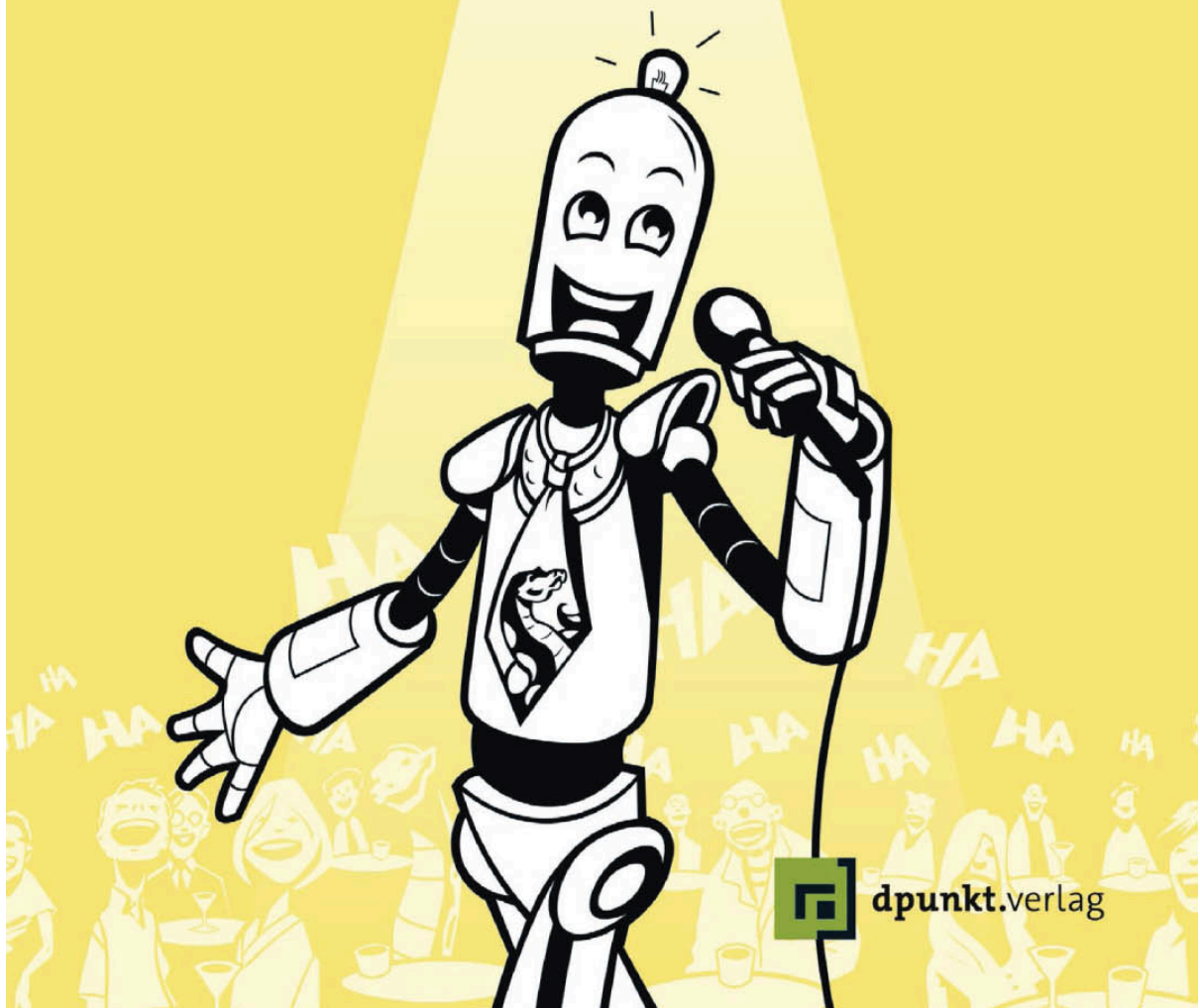


Python One-Liners

Profi-Programmierung durch kurz gefasstes Python

Christian Mayer



Inhalt

Cover

Der Autor

Titel

Impressum

Inhalt

Danksagung

Zur deutschen Ausgabe

Vorwort

Einführung

Ein Beispiel für einen Python-Einzeiler

Ein Hinweis zur Lesbarkeit

An wen richtet sich dieses Buch?

Was werden Sie lernen?

Online-Ressourcen

1 Python-Auffrischkurs

Grundlegende Datenstrukturen

Numerische Datentypen und -strukturen

Boolesche Werte

Strings

Das Schlüsselwort None

Container-Datenstrukturen

Listen

Stacks

Mengen

Dictionaries

Zugehörigkeit

List und d

Kontrollfluss

if, else und elif

Schleifen

Funktionen

Lambdas

Zusammenfassung

2 Python-Tricks

Mit einer List Comprehension Spitzenverdiener finden

Die Grundlagen

Der Code

Wie es funktioniert

Mit einer List Comprehension Wörter mit hohem Informationsgehalt finden

Die Grundlagen

Der Code

Wie es funktioniert

Eine Datei lesen

Die Grundlagen

Der Code

Wie es funktioniert

Lambda- und Map-Funktionen verwenden

Die Grundlagen

Der Code

Wie es funktioniert

Mit Slicing passende Teilstring-Umgebungen extrahieren

Die Grundlagen

Der Code

Wie es funktioniert

List Comprehension und Slicing miteinander kombinieren

Die Grundlagen

Der Code

Wie es funktioniert

Nutzen Sie die Slice-Zuweisung zum Korrigieren von kaputten Listen

Die Grundlagen

Der Code

Wie es funktioniert

Herzgesundheitsdaten mit Listenverkettungen analysieren

Die Grundlagen

Der Code

Wie es funktioniert

Mithilfe von Generatorausdrücken Unternehmen finden, die den Mindestlohn unterschreiten

Die Grundlagen

Der Code

Wie es funktioniert

Datenbanken mit der zip()-Funktion formatieren

Die Grundlagen

Der Code

Wie es funktioniert

Zusammenfassung

3 Data Science

Einfache zweidimensionale Array-Berechnungen

Die Grundlagen

Der Code

Wie es funktioniert

Mit NumPy-Arrays arbeiten: Slicing, Broadcasting und Array-Typen

Die Grundlagen

Der Code

Wie es funktioniert

Bedingte Array-Suche, Filterung und Broadcasting zum Erkennen von Extremwerten

Die Grundlagen

Der Code

Wie es funktioniert

Boolesche Indizierung zum Filtern zweidimensionaler Arrays

Die Grundlagen

Der Code

Wie es funktioniert

Broadcasting, Slice-Zuweisung und Umformen, um jedes i-te Array-Element zu entfernen

Die Grundlagen

Der Code

Wie es funktioniert

Wann Sie die `sort()`-Funktion und wann Sie die `argsort()`-Funktion in NumPy benutzen

Die Grundlagen

Der Code

Wie es funktioniert

Wie Sie mit Lambda-Funktionen und boolescher Indizierung Arrays filtern

Die Grundlagen

Der Code

Wie es funktioniert

Wie Sie erweiterte Array-Filter mit Statistik, Mathematik und Logik herstellen

Die Grundlagen

Der Code

Wie es funktioniert

Einfache Assoziationsanalyse: Menschen, die X gekauft haben, kauften auch Y

Die Grundlagen

Der Code

Wie es funktioniert

Komplexere Assoziationsanalyse zum Finden von Bestseller-Paketen

Die Grundlagen

Der Code

Wie es funktioniert

Zusammenfassung

4 Machine Learning

Die Grundlagen des Supervised Machine Learning

Trainingsphase

Inferenzphase

Lineare Regression

Die Grundlagen

Der Code

Wie es funktioniert

Logistische Regression in einer Zeile

Die Grundlagen

Der Code

Wie es funktioniert

K-Means-Clusteranalyse in einer Zeile

Die Grundlagen

Der Code

Wie es funktioniert

K-Nearest Neighbors in einer Zeile

Die Grundlagen

Der Code

Wie es funktioniert

Analyse neuronaler Netzwerke in einer Zeile

Die Grundlagen

Der Code

Wie es funktioniert

Decision-Tree Learning in einer Zeile

Die Grundlagen

Der Code

Wie es funktioniert

Die minimale Varianz einer Zeile berechnen

Die Grundlagen

Der Code

Wie es funktioniert

Einfache Statistiken in einer Zeile

Die Grundlagen

Der Code

Wie es funktioniert

Klassifikation mit Support-Vector Machines in einer Zeile

Die Grundlagen

Der Code

Wie es funktioniert

Klassifikation mit Random Forests in einer Zeile

Die Grundlagen

Der Code

Wie es funktioniert

Zusammenfassung

5 Reguläre Ausdrücke

Einfache Textmuster in Strings finden

Die Grundlagen

Der Code

Wie es funktioniert

Schreiben Sie Ihren ersten Web-Scraper mit regulären Ausdrücken

Die Grundlagen

Der Code

Wie es funktioniert

Hyperlinks von HTML-Dokumenten analysieren

Die Grundlagen

Der Code

Wie es funktioniert

Dollars aus einem String extrahieren

Die Grundlagen

Der Code

Wie es funktioniert

Unsichere HTTP-URLs finden

Die Grundlagen

Der Code

Wie es funktioniert

Das Zeitformat der Benutzereingabe validieren, Teil 1

Die Grundlagen

Der Code

Wie es funktioniert

Das Zeitformat der Benutzereingabe validieren, Teil 2

Die Grundlagen

Der Code

Wie es funktioniert

Duplikate in String entdecken

Die Grundlagen

Der Code

Wie es funktioniert

Wortwiederholungen erkennen

Die Grundlagen

Der Code

Wie es funktioniert

Regex-Muster in einem mehrzeiligen String modifizieren

Die Grundlagen

Der Code

Wie es funktioniert

Zusammenfassung

6 Algorithmen

Mit Lambda-Funktionen und Sortieren Anagramme finden

Die Grundlagen

Der Code

Wie es funktioniert

Mit Lambda-Funktionen und negativem Slicing Palindrome finden

Die Grundlagen

Der Code

Wie es funktioniert

Permutationen zählen mit rekursiven Fakultätsfunktionen

Die Grundlagen

Der Code

Wie es funktioniert

Die Levenshtein-Distanz finden

Die Grundlagen

Der Code

Wie es funktioniert

Berechnen der Potenzmenge mittels funktionaler Programmierung

Die Grundlagen

Der Code

Wie es funktioniert

Caesar-Verschlüsselung mittels erweiterter Indizierung und List Comprehension

Die Grundlagen

Der Code

Wie es funktioniert

Mit dem Sieb des Eratosthenes Primzahlen finden

Die Grundlagen

Der Code

Wie es funktioniert

Berechnen der Fibonacci-Folge mit der reduce()-Funktion

Die Grundlagen

Der Code

Wie es funktioniert

Ein rekursiver binärer Suchalgorithmus

Die Grundlagen

Der Code

Wie es funktioniert

Ein rekursiver Quicksort-Algorithmus

Die Grundlagen

Der Code

Wie es funktioniert

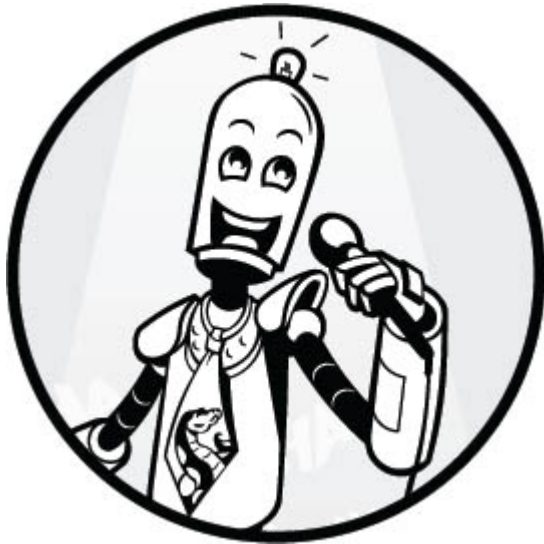
Zusammenfassung

Nachwort

Index

4

Machine Learning



Machine Learning findet sich in fast jedem Bereich der Informatik. In den letzten Jahren besuchte ich Konferenzen auf so unterschiedlichen Gebieten wie verteilte Systeme, Datenbanken und Stream-Processing, und egal wo, dort gab es Machine Learning. Manchmal drehte sich sogar mehr als die Hälfte der präsentierten Forschungsideen um Machine-Learning-Methoden.

Als Informatiker müssen Sie die fundamentalen Machine-Learning-Ideen und -Algorithmen kennen, um Ihre Kenntnisse und Fertigkeiten abzurunden. Dieses Kapitel bietet Ihnen eine Einführung in die wichtigsten Algorithmen und Methoden des Machine Learning und präsentiert 10 praktische Einzeiler zum Anwenden dieser Algorithmen in Ihren eigenen Projekten.

Die Grundlagen des Supervised Machine Learning

Hauptziel des Machine Learning ist es, mithilfe vorhandener Daten akkurate Vorhersagen zu treffen. Nehmen wir einmal an, Sie wollen einen Algorithmus schreiben, der den Wert einer bestimmten Aktie über die nächsten zwei Tage vorhersagt. Um dieses Ziel zu erreichen, müssen Sie ein Machine-Learning-Modell trainieren. Aber was genau ist ein *Modell*?

Aus Sicht des Benutzers von Machine Learning sieht das Machine-Learning-Modell wie eine Blackbox aus (Abbildung 4-1): Sie geben Daten hinein und

bekommen Vorhersagen heraus.



Abb. 4-1 Ein Machine-Learning-Modell, dargestellt als Blackbox

In diesem Modell bezeichnen Sie die Eingabedaten als *Features* und kennzeichnen sie mit der Variablen x . Die Eingabedaten können ein numerischer Wert oder ein mehrdimensionaler Vektor aus numerischen Werten sein. Die Box zaubert dann ein bisschen und verarbeitet Ihre Eingabedaten. Nach einer Weile erhalten Sie die Vorhersage y zurück. Hierbei handelt es sich um die für diese Eingangs-Features vorhergesagte Ausgabe. Bei Regressionsproblemen besteht die Vorhersage aus einem oder mehreren numerischen Werten – genau wie die Eingangs-Features.

Supervised Machine Learning unterteilt sich in zwei getrennte Phasen: die *Trainingsphase* und die *Inferenzphase*.

Trainingsphase

Während der *Trainingsphase* teilen Sie Ihrem Modell die gewünschte Ausgabe y' für eine bestimmte Eingabe x mit. Wenn das Modell die Vorhersage y ausgibt, vergleichen Sie sie mit y' . Sind sie nicht identisch, aktualisieren Sie das Modell, damit es eine Ausgabe generiert, die näher an y' liegt, wie Abbildung 4-2 zeigt. Schauen wir uns ein Beispiel aus der Bilderkennung an. Nehmen wir einmal an, Sie trainieren ein Modell, Namen von Obstsorten vorherzusagen (Ausgaben), wenn ihm bestimmte Bilder präsentiert werden (Eingaben). Ihre spezielle Trainingseingabe ist z. B. das Bild einer Banane, doch Ihr Modell sagt fälschlicherweise einen *Apfel* vorher. Da sich Ihre gewünschte Ausgabe von der Modellvorhersage unterscheidet, ändern Sie das Modell dergestalt, dass es beim nächsten Mal korrekt *Banane* vorhersagt.

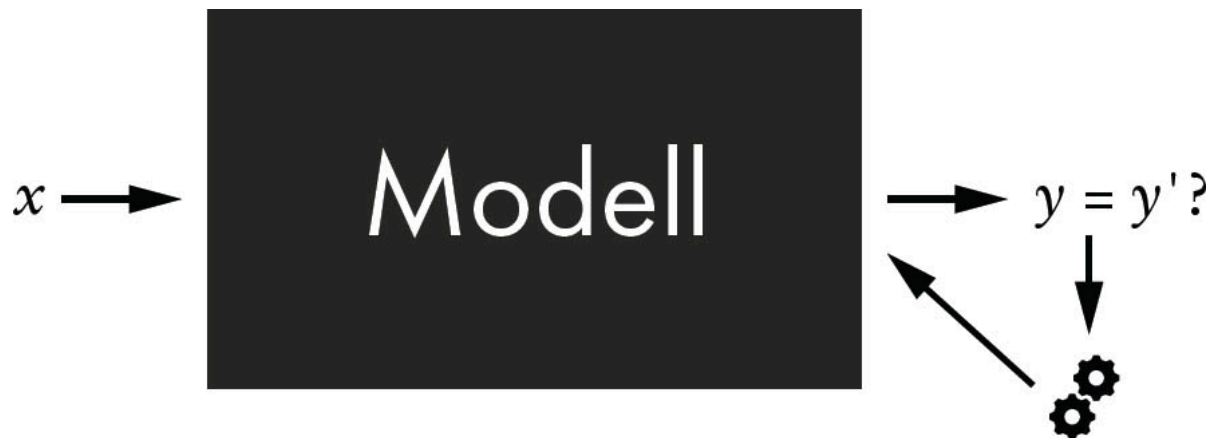


Abb. 4-2 Die Trainingsphase eines Machine-Learning-Modells

Wenn Sie dem Modell weiterhin für viele unterschiedliche Eingaben die gewünschten Ausgaben mitteilen, trainieren Sie es mithilfe Ihrer *Trainingsdaten*. Im Laufe der Zeit lernt das Modell, welche Ausgabe Sie für bestimmte Eingaben haben möchten. Deswegen sind Daten im 21. Jahrhundert so wichtig: Ihr Modell ist nur so gut wie seine Trainingsdaten. Ohne gute Trainingsdaten wird es mit hoher Wahrscheinlichkeit scheitern. Grob gesagt, überwachen die Trainingsdaten den Machine-Learning-Prozess. Deshalb bezeichnen wir es als *Supervised Learning (überwachtes Lernen)*.

Inferenzphase

Während der *Inferenzphase* nutzen Sie das trainierte Modell, um Ausgabewerte für neue Eingangs-Features x vorherzusagen. Beachten Sie, dass das Modell die Macht hat, Ausgaben für Eingaben vorherzusagen, die es niemals zuvor in den Trainingsdaten beobachtet hat. Zum Beispiel kann das Obstvorhersagemodell aus der *Trainingsphase* nun den Namen der Obstsorten (die es in den Trainingsdaten gelernt hat) anhand von Bildern identifizieren, die es noch nie gesehen hat. Mit anderen Worten besitzen geeignete Machine-Learning-Modelle die Fähigkeit, zu *generalisieren*: Sie nutzen ihre Erfahrungen aus den Trainingsdaten, um Ausgaben für neue Eingaben vorherzusagen. Im Prinzip bedeutet dies, dass Modelle, die gut generalisieren (verallgemeinern), akkurate Vorhersagen für neue Eingangsdaten erzeugen. Die generalisierte Vorhersage für noch nie gesehene Eingangsdaten ist eine der Stärken des Machine Learning und gehört zu den Hauptgründen für ihre Popularität in einer Vielzahl von Anwendungen.

Lineare Regression

Lineare Regression ist der Machine-Learning-Algorithmus, den Sie am häufigsten in Machine-Learning-Tutorials für Anfänger finden. Er wird üblicherweise bei *Regressionsproblemen* verwendet, bei denen das Modell fehlende Daten anhand vorhandener Daten vorhersagt. Ein beträchtlicher Vorteil der linearen Regression sowohl für Lehrer als auch für Anwender ist ihre Einfachheit. Das bedeutet jedoch nicht, dass sie keine echten Probleme lösen kann! Es gibt eine Menge praktischer Anwendungsfälle in den unterschiedlichsten Bereichen für die lineare Regression, wie etwa Marktforschung, Astronomie und Biologie. In diesem Abschnitt erfahren Sie alles Nötige, um mit der linearen Regression zu beginnen.

Die Grundlagen

Wie können Sie mit linearer Regression die Aktienpreise an einem bestimmten Tag vorhersagen? Bevor ich diese Frage beantworte, kommen hier einige Definitionen.

Jedes Machine-Learning-Modell besteht aus Modellparametern. *Modellparameter* sind interne Konfigurationsvariablen, die aus den Daten geschätzt werden. Diese Modellparameter bestimmen, wie exakt das Modell die Vorhersage angesichts der Eingangs-Features berechnet. Bei der linearen Regression werden die Modellparameter als *Koeffizienten* bezeichnet. Sie erinnern sich vielleicht an die Formeln für zweidimensionale Geraden aus Ihrer Schulzeit: $f(x) = ax + c$. Die beiden Variablen a und c sind die Koeffizienten in der linearen Gleichung $ax + c$. Sie können beschreiben, wie jede Eingabe x in eine Ausgabe $f(x)$ verwandelt wird, sodass alle Ausgaben zusammen eine Gerade im zweidimensionalen Raum beschreiben. Durch Ändern der Koeffizienten können Sie jede beliebige Gerade im zweidimensionalen Raum beschreiben.

Bei vorgegebenen Eingangs-Features x_1, x_2, \dots, x_k kombiniert das lineare Regressionsmodell die Eingangs-Features mit den Koeffizienten a_1, a_2, \dots, a_k , um mit dieser Formel die vorhergesagte Ausgabe y zu berechnen:

$$y = f(x) = a_0 + a_1 \times x_1 + a_2 \times x_2 + \dots + a_k \times x_k$$

In unserem Aktienbeispiel haben Sie ein einziges Eingangs-Feature, x , den Tag. Sie geben den Tag x ein und hoffen, einen Aktienpreis zu erhalten, die Ausgabe y . Dies vereinfacht das lineare Regressionsmodell auf die Formel einer zweidimensionalen Geraden:

$$y = f(x) = a_0 + a_1x$$

Betrachten wir drei Geraden, für die Sie nur die zwei Modellparameter a_0 und a_1 ändern, in Abbildung 4–3. Die erste Achse beschreibt die Eingabe x . Die zweite Achse beschreibt die Ausgabe y . Die Gerade repräsentiert die (lineare) Beziehung zwischen Eingabe und Ausgabe.

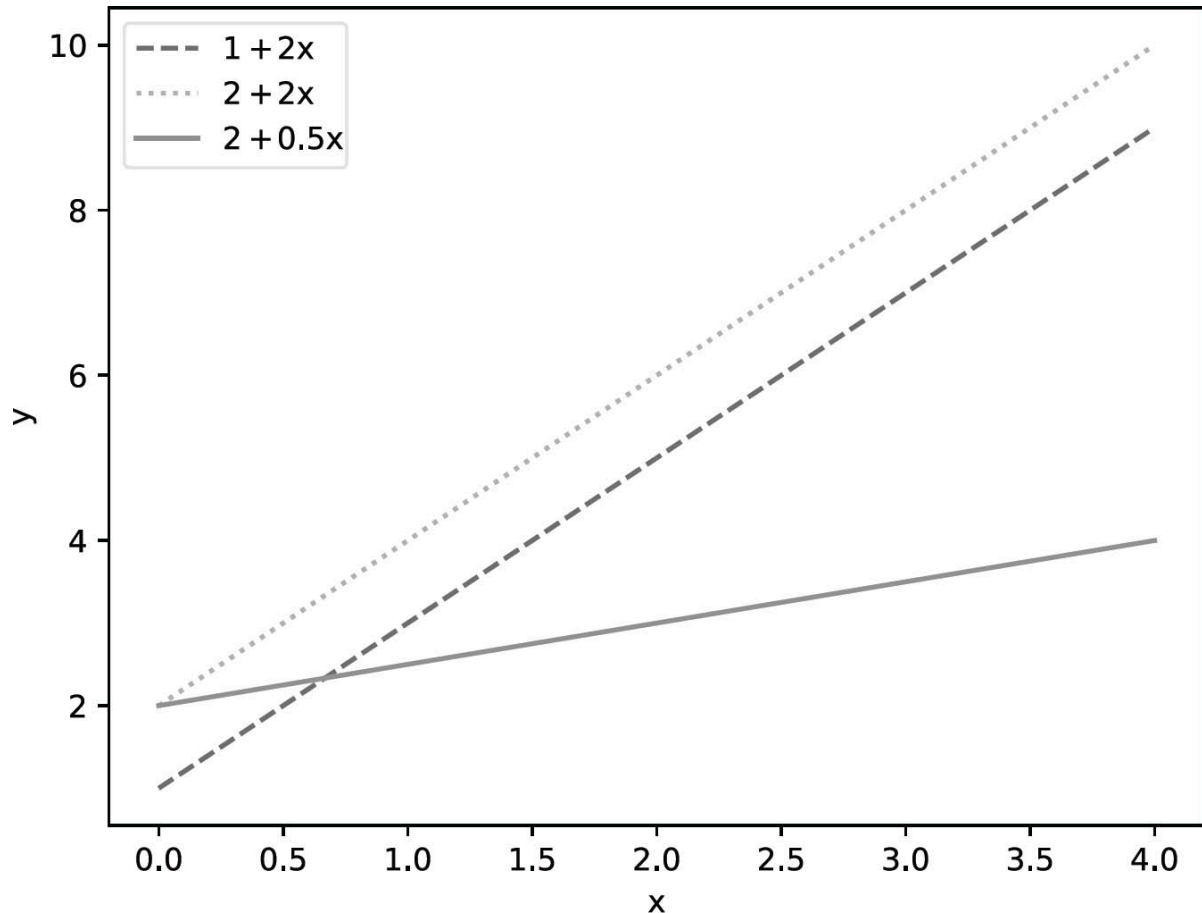


Abb. 4-3 *Drei lineare Regressionsmodelle (Geraden), beschrieben durch unterschiedliche Modellparameter (Koeffizienten). Jede Gerade repräsentiert eine einzigartige Beziehung zwischen den Eingangs- und Ausgangsvariablen.*

Nehmen Sie an, dass in unserem Aktienpreisbeispiel unsere Trainingsdaten die Indizes der drei Tage sind, $[0, 1, 2]$, zugeordnet zu den Aktienpreisen $[155, 156, 157]$. Oder anders gesagt:

- Eingabe $x=0$ sollte Ausgabe $y=155$ auslösen.
- Eingabe $x=1$ sollte Ausgabe $y=156$ auslösen.
- Eingabe $x=2$ sollte Ausgabe $y=157$ auslösen.

Welche Zeile passt am besten zu unseren Trainingsdaten? Ich habe die Trainingsdaten in Abbildung 4–4 gezeichnet.

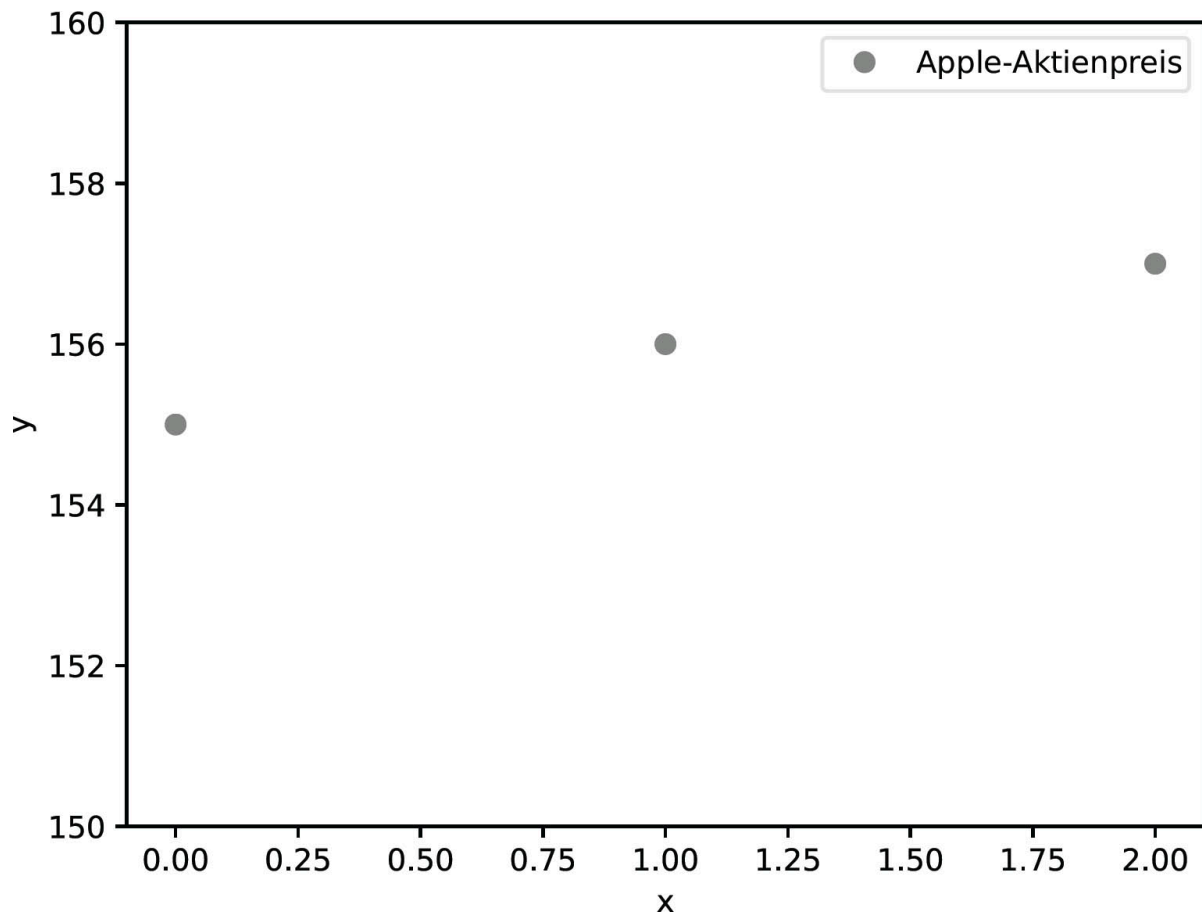


Abb. 4-4 Unsere Trainingsdaten, mit ihrem Index im Feld als x-Koordinate und ihrem Preis als y-Koordinate

Um die Zeile zu finden, die die Daten am besten beschreibt, und um damit ein lineares Regressionsmodell zu erzeugen, müssen wir die Koeffizienten ermitteln. Das ist die Stelle, an der das Machine Learning ins Spiel kommt. Es gibt zwei hauptsächliche Wege, um die Modellparameter für die lineare Regression zu bestimmen. Erstens können Sie analytisch die am besten passende Gerade berechnen, die zwischen diesen Punkten verläuft (die Standardmethode für die lineare Regression). Zweitens können Sie verschiedene Modelle ausprobieren, wobei Sie alle gegen die benannten Sample-Daten testen und sich schließlich für das beste entscheiden. In jedem Fall bestimmen Sie das beste Modell durch einen Prozess, der als *Fehlerminimierung* bezeichnet wird, bei dem das Modell die quadratische Differenz der vorhergesagten Modellwerte und der idealen Ausgabe minimiert (oder die Koeffizienten auswählt, die zu einer minimalen quadratischen Differenz führen) und das Modell mit dem niedrigsten Fehler wählt.

Für unsere Daten erhalten Sie die Koeffizienten $a_0 = 155,0$ und $a_1 = 1,0$. Diese setzen Sie in die Formel für die lineare Regression ein:

$$y = f(x) = a_0 + a_1x = 155.0 + 1.0 \times x$$

und zeichnen sowohl die Gerade als auch die Trainingsdaten in dasselbe Diagramm, wie in Abbildung 4–5 gezeigt wird.

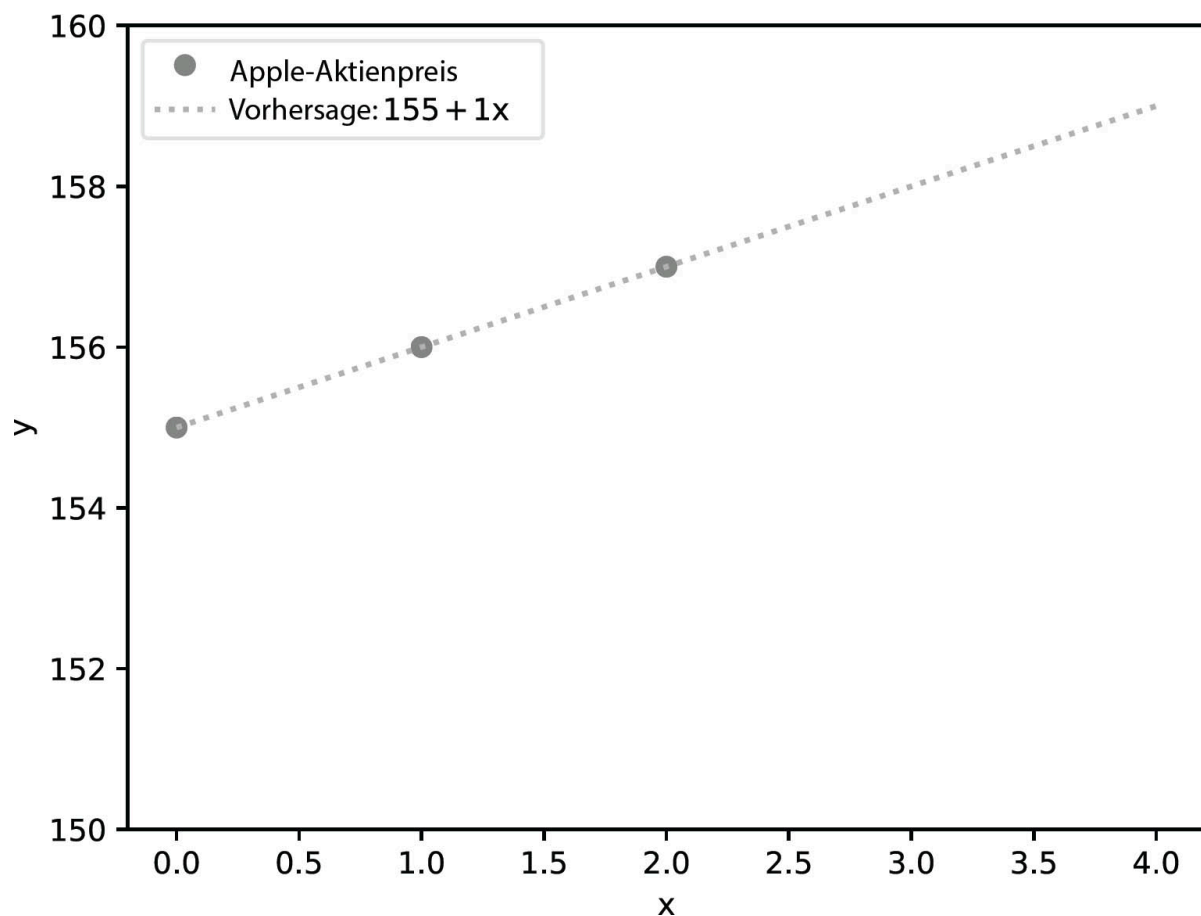


Abb. 4–5 Eine Vorhersage, erstellt mit unserem linearen Regressionsmodell

Passt perfekt! Die quadratische Differenz zwischen der Geraden (Modellvorhersage) und den Trainingsdaten ist null – Sie haben also das Modell gefunden, das den Fehler minimiert. Mithilfe dieses Modells können Sie nun den Aktienpreis für jeden Wert von x vorhersagen. Nehmen wir etwa an, Sie wollen den Aktienpreis an $x = 4$ vorhersagen. Dazu benutzen Sie einfach das Modell, um $f(x) = 155,0 + 1,0 \times 4 = 159,0$ zu berechnen. Der vorhergesagte Aktienpreis an Tag 4 beträgt \$159. Ob diese Vorhersage natürlich akkurat die Verhältnisse in der wirklichen Welt widerspiegelt, ist eine andere Frage.

Jetzt haben Sie eine Vorstellung davon, was passiert. Schauen wir uns an, wie wir das in Code gießen können.

Der Code

Listing 4–1 zeigt Ihnen, wie Sie ein einfaches lineares Regressionsmodell in eine einzige Codezeile umsetzen können (Sie müssen zuerst die scikit-learn-

Bibliothek installieren, indem Sie in Ihrer Shell `pip install sklearn` ausführen).

```
from sklearn.linear_model import LinearRegression

import numpy as np

## Daten (Apple-Aktienpreise)

apple = np.array([155, 156, 157])

n = len(apple)

## Einzeiler

model = LinearRegression().fit(np.arange(n).reshape((n,1)),
apple)

## Ergebnis und Ausgabe

print(model.predict([[3],[4]]))
```

Listing 4-1 *Ein einfaches lineares Regressionsmodell*

Können Sie erraten, wie die Ausgabe dieses Codeschnipsels aussieht?

Wie es funktioniert

Dieser Einzeiler verwendet zwei Python-Bibliotheken: NumPy und scikit-learn. Die erste ist die De-facto-Standardbibliothek für numerische Berechnungen (wie Matrix-Operationen). Die zweite ist die umfangreichste Bibliothek für Machine Learning und enthält Implementierungen Hunderter Machine-Learning-Algorithmen und -Techniken.

Sie werden fragen: »Warum benutzen Sie Bibliotheken in einem Python-Einzeiler? Ist das nicht geschummelt?« Das ist eine gute Frage, und die Antwort lautet Ja. Jedes Python-Programm – mit oder ohne Bibliotheken – nutzt

höherwertige Funktionalitäten, die auf niedrigen Operationen aufsetzen. Es wäre nicht übermäßig sinnvoll, das Rad neu zu erfinden, wenn Sie vorhandene Codegrundlagen nutzen können. Ehrgeizige Programmierer verspüren oft den Drang, alles selbst implementieren zu müssen – meist auf Kosten ihrer Produktivität. In diesem Buch werden wir uns das breite Spektrum an leistungsfähiger Funktionalität zunutze machen, das von einigen der weltbesten Python-Programmierer und -Vorreiter implementiert wurde. In all diesen Bibliotheken steckt jahrelange Entwicklungs- und Optimierungsarbeit talentierter Programmierer.

Gehen wir Listing 4-1 Schritt für Schritt durch. Zuerst erzeugen wir einen einfachen Datensatz aus drei Werten und speichern dessen Länge in einer eigenen Variablen `n`, um den Code präziser zu machen. Unsere Daten sind drei Apple-Aktienpreise für drei aufeinanderfolgende Tage. Die Variable `apple` enthält den Datensatz als eindimensionales NumPy-Array.

Als Zweites erstellen wir das Modell, indem wir `LinearRegression()` aufrufen. Doch welches sind die Modellparameter? Um sie zu finden, rufen wir die Funktion `fit()` zum Trainieren des Modells auf. Die `fit()`-Funktion nimmt zwei Argumente entgegen: die Eingangs-Features der Trainingsdaten und die idealen Ausgaben für diese Eingaben. Unsere idealen Ausgaben sind die wirklichen Aktienpreise der Apple-Aktie. Für die Eingangs-Features dagegen verlangt `fit()` ein Array mit dem folgenden Format:

```
[<Trainingsdaten_1>,  
  
<Trainingsdaten_2>,  
  
– schnipp –  
  
<Trainingsdaten_n>]
```

wobei jeder Trainingsdatenwert eine Sequenz aus Feature-Werten ist:

```
<Trainingsdaten> = [Feature_1, Feature_2, ..., Feature_k]
```

In unserem Fall besteht die Eingabe nur aus einem einzigen Feature x (dem aktuellen Tag). Auch die Vorhersage besteht nur aus einem einzigen Wert y (dem

aktuellen Aktienpreis). Um das Eingangs-Array in die korrekte Form zu bringen, müssen Sie es in diese seltsam aussehende Matrixform umformen:

```
[[0],  
  
 [1],  
  
 [2]]
```

Eine Matrix mit nur einer einzigen Spalte wird als *Spaltenvektor* bezeichnet. Sie verwenden `np.arange()`, um die Sequenz der zunehmenden x -Werte zu erzeugen; anschließend konvertieren Sie mit `reshape((n, 1))` das eindimensionale NumPy-Array in ein zweidimensionales Array mit einer Spalte und n Zeilen (siehe Kapitel 3). Beachten Sie, dass scikit-learn als Ausgabe ein eindimensionales Array erlaubt (ansonsten müssten Sie das `apple`-Daten-Array ebenfalls umformen).

Sobald es die Trainingsdaten und die idealen Ausgaben hat, führt `fit()` eine Fehlerminimierung durch: Es sucht die Modellparameter (d. h., die *Zeile*), für die der Unterschied, also die Differenz, zwischen den vorhergesagten Modellwerten und den gewünschten Ausgaben minimal ist.

Wenn `fit()` mit seinem Modell zufrieden ist, gibt es ein Modell zurück, mit dem Sie dann zwei neue Aktienwerte vorhersagen können. Dazu verwenden Sie die Funktion `predict()`. Die `predict()`-Funktion stellt die gleichen Anforderungen an die Eingabe wie `fit()`. Um sie zu erfüllen, übergeben Sie eine einspaltige Matrix mit unseren zwei neuen Werten, für die Sie Vorhersagen wünschen:

```
print(model.predict([[3],[4]]))
```

Da unsere Fehlerminimierung null ergeben hat, sollten Sie die perfekt linearen Ausgaben 158 und 159 erhalten. Das passt gut zu der Geraden, die in Abbildung 4–5 gezeichnet wurde. Oft jedoch ist es nicht möglich, ein solch perfekt passendes lineares Modell mit nur einer einzigen Geraden zu finden. Falls Sie z. B. die gezeigte Funktion bei den Aktienpreisen `[157, 156, 159]` ausführen und zeichnen, erhalten Sie die Gerade aus Abbildung 4–6.

Wie beschrieben, findet die `fit()`-Funktion in diesem Fall die Gerade, die den quadratischen Fehler zwischen den Trainingsdaten und den Vorhersagen

minimiert.

Fassen wir das noch einmal zusammen. Lineare Regression ist eine Machine-Learning-Technik, bei der Ihr Modell Koeffizienten als Modellparameter lernt. Das resultierende lineare Modell (z. B. eine Gerade im zweidimensionalen Raum) liefert Ihnen direkt Vorhersagen für neue Eingangsdaten. Das Vorhersagen numerischer Werte bei vorgegebenen numerischen Eingangswerten gehört in die Klasse der Regressionsprobleme. Im nächsten Abschnitt lernen Sie ein weiteres wichtiges Gebiet des Machine Learning kennen, die sogenannte Klassifikation.

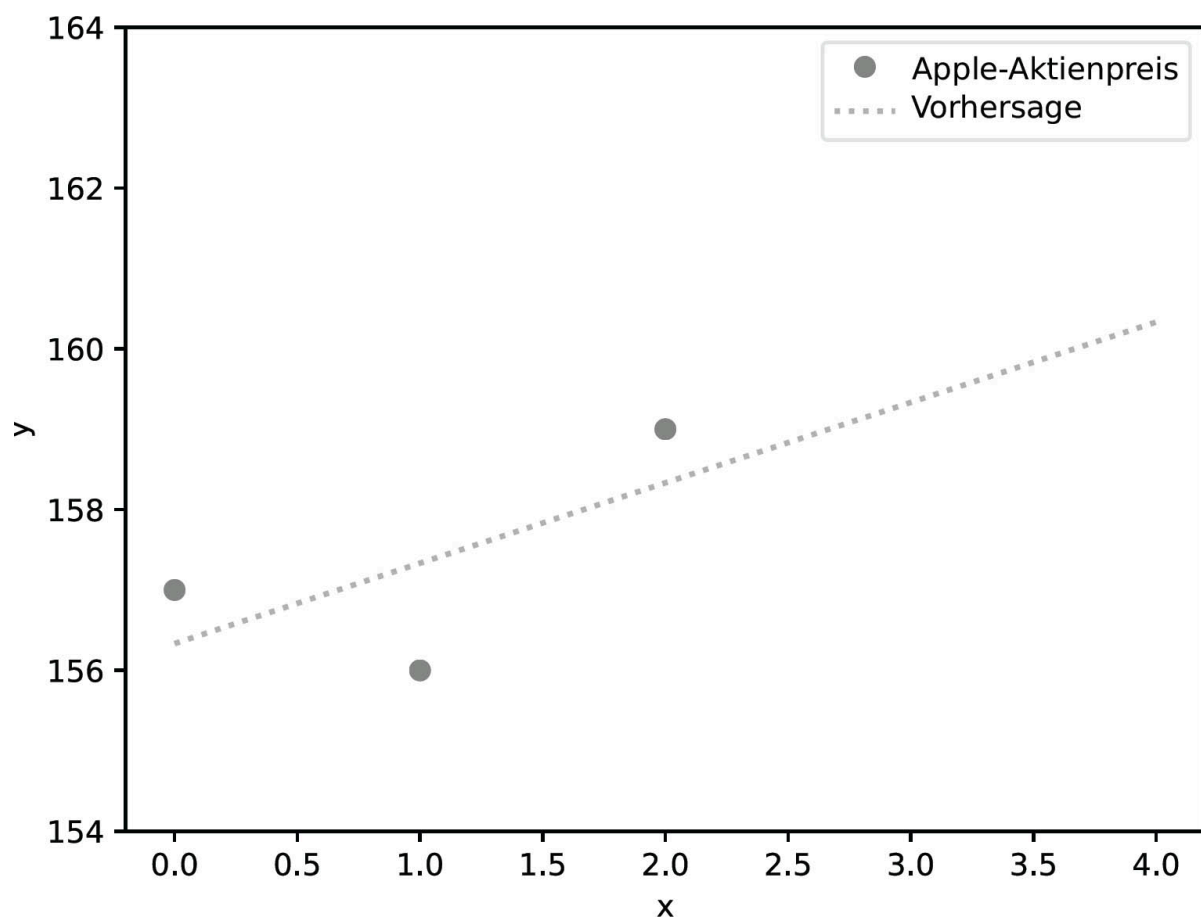


Abb. 4-6 Ein lineares Regressionsmodell mit unvollkommener Anpassung

Logistische Regression in einer Zeile

Die logistische Regression wird gemeinhin für *Klassifikationsprobleme* verwendet, bei denen Sie vorhersagen, ob eine Probe zu einer bestimmten Kategorie (oder Klasse) gehört. Dies unterscheidet sich von den Regressionsproblemen, bei denen Sie eine Probe erhalten und einen numerischen Wert vorhersagen, der in einen fortlaufenden Bereich fällt. Ein Klassifikationsproblem wäre z. B. die Unterteilung von Twitter-Anwendern in Männer und Frauen anhand bestimmter Eingangs-Features wie etwa *Posting-*

Häufigkeit oder *Anzahl von Antworten auf Tweets*. Das logistische Regressionsmodell gehört zu den grundlegendsten Machine-Learning-Modellen. Viele der in diesem Abschnitt vorgestellten Konzepte bilden die Grundlage für weitergehende Machine-Learning-Techniken.

Die Grundlagen

Um die logistische Regression vorzustellen, wollen wir noch einmal kurz schauen, wie die lineare Regression funktioniert: Mithilfe der vorgegebenen Trainingsdaten berechnen Sie eine Gerade, die zu diesen Trainingsdaten passt und das Ergebnis für die Eingabe x vorhersagt. Im Allgemeinen eignet sich die lineare Regression hervorragend für das Vorhersagen einer *kontinuierlichen* Ausgabe, die unendlich viele verschiedene Werte haben kann. Der vorhergesagte Aktienpreis im gezeigten Beispiel könnte etwa eine beliebige Anzahl positiver Werte haben.

Doch was machen wir, wenn die Ausgabe nicht kontinuierlich ist, sondern *kategorisch*, also zu einer begrenzten Anzahl von Gruppen oder Kategorien gehört? Nehmen Sie z. B. an, Sie möchten anhand der Anzahl der gerauchten Zigaretten eines Patienten die Wahrscheinlichkeit von Lungenkrebs vorhersagen. Jeder Patient kann entweder Lungenkrebs haben oder nicht. Anders als bei den Aktienpreisen gibt es hier nur diese zwei möglichen Ergebnisse. Das Vorhersagen der Wahrscheinlichkeit von kategorischen Ergebnissen ist die wichtigste Motivation für die logistische Regression.

Die Sigmoid-Funktion

Während die lineare Regression eine Gerade an die Trainingsdaten anpasst, passt die logistische Regression eine S-förmige Kurve an, die sogenannte *Sigmoid-Funktion*. Die S-Kurve hilft Ihnen, binäre Entscheidungen zu treffen (z. B. Ja/Nein). Für die meisten Eingangswerte liefert die Sigmoid-Funktion einen Wert zurück, der entweder sehr dicht an 0 liegt (eine Kategorie) oder sehr dicht an 1 (die andere Kategorie). Es ist relativ unwahrscheinlich, dass Ihr Eingangswert eine uneindeutige Ausgabe generiert. Es ist möglich, für einen Eingangswert eine Wahrscheinlichkeit von 0,5 zu erhalten – allerdings soll die Form der Kurve dies im praktischen Fall nicht zulassen (für die meisten möglichen Werte auf der horizontalen Achse ist der Wahrscheinlichkeitswert entweder sehr dicht bei 0 oder sehr dicht bei 1). Abbildung 4–7 zeigt die Kurve einer logistischen Regression für das Lungenkrebs-Szenario.

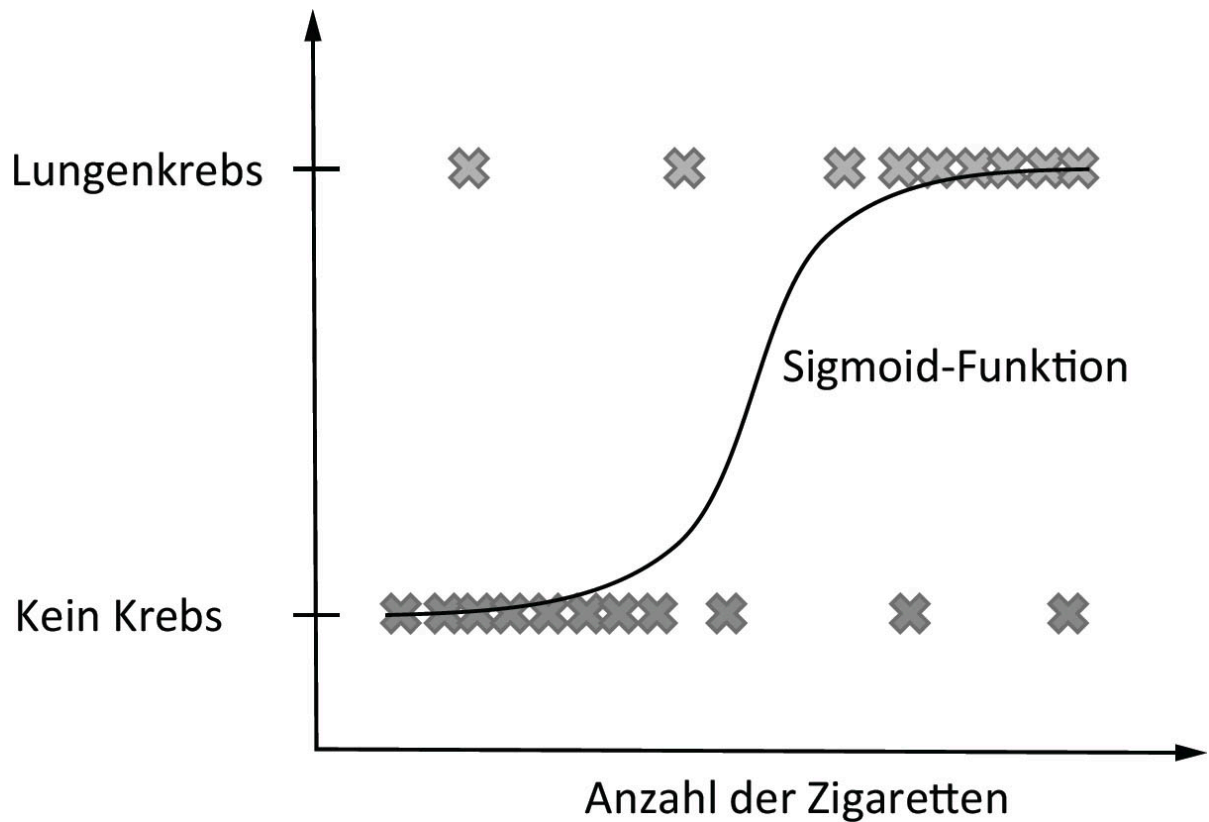


Abb. 4-7 Eine logistische Regressionskurve, die Krebs anhand der Zigarettenzahl vorhersagt



Hinweis

Sie können die logistische Regression für eine *multinomiale Klassifizierung* einsetzen, um die Daten in mehr als zwei Klassen zu unterteilen. Dazu benutzen Sie die Verallgemeinerung der Sigmoid-Funktion, die als *Softmax-Funktion* bezeichnet wird und ein Tupel aus Wahrscheinlichkeiten zurückliefert, nämlich eines für jede Klasse. Die Sigmoid-Funktion verwandelt das/die Eingangs-Feature/s in einen einzigen Wahrscheinlichkeitswert. Aus Gründen der Klarheit und Lesbarkeit werde ich mich in diesem Abschnitt jedoch auf die *binomiale Klassifizierung* und die Sigmoid-Funktion konzentrieren.

Die Sigmoid-Funktion in Abbildung 4-7 nähert anhand der Anzahl der Zigaretten, die ein Patient raucht, die Wahrscheinlichkeit an, dass er Lungenkrebs hat. Diese Wahrscheinlichkeit hilft Ihnen, eine robuste Entscheidung zu dem Thema zu treffen, wenn die einzige Information, die Sie haben, die Anzahl der Zigaretten ist, die der Patient raucht: Hat der Patient Lungenkrebs?

Schauen Sie sich die Vorhersagen in Abbildung 4-8 an, die Ihnen zwei neue Patienten zeigen (in Hellgrau im unteren Teil des Diagramms). Sie wissen nichts über sie bis auf die Anzahl der Zigaretten, die sie rauchen. Sie haben Ihr logistisches Regressionsmodell (die Sigmoid-Funktion) trainiert, die für jeden

neuen Eingangswert x einen Wahrscheinlichkeitswert zurückliefert. Liegt die Wahrscheinlichkeit, die von der Sigmoid-Funktion angegeben ist, über 50 Prozent, dann sagt das Modell *Lungenkrebs positiv* vorher, ansonsten *Lungenkrebs negativ*.

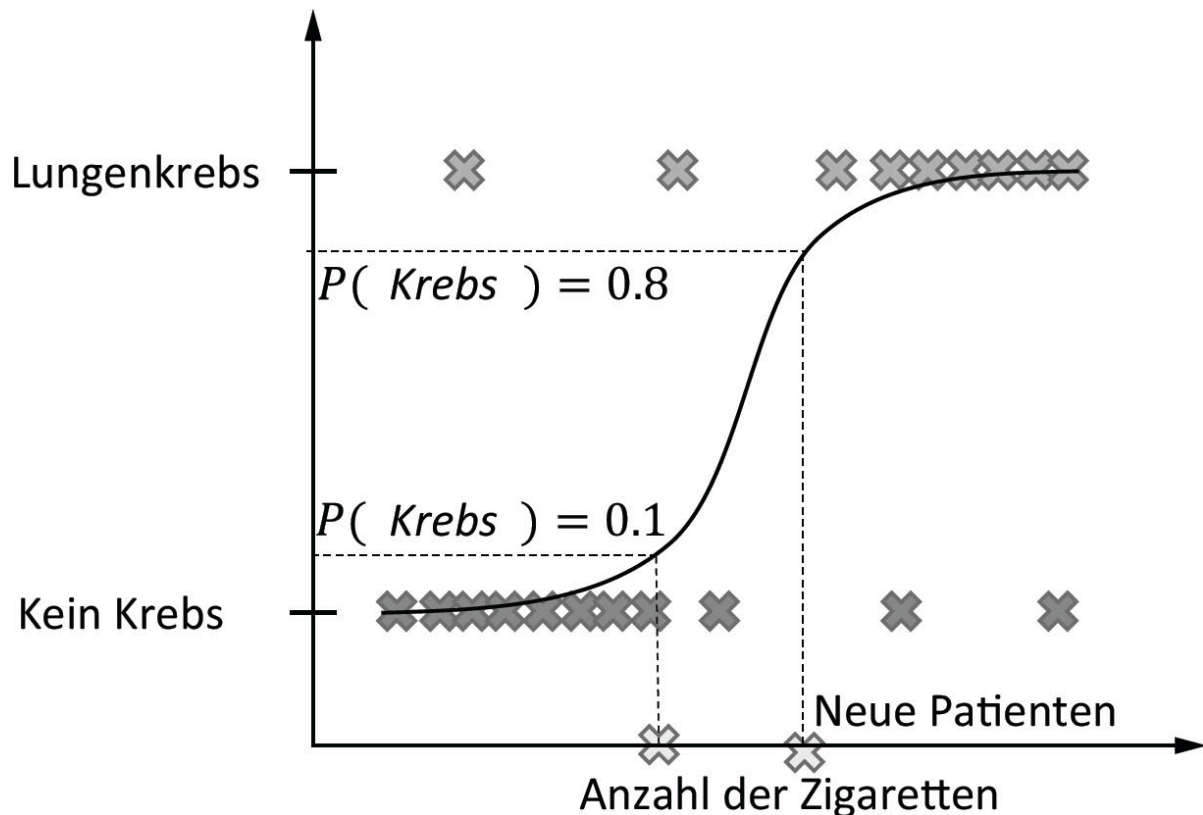


Abb. 4-8 Logistische Regression zum Schätzen der Wahrscheinlichkeiten eines Ergebnisses

Das Maximum-Likelihood-Modell finden

Die wichtigste Frage für die logistische Regression ist, wie man die korrekte Sigmoid-Funktion wählt, also die Funktion, die am besten zu den Trainingsdaten passt. Die Antwort liegt in der *Likelihood* jedes Modells: der Wahrscheinlichkeit, dass das Modell die beobachteten Trainingsdaten generieren würde. Sie möchten das Modell mit der maximalen Wahrscheinlichkeit auswählen. Ihr Gefühl sagt Ihnen, dass dieses Modell am besten den realen Prozess annähert, der die Trainingsdaten generiert hat.

Um die Wahrscheinlichkeit eines bestimmten Modells für eine gegebene Menge an Trainingsdaten zu bestimmen, berechnen Sie die Wahrscheinlichkeit für jeden einzelnen Trainingsdatenpunkt und multiplizieren diese dann miteinander, um die Wahrscheinlichkeit der gesamten Menge an Trainingsdaten zu erhalten. Wie erfolgt nun die Berechnung der Wahrscheinlichkeit eines einzelnen Trainingsdatenpunkts? Wenden Sie einfach die Sigmoid-Funktion dieses Modells auf den Trainingsdatenpunkt an; damit erhalten Sie die

Wahrscheinlichkeit des Datenpunkts unter diesem Modell. Um das *Maximum-Likelihood-Modell* für alle Datenpunkte auszuwählen, wiederholen Sie diese Wahrscheinlichkeitsberechnung für unterschiedliche Sigmoid-Funktionen (indem Sie die Sigmoid-Funktion ein bisschen verschieben), wie in Abbildung 4-9 gezeigt wird.

Im vorangegangenen Absatz habe ich beschrieben, wie man die Maximum-Likelihood-Sigmoid-Funktion (Modell) bestimmt. Diese Sigmoid-Funktion ist am besten an die Daten angepasst – sodass Sie sie benutzen können, um neue Datenpunkte vorherzusagen.

Nachdem wir die Theorie behandelt haben, können wir uns nun anschauen, wie Sie die logistische Regression als Python-Einzeiler implementieren.

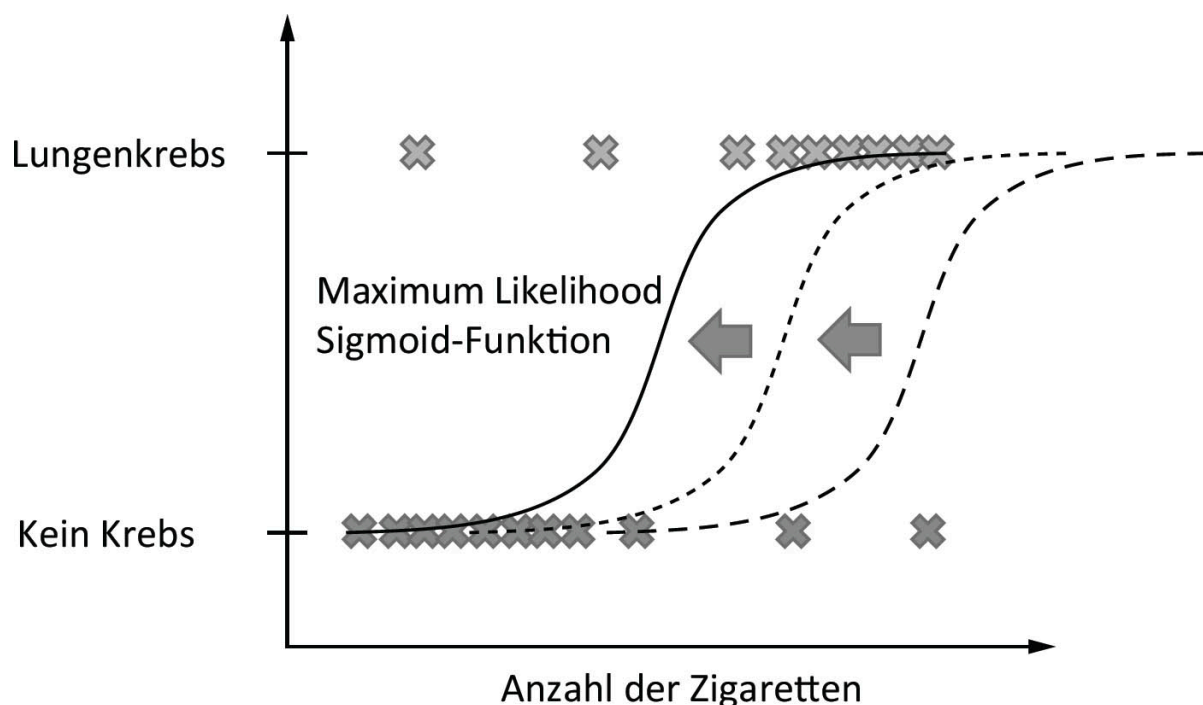


Abb. 4-9 Testen unterschiedlicher Sigmoid-Funktionen zum Bestimmen der Maximum Likelihood

Der Code

Sie haben gesehen, wie sich die logistische Regression auf ein Gesundheitsbeispiel anwenden lässt (das Korrelieren des Zigarettenkonsums mit der Krebswahrscheinlichkeit). Diese »virtueller Doc«-Anwendung wäre eine tolle Idee für eine Smartphone-App, oder? Programmieren wir unseren ersten virtuellen Doc mit logistischer Regression, wie Listing 4-2 zeigt – in einer einzigen Zeile Python-Code!

```

from sklearn.linear_model import LogisticRegression

import numpy as np

## Daten (Anzahl Zigaretten, Krebs)

X = np.array([[0, "Nein"],
              [10, "Nein"],
              [60, "Ja"],
              [90, "Ja"]])

## Einzeiler

model = LogisticRegression().fit(X[:,0].reshape(n,1),
X[:,1])

## Ergebnis und Ausgabe

print(model.predict([[2],[12],[13],[40],[90]]))

```

Listing 4-2 Ein logistisches Regressionsmodell

Raten Sie: Welche Ausgabe hat dieser Codeschnipsel?

Wie es funktioniert

Die Trainingsdaten X bestehen aus vier Patientenakten (die Zeilen) mit zwei Spalten. Die erste Spalte enthält die Anzahl der Zigaretten, die die Patienten rauchen (*Eingangs-Feature*), während in der zweiten Spalte die *Klassen-Label* stehen, die sagen, ob sie an Lungenkrebs leiden.

Sie erzeugen das Modell, indem Sie den Konstruktor `LogisticRegression()` aufrufen. Sie rufen die `fit()`-Funktion an diesem Modell auf; `fit()` nimmt zwei Argumente entgegen: die Eingabe

(Zigarettenkonsum) und die Ausgangsklassen-Label (Krebs). Die `fit()`-Funktion erwartet ein zweidimensionales Eingangs-Array-Format mit einer Zeile pro Trainingsdatenprobe und einer Spalte pro Feature für diese Trainingsdatenprobe. In diesem Fall haben Sie nur einen einzigen Feature-Wert, weshalb Sie die eindimensionale Eingabe mit der `reshape()`-Operation in ein zweidimensionales NumPy-Array verwandeln. Das erste Argument für `reshape()` gibt die Anzahl der Zeilen an, das zweite die Anzahl der Spalten. Sie interessieren sich nur für die Anzahl der Spalten, die hier 1 beträgt. Sie übergeben `-1` als Anzahl der gewünschten Zeilen, was NumPy signalisiert, dass es die Anzahl der Zeilen automatisch ermitteln soll.

Die Eingangstrainingsdaten sehen nach dem Umformen folgendermaßen aus (im Prinzip entfernen Sie einfach nur die Klassen-Label und lassen die zweidimensionale Array-Form unverändert):

```
[[0],  
  
 [10],  
  
 [60],  
  
 [90]]
```

Als Nächstes sagen Sie anhand der Anzahl der gerauchten Zigaretten vorher, ob ein Patient Lungenkrebs hat: Ihre Eingabe ist 2, 12, 13, 40, 90 Zigaretten. Daher erhalten Sie folgende Ausgabe:

```
# ['Nein' 'Nein' 'Ja' 'Ja' 'Ja']
```

Das Modell sagt vorher, dass die ersten beiden Patienten Lungenkrebs-negativ sind, während die letzten drei Lungenkrebs-positiv sind.

Schauen wir uns die Wahrscheinlichkeiten der Sigmoid-Funktion genauer an, die zu dieser Vorhersage geführt haben! Führen Sie nach Listing 4-2 einfach einmal den folgenden Code aus:

```
for i in range(20):
```

```
print("x=" + str(i) + " -> " +  
      str(model.predict_proba([[i]])))
```

Die Funktion `predict_proba()` nimmt als Eingabe die Anzahl der Zigaretten entgegen und liefert ein Array zurück, das die Wahrscheinlichkeit für Lungenkrebs-negativ (bei Index 0) und Lungenkrebs-positiv (Index 1) enthält. Wenn Sie diesen Code ausführen, sollten Sie diese Ausgabe erhalten:

```
x=0 -> [[0.67240789 0.32759211]]
```

```
x=1 -> [[0.65961501 0.34038499]]
```

```
x=2 -> [[0.64658514 0.35341486]]
```

```
x=3 -> [[0.63333374 0.36666626]]
```

```
x=4 -> [[0.61987758 0.38012242]]
```

```
x=5 -> [[0.60623463 0.39376537]]
```

```
x=6 -> [[0.59242397 0.40757603]]
```

```
x=7 -> [[0.57846573 0.42153427]]
```

```
x=8 -> [[0.56438097 0.43561903]]
```

```
x=9 -> [[0.55019154 0.44980846]]
```

```
x=10 -> [[0.53591997 0.46408003]]
```

```
x=11 -> [[0.52158933 0.47841067]]
```

```
x=12 -> [[0.50722306 0.49277694]]
```

```
x=13 -> [[0.49284485 0.50715515]]
```

x=14 → [[0.47847846 0.52152154]]

x=15 → [[0.46414759 0.53585241]]

x=16 → [[0.44987569 0.55012431]]

x=17 → [[0.43568582 0.56431418]]

x=18 → [[0.42160051 0.57839949]]

x=19 → [[0.40764163 0.59235837]]

Falls die Wahrscheinlichkeit, dass Lungenkrebs-negativ herauskommt, höher ist als die Wahrscheinlichkeit für Lungenkrebs-positiv, dann ist das vorhergesagte Ergebnis *Lungenkrebs-negativ*. Das passiert zum letzten Mal für x=12. Hat der Patient mehr als 12 Zigaretten geraucht, klassifiziert der Algorithmus ihn als *Lungenkrebs-positiv*.

Sie haben also jetzt gelernt, wie Sie Probleme mit logistischer Regression klassifizieren können. Dabei hilft Ihnen die scikit-learn-Bibliothek. Die Idee der logistischen Regression ist es, eine S-Kurve (die Sigmoid-Funktion) an die Daten anzupassen. Diese Funktion weist jedem neuen Datenpunkt und jeder möglichen Klasse einen numerischen Wert zwischen 0 und 1 zu. Der numerische Wert modelliert die Wahrscheinlichkeit, dass dieser Datenpunkt zu der gegebenen Klasse gehört. In der Praxis haben Sie allerdings oft Trainingsdaten, aber kein Klassen-Label das den Trainingsdaten zugewiesen ist. So haben Sie z. B. Kundendaten (wie Alter und Einkommen), kennen aber für die einzelnen Datenpunkte keine Klassen-Label. Um dennoch sinnvolle Erkenntnisse aus dieser Art von Daten zu ziehen, lernen Sie gleich eine weitere Kategorie des Machine Learning kennen: das *Unsupervised Learning (unüberwachtes Lernen)*. Genauer gesagt, lernen Sie, wie Sie ähnliche Cluster aus Datenpunkten finden, eine wichtige Teilmenge des Unsupervised Learning.

K-Means-Clusteranalyse in einer Zeile

Wenn es einen Clusteranalyse-Algorithmus gibt, den Sie kennen müssen – ob Sie nun Informatiker, Data Scientist oder Machine-Learning-Experte sind –, ist es der *k-Means-Algorithmus*. In diesem Abschnitt lernen Sie die allgemeine Idee

dahinter kennen und erfahren, wann und wie Sie ihn in einer einzigen Zeile aus Python-Code einsetzen können.

Die Grundlagen

In den vorangegangenen Abschnitten ging es um das *Supervised Learning*, bei dem wir *benannte* Trainingsdaten haben. Mit anderen Worten, Sie kennen den Ausgabewert jedes Eingabewerts in den Trainingsdaten. In der Praxis ist das aber nicht immer der Fall. Oft sehen Sie sich *unbenannten* Daten gegenüber – vor allem in vielen Datenanalyseanwendungen –, bei denen nicht klar ist, was »die optimale Ausgabe« bedeutet. In diesen Situationen ist eine Vorhersage unmöglich (weil es überhaupt keine Ausgabe gibt), aber Sie können trotzdem nützliches Wissen aus diesen unbenannten Datensätzen herausfiltern (z. B. können Sie Cluster ähnlicher unbenannter Daten finden). Modelle, die unbenannte Daten nutzen, fallen in die Kategorie des *Unsupervised Learning*.

Nehmen Sie etwa an, dass Sie in einem Start-up arbeiten, das unterschiedliche Zielmärkte mit verschiedenen Einkommensstufen und Altersgruppen bedient. Ihr Chef sagt Ihnen, dass Sie eine bestimmte Anzahl von Zielpersonengruppen oder -rollen finden sollen, die am besten zu Ihren Zielmärkten passen. Sie können nun mithilfe von Methoden der Clusteranalyse die *durchschnittlichen Kundenrollen* identifizieren, die Ihr Unternehmen bedient. Abbildung 4–10 zeigt ein Beispiel.

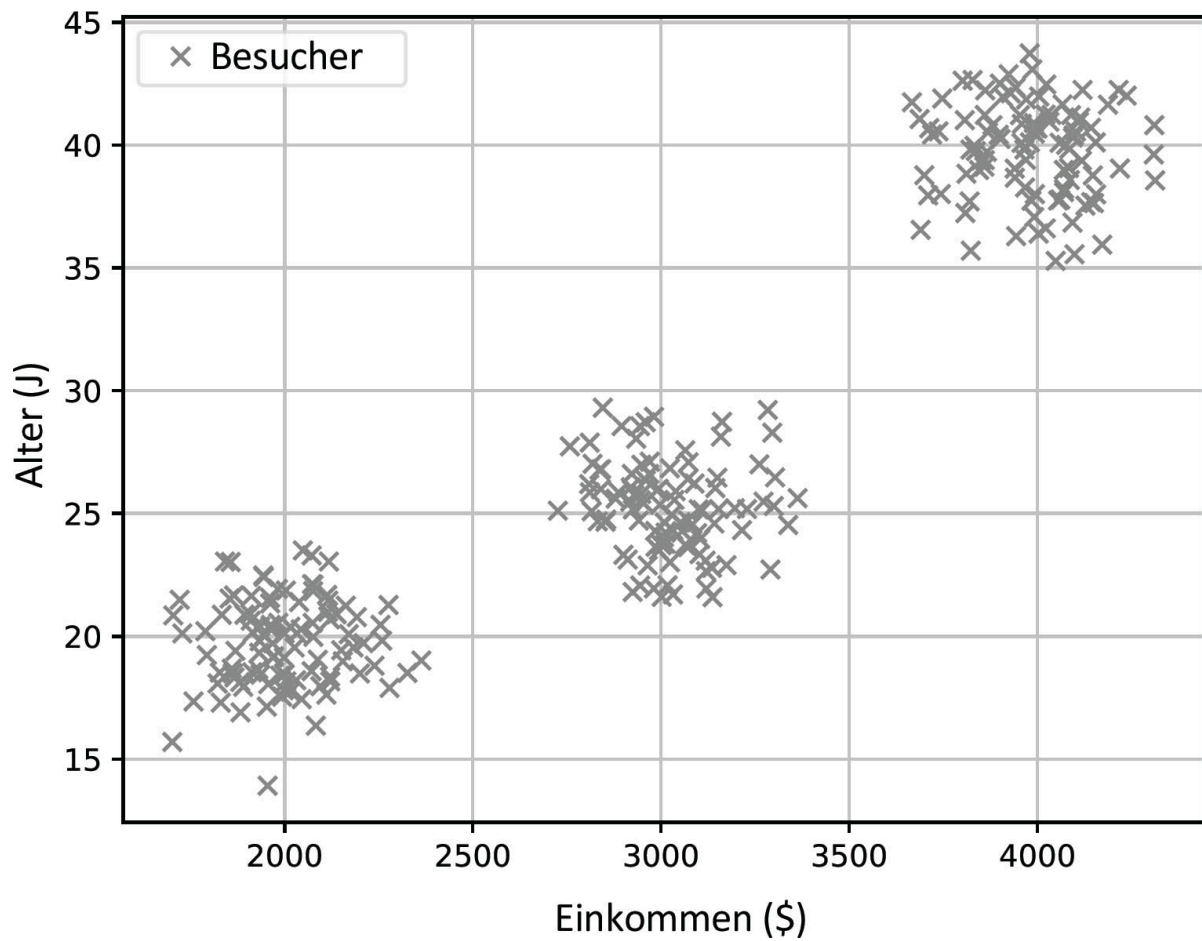


Abb. 4-10 Beobachtete Kundendaten im zweidimensionalen Raum

Hier lassen sich leicht drei Typen von Rollen mit unterschiedlichen Einkommen und Altersangaben erkennen. Doch wie ermitteln Sie diese algorithmisch? Dies ist die Domäne der Clusteranalyse-Algorithmen wie des bekannten k-Means-Algorithmus. Bei vorgegebenen Datensätzen und einem Integer k sucht der k-Means-Algorithmus k Daten-Cluster derart, dass der Unterschied zwischen dem Zentrum des Clusters (dem sogenannten *Schwerpunkt*) und den Daten in dem Cluster minimal ist. Mit anderen Worten, Sie können die unterschiedlichen Rollen finden, indem Sie den k-Means-Algorithmus auf Ihre Datensätze anwenden, wie in Abbildung 4-11 gezeigt wird.

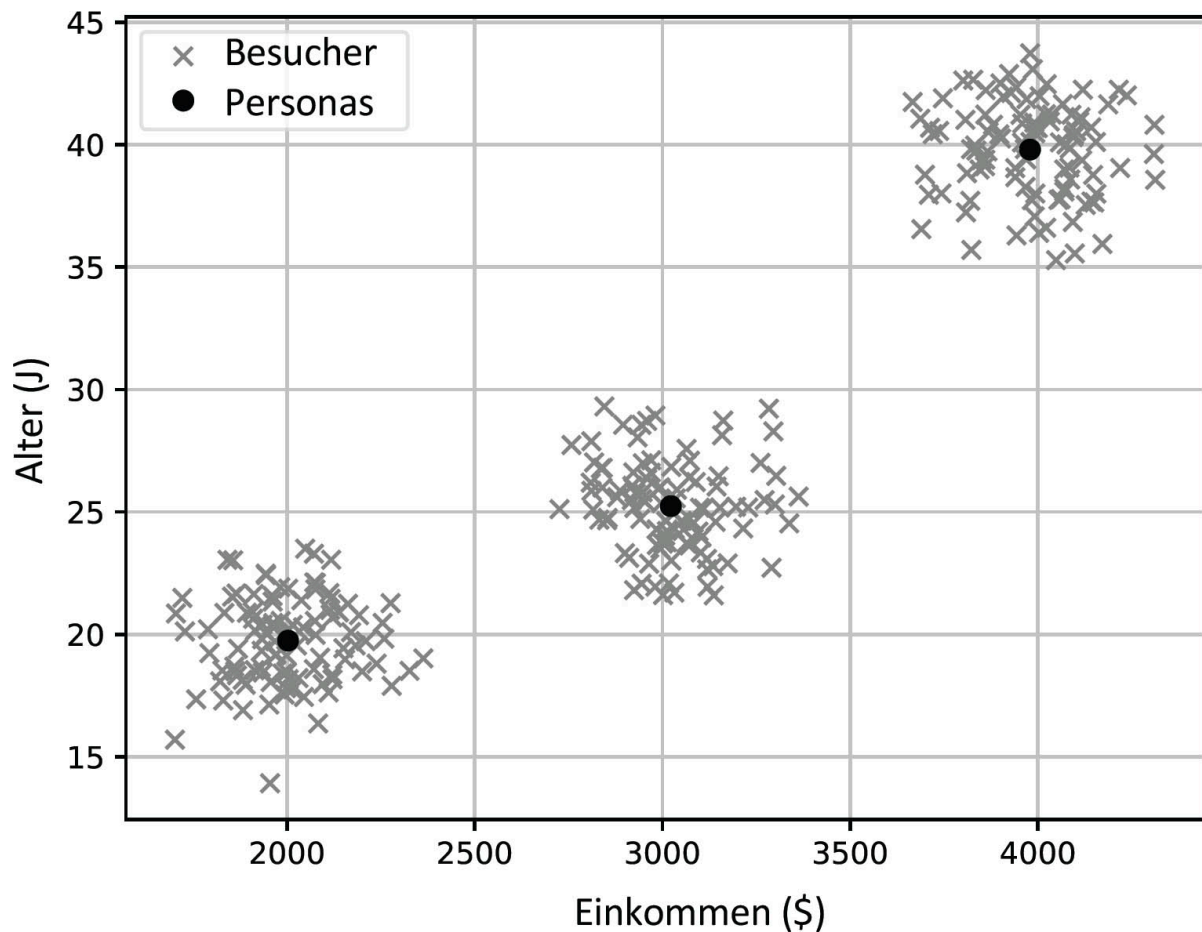


Abb. 4-11 Kundendaten mit Kundenrollen (Cluster-Schwerpunkt) im zweidimensionalen Raum

Die Cluster-Zentren (schwarze Punkte) entsprechen den zusammengeballten Kundendaten. Jedes Cluster-Zentrum kann als eine Kundenrolle betrachtet werden. Das bedeutet, dass Sie drei idealisierte Rollen haben: ein 20-Jähriger, der \$2000 verdient, ein 25-Jähriger, der \$3000 verdient, und ein 40-Jähriger, der \$4000 verdient. Und das Tolle ist, dass der k-Means-Algorithmus diese Cluster-Zentren sogar in hochdimensionalen Räumen findet (in denen Menschen Probleme hätten, die Rollen visuell zu ermitteln).

Der k-Means-Algorithmus verlangt »die Anzahl der Cluster-Zentren k « als Eingabe. In diesem Fall schauen Sie sich die Daten an und definieren »auf magische Weise« $k = 3$. Komplexere Algorithmen können die Anzahl der Cluster-Zentren automatisch feststellen (lesen Sie z. B. den Artikel »Learning the k in K-Means« von Greg Hamerly und Charles Elkan aus dem Jahre 2004).

Wie funktioniert nun also der k-Means-Algorithmus? Im Prinzip führt er folgende Prozedur aus:

Initialisieren zufälliger Cluster-Zentren (Schwerpunkte).

Wiederholen bis zur Konvergenz:

Zuweisen jedes Datenpunkts zum nächstgelegenen Cluster-Zentrum.

Neuberechnen jedes Cluster-Zentrums als Schwerpunkt aller ihm zugewiesenen

Datenpunkte.

Dies führt zu mehreren Durchläufen: Sie weisen zuerst die Daten den k Cluster-Zentren zu und dann berechnen Sie jedes Cluster-Zentrum als Schwerpunkt der Daten neu, die ihm zugewiesen sind.

Implementieren wir das!

Betrachten Sie folgendes Problem: Suchen Sie bei vorgegebenen zweidimensionalen Gehaltsdaten (*geleistete Stunden, verdientes Gehalt*) zwei Cluster von Angestellten in der gegebenen Datenmenge, die eine ähnliche Stundenanzahl leisten und ein ähnliches Gehalt bekommen.

Der Code

Wie können Sie all das in einer einzigen Codezeile schaffen? Zum Glück besitzt die scikit-learn-Bibliothek in Python bereits eine effiziente Implementierung des k-Means-Algorithmus. Listing 4-3 zeigt den Code mit dem Einzeiler, der die k-Means-Clusteranalyse für Sie erledigt.

```
## Abhängigkeiten

from sklearn.cluster import KMeans

import numpy as np

## Daten (Arbeit (h) / Gehalt ($))

X = np.array([[35, 7000], [45, 6900], [70, 7100],
```

```
[20, 2000], [25, 2200], [15, 1800]])

## Einzeiler

kmeans = KMeans(n_clusters=2).fit(X)

## Ergebnis und Ausgabe

cc = kmeans.cluster_centers_

print(cc)
```

Listing 4-3 *K-Means-Clusteranalyse in einer Zeile*

Welche Ausgabe hat dieser Code? Versuchen Sie, eine Lösung zu erraten, auch wenn Sie nicht alle syntaktischen Details verstehen. Dies wird Ihnen helfen, den Algorithmus besser zu durchschauen.

Wie es funktioniert

In den ersten Zeilen importieren Sie das `KMeans`-Modul aus dem `sklearn.cluster`-Paket. Dieses Modul kümmert sich um die Clusteranalyse selbst. Sie müssen außerdem die `NumPy`-Bibliothek importieren, weil das `KMeans`-Modul mit `NumPy`-Arrays arbeitet.

Unsere Daten sind zweidimensional. Sie setzen die Anzahl der Arbeitsstunden mit dem Gehalt einiger Angestellter in Beziehung. Abbildung 4-12 zeigt die sechs Datenpunkte in diesem Angestellten-Datensatz.

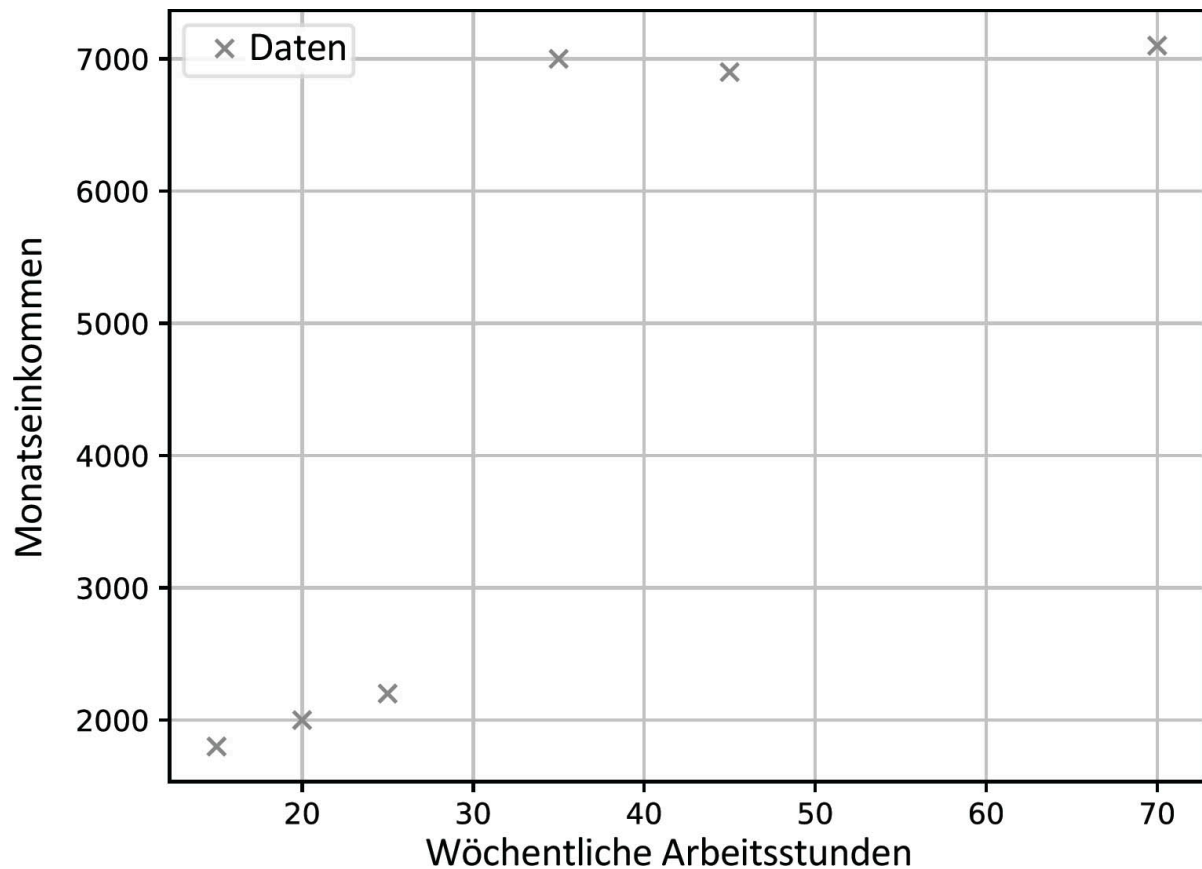


Abb. 4-12 Gehaltsdaten der Angestellten

Ziel ist es nun, die zwei Cluster-Zentren zu finden, die am besten zu diesen Daten passen:

```
## Einzeiler
```

```
kmeans = KMeans(n_clusters=2).fit(X)
```

In dem Einzeiler erzeugen Sie ein neues KMeans-Objekt, das sich für Sie um den Algorithmus kümmert. Wenn Sie das KMeans-Objekt anlegen, definieren Sie mit dem Funktionsargument `n_clusters` die Anzahl der Cluster-Zentren. Anschließend rufen Sie einfach die Instanzmethode `fit(X)` auf, um den k-Means-Algorithmus auf den Eingangsdaten `X` auszuführen. Das KMeans-Objekt enthält nun alle Ergebnisse. Nun müssen nur noch die Ergebnisse aus dessen Attributen abgerufen werden:

```
cc = kmeans.cluster_centers_
```

```
print(cc)
```

Beachten Sie, dass im `sklearn`-Paket die Konvention besteht, für einige Attributnamen einen nachgestellten Unterstrich zu verwenden (z. B. `cluster_centers_`), um anzuzeigen, dass diese Attribute innerhalb der Trainingsphase (der `fit()`-Funktion) dynamisch erzeugt wurden. Vor der Trainingsphase gibt es diese Attribute noch gar nicht. Das ist keine allgemeine Python-Konvention (nachgestellte Unterstriche werden normalerweise nur verwendet, um Namenskonflikte mit Python-Schlüsselwörtern zu vermeiden – `variable_list_` statt `list`). Wenn Sie sich daran gewöhnt haben, werden Sie die konsistente Nutzung von Attributen im `sklearn`-Paket zu schätzen wissen. Welches sind also die Cluster-Zentren, und wie sieht die Ausgabe dieses Codes aus? Werfen Sie einen Blick auf Abbildung 4–13.

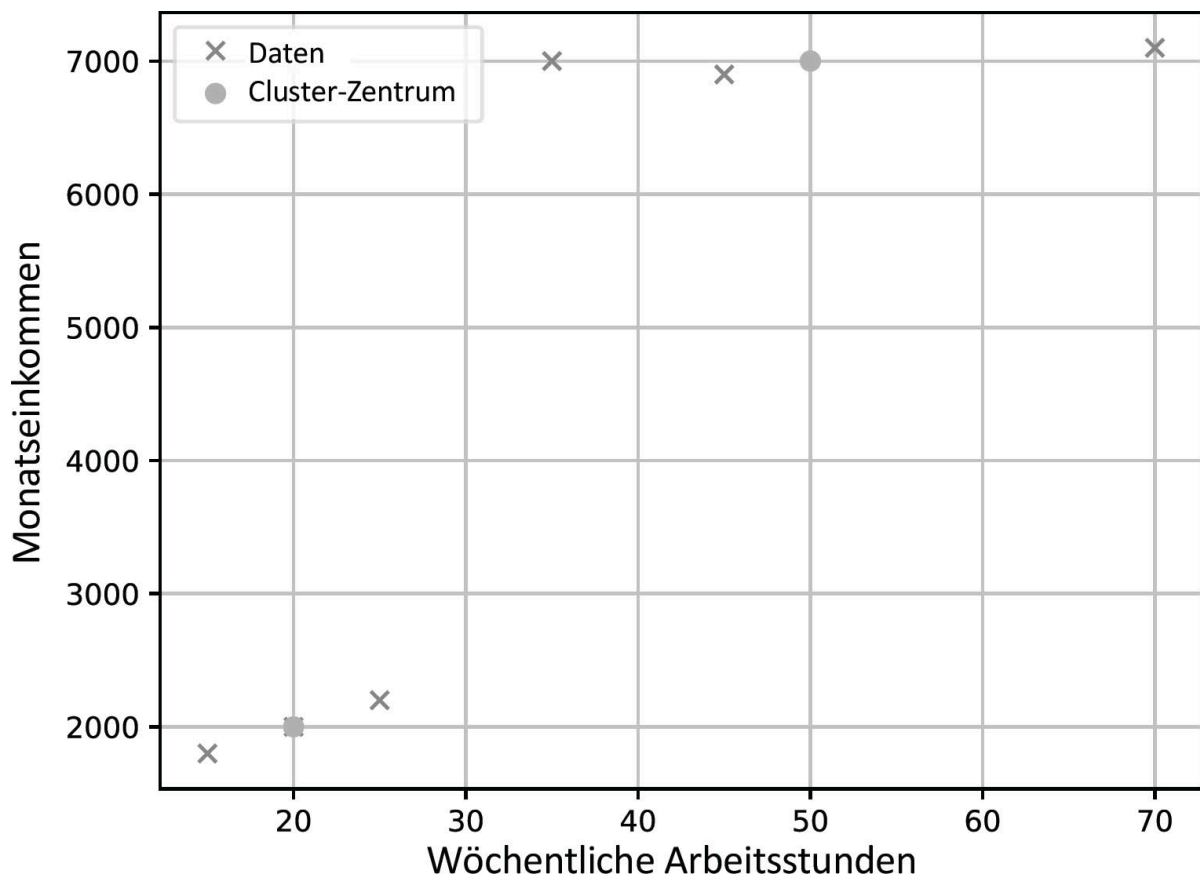


Abb. 4–13 Angestelltegehälter mit Cluster-Zentren im zweidimensionalen Raum

Sie erkennen, dass die zwei Cluster-Zentren (20, 2000) und (50, 7000) sind. Dieses Ergebnis liefert auch der Python-Einzeiler. Diese Cluster entsprechen zwei idealisierten Angestelltenrollen: Die erste arbeitet 20 Stunden pro Woche und verdient \$2000 im Monat, während die zweite 50 Stunden pro Woche arbeitet

und dafür \$7000 im Monat erhält. Diese zwei Rollentypen passen ziemlich gut zu den Daten. Das Ergebnis des Einzelers sieht daher folgendermaßen aus:

```
## Ergebnis und Ausgabe

cc = kmeans.cluster_centers_

print(cc)

'''

[[ 50. 7000.]

 [ 20. 2000.]]

'''
```

In diesem Abschnitt haben Sie ein wichtiges Teilgebiet des Unsupervised Learnings kennengelernt: die Clusteranalyse. Der k-Means-Algorithmus ist eine einfache, effiziente und beliebte Methode, um k Cluster aus mehrdimensionalen Daten zu extrahieren. Hinter den Kulissen berechnet der Algorithmus iterativ Cluster-Zentren neu und weist jeden Datenwert immer wieder neu dem nächstgelegenen Cluster-Zentrum zu, bis er die optimalen Cluster gefunden hat. Allerdings eignen sich Cluster nicht immer, um ähnliche Datenelemente zu finden. Viele Datensätze neigen nicht zur Cluster-Bildung, aber dennoch möchten Sie für Machine Learning und Vorhersagen die Abstandsinformationen nutzen. Bleiben wir im mehrdimensionalen Raum und erkunden wir einen anderen Weg, um den (euklidischen) Abstand von Datenwerten zu verwenden: den k-Nearest Neighbors-Algorithmus.

K-Nearest Neighbors in einer Zeile

Der beliebte *Algorithmus k Nearest Neighbors (KNN)* wird in vielen Anwendungen für Regression und Klassifizierung verwendet, etwa in Empfehlungssystemen, Bildklassifizierung und Finanzdatenvorhersage. Er bildet die Grundlage für viele komplexere Machine-Learning-Techniken (z. B. Information Retrieval). Es

besteht kein Zweifel, dass das Verständnis von KNN ein wichtiger Grundbaustein einer umfassenden Informatikausbildung ist.

Die Grundlagen

Der KNN-Algorithmus ist eine robuste, einfache und beliebte Machine-Learning-Methode. Er ist einfach zu implementieren, aber dennoch konkurrenzfähig und schnell. Alle anderen Machine-Learning-Modelle, die wir bisher diskutiert haben, nutzen die Trainingsdaten, um eine *Repräsentation* der Originaldaten zu berechnen. Mit dieser Repräsentation können Sie dann neue Daten vorhersagen, klassifizieren oder anhäufen. So definieren z. B. die linearen und logistischen Regressionsalgorithmen Lernparameter, während der Clusteranalyse-Algorithmus basierend auf den Trainingsdaten Cluster-Zentren berechnet. Der KNN-Algorithmus dagegen ist anders. Im Gegensatz zu den anderen Ansätzen berechnet er kein neues Modell (oder eine neue Repräsentation), sondern benutzt die *ganze Datenmenge* als Modell.

Ja, das haben Sie richtig gelesen. Das Machine-Learning-Modell ist nichts weiter als eine Menge an Beobachtungen. Jede einzelne Instanz Ihrer Trainingsdaten ist ein Teil Ihres Modells. Das hat Vor- und Nachteile. Ein Nachteil liegt darin, dass das Modell schnell sehr aufgebläht werden kann, wenn die Trainingsdaten zunehmen – sodass als Vorverarbeitung ein Sampling oder eine Filterung nötig werden könnte. Ein großer Vorteil ist dagegen die Einfachheit der Trainingsphase (Sie fügen einfach die neuen Datenwerte zu dem Modell hinzu). Zusätzlich können Sie den KNN-Algorithmus für Vorhersage oder Klassifizierung benutzen. Bei gegebenem Eingangsvektor x wenden Sie folgende Strategie an:

1. Suchen Sie die k nächsten Nachbarn von x (entsprechend einer vordefinierten Abstandsmetrik).
2. Fassen Sie die k nächsten Nachbarn zu einem einzigen Vorhersage- oder Klassifikationswert zusammen. Sie können dazu eine beliebige Sammelfunktion benutzen wie Durchschnitt, Mittelwert, Maximum oder Minimum.

Nehmen wir uns ein Beispiel vor. Ihr Unternehmen verkauft Häuser. Es hat eine große Datenbank an Kunden und Hauspreisen zusammengetragen (siehe Abbildung 4–14). Eines Tags fragt Ihr Kunde Sie, wie viel er wohl für ein Haus mit 52 Quadratmetern bezahlen muss. Sie fragen Ihr KNN-Modell ab, und es liefert Ihnen sofort die Antwort \$33.167. Und tatsächlich findet Ihr Kunde in genau dieser Woche ein Haus für \$33.489. Wie kam das KNN-System zu dieser überraschend exakten Vorhersage?

Zuerst berechnet das KNN-System mithilfe der euklidischen Distanz einfach die $k = 3$ nächsten Nachbarn zur Abfrage $D = 52 \text{ Quadratmeter}$. Die drei nächsten Nachbarn sind A, B und C mit den Preisen $\$34.000$, $\$33.500$ bzw. $\$32.000$. Anschließend fasst es die drei nächsten Nachbarn zusammen, indem es den einfachen Durchschnitt ihrer Werte berechnet. Da $k = 3$ ist in diesem Beispiel, bezeichnen Sie das Modell als $3NN$. Natürlich können Sie die Ähnlichkeitsfunktionen, den Parameter k und die Sammelmethode variieren, um zu anspruchsvolleren Vorhersagemodellen zu kommen.

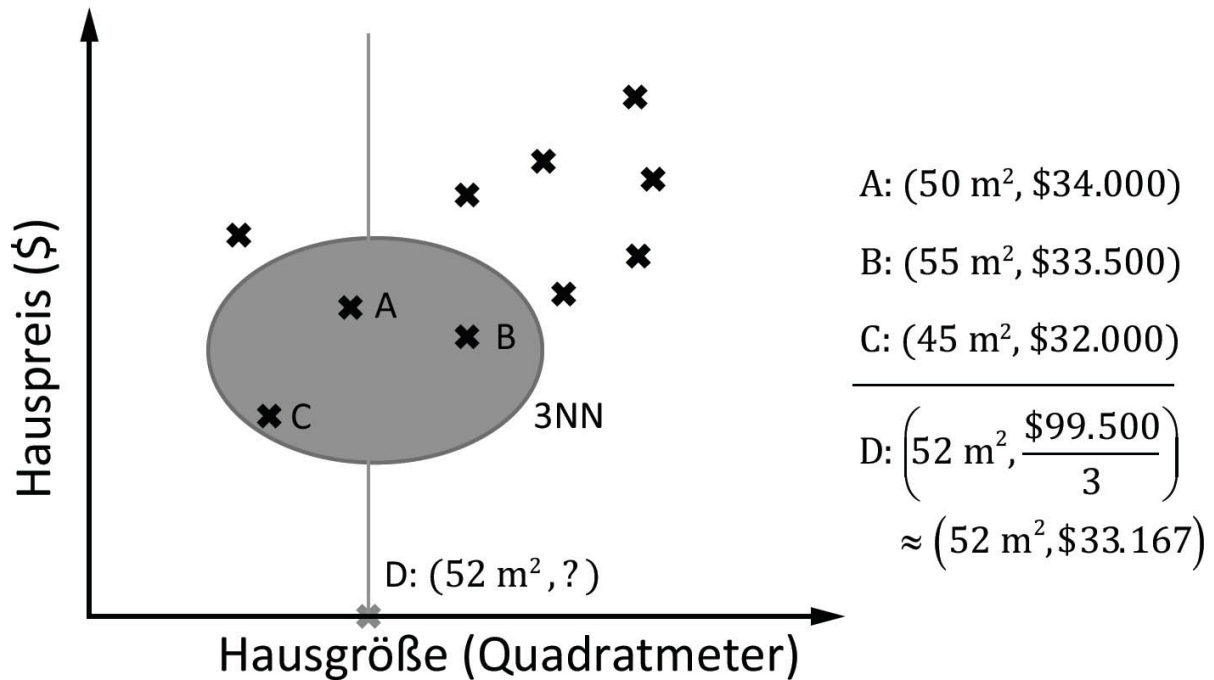


Abb. 4-14 Berechnen des Preises von Haus D auf der Grundlage der drei nächsten Nachbarn A, B und C

Ein weiterer Vorteil von KNN besteht darin, dass es sich leicht anpassen lässt, wenn neue Beobachtungen gemacht werden. Das gilt für Machine-Learning-Modelle im Allgemeinen nicht. Eine offensichtliche Schwäche in dieser Hinsicht ist, dass die Berechnungskomplexität der Suche nach den k nächsten Nachbarn immer mehr zunimmt, je mehr Punkte Sie hinzufügen. Um das auszugleichen, können Sie Werte, die für das Modell uninteressant geworden sind, laufend entfernen.

Wie erwähnt, lässt sich KNN auch für Klassifikationsprobleme einsetzen. Statt den Durchschnitt über die k nächsten Nachbarn zu ermitteln, können Sie einen Abstimmungsmechanismus verwenden: Jeder nächste Nachbar stimmt für seine Klasse, und die Klasse mit den meisten Stimmen gewinnt.

Der Code

Schauen wir uns an, wie Sie KNN in Python benutzen – in einer einzigen Zeile Code (siehe Listing 4–4).

```
## Abhängigkeiten

from sklearn.neighbors import KNeighborsRegressor

import numpy as np

## Daten (Hausgröße (Quadratmeter) / Hauspreis ($))

X = np.array([[35, 30000], [45, 45000], [40, 50000],
              [35, 35000], [25, 32500], [40, 40000]])

## Einzeiler

KNN =
KNeighborsRegressor(n_neighbors=3).fit(X[:,0].reshape(-1,1),
X[:,1])

## Ergebnis und Ausgabe

res = KNN.predict([[30]])

print(res)
```

Listing 4–4 Das Ausführen des KNN-Algorithmus in einer Zeile Python-Code

Raten Sie: Welche Ausgabe wird dieser Code haben?

Wie es funktioniert

Damit Sie das Ergebnis einfacher erkennen können, zeichnen wir ein Diagramm aus den Daten dieses Codes in Abbildung 4–15.

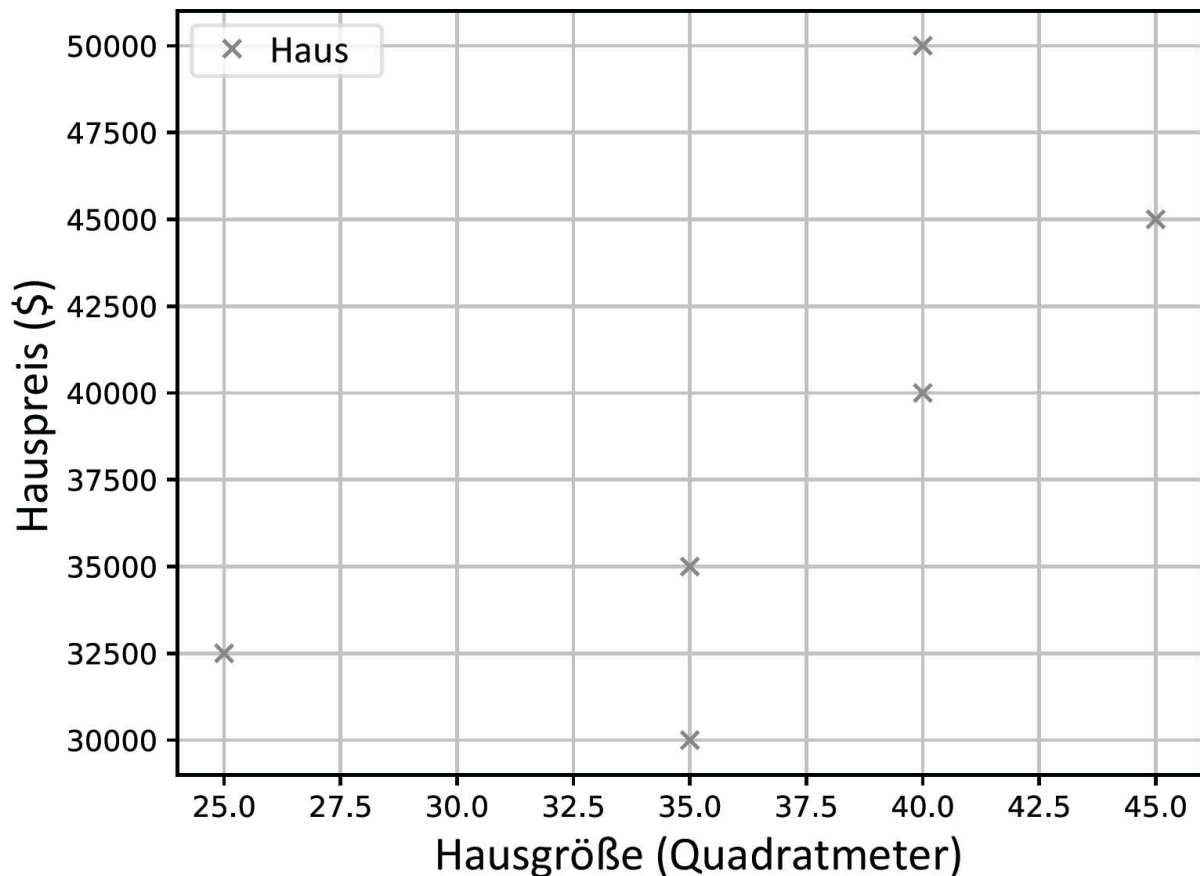


Abb. 4-15 Hausdaten im zweidimensionalen Raum

Erkennen Sie den allgemeinen Trend? Mit zunehmender Größe Ihres Hauses können Sie ein lineares Anwachsen seines Marktpreises erwarten. Verdoppeln Sie die Quadratmeter, verdoppelt sich auch der Preis.

Im Code (siehe Listing 4-4) verlangt der Kunde eine Preisvorhersage für ein Haus mit 30 Quadratmetern. Was sagt KNN mit $k = 3$ (kurz 3NN) vorher? Schauen Sie in Abbildung 4-16.

Schön, nicht wahr? Der KNN-Algorithmus findet die drei am dichtesten gelegenen Häuser in Bezug auf die Größe und berechnet den vorhergesagten Preis als Durchschnitt der $k=3$ nächsten Nachbarn. Daher das Ergebnis von \$32.500.

Falls die Datenkonvertierungen in dem Einzeiler Sie verwirren, erkläre ich noch einmal kurz, was hier passiert:

```
KNN =
KNeighborsRegressor(n_neighbors=3).fit(X[:,0].reshape(-1,1),
X[:,1])
```

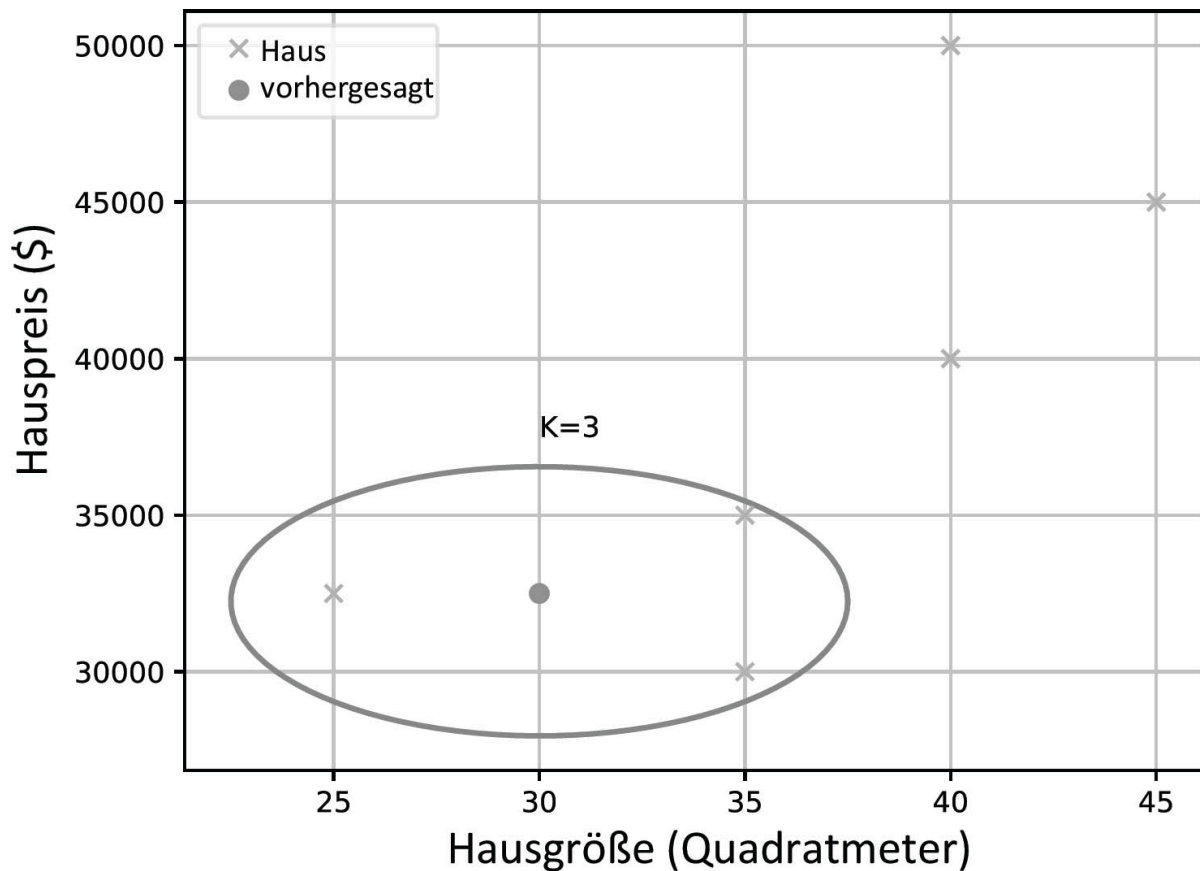


Abb. 4-16 Hausdaten im zweidimensionalen Raum mit vorhergesagtem Hauspreis für einen neuen Datenpunkt (Hausgröße gleich 30 Quadratmeter) mittels KNN

Zuerst erzeugen Sie ein neues Machine-Learning-Modell namens `KNeighborsRegressor`. Falls Sie KNN für die Klassifizierung verwenden, benutzen Sie `KNeighborsClassifier`.

Anschließend trainieren Sie das Modell mithilfe der `fit()`-Funktion mit zwei Parametern. Der erste Parameter definiert die Eingabe (die Hausgröße) und der zweite die Ausgabe (den Hauspreis). Beide Parameter müssen eine Array-artige Datenstruktur aufweisen. Um z. B. 30 als Eingabe zu benutzen, müssen Sie diesen Wert als `[30]` übergeben. Das liegt daran, dass die Eingabe im Allgemeinen mehrdimensional statt eindimensional sein kann. Deshalb formen Sie die Eingabe um:

```
print(X[:,0])

"[35 45 40 35 25 40]"

print(X[:,0].reshape(-1,1))
```

```
""""  
  
[[35]  
  
[45]  
  
[40]  
  
[35]  
  
[25]  
  
[40]]  
  
""""
```

Falls Sie übrigens dieses eindimensionale NumPy-Array als Eingabe für die `fit()`-Funktion benutzen, funktioniert das nicht, weil die Funktion ein Array aus (Array-artigem) Beobachten erwartet und kein Array aus Integer-Werten.

Dieser Einzeiler hat Ihnen also nun gezeigt, wie Sie Ihren ersten KNN-Regressor in einer einzigen Codezeile herstellen. Wenn Sie viele veränderliche Daten und Modell-Updates haben, ist KNN Ihr bester Freund! Kommen wir nun zu einem heutzutage unglaublich beliebten Machine-Learning-Modell: zu neuronalen Netzwerken.

Analyse neuronaler Netzwerke in einer Zeile

Neuronale Netzwerke (*Neural Networks*) haben in den letzten Jahren immens an Popularität gewonnen. Das liegt zum Teil daran, dass sich die Algorithmen und Lerntechniken in diesem Bereich verbessert haben, aber auch weil die Hardware leistungsfähiger geworden ist und wir heute über General-Purpose-GPU- (GPGPU-) Technologie verfügen. In diesem Abschnitt lernen Sie das *mehrlagige Perzeptron (Multilayer Perceptron; MLP)* kennen, das zu den beliebtesten neuronalen Netzwerken gehört. Nachdem Sie die nächsten Abschnitte gelesen haben, werden Sie in der Lage sein, in einer einzigen Zeile aus Python-Code Ihr eigenes neuronales Netzwerk zu schreiben!

Die Grundlagen

Für diesen Einzeiler habe ich gemeinsam mit Python-Kollegen von meiner Mailingliste einen speziellen Datensatz vorbereitet. Ziel war es, einen realen, nachvollziehbaren Datensatz zu schaffen, weshalb ich meine Abonnenten bat, für dieses Kapitel an einem Experiment zur Datengenerierung teilzunehmen.

Die Daten

Sie lesen dieses Buch, interessieren sich also dafür, Python zu lernen. Um einen interessanten Datensatz herzustellen, stellte ich meinen Abonnenten sechs anonymisierte Fragen über ihre Python-Kenntnisse und ihr Einkommen. Die Antworten auf diese Fragen dienen als Trainingsdaten für das einfache neuronale Beispielnetzwerk.

Die Trainingsdaten basieren auf den Antworten auf diese sechs Fragen:

- Wie viele Stunden lang haben Sie sich in den letzten sieben Tagen Python-Code angeschaut?
- Seit wie vielen Jahren beschäftigen Sie sich intensiv mit Informatik?
- Wie viele Programmierbücher besitzen Sie?
- Welchen Prozentsatz Ihrer Python-Zeit bringen Sie mit dem Arbeiten an echten Projekten zu?
- Wie viel verdienen Sie pro Monat (auf \$1000 gerundet) mit Ihren technischen Fertigkeiten (im weitesten Sinne)?
- Welches ist Ihre ungefähre Finxter-Bewertung, gerundet auf 100 Punkte?

Die ersten fünf Fragen sollen Ihre Eingabe bilden, während die sechste Frage die Ausgabe für die neuronale Netzwerkanalyse darstellen soll. In diesem Einzeilerabschnitt untersuchen Sie die neuronale Netzwerkregression. Mit anderen Worten, basierend auf den numerischen Eingangs-Features sagen Sie einen numerischen Wert (Ihre Python-Fertigkeiten) vorher. Wir werden uns in diesem Buch nicht mit der Klassifikation mithilfe neuronaler Netzwerke befassen, die eine weitere große Stärke neuronaler Netzwerke ist.

Die sechste Frage schätzt das Maß der Fertigkeiten eines Python-Programmierers. Finxter (<https://finxter.com/>) ist unsere rätselbasierte Lernanwendung, die Python-Programmierern eine Bewertung zuweist, die sich danach richtet, wie gut sie Python-Rätsel lösen. Auf diese Weise können Sie quantifizieren, wie gut Sie Python beherrschen.

Beginnen wir damit, zu visualisieren, wie die einzelnen Fragen die Ausgabe (die Bewertung eines Python-Programmierers) beeinflussen (siehe Abbildung 4-17).

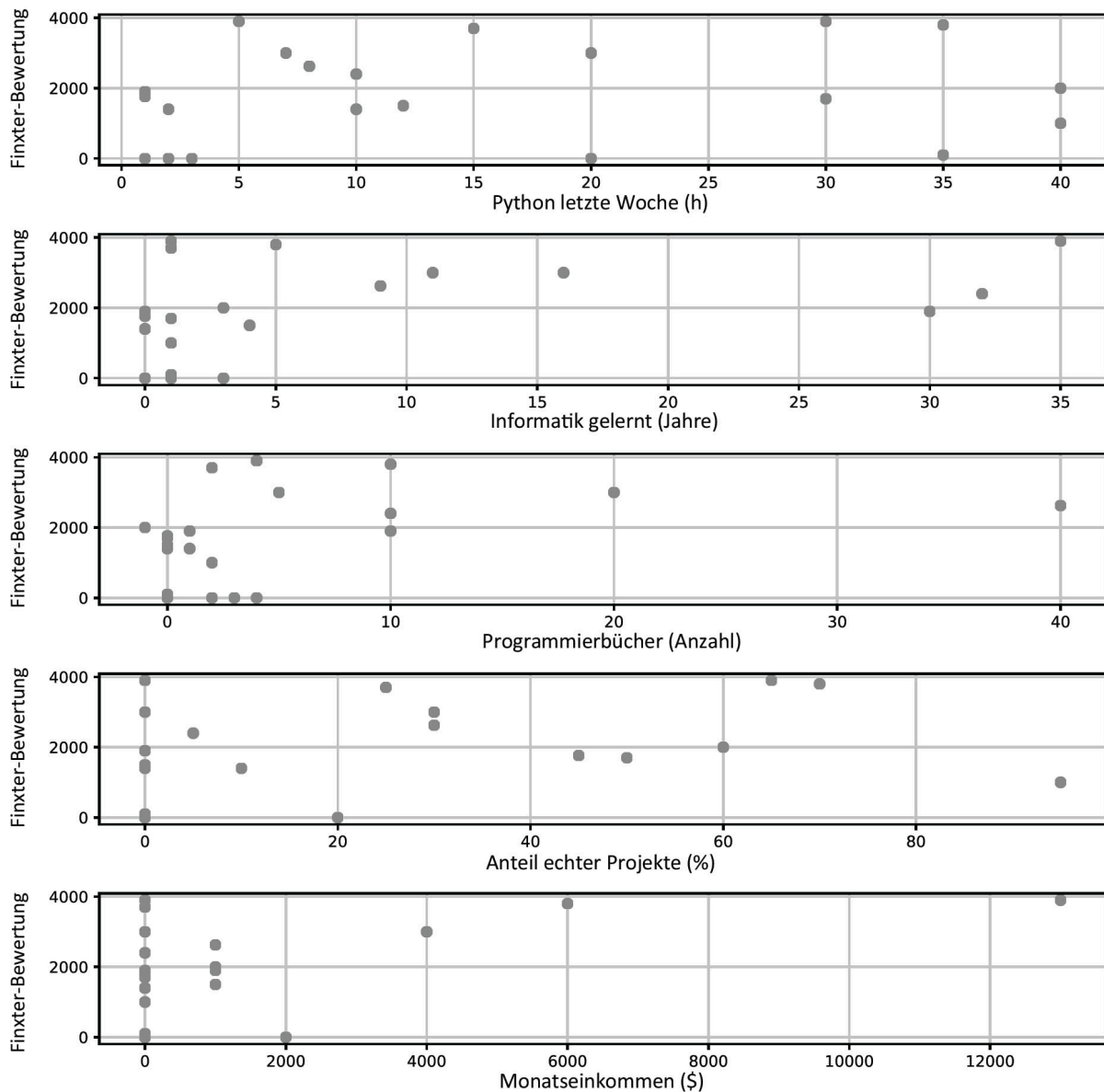


Abb. 4-17 Beziehung zwischen den Antworten des Fragebogens und der Python-Bewertung auf Finxter

Beachten Sie, dass diese Diagramme nur zeigen, wie jedes Feature (jede Frage) für sich die fertige Finxter-Bewertung beeinflusst. Sie verraten Ihnen nicht, wie sich eine Kombination aus zwei oder mehr Features auswirkt. Beachten Sie außerdem, dass manche Pythonistas nicht alle sechs Fragen beantwortet haben; in diesen Fällen nutzte ich den Dummy-Wert -1.

Was ist ein künstliches neuronales Netzwerk?

Die Idee, ein theoretisches Modell des menschlichen Gehirns (des biologischen neuronalen Netzwerks) zu erschaffen, wurde in den letzten Jahrzehnten umfassend untersucht. Die Grundlagen künstlicher neuronaler Netzwerke stammen sogar schon aus den 1940ern und 1950ern! Seitdem wurde das Konzept der künstlichen neuronalen Netzwerke verfeinert und kontinuierlich verbessert.

Der Grundgedanke ist, die große Aufgabe des Lernens und der Inferenz (des Schlussfolgerns) in mehrere Mikroaufgaben zu zerlegen. Diese Mikroaufgaben sind nicht unabhängig, sondern voneinander abhängig und miteinander verflochten. Das Gehirn besteht aus Milliarden von Neuronen, die mit Billionen von Synapsen verbunden sind. Im vereinfachten Modell ist das Lernen lediglich das Verändern der *Stärke* der Synapsen (in künstlichen neuronalen Netzwerken auch als *Gewichte* oder *Parameter* bezeichnet). Wie »erzeugen« Sie also im Modell eine neue Synapse? Ganz einfach: Sie erhöhen ihr Gewicht von null auf einen Wert, der nicht null ist.

Abbildung 4–18 zeigt ein einfaches neuronales Netzwerk mit drei Schichten (Eingang, verborgen, Ausgang). Jede Schicht besteht aus mehreren Neuronen, die von der Eingangsschicht über die verborgene Schicht bis zur Ausgangsschicht miteinander verbunden sind.

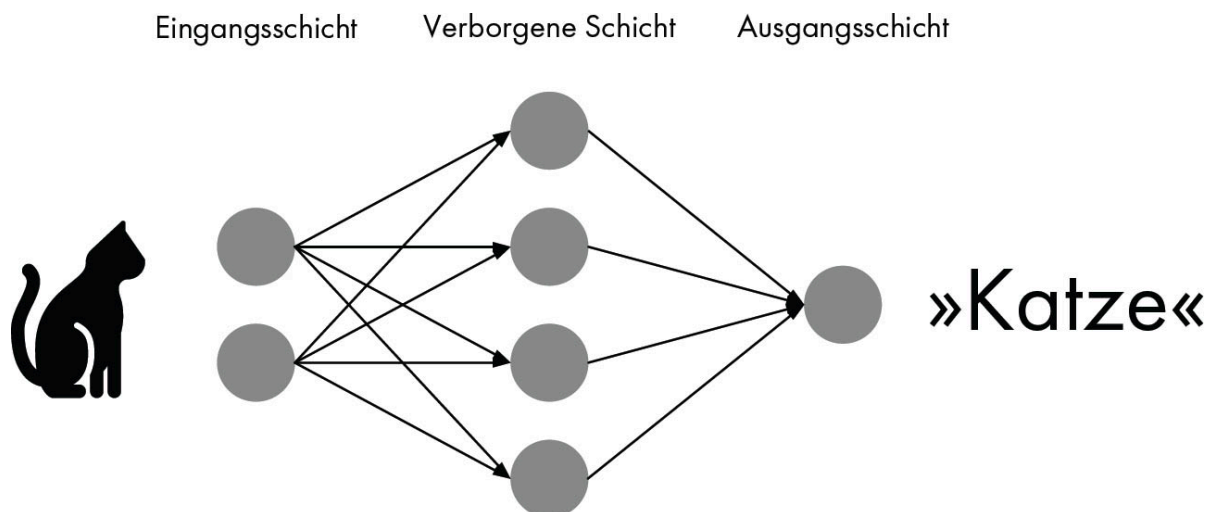


Abb. 4–18 Eine Analyse mit einem einfachen neuronalen Netzwerk zur Klassifikation von Tieren

In diesem Beispiel wurde das neuronale Netzwerk trainiert, Tiere in Bildern zu erkennen. In der Praxis würden Sie ein Eingangsneuron pro Bildpixel in der Eingangsschicht benutzen. Das heißt, Sie hätten möglicherweise Millionen von Eingangsneuronen, die mit Millionen von verborgenen Neuronen verbunden sind. Oft ist jedes Ausgangsneuron für einen kleinen Teil der Gesamtausgabe verantwortlich. Um z. B. zwei unterschiedliche Tiere zu erkennen (wie etwa

Katzen und Hunde), verwenden Sie nur ein einziges Neuron in der Ausgangsschicht, das zwei unterschiedliche Zustände (0=cat, 1=dog) modellieren kann.

Die Idee dahinter ist, dass jedes Neuron aktiviert werden kann oder »feuert«, wenn ein bestimmter Eingangsimpuls bei ihm ankommt. Jedes Neuron entscheidet auf Basis der Stärke des Eingangsimpulses unabhängig, ob es feuert oder nicht. Auf diese Weise simulieren Sie das menschliche Gehirn, in dem die Neuronen einander über Impulse aktivieren. Die Aktivierung der Eingangsneuronen setzt sich durch das Netzwerk fort, bis die Ausgangsneuronen erreicht werden. Manche Ausgangsneuronen werden aktiviert, andere dagegen nicht. Das spezielle Muster der feuernden Ausgangsneuronen bildet dann die fertige Ausgabe (oder Vorhersage) Ihres künstlichen neuronalen Netzwerks. In Ihrem Modell könnte ein feuerndes Ausgangsneuron eine 1 codieren und ein nicht feuerndes Ausgangsneuron eine 0. Auf diese Weise können Sie Ihr neuronales Netzwerk darauf trainieren, alles vorherzusagen, das als eine Folge aus 0 und 1 codiert werden kann (also alles, was ein Computer repräsentieren kann).

Schauen wir uns in Abbildung 4–19 genauer an, wie Neuronen mathematisch funktionieren.

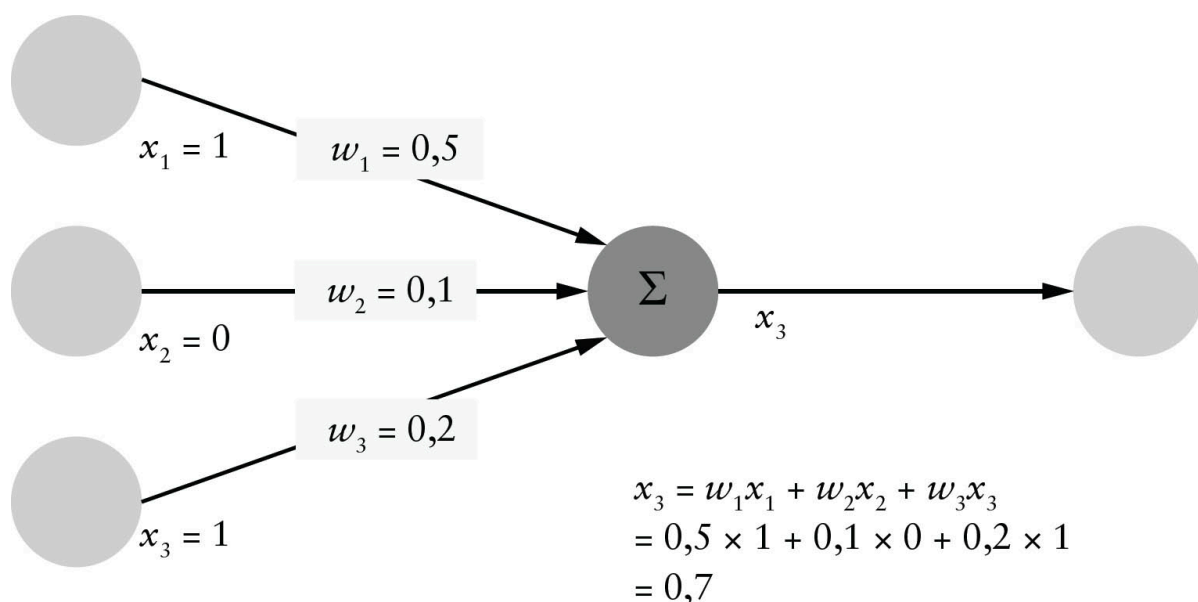


Abb. 4–19 Mathematisches Modell eines einzelnen Neurons: Die Ausgabe ist eine Funktion der drei Eingaben.

Jedes Neuron ist mit den anderen Neuronen verbunden, aber nicht alle Verbindungen sind gleich. Stattdessen ist jeder Verbindung ein Gewicht zugewiesen. Im Prinzip gibt ein feuerndes Neuron einen Impuls von 1 an die nachfolgenden Nachbarn weiter, während ein nicht feuerndes Neuron einen Impuls von 0 verbreitet. Stellen Sie sich das Gewicht so vor: Es zeigt an, wie viel

des Impulses des feuernenden Eingangsneurons über die Verbindung an das nächste Neuron weitergegeben wird. Mathematisch gesehen multiplizieren Sie den Impuls mit dem Gewicht der Verbindung, um so die Eingabe für das nächste Neuron zu berechnen. In unserem Beispiel addiert das Neuron einfach alle Eingaben, um seine eigene Ausgabe zu berechnen. Dies ist die *Aktivierungsfunktion*, die beschreibt, wie genau die Eingaben eines Neurons eine Ausgabe generieren. In unserem Beispiel feuert ein Neuron mit höherer Wahrscheinlichkeit, wenn seine relevanten Eingangsneuronen ebenfalls feuern. Auf diese Weise breiten sich die Impulse durch das neuronale Netzwerk aus.

Was macht nun der Lernalgorithmus? Er verwendet die Trainingsdaten, um die Gewichte w des neuronalen Netzwerks auszuwählen. Bei einem bestimmten Trainingseingangswert x führen unterschiedliche Gewichte w zu unterschiedlichen Ausgaben. Daher ändert der Lernalgorithmus nach und nach – in vielen Iterationen – die Gewichte w , bis die Ausgangsschicht ähnliche Ergebnisse erzeugt wie die Trainingsdaten. Mit anderen Worten, der Trainingsalgorithmus reduziert schrittweise den Fehler bei der Vorhersage der Trainingsdaten.

Es gibt viele Netzwerkstrukturen, Trainingsalgorithmen und Aktivierungsfunktionen. Dieses Kapitel zeigt Ihnen auf praktische Weise, wie Sie das neuronale Netzwerk in nur einer Codezeile einsetzen. Später können Sie dann bei Bedarf die feineren Details lernen. (Sie könnten z. B. mit den verschiedenen Wikipedia-Artikeln zum Thema »Neuronales Netz« und »Künstliches neuronales Netz« beginnen.)

Der Code

Unser Ziel ist ein neuronales Netzwerk, das die Güte der Python-Fertigkeiten (Finxter-Bewertung) vorhersagt, indem es die fünf Eingangs-Features (Antworten auf Fragen) zurate zieht:

WOCHE Wie viele Stunden haben Sie in den letzten sieben Tagen Python-Code angeschaut?

JAHRE Seit wie vielen Jahren beschäftigen Sie sich intensiv mit Informatik?

BÜCHER Wie viele Programmierbücher besitzen Sie?

PROJEKTE Welchen Prozentsatz Ihrer Python-Zeit verbringen Sie damit, echte Projekte zu implementieren?

GEHALT Wie viel verdienen Sie pro Monat (auf \$1000 gerundet) mit Ihren technischen Fertigkeiten (im weitesten Sinne)?

Auch hier greifen wir auf die Leistungen unserer Vorgänger zurück und nutzen wie in Listing 4–5 die scikit-learn-Bibliothek (sklearn) für die neuronale Netzwerkregression.

```
## Abhängigkeiten

from sklearn.neural_network import MLPRegressor

import numpy as np

## Fragebogendaten (WOCHE, JAHRE, BÜCHER, PROJEKTE, GEHALT,
BEWERTUNG)

X = np.array(

    [[20, 11, 20, 30, 4000, 3000],

     [12, 4, 0, 0, 1000, 1500],

     [2, 0, 1, 10, 0, 1400],

     [35, 5, 10, 70, 6000, 3800],

     [30, 1, 4, 65, 0, 3900],

     [35, 1, 0, 0, 0, 100],

     [15, 1, 2, 25, 0, 3700],

     [40, 3, -1, 60, 1000, 2000],

     [40, 1, 2, 95, 0, 1000],

     [10, 0, 0, 0, 0, 1400],
```

```

[30, 1, 0, 50, 0, 1700],

[1, 0, 0, 45, 0, 1762],

[10, 32, 10, 5, 0, 2400],

[5, 35, 4, 0, 13000, 3900],

[8, 9, 40, 30, 1000, 2625],

[1, 0, 1, 0, 0, 1900],

[1, 30, 10, 0, 1000, 1900],

[7, 16, 5, 0, 0, 3000]])

## Einzeiler

neural_net = MLPRegressor(max_iter=10000).fit(X[:, :-1],
X[:, -1])

## Ergebnis

res = neural_net.predict([[0, 0, 0, 0, 0]])

print(res)

```

Listing 4-5 Analyse mit neuronalem Netzwerk in einer einzigen Codezeile

Für einen Menschen ist es unmöglich, die Ausgabe korrekt herauszubekommen – aber vielleicht wollen Sie es doch versuchen?

Wie es funktioniert

In den ersten Zeilen erzeugen Sie den Datensatz. Die Machine-Learning-Algorithmen in der scikit-learn-Bibliothek verwenden ein ähnliches Eingangsformat. Jede Zeile ist eine einzelne Beobachtung mit mehreren Features. Je mehr Zeilen es gibt, umso mehr Trainingsdaten existieren; je mehr Spalten es gibt, umso mehr Features hat jede Beobachtung. In diesem Fall haben Sie fünf Features für die Eingabe und ein Feature für den Ausgangswert pro Trainingsdaten.

Der Einzeiler erzeugt mithilfe des Konstruktors der MLPRegressor-Klasse ein neuronales Netzwerk. Ich übergab `max_iter=10000` als Argument, weil das Training nicht zum Ende kommt, wenn man die Standardanzahl an Iterationen (`max_iter=200`) benutzt.

Anschließend rufen Sie die `fit()`-Funktion auf, die die Parameter des neuronalen Netzwerks ermittelt. Nach dem Aufruf von `fit()` ist das neuronale Netzwerk erfolgreich initialisiert. Die `fit()`-Funktion nimmt ein mehrdimensionales Eingangs-Array (eine Beobachtung pro Zeile, ein Feature pro Spalte) und ein eindimensionales Ausgangs-Array (Größe = Anzahl der Beobachtungen) entgegen.

Jetzt fehlt nur noch der Aufruf der `predict`-Funktion mit einigen Eingangswerten:

```
## Ergebnis

res = neural_net.predict([[0, 0, 0, 0, 0]])

print(res)

# [94.94925927]
```

Beachten Sie, dass die eigentliche Ausgabe aufgrund der nicht deterministischen Natur der Funktion und des unterschiedlichen Konvergenzverhaltens ein wenig anders ausfallen kann.

Ganz einfach ausgedrückt: Falls Sie ...

- ... in der letzten Woche 10 Stunden geübt haben,
- ... Ihre Informatikstudien vor 0 Jahren begonnen haben,

- ... 0 Programmierbücher besitzen,
- ... 0 Prozent Ihrer Zeit mit dem Implementieren echter Python-Projekte zugebracht haben und
- ... \$0 mit Ihren Programmierkünsten verdienen,

schätzt das neuronale Netzwerk Ihre Fertigkeiten als *sehr* gering ein (eine Finxter-Bewertung von 94 bedeutet, dass Sie Probleme haben, das Python-Programm `print("hello, world")` zu verstehen).

Ändern wir das also: Was passiert, wenn Sie 20 Stunden pro Woche für das Lernen aufwenden und nach einer Woche erneut das neuronale Netzwerk bemühen:

```
## Ergebnis

res = neural_net.predict([[20, 0, 0, 0, 0]])

print(res)

# [440.40167562]
```

Nicht schlecht – Ihre Fertigkeiten haben sich deutlich verbessert! Aber Sie sind mit dieser Bewertung immer noch nicht glücklich, oder? (Ein überdurchschnittlicher Python-Programmierer hat auf Finxter wenigstens eine Bewertung von 1.500 bis 1.700.)

Kein Problem. Kaufen Sie 10 Python-Bücher (nur noch neun nötig nach diesem hier). Was passiert mit Ihrer Bewertung:

```
## Ergebnis

res = neural_net.predict([[20, 0, 10, 0, 0]])

print(res)

# [953.6317602]
```

Sie machen erneut große Fortschritte und verbessern Ihre Bewertung! Es reicht aber nicht, die Python-Bücher nur zu kaufen. Sie müssen sie auch gründlich durcharbeiten! Machen Sie das für ein Jahr:

```
## Ergebnis

res = neural_net.predict([[20, 1, 10, 0, 0]])

print(res)

# [999.94308353]
```

Es passiert nicht viel. Das ist die Stelle, an der ich dem neuronalen Netzwerk nicht allzu sehr vertraue. Meiner Meinung nach sollten Sie eine viel bessere Leistung von wenigstens 1.500 erreicht haben. Allerdings zeigt das auch, dass das neuronale Netzwerk nur so gut sein kann wie seine Trainingsdaten. Sie haben nur beschränkte Daten, und das neuronale Netzwerk kann diese Beschränkung kaum überwinden: In einer Handvoll von Datenpunkten steckt einfach zu wenig Wissen.

Aber Sie geben nicht auf, oder? Als Nächstes wenden Sie 50 Prozent Ihrer Python-Zeit dafür auf, Ihre Fertigkeiten als selbstständiger Python-Programmierer zu verkaufen:

```
## Ergebnis

res = neural_net.predict([[20, 1, 10, 50, 1000]])

print(res)

# [1960.7595547]
```

Boom! Plötzlich betrachtet das neuronale Netzwerk Sie als Python-Experten. Wirklich eine weise Vorhersage des neuronalen Netzwerks! Lernen Sie wenigstens ein Jahr lang, Python, verdienen Sie Geld mit Ihren Python-Fähigkeiten, und Sie werden ein großartiger Programmierer.

Sie haben also die Grundlagen neuronaler Netzwerke kennengelernt und erfahren, wie Sie sie in einer einzigen Zeile Python-Code einsetzen. Interessanterweise deuten die Fragebogendaten an, dass es viel zu Ihrem Lernerfolg beiträgt, mit praktischen Projekten zu beginnen – vielleicht sogar selbstständig Projekte zu bearbeiten. Zumindest dem neuronalen Netzwerk ist das klar. Wenn Sie wissen wollen, mit welcher Strategie ich zum Selbstständigen geworden bin, schauen Sie sich mein kostenloses Webinar über modernes Arbeiten als selbstständiger Python-Programmierer an: <https://blog.finxter.com/webinar-freelancer/>.

Im nächsten Abschnitt steigen Sie tiefer in eine weitere leistungsstarke Modellrepräsentation ein: Entscheidungsbäume. Während das Training neuronaler Netzwerke recht aufwendig und teuer sein kann (oft brauchen Sie mehrere Computer und viele Stunden oder gar Wochen zum Trainieren), sind Entscheidungsbäume regelrechte Leichtgewichte. Dennoch sind sie schnelle, effektive Möglichkeiten, Muster aus Ihren Trainingsdaten zu ziehen.

Decision-Tree Learning in einer Zeile

Entscheidungsbäume (Decision Trees) sind leistungsstarke und intuitive Werkzeuge in Ihrem Machine-Learning-Werkzeugkasten. Ein großer Vorteil von Entscheidungsbäumen besteht darin, dass sie anders als viele andere Machine-Learning-Techniken vom Menschen gelesen werden können. Sie können ganz leicht einen Entscheidungsbaum trainieren und dann Ihren Vorgesetzten zeigen, die nichts über das Machine Learning wissen müssen, um zu verstehen, was Ihr Modell macht. Das ist besonders für Data Scientists großartig, die ihre Resultate häufig vor dem Management präsentieren und verteidigen müssen. In diesem Abschnitt zeige ich Ihnen, wie Sie Entscheidungsbäume in einer einzigen Zeile Python-Code verwenden.

Die Grundlagen

Anders als viele Machine-Learning-Algorithmen könnten Ihnen die Ideen hinter den Entscheidungsbäumen aus Ihren eigenen Erfahrungen heraus bekannt vorkommen. Sie stellen eine strukturierte Methode dar, Entscheidungen zu treffen. Jede Entscheidung eröffnet neue Wege bzw. Zweige. Indem Sie einen Haufen Fragen beantworten, landen Sie schließlich beim empfohlenen Ergebnis. Abbildung 4–20 zeigt ein Beispiel.

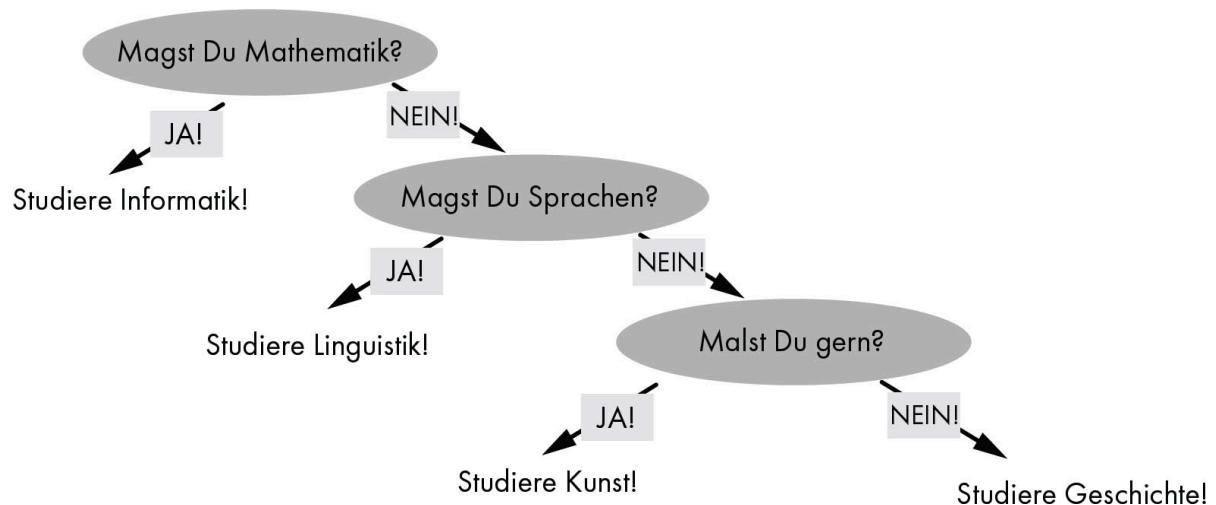


Abb. 4-20 Ein vereinfachter Entscheidungsbaum zum Empfehlen eines Studienfachs

Entscheidungsbäume werden für Klassifikationsprobleme verwendet wie »Welches Fach sollte ich angesichts meiner Interessen studieren?«. Sie starten ganz oben. Dann beantworten Sie wiederholt Fragen und wählen die Antworten aus, die Ihre Eigenarten am besten beschreiben. Am Ende erreichen Sie einen *Blattknoten* des Baums, einen Knoten ohne *Kinder*. Dies ist die angesichts Ihrer Features empfohlene Klasse.

Das Lernen mit Entscheidungsbäumen besitzt viele Nuancen. Im gezeigten Beispiel hat die erste Frage mehr Gewicht als die letzte. Wenn Sie Mathematik mögen, wird der Entscheidungsbaum niemals Kunst oder Linguistik empfehlen. Das ist sinnvoll, da einige Eigenschaften für die Klassifikationsentscheidung wichtiger sein könnten als andere. So könnte etwa ein Klassifikationssystem, das Ihren aktuellen Gesundheitszustand vorhersagt, Ihr Geschlecht (Feature) benutzen, um viele Krankheiten (Klasse) praktischerweise gleich auszuschließen.

Das bedeutet, dass die Reihenfolge der Entscheidungsknoten Einfluss auf die Optimierungsmöglichkeiten des Systems hat: Setzen Sie die Features an die Spitze des Baums, die einen großen Einfluss auf die letztendliche Klassifizierung haben. Beim Decision-Tree Learning setzen Sie Fragen mit geringem Einfluss ans Ende, wie Abbildung 4-21 zeigt.

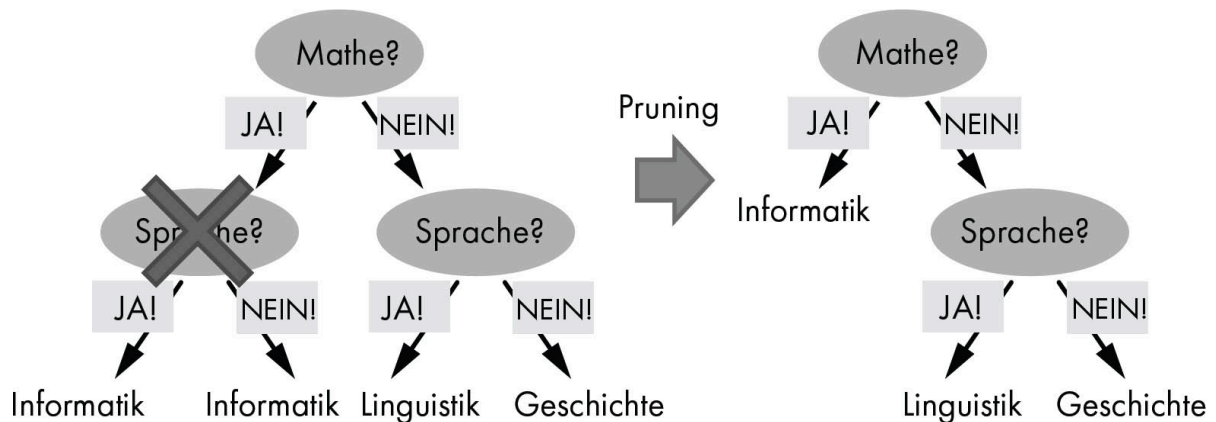


Abb. 4-21 Das Pruning des Entscheidungsbaums verbessert seine Effizienz.

Nehmen Sie an, der volle Entscheidungsbaum sieht aus wie der linke Baum. Für jede Kombination aus Features gibt es ein separates Klassifikationsergebnis (die Blätter des Baums). Manche Features liefern Ihnen jedoch möglicherweise keine zusätzlichen Informationen in Bezug auf das Klassifikationsproblem (wie etwa der erste Sprachentscheidungsknoten im Beispiel). Beim Decision-Tree Learning würde man diese Knoten aus Gründen der Effizienz entfernen. Dieser Prozess wird als *Pruning* (Beschneiden, Zurechtstutzen) bezeichnet.

Der Code

Sie können in einer einzigen Zeile Python-Code Ihren eigenen Entscheidungsbaum herstellen. Listing 4-6 zeigt Ihnen, wie das geht.

```
## Abhängigkeiten

from sklearn import tree

import numpy as np

## Daten: Studentenleistungen in (Mathe, Sprachen,
Kreativität) -> Studien-

fach

X = np.array([[9, 5, 6, "Informatik"],
              [1, 8, 1, "Linguistik"],
```

```

[5, 7, 9, "Kunst"]])

## Einzeiler

Tree = tree.DecisionTreeClassifier().fit(X[:, :-1], X[:, -1])

## Ergebnis und Ausgabe

student_0 = Tree.predict([[8, 6, 5]])

print(student_0)

student_1 = Tree.predict([[3, 7, 9]])

print(student_1)

```

Listing 4-6 Entscheidungsbaumklassifikation in einer einzigen Codezeile

Erraten Sie doch einmal die Ausgabe dieses Codeschnipsels!

Wie es funktioniert

Die Daten in diesem Code beschreiben drei Studenten mit ihren geschätzten Fähigkeiten (auf einer Skala von von 1 bis 10) in den drei Bereichen Mathematik, Sprachen und Kreativität. Sie kennen außerdem die Studienfächer dieser Studenten. So ist etwa der erste Student sehr gut in Mathematik und studiert Informatik. Der zweite Student ist in Sprachen viel besser als in den beiden anderen Bereichen und studiert Linguistik. Der dritte Student ist besonders kreativ und studiert Kunst.

Der Einzeiler erzeugt ein neues Entscheidungsbaumobjekt und trainiert das Modell, indem er die `fit()`-Funktion mit den benannten Trainingsdaten ausführt (die letzte Spalte ist das Label). Intern erzeugt er drei Knoten, d. h. einen für jedes Feature: Mathe, Sprachen und Kreativität. Beim Vorhersagen des Fachs für `student_0` (Mathe = 8, Sprachen = 6, Kreativität = 5) liefert der Entscheidungsbaum `Informatik` zurück. Er hat gelernt, dass dieses Feature-Muster (hoch, mittel, mittel) ein Anzeichen für das erste Fach ist. Wird dagegen

nach (3, 7, 9) gefragt, sagt der Entscheidungsbaum Kunst vorher, weil er gelernt hat, dass die Leistungsbewertung (niedrig, mittel, hoch) auf das dritte Fach hindeutet.

Beachten Sie, dass der Algorithmus nicht deterministisch ist. Mit anderen Worten, es können sich andere Ergebnisse zeigen, wenn der Code zweimal ausgeführt wird. Das ist bei Machine-Learning-Algorithmen üblich, die mit Zufallsgeneratoren arbeiten. In diesem Fall ist die Reihenfolge der Features zufällig organisiert, sodass der letzte Entscheidungsbaum eine andere Anordnung der Features haben kann.

Entscheidungsbäume bilden also eine intuitive Möglichkeit, vom Menschen lesbare Machine-Learning-Modelle herzustellen. Jeder Zweig repräsentiert eine Wahl, die auf einem einzigen Feature einer neuen Probe beruht. Die Blätter des Baums stellen die fertige Vorhersage (Klassifikation oder Regression) dar. Als Nächstes verlassen wir die Machine-Learning-Algorithmen für einen Augenblick und wenden uns einem wichtigen Konzept im Machine Learning zu, der Varianz.

Die minimale Varianz einer Zeile berechnen

Sie haben vielleicht schon von den Vs in Big Data gehört: Volume (Datenmenge), Velocity (Änderungsgeschwindigkeit der Daten), Variety (Datenvielfalt), Veracity (Glaubwürdigkeit der Daten) und Value (unternehmerischer Wert der Daten). Die *Varianz* ist ein weiteres wichtiges V: Sie misst die erwartete (quadratische) Abweichung der Daten von ihrem Mittelwert. In der Praxis ist die Varianz ein wichtiges Maß, das vor allem bei Finanzdienstleistungen, der Wettervorhersage und der Bildbearbeitung Relevanz besitzt.

Die Grundlagen

Die Varianz misst, wie sehr sich die Daten im ein- oder mehrdimensionalen Raum rund um ihren Durchschnittswert herum ausbreiten. Sie werden gleich ein grafisches Beispiel sehen. Wenn man genau ist, gehört die Varianz sogar zu den wichtigsten Eigenschaften beim Machine Learning. Sie erfasst das Muster der Daten auf verallgemeinerte Weise – und beim Machine Learning geht es ja vor allem um Mustererkennung.

Viele Machine-Learning-Algorithmen greifen in der einen oder anderen Form auf die Varianz zurück. Beispielsweise ist das *Verzerrung-Varianz-Dilemma* (Bias-Variance Trade-off) ein bekanntes Problem beim Machine Learning: Hochentwickelte Machine-Learning-Modelle riskieren eine Überanpassung der Daten (hohe Varianz), stellen die Trainingsdaten aber sehr akkurat dar (niedrige

Verzerrung). Andererseits generalisieren einfache Modelle oft sehr gut (niedrige Varianz), stellen aber die Daten nicht akkurat dar (hohe Verzerrung).

Was genau ist also Varianz? Sie ist einfach eine statistische Eigenschaft, die festhält, wie sehr sich die Datenmenge von ihrem Mittelwert her ausbreitet. Abbildung 4-22 zeigt ein Beispieldiagramm mit zwei Datenmengen: Eine hat eine niedrige Varianz, die andere eine hohe.

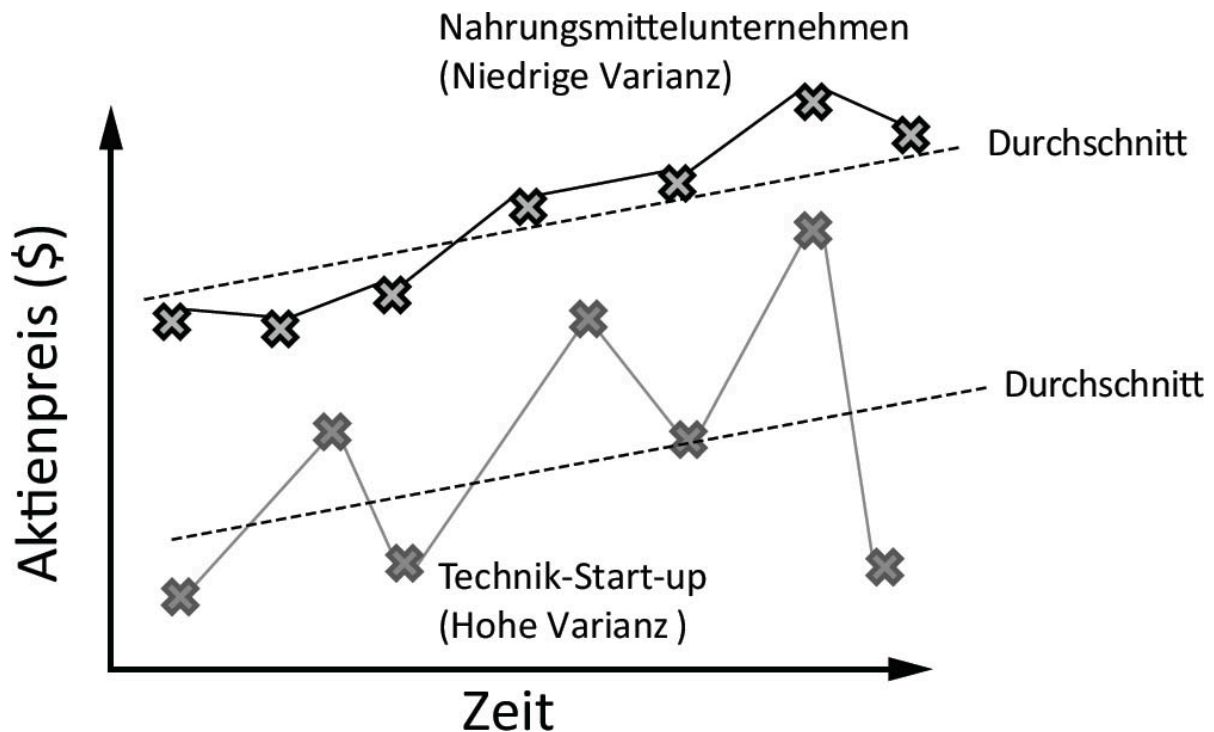


Abb. 4-22 Vergleich der Varianz zweier Aktienpreise

Dieses Beispiel zeigt die Aktienpreise zweier Unternehmen. Der Aktienpreis des Technik-Start-ups schwankt stark um seinen Durchschnittswert. Der Aktienpreis des Nahrungsmittelunternehmens ist dagegen recht stabil und weicht nur wenig von seinem Durchschnittswert ab. Mit anderen Worten, das Technik-Start-up hat eine hohe Varianz, das Nahrungsmittelunternehmen eine niedrige.

Sie können die Varianz $var(X)$ einer Menge numerischer Werte X mit der folgenden Formel berechnen:

$$var(X) = \sum_{x \in X} (x - \bar{x})^2$$

Der Wert \bar{x} ist der Durchschnittswert der Daten in X .

Der Code

Viele Investoren wollen, wenn sie älter werden, das Gesamtrisiko ihres Investitionsportfolios verringern. Laut der führenden Investitionsphilosophie sollte man Aktien mit niedrigerer Varianz wählen, da diese weniger riskant sind. Grob gesagt, Sie können weniger Geld verlieren, wenn Sie in ein stabiles, vorhersagbares und großes Unternehmen investieren, als wenn Sie Ihr Geld in ein kleines Start-up stecken.

Ziel des Einzeilers in Listing 4-7 ist es, in Ihrem Portfolio die Aktien mit minimaler Varianz zu identifizieren. Wenn Sie mehr Geld in solche Aktien investieren, sollte die Gesamtvarianz Ihres Portfolios sinken.

```
## Abhängigkeiten

import numpy as np

## Daten (Zeilen: Aktien / Spalten: Aktienpreise)

X = np.array([[25,27,29,30],

              [1,5,3,2],

              [12,11,8,3],

              [1,1,2,2],

              [2,6,2,2]])

## Einzeiler

# Finde die Aktie mit der kleinsten Varianz

min_row = min([(i,np.var(X[i,:])) for i in range(len(X))],
              key=lambda x:

              x[1])
```

```

## Ergebnis und Ausgabe

print("Zeile mit minimaler Varianz: " + str(min_row[0]))

print("Varianz: " + str(min_row[1]))

```

Listing 4-7 Berechnen der minimalen Varianz in einer einzigen Codezeile

Welche Ausgabe hat dieser Code?

Wie es funktioniert

Wie üblich definieren Sie zuerst die Daten, auf denen Sie den Einzeiler ausführen wollen (siehe oben in Listing 4-7). Das NumPy-Array X enthält fünf Zeilen (eine Zeile pro Aktie in Ihrem Portfolio) mit vier Werten pro Zeile (Aktienpreise).

Ziel ist es, ID und Varianz der Aktie mit der kleinsten Varianz zu finden. Deshalb ist die äußerste Funktion des Einzeilers die Funktion `min()`. Sie führen die `min()`-Funktion auf einer Sequenz aus Tupeln (a, b) aus, wobei der erste Tupel-Wert a der Zeilenindex (Aktienindex) ist und der zweite Tupel-Wert b die Varianz der Zeile.

Sie fragen sich jetzt vielleicht: Welches ist der Minimalwert einer Sequenz aus Tupeln? Natürlich müssen Sie diese Operation richtig definieren, bevor Sie sie benutzen. Dazu nehmen Sie das `key`-Argument der `min()`-Funktion. Das `key`-Argument nimmt eine Funktion entgegen, die bei einem Sequenzwert einen vergleichbaren Objektwert zurückgibt. Unsere Sequenzwerte sind wieder Tupel, und Sie müssen das Tupel mit der minimalen Varianz finden (den zweiten Tupel-Wert). Da die Varianz der zweite Wert ist, geben Sie `x[1]` als Grundlage für den Vergleich zurück. Mit anderen Worten, das Tupel mit dem minimalen zweiten Tupel-Wert gewinnt.

Schauen wir uns an, wie die Sequenz aus Tupel-Werten erzeugt wird. Sie verwenden eine List Comprehension zum Erzeugen eines Tupels für einen Zeilenindex (Aktie). Das erste Tupel-Element ist einfach der Index von Zeile i . Das zweite Tupel-Element ist die Varianz dieser Zeile. Sie verwenden die NumPy-Funktion `var()` zusammen mit Slicing zur Berechnung der Zeilenvarianz.

Das Ergebnis des Einzeilers ist deshalb:

```

"""

```

Zeile mit minimaler Varianz: 3

Variance: 0.25

```
"""
```

Ich möchte noch hinzufügen, dass es eine alternative Methode gibt, das Problem zu lösen. Wäre dies kein Buch über Python-Einzeiler, würde ich die folgende Lösung dem Einzeiler vorziehen:

```
var = np.var(X, axis=1)

min_row = (np.where(var==min(var)), min(var))
```

In der ersten Zeile berechnen Sie die Varianz des NumPy-Arrays `X` entlang den Spalten (`axis=1`). In der zweiten Zeile erzeugen Sie das Tupel. Der erste Tupel-Wert ist der Index des Minimums im Varianz-Array. Der zweite Tupel-Wert ist das Minimum im Varianz-Array. Beachten Sie, dass mehrere Zeilen die gleiche (minimale) Varianz haben können.

Diese Lösung ist besser lesbar. Es gibt also ganz klar einen Konflikt zwischen Knappheit und Lesbarkeit. Nur weil Sie alles in eine einzige Zeile Code stopfen können, sollten Sie das nicht immer und unbedingt tun. Am Ende ist es viel besser, knappen *und* lesbaren Code zu schreiben, statt Ihren Code mit unnötigen Definitionen, Kommentaren oder Zwischenschritten aufzublähen.

Nachdem Sie in diesem Abschnitt die Grundlagen der Varianz kennengelernt haben, sind Sie bereit für die Berechnung einiger einfacher Statistiken.

Einfache Statistiken in einer Zeile

Als Data Scientist und Machine-Learning-Expertin müssen Sie sich mit den Grundlagen der Statistik auskennen. Manche Machine-Learning-Algorithmen basieren völlig auf Statistik (z. B. Bayes'sche Netze).

So ist etwa das Extrahieren grundlegender statistischer Werte aus Matrizen (wie etwa Durchschnitt, Varianz und Standardabweichung) ein wichtiger Bestandteil der Analyse einer Vielzahl von Datenmengen wie Finanzdaten, Gesundheitsdaten oder Daten aus sozialen Medien. Mit dem Aufstieg des

Machine Learning und der Data Science ist es immer wichtiger, zu wissen, wie man NumPy benutzt – das im Herzen der Data Science, Statistiken und linearen Algebra von Python liegt.

In diesem Einzeiler lernen Sie, wie Sie mit NumPy einfache und grundlegende Statistiken berechnen.

Die Grundlagen

In diesem Abschnitt wird erklärt, wie Sie den Durchschnitt, die Standardabweichung und die Varianz entlang einer Achse berechnen. Diese drei Berechnungen sind sehr ähnlich – wenn Sie eine verstehen, dann verstehen Sie auch die anderen.

Sie wollen Folgendes erreichen: Sie haben ein NumPy-Array mit Aktiendaten, in dem die Zeilen die unterschiedlichen Unternehmen und die Spalten deren tägliche Aktienpreise anzeigen, und wollen nun für jedes Unternehmen den Durchschnitt und die Standardabweichung seines Aktienpreises ermitteln (siehe Abbildung 4–23).

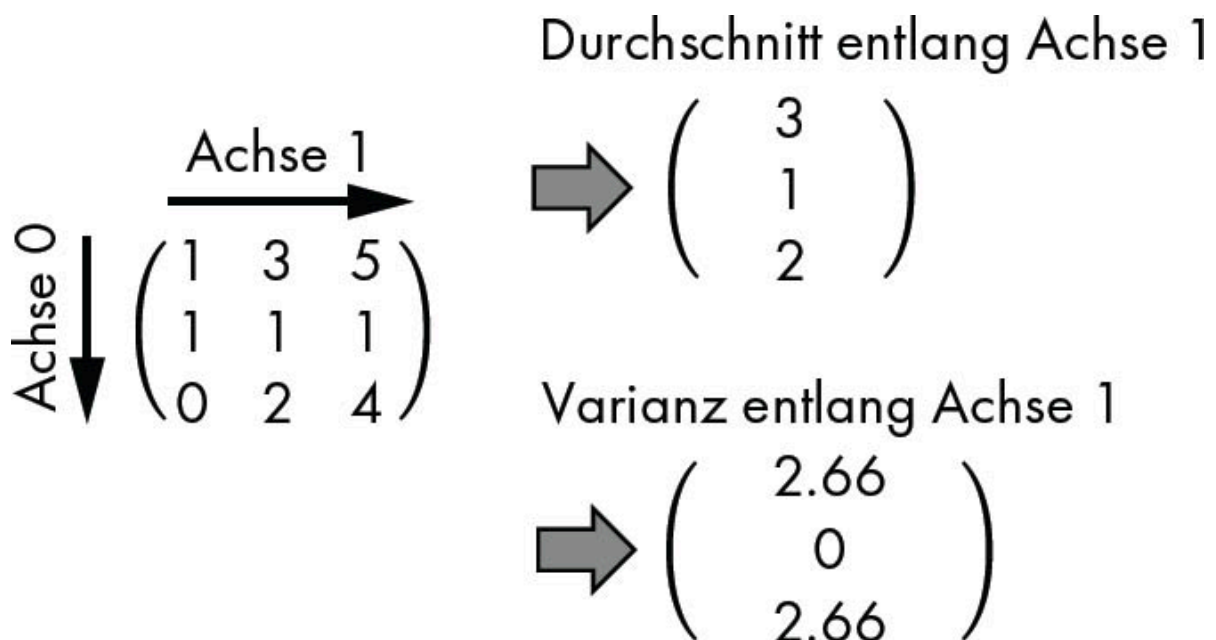


Abb. 4–23 Durchschnitt und Varianz entlang der Achse 1

In diesem Beispiel ist das NumPy-Array zweidimensional, in der Praxis kann es aber viel mehr Dimensionen aufweisen.

Einfacher Durchschnitt, Varianz, Standardabweichung

Bevor wir untersuchen, wie wir das in NumPy erreichen, wollen wir uns das nötige Hintergrundwissen aufbauen. Nehmen wir an, Sie wollen den einfachen

Durchschnitt, die Varianz oder die Standardabweichung über alle Werte in einem NumPy-Array errechnen. Sie haben in diesem Kapitel bereits Beispiele für den Durchschnitt und die Varianz-Funktion gesehen. Die Standardabweichung ist einfach die Quadratwurzel der Varianz. Sie können dies alles leicht mit den folgenden Funktionen ermitteln:

```
import numpy as np

X = np.array([[1, 3, 5],
              [1, 1, 1],
              [0, 2, 4]])

print(np.average(X))

# 2.0

print(np.var(X))

# 2.4444444444444446

print(np.std(X))

# 1.5634719199411433
```

Vermutlich haben Sie gemerkt, dass Sie diese Funktionen auf das zweidimensionale NumPy-Array X anwenden. NumPy reduziert dieses Array einfach und berechnet die Funktionen dann auf dem reduzierten Array. So wird z. B. der einfache Durchschnitt des reduzierten NumPy-Arrays X folgendermaßen berechnet:

$$(1 + 3 + 5 + 1 + 1 + 1 + 0 + 2 + 4) / 9 = 18 / 9 = 2.0$$

Berechnen von Durchschnitt, Varianz, Standardabweichung entlang einer Achse

Manchmal jedoch wollen Sie diese Funktionen entlang einer Achse berechnen. Dazu geben Sie das Schlüsselwort `axis` als Argument zu den entsprechenden Funktionen an (in Kapitel 4 finden Sie eine ausführliche Vorstellung des `axis`-Arguments).

Der Code

Listing 4-8 zeigt Ihnen genau, wie Sie Durchschnitt, Varianz und Standardabweichung entlang einer Achse berechnen. Unser Ziel ist es, diese Werte für alle Aktien in einer zweidimensionalen Matrix zu ermitteln, wobei die Zeilen die Aktien repräsentieren und die Spalten für die täglichen Aktienpreise stehen.

```
## Abhängigkeiten

import numpy as np

## Aktienpreisdaten: 5 Unternehmen

# (Zeile=[Preis_Tag_1, Preis_Tag_2, ...])

x = np.array([[8, 9, 11, 12],
              [1, 2, 2, 1],
              [2, 8, 9, 9],
              [9, 6, 6, 3],
              [3, 3, 3, 3]])

## Einzeiler
```

```

avg, var, std = np.average(x, axis=1), np.var(x, axis=1),
np.std(x, axis=1)

## Ergebnis und Ausgabe

print("Durchschnitte: " + str(avg))

print("Varianzen: " + str(var))

print("Standardabweichungen: " + str(std))

```

Listing 4-8 Berechnen einfacher Statistiken entlang einer Achse

Erraten Sie die Ausgabe des Codes!

Wie es funktioniert

Der Einzeiler verwendet das Schlüsselwort `axis`, um die Achse anzugeben, entlang der die Durchschnitte, Varianzen und Standardabweichungen berechnet werden sollen. Falls Sie z. B. diese drei Funktionen entlang der `axis=1` ausführen, wird jede Zeile zu einem einzigen Wert zusammengefasst. Das resultierende NumPy-Array besitzt also entsprechend eine reduzierte Dimensionalität von eins.

Das Ergebnis sieht folgendermaßen aus:

```

"""

Durchschnitte: [10.  1.5  7.   6.   3. ]

Varianzen: [2.5  0.25 8.5  4.5  0.  ]

Standardabweichungen: [1.58113883 0.5  2.91547595
2.12132034 0.   ]

"""

```

Bevor wir zum nächste Einzeiler kommen, möchte ich Ihnen zeigen, wie Sie diese Idee für ein höher dimensionales NumPy-Array nutzen.

Wenn Sie für höher dimensionale NumPy-Array den Durchschnitt entlang einer Achse berechnen, fassen Sie immer die Achse zusammen, die im `axis`-Argument definiert ist. Hier sehen Sie ein Beispiel:

```
import numpy as np

x = np.array([[[1,2], [1,1]],
              [[1,1], [2,1]],
              [[1,0], [0,0]]])

print(np.average(x, axis=2))

print(np.var(x, axis=2))

print(np.std(x, axis=2))

"""

[[1.5  1.  ]

 [1.   1.5]

 [0.5  0.  ]]

[[0.25  0.  ]

 [0.   0.25]

 [0.25  0.  ]]
```

```
[[0.5 0. ]
```

```
[0.  0.5]
```

```
[0.5 0. ]]
```

```
"""
```

Es gibt drei Beispiele für das Berechnen von Durchschnitt, Varianz und Standardabweichung entlang Achse 2 (siehe Kapitel 5; die innerste Achse). Mit anderen Worten, alle Werte von Achse 2 werden zu einem einzigen Wert kombiniert, der dazu führt, dass Achse 2 aus dem resultierenden Array verschwindet. Schauen Sie sich die drei Beispiele genauer an und versuchen Sie herauszufinden, wie genau Achse 2 zu einem einzigen Durchschnitts-, Varianz- oder Standardabweichungswert zusammengefasst wird.

Eine Vielzahl von Datenmengen (darunter Finanzdaten, Gesundheitsdaten und Daten aus sozialen Medien) verlangen von Ihnen, grundlegende Erkenntnisse aus ihnen zu ziehen. In diesem Abschnitt haben Sie ein tieferes Verständnis darüber gewonnen, wie Sie den leistungsstarken NumPy-Werkzeugkasten benutzen können, um schnell und effizient einfache Statistiken aus mehrdimensionalen Arrays zu gewinnen. Das dient als Vorverarbeitungsschritt für viele Machine-Learning-Algorithmen.

Klassifikation mit Support-Vector Machines in einer Zeile

Support-Vector Machines (SVMs) haben in den letzten Jahren ungemein an Popularität gewonnen, weil sie selbst in hochdimensionalen Räumen eine robuste Klassifikationsleistung bieten. Überraschenderweise funktionieren SVMs sogar dann, wenn es mehr Dimensionen (Features) als Datenelemente gibt. Das ist ungewöhnlich für Klassifikationsalgorithmen wegen des *Fluchs der Dimensionalität*: Mit zunehmender Dimensionalität werden die Daten außerordentlich knapp, sodass die Algorithmen Schwierigkeiten haben, Muster in dem Datensatz zu finden. Das Verständnis der grundsätzlichen Idee der SVMs ist ein entscheidender Schritt auf dem Weg zum wahren Machine-Learning-Experten.

Die Grundlagen

Wie funktionieren Klassifikationsalgorithmen? Sie verwenden die Trainingsdaten, um eine Entscheidungsgrenze zu finden, welche die Daten in der einen Klasse von den Daten in der anderen Klasse trennt (in »Logistische Regression in einer Zeile« auf Seite 104 wäre die Entscheidungsgrenze, ob die Wahrscheinlichkeit der Sigmoid-Funktion unter oder über der Schwelle von 0,5 liegt).

Ein Überblick über die Klassifikation

Abbildung 4–24 zeigt ein Beispiel für einen allgemeinen Klassifikator.

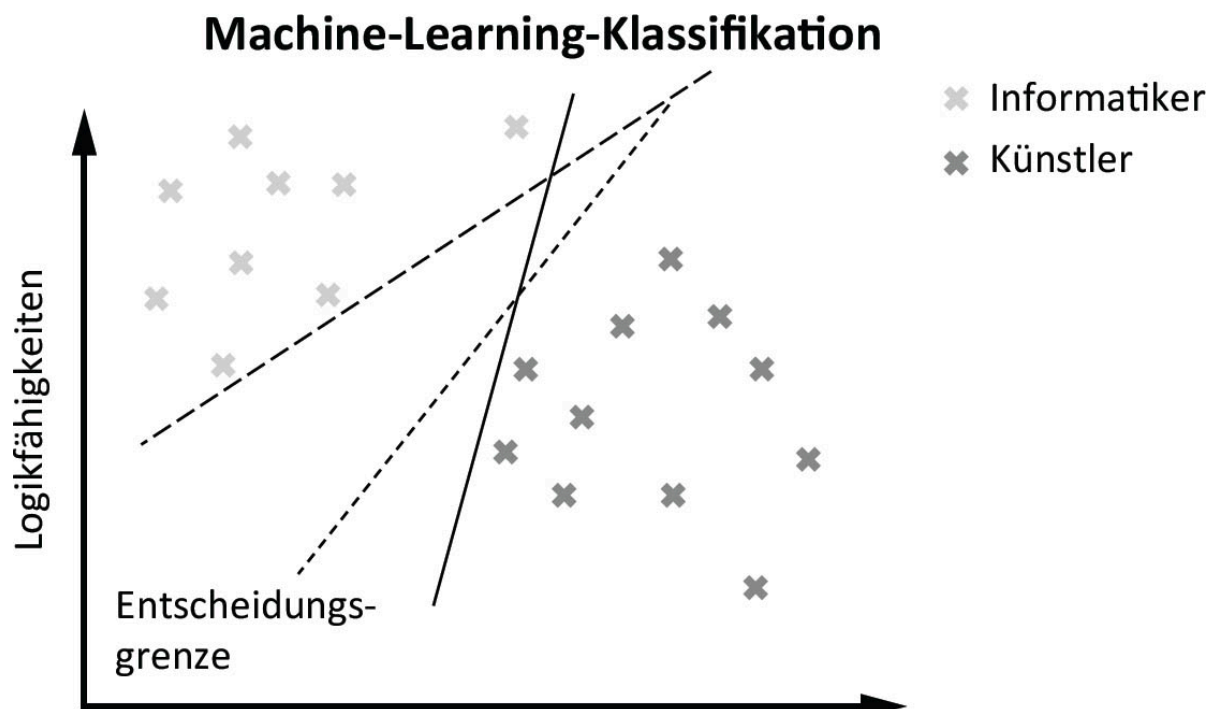


Abb. 4–24 *Verschiedenartige Fertigkeiten von Informatikern und Künstlern*

Nehmen Sie einmal an, Sie wollen ein Empfehlungssystem für angehende Universitätsstudenten bauen. Die Abbildung visualisiert die Trainingsdaten, die aus Benutzern bestehen, die entsprechend ihrer Fertigkeiten in zwei Bereiche unterteilt wurden: Logik und Kreativität. Manche Menschen sind sehr Logikorientiert und haben relativ wenig Kreativität; andere dagegen haben sehr viel Kreativität, aber keinen Sinn für Logik. Die erste Gruppe ist mit *Informatiker* gekennzeichnet, die zweite Gruppe mit *Künstler*.

Um neue Benutzer zu klassifizieren, muss das Machine-Learning-Modell eine Entscheidungsgrenze finden, die die Informatiker von den Künstlern trennt. Grob gesagt, klassifizieren Sie einen Benutzer entsprechend seiner Position zur Entscheidungsgrenze. Im Beispiel werden Benutzer, die im linken Bereich

landen, zu den Informatikern gezählt und Benutzer im rechten Bereich zu den Künstlern.

Im zweidimensionalen Raum ist die Entscheidungsgrenze entweder eine Gerade oder eine Kurve (höherer Ordnung). Eine Gerade wird als *linearer Klassifikator* bezeichnet, eine Kurve dagegen als *nichtlinearer Klassifikator*. In diesem Abschnitt befassen wir uns nur mit linearen Klassifikatoren.

Abbildung 4–24 zeigt drei Entscheidungsgrenzen, die alle gültige Datentrenner sind. Es ist in unserem Beispiel nicht möglich, zu quantifizieren, welche der angegebenen Entscheidungsgrenzen besser ist; sie alle führen zu perfekter Genauigkeit beim Klassifizieren der Trainingsdaten.

Aber welches ist die beste Entscheidungsgrenze?

Support-Vector Machines liefern eine eindeutige und wunderbare Antwort auf diese Frage. Die beste Entscheidungsgrenze bietet wohl einen maximalen Sicherheitsspielraum. Mit anderen Worten, SVMs maximieren den Abstand zwischen den nächstgelegenen Datenpunkten und der Entscheidungsgrenze. Ziel ist es, den Fehler neuer Punkte zu minimieren, die nahe an der Entscheidungsgrenze liegen.

Abbildung 4–25 zeigt ein Beispiel.

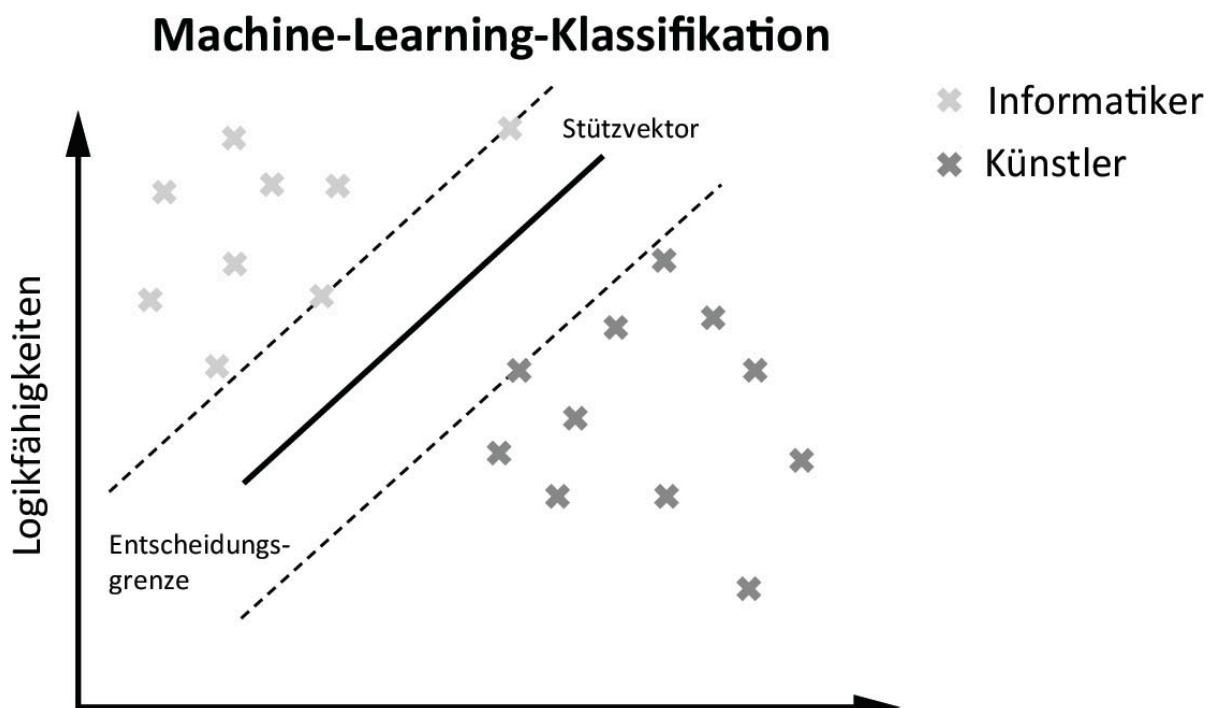


Abb. 4–25 Support-Vector Machines maximieren die Fehlerspanne.

Der SVM-Klassifikator wählt die Stützvektoren so, dass die Zone zwischen den Stützvektoren so dick wie möglich wird. Hier sind die Stützvektoren die

Datenpunkte, die auf den zwei gepunkteten Linien liegen, die parallel zur Entscheidungsgrenze verlaufen. Diese Linien werden als *Margins* (Ränder) bezeichnet. Die Entscheidungsgrenze ist die Linie in der Mitte mit dem maximalen Abstand zu den Rändern. Da die Zone zwischen den Rändern und der Entscheidungsgrenze maximiert wird, soll die *Fehlerspanne* maximal sein, wenn neue Datenpunkte klassifiziert werden. Für viele praktische Probleme kann also eine hohe Klassifizierungsgenauigkeit erreicht werden.

Der Code

Ist es möglich, in nur einer Zeile Python-Code Ihre eigene SVM herzustellen? Schauen Sie sich Listing 4–9 an.

```
## Abhängigkeiten

from sklearn import svm

import numpy as np

## Daten: Studentenleistungen in (Mathe, Sprachen,
Kreativität) -> Studien-

fach

X = np.array([[9, 5, 6, "Informatik"],
              [10, 1, 2, "Informatik"],
              [1, 8, 1, "Literatur"],
              [4, 9, 3, "Literatur"],
              [0, 1, 10, "Kunst"],
              [5, 7, 9, "Kunst"]])
```

```

## Einzeiler

svm = svm.SVC().X[:, :-1], X[:, -1])

## Ergebnis und Ausgabe

student_0 = svm.predict([[3, 3, 6]])

print(student_0)

student_1 = svm.predict([[8, 1, 1]])

print(student_1)

```

Listing 4-9 SVM-Klassifikation in einer einzigen Zeile Code

Raten Sie, welche Ausgabe dieser Code liefert.

Wie es funktioniert

Der Code demonstriert, wie Sie Support-Vector Machines in Python in der einfachsten Form benutzen können. Das NumPy-Array enthält die benannten Trainingsdaten mit einer Zeile pro Benutzer und einer Spalte pro Feature (Fertigkeiten in Mathe, Sprachen und Kreativität). Die letzte Spalte ist das Label (die Klasse).

Da Sie dreidimensionale Daten haben, separiert die Support-Vector Machine die Daten mithilfe zweidimensionaler Ebenen (lineare Trennung) statt mit eindimensionalen Geraden. Es ist also auch möglich, drei statt (wie in den früheren Beispielen) zwei Klassen voneinander zu trennen.

Der Einzeiler ist recht einfach: Sie erzeugen zuerst das Modell mit dem Konstruktor der `svm.SVC`-Klasse (*SVC* bedeutet *Support-Vector Classification*). Dann rufen Sie die Funktion `fit()` auf, um das Training anhand Ihrer benannten Trainingsdaten durchzuführen.

Im Ergebnisteil des Codeschnipsels rufen Sie die `predict()`-Funktion auf den neuen Beobachtungen auf. Da die Fertigkeiten von `student_0` mit Mathe = 3, Sprachen = 3 und Kreativität = 6 angegeben sind, sagt die Support-Vector Machine vorher, dass das Label *Kunst* zu dem Studenten passt. Die Fertigkeiten

von `student_1` sind mit `Mathe = 8`, `Sprachen = 1` und `Kreativität = 1` angegeben. Entsprechend gibt die Support-Vector Machine das Label *Informatik* aus.

Hier ist die fertige Ausgabe des Einzeilers:

```
## Ergebnis und Ausgabe

student_0 = svm.predict([[3, 3, 6]])

print(student_0)

# ['Kunst']

student_1 = svm.predict([[8, 1, 1]])

print(student_1)

## ['Informatik']
```

SVMs arbeiten also auch dann gut in hochdimensionalen Räumen, wenn es mehr Features als Trainingsdatenvektoren gibt. Die Idee des Maximierens des *Sicherheitsspielraums* ist intuitiv und führt zu einer soliden Leistung, wenn *Grenzfälle* klassifiziert werden sollen – d. h. Vektoren, die in den Sicherheitsbereich fallen. Im letzten Abschnitt dieses Kapitels treten wir einen Schritt zurück und schauen uns einen Meta-Algorithmus für die Klassifikation an: Ensemble Learning mit Random Forests.

Klassifikation mit Random Forests in einer Zeile

Kommen wir zu einer aufregenden Machine-Learning-Technik: *Ensemble Learning*. Hier ist mein Tipp, wenn Ihre Vorhersagegenauigkeit zu wünschen übriglässt, Sie aber um jeden Preis die Deadline schaffen müssen: Probieren Sie diesen Meta-Lern-Ansatz, der die Vorhersagen (oder Klassifikationen) mehrerer Machine-Learning-Algorithmen miteinander kombiniert. In vielen Fällen erhalten Sie in letzter Minute bessere Ergebnisse.

Die Grundlagen

In den vorangegangenen Abschnitten haben Sie verschiedene Machine-Learning-Algorithmen kennengelernt, mit denen Sie schnelle Ergebnisse erzielen. Unterschiedliche Algorithmen besitzen jedoch unterschiedliche Stärken. Zum Beispiel können Klassifikatoren aus neuronalen Netzwerken ausgezeichnete Ergebnisse für komplexe Probleme erzielen. Sie neigen allerdings auch zur Überanpassung der Daten, weil sie besonders gut darin sind, sich feinste Datenmuster zu merken. Ensemble Learning für Klassifikationsprobleme überwindet teilweise das Problem, dass Sie nicht schon vorher wissen, welche Machine-Learning-Technik am besten funktioniert.

Wie geht das? Sie erzeugen einen Meta-Klassifikator, der aus mehreren Typen oder Instanzen von fundamentalen Machine-Learning-Algorithmen besteht. Mit anderen Worten, Sie trainieren mehrere Modelle. Um eine einzelne Beobachtung zu klassifizieren, bitten Sie alle Modelle, die Eingabe unabhängig voneinander zu klassifizieren. Anschließend geben Sie die Klasse, die für Ihre Eingabe am häufigsten zurückgeliefert wurde, als *Meta-Vorhersage* aus. Dies ist dann die Ausgabe Ihres Ensemble-Learning-Algorithmus.

Random Forests sind eine Sonderform der Ensemble-Learning-Algorithmen. Sie konzentrieren sich auf Decision-Tree Learning. Ein Wald besteht aus vielen Bäumen. In vergleichbarer Weise besteht ein Random Forest aus vielen Entscheidungsbäumen. Jeder Entscheidungsbaum soll während der Trainingsphase einen Aspekt von Zufälligkeit in die Prozedur der Baumgenerierung einbringen (z. B. welcher Baumknoten zuerst gewählt werden soll). Dies führt zu verschiedenen Entscheidungsbäumen – genau das, was Sie wollen.

Abbildung 4–26 zeigt, wie die Vorhersage im folgenden Szenario für einen trainierten Random Forest funktioniert. Alice ist sehr gut in Mathematik und Sprachen. Das *Ensemble* besteht aus drei Entscheidungsbäumen (die einen Random Forest bilden). Um Alice zu klassifizieren, wird jeder Entscheidungsbaum nach Alices Klassifikation gefragt. Zwei der Entscheidungsbäume klassifizieren Alice als Informatikerin. Da dies die Klasse mit den meisten Stimmen ist, wird sie als Ergebnis der Klassifikation zurückgegeben.

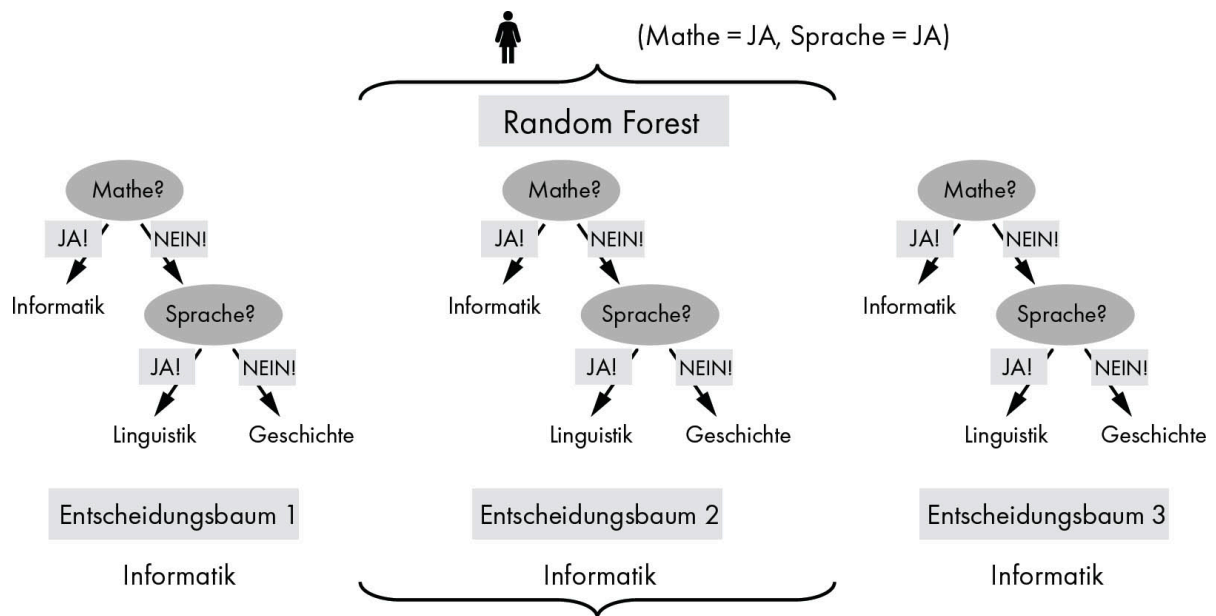


Abb. 4-26 Der Random-Forest-Klassifikator fasst die Ausgaben der drei Entscheidungsbäume zusammen.

Der Code

Bleiben wir bei dem Beispiel der Studienfachzuordnung aufgrund der Fähigkeiten eines Studenten in drei Gebieten (Mathematik, Sprachen, Kreativität). Sie glauben vielleicht, dass es kompliziert sei, eine Ensemble-Learning-Methode in Python zu implementieren. Dank der umfangreichen scikit-learn-Bibliothek ist es das jedoch nicht (siehe Listing 4-10).

```
## Abhängigkeiten

import numpy as np

from sklearn.ensemble import RandomForestClassifier

## Daten: Studentenleistungen in (Mathe, Sprachen,
Kreativität) -> Studien-

fach

X = np.array([[9, 5, 6, "Informatik"],
```

```

        [5, 1, 5, "Informatik"],

        [8, 8, 8, "Informatik"],

        [1, 10, 7, "Literatur"],

        [1, 8, 1, "Literatur"],

        [5, 7, 9, "Kunst"],

        [1, 1, 6, "Kunst"]])

## Einzeiler

Forest =
RandomForestClassifier(n_estimators=10).fit(X[:, :-1],
X[:, -1])

## Ergebnis

students = Forest.predict([[8, 6, 5],

                            [3, 7, 9],

                            [2, 2, 1]])

print(students)

```

Listing 4-10 Ensemble Learning mit Random-Forest-Klassifikatoren

Raten Sie einmal: Welche Ausgabe hat dieser Code?

Wie es funktioniert

Nach dem Initialisieren der benannten Trainingsdaten in Listing 4-10 erzeugt der Code einen Random Forest, indem er den Konstruktor auf der Klasse

`RandomForestClassifier` einsetzt. Dabei definiert der Parameter `n_estimators` die Anzahl der Bäume im Forest. Als Nächstes füllen Sie das Modell, das aus der vorherigen Initialisierung stammt (einen leeren Forest) durch den Aufruf der Funktion `fit()`. Dazu bestehen die Eingangstrainingsdaten aus allen Spalten des Array `X` mit Ausnahme der letzten Spalte, während die Label der Trainingsdaten in dieser letzten Spalte definiert sind. Wie in den vorherigen Beispielen extrahieren Sie die entsprechenden Spalten mittels Slicing aus dem Daten-Array `X`.

Der Klassifikationsteil ist in diesem Codeschnipsel ein bisschen anders. Ich wollte Ihnen zeigen, wie Sie statt einer gleich mehrere Beobachtungen klassifizieren können. Dazu erzeugen Sie ein mehrdimensionales Array mit einer Zeile pro Beobachtung.

Hier folgt die Ausgabe des Codes:

```
## Ergebnis

students = Forest.predict([[8, 6, 5],
                            [3, 7, 9],
                            [2, 2, 1]])

print(students)

# ['Informatik' 'Kunst' 'Kunst']
```

Beachten Sie, dass das Ergebnis auch hier immer noch nicht deterministisch ist (d. h., bei mehrfachem Ausführen des Codes können sich unterschiedliche Ergebnisse zeigen), weil der Random-Forest-Algorithmus von dem Zufallszahlengenerator abhängt, der zu verschiedenen Zeitpunkten unterschiedliche Zahlen zurückliefert. Sie können diesen Aufruf deterministisch machen, indem Sie das Integer-Argument `random_state` benutzen. Sie können z. B. `random_state=1` setzen, wenn Sie den Random-Forest-Konstruktor aufrufen: `RandomForestClassifier(n_estimators=10, random_state=1)`. In diesem Fall erhalten Sie jedes Mal, wenn Sie einen neuen Random-Forest-Klassifikator erzeugen, dieselben Ergebnisse, weil immer

dieselben Zufallszahlen generiert werden: Sie alle basieren auf dem Seed-Integer 1.

In diesem Abschnitt habe ich Ihnen einen Meta-Ansatz für die Klassifikation vorgestellt: Die Ausgabe verschiedener Entscheidungsbäume dient der Reduzierung der Varianz des Klassifikationsfehlers. Dies ist eine Version des Ensemble Learning, das mehrere Grundmodelle zu einem Meta-Modell kombiniert, das in der Lage ist, deren jeweilige Stärken auszunutzen.



Hinweis

Zwei unterschiedliche Entscheidungsbäume können zu einer hohen Varianz des Fehlers führen: Der eine generiert gute Ergebnisse, während der andere es nicht tut. Durch Random Forests schwächen Sie diesen Effekt ab.

Variationen dieser Idee sind beim Machine Learning gebräuchlich – und falls Sie schnell Ihre Vorhersagegenauigkeit verbessern wollen, führen Sie einfach mehrere Machine-Learning-Modelle aus und werten deren Ergebnisse aus, um das beste zu finden (ein kleines Geheimnis von Machine-Learning-Anwendern). In einer gewissen Weise führt Ensemble Learning automatisch die Aufgabe aus, die Experten oft in praktischen Machine-Learning-Pipelines erledigen: Das Auswählen, Vergleichen und Kombinieren der Ausgaben unterschiedlicher Machine-Learning-Modelle. Die große Stärke des Ensemble Learning liegt darin, dass dies individuell für jeden Datenwert zur Laufzeit erledigt werden kann.

Zusammenfassung

Dieses Kapitel behandelte 10 grundlegende Machine-Learning-Algorithmen, die entscheidend sind für Ihren Erfolg auf diesem Gebiet. Sie haben Regressionsalgorithmen zum Vorhersagen von Werten kennengelernt, wie die lineare Regression, KNNs und neuronale Netzwerke. Sie haben etwas über Klassifikationsalgorithmen gelernt, wie die logistische Regression, Decision-Tree Learning, SVMs und Random Forests. Darüber hinaus haben Sie gelernt, wie Sie einfache Statistiken mehrdimensionaler Daten-Arrays berechnen und wie Sie den k-Means-Algorithmus für Unsupervised Learning verwenden. Diese Algorithmen und Methoden gehören zu den wichtigsten im Bereich des Machine Learning, und es gibt noch viele mehr, die Sie studieren sollten, wenn Sie ernsthaft auf diesem Gebiet arbeiten wollen. Dieses Lernen zahlt sich aus – in den USA verdienen Machine-Learning-Experten üblicherweise sechsstelligen Gehälter (eine einfache Suche im Web sollte das bestätigen)! Studenten, die tiefer in das Machine Learning einsteigen wollen, empfehle ich den

ausgezeichneten (und kostenlosen) Coursera-Kurs von Andrew Ng. Sie finden das Kursmaterial online. Fragen Sie einfach Ihre Liebessuchmaschine.

Im nächsten Kapitel befassen Sie sich mit einer der wichtigsten (und am meisten unterschätzten) Fertigkeiten hocheffizienter Programmierer: reguläre Ausdrücke. Während sich dieses Kapitel eher auf der konzeptuellen Seite bewegt (Sie haben die allgemeinen Ideen kennengelernt, während die eigentliche Arbeit von der scikit-learn-Bibliothek gestemmt wurde), wird das nächste Kapitel ausgesprochen technisch sein. Krempeln Sie also die Ärmel hoch und lesen Sie weiter!

Index

Symbole

- (^)-Nicht-(Not)-Regex-Operator 166, 173–174
- (+)-Operator
 - Addition 2, 51
 - Verkettung 8, 195–196
 - Wenigstens-Eins-Regex 159
- (|)-Operator
 - Oder-Regex- 159, 171
 - Vereinigungs- (Union-) 195
- (") doppelte Anführungszeichen 5
- ('') dreifache Anführungszeichen 5
- (''') dreifache Anführungszeichen 5
- (') einfache Anführungszeichen 5
- (\)-Escape-Präfix 163, 165, 167, 173
- (\)-Gruppen-Regex-Operator 158–160
- { }-(Instances)-Regex-Operator 191–194
- (/)-Integer-Division-Operator 2
- (%)-Modulo-Operator 2, 198
- (_) nachgestellter Unterstrich 116
- (?!)-negativer-Vorgriff-Regex-Operator 176
- (*?)-nicht-gieriger Asterisk-Regex-Operator 159
- (\n)-Newline-Zeichen 5, 26, 27, 154
- (?)-Null-oder-Eins-Regex 154, 155, 159, 165
- (-)-Operator
 - Negation 2
 - Subtraktion 2, 51
- []-Operator
 - Indizierung 55
 - Listenerzeugung 7
- (*)-Operator
 - Asterisk-Regex 153–154, 156

- Entpacken 46
- Multiplikation 2, 51, 54, 59
- Replikation 42–43
- (/)-Operator 2, 51
- (?P)-benannte-Gruppe-Regex-Operator 172–173
- (**)-Power-Operator 2
- (.)-Punkt-Regex-Operator 153, 158–160
- (\s)-Whitespace-Zeichen (Leerraum) 5, 172–175
- (_)-Throwaway-Parameter 208
- (\t)-Tabulator-Zeichen 5

A

- abs()-Funktion 2, 85
- Absolutwert 82, 85
- Absprungrate 82
- Addition-(+)-Operator 2, 51
- Air-Quality-Index-(AQI)-Ausreißer-Beispiel 64–67
- Aktivierungsfunktion 127
- Algorithmen
 - Ausreißer-Erkennung 82, 86–87
 - binäre Suche 209
 - Clusteranalyse 111–114
 - Fibonacci-Folgen 207–209
 - Laufzeitkomplexität 182, 183, 210
 - Levenshtein-Distanz 189–192
 - lineare Regression 97–103
 - Palindrom-Erkennung 183–184
 - Permutationen berechnen 185–187
 - Potenzmenge erzeugen 193–196
 - Primzahlengenerierung 200–207
 - Quicksort 213–215
 - rekursive 186–188
 - und Programmiermeisterschaft 179–180
 - Verschleierung 197–200
- all() 90
- Anagramm-Erkennungs-Beispiel 180–182

- and (Schlüsselwort) 3–4
- Angestellten-Daten-Beispiel
 - arithmetisch 54
 - Clusteranalyse 114–116
 - Dictionaries 22, 43–44, 46
- any()-Funktion 44–45
- append()-Listen-Methode 10, 209
- arange()-Funktion 103
- argsort()-Funktion 76–77, 78–79
- Arithmetische Operationen 2
- Arrays. *Siehe* NumPy-Arrays
- Assoziationsanalyse 87–88
- Asterisk(*)-Regex-Operator 156
- astype()-Funktion 70, 80
- Ausdruck, in Listen-Abstraktion 14, 22–24
- Ausreißer-Erkennung 64–67, 83, 86–87
- Autokorrekturfunktion 189
- average()-Funktion 52, 66–67, 74, 139
- axis-Argument 73, 74, 77, 87, 89, 90, 140

B

- benannte Gruppen 172–173
- benannte vs. unbenannte Daten 111–112
- Benutzereingabenprüfungsbeispiel 167–170
- Bestseller-Filter-Beispiele 80–82
- Bestseller-Paket-Assoziationsbeispiel 90–92
- Bias-Variance Trade-off 134–135
- binärer Suchalgorithmus 209, 210, 212
- boolesche Daten
 - als NumPy-Array-Datentyp 60
 - Array-Operationen 64, 66
 - Werte und Evaluation 2–4, 68
- boolesche Indizierung 68–70, 81
- Bounce Rate 82
- break-Schlüsselwort 17
- Broadcasting 62–63

Beispiele 59–60, 62–63, 65–66

Definition 57, 60

C

Caesar-Verschlüsselung 196

Centroid. *Siehe* Schwerpunkt

Christmas-Zitat-Beispiel 160

close()-Befehl (Datei) 28

Clusteranalyse-Algorithmus 111–114

cluster_centers_-Attribut 114–116

continue-Schlüsselwort 17

D

Datei-Lese-Beispiel 27–29

Datentypen

- Boolean 2–4

- hash-fähige 11–12

- None-Schlüsselwort 4–6

- numerische 2

- Sammlung 11–12

- Strings 5–6

- und NumPy-Arrays 60–61, 63, 68

DecisionTreeClassifier-Modul 133–134

Dictionaries

- Angestellten-Daten-Beispiel 22, 43–44, 46

- Datenstruktur 12–13

Dimensionalität

- Fluch der 141

- und NumPy-Arrays 50–52, 55–56

Division-(/)-Operator 2, 51

doppelte Anführungszeichen (") 5

dreifache Anführungszeichen (') und (""") 5

dtype-property 64, 66

Duplikate-Erkennungsbeispiel 171–173

E

eckige Klammern ([])

 Bereichsdefinitions-Regex-Operator 166–167

 Dimensionalität eines NumPy-Arrays 51

 Indizierungsoperator 55

 Operator zur Listenerzeugung 7

Eckige-Klammer-Umgebung [] 163

Editierdistanz 189. *Siehe auch* Levenshtein-Distanz

einfache Anführungszeichen (') 5

Einkommensberechnungsbeispiel 53–54

elementweise Operationen 51, 62

elif-Schlüsselwort 15

else-Schlüsselwort 15

endwith()-String-Methode 5

Ensemble Learning 145–148

Entpacken-Operator (*) 46

Entscheidungsbäume 131–134, 146

erweiterte Indizierung 67, 79

Escape-Präfix (\) 163, 165, 167, 173

extend()-Listen-Methode 8

F

Fakultätsberechnungsbeispiel 185–186

falsch positiv 157

False-Werte. *Siehe auch* boolesche Daten

 und while-Schleifen 16–17

 von Python-Objekten 190–191

Features und Vorhersagen 96

Fehlerminimierung 100–101, 103

Fehlerspanne 143

Fibonacci-Folgen-Algorithmus 207–209

FIFO (First-in-First-out)-Struktur 10–11

Filterung 63–64, 80–81, 86–87. *Siehe auch* Assoziationsanalyse

findall()-Funktion 152, 153–155, 160–163, 164, 167, 168, 173–174

find()-String-Methode 5

Finxter-Bewertungen 123–124, 127–128

fit()-Funktion

und Entscheidungsbäume 133–134

und K-Nearest-Neighbors-Algorithmus 119–120

und lineare Regression 102–104

und logistische Regression 108

und neuronale Netzwerkanalyse 128–129

und Random Forests 147–148

und Support-Vector Machines 144

Float-Datentyp und -Operationen 2, 60

float()-Funktion 2

for-Schleifen 19, 23

fullmatch()-Funktion 168–169, 170

functools-Bibliothek 194

Funktionen. *Siehe auch* individuelle Funktions-Bezeichnungen; Lambda-Funktionen

Definition 17–18

G

Generatordrucke 43, 43–44

gierige («greedy») Mustererkennung 154–155, 156

Grenzfälle 145

Gruppen-()-Regex-Operator 158–159, 160, 163–164

H

Hadamard-Produkt 54

hash-fähige Datentypen 11–12

hash()-Funktion 11, 12

Hauspreis-Beispiel 118–121

Herzzyklusdaten-Beispiel 40–42

Histogramming 183

Histogramm zeichnen 84–87

Hyperlink-Analyse-Beispiel 161–162

I

if-Schlüsselwort 14, 15

Index

- als Argumente 32
- erweiterte Indizierung 67, 79
- und argsort()-Funktion 76–78
- und boolesche Arrays 68–70, 81

index()-Listen-Methode 10

Inferenzphase 96

initializer-Argument 194–195

Inkrementor-Funktion 19

in-Schlüsselwort 13, 14

insert()-Listenmethode 8

Instagram-Influencer-Filter-Beispiel 69–70

Instances-Regex-Operator ({}) 191

Integer-Datentyp und -Operationen 2, 60

Integer-Division-Operator (//) 2

int()-Funktion 2

Investitionsportfoliorisikobeispiel 135–137

is-Schlüsselwort 7

items()-Dictionary-Methode 13, 22

iterable (reduce()-Argumente 194–195

J

join()-String-Methode 5, 198

K

kategorische Ausgabe 105

keys()-Funktion 13

Klassen-Label 109

Klassifikator 142

Klassifizierungs-Algorithmen

- Entscheidungsbäume 131–133
- Fluch der Dimensionalität 141
- Klassifikationsprobleme 104
- k Nearest Neighbors (KNN) 117–121
- Konzepte 142
- logistische Regression 104–109
- Support-Vector Machines (SVMs) 141, 143–145

k-Means-Algorithmus 111–114
KMeans-Modul 114–116
k-Nearest-Neighbors-(KNN)-Algorithmus 117–121
KNeighborsClassifier-Modul 121
KNeighborsRegressor-Modul 121–123
Koeffizienten 104–107
kollaboratives Filtern 88–92

L

Lambda-Funktionen
 Definition 18–19, 29–31
 rekursive 188–189
lambda-Schlüsselwort 18
Laufzeitkomplexität 182, 183, 210
len()-Funktion 7
len()-String-Methode 6
Levenshtein-Distanz-Algorithmus 189
lineare Regression 97–105
 codieren 101–103
 Konzepte und Formeln 97–99
linearer Klassifikator 142
LinearRegression-Modul 101
List Comprehension
 Beispiele 25–27
 Formel 14, 22–24
 und Generatorausdrücke 43
 und Slicing 35–36
Listen. *Siehe auch* List Comprehension
 Definition 7
 Listenverkettung 8–9, 40–41, 195. Operationen 8–10
 vs. NumPy-Arrays 51–52
logische Und-Operation 85–86
LogisticRegression-Modul 108–109
logistische Regression 104–108
lower()-String-Methode 5
Lungenkrebs-logistische-Regression-Beispiel 105–109

M

Machine Learning

- Ensemble Learning 145–148
 - Entscheidungsbäume 131–133
 - Klassifikationskonzepte 142
 - k-Means-Cluster-Algorithmus 111–115
 - k-Nearest-Neighbors-(KNN)-Algorithmus 117–121
 - Lineare-Regressions-Algorithmus 97–102
 - Logistische-Regressions-Algorithmus 104–108
 - Modellparameter 98
 - Neuronale Netzwerkanalyse 123–127
 - Supervised 96–97
 - Support-Vector Machines 141, 143–145
 - Überblick 95, 149
 - Unsupervised 111–112
 - Varianz 134
 - Verzerrung-Varianz-Dilemma 134–135
- map()-Funktion 18–19
- Margins (Ränder) 143
- match()-Funktion 158–159, 160–161
- Matplotlib-Bibliothek 41, 84–85
- max()-Funktion 52–53, 54, 91
- Maximum-Likelihood-Modell 107–108
- max_iter-Argument 128–129
- mean 84, 86
- mean()-Funktion 86
- mehrlagiges Perzeptron. *Siehe* Multilayer Perceptron (MLP)
- mehrzeilige Strings 5
- Meta-Vorhersage 146
- Mindestlohnprüfungsbeispiel 43–45
- min()-Funktion 52, 136
- MLPRegressor-Modul 127–129
- Modulo-Operator (%) 2, 198
- Multilayer Perceptron (MLP) 122–127
- multinomiale Klassifizierung 106

Multiplikation-(*)-Operator 2, 51, 54, 59

Multiplikation von Arrays 51, 59

Multiset-Datenstruktur 12

Mutability. *Siehe* Veränderbarkeit

N

nachgestellter Unterstrich () 116

n_clusters-Argument 114

ndim-Attribut 58–59

Negationsoperator (-) 2

negativer Vorgriff 176–177

negativer-Vorgriff-Regex-Operator (?!) 176

Neuronale Netzwerke (Neural Networks)

- Beispiele 122–123

- Konzepte künstlicher 125–126

- programmieren 127–130

Newline-Zeichen (\n) 5, 26, 27, 154

nicht-gierige («non-greedy») Mustererkennung 154–155

nicht-gieriger Asterisk(*)-Regex-Operator 154–155

None-Schlüsselwort 6, 7

nonzero()-Funktion 64–66

normal 64–66

normal()-Funktion 84

not (schlüsselwort) 3–4

Null-oder-Eins-Regex (?) 154, 155, 159, 165

numerische Datentypen und Operationen 2

NumPy-Arrays

- Achsen und Dimensionalität 55–57

- arithmetische Operationen auf 51–54

- axis-Argument 73–74, 77–78, 89

- Berechnen der minimalen Varianz 134–136

- boolesche Operationen 64–66

- Broadcasting 59, 64–65

- erzeugen 51–52

- Filter 80–81

- Indizierung 55, 68–70

Slicing 56–58, 78
Sortieren 75–78
statistische Berechnungen 137–140
und Datentypen 60–61, 63, 70
NumPy-Bibliothek 50, 54, 87

O

Oder-Regex-Operator (|) 159, 171
or (Schlüsselwort) 3–4

P

Palindrom-Erkennungsbeispiel 183–184
Pandas-Bibliothek 32, 49
Permutations-Erkennungsbeispiel 185–187
Pivotelement 214–216
plot()-Funktion 41–42
pop()-Listen-Methode 10
Potenzmengen 193–196
Power-Operator (**) 2
predict(128–129
predict()-Funktion 102, 119–121, 130
predict_proba()-Funktion 110
Primzahlen
 Code-Beispiel 201–207
 Erkennungsbeispiel 199–201
Primzahlenfilterbeispiel 202–206
Pruning 132
Punkt-(.)-Regex 153, 158–160

Q

Quicksort-Algorithmus 213–215

R

RandomForestClassifier-Modul 147
Random Forests 146–149
random-Modul 84
random_state-Parameter 148

range()-Funktion 14, 23, 200–201, 206
reduce()-Funktion 194–196, 201, 206, 207–208
Regex-Funktionen 160, 168
 Entfernen von falsch positiven Ergebnissen 157–159
Regex-Zeichen 153–155, 157, 158, 159–160, 163
Regressionsprobleme
 und K-Nearest-Neighbors-Algorithmus 117–118
 und linearer-Regressions-Algorithmus 97
 vs. Klassifikationsprobleme 104
Reguläre Ausdrücke. *Siehe auch* Regex-Funktionen; Regex-Zeichen
 compiled patterns 158–159
 gierige und nicht-gierige Mustererkennung 154
 Gruppen und benannte Gruppen 165–166, 174–175
 negativer Vorgriff 176–177
 zur Benutzereingabenprüfung 171–175
 zur Doppelzeichenerkennung 173
 zur Wortwiederholungserkennung 174–175
 zur Zeichenersetzung 176–177
Rekursion und rekursive Funktionen 186–188, 209–211, 213–215
re-Modul 152–154
remove()-Listen-Methode 9
replace()-String-Methode 5
Replikationsoperator (*) 42–43
reshape()-Funktion 72–73, 102–103, 109, 119–121
return-Schlüsselwort 18
reverse()-Listen-Methode 9
ROT13-Algorithmus 197–200
Rückgabewert 7, 14, 29, 209

S

Sammlungs-Datentypen 11–12
SAT-Ergebnis-Analyse-Beispiel 78–79
Schleifen 15–16
(Schlüssel, Wert)-Paare 12
Schwerpunkt 112
scikit-learn-Bibliothek 32, 49, 101–102, 114

search()-Funktion 160, 168

Set

 Datenstruktur 10

 Potenzmengenberechnungsbeispiel 193–195

 Überprüfen von Zugehörigkeiten 14

Set Comprehension 14

shape-Attribut 57–59, 84

Sicherheitsspielraum 145

Sieb des Eratosthenes 201–206

Sigmoid-Funktion 105–107

sklearn-Paket 114

Slicing

 mehrdimensionales 56–58

 mit negativer Schrittgröße 78, 184

 Syntax und Beispiele 32–35

 und List Comprehension 35–36

Softmax-Funktion 106

sorted()-Python-Funktion 76, 77, 182

Sortieren 75–78, 182, 214–215

sort()-Listen-Methode 9

sort()-NumPy-Funktion 75–76, 77

split()-Funktion 26

Stack Overflow 202

Stacks 10

start-Argument 32, 183

startswith()-String-Methode 5

statistische Berechnungen 137–140

step-Argument 32, 183

stop-Argument 32, 183

Strings. *Siehe auch* mehrzeilige Strings

 ausgewählte Methoden 5–6

 Datentypen 6

strip()-String-Methode 5, 27–29

str()-String-Methode 5

sub()-Regex-Funktion 178

Subtraktion(-)-Operator 2, 51
sum()-Funktion 91, 92
Supervised Machine Learning 96–97
Support-Vector Classification (SVC) 144
Support-Vector Machines (SVMs) 141, 143–145

T

Tabulator-Zeichen (\t) 5
Team-Rankings-Beispiel 185–186
Teile-und-herrsche-Algorithmen 214
TensorFlow-Bibliothek 32, 49
Throwaway-Parameter (_) 208
toter Code 17
Trainingsdaten 97–98, 105, 107, 111
Trainingsphase 96
Tree-Modul 133
Trees. *Siehe* Entscheidungsbäume
True-Wert. *Siehe auch* boolesche Daten
 von Python-Objekten 190–191
 und while-Schleifen 16–17

U

Überprüfen von Zugehörigkeiten 14
unbenannte vs. benannten Daten 111–112
Unsupervised Machine Learning 111–112
upper()-String-Methode 5
urllib.request-Modul 156
urlopen()-Methode 156
URL-Suche-Beispiel 166–167

V

values()-Funktion 13, 44–45
van Rossum, Guido 43
var()-Funktion 135, 137–140
Varianz 134–138, 149
Veränderbarkeit 7–8

Vereinigungsoperator (|) 195

Verkettung

(+)-Operator 195

Listen 8, 40

Strings 5

Verschleierungsalgorithmus 197–200

Verschlüsselung 196–197

Verzerrung-Varianz-Dilemma 134–135

Vorhersagen und Features 96–97

W

Web-Scraper-Beispiel 156–158

Wenigstens-Eins-Regex-Operator (+) 159

where()-Funktion 137

while-Schleifen 15–16

Whitespace-Zeichen (\s) 5, 172–175

Wortwiederholungsbeispiel 174–175

X

xkcd()-Funktion 84–85

Z

Zeitformatvalidierungsbeispiel 168–172

zip()-Funktion 21–23, 45

Zufallszahlen in Entscheidungsbäumen 134, 148–149