KAPITEL 1

Einführung

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

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

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

Warum Machine Learning?

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

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

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

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

Welche Probleme kann Machine Learning lösen?

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

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

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

Beispiele für überwachtes maschinelles Lernen sind:

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

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

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

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

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

Beispiele für unüberwachtes Lernen sind:

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

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

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

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

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

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

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

Ihre Aufgabe und Ihre Daten kennen

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

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

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

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

Warum Python?

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

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

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

scikit-learn

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

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

Installieren von scikit-learn

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

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

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

Python(x,y) (http://python-xy.github.io/)
: Eine freie Python-Distribution für wissenschaftliches Arbeiten, insbesondere unter Windows. Python(x,y) enthält NumPy, SciPy, `matplotlib`, `pandas`, IPython und `scikit-learn`.

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

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

Grundlegende Bibliotheken und Werkzeuge

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

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

Jupyter Notebook

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

NumPy

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

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

`In[1]:`

```
import numpy as np

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

`Out[1]:`

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

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

SciPy

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

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

In[2]:

```
from scipy import sparse

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

Out[2]:

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

In[3]:

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

Out[3]:

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

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

In[4]:

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

Out[4]:

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

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

matplotlib

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

`In[5]:`

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

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

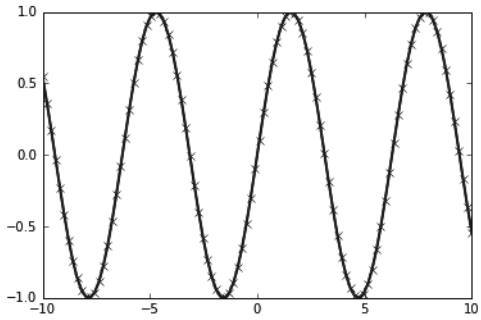

Abbildung 1-1: Mit matplotlib erstelltes Liniendiagramm einer Sinusfunktion

pandas

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

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

In[6]:

```
import pandas as pd

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

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

Dadurch erhalten wir folgende Ausgabe:

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

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

In[7]:

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

Dadurch erhalten wir folgendes Ergebnis:

	Age	Location	Name
2	**53**	**Berlin**	**Peter**
3	**33**	**London**	**Linda**

mglearn

Dieses Buch wird von Code begleitet, den Sie auf GitHub (*https://github.com/amueller/introduction_to_ml_with_python*) finden. Der begleitende Code enthält

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

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

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

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

Python 2 versus Python 3

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

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

In diesem Buch verwendete Versionen

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

In[8]:

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

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

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

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

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

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

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

Out[8]:

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

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

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

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

Eine erste Anwendung: Klassifizieren von Iris-Spezies

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

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

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

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

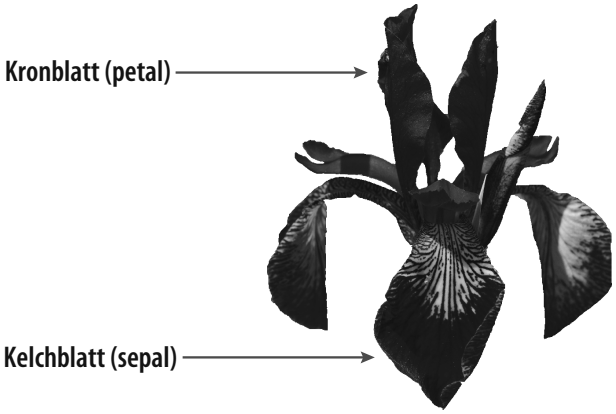

Abbildung 1-2: Teile der Irisblüte

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

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

Die Daten kennenlernen

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

In[9]:

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

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

In[10]:

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

Out[10]:

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

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

In[11]:

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

Out[11]:

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

Der Wert des Schlüssels `target_names` ist ein Array aus Strings, die die Spezies der vorherzusagenden Blüten enthalten:

In[12]:

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

Out[12]:

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

Der Wert von `feature_names` ist eine Liste von Strings mit den Beschreibungen jedes Merkmals:

In[13]:

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

Out[13]:

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

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

In[14]:

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

Out[14]:

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

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

In[15]:

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

Out[15]:

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

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

In[16]:

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

Out[16]:

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

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

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

In[17]:

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

Out[17]:

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

`target` ist ein eindimensionales Array mit einem Eintrag pro Blüte:

In[18]:

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

Out[18]:

```
Abmessungen der Zielgröße: (150,)
```

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

In[19]:

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

Out[19]:

```
Zielwerte:
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

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

Erfolg nachweisen: Trainings- und Testdaten

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

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

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

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

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

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

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

`In[20]:`

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

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

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

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

In[21]:

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

Out[21]:

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

In[22]:

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

Out[22]:

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

Das Wichtigste zuerst: Sichten Sie Ihre Daten

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

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

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

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

In[23]:

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

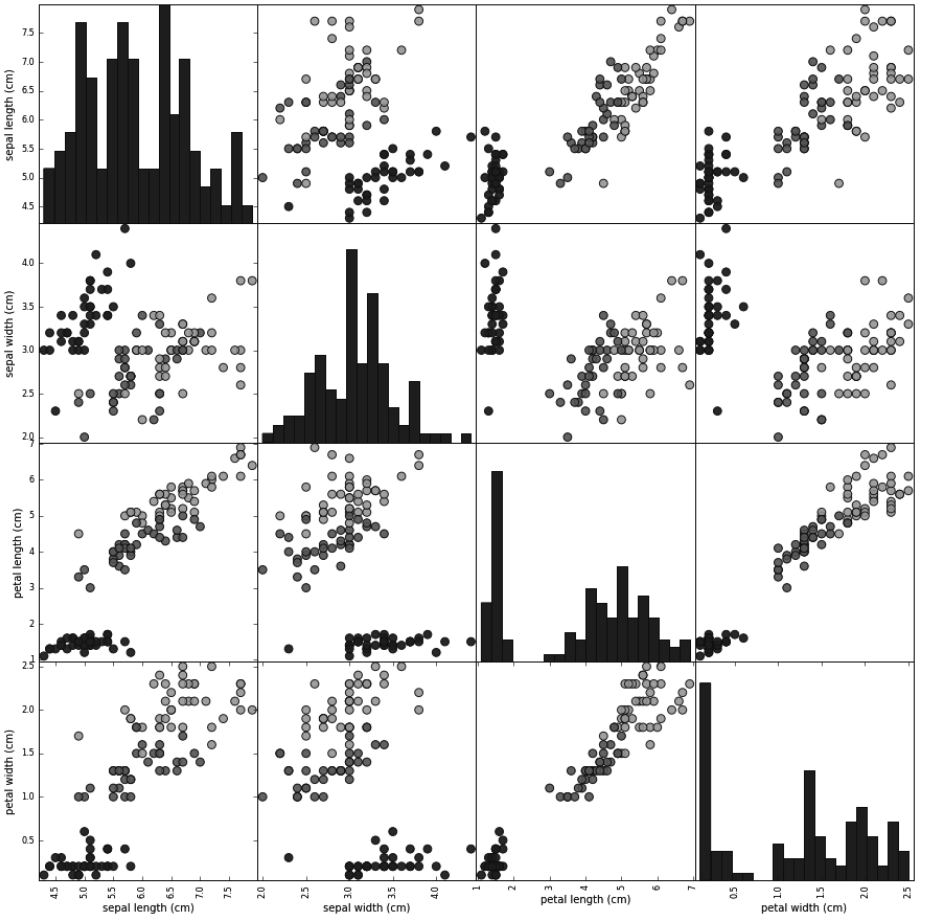

Abbildung 1-3: Paarplot für den Iris-Datensatz, nach Kategoriebezeichnung eingefärbt

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

Ihr erstes Modell konstruieren: k-nächste-Nachbarn

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

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

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

`In[24]:`

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

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

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

`In[25]:`

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

`Out[25]:`

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

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

Vorhersagen treffen

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

`In[26]:`

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

`Out[26]:`

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

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

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

`In[27]:`

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

`Out[27]:`

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

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

Evaluieren des Modells

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

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

In[28]:

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

Out[28]:

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

In[29]:

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

Out[29]:

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

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

In[30]:

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

Out[30]:

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

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

Zusammenfassung und Ausblick

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

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

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

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

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

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

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

In[31]:

```
X_train, X_test, y_train, y_test = train_test_split(
    iris_dataset['data'], iris_dataset['target'], random_state=0)

knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)

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

Out[31]:

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

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