O'REILLY®

3. Auflage
Aktualisiert
und erweitert

Praxiseinstieg Machine Learning

mit Scikit-Learn, Keras und TensorFlow

Konzepte, Tools und Techniken
für intelligente Systeme

powered by
jupyter

Aurélien Géron
Übersetzung von Kristian Rother
und Thomas Demmig

3. AUFLAGE

Praxiseinstieg Machine Learning mit Scikit-Learn, Keras und TensorFlow

Konzepte, Tools und Techniken für intelligente Systeme

Aurélien Géron

Deutsche Übersetzung von Kristian Rother & Thomas Demmig

O'REILLY®

Aurélien Géron

Lektorat: Alexandra Follenius
Übersetzung: Kristian Rother, Thomas Demmig
Korrektorat: Sibylle Feldmann, *www.richtiger-text.de*
Satz: III-satz, *www.drei-satz.de*
Herstellung: Stefanie Weidner
Umschlaggestaltung: Karen Montgomery, Michael Oréal, *www.oreal.de*
Druck und Bindung: mediaprint solutions 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 978-3-96009-212-4
PDF 978-3-96010-760-6
ePub 978-3-96010-761-3
mobi 978-3-96010-762-0

3., aktualisierte und erweiterte Auflage 2023

Wieblinger Weg 17
69123 Heidelberg

Hinweis:
Dieses Buch wurde mit mineralölfreien Farben auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie. Hergestellt in Deutschland.

Schreiben Sie uns:
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: *kommentar@oreilly.de*.

5 4 3 2 1 0

Inhalt

Vorwort

Der Machine-Learning-Tsunami

Im Jahr 2006 erschien ein Artikel (*https://homl.info/136*) von Geoffrey Hinton et al.[1], in dem vorgestellt wurde, wie sich ein neuronales Netz zum Erkennen handgeschriebener Ziffern mit ausgezeichneter Genauigkeit (> 98 %) trainieren lässt. Ein Deep Neural Network ist ein (sehr) vereinfachtes Modell unseres zerebralen Kortex, und es besteht aus einer Folge von Schichten mit künstlichen Neuronen. Die Autoren nannten diese Technik »Deep Learning«. Zu dieser Zeit wurde das Trainieren eines Deep-Learning-Netzes im Allgemeinen als unmöglich angesehen,[2] und die meisten Forscher hatten die Idee in den 1990ern aufgegeben. Dieser Artikel ließ das Interesse der wissenschaftlichen Gemeinde wieder aufleben, und schon nach kurzer Zeit zeigten weitere Artikel, dass Deep Learning nicht nur möglich war, sondern umwerfende Dinge vollbringen konnte, zu denen kein anderes Machine-Learning-(ML-)Verfahren auch nur annähernd in der Lage war (mithilfe enormer Rechenleistung und riesiger Datenmengen). Dieser Enthusiasmus breitete sich schnell auf weitere Teilgebiete des Machine Learning aus.

Zehn Jahre später hat Machine Learning ganze Industriezweige erobert: Es ist zu einem Herzstück heutiger Spitzentechnologien geworden und dient dem Ranking von Suchergebnissen im Web, kümmert sich um die Spracherkennung Ihres Smartphones, gibt Empfehlungen für Videos und steuert vielleicht sogar Ihr Auto.

1 Geoffrey Hinton et al., »A Fast Learning Algorithm for Deep Belief Nets«, *Neural Computation* 18 (2006): 1527–1554.

2 Obwohl die Konvolutionsnetze von Yann Lecun bei der Bilderkennung seit den 1990ern gut funktioniert hatten, auch wenn sie nicht allgemein anwendbar waren.

Machine Learning in Ihren Projekten

Deshalb interessieren Sie sich natürlich auch für Machine Learning und möchten an der Party teilnehmen!

Womöglich möchten Sie Ihrem selbst gebauten Roboter einen eigenen Denkapparat geben? Ihn Gesichter erkennen lassen? Oder ihn lernen lassen, herumzulaufen?

Oder vielleicht besitzt Ihr Unternehmen Unmengen an Daten (Logdateien, Finanzdaten, Produktionsdaten, Sensordaten, Hotline-Statistiken, Personalstatistiken und so weiter), und Sie könnten vermutlich einige verborgene Schätze heben, wenn Sie nur wüssten, wo Sie danach suchen müssten. Mit Machine Learning können Sie das Folgende (und noch viel mehr (*https://homl.info/usecases*)) erreichen:

- Kundensegmente finden und für jede Gruppe die beste Marketingstrategie entwickeln.
- Jedem Kunden anhand des Kaufverhaltens ähnlicher Kunden Produktempfehlungen geben.
- Betrügerische Transaktionen mit hoher Wahrscheinlichkeit erkennen.
- Den Unternehmensgewinn im nächsten Jahr vorhersagen.

Was immer der Grund ist, Sie haben beschlossen, Machine Learning zu erlernen und in Ihren Projekten umzusetzen. Eine ausgezeichnete Idee!

Ziel und Ansatz

Dieses Buch geht davon aus, dass Sie noch so gut wie nichts über Machine Learning wissen. Unser Ziel ist es, Ihnen die Grundbegriffe, ein Grundverständnis und die Werkzeuge an die Hand zu geben, mit denen Sie Programme zum *Lernen aus Daten* entwickeln können.

Wir werden eine Vielzahl von Techniken besprechen, von den einfachsten und am häufigsten eingesetzten (wie der linearen Regression) bis zu einigen Deep-Learning-Verfahren, die regelmäßig Wettbewerbe gewinnen. Wir werden dazu für den Produktionsbetrieb geschaffene Python-Frameworks verwenden:

- Scikit-Learn (*http://scikit-learn.org/*) ist sehr einfach zu verwenden, enthält aber effiziente Implementierungen vieler Machine-Learning-Algorithmen. Damit ist es ein großartiger Ausgangspunkt, um Machine Learning zu erlernen. Scikit-Learn wurde von David Cournapeau im Jahr 2007 erstellt und wird mittlerweile von einem Forschungsteam am Nationalen Forschungsinstitut für Informatik und Automatisierung (INRIA) in Frankreich betreut.
- TensorFlow (*http://tensorflow.org/*) ist eine komplexere Bibliothek für verteiltes Rechnen. Mit ihr können Sie sehr große neuronale Netze effizient trainieren und ausführen, indem Sie die Berechnungen auf bis zu Hunderte von Servern mit mehreren GPUs (*Graphics Processing Units*) verlagern. TensorFlow

(TF) wurde von Google entwickelt und läuft in vielen umfangreichen Machine-Learning-Anwendungen. Die Bibliothek wurde im November 2015 als Open Source veröffentlicht, im September 2019 erschien Version 2.0.

- Keras (*https://keras.io/*) ist eine High-Level-Deep-Learning-API, die das Trainieren und Ausführen neuronaler Netze sehr einfach macht. Sie kommt zusammen mit TensorFlow und greift für alle rechenintensiven Aufgaben darauf zurück.

Dieses Buch verfolgt einen praxisorientierten Ansatz, bei dem Sie ein intuitives Verständnis von Machine Learning entwickeln, indem Sie sich mit konkreten Beispielen und ein klein wenig Theorie beschäftigen.

Auch wenn Sie dieses Buch lesen können, ohne Ihren Laptop in die Hand zu nehmen, empfehlen wir Ihnen, mit den Codebeispielen herumzuexperimentieren.

Codebeispiele

Alle Codebeispiele in diesem Buch sind Open Source und stehen online unter *https://github.com/ageron/handson-mls3* als Jupyter Notebooks zur Verfügung. Dabei handelt es sich um interaktive Dokumente mit Texten, Bildern und ausführbaren Codeabschnitten (in unserem Fall Python). Am einfachsten beginnen Sie damit, diese Notebooks mit Google Colab auszuführen. Das ist ein kostenloser Service, der es Ihnen ermöglicht, beliebige Jupyter Notebooks direkt online laufen zu lassen, ohne etwas auf Ihrem Rechner installieren zu müssen. Sie brauchen nur einen Webbrowser und einen Google-Account.

In diesem Buch gehe ich davon aus, dass Sie Google Colab nutzen, aber ich habe die Notebooks auch auf anderen Onlineplattformen wie Kaggle oder Binder getestet, sodass Sie sie dort ebenfalls einsetzen können, wenn Ihnen das lieber ist. Alternativ können Sie die erforderlichen Bibliotheken und Tools (oder das Docker-Image für dieses Buch) direkt auf Ihrem Computer installieren und die Notebooks dort laufen lassen. Die Anweisungen dazu finden Sie unter *https://homl.info/install*.

Verwenden von Codebeispielen

Dieses Buch ist dazu da, Ihnen beim Erledigen Ihrer Arbeit zu helfen. Wollen Sie über die Codebeispiele hinaus zusätzliche Inhalte in einem Umfang verwenden, der die Regeln eines fairen Einsatzes verletzen würde (zum Beispiel der Verkauf oder Vertrieb von Inhalten aus O'Reilly-Büchern oder das Einbinden eines beträchtlichen Teils dieses Buchs in die Dokumentation Ihres Produkts), wenden Sie sich bitte für eine Erlaubnis an *kommentar@oreilly.de*.

Wir freuen uns über Zitate, verlangen diese aber nicht. Ein Zitat enthält Titel, Autor, Verlag und ISBN. Beispiel: »*Praxiseinstieg Machine Learning mit Scikit-Learn, Keras und TensorFlow* von Aurélien Géron, O'Reilly 2023, ISBN 978-3-96009-212-4.«

Voraussetzungen

Dieses Buch geht davon aus, dass Sie ein wenig Programmiererfahrung mit Python haben. Sollten Sie Python noch nicht kennen, ist *http://learnpython.org/* ein ausgezeichneter Ausgangspunkt. Auch das offizielle Tutorial auf python.org (*https://docs.python.org/3/tutorial/*) ist recht gut.

Es setzt ebenfalls voraus, dass Sie mit den wichtigsten wissenschaftlichen Bibliotheken in Python vertraut sind, insbesondere mit NumPy (*http://numpy.org/*), Pandas (*http://pandas.pydata.org/*) und Matplotlib (*http://matplotlib.org/*). Falls Sie diese Bibliotheken noch nie verwendet haben, ist das nicht schlimm – ihr Einsatz lässt sich leicht erlernen, und ich habe für jede von ihnen ein Tutorial erstellt. Diese finden Sie online unter *https://homl.info/tutorials*.

Wenn Sie vollständig verstehen wollen, wie die Algorithmen zum Machine Learning funktionieren (und nicht nur, wie man sie einsetzt), sollten Sie ein Grundverständnis von einigen mathematischen Konzepten haben – insbesondere von linearer Algebra. Sie sollten wissen, was Vektoren und Matrizen sind und wie Sie einige einfache Operationen mit ihnen ausführen, zum Beispiel das Addieren von Vektoren oder das Transponieren und Multiplizieren von Matrizen. Sollten Sie eine kurze Einführung in die lineare Algebra benötigen (sie ist wirklich kein Hexenwerk), habe ich unter *https://homl.info/tutorials* ein Tutorial bereitgestellt. Sie finden dort ebenfalls eines zur Differenzialrechnung, was hilfreich sein kann, um zu verstehen, wie neuronale Netze trainiert werden, aber für das grobe Verstehen der wichtigsten Konzepte ist es nicht unbedingt von entscheidender Bedeutung. Dieses Buch greift gelegentlich auch auf andere mathematische Konzepte zurück, beispielsweise auf Exponentialfunktionen und Logarithmen, ein bisschen Wahrscheinlichkeitstheorie und ein paar Statistikaspekte, aber alles nicht zu ausgefallen. Brauchen Sie dazu Hilfe, schauen Sie mal bei *https://khanacademy.org* vorbei, wo Sie online viele ausgezeichnete und kostenlose Mathematikkurse finden.

Wegweiser durch dieses Buch

Dieses Buch ist in zwei Teile aufgeteilt. Teil I behandelt folgende Themen:

- Was ist Machine Learning? Welche Aufgaben lassen sich damit lösen? Was sind die wichtigsten Kategorien und Grundbegriffe von Machine-Learning-Systemen?
- Die Schritte in einem typischen Machine-Learning-Projekt.

- Lernen durch Anpassen eines Modells an Daten.
- Optimieren einer Kostenfunktion.
- Bearbeiten, Säubern und Vorbereiten von Daten.
- Merkmale auswählen und entwickeln.
- Ein Modell auswählen und dessen Hyperparameter über Kreuzvalidierung optimieren.
- Die Herausforderungen beim Machine Learning, insbesondere Underfitting und Overfitting (das Gleichgewicht zwischen Bias und Varianz).
- Die verbreitetsten Lernalgorithmen: lineare und polynomielle Regression, logistische Regression, *k*-nächste Nachbarn, Support Vector Machines, Entscheidungsbäume, Random Forests und Ensemble-Methoden.
- Dimensionsreduktion der Trainingsdaten, um dem »Fluch der Dimensionalität« etwas entgegenzusetzen.
- Andere Techniken des unüberwachten Lernens, unter anderem Clustering, Dichteabschätzung und Anomalieerkennung.

Teil II widmet sich diesen Themen:

- Was sind neuronale Netze? Wofür sind sie geeignet?
- Erstellen und Trainieren neuronaler Netze mit TensorFlow und Keras.
- Die wichtigsten Architekturen neuronaler Netze: Feed-Forward-Netze für Tabellendaten, Convolutional Neural Networks zur Bilderkennung, rekurrente Netze und Long-Short-Term-Memory-(LSTM-)Netze zur Sequenzverarbeitung, Encoder/Decoder und Transformer für die Sprachverarbeitung (und mehr!), Autoencoder sowie Generative Adversarial Networks (GANs) und Diffusionsmodelle zum generativen Lernen.
- Techniken zum Trainieren von Deep-Learning-Netzen.
- Wie man einen Agenten erstellt (zum Beispiel einen Bot in einem Spiel), der durch Versuch und Irrtum gute Strategien erlernt und dabei Reinforcement Learning einsetzt.
- Effizientes Laden und Vorverarbeiten großer Datenmengen.
- Trainieren und Deployen von TensorFlow-Modellen im großen Maßstab.

Der erste Teil baut vor allem auf Scikit-Learn auf, der zweite Teil verwendet TensorFlow.

Springen Sie nicht zu schnell ins tiefe Wasser: Auch wenn Deep Learning zweifelsohne eines der aufregendsten Teilgebiete des Machine Learning ist, sollten Sie zuerst Erfahrungen mit den Grundlagen sammeln. Außerdem lassen sich die meisten Aufgabenstellungen recht gut mit einfacheren Techniken wie Random Forests und Ensemble-Methoden lösen (die in Teil I besprochen werden). Deep Learning ist am besten für komplexe Aufgaben

wie Bilderkennung, Spracherkennung und Sprachverarbeitung geeignet. Sie müssen aber genug Daten, Rechenleistung und Geduld haben (sofern Sie nicht ein vortrainiertes neuronales Netz nutzen können, wie Sie noch sehen werden).

Änderungen zwischen der ersten und der zweiten Auflage

Haben Sie schon die erste Auflage gelesen, finden Sie hier die wichtigsten Unterschiede zwischen der ersten und zweiten Auflage:

- Der gesamte Code wurde von TensorFlow 1.x auf TensorFlow 2.x gehoben, und ich habe einen Großteil des Low-Level-Codes mit TensorFlow (Graphen, Sessions, Feature Columns, Estimators und so weiter) durch viel einfacheren Keras-Code ersetzt.
- In der zweiten Auflage wurde die Data-API zum Laden und Vorverarbeiten großer Datasets aufgenommen, die Distribution Strategies API zum Trainieren und Deployen von TF-Modellen im großen Maßstab, TF Serving und die Google Cloud API Platform, mit denen Modelle in Produkte umgewandelt werden können, sowie (zumindest angerissen) TF Transform, TFLite, TF Addons/Seq2Seq, TensorFlow.js und TF Agents.
- Es wurden auch viele zusätzliche ML-Themen hinzugenommen, unter anderem ein neues Kapitel zu unüberwachtem Lernen, Computer-Vision-Techniken zur Objekterkennung und zur semantischen Segmentierung, außerdem das Verarbeiten von Sequenzen durch Convolutional Neural Networks (CNNs), die Verarbeitung natürlicher Sprache (Natural Language Processing, NLP) mit rekurrenten neuronalen Netzen (RNNs), CNNs und Transformer, GANs und vieles mehr.

Auf *https://homl.info/changes2* finden Sie mehr Details.

Änderungen zwischen der zweiten und der dritten Auflage

Haben Sie die zweite Auflage gelesen, finden Sie hier die wichtigsten Änderungen zwischen zweiter und dritter Auflage:

- Der gesamte Code wurde an die neuesten Bibliotheksversionen angepasst. Insbesondere wurden in dieser dritten Auflage viele neue Ergänzungen von Scikit-Learn berücksichtigt (zum Beispiel das Feature Name Tracking, histogrammbasiertes Gradient Boosting, Label Propagation und mehr). Zudem wurden die Bibliothek *Keras Tuner* für das Tuning von Hyperparametern, die

Transformers-Bibliothek von Hugging Face für die Verarbeitung natürlicher Sprache sowie die neuen Vorverarbeitungsschichten und Data Augmentation Layer mit aufgenommen.

- Es wurden diverse Vision-Modelle hinzugefügt (ResNeXt, DenseNet, MobileNet, CSPNet und EfficientNet) und dazu Entscheidungshilfen für das Wählen des richtigen Modells.
- In Kapitel 15 werden nun statt generierter Zeitserien die *Chicago Bus and Rail Ridership*-Daten analysiert, und das ARMA-Modell mit seinen Varianten wird vorgestellt.
- Kapitel 16 zur Verarbeitung natürlicher Sprache baut nun ein Modell zum Übersetzen aus dem Englischen ins Spanische, wobei zuerst ein Encoder-Decoder-RNN und dann ein Transformer-Modell zum Einsatz kommen. Das Kapitel behandelt zudem Sprachmodelle wie Switch Transformers, DistilBERT, T5 und PaLM (mit einem Chain-of-Thought Prompting), außerdem stellt es Vision Transformers (ViTs) vor und gibt einen Überblick über ein paar auf Transformern basierende visuelle Modelle, wie zum Beispiel Data-Efficient Image Transformers (DeiTs), Perceivers und DINO. Zudem gibt es einen kurzen Hinweis auf ein paar große, multimodale Modelle – unter anderem CLIP, DALL·E, Flamingo und GATO.
- In Kapitel 17 zu generativem Lernen werden nun Diffusionsmodelle vorgestellt, und es wird gezeigt, wie Sie ein Denoising Diffusion Probabilistic Model (DDPM) von Grund auf implementieren.
- Kapitel 19 ist von der Google Cloud AI Platform nach Google Vertex AI umgezogen und nutzt jetzt verteilte Keras Tuner für die Hyperparametersuche im großen Maßstab. Es führt nun TensorFlow.js-Code ein, mit dem Sie online experimentieren können. Zudem werden zusätzliche verteilte Trainingstechniken vorgestellt, unter anderem PipeDream und Pathways.
- Um all die neuen Inhalte unterbringen zu können, stehen manche Abschnitte nur noch online zur Verfügung, unter anderem die Installationsanweisungen, die Hauptkomponentenzerlegung (PCA), mathematische Details zu bayesschen gaußschen Mischverteilungsmodellen, TF Agents und die früheren Anhänge A (Lösungen zu den Übungsaufgaben), C (die Mathematik von Support Vector Machines) und E (weitere Architekturen für neuronale Netze). Diese drei früheren Anhänge finden Sie auf der deutschen Website zum Buch unter *https://dpunkt.de/produkt/praxiseinstieg-machine-learning-mit-scikit-learn-keras-und-tensorflow-2/* auf der Registerkarte *Zusatzmaterial*.

Auf *https://homl.info/changes3* finden Sie zusätzliche Informationen zu den Änderungen.

Ressourcen im Netz

Es gibt viele ausgezeichnete Ressourcen, mit deren Hilfe sich Machine Learning erlernen lässt. Der ML-Kurs auf Coursera (*https://homl.info/ngcourse*) von Andrew Ng ist faszinierend, auch wenn er einen beträchtlichen Zeitaufwand bedeutet.

Darüber hinaus finden Sie viele interessante Webseiten über Machine Learning, darunter natürlich den ausgezeichneten User Guide (*https://homl.info/skdoc*) von Scikit-Learn. Auch Dataquest (*https://www.dataquest.io/*), das sehr ansprechende Tutorials und ML-Blogs bietet, sowie die auf Quora (*https://homl.info/1*) aufgeführten ML-Blogs könnten Ihnen gefallen.

Natürlich bieten darüber hinaus viele andere Bücher eine Einführung in Machine Learning, insbesondere:

- Joel Grus, *Einführung in Data Science: Grundprinzipien der Datenanalyse mit Python (https://oreilly.de/produkt/einfuehrung-in-data-science-2/)* (O'Reilly, 2. Auflage). Dieses Buch stellt die Grundlagen von Machine Learning vor und implementiert die wichtigsten Algorithmen in reinem Python (von Grund auf).
- Stephen Marsland, *Machine Learning: An Algorithmic Perspective, 2nd Edition* (Chapman & Hall). Dieses Buch ist eine großartige Einführung in Machine Learning, die viele Themen ausführlich behandelt. Es enthält Codebeispiele in Python (ebenfalls von Grund auf, aber mit NumPy).
- Sebastian Raschka, *Machine Learning mit Python und Keras, TensorFlow 2 und Scikit-learn: Das umfassende Praxis-Handbuch für Data Science, Deep Learning und Predictive Analytics* (mitp Professional, 3. Auflage). Eine weitere ausgezeichnete Einführung in Machine Learning. Dieses Buch konzentriert sich auf Open-Source-Bibliotheken in Python (Pylearn 2 und Theano).
- François Chollet, *Deep Learning with Python* (Manning, 2nd Edition). Ein sehr praxisnahes Buch, das klar und präzise viele Themen behandelt – wie Sie es vom Autor der ausgezeichneten Keras-Bibliothek erwarten können. Es zieht Codebeispiele der mathematischen Theorie vor.
- Andriy Burkov, *Machine Learning kompakt: Alles, was Sie wissen müssen* (mitp Professional). Dieses sehr kurze Buch behandelt ein beeindruckendes Themenspektrum gut verständlich, scheut dabei aber nicht vor mathematischen Gleichungen zurück.
- Yaser S. Abu-Mostafa, Malik Magdon-Ismail und Hsuan-Tien Lin, *Learning from Data* (AMLBook). Als eher theoretische Abhandlung von ML enthält dieses Buch sehr tiefgehende Erkenntnisse, insbesondere zum Gleichgewicht zwischen Bias und Varianz (siehe Kapitel 4).
- Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach, 4th Edition* (Pearson). Dieses ausgezeichnete (und umfangreiche) Buch deckt

eine unglaubliche Stoffmenge ab, darunter Machine Learning. Es hilft dabei, ML in einem breiteren Kontext zu betrachten.

- Jeremy Howard and Sylvain Gugger, *Deep Learning for Coders with fastai and PyTorch* (O'Reilly). Eine wunderbar klare und praxisnahe Einführung in Deep Learning mithilfe der fastai- und PyTorch-Bibliotheken.

Eine gute Möglichkeit zum Lernen sind schließlich Webseiten mit ML-Wettbewerben wie Kaggle.com (*https://kaggle.com/*). Dort können Sie Ihre Fähigkeiten an echten Aufgaben üben und Hilfe und Tipps von einigen der besten ML-Profis erhalten.

In diesem Buch verwendete Konventionen

Die folgenden typografischen Konventionen werden in diesem Buch verwendet:

Kursiv
: Kennzeichnet neue Begriffe, URLs, E-Mail-Adressen, Dateinamen und Dateiendungen.

`Konstante Zeichenbreite`
: Wird für Programmlistings und für Programmelemente in Textabschnitten wie Namen von Variablen und Funktionen, Datenbanken, Datentypen, Umgebungsvariablen, Anweisungen und Schlüsselwörter verwendet.

`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 Symbol steht für einen Tipp oder eine Empfehlung.

Dieses Symbol steht für einen allgemeinen Hinweis.

Dieses Symbol steht für eine Warnung oder erhöhte Aufmerksamkeit.

Danksagungen

In meinen wildesten Träumen hätte ich mir niemals vorgestellt, dass die erste und zweite Auflage dieses Buchs solch eine Verbreitung finden würde. Ich habe so viele Nachrichten von Lesern erhalten – oft mit Fragen, manche mit freundlichen Hinweisen auf Fehler und die meisten mit ermutigenden Worten. Ich kann gar nicht sagen, wie dankbar ich all diesen Lesern für ihre Unterstützung bin. Vielen, vielen Dank! Scheuen Sie sich nicht, sich auf GitHub (*https://homl.info/issues3*) zu melden, wenn Sie Fehler in den Codebeispielen finden (oder einfach etwas fragen wollen) oder um auf Fehler im Text aufmerksam (*https://homl.info/errata3*) zu machen. Manche Leser haben mir auch geschrieben, wie dieses Buch ihnen dabei geholfen hat, ihren ersten Job zu bekommen oder ein konkretes Problem zu lösen, an dem sie gearbeitet haben. Ich finde ein solches Feedback unglaublich motivierend. Hat Ihnen dieses Buch geholfen, würde ich mich freuen, wenn Sie mir Ihre Geschichte erzählen würden – entweder privat (zum Beispiel über LinkedIn (*https://linkedin.com/in/aurelien-geron/*)) oder öffentlich (beispielsweise in einem Tweet an @aureliengeron oder über ein Amazon-Review (*https://homl.info/amazon3*)).

Ein großer Dank geht ebenfalls an all die wundervollen Menschen, die ihre Zeit und Expertise angeboten haben, um diese dritte Auflage zu begutachten, Fehler zu korrigieren und unzählige Vorschläge zu machen. Diese Auflage ist dank ihnen so viel besser geworden: Olzhas Akpambetov, George Bonner, François Chollet, Siddha Gangju, Sam Goodman, Matt Harrison, Sasha Sobran, Lewis Tunstall, Leandro von Werra und mein lieber Bruder Sylvain. Ihr seid toll!

Ebenfalls dankbar bin ich den vielen Menschen, die mich auf meinem Weg unterstützt haben, indem sie meine Fragen beantworteten, Verbesserungen vorschlugen und zum Code auf GitHub beitrugen – insbesondere Yannick Assogba, Ian Beauregard, Ulf Bissbort, Rick Chao, Peretz Cohen, Kyle Gallatin, Hannes Hapke, Victor Khaustov, Soonson Kwon, Eric Lebigot, Jason Mayes, Laurence Moroney, Sara Robinson, Joaquín Ruales und Yuefeng Zhou.

Dieses Buch würde ohne die fantastischen Mitarbeiterinnen und Mitarbeiter von O'Reilly nicht existieren, insbesondere nicht ohne Nicole Taché, die mir sehr hilfreiches Feedback gab und immer aufmunternd, unterstützend und hilfreich war – eine bessere Lektorin könnte ich mir nicht vorstellen. Ein großer Dank geht auch an Michele Cronin, die mich durch die letzten Kapitel begleitet und mich über die Ziellinie gebracht hat. Vielen Dank an das ganze Produktionsteam, insbesondere an Elizabeth Kelly und Kristen Brown. Ebenfalls vielen Dank an Kim Cofer für das Redigieren und an Johnny O'Toole, der sich um die Beziehung zu Amazon kümmerte und viele meiner Fragen beantwortet hat. Danke an Kate Dullea für das großartige Verbessern meiner Illustrationen. Danke an Marie Beaugurea, Ben Lorica, Mike Loukides und Laurel Ruma dafür, dass sie an dieses Projekt geglaubt und seinen Rahmen definiert haben. Danke an Matt Hacker und alle anderen im Atlas-Team, die all meine technischen Fragen in Bezug auf Formatierung, AsciiDoc,

MathML und LaTeX beantworten konnten, und an Nick Adams, Rebecca Demarest, Rachel Head, Judith McConville, Helen Monroe, Karen Montgomery, Rachel Roumeliotis und alle anderen bei O'Reilly, die zu diesem Buch beigetragen haben.

Ich werde nie all die wunderbaren Menschen vergessen, die mir bei der ersten und zweiten Auflage dieses Buchs geholfen haben: Freunde, Kollegen und Experten – einschließlich vieler Mitglieder des TensorFlow-Teams. Die Liste ist lang: Olzhas Akpambetov, Karmel Allison, Martin Andrews, David Andrzejewski, Paige Bailey, Lukas Biewald, Eugene Brevdo, William Chargin, François Chollet, Clément Courbet, Robert Crowe, Mark Daoust, Daniel »Wolff« Dobson, Julien Dubois, Mathias Kende, Daniel Kitachewsky, Nick Felt, Bruce Fontaine, Justin Francis, Goldie Gadde, Irene Giannoumis, Ingrid von Glehn, Vincent Guilbeau, Sandeep Gupta, Priya Gupta, Kevin Haas, Eddy Hung, Konstantinos Katsiapis, Viacheslav Kovalevskyi, Jon Krohn, Allen Lavoie, Karim Matrah, Grégoire Mesnil, Clemens Mewald, Dan Moldovan, Dominic Monn, Sean Morgan, Tom O'Malley, James Pack, Alexander Pak, Haesun Park, Alexandre Passos, Ankur Patel, Josh Patterson, André Susano Pinto, Anthony Platanios, Anosh Raj, Oscar Ramirez, Anna Revinskaya, Saurabh Saxena, Salim Sémaoune, Ryan Sepassi, Vitor Sessak, Jiri Simsa, Iain Smears, Xiaodan Song, Christina Sorokin, Michel Tessier, Wiktor Tomczak, Dustin Tran, Todd Wang, Pete Warden, Rich Washington, Martin Wicke, Edd Wilder-James, Sam Witteveen, Jason Zaman, Yuefeng Zhou und mein Bruder Sylvain.

Schließlich bin ich meiner geliebten Frau Emmanuelle und unseren drei wunderbaren Kindern Alexandre, Rémi und Gabrielle unendlich dafür dankbar, dass sie mich zur Arbeit an diesem Buch ermutigt haben. Ihre unstillbare Neugier war unbezahlbar: Indem ich einige der schwierigsten Konzepte in diesem Buch meiner Frau und meinen Kindern erklärt habe, konnte ich meine Gedanken ordnen und viele Teile direkt verbessern. Und sie haben mir sogar Kekse und Kaffee vorbeigebracht. Was kann man sich noch mehr wünschen?

TEIL I

Die Grundlagen des Machine Learning

KAPITEL 1

Die Machine-Learning-Umgebung

Hätten Sie vor noch gar nicht so langer Zeit zu Ihrem Telefon gegriffen und es gebeten, Ihnen den Weg nach Hause zu zeigen, wären Sie mit Stille bestraft worden – und die Leute um Sie herum hätten an Ihrer geistigen Gesundheit gezweifelt. Aber Machine Learning ist nicht mehr länger reine Science-Fiction: Milliarden Menschen auf dem Planeten nutzen sie tagtäglich. Tatsächlich gibt es Machine Learning in bestimmten spezialisierten Anwendungsbereichen wie der optischen Zeichenerkennung (OCR) schon seit Jahrzehnten. Aber die erste weitverbreitete Anwendung von ML, die das Leben von Hunderten Millionen Menschen verbesserte, hat die Welt in den 1990er-Jahren erobert: Es war der *Spamfilter*. Es ist nicht gerade ein Skynet mit eigenem Bewusstsein, aber technisch gesehen ist es Machine Learning (es hat inzwischen so gut gelernt, dass Sie nur noch selten eine E-Mail als Spam kennzeichnen müssen). Dem Spamfilter folgten etliche weitere Anwendungen von ML, die still und heimlich Hunderte Produkte und Funktionen aus dem Alltag steuern: Stimmerkennung, automatische Übersetzungen, Bildersuche, Produktempfehlungen und viele weitere.

Wo aber beginnt Machine Learning, und wo hört es auf? Worum genau geht es, wenn eine Maschine etwas *lernt*? Wenn ich mir eine Kopie aller Wikipedia-Artikel herunterlade, hat mein Computer dann schon etwas gelernt? Ist er auf einmal schlauer geworden? In diesem Kapitel werden wir erst einmal klarstellen, was Machine Learning ist und wofür Sie es einsetzen könnten.

Bevor wir aber beginnen, den Kontinent des Machine Learning zu erforschen, werfen wir einen Blick auf die Landkarte und lernen die wichtigsten Regionen und Orientierungspunkte kennen: überwachtes und unüberwachtes Lernen (mit seinen Varianten), Online- und Batch-Learning, instanzbasiertes und modellbasiertes Lernen. Anschließend betrachten wir die Arbeitsabläufe in einem typischen ML-Projekt, diskutieren die dabei wichtigsten Herausforderungen und besprechen, wie Sie ein Machine-Learning-System auswerten und optimieren können.

In diesem Kapitel werden diverse Grundbegriffe (und wird Fachjargon) eingeführt, die jeder Data Scientist auswendig kennen sollte. Es wird ein abstrakter und recht

einfacher Überblick bleiben (das einzige Kapitel mit wenig Code), aber mein Ziel ist es, sicherzustellen, das alles glasklar ist, bevor wir mit dem Buch fortfahren. Schnappen Sie sich also einen Kaffee, und los geht's!

Wenn Sie bereits mit den Grundlagen von Machine Learning vertraut sind, können Sie direkt mit Kapitel 2 fortfahren. Falls Sie sich nicht sicher sind, versuchen Sie, die Fragen am Ende des Kapitels zu beantworten, bevor Sie weitergehen.

Was ist Machine Learning?

Machine Learning ist die Wissenschaft (und Kunst), Computer so zu programmieren, dass sie *anhand von Daten lernen*.

Hier ist eine etwas allgemeinere Definition:

> [Maschinelles Lernen ist das] Fachgebiet, das Computern die Fähigkeit zu lernen verleiht, ohne explizit programmiert zu werden.
>
> *– Arthur Samuel 1959*

Und eine eher technisch orientierte:

> Man sagt, dass ein Computerprogramm dann aus Erfahrungen *E* in Bezug auf eine Aufgabe *T* und ein Maß für die Leistung *P* lernt, wenn seine durch *P* gemessene Leistung bei *T* mit der Erfahrung *E* anwächst.
>
> *– Tom Mitchell 1997*

Ihr Spamfilter ist ein maschinelles Lernprogramm, das aus Beispielen für Spam-E-Mails (vom Nutzer markierten) und gewöhnlichen E-Mails (Nicht-Spam, auch »Ham« genannt) lernt, Spam zu erkennen. Diese vom System verwendeten Lernbeispiele nennt man den *Trainingsdatensatz*. Jedes Trainingsbeispiel nennt man einen *Trainingsdatenpunkt* (oder eine *Instanz*). Der Teil eines Systems zum maschinellen Lernen, der lernt und Vorhersagen macht, wird als *Modell* bezeichnet. Neuronale Netze und Random Forests sind Beispiele für Modelle.

In diesem Fall besteht die Aufgabe *T* darin, neue E-Mails als Spam zu kennzeichnen, die Erfahrung *E* entspricht den *Trainingsdaten*. Nur das Leistungsmaß *P* ist noch zu definieren; Sie könnten z. B. den Anteil korrekt klassifizierter E-Mails verwenden. Dieses Leistungsmaß nennt man *Genauigkeit*. Es wird bei Klassifikationsaufgaben häufig verwendet.

Falls Sie gerade eine Kopie aller Artikel von Wikipedia heruntergeladen haben, verfügt Ihr Computer über eine Menge zusätzlicher Daten, verbessert sich dadurch aber bei keiner Aufgabe. Deshalb ist dies kein Machine Learning.

Warum wird Machine Learning verwendet?

Überlegen Sie einmal, wie Sie mit herkömmlichen Programmiertechniken einen Spamfilter schreiben würden (siehe Abbildung 1-1):

1. Zuerst würden Sie sich ansehen, wie Spam typischerweise aussieht. Sie würden feststellen, dass einige Wörter oder Phrasen (wie »Für Sie«, »Kreditkarte«, »kostenlos« und »erstaunlich«) in der Betreffzeile gehäuft auftreten. Möglicherweise würden Ihnen auch weitere Muster im Namen des Absenders, im Text und in anderen Teilen der E-Mail auffallen.
2. Sie würden für jedes der von Ihnen erkannten Muster einen Algorithmus schreiben, der dieses erkennt, und Ihr Programm würde E-Mails als Spam markieren, sobald eine bestimmte Anzahl dieser Muster erkannt wird.
3. Sie würden Ihr Programm testen und die Schritte 1 und 2 wiederholen, bis es gut genug ist.

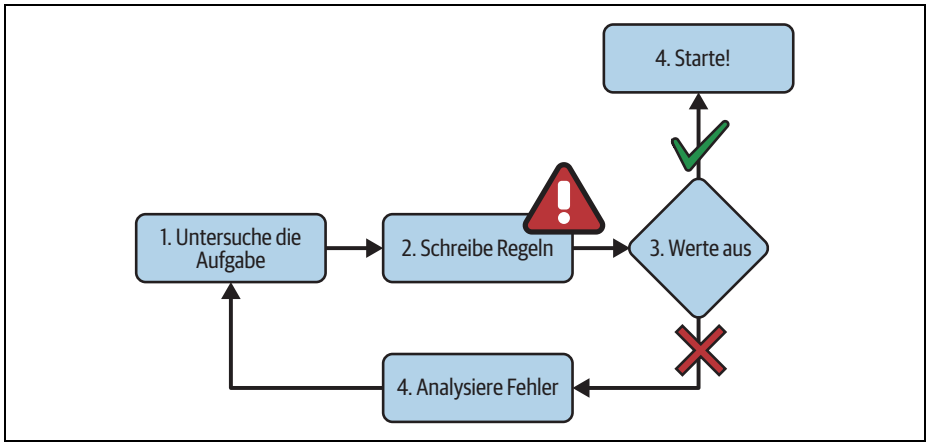

Abbildung 1-1: Die herkömmliche Herangehensweise

Da diese Aufgabe schwierig ist, wird Ihr Programm vermutlich eine lange Liste komplexer Regeln beinhalten – und ganz schön schwer zu warten sein.

Dagegen lernt ein mit Machine-Learning-Techniken entwickelter Spamfilter automatisch, welche Wörter und Phrasen Spam gut vorhersagen, indem er im Vergleich zu den Ham-Beispielen ungewöhnlich häufige Wortmuster in den Spambeispielen erkennt (siehe Abbildung 1-2). Das Programm wird viel kürzer, leichter zu warten und wahrscheinlich auch treffsicherer.

Wenn außerdem die Spammer bemerken, dass alle ihre E-Mails mit »Für Sie« geblockt werden, könnten sie stattdessen auf »4U« umsatteln. Ein mit traditionellen Programmiertechniken entwickelter Spamfilter müsste aktualisiert werden, um die E-Mails mit »4U« zu markieren. Wenn die Spammer sich ständig um Ihren Spamfilter herumarbeiten, werden Sie ewig neue Regeln schreiben müssen.

Ein auf Machine Learning basierender Spamfilter bemerkt dagegen automatisch, dass »4U« auffällig häufig in von Nutzern als Spam markierten Nachrichten vorkommt, und beginnt, diese ohne weitere Intervention auszusortieren (siehe Abbildung 1-3).

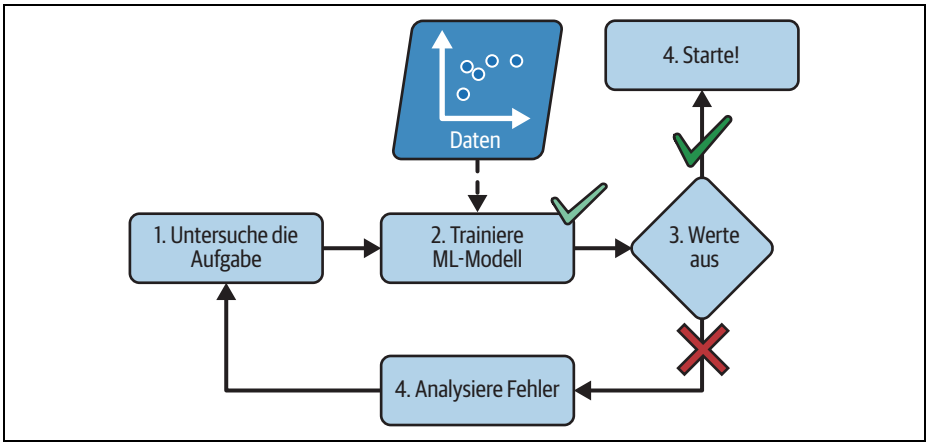

Abbildung 1-2: Der Machine-Learning-Ansatz

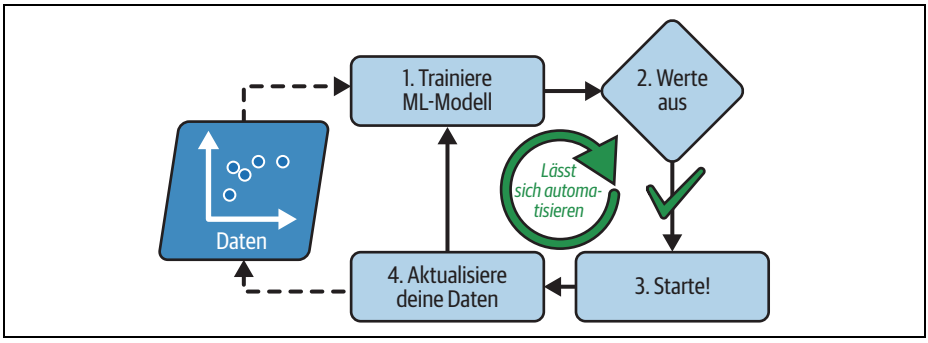

Abbildung 1-3: Automatisches Einstellen auf Änderungen

Machine Learning brilliert außerdem bei Aufgaben, die entweder zu komplex für herkömmliche Verfahren sind oder für die kein bekannter Algorithmus existiert. Betrachten Sie z. B. Spracherkennung: Angenommen, Sie beginnen mit einer einfachen Aufgabe und schreiben ein Programm, das die Wörter »one« und »two« unterscheiden kann. Sie bemerken, dass das Wort »two« mit einem hochfrequenten Ton (»T«) beginnt, und könnten demnach einen Algorithmus hartcodieren, der die Intensität hochfrequenter Töne misst und dadurch »one« und »two« unterscheiden kann. Natürlich skaliert diese Technik nicht auf Tausende Wörter, die von Millionen von Menschen mit Hintergrundgeräuschen und in Dutzenden von Sprachen gesprochen werden. Die (zumindest heutzutage) beste Möglichkeit ist, einen Algo-

rithmus zu schreiben, der eigenständig aus vielen aufgenommenen Beispielen für jedes Wort lernt.

Schließlich kann Machine Learning auch Menschen beim Lernen unterstützen (siehe Abbildung 1-4): ML-Modelle lassen sich untersuchen, um zu erkennen, was sie gelernt haben (auch wenn dies bei manchen Modellen kompliziert sein kann). Wenn z. B. der Spamfilter erst einmal mit genug Spam trainiert wurde, lässt sich die Liste von Wörtern und Wortkombinationen inspizieren, die als gute Merkmale von Spam erkannt wurden. Manchmal kommen dabei überraschende Korrelationen oder neue Trends ans Tageslicht und führen dadurch zu einem besseren Verständnis der Aufgabe. Das Durchwühlen großer Datenmengen zum Entdecken nicht unmittelbar ersichtlicher Muster nennt man *Data Mining*, und Machine Learning ist sehr gut darin.

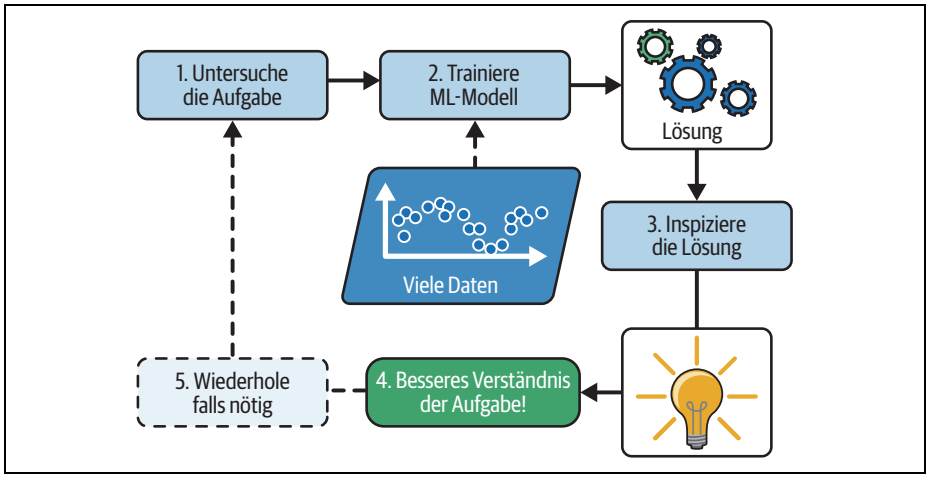

Abbildung 1-4: Machine Learning hilft Menschen beim Lernen.

Zusammengefasst, ist Machine Learning hervorragend geeignet für:

- Aufgaben, bei denen die existierenden Lösungen eine Menge Feinarbeit oder lange Listen von Regeln erfordern (ein Machine-Learning-Modell vereinfacht oft den Code und schneidet besser ab als der klassische Ansatz),
- komplexe Aufgaben, für die es mit herkömmlichen Methoden überhaupt keine gute Lösung gibt (die besten Machine-Learning-Techniken können vielleicht eine Lösung finden),
- fluktuierende Umgebungen (ein Machine-Learning-System kann leicht mit neuen Daten trainiert und somit immer aktuell gehalten werden) sowie
- um Erkenntnisse über komplexe Aufgabenstellungen und große Datenmengen zu gewinnen.

Anwendungsbeispiel

Schauen wir uns ein paar konkrete Beispiele für Aufgaben des Machine Learning und die dabei eingesetzten Techniken an:

Produktbilder in der Herstellung analysieren, um sie automatisch zu klassifizieren
: Dies ist Bildklassifikation, meist ausgeführt durch Convolutional Neural Networks (CNNs, siehe Kapitel 14) oder manchmal Transformer (siehe Kapitel 16).

Tumoren in Gehirnscans erkennen
: Das ist eine semantische Bildsegmentierung, bei der jedes Pixel im Bild klassifiziert wird (da wir die genaue Position und Form des Tumors bestimmen wollen), meist ebenfalls mithilfe von CNNs oder Transformern.

Nachrichtenartikel automatisch klassifizieren
: Dies ist Verarbeitung natürlicher Sprache (NLP, Natural Language Processing) und, spezifischer, Textklassifikation, die sich über rekurrente neuronale Netze (RNNs) und CNNs angehen lässt, wobei Transformer sogar noch besser funktionieren (siehe Kapitel 16).

Beleidigende Kommentare in Diskussionsforen automatisch markieren
: Dabei handelt es sich ebenfalls um Textklassifikation mit den gleichen NLP-Tools.

Automatisch lange Dokumente zusammenfassen
: Dies ist ein Zweig der NLP namens Textextrahierung, ebenfalls mit den gleichen Tools.

Einen Chatbot oder persönlichen Assistenten erstellen
: Dazu gehören viele NLP-Komponenten, unter anderem das Verstehen natürlicher Sprache (Natural Language Understanding, NLU) und Module zum Beantworten von Fragen.

Den Umsatz Ihrer Firma für das nächste Jahr basierend auf vielen Performancemetriken vorhersagen
: Dies ist eine Regressionsaufgabe (also das Vorhersagen von Werten), die über ein Regressionsmodell wie ein lineares oder polynomielles Regressionsmodell (siehe Kapitel 4), eine Regressions-SVM (siehe Kapitel 5), einen Regressions-Random-Forest (siehe Kapitel 7) oder ein künstliches neuronales Netzwerk (siehe Kapitel 10) angegangen werden kann. Wollen Sie Zeitreihen vergangener Metriken mit einbeziehen, können Sie RNNs, CNNs oder Transformer nutzen (siehe die Kapitel 15 und 16).

Kreditkartenmissbrauch erkennen
: Dies ist Anomalieerkennung, die über Isolation Forests, gaußsche Mischverteilungsmodelle (siehe Kapitel 9) oder Autoencoder umgesetzt wird (siehe Kapitel 17).

Kunden anhand ihrer Einkäufe segmentieren, sodass Sie für jeden Bereich unterschiedliche Marketingstrategien entwerfen können

Das ist Clustering, das sich über *k*-Means, DBSCAN oder anderes erreichen lässt (siehe Kapitel 9).

Einen komplexen, hochdimensionalen Datensatz in einem klaren und verständlichen Diagramm darstellen

Hier geht es um Datenvisualisierung, meist unter Verwendung von Techniken zur Datenreduktion (siehe Kapitel 8).

Einem Kunden basierend auf dessen bisherigen Käufen ein Produkt empfehlen, das ihn interessieren könnte

Dies ist ein Empfehlungssystem. Ein Ansatz ist, Käufe aus der Vergangenheit (und andere Informationen über den Kunden) in ein künstliches neuronales Netzwerk einzuspeisen (siehe Kapitel 10) und es dazu zu bringen, den wahrscheinlichsten nächsten Kauf auszugeben. Dieses neuronale Netzwerk würde typischerweise mit bisherigen Abfolgen von Käufen aller Kunden trainiert werden.

Einen intelligenten Bot für ein Spiel bauen

Dies wird oft durch Reinforcement Learning (RL, siehe Kapitel 18) angegangen. Dabei handelt es sich um einen Zweig des Machine Learning, der Agenten trainiert (zum Beispiel Bots), um die Aktionen auszuwählen, die mit der Zeit in einer gegebenen Umgebung (wie dem Spiel) ihre Belohnungen maximieren (beispielsweise kann ein Bot immer dann eine Belohnung erhalten, wenn der Spieler Lebenspunkte verliert). Das berühmte AlphaGo-Programm, das den Weltmeister im Go geschlagen hat, wurde mithilfe von RL gebaut.

Diese Liste könnte immer weiter fortgeführt werden, aber hoffentlich haben Sie auch so schon einen Eindruck von der unglaublichen Breite und Komplexität der Aufgaben erhalten, die Machine Learning angehen kann, und die Art von Techniken, die Sie dafür verwenden würden.

Unterschiedliche Machine-Learning-Systeme

Es gibt so viele verschiedene Arten von Machine-Learning-Systemen, dass es hilfreich ist, die Verfahren nach folgenden Kriterien in grobe Kategorien einzuteilen:

- Wie sie während des Trainings überwacht werden (überwachtes, unüberwachtes, teilüberwachtes, selbstüberwachtes Lernen und anderes).
- Ob sie inkrementell dazulernen können oder nicht (Online-Learning gegenüber Batch-Learning).
- Ob sie einfach neue Datenpunkte mit den bereits bekannten Datenpunkten vergleichen oder stattdessen Muster in den Trainingsdaten erkennen, um ein Vorhersagemodell aufzubauen, wie es auch Wissenschaftler tun (instanzbasiertes gegenüber modellbasiertem Lernen).

Diese Kriterien schließen sich nicht gegenseitig aus; sie lassen sich beliebig miteinander kombinieren. Zum Beispiel kann ein moderner Spamfilter ständig mit einem neuronalen Netzwerkmodell mit durch Menschen bereitgestellten Beispielen für Spam und Ham dazulernen; damit ist er ein modellbasiertes, überwachtes Onlinelernsystem. Betrachten wir jedes dieser Kriterien etwas genauer.

Trainingsüberwachung

Machine-Learning-Systeme lassen sich entsprechend der Menge und Art der Überwachung beim Trainieren einordnen. Es gibt dabei viele Kategorien, aber wir werden nur die wichtigsten behandeln: überwachtes Lernen, unüberwachtes Lernen, selbstüberwachtes Lernen, teilüberwachtes Lernen und verstärkendes Lernen (Reinforcement Learning).

Überwachtes Lernen

Beim *überwachten Lernen* enthalten die dem Algorithmus gelieferten Trainingsdaten die gewünschten Lösungen, genannt *Labels* (siehe Abbildung 1-5).

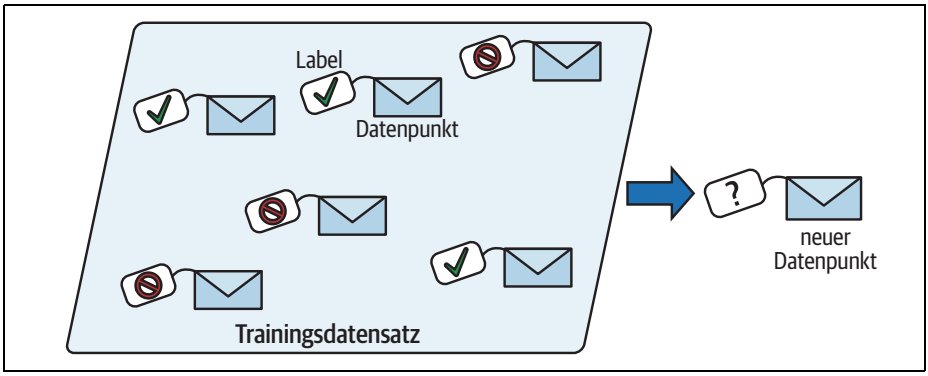

Abbildung 1-5: Ein Trainingsdatensatz mit Labels zur Spamerkennung (ein Beispiel für überwachtes Lernen)

Klassifikation ist eine typische überwachte Lernaufgabe. Spamfilter sind hierfür ein gutes Beispiel: Sie werden mit vielen Beispiel-E-Mails und deren *Kategorie* (Spam oder Ham) trainiert und müssen lernen, neue E-Mails zu klassifizieren.

Eine weitere typische Aufgabe ist, eine numerische *Zielgröße* vorherzusagen, wie etwa den Preis eines Autos auf Grundlage gegebener *Merkmale* (gefahrene Kilometer, Alter, Marke und so weiter), den sogenannten *Prädiktoren*. Diese Art Aufgabe bezeichnet man als *Regression* (siehe Abbildung 1-6).[1] Um das System zu trainie-

1 Wissenswert: Dieser seltsam anmutende Begriff wurde von Francis Galton in die Statistik eingeführt, der beobachtete, dass die Kinder hochgewachsener Eltern zu einer geringeren Körpergröße als ihre Eltern neigen. Da die Kinder kleiner waren, nannte er dies *Regression zum Mittelwert*. Der Name wurde anschließend auch für seine Methode zur Analyse von Korrelationen zwischen Variablen verwendet.

ren, benötigt es viele Beispielfahrzeuge mitsamt ihren Merkmalen und Targets (also den Preisen).

Viele Regressionsalgorithmen lassen sich auch zur Klassifikation einsetzen und umgekehrt. Zum Beispiel ist die *logistische Regression* eine verbreitete Methode für Klassifikationsaufgaben, da sie die Wahrscheinlichkeit der Zugehörigkeit zu einer bestimmten Kategorie als Ergebnis liefert (z. B. 20%ige Chance für Spam).

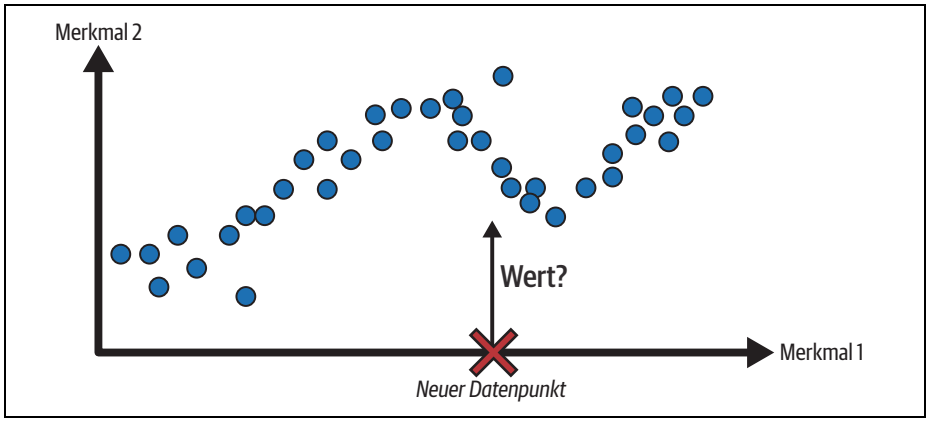

Abbildung 1-6: Ein Regressionsproblem: Sage mithilfe eines Eingangsmerkmals einen Wert voraus (es gibt meist mehrere Eingangsmerkmale und manchmal auch mehrere Ausgangsmerkmale).

Die Begriffe *Target* und *Label* werden im überwachten Lernen im Allgemeinen synonym behandelt, aber *Target* ist bei Regressionsaufgaben gebräuchlicher, während man bei Klassifizierungsaufgaben eher von *Labels* spricht. *Merkmale* werden auch manchmal als *Prädiktoren* oder *Attribute* bezeichnet. Diese Begriffe können sich auf einzelne Datenwerte beziehen (zum Beispiel »das Merkmal ›Kilometerstand‹ dieses Autos entspricht 15.000«) oder aber auf alle Datenwerte (zum Beispiel »das Merkmal ›Kilometerstand‹ ist stark mit dem Preis korreliert«).

Unüberwachtes Lernen

Beim *unüberwachten Lernen*sind die Trainingsdaten, wie der Name vermuten lässt, nicht gelabelt (siehe Abbildung 1-7). Das System versucht, ohne Anleitung zu lernen.

Nehmen wir an, Sie hätten eine Menge Daten über Besucher Ihres Blogs. Sie möchten einen *Clustering-Algorithmus* verwenden, um Gruppen ähnlicher Besucher zu entdecken (siehe Abbildung 1-8). Sie verraten dem Algorithmus nichts darüber, welcher Gruppe ein Besucher angehört, er findet die Verbindungen ohne Ihr Zutun heraus. Beispielsweise könnte der Algorithmus bemerken, dass 40% Ihrer Besucher Teenager mit einer Vorliebe für Comics sind, die Ihr Blog nach der Schule lesen, 20% dagegen sind erwachsene Science-Fiction-Fans, die am Wochenende vorbeischauen. Wenn Sie ein *hierarchisches Cluster-Verfahren* verwenden, können Sie so-

gar jede Gruppe in weitere Untergruppen zerlegen, was hilfreich sein kann, wenn Sie Ihre Blogartikel auf diese Zielgruppen zuschneiden möchten.

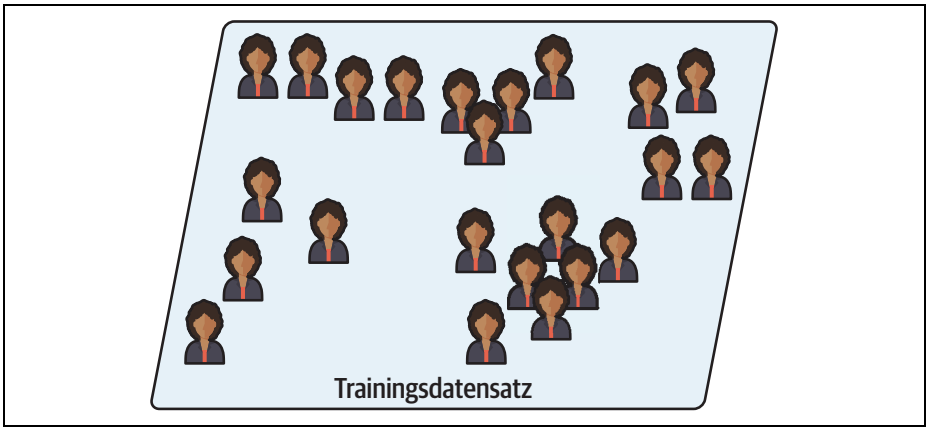

Abbildung 1-7: Ein Trainingsdatensatz ohne Labels für unüberwachtes Lernen

Algorithmen zur *Visualisierung* sind ebenfalls ein gutes Beispiel für unüberwachtes Lernen: Sie übergeben diesen eine Menge komplexer Daten ohne Labels und erhalten eine 2-D- oder 3-D-Repräsentation der Daten, die Sie leicht grafisch darstellen können (siehe Abbildung 1-9).

Solche Algorithmen versuchen, die Struktur der Daten so gut wie möglich zu erhalten (z. B. Cluster in den Eingabedaten am Überlappen in der Visualisierung zu hindern), sodass Sie leichter verstehen können, wie die Daten aufgebaut sind, und womöglich auf unvermutete Muster stoßen.

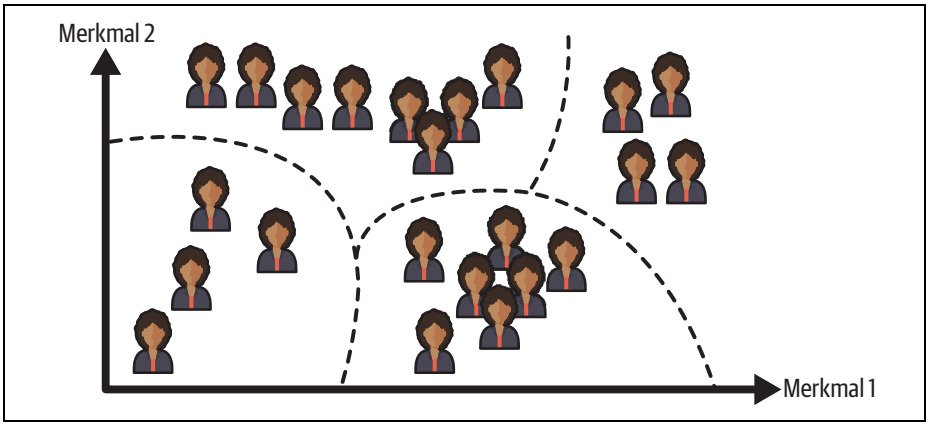

Abbildung 1-8: Clustering

Eine verwandte Aufgabe ist die *Dimensionsreduktion*, bei der das Ziel die Vereinfachung der Daten ist, ohne dabei allzu viele Informationen zu verlieren. Dazu lassen sich mehrere korrelierende Merkmale zu einem vereinigen. Beispielsweise korreliert

der Kilometerstand eines Autos stark mit seinem Alter, daher kann ein Algorithmus zur Dimensionsreduktion beide zu einem Merkmal verbinden, das die Abnutzung des Fahrzeugs repräsentiert. Dies nennt man auch *Extraktion von Merkmalen*.

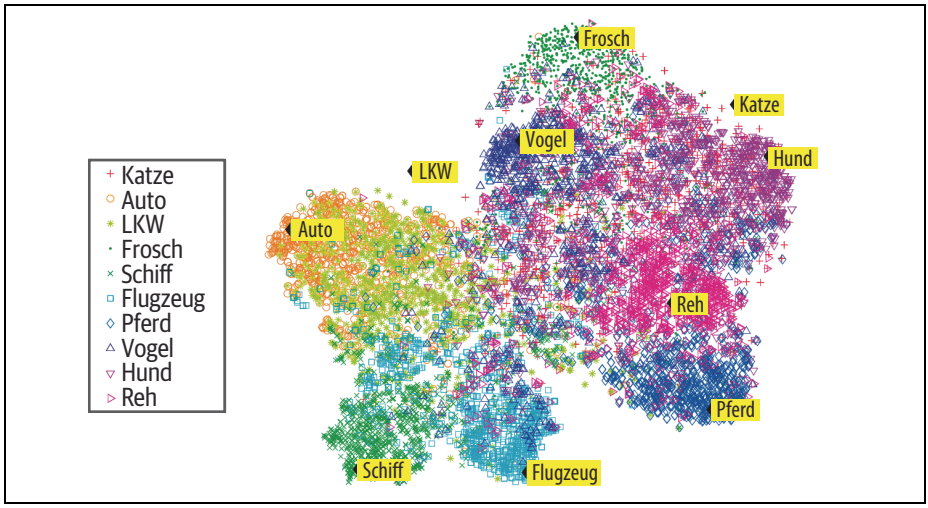

Abbildung 1-9: Beispiel für eine t-SNE-Visualisierung semantischer Cluster[2]

Meist ist es eine gute Idee, die Dimensionen Ihrer Trainingsdaten zu reduzieren, bevor Sie sie in einen anderen Machine-Learning-Algorithmus einspeisen (wie etwa einen überwachten Lernalgorithmus). Er wird viel schneller arbeiten, und die Daten beanspruchen weniger Platz auf der Festplatte und im Speicher. In manchen Fällen ist auch das Ergebnis besser.

Eine weitere wichtige unüberwachte Aufgabe ist das *Erkennen von Anomalien* – beispielsweise ungewöhnliche Transaktionen auf Kreditkarten, die auf Betrug hindeuten, das Abfangen von Produktionsfehlern oder das automatische Entfernen von Ausreißern aus einem Datensatz, bevor dieser in einen weiteren Lernalgorithmus eingespeist wird. Dem System werden beim Training vor allem gewöhnliche Datenpunkte präsentiert, sodass es lernt, sie zu erkennen. Sieht es dann einen neuen Datenpunkt, kann es entscheiden, ob dieser wie ein normaler Punkt oder eine Anomalie aussieht (siehe Abbildung 1-10). Eine sehr ähnliche Aufgabe ist die *Novelty Detection*: Ihr Ziel ist es, neue Instanzen zu erkennen, die anders als alle anderen Instanzen im Trainingsdatensatz aussehen. Dafür brauchen Sie einen sehr »sauberen« Trainingsdatensatz, der von allen Instanzen befreit ist, die der Algorithmus erkennen soll. Haben Sie beispielsweise Tausende von Bildern mit Hunden

2 Beachten Sie, dass die Tiere recht gut von Fahrzeugen unterschieden werden und wie nahe sich Pferde und Rehe sind, aber wie weit entfernt von Vögeln. Abbildung mit Erlaubnis reproduziert, Quelle: Richard Socher et al., »Zero-Shot Learning Through Cross-Modal Transfer«, *Proceedings of the 26th International Conference on Neural Information Processing Systems* 1 (2013): 935–943.

und sind nur auf 1 % dieser Bilder Chihuahuas zu sehen, sollte ein Algorithmus zur Novelty Detection neue Bilder von Chihuahuas nicht als Besonderheiten behandeln. Ein Algorithmus zur Anomalieerkennung mag diese Hunde allerdings als so selten ansehen – und als so anders als andere Hunde –, dass er sie als Anomalien klassifizieren würde (nichts gegen Chihuahuas).

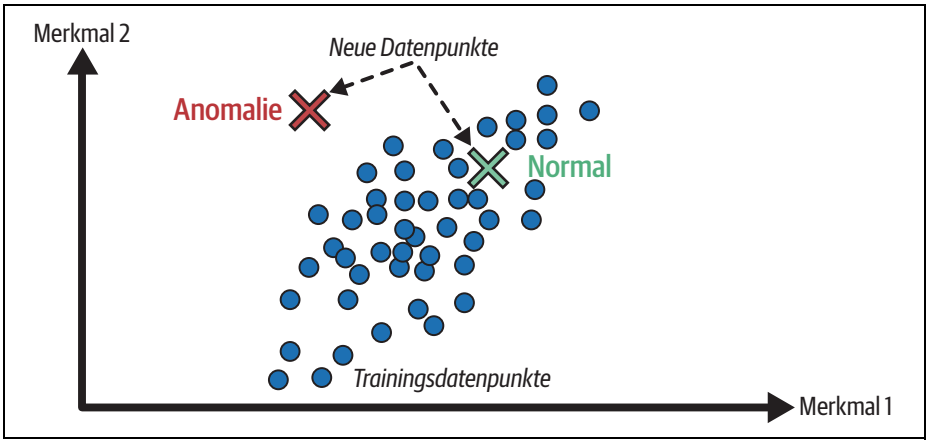

Abbildung 1-10: Erkennen von Anomalien

Schließlich ist auch das *Lernen von Assoziationsregeln* eine verbreitete unüberwachte Lernaufgabe, bei der das Ziel ist, in große Datenmengen einzutauchen und interessante Beziehungen zwischen Merkmalen zu entdecken. Wenn Sie beispielsweise einen Supermarkt führen, könnten Assoziationsregeln auf Ihren Verkaufsdaten ergeben, dass Kunden, die Grillsoße und Kartoffelchips einkaufen, tendenziell auch Steaks kaufen. Daher sollten Sie diese Artikel in unmittelbarer Nähe zueinander platzieren.

Teilüberwachtes Lernen

Da das Labeling normalerweise zeitaufwendig und teuer ist, werden Sie oftmals sehr viele ungelabelte und wenige gelabelte Instanzen haben. Einige Algorithmen können mit nur teilweise gelabelten Trainingsdaten arbeiten. Dies bezeichnet man als *teilüberwachtes Lernen* (siehe Abbildung 1-11).

Einige Fotodienste wie Google Photos bieten hierfür ein gutes Beispiel. Sobald Sie all Ihre Familienfotos in den Dienst hochgeladen haben, erkennt dieser automatisch, dass die gleiche Person A auf den Fotos 1, 5 und 11 vorkommt, während Person B auf den Fotos 2, 5 und 7 zu sehen ist. Dies ist der unüberwachte Teil des Algorithmus (Clustering). Nun muss das System nur noch wissen, wer diese Personen

sind. Ein Label pro Person[3] genügt, um jede Person in jedem Foto zuzuordnen, was bei der Suche nach Fotos äußerst nützlich ist.

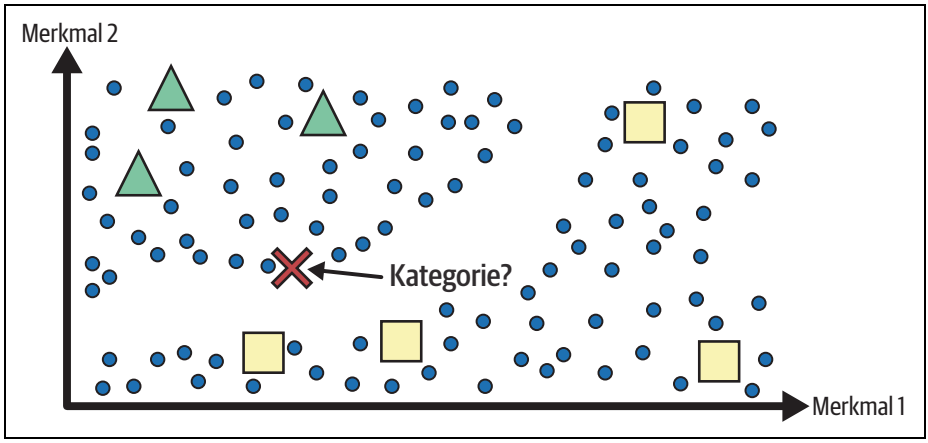

Abbildung 1-11: Teilüberwachtes Lernen mit zwei Klassen (Dreiecken und Quadraten): Die ungelabelten Beispiele (Kreise) helfen dabei, eine neue Instanz (das Kreuz) in die Dreiecksklasse statt in die Quadratklasse einzuordnen, auch wenn sie näher an den gelabelten Quadraten ist.

Die meisten Algorithmen für teilüberwachtes Lernen sind Kombinationen aus unüberwachten und überwachten Verfahren. So kann beispielsweise ein Clustering-Algorithmus genutzt werden, um ähnliche Instanzen zu gruppieren; danach werden alle nicht gelabelten Instanzen mit dem am häufigsten vorkommenden Label im Cluster ausgezeichnet. Ist der gesamte Datensatz gelabelt, kann man beliebige überwachte Lernalgorithmen verwenden.

Selbstüberwachtes Lernen

Ein weiterer Ansatz beim Machine Learning beinhaltet tatsächlich das Generieren eines vollständig gelabelten Datensatzes aus einem vollständig ungelabelten Datensatz. Auch hier gilt: Ist der gesamte Datensatz mit Labels versehen, kann jeder überwachte Lernalgorithmus damit verwendet werden. Dieser Ansatz wird als *selbstüberwachtes Lernen* bezeichnet.

Haben Sie beispielsweise einen großen Datensatz mit ungelabelten Bildern, können Sie zufällig einen kleinen Teil jedes Bilds abdecken und dann ein Modell darin trainieren, das ursprüngliche Bild wiederherzustellen (Abbildung 1-12). Während des Trainings werden die maskierten Bilder als Eingaben für das Modell genutzt, und die ursprünglichen Bilder dienen als Label.

3 Dies ist der Fall, wenn das System perfekt funktioniert. In der Praxis werden meist einige Cluster pro Person erstellt. Manchmal werden zwei ähnliche Personen verwechselt, sodass Sie einige Labels pro Person angeben und außerdem ein paar Cluster von Hand aufräumen müssen.

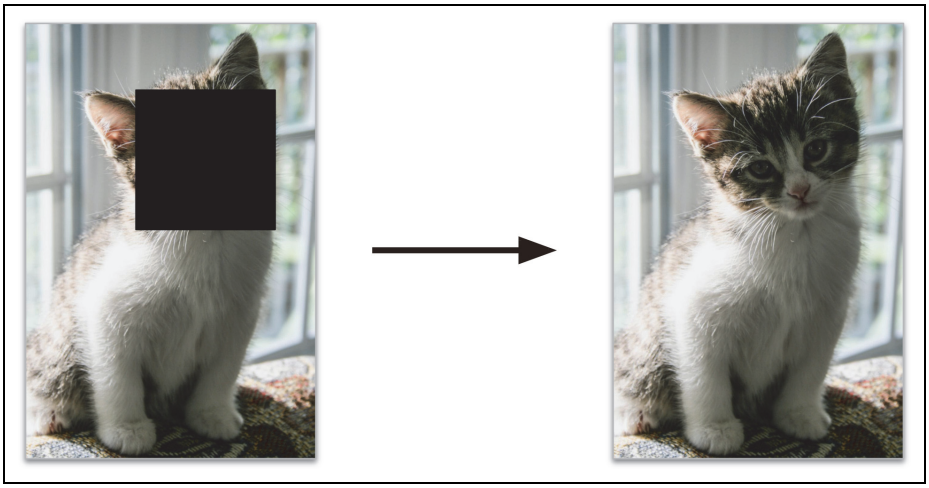

Abbildung 1-12: Beispiel für selbstüberwachtes Lernen: Eingabe (links) und Target (rechts)

Das sich ergebende Modell kann schon für sich allein ziemlich nützlich sein – etwa um beschädigte Bilder zu reparieren oder unerwünschte Objekte aus Bildern zu entfernen. Aber meist ist ein Modell, das mit selbstüberwachtem Lernen trainiert wurde, nicht das wirkliche Ziel. Sie werden es im Allgemeinen noch für eine etwas andere Aufgabe anpassen und feintunen wollen – eine Aufgabe, die Sie eigentlich lösen wollen.

Stellen Sie sich zum Beispiel vor, dass Ihr Ziel ein Modell zum Klassifizieren von Haustieren ist: Mit einem Bild eines beliebigen Haustiers erfahren Sie, welche Spezies das ist. Haben Sie einen großen Datensatz mit nicht gelabelten Fotos von Haustieren, können Sie beginnen, indem Sie ein Bildreparaturmodell mithilfe von selbstüberwachtem Lernen trainieren. Funktioniert das gut, sollte es dazu in der Lage sein, die verschiedenen Haustierspezies unterscheiden zu können: Repariert es ein Bild einer Katze, deren Gesicht verdeckt ist, darf es kein Gesicht eines Hundes einfügen. Sofern die Architektur des Modells dies erlaubt (und die meisten Architekturen neuronaler Netze tun das), ist es dann möglich, das Modell so anzupassen, dass es Haustierspezies vorhersagt, statt Bilder zu reparieren. Der letzte Schritt besteht darin, das Modell mit einem gelabelten Datensatz noch zu optimieren: Es weiß schon, wie Katzen, Hunde und andere Haustiere aussehen, daher ist dieser Schritt nur erforderlich, damit das Modell die Abbildung zwischen den schon bekannten Spezies und den erwarteten Labels vornehmen kann.

Das Übertragen von Wissen von einer Aufgabe auf eine andere wird als *Transfer Learning* bezeichnet, und es ist eine der wichtigsten Techniken des modernen Machine Learning – insbesondere wenn Deep Neural Networks zum Einsatz kommen (also neuronale Netze aus viele Neuronenschichten). Wir werden darauf detaillierter in Teil II eingehen.

Manche sehen das selbstüberwachte Lernen als Teil des unüberwachten Lernens an, da es mit vollständig ungelabelten Datensätzen arbeitet. Aber selbstüberwachtes Lernen nutzt während des Trainings (generierte) Labels, daher liegt es dahin gehend näher am überwachten Lernen. Und der Begriff »unüberwachtes Lernen« wird im Allgemeinen genutzt, wenn man mit Aufgaben wie Clustering, Dimensionsreduktion oder Anomalieerkennung zu tun hat, während sich das selbstüberwachte Lernen auf die gleichen Aufgaben wie überwachtes Lernen konzentriert – also vor allem Klassifikation und Regression. Kurz gesagt, es ist am besten, selbstüberwachtes Lernen als eigene Kategorie anzusehen.

Reinforcement Learning

Reinforcement Learning ist etwas völlig anderes. Das Lernsystem, in diesem Zusammenhang als *Agent* bezeichnet, beobachtet eine Umgebung, wählt Aktionen und führt diese aus. Dafür erhält es *Belohnungen* (oder *Strafen* in Form negativer Belohnungen wie in Abbildung 1-13). Das System muss selbst herausfinden, was die beste Strategie oder *Policy* ist, um mit der Zeit die meisten Belohnungen zu erhalten. Eine Policy definiert, welche Aktion der Agent in einer gegebenen Situation auswählt.

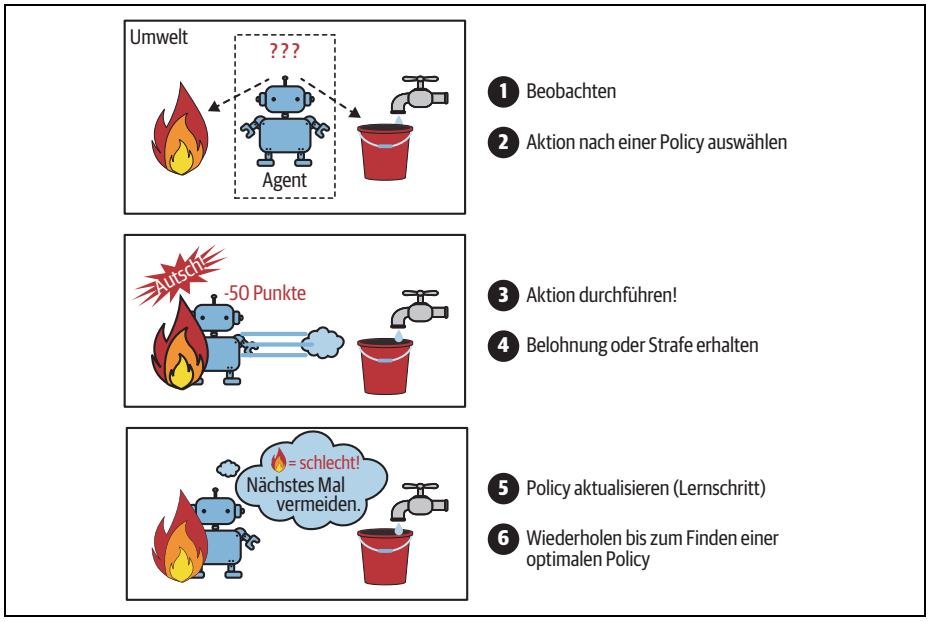

Abbildung 1-13: Reinforcement Learning

Beispielsweise verwenden viele Roboter Reinforcement-Learning-Algorithmen, um laufen zu lernen. Auch das Programm AlphaGo von DeepMind ist ein gutes Beispiel für Reinforcement Learning: Es geriet im Mai 2017 in die Schlagzeilen, als es den damaligen Weltranglistenersten Ke Jie im Brettspiel *Go* schlug. AlphaGo er-

lernte die zum Sieg führende Policy, indem es Millionen von Partien analysierte und anschließend viele Spiele gegen sich selbst spielte. Beachten Sie, dass das Lernen während der Partien gegen den Weltmeister abgeschaltet war; AlphaGo wandte nur die bereits erlernte Policy an. Wie Sie im nächsten Abschnitt sehen werden, wird das *Offline-Learning* genannt.

Batch- und Online-Learning

Ein weiteres Kriterium zum Einteilen von Machine-Learning-Systemen ist, ob das System aus einem kontinuierlichen Datenstrom inkrementell lernen kann.

Batch-Learning

Beim *Batch-Learning* kann das System nicht inkrementell lernen, es muss mit sämtlichen verfügbaren Daten trainiert werden. Dies dauert meist lange und beansprucht Rechenkapazitäten. Es wird daher in der Regel offline durchgeführt. Zuerst wird das System trainiert und anschließend in einer Produktivumgebung eingesetzt, wo es ohne weiteres Lernen läuft; es wendet lediglich das bereits Erlernte an. Dies nennt man *Offline-Learning*.

Leider tendiert die Performance eines Modells dazu, mit der Zeit schlechter zu werden – ganz einfach, weil sich die Welt drumherum weiterentwickelt, während das Modell stehen bleibt. Dieses Phänomen wird oft als *Model Rot* oder *Data Drift* bezeichnet. Die Lösung besteht darin, das Modell regelmäßig mit aktuellen Daten zu trainieren. Wie oft Sie das tun müssen, hängt vom Anwendungsfall ab: Klassifiziert das Modell Bilder von Katzen und Hunden, wird seine Qualität sehr langsam schlechter werden, aber wenn es mit sich schnell entwickelnden Systemen zu tun hat – zum Beispiel Vorhersagen für den Finanzmarkt –, wird es vermutlich sehr schnell veralten.

Auch ein Modell, das dafür trainiert wurde, Bilder von Katzen und Hunden zu klassifizieren, muss eventuell regelmäßig neu trainiert werden. Nicht etwa, weil Katzen und Hunde über Nacht mutieren, sondern weil sich Kameras und Bildformate ändern und Bildschärfe, Helligkeit und Seitenverhältnisse anders werden. Zudem sind im kommenden Jahr vielleicht andere Rassen in Mode, oder die Menschen entscheiden sich dazu, ihre Haustiere mit kleinen Hütchen auszustatten – wer weiß das schon?

Wenn Sie möchten, dass ein Batch-Learning-System etwas über neue Daten erfährt (beispielsweise neuartigen Spam), müssen Sie eine neue Version des Systems ein weiteres Mal mit dem gesamten Datensatz trainieren (nicht einfach nur den neuen Datensatz, sondern auch den alten). Anschließend müssen Sie das alte System anhalten und durch das neue ersetzen.

Glücklicherweise lässt sich der gesamte Prozess aus Training, Evaluation und Inbetriebnahme eines Machine-Learning-Systems recht leicht automatisieren (wie in

Abbildung 1-3 gezeigt). So kann sich selbst ein Batch-Learning-System anpassen. Aktualisieren Sie einfach die Daten und trainieren Sie eine neue Version des Systems so oft wie nötig.

Dies ist eine einfache Lösung und funktioniert meist gut, aber das Trainieren mit dem gesamten Datensatz kann viele Stunden beanspruchen. Daher würde man das neue System nur alle 24 Stunden oder wöchentlich trainieren. Wenn Ihr System sich an schnell ändernde Daten anpassen muss (z. B. um Aktienkurse vorherzusagen), benötigen Sie eine anpassungsfähigere Lösung.

Außerdem beansprucht das Trainieren auf dem gesamten Datensatz eine Menge Rechenkapazität (CPU, Hauptspeicher, Plattenplatz, I/O-Kapazität, Netzwerkbandbreite und so weiter). Wenn Sie eine Menge Daten haben und Ihr System automatisch jeden Tag trainieren lassen, kann Sie das am Ende eine Stange Geld kosten. Falls die Datenmenge sehr groß ist, kann der Einsatz von Batch-Learning sogar unmöglich sein.

Wenn Ihr System autonom lernen muss und die Ressourcen dazu begrenzt sind (z. B. eine Applikation auf einem Smartphone oder ein Fahrzeug auf dem Mars), ist das Herumschleppen großer Mengen an Trainingsdaten oder das Belegen einer Menge Ressourcen für mehrere Stunden am Tag kein gangbarer Weg.

In all diesen Fällen sind Algorithmen, die inkrementell lernen können, eine bessere Alternative.

Online-Learning

Beim *Online-Learning*wird das System nach und nach trainiert, indem einzelne Datensätze nacheinander oder in kleinen Paketen, sogenannten *Mini-Batches*, hinzugefügt werden. Jeder Lernschritt ist schnell und billig, sodass das System aus neuen Daten lernen kann, sobald diese verfügbar sind (siehe Abbildung 1-14).

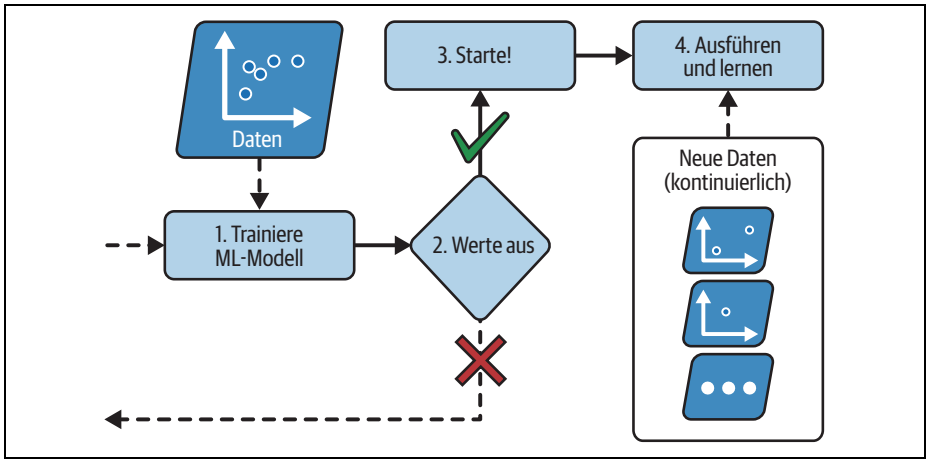

Abbildung 1-14: Beim Online-Learning wird ein Modell trainiert und in den Produktivbetrieb übernommen, wo es mit neu eintreffenden Daten weiterlernt.

Online-Learning eignet sich gut für ein System, das sich sehr schnell an Veränderungen anpassen muss (zum Beispiel um neue Muster im Aktienmarkt zu erkennen). Auch wenn Ihnen nur begrenzte Rechenkapazitäten zur Verfügung stehen, ist es eine sinnvolle Option – etwa wenn das Modell auf einem mobilen Device trainiert wird.

Algorithmen zum Online-Learning lassen sich ebenfalls zum Trainieren von Modellen mit riesigen Datensätzen einsetzen, die nicht in den Hauptspeicher eines Rechners passen (dies nennt man auch *Out-of-Core*-Lernen). Der Algorithmus lädt einen Teil der Daten, führt einen Trainingsschritt auf den Daten aus und wiederholt den Prozess, bis er sämtliche Daten verarbeitet hat (siehe Abbildung 1-15).

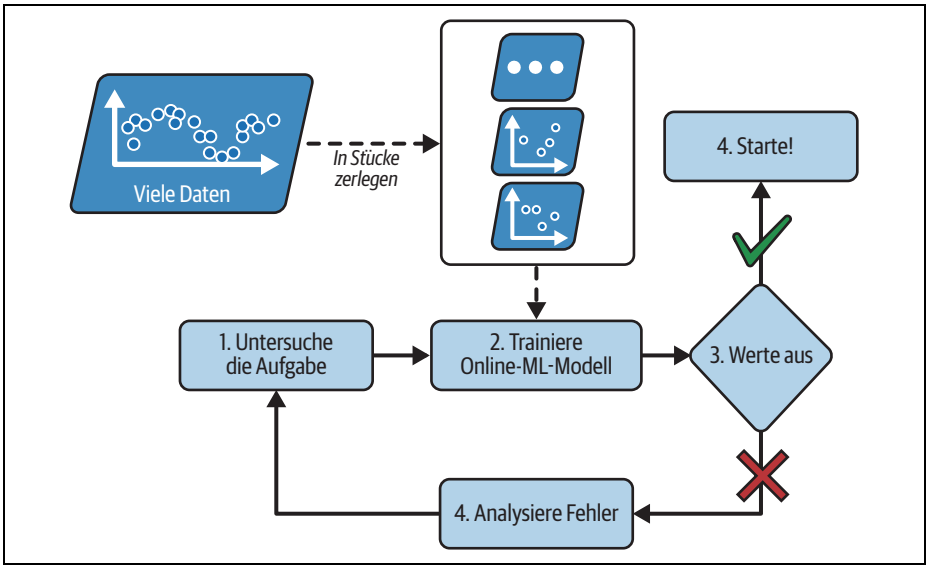

Abbildung 1-15: Verwenden von Online-Learning zum Bewältigen riesiger Datensätze

Ein wichtiger Parameter bei Online-Learning-Systemen ist, wie schnell sie sich an sich verändernde Daten anpassen. Man spricht hier von der *Lernrate*. Wenn Sie die Lernrate hoch ansetzen, wird sich Ihr System schnell auf neue Daten einstellen, aber die alten Daten auch leicht wieder vergessen (Sie möchten sicher nicht, dass ein Spamfilter nur die zuletzt gesehenen Arten von Spam erkennt). Wenn Sie die Lernrate dagegen niedrig ansetzen, entwickelt das System eine höhere Trägheit; das bedeutet, es lernt langsamer, ist aber auch weniger anfällig für Rauschen in den neuen Daten oder für Folgen nicht repräsentativer Datenpunkte (Outlier).

Out-of-Core-Lernen wird für gewöhnlich offline durchgeführt (also nicht auf einem Produktivsystem), daher ist der Begriff *Online-Learning* etwas irreführend. Stellen Sie sich darunter eher *inkrementelles Lernen* vor.

Eine große Herausforderung beim Online-Learning besteht darin, dass in das System eingespeiste minderwertige Daten zu einer Verschlechterung seiner Qualität führen, eventuell sogar sehr zügig (abhängig von der Datenqualität und der Lernrate). Wenn es sich dabei um ein Produktivsystem handelt, werden Ihre Kunden dies bemerken. Beispielsweise könnten minderwertige Daten von einem Fehler stammen (wie einem fehlerhaften Sensor an einem Roboter) oder auch von jemandem, der versucht, das System auszutricksen (indem er zum Beispiel sein Ranking in einer Suchmaschine durch massenhafte Anfragen zu verbessern versucht). Um dieses Risiko zu reduzieren, müssen Sie Ihr System aufmerksam beobachten und den Lernprozess beherzt abschalten (und eventuell auf einen früheren Zustand zurücksetzen), sobald Sie einen Qualitätsabfall bemerken. Sie können auch die Eingabedaten verfolgen und auf ungewöhnliche Daten reagieren (z. B. über einen Algorithmus zur Erkennung von Anomalien).

Instanzbasiertes versus modellbasiertes Lernen

Eine weitere Möglichkeit, maschinelle Lernverfahren zu kategorisieren, ist die Art, wie diese *verallgemeinern*. Bei den meisten Aufgaben im Machine Learning geht es um das Treffen von Vorhersagen. Dabei muss ein System in der Lage sein, aus einer Anzahl von Trainingsbeispielen auf nie zuvor gesehene Beispiele zu verallgemeinern. Es ist hilfreich, aber nicht ausreichend, eine gute Qualität auf den Trainingsdaten zu erzielen; das wirkliche Ziel ist, eine gute Qualität auf neuen Datenpunkten zu erreichen.

Es gibt beim Verallgemeinern zwei Ansätze: instanzbasiertes Lernen und modellbasiertes Lernen.

Instanzbasiertes Lernen

Die vermutlich trivialste Art zu lernen, ist das einfache Auswendiglernen. Wenn Sie auf diese Weise einen Spamfilter erstellten, würde dieser einfach alle E-Mails aussortieren, die mit bereits von Nutzern markierten E-Mails identisch sind – nicht die schlechteste Lösung, aber sicher nicht die beste.

Anstatt einfach mit bekannten Spam-E-Mails identische Nachrichten zu markieren, könnte Ihr Spamfilter auch so programmiert sein, dass darüber hinaus bekannten Spam-E-Mails sehr ähnliche Nachrichten markiert werden. Dazu ist ein *Ähnlichkeitsmaß* zwischen zwei E-Mails nötig. Ein (sehr einfaches) Maß für die Ähnlichkeit zweier E-Mails könnte die Anzahl gemeinsamer Wörter sein. Das System könnte eine E-Mail als Spam markieren, wenn diese viele gemeinsame Wörter mit einer bekannten Spamnachricht aufweist.

Dies nennt man *instanzbasiertes Lernen*: Das System lernt die Beispiele auswendig und verallgemeinert dann mithilfe eines Ähnlichkeitsmaßes auf neue Fälle, wobei es sie mit den gelernten Beispielen (oder einer Untermenge davon) vergleicht. So

würden beispielsweise in Abbildung 1-16 die neuen Instanzen als Dreieck klassifiziert werden, weil die Mehrheit der ähnlichsten Instanzen zu dieser Klasse gehört.

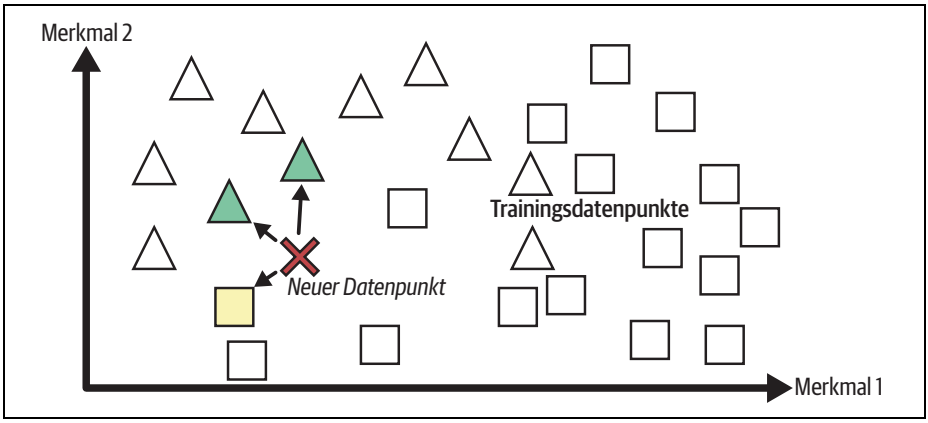

Abbildung 1-16: Instanzbasiertes Lernen

Modellbasiertes Lernen und ein typischer ML-Workflow

Eine weitere Möglichkeit, von einem Beispieldatensatz zu verallgemeinern, ist, ein Modell aus den Beispielen zu entwickeln und dieses Modell dann für *Vorhersagen* zu verwenden. Das wird als *modellbasiertes Lernen* bezeichnet (siehe Abbildung 1-17).

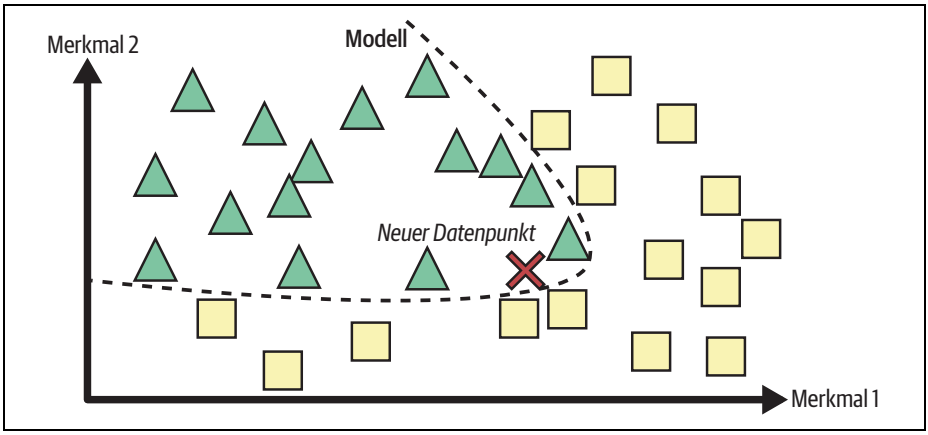

Abbildung 1-17: Modellbasiertes Lernen

Nehmen wir an, Sie möchten herausfinden, ob Geld glücklich macht. Sie laden dazu die Daten des *Better Life Index* von der Webseite des OECD (*https://www.oecdbetterlifeindex.org*) herunter und Statistiken zum Pro-Kopf-Bruttoinlandsprodukt (BIP) von der Webseite der Weltbank (*https://ourworldindata.org*). Anschließend führen Sie beide Tabellen zusammen und sortieren nach dem BIP pro Kopf. Tabelle 1-1 zeigt einen Ausschnitt aus dem Ergebnis.

Tabelle 1-1: Macht Geld Menschen glücklicher?

Land	BIP pro Kopf (USD)	Zufriedenheit
Türkei	28384	5,5
Ungarn	31008	5,6
Frankreich	42026	6,5
Vereinigte Staaten	60236	6,9
Neuseeland	42404	7,3
Australien	48698	7,3
Dänemark	55938	7,6

Stellen wir die Daten für einige dieser Länder dar (siehe Abbildung 1-18).

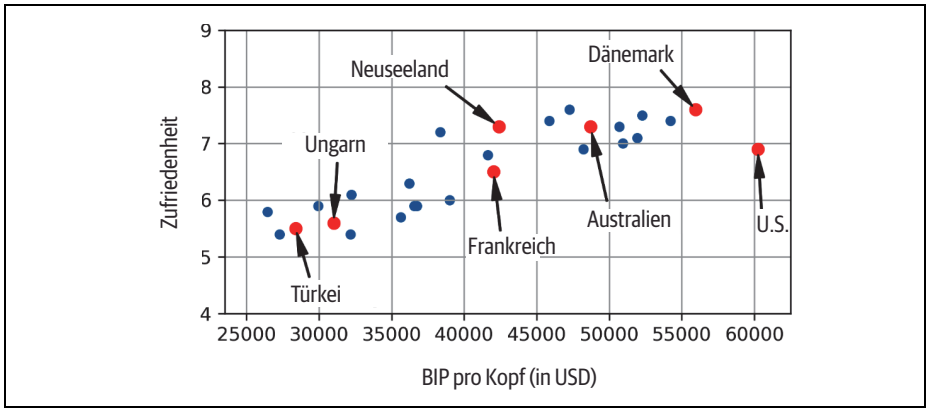

Abbildung 1-18: Sehen Sie hier eine Tendenz?

Es scheint so etwas wie einen Trend zu geben! Auch wenn die Daten *verrauscht* sind (also teilweise zufällig), sieht es so aus, als stiege die Zufriedenheit mehr oder weniger mit dem Pro-Kopf-BIP des Lands linear an. Sie beschließen also, die Zufriedenheit als lineare Funktion des Pro-Kopf-Bruttoinlandsprodukts zu modellieren. Diesen Schritt bezeichnet man als *Modellauswahl*: Sie wählen ein *lineares Modell* der Zufriedenheit mit genau einem Merkmal aus, nämlich dem Pro-Kopf-BIP (siehe Formel 1-1).

Formel 1-1: Ein einfaches lineares Modell

$$\text{Zufriedenheit} = \theta_0 + \theta_1 \times \text{BIP\_pro\_Kopf}$$

Diesen Modell enthält zwei *Modellparameter*, θ_0 und θ_1.[4] Indem Sie diese Parameter verändern, kann das Modell jede lineare Funktion annehmen, wie Sie in Abbildung 1-19 sehen können.

4 Der griechische Buchstabe θ (Theta) wird per Konvention häufig zum Darstellen von Modellparametern verwendet.

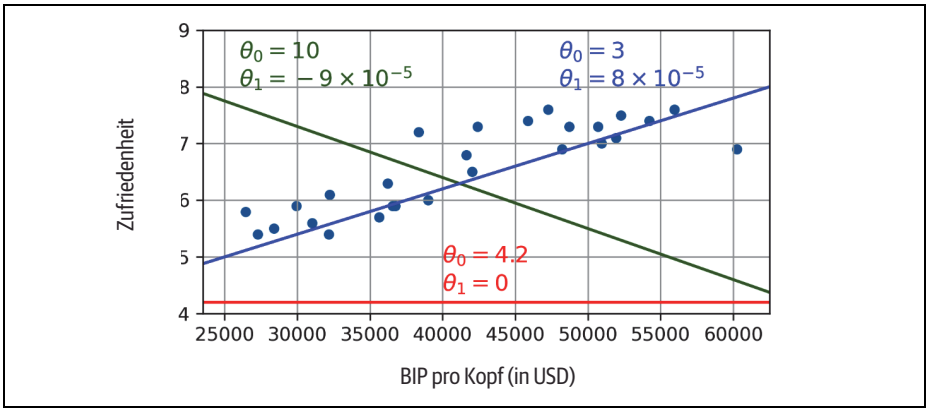

Abbildung 1-19: Einige mögliche lineare Modelle

Bevor Sie Ihr Modell verwenden können, müssen Sie die Werte der Parameter θ_0 und θ_1 festlegen. Woher sollen Sie wissen, welche Werte zur bestmöglichen Qualität führen? Um diese Frage zu beantworten, müssen Sie ein Maß für die Qualität festlegen. Sie können dafür eine *Nutzenfunktion* (oder *Fitnessfunktion*) verwenden, die die *Güte* Ihres Modells bestimmt, oder alternativ eine *Kostenfunktion* definieren, die misst, wie *schlecht* das Modell ist. Bei linearen Modellen verwendet man typischerweise eine Kostenfunktion, die die Entfernung zwischen den Vorhersagen des linearen Modells und den Trainingsbeispielen bestimmt; das Ziel ist, diese Entfernung zu minimieren.

An dieser Stelle kommt der Algorithmus zur linearen Regression ins Spiel: Sie speisen Ihre Trainingsdaten ein, und der Algorithmus ermittelt die für Ihre Daten bestmöglichen Parameter des linearen Modells. Dies bezeichnet man als *Trainieren* des Modells. In unserem Fall ermittelt der Algorithmus $\theta_0 = 3{,}75$ und $\theta_1 = 6{,}78 \times 10^{-5}$ als optimale Werte für die beiden Parameter.

Verwirrenderweise kann das Wort »Modell« auf eine Art von Modell (zum Beispiel lineare Regression), auf eine vollständig spezifizierte Modellarchitektur (zum Beispiel lineare Regression mit einer Eingabe und einer Ausgabe) oder auf das abschließende trainierte Modell, das für Vorhersagen genutzt werden kann (zum Beispiel lineare Regression mit einer Eingabe und einer Ausgabe und $\theta_0 = 3{,}75$ und $\theta_1 = 6{,}78 \times 10^{-5}$), verwendet werden. Die Modellauswahl besteht darin, die Art des Modells auszuwählen und seine Architektur vollständig zu definieren. Beim Trainieren eines Modells geht es darum, einen Algorithmus laufen zu lassen, der die Modellparameter findet, mit denen es am besten zu den Trainingsdaten passt (und hoffentlich gute Vorhersagen für neue Daten trifft).

Nun passt das Modell (für ein lineares Modell) bestmöglich zu den Trainingsdaten, wie Abbildung 1-20 zeigt.

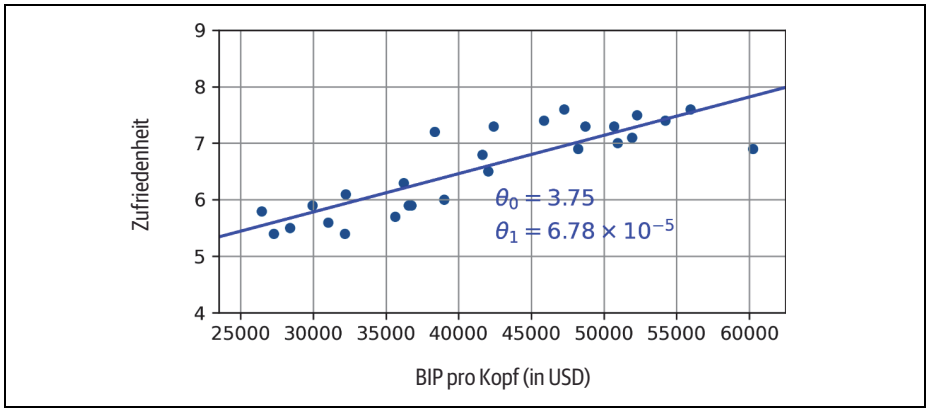

Abbildung 1-20: Das an die Trainingsdaten optimal angepasste lineare Modell

Sie sind nun endlich so weit, das Modell für Vorhersagen einzusetzen. Nehmen wir an, Sie möchten wissen, wie glücklich Zyprioten sind. Die Daten des OECD liefern darauf keine Antwort. Glücklicherweise können Sie unser Modell verwenden, um eine gute Vorhersage zu treffen: Sie schlagen das Pro-Kopf-BIP für Zypern nach, finden 37655 USD und wenden Ihr Modell an. Dabei finden Sie heraus, dass die Zufriedenheit irgendwo um $3{,}75 + 37655 \times 6{,}78 \times 10^{-5} = 6{,}30$ liegt.

Um Ihren Appetit auf die folgenden Kapitel anzuregen, zeigt Beispiel 1-1 den Python-Code, der die Daten lädt, die Eingabedaten `X` von den Labels `y` trennt, einen Scatterplot zeichnet, ein lineares Modell trainiert und eine Vorhersage trifft.[5]

Beispiel 1-1: Trainieren und Ausführen eines linearen Modells mit Scikit-Learn

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression

# Daten laden und vorbereiten
data_root = "https://github.com/ageron/data/raw/main/"
lifesat = pd.read_csv(data_root + "lifesat/lifesat.csv")
X = lifesat[["GDP per capita (USD)"]].values
y = lifesat[["Life satisfaction"]].values

# Daten visualisieren
lifesat.plot(kind='scatter', grid=True,
             x="GDP per capita (USD)", y="Life satisfaction")
plt.axis([23_500, 62_500, 4, 9])
plt.show()
# Ein lineares Modell auswählen
model = LinearRegression()
```

5 Es ist in Ordnung, wenn Sie nicht gleich den gesamten Code nachvollziehen können; wir werden Scikit-Learn in den folgenden Kapiteln vorstellen.

```
# Das Modell trainieren
model.fit(X, y)

# Eine Vorhersage für Zypern treffen
X_new = [[37_655.2]]  # Pro-Kopf-BIP für Zypern 2020
print(model.predict(X_new)) # Ausgabe: [[6.30165767]]
```

Hätten Sie einen instanzbasierten Lernalgorithmus verwendet, würden Sie herausbekommen, dass Israel das zu Zypern ähnlichste Pro-Kopf-BIP hat (38341 USD). Da uns die OECD-Daten die Zufriedenheit mit 7,2 angeben, hätten Sie für Zypern eine Zufriedenheit von 7,2 vorhergesagt. Wenn Sie ein wenig herauszoomen und sich die nächstgelegenen Länder ansehen, finden Sie Litauen und Slowenien mit Zufriedenheitswerten von jeweils 5,9. Der Mittelwert dieser drei Werte ist 6,33, was sehr nah an Ihrer modellbasierten Vorhersage liegt. Dieses einfache Verfahren nennt man *k-nächste-Nachbarn*-Regression (in diesem Beispiel mit $k = 3$).

Das Ersetzen des linearen Regressionsmodells durch *k*-nächste-Nachbarn-Regression im obigen Code erfordert lediglich das Ersetzen dieser Zeilen:

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
```

durch diese zwei:

```
from sklearn.neighbors import KNeighborsRegressor
model = KNeighborsRegressor(n_neighbors=3)
```

Wenn alles gut gegangen ist, wird Ihr Modell gute Vorhersagen treffen. Wenn nicht, müssen Sie weitere Merkmale heranziehen (Beschäftigungsquote, Gesundheit, Luftverschmutzung und Ähnliches), sich mehr oder hochwertigere Trainingsdaten beschaffen oder ein mächtigeres Modell auswählen (z. B. ein polynomielles Regressionsmodell).

Zusammengefasst:

- Sie haben die Daten untersucht.
- Sie haben ein Modell ausgewählt.
- Sie haben es auf Trainingsdaten trainiert (d. h., der Trainingsalgorithmus hat nach den Modellparametern gesucht, die eine Kostenfunktion minimieren).
- Schließlich haben Sie das Modell verwendet, um für neue Fälle Vorhersagen zu treffen (dies nennt man *Inferenz*), und hoffen, dass das Modell gut verallgemeinert.

So sieht ein typisches Machine-Learning-Projekt aus. In Kapitel 2 können Sie dies selbst erfahren, indem Sie ein Projekt vom Anfang bis zum Ende durcharbeiten. Wir haben bisher ein weites Feld beschritten: Sie wissen bereits, worum es beim Machine Learning wirklich geht, warum es nützlich ist, welche Arten von ML-Sys-

temen verbreitet sind und wie ein typischer Arbeitsablauf aussieht. Nun werden wir uns anschauen, was beim Lernen schiefgehen kann und präzise Vorhersagen verhindert.

Die wichtigsten Herausforderungen beim Machine Learning

Kurz gesagt, da Ihre Hauptaufgabe darin besteht, ein Modell auszuwählen und mit Daten zu trainieren, sind die zwei möglichen Fehlerquellen dabei ein »schlechtes Modell« sowie »schlechte Daten«. Beginnen wir mit Beispielen für schlechte Daten.

Unzureichende Menge an Trainingsdaten

Damit ein Kleinkind lernt, was ein Apfel ist, genügt es, dass Sie auf einen Apfel zeigen und »Apfel« sagen (und die Prozedur einige Male wiederholen). Damit ist das Kind in der Lage, Äpfel in allen möglichen Farben und Formen zu erkennen. Genial.

Machine Learning ist noch nicht ganz so weit; bei den meisten maschinellen Lernverfahren ist eine Vielzahl an Daten erforderlich, damit sie funktionieren. Selbst bei sehr einfachen Aufgaben benötigen Sie üblicherweise Tausende von Beispielen, und bei komplexen Aufgaben wie Bild- oder Spracherkennung können es auch Millionen sein (es sei denn, Sie können Teile eines existierenden Modells wiederverwenden).

Die unverschämte Effektivität von Daten

In einem berühmten Artikel (*https://homl.info/6*) aus dem Jahr 2001 zeigten die Forscher Michele Banko und Eric Brill bei Microsoft, dass sehr unterschiedliche maschinelle Lernalgorithmen, darunter sehr primitive, bei einem sehr komplexen Problem wie der Unterscheidung von Sprache etwa gleich gut abschnitten,[6] wenn man ihnen nur genug Daten zur Verfügung stellt (wie Sie in Abbildung 1-21 sehen können).

Die Autoren drücken dies folgendermaßen aus: »Diese Ergebnisse legen nahe, dass wir unsere Entscheidung über das Investieren von Zeit und Geld in die Entwicklung von Algorithmen gegenüber der Entwicklung eines Datenkorpus neu bewerten sollten.«

6 Beispielsweise aus dem Kontext zu ermitteln, ob man »to«, »two« oder »too« schreiben muss.

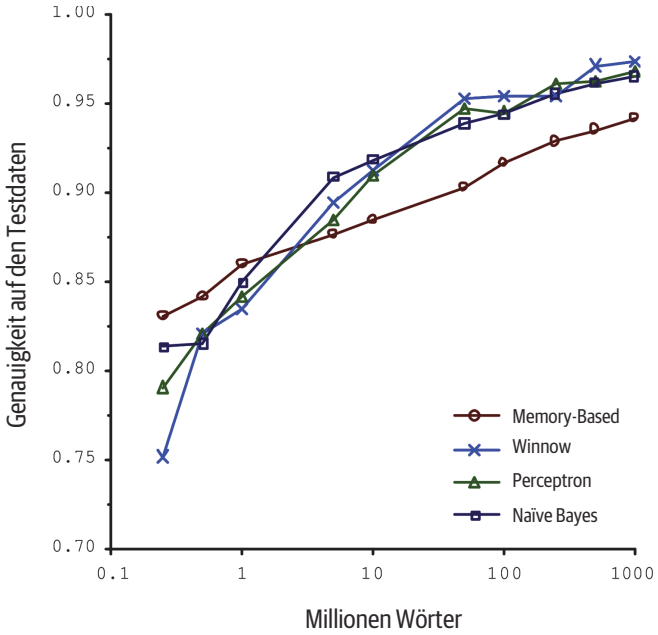

Abbildung 1-21: Die Wichtigkeit der Daten im Vergleich zum Algorithmus[7]

Dass Daten bei komplexen Problemen wichtiger als Algorithmen sind, wurde von Peter Norvig et al. in einem Artikel mit dem Titel »The Unreasonable Effectiveness of Data« (*https://homl.info/7*), veröffentlicht im Jahr 2009, weiter thematisiert.[8] Es sollte jedoch betont werden, dass kleine und mittelgroße Datensätze nach wie vor sehr häufig sind und dass es nicht immer einfach oder billig ist, an zusätzliche Trainingsdaten heranzukommen. Daher schreiben Sie die Algorithmik besser nicht gleich ab.

Nicht repräsentative Trainingsdaten

Um gut zu verallgemeinern, ist es entscheidend, dass Ihre Trainingsdaten die zu verallgemeinernden neuen Situationen repräsentieren. Dies ist sowohl beim instanzbasierten als auch beim modellbasierten Lernen der Fall.

Beispielsweise waren die zuvor zum Trainieren eines linearen Modells eingesetzten Länder nicht perfekt repräsentativ; Länder mit einem BIP von weniger als 23500

7 Die Abbildung wurde mit Erlaubnis von Michele Banko und Eric Brill reproduziert aus »Scaling to Very Very Large Corpora for Natural Language Disambiguation«, *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics (2001)*, 26–33.

8 Peter Norvig et al., »The Unreasonable Effectiveness of Data«, *IEEE Intelligent Systems* 24, no. 2 (2009): 8–12.

USD oder mehr als 62500 USD pro Einwohner fehlten. Abbildung 1-22 zeigt, wie die Daten mit den fehlenden Ländern aussehen.

Wenn Sie mit diesen Daten ein lineares Modell trainieren, erhalten Sie die durchgezogene Linie. Das alte Modell ist durch die gepunktete Linie gekennzeichnet. Wie Sie sehen, verändert sich nicht nur das Modell durch die Daten. Es wird auch deutlich, dass ein einfaches lineares Modell vermutlich nie gut funktionieren wird. Es sieht ganz danach aus, dass reiche Länder nicht glücklicher als Länder mit mittlerem Wohlstand sind (sie wirken sogar etwas weniger glücklich). Auch einige arme Länder scheinen glücklicher als viele reiche Länder zu sein.

Das mit einem nicht repräsentativen Datensatz trainierte Modell trifft also ungenaue Vorhersagen, besonders bei sehr armen und sehr reichen Ländern.

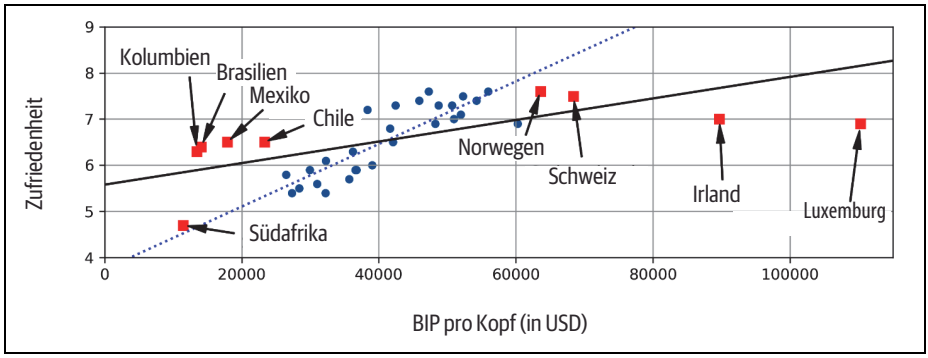

Abbildung 1-22: Ein repräsentativerer Trainingsdatensatz

Es ist daher wichtig, einen Trainingsdatensatz zu verwenden, in dem die zu verallgemeinernden Fälle abgebildet sind. Oft ist dies schwieriger, als es sich anhört: Wenn die Stichprobe zu klein ist, erhalten Sie *Stichprobenrauschen* (also durch Zufall nicht repräsentative Daten). Selbst sehr große Stichproben können nicht repräsentativ sein, wenn die Methode zur Erhebung fehlerhaft ist. Dies nennt man auch *Stichprobenverzerrung*.

Beispiele für Stichprobenverzerrungen

Das vermutlich berühmteste Beispiel für Stichprobenverzerrung stammt aus der US-Präsidentschaftswahl von 1936, bei der Landon gegen Roosevelt antrat: Der *Literary Digest* führte damals eine sehr große Umfrage durch, bei der Briefe an etwa 10 Millionen Menschen verschickt wurden. Nach dem Sammeln von 2,4 Millionen Antworten wurde daraus mit hoher Konfidenz vorhergesagt, dass Landon 57 % der Stimmen erhalten würde. Tatsächlich gewann aber Roosevelt mit 62 % der Stimmen. Der Fehler lag in der Methode, die *Literary Digest* beim Erheben der Stichprobe einsetzte:

- Zum einen verwendete *Literary Digest* Telefonbücher, Abonnentenlisten, Mitgliederlisten von Klubs und so weiter, um an die Adressen zum Verschicken der Umfrage zu kommen. In allen diesen Listen waren wohlhabendere Menschen stärker vertreten, die wahrscheinlich eher für die Republikaner (und damit Landon) stimmen würden.
- Zum anderen antworteten weniger als 25% der Menschen auf die Umfrage. Auch dies führte zu einer Stichprobenverzerrung, da Menschen, die sich nicht für Politik interessierten, Menschen, die den *Literary Digest* nicht mochten, und andere wichtige Gruppen potenziell aussortiert wurden. Diese Art von Stichprobenverzerrung nennt man auch *Schweigeverzerrung*.

Ein weiteres Beispiel: Sagen wir, Sie möchten ein System zum Erkennen von Funk-Musikvideos konstruieren. Eine Möglichkeit zum Zusammenstellen der Trainingsdaten wäre, »funk music« bei YouTube einzugeben und die erhaltenen Videos zu verwenden. Allerdings nehmen Sie dabei an, dass Ihnen die Suchmaschine von YouTube Videos liefert, die repräsentativ für alle Funk-Musikvideos auf YouTube sind. In der Realität werden einige beliebte Künstler in den Suchergebnissen jedoch überrepräsentiert sein (und wenn Sie in Brasilien leben, erhalten Sie eine Menge Videos zu »funk carioca«, die sich überhaupt nicht wie James Brown anhören). Wie aber sonst sollte man einen großen Datensatz sammeln?

Minderwertige Daten

Wenn Ihre Trainingsdaten voller Fehler, Ausreißer und Rauschen sind (z.B. wegen schlechter Messungen), ist es für das System schwieriger, die zugrunde liegenden Muster zu erkennen. Damit ist es weniger wahrscheinlich, dass Ihr System eine hohe Qualität erzielt. Meistens lohnt es sich, Zeit in das Säubern der Trainingsdaten zu investieren. Tatsächlich verbringen die meisten Data Scientists einen Großteil ihrer Zeit mit nichts anderem, beispielsweise:

- Wenn einige Datenpunkte deutliche Ausreißer sind, hilft es, diese einfach zu entfernen oder die Fehler manuell zu beheben.
- Wenn manche Merkmale lückenhaft sind (z.B. 5% Ihrer Kunden ihr Alter nicht angegeben haben), müssen Sie sich entscheiden, ob Sie dieses Merkmal insgesamt ignorieren wollen oder ob Sie die entsprechenden Datenpunkte entfernen, die fehlenden Werte ergänzen (z.B. mit dem Median) oder ein Modell mit diesem Merkmal und eines ohne dieses Merkmal trainieren möchten.

Irrelevante Merkmale

Eine Redewendung besagt: Müll rein, Müll raus. Ihr System wird nur etwas erlernen können, wenn Ihre Trainingsdaten genug relevante Merkmale und nicht zu

viele irrelevante enthalten. Ein für den Erfolg eines Machine-Learning-Projekts maßgeblicher Schritt ist, die Merkmale zum Trainieren gut auszuwählen. Zu diesem *Entwicklung von Merkmalen* genannten Vorgang gehören:

- *Auswahl von Merkmalen* (aus den vorhandenen die nützlichsten Merkmale für das Trainieren auswählen).
- *Extraktion von Merkmalen* (vorhandene Merkmale miteinander kombinieren, sodass ein nützlicheres entsteht – wie wir oben gesehen haben, helfen dabei Algorithmen zur Dimensionsreduktion).
- Erstellen neuer Merkmale durch das Erheben neuer Daten.

Nun, da wir viele Beispiele für schlechte Daten kennengelernt haben, schauen wir uns auch noch einige schlechte Algorithmen an.

Overfitting der Trainingsdaten

Angenommen, Sie sind im Ausland unterwegs und der Taxifahrer zockt Sie ab. Sie mögen versucht sein, zu sagen, dass *alle* Taxifahrer in diesem Land Betrüger seien. Menschen neigen häufig zu übermäßiger Verallgemeinerung, und Maschinen können leider in die gleiche Falle tappen, wenn wir nicht vorsichtig sind. Beim Machine Learning nennt man dies *Overfitting*: Dabei funktioniert das Modell auf den Trainingsdaten, kann aber nicht gut verallgemeinern.

Abbildung 1-23 zeigt ein Beispiel für ein polynomielles Modell höheren Grades für die Zufriedenheit, das die Trainingsdaten stark overfittet. Es erzielt zwar auf den Trainingsdaten eine höhere Genauigkeit als das einfache lineare Modell, aber würden Sie den Vorhersagen dieses Modells wirklich trauen?

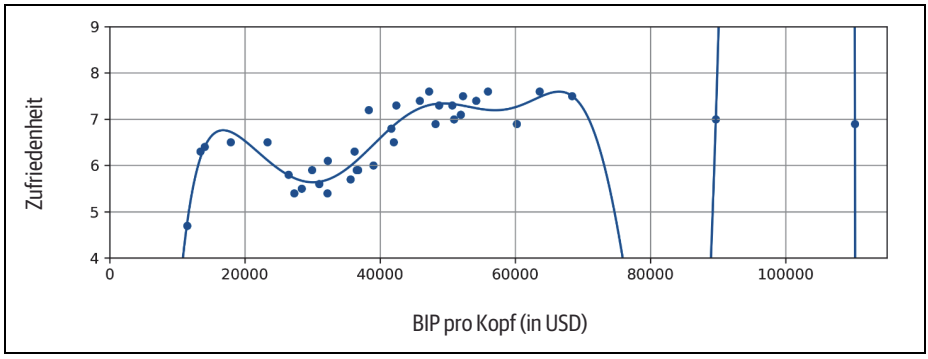

Abbildung 1-23: Overfitting der Trainingsdaten

Komplexe Modelle wie Deep-Learning-Netze können subtile Muster in den Daten erkennen. Wenn aber der Trainingsdatensatz verrauscht oder zu klein ist, wodurch die Stichprobe verrauscht ist, entdeckt das Modell Muster im Rauschen selbst (wie beim Taxifahrer-Beispiel). Diese Muster lassen sich natürlich nicht auf neue Daten übertragen. Nehmen wir beispielsweise an, Sie stellten Ihrem Modell für die Zufrie-

denheit viele weitere Merkmale zur Verfügung, darunter wenig informative wie den Namen des Lands. Ein komplexes Modell könnte dann herausfinden, dass alle Länder mit einer Zufriedenheit über 7 ein *w* im (englischen) Namen haben: New Zeeland (7,3), Norway (7,6), Sweden (7,3) und Switzerland (7,5). Wie sicher können Sie sein, dass sich diese *w*-Regel auf Rwanda oder Zimbabwe anwenden lässt? Natürlich trat dieses Muster in den Trainingsdaten rein zufällig auf, aber das Modell ist nicht in der Lage, zu entscheiden, ob ein Muster echt ist oder durch das Rauschen in den Daten entsteht.

Overfitting tritt auf, wenn das Modell angesichts der Menge an Trainingsdaten und der Menge an Rauschen zu komplex ist. Mögliche Lösungen sind:

- das Modell zu vereinfachen, indem man es durch eines mit weniger Parametern ersetzt (z. B. ein lineares Modell statt eines polynomiellen Modells höheren Grades), die Anzahl der Merkmale im Trainingsdatensatz verringert oder dem Modell Restriktionen auferlegt,
- mehr Trainingsdaten zu sammeln
- oder das Rauschen in den Trainingsdaten zu reduzieren (z. B. Datenfehler zu beheben und Ausreißer zu entfernen).

Einem Modell Restriktionen aufzuerlegen, um es zu vereinfachen und das Risiko für Overfitting zu reduzieren, wird als *Regularisierung* bezeichnet. Beispielsweise hat das oben definierte lineare Modell zwei Parameter, θ_0 und θ_1. Damit hat der Lernalgorithmus zwei *Freiheitsgrade*, mit denen das Modell an die Trainingsdaten angepasst werden kann: Sowohl die Höhe (θ_0) als auch die Steigung (θ_1) der Geraden lassen sich verändern. Wenn wir $\theta_1 = 0$ erzwingen würden, hätte der Algorithmus nur noch einen Freiheitsgrad, und es würde viel schwieriger, die Daten gut zu fitten: Die Gerade könnte sich nur noch nach oben oder unten bewegen, um so nah wie möglich an den Trainingsdatenpunkten zu landen. Sie würde daher in der Nähe des Mittelwerts landen. Wirklich ein sehr einfaches Modell! Wenn wir dem Modell erlauben, θ_1 zu verändern, aber einen kleinen Wert erzwingen, hat der Lernalgorithmus zwischen einem und zwei Freiheitsgrade. Das entstehende Modell ist einfacher als das mit zwei Freiheitsgraden, aber komplexer als das mit nur einem. Ihre Aufgabe ist es, die richtige Balance zwischen dem perfekten Fitten der Daten und einem möglichst einfachen Modell zu finden, sodass es gut verallgemeinert.

Abbildung 1-24 zeigt drei Modelle: Die gepunktete Linie steht für das ursprüngliche mit den als Kreis dargestellten Ländern trainierte Modell (ohne die als Quadrat dargestellten Länder), die durchgezogene Linie für unser zweites mit allen Ländern (Kreise und Quadrate) trainiertes Modell, und die gestrichelte Linie ist ein Modell, das mit den gleichen Daten wie das erste Modell trainiert wurde, aber mit zusätzlicher Regularisierung. Sie sehen, dass die Regularisierung eine geringere Steigung erzwungen hat. Dieses Modell passt nicht so gut zu den Trainingsdaten wie das erste

Modell, erlaubt aber eine bessere Verallgemeinerung auf neue Beispiele, die es während des Trainings nicht kannte (Quadrate).

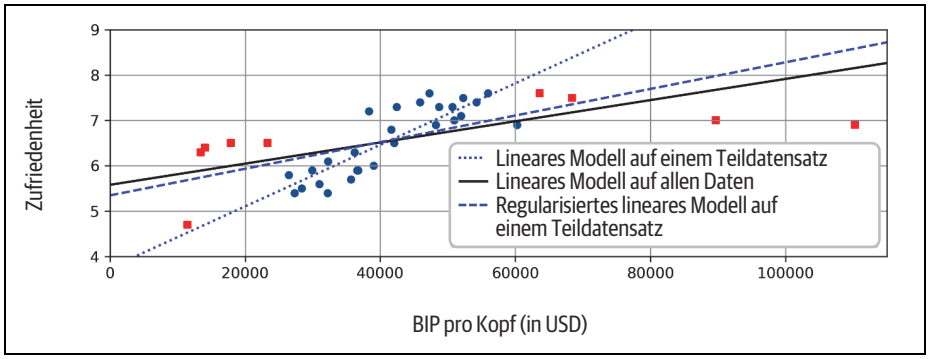

Abbildung 1-24: Regularisierung reduziert das Risiko für Overfitting.

Die Stärke der Regularisierung beim Lernen lässt sich über einen *Hyperparameter* kontrollieren. Ein Hyperparameter ist ein Parameter des Lernalgorithmus (nicht des Modells). Als solcher unterliegt er nicht dem Lernprozess selbst; er muss vor dem Training gesetzt werden und bleibt über den gesamten Trainingszeitraum konstant. Wenn Sie den Hyperparameter zur Regularisierung auf einen sehr großen Wert setzen, erhalten Sie ein beinahe flaches Modell (eine Steigung nahe null); Sie können sich sicher sein, dass der Lernalgorithmus die Trainingsdaten nicht overfittet, eine gute Lösung wird aber ebenfalls unwahrscheinlicher. Die Feineinstellung der Hyperparameter ist ein wichtiger Teil bei der Entwicklung eines Machine-Learning-Systems (im nächsten Kapitel lernen Sie ein detailliertes Beispiel kennen).

Underfitting der Trainingsdaten

Wie Sie sich denken können, ist *Underfitting* das genaue Gegenteil von Overfitting: Es tritt auf, wenn Ihr Modell zu einfach ist, um die in den Daten enthaltene Struktur zu erlernen. Beispielsweise ist ein lineares Modell der Zufriedenheit anfällig für Underfitting; die Realität ist einfach komplexer als unser Modell, sodass Vorhersagen selbst auf den Trainingsdaten zwangsläufig ungenau werden.

Die wichtigsten Möglichkeiten, dieses Problem zu beheben, sind:

- ein mächtigeres Modell mit mehr Parametern zu verwenden,
- dem Lernalgorithmus bessere Merkmale zur Verfügung zu stellen (Entwicklung von Merkmalen) oder
- die Restriktionen des Modells zu verringern (z. B. die Hyperparameter zur Regularisierung zu verringern).

Zusammenfassung

Inzwischen wissen Sie schon eine Menge über Machine Learning. Wir haben aber so viele Begriffe behandelt, dass Sie sich vielleicht ein wenig verloren vorkommen. Betrachten wir deshalb noch einmal das Gesamtbild:

- Beim Machine Learning geht es darum, Maschinen bei der Lösung einer Aufgabe zu verbessern, indem sie aus Daten lernen, anstatt explizit definierte Regeln zu erhalten.
- Es gibt viele unterschiedliche Arten von ML-Systemen: überwachte und unüberwachte, Batch- und Online-Learning, instanzbasierte und modellbasierte Systeme.
- In einem ML-Projekt sammeln Sie Daten in einem Trainingsdatensatz und speisen diesen in einen Lernalgorithmus ein. Wenn der Algorithmus auf einem Modell basiert, tunt er einige Parameter, um das Modell an die Trainingsdaten anzupassen (d.h., um gute Vorhersagen auf den Trainingsdaten selbst zu treffen). Danach ist es hoffentlich in der Lage, auch für neue Daten gute Vorhersagen zu treffen. Wenn der Algorithmus instanzbasiert ist, lernt er die Beispiele einfach auswendig und verallgemeinert auf neue Instanzen durch ein Ähnlichkeitsmaß, um sie mit den bekannten Instanzen zu vergleichen.
- Wenn der Trainingsdatensatz zu klein ist oder die Daten nicht repräsentativ, verrauscht oder durch irrelevante Merkmale verunreinigt sind, wird das System keine hohe Qualität erbringen (Müll rein, Müll raus). Schließlich darf Ihr Modell weder zu einfach (dann underfittet es) noch zu komplex sein (dann overfittet es).

Es gilt noch ein letztes wichtiges Thema zu behandeln: Sobald Sie ein Modell trainiert haben, sollten Sie nicht nur »hoffen«, dass es gut verallgemeinert, Sie sollten es auch evaluieren und, falls nötig, Feinabstimmungen vornehmen. Sehen wir einmal, wie das geht.

Testen und Validieren

Ein Modell mit neuen Datenpunkten auszuprobieren, ist tatsächlich die einzige Möglichkeit, zu erfahren, ob es gut auf neue Daten verallgemeinert. Sie können das Modell dazu in einem Produktivsystem einsetzen und beobachten, wie es funktioniert. Wenn sich Ihr Modell aber als definitiv untauglich herausstellt, werden sich Ihre Nutzer beschweren – nicht die beste Idee.

Eine bessere Alternative ist, Ihre Daten in zwei Datensätze zu unterteilen: den *Trainingsdatensatz* und den *Testdatensatz*. Wie die Namen vermuten lassen, trainieren Sie Ihr Modell mit dem Trainingsdatensatz und testen es mit dem Testdatensatz. Die Abweichung bei neuen Datenpunkten bezeichnet man als *Verallgemeinerungsfehler* (engl. *Out-of-Sample Error*). Indem Sie Ihr Modell auf dem Testdatensatz

evaluieren, erhalten Sie eine Schätzung dieser Abweichung. Der Wert verrät Ihnen, wie gut Ihr Modell auf zuvor nicht bekannten Datenpunkten abschneidet.

Wenn der Fehler beim Training gering ist (Ihr Modell also auf dem Trainingsdatensatz wenige Fehler begeht), der Verallgemeinerungsfehler aber groß, overfittet Ihr Modell die Trainingsdaten.

Es ist üblich, 80 % der Daten zum Trainieren zu nehmen und 20 % zum Testen *zurückzuhalten*. Das hängt aber auch von der Größe des Datensatzes ab: Enthält er 10 Millionen Instanzen, bedeutet ein Zurückhalten von 1 %, dass Ihr Testdatensatz aus 100.000 Instanzen besteht – vermutlich mehr als genug für eine gute Abschätzung des Verallgemeinerungsfehlers.

Hyperparameter anpassen und Modellauswahl

Das Evaluieren eines Modells ist einfach: Verwenden Sie einen Testdatensatz. Aber möglicherweise schwanken Sie zwischen zwei Typen von Modellen (z. B. einem linearen und einem polynomiellen Modell): Wie sollen Sie sich zwischen ihnen entscheiden? Sie können natürlich beide trainieren und mit dem Testdatensatz vergleichen, wie gut beide verallgemeinern.

Nehmen wir an, das lineare Modell verallgemeinert besser, aber Sie möchten durch Regularisierung dem Overfitting entgegenwirken. Die Frage ist: Wie wählen Sie den Wert des Regularisierungshyperparameters aus? Natürlich können Sie 100 unterschiedliche Modelle mit 100 unterschiedlichen Werten für diesen Hyperparameter trainieren. Angenommen, Sie fänden einen optimalen Wert für den Hyperparameter, der den niedrigsten Verallgemeinerungsfehler liefert, z. B. 5 %. Sie bringen Ihr Modell daher in die Produktivumgebung, aber leider schneidet es nicht wie erwartet ab und produziert einen Fehler von 15 %. Was ist passiert?

Das Problem ist, dass Sie den Verallgemeinerungsfehler mithilfe des Testdatensatzes mehrfach bestimmt und das Modell und seine Hyperparameter so angepasst haben, dass es *für diesen Datensatz* das beste Modell hervorbringt. Damit erbringt das Modell bei neuen Daten wahrscheinlich keine gute Qualität.

Eine übliche Lösung dieses Problems nennt sich *Hold-out-Validierung*: Sie halten einfach einen Teil des Trainingsdatensatzes zurück, um die verschiedenen Modellkandidaten zu bewerten, und wählen den besten aus. Dieser neue zurückgehaltene Datensatz wird als *Validierungsdatensatz* (oder manchmal auch als *Entwicklungsdatensatz* oder *Dev Set*) bezeichnet. Genauer gesagt, trainieren Sie mehrere Modelle mit unterschiedlichen Hyperparametern auf dem Trainingsdatensatz und wählen das auf dem Validierungsdatensatz am besten abschneidende Modell und die Hyperparameter aus. Nach diesem Hold-out-Validierungsprozess trainieren Sie das beste Modell mit dem vollständigen Trainingsdatensatz (einschließlich des Validierungsdatensatzes) und erhalten so das endgültige Modell. Schließlich führen Sie ei-

nen einzelnen abschließenden Test an diesem finalen Modell mit dem Testdatensatz durch, um den Verallgemeinerungsfehler abzuschätzen.

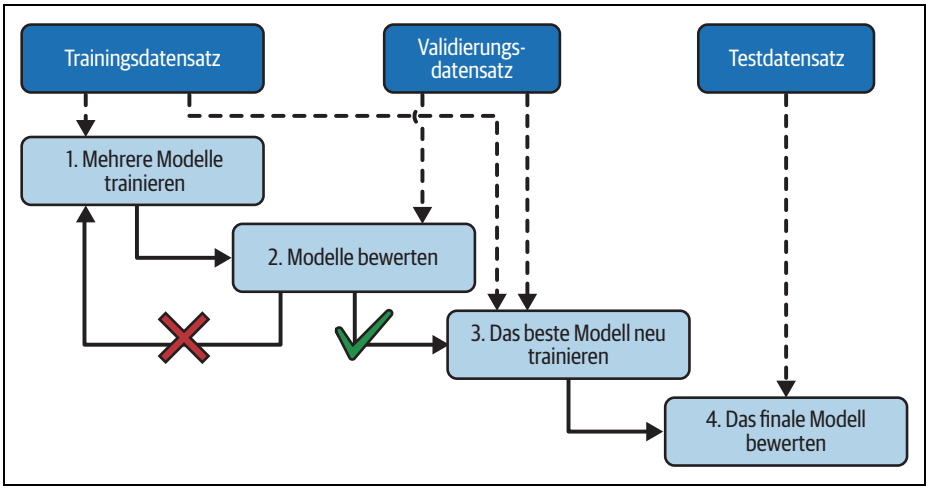

Abbildung 1-25: Modellauswahl durch Hold-out-Validierung.

Dieses Vorgehen funktioniert meist ziemlich gut. Ist aber der Validierungsdatensatz zu klein, wird die Modellbewertung ungenau sein – Sie landen eventuell unabsichtlich bei einem suboptimalen Modell. Ist der Validierungsdatensatz hingegen zu groß, wird der verbleibende Trainingsdatensatz viel kleiner sein als der vollständige Trainingsdatensatz. Warum ist das schlecht? Nun, da das finale Modell mit dem vollständigen Trainingsdatensatz trainiert werden wird, ist es nicht ideal, Modellkandidaten zu vergleichen, die mit einem viel kleineren Trainingsdatensatz trainiert wurden. Das wäre so, als würden Sie den schnellsten Sprinter auswählen, damit er an einem Marathon teilnimmt. Eine Möglichkeit, dieses Problem zu lösen, ist eine wiederholt durchgeführte *Kreuzvalidierung* mit vielen kleinen Validierungsdatensätzen. Jedes Modell wird einmal per Validierungsdatensatz geprüft, nachdem es mit dem Rest der Daten trainiert wurde. Durch ein Mitteln aller Bewertungen eines Modells können Sie dessen Genauigkeit deutlich besser bestimmen. Es gibt aber einen Nachteil: Die Trainingsdauer wird mit der Anzahl der Validierungsdatensätze multipliziert.

Datendiskrepanz

In manchen Fällen kommt man zwar leicht an große Mengen von Trainingsdaten, diese sind aber keine perfekte Repräsentation der im Produktivumfeld verwendeten Daten. Stellen Sie sich zum Beispiel vor, Sie wollten eine Mobil-App bauen, die Bilder von Blumen aufnimmt und automatisch deren Spezies bestimmt. Sie können problemlos Millionen von Blumenbildern aus dem Web laden, aber diese werden nicht perfekt die Bilder repräsentieren, die tatsächlich mit der App auf einem

Smartphone aufgenommen werden. Vielleicht haben Sie nur 10.000 repräsentative Bilder (also solche, die mit der App aufgenommen wurden).

In diesem Fall ist es am wichtigsten, daran zu denken, dass der Validierungsdatensatz und der Testdatensatz in Bezug auf die produktiv genutzten Daten so repräsentativ wie möglich sein müssen, daher sollten sie nur aus repräsentativen Bildern bestehen: Sie können sie mischen und die eine Hälfte in den Validierungsdatensatz sowie die andere in den Testdatensatz stecken (wobei Sie darauf achten müssen, keine Dubletten oder Nahezu-Dubletten in beiden Sets zu haben). Beobachten Sie nach dem Trainieren Ihres Modells mit den Bildern aus dem Web, dass die Genauigkeit des Modells beim Validierungsdatensatz enttäuschend ist, werden Sie nicht wissen, ob das daran liegt, dass das Modell bezüglich des Trainingsdatensatzes overfittet, oder ob es einfach einen Unterschied zwischen den Bildern aus dem Web und denen aus der App gibt.

Eine Lösung ist, einen Teil der Trainingsbilder (aus dem Web) in einen weiteren Datensatz zu stecken, den Andrew Ng als *Train-Dev-Set* bezeichnet. Nachdem das Modell trainiert wurde (mit dem Trainingsdatensatz, *nicht* mit dem Train-Dev-Set), können Sie es auf das Train-Dev-Set loslassen. Funktioniert es schlecht, muss es bezüglich des Trainingsdatensatzes overfittet sein, und Sie sollten versuchen, das Modell zu vereinfachen oder zu regularisieren sowie mehr Trainingsdaten zu erhalten und diese zu säubern. Funktioniert es für den Train-Dev-Datensatz gut, können Sie es auf den Dev-Datensatz loslassen. Funktioniert es dort schlecht, muss das Problem aus der Datendiskrepanz entstanden sein. Sie können versuchen, die Lösung des Problems durch eine Vorverarbeitung der Webbilder anzugehen, sodass sie mehr wie die Bilder aus der App aussehen, und dann das Modell neu zu trainieren. Haben Sie irgendwann ein Modell, das sowohl für den Train-Dev-Datensatz wie auch für den Validierungsdatensatz gut funktioniert, können Sie es ein letztes Mal mit dem Testdatensatz auswerten, um zu prüfen, wie gut es vermutlich im Produktivumfeld agieren wird.

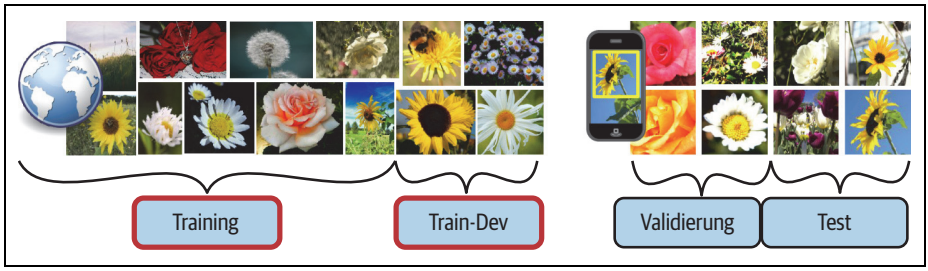

Abbildung 1-26: Gibt es nur wenige echte Daten (rechts), können Sie ähnliche, häufiger vorhandene Daten zum Trainieren nutzen und ein paar davon in einem Train-Dev-Set zurückhalten, um auf Overfitting zu prüfen – die echten Daten werden dann genutzt, um auf Datendiskrepanz zu kontrollieren (Validierungsdatensatz) und die Qualität des finalen Modells zu evaluieren (Testdatensatz).

Das No-Free-Lunch-Theorem

Ein Modell ist eine vereinfachte Repräsentation der Daten. Die Vereinfachung soll überflüssige Details eliminieren, die sich vermutlich nicht auf neue Datenpunkte verallgemeinern lassen. Wählen Sie eine bestimmte Art von Modell aus, treffen Sie implizit *Annahmen* über die Daten. Entscheiden Sie sich beispielsweise für ein lineares Modell, gehen Sie implizit davon aus, dass die Daten grundsätzlich linear sind und die Distanz zwischen den Datenpunkten und der Geraden lediglich Rauschen ist, das man ignorieren kann.

In einem berühmten Artikel aus dem Jahr 1996 (*https://homl.info/8*)[9] zeigte David Wolpert, dass es keinen Grund gibt, ein Modell gegenüber einem anderen zu bevorzugen, wenn Sie absolut keine Annahmen über die Daten treffen. Dies nennt man auch das *No-Free-Lunch*-(NFL-)Theorem. Bei einigen Datensätzen ist das optimale Modell ein lineares Modell, während bei anderen ein neuronales Netz am besten geeignet ist. Es gibt kein Modell, das garantiert *a priori* besser funktioniert (daher der Name des Theorems). Der einzige Weg, wirklich sicherzugehen, ist, alle möglichen Modelle zu evaluieren. Da dies nicht möglich ist, treffen Sie in der Praxis einige wohlüberlegte Annahmen über die Daten und evaluieren nur einige sinnvoll ausgewählte Modelle. Bei einfachen Aufgaben könnten Sie beispielsweise lineare Modelle mit unterschiedlich starker Regularisierung auswerten, bei einer komplexen Aufgabe hingegen verschiedene neuronale Netze.

Übungen

In diesem Kapitel haben wir einige der wichtigsten Begriffe zum Machine Learning behandelt. In den folgenden Kapiteln werden wir uns eingehender damit beschäftigen und mehr Code schreiben, aber zuvor sollten Sie sicherstellen, dass Sie die folgenden Fragen beantworten können:

1. Wie würden Sie Machine Learning definieren?
2. Können Sie vier Arten von Anwendungen nennen, für die Machine Learning gut geeignet ist?
3. Was ist ein gelabelter Trainingsdatensatz?
4. Was sind die zwei verbreitetsten Aufgaben beim überwachten Lernen?
5. Können Sie vier häufig anzutreffende Aufgaben für unüberwachtes Lernen nennen?
6. Was für einen Algorithmus würden Sie verwenden, um einen Roboter über verschiedene unbekannte Oberflächen laufen zu lassen?

9 David Wolpert, »The Lack of A Priori Distinctions Between Learning Algorithms«, *Neural Computation* 8 (1996): 1341–1390.

7. Welche Art Algorithmus würden Sie verwenden, um Ihre Kunden in unterschiedliche Gruppen einzuteilen?
8. Würden Sie die Aufgabe, Spam zu erkennen, als überwachte oder unüberwachte Lernaufgabe einstufen?
9. Was ist ein Onlinelernsystem?
10. Was ist Out-of-Core-Lernen?
11. Welche Art Algorithmus beruht auf einem Ähnlichkeitsmaß, um Vorhersagen zu treffen?
12. Worin besteht der Unterschied zwischen einem Modellparameter und einem Modellhyperparameter?
13. Wonach suchen modellbasierte Algorithmen? Welche Strategie führt am häufigsten zum Erfolg? Wie treffen sie Vorhersagen?
14. Können Sie vier der wichtigsten Herausforderungen beim Machine Learning benennen?
15. Welches Problem liegt vor, wenn Ihr Modell auf den Trainingsdaten eine sehr gute Qualität erbringt, aber schlecht auf neue Daten verallgemeinert? Nennen Sie drei Lösungsansätze.
16. Was ist ein Testdatensatz, und warum sollte man einen verwenden?
17. Was ist der Zweck eines Validierungsdatensatzes?
18. Was ist das Train-Dev-Set, wann brauchen Sie es, und wie verwenden Sie es?
19. Was kann schiefgehen, wenn Sie Hyperparameter mithilfe der Testdaten einstellen?

Die deutschsprachigen Lösungen zu diesen Übungen finden Sie unter *https://dpunkt.de/produkt/praxiseinstieg-machine-learning-mit-scikit-learn-keras-und-tensorflow-2/* auf der Registerkarte *Zusatzmaterial*, die englischsprachigen Lösungen am Ende des Notebooks zu diesem Kapitel unter *https://homl.info/colab3*.

KAPITEL 2

Ein Machine-Learning-Projekt von A bis Z

In diesem Kapitel werden Sie ein Beispielprojekt vom Anfang bis zum Ende durchleben. Wir nehmen dazu an, Sie seien ein frisch angeheuerter Data Scientist in einer Immobilienfirma. Dieses Beispiel ist frei erfunden; das Ziel ist hier, die wichtigsten Schritte eines Machine-Learning-Projekts zu illustrieren, und nicht, etwas über den Immobilienmarkt zu erfahren. Das sind die wichtigsten Schritte, die es zu bearbeiten gilt:

1. Betrachte das Gesamtbild.
2. Beschaffe die Daten.
3. Erkunde und visualisiere die Daten, um daraus Erkenntnisse zu gewinnen.
4. Bereite die Daten für Machine-Learning-Algorithmen vor.
5. Wähle ein Modell aus und trainiere es.
6. Verfeinere das Modell.
7. Präsentiere die Lösung.
8. Nimm das System in Betrieb, beobachte und warte es.

Der Umgang mit realen Daten

Wenn Sie Machine Learning gerade erst erlernen, experimentieren Sie am besten mit realen Daten, nicht mit künstlich generierten Datensätzen. Glücklicherweise stehen Tausende frei verfügbarer Datensätze aus allen möglichen Fachgebieten zur Auswahl. Hier sind einige Quellen, unter denen Sie passende Daten finden können:

- Beliebte Archive mit frei verfügbaren Daten:
  - OpenML.org (*https://openml.org*)
  - Kaggle.com (*https://kaggle.com*)
  - PapersWithCode.com (*https://paperswithcode.com/datasets*)
  - UC Irvine Machine Learning Repository (*https://archive.ics.uci.edu/ml/*)
  - Datensätze von Amazon AWS (*https://registry.opendata.aws/*)
  - Datensätze von TensorFlow (*https://tensorflow.org/datasets*)

- Metaseiten (Listen von Datenarchiven):
  - Data Portals.org (*https://dataportals.org/*)
  - OpenDataMonitor.eu (*https://opendatamonitor.eu/*)
- Andere Seiten, die beliebte offene Datenarchive auflisten:
  - Wikipedia-Seite mit Machine-Learning-Datensätzen (*https://homl.info/9*)
  - Quora.com (*https://homl.info/10*)
  - subreddit zu Datensätzen (*https://reddit.com/r/datasets*)

Für dieses Kapitel suchen wir uns einen Datensatz zu Immobilienpreisen in Kalifornien aus dem StatLib Repository aus (siehe Abbildung 2-1).[1] Dieser Datensatz basiert auf Informationen aus der kalifornischen Volkszählung von 1990. Er ist nicht gerade aktuell (in der San Francisco Bay konnte man sich damals noch ein nettes Häuschen leisten), bietet aber viele Eigenschaften, anhand deren sich gut lernen lässt. Wir werden deshalb so tun, als wären die Daten aktuell. Wir haben aus didaktischen Gründen auch ein zusätzliches Merkmal hinzugefügt und einige Merkmale entfernt.

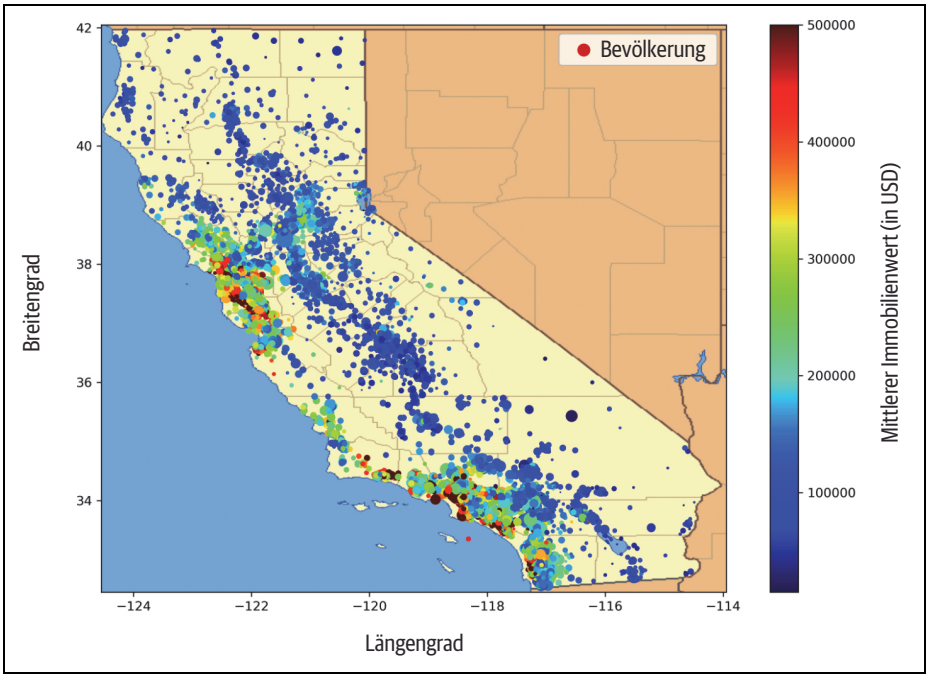

Abbildung 2-1: Immobilienpreise in Kalifornien

1 Der ursprüngliche Datensatz erschien in: R. Kelley Pace and Ronald Barry, »Sparse Spatial Autoregressions«, *Statistics & Probability Letters* 33, no. 3 (1997): 291–297.

Betrachte das Gesamtbild

Willkommen beim Machine-Learning-Immobilienkonzern! Ihre erste Aufgabe ist, ein Modell der Immobilienpreise in Kalifornien mithilfe der Daten aus der kalifornischen Volkszählung zu erstellen. Diese Daten enthalten für jede Blockgruppe in Kalifornien Metriken wie die Bevölkerung, das mittlere Einkommen, die mittleren Immobilienpreise und so weiter. Blockgruppen sind die kleinste geografische Einheit, für die das US Census Bureau Daten veröffentlicht (eine Blockgruppe hat typischerweise eine Bevölkerung von 600 bis 3.000 Menschen). Wir werden diese einfach »Bezirke« nennen.

Ihr Modell sollte aus diesen Daten lernen und in der Lage sein, den mittleren Immobilienpreis in einem beliebigen Bezirk aus allen übrigen Metriken vorherzusagen.

Da Sie ein gut organisierter Data Scientist sein möchten, besteht Ihr erster Arbeitsschritt darin, Ihre Checkliste für Machine-Learning-Projekte zu zücken. Sie können mit der Liste in Anhang A beginnen; diese sollte für die meisten Machine-Learning-Projekte annehmbar funktionieren. Sie sollten sie aber an Ihre Bedürfnisse anpassen. In diesem Kapitel werden wir viele Punkte dieser Checkliste abarbeiten, aber auch einige selbsterklärende oder in späteren Kapiteln besprochene überspringen.

Die Aufgabe abstecken

Die allererste Frage an Ihre Vorgesetzte ist, was denn eigentlich das Geschäftsziel sei; ein Modell zu erstellen, ist vermutlich nicht das eigentliche Ziel. In welcher Weise möchte Ihre Firma das Modell voraussichtlich nutzen und davon profitieren? Von der Antwort hängt ab, wie Sie Ihre Aufgabenstellung formulieren, welche Algorithmen Sie auswählen, welches Qualitätsmaß Sie zum Auswerten des Modells verwenden und wie viel Aufwand Sie in die Optimierung stecken sollten.

Ihre Vorgesetzte antwortet, dass die Ausgabe Ihres Modells (eine Vorhersage des mittleren Immobilienpreises eines Bezirks) zusammen mit anderen *Signalen*[2] in ein anderes Machine-Learning-System eingespeist werden soll (siehe Abbildung 2-2). Dieses nachgeschaltete System soll entscheiden, ob sich Investitionen in einer bestimmten Gegend lohnen oder nicht. Die Richtigkeit dieser Entscheidungen wirkt sich direkt auf die Einnahmen aus.

Die nächste Frage, die Sie Ihrer Chefin stellen sollten, ist, was für eine Lösung bereits verwendet wird (falls überhaupt). Häufig erhalten Sie dabei einen Referenzwert für die Qualität sowie Hinweise zum Lösen der Aufgabe. Ihre Vorgesetzte antwortet Ihnen, dass die Immobilienpreise der Bezirke im Moment von Experten

2 Eine in ein Machine-Learning-System eingegebene Information wird oft nach Claude Shannons Informationstheorie, die er an den Bell Labs entwickelt hat, um die Telekommunikation zu verbessern, als *Signal* bezeichnet: Ein hoher Quotient von Signal und Hintergrundrauschen ist wünschenswert.

manuell geschätzt werden: Ein Team sammelt aktuelle Informationen über einen Bezirk, und wenn es den mittleren Immobilienpreis nicht ermitteln kann, werden diese mithilfe komplexer Regeln geschätzt.

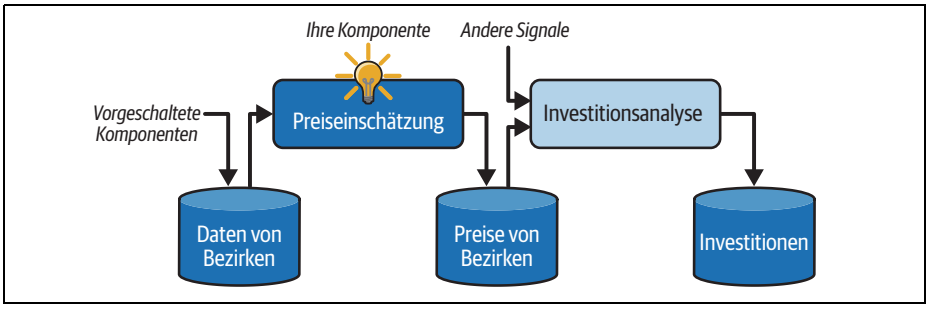

Abbildung 2-2: Eine Machine-Learning-Pipeline für Immobilieninvestitionen

Dieses Verfahren ist sowohl kosten- als auch zeitintensiv, und die Schätzungen sind nicht besonders gut; wenn ein Team den mittleren Immobilienpreis herausfindet, stellt es häufig fest, dass es mit seiner Schätzung um mehr als 10 % danebenlag. Deshalb möchte das Unternehmen ein Modell trainieren, um den mittleren Immobilienpreis eines Bezirks aus anderen Angaben über den Bezirk vorherzusagen. Die Daten aus der Volkszählung könnten eine großartige, für diesen Zweck gut nutzbare Datenquelle sein, da sie den mittleren Immobilienpreis und andere Daten für Tausende Bezirke enthalten.

Pipelines

Eine Abfolge von *Komponenten* zur Datenverarbeitung nennt man eine *Pipeline*. Pipelines sind in Machine-Learning-Systemen sehr häufig, weil dabei eine Menge Daten zu bearbeiten und viele Datentransformationen anzuwenden sind.

Diese Komponenten werden üblicherweise asynchron ausgeführt. Jede Komponente liest eine große Datenmenge ein, verarbeitet sie und schiebt die Ergebnisse in einen anderen Datenspeicher. Etwas später liest die nächste Komponente der Pipeline diese Daten ein, produziert ihre eigene Ausgabe und so weiter. Jede Komponente ist einigermaßen eigenständig: Als Schnittstelle zwischen den Komponenten dient der Datenspeicher. Dadurch ist das System recht einfach zu erfassen (mithilfe eines Datenflussdiagramms), und mehrere Teams können sich auf unterschiedliche Komponenten konzentrieren. Wenn außerdem eine Komponente ausfällt, können die nachgeschalteten Komponenten oft normal weiterarbeiten (zumindest für eine Weile), indem sie einfach die letzte Ausgabe der ausgefallenen Komponente verwenden. Dadurch ist diese Architektur recht robust.

Andererseits kann eine ausgefallene Komponente eine ganze Weile unbemerkt bleiben, falls das System nicht angemessen überwacht wird. Die Daten veralten dann, und die Qualität des Gesamtsystems sinkt.

Mit all diesen Informationen sind Sie nun so weit, Ihr System zu entwerfen. Erstens müssen Sie herausfinden, wie das Training ablaufen soll: Handelt es sich um überwachtes Lernen, unüberwachtes, teilüberwachtes oder selbstüberwachtes Lernen oder Reinforcement Learning? Ist es eine Klassifikationsaufgabe, eine Regressionsaufgabe oder etwas anderes? Sollten Sie Techniken aus dem Batch-Learning oder dem Online-Learning verwenden? Bevor Sie weiterlesen, nehmen Sie sich einen Moment Zeit, und versuchen Sie, sich diese Fragen selbst zu beantworten.

Haben Sie die Antworten gefunden? Schauen wir mal: Es ist ganz klar eine typische überwachte Lernaufgabe, da das Modell mit Beispielen mit *Labels* trainiert werden kann (jeder Datenpunkt enthält die erwartete Ausgabe, d.h. den mittleren Immobilienpreis eines Bezirks). Es ist außerdem eine typische Regressionsaufgabe, da Sie einen Zahlenwert vorhersagen sollen. Genauer gesagt, handelt es sich um eine *multiple Regressionsaufgabe*, da das System mehrere Eigenschaften zum Treffen einer Vorhersage heranziehen wird (die Bevölkerung eines Bezirks, das mittlere Einkommen und so weiter). Gleichzeitig ist es auch eine eine *univariate Regressionsaufgabe*, da wir für jeden Bezirk nur einen einzelnen Wert vorhersagen möchten. Würden wir versuchen, mehrere Werte pro Bezirk vorherzusagen, handelte es sich um eine *multivariate Regressionsaufgabe*. Schließlich gibt es keinen kontinuierlichen Strom neuer Daten in das System. Es gibt keinen besonderen Grund, sich auf schnell veränderliche Daten einzustellen, und der Datensatz ist so klein, dass er im Speicher Platz findet. Daher reicht gewöhnliches Batch-Learning völlig aus.

Wenn die Datenmenge riesig wäre, könnten Sie das Batch-Learning entweder auf mehrere Server verteilen (z.B. mit der *MapReduce*-Technik) oder stattdessen eine Technik zum Online-Learning verwenden.

Wähle ein Qualitätsmaß aus

Der nächste Schritt besteht darin, ein geeignetes Qualitätsmaß auszuwählen. Ein typisches Qualitätsmaß für Regressionsaufgaben ist die *Wurzel der mittleren quadratischen Abweichung* (*Root Mean Square Error*, RMSE). Sie entspricht der Größe des Fehlers, den das System im Mittel bei Vorhersagen macht, wobei großen Fehlern ein höheres Gewicht beigemessen wird. Formel 2-1 zeigt die mathematische Formel zur Berechnung des RMSE.

Formel 2-1: Wurzel der mittleren quadratischen Abweichung (RMSE)

$$\text{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(h(\mathbf{x}^{(i)}) - y^{(i)}\right)^2}$$

Schreibweisen

In dieser Gleichung werden mehrere im Machine Learning übliche Schreibweisen verwendet, die wir im gesamten Buch wiedersehen werden:

- m ist die Anzahl Datenpunkte im Datensatz, für den der RMSE bestimmt wird.
  - Wenn Sie beispielsweise den RMSE für einen Validierungsdatensatz mit 2.000 Bezirken auswerten, dann gilt $m = 2000$.
- $\mathbf{x}^{(i)}$ ist ein Vektor der Werte aller Merkmale (ohne das Label) des i. Datenpunkts im Datensatz, und $y^{(i)}$ ist das dazugehörige Label (der gewünschte Ausgabewert für diesen Datenpunkt).
  - Wenn der erste Bezirk beispielsweise bei –118,29° Länge und 33,91° nördlicher Breite liegt, 1.416 Einwohner mit einem mittleren Einkommen von 38.372 USD hat und der mittlere Immobilienpreis 156.400 USD beträgt (die übrigen Merkmale ignorieren wir noch), dann gilt:

$$\mathbf{x}^{(1)} = \begin{pmatrix} -118,29 \\ 33,91 \\ 1416 \\ 38372 \end{pmatrix}$$

und:

$$y^{(1)} = 156400$$

- $\mathbf{X}$ ist eine Matrix mit den Werten sämtlicher Merkmale (ohne Labels) für alle Datenpunkte im Datensatz. Pro Datenpunkt gibt es eine Zeile, und die i. Zeile entspricht der transponierten Form von $\mathbf{x}^{(i)}$, auch als $(\mathbf{x}^{(i)})^{\mathrm{T}}$ geschrieben.[3]
  - Beispielsweise sieht die Matrix $\mathbf{X}$ für den oben beschriebenen ersten Bezirk folgendermaßen aus:

$$\mathbf{X} = \begin{pmatrix} (\mathbf{x}^{(1)})^{\mathrm{T}} \\ (\mathbf{x}^{(2)})^{\mathrm{T}} \\ \vdots \\ (\mathbf{x}^{(1999)})^{\mathrm{T}} \\ (\mathbf{x}^{(2000)})^{\mathrm{T}} \end{pmatrix} = \begin{pmatrix} -118.29 & 33.91 & 1416 & 38372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

- h ist die Vorhersagefunktion Ihres Systems, auch *Hypothese* genannt. Wenn Ihr System den Merkmalsvektor eines Datenpunkts $\mathbf{x}^{(i)}$ erhält, gibt es den Vorhersagewert $\hat{y}^{(i)} = h(\mathbf{x}^{(i)})$ für diesen Datenpunkt aus.
  - Sagt Ihr System beispielsweise im ersten Bezirk einen mittleren Immobilienpreis von 158.400 USD vorher, ist $\hat{y}^{(1)} = h(\mathbf{x}^{(1)}) = 158400$. Der Vorhersagefehler für diesen Bezirk ist dann $\hat{y}^{(1)} - y^{(1)} = 2000$.

3 Der Transponierungsoperator wandelt einen Spaltenvektor in einen Zeilenvektor um und umgekehrt.

- RMSE(X,h) ist die auf den Beispieldaten mit Ihrer Hypothese h gemessene Kostenfunktion.

Wir verwenden kursive Kleinbuchstaben für Skalare (wie m oder $y^{(i)}$) und Funktionen (wie h), fett gedruckte Kleinbuchstaben für Vektoren (wie $\mathbf{x}^{(i)}$) und fett gedruckte Großbuchstaben für Matrizen (wie $\mathbf{X}$).

Obwohl der RMSE bei Regressionsaufgaben grundsätzlich das Qualitätsmaß erster Wahl ist, sollten Sie in manchen Situationen eine andere Funktion vorziehen. Nehmen wir an, es gäbe viele Ausreißer unter den Bezirken. In diesem Fall könnten Sie den *mittleren absoluten Fehler* (MAE, auch als mittlere absolute Abweichung bezeichnet; siehe Formel 2-2) berücksichtigen:

Formel 2-2: Mittlerer absoluter Fehler

$$\mathrm{MAE}(\mathbf{X}, h) = \frac{1}{m}\sum_{i=1}^{m}\left| h(\mathbf{x}^{(i)}) - y^{(i)} \right|$$

Sowohl RMSE als auch MAE quantifizieren den Abstand zwischen zwei Vektoren: dem Vektor aller Vorhersagen und dem Vektor mit den Zielwerten. Dabei sind unterschiedliche Abstandsmaße oder Normen möglich:

- Die Wurzel einer quadratischen Summe (RMSE) entspricht dem *euklidischen Abstand*: Dies ist der Ihnen vertraute Abstand. Sie wird auch als ℓ_2*-Norm* oder $\|\cdot\|_2$ (oder einfach $\|\cdot\|$) bezeichnet.
- Das Berechnen einer Summe von Absolutwerten (MAE) entspricht der ℓ_1*-Norm*, geschrieben als $\|\cdot\|_1$. Bisweilen nennt man sie auch *Manhattan-Metrik*, da sie den Abstand zwischen zwei Punkten in einer Stadt angibt, in der man sich nur entlang rechtwinkliger Häuserblöcke bewegen kann.
- Allgemeiner ist die ℓ_k*-Norm* eines Vektors $\mathbf{v}$ mit n Elementen als $\|\mathbf{v}\|_k = \left(|v_1|^k + |v_2|^k + \ldots + |v_n|^k\right)^{\frac{1}{k}}$ definiert. ℓ_0 gibt einfach nur die Anzahl der Elemente ungleich null im Vektor wieder, und ℓ_∞ berechnet den größten Absolutwert im Vektor.
- Je höher der Index einer Norm ist, umso stärker berücksichtigt er große Werte und vernachlässigt kleinere. Deshalb ist der RMSE empfindlicher für Ausreißer als der MAE. Sind Ausreißer aber exponentiell selten (wie in einer Glockenkurve), funktioniert der RMSE sehr gut und ist grundsätzlich vorzuziehen.

Überprüfe die Annahmen

Es ist eine gute Angewohnheit, die bisher (von Ihnen oder von anderen) getroffenen Annahmen aufzuschreiben und zu überprüfen; damit können Sie größere Schwierigkeiten früh erkennen. Die von Ihrem System vorhergesagten Preise für einzelne Bezirke werden in ein nachgeschaltetes Machine-Learning-System eingegeben. Wir nehmen an, dass die Preise als solche verwendet werden. Was aber, wenn das nachgeschaltete System stattdessen die Preise in Kategorien einteilt (z.B. »günstig«, »mittel« oder »teuer«) und anstelle der Preise dann diese Kategorien verwendet? In dem Fall wäre es überhaupt nicht wichtig, den Preis genau vorherzusagen; Ihr System müsste lediglich die Kategorie richtig bestimmen. Ist das der Fall, sollte die Aufgabe als Klassifikation beschrieben werden, nicht als Regression. Zu dieser Art von Erkenntnissen möchte man nicht erst nach monatelanger Arbeit an einem Regressionssystem gelangen.

Nachdem Sie mit dem für das nachgeschaltete System zuständigen Team gesprochen haben, sind Sie sich glücklicherweise sicher, dass Sie tatsächlich die Preise und keine Kategorien benötigen. Großartig! Damit sind wir gut aufgestellt und haben grünes Licht, um mit dem Programmieren zu beginnen!

Beschaffe die Daten

Es ist Zeit, sich die Hände schmutzig zu machen. Sie können sich gern Ihren Laptop nehmen und die folgenden Codebeispiele in einem Jupyter Notebook nachvollziehen. Wie schon in der Einleitung erwähnt, sind alle Codebeispiele in diesem Buch Open Source. Sie finden sie als Jupyter Notebook unter *https://github.com/ageron/handson-mls3*. Dabei handelt es sich um interaktive Dokumente mit Text, Bildern und ausführbaren Codeschnipseln (in unserem Fall in Python). In diesem Buch gehe ich davon aus, dass Sie diese Notebooks auf Google Colab ausführen – einem kostenlosen Dienst, auf dem Sie beliebige Jupyter Notebooks direkt online laufen lassen können, ohne etwas auf Ihrem Rechner installieren zu müssen. Wollen Sie eine andere Onlineplattform nutzen (zum Beispiel Kaggle) oder möchten Sie alles lokal auf Ihrem eigenen Computer installieren, schauen Sie sich die Anleitungen dazu bitte auf der GitHub-Seite des Buchs an.

Die Codebeispiele mit Google Colab ausführen

Öffnen Sie einen Webbrowser und rufen Sie *https://homl.info/colab3* auf: Damit landen Sie bei Google Colab, und es wird eine Liste der Jupyter Notebooks für dieses Buch angezeigt (siehe Abbildung 2-3). Es gibt ein Notebook pro Kapitel und dazu ein paar zusätzliche Notebooks und Tutorials für NumPy, Matplotlib, Pandas, lineare Algebra und die Differenzialrechnung. Klicken Sie beispielsweise auf *02_end_to_end_machine_learning_project.ipynb*, wird das Notebook für Kapitel 2 in Google Colab geöffnet (siehe Abbildung 2-4).

Ein Jupyter Notebook besteht aus einer Liste mit Zellen. Jede Zelle enthält entweder ausführbaren Code oder Text. Klicken Sie einmal auf die erste Textzelle (die den Satz »Welcome to Machine Learning Housing Corp.!« enthält). Damit wird sie zum Editieren geöffnet. Beachten Sie, dass Jupyter Notebooks die Markdown-Syntax zum Formatieren nutzen (zum Beispiel `**fett**`, `*kursiv*`, `# Titel`, `[Link text](url)` und so weiter). Versuchen Sie, diesen Text zu verändern, und drücken Sie dann Umschalt-Enter, um das Ergebnis zu sehen.

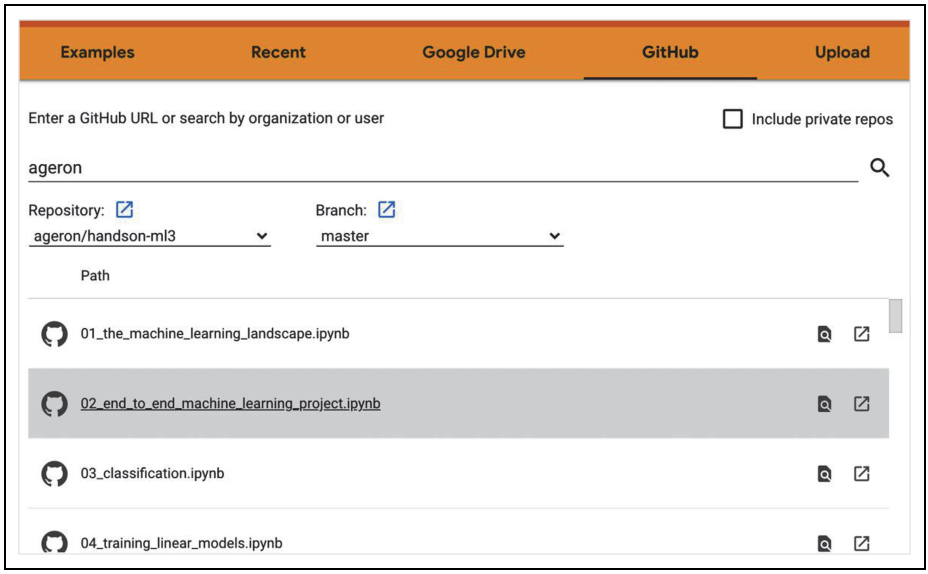

Abbildung 2-3: Liste mit Notebooks in Google Colab

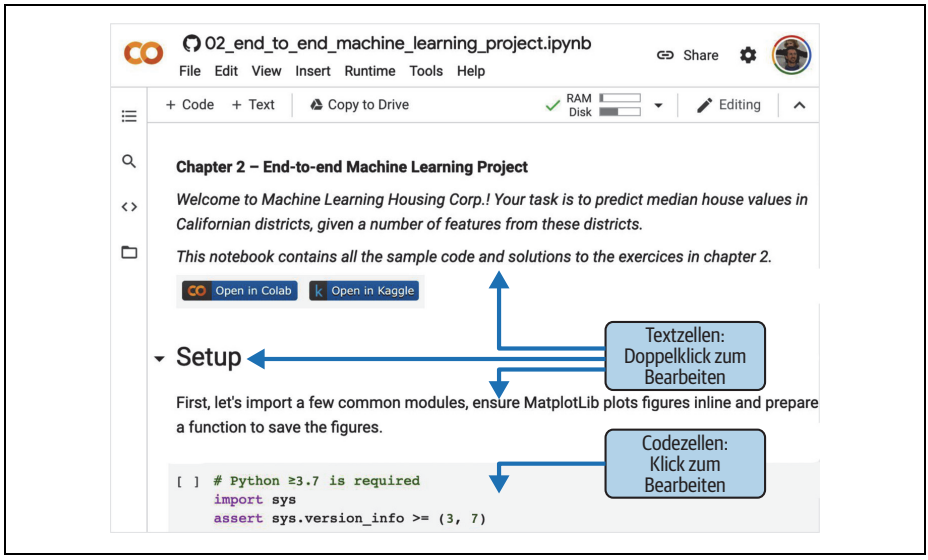

Abbildung 2-4: Ihr Notebook in Google Colab

Als Nächstes erstellen Sie eine neue Codezelle, indem Sie »Insert → Code cell« aus dem Menü wählen. Alternativ können Sie auf den Button »+ Code« in der Toolbar klicken oder sich mit Ihrer Maus über das untere Ende einer Zelle bewegen, bis Sie »+ Code« und »+ Text« sehen, und dann auf »+ Code« klicken. In die neue Codezelle geben Sie etwas Python-Code ein, wie zum Beispiel `print("Hello World")`, und drücken dann Umschalt-Enter, um diesen Code auszuführen (oder Sie klicken auf den ▷-Button auf der linken Seite der Zelle).

Sind Sie nicht mit Ihrem Google-Account angemeldet, werden Sie gebeten, dies nun nachzuholen (wenn Sie noch keinen Google-Account haben, müssen Sie einen erstellen). Sind Sie angemeldet und versuchen Sie, den Code auszuführen, werden Sie eine Sicherheitswarnung erhalten, die Ihnen mitteilt, dass dieses Notebook nicht von Google erstellt wurde. Eine böswillige Person könnte ein Notebook erstellen, das versucht, Sie dazu zu bringen, Ihre Anmeldedaten einzugeben, sodass sie auf Ihre persönlichen Daten zugreifen kann. Bevor Sie also ein Notebook ausführen, stellen Sie immer sicher, dass Sie dessen Autor oder Autorin vertrauen (oder prüfen Sie genau, was jede Codezelle tut, bevor Sie sie starten). Wenn Sie mir vertrauen (oder vorhaben, jede Codezelle genau zu prüfen), können Sie nun auf »Run anyway« klicken.

Colab wird jetzt eine neue *Runtime* für Sie allokieren: Dabei handelt es sich um eine kostenlose virtuelle Maschine, die auf den Google-Servern läuft und eine Reihe von Tools und Python-Bibliotheken enthält – unter anderem all das, was Sie für die meisten Kapitel brauchen (in manchen Kapiteln müssen Sie einen Befehl ausführen, um zusätzliche Bibliotheken zu installieren). Das wird ein paar Sekunden brauchen. Als Nächstes wird sich Colab automatisch mit dieser Runtime verbinden und sie verwenden, um Ihre neue Codezelle auszuführen. Wichtig zu wissen ist, dass der Code auf der Runtime läuft, *nicht* auf Ihrem Rechner. Die Ausgabe des Codes wird unter der Zelle angezeigt. Herzlichen Glückwunsch – Sie haben Python-Code auf Colab ausgeführt!

Um eine neue Codezelle einzufügen, können Sie auch Strg-M (oder Cmd-M in macOS), gefolgt von einem A (um sie oberhalb der aktiven Zelle einzufügen) oder einem B (zum Einfügen unterhalb) drücken. Es gibt noch viele weitere Tastenkürzel: Sie können sich diese mit Strg-M (oder Cmd-M), gefolgt von H anzeigen lassen und sie anpassen. Entscheiden Sie sich dazu, die Notebooks auf Kaggle oder auf Ihrem eigenen Rechner mit JupyterLab oder einer IDE wie Visual Studio Code mit der Jupyter-Extension laufen zu lassen, werden Ihnen ein paar kleine Unterschiede auffallen – Runtimes werden als *Kernel* bezeichnet, die Benutzeroberfläche und die Tastenkürzel sind etwas anders und so weiter –, aber ein Wechsel von einer Jupyter-Umgebung zu einer anderen ist nicht allzu schwer.

Ihre Codeänderungen und Daten sichern

Sie können Änderungen an Colab-Notebooks vornehmen, aber diese bleiben nur so lange bestehen, wie Sie Ihren Browser-Tab offen lassen. Sobald Sie ihn schließen, sind sie verloren. Um das zu verhindern, achten Sie darauf, eine Kopie des Notebooks auf Ihrem Google Drive zu sichern, indem Sie »File → Save a copy in Drive« anklicken. Alternativ laden Sie das Notebook auf Ihren Computer herunter, indem Sie »File → Download → Download .ipynb« aufrufen. Dann können Sie zu einem späteren Zeitpunkt *https://colab.research.google.com* wieder aufrufen und das Notebook erneut öffnen (entweder von Google Drive oder indem Sie es von Ihrem Computer hochladen).

Google Colab ist nur für eine interaktive Nutzung gedacht: Sie können in den Notebooks herumspielen und am Code drehen, aber Sie können sie nicht längere Zeit unbeaufsichtigt laufen lassen, da dann die Runtime heruntergefahren wird und alle Daten verloren gehen.

Erzeugt das Notebook Daten, die Ihnen wichtig sind, sollten Sie dafür sorgen, dass Sie sie herunterladen, bevor die Runtime beendet wird. Dazu klicken Sie auf das »Files«-Icon (siehe Schritt 1 in Abbildung 2-5), suchen nach der Datei, die Sie herunterladen wollen, klicken auf die vertikalen Punkte daneben (Schritt 2) und dann auf »Download« (Schritt 3). Alternativ können Sie Ihr Google Drive auf der Runtime mounten, sodass das Notebook Dateien direkt von Google Drive lesen und darauf schreiben kann – so, als würde es sich um ein lokales Verzeichnis handeln. Dazu klicken Sie auf das »Files«-Icon (Schritt 1), dann auf das Google-Drive-Icon (in Abbildung 2-5 umkreist) und folgen den Anweisungen auf dem Bildschirm.

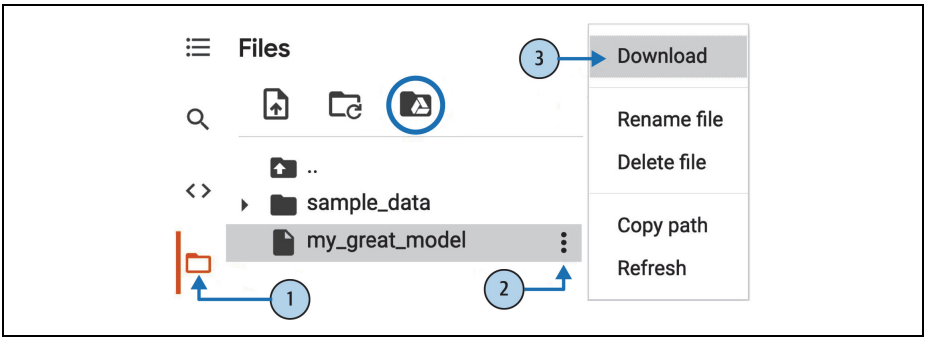

Abbildung 2-5: Eine Datei von einer Google Colab Runtime herunterladen (Schritte 1 bis 3) oder Ihr Google Drive mounten (eingekreist)

Standardmäßig wird Ihr Google Drive unter */content/drive/MyDrive* gemountet. Wollen Sie eine Datendatei sichern, kopieren Sie sie einfach in dieses Verzeichnis, indem Sie `!cp /content/my_great_model /content/drive/MyDrive` aufrufen. Alle Befehle, die mit einem Ausrufezeichen (`!`) beginnen, werden als Shell-Befehl und nicht

als Python-Code behandelt: cp ist der Linux-Shell-Befehl, mit dem sich eine Datei von einem Pfad in einen anderen kopieren lässt. Colab-Runtimes laufen unter Linux (genauer unter Ubuntu).

Interaktivität – mächtig, aber gefährlich

Jupyter Notebooks sind interaktiv, und das ist eine tolle Sache: Sie können jede Zelle für sich ausführen, jederzeit stoppen, eine Zelle einfügen, mit dem Code spielen, einen Schritt zurückgehen und dieselbe Zelle erneut ausführen und so weiter – und ich möchte Sie dazu ermutigen, das auch unbedingt zu tun. Führen Sie die Zellen einfach eine nach der anderen aus, ohne mit ihnen zu experimentieren, werden Sie nicht so schnell dazulernen. Aber diese Flexibilität hat ihren Preis: Es ist sehr leicht, Zellen in der falschen Reihenfolge auszuführen oder zu vergessen, eine Zelle zu starten. Wenn das passiert, wird der folgende Code sehr wahrscheinlich Probleme haben. So enthält beispielsweise die allererste Codezelle in jedem Notebook Setup-Code (wie zum Beispiel Importe), der auf jeden Fall als Erstes ausgeführt werden sollte, weil sonst nichts funktionieren wird.

Sehen Sie sich einmal einem seltsamen Fehler gegenüber, versuchen Sie, die Runtime neu zu starten (im Menü über »Runtime → Restart runtime«) und dann alle Zellen vom Beginn des Notebooks an erneut auszuführen. Das löst oft das Problem. Wenn das nicht hilft, hat wahrscheinlich eine Ihrer Änderungen für den Fehler gesorgt: Setzen Sie es einfach auf den Originalzustand zurück und versuchen Sie es erneut. Gibt es dann immer noch Probleme, melden Sie das bitte bei GitHub.

Code im Buch versus Notebook-Code

Sie werden manchmal kleine Unterschiede zwischen dem Code in diesem Buch und dem Code in den Notebooks finden. Das kann aus verschiedenen Gründen passieren:

- Eine Bibliothek hat sich ein wenig geändert, wenn Sie diese Zeilen lesen, oder vielleicht habe ich selbst trotz aller Bemühungen einen Fehler im Buch gemacht. Leider kann ich den Code in Ihrem Buch nicht per Magie beheben (sofern Sie nicht eine elektronische Variante nutzen und die neueste Version herunterladen), aber ich kann die Notebooks korrigieren. Stolpern Sie also über einen Fehler, nachdem Sie Code aus diesem Buch kopiert haben, suchen Sie bitte in den Notebooks nach der korrigierten Version: Ich werde versuchen, sie fehlerfrei und mit den aktuellsten Versionen der Bibliotheken lauffähig zu halten.
- Die Notebooks enthalten zusätzlichen Code, um die Abbildungen schöner zu machen (zusätzliche Bezeichnungen, Anpassen der Schriftgröße und so weiter) und mit höherer Auflösung für das Buch sichern zu können. Diesen zusätzlichen Code können Sie bedenkenlos ignorieren, wenn Sie das wollen.

Ich habe den Code so optimiert, dass er besser lesbar und einfacher ist: Er ist so linear und flach wie möglich, und er definiert sehr wenige Funktionen oder Klassen. Ziel ist, sicherzustellen, dass sich der Code, den Sie ausführen, direkt vor Ihnen befindet und nicht irgendwo unter diversen Abstraktionsschichten begraben ist, die Sie erst beiseiteräumen müssen. Dies erleichtert auch das Experimentieren mit dem Code. Aus Gründen der Einfachheit gibt es keine Fehlerbehandlung, und ich habe ein paar der am wenigsten gebräuchlichen Importe direkt dort untergebracht, wo ich sie brauchte (statt sie am Beginn der Datei zu platzieren, wie das durch den PEP 8 Python Style Guide empfohlen wird). Ihr produktiver Code wird aber trotzdem nicht groß anders sein – nur etwas modularer sowie mit Fehlerbehandlung und mehr Tests.

Okay! Nachdem Sie sich etwas mit Colab vertraut gemacht haben, können Sie nun die Daten herunterladen.

Die Daten herunterladen

In einer typischen Arbeitsumgebung wären Ihre Daten in einer relationalen Datenbank (oder einem anderen Datenspeicher) und über viele Tabellen, Dokumente oder Dateien verteilt. Um auf sie zuzugreifen, müssten Sie zuerst Zugriffsrechte und Passwörter erhalten[4] und sich mit dem Datenmodell vertraut machen. In diesem Projekt sind die Dinge jedoch deutlich einfacher: Sie laden *housing.tgz* herunter, eine einzelne komprimierte Datei, in der sämtliche Daten als kommaseparierte Datei (CSV) namens *housing.csv* vorliegen.

Statt die Daten manuell herunterzuladen und auszupacken, ist es im Allgemeinen besser, eine Funktion zu schreiben, die das für Sie erledigt. Eine solche Funktion ist besonders dann nützlich, wenn sich die Daten regelmäßig ändern, weil Sie so die jeweils neuesten Daten mit einem selbst geschriebenen Skript herunterladen können (Sie könnten dieses automatisch in regelmäßigen Abständen ausführen lassen). Den Prozess der Datenbeschaffung zu automatisieren, hilft außerdem, wenn Sie den Datensatz auf mehreren Maschinen installieren möchten.

Mit der folgenden Funktion können Sie die Daten herunterladen:

```
from pathlib import Path
import pandas as pd
import tarfile
import urllib.request

def load_housing_data():
    tarball_path = Path("datasets/housing.tgz")
    if not tarball_path.is_file():
        Path("datasets").mkdir(parents=True, exist_ok=True)
        url = "https://github.com/ageron/data/raw/main/housing.tgz"
```

4 Sie müssten eventuell auch gesetzliche Vorgaben berücksichtigen, z. B. Felder mit persönlichen Daten, die niemals in ungeschützte Datenspeicher kopiert werden dürfen.

```
        urllib.request.urlretrieve(url, tarball_path)
        with tarfile.open(tarball_path) as housing_tarball:
            housing_tarball.extractall(path="datasets")
    return pd.read_csv(Path("datasets/housing/housing.csv"))

housing = load_housing_data()
```

Wenn Sie nun `load_housing_data()` aufrufen, wird nach der Datei *datasets/housing.tgz* gesucht. Wird diese nicht gefunden, erstellt die Funktion das Verzeichnis *datasets* im aktuellen Verzeichnis (was in Colab standardmäßig */content* ist), lädt *housing.tgz* aus dem GitHub-Repository *ageron/data* herunter und extrahiert sie in das Verzeichnis *datasets* – das erzeugt das Verzeichnis *datasets/housing* mit der Datei *housing.csv*. Schließlich lädt die Funktion diese CSV-Datei in ein DataFrame-Objekt von Pandas und gibt es zurück.

Wirf einen kurzen Blick auf die Datenstruktur

Schauen wir uns die ersten fünf Zeilen des DataFrames mit der Methode `head()` an (siehe Abbildung 2-6).

```
housing.head()
```

	longitude	latitude	housing_median_age	median_income	ocean_proximity	median_house_value
0	-122.23	37.88	41.0	8.3252	NEAR BAY	452600.0
1	-122.22	37.86	21.0	8.3014	NEAR BAY	358500.0
2	-122.24	37.85	52.0	7.2574	NEAR BAY	352100.0
3	-122.25	37.85	52.0	5.6431	NEAR BAY	341300.0
4	-122.25	37.85	52.0	3.8462	NEAR BAY	342200.0

Abbildung 2-6: Die ersten fünf Zeilen im Datensatz

Jede Zeile steht für einen Bezirk. Es gibt zehn Merkmale (nicht alle sind im Screenshot zu sehen): `longitude`, `latitude`, `housing_median_age`, `total_rooms`, `total_bedrooms`, `population`, `households`, `median_income`, `median_house_value` und `ocean_proximity`.

Die Methode `info()` hilft, schnell eine Beschreibung der Daten zu erhalten. Dies sind insbesondere die Anzahl der Zeilen, der Typ jedes Attributs und die Anzahl der Werte ungleich null:

```
>>> housing.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   longitude           20640 non-null  float64
 1   latitude            20640 non-null  float64
 2   housing_median_age  20640 non-null  float64
```

```
 3   total_rooms         20640 non-null  float64
 4   total_bedrooms      20433 non-null  float64
 5   population          20640 non-null  float64
 6   households          20640 non-null  float64
 7   median_income       20640 non-null  float64
 8   median_house_value  20640 non-null  float64
 9   ocean_proximity     20640 non-null  object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

Enthält in diesem Buch ein Codebeispiel wie dieses hier eine Kombination aus Code und Ausgabe, wird es aus Gründen der besseren Lesbarkeit wie im Python-Interpreter formatiert: Den Codezeilen ist >>> (oder ... für eingerückte Blöcke) vorangestellt, während die Ausgabe kein Präfix besitzt.

Im Datensatz gibt es 20.640 Datenpunkte. Damit ist er für Machine-Learning-Verhältnisse eher klein, für den Anfang ist das aber ausgezeichnet! Beachten Sie, dass das Merkmal `total_bedrooms` nur 20.433 Werte ungleich null hat, es gibt also 207 Bezirke ohne diese Angabe. Darum werden wir uns später kümmern müssen.

Bis auf das Feld `ocean_proximity` sind sämtliche Merkmale numerisch. Dessen Typ ist `object`, und es könnte beliebige Python-Objekte enthalten. Da Sie aber diese Daten aus einer CSV-Datei geladen haben, muss es sich dabei natürlich um Text handeln. Beim Betrachten der ersten fünf Zeilen haben Sie möglicherweise bemerkt, dass sich die Werte in der Spalte `ocean_proximity` wiederholen. Es handelt sich dabei also um ein kategorisches Merkmal. Sie können mit der Methode `value_counts()` herausfinden, welche Kategorien es gibt und wie viele Bezirke zu jeder Kategorie gehören:

```
>>> housing["ocean_proximity"].value_counts()
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

Betrachten wir auch die anderen Spalten. Die Methode `describe()` fasst die numerischen Merkmale zusammen (siehe Abbildung 2-7).

Die Zeilen `count`, `mean`, `min` und `max` sind selbsterklärend. Beachten Sie, dass die leeren Werte ignoriert werden (z.B. beträgt `count` bei `total_bedrooms` 20433, nicht 20640). Die Zeile `std` enthält die *Standardabweichung*, die die Streuung der Werte angibt.[5] Die Zeilen mit `25%`, `50%` und `75%` zeigen die entsprechenden *Perzentile*: Ein

5 Die Standardabweichung wird im Allgemeinen mit σ angegeben (dem griechischen Buchstaben Sigma) und ist die Quadratwurzel der *Varianz*, der durchschnittlichen quadratischen Abweichung vom Mittelwert. Wenn ein Merkmal der sehr häufigen glockenförmigen *Normalverteilung* folgt (auch *Gaußverteilung* genannt), gilt die »68-95-99,7«-Regel: Etwa 68 % der Werte liegen innerhalb von 1σ des Mittelwerts, 95 % innerhalb von 2σ und 99,7 % innerhalb von 3σ.

Perzentil besagt, dass ein bestimmter prozentualer Anteil der Beobachtungen unterhalb eines Werts liegt. Beispielsweise haben 25% der Bezirke ein `housing_median_age` unter 18, 50% liegen unter 29, und 75% liegen unter 37. Diese nennt man oft das 25. Perzentil (oder 1. *Quartil*), den Median und das 75. Perzentil (oder 3. Quartil).

```
housing.describe()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	median_house_value
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	206855.816909
std	2.003532	2.135952	12.585558	2181.615252	421.385070	115395.615874
min	-124.350000	32.540000	1.000000	2.000000	1.000000	14999.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	119600.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	179700.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	264725.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	500001.000000

Abbildung 2-7: Zusammenfassung aller numerischen Merkmale

Eine andere Möglichkeit, schnell einen Eindruck von den Daten, die wir sehen, zu erhalten, ist, für jedes numerische Merkmal ein Histogramm zu plotten. Ein Histogramm zeigt die Anzahl Datenpunkte (auf der vertikalen Achse), die in einem bestimmten Wertebereich (auf der horizontalen Achse) liegen. Sie können diese entweder für jedes Merkmal einzeln plotten oder die Methode `hist()` für den gesamten Datensatz aufrufen (wie im folgenden Codebeispiel) und für jedes numerische Merkmal ein Histogramm erhalten (siehe Abbildung 2-8):

```
import matplotlib.pyplot as plt

housing.hist(bins=50, figsize=(12, 8))
plt.show()
```

In diesen Histogrammen gibt es einiges zu sehen:

- Erstens sieht das mittlere Einkommen nicht nach Werten in US-Dollar aus (USD). Nachdem Sie sich mit dem Team, das die Daten erhoben hat, in Verbindung gesetzt haben, erfahren Sie, dass die Daten skaliert wurden und für höhere mittlere Einkommen nach oben bei 15 (genau 15,0001) und für geringere mittlere Einkommen nach unten bei 0,5 (genau 0,4999) abgeschnitten wurden. Die Zahlen stehen ungefähr für 10.000 USD (eine 3 bedeutet also etwa 30.000 USD). Es ist im Machine Learning durchaus üblich, mit solchen vorverarbeiteten Merkmalen zu arbeiten, was nicht notwendigerweise ein Problem darstellt. Sie sollten aber versuchen, nachzuvollziehen, wie die Daten berechnet wurden.
- Das mittlere Alter und der mittlere Wert von Gebäuden wurden ebenfalls gekappt. Letzteres könnte sich als ernstes Problem herausstellen, da Ihre Ziel-

größe (Ihr Label) betroffen ist. Ihre Machine-Learning-Algorithmen könnten dann lernen, dass es keine Preise jenseits dieser Obergrenze gibt. Sie müssen mit Ihrem Team (dem Team, das die Ausgabe Ihres Systems nutzen möchte) klären, ob das ein Problem darstellt oder nicht. Wenn Ihnen erklärt wird, dass auch jenseits von 500.000 USD präzise Vorhersagen nötig sind, haben Sie zwei Alternativen:

  - Für die nach oben begrenzten Bezirke korrekte Labels zu sammeln.
  - Die entsprechenden Bezirke aus dem Trainingsdatensatz zu entfernen (auch aus dem Testdatensatz, da Ihr System nicht als schlechter eingestuft werden sollte, wenn es Werte jenseits von 50.000 USD vorhersagt).

- Diese Attribute haben sehr unterschiedliche Wertebereiche. Wir werden das weiter unten in diesem Kapitel besprechen, wenn wir uns dem Skalieren von Merkmalen widmen.
- Schließlich sind viele der Histogramme *rechtsschief*: Sie erstrecken sich viel weiter vom Median nach rechts als nach links. Dadurch wird das Erkennen von Mustern für einige Machine-Learning-Algorithmen schwieriger. Wir werden später versuchen, diese Merkmale zu einer annähernd glockenförmigen Verteilung zu transformieren.

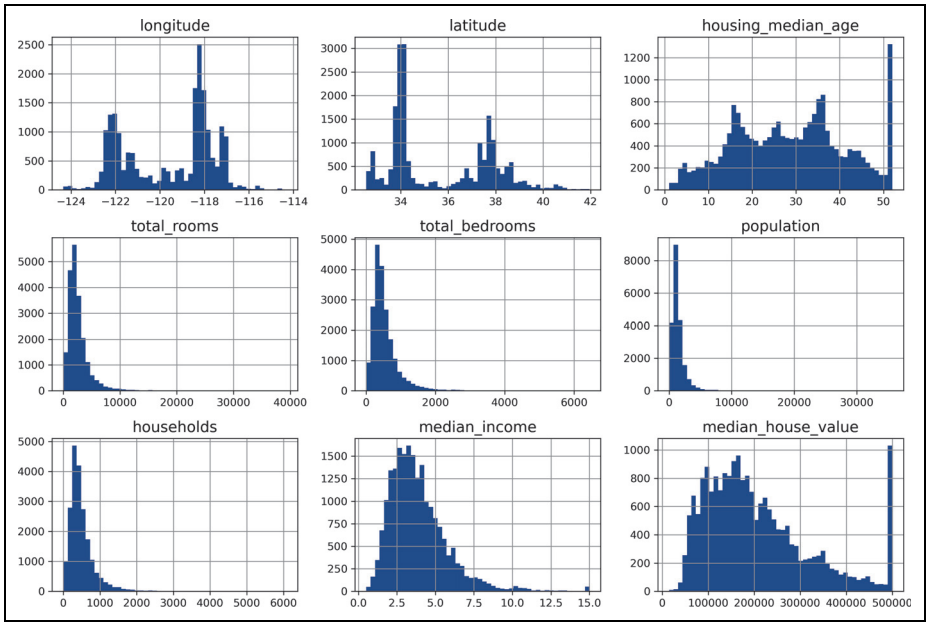

Abbildung 2-8: Ein Histogramm für jedes numerische Attribut

Hoffentlich haben Sie nun einen besseren Eindruck von den Daten, mit denen Sie sich beschäftigen.

Warten Sie! Bevor Sie sich die Daten weiter ansehen, sollten Sie einen Testdatensatz erstellen, beiseitelegen und nicht hineinschauen.

Erstelle einen Testdatensatz

Es mag sich seltsam anhören, an dieser Stelle einen Teil der Daten freiwillig beiseitezulegen. Schließlich haben wir gerade erst einen kurzen Blick auf die Daten geworfen, und Sie sollten bestimmt noch weiter analysieren, bevor Sie sich für einen Algorithmus entscheiden, oder? Das ist zwar richtig, aber Ihr Gehirn besitzt ein faszinierendes System zur Mustererkennung. Es ist daher äußerst anfällig für Overfitting: Wenn Sie sich die Testdaten ansehen, könnten Sie auf ein interessantes Muster im Datensatz stoßen, das Sie zur Auswahl eines bestimmten Machine-Learning-Modells veranlasst. Wenn Sie den Fehler der Verallgemeinerung anhand des Testdatensatzes schätzen, wird Ihr Schätzwert zu optimistisch ausfallen, und Sie würden in der Folge ein System starten, das die erwartete Vorhersagequalität nicht erfüllt. Dies nennt man auch das *Data-Snooping-Bias*.

Einen Testdatensatz zu erstellen, ist theoretisch einfach: Wählen Sie zufällig einige Datenpunkte aus, meist 20% des Datensatzes (oder weniger, wenn Ihr Datensatz sehr groß ist), und legen Sie sie beiseite:

```
import numpy as np

def shuffle_and_split_data(data, test_ratio):
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

Sie können diese Funktion anschließend folgendermaßen verwenden:

```
>>> train_set, test_set = shuffle_and_split_data(housing, 0.2)
>>> len(train_set)
16512
>>> len(test_set)
4128
```

Das funktioniert, ist aber noch nicht perfekt: Wenn Sie dieses Programm erneut ausführen, erzeugt es einen anderen Testdatensatz! Sie (oder Ihre Machine-Learning-Algorithmen) werden mit der Zeit den kompletten Datensatz als Gesamtes sehen, was Sie ja genau vermeiden möchten.

Eine Lösungsmöglichkeit besteht darin, den Testdatensatz beim ersten Durchlauf zu speichern und in späteren Durchläufen zu laden. Eine andere Möglichkeit ist,

den Seed-Wert des Zufallsgenerators festzulegen (z.B. mit `np.random.seed(42)`[6]), bevor Sie `np.random.permutation()` aufrufen, sodass jedes Mal die gleichen durchmischten Indizes generiert werden.

Allerdings scheitern beide Lösungsansätze, sobald Sie einen aktualisierten Datensatz erhalten. Um auch danach über eine stabile Trennung zwischen Trainings- und Testdatensatz zu verfügen, können Sie als Alternative einen eindeutigen Identifikator verwenden, um zu entscheiden, ob ein Datenpunkt in den Testdatensatz aufgenommen werden soll oder nicht (vorausgesetzt, die Datenpunkte haben eindeutige, unveränderliche Identifikatoren). Sie könnten beispielsweise aus dem Identifikator eines Datenpunkts einen Hash berechnen und den Datenpunkt in den Testdatensatz aufzunehmen, falls der Hash kleiner oder gleich 20% des maximalen Hashwerts ist. Damit stellen Sie sicher, dass der Testdatensatz über mehrere Durchläufe konsistent ist, selbst wenn Sie ihn aktualisieren. Ein neuer Datensatz enthält auf diese Weise 20% der neuen Datenpunkte, aber keinen der Datenpunkte, die zuvor im Trainingsdatensatz waren.

Hier folgt eine mögliche Implementierung:

```
from zlib import crc32

def is_id_in_test(identifier, test_ratio):
    return crc32(np.int64(identifier)) < test_ratio * 2**32

def split_data_with_id_hash(data, test_ratio, id_column):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: is_in_test_set(id_, test_ratio))
    return data.loc[~in_test_set], data.loc[in_test_set]
```

Leider gibt es im Immobilien-Datensatz keine Identifikatorspalte. Die Lösung ist, den Zeilenindex als ID zu nutzen:

```
housing_with_id = housing.reset_index()   # fügt die Spalte `index` hinzu
train_set, test_set = split_data_with_id_hash(housing_with_id, 0.2, "index")
```

Wenn Sie den Zeilenindex als eindeutigen Identifikator verwenden, müssen Sie sicherstellen, dass die neuen Daten am Ende des Datensatzes angehängt werden und nie eine Zeile gelöscht wird. Falls das nicht möglich ist, können Sie immer noch versuchen, einen eindeutigen Identifikator aus den stabilsten Merkmalen zu entwickeln. Beispielsweise werden geografische Länge und Breite garantiert für die nächsten paar Millionen Jahre stabil bleiben, daher könnten Sie diese folgendermaßen zu einer ID kombinieren:[7]

6 Sie werden häufig sehen, dass der Seed-Wert auf 42 gesetzt wird. Diese Zahl hat keine besondere Bedeutung, außer dass sie die Antwort auf die ultimative Frage nach dem Leben, dem Universum und dem ganzen Rest ist.

7 Die Koordinatenangaben sind recht grob, daher werden viele Bezirke eine identische ID erhalten und somit im gleichen Teildatensatz landen (Test oder Training). Damit haben wir unglücklicherweise ein Bias in der Auswahl.

```
housing_with_id["id"] = housing["longitude"] * 1000 + housing["latitude"]
train_set, test_set = split_data_with_id_hash(housing_with_id, 0.2, "id")
```

Scikit-Learn enthält einige Funktionen, die Datensätze auf unterschiedliche Weise in Teildatensätze aufteilen. Die einfachste Funktion darunter ist `train_test_split()`, die so ziemlich das Gleiche tut wie die oben definierte Funktion `shuffle_and_split_data()`, aber einige zusätzliche Optionen bietet. Erstens gibt es den Parameter `random_state`, der den Seed-Wert des Zufallszahlengenerators festlegt, und zweitens können Sie mehrere Datensätze mit einer identischen Anzahl Zeilen übergeben, die anhand der gleichen Indizes aufgeteilt werden (das ist sehr nützlich, z. B. wenn Sie ein separates DataFrame mit den Labels haben):

```
from sklearn.model_selection import train_test_split

train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
```

Bisher haben wir ausschließlich zufallsbasierte Methoden zur Stichprobenauswahl betrachtet. Wenn Ihr Datensatz groß genug ist (insbesondere im Vergleich zur Anzahl der Merkmale), ist daran nichts auszusetzen. Ist er aber nicht groß genug, besteht das Risiko, ein erhebliches Stichproben-Bias zu verursachen. Wenn ein Umfrageunternehmen 1.000 Personen anruft, um diesen Fragen zu stellen, wählt es nicht einfach nur zufällig 1.000 Probanden aus dem Telefonbuch aus. Es versucht, dafür zu sorgen, dass diese 1.000 Personen repräsentativ für die gesamte Bevölkerung sind – zumindest in Bezug auf die zu stellenden Fragen. Beispielsweise besteht die Bevölkerung der USA aus 51,1 % Frauen und 48,9 % Männern, also sollte eine gut aufgebaute Studie in den USA dieses Verhältnis auch in der Stichprobe repräsentieren: 511 Frauen und 489 Männer (zumindest wenn es möglich erscheint, dass die Antworten vom Geschlecht abhängen). Dies bezeichnet man als *stratifizierte Stichprobe*: Die Bevölkerung wird in homogene Untergruppen, die *Strata*, aufgeteilt, und aus jedem Stratum wird die korrekte Anzahl Datenpunkte ausgewählt. Damit ist garantiert, dass der Testdatensatz die Gesamtbevölkerung angemessen repräsentiert. Würde die Stichprobe rein zufällig ausgewählt, gäbe es eine 10,7%ige Chance, dass die Stichprobe verzerrt ist und entweder weniger als 48,5 % Frauen oder mehr als 53,5 % Frauen im Datensatz enthalten sind. In beiden Fällen wären die Ergebnisse wahrscheinlich mit einem erheblichen Bias behaftet.

Nehmen wir an, Experten hätten Ihnen in einer Unterhaltung erklärt, dass das mittlere Einkommen ein sehr wichtiges Merkmal zur Vorhersage des mittleren Immobilienpreises ist. Sie möchten sicherstellen, dass der Testdatensatz die unterschiedlichen im Datensatz enthaltenen Einkommensklassen gut repräsentiert. Da das mittlere Einkommen ein stetiges numerisches Merkmal ist, müssen Sie zuerst ein kategorisches Merkmal für das Einkommen generieren. Betrachten wir das Histogramm des Einkommens etwas genauer (siehe Abbildung 2-8): Die meisten mittleren Einkommen liegen bei 1,6 bis 6 (also 15.000 bis 60.000 USD), aber einige mittlere Einkommen liegen deutlich über 6. Es ist wichtig, dass Ihr Datensatz für jedes Stratum eine genügende Anzahl Datenpunkte enthält, andernfalls liegt ein Bias für die Schätzung der Wichtigkeit des Stratums vor. Das heißt, Sie sollten nicht zu viele Strata haben, und jedes Stratum sollte groß genug sein. Der folgende Code nutzt