Bernhard Wurm
Sebastian Steininger

Inkl. Lernumgebung

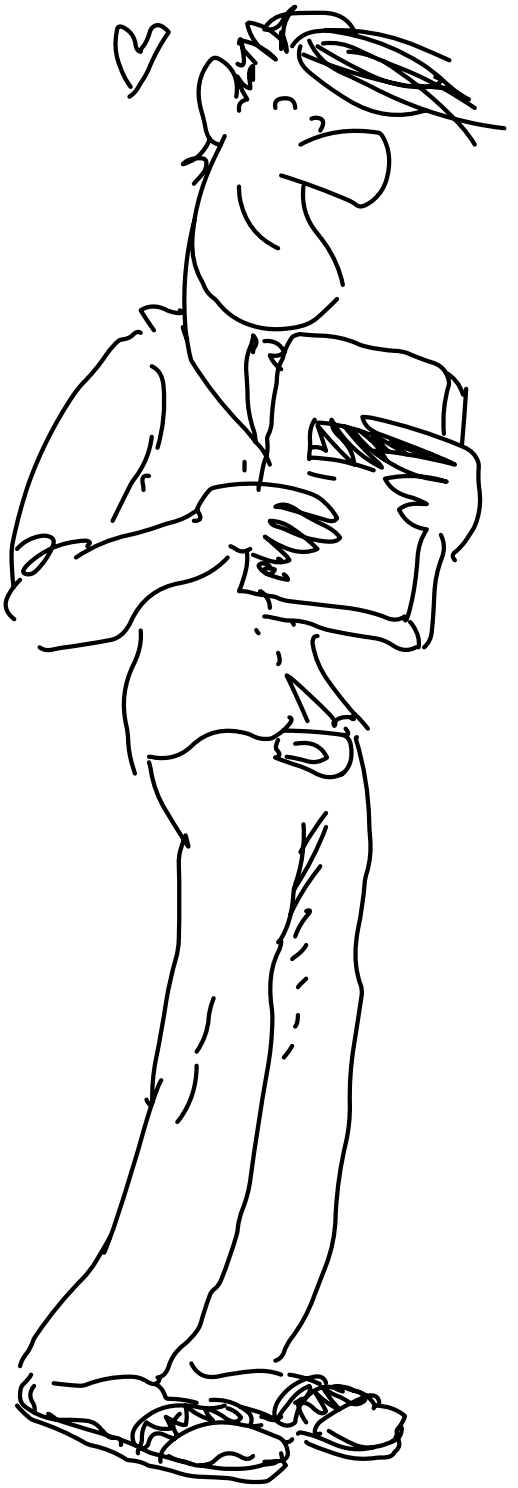
Mit Syntax-Highlighting!

Schrödinger programmiert KI

Das etwas andere Fachbuch

- Von k-nearest Neighbours bis zur eigenen LLM
- Mit Online-Lernumgebung, fantastisch illustriert
- Durchblicken, coden und genießen!

Rheinwerk Computing



MIT TALENT, KATZEN-
PHOBIE UND LÄSSIGEM
SCHUHWERK BESTACH
SCHRÖDINGER DIE
RHEINWERK-JURY.
FÜR IHN GEHT JETZT
EIN TRAUM IN ERFÜLLUNG.

Liebe Leserin, lieber Leser,

Du hast dir was vorgenommen:

künstliche Intelligenz programmieren!

Und dabei unterstützen wir (und natürlich Schrödinger) dich tatkräftig. Lass dich von zwei hervorragenden Autoren begleiten, die sich ordentlich ins Zeug gelegt haben, um dir **Neuronale Netze, Entscheidungsbäume und allerhand Algorithmen** verständlich und Schritt für Schritt näherzubringen.

Schrödinger nimmt dir dabei das Lernen zwar nicht ab, stellt aber mit Sicherheit die ein oder andere gute Frage und tüftelt mit dir am Code, **bis alles sitzt und du alles verstanden hast**. Dank eingefärbtem Code, jeder Menge Übungen und Tipps und Tricks werdet ihr das Kind schon schaukeln!

Na dann auf in die wilde Welt der KI – wir wünschen viel Spaß!

Hast du Feedback oder Fragen? Dann melde dich gerne über *schroedinger@rheinwerk-verlag.de* bei uns.

—ZWEI—

Auf gute Nachbarschaft

Abstands-metriken, K-Means, DBScan und K-Nearest-Neighbor

Auch Datenpunkte haben Nachbarn. Sobald geklärt ist, was »nah« genau heißen soll, geht es um drei Algorithmen, die voll auf Nachbarschaft abfahren: K-Means, den Influencer unter den Nachbarn, der immer im Mittelpunkt stehen muss, sowie den angepassten K-Nearest-Neighbor, der keine eigene Meinung zu haben scheint, und schließlich DBScan, der eine Party nur dann schmeißt, wenn auch genug Besucher kommen.

Geh auf Distanz!

Nun, lieber Schrödinger, zeige ich dir ein paar Klassifizierungsverfahren. Diese Verfahren kannst du dir alle räumlich vorstellen. Die Algorithmen betrachten die Daten(punkte) im Raum und deren Abstände zueinander.

Welchen Raum meinst du?

Die Merkmale – also die Features – spannen einen Raum auf. Einen Raum mit vielen Dimensionen. Jedes Merkmal spannt dabei eine eigene Dimension auf. Die Werte des Merkmals werden in dieser Achse eingereiht. So erhalten wir für jeden Datensatz einen Vektor mit Werten. Jeder Wert entspricht einem Merkmal.

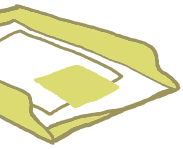
$$\begin{matrix} Höchstgeschwindigkeit = 280\ km/h \\ Leistung = 306\ PS \\ Gewicht = 2200\ kg \end{matrix} \quad ==> \quad \begin{pmatrix} 280 \\ 306 \\ 2200 \end{pmatrix}$$

[Zettel]
Zur Veranschaulichung werden wir uns immer mit einer, meistens zwei, manchmal auch drei Dimensionen begnügen. Aber es funktioniert immer auch mit vielen Hunderten Dimensionen!

Das Schöne an Punkten in einer Ebene oder auch im (n-dimensionalen) Raum ist, dass wir Abstände messen können.
Wir haben bereits besprochen, dass es häufig nur darum geht, Datensätze so in einem Raum anzuordnen, dass diese einer bestimmten Semantik – also Bedeutung – folgen.
Beispielsweise gibt es für die Verarbeitung von Sprache ein Modell mit dem Namen »Word2Vec«.
Dieses Modell erlaubt es, Wörter auf eine ganz bestimmte Art und Weise in einen Vektor umzurechnen.

Und was bringt das?

Die Vektoren sind so angeordnet, dass wir damit rechnen können und dass sie die Bedeutung abbilden. Beispielsweise können wir die Wörter »König« und »Sohn« in Vektoren umrechnen, sie addieren und wieder in ein Wort zurückwandeln. Dieser Ziel-Vektor zeigt ziemlich genau dorthin, wo auch das Wort »Prinz« steht. Addieren wir zum Wort »Prinz« das Wort »Mädchen«, sind wir in der Nähe der »Prinzessin«.



[Ablage]
Die Vektoren sind die Positionen der Wörter in einem hochdimensionalen Raum.

Genial!

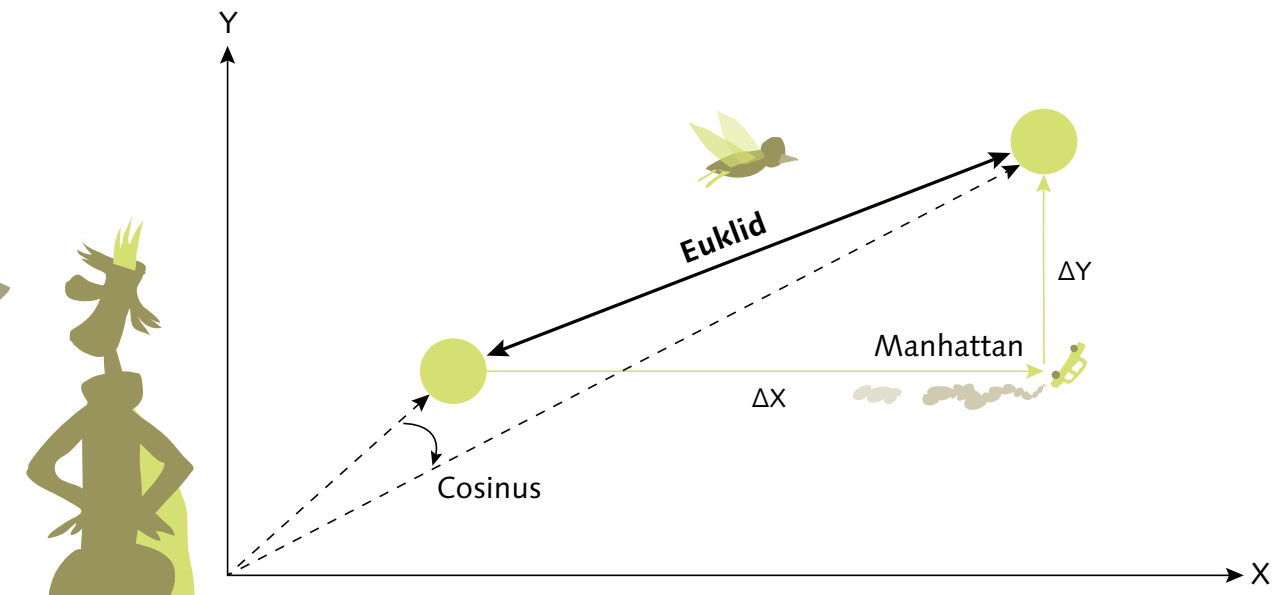
Um Algorithmen auf die Daten anzuwenden, brauchen wir ein Maß für den Abstand, also die Distanz zwischen Datenpunkten – eine **Distanzmetrik**. Es gibt verschiedene Distanzmetriken, wir schauen uns ein paar einfache an.

Wie willst du den Abstand anders messen als mit der Linie zwischen den Punkten?

Die Linie ist der intuitivste Abstand – die euklidische Distanz. Ich zeige dir noch andere einfache Arten, die Distanz zu messen.

[Zettel]
Auch die aktuell größten und genialsten Systeme sind am Ende Wahrscheinlichkeitsmaschinen und basieren auf Abständen in hochdimensionalen Räumen.

Wir beschränken uns zur Anschauung auf den zweidimensionalen Raum, das ist nicht nur in einem Buch zum Lernen leichter. Schau dir die beiden grünen Punkte im Bild an. Welchen Abstand haben die?



Grafische Darstellung der Abstandsmetriken

Die Linie, wo „Euklid" dransteht.

Das ist der Abstand zwischen den beiden Punkten.

Das ist die euklidische Distanz, das intuitivste Maß für den Abstand.

$$dist_{\text{Euklid}} = \sqrt{\Delta x^2 + \Delta y^2}$$

[Begriffsdefinition]
Die **euklidische Distanz** ist die Länge der direkten Verbindung zwischen zwei Punkten und kann mithilfe des pythagoreischen Lehrsatzes ausgerechnet werden.

Ah ja, das rechtwinklige Dreieck.

Genau, der Abstand ist die Hypotenuse, also das c in Pythagoras'.

$$a^2 + b^2 = c^2$$

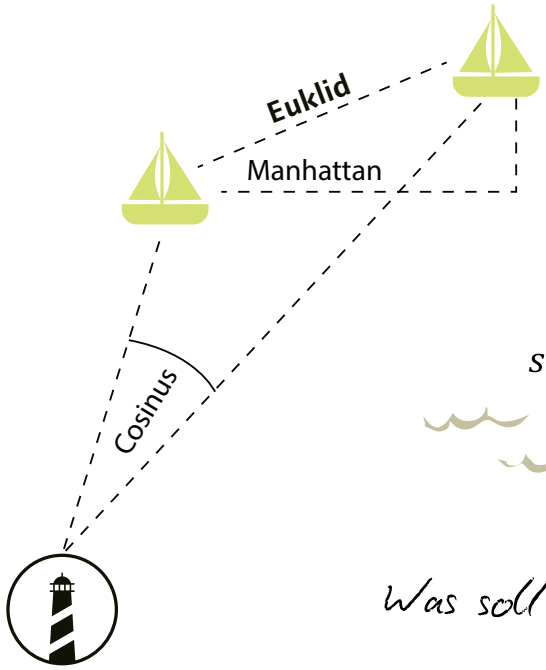
Jetzt stell dir vor, du bist in einer Stadt mit Straßen wie ein Schachbrettmuster, zum Beispiel in Mannheim oder Manhattan. Und du bist kein Vogel und darfst nur die Straßen entlang gehen. Dann sind im Bild die hellgrünen Linien der Weg zwischen den Punkten. Dann bekommst du die **Manhattan-Distanz**. Die ist viel einfacher zu berechnen: für jede Dimension den Abstand. Du summierst also lediglich die Einzelkomponenten auf.

$$dist_{\text{Manh}} = \Delta x + \Delta y$$

[Zettel]
Die euklidische Distanz hat den Nachteil, dass die Berechnung des Quadrats sowie die Berechnung der Wurzel relativ aufwendige Operationen für den Computer sind. Außerdem geht es sehr häufig nicht darum, eine exakte Distanz zu ermitteln, sondern nur darum, welche Elemente näher beisammen sind. Daher wird gerne die Manhattan-Distanz verwendet.

Hin und wieder geht es nicht um den Abstand, sondern die Richtung. Hierfür existiert die **Cosinus-Ähnlichkeit** (Cosine-Similarity), die beschreibt, ob sich die beiden Punkte in der gleichen Richtung befinden oder nicht.

Wenn ein Beobachter im Leuchtturm (am Ursprung) zwei Schiffe erblickt, sagt uns die Cosinus-Ähnlichkeit, wie »nah« die beiden Schiffe einander aus seiner Blickrichtung sind. Ein hoher Wert bedeutet, dass er sein Fernrohr nur wenig bewegen muss, um vom einen zum anderen Schiff zu schwenken. Ein niedriger Wert bedeutet, dass er es deutlich weiterbewegen muss.



$$sim_{\cos} = \cos(\theta) = \frac{a \cdot b}{\|a\| \cdot \|b\|}$$

Was soll denn das bedeuten?

Distanzmetriken vom Leuchtturm aus betrachtet

Ups, sorry. Also:

A und B sind die beiden Punkte. Im Zähler werden die x-Werte und y-Werte von den Punkten multipliziert, die Ergebnisse addiert. Im Nenner haben wir den euklidischen Abstand vom Nullpunkt zum Punkt, den wir betrachten. Ich schreibe dir nochmals mit den Details für die Punkte A und B hin:

$$sim_{\cos}(a, b) = \frac{a_x b_x + a_y b_y}{\sqrt{{a_x}^2 + {a_y}^2}\sqrt{{b_x}^2 + {b_y}^2}}$$

Oben im Zähler werden die Komponenten der Vektoren multipliziert und die Einzelwerte addiert und unten im Nenner werden die Abstände zum Nullpunkt (also die Längen der Vektoren) miteinander multipliziert.

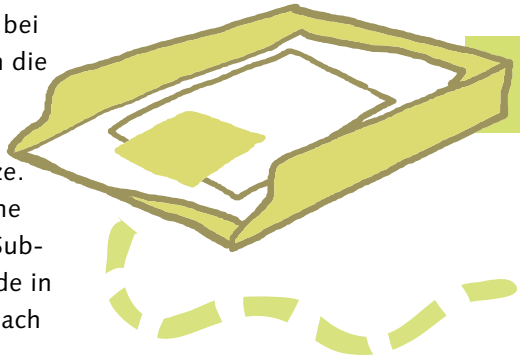
[Hintergrundinfo]

Die Cosinus-Ähnlichkeit wurde bereits in den 1960er-Jahren für die Ermittlung von Ähnlichkeiten zwischen Texten verwendet. Sie wurde in Suchmaschinen (damals »Information-Retrieval-Systeme«) eingesetzt.

Na ja, okay.

[Ablage]

Die Cosinus-Ähnlichkeit wird beispielsweise bei Bag-of-Words-Verfahren (damit beschäftigen wir uns später) und anderen Verfahren eingesetzt, bei denen es darum geht, ob zwei Vektoren unabhängig von ihrer Länge in die gleiche Richtung zeigen und somit eine ähnliche Bedeutung haben. Vereinfacht kannst du dir es so vorstellen: »Heute scheint die Sonne« und »Heute scheint die Sonne besonders stark« sind zwei ähnliche Sätze. Werden sie in Vektoren abgebildet, so zeigen diese in eine sehr ähnliche Richtung – in der Zeitachse zeigen beide auf den heutigen Tag, in der Subjekt-Achse beide in Richtung Sonne, in der Intensitätsachse zeigen beide in eine positive Richtung, wenn auch der eine länger ist als der andere. Nach der Cosinus-Ähnlichkeit sind sich diese beiden Vektoren sehr ähnlich.



Und jetzt halt dich fest: Das geht auch mit **sehr vielen Dimensionen**.
Wie angekündigt funktionieren die Metriken in vielen Dimensionen.

Die könnten wir nicht mehr mit Buchstaben wie x, y und z erfassen.

Warum nicht?

Schrödinger, es sind viel mehr als 26!

Also nennen wir sie x_i. Jedes i steht für eine Dimension.
Dann können wir die entsprechenden Formeln verallgemeinern.

Die Manhattan-Distanz in *N* Dimensionen

$$dist_{\text{Manh}} = \sum_{i=1}^{N} \Delta x_i$$

Also statt nur zwei Werte zu addieren, zeigt die Summe an, dass du die Werte für alle N Dimensionen aufaddierst.

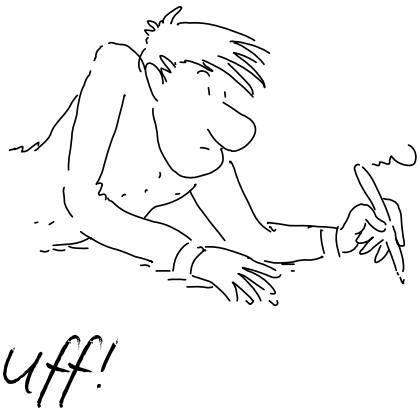
Die euklidische Distanz in *N* Dimensionen

$$dist_{\text{Euklid}} = \sqrt{\sum_{i=1}^{N} \Delta x_i^{\,2}}$$

Warte! Das kann ich!

Wunderbar!

Du hast verstanden, wie du die Ähnlichkeitsformeln verallgemeinerst.
Und jetzt die Cosinus-Ähnlichkeit.



$$sim_{\cos}(a,b) = \cos(\theta) = \frac{\sum_{i=1}^{N} a_i b_i}{\sqrt{\sum_{i=1}^{N} a_i^2}\sqrt{\sum_{i=1}^{N} {b_i}^2}}$$

Schaut wieder schlimmer aus, als es ist. Unten stehen die Längen der beiden Vektoren, also der Abstand vom Nullpunkt zum Punkt, auf den der jeweilige Vektor zeigt – genauer gesagt die euklidische Distanz –, und die werden wieder multipliziert. Im Zähler stehen die aufaddierten Produkte der Einzelkomponenten des Vektors, also die Werte der einzelnen Merkmale.

[Achtung]
Euklid und Manhattan sind **Distanzen**, die Cosinus-Ähnlichkeit ist wirklich eine **Ähnlichkeit**. Bei den Distanzen gilt: je größer, desto ungleicher. Bei den Ähnlichkeiten verhält es sich umgekehrt. Um die Cosinus-Distanz zu erhalten, rechnest du einfach 1 – Ähnlichkeit.

Und hier noch die Cosinus-Distanz für N Dimensionen

$$dist_{\cos}(a,b) = 1 - sim_{\cos}(a,b)$$

Während bei unseren folgenden Algorithmen erst einmal Manhattan und Euklid zum Einsatz kommen, ist die Cosinus-Distanz wichtig für **NLP** (**Natural Language Processing**) – wenn es also um Sprachen und Suchmaschinen geht. Das Beispiel mit Word2Vec kennst du ja bereits.

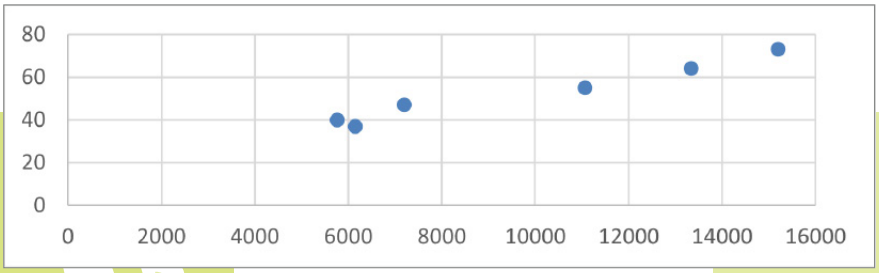
Tanz nicht aus der Reihe! – Normalisierung

Bei der Messung von Entfernungen müssen wir noch darauf achten, dass die Werte normalisiert sind.

Zwei Fragen:
wieso und wie?

Stell dir zwei Merkmale vor, die Länge eines Flugzeuges und die Reichweite. Die Länge wird in einigen Metern gemessen, während die Reichweite eines Flugzeuges Tausende Kilometer betragen kann. Wenn du dir die Vektoren im Raum vorstellst, dann ist der Einfluss der Flugzeuglänge minimal.

Ich habe hier sechs Flugzeuge für dich dargestellt. Die Länge beträgt 37 m bis 73 m. Die Reichweiten liegen jedoch zwischen 5765 km und 15200 km. Ändert sich die Länge um 10 %, dann macht das kaum einen Unterschied in der Positionierung der Punkte. Ändert sich jedoch die Reichweite um 10 %, dann sieht das Bild gleich ganz anders aus.



Viele Algorithmen reagieren auf derartige Unausgewogenheiten allergisch. Sie wollen Werte, die sich im gleichen Wertebereich aufhalten und nicht aus der Reihe tanzen. Deshalb **normalisiert** man die Daten.

[Achtung]

Eine wesentliche Aufgabe bei der Datenvorbereitung ist die **Normalisierung** der Daten. Die dient dazu, dass Algorithmen gleiche bzw. ähnliche Bedingungen bei allen Merkmalen vorfinden und nicht durch ein Ungleichgewicht bestimmten Werten in Merkmalsvektoren zu viel Bedeutung zukommen lassen, während sie andere Werte im Merkmalsvektor ignorieren.

[Begriffsdefinition]
Der **Z-Score** bezeichnet den normalisierten Wert eines Merkmals innerhalb eines Datensatzes und ist wie folgt definiert:

$$z = \frac{x - \bar{x}}{\sigma}$$

Wir benötigen also die Standardabweichung und den Durchschnitt der Werte des Merkmals. Anschließend gehen wir jeden Wert durch, subtrahieren den Durchschnitt und dividieren das Ergebnis durch die Standardabweichung des Merkmals.

Ja, die Standardabweichung war nochmal genau ...

[Begriffsdefinition]
Die **Standardabweichung** misst die Streuung und beschreibt die durchschnittliche Abweichung zum Mittelwert. Sie ist wie folgt definiert:

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

[Zettel]
Die Z-Werte sind Werte, deren Durchschnitt 0 und eine Standardabweichung von 1 ergeben, und zwar unabhängig von der Größe und Einheit der Originalwerte.

[Begriffsdefinition]
Der Z-Wert wird oftmals auch als **Standard-Scaler** bezeichnet.

Durch diesen Mechanismus hast du nun normalisierte Werte, mit denen Algorithmen besser arbeiten können, selbst wenn sie sich leicht ablenken lassen.

Wie ähnlich wir uns doch sind

Ganz kurz müssen wir uns noch mit den Abstandsmetriken beschäftigen, sei es rechnerisch – nur um sicherzugehen, du kannst das bestimmt schon – oder eben im Code. Denn wie gesagt: Nahezu alle KI-Systeme machen sich Vektoren und Abstände zunutze. Implementieren wir also ein paar kleine Funktionen, die die Metriken ausgeben.

Distanzen mit Pythagoras messen

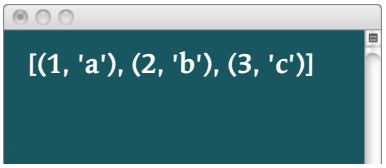
Euklidische Distanz

```
import math
def euclidean_distance(point1, point2):
  return math.sqrt(sum((x1 - x2)**2 for x1, x2 in zip*1(point1,
point2)))
```

*1 Ein Punkt hat mehrere Koordinaten und ist so abgebildet: `[x1, x2]`. Die Funktion `zip` kombiniert zwei Listen.

So funktioniert die zip-Funktion

```
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']
# Verwende zip, um die beiden Listen zu kombinieren
zipped = zip(list1, list2)*1
print(list(zipped))
```



[Einfache Aufgabe]
Was ist der Abstand zwischen den Punkten `p1(3,5)` und `p2(4,6)`?

[Erledigt!]

```
p1 = (3,5)
p2 = (4,6)
print(euclidean_distance(p1, p2))
```

Abstände in New York

Bei der **Manhattan-Distanz** werden lediglich die positiven Differenzen der Koordinaten gebildet und aufaddiert.

[Einfache Aufgabe]

Schreibe nun die entsprechende Funktion **`manhattan_distance`**.

[Erledigt!]

```
def manhattan_distance(point1, point2):
  return sum(abs(x1 – x2) for x1, x2 in zip(point1, point2))
```

Wenn wir die gleichen Punkte einsetzen, erhalten wir hier einen Abstand von 2.

Die Cosinus-Ähnlichkeit

Du erinnerst dich an die Cosinus-Ähnlichkeit, die lediglich prüft, ob die beiden Vektoren (der Pfeil von (0,0) zum Punkt) in die gleiche oder in eine ähnliche Richtung zeigen. Diese Ähnlichkeit ist kein Abstand, sondern umgekehrt kleiner, wenn die Richtung weiter auseinanderliegt. Man kann daraus aber den **Richtungsabstand** ermitteln. Der Abstand ist dann eins minus der Cosinus-Ähnlichkeit.

Übersetzen wir also die mathematische Formel in

Code für den Richtungsabstand

```
def cosine_similarity(vector1, vector2):
  dot_product = sum(x * y for x, y in zip(vector1, vector2))*1
  magnitude1 = math.sqrt(sum(x ** 2 for x in vector1))*2
  magnitude2 = math.sqrt(sum(y ** 2 for y in vector2))

  if magnitude1 == 0 or magnitude2 == 0:
    return 0.0*3

  return dot_product / (magnitude1 * magnitude2)
def cosine_distance(vector1, vector2):
  similarity = cosine_similarity(vector1, vector2)
  return 1 - similarity
```

*1 Das ist der Zähler in der Formel.

*2 Die beiden **magnitude**-Werte kommen dann in den Nenner.

*3 Die Ähnlichkeit ist 0, wenn einer der Vektoren ein Nullvektor ist.

Hier ergibt sich eine Distanz zwischen unseren beiden Punkten von 0,001.
Also zeigen diese Vektoren in die gleiche Richtung!

Unsere Funktionen funktionieren jetzt aber nur im 2D-Raum?

Bist du dir sicher? Probiere es aus:

```
p1 = (3, 5, 1)
p2 = (4, 6, 2)
```

```
Euklid: 1.732
Manhattan: 3
Cosinus: 0.006
```

Abstände in der Nussschale

Raucht dir schon der Kopf?

Kleine Zusammenfassung gefällig?

KI-Systeme, egal ob Algorithmen oder Modelle, basieren auf **Abständen** und der **Anordnung von Daten in einem mehrdimensionalen Raum**. Was wir daher benötigen, ist die Möglichkeit, Abstände zu messen. Und da der Computer gut mit Zahlen kann, wird alles in Zahlen umgewandelt. Mit Zahlen können wir gut Abstände messen.

Wir verwenden nun Abstände um Daten, die nahe beisammen sind, zu Gruppen zusammenzuführen. Das ist **Clustering**. Wir verwenden also Abstände, um neue Datensätze im Raum einzuordnen und auf Basis von bekannten Daten in diesem Raum, die nah an unserem Datenpunkt sind, Durchschnittswerte zu generieren und somit Vorhersagen zu machen. Das ist beispielsweise **Regression mit K-Nearest-Neighbor**.

Die Gesichtserkennung auf einem Smartphone macht nichts anderes als die Bilder, die der Rechner von dir bereits gesehen hat, so in einem Raum anzuordnen, dass alle Bilder von dir in einem engen Bereich sind, während Bilder von anderen Personen einen größeren Abstand haben. Ist der Abstand gering genug, so wird der Rechner entsperrt.

Der Computer ordnet ein Bild ein?

Nein, der Computer errechnet aus deinem Bild Merkmale, also wieder Vektoren, die entsprechend eingeordnet werden.

[Zettel]
Selbst Wörter werden in Vektoren umgerechnet, um anschließend Ähnlichkeiten zwischen Wörtern zu ermitteln.

Der Durchschnitts-Nachbar

Sehen wir uns den ersten konkreten Algorithmus zur Bildung von Clustern an: **K-Means**. Wenn du Daten – scheinbar ohne Bedeutung – hast, und diese in Bereiche zusammenfassen möchtest, dann bist du oftmals mit K-Means gut bedient.

Ich bitte um ein Beispiel.

Du hast Farbbilder (mit bis zu 16,7 Mio. unterschiedlichen Farben) und du möchtest diese Farben auf zum Beispiel 256 Farben reduzieren, um die Bilder stärker zu komprimieren. Dann stellt sich die Frage: Welche der 256 Farben willst du verwenden?

Du könntest den ganzen Farbraum in möglichst gleiche Teile aufteilen und dann entsprechend repräsentative Farben verwenden. Wahrscheinlich hast du aber einzelne Bereiche, die gar nicht vorkommen, und andere Farbbereiche ließen sich feiner aufteilen.

K-Means kann dir die Antwort liefern, welche Farben du verwenden sollst.

Egal, was du gruppieren möchtest, ob Dokumente, Farben, Kundensegmentierung oder Gene (Genexpressionsanalyse) – K-Means macht genau das: Daten gruppieren. Das nennt man auch »klassifizieren«.

Und woher weiß der Algorithmus, wie viele Gruppen erstellt werden sollen?

Erst einmal gar nicht. Das ist das K im K-Means, und das musst du festlegen. Bei der Farbreduktion der Bilder wäre K beispielsweise 256. Er würde dir damit 256 Cluster erstellen. Wenn du Kunden in 3 Segmente einteilen möchtest, dann verwendest du als K den Wert 3.

Der Ablauf des Algorithmus ist recht einfach.

1. Lege fest, wie viele Cluster du erstellen möchtest. Lege also das K fest.
2. Erzeuge K Zentroide (Stellvertreter), die du an zufällige Positionen setzt.
3. Weise jeden Datenpunkt dem Zentroiden zu, der den geringsten Abstand zum Datenpunkt aufweist.
4. Setze nun die Zentroide in das Zentrum der zugewiesenen Daten.
5. Wiederhole den Vorgang so oft du willst und brich ab, wenn sich nichts mehr ändert.

[Zettel]
Am einfachsten positionierst du die Zentroide, indem du für jede Dimension eine Zufallszahl zwischen dem Minimal- und Maximalwert dieser Dimension erzeugst.

[Zettel]
Das Ergebnis des K-Means-Algorithmus sind die K Stellvertreter (Zentroide), die die K Cluster repräsentieren. Außerdem kannst du mithilfe dieser Zentroide auch neue Datensätze den Clustern zuordnen. Du musst lediglich die Abstände zu den Zentroiden berechnen. Der neue Datenpunkt gehört dann zum Cluster mit dem geringsten Abstand zum Zentroiden.

Großartig, ich kann Gruppen generieren und neue Punkte Gruppen zuordnen.

[Begriffsdefinition]
Die Zentroide sitzen im Schwerpunkt des entsprechenden Clusters. Der Schwerpunkt ist nichts anderes als das Zentrum eines Clusters. Das Zentrum kannst du ermitteln, indem du einfach die Durchschnittswerte (Mittelwerte) für jede Dimension ermittelst.

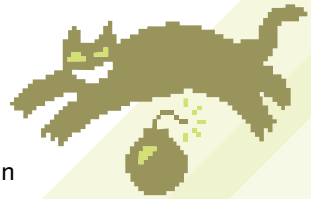
Aber warte. Wir starten mit irgendwelchen Zufallswerten.

Du fragst dich, wie da etwas Sinnvolles herauskommen kann?

Das passiert, indem du diese Zuordnung und Positionierung mehrfach durchführst: 10 bis 100 Mal.

Ist das garantiert das beste Ergebnis? Nein!

[Achtung]
Da K-Means mit zufälligen Positionen startet, kann der Algorithmus bei den gleichen Daten und bei mehrfacher Durchführung auch zu unterschiedlichen Cluster-Ergebnissen führen.



Dann weiß ich wieder nicht, ob das Ergebnis passt!

Ja, wir müssen das Ergebnis noch qualitativ überprüfen und unter Umständen den Algorithmus erneut durchlaufen lassen.

Wie wir diese Überprüfung durchführen können, zeige ich dir im nächsten Kapitel. Vorab möchte ich das Thema praktisch mit Code durchgehen und dir anschließend zeigen, was du tun kannst, wenn du dir nicht sicher bist, welches K du wählen sollst.

Das ist ja nett, aber was ist eigentlich genau das Ergebnis des Algorithmus?

[Zettel]
Das Ergebnis des K-Means-Algorithmus sind die Positionen der Zentroide. Mithilfe dieser Positionen können alle Datenpunkte und neue Datenpunkte den Gruppen zugeordnet werden. Somit sind mit den Positionen der Zentroide und mit der Abstandsmetrik die Cluster definiert.

[Achtung]
K-Means ist empfindlich gegenüber Skalierungsthemen. Deshalb solltest du die Werte vor der Anwendung des Algorithmus mit dem Z-Score standardisieren. Damit du später neue Datenpunkte korrekt zuordnen kannst, musst du zusätzlich zu den Clusterzentren auch den Mittelwert und die Standardabweichung der ursprünglichen Daten speichern. Diese beiden Parameter haben wir für die Standardisierung benötigt und du brauchst sie, um neue Werte auf die gleiche Weise zu standardisieren wie die ursprünglichen. Nur so ist eine konsistente Zuordnung möglich.

Dramaqueens und Sportskanonen

Ich möchte das Thema nochmal anhand eines Datensatzes mit dir durchgehen.
Ich zeige dir auch sofort das grafische Ergebnis und dann geht es ran an den Code.
Wir nehmen also unterschiedliche Tiere mit den Merkmalen Bewegungsdrang und dem Drama-Potenzial.

Tier	Drama-Potenzial	Bewegungs-drang	Begründung
Pfau	10	2	Präsentiert stolz sein Rad, steht aber meist nur rum.
Faultier	1	1	Null Stress, null Show – lächelt in Zeitlupe.
Zwergkaninchen	6	9	Springt bei jedem Rascheln in die Luft – Zoomies um 3 Uhr nachts!
Koala	3	2	Schläft einfach weiter – eukalyptushungriges Stativ.
Krake	8	7	Spritzt Tinte und entkommt aus Aquarien. 8 Arme = 8× Action!
Igel	5	4	Rollt sich bei Gefahr zur stacheligen Murmel – nächtlicher Snack-Läufer.
Ente	7	6	Quakt lautstark um Brotkrumen – watschelt, schwimmt, fliegt kurz.
Panda	4	3	Fällt gelegentlich vom Baum – Bambus kauen ist Sport.
Hyäne	9	8	Lacht hysterisch im Mondlicht – rennt Rudeln hinterher.
Erdmännchen	8	9	Steht auf zwei Beinen und schreit »GEFAHR!« – Buddel-Marathons.
Känguru	7	8	Boxt und hüpft durch die Gegend – immer in Bewegung.
Schildkröte	2	1	Langsam und gemütlich – null Drama, null Hektik.
Papagei	9	6	Plappert den ganzen Tag – fliegt und klettert viel.
Eichhörnchen	5	7	Sammelt Nüsse wie verrückt – flitzt durch die Bäume.
Flamingo	8	5	Steht elegant auf einem Bein – balanciert und watet gemächlich.

Wir wollen diese Tiere nun in 4 Cluster einteilen.

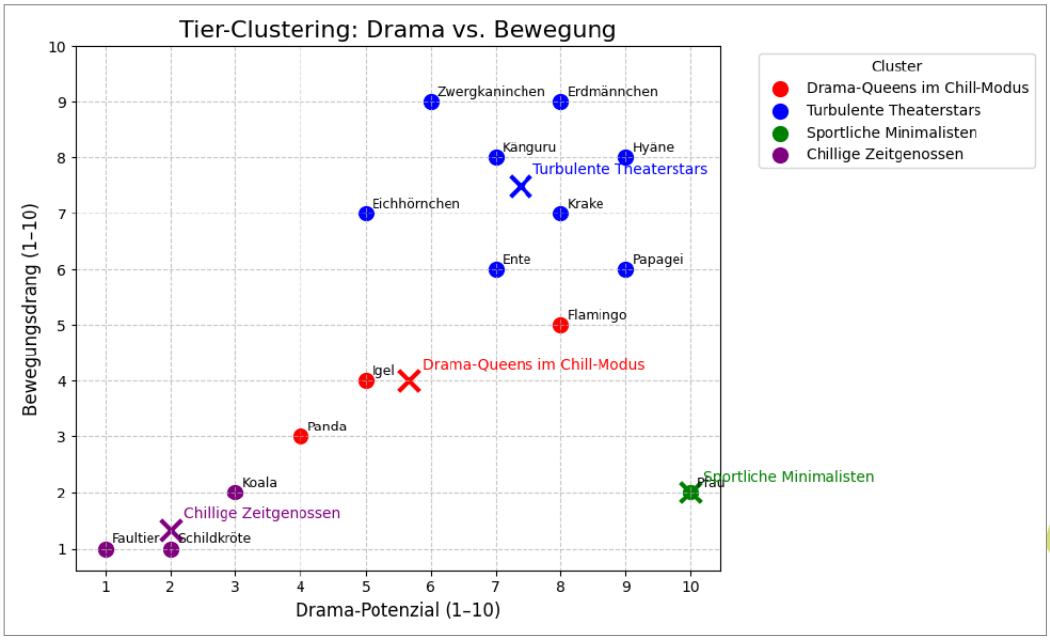
K ist 4. Ist meist.

Wenn nun der Algorithmus durchläuft, dann werden 4 zufällige Werte für das Drama-Potenzial und den Bewegungsdrang erstellt. Das sind die Werte unserer initialen Zentroide.

Anschließend weisen wir die Tiere dem räumlich nächsten Cluster zu und verschieben den Zentroiden des Clusters in den neuen Schwerpunkt. Der Cluster hat einen Zentroiden mit dem durchschnittlichen Bewegungsdrang und dem durchschnittlichen Drama-Potenzial der ihm zugewiesenen Tiere.

Anschließend weisen wir die Tiere wieder dem räumlich nächsten Zentroiden zu und berechnen erneut die Position der Zentroide mit den durchschnittlichen Bewegungsdrang- und Drama-Werten.

Am Ende haben wir die Tiere in 4 Cluster eingeteilt.



Zuordnung der Tiere in Cluster mit den entsprechenden Zentroiden

Es können sogar Werte abgesondert werden, wie in unserem Fall der Pfau, der einen eigenen Cluster bildet.

Des Pfau ist eben eine Klasse für sich.

Schwere Stellvertreter

Ich möchte mit dir kurz einen Teil eines berühmten Machine-Learning-Datensatzes verwenden, und zwar des Iris-Flower-Datensatzes. In diesem Datensatz befinden sich Messdaten verschiedener Orchideenblüten. Es wurden die Kelchblätter (Sepal) und die Blütenblätter (Petal) unterschiedlicher Spezies vermessen und aufgelistet. Es ist ein Datensatz mit wenigen Merkmalen und daher für unsere Visualisierungszwecke hier gut geeignet.

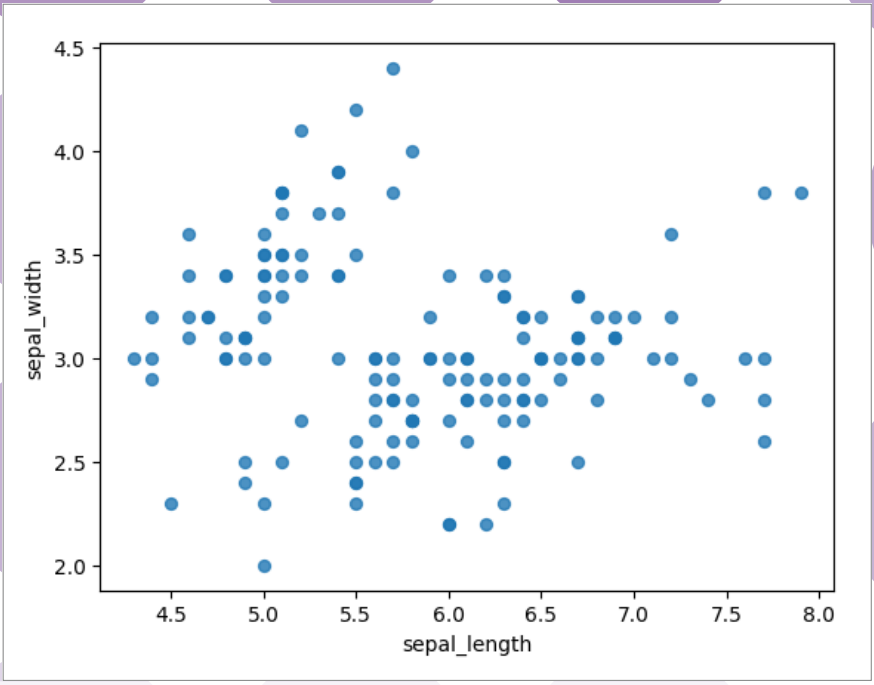
Wir werden den Datensatz in unterschiedlichen Schritten visualisieren, damit du genau siehst, was hier passiert. Damit es einfach bleibt in der Visualisierung, verwenden wir nur zwei Attribute.

Wenn ich ihn mit zwei Attributen verstehe, funktioniert das bestimmt auch mit 100 Attributen.

Genau! Dieser Datensatz hier besitzt allerdings erst einmal nur 4 Attribute insgesamt.

Wir verwenden nun K-Means mit zwei Clustern und den Attributen **`sepal_length`** und **`sepal_width`**.

```
from matplotlib import pyplot as plt
df.plot(kind='scatter', x='sepal_length', y='sepal_width', s=32, alpha=.8)
```



Darstellung der beiden Attribute »sepal_width« und »sepal_length« als Scatter-Plot

Die Visualisierung zeigt, wie die Messergebnisse verteilt sind. Nun kannst du dir überlegen, wie du selbst die Daten in zwei Gruppen einteilen würdest. Laut Algorithmus verwenden wir zufällige Positionen für die Zentroiden.

[Notiz]
Die Position der Zentroide soll irgendwo innerhalb der Daten sein, daher verwenden wir einen Zufallswert zwischen den Minimal- und Maximalwerten.

Fehlt uns hier nicht die Standardisierung der Werte?

[Achtung]
Du hast vollkommen Recht! Ich möchte dir jetzt erst einmal den Algorithmus und den Ablauf zeigen. Die Werte sind bei diesem Beispiel nicht so weit auseinander und es ergibt sich ein schöneres Bild, daher verzichten wir hier vorerst auf die Standardisierung.

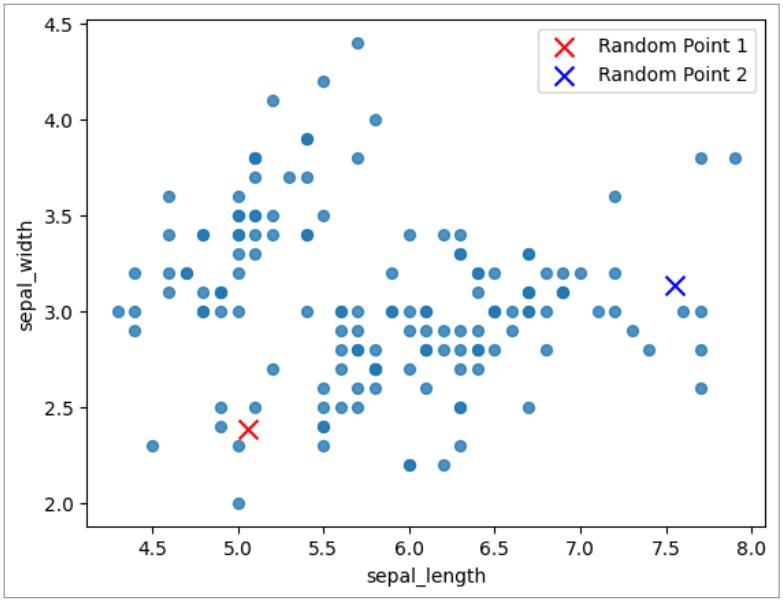
```
import random
sl_min = df['sepal_length'].min()
sl_max = df['sepal_length'].max()
sw_min = df['sepal_width'].min()
sw_max = df['sepal_width'].max()
centroid1_x = random.uniform(sl_min, sl_max)
centroid1_y = random.uniform(sw_min, sw_max)

centroid2_x = random.uniform(sl_min, sl_max)
centroid2_y = random.uniform(sw_min, sw_max)
```

Jetzt zeichnen wir uns das Diagramm neu – inklusive der Zentroide.

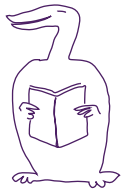
```
df.plot(kind='scatter', x='sepal_length', y='sepal_width', s=32, alpha=.8)
```

Die Zentroide zum Plot hinzufügen

```
plt.scatter( centroid1_x,  centroid1_y, color='red', marker='x',
s=100, label='Zentroid 1')
plt.scatter( centroid2_x,  centroid2_x, color='blue', marker='x',
s=100, label='Zentroid 2')
plt.legend()
plt.show()
```



Der Datensatz mit den Zentroiden



Der nächste Schritt im Algorithmus ist die Zuordnung jedes Datensatzes zum entsprechenden Zentroiden. Hierfür verwenden wir **numpy**. Das ist eine beliebte Python-Library mit unzähligen nützlichen Datenstrukturen und Funktionen für Berechnungen.

```
import numpy as np
df['distance_to_centroid1'] = np.sqrt((df['sepal_length'] -
centroid1_x)**2 + (df['sepal_width'] - centroid1_y)**2)*2
df['distance_to_centroid2'] = np.sqrt((df['sepal_length'] -
centroid2_x)**2 + (df['sepal_width'] - centroid2_y)**2)
```

Es werden zwei Spalten zum Pandas-DataFrame hinzugefügt: die Distanz zum ersten Zentroiden und die Distanz vom Datenpunkt zum zweiten Zentroiden.

[Einfache Aufgabe]
Sieh dir den Code genau an. Welche Distanzmetrik wird verwendet?

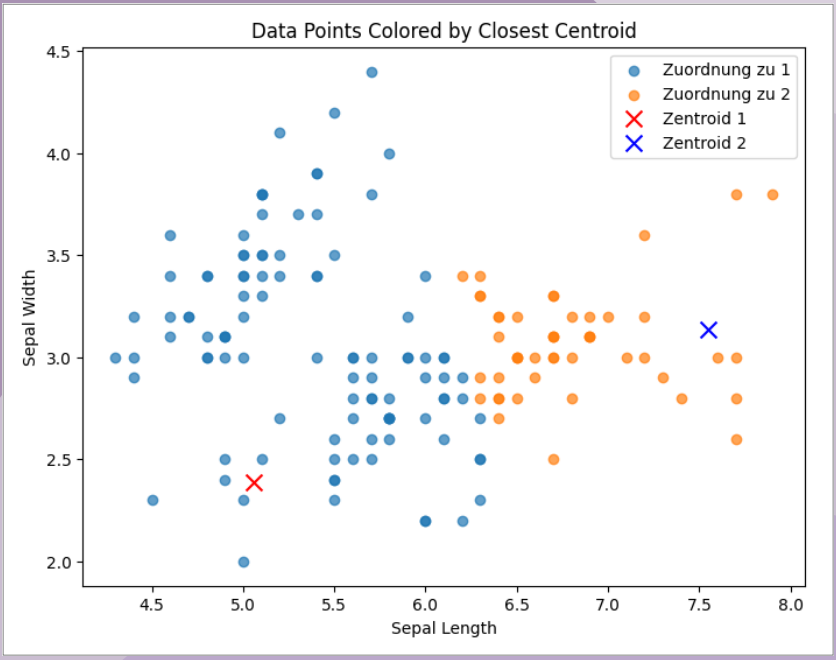
Wurzel ... X-Quadrat ...
das ist der Euklid!

Wunderbar, du hast das vollkommen richtig erkannt.
Wir erstellen eine neue Pseudospalte, in der der Zentroid steht.

In der Spalte **closest_centroid** steht nun entweder 1 oder 2, je nachdem, welcher Zentroid dem Datensatz näher ist.

```
df['closest_centroid'] = np.where(df['distance_to_centroid1']
< df['distance_to_centroid2'], 1, 2)
```


```
plt.figure(figsize=(8, 6))
for centroid in [1, 2]:
  subset = df[df['closest_centroid'] == centroid]
  plt.scatter(subset['sepal_length'],
subset['sepal_width'], label=f'Zuordnung zu {centroid}', alpha=0.7)

plt.scatter(centroid1_x, centroid1_y, color='red', marker='x', s=100,
label='Zentroid 1')
plt.scatter(centroid2_x, centroid2_y, color='blue', marker='x', s=100,
label='Zentroid 2')

plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.title('Data Points Colored by Closest Centroid')
plt.legend()
plt.show()
```



Initialzuordnung der Datenpunkte zu den Zentroiden

Durch die zufällig gewählten Startpositionen ergeben sich nun die Zuordnungen in der Abbildung. Offensichtlich sind die Zentroide aber nicht im Schwerpunkt der Punkte.

Also auf zum nächsten Schritt:
die Zentroide in die Schwerpunkte verschieben.

Wir berechnen uns also den Durchschnittswert der X-Werte (**sepal_length**) von den Daten, die dem ersten Zentroiden zugeordnet sind, und setzen den entsprechenden X-Wert. Gleiches wird mit den Y-Werten (**sepal_width**) gemacht und anschließend wiederholen wir das Szenario für den zweiten Zentroiden.

```
new_centroid1_x = df[df['closest_centroid'] == 1]['sepal_length'].mean()
new_centroid1_y = df[df['closest_centroid'] == 1]['sepal_width'].mean()
```

Wir selektieren nur die Daten des ersten Zentroiden und verwenden die Werte der **sepal_length**. Davon wird der Mittelwert berechnet.

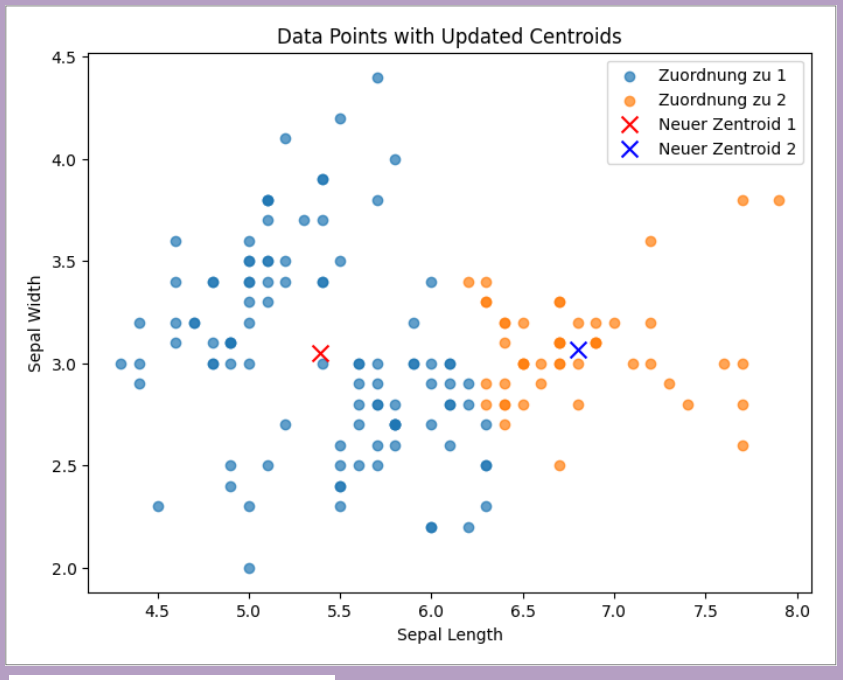
```
new_centroid2_x = df[df['closest_centroid'] == 2]['sepal_length'].mean()
new_centroid2_y = df[df['closest_centroid'] == 2]['sepal_width'].mean()
```

[Code bearbeiten]

Zeichne doch nun selbst die Grafik mit den neuen Zentroid-Werten.

[Erledigt!]

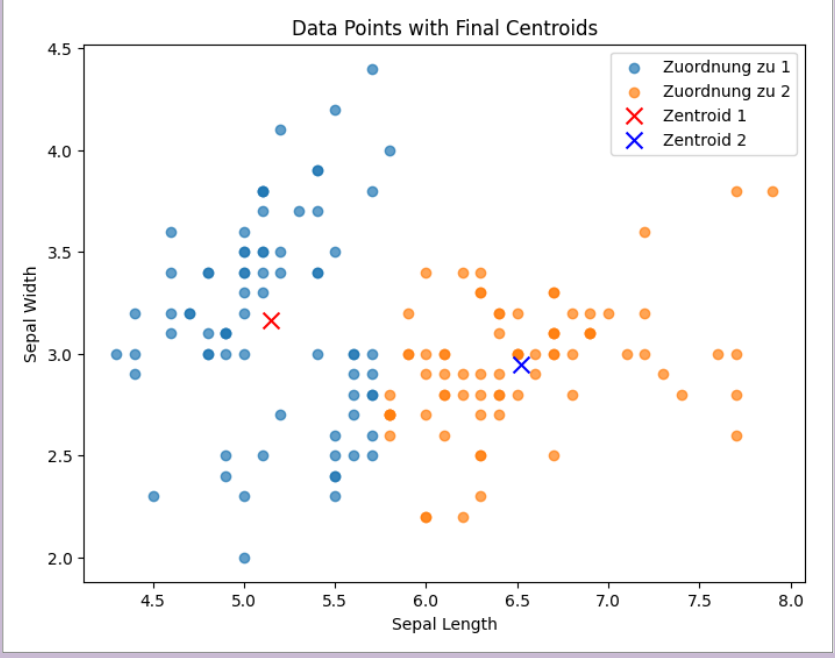
```
plt.figure(figsize=(8, 6))
for centroid in [1, 2]:
  subset = df[df['closest_centroid'] == centroid]
  plt.scatter(subset['sepal_length'], subset['sepal_width'],
label=f'Zuordnung zu {centroid}', alpha=0.7)
plt.scatter(new_centroid1_x, new_centroid1_y, color='red',
            marker='x', s=100, label='Neuer Zentroid 1')
plt.scatter(new_centroid2_x, new_centroid2_y, color='blue',
            marker='x', s=100, label='Neuer Zentroid 2')
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.title('Data Points with Updated Centroids')
plt.legend()
plt.show()
```



Neupositionierung der Zentroide

Nun stimmen die Zuordnungen allerdings nicht mehr, da wir die Zentroide verschoben haben. Wir ordnen die Datenpunkte daher den neuen Zentroiden zu.
Sobald sich etwas ändert, sitzen die Zentroide wieder nicht im Schwerpunkt und dieser Schritt wird wiederholt – und so geht es weiter …

Nach 10 Durchläufen ergeben sich folgende Zentroide und Zuordnungen:



K-Means-Clustering nach 10 Durchläufen

Hättest du auch diese Cluster gebildet?

Ehm, nein!
Ich hätte wohl eher die Punkte links oben und den Rest zusammengefasst.

[Achtung]

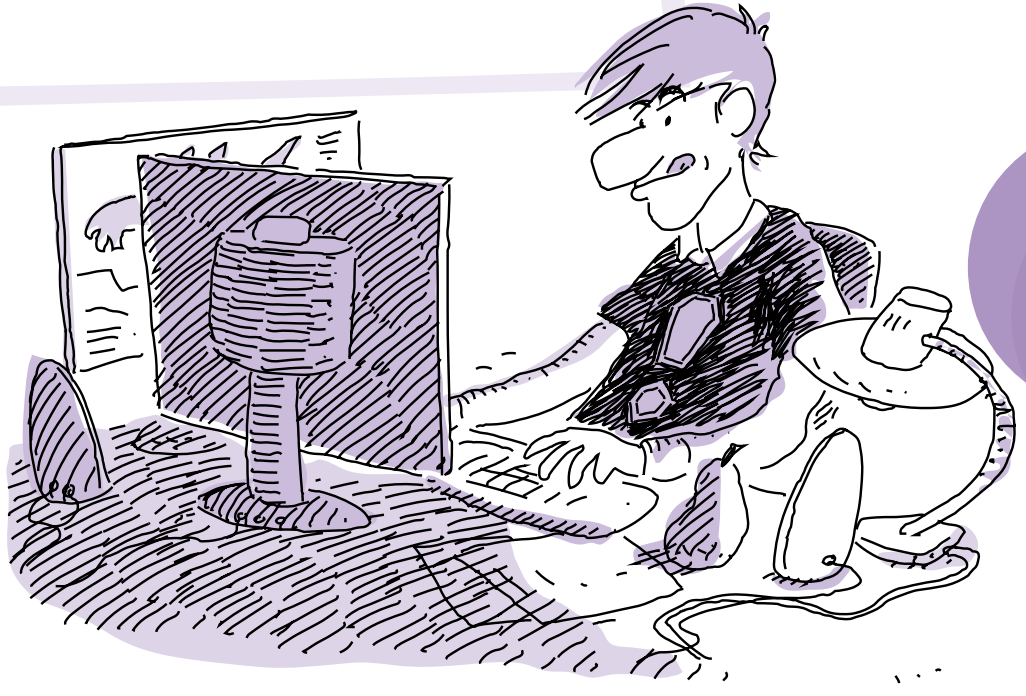
Aufgrund der zufälligen Startpunkte der Zentroide können sich bei dir andere Cluster ergeben, wenn du den Code ausführst. Das ist nicht falsch, sondern liegt einfach an der Art und Weise, wie dieser Algorithmus funktioniert.

[Schwierige Aufgabe]
Implementiere den Prozess nun so, dass er automatisch 10 oder 100 Mal in einer Schleife die Zentroide anpasst. Am Ende soll die angepasste Plot-Ausgabe erscheinen.

Schritt 1: Initialisierung der zufälligen Zentroide

*1 Eine kleine Hilfsfunktion für die Distanzberechnung mit dem Euklid

```
zentroid1_x = random.uniform(df['sepal_length'].min(), df['sepal_length'].max())
zentroid1_y = random.uniform(df['sepal_width'].min(), df['sepal_width'].max())
zentroid2_x = random.uniform(df['sepal_length'].min(), df['sepal_length'].max())
zentroid2_y = random.uniform(df['sepal_width'].min(), df['sepal_width'].max())

def euclid(x1, y1, x2, y2):*1
    return np.sqrt((x1 - x2)**2 + (y1 - y2)**2)
```



Schritt 2 und 3: Zuordnung Neu-Positionierung

*1 Eine Schleife für die Anzahl der Durchläufe

*2 Eine Lambda-Funktion, die für jede Zeile aufgerufen wird, um die Distanz zwischen dem Datensatz und dem Zentroid zu ermitteln

*3 Anwenden der Distanzberechnung und Speichern der Ergebnisse in einer neuen Spalte

```
for _ in range(10):*1
  dist_func_1 = lambda row*2: euclid(row['sepal_length'], row['sepal_width'],
zentroid1_x, zentroid1_y)
  dist_func_2 = lambda row: euclid(row['sepal_length'], row['sepal_width'],
zentroid2_x, zentroid2_y)
  df['dist_to_1'] = df.apply(dist_func_1, axis=1)*3
  df['dist_to_2'] = df.apply(dist_func_2, axis=1)

  df['closest_centroid'] = np.where(df['dist_to_1'] < df['dist_to_2'], 1, 2)*4

  new_centroid1_x = df[df['closest_centroid'] == 1]['sepal_length'].mean()*5
  new_centroid1_y = df[df['closest_centroid'] == 1]['sepal_width'].mean()
  new_centroid2_x = df[df['closest_centroid'] == 2]['sepal_length'].mean()
  new_centroid2_y = df[df['closest_centroid'] == 2]['sepal_width'].mean()

  if abs(new_centroid1_x- zentroid1_x) < 0.01 and
    abs(new_centroid1_y - zentroid1_y) < 0.01 and
    abs(new_centroid2_x - zentroid2_x) < 0.01 and
    abs(new_centroid2_y - zentroid2_y) < 0.01 :*6
      break

  zentroid1_x = new_centroid1_x*7
  zentroid1_y = new_centroid1_y
  zentroid2_x = new_centroid2_x
  zentroid2_y = new_centroid2_y
```

*4 Die Ermittlung des nächsten Zentroiden – und den Wert 1 oder eben 2 in der Spalte `closest_centroid` merken

*5 Die neuen Schwerpunkte für die Positionierung berechnen

*6 Eine optionale Zusatzprüfung: Ist der Algorithmus genug zu einem bestimmten Wert konvergiert? Wenn sich die Zentroide kaum noch bewegen, kann abgebrochen werden.

*7 Repositionierung der Zentroide

Nach der Schleife kannst du das Diagramm noch ausgeben.

```
plt.figure(figsize=(8, 6))
for centroid in [1, 2]:
    subset = df[df['closest_centroid'] == centroid]
    plt.scatter(subset['sepal_length'],
subset['sepal_width'], label=f'Zuordnung zu
{centroid}', alpha=0.7)

plt.scatter(zentroid1_x, zentroid1_y, color='red',
            marker='x', s=100,
label='Zentroid 1')
plt.scatter(zentroid2_x, zentroid2_y, color='blue',
            marker='x', s=100,
label='Zentroid 2')

plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.title('Data Points with Final Centroids')
plt.legend()
plt.show()
```


[Code bearbeiten]

Und nun, lieber Schrödinger, sollten wir vielleicht doch noch die Standardisierung mit dem Z-Score einbauen, dann haben wir es richtig gemacht.

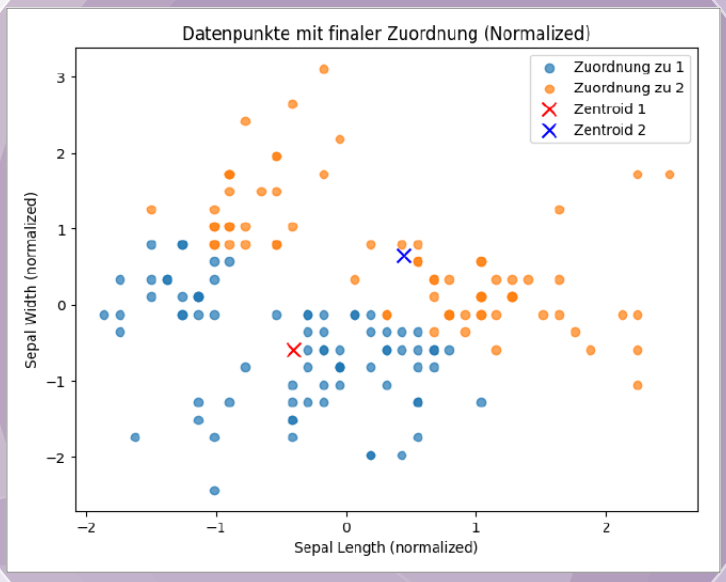
[Erledigt!]

```
for column in ['sepal_length', 'sepal_width']:
    df[column] = (df[column] - df[column].mean()) /
    df[column].std()
```

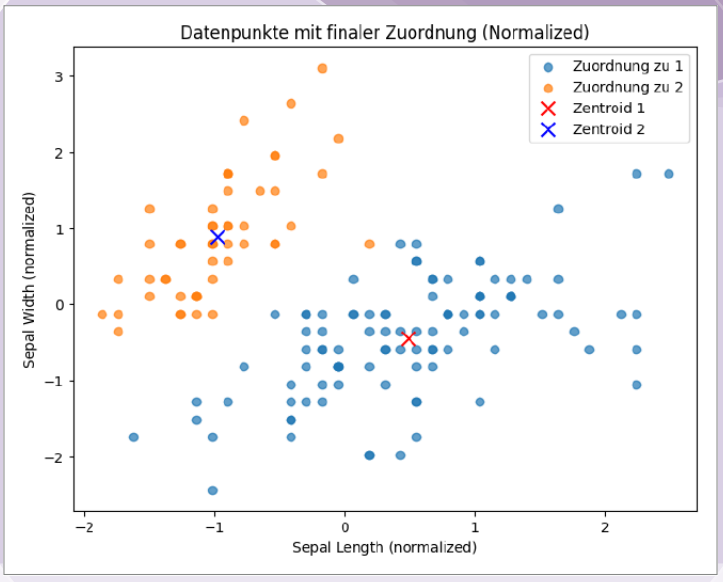

[Achtung]

Die Standardisierung ist normalerweise der allererste Schritt, noch bevor du die Zentroide erstellst.

Vergiss nicht, dass der Algorithmus unterschiedliche Clusterergebnisse liefern kann. Ich habe hier zwei Abbildungen mit 100 Durchläufen und den standardisierten Werten, dennoch kommen zwei ganz unterschiedliche Cluster heraus. Welches der beiden Ergebnisse besser ist, sehen wir uns im nächsten Kapitel über die Clusteranalyse an.



100 Durchläufe und entsprechendes Clustering



Wiederum 100 Durchläufe, allerdings eine komplett andere Clusterbildung

Okay, verstanden.

Nun kannst du diesen Algorithmus noch erweitern, sodass er nicht nur mit zwei, sondern mit beliebig vielen Dimensionen arbeitet.

Das gibt es doch bestimmt schon fertig.

Natürlich. Da du nun weißt, wie dieser Algorithmus funktioniert, kannst du dir, wenn du willst, die Arbeit sparen und eine fertige Library verwenden. Dann lass uns das Gelernte lieber noch etwas festigen mit der SKLearn-Library.

Orchideentypen

[Notebook]
Den Code für den folgenden Abschnitt findest du unter **Kapitel 2/02-kmeans-iris-sklearn.ipynb**.

Nachdem du nun den K-Means-Algorithmus mit allen Facetten selbst implementieren kannst, zeige ich dir jetzt, wie du das mit einer Library umsetzen kannst. Und wie so oft beim maschinellen Lernen ist die Datenvorbereitung für die Library der Vorgang, der den meisten Aufwand erzeugt.

Verwenden wir doch den gleichen Datensatz wie bisher – nur, dass wir nicht zwei, sondern drei Cluster erzeugen und alle vier Merkmale verwenden, die unser Datensatz hergibt.

[Zettel]
Der Originaldatensatz beinhaltet auch die entsprechenden Label für die Blütenklassifizierung. Daher wissen wir bereits, dass es drei unterscheidbare Iris-Arten sein sollten.

```
from sklearn.cluster import KMeans*1
from sklearn.preprocessing import StandardScaler*2
path = kagglehub.dataset_download("smritisingh1997/species-segmentation-using-iris-dataset")
filename = path + '/iris-dataset.csv'
df = pd.read_csv(filename)

features = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width']*3
X*4 = df[features]
```

*1 Schön, die Implementierung heißt wie der Algorithmus.

*2 Dieser Scaler ist der Vorbereitungsschritt für die Standardisierung mit dem Z-Score.

*3 Das hier sind die Merkmale/Spalten, die wir betrachten wollen. In unserem Fall sind es alle vier.

*4 In Machine-Learning-Algorithmen werden die Merkmale immer mit **X** benannt, die Labels mit **Y**. Das kommt wohl aus der Statistik.

```
# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)*5

kmeans = KMeans(n_clusters=3*6, random_state=0*7)
df['cluster'] = kmeans.fit_predict(X_scaled)*8

print(df.head())
```

*5 Skalierung durchführen.

*6 Durchführen des K-Means-Algorithmus mit 3 Clustern.

*7 Der Wert 0 bedeutet, dass an zufälligen Positionen gestartet wird. Wenn du hier einen beliebigen anderen Wert einsetzt, wird das als Seed für den Zufallsgenerator verwendet.

*8 Die bestehenden Daten werden nun den Clustern zugeordnet.

[Zettel]
Zufallszahlengeneratoren in der Informatik liefern keine echten Zufallswerte, sondern eine möglichst lange Sequenz, die sich zwar irgendwann wiederholt, aber sehr zufällig wirkt. Der **Seed** ist eine Art Startpunkt dieser Sequenz. Durch die Festlegung des Startpunktes wird der Zufall reproduzierbar, denn er legt die Reihenfolge fest. Das klingt paradox, wird aber immer wieder benötigt, um reproduzierbare Ergebnisse zu erhalten. Wird der Seed dem Zufall überlassen, so wird ein Wert von der aktuellen Uhrzeit, Hardware etc. abgeleitet – also von Faktoren, die sich immer wieder ändern.

Langsam glaube ich, die KI-Typen können gar nicht programmieren.

Das hast du jetzt gesagt. Aber es stimmt schon, dass hier sehr viel weniger Know-how zum Thema Softwareentwicklung benötigt wird als bei der klassischen Softwareentwicklung, da nahezu jeder Algorithmus als Library verfügbar ist. Dafür ist im Bereich des maschinellen Lernens viel mehr Statistik und Mathematik erforderlich.

Bestimmt fragst du dich, wie du nun die Ergebnisse – also die Skalierungs-Parameter und die Positionen der Zentroide erhältst.

Jetzt, wo du es sagst ...

```
scaler_params = {
  'mean': scaler.mean_,
  'scale': scaler.scale_
}
print('Das K-Means-Modell')
print('Scaler-Parameter:', scaler_params)
print('Zentroide:', kmeans.cluster_centers_)
```

```
Das K-Means-Modell
Scaler-Parameter: {'mean': array([5.84333333, 3.054, 3.75866667,
1.19866667]), 'scale': array([0.82530129, 0.43214658, 1.75852918,
0.76061262])}
Zentroide: [[-0.07723421 -0.92778421  0.32291983  0.23786769]
 [-1.01457897  0.84230679 -1.30487835 -1.25512862]
 [ 1.06889068  0.06560955  0.9690362   1.00197871]]
```

[Zettel]
Die Mittelwerte und Standardabweichungen, die für die Standardisierung verwendet wurden, existieren für jede Dimension.

Mit der Library **`pickle`** kannst du beispielsweise Werte laden und speichern.

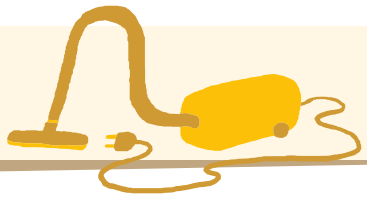
```
import pickle
scaler_params = {
  'mean': scaler.mean_,
  'scale': scaler.scale_
}
with open('scaler_params.pkl', 'wb'*1) as f:
  pickle.dump(scaler_params, f)
```

*1 Die Datei schreibend öffnen

```
# Laden der Werte aus der Datei
with open('scaler_params.pkl', 'rb'*2) as f:
  scaler_params = pickle.load(f)
# Neuen Scaler erstellen und Parameter setzen
new_scaler = StandardScaler()
new_scaler.mean_ = scaler_params['mean']
new_scaler.scale_ = scaler_params['scale']
```

*2 Die Datei lesend öffnen

[Einfache Aufgabe]
Implementiere das Speichern und Laden für die Cluster-Zentroide.



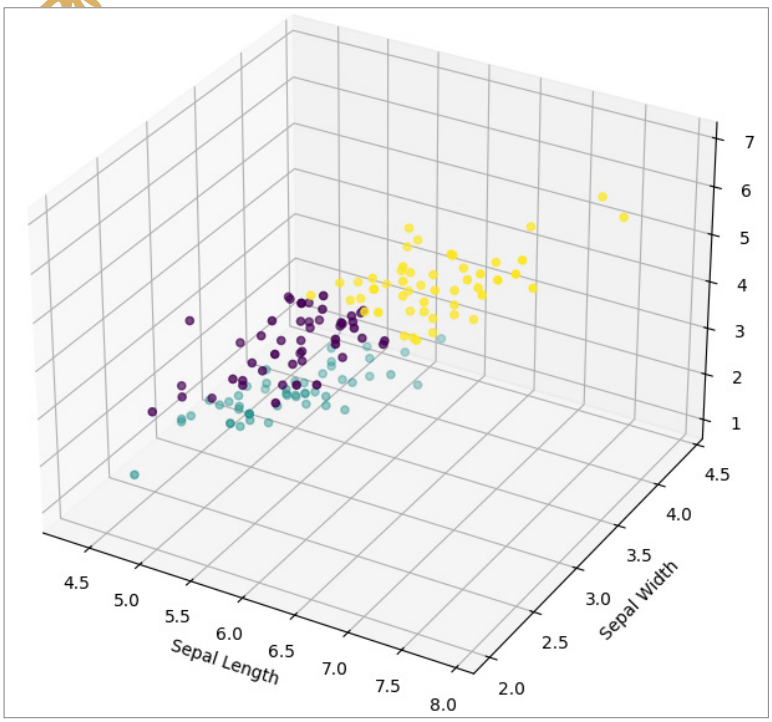
```
# Cluster-Zentren schreiben
with open('cluster_centers.pkl', 'wb') as f:
  pickle.dump(kmeans.cluster_centers_, f)

# Cluster-Zentren laden
with open('cluster_centers.pkl', 'rb') as f:
  cluster_centers = pickle.load(f)

# Neues K-Means-Modell erstellen und Cluster-Zentren setzen
new_kmeans = KMeans(n_clusters=3, random_state=0)
new_kmeans.cluster_centers_ = cluster_centers
```

[Lösung]

3D-Visualisierung ist wunderbar, aber sei dir bewusst, dass eine (möglicherweise wesentliche) Dimension fehlt.

[Schwierige Aufgabe]
Visualisiere mithilfe eines 3D-Scatter-Plots die Merkmale **`sepal_length`**, **`sepal_width`** und **`petal_length`**. Die Farbe **`c`** soll je nach Cluster unterschiedlich sein. Blättere zurück zur Seite 51, auf der wir gemeinsam ein 3D-Diagramm erstellt haben, und passe den Code entsprechend an.

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

fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

ax.scatter(df['sepal_length'], df['sepal_width'], df['petal_length'],
c=df['cluster'], cmap='viridis')
ax.set_xlabel('Sepal Length')
ax.set_ylabel('Sepal Width')
ax.set_zlabel('Petal Length')
plt.show()
```



Darstellung der Features in einem 3D-Scatter-Diagramm

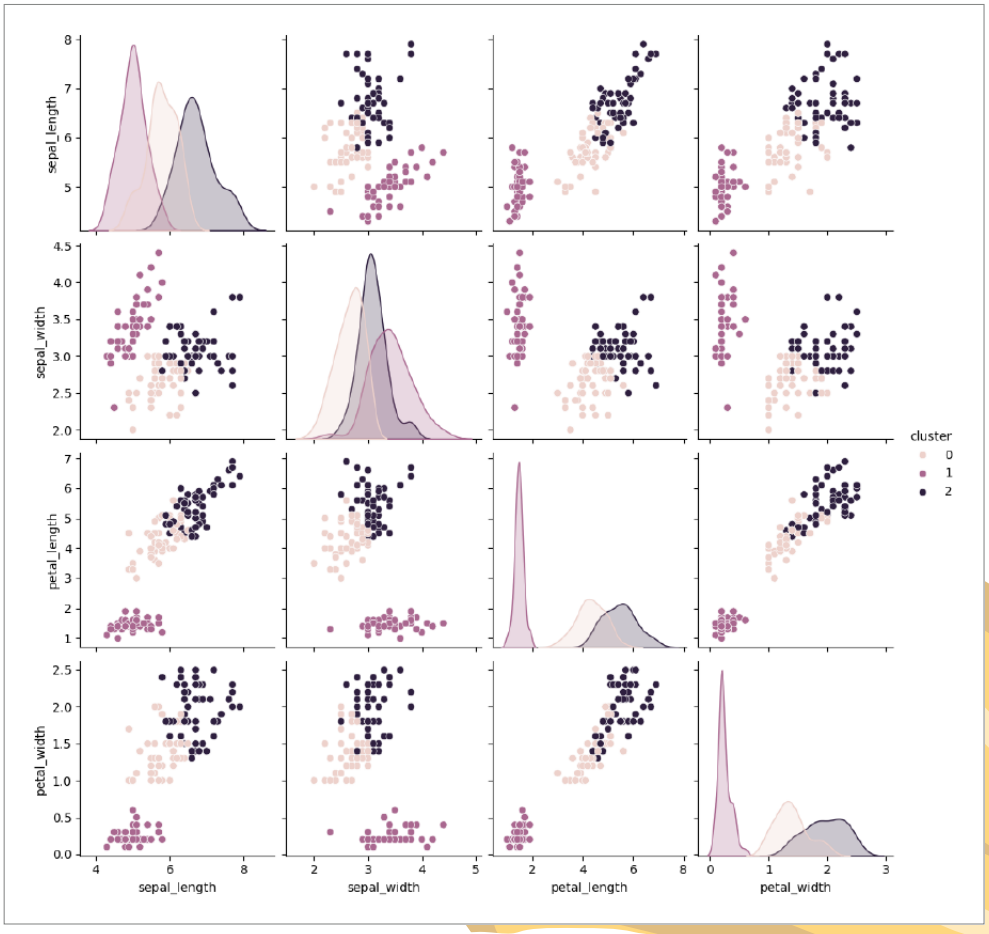
Im Diagramm siehst du, dass die Elemente im gelben Cluster besser abgetrennt sind als die anderen beiden Cluster. Nun jedoch alle Elemente und Kombinationen durchzugehen, das wäre schon etwas aufwendig.

Jetzt zeige ich dir einen Mega-Geheimtrick zur Visualisierung.

[Notiz]

Die Library **seaborn** zeigt dir gleich alle möglichen Zusammenhänge und Diagramme der Merkmalsaufteilungen.

```
import matplotlib.pyplot as plt
import seaborn as sns

sns.pairplot(df, hue='cluster', vars=features)
plt.show()
```

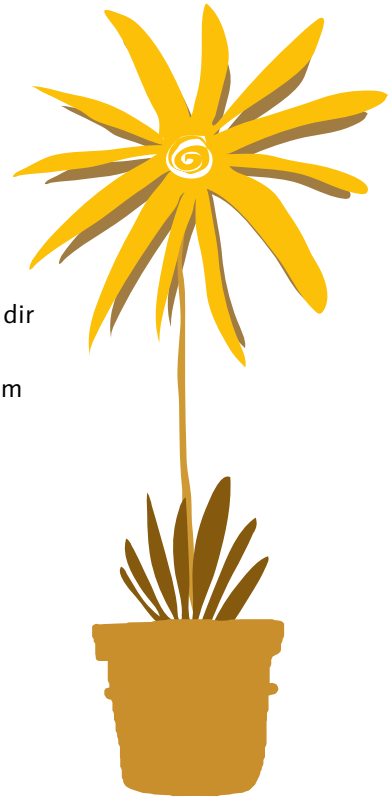


Darstellung der Zusammenhänge der unterschiedlichen Dimensionen

Wie cool ist das denn?!

Egal, welche Merkmale du dir in den Diagrammen ansiehst – der eine Cluster ist in jeder Dimension sehr gut differenzierbar. Die anderen beiden sind schwer eindeutig zu trennen.

[Einfache Aufgabe]
Welche beiden Diagramme zeigen dir noch die beste Trennung zwischen den beiden hellen Clustern und dem dunklen?



[Lösung]
`petal_length` und **`sepal_width`** sowie **`sepal_width`** und **`petal_width`**

Bei diesen Diagrammen kannst du noch am ehesten eine Trennlinie ziehen. Auch **`sepal_width`** und **`petal_length`** geht noch ziemlich gut. Aber fehlerlos kannst du kaum eine gerade Linie ziehen.

Du willst offenbar dringend weiter zur Clusteranalyse kommen, aber erst zeige ich dir noch zwei weitere Algorithmen, einerseits zum Clustern und andererseits zur Klassifizierung. Ein Nachteil von K-Means ist nämlich die Sensitivität gegenüber Ausreißern – also Datenpunkten, die weit abgeschlagen sind. Der folgende Algorithmus DBScan hat diesen Nachteil nicht, er stellt diese Ausreißer sogar explizit heraus.

Dicke Freunde

Zum Clustern und zum Erkennen von Ausreißern ist der **DBScan**-Algorithmus eine gute Wahl.

Ein Datenbank-Algorithmus?

Nein, DB steht nicht für Datenbank, sondern für **Density Based** – also dichtebasiert. Dieser Algorithmus bildet Cluster von Datenpunkten, die eine bestimmte Dichte aufweisen, und wurde 1996 veröffentlicht.

Es ist hier auch nicht erforderlich, dass du die Anzahl der Cluster vorab festlegst. Die Anzahl ergibt sich automatisch durch den Algorithmus. Ganz ohne Parameter geht es jedoch auch nicht. Wir benötigen hier zwei Werte: eine Mindestanzahl von Datenpunkten (`MinPts`) und einen Radius ε (sprich: Epsilon), der den Bereich definiert, der analysiert wird.

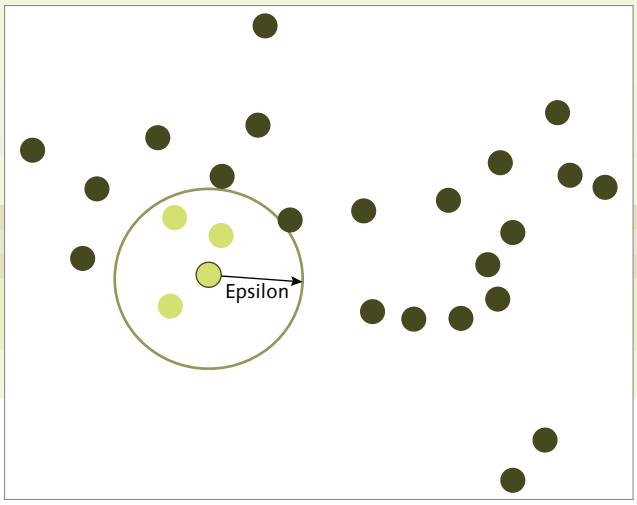
Der Algorithmus ordnet die Punkte in drei Kategorien ein:

1. **Kernpunkte**: Punkte, die mindestens `MinPts` Nachbarn innerhalb eines Radius von ε haben.
2. **Randpunkte**: Punkte, die innerhalb des ε-Radius eines Kernpunkts liegen, aber selbst nicht genügend Nachbarn haben, um Kernpunkte zu sein.
3. **Rauschpunkte**: Punkte, die weder Kernpunkte noch Randpunkte sind und somit als Ausreißer betrachtet werden.

Kernpunkte, Randpunkte?
Erklär mir erst mal, wie der Algorithmus funktioniert, dann komm ich bestimmt noch drauf.

Der Algorithmus hat folgende Schritte:

1. Wähle einen unbesuchten Punkt aus deinem Datensatz aus.
2. Bestimmte die ε-Nachbarschaft des Punktes – welche Punkte sind innerhalb des Radius ε vorhanden?
3. Wenn die Anzahl der gefundenen Nachbarn größer oder gleich den `MinPts` ist, markiere den Punkt als Kernpunkt und erstelle einen neuen Cluster.
4. Füge alle Punkte in der ε-Nachbarschaft zum Cluster hinzu.
5. Wiederhole den Prozess für jeden Punkt im Cluster, bis keine neuen Punkte mehr hinzugefügt werden können.
6. Wiederhole die Schritte 1–5, bis kein Punkt mehr unbesucht ist.
7. Alle Punkte, die zu wenige Nachbarn haben, um einen Cluster zu bilden, sind Rauschpunkte – also Ausreißer.

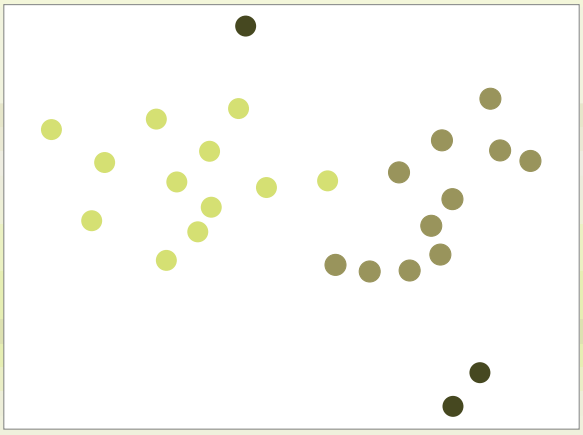


Konzept des DBScan-Algorithmus

In der Abbildung siehst du, wie ein Punkt ausgewählt und die Anzahl der Nachbarn ermittelt wurde, die nicht weiter als ε davon entfernt sind. All diese Nachbarn werden dem Cluster (dem blauen Kreis) zugeordnet, sofern mindestens `MinPts` Nachbarn vorhanden sind.

Anschließend wird jeder dieser Punkte als neues Zentrum mit dem gleichen ε-Wert analysiert und so der Cluster schrittweise erweitert. Wenn nicht genügend Nachbarn vorhanden sind, wird der Punkt nicht dem Cluster zugeordnet.

Wenn der Algorithmus durchgelaufen ist, ergeben sich beispielsweise `folgende Cluster:`

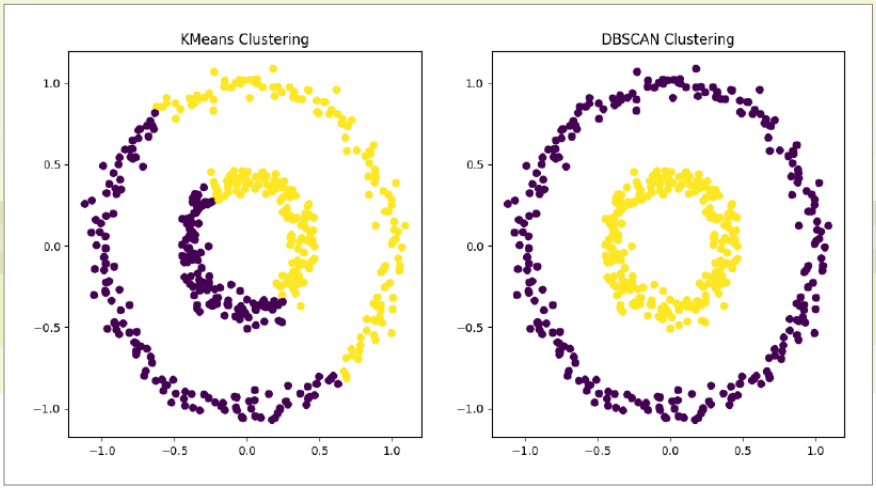


Clusterergebnis des DBScan-Algorithmus.
Es wurden zwei Cluster und drei Ausreißer gefunden.

Dieser Algorithmus hat wieder wenige **einfache Schritte** und **erkennt selbst die Anzahl der vorhandenen Cluster** sowie die Ausreißer. Zusätzlich ergibt sich der Vorteil, dass der Algorithmus im Gegensatz zu K-Means Cluster **mit beliebigen Formen und Größen** erkennt. Und er ist eben robust gegenüber Ausreißern.

Wie meinst du das mit den Formen?

Wenn du beispielsweise einen Cluster hast, der einen anderen ganz oder teilweise umschließt, dann könnte der K-Means-Algorithmus diesen nicht erkennen. Der DBScan folgt aber der Dichte der Punkte und kann eine solche Form erkennen. Ich zeige dir ein Vergleichsbild:



Der K-Means-Algorithmus schneidet den Datensatz, während der DBScan der Dichte folgt.

Das sind drei schöne Vorteile, da lege ich den anderen Algorithmus gleich zur Seite.

Nicht so schnell!

Wie du weißt, gibt es keinen Vorteil ohne Nachteil.

Die Nachteile des Algorithmus

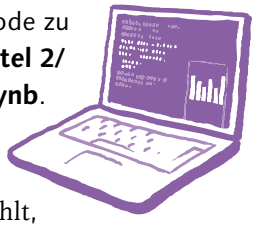
- Wenn die Punktedichte stark variiert, hat der Algorithmus Probleme. Die Scatter-Diagramme helfen dir bei der Darstellung und Einschätzung.
- Der Rechenaufwand des Algorithmus ist relativ hoch. Die Ermittlung der ε-Nachbarschaften ist rechenintensiv. Bei großen Datenmengen empfehle ich dir, zu überlegen, wie du einen Index aufbauen kannst, um möglichst effizient die Nachbarschaften zu ermitteln – zum Beispiel durch einen R*-Baum oder einen KD-Baum. Dadurch kannst du diesen Nachteil ausräumen.
- Leider ist der Algorithmus empfindlich gegenüber den Parametern. Die Wahl der sogenannten Hyperparameter ε und `MinPts` kann das Ergebnis stark beeinflussen.

Nichtsdestotrotz sind die Einsatzgebiete des DBScan-Algorithmus vielfältig.
Du kannst Bilder auf Basis ihrer Farbintensitäten bzw. ihrer Farbwerte **segmentieren** (also entsprechende Bereiche in Bildern feststellen), oder du nutzt den Algorithmus zur Anomalieerkennung (**Ausreißererkennung**) bei Finanzdaten, im Netzwerkverkehr oder in Produktionsprozessen. Auch bei der **Analyse** von Geodaten, Erdbebendaten oder der Verbreitung von Pflanzenarten findet dieser Algorithmus seine Anwendung, zum Beispiel durch die Bildung geografischer Cluster.

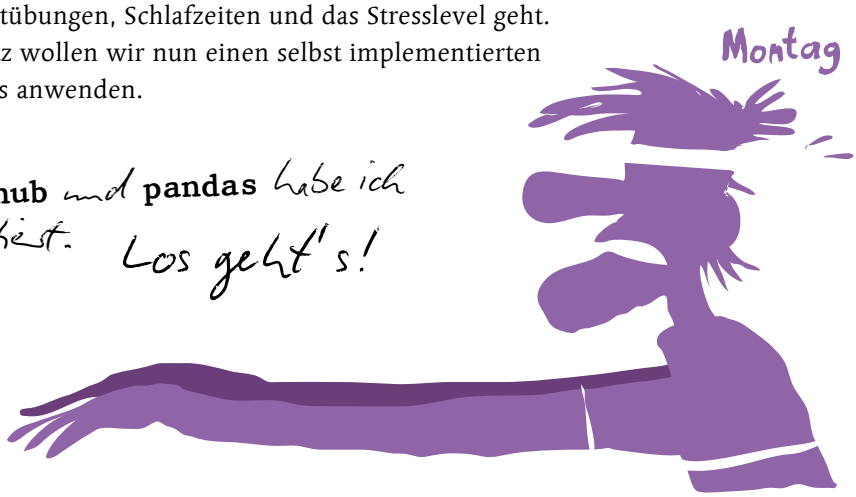
Wow, der hat ja wirklich überall seine Finger im Spiel!

Stressige Tage

[Notebook]
Wenn du chillen willst, statt stressig Code zu tippen, findest du den Code hier: **Kapitel 2/ DB Scan 1 – Selbst implementiert.ipynb**.



Ich habe einen Kaggle-Datensatz für dich ausgewählt, bei dem es um Sportübungen, Schlafzeiten und das Stresslevel geht. Auf diesen Datensatz wollen wir nun einen selbst implementierten DBScan-Algorithmus anwenden.




```
path = kagglehub.dataset_download("forrestcarltonl/stress-levels-dataset")
filename = path + "/Stress_levels_dataset.csv"
df = pd.read_csv(filename)
print(df.info())
```

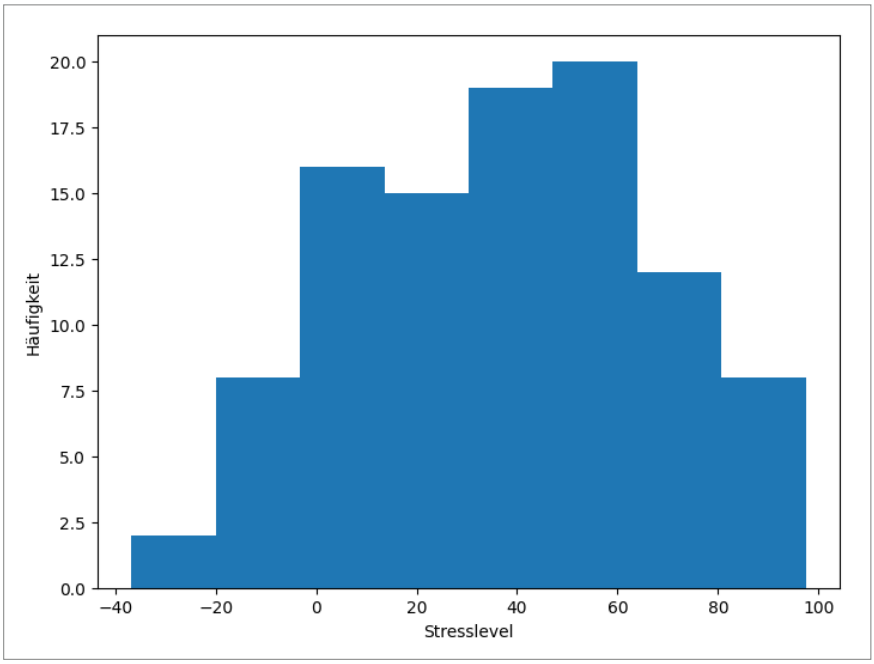
[Einfache Aufgabe]
Wie viele Datensätze hat der Datensatz?

Die Spalten in unserem Datensatz heißen **`Hours_of_Exercise_per_Week`**, **`Hours_of_Sleep_per_Night`** und **`Stress_Level`**.

[Schwierige Aufgabe]
Sieh dir das Histogramm des Stresslevels an – damit stellst du die Verteilung der einzelnen Stresslevel dar. Erstelle das Histogramm mit 8 Balken.

[Erledigt!]

```
import matplotlib.pyplot as plt
plt.figure(figsize=(8, 6))
df['Stress_Level'].hist(bins=8, grid=False)

plt.xlabel('Stresslevel')
plt.ylabel('Häufigkeit')
plt.show()
```



Verteilung der Stresslevel

DIENSTAG

Die Weste reichen von tiefenentspannt bis komplett unter Druck.
Ganz schön viel Stress!
Am häufigsten kommt ein Stresswert von 60 in den Daten vor.

Nun wissen wir, wie die Daten aussehen, und können uns an den Algorithmus machen. Wir benötigen eine Distanzfunktion, da wir die Abstände zwischen zwei beliebigen Datenpunkten messen müssen.

Da haben wir ja schon unsere **euclidean_distance**-Funktion.

```
import math
def euclidean_distance(point1, point2):
  return math.sqrt(sum((x - y)**2 for x, y in zip(point1, point2)))
```

Einen kleinen Teil haben wir schon. Jetzt müssen wir es nur noch schaffen, die Nachbarn von einem Datenpunkt zu finden, deren Abstand zum Punkt nicht größer ist als ε.

```
def region_query(data, point, eps):  neighbors = []
  for i, other_point in enumerate(data)*1:
    if point != other_point and euclidean_distance(point, other_point) <= eps*2:
      neighbors.append(i)*3
  return neighbors
```

*1 Es werden alle Datenpunkte durchgegangen und sowohl der Index **i** als auch der Punkt an sich betrachtet.

*2 Wenn es sich nicht um den gleichen Punkt handelt, wird die Distanz zum Punkt berechnet. Ist diese kleiner als das gewählte ε, dann ist es ein relevanter Nachbar.

*3 Wir merken uns lediglich die Indizes der Nachbarn.

Die Hilfsfunktionen sind also erledigt.
Jetzt ran an den DBScan!

MITTWOCH

```
def dbscan(data, eps, min_pts):
  labels = [-1] * len(data)*1
  cluster_id = 0
  for i, point in enumerate(data):
    if labels[i] != -1:*2
      continue

    neighbors = region_query(data, point, eps)*3
    if len(neighbors) < min_pts:
       continue*4

    cluster_id += 1*5
    labels[i] = cluster_id
    seed_set = neighbors.copy()*6

    while seed_set:
      current_point_index = seed_set.pop(0)*7
      if labels[current_point_index] == -1:
        labels[current_point_index] = cluster_id
      elif labels[current_point_index] == 0:*8
        labels[current_point_index] = cluster_id
      else
        continue  # Bereits Teil von einem Cluster

      current_point_neighbors = region_query(data, data[current_point_index], eps)*9

      if len(current_point_neighbors) >= min_pts:
        seed_set.extend([n for n in current_point_neighbors if n not in seed_
set])*10
  return labels
```

*1 Alle Daten vorab als Ausreißer markieren. Wir beweisen dann im Laufe des Algorithmus das Gegenteil.

*2 Wenn der Datensatz bereits zugeordnet ist, ignorieren wir diesen.

*3 Die Nachbarn im Umkreis ermitteln.

*4 Wenn nicht genügend Nachbarn vorhanden sind, bleibt der Datenpunkt ein Ausreißer. Wir sehen uns gleich den nächsten an.

*5 Yippie, wir haben einen neuen Cluster gefunden!

*6 Wir kopieren die Nachbarn, denn diese wollen wir im nächsten Schritt weiter analysieren, und sehen, ob sich der Cluster hier erweitert. Alle Punkte, die den Cluster potenziell erweitern können, werden in diese Liste aufgenommen.

*7 Solang noch ein Punkt in der Liste übrig ist, wird alles wiederholt. Der erste Punkt wird rausgenommen. Falls dieser noch keinem Cluster zugewiesen ist, weisen wir diesen dem Cluster zu.

*8 Randbereichprüfung. Der Punkt ist zwar nahe genug, hat aber selbst zu wenige Nachbarn, um als Zentrum zu fungieren. Das wird ein Randpunkt.

*9 Von diesem Punkt aus werden erneut die Nachbarn gesucht.

*10 Nachdem geprüft wurde, ob genügend Nachbarn in der Nähe sind, werden all diese erneut in die Liste der potenziellen neuen Clustermitglieder hinzugefügt, damit der Cluster wachsen kann.

Du bist bestimmt schon gespannt auf das Ergebnis.

Wir haben es fast geschafft!

Nur noch die Funktion aufrufen und die Ergebnisse zeichnen.

[Code bearbeiten]

Rufe die Funktion mit den zwei Spalten **`Hours_of_Exercise_per_Week`** und **`Stress_Level`** auf.

Aber was soll ich bei den Mindestpunkten und ε einsetzen?

Nimm erst einmal 6.5 für ε und 4 für die Mindestpunkte.

[Notiz]

Das Finden der Parameterwerte ist oftmals ein Herantasten. Ich zeige dir im Laufe der Zeit noch, wie du diese sogenannten Hyperparameter ermitteln kannst.

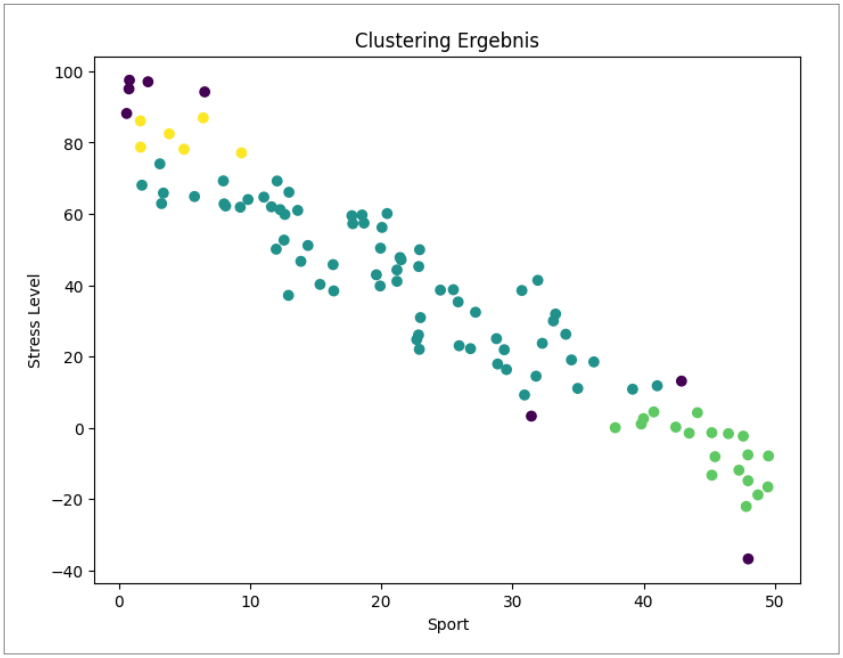
[Erledigt!]

```
selected_columns =
['Hours_of_Exercise_per_Week', 'Stress_Level']
data = df[selected_columns].values.tolist()
eps = 6.5
min_pts = 4
labels = dbscan(data, eps, min_pts)
```

Wunderbar, wir haben die Labels – also die Cluster.

Lass uns die Cluster aufzeichnen und sehen, wie die Stresslevels mit der körperlichen Ertüchtigung zusammenhängen und wie diese durch den DBScan-Algorithmus in Cluster zusammengefasst werden.

```
import matplotlib.pyplot as plt
x_coords = [point[0] for point in data]
y_coords = [point[1] for point in data]
plt.figure(figsize=(8, 6))
plt.scatter(x_coords, y_coords, c=labels, cmap='viridis')
plt.title('Clustering Ergebnis')
plt.xlabel('Sport')
plt.ylabel('Stress Level')
plt.colorbar(label='Cluster')
plt.show()
```



Clustering der Datenpunkte durch DBScan

[Einfache Aufgabe]
Wie viele Cluster kannst du aus der Grafik ablesen?

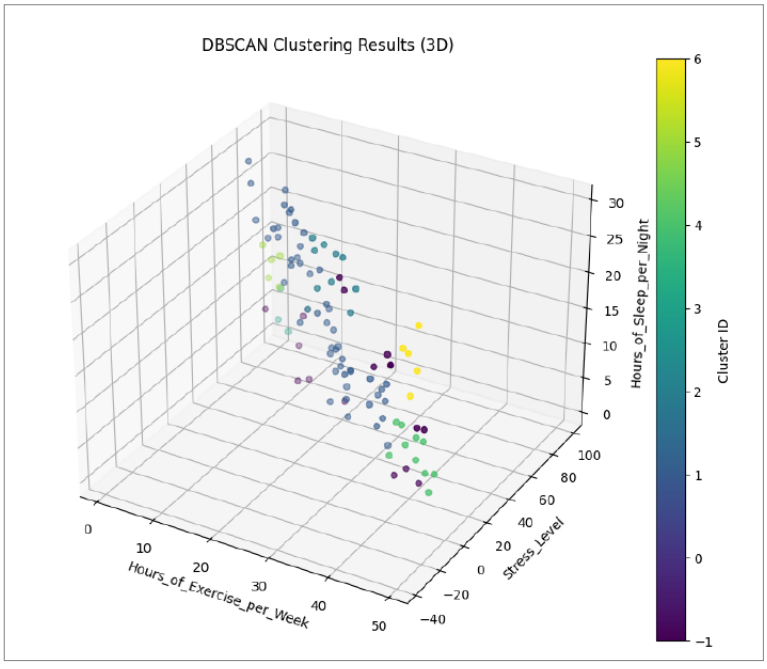
Drei, nein, ... sieben?

Es sind drei. Das Gelb und die zwei Grüntöne. Die lila Punkte sind diejenigen mit dem Wert –1, die keinem Cluster zugeordnet worden sind. Das sind unsere Anomalien.

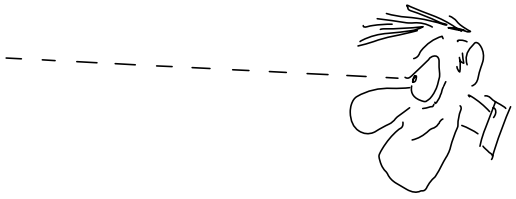
Aber wie du weißt, funktionieren diese Algorithmen ja in mehreren Dimensionen. Und eine unausrottbare Eigenschaft von KI-Datensätzen ist, dass die Datensätze immer viele Dimensionen haben. Gut, hier haben wir jetzt nur drei, aber meist sind es noch mehr.

[Schwierige Aufgabe]
Aktualisiere den Code mit der dritten Dimension und gib ein 3D-Diagramm aus. Verwende nun 8 als ε und 3 als Mindestpunkteanzahl.

```
selected_columns = ['Hours_of_Exercise_per_Week', 'Stress_Level',
'Hours_of_Sleep_per_Night']
data = df[selected_columns].values.tolist()

eps = 8
min_pts = 3
labels = dbscan(data, eps, min_pts)

# Prepare data for plotting
x_coords = [point[0] for point in data]
y_coords = [point[1] for point in data]
z_coords = [point[2] for point in data]

# Plotting
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
scatter = ax.scatter(x_coords, y_coords, z_coords, c=labels, cmap='viridis')

ax.set_xlabel(selected_columns[0])
ax.set_ylabel(selected_columns[1])
ax.set_zlabel(selected_columns[2])
ax.set_title('DBSCAN Clustering Results (3D)')
plt.colorbar(scatter, label='Cluster ID')
```



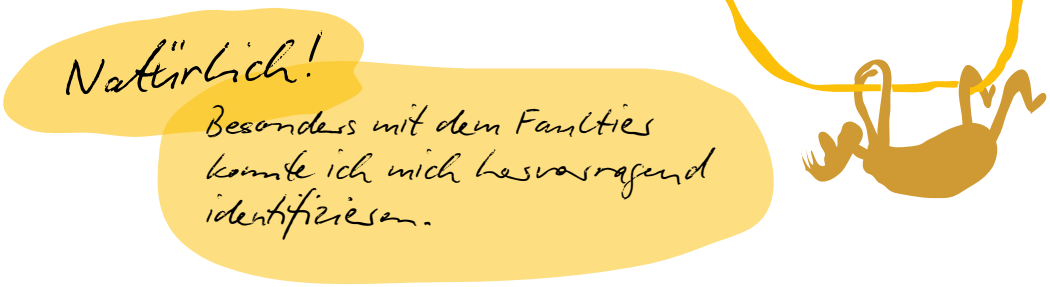
Clustering-Darstellung im 3D-Raum



Ja, da sind wieder ein paar Ausreißer zu sehen. Und ich habe dieses Mal mehr Cluster erhalten.

Drama-Nachbarn, die nicht ins Bild passen

Du erinnerst dich bestimmt an unsere Drama-Tiere.



Wir haben Cluster für die Tiere mithilfe von K-Means erstellt. Und nun verwenden wir wieder SKLearn, um diese Cluster mithilfe von DBScan zu unterteilen. Wir wollen uns ansehen, ob es Punkte gibt, die nicht ins Bild passen.

[Einfache Aufgabe]
Kopiere dir die Daten in ein neues Notebook und lade sie wie gewohnt in einen Pandas-Dataframe.

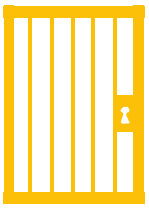
[Notebook]
Den gesamten Code findest du hier: **Kapitel 2/DBScan2 – Tiere.ipynb**.

Standardisieren wir die Werte als Erstes mit dem bekannten **`StandardScaler`**.

```
from sklearn.cluster import DBSCAN
from sklearn.preprocessing import StandardScaler
# Merkmale auswählen
X = df[["Drama", "Bewegung"]].values
# Skalieren
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

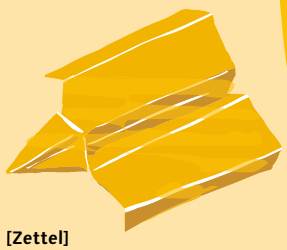
Nun führen wir den DBScan aus und schreiben die ermittelten Cluster zurück in den DataFrame – in eine eigene Spalte mit dem Namen **`Cluster`**.

```
dbscan = DBSCAN(eps=0.9, min_samples=3)
clusters = dbscan.fit_predict(X_scaled)
df['Cluster'] = clusters
```



Fertig.

Nein, nein, ich will das jetzt schon grafisch sehen.



[Zettel]
Ausreißer werden dem Cluster –1 zugeordnet.

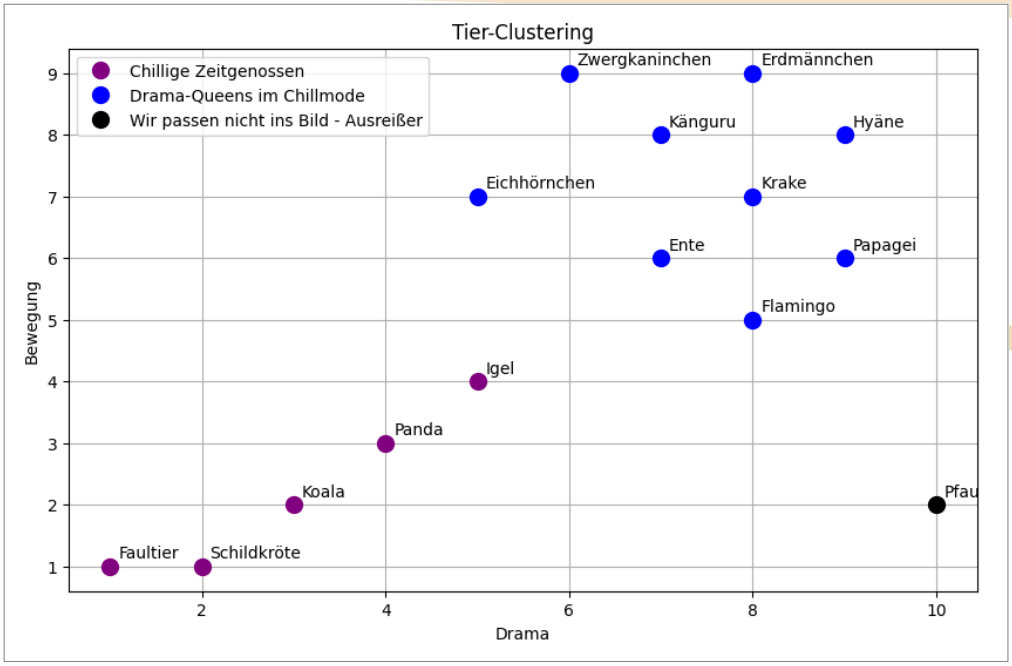
```
colors = {}
labels = {}
for k in unique_labels:
  if k == -1:*1
    colors[k] = [0, 0, 0, 1]
    labels[k] = 'Wir passen nicht ins Bild - Ausreißer'
  elif k == 0:
    colors[k] = 'purple'
    labels[k] = 'Chillige Zeitgenossen'
  elif k == 1:
    colors[k] = 'blue'
    labels[k] = 'Drama-Queens im Chillmode'
  else
      colors[k] = plt.cm.Spectral(k/len(unique_labels))*2
      labels[k] = f'Cluster {k}'

for k in unique_labels:
  class_member_mask = (clusters == k)
  xy = X[class_member_mask]
  names = df['Tier'][class_member_mask].values

  plt.plot(xy[:, 0], xy[:, 1], 'o', color=colors[k],
           markersize=10, label=labels.get(k,f'Cluster {k}'))*3
  for i, txt in enumerate(names):*4
    plt.annotate(txt, (xy[i, 0], xy[i, 1]),
          textcoords="offset points", xytext=(5,5), ha='left')

plt.title('Tier-Clustering')
plt.xlabel('Drama')
plt.ylabel('Bewegung')
plt.legend()
plt.grid(True)
plt.show()
```

*1 Ausreißer haben den Wert –1.

*2 Sollten sich weitere Cluster ergeben, wird eine Farbe und die Clusterbezeichnung gewählt.

*3 Zeichnen der Clusterpunkte.

*4 Die Tierbezeichnung wollen wir auch sehen.



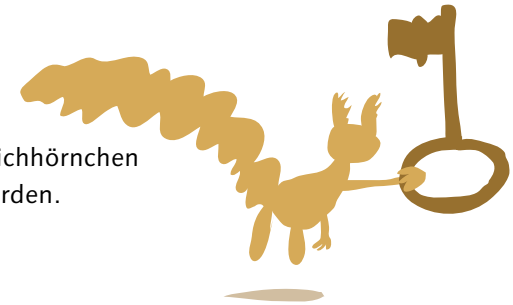
Clustering der Tiere mit DBScan

[Einfache Aufgabe]

Passe den ε-Wert an. Wähle zum Ausprobieren **0.8**. Was kannst du beobachten?

[Lösung]

Jetzt ist auch das Eichhörnchen ein Ausreißer geworden.



Wie wir mehrfach gesehen haben, ist der Algorithmus in der Lage, Ausreißer zu erkennen. Diese werden entweder im Detail analysiert, wenn es beispielsweise um ungewöhnliche Kreditkartenabrechnungen geht, oder eben entfernt, weil Ausreißer im Datensatz möglicherweise störend sind.

[Schwierige Aufgabe]

Entferne die Ausreißer und zeichne die Grafik erneut.

```
# Nachdem der DBScan ausgeführt wurde, ändert sich der Code:
df['Cluster'] = clusters
df_no_outliers = df[df['Cluster'] != -1]*1

# Merkmale zum Plotten ohne Ausreißer auswählen
X_no_outliers = df_no_outliers[["Drama", "Bewegung"]].values*2
clusters_no_outliers = df_no_outliers['Cluster'].values*3

plt.figure(figsize=(12, 8))
unique_labels = set(clusters_no_outliers)*4

colors = {}
labels = {}
for k in unique_labels:
    if k == 0:
        colors[k] = 'purple'
        labels[k] = 'Chillige Zeitgenossen'
    elif k == 1:
        colors[k] = 'blue'
        labels[k] = 'Drama-Queens im Chillmode'
    else
        colors[k] = plt.cm.Spectral(k/len(unique_labels))
        labels[k] = f'Cluster {k}'

for k in unique_labels:
    class_member_mask = (clusters_no_outliers == k)*5
    xy = X_no_outliers[class_member_mask]
    names = df_no_outliers['Tier'][class_member_mask].values

    plt.plot(xy[:, 0], xy[:, 1], 'o', color=colors[k],
             markersize=10, label=labels.get(k,f'Cluster {k}'))

    for i, txt in enumerate(names):
        plt.annotate(txt, (xy[i, 0], xy[i, 1]), textcoords="offset points",
        xytext=(5,5), ha='left')

plt.title('Tier-Clustering (ohne Ausreißer)')
plt.xlabel('Drama')
plt.ylabel('Bewegung')
plt.legend()
plt.grid(True)
plt.show()
```

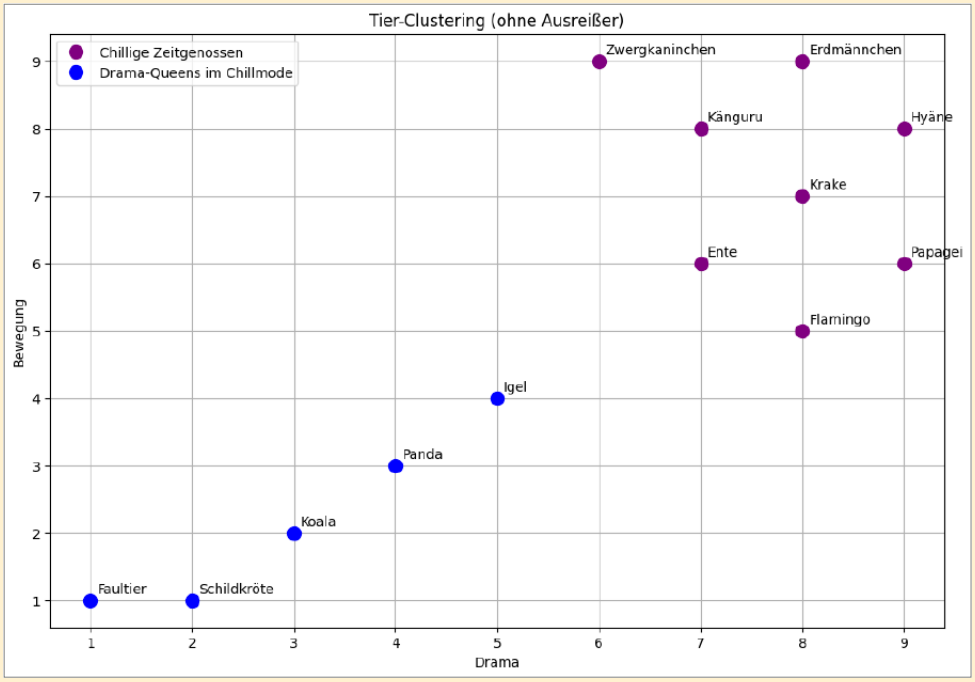
*1 Wir filtern und entfernen alles, was den Cluster –1 besitzt.

*2 Wir laden uns die Daten neu, da diese nun gefiltert sind.

*3 Auch die Cluster laden wir uns erneut.

*4 Die Labels nicht vergessen.

*5 Und nun werden zum Zeichnen die gefilterten Werte verwendet.

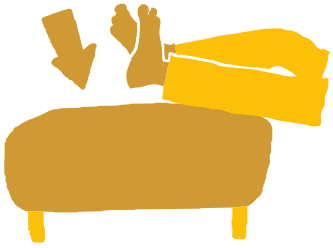


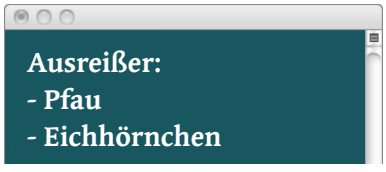
Tier-Clustering ohne Ausreißer mit einem ε-Wert von 0,8 und einer Mindestpunkteanzahl von 3

[Einfache Aufgabe]
Lass dir zur Kontrolle noch die Ausreißer ausgeben.

[Lösung]

```
outliers = df[df['Cluster'] == -1]['Tier'].values
print("Ausreißer:")
for outlier in outliers:
  print(f"- {outlier}")
```





Wunderbar, du bist bereit für den nächsten Algorithmus. Bisher haben wir immer Datenpunkte gruppiert. Nun wird es Zeit, sich darum zu kümmern, neue Datenpunkte bestehenden Clustern zuzuordnen.

Neue Nachbarn

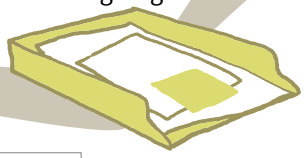
In vielen Recommendation-Engines wird der Algorithmus **K-Nearest-Neighbors**, kurz **KNN**, eingesetzt. Wenn dir der nächste Film, der dir wahrscheinlich gefällt, der nächste Song oder ein Produkt in einem Onlineshop vorgeschlagen werden soll, steckt oftmals der KNN-Algorithmus dahinter. Auch in der medizinischen Diagnostik, bei der Bildanalyse oder im Finanzbereich findet der Algorithmus Anwendung.

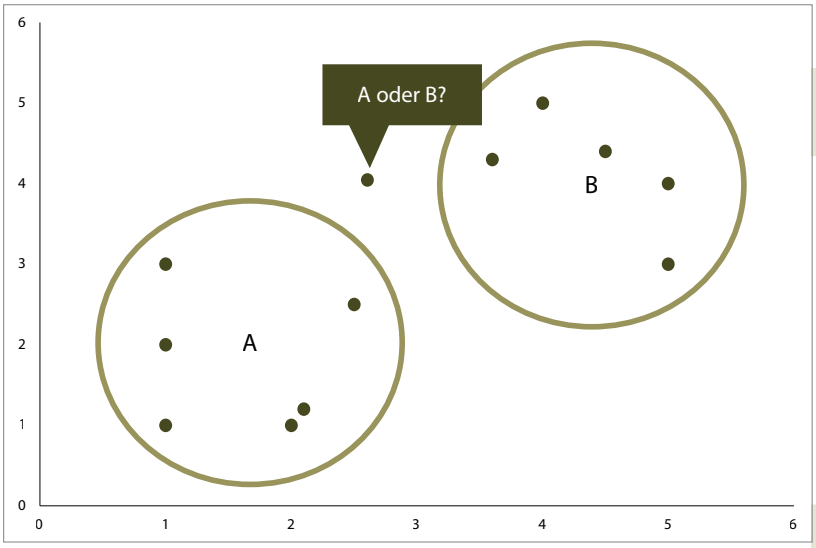
[Achtung]
Die Bezeichnung **KNN** wird auch gerne als Abkürzung für »künstliche neuronale Netze« verwendet. Nicht verwechseln!

Wir haben mit KNN nun einen Algorithmus, der in die Kategorie Supervised Learning fällt – also überwachtes Lernen. Du benötigst bereits fertige Cluster und weist neue Datenpunkte diesen Clustern zu.

Clustern kann ich ja jetzt.

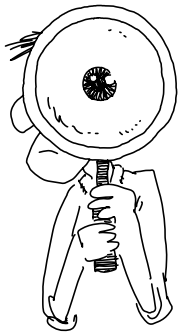
[Ablage]
KNN beantwortet die Frage, zu welchem bestehenden Cluster ein neuer Datenpunkt hinzugefügt werden soll.





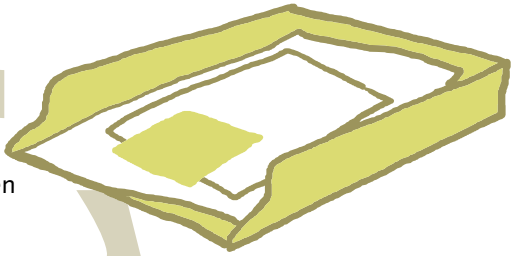
Zu welchem Cluster gehört der neue Datenpunkt?

KNN ist ein demokratischer Algorithmus. Es wird ein Mehrheitsentscheid verwendet, um die Clusterzugehörigkeit zu entscheiden. Du musst für diesen Algorithmus nur einen Parameter festlegen: das K.

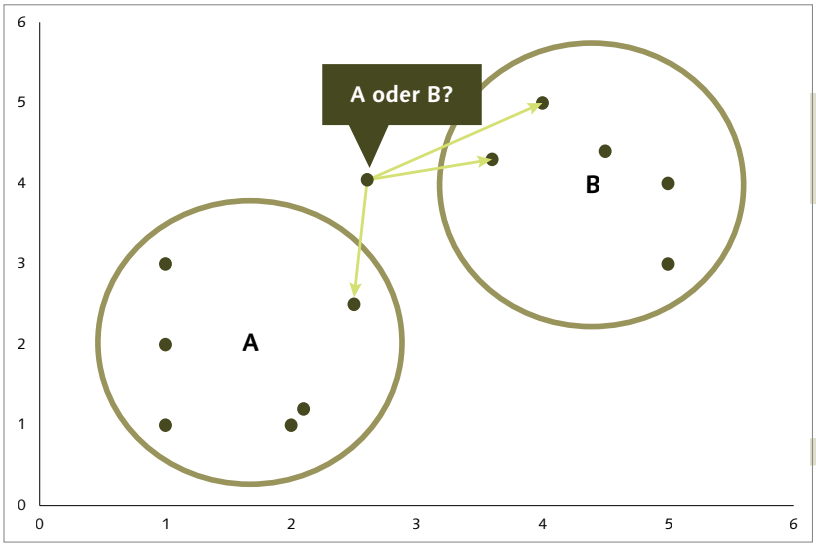


Und wofür steht das K?

[Ablage]
K ist die Anzahl der nächsten Nachbarn, die betrachtet werden sollen.



Du benötigst also die aktuelle Clusterzuordnung der Datenpunkte, die Anzahl der nächsten Nachbarn, die du verwenden willst, und – wie so oft – eine Distanzmetrik: Euklid oder Manhattan etc.

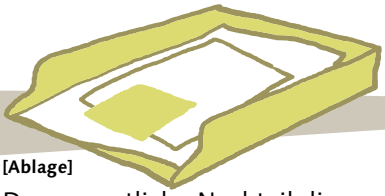


Analyse der drei nächsten Nachbarn

Der Algorithmus sucht sich also die K Nachbarn mit der minimalen Distanz. Er analysiert, welche der Nachbarn zu welchem Cluster gehören und schließt sich der Mehrheit an. In der Abbildung ist zu sehen, dass für unseren Punkt zwei Nachbarn aus dem Cluster B und ein Nachbar aus dem Cluster A in Frage kommen. Diese werden untersucht. Demnach entscheidet der Algorithmus, dass der neue Datenpunkt dem Cluster B zugewiesen wird.

Das wars schon?

Ja, so einfach ist es.



[Ablage]
Der wesentliche Nachteil dieses Algorithmus ist, dass er empfindlich gegenüber irrelevanten Merkmalen ist. Also erweitere deinen Datensatz nicht um nutzlose Merkmale wie zum Beispiel IDs und verwende die Merkmalskorrelation, um unnötige Features zu entfernen.

[Achtung]
Was KNN definitiv nicht kann, ist die Generierung neuer Cluster. Er ermöglicht **nur eine Zuordnung** zu den bestehenden Clustern.

Wertvorhersage mit KNN

KNN kann nicht nur neue Datensätze zu Clustern zuordnen. Du kannst den Algorithmus auch benutzen, um **Werte** vorherzusagen. Stell dir beispielsweise einen Datensatz für Fahrräder und Fahrradpreise vor, in dem Merkmale wie Preis, Gewicht, Anzahl der Gänge etc. aufgelistet sind. Wenn du jetzt bei einem neuen Fahrrad den Preis einordnen möchtest, dann verwendest du alle Informationen aus dem Datensatz (mit Ausnahme des Preises) und sortierst den Datenpunkt ein, vergleichst also Gewicht, Anzahl der Gänge und so weiter. Anschließend nimmst du die entsprechenden Nachbarn, also Räder mit ähnlichen Merkmalen, und verwendest den Durchschnittspreis (oder den mit den Abständen gewichteten Durchschnittspreis) der Nachbarn.

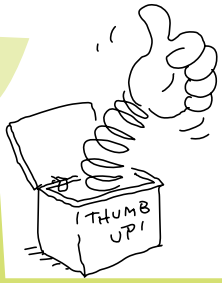
Clever! Und nebenbei schaue ich mir meinen Datensatz auch noch mal ganz genau an.

Du kannst sogar die Linien oder Flächen berechnen, an denen sich die Zuordnungen zum einen oder anderen Wert ändern – je nachdem, wie viele Nachbarn rundherum sind und wann sich die Nähe zum nächsten Nachbarn ergibt.

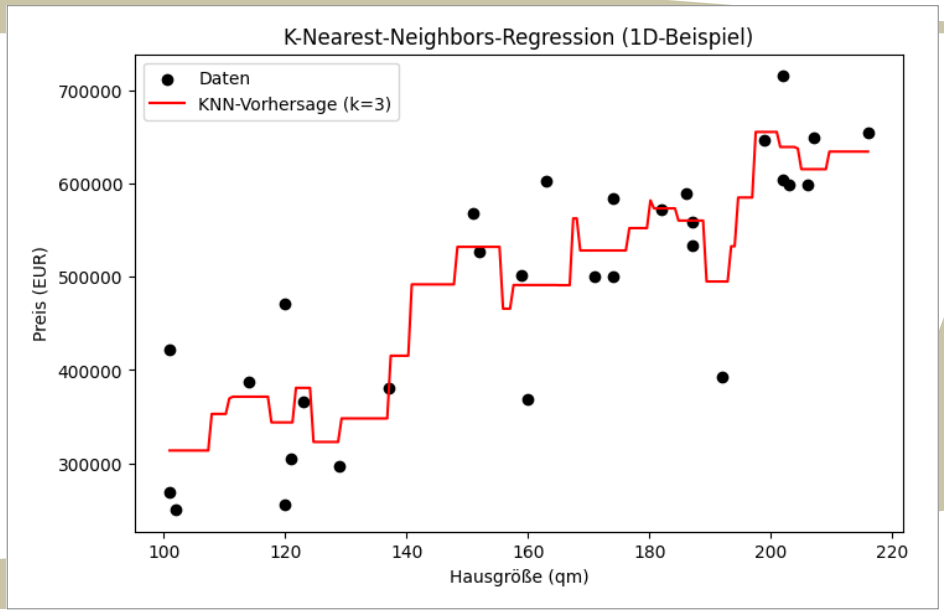
Moment, das habe ich jetzt nicht ganz verstanden. Wie meinst du das?

Angenommen, wir haben lediglich eine Quadratmeteranzahl und den Preis eines Hauses. Normalerweise ist der Preis natürlich noch von vielen weiteren Parametern abhängig, aber zur einfachen Visualisierung setzen wir mal diese vereinfachte Zuordnung voraus. Nun hast du ein neues Haus mit einer bestimmten Quadratmeteranzahl und möchtest den Hauspreis mithilfe von KNN ermitteln. Also suchst du dir drei Häuser aus deinem Datensatz, die eine ähnliche Quadratmeterzahl haben, und nimmst den Durchschnittspreis von den drei Häusern – das ist der Preis, den du für das Haus erwarten kannst. Wenn sich jedoch die Quadratmeter leicht erhöhen oder reduzieren, kann es sein, dass andere Häuser aus dem Datensatz nun deinem am ähnlichsten sind und sich ein ganz anderer Durchschnittspreis ergibt.

Okay, verstanden.



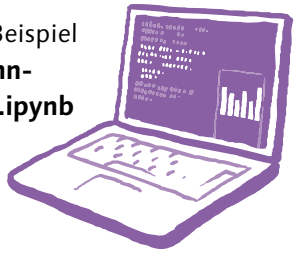
Und wenn du alle Werte durchgehst und jeweils die Berechnung betrachtest, siehst du, wo die Preissprünge sind. Das siehst auf der Abbildung noch mal genauer.



Preissprünge je nach Quadratmetern und Wertvorhersagen für den Durchschnittspreis in dieser Größe

Gesellschaftsspiele in der Nachbarschaft

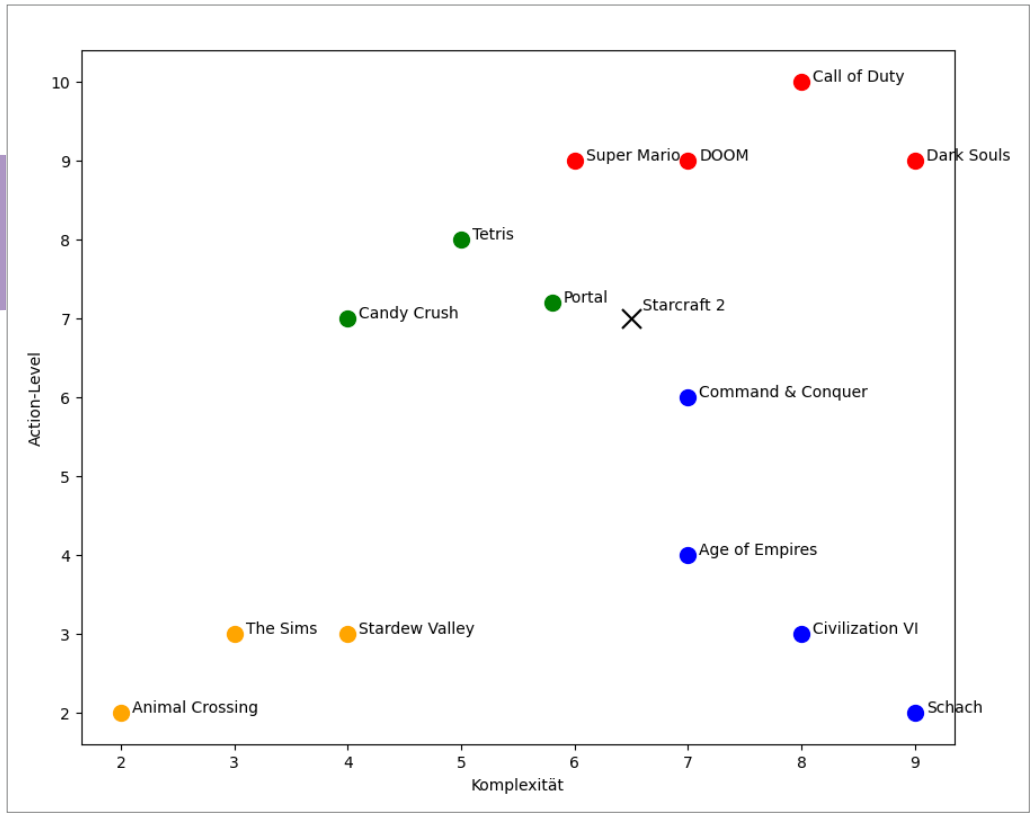
[Notebook]
Der Code zu diesem Beispiel ist in **Kapitel 2/03-knn-games-classification.ipynb** zu finden.



Du hast jetzt verstanden, wie der Algorithmus funktioniert, und dass er flexibel einsetzbar ist – sowohl zur Klassifizierung als auch zur Vorhersage konkreter Zahlenwerte, also für Regressionsaufgaben. Lass ihn uns implementieren. Bestimmt juckt es dich schon in den Fingern. Das ist unser Datensatz:

Spiel	Komplexität	Action-Level	Kategorie
Schach	9	2	Strategie
Civilization VI	8	3	Strategie
Age of Empires	7	4	Strategie
Command & Conquer	7	6	Strategie
Tetris	5	8	Puzzle
Candy Crush	4	7	Puzzle
Portal	5.8	7.2	Puzzle
Dark Souls	9	9	Action
Call of Duty	8	10	Action
DOOM	7	9	Action
Super Mario	6	6	Action
Animal Crossing	2	2	Simulation
The Sims	3	3	Simulation
Stardew Valley	4	3	Simulation

Wir wollen wissen, wie Starcraft 2 mit einem Komplexitätslevel von 6.5 und einem Action-Level von 7 klassifiziert wird. Im Diagramm sieht das wie folgt aus:



Spieleklassifizierung mit K-Nearest-Neighbors

Wunderbar, unser Datensatz soll wie folgt aussehen: erst die Bezeichnung des Spiels, dann die Komplexität in Form eines Punktes und dann das Action-Level.

```
game_data = {
  "Schach": (9, 2), "Civilization VI": (8, 3),… }
```

Zusätzlich benötigen wir das Label für alle Spiele, da diese bereits klassifiziert sind.

```
labels = {
  "Schach": "Strategie", "Civilization VI": "Strategie",
"Age of Empires": "Strategie", … }
```

Jetzt kommt unser neuer Datenpunkt, den wir einordnen wollen, und die Hilfsfunktion für den euklidischen Abstand zwischen zwei Punkten:

```
new_game = (6.5, 7)  # Starcraft 2
```

[Einfache Aufgabe]
Erstelle die Hilfsfunktion **`euclidean_distance`**.

[Erledigt!]

```
import math
def euclidean_distance(p1, p2):
  return math.sqrt((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2)
```

Super, Schrödinger! Die Implementierung von KNN ist nicht nur bei der Erklärung einfach, sondern auch im Code. Wir berechnen die Distanz zwischen dem neuen Punkt und jedem anderen Punkt im Datensatz und merken uns diese Informationen in einer Liste. Anschließend sortieren wir den Datensatz nach der Distanz, verwenden lediglich die K ersten Elemente und wählen von denen die häufigste Kategorie aus.

```
def knn_manual(train_data, train_labels, new_point, k=3):
  distances = []

  for game, coords in train_data.items():*1
    distance = euclidean_distance(coords, new_point)
    distances.append((distance, train_labels[game]))*2

  distances.sort()*3
  k_nearest = distances[:k]

  category_count = Counter(label for _, label in k_nearest)*4

  return category_count.most_common(1)[0][0]*5
```

*1 Berechne die Distanz zu jedem bekannten Spiel.

*2 Distanz und Label werden als Tupel in das Array hinzugefügt.

*3 Sortieren nach Distanz und die **k** nächsten Nachbarn auswählen.

*4 Abstimmen und Stimmen auszählen.

*5 Am häufigsten vertretene Kategorie zurückgeben.

[Code bearbeiten]
Vergiss nicht, das `Counter`-Objekt von `collections` zu importieren.

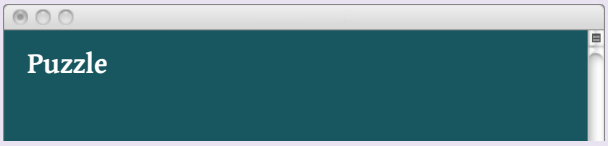
[Erledigt!]
```
from collections import Counter
```

Na, bist du schon gespannt, wie Starcraft 2 klassifiziert wird?

Starcraft 2 ist ganz klar ein Echtzeit-Strategiespiel.

Dann fragen wir mal unseren Algorithmus:

```
predicted_category = knn_manual(game_data, labels, new_game, k=3)
print(f"Starcraft 2 wurde klassifiziert als: {predicted_category}")
```



Was ist das denn für ein Voodoo?

Niemals!

[Code bearbeiten]
Sieh dir nochmal die Werte an, vielleicht haben wir uns ganz am Anfang mit einem Komplexitätswert von 6,5 ja verschätzt.

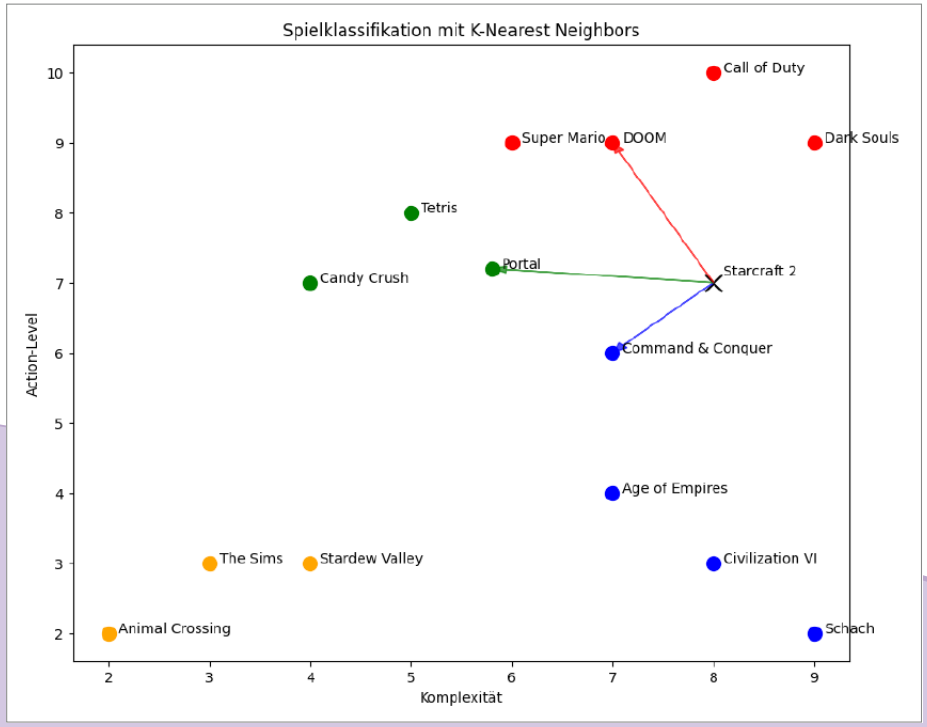
Stimmt, das sind mindestens 8, wenn man sich die anderen Spiele in der Tabelle so anschaut.

Eins, zwei, Zamberei!
Schon haben wir die Strategie-Kategorie.

[Achtung]
Ja, die Eingangsdaten müssen natürlich korrekt sein, damit der Algorithmus ein richtiges Ergebnis ausspuckt. Das soll aber keine Einladung sein, dass du künftig immer die Eingangswerte so anpasst, dass das Ergebnis deinen Erwartungen entspricht.

Auf die Idee würde ich nie kommen!

Übrigens sind **das** jetzt die Nachbarn, die wir betrachtet haben.
So eindeutig ist das gar nicht …



Die stimmberechtigten Nachbarn

Sagtest du nicht, das ist eine Mehrheitsentscheidung?

[Notiz]
Bei nicht eindeutiger Abstimmung erhält der Cluster mit dem Punkt mit der kürzesten Distanz den Zuschlag.

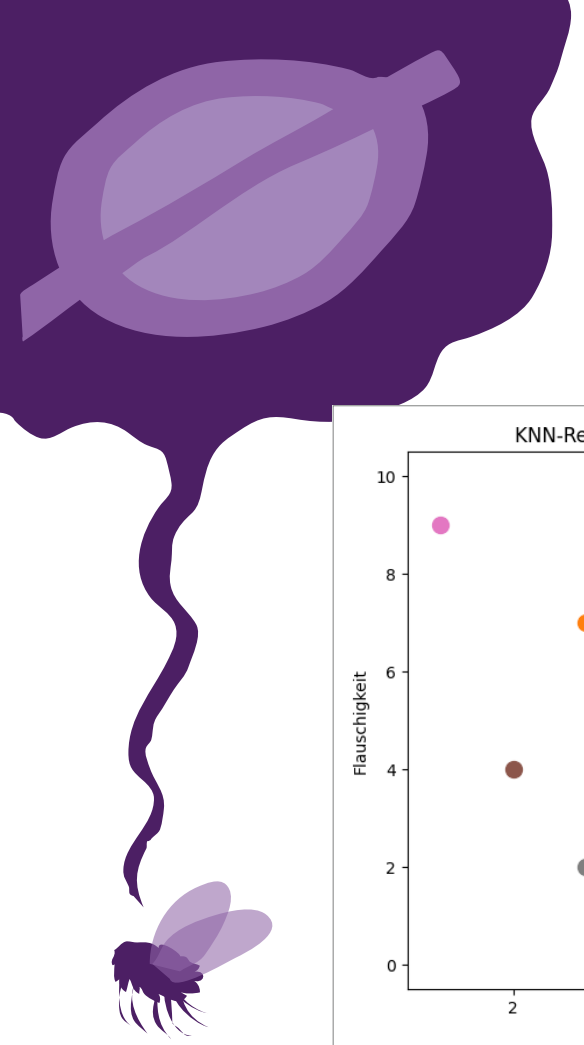
Lärmbelästigung in der Nachbarschaft?

[Notebook]
Den zugehörigen Code findest du unter **Kapitel 2/04-knn-loudness.ipynb**.

Jetzt schauen wir uns auch noch ein Regressionsbeispiel an, in dem wir einen konkreten Wert vorhersagen. Und zwar wollen wir auf Basis von Größe und Flauschigkeit auf die Lautstärke von Tieren schließen.

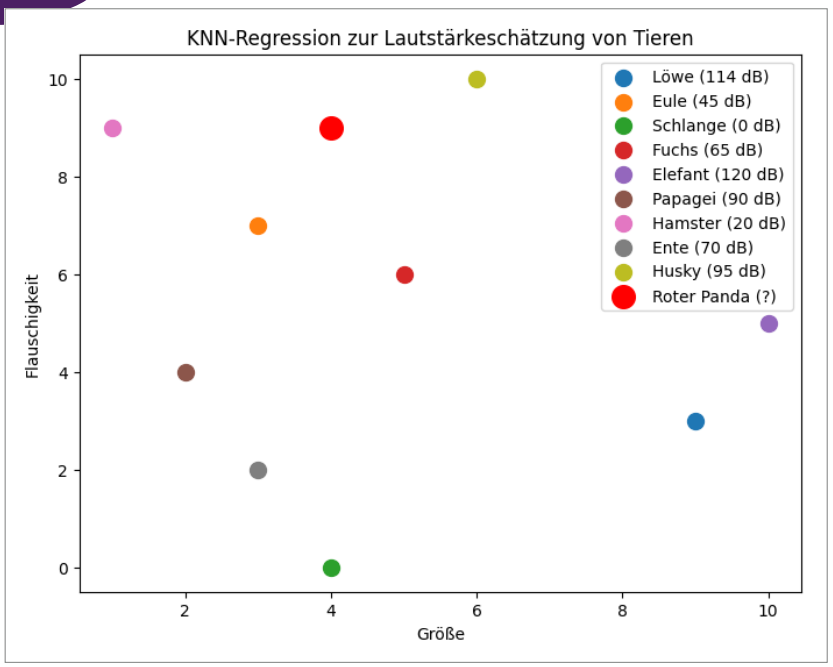
Tier	Größe (1–10)	Flauschigkeit (10–10)	Lautstärke (dB)
Löwe	9	3	114
Eule	3	7	45
Schlange	4	0	0
Fuchs	5	6	65
Elefant	10	5	120
Papagei	2	4	90
Hamster	1	9	20
Ente	3	2	70
Husky	6	10	95

[Schwierige Aufgabe]
Versuchen wir doch mal, den Roten Panda mit der Größe von 4 und der Flauschigkeit von 9 einer Lautstärke zuzuordnen.



[Notiz]

Das Verfahren von KNN bei der Wertvorhersage ist das gleiche Prozedere wie bei der Clusterzuordnung. Der einzige Unterschied liegt darin, am Ende keinen Mehrheitsentscheid zu machen, sondern den (gewichteten) Durchschnittswert zu berechnen.



Flauschigkeit und Größe der Tiere

```
def knn_regression(train_data, new_point, k=3):
  distances = []

  for animal, (size, fluff, loudness) in train_data.items():*1
    distance = euclidean_distance((size, fluff), new_point)
    distances.append((distance, loudness))

  distances.sort()*2
  k_nearest = distances[:k]
```

*1 Für jedes Tier die Distanz zum Roten Panda berechnen.

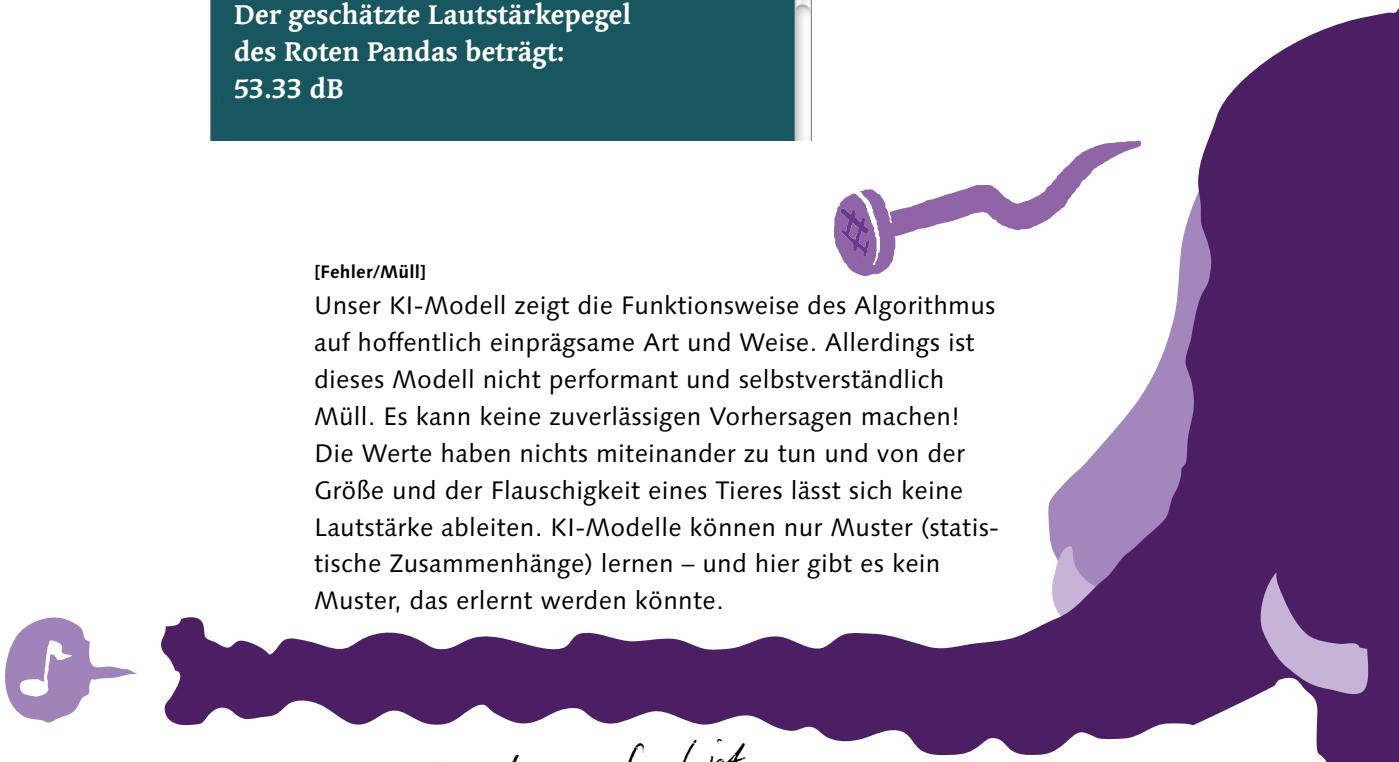
*2 Sortieren und die k Nachbarn selektieren.

```
    avg_loudness = np.mean([loudness for _, loudness in k_nearest])*3
    return avg_loudness

predicted_loudness = knn_regression(animal_data, new_animal, k=3)*4
print(f"Der geschätzte Lautstärkepegel des Roten Pandas beträgt: {predicted_
loudness:.2f} dB")
```

*4 Die Lautstärke des Pandas vorhersagen.

*3 Hier ist der Unterschied zur Clusterzuordnung: Es wird der Durchschnittswert der Nachbarn gebildet und keine Abstimmung durchgeführt.

```
Der geschätzte Lautstärkepegel
des Roten Pandas beträgt:
53.33 dB
```



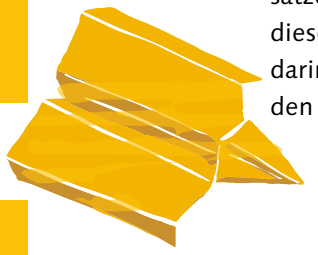
[Fehler/Müll]

Unser KI-Modell zeigt die Funktionsweise des Algorithmus auf hoffentlich einprägsame Art und Weise. Allerdings ist dieses Modell nicht performant und selbstverständlich Müll. Es kann keine zuverlässigen Vorhersagen machen! Die Werte haben nichts miteinander zu tun und von der Größe und der Flauschigkeit eines Tieres lässt sich keine Lautstärke ableiten. KI-Modelle können nur Muster (statistische Zusammenhänge) lernen – und hier gibt es kein Muster, das erlernt werden könnte.

Ich habe ja geahnt, dass da was faul ist, als du mich gebeten hast, die Flauschigkeit von Schlangen einzuschätzen ...

Dann wagen wir uns jetzt mal an einen richtigen Datensatz und die Verwendung einer Library.

An die Nachbarn angepasst statt nur geschätzt



[Zettel]
SKLearn bietet für bestimmte Datensätze eine einfache Art und Weise, diese zu laden. Unter anderem ist darin der Iris-Datensatz enthalten, den wir bereits benutzt haben.

[Notebook]
Den Code zu diesem Beispiel findest du hier: **Kapitel 2/05-knn-sklearn.ipynb**.

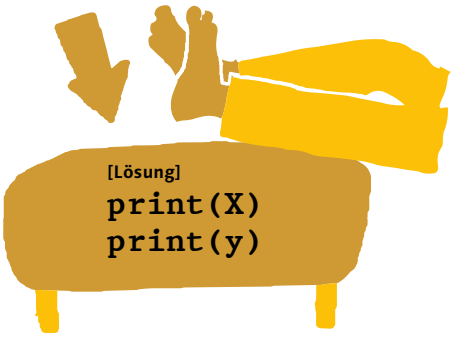
Der Iris-Datensatz enthält auch Labels für die Blumen und ist daher wunderbar für eine kleine K-Nearest-Neighbors-Übung einsetzbar:

```
import numpy as np
from sklearn import datasets
from sklearn.neighbors import KNeighborsClassifier

iris = datasets.load_iris()*1
X = iris.data*2
y = iris.target*3
```

*2 Die Daten an sich sind in der Eigenschaft **data** enthalten.

*1 Iris-Datensatz laden.

*3 Die Labels auslesen.



Das ist ein Array von Arrays mit vielen Werten.

Richtig, jedes Array entspricht einer Blume. Und da es viele Blumen sind, ist es ein Array von Arrays. Die Labels sind nur Werte von 0–2.

Wenn du **`iris.target_names`** verwendest, siehst du die Bezeichnungen der Blumen-Klassen.

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

Verwendest du **`iris.feature_names`** bekommst du die Spaltenbezeichnungen angezeigt.

```
['sepal length (cm)', 'sepal
width (cm)', 'petal length
(cm)', 'petal width (cm)']
```

Nachdem du jetzt die Daten verstanden hast, wollen wir doch mal sehen, wie KNN die Klassifizierung durchführt.

```
knn = KNeighborsClassifier(n_neighbors=3)*1
knn.fit(X, y)*2

new_flower = np.array([[5.1, 3.5, 1.4, 0.2]])*3
prediction = knn.predict(new_flower)*4
print(f'Die Vorhersage für die neue Blume ist: {iris.target_names[prediction][0]}')
```

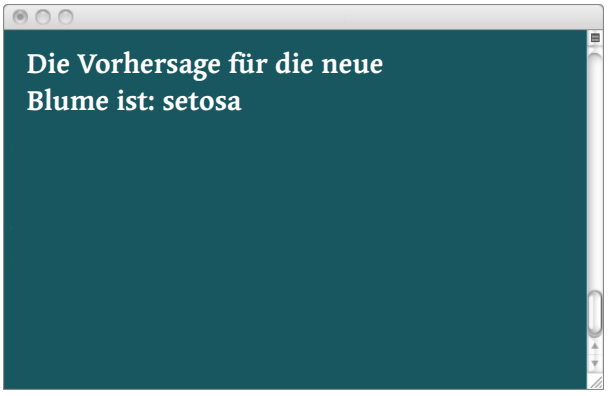
*1 KNN verwenden mit 3 Nachbarn

*2 Trainieren

*3 Beispielblume initialisieren

*4 Vorhersage treffen

Oh, aber das sind doch einfach nur Einzeiler?



Nur ein Einzeiler, und dennoch fehlt etwas Wesentliches – die Normalisierung!

[Lösung]

```
from sklearn.preprocessing import StandardScaler

# Vor dem Aufruf der fit-Funktion das Normalisieren der Werte
nicht vergessen!
scaler = StandardScaler()
X = scaler.fit_transform(X)
```

Lust auf eine Regressionsaufgabe?

Selbstverständlich.
Legen wir los!

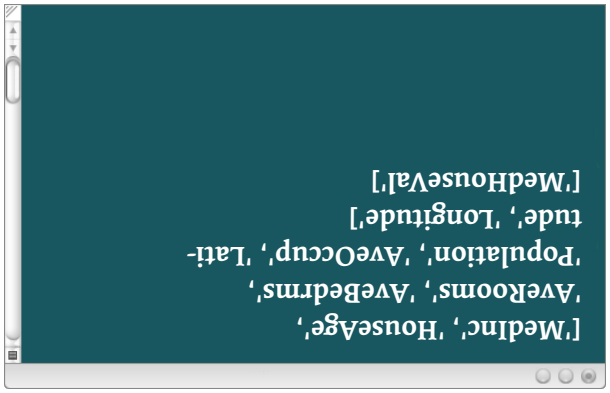
[Lösung]

Hier findest du die fertige Lösung:
Kapitel 2/06-knn-sklearn-regression.ipynb.

Es gibt auch einen Datensatz mit dem Namen **`fetch_california_housing`**, den du von **`sklearn.datasets`** importieren kannst.

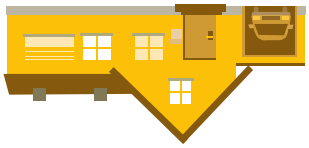
Die Struktur ist wieder dieselbe und wir verwenden diesmal nicht den **`KNeighborsClassifier`**, sondern **`KNeighborsRegressor`** und fünf Nachbarn.

[Schwierige Aufgabe]
Importiere den Datensatz und lass dir die Merkmale und das Label ausgeben.




```
# California-Housing-Datensatz laden
california = fetch_california_housing()
X = california.data
y = california.target
print(california.feature_names)
print(california.target_names)
```

```
from sklearn.datasets import fetch_california_housing
from sklearn.neighbors import KNeighborsRegressor
from sklearn.preprocessing import StandardScaler
```



Wie du siehst, sind die Merkmale im Datensatz wie folgt aufgebaut:

- **MedInc**: Medianes Einkommen in der Blockgruppe
- **HouseAge**: Durchschnittliches Alter der Häuser in der Blockgruppe
- **AveRooms**: Durchschnittliche Anzahl der Zimmer pro Haushalt
- **AveBedrms**: Durchschnittliche Anzahl der Schlafzimmer pro Haushalt
- **Population**: Bevölkerung der Blockgruppe
- **AveOccup**: Durchschnittliche Anzahl der Bewohner pro Haushalt
- **Latitude**: Geografische Breite der Blockgruppe
- **Longitude**: Geografische Länge der Blockgruppe

[Einfache Aufgabe]
Erstelle den Regressor und trainiere das Modell.

[Lösung]

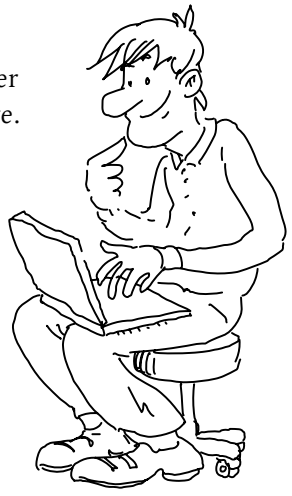
```
knn_regressor = KNeighborsRegressor(n_neighbors=5)
# Normalisieren der Werte nicht vergessen!
scaler = StandardScaler()
X = scaler.fit_transform(X)
knn_regressor.fit(X, y)
```

Nun reicht es, ein Beispielhaus zu verwenden und wieder die **`predict`**-Methode aufzurufen.

```
new_house = np.array([[8.5, 41.0, 6.9, 1.1, 322.0, 2.5, 37.88, -122.23]])
predicted_price = knn_regressor.predict(new_house)
print(f'Der Preis für das neue Haus ist: ${predicted_price[0] * 100000:.2f}')
```

```
Der vorhergesagte Preis für
das neue Haus ist: $205420.00
```

Nun kannst du nicht nur Cluster erstellen, sondern diese auch für Klassifizierungen oder für Wertvorhersagen benutzen. Deinem Empfehlungssystem steht nichts mehr im Wege.



Vielleicht kann ich hier etwas machen, das mir das nächste Mittagessen vorschlägt.

Und was, wenn der Algorithmus sagt, heute gibt es Dinkelpfannkuchen?

Dann stimmt was nicht.

Sehen wir uns im nächsten Kapitel an, wie du prüfen kannst, wie gut die generierten Cluster sind – und was du damit anfangen kannst.

- Viele Algorithmen arbeiten mit **Abstandsmetriken**, die nicht nur in zwei- oder dreidimensionalen Räumen funktionieren, sondern auch in hochdimensionalen Datenräumen zuverlässig eingesetzt werden können.
- **Clustering**-Algorithmen erstellen Cluster von Datenpunkten in diesem hochdimensionalen Raum, die näher beisammen sind.
- **K-Means**-Daten sollten standardisiert oder normalisiert werden, da diese anfällig für unterschiedliche Skalierungen sind.
- Zur Standardisierung wird der **Z-Score** verwendet.
- **K-Means-Clustering** startet mit zufälligen Positionen für die **Zentroide**.
- Bei K-Means werden die Datenpunkte dem nächsten Cluster zugeordnet – also dem Cluster mit dem geringsten Abstand laut Abstandsmetrik.
- Nach der Zuordnung zu einem Cluster wird beim K-Means der Zentroid neu berechnet und in den Schwerpunkt des Clusters gesetzt. Die Zuordnung beginnt erneut.
- Zur Ermittlung von K bei K-Means kannst du die **Ellenbogenanalyse** oder die **Silhouetten-Analyse** durchführen – dazu mehr im nächsten Kapitel. Der **DBScan-Algorithmus** kann Ausreißer/Anomalien erkennen.
- Der DBScan-Algorithmus ist ebenfalls auf Standardisierung oder Normalisierung angewiesen.
- Im Gegensatz zu K-Means generiert der DBScan-Algorithmus beliebige – auch gebogene – Cluster-Formen.
- Wir benötigen die **Dichteangabe** – die minimale Anzahl der Datenpunkte – sowie den **Radius**, den wir betrachten, als **Hyperparameter** beim DBScan. Die Anzahl der Cluster ergibt sich dann aus der Bildungsregel der Cluster.
- K-Means und DBScan sind zwei Algorithmen aus der Klasse des **unüberwachten Lernens**.
- KNN steht für **K-Nearest-Neighbors**. Also die K nächsten Nachbarn, wobei du den Wert für K wählst – also wie viele Nachbarn du betrachtest.
- KNN fällt unter **überwachtes Lernen**. Du benötigst also bereits fertige Cluster, um diesen anwenden zu können.
- Der KNN-Algorithmus kann sowohl zur Klassifikation als auch zur **Regression** eingesetzt werden.

INHALTSVERZEICHNIS

Kapitel 1: Große Neue Welt

Features

Kapitel 2: Auf gute Nachbarschaft

Abstandsmetriken, K-Means, DBScan und K-Nearest-Neighbor

Kapitel 3: Was uns trennt und verbindet

Clusteranalyse

Seite 141

Kapitel 4: Pflanzenkunde

Entscheidungsbäume

Seite 165

Kapitel 5: Pflanzen im Fitnessstudio

Modellbeurteilungen

Seite 223

Kapitel 6: Wenn du den Wald vor lauter Bäumen nicht siehst

Von Random Forest bis BoostedDecision Trees

Kapitel 7: Schreib, was ich denke!

Levenshtein und N-Gramme

Kapitel 8: Ich denke, also bin ich!

Neuronale Netze

Kapitel 9: Schulbeginn

Wie neuronale Netze lernen

Kapitel 10: Herr Ingenieur Breitfuss

Feature Engineering

Kapitel 11: Von der Schule an die Uni

Neuronale Netze anwenden

Kapitel 12: Ein Bild sagt mehr als tausend Worte

Neuronale Netze und Bilder

Kapitel 13: Heute ist das Gestern von morgen

Zeitreihen

Kapitel 14: Abrakadabra, Text verwandle dich!

Tokenizing und Embedding

Kapitel 15: Transformer auf der Jagd nach Bedeutung

Transformer

Schrödinger programmiert KI

Design & Elektronik

„HÄTTE ES DOCH SOLCHE BÜCHER VOR 20 JAHREN SCHON GEGEBEN!“

Christian Mantey, IT-Dozent

„Überraschend gut! Sehr zu empfehlen!“

Leser-Feedback

„Jetzt bin ich platt. Ich kann nicht aufhören zu lesen.“

c’t zur Schrödinger-Reihe

„Ein neuer Weg bei der Vermittlung von Entwickler-Fachwissen.“

Leser-Feedback

„Das Layout der Seiten ist genial.“

DAS ALLES UND NOCH VIEL MEHR:

- Installation und erste Schritte
- Die Mathematik, mit der Maschinen lernen
- K-nearest Neighbours
- Deep Learning mit Bibliotheken: scikit-learn, TensorFlow und Keras
- Vorhersagen mit Regression
- Entscheidungsbäume (und -wälder)
- Klassifizierung: Hund oder Katze?
- Einen einfachen Chatbot bauen

Schrödinger ist unser Mann fürs Programmieren. Er kann schon was, aber mit künstlicher Intelligenz hat er sich noch nicht befasst. Zum Glück hat er einen Kumpel, der für alles ein Beispiel findet, das du nicht wieder vergisst.

Vom Feinsten! Endlich verstehen, wie KI funktioniert. Gemeinsam mit Schrödinger lernst du alles, was du von K-nearest Neighbours bis hin zum eigenen GPT brauchst. Dabei wird getüftelt, erklärt und repariert, bis alles funktioniert – und ihr alles verstanden habt.

SCHRÖDINGER GARANTIERT:

- Gründlicher **Einstieg**
- **Profi-Coding** und Unmengen guter Tipps
- Beispiele und Aufgaben mit **Lösungen**
- Für Einsteiger und Umsteiger **perfekt**

Ein echtes Fachbuch, nur eben ganz anders!

Unsere Autoren: **Bernhard Wurm** und **Sebastian Steininger** unterrichten Informatik und Maschinelles Lernen an der HTL Neuenfelden. Sie haben gemeinsam schon so manches IT-Projekt zum Erfolg geführt; jetzt erklären sie Schrödinger, wie man eine KI programmiert.

Rheinwerk Computing

Gedruckt in Deutschland
Mineralölfreie Druckfarben
Zertifiziertes Papier

Für Windows, Mac und Linux
Programmierung
ISBN 978-3-367-10901-2

€ 49,90 [D] € 51,30 [A]