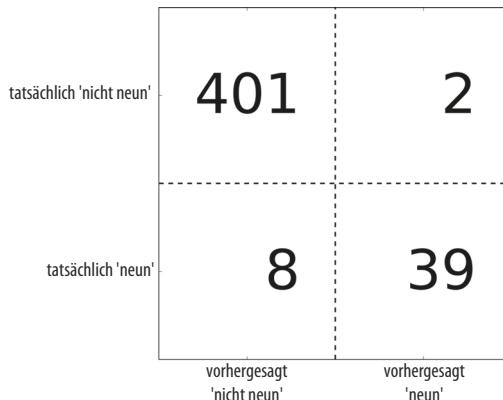


Abbildung 5-10: Konfusionsmatrix der Klassifikationsaufgabe "neun vs. Rest"

Wenn wir »neun« als die positive Kategorie festlegen, können wir die Einträge der Konfusionsmatrix mit den zuvor eingeführten Begriffen *falsch positiv* und *falsch negativ* belegen. Um das Bild zu vervollständigen, nennen wir die korrekt klassifizierten Datenpunkte der positiven Kategorie *richtig positiv* und die korrekt klassifizierten Punkte der negativen Kategorie *richtig negativ*. Diese Begriffe werden für gewöhnlich mit FP, FN, RP (engl. TP) und RN (engl. TN) abgekürzt. Sie führen zu folgender Interpretation der Konfusionsmatrix (Abbildung 5-11):

---

<sup>3</sup> Die Hauptdiagonale eines zweidimensionalen Array oder einer Matrix A ist A[i, i].

In[45]:

```
mglearn.plots.plot_binary_confusion_matrix()
```

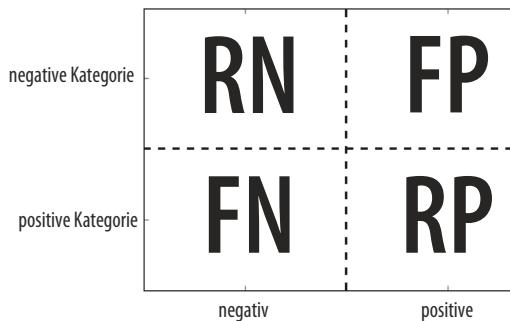


Abbildung 5-11: Konfusionsmatrix für die binäre Klassifikation

Verwenden wir nun die Konfusionsmatrix, um die weiter oben angepassten Modelle zu vergleichen (die zwei Dummy-Modelle, den Entscheidungsbaum und die logistische Regression):

In[46]:

```
print("Häufigste Kategorie:")
print(confusion_matrix(y_test, pred_most_frequent))
print("\nDummy-Modell:")
print(confusion_matrix(y_test, pred_dummy))
print("\nEntscheidungsbaum:")
print(confusion_matrix(y_test, pred_tree))
print("\nLogistische Regression")
print(confusion_matrix(y_test, pred_logreg))
```

Out[46]:

```
Häufigste Kategorie:
[[403  0]
 [ 47  0]]
Dummy-Modell:
[[361  42]
 [ 43   4]]
Entscheidungsbaum:
[[390  13]
 [ 24  23]]
Logistische Regression
[[401  2]
 [  8  39]]
```

Beim Betrachten der Konfusionsmatrix wird schnell klar, dass mit `pred_most_frequent` etwas nicht stimmt, weil es stets die gleiche Kategorie vorhersagt. `pred_dummy` hingegen hat eine sehr kleine Anzahl richtig positiver (4), besonders im Vergleich mit der Anzahl falsch negativer und falsch positiver – es gibt viel mehr falsch positive als richtig positive! Die vom Entscheidungsbaum getroffenen Vorhersagen sind

deutlich sinnvoller als die Dummy-Vorhersagen, auch wenn die Genauigkeit fast die gleiche war. Schließlich sehen wir, dass die logistische Regression in allen Belangen besser als `pred_tree` abschneidet: Sie enthält mehr richtig positive und richtig negative und weniger falsch positive und falsch negative. Aus diesem Vergleich geht klar hervor, dass nur der Entscheidungsbaum und die logistische Regression sinnvolle Vorhersagen treffen konnten und dass die logistische Regression in allen Aspekten besser als der Baum funktioniert. Das Inspizieren der gesamten Konfusionsmatrix ist ein wenig beschwerlich, und obwohl wir eine Menge aus der Betrachtung der Matrix erfahren haben, war der Prozess manuell und qualitativ. Es gibt mehrere Möglichkeiten, die Information in einer Konfusionsmatrix zusammenzufassen. Diese werden wir als Nächstes erörtern.

**Der Zusammenhang mit der Genauigkeit.** Wir haben bereits eine Möglichkeit gesehen, die Ergebnisse der Konfusionsmatrix zusammenzufassen – nämlich indem wir die Genauigkeit berechnen. Diese lässt sich folgendermaßen ausdrücken:

$$\text{Genauigkeit} = \frac{\text{RP} + \text{RN}}{\text{RP} + \text{RN} + \text{FP} + \text{FN}}$$

Anders ausgedrückt ist die Genauigkeit die Anzahl korrekter Vorhersagen (RP und RN) geteilt durch die Anzahl aller Datenpunkte (die Summe aller Einträge der Konfusionsmatrix).

**Relevanz, Sensitivität und F-Maß.** Es gibt einige weitere Möglichkeiten zum Zusammenfassen einer Konfusionsmatrix. Dabei werden am häufigsten die Relevanz und die Sensitivität verwendet. Die *Relevanz* misst, wie viele der als positiv vorhergesagten Datenpunkte tatsächlich positiv sind:

$$\text{Relevanz} = \frac{\text{RP}}{\text{RP} + \text{FP}}$$

Die Relevanz wird als Metrik eingesetzt, wenn die Anzahl der falsch positiven begrenzt werden soll. Beispielsweise soll ein Modell vorhersagen, ob ein neues Medikament im Rahmen einer klinischen Studie eine Krankheit effektiv behandeln kann. Klinische Studien sind sündhaft teuer, und ein pharmazeutisches Unternehmen wird solch einen Versuch nur starten, wenn es sich sehr sicher sein kann, dass das Medikament auch wirklich funktioniert. Deshalb ist es wichtig, dass das Modell nicht viele falsch positive produziert – es also eine hohe Relevanz hat. Die Relevanz nennt man auch *positiver Vorhersagewert* (PPV).

Die *Sensitivität* dagegen misst, wie viele der positiven Datenpunkte von den positiven Vorhersagen erfasst werden:

$$\text{Sensitivität} = \frac{\text{RP}}{\text{RP} + \text{FN}}$$

Die Sensitivität kommt als Metrik dann zum Einsatz, wenn wir alle positiven Datenpunkte finden müssen; das heißt, wenn es wichtig ist, falsch negative zu vermeiden. Das obige Beispiel mit der Krebsdiagnose ist ein gutes Beispiel dafür: Es ist wichtig, alle kranken Patienten zu erkennen, auch wenn möglicherweise gesunde Personen in die Vorhersage aufgenommen werden. Andere Bezeichnungen für die Sensitivität sind *Empfindlichkeit*, *Trefferquote* oder *Richtig-positiv-Rate* (RPR).

Es existiert ein Gleichgewicht zwischen dem Optimieren der Sensitivität und dem Optimieren der Relevanz. Sie können trivialerweise eine perfekte Sensitivität erzielen, indem Sie alle Datenpunkte der positiven Kategorie zuordnen – es gibt dann keine falsch negativen und keine richtig negativen. Allerdings führt die Vorhersage aller Punkte als positive zu vielen falsch positiven, und daher ist die Relevanz sehr niedrig. Wenn Sie dagegen ein Modell finden, das nur einen einzigen Datenpunkt, bei dem es sich absolut sicher ist, als positiv vorhersagt und den Rest als negativ, erhalten Sie eine perfekte Sensitivität (vorausgesetzt dieser Datenpunkt ist tatsächlich positiv). Die Relevanz ist dann aber sehr schlecht.

Auch wenn Relevanz und Sensitivität sehr wichtige Metriken sind, gibt Ihnen eine von beiden allein keinen Aufschluss über das Gesamtbild. Eine Möglichkeit, diese zusammenzufassen, ist der *F-Score* oder *F-Maß*, das harmonische Mittel von Relevanz und Sensitivität:



Relevanz und Sensitivität sind nur zwei von vielen Metriken für Klassifikationen, die von RP, FP, RN und FN abgeleitet sind. Sie finden eine ausführliche Zusammenfassung dieser Maße auf Wikipedia ([https://en.wikipedia.org/wiki/Sensitivity\\_and\\_specificity](https://en.wikipedia.org/wiki/Sensitivity_and_specificity)). In der Community für maschinelles Lernen sind Relevanz und Sensitivität die am häufigsten verwendeten Metriken für binäre Klassifikation, aber andere Communitys mögen andere Metriken aus diesem Bereich bevorzugen.

$$F = 2 \cdot \frac{\text{relevanz} \cdot \text{sensitivität}}{\text{relevanz} + \text{sensitivität}}$$

Diese Variante ist auch als *F<sub>1</sub>-Score* bekannt. Da das Maß sowohl Relevanz als auch Sensitivität berücksichtigt, ist es bei nicht balancierten Datensätzen ein besseres Maß als die Genauigkeit. Wir werden es nun für die weiter oben erhaltenen Vorhersagen auf dem Datensatz »neun vs. Rest« berechnen. Wir gehen hier davon aus, dass die Kategorie »neun« die positive Kategorie ist (sie ist mit True beschriftet, der Rest dagegen als False), sodass die positive Kategorie in der Minderheit ist:

### In[47]:

```
from sklearn.metrics import f1_score
print("F1-Score häufigste Kategorie: {:.2f}".format(
    f1_score(y_test, pred_most_frequent)))
print("F1-Score Dummy-Modell: {:.2f}".format(f1_score(y_test, pred_dummy)))
print("F1-Score Entscheidungsbaum: {:.2f}".format(f1_score(y_test, pred_tree)))
```

```
print("F1-Score logistische Regression: {:.2f}".format(  
    f1_score(y_test, pred_logreg)))
```

**Out[47]:**

```
F1-Score häufigste Kategorie: 0.00  
F1-Score Dummy-Modell: 0.10  
F1-Score Entscheidungsbaum: 0.55  
F1-Score logistische Regression: 0.89
```

Wir stellen zwei Dinge fest. Erstens erhalten wir für die Vorhersage `most_frequent` eine Fehlermeldung, da es für die positive Klasse keine Vorhersagen gab (dadurch wird der Nenner im *F*-Score null). Wir sehen auch einen sehr deutlichen Unterschied zwischen den Dummy-Vorhersagen und den Vorhersagen des Entscheidungsbaumes, was beim Betrachten der Genauigkeit allein nicht klar war. Mit dem *F*-Score haben wir die Vorhersageleistung wieder in einer Zahl zusammengefasst. Allerdings erfasst der *F*-Score unsere eigene Einschätzung dessen, was ein gutes Modell ausmacht, viel besser als die Genauigkeit. Ein Nachteil des *F*-Score ist aber, dass er sich nicht so leicht wie die Genauigkeit erklären und interpretieren lässt.

Falls wir uns eine übersichtlichere Zusammenfassung von Relevanz, Sensitivität und *F*<sub>1</sub>-Score wünschen, können wir die Hilfsfunktion `classification_report` aufrufen, um alle drei Metriken auf einmal zu berechnen und hübsch formatiert auszugeben:

**In[48]:**

```
from sklearn.metrics import classification_report  
print(classification_report(y_test, pred_most_frequent,  
                            target_names=["nicht neun", "neun"]))
```

**Out[48]:**

	precision	recall	f1-score	support
nicht neun	0.90	1.00	0.94	403
neun	0.00	0.00	0.00	47
avg / total	0.80	0.90	0.85	450

Die Funktion `classification_report` liefert uns eine Zeile pro Kategorie (hier `True` und `False`) mit Relevanz (precision), Sensitivität (recall) und *F*-Score mit dieser Kategorie als positive Kategorie. Wir hatten zuvor die weniger häufige Kategorie »neun« als positive Kategorie betrachtet. Wenn wir die positive Kategorie zu »nicht neun« ändern, sehen wir in der Ausgabe von `classification_report`, dass wir mit dem Modell `most_frequent` einen *F*-Score von 0.94 erhalten. Außerdem hat die Kategorie »nicht neun« eine Sensitivität von 1, da wir sämtliche Datenpunkte als »nicht neun« klassifiziert haben. Die letzte Spalte neben dem *F*-Score gibt den *support* jeder Kategorie an. Dies steht einfach für die Anzahl Datenpunkte in dieser Kategorie entsprechend den tatsächlichen Werten.

Die letzte Zeile im Klassifikationsbericht zeigt einen gewichteten Durchschnitt (nach der Anzahl Punkte in jeder Kategorie) der Zahlen in jeder Kategorie. Hier sind zwei

weitere Berichte, einer für den Dummy-Klassifikator, einer für die logistische Regression:

**In[49]:**

```
print(classification_report(y_test, pred_dummy,
                            target_names=["nicht neun", "neun"]))
```

**Out[49]:**

	precision	recall	f1-score	support
nicht neun	0.90	0.92	0.91	403
neun	0.11	0.09	0.10	47
avg / total	0.81	0.83	0.82	450

**In[50]:**

```
print(classification_report(y_test, pred_logreg,
                            target_names=["nicht neun", "neun"]))
```

**Out[50]:**

	precision	recall	f1-score	support
nicht neun	0.98	1.00	0.99	403
neun	0.95	0.83	0.89	47
avg / total	0.98	0.98	0.98	450

Beim Lesen dieser Berichte könnten Ihnen auffallen, dass die Unterschiede zwischen dem Dummy-Modell und einem sehr guten Modell nicht mehr ganz so deutlich sind. Die Entscheidung, welche Kategorie die positive Kategorie sein soll, hat einen großen Einfluss auf die Metriken. Für die Kategorie »neun« ist der F-Score der Dummy-Klassifikation 0.13 (gegenüber 0.89 bei der logistischen Regression), während diese Werte für die Kategorie »nicht neun« jeweils 0.90 und 0.99 betragen. Die letzten beiden Werte sehen beide nach sinnvollen Vorhersagen aus. Betrachtet man alle diese Zahlen gemeinsam, ergibt sich jedoch ein recht genaues Bild, und die Überlegenheit des logistischen Regressionsmodells tritt deutlich zutage.

## Berücksichtigen von Unsicherheit

Die Konfusionsmatrix und der Klassifikationsbericht bilden eine sehr detaillierte Analysemöglichkeit für einen bestimmten Satz von Vorhersagen. Allerdings haben die Vorhersagen selbst bereits eine Menge Information verworfen, die im Modell enthalten ist. Wie in Kapitel 2 besprochen, enthalten die meisten Klassifikatoren die Methoden `decision_function` oder `predict_proba`, um die Zuverlässigkeit einer Vorhersage anzugeben. Eine Vorhersage lässt sich als Anwendung eines Schwellenwertes auf die Ausgabe einer Entscheidungsfunktion wie `decision_function` oder `predict_proba` an einem festen Punkt beschreiben – bei der binären Klassifikation verwenden wir 0 für die Entscheidungsfunktion und 0.5 für `predict_proba`.

Das folgende Beispiel ist eine nicht balancierte Klassifikationsaufgabe mit 400 Datenpunkten in der negativen Kategorie gegenüber 50 Punkten in der positiven Kategorie. Die Trainingsdaten sind auf der linken Seite in Abbildung 5-12 angezeigt. Wir trainieren ein Kernel-SVM-Modell auf diesen Daten. Die Diagramme auf der rechten Seite der Trainingsdaten stellen die Werte der Entscheidungsfunktion als Heatmap dar. Sie können oben in der Mitte des Diagramms einen schwarzen Kreis sehen, der den Nullwert von `decision_function` anzeigt. Punkte innerhalb dieses Kreises werden der positiven Kategorie zugeordnet, Punkte außerhalb der negativen Kategorie:

**In[51]:**

```
from mglearn.datasets import make_blobs
X, y = make_blobs(n_samples=(400, 50), centers=2, cluster_std=[7.0, 2],
                  random_state=22)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
svc = SVC(gamma=.05).fit(X_train, y_train)
```

**In[52]:**

```
mglearn.plots.plot_decision_threshold()
```

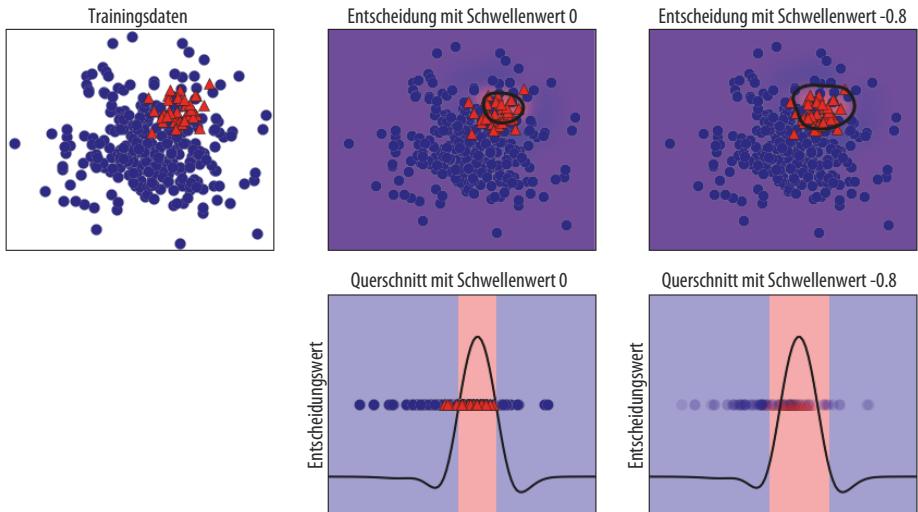


Abbildung 5-12: Heatmap der Entscheidungsfunktion und Einfluss von Änderungen des Schwellenwerts

Wir können die Funktion `classification_report` verwenden, um Relevanz und Sensitivität beider Kategorien auszuwerten:

**In[53]:**

```
print(classification_report(y_test, svc.predict(X_test)))
```

**Out[53]:**

	precision	recall	f1-score	support
0	0.97	0.89	0.93	104
1	0.35	0.67	0.46	9
avg / total	0.92	0.88	0.89	113

Für Kategorie 1 erhalten wir eine recht kleine Relevanz (precision), und die Sensitivität (recall) ist mittelmäßig. Weil Kategorie 0 so viel größer ist, konzentriert sich der Klassifikator auf die korrekte Vorhersage von Kategorie 0 und vernachlässigt die kleinere Kategorie 1.

Nehmen wir an, in unserer Anwendung sei eine hohe Sensitivität für Kategorie 1 wichtig, wie im Beispiel der oben besprochenen Krebsvorsorgeuntersuchung. Wir würden also mehr falsch positive (falsche Kategorie 1) in Kauf nehmen, um im Gegenzug mehr richtig positive (welche die Sensitivität ansteigen lassen) zu erhalten. Die von `svc.predict` generierten Vorhersagen erfüllen diese Voraussetzung nicht, aber wir können die Vorhersagen auf eine höhere Sensitivität für Kategorie 1 einstellen, indem wir den Schwellenwert für die Entscheidung weg von 0 ändern. Standardmäßig werden Punkte, für die `decision_function` mehr als 0 liefert, der Kategorie 1 zugeordnet. Wir möchten *mehr* Punkte in Kategorie 1 erhalten, daher *senken* wir den Schwellenwert:

**In[54]:**

```
y_pred_lower_threshold = svc.decision_function(X_test) > -.8
```

Betrachten wir den Klassifikationsbericht für diese Vorhersage:

**In[55]:**

```
print(classification_report(y_test, y_pred_lower_threshold))
```

**Out[55]:**

	precision	recall	f1-score	support
0	1.00	0.82	0.90	104
1	0.32	1.00	0.49	9
avg / total	0.95	0.83	0.87	113

Wie erwartet, ist die Sensitivität für Kategorie 1 angestiegen, während die Relevanz gesunken ist. Wir klassifizieren nun einen größeren Teil des Raumes als Kategorie 1, wie im Diagramm oben rechts in Abbildung 5-12 dargestellt. Wenn Ihnen die Sensitivität wichtiger ist als die Relevanz oder umgekehrt oder Ihre Daten sehr schlecht balanciert sind, ist das Ändern der Entscheidungsschwelle die leichteste Möglichkeit, ein besseres Ergebnis zu erhalten. Da `decision_function` beliebige Werte annehmen kann, ist das Angeben einer Faustregel zum Wählen des Schwellenwertes schwierig.



Wenn Sie einen Schwellenwert setzen, müssen Sie aufpassen, dass Sie dazu nicht die Testdaten verwenden. Wie bei jedem anderen Parameter führt auch das Ändern der Entscheidungsschwelle auf den Testdaten leicht zu einem übermäßig optimistischen Ergebnis. Verwenden Sie stattdessen Validationsdaten oder eine Kreuzvalidierung.

Bei Modellen, die die Methode `predict_proba` implementieren, ist die Auswahl eines Schwellenwertes einfacher, da der Rückgabewert von `predict_proba` auf einer Skala von 0 bis 1 festgelegt ist. Bei dem voreingestellte Schwellenwert von 0.5 wird ein Punkt der positiven Kategorie zugeordnet, wenn das Modell sich dessen zu mehr als 50 % sicher ist. Erhöhen des Schwellenwertes bedeutet, dass das Modell für eine positive Entscheidung sicherer sein muss (und weniger sicher für eine negative Entscheidung). Die Arbeit mit Wahrscheinlichkeiten kann zugänglicher sein als ein abstrakter Schwellenwert, es berechnen jedoch nicht alle Modelle realistische Wahrscheinlichkeiten (z. B. ist sich ein zu voller Tiefe entwickelter Entscheidungsbaum seiner Entscheidungen immer zu 100 % sicher, sogar wenn er sich häufig irrt). Dies bezeichnet man als *Kalibrierung*: Ein kalibriertes Modell liefert ein genaues Maß für die Unsicherheit seiner Vorhersagen. Eine detaillierte Besprechung von Kalibration sprengt den Rahmen dieses Buches, aber Sie können mehr Details im Artikel »Predicting Good Probabilities with Supervised Learning« ([http://www.machinelearning.org/proceedings/icml2005/papers/079\\_GoodProbabilities\\_NiculescuMizilCaruana.pdf](http://www.machinelearning.org/proceedings/icml2005/papers/079_GoodProbabilities_NiculescuMizilCaruana.pdf)) von Alexandru Niculescu-Mizil und Rich Caruana finden.

### Relevanz-Sensitivitäts-Kurven und ROC-Kurven

Wie soeben besprochen, können wir die Balance zwischen Relevanz und Sensitivität durch Ändern des Schwellenwertes für die Entscheidung eines Klassifikators verschieben. Nehmen wir an, Sie möchten weniger als 10 % der positiven Datenpunkte verlieren, also eine Sensitivität von 90 % erhalten. Die Entscheidung hierfür hängt von Ihrer Anwendung ab und sollte von Ihren Geschäftszielen bestimmt sein. Sobald ein bestimmtes Ziel gesetzt ist – z. B. ein bestimmter Wert für die Sensitivität oder Relevanz einer Kategorie –, können Sie den Schwellenwert entsprechend einstellen. Es ist immer möglich, den Schwellenwert so einzustellen, dass ein Ziel wie die Sensitivität von 90 % erreicht wird. Der schwierige Teil ist, ein Modell zu entwickeln, das mit diesem Schwellenwert noch immer eine gute Relevanz besitzt – wenn Sie alles als positiv klassifizieren, bekommen Sie eine Sensitivität von 100 %, aber Ihr Modell wird nutzlos.

Eine Anforderung an einen Klassifikator wie 90 % Sensitivität wird häufig auch als Setzen des *Arbeitspunktes* bezeichnet. Festlegen eines Arbeitspunktes ist in geschäftlichen Situationen hilfreich, um Kunden oder anderen Gruppen in Ihrer Organisation gegenüber eine bestimmte Leistung zu garantieren.

Beim Entwickeln eines neuen Modells ist es meist nicht ganz klar, was der Arbeitspunkt sein wird. Dazu und zum besseren Verständnis einer Modellierungsaufgabe ist es lehrreich, sich sämtliche möglichen Schwellenwerte bzw. die gesamte Wechselbeziehung zwischen Relevanz und Sensitivität *zeitgleich* anzusehen. Dies ist mit einem Werkzeug namens *Relevanz-Sensitivitäts-Kurve* möglich. Sie finden im Modul `sklearn.metrics` eine Funktion zum Berechnen einer Relevanz-Sensitivitäts-Kurve. Sie benötigt die tatsächlichen Labels und die mit `decision_function` oder `predict_proba` vorhergesagten Unsicherheiten:

**In[56]:**

```
from sklearn.metrics import precision_recall_curve
precision, recall, thresholds = precision_recall_curve(
    y_test, svc.decision_function(X_test))
```

Die Funktion `precision_recall_curve` liefert uns eine sortierte Liste von Relevanz- und Sensitivitätswerten für alle möglichen Schwellenwerte (alle in der Entscheidungsfunktion vorhandenen Werte), sodass wir eine Kurve wie die in Abbildung 5-13 zeichnen können:

**In[57]:**

```
# Verwende mehr Datenpunkte, um eine weichere Kurve zu erhalten
X, y = make_blobs(n_samples=(4000, 500), centers=2, cluster_std=[7.0, 2],
                  random_state=22)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
svc = SVC(gamma=.05).fit(X_train, y_train)
precision, recall, thresholds = precision_recall_curve(
    y_test, svc.decision_function(X_test))
# finde den der Null nächsten Schwellenwert
close_zero = np.argmin(np.abs(thresholds))
plt.plot(precision[close_zero], recall[close_zero], 'o', markersize=10,
          label="threshold zero", fillstyle="none", c='k', mew=2)

plt.plot(precision, recall, label="precision recall curve")
plt.xlabel("Relevanz")
plt.ylabel("Sensitivität")
```

Jeder Punkt entlang der Kurve in Abbildung 5-13 entspricht einem möglichen Schwellenwert von `decision_function`. Wir sehen dort beispielsweise, dass wir eine Sensitivität von 0.4 bei einer Relevanz von etwa 0.75 erreichen können. Der schwarze Kreis markiert den Punkt, der dem voreingestellten Schwellenwert 0 entspricht. Dieser Punkt wird beim Aufruf der Methode `predict` angesteuert.

Je näher eine Kurve der oberen rechten Ecke ist, desto besser ist der Klassifikator. Ein Punkt oben rechts bedeutet hohe Relevanz *und* hohe Sensitivität beim gleichen Schwellenwert. Die Kurve beginnt in der oberen linken Ecke, die einem sehr niedrigen Schwellenwert entspricht. Dort wird alles der positiven Kategorie zugeordnet. Beim Erhöhen des Schwellenwertes bewegt sich die Kurve auf eine höhere Relevanz zu, aber auch auf eine niedrigere Sensitivität. Wenn wir den Schwellenwert noch weiter erhöhen, gelangen wir in einen Bereich, in dem die meisten als positiv klas-

sifizierten Punkte richtig positive sind, die Relevanz also hoch und die Sensitivität niedrig ist. Je höher ein Modell die Sensitivität hält, während sich die Relevanz erhöht, desto besser ist das Modell.

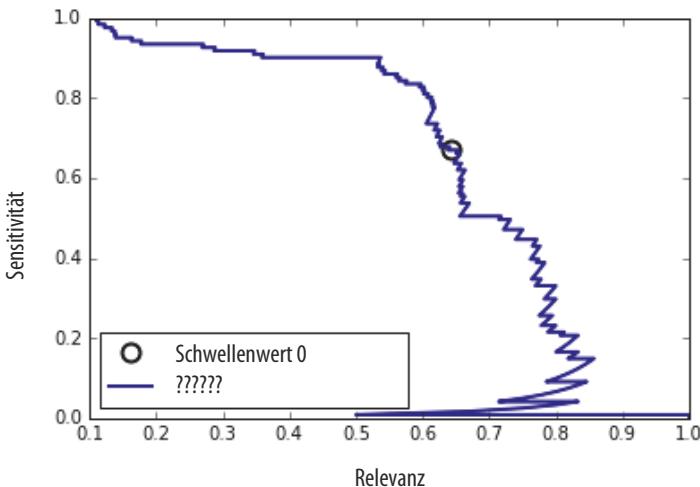


Abbildung 5-13: Relevanz-Sensitivitäts-Kurve für SVC( $\gamma=0.05$ )

Beim genaueren Betrachten dieser Kurve sehen wir, dass dieses Modell eine Relevanz von etwa 0.5 mit einer sehr hohen Sensitivität erreichen kann. Falls wir eine sehr viel höhere Relevanz wünschen, müssen wir dafür eine Menge Sensitivität opfern. Anders gesagt, ist die Kurve auf der linken Seite recht flach, und die Sensitivität verringert sich kaum, wenn wir eine höhere Relevanz benötigen. Bei einer Relevanz über 0.5 kostet uns jeder Zugewinn an Relevanz eine Menge Sensitivität.

Unterschiedliche Klassifikatoren funktionieren in unterschiedlichen Bereichen dieser Kurve unterschiedlich gut – sie haben also einen unterschiedlichen Arbeitspunkt. Vergleichen wir ein SVM-Modell mit einem auf dem gleichen Datensatz trainierten Random Forest. Der RandomForestClassifier besitzt keine Methode `decision_function`, nur `predict_proba`. Die Funktion `precision_recall_curve` erwartet als zweites Argument ein Konfidenzmaß für die positive Kategorie (Kategorie 1). Daher übergeben wir die Wahrscheinlichkeit, dass ein Datenpunkt zu Kategorie 1 gehört – also `rf.predict_proba(X_test)[:, 1]`. Der voreingestellte Schwellenwert für `predict_proba` bei der binären Klassifikation ist 0.5, dies ist der im Diagramm markierte Punkt (siehe Abbildung 5-14):

**In[58]:**

```
from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(n_estimators=100, random_state=0, max_features=2)
rf.fit(X_train, y_train)

# RandomForestClassifier hat predict_proba, aber nicht decision_function
```

```

precision_rf, recall_rf, thresholds_rf = precision_recall_curve(
    y_test, rf.predict_proba(X_test)[:, 1])

plt.plot(precision, recall, label="svc")

plt.plot(precision[close_zero], recall[close_zero], 'o', markersize=10,
         label="threshold zero svc", fillstyle="none", c='k', mew=2)

plt.plot(precision_rf, recall_rf, label="rf")

close_default_rf = np.argmin(np.abs(thresholds_rf - 0.5))
plt.plot(precision_rf[close_default_rf], recall_rf[close_default_rf], '^', c='k',
         markersize=10, label="threshold 0.5 rf", fillstyle="none", mew=2)

plt.xlabel("Relevanz")
plt.ylabel("Sensitivität")
plt.legend(loc="best")

```

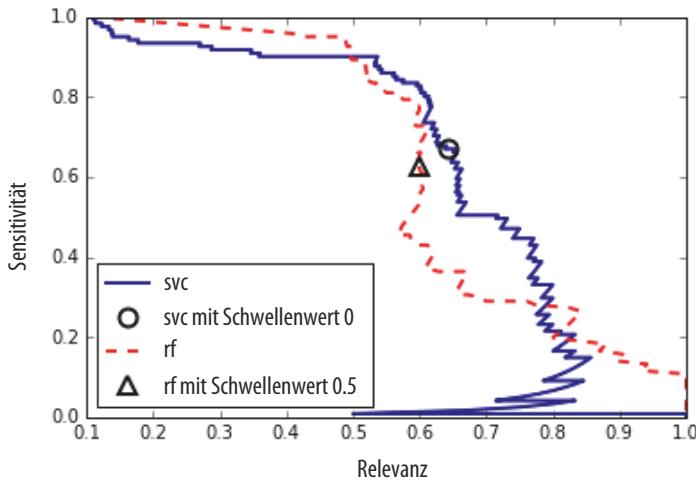


Abbildung 5-14: Vergleich von Relevanz-Sensitivitäts-Kurven einer SVM und eines Random Forest

Im Vergleichsdiagramm können wir beobachten, dass der Random Forest in den Extremlagen besser abschneidet, wenn also eine sehr hohe Sensitivität oder sehr hohe Relevanz erforderlich ist. In der Mitte (etwa bei Relevanz = 0.7) schneidet das SVM-Modell besser ab. Würden wir nur den  $F_1$ -Score zum Vergleich der gesamten Vorhersageleistung heranziehen, würden wir diese subtilen Unterschiede übersehen. Der  $F_1$ -Score erfasst nur einen Punkt auf der Relevanz-Sensitivitäts-Kurve, nämlich den beim voreingestellten Schwellenwert:

In[59]:

```

print("F1-Score für den Random Forest: {:.3f}".format(
    f1_score(y_test, rf.predict(X_test))))
print("F1-Score für den SVC: {:.3f}".format(f1_score(y_test, svc.predict(X_test))))

```

## Out[59]:

```
F1-Score für den Random Forest: 0.610  
F1-Score für den SVC: 0.656
```

Der Vergleich zweier Relevanz-Sensitivitäts-Kurven liefert eine Menge detaillierter Erkenntnisse, ist aber weitgehend Handarbeit. Zum automatischen Vergleich von Modellen müssten wir die Information in der Kurve zusammenfassen, ohne uns auf einen bestimmten Schwellenwert oder Arbeitspunkt zu beschränken. Eine Möglichkeit zum Zusammenfassen einer Relevanz-Sensitivitäts-Kurve ist, das Integral oder die Fläche unter der Kurve zu berechnen. Diese nennt man auch *mittlere Relevanz*.<sup>4</sup> Sie können die mittlere Relevanz über die Funktion `average_precision_score` berechnen. Weil wir eine Relevanz-Sensitivitäts-Kurve berechnen und mehrere Schwellenwerte in Betracht ziehen möchten, müssen wir das Ergebnis von `decision_function` oder `predict_proba` an `average_precision_score` übergeben (und nicht das Ergebnis von `predict`):

## In[60]:

```
from sklearn.metrics import average_precision_score  
ap_rf = average_precision_score(y_test, rf.predict_proba(X_test)[:, 1])  
ap_svc = average_precision_score(y_test, svc.decision_function(X_test))  
print("Mittlere Relevanz des Random Forest: {:.3f}".format(ap_rf))  
print("Mittlere Relevanz des SVC: {:.3f}".format(ap_svc))
```

## Out[60]:

```
Mittlere Relevanz des Random Forest: 0.666  
Mittlere Relevanz des SVC: 0.663
```

Bei der Mittelwertbildung über alle möglichen Schwellenwerte sehen wir, dass Random Forest und SVC ähnlich gut abschneiden. Der Random Forest liegt sogar leicht vorn. Dieses Ergebnis unterscheidet sich deutlich von dem der oben berechneten F1-Scores. Weil die Fläche unter der Kurve (AUC) der Fläche unter einer Kurve von 0 bis 1 entspricht, liegt die Fläche unter der Kurve (AUC) immer bei Werten zwischen 0 (schlechterer) und 1 (bester). Die mittlere Relevanz eines Klassifikators, der `decision_function` zufällig zuweist, entspricht dem Anteil positiver Datenpunkte im Datensatz.

## Receiver Operating Characteristic (ROC) und AUC

Es gibt ein weiteres, verbreitetes Werkzeug zum Analysieren des Verhaltens von Klassifikatoren bei verschiedenen Schwellenwerten: die *Receiver-Operating-Characteristic*-, kurz *ROC-Kurve*. Ähnlich zur Relevanz-Sensitivität-Kurve betrachtet die ROC-Kurve alle möglichen Schwellenwerte eines gegebenen Klassifikators, aber anstatt Relevanz und Sensitivität zu berechnen, zeigt sie die *Falsch-positiv-Rate* (FPR) über der *Richtig-positiv-Rate* (RPR). Die Richtig-positiv-Rate ist wie

---

<sup>4</sup> Es gibt einige kleine technische Unterschiede zwischen der Fläche unter einer Relevanz-Sensitivitäts-Kurve und der mittleren Relevanz. Der Begriff gibt aber die wesentliche Idee wieder.

oben erwähnt einfach ein anderer Name für die Sensitivität, während die Falsch-positiv-Rate der Anteil an falsch positiven an allen negativen Datenpunkten ist:

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{RN}}$$

Die ROC-Kurve lässt sich mit der Funktion `roc_curve` berechnen (siehe Abbildung 5-15):

In[61]:

```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_test, svc.decision_function(X_test))

plt.plot(fpr, tpr, label="ROC-Kurve")
plt.xlabel("FPR")
plt.ylabel("RPR (Sensitivität)")
# finde den Schwellenwert, der der Null am nächsten ist
close_zero = np.argmin(np.abs(thresholds))
plt.plot(fpr[close_zero], tpr[close_zero], 'o', markersize=10,
         label="threshold zero", fillstyle="none", c='k', mew=2)
plt.legend(loc=4)
```

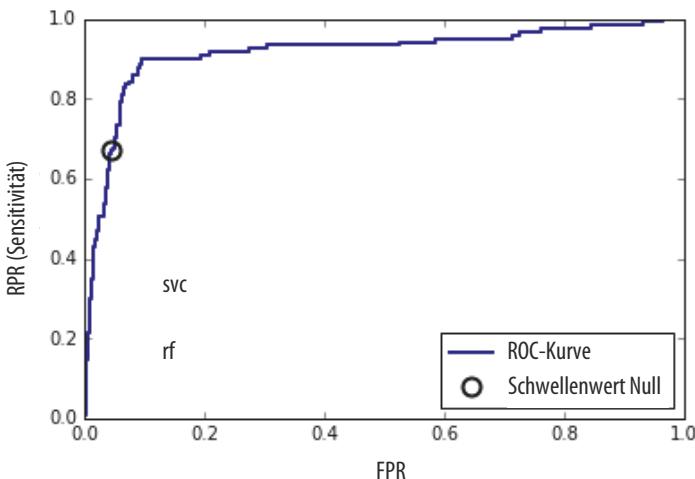


Abbildung 5-15: ROC-Kurve für ein SVM-Modell

Die ideale ROC-Kurve ist nahe an der oberen linken Ecke: Sie möchten idealerweise einen Klassifikator, der eine *hohe Sensitivität* bei gleichzeitig *niedriger Falsch-positiv-Rate* aufweist. Verglichen mit dem voreingestellten Schwellenwert 0, zeigt die Kurve, dass wir deutlich höhere Sensitivität erzielen können (um 0.9), wobei sich die FPR nur leicht erhöht. Ein der oberen linken Ecke am nächsten gelegene Punkt könnte ein besserer Arbeitspunkt sein als der voreingestellte. Auch hier sollten Sie im Hinterkopf behalten, dass die Auswahl eines Schwellenwertes niemals auf dem Testdatensatz, sondern auf separaten Validationsdaten vorgenommen werden sollte.

In Abbildung 5-16 finden Sie einen Vergleich der ROC-Kurven für einen Random Forest und die SVM:

In[62]:

```
from sklearn.metrics import roc_curve
fpr_rf, tpr_rf, thresholds_rf = roc_curve(y_test, rf.predict_proba(X_test)[:, 1])

plt.plot(fpr, tpr, label="ROC-Kurve SVC")
plt.plot(fpr_rf, tpr_rf, label="ROC-Kurve RF")

plt.xlabel("FPR")
plt.ylabel("RPR (Sensitivität)")
plt.plot(fpr[close_zero], tpr[close_zero], 'o', markersize=10,
         label="threshold zero SVC", fillstyle="none", c='k', mew=2)
close_default_rf = np.argmin(np.abs(thresholds_rf - 0.5))
plt.plot(fpr_rf[close_default_rf], tpr_rf[close_default_rf], '^', markersize=10,
         label="threshold 0.5 RF", fillstyle="none", c='k', mew=2)

plt.legend(loc=4)
```

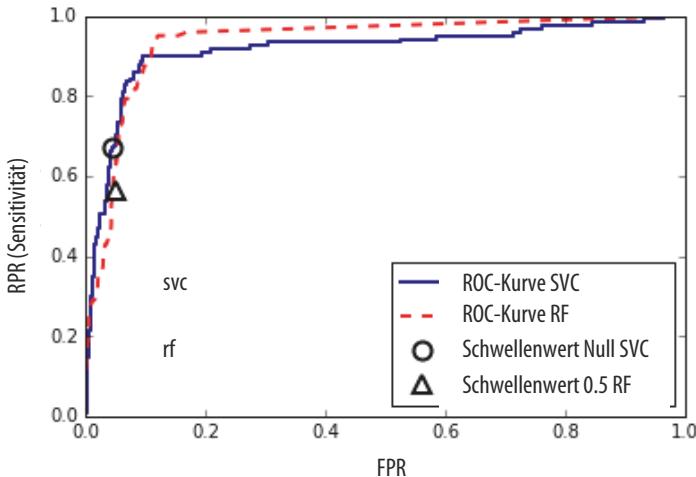


Abbildung 5-16: Vergleich der ROC-Kurven für eine SVM und einen Random Forest

Wie bei der Relevanz-Sensitivitäts-Kurve möchte man meist auch die ROC-Kurve in einer einzigen Zahl zusammenfassen, die die Fläche unter der Kurve beziffert (auch genannt *Area under the Curve*, AUC, wobei mit der Kurve im Sprachgebrauch immer die ROC-Kurve gemeint ist). Wir können die Fläche unter der ROC-Kurve mit der Funktion `roc_auc_score` ermitteln:

In[63]:

```
from sklearn.metrics import roc_auc_score
rf_auc = roc_auc_score(y_test, rf.predict_proba(X_test)[:, 1])
svc_auc = roc_auc_score(y_test, svc.decision_function(X_test))
print("AUC für den Random Forest: {:.3f}".format(rf_auc))
print("AUC für den SVC: {:.3f}".format(svc_auc))
```

**Out[63]:**

```
AUC für den Random Forest: 0.937  
AUC für den SVC: 0.916
```

Beim Vergleich von Random Forest und SVM über den AUC-Score bemerken wir, dass der Random Forest deutlich besser als die SVM abschneidet. Weil die Kurve zwischen 0 und 1 verläuft, ergibt die mittlere Relevanz, also die Fläche unter der Kurve, immer einen Wert zwischen 0 (schlechter) und 1 (bester). Eine zufällige Vorhersage liefert immer ein AUC von 0.5, egal wie schlecht balanciert die Kategorien im Datensatz sind. Damit ist AUC eine viel bessere Metrik für nicht balancierte Klassifikationsaufgaben als die Genauigkeit. Der AUC-Score lässt sich als *Rangfolge* der positiven Datenpunkte interpretieren. Er ist zur Wahrscheinlichkeit äquivalent, mit der ein zufällig ausgewählter Punkt aus der positiven Kategorie vom Klassifizierer einen höheren Score zugeordnet bekommt als ein zufällig ausgewählter Punkt aus der negativen Kategorie. Ein perfekter AUC von 1 bedeutet also, dass alle positiven Punkte einen höheren Score haben als alle negativen Punkte. Bei Klassifikationsaufgaben mit nicht balancierten Kategorien ist die Verwendung des AUC zur Modellauswahl oft viel bedeutsamer als die Genauigkeit.

Kehren wir noch einmal zurück zur weiter oben betrachteten Aufgabe der Klassifikation sämtlicher Neunen im Datensatz digits gegenüber allen anderen Ziffern. Wir werden diesen Datensatz mit einer SVM mit drei unterschiedlichen Einstellungen der Kernel-Bandbreite gamma klassifizieren (siehe Abbildung 5-17):

**In[64]:**

```
y = digits.target == 9  
  
X_train, X_test, y_train, y_test = train_test_split(  
    digits.data, y, random_state=0)  
  
plt.figure()  
  
for gamma in [1, 0.05, 0.01]:  
    svc = SVC(gamma=gamma).fit(X_train, y_train)  
    accuracy = svc.score(X_test, y_test)  
    auc = roc_auc_score(y_test, svc.decision_function(X_test))  
    fpr, tpr, _ = roc_curve(y_test, svc.decision_function(X_test))  
    print("gamma = {:.2f}  accuracy = {:.2f}  AUC = {:.2f}".format(  
        gamma, accuracy, auc))  
    plt.plot(fpr, tpr, label="gamma={:.3f}".format(gamma))  
plt.xlabel("FPR")  
plt.ylabel("RPR")  
plt.xlim(-0.01, 1)  
plt.ylim(0, 1.02)  
plt.legend(loc="best")
```

**Out[64]:**

```
gamma = 1.00  accuracy = 0.90  AUC = 0.50  
gamma = 0.05  accuracy = 0.90  AUC = 0.90  
gamma = 0.01  accuracy = 0.90  AUC = 1.00
```

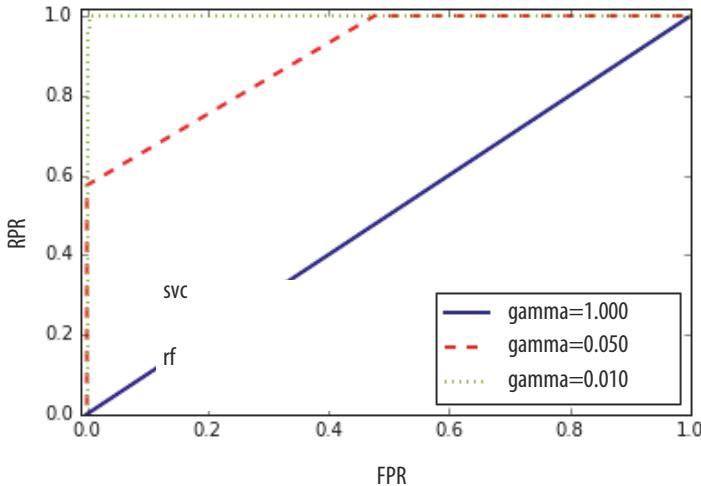


Abbildung 5-17: Vergleich von ROC-Kurven von SVMs mit unterschiedlichen Werten für  $\gamma$

Die Genauigkeit aller drei Werte für  $\gamma$  ist mit 90 % identisch. Dies könnte die gleiche Vorhersagequalität sein wie eine zufällige Klassifikation, oder auch nicht. Wenn wir jedoch die AUC-Werte und die entsprechenden Kurven betrachten, sehen wir einen deutlichen Unterschied zwischen den drei Modellen. Bei  $\gamma = 1.0$  liegt die AUC beim Zufallsniveau, also ist die Ausgabe von `decision_function` so gut wie eine zufällige. Mit  $\gamma = 0.05$  verbessert sich die Vorhersage drastisch auf eine AUC von 0.9. Schließlich erhalten wir mit  $\gamma = 0.01$  eine perfekte AUC von 1.0. Damit wurden alle positiven Punkte von der Entscheidungsfunktion höher als alle negativen Punkte eingeordnet. Anders gesagt, könnte dieses Modell die Daten mit dem richtigen Schwellenwert perfekt klassifizieren!<sup>5</sup> Mit dieser Erkenntnis können wir den Schwellenwert dieses Modells anpassen und großartige Vorhersagen vornehmen. Hätten wir nur die Genauigkeit betrachtet, hätten wir dies nie herausgefunden.

Aus diesem Grund empfehlen wir Ihnen wärmstens, die AUC zum Evaluieren von Modellen auf nicht balancierten Daten einzusetzen. Bedenken Sie, dass die AUC den voreingestellten Schwellenwert nicht verwendet. Um aus einem Modell mit hoher AUC nützliche Klassifikationsergebnisse zu erhalten, müssen Sie den Schwellenwert der Entscheidungsfunktion einstellen.

---

<sup>5</sup> Wenn Sie sich die Kurve bei  $\gamma = 0.01$  genau ansehen, bemerken Sie einen kleinen Knick oben links. Dies bedeutet, dass mindestens ein Punkt nicht korrekt zugeordnet wurde. Die AUC von 1.0 ist eine Folge des Rundens auf die zweite Dezimalstelle.

## Metriken zur Klassifikation mehrerer Kategorien

Nun haben wir die Auswertung binärer Klassifikationsaufgaben zur Genüge besprochen und wenden uns Metriken zur Evaluierung der Klassifikation mehrerer Kategorien zu. Im Wesentlichen sind alle Metriken zur Klassifikation mehrerer Kategorien von binären Klassifikationsmetriken abgeleitet und über sämtliche Kategorien gemittelt. Die Genauigkeit bei der Klassifikation mehrerer Kategorien ist auch hier als Anteil der korrekt zugeordneten Beispiele definiert. Auch hier ist die Genauigkeit kein gutes Maß, wenn die Kategorien nicht balanciert sind. Stellen Sie sich eine Klassifikation mit drei Kategorien vor, bei der 85 % der Punkte zu Kategorie A gehören, 10 % zu Kategorie B und 5 % zu Kategorie C. Was bedeutet eine Genauigkeit von 85 % auf diesem Datensatz? Im Allgemeinen sind die Ergebnisse einer Klassifikation mit mehreren Kategorien schwerer zu verstehen als im binären Fall. Außer der Genauigkeit sind die Konfusionsmatrix und der im binären Fall betrachtete Klassifikationsbericht verbreitete Werkzeuge. Wenden wir diese beiden Methoden auf die Klassifikation aller zehn handschriftlichen Ziffern im Datensatz digits an:

**In[65]:**

```
from sklearn.metrics import accuracy_score
X_train, X_test, y_train, y_test = train_test_split(
    digits.data, digits.target, random_state=0)
lr = LogisticRegression().fit(X_train, y_train)
pred = lr.predict(X_test)
print("Genauigkeit: {:.3f}".format(accuracy_score(y_test, pred)))
print("Konfusionsmatrix:\n{}".format(confusion_matrix(y_test, pred)))
```

**Out[65]:**

```
Genauigkeit: 0.953
Konfusionsmatrix:
[[37  0  0  0  0  0  0  0  0  0]
 [ 0 39  0  0  0  0  2  0  2  0]
 [ 0  0 41  3  0  0  0  0  0  0]
 [ 0  0  1 43  0  0  0  0  0  1]
 [ 0  0  0  0 38  0  0  0  0  0]
 [ 0  1  0  0  0 47  0  0  0  0]
 [ 0  0  0  0  0  0 52  0  0  0]
 [ 0  1  0  1  1  0  0 45  0  0]
 [ 0  3  1  0  0  0  0  0 43  1]
 [ 0  0  0  1  0  1  0  0  1 44]]
```

Das Modell weist eine Genauigkeit von 95.3 % auf. Dadurch wissen wir bereits, dass wir recht gut dastehen. Die Konfusionsmatrix liefert uns die Details. Wie im binären Fall steht jede Zeile für ein korrektes Label und jede Spalte für ein vorhergesagtes. Sie finden ein ansprechenderes Diagramm in Abbildung 5-18:

**In[66]:**

```
scores_image = mglearn.tools.heatmap(
    confusion_matrix(y_test, pred), xlabel='Vorhergesagtes Label',
```

```

ylabel='True label', xticklabels=digits.target_names,
      yticklabels=digits.target_names, cmap=plt.cm.gray_r, fmt="%d")
plt.title("Konfusionsmatrix")
plt.gca().invert_yaxis()

```

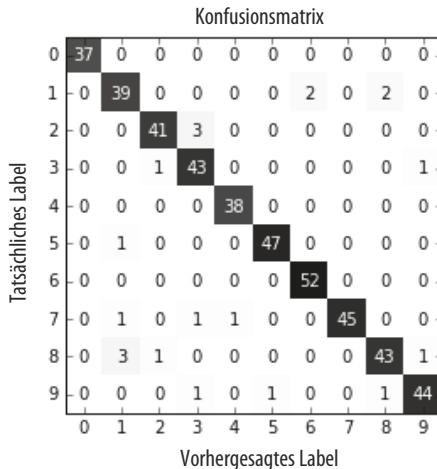


Abbildung 5-18: Konfusionsmatrix für die Klassifikation von 10 Ziffern

In der ersten Kategorie, der Ziffer 0, gibt es 37 Datenpunkte. Alle diese Punkte wurden der Kategorie 0 zugeordnet (es gibt keine falsch negativen). Wir erkennen das daran, dass alle übrigen Einträge in der ersten Zeile der Konfusionsmatrix 0 sind. Wir sehen auch, dass keine anderen Ziffern fälschlicherweise als 0 klassifiziert wurden, weil alle anderen Einträge in der ersten Spalte der Konfusionsmatrix ebenfalls 0 sind (es gibt keine falsch positiven). Einige Ziffern wurden mit anderen verwechselt – zum Beispiel wurde die Ziffer 2 (dritte Zeile) dreimal als Ziffer 3 (vierte Spalte) klassifiziert. Es gab auch eine Ziffer 3, die als eine 2 erkannt wurde (dritte Spalte, vierte Zeile) und eine Ziffer 8, die als 2 klassifiziert wurde (dritte Spalte, vierte Zeile).

Mit der Funktion `classification_report` können wir die Relevanz, die Sensitivität und den F-Score jeder Kategorie berechnen:

**In[67]:**

```
print(classification_report(y_test, pred))
```

**Out[67]:**

	precision	recall	f1-score	support
0	1.00	1.00	1.00	37
1	0.89	0.91	0.90	43
2	0.95	0.93	0.94	44
3	0.90	0.96	0.92	45
4	0.97	1.00	0.99	38
5	0.98	0.98	0.98	48
6	0.96	1.00	0.98	52
7	1.00	0.94	0.97	48

8	0.93	0.90	0.91	48
9	0.96	0.94	0.95	47
avg / total	0.95	0.95	0.95	450

Es ist nicht überraschend, dass Relevanz und Sensitivität bei Kategorie 0 perfekt sind, da es in dieser Kategorie keine Verwechslungen gab. Bei Kategorie 7 dagegen ist die Relevanz 1, da keine andere Ziffer versehentlich als 7 klassifiziert wurde. Bei der Ziffer 6 gibt es dagegen keine falsch negativen, daher ist dort die Sensitivität 1. Wir sehen auch, dass das Modell mit den Kategorien 8 und 3 seine Schwierigkeiten hatte.

Die am häufigsten verwendete Metrik bei nicht balancierten Datensätzen bei mehreren Kategorien ist eine für mehrere Kategorien erweiterte Variante des  $F$ -Score. Der Grundgedanke des  $F$ -Score für mehrere Kategorien ist, einen binären  $F$ -Score pro Kategorie zu berechnen, wobei diese Kategorie die positive Kategorie ist und die anderen Kategorien gemeinsam die negativen Kategorien bilden. Anschließend lassen sich diese  $F$ -Scores für alle Kategorien mit einer der folgenden Strategien mitteilen:

- "macro"-Durchschnittsbildung berechnet die ungewichteten  $F$ -Scores für jede Kategorie. Damit erhalten alle Kategorien unabhängig von ihrer Größe das gleiche Gewicht.
- "weighted"-Durchschnittsbildung berechnet den Mittelwert der  $F$ -Scores, wobei jede Kategorie nach ihrer Größe gewichtet wird. Diesen Score sehen wir im Klassifikationsbericht.
- "micro"-Durchschnittsbildung berechnet die Gesamtzahl an falsch positiven, falsch negativen und richtig positiven über sämtliche Kategorien und berechnet daraus Relevanz, Sensitivität und  $F$ -Score mit diesen Zahlen.

Wenn Ihnen jeder *Datenpunkt* gleich wichtig ist, empfehlen wir die "micro"-Durchschnittsbildung für den  $F_1$ -Score; wenn Ihnen jede *Kategorie* gleich wichtig ist, empfehlen wir den  $F_1$ -Score mit "macro"-Durchschnittsbildung:

**In[68]:**

```
print("F1-Score mit Micro-Durchschnittsbildung: {:.3f}".format
      (f1_score(y_test, pred, average="micro")))
print("F1-Score mit Macro-Durchschnittsbildung: {:.3f}".format
      (f1_score(y_test, pred, average="macro")))
```

**Out[68]:**

```
F1-Score mit Micro-Durchschnittsbildung: 0.953
F1-Score mit Macro-Durchschnittsbildung: 0.954
```

## Regressionsmetriken

Die Evaluierung einer Regression lässt sich ähnlich detailliert wie bei einer Klassifikation durchführen – beispielsweise durch Analysieren, ob man die Zielgröße über-

mäßig oder unzureichend vorhersagt. Allerdings genügt der standardmäßig von der Methode `score` eines Regressors berechnete  $R^2$ -Wert für die meisten Anwendungen, die wir gesehen haben. Bisweilen werden Geschäftsentscheidungen auf der Grundlage des mittleren quadratischen Fehlers oder des mittleren absoluten Fehlers getroffen, was ein Anreiz dafür sein kann, Modelle für diese Metriken zu optimieren. Im Allgemeinen haben wir die Erfahrung gemacht, dass zum Auswerten von Regressionsmodellen der  $R^2$ -Wert eine verständlichere Metrik ist.

## Verwenden von Metriken zur Modellauswahl

Wir haben viele Methoden zur Evaluierung und ihre Anwendung mit Vergleichsdaten und einem Modell im Detail besprochen. Allerdings möchte man Metriken wie die AUC bei der Modellauswahl mit `GridSearchCV` oder `cross_val_score` einsetzen. Glücklicherweise gibt es in `scikit-learn` eine sehr einfache Möglichkeit, beides zu erreichen. Das Argument `scoring` lässt sich sowohl bei `GridSearchCV` als auch bei `cross_val_score` verwenden. Sie können einfach die gewünschte Evaluierungsmetrik als String angeben. Beispielsweise könnten wir den SVM-Klassifikator auf der Aufgabe »neun vs. Rest« mit dem Datensatz `digits` mit dem AUC-Score auswerten wollen. Indem wir für den Parameter `scoring` den Wert `"roc_auc"` einsetzen, ändern wir die Voreinstellung (Genauigkeit) auf AUC:

**In[69]:**

```
# die Genauigkeit ist als Score voreingestellt
print("voreingestellter Score: {}".format(
    cross_val_score(SVC(), digits.data, digits.target == 9)))
# Angaben von scoring="accuracy" ändert das Ergebnis nicht
explicit_accuracy = cross_val_score(SVC(), digits.data, digits.target == 9,
                                      scoring="accuracy")
print("explizit eingestellte Genauigkeit: {}".format(explicit_accuracy))
roc_auc = cross_val_score(SVC(), digits.data, digits.target == 9,
                           scoring="roc_auc")
print("AUC-Score: {}".format(roc_auc))
```

**Out[69]:**

```
voreingestellter Score: [ 0.9  0.9  0.9]
explizit eingestellte Genauigkeit: [ 0.9  0.9  0.9]
AUC-Score: [ 0.994  0.99   0.996]
```

In ähnlicher Weise können wir die zur Auswahl der besten Parameter verwendete Metrik in `GridSearchCV` einstellen:

**In[70]:**

```
X_train, X_test, y_train, y_test = train_test_split(
    digits.data, digits.target == 9, random_state=0)

# wir geben zur Verdeutlichung ein suboptimales Gitter an:
param_grid = {'gamma': [0.0001, 0.01, 0.1, 1, 10]}
# mit der voreingestellten Genauigkeit
grid = GridSearchCV(SVC(), param_grid=param_grid)
```

```

grid.fit(X_train, y_train)
print("Gittersuche mit Genauigkeit:")
print("Beste Parameter:", grid.best_params_)
print("Bester Score der Kreuzvalidierung (Genauigkeit): {:.3f}".format(grid.best_score_))
print("AUC für die Testdaten: {:.3f}".format(
    roc_auc_score(y_test, grid.decision_function(X_test))))
print("Genauigkeit auf den Testdaten: {:.3f}".format(grid.score(X_test, y_test)))

```

**Out[70]:**

```

Gittersuche mit Genauigkeit
Beste Parameter: {'gamma': 0.0001}
Bester Score der Kreuzvalidierung (Genauigkeit): 0.970
AUC für die Testdaten: 0.992
Genauigkeit auf den Testdaten: 0.973

```

**In[71]:**

```

# nun verwenden wir stattdessen den AUC-Score:
grid = GridSearchCV(SVC(), param_grid=param_grid, scoring="roc_auc")
grid.fit(X_train, y_train)
print("\nGittersuche mit AUC")
print("Beste Parameter:", grid.best_params_)
print("Bester Score der Kreuzvalidierung (AUC): {:.3f}".format(grid.best_score_))
print("AUC auf den Testdaten: {:.3f}".format(
    roc_auc_score(y_test, grid.decision_function(X_test))))
print("Genauigkeit auf den Testdaten: {:.3f}".format(grid.score(X_test, y_test)))

```

**Out[71]:**

```

Gittersuche mit AUC
Beste Parameter: {'gamma': 0.01}
Bester Score der Kreuzvalidierung (AUC): 0.997
AUC auf den Testdaten: 1.000
Genauigkeit auf den Testdaten: 1.000

```

Verwenden wir die Genauigkeit, wird der Parameter `gamma=0.0001` ausgewählt, beim AUC-Score dagegen `gamma=0.01`. Die Genauigkeit der Kreuzvalidierung ist in beiden Fällen mit der Genauigkeit der Testdaten konsistent. Beim Verwenden des AUC finden wir aber Parameter, die zu einem höheren AUC-Score und sogar einer höheren Genauigkeit führen.<sup>6</sup>

Die wichtigsten Werte für den Parameter `scoring` bei der Klassifikation sind `accuracy` (der voreingestellte Wert), `roc_auc` für die Fläche unter der ROC-Kurve, `average_precision` für die Fläche unter der Relevanz-Sensitivitäts-Kurve, `f1`, `f1_macro`, `f1_micro` und `f1_weighted` für den binären  $F_1$ -Score und dessen unterschiedlich gewichtete Varianten. Bei der Regression sind die am häufigsten verwendeten Werte `r2` für den  $R^2$ -Score, `mean_squared_error` für den mittleren quadratischen Fehler und `mean_absolute_error` für den mittleren Absolutfehler. Sie finden eine vollständige Liste der möglichen Argumente in der Dokumentation ([http://scikit-learn.org/stable/modules/model\\_evaluation.html#the-scoring-parameter-defining-](http://scikit-learn.org/stable/modules/model_evaluation.html#the-scoring-parameter-defining-)

---

<sup>6</sup> Dass wir mit der AUC eine Lösung mit höherer Genauigkeit finden, liegt daran, dass die Genauigkeit ein schlechtes Maß ist, um die Vorhersagequalität auf nicht balancierten Daten zu bestimmen.

*model-evaluation-rules*) oder indem Sie sich das Dictionary SCORER im Modul `metrics.scorer` ansehen.

**In[72]:**

```
from sklearn.metrics.scorer import SCORERS
print("verfügbare Scores:\n{}".format(sorted(SCORERS.keys())))
```

**Out[72]:**

```
verfügbare Scores:
['accuracy', 'adjusted_rand_score', 'average_precision', 'f1', 'f1_macro',
 'f1_micro', 'f1_samples', 'f1_weighted', 'log_loss', 'mean_absolute_error',
 'mean_squared_error', 'median_absolute_error', 'precision', 'precision_macro',
 'precision_micro', 'precision_samples', 'precision_weighted', 'r2', 'recall',
 'recall_macro', 'recall_micro', 'recall_samples', 'recall_weighted', 'roc_auc']
```

## Zusammenfassung und Ausblick

In diesem Kapitel haben wir Kreuzvalidierung, Gittersuche und Metriken zur Evaluation besprochen, die Fundamente beim Evaluieren und Verbessern maschiner Lernalgorithmen. Die in diesem Kapitel besprochenen Werkzeuge sind gemeinsam mit den in den Kapiteln 2 und 3 beschriebenen Algorithmen das tägliche Brot jedes Anwenders maschiner Lernverfahren.

Wir haben in diesem Kapitel zwei besondere Punkte besprochen, die man nicht oft genug wiederholen kann, weil sie von neuen Anwendern oft übersehen werden. Der erste hat mit der Kreuzvalidierung zu tun. Kreuzvalidierung oder die Nutzung eines Testdatensatzes ermöglicht uns, zu bewerten, welche Leistung ein maschinelles Lernmodell in der Zukunft erbringen wird. Wenn wir aber die Testdaten oder die Kreuzvalidierung verwenden, um ein Modell auszuwählen oder Modellparameter zu bestimmen, so »verbrauchen« wir die Testdaten. Die gleichen Testdaten zum Bewerten der zukünftigen Modellqualität führen dann zu übertrieben optimistischen Schätzungen. Wir müssen uns daher auf die Teilung in Trainingsdaten zum Aufbau des Modells, Validierungsdaten zur Modell- und Parameterauswahl und Testdaten zur Auswertung des Modells stützen. Anstelle einer einfachen Teilung können wir jede dieser Teilungen über eine Kreuzvalidierung durchführen. Die häufigste Form dabei ist (wie oben beschrieben) eine Teilung in Trainings- und Testdaten zur Evaluierung und das Verwenden von Kreuzvalidierung auf den Trainingsdaten zur Modell- und Parameterauswahl.

Der zweite Punkt hängt mit der Wichtigkeit der Metrik zur Evaluation oder der Scoring-Funktion zur Modellauswahl und -auswertung zusammen. Die Theorie, wie man auf der Grundlage von Vorhersagen eines maschinellen Lernmodells Geschäftsentscheidungen fällt, liegt außerhalb der Themen dieses Buches.<sup>7</sup> Es

---

<sup>7</sup> Wir können Ihnen wärmstens das Buch von Foster Provost und Tom Fawcett, *Data Science for Business* (O'Reilly, ISBN 978-1-4493-6132-7), empfehlen, in dem Sie mehr zu diesem Thema finden.

kommt jedoch selten vor, dass das Endziel einer maschinellen Lernaufgabe ist, ein Modell mit hoher Genauigkeit zu konstruieren. Stellen Sie sicher, dass Sie eine Metrik zum Evaluieren und Auswählen eines Modells verwenden, die sich exakt dafür eignet, wofür das Modell am Ende verwendet wird. Im wirklichen Leben sind die Kategorien bei Klassifikationsaufgaben selten balanciert, und daher haben falsch positive und falsch negative häufig sehr unterschiedliche Folgen. Stellen Sie sicher, dass Sie verstehen, welche Folgen dies sind, und wählen Sie die Metrik zur Evaluierung dementsprechend aus.

Die beschriebenen Techniken zur Evaluierung und Auswahl von Modellen sind die bisher wichtigsten Werkzeuge eines Data Scientists. Die in diesem Kapitel beschriebene Gittersuche und Kreuzvalidierung lässt sich nur auf ein einzelnes überwachtes Modell anwenden. Wir haben aber bereits gesehen, dass viele Modelle Vorverarbeitungsschritte benötigen und dass für manche Anwendungen wie die Gesichtserkennung in Kapitel 3 eine andere Repräsentation der Daten nützlich sein kann. Im nächsten Kapitel werden wir die Klasse `Pipeline` kennenlernen, mit der wir die Gittersuche und Kreuzvalidierung auch mit diesen komplexeren Ketten von Algorithmen verwenden können.

# Verkettete Algorithmen und Pipelines

Wie in Kapitel 4 besprochen, ist die Repräsentation der Daten für viele maschinelle Lernalgorithmen von entscheidender Bedeutung. Dies fängt mit dem Skalieren der Daten und dem manuellen Kombinieren von Merkmalen an und erstreckt sich bis zum Ermitteln von Merkmalen durch unüberwachtes Lernen, wie in Kapitel 3 beschrieben. Deshalb benötigen die meisten Anwendungen nicht nur einen einzelnen Algorithmus, sondern die Verkettung mehrerer Verarbeitungsschritte und maschineller Lernmodelle. In diesem Kapitel beschäftigen wir uns mit der Verwendung der Klasse `Pipeline`, um den Aufbau verketteter Transformationen und Modelle zu vereinfachen. Insbesondere werden wir sehen, wie wir `Pipeline` und `GridSearchCV` miteinander kombinieren können, um gleichzeitig nach Parametern für alle Verarbeitungsschritte zu suchen.

Als Beispiel dafür, wie wichtig die Verkettung von Modellen ist, haben wir bereits festgestellt, dass die Verwendung des `MinMaxScaler` zur Vorverarbeitung des Datensatzes `cancer` die Vorhersagequalität eines Kernel-SVM erheblich steigert. Dies ist der Code zum Aufteilen der Daten, Berechnen von Minimum und Maximum, Skalieren der Daten und Trainieren der SVM:

**In[1]:**

```
from sklearn.svm import SVC
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

# lade die Daten und teile sie auf
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

# berechne Minimum und Maximum der Trainingsdaten
scaler = MinMaxScaler().fit(X_train)
```

In[2]:

```
# skaliere die Trainingsdaten um  
  
X_train_scaled = scaler.transform(X_train)  
svm = SVC()  
# trainiere eine SVM auf den umskalierten Trainingsdaten  
svm.fit(X_train_scaled, y_train)  
# skaliere die Testdaten um und berechne den Score  
X_test_scaled = scaler.transform(X_test)  
print("Test Score: {:.2f}".format(svm.score(X_test_scaled, y_test)))
```

Out[2]:

```
Test Score: 0.95
```

## Parameterauswahl mit Vorverarbeitung

Nehmen wir an, wir möchten mithilfe von GridSearchCV bessere Parameter für SVC finden, wie in Kapitel 5 vorgestellt. Wie sollten wir diese Aufgabe angehen? Ein nai- ver Ansatz könnte wie folgt aussehen:

In[3]:

```
from sklearn.model_selection import GridSearchCV  
# nur zur Veranschaulichung, diesen Code nicht verwenden!  
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],  
             'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}  
grid = GridSearchCV(SVC(), param_grid=param_grid, cv=5)  
grid.fit(X_train_scaled, y_train)  
print("Beste Genauigkeit nach Kreuzvalidierung: {:.2f}".format(grid.best_score_))  
print("Bester Score auf den Testdaten: {:.2f}".format(  
      grid.score(X_test_scaled, y_test)))  
print("Beste Parameter: ", grid.best_params_)
```

Out[3]:

```
Beste Genauigkeit nach Kreuzvalidierung: 0.98  
Bester Score auf den Testdaten: 0.97  
Beste Parameter: {'gamma': 1, 'C': 1}
```

Hier haben wir die Gittersuche über die Parameter von SVC mit den skalierten Daten ausgeführt. Dabei gibt es aber einen subtilen Fallstrick. Beim Skalieren der Daten haben wir *sämtliche Daten im Trainingsdatensatz* verwendet, um herauszufinden, wie wir das Modell trainieren sollen. Anschließend verwenden wir die *skalierten Trainingsdaten*, um unsere Gittersuche mit Kreuzvalidierung auszuführen. Bei jeder Teilung im Verlauf der Kreuzvalidierung wird ein anderer Fold der ursprünglichen Trainingsdaten zum Trainieren innerhalb der Teilung verwendet und der Rest zum Testen. Mithilfe der Testdaten messen wir, wie gut das auf dem Trainingsteil trainierte Modell mit neuen Daten umgehen kann. Allerdings haben wir die Information aus dem Fold zum Testen bereits beim Skalieren verwendet. Bei

der Kreuzvalidierung ist ja jeder Fold Teil des Trainingsdatensatzes, und wir haben die Information aus dem gesamten Trainingsdatensatz verwendet, um die richtige Skalierung der Daten herauszufinden. *Dies ist etwas grundsätzlich anderes, als dem Modell neue Daten zu präsentieren.* Wenn wir neue Daten beobachten (z. B. in Form unseres Testdatensatzes), haben wir diese Daten nicht beim Skalieren der Trainingsdaten berücksichtigt, und sie können daher ein anderes Minimum und Maximum aufweisen als die Trainingsdaten. Das folgende Beispiel (Abbildung 6-1) zeigt, wie sich die Verarbeitung der Daten bei der Kreuzvalidierung und der endgültigen Evaluation unterscheidet:

In[4]:

```
mglearn.plots.plot_improper_processing()
```

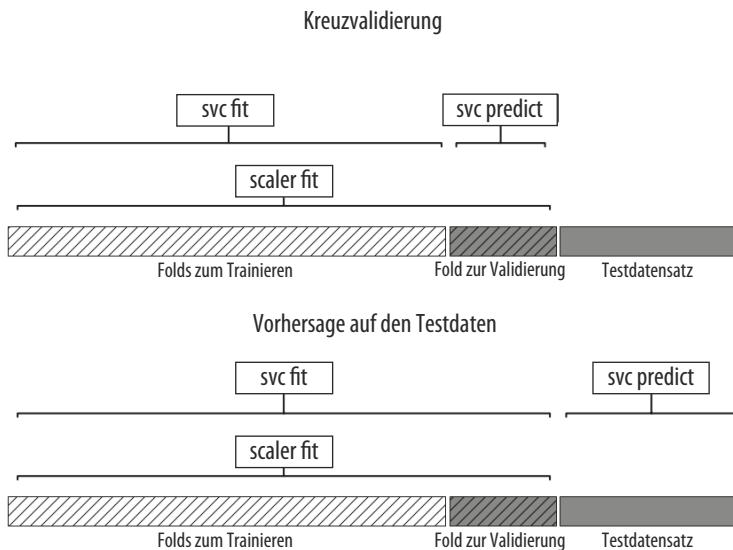


Abbildung 6-1: Nutzung der Daten bei der Vorverarbeitung außerhalb der Kreuzvalidierungsschleife

Die Teilungen der Kreuzvalidierung bilden also die Daten für den Modellbildungsprozess nicht mehr korrekt ab. Wir haben diesen Teil der Daten bereits in den Modellbildungsprozess einfließen lassen. Das führt zu übermäßig optimistischen Ergebnissen bei der Kreuzvalidierung und möglicherweise zur Auswahl suboptimaler Parameter.

Um dieses Problem zu umgehen, sollte die Aufteilung des Datensatzes zur Kreuzvalidierung *vor jeglicher Vorverarbeitung* durchgeführt werden. Jeder Vorgang, der Informationen aus dem Datensatz verwendet, sollte erst auf den Trainingsteil des Datensatzes angewendet werden, sodass die Kreuzvalidierung die »äußerste Schleife« in unserer Verarbeitung ist.

Um dies in scikit-learn mit den Funktionen `cross_val_score` und `GridSearchCV` umzusetzen, verwenden wir die Klasse `Pipeline`. Die Klasse `Pipeline` ermöglicht es uns, mehrere Verarbeitungsschritte zu einem scikit-learn-Estimator miteinander zu verbinden. Die Klasse `Pipeline` selbst enthält die Methoden `fit`, `predict` und `score` und verhält sich wie jedes andere Modell in scikit-learn. Der häufigste Anwendungsfall der Klasse `Pipeline` ist, Schritte zur Vorverarbeitung (wie Skalieren der Daten) mit einem überwachten Modell, z. B. einem Klassifikator, zu verknüpfen.

## Erstellen von Pipelines

Sehen wir uns einmal an, wie wir die Arbeitsschritte zum Trainieren eines SVM nach Skalieren der Daten mit dem `MinMaxScaler` über die Klasse `Pipeline` formulieren können (zunächst ohne Gittersuche). Zuerst erstellen wir ein Pipeline-Objekt, indem wir eine Liste der Schritte angeben. Jeder Schritt ist dabei ein Tupel aus Name (ein beliebiger String Ihrer Wahl<sup>1</sup>) und einer Instanz eines Estimators:

**In[5]:**

```
from sklearn.pipeline import Pipeline
pipe = Pipeline([("scaler", MinMaxScaler()), ("svm", SVC())])
```

Hier haben wir zwei Schritte erstellt: Der erste trägt den Namen "scaler" und ist eine Instanz von `MinMaxScaler`, der zweite trägt den Namen "svm" und ist eine Instanz von `SVC`. Nun können wir die Pipeline wie jeden anderen Estimator in scikit-learn anpassen:

**In[6]:**

```
pipe.fit(X_train, y_train)
```

Dabei ruft `pipe.fit` zuerst `fit` für den ersten Schritt (den Skalierer) auf, transformiert die Trainingsdaten anschließend mit diesem Skalierer und passt dann das SVM-Modell mit den skalierten Daten an. Um eine Evaluation auf den Testdaten durchzuführen, rufen wir einfach `pipe.score` auf:

**In[7]:**

```
print("Score auf den Testdaten: {:.2f}".format(pipe.score(X_test, y_test)))
```

**Out[7]:**

```
Score auf den Testdaten: 0.95
```

Der Aufruf der Methode `score` aus der Pipeline transformiert zuerst die Testdaten mithilfe des Skalierers und ruft anschließend die Methode `score` des SVM-Modells mit den skalierten Testdaten auf. Wie Sie sehen, ist das Ergebnis zu dem mit dem Code am Anfang des Kapitels erzielten identisch, als wir die Transformation von Hand durchgeführt hatten. Durch Verwenden der Pipeline konnten wir den Code

---

<sup>1</sup> Mit einer Ausnahme: Der Name darf keinen doppelten Unterstrich `__` enthalten.

für unseren Prozess aus »Vorverarbeitung + Klassifizierung« erheblich vereinfachen. Der Hauptvorteil der Verwendung einer Pipeline ist jedoch, dass wir diese als einzelnen Estimator in `cross_val_score` oder `GridSearchCV` verwenden können.

## Pipelines zur Gittersuche einsetzen

Eine Pipeline zu einer Gittersuche zu verwenden, funktioniert genauso wie bei jedem beliebigen anderen Estimator. Wir definieren das zu durchsuchende Parametergitter und konstruieren ein `GridSearchCV` aus der Pipeline und dem Parametergitter. Beim Angeben des Parametergitters gibt es jedoch eine geringfügige Änderung. Wir müssen bei jedem Parameter angeben, zu welchem Schritt der Pipeline er gehört. Die beiden einzustellenden Parameter, `C` und `gamma`, gehören zu `SVC`, dem zweiten Schritt. Wir haben diesem Schritt den Namen "svm" zugewiesen. Die Syntax zur Definition eines Parametergitters für eine Pipeline enthält für jeden Parameter den Namen des Schrittes, gefolgt von einem `_` (einem doppelten Unterstrich) und dem Namen des Parameters. Um den Parameter `C` des Modells `SVC` zu durchsuchen, müssen wir also "`svm_C`" als Schlüssel im Dictionary mit dem Parametergitter angeben. Für `gamma` tun wir dies analog dazu:

**In[8]:**

```
param_grid = {'svm_C': [0.001, 0.01, 0.1, 1, 10, 100],  
             'svm_gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
```

Mit diesem Parametergitter können wir `GridSearchCV` wie gewohnt verwenden:

**In[9]:**

```
grid = GridSearchCV(pipe, param_grid=param_grid, cv=5)  
grid.fit(X_train, y_train)  
print("Beste Genauigkeit nach Kreuzvalidierung: {:.2f}".format(grid.best_score_))  
print("Bester Score auf den Testdaten: {:.2f}".format(grid.score(X_test, y_test)))  
print("Beste Parameter: {}".format(grid.best_params_))
```

**Out[9]:**

```
Beste Genauigkeit nach Kreuzvalidierung: 0.98  
Bester Score auf den Testdaten: 0.97  
Beste Parameter: {'svm_C': 1, 'svm_gamma': 1}
```

Im Gegensatz zur zuvor ausgeführten Gittersuche wird nun bei jeder Teilung im Zuge der Kreuzvalidierung der `MinMaxScaler` nur mit den Trainings-Folds neu eingestellt, sodass keine Information aus dem Testteil in die Parametersuche einfließt. Vergleichen Sie dies (Abbildung 6-2) mit Abbildung 6-1 weiter oben:

**In[10]:**

```
mglearn.plots.plot_proper_processing()
```

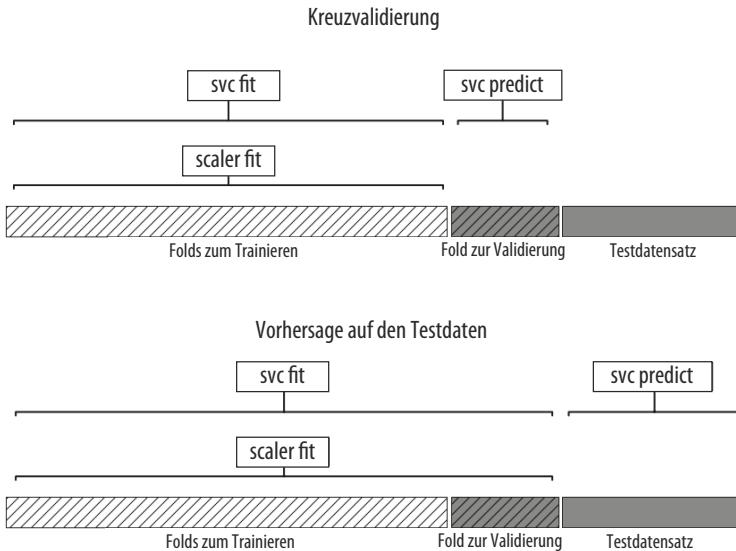


Abbildung 6-2: Verwendung der Daten mit Vorverarbeitung innerhalb der Kreuzvalidierungsschleife mit einer Pipeline

Die Auswirkungen des Entweichens von Informationen im Zuge der Kreuzvalidierung hängen von der Art des Vorverarbeitungsschrittes ab. Die Skalierung der Daten mit einem Testdatensatz hat für gewöhnlich keine schlimmen Auswirkungen, wohingegen die Verwendung der Testdaten zur Extraktion und Auswahl von Merkmalen zu grundlegend unterschiedlichen Ergebnissen führt.

## Illustration des Austretens von Information

Ein großartiges Beispiel für das Austreten von Information während der Kreuzvalidierung findet sich im Buch von Trevor Hastie, Robert Tibshirani und Jerome Friedman, *The Elements of Statistical Learning* (Springer, ISBN 978-0-387-84857-0), das wir hier in veränderter Form wiedergeben. Betrachten wir eine synthetische Regressionsaufgabe mit 100 Datenpunkten und 10000 Merkmalen, die unabhängig voneinander einer Gaußverteilung entnommen sind. Wir entnehmen auch die Zielgröße einer Gaußverteilung:

**In[11]:**

```
rnd = np.random.RandomState(seed=0)
X = rnd.normal(size=(100, 10000))
y = rnd.normal(size=(100,))
```

So wie wir diesen Datensatz erstellt haben, gibt es keinen statistischen Zusammenhang zwischen den Daten X und der Zielgröße y (sie sind voneinander unabhängig). Es sollte also nicht möglich sein, irgendetwas anhand dieses Datensatzes zu lernen. Nun probieren wir Folgendes: Zuerst wählen wir das informativste der 10000 Merk-

male mithilfe der Merkmalsauswahl `SelectPercentile` und werten anschließend einen Ridge-Regressor über Kreuzvalidierung aus:

**In[12]:**

```
from sklearn.feature_selection import SelectPercentile, f_regression

select = SelectPercentile(score_func=f_regression, percentile=5).fit(X, y)
X_selected = select.transform(X)
print("X_selected.shape: {}".format(X_selected.shape))
```

**Out[12]:**

```
X_selected.shape: (100, 500)
```

**In[13]:**

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import Ridge
print("Genauigkeit der Kreuzvalidierung (KV nur auf Ridge): {:.2f}".format(
    np.mean(cross_val_score(Ridge(), X_selected, y, cv=5))))
```

**Out[13]:**

```
Genauigkeit der Kreuzvalidierung (KV nur auf Ridge): 0.91
```

Der durch die Kreuzvalidierung berechnete  $R^2$ -Wert beträgt 0.91, weist also auf ein sehr gutes Modell hin. Dies kann nicht richtig sein, da unsere Daten vollständig zufällig sind. Hier hat unsere Merkmalsauswahl unter den 10000 zufälligen Merkmalen solche ausgewählt, die (zufällig) sehr gut mit der Zielgröße korrelieren. Weil wir die Merkmalsauswahl *außerhalb* der Kreuzvalidierung vornehmen, konnten wir sowohl in den Trainings- als auch den Testdaten korrelierte Merkmale finden. Es ist also reichlich Information aus den Testdaten ausgetreten, und das Ergebnis ist äußerst unrealistisch. Vergleichen wir dieses Ergebnis mit einer als Pipeline korrekt aufgesetzten Kreuzvalidierung:

**In[14]:**

```
pipe = Pipeline([("select", SelectPercentile(score_func=f_regression,
                                               percentile=5)),
                 ("ridge", Ridge())])
print("Genauigkeit der Kreuzvalidierung (Pipeline): {:.2f}".format(
    np.mean(cross_val_score(pipe, X, y, cv=5))))
```

**Out[14]:**

```
Genauigkeit der Kreuzvalidierung (Pipeline): -0.25
```

Diesmal ist der  $R^2$ -Wert *negativ*, was auf ein sehr schwaches Modell deutet. Mit der Pipeline findet die Merkmalsauswahl nun *innerhalb* der Kreuzvalidierungs-Schleife statt. Damit können Merkmale nur anhand der Trainingsabschnitte der Daten ausgewählt werden, nicht anhand der Testabschnitte. Die Auswahlprozedur findet zwar Merkmale, die mit der Zielgröße in den Trainingsdaten korreliert sind, aber da die Daten vollständig zufällig sind, sind die entsprechenden Merkmale in den Testdaten nicht miteinander korreliert. In diesem Beispiel macht die Reihenfolge von Merkmalsauswahl und Kreuzvalidierung den Unterschied zwischen einem scheinbar gut funktionierenden und einem völlig unzulänglichen Modell aus.

# Die allgemeine Pipeline-Schnittstelle

Die Klasse Pipeline ist nicht auf Vorverarbeitung und Klassifizierung beschränkt, sondern kann eine beliebige Anzahl Estimatoren miteinander verbinden. Zum Beispiel könnten Sie eine Pipeline mit insgesamt vier Schritten konstruieren, z. B. Extraktion von Merkmalen, Merkmalsauswahl, Skalieren und Klassifikation. Ebenso könnte der letzte Schritt anstelle der Klassifikation auch eine Regression oder ein Clustering sein.

Als einzige Voraussetzung müssen alle Estimatoren in einer Pipeline außer dem letzten Schritt die Methode `transform` enthalten, sodass sie eine neue Repräsentation der Daten für den nächsten Schritt erstellen können.

Beim Aufruf von `Pipeline.fit` ruft die Pipeline intern für jeden Schritt nacheinander `fit` und anschließend `transform` auf,<sup>2</sup> wobei als Eingabe die Ausgabe der Methode `transform` aus dem vorigen Schritt verwendet wird. Beim letzten Schritt der Pipeline wird einfach nur `fit` aufgerufen.

Ohne zu sehr ins Detail zu gehen, ist dies folgendermaßen implementiert. Das Attribut `pipeline.steps` ist eine Liste von Tupel, sodass `pipeline.steps[0][1]` der erste Estimator ist, `pipeline.steps[1][1]` der zweite Estimator usw.:

In[15]:

```
def fit(self, X, y):
    X_transformed = X
    for name, estimator in self.steps[:-1]:
        # iteriere über alle außer dem letzten Schritt
        # fitte und transformiere die Daten
        X_transformed = estimator.fit_transform(X_transformed, y)
    # fitte beim letzten Schritt
    self.steps[-1][1].fit(X_transformed, y)
    return self
```

Bei der Vorhersage mit einer Pipeline müssen wir die Daten ebenfalls in allen außer dem letzten Schritt transformieren und anschließend im letzten Schritt `predict` aufrufen:

In[16]:

```
def predict(self, X):
    X_transformed = X
    for step in self.steps[:-1]:
        # iteriere über alle außer dem letzten Schritt
        # transformiere die Daten
        X_transformed = step[1].transform(X_transformed)
    # Vorhersage beim letzten Schritt
    return self.steps[-1][1].predict(X_transformed)
```

---

<sup>2</sup> Oder einfach nur `fit_transform`.

In Abbildung 6-3 ist dieser Prozess mit zwei Transformatoren, T1 und T2, sowie einem Klassifikator (Classifier) veranschaulicht.

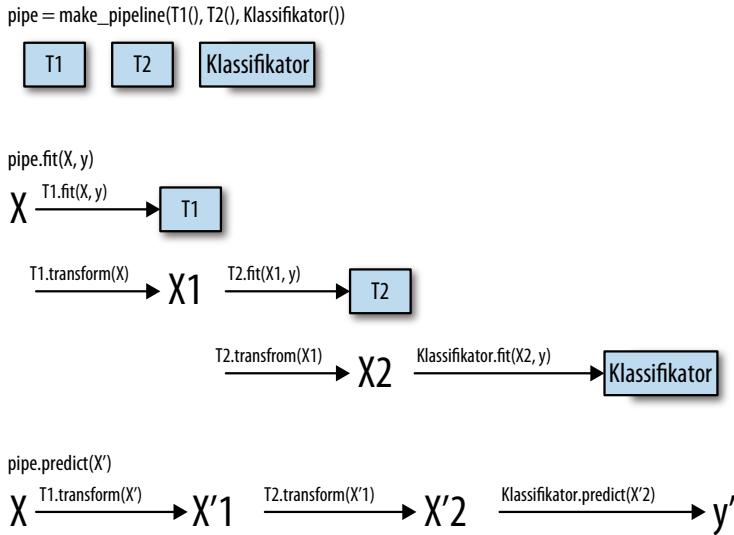


Abbildung 6-3: Überblick über das Trainieren in einer Pipeline und den Vorhersageprozess

Eine Pipeline ist sogar noch allgemeiner einsetzbar. Es ist nicht notwendig, dass der letzte Schritt einer Pipeline die Methode `predict` enthält. Wir könnten auch eine Pipeline erstellen, die z. B. nur aus einem Skalierer und einer Hauptkomponentenzerlegung (PCA) besteht. Da der letzte Schritt (PCA) die Methode `transform` enthält, könnten wir `transform` für die gesamte Pipeline aufrufen und als Ausgabe die von `PCA.transform` berechneten Daten erhalten. Im letzten Schritt einer Pipeline ist nur die Methode `fit` obligatorisch.

## Bequemes Erstellen von Pipelines mit `make_pipeline`

Eine Pipeline mit der oben beschriebenen Syntax zu erstellen, ist bisweilen etwas mühevoll. Oft benötigen wir keine benutzerdefinierten Namen für die einzelnen Schritte. Es gibt eine bequemere Funktion, `make_pipeline`, die uns eine Pipeline erstellt und jeden Schritt automatisch nach seiner Klasse benennt. Die Syntax von `make_pipeline` ist Folgende:

**In[17]:**

```

from sklearn.pipeline import make_pipeline
# Standardsyntax
pipe_long = Pipeline([("scaler", MinMaxScaler()), ("svm", SVC(C=100))])
# abgekürzte Syntax
pipe_short = make_pipeline(MinMaxScaler(), SVC(C=100))
  
```

Die Objekte `pipe_long` und `pipe_short` tun exakt das Gleiche, aber den Schritten in `pipe_short` wurden automatisch Namen zugewiesen. Wir können über das Attribut `steps` diese Namen inspizieren:

**In[18]:**

```
print("Schritte der Pipeline:\n{}".format(pipe_short.steps))
```

**Out[18]:**

```
Schritte der Pipeline:  
[('minmaxscaler', MinMaxScaler(copy=True, feature_range=(0, 1))),  
 ('svc', SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,  
             decision_function_shape=None, degree=3, gamma='auto',  
             kernel='rbf', max_iter=-1, probability=False,  
             random_state=None, shrinking=True, tol=0.001,  
             verbose=False))]
```

Die Schritte heißen `minmaxscaler` und `svc`. Im Allgemeinen sind dies die Klassenbezeichner in Kleinbuchstaben. Falls mehrere Schritte die gleiche Klasse enthalten, wird eine Zahl angehängt:

**In[19]:**

```
from sklearn.preprocessing import StandardScaler  
from sklearn.decomposition import PCA  
  
pipe = make_pipeline(StandardScaler(), PCA(n_components=2), StandardScaler())  
print("Schritte der Pipeline:\n{}".format(pipe.steps))
```

**Out[19]:**

```
Schritte der Pipeline:  
[('standardscaler-1', StandardScaler(copy=True, with_mean=True, with_std=True)),  
 ('pca', PCA(copy=True, iterated_power=4, n_components=2, random_state=None,  
             svd_solver='auto', tol=0.0, whiten=False)),  
 ('standardscaler-2', StandardScaler(copy=True, with_mean=True, with_std=True))]
```

Wie Sie sehen, erhielt der erste `StandardScaler`-Schritt den Namen `standardscaler-1` und der zweite den Namen `standardscaler-2`. Bei derartigen Szenarien ist es aber sinnvoller, die Pipeline mit expliziten Namen aufzubauen, damit jeder Schritt einen aussagekräftigeren Namen erhält.

## Zugriff auf Attribute von Schritten

Oft möchte man Attribute eines der Schritte einer Pipeline inspizieren – beispielsweise die Koeffizienten eines linearen Modells oder die von PCA ermittelten Hauptkomponenten. Der einfachste Weg, an die Schritte einer Pipeline heranzukommen, ist das Attribut `named_steps`, ein Dictionary mit den Namen der Schritte als Schlüssel und den Estimatoren als Werte:

**In[20]:**

```
# passe die zuvor erstellte Pipeline an den Datensatz cancer an  
pipe.fit(cancer.data)
```

```
# extrahiere die ersten zwei Hauptkomponenten aus dem Schritt "pca"
components = pipe.named_steps["pca"].components_
print("components.shape: {}".format(components.shape))
```

**Out[20]:**

```
components.shape: (2, 30)
```

## Zugriff auf Attribute in einer Pipeline mit Gittersuche

Wie weiter oben in diesem Kapitel besprochen, ist Gittersuche eines der Haupteinsatzgebiete von Pipelines. Eine übliche Aufgabe ist, einige der Schritte einer Pipeline im Rahmen einer Gittersuche anzusprechen. Als Beispiel führen wir eine Gittersuche für einen LogisticRegression-Klassifikator auf dem Datensatz cancer durch, wobei wir eine Pipeline und einen StandardScaler zum Skalieren der Daten vor der Übergabe an den Klassifikator LogisticRegression verwenden. Wir erstellen zuerst die Pipeline mithilfe der Funktion `make_pipeline`:

**In[21]:**

```
from sklearn.linear_model import LogisticRegression
pipe = make_pipeline(StandardScaler(), LogisticRegression())
```

Anschließend erzeugen wir das Parametergitter. Wie in Kapitel 2 erläutert, lässt sich LogisticRegression über den Regularisierungsparameter C einstellen. Wir verwenden für diesen Parameter ein logarithmisches Gitter und suchen zwischen 0.01 und 100. Weil wir die Funktion `make_pipeline` verwendet haben, ist der Name des Schrittes `LogisticRegression` in der Pipeline das kleingeschriebene `logisticregression`. Daher müssen wir das Parametergitter für `logisticregression__C` angeben, um den Parameter C zu optimieren:

**In[22]:**

```
param_grid = {'logisticregression__C': [0.01, 0.1, 1, 10, 100]}
```

Wie gewöhnlich teilen wir den Datensatz cancer in Trainings- und Testdaten auf und führen die Gittersuche durch:

**In[23]:**

```
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=4)
grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(X_train, y_train)
```

Wie können wir auf die Koeffizienten für das beste von GridSearchCV gefundene LogisticRegression-Modell zugreifen? Aus Kapitel 5 wissen wir, dass das beste Modell, welches GridSearchCV auf den Trainingsdaten trainiert hat, in `grid.best_estimator_` abgelegt ist:

**In[24]:**

```
print("Bester Estimator:\n{}".format(grid.best_estimator_))
```

**Out[24]:**

```
Bester Estimator:  
Pipeline(steps=[  
    ('standardscaler', StandardScaler(copy=True, with_mean=True, with_std=True)),  
    ('logisticregression', LogisticRegression(C=0.1, class_weight=None,  
        dual=False, fit_intercept=True, intercept_scaling=1, max_iter=100,  
        multi_class='ovr', n_jobs=1, penalty='l2', random_state=None,  
        solver='liblinear', tol=0.0001, verbose=0, warm_start=False))])
```

Dieser `best_estimator_` ist in unserem Fall eine Pipeline mit zwei Schritten, `standardscaler` und `logisticregression`. Um auf den Schritt `logisticregression` zuzugreifen, können wir wie oben erwähnt das Attribut der Pipeline `named_steps` verwenden:

**In[25]:**

```
print("logistische Regression:\n{}".format(  
    grid.best_estimator_.named_steps["logisticregression"]))
```

**Out[25]:**

```
logistische Regression:  
LogisticRegression(C=0.1, class_weight=None, dual=False, fit_intercept=True,  
    intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,  
    penalty='l2', random_state=None, solver='liblinear', tol=0.0001,  
    verbose=0, warm_start=False)
```

Nun, da wir eine trainierte Instanz von `LogisticRegression` haben, können wir auf die Koeffizienten (Gewichte) zu jedem Merkmal der Eingabe zugreifen:

**In[26]:**

```
print("Koeffizienten der logistischen Regression:\n{}".format(  
    grid.best_estimator_.named_steps["logisticregression"].coef_))
```

**Out[26]:**

```
Koeffizienten der logistischen Regression:  
[[ -0.389 -0.375 -0.376 -0.396 -0.115  0.017 -0.355 -0.39 -0.058  0.209  
  -0.495 -0.004 -0.371 -0.383 -0.045  0.198  0.004 -0.049  0.21  0.224  
  -0.547 -0.525 -0.499 -0.515 -0.393 -0.123 -0.388 -0.417 -0.325 -0.139]]
```

Diese Schreibweise ist zwar etwas lang, aber zum Verständnis Ihrer Modelle oft hilfreich.

## Gittersuche für Vorverarbeitungsschritte und Modellparameter

Mit Pipelines lassen sich sämtliche Verarbeitungsschritte unseres maschinellen Lernprozesses in einem einzigen scikit-learn-Estimator versammeln. Ein nützlicher Nebeneffekt hiervon ist, dass wir das Ergebnis eines überwachten Lernmo-

dells, etwa das einer Regression oder Klassifikation, dazu nutzen können, *die Parameter der Vorverarbeitung anzupassen*. In den vorangegangenen Kapiteln haben wir vor der Ridge-Regression auf dem Datensatz boston polynomielle Merkmale erstellt. Wir werden dies nun mithilfe einer Pipeline wiederholen. Die Pipeline enthält drei Schritte: das Skalieren der Daten, die Berechnung der polynomiellen Merkmale und die Ridge-Regression:

**In[27]:**

```
from sklearn.datasets import load_boston
boston = load_boston()
X_train, X_test, y_train, y_test = train_test_split(boston.data, boston.target,
                                                    random_state=0)
from sklearn.preprocessing import PolynomialFeatures
pipe = make_pipeline(
    StandardScaler(),
    PolynomialFeatures(),
    Ridge())
```

Wie sollen wir entscheiden, welchen Grad die gewählten Polynomialterme haben sollen oder ob wir überhaupt Polynome oder Interaktionen verwenden sollen? Idealerweise sollte der Parameter `degree` entsprechend dem Ergebnis der Klassifikation eingestellt werden. Mit unserer Pipeline können wir die Parameter `degree` und den Parameter `alpha` von `Ridge` gemeinsam durchsuchen. Dazu definieren wir ein `param_grid`, das beide Parameter mit den entsprechenden Namen der Schritte enthält:

**In[28]:**

```
param_grid = {'polynomialfeatures_degree': [1, 2, 3],
              'ridge_alpha': [0.001, 0.01, 0.1, 1, 10, 100]}
```

Nun können wir unsere Gittersuche wiederholen:

**In[29]:**

```
grid = GridSearchCV(pipe, param_grid=param_grid, cv=5, n_jobs=-1)
grid.fit(X_train, y_train)
```

Das Ergebnis der Kreuzvalidierung lässt sich wie in Kapitel 5 als Heatmap visualisieren (Abbildung 6-4):

**In[30]:**

```
plt.matshow(grid.cv_results_['mean_test_score'].reshape(3, -1),
            vmin=0, cmap="viridis")
plt.xlabel("ridge_alpha")
plt.ylabel("polynomialfeatures_degree")
plt.xticks(range(len(param_grid['ridge_alpha'])), param_grid['ridge_alpha']),
plt.yticks(range(len(param_grid['polynomialfeatures_degree'])),
           param_grid['polynomialfeatures_degree'])
plt.colorbar()
```

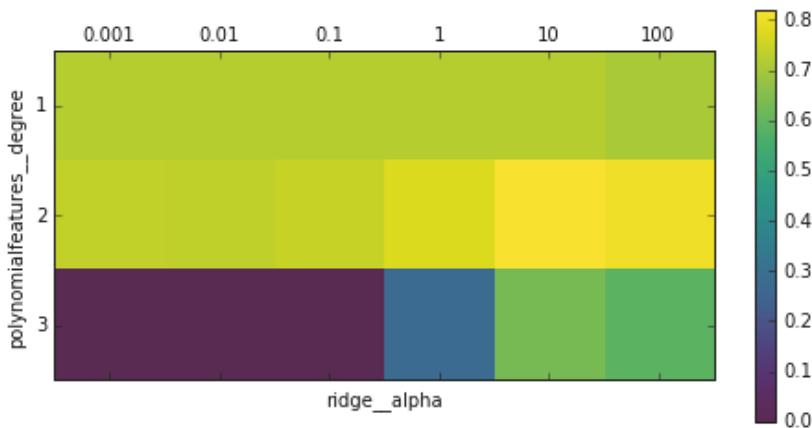


Abbildung 6-4: Heatmap der mittleren Genauigkeit aus der Kreuzvalidierung in Abhängigkeit des Grades der polynomiellen Merkmale und des Parameters alpha der Ridge-Regression

Aus den Ergebnissen der Kreuzvalidierung erkennen wir, dass Polynomialterme 2. Grades hilfreich, Polynome 3. Grades hingegen viel schlechter als die des 1. oder 2. Grades sind. Dies lässt sich auch anhand der besten gefundenen Parameterkombination ablesen:

**In[31]:**

```
print("Beste Parameter: {}".format(grid.best_params_))
```

**Out[31]:**

```
Beste Parameter: {'polynomialfeatures_degree': 2, 'ridge_alpha': 10}
```

Damit erhalten wir folgenden Score:

**In[32]:**

```
print("Genauigkeit auf den Testdaten: {:.2f}".format(grid.score(X_test, y_test)))
```

**Out[32]:**

```
Genauigkeit auf den Testdaten: 0.77
```

Zum Vergleich führen wir eine Gittersuche ohne polynomielle Merkmale durch:

**In[33]:**

```
param_grid = {'ridge_alpha': [0.001, 0.01, 0.1, 1, 10, 100]}
pipe = make_pipeline(StandardScaler(), Ridge())
grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(X_train, y_train)
print("Genauigkeit ohne polynomielle Merkmale: {:.2f}".format(grid.score(X_test, y_test)))
```

**Out[33]:**

```
Genauigkeit ohne polynomielle Merkmale: 0.63
```

Wie wir aus dem Ergebnis der in Abbildung 6-4 visualisierten Gittersuche erwarten würden, führt das Auslassen der polynomiellen Merkmale zu einem deutlich schlechteren Ergebnis.

Eine gemeinsame Suche über Parameter von Vorverarbeitungsschritten und Modellparametern ist eine sehr mächtige Herangehensweise. Bedenken Sie jedoch, dass `GridSearchCV` sämtliche Kombinationen der angegebenen Parameter ausprobiert. Daher führt das Hinzufügen weiterer Parameter zum Gitter zu einem exponentiellen Anstieg der zu konstruierenden Modelle.

## Gittersuche nach dem richtigen Modell

Es sind sogar noch weitere Kombinationen von `GridSearchCV` und Pipeline möglich: Sie können auch die in der Pipeline auszuführenden Schritte durchsuchen (z. B. ob `StandardScaler` oder `MinMaxScaler` verwendet wird). Dies vergrößert den Suchraum noch mehr und sollte mit Vorsicht eingesetzt werden. Alle möglichen Lösungen auszuprobieren, ist beim maschinellen Lernen normalerweise keine Erfolg versprechende Strategie. Hier vergleichen wir dennoch exemplarisch einen `RandomForestClassifier` und einen `SVC` auf dem Datensatz `iris`. Wir wissen, dass für `SVC` möglicherweise skaliert werden muss. Wir probieren also aus, ob der `StandardScaler` oder keine Vorverarbeitung verwendet werden soll. Beim `RandomForestClassifier` wissen wir, dass keine Vorverarbeitung nötig ist. Dazu definieren wir eine Pipeline mit explizit benannten Schritten. Wir benötigen zwei Schritte, zuerst die Vorverarbeitung und anschließend einen Klassifikator. Wir können dazu Instanzen von `SVC` und `StandardScaler` erstellen:

**In[34]:**

```
pipe = Pipeline([('preprocessing', StandardScaler()), ('classifier', SVC())])
```

Nun können wir das zu durchsuchende `parameter_grid` definieren. Wir möchten als `classifier` entweder einen `RandomForestClassifier` oder `SVC` einsetzen. Da beide unterschiedliche Parameter und Vorverarbeitungsschritte benötigen, können wir dies als Liste von Suchgittern wie in Abschnitt »Suche über Räume, die keine Gitter sind« auf Seite 255 angeben. Um einen Estimator einem Schritt zuzuweisen, können wir den Namen des jeweiligen Schrittes als Parameternamen angeben. Wenn wir einen Schritt in der Pipeline ganz überspringen möchten (z. B. weil wir für den `RandomForest` keine Vorverarbeitung benötigen), können wir diesen Schritt auf `None` setzen:

**In[35]:**

```
from sklearn.ensemble import RandomForestClassifier

param_grid = [
    {'classifier': [SVC()], 'preprocessing': [StandardScaler(), None],
     'classifier_gamma': [0.001, 0.01, 0.1, 1, 10, 100],
     'classifier_C': [0.001, 0.01, 0.1, 1, 10, 100]},
```

```
{'classifier': [RandomForestClassifier(n_estimators=100)],  
 'preprocessing': [None], 'classifier_max_features': [1, 2, 3]}
```

Nun können wir wie gewohnt eine Instanz dieser Gittersuche bilden und wenden diese auf den Datensatz cancer an:

**In[36]:**

```
X_train, X_test, y_train, y_test = train_test_split(  
    cancer.data, cancer.target, random_state=0)  
  
grid = GridSearchCV(pipe, param_grid, cv=5)  
grid.fit(X_train, y_train)  
  
print("Beste Parameter:\n{}\n".format(grid.best_params_))  
print("Bester Score aus der Kreuzvalidierung: {:.2f}\n".format(grid.best_score_))  
print("Genauigkeit auf den Testdaten: {:.2f}\n".format(grid.score(X_test, y_test)))
```

**Out[36]:**

```
Beste Parameter:  
{'classifier':  
    SVC(C=10, cache_size=200, class_weight=None, coef0=0.0,  
        decision_function_shape=None, degree=3, gamma=0.01, kernel='rbf',  
        max_iter=-1, probability=False, random_state=None, shrinking=True,  
        tol=0.001, verbose=False),  
    'preprocessing':  
    StandardScaler(copy=True, with_mean=True, with_std=True),  
    'classifier_C': 10, 'classifier_gamma': 0.01}  
  
Bester Score aus der Kreuzvalidierung: 0.99  
Genauigkeit auf den Testdaten: 0.98
```

Als Ergebnis der Gittersuche erhalten wir, dass SVC und StandardScaler mit den Parametern  $C=10$  und  $\gamma=0.01$  das beste Ergebnis liefern.

## Zusammenfassung und Ausblick

In diesem Kapitel haben wir die Klasse Pipeline kennengelernt, ein Mehrzwecktool zum Verketten mehrerer Arbeitsschritte in einem maschinellen Lernprozess. Echte Anwendungen maschinellen Lernens setzen ein Modell nur selten isoliert ein. Stattdessen findet man meist eine Folge von Verarbeitungsschritten. Mit Pipelines lassen sich mehrere Schritte zu einem Python-Objekt bündeln, das uns die von scikit-learn gewohnte Schnittstelle aus `fit`, `predict` und `transform` zur Verfügung stellt. Insbesondere beim Auswerten von Modellen und bei der Parameterauswahl über Gittersuche ist das Verwenden der Klasse Pipeline zum Bündeln sämtlicher Schritte für eine ordentliche Auswertung notwendig. Die Klasse Pipeline verkürzt außerdem den Code und senkt daher die Fehleranfälligkeit gegenüber der Verarbeitung ohne Pipelines (z. B. Fehler wie das Vergessen einer Transformation auf den Testdaten oder das Anwenden der Transformationen in einer anderen Reihenfolge). Aus Merkmalsauswahl, Vorverarbeitung und Modellen die richtige Kombi-

nation auszuwählen, ist eine Kunst. Oft muss man sich nach Versuch und Irrtum vortasten. Mit Pipelines wird dieses »Ausprobieren« vieler Arbeitsschritte aber recht einfach. Beim Herumexperimentieren sollten Sie aufpassen, Ihren Prozess nicht zu kompliziert zu machen, und evaluieren, ob jede in Ihrem Modell eingebundene Komponente wirklich notwendig ist.

Mit diesem Kapitel beenden wir unsere Betrachtung allgemein einsetzbarer Werkzeuge und Algorithmen in scikit-learn. Sie verfügen nun über sämtliche benötigte Fähigkeiten und kennen alle nötigen Mechanismen, um maschinelles Lernen in der Praxis anzuwenden. Im nächsten Kapitel werden wir uns ausführlich einer in der Praxis sehr häufigen Art von Daten widmen, die zur korrekten Verarbeitung besonderes Wissen erfordert: den Textdaten.



# Verarbeiten von Textdaten

In Kapitel 4 haben wir über zwei Arten von Merkmalen gesprochen, die Eigenschaften von Daten repräsentieren können: kontinuierliche Merkmale, die eine Quantität beschreiben, und kategorische Merkmale, die Elemente aus einer festgelegten Liste enthalten. In vielen Anwendungen gibt es eine dritte Art Merkmal, nämlich Text. Möchten wir beispielsweise eine E-Mail als erwünschte Nachricht oder Spam einordnen, enthält der Inhalt der E-Mail mit Sicherheit wichtige Informationen für diese Klassifizierungsaufgabe. Oder uns könnte die Meinung eines Politikers zur Immigration interessieren. Dabei können uns seine Reden oder Tweets von Nutzen sein. Im Kundendienst möchte man häufig wissen, ob eine Nachricht eine Beschwerde oder eine Nachfrage enthält. Wir können die Betreffzeile und den Inhalt einer Nachricht verwenden, um die Absicht des Kunden automatisch zu ermitteln, wodurch wir die Nachricht an die entsprechende Abteilung leiten oder sogar eine vollautomatische Antwort verschicken können.

Textdaten werden üblicherweise als Strings abgelegt, die wiederum aus Zeichen bestehen. In jedem der eben vorgestellten Beispiele variiert die Länge der Textdaten. In dieser Hinsicht unterscheidet sich Text sehr deutlich von den bisher betrachteten numerischen Merkmalen. Daher müssen wir die Daten verarbeiten, bevor wir unsere maschinellen Lernalgorithmen darauf ansetzen können.

## Arten von als Strings repräsentierter Daten

Bevor wir uns die Verarbeitungsschritte zur Repräsentation von Textdaten für maschinelles Lernen ansehen, betrachten wir kurz unterschiedliche Arten von Textdaten. In Ihrem Datensatz ist Text meist nur ein String, aber nicht alle Strings sollten als Text behandelt werden. Ein String-Merkmal kann manchmal kategoriale Variablen repräsentieren, wie wir in Kapitel 5 besprochen haben. Ohne sich die Daten anzusehen, kann man nicht wissen, wie man ein String-Merkmal behandeln sollte.

Es gibt vier Arten von Stringdaten, denen Sie begegnen können:

- Kategorische Daten
- Freie Strings, die sich semantisch Kategorien zuordnen lassen
- Strukturierte Stringdaten
- Textdaten

*Kategorische Daten* sind Daten aus einer festgelegten Liste. Beispielsweise könnten Sie Leute in einer Umfrage nach ihrer Lieblingsfarbe befragen, und diese aus einem Menü »rot,« »grün,« »blau,« »gelb,« »schwarz,« »weiß,« »lila,« oder »pink« auswählen lassen. Dabei erhalten Sie einen Datensatz mit exakt acht möglichen Werten, und damit eindeutig eine kategorische Variable. Sie können dies durch Überfliegen der Daten prüfen (wenn Sie sehr viele unterschiedliche Strings sehen, ist es vermutlich keine kategorische Variable) und durch Berechnen der eindeutigen Werte im Datensatz sowie durch ein Histogramm ihrer Häufigkeiten bestätigen. Sie könnten auch prüfen, ob jede Variable einer für Ihre Anwendung sinnvollen Kategorie entspricht. Zum Beispiel könnte mitten während der Umfrage jemand einen Tippfehler wie »schwaz« statt »schwarz« korrigiert haben. Als Folge davon würde Ihr Datensatz sowohl »schwaz« als auch »schwarz« enthalten. Beides hat aber die gleiche Bedeutung und sollte entsprechend konsolidiert werden.

Nehmen wir an, Sie würden den Nutzern ein Textfeld statt des Menüs zur Eingabe der Lieblingsfarbe präsentieren. Viele Leute würden mit dem Namen einer Farbe wie »schwarz« oder »blau« antworten. Andere würden Tippfehler begehen, unterschiedliche Schreibweisen wie »grün« und »gruen« verwenden oder ausgefallene Namen wie »mitternachtsblau« eingeben. Sie würden auch einige sehr seltsame Einträge erhalten. Einige gute Beispiele finden sich in der xkcd Color Survey (<https://blog.xkcd.com/2010/05/03/color-survey-results/>), bei der Probanden Farben benennen sollten und sich Namen wie »Velociraptor-Kloake« und »orange bei meinem Zahnarzt, ich erinnere mich noch daran, wie seine Schuppen in mein offenes Maul rieselten« ausdachten, die sich nur sehr schwer automatisch Farben zuordnen lassen (oder überhaupt nicht). Die Antworten aus Textfeldern zählen zur zweiten Sorte in der Liste, *freie Strings, die sich semantisch Kategorien zuordnen lassen*. Es wird wohl das Beste sein, solche Daten als kategorische Variablen zu kodieren, bei denen Sie die Kategorien entweder durch Auswahl der häufigsten Werte bestimmen oder Kategorien definieren, die Antworten für Ihre Anwendung sinnvoll zuordnen. Es könnte beispielsweise einige Kategorien für die Standardfarben geben, eventuell eine Kategorie »bunt« für Antworten wie »grün und rot gestreift« und eine Kategorie »andere« für nicht zuordenbare Dinge. Diese Art Vorsortierung von Strings kann eine Menge händische Arbeit erfordern und ist nicht leicht automatisierbar. Wenn Sie die Möglichkeit haben, die Datensammlung zu beeinflussen, empfehlen wir Ihnen, auf von Hand eingegebene Werte zu verzichten, wenn eine kategorische Variable angebrachter ist. Oft korrespondieren von Hand eingegebene Werte nicht mit festen Kategorien, besitzen aber dennoch eine *Struktur* wie Adressen, Namen

von Orten und Menschen, Kalenderdaten, Telefonnummern und andere Identifikatoren. Diese Art von String ist oft sehr schwer zu parsen, und ihre Behandlung hängt sehr stark vom Kontext und vom Fachgebiet ab. Eine systematische Abhandlung dieser Fälle sprengt den Rahmen dieses Buches.

Die letzte Art von Stringdaten ist *Freitext*, der aus Phrasen oder Sätzen besteht. Beispiele dafür sind Tweets, Chatprotokolle und Hotelbewertungen, aber auch die gesammelten Werke von Shakespeare, der Inhalt der Wikipedia oder die im Projekt Gutenberg gesammelten 50000 E-Books. Alle diese Sammlungen enthalten vor allem Informationen in Satzform.<sup>1</sup> Der Einfachheit halber gehen wir davon aus, dass alle unsere Dokumente in der gleichen Sprache verfasst sind.<sup>2</sup> Im Zusammenhang mit der Analyse von Text wird ein Datensatz oft als *Korpus* und jeder durch einen einzelnen Text repräsentierte Datenpunkt als *Dokument* bezeichnet. Diese Begriffe stammen aus den Gebieten *Information Retrieval* (IR) und *Natural Language Processing* (NLP), die sich beide vor allem mit Textdaten befassen.

## Anwendungsbeispiel: Meinungsanalyse zu Filmbewertungen

Als umfassendes Beispiel dieses Kapitels werden wir einen Datensatz von Filmbewertungen aus der vom in Stanford tätigen Forscher Andrew Maas gesammelten IMDb (Internet Movie Database) verwenden.<sup>3</sup> Dieser Datensatz enthält den Text der Bewertungen und eine Angabe darüber, ob die Bewertung positiv oder negativ ist. Auf der IMDb-Webseite selbst finden sich Bewertungen von 1 bis 10. Um die Modellierung zu vereinfachen, fassen wir diese Annotation als Klassifikationsdatensatz mit zwei Kategorien auf, wobei wir Bewertungen von 6 oder höher als positiv und den Rest als negativ einstufen. Wir lassen die Frage offen, ob diese eine gute Repräsentation der Daten ist, und verwenden einfach die Daten von Andrew Maas.

Nach dem Entpacken der Daten erhält der Datensatz zwei Verzeichnisse mit Textdateien, eines als Trainingsdatensatz, das andere als Testdatensatz. Jedes von beiden hat zwei Unterverzeichnisse mit den Namen *pos* und *neg*:

**In[1]:**

```
!tree -L 2 data/aclImdb
```

- 
- 1 Es ist diskutierbar, ob die von Tweets verlinkten Webseiten mehr Information enthalten als die Tweets selbst.
  - 2 Der größte Teil dieses Kapitels funktioniert für alle Sprachen mit lateinischen Buchstaben und teilweise auch für andere Sprachen mit abgegrenzten Wörtern. Mandarin beispielsweise kennt keine Wortgrenzen und bringt noch andere Schwierigkeiten mit sich, die die Anwendung der Techniken in diesem Kapitel erschwert.
  - 3 Der Datensatz ist unter <http://ai.stanford.edu/~amaas/data/sentiment/> verfügbar.

**Out[1]:**

```
data/aclImdb
└── test
    ├── neg
    └── pos
── train
    ├── neg
    ├── pos
    └── unsup
6 directories, 0 files
```

Das Verzeichnis *pos* enthält sämtliche positiven Beurteilungen, jede als separate Textdatei. Im Verzeichnis *neg* ist es genauso. Das Verzeichnis *unsup* enthält nicht gelabelte Daten, die wir nicht verwenden werden und daher löschen:

**In[2]:**

```
!rm -r data/aclImdb/train/unsup
```

In scikit-learn gibt es die Hilfsfunktion `load_files` zum Laden von Dateien aus einer solchen Verzeichnisstruktur, wobei jedes Unterverzeichnis einer Kategorie entspricht. Wir laden zunächst die Trainingsdaten mit der Funktion `load_files`:

**In[3]:**

```
from sklearn.datasets import load_files

reviews_train = load_files("data/aclImdb/train/")
# load_files liefert ein Bunch mit Texten und Kategorien
text_train, y_train = reviews_train.data, reviews_train.target
print("Typ von text_train: {}".format(type(text_train)))
print("Länge von text_train: {}".format(len(text_train)))
print("text_train[1]:\n{}".format(text_train[1]))
```

**Out[3]:**

```
Typ von text_train: <class 'list'>
Länge von text_train: 25000
text_train[1]:
b'Words can\'t describe how bad this movie is. I can\'t explain it by writing
only. You have too see it for yourself to get at grip of how horrible a movie
really can be. Not that I recommend you to do that. There are so many
cliché\xc3\x9as, mistakes (and all other negative things you can imagine) here
that will just make you cry. To start with the technical first, there are a
LOT of mistakes regarding the airplane. I won\'t list them here, but just
mention the coloring of the plane. They didn\'t even manage to show an
airliner in the colors of a fictional airline, but instead used a 747
painted in the original Boeing livery. Very bad. The plot is stupid and has
been done many times before, only much, much better. There are so many
ridiculous moments here that i lost count of it really early. Also, I was on
the bad guys\' side all the time in the movie, because the good guys were so
stupid. "Executive Decision" should without a doubt be you're choice over
this one, even the "Turbulence"-movies are better. In fact, every other
movie in the world is better than this one.'
```

Sie sehen, dass `text_train` eine Liste der Länge 25000 ist, wobei jeder Eintrag ein String mit einer Bewertung ist. Wir haben die Bewertung mit dem Index 1 ausgege-

ben. Sie sehen auch, dass die Bewertung einige HTML-Zeilenumbrüche (<br />) enthält. Auch wenn diese vermutlich keinen großen Einfluss auf unsere maschinellen Lernmodelle haben werden, sollten wir die Daten dennoch säubern und die Umbrüche entfernen, bevor wir fortfahren:

**In[4]:**

```
text_train = [doc.replace(b"<br />", b" ") for doc in text_train]
```

Der Typ der Einträge in text\_train hängt von Ihrer Python-Version ab. In Python 3 ist es der Typ bytes, der eine Binärkodierung von Stringdaten darstellt. In Python 2 enthält text\_train Strings. Wir werden uns an dieser Stelle nicht mit den Details der verschiedenen Stringtypen in Python auseinandersetzen, empfehlen Ihnen aber, sich die Seite the Python 2 (<https://docs.python.org/2/howto/unicode.html>) und/oder Python 3 documentation (<https://docs.python.org/3/howto/unicode.html>) zu Strings und Unicode durchzulesen.

Im Datensatz sind die positiven und negativen Kategorien balanciert, sodass es genauso viele positive wie negative Strings gibt:

**In[5]:**

```
print("Dokumente pro Kategorie (Training): {}".format(np.bincount(y_train)))
```

**Out[5]:**

```
Dokumente pro Kategorie (Training): [12500 12500]
```

Den Testdatensatz laden wir auf die gleiche Weise:

**In[6]:**

```
reviews_test = load_files("data/aclImdb/test/")
text_test, y_test = reviews_test.data, reviews_test.target
print("Anzahl Dokumente in den Testdaten: {}".format(len(text_test)))
print("Dokumente pro Kategorie (Test): {}".format(np.bincount(y_test)))
text_test = [doc.replace(b"<br />", b" ") for doc in text_test]
```

**Out[6]:**

```
Anzahl Dokumente in den Testdaten: 25000
Dokumente pro Kategorie (Test): [12500 12500]
```

Wir möchten die Aufgabe lösen, einer Bewertung mithilfe des Bewertungstextes die Bezeichnung »positiv« oder »negativ« zuzuordnen. Dies ist eine gewöhnliche Klassifizierungsaufgabe. Allerdings befinden sich die Textdaten nicht in einem für maschinelle Lernmodelle geeigneten Format. Wir müssen die Stringrepräsentation in eine numerische Repräsentation überführen, auf die wir unsere maschinellen Lernalgorithmen anwenden können.

## Repräsentation von Text als Bag-of-Words

Eine sehr einfache, aber effektive und häufig eingesetzte Methode zur Repräsentation von Text zum maschinellen Lernen ist der *Bag-of-Words*-Ansatz. Mit dieser

Repräsentation ignorieren wir einen Großteil der Struktur des Eingabetextes wie Kapitel, Absätze, Sätze und Formatierung und zählen lediglich, *wie häufig jedes Wort in jedem Text im Korpus vorkommt*. Das Verwerfen der Struktur und Auszählen von Wörtern ruft das Bild eines »Beutels« als Repräsentation von Text hervor.

Aus einem Korpus von Dokumenten die Repräsentation als Bag-of-Words zu berechnen, beinhaltet die folgenden drei Schritte:

1. *Tokenisierung*. Teile jedes Dokument in die darin enthaltenen Wörter (genannt *Tokens*) auf, beispielsweise durch Trennung bei Leer- und Satzzeichen.
2. *Aufbau des Vokabulars*. Erstelle aus allen in den Dokumenten vorkommenden Wörtern ein Vokabular und nummeriere diese (z. B. in alphabetischer Reihenfolge).
3. *Kodieren*. Zähle in jedem Dokument, wie oft jedes der Wörter im Vokabular im Dokument enthalten ist.

In Schritt 1 und 2 gibt es einige Besonderheiten, die wir später in diesem Kapitel genauer besprechen werden. Sehen wir uns erst einmal an, wie wir den Bag-of-Words-Ansatz in scikit-learn verwenden können. Abbildung 7-1 illustriert diesen Prozess anhand des Strings "This is how you get ants.". Die Ausgabe ist ein Vektor mit Wortanzahlen pro Dokument. Für jedes Wort im Vokabular erhalten wir eine Anzahl, wie oft es in jedem Dokument auftritt. Das bedeutet, unsere numerische Repräsentation enthält ein Merkmal für jedes unterschiedliche Wort im gesamten Datensatz. Beachten Sie, dass die Reihenfolge der Wörter im ursprünglichen String in der Repräsentation als Bag-of-Words völlig irrelevant ist.

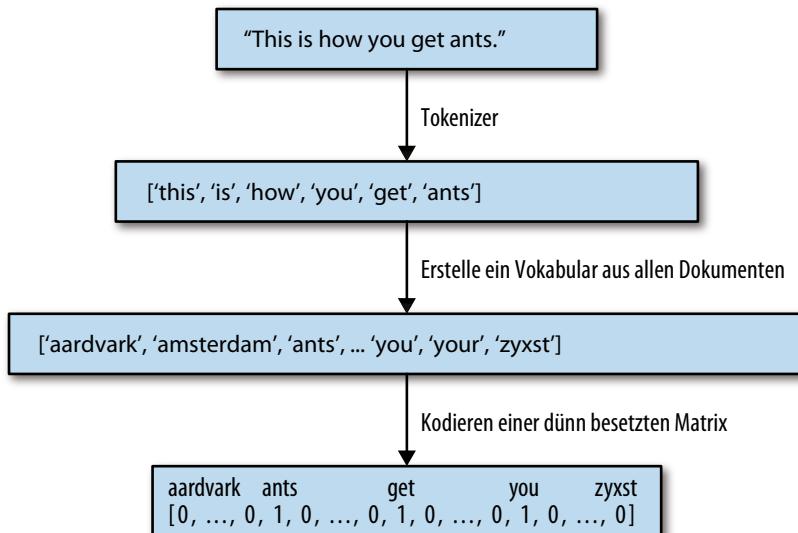


Abbildung 7-1: Verarbeitungsschritte beim Bag-of-Words-Ansatz

## Anwenden von Bag-of-Words auf einen einfachen Datensatz

Die Repräsentation als Bag-of-Words ist im Transformer CountVectorizer implementiert. Wenden wir diesen zunächst auf einen einfachen Datensatz an, um ihn einmal bei der Arbeit zu sehen:

**In[7]:**

```
bards_words = ["The fool doth think he is wise,",  
                "but the wise man knows himself to be a fool"]
```

Wir importieren und instanziieren den CountVectorizer und wenden fit auf unsere Daten wie folgt an:

**In[8]:**

```
from sklearn.feature_extraction.text import CountVectorizer  
vect = CountVectorizer()  
vect.fit(bards_words)
```

Das Anpassen des CountVectorizer besteht aus der Tokenisierung der Trainingsdaten und dem Aufbau des Vokabulars, auf das wir über das Attribut vocabulary\_ zugreifen können:

**In[9]:**

```
print("Größe des Vokabulars: {}".format(len(vect.vocabulary_)))  
print("Inhalt des Vokabulars:\n {}".format(vect.vocabulary_))
```

**Out[9]:**

```
Größe des Vokabulars: 13  
Inhalt des Vokabulars:  
{'the': 9, 'himself': 5, 'wise': 12, 'he': 4, 'doth': 2, 'to': 11, 'knows': 7,  
'man': 8, 'fool': 3, 'is': 6, 'be': 0, 'think': 10, 'but': 1}
```

Das Vokabular enthält 13 Wörter von "be" bis "wise".

Um die Repräsentation als Bag-of-Words aus den Trainingsdaten zu erzeugen, müssen wir die Methode transform aufrufen:

**In[10]:**

```
bag_of_words = vect.transform(bards_words)  
print("bag_of_words: {}".format(repr(bag_of_words)))
```

**Out[10]:**

```
bag_of_words: <2x13 sparse matrix of type '<class 'numpy.int64'>'  
with 16 stored elements in Compressed Sparse Row format>
```

Die Repräsentation als Bag-of-Words wird in einer SciPy sparse matrix gespeichert, die nur Einträge größer null enthält (siehe Kapitel 1). Die Matrix hat die Form  $2 \times 13$ , eine Zeile für jeden der zwei Datenpunkte und ein Merkmal für jedes Wort im Vokabular. Eine dünn besetzte Matrix wird verwendet, weil die meisten Dokumente nur einen kleinen Teil der Wörter im Vokabular verwenden, wodurch

die meisten Einträge im Merkmals-Array 0 sind. Denken Sie einmal daran, wie viele unterschiedliche Wörter in einer Filmbewertung im Vergleich zu allen englischen Wörtern (denn diese bildet das Vokabular ab) auftreten. Alle diese Nullen zu speichern, wäre hinderlich und würde außerdem Speicher vergeuden. Möchten wir uns den Inhalt der dünn besetzten Matrix ansehen, können wir sie mit der Methode `toarray` in ein »dichtes« NumPy-Array überführen (in dem auch die Nullen einge tragen sind):<sup>4</sup>

**In[11]:**

```
print("Dichte Repräsentation von bag_of_words:\n{}".format(
    bag_of_words.toarray()))
```

**Out[11]:**

```
Dichte Repräsentation von bag_of_words:
[[0 0 1 1 1 0 1 0 0 1 1 0 1]
 [1 1 0 1 0 1 0 1 1 1 0 1 1]]
```

Wir sehen, dass die Anzahl für jedes Wort entweder 0 oder 1 beträgt: Keiner der beiden Strings in `bards_words` enthält ein Wort doppelt. Betrachten wir, wie sich diese Merkmalsvektoren interpretieren lassen. Der erste String ("The fool doth think he is wise,") wird durch die erste Zeile repräsentiert und enthält das erste Wort im Vokabular, "be", null Mal. Auch das zweite Wort im Vokabular, "but", ist nicht enthalten. Es enthält das dritte Wort, "doth", einmal usw. Beim Vergleich beider Zeilen erkennen wir, dass das vierte Wort, "fool", das zehnte Wort, "the", und das 13. Wort, "wise", in beiden Strings vorkommt.

## Bag-of-Words der Filmbewertungen

Nachdem wir den Bag-of-Words-Ansatz im Detail gesehen haben, wenden wir ihn auf unsere Meinungsanalyse der Filmbewertungen an. Wir haben unsere Trainings- und Testdaten aus den IMDb-Reviews als Listen von Strings (`text_train` und `text_test`) eingelesen und werden diese nun verarbeiten:

**In[12]:**

```
vect = CountVectorizer().fit(text_train)
X_train = vect.transform(text_train)
print("X_train:\n{}".format(repr(X_train)))
```

**Out[12]:**

```
X_train:
<25000x74849 sparse matrix of type '<class 'numpy.int64'>'>
      with 3431196 stored elements in Compressed Sparse Row format>
```

Die Abmessungen von `X_train`, der Repräsentation der Trainingsdaten als Bag-of-Words, betragen  $25000 \times 74849$ . Das Vokabular enthält also 74849 Einträge. Die

---

<sup>4</sup> Das ist möglich, weil wir ein kleines Übungsbeispiel mit nur 13 Wörtern verwenden. Mit jedem echten Datensatz würden wir einen `MemoryError` erhalten.

Daten sind wieder als dünn besetzte SciPy-Matrix gespeichert. Sehen wir uns das Vokabular etwas genauer an. Mit der Methode `get_feature_name` des Vektorisierers können wir auf das Vokabular als Liste zugreifen, wobei jeder Eintrag einem Merkmal entspricht:

**In[13]:**

```
feature_names = vect.get_feature_names()
print("Anzahl Merkmale: {}".format(len(feature_names)))
print("Erste 20 Merkmale:\n{}".format(feature_names[:20]))
print("Merkmale 20010 bis 20030:\n{}".format(feature_names[20010:20030]))
print("Jedes 2000. Merkmal:\n{}".format(feature_names[::2000]))
```

**Out[13]:**

```
Anzahl Merkmale: 74849
Erste 20 Merkmale:
['00', '000', '000000000001', '00001', '00015', '0005', '001', '003830',
 '006', '007', '0079', '0080', '0083', '0093638', '00am', '00pm', '00s',
 '01', '01pm', '02']
Merkmale 20010 bis 20030:
['dratted', 'draub', 'draught', 'draughts', 'draughtswoman', 'draw', 'drawback',
 'drawbacks', 'drawer', 'drawers', 'drawing', 'drawings', 'drawl',
 'drawled', 'drawling', 'drawn', 'draws', 'draza', 'dre', 'drea']
Jedes 2000. Merkmal:
['00', 'aesir', 'aquarian', 'barking', 'blustering', 'bête', 'chicanery',
 'condensing', 'cunning', 'detox', 'draper', 'enshrined', 'favorit', 'freezer',
 'goldman', 'hasan', 'huitieme', 'intelligible', 'kantrowitz', 'lawful',
 'maars', 'megalunged', 'mostey', 'norrland', 'padilla', 'pincher',
 'promisingly', 'receptionist', 'rivals', 'schnaas', 'shunning', 'sparse',
 'subset', 'temptations', 'treatises', 'unproven', 'walkman', 'xylophonist']
```

Überraschenderweise sind alle der ersten zehn Einträge im Vokabular Zahlen. Diese Zahlen tauchen irgendwo in den Bewertungen auf und werden daher als Wörter extrahiert. Die meisten dieser Zahlen haben keine direkte Bedeutung – außer "007", was im Zusammenhang mit Filmen vermutlich den Charakter James Bond meint.<sup>5</sup> Die bedeutungsvollen von den bedeutungslosen »Wörtern« zu trennen, ist manchmal verzwickt. Sehen wir uns das Vokabular weiter an, finden wir eine Anzahl englischer Wörter, die mit »dra« anfangen. Sie bemerken vielleicht, dass sowohl die Singular- als auch die Pluralform von "draught", "drawback" und "drawer" als einzelne Wörter im Vokabular enthalten sind. Diese Wörter haben sehr eng verwandte Bedeutungen, und sie als unterschiedliche Wörter zu betrachten, ist nicht unbedingt ideal.

Bevor wir versuchen, die Extraktion der Merkmale zu verbessern, sollten wir einen Klassifikator erstellen, um ein Qualitätsmaß zu erhalten. Wir haben die Kategorien für die Trainingsdaten in `y_train` und die Repräsentation als Bag-of-Words in `X_train` abgelegt und können somit einen Klassifikator auf diesen Daten trainieren. Für höher dimensionierte, dünn besetzte Daten wie diese funktionieren lineare Modelle wie die logistische Regression oft am besten.

---

<sup>5</sup> Eine schnelle Analyse der Daten bestätigt diese Vermutung. Probieren Sie es selbst aus.

Zu Beginn werten wir LogisticRegression durch Kreuzvalidierung aus:<sup>6</sup>

In[14]:

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
scores = cross_val_score(LogisticRegression(), X_train, y_train, cv=5)
print("Mittlere Genauigkeit der Kreuzvalidierung: {:.2f}".format(np.mean(scores)))
```

Out[14]:

```
Mittlere Genauigkeit der Kreuzvalidierung: 0.88
```

Wir erhalten für den Score der Kreuzvalidierung einen Mittelwert von 88 %, was für eine balancierte binäre Klassifizierungsaufgabe eine anständige Leistung darstellt. Wir wissen, dass LogisticRegression den Regularisierungsparameter C besitzt, den wir über eine Gittersuche einstellen können:

In[15]:

```
from sklearn.model_selection import GridSearchCV
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10]}
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Bester Score aus der Kreuzvalidierung: {:.2f}".format(grid.best_score_))
print("Bester Parameter: ", grid.best_params_)
```

Out[15]:

```
Bester Score aus der Kreuzvalidierung: 0.89
Bester Parameter: {'C': 0.1}
```

Mit C=0.1 erhalten wir bei der Kreuzvalidierung einen Score von 89 %. Wir können nun die Leistung zum Verallgemeinern dieser Parameter auf dem Testdatensatz prüfen:

In[16]:

```
X_test = vect.transform(text_test)
print("{:.2f}".format(grid.score(X_test, y_test)))
```

Out[16]:

```
0.88
```

Schauen wir nun, ob sich die Extraktion von Wörtern verbessern lässt. CountVectorizer extrahiert Tokens über reguläre Ausdrücke. Normalerweise wird dazu der reguläre Ausdruck "\b\w+\b" verwendet. Falls Sie sich mit regulären Ausdrücken nicht auskennen, dieser erkennt alle Zeichenfolgen, die aus zwei Buchstaben oder Ziffern (\w) bestehen und durch Trennzeichen (\b) begrenzt sind. Der Ausdruck erkennt einbuchstabige Wörter nicht und spaltet Wörter wie »doesn't« oder »bit.

---

<sup>6</sup> Der aufmerksame Leser bemerkt, dass wir die Lektion aus Kapitel 6 über Kreuzvalidierung und das Erstellen von Pipelines außer Acht lassen. Mit den Standardeinstellungen für CountVectorizer werden keine Statistiken erstellt, somit sind unsere Ergebnisse richtig. Für eine echte Anwendung wäre es besser, von Anfang an Pipeline zu verwenden, aber wir lassen es hier der Einfachheit halber weg.

ly« auf, erkennt jedoch »h8ter« als einzelnes Wort. Der CountVectorizer wandelt sämtliche Wörter anschließend in Kleinbuchstaben um, sodass »soon«, »Soon« und »sOon« alle zum gleichen Token (und Merkmal) führen. Dieser einfache Mechanismus ist in der Praxis sehr effektiv, aber wie wir oben sehen, erhalten wir viele nicht informative Merkmale (etwa die Zahlen). Eine Möglichkeit, dem entgegenzuwirken, ist, nur Tokens zu verwenden, die in mindestens zwei Dokumenten vorkommen (oder in fünf oder mehr). Ein Token, das nur in einem Dokument vorkommt, ist wahrscheinlich ohnehin nicht im Testdatensatz enthalten und damit nicht hilfreich. Wir können die minimale Anzahl Dokumente, in denen ein Token auftreten soll, über den Parameter `min_df` festlegen:

**In[17]:**

```
vect = CountVectorizer(min_df=5).fit(text_train)
X_train = vect.transform(text_train)
print("X_train mit min_df: {}".format(repr(X_train)))
```

**Out[17]:**

```
X_train mit min_df: <25000x27271 sparse matrix of type '<class 'numpy.int64'>'  
with 3354014 stored elements in Compressed Sparse Row format>
```

Indem wir mindestens fünf Instanzen eines Tokens voraussetzen, senken wir die Anzahl Merkmale auf 27271 wie in der obigen Ausgabe zu sehen ist – also nur etwa ein Drittel der ursprünglichen Merkmale. Betrachten wir noch einmal einige Tokens:

**In[18]:**

```
feature_names = vect.get_feature_names()

print("Erste 50 Merkmale:\n{}".format(feature_names[:50]))
print("Merkmale 20010 bis 20030:\n{}".format(feature_names[20010:20030]))
print("Jedes 700. Merkmal:\n{}".format(feature_names[::700]))
```

**Out[18]:**

```
Erste 50 Merkmale:  
['00', '000', '007', '00s', '01', '02', '03', '04', '05', '06', '07', '08',  
 '09', '10', '100', '1000', '100th', '101', '102', '103', '104', '105', '107',  
 '108', '10s', '10th', '11', '110', '112', '116', '117', '11th', '12', '120',  
 '12th', '13', '135', '13th', '14', '140', '14th', '15', '150', '15th', '16',  
 '160', '1600', '16mm', '16s', '16th']  
Merkmale 20010 bis 20030:  
['repentance', 'repercussions', 'repertoire', 'repetition', 'repetitions',  
 'repetitious', 'repetitive', 'rephrase', 'replace', 'replaced', 'replacement',  
 'replaces', 'replacing', 'replay', 'replayable', 'replayed', 'replaying',  
 'replays', 'replete', 'replica']  
Jedes 700. Merkmal:  
['00', 'affections', 'appropriately', 'barbra', 'blurbs', 'butchered',  
 'cheese', 'commitment', 'courts', 'deconstructed', 'disgraceful', 'dvds',  
 'eschews', 'fell', 'freezer', 'goriest', 'hauser', 'hungary', 'insinuate',  
 'juggle', 'leering', 'maelstrom', 'messiah', 'music', 'occasional', 'parking',  
 'pleasantville', 'pronunciation', 'recipient', 'reviews', 'sas', 'shea',  
 'sneers', 'steiger', 'swastika', 'thrusting', 'tvs', 'vampyre', 'westerns']
```

Es gibt eindeutig viel weniger Zahlen, und einige obskure Wörter und Tippfehler sind verschwunden. Überprüfen wir, wie das Modell nun bei einer erneuten Gittersuche abschneidet:

**In[19]:**

```
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Bester Score aus der Kreuzvalidierung: {:.2f}".format(grid.best_score_))
```

**Out[19]:**

```
Bester Score aus der Kreuzvalidierung: 0.89
```

Die beste Genauigkeit aus der Validierung durch Gittersuche ist nach wie vor 89 %. Wir haben unser Modell nicht verbessert, aber weniger Merkmale beschleunigen den Ablauf, und das Verwerfen nutzloser Merkmale macht das Modell leichter zu interpretieren.



Wird die Methode `transform` des `CountVectorizer` auf einem Dokument mit nicht in den Trainingsdaten enthaltenen Wörtern aufgerufen, werden diese Wörter ignoriert. Dies stellt für die Klassifikation kein Problem dar, da wir aus Wörtern, die in den Trainingsdaten nicht auftreten, ohnehin nichts lernen können. Für gewisse Einsatzgebiete wie Spamfilterung kann es hilfreich sein, die Anzahl der Wörter »außerhalb des Vokabulars« als eigenes Merkmal zu kodieren. Dazu müssen Sie `min_df` setzen, sonst ist dieses Feature während des Trainings inaktiv.

## Stoppwörter

Eine andere Möglichkeit, nicht informative Wörter loszuwerden, ist das Verwerfen von Wörtern, die zu häufig auftreten, um informativ zu sein. Es gibt dazu zwei Ansätze: eine sprachspezifische Liste von Stoppwörtern verwenden oder sehr häufige Wörter verwerfen. `scikit-learn` enthält im Modul `feature_extraction.text` eine Liste englischer Stoppwörter:

**In[20]:**

```
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS
print("Anzahl Stoppwörter: {}".format(len(ENGLISH_STOP_WORDS)))
print("Jedes 10. Stoppwort:\n{}".format(list(ENGLISH_STOP_WORDS)[:10]))
```

**Out[20]:**

```
Anzahl Stoppwörter: 318
Jedes 10. Stoppwort:
['above', 'elsewhere', 'into', 'well', 'rather', 'fifteen', 'had', 'enough',
'herein', 'should', 'third', 'although', 'more', 'this', 'none', 'seemed',
'nobody', 'seems', 'he', 'also', 'fill', 'anyone', 'anything', 'me', 'the',
'yet', 'go', 'seeming', 'front', 'beforehand', 'forty', 'i']
```

Natürlich senkt das Entfernen der Stopwörter aus der Liste die Anzahl der Merkmale um die Länge der Liste – in diesem Fall 318 –, aber es könnte die Vorhersagequalität verbessern. Überprüfen wir dies:

**In[21]:**

```
# Die Angabe von stop_words="english" verwendet die eingebaute Liste.  
# Wir können auch eine eigene angeben.  
vect = CountVectorizer(min_df=5, stop_words="english").fit(text_train)  
X_train = vect.transform(text_train)  
print("X_train mit Stopwörtern:\n{}".format(repr(X_train)))
```

**Out[21]:**

```
X_train mit Stopwörtern:  
<25000x26966 sparse matrix of type '<class 'numpy.int64'>'  
      with 2149958 stored elements in Compressed Sparse Row format>
```

Es gibt nun 305 Merkmale weniger im Datensatz (27271–26966), die meisten der Stopwörter kamen also vor. Wiederholen wir die Gittersuche:

**In[22]:**

```
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)  
grid.fit(X_train, y_train)  
print("Bester Score aus der Kreuzvalidierung: {:.2f}".format(grid.best_score_))
```

**Out[22]:**

```
Bester Score aus der Kreuzvalidierung: 0.88
```

Die Vorhersagequalität bei der Gittersuche ist mit den Stopwörtern leicht abgefallen – dies ist kein Grund zur Sorge, aber da 305 von 27000 Merkmalen zu verwerfen, vermutlich die Leistung und Interpretierbarkeit nicht besonders ändert, gibt es keinen Grund, diese Liste zu verwenden. Festgelegte Listen sind vor allem bei kleineren Datensätzen hilfreich, da diese nicht genug Information enthalten, mit denen das Modell die Stopwörter selbst erkennen könnte. Als Übung können Sie den zweiten Ansatz ausprobieren, häufig vorkommende Wörter über den Parameter `max_df` im `CountVectorizer` verwerfen und sehen, wie dies die Anzahl der Merkmale und Vorhersagequalität beeinflusst.

## Umskalieren der Daten mit tf-idf

Anstatt als unwichtig eingestufte Merkmale zu verwerfen, können wir alternativ dazu Merkmale entsprechend ihrem geschätzten Informationsgehalt umskalieren. Dabei ist ein verbreitetes Verfahren die *term frequency-inverse document frequency* (tf-idf)-Methode. Die Idee bei dieser Methode ist, Begriffen ein höheres Gewicht zu verleihen, die in *einem* Dokument besonders häufig – aber nicht in vielen Dokumenten im Korpus – vorkommen. Falls ein Wort in einem bestimmten Dokument vorkommt, aber nur in wenigen Dokumenten verwendet wird, ist es möglicherweise ein guter Deskriptor für den Inhalt dieses Dokuments. In `scikit-learn` ist das tf-idf-Verfahren in zwei Klassen implementiert: `TfidfTransformer` transformiert die

vom CountVectorizer erstellte dünn besetzte Matrix, und TfidfVectorizer nimmt sowohl die Bag-of-Words-Extraktion von Merkmalen als auch die tf-idf-Transformation vor. Es gibt mehrere Varianten der Umskalierung durch tf-idf, die auf Wikipedia (<https://en.wikipedia.org/wiki/Tf-idf>) dokumentiert sind. Der tf-idf-Score des Wortes  $w$  im Dokument  $d$  ist entsprechend der Implementierung in den Klassen TfidfTransformer und TfidfVectorizer wie folgt definiert:<sup>7</sup>

$$\text{tfidf}(w, d) = \text{tf} * \log\left(\frac{N + 1}{N_w + 1}\right) + 1$$

Dabei ist  $N$  die Anzahl der Dokumente im Trainingsdatensatz,  $N_w$  ist die Anzahl der Dokumente im Trainingsdatensatz, die das Wort  $w$  enthalten, und  $\text{tf}$  (die Vorkommenshäufigkeit) ist die Anzahl des Wortes  $w$  im Dokument  $d$  (dem zu transformierenden oder zu kodierenden Dokument). Beide Klassen verwenden nach Berechnen der tf-idf-Repräsentation eine L2-Normalisierung; anders gesagt, sie skalieren die Repräsentation jedes Dokuments auf die 1. euklidische Norm um. Das Umskalieren auf diese Weise führt dazu, dass die Länge des Dokuments (die Anzahl der Wörter) keinen Einfluss auf die Darstellung als Vektor hat.

Weil tf-idf sich statistische Eigenschaften der Trainingsdaten zunutze macht, verwenden wir eine Pipeline ähnlich den in Kapitel 6 beschriebenen. Damit stellen wir sicher, dass die Gittersuche zu einem gültigen Ergebnis führt. Damit erhalten wir folgenden Code:

**In[23]:**

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import make_pipeline
pipe = make_pipeline(TfidfVectorizer(min_df=5),
                      LogisticRegression())
param_grid = {'logisticregression__C': [0.001, 0.01, 0.1, 1, 10]}

grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(text_train, y_train)
print("Bester Score aus der Kreuzvalidierung: {:.2f}".format(grid.best_score_))
```

**Out[23]:**

Bester Score aus der Kreuzvalidierung: 0.89

Wie Sie sehen, führt die Verwendung von tf-idf gegenüber der reinen Wortanzahl zu einer leichten Verbesserung. Wir können inspizieren, welche Wörter tf-idf als besonders wichtig eingestuft hat. Bedenken Sie, dass die Skalierung mit tf-idf dazu gedacht ist, Wörter zu finden, mit denen sich Dokumente unterscheiden lassen. Es ist aber eine vollkommen unüberwachte Technik. Daher hat »wichtig« an dieser Stelle nicht unbedingt etwas mit den für uns interessanten Kategorien »positive Bewertung« und

---

<sup>7</sup> Wir geben die Formel hier der Vollständigkeit halber an; Sie müssen sie sich nicht merken, um die tf-idf-Kodierung zu verwenden.

»negative Bewertung« zu tun. Zunächst extrahieren wir den TfidfVectorizer aus der Pipeline:

**In[24]:**

```
vectorizer = grid.best_estimator_.named_steps["tfidfvectorizer"]
# transformiere den Trainingsdatensatz
X_train = vectorizer.transform(text_train)
# finde den größten Wert für jedes Merkmal im Datensatz
max_value = X_train.max(axis=0).toarray().ravel()
sorted_by_tfidf = max_value.argsort()
# generiere Namen der Merkmale
feature_names = np.array(vectorizer.get_feature_names())

print("Merkmale mit niedrigstem tfidf:\n{}".format(
    feature_names[sorted_by_tfidf[:20]]))

print("Merkmale mit höchstem tfidf: \n{}".format(
    feature_names[sorted_by_tfidf[-20:]]))
```

**Out[24]:**

```
Merkmale mit niedrigstem tfidf:
['poignant' 'disagree' 'instantly' 'importantly' 'lacked' 'occurred'
 'currently' 'altogether' 'nearby' 'undoubtedly' 'directs' 'fond' 'stinker'
 'avoided' 'emphasis' 'commented' 'disappoint' 'realizing' 'downhill'
 'inane']

Merkmale mit höchstem tfidf:
['coop' 'homer' 'dillinger' 'hackenstein' 'gadget' 'taker' 'macarthur'
 'vargas' 'jesse' 'basket' 'dominick' 'the' 'victor' 'bridget' 'victoria'
 'khouri' 'zizek' 'rob' 'timon' 'titanic']
```

Merkmale mit niedrigem tf-idf werden entweder sehr häufig in Dokumenten verwendet oder sind selten und nur in sehr langen Dokumenten zu finden. Interessanterweise lassen sich viele der Merkmale mit hohem tf-idf bestimmten Programmen oder Serien zuordnen. Diese Begriffe kommen nur in Bewertungen für diese Programme oder Filme vor, werden in den entsprechenden Bewertungen aber häufig verwendet. Das ist im Fall von Beispielen wie "pokemon", "smallville" und "doodle-bops" verständlich, aber auch "scanners" meint hier den Titel eines Films. Diese Wörter helfen uns vermutlich nicht sehr bei unserer Klassifizierungsaufgabe zu Meinungen (außer eine Serien wird generell positiv oder negativ bewertet), enthalten aber eine Menge Information über die Bewertungen.

Wir beobachten außerdem Wörter mit niedriger inverser Dokumentenfrequenz – also solche, die häufig vorkommen und daher als weniger wichtig erachtet werden. Die Werte für die inverse Dokumentenfrequenz sind für den Trainingsdatensatz im Attribut `idf_` gespeichert:

**In[25]:**

```
sorted_by_idf = np.argsort(vectorizer.idf_)
print("Merkmale mit geringstem idf:\n{}".format(
    feature_names[sorted_by_idf[:100]]))
```

**Out[25]:**

```
Merkmale mit geringstem idf:  
['the' 'and' 'of' 'to' 'this' 'is' 'it' 'in' 'that' 'but' 'for' 'with'  
'was' 'as' 'on' 'movie' 'not' 'have' 'one' 'be' 'film' 'are' 'you' 'all'  
'at' 'an' 'by' 'so' 'from' 'like' 'who' 'they' 'there' 'if' 'his' 'out'  
'just' 'about' 'he' 'or' 'has' 'what' 'some' 'good' 'can' 'more' 'when'  
'time' 'up' 'very' 'even' 'only' 'no' 'would' 'my' 'see' 'really' 'story'  
'which' 'well' 'had' 'me' 'than' 'much' 'their' 'get' 'were' 'other'  
'been' 'do' 'most' 'don' 'her' 'also' 'into' 'first' 'made' 'how' 'great'  
'because' 'will' 'people' 'make' 'way' 'could' 'we' 'bad' 'after' 'any'  
'too' 'then' 'them' 'she' 'watch' 'think' 'acting' 'movies' 'seen' 'its'  
'him']
```

Wie erwartet, enthält die Liste vor allem englische Stoppwörter wie "the" und "no". Es gibt aber auch einige für Filmbewertungen spezifische Wörter wie "movie", "film", "time", "story" usw. Interessanterweise gehören "good", "great" und "bad" nach dem tf-idf-Maß ebenfalls zu den häufigsten und daher »weniger relevanten« Wörtern, obwohl wir erwarten könnten, dass diese für die Meinungsanalyse wichtig sind.

## Untersuchen der Koeffizienten des Modells

Zum Abschluss betrachten wir etwas genauer, was unser logistisches Regressionsmodell aus den Daten gelernt hat. Weil es so viele Merkmale gibt – nach Entfernen der selteneren noch 27271 –, können wir uns unmöglich alle Koeffizienten zeitgleich ansehen. Wir können aber die größten Koeffizienten betrachten und welchen Wörtern diese entsprechen. Wir verwenden das mit den tf-idf-basierten Merkmalen zuletzt trainierte Modell.

Das folgende Balkendiagramm (Abbildung 7-2) zeigt die 25 größten und 25 kleinsten Koeffizienten des logistischen Regressionsmodells, wobei die Balken die Größe des Koeffizienten anzeigen:

**In[26]:**

```
mglearn.tools.visualize_coefficients(  
    grid.best_estimator_.named_steps["logisticregression"].coef_,  
    feature_names, n_top_features=40)
```

Die negativen Koeffizienten auf der linken Seite gehören zu Wörtern, die laut unserem Modell auf negative Bewertungen deuten, während die positiven Koeffizienten auf der rechten Seite zu Wörtern gehören, die auf positive Bewertungen deuten. Die meisten Wörter sind intuitiv verständlich, wie etwa "worst", "waste", "disappointment" und "laughable" für schlechte Bewertungen, während "excellent", "wonderful", "enjoyable" und "refreshing" bei positiven Bewertungen auftreten. Einige Wörter sind etwas weniger klar, wie etwa "bit", "job" und "today", diese könnten Teil von Phrasen wie "good job" oder "best today" sein.

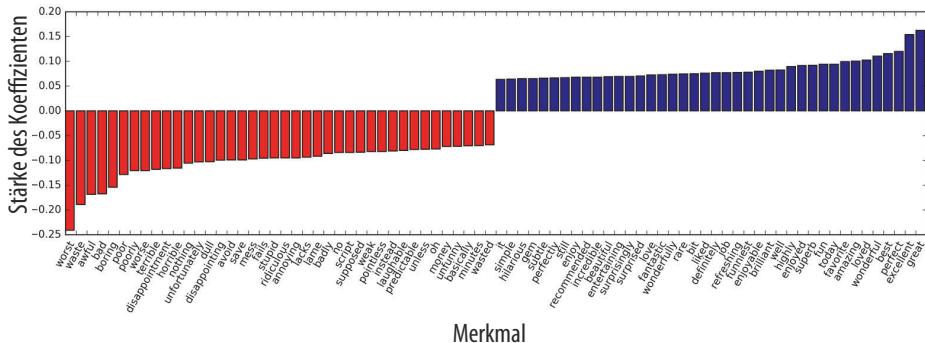


Abbildung 7-2: Größte und kleinste Koeffizienten der auf tf-idf-Merkmalen trainierten logistischen Regression

# Bag-of-Words mit mehr als einem Wort (n-Gramme)

Einer der Hauptnachteile der Repräsentation als Bag-of-Words ist, dass die Reihenfolge der Wörter komplett ignoriert wird. Daher führen die Strings »it's bad, not good at all« und »it's good, not bad at all« zu der exakt gleichen Repräsentation, obwohl ihre Bedeutung umgekehrt ist. »Nicht« vor einem Wort zu plazieren, ist nur ein Beispiel (wenn auch ein extremes) dafür, dass der Kontext eine Rolle spielt. Glücklicherweise lässt sich der Kontext auch mit dem Bag-of-Words-Ansatz erfassen, indem man nicht die Anzahl einzelner Tokens betrachtet, sondern auch Paare oder Tripel aufeinanderfolgender Tokens auszählt. Paare von Tokens nennt man *Bigramme*, Triplett von Tokens *Trigramme*, und allgemein werden Folgen von Tokens als *n-Gramme* bezeichnet. Wir können die Spannbreite betrachteter Tokens über den Parameter `ngram_range` im `CountVectorizer` oder `TfidfVectorizer` ändern. Der Parameter `ngram_range` ist ein Tupel aus der minimalen und maximalen Länge der zu berücksichtigenden Folge von Tokens. Hier folgt ein Beispiel mit dem oben verwendeten Beispieldatensatz:

In[27]:

```
print("bards_words:\n{}".format(bards_words))
```

Out[27]:

```
bards_words:  
['The fool doth think he is wise,',  
 'but the wise man knows himself to be a fool']
```

Mit der Standardeinstellung wird pro Tokenfolge mit der Mindest- und Höchstlänge eins ein Merkmal erzeugt. Die Folgen sind also genau ein Token lang (diese nennt man auch *Unigramme*):

**In[28]:**

```
cv = CountVectorizer(ngram_range=(1, 1)).fit(bards_words)
print("Größe des Vokabulars: {}".format(len(cv.vocabulary_)))
print("Vokabular:\n{}".format(cv.get_feature_names()))
```

**Out[28]:**

```
Größe des Vokabulars: 13
Vokabular:
['be', 'but', 'doth', 'fool', 'he', 'himself', 'is', 'knows', 'man', 'the',
 'think', 'to', 'wise']
```

Um nur die Bigramme zu betrachten – also je zwei aufeinanderfolgende Tokens –, setzen wir `ngram_range` auf (2, 2):

**In[29]:**

```
cv = CountVectorizer(ngram_range=(2, 2)).fit(bards_words)
print("Größe des Vokabulars: {}".format(len(cv.vocabulary_)))
print("Vokabular:\n{}".format(cv.get_feature_names()))
```

**Out[29]:**

```
Größe des Vokabulars: 14
Vokabular:
['be fool', 'but the', 'doth think', 'fool doth', 'he is', 'himself to',
 'is wise', 'knows himself', 'man knows', 'the fool', 'the wise',
 'think he', 'to be', 'wise man']
```

Mit längeren Folgen von Tokens erhalten wir normalerweise mehr Merkmale und genauere Merkmale. Zwischen den beiden Phrasen in `bard_words` gibt es kein gemeinsames Bigramm:

**In[30]:**

```
print("Transformierte Daten (dicht):\n{}".format(cv.transform(bards_words).toarray()))
```

**Out[30]:**

```
Transformierte Daten (dicht):
[[0 0 1 1 0 1 0 0 1 0 1 0 0]
 [1 1 0 0 0 1 0 1 1 0 1 0 1 1]]
```

Bei den meisten Anwendungen sollte die Minimalanzahl von Tokens eins sein, da schon einzelne Wörter meist eine Menge Bedeutung erfassen. In vielen Fällen hilft das Hinzufügen von Bigrammen. Das Hinzufügen längerer Folgen – bis zu 5-Grammen – kann ebenfalls helfen, führt aber auch zu einem explosionsartigen Anstieg der Anzahl Merkmale und begünstigt Overfitting, da es damit sehr viele hochspezifische Merkmale gibt. Im Prinzip könnte die Anzahl Bigramme die quadrierte Anzahl der Unigramme und die Anzahl Trigramme könnte die Anzahl der Unigramme hoch drei sein, womit der Merkmalsraum riesig wird. In der Praxis ist die Anzahl der tatsächlich in den Daten vorkommenden  $n$ -Gramme aufgrund der Struktur der Sprache sehr viel kleiner, aber immer noch groß.

Die Unigramme, Bigramme und Trigramme aus bards\_words sehen wie folgt aus:

In[31]:

```
cv = CountVectorizer(ngram_range=(1, 3)).fit(bards_words)
print("Größe des Vokabulars: {}".format(len(cv.vocabulary_)))
print("Vokabular:\n{}".format(cv.get_feature_names()))
```

Out[31]:

```
Größe des Vokabulars: 39
Vokabular:
['be', 'be fool', 'but', 'but the', 'but the wise', 'doth', 'doth think',
'doth think he', 'fool', 'fool doth', 'fool doth think', 'he', 'he is',
'he is wise', 'himself', 'himself to', 'himself to be', 'is', 'is wise',
'knows', 'knows himself', 'knows himself to', 'man', 'man knows',
'man knows himself', 'the', 'the fool', 'the fool doth', 'the wise',
'the wise man', 'think', 'think he', 'think he is', 'to', 'to be',
'to be fool', 'wise', 'wise man', 'wise man knows']
```

Probieren wir den TfIdfVectorizer auf den IMDb-Filmbewertungen aus und ermitteln wir die optimalen Bereich für  $n$ -Gramme über eine Gittersuche:

In[32]:

```
pipe = make_pipeline(TfidfVectorizer(min_df=5), LogisticRegression())
# wegen des relativ großen Gitters und der Verwendung von Trigrammen
# läuft die Gittersuche sehr lange
param_grid = {"logisticregression__C": [0.001, 0.01, 0.1, 1, 10, 100],
              "tfidfvectorizer__ngram_range": [(1, 1), (1, 2), (1, 3)]}

grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(text_train, y_train)
print("Bester Score aus der Kreuzvalidierung: {:.2f}".format(grid.best_score_))
print("Beste Parameter:\n{}".format(grid.best_params_))
```

Out[32]:

```
Bester Score aus der Kreuzvalidierung: 0.91
Beste Parameter:
{'tfidfvectorizer__ngram_range': (1, 3), 'logisticregression__C': 100}
```

Wie Sie aus dem Ergebnis sehen, haben wir die Vorhersagegüte durch die Berücksichtigung von Bigrammen und Trigrammen um etwas mehr als ein Prozent verbessert. Wir können die Genauigkeit der Kreuzvalidierung als Funktion von ngram\_range und des Parameters C als Heatmap wie in Kapitel 5 darstellen (siehe Abbildung 7-3):

In[33]:

```
# extrahiere Scores aus der Gittersuche
scores = grid.cv_results_['mean_test_score'].reshape(-1, 3).T
# zeichne eine Heatmap
heatmap = mglearn.tools.heatmap(
    scores, xlabel="C", ylabel="ngram_range", cmap="viridis", fmt=".3f",
    xticklabels=param_grid['logisticregression__C'],
    yticklabels=param_grid['tfidfvectorizer__ngram_range'])
plt.colorbar(heatmap)
```

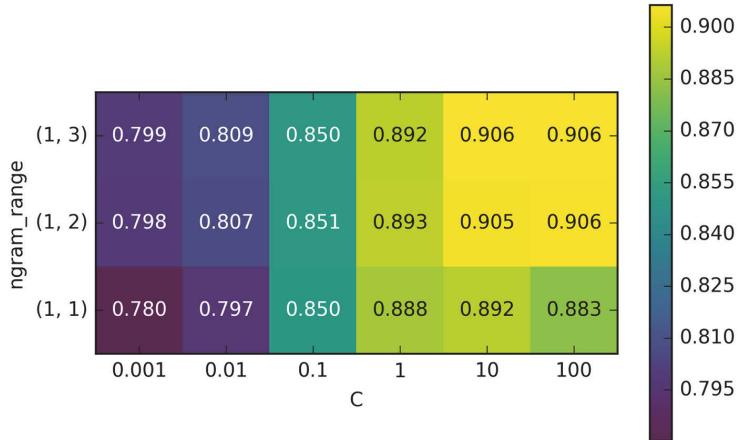


Abbildung 7-3: Darstellung der mittleren Genauigkeit aus der Kreuzvalidierung als Heatmap in Abhängigkeit von den Parametern `ngram_range` und `C`

Aus der Heatmap sehen wir, dass Bigramme die Vorhersagequalität deutlich steigern, während Trigramme nur eine kleine Verbesserung der Genauigkeit bewirken. Um die Verbesserung des Modells besser zu verstehen, können wir die wichtigen Koeffizienten des besten Modells visualisieren, darunter Unigramme, Bigramme und Trigramme (siehe Abbildung 7-4):

In[34]:

```
# extrahiere Namen und Koeffizienten von Merkmalen
vect = grid.best_estimator_.named_steps['tfidfvectorizer']
feature_names = np.array(vect.get_feature_names())
coef = grid.best_estimator_.named_steps['logisticregression'].coef_
mglearn.tools.visualize_coefficients(coef, feature_names, n_top_features=40)
```

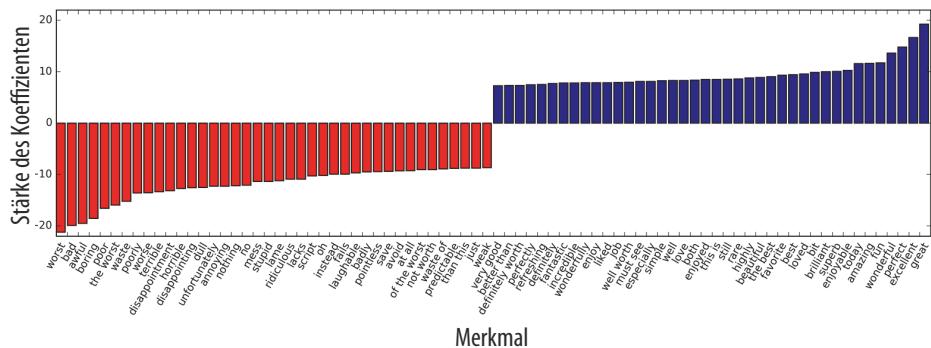


Abbildung 7-4: Wichtigste Merkmale beim Verwenden von Unigrammen, Bigrammen und Trigrammen mit tf-idf-Umskalierung

Es gibt einige besonders interessante Merkmale, die das Wort »worth« enthalten, die es im Unigramm-Modell nicht gab: "not worth" deutet auf eine negative

Bewertung hin, wo hingegen "definitely worth" und "well worth" eine positive Bewertung anzeigen. Dies ist ein ausgezeichnetes Beispiel dafür, wie Kontext die Bedeutung des Wortes "worth" beeinflusst.

Als Nächstes stellen wir nur Trigramme dar, um mehr darüber zu erfahren, warum diese Merkmale hilfreich sind. Viele der nützlichen Bigramme und Trigramme enthalten häufige Wörter, die für sich allein nicht informativ wären, wie in den Phrasen "none of the", "the only good", "on and on", "this is one", "of the most" usw. Allerdings ist der Beitrag dieser Merkmale im Vergleich zur Wichtigkeit der Unigramm-Merkmale gering, wie Sie in Abbildung 7-5 sehen können:

**In[35]:**

```
# finde Trigramm-Merkmale
mask = np.array([len(feature.split(" ")) for feature in feature_names]) == 3
# visualisiere nur Trigramm-Merkmale
mglearn.tools.visualize_coefficients(coef.ravel()[mask],
                                       feature_names[mask], n_top_features=40)
```

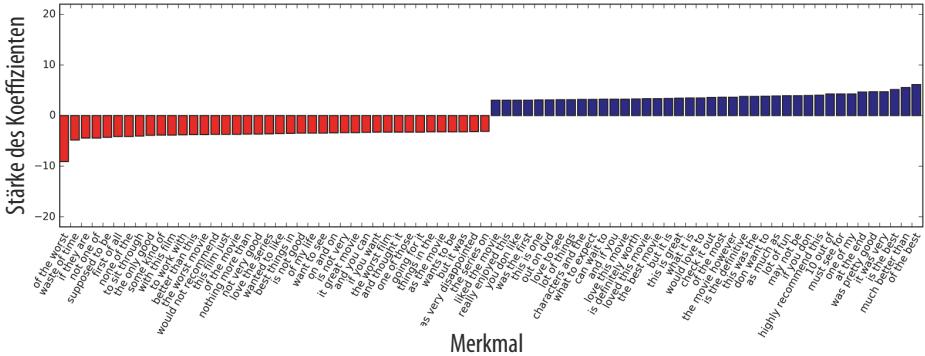


Abbildung 7-5: Darstellung der wichtigsten Trigramm-Merkmale des Modells

## Fortgeschrittene Tokenisierung, Stemming und Lemmatisierung

Wie zuvor erwähnt, ist die Ermittlung der Merkmale durch CountVectorizer und TfIdfVectorizer relativ simpel, und es gibt wesentlich ausgereiferte Methoden. Ein in fortgeschrittenen Anwendungen zur Textprozessierung häufig verbesserter Schritt ist der erste im Bag-of-Words-Ansatz: die Tokenisierung. In diesem Schritt wird definiert, was im Zusammenhang mit der Extraktion von Merkmalen als Wort gilt und was nicht.

Wir haben weiter oben beobachtet, dass das Vokabular häufig die Singular- und Pluralform einiger Wörter enthält, z. B. "drawback" und "drawbacks", "drawer" und "drawers" oder "drawing" und "drawings". Für ein Bag-of-Words-Modell liegen die

Bedeutungen von "drawback" und "drawbacks" so nah beieinander, dass sie zu unterscheiden nur Overfitting begünstigen würde und das Modell die Trainingsdaten nicht voll ausnutzen könnte. Wir haben auch herausgefunden, dass im Vokabular Wörter wie "replace", "replaced", "replacement", "replaces" und "replacing" enthalten sind, die unterschiedliche Verbformen und eine substantivierte Form des Verbs »to replace.« sind. Wie bei Singular und Plural eines Substantivs sollten auch unterschiedliche Verbformen und ähnliche Wörter nicht als separate Tokens behandelt werden, da dies einem verallgemeinernden Modell nicht zuträglich ist.

Dieses Problem lässt sich lösen, indem jedes Wort durch seinen *Wortstamm* repräsentiert wird. Dazu werden alle Wörter mit dem gleichen Wortstamm identifiziert (oder miteinander *verschmolzen*). Wird dies über eine regelbasierte Heuristik gelöst, z. B. das Löschen verbreiteter Endungen, spricht man von *Stemming*. Wenn stattdessen eine Tabelle mit bekannten Wortformen verwendet (ein explizites, von Menschen geprüftes System) und die Rolle eines Wortes im Satz berücksichtigt wird, spricht man von *Lemmatisierung*. Die standardisierte Form eines Wortes bezeichnet man in diesem Falle auch als *Lemma*. Beide Methoden, Lemmatisierung und Stemming, sind Arten von *Normalisierung*, die versuchen, eine normalisierte Form eines Wortes zu ermitteln. Ein weiterer interessanter Fall von Normalisierung ist die Korrektur von Tippfehlern, die in der Praxis nützlich ist, aber den Rahmen dieses Buches sprengt.

Um die Normalisierung besser zu verstehen, vergleichen wir eine Methode zum Stemming – den Porter Stemmer, eine verbreitete Sammlung von Heuristiken (hier aus dem Paket nltk) – mit der im Paket spacy implementierten Lemmatisierung:<sup>8</sup>.

In[36]:

```
import spacy
import nltk

# lade das Modell für englische Sprache aus spacy
en_nlp = spacy.load('en')
# bilde eine Instanz des Porter Stemmer aus nltk
stemmer = nltk.stem.PorterStemmer()

# definiere eine Funktion zum Vergleich der Lemmatisierung in spacy
# mit dem Stemming in nltk
def compare_normalization(doc):
    # tokenisiere ein Dokument in spacy
    doc_spacy = en_nlp(doc)
    # gib von spacy gefundene Lemmata aus
    print("Lemmatisierung:")
    print([token.lemma_ for token in doc_spacy])
    # gib vom Porter stemmer gefundene Token aus
```

---

<sup>8</sup> Details zur Implementierung finden Sie in der Dokumentation zu nltk (<http://www.nltk.org/>) und spacy (<https://spacy.io/docs/>). Hier sind wir mehr an den allgemeinen Prinzipien interessiert.

```
print("Stemming:")
print([stemmer.stem(token.norm_.lower()) for token in doc_spacy])
```

Wir vergleichen die Lemmatisierung mit dem Porter Stemmer anhand eines für diesen Zweck entwickelten Satzes, der die Unterschiede verdeutlicht:

In[37]:

```
compare_normalization(u"Our meeting today was worse than yesterday, "
                      "I'm scared of meeting the clients tomorrow.")
```

Out[37]:

```
Lemmatisation:
['our', 'meeting', 'today', 'be', 'bad', 'than', 'yesterday', ',', 'i', 'be',
 'scared', 'of', 'meet', 'the', 'client', 'tomorrow', '.']
Stemming:
['our', 'meet', 'today', 'wa', 'wors', 'than', 'yesterday', ',', 'i', 'm',
 'scare', 'of', 'meet', 'the', 'client', 'tomorrow', '.']
```

Stemming beschränkt sich immer auf das Abschneiden des Wortes auf einen Wortstamm, sodass "was" zu "wa" umgewandelt wird, während Lemmatisierung die korrekte Grundform, "be", ermitteln kann. In ähnlicher Weise kann Lemmatisierung "worse" zu "bad" umwandeln, während beim Stemming "wors" herauskommt. Ein weiterer Hauptunterschied ist, dass Stemming die beiden Vorkommen von "meeting" zu "meet" reduziert. Bei der Lemmatisierung wird das erste Auftreten von "meeting" als Substantiv erkannt, das zweite als Verb und wird dementsprechend zu "meet" reduziert. Im Allgemeinen ist Lemmatisierung ein viel umständlicherer Prozess als Stemming, führt aber für gewöhnlich zu besseren Ergebnissen beim Normalisieren von Tokens zum maschinellen Lernen.

Auch wenn keine dieser Normalisierungen in scikit-learn implementiert ist, können Sie Ihren eigenen Tokenizer zum Umwandeln von Dokumenten in eine Liste von Tokens im CountVectorizer angeben. Wir können die Lemmatisierung aus spacy verwenden, um eine Funktion zum Umwandeln eines Strings in eine Liste von Lemmata zu schreiben:

In[38]:

```
# Technischer Hinweis: Wir möchten den Regex-basierten Tokenizer
# aus dem CountVectorizer weiterverwenden und nur die
# Lemmatisierung aus spacy übernehmen. Dazu ersetzen wir
# en_nlp.tokenizer (den Tokenizer in spacy)
# mit der Regex-basierten Tokenisierung.
import re
# regulärer Ausdruck in CountVectorizer
regexp = re.compile('(?u)\\b\\w\\w+\\b')

# lade das Sprachmodell aus spacy und
# sichere den alten Tokenizer
en_nlp = spacy.load('en')
old_tokenizer = en_nlp.tokenizer
# ersetze den Tokenizer mit dem regulären Ausdruck
en_nlp.tokenizer = lambda string: old_tokenizer.tokens_from_list()
```

```

    regexp.findall(string))

# erstelle einen eigenen Tokenizer mit der in spacy
# enthaltenen Pipeline (und unserem Tokenizer)
def custom_tokenizer(document):
    doc_spacy = en_nlp(document, entity=False, parse=False)
    return [token.lemma_ for token in doc_spacy]

# erstelle einen CountVectorizer mit den erstellten Tokenizer
lemma_vect = CountVectorizer(tokenizer=custom_tokenizer, min_df=5)

```

Transformieren wir die Daten und inspizieren wir die Größe des Vokabulars:

**In[39]:**

```

# transformiere text_train mit dem CountVectorizer und Lemmatisierung
X_train_lemma = lemma_vect.fit_transform(text_train)
print("X_train_lemma.shape: {}".format(X_train_lemma.shape))

# normaler CountVectorizer zum Vergleich
vect = CountVectorizer(min_df=5).fit(text_train)
X_train = vect.transform(text_train)
print("X_train.shape: {}".format(X_train.shape))

```

**Out[39]:**

```

X_train_lemma.shape: (25000, 21596)
X_train.shape: (25000, 27271)

```

Wie Sie der Ausgabe entnehmen können, wurde die Anzahl Merkmale durch die Lemmatisierung von 27271 (mit dem gewöhnlichen CountVectorizer) auf 21596 gesenkt. Lemmatisierung kann als eine Art Regularisierung angesehen werden, da sie bestimmte Merkmale miteinander verschmilzt. Wir rechnen deshalb damit, dass die Lemmatisierung bei kleinen Datensätzen zu den größten Verbesserungen führt. Um zu illustrieren, wie Lemmatisierung helfen kann, verwenden wir StratifiedShuffleSplit zur Kreuzvalidierung und nehmen nur 1 % der Daten als Trainingsdaten und den Rest als Testdaten:

**In[40]:**

```

# führe eine Gittersuche mit nur 1 % der Daten als Trainingsdatensatz durch
from sklearn.model_selection import StratifiedShuffleSplit

param_grid = {'C': [0.001, 0.01, 0.1, 1, 10]}
cv = StratifiedShuffleSplit(n_iter=5, test_size=0.99,
                           train_size=0.01, random_state=0)
grid = GridSearchCV(LogisticRegression(), param_grid, cv=cv)
# führt eine Gittersuche mit dem gewöhnlichen CountVectorizer durch
grid.fit(X_train, y_train)
print("Bester Score aus der Kreuzvalidierung "
      "(gewöhnlicher CountVectorizer): {:.3f}".format(grid.best_score_))
# führe eine Gittersuche mit Lemmatisierung durch
grid.fit(X_train_lemma, y_train)
print("Bester Score aus der Kreuzvalidierung "
      "(Lemmatisierung): {:.3f}".format(grid.best_score_))

```

**Out[40]:**

```
Bester Score aus der Kreuzvalidierung (gewöhnlicher CountVectorizer): 0.721
Bester Score aus der Kreuzvalidierung (Lemmatisierung): 0.731
```

In diesem Fall hat die Lemmatisierung zu einer bescheidenen Verbesserung der Vorhersagequalität geführt. Wie bei vielen Techniken zum Extrahieren von Merkmalen hängt das Ergebnis vom Datensatz ab. Lemmatisierung und Stemming helfen manchmal, bessere (oder zumindest kompaktere) Modelle zu konstruieren, daher empfehlen wir Ihnen, diese Techniken zum Optimieren der Leistung bei einer konkreten Aufgabe in Betracht zu ziehen.

## Modellierung von Themen und Clustering von Dokumenten

Eine besonders häufig auf Textdaten angewandte Technik ist die *Modellierung von Themen*, ein Oberbegriff für die Aufgabe, jedes Dokument einem oder mehreren *Themen* zuzuordnen, für gewöhnlich unüberwacht. Ein gutes Beispiel hierfür sind Daten zu Nachrichten, die sich in Themengebiete wie »Politik«, »Sport«, »Wirtschaft« usw. kategorisieren lassen. Wenn jedes Dokument einem einzelnen Thema zugeordnet wird, ist dies eine Clustering-Aufgabe, ähnlich denen in Kapitel 3. Wenn jedes Dokument zu mehr als einem Thema gehören kann, gehört die Aufgabe zu den Methoden zur Komponentenzerlegung aus Kapitel 3. Jede der ermittelten Komponenten bildet dann ein Thema, und die Koeffizienten der Komponenten in der Repräsentation eines Dokuments geben an, wie stark dieses Dokument zu einem bestimmten Thema gehört. Wenn man über das Modellieren von Themen spricht, geht es meist um eine bestimmte Methode zur Dekomposition namens *Latent Dirichlet Allocation* (kurz LDA).<sup>9</sup>

### Latent Dirichlet Allocation

Intuitiv versucht das LDA-Modell, Gruppen von Wörtern (die Themen) zu finden, die häufig gemeinsam auftreten. LDA geht außerdem davon aus, dass jedes Dokument als »Mischung« einer Untermenge aller Themen aufgefasst wird. Es ist wichtig zu verstehen, dass für ein maschinelles Lernmodell ein »Thema« nicht das gleiche ist wie in der Alltagssprache, sondern eher den von Hauptkomponentenzerlegung oder NMF ermittelten Komponenten entspricht (welche wir in Kapitel 3 besprochen haben). Diese Themen können eine Bedeutung haben, müssen es aber nicht. Sogar wenn es für ein »Thema« aus der LDA eine Bedeutung gibt, ist es nicht unbedingt das, was wir als Thema bezeichnen würden. Beim Nachrichtenbeispiel

---

<sup>9</sup> Es gibt ein weiteres maschinelles Lernmodell mit der Abkürzung LDA: Lineare Diskriminantenanalyse, ein lineares Modell zur Klassifikation. Dies sorgt für eine Menge Verwirrung. In diesem Buch ist mit LDA stets Latent Dirichlet Allocation gemeint.

könnten wir also eine Sammlung von Artikeln über Sport, Politik und Wirtschaft von zwei bestimmten Journalisten haben. In einem Artikel über Politik würden wir Wörter wie »Ministerpräsident«, »abstimmen«, »Partei« usw. erwarten, in einem Sportartikel dagegen Wörter wie »Mannschaft«, »Punkte« und »Saison«. Wörter aus diesen Gruppen treten wahrscheinlicher gemeinsam auf. Dagegen ist es unwahrscheinlicher, dass »Mannschaft« und »Ministerpräsident« gemeinsam vorkommen. Dies sind allerdings nicht die einzigen Gruppen von Wörtern, die gemeinsam auftreten können. Die zwei Journalisten könnten unterschiedliche Phrasen oder Wortwahl bevorzugen. Beispielsweise könnte der eine gerne das Wort »demarkieren« verwenden, der andere das Wort »polarisieren«. Weitere »Themen« wären dann »von Journalist A häufig verwendete Wörter« und »von Journalist B häufig verwendete Wörter«, auch wenn diese keine Themen im eigentlichen Sinn darstellen.

Wenden wir LDA auf unseren Datensatz von Filmbewertungen an, um sie in Aktion zu sehen. Bei unüberwachten Modellen für Textdokumente ist es meist hilfreich, sehr häufige Wörter zu entfernen, da diese sonst die Analyse dominieren können. Wir entfernen Wörter, die in mindestens 15 Prozent der Dokumente auftreten, und limitieren das Bag-of-Words-Modell auf die nach Entfernen der obersten 15 Prozent häufigsten 10000 Wörter:

**In[41]:**

```
vect = CountVectorizer(max_features=10000, max_df=.15)
X = vect.fit_transform(text_train)
```

Wir trainieren ein Modell mit zehn Themen, was ausreichend wenige sind, sodass wir sie uns alle ansehen können. Ähnlich zu den Komponenten bei der NMF haben die Themen keine inhärente Reihenfolge, und das Ändern der Anzahl von Themen verändert sämtliche Themen.<sup>10</sup> Wir verwenden die Lernmethode "batch", die etwas langsamer ist als die Standardeinstellung ("online"), aber üblicherweise zu besseren Ergebnissen führt. Wir erhöhen außerdem "max\_iter", was ebenfalls das Modell verbessert:

**In[42]:**

```
from sklearn.decomposition import LatentDirichletAllocation
lda = LatentDirichletAllocation(n_topics=10, learning_method="batch",
                                 max_iter=25, random_state=0)
# Wir erstellen das Modell und transformieren die Daten in einem Schritt.
# die Berechnung der Transformation dauert etwas,
# beides in einem Schritt zu tun, spart etwas Zeit
document_topics = lda.fit_transform(X)
```

Wie die Methoden zur Zerlegung in Kapitel 3 besitzt LatentDirichletAllocation das Attribut `components_`, in dem die Wichtigkeit jedes Wortes für jedes Thema abgelegt ist. Die Größe von `components_` ist (`n_topics`, `n_words`):

<sup>10</sup> Tatsächlich lösen NMF und LDA sehr ähnliche Aufgaben, wir könnten Themen auch mit NMF extrahieren.

In[43]:

```
lda.components_.shape
```

Out[43]:

```
(10, 10000)
```

Um die Bedeutung der unterschiedlichen Themen besser zu verstehen, betrachten wir die wichtigsten Wörter für jedes der Themen. Die Funktion `print_topics` formatiert diese Merkmale ansprechend:

In[44]:

```
# Für jedes Thema (eine Zeile in components_),
# sortiere die Merkmale (aufsteigend)
# Kehre die Zeilen mit [:, ::-1] für absteigende Reihenfolge um
sorting = np.argsort(lda.components_, axis=1)[:, ::-1]
# Ermittle die Namen der Merkmale aus dem Vektorisierer
feature_names = np.array(vect.get_feature_names())
```

In[45]:

```
# Gib die 10 Themen aus:
mglearn.tools.print_topics(topics=range(10), feature_names=feature_names,
                           sorting=sorting, topics_per_chunk=5, n_words=10)
```

Out[45]:

topic 0	topic 1	topic 2	topic 3	topic 4
between	war	funny	show	didn
young	world	worst	series	saw
family	us	comedy	episode	am
real	our	thing	tv	thought
performance	american	guy	episodes	years
beautiful	documentary	re	shows	book
work	history	stupid	season	watched
each	new	actually	new	now
both	own	nothing	television	dvd
director	point	want	years	got
topic 5	topic 6	topic 7	topic 8	topic 9
horror	kids	cast	performance	house
action	action	role	role	woman
effects	animation	john	john	gets
budget	game	version	actor	killer
nothing	fun	novel	oscar	girl
original	disney	both	cast	wife
director	children	director	plays	horror
minutes	10	played	jack	young
pretty	kid	performance	joe	goes
doesn	old	mr	performances	around

Den Listen wichtiger Wörter nach zu urteilen, scheint es bei Thema 1 um geschichtliche und Kriegsfilme zu gehen, bei Thema 2 um schlechte Komödien, Thema 3 könnten TV-Serien sein. Thema 4 enthält viele allgemeine Begriffe, Thema 6 scheinen Kinderfilme zu sein, und Thema 8 dreht sich um Bewertungen im Kontext von Auszeichnungen. Mit nur zehn Themen ist jedes Thema zwangsläufig sehr breit, sodass sie gemeinsam alle unterschiedlichen Arten von Bewertungen im Datensatz repräsentieren können.

Als Nächstes werden wir noch ein Modell trainieren, diesmal mit 100 Themen. Mehr Themen erschweren die Analyse deutlich, es wird aber wahrscheinlicher, dass sich die Themen auf interessante Untermengen der Daten spezialisieren:

**In[46]:**

```
lda100 = LatentDirichletAllocation(n_topics=100, learning_method="batch",
                                    max_iter=25, random_state=0)
document_topics100 = lda100.fit_transform(X)
```

Da es etwas zu viel wäre, alle 100 Themen zeitgleich zu betrachten, haben wir einige interessante und repräsentative ausgewählt:

**In[47]:**

```
topics = np.array([7, 16, 24, 25, 28, 36, 37, 45, 51, 53, 54, 63, 89, 97])

sorting = np.argsort(lda100.components_[:, ::-1])
feature_names = np.array(vect.get_feature_names())
mlearn.tools.print_topics(topics=topics, feature_names=feature_names,
                           sorting=sorting, topics_per_chunk=7, n_words=20)
```

**Out[47]:**

topic 7	topic 16	topic 24	topic 25	topic 28
-----	-----	-----	-----	-----
thriller	worst	german	car	beautiful
suspense	awful	hitler	gets	young
horror	boring	nazi	guy	old
atmosphere	horrible	midnight	around	romantic
mystery	stupid	joe	down	between
house	thing	germany	kill	romance
director	terrible	years	goes	wonderful
quite	script	history	killed	heart
bit	nothing	new	going	feel
de	worse	modesty	house	year
performances	waste	cowboy	away	each
dark	pretty	jewish	head	french
twist	minutes	past	take	sweet
hitchcock	didn	kirk	another	boy
tension	actors	young	getting	loved
interesting	actually	spanish	doesn	girl
mysterious	re	enterprise	now	relationship
murder	supposed	von	night	saw
ending	mean	nazis	right	both
creepy	want	spock	woman	simple

topic 36	topic 37	topic 41	topic 45	topic 51
-----	-----	-----	-----	-----
performance	excellent	war	music	earth
role	highly	american	song	space
actor	amazing	world	songs	planet
cast	wonderful	soldiers	rock	superman
play	truly	military	band	alien
actors	superb	army	soundtrack	world
performances	actors	tarzan	singing	evil
played	brilliant	soldier	voice	humans
supporting	recommend	america	singer	aliens
director	quite	country	sing	human
oscar	performance	americans	musical	creatures
roles	performances	during	roll	miike
actress	perfect	men	fan	monsters
excellent	drama	us	metal	apes
screen	without	government	concert	clark
plays	beautiful	jungle	playing	burton
award	human	vietnam	hear	tim
work	moving	ii	fans	outer
playing	world	political	prince	men
gives	recommended	against	especially	moon
topic 53	topic 54	topic 63	topic 89	topic 97
-----	-----	-----	-----	-----
scott	money	funny	dead	didn
gary	budget	comedy	zombie	thought
streisand	actors	laugh	gore	wasn
star	low	jokes	zombies	ending
hart	worst	humor	blood	minutes
lundgren	waste	hilarious	horror	got
dolph	10	laughs	flesh	felt
career	give	fun	minutes	part
sabrina	want	re	body	going
role	nothing	funniest	living	seemed
temple	terrible	laughing	eating	bit
phantom	crap	joke	flick	found
judy	must	few	budget	though
melissa	reviews	moments	head	nothing
zorro	imdb	guy	gory	lot
gets	director	unfunny	evil	saw
barbra	thing	times	shot	long
cast	believe	laughed	low	interesting
short	am	comedies	fulci	few
serial	actually	isn	re	half

Dieses Mal scheinen die extrahierten Themen mehr ins Detail zu gehen, auch wenn einige schwer zu interpretieren sind. In Thema 7 geht es um Horrorfilme und Thriller; Themen 16 und 54 scheinen schlechte Bewertungen zu erfassen, während Thema 63 positive Bewertungen von Komödien enthält. Möchten wir weitere Schlussfolgerungen aus den gefundenen Themen ziehen, sollten wir unsere intuitiven Einsichten aus den Wörterlisten durch genauere Betrachtung der ihnen zugeordneten Dokumente überprüfen. Beispielsweise scheint es bei Thema 45 um Musik zu gehen. Schauen wir einmal, welche Bewertungen diesem Thema zugeordnet sind:

**In[48]:**

```
# sortiere das "Musikthema" 45 nach Gewicht
music = np.argsort(document_topics100[:, 45])[:-1]
# gib die fünf Dokumente aus, in denen das Thema
# am stärksten ausgeprägt ist
for i in music[:10]:
    # gib die ersten zwei Sätze aus
    print(b''.join(text_train[i].split(b'.')[:2]) + b'.\n")
```

**Out[48]:**

```
b'I love this movie and never get tired of watching. The music in it is great.\n'
b"I enjoyed Still Crazy more than any film I have seen in years. A successful
band from the 70's decide to give it another try.\n"
b'Hollywood Hotel was the last movie musical that Busby Berkeley directed for
Warner Bros. His directing style had changed or evolved to the point that
this film does not contain his signature overhead shots or huge production
numbers with thousands of extras.\n'
b"What happens to washed up rock-n-roll stars in the late 1990's?
They launch a comeback / reunion tour. At least, that's what the members of
Strange Fruit, a (fictional) 70's stadium rock group do.\n"
b'As a big-time Prince fan of the last three to four years, I really can't
believe I've only just got round to watching "Purple Rain". The brand new
2-disc anniversary Special Edition led me to buy it.\n'
b"This film is worth seeing alone for Jared Harris' outstanding portrayal
of John Lennon. It doesn't matter that Harris doesn't exactly resemble
Lennon; his mannerisms, expressions, posture, accent and attitude are
pure Lennon.\n"
b"The funky, yet strictly second-tier British glam-rock band Strange Fruit
breaks up at the end of the wild'n'wacky excess-ridden 70's. The individual
band members go their separate ways and uncomfortably settle into lackluster
middle age in the dull and uneventful 90's: morose keyboardist Stephen Rea
winds up penniless and down on his luck, vain, neurotic, pretentious lead
singer Bill Nighy tries (and fails) to pursue a floundering solo career,
paranoid drummer Timothy Spall resides in obscurity on a remote farm so he
can avoid paying a hefty back taxes debt, and surly bass player Jimmy Nail
installs roofs for a living.\n"
b"I just finished reading a book on Anita Loos' work and the photo in TCM
Magazine of MacDonald in her angel costume looked great (impressive wings),
so I thought I'd watch this movie. I'd never heard of the film before, so I
had no preconceived notions about it whatsoever.\n"
b'I love this movie!!! Purple Rain came out the year I was born and it has had
my heart since I can remember. Prince is so tight in this movie.\n'
b"This movie is sort of a Carrie meets Heavy Metal. It's about a highschool
guy who gets picked on a lot and he totally gets revenge with the help of a
Heavy Metal ghost.\n"
```

Wie wir sehen, deckt dieses Thema eine Bandbreite von Bewertungen rund um Musik ab, darunter Musicals, biografische Filme sowie ein schwer zu beschreibendes Genre im letzten Kommentar. Eine andere interessante Perspektive ist, zu inspirieren, wie viel Gewicht jedes Thema insgesamt erhält, indem man `document_topics` über alle Bewertungen aufsummiert. Wir benennen jedes Thema nach den zwei häufigsten Wörtern. Abbildung 7-6 zeigt die erlernten Gewichte:

In[49]:

```
fig, ax = plt.subplots(1, 2, figsize=(10, 10))
topic_names = ["{:>2} ".format(i) + " ".join(words)
               for i, words in enumerate(feature_names[sorting[:, :2]])]
# Balkendiagramm mit zwei Säulen:
for col in [0, 1]:
    start = col * 50
    end = (col + 1) * 50
    ax[col].barh(np.arange(50), np.sum(document_topics100, axis=0)[start:end])
    ax[col].set_yticks(np.arange(50))
    ax[col].set_yticklabels(topic_names[start:end], ha="left", va="top")
    ax[col].invert_yaxis()
    ax[col].set_xlim(0, 2000)
    yax = ax[col].get_yaxis()
    yax.set_tick_params(pad=130)
plt.tight_layout()
```

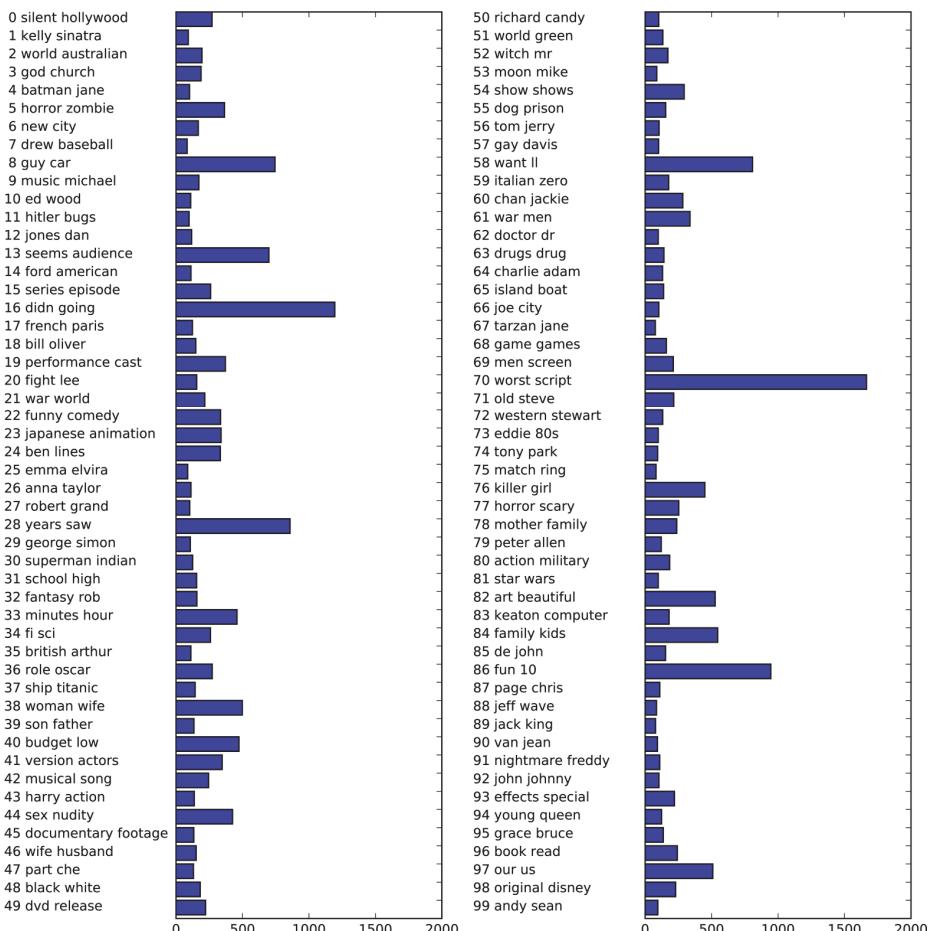


Abbildung 7-6: Durch LDA erlernte Themengewichte

Die wichtigsten Themen sind Thema 97, das vor allem aus Stopwörtern besteht, eventuell mit leicht negativer Tendenz, Thema 16, bei dem es deutlich um schlechte Bewertungen geht, danach die genrespezifischen Themen 36 und 37, die beide lobende Worte enthalten.

Es sieht danach aus, dass LDA vor allem zwei Arten von Themen gefunden hat, genrespezifische und bewertungsspezifische, sowie einige unspezifische. Dies ist eine interessante Beobachtung, da die meisten Bewertungen sowohl filmspezifische Kommentare als auch Kommentare zum Untermauern der Bewertung enthalten.

Themenmodelle wie LDA sind interessante Methoden zum Verständnis großer Textkorpora, für die es keine Klassifizierung gibt – oder sogar wenn wie hier eine Klassifizierung vorliegt. Der LDA-Algorithmus ist jedoch zufallsbasiert, und Ändern des Parameters `random_state` kann zu sehr unterschiedlichen Ergebnissen führen. Auch wenn es hilfreich ist, Themen zu identifizieren, sollte man mit einem unüberwachten Modell gezogene Schlussfolgerungen mit einem Augenzwinkern betrachten, und wir empfehlen Ihnen, Ihren Eindruck durch Betrachten der Dokumente für ein bestimmtes Thema zu verifizieren. Die mit der Methode `LDA.transform` generierte kompakte Repräsentation eignet sich manchmal auch für überwachtes Lernen. Dies erweist sich als besonders hilfreich, wenn nur wenige Trainingsbeispiele verfügbar sind.

## Zusammenfassung und Ausblick

In diesem Kapitel haben wir uns mit den Grundlagen von Textverarbeitung beschäftigt, auch als *Natural Language Processing* (NLP) bekannt, und als beispielhafte Anwendung Filmbewertungen klassifiziert. Die hier besprochenen Werkzeuge dienen als guter Ausgangspunkt beim Verarbeiten von Textdaten. Die Repräsentation als Bag-of-Words ist eine einfache und mächtige Lösung, insbesondere für Klassifizierungsaufgaben wie Erkennung von Spam und Betrug oder zur Meinungsanalyse. Wie so oft beim maschinellen Lernen ist die Repräsentation der Daten auch bei NLP-Anwendungen entscheidend. Das Untersuchen der extrahierten Tokens und *n*-Gramme gibt wichtige Hinweise für den Modellierungsprozess. Bei Anwendungen aus der Textprozessierung ist es oft möglich, die Modelle sinnvoll zu inspirieren, wie wir in diesem Kapitel für sowohl überwachter als auch unüberwachter Aufgaben gesehen haben. Sie sollten sich dies in der Praxis beim Verwenden von NLP-basierten Methoden zunutze machen.

NLP/Sprachtechnologie und Textprozessierung sind ein großes Forschungsfeld, daher sprengt es den Rahmen, fortgeschrittene Methoden im Detail zu beleuchten. Wenn Sie mehr erfahren möchten, empfehlen wir Ihnen das O'Reilly-Buch »Natural Language Processing with Python« (<http://shop.oreilly.com/product/9780596516499.do>) von Steven Bird, Ewan Klein und Edward Loper (ISBN 978-0-596-51649-9), das einen Überblick über NLP mit einer Einführung in das Python-Paket `nltk` verbindet.

Ein weiteres, eher an Konzepten orientiertes Buch ist das Standardwerk »Introduction to Information Retrieval« (<http://nlp.stanford.edu/IR-book/>) von Christopher Manning, Prabhakar Raghavan und Hinrich Schütze (Cambridge University Press, ISBN 978-0-521-86571-5), in dem grundlegende Algorithmen zu Information Retrieval, NLP und maschinellem Lernen beschrieben werden. Zu beiden Büchern gibt es frei erhältliche Online-Versionen. Wie weiter oben besprochen, enthalten die Klassen CountVectorizer und TfidfVectorizer nur relativ einfache Implementierungen von Methoden zur Textprozessierung. Wir empfehlen daher die fortgeschrittenen Methoden aus den Python-Paketen spacy (ein relativ neues, aber sehr effizientes und gut designtes Paket), nltk (eine sehr etablierte und vollständige, aber etwas in die Jahre gekommene Bibliothek) und gensim (ein NLP-Paket mit Fokus auf Themenmodellierung).

Es gibt einige sehr aufregende Entwicklungen jüngeren Datums im Gebiet der Textprozessierung, die außerhalb der Themen dieses Buches liegen und mit neuronalen Netzen zu tun haben. Die erste ist die Verwendung kontinuierlicher Vektoren zu Repräsentationen, die auch als »word vectors« oder »distributed word representations« bezeichnet werden. Diese sind in der Bibliothek word2vec implementiert. Die Originalpublikation »Distributed Representations of Words and Phrases and Their Compositionality« (<http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf>) von Thomas Mikolov u. a. enthält eine großartige Einführung in das Thema. Sowohl spacy als auch gensim enthalten Funktionen für die im Artikel und in Folgepublikationen vorgestellten Techniken.

Ein anderes Teilgebiet von NLP, das in den letzten Jahren Fahrt aufgenommen hat, ist die Verwendung von *Recurrent Neural Networks* (RNNs) zur Textprozessierung. RNNs sind eine besonders mächtige Art neuronaler Netze, die im Gegensatz zu Klassifikationsmodellen als Ausgabe wieder Text produzieren können. Die Fähigkeit, eine Textausgabe zu produzieren, macht RNNs geeignet für automatische Übersetzung und Zusammenfassung. Eine Einführung ins Thema findet sich im eher technischen Artikel »Sequence to Sequence Learning with Neural Networks« (<http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf>) von Ilya Sutskever, Oriol Vinyals und Quoc Le. Eine praktisch orientierte Einführung mit dem Framework tensorflow findet sich auf der TensorFlow-Website (<https://www.tensorflow.org/versions/r0.8/tutorials/seq2seq/index.html>).



# Zusammenfassung und weiterführende Ressourcen

Sie wissen nun, wie Sie die wichtigsten Algorithmen für überwachtes und unüberwachtes maschinelles Lernen anwenden können, um eine Vielzahl von Fragestellungen zu beantworten. Bevor wir Sie verlassen, damit Sie all die Möglichkeiten maschinellen Lernens selbst erkunden können, möchten wir Ihnen einige letzte Ratschläge mitgeben, Sie auf einige zusätzliche Ressourcen hinweisen und Tipps geben, wie Sie Ihre Fähigkeiten zum maschinellen Lernen und Data Science weiter verbessern können.

## Herangehensweise an eine Fragestellung beim maschinellen Lernen

Nun, da Sie all die großartigen in diesem Buch eingeführten Methoden in Griffweite haben, mag die Versuchung groß erscheinen, gleich loszulegen und Ihren Lieblingsalgorithmus auf Ihre bevorzugte datengetriebene Fragestellung loszulassen. Für gewöhnlich ist dies aber kein guter Weg, eine Analyse zu beginnen. Normalerweise ist der Algorithmus zum maschinellen Lernen nur ein kleiner Teil eines größeren Datenanalyse- und Entscheidungsprozesses. Um maschinelles Lernen effektiv einzusetzen, müssen wir zunächst einen Schritt zurück gehen und das übergeordnete Problem betrachten. Erstens sollten Sie darüber nachdenken, welche Art Frage Sie beantworten möchten. Möchten Sie eine erkundende Analyse durchführen und einfach nur schauen, ob die Daten etwas Interessantes enthalten? Oder haben Sie bereits ein bestimmtes Ziel im Auge? Oft beginnt man mit einem Ziel wie dem, betrügerische Transaktionen zu erkennen, Filme zu empfehlen oder unbekannte Planeten zu entdecken. Wenn Sie solch ein Ziel haben, sollten Sie zunächst darüber nachdenken, wie Sie Erfolg für sich definieren und messen und was eine erfolgreiche Umsetzung für Ihr Geschäft oder Ihre Forschung bedeutet, bevor Sie ein System dazu bauen. Sagen wir einmal, Ihr Ziel wäre die Erkennung von Betrug.

Dann eröffnen sich die folgenden Fragen:

- Wie messe ich, ob meine Betrugserkennung überhaupt funktioniert?
- Habe ich die richtigen Daten, um einen Algorithmus zu bewerten?
- Wenn ich damit Erfolg habe, was werden die Auswirkungen der Lösung auf das Geschäft sein?

Wie bereits in Kapitel 5 besprochen, ist es das Beste, wenn Sie die Leistungsfähigkeit eines Algorithmus direkt mithilfe einer Business-Metrik bewerten können, wie zum Beispiel höherer Gewinn oder geringerer Verlust. Allerdings ist das oft schwierig zu erreichen. Eine leichter zu beantwortende Frage wäre »Was, wenn ich ein perfektes Modell bauen könnte?«. Wenn die perfekte Betrugserkennung Ihrer Firma 100 Euro pro Monat an Ausgaben einsparen würde, ist diese Ersparnis vermutlich die Mühe zur Entwicklung eines Algorithmus nicht wert. Wenn das Modell hingegen Ihrer Firma jeden Monat Zehntausende Euro erspart, könnte die Fragestellung eine Untersuchung wert sein.

Nehmen wir an, Sie hätten das zu lösende Problem definiert und wissen, dass eine Lösung Ihr Projekt maßgeblich beeinflussen könnte. Sie haben auch sichergestellt, dass Sie die richtigen Informationen haben, um den Erfolg bewerten zu können. Die nächsten Schritte sind üblicherweise, Daten zu beschaffen und einen funktionierenden Prototyp zu entwickeln. In diesem Buch haben wir über viele anwendbare Modelle gesprochen und darüber, wie Sie diese Modelle angemessen evaluieren und optimieren können. Denken Sie jedoch beim Ausprobieren von Modellen daran, dass dies nur ein kleiner Teil eines größeren Arbeitsablaufes ist. Bei Data Science ist die Modellentwicklung oft Teil einer Feedbackschleife aus Datensammlung, Datenaufbereitung, Modellentwicklung und Modellanalyse. Die Analyse der Fehler, die ein Modell begeht, gibt oft Aufschluss darüber, welche Daten fehlen, welche zusätzlichen Daten erhoben werden könnten oder wie die Aufgabe anders formuliert werden sollte, um das maschinelle Lernen effektiver zu gestalten. Weitere oder andere Daten zu sammeln oder die Fragestellung leicht anders zu formulieren, kann sich weitaus mehr lohnen als endlose Gittersuchen zur Optimierung einzelner Parameter.

## Der menschliche Faktor

Sie sollten sich auch überlegen, ob und wie sich Menschen an der Untersuchung beteiligen. Einige Prozesse (wie die Erkennung von Fußgängern in einem selbstfahrenden Auto) verlangen nach unmittelbaren Entscheidungen. Andere benötigen keine sofortige Antwort, und daher ist es möglich, unsichere Entscheidungen von Menschen bestätigen zu lassen. Medizinische Anwendungen zum Beispiel benötigen eine sehr hohe Genauigkeit, die nicht unbedingt durch maschinelles Lernen allein erreicht werden kann. Wenn aber ein Algorithmus 90 Prozent, 50 Prozent oder vielleicht nur 10 Prozent der Entscheidungen automatisch treffen kann, reicht das unter Umständen aus, um die Antwort zu beschleunigen oder Kosten zu redu-

zieren. Viele Anwendungen werden von den »einfachen Fällen« dominiert, für die ein Algorithmus die Entscheidung vornehmen kann und die wenigen »komplizierteren Fälle« an einen Menschen weiterreicht.

## Vom Prototyp zum Produktivsystem

Die in diesem Buch diskutierten Werkzeuge sind für viele Anwendungen maschinellen Lernens geeignet und erlauben sehr schnelle Analysen und die Entwicklung von Prototypen. Python und scikit-learn werden in vielen Organisationen auch in Produktivsystemen eingesetzt – sogar in sehr großen Unternehmen wie internationalen Banken und globalen Social Media-Unternehmen.

Allerdings weisen viele Unternehmen eine komplexe Infrastruktur auf, sodass es nicht immer einfach ist, Python in diese Systeme einzubinden. In vielen Firmen verwenden die Datenanalyse-Teams Sprachen wie Python und R, mit denen sie Ideen schnell ausprobieren können. Die für die Produktivumgebung zuständigen Teams hingegen verwenden Sprachen wie Go, Scala, C++ und Java, um robuste, skalierbare Systeme zu konstruieren. Für eine Datenanalyse bestehen andere Anforderungen als für die Entwicklung von automatisierten Diensten, und daher ist der Einsatz anderer Sprachen für diesen Zweck gerechtfertigt. Ein verbreiteter Ansatz ist, die vom Analyseteam gefundene Lösung in einer leistungsfähigeren Sprache für die vorhandene Umgebung neu zu implementieren. Dies kann einfacher sein, als eine ganze Bibliothek oder Programmiersprache einzubinden und unterschiedliche Datenformate hin und her zu konvertieren.

Unabhängig davon, ob Sie scikit-learn in einer Produktivumgebung einsetzen können, ist es wichtig, die Unterschiede der Anforderungen im Vergleich zu Skripten zu einmaligen Analysen zu berücksichtigen. Wenn ein Algorithmus in ein größeres System integriert wird, gewinnen Aspekte der Softwareentwicklung wie Zuverlässigkeit, Vorhersagbarkeit, Laufzeit und Speicherverbrauch an Bedeutung. Einfaches Design trägt entscheidend dazu bei, Systeme zum maschinellen Lernen bereitzustellen, die in diesen Bereichen gut abschneiden. Inspizieren Sie jeden Teil Ihrer Datenverarbeitungs- und Vorhersagepipeline kritisch und fragen Sie sich, wie viel Komplexität jeder Schritt mit sich bringt, wie robust jede Komponente gegenüber Änderungen in den Daten oder der rechnerischen Infrastruktur ist und ob der Nutzen jeder Komponente diese Komplexität rechtfertigt. Wenn Sie Systeme konstruieren, die maschinelles Lernen beinhalten, empfehlen wir wärmstens den von Forschern im Team für maschinelles Lernen bei Google veröffentlichten Artikel »Machine Learning: The High Interest Credit Card of Technical Debt« (<http://research.google.com/pubs/pub43146.html>). Der Artikel betont den Zielkonflikt zwischen Aufbau und Pflege von Software für maschinelles Lernen in einer großen Produktivumgebung. Auch wenn das Thema der technischen Verschuldung in großen oder langen Projekten tatsächlich besonders gravierend ist, können die dabei gewonnenen Erkenntnisse uns helfen, auch bei auf kürzere Laufzeit angelegten und kleineren Systemen bessere Software zu entwickeln.

## Testen von Produktivsystemen

In diesem Buch haben wir die Auswertung algorithmischer Vorhersagen mithilfe eines im Voraus festgelegten Testdatensatzes behandelt. Dies ist als *Offline-Evaluation* bekannt. Wenn Ihr System zum maschinellen Lernen von Nutzern direkt verwendet wird, ist dies allerdings nur der erste Schritt zur Evaluation eines Algorithmus. Die nächsten Schritte sind für gewöhnlich *Online-Tests* oder *Live-Tests*, bei denen die Folgen der Anwendung eines Algorithmus im gesamten System ausgewertet werden. Die Änderung der Empfehlungen oder Suchergebnisse, die Benutzer einer Webseite erhalten, kann deren Verhalten drastisch beeinflussen und unvorhergesehene Konsequenzen zur Folge haben. Um sich vor solchen Überraschungen zu schützen, verwenden die meisten Dienste mit Nutzern *A/B-Tests*, eine Art Blindstudie. Bei A/B-Tests erhält ein bestimmter Teil der Nutzer unwissentlich eine bestimmte Variante einer Webseite oder eines Dienstes unter Verwendung von Algorithmus A, während die übrigen Nutzer Algorithmus B verwenden. Für beide Gruppen werden die relevanten Erfolgskriterien für eine bestimmte Zeit aufgezeichnet. Anschließend werden die Metriken für Algorithmus A und B verglichen und anhand der Metriken eine Auswahl zwischen den beiden Verfahren getroffen. Mithilfe von A/B-Tests können wir Algorithmen »in freier Wildbahn« evaluieren, wodurch wir unerwartete Folgen der Interaktion von Nutzern mit unserem Modell entdecken können. Oft ist A das neue Modell, während B das vorhandene System ist. Es gibt weitere, „ausgefultere Mechanismen wie *Bandit-Algorithmen* für Online-Tests die über A/B-Tests hinausgehen. Eine gute Einführung in dieses Thema findet sich im Buch »Bandit Algorithms for Website Optimization« (<http://shop.oreilly.com/product/0636920027393.do>) von John Myles White (O'Reilly, ISBN 978-1-4493-4133-6).

## Konstruieren eines eigenen Estimators

Dieses Buch hat zahlreiche der in `scikit-learn` implementierten Werkzeuge und Algorithmen behandelt, die sich auf viele verschiedene Aufgaben anwenden lassen. Allerdings sind für Ihre Daten in der Praxis meist Verarbeitungsschritte nötig, die in `scikit-learn` nicht implementiert sind. Möglicherweise reicht eine Vorverarbeitung der Daten vor der Übergabe an ein Modell oder eine Pipeline aus `scikit-learn` bereits aus. Wenn Ihre Vorverarbeitung aber von den Daten selbst abhängt und Sie eine Gittersuche oder Kreuzvalidierung durchführen möchten, wird es etwas komplizierter.

In Kapitel 6 haben wir erwähnt, wie wichtig es ist, sämtliche datenabhängige Verarbeitung innerhalb der Schleife zur Kreuzvalidierung zu plazieren. Wie können Sie also Ihre eigene Verarbeitung mit den Werkzeugen in `scikit-learn` kombinieren? Es gibt eine einfache Lösung: Bauen Sie Ihren eigenen Estimator! Es ist recht leicht, einen Estimator zu implementieren, der mit der `scikit-learn`-Schnittstelle kompatibel ist, sodass er sich mit `Pipeline`, `GridSearchCV` und `cross_val_score` verwenden lässt. Sie finden ausführliche Anweisungen dazu in der Dokumentation zu `scikit-`

learn (<http://scikit-learn.org/stable/developers/contributing.html#rolling-your-own-estimator>). Der Grundgedanke dabei ist: Am einfachsten lässt sich eine Transformationsklasse implementieren, indem sie von `BaseEstimator` und `TransformerMixin` erbt und dann die Methoden `init`, `fit` und `predict` etwa folgendermaßen überschreibt:

\*In[1]:\*

```
from sklearn.base import BaseEstimator, TransformerMixin

class MyTransformer(BaseEstimator, TransformerMixin):
    def __init__(self, first_parameter=1, second_parameter=2):
        # Alle Parameter müssen in der Funktion __init__ aufgeführt sein
        self.first_parameter = 1
        self.second_parameter = 2

    def fit(self, X, y=None):
        # fit sollte lediglich die Parameter X und y erhalten
        # sogar wenn Sie ein unbeaufsichtigtes Modell konstruieren,
        # müssen Sie y als Argument zulassen!

        # Code zum Anpassen des Modells gehört hierhin.
        print("das Modell wird nun angepasst")
        # fit gibt self zurück
        return self

    def transform(self, X):
        # transform nimmt X als einzigen Parameter an

        # wende eine Transformation auf X an
        X_transformed = X + 1
        return X_transformed
```

Ein Klassifikator oder Regressor lässt sich auf ähnliche Weise implementieren, nur muss statt von `TransformerMixin` von `ClassifierMixin` oder `RegressorMixin` geerbt werden. Außerdem müssten Sie statt `transform` die Funktion `predict` implementieren.

Wie Sie aus dem hier gezeigten Beispiel sehen, benötigt die Implementierung eines eigenen Estimators recht wenig Code. Die meisten Nutzer von scikit-learn legen mit der Zeit eine Sammlung selbst gebauter Modelle an.

## Wie geht es von hier aus weiter?

Dieses Buch liefert Ihnen eine Einführung ins maschinelle Lernen und wird Sie zu einem effektiven Anwender machen. Wenn Sie jedoch Ihre Fähigkeiten im maschinellen Lernen weiterentwickeln möchten, finden Sie hier einige Buchempfehlungen und spezielle Quellen, um tiefer in das Thema einzutauchen.

## Theorie

In diesem Buch haben wir versucht, eine intuitive Erklärung für die Funktionsweise der verbreitetsten Algorithmen zum maschinellen Lernen zu liefern, ohne dass

Grundlagen der Mathematik oder Informatik nötig sind. Allerdings verwenden viele der vorgestellten Modelle Begriffe aus der Wahrscheinlichkeitstheorie, linearen Algebra und Optimierung. Auch wenn es nicht notwendig ist, sämtliche Details der Implementierung dieser Algorithmen zu kennen, meinen wir, dass etwas theoretisches Wissen zu den Algorithmen Sie zu einem besseren Data Scientist machen wird. Über die Theorie maschinellen Lernens sind viele gute Bücher geschrieben worden. Falls wir Sie für die Möglichkeiten begeistern konnten, die maschinelles Lernen eröffnet, empfehlen wir, sich mit mindestens einem davon genauer zu befassen. Wir haben im Vorwort bereits das Buch von Hastie, Tibshirani und Friedman *The Elements of Statistical Learning* (Springer) erwähnt, aber das darf man an dieser Stelle ruhig noch einmal wiederholen. Ein zweites, gut verständliches Buch mit Python-Code ist *Machine Learning: An Algorithmic Perspective* von Stephen Marsland (Chapman and Hall/CRC). Zwei weitere empfehlenswerte Klassiker sind *Pattern Recognition and Machine Learning* von Christopher Bishop (Springer), ein Buch, das eine probabilistische Sichtweise betont, sowie *Machine Learning: A Probabilistic Perspective* von Kevin Murphy (MIT Press), eine umfassende (über 1000 Seiten) Dissertation zu Methoden maschinellen Lernens mit ausführlichen Betrachtungen hochmoderner Ansätze, die weit über das hinausgehen, was wir in diesem Buch abdecken konnten.

## Andere Umgebungen und Programmpakete zum maschinellen Lernen

Obwohl scikit-learn unser bevorzugtes Programmpaket zum maschinellen Lernen<sup>1</sup> und Python unsere bevorzugte Programmiersprache dazu ist, gibt es eine ganz Reihe weiterer Optionen. Je nach Ihren Bedürfnissen sind Python und scikit-learn in Ihrer Ausgangslage nicht unbedingt die beste Wahl.

Python ist oft zum Ausprobieren und Evaluieren von Modellen bestens geeignet, größere Webdienste und Anwendungen sind jedoch häufiger in Java oder C++ geschrieben, und die Integration mit diesen Systemen könnte zum Ausliefern Ihres Modells notwendig sein. Ein weiterer Grund, andere Technologien als scikit-learn zu erwägen, ist, wenn Sie mehr an statistischer Modellierung und Inferenz als an Vorhersage interessiert sind. In diesem Falle sollten Sie sich das Python-Paket statsmodels näher anschauen, das mehrere lineare Modelle mit einer statistisch orientierten Schnittstelle implementiert. Wenn Sie nicht gerade mit Python verheiratet sind, können Sie auch R, eine weitere lingua franca von Data Scientists, in Betracht ziehen. R ist eine für statistische Analysen entworfene Sprache und für seine ausgezeichneten Fähigkeiten zur Visualisierung sowie für viele verfügbare (oft hochspezialisierte) Pakete zur statistischen Modellierung bekannt.

---

<sup>1</sup> Andreas ist eventuell in dieser Angelegenheit voreingenommen.

Ein weiteres beliebtes Paket zum maschinellen Lernen ist `vowpal wabbit` (zum Vermeiden von Zungenbrechern oft als `vw` bezeichnet), ein hochoptimiertes in C++ implementiertes Programmpaket mit Kommandozeilenschnittstelle. `vw` ist besonders für große Datensätze und gestreamte Daten geeignet. Um Algorithmen zum maschinellen Lernen auf einen Cluster verteilt auszuführen, ist eine der im Moment am weitesten verbreitete Lösung `mllib`, eine Scala-Bibliothek, die auf der verteilten Rechenumgebung `spark` aufbaut.

## Ranking, Empfehlungssysteme und andere Arten von Lernen

Weil dies ein Einführungsbuch ist, haben wir uns auf die geläufigsten Aufgaben beim maschinellen Lernen beschränkt: Klassifikation und Regression beim überwachten Lernen und Clustering und Signalzerlegung beim unüberwachten Lernen. Es gibt viele weitere Arten maschinellen Lernens mit entsprechend bedeutsamen Anwendungen. Zwei davon sind besonders wichtige Themen, die wir in diesem Buch nicht behandelt haben. Die erste ist *Ranking*, bei der wir Antworten zu einer bestimmte Suchanfrage nach Relevanz sortiert erhalten möchten. Sie haben ein Ranking-System vermutlich heute bereits verwendet; Suchmaschinen funktionieren nach diesem Prinzip. Sie geben eine Anfrage ein und erhalten eine nach Relevanz sortierte Ergebnisliste. Eine großartige Einführung ins Ranking findet sich im Buch von Manning, Raghavan und Schütze, *Introduction to Information Retrieval*. Das zweite Thema sind *Empfehlungssysteme*, die Nutzern Vorschläge aufgrund ihrer Präferenzen machen. Sie sind Empfehlungssystemen wahrscheinlich unter Überschriften begegnet wie »Leute, die Sie kennen könnten«, »Kunden, die diesen Artikel gekauft haben, interessieren sich auch für« oder »Für Sie ausgewählt«. Es gibt reichlich Literatur zum Thema, und wenn Sie sich gleich hineinstürzen möchten, könnte Sie die inzwischen klassische »Netflix Prize Challenge« (<http://www.netflixprize.com/>) interessieren, bei der das Videoportal Netflix einen großen Datensatz von Filmpräferenzen veröffentlichte und einen Preis von 1 Million Dollar für das Team ausschrieb, das die besten Empfehlungen geben konnte. Eine weitere verbreitete Anwendung ist die Vorhersage von Zeitreihen (wie etwa Aktienkursen), der ebenfalls ein Stapel Bücher gewidmet ist. Es gibt noch viele weitere Fragestellungen für maschinelles Lernen – viel mehr, als wir hier aufzählen können –, und wir ermutigen Sie, sich Informationen aus Büchern, Forschungsartikeln und Online-Communitys zu suchen, um die am besten zu Ihrer Situation passenden Paradigmen zu finden.

## Probabilistische Modellierung, Inferenz und probabilistische Programmierung

Die meisten Pakete für maschinelles Lernen stellen vordefinierte Modelle bereit, die einen bestimmten Algorithmus ausführen. Viele praktische Anwendungen weisen

jedoch eine Struktur auf, die bei entsprechender Integration in das Modell die Qualität der Vorhersage deutlich verbessert. Häufig lässt sich die Struktur eines bestimmten Problems in der Sprache der Wahrscheinlichkeitstheorie ausdrücken. Diese Struktur leitet sich für gewöhnlich aus einem mathematischen Modell der zu vorhersagenden Situation ab. Um zu veranschaulichen, was wir mit einem strukturierten Problem meinen, betrachten Sie das folgende Beispiel.

Sagen wir, Sie möchten eine mobile Anwendung erstellen, die unter freiem Himmel eine sehr genaue Schätzung der Position liefert, sodass Nutzer sich auf einer historischen Stätte zurechtfinden. Ein Mobiltelefon enthält zahlreiche Sensoren, mit denen Sie präzise Messungen der Position vornehmen können, wie GPS, Beschleunigungssensor und Kompass. Sie haben auch eine genaue Karte des Gebiets. Dieses Problem ist in hohem Maße strukturiert. Sie wissen, wo die Pfade und die interessanten Punkte auf der Karte liegen. Sie haben durch das GPS auch die grobe Position und durch Kompass und Beschleunigungssensor des Geräts sehr genaue relative Messungen. Aber alle diese Daten in ein Black-Box-System zum maschinellen Lernen einzuspeisen, ist nicht unbedingt die beste Idee. Damit würden Sie sämtliche Information über das Geschehen in der wirklichen Welt wieder verlieren. Wenn der Kompass und der Beschleunigungssensor sagen, dass sich der Nutzer nach Norden bewegt, und das GPS sagt, dass er sich nach Süden bewegt, können Sie dem GPS vermutlich nicht über den Weg trauen. Wenn Ihre Schätzung der Position sagt, dass der Nutzer soeben durch eine Mauer gelaufen ist, sollten Sie ebenfalls sehr skeptisch sein. Mit einem probabilistischen Modell ist es hingegen möglich, diese Situation auszudrücken und anschließend durch maschinelles Lernen oder probabilistische Inferenz einzuschätzen, wie sehr man der jeweiligen Messung trauen kann und was die jeweils beste Schätzung der Position eines Nutzers ist.

Haben Sie die Situation und das Zusammenwirken verschiedener Faktoren erst einmal als Modell ausgedrückt, gibt es Methoden, um aus den so zugeschnittenen Modellen direkt Vorhersagen zu berechnen. Die allgemeinsten Methoden zu diesem Zweck sind probabilistische Programmiersprachen. Diese erlauben es, ein Lernproblem auf sehr elegante und kompakte Weise auszudrücken. Beispiele beliebter probabilistischer Sprachen sind PyMC (das in Python verwendbar ist) und Stan (ein in mehreren Programmiersprachen verwendbares Framework, u. a. Python). Auch wenn diese Pakete ein Grundverständnis von Wahrscheinlichkeitstheorie erfordern, erleichtern sie doch die Entwicklung neuer Modelle erheblich.

## Neuronale Netze

Auch wenn wir neuronale Netze als Thema kurz in den Kapiteln 2 und 7 angeschnitten haben, sind diese ein sich rasant entwickelndes Teilgebiet des maschinellen Lernens. Innovationen und neue Anwendungsgebiete werden wöchentlich bekannt. Jüngste Fortschritte beim maschinellen Lernen und künstlicher Intelligenz wie der Sieg des Programms Alpha Go gegen meisterhafte menschliche Spieler im Brettspiel Go, die sich stetig verbesserte Leistung von Spracherkennung und die Verfügbar-

keit von beinahe sofortiger Sprachübersetzung haben von dieser Entwicklung profitiert. Der Fortschritt in diesem Feld ist derart rasant, dass jede Angabe zum technischen Stand schon in kurzer Zeit veraltet sein wird. Das kürzlich erschienene Buch *Deep Learning* von Ian Goodfellow, Yoshua Bengio und Aaron Courville (MIT Press) bietet aber eine reichhaltige Einführung in das Thema.<sup>2</sup>

## Skalieren auf größere Datensätze

In diesem Buch sind wir stets davon ausgegangen, dass unsere Daten in ein NumPy-Array oder eine SciPy Sparse Matrix im Hauptspeicher (RAM) passen. Obwohl der Hauptspeicher moderner Server oft Hunderte Gigabytes (GB) groß ist, stellt dies eine grundsätzliche Obergrenze dar, was den bearbeitbaren Umfang der Daten angeht. Nicht jeder kann sich ein derart teures Gerät leisten oder bei einem Provider in der Cloud mieten. Für die meisten Anwendungen sind die zum Aufbau eines Systems zum maschinellen Lernen verwendeten Daten relativ klein, und nur wenige Datensätze zum maschinellen Lernen umfassen Hunderte Gigabytes an Daten oder mehr. In den meisten Fällen genügt es daher, den Hauptspeicher zu erweitern oder einen Server bei einem Clouddienstleister zu mieten. Falls sich bei Ihnen allerdings Terabytes an Daten auftürmen oder Sie große Datenmengen mit einem begrenzten Budget verarbeiten müssen, gibt es zwei grundlegende Herangehensweisen: *Out-of-Core-Lernen* und *Parallelisierung über einen Cluster*.

Out-of-Core-Lernen beschreibt das Lernen aus Datensätzen, die nicht in den Hauptspeicher passen, aber bei dem das Lernen auf einem einzelnen Computer stattfindet (oder sogar einem einzelnen Prozessor). Die Daten werden von einer Quelle auf der Festplatte oder aus dem Netzwerk gelesen, entweder nur jeweils ein Eintrag oder in Portionen zu mehreren Einträgen, die jeweils in den Speicher passen. Dieser Teildatensatz wird dann verarbeitet, und das Modell wird mit den gelernten Informationen aktualisiert. Anschließend wird die Datenportion verworfen und die nächste gelesen. Out-of-Core-Lernen ist für einige der Modelle in scikit-learn implementiert, und Sie können Details darüber online auf user guide ([http://scikit-learn.org/stable/modules/scaling\\_strategies.html#scaling-with-instances-using-out-of-core-learning](http://scikit-learn.org/stable/modules/scaling_strategies.html#scaling-with-instances-using-out-of-core-learning)) finden.

Weil beim Out-of-Core-Lernen alle Daten von einem einzigen Computer verarbeitet werden müssen, kann die Laufzeit für sehr große Datensätze lang werden. Auch lassen sich nicht alle Algorithmen für maschinelles Lernen auf diese Weise implementieren.

Die zweite Strategie zum Skalieren ist, die Daten auf mehrere Computer in einem Rechnercluster zu verteilen und jeden Computer einen Teil der Daten verarbeiten zu lassen. Für einige Modelle funktioniert dies wesentlich schneller, und die Größe der Daten ist lediglich durch die Größe des Clusters begrenzt. Allerdings erfordern

---

<sup>2</sup> Eine Vorabversion von *Deep Learning* ist unter <http://www.deeplearningbook.org/> einsehbar.

derlei Berechnungen oft eine einigermaßen aufwendige Infrastruktur. Eine der momentan beliebtesten Plattformen für verteiltes Rechnen ist spark, das auf Hadoop aufbaut. spark enthält mit dem Paket `MLlib` einige Funktionalität für maschinelles Lernen. Wenn Ihre Daten sich bereits in einem Hadoop-Filesystem befinden oder Sie bereits spark zur Vorverarbeitung der Daten einsetzen, könnte dies die leichteste Variante sein. Falls Sie über keine eigene Infrastruktur zu diesem Zweck verfügen, könnten der Aufbau und die Integration eines Clusters mit spark zu viel Aufwand bedeuten. Das weiter oben erwähnte Paket `vw` stellt einige Features für verteiltes Rechnen bereit und könnte in diesem Fall die bessere Lösung sein.

## Verfeinern Sie Ihre Fähigkeiten

Wie bei so vielen Dingen im Leben werden Sie nur durch stete praktische Anwendung der in diesem Buch behandelten Themen zum Experten.

Extrahieren von Eigenschaften, Vorverarbeitung, Visualisierung und Konstruieren von Modellen unterscheiden sich erheblich von Aufgabe zu Aufgabe und von Datensatz zu Datensatz. Möglicherweise verfügen Sie ja bereits über mehrere Datensätze und Aufgaben. Falls Sie gerade keine Aufgabe im Sinn haben, sind Wettbewerbe im maschinellen Lernen ein guter Ausgangspunkt. Bei diesen Wettbewerben wird ein Datensatz mit einer gegebenen Aufgabenstellung veröffentlicht, und mehrere Teams versuchen, die bestmöglichen Vorhersagen zu treffen. Viele Unternehmen, gemeinnützige Organisationen und Universitäten richten solche Wettbewerbe aus. Einer der beliebtesten Orte, an denen sie zu finden sind, ist Kaggle (<https://www.kaggle.com/>), eine Webseite, die regelmäßig Wettbewerbe für Data Science abhält, einige davon mit beträchtlichen Preisgeldern.

Die Foren auf Kaggle sind auch eine gute Informationsquelle über die neuesten Werkzeuge und Tricks beim maschinellen Lernen. Auf der Webseite findet sich auch eine große Auswahl Datensätze. Noch mehr Datensätze mit zugehörigen Aufgaben lassen sich auf der Plattform OpenML (<http://www.openml.org/>) finden, die mehr als 20000 Datensätze mit über 50000 Aufgabenstellungen für maschinelles Lernen anbietet. Mit diesen Datensätzen zu arbeiten, ist eine gute Gelegenheit, Ihre Fähigkeiten im maschinellen Lernen zu trainieren. Ein Nachteil von Wettbewerben ist jedoch, dass die zu optimierende Metrik und für gewöhnlich auch ein vorbereiteter Datensatz bereits vorgegeben sind. Denken Sie unbedingt daran, dass die Definition der Fragestellung und das Sammeln der Daten ebenfalls wichtige Aspekte von realen Aufgabenstellungen sind und dass das Problem auf die richtige Art und Weise zu formulieren, sehr viel entscheidender sein kann, als das letzte Prozent Genauigkeit aus einem Klassifikationsverfahren herauszuquetschen.

## **Schlussbemerkung**

Wir hoffen, dass wir Sie von der Nützlichkeit maschinellen Lernens für vielerlei Anwendungsbereiche und der Einfachheit der praktischen Umsetzung überzeugen konnten. Durchwühlen Sie beharrlich Ihre Daten und verlieren Sie dabei das große Ganze nicht aus den Augen.



---

# Index

## A

A/B-Tests 344  
Abmessung, Definition 16  
Abschätzen von Unsicherheit 118  
  Klassifikation mehrerer Kategorien 118  
Abstimmen 37  
agglomeratives Clustering 169, 178, 189  
  Ähnlichkeitsmaße 170  
  Beispiel für 171  
  evaluieren und vergleichen 178  
  Funktionsweise von 169  
  hierarchisches Clustering 172  
algorithm, Parameter 112  
Algorithmen  
  *siehe auch* Modelle; Problemlösung  
  auswerten 30  
  Beispieldatensätze 32  
  minimaler Code zum Anwenden 24  
  skalieren  
    MinMaxScaler 97, 127, 177, 212,  
      292, 303  
    Normalizer 126  
    RobustScaler 125  
    StandardScaler 108, 125, 130, 136,  
      142, 177, 298  
überwacht, Klassifizierung  
  Entscheidungsbäume 68  
  Gradient Boosting 85, 113, 118  
  k-nächste-Nachbarn 36  
  lineare SVMs 56  
  logistische Regression 56  
  naive Bayes 66  
  neuronale Netze 99  
  Random Forests 80  
  Support Vector Machines mit Kernel  
    88

überwacht, Regression  
  Entscheidungsbäume 68  
  Gradient Boosting 85  
  k-nächste-Nachbarn 41  
  Lasso 52  
  lineare Regression (OLS) 47, 204  
  neuronale Netze 99  
  Random Forests 80  
  Ridge 49, 65, 105, 214, 216, 294,  
    300  
unüberwacht, Clustering  
  agglomeratives Clustering 169, 178,  
    189  
  DBSCAN 174  
  k-Means 158  
unüberwacht, Manifold Learning  
  t-SNE 154  
unüberwacht, Signalzerlegung  
  Hauptkomponentenzerlegung 132  
  Nicht-negative-Matrix-Faktorisie-  
    rung 147  
alpha-Parameter bei linearen  
  Modellen 49  
Anaconda 7  
Analyse der Varianz (ANOVA) 218  
angepasster Rand-Index (ARI) 178  
Anwendung Iris-Klassifikation 14  
Datensatz zur 15  
Evaluation des Modells 23  
Inspektion von Daten 19  
k-nächste-Nachbarn 21  
Trainings- und Testdaten 17  
Vorhersagen treffen 22  
Ziele 14  
Arbeitspunkte 273  
Area under the Curve (AUC) 279  
Aufbau des Vokabulars 312

Austreten von Information 294

äußere Knoten 69

## B

Bag-of-Words-Repräsentation

Anwendung auf einen künstlichen  
Datensatz 313

Beispiel Iris-Blüten-Klassifikation

Überblick 23

Beispiel Meinungsanalyse 309

BernoulliNB 66

Bigramme 323

binäre Klassifizierung 27, 55, 261

Binning 204

Blätter 69

Bootstrap-Stichproben 80

Boston Immobilien-Datensatz 35

Bunch-Objekte 34

Business-Metrik 260, 342

## C

cancer, Datensatz 34

Clustering von Dokumenten 331

Clustering-Algorithmen

agglomeratives Clustering 169

auswerten mit Ground Truth 178

DBSCAN 174

evaluieren ohne Referenzdaten 180

k-Means-Clustering 158

Vergleich auf dem Gesichter-Datensatz  
181

Ziele von 158

Zusammenfassung der 192

Cluster-Mittelpunkte 158

Clusterverfahren

Anwendungen von 123

Codebeispiele

herunterladen XII

Nutzungsrechte XII

coef\_, Attribut 47, 50

Core Samples/Kernobjekte 174

cos, Funktion 214

CountVectorizer 313, 318

cross\_val\_score, Funktion 238, 292

## D

datengetriebene Forschung 1

Datenpunkte, Definition 4

Daten-Repräsentation 4

*siehe auch* Merkmale extrahieren und  
generieren; Textdaten

Analogie zu Tabellen 4

Auswirkung auf die Leistung des Modells  
195

automatische Auswahl von Merkmalen  
218

Binning und 204

Daten verstehen 5

in Trainings- und Testdatensätzen 201

Integer-Merkmale 202

kategorische Merkmale 196

Komplexität des Modells vs. Größe des  
Datensatzes 31

Überblick 233

univariate nichtlinear Transformation  
214

Datensätze

mit vielen Dimensionen 33

mit wenigen Dimensionen 33

Datentransformationen 126

*siehe auch* preprocessing

DBSCAN 174

ermittelte Cluster-Zuordnung 178

evaluieren und vergleichen 178

Funktionsweise von 174

Parameter 176

Stärken und Schwächen 174

decision\_function 270

Deep Learning

*siehe* neuronale Netze

Dendrogramme 172

dichte Regionen 174

Dichte-erreichbare Punkte 175

Dimensionsreduktion 134, 147

Diskretisierung 204

Dokumente, Definition 309

dual\_coef\_, Attribut 94

dünn besetzte Datensätze 44

## E

Eigengesichter 139

1-aus-N-Kodierung 197

Empfehlungssysteme 347

Ensembles

Definition 80

Random Forests 80

Regressionsbäume mit Gradient  
Boosting 85

Enthought Canopy 7

Entscheidungsbäume 68

analysieren 73

aufbauen 69

if/else-Struktur von 68  
Kontrollieren der Komplexität von 71  
Parameter 79  
Repräsentation von Daten und 204  
Stärken und Schwächen 79  
vs. Random Forests 80  
Wichtigkeit von Merkmalen bei 74  
Entscheidungsfunktion 113  
Entscheidungsgrenzen 39, 55  
Erkennung von Ausreißern 183  
erste Knoten 69  
Estimator 21  
estimator\_, Attribut des RFECV 82  
Evaluationsmetriken 260, 284  
Auswahl von Metriken 260  
Modellauswahl und 285  
Regressionsmetriken 284  
Testen von Produktivsystemen 344  
zur binären Klassifikation 261  
zur Klassifikation mehrerer Kategorien 282  
exp, Funktion 214  
Expertenwissen 224

## F

f(x)=y, Formel 18  
Faktorenanalyse (FA) 154  
falsch positive/falsch negative Fehler 262  
Falsch-positiv-Rate (FPR) 277  
feature\_names, Attribut 35  
Feed-Forward-Netze 99  
Filmbewertungen 309  
fit\_transform, Methode 130  
fit, Methode 21, 65, 112, 127  
Fließkommazahlen 28  
Folds 236  
forge, Datensatz 32  
freie Stringdaten 308  
Freitext 309

## G

gamma, Parameter 96  
GaussianNB 66  
Gaußscher Kernel für SVC 93, 96  
Genauigkeit 23, 267  
Gesichtserkennung 139, 149  
get\_dummies, Funktion 202  
get\_support, Methode zur Auswahl von Merkmalen 220  
Gewichte 47, 101  
gewöhnliche kleinsten Quadrate (OLS) 46

Gittersuche  
alternative Strategien zur 257  
einfaches Beispiel für 245  
Einsetzen von Pipelines zur 293  
mit Kreuzvalidierung 248  
Modellauswahl mit 303  
parallelisieren mit Kreuzvalidierung 259  
Parameter optimieren mit 244  
Suche über Räume, die keine Gitter sind 255  
Vermeiden von Overfitting 246  
verschachtelte Kreuzvalidierung 257  
Vorverarbeitungspipeline 300  
Zugriff auf Attribute einer Pipeline 299  
graphviz, Modul 73  
GridSearchCV 248, 285, 289, 299, 344  
best\_estimator\_, Attribut 252  
best\_params\_, Attribut 251  
best\_score\_, Attribut 251

## H

Hauptkomponentenzerlegung (PCA)  
Beispiel für 132  
Extraktion von Merkmalen mit 139  
Nachteile von 138  
unüberwachte Eigenschaften von 137  
Visualisierung mit 134  
Whitening 142  
Heatmap 138  
hierarchisches Clustering 172  
Histogramme 136

## I

Inferenz 347  
Information Retrieval (IR) 309  
Integer-Merkmale 202  
»intelligente« Anwendungen 1  
Interaktionen 36, 208  
intercept\_, Attribut 47  
Iris-Klassifikation als Anwendung  
Aufgabe mit mehreren Klassen 28  
iterative Auswahl von Merkmalen 222

## J

Jupyter Notebook 8

## K

Kalibrierung 273  
kategoriale Merkmale 196  
als Zahlen kodiert 202  
Beispiel für 196

- kategorische Daten, Definition 308  
Repräsentation in Trainings- und Testdatensätzen 201  
Repräsentation mittels One-Hot-Kodierung 197  
kategoriale Variablen  
    *siehe* kategoriale Merkmale  
Kernel mit radialer Basisfunktion (RBF) 93  
Klassen 136  
Klassenbezeichnungen 27  
Klassifikation mit mehreren Kategorien 61, 118, 282  
    Abschätzen von Unsicherheit 118  
    lineare Modelle zur 61  
Klassifikaionsaufgaben  
    Beispiel Irisblüten-Klassifikation 14  
    Beispiele für 28  
    binäre vs. mehrere Klassen 27  
    k-nächste-Nachbarn 36  
    lineare Modelle 55  
    naive Bayes-Klassifikatoren 66  
    vs. Regressionsaufgaben 28  
    Ziele für 27  
Klassifikatoren  
    DecisionTreeClassifier 72, 263  
    DecisionTreeRegressor 72, 77  
    KNeighborsClassifier 21, 38  
    KNeighborsRegressor 43  
    LinearSVC 56, 62, 65, 66  
    LogisticRegression 56, 65, 194, 237, 264, 299, 315  
    MLPClassifier 102  
    naive Bayes 66  
    Schätzung der Unsicherheit von 112  
    SVC 56, 96, 126, 131, 245, 253, 258, 289, 297  
k-Means-Clustering 158  
    Anwendung mit scikit-learn 159  
    Beispiel für 158  
    Cluster-Mittelpunkte 158  
    evaluieren und vergleichen 178  
    komplexe Datensätze 168  
    Stärken und Schwächen 169  
    Vektorquantisierung mit 165  
    Versagen von 162  
    vs. Klassifizierung 161  
k-nächste-Nachbarn (k-NN)  
    Analyse des KNeighborsClassifier 39  
    Analyse des KNeighborsRegressor 43  
    Klassifikation 36  
    konstruieren 21  
    Parameter 44  
    Regression 41  
    Stärken und Schwächen 44  
    Vorhersagen mit 36  
    vs. lineare Modelle 46  
knn, Objekt 21  
Kodieren 312  
Koeffizienten mit Textdaten 322  
Komplexität  
    Datensätze 168  
    des Modells vs. Größe des Datensatzes 31  
    kontrollieren 71  
Konfusionsmatrizen 264  
kontinuierliche Merkmale 202  
Korpus 309  
Kreuzvalidierung 236  
    Analyse der Ergebnisse 252  
    Gittersuche und 248  
    Grundprinzip der 236  
    in scikit-learn 237  
    Kreuzvalidierungs-Splitter 240  
    Leave-One-Out Kreuzvalidierung 241  
    mit Gruppen 243  
    Nutzen von 238  
    Parallelisieren mit Gittersuche 259  
    Shuffle-Split Kreuzvalidierung 242  
    stratifizierte k-fache 238  
    verschachtelte 257  
    Vorteile 238
- L**
- L1-Regularisierung 52  
L2-Regularisierung 49, 60, 65  
Lasso-Modell 52  
Latent Dirichlet Allocation (LDA) 331  
learning\_rate, Parameter 86  
Leave-One-Out-Kreuzvalidierung 241  
Lemmatisierung 327  
Lernen aus der Vergangenheit 225  
lineare Funktionen 55  
lineare Modelle 45, 55, 61  
    gewöhnliche kleinste Quadrate 47  
    Klassifikation 55  
    Klassifikation mit mehreren Kategorien 61  
    Lasso 52  
    lineare SVMs 56  
    logistische Regression 56  
    Parameter 65  
    Regression 45  
    Repräsentation von Daten und 204  
    Ridge-Regression 49

- Stärken und Schwächen 65  
 Vorhersagen mit 45  
 vs. k-nächste-Nachbarn 46  
 lineare Regression 47, 204, 208  
 lineare Support Vector Machines (SVMs) 56  
 Linkage Arrays 172  
 Live-Tests 344  
 log, Funktion 215
- M**
- make\_pipeline, Funktion  
   Anzeigen des Attributs steps 298  
   Pipeline mit Gittersuche und 299  
   Syntax 297  
   Zugriff auf Attribute von Schritten 298
- Manifold Learning-Algorithmus  
   Anwendungsbeispiele für 154  
   Beispiele für 155  
   Ergebnisse von 157  
   Visualisierung mit 154
- manuell kodierte Regeln, Nachteile von 1
- maschinelles Lernen  
   Anwendungen 1  
   Beispiele für 1, 14  
   Daten verstehen 5  
   Evaluierung und Verbesserung von  
     Modellen 235  
   Herangehensweise an eine Fragestellung  
     341  
   Konstruieren eigener Systeme IX  
   Lernvoraussetzungen IX  
   Mathematische Grundlagen IX  
   Ressourcen XI, 345  
   scikit-learn und 6  
   überwachtes Lernen 27  
   unüberwachtes Lernen 123  
   Verarbeiten von Textdaten 307  
   verkettete Algorithmen und Pipelines  
     289  
   Vorteile von Python für 5  
   Vorverarbeiten und Skalieren 124
- mathematische Funktionen zur Transfor-  
   mation von Merkmalen 214
- matplotlib 10
- max\_features, Parameter 81
- mehrschichtige Perzeptrons (MLPs) 99
- menschliche Beteiligung/Aufsicht 342
- Merkmale extrahieren und generieren  
   *siehe auch* Repräsentation von Daten;  
     Textdaten  
   automatische Auswahl von Merkmalen  
     218
- Definition 4, 36, 195  
 interagierende Merkmale 208  
 kategorische Merkmale 196  
 mit Nicht-negativer-Matrix-Faktori-  
   sierung 147  
 mittels Hauptkomponentenzerlegung  
     139
- polynomiale Merkmale 208  
 Überblick 233  
 univariate nichtlineare Transformation  
     214
- Verbessern von Daten durch 195  
 Verwenden von Expertenwissen 224
- Merkmale, Definition 4
- Meta-Estimatoren für Bäume und Wälder  
     251
- Method Chaining 65
- mglearn 11
- mittlere Relevanz 277
- mllib 347
- modellbasierte Auswahl von Merkmalen  
     221
- Modelle  
   *siehe auch* Algorithmen  
   Anwendung Iris-Klassifikation 14  
   Auswahl 285  
   Auswahl durch Gittersuche 303  
   Auswirkung der Repräsentation von  
     Daten auf 195  
   Evaluationsmetriken 260  
   Evaluierung und Verbesserung 235  
   kalibrierte 273  
   Koeffizienten mit Textdaten 322  
   Komplexität vs. Größe des Datensatzes  
     31  
   Kreuzvalidierung von 236  
   Overfitting vs. Underfitting 30  
   Parameter über Gittersuche optimieren  
     244  
   Theorie dahinter 345  
   verallgemeinernde 28  
   Vorverarbeitungspipeline und 300
- Modellierung von Themen, mit LDA 331
- MultinomialNB 66
- N**
- naive Bayes-Klassifikatoren 66  
   Arten in scikit-learn 66  
   Parameter 67  
   Stärken und Schwächen 68
- Natural Language Processing (NLP) 309,  
     338

- negative Klasse 28  
Netflix Prize Challenge 347  
neuronale Netze (Deep Learning) 99  
Abschätzen der Komplexität bei 111  
Feineinstellung 102  
Genauigkeit von 108  
jüngste Fortschritte bei 348  
Randomisierung bei 106  
Stärken und Schwächen 111  
Vorhersage mit 99
- n-Gramme 323  
nicht balancierte Datensätze 262  
Nicht-negative-Matrix-Faktorisierung (NMF)  
Anwendung auf Bilder von Gesichtern 149  
Anwendung auf künstliche Daten 148  
Anwendungen für 147
- normalisierte gegenseitige Information (NMI) 178
- Normalisierung 328  
NumPy (Numeric Python), Bibliothek 8  
Nutzungsrechte XII
- O**
- Offline-Evaluation 344  
One-Hot-Kodierung 197  
One-vs.-Rest-Ansatz 61  
Online-Tests 344  
OpenML, Plattform 350  
Out-of-Core-Lernen 349  
Overfitting 30, 246
- P**
- Paarplots 19  
pandas  
Daten in One-Hot-Kodierung umwandeln 198  
get\_dummies, Funktion 202  
Prüfen von String-kodierten Daten 198  
Spaltenindizierung in 200  
Vorteile 10  
Parallelisierung auf Clustern 349  
Parameter C bei SVC 95  
Pipelines  
*siehe* verkettete Algorithmen und Pipelines  
polynomiale Kernel 93  
polynomiale Merkmale 208  
polynomiale Regression 211  
positive Klasse 28  
POSIX Zeit 226
- Prä- und Post-Pruning 72  
Präzision 342  
predict\_proba, Funktion 116, 270  
predict, Methode 22, 38, 66, 252  
probabilistische Modellierung 347  
probabilistisches Programmieren 347  
Problemlösung  
Auswahl von Werkzeugen 343  
Business-Metriken und 342  
einfache vs. komplizierte Fälle 342  
erster Ansatz zu 341  
Konstruieren eines eigenen Estimators 344  
Ressourcen 345  
Schritte zur 342  
Testen Ihres Systems 344
- Produktivsysteme  
Auswahl von Werkzeugen 343  
Testen 344
- Pruning von Entscheidungsbäumen 72
- Pseudozufallszahlen 18  
PyMC Sprache 348  
Python  
Distributionen 6  
Python 2 vs. Python 3 12  
Python(x,y) 7  
statsmodels, Paket 346  
Vorteile von 5
- R**
- R als Sprache 346  
Random Forests 80  
analysieren 82  
Aufbau 80  
Parameter 85  
Randomisierung in 80  
Repräsentation von Daten und 204  
Stärken und Schwächen 84  
Vorhersagen mit 81  
vs. Entscheidungsbäume 80  
vs. Regressionsbäume mit Gradient Boosting 85  
random\_state, Parameter 18  
Ranking 347  
Rastersuche  
*siehe* Gittersuche  
Realzahlen 28  
Receiver-Operating-Characteristic-Kurve (ROC-Kurve) 277  
rectified linear unit (relu) 101  
rectifying nonlinearity 101  
Recurrent Neural Networks (RNNs) 339

Regression 41, 45  
`f_regression` 219, 294  
 LinearRegression 47, 78, 230  
 Regressionsaufgaben  
     Beispiele für 28  
     Boston Immobilien-Datensatz 35  
     Darstellung des Datensatzes wave 33  
     Evaluationsmetriken 284  
     k-nächste-Nachbarn 41  
     Lasso 52  
     lineare Modelle 45  
     Ridge-Regression 49  
     vs. Klassifikationsaufgaben 28  
     Ziele für 28  
 Regressionsbäume mit Gradient Boosting  
     Genauigkeit der Trainingsdaten 87  
     `learning_rate`, Parameter 86  
     Parameter 88  
     Stärken und Schwächen 88  
     vs. Random Forests 85  
     zur Auswahl von Merkmalen 204  
 Regularisierung  
     L1-Regularisierung 52  
     L2-Regularisierung 49, 60  
 reine Blätter 71  
 rekursive Eliminierung von Merkmalen  
     (RFE) 222  
 Relevanz 267  
 Relevanz-Sensitivitäts-Kurven 273  
 Repräsentation als Bag-of-Words  
     Anwendung auf Filmbewertungen 314  
     Arbeitsschritte 311  
     mehr als ein Wort (n-Gramme) 323  
 Ressourcen XI, 345  
 Ressourcen im Netz XI  
 richtig positive/richtig negative 265  
 Richtig-positiv-Rate (RPR) 268, 277  
 Ridge-Regression 49  
 Robustheitsbasiertes Clustering 181

**S**

Schätzen von Unsicherheit  
     Anwendungen für 112  
     bei der Auswertung binärer Klassifikation 270  
     Entscheidungsfunktion 113  
     Vorhersagen von Wahrscheinlichkeiten 116  
 schwache Lerner 85  
 scikit-learn  
     alternative Umgebungen 346  
     Bibliotheken und Werkzeuge 7

Bunch-Objekte 34  
 cancer, Datensatz 34  
 Daten und Labels in 18  
 Dokumentation 6  
`feature_names`, Attribut 35  
`fit_transform`, Methode 130  
`fit`, Methode 21, 65, 112, 127  
 installieren 6  
 knn-Objekt 21  
 minimaler Code für 24  
`predict`, Methode 22, 38, 66  
 Python 2 vs. Python 3 12  
`random_state`, Parameter 18  
`score`, Methode 23, 39, 43  
 Skalierungsmechanismen in 131  
`transform`, Methode 127  
 User Guide 6  
 verwendete Versionen 13  
 Vorteile 6  
 scikit-learn, Klassen und Funktionen  
     `accuracy_score` 179  
     `adjusted_rand_score` 178  
     AgglomerativeClustering 169, 178, 189  
     `average_precision_score` 277  
     BaseEstimator 344  
     classification\_report 269, 283  
     confusion\_matrix 264  
     CountVectorizer 313  
     cross\_val\_score 237, 240, 285, 292, 344  
     DBSCAN 174  
     DecisionTreeClassifier 72, 263  
     DecisionTreeRegressor 72, 77  
     DummyClassifier 263  
     ElasticNet Klasse 55  
     ENGLISH\_STOP\_WORDS 318  
     Estimator 21  
     `export_graphviz` 73  
     `f_regression` 219, 294  
     `f1_score` 268, 276  
     `fetch_lfw_people` 139  
     GradientBoostingClassifier 85, 113, 118  
     GridSearchCV 248, 285, 289, 299, 344  
     GroupKFold 243  
     KFold 240, 244  
     KMeans 163  
     KNeighborsClassifier 21, 38  
     KNeighborsRegressor 43  
     Lasso 52  
     LatentDirichletAllocation 331  
     LeaveOneOut 241  
     LinearRegression 47, 78, 230  
     LinearSVC 56, 62, 65, 66

load\_boston 35, 212, 300  
load\_breast\_cancer 34, 40, 58, 72, 126,  
    136, 219, 289  
load\_digits 155, 263  
load\_files 310  
load\_iris 15, 118, 237  
LogisticRegression 56, 65, 194, 237,  
    264, 299, 315  
make\_blobs 89, 113, 128, 162, 175, 271  
make\_circles 113  
make\_moons 82, 102, 164, 177  
make\_pipeline 297  
MinMaxScaler 97, 125, 127, 177, 212,  
    292, 293, 303  
MLPClassifier 102  
NMF 132, 151, 168, 331  
Normalizer 126  
OneHotEncoder 202, 230  
ParameterGrid 259  
PCA 132, 168, 181, 297, 331  
Pipeline 289, 304  
PolynomialFeatures 209, 231, 300  
precision\_recall\_curve 274  
RandomForestClassifier 80, 221, 275,  
    303  
RandomForestRegressor 80, 214, 222  
RFE 222  
Ridge 49, 65, 105, 214, 216, 294, 300  
RobustScaler 125  
roc\_auc\_score 279  
roc\_curve 278  
SCORERS 287  
SelectFromModel 221  
SelectPercentile 219, 294  
ShuffleSplit 242, 243  
silhouette\_score 180  
StandardScaler 108, 125, 130, 136, 142,  
    177, 298  
StratifiedKFold 244, 259  
StratifiedShuffleSplit 243, 330  
SVC 56, 96, 126, 131, 245, 253, 289, 297  
SVR 88, 212  
TfidfVectorizer 319  
train\_test\_split 18, 235, 271, 274  
TransformerMixin 344  
TSNE 156  
SciPy 8  
score, Methode 23, 39, 43, 252, 292  
Sensitivität 267, 268  
Shuffle-Split-Kreuzvalidierung 242  
sin, Funktion 214

Skalieren  
    Anwenden von Datentransformationen  
        126  
    Arten 125  
    auf größere Datensätze 349  
    Auswirkungen auf überwachtes Lernen  
        130  
    Trainings- und Testdaten 128  
    Zweck 124  
soft voting-Strategie 81  
spark, Rechenumgebung 347  
Sparse Coding (Dictionary Learning) 154  
Stan, Sprache 348  
statsmodels, Paket 346  
Stemming 327  
Stichprobe, Definition 4  
Stopwörter 318  
stratifizierte k-fache Kreuzvalidierung 238  
Streudiagramme 19  
String-kodierte kategorische Daten 198  
Support Vector Machines mit Kernel (SVMs)  
    Kernel-Trick 93  
    lineare Modelle und nichtlineare  
        Merkmale 89  
    Mathematik von 89  
    Optimieren von SVM-Parametern 95  
    Parameter 99  
    Stärken und Schwächen 98  
    Verständnis 93  
    Vorhersagen mit 94  
    Vorverarbeitung von Daten für 97  
        vs. lineare Support Vector Machines 88  
Support-Vektoren 93  
synthetische Datensätze 32

**T**  
Tangens hyperbolicus (tanh) 101  
Teilungen 236  
term frequency-inverse document frequency  
    (tf-idf) 319  
Testdatensätze  
    Boston Immobilien-Datensatz 35  
    Definition 17  
    forge-Datensatz 32  
    wave-Datensatz 33  
    Wisconsin Brustkrebs-Datensatz 34  
Textdaten 307  
    Arten von 307  
    Beispiel Meinungsanalyse 309  
    Beispiele für 307  
    Daten mit tf-idf umskalieren 319

- Modellierung von Themen und  
Clustering von Dokumenten  
331
- Modellkoeffizienten 322
- Repräsentation als Bag-of-Words 311
- Stoppwörter 318
- Überblick 338
- `train_test_split`, Funktion 238
- `transform`, Methode 127, 296, 318
- Transformationen
- Auswahl 217
  - univariate nichtlinear 214
  - unüberwacht 123
- `tree`, Modul 73
- Trefferquote 268
- Trigramme 323
- t-SNE Algorithmus
- siehe* Manifold Learning-Algorithmen
- U**
- überwachtes Lernen 27
- siehe auch* Klassifizierungsaufgaben;  
Regressionsaufgaben
- Algorithmen für
- Ensembles von Entscheidungsbäumen 80
  - Entscheidungsbäume 68
  - k-nächste-Nachbarn 36
  - lineare Modelle 45
  - naive Bayes-Klassifikatoren 66
  - neuronale Netze (Deep Learning) 99
  - Support Vector Machines mit Kernel 88
  - Überblick 2
  - Beispieldatensätze 32
  - Beispiele für 3
  - Komplexität des Modells vs. Größe des Datensatzes 31
  - Overfitting vs. Underfitting 30
  - Repräsentation von Daten 4
  - Schätzung der Unsicherheit 112
  - Überblick über 120
  - Verallgemeinerung 28
  - Ziele für 27
- Umgebungen 346
- Umskalieren
- Beispiel für 124
  - Kernel-SVMs 97
- Unabhängigkeitsanalyse (ICA) 154
- Underfitting 30
- Unigramme 323
- univariate nichtlinear Transformation 214
- univariate Statistiken 218
- unüberwachte Transformationen 123
- unüberwachtes Lernen 123
- Algorithmen für
- agglomeratives Clustering 169
  - Clusteranalyse 158
  - DBSCAN 174
  - Hauptkomponentenzerlegung 132
  - k-Means-Clustering 158
  - Manifold Learning mit t-SNE 154
  - Nicht-negative-Matrix-Faktorisierung 147
- Überblick 3
- Arten 123
- Beispiele für 4
- Herausforderungen für 124
- Repräsentation von Daten 4
- Skalieren und Vorverarbeiten für 124
- Überblick zu 193
- V**
- `value_counts`, Funktion 198
- Vektorquantisierung 165
- Verallgemeinerung
- Beispiele für 29
  - Definition 17
  - Modelle konstruieren zur 28
- verborgene Einheiten 100
- verborgene Schichten 101
- verkettete Algorithmen und Pipelines 289
- Bedeutung von 289
  - Erstellen von Pipelines 292
  - Gittersuche zur Modellauswahl 303
  - Parameterauswahl mit Vorverarbeitung 290
- Pipelines mit `make_pipeline` erstellen 297
- Pipelines zur Gittersuche einsetzen 293
- Pipeline-Schnittstelle 296
- Überblick 304
- Vorverarbeitungsschritte zur Gittersuche 300
- Verlustfunktionen 55
- verschachtelte Kreuzvalidierung 257
- verteiltes Rechnen 347
- Vorhersagen
- für die Zukunft 225
  - von Zeitreihen 347
- Vorverarbeitung 124
- Anwenden von Datentransformationen 126
  - Arten von 125

Auswirkungen auf überwachtes Lernen  
    130  
Parameterauswahl durch 290  
Pipelines und 300  
Skalieren von Trainings- und Testdaten  
    128  
    und Skalieren 124  
    Zweck von 124  
vowpal wabbit 347

## W

wave, Datensatz 33  
Wettbewerbe 350  
Whitening 142

Wichtigkeit von Merkmalen 74  
Wisconsin Brustkrebs-Datensatz 34  
Wortstämme 328  
Wurzeln 69

## X

xgboost, Paket 88  
xkcd Color Survey 308

## Z

Zitieren XII  
Zuordnung mehrerer Klassen  
    vs. binäre Klassifikation 27  
zurückgehaltene Daten 17

# Über die Autoren

**Andreas Müller** hat seine Doktorarbeit an der Universität Bonn zu maschinellem Lernen absolviert. Nachdem er als Forscher für maschinelles Lernen bei Amazon für ein Jahr an Anwendungen zur Bilderkennung gearbeitet hatte, schloss er sich dem Center for Data Science an der New York University an. Die letzten vier Jahre lang war er im Kernteam an der Wartung und Entwicklung von scikit-learn beteiligt, einem im industriellen und akademischen Bereich verbreiteten Werkzeug für maschinelles Lernen. Er hat weitere weitverbreitete Programm Pakete zu maschinellem Lernen selbst entwickelt oder dazu beigetragen. Sein Ziel ist es, frei verfügbare Werkzeuge zu schaffen, um die Einstiegshürden für die Anwendung maschinellen Lernens niedriger zu legen, reproduzierbare Wissenschaft zu fördern und den Zugang zu qualitativ hochwertigen Algorithmen für maschinelles Lernen zu demokratisieren.

**Sarah Guido** ist eine Data Scientist, die lange Zeit in Start-ups gearbeitet hat. Sie liebt Python, maschinelles Lernen, große Datenmengen und die Technologiewelt im Allgemeinen. Sarah ist im Vortragen auf Konferenzen versiert und hat an der University of Michigan promoviert. Sie lebt derzeit in New York City.

# Über die Übersetzer

**Dr. Kristian Rother** arbeitet als Trainer für Forschungsmethoden, Datenverarbeitung und Kommunikation. Er ist in der Python-Community und der Rhetorikorganisation Toastmasters aktiv, um das Beste aus beiden Welten zu kombinieren. Kristian war bis 2011 als Wissenschaftler in der Bioinformatik aktiv, wo er Python-Software zum Modellieren der 3-D-Strukturen von Makromolekülen entwickelt hat. Er hat 2006 an der HU Berlin promoviert.

# Kolophon

Das Tier auf dem Cover von *Einführung in Machine Learning mit Python* ist ein Schlammteufel (*Cryptobranchus alleganiensis*), eine Amphibie, die im Osten der Vereinigten Staaten lebt (von New York bis Georgia). Schon die ersten Einwanderer gaben dem Tier passende Spitznamen wie zum Beispiel »Allegheny Alligator«, »Rotzotter« oder »Wasserhund«. Die Herkunft des Namens »Schlammteufel« ist unklar: Angeblich gaben deutschstämmige Siedler dem Tier diesen Namen, da sie es für einen Dämon hielten, der aus der Hölle zurückgekehrt war.

Der Schlammteufel gehört zur Familie der Riesensalamander und kann bis zu 70 Zentimeter lang werden. Nur zwei Arten in Asien sind noch größer. Der Körper ist relativ flach, mit dicken Hautfalten an beiden Flanken. Obwohl er Kiemen an den Seiten des Halses hat,

atmet der Schlammtreufel vor allem durch seine Haut: Der Sauerstoff strömt durch Kapillare ein und aus, die sich dicht unter der Oberfläche der Haut befinden.

Aus diesem Grund braucht der Schlammtreufel frisches, sauerstoffreiches Wasser von Bächen und Flüssen. Er lebt versteckt unter Steinen und erbeutet seine Nahrung mithilfe des Geruchssinns, aber auch kleinste Vibrationen im Wasser werden wahrgenommen. Die Beute besteht aus kleinen Krebsen, Fischen und Eiern – manchmal sogar den eigenen. In der Nahrungskette spielt er eine wichtige Rolle, da er selbst ein beliebtes Beutetier für Fische, Schlangen und Schildkröten ist.

Die Schlammtreufel-Bestände sind in den letzten Jahrzehnten dramatisch zurückgegangen. Das liegt zum einen an der Verschlechterung der Wasserqualität. Durch ihr Atmungssystem reagieren sie extrem sensibel auf Wasserverunreinigungen. Landwirtschaft und Verbauung in der Nähe von Fließgewässern sorgen andererseits für eine Zunahme von Sedimenten und Chemikalien im Wasser. Man versucht, die bedrohte Art zu schützen, indem Individuen in Aufzuchtstationen bis zu einem gewissen Alter aufgepäppelt werden, die man dann in sauberen Gewässern ansiedelt.

Viele der Tiere auf den O'Reilly-Covern sind vom Aussterben bedroht. Doch jedes einzelne von ihnen ist für den Erhalt unserer Erde wichtig. Wie man bedrohten Arten helfen kann, erfahren Sie auf [animals.oreilly.com](http://animals.oreilly.com).

Der Umschlagsentwurf dieses Buches basiert auf dem Reihenlayout von Edie Freedman und stammt von Karen Montgomery und Michael Oreal, die hierfür einen Stich aus *Wood's Animate Creation* verwendet haben. Auf dem Cover verwenden wir die Schriften URW Typewriter und Guardian Sans, als Textschrift die Linotype Birk, die Überschriftenschrift ist die Adobe Myriad Condensed, und die Nichtproportionalschrift für Codes ist LucasFonts TheSans Mono Condensed.