Python-Kurs.eu – Machine Learning mit Python

Konzepte, Beispiele und Experimente

Bernd Klein

14. August 2026

Python-Kurs.eu – Machine Learning mit Python

Konzepte, Beispiele und Experimente

Autor: Bernd Klein
Website: https://www.python-kurs.eu
Stand dieser Ausgabe: 14. August 2026

Diese Ausgabe wurde automatisiert aus den autoritativen Jupyter-Notebooks von `python-kurs.eu` erzeugt. Satz und technische Produktion erfolgen reproduzierbar mit Python, Pandoc, LuaLaTeX und - soweit erforderlich - PythonTeX.

Die Online-Fassung kann aktueller sein als diese PDF-Ausgabe. Trotz sorgfältiger Prüfung kann für die Fehlerfreiheit von Texten und Programmen keine Gewähr übernommen werden.

Die im Anhang beworbenen Verlagstitel sind eigenständige Bücher beim Carl Hanser Verlag. Die vorliegende automatisch erzeugte Ausgabe ist **keine Veröffentlichung des Carl Hanser Verlags**.

Das Python-Logo wird zur Kennzeichnung des Bezugs zur Programmiersprache Python verwendet. Python und das Python-Logo sind Marken der Python Software Foundation.

Inhaltsverzeichnis

1 Maschinelles Lernen Mit Python **1**
1.0.1 Einführung in maschinelles Lernen mit Python 1

2 Maschinelles Lernen Terminologie **6**
2.0.1 Maschinelles Lernen Terminologie 6

3 Metriken **9**
3.0.1 Metriken zu Evaluation 9

4 Maschinelles Lernen Daten Visualisierung **16**
4.0.1 Repräsentierung und Visualisierung von Daten 16
4.0.2 Visualisierung der Merkmale des Iris-Datensatzes 20
4.0.3 Streudiagramm Matrix 21

5 Datensaetze In Sklearn **24**
5.0.1 Verfügbare Datensätze in scikit-learn 24
5.0.2 Digits-Datenset laden 24

6 Synthetische Daten Erzeugen Maschinelles Lernen **34**
6.0.1 Erzeugung synthetischer Daten 34

7 Maschinelles Lernen Datensaetze **47**
7.0.1 Aufteilungen der Daten 47

8 Naechste Nachbarn Klassifikation **55**
8.0.1 Nächste-Nachbarn-Klassifikation 55

9 Naechste Nachbarn Klassifikation Sklearn **72**
9.0.1 Nächste-Nachbarn-Klassifikation mit sklearn 72

10 Neuronale Netze **92**
10.0.1 Neuronale Netzwerke 92

11 Trennlinien Zwischen Klassen **96**
11.0.1 Von Trennlinien zu Neuronalen Netzen 96

12 Einfaches Neuronales Netz **116**
12.0.1 Einfaches neuronales Netz 116

13 Neuronale Netze mit scikit-learn **130**
13.0.1 Multi-Layer-Perceptron mit `MLPClassifier` 130
13.0.2 Ein kleines XOR-Beispiel 130
13.0.3 Gewichte und Bias-Werte 131
13.0.4 Ein realistischeres Beispiel mit Trainings- und Testdaten 132
13.0.5 Entscheidungsgrenze 134

13.0.6 Lernkurve des Optimierers 135
13.0.7 Wichtige Hyperparameter 136
13.0.8 Zusammenfassung 136

14 Naive Bayes Klassifikator Einfuehrung 138
14.0.1 Naive Bayes Klassifikator 138
14.0.2 Naive Bayes Klassifikator 142

15 Naive Bayes Klassifikator Scikit 151
15.0.1 Naive-Bayes-Klassifikator mit scikit-learn 151

16 Gaussian Mixture Models und Expectation-Maximization 153
16.0.1 Probabilistisches Clustering mit scikit-learn 153
16.0.2 GMM und k-means: wichtige Unterschiede 153
16.0.3 Mathematisches Modell 153
16.0.4 Expectation-Maximization (EM) 154
16.0.5 EM in einer Dimension selbst implementiert 154
16.0.6 Zweidimensionales GMM mit scikit-learn 157
16.0.7 Wie viele Komponenten? 160
16.0.8 Kovarianztypen 161
16.0.9 Numerische Singularitäten und Regularisierung 161
16.0.10 `BayesianGaussianMixture` 162
16.0.11 Zusammenfassung 162

Weitere Bücher von Bernd Klein 163

1 Maschinelles Lernen Mit Python

1.0.1 Einführung in maschinelles Lernen mit Python

Unterschied zwischen "maschinellem Lernen" (Machine Learning) und "Künstlicher Intelligenz" (Artificial Intelligence)

Andrew Moore, ehemaliger Dekan der School of Computer Science an der Carnegie Mellon University: "Artificial intelligence is the science and engineering of making computers behave in ways that, until recently, we thought required human intelligence."

Schon die Frage "Was ist Intelligenz?" lässt sich nur schwer beantworten. "Was ist künstliche Intelligenz?" hängt von der Beantwortung der vorigen Frage ab.

Hilfreich ist die Unterteilung in

schwache KI und **starke KI**

Starke und schwache KI

schwache KI:

- beschäftigt sich mit konkreten Anwendungsproblemen
- Unterstüzung des menschlichen Denkens in Teilbereichen
- lernfähig in dem Teilbereich
- kein Bewusstsein

starke KI: - "allgemeine Intelligenz" - Vergleichbar mit der menschlichen Intelligenz, muss aber nicht gleich sein, könnte andersartig sein - logisches Denken - Kommunikationsfähigkeit, natürliche Sprache - generell lernfähig - Bewusstsein? - Empfindungsvermögen, Emotionen? - Eigenwahrnehmung?

Starke KI wären Computersysteme, die auf Augenhöhe mit Menschen arbeiten und diese bei schwierigen Aufgaben unterstützen können. Demgegenüber geht es bei schwacher KI darum, konkrete Anwendungsprobleme zu meistern. Das menschliche Denken und technische Anwendungen sollen hier in Einzelbereichen unterstützt werden. Die Fähigkeit zu lernen ist eine Hauptanforderung an KI-Systeme und muss ein integraler Bestandteil von Anfang an sein, d.h. sie darf nicht erst nachträglich hinzugefügt werden. Ein zweites Hauptkriterium ist die Fähigkeit eines KI-Systems, mit Unsicherheit und probabilistischen Informationen umzugehen. Insbesondere sind solche Anwendungen von Interesse, zu deren Lösung nach allgemeinem Verständnis eine Form von „Intelligenz" notwendig zu sein scheint. Letztlich geht es der schwachen KI somit um die Simulation intelligenten Verhaltens mit Mitteln der Mathematik und der Informatik, es geht ihr nicht um Schaffung von Bewusstsein oder um ein tieferes Verständnis von Intelligenz. Während die Schaffung starker KI an ihrer philosophischen Fragestellung bis heute scheiterte, sind auf der Seite der schwachen KI in den letzten Jahren bedeutende Fortschritte erzielt worden.

in fortschrittliches KI-System muss nicht zwangsläufig viele Ähnlichkeiten mit dem mensch-

lichen Wesen aufweisen. Es wird wahrscheinlich über eine abweichende kognitive Architektur verfügen und in seinen Entwicklungsstadien ebenso wenig mit den evolutionären kognitiven Phasen des menschlichen Denkens vergleichbar sein (Evolution des Denkens). Vor allem ist nicht anzunehmen, dass eine künstliche Intelligenz Gefühle wie Liebe, Hass, Angst oder Freude besitzt. Dennoch kann es ein Verhalten simulieren, das diesen Gefühlen ähnelt.

Was versteht man unter Maschinellem Lernen?

Beginnen wir mit einem sehr "alten" Versuch einer Definition von Arthur Samuek, einem IBM-Pionier:

> "Machine Learning: Field of study that gives computers the ability to learn without being explicitly programmed." ("Maschinelles Lernen: Ein Studienbereich, in dem Computer lernen können, ohne explizit programmiert zu werden.")

Ein guter Versuch, aber es bleiben viele Fragen offen. Unter anderem die Frage, was "Lernen" bedeutet. Fast 40 Jahre später, 1998, prägt Tom Mitchell ein "wohlgestelltes Lernproblem" wiefolgt:

> "Well posed Learning Problem: A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E." (Ein wohlgestelltes Lernproblem: Ein Computerprogramm soll aus Erfahrung E in Bezug auf eine Aufgabe T und ein Leistungsmaß P lernen, wenn sich seine Leistung auf T, gemessen durch P, mit Erfahrung E verbessert.)

(Anmerkung: Ein mathematisches Problem heißt korrekt gestellt (auch wohlgestellt, gut gestellt oder sachgemäß gestellt), wenn folgende Bedingungen erfüllt sind:

- Das Problem hat eine Lösung (Existenz).
- Diese Lösung ist eindeutig bestimmt (Eindeutigkeit).
- Diese Lösung hängt stetig von den Eingangsdaten ab (Stabilität).)

Also, was ist maschinelles Lernen?

Maschinelles Lernen ist der Prozess des automatischen Extrahierens von Wissen aus Daten, normalerweise mit dem Ziel, Vorhersagen über neue, unsichtbare Daten zu treffen. Wie bereits erwähnt, könnte ein Spamfilter mittels eines Klassifikators basierend auf maschinellem Lernen realisiert werden.

Im Zentrum des maschinellen Lernens steht das Konzept, die Entscheidungsfindung aus Daten zu automatisieren, ohne dass der Benutzer explizite Regeln angibt, wie diese Entscheidung getroffen werden soll. Für den Fall von E-Mails gibt der Benutzer keine Liste von Wörtern oder Merkmalen an, die eine E-Mail zu Spam machen. Stattdessen gibt der Benutzer Beispiele für Spam- und Nicht-Spam-E-Mails an, die als solche gekennzeichnet sind. Dabei handelt es sich dann um das sogenannte Lernset.

Das Ziel eines maschinellen Lernmodells ist die Vorhersage neuer, zuvor nicht sichtbarer Daten. In einer realen Anwendung sind wir nicht daran interessiert, eine bereits gekennzeichnete E-Mail als Spam zu markieren oder nicht. Stattdessen möchten wir dem Benutzer das Leben erleichtern, indem wir neue eingehende E-Mails automatisch klassifizieren.

Diese Beispiele werden dann vom Algorithmus gelernt oder trainiert.

Maschinelles Lernen

Maschinelles Lernen bedeutet, dass ein Algorithmus (die Maschine) automatisch lernt. Dies bedeutet, dass es in der Lage ist, das notwendige Wissen automatisch aus gegebenen Daten zu extrahieren. Ziel ist es, Vorhersagen für neue, unsichtbare Daten zu treffen. Es gibt eine andere Art, es auszudrücken: In traditionellen heuristischen Entscheidungsalgorithmen legen die Programmierer die Regeln fest, nach denen die Entscheidungen getroffen werden. Beim maschinellen Lernen werden solche Entscheidungsregeln aus Beispieldaten gelernt. Menschen legen jedoch weiterhin unter anderem Daten, Merkmale, Modellklasse, Zielgröße und Evaluationsverfahren fest; das Lernen dieser Regeln erfolgt also nicht völlig ohne menschliche Entscheidungen.

Taxonomie des maschinellen Lernens

Eine grundlegende Unterscheidung ist:

- **Überwachtes Lernen (supervised learning):** Zu den Trainingsbeispielen sind Zielwerte bzw. Labels bekannt.
- **Unüberwachtes Lernen (unsupervised learning):** Es sind keine Zielwerte vorgegeben; gesucht werden Strukturen in den Daten.

Daneben gibt es weitere Lernformen, beispielsweise semi-überwachtes Lernen. In diesem Tutorial liegt der Schwerpunkt auf überwachten Verfahren.

Überwachtes Lernen: Klassifikation und Regression

Beim *überwachten Lernen* haben wir einen Datensatz, der sowohl aus Eingabemerkmalen als auch aus einer gewünschten Ergebnis besteht, wie im Beispiel für Spam / No-Spam. Die Aufgabe besteht darin, ein Modell (oder Programm) zu erstellen, das die gewünschte Ausgabe eines unbekannten Objektes vorhersagen kann anhand von Merkmalen.

Einige kompliziertere Beispiele sind:

- Bestimmung eines Zeichen, wenn ein Pixelbild des Zeichens vorliegt.
- Man hat ein Bild eines der Tiere „Hunde“, „Katzen“, „Kühe“, „Schafe“ und möchte nun bestimmen, um welches Tier es sich handelt.
- Personen auf Fotos bestimmen.
- Bestimmen Sie anhand einer Liste von Büchern oder Filmen, welche anderen Bücher oder Filme einer Person gefallen könnten.

Gemeinsam ist diesen Aufgaben, dass es eine oder mehrere unbekannte dem Objekt zuzuordnende Quantitäten gibt, die aus beobachteten Eigenschaften erschlossen werden müssen.

Das überwachte Lernen wird weiter in zwei Kategorien unterteilt:

- Klassifikation:
- Regression:

Auf den Punkt gebracht: Bei der Klassifizierung geht es um die **Vorhersage eines Labels** und bei der Regression um die **Vorhersage einer Quantität**.

- **Bei der Klassifizierung ist das Label diskret**, z. B. “Spam” oder “kein Spam”. Mit anderen Worten bietet es eine klare Unterscheidung zwischen Kategorien. Klassenlabels

sind diskrete Zielwerte. Häufig sind sie nominal, etwa `Spam/kein Spam`; es gibt aber auch Klassifikationsprobleme mit geordneten Kategorien (ordinale Klassifikation). Entscheidend für die Abgrenzung zur Regression ist hier, dass diskrete Klassen und nicht ein kontinuierlicher Zahlenwert vorhergesagt werden.

- **Bei der Regression ist die Bezeichnung fortlaufend**, d.h. eine Float-Ausgabe.

Beispielsweise ist die Aufgabe, festzustellen, welches Tier („Hund“, „Katze“, „Kuh“, „Schaf“) in einem Bild dargestellt ist, ein Klassifizierungsproblem, d.h. vier verschiedene Kategorien. Auf der anderen Seite könnten wir das Alter eines Objekts anhand einiger Beobachtungen abschätzen wollen: Dies wäre ein Regressionsproblem. weil das Etikett (Alter) eine kontinuierliche Größe ist.

Beim überwachten Lernen wird immer zwischen einem **Trainingssatz** (Lernset), für den das gewünschte Ergebnis angegeben bzw. bekannnt ist, und einem **Testsatz** unterschieden, für den das gewünschte Ergebnis abgeleitet oder berechnet werden muss. Das Lernmodell passt das Vorhersagemodell an den Trainingssatz an, und wir verwenden den Testsatz, um seine Generalisierungsleistung zu bewerten.

Unüberwachtes Lernen

Beim “Unüberwachten Lernen” ist den Daten keine gewünschte Ausgabe zugeordnet. Stattdessen sind wir daran interessiert, irgendeine Form von Wissen oder Modell aus den gegebenen Daten zu extrahieren. In gewissem Sinne können Sie sich unbeaufsichtigtes Lernen als Mittel vorstellen, um Labels (Etiketten) aus den Daten selbst zu entdecken. Unüberwachtes Lernen ist oft schwieriger zu verstehen und zu bewerten.

Unbeaufsichtigtes Lernen umfasst Aufgaben wie Dimensionsreduktion, Clustering und Dichteschätzung. Zum Beispiel können wir in den oben diskutierten Irisdaten unbeaufsichtigte Methoden verwenden, um Kombinationen der Messungen zu bestimmen, die die Struktur der Daten am besten darstellen. Wie wir unten sehen werden, kann eine solche Projektion der Daten verwendet werden, um den vierdimensionalen Datensatz in zwei Dimensionen zu visualisieren. Einige weitere unbeaufsichtigte Lernprobleme sind:

- Bestimmen Sie anhand detaillierter Beobachtungen entfernter Galaxien, welche Merkmale oder Merkmalskombinationen die Informationen am besten zusammenfassen.
- Trennen Sie eine Mischung aus zwei Schallquellen (z. B. eine Person spricht, während Musik spielt)
- Isolieren Sie in einem gegebenen Video ein sich bewegendes Objekt und kategorisieren Sie es in Bezug auf andere sich bewegende Objekte, die gesehen wurden.
- Bei einer großen Sammlung von Nachrichtenartikeln finden Sie in diesen Artikeln wiederkehrende Themen.
- Bei einer bestimmten Bildersammlung werden ähnliche Bilder zu einem Cluster zusammengefasst (um sie beispielsweise bei der Visualisierung einer Sammlung zu gruppieren).

Manchmal können die beiden sogar kombiniert werden: z. Unbeaufsichtigtes Lernen kann verwendet werden, um nützliche Merkmale in heterogenen Daten zu finden, und diese Merkmale können dann in einem überwachten Rahmen verwendet werden.

Beispiele für maschinelles Lernen

- Spam Filter: Der Algorithmus lernt ein Vorhersagemodell aus Daten, die als "Spam" und "kein Spam" (ham) gekennzeichnet sind. Nach dem Training kann für neue E-Mails vorhergesagt werden, ob es sich um Spam handelt oder nicht.
- Zeichenerkennung
- Objekterkennung in Bildern
- und viele mehr

Wie bereits erwähnt, könnte ein Spamfilter unter Verwendung eines Klassifikators implementiert werden, der auf maschinellem Lernen basiert.

Im Zentrum des maschinellen Lernens steht das Konzept der Automatisierung der Entscheidungsfindung aus Daten, ohne dass der Benutzer explizite Regeln für die Entscheidungsfindung festlegt. Bei E-Mails stellt der Benutzer keine Liste von Wörtern oder Funktionen bereit, die eine E-Mail als Spam auszeichnen. Stattdessen stellt der Benutzer Beispiele für Spam- und Nicht-Spam-E-Mails bereit, die als solche gekennzeichnet sind. Dies ist das sogenannte Lernset.

Ziel eines maschinellen Lernmodells ist es, neue, bisher unsichtbare Daten vorherzusagen. In einer realen Anwendung sind wir nicht daran interessiert, eine bereits markierte E-Mail als Spam zu markieren oder nicht. Stattdessen möchten wir den Benutzern das Leben erleichtern, indem wir neue eingehende E-Mails automatisch klassifizieren.

Diese Beispiele werden dann vom Algorithmus gelernt oder trainiert:

Nach der Lernphase müssen wir beurteilen, wie gut das Modell auf **unbekannte Daten** generalisiert. Ein Ergebnis auf den Trainingsdaten kann zur Diagnose nützlich sein, ist aber keine unabhängige Qualitätsbewertung: Das Modell hat diese Daten bereits gesehen.

Für die Modellwahl oder die Abstimmung von Hyperparametern verwenden wir eine Validierungsmenge oder – meist besser – Kreuzvalidierung auf den Trainingsdaten. Der eigentliche Testsatz bleibt bis zur abschließenden Bewertung unangetastet. So vermeiden wir, dass Entscheidungen indirekt an die Testdaten angepasst werden.

Wenn wir mit dem gewählten Modell zufrieden sind und es abschließend auf bislang unberührten Testdaten bewertet haben, kann es Vorhersagen für neue Daten erzeugen:

Die Daten werden dem Algorithmus normalerweise als zweidimensionales Array (oder Matrix) von Zahlen präsentiert. Jeder Datenpunkt (auch als *Training* oder *Trainingsinstanz* bezeichnet), von dem wir entweder lernen oder eine Entscheidung treffen möchten, wird als eine Liste von Zahlen dargestellt, ein sogenannter Merkmalsvektor, und seine enthaltenen Merkmale repräsentieren die Eigenschaften dieses Objektes.

2 Maschinelles Lernen Terminologie

2.0.1 Maschinelles Lernen Terminologie

Klassifikator Ein Programm oder eine Funktion die unstrukturierte Daten zu Klassen zuordnet, wird als Klassifikator bezeichnet.

Konfusionsmatrix Eine Konfusionsmatrix, auch Kontingenztablle oder Fehlermatrix genannt, wird für die Darstellung der Performance eines Klassifikators verwendet.

Die Spalten der Matrix stellen die Instanzen der vorhergesagten Klassen dar. Die Zeilen stellen die Instanzen der aktuellen Klasse dar. (Hinweis: Es kann ebenso auch andersherum dargestellt werden.)

Im Falle der binären Klassifikation hat die Tabelle 2 Spalten und 2 Zeilen.

Beispiel:

KonfusionsMatrix

Vorhergesagte Klassen

männlich

weiblich

Aktuelle Klassen

männlich

42

8

weiblich

18

32

Das bedeutet, dass der Klassifikator in 42 Fällen korrekt eine männliche Person vorhergesagt hat und in 8 Fällen fälschlicherweise männliche Personen als weibliche vorhergesagt hat. In 32 Fällen hat er bei weiblichen Personen richtig gelegen, jedoch bei 18 Personen wurde männlich statt weiblich vorhergesagt.

Genauigkeit und Fehlerrate Die Genauigkeit (Accuracy) ist der Anteil der korrekten Vorhersagen an allen Vorhersagen. Die Fehlerrate ist ihr Komplement: `1 - Accuracy`.

Der Klassifikator in unserem Beispiel hat 42 männliche und 32 weibliche Personen korrekt vorhergesagt.

Die Genauigkeit kann damit wie folgt berechnet werden:

genauigkeit $= (42 + 32)/(42 + 8 + 18 + 32)$

was 0.72 ergibt.

Angenommen wir haben einen Klassifikator, der immer “weiblich” vorhersagt. Damit haben wir eine Genauigkeit von 50 %.

KonfusionsMatrix

Vorhergesagte Klassen

männlich

weiblich

Aktuelle Klassen

männlich

0

50

weiblich

0

50

Wir demonstrieren das s.g. Genauigkeits-Paradox.

Ein Spam-Erkennungs-Klassifikator wird durch die folgende Konfusionsmatrix beschrieben:

KonfusionsMatrix

Vorhergesagte Klassen

spam

ham

Aktuelle Klassen

spam

4

1

ham

4

91

Die Genauigkeit dieses Klassifikators ist $(4 + 91)/100$, also 95 %.

Der folgende Klassifikator bestimmt “ham” mit der gleichen Genauigkeit.

KonfusionsMatrix

Vorhergesagte Klassen

spam

ham

Aktuelle Klassen

spam

0

5

ham

0

95

Die Genauigkeit dieses Klassifikators liegt bei 95 %, selbst dann, wenn er nicht in der Lage ist Spam zu erkennen.

Präzision und Recall KonfusionsMatrix

Vorhergesagte Klassen

negativ

positiv

Aktuelle Klassen

negativ

TN

FP

positiv

FN

TP

Genauigkeit: $(TN + TP)/(TN + TP + FN + FP)$

Präzision: $TP/(TP + FP)$

Recall (Sensitivität): $ TP / (TP + FN)$

Überwachtes Lernen (Supervised Learning) Das Machine Learning-Programm wird sowohl mit Daten, als auch mit den zugehörigen Bezeichnungen (Labels) versorgt. Die zu lernenden Daten müssen also vorher durch einen Menschen manuell bezeichnet werden.

Nicht überwachtes Lernen (Unsupervised Learning) Es sind keine Bezeichnungen (Labels) gegeben. Der Machine Learning-Algorithmus muss aus den Daten Gemeinsamkeiten (Cluster) herausfinden.

Bestärkendes Lernen (Reinforcement learning) Ein Computer-Programm interagiert mit seiner Umgebung. Es bekommt also positive und/oder negative Rückmeldung um sein Verhalten zu verbessern.

3 Metriken

3.0.1 Metriken zu Evaluation

Einführung

Nicht nur beim maschinellen Lernen, sondern auch im allgemeinen Leben, insbesondere im Geschäftsleben, werden Sie Fragen wie "Wie genau ist Ihr Produkt?" oder "Wie präzise ist Ihre Maschine?" hören. Wenn Leute Antworten wie "Dies ist das genaueste Produkt auf seinem Gebiet!" oder "Diese Maschine hat die höchste vorstellbare Präzision!" erhalten, fühlen sich beide Antworten gut an. Sollten sie nicht? Wenn Sie beide Antworten gleichzeitig erhalten ja, aber jeweils eine alleine könnte problematisch sein. In der Tat werden die Begriffe "genau" und "präzise" sehr oft synonym verwendet. Wir werden später im Text genaue Definitionen geben, aber kurz gesagt können wir sagen: Genauigkeit ist ein Maß für die Nähe einiger Messungen zu einem bestimmten Wert, während Präzision die Nähe der Messungen zueinander ist.

Diese Begriffe sind auch beim maschinellen Lernen von äußerster Wichtigkeit. Wir brauchen sie zur Bewertung von ML-Algorithmen oder besser zu ihren Ergebnissen.

In diesem Kapitel unseres Python-Lernprogramms für maschinelles Lernen werden vier wichtige Metriken vorgestellt. Diese Metriken werden verwendet, um die Ergebnisse von Klassifizierungen auszuwerten.

Die Metriken sind:

- Genauigkeit (engl. Accuracy)
- Präzision (engl. precision)
- Recall
- F-Maß (englisch F1-score)

Wir werden jede dieser Metriken vorstellen und ihre Vor- und Nachteile diskutieren. Jede Metrik misst andere Aspekte der Leistung eines Klassifikators. Die Metriken sind für alle Kapitel unseres Tutorials zum maschinellen Lernen von größter Bedeutung.

Genauigkeit gegenüber Präzision

Die **Genauigkeit** ist ein Maß für die Nähe von Messungen zu einem bestimmten Wert (dem gewünschten Wert), während die **Präzision** ein Maß für die Nähe der Messungen zueinander ist, d.h. also nicht notwendig zu einem tatsächlichen oder gewünschten Wert. Mit anderen Worten: Wenn wir einen Satz von Datenpunkten aus wiederholten Messungen derselben Größe haben, wird der Satz als genau bezeichnet, wenn ihr Durchschnitt nahe am tatsächlichen Wert liegt. Andererseits nennen wir die Menge präzise, wenn die Messwerte nahe beieinander liegen, aber möglicherweise entfernt vom tatsächlichen Wert. Die beiden Konzepte sind unabhängig voneinander, was bedeutet, dass der Datensatz genau, präzise, präzise und genau oder keins von beides sein kann. Wir zeigen dies im folgenden Diagramm:

Wahrheitsmatrix

Bevor wir mit dem Begriff "Genauigkeit" fortfahren, möchten wir sicherstellen, dass Sie verstehen, worum es in einer Wahrheitsmatrix, auch Konfusionsmatrix (englisch confusion matrix) genannt, geht.

Eine Konfusionsmatrix wird verwendet, um die Leistung eines Klassifikators zu visualisieren.

Die Spalten der Matrix repräsentieren die Instanzen der vorhergesagten Klassen und die Zeilen repräsentieren die Instanzen der tatsächlichen Klassen. (Hinweis: Es kann auch umgekehrt sein.)

Bei der binären Klassifizierung hat die Tabelle 2 Zeilen und 2 Spalten.

Wir wollen dieses Konzept mit einem Beispiel erläutern.

Beispiel:

Konfusionsmatrix

Prognostizierte Klassen

Katze

Hund

Tatsächliche Klassen

Katze

42

8

Hund

18

32

Dies bedeutet, dass der Klassifikator eine Katze in 42 Fällen korrekt vorhergesagt hat und 8 Katzeninstanzen fälschlicherweise als Hund vorhergesagt hat. Es wurden 32 Fälle als Hund korrekt vorhergesagt. 18 Fälle wurden fälschlicherweise als Katze statt als Hund vorhergesagt.

Genauigkeit bei der Klassifizierung

Wir interessieren uns für maschinelles Lernen und Genauigkeit wird auch als statistische Messgröße verwendet. Die Genauigkeit ist ein statistisches Maß, das als Quotient korrekter Vorhersagen (sowohl True Positives (TP) als auch True Negatives (TN)) eines Klassifikators geteilt durch die Summe aller vom Klassifikator gemachten Vorhersagen, einschließlich falsch Positive (FP) und falsch Negative (FN) definiert wird. Daher lautet die Formel zur Quantifizierung der binären Genauigkeit :

$$Genauigkeit = \frac{TP + TN}{TP + TN + FP + FN}$$

wobei gilt: TP = True positive; FP = False positive; TN = True negative; FN = False negative

Die dazugehörige Konfusionsmatrix sieht wie folgt aus:

ConfusionMatrix

Vorhergesagte Klassen

negativ

positiv

Tatsächliche Klassen

negativ

TN

FP

positiv

FN

TP

Wir werden nun die Genauigkeit für die Ergebnisse der Katzen- und Hundeklassifizierung berechnen. Anstelle von “richtig” und “falsch” sehen wir hier “Katze” und “Hund”. Wir können die Genauigkeit so berechnen :

```
TP = 42
TN = 32
FP = 8
FN = 18

genauigkeit = (TP + TN)/(TP + TN + FP + FN)
print(genauigkeit)
```

```
0.74
```

Nehmen wir an, dass eir einen Klassifikator hätten, der immer “Hund” vorhersagt.

Konfusionsmtrix

Vorhergesagte Klassen

Katze

Hund

Actual classes

Katze

0

50

Hund

0

50

In diesem Fall haben wir eine Genauigkeit von 0.5:

```
TP, TN, FP, FN = 0, 50, 50, 0
genauigkeit = (TP + TN)/(TP + TN + FP + FN)
print(genauigkeit)
```

```
0.5
```

Genauigkeitsparadoxon

Wir werden das sogenannte Genauigkeitsparadoxon demonstrieren.

Ein Spam-Erkennungsklassifizierer wird durch die folgende Konfusionsmatrix beschrieben. Dabei steht “spam” für die Spam-Emails und “ham” für die guten Emails/p>

Konfusionsmatrix

Vorhergesagte Klassen

spam

ham

Tatsächliche Klassen

spam

4

1

ham

4

91

```
TP, TN, FP, FN = 4, 91, 4, 1
accuracy = (TP + TN) / (TP + TN + FP + FN)
print(accuracy)
```

```
0.95
```

Der folgende Klassifikator sagt immer nur “ham” voraus und hat die gleich Genauigkeit.

Konfusionsmatrix

Vorhergesagte Klassen

spam

ham

Tatsächliche Klassen

spam

0

5

ham

0

95

```
TP, TN, FP, FN = 0, 95, 0, 5
accuracy = (TP + TN) / (TP + TN + FP + FN)
print(accuracy)
```

```
0.95
```

Die Genauigkeit dieses Klassifikators beträgt 95%, obwohl er überhaupt keinen Spam erkennen kann.

Präzision und Recall

Ab jetzt betrachten wir **Spam als positive Klasse**. Damit gilt:

- `TP`: Spam wurde korrekt als Spam erkannt.
- `FP`: Eine Ham-Nachricht wurde fälschlich als Spam markiert.
- `FN`: Spam wurde fälschlich als Ham eingestuft.
- `TN`: Ham wurde korrekt als Ham erkannt.

Die **Präzision (Precision)** beantwortet die Frage: *Welcher Anteil der als positiv vorhergesagten Fälle ist tatsächlich positiv?*

$$\text{Precision} = \frac{TP}{TP + FP}$$

Der **Recall** (Sensitivität) beantwortet: *Welcher Anteil der tatsächlich positiven Fälle wurde gefunden?*

$$\text{Recall} = \frac{TP}{TP + FN}$$

Konfusionsmatrix

Vorhergesagte Klassen

spam

ham

Tatsächliche Klassen

spam

12

14

ham

0

114

In der obigen Matrix stehen 12 korrekt erkannte Spam-Nachrichten (`TP`), 14 übersehene Spam-Nachrichten (`FN`), keine fälschlich als Spam markierte Ham-Nachricht (`FP`) und 114

korrekt erkannte Ham-Nachrichten (TN).

```
TP, TN, FP, FN = 12, 114, 0, 14
precision = TP / (TP + FP)
recall = TP / (TP + FN)
print(f"Precision: {precision:.3f}")
print(f"Recall:    {recall:.3f}")
```

```
Precision: 1.000
Recall:    0.462
```

Die Präzision ist hier 1.0: Alles, was der Filter als Spam bezeichnet, ist tatsächlich Spam. Der Recall ist dagegen deutlich kleiner als 1.0, weil 14 Spam-Nachrichten nicht erkannt wurden.

Welche Fehlerart schwerer wiegt, hängt von der Anwendung ab. Bei einem Spamfilter können False Positives besonders problematisch sein, weil erwünschte Nachrichten im Spamordner landen oder sogar automatisch verworfen werden. In anderen Anwendungen – etwa bei bestimmten medizinischen Screenings – können False Negatives schwerer wiegen. Deshalb sollte die Wahl einer Metrik immer die Kosten der jeweiligen Fehler berücksichtigen.

```
from sklearn.metrics import precision_score, recall_score

y_true = [1] * (TP + FN) + [0] * (FP + TN)
y_pred = [1] * TP + [0] * FN + [1] * FP + [0] * TN

print(f"Precision (sklearn): {precision_score(y_true, y_pred):.3f}")
print(f"Recall (sklearn):    {recall_score(y_true, y_pred):.3f}")
```

```
Precision (sklearn): 1.000
Recall (sklearn):    0.462
```

F1-Score

Der F1-Score ist das harmonische Mittel aus Precision und Recall:

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Er ist besonders nützlich, wenn beide Größen berücksichtigt werden sollen. Er ersetzt aber keine fachliche Entscheidung darüber, ob False Positives oder False Negatives unterschiedlich teuer sind.

```
from sklearn.metrics import accuracy_score, f1_score

print(f"Accuracy:  {accuracy_score(y_true, y_pred):.3f}")
print(f"Precision: {precision_score(y_true, y_pred):.3f}")
print(f"Recall:    {recall_score(y_true, y_pred):.3f}")
print(f"F1:        {f1_score(y_true, y_pred):.3f}")
```

```
Accuracy:  0.900
Precision: 1.000
Recall:    0.462
```

```
F1:         0.632
```

Accuracy, Precision, Recall und F1 betrachten unterschiedliche Aspekte eines Klassifikators. Bei unausgeglichenen Klassen kann eine hohe Accuracy beispielsweise wenig aussagekräftig sein. Für eine vollständige Beurteilung sind deshalb häufig mehrere Metriken sowie die Konfusionsmatrix sinnvoll.

4 Maschinelles Lernen Daten Visualisierung

4.0.1 Repräsentierung und Visualisierung von Daten

Einleitung

Beim maschinellen Lernen geht es darum, Modelle an Daten anzupassen. Aus diesem Grund beginnen wir damit zu zeigen wie Daten dargestellt werden können, um vom Computer verstanden zu werden.

Zu Beginn dieses Kapitels haben wir Tom Mitchells Definition des maschinellen Lernens zitiert: Ein wohlgestelltes Lernproblem: Ein Computerprogramm soll aus der Erfahrung E in Bezug auf eine Aufgabe T und ein Leistungsmaß P lernen, wenn sich seine Leistung auf T, gemessen durch P, mit Erfahrung E verbessert. Daten sind der "Rohstoff" für maschinelles Lernen. ML lernt aus Daten. In Mitchells Definition sind "Daten" hinter den Begriffen "Erfahrung E" und "Leistungsmaß P" verborgen. Wie bereits erwähnt, benötigen wir gelabelte Daten, um unseren Algorithmus zu trainieren und zu testen.

Es empfiehlt sich jedoch, dass man sich mit seinen Daten vertraut macht, bevor man mit dem Training des Klassifikators beginnt.

Numpy bietet ideale Datenstrukturen zur Darstellung der Daten und Matplotlib bietet hervorragende Möglichkeiten zur Visualisierung der Daten. Wie man hierbei vorgeht, wollen wir im Folgenden anhand der Daten zeigen, die sich im Modul `sklearn` befinden.

Ein einfaches Beispiel: das Iris-Datenset

Was war das erste Programm, das Sie gesehen haben? Ich wette, es könnte ein Programm gewesen sein, das "Hello World" in einer Programmiersprache ausgegeben hat. Höchstwahrscheinlich habe ich recht. Fast jedes Einführungsbuch oder Tutorial zur Programmierung beginnt mit einem solchen Programm. Es ist eine Tradition, die auf das Buch "The C Programming Language" von 1968 von Brian Kernighan und Dennis Ritchie zurückgeht!

Die Wahrscheinlichkeit, dass der erste Datensatz, den Sie in einem Einführungstutorial zum maschinellen Lernen sehen, der "Iris-Datensatz" ist, ist ähnlich hoch. Der Iris-Datensatz enthält die Messungen von 150 Blüten von Schwertlilien (englisch "iris") aus drei verschiedenen Arten: Als Beispiel für einen einfachen Datensatz sehen wir uns die von scikit-learn gespeicherten Irisdaten an. Die Daten bestehen aus Messungen von drei verschiedenen Irisblütenarten. Es gibt drei verschiedene Arten von Schwertlilien (Iris) in diesem speziellen Datensatz wie unten dargestellt:

Iris Setosa (Borsten-Schwertlilie)

Iris Versicolor (Verschiedenfarbige-Schwertlilie)

Iris Virginica

Der Iris-Datensatz wird häufig wegen seiner Einfachheit verwendet. Dieser Datensatz ist in scikit-learn enthalten.

Die Daten bestehen aus den ermittelten Abmessungen der Kronblätter (Fachbegriff: Petalum oder engl. petal) und der Kelchblätter (Fachbegriff: Sepalum) von drei verschiedenen Lilienarten:

- Merkmale im Iris-Datensatz:
    1. Kelchlänge (sepal length) in cm
    2. Kelchbreite (sepal width) in cm
    3. Kronblattlänge (petal length) in cm
    4. Kronblattbreite (petal width) in cm
- Zu prognostizierende Zielklassen:
    1. Iris Setosa
    2. Iris Versicolour
    3. Iris Virginica

`scikit-learn` beinhaltet eine Kopie der Iris-CSV-Datei zusammen mit einer Hilfsfunktion, um diese Daten in numpy-Arrays zu laden:

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

Der resultierende Datensatz ist ein "Bunch"-Objekt:

```
type(iris)
```

```
sklearn.utils.Bunch
```

The resulting dataset is a `Bunch` object: you can see what's available using the method `keys()`:

```
iris.keys()
```

```
dict_keys(['data', 'target', 'target_names', 'DESCR', 'feature_names',
↪  'filename'])
```

Ein Bunch-Objekt ähnelt einem Dictionary, ermöglicht jedoch zusätzlich den Zugriff auf die Schlüssel im Attributstil:

```
print(iris["target_names"])
print(iris.target_names)
```

iris.data.shape enthält die Anzahl der Merkmale und Samples (Stichproben):

```
n_samples, n_features = iris.data.shape
print('Anzahl der Sampels:', n_samples)
print('Anzahl der Merkmale:', n_features)
# the sepal length, sepal width, petal length and petal width of the first
↪  sample (first flower)
print(iris.data[0])
```

```
Anzahl der Sampels: 150
Anzahl der Merkmale: 4
[5.1 3.5 1.4 0.2]
```

Die Merkmale jeder Blume werden im Attribut `data` des Datensatzes gespeichert. Werfen wir einen Blick auf einige der Sampels:

```
# Blumen mit den Indizes 12, 26, 89 und 114
iris.data[[12, 26, 89, 114]]
```

```
array([[4.8, 3. , 1.4, 0.1],
       [5. , 3.4, 1.6, 0.4],
       [5.5, 2.5, 4. , 1.3],
       [5.8, 2.8, 5.1, 2.4]])
```

Die Informationen über die Klasse jeder Probe, d.h. die Labels, werden im Attribut "target" des Datensatzes gespeichert:

```
print(iris.data.shape)
print(iris.target.shape)
```

```
(150, 4)
(150,)
```

```
print(iris.target)
```

```
[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 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 2]
```

```
import numpy as np

np.bincount(iris.target)
```

```
array([50, 50, 50])
```

Mit der Bincount-Funktion von NumPy (oben) können wir sehen, dass die Klassen in diesem Datensatz gleichmäßig verteilt sind - es gibt 50 Blumen von jeder Art, wobei

- class 0: Iris-Setosa
- class 1: Iris-Versicolor
- class 2: Iris-Virginica

Diese Klassennamen werden im letzten Attribut gespeichert, nämlich `target_names`:

```
print(iris.target_names)
```

```
['setosa' 'versicolor' 'virginica']
```

Daten in sklearn (scikit-learn)

Bei den Daten in scikit-learn kann man mit wenigen Ausnahmen davon ausgehen, dass sie als **zwei-dimensionale Arrays** abgespeichert sind. Ihre Gestalt (shape) ist von der Form "(n_samples, n_features)".

- **n_samples:** Die Anzahl der Samples: Jedes Sample ist ein zu verarbeitendes Objekt. Eine Stichprobe kann ein Dokument, ein Bild, ein Ton, ein Video oder ein astronomisches Objekt sein, Eine Stichprobe entspricht einer Zeile in einer Datenbank oder CSV-Datei, oder was auch immer man mit einem festen Satz von quantitativen Merkmalen beschreiben können.
- **n_features:** Die Anzahl der Merkmale oder besonderen Merkmale, die zur Beschreibung der einzelnen Merkmale verwendet werden können Artikel in quantitativer Weise. Features sind im Allgemeinen reelle Werte, können aber auch boolesche Werte oder Werte sein in einigen Fällen diskret bewertet.

Die Anzahl der Features muss im Voraus festgelegt werden. Es kann jedoch sehr hoch dimensioniert sein (z.B. Millionen von Merkmalen), wobei die meisten von ihnen "Null" für eine gegebene Stichprobe sein können. Das ist ein Fall wo `scipy.sparse` Matrizen nützlich sein können, sind sie viel speichereffizienter als NumPy-Arrays.

Wie wir uns aus dem vorherigen Abschnitt (oder dem Jupyter-Notizbuch) erinnern, stellen wir Beispiele (Datenpunkte oder Instanzen) als Zeilen im Datenarray dar und speichern die entsprechenden Features, die "Dimensionen", als Spalten.

Die Daten für eine Blumen bestehen aus einer Zeile im Datenarray, und die Spalten (Merkmale) geben die Blumenmaße in Zentimetern an. Zum Beispiel können wir diesen Iris-Datensatz, bestehend aus 150 Beispielen und 4 Merkmalen, einem zweidimensionalen Array oder einer Matrix $\mathbb{R}^{150\times 4}$, im folgenden Format darstellen:

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

Der hochgestellte Index bezeichnet die i-te Zeile, und der tiefgestellte Index bezeichnet das j-te Merkmal.

Im Allgemeinen haben wir n Zeilen und k Spalten:

$$\mathbf{X} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & \dots & x_k^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & \dots & x_k^{(2)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_1^{(n)} & x_2^{(n)} & x_3^{(n)} & \dots & x_k^{(n)} \end{bmatrix}.$$

```
print(iris.data.shape)
print(iris.target.shape)
```

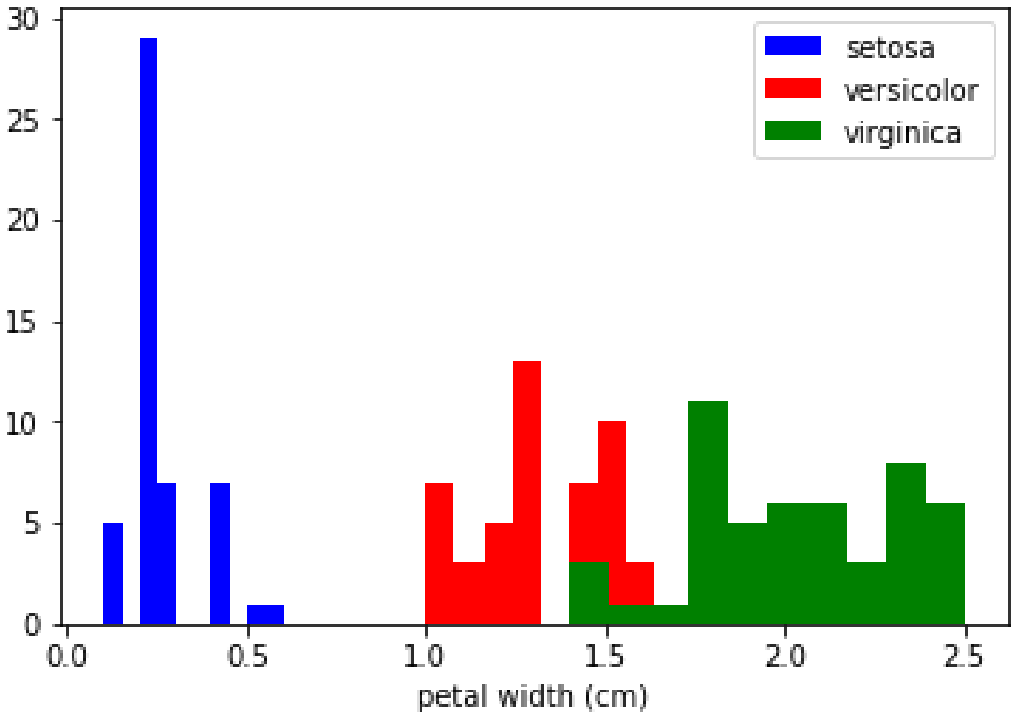
```
(150, 4)
(150,)
```

4.0.2 Visualisierung der Merkmale des Iris-Datensatzes

Diese Daten sind vierdimensional, aber wir können eine oder zwei der Dimensionen visualisieren zu einem Zeitpunkt mit einem einfachen Histogramm oder Streudiagramm.

Histograms of the features

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
x_index = 3
colors = ['blue', 'red', 'green']

for label, color in zip(range(len(iris.target_names)), colors):
    ax.hist(iris.data[iris.target==label, x_index],
            label=iris.target_names[label],
            color=color)

ax.set_xlabel(iris.feature_names[x_index])
ax.legend(loc='upper right')
fig.show()
```



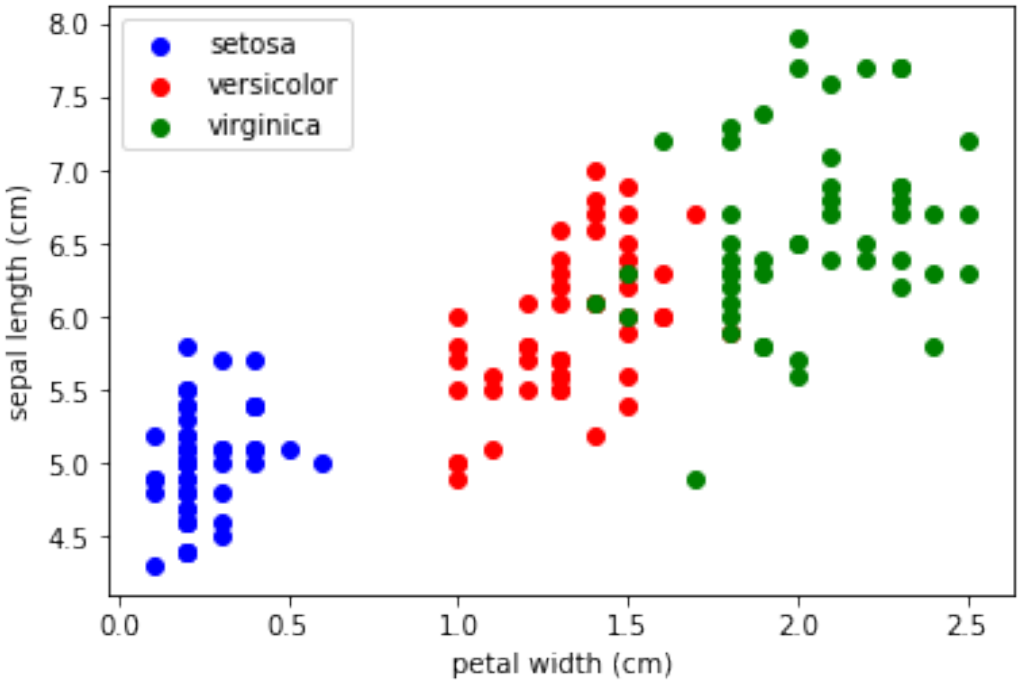
Übung

Im obigen Code haben wir x_index auf 3 gesetzt, das bedeutet, dass wir uns das Histogramm der Breite der Kronblätter (petals) erzeugt haben. Schauen Sie sich entsprechend die Histogramme der anderen Größen von den Kelch- (sepals) und Kronblättern an.

Streudiagramm mit zwei Merkmalen

Das nächste Diagramm zeigt zwei Merkmale in einem Diagramm:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()

x_index = 3
y_index = 0

colors = ['blue', 'red', 'green']

for label, color in zip(range(len(iris.target_names)), colors):
    ax.scatter(iris.data[iris.target==label, x_index],
                iris.data[iris.target==label, y_index],
                label=iris.target_names[label],
                c=color)

ax.set_xlabel(iris.feature_names[x_index])
ax.set_ylabel(iris.feature_names[y_index])
ax.legend(loc='upper left')
plt.show()
```

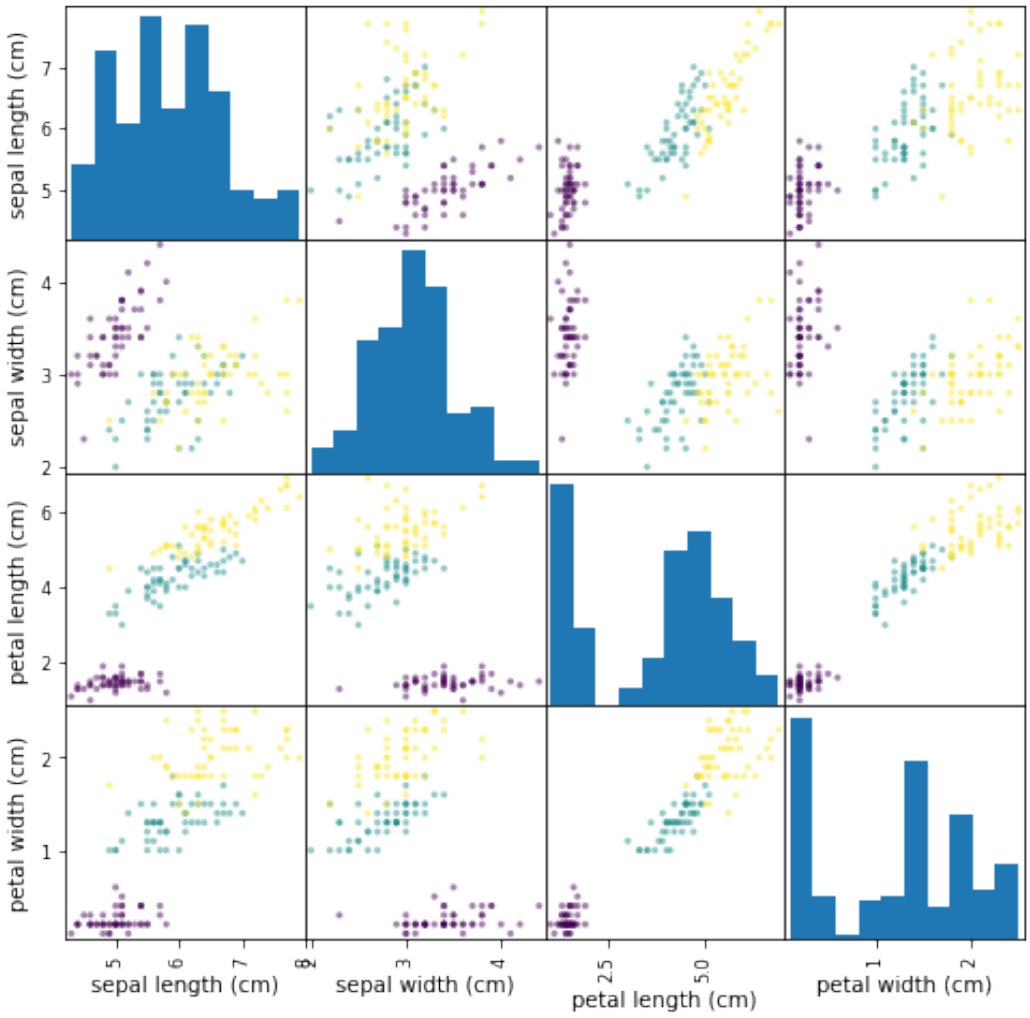


4.0.3 Streudiagramm Matrix

Anstatt die Daten jeweils als einzelne Plots zu betrachten, können wir ein gängiges von Analysten verwendetes Tool vewenden. Es erzeugt eine **Streudiagramm-Matrix** (Scatterplot-Matrix).

Streudiagramm-Matrizen zeigen Streudiagramme zwischen allen Features im Datensatz sowie Histogramme, um die Verteilung der einzelnen Features anzuzeigen.

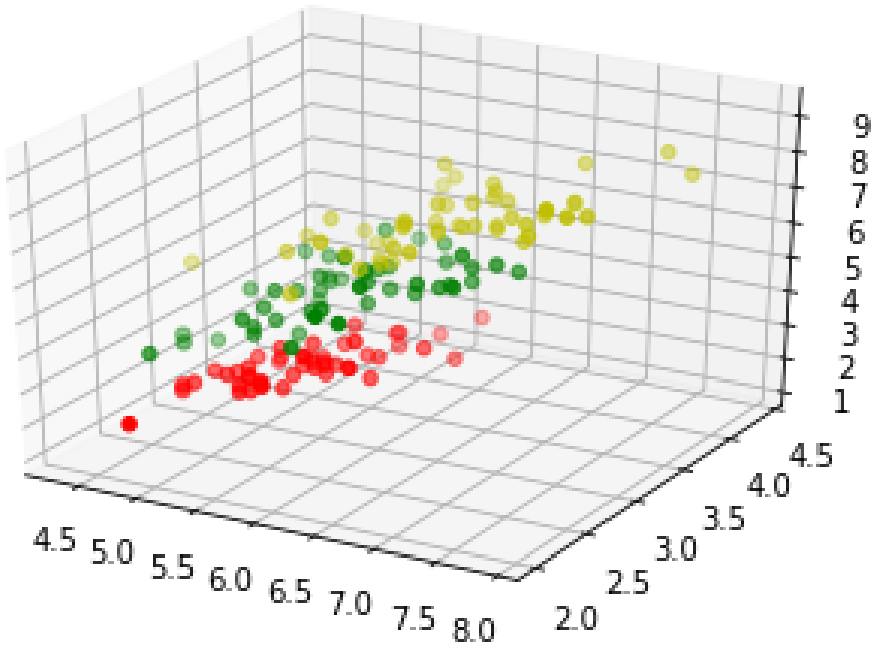
```
import pandas as pd

iris_df = pd.DataFrame(iris.data, columns=iris.feature_names)
pd.plotting.scatter_matrix(iris_df,
                           c=iris.target,
                           figsize=(8, 8)
                          );
```



3-Dimensionale Visualisierung

```
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from mpl_toolkits.mplot3d import Axes3D
iris = load_iris()
X = []
for iclass in range(3):
    X.append([[], [], []])
    for i in range(len(iris.data)):
        if iris.target[i] == iclass:
            X[iclass][0].append(iris.data[i][0])
```

```
            X[iclass][1].append(iris.data[i][1])
            X[iclass][2].append(sum(iris.data[i][2:]))

colours = ("r", "g", "y")
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

for iclass in range(3):
    ax.scatter(X[iclass][0], X[iclass][1], X[iclass][2], c=colours[iclass])
plt.show()
```



5 Datensaetze In Sklearn

5.0.1 Verfügbare Datensätze in scikit-learn

Scikit-learn stellt verschiedene Hilfen zum Laden und Erzeugen von Datensätzen bereit. Für Lern- und Demonstrationszwecke sind besonders drei Gruppen wichtig:

- **Toy-Datensätze:** kleine Standarddatensätze, die direkt mit scikit-learn ausgeliefert werden, beispielsweise Iris, Digits oder Wine.
- **Real-World-Datensätze:** größere Datensätze, die bei Bedarf mit `fetch_*`-Funktionen heruntergeladen werden.
- **Synthetische Datensätze:** mit Funktionen wie `make_blobs`, `make_classification` oder `make_regression` kontrolliert erzeugte Daten.

Loader und Fetcher liefern typischerweise ein `Bunch`-Objekt mit Attributen wie `data`, `target` und `DESCR`; viele Loader unterstützen außerdem `return_X_y=True` und `as_frame=True`.

Bei herunterladbaren Datensätzen können Download und Speicherbedarf deutlich größer sein als bei den eingebauten Toy-Datensätzen.

```
from sklearn import datasets
```

Warnung: Viele dieser Datasets sind sehr groß und das Herunterladen kann sehr lange dauern!

5.0.2 Digits-Datenset laden

Wir werden uns nun einen dieser Datensätze genauer ansehen. Wir schauen uns den Ziffern-Datensatz an. Wir laden ihn zuerst:

```
from sklearn.datasets import load_digits
digits = load_digits()
```

Wir können uns wieder einen Überblick über die verfügbaren Attribute verschaffen, indem wir uns die “keys” anschauen:

```
digits.keys()
```

```
dict_keys(['data', 'target', 'frame', 'feature_names', 'target_names', 'images',
↪  'DESCR'])
```

Werfen wir einen Blick auf die Anzahl der Elemente und Funktionen:

```
n_samples, n_features = digits.data.shape
print((n_samples, n_features))
```

```
(1797, 64)
```

```
print(digits.data[0])
print(digits.target)
```

```
[ 0.  0.  5. 13.  9.  1.  0.  0.  0.  0. 13. 15. 10. 15.  5.  0.  0.  3.
 15.  2.  0. 11.  8.  0.  0.  4. 12.  0.  0.  8.  8.  0.  0.  5.  8.  0.
  0.  9.  8.  0.  0.  4. 11.  0.  1. 12.  7.  0.  0.  2. 14.  5. 10. 12.
  0.  0.  0.  0.  6. 13. 10.  0.  0.  0.]
[0 1 2 ... 8 9 8]
```

Die Daten liegen auch unter digits.images vor. Dabei handelt es sich um die Rohdaten der Bilder in der Form 8 Zeilen und 8 Spalten.

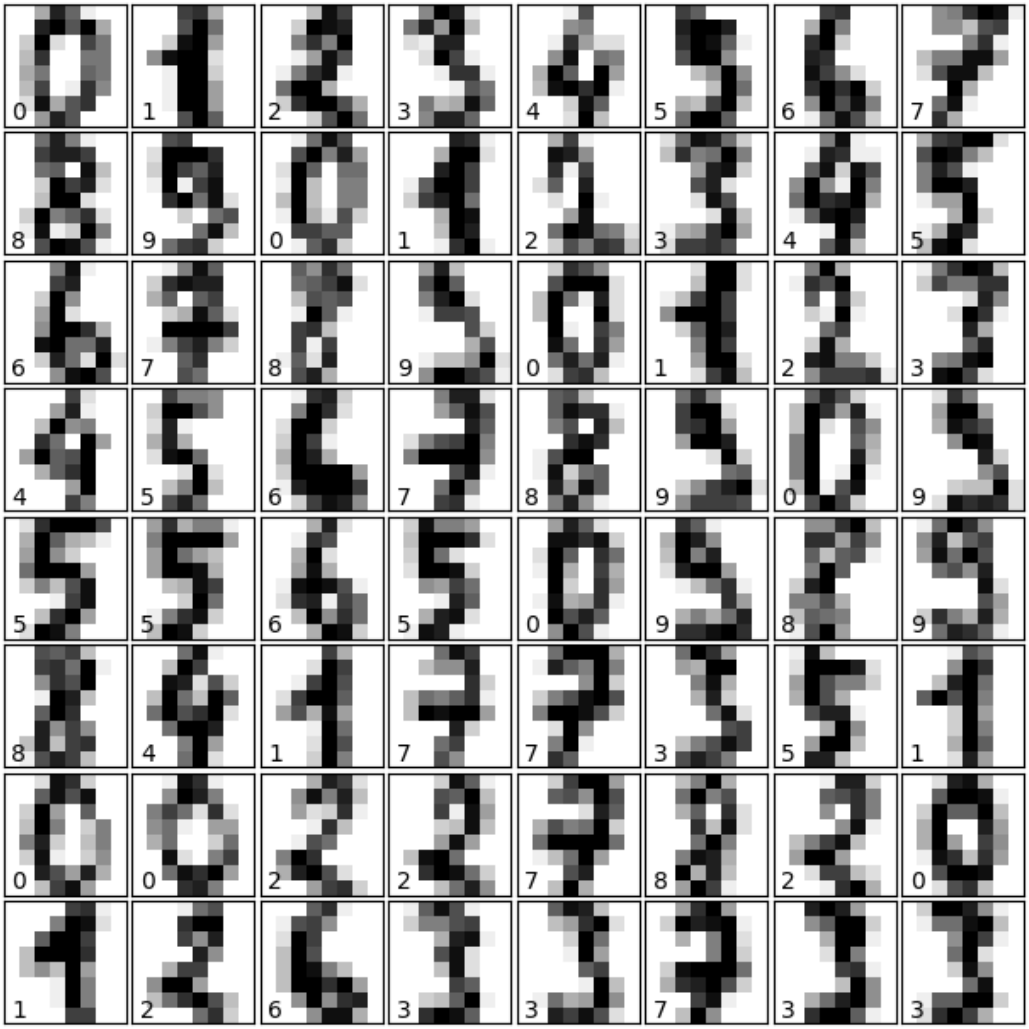
Bei “data” entspricht ein Bild einem eindimensionen Numpy-Array mit der Länge 64, und images enthält 2-dimensionale numpy-Arrays mit der Shape (8, 8)

```
print("Shape eines Items: ", digits.data[0].shape)
print("Datentype eines Items: ", type(digits.data[0]))
print("Shape eines Items: ", digits.images[0].shape)
print("Datentype eines Items: ", type(digits.images[0]))
```

```
Shape eines Items:  (64,)
Datentype eines Items:  <class 'numpy.ndarray'>
Shape eines Items:  (8, 8)
Datentype eines Items:  <class 'numpy.ndarray'>
```

Nun wollen wir die Daten visualisieren:

```
import matplotlib.pyplot as plt
# set up the figure
fig = plt.figure(figsize=(6, 6))  # figure size in inches
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)

# plot der Ziffern:
# jedes Bild besteht aus 8x8 Pixel.
for i in range(64):
    ax = fig.add_subplot(8, 8, i + 1, xticks=[], yticks=[])
    ax.imshow(digits.images[i], cmap=plt.cm.binary, interpolation='nearest')

    # label the image with the target value
    ax.text(0, 7, str(digits.target[i]))
```



Übungen

Aufgabe 1 sklearn enthält einen „Wein-Datensatz“.

- Suchen und laden Sie diesen Datensatz
- Können Sie eine Beschreibung finden?
- Wie heißen die Klassen und die Features?
- Wo sind die Daten und die Daten und die Labels?

Aufgabe 2 Erzeugen Sie einen scatter-Plot mit den Features `ash` und `color_intensity` des Wine-Datensatzes.

Aufgabe 3 Erzeugen Sie eine Scatter-Matrix für das Wine-Datenset

Aufgabe 4 Finden und laden Sie den Olivetti-Datensatz der Photos mit Gesichtern enthält und stellen Sie die Gesichter dar.

Lösungen

Lösung zu Aufgabe 1 Laden des “Wine”-Datensets:

```
from sklearn import datasets

wine = datasets.load_wine()
```

Die Beschreibung erhält man mittels:

```
print(wine.DESCR)
```

```
.. _wine_dataset:

Wine recognition dataset
------------------------

**Data Set Characteristics:**

:Number of Instances: 178
:Number of Attributes: 13 numeric, predictive attributes and the class
:Attribute Information:
    - Alcohol
    - Malic acid
    - Ash
    - Alcalinity of ash
    - Magnesium
    - Total phenols
    - Flavanoids
    - Nonflavanoid phenols
    - Proanthocyanins
    - Color intensity
    - Hue
    - OD280/OD315 of diluted wines
    - Proline
    - class:
        - class_0
        - class_1
        - class_2

:Summary Statistics:

============================== ==== ===== ======= =====
                                Min   Max   Mean     SD
============================== ==== ===== ======= =====
Alcohol:                      11.0  14.8    13.0   0.8
Malic Acid:                   0.74  5.80    2.34  1.12
Ash:                          1.36  3.23    2.36  0.27
Alcalinity of Ash:            10.6  30.0    19.5   3.3
Magnesium:                    70.0 162.0    99.7  14.3
Total Phenols:                0.98  3.88    2.29  0.63
Flavanoids:                   0.34  5.08    2.03  1.00
Nonflavanoid Phenols:         0.13  0.66    0.36  0.12
Proanthocyanins:              0.41  3.58    1.59  0.57
Colour Intensity:              1.3  13.0     5.1   2.3
Hue:                          0.48  1.71    0.96  0.23
```

```
OD280/OD315 of diluted wines: 1.27  4.00    2.61  0.71
Proline:                       278  1680     746   315
============================== ==== ===== ======= =====

:Missing Attribute Values: None
:Class Distribution: class_0 (59), class_1 (71), class_2 (48)
:Creator: R.A. Fisher
:Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
:Date: July, 1988

This is a copy of UCI ML Wine recognition datasets.
https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data

The data is the results of a chemical analysis of wines grown in the same
region in Italy by three different cultivators. There are thirteen different
measurements taken for different constituents found in the three types of
wine.

Original Owners:

Forina, M. et al, PARVUS -
An Extendible Package for Data Exploration, Classification and Correlation.
Institute of Pharmaceutical and Food Analysis and Technologies,
Via Brigata Salerno, 16147 Genoa, Italy.

Citation:

Lichman, M. (2013). UCI Machine Learning Repository
[https://archive.ics.uci.edu/ml]. Irvine, CA: University of California,
School of Information and Computer Science.

.. dropdown:: References

    (1) S. Aeberhard, D. Coomans and O. de Vel,
    Comparison of Classifiers in High Dimensional Settings,
    Tech. Rep. no. 92-02, (1992), Dept. of Computer Science and Dept. of
    Mathematics and Statistics, James Cook University of North Queensland.
    (Also submitted to Technometrics).

    The data was used with many others for comparing various
    classifiers. The classes are separable, though only RDA
    has achieved 100% correct classification.
    (RDA : 100%, QDA 99.4%, LDA 98.9%, 1NN 96.1% (z-transformed data))
    (All results using the leave-one-out technique)

    (2) S. Aeberhard, D. Coomans and O. de Vel,
    "THE CLASSIFICATION PERFORMANCE OF RDA"
    Tech. Rep. no. 92-01, (1992), Dept. of Computer Science and Dept. of
    Mathematics and Statistics, James Cook University of North Queensland.
    (Also submitted to Journal of Chemometrics).
```

Die Namen der Klassen und der Features erhalten wir mit den Attributen “target_names” und “feature_names”:

```
print(wine.target_names)
print(wine.feature_names)
```

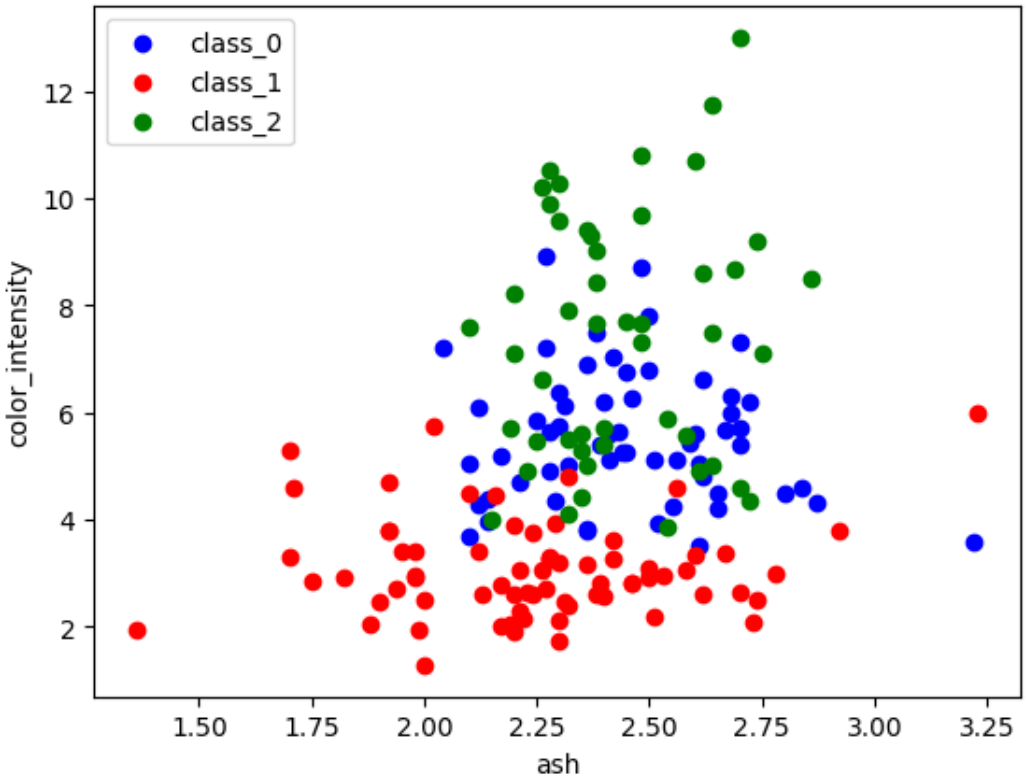
```
['class_0' 'class_1' 'class_2']
```

```
['alcohol', 'malic_acid', 'ash', 'alcalinity_of_ash', 'magnesium',
↪  'total_phenols', 'flavanoids', 'nonflavanoid_phenols', 'proanthocyanins',
↪  'color_intensity', 'hue', 'od280/od315_of_diluted_wines', 'proline']
```

```
daten = wine.data
gelabelte_daten = wine.target
```

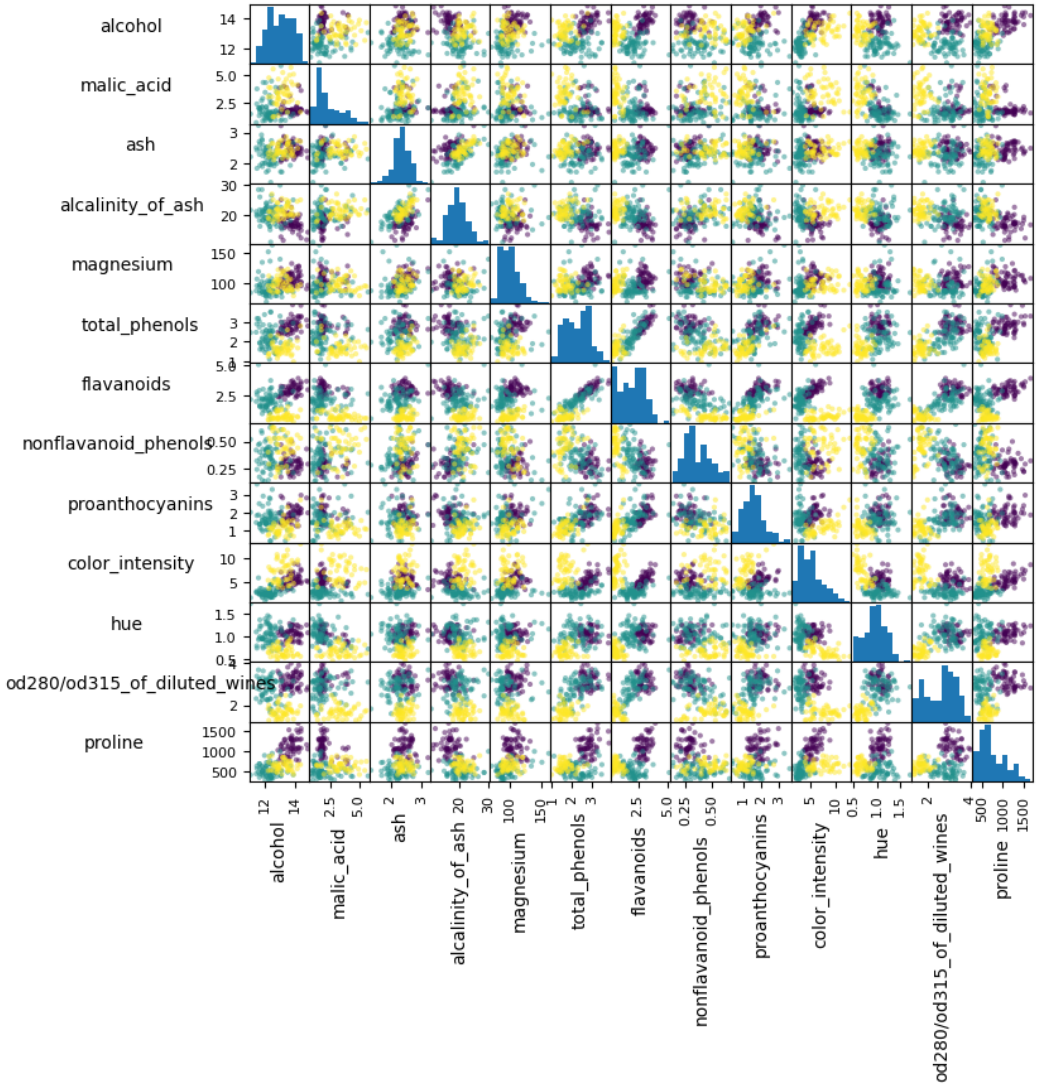
Lösung zu Aufgabe 2

```
from sklearn import datasets
import matplotlib.pyplot as plt

wine = datasets.load_wine()

features = 'ash', 'color_intensity'
features_index = [wine.feature_names.index(features[0]),
                  wine.feature_names.index(features[1])]


colors = ['blue', 'red', 'green']

for label, color in zip(range(len(wine.target_names)), colors):
    plt.scatter(wine.data[wine.target==label, features_index[0]],
                wine.data[wine.target==label, features_index[1]],
                label=wine.target_names[label],
                c=color)

plt.xlabel(features[0])
plt.ylabel(features[1])
plt.legend(loc='upper left')
plt.show()
```



Lösung zu Aufgabe 3

```
import pandas as pd
from sklearn import datasets

wine = datasets.load_wine()
def rotate_labels(df, axes):
    """ changing the rotation of the label output,
    y labels horizontal and x labels vertical """
    n = len(df.columns)
    for x in range(n):
        for y in range(n):
            # to get the axis of subplots
            ax = axs[x, y]
            # to make x axis name vertical
            ax.xaxis.label.set_rotation(90)
            # to make y axis name horizontal
            ax.yaxis.label.set_rotation(0)
            # to make sure y axis names are outside the plot area
            ax.yaxis.labelpad = 50

wine_df = pd.DataFrame(wine.data, columns=wine.feature_names)
axs = pd.plotting.scatter_matrix(wine_df,
                                 c=wine.target,
                                 figsize=(8, 8),
                                );

rotate_labels(wine_df, axs)
```



Lösung zur Aufgabe 4

```
from sklearn.datasets import fetch_olivetti_faces
```

```
# Laden des Datensatzes
faces = fetch_olivetti_faces()
```

```
downloading Olivetti faces from https://ndownloader.figshare.com/files/5976027 to
↪  /home/bernd/scikit_learn_data
```

```
faces.keys()
```

```
dict_keys(['data', 'images', 'target', 'DESCR'])
```

```
n_samples, n_features = faces.data.shape
print((n_samples, n_features))
```

```
(400, 4096)
```

```
np.sqrt(4096)
```

```
64.0
```

```
faces.images.shape
```

```
(400, 64, 64)
```

```
faces.data.shape
```

```
(400, 4096)
```

```
print(np.all(faces.images.reshape((400, 4096)) == faces.data))
```

```
True
```

```
# set up the figure
fig = plt.figure(figsize=(6, 6))  # figure size in inches
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)

# plot the digits: each image is 8x8 pixels
for i in range(64):
    ax = fig.add_subplot(8, 8, i + 1, xticks=[], yticks=[])
    ax.imshow(faces.images[i], cmap=plt.cm.bone, interpolation='nearest')

    # label the image with the target value
    ax.text(0, 7, str(faces.target[i]))
```

Weitere Datensätze

sklearn verfügt über viele weitere Datensätze. Wenn Sie noch weitere benötigen, finden Sie mehr in dieser interessanten Liste von Datensätzen bei Wikipedia.

6 Synthetische Daten Erzeugen Maschinelles Lernen

6.0.1 Erzeugung synthetischer Daten

Synthetische Daten mit Python

Ein Problem des maschinellen Lernens, - insbesondere wenn man sich am Einarbeiten ist mehr über die Algorithmen erfahren möchte, - besteht darin, dass es oft schwierig ist, geeignete Testdaten zu erhalten. Einige kosten viel Geld, andere sind nicht frei verfügbar, weil sie urheberrechtlich geschützt sind. Deshalb können künstlich erzeugte Testdaten in einigen Fällen eine Lösung sein.

Aus diesem Grund befasst sich dieses Kapitel unseres Tutorials mit der künstlichen Datenerzeugung. In diesem Kapitel geht es darum, künstliche Daten zu erstellen. In den vorherigen Kapiteln unseres Tutorials haben wir gelernt, dass Scikit-Learn (sklearn) verschiedene Datensätze enthält. Einerseits gibt es kleine Spielzeug-Datensätze, andererseits bietet dieses Modul auch größere Datensätze, die häufig verwendet werden, um Algorithmen zu testen oder auch als Benchmark zu dienen. `sklearn` bietet uns Daten, die aus der "realen Welt" stammen.

All dies ist großartig, aber in vielen Fällen reicht dies immer noch nicht aus. Vielleicht findet man die richtigen Art von Daten, aber man benötigt mehr Daten dieser Art, oder die Daten entsprechen nicht vollständig dem, was man sucht. So benötigt man beispielsweise komplexere oder weniger komplexere Daten. Dies ist der Punkt, an dem man in Betracht ziehen sollte, die Daten selbst zu erstellen. Hier bietet `sklearn` Hilfe an. Es enthält verschiedene random-Beispielgeneratoren, mit denen benutzerdefinierte künstliche Datasets erstellt werden können. Datasets, die den eigenen Ideen von Größe und Komplexität entsprechen.

Der folgende Python-Code ist ein einfaches Beispiel, eines welches noch nicht `sklearn` benutzt. Wir erzeugen künstliche Wetterdaten für einige deutsche Städte. Dazu verwenden wir Pandas und Numpy, um die Daten zu erstellen:

```
import numpy as np
import pandas as pd


cities = ['Berlin', 'Frankfurt', 'Hamburg',
          'Nuremberg', 'Munich', 'Stuttgart',
          'Hanover', 'Saarbruecken', 'Cologne',
          'Constance', 'Freiburg', 'Karlsruhe'
         ]

n= len(cities)
data = {'Temperature': np.random.normal(24, 3, n),
        'Humidity': np.random.normal(78, 2.5, n),
        'Wind': np.random.normal(15, 4, n)
```

```
        }
df = pd.DataFrame(data=data, index=cities)
df
```

	Temperature	Humidity	Wind
Berlin	22.320235	80.220367	23.039794
Frankfurt	25.580538	78.025427	13.852446
Hamburg	29.184371	80.720426	5.178373
Nuremberg	24.162373	77.096790	21.191182
Munich	20.491273	79.605320	18.713445
Stuttgart	18.519613	82.019420	15.877972
Hanover	21.291383	80.000568	10.231694
Saarbruecken	27.821346	79.696954	10.818042
Cologne	22.364220	74.496690	14.037689
Constance	26.309676	81.463110	14.709574
Freiburg	18.229215	76.792430	13.387500
Karlsruhe	26.264173	75.027791	12.973147

Ein weiteres Beispiel

Wir erzeugen nun synthetische Daten für vier nicht existierende Blumensorten:

- Flos Pythonem
- Flos Java
- Flos Margarita
- Flos artificialis

Die durchschnittlichen RGB-Farbwerte entsprechen folgenden Werten: - (255, 0, 0) - (245, 107, 0) - (206, 99, 1) - (255, 254, 101)

Der durchschnittliche Durchmesser des Calyx ist: - 3.8 - 3.3 - 4.1 - 2.9

Flos pythonem(254, 0, 0)

Flos Java(245, 107, 0)

Flos margarita(206, 99, 1)

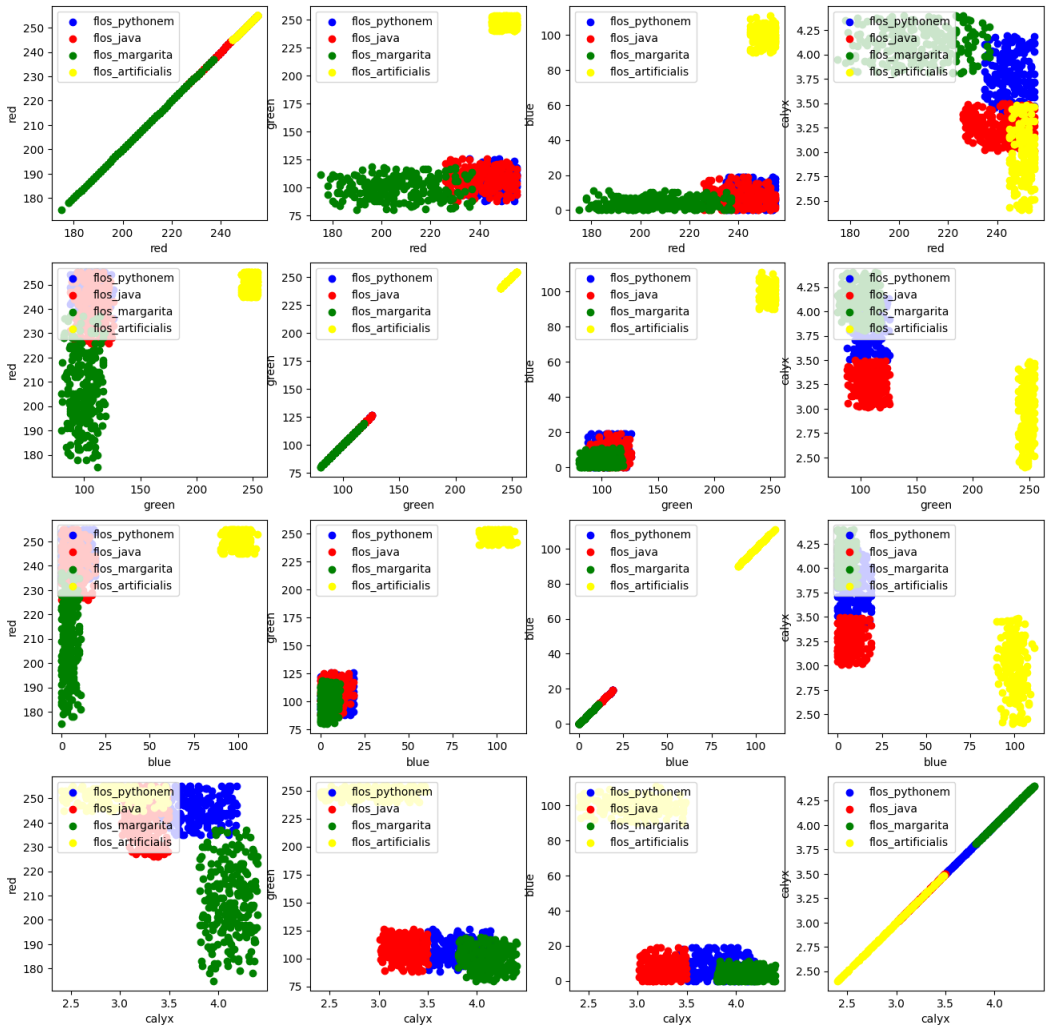
Flos artificialis(255, 254, 101)

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from scipy.stats import truncnorm

def truncated_normal(mean=0, sd=1, low=0, upp=10, type=int):
    return truncnorm(
        (low - mean) / sd, (upp - mean) / sd, loc=mean, scale=sd)

def truncated_normal_floats(mean=0, sd=1, low=0, upp=10, num=100):
    res = truncated_normal(mean=mean, sd=sd, low=low, upp=upp)
    return res.rvs(num)

def truncated_normal_ints(mean=0, sd=1, low=0, upp=10, num=100):
    res = truncated_normal(mean=mean, sd=sd, low=low, upp=upp)
    return res.rvs(num).astype(np.uint8)
```

```
# number of items for each flower class:
number_of_items_per_class = [190, 205, 230, 170]
flowers = {}
# flos Pythonem:
number_of_items = number_of_items_per_class[0]
reds = truncated_normal_ints(mean=254, sd=18, low=235, upp=256,
                             num=number_of_items)
greens = truncated_normal_ints(mean=107, sd=11, low=88, upp=127,
                             num=number_of_items)
blues = truncated_normal_ints(mean=0, sd=15, low=0, upp=20,
                             num=number_of_items)
calyx_dia = truncated_normal_floats(3.8, 0.3, 3.4, 4.2,
                             num=number_of_items)
data = np.column_stack((reds, greens, blues, calyx_dia))
flowers["flos_pythonem"] = data

# flos Java:
number_of_items = number_of_items_per_class[1]
reds = truncated_normal_ints(mean=245, sd=17, low=226, upp=256,
                             num=number_of_items)
greens = truncated_normal_ints(mean=107, sd=11, low=88, upp=127,
                             num=number_of_items)
blues = truncated_normal_ints(mean=0, sd=10, low=0, upp=20,
                             num=number_of_items)
calyx_dia = truncated_normal_floats(3.3, 0.3, 3.0, 3.5,
                             num=number_of_items)
data = np.column_stack((reds, greens, blues, calyx_dia))
flowers["flos_java"] = data

# flos Margarita:
number_of_items = number_of_items_per_class[2]
reds = truncated_normal_ints(mean=206, sd=17, low=175, upp=238,
                             num=number_of_items)
greens = truncated_normal_ints(mean=99, sd=14, low=80, upp=120,
                             num=number_of_items)
blues = truncated_normal_ints(mean=1, sd=5, low=0, upp=12,
                             num=number_of_items)
calyx_dia = truncated_normal_floats(4.1, 0.3, 3.8, 4.4,
                             num=number_of_items)
data = np.column_stack((reds, greens, blues, calyx_dia))
flowers["flos_margarita"] = data

# flos artificialis:
number_of_items = number_of_items_per_class[3]
reds = truncated_normal_ints(mean=255, sd=8, low=245, upp=256,
                             num=number_of_items)
greens = truncated_normal_ints(mean=254, sd=10, low=240, upp=255,
                             num=number_of_items)
blues = truncated_normal_ints(mean=101, sd=5, low=90, upp=112,
                             num=number_of_items)
calyx_dia = truncated_normal_floats(2.9, 0.4, 2.4, 3.5,
                             num=number_of_items)
data = np.column_stack((reds, greens, blues, calyx_dia))
flowers["flos_artificialis"] = data
```

```
data = np.concatenate((flowers["flos_pythonem"],
                       flowers["flos_java"],
                       flowers["flos_margarita"],
                       flowers["flos_artificialis"]
                      ), axis=0)

# assigning the labels
target = np.zeros(sum(number_of_items_per_class)) # 4 flowers
previous_end = 0
for i in range(1, 5):
    num = number_of_items_per_class[i-1]
    beg = previous_end
    target[beg: beg + num] += i
    previous_end = beg + num

conc_data = np.concatenate((data, target.reshape(target.shape[0], 1)),
                           axis=1)

from pathlib import Path
Path("data").mkdir(exist_ok=True)
np.savetxt("data/strange_flowers.txt", conc_data, fmt="%2.2f")
```

```
import matplotlib.pyplot as plt

target_names = list(flowers.keys())
feature_names = ['red', 'green', 'blue', 'calyx']
n = 4
fig, ax = plt.subplots(n, n, figsize=(16, 16))

colors = ['blue', 'red', 'green', 'yellow']

for x in range(n):
    for y in range(n):
        xname = feature_names[x]
        yname = feature_names[y]
        for color_ind in range(1, len(target_names)+1):
            ax[x, y].scatter(data[target==color_ind, x],
                             data[target==color_ind, y],
                             label=target_names[color_ind-1],
                             c=colors[color_ind-1])

        ax[x, y].set_xlabel(xname)
        ax[x, y].set_ylabel(yname)
        ax[x, y].legend(loc='upper left')


plt.show()
```



Synthetische Daten mit sklearn erzeugen

Deutlich einfacher gestaltet sich die Erzeugung von synthetischen Daten mit dem Modul `sklearn`.

Die in `sklearn` verfügbaren Funktionalitäten können wir folgt gruppiert werden:

1. Generatoren zur Klassifikation und Clustering
2. Generatoren zur Erzeugung von Daten zur Regression
3. Generatoren für "Manifold Learning"
4. Generatoren für Zerlegungs-Probleme (Decomposition)

Generatoren zur Klassifikation und Clustering Wir starten mit der Funktion `make_blobs` von `sklearn.datasets` um Klecks-ähnliche (englisch: blob) Daten-Strukturen zu erzeugen. Indem wir den Wert von `centers` auf `n_classes` setzen, bestimmen wir die Anzahl der Blobs, d. h. der Cluster. `n_samples` entspricht der Anzahl der Datenpunkte, gleichverteilt über alle Klassen. Falls `random_state` nicht gesetzt ist, werden wir jedesmal, wenn wir die Funktion aufrufen, andere Zufallwerte erhalten. Wir weisen diesem Parameter eine ganze Zahl zu, um reproduzierbare Werte zu erzeugen.

```
from sklearn.datasets import make_blobs
```

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

data, labels = make_blobs(
    n_samples=1000,
    centers=np.array([[2, 3], [4, 5], [7, 9]]),
    random_state=1,
)

# Für das Speichern hängen wir die Labels nur temporär als Spalte an.
# `labels` selbst bleibt eindimensional, wie es scikit-learn erwartet.
labels_column = labels[:, np.newaxis]
all_data = np.concatenate((data, labels_column), axis=1)
np.savetxt("squirrels.txt", all_data)
all_data[:10]
```

```
array([[ 1.72415394,  4.22895559,  0.        ],
       [ 4.16466507,  5.77817418,  1.        ],
       [ 4.51441156,  4.98274913,  1.        ],
       [ 1.49102772,  2.83351405,  0.        ],
       [ 6.0386362 ,  7.57298437,  2.        ],
       [ 5.61044976,  9.83428321,  2.        ],
       [ 5.69202866, 10.47239631,  2.        ],
       [ 6.14017298,  8.56209179,  2.        ],
       [ 2.97620068,  5.56776474,  1.        ],
       [ 8.27980017,  8.54824406,  2.        ]])
```

`reshape` und `concatenate` im kleinen Beispiel Im vorherigen Beispiel werden die von `make_blobs` erzeugten Klassenlabels zunächst zu einer Spalte umgeformt und anschließend mit den Merkmalsdaten verbunden. Das folgende Minimalbeispiel zeigt die beiden Schritte ohne weiteren Kontext:

```
import numpy as np
a = np.array( [[1, 2], [3, 4]])
b = np.array( [5, 6])
b = b.reshape((b.shape[0], 1))
print(b)
x = np.concatenate( (a, b), axis=1)
x
```

```
[[5]
 [6]]

array([[1, 2, 5],
       [3, 4, 6]])
```

Daten wieder einlesen und Merkmale und Labels trennen

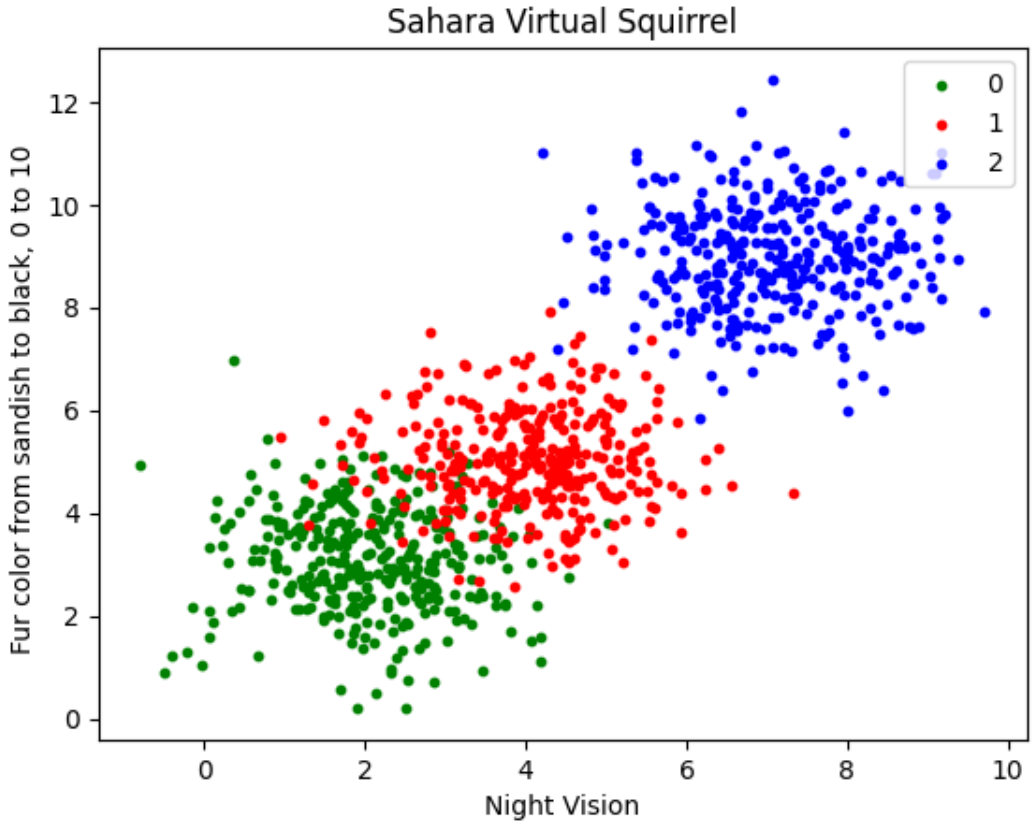
Die zuvor gespeicherten Daten können mit `numpy.loadtxt` wieder eingelesen werden. Anschließend trennen wir die Merkmalsmatrix `data` und den Zielvektor `labels` wieder voneinander:

```
file_data = np.loadtxt("squirrels.txt")
data = file_data[:,:-1]
```

```
labels = file_data[:,2:]
labels = labels.reshape((labels.shape[0]))
```

```
import matplotlib.pyplot as plt
colours = ('green', 'red', 'blue', 'magenta', 'yellow', 'cyan')
n_classes = 3
fig, ax = plt.subplots()
for n_class in range(0, n_classes):
    ax.scatter(data[labels==n_class, 0], data[labels==n_class, 1],
               c=colours[n_class], s=10, label=str(n_class))
ax.set(xlabel='Night Vision',
       ylabel='Fur color from sandish to black, 0 to 10 ',
       title='Sahara Virtual Squirrel')
ax.legend(loc='upper right')
```

```
<matplotlib.legend.Legend at 0x7fc1037cfb10>
```



Trainings- und Testdaten bilden

Für ein Klassifikationsbeispiel teilen wir die synthetischen Daten mit `train_test_split` reproduzierbar in Trainings- und Testdaten. Danach trainieren wir einen `KNeighborsClassifier` und bestimmen seine Trefferquote auf den Testdaten:

```
from sklearn.model_selection import train_test_split
train_data, test_data, train_labels, test_labels = train_test_split(
    data, labels, test_size=0.2, random_state=42, stratify=labels
```

```
)
```

```
# import model
from sklearn.neighbors import KNeighborsClassifier
# create classifier
knn = KNeighborsClassifier(n_neighbors=8)
# train
knn.fit(train_data, train_labels)
# test on test data:
calculated_labels = knn.predict(test_data)
calculated_labels
```

```
array([1., 1., 0., 1., 1., 0., 2., 0., 1., 1., 0., 2., 0., 0., 2., 0., 0.,
       0., 2., 2., 0., 0., 1., 1., 1., 2., 1., 0., 2., 1., 2., 2., 0., 1.,
       0., 1., 1., 1., 0., 2., 0., 2., 1., 2., 0., 0., 1., 1., 0., 1., 1.,
       1., 2., 0., 1., 0., 2., 2., 0., 1., 2., 1., 1., 0., 2., 0., 1., 1.,
       2., 2., 2., 1., 2., 2., 0., 2., 2., 0., 0., 2., 2., 2., 2., 1., 0.,
       2., 1., 2., 1., 2., 2., 1., 0., 2., 2., 1., 2., 0., 2., 0., 1., 1.,
       1., 2., 1., 2., 0., 0., 2., 0., 0., 2., 2., 2., 0., 2., 2., 2., 2.,
       2., 1., 1., 2., 1., 0., 0., 0., 0., 0., 0., 0., 2., 0., 1., 1., 1.,
       1., 1., 2., 1., 2., 0., 0., 0., 0., 2., 1., 0., 0., 0., 0., 1., 1.,
       0., 0., 1., 2., 1., 1., 0., 2., 0., 1., 1., 0., 0., 2., 2., 1., 0.,
       0., 1., 2., 2., 2., 0., 2., 0., 1., 1., 0., 0., 2., 0., 1., 1., 0.,
       1., 2., 0., 2., 1., 2., 0., 2., 1., 0., 1., 1., 1.])
```

```
from sklearn import metrics
print("Accuracy:", metrics.accuracy_score(test_labels, calculated_labels))
```

```
Accuracy: 0.935
```

Weitere interessante synthetische Verteilungen

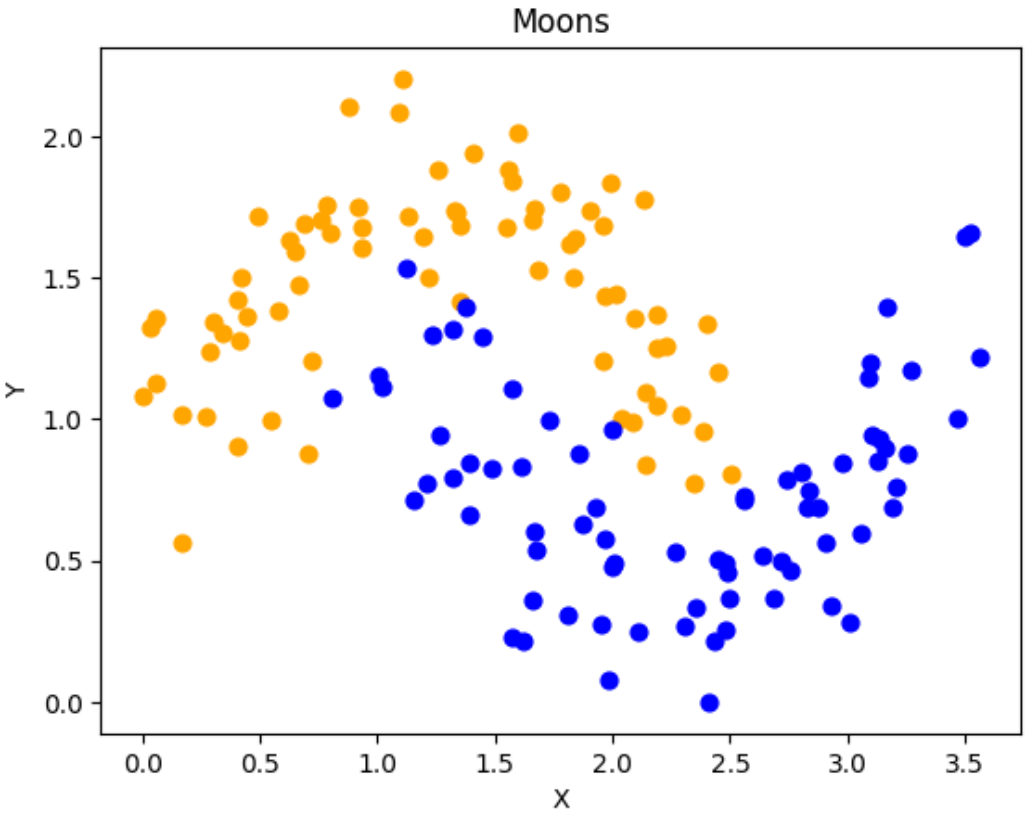
Neben Gaußschen Clustern bietet `sklearn.datasets` Generatoren für nichtlineare Strukturen. `make_moons` erzeugt zwei ineinandergreifende Halbmonde, `make_circles` zwei konzentrische Kreise. Solche Datensätze sind nützlich, um zu demonstrieren, wie Klassifikationsverfahren mit nichtlinearen Entscheidungsgrenzen umgehen.

```
import numpy as np
import sklearn.datasets as ds

data, labels = ds.make_moons(
    n_samples=150,
    shuffle=True,
    noise=0.19,
    random_state=42
)

# Verschiebe beide Koordinaten so, dass die kleinsten Werte bei 0 liegen.
data -= data.min(axis=0)
data.min(axis=0)
```

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

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ax.scatter(data[labels==0, 0], data[labels==0, 1],
               c='orange', s=40, label='oranges')
ax.scatter(data[labels==1, 0], data[labels==1, 1],
               c='blue', s=40, label='blues')
ax.set(xlabel='X',
       ylabel='Y',
       title='Moons')
#ax.legend(loc='upper right');
```

```
[Text(0.5, 0, 'X'), Text(0, 0.5, 'Y'), Text(0.5, 1.0, 'Moons')]
```



Wertebereiche skalieren

Sollen Werte aus einem Intervall `[min, max]` linear auf ein neues Intervall `[a, b]` abgebildet werden, verwenden wir

$$f(x) = \frac{(b-a)(x-min)}{max-min} + a.$$

Das folgende Beispiel wendet diese Transformation auf beide Koordinaten eines `make_moons`-Datensatzes an:

```
min_x_new, max_x_new = 33, 88
```

```
min_y_new, max_y_new = 12, 20
data, labels = ds.make_moons(n_samples=100,
                             shuffle=True,
                             noise=0.05,
                             random_state=None)
min_x, min_y = np.ndarray.min(data[:,0]), np.ndarray.min(data[:,1])
max_x, max_y = np.ndarray.max(data[:,0]), np.ndarray.max(data[:,1])
#data -= np.array([min_x, 0])
#data *= np.array([(max_x_new - min_x_new) / (max_x - min_x), 1])
#data += np.array([min_x_new, 0])
#data -= np.array([0, min_y])
#data *= np.array([1, (max_y_new - min_y_new) / (max_y - min_y)])
#data += np.array([0, min_y_new])
data -= np.array([min_x, min_y])
data *= np.array([(max_x_new - min_x_new) / (max_x - min_x), (max_y_new -
↪  min_y_new) / (max_y - min_y)])
data += np.array([min_x_new, min_y_new])
#np.ndarray.min(data[:,0]), np.ndarray.max(data[:,0])
data[:6]
```

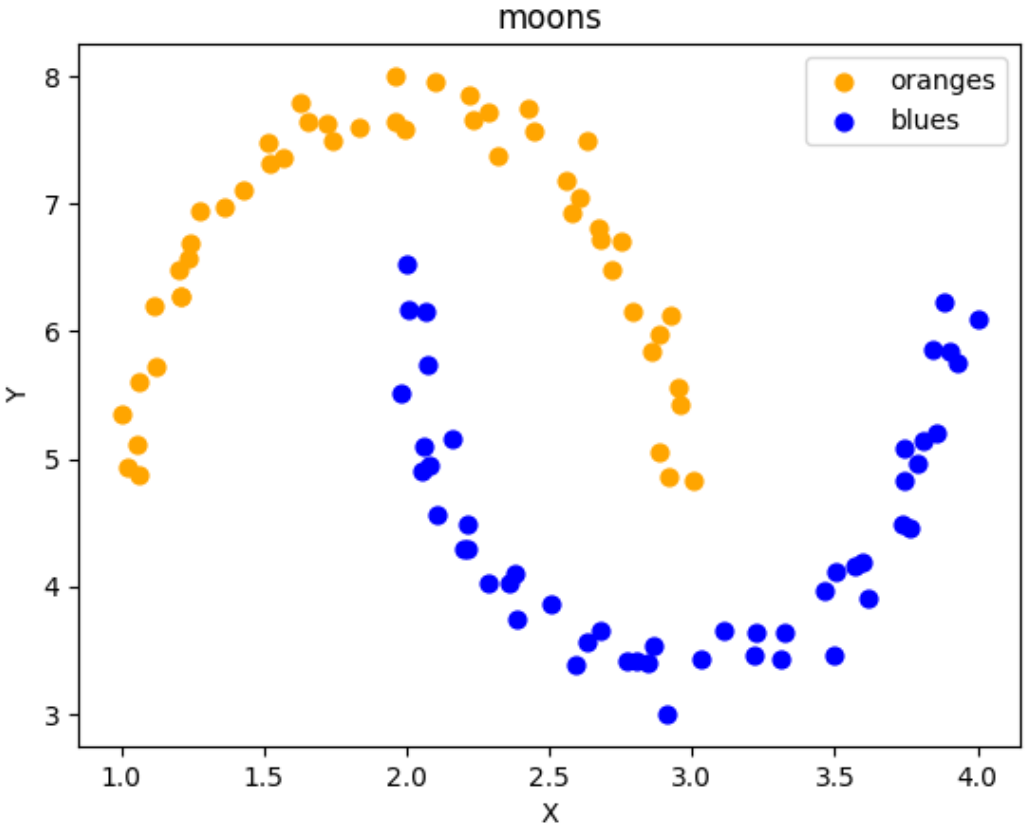
```
array([[84.75580656, 15.53468745],
       [34.09401859, 15.67229095],
       [37.75927425, 18.01338358],
       [76.37107584, 13.2117179 ],
       [50.37364723, 17.25049973],
       [63.70538119, 18.5262856 ]])
```

```
def scale_data(data, new_limits, inplace=False ):
    if not inplace:
        data = data.copy()
    min_x, min_y = np.ndarray.min(data[:,0]), np.ndarray.min(data[:,1])
    max_x, max_y = np.ndarray.max(data[:,0]), np.ndarray.max(data[:,1])
    min_x_new, max_x_new = new_limits[0]
    min_y_new, max_y_new = new_limits[1]
    data -= np.array([min_x, min_y])
    data *= np.array([(max_x_new - min_x_new) / (max_x - min_x), (max_y_new -
    ↪  min_y_new) / (max_y - min_y)])
    data += np.array([min_x_new, min_y_new])
    if inplace:
        return None
    else:
        return data


data, labels = ds.make_moons(n_samples=100,
                             shuffle=True,
                             noise=0.05,
                             random_state=None)
scale_data(data, [(1, 4), (3, 8)], inplace=True)
data[:10]
```

```
array([[1.62803808, 7.79775805],
       [2.50808739, 3.86021853],
       [2.95293319, 5.55644351],
       [2.31749794, 7.37316116],
       [3.92818055, 5.75614403],
```

```
       [2.00418923, 6.1702033 ],
       [3.00777824, 4.82584303],
       [4.        , 6.09192639],
       [1.11976711, 5.72520204],
       [1.65727635, 7.64797219]])
```

```
fig, ax = plt.subplots()
ax.scatter(data[labels==0, 0], data[labels==0, 1],
               c='orange', s=40, label='oranges')
ax.scatter(data[labels==1, 0], data[labels==1, 1],
               c='blue', s=40, label='blues')
ax.set(xlabel='X',
       ylabel='Y',
       title='moons')

ax.legend(loc='upper right');
```

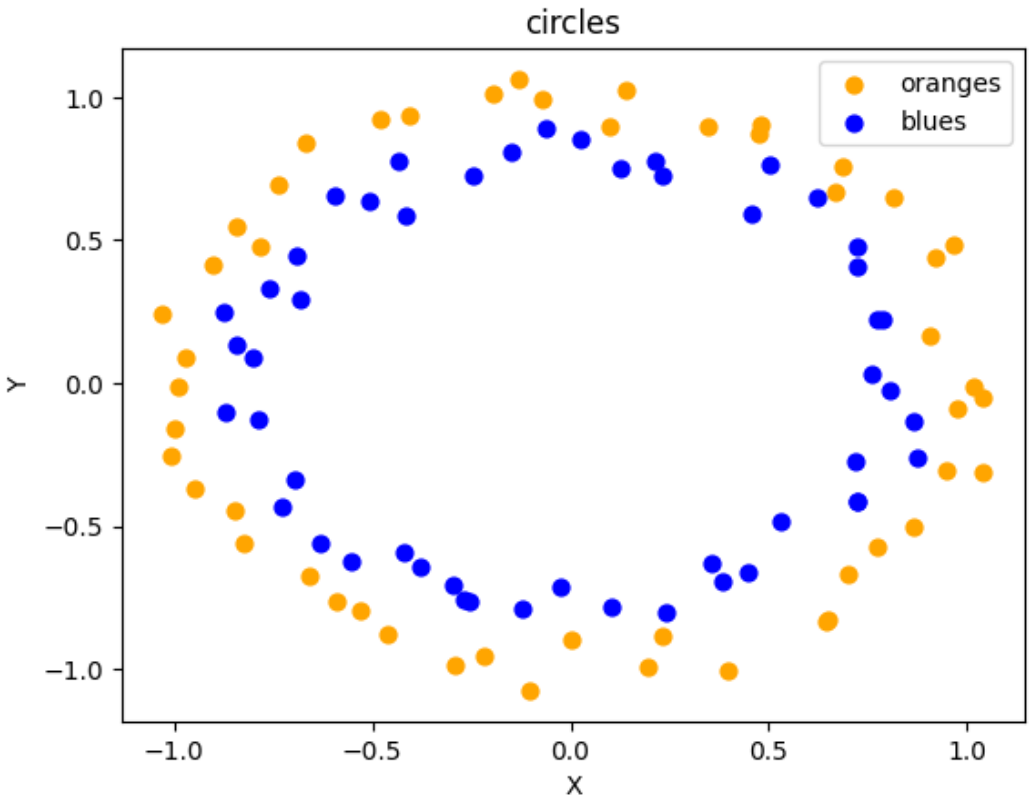



```
import sklearn.datasets as ds
data, labels = ds.make_circles(n_samples=100,
                               shuffle=True,
                               noise=0.05,
                               random_state=None)
```

```
fig, ax = plt.subplots()
ax.scatter(data[labels==0, 0], data[labels==0, 1],
               c='orange', s=40, label='oranges')
ax.scatter(data[labels==1, 0], data[labels==1, 1],
```

```
                c='blue', s=40, label='blues')
ax.set(xlabel='X',
       ylabel='Y',
       title='circles')
ax.legend(loc='upper right')
```

```
<matplotlib.legend.Legend at 0x7fc102940cd0>
```



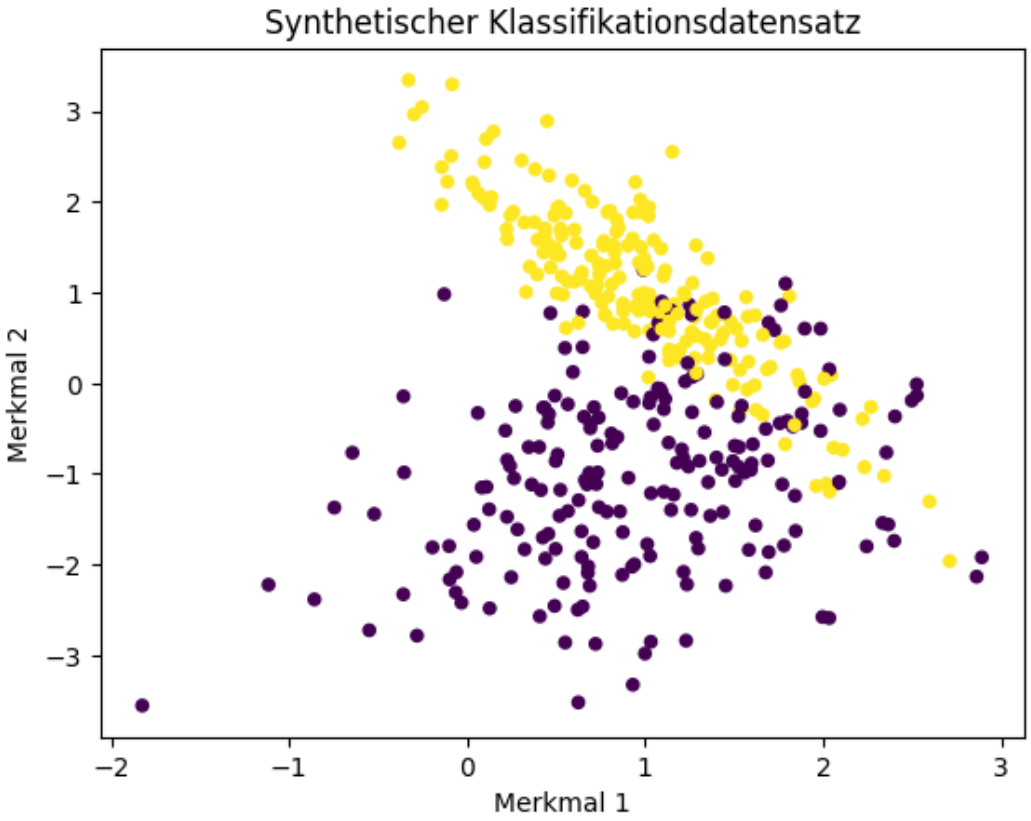
make_classification

Mit `make_classification` lassen sich kontrollierte Klassifikationsprobleme mit einer wählbaren Zahl informativer und redundanter Merkmale erzeugen:

```
from sklearn.datasets import make_classification
import matplotlib.pyplot as plt

X, y = make_classification(
    n_samples=400,
    n_features=2,
    n_redundant=0,
    n_informative=2,
    n_clusters_per_class=1,
    random_state=42
)

fig, ax = plt.subplots()
ax.scatter(X[:, 0], X[:, 1], c=y, s=20)
```

```
ax.set(title="Synthetischer Klassifikationsdatensatz", xlabel="Merkmal 1",
↪  ylabel="Merkmal 2")
```

```
[Text(0.5, 1.0, 'Synthetischer Klassifikationsdatensatz'),
 Text(0.5, 0, 'Merkmal 1'),
 Text(0, 0.5, 'Merkmal 2')]
```



7 Maschinelles Lernen Datensaetze

7.0.1 Aufteilungen der Daten

Datensets in Lern- und Testsets trennen

Sie haben Ihre Daten bereit und möchten den Klassifikator trainieren? Aber seien Sie vorsichtig: Wenn Ihr Klassifikator fertig ist, benötigen Sie einige Testdaten, um Ihren Klassifikator zu bewerten. Wenn Sie Ihren Klassifikator mit den zum Lernen verwendeten Daten bewerten, sehen Sie möglicherweise überraschend gute Ergebnisse. Was wir tatsächlich testen möchten, ist die Leistung der Klassifizierung auf unbekannten Daten.

Zu diesem Zweck müssen wir unsere Daten in zwei Teile aufteilen:

1. Einen Trainingssatz, mit dem der Lernalgorithmus das Modell anpasst oder lernt
2. Ein Testset zur Bewertung der Generalisierungsleistung des Modells

Wenn man bedenkt, wie das maschinelle Lernen normalerweise funktioniert, ist die Idee einer Aufteilung zwischen Lern- und Testdaten sinnvoll. Real existierende Systeme trainieren auf existierenden Daten und wenn dann andere also neue Daten (von Kunden, Sensoren oder anderen Quellen) kommen, muss der trainierte Klassifikator diese neue Daten vorhersagen bzw. klassifizieren. Wir können dies während des Trainings mit einem Trainings- und Testdatenset simulieren - die Testdaten sind eine Simulation von "zukünftigen Daten", die während der Produktion in das System eingehen werden.

In diesem Kapitel unseres Tutorials über Maschinelles Lernen mit Python erfahren Sie, wie Sie die Aufspaltung mit einfachem Python tun können. Danach werden wir zeigen, dass es nicht notwendig ist, dies manuell zu tun, da die Funktion `train_test_split` aus dem Modul `model_selection` dies für uns tun kann.

Wenn der Datensatz nach Labels sortiert ist, müssen wir ihn vor dem Teilen mischen.

Neben dem Trainingssatz wird häufig eine **Validierungsmenge** benötigt, um Modelle oder Hyperparameter auszuwählen. Der **Testsatz** sollte dagegen möglichst nur für die abschließende, unabhängige Bewertung verwendet werden. Bei kleinen und mittleren Datensätzen ersetzt Kreuzvalidierung auf dem Trainingsanteil oft eine feste Validierungsmenge. Wichtig ist die Rollenverteilung: Training zum Anpassen des Modells, Validierung/Kreuzvalidierung zur Modellwahl und Testdaten für die abschließende Generalisierungsbewertung.

In unserem Tutorial werden wir jedoch nur Aufteilungen in Lern- und Testdatensätze verwenden.

Aufteilung des Iris-Datensatzes

Wir werden die zuvor besprochenen Themen anhand des Iris-Datensatzes demonstrieren.

Die 150 Datensätze des Iris-Datensatzes sind sortiert, d.h. die ersten 50 Daten entsprechen der ersten Blumeklasse (0 = Setosa), die nächsten 50 der zweiten Blumenklasse (1 =

Versicolor) und die restlichen Daten entsprechen der letzten Klasse (2 = Virginica).

Würden wir nun unsere Daten im Verhältnis 2/3 (Lernset) und 1/3 (Testset) aufteilen, enthielte das Lernset alle Blumen der beiden ersten Klassen und das Testset alle Blumen der dritten Klasse. Der KLassifikator könnte also nur zwei Klassen lernen und die dritte Klasse wäre komplett unbekannt. Deshalb müssen wir die Daten dringend mischen.

Unter der Annahme, dass alle Stichproben unabhängig voneinander sind, möchten wir den Datensatz **zufällig mischen, bevor wir den Datensatz** wie oben dargestellt aufteilen.

Im Folgenden teilen wir die Daten manuell auf:

```
import numpy as np
from sklearn.datasets import load_iris
iris = load_iris()
```

Wenn wir uns die Labels anschauen, sehen wir, dass die Daten, wie eben beschrieben, sortiert sind:

```
iris.target
```

```
array([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, 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])
```

Als erstes müssen wir die Daten neu anordnen, damit sie nicht mehr sortiert werden. Dazu benutzen wir die permutation-Funktion des random-Submoduls von Numpy:

```
rng = np.random.default_rng(42)
indices = rng.permutation(len(iris.data))
indices
```

```
array([ 70,  79, 134,   3, 123, 120,  81, 105, 130,  33, 124,   4,  24,
       111, 129,  25,   2,  20, 125,  85,  37, 143, 126,  91,   0,  87,
        78,  27,  82,  90,  94,  56,  96, 103,  52,  13, 114,  28, 131,
         5,  66, 116, 106, 135,  12,  89,  54,  68, 122,  46,  50, 101,
       144,  45,  86,  73,   7,  18,  51, 146,  72,  92,  38,   8, 100,
        98,  21,  71,  59,  65, 132, 108,  93,  26,  75, 133, 139,  74,
        10,  88,  29,  63,  55,  95,  42,  17,  39, 149,  53,  76, 107,
        40,  80,  69, 118,   1, 148, 104, 119,  11, 145,  84,  83,  43,
       128,  32,  61,  57, 115,  23, 147, 121,  15,  48,  60, 109,   9,
        62,  31, 127, 117,  16,  44, 112, 138,  99, 142,  67,  34,  30,
        58,  35,  77,  49,   6, 102, 137,  41, 110,  22,  19,  64,  47,
       113, 140,  14, 141,  36,  97, 136])
```

```
n_training_samples = 12
learnset_data = iris.data[indices[:-n_training_samples]]
learnset_labels = iris.target[indices[:-n_training_samples]]
testset_data = iris.data[indices[-n_training_samples:]]
testset_labels = iris.target[indices[-n_training_samples:]]
print(learnset_data[:4], learnset_labels[:4])
```

```
print(testset_data[:4], testset_labels[:4])
```

```
[[5.9 3.2 4.8 1.8]
 [5.7 2.6 3.5 1. ]
 [6.1 2.6 5.6 1.4]
 [4.6 3.1 1.5 0.2]] [1 1 2 0]
[[6.5 3.2 5.1 2. ]
 [4.6 3.6 1.  0.2]
 [5.1 3.8 1.5 0.3]
 [5.6 2.9 3.6 1.3]] [2 0 0 1]
```

Aufteilungen mit Sklearn

Obwohl es nicht schwierig war müssen wir die Aufteilung nicht, wie oben gezeigt, manuell durchführen. Da dies beim maschinellen Lernen häufig erforderlich ist, verfügt scikit-learn über eine vordefinierte Funktion zum Aufteilen von Daten in Trainings- und Testsätze.

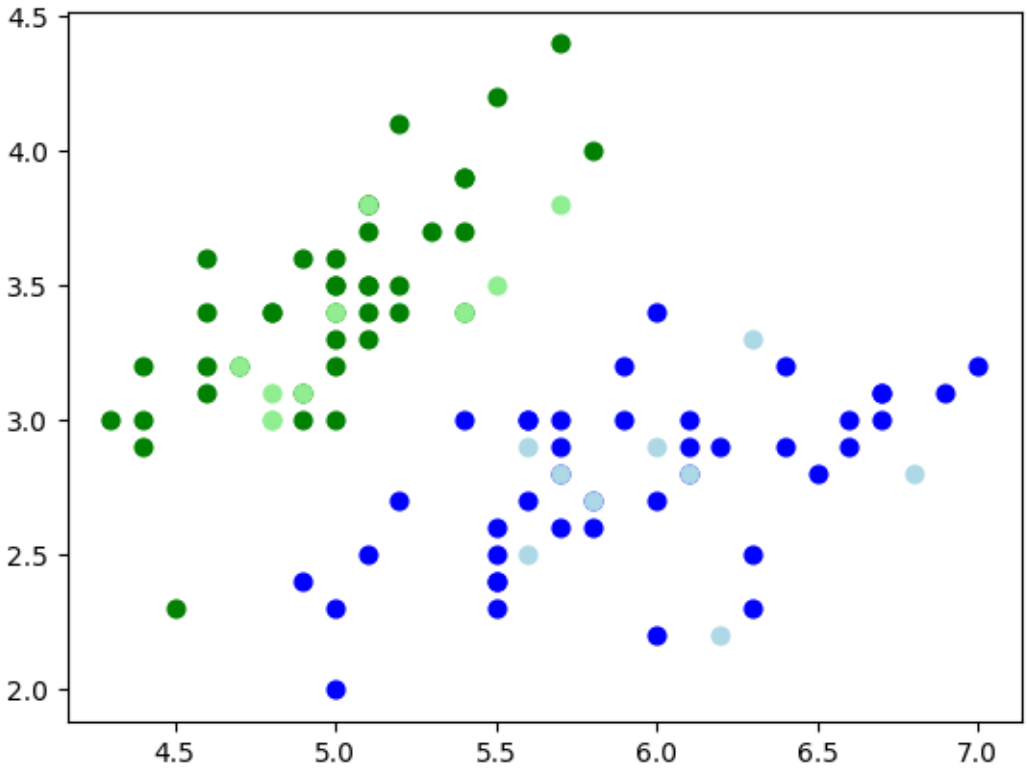
Wir werden dies im Folgenden demonstrieren. Wir werden 80% der Daten als Trainings- und 20% als Testdaten verwenden. Ebensogut hätten wir 70% und 30% nehmen können, denn es gibt keine festen Regeln. Das Wichtigste ist, dass Sie Ihr System fair anhand von Daten bewerten, die es während des Trainings *nicht* gesehen hat! Außerdem müssen in beiden Datensätze genügend Daten enthalten sein.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
iris = load_iris()
data, labels = iris.data, iris.target

res = train_test_split(data, labels,
                       train_size=0.8,
                       test_size=0.2,
                       random_state=42  # reproduzierbare Aufteilung
                      )
train_data, test_data, train_labels, test_labels = res

n = 7
print(f"Die ersten {n}-Datensätze:")
print(test_data[:7])
print(f"Die zugehörigen ersten {n}-Labels:")
print(test_labels[:7])
```

```
Die ersten 7-Datensätze:
[[6.1 2.8 4.7 1.2]
 [5.7 3.8 1.7 0.3]
 [7.7 2.6 6.9 2.3]
 [6.  2.9 4.5 1.5]
 [6.8 2.8 4.8 1.4]
 [5.4 3.4 1.5 0.4]
 [5.6 2.9 3.6 1.3]]
Die zugehörigen ersten 7-Labels:
[1 0 2 1 1 0 1]
```

Fahren wir fort, unsere Trainings- und Testdatensätze zu visualisieren. In unserem Diagramm werden grüne und blaue Punkte die Trainingsdaten darstellen, während hellgrüne und hellblaue Punkte die Testdaten symbolisieren.

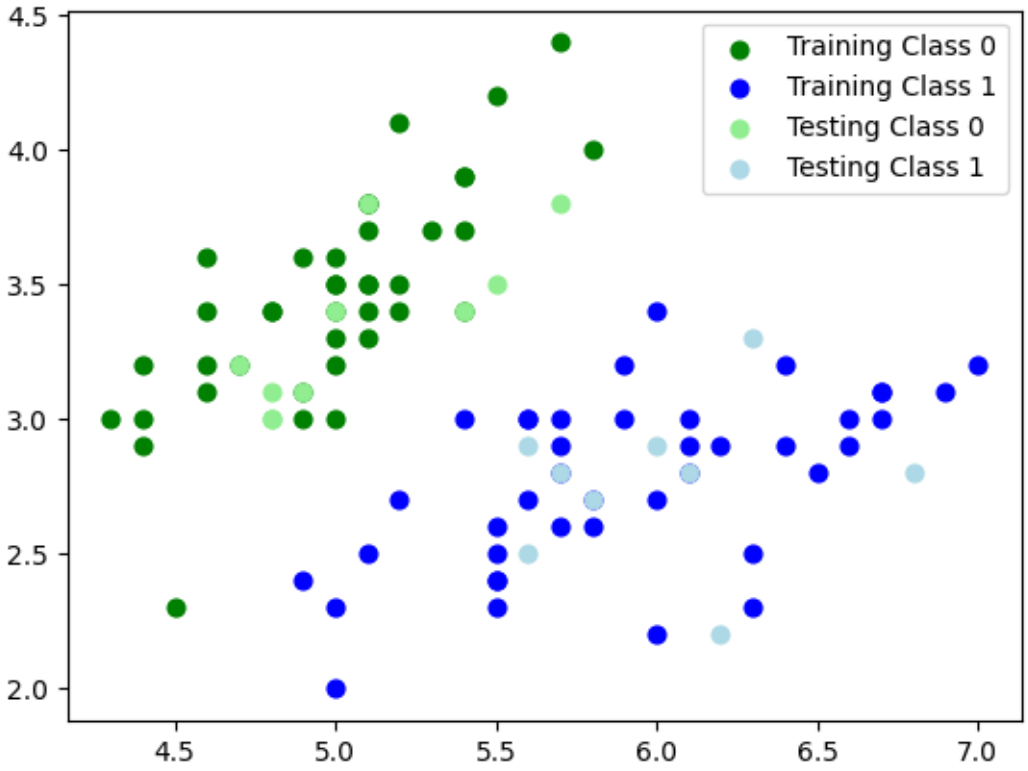
```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
# plotting learn data
colours = ('green', 'blue')
for n_class in range(2):
    ax.scatter(train_data[train_labels==n_class][:, 0],
               train_data[train_labels==n_class][:, 1],
               c=colours[n_class], s=40, label=str(n_class))


# plotting test data
colours = ('lightgreen', 'lightblue')
for n_class in range(2):
    ax.scatter(test_data[test_labels==n_class][:, 0],
               test_data[test_labels==n_class][:, 1],
               c=colours[n_class], s=40, label=str(n_class))

ax.plot();
```



Der folgende Code verbessert die Lesbarkeit und die Modularität, indem er eine separate Funktion für das Plotten der Daten erstellt. Außerdem wird dem Diagramm eine Legende zur besseren Interpretation der Klassen hinzugefügt:

```
import matplotlib.pyplot as plt

# Define function for scatter plot
def plot_data(data, labels, ax, colours, label):
```

```
    for n_class, colour in enumerate(colours):
        ax.scatter(data[labels == n_class][:, 0],
                   data[labels == n_class][:, 1],
                   c=colour, s=40, label=label.format(n_class))

# Create figure and axis objects
fig, ax = plt.subplots()

# Plot training data
train_colours = ['green', 'blue']
plot_data(train_data, train_labels, ax, train_colours,
↪  label="Training Class {}")

# Plot testing data
test_colours = ['lightgreen', 'lightblue']
plot_data(test_data, test_labels, ax, test_colours, label="Testing Class {}")

# Add legend
ax.legend()

# Show plot
plt.show()
```



Geschichtete Zufallsprobe

Insbesondere bei relativ kleinen Datenmengen ist es besser, die Aufteilung zu schichten. Schichtung bedeutet, dass wir den ursprünglichen Klassenanteil des Datensatzes in den Test- und Trainingssätzen beibehalten. Wir berechnen die Klassenanteile der vorherigen Aufteilung in Prozent unter Verwendung des folgenden Codes. Um die Anzahl der Vorkommen jeder

Klasse zu berechnen, verwenden wir die Numpy-Funktion 'bincount'. Es zählt die Anzahl der Vorkommen jedes Werts in dem als Argument übergebenen Array nicht negativer Integers.

```
import numpy as np
print('All:', np.bincount(labels) / float(len(labels)) * 100.0)
print('Training:', np.bincount(train_labels) / float(len(train_labels)) *
↪  100.0)
print('Test:', np.bincount(test_labels) / float(len(test_labels)) * 100.0)
```

```
All: [33.33333333 33.33333333 33.33333333]
Training: [33.33333333 34.16666667 32.5       ]
Test: [33.33333333 30.         36.66666667]
```

Um die Aufteilung zu schichten, können wir das Label-Array als zusätzliches Argument an die Funktion `train_test_split` übergeben:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
iris = load_iris()
data, labels = iris.data, iris.target

res = train_test_split(data, labels,
                       train_size=0.8,
                       test_size=0.2,
                       random_state=42,
                       stratify=labels)
train_data, test_data, train_labels, test_labels = res

print('All:', np.bincount(labels) / float(len(labels)) * 100.0)
print('Training:', np.bincount(train_labels) / float(len(train_labels)) *
↪  100.0)
print('Test:', np.bincount(test_labels) / float(len(test_labels)) * 100.0)
```

```
All: [33.33333333 33.33333333 33.33333333]
Training: [33.33333333 33.33333333 33.33333333]
Test: [33.33333333 33.33333333 33.33333333]
```

Dies ist ein bewusst einfaches Beispiel für eine geschichtete Aufteilung, da der Iris-Datensatz drei gleich große Klassen mit jeweils 50 Stichproben besitzt.

Wir werden jetzt mit der Datei `strange_flowers.txt` des Verzeichnisses `data` arbeiten. Dieser Datensatz wird im Kapitel Synthetische Daten für maschinelles Lernen erzeugen erstellt. Die Klassen in diesem Datensatz haben eine unterschiedliche Anzahl von Elementen. Zuerst laden wir die Daten:

```
content = np.loadtxt("data/strange_flowers.txt", delimiter=" ")
data = content[:, :-1]    # Letzte Spalte wird abgeschnitten
labels = content[:, -1]
labels.dtype
labels.shape
```

```
(795,)
```

```
res = train_test_split(data, labels,
                       train_size=0.8,
```

```
                        test_size=0.2,
                        random_state=42,
                        stratify=labels)
train_data, test_data, train_labels, test_labels = res

# np.bincount expects non negative integers:
print('All:', np.bincount(labels.astype(int))  / float(len(labels)) * 100.0)
print('Training:', np.bincount(train_labels.astype(int)) /
↪  float(len(train_labels)) * 100.0)
print('Test:', np.bincount(test_labels.astype(int)) / float(len(test_labels)) *
↪  100.0)
```

```
All: [ 0.         23.89937107 25.78616352 28.93081761 21.3836478 ]
Training: [ 0.         23.89937107 25.78616352 28.93081761 21.3836478 ]
Test: [ 0.         23.89937107 25.78616352 28.93081761 21.3836478 ]
```

Aufgaben

Aufgabe 1 In dieser Übung geht es um die Aufteilung von Trainings- und Testdaten auf den Wein-Datensatz von scikit-learn. Der Wein-Datensatz enthält die Ergebnisse einer chemischen Analyse von drei verschiedenen Weinsorten. Jede Probe wird in eine von drei Klassen eingeteilt. Die Aufgabe besteht darin, diesen Datensatz in einen Trainings- und einen Testdatensatz mit einem Verhältnis von 70-30 aufzuteilen, wobei 70% der Daten für das Training und 30% für den Test verwendet werden.

Aufgabe 2 Erzeuge nun eine stratifizierte Ausgabe.

Lösungen

Lösung zu Aufgabe 1

```
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split

# Load the Wine dataset
wine = load_wine()

X = wine.data
y = wine.target

# Perform train-test split with 70% training and 30% testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
↪  random_state=42)

# Print the shapes of the resulting train and test sets
print("X_train shape:", X_train.shape)
print("X_test shape:", X_test.shape)
print("y_train shape:", y_train.shape)
print("y_test shape:", y_test.shape)
```

```
X_train shape: (124, 13)
X_test shape: (54, 13)
```

```
y_train shape: (124,)
y_test shape: (54,)
```

Lösung zur Aufgabe 2

```
res = train_test_split(X, y,
                       test_size=0.3,
                       random_state=42,
                       stratify=y    # i.e. the labels
                      )
train_data, test_data, train_labels, test_labels = res

# np.bincount expects non negative integers:
print('All:', np.bincount(y.astype(int))  / float(len(y)) * 100.0)
print('Training:', np.bincount(train_labels.astype(int)) /
↪  float(len(train_labels)) * 100.0)
print('Test:', np.bincount(test_labels.astype(int)) / float(len(test_labels)) *
↪  100.0)
```

```
All: [33.14606742 39.88764045 26.96629213]
Training: [33.06451613 40.32258065 26.61290323]
Test: [33.33333333 38.88888889 27.77777778]
```

```
print("Training set class distribution:")
print({label: count for label, count in zip(wine.target_names,
                                            [sum(train_labels == i) for i in
                                            ↪  range(len(wine.target_names))]
                                            ↪  )})

print("\nTesting set class distribution:")
print({label: count for label, count in zip(wine.target_names,
                                            [sum(test_labels == i) for i in ra
                                            ↪  nge(len(wine.target_names))])})
```

```
Training set class distribution:
{'class_0': 41, 'class_1': 50, 'class_2': 33}

Testing set class distribution:
{'class_0': 18, 'class_1': 21, 'class_2': 15}
```

8 Naechste Nachbarn Klassifikation

8.0.1 Nächste-Nachbarn-Klassifikation

Einführung

"Zeig mir deine Freunde und ich sag' dir wer du bist"

Das Konzept des Nächsten-Nachbarn-Klassifikators oder genauer des k-nächsten-Nachbarn-Klassifikators kann kaum einfacher beschrieben werden, wie mit diesem Sprichwort. Das ist eine alte Redensart, die es in vielen Sprachen und Kulturen gibt. Es wird auch in anderen Worten in der Bibel erwähnt: "Wer mit den Weisen umgeht, der wird weise; wer aber der Toren Geselle ist, der wird Unglück haben." (Sprüche 13,20)

Das k-Nearest-Neighbor Konzept ist also Teil unseres täglichen Lebens und unseres Urteilsvermögens: Stellen Sie sich vor Sie treffen eine Gruppe von Menschen. Sie sind alle jung, stylisch und sportlich. Sie reden über ihren Freund Ben, der gerade nicht dabei ist. Wie ist ihre Vorstellung zu Ben? Richtig, Sie werden wohl denken, dass Ben auch jung, stylisch und sportlich ist.

Wenn du erfährst dass Ben in einer Nachbarschaft lebt, in der konservativ gewählt wird und das Durchschnittseinkommen über 200.000 Dollar im Jahr liegt. Seine beiden direkten Nachbarn verdienen sogar mehr als 300.000 Dollar im Jahr. Was denkst du über Ben? Du wirst ihn sehr wahrscheinlich nicht für einen "Underdog" halten. Vielmehr wirst du glauben, dass er ebenfalls wohlhabend ist und höchst-wahrscheinlich nicht weniger als seine Nachbarn verdienen.

Betrachten wir das Konzept mehr aus der Richtung der Mathematik:

Wie im Kaptiel Aufteilungen der Daten dargelegt, benötigen wir gelabelte Lern- und Testdaten. Im Unterschied zu anderen Klassifikatoren findet bei den reinen Nächsten-Nachbarn-Klassifikatoren jedoch kein Lernen statt, sondern das sogenannte Lernset LS ist Grundbestandteil des Klassifikators. Der k-Nächste_Nachbarn-Klassifikator (k-NN) arbeitet direkt mit den gelernten Stichproben, anstatt Regeln zu erstellen, wie dies andere Klassifizierungsmethoden tun.
Die Aufgabe der Klassifikation besteht in der Zuordnung von Kategorien oder Klassen zu einem Element einer zu klassifizierenden Mengen von Objekten.

Nächste-Nachbarn-Algorithmus:

Es ist eine Menge $K = \{c_1, c_2, \cdots, c_m\}$ von Kategorien gegeben, die auch Klassen genannt werden. z.B. {"male", "female"}.

Außerdem sei eine weitere Menge LS gegeben, die gelabellte Objekte enthält.

$$LS = \{(o_1, c_{o_1}), (o_2, c_{o_2}), \cdots (o_n, c_{o_n})\}$$

Da es keinen Sinn ergibt, dass man weniger gelabellte Objekte als Klassen hat, muss gelten:

$n > m$ und im allgemeinen sogar $n \ggg m$ (n sehr viel größer als m.)

Die Aufgabe der Klassifikation besteht nun darin einem Objekt o ein Kategorie/Klasse c zuzuordnen. Dazu müssen wir zwei Fälle unterscheiden:

- 1. Fall:
  Die Instanz o ist in LS enthalten, d.h. es gibt ein Tupel $(o, c) \in LS$
  In diesem Fall nehmen wir die Klasse c als Klassifikationsergebnis.
- 2. Fall:
  Nun nehmen wir an, dass o nicht in LS enthalten ist, genauer gesagt:
  $\forall c \in C, (o, c) \notin LS$
  o wird nun mit allen Instanzen von LS verglichen. Eine Abstandsmetrik d wird für die Vergleiche benutzt.
  Wir bestimmen die k nächsten Nachbarn von o, d.h. die Instanzen mit den kleinsten Entfernungen.
  k ist eine benutzerdefinierte positive ganze Zahl, die überlicherweise nicht sehr groß ist.
  Um die k-nächsten Nachbarn zu bestimmen, ordnen wir LS auf die foldende Art um:
  $(o_{i_1}, c_{o_{i_1}}), (o_{i_2}, c_{o_{i_2}}), \cdots (o_{i_n}, c_{o_{i_n}})$
  wobei für alle $1 \leq j \leq n-1$ gilt $d(o_{i_j}, o) \leq d(o_{i_{j+1}}, o)$ Die Menge der nächsten Nachbarn N_k besteht aus den ersten k Elementen dieser Ordnung, d.h.
  $N_k = \{(o_{i_1}, c_{o_{i_1}}), (o_{i_2}, c_{o_{i_2}}), \cdots (o_{i_k}, c_{o_{i_k}})\}$
  Die am häufigsten vorkommende Klasse in dieser Menge N_k wird zum Ergebnis der Klassifikation o. Wenn keine Klasse eindeutig am häufigsten vorkommt, muss eine definierte Tie-Break-Regel verwendet werden; die konkrete Regel hängt von der Implementierung ab.

Es gibt keinen allgemein optimalen Wert für `k`. Kleine Werte erzeugen flexible Entscheidungsgrenzen und reagieren stärker auf einzelne Datenpunkte; größere Werte glätten die Grenze. `k` ist daher ein Hyperparameter und sollte anhand von Validierungsdaten oder Kreuzvalidierung gewählt werden. Da k-NN auf Abständen beruht, ist außerdem die Skalierung der Merkmale wichtig, wenn sie unterschiedliche Einheiten oder Größenordnungen besitzen.

Der Algorithmus für den k-Nearest-Neighbor Klassifikator ist einer der einfachsten von allen Machine Learning Algorithmen. k-NN gehört zu den Typen des instanzbasierten Lernens, oder lazy-Lernens. In der Theorie des Maschinellen Lernens wird **Lazy-lernen** als Lernmethode verstanden, in der die Verallgemeinerung der Trainingsdaten verzögert wird, bis eine Abfrage an das System vorgenommen wird. Andererseits haben wir das **Eager-Lernen**, wo das System die Trainingsdaten in der Regel verallgemeint, bevor es Abfragen erhält. In anderen Worten: Die Funktion wird nur lokal approximiert und alle Berechnungen werden bei der jeweils aktuellen Klassifikation durchgeführt.

Das folgende Bild zeigt auf einfache Weise, wie der nächste Nachbar-Klassifikator funktioniert. Das Puzzleteil ist unbekannt. Um herauszufinden, welches Tier es sein könnte, müssen wir die Nachbarn finden. Wenn `k = 1`, ist der einzige Nachbarn eine Katze und wir nehmen in diesem Fall an, dass das Puzzleteil auch eine Katze sein sollte. Wenn `k = 4` gilt, enthalten die nächsten Nachbarn ein Huhn und drei Katzen. In diesem Fall können wir annehmen, dass unser Objekt, also das Puzzleteil, eine Katze sein sollte.

k-Nächster-Nachbar von Grund auf

Dataset vorbereiten Bevor wir einen Nearest-Neighbor Klassifikator schreiben, müssen wir über die Daten nachdenken, das heißt den **Lern-** und den **Testdatensatz**. Wir benutzen die "Iris"-Daten, die durch das **sklearn**-Modul zur Verfügung gestellt werden.

Der Datensatz beinhaltet 50 Beispiele von drei Arten von Schwertlilien (Englisch: Iris).

- Iris setosa (Borsten-Schwertlilie)
- Iris virginica und
- Iris versicolor (Verschiedenfarbige-Schwertlilie)

Vier Features (Merkmale) werden bei jeder Probe gemessen: - Die Länge der Kelchblätter in Zentimeter - Die Breite der Kelchblätter in Zentimeter - Die Länge der Blütenblätter in Zentimeter - Die Breite der Blütenblätter in Zentimeter

```
import numpy as np
from sklearn import datasets

iris = datasets.load_iris()
data = iris.data
labels = iris.target

for i in [0, 79, 99, 101]:
    print(f"Index: {i:3}, Merkmale: {data[i]}, Label: {labels[i]}")
```

```
Index:   0, Merkmale: [5.1 3.5 1.4 0.2], Label: 0
Index:  79, Merkmale: [5.7 2.6 3.5 1. ], Label: 1
Index:  99, Merkmale: [5.7 2.8 4.1 1.3], Label: 1
Index: 101, Merkmale: [5.8 2.7 5.1 1.9], Label: 2
```

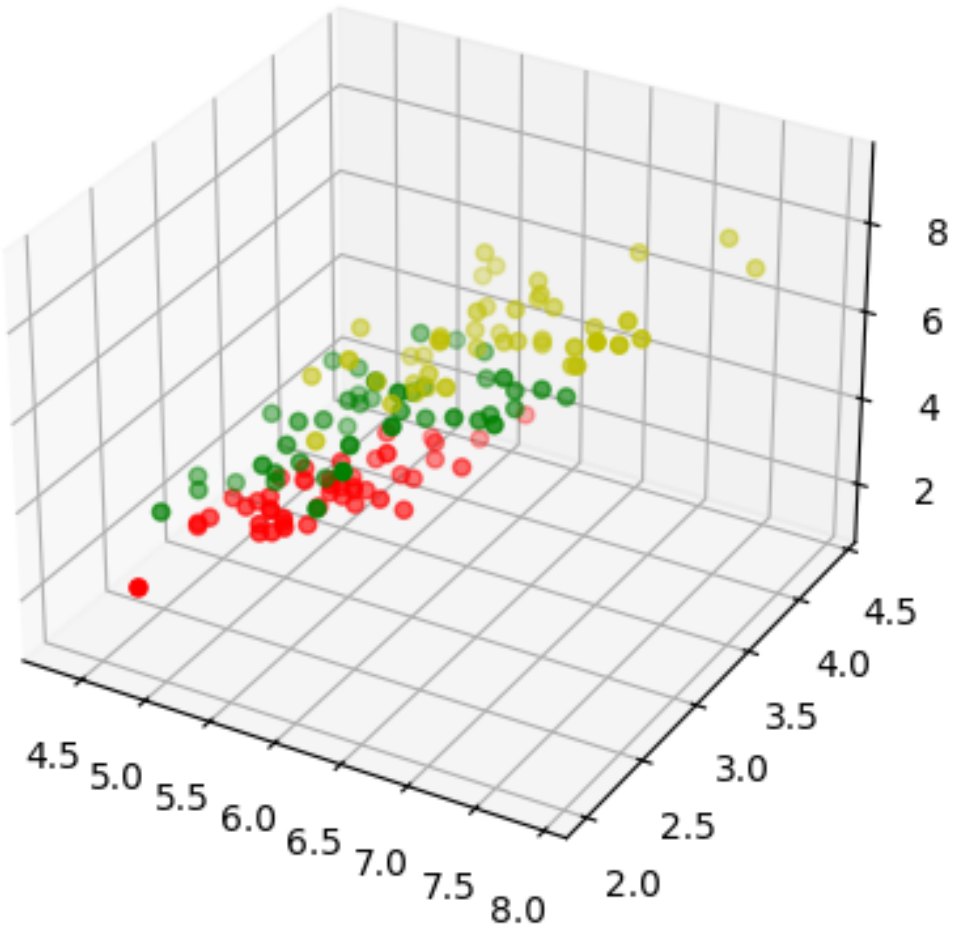
Erstellen wir ein **Learnset** aus den oben genannten Daten. Wir verwenden die Funktion permutation aus np.random um die Daten zufällig aufzuteilen.

```
# Seeding ist nur für die Website erforderlich,
# damit die Werte immer gleich sind:
np.random.seed(42)
indices = np.random.permutation(len(data))

n_training_samples = 12
learn_data = data[indices[:-n_training_samples]]
learn_labels = labels[indices[:-n_training_samples]]
test_data = data[indices[-n_training_samples:]]
test_labels = labels[indices[-n_training_samples:]]

print("Die ersten Stichproben unseres Lernsets:")
print(f"{'Index':7s}{'Daten':20s}{'Label':3s}")
for i in range(5):
    print(f"{i:4d}   {learn_data[i]}   {learn_labels[i]:3}")

print("Die ersten Stichproben unseres Testsets:")
print(f"{'Index':7s}{'Daten':20s}{'Label':3s}")
for i in range(5):
    print(f"{i:4d}   {test_data[i]}   {test_labels[i]:3}")
```

Der folgende Code dienst nur der Visualisierung der Daten aus dem **Lernset**. Unsere Daten

bestehen aus vier Werten pro Schwertlilie. Um die Daten etwas zu reduzieren, summieren wir die dritten und vierten Werte auf. So können wir die Werte in einem 3-Dimensionalen Raum darstellen:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

X = []
for iclass in range(3):
    X.append([[], [], []])
    for i in range(len(learn_data)):
        if learn_labels[i] == iclass:
            X[iclass][0].append(learn_data[i][0])
            X[iclass][1].append(learn_data[i][1])
            X[iclass][2].append(sum(learn_data[i][2:]))

colours = ("r", "g", "y")

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

for iclass in range(3):
    ax.scatter(X[iclass][0], X[iclass][1], X[iclass][2], c=colours[iclass])
plt.show()
```



Entfernungsmetriken Wir wir bereits ausführlich erwähnt haben, berechnen wir die Abstände zwischen den Punkten der Stichprobe und dem zu klassifizierenden Objekt. Zur Berechnung dieser Abstände benötigen wir eine Abstandsfunktion.

In n-dimensionalen Vektorräumen, benutzt man meistens einer der folgenden drei Abstandsmetriken:

- **Euklidischer-Abstand**

  Der euklidische Abstand zwischen zwei Punkten `x` und `y` entweder in der Ebene oder im 3-dimensionalen Raum entspricht der Länge der Strecke, die diese beiden Punkte verbindet. Es kann aus den kartesischen Koordinaten der Punkte mit dem Satz des Pythagoras berechnet werden, daher wird es gelegentlich auch die Pythagoros-Entfernung bezeichnet. Die allgemeine Formel lautet:

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

- **Manhatten-Abstand**

  Es ist definiert als die Summe der absoluten Werte zwischen den Koordinaten von x

und y :

$$d(x,y) = \sum_{i=1}^{n} |x_i - y_i|$$

- **Minkowski-Abstand**

  Der Minkowski-Abstand verallgemeinert den Euklidischen und Manhatten-Abstand in einer Distanzmetrik. Wenn wir den Parameter `p` in der folgenden Formel auf 1 einstellen, erhalten wir den Manhatten-Abstand, und mit dem Wert 2 erhalten wir den Euklidischen-Abstand:

$$d(x,y) = \left(\sum_{i=1}^{n} (x_i - y_i)^p \right)^{\frac{1}{p}}$$

Das folgende Diagramm veranschaulicht den Zusammenhang zwischen Manhatten- und Euklidschem Abstand:

Die blaue Linie veranschaulicht den Euklidschen Abstand zwischen dem grünen und dem roten Punkt. Ansonsten kann man sich auch über die orange, grüne oder gelbe Linie vom grünen Punkt zum roten Punkt bewegen. Die Linien entsprechen der Manhatten-Distanz. Die Länge ist jeweils gleich.

Die Nachbarn ermitteln Um Ähnlichkeiten zwischen zwei Objekten zu ermitteln brauchen wir eine Distanz-Funktion. In unserem Beispiel benutzen wir den Euklidischen Abstand.

Der euklidische Abstand zweier Punkte in der Ebene oder im Raum ist die zum Beispiel mit einem Lineal gemessene Länge einer Strecke, die diese zwei Punkte verbindet. In kartesischen Koordinaten kann der euklidische Abstand mit Hilfe des Satzes von Pythagoras berechnet werden. Mit Hilfe der so gewonnenen Formel kann der Begriff des euklidischen Abstands auf n-dimensionale Koordinatenräume verallgemeinert werden. Berechnen können wir den Abstand mit der Funktion `norm` aus dem Modul `np.linalg`:

```
def distance(instance1, instance2):
    """ Berechnet den Eukliischen Abstand zwischen
    zwei Instanzen"""
    return np.linalg.norm(np.subtract(instance1, instance2))

print(distance([3, 5], [1, 1]))
print(distance(learn_data[3], learn_data[44]))
```

```
4.47213595499958
3.4190641994557516
```

Die Funktion get_neighbors liefert eine Liste mit k Nachbarn zurück, die der Instanz test_instance am nächsten sind:

```
def get_neighbors(training_set,
                  labels,
                  test_instance,
                  k,
                  distance):
    """
    get_neighbors berechnet eine Liste mit k-Nachbarn,
```

```
    die der Instanz 'test_instance' am nächsten liegen.

    'get_neighbors' liefert eine Liste mit k 3er-Tupel zurück.
    Jedes Tupel besteht aus (index, dist, label), wobei gilt

    index    ist der Index aus training_set,
    dist     ist die Distanz zwischen der test_instance und der Instanz training_set[ind
    distance ist eine Referenz auf die Funktion, welche die Distanz berechnet
    """
    distances = []
    for index in range(len(training_set)):
        dist = distance(test_instance, training_set[index])
        distances.append((training_set[index], dist, labels[index]))
    distances.sort(key=lambda x: x[1])
    neighbors = distances[:k]
    return neighbors
```

Wir testen die Funktion mit unseren Iris-Proben:

```
for i in range(5):
    neighbors = get_neighbors(learn_data,
                              learn_labels,
                              test_data[i],
                              3,
                              distance=distance)

    print("Index:          ",i,'\n',
          "Testset-Data:  ",test_data[i],'\n',
          "Testset-Label: ",test_labels[i],'\n',
          "Nachbarn:      ",neighbors,'\n')
```

```
Index:          0
 Testset-Data:   [5.7 2.8 4.1 1.3]
 Testset-Label:  1
 Nachbarn:       [(array([5.7, 2.9, 4.2, 1.3]), 0.14142135623730995, 1),
 ↪  (array([5.6, 2.7, 4.2, 1.3]), 0.17320508075688815, 1), (array([5.6, 3. ,
 ↪  4.1, 1.3]), 0.22360679774997935, 1)]

Index:          1
 Testset-Data:   [6.5 3.  5.5 1.8]
 Testset-Label:  2
 Nachbarn:       [(array([6.4, 3.1, 5.5, 1.8]), 0.1414213562373093, 2),
 ↪  (array([6.3, 2.9, 5.6, 1.8]), 0.24494897427831785, 2), (array([6.5, 3. ,
 ↪  5.2, 2. ]), 0.3605551275463988, 2)]

Index:          2
 Testset-Data:   [6.3 2.3 4.4 1.3]
 Testset-Label:  1
 Nachbarn:       [(array([6.2, 2.2, 4.5, 1.5]), 0.2645751311064586, 1),
 ↪  (array([6.3, 2.5, 4.9, 1.5]), 0.574456264653803, 1), (array([6. , 2.2, 4. ,
 ↪  1. ]), 0.5916079783099617, 1)]

Index:          3
 Testset-Data:   [6.4 2.9 4.3 1.3]
 Testset-Label:  1
```

```
 Nachbarn:       [(array([6.2, 2.9, 4.3, 1.3]), 0.20000000000000018, 1),
 ↪  (array([6.6, 3. , 4.4, 1.4]), 0.2645751311064587, 1), (array([6.6, 2.9, 4.6,
 ↪  1.3]), 0.3605551275463984, 1)]

Index:          4
 Testset-Data:   [5.6 2.8 4.9 2. ]
 Testset-Label:  2
 Nachbarn:       [(array([5.8, 2.7, 5.1, 1.9]), 0.31622776601683755, 2),
 ↪  (array([5.8, 2.7, 5.1, 1.9]), 0.31622776601683755, 2), (array([5.7, 2.5, 5. ,
 ↪  2. ]), 0.33166247903553986, 2)]
```

Voting für ein einzelnes Ergebnis Wir schreiben eine Voting-Funktion (Abstimmung). Diese Funktion benutzt die Klasse `Counter` aus `collections` um die Anzahl der Klassen in einer Instanzen-Liste zu ermitteln. Die Instanzen-Liste stellt natürlich die Nachbarn dar. Die Funktion `vote` liefert die passendste Klasse zurück.

```
from collections import Counter

def vote(neighbors):
    class_counter = Counter()
    for neighbor in neighbors:
        class_counter[neighbor[2]] += 1
    return class_counter.most_common(1)[0][0]
```

Testen wir die vote Funktion mit unseren Trainings-Proben:

```
for i in range(n_training_samples):
    neighbors = get_neighbors(learn_data,
                              learn_labels,
                              test_data[i],
                              3,
                              distance=distance)
    print("Index: ", i,
          ", Ergebnis(vote): ", vote(neighbors),
          ", Bezeichnung: ", test_labels[i],
          ", Daten: ", test_data[i])
```

```
Index:  0 , Ergebnis(vote):  1 , Bezeichnung:  1 , Daten:  [5.7 2.8 4.1 1.3]
Index:  1 , Ergebnis(vote):  2 , Bezeichnung:  2 , Daten:  [6.5 3.  5.5 1.8]
Index:  2 , Ergebnis(vote):  1 , Bezeichnung:  1 , Daten:  [6.3 2.3 4.4 1.3]
Index:  3 , Ergebnis(vote):  1 , Bezeichnung:  1 , Daten:  [6.4 2.9 4.3 1.3]
Index:  4 , Ergebnis(vote):  2 , Bezeichnung:  2 , Daten:  [5.6 2.8 4.9 2. ]
Index:  5 , Ergebnis(vote):  2 , Bezeichnung:  2 , Daten:  [5.9 3.  5.1 1.8]
Index:  6 , Ergebnis(vote):  0 , Bezeichnung:  0 , Daten:  [5.4 3.4 1.7 0.2]
Index:  7 , Ergebnis(vote):  1 , Bezeichnung:  1 , Daten:  [6.1 2.8 4.  1.3]
Index:  8 , Ergebnis(vote):  1 , Bezeichnung:  2 , Daten:  [4.9 2.5 4.5 1.7]
Index:  9 , Ergebnis(vote):  0 , Bezeichnung:  0 , Daten:  [5.8 4.  1.2 0.2]
Index:  10 , Ergebnis(vote):  1 , Bezeichnung:  1 , Daten:  [5.8 2.6 4.  1.2]
Index:  11 , Ergebnis(vote):  2 , Bezeichnung:  2 , Daten:  [7.1 3.  5.9 2.1]
```

Wir sehen, dass die Vorhersagen mit den Bezeichnungen übereinstimmen, ausgenommen der Fall mit dem Index 8.

`vote_prob` ist eine Funktion wie `vote`, liefert aber den Klassen-Namen und dessen Wahrscheinlichkeit:

```
def vote_prob(neighbors):
    class_counter = Counter()
    for neighbor in neighbors:
        class_counter[neighbor[2]] += 1
    labels, votes = zip(*class_counter.most_common())
    winner = class_counter.most_common(1)[0][0]
    votes4winner = class_counter.most_common(1)[0][1]
    return winner, votes4winner/sum(votes)
```

```
for i in range(n_training_samples):
    neighbors = get_neighbors(learn_data,
                              learn_labels,
                              test_data[i],
                              5,
                              distance=distance)
    print("Index: ", i,
          ", Ergebnis(vote_prob): ", vote_prob(neighbors),
          ", Bezeichnung: ", test_labels[i],
          ", Daten: ", test_data[i])
```

```
Index:  0 , Ergebnis(vote_prob):  (1, 1.0) , Bezeichnung:  1 , Daten:  [5.7 2.8
↪  4.1 1.3]
Index:  1 , Ergebnis(vote_prob):  (2, 1.0) , Bezeichnung:  2 , Daten:  [6.5 3.
↪  5.5 1.8]
Index:  2 , Ergebnis(vote_prob):  (1, 1.0) , Bezeichnung:  1 , Daten:  [6.3 2.3
↪  4.4 1.3]
Index:  3 , Ergebnis(vote_prob):  (1, 1.0) , Bezeichnung:  1 , Daten:  [6.4 2.9
↪  4.3 1.3]
Index:  4 , Ergebnis(vote_prob):  (2, 1.0) , Bezeichnung:  2 , Daten:  [5.6 2.8
↪  4.9 2. ]
Index:  5 , Ergebnis(vote_prob):  (2, 0.8) , Bezeichnung:  2 , Daten:  [5.9 3.
↪  5.1 1.8]
Index:  6 , Ergebnis(vote_prob):  (0, 1.0) , Bezeichnung:  0 , Daten:  [5.4 3.4
↪  1.7 0.2]
Index:  7 , Ergebnis(vote_prob):  (1, 1.0) , Bezeichnung:  1 , Daten:  [6.1 2.8
↪  4.  1.3]
Index:  8 , Ergebnis(vote_prob):  (1, 1.0) , Bezeichnung:  2 , Daten:  [4.9 2.5
↪  4.5 1.7]
Index:  9 , Ergebnis(vote_prob):  (0, 1.0) , Bezeichnung:  0 , Daten:  [5.8 4.
↪  1.2 0.2]
Index:  10 , Ergebnis(vote_prob):  (1, 1.0) , Bezeichnung:  1 , Daten:  [5.8 2.6
↪  4.  1.2]
Index:  11 , Ergebnis(vote_prob):  (2, 1.0) , Bezeichnung:  2 , Daten:  [7.1 3.
↪  5.9 2.1]
```

Der gewichtete Nearest-Neighbor-Klassifikator Wir haben bis jetzt nur k Elemente aus der Nachbarschaft eines unbekannten Objektes "UO" betrachtet, und dessen Mehrheit. Die Mehrheit der Stimmabgaben hat sich im vorigen Beispiel stark gezeigt, wurde aber bei der folgenden Argumentation nicht berücksichtigt: "Je weiter ein Nachbar, desto mehr weicht er vom 'echten' Ergebnis ab." Um es mit anderen Worten zu sagen: "Wir können dem nächsten Nachbarn mehr vertrauen als dem, der am weitesten entfernt ist." Nehmen wir an, dass das unbekannte Objekt "UO" 11 Nachbarn hat. Die naheliegendsten fünf Nachbarn gehören zur Klasse A und alle anderen sechs, die weiter entfernt sind, gehören zur Klasse B. Zu

welcher Klasse wir "UO" wohl gehören? Das vorherige Vorgehen würde "Klasse B" sagen, denn wir haben ein Verhältnis von 6 zu 5 für Klasse B. Auf der anderen Seite gehören die naheliegendsten zur Klasse A und das sollte stärker berücksichtigt werden. Um die Strategie weiterzuverfolgen, können wir den Nachbarn Gewichtungen zuweisen: Der naheliegendste Nachbar einer Instanz erhält ein Gewicht von 1/1, der zweitnächste bekommt ein Gewicht von 1/2. Mit den weiteren Nachbarn wird ebenso weiterverfahren bis zu einem Gewicht von $1/k$ für den am weitesten entfernten Nachbarn.

Wir verwenden also die harmonische Serie als Gewichtung:

$$\sum_{i}^{k} 1/(i+1) = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{k}$$

In der folgenden Funktion haben wir dies implementiert:

```
def vote_harmonic_weights(neighbors, all_results=True):
    class_counter = Counter()
    number_of_neighbors = len(neighbors)
    for index in range(number_of_neighbors):
        class_counter[neighbors[index][2]] += 1/(index+1)
    labels, votes = zip(*class_counter.most_common())
    #print(labels, votes)
    winner = class_counter.most_common(1)[0][0]
    votes4winner = class_counter.most_common(1)[0][1]
    if all_results:
        total = sum(class_counter.values(), 0.0)
        for key in class_counter:
             class_counter[key] /= total
        return winner, class_counter.most_common()
    else:
        return winner, votes4winner / sum(votes)
```

```
for i in range(n_training_samples):
    neighbors = get_neighbors(learn_data,
                              learn_labels,
                              test_data[i],
                              6,
                              distance=distance)
    print("Index: ", i,
          ", Ergebnis(vote_harmonic_weights): ",
          vote_harmonic_weights(neighbors,
                                all_results=True))
```

```
Index:  0 , Ergebnis(vote_harmonic_weights):  (1, [(1, 1.0)])
Index:  1 , Ergebnis(vote_harmonic_weights):  (2, [(2, 1.0)])
Index:  2 , Ergebnis(vote_harmonic_weights):  (1, [(1, 1.0)])
Index:  3 , Ergebnis(vote_harmonic_weights):  (1, [(1, 1.0)])
Index:  4 , Ergebnis(vote_harmonic_weights):  (2, [(2, 0.9319727891156463), (1,
↪  0.06802721088435375)])
Index:  5 , Ergebnis(vote_harmonic_weights):  (2, [(2, 0.8503401360544217), (1,
↪  0.14965986394557826)])
Index:  6 , Ergebnis(vote_harmonic_weights):  (0, [(0, 1.0)])
Index:  7 , Ergebnis(vote_harmonic_weights):  (1, [(1, 1.0)])
Index:  8 , Ergebnis(vote_harmonic_weights):  (1, [(1, 1.0)])
```

```
Index:  9 , Ergebnis(vote_harmonic_weights):  (0, [(0, 1.0)])
Index:  10 , Ergebnis(vote_harmonic_weights):  (1, [(1, 1.0)])
Index:  11 , Ergebnis(vote_harmonic_weights):  (2, [(2, 1.0)])
```

Das vorherige Vorgehen berücksichtigt lediglich die Reihenfolge der Nachbarn entsprechend der Entfernung. Wir können die Auswahl aber noch verbessern indem die aktuelle tatsächliche Entfernung berücksichtigt wird. Dafür schreiben wir eine neue Funktion:

```
def vote_distance_weights(neighbors, all_results=True):
    class_counter = Counter()
    number_of_neighbors = len(neighbors)
    for index in range(number_of_neighbors):
        dist = neighbors[index][1]
        label = neighbors[index][2]
        class_counter[label] += 1 / (dist**2 + 1)
    labels, votes = zip(*class_counter.most_common())
    #print(labels, votes)
    winner = class_counter.most_common(1)[0][0]
    votes4winner = class_counter.most_common(1)[0][1]
    if all_results:
        total = sum(class_counter.values(), 0.0)
        for key in class_counter:
            class_counter[key] /= total
        return winner, class_counter.most_common()
    else:
        return winner, votes4winner / sum(votes)
```

```
for i in range(n_training_samples):
    neighbors = get_neighbors(learn_data,
                              learn_labels,
                              test_data[i],
                              6,
                              distance=distance)
    print("Index: ", i,
          ", Ergebnis(vote_distance_weights): ",
          ↪  vote_distance_weights(neighbors,
                                   all_results=True))
```

```
Index:  0 , Ergebnis(vote_distance_weights):  (1, [(1, 1.0)])
Index:  1 , Ergebnis(vote_distance_weights):  (2, [(2, 1.0)])
Index:  2 , Ergebnis(vote_distance_weights):  (1, [(1, 1.0)])
Index:  3 , Ergebnis(vote_distance_weights):  (1, [(1, 1.0)])
Index:  4 , Ergebnis(vote_distance_weights):  (2, [(2, 0.8490154592118361), (1,
↪  0.15098454078816387)])
Index:  5 , Ergebnis(vote_distance_weights):  (2, [(2, 0.6736137462184479), (1,
↪  0.3263862537815521)])
Index:  6 , Ergebnis(vote_distance_weights):  (0, [(0, 1.0)])
Index:  7 , Ergebnis(vote_distance_weights):  (1, [(1, 1.0)])
Index:  8 , Ergebnis(vote_distance_weights):  (1, [(1, 1.0)])
Index:  9 , Ergebnis(vote_distance_weights):  (0, [(0, 1.0)])
Index:  10 , Ergebnis(vote_distance_weights):  (1, [(1, 1.0)])
Index:  11 , Ergebnis(vote_distance_weights):  (2, [(2, 1.0)])
```

Übung und Beispiel: Klassifizierung von Früchten mit K-Nächste-Nachbarn

Datensatz: Erzeugen Sie einen Datensatz mit Informationen über Früchte mit drei Merkmalen: Süße, Säuregehalt und Gewicht (in Gramm). Wir betrachten drei Arten von Früchten: “Apfel”, “Zitrone”, und “Mango”.

Zielsetzung: Erstellen eines K-nearest neighbors Klassifikators zur Vorhersage der Obstsorte auf der Grundlage von Süße, Säuregehalt und Gewicht.

Zunächst werden wir Zufallsdaten mit Früchten und entsprechenden Daten erstellen:

Wir verwenden die [Brix-Skala] (https://en.wikipedia.org/wiki/Brix) für die Süße von Äpfeln.

```
import numpy as np

np.random.seed(42)

def create_features(number_samples, *min_max_features):
    ⌋
    ↪  """ Creates an array with number_samples rows and len(min_max_features) columns """
    features = []
    for min_val, max_val,rounding in min_max_features:
        features.append(np.random.uniform(min_val, max_val,
        ↪  number_samples).round(rounding))
    result = np.column_stack(features)
    return result

num_apples, num_mangos, num_lemons = 40, 42, 45
sweetness = (10, 18, 0)
acidity = 3.4, 4, 2
weight = 140.0, 250.0, 0
apples = create_features(num_apples, sweetness, acidity, weight)
apples
```

```
array([[ 13.  ,   3.47, 235.  ],
       [ 18.  ,   3.7 , 209.  ],
       [ 16.  ,   3.42, 176.  ],
       [ 15.  ,   3.95, 147.  ],
       [ 11.  ,   3.56, 174.  ],
       [ 11.  ,   3.8 , 176.  ],
       [ 10.  ,   3.59, 220.  ],
       [ 17.  ,   3.71, 210.  ],
       [ 15.  ,   3.73, 238.  ],
       [ 16.  ,   3.51, 192.  ],
       [ 10.  ,   3.98, 153.  ],
       [ 18.  ,   3.87, 218.  ],
       [ 17.  ,   3.96, 224.  ],
       [ 12.  ,   3.94, 202.  ],
       [ 11.  ,   3.76, 225.  ],
       [ 11.  ,   3.95, 194.  ],
       [ 12.  ,   3.45, 198.  ],
       [ 14.  ,   3.52, 187.  ],
       [ 13.  ,   3.43, 143.  ],
       [ 12.  ,   3.6 , 152.  ],
       [ 15.  ,   3.63, 143.  ],
       [ 11.  ,   3.56, 210.  ],
```

```
       [ 12.  ,   3.9 , 175.  ],
       [ 13.  ,   3.61, 196.  ],
       [ 14.  ,   3.57, 240.  ],
       [ 16.  ,   3.73, 167.  ],
       [ 12.  ,   3.48, 185.  ],
       [ 14.  ,   3.88, 223.  ],
       [ 15.  ,   3.44, 165.  ],
       [ 10.  ,   3.99, 148.  ],
       [ 15.  ,   3.86, 172.  ],
       [ 11.  ,   3.52, 158.  ],
       [ 11.  ,   3.4 , 242.  ],
       [ 18.  ,   3.89, 229.  ],
       [ 18.  ,   3.82, 210.  ],
       [ 16.  ,   3.84, 236.  ],
       [ 12.  ,   3.86, 228.  ],
       [ 11.  ,   3.44, 161.  ],
       [ 15.  ,   3.62, 238.  ],
       [ 14.  ,   3.47, 199.  ]])
```

```
sweetness = (6, 14, 0)
acidity = 5.8, 6, 1
weight = 100.0, 300.0, 0
mangos = create_features(num_mangos, sweetness, acidity, weight)

sweetness = (6, 12, 0)
acidity = 2.0, 2.6, 1
weight = 130, 170, 0
lemons = create_features(num_lemons, sweetness, acidity, weight)
```

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Combine the data and create labels
X = np.vstack((apples, mangos, lemons))
y = np.array(['Apple'] * num_apples + ['Mango'] * num_mangos + ['lemon'] *
↪  num_lemons)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
↪  random_state=42)
```

```
n_test_samples = len(X_test)

for i in range(n_test_samples):
    neighbors = get_neighbors(X_train,
                              y_train,
                              X_test[i],
                              6,
                              distance=distance)

    print("index: ", i,
          ", result of vote: ",
          vote_harmonic_weights(neighbors,
                                all_results=True))
```

```
index:  0 , result of vote:  ('Apple', [('Apple', 0.7959183673469387), ('Mango',
↪  0.2040816326530612)])
index:  1 , result of vote:  ('lemon', [('lemon', 0.8979591836734694), ('Apple',
↪  0.10204081632653063)])
index:  2 , result of vote:  ('Mango', [('Mango', 1.0)])
index:  3 , result of vote:  ('Mango', [('Mango', 1.0)])
index:  4 , result of vote:  ('Mango', [('Mango', 0.8503401360544217), ('lemon',
↪  0.14965986394557826)])
index:  5 , result of vote:  ('lemon', [('lemon', 0.9319727891156463), ('Apple',
↪  0.06802721088435375)])
index:  6 , result of vote:  ('Mango', [('Mango', 0.8503401360544217), ('lemon',
↪  0.14965986394557826)])
index:  7 , result of vote:  ('lemon', [('lemon', 0.8503401360544217), ('Mango',
↪  0.14965986394557826)])
index:  8 , result of vote:  ('lemon', [('lemon', 0.9319727891156463), ('Mango',
↪  0.06802721088435375)])
index:  9 , result of vote:  ('Mango', [('Mango', 1.0)])
index:  10 , result of vote:  ('lemon', [('lemon', 0.5918367346938775), ('Mango',
↪  0.4081632653061224)])
index:  11 , result of vote:  ('Apple', [('Apple', 0.7619047619047619), ('Mango',
↪  0.23809523809523805)])
index:  12 , result of vote:  ('Apple', [('Apple', 0.5102040816326531), ('lemon',
↪  0.272108843537415), ('Mango', 0.217687074829932)])
index:  13 , result of vote:  ('lemon', [('lemon', 0.9319727891156463), ('Mango',
↪  0.06802721088435375)])
index:  14 , result of vote:  ('Mango', [('Mango', 0.6462585034013606), ('Apple',
↪  0.35374149659863946)])
index:  15 , result of vote:  ('lemon', [('lemon', 0.8163265306122448), ('Apple',
↪  0.1020408163265306), ('Mango', 0.08163265306122448)])
index:  16 , result of vote:  ('lemon', [('lemon', 0.8979591836734694), ('Mango',
↪  0.10204081632653063)])
index:  17 , result of vote:  ('lemon', [('lemon', 1.0)])
index:  18 , result of vote:  ('lemon', [('lemon', 1.0)])
index:  19 , result of vote:  ('Apple', [('Apple', 0.8503401360544217), ('Mango',
↪  0.14965986394557826)])
index:  20 , result of vote:  ('Apple', [('Apple', 0.8639455782312925), ('Mango',
↪  0.13605442176870747)])
index:  21 , result of vote:  ('lemon', [('lemon', 0.9183673469387754), ('Mango',
↪  0.08163265306122245)])
index:  22 , result of vote:  ('lemon', [('lemon', 0.9319727891156463), ('Mango',
↪  0.06802721088435375)])
index:  23 , result of vote:  ('Mango', [('Mango', 0.9319727891156463), ('Apple',
↪  0.06802721088435375)])
index:  24 , result of vote:  ('Apple', [('Apple', 0.8163265306122448), ('Mango',
↪  0.18367346938775508)])
index:  25 , result of vote:  ('Mango', [('Mango', 1.0)])
```

Um unsere Klassifizierung zu bewerten, schreiben wir nun eine Bewertungsfunktion:

```python
def evaluate(X_train, X_test, y_train, y_test, threshold=0):
    """
    Bewertet die Leistung eines K-Nächste-Nachbarn-Klassifikators.

    Parameter:
        X_train (ndarray): Trainingsmerkmale.
        X_test (ndarray): Test-Merkmale.
        y_train (ndarray): Trainingsbezeichnungen.
        y_test (ndarray): Testkennungen.
```

```
        ↪        threshold (Float): Schwellenwert für die Entscheidungssicherheit (Standard:
                                    ↪                                Wenn die Wahrscheinlichkeit der vorhe
                                    ↪                                wird die Probe als 'undecided' (unents

    Rückgabe:
        ↪        dict: Ein Wörterbuch, das die Anzahl der richtigen Vorhersagen, falschen Vo
             und unentschiedenen Vorhersagen beinhaltet

    Anmerkung:
        ↪        Die Funktion geht davon aus, dass es eine andere Funktion namens get_neighbo
        ↪        die k-nächsten Nachbarn für jede Stichprobe abruft, und die vom K-nearest ne
        ↪        vom K-Nächste-Nachbarn-Algorithmus verwendete Abstandsmaß ist in einer Varia
    """
    evaluation = dict(corrects=0, wrongs=0, undecided=0)
    n_test_samples = len(X_test)
    for i in range(n_test_samples):
        neighbors = get_neighbors(X_train,
                                  y_train,
                                  X_test[i],
                                  6,
                                  distance=distance)
        class_label, probability = vote_distance_weights(neighbors,
        ↪  all_results=False)
        if class_label == y_test[i]:
            if probability >= threshold:
                evaluation['corrects'] += 1
            else:
                evaluation['undecided'] += 1
        else:
            if probability >= threshold:
                evaluation['wrongs'] += 1
            else:
                evaluation['undecided'] += 1
    return evaluation


evaluate(X_train, X_test, y_train, y_test, 0.5)
```

```
{'corrects': 21, 'wrongs': 5, 'undecided': 0}
```

```
evaluate(X_train, X_test, y_train, y_test, 0.9)
```

```
{'corrects': 13, 'wrongs': 1, 'undecided': 12}
```

kNN in der Linguistik

Das nächste Beispiel kommt aus der Computer-Linguistik. Wir schauen uns an, wie wir mit einem k-Nearest-Neighbor-Klassifikator falsch geschriebene Worte erkennen können.

Dazu verwenden wir das Modul levenshtein, dass wir in unserem Tutorial zu Levenshtein Distanz bereits implementiert haben.

```
from levenshtein import levenshtein

cities = open("data/city_names.txt").readlines()
cities = [city.strip() for city in cities]

for city in ["Freiburg", "Frieburg", "Freiborg",
             "Hamborg", "Sahrluis"]:
    neighbors = get_neighbors(cities,
                              cities,
                              city,
                              2,
                              distance=levenshtein)

    print("Ergebnis(vote_distance_weights): ",
    ↪  vote_distance_weights(neighbors))
```

```
Ergebnis(vote_distance_weights):  ('Freiberg', [('Freiberg', 0.8333333333333334),
↪  ('Freising', 0.16666666666666669)])
Ergebnis(vote_distance_weights):  ('Lüneburg', [('Lüneburg', 0.5), ('Duisburg',
↪  0.5)])
Ergebnis(vote_distance_weights):  ('Freiberg', [('Freiberg', 0.8333333333333334),
↪  ('Freising', 0.16666666666666669)])
Ergebnis(vote_distance_weights):  ('Hamburg', [('Hamburg', 0.7142857142857143),
↪  ('Bamberg', 0.28571428571428575)])
Ergebnis(vote_distance_weights):  ('Saarlouis', [('Saarlouis',
↪  0.8387096774193549), ('Bayreuth', 0.16129032258064516)])
```

Marvin und James führen uns in die Problematik des nächsten Beispiels ein:

Kannst du Marvin und James helfen?

Sie benötigen ein Englisch-Wörterbuch und einen k-nächsten-Nachbarn-Klassifikator, um das folgende Problem zu lösen. Sollten Sie mit Linux (speziell Ubuntu) arbeiten, finden Sie eine Datei mit einem British-English-Wörterbuch unter /usr/share/dict/british-english. Ansonsten können Sie die Datei hier herunterladen (british-english.txt).

Wir verwenden im nächsten Beispiel Worte, die ziemlich falsch geschrieben sind. Wir sehen dass unsere einfache vote_prob Funktion gute Arbeit leistet, ausser in zwei Fällen: Bei der Korrektur von “holpposs” zu “helpless” und “blagrufoo” zu “barefoot”. Während unsere vote_distance_weights Funktion in allen Fällen richtig korrigiert. Zugegeben, wir haben an “liberty” gedacht, als wir “liberdi” geschrieben haben. “liberal” ist aber auch eine gute Wahl.

```
words = []
with open("british-english.txt") as fh:
    for line in fh:
        word = line.strip()
        words.append(word)

for word in ["holpful", "kundnoss", "holpposs", "thoes", "innerstand",
             "blagrufoo", "liberdi"]:
    neighbors = get_neighbors(words,
                              words,
                              word,
```

```
                                3,
                                distance=levenshtein)

    print("vote_distance_weights: ", vote_distance_weights(neighbors,
                                                           all_results=False))
    print("vote_prob: ", vote_prob(neighbors))
```

```
vote_distance_weights:  ('helpful', 0.5555555555555556)
vote_prob:  ('helpful', 0.3333333333333333)
vote_distance_weights:  ('kindness', 0.5)
vote_prob:  ('kindness', 0.3333333333333333)
vote_distance_weights:  ('helpless', 0.3333333333333333)
vote_prob:  ('helpless', 0.3333333333333333)
vote_distance_weights:  ('hoes', 0.3333333333333333)
vote_prob:  ('hoes', 0.3333333333333333)
vote_distance_weights:  ('understand', 0.5)
vote_prob:  ('understand', 0.3333333333333333)
vote_distance_weights:  ('barefoot', 0.4333333333333333)
vote_prob:  ('barefoot', 0.3333333333333333)
vote_distance_weights:  ('liberal', 0.4)
vote_prob:  ('liberal', 0.3333333333333333)
```

9 Naechste Nachbarn Klassifikation Sklearn

9.0.1 Nächste-Nachbarn-Klassifikation mit sklearn

Einführung

Die zugrundeliegenden Konzepte des K-Nächsten-Nachbarn-Klassifikators (KNN) finden Sie im Kapitel [Nächste-Nachbarn-Klassifikation] (naechste_nachbarn_klassifikation.php) unseres Tutorials über Maschinelles Lernen. In diesem Kapitel zeigten wir auch einfache Funktionen, die in Python geschrieben wurden, um die grundlegenden Prinzipien zu demonstrieren.

Anstatt diese Funktionen zu nutzen, - obwohl sie beeindruckende Ergebnisse zeigten, - empfehlen wir, die Funktionalitäten des `sklearn-Moduls` zu verwenden.

kNN-Klassifikatoren von sklearn

`neighbors` ist ein Paket des `sklearn` module, welches Funktionalitäten für Nächste-Nachbarn-Klassifikatoren zur Verfügung stellt.

Die Klassen in `sklearn.neighbors` können sowohl numpy arrays als auch scipy.sparse-Matrizen als Eingabe verarbeiten. Für dichte Matrizen werden eine große Anzahl möglicher Entfernungsmetriken unterstützt. Für dünnbesetzte Matrizen werden beliebige Minkowski-Metriken für Suchvorgänge unterstützt.

scikit-learn implementiert zwei verschiedene nächste-Nachbarn-Klassifikatoren:

K-Nächster-Nachbar

basiert auf den k-nächsten Nachbarn der Stichprobe, die zu klassifizieren sind. Der Wert `k` ist ein vom Benutzer spezifizierter Wert. Dies ist der am häufigsten engewendete Klassifikator der beiden.

Radius-Nächster-Nachbar

basiert auf der Anzahl der Nachbarn einer Stichprobe innerhalb eines festen Umkreises `r`. `r` ist eine von den Anwendenden zu bestimmende `flot`-Zahl. Dieser Klassifikationtyp wird seltener angewendet.

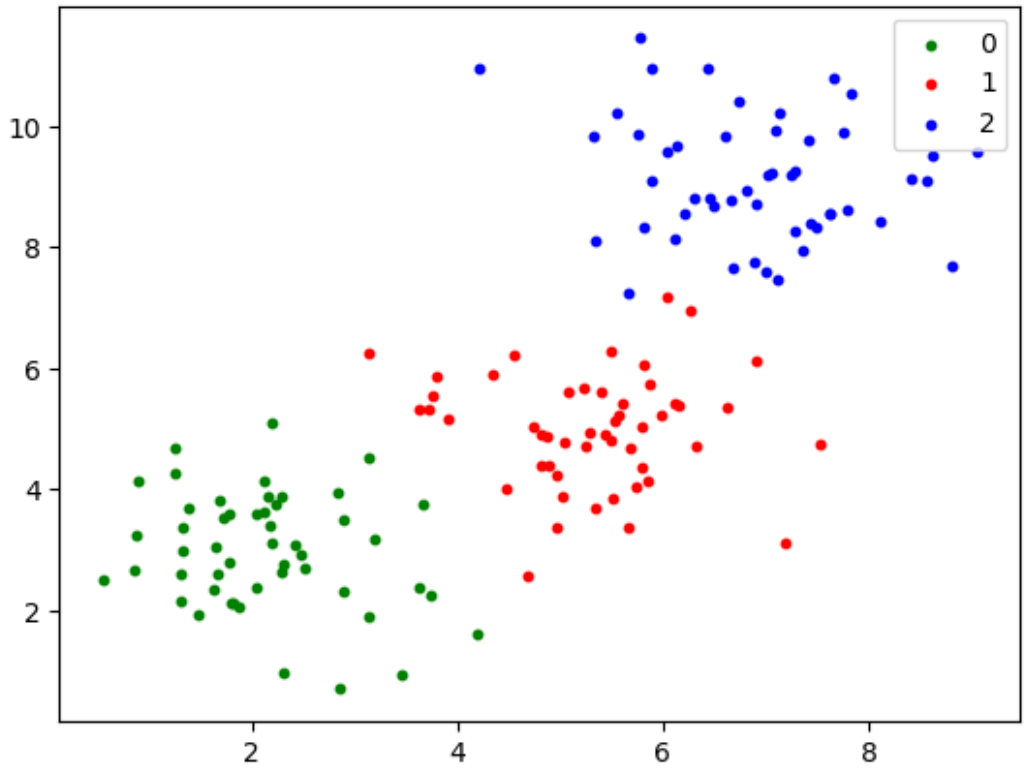
KNeighborsClassifier Wir erstellen einen künstlichen Datensatz mit drei Klassen, um den K-Nächster Nachbarnklassifizierer `KNeighborsClassifier` von `sklearn.neighbors` testen.

```
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
import numpy as np
```

```
centers = [[2, 3], [5, 5], [7, 9]]
n_classes = len(centers)
data, labels = make_blobs(n_samples=150,
                          centers=np.array(centers),
                          random_state=1)
```

Let us visualize what we have created:

```
import matplotlib.pyplot as plt

colours = ('green', 'red', 'blue')
n_classes = 3

fig, ax = plt.subplots()
for n_class in range(0, n_classes):
    ax.scatter(data[labels==n_class, 0], data[labels==n_class, 1],
               c=colours[n_class], s=10, label=str(n_class))



ax.legend(loc='upper right');
```



Zuerst müssen wir die Daten jetzt in separate Trainings- und Testsets aufteilen, ein Prozess, den wir in unserem Kapitel Aufteilungen der Datensätze unseres Kurses “Maschinelles Lernen” ausführlich beschrieben haben.

```
from sklearn.model_selection import train_test_split
res = train_test_split(data, labels,
```

```
                        train_size=0.8,
                        test_size=0.2,
                        random_state=1)

train_data, test_data, train_labels, test_labels = res
```

Wir sind nun bereit die Klassifikation mit `KNeighborsClassifier` durchzuführen:

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier()
knn.fit(train_data, train_labels)

predicted = knn.predict(test_data)
print("Predictions from the classifier:")
print(predicted)
print("Target values:")
print(test_labels)
```

```
Predictions from the classifier:
[2 2 2 0 0 1 1 2 2 1 0 1 0 0 2 0 0 0 1 0 0 1 2 2 0 0 0 1 2 1]
Target values:
[2 2 2 0 0 1 1 2 2 1 0 1 0 0 2 0 0 0 1 0 0 1 1 2 0 0 0 1 2 1]
```

Der Parameter `metric` ist standardmäßig Minkowski. Wir haben die Minkowski-Distanz in unserem Kapitel Nächste-Nachbarn-Klassifikation unseres Kurses “Maschinelles Lernen” ausführlich erklärt und in Python-Code implementiert. Der Parameter `p` entspricht dem `p` in der Minkowski-Formel: Wenn `p` auf 1 gesetzt wird, entspricht dies der Verwendung der Manhattan-Distanz, und die euklidische Distanz wird verwendet, wenn `p` den Wert 2 zugewiesen bekommt.

Der Parameter `algorithm` bestimmt, welcher Algorithmus verwendet wird, nämlich:

- `ball_tree` verwendet `BallTree`
- `kd_tree` verwendet `KDTree`
- `brute` verwendet eine brute-force-Suche. Wir setzen den Parameter auf auto, der versucht, basierend auf den Werten, die der fit-Methode übergeben werden, den geeignetsten Algorithmus zu wählen.

Der Parameter `leaf_size` wird von `BallTree` oder `KDTree` benötigt. Es kann die Geschwindigkeit des Aufbaus und der Abfrage sowie den benötigten Speicherplatz für den Baum beeinflussen. Der optimale Wert hängt von der Art des Problems ab.

```
# Explizite Parameter dienen hier nur der Demonstration.
knn = KNeighborsClassifier(
    n_neighbors=5,
    weights="uniform",
    metric="minkowski",
    p=2,
)
knn.fit(train_data, train_labels)
predicted = knn.predict(test_data)
print("Vorhersagen:", predicted)
print("Sollwerte:    ", test_labels)
```

```
Vorhersagen: [2 2 2 0 0 1 1 2 2 1 0 1 0 0 2 0 0 0 1 0 0 1 2 2 0 0 0 1 2 1]
```

```
Sollwerte:     [2 2 2 0 0 1 1 2 2 1 0 1 0 0 2 0 0 0 1 0 0 1 1 2 0 0 0 1 2 1]
```

Um das Ergebnis zu evaluieren benötigen wir die Funktion `accuracy_score` aus dem Modul `sklearn.metrics`. Um zu sehen, wie sie arbeitet, benutzen wir folgendes kleines Beispiel:

```
from sklearn.metrics import accuracy_score
y_pred = [0, 2, 1, 3, 2]
y_true = [0, 1, 2, 3, 2]
print(accuracy_score(y_true, y_pred))
print(accuracy_score(y_true, y_pred, normalize=False))
```

```
0.6
3.0
```

Der erste Aufruf von `accuracy_score` liefert `0.6` zurück, was bedeutet, dass 60% der Stichproben korrekt bestimmt wurden. Der Parameter `normalize` wurde per Default auf `True` gesetzt. Beim zweiten Aufruf setzen wir `normalize` auf `False`. Dies bedeutet, dass wir die Anzahl der Elemente erhalten, die korrekt vorhergesagt worden waren.

Nun sind wir bereit, unsere Ergebnisse aus der Klassifizierung des vorigen Beispiels zu bewerten:

```
print(accuracy_score(test_labels, predicted))
```

```
0.9666666666666667
```

```
data[:5], labels[:5]
```

```
(array([[5.76314662, 9.87583893],
        [7.01652757, 9.17718772],
        [8.55880554, 9.1094027 ],
        [4.86355526, 4.88094581],
        [6.12141771, 5.40890054]]),
 array([2, 2, 2, 1, 1]))
```

Im folgenden Beispiel benutzen wir den Iris-Datensatz:

```
from sklearn import datasets
from sklearn.model_selection import train_test_split

iris = datasets.load_iris()
data, labels = iris.data, iris.target

res = train_test_split(data, labels,
                       train_size=0.8,
                       test_size=0.2,
                       random_state=42)
train_data, test_data, train_labels, test_labels = res
```

```
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(train_data, train_labels)

print("Ergebnisse der Klassifikation:")
test_data_predicted = knn.predict(test_data)
```

```
print(test_data_predicted)
print("Sollwerte:")
print(test_labels)
```

```
Ergebnisse der Klassifikation:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0]
Sollwerte:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0]
```

```
print(accuracy_score(test_labels, test_data_predicted))
```

```
1.0
```

```
print("Ergebnisse der Klassifikation:")
learn_data_predicted = knn.predict(train_data)
print(learn_data_predicted)
print("Sollwerte:")
print(train_labels)
print(accuracy_score(learn_data_predicted, train_labels))
```

```
Ergebnisse der Klassifikation:
[0 0 1 0 0 2 1 0 0 0 2 1 1 0 0 1 2 2 1 2 1 2 1 0 2 1 0 0 0 1 2 0 0 0 1 0 1
 2 0 1 2 0 2 2 1 1 2 1 0 1 2 0 0 1 1 0 2 0 0 2 1 2 2 2 2 1 0 0 2 2 0 0 0 2
 2 0 2 2 0 1 1 2 1 2 0 2 1 2 1 1 1 0 1 1 0 1 2 2 0 1 2 2 0 2 0 1 2 2 1 2 1
 1 2 2 0 1 1 0 1 2]
Sollwerte:
[0 0 1 0 0 2 1 0 0 0 2 1 1 0 0 1 2 2 1 2 1 2 1 0 2 1 0 0 0 1 2 0 0 0 1 0 1
 2 0 1 2 0 2 2 1 1 2 1 0 1 2 0 0 1 1 0 2 0 0 1 1 2 1 2 2 1 0 0 2 2 0 0 0 1
 2 0 2 2 0 1 1 2 1 2 0 2 1 2 1 1 1 0 1 1 0 1 2 2 0 1 2 2 0 2 0 1 2 2 1 2 1
 1 2 2 0 1 2 0 1 2]
0.9666666666666667
```

RadiusNeighborsClassifier Der K-Nächste-Nachbar-Klassifikator arbeitet, indem er einen Kreis um die unbekannte Stichprobe (d. h. das zu klassifizierende Element) erweitert, bis er genau k benachbarte Elemente umfasst. Im Gegensatz dazu verwendet der Radius Neighbors Classifier einen festen Radius, um seinen Suchraum zu definieren. Er identifiziert alle Elemente innerhalb des Trainingsdatensatzes, die innerhalb dieses vordefinierten Radius um das zu klassifizierende Element liegen. Folglich führt die Methode mit dem festen Radius dazu, dass dichte Regionen der Merkmalsverteilung informativere Beiträge liefern, während spärliche Regionen weniger Informationen beisteuern.

Im Zusammenhang mit dem KNN-Klassifikator (k-nearest neighbor) erweitert der Algorithmus mit zunehmender Anzahl der Nachbarn (k) seinen Suchradius, um mehr benachbarte Punkte in den Klassifikationsprozess einzubeziehen. Wird k zu hoch angesetzt oder ist die Umgebung zu spärlich besetzt, kann es passieren, dass der Klassifikator Punkte einer anderen Klasse einbezieht, die weit von der zu klassifizierenden Probe entfernt sind.

Diese Situation kann eintreten, wenn der Datensatz Regionen enthält, in denen die Klassen nicht gut voneinander getrennt sind, was zu überlappenden Regionen zwischen den Klassen führt. Wenn sich der Suchradius vergrößert, kann der Algorithmus beginnen, Punkte aus weit entfernten Klassen einzubeziehen, was zu Fehlklassifizierungen oder Leistungseinbußen führen kann.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs

# Generate the dataset
centers = [[-2, 2], [2, 2]]
X, y = make_blobs(n_samples=100, centers=centers, cluster_std=0.5,
↪  random_state=42)

X[:3], y[:3]
```

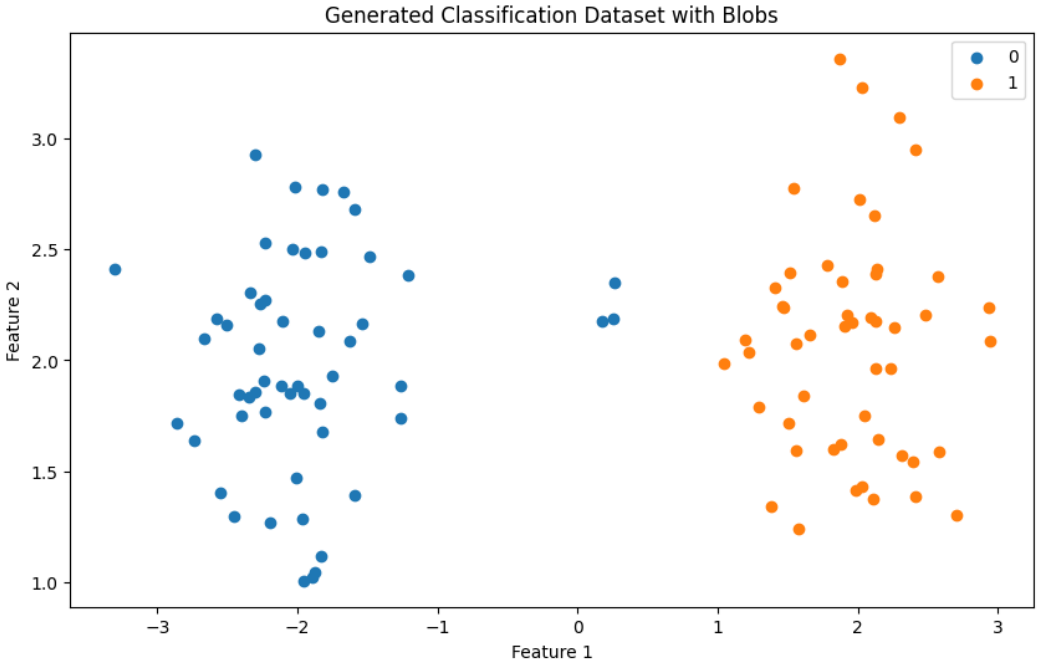
```
(array([[ 2.23661881,  1.96358554],
        [-2.41960876,  1.84539381],
        [ 1.04061439,  1.98674306]]),
 array([1, 0, 1]))
```

Wir erstellen nun einen isolierten Cluster von 3 Elementen außerhalb des Hauptclusters der Klasse 0. Wir fügen diese Elemente mit `concatenate` zum Hauptcluster hinzu:

```
centers = [[0.2, 2.2]]
X1, y1 = make_blobs(n_samples=3, centers=centers, cluster_std=0.1,
↪  random_state=42)

X = np.concatenate((X, X1), axis=0)
y = np.concatenate((y, y1))
```

Wir wollen nun die erzeugten Cluster visualisieren:

```
# Plot the data
plt.figure(figsize=(10, 6))
for class_value in range(2):
    # select indices of points with the current class label
    row_ix = np.where(y == class_value)
    # plot points for the current class
    plt.scatter(X[row_ix, 0], X[row_ix, 1], label=str(class_value))
plt.title('Generated Classification Dataset with Blobs')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.show()
```



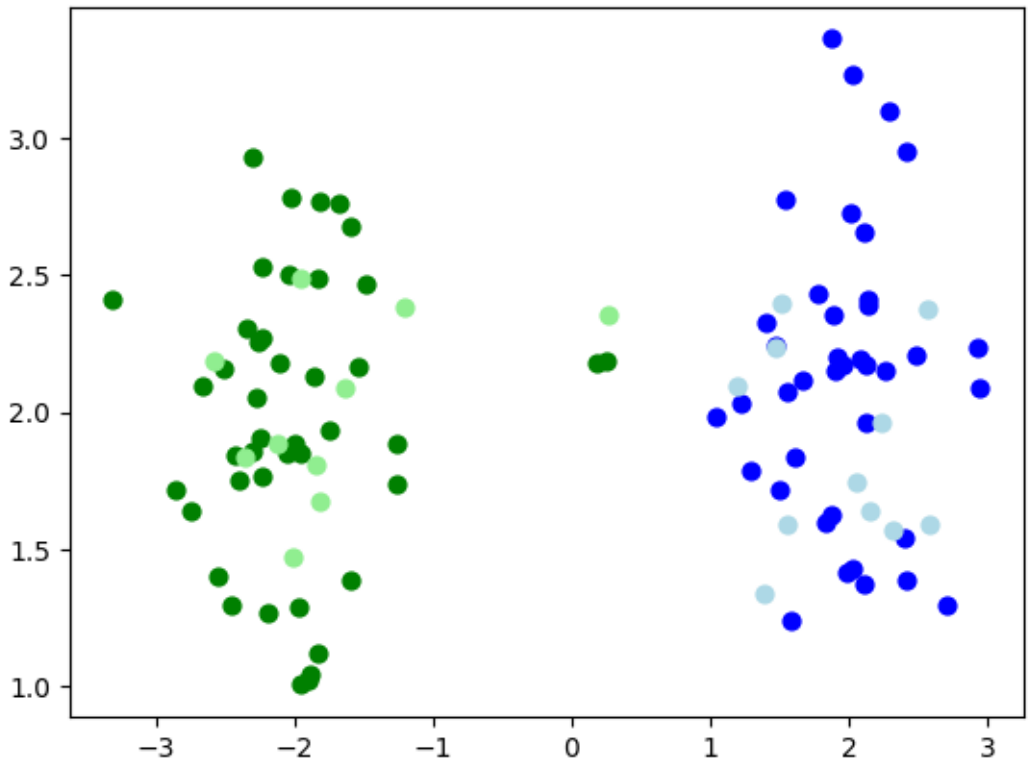
Mit dem folgenden Code teilen wir den Datensatz sowohl für die Merkmale als auch für die Bezeichnungen in Trainings- und Testsätze auf, wobei 80 % der Daten für das Training und 20 % (test_size=0.2) für das Testen verwendet werden.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
↪  random_state=42)
```

Wir werden nun unsere Trainings- und Testdaten visualisieren. Grüne und blaue Punkte repräsentieren die Trainingsdaten, während hellgrüne und hellblaue Punkte die Testdaten darstellen. Wir sehen, dass innerhalb unserer grünlichen Klasse (Klasse 1) drei Punkte in unmittelbarer Nähe zur bläulichen Klasse liegen. In solchen Fällen schneidet der Radius Neighbors Classifier tendenziell besser ab, es sei denn, wir setzen den Wert von k auf eine Zahl kleiner als 4.

```
import matplotlib.pyplot as plt


fig, ax = plt.subplots()
# plotting learn data
colours = ('green', 'blue')
for n_class in range(2):
    ax.scatter(X_train[y_train==n_class][:, 0],
               X_train[y_train==n_class][:, 1],
               c=colours[n_class], s=40, label=str(n_class))


# plotting test data
colours = ('lightgreen', 'lightblue')
for n_class in range(2):
    ax.scatter(X_test[y_test==n_class][:, 0],
               X_test[y_test==n_class][:, 1],
               c=colours[n_class], s=40, label=str(n_class))
```

```
ax.plot();
```



Wir werden unsere Datensätze mit Hilfe des “RadiusNeighborsClassifier” klassifizieren:

```
from sklearn.neighbors import RadiusNeighborsClassifier

# Instantiate the RadiusNeighborsClassifier
rnc = RadiusNeighborsClassifier(radius=0.5)

# Fit the model to the training data
rnc.fit(X_train, y_train)

# Predict the labels for the testing data
y_pred = rnc.predict(X_test)

# Evaluate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

```
Accuracy: 1.0
```

Und nun zum Vergleich mit dem `KNeighborsClassifier`:

```
# Instantiate the KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=15)

# Fit the model to the training data
knn.fit(X_train, y_train)
```

```
# Predict the labels for the testing data
y_pred = knn.predict(X_test)

# Evaluate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

```
Accuracy: 0.9523809523809523
```

Wenn man die Anzahl der Nachbarn passend festlegt, kann der k-nearest neighbors-Klassifikator auch in diesem Szenario effektiv arbeiten.

```
knn = KNeighborsClassifier(n_neighbors=4)

# Fit the model to the training data
knn.fit(X_train, y_train)

# Predict the labels for the testing data
y_pred = knn.predict(X_test)

# Evaluate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

```
Accuracy: 1.0
```

Ein weiteres einfaches Beispiel mit RadusNeighborsClassifier

```
from sklearn.neighbors import RadiusNeighborsClassifier

X = [[0, 1], [0.5, 1], [3, 1], [3, 2], [1.3, 0.8], [2.5, 2.5]]
y = [0, 0, 1, 1, 0, 1]

neigh = RadiusNeighborsClassifier(radius=1.0)
neigh.fit(X, y)

print(neigh.predict([[1.5, 1.2]]))

print(neigh.predict([[3.1, 2.1]]))
```

```
[0]
[1]
```

Wenn wir versuchen, eine Vorhersage auf Daten wie [30, 20] zu machen, kann der Algorithmus keine Nachbarn für den Radius 1,0 finden. Es wird also eine Ausnahme mit dem folgenden Text ausgelöst:

```
ValueError: No neighbors found for test samples array([0]), you can try using larger radius
```

Es gibt einen Parameter zur Einstellung des Labels für Ausreißer, d.h. outlier_label.

Es gibt drei Möglichkeiten, es zu verwenden:

1) Manuelle Beschriftung: str- oder int-Beschriftung (sollte der gleiche Typ sein, den wir in unseren Daten verwenden) oder Liste manueller Beschriftungen, wenn mehrere Ausgaben verwendet werden.
2) Es kann auf den Wert 'most_frequent' gesetzt werden. Dadurch wird das am häufigsten vorkommende Label des Datensatzes Ausreißern zugewiesen.
3) Wenn es auf None (Standardwert) gesetzt ist, wird eine ValueError ausgelöst, wenn ein Ausreißer erkannt wird.

Lassen Sie uns das noch einmal mit 'most_frequent' machen.

```
neigh = RadiusNeighborsClassifier(radius=1.0,
                                  outlier_label='most_frequent')
neigh.fit(X, y)

print(neigh.predict([[1.5, 1.2]]))

# the following is the previously mentioned outlier:
print(neigh.predict([[30, 20]]))
```

```
[0]
[0]
```

Alternativ setzen wir die Ausreißerklasse auf 2. Wir fügen ein Ausreißerelement zu unserem Lernsatz hinzu:

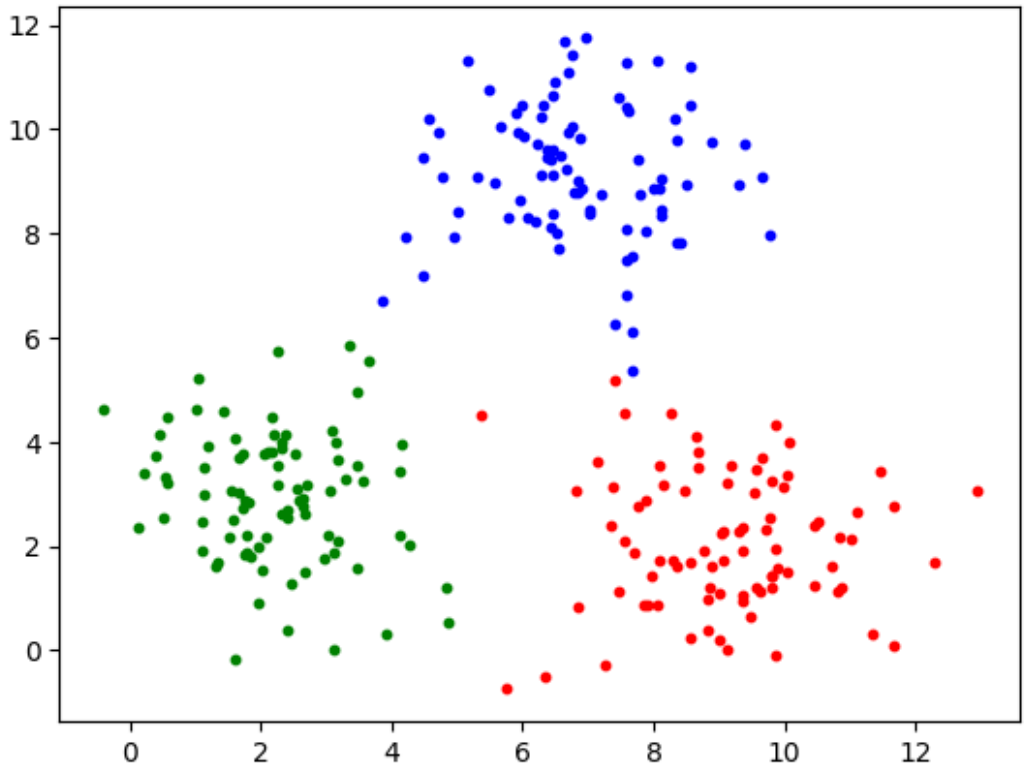
```
from sklearn.neighbors import RadiusNeighborsClassifier

X = [[0, 1], [0.5, 1], [3, 1], [3, 2], [1.3, 0.8], [2.5, 2.5], [2.4, 2.6],
↪  [10000, -2321]]
y = [0, 0, 1, 1, 0, 1, 1, 2]

neigh = RadiusNeighborsClassifier(radius=1.0,
                                  outlier_label=2)
neigh.fit(X, y)

print(neigh.predict([[1.5, 1.2]]))
print(neigh.predict([[30, 20]]))
```

```
[0]
[2]
```

Wir wollen nun wieder auf einem umfangreicheren Datenset arbeiten, welches wir zuerst mittels `make_blobs` generieren:

```
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
import numpy as np

centers = [[2, 3], [9, 2], [7, 9]]
n_classes = len(centers)
data, labels = make_blobs(n_samples=255,
                          centers=np.array(centers),
                          cluster_std = 1.3,
                          random_state=1)
```

```
import matplotlib.pyplot as plt

colours = ('green', 'red', 'blue')
n_classes = 3

fig, ax = plt.subplots()
for n_class in range(0, n_classes):
    ax.scatter(data[labels==n_class, 0], data[labels==n_class, 1],
               c=colours[n_class], s=10, label=str(n_class))
```




```
res = train_test_split(data, labels,
                       train_size=0.8,
                       test_size=0.2,
                       random_state=1)
train_data, test_data, train_labels, test_labels = res
```

Fügen wir eine Zeile am Ende der Trainingsdaten hinzu, die Ausreißerdaten enthält, d.h. die keiner Klasse angehören:

```
outlier = [4242.2, 4242.2]
train_data = np.vstack([train_data, outlier])
train_data[-3:]
```

```
array([[   8.42869523,    7.82787516],
       [   8.01064497,    8.84559748],
       [4242.2       , 4242.2       ]])
```

Nun müssen wir noch ein Label als `outlier`-Label festlegen.

```
outlier_label = len(np.unique(labels))
train_labels = np.append(train_labels, outlier_label)
train_labels[-10:]
```

```
array([0, 0, 0, 1, 0, 0, 0, 2, 2, 3])
```

Mit dem Befehl unique können wir überprüfen, welche Klassen verfügbar sind:

```
np.unique(train_labels)
```

```
array([0, 1, 2, 3])
```

Im folgenden Code initialisieren wir einen Radius-Neighbors-Classifier mit einem festgelegten Radius, trainieren ihn mit einem Trainingsdatensatz und verwenden dann den trainierten Classifier, um Labels für einen separaten Testdatensatz vorherzusagen.

```
rnn = RadiusNeighborsClassifier(radius=1)
rnn.fit(train_data, train_labels)
predicted = rnn.predict(test_data)
```

```
print(accuracy_score(test_labels, predicted))
```

```
1.0
```

Nun vekleinern wir den Radius:

```
rnn = RadiusNeighborsClassifier(radius=0.9,
                                outlier_label=outlier_label)
rnn.fit(train_data, train_labels)
predicted = rnn.predict(test_data)
print(accuracy_score(test_labels, predicted))
```

```
0.9803921568627451
```

Wir erzeugen weitere Ausreißer, um sie zu testen:

```
centers = [[100, 300]]
data_outliers, labels_outliers = make_blobs(n_samples=10,
                                  centers=np.array(centers),
                                  random_state=1)
```

```
predicted = rnn.predict(data_outliers)
predicted
```

```
array([3, 3, 3, 3, 3, 3, 3, 3, 3, 3])
```

Wir vergleichen dies nun mit einem k-nearest-neighbor classifier:

```
k = 15
knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(train_data, train_labels)
```

```
predicted = knn.predict(test_data)
print(accuracy_score(test_labels, predicted))
```

```
1.0
```

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(test_labels, predicted)
print(cm)
```

```
[[24  0  0]
 [ 0 18  0]
 [ 0  0  9]]
```

```
predicted = knn.predict(data_outliers)
predicted
```

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

Wir können sehen, dass alle Ausreißer fälschlicherweise als Klasse 2 klassifiziert wurden, weil dies die nächstgelegene vorhandene Klasse zu den Ausreißern ist. Im Folgenden erstellen wir drei Cluster von Ausreißern:

```
centers = [[100, 300], [10, -10], [-200, -200]]
data_outliers2, labels_outliers2 = make_blobs(n_samples=30,
                                              centers=np.array(centers),
                                              random_state=1)

predicted = knn.predict(data_outliers2)
predicted
```

```
array([2, 2, 2, 1, 0, 2, 1, 1, 1, 1, 1, 0, 0, 2, 0, 0, 2, 0, 2, 0, 1, 1,
       2, 2, 1, 2, 0, 1, 0, 0])
```

Die Ausreißer werden den vorhandenen Clustern zugeordnet, obwohl sie weit entfernt von ihnen liegen. Andererseits wird der Radius-Neighbors-Classifier sie als Ausreißer erkennen:

```
rnn = RadiusNeighborsClassifier(radius=0.9,
                                outlier_label=outlier_label)
rnn.fit(train_data, train_labels)
predicted = rnn.predict(data_outliers2)
predicted
```

```
array([3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
       3, 3, 3, 3, 3, 3, 3, 3])
```

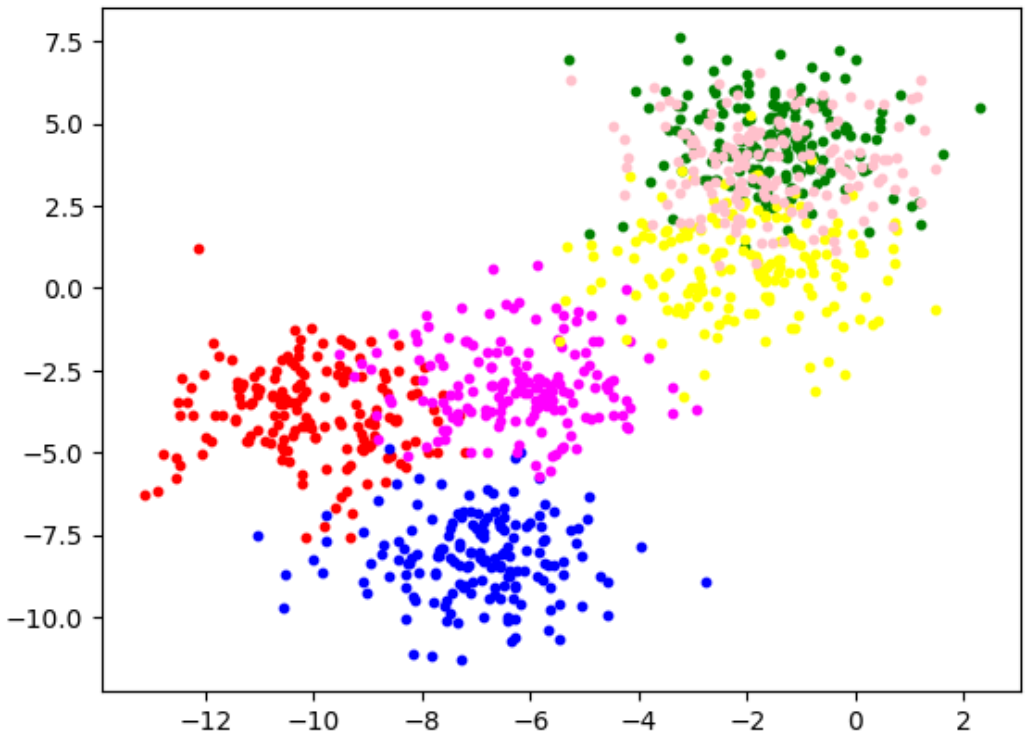
Bestimmung eines geeigneten k-Werts

Eine feste Formel für das optimale `k` gibt es nicht. Insbesondere ist die häufig zitierte Faustregel `k` √`n` kein allgemeines Optimierungsverfahren. `k` ist ein Hyperparameter und sollte auf den Trainingsdaten mit Validierung oder Kreuzvalidierung ausgewählt werden. Der Testsatz bleibt dabei unangetastet und wird erst für die abschließende Bewertung verwendet.

Da k-NN auf Distanzen beruht, kombinieren wir die Standardisierung und den Klassifikator in einer `Pipeline`. Dadurch wird der `StandardScaler` in jedem Cross-Validation-Split nur auf dem jeweiligen Trainingsanteil angepasst.

```
import matplotlib.pyplot as plt

from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
import numpy as np


n_classes = 6
data, labels = make_blobs(n_samples=1000,
                          centers=n_classes,
                          cluster_std = 1.3,
                          random_state=1)
```

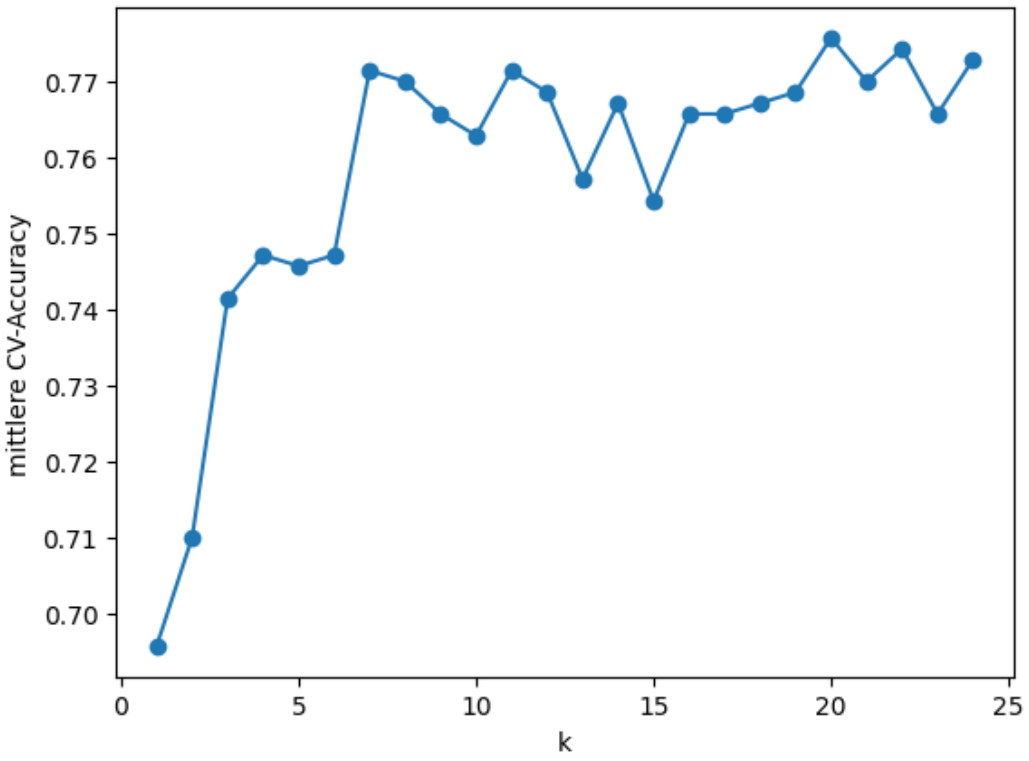
```
import matplotlib.pyplot as plt

colours = ('green', 'red', 'blue', 'magenta', 'yellow', 'pink')

fig, ax = plt.subplots()
for n_class in range(0, n_classes):
    ax.scatter(data[labels==n_class, 0], data[labels==n_class, 1],
               c=colours[n_class], s=10, label=str(n_class))
```




```
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
```

```
train_data, test_data, train_labels, test_labels = train_test_split(
    data, labels, test_size=0.3, random_state=1, stratify=labels
)

pipe = make_pipeline(StandardScaler(), KNeighborsClassifier())
param_grid = {"kneighborsclassifier__n_neighbors": range(1, 25)}
search = GridSearchCV(pipe, param_grid=param_grid, cv=5)
search.fit(train_data, train_labels)

ks = [p["kneighborsclassifier__n_neighbors"] for p in
↪  search.cv_results_["params"]]
scores = search.cv_results_["mean_test_score"]

fig, ax = plt.subplots()
ax.set_xlabel("k")
ax.set_ylabel("mittlere CV-Accuracy")
ax.plot(ks, scores, "o-")

print("Bestes k:", search.best_params_["kneighborsclassifier__n_neighbors"])
print("CV-Accuracy:", search.best_score_)
print("Test-Accuracy:", search.score(test_data, test_labels))
```

```
Bestes k: 20
CV-Accuracy: 0.7757142857142856
Test-Accuracy: 0.7933333333333333
```



Aufgaben

Aufgabe 1 Klassifizieren Sie die Daten in “strange_flowers.txt” mit einem k-Nächste-Nachbar-Klassifikator.

Aufgabe 2 Klassifizieren Sie die Daten in “fruits_data.txt” mit einem k nearest neighbor classifier.

Aufgabe 3 Benutzen Sie sklearn, um die Städtenamen aus dem vorherigen Kapitel zu korrigieren:

Die falsch geschriebenen Städtenamen sind: “Freiburg”, “Frieburg”, “Freiborg”, “Hamborg”, “Sahrluis”

Die richtigen Städtenamen sind in `data/city_names.txt` gespeichert.

Die Levenshtein-Distanz kann importiert werden mit

Levenshtein importieren”.

Es muss installiert werden:

pip install python-Levenshtein

Aufgabe 4 Machen Sie das Gleiche nun für die falsch geschriebenen Wörter “holpful”, “kundnoss”, “holpposs”, “thoes”, “innerstand”, “blagrufoo” und “liberdi”

Verwenden Sie die Datei british-english.txt für eine Liste aller richtig geschriebenen englischen Wörter!

Lösungen

Lösung zu Aufgabe 1

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, accuracy_score

dataset = pd.read_csv(
    "data/strange_flowers.txt",
    header=None,
    names=["red", "green", "blue", "size", "label"],
    sep=r"\s+",
)
dataset
```

Anstatt Pandas zu verwenden, um die Daten von “strange_flowers.txt” einzulesen, könnten wir auch “loadtxt” von numpy verwenden:

```
import numpy as np

raw_data = np.loadtxt("data/strange_flowers.txt")
data = raw_data[:,:-1]
labels = raw_data[:,-1]
```

Wir fahren nun mit dem Pandas DataFrame Objekt `dataset` fort, das wir mit `read_csv`

eingelesen haben:

```
data = dataset.drop(columns="label")
labels = dataset["label"]
X_train, X_test, y_train, y_test = train_test_split(
    data, labels, test_size=0.2, random_state=0, stratify=labels
)
X_train.head()
```

Für distanzbasierte Verfahren sollten Merkmale mit unterschiedlichen Größenordnungen skaliert werden. Statt den `StandardScaler` separat anzupassen, verwenden wir eine Pipeline. So gehören Vorverarbeitung und Klassifikator zu einem einzigen Modellobjekt und die Skalierung kann bei späterer Kreuzvalidierung nicht versehentlich Informationen aus Validierungsdaten verwenden.

```
classifier = make_pipeline(
    StandardScaler(),
    KNeighborsClassifier(n_neighbors=5),
)
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)
y_pred
```

Wir verwenden hier bewusst `k=5` als einfachen Ausgangswert. Für eine belastbare Modellwahl sollte `k` – wie oben gezeigt – mit Kreuzvalidierung auf den Trainingsdaten bestimmt werden.

```
cm = confusion_matrix(y_test, y_pred)
print(cm)
print("Accuracy:", accuracy_score(y_test, y_pred))
```

```
# Die Pipeline kann direkt auch für neue Daten verwendet werden:
classifier.predict(X_test[:3])
```

Die Konfusionsmatrix und die Accuracy wurden oben auf dem bislang ungesehenen Testanteil berechnet.

```
# bereits oben berechnet
```

```
# bereits oben berechnet
```

Lösung zu Aufgabe 2

```
import numpy as np
import pandas as pd
from sklearn.neighbors import NearestNeighbors
from collections import Counter

# Read in the CSV file
df = pd.read_csv('data/fruits_data.csv')

# Extract features (X) and labels (y)
```

```
X = df[['Sweetness', 'Acidity', 'Weight']].values
y = df['Fruit'].values

X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    random_state=0,
                                                    test_size=0.2)
```

```
classifier = make_pipeline(
    StandardScaler(),
    KNeighborsClassifier(n_neighbors=5),
)
```

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

```
cm = confusion_matrix(y_test, y_pred)
print(cm)
```

```
print(accuracy_score(y_test, y_pred))
```

```
# Für die Wahl von k kann auch hier GridSearchCV auf der Pipeline verwendet
↪ werden.
```

Lösung zu Aufgabe 3

```
import Levenshtein

# Load the file containing correct city names
with open('data/city_names.txt', 'r') as file:
    correct_city_names = file.readlines()
correct_city_names = [name.strip() for name in correct_city_names]

# Misspelled city names
misspelled_city_names = ["Freiburg", "Frieburg", "Freiborg", "Hamborg",
↪ "Sahrluis"]

# Find the closest match for each misspelled city name
for misspelled_city in misspelled_city_names:
    min_distance = float('inf')
    closest_match = None

    # Calculate Levenshtein distance to all correct city names
    for correct_city in correct_city_names:
        distance = Levenshtein.distance(misspelled_city, correct_city)
        if distance < min_distance:
            min_distance = distance
            closest_match = correct_city

    print(f"Closest match for '{misspelled_city}': {closest_match}")
```

```
Closest match for 'Freiburg': Freiberg
Closest match for 'Frieburg': Lüneburg
Closest match for 'Freiborg': Freiberg
Closest match for 'Hamborg': Hamburg
Closest match for 'Sahrluis': Saarlouis
```

Lösung zu Aufgabe 4

```
import Levenshtein

# Load the file containing correct city names
with open('british-english.txt', 'r') as file:
    correct_words = file.readlines()
correct_words = [name.strip() for name in correct_words]

# misspelled ords
misspelled_words = ["holpful", "kundnoss", "holpposs",
                    "thoes", "innerstand", "blagrufoo",
                    "liberdi"]

# Find the closest match for each misspelled city name
for misspelled_word in misspelled_words:
    min_distance = float('inf')
    closest_match = None

    # Calculate Levenshtein distance to all correct city names
    for correct_word in correct_words:
        distance = Levenshtein.distance(misspelled_word, correct_word)
        if distance < min_distance:
            min_distance = distance
            closest_match = correct_word

    print(f"Closest match for '{misspelled_word}': {closest_match}")
```

```
Closest match for 'holpful': helpful
Closest match for 'kundnoss': kindness
Closest match for 'holpposs': helpless
Closest match for 'thoes': hoes
Closest match for 'innerstand': understand
Closest match for 'blagrufoo': barefoot
Closest match for 'liberdi': liberal
```

Wir sind mit dem Ergebnis von ‘hoes’ für ‘thoes’ unzufrieden. Daher verbessern wir das Programm, um auch die zweitnächste Übereinstimmung anzuzeigen.

```
import Levenshtein

# Load the file containing correct words
with open('british-english.txt', 'r') as file:
    correct_words = file.readlines()
correct_words = [word.strip() for word in correct_words]

# Misspelled words
misspelled_words = ["holpful", "kundnoss", "holpposs",
                    "thoes", "innerstand", "blagrufoo",
                    "liberdi"]
```

```
# Find the closest and second closest match for each misspelled word
for misspelled_word in misspelled_words:
    distances = []

    # Calculate Levenshtein distance to all correct words
    for correct_word in correct_words:
        distance = Levenshtein.distance(misspelled_word, correct_word)
        distances.append((distance, correct_word))

    # Sort distances by the first element (distance)
    distances.sort(key=lambda x: x[0])

    # Get the closest and second closest matches
    closest_match = distances[0][1]
    second_closest_match = distances[1][1]

    print(f"Misspelled word: {misspelled_word}")
    print(f"Closest match: {closest_match}")
    print(f"Second closest match: {second_closest_match}")
    print()
```

```
Misspelled word: holpful
Closest match: helpful
Second closest match: doleful

Misspelled word: kundnoss
Closest match: kindness
Second closest match: fondness

Misspelled word: holpposs
Closest match: helpless
Second closest match: hippo's

Misspelled word: thoes
Closest match: hoes
Second closest match: shoes

Misspelled word: innerstand
Closest match: understand
Second closest match: interstate

Misspelled word: blagrufoo
Closest match: barefoot
Second closest match: Baguio

Misspelled word: liberdi
Closest match: liberal
Second closest match: liberty
```

10 Neuronale Netze

10.0.1 Neuronale Netzwerke

Einführung

Wenn wir von "Neuronalen Netzen" sprechen, meinen wir artifizielle neuronale Netze (ANN). Die Idee eines ANN basiert auf biologischen neuronalen Netzen, wie sie z.B. im Gehirn von Lebewesen vorkommen.

Die Basiseinheit eines neuronalen Netzes - sowohl eines artifiziellen als auch eines lebendigen - ist das Neuron. Ein biologisches Neuron besteht aus drei Hauptkomponenten: Dem Soma (Zellkörper), den Dendriten und dem Axon.

Die Dendriten zweigen aus dem Soma ab wie die Verzweigung eines Baumes und verjüngen sich mit jeder neuen Aufteilung. Sie erhalten Signale (Impulse) von anderen Neuronen an den Synapsen, wo sie sich mit anderen Zellen verbinden. Das Axon - wovon immer nur eines existiert - kommt ebenfalls aus dem Soma und erstreckt sich normalerweise über weitere Distanzen als die Dendriten. Es wird gebraucht, um das Ausgangssignal des Neurons an andere Neuronen zu übergeben oder genauer gesagt an die Synapsen anderer Neurone.

Biologisches Neuron

Die folgende Abbildung [Quasar Jarosz] (https://en.wikipedia.org/wiki/User:Quasar_Jarosz), aus Wikipedia, zeigt dies:

Abstraktion eines biologischen und eines artifiziellen Neurons

Auch wenn die obige Abbildung für eine/n Biologen/in schon eine Abstraktion darstellt, können wir sie noch mehr abstrahieren:

Ein Perzeptron ist eine stark vereinfachte mathematische Recheneinheit, die historisch von biologischen Neuronen inspiriert wurde. Es simuliert die biologische Funktionsweise eines Neurons jedoch nicht im Detail.

Was im Inneren eines solchen Perceptrons oder Neurons vorgeht, ist erstaunlich einfach. Die Eingangssignale werden mit den Gewichten (engl. "weights") multipliziert, jeder Eingang hat sein zugeordnetes Gewicht. So wird jeder Eingang individuell angepasst für jedes x_i. Wir können alle Eingänge als den Eingangsvektor verstehen und die zugehörigen Gewichte als den Gewichtsvektor.

Wenn also ein Signal ankommt, wird es mit dem Wert des Gewichts des individuellen Eingangs multipliziert, das diesem Eingang zugeordnet ist. So hat ein Neuron mit drei Eingängen auch drei Gewichte, die individuell angepasst werden können. Die Anpassung der Gewichte erfolgt normalerweise in der Lernphase des Netzes.

Danach werden die Eingangssignale summiert. Es ist auch möglich, einen sogenannten Bias

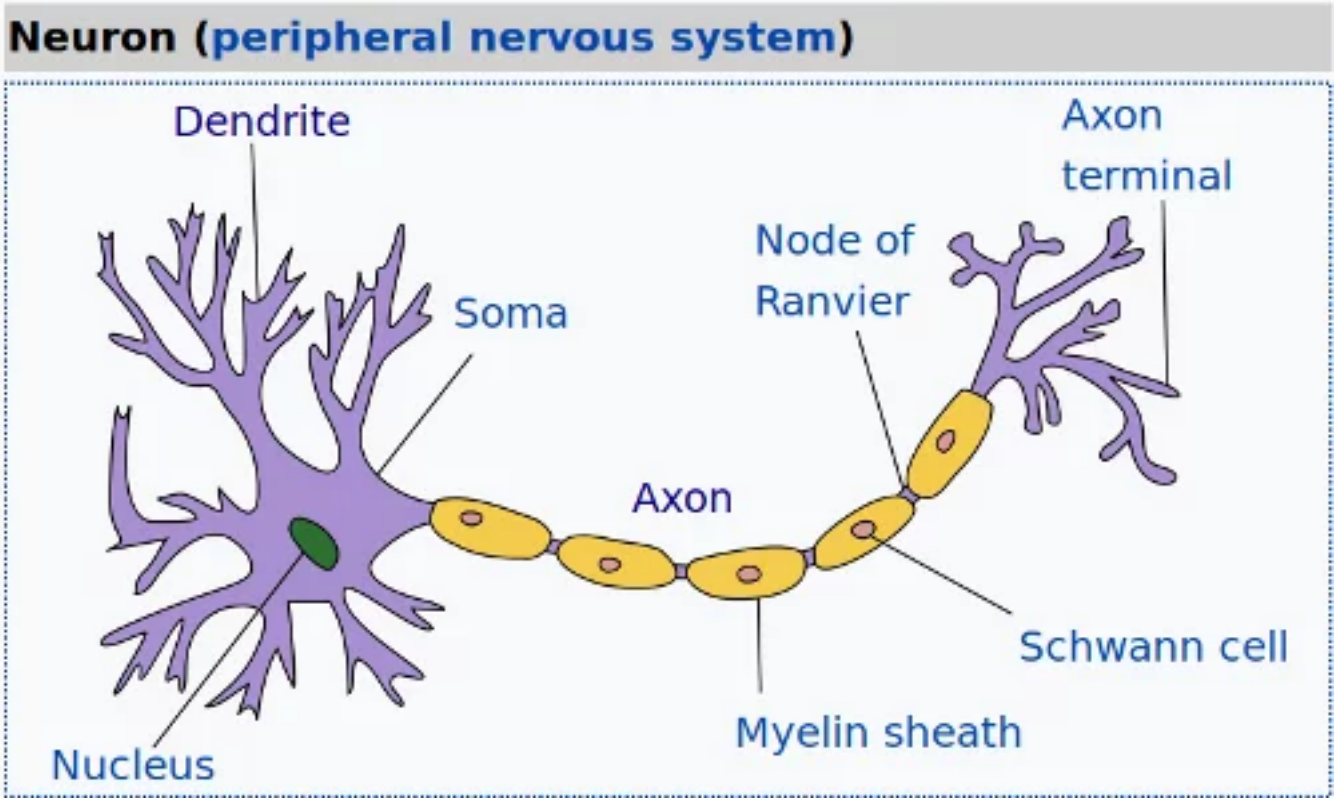


Abbildung 10.1: Image of a Neuron in biology

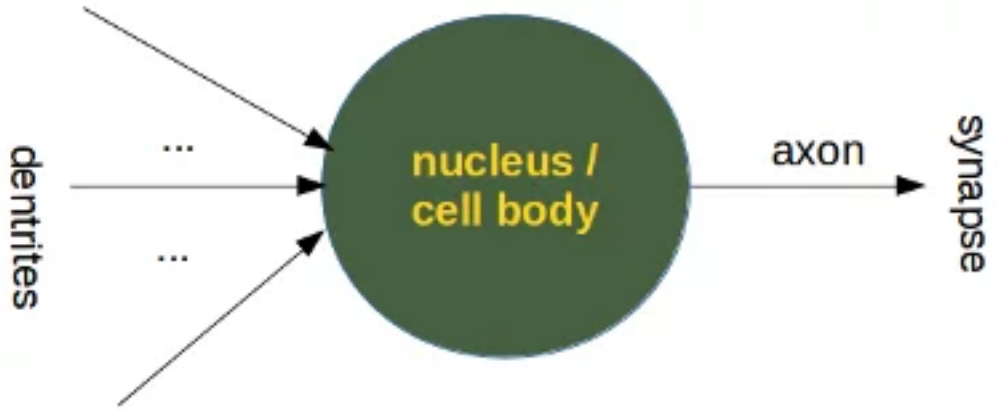


Abbildung 10.2: Abstract View of a Neuron

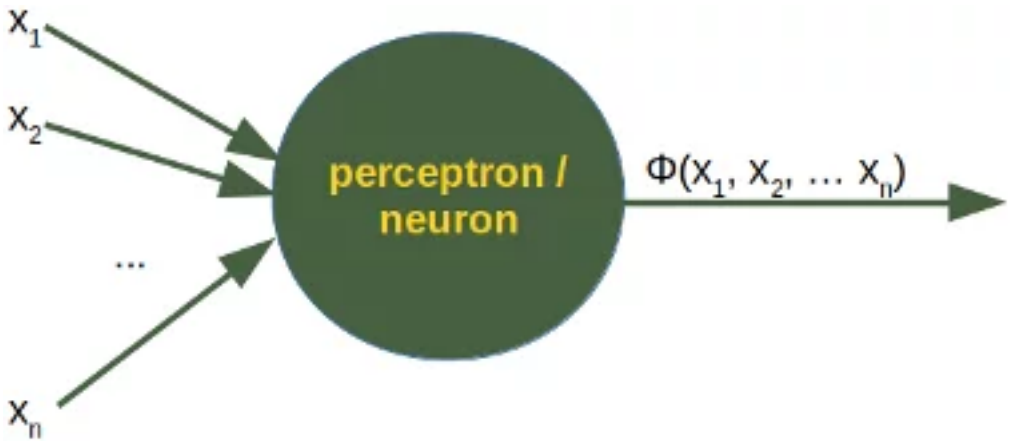


Abbildung 10.3: Image of a Perceptron of a Neural Network

"b" dazu zu addieren, der ein von anderen Neuronen unabhängiges Eingangssignal darstellt. Auch dieser Bias kann während der Lernphase angepasst werden.

Schliesslich wird er Ausgangswert festgelegt. Dafür wird eine Aktivierungs- oder Stufenfunktion Φ auf die gewichtete Summe der Eingangswerte angewendet.

Die einfachste Form einer Aktivierungsfunktion ist eine Binärfunktion. Wenn das Ergebnis der Summation grösser als ein Schwellenwert s ist, ist das Ergebnis von Φ 1, sonst 0.

$$\Phi(x) = \begin{cases} 1 & \text{wx + b > s} \\ 0 & \text{otherwise} \end{cases}$$

Anzahl Neuronen in der Biologie

Wir werden im den folgenden Kapiteln artifizielle neuronale Netze verschiedener Grössen und Strukturen untersuchen. Interessant ist ein Blick auf die Gesamtzahl der Neuronen von verschiedenen Lebewesen.

Beim Fadenwurm Caenorhabditis elegans, einem der meist erforschten Modellorganismnen der modernen Biologie gibt es so genau 385 Neuronen. (Quelle Wikipedia).

Beim Menschen geht man bei aktuellen Schätzungen von etwa 86 Milliarden Nervenzellen aus.(Max Planck Institut für Hirnforschung Berlin).

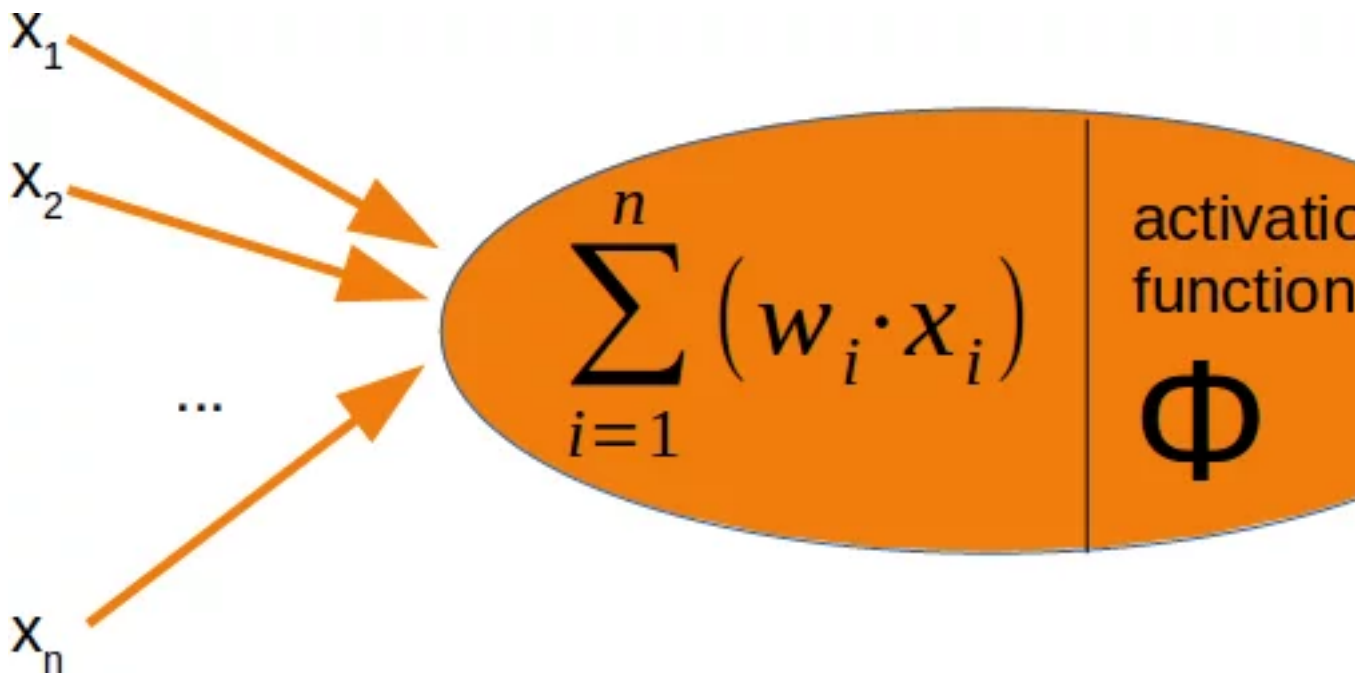


Abbildung 10.4: Detailled view of a Perceptron of a Neural Network

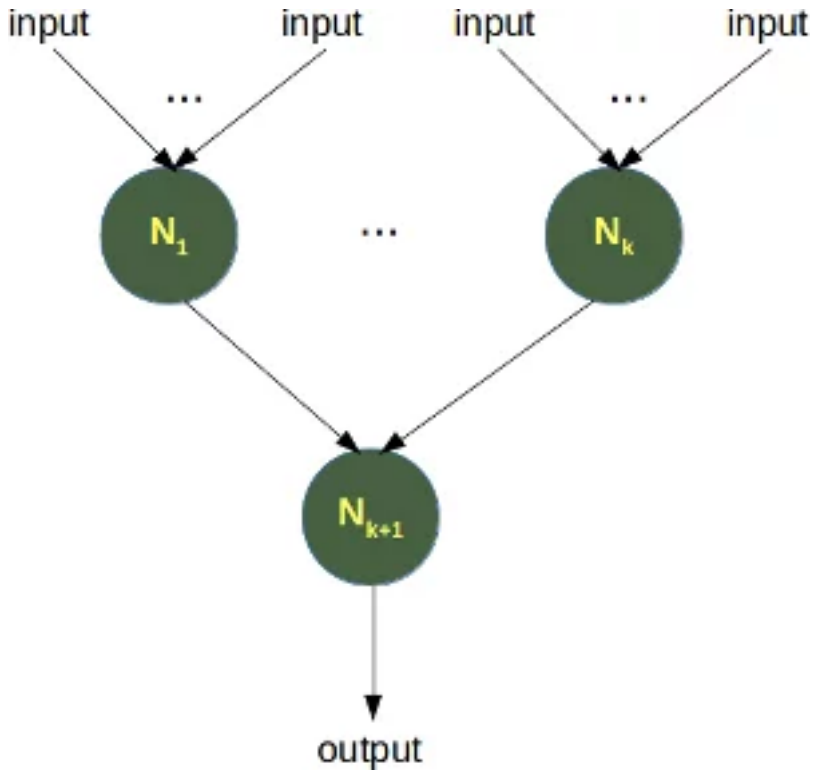


Abbildung 10.5: Building Principle of a Simple Artificial Neural Network

11 Trennlinien Zwischen Klassen

11.0.1 Von Trennlinien zu Neuronalen Netzen

Wir werden in diesem Kapitel unseres Tutorials ein einfaches neuronales Netz entwickeln. Es wird ein Netzwerk sein, das zwei Klassen trennen kann, die durch eine gerade Linie im zweidimensionalen Raum abzugrenzen sind.

Lineare Separierung

Bevor wir mit dem Programmieren des einfachen neuronalen Netzwerks beginnen, werden wir noch ein anderes Konzept vorstellen. Wir wollen nach Geraden suchen, die zwei Punkte oder besser die Punkte zweier Klassen in der Ebene trennen können.

Wir werden dabei zunächst nur Geraden berücksichtigen, die durch den Ursprung des Koordinatensystems gehen. Den generellen Fall, also beliebige Geraden, werden wir später im Tutorial behandeln.

Man könnte sich etwa vorstellen, dass man zwei Attribute hat, die die Eigenschaften eines essbaren Objektes, wie einer Frucht, beschreiben, zum Beispiel “Süsse” und “Säuerlichkeit”.

Dies könnte man als Punkte im zweidimensionalen Raum darstellen. Die X-Achse wird für die Werte der Süsse genutzt und die Y-Achse entsprechend für die Werte der Säuerlichkeit. Stellen Sie sich vor, wir hätten zwei Früchte als Punkte in dieser Ebene dargestellt, eine Orange an der Position (3.5,1.8) und eine Zitrone bei (1.1,3.9).

Nun könnten wir Trennlinien definieren, die Punkte trennen, die mehr zitronenartig oder mehr orangenartig sind.

Im folgenden Diagramm haben wir eine Zitrone und eine Orange eingetragen. Die grüne Linie trennt die beiden Punkte. Wir nehmen an, dass alle anderen Zitronen oberhalb dieser Linie sind und alle Orangen darunter.

Die grüne Gerade ist definiert als

$$y = mx$$

Dabei ist m die Steigung oder der Gradient der Gerade und x die unabhängige Variable

der Funktion

$$m = \frac{p_2}{p_1}$$

Das bedeutet, dass ein Punkt $P' = (p'_1, p'_2)$ auf dieser Geraden liegt, wenn die folgende Bedingung erfüllt ist:

$$mp'_1 - p'_2 = 0$$

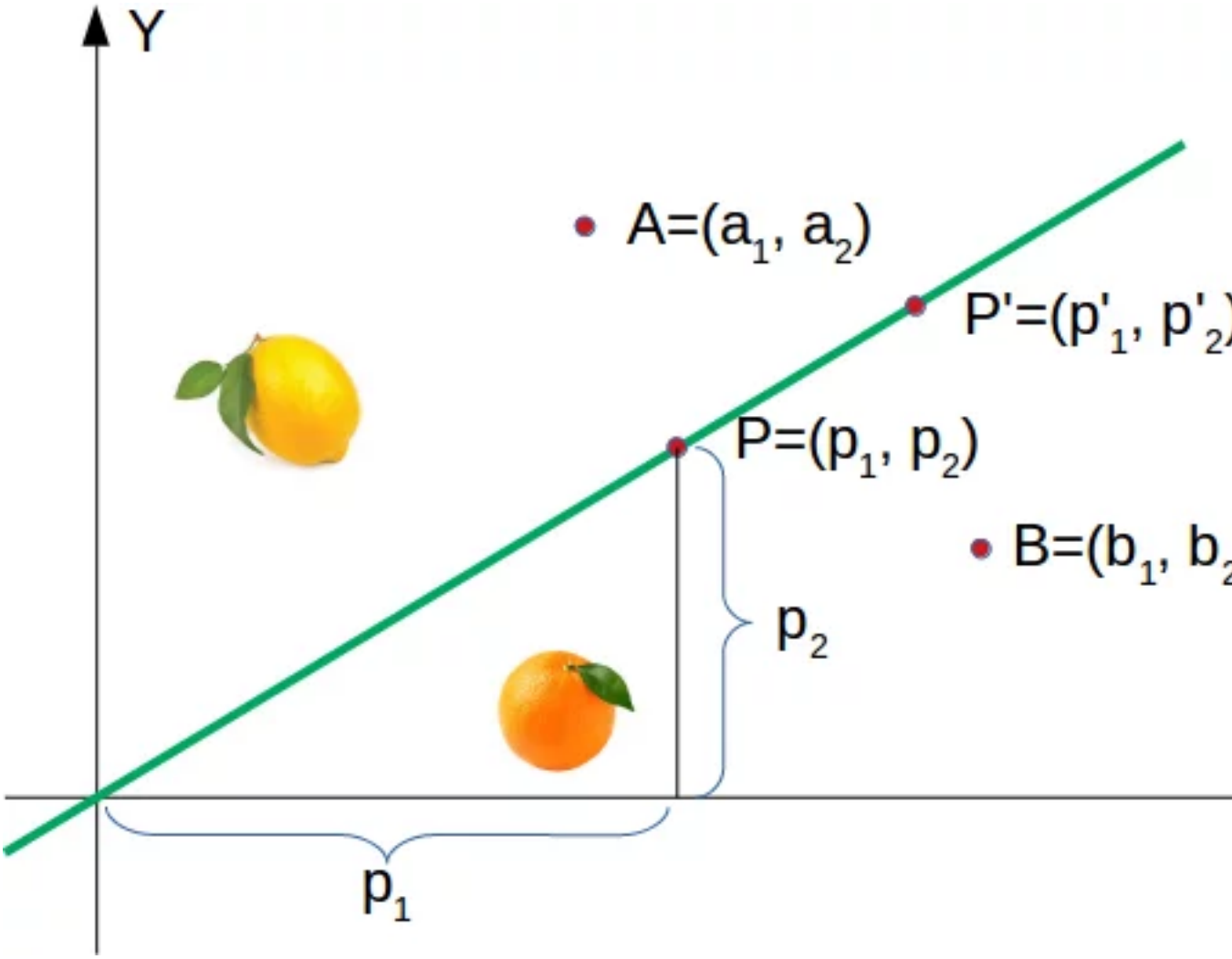
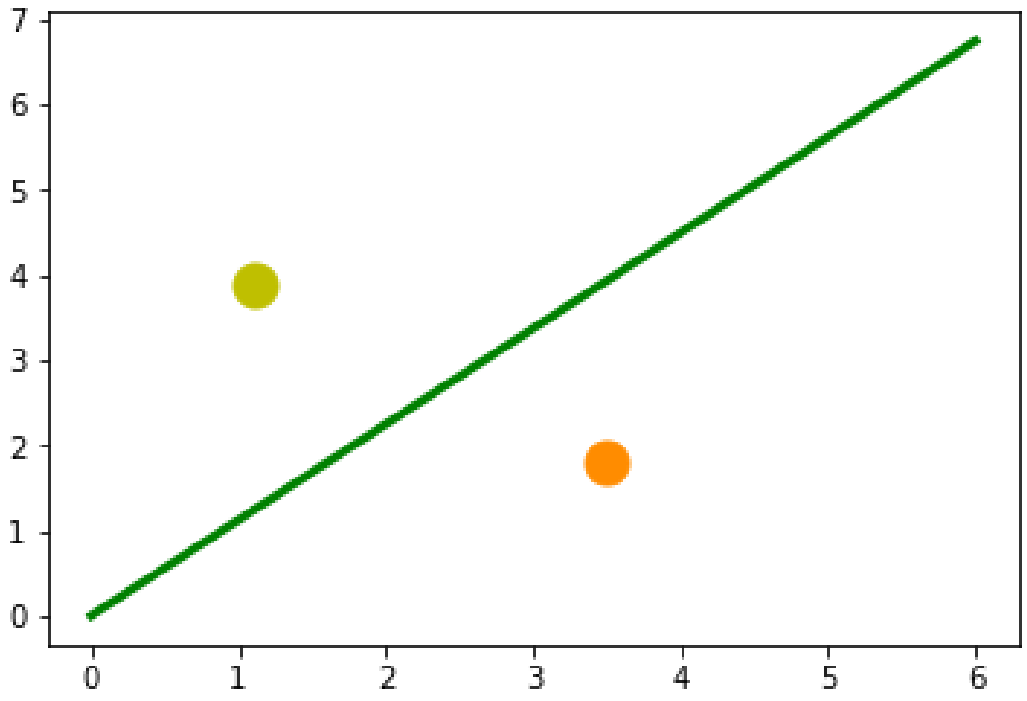


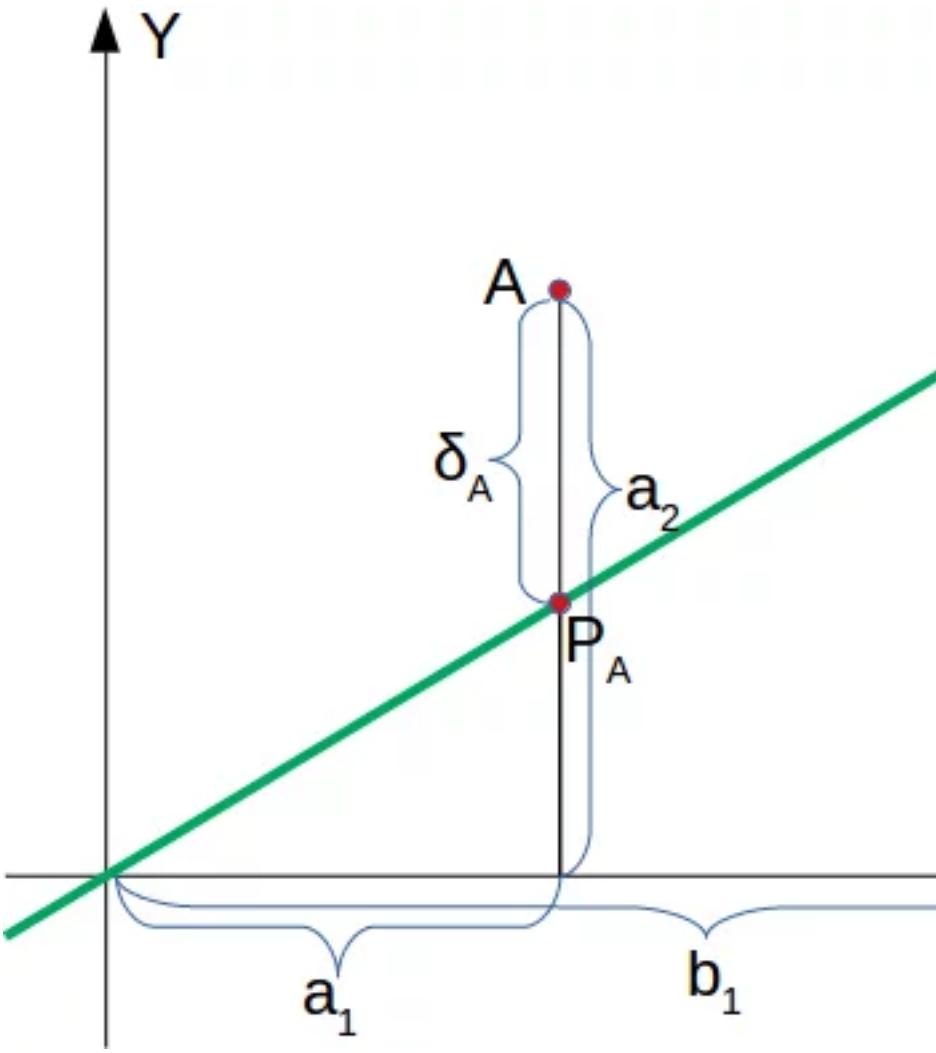
Abbildung 11.1: Divion boundary

Das folgende Python Programm erzeugt eine Darstellung der gerade beschriebenen Situation:

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

X = np.arange(0, 7)
fig,ax = plt.subplots()

ax.plot(3.5, 1.8, "or",
        color="darkorange",
        markersize=15)


point_on_line = (4, 4.5)
ax.plot(1.1, 3.9, "oy", markersize=15)
# calculate gradient:
m = point_on_line[1] / point_on_line[0]
ax.plot(X, m * X, "g-", linewidth=3)
plt.show()
```



Es ist klar, dass ein Punkt $A = (a_1, a_2)$ nicht auf der Geraden liegt, wenn $m \cdot a_1 - a_2$ nicht gleich 0 ist. Wir wollen aber mehr wissen. Wir wollen wissen, ob ein beliebiger Punkt oberhalb



oder unterhalb der Geraden liegt.

Wenn ein Punkt $m \cdot a_1 - a_2$ unterhalb der Geraden liegt, muss es ein $\delta_B > 0$ geben, so dass sich der Punkt $(b_1, b_2 + \delta_B)$ auf der Geraden befindet.

Das heisst, dass gilt:

$$m \cdot b_1 - (b_2 + \delta_B) = 0$$

Welches umgestellt werden kann zu:

$$m \cdot b_1 - b_2 = \delta_B$$

So haben wir schliesslich ein Kriterium dafür, dass ein Punkt unterhalb der Geraden liegt. $m \cdot b_1 - b_2$ ist positiv, weil δ_B positiv ist.

Die Überlegung für "der Punkt ist oberhalb der Geraden" ist analog: Wenn ein Punkt $A = (a_1, a_2)$ oberhalb der Geraden ist, muss es ein $\delta_A > 0$ geben, so dass der Punkt $(a_1, a_2 - \delta_A)$ auf der Geraden liegt.

Das heisst, dass gilt

$$m \cdot a_1 - (a_2 - \delta_A) = 0$$

was wiederum umgestellt werden kann zu

$$m \cdot a_1 - a_2 = -\delta_A$$

Zusammengefasst können wir sagen: Ein Punkt $P(p_1, p_2)$ liegt

- unterhalb der Geraden, wenn $m \cdot p_1 - p_2 > 0$
- auf der Geraden, wenn $m \cdot p_1 - p_2 = 0$
- oberhalb der Geraden, wenn $m \cdot p_1 - p_2 < 0$

Wir können das nun bei unseren Früchten überprüfen. Die Zitrone hat die Koordinaten (1.1,3.9) und die Orange (3.5,1.8). Der Punkt auf der Geraden, den wir nehmen, um unsere Trennungslinie zu definieren, hat die Koordinaten (4,4.5). So ist also m 4.5/4.

```
lemon = (1.1, 3.9)
orange = (3.5, 1.8)
m = 4.5 / 4

# check if orange is below the line,
# positive value is expected:
print(orange[0] * m - orange[1])

# check if lemon is above the line,
# negative value is expected:
print(lemon[0] * m - lemon[1])
```

```
2.1375
-2.6624999999999996
```

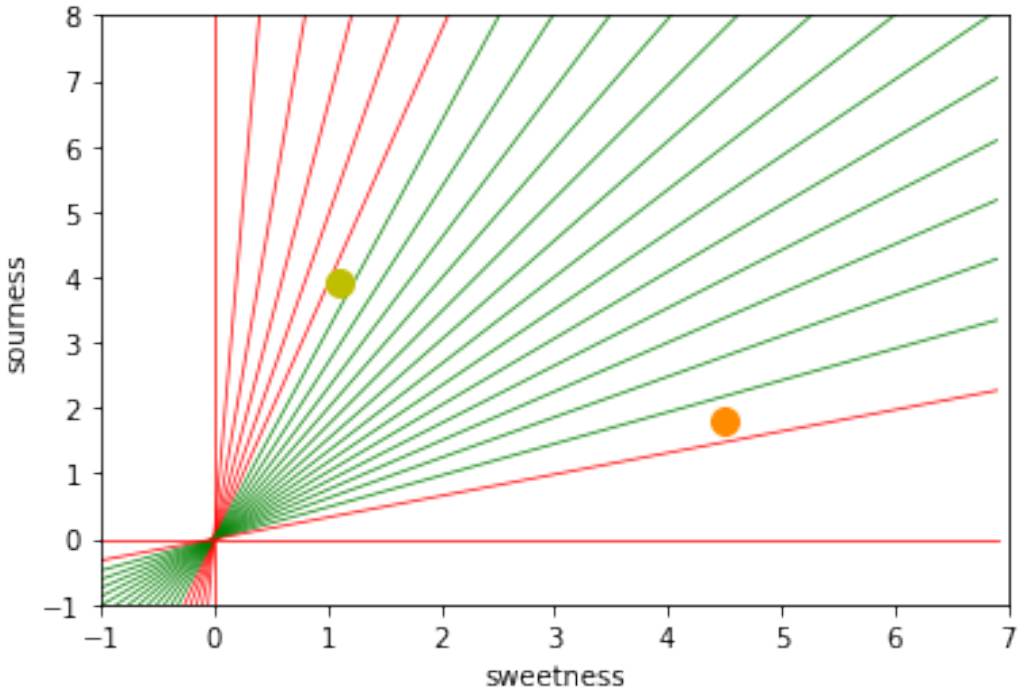
Wir haben die grüne Linie nicht durch mathematische Methoden oder Formeln bestimmt, sondern willkürlich durch das Anschauen des Diagramms. Wir hätten genausogut andere Geraden nehmen können.

Das nachfolgende Python Programm berechnet und zeichnet einige Linien. Alle gehen durch den Ursprung, also durch (0,0). Die rot dargestellten sind absolut unbrauchbar, um die beiden Früchte zu trennen, weil in diesen Fällen die Zitrone und die Orange auf derselben Seite der Geraden liegen.

Aber es ist sofort zu sehen, dass sogar die grünen Linien nicht besonders brauchbar sein könnten, wenn wir mehr als nur diese zwei Früchte hätten. Einige Zitronen könnten durchaus süsser sein und einige Orangen ganz schön sauer.

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

def create_distance_function(a, b, c):
    """ 0 = ax + by + c """
    def distance(x, y):
        """
        returns tuple (d, pos)
        d is the distance
        If pos == -1 point is below the line,
        0 on the line and +1 if above the line
        """
        nom = a * x + b * y + c
```

```
        if nom == 0:
            pos = 0
        elif (nom<0 and b<0) or (nom>0 and b>0):
            pos = -1
        else:
            pos = 1
        return (np.absolute(nom) / np.sqrt( a ** 2 + b ** 2), pos)
    return distance

orange = (4.5, 1.8)
lemon = (1.1, 3.9)
fruits_coords = [orange, lemon]

fig, ax = plt.subplots()
ax.set_xlabel("sweetness")
ax.set_ylabel("sourness")
x_min, x_max = -1, 7
y_min, y_max = -1, 8
ax.set_xlim([x_min, x_max])
ax.set_ylim([y_min, y_max])
X = np.arange(x_min, x_max, 0.1)

step = 0.05
for x in np.arange(0, 1+step, step):
    slope = np.tan(np.arccos(x))
    dist4line1 = create_distance_function(slope, -1, 0)
    Y = slope * X
    results = []
    for point in fruits_coords:
        results.append(dist4line1(*point))
    if (results[0][1] != results[1][1]):
        ax.plot(X, Y, "g-", linewidth=0.8, alpha=0.9)
    else:
        ax.plot(X, Y, "r-", linewidth=0.8, alpha=0.9)

size = 10
for (index, (x, y)) in enumerate(fruits_coords):
    if index== 0:
        ax.plot(x, y, "o",
                color="darkorange",
                markersize=size)
    else:
        ax.plot(x, y, "oy",
                markersize=size)


plt.show()
```



Im Prinzip haben wir hier eine Klassifizierung durchgeführt durch unsere Trennlinien. Auch wenn kaum einer das so bezeichnen würde.

Man kann sich leicht vorstellen, man hätte mehr Orangen und Zitronen mit variierenden Süsse- und Sauerkeitswerten. Dies hiesse, wir hätten eine Zitronen-Klasse (`class2`) und einen Orangen-Klasse (`class1`), so wie im folgenden Diagramm dargestellt.

Lassen Sie uns jetzt einmal Zitronen und Orangen mit einem Python Programm “erzeugen”. Wir werden diese beiden Klassen herstellen, indem wir zufällig Punkte erzeugen um einen definierten Mittelpunkt, innerhalb eines bestimmten Radius.

Der folgende Python-Kode wir diese beiden Klassen generieren.

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

def points_within_circle(radius,
                         center=(0, 0),
                         number_of_points=100):
    center_x, center_y = center
    r = radius * np.sqrt(np.random.random((number_of_points,)))
    theta = np.random.random((number_of_points,)) * 2 * np.pi
    x = center_x + r * np.cos(theta)
    y = center_y + r * np.sin(theta)
    return x, y

X = np.arange(0, 8)
fig, ax = plt.subplots()
oranges_x, oranges_y = points_within_circle(1.6, (5, 2), 100)
lemons_x, lemons_y = points_within_circle(1.9, (2, 5), 100)

ax.scatter(oranges_x,
           oranges_y,
```

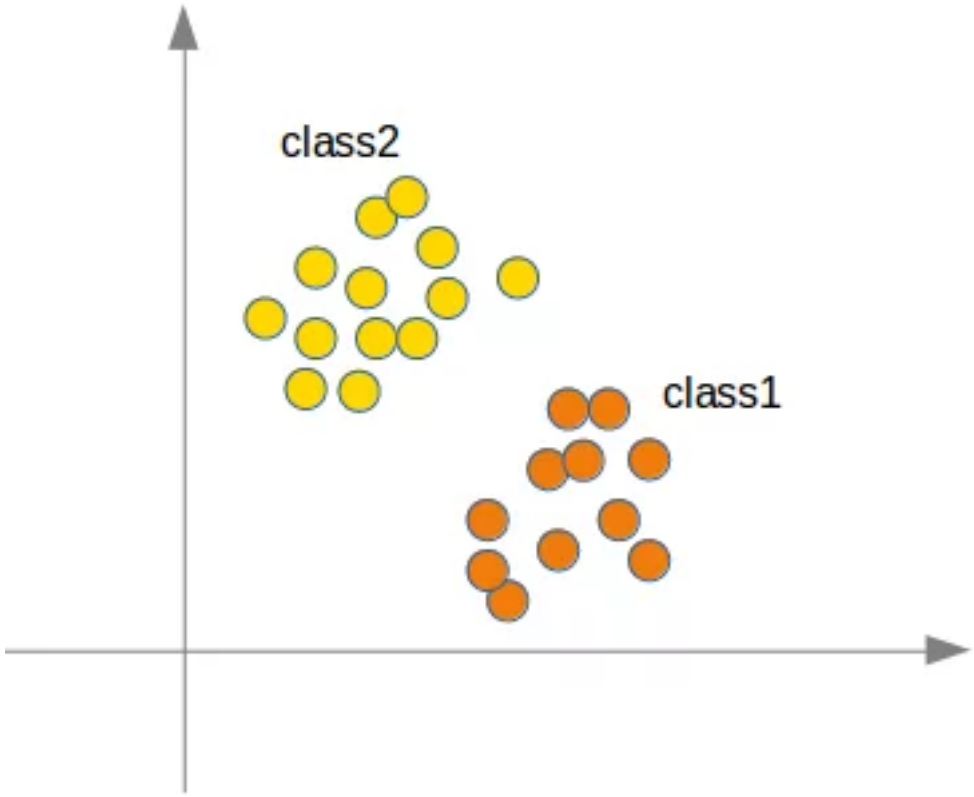
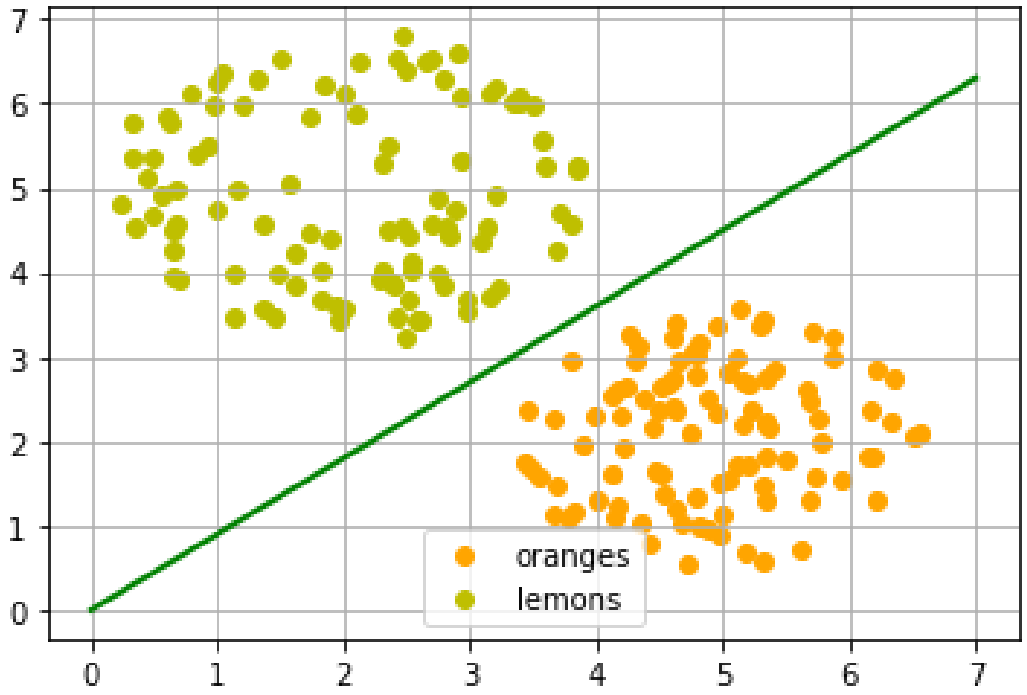


Abbildung 11.2: Two clusters of 2-dimensional points

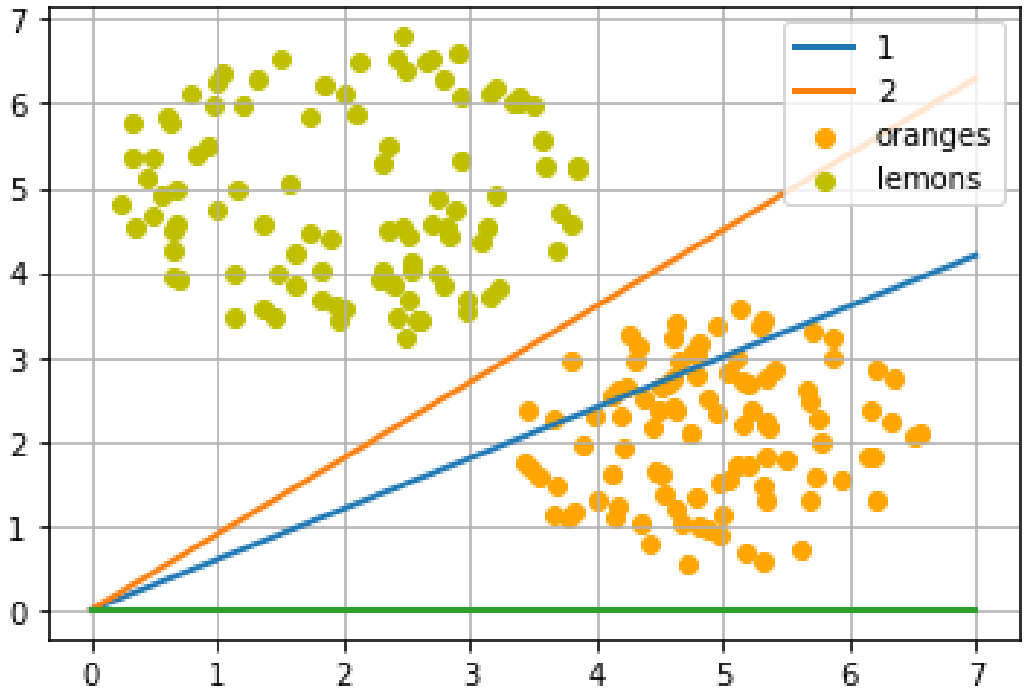
```
            c="orange",
            label="oranges")
ax.scatter(lemons_x,
            lemons_y,
            c="y",
            label="lemons")

ax.plot(X, 0.9 * X, "g-", linewidth=2)

ax.legend()
ax.grid()
plt.show()
```



Die Trennlinie wurde wieder willkürlich nach dem optischen Eindruck festgelegt. Es stellt sich die Frage: Wie machen wir das sytematisch? Weiter schauen wir nur auf Geraden durch den Ursprung, die allein durch ihre Steigung definiert sind. Das folgende Python Programm berechnet eine Trennlinie, indem es durch alle Früchte geht, und die Steigung der Trennlinie, die wir kalkulieren wollen, dynamisch anpasst. Wenn ein Punkt fälschlich oberhalb der Geraden liegt, werden wir die Steigung um den Wert von `learning_rate` erhöhen. Im umgekehrten Fall wird die Steigung um den Wert `learning_rate` vermindert.

```
import numpy as np
import matplotlib.pyplot as plt
from itertools import repeat
from random import shuffle



X = np.arange(0, 8)
fig, ax = plt.subplots()

ax.scatter(oranges_x,
           oranges_y,
           c="orange",
           label="oranges")
ax.scatter(lemons_x,
           lemons_y,
           c="y",
           label="lemons")

fruits = list(zip(oranges_x,
                  oranges_y,
                  repeat(0, len(oranges_x))))
fruits += list(zip(lemons_x,
                   lemons_y,
```

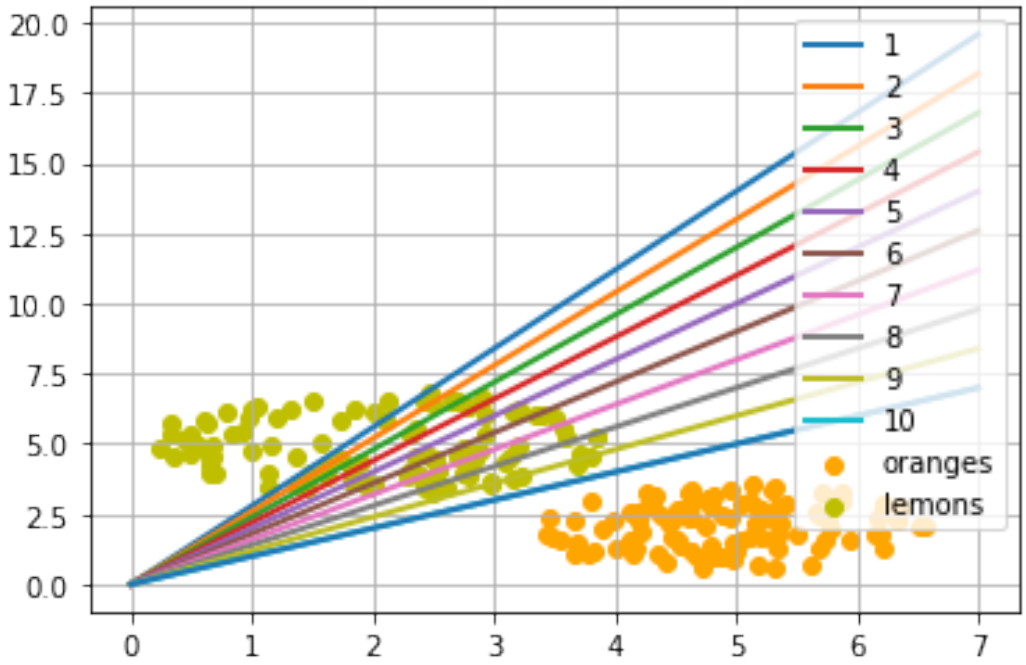
```
                    repeat(1, len(oranges_x))))
shuffle(fruits)

def adjust(learning_rate=0.3, slope=0.3):
    line = None
    counter = 0
    for x, y, label in fruits:
        res = slope * x - y
        #print(label, res)
        if label == 0 and res < 0:
            # point is above line but should be below
            # => increment slope
            slope += learning_rate
            counter += 1
            ax.plot(X, slope * X,
                    linewidth=2, label=str(counter))

        elif label == 1 and res > 0:
            # point is below line but should be above
            # => decrement slope
            #print(res, label)
            slope -= learning_rate
            counter += 1
            ax.plot(X, slope * X,
                    linewidth=2, label=str(counter))
    return slope


adjust()
slope = ax.plot(X,slope * X, linewidth=2)
ax.legend()
ax.grid()
plt.show()

print(slope)
```




```
[<matplotlib.lines.Line2D object at 0x7fb572cb8640>]
```

Lassen Sie uns mit einer anderen Steigung von der Zitronen-Seite aus beginnen.

```
X = np.arange(0, 8)
fig, ax = plt.subplots()
ax.scatter(oranges_x,
           oranges_y,
           c="orange",
           label="oranges")
ax.scatter(lemons_x,
           lemons_y,
           c="y",
           label="lemons")

slope = adjust(learning_rate=0.2, slope=3)
ax.plot(X,
        slope * X,
        linewidth=2)
ax.legend()
ax.grid()
plt.show()

print(slope)
```




```
0.9999999999999996
```

Ein einfaches Neuronales Netzwerk

Wir konnten die zwei Klassen mit einer geraden Linie trennen. Vielleicht fragen Sie sich, was das mit neuronalen Netzwerken zu tun hat. Damit beschäftigen wir uns im Folgenden.

Wir werden nun ein neuronales Netz definieren, um den vorher erzeugten Datensatz zu klassifizieren. Unser Netzwerk wird dabei nur aus einem einzigen Neuron bestehen, einem Neuron mit zwei Eingängen, einem für “Sauerkeit” und einem für “Süsse”.

Die zwei Eingänge -in dem untenstehenden Programm als `in_data` bezeichnet - müssen mit Gewichten (engl. weights) ausgestattet werden.

Um unser Problem zu lösen, definieren wir eine Perceptron Klasse. Eine Instanz dieser Klasse ist dann ein Perceptron (oder Neuron). Es wird mit input_lenth initialisiert werden, also der Anzahl von Eingangswerten, und den Gewichten der Eingänge als Liste, Tupel oder Array. Wenn keine Werte für die Gewichte angegeben wurden, oder wenn dieser Parameter auf None gesetzt ist, nehmen wir den Wert 1/ input_length zum Initialisieren an.

Im folgenden Beispiel haben wir die Werte von -.45 und .5 als Werte für die Gewichte verwendet. Es ist natürlich nicht der normale Weg, dies so durchzuführen. Ein neuronales Netzwerk berechnet die Gewichte während der Trainigsphase selbst, wie wir später sehen werden.

```
import numpy as np

class Perceptron:

    def __init__(self, weights):
        """
        'weights' can be a numpy array, list or a tuple with the
        actual values of the weights. The number of input values
```

```
        is indirectly defined by the length of 'weights'
        """
        self.weights = np.array(weights)

    def __call__(self, in_data):
        weighted_input = self.weights * in_data
        weighted_sum = weighted_input.sum()
        return weighted_sum

p = Perceptron(weights=[-0.45, 0.5])

for point in zip(oranges_x[:10], oranges_y[:10]):
    res = p(point)
    print(res, end=", ")

for point in zip(lemons_x[:10], lemons_y[:10]):
    res = p(point)
    print(res, end=", ")
```

```
-2.1589289019956572, -0.7188549528209387, -0.9544616671525588,
↪ -1.3118918749640691, -0.8604992239857685, -0.9087108494944371,
↪ -0.6205558861197846, -0.3772085775128349, -0.9237841120028571,
↪ -1.0851191971730245, 0.8757743511843614, 1.6528023152447018,
↪  1.9665897525289386, 2.2046388879193586, 1.0172383956710098,
↪  1.8123567421615892, 2.6554758385381914, 2.670275489336172,
↪  0.9188208010194461, 1.9205142972218014,
```

Wir können sehen, dass wir einen negativen Wert bekommen, wenn wir eine Orange eingeben und einen positiven Wert für eine Zitrone. Damit können wir nun die Genauigkeit (engl. accuracy) unseres Netzwerkes für diesen Datensatz berechnen:

```
from collections import Counter
evaluation = Counter()
for point in zip(oranges_x, oranges_y):
    res = p(point)
    if res < 0:
        evaluation['corrects'] += 1
    else:
        evaluation['wrongs'] += 1


for point in zip(lemons_x, lemons_y):
    res = p(point)
    if res >= 0:
        evaluation['corrects'] += 1
    else:
        evaluation['wrongs'] += 1

print(evaluation)
```

```
Counter({'corrects': 200})
```

Wie funktioniert nun dieser Vorgang? Wir summieren die Eingangswerte multipliziert mit den jeweiligen Gewichten und bekommen negative und positive Werte. Schauen wir uns an, was passiert, wenn die Summe der mit den Gewichten multiplizierten Eingangswerten 0 ergibt.

$$w_1 \cdot x_1 + w_2 \cdot x_2 = 0$$

Wir können dies umformulieren zu:

$$x_2 = -\frac{w_1}{w_2} \cdot x_1$$

Wir können das nun mit einer generellen Geradengleichung vergleichen

$$y = m \cdot x + c$$

wo

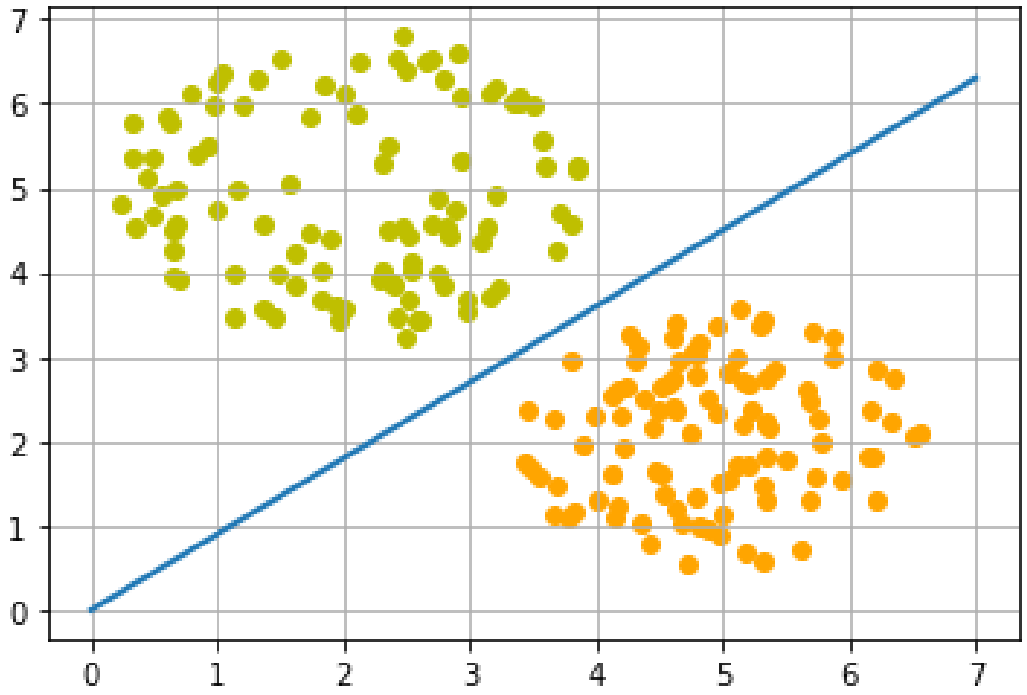
- m die Steigung der Geraden ist.
- c ist der Schnittpunkt der Geraden mit der Y-Achse.
- x ist die unabhängige Varable der Funktion.

Wir können leicht sehen, dass unsere Gleichung von oben der Definition einer Geraden entspricht und die Steigung (oder der Gradient) m $-\frac{w_1}{w_2}$ ist und $c = 0$.

Diese gerade Linie, die die Orangen und Zitronen trennt, heisst **Entscheidungsgrenze** (engl. **decision boundary**).

Wir zeigen das jetzt mit einem Python Programm:

```
import time
import matplotlib.pyplot as plt
slope = 0.1

X = np.arange(0, 8)
fig, ax = plt.subplots()
ax.scatter(oranges_x,
           oranges_y,
           c="orange",
           label="oranges")
ax.scatter(lemons_x,
           lemons_y,
           c="y",
           label="lemons")

slope = 0.45 / 0.5
ax.plot(X, slope * X,  linewidth=2)


ax.grid()
plt.show()

print(slope)
```




```
0.9
```

Das Trainieren eines neuronalen Netzwerkes

Wie wir schon im vorhergehenden Abschnitt ausgeführt haben: Wir haben unser Netzwerk nicht trainiert. Wir haben die Gewichte auf Werte gesetzt, von denen wir wussten, dass sie eine Trennlinie erzeugen würden. Jetzt werden wir zeigen, wie man stattdessen unser einfaches neuronales Netz trainieren kann.

Bevor wir damit anfangen, werden wir unseren Datensatz in Trainings- und Testdaten aufteilen im folgenden Pythopn Programm. Indem wir random_state auf den Wert 42 setzen, erzeugen wir für jeden Durchlauf denselben Output, wir starten also den (Pseudo-)Zufallszahlengenerator jeweils in identischer Weise, was sehr beim Debugging helfen kann.

```
from sklearn.model_selection import train_test_split

oranges = list(zip(oranges_x, oranges_y))
lemons = list(zip(lemons_x, lemons_y))

data = oranges + lemons
labels = [0] * len(oranges) + [1] * len(lemons)

train_data, test_data, train_labels, test_labels = train_test_split(
    data, labels, test_size=0.2, random_state=42, stratify=labels
)
print(train_data[:10], train_labels[:10])
```

Da wir mit zwei willkürlichen Gewichten starten, können wir natürlich kein korrektes Resultat erwarten. Für einige Punkte (Früchte) kann das richtige Ergebnis zurückgegeben werden, also zum Beispiel 1 für eine Orange und 0 für einen Zitrone. Wenn das Ergebnis falsch ist, müssen wir unsere Gewichte korrigieren. Zunächst müssen wir dazu den Fehler des

Netzwerks berechnen. Dieser Fehler ist die Differenz zwischen dem Ziel- oder Erwartungswert (`target_result`) und dem berechneten Wert (`calculated_result`). Mit diesem Fehler müssen wir dann unsere Gewichte mit einem hinzuzufügenden Wert berichtigen, z.B. $w_1 = w_1 + \Delta w_1$ and $w_2 = w_2 + \Delta w_2$.

Ist der Fehler `e` 0 , müssen wir nichts tun, der Zielwert ist gleich dem berehneten Wert. Für diese Eingangswerte ist das Netzwerk perfekt. Ist der Fehler nicht 0, müssen wir die Gewichte ändern. Wir addieren kleine Werte dazu. Diese können positiv oder negativ sein. Das Ausmass der notwendigen Änderung der Gewichte hängt vom Fehler und von den Eingangswerten ab. Lassen Sie uns annehmen, $x_1 = 0$ and $x_2 > 0$. In diesem Fall hängt das Ergebnis nur vom Eingangswert x_2 ab. Das bedeutet aber, dass wir nur w_2 ändern müssen , um den Fehler zu mimimieren. Ist der Fehler negativ, müssen wir einen kleinen negativen Wert dazu addieren, im gegensätzlichen Fall umgekehrt. Hieraus können wir ersehen, dass was immer auch die Eingangswerte sind, wir können sie mit dem Fehler multiplizieren und erhalten Werte, die wir zu den Gewichten hinzufügen können. Aber eine Sache fehlt noch, das Netzwerk würde in viel zu grossen Schritten lernen und könnte dann überkorrigieren. Wir haben ja vielen Datensätze und jedes Beispiel sollte die Gewichte nur ein wenig ändern, damit insgesamt das Netzwerk sich in die richtige Richtung verändert und nicht über das Ziel hinausschiesst, wenn ein einzelnes Eingabebeispiel zu einem grossen Fehler führt. Wir multiplizieren den Fehler deshalb mit einem kleinen Wert, der Lernrate (`self.learning_rate`). Diese Lernrate bestimmt, wie schnell sich die Gewichte adaptieren. Kleine Werte machen den Lernprozess langsamer, grosse Werte können zu suboptimalen Gewichten und Übersteuern des Netzwerkes führen. Wir werden in unserem Kapitel über Fehlerrückführung (engl. backpropagation) genauer darauf zurückkommen.

Wir sind nun soweit, dass wir den Kode schreiben können, um die Gewichte anzupassen, also unser Netzwerk zu trainieren. Dazu dient die Methode “adjust” in unserer Perceptron Klasse. Diese Methode soll den Fehler des Netzwerkes korrigieren.

```
import numpy as np
from collections import Counter

class Perceptron:

    def __init__(self,
                 weights,
                 learning_rate=0.1):
        """
        'weights' can be a numpy array, list or a tuple with the
        actual values of the weights. The number of input values
        is indirectly defined by the length of 'weights'
        """
        self.weights = np.array(weights)
        self.learning_rate = learning_rate

    @staticmethod
    def unit_step_function(x):
        if  x < 0:
            return 0
        else:
            return 1

    def __call__(self, in_data):
        weighted_input = self.weights * in_data
        weighted_sum = weighted_input.sum()
```

```
        #print(in_data, weighted_input, weighted_sum)
        return Perceptron.unit_step_function(weighted_sum)

    def adjust(self,
               target_result,
               calculated_result,
               in_data):
        if type(in_data) != np.ndarray:
            in_data = np.array(in_data)  #
        error = target_result - calculated_result
        if error != 0:
            correction = error * in_data * self.learning_rate
            self.weights += correction
            #print(target_result, calculated_result, error, in_data,
            ↪  correction, self.weights)

    def evaluate(self, data, labels):
        evaluation = Counter()
        for index in range(len(data)):
            label = int(round(p(data[index]),0))
            if label == labels[index]:
                evaluation["correct"] += 1
            else:
                evaluation["wrong"] += 1
        return evaluation


p = Perceptron(weights=[0.1, 0.1],
               learning_rate=0.3)

for index in range(len(train_data)):
    p.adjust(train_labels[index],
             p(train_data[index]),
             train_data[index])

evaluation = p.evaluate(train_data, train_labels)
print(evaluation.most_common())
evaluation = p.evaluate(test_data, test_labels)
print(evaluation.most_common())

print(p.weights)
```

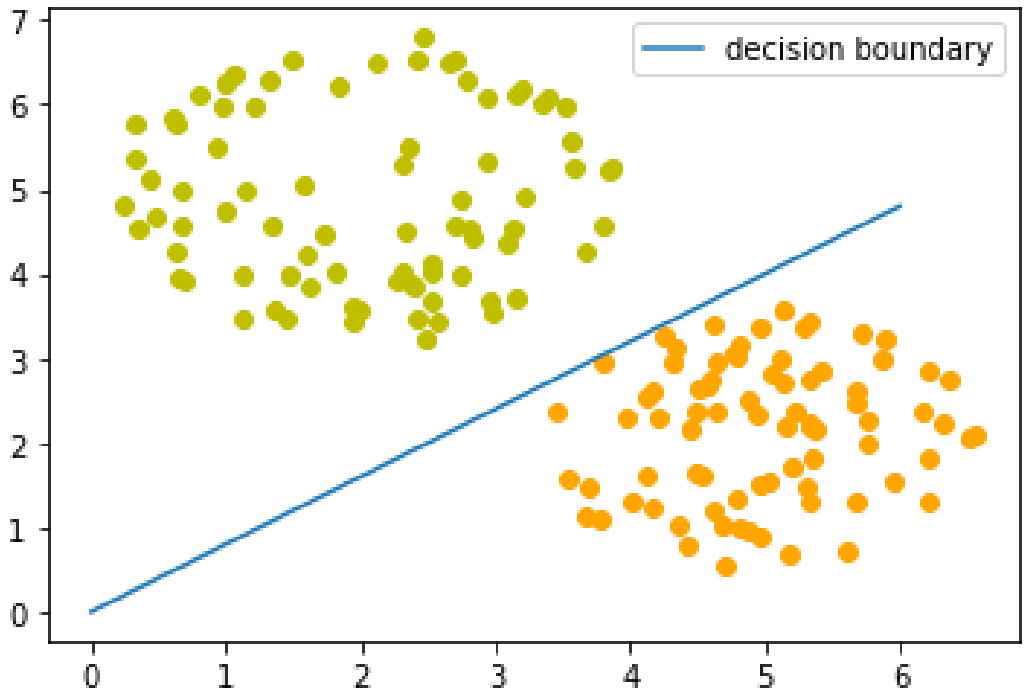
```
[('correct', 160)]
[('correct', 40)]
[-2.65007298  3.31182463]
```

In diesem bewusst einfach und klar trennbar konstruierten Datensatz erreicht das Perzeptron sowohl auf Trainings- als auch auf Testdaten eine sehr hohe Trefferquote. Das ist kein allgemeiner Leistungsnachweis für Perzeptrons, sondern eine Eigenschaft dieses Beispiels.

Wir visualisieren die Entscheidungsgrenze mit dem untenstehenden Programm:

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

X = np.arange(0, 7)
fig, ax = plt.subplots()
```

```
lemons = [train_data[i] for i in range(len(train_data)) if train_labels[i] ==
↪  1]
lemons_x, lemons_y = zip(*lemons)
oranges = [train_data[i] for i in range(len(train_data)) if train_labels[i] ==
↪  0]
oranges_x, oranges_y = zip(*oranges)

ax.scatter(oranges_x, oranges_y, c="orange")
ax.scatter(lemons_x, lemons_y, c="y")

w1 = p.weights[0]
w2 = p.weights[1]
m = -w1 / w2
ax.plot(X, m * X, label="decision boundary")
ax.legend()
plt.show()
print(p.weights)
```



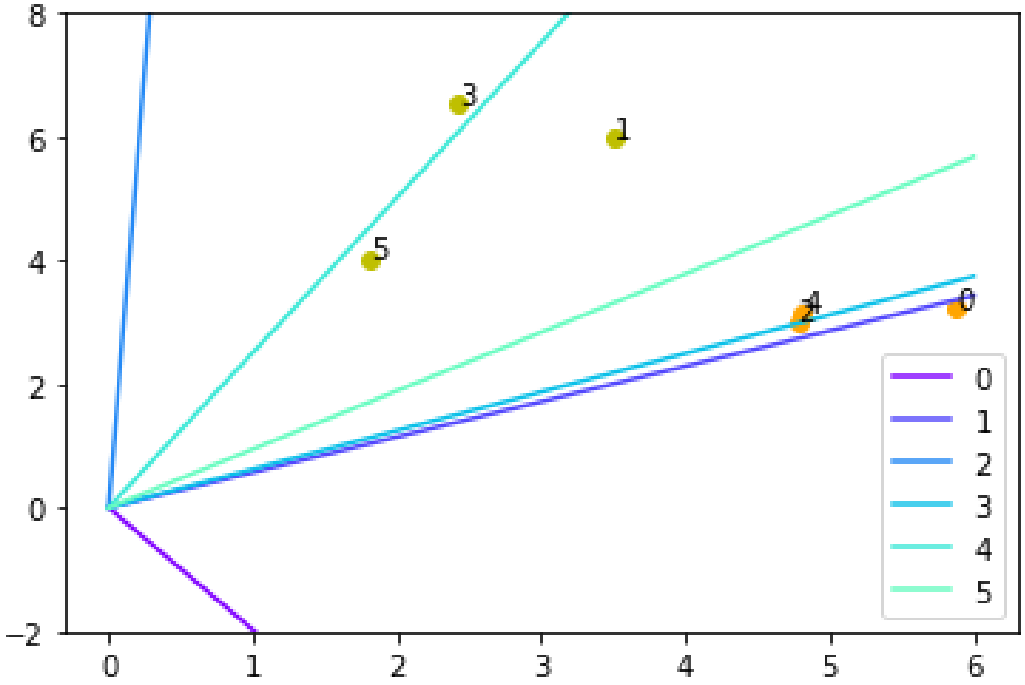

```
[-2.65007298  3.31182463]
```

Schauen wir uns diesen Algorithmus “in Bewegung” an.

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

p = Perceptron(weights=[0.1, 0.1],
               learning_rate=0.2)
number_of_colors = 12
colors = cm.rainbow(np.linspace(0, 1, number_of_colors))

fig, ax = plt.subplots()
```

```
ax.set_xticks(range(8))
ax.set_ylim([-2, 8])

counter = 0
for index in range(len(train_data)):
    old_weights = p.weights.copy()
    p.adjust(train_labels[index],
             p(train_data[index]),
             train_data[index])
    if not np.array_equal(old_weights, p.weights):
        color = "orange" if train_labels[index] == 0 else "y"
        ax.scatter(train_data[index][0],
                   train_data[index][1],
                   color=color)
        ax.annotate(str(counter),
                    (train_data[index][0], train_data[index][1]))
        m = -p.weights[0] / p.weights[1]
        print(index, m, p.weights, train_data[index])
        ax.plot(X, m * X, label=str(counter), color=colors[counter])
        counter += 1
ax.legend()
plt.show()
```

```
0 -1.9752467043187365 [-1.07572333 -0.54460201] (5.878616643765293,
↪  3.223010058442487)
1 0.5725383588053311 [-0.37352571  0.65240294] (3.510988105536155,
↪  5.985024750569256)
6 28.235854054892872 [-1.32949524  0.04708536] (4.779847668111197,
↪  3.0265879119267756)
8 0.6244339295305609 [-0.8445382   1.35248609] (2.4247851849606583,
↪  6.527003673497848)
10 2.509813160065015 [-1.80715234  0.72003461] (4.813070668850141,
↪  3.1622574073521115)
12 0.946328520188403 [-1.44461857  1.52655081] (1.8126688477427455,
↪  4.032581000894692)
```



Jeder der Punkte im obigen Diagramm verursacht eine Änderung der Gewichte. Wir sehen sie in der Reihenfolge ihres Erscheinens durchnummeriert und ebenfalls die jeweilige gerade Linie. So können wir sehen, wie das Netzwerk “lernt”, sich also adaptiert.

12 Einfaches Neuronales Netz

12.0.1 Einfaches neuronales Netz

Linear separierbare Datensätze

Wie wir im vorherigen Kapitel unseres Tutorials über maschinelles Lernen gezeigt haben, genügte ein neuronales Netzwerk, das nur aus einem Perzeptron besteht, um unsere Beispielklassen zu trennen. Natürlich haben wir diese Klassen sorgfältig entworfen, damit es funktioniert. Es gibt viele Klassen-Cluster, bei denen es nicht funktioniert. Wir werden uns einige andere Beispiele ansehen und Fälle diskutieren, in denen es nicht möglich ist, die Klassen zu trennen.

Unsere bisherigen Klassen waren linear separierbar. **lineare Separierbarkeit** ergibt Sinn in der euklidischen Geometrie. Zwei Mengen von Punkten (oder Klassen) in einer Ebene bezeichnet man als **linear separierbar** oder trennbar, wenn mindestens eine gerade Linie in der Ebene vorhanden ist, so dass sich alle Punkte der einen Klasse auf einer Seite der Linie und alle Punkte der anderen Klasse auf der anderen Seite befinden.

Etwas formaler:

Wenn zwei Datencluster (Klassen) durch eine Trennlinie in Form einer linearen Gleichung getrennt werden können

$$\sum_{i=1}^{n} x_i \cdot w_i = 0$$

nennt man sie **linear separierbar**.

Andernfalls, falls eine solche Trennlinie nicht existiert, bezeichnet man die beiden Klassen als nicht linear separierbar. In diesem Fall können wir kein einfaches neuronales Netzwerk verwenden.

Perzeptron für die AND-Funktion

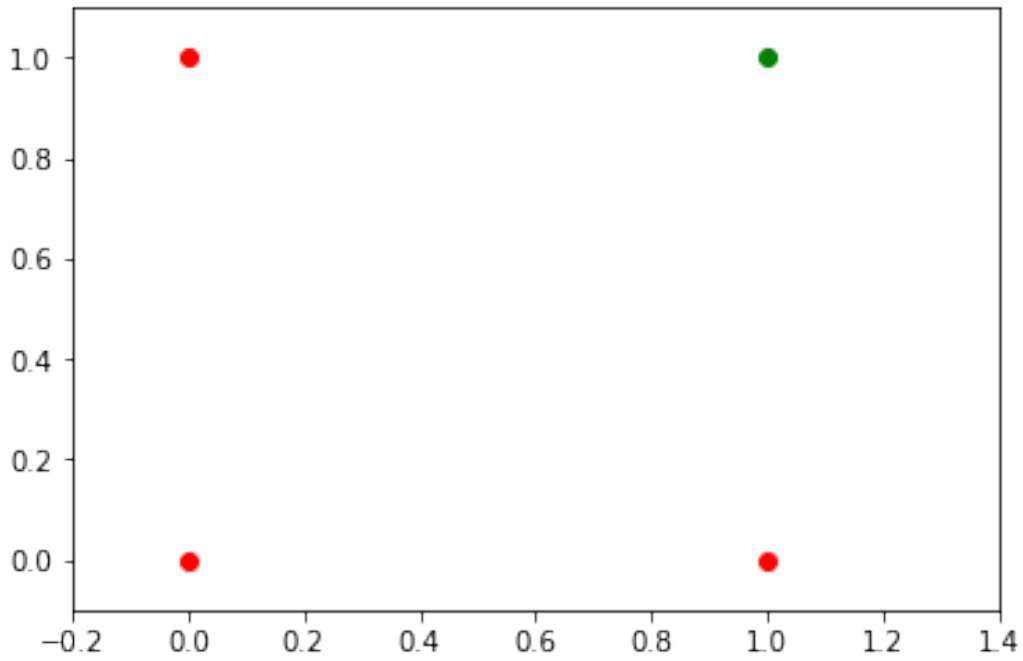
In unserem nächsten Beispiel programmieren wir ein einfaches Neuronales Netzwerk, welches die logische AND-Funktion implementiert. Sie hat zwei Eingaben und ist wie folgt definiert:

Eingabe1	Eingabe2	Ausgabe
0	0	0
0	1	0
1	0	0
1	1	1

Wir haben im vorherigen Kapitel gelernt, dass ein neuronales Netzwerk mit einem Perzeptron und zwei Eingan als Trennlinie interpretiert werden kann, d. h. eine Gerade, die zwei Klassen teilt. Die beiden Klassen, die wir in unserem Beispiel klassifizieren möchten, sehen so aus:

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

fig, ax = plt.subplots()
xmin, xmax = -0.2, 1.4
X = np.arange(xmin, xmax, 0.1)
ax.scatter(0, 0, color="r")
ax.scatter(0, 1, color="r")
ax.scatter(1, 0, color="r")
ax.scatter(1, 1, color="g")
ax.set_xlim([xmin, xmax])
ax.set_ylim([-0.1, 1.1])
m = -1
#ax.plot(X, m * X + 1.2, label="decision boundary")
plt.plot()
```

```
[]
```

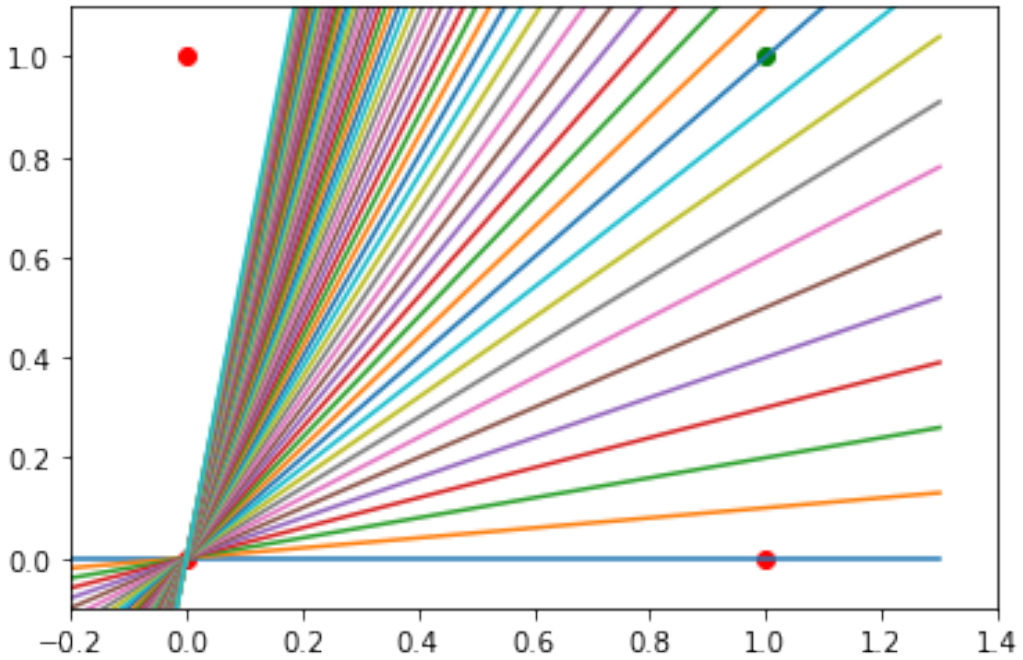


Wir haben auch herausgefunden, dass ein solches primitives neuronales Netzwerk nur in der Lage ist, Geraden durch den Ursprung zu beschreiben. Also Trennlinien der folgenden Art:

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

fig, ax = plt.subplots()
xmin, xmax = -0.2, 1.4
X = np.arange(xmin, xmax, 0.1)
ax.set_xlim([xmin, xmax])
ax.set_ylim([-0.1, 1.1])
m = -1
for m in np.arange(0, 6, 0.1):
```

```
    ax.plot(X, m * X )
ax.scatter(0, 0, color="r")
ax.scatter(0, 1, color="r")
ax.scatter(1, 0, color="r")
ax.scatter(1, 1, color="g")
plt.plot()
```

```
[]
```



Wir können sehen, dass sich keine dieser Geraden als Trennlinie verwendet lässt. Keine Gerade durch den Ursprung ist dazu geeignet.

Wir brauchen eine Gerade

$$y = m \cdot x + c$$

wobei der Achsenabschnitt `c` verschieden von 0 ist.

Die Gerade

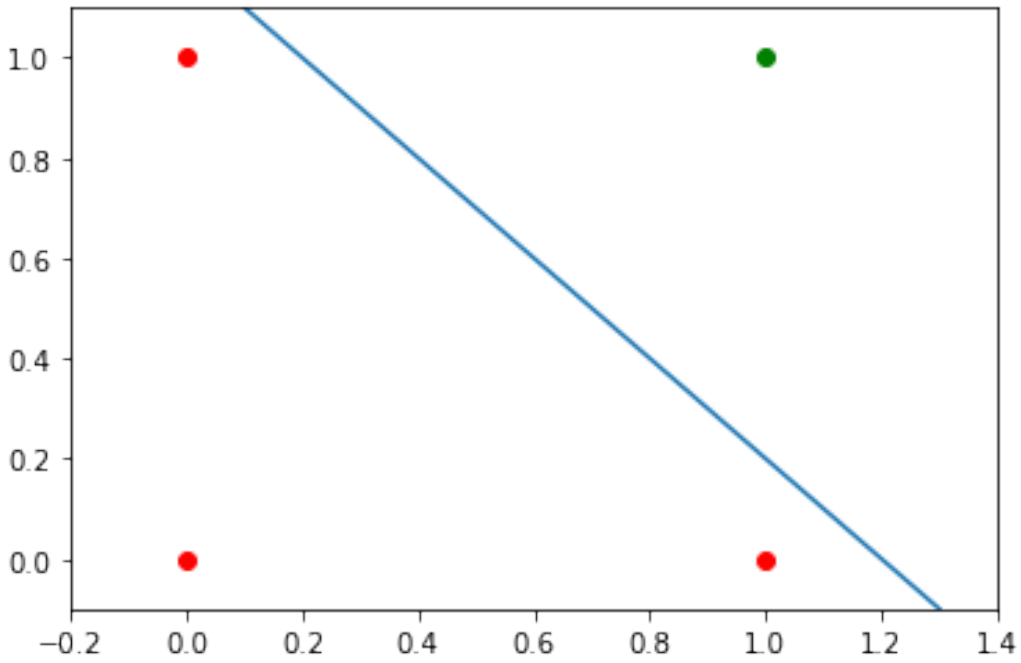
$$y = -x + 1.2$$

könnte beispielsweise als Trennlinie für unser Problem verwendet werden:

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

fig, ax = plt.subplots()
xmin, xmax = -0.2, 1.4
X = np.arange(xmin, xmax, 0.1)
ax.scatter(0, 0, color="r")
```

```
ax.scatter(0, 1, color="r")
ax.scatter(1, 0, color="r")
ax.scatter(1, 1, color="g")
ax.set_xlim([xmin, xmax])
ax.set_ylim([-0.1, 1.1])
m, c = -1, 1.2
ax.plot(X, m * X + c )
plt.plot()
```

```
[]
```



Es stellt sich die Frage, ob wir eine Lösung mit geringfügigen Modifikationen unseres Netzwerkmodells finden können? Oder mit anderen Worten: Können wir ein Perzeptron erstellen, welches in der Lage ist, beliebige Trennlinien zu finden?

Die Lösung besteht in der Zugabe eines Bias-Knotens.

Perzeptron mit Bias Ein Perceptron mit zwei Eingangswerten und einem Bias-Wert entspricht einer allgemeinen Geraden. Mit Hilfe des Bias-Werts `b` können wir das Perzeptron so trainieren, dass es allgemeine Geraden (Trennlinien) mit von 0 verschiedenem Achsenabschnitt `c` beschreiben kann.

Während sich die Eingangswerte ändern können, bleibt der Bias-Wert immer konstant. Nur das Gewicht des Bias-Knotens kann angepasst werden.

Nun enthält die lineare Gleichung für ein Perzeptron einen Bias:

$$\sum_{i=1}^{n} w_i \cdot x_i + w_{n+1} \cdot b = 0$$

In unserem Fall schaut es wie folgt aus:

$$w_1 \cdot x_1 + w_2 \cdot x_2 + w_3 \cdot b = 0$$

dies ist äquivalent mit

$$x_2 = -\frac{w_1}{w_2} \cdot x_1 - \frac{w_3}{w_2} \cdot b$$

Dies bedeutet:

$$m = -\frac{w_1}{w_2}$$

und

$$c = -\frac{w_3}{w_2} \cdot b$$

```
%%capture
%%writefile perceptrons.py

import numpy as np
from collections import Counter

class Perceptron:

    def __init__(self,
                 weights,
                 bias=1,
                 learning_rate=0.3):
        """
        'weights' kann ein Numpy-Array, eine Liste oder ein
        Tupel mit den Werten der Gewichte. Die Anzahl der Eingabewerte
         ist indirekt durch die Länge von 'weights' definiert
        """
        self.weights = np.array(weights)
        self.bias = bias
        self.learning_rate = learning_rate

    @staticmethod
    def unit_step_function(x):
        if  x <= 0:
            return 0
        else:
            return 1

    def __call__(self, in_data):
        in_data = np.concatenate( (in_data, [self.bias]) )
        result = self.weights @ in_data
        return Perceptron.unit_step_function(result)

    def adjust(self,
               target_result,
               in_data):
        if type(in_data) != np.ndarray:
            in_data = np.array(in_data)  #
```

```
        calculated_result = self(in_data)
        error = target_result - calculated_result
        if error != 0:
            in_data = np.concatenate( (in_data, [self.bias]) )
            correction = error * in_data * self.learning_rate
            self.weights += correction

    def evaluate(self, data, labels):
        evaluation = Counter()
        for sample, label in zip(data, labels):
            result = self(sample) # predict
            if result == label:
                evaluation["correct"] += 1
            else:
                evaluation["wrong"] += 1
        return evaluation
```

Wir gehen davon aus, dass der obige Python-Code mit der Perceptron-Klasse im aktuellen Arbeitsverzeichnis unter dem Namen 'perceptrons.py' gespeichert wird.

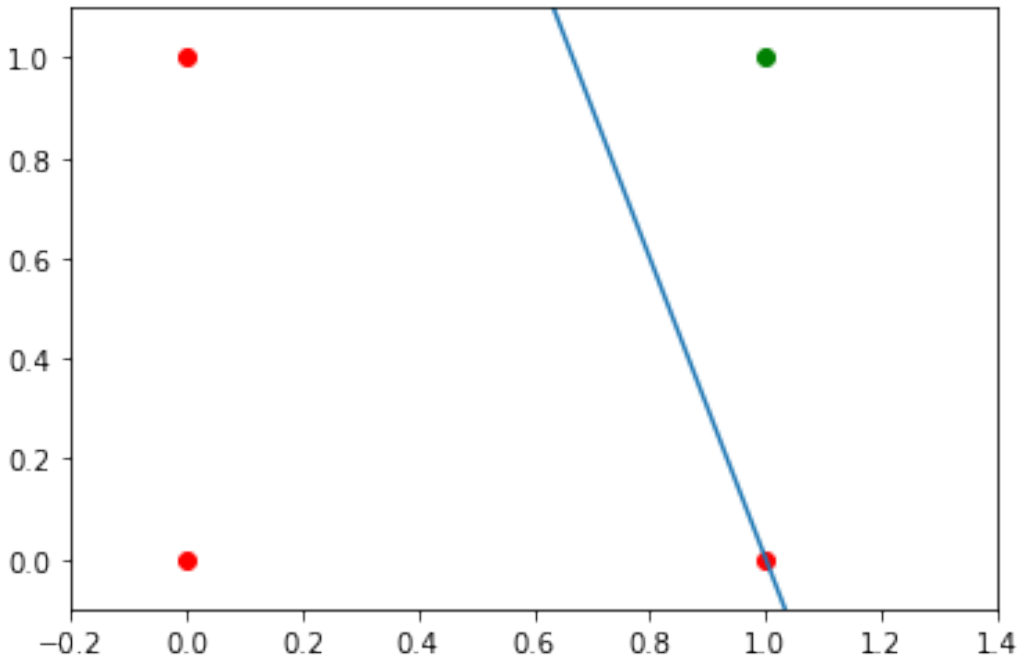
```
import numpy as np
from perceptrons import Perceptron

def labelled_samples(n):
    for _ in range(n):
        s = np.random.randint(0, 2, (2,))
        yield (s, 1) if s[0] == 1 and s[1] == 1 else (s, 0)

p = Perceptron(weights=[0.3, 0.3, 0.3],
               learning_rate=0.2)

for in_data, label in labelled_samples(30):
    p.adjust(label,
             in_data)

test_data, test_labels = list(zip(*labelled_samples(30)))

evaluation = p.evaluate(test_data, test_labels)
print(evaluation)
```

```
Counter({'correct': 30})
```

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

fig, ax = plt.subplots()
xmin, xmax = -0.2, 1.4
X = np.arange(xmin, xmax, 0.1)
ax.scatter(0, 0, color="r")
ax.scatter(0, 1, color="r")
ax.scatter(1, 0, color="r")
ax.scatter(1, 1, color="g")
ax.set_xlim([xmin, xmax])
ax.set_ylim([-0.1, 1.1])
m = -p.weights[0] / p.weights[1]
c = -p.weights[2] / p.weights[1]
```

```
print(m, c)
ax.plot(X, m * X + c )
plt.plot();
```

```
-3.0000000000000004 3.0000000000000013
```

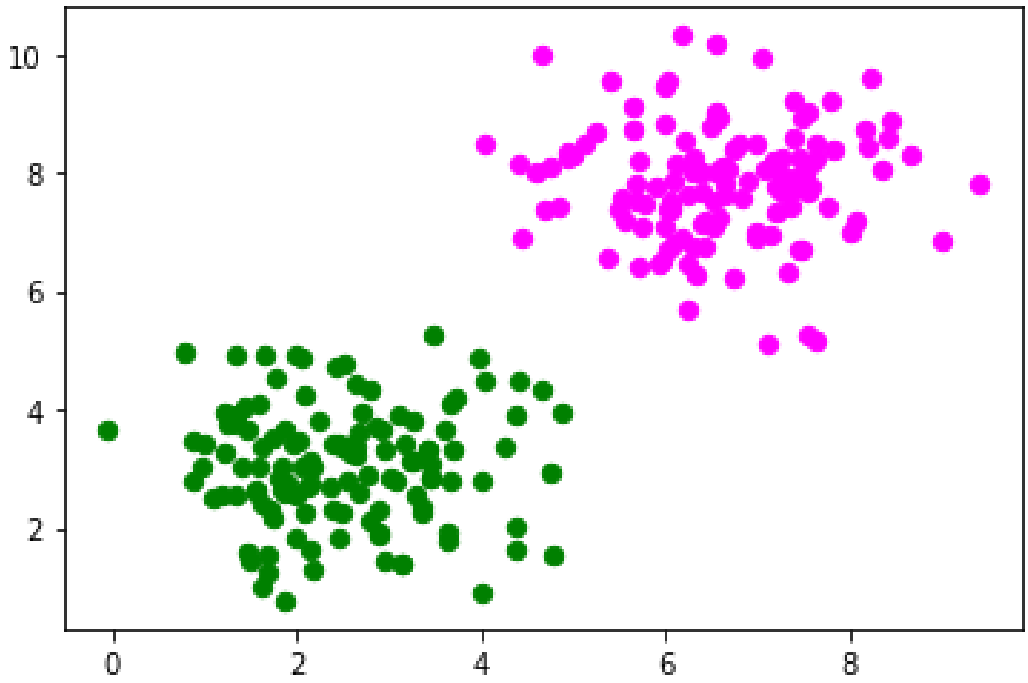


Wir erstellen ein anderes Beispiel mit linear trennbaren Datensätzen, für die wir auch einen Bias-Knoten zur Trennung benötigen. Wir werden die `make_blobs`-Funktion von `sklearn.datasets` dazu benutzen:

```
from sklearn.datasets import make_blobs

n_samples = 250
samples, labels = make_blobs(n_samples=n_samples,
                             centers=([2.5, 3], [6.7, 7.9]),
                             random_state=0)
```

Wir werden nun die vorher erzeugten Daten visualisieren:

```
import matplotlib.pyplot as plt

colours = ('green', 'magenta', 'blue', 'cyan', 'yellow', 'red')
fig, ax = plt.subplots()


for n_class in range(2):
    ax.scatter(samples[labels==n_class][:, 0], samples[labels==n_class][:, 1],
               c=colours[n_class], s=40, label=str(n_class))
```



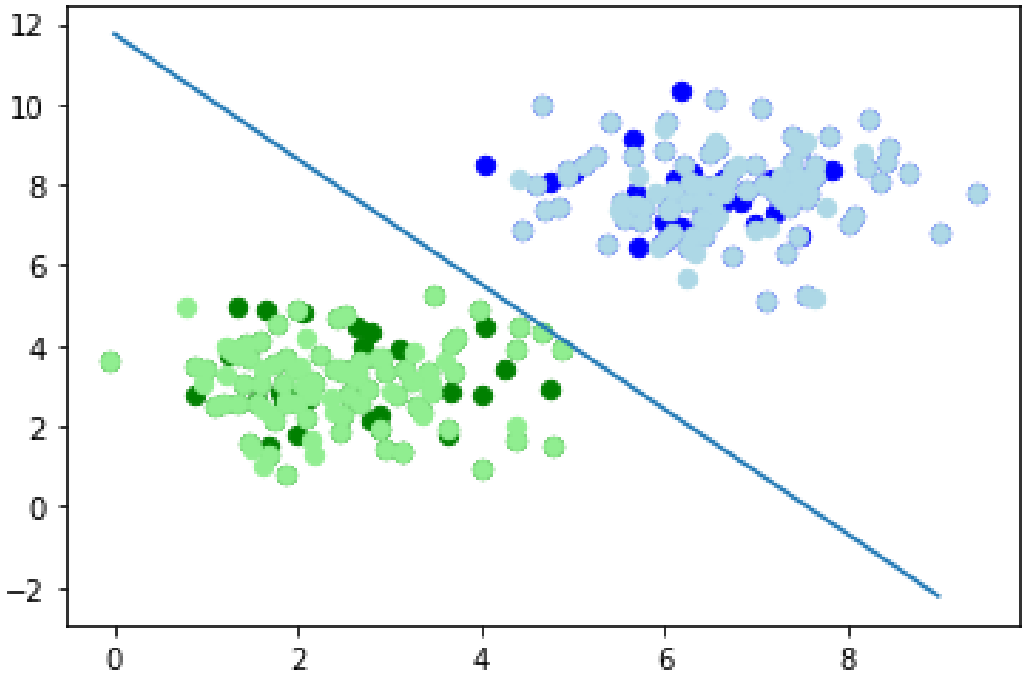

```
from sklearn.model_selection import train_test_split

learn_data, test_data, learn_labels, test_labels = train_test_split(
    samples, labels, test_size=0.2, random_state=42, stratify=labels
)

from perceptrons import Perceptron

p = Perceptron(weights=[0.3, 0.3, 0.3], learning_rate=0.8)

for sample, label in zip(learn_data, learn_labels):
    p.adjust(label, sample)

print("Training:", p.evaluate(learn_data, learn_labels))
print("Test:    ", p.evaluate(test_data, test_labels))
```

Nun wollen wir auch die Trennlinie darstellen:

```
import matplotlib.pyplot as plt


fig, ax = plt.subplots()

# plotting learn data
colours = ('green', 'blue')
for n_class in range(2):
    ax.scatter(learn_data[learn_labels==n_class][:, 0],
               learn_data[learn_labels==n_class][:, 1],
               c=colours[n_class], s=40, label=str(n_class))

# plotting test data
colours = ('lightgreen', 'lightblue')
for n_class in range(2):
```

```
    ax.scatter(test_data[test_labels==n_class][:, 0],
               test_data[test_labels==n_class][:, 1],
               c=colours[n_class], s=40, label=str(n_class))



X = np.arange(np.max(samples[:,0]))
m = -p.weights[0] / p.weights[1]
c = -p.weights[2] / p.weights[1]
print(m, c)
ax.plot(X, m * X + c )
plt.plot()
plt.show()
```

```
-1.5513529034664024 11.736643489707035
```



Im folgenden Abschnitt werden wir das XOR-Problem für neuronale Netzwerke einführen. Es ist das einfachste Beispiel eines nicht linear trennbaren neuronalen Netzwerks. Es kann mit einer zusätzlichen Neuronenschicht gelöst werden, die als verborgene Schicht (hidden layer) bezeichnet wird.

Das XOR-Problem für neuronale Netze

Die XOR-Funktion wird durch folgende Tabelle definiert:

Eingabe1	Eingabe2	XOR-Ausgabe
0	0	0
0	1	1
1	0	1

Eingabe1	Eingabe2	XOR-Ausgabe
1	1	0

Dieses Problem kann nicht mit einem einfachen Netzwerk gelöst werden, wie wir in folgendem Diagramm sehen können:

Egal welche Gerade man wählt, es wird einem niemals gelingen, die blauen Punkte auf einer Seite und den orangefarbenen Punkten auf der anderen Seite zu haben. Dies ist in der folgenden Abbildung dargestellt. Die orangefarbenen Punkte befinden sich auf der orangefarbenen Linie. Dies bedeutet, dass dies keine Trennlinie sein kann. Wenn wir diese Zeile parallel bewegen - egal in welche Richtung, es gibt immer zwei orangefarbene und einen blauen Punkt auf einer Seite und nur einen blauen Punkt auf der anderen Seite. Wenn wir die orangefarbene Linie nicht parallel bewegen, gibt es auf beiden Seiten einen blauen und einen orangefarbenen Punkt, außer wenn die Linie einen orangefarbenen Punkt durchläuft. Es gibt also keine Möglichkeit, dass eine einzige gerade Linie diese Punkte trennen kann.

Um dieses Problem zu lösen, müssen wir eine neue Art von neuronalen Netzwerken einführen, ein Netzwerk mit sogenannten verborgenen Schichten (hidden layers). Eine verborgene Schicht ermöglicht es dem Netzwerk, die Eingabedaten neu zu organisieren oder neu anzuordnen.

Wir benötigen nur eine verborgene Schicht mit zwei Neuronen. Eine arbeitet wir ein AND-Gatter und die andere wie ein OR-Gatter. Die Ausgabe “feuert”, wenn das OR-Gatter feuert und das AND-Gatter nicht.

Wie bereits erwähnt, können wir keine Trennlinie finden, die die orangefarbenen Punkte von den blauen Punkten trennt. Sie können jedoch durch zwei Linien getrennt werden, z. B. L1 und L2 in dem folgenden Diagramm:

Um dieses Problem zu lösen benötigen wir ein Netzwerk der folgenden Art, d. h. eines mit einer verborgenen Schicht N1 und N2

Das Perzeptron N1 bestimmt eine Trennlinie, z. B. L1 und das Perzeptron N2 bestimmt die andere Trennlinie L2. N3 löst dann unser Problem:

Die Implementierung von diesem Netzwerk muss bis zum nächsten Kapitel unseres Turorials über maschinelles Lernen warten.

Aufgaben

Aufgabe 1 Erweitere die AND-Funktion, dass sie Fließkommazahlen zwischen 0 und 1 in der folgenden Art zurückliefert:

Input1

Input2

Output

x1 < 0.5

x2 < 0.5

0

x1 < 0.5

x2 >= 0.5

0

x1 >= 0.5

x2 < 0.5

0

x1 >= 0.5

x2 >= 0.5

1

Versuche ein neuronales Netzwerk mit nur einem Perzeptron zu trainieren. Warum funktioniert dies nicht?

Aufgabe 2 Ein Punkt gehört zu einer Klasse 0, falls $x_1 < 0.5$ ist und gehört zur Klasse 1, falls $x_1 >= 0.5$ ist. Trainiere ein Netzwerk mit einem Perzeptron, um beliebige Punkte zu trainieren. Was kann man über die Trennlinie sagen? Was über die Eingabewerte?

Lösungen zu den Aufgaben

Lösung zur ersten Aufgabe

```
from perceptrons import Perceptron

p = Perceptron(weights=[0.3, 0.3, 0.3],
               bias=1,
               learning_rate=0.2)

def labelled_samples(n):
    for _ in range(n):
        s = np.random.random((2,))
        yield (s, 1) if s[0] >= 0.5 and s[1] >= 0.5 else (s, 0)

for in_data, label in labelled_samples(100):
    p.adjust(label,
             in_data)

test_data, test_labels = list(zip(*labelled_samples(120)))

evaluation = p.evaluate(test_data, test_labels)
print(evaluation)
```

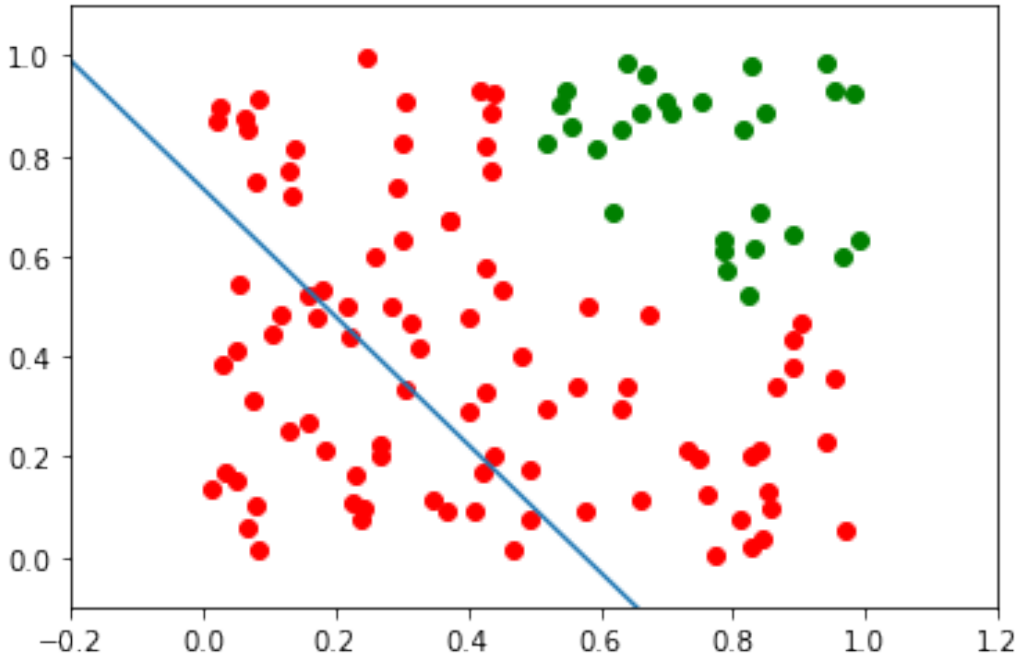
```
Counter({'wrong': 61, 'correct': 59})
```

Der einfachste Weg zu sehen, dass es nicht funktioniert kann, besteht darin, die Daten zu visualisieren:

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

ones = [test_data[i] for i in range(len(test_data)) if test_labels[i] == 1]
zeroes = [test_data[i] for i in range(len(test_data)) if test_labels[i] == 0]
```

```
fig, ax = plt.subplots()
xmin, xmax = -0.2, 1.2
X, Y = list(zip(*ones))
ax.scatter(X, Y, color="g")
X, Y = list(zip(*zeroes))
ax.scatter(X, Y, color="r")
ax.set_xlim([xmin, xmax])
ax.set_ylim([-0.1, 1.1])
c = -p.weights[2] / p.weights[1]
m = -p.weights[0] / p.weights[1]
X = np.arange(xmin, xmax, 0.1)
ax.plot(X, m * X + c, label="decision boundary")
```

```
[<matplotlib.lines.Line2D at 0x7fdfd17fd6d0>]
```



Wir können erkennen, dass sich die grünen und die roten Punkte nicht durch eine Gerade trennen lassen. We can see that the green points and the red points are not separable by one straight line.

Lösung zur zweiten Aufgabe

```
from perceptrons import Perceptron

import numpy as np
from collections import Counter

def labelled_samples(n):
    for _ in range(n):
        s = np.random.random((2,))
        yield (s, 0) if s[0] < 0.5 else (s, 1)
```

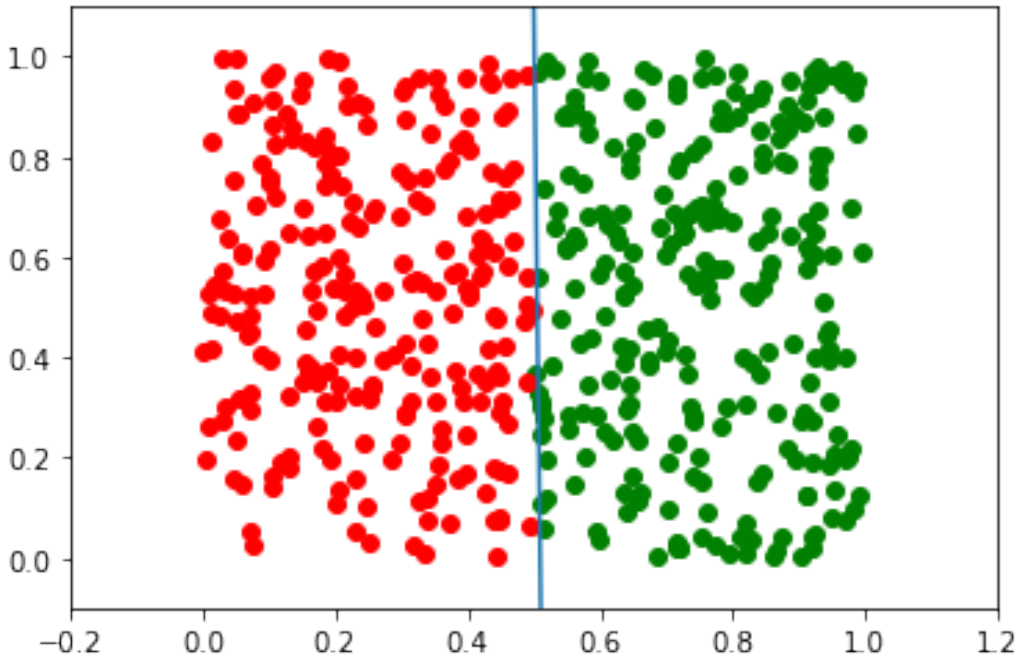
```
p = Perceptron(weights=[0.3, 0.3, 0.3],
               learning_rate=0.4)

for in_data, label in labelled_samples(300):
    p.adjust(label,
             in_data)

test_data, test_labels = list(zip(*labelled_samples(500)))

print(p.weights)
p.evaluate(test_data, test_labels)
```

```
[ 1.76735883  0.01530388 -0.9       ]
```

```
Counter({'correct': 499, 'wrong': 1})
```

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

ones = [test_data[i] for i in range(len(test_data)) if test_labels[i] == 1]
zeroes = [test_data[i] for i in range(len(test_data)) if test_labels[i] == 0]

fig, ax = plt.subplots()
xmin, xmax = -0.2, 1.2
X, Y = list(zip(*ones))
ax.scatter(X, Y, color="g")
X, Y = list(zip(*zeroes))
ax.scatter(X, Y, color="r")
ax.set_xlim([xmin, xmax])
ax.set_ylim([-0.1, 1.1])
c = -p.weights[2] / p.weights[1]
m = -p.weights[0] / p.weights[1]
X = np.arange(xmin, xmax, 0.1)
ax.plot(X, m * X + c, label="decision boundary")
```

```
[<matplotlib.lines.Line2D at 0x7fdfd17e29d0>]
```




```
p.weights, m
```

```
(array([ 2.03831116, -0.1785671 , -0.9       ]), 11.414819026425487)
```

Die Steigung `m` wird steiler und steiler in solchen Situation, d. h. strebt gegen Unendlich.

13 Neuronale Netze mit scikit-learn

13.0.1 Multi-Layer-Perceptron mit `MLPClassifier`

Scikit-learn enthält mit `MLPClassifier` ein vollständig verbundenes, vorwärtsgerichtetes neuronales Netz für Klassifikationsaufgaben. Ein Multi-Layer-Perceptron (MLP) besteht aus einer Eingabeschicht, einer oder mehreren verborgenen Schichten (*hidden layers*) und einer Ausgabeschicht.

Die Neuronen einer Schicht berechnen zunächst eine affine Transformation

[z^{(l+1)} = a^{(l)} W^{(l)} + b^{(l)}]

und wenden anschließend — außer an der Eingabeschicht — eine Aktivierungsfunktion an. Die Gewichte (W^{(l)}) und Bias-Vektoren (b^{(l)}) werden beim Training angepasst.

`MLPClassifier` eignet sich gut für kompakte klassische Machine-Learning-Beispiele. Für große Deep-Learning-Modelle, GPUs, Convolutional Networks oder Transformer verwendet man typischerweise spezialisierte Frameworks.

Ein wichtiger praktischer Punkt: MLPs reagieren empfindlich auf sehr unterschiedlich skalierte Eingabemerkmale. Deshalb kombinieren wir den Klassifikator in den Beispielen mit `StandardScaler` in einer `Pipeline`.

Dokumentation: MLPClassifier

13.0.2 Ein kleines XOR-Beispiel

Die XOR-Funktion ist nicht linear separierbar. Ein einzelnes lineares Perzeptron reicht deshalb nicht aus; eine verborgene Schicht kann die nötige nichtlineare Trennung lernen.

Wir verwenden nur vier Trainingspunkte. Für so einen winzigen Datensatz ist der Solver `lbfgs` gut geeignet. Die Skalierung wäre hier wegen der Werte 0 und 1 nicht zwingend nötig, zeigt aber bereits den später üblichen Aufbau als Pipeline.

```
import numpy as np

from sklearn.neural_network import MLPClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

X_xor = np.array([
    [0.0, 0.0],
    [0.0, 1.0],
    [1.0, 0.0],
    [1.0, 1.0],
])
y_xor = np.array([0, 1, 1, 0])

xor_model = Pipeline([
```

```
    ("scaler", StandardScaler()),
    ("mlp", MLPClassifier(
        hidden_layer_sizes=(4,),
        solver="lbfgs",
        max_iter=2000,
        random_state=1,
    )),
])

xor_model.fit(X_xor, y_xor)

print("Vorhersagen:", xor_model.predict(X_xor))
print("Wahrscheinlichkeiten für Klasse 1:")
print(np.round(xor_model.predict_proba(X_xor)[:, 1], 3))
```

```
Vorhersagen: [0 1 1 0]
Wahrscheinlichkeiten für Klasse 1:
[0. 1. 1. 0.]
```

Die Methode `predict` liefert die vorhergesagte Klasse. `predict_proba` gibt dagegen die vom Modell geschätzten Klassenwahrscheinlichkeiten zurück.

Bei einem binären Klassifikator besitzt `predict_proba(X)` zwei Spalten. Welche Klasse zu welcher Spalte gehört, steht in `classes_`. Man sollte daher nicht grundsätzlich annehmen, dass Spalte 0 immer eine bestimmte inhaltliche Klasse bedeutet.

```
mlp_xor = xor_model.named_steps["mlp"]

print("Klassen:", mlp_xor.classes_)
print("Anzahl Schichten:", mlp_xor.n_layers_)
print("Output-Aktivierung:", mlp_xor.out_activation_)
```

```
Klassen: [0 1]
Anzahl Schichten: 3
Output-Aktivierung: logistic
```

13.0.3 Gewichte und Bias-Werte

Nach dem Training stehen die Gewichte in `coefs_` und die Bias-Werte in `intercepts_`.

Für eine Verbindung von Schicht (l) nach Schicht (l+1) hat `mlp.coefs_[l]` die Form

```
(Anzahl Neuronen in Schicht l,
 Anzahl Neuronen in Schicht l+1)
```

Der zugehörige Bias-Vektor ist `mlp.intercepts_[l]`.

Wichtig ist die Trennung: Der Bias wird in scikit-learn **nicht** als zusätzliches Gewicht eines künstlichen Eingangs mit konstantem Wert 1 gespeichert, sondern separat in `intercepts_`.

```
for layer, (weights, bias) in enumerate(
    zip(mlp_xor.coefs_, mlp_xor.intercepts_)
):
    print(
```

```
        f"Übergang {layer} -> {layer + 1}: "
        f"W.shape={weights.shape}, b.shape={bias.shape}"
    )
```

```
Übergang 0 -> 1: W.shape=(2, 4), b.shape=(4,)
Übergang 1 -> 2: W.shape=(4, 1), b.shape=(1,)
```

```
print("Gewichte zum ersten Neuron der Hidden-Schicht:")
print(mlp_xor.coefs_[0][:, 0])

print("\nBias dieses Neurons:")
print(mlp_xor.intercepts_[0][0])
```

```
Gewichte zum ersten Neuron der Hidden-Schicht:
[-1.89420336 -1.44240859]

Bias dieses Neurons:
0.8289397457669401
```

Die konkreten Zahlen hängen von Initialisierung, Solver und Trainingsdaten ab. Bei einem neuronalen Netz sollte man einzelne Gewichte deshalb normalerweise nicht isoliert interpretieren. Interessanter sind die Gesamtleistung des Modells und sein Verhalten auf bisher ungesehenen Daten.

13.0.4 Ein realistischeres Beispiel mit Trainings- und Testdaten

Das nächste Beispiel verwendet `make_moons`. Die beiden Klassen sind nicht durch eine Gerade trennbar. Wir erzeugen deshalb eine nichtlineare Klassifikationsaufgabe und teilen die Daten anschließend in ein Trainings- und ein Testset.

Die Pipeline sorgt dafür, dass `StandardScaler` ausschließlich aus den Trainingsdaten lernt. Das ist wichtig, um *Data Leakage* zu vermeiden.

```
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split

X, y = make_moons(
    n_samples=600,
    noise=0.25,
    random_state=7,
)

X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.25,
    stratify=y,
    random_state=7,
)

print("Training:", X_train.shape)
print("Test:", X_test.shape)
```

```
Training: (450, 2)
```

```
Test: (150, 2)
```

```python
moon_model = Pipeline([
    ("scaler", StandardScaler()),
    ("mlp", MLPClassifier(
        hidden_layer_sizes=(50, 25),
        activation="relu",
        solver="adam",
        alpha=1e-4,
        learning_rate_init=0.005,
        max_iter=1500,
        early_stopping=True,
        validation_fraction=0.15,
        n_iter_no_change=30,
        random_state=1,
    )),
])

moon_model.fit(X_train, y_train)

print(f"Training-Accuracy: {moon_model.score(X_train, y_train):.3f}")
print(f"Test-Accuracy:     {moon_model.score(X_test, y_test):.3f}")
```

```
Training-Accuracy: 0.967
Test-Accuracy:     0.933
```

Die Test-Accuracy ist für die Beurteilung wesentlich aussagekräftiger als die Trainings-Accuracy. Eine sehr hohe Trainingsleistung allein sagt noch nicht, wie gut das Modell auf neue Daten generalisiert.

Mit `early_stopping=True` trennt `MLPClassifier` bei den Solvern `adam` und `sgd` intern einen Teil der Trainingsdaten als Validierungsdaten ab. Bleibt die Validierungsleistung über mehrere Iterationen ohne ausreichende Verbesserung, wird das Training beendet.

```python
from sklearn.metrics import confusion_matrix, classification_report

y_pred = moon_model.predict(X_test)

print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

print("\nClassification Report:")
print(classification_report(y_test, y_pred, digits=3))
```

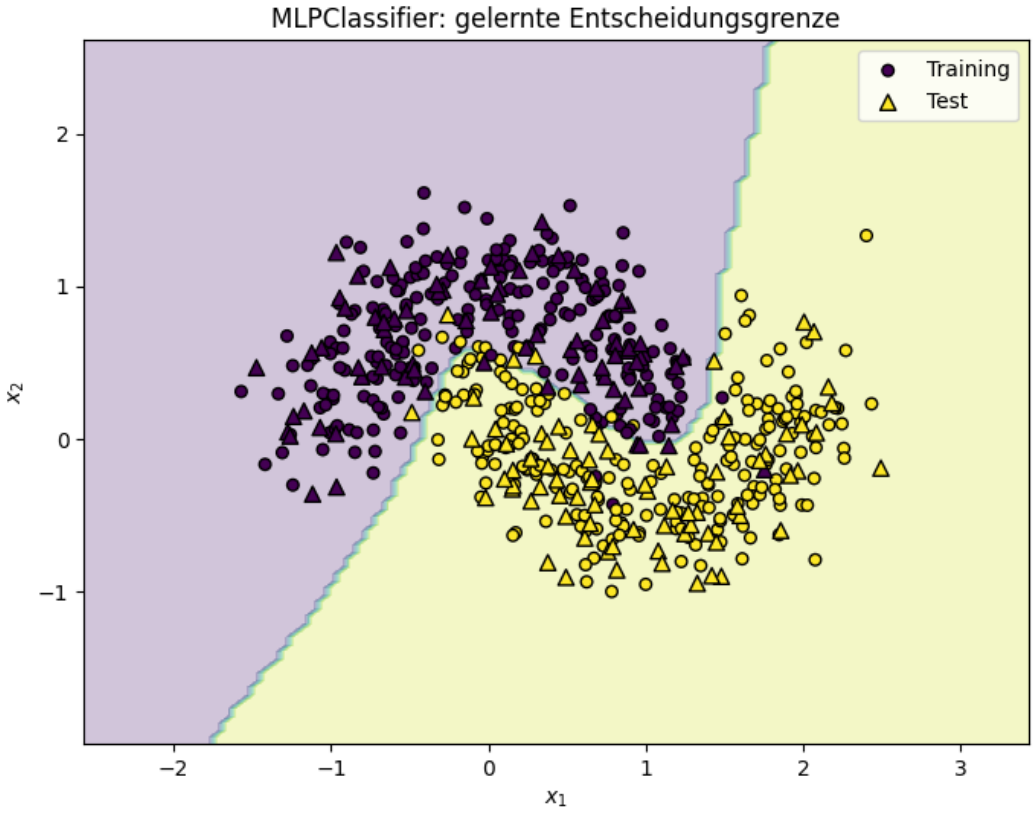
```
Confusion Matrix:
[[69  6]
 [ 4 71]]

Classification Report:
              precision    recall  f1-score   support

           0      0.945     0.920     0.932        75
           1      0.922     0.947     0.934        75

    accuracy                          0.933       150
   macro avg      0.934     0.933     0.933       150
```

```
weighted avg       0.934     0.933     0.933       150
```

13.0.5 Entscheidungsgrenze

Für zweidimensionale Lehrbeispiele lässt sich die vom MLP gelernte nichtlineare Entscheidungsgrenze direkt darstellen. Die Pipeline kann dabei unverändert verwendet werden; die Skalierung geschieht intern.

```python
import matplotlib.pyplot as plt
from sklearn.inspection import DecisionBoundaryDisplay

fig, ax = plt.subplots(figsize=(8, 6))

DecisionBoundaryDisplay.from_estimator(
    moon_model,
    X_train,
    response_method="predict",
    alpha=0.25,
    ax=ax,
)

ax.scatter(
    X_train[:, 0],
    X_train[:, 1],
    c=y_train,
    edgecolors="k",
    s=30,
    label="Training",
)
ax.scatter(
    X_test[:, 0],
    X_test[:, 1],
    c=y_test,
    edgecolors="k",
    marker="^",
    s=55,
    label="Test",
)

ax.set_title("MLPClassifier: gelernte Entscheidungsgrenze")
ax.set_xlabel("$x_1$")
ax.set_ylabel("$x_2$")
ax.legend()
plt.show()
```

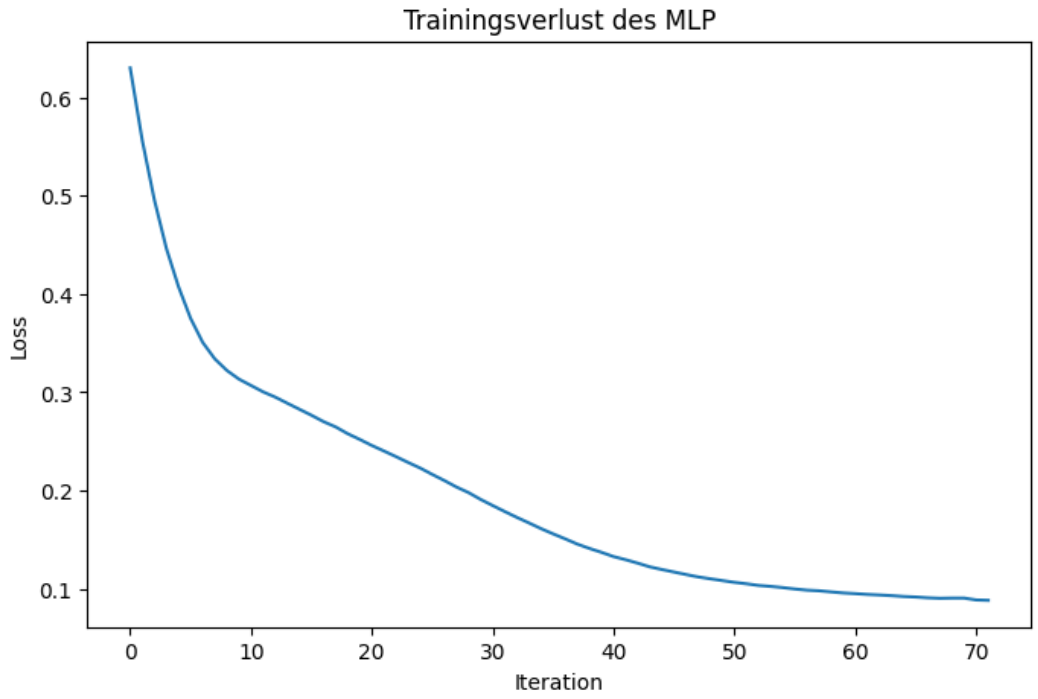


13.0.6 Lernkurve des Optimierers

Bei `adam` und `sgd` enthält `loss_curve_` den Trainingsverlust pro Iteration. Wenn Early Stopping aktiv ist, stehen zusätzlich die internen Validierungsscores in `validation_scores_`.

Die Kurven sind diagnostische Hilfsmittel. Sie ersetzen keine unabhängige Testauswertung.

```
mlp_moons = moon_model.named_steps["mlp"]

plt.figure(figsize=(8, 5))
plt.plot(mlp_moons.loss_curve_)
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.title("Trainingsverlust des MLP")
plt.show()

print("Benötigte Iterationen:", mlp_moons.n_iter_)
```




```
Benötigte Iterationen: 72
```

13.0.7 Wichtige Hyperparameter

Einige besonders wichtige Parameter von `MLPClassifier` sind:

- `hidden_layer_sizes`: Anzahl und Größe der verborgenen Schichten. `(30, 15)` bedeutet zwei Hidden Layers mit 30 beziehungsweise 15 Neuronen.
- `activation`: Aktivierungsfunktion der Hidden Layers, beispielsweise `rrelu"` oder `ttanh"`.
- `solver`: Optimierungsverfahren. `adam` ist ein guter allgemeiner Ausgangspunkt; `lbfgs` kann auf kleinen Datensätzen sehr gut funktionieren.
- `alpha`: Stärke der L2-Regularisierung der Gewichte.
- `learning_rate_init`: Anfangslernrate für `adam` und `sgd`.
- `max_iter`: maximale Anzahl Iterationen.
- `early_stopping`: beendet das Training anhand einer internen Validierungsmenge, wenn sich die Validierungsleistung nicht mehr verbessert.
- `random_state`: macht zufallsabhängige Teile des Trainings reproduzierbar.

Die beste Kombination hängt vom Datensatz ab. Hyperparameter sollten anhand von Cross-Validation auf den Trainingsdaten gewählt werden — nicht anhand des endgültigen Testsatzes.

13.0.8 Zusammenfassung

Ein robuster scikit-learn-Workflow für ein MLP sieht typischerweise so aus:

1. Trainings- und Testdaten sauber trennen.
2. Vorverarbeitung und `MLPClassifier` in einer Pipeline kapseln.
3. Hyperparameter nur auf den Trainingsdaten abstimmen.

4. Die endgültige Güte einmal auf dem bisher unberührten Testsatz beurteilen.
5. Bei stochastischem Training einen festen `random_state` verwenden, wenn reproduzierbare Lehrbeispiele gewünscht sind.

Scikit-learn stellt damit eine kompakte Möglichkeit bereit, klassische Multi-Layer-Perceptrons in denselben Modellwahl- und Evaluationsworkflow wie andere scikit-learn-Schätzer einzubetten.

14 Naive Bayes Klassifikator Einfuehrung

14.0.1 Naive Bayes Klassifikator

Definition

Im Bereich des Machine Learnings ist ein Bayes Klassifikator ein einfacher probabilistischer Klassifikator, der auf dem Bayes' Theorem basiert. Das Feature-Model, dass von einem Naive Bayes Klassifikator verwendet wird, macht starke Unabhängigkeitsannahmen. Das bedeutet, dass die Existenz eines bestimmten Features einer Klasse unabhängig ist zur Existenz jedes anderen Features der Klasse.

Definition unabhängiger Ereignisse:

Zwei Ereignisse E und F sind unabhängig, wenn beide E und F positive Wahrscheinlichkeiten haben und wenn P(E|F) = P(E) und P(F|E) = P(F) ist.

Wie bereits erwähnt, basiert der Naive Bayes Klassifikator auf dem Bayes' Theorem. Das Bayes' Theorem basiert auf bedingten Wahrscheinlichkeiten, die wir jetzt definieren:

Bedingte Wahrscheinlichkeit

$P(A|B)$ steht für "die bedingte Wahrscheinlichkeit von A mit gegebenen B", oder "die Wahrscheinlichkeit von A unter der Bedingung von B". Also die Wahrscheinlichkeit eines Ereignisses A unter der Annahme dass das Ereignis B eintritt. Wenn also in einem zufälligen Experiment das Ereignis B bekannt ist, werden die möglichen Ergebnisse des Experiments auf B reduziert, und daher wird die Wahrscheinlichkeit des Auftretens von A von der unbedingten Wahrscheinlichkeit in die bedingte Wahrscheinlichkeit von B geändert. Die gemeinsame Wahrscheinlichkeit ist die verbundene Wahrscheinlichkeit zweier Ereignisse. Es gibt drei Schreibweisen für die die gemeinsame Wahrscheinlichkeit von A und B.

- $P(AB)$ oder
- $P(AB)$ oder
- $P(A, B)$

Die bedingte Wahrscheinlichkeit wird wie folgt definiert:

$$P(A|B) = \frac{P(AB)}{P(B)}$$

Beispiele zur bedingten Wahrscheinlichkeit

Deutsch sprechende Schweizer Es leben über 8,4 Millionen Menschen in der Schweiz. Über 64% davon sprechen Deutsch. Auf der Erde leben insgesamt über 7,5 Milliarden Menschen.

Wenn Aliens einen Menschen auf der zufällig "wegbeamen" würden, wie hoch wäre die Wahrscheinlichkeit, dass es sich dabei um einen deutsch sprechenden Schweizer handelt?

Wir haben also folgende Ereignisse: - S: ist ein Schweizer - GS: spricht Deutsch

Die Wahrscheinlichkeit für einen zufällig ausgewählten Schweizer:

$$P(S) = \frac{8.4}{7500} = 0.00112$$

Wenn wir wissen, dass jemand Schweizer ist, so liegt die Wahrscheinlichkeit für einen deutsch sprechenden Schweizer bei 0.64 (64%). Dass entspricht der bedingten Wahrscheinlichkeit:

$$P(GS|S) = 0.64$$

Mit folgender Formel berechnen wir also die Wahrscheinlichkeit dafür, dass es sich bei dem Erdling um einen deutsch sprechenden Schweizer handelt:

$$P(GS|S) = \frac{P(GSS)}{P(S)}$$

Setzen wir noch die Werte ein:

$$0.64 = \frac{P(GSS)}{0.00112}$$

Ergebnis:

$$P(GSS) = 0.0007168$$

Unsere Aliens bekommen also mit einer Change von 0,07168% einen deutsch sprechenden Schweizer.

Falsche Positive und Falsche Negative Ein medizinisches Forschungs-Labor schlägt vor eine große Gruppe von Menschen hinsichtlich einer Erkrankung mit einem Screening zu testen. Ein Argument gegen das Screening ist das Problem von falschen Positiv Ergebnissen.

Angenommen dass 0,1% der Gruppe von der Erkrankung betroffen sind und der Rest gesund ist:

$$P(ick") = 0,1\% = 0.01$$

und

$$P("well") = 99,9\% = 0.999$$

Folgendes ist wahr für ein Screening: Wenn Sie die Krankheit haben, so wird der Test in 99% der Fälle Positiv sein. Sollten Sie die Krankheit nicht haben, so wird der Test in 99% Negativ sein:

$$P(ttestpositive"well") = 1\%$$

und

$$P(ttestnegative"well") = 99\%$$

Angenommen, dass Screening wird an einer Person durchgeführt die erkrankt ist, so gibt es eine Chance von 1% für ein falsches Negativ Ergebnis (und eine Chance von 99% für ein korrektes Positiv Ergebnis):

$$P(ttestnegativeick") = 1\%$$

und

$$P(ttestpositiveick") = 99\%$$

Es gibt also 999 falsche Positiv Ergebnisse und 1 falsches Negativ Ergebnis.

Problem:

In vielen Fällen nehmen sogar medizinische Fachkräfte an, dass der Test zu 99% positiv ist, wenn die Krankheit vorhanden ist und zu 99% negativ, wenn die Krankheit nicht vorhanden ist. Von den 1098 Fällen die ein positives Ergebnis liefern sind nur 99 (9%) Fälle korrekt und 999 Fälle sind falsche Positiv Ergebnisse (91%). Wenn eine Person also ein positives Test-Ergebnis bekommt, liegt die Wahrscheinlichkeit bei etwa 9% dass Er oder Sie tatsächlich die Krankheit hat.

$$P(ickttestpositive") = 99/1098 = 9.02\%$$

Bayes' Theorem

Wir haben die bedingte Wahrscheinlichkeit $P(GS|S)$ berechnet. Dabei ging es um die Wahrscheinlichkeit dass eine Person Deutsch spricht, wenn er bekannterweise ein Schweizer ist. Mit folgender Gleichung können wir dies berechnen:

$$P(GS|S) = \frac{P(GS, S)}{P(S)}$$

Wie sieht es mit der Berechnung der Wahrscheinlichkeit $P(S|GS)$ aus, also der Wahrscheinlichkeit dass jemand Schweizer ist unter der Annahme dass er Deutsch spricht?

Die Gleichung sieht so aus:

$$P(S|GS) = \frac{P(GS, S)}{P(GS)}$$

Isolieren wir aus beiden Gleichungen $P(GS, S)$:

$$P(GS, S) = P(GS|S)P(S)$$

$$P(GS, S) = P(S|GS)P(GS)$$

So wie die linken Seiten gleich sind, sollten die rechten Seiten ebenfalls gleich sein:

$$P(GS|S) * P(S) = P(S|GS)P(GS)$$

Diese Gleichung kann auch so geschrieben werden:

$$P(S|GS) = \frac{P(GS|S)P(S)}{P(GS)}$$

Das Ergebnis entspricht dem **Bayes' Theorem**.

Um das Problem zu lösen - also die Wahrscheinlichkeit dass ein Person Schweizer ist, wenn wir wissen dass er Deutsch spricht - müssen wir lediglich die rechte Seite berechnen. Aus dem vorigen Beispiel kennen wir bereits

$$P(GS|S) = 0.64$$

und

$$P(S) = 0.00112$$

Die Anzahl der nativ Deutsch sprechenden Menschen auf der Welt entspricht etwa 101 Millionen, somit kennen wir

$$P(GS) = \frac{101}{7500} = 0.0134667$$

Schließlich setzen wir die Werte in die Gleichung ein und berechnen $P(S|GS)$:

$$P(S|GS) = \frac{P(GS|S)P(S)}{P(GS)} = \frac{0.64 * 0.00112}{0.0134667} = 0.0532276$$

Es leben über 8,4 Millionen Menschen in der Schweiz. Über 64% davon sprechen Deutsch. Auf der Erde leben insgesamt über 7,5 Milliarden Menschen.

Wenn Aliens einen Menschen auf der zufällig "wegbeamen" würden, wie hoch wäre die Wahrscheinlichkeit, dass es sich dabei um einen deutsch sprechenden Schweizer handelt?

Wir haben also folgende Ereignisse: - S: ist ein Schweizer - GS: spricht Deutsch

Die Wahrscheinlichkeit für einen zufällig ausgewählten Schweizer:

$$P(S) = \frac{8.4}{7500} = 0.00112$$

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

$P(A|B)$ ist die bedingte Wahrscheinlichkeit von A und B ist gegeben (posteriore Wahrscheinlichkeit). $P(B)$ ist die vorherige Wahrscheinlichkeit von B und $P(A)$ entspricht der vorherigen Wahrscheinlichkeit von A. $P(B|A)$ ist die bedingte Wahrscheinlichkeit von B bei gegebenen A, auch "likely-hood" genannt.

Ein Vorteil des Naive Bayes Klassifikators ist, dass nur eine kleine Anzahl an Trainings-Daten notwendig ist um die Parameter abzuschätzen, die für die Klassifizierung notwendig sind. Da unabhängige Variablen angenommen werden, müssen nur die Varianzen der Variablen für jede Klasse bestimmt werden und nicht die gesamte Kovarianzmatrix.

14.0.2 Naive Bayes Klassifikator

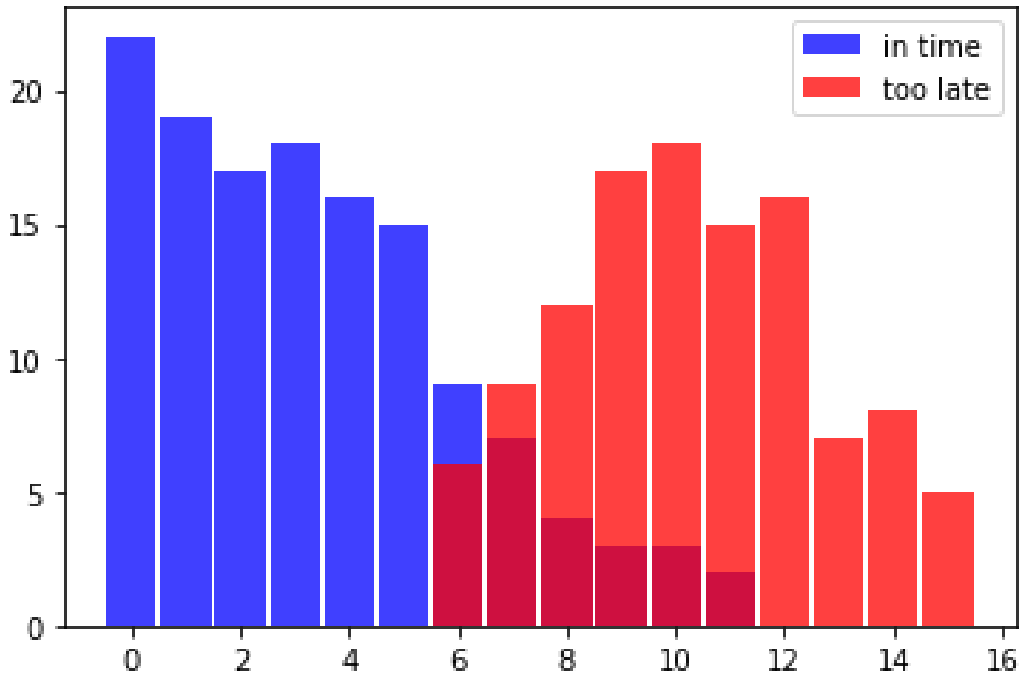
Einführungsübung

Begeben wir uns auf eine Zug-Reise um unseren ersten einfachen Naive Bayes Klassifikator zu erstellen. Nehmen wir an, dass wir uns in Hamburg befinden und nach München reisen möchten. In Frankfurt am Main müssen wir umsteigen. Von früheren Zug-Reisen wissen wir, dass unser Zug von Hamburg sich verspäten wird und somit nicht genügend Zeit bleibt um den Anschluss-Zug in Frankfurt zu bekommen. Die Wahrscheinlichkeit, dass wir den Anschluss-Zug verpassen werden hängt davon ab, wie hoch die mögliche Verspätung sein wird. Der Anschluss-Zug wird nicht länger als 5 Minuten warten. Manchmal hat der Anschluss-Zug aber ebenfalls Verstpätung.

Folgende Listen in_time (der Zug aus Hamburg ist pünktlich um den Anschluss-Zug nach München zu bekommen) und too_late (Anschluss-Zug verpasst) zeigen die Situation über einige Wochen. Die erste Komponente jedes Tuple stellt die Minuten dar, die der Zug zu spät angekommen ist und die zweite Komponente enthält die Anzahl der Vorkommnisse dazu.

```
# Tuple enthalten (Verspätung in Minuten, Anzahl der Verspätungen)
in_time = [(0, 22), (1, 19), (2, 17), (3, 18),
           (4, 16), (5, 15), (6, 9), (7, 7),
           (8, 4), (9, 3), (10, 3), (11, 2)]
too_late = [(6, 6), (7, 9), (8, 12), (9, 17),
            (10, 18), (11, 15), (12,16), (13, 7),
            (14, 8), (15, 5)]
```

```
%matplotlib inline

import matplotlib.pyplot as plt

X, Y = zip(*in_time)

X2, Y2 = zip(*too_late)

bar_width = 0.9
plt.bar(X, Y, bar_width,  color="blue", alpha=0.75, label="in time")
bar_width = 0.9
plt.bar(X2, Y2, bar_width,  color="red", alpha=0.75, label="too late")
plt.legend(loc='upper right')
plt.show()
```



Aus diesen Daten können wir die Wahrscheinlichkeit dafür ableiten, wenn wir eine Minute zu spät sind und den Anschluss-Zug bekommen wollen. Denn wir haben 19 erfolgreiche Fälle und keinen verpassten Fall. Es gibt also kein Tuple in der Liste too_late, in dem als erste Komponente eine 1 enthalten ist.

Dieses Ereignis “Zug ist pünktlich angekommen um den Anschluss-Zug zu erreichen” bezeichnen wir mit S (success) und das unglückliche Ereignis “Zug ist zu spät um den Anschluss-Zug zu bekommen” mit M (miss).

Wir können jetzt die Wahrscheinlichkeit formal definieren für “Zug bekommen, wenn wir eine Minute zu spät sind”:

$$P(S|1) = 19/19 = 1$$

Wir haben den Fakt genutzt, dass das Tuple $(1, 19)$ in in_time ist und es kein Tupel in der Liste too_late gibt mit 1 als erste Komponente.

Es wird kritisch wenn wir den Zug nach München wollen, aber 6 Minuten zu spät sind. Jetzt liegt die Chance nur noch bei 60%:

$$P(S|6) = 9/9 + 6 = 0.6$$

Entsprechend liegt die Wahrscheinlichkeit den Zug zu verpassen, wenn wir 6 Minuten zu spät sind, bei:

$$P(S|6) = 6/9 + 6 = 0.4$$

Wir können eine Klassifikator-Funktion schreiben, die uns die Wahrscheinlichkeit ausgibt, für die Erreichung des Anschluss-Zuges:

```
in_time_dict = dict(in_time)
too_late_dict = dict(too_late)

def catch_the_train(min):
    s = in_time_dict.get(min, 0)
    if s == 0:
        return 0
    else:
        m = too_late_dict.get(min, 0)
        return s / (s + m)

for minutes in range(-1, 13):
    print(minutes, catch_the_train(minutes))
```

```
-1 0
0 1.0
1 1.0
2 1.0
3 1.0
4 1.0
5 1.0
6 0.6
7 0.4375
8 0.25
9 0.15
10 0.14285714285714285
11 0.11764705882352941
12 0
```

Ein Naive Bayes Klassifikator Beispiel

Daten vorbereiten Wir verwenden die Datei person_data.txt. Sie enthält 100 zufällige Personen-Daten, männlich und weiblich, mit Körpergrößen, Gewicht und Geschlechts-Informationen.

```
import numpy as np

genders = ["male", "female"]
persons = []
with open("data/person_data.txt") as fh:
    for line in fh:
        persons.append(line.strip().split())

firstnames = {}
heights = {}

for gender in genders:
    firstnames[gender] = [ x[0] for x in persons if x[4]==gender]
    heights[gender] = [ x[2] for x in persons if x[4]==gender]
    heights[gender] = np.array(heights[gender], dtype=int)

for gender in ("female", "male"):
    print(gender + ":")
```

```
    print(firstnames[gender][:10])
    print(heights[gender][:10])
```

Warnung: Es könnte eine Verwechslung geben zwischen einer Python-Klasse und einer Naive Bayes-Klasse geben. Wir versuchen an den entsprechenden Stellen explizit zu erwähnen, was gemeint ist.

Design einer Feature-Klasse Wir definieren jetzt eine Python-Klasse Feature für die Features, die wir für die spätere Klassifikation brauchen.

Die Feature-Klasse braucht eine Bezeichnung (label), z.B. "Körpergrößen" oder "Vornamen". Wenn die Feature-Werte numerisch sind, könnten wir sie vielleicht auch auf die möglichen Feature-Werte reduzieren. Die Körpergrößen der Personen haben einen großen Wertebereich und wir haben lediglich 50 gemessene Werte für unsere Naive Bayes-Klasse "male" und "female". Wir setzen bin_width auf 5 und somit die Wertebereiche "130 bis 134", "135 bis 139", "140 bis 144", und so weiter. Vornamen können wir nicht zusammenfassen und somit setzen wir bin_width auf None.

Die Methode frequency liefert die Anzahl der Vorkommnisse für ein bestimmtes Feature oder zusammengefassten Wertebreich.

```
from collections import Counter
import numpy as np

class Feature:

    def __init__(self, data, name=None, bin_width=None):
        self.name = name
        self.bin_width = bin_width
        if bin_width:
            self.min, self.max = min(data), max(data)
            bins = np.arange((self.min // bin_width) * bin_width,
                             (self.max // bin_width) * bin_width,
                             bin_width)
            freq, bins = np.histogram(data, bins)
            self.freq_dict = dict(zip(bins, freq))
            self.freq_sum = sum(freq)
        else:
            self.freq_dict = dict(Counter(data))
            self.freq_sum = sum(self.freq_dict.values())

    def frequency(self, value):
        if self.bin_width:
            value = (value // self.bin_width) * self.bin_width
        if value in self.freq_dict:
            return self.freq_dict[value]
        else:
            return 0
```

Erstellen wir jetzt zwei Feature-Instanzen für die Körpergrößen des Personen-Data-Sets. Eine Feature-Instanz beinhaltet die Körpergrößen für die Naive Bayes-Klasse "male" und die andere Instanz die Körpergrößen für die Naive Bayes-Klasse "female":

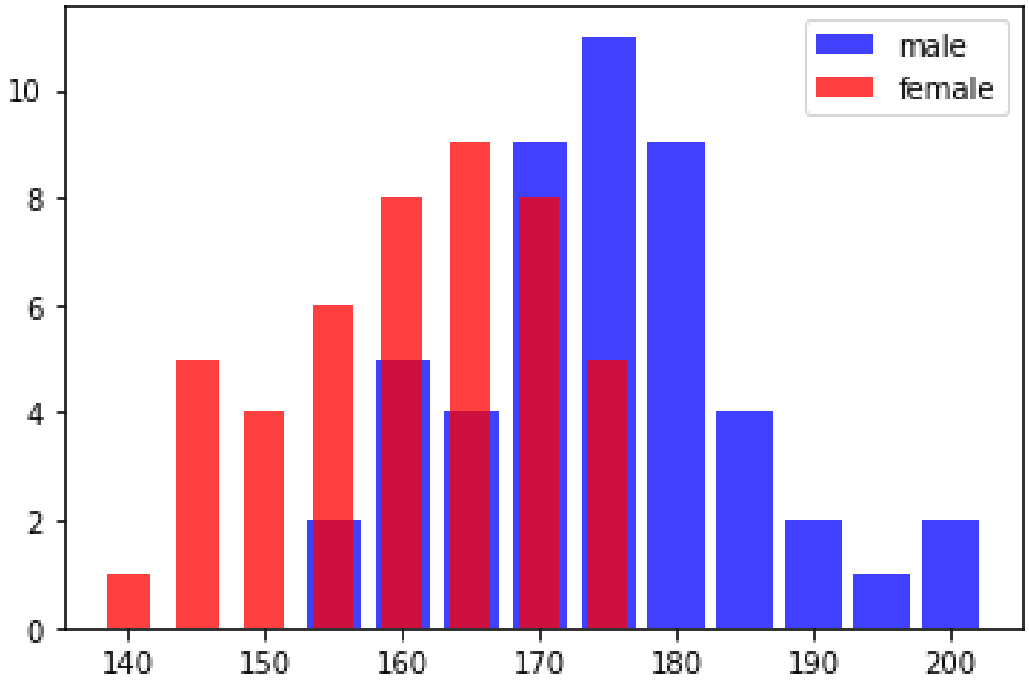
```
fts = {}
```

```
for gender in genders:
    fts[gender] = Feature(heights[gender], name=gender, bin_width=5)
    print(gender, fts[gender].freq_dict)
```

```
male {155: 2, 160: 5, 165: 4, 170: 9, 175: 11, 180: 9, 185: 4, 190: 2, 195: 1,
↪  200: 2}
female {140: 1, 145: 5, 150: 4, 155: 6, 160: 8, 165: 9, 170: 8, 175: 5}
```

Balken-Diagramm zur Frequenzverteilung Wir haben die Frequenzen der Features ausgegeben, jedoch ist die Darstellung in einem Balken-Diagramm deutlich besser. Mit folgendem Code generieren wir eine entsprechende Ausgabe:

```
for gender in genders:
    frequencies = list(fts[gender].freq_dict.items())
    frequencies.sort(key=lambda x: x[1])
    X, Y = zip(*frequencies)
    color = "blue" if gender=="male" else "red"
    bar_width = 4 if gender=="male" else 3
    plt.bar(X, Y, bar_width, color=color, alpha=0.75, label=gender)


plt.legend(loc='upper right')
plt.show()
```



Nun muss eine Naive Bayes-Klasse in Python erstellt werden. Wir nennen sie NBclass. Eine NBclass-Instanz beinhaltet eine oder mehrere Feature-Instanzen. Die Bezeichnung der NBclass wird in self.name abgelegt.

```
class NBclass:
```

```
        def __init__(self, name, *features):
            self.features = features
            self.name = name

        def probability_value_given_feature(self,
                                            feature_value,
                                            feature):
            """
            p_value_given_feature liefert die Wahrscheinlichkeit p
            für ein feature_value

            Ich versteh den Satz nicht!!!

            p_value_given_feature returns the probability p
            for a feature_value 'value' of the feature  to occurr
            corresponds to P(d_i | p_j)
            where d_i is a feature variable of the feature i
            """

            if feature.freq_sum == 0:
                return 0
            else:
                return feature.frequency(feature_value) / feature.freq_sum
```

Folgender Code erzeugen wir NBclass-Instanzen mit einem Feature, d.h. den Körpergrößen-Feature. Wir verwenden die Feature-Instanzen von fts, die wir schon vorher erstellt haben:

```
cls = {}
for gender in genders:
    cls[gender] = NBclass(gender, fts[gender])
```

Der letzte Schritt für einen einfachen Naive Bayes-Klassifikator ist eine Klasse Classifier, die wiederum die Klassen NBclass und Feature verwendet.

```
class Classifier:

    def __init__(self, *nbclasses):
        self.nbclasses = nbclasses

    def prob(self, *d, best_only=True):

        nbclasses = self.nbclasses
        probability_list = []

        for nbclass in nbclasses:
            ftrs = nbclass.features
            prob = 1

            for i in range(len(ftrs)):
                prob *= nbclass.probability_value_given_feature(d[i], ftrs[i])

            probability_list.append( (prob, nbclass.name) )

        prob_values = [f[0] for f in probability_list]
        prob_sum = sum(prob_values)
```

```
        if prob_sum==0:
            number_classes = len(self.nbclasses)
            pl = []

            for prob_element in probability_list:
                pl.append( ((1 / number_classes), prob_element[1]))

            probability_list = pl

        else:
            probability_list = [ (p[0] / prob_sum, p[1])  for p in
            ↪  probability_list]

        if best_only:
            return max(probability_list)

        else:
            return probability_list
```

Wir erstellen einen Klassifikator mit einer Feature-Instanz 'Körpergrößen'. Wir prüfen die Werte zwischen 130 und 220 cm.

```
c = Classifier(cls["male"], cls["female"])

for i in range(130, 220, 5):
    print(i, c.prob(i, best_only=False))
```

```
130 [(0.5, 'male'), (0.5, 'female')]
135 [(0.5, 'male'), (0.5, 'female')]
140 [(0.0, 'male'), (1.0, 'female')]
145 [(0.0, 'male'), (1.0, 'female')]
150 [(0.0, 'male'), (1.0, 'female')]
155 [(0.23834196891191708, 'male'), (0.7616580310880829, 'female')]
160 [(0.36977491961414793, 'male'), (0.6302250803858521, 'female')]
165 [(0.29439999999999994, 'male'), (0.7056, 'female')]
170 [(0.5136476426799008, 'male'), (0.4863523573200993, 'female')]
175 [(0.673768308921438, 'male'), (0.3262316910785619, 'female')]
180 [(1.0, 'male'), (0.0, 'female')]
185 [(1.0, 'male'), (0.0, 'female')]
190 [(1.0, 'male'), (0.0, 'female')]
195 [(1.0, 'male'), (0.0, 'female')]
200 [(1.0, 'male'), (0.0, 'female')]
205 [(0.5, 'male'), (0.5, 'female')]
210 [(0.5, 'male'), (0.5, 'female')]
215 [(0.5, 'male'), (0.5, 'female')]
```

Es gibt keine Personen - weder männlich noch weiblich - in unserem Lern-Set, mit einer Körpergröße zwischen 140 und 144 cm. Deshalb kann der Klassifikator nicht auf gelernte Daten aus diesem Wertebereich zugreifen und wir bekommen ein 50-50-Ergebnis.

Wir können den Klassifikator ebenfalls mit unseren Vornamen trainieren:

```
fts = {}
cls = {}
for gender in genders:
    fts_names = Feature(firstnames[gender], name=gender)
    cls[gender] = NBclass(gender, fts_names)
```

```
c = Classifier(cls["male"], cls["female"])

testnames = ['Edgar', 'Benjamin', 'Fred', 'Albert', 'Laura',
             'Maria', 'Paula', 'Sharon', 'Jessie']
for name in testnames:
    print(name, c.prob(name, best_only=False))
```

```
Edgar [(0.5, 'male'), (0.5, 'female')]
Benjamin [(1.0, 'male'), (0.0, 'female')]
Fred [(1.0, 'male'), (0.0, 'female')]
Albert [(1.0, 'male'), (0.0, 'female')]
Laura [(0.0, 'male'), (1.0, 'female')]
Maria [(0.0, 'male'), (1.0, 'female')]
Paula [(0.0, 'male'), (1.0, 'female')]
Sharon [(0.0, 'male'), (1.0, 'female')]
Jessie [(0.33333333333333337, 'male'), (0.6666666666666667, 'female')]
```

Der Name "Jessie" ist mehrdeutig. Es gibt 66 Jungs und 100 Mädchen mit diesem Namen. Aus dem vorherigen Klassifikations-Ergebnis lernen wir, dass die Wahrscheinlichkeit bei über zwei-drittel für "female" liegt, was durch unser Daten-Set person errechnet wurde:

```
[person for person in persons if person[0] == "Jessie"]
```

```
[['Jessie', 'Thomas', '170', '50', 'female'],
 ['Jessie', 'Morgan', '170', '75.0', 'male'],
 ['Jessie', 'Davis', '157', '47', 'female'],
 ['Jessie', 'Johnson', '163', '65.0', 'male'],
 ['Jessie', 'Washington', '169', '70', 'female'],
 ['Jessie', 'Bell', '166', '84', 'female']]
```

Jessie Washington ist nur 159 cm groß. Schauen wir uns die Ergebnisse des Klassifikators an, der mit Körpergrößen trainiert wurde, erkennen wir dass die Wahrscheinlichkeit dafür, dass eine 159 cm große Person weiblich ist, bei 0.875 liegt. Was ist mit einer unbekannten Person die "Jessie" heisst und 159 cm groß ist? Ist sie männlich oder weiblich?

Dafür trainieren wir einen Naive Bayes-Klassifikator mit zwei Feature-Instanzen, d.h. mit Körpergrößen und Vornamen:

```
cls = {}
for gender in genders:
    fts_heights = Feature(heights[gender], name="heights", bin_width=5)
    fts_names = Feature(firstnames[gender], name="names")

    cls[gender] = NBclass(gender, fts_names, fts_heights)


c = Classifier(cls["male"], cls["female"])

for d in [("Maria", 140), ("Anthony", 200), ("Anthony", 153),
          ("Jessie", 188) , ("Jessie", 159), ("Jessie", 160) ]:
    print(d, c.prob(*d, best_only=False))
```

```
('Maria', 140) [(0.0, 'male'), (1.0, 'female')]
```

```
('Anthony', 200) [(1.0, 'male'), (0.0, 'female')]
('Anthony', 153) [(0.5, 'male'), (0.5, 'female')]
('Jessie', 188) [(1.0, 'male'), (0.0, 'female')]
('Jessie', 159) [(0.13529411764705881, 'male'), (0.8647058823529411, 'female')]
('Jessie', 160) [(0.22682445759368838, 'male'), (0.7731755424063117, 'female')]
```

Die grundlegende Theorie

Unser Klassifikator basiert auf dem Bayes' Theorem:

Our classifier from the previous example is based on the Bayes theorem:

$$P(c_j|d) = \frac{P(d|c_j)P(c_j)}{P(d)}$$

dabei gilt

- $P(c_j|d)$ ist die Wahrscheinlichkeit einer Instanz d in der Klasse c_j zu sein. Dass wollen wir mit unserem Klassifikator berechnen.
- $P(d|c_j)$ ist die Wahrscheinlichkeit zur Generierung der Instanz d, wenn die Klasse c_j gegeben ist.
- $P(c_j)$ ist die Wahrscheinlichkeit für das Vorkommen der Klasse c_j. Wir verwenden dies aber nicht in unserem Klassifikator, weil beide Klassen in unserem Beispiel gleichermaßen wahrscheinlich sind.
- $P(d)$ ist die Wahrscheinlichkeit für das Vorkommen einer Instanz d. Dies wird in der Berechnung nicht benötigt, da dies für alle Klassen gleich ist.

Wir haben lediglich ein Feature in unserem vorigen Beispiel verwendet, d.h. sowohl die Körpergrößen als auch die Vornamen.

Es ist möglich einen Naive Bayes Klassifikator zu definieren mit mehreren Feature-Instanzen. z.B. $d = (d_1, d_2, ..., d_n)$

Wir erhalten dadurch folgende Formel:

$$P(c_j|d) = \frac{1}{P(d)} \prod_{i=1}^{n} P(d_i|c_j)P(c_j)$$

$\frac{1}{P(d)}$ hängt nur von den Werten $d_1, d_2, ...d_n$ ab. Es handelt sich dabei um eine Konstante, weil die Werte der Feature-Variablen bekannt sind.

15 Naive Bayes Klassifikator Scikit

15.0.1 Naive-Bayes-Klassifikator mit scikit-learn

Im vorigen Kapitel haben wir einen Naive-Bayes-Klassifikator von Grund auf implementiert. Hier verwenden wir `GaussianNB` aus scikit-learn. Der Gaussian-Naive-Bayes-Klassifikator modelliert die Verteilung jedes numerischen Merkmals innerhalb einer Klasse durch eine Gaußverteilung und kombiniert diese Wahrscheinlichkeiten unter der Naive-Bayes-Unabhängigkeitsannahme.

Im ersten Beispiel verwenden wir den Iris-Datensatz. Wichtig ist, Training und Bewertung zu trennen: Wir trainieren nur auf dem Trainingsanteil und berechnen den Report auf bislang ungesehenen Testdaten.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.naive_bayes import GaussianNB

iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, test_size=0.25, random_state=42,
    ↪  stratify=iris.target
)

model = GaussianNB()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

print(classification_report(y_test, y_pred, target_names=iris.target_names))
print("Konfusionsmatrix:\n", confusion_matrix(y_test, y_pred))
```

```
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        12
  versicolor       0.86      0.92      0.89        13
   virginica       0.92      0.85      0.88        13

    accuracy                           0.92        38
   macro avg       0.92      0.92      0.92        38
weighted avg       0.92      0.92      0.92        38

Konfusionsmatrix:
 [[12  0  0]
 [ 0 12  1]
 [ 0  2 11]]
```

Wir verwenden nun einmal die Personen-Daten aus dem vorigen Kapitel um einen weiteren Klassifikator zu trainieren:

```
import numpy as np

def prepare_person_dataset(fname):
    genders = ["male", "female"]
    persons = []
    with open(fname) as fh:
        for line in fh:
            persons.append(line.strip().split())

    dataset = [] #Größe, Gewicht und Geschlecht

    for person in persons:
        height_weight = (float(person[2]), float(person[3]))
        dataset.append( (height_weight, person[4]))
    return dataset

learnset = prepare_person_dataset("data/person_data.txt")
testset = prepare_person_dataset("data/person_data_testset.txt")
#print(learnset)
```

```
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.naive_bayes import GaussianNB

model = GaussianNB()
X_train_persons, y_train_persons = zip(*learnset)
X_test_persons, y_test_persons = zip(*testset)

X_train_persons = np.asarray(X_train_persons, dtype=float)
X_test_persons = np.asarray(X_test_persons, dtype=float)
y_train_persons = np.asarray(y_train_persons)
y_test_persons = np.asarray(y_test_persons)

model.fit(X_train_persons, y_train_persons)
y_pred_persons = model.predict(X_test_persons)

print(classification_report(y_test_persons, y_pred_persons, zero_division=0))
print("Konfusionsmatrix:\n", confusion_matrix(y_test_persons, y_pred_persons))
```

16 Gaussian Mixture Models und Expectation-Maximization

16.0.1 Probabilistisches Clustering mit scikit-learn

Ein **Gaussian Mixture Model (GMM)** beschreibt eine Wahrscheinlichkeitsverteilung als gewichtete Mischung mehrerer Gauß-Verteilungen. GMMs werden häufig für unüberwachtes Lernen, Clustering und Dichteschätzung eingesetzt.

Anders als ein rein hartes Clustering kann ein GMM für jeden Datenpunkt angeben, mit welcher Wahrscheinlichkeit er zu den einzelnen Komponenten gehört. Diese Wahrscheinlichkeiten werden häufig *Responsibilities* genannt.

In scikit-learn wird ein klassisches GMM durch `sklearn.mixture.GaussianMixture` bereitgestellt.

Dokumentation: GaussianMixture

16.0.2 GMM und k-means: wichtige Unterschiede

GMM wird gelegentlich mit k-means verwechselt. Beide Verfahren können Clusterstrukturen entdecken, verfolgen aber unterschiedliche Modelle.

k-means

- ordnet jeden Punkt genau einem Cluster zu (*hard assignment*),
- minimiert quadrierte euklidische Abstände zu Zentren,
- führt dadurch zu Voronoi-Zellen um die Clusterzentren.

Gaussian Mixture Model

- modelliert eine vollständige Wahrscheinlichkeitsdichte,
- liefert für jeden Punkt Wahrscheinlichkeiten für alle Komponenten (*soft assignment*),
- kann mit `covariance_type=ffull"` unterschiedlich orientierte ellipsoidale Komponenten beschreiben,
- berücksichtigt neben Lage und Kovarianz auch das Mischungsgewicht jeder Komponente.

Ein GMM ist daher kein „kNN mit Ellipsen“. **k-nearest neighbors (kNN)** ist ein anderes, überwiegend überwachtes Verfahren und gehört konzeptionell nicht in diesen Vergleich.

16.0.3 Mathematisches Modell

Für (K) Komponenten lautet die Mischungsverteilung

[p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k,\mathcal{N}(\mathbf{x}\mid \boldsymbol{\mu}_k,\boldsymbol{\Sigma}_k),]

wobei (\pi_k \ge 0), (\sum_k \pi_k = 1), (\boldsymbol{\mu}_k) den Mittelwertvektor und (\boldsymbol{\Sigma}_k) die Kovarianzmatrix bezeichnet.

Die Zugehörigkeitswahrscheinlichkeit (*Responsibility*) eines Datenpunkts (\mathbf{x}_i) zur Komponente (k) ist

[\gamma_{ik} = \frac{\pi_k,\mathcal{N}(\mathbf{x}_i\mid\boldsymbol{\mu}_k,\boldsymbol{\Sigma}*k)} {\sum*{j=1}^{K}\pi_j,\mathcal{N}(\mathbf{x}_i\mid\boldsymbol{\mu}_j,\boldsymbo]

Für jeden Datenpunkt gilt (\sum_k \gamma_{ik}=1).

16.0.4 Expectation-Maximization (EM)

Die Parameter eines GMM werden typischerweise iterativ mit dem **Expectation-Maximization-Algorithmus** geschätzt.

E-Schritt

Mit den aktuellen Parametern werden die Responsibilities (\gamma_{ik}) berechnet.

M-Schritt

Mit (N_k = \sum_i \gamma_{ik}) werden die Parameter neu geschätzt:

[\pi_k = \frac{N_k}{N}, \qquad \boldsymbol{\mu}*k = *frac{1}{N_k}\sum_i \gamma*{ik}\mathbf{x}_i,]

[\boldsymbol{\Sigma}*k = *frac{1}{N_k}\sum_i *gamma*{ik} (\mathbf{x}_i-\boldsymbol{\mu}_k) (\mathbf{x}_i-\boldsymbol{\mu}_k)^T.]

Danach beginnt erneut der E-Schritt. EM erhöht die Log-Likelihood bei jedem regulären Iterationsschritt beziehungsweise lässt sie unverändert, kann aber in einem **lokalen** Optimum enden. Deshalb sind Initialisierung und gegebenenfalls mehrere Initialisierungen (`n_init`) wichtig.

16.0.5 EM in einer Dimension selbst implementiert

Zunächst implementieren wir den Kern des Verfahrens für drei eindimensionale Gauß-Komponenten. Das Beispiel ist bewusst kompakt gehalten: Es soll die E- und M-Schritte sichtbar machen, nicht `GaussianMixture` nachbauen.

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

rng = np.random.default_rng(42)

x = np.concatenate([
    rng.normal(-4.0, 0.8, 180),
    rng.normal(0.5, 1.1, 260),
    rng.normal(5.0, 0.6, 160),
])
```

```
rng.shuffle(x)

def normal_pdf(x, mean, variance):
    return np.exp(-0.5 * (x - mean) ** 2 / variance) / np.sqrt(
        2 * np.pi * variance
    )

K = 3
weights = np.full(K, 1 / K)
means = np.array([-5.5, -0.5, 4.0], dtype=float)
variances = np.full(K, 2.0)

log_likelihood = []

for _ in range(40):
    weighted_densities = np.column_stack([
        weights[k] * normal_pdf(x, means[k], variances[k])
        for k in range(K)
    ])

    total_density = weighted_densities.sum(axis=1, keepdims=True)
    responsibilities = weighted_densities / total_density
    log_likelihood.append(np.log(total_density[:, 0]).sum())

    N_k = responsibilities.sum(axis=0)
    weights = N_k / len(x)
    means = (responsibilities * x[:, None]).sum(axis=0) / N_k
    variances = (
        responsibilities * (x[:, None] - means) ** 2
    ).sum(axis=0) / N_k

order = np.argsort(means)

print("Gewichte:", np.round(weights[order], 3))
print("Mittelwerte:", np.round(means[order], 3))
print("Standardabweichungen:", np.round(np.sqrt(variances[order]), 3))
print("Zeilensummen der Responsibilities:",
      np.round(responsibilities[:5].sum(axis=1), 6))
```

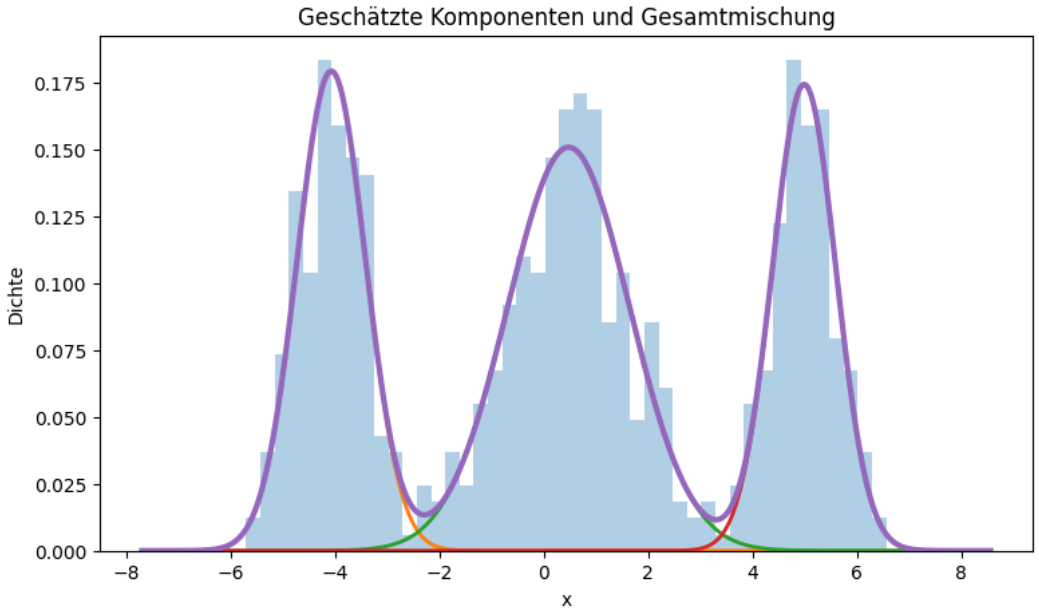
```
Gewichte: [0.295 0.44  0.265]
Mittelwerte: [-4.076  0.472  4.988]
Standardabweichungen: [0.655 1.164 0.607]
Zeilensummen der Responsibilities: [1. 1. 1. 1. 1.]
```

Die Zeilensummen der Responsibility-Matrix sind 1. Jeder Datenpunkt verteilt seine Zugehörigkeit also probabilistisch über alle Komponenten.

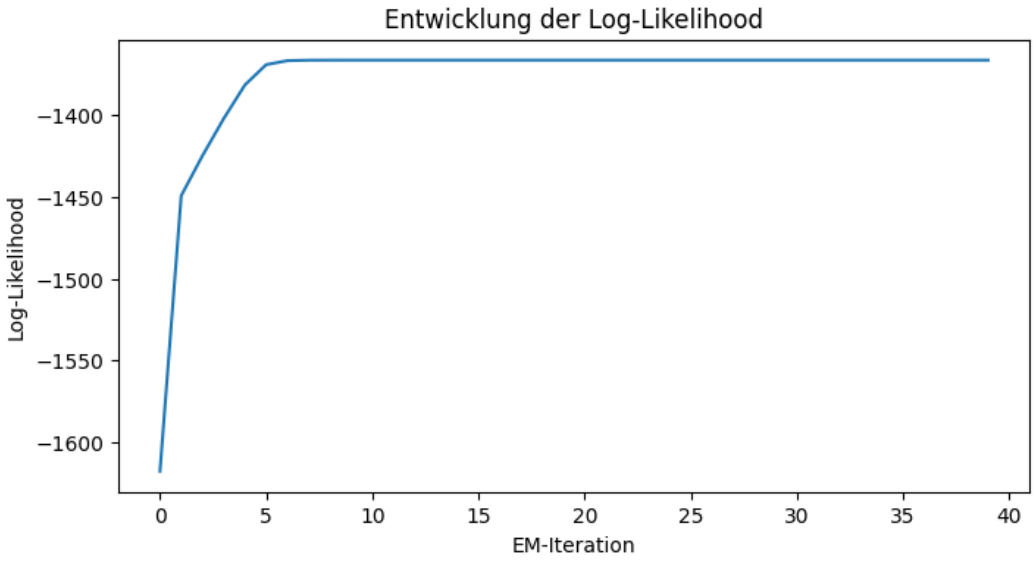
Die geschätzten Mittelwerte und Standardabweichungen sollten nahe bei den Parametern liegen, mit denen die Daten erzeugt wurden. Die Reihenfolge der Komponenten ist grundsätzlich bedeutungslos: „Komponente 0“ besitzt keine fest vorgegebene inhaltliche Bedeutung.

```
grid = np.linspace(x.min() - 2, x.max() + 2, 600)

plt.figure(figsize=(9, 5))
plt.hist(x, bins=45, density=True, alpha=0.35)
```

```
mixture_density = np.zeros_like(grid)

for k in order:
    component = weights[k] * normal_pdf(grid, means[k], variances[k])
    mixture_density += component
    plt.plot(grid, component, linewidth=2)

plt.plot(grid, mixture_density, linewidth=3)
plt.xlabel("x")
plt.ylabel("Dichte")
plt.title("Geschätzte Komponenten und Gesamtmischung")
plt.show()
```




```
plt.figure(figsize=(8, 4))
plt.plot(log_likelihood)
plt.xlabel("EM-Iteration")
plt.ylabel("Log-Likelihood")
plt.title("Entwicklung der Log-Likelihood")
plt.show()

diffs = np.diff(log_likelihood)
print("Kleinste Änderung:", diffs.min())
```




```
Kleinste Änderung: -4.547473508864641e-13
```

Numerisch können durch Rundung winzige Abweichungen auftreten. Im idealen EM-Ablauf sinkt die Log-Likelihood jedoch nicht. Das Verfahren stoppt, wenn sich die Zielfunktion beziehungsweise die Parameter nur noch vernachlässigbar ändern.

16.0.6 Zweidimensionales GMM mit scikit-learn

Nun verwenden wir `GaussianMixture` auf einem zweidimensionalen Datensatz. Die Daten werden linear transformiert, damit die Cluster nicht nur kreisförmig, sondern anisotrop und unterschiedlich orientiert erscheinen.

```python
from sklearn.datasets import make_blobs
from sklearn.mixture import GaussianMixture

X, _ = make_blobs(
    n_samples=700,
    centers=3,
    cluster_std=(0.8, 1.0, 0.7),
    random_state=12,
)

transformation = np.array([
    [0.7, -0.6],
    [0.35, 1.1],
])
X = X @ transformation

gmm_model = GaussianMixture(
    n_components=3,
    covariance_type="full",
    n_init=5,
    reg_covar=1e-6,
    random_state=12,
)

gmm_model.fit(X)
```

```
print("Konvergiert:", gmm_model.converged_)
print("Iterationen:", gmm_model.n_iter_)
print("Gewichte:", np.round(gmm_model.weights_, 3))
print("Mittelwerte:")
print(np.round(gmm_model.means_, 3))
```

```
Konvergiert: True
Iterationen: 3
Gewichte: [0.336 0.331 0.333]
Mittelwerte:
[[-3.226  9.379]
 [-2.991  3.574]
 [-3.923 15.016]]
```

`GaussianMixture` verwendet EM zur Maximum-Likelihood-Schätzung. `n_init=5` startet die Parameterschätzung mehrfach und behält die beste Lösung. Das ist hilfreich, weil EM von der Initialisierung abhängt.

`reg_covar` addiert einen kleinen nichtnegativen Wert auf die Diagonale der Kovarianzmatrizen. Dadurch werden die Kovarianzen numerisch stabiler und positiv definit gehalten.

```
probabilities = gmm_model.predict_proba(X[:5])
hard_labels = gmm_model.predict(X[:5])

print("Responsibilities der ersten fünf Punkte:")
print(np.round(probabilities, 3))
print("\nZeilensummen:", np.round(probabilities.sum(axis=1), 6))
print("Harte Zuordnung:", hard_labels)
```

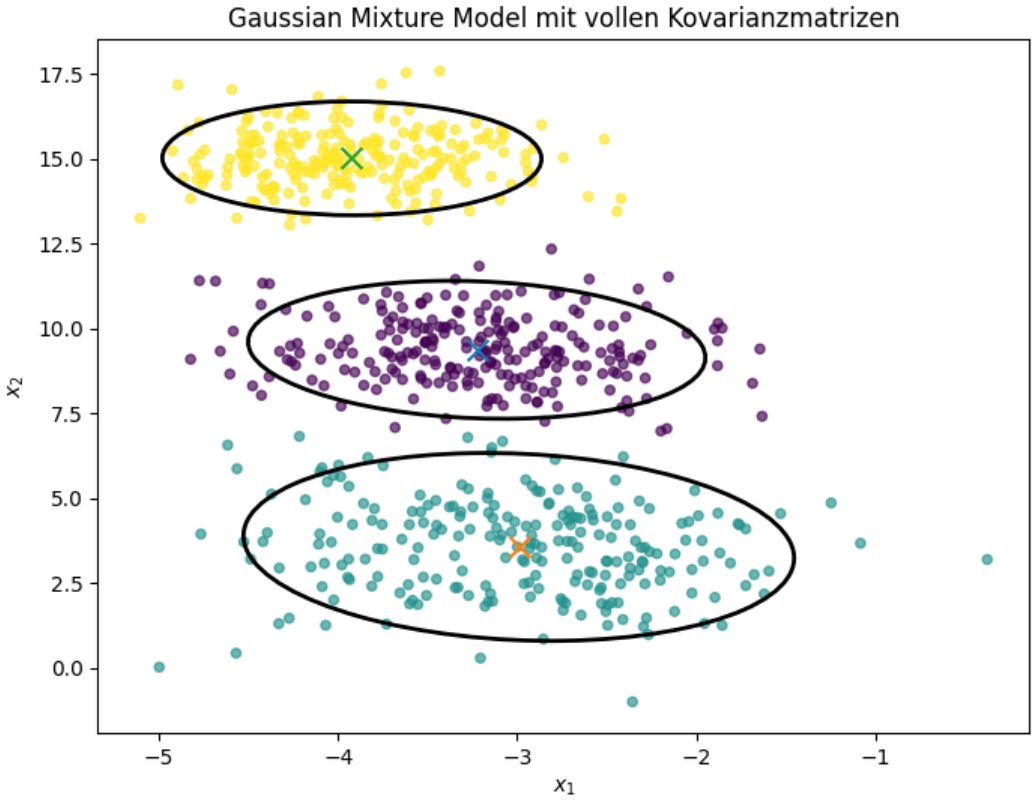
```
Responsibilities der ersten fünf Punkte:
[[0. 0. 1.]
 [0. 0. 1.]
 [0. 1. 0.]
 [1. 0. 0.]
 [0. 1. 0.]]

Zeilensummen: [1. 1. 1. 1. 1.]
Harte Zuordnung: [2 2 1 0 1]
```

`predict_proba` liefert die Responsibilities. `predict` reduziert diese Information auf die jeweils wahrscheinlichste Komponente.

Die numerischen Clusterlabels sind **beliebig**. Bei unüberwachtem Lernen darf man nicht erwarten, dass die Nummern automatisch mit eventuell vorhandenen Klassenlabels übereinstimmen.

```
from matplotlib.patches import Ellipse

def add_covariance_ellipse(ax, mean, covariance, n_std=2.0):
    eigenvalues, eigenvectors = np.linalg.eigh(covariance)
    order = np.argsort(eigenvalues)[::-1]
    eigenvalues = eigenvalues[order]
    eigenvectors = eigenvectors[:, order]

    angle = np.degrees(
```

```
        np.arctan2(eigenvectors[1, 0], eigenvectors[0, 0])
    )
    width, height = 2 * n_std * np.sqrt(eigenvalues)

    ellipse = Ellipse(
        xy=mean,
        width=width,
        height=height,
        angle=angle,
        fill=False,
        linewidth=2,
    )
    ax.add_patch(ellipse)

labels = gmm_model.predict(X)

fig, ax = plt.subplots(figsize=(8, 6))
ax.scatter(X[:, 0], X[:, 1], c=labels, s=20, alpha=0.65)

for mean, covariance in zip(gmm_model.means_, gmm_model.covariances_):
    add_covariance_ellipse(ax, mean, covariance)
    ax.scatter(mean[0], mean[1], marker="x", s=100)

ax.set_title("Gaussian Mixture Model mit vollen Kovarianzmatrizen")
ax.set_xlabel("$x_1$")
ax.set_ylabel("$x_2$")
plt.show()
```



16.0.7 Wie viele Komponenten?

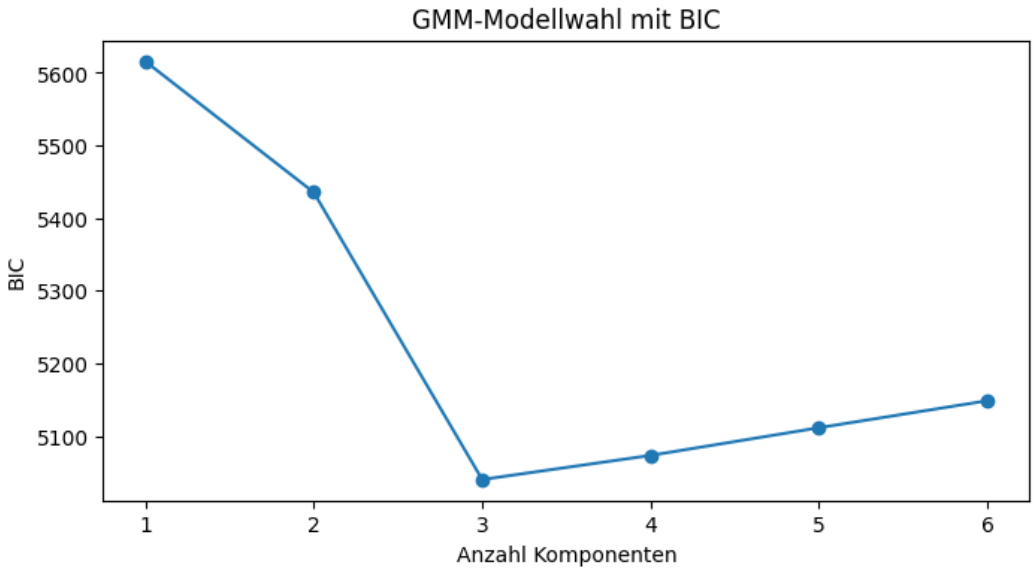
Die Anzahl der Komponenten ist nicht immer bekannt. Die reine Trainings-Log-Likelihood ist dafür ungeeignet, weil ein komplexeres Modell mit mehr Komponenten die Daten meist mindestens ebenso gut anpassen kann.

Für GMMs können beispielsweise **AIC** und **BIC** verwendet werden. Beide bestrafen zusätzliche Modellkomplexität. Bei BIC gilt: **kleiner ist besser**.

Das folgende Beispiel vergleicht Modelle mit einer bis sechs Komponenten.

```
component_counts = range(1, 7)
bic_values = []
models = []

for n_components in component_counts:
    model = GaussianMixture(
        n_components=n_components,
        covariance_type="full",
        n_init=5,
        reg_covar=1e-6,
        random_state=12,
    ).fit(X)
    models.append(model)
    bic_values.append(model.bic(X))

best_index = int(np.argmin(bic_values))
best_components = list(component_counts)[best_index]

print("Nach BIC bevorzugte Komponentenanzahl:", best_components)

plt.figure(figsize=(8, 4))
plt.plot(list(component_counts), bic_values, marker="o")
plt.xlabel("Anzahl Komponenten")
plt.ylabel("BIC")
plt.title("GMM-Modellwahl mit BIC")
plt.show()
```

```
Nach BIC bevorzugte Komponentenanzahl: 3
```



BIC ist ein Auswahlkriterium, kein Beweis dafür, dass eine bestimmte „wahre“ Clusterzahl existiert. Ob die statistischen Komponenten fachlich sinnvolle Gruppen darstellen, muss immer im Kontext der Daten beurteilt werden.

Scikit-learn zeigt in seinem offiziellen Beispiel ebenfalls die Auswahl von Komponentenanzahl und Kovarianztyp mit Informationskriterien:

Gaussian Mixture Model Selection

16.0.8 Kovarianztypen

`GaussianMixture` unterstützt mehrere Annahmen für die Kovarianzmatrizen:

- `ffull"`: jede Komponente besitzt eine eigene vollständige Kovarianzmatrix,
- `ttied"`: alle Komponenten teilen sich eine vollständige Kovarianzmatrix,
- `"diag"`: jede Komponente besitzt nur diagonale Kovarianzen,
- `ßpherical"`: pro Komponente wird nur eine Varianz verwendet.

`ffull"` ist am flexibelsten, benötigt aber auch die meisten Parameter. Bei wenig Daten oder hoher Dimensionalität können einfachere Kovarianzmodelle stabiler sein.

16.0.9 Numerische Singularitäten und Regularisierung

Ein klassisches Problem von Gauß-Mischungen entsteht, wenn eine Komponente auf sehr wenige Punkte oder praktisch auf einen einzelnen Punkt kollabiert. Dann kann ihre Kovarianzmatrix nahezu singulär werden und die Likelihood ohne geeignete Regularisierung degenerieren.

Scikit-learn begegnet dem unter anderem mit `reg_covar`. Der Parameter fügt einen kleinen Wert zur Diagonale der Kovarianzmatrix hinzu. Der Standardwert ist in der aktuellen API `1e-6`.

Mehrere Initialisierungen (`n_init`) und eine sachgerechte Wahl der Komponentenanzahl sind ebenfalls wichtig. Eine extrem große Anzahl von Komponenten ist nicht automatisch ein besseres Modell.

16.0.10 `BayesianGaussianMixture`

Wenn die Zahl der relevanten Komponenten nicht fest vorgegeben werden soll, bietet scikit-learn außerdem `BayesianGaussianMixture`. Dieses Modell verwendet variationale Bayes-Schätzung und kann bei ausreichend vielen initial erlaubten Komponenten unwichtige Komponenten stark heruntergewichten.

Das ist konzeptionell ein anderes Schätzverfahren als das klassische Maximum-Likelihood-GMM mit EM und sollte deshalb nicht einfach als austauschbarer Parameter betrachtet werden.

Dokumentation: BayesianGaussianMixture

16.0.11 Zusammenfassung

Gaussian Mixture Models sind besonders nützlich, wenn

- Cluster probabilistisch statt nur hart zugeordnet werden sollen,
- Komponenten unterschiedliche Streuungen und Korrelationen besitzen können,
- eine Dichteschätzung gewünscht ist.

Die zentralen Punkte sind:

1. GMM modelliert eine Mischung von Wahrscheinlichkeitsverteilungen.
2. EM wechselt zwischen Responsibility-Berechnung und Parameterschätzung.
3. Initialisierung ist wichtig, weil EM lokale Optima finden kann.
4. `predict_proba` liefert die weichen Clusterzugehörigkeiten.
5. AIC/BIC können bei der Modellwahl helfen.
6. `reg_covar` verbessert die numerische Stabilität der Kovarianzmatrizen.

Damit ist GMM deutlich mehr als nur ein alternatives Verfahren zum Zeichnen von Clustergrenzen: Es ist ein probabilistisches generatives Modell der Datenverteilung.

Weitere Bücher von Bernd Klein

Die folgenden Verlagstitel sind eigenständige Veröffentlichungen beim Carl Hanser Verlag. Diese automatisch erzeugte PDF-Ausgabe von `python-kurs.eu` ist keine Hanser-Veröffentlichung.

Einführung in Python 3

Für Ein- und Umsteiger

Bernd Klein

ISBN: 978-3-446-46379-0

4. Auflage, 2021

Mehr beim Carl Hanser Verlag

Numerisches Python

Arbeiten mit NumPy, Matplotlib und Pandas

Bernd Klein

ISBN: 978-3-446-48584-6

3. Auflage, 2025

Mehr beim Carl Hanser Verlag

Funktionale Programmierung mit Python

Funktionale Konzepte praxisnah in Python einsetzen

Bernd Klein, Philip Klein

ISBN: 978-3-446-48191-6

1. Auflage, 2025

Mehr beim Carl Hanser Verlag

Hinweis zur Ausgabe

Diese Ausgabe wurde automatisiert aus den Quellen von `python-kurs.eu` erzeugt. Die Online-Fassung kann aktueller sein als diese PDF-Ausgabe.