



# Data Science





# Inhaltsverzeichnis

<b>1. GRUNDLAGEN DER DATA SCIENCE.....</b>	<b>5</b>
<b>1.1. Was ist Data Science und welche Datentypen gibt es?.....</b>	<b>6</b>
1.1.1. Was ist Data Science?.....	6
1.1.2. Praxisbeispiele für Data Science .....	8
1.1.3. Welche Datentypen gibt es?.....	9
<b>1.2. Information und Entropie.....</b>	<b>11</b>
1.2.1. Was ist Information? .....	11
1.2.2. Was ist Entropie? .....	13
1.2.3. Herleitung der Entropie .....	14
1.2.4. Eine Formel für Information.....	19
1.2.5. Informationsgewinn von Daten .....	20
<b>1.3. Statistik .....</b>	<b>23</b>
1.3.1. Modus, Median und Mittelwert.....	24
1.3.2. Varianz und Standardabweichung .....	28
1.3.3. Kovarianz und Korrelation .....	32
<b>1.4. Die pandas-Bibliothek .....</b>	<b>35</b>
1.4.1. Dataframes.....	36
1.4.2. Excel Dateien einlesen .....	38
1.4.3. CSV- und JSON-Files einlesen.....	39
1.4.4. Datenreinigung.....	43
1.4.5. Skalierung .....	49
<b>1.5. Die Matplotlib-Bibliothek .....</b>	<b>53</b>
1.5.1. Plots erstellen.....	53
1.5.2. Subplots erstellen.....	58
1.5.3. Streudiagramme erstellen .....	60
1.5.4. Histogramme erstellen.....	61
1.5.5. Kuchendiagramme erstellen .....	62
<b>1.6. Algorithmen-Komplexität .....</b>	<b>64</b>
1.6.1. Was ist die Komplexität eines Algorithmus? .....	64
1.6.2. Arten von Komplexität .....	67
1.6.3. Komplexitätsklassen.....	69



1.6.4. Komplexitätsklassen im Vergleich .....	79
<b>2. MACHINE LEARNING.....</b>	<b>82</b>
<b>2.1. Was ist Machine Learning?.....</b>	<b>83</b>
2.1.1. Supervised Learning.....	84
2.1.2. Unsupervised Learning.....	87
2.1.3. Reinforcement Learning .....	88
2.1.4. Trainings- und Testdaten .....	89
2.1.5. Over- und Underfitting .....	89
<b>2.2. Lineare Regression.....</b>	<b>90</b>
2.2.1. Die Regressions-Gerade.....	93
2.2.2. Das Bestimmtheitsmaß .....	96
2.2.3. Programmierung einer linearen Regression mit dem scipy-Modul.....	99
<b>2.3. Multiple Regression .....</b>	<b>102</b>
2.3.1. Was ist multiple Regression? .....	102
2.3.2. Programmierung einer multiplen Regression.....	105
2.3.3. Multiple Regression mit kategorischen Daten.....	108
<b>2.4. Polynom-Regression .....</b>	<b>111</b>
2.4.1. Was ist Polynom-Regression? .....	111
2.4.2. Programmierung einer Polynom-Regression.....	115
2.4.3. K-fache Kreuzvalidierung.....	119
<b>2.5. Der K-nearest-neighbor-Algorithmus.....</b>	<b>123</b>
2.5.1. Was ist der K-nearest-neighbor-Algorithmus? .....	123
2.5.2. Metriken zur Abstandsbestimmung.....	128
2.5.3. Algorithmen zur Bestimmung der nächsten Nachbarn.....	131
2.5.4. Programmierung des KNN mit scikit-learn .....	140
<b>2.6. Die logistische Regression .....</b>	<b>143</b>
2.6.1. Was ist die logistische Regression? .....	144
2.6.2. Logistische Regression und Wahrscheinlichkeit .....	145
2.6.3. Die Maximum-Likelihood-Methode .....	150
2.6.4. Interpretation der Parameter.....	153
2.6.5. Multinomiale logistische Regression.....	156
2.6.6. Programmierung der logistischen Regression .....	158
<b>2.7. Decision Trees.....</b>	<b>161</b>
2.7.1. Was ist ein Decision Tree? .....	161
2.7.2. Aufbau eines Decision Trees.....	163
2.7.3. Training des Decision Trees.....	167



2.7.4. Regression Trees .....	178
2.7.5. Programmierung eines Decision Trees.....	183
2.7.6. Vor- und Nachteile eines Decision Trees.....	188
<b>2.8. Bagging und Boosting.....</b>	<b>190</b>
2.8.1. Bias und Varianz.....	191
2.8.2. Ensemble-Methoden .....	197
2.8.3. Programmierung eines Random Forest .....	201
<b>2.9. K-Means Clustering.....</b>	<b>202</b>
2.9.1. Unsupervised-Learning-und-Clustering-Algorithmen .....	203
2.9.2. K-Means-Clustering .....	205
2.9.3. Anwendungen des K-Means-Algorithmus.....	210
2.9.4. Programmierung des K-Means-Algorithmus.....	210
2.9.5. Die richtige Anzahl an Clustern bestimmen .....	212
<b>2.10. Hierarchisches Clustering .....</b>	<b>215</b>
2.10.1. Agglomeratives Clustering .....	216
2.10.2. Divisive Clustering .....	218
2.10.3. Vor- und Nachteile des hierarchischen Clustering .....	218
2.10.4. Die Fusionierungsalgorithmen.....	219
2.10.5. Dendrogramme.....	225
2.10.6. Programmierung des hierarchischen Clustering .....	227
<b>3. NEURAL NETWORKS.....</b>	<b>230</b>
<b>3.1. Neuronen.....</b>	<b>231</b>
<b>3.2. Signalübertragung in einem Neural Network.....</b>	<b>237</b>
<b>3.3. Signalübertragung in einem drei-schichtigen Neural Network.....</b>	<b>241</b>
<b>3.4. Training des Neural Networks .....</b>	<b>243</b>
3.4.1. Fehler-Backpropagierung .....	244
3.4.2. Gradienten-Verfahren und Fehlerfunktion .....	246
3.4.3. Aktualisierung der Gewichte.....	250
3.4.4. Vorbereitung der Daten.....	254
<b>3.5. Programmierung eines Neural Networks.....</b>	<b>258</b>
<b>Dieses Kapitel enthält ausschließlich Übungsaufgaben. Diese sind in der Kursstrecke auf der Lernplattform direkt hinterlegt.....</b>	<b>258</b>
<b>3.6. Neural Networks mit PyTorch.....</b>	<b>258</b>
<b>4. DATA-SCIENCE-METHODEN.....</b>	<b>279</b>



<b>4.1. Erzeugung von synthetischen Daten.....</b>	<b>279</b>
4.1.1. Python Faker.....	281
<b>4.2. Datenvisualisierung – Geographische Daten .....</b>	<b>282</b>
4.2.1. Grundlagen.....	282
4.2.2. Projektionen.....	284
<b>4.3. Matplotlib Stylesheets .....</b>	<b>287</b>
<b>4.4. 3-dimensionale Plots.....</b>	<b>288</b>



## 1. Grundlagen der Data Science

### Lernziele für dieses Kapitel:

**Grobziel:** Die Lernenden können die Grundlagen der Data Science anwenden und einfache Datenanalysen ausführen.

Feinziele (ca. 3-5)	Inhalte
1. Die Lernenden können erklären, womit sich die Data Science beschäftigt, und können die verschiedenen Datentypen beschreiben.	<ul style="list-style-type: none"><li>• Data Science als Überlappung von Mathematik, Informatik und fachspezifischem Wissen</li><li>• Fokus auf Analyse und Vorhersage</li><li>• Auditiv, visuelle und Textdaten</li><li>• Einteilung in kategorisch und numerisch</li><li>• Unteraufteilung in nominal, ordinal, diskret und kontinuierlich</li></ul>
2. Die Lernenden können den Informationsgewinn von Daten mit Hilfe der Entropie ermitteln.	<ul style="list-style-type: none"><li>• Information und Entropie</li><li>• Informationsgewinn von Daten</li></ul>
3. Die Lernenden können die grundlegenden Begriffe der Statistik verdeutlichen.	<ul style="list-style-type: none"><li>• Modus, Median und Mittelwert</li><li>• Varianz und Standardabweichung</li></ul>



	<ul style="list-style-type: none"><li>• Kovarianz und Korrelation</li></ul>
4. Die Lernenden können mit Hilfe der pandas-Bibliothek Dateien visualisieren.	<ul style="list-style-type: none"><li>• Excel</li><li>• CSV</li><li>• JSON</li></ul>
5. Die Lernenden können mit Matplotlib Daten visualisieren	<ul style="list-style-type: none"><li>• Streudiagramme</li><li>• Histogramme</li><li>• Kuchendiagramme</li></ul>
6. Die Lernenden können die Effizienz von Algorithmen einschätzen.	<ul style="list-style-type: none"><li>• Lineare Komplexität</li><li>• Quasilineare Komplexität</li><li>• Quadratische Komplexität</li><li>• Logarithmische Komplexität</li></ul>

## 1.1. Was ist Data Science und welche Datentypen gibt es?

In diesem Kapitel beschäftigen wir uns als erstes mit der Frage, was Data Science eigentlich ist, und werden feststellen, dass es die Überlagerung von Mathematik, Informatik und branchenspezifischem Fachwissen ist. Danach sehen wir uns ein paar Anwendungsgebiete der Data Science an, um ihre Wichtigkeit in jedem Bereich der heutigen Zeit hervorzuheben. Anschließend werden wir uns mit den verschiedenen Datentypen befassen und lernen, wie diese kategorisiert werden können.

### 1.1.1. Was ist Data Science?

Bevor wir uns gleich zu Beginn auf die Data Science und ihre zahlreichen Methoden stürzen, befassen wir uns mit den Fragen, was Data Science eigentlich ist und wo sie verwendet wird.



Also, was ist Data Science?

Bei Data Science handelt es sich – wie der Name schon sagt – um die Wissenschaft der Daten. Genauer gesagt geht es darum, wissenschaftliche Methoden anzuwenden, um Daten besser zu verstehen.

Man kann die Data Science als eine Schnittstelle zwischen den drei Bereichen Mathematik, Informatik und dem Fachbereich, dessen Daten besser verstanden werden sollen, einordnen. Diese variablen Fachbereiche können alles Mögliche sein. Es kann sich dabei um biologische Daten aus einem Labor oder um Konsumentendaten aus einem E-Commerce-Store handeln.

In der Data Science nehmen wir mathematische Methoden aus der Statistik, implementieren sie auf einem Computer und lassen diesen die zahlreichen Daten analysieren. Um diese Daten allerdings zu verstehen, braucht es Wissen über den entsprechenden Fachbereich.

Ohne Kenntnisse der Biologie oder ohne Grundwissen im Marketing wird es trotz der eleganten mathematischen Methoden schwierig, einen Sinn aus den Daten zu ziehen.

Wie sieht dieses Verstehen eines Datensatzes aber aus? Womit befasst sich die Data Science eigentlich genau?

Grob kann man die Arbeit als Datenwissenschaftler\*in in drei Bereiche einteilen:

1. Man befasst sich mit der Analyse von großen Datenmengen und versucht, wertvolle Informationen aus diesen herauszufiltern.
2. Man sucht nach Anomalien in den vorliegenden Daten.
3. Man versucht, anhand der eingehenden Analyse von historischen Daten zukünftige Ereignisse vorherzusagen. Dadurch kann man dann einem Unternehmen beispielsweise Handlungsempfehlungen für die Zukunft geben (unter der Voraussetzung, dass man Branchenwissen besitzt). Außerdem werden Vorhersagen verwendet, um bereits



bestehende Prozesse zu optimieren und im besten Fall auch zu automatisieren.

Um das ganze Abstrakte mal zu verdauen, schauen wir uns ein paar Beispiele an und sehen, wo Data Science eigentlich verwendet wird.

## 1.1.2. Praxisbeispiele für Data Science

### Marketing

Das wohl bekannteste Einsatzbeispiel für die Data Science findet sich im Marketing; genauer gesagt im E-Commerce-Marketing. Dort werden Data-Science-Methoden vor allem zum personalisierten Zuschnitt von Marketing-Strategien angewendet. Bei der riesigen Produktpalette der heutigen E-Commerce-Stores hätte ein klassisches Marketing-Team es ganz schwer, das Produkt richtig auszuwählen, das dem Kunden/Kundinnen als erstes präsentiert werden soll. Stattdessen werden historische Daten zum Transaktionsverhalten der Kunden/Kundinnen, ihrer Demographie und ihres Verhaltens von Machine-Learning-Algorithmen analysiert, um jedem Kunden/jeder Kundin ein individuell auf ihn/sie abgestimmtes Produkt zeigen zu können.

### Gesundheitswesen

Weniger populär dagegen ist der Einsatz der Data Science in der Gesundheitsbranche. Hier führt man mit den behandelten Patienten/Patientinnen Ähnlichkeitsstudien durch, um die Medikation zu verbessern.

### Logistik

Ein weiteres Anwendungsgebiet der Data Science findet sich in der Logistik. Hier ist es eine besondere Herausforderung, die Planung und Organisation von Auslieferungen effizient zu planen. Dabei spielen zahlreiche Faktoren eine Rolle. Um die Fahrtzeiten beispielsweise optimieren zu können,



benötigt man unter anderem Daten zu den Witterungsbedingungen, Fahrzeiten, Personalkosten oder dem Verkehr.

### **1.1.3. Welche Datentypen gibt es?**

Wir wissen jetzt, dass es in der Data Science darum geht, Daten mithilfe von mathematischen Methoden, Computer-Algorithmen und spezifischem Branchenwissen zu analysieren und – im Falle von historischen Daten – im besten Fall die Zukunft vorhersehen zu können. Die Frage, die sich jetzt stellt, ist: Wie sehen diese Daten aus?

Einfach gesagt lassen sich die Typen von Daten, mit denen Datenwissenschaftler\*innen in erster Linie zu tun haben, in drei Kategorien einteilen:

- Auditive Daten
- Visuelle Daten
- Textdaten

Auditive Daten sind alle Arten von Daten, die unseren Hörsinn beeinflussen, also Tonaufzeichnungen. Hier wird die Data Science hilfreich, um diese dahingehend zu analysieren, dass ein Computer-Programm gesprochenen Text versteht und Befehle befolgen kann, ohne dass sie dem Programm in einer spezifischen Programmiersprache beigebracht wurden. Bei visuellen Daten handelt es sich um Bilder oder Videos. Data-Science-Methoden helfen hier beispielsweise bei der Gesichtserkennung. Mit visuellen und auditiven Daten werden wir uns später im Kurs noch befassen. Jetzt zu Beginn wollen wir uns mit Textdaten auseinandersetzen.

Auch die Textdaten lassen sich noch einmal in zwei unterschiedliche Kategorien aufteilen:



## Numerische Daten

Wenn wir von numerischen Daten sprechen, dann meinen wir Daten, die durch Zahlenwerte ausgedrückt werden können. Hier ist zu beachten, dass nicht jede Ziffernfolge gleich numerisch ist. Postleitzahlen oder Hausnummern beispielsweise haben keinen eigentlichen Zahlenwert. Auch numerische Daten fallen in zwei Kategorien – **diskrete und kontinuierliche Zahlenwerte.**

Diskrete Daten können nur bestimmte numerische Werte haben. Solche Daten entstehen eigentlich immer bei Zählungen. Wenn es also um die Anzahl der Kunden/Kundinnen pro Tag geht oder die Anzahl der Downloads von einer Website, dann sprechen wir von numerisch diskreten Daten. Auf der anderen Seite sind dann die numerisch kontinuierlichen Daten solche, die jeden beliebigen Zahlenwert annehmen können. Diese Zahlenwerte sind nur durch die Genauigkeit der Messmethode eingeschränkt. Beispiele für numerisch kontinuierliche Daten wären Körpergröße, Gewicht oder Fahrtgeschwindigkeit eines LKWs.

## Kategoriale Daten

Die andere Sorte von Textdaten neben den numerischen Werten sind die sogenannten kategorischen Daten. Diese haben im Allgemeinen keinen Zahlenwert, das heißt sie können nicht gemessen werden. Kategoriale Daten repräsentieren vor allem Variablen, die in Gruppen eingeteilt werden können – die kategorisiert werden können. Die oben erwähnten Beispiele Postleitzahl und Hausnummer beispielsweise sind kategoriale Daten, da sie die Adressen in Kategorien und Unterkategorien einteilen. Wie bei den numerischen Daten gibt es auch bei den kategorischen Daten zwei Typen, mit denen man in der Data Science immer wieder zu tun hat – mit **nominalen** und **ordinalen Daten.**



Beispiele für nominale Daten sind Geschlecht, Altersklasse oder Nationalität. Bei diesen Daten spielt die Reihenfolge keine Rolle.



Bei ordinalen Daten dagegen schon, diese haben eine gewisse Ordnung inne. Ordinale Daten haben eine logische Reihenfolge, wie beispielsweise die Kundenzufriedenheit auf einer Skala von 1 bis 5, das akademische Niveau oder das Stadium einer Krankheit. Die Zahlen dienen hier nicht als numerische Werte, sondern um die Daten in geordnete Kategorien einzuteilen.

## 1.2. Information und Entropie

In diesem Kapitel beschäftigen wir uns mit dem Begriff der Information und finden heraus, wie wir den Informationsgehalt von Datensätzen berechnen können. Das ist sehr engverbunden mit der sogenannten Entropie, die ihren Ursprung in der Physik hatte, 1948 aber vom Mathematiker Claude Shannon uminterpretiert wurde, um die Informationstheorie zu gründen. Wir werden lernen, woher die Formel zur Entropieberechnung kommt und wie man sie konkret auf Datensätze anwendet. Anschließend werden wir das Gelernte nutzen, um den sogenannten Informationsgewinn von Datensätzen kennenzulernen – ein wichtiges Werkzeug, das später im Machine Learning angewendet wird, um einen Decision Tree optimal zu konstruieren.

### 1.2.1. Was ist Information?

Warum analysieren wir Daten? Wir wollen sie verstehen und zukünftige Ereignisse vorhersehen können. Die Analyse der Daten versorgt uns also mit Informationen. Die Frage, die wir uns jetzt stellen können, ist: Was ist eigentlich Information?

Was passiert, wenn wir die Information erhalten, dass das Wetter heute schlecht ist? Oder dass die Erde der dritte Planet von der Sonne aus gezählt ist? Unser Wissen wird erweitert.



Bevor wir wussten, dass das Wetter heute schlecht wird, waren wir uns nicht sicher, wie es sich entwickeln würde. Genauso verhält es sich bei der Erde und ihrer Position im Sonnensystem. Bevor wir wussten, dass sie die dritte Position hinter der Sonne einnimmt, waren wir uns dessen nicht sicher. Information ist also ein Maß dafür, wie unsicher wir uns einer bestimmten Sache sind. Je weniger Information wir über eine Sache haben, umso unsicherer sind wir uns dessen. Information ist also alles, was unsere Unsicherheit reduziert.

Um das Konzept von Information etwas besser zu verstehen, schauen wir uns ein einfaches Beispiel an. Stellen wir uns vor, wir hätten eine undurchsichtige Box, in der sich drei Bälle befinden. Einer der Bälle ist rot, die anderen beiden sind blau und wir wissen das.

Die Wahrscheinlichkeit, einen roten Ball zu ziehen, liegt also bei  $1/3$ , während die Wahrscheinlichkeit, einen blauen Ball aus der Kiste zu ziehen, bei  $2/3$  liegt.

Jetzt ziehen wir blind eine Kugel aus der Box und sehen, dass wir die rote Kugel gezogen haben. Dadurch, dass nur noch blaue Kugeln in der Box übrig sind, besteht keine Unsicherheit mehr darüber, welche Kugelfarbe als nächstes gezogen wird. Der Informationsgehalt des Prozesses „Rausziehen der Kugel“ ist also sehr groß.

Legen wir die rote Kugel aber wieder zurück, wiederholen das Experiment und ziehen als erstes eine blaue Kugel, sind wir uns sehr unsicher, welche Kugel wir als nächstes erwischen. Zwei Bälle sind noch übrig, ein blauer und ein roter. Jeder kann mit der gleichen Wahrscheinlichkeit gezogen werden. Der Informationsgehalt des Prozesses war also gering, er hat unsere Unsicherheit über den Inhalt der Box tatsächlich noch vergrößert.



Jetzt sind Informationstheoretiker\*innen natürlich daran interessiert, den Informationsgehalt von Prozessen quantitativ zu erfassen – irgendwie will man ja vergleichen können, welcher Prozess mehr Information liefert und welcher weniger. Damit wir also Information quantitativ beschreiben können, müssen wir uns mit der Unsicherheit befassen. Und die Unsicherheit beschreibt man mit der sogenannten **Entropie**.

### 1.2.2. Was ist Entropie?

Wir haben bereits gesehen, dass Entropie irgendwie mit der Unsicherheit und dem Informationsgehalt zusammenhängt, aber wie genau das in der Data Science ist und wo sie angewendet wird, schauen wir uns in diesem Abschnitt an.

Vereinfacht gesagt ist die Entropie ein Maß dafür, welchen Informationsgehalt eine Menge an Daten enthält. Präziser ausgedrückt geht es hierbei um die Reinheit der Daten. Je höher die Entropie eines Datensatzes ist, umso unreiner sind die Daten und umso geringer auch die in ihnen enthaltene Information.

Ursprünglich stammt der Begriff der Entropie aus der Physik, wo es allerdings um die Unreinheit von Gas- und Flüssigkeitsgemischen geht. Das beste Beispiel dafür wäre ein Kirsch-Bananen-Saft, der unten gelb und oben rot ist. Bei einem solchen Getränk würde es sich um ein reines Gemisch handeln, weil die beiden Farbanteile Gelb und Rot sehr gut und deutlich voneinander zu unterscheiden sind.

Mischt man das Getränk allerdings durch, dann verschwimmen die Grenzen zwischen Gelb und Rot und die Unreinheit steigt.

Den Begriff der physikalischen Entropie hat der Mathematiker Claude Shannon schließlich 1948 genutzt, um mit ihr in seiner Arbeit „A



“Mathematical Theory of Communication“ das Fundament für die Informationstheorie zu legen.

Heute findet die Entropie ihre Anwendung in der Data Science bei zahlreichen Machine-Learning-Methoden, wie wir im Verlauf dieses Kurses sehen werden. Steht man beispielsweise vor der Aufgabe, Datensätze mit bestimmten Features zu kategorisieren, fragt man sich: Mit welchem Feature sollte ich meine Kategorisierung starten? Hierbei hilft die Entropie, denn mit ihr kann man den Informationsgewinn eines jeden Features berechnen und diese miteinander vergleichen.

Ein anderes Anwendungsgebiet der Entropie liegt in der Optimierung von Machine-Learning-Algorithmen und neuronalen Netzen. Mit der sogenannten Kreuzentropie lässt sich die Qualität des Machine-Learning-Algorithmus überprüfen, wodurch dieser im Verlauf des Prozesses besser trainiert werden kann. Nun, da wir wissen, wo die Entropie eigentlich herkommt und wofür wir sie in der Data Science eigentlich brauchen, schauen wir uns an, wie wir sie berechnen können.

### 1.2.3. Herleitung der Entropie

Um zu verstehen, wie man die Entropie eines Datensatzes mit einer Formel berechnen kann, schauen wir uns den Begriff der „Überraschung“ an. Am besten versteht man diesen mit einem Gedankenexperiment.

Wie zuvor im ersten Abschnitt geht es um Kisten mit Bällen drin. Wir stellen uns drei Boxen vor, jede mit roten und blauen Bällen.

In Box Nummer 1 befinden sich 6 blaue und 1 rote Kugel.

In Box Nummer 2 befinden sich 6 rote und 1 blaue Kugel.

Und in Box Nummer 3 sind nur 6 blaue Kugeln.



Wenn wir uns mit diesem Wissen Box Nummer 1 zuwenden und blind hineingreifen würden, wäre es eine große Überraschung von uns, eine rote Kugel herauszuziehen. Immerhin wissen wir ja, dass in Box Nummer 1 sechsmal so viele blaue wie rote Kugeln sind, die Wahrscheinlichkeit, eine rote Kugel zu ziehen, war sehr klein und lag nur bei 1/7.

Analog verhält es sich mit Box Nummer 2, doch diesmal umgekehrt. Würde man jetzt blind hineingreifen und eine blaue Kugel ziehen, wäre man ziemlich überrascht, immerhin sind sechsmal mehr rote Kugeln in der Box.

Anhand dieser Beispiele können wir sehen, dass die Überraschung proportional zum Inversen der Wahrscheinlichkeit ist.

$$U(E) \propto \frac{1}{P(E)}$$

Hierbei ist  $U(E)$  die Überraschung, dass ein Ereignis  $E$  passiert, und  $P(E)$  ist die Wahrscheinlichkeit, dass Ereignis  $E$  eintritt.

Je geringer die Wahrscheinlichkeit eines Ereignisses, desto größer die Überraschung, wenn es dann am Ende eintritt. Umgekehrt, je wahrscheinlicher etwas ist, desto weniger überrascht es, wenn es tatsächlich passiert.

Allerdings ist das Inverse der Wahrscheinlichkeit alleine keine gute Methode, um das Maß an Überraschung bei einem Ereignis zu beschreiben. Das wird klar, wenn wir uns Box Nummer 3 anschauen.

Bei Box Nummer 3 sieht es ein bisschen anders aus. Mit der Information, dass nur blaue Kugeln in dieser Kiste sind, wären wir überhaupt nicht überrascht, wenn wir eine blaue Kugel ziehen.



Unsere Überraschung wäre gleich 0. Die Wahrscheinlichkeit, eine Kugel herauszuziehen, wäre 1, und somit wäre das Inverse davon auch 1. Wir sehen, dass das Inverse der Wahrscheinlichkeit alleine nicht reicht, um die Überraschung mathematisch korrekt auszudrücken, da für ein 100 % eintreffendes Ereignis die Überraschung nicht 0, sondern 1 ist.

Stattdessen können wir den Logarithmus des Inversen der Wahrscheinlichkeit nehmen:

$$U(E) = \log\left(\frac{1}{P(E)}\right)$$

Wenn wir hier die Wahrscheinlichkeit  $P(E)=1$  für ein sicher eintreffendes Ereignis einsetzen, erhalten wir:

$$\log\left(\frac{1}{1}\right) = \log(1) = 0$$

In der Regel wird in der Informationstheorie der Logarithmus zur Basis 2 verwendet, wir haben also unsere Formel für die Überraschung:

$$U(E) = \log_2\left(\frac{1}{P(E)}\right)$$

Wie hängt das Ganze jetzt mit dem Begriff der Entropie zusammen? Dafür schauen wir uns ein weiteres Gedankenexperiment an.

Nehmen wir an, wir hätten eine Münze, die von den Wahrscheinlichkeiten her nicht ganz so ausgeglichen ist. Anstatt, dass die Münze, nachdem sie geworfen wurde, mit der gleichen Wahrscheinlichkeit entweder auf Kopf oder Zahl landen kann, haben unsere Daten etwas anderes ergeben. 100-



mal wurde die Münze geworfen und in 90 Fällen ist sie mit Kopf nach oben gelandet, in den übrigen 10 hat die Zahl nach oben geschaut. Die Wahrscheinlichkeit, dass die Münze Kopf zeigt, ist also 0.9 und für Zahl 0.1. Die Überraschung, dass die Münze Kopf zeigt, wäre:

$$U(E = \text{Kopf}) = \log_2\left(\frac{1}{0.9}\right) = 0.15$$

Auf der anderen Seite ist die Überraschung, Zahl zu bekommen:

$$U(E = \text{Zahl}) = \log_2\left(\frac{1}{0.1}\right) = 3.32$$

Wir werfen die Münze nochmal, diesmal nur dreimal hintereinander. Die ersten beiden Male zeigt sie wieder Kopf, beim dritten Mal Zahl. Die Wahrscheinlichkeit, dass dieses Ereignis eintritt, ist:

$$P(E) = 0.9 \cdot 0.9 \cdot 0.1 = 0.081$$

Die Überraschung, dass genau dieses Ereignis stattfindet – zweimal Kopf und einmal Zahl – wäre:

$$\begin{aligned} U(E) &= \log_2\left(\frac{1}{0.9 \cdot 0.9 \cdot 0.1}\right) = \log_2(1) - (\log_2(0.9) + \log_2(0.9) + \log_2(0.1)) = \\ &0.15 + 0.15 + 3.32 = 3.62 \end{aligned}$$

Wir sehen also, dass die Überraschung dieser verketteten Ereignisse gleich der Summe der einzelnen Überraschungen ist.

Jetzt wollen wir die Münze wieder 10-mal werfen, doch diesmal berechnen wir vorher die gesamte Überraschung, die das hundertmalige Werfen der Münze mit sich bringen wird.



Die Wahrscheinlichkeit, dass die Münze auf Kopf landen wird, ist 0.9, und bei 10 Würfen erwarten wir also die Münze in 9 Fällen auf Kopf. Das multiplizieren wir mit der Überraschung 0.15 und dann erhalten wir die gesamte, zu erwartende Überraschung für Kopf. Das Gleiche machen wir für Zahl, wo es 0.1 multipliziert mit 100 multipliziert mit 3.32 ist. Die gesamte erwartete Überraschung, die uns bei diesem Experiment erwartet, liegt bei:

$$U(E = 100) = 0.9 \cdot 100 \cdot 0.15 + 0.1 \cdot 100 \cdot 3.32 = 46.7$$

Wenn wir diese gesamte erwartete Überraschung durch die Anzahl der Würfe (100) teilen, erhalten wir die durchschnittliche, zu erwartende Überraschung. Diese ist dann die sogenannte Entropie H(E):

$$H(E) = \frac{0.9 \cdot 100 \cdot 0.15 + 0.1 \cdot 100 \cdot 3.32}{100} = 0.47$$

Als konkrete Formel ausgedrückt lässt sich die Entropie folgendermaßen schreiben:

$$H(E) = \sum_i p_i \log_2 \left( \frac{1}{p_i} \right)$$

Diese Formel können wir noch umschreiben:

$$H(E) = \sum_i p_i \log_2 \left( \frac{1}{p_i} \right) = \sum_i p_i (\log_2(1) - \log(p_i)) = - \sum_i p_i \log_2(p_i)$$

Das ist auch die Formel, die man normalerweise in Sachbüchern zur Informationstheorie oder der Data Science findet.



## 1.2.4. Eine Formel für Information

Kommen wir wieder zum Anfang zurück und befassen uns mit einer konkreten Definition des Wortes Information. Mit dem Wissen um die Entropie können wir die Information, die uns ein bestimmtes Ereignis liefert, als die Differenz der Entropien vor und nach dem Ereignis definieren.

$$I(E) = H(\text{vorher}) - H(\text{nachher})$$

Ist die Entropie nachher geringer, wird der Informationsgehalt des Ereignisses positiv sein. Ist im anderen Fall die Entropie nach dem Ereignis größer, dann ist der Informationsgehalt dieses Ereignisses negativ – es ist Information verloren gegangen.

Schauen wir uns das Ganze einfach an einem Beispiel mal an.

Nehmen wir wieder die Box vom Anfang, in der eine rote Kugel und zwei blaue Kugeln waren. Die Entropie dieses Zustands ist:

$$H\left(\frac{1}{3}, \frac{2}{3}\right) = -\frac{1}{3} \log_2 \frac{1}{3} - \frac{2}{3} \log_2 \frac{2}{3} \approx -\frac{1}{3} \cdot (-1.585) - \frac{2}{3} \cdot (-0.585) = 0.918$$

Jetzt ziehen wir eine rote Kugel zuerst raus und lassen damit zwei blaue Kugeln in der Kiste übrig. Nun ist die Wahrscheinlichkeit, eine rote Kugel zu ziehen, 0 (es ist ja keine mehr übrig). Für die blaue dagegen ist die Wahrscheinlichkeit 1. Die Entropie nach dem Ziehen der roten Kugel ist also:

$$H(0, 1) = -0 \log_2 0 - 1 \log_2 1 = 0 + 0 = 0$$



Da nur blaue Kugeln übrig sind, ist die erwartete Überraschung, eine herauszuziehen, gleich 0. Das Maß an Information, dass dieses Ereignis also mit sich bringt, beträgt:

$$0.918 - 0 = 0.918$$

Die Information ist also maximal, mehr hätte man aus dem Ziehen der roten Kugel nicht erfahren können. Spulen wir das Experiment wieder auf den Anfang zurück und wiederholen es mit allen drei Kugeln in der Box. Diesmal ziehen wir allerdings eine blaue Kugel zuerst raus, womit eine rote und eine blaue Kugel übrig bleiben. Die Entropie vor dem Herausziehen bleibt gleich, die Entropie danach ist:

$$H\left(\frac{1}{2}, \frac{1}{2}\right) = -\frac{1}{2} \log_2 \frac{1}{2} - \frac{1}{2} \log_2 \frac{1}{2} = -\frac{1}{2}(-1) - \frac{1}{2}(-1) = \frac{1}{2} + \frac{1}{2} = 1$$

Berechnen wir nun die Differenz zur Entropie vor dem Ereignis, erhalten wir:

$$0.918 - 1 = -0.082$$

Der Informationsgehalt ist negativ! Das liegt daran, dass durch das Ziehen der blauen Kugel Information verloren gegangen ist. Da die blaue Kugel zuerst gezogen wurde, hat sich die Unsicherheit über den Ausgang eines erneuten Ziehens erhöht. Positive Informationsgehalte repräsentieren also einen Zuwachs an Wissen, negative Werte stehen für einen Verlust von Information.

## 1.2.5. Informationsgewinn von Daten

Datensätze lassen sich im Allgemeinen beschreiben als Klassen mit bestimmten Features. Beispielsweise könnten wir einen Datensatz zu den



Besuchenden eines Online-Stores haben. Zu allen Besuchenden wurde die Verweildauer, der Herkunftsort und das Alter festgehalten. Die Besuchenden lassen sich einteilen in Käufer\*innen und Nicht-Käufer\*innen. Die Besuchenden bezeichnet man als Klassen und die Verweildauer, den Herkunftsort sowie das Alter als sogenannte Features.

Im Machine Learning stellt man sich oft der Aufgabe, in einem Test-Datensatz ein Muster zu erkennen und so beispielsweise einen Zusammenhang herzustellen zwischen der Verweildauer und dem Kaufwillen, oder zwischen dem Herkunftsor und dem Kaufwillen, oder zwischen dem Alter und dem Kaufwillen.

Welches Feature am besten geeignet ist, um eine Kategorisierung der Besuchenden vorzunehmen, findet man mit dem sogenannten Informationsgewinn eines jeden Features heraus.

Wir müssten in unserem Beispiel also den Informationsgewinn für die Verweildauer, den Herkunftsor und das Alter berechnen und anschließend miteinander vergleichen.

Ähnlich wie die Entropie ist auch der Informationsgewinn als der erwartete Informationsgehalt definiert, den uns eine Aufteilung nach einem bestimmten Feature liefert.

$$IG(E) = \sum_i p_i(E) \cdot I(E)$$

Wie berechnen wir den Informationsgewinn eines Features aber konkret? Schauen wir uns dafür folgenden, allgemein gehaltenen Datensatz S an:



$a$	$b$	class
0	0	positive
0	1	positive
1	0	negative
1	1	positive
0	0	negative

Es gibt zwei Klassen an Objekten – positiv und negativ (in der Regel kann es auch mehrere Klassen geben). Und zu den Klassen sind auch zwei Features festgehalten, nämlich  $a$  und  $b$ , beide nehmen nur binäre Werte an, also entweder 0 oder 1. Möchten wir nun den Informationsgewinn des Features  $a$  mit dem von Feature  $b$  vergleichen, müssen wir zuerst die Entropie der Klassen zu Beginn berechnen:

$$H(S) = -\frac{2}{5} \log_2 \frac{2}{5} - \frac{3}{5} \log_2 \frac{3}{5} = 0.971$$

Nun teilen wir die Klassen anhand der Features auf, beginnend mit Feature  $a$ . Wenn Feature  $a$  den Wert 0 hat, dann gehören dazu zwei positive Objekte und ein negatives Objekt. Im Fall, dass das Feature  $a$  den Wert 1 hat, gehören ein positives und ein negatives Objekt dazu.

$$S | a = 0 : \begin{bmatrix} 2 \text{ positive objects} \\ 1 \text{ negative object} \end{bmatrix} \quad S | a = 1 : \begin{bmatrix} 1 \text{ positive object} \\ 1 \text{ negative object} \end{bmatrix}$$

Die Entropie für die beiden Fälle oben ist:

$$H(S|a = 0) = H\left(\frac{2}{3}, \frac{1}{3}\right) = 0.918$$



$$H(S|a = 1) = H\left(\frac{1}{2}, \frac{1}{2}\right) = 1$$

Um den Informationsgewinn für dieses Feature zu bestimmen, müssen wir den erwarteten Informationsgehalt für die Aufteilung nach dem Feature a bestimmen.

$$\begin{aligned} IG(a) &= P(a = 0) \cdot I(a = 0) + P(a = 1) \cdot I(A = 1) = \\ &P(a = 0)(H(S) - H(S|a = 0)) + P(a = 1) \cdot (H(S) - H(S|a = 1)) = \\ &(P(a = 0) + P(a = 1))H(S) - P(a = 0)H(S|a = 0) - P(a = 1)H(S|a = 1) = \\ &H(S) - P(a = 0)H(S|a = 0) - P(a = 1)H(S|a = 1) = \\ &0.971 - 0.6 \cdot 0.918 - 0.4 \cdot 1 = 0.0202 \end{aligned}$$

Aufgabe: Bestimme den Informationsgewinn für das Feature b.

Lösung:

$$\begin{aligned} IG(b) &= H(S) - P(b = 0)(H(S|b = 0) - P(b = 1)H(S|b = 1) = \\ &0.971 - 0.6 \cdot 0.918 - 0.4 \cdot 0 = 0.4202 \end{aligned}$$

Wir sehen also, dass der Informationsgewinn von Feature b höher ist als der von Feature a. Er eignet sich also besser, um die Klassen der einzelnen Objekte vorherzusagen, weil aus diesem Feature mehr Information über die Klasse eines Objektes gewonnen werden kann. Diese Methode, um Features auf ihren Informationsgewinn hin zu analysieren, werden wir später nochmal nutzen, um Decision Trees optimal zu konstruieren.

### 1.3. Statistik

In diesem Kapitel werden wir die wichtigsten statistischen Methoden zur Analyse von Datensätzen kennenlernen. Anfangen werden wir mit dem Modus, dem Median und dem Mittelwert, welche die wichtigsten sogenannten Lagemaße der Statistik sind. Anschließend werden wir zur



Übung und zur Festigung des Wissens für jedes Lagemaß einen Python-Code schreiben, der für beliebige Datenmengen die Werte bestimmen kann. Im zweiten Abschnitt befassen wir uns mit den sogenannten Streumaßen, wobei wir uns die beiden wichtigsten – Varianz und Standardabweichung – anschauen. Wie auch bei den Lagemaßen werden wir hier einen Code schreiben, der uns automatisch das Maß an Streuung in den Daten berechnet. Im dritten und letzten Abschnitt sehen wir uns die Korrelationsmaße an, die beschreiben, wie stark der Zusammenhang zwischen verschiedenen Features eines Datensatzes ist. Hier werden wir zuerst die Kovarianz kennenlernen, die wir dann verwenden, um den Korrelationskoeffizienten zu berechnen. Auch hier wartet zum Abschluss eine Programmierübung, bei der der Korrelationskoeffizient automatisch bestimmt werden soll.

### 1.3.1. Modus, Median und Mittelwert

In diesem Abschnitt beschäftigen wir uns mit den sogenannten Lagemaßen der Statistik, wobei wir uns auf die drei wichtigsten konzentrieren: Modus, Median und Mittelwert. Der Mittelwert ist uns bestimmt schon an der einen oder anderen Stelle über den Weg gelaufen, aber die anderen beiden Begriffe klingen schon etwas komplexer. Tatsächlich sind sie aber recht einfach zu verstehen. Zunächst einmal die Frage: Was ist eigentlich ein Lagemaß?

Mit einem Lagemaß versucht man in einer Datenmenge eine Form von zentraler Tendenz zu untersuchen, also in welche Richtung ein Datensatz tendiert. Sie sind also dafür da, um sich ein erstes grundlegendes Verständnis von einem Datensatz zu machen und die Lage einzuschätzen.

Der Modus ist das einfachste Lagemaß, er ist nämlich einfach der Wert in einer Datenmenge, der am häufigsten vorkommt. Der Median teilt den Datensatz in zwei gleichgroße Abschnitte und der Mittelwert ist einfach



klassisch als Durchschnitt der Werte eines Datensatzes bekannt. Schauen wir uns die Lagemaße etwas mehr im Detail an.

## Der Modus

Der Modus wird auch Modalwert genannt und ist – wie oben bereits erwähnt – der Wert einer Datenmenge, der am häufigsten vorkommt. Dieses Lagemaß wird häufig für kategorische Daten verwendet, egal ob nominale oder ordinale Daten. Der Modus ist auch recht simpel zu bestimmen, man muss einfach nur die unterschiedlichen Merkmalsausprägungen eines Datensatzes zählen und schauen, welche am häufigsten vorgekommen ist. Nehmen wir als Beispiel die Liste von Zahlen:

$$x = \{5, 2, 1, 2, 6\}$$

Der Zahlenwert, der am häufigsten in dieser Liste vorkommt, ist die 2, deswegen ist das auch der Modus der Datenmenge und wir schreiben:

$$x_{mod} = 2$$

Natürlich kann es auch vorkommen, dass eine Datenmenge mehrere Modalwerte hat, falls mehrere Merkmalsausprägungen mehrfach auftauchen.

## Der Median

Der Median – wie wir oben bereits kurz erwähnt haben – teilt eine Datenmenge in zwei gleichgroße Abschnitte. Genauer gesagt sortiert man die Datenmenge der Größe nach und teilt sie dann so auf, dass die Hälfte der Elemente kleiner als der Median ist und die andere Hälfte größer. Der Median ist also der Wert, der genau in der Mitte liegt. Da man, um den Median zu bestimmen, die Datenmenge der Größe nach sortieren muss, macht er für kategorisch nominale Daten weniger Sinn und kann nur bei ordinalen oder numerischen Daten bestimmt werden. Einer der Vorteile des



Medians liegt darin, dass extreme Ausreißerwerte in der Datenmenge den Wert des Medians nicht stark ändern. Schauen wir uns das Ganze mehr im Detail an.

### Berechnung des Medians

Zunächst einmal müssen wir die Daten sortieren. Anschließend müssen wir herausfinden, ob die Anzahl der Elemente in dem Datensatz gerade oder ungerade ist – das beeinflusst nämlich die Formel zur Bestimmung des Medians. Sehen wir uns zunächst den einfachen Fall einer Menge mit einer ungeraden Menge an Elementen an. Hier gilt dann, dass der Median folgendermaßen bestimmt werden kann:

$$x_{med} = x_{\frac{n+1}{2}}$$

In dieser Formel ist  $n$  die Anzahl der Elemente, die in dem entsprechenden Datensatz zu finden ist. Wenn die Anzahl der Elemente andererseits gerade ist, bestimmt man den Median auf die folgende Weise:

$$x_{med} = \frac{1}{2}(x_{\frac{n}{2}} + x_{\frac{n}{2}+1})$$

Im Fall einer geraden Anzahl von Elementen ist der Median also der Mittelwert der beiden mittleren Werte in der sortierten Liste. Nehmen wir einfach mal das Beispiel von oben, die Liste  $x$  mit 5 Elementen. Als erstes sortieren wir die Liste:

$$x = \{5, 2, 1, 2, 6\} \rightarrow \{1, 2, 2, 5, 6\}$$

Da die Anzahl der Elemente ungerade ist, verwenden wir die erste Formel zur Bestimmung des Medians:



$$x_{med} = x_{\frac{5+1}{2}} = 2$$

Die Anzahl der Elemente, die kleiner als 2 sind, entspricht genau der Anzahl der Elemente, die größer als 2 sind. Schauen wir uns nun ein Beispiel mit einer geraden Anzahl an Elementen im Datensatz an und fügen eine 7 zu obiger Liste hinzu:

{1, 2, 2, 5, 6, 7}

Der Median berechnet sich jetzt folgendermaßen:

$$x_{med} = \frac{1}{2}(x_{\frac{6}{2}} + x_{\frac{6}{2}+1}) = \frac{1}{2}(x_3 + x_4) = \frac{7}{2} = 3.5$$

Der Median ist jetzt also 3.5. Wir sehen, dass 3 Elemente in der Liste kleiner als 3.5 (1,2,2) und 3 Elemente größer als 3.5 (5,6,7) sind. Im Fall einer geraden Anzahl von Elementen ist der Median also nicht mehr zwingend ein Teil der Datenmenge. Fügen wir nun noch einen Wert hinzu, der stark von den übrigen Werten abweicht, und zwar die Zahl 50:

{1, 2, 2, 5, 6, 7, 50}

In diesem Fall ist der Median:

$$x_{med} = x_{\frac{8}{2}} = 5$$

Wir sehen also, dass sich der Median – obwohl sich ein völlig aus der Bahn geratener Wert in der Datenmenge befindet – nur kaum ändert, nämlich von 3.5 auf 5.

Der Median wird häufig angewendet, um beispielsweise bei Kundenbewertungen eine erste Tendenz zu erkennen.



## Der Mittelwert

Schauen wir uns nun das bekannteste statistische Lagemaß an, den sogenannten Mittelwert oder einfach nur Durchschnitt einer Datenmenge. Die Formel zur Berechnung ist recht simpel:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$$

---

Wenden wir diese Formel beispielsweise auf die folgende bereits bekannte Liste von numerischen Werten an:

$$\{1, 2, 2, 5, 6\}$$

dann erhalten wir den Mittelwert:

$$\bar{x} = \frac{16}{5}$$

### 1.3.2. Varianz und Standardabweichung

Neben den Lagemaßen spielen die sogenannten Streuparameter ebenso eine große Rolle in der Statistik. Die beiden wichtigsten sind die Varianz und die Standardabweichung, sie stehen in einer engen Beziehung zueinander. Fangen wir damit an, die Varianz besser zu verstehen. In Worten ausgedrückt ist sie definiert als die durchschnittliche Abweichung der Daten in einem Datensatz vom Erwartungswert und diese Differenz ins Quadrat genommen. In Formelschreibweise sieht die Varianz folgendermaßen aus:



$$\sigma^2 = \sum_{i=1}^n (x_i - \mu)^2 \cdot p_i$$

$\sigma^2$  ist das Symbol für die Varianz

$\mu$  ist der Erwartungswert

$x_i$  ist der jeweilige Datenpunkt

$p_i$  ist die Wahrscheinlichkeit, den jeweiligen Datenpunkt zu finden

Mit der Varianz schätzt man also ab, wie weit die jeweiligen Datenpunkte vom Erwartungswert der Daten abweichen. In den meisten Fällen handelt

Wochentag	MO	DI	MI	DO	FR	SA	SO
Temperatur Maximal	28	29	27	21	18	28	24

es sich bei diesem Erwartungswert einfach um den Mittelwert der Daten. Schauen wir uns das Ganze an einem Beispiel an. Nehmen wir an, wir hätten eine Woche lang um die Mittagszeit die Temperatur gemessen und die Daten festgehalten:

Der Erwartungswert ist einfach die durchschnittliche Temperatur:

$$\frac{28 + 29 + 27 + 21 + 18 + 28 + 24}{7} = \frac{175}{7} = 25$$

Die durchschnittliche Temperatur liegt also bei 25 Grad. Jetzt setzen wir einfach nur die einzelnen Tageswerte zusammen mit dem Durchschnitt in die Formel für die Varianz ein und erhalten folgende Summe von Termen:

$$\begin{aligned} \sigma^2 &= (28 - 25)^2 \cdot \frac{1}{7} + (29 - 25)^2 \cdot \frac{1}{7} + (27 - 25)^2 \cdot \frac{1}{7} + (21 - 25)^2 \cdot \frac{1}{7} + (18 - \\ &\quad 25)^2 \cdot \frac{1}{7} + (28 - 25)^2 \cdot \frac{1}{7} + (24 - 25)^2 \cdot \frac{1}{7} = 14.86 \end{aligned}$$



Wir sehen also, dass die mittlere quadratische Abweichung der Temperaturen zum Durchschnitt 14.86 Grad<sup>2</sup> sind. Das ist jetzt als Streuparameter schwer zu interpretieren, da es sich um das Quadrat der Abweichung zum Mittelwert handelt. Die tatsächliche Abweichung – oder Streuung – vom Mittelwert sehen wir anhand der sogenannten **Standardabweichung**.

Die Standardabweichung beschreibt die durchschnittliche Entfernung aller gemessenen Werte vom Mittelwert. Sie ist definiert als die Quadratwurzel der Varianz, was erklärt warum die Varianz als Symbol den griechischen Buchstaben Sigma zum Quadrat hatte. In Formelschreibweise sieht die Standardabweichung folgendermaßen aus:

$$\sigma = \sqrt{\sum_{i=1}^n (x_i - \mu)^2 \cdot p_i}$$

Im obigen Beispiel mit den täglich aufgezeichneten Temperaturen würden wir als Standardabweichung den folgenden Wert erhalten:

$$\sigma = \sqrt{14.86} = 3.85$$

Das heißt, die durchschnittliche Abweichung aller Temperaturen vom Mittelwert der gemessenen Temperaturen ist 3.85 Grad. Um das Ganze zu vertiefen, schauen wir uns noch ein Beispiel an. Nehmen wir an, wir hätten Kundenbewertungen auf einer Skala von 1 bis 6. Die häufigste Bewertung ist die 3 mit siebenmaligem Auftreten:

Note	1	2	3	4	5	6
Häufigkeit	4	6	7	4	2	2



Uns interessiert nun die Standardabweichung, also wie stark die Bewertungen im Durchschnitt vom Durchschnitt abweichen. Um die Standardabweichung zu bestimmen, gehen wir in drei Schritten vor:

1. Berechne den Durchschnitt.
2. Berechne die Varianz.
3. Ziehe die Quadratwurzel aus der Varianz.

Der Durchschnitt sieht folgendermaßen aus:

$$\mu = \frac{1 \cdot 4 + 2 \cdot 6 + 3 \cdot 7 + 4 \cdot 4 + 5 \cdot 2 + 6 \cdot 2}{25} = 3$$

Die durchschnittliche Bewertung ist also die Note 3 gewesen. Nun können wir mit diesem Wissen die Varianz berechnen:

$$\sigma^2 = \frac{-2^2 \cdot 4 + -1^2 \cdot 6 + 0^2 \cdot 7 + 1^2 \cdot 4 + 2^2 \cdot 2 + 3^2 \cdot 2}{25} = 2.08$$

Das heißt die Varianz ist 2.08 Noten<sup>2</sup>. Um jetzt die Standardabweichung zu erhalten, ziehen wir daraus noch die Quadratwurzel:

$$\sigma = \sqrt{2.08} = 1.44$$

Die durchschnittliche Abweichung der Bewertungen von der durchschnittlichen Note 3 ist also 1.44 Notenpunkte.



### 1.3.3. Kovarianz und Korrelation

In diesem letzten Abschnitt zur Statistik wollen wir uns damit beschäftigen, wie wir den Zusammenhang zwischen verschiedenen Features eines Datensatzes quantitativ (also mathematisch) beschreiben können. Uns interessiert also, wie die Features eines Datensatzes miteinander korrelieren. Das lässt sich mit dem sogenannten **Korrelationskoeffizienten** beschreiben, und um diesen bestimmen zu können, müssen wir uns zuerst anschauen, was der Begriff der **Kovarianz** bedeutet.

#### Kovarianz

Trocken ausgedrückt wird die Kovarianz verwendet, um festzustellen, ob zwischen zwei Features in einem Datensatz ein linearer und monotoner Zusammenhang besteht. Man kann die Kovarianz beispielsweise anwenden, um in einem Unternehmen den Zusammenhang zwischen der Anzahl der Mitarbeitenden und den produzierten Waren zu bestimmen. In

Formelschreibweise

sieht die Kovarianz folgendermaßen aus:

$$Cov(x, y) = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{N}$$

$$Cov(x, y) = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{N - 1}$$

**Cov(x,y)** steht für die Kovarianz der beiden Daten Features x und y

$x_i$  und  $y_i$  sind die einzelnen Datenpunkte des jeweiligen Features

$\bar{x}$  und  $\bar{y}$  sind die Mittelwerte der beiden Features

N ist die gesamte Menge an Datenpunkten in den beiden Features x und y

Diese Formel sieht nun doch ein bisschen unheimlich aus, deshalb wenden wir sie gleich in einem Beispiel an und sehen, wie sie zum Einsatz kommt. Nehmen wir an, wir hätten für einen Zoo die Daten über die Anzahl der Sonnenstunden in den jeweiligen Monaten und die Anzahl der Besuchenden in den entsprechenden Monaten letztes Jahr gemacht. Uns



interessiert nun, welcher Zusammenhang zwischen der Anzahl der Sonnenstunden x und der Anzahl der Besuchenden y bestehen könnte.

Monat i	Anzahl Sonnenstunden x	Besucherzahl y
1	1,6	28300
2	2,6	28000
3	3,7	41000
4	5,3	40000
5	6,9	48000
6	7,1	47500
7	7,2	43000
8	6,7	50700
9	5,1	50000
10	3,6	48000
11	2,1	25000
12	1,4	24000

Bevor wir die Kovarianz der beiden Features Anzahl Sonnenstunden und Besuchendenanzahl berechnen, müssen wir die Mittelwerte der beiden Spalten bestimmen. Das geht recht schnell und hier erhalten wir:

$$\bar{x} = 4.44$$

$$\bar{y} = 39458.3$$

Durchschnittlich hatte also jeder Monat durchschnittlich 4.44 Sonnenstunden und durchschnittlich waren monatlich 39458 Menschen zu Besuch. Wenn wir das nun in die Formel für die Kovarianz einsetzen, erhalten wir:

$$Cov(x, y) = 19117.35$$



Was bedeutet jetzt dieser große Wert? Heißt das, es besteht ein großer Zusammenhang zwischen der Anzahl der Sonnenstunden und der Anzahl der Besuchenden? Richtig verständlich wird die Kovarianz erst, wenn wir mit ihr die **Korrelation** zwischen zwei Features bestimmen.

## Der Korrelationskoeffizient

Der Korrelationskoeffizient basiert auf der Kovarianz und nimmt Werte zwischen -1 und 1 an. Er ist ein Maß dafür, wie stark zwei Features eines Datensatzes miteinander zusammenhängen. Ist der Wert kleiner als 0, dann besteht ein negativer Zusammenhang zwischen den Features. Je kleiner die Werte des einen Features, umso größer werden die Werte des anderen Features. Es passiert also das Gegenteil. Ist der Wert des Korrelationskoeffizienten größer als Null, dann besteht ein positiver Zusammenhang zwischen den Features. Das bedeutet, wenn die Werte des einen Features wachsen, wachsen auch die des anderen. Genauso wenn die Werte des einen fallen, fallen auch die anderen mit. Wenn der Korrelationskoeffizient exakt Null ist, dann besteht überhaupt keine Verbindung zwischen den beiden Features. Wie berechnet man nun aber die Korrelation? Die Formel sieht folgendermaßen aus:

$$r_{xy} = \frac{s_{xy}}{s_x s_y}$$

$r_{xy}$  ist der **Korrelationskoeffizient** zwischen den beiden Variablen x und y

$s_x$  und  $s_y$  sind die Standardabweichungen der beiden Variablen

Wollen wir also in unserem Beispiel oben mit den Daten aus dem Zoo die Korrelation zwischen den Sonnenstunden und der Anzahl der Besuchenden bestimmen, müssen wir noch die Standardabweichungen der beiden Features bestimmen.

$$s_x = 2.13$$

$$s_y = 9862$$



Wenn wir das zusammen mit der Kovarianz in die obige Formel einsetzen, erhalten wir als Ergebnis:

$$r_{xy} = 0.91$$

Der Korrelationskoeffizient liegt also sehr nah bei 1, weswegen wir sagen können, dass ein starker Zusammenhang zwischen den Sonnenstunden und der Anzahl der Besuchenden im Zoo besteht. Je mehr die Sonne scheint, umso mehr Besuchende strömen in den Zoo. Für die Praxis lässt sich festhalten:

Ist die Korrelation etwa  $\pm 0.1$ , dann besteht ein kleiner Zusammenhang

Ist die Korrelation etwa  $\pm 0.3$  dann besteht ein mittlerer Zusammenhang

Ist die Korrelation etwa  $\pm 0.5$  oder größer, ist der Zusammenhang groß

## 1.4. Die pandas-Bibliothek

In diesem Kapitel beschäftigen wir uns mit einem nützlichen Tool zur Datenanalyse, eigens gefertigt für die Programmiersprache Python. Die pandas-Bibliothek enthält zahlreiche Funktionen, um Daten zu strukturieren, zu reinigen und zu analysieren. Ebenso kann man sich aus zahlreichen Datenformaten (wie Excel, CSV, JSON) die Daten mit pandas in den Python-Code importieren und in ein einheitliches Format bringen, das als Dataframe bekannt ist.

Daten zu analysieren, scheint eine Menge Code und Rechenarbeit zu bedeuten; das haben wir zumindest in den letzten paar Kapiteln bemerkt. Um bei späteren Datenanalysen Codezeilen zu sparen, gibt es die pandas-Bibliothek, eine Sammlung von Funktionen und Klassen zur Analyse von hauptsächlich tabellarischen Daten. Wenn man mit dem Anaconda-Navigator arbeitet und beispielsweise **spyder** als IDE verwendet, sollte die pandas-Bibliothek bereits vorinstalliert sein. Falls nicht, geht dies ganz einfach mit dem pip-Befehl:



```
C:\Users\Your Name>pip install pandas
```

Falls das ohne Fehlermeldung funktioniert hat, sollte der folgende Code ohne Fehlermeldung funktionieren:

```
import pandas as pd
```

Jetzt schauen wir uns zuerst an, was Dataframes sind und wie man mit ihnen umgeht. Anschließend lernen wir, wie wir ein Excel-File einlesen und als Dataframe darstellen können.

### 1.4.1. Dataframes

Bei Dataframes handelt es sich um das grundlegende Datenformat, mit dem in pandas Daten tabellarisch strukturiert werden. Dataframes sind 2-dimensionale Strukturen, wie 2- dimensionale Arrays oder einfach eine Tabelle mit Zeilen und Spalten. Erstellen wir einfach mal ein Dataframe, um zu sehen, wie das denn aussieht:

```
import pandas as pd

#Dataframes
data = {"Alter": [42, 5, 67], "Ausdauer": [1, 2, 1]}
df = pd.DataFrame(data)
print(df)
```

Ausgabe:

	Alter	Ausdauer
0	42	1
1	5	2
2	67	1

Wir haben unsere Daten als Dictionary mit zwei Einträgen definiert, wobei jedem Eintrag eine Liste mit gleich vielen Elementen zugeordnet ist. Dieses



Dictionary packen wir dann in das Dataframe-Format der pandas-Bibliothek und erhalten eine tabellarische Darstellung der Daten.

### Eine spezielle Spalte auswählen

Möchten wir uns jetzt nicht die ganze Tabelle, sondern nur eine Spalte ausgeben lassen, geht das ganz einfach mit dem folgenden Befehl:

```
alter = df.get("Alter")
print(alter)
```

Und die Ausgabe sieht folgendermaßen aus:

```
0    42
1     5
2    67
Name: Alter, dtype: int64
```

### Eine spezielle Zeile auswählen

Um eine spezielle Zeile auszuwählen, verwendet man den loc[]-Befehl:

```
#Zeilen ausgeben lassen
zeile0 = df.loc[0]
print(zeile0)
```

In eckigen Klammern gibt man die Zeilennummer an, die bei 0 anfängt zu zählen. Die Ausgabe sieht folgendermaßen aus:

```
Alter      42
Ausdauer   1
Name: 0, dtype: int64
```



Möchte man sich mehrere Zeilen ausgeben lassen, geht das mit einem zusätzlichen „[]“:

```
#Zeilen ausgeben lassen  
zeile0 = df.loc[[0,1]]  
print(zeile0)
```

Die Ausgabe ist:

	Alter	Ausdauer
0	42	1
1	5	2

### 1.4.2. Excel Dateien einlesen

Wenn man sich als Data Scientist mit der Analyse von großen Datenmengen befasst, möchte man diese nicht alle einzeln in den Code hineinkopieren. Häufig liegen Datensätze in Unternehmen auch in Excel-Form vor, weswegen es super wäre, diese Datenformate einfach in Python einlesen und dann weiterverarbeiten zu können. Genau das ist mit der pandas- Bibliothek möglich. Wir können ein xlsx-File mit der pandas-Funktion ExcelFile() einlesen und in ein Dataframe packen. Somit haben wir dann alle Daten in tabellarischer Form im Code und bereit zum Verarbeiten.

Als Beispiel habe ich ein Excel-File mit dem Dateinamen Daten.xlsx erstellt, in welchem ein Tabellenblatt beschrieben wurde. Dort ist eine Tabelle mit den drei Spalten Alter, Dauer und Zeit. Damit der Code funktioniert, muss das Python-File im gleichen Ordner wie das Excel-File gespeichert sein.

```
file = 'Daten.xlsx'  
xl = pd.ExcelFile(file)  
df1 = xl.parse("Tabellenblatt1")
```

Die Ausgabe sieht folgendermaßen aus:



	Alter	Dauer	Zeit
0	23.0	7.0	4.0
1	32.0	56.0	5.0
2	3.0	6.0	65.0
3	4.0	67.0	65.0
4	23.0	7.0	4.0
5	3.0	67.0	4.0
6	2.0	67.0	4.0
7	342.0	67.0	4.0
8	43.0	7.0	4.0

### 1.4.3. CSV- und JSON-Files einlesen

Als Data Scientist hat man es nicht nur mit Daten im Excel-Format zu tun. Andere beliebte Formate zur Speicherung von großen Datenmengen sind das CSV-Format und das JSON-Format. Wie wir diese Formate in Python mit Hilfe der pandas-Bibliothek einlesen und verarbeiten können, lernen wir in diesem Abschnitt.

#### CSV-Files einlesen

CSV-Files sind besonders beliebt bei der Speicherung von großen Datenmengen, weil das Format recht einfach zu erzeugen und zu verstehen ist. CSV steht hier für „comma-separated values“, was bedeutet, dass die einzelnen Einträge der Datenbank durch Kommata getrennt sind. In einem CSV-File sind die Daten zeilenweise angeordnet, wobei jede Zeile durch einen Zeilenumbruch von der nächsten getrennt wird. Eine Zeile entspricht somit einem Datensatz. In der Zeile selber sind die Daten durch Kommata, Semikolons, Leerzeichen oder andere Tabulatoren getrennt. Schauen wir uns mal an, wie eine solche CSV-Datei eigentlich aussieht.

Wir erinnern uns an das Projekt von der vorherigen Woche, in welchem wir zwei Tabellen aus einem Excel-File eingelesen und analysiert haben. Nehmen wir der Einfachheit halber die kleinere Tabelle aus Tabellenblatt 1



als Beispiel; Wir haben 6 Spalten und 15 Zeilen, die erste Zeile gibt die Spaltenbezeichnung wieder, die restlichen enthalten die Daten.

Tag	Wetteraussicht	Temperaturkategorie	Luftfeuchtigkeit	Windstärke	Draußen Essen
1	Sonnig	Heiß	Hoch	Schwach	Nein
2	Sonnig	Heiß	Hoch	Stark	Nein
3	Bewölkt	Heiß	Hoch	Schwach	Ja
4	Regnerisch	Mild	Hoch	Schwach	Ja
5	Regnerisch	Kalt	Normal	Schwach	Ja
6	Regnerisch	Kalt	Normal	Stark	Nein
7	Bewölkt	Kalt	Normal	Stark	Ja
8	Sonnig	Mild	Hoch	Schwach	Nein
9	Sonnig	Kalt	Normal	Schwach	Ja
10	Regnerisch	Mild	Normal	Schwach	Ja
11	Sonnig	Mild	Normal	Stark	Ja
12	Bewölkt	Mild	Hoch	Stark	Ja
13	Bewölkt	Heiß	Normal	Schwach	Ja
14	Regnerisch	Mild	Hoch	Stark	Nein

Im CSV-Format würden die Daten folgendermaßen aussehen:

Tag, Wetteraussicht, Temperatur, Luftfeuchtigkeit, Wind, Außenverkauf

1. sonnig, heiß, hoch, schwach, nein

2. sonnig, heiß, hoch, stark, nein



3. bewölkt, heiß, hoch, schwach, ja
4. regnerisch, mild, hoch, schwach, ja
5. regnerisch, kalt, normal, schwach, ja
6. regnerisch, kalt, normal, stark, nein
7. bewölkt, kalt, normal, stark, ja
8. sonnig, mild, hoch, schwach, nein
9. sonnig, kalt, normal, schwach, ja
10. regnerisch, mild, normal, schwach, ja
11. sonnig, mild, normal, stark, ja
12. bewölkt, mild, hoch, stark, ja
13. bewölkt, heiß, normal, schwach, ja
14. regnerisch, mild, hoch, stark, nein

Die erste Zeile gibt die Spaltenbeschreibungen wieder, wobei jede Spaltenbeschreibung durch ein Komma von den anderen getrennt ist. Danach werden die Daten zeilenweise aufgeführt, jeweils durch Kommata voneinander getrennt.

Wenn wir jetzt ein solches CSV-File in unseren Python-Code einlesen wollen, dann möchten wir es ja am Ende in einem Dataframe haben, um damit arbeiten zu können. Das kriegt man ganz einfach mit ein paar wenigen Zeilen hin.

```
import pandas as pd  
  
df = pd.read_csv('data.csv')  
  
print(df.to_string())
```

Wir rufen einfach die `read_csv()`-Funktion auf, die in der `pandas`-Bibliothek enthalten ist. Als Argument übergeben wir den Dateinamen, falls sich die



Datei im gleichen Verzeichnis wie der Programmcode befindet, andernfalls den Dateipfad.

## JSON-Files einlesen

Große Datenmengen werden auch manchmal in sogenannten JSON-Files gespeichert. JSON ist hierbei eine Abkürzung für „Javascript Object Notation“, was uns wissen lässt, dass dieses Datenformat aus dem Webdevelopment kommt. JSON-Files sind leichtgewichtig, einfach zu lesen und eignen sich hervorragend zur Speicherung von temporären Daten, wie beispielsweise ausgefüllten Benutzerformularen auf einer Website.

JSON-Dateien sind ebenso wie CSV-Dateien reine Text-Files, allerdings liegt der Unterschied zwischen den beiden in der internen Struktur. JSON-Files sind genauso wie Python Dictionaries aufgebaut. Sie enthalten einen Satz an Namens- und Wertepaaren, die zwischen geschweiften Klammern eingefügt werden.

```
{
  "Tag": [
    {"0": "1.0", "1": "2.0", "2": "3.0", "3": "4.0", "4": "5.0", "5": "6.0", "6": "7.0", "7": "8.0", "8": "9.0", "9": "10.0", "10": "11.0", "11": "12.0", "12": "13.0", "13": "14.0"},  
  "Wetteraussicht":  
    {"0": "sonnig", "1": "sonnig", "2": "bew\u00fclkt", "3": "regnerisch", "4": "regnerisch", "5": "regnerisch", "6": "bew\u00fclkt", "7": "so  
nnig", "8": "sonnig", "9": "regnerisch", "10": "sonnig", "11": "bew\u00fclkt", "12": "bew\u00fclkt", "13": "regnerisch"},  
  "Temperaturkategorie":  
    {"0": "heiss", "1": "heiss", "2": "heiss", "3": "mild", "4": "kalt", "5": "kalt", "6": "kalt", "7": "mild", "8": "kalt", "9": "mild", "10": "mil  
d", "11": "mild", "12": "heiss", "13": "mild"},  
  "Luftfeuchtigkeit":  
    {"0": "hoch", "1": "hoch", "2": "hoch", "3": "hoch", "4": "normal", "5": "normal", "6": "normal", "7": "hoch", "8": "normal", "9": "normal", "1  
0": "normal", "11": "hoch", "12": "normal", "13": "hoch"},  
  "Windst\u00e4rke":  
    {"0": "schwach", "1": "stark", "2": "schwach", "3": "schwach", "4": "schwach", "5": "stark", "6": "stark", "7": "schwach", "8": "schwach", "9  
": "schwach", "10": "stark", "11": "stark", "12": "schwach", "13": "stark"},  
  "Draussen Essen":  
    {"0": "nein", "1": "nein", "2": "ja", "3": "ja", "4": "ja", "5": "nein", "6": "ja", "7": "nein", "8": "ja", "9": "ja", "10": "ja", "11": "ja", "12  
": "ja", "13": "nein"}  
}
```

In der Abbildung oben ist das erste Tabellenblatt vom Projekt der vorherigen Woche im JSON-Format zu sehen. Der Datensatz beginnt mit einer geschweiften Klammer, dann steht da der erste Spaltenname Tag gefolgt von einem Doppelpunkt. Hinter dem Doppelpunkt kommt dann eine geschweifte Klammer, in der alle Werte aus der Spalte Tag aufgeführt sind. Wenn die Spalte zu Ende ist, wird mit einer geschweiften Klammer geschlossen. Dann kommt die nächste Spalte mit ihren Werten.



Möchte man ein JSON-File in ein pandas-Dataframe laden, geht das mit den folgenden wenigen Zeilen Code:

```
import pandas as pd

df = pd.read_json('data.json')

print(df.to_string())
```

## 1.4.4. Datenreinigung

Als Data Scientist möchte man wertvolle Informationen aus großen Datenmengen gewinnen, allerdings ist der Informationsgehalt unserer Analysen stark abhängig von der Qualität der Daten. Egal ob wir es mit Wetterdaten oder den Postanschriften unserer Kunden/Kundinnen zu tun haben, es können immer wieder einige Datenreihen fehlerhafte Elemente enthalten. Und solche fehlerhaften Elemente beinträchtigen die Qualität der Analyse. Daher ist ein wichtiger Schritt eines jeden Data-Science-Projekts, die Daten zu bereinigen. Die pandas-Bibliothek liefert hierfür einige nützliche Tools, die wir in diesem Abschnitt kennenlernen werden.

Mit diesen Tools können wir dann

- leere Zellen ausfindig machen,
- Datumsangaben korrigieren und vereinheitlichen,
- falsche Daten im Allgemeinen bearbeiten und
- Duplikate erkennen und löschen.



## Leere Zellen

Da leere Zellen das Ergebnis einer Analyse stark beeinflussen können, sind diese das Erste, womit man sich als Data Scientist bei einem frischen Datensatz befasst. Ein Weg, um mit leeren Zellen umzugehen, ist, die Datenreihen mit leeren Zellen ganz einfach aus dem gesamten Datensatz zu löschen. In der Regel stellt das auch kein Problem dar, wenn man es mit großen Datensätzen zu tun hat.

Hat man es mit einer solchen leeren Zelle zu tun, dann wird ihr Inhalt typischerweise als Nullwert bezeichnet. Um Reihen mit solchen Nullwerten aus einem Datensatz zu entfernen, verwendet man die `dropna()`-Funktion.

```
import pandas as pd

df = pd.read_csv('data.csv')

new_df = df.dropna()

print(new_df.to_string())
```

Die `dropna()`-Funktion nimmt das Dataframe, in dem der Datensatz gespeichert ist, entfernt alle Zeilen mit Nullwerten und gibt das Ergebnis als neues Dataframe zurück. Die `dropna()`-Funktion verändert also den originalen Datensatz nicht. Möchte man allerdings den originalen Datensatz verändern, dann kann man der `dropna()`-Funktion als Argument den Befehl `inplace=True` mitgeben:



```
import pandas as pd

df = pd.read_csv('data.csv')

df.dropna(inplace = True)

print(df.to_string())
```

So bekommt man dann kein neues Dataframe, sondern das originale, nur ohne die Zeilen mit Nullwerten. Möchte man jetzt aber nicht jede Zeile mit leeren Zellen löschen, sondern diese Zellen mit irgendwelchen Werten füllen, kann man das mit der `fillna()`-Funktion machen:

```
import pandas as pd

df = pd.read_csv('data.csv')

df.fillna(130, inplace = True)
```

In diesem Beispiel wird jede leere Zelle mit dem Wert 130 gefüllt, egal in welcher Spalte oder Zeile sich die leere Zelle befunden hat. Möchte man nur die leeren Zellen einer spezifischen Spalte mit einem Wert besetzen, muss man als erstes für das Dataframe den Spaltennamen spezifizieren:

```
import pandas as pd

df = pd.read_csv('data.csv')

df["Calories"].fillna(130, inplace = True)
```



Eine häufig verwendete Methode, um leere Zellen zu füllen, ist, den Mittelwert, Median oder Modus der Werte in die entsprechende Spalte einzufügen. Das kann man mit den in der pandas-Bibliothek enthaltenen Funktionen mean(), median() und mode() tun. Dann ersetzt man mit derfillna()-Funktion in der entsprechenden Spalte die Nullwerte:

```
import pandas as pd

df = pd.read_csv('data.csv')

x = df["Calories"].mean()

df["Calories"].fillna(x, inplace = True)
```

### Falsches Format

Häufig, wenn wir es mit Datumsangaben in Datensätzen zu tun haben, tauchen Zellen auf, in denen das Datum nicht im korrekten Format eingefügt ist. Wenn man vor diesem Problem steht, hat man in der Regel zwei Optionen:

- Man entfernt die jeweiligen Zeilen.
- Man bringt alle Zellen einer Spalte in ein einheitliches Format.

```
import pandas as pd

df = pd.read_csv('data.csv')

df['Date'] = pd.to_datetime(df['Date'])

print(df.to_string())
```

Wie wir Zeilen löschen können, haben wir bereits gesehen. Möchten wir aber alle Daten in der Spalte mit dem Datum in ein einheitliches Format bringen, geht das mit der to\_datetime()-Funktion. Diese bekommt als



Argument die entsprechende Spalte aus dem Dataframe, die vereinheitlicht werden soll.

## Falsche Daten

Wenn wir es mit falschen Daten zu tun haben, heißt das nicht unbedingt, dass die Daten im falschen Format vorliegen oder die Zellen nur Nullwerte haben. Manchmal sind die Daten in einigen Zellen einfach nur falsch, weil sie nicht in das Muster der übrigen Zahlen passen. Ein Beispiel dafür wäre, wenn in einer Zelle 199 anstatt 1.99 steht. Solche falschen Daten erkennt man meist, wenn man sich den Datensatz anschaut und Ausreißer ausfindig macht, da man ja eine gewisse Vorstellung davon hat, wie die Daten auszusehen haben. Wenn wir uns beispielsweise in einem Datensatz die Spalte mit der Trainingsdauer einer Person im Fitnessstudio anschauen und alle Werte zwischen 30 und 90 Minuten sind, können wir davon ausgehen, dass ein Workout mit einer Länge von 450 Minuten nicht stimmen kann und womöglich 45 Minuten gemeint waren. Auch hier gibt es wieder zwei Möglichkeiten, um mit den fehlerhaften Daten umzugehen:

- Wir löschen die Zeilen mit falschen Daten.
- Wir ersetzen die Werte.

Das Ersetzen der Werte funktioniert mit dem Befehl `loc[Zeilennummer, Spaltenname]`:

```
df.loc[7, 'Duration'] = 45
```

Hier ändern wir in der Duration-Spalte den Wert der Zelle in der 7. Zeile von 450 auf 45. In kleinen Datensätzen kann man noch – so wie im Code-Beispiel oben – alle falschen Daten händisch bereinigen und durch neue Werte ersetzen, bei großen Datensätzen ist das kaum noch möglich. Hier muss man Regeln dafür festlegen, welche Datenwerte wodurch ersetzt werden sollen:



```
for x in df.index:  
    if df.loc[x, "Duration"] > 120:  
        df.loc[x, "Duration"] = 120
```

Mit einer for-Schleife geht man durch jede Zeile des Dataframes und ändert in der jeweiligen Zeile in der Spalte Duration den Wert, falls dieser größer als 120 ist und setzt den neuen Wert auf 120. Man kann natürlich solche falschen Werte auch beispielsweise durch den Mittelwert aus den Daten ersetzen, deren Werte kleiner als 120 sind.

Möchte man dagegen alle Zeilen mit falschen Daten einfach aus dem Datensatz entfernen, geht das ebenfalls über eine for-Schleife mit dem drop()-Befehl.

```
for x in df.index:  
    if df.loc[x, "Duration"] > 120:  
        df.drop(x, inplace = True)
```

## Duplikate

Zu guter Letzt wollen wir uns noch anschauen, wie wir mit Duplikaten in einem Datensatz umgehen können. Duplikate sind Zeilen in einem Datensatz, die doppelt oder mehrfach auftreten. Um Duplikate in einem Datensatz entdecken zu können, verwendet man die duplicated()-Funktion. Diese gibt für jede Zeile einen Booleschen Wert zurück, also entweder TRUE oder FALSE. Wenn die duplicated()-Funktion eine Zeile als Duplikat identifiziert hat, gibt sie den Wert TRUE zurück, andernfalls FALSE.



```
import pandas as pd

df = pd.read_csv('data.csv')

print(df.duplicated())
```

Möchte man Duplikate schließlich aus dem Datensatz entfernen, schaffen wir das mit der drop\_duplicates()-Funktion:

```
import pandas as pd

df = pd.read_csv('data.csv')

df.drop_duplicates(inplace = True)

print(df.to_string())
```

## 1.4.5. Skalierung

Daten zu skalieren heißt, sie so zu transformieren, dass sie in eine bestimmte, einheitliche Skala passen, wie zum Beispiel von 0 bis 100 oder – was häufiger vorkommt in der Praxis – von 0 bis 1. Wenn ein Satz an Daten skaliert ist, bedeutet das, dass eine Änderung eines numerischen Wertes um 1 überall die gleiche Bedeutung hat. Das wird vor allem bei Machine-Learning-Algorithmen wie der linearen Regression, K-nearest Neighbours oder neuronalen Netzen wichtig. Warum aber genau müssen wir Daten skalieren? Schauen wir uns das anhand eines einfachen Beispiels an.

Nehmen wir an, wir hätten die Preise eines Produkts in EUR und in Yen angegeben. Annäherungsweise können wir sagen, dass 1 EUR ungefähr 100 Yen entspricht. Wenn man jetzt die Preise für das Produkt nicht skaliert und auf eine einheitliche Skala bringt, wird ein Machine-Learning-Algorithmus



bei der Analyse denken, dass eine Preisänderung von 1 Yen die gleiche Gewichtung wie eine Preisänderung um 1 EUR hat!

Währungen kann man jetzt natürlich ineinander umrechnen, aber es gibt Datensätze, die sich nicht so einfach ineinander umrechnen lassen. Wenn wir es zum Beispiel mit den beiden Features Körpergröße und Gewicht zu tun haben (und beide in den Machine-Learning-Algorithmus zur Analyse eingespeist werden), werden wir uns schwertun, eine Umrechnungsformel von der Körpergröße zum Gewicht aufzustellen. Oder wie viele Zentimeter entsprechen einem Kilogramm?

Dafür ist die Skalierung da. Es gibt viele unterschiedliche Skalierungsmethoden und wir werden in diesem Abschnitt die drei wichtigsten kennenlernen.

### **Simple-Feature-Skalierung**

Die einfachste Methode, um numerische Daten auf eine einheitliche Skala zu bringen, ist die sogenannte Simple-Feature-Skalierung. Hierbei nimmt man einen Datensatz und sucht zunächst den größten vorkommenden Wert, der dann als Ausgangspunkt für die Skalierung hergenommen wird. Hat man den größten Wert, dann teilt man jeden Datenwert einfach durch diesen maximalen Wert und skaliert damit die Daten auf einen Bereich zwischen 0 und 1. Mathematisch haben wir ganz einfach einen Bruch:

$$X_{new} = \frac{X_{old}}{X_{max}}$$

### **Min-Max-Skalierung**

Weitaus beliebter in der Anwendung als die Simple-Feature-Skalierung ist die sogenannte Min-Max-Skalierung. Für diese Skalierungsmethode benötigen wir das Minimum und das Maximum des zu skalierenden Datensatzes. Wir nehmen einen Datenwert und subtrahieren von diesem



den minimalen in den Daten vorkommenden Wert. Das Ergebnis teilen wir dann durch die Differenz aus dem Minimum und dem Maximum in dem Datensatz. Als Formel ausgeschrieben sieht das folgendermaßen aus:

$$X_{new} = \frac{X_{old} - X_{min}}{X_{max} - X_{min}}$$

## Standardisierung von Daten

Eine weitere und etwas ausgefeilte Methode zur Skalierung von Daten ist die sogenannte Standardisierung. Mit dieser Methode transformiert man die Daten so, dass ihr Mittelwert 0 und die Standardabweichung 1 ist. Es gibt verschiedene Methoden zur Standardisierung von Daten, wir wollen uns hier die häufig verwendete Methode des Z-Score anschauen. Möchte man mit Hilfe des Z-Scores die Daten standardisieren, sieht das als mathematische Formel ausgeschrieben folgendermaßen aus:

$$Z = \frac{X - \mu}{\sigma}$$

X ist der Datenwert

$\mu$  ist der Mittelwert der Daten

$\sigma$  ist die Standardabweichung der Daten

Um besser zu verstehen, wo die Z-Score-Standardisierung angewendet wird, schauen wir uns die Prüfungsergebnisse der beiden Studierenden Alice und Bob an, die an zwei unterschiedlichen Unis mit unterschiedlichen Bewertungssystem studieren.

Alice hat in einem Test 85 Punkte bekommen, in welchem der Mittelwert bei 75 Punkten und die Standardabweichung vom Mittelwert bei 5 Punkten lag.

Bob auf der anderen Seite hat in seinem Test 615 Punkte bekommen, wobei der Mittelwert bei 600 Punkten lag und die Standardabweichung 50 Punkte betrug.

Wie können wir jetzt die Ergebnisse der beiden vergleichen? Woher wissen wir, wer von den beiden das bessere Prüfungsergebnis hat?



Hier hilft dann die Standardisierung. Schauen wir uns an, welchen Z-Score die beiden Prüfungsergebnisse haben.

Alice mit ihren 85 Punkten hat einen Z-Score von  $(85-75)/5 = 2$ .

Bob mit seinen 615 Punkten hat einen Z-Score von  $(615-600)/50 = 0.3$ .

Das bedeutet, dass das Ergebnis von Alice 2 Standardabweichungen über dem Mittelwert liegt, während Bobs Ergebnis nur 0.3 Standardabweichungen über dem Mittelwert liegt. Das wiederum heißt, dass sich Alice stärker vom Mittelwert abhebt und somit das bessere Prüfungsergebnis hat.

Ein solcher Vergleich wäre mit einer bloßen Simple-Feature- oder Min-Max-Skalierung kaum möglich gewesen.



## 1.5. Die Matplotlib-Bibliothek

In diesem Kapitel lernen wir die wichtige Matplotlib-Bibliothek kennen, mit welcher wir unsere Datenanalysen mit visuellen Darstellungen unterstützen können. Wir lernen, wie man einen Plot erstellt und ihn graphisch aufbereitet. Dann lernen wir die sogenannte Subplot()-Funktion kennen, mit welcher wir mehrere Plots in einer Abbildung einheitlich präsentieren können. Anschließend schauen wir uns Streudiagramme, Histogramme und Kuchendiagramme an.

### 1.5.1. Plots erstellen

Als Data Scientist müssen wir Daten nicht nur analysieren, wir müssen auch die Ergebnisse der Analyse visuell darstellen können. Dabei hilft die Matplotlib-Bibliothek. In der Regel verwendet man meistens die Unterbibliothek pyplot, weshalb die Import-Anweisung zu Beginn des Programmcodes folgendermaßen aussieht:

#### Ausgabe:

```
import matplotlib.pyplot as plt
```

Fangen wir mit einem ganz einfachen Plot an. Wir definieren zwei Listen, eine für die x-Achse und eine für die y-Achse, und tragen dann die Werte der Liste gegeneinander in ein Koordinatensystem auf. Der Code sieht so aus:



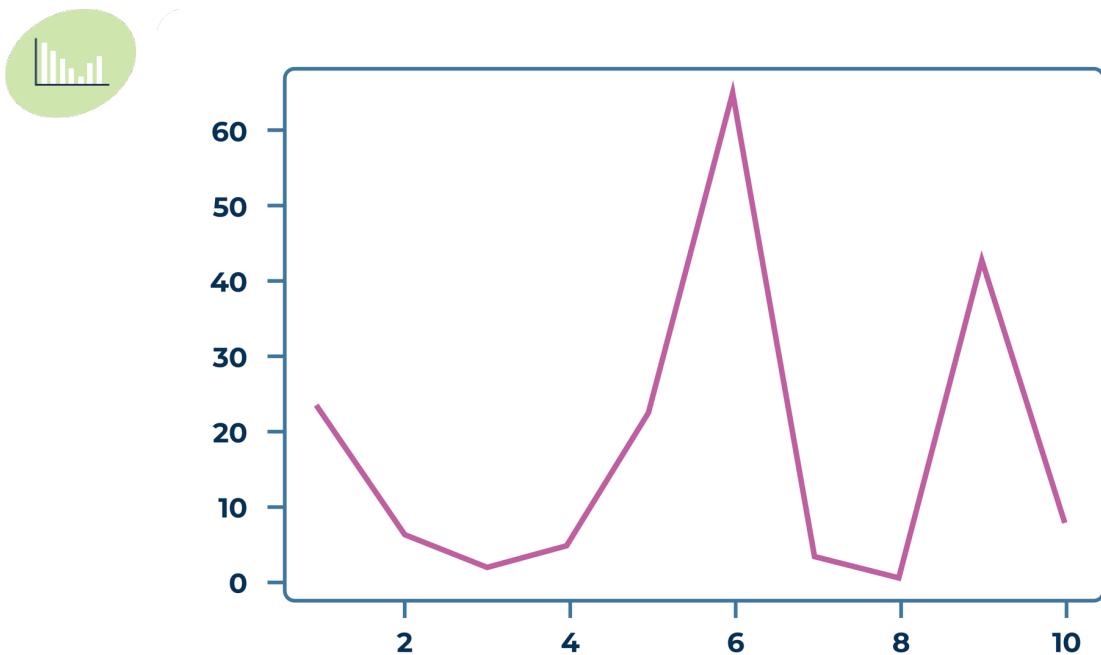
```
import matplotlib.pyplot as plt

# Einfacher Plot

x = [1,2,3,4,5,6,7,8,9,10]
y = [23,6,2,5,23,65,3,1,43,7]
y2 = [2,65,54,6,43,23,3,1,4,72]

plt.plot(x,y)
plt.show()
```

Wir haben hier auch schon eine zweite Liste für die y-Achse definiert, die wir nachher verwenden werden, um mehrere Graphen in einem Plot zu sehen. Mit dem Befehl plt.plot(x,y) trägt man die Punktpaare der beiden Listen auf (wichtig ist hier, dass beide Listen die gleiche Länge haben). Mit dem Befehl plt.show() wird am Ende der Graph dargestellt, was so aussieht:

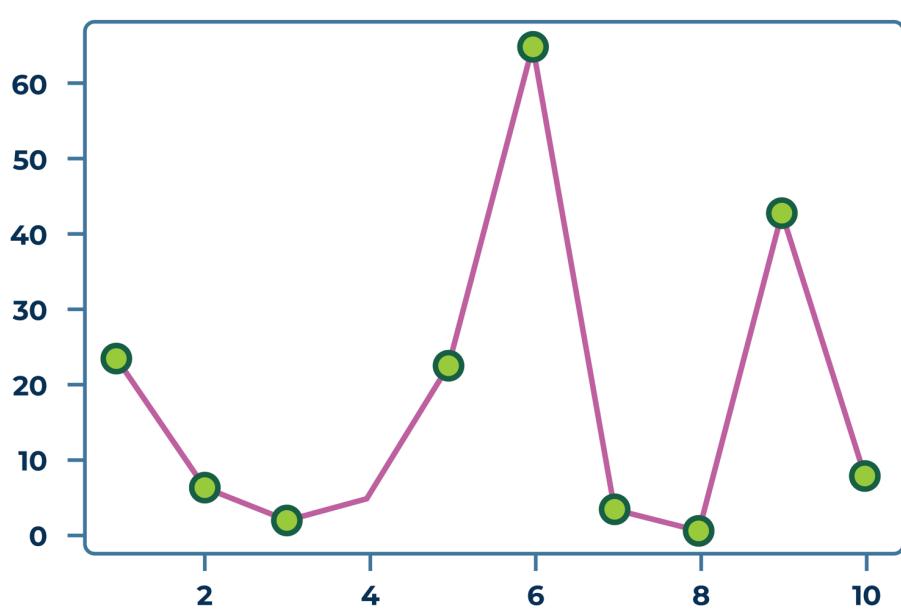




Wir sehen also, dass beim Plotten automatisch Linien zwischen den Datenpunkten (den Paaren von x- und y-Werten) eingezeichnet werden. Wir können die Punkte – die auch als Marker bezeichnet werden – auch darstellen und ihre Größe und Farbe bestimmen.

```
# Marker: Größe, Farbe und Breite  
plt.plot(x,y,marker = 'o',markersize = 5, markerfacecolor = 'r')  
plt.show()
```

Den Markertyp legt man mit dem Argument `marker=“` fest. Es gibt verschiedene Typen wie den hier verwendeten Kreis oder Sternchen „\*“ oder einfach nur ein Punkt „.“. Die Größe der Marker wird mit dem Argument `markersize` festgelegt und die Farbe der Punkte mit dem Argument `markerfacecolor`. Das Ergebnis sieht dann folgendermaßen aus:

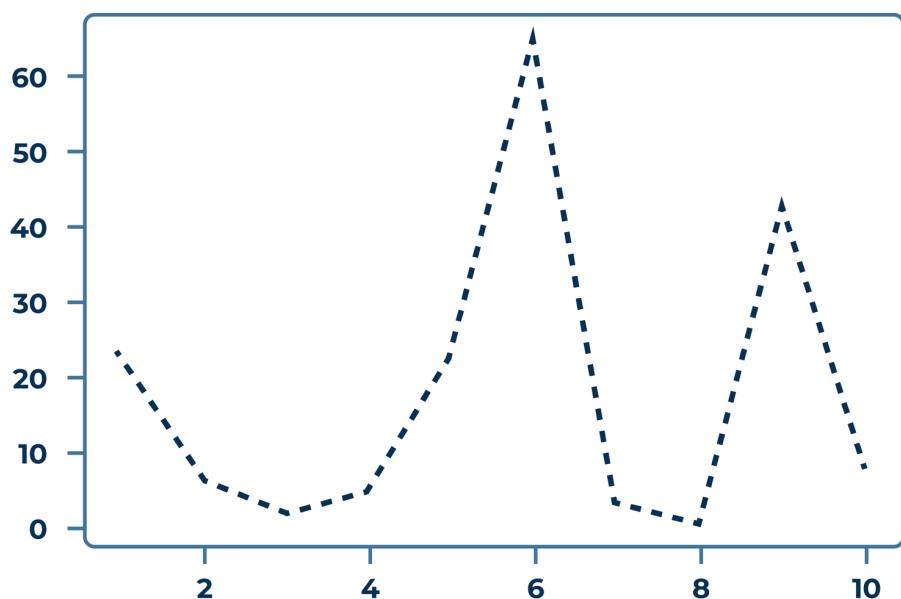




Auch die Verbindungslien zwischen den Markern können wir verändern. Wir können aus ihnen gepunktete Linien machen, die Farbe und auch die Breite verändern.

```
#Linien: Stil, Farbe und Breite  
plt.plot(x,y,linestyle ='dotted',color='g',linewidth='3')  
plt.show()
```

Das Argument linestyle legt den Stil der Verbindungslien fest – es kann gepunktet sein (dotted) oder auch gestrichelt (dashed). Die Farbe wird mit dem Argument color festgelegt und die Breite der Linie mit linewidth. Das Ergebnis sieht dann folgendermaßen aus:



Jetzt ist es natürlich wenig vorteilhaft, ein Diagramm ohne Beschriftungen zu haben. Dafür kann man mit pyplot den Achsen Beschriftungen und dem ganzen Diagramm einen Titel und eine Legende für die verschiedenen vorkommenden Graphen geben:



```
#Labels

font_achsen = {'family': 'serif', 'color': 'red', 'size':12}
font_titel = {'family': 'serif', 'color': 'green', 'size':20}

plt.plot(x,y,linestyle ='dotted',color='r',linewidth=2,label="Daten 2022")
plt.plot(x,y2,linestyle='dashed',color='b',linewidth=1,label="Daten 2021")

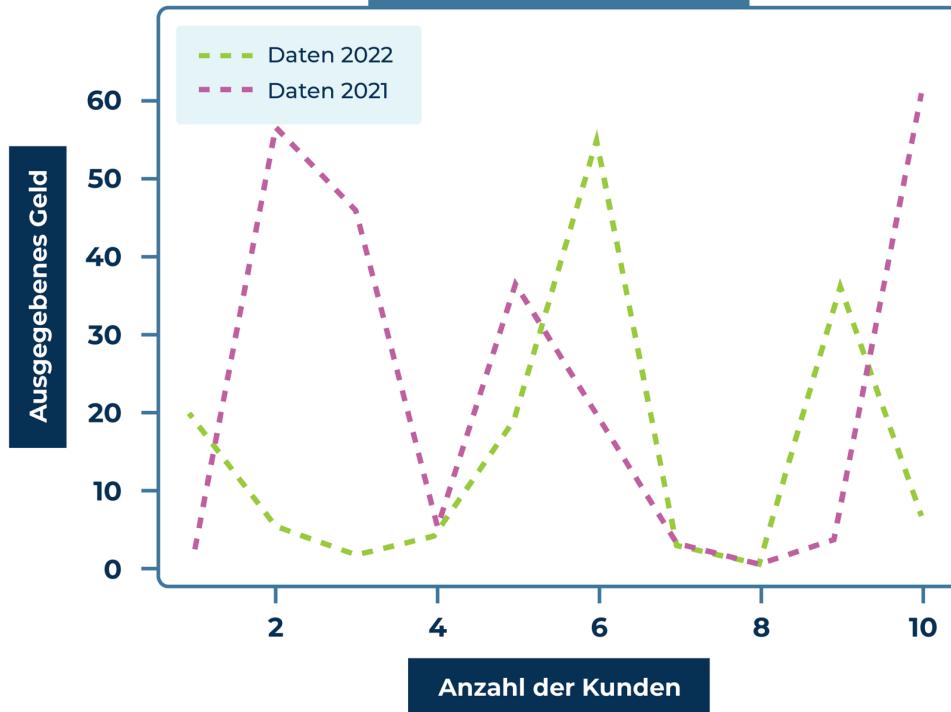
plt.xlabel("Anzahl der Kunden",fontdict=font_achsen)
plt.ylabel("Ausgegebenes Geld",fontdict=font_achsen)
plt.title("Kundendaten", loc="center",fontdict=font_titel)
plt.legend(loc='upper left')

plt.show()
```

In einem sogenannten Font-Dictionary definieren wir zunächst die Schrifteigenschaften für die Achsen und den Titel. Es gibt drei Eigenschaften: die Schriftart, die Schriftfarbe und die Schriftgröße. Mit der Funktion plt.xlabel() beschriftet man dann die Achsen und bestimmt, welches Font-Dictionary verwendet werden soll. Analog geht man beim Titel vor. Möchte man bei mehreren Graphen auch noch eine Legende in dem Plot haben, kriegt man das mit der plt.legend()-Funktion hin. Dafür muss man zuvor in den plt.plot()-Funktionen jedem Graph ein Label geben. Das Ergebnis für den obigen Beispiel-Code sieht folgendermaßen aus:



## Kundendaten



### 1.5.2. Subplots erstellen

Mit der subplot()-Funktion kann man mehrere Plots in einer Abbildung zusammenfassen. Die Abbildung ist dann tabellarisch mit Zeilen und Spalten aufgeteilt. Diese werden der subplot()-Funktion als Argumente mitgegeben. Die subplot()-Funktion hat drei Argumente:

- Anzahl der Reihen der Abbildung
- Anzahl der Spalten der Abbildung
- An welcher Position der folgende Plot stehen soll (erste, zweite, dritte etc.)



```
#Subplot 1
plt.subplot(1,2,1)
plt.plot(x,y,linestyle = 'dotted',color='r',linewidth=2,label="Daten 2022")
plt.title("Daten 2022")

#Subplot 2
plt.subplot(1,2,2)
plt.plot(x,y2,linestyle= 'dashed',color='b',linewidth=1,label="Daten 2021")
plt.title("Daten 2021")

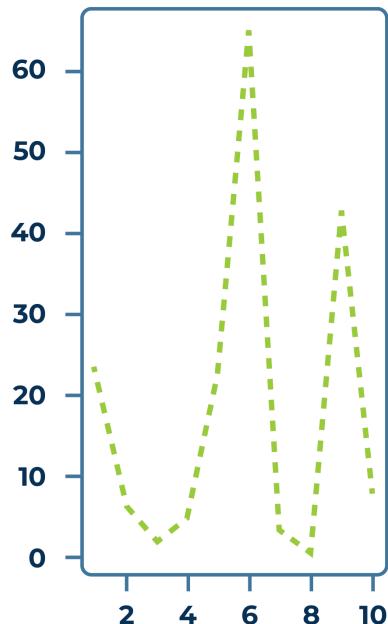
plt.suptitle("Kundendaten")
plt.show()
```

Einen Subplot zu erstellen, funktioniert ganz simpel. Vor der plot()-Funktion kommt die subplot()-Funktion, in welcher steht, an welcher Stelle der folgende Plot stehen soll. Dem Subplot kann man ganz normal mit der title()-Funktion einen Titel geben. Die gesamte Abbildung bekommt einen Titel – einen sogenannten Super-Titel – durch die suptitle()-Funktion. Das sieht dann für obigen Code so aus:

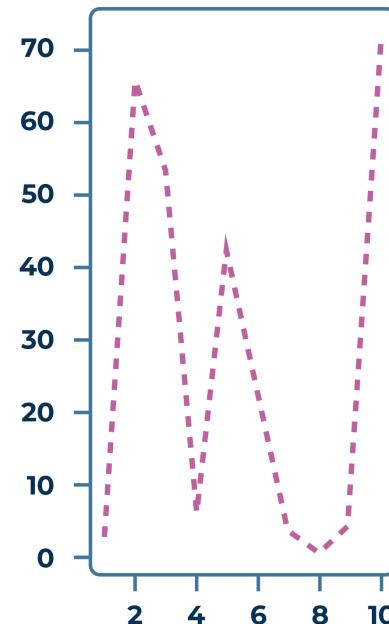


## Kundendaten

Daten 2022



Daten 2021



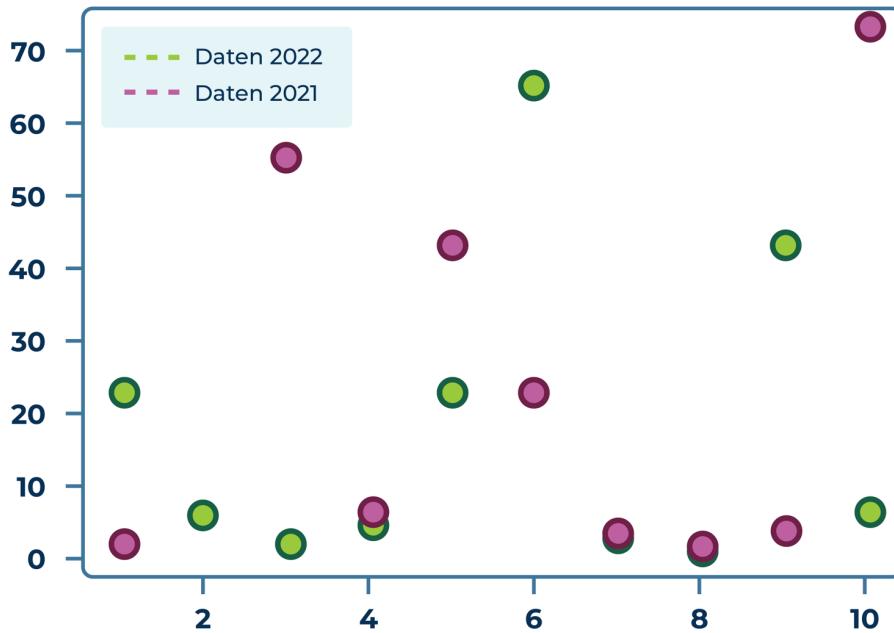
### 1.5.3. Streudiagramme erstellen

Wenn wir nicht gerade analytische Funktionen plotten, wollen wir eigentlich zwischen den Datenpaaren keine Verbindungslien. Dafür sind sogenannte Streudiagramme – in Python Scatter Plots – sehr gut geeignet. Stellen wir mal die beiden Arrays aus den obigen Beispielen als Streudiagramme dar:

```
#Streudiagramme
plt.scatter(x,y,color='g',alpha=0.3,s=40,label='Daten 2022')
plt.scatter(x,y2,color='b',alpha=0.6,s=15,label='Daten 2021')
plt.legend(loc='upper left')
plt.show()
```



Mit dem color-Parameter bestimmt man die Farbe, mit dem alpha-Parameter bestimmt man die Transparenz der Datenpunkte und mit dem s-Parameter legt man die Größe der Datenpunkte fest. Für obigen Code sieht das dann folgendermaßen aus:



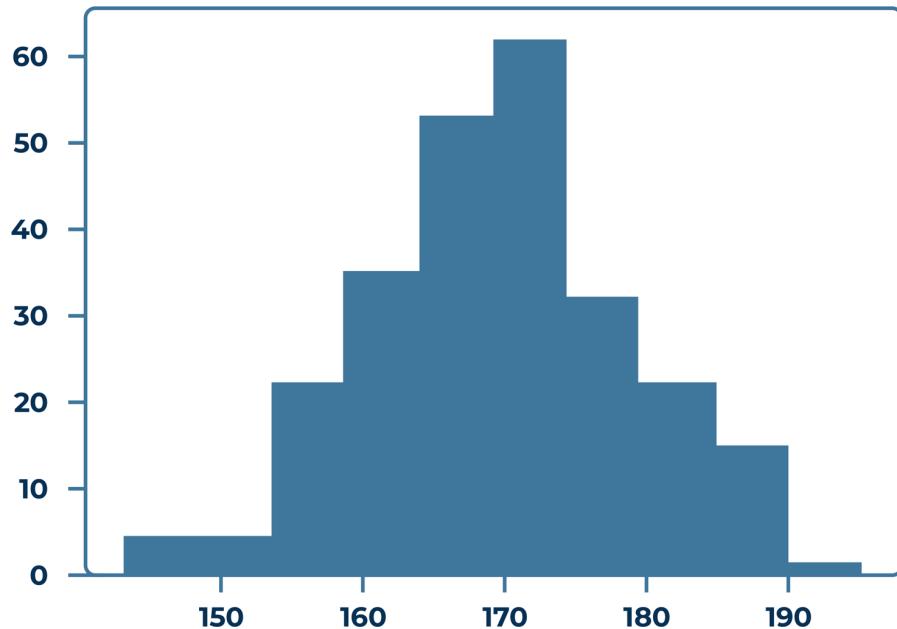
#### 1.5.4. Histogramme erstellen

Histogramme sind graphische Darstellungen von Häufigkeitsverteilungen. Bei einem Histogramm gibt die Höhe der Rechtecke Auskunft über die Anzahl der Messwerte und die

Breite gibt an, in welches Intervall auf der x-Achse die Messwerte fallen. Erstellen wir beispielsweise ein Array mit 250 zufälligen Zahlen deren Mittelwert 170 mit einer Standardabweichung von 10 ist und stellen diese zufälligen Werte dann als Histogramm dar. Die dabei rauskommende Form wird Normalverteilung genannt.



```
#Histogramme  
import numpy as np  
random_array = np.random.normal(170,10,250)  
plt.hist(random_array)
```



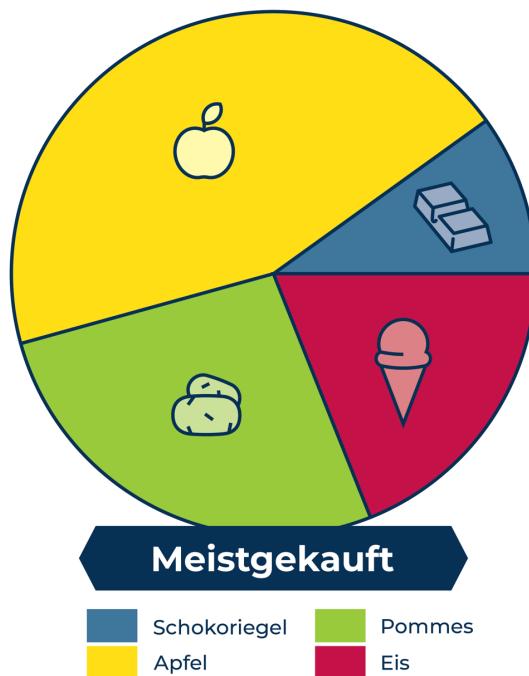
### 1.5.5. Kuchendiagramme erstellen

Mit der `pie()`-Funktion lassen sich Daten als Kuchendiagramm darstellen. Die einzelnen Küchenstücke werden dabei entgegen dem Uhrzeigersinn gezählt. Ein beispielhafter Code dafür:



```
#Kuchendiagramm  
y3 = [12,54,33,23]  
mylabels = ["Schokoriegel", "Apfel", "Pommes", "Eis"]  
plt.pie(y3, labels = mylabels)  
plt.legend(title="Meistgekauft", prop={'size':5}, loc='lower right')  
plt.show()
```

In der mylabels-Liste definiert man die Namen der einzelnen Kuchenstücke. Die pie()-Funktion berechnet für jeden Wert in der Liste y3 den Anteil an der Gesamtsumme der Werte. Mit der legend()-Funktion kann man der Abbildung eine Legende geben, wobei man die Größe mit dem Befehl prop={'size'=5} angibt.





## 1.6. Algorithmen-Komplexität

In diesem Kapitel werden wir uns mit der Analyse von Algorithmen in Bezug auf ihre Laufzeit beschäftigen. Zunächst lernen wir, was die Komplexität eines Algorithmus ist und wieso uns das als Data Scientist interessiert. Nachdem wir die beiden Grundarten der Komplexität – Platz- und Zeitkomplexität – kennengelernt haben, fokussieren wir uns auf die Analyse der Zeitkomplexität von Algorithmen und schauen uns die verschiedenen Komplexitätsklassen an. Dabei lernen wir die wichtigen Algorithmen Binary Search, Merge-Sort und Quick-Sort kennen.

### 1.6.1. Was ist die Komplexität eines Algorithmus?

Als Data Scientist steht man vor der Aufgabe, mit Hilfe der Rechenkapazität eines Computers Muster in großen Datenmengen zu erkennen und daraus Rückschlüsse über eine mögliche Zukunft zu ziehen. Dafür muss man dem Computer genaue Anweisungen geben; er braucht eine Schritt-für-Schritt-Anleitung, was er wie zu tun hat. Diese Anleitung nennt sich Algorithmus. Wenn wir also große Datenmengen analysieren wollen, müssen wir Algorithmen entwerfen und diese für den Computer in Programmcode übersetzen. Wir haben bereits Algorithmen geschrieben, um den Informationsgewinn von Daten zu vergleichen oder bestimmte statistische Merkmale zu berechnen. Manche Algorithmen sind effizient, manche weniger. Schauen wir uns das anhand eines kurzen Beispiels an.

Nehmen wir an, wir sollen eine Funktion in Python programmieren, die die Fakultät einer ganzen Zahl berechnet. Die Fakultät einer Zahl erhält man, wenn alle Zahlen, die kleiner als die gewählte Zahl sind, mit dieser multipliziert werden. Für die Fakultät der Zahl 3 sieht das mathematisch ausgeschrieben dann so aus:

$$3! = 1 \cdot 2 \cdot 3$$



Dafür wollen wir jetzt zwei kleine Algorithmen bauen, die diese wiederkehrende Multiplikation für uns ausführen, und werden sie am Ende dann miteinander vergleichen. Der erste Algorithmus sieht so aus:

```
def fact(n):
    product = 1
    for i in range(n):
        product = product * (i+1)
    return product

print(fact(5))
```

In der Funktion definieren wir zunächst die Variable `product` und weisen ihr den Wert 1 zu, da das auch der Wert ist, mit dem die ganze Multiplikation anfängt. Dann starten wir eine `for`-Schleife, die jede Zahl von 1 bis n durchgeht und mit dem `product`-Wert vom vorherigen Schleifen-Durchlauf multipliziert. Wenn die Schleife durch ist, gibt die Funktion den abschließenden Wert der `product`-Variable wieder.

Der zweite Algorithmus soll nun etwas anders aufgebaut sein und wird eine rekursive Funktion nutzen:

```
def fact2(n):
    if n == 0:
        return 1
    else:
        return n * fact2(n-1)

print(fact2(5))
```



Die Funktion beginnt damit, in einer if-Verknüpfung zu testen, ob die Zahl, von der wir die Fakultät berechnen sollen, 0 ist, denn dann ist die Fakultät per Definition 1. Andernfalls multipliziert die Funktion den aktuellen Wert immer mit dem Wert der Funktion vom vorherigen Wert. Die Funktion ruft sich also immer wieder selbst auf – daher der Name „rekursive Funktion“.

Wir haben also zwei unterschiedliche Algorithmen programmiert, mit denen wir die Fakultät berechnen können, aber woher wissen wir, welcher besser und effizienter ist? Eine Möglichkeit, um diese Frage zu beantworten, ist, die Zeit zu ermitteln, die zum Ausführen des Algorithmus notwendig ist. In Spyder kann man das ganz einfach in der Kommandozeile mit dem folgenden Befehl herausfinden:

```
%timeit fact(50)
```

Wenn wir das für die beiden oben definierten Funktionen ausführen, erhalten wir folgende Ergebnisse (die genauen Werte können von Computer zu Computer unterschiedlich sein):

```
In [5]: %timeit fact(50)
205 ns ± 1.22 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

```
In [9]: %timeit fact2(50)
5.99 µs ± 33.7 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

Der erste Algorithmus mit der for-Schleife braucht also 205 Nanosekunden für einen Schleifen- Durchlauf, während der zweite Algorithmus mit der rekursiven Funktion 6 Mikrosekunden braucht. Der erste Algorithmus ist also wesentlich schneller. Das ist bei kleinen Zahlen kein Problem, aber je größer die Zahl wird, umso mehr Schleifen muss der Algorithmus durchlaufen, weshalb dann die längere Zeit deutlich ins Gewicht fallen wird. Vor dem gleichen Problem stehen wir, wenn wir große Datenmengen



verarbeiten wollen: Wir müssen möglichst effiziente Algorithmen entwerfen und anwenden.

Allerdings – wie oben kurz erwähnt – ist die Laufzeit keine besonders gute Metrik, um die Effizienz eines Algorithmus zu messen, da sie von der verwendeten Hardware abhängig ist. Uns interessiert allerdings unabhängig von der Hardware, wie sich die Laufzeit eines Algorithmus ändert, wenn man die Anzahl der Eingabeelemente wachsen lässt. In der Fachsprache nennt man das die Komplexität eines Algorithmus.

## 1.6.2. Arten von Komplexität

Bei der Analyse der Komplexität eines Algorithmus geht es also darum, herauszufinden, wie sich der Aufwand eines Algorithmus abhängig von der Zahl der Eingabeelemente ändert.

Im Allgemeinen unterscheidet man hier zwischen zwei Arten von Komplexität:

- Die Platzkomplexität
- Die Zeitkomplexität

### Platzkomplexität

Die Platzkomplexität beschreibt, wie viel zusätzlichen Speicherplatz ein Algorithmus in Abhängigkeit von der Anzahl der Eingabeelemente benötigt. Damit ist allerdings nicht der Speicher gemeint, den die Eingabedaten selbst benötigen. Hier ist es nämlich offensichtlich, dass man für eine doppelt so große Menge an Eingabeelementen auch doppelt so viel Speicherplatz benötigt. Was eigentlich gemeint ist, ist der Speicherplatz, den der Algorithmus im Prozess der Problemlösung braucht, um beispielsweise Schleifen-Variablen, Hilfsvariablen oder einfach nur temporäre Datenstrukturen zu speichern. Wir sind allerdings mehr an der



Laufzeitanalyse und nicht an der Speicherplatz-Analyse interessiert. Dafür ist die Zeitkomplexität da.

## Zeitkomplexität

Die Zeitkomplexität eines Algorithmus beschreibt die Änderung der Laufzeit eines Algorithmus in Abhängigkeit von der Anzahl der Eingabeelemente. Einfacher ausgedrückt finden wir also eine Antwort auf die Frage, wie viel langsamer ein Algorithmus läuft, wenn wir die Menge an Eingabedaten vergrößern. Zwei typische Fragen, die aus der Analyse der Zeitkomplexität von Algorithmen kommen, sind:

Wenn wir eine unsortierte Liste haben und die Anzahl an Elementen in dieser Liste verdoppeln, wie viel länger braucht dann ein Algorithmus mindestens, um ein ganz bestimmtes Element in dieser Liste zu finden?

Antwort: Doppelt so lange

Wenn wir eine sortierte Liste haben und die Anzahl an Elementen in dieser Liste verdoppeln, wie viele zusätzliche Schritte muss der Algorithmus dann durchlaufen, um ein bestimmtes Element zu finden?

Antwort: Einen Schritt mehr

Warum das die korrekten Antworten auf die Fragen sind, werden wir im Laufe des Kapitels noch verstehen. Jetzt allerdings stehen wir erst einmal vor der Frage, wie wir diese Zeitkomplexität eines Algorithmus überhaupt messen können. Die Antwort darauf sind die sogenannten Komplexitätsklassen.



### 1.6.3. Komplexitätsklassen

Bei der Analyse der Komplexität teilt man die Algorithmen in verschiedene Komplexitätsklassen ein, die die Laufzeit des Algorithmus in Abhängigkeit von der Anzahl der Eingabeelemente beschreiben. Somit lassen sich verschiedene Algorithmen für das gleiche Problem anhand ihrer Effizienz vergleichen. Jede Komplexitätsklasse wird mit dem sogenannten Landau-O gekennzeichnet. Was das genau heißt, sehen wir gleich. Im Folgenden werden wir uns die wichtigsten Komplexitätsklassen anschauen und lernen, wie wir einen Algorithmus in eine von diesen Klassen einordnen können. Wir beginnen mit den leicht verständlichen und hören mit den komplexeren Klassen auf. Am Ende schauen wir uns die Komplexitätsklassen noch im Vergleich an.

#### Konstante Komplexität O(1)

Die einfachste Komplexitätsklasse ist die sogenannte konstante Komplexität. Sie wird mathematisch als  $O(1)$  bezeichnet. Das  $O(1)$  bedeutet, dass die Laufzeitfunktion nicht schneller als die konstante Funktion  $F(n)=1$  wächst. Diese Funktion ist einfach nur eine horizontale Gerade, die für jeden eingegebenen  $n$ -Wert die 1 liefert. Wenn ein Algorithmus eine konstante Komplexität hat, dann bedeutet das, dass die Laufzeit des Algorithmus unabhängig von der Anzahl der Eingabeelemente ist. Das heißt, egal ob wir 100 oder 1 Million Eingabeelemente haben, die Laufzeit bleibt konstant und damit proportional zur konstanten Funktion  $F(n)=1$ . Jetzt können wir uns natürlich die Frage stellen, welcher Algorithmus in der realen Welt eine konstante Komplexität hat.



Stellen wir uns beispielsweise vor, wir hätten eine Liste der Länge n und wir wollen mit dem Befehl liste [index] das Element an der indizierten Stelle ausgeben. Der Programmcode würde folgendermaßen aussehen:

```
def constant_algo(items):
    result = items[3]
    print(result)

constant_algo([4, 5, 6, 8])
```

Wir haben ganz einfach eine Funktion definiert, die von der eingegebenen Liste das dritte Element zurückgibt. Egal wie lang diese Liste ist, es wird immer gleich lange dauern, um das dritte Element aus der Liste abzurufen. Dabei ist es egal, ob es das dritte oder das hundertste Element ist. Ein weiteres Beispiel für einen Algorithmus mit einer konstanten Komplexität wäre eine Funktion, die für die eingegebene Liste jeweils das erste Element der Liste quadriert. Der Programmcode könnte folgendermaßen aussehen:

```
def constant_algo(items):
    result = items[0] * items[0]
    print(result)

constant_algo([2, 3, 4, 1])
```

Von der eingegebenen Liste wird immer das erste Element herausgesucht und mit sich selbst multipliziert. Auch hier ist es egal, wie groß die anfängliche Liste ist, es dauert immer gleich lange, das erste Element zu quadrieren. Schauen wir uns jetzt mal Komplexitätsklassen an, bei denen die Anzahl der Eingabeelemente einen Einfluss auf die Laufzeit hat.



## Lineare Komplexität O(n)

Wenn wir einen Algorithmus vor uns haben, der eine lineare Komplexität hat, dann bedeutet das, dass der Aufwand linear mit der Anzahl der Eingabeelemente n wächst. Der mathematische Ausdruck  $O(n)$  bedeutet, dass die Laufzeitfunktion in Abhängigkeit von der Anzahl der eingegebenen Elemente n nicht schneller als die lineare Funktion  $F(n)=n$  wächst. Praktisch heißt das, wenn wir die Anzahl der Eingabeelemente verdoppeln, verdoppelt sich auch der Aufwand des Algorithmus. Was wäre denn ein Beispiel für einen Algorithmus mit einer konstanten Komplexität?

Für die Praxis kann man sich merken, dass Schleifen im Programmcode fast immer eine lineare Komplexität bedeuten. Stellen wir uns beispielsweise wieder vor, wir hätten eine Liste von wild durcheinander gewürfelten Zahlen, die ohne Sortierung in die Liste gepackt wurden:

```
liste = [4,1,75,23,7,2,1,97,23,6]
```

Wir wollen jetzt herausfinden, welchen Platz das Element mit dem Wert 6 hat. In unserem Fall ist es das letzte Element. Um also ein bestimmtes Element in einer Liste zu finden, muss man jeden Listeneintrag durchgehen und mit dem gesuchten Element vergleichen. In Python kann man das mit einer for-Schleife realisieren.

```
def linear_algo(items):
    for item in items:
        if item ==6:
            print(items.index(item))

linear_algo([4,1,75,23,7,2,1,97,23,6])
```

Da wir jedes Listenelement aufrufen und mit dem gesuchten Element vergleichen müssen, haben wir es mit einer linearen Laufzeit zu tun. Wenn sich die Anzahl der Elemente in der Liste verdoppelt, muss die Schleife auch doppelt so viele Durchgänge durchlaufen. Ein anderes Beispiel für einen



Algorithmus mit konstanter Laufzeit ist ein Programm, welches alle Elemente einer Liste addiert. Die Schleife muss dafür jedes Listenelement einzeln aufrufen und zum bisherigen Ergebnis dazu addieren. Auch hier führt eine Verdopplung der Anzahl der Eingabeelemente zu einer Verdopplung des Aufwands, den der Algorithmus hat.

## Quadratische Komplexität

Wenn wir es mit einem Algorithmus mit einer quadratischen Komplexität zu tun haben, dann heißt das, dass die Laufzeit mit dem Quadrat der Anzahl der Eingabeelemente wächst. Etwas verständlicher ausgedrückt bedeutet das, dass eine Verdopplung der Anzahl der Eingabeelemente zu einer Vervierfachung des Aufwands führt. Genauso führt eine Verzehnfachung der Eingabeelemente zu einem hundertfach größeren Aufwand! Das beste Beispiel für einen Algorithmus mit einer quadratischen Komplexität sind zwei ineinander verschachtelte for-Schleifen. Nehmen wir als Beispiel an, wir hätten eine Liste, deren Elemente wiederum eine Liste sind. Was uns interessiert, ist die Summe der Elemente aus den einzelnen Unterlisten.

## Logarithmische Komplexität $O(\log(n))$

Nun wollen wir uns mit etwas komplexeren Komplexitätsklassen auseinandersetzen und wir beginnen mit der sogenannten logarithmischen Komplexität. Wenn ein Algorithmus eine logarithmische Laufzeit hat, dann bedeutet das, dass der Aufwand ungefähr so stark wächst wie die Logarithmus-Funktion. Und diese steigt sehr langsam. Praktisch ausgedrückt bedeutet eine logarithmische Komplexität, dass eine Verdopplung der Anzahl der Eingabeelemente zu einem Wachstum um einen konstanten Betrag führt. Um das besser zu verdeutlichen, nehmen wir an, wir hätten einen Algorithmus (mit logarithmischer Komplexität), der 1000 Eingabeelemente verarbeiten soll. Jetzt verdoppeln wir diese Anzahl auf 2000 und sehen, dass der Algorithmus 1 Sekunde länger gebraucht hat. Wenn wir jetzt die Anzahl der Eingabeelemente nochmal auf 4000 verdoppeln, wird der Algorithmus – aufgrund seiner logarithmischen



Laufzeit – wieder nur 1 Sekunde länger brauchen als bei 2000 Eingabeelementen. Wenn man das Ganze aus der Perspektive der Schritte betrachtet, die ein Algorithmus durchlaufen muss, dann wäre bei einer Verdopplung der Anzahl der Eingabeelemente nur ein zusätzlicher Rechenschritt für den Computer notwendig. Ein praktisches Beispiel für einen Algorithmus mit einer logarithmischen Laufzeit ist die sogenannte binäre Suche.

Die binäre Suche läuft ähnlich wie die Suche nach einer Nummer im Telefonbuch. Nehmen wir an, wir würden nach der Telefonnummer von einer Frau Schneider suchen. Wir schlagen das Telefon zufällig auf und sind beim Buchstaben G. Wir wissen also, dass die gesuchte Nummer sich in der zweiten rechten Hälfte des Telefonbuchs befinden muss. Wir schlagen in der zweiten Hälfte wieder eine zufällige Seite auf und landen beim Buchstaben T. Wir wissen also, dass sich die Nummer im linken Viertel finden muss. So fahren wir weiter fort und nähern uns immer mehr der korrekten Seite.

Um die binäre Suche zu verstehen, schauen wir uns jetzt eine Liste mit Zahlen an, wobei wir voraussetzen, dass die Liste sortiert ist:

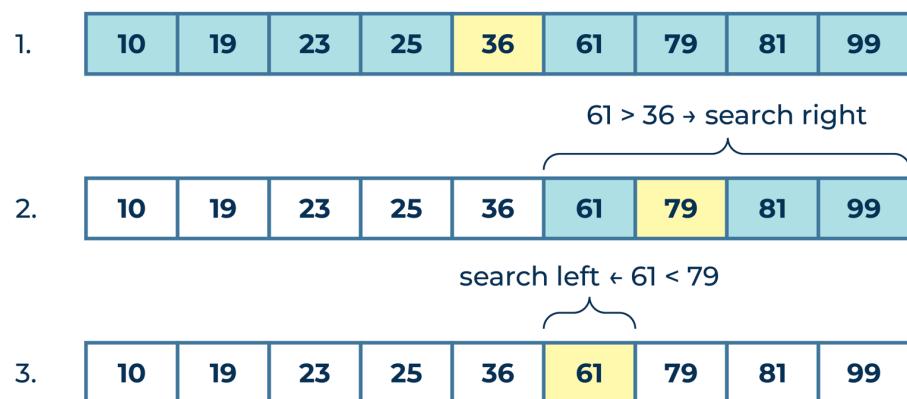
{ 10, 19, 23, 25, 36, 61, 79, 81, 99 }

Wir wollen jetzt die Position ausfindig machen, an der sich die Zahl 61 befindet. Dafür verwenden wir ein zur Suche im Telefonbuch analoges Prinzip, das in der Theorie der Algorithmen als „Teile-und-Herrsche“-Prinzip bekannt ist. Das bedeutet, dass man, egal vor welcher Aufgabe man steht, immer die große Aufgabe in kleine Teilaufgaben unterteilen muss, deren Einzellösungen miteinander kombiniert die Lösung der großen Aufgabe liefern. Im praktischen Beispiel der binären Suche teilen wir die Liste in der Hälfte und sparen uns dabei den Aufwand, die andere Hälfte zu durchsuchen. Die zu durchsuchende Hälfte teilen wir aber wieder, wodurch



der Aufwand nochmal kleiner wird. Die Gesamtaufgabe wurde also in kleinere Teilaufgaben unterteilt. Wie sieht die binäre Suche anhand des konkreten Beispiels oben aus?

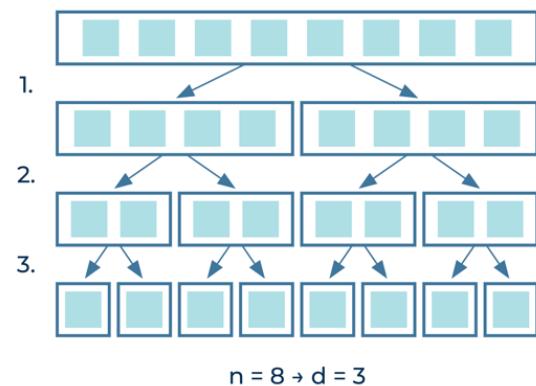
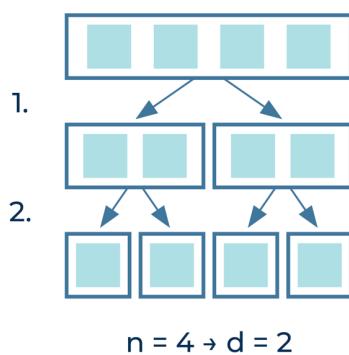
Wir fangen damit an, dass wir die Liste in der Hälfte teilen. Das mittlere Element ist die 36 und da die Liste sortiert ist, wissen wir, dass die 61 sich auf der rechten Seite befinden muss. Wir fokussieren uns also im zweiten Schritt nur auf die rechte Hälfte und teilen diese wieder, diesmal bei der 79. Da 79 größer ist als 61, wissen wir, dass sich die 61 links von der 79 befinden muss. Da aber zwischen der 36 und der 79 nur ein Element ist, haben wir die 61 gefunden. Wir haben es also in drei Schritten geschafft, die Zahl zu finden; hätten wir einfach die lineare Suche angewandt und jedes Element von vorne nach hinten überprüft, hätten wir sechs Schritte gebraucht, also doppelt so viele!



Wieso hat jetzt aber die binäre Suche eine logarithmische Komplexität? Dafür schauen wir uns jetzt ein einfacheres Beispiel mit einer Liste mit 4 Elementen an. Wenn wir diese Liste teilen, erhalten wir 2 Listen mit jeweils 2 Elementen. Wenn wir dann jede dieser Teillisten nochmal teilen, erhalten wir 4 Listen mit jeweils einem Element. Wir haben also zwei Schritte gebraucht, um diesen Zustand zu erreichen.



Verdoppeln wir jetzt die Anzahl der Elemente auf 8 und schauen uns die Prozedur noch einmal an. Wir teilen diese neue Liste und bekommen 2 Listen mit jeweils 4 Elementen. Diese beiden Listen teilen wir nochmal und bekommen 4 Listen mit jeweils 2 Elementen. Diese Teillisten ergeben nach nochmaliger Halbierung 8 Teillisten mit jeweils einem Element. Es hat also diesmal drei Schritte gebraucht, um diesen Zustand zu erreichen.



In der Grafik steht  $n$  für die Anzahl der Elemente in der Liste und  $d$  für die Anzahl der Schritte, die es braucht, um Teillisten mit jeweils nur einem Element zu haben. Der Zusammenhang zwischen der Anzahl der Schritte und der Anzahl der Elemente kann mathematisch folgendermaßen ausgedrückt werden:

$$2^d = n$$

Wenn wir diese Gleichung nach der Anzahl der Schritte  $d$  umstellen, um eine Abhängigkeit von der Anzahl der Elemente  $n$  zu erhalten, bekommen wir folgenden Ausdruck:

$$d = \log_2(n)$$

Die Anzahl der Schritte, die es braucht, entspricht also dem Logarithmus von der Anzahl der Elemente zur Basis 2. Daher kommt auch die



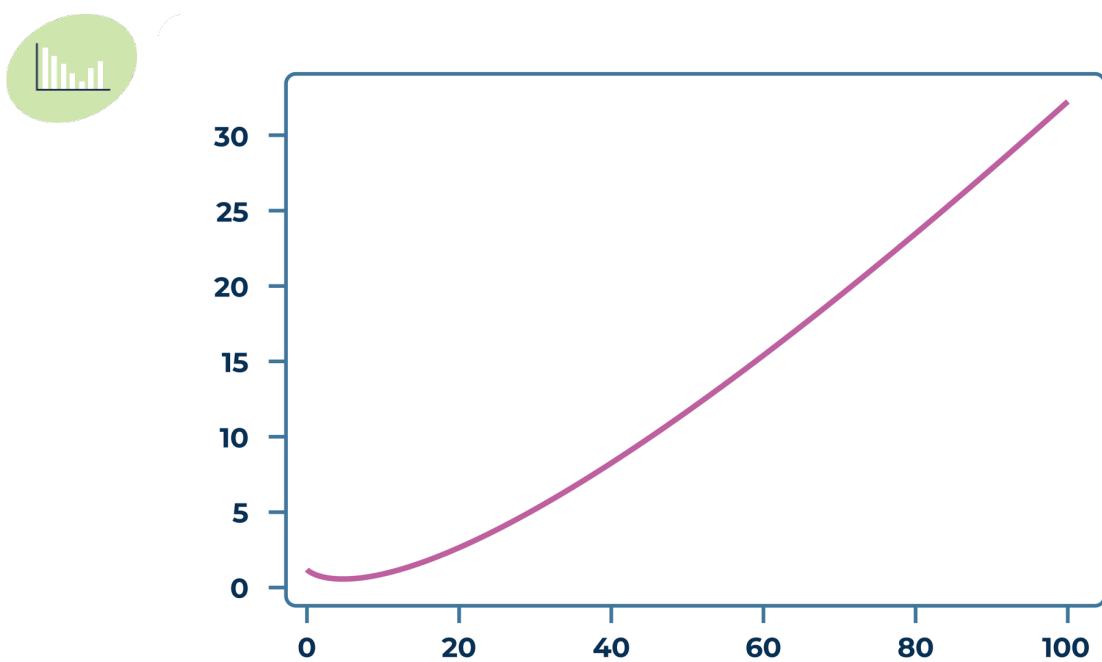
Bezeichnung „logarithmische Komplexität“. Diese Komplexitätsklasse bietet die Grundlage für die quasi-lineare Komplexität, die sich in der Praxis vor allem bei Sortieralgorithmen wie dem Merge-Sort oder dem Quick-Sort finden lässt.

### Quasilineare Komplexität $O(n \log(n))$

Bei einem Algorithmus mit einer quasi-linearen Laufzeit wächst der Aufwand ähnlich wie bei der linearen Komplexität, allerdings ein klein wenig stärker – daher der Name quasi-linear. Mathematisch wird das durch die Multiplikation von der Anzahl der Eingabeelemente  $n$  mit dem Logarithmus von  $n$  beschrieben:

$$O(n \log(n))$$

Graphisch sieht die quasi-lineare Funktion folgendermaßen aus:



Wir sehen, dass die Funktion zu Beginn noch einen leichten Bogen hat, der ein stärkeres Wachstum andeutet, für wachsende Werte auf der x-Achse



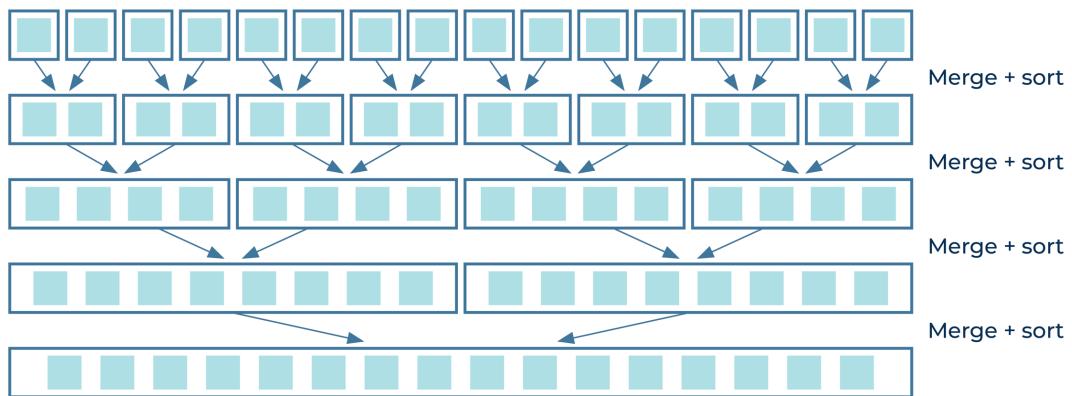
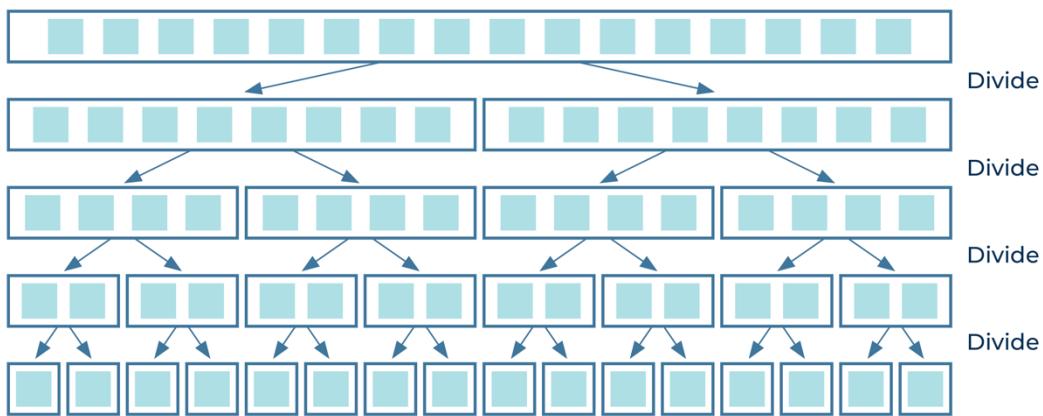
nähert sich die Kurve allerdings immer deutlicher einer einfachen linearen Funktion an und ähnelt mehr einer Geraden.

Ein Beispiel für einen Algorithmus mit quasi-linearer Komplexität ist der Quick-Sort und der Quick-Sort-Algorithmus. Schauen wir uns zunächst die Funktionsweise des Quick-Sort- Algorithmus an, um zu verstehen, was quasi-lineare Komplexität in der Praxis bedeutet.

Der Quick-Sort-Algorithmus funktioniert – wie die binäre Suche – mit dem „Teile-und- Herrsche“-Prinzip. Dabei geht man auch im ersten Schritt genau wie bei der binären Suche vor. Allerdings teilen wir die Liste nicht zwingend genau in der Hälfte. Wo genau die Liste aufgeteilt wird, entscheidet das sogenannte Pivot-Element. Pivot ist französisch und bedeutet Dreh- und Angelpunkt. Um das Pivot-Element dreht sich der ganze Algorithmus. Im Prinzip gibt es drei grundlegende Strategien, um das Pivot-Element auszuwählen:

- Man wählt das mittlere Element der Liste und dann auch bei den Teillisten immer das mittlere Element.
- Man nimmt das letzte Element der Liste.
- Man wählt ein zufälliges Element aus.

Hat man das Pivot Element ausgesucht, wird die Liste anhand dessen so umsortiert, dass die Elemente, die kleiner als das Pivot-Element sind, im linken Bereich landen und die Elemente, die größer sind, im rechten Bereich. Das Pivot-Element wird zwischen den zwei Bereichen positioniert, womit es dann schon mal an der richtigen Position ist.

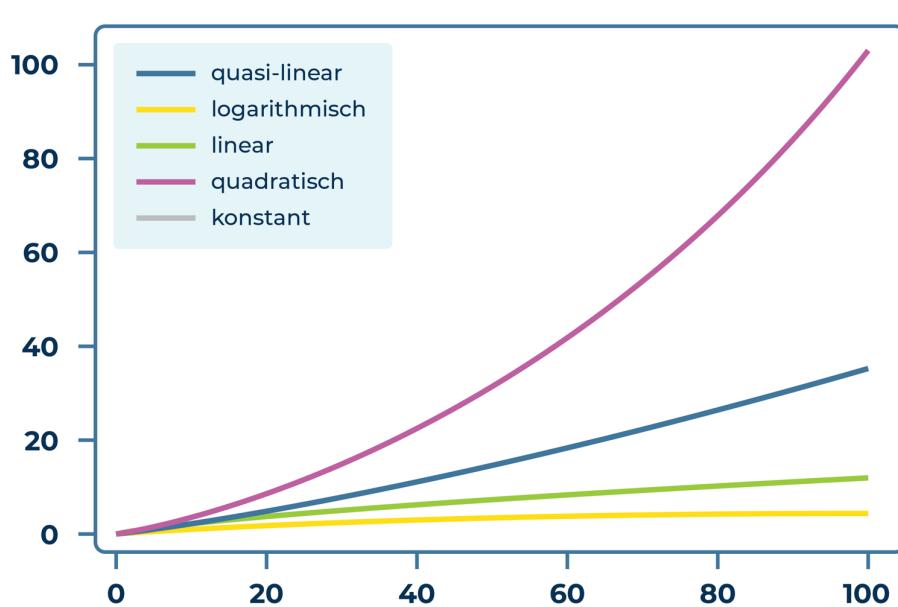


Wieso hat jetzt dieser Algorithmus eine quasi-lineare Komplexität? Wir erinnern uns, dass die quasi-lineare Komplexität aus einem linearen und einem logarithmischen Teil besteht. Beim Quick-Sort-Algorithmus wird zuerst die große Liste immer weiter geteilt, bis man nur noch Teillisten mit einem Element hat; das ist der gleiche Vorgang wie bei der binären Suche, und von dieser wissen wir, dass sie eine logarithmische Komplexität hat. Wenn wir dann aber die einzelnen Teillisten wieder zusammenführen, müssen wir in jedem Schritt  $n$  Vergleiche durchführen – so viele Vergleiche, wie es Elemente in der Liste gibt. Die Anzahl der Schritte, die der Quick-Sort-Algorithmus also durchlaufen muss, ist der Logarithmus multipliziert mit der Anzahl der Elemente. Mathematisch ausgedrückt ist das also  $O(n \log(n))$ , was die gesuchte quasi-lineare Komplexität ist.



### 1.6.4. Komplexitätsklassen im Vergleich

Wir haben jetzt die wichtigsten Komplexitätsklassen kennengelernt, aber wie sehen diese Klassen denn eigentlich im Vergleich aus? Wenn ich zwei Algorithmen aus unterschiedlichen Komplexitätsklassen habe, welcher ist dann schneller? Dafür schauen wir uns die einzelnen Klassen als Plot an.



In der Abbildung sind die einzelnen Komplexitätsklassen gegen die Anzahl der Eingabeelemente auf der x-Achse aufgetragen. Wir sehen, dass die quadratische Laufzeit das schnellste Wachstum hat und sich deutlich von den anderen abhebt. Darunter befindet sich der quasi- lineare Graph, der etwas stärker als der lineare Graph ansteigt. Letzterer steigt im Vergleich zum quadratischen Graphen allerdings kaum.

Es gibt auch noch andere Komplexitätsklassen, allerdings sind Algorithmen mit einer solchen Laufzeit so langsam, dass wir vermeiden wollen, diese zur Problemlösung einzusetzen.

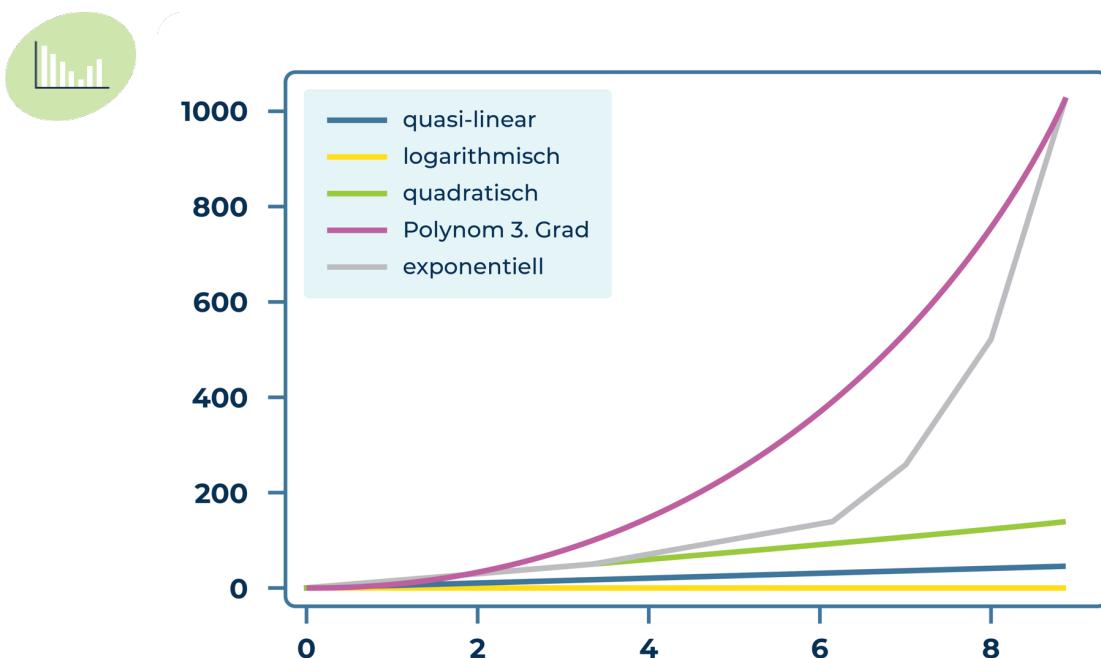


$O(n^m)$  - polynomieller Aufwand

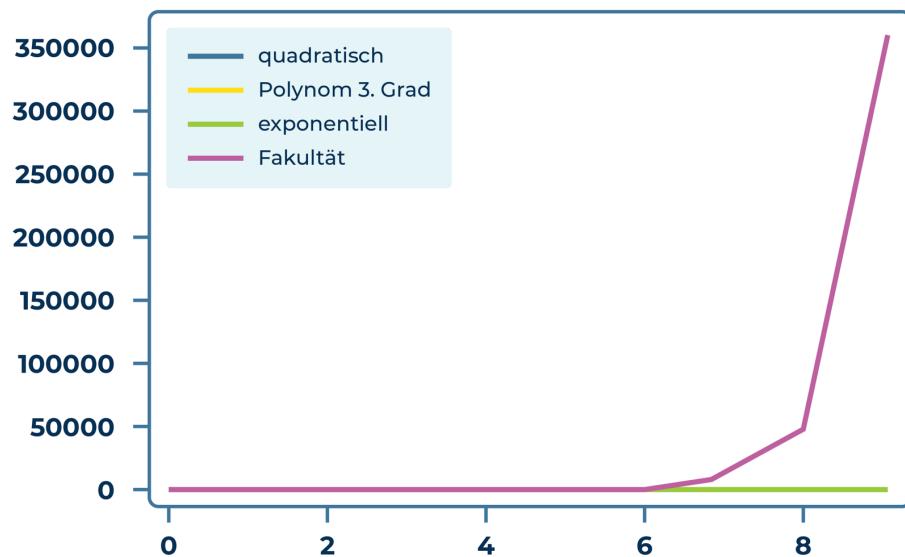
$O(2^n)$  - exponentieller Aufwand

$O(n!)$  - faktorieller Aufwand

Um zu verstehen, wieso wir Algorithmen mit einer solchen Komplexität vermeiden wollen, schauen wir uns wieder die graphische Darstellung dieser Komplexitätsklassen an, allerdings im Vergleich zu den uns bereits bekannten.



In der obigen Abbildung können wir sehen, dass der polynomiale und der exponentielle Graph unheimlich viel stärker als der quadratische oder der quasi-lineare Graph wachsen. Wenn man dann noch den Graphen der Fakultäts-Funktion hinzunimmt, werden alle übrigen Kurven quasi konstant im Vergleich dazu.



Abschließend können wir also sagen, dass bei großen Mengen an Eingabeelementen (mehrere Milliarden) Algorithmen mit konstanter, linearer, logarithmischer und quasi-linearer Laufzeit in einer überschaubaren Zeit Ergebnisse liefern können. Bei Algorithmen mit einer höheren Komplexität wie zum Beispiel quadratisch oder polynominal kann man schnell auf einen theoretischen Aufwand von mehreren Jahren kommen.



## 2. Machine Learning

### Lernziele für dieses Kapitel:

<b>Grobziel:</b> Die Lernenden können Machine Learning Algorithmen anwenden und mit diesen Datenanalysen ausführen.	
Feinziele (ca. 3-5)	Inhalte
1. Die Lernenden können die verschiedenen Kategorien von Machine-Learning-Algorithmen aufzählen.	<ul style="list-style-type: none"><li>• Supervised Learning</li><li>• Unsupervised Learning</li><li>• Reinforcement Learning</li></ul>
2. Die Lernenden können mit Regressionsalgorithmen Daten analysieren.	<ul style="list-style-type: none"><li>• Regressionsanalyse</li><li>• Lineare Regression</li><li>• Multiple Regression</li><li>• Polynom-Regression</li></ul>
3. Die Lernenden können mit Klassifizierungsalgorithmen Daten analysieren.	<ul style="list-style-type: none"><li>• Logistische Regression</li><li>• K-nearest-neighbors</li><li>• Decision Trees</li></ul>
4. Die Lernenden können mit Clustering-Algorithmen Daten analysieren.	<ul style="list-style-type: none"><li>• K-Means Clustering</li><li>• Hierarchisches Clustering</li></ul>



In diesem Abschnitt werden wir lernen, was Machine Learning eigentlich ist, wie es mit Künstlicher Intelligenz und neuronalen Netzen zusammenhängt und welche verschiedenen Machine-Learning-Algorithmen es gibt. Zunächst einmal werden wir die Unterschiede zwischen Supervised, Unsupervised und Reinforcement Learning kennenlernen. Dann bekommen wir einen kurzen Einblick in die verschiedenen Algorithmen, die in diese Kategorien fallen und die wir im Laufe dieses Kurses im Detail betrachten werden. Anschließend lernen wir, wieso wir bei Supervised-Learning-Algorithmen unsere Daten in Trainings- und Testdaten einteilen müssen und wieso Over- und Underfitting häufig auftretende Probleme bei der Analyse von Daten mit Machine-Learning-Algorithmen sind.

## 2.1. Was ist Machine Learning?

In der Data Science beschäftigen wir uns damit, aus großen Mengen von Daten Informationen zu ziehen und Vorhersagen für die Zukunft zu treffen. Bisher haben wir gelernt, in welchen Formen Daten gespeichert werden und wie wir sie aufrufen können. Außerdem haben wir einige grundlegende Methoden aus der Statistik kennengelernt, mit denen wir erste Einblicke in verschiedene Datensätze gewinnen konnten. Auf diesen statistischen Methoden baut das Machine Learning auf, einer der wichtigsten Bereiche in der Data Science zur Analyse von Daten. Machine Learning wird häufig mit Künstlicher Intelligenz oder neuronalen Netzen durcheinandergebracht, doch was hat es wirklich damit auf sich? Was ist Machine Learning?

Klären wir zunächst einmal die Terminologie.

Machine Learning ist ein Teilbereich der Künstlichen Intelligenz. Und neuronale Netze sind ein Teilbereich des Machine Learnings (den wir später im Laufe dieses Kurses noch kennenlernen werden). Machine Learning lässt sich vereinfacht gesagt als eine Ansammlung von verschiedenen



Algorithmen sehen, mit denen Computer (Maschinen) mehr oder weniger selbstständig Einblicke in Daten gewinnen (Learning) und anhand dessen Vorhersagen für die Zukunft treffen können. Dabei lassen sich Machine-Learning-Algorithmen in verschiedene Kategorien klassifizieren:

- Überwachtes Lernen (Supervised Learning)
- Unüberwachtes Lernen (Unsupervised Learning)
- Verstärkendes Lernen (Reinforcement Learning)

Schauen wir uns mal im Detail an, welche Machine-Learning-Algorithmen in welche Kategorie fallen und wo sie in der Praxis Anwendung finden.

### **2.1.1. Supervised Learning**

Überwachtes Lernen bedeutet, dass man einen Machine-Learning-Algorithmus auf einen Datensatz ansetzt und diesen in diesem Muster erkennen lässt. Hat er dann ein Muster erkannt, sagt man dem Algorithmus, ob dieses richtig oder falsch ist. Das ist der sogenannte „Supervised“-Teil. Wenn der Algorithmus dann trainiert ist, kann man ihn nutzen, um Vorhersagen für die Zukunft zu treffen.

Supervised-Learning-Algorithmen lassen sich wiederum in zwei Kategorien aufteilen:

- Regressionsanalyse
- Klassifizierung

Die Regressionsanalyse wird hauptsächlich genutzt, um Prognosen über die Zukunft zu treffen. Das findet oft Einsatz bei Fragen nach Umsatzerwartungen, Wetterprognosen oder um zum Beispiel den Stromverbrauch eines Stadtviertels vorhersehen zu können. Im Allgemeinen geht es bei der Regressionsanalyse darum, kontinuierliche



numerische Werte vorhersagen zu können. In diesem Kurs werden wir die drei wichtigsten Regressions-Algorithmen kennenlernen:

- Lineare Regression
- Multiple Regression
- Polynom-Regression

Allgemein gesagt werden diese Regressions-Algorithmen verwendet, um Beziehungen zwischen verschiedenen Variablen zu erkennen. Bei der linearen Regression hat man als Ziel, einen linearen Zusammenhang zwischen den beobachteten Daten und einer einzelnen unabhängigen Variable festzustellen.

Die multiple Regression funktioniert wie die lineare Regression, nur mit mehreren unabhängigen Variablen zur Beschreibung des linearen Zusammenhangs.

Bei der Polynom-Regression versucht man, die beobachteten Daten durch ein Polynom zu beschreiben.

Bei der Klassifizierung geht es – wie der Name schon sagt – darum, Daten zu klassifizieren. Das findet vor allem Anwendung bei der Erkennung von verschiedenen Objekten in Bildern, bei der Texterkennung, bei der Spam-Erkennung im E-Mail-Postfach oder bei der Klassifizierung von Kunden bzw. Kundinnen, um das Marketing speziell auf bestimmte Kundengruppen zuschneiden zu können. Paradoxe Weise gehört ein Regressions-Algorithmus – die logistische Regression – zu den Klassifizierungsalgorithmen. Folgende Klassifizierungsalgorithmen werden wir im Laufe dieses Kurses kennenlernen:

- Logistische Regression
- K-nearest-neighbors



- Decision Trees
- Random Forests
- Naive Bayes
- Support Vector Machines
- Neuronale Netze

Die logistische Regression nutzt eine sogenannte Sigmoid-Funktion, um die Wahrscheinlichkeit, dass ein bestimmtes Ereignis eintritt, zu berechnen. Sie eignet sich perfekt zur Klassifizierung von binären Variablen.

Die K-nearest-neighbors-Methode verwendet Metriken wie beispielsweise den Euklidischen Abstand, um den Abstand zwischen Datenpunkten zu berechnen. Um die Daten zu klassifizieren, nutzt dieser Algorithmus die k-nächsten Nachbarn von jedem Datenpunkt.

Wenn man die Decision-Tree-Methode zur Klassifizierung von Daten verwendet, baut man mit vielen if-else-Statements ein Klassifizierungs-Modell in Form eines Baums mit Ästen und Blättern auf. Mit jeder Verästelung wird die Klassifizierung verfeinert.

Die Random-Forest-Methode ist einfach eine Kombination aus mehreren Decision Trees.

Der Naive-Bayes-Algorithmus basiert auf dem Satz von Bayes (Bayes' Theorem) aus der Statistik und nimmt an, dass jedes Feature von einem Datensatz unabhängig von den anderen Features ist; das heißt, es besteht keine Korrelation zwischen diesen. Dieser Algorithmus funktioniert nicht besonders gut, wenn man es mit komplexen Daten zu tun hat, weil in den meisten Datensätzen eben schon Korrelationen zwischen den einzelnen Features bestehen.



Beim Support-Vector-Machine-Algorithmus werden die Daten durch Punkte in einem mehrdimensionalen Raum repräsentiert. Hat ein Datensatz  $n$  Features, werden die Daten durch Punkte in einem  $n$ -dimensionalen Raum repräsentiert. Mit Hilfe von sogenannten Hyperebenen werden die Datenpunkte dann voneinander getrennt.

Zu guter Letzt gibt es noch die neuronalen Netze, die ihren Ursprung in der Struktur der Neuronen im Gehirn haben. Ein neuronales Netz ist ein Netzwerk aus Knoten und Kanten, wobei jeder Kante als Verbindung zwischen den Knoten ein Gewicht zugeordnet wird. Wenn Daten durch das Netz durchgejagt werden, werden nach jedem Durchlauf die Gewichte der Verknüpfungen angepasst, bis das Ergebnis, das das neuronale Netz liefert, korrekt ist.

## 2.1.2. Unsupervised Learning

Beim unüberwachten Lernen setzt man einen Machine-Learning-Algorithmus auf einen Datensatz an und lässt ihn Muster in diesem erkennen, ohne dass man ihm sagt, ob die Muster korrekt sind oder nicht. Er erkennt es selbst. Das wird vor allem beim Clustering von Daten angewandt. Unsupervised-Learning-Algorithmen werden angewendet, um in großen Datenmengen die Dimensionen zu reduzieren, Informationen zu komprimieren und Strukturen zu erkennen. Gerade in der Marktsegmentierung, zur Bildklassifizierung oder zur Erstellung von Empfehlungssystemen werden gerne Unsupervised-Learning-Algorithmen genutzt. In diesem Kurs werden wir die folgenden Unsupervised-Learning-Algorithmen kennenlernen:

- K-Means Clustering
- Agglomeratives Hierarchisches Clustering
- Divisive Hierarchical Clustering
- DBSCAN (Density based spatial clustering of applications with noise)



Beim K-Means Clustering legt man zu Beginn die Anzahl k der Cluster fest. Der Algorithmus nutzt dann Abstandsmetriken, um den Abstand der Datenpunkte zum jeweiligen Zentrum der Cluster zu berechnen. Abhängig vom Abstand werden die Datenpunkte dann einem Cluster zugeordnet.

Beim Agglomerativen Hierarchischen Clustering wird jeder Datenpunkt zunächst als eigener Cluster angesehen. Dann werden die einzelnen Sub-Cluster mit Hilfe von Abstands-Metriken und dem jeweiligen, zu Beginn festgelegten Kriterium miteinander verschmolzen. Diese Art des Hierarchischen Clustering wird auch als Bottom-Up-Methode bezeichnet.

Beim Divisive Hierarchical Clustering geht man zu Beginn davon aus, dass alle Datenpunkte zu Beginn ein einzelner großer Cluster sind. Dann wird dieser Riesen-Cluster mit Hilfe von Abstandsmetriken und dem zu Beginn festgelegten Kriterium in Sub-Cluster aufgeteilt. Diese Art des Hierarchischen Clustering wird auch als Top-Down-Methode bezeichnet.

Der DBSCAN-Algorithmus basiert auf der Dichte der Datenpunkte. Wo der K-Means-Clustering-Algorithmus beispielsweise nur für Datenpunkte funktioniert, die eine sphärische Form haben, läuft die DBSCAN-Methode auch bei Daten mit beliebiger Form und reagiert weniger sensibel auf Ausreißer in den Daten.

### 2.1.3. Reinforcement Learning

Beim Reinforcement Learning erhält ein Machine-Learning-Algorithmus eine Belohnung, wenn er die richtigen Muster in einem Datensatz erkennt. Da dies in der klassischen Data Science seltener bis kaum Anwendung findet, werden wir es in diesem Kurs nicht behandeln. Angewandt werden Reinforcement-Algorithmen vor allem in der Robotik, bei der Verkehrssteuerung oder beim autonomen Fahren.



## 2.1.4. Trainings- und Testdaten

Beim Supervised Learning werden Datensätze in Trainings- und Testdaten aufgeteilt, um überprüfen zu können, wie gut ein Machine-Learning-Algorithmus Vorhersagen für Daten treffen kann, die ihm vorher noch nicht bekannt waren. Hat man also einen Datensatz und möchte diesen mit einem Machine-Learning-Modell beschreiben, teilt man den Datensatz in einen Trainings-Datensatz und einen Test-Datensatz auf. Mit dem Trainings-Datensatz trainiert man den Machine-Learning-Algorithmus. Da man dann nach dem Training wissen möchte, wie gut dieser funktioniert, kann man nicht einfach beliebige Daten aus dem Trainings-Datensatz verwenden, da der Algorithmus diese ja bereits kennt und somit voreingenommen ist. Daher nimmt man die Test-Daten nun, die dem Algorithmus unbekannt sind, und schaut, wie gut er diese vorhersagen kann. Üblicherweise teilt man seinen Datensatz so auf, dass 80 % der Daten zum Training und 20 % der Daten zum Testen des Modells verwendet werden.

## 2.1.5. Over- und Underfitting

Häufige Probleme, die bei der Anwendung von Machine-Learning-Algorithmen auftreten, sind das sogenannte Over- und Underfitting von Daten.

Underfitting bedeutet, dass das Modell nicht gut genug an die Daten angepasst ist und somit keine verwertbaren Vorhersagen treffen kann. Das Modell funktioniert weder bei den Trainings- noch bei den Testdaten. Einer der Gründe dafür ist oft, dass die Größe des Trainings- Datensatzes nicht ausreicht. Auch kann es sein, dass die Daten nicht richtig bereinigt wurden und somit eine Art Rauschen entstanden ist, das den Machine-Learning-Algorithmus negativ beeinflusst. Ein anderer Grund kann auch sein, dass



man das falsche Modell verwendet und es einfach zu simpel zur Beschreibung der Daten ist.

Beim Overfitting auf der anderen Seite ist das Machine-Learning-Modell zu gut an die Trainings-Daten angepasst und kann keine verwertbaren Vorhersagen mehr für die Testdaten liefern. Das Modell funktioniert also hervorragend bei den Trainingsdaten, versagt aber komplett bei den Testdaten. Dieses Problem tritt häufig auf, wenn der Machine-Learning-Algorithmus zu komplex und zu flexibel ist, wodurch er sich an die Trainings-Daten zu gut anpasst.

## 2.2. Lineare Regression

In diesem Abschnitt werden wir den ersten Machine-Learning-Algorithmus kennenlernen – die sogenannte lineare Regression. Wir werden verstehen, wie sie funktioniert, wie man mit ihr Vorhersagen für Daten treffen kann, zwischen denen ein linearer Zusammenhang besteht, und wie man sie programmiert. Anschließend lernen wir, was das Bestimmtheitsmaß ist, welchen Zusammenhang es zur linearen Regression hat und wie man es programmiert. Zum Schluss lernen wir noch, wie man die lineare Regression einfach und mit wenig Code mit Hilfe des `scipy`-Moduls implementieren kann.

Der erste Machine-Learning-Algorithmus, den wir kennenlernen werden, ist die sogenannte lineare Regression und dieser ist auch der am einfachsten zu verstehende. Bei der linearen Regression versucht man, die Werte einer Variablen anhand der Werte einer oder mehrerer anderer Variablen vorherzusehen. Die Variable, deren Werte wir vorhersagen möchten, heißt Kriterium oder abhängige Variable, die Variablen, die wir zur Vorhersage nutzen, heißen Prädiktoren oder unabhängige Variablen.



Wenn wir es nur mit einer einzelnen unabhängigen Variable zu tun haben, dann sprechen wir von einer einfachen linearen Regression, oft einfach nur kurz lineare Regression. Wenn wir es dagegen mit mehreren unabhängigen Variablen zur Vorhersage des Kriteriums zu tun haben, dann nennt man das Verfahren eine multiple Regression.

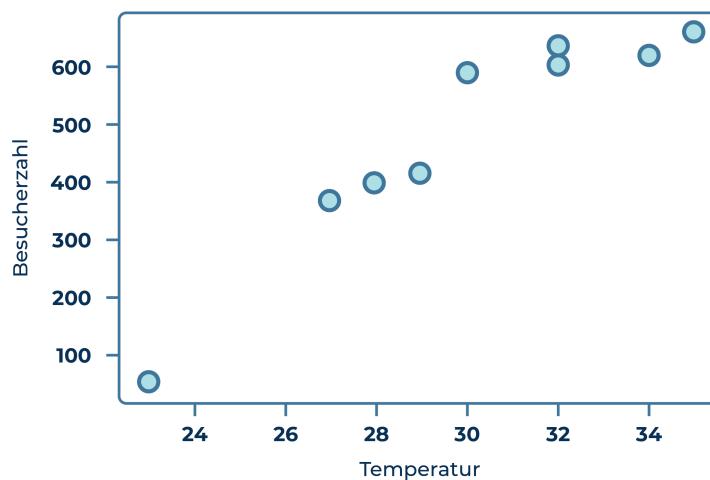
Bei der linearen Regression geht es also darum, dass wir einen Zusammenhang zwischen dem Kriterium und den unabhängigen Variablen feststellen wollen – genauer gesagt einen linearen Zusammenhang. Um besser zu verstehen, was eine lineare Regression genau ist, schauen wir uns ein einfaches Beispiel an.



Stellen wir uns vor, wir betreiben ein Freibad und möchten herausfinden, wie die Besuchendenzahl unseres Freibads mit der Temperatur zusammenhängt. Natürlich können wir uns gleich denken, dass bei höheren Temperaturen mehr Besuchende ins Freibad strömen werden; uns interessiert jedoch eine Vorhersage der Besuchendenzahl in Abhängigkeit von der Temperatur. Was wir also machen, ist, dass wir über einige Tage die durchschnittliche Temperatur messen und gemeinsam mit der Besuchendenanzahl in eine Tabelle eintragen.

<b>Temperatur x_k</b>	28	23	32	35	29	30	27	34	32
<b>Gästezahl y_k</b>	400	60	630	660	420	590	376	620	612

In einem Streudiagramm graphisch aufgetragen sehen die Daten folgendermaßen aus:



Anhand dieser Tabelle können wir also vorhersagen, dass bei einer Temperatur von 30 Grad beispielsweise 590 Besuchende ins Freibad gehen werden. Allerdings wird für den morgigen Tag eine Temperatur von 33 Grad vorhergesagt und wir haben keinerlei Daten dazu in unserem Datensatz. Dennoch wollen wir irgendwie vorhersagen, wie viele Besuchende wir letztendlich erwarten können. Und genau für diese Aufgabe hilft die lineare Regression.

Um mit Hilfe der linearen Regression die Besuchendenzahl für den kommenden Tag vorherzusagen, müssen wir eine sogenannte Regressions-Gleichung aufstellen. Die Regressions- Gleichung ist eine einfache lineare Funktion, die graphisch in einem Diagramm aufgetragen eine Gerade ergibt, in diesem Fall eine sogenannte Regressions-Gerade, die durch die Datenpunkte hindurchgeht.

Mathematisch ausgedrückt nimmt die Regressions-Gerade die folgende Form an:

$$y_k = b x_k + a$$



Hierbei ist  $y_k$  der k-te Wert des Kriteriums, den wir in Abhängigkeit vom k-ten Wert des Prädiktors  $x_k$  vorhersagen möchten. Außerdem finden wir in der Gleichung für die Regressions-Geraden noch die Werte b und a. Hierbei steht b für die Steigung der Geraden und a für den y-Achsenabschnitt, also den Schnittpunkt der Regressions-Geraden mit der y-Achse. Die Steigung b und der y-Achsenabschnitt a sind abhängig von den gemessenen Datenpunkten der unabhängigen und der abhängigen Variablen. Um diese bestimmen zu können, müssen wir auf unsere Kenntnisse aus der Statistik zurückgreifen.

## 2.2.1. Die Regressions-Gerade

Um nun die Gleichung der Regressions-Geraden aufstellen zu können, benötigen wir die Mittelwerte und die Standardabweichungen von sowohl der abhängigen als auch der unabhängigen Variable. Anschließend müssen wir auch noch die Standardabweichungen nutzen, um die Korrelation zwischen Kriterium und Prädiktor feststellen zu können. Gehen wir hierfür wieder zurück zu unserem Beispiel mit der Besuchendenzahl im Freibad in Abhängigkeit von der durchschnittlichen Temperatur am jeweiligen Tag. Der Mittelwert und die Standardabweichung der Temperatur nehmen die folgenden Werte an:

$$\bar{x} = 30$$

$$s_x = 3.52$$

Für die Besuchendenanzahl berechnen sich der Mittelwert und die Standardabweichung zu:

$$\bar{y} = 485.33$$

$$s_y = 182.87$$



Jetzt müssen wir nur noch die Korrelation zwischen den beiden Variablen feststellen, wofür wir als erstes die Kovarianz berechnen müssen:

$$s_{xy} = \frac{1}{n-1} \sum (x_k - \bar{x})(y_k - \bar{y}) = 610.66$$

Das setzen wir in die bekannte Formel für die Korrelation ein und erhalten:

$$r_{xy} = \frac{s_{xy}}{s_x \cdot s_y} = 0.94$$

Nun haben wir alles, was wir brauchen, um die Parameter für unsere Regressions-Gerade zu bestimmen. Die Steigung der Geraden  $b$  ist abhängig von den Standardabweichungen der beiden Variablen und der Korrelation zwischen ihnen und kann mit folgender Formel berechnet werden:

$$b = \frac{s_y}{s_x} \cdot r_{xy}$$

Der y-Achsenabschnitt  $a$  auf der anderen Seite ist auch abhängig von den Standardabweichungen und der Korrelation, aber zusätzlich auch noch von den Mittelwerten der beiden Variablen:

$$a = -\frac{s_y}{s_x} \cdot r_{xy} \cdot \bar{x} + \bar{y}$$

Wenn wir jetzt die oben berechneten Werte für die Mittelwerte, Standardabweichungen und Korrelation in die beiden Formeln einsetzen, erhalten wir folgendes Ergebnis:



$$b = \frac{s_y}{s_x} \cdot r_{xy} = 55.20$$

$$a = -\frac{s_y}{s_x} \cdot r_{xy} \cdot \bar{x} + \bar{y} = -1170.82$$

Das bedeutet, dass die Gleichung für unsere Regressions-Gerade am Ende so aussieht:

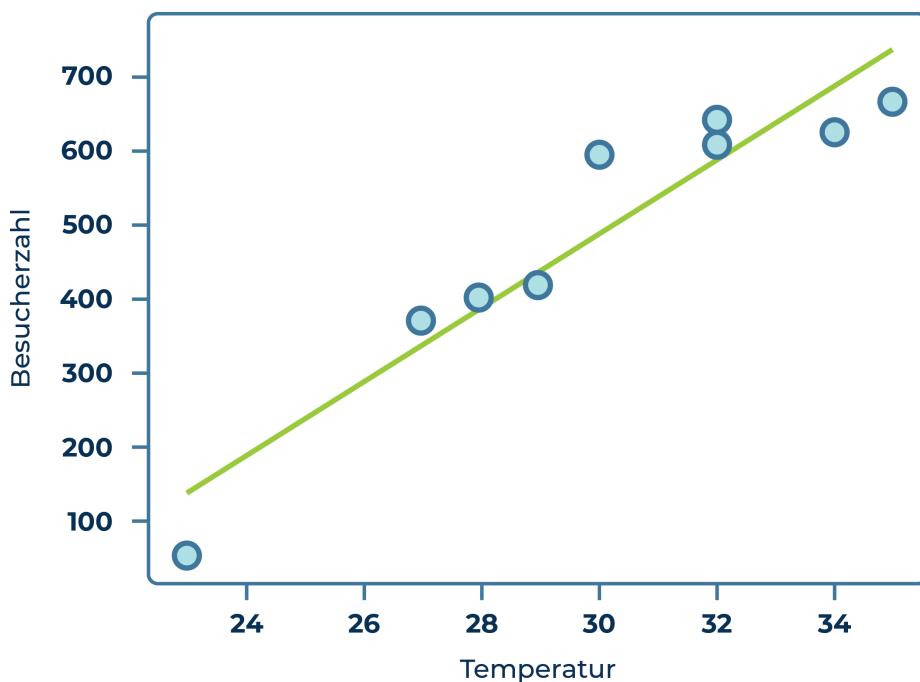
$$y_k = 55.20 \cdot x_k - 1170.82$$

In diese Gleichung können wir nun beliebige Temperaturen einsetzen und erhalten dann eine Vorhersage für die zu erwartende Anzahl an Besuchenden. Je mehr Daten wir sammeln können, umso genauer wird

$$y_k = 55.20 \cdot 33 - 1170.82 = 650.94$$

auch die Anpassung der Geraden an die Datenpunkte und damit auch die Vorhersage. Kommen wir wieder zurück auf die ursprüngliche Fragestellung, in welcher wir die Anzahl der Besuchenden vorhersehen wollten, wenn die durchschnittliche Temperatur 33 Grad beträgt. Dafür müssen wir lediglich in die obige Gleichung für die Regressions-Gerade die erwartete Temperatur eintragen und erhalten dann als Ergebnis:

Das lineare Regressions-Model sagt uns also 651 Besuchende voraus, wobei wir aufgerundet haben, weil wir kaum 0.94 Besuchende zählen werden. Die Regressions-Gerade im Streudiagramm sieht dann folgendermaßen aus:



## 2.2.2. Das Bestimmtheitsmaß

Wir wissen nun, wie wir auf einen Datensatz ein lineares Regressions-Modell anwenden können, allerdings wissen wir nicht, wie gut dieses Modell eigentlich ist.

Die Frage, die wir uns stellen, ist also: Wie gut sind eigentlich die Vorhersagen, die wir mit unserer linearen Regression treffen?

Um diese Frage beantworten zu können, benötigen wir das sogenannte Bestimmtheitsmaß, welches wir uns in diesem Abschnitt genauer ansehen werden.

Das Bestimmtheitsmaß gibt an, wie nah beieinander die von der linearen Regression vorhergesagten Werte und die tatsächlich gemessenen Werte liegen. Um das Bestimmtheitsmaß eines linearen Regressions-Modells berechnen zu können, bestimmen wir das Verhältnis zwischen der Varianz



der von der Regression vorhergesagten Werte zu der Varianz der Daten selbst.

Die Berechnung des Bestimmtheitsmaßes teilt sich in drei Schritte auf:

1. Berechnung der Streuung der gemessenen Daten.
2. Berechnung der Streuung der von der Regression vorhergesagten Daten.
3. Berechnung des Verhältnisses zwischen den beiden Streuungen.

Das klingt jetzt alles sehr abstrakt, deshalb schauen wir uns die Vorgehensweise bei der Berechnung anhand des Beispiels aus der vorherigen Aufgabe mit der Schuh- und Körpergröße an.

### **Schritt 1: Berechnung der Streuung der gemessenen Daten**

Zunächst einmal müssen wir den Mittelwert der Schuhgrößen ermitteln, was recht schnell geht:

$$(42 + 44 + 43) / 3 = 43$$

Die Streuung ist dann einfach die Summe über die quadrierten Abweichungen vom Mittelwert:

$$(42 - 43)^2 + (44 - 43)^2 + (43 - 43)^2 = 2$$

### **Schritt 2: Berechnung der Streuung des Regressions-Modells**

Hierfür müssen wir die quadrierten Abstände zwischen den vom Regressions-Modell vorhergesagten Werten und dem Mittelwert

$$y_1 = 0.05 \cdot 170 + 34 = 42.5$$

$$y_2 = 0.05 \cdot 180 + 34 = 43$$

$$y_3 = 0.05 \cdot 190 + 34 = 43.5$$



berechnen. Als erstes benötigen wir dafür die in Abhängigkeit von der Körpergröße prognostizierten Schuhgrößen:

Damit können wir dann die Streuung folgendermaßen berechnen:

$$(42.5 - 43)^2 + (43 - 43)^2 + (43.5 - 43)^2 = 0.5$$

### Schritt 3: Bestimmung des Verhältnisses der Streuungen

Das Verhältnis zwischen den beiden gerade berechneten Streuungen sieht allgemein mathematisch ausgedrückt folgendermaßen aus:

$$\text{Bestimmtheitsmaß} = \frac{\text{Regressionsstreuung}}{\text{Gesamtstreuung}}$$

Im Fall unseres Beispiels mit den Schuhgrößen wäre das Bestimmtheitsmaß dann:

$$B = \frac{0.5}{2} = 0.25$$

Was fangen wir jetzt mit diesem Wert an? Zunächst einmal: Das Bestimmtheitsmaß nimmt immer einen Wert zwischen 0 und 1 an. Je näher der Wert an der 1 dran ist, umso besser passt das lineare Regressions-Modell zu den gemessenen Daten. Je näher der Wert an der 0 dran ist, umso schlechter passt das Modell. In unserem Fall ist das Bestimmtheitsmaß 0.25, was näher an der 0 als an der 1 dran ist, insofern passt das Modell nicht besonders gut zu den gemessenen Daten. Im Streudiagramm in der vorherigen Aufgabe konnten wir schon sehen, dass die Regressions-Gerade und die Datenpunkte nicht wirklich gut zusammenpassen, da zwischen den vorhergesagten und den tatsächlichen Werten eine große Abweichung bestand.



## 2.2.3. Programmierung einer linearen Regression mit dem **scipy-Modul**

In diesem letzten Abschnitt wollen wir uns anschauen, wie wir die lineare Regression etwas einfacher mit weniger Code implementieren können. Hierfür werden wir die `scipy`-Bibliothek benötigen. Aus der `scipy`-Bibliothek importieren wir das Modul `stats`, welches eine Funktion zur Ausführung einer linearen Regression enthält.

Die Funktion im **stats**-Modul für die lineare Regression heißt `linregress()` und hat zwei Argumente: das Kriterium und den Prädiktor. Die Funktion gibt verschiedene Werte zurück:

- Slope: Hiermit ist die Steigung der Regressions-Geraden gemeint.
- Intercept: Das ist der y-Achsenabschnitt.
- RValue: Hiermit ist die Korrelation zwischen dem Prädiktor und dem Kriterium gemeint; um genauer zu sein, der sogenannte Pearson-Korrelationskoeffizient.
- PValue: Damit ist der sogenannte P-Wert gemeint, ein bei Hypothesentests in der Statistik angewandter Wert, der die Wahrscheinlichkeit angibt, ob bei der Berechnung der Regressionsgeraden der Wert für die Steigung zufällig entstanden ist. Ein niedriger p-Wert ( $p < 0.05$ ) gibt an, dass man die Nullhypothese verwerfen kann. Vereinfacht ausgedrückt bedeutet ein niedriger p-Wert, dass der Prädiktor einen bedeutenden Beitrag zum Regressions-Modell leistet. Auf der anderen Seite bedeutet ein großer p-Wert, dass Änderungen im Prädiktor keinen Einfluss auf das Ergebnis der Regression haben und somit nicht zur Verbesserung des Modells beitragen.
- StdErr: Das ist der Standardfehler, der bei der Berechnung der Steigung der Regressions- Geraden auftritt. Dieser gibt an, wie



stark die beobachteten Datenwerte von den von der Regressions-Geraden vorhergesagten Werten abweichen.

Um für unsere Daten oben eine lineare Regression ausführen zu können, benötigen wir nur die ersten drei Werte, die die linregress()-Funktion zurückgibt. Mit Hilfe des R-Wertes können wir abschätzen, ob tatsächlich ein lineares Verhältnis zwischen dem Prädiktor und dem Kriterium bestehen könnte. Der Code zur Anwendung eines linearen Regressions-Modells sieht folgendermaßen aus:

```
import matplotlib.pyplot as plt
from scipy import stats

x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]

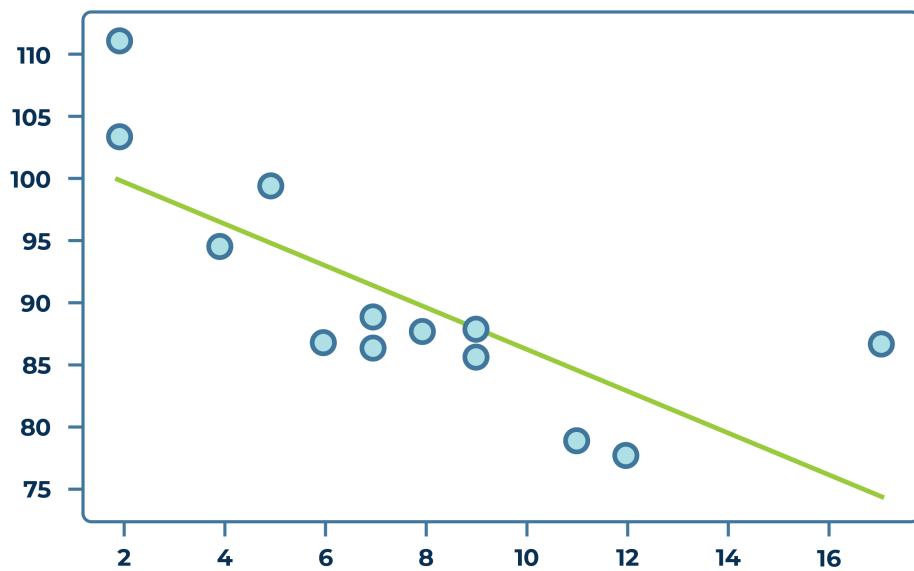
slope, intercept, r, p, std_err = stats.linregress(x, y)

def myfunc(x):
    return slope * x + intercept

mymodel = list(map(myfunc, x))

plt.scatter(x, y)
plt.plot(x, mymodel)
plt.show()

print(r)
```



Der Korrelationskoeffizient nimmt in diesem Fall den Wert -0.758 an, was uns sagt, dass ein recht starkes lineares Verhältnis zwischen dem Prädiktor und dem Kriterium besteht. Wenn der Wert der unabhängigen Variable steigt, sinkt der Wert der abhängigen Variable. Schauen wir uns mal ein

```
import matplotlib.pyplot as plt
from scipy import stats

x = [89, 43, 36, 36, 95, 10, 66, 34, 38, 20, 26, 29, 48, 64, 6, 5, 36, 66, 72, 40]
y = [21, 46, 3, 35, 67, 95, 53, 72, 58, 10, 26, 34, 90, 33, 38, 20, 56, 2, 47, 15]

slope, intercept, r, p, std_err = stats.linregress(x, y)

def myfunc(x):
    return slope * x + intercept

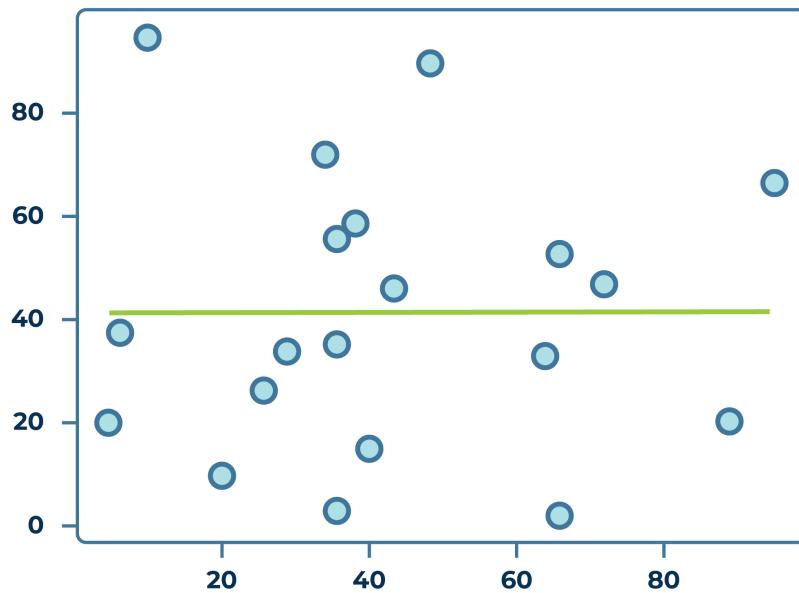
mymodel = list(map(myfunc, x))

plt.scatter(x, y)
plt.plot(x, mymodel,color="green")
plt.show()

print(r)
```



Beispiel an, bei dem die lineare Regression kein guter Weg ist, um Vorhersagen anhand der Daten treffen zu können.



Wir können am Streudiagramm bereits erkennen, dass die Regressions-Gerade keine gute Beschreibung für die Daten liefert, da diese viel zu breit verteilt sind. Der Korrelationskoeffizient gibt uns dafür noch einmal Gewissheit, denn der liegt bei 0.013.

## 2.3. Multiple Regression

### 2.3.1. Was ist multiple Regression?

In diesem Abschnitt wird es um die Erweiterung der linearen Regression auf mehrere Prädiktoren gehen: der sogenannten multiplen Regression. Wir werden die Grundlagen dieser verstehen und anschließend mit Hilfe der scikit-learn-Bibliothek von Python eine multiple Regression ausführen. Schließlich lernen wir noch, wie wir kategoriale Daten in eine multiple Regression miteinbeziehen können. Dafür lernen wir die Methode des One-



Hot-Encoding kennen, die sich simpel mit Hilfe der pandas-Bibliothek implementieren lässt. Anschließend führen wir eine multiple Regression mit numerischen und kategorischen Daten durch.

Wie wir es vorher schon in der Einführung zur linearen Regression gesehen haben, ist die multiple Form eine Art der linearen Regression. Anstatt einer unabhängigen Variable nutzt man bei der multiplen Regression mehrere Variablen als Prädiktoren für das Verhalten des Kriteriums. Man versucht also, eine lineare Funktion zu finden, die von mehreren unabhängigen Variablen abhängt und das Verhalten der abhängigen Variable beschreiben kann.

Mathematisch ausgedrückt sähe das folgendermaßen aus:

$$y = b_0 + b_1x_1 + b_2x_2 + \dots + b_kx_k$$

Hierbei sind die Variablen  $x_1, x_2, \dots, x_k$  die unabhängigen Variablen und y das Kriterium und  $b_0, b_1, \dots, b_k$  die Koeffizienten

Wichtig ist hierbei, dass zwischen den Prädiktoren keine sogenannte Multikolinearität bestehen sollte. Das bedeutet, dass eine Korrelation zwischen den Prädiktoren selbst besteht, die nicht vernachlässigbar klein ist. Dadurch kann man die Prädiktoren auch kaum noch „unabhängige Variablen“ nennen, da sie ja eben voneinander abhängig sind. Dadurch, dass der korrelierende Prädiktor nicht wirklich viel mehr Information zur Regressions-Analyse beiträgt, kann es passieren, dass die Variable sogar die Regressions-Analyse sabotieren kann.

Die Aufgabe eines multiplen Regressions-Algorithmus ist es, die Koeffizienten der obigen Funktion so zu bestimmen, dass die Funktion die Daten am besten beschreibt. Das passiert mit Hilfe der Methode der kleinsten Quadrate, eine Vorschrift zur Bestimmung der optimalen



Koeffizienten, die von Carl Friedrich Gauß entwickelt wurde. Hierbei möchte man die quadrierten Abstände zwischen den gemessenen Datenpunkten und der Regressions-Geraden minimieren. Das bedeutet, dass der Algorithmus verschiedene Werte für die Koeffizienten systematisch ausprobiert und sich den Werten für die Koeffizienten annähert, die den kleinsten quadratischen Abstand zur Folge haben.

Da wir es bei der multiplen Regression mit mehreren unabhängigen Variablen zu tun haben, erhalten wir auch keine Geraden-Gleichung vom Regressions-Algorithmus, sondern eine Gleichung für eine sogenannte Hyperebene. Wie kann man sich eine Hyperebene nun vorstellen?

Nehmen wir dafür mal an, wir hätten Daten, die von zwei Prädiktoren abhängig sind. Wenn wir die Datenpunkte der abhängigen und der beiden unabhängigen Variablen in einem Streudiagramm visualisieren, sehen wir Punkte in einem 3-dimensionalen Raum. Zwei Dimensionen sind für die beiden unabhängigen Variablen und eine für die abhängige Variable. Anstatt einer Geraden spuckt der Regressions-Algorithmus in diesem Fall eine Gleichung für eine 2-dimensionale Ebene im 3-dimensionalen Datenraum aus. Diese Ebene ist an sich auch schon eine Hyperebene. Bei einer Hyperebene in einem n-dimensionalen Raum handelt es sich um eine Ebene, die genau eine Dimension weniger als der Raum selbst hat, also die Dimension n-1.

Erweitern wir unser erstes Beispiel um einen weiteren Prädiktor, dann werden die Datenpunkte in einem 4-dimensionalen Streudiagramm dargestellt, wobei eine Dimension wieder auf die unabhängige Variable fällt und die übrigen drei auf die unabhängigen Variablen, die zur Beschreibung des Kriteriums dienen. Ein Regressions-Algorithmus würde in diesem Fall die Gleichung für eine 3-dimensionale Hyperebene im 4-dimensionalen Datenraum ausgeben.



Glücklicherweise ist es für die praktische Ausführung eines multiplen Regressions-Algorithmus nicht so wichtig, die Optimierungsmethode für die Koeffizienten im Detail zu kennen. Hierfür gibt es eine nützliche Python-Bibliothek mit dem Namen scikit-learn, abgekürzt beim Programmieren immer mit sklearn.

Die scikit-learn-Bibliothek basiert auf den bekannten Python-Bibliotheken scipy, numpy und matplotlib und ist eine Ansammlung an Methoden, um Machine-Learning-Algorithmen zu bauen und anzuwenden. Mit dieser Bibliothek werden wir nun einen Datensatz mit Hilfe der multiplen Regression analysieren.

### 2.3.2. Programmierung einer multiplen Regression

Betrachten wir mal den Datensatz, der in dem File Car\_data.csv gespeichert ist. Dieser enthält Daten zur Automarke, dem Model, dem Volumen des Motors in Kubikzentimetern, dem Gewicht des Autos in Kilogramm und dem CO2-Ausstoß des Autos in Gramm pro gefahrenem Kilometer. Ein Ausschnitt dieser Daten kann in folgendem Bild gesehen werden:

Car	Model	Volume	Weight	CO2
Toyota	Aygo	1000	790	99
Mitsubishi	Space Star	1200	1160	95
Skoda	Citigo	1000	929	95
Fiat	500	900	865	90

Wir können uns denken, dass das Motor-Volumen und das Gewicht des Autos einen erheblichen Einfluss auf den CO2-Ausstoß haben könnten, also wollen wir das mit einer multiplen Regression untersuchen. Volumen und Gewicht sind unsere Prädiktoren und der CO2- Ausstoß ist das Kriterium oder die abhängige Variable.

Um nun eine multiple Regression auf diese Daten laufen lassen zu können, müssen wir folgenden Code schreiben:



```
import pandas
from sklearn import linear_model

df = pandas.read_csv("Car_data.csv")

X = df[['Weight', 'Volume']]
y = df['CO2']

regr = linear_model.LinearRegression()
regr.fit(X, y)
```

Zunächst einmal müssen wir – wie eigentlich immer zu Beginn eines Data-Science-Projektes – die pandas-Bibliothek importieren. Anschließend importieren wir die scikit-learn-Bibliothek, allerdings nur eine spezielle Methode, nämlich die linear\_model-Klasse.

Im nächsten Schritt importieren wir die Autodaten aus dem CSV-File in ein Dataframe und definieren dann, was die x-Variablen (die Prädiktoren) und die y-Variable (das Kriterium) sein sollen.

Nun können wir ein lineares Regressionsobjekt erzeugen und bezeichnen dieses mit dem Namen regr. Das erzeugte Objekt enthält eine fit()-Funktion, die dann die Hyperebene optimal an die Daten anpasst.

Möchten wir das regr-Modell nun nutzen, um eine Vorhersage zu treffen, dann funktioniert das mit der predict()-Methode:

```
#predict the CO2 emission of a car where the
#weight is 2300kg, and the volume is 1300cm3:
predictedCO2 = regr.predict([[2300, 1300]])

print(predictedCO2)
```

Im obigen Beispiel wollen wir die CO2-Emissionen für ein Auto mit einem Gewicht von 2300 Kilogramm und einem Motorvolumen von 1300



Kubikzentimeter vorhersagen. Das Ergebnis sind dann 107.2087328 Gramm ausgestoßenes CO<sub>2</sub> für jeden gefahrenen Kilometer.

Jetzt haben wir vorher ja erfahren, dass der Regressions-Algorithmus versucht, die Koeffizienten der Hyperebenen-Gleichung so zu bestimmen, dass die quadrierten Abstände zwischen vorhergesagten und gemessenen Werten minimal sind. Jeder Koeffizient gehört zu einer der unabhängigen Variablen und gibt an, wie stark sich die abhängige Variable ändern würde, wenn man den Wert des Koeffizienten variieren würde. Diese Koeffizienten kann man sich natürlich bei der Verwendung der scikit-learn-Bibliothek auch ausgeben lassen, um den Einfluss der einzelnen Prädiktoren separat untersuchen zu können.

Ausgeben lassen kann man sich die Koeffizienten mit der folgenden Code-Zeile:

```
print(regr.coef_)
```

Im Fall unseres obigen Beispiels mit den zwei Prädiktoren erhalten wir auch zwei Koeffizienten- Werte.

```
[0.00755095  0.00780526]
```

Der erste Wert ist der Koeffizient für das Gewicht und der zweite derjenige für das Motorvolumen. Was aber sagen diese Koeffizienten jetzt aus?

Betrachten wir zunächst einmal den ersten Koeffizienten für das Gewicht mit einem Wert von 0.00755095. Dieser bedeutet, dass sich bei einer Erhöhung des Gewichts um 1 Kilogramm der CO<sub>2</sub>-Ausstoß für jeden gefahrenen Kilometer um 0.00755095 Gramm erhöht. Analog gilt für den zweiten Koeffizienten, dass eine Erhöhung des Motorvolumens um einen Kubikzentimeter einen Anstieg der CO<sub>2</sub>-Emissionen von 0.00780526



Gramm für jeden gefahrenen Kilometer zur Folge hat. Nehmen wir mal den Wert an CO2-Emissionen, den wir für ein Auto mit 2300 Kilogramm Gewicht und 1300 Kubikzentimetern Motorvolumen berechnet haben. Dieser hat 107.2087328 Gramm betragen. Jetzt erhöhen wir das Gewicht des Autos um 1000 Kilogramm und lassen das Motorvolumen gleich. Nun spuckt uns der Regressions-Algorithmus einen Wert von 114.75968 Gramm pro gefahrenem Kilometer aus. Das können wir auch leicht mit dem Koeffizienten für das Gewicht berechnen:

$$107.2087328 + (1000 * 0.00755095) = 114.75968$$

Wir sehen, dass also das gleiche Ergebnis herauskommt, was uns zeigt, dass es funktioniert.

### 2.3.3. Multiple Regression mit kategorischen Daten

Bisher haben wir nur numerische Werte in unsere multiple Regression miteinbezogen, was ist allerdings mit den kategorischen Daten wie der Automarke oder dem Automodell? Immerhin können wir uns denken, dass die Automarke selbst auch einen Einfluss auf die CO2-Emissionen eines Fahrzeugs haben könnte. Da die meisten Machine-Learning-Algorithmen allerdings nur mit numerischen Daten funktionieren und trainiert werden können, müssen wir einen Weg finden, die kategorischen Daten in unsere Analyse miteinzubeziehen. Dadurch würden wir unser Machine-Learning-Modell verfeinern können. Wie also können wir bei unserem multiplen Regressions-Algorithmus beispielsweise eine lineare Beziehung zwischen den CO2-Emissionen (numerische Daten) und der Automarke (kategorische Daten) herstellen?

Wir müssen die kategorischen Daten so transformieren, dass sie durch numerische Werte repräsentiert werden. Das funktioniert mit der sogenannten One-Hot-Encoding-Methode.

#### One-Hot-Encoding



Um unsere kategorischen Daten – in unserem Fall die Automarke – in numerische Werte transformieren zu können, führen wir für jede Automarke eine eigene Spalte ein. Anstatt einer Spalte also, in der alle Automarken drinstehen, gibt es nun je eine für Audis, BMWs, Volvos und so weiter. Die Werte in diesen Spalten sind dann numerisch und können entweder 1 oder 0 sein. Wenn beispielsweise in der ersten Zeile die Automarke ein Audi ist, dann wird in der Audi- Spalte eine 1 stehen und in allen übrigen Automarken-Spalten eine 0. Diese Methode der Transformation in numerische Werte wird One-Hot-Encoding genannt.

Diese Methode klingt nach einer Menge Aufwand, glücklicherweise muss man diese Codierung nicht händisch vornehmen. Dafür gibt es eine praktische und nützliche Funktion, die von der pandas-Bibliothek zur Verfügung gestellt wird. Mit der `get_dummies()`-Funktion lassen sich die kategorischen Daten „numerifizieren“.

```
import pandas as pd

cars = pd.read_csv('data.csv')
ohe_cars = pd.get_dummies(cars[['Car']])

print(ohe_cars.to_string())
```

Zuerst importieren wir die pandas-Bibliothek und lesen dann gleich das CSV-File mit den Daten ein. Soweit kommt nichts Unbekanntes vor. Im nächsten Schritt dann wenden wir die `get_dummies()`-Funktion auf die Spalte „Car“ an, in welcher die Automarken festgehalten wurden. Das Ergebnis wird im Dataframe `ohe_cars` gespeichert und anschließend als String ausgegeben. Ein Ausschnitt davon ist in der Abbildung unten zu sehen:



Car_Mini	Car_Mitsubishi	Car_Opel	Car_Skoda	Car_Suzuki	Car_Toyota	Car_VW
0	0	0	0	0	1	0
0	1	0	0	0	0	0
0	0	0	1	0	0	0
0	0	0	0	0	0	0
1	0	0	0	0	0	0
0	0	0	0	0	0	1

Wie können wir jetzt aber diese Codierung in unserem multiplen Regressions-Algorithmus benutzen? Wir müssen dafür die Daten miteinander verketteten, also eine Verbindung zwischen den „numerifizierten“ kategorischen Daten und den numerischen Daten Gewicht und Volumen herstellen. Das funktioniert am besten mit der concat()-Funktion, die auch von der pandas- Bibliothek bereitgestellt wird:

```
import pandas
from sklearn import linear_model

cars = pandas.read_csv("data.csv")
ohe_cars = pandas.get_dummies(cars[['Car']])

X = pandas.concat([cars[['Volume', 'Weight']], ohe_cars], axis=1)
y = cars['CO2']

regr = linear_model.LinearRegression()
regr.fit(X,y)

##predict the CO2 emission of a Volvo where the
##weight is 2300kg, and the volume is 1300cm3:
predictedCO2 = regr.predict([[2300, 1300, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0]])

print(predictedCO2)
```

Die ersten vier Zeilen sind noch vom Beginn dieses Abschnitts übernommen worden, dann werden die Prädiktoren für die Regressions-Analyse festgelegt. Hier kommt die concat()- Funktion zum Einsatz. Wichtig ist hier das Argument axis=1, welches festlegt, entlang welcher Achse die Daten miteinander verkettet werden sollen. Der Fall axis=0 bedeutet, dass die Daten anhand der Zeilen miteinander verkettet werden sollen, der Fall axis=1 bedeutet, dass das anhand der Spalten geschehen soll (was unser Fall



ist, weil wir ja die Numerifizierung über das Hinzufügen von zusätzlichen Spalten geschafft haben).

Haben wir die Prädiktoren über die concat()-Funktion definiert, müssen wir nur noch das Kriterium festlegen und wie zu Beginn dieses Kapitels das lineare Regressions-Modell darauf ansetzen. Wollen wir am Ende noch eine Vorhersage unter Berücksichtigung eines Autotyps treffen, müssen wir die entsprechende Codierung für diesen in unserer regr.predict()-Funktion beachten. Im Fall eines Volvos also sind alle Spalten 0 bis auf die vorletzte, welche den Wert 1 hat. Der obige Code liefert dann als Ergebnis 122.45153299 Gramm CO<sub>2</sub>-Ausstoß für jeden gefahrenen Kilometer.

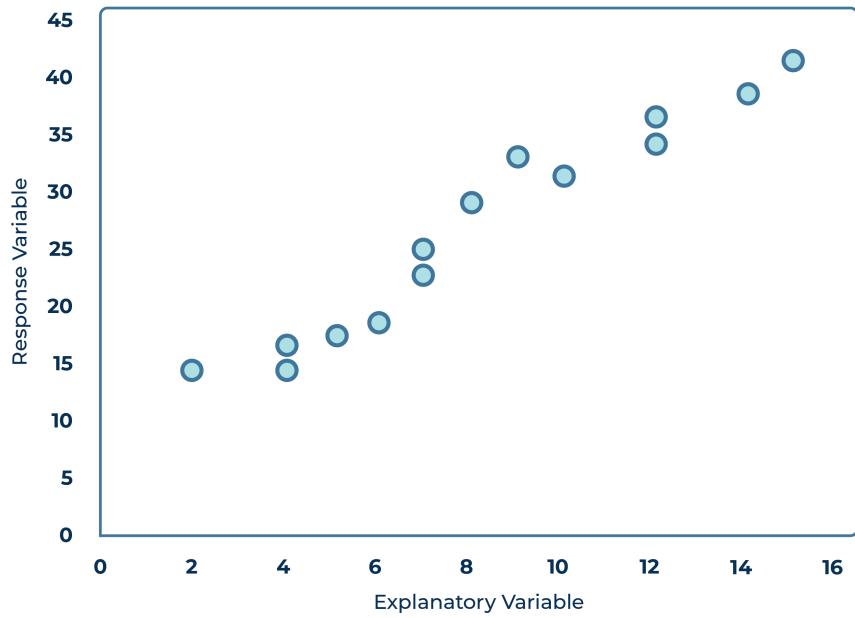
## 2.4. Polynom-Regression

### 2.4.1. Was ist Polynom-Regression?

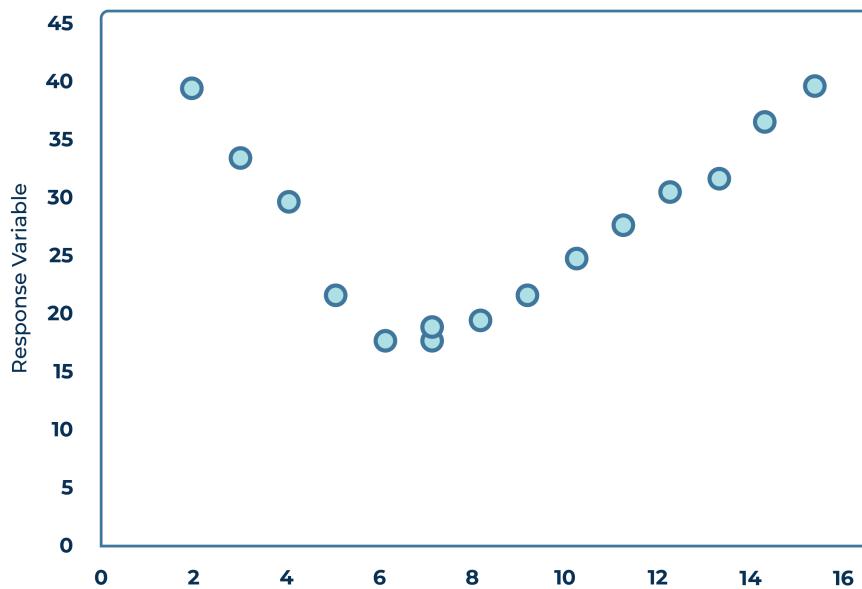
Bisher sind wir bei der einfachen und bei der multiplen linearen Regression davon ausgegangen, dass der Zusammenhang zwischen dem Kriterium und den Prädiktoren linear ist. In der Praxis ist das aber natürlich nicht immer der Fall. Wenn man nun sieht, dass zwischen den Daten kaum ein lineares Verhältnis bestehen kann, kann man eine Beschreibung der Daten mit Polynom-Regression versuchen. Prinzipiell gibt es zwei Arten, auf die man erkennen kann, ob die Beziehung zwischen den Variablen linear ist oder nicht.

#### 1. Erstellen eines Streudiagramms:

Die erste Methode mag trivial erscheinen, tatsächlich ist die Visualisierung der Daten als Data Scientist immer ein wichtiger Schritt, um erste Muster zu erkennen und die Wahl der Analyse- Methode zu optimieren. Genauso ist es auch hier bei der Polynom-Regression. Schauen wir uns folgendes Streudiagramm an:



In diesem Streudiagramm können wir erkennen, dass ein linearer Zusammenhang zwischen den Daten bestehen könnte. Bei diesen Daten würde eine lineare Regression also recht gut funktionieren. Anders jedoch sieht es in diesem Streudiagramm aus:



Bei diesen Daten würde eine lineare Regression gar keinen Sinn mehr machen, da die Beziehung zwischen den Variablen aufgrund der Krümmung nicht mehr linear sein kann. Der Fehler, der durch eine lineare Annäherung der Daten entstehen würde, wäre gewaltig. Hier würde eine Polynom-Regression Sinn machen.

## 2. Berechnung des Bestimmtheits-Maßes:

Wir haben im Abschnitt zur linearen Regression (Kapitel 2.2) bereits das Bestimmtheits-Maß als Verhältnis zwischen der Regressions-Streuung und der Gesamtstreuung kennengelernt, wobei die Gesamtstreuung die quadratische Abweichung der Messwerte vom Mittelwert war.

$$\text{Bestimmtheitsmaß} = \frac{\text{Regressionsstreuung}}{\text{Gesamtstreuung}}$$

Wenn wir auf einen Datensatz ein lineares Regressions-Modell anwenden und für dieses dann das Bestimmtheits-Maß berechnen, können wir feststellen, ob eine Polynom-Regression vielleicht besser wäre. Denn wenn

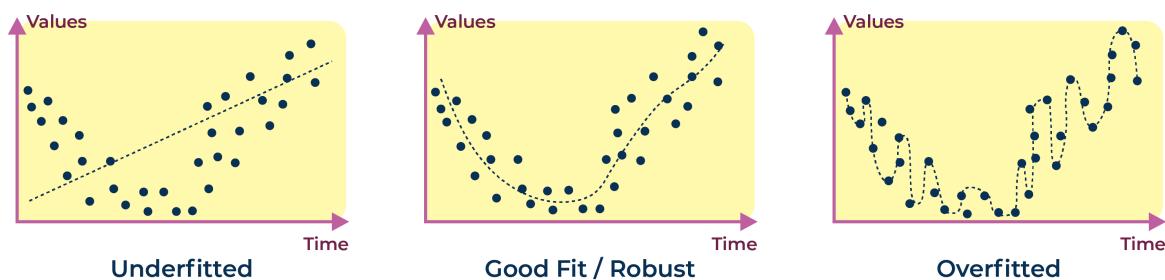


das Bestimmtheits-Maß einen kleinen Wert annimmt, wissen wir, dass das lineare Modell keine gute Beschreibung liefert, was auf einen nicht-linearen Zusammenhang deutet. Dieser kann dann wahrscheinlich besser durch eine Polynom-Regression modelliert werden.

Bei der Polynom-Regression versucht man, die Datenpunkte durch ein Polynom vom Grad  $n$  anzunähern und damit zukünftige Messungen vorhersehen zu können. Mathematisch ausgeschrieben hat das Polynom die folgende Form:

$$Y = b_0 + b_1x + b_2x^2 + \dots + b_nx^n$$

Je höher der Grad des Polynoms ist, umso besser kann sich das Regressions-Modell an die Daten anpassen. Allerdings kann es hierbei auch schnell zu einer Überanpassung kommen, da durch den hohen Grad des Polynoms das Modell zu flexibel wird und sich zu gut an die Daten anpasst.



Versucht man also nicht-lineare Daten durch ein lineares Modell zu beschreiben, haben wir es mit einer Unter-Anpassung (Underfitting) zu tun, wenn der Grad des Polynoms zu hoch ist, mit einer Über-Anpassung (Overfitting). Das Ziel ist also, den perfekten Grad für das Polynom zu finden. In der Praxis verwendet man meistens keine Polynom mit einem Grad größer als 4.



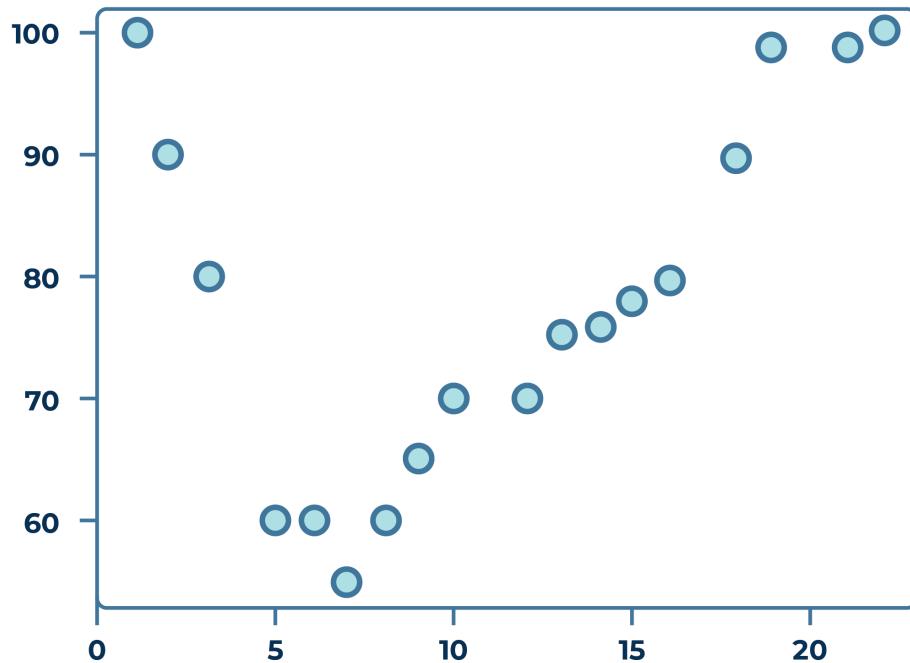
## 2.4.2. Programmierung einer Polynom-Regression

Eine Polynom-Regression in Python zu programmieren, geht dank der Numpy-Bibliothek in vergleichbar wenigen Zeilen Code. Wir nennen unseren Prädiktor x und unser Kriterium y.

```
x = [1, 2, 3, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 16, 18, 19, 21, 22]
y = [100, 90, 80, 60, 60, 55, 60, 65, 70, 70, 75, 76, 78, 79, 90, 99, 99, 100]
```

Visualisieren wir diese Daten zunächst einmal in einem Streudiagramm:

```
import matplotlib.pyplot as plt
plt.scatter(x, y)
plt.show()
```





Wir sehen, dass eine lineare Regression ungeeignet ist, deshalb wollen wir es mit ein paar Polynomen versuchen. Mit numpy erledigt man das quasi in einer Zeile:

```
mymodel = numpy.poly1d(numpy.polyfit(x, y, 3))
```

Die numpy.polyfit()-Funktion hat als Argument den Prädiktor, das Kriterium und den Grad des Polynoms, das wir zur Anpassung an die Daten verwenden wollen. Wir haben bereits erfahren, dass man in der Praxis keine Polynome mit einem Grad größer als 4 verwendet, das heißt es bleiben uns zusammen mit der einfachen linearen Regression vier Arten von Polynom-Regressionen, die wir auf unsere Daten trainieren können. Führen wir das mal für obigen Datensatz aus:

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

x = [1,2,3,5,6,7,8,9,10,12,13,14,15,16,18,19,21,22]
y = [100,90,80,60,60,55,60,65,70,70,75,76,78,79,90,99,99,100]

#Polynom Regression
mymodel1 = np.poly1d(np.polyfit(x,y,1))
mymodel2 = np.poly1d(np.polyfit(x,y,2))
mymodel3 = np.poly1d(np.polyfit(x,y,3))
mymodel4 = np.poly1d(np.polyfit(x,y,4))

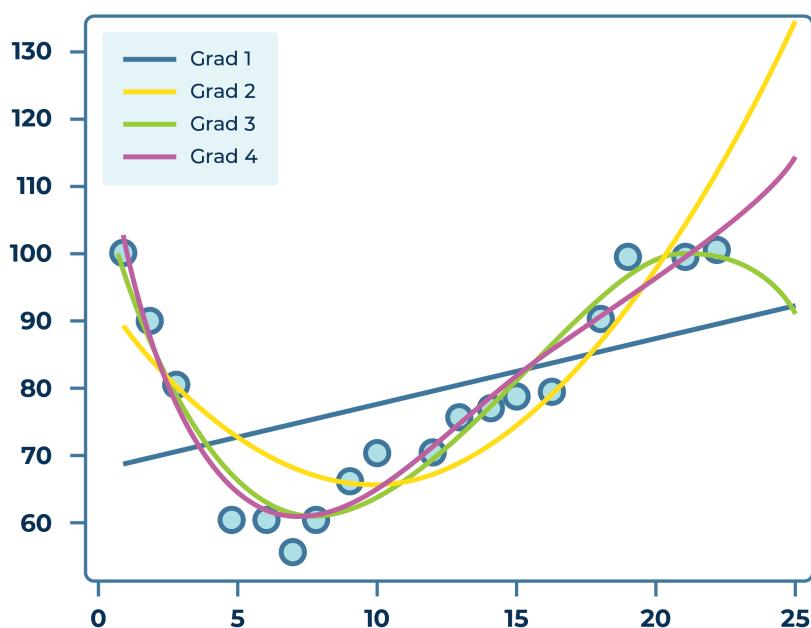
myline = np.linspace(1, 25,100)

plt.scatter(x,y)

plt.plot(myline,mymodel1(myline),label="Grad 1")
plt.plot(myline,mymodel2(myline),label="Grad 2")
plt.plot(myline,mymodel3(myline),label="Grad 3")
plt.plot(myline,mymodel4(myline),label="Grad 4")
plt.legend()
plt.show()
```



Die Polynom-Regression visualisiert sieht dann folgendermaßen aus:

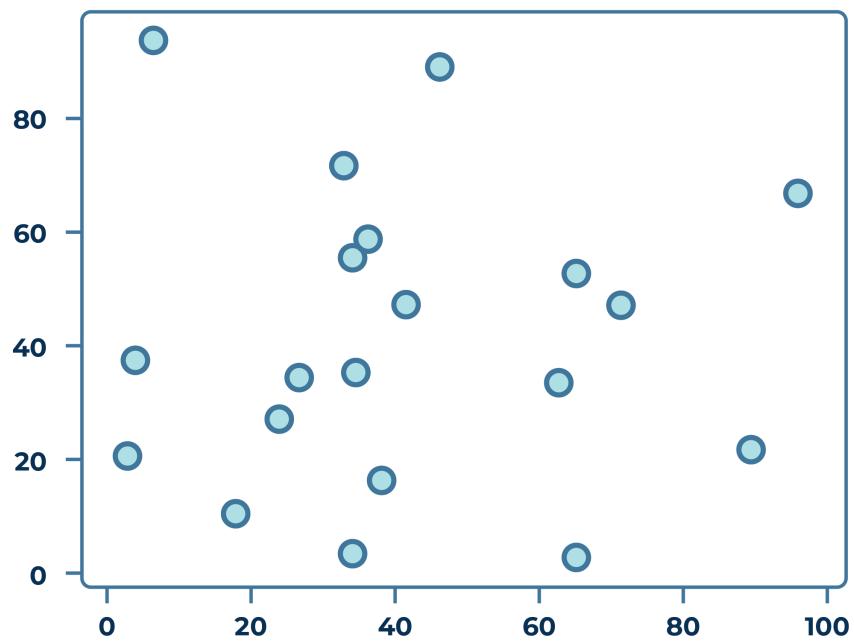


Wir können sehen, dass das lineare Regressions-Modell mit der blauen Geraden weit am Ziel vorbeiflogen und eine nicht zufriedenstellende Modellierung der Daten ist. Die quadratische Funktion passt sich schon besser an die Form an, die kubische Funktion scheint ideal für die

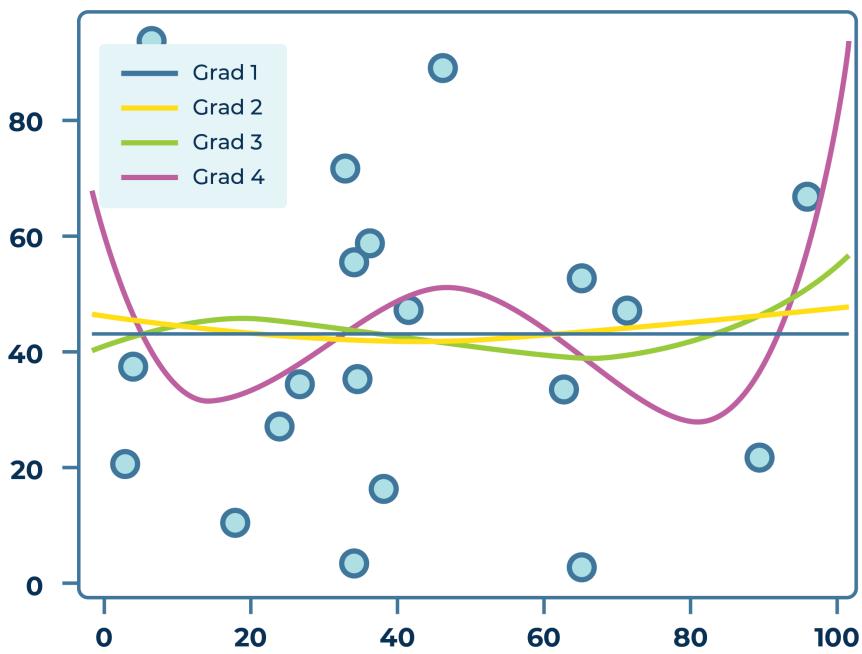
```
x = [89, 43, 36, 36, 95, 10, 66, 34, 38, 20, 26, 29, 48, 64, 6, 5, 36, 66, 72, 40]
y = [21, 46, 3, 35, 67, 95, 53, 72, 58, 10, 26, 34, 90, 33, 38, 20, 56, 2, 47, 15]
```

Beschreibung der Daten zu passen, sogar besser als die quadratische Funktion. Es gibt allerdings auch Datensätze, da macht selbst eine Polynom-Regression keinen Sinn. Nehmen wir beispielsweise folgenden Datensatz:

In einem Streudiagramm visualisiert erkennen wir, dass die Daten so weit in der Ebene verstreut sind, dass selbst eine Polynom-Regression die Verteilung der Daten nicht adäquat beschreiben könnte:



Wenden wir die vier Polynom-Regressionen auf diesen Datensatz an, sehen die entsprechenden Polynome graphisch folgendermaßen aus:



Wenn wir jetzt aber mit einem Datensatz hantieren, bei dem eine Polynom-Regression funktionieren könnte, stellen wir uns doch die Frage, welcher Polynom-Grad die Daten am besten beschreibt und für Vorhersagen geeignet ist. Um das herauszufinden, nutzt man eine im Machine Learning häufig genutzte Methode der sogenannten k-fachen Kreuzvalidierung.

### 2.4.3. K-fache Kreuzvalidierung

Die k-fache Kreuzvalidierung (englisch: k-fold Cross Validation) ist eine im Machine Learning häufig genutzte Methode, um Algorithmen miteinander vergleichen und dann bestimmen zu können, welcher von ihnen am besten zur Lösung eines Problems geeignet ist. In diesem Abschnitt werden wir uns auf die k-fache Kreuzvalidierung mit Bezug auf die Polynom- Regression konzentrieren.



Die Grundlage für die k-fache Kreuzvalidierung bildet der sogenannte mittlere quadratische Fehler, den wir ab jetzt mit MQF abkürzen werden. Der MQF wird mit der folgenden Formel berechnet:

$$MQF = \frac{\sum(y_i - f(x_i))^2}{n}$$

Hierbei ist  $y_i$  der  $i$ -te gemessene Datenwert der abhängigen Variable und  $f(x_i)$  die  $i$ -te Vorhersage des Modells

$n$  ist die Gesamtanzahl der Daten

Wenn wir nun wissen wollen, welcher Polynom-Grad am besten geeignet ist, um die Daten zu beschreiben, können wir ganz einfach unsere Daten in einen Trainings- und einen Test- Datensatz aufteilen. Wie im Abschnitt zur Einführung ins Machine Learning erwähnt, ist eine übliche Aufteilung:

- 80 % Trainingsdaten
- 20 % Testdaten.

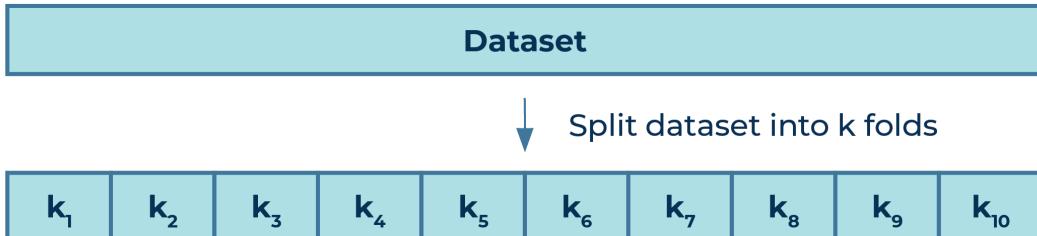
Wir könnten dann einfach jede Polynom-Regression auf die Trainingsdaten trainieren und dann an den Testdaten den MQF bestimmen, um die Effizienz der Algorithmen miteinander vergleichen zu können. Das führt allerdings häufig zu Problemen, da der MQF für die Testdaten stark variieren kann, je nachdem welche 20 % des Datensatzes man nun als Test-Datensatz ausgewählt hat. Wir brauchen also eine Möglichkeit, möglichst viele Variationen an Test- Datensätzen zu bekommen. Und hier kommt die Methode der k-fachen Kreuzvalidierung ins Spiel.

Gehen wir diese mal Schritt für Schritt durch. Vorgehensweise:

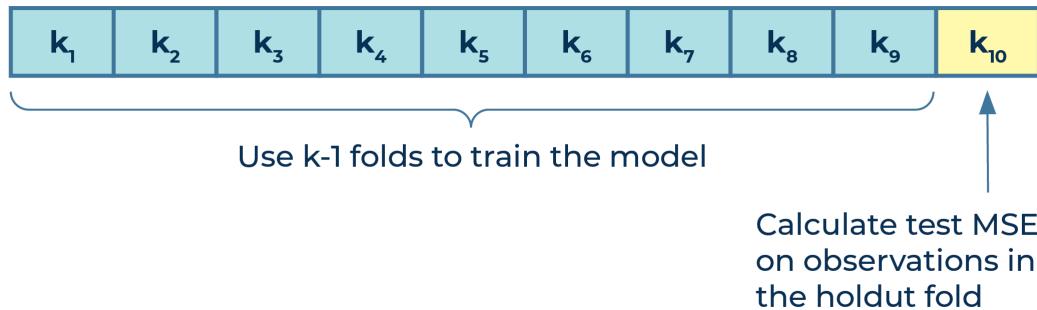
Zunächst einmal müssen wir unseren Datensatz in  $k$  Gruppen oder Faltungen (englisch: Folds) aufteilen. Daher kommt auch das k-fache im



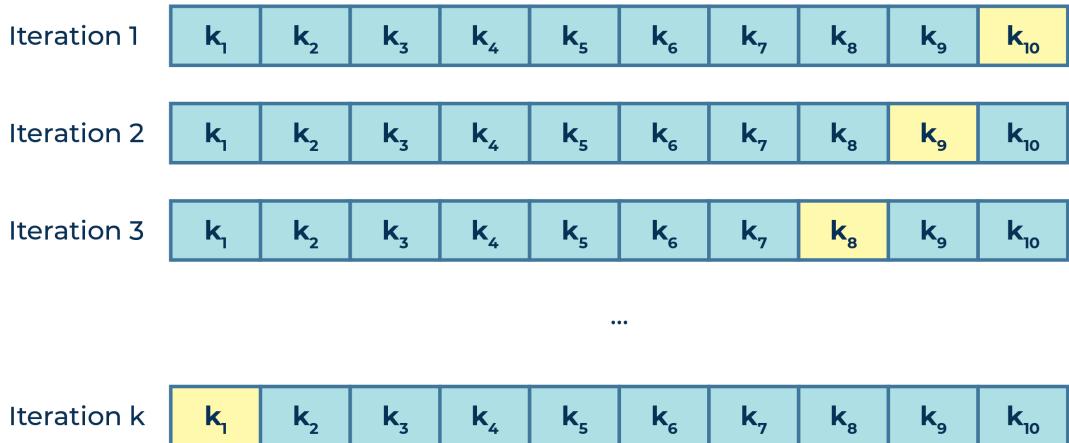
Namen der Methode. Bei der Aufteilung des Datensatzes in  $k$  Gruppen achtet man darauf, dass sie alle ungefähr die gleiche Größe haben.



Jetzt wählt man eine der Gruppen als sogenanntes Hold-Out-Set aus. Dieses wird für den Durchlauf der Test-Datensatz sein. Die übrigen  $k-1$  Gruppen dienen dann als Trainings-Datensatz. Mit diesem trainiert man die Modelle. Nachdem das Training abgeschlossen ist, berechnet man den MQF für das Hold-Out-Set.



Diesen Vorgang wiederholt man jetzt für jede der  $k$  Gruppen einmal, sodass jede Faltung mindestens einmal als Test-Datensatz zur Berechnung des MQF verwendet wird.



Am Ende berechnen wir für jedes Modell den gesamten MQF über die k-Faltungen:

$$MQF_{gesamt} = \sum_{i=0}^k MQF_i$$

Die Frage, die wir uns jetzt natürlich stellen, ist: Wie viele Faltungen soll so ein Datensatz bekommen? Die Frage ist einfach beantwortet. In der Praxis hat es sich als sinnvoll.

herausgestellt, zwischen 5 und 10 Faltungen bei der k-fachen Kreuzvalidierung zu verwenden. Wenn wir nun also herausfinden wollen, welcher Polynom-Grad am ehesten zur Beschreibung der Daten geeignet ist, müssen wir unseren Datensatz in 5 bis 10 Gruppen aufteilen und dann für jedes der allgemeinen Polynome

$$\begin{aligned} Y_1 &= b_0 + b_1 x \\ Y_2 &= b_0 + b_1 x + b_2 x^2 \\ Y_3 &= b_0 + b_1 x + b_2 x^2 + b_3 x^3 \\ Y_4 &= b_0 + b_1 x + b_2 x^2 + b_3 x^3 + b_4 x^4 \end{aligned}$$

- eine k-fache Kreuzvalidierung durchführen, um herauszufinden, welches Modell am besten auf die Daten passt.



## 2.5. Der K-nearest-neighbor-Algorithmus

In diesem Kapitel befassen wir uns mit dem ersten und wahrscheinlich auch einfachsten Klassifizierungsalgorithmus, der sogenannten K-nearest-neighbor-Methode. Wir werden lernen, wie sie funktioniert und welche Abstandsmetriken für die Messung der Abstände zwischen den Datenpunkten verwendet werden. Anschließend lernen wir zwei Algorithmen kennen, die später im scikit-learn-Modul verwendet werden können, um die nächsten Nachbarn zu einem Datenpunkt zu finden. Diese beiden Algorithmen heißen Ball-Tree und KD-Tree. Anschließend setzen wir den K-nearest-neighbor-Algorithmus in Python um, sowohl „from scratch“ als auch eleganter mit der scikit-learn-Bibliothek.

### 2.5.1. Was ist der K-nearest-neighbor-Algorithmus?

Nachdem wir uns in den letzten Abschnitten mit Regressions-Algorithmen zur Vorhersage von numerischen Werten beschäftigt haben, begeben wir uns jetzt in eine neue Kategorie des Machine Learning: die Klassifizierungsalgorithmen.

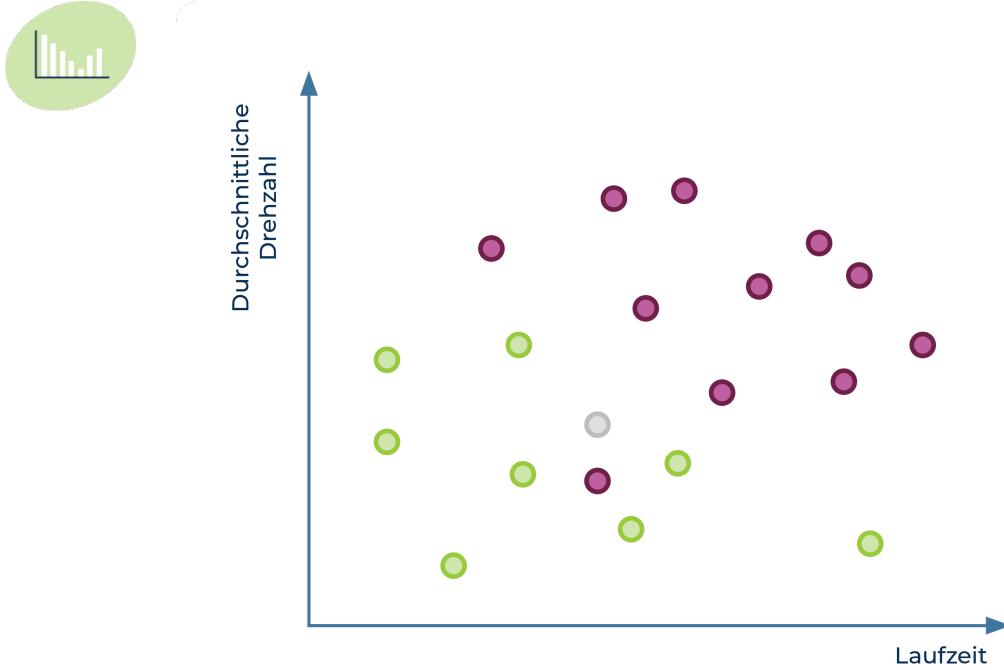
Diese gehören – genau wie die Regressions-Algorithmen – zu den Supervised-Learning- Methoden, brauchen also einen Datensatz mit Klassifizierung als Referenz um Vorhersagen treffen zu können.

Einer der einfachsten und am häufigsten verwendeten Algorithmen zur Klassifizierung von Datensätzen ist der sogenannte K-nearest-neighbor-Algorithmus (abgekürzt KNN). Dieser wurde bereits 1967 vom Stanford-Professor Thomas Cover in einem Paper vorgeschlagen und wird heute gerne zu Beginn eines Data-Science-Projekts genutzt, um sich einen ersten



Überblick über die Daten zu verschaffen. Ein großer Vorteil dieses Machine-Learning-Algorithmus ist – wenn man ihn mit anderen Supervised-Learning-Methoden wie den neuronalen Netzen vergleicht –, dass er kein aufwändiges Training im Vorhinein benötigt. Bei der Klassifizierung nutzt der Algorithmus immer alle zur Verfügung stehenden Daten.

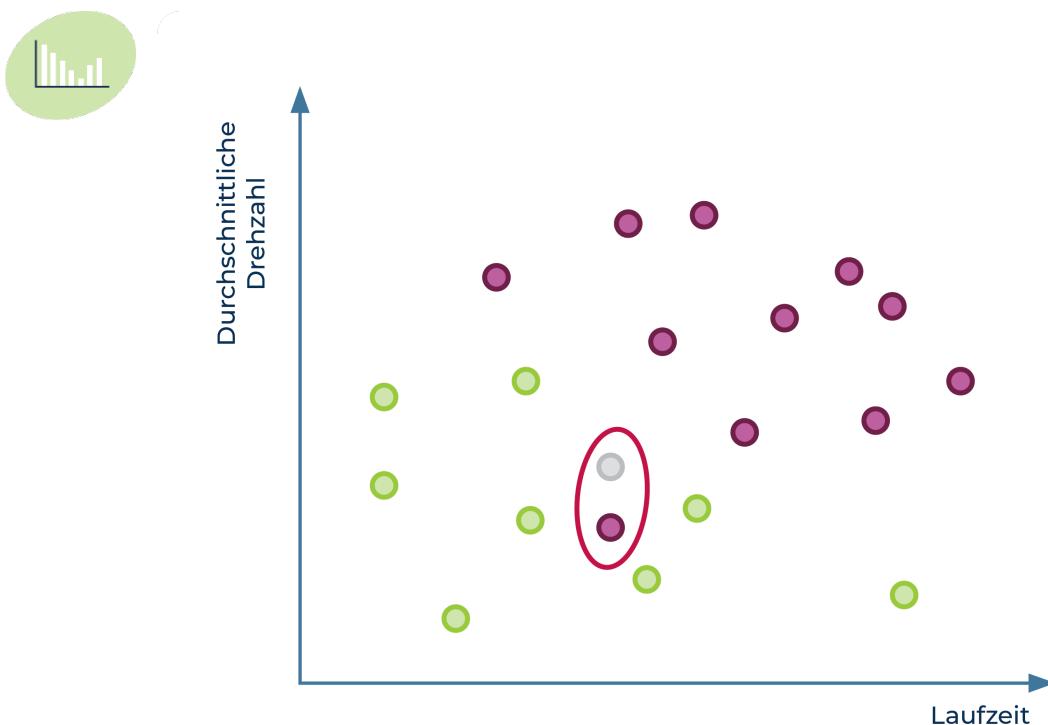
Wie funktioniert denn jetzt der KNN-Algorithmus eigentlich? In einem Satz ausgedrückt: Der KNN-Algorithmus identifiziert die Klasse eines Datenpunktes anhand dessen umgebenden Nachbarn. Um das zu verstehen, schauen wir uns ein Beispiel an. Stellen wir uns dafür zunächst einmal ein 2-dimensionales Koordinatensystem vor, in welchem Datenpunkte in zwei verschiedenen Farben eingetragen sind: rot und grün. Jeder Datenpunkt gehört zu einem Motor, der entweder kaputt gegangen ist oder noch funktioniert. Ist der Datenpunkt rot, dann ist der Motor kaputt gegangen, ist er grün, dann funktioniert der Motor noch. Die x-Achse soll für die Laufzeit des Motors stehen, während die y-Achse für die durchschnittliche Drehzahl des Motors steht. In dieses Koordinatensystem tragen wir jetzt noch einen grauen Punkt ein. Wir wissen dessen durchschnittliche Drehzahl und die Laufzeit, haben aber keine Ahnung über dessen Klassifizierung.



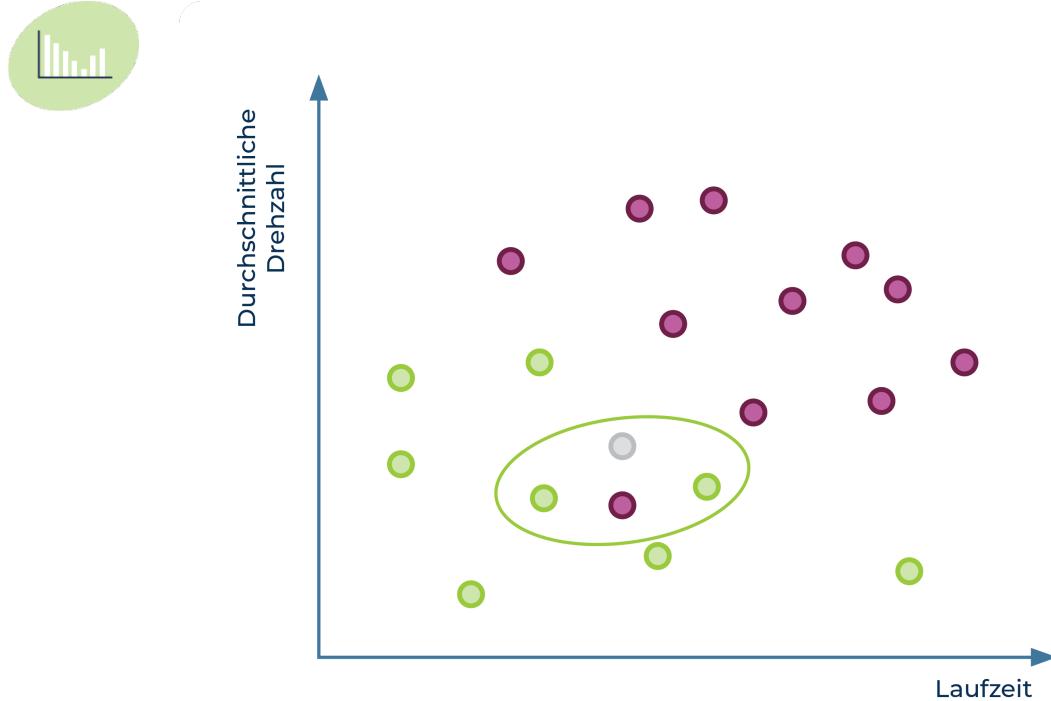


ins Spiel:  $k$  steht für die Anzahl der benachbarten Punkte, die man als Grundlage zur Klassifizierung des grauen Punktes nutzt.

Wählen wir für  $k$  beispielsweise den Wert 1, dann ist nur ein einziger Punkt für die Klassifizierung des grauen Punktes verantwortlich, nämlich derjenige, der dem grauen Punkt am nächsten ist. Im obigen Beispiel wäre der nächste Nachbar zum grauen Punkt ein roter Punkt, wodurch der Algorithmus diesen dann als einen Ausfall klassifizieren wird.



Wählt man dagegen  $k=3$ , sieht die Situation schon wieder ganz anders aus. Hier wären die drei nächsten Nachbarn zwei grüne und ein roter Punkt, wodurch der KNN-Algorithmus als Schlussfolgerung den grauen Punkt als funktionstüchtig klassifizieren wird.



Wir sehen also, dass die Wahl des Wertes für  $k$  den Ausgang des Algorithmus beeinflusst. Woher wissen wir aber, welchen Wert wir für die Anzahl der nächsten Nachbarn wählen müssen?

Die Antwort ist recht vage: Es kommt auf die Daten an.

Im Allgemeinen lässt sich zwar sagen, dass der  $k$ -Wert nicht zu klein gewählt werden sollte, vor allem darf man ihn nicht auf 1 setzen. Das hat den Grund, dass bei niedrig gewähltem  $k$  Ausreißer in den Daten zu stark ins Gewicht fallen, wodurch der KNN-Algorithmus seltener korrekte Klassifizierungen durchführen kann.

Tatsächlich darf der Wert für  $k$  auch nicht zu groß sein. Und was zu groß ist, hängt von der Größe des Datensatzes ab. In unserem Beispiel haben wir einen Datensatz mit insgesamt 20 Datenpunkten. 11 dieser Datenpunkte repräsentieren kaputte Motoren und nur 9 davon stehen für funktionstüchtige Motoren. Würden wir also den  $k$ -Wert auf 20 setzen,



würden wir bei jeder Klassifizierung als Ergebnis einen kaputten Motor erhalten.

Ein weiterer wichtiger Punkt, wenn man einen geeigneten Wert für k auswählen möchte, ist, ob er gerade oder ungerade ist. Möchte man beispielsweise einen Datenpunkt in eine von zwei Kategorien einteilen, kann ein gerader k-Wert dazu führen, dass von jeder Klasse gleich viele Datenpunkte im Kreis der nächsten Nachbarn sind. Das macht eine Klassifizierung unmöglich, da der Datenpunkt nicht eindeutig einer der beiden Gruppen zugeordnet werden kann.

Wir wissen nun, wie der Algorithmus im Prinzip funktioniert, haben allerdings zu Beginn einfach ohne Erklärung gesagt, dass wir den Abstand zwischen den Datenpunkten nutzen. Die Frage ist jetzt: Wie bestimmt man den Abstand zwischen Daten?

## 2.5.2. Metriken zur Abstandsbestimmung

Um den Abstand zwischen den Datenpunkten berechnen zu können, verwendet man sogenannte Metriken. Die Datenpunkte werden dafür in eine Vektor-Form gebracht. Anschließend bestimmt man den Abstand zwischen den Datenpunkten anhand der Differenzen der Komponenten. Beim KNN-Algorithmus werden in der Regel zwei Arten der Metrik angewandt: die Manhattan-Metrik und die euklidische Metrik.

### Die Manhattan-Metrik

In der Manhattan-Metrik wird der Abstand zwischen zwei Datenpunkten als die Summe der Beträge der Differenzen zwischen den einzelnen Komponenten des Vektors definiert. Das war jetzt sehr abstrakt. Schauen wir uns erstmal die ausgeschriebene Formel an:



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

Hierbei sind x und y die Datenpunkte und xi und yi die Komponenten des jeweiligen Datenpunktes. Die Summe geht bis n, wobei n für die Anzahl der

$$d(x, y) = \sum_{i=1}^2 |x_i - y_i| = |3 - 1| + |2 - 0| = |2| + |2| = 4$$

Parameter zur Klassifizierung steht. Es werden also die Abstände zwischen n-dimensionalen Vektoren bestimmt. Um das Ganze zu veranschaulichen, schauen wir uns ein kleines Beispiel an. Berechnen wir den Abstand in der Manhattan-Metrik zwischen dem Punkt x=(3,2) und y=(1,0):

Wir berechnen den Abstand zwischen zwei 2-dimensionalen Vektoren, weswegen die Summe nur bis 2 geht. Wir berechnen die Differenzen der beiden ersten Komponenten und nehmen den Betrag davon, dann addieren wir das zum Betrag der Differenz der beiden zweiten Komponenten.

### Die euklidische Metrik

Eine häufiger verwendete Methode zur Abstandsbestimmung ist die sogenannte euklidische Metrik. Ihre mathematische Formel sieht etwas unheimlicher aus:

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

Hierbei sind y und x wieder die n-dimensionalen Vektoren. In der Wurzel haben wir wieder eine Summe, die bis n geht. Diesmal allerdings

$$d(x, y) = \sqrt{(y_1 - x_1)^2 + (y_2 - x_2)^2}$$



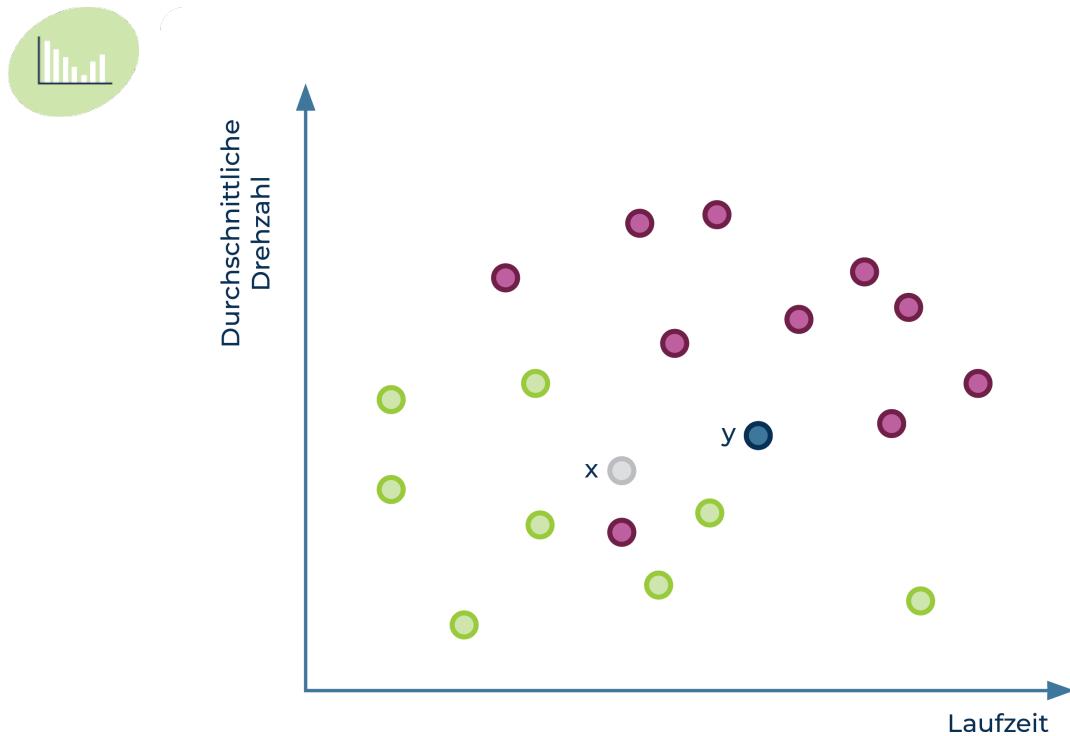
summieren wir nicht die Beträge der Differenzen, sondern die quadrierten Differenzen der einzelnen Komponenten. Im KNN-Algorithmus wird diese Formel (oder die Manhattan-Metrik-Formel) dann auf alle Datenpunkte angewendet. Es muss also der euklidische Abstand zwischen dem grauen und jedem der 20 anderen Datenpunkte berechnet werden. Da wir im Fall unseres Beispiels oben mit den Motoren nur zwei Parameter haben, beschränkt sich die Summe unter der Wurzel auf zwei Summanden: Schauen wir uns mal anhand eines Beispiels an, wie wir konkret den euklidischen Abstand zwischen zwei Motoren berechnen können. Nehmen wir an, die Laufzeit des grauen Motors (repräsentiert durch den Vektor  $x$ ) beträgt 5000 Stunden und die durchschnittliche Drehzahl in dieser Zeit waren 1660 Umdrehungen pro Minute.

Wir wollen nun den euklidischen Abstand zu einem Motor mit 6300 Stunden Laufzeit und einer durchschnittlichen Drehzahl von 1820 Umdrehungen pro Minute berechnen (repräsentiert durch den Vektor  $y$ ). Die beiden Vektoren als Tupel aufgeschrieben sind:

$$x = (5000, 1660)$$

$$y = (6300, 1820)$$

In folgendem Streudiagramm steht der graue Punkt für den Motor  $x$  und der blaue Punkt für den Motor  $y$ .



Wenn wir nun die Werte in die Formel für den euklidischen Abstand einsetzen, erhalten wir als Ergebnis:

$$d(x, y) = \sqrt{(6300 - 5000)^2 + (1820 - 1660)^2} = 1309,8$$

### 2.5.3. Algorithmen zur Bestimmung der nächsten Nachbarn

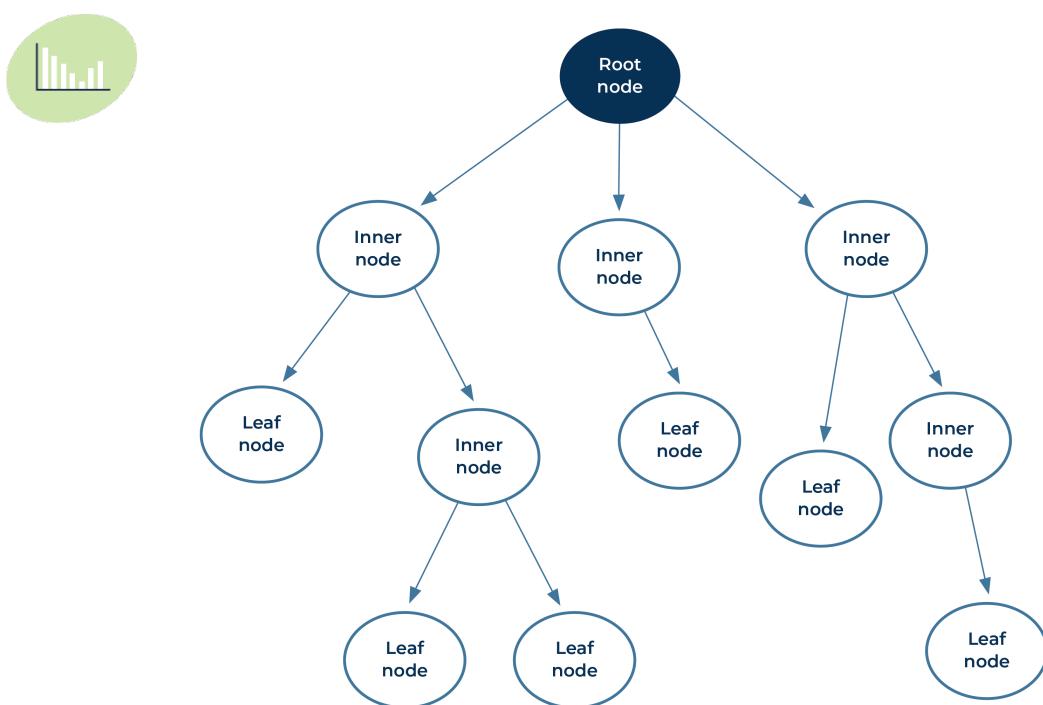
Jetzt wissen wir zwar, wie wir die Abstände zwischen den einzelnen Datenpunkten berechnen können, aber wie finden wir die k-nächsten Nachbarn eines Datenpunktes? Der einfachste Weg ist, die Liste der Differenzen aufsteigend zu sortieren und dann die Klassen der Punkte zu notieren, die zu den ersten k Differenzen gehören. Dafür müsste man aber erst die Liste aller Differenzen sortieren, was bei wachsender Datenmenge sehr aufwendig werden kann. Deshalb hat man sich andere Algorithmen ausgedacht, die auf sogenannten Baumstrukturen basieren. In diesem Abschnitt werden wir uns anschauen, was es mit Bäumen zur



Strukturierung von Daten auf sich hat und wie sie auf die beiden Algorithmen Ball-Tree und KD-Tree angewendet werden.

## Bäume

In der Data Science oder einfach in der Informatik allgemein versteht man unter einem Baum eine Struktur, mit welcher Daten hierarchisch organisiert werden. Dabei orientiert man sich beim Aufbau eines Baums an seinem natürlichen Vorbild. Jeder Baum fängt mit einer sogenannten Root Node an. Auf diese folgen dann die Verästelungen zu den inneren Nodes. Diese inneren Nodes unterscheidet man dann in Parent und Children Nodes. Als Parent Node bezeichnet man jene Knoten im Baum, die über einem anderen Node positioniert sind. Children Nodes auf der anderen Seite sind unter einem anderen Node positioniert. Zu guter Letzt muss man noch die Leaf Nodes kennen, das Blatt am Ende einer Verästelung. Da hört der Baum dann auf.





Dieses Wissen über Baum-Strukturen sollte nun ausreichen, um die Funktionsweise des Ball-Tree- und des KD-Tree-Algorithmus zu verstehen.

## Der Ball-Tree-Algorithmus

Der erste Algorithmus, den wir uns anschauen werden, ist der sogenannte Ball-Tree-Algorithmus. Dieser teilt die Datenpunkte immer wieder in 2 Cluster auf, gehört also zur Kategorie der sogenannten binären Bäume (weil jeder Node maximal zwei Kinder hat). Wenn man sich die Datenpunkte in einem Koordinatensystem vorstellt, dann ist jeder der Cluster von einem Kreis (im 2-dimensionalen Fall), einer Sphäre (im 3-dimensionalen Fall) oder einer Hyper-Sphäre (im n-dimensionalen Fall) umgeben. Die Hyper-Sphäre ist hierbei analog zur Hyper-Ebene zu verstehen, die wir im Kapitel 2.3 über die multiple Regression kennengelernt haben. Durch diese Hypersphäre, die wie ein Ball die Datenpunkte umschließt, hat der Ball-Tree-Algorithmus seinen Namen bekommen. Aber wie funktioniert dieser jetzt eigentlich genau? Gehen wir den Algorithmus mal Schritt für Schritt durch.

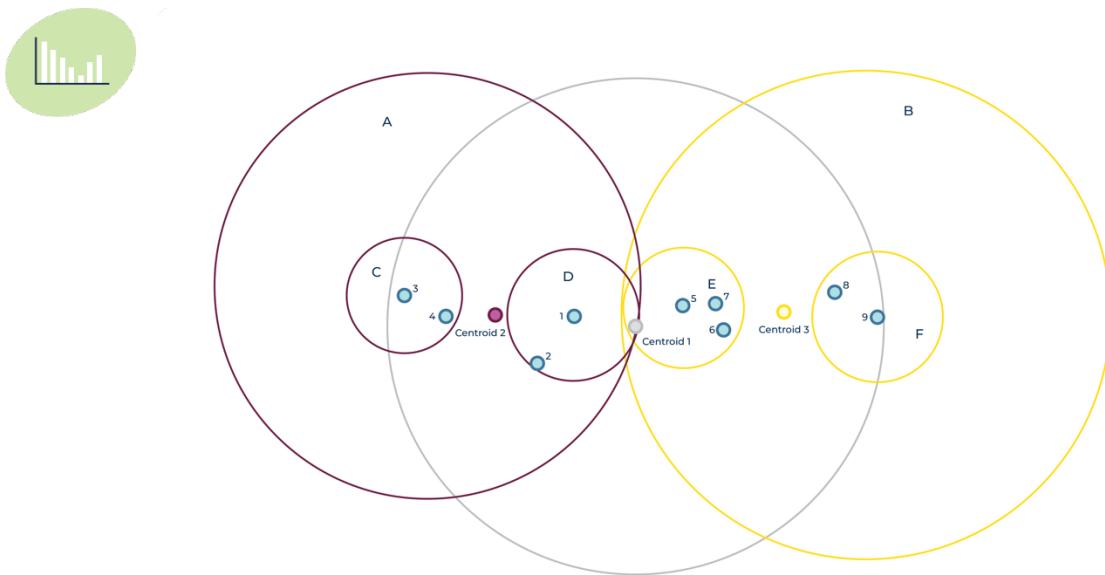
Zunächst einmal legt man eine Hypersphäre um alle Datenpunkte (ab jetzt wird der Begriff Hypersphäre gleichbedeutend mit Kreis und Sphäre verwendet). Vom zentralen Datenpunkt dieser Hypersphäre aus gemessen wählt man den am weitesten entfernten Datenpunkt aus. Diesen macht man zum Zentrum des ersten Clusters.

Von diesem neuen Zentrum aus wählt man nun den Datenpunkt, der am weitesten entfernt liegt. Dieser Punkt wird dann zum Zentrum des zweiten Clusters.

Alle übrigen Punkte werden dann einem der beiden Cluster zugeordnet, je nachdem zu wessen Zentrum sie näher gelegen sind. Hierbei ist wichtig, dass der Algorithmus jeden Punkt nur einem einzigen Cluster zuordnet.

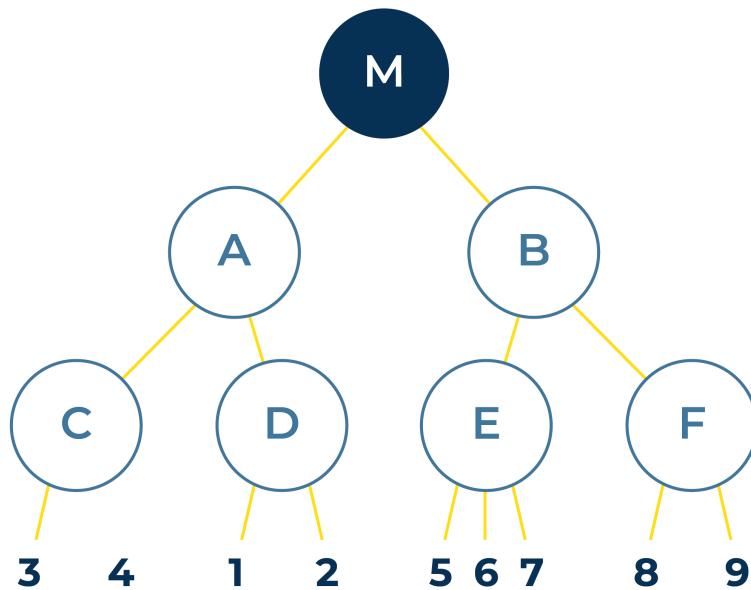


Diese Prozedur der Aufteilung wird dann für jeden der beiden Cluster separat wiederholt und wird so oft fortgeführt, wie es zu Beginn des Algorithmus definiert wurde. Der Vorgang ist in der folgenden Graphik mit Kreisen visualisiert:



Der graue Punkt in der Mitte ist das Zentrum des grauen Kreises, der alle Datenpunkte enthält. Die am weitesten von Centroid 1 entfernten Punkte sind 3 und 9. Diese werden zu den Zentren der neuen Cluster A und B. Diese überschneiden sich zwar, aber Centroid 1 ist eindeutig dem Cluster A zugeordnet. Innerhalb der Cluster A und B wird dann der Vorgang nochmal wiederholt, wobei die Cluster C und D entstehen. Analog dazu wird Cluster B in die Sub-Cluster E und F aufgeteilt. Dann endet der Algorithmus nach diesem Schritt.

Als Baum sieht die Aufteilung der Datenpunkte folgendermaßen aus:



Der Root Node M steht für den grauen Kreis, der zu Beginn alle Datenpunkte enthalten hat. Von diesem geht die Aufteilung in die Cluster A und B und anschließend die weitere Aufteilung in C, D, E und F. Die Punkte, die am Ende die gleiche Parent Node haben, sind Nachbarn im n- dimensionalen Raum. Wenn man dann über den nächsten Parent Node zu den Cousins und Cousinen kommt, ist man bei der nächsten Stufe an nahen Nachbarn angekommen. So hat der Ball-Tree-Algorithmus am Ende die Datenpunkte so strukturiert, dass er zu einem beliebigen Datenpunkt schnell die nächsten Nachbarn finden kann.

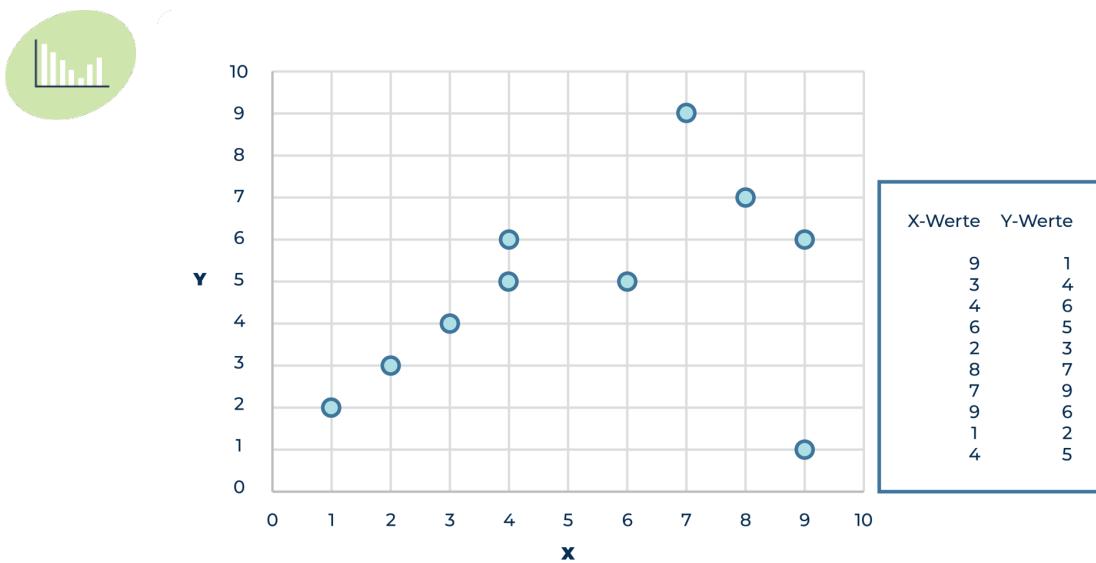
### Der KD-Tree-Algorithmus

Dieser nächste Algorithmus ist vom Prinzip her ähnlich wie der Ball-Tree-Algorithmus aufgebaut. Auch bei ihm hat man am Ende die Daten in einer binären Baumstruktur organisiert. An jedem Node werden die Datenpunkte in zwei Cluster aufgeteilt. Allerdings wird bei der Aufteilung in diesem Fall nicht der euklidische (oder Manhattan-) Abstand als Split-Kriterium



verwendet, sondern der Median der jeweiligen Parameter. Schauen wir uns am besten anhand eines Beispiels an, wie das funktioniert.

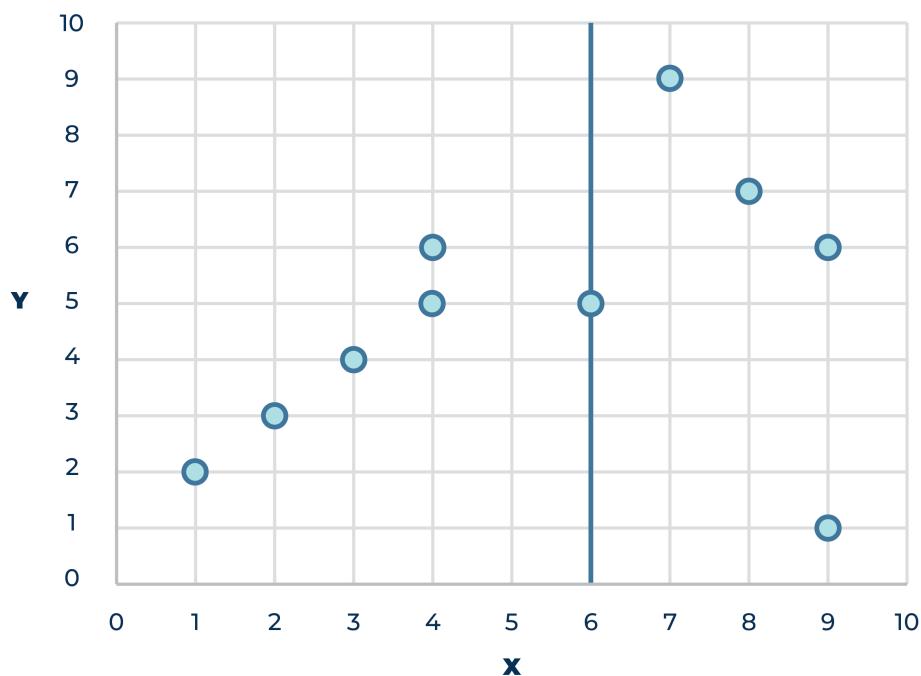
Nehmen wir an, wir hätten folgendes Streudiagramm, auf dessen rechter Seite die x- und y-Koordinaten der Punkte festgehalten sind. Wir haben es also mit einem 2-dimensionalen Vektorraum zu tun, also nur 2 Parameter zur Klassifizierung eines Datenpunktes.



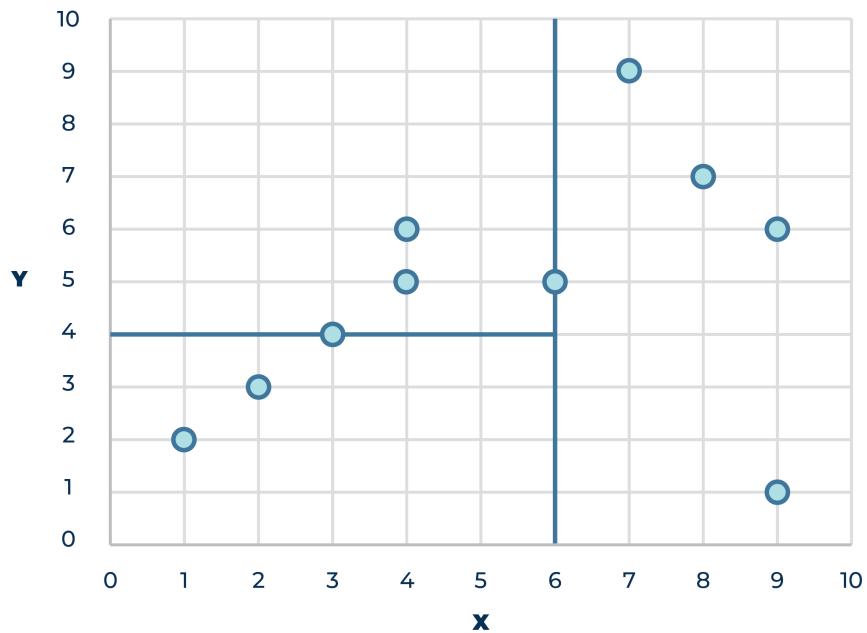
Wie funktioniert also jetzt der KD-Tree-Algorithmus? Er fängt mit dem Median der ersten Achse (der x-Achse) an und sortiert die 2-dimensionalen Datenpunkte entsprechend in die beiden Cluster „kleiner als der Median“ und „größer als der Median“. Die x-Werte in obigem Beispiel sortiert sind 1,2,3,4,4,6,7,8,9,9, der Median ist folglich 6. Die Datenpunkte werden also so aufgeteilt:

- Kleiner als der Median: (1,2) (2,3) (3,4) (4,5) (4,6)
- Größer als der Median: (6,5) (7,9) (8,7) (9,6) (9,1)

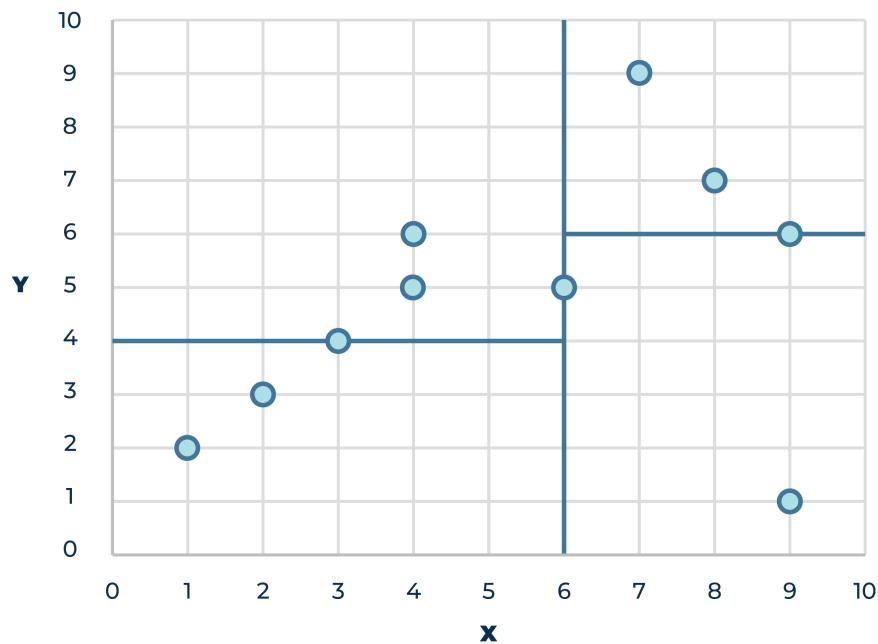
Dadurch haben wir die Daten in zwei Cluster aufgeteilt. Graphisch wird das durch eine senkrechte Linie erreicht, die durch x=6 hindurchgeht.



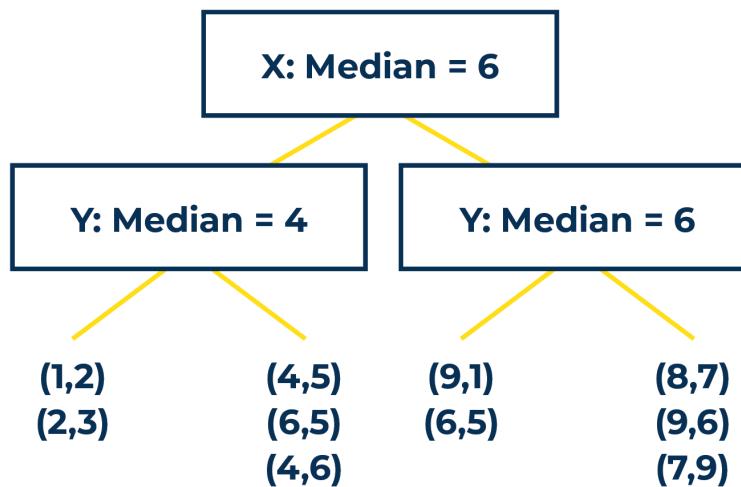
Im nächsten Schritt schauen wir uns die y-Achse an. Da wir jetzt schon zwei Cluster haben, müssen wir die Datenpunkte in jedem Cluster nun separat betrachten. Im linken Cluster haben wir die sortierten y-Werte 2,3,4,5,6 mit dem Median 4. Das heißt, der Cluster wird bei  $y=4$  in zwei Sub-Cluster aufgeteilt. Die Datenpunkte im oberen Sub-Cluster sind (4,6) (3,4) (4,5), die im unteren Sub-Cluster (2,3) (1,2). Graphisch ist das eine horizontale Linie, die bei der vertikalen Linie stoppt:



Jetzt müssen wir nur noch das Gleiche für die rechte Seite machen. Die sortierten y-Werte im rechten Cluster sind 6,7,8,9,9 mit dem Median 8. Das führt zu zwei Subclustern, die bei  $y=8$  getrennt sind. Im oberen Sub-Cluster sind die Punkte (8,7) (7,9) (9,6) und im unteren die Punkte (9,1) und (6,5). Die endgültige Aufteilung der Daten durch den Algorithmus sieht dann so aus:



Dieser Aufteilungsprozess sieht als Baumstruktur folgendermaßen aus:





Die Anzahl der Schritte, die der Algorithmus durchläuft, ist also anders als beim Ball-Tree-Algorithmus nicht zu Beginn durch die Nutzenden definiert, sondern durch die Anzahl der Parameter, die zur Klassifizierung verwendet werden.

## Vergleich der Algorithmen

Ein weiterer Algorithmus, den wir noch nicht erwähnt haben, ist die sogenannte Brute-Force- Methode. Diese ist recht simpel: Man probiert einfach alle Datenpunkte durch. Das klingt gut, weil man sich dann sicher sein kann, dass der Algorithmus bei der Wahl der nächsten Nachbarn auch tatsächlich die nächsten Nachbarn wählt. Allerdings funktioniert die Brute-Force-Methode nur bei kleinen Datensätzen, da bei größeren der Rechenaufwand nicht mehr tragbar wird.

Der KD-Tree-Algorithmus ist die beste Wahl, wenn man es mit niedrig-dimensionalen Datensätzen zu tun hat, wenn also nicht so viele Parameter zur Klassifizierung verwendet werden. Hat man es dagegen mit einem höher-dimensionalen Datensatz zu tun, wählt man am besten die Ball-Tree-Methode aus.

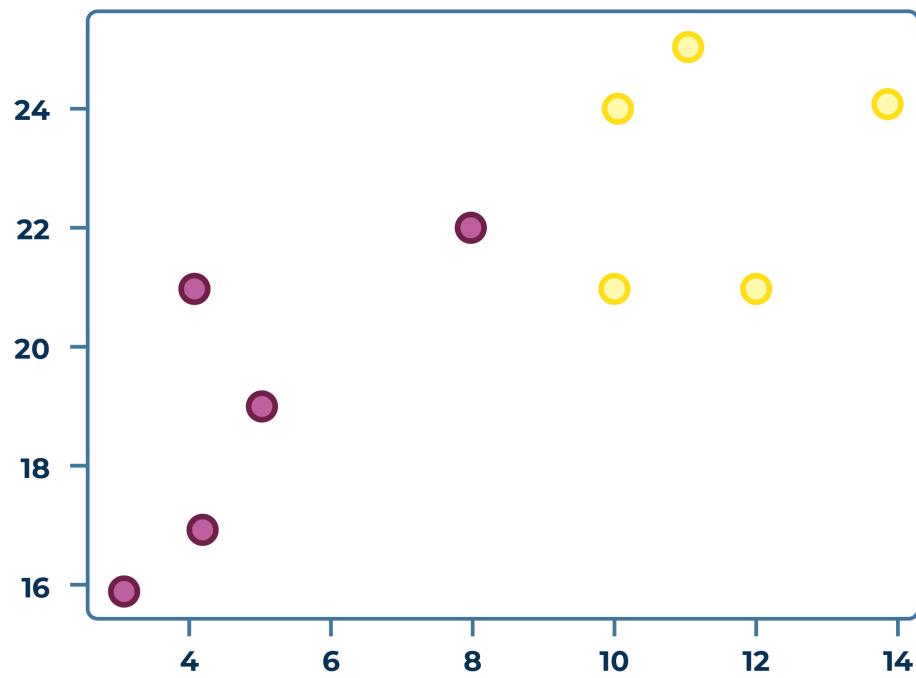
### 2.5.4. Programmierung des KNN mit scikit-learn

Nehmen wir folgende Daten als Beispiel:

```
import matplotlib.pyplot as plt

x = [4, 5, 10, 4, 3, 11, 14, 8, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]
classes = [0, 0, 1, 0, 0, 1, 1, 0, 1, 1]

plt.scatter(x, y, c=classes)
plt.show()
```



In diese Daten wollen wir einen unbekannten Datenpunkt hineinsetzen und mit dem KNN- Algorithmus klassifizieren. Der Programmcode dafür sieht so aus:

```
from sklearn.neighbors import KNeighborsClassifier

data = list(zip(x, y))
knn = KNeighborsClassifier(n_neighbors=5)

knn.fit(data, classes)

new_x = 8
new_y = 21
new_point = [(new_x, new_y)]

prediction = knn.predict(new_point)
print(prediction)

plt.scatter(x + [new_x], y + [new_y], c=classes + [prediction[0]])
plt.text(x=new_x-1.7, y=new_y-0.7, s=f"new point, class: {prediction[0]}")
plt.show()
```



Es fängt mit dem Import der KNeighborsClassifier-Klasse aus der Unterbibliothek neighbors in der sklearn-Bibliothek an. Mit dieser Klasse erzeugt man dann ein Objekt mit dem Namen knn. In unserem Fall hat das Objekt die Eigenschaft n\_neighbors=5, was bedeutet, dass für den KNN-Algorithmus 5 nächste Nachbarn als Basis für die Entscheidung verwendet werden. Weitere wichtige Parameter, die ein Objekt der KNeighborsClassifier haben kann, sind:

- weights{'uniform', 'distance'} or callable, default='uniform':  
Die Gewichte, die die Abstände zwischen den Punkten bekommen sollen.
  - uniform: alle Abstände in jede Richtung haben die gleiche Gewichtung. Dieses ist das default-Argument.
  - distance: die Abstände sind invers zu ihrem Abstand gewichtet, was bedeutet, dass nähere Punkte am zu klassifizierenden Punkt eine höhere Gewichtung bekommen.
  - [callable]: eine von dem/der Programmierer\*in definierte Abstandsfunktion
- algorithm{'auto', 'ball\_tree', 'kd\_tree', 'brute'}, default='auto'  
Welcher Algorithmus soll gewählt werden, um die nächsten Nachbarn zu bestimmen?
  - Ball-Tree-Algorithmus
  - KD-Tree-Algorithmus
  - Brute-Force-Methode
  - default='auto': abhängig von den Daten wählt das Classifier-Objekt selbst den passenden Algorithmus aus
- p: Ein ganzzahliger Parameter (Integer), der festlegt, welche Metrik zur Abstandsbestimmung verwendet wird
  - per default ist p=2, was für die euklidische Metrik steht
  - wählt man p=1, wird die Manhattan-Metrik verwendet



Das erzeugte Objekt hat nicht nur verschiedene Eigenschaften, sondern auch verschiedene Methoden, die wir einsetzen können. Die wichtigsten sind:

- `fit(X,y)`: füttet den Classifier auf das Trainingsdatenset; X sind die Trainingsdaten als Tupel (für mehrere Parameter), y sind die Klassen
- `predict(X)`: sagt eine Klassifizierung für einen Testdatensatz X (Tupel) vorher
- `kneighbors(X, n_neighbors, return_distance=True)`: gibt die k Nachbarn eines Datenpunktes zurück
  - X: Datenpunkt
  - n\_neighbors: Anzahl der Nachbarn
  - return\_distance: hier gibt man einen Booleschen Wert True oder False ein, je nachdem ob man sich die Abstände zu den Punkten ausgeben lassen will

## 2.6. Die logistische Regression

In diesem Abschnitt werden wir die logistische Regression kennenlernen, einen Regressions- Algorithmus, der gerne zur Klassifikation von binären Variablen verwendet wird. Wir werden verstehen, wieso die logistische Regression nicht in die Kategorie der Regressions-Algorithmen gehört und wie ihr Zusammenhang zu Wahrscheinlichkeiten ist. Dann sehen wir uns an, wie der eigentliche Algorithmus genau funktioniert und lernen dabei die Maximum-Likelihood-Methode kennen. Anschließend schauen wir uns noch an, wie eine logistische Regression bei Daten funktioniert, die nicht in binärer Form vorliegen. Mit diesem Wissen programmieren wir dann mit Hilfe der scikit-learn-Bibliothek einen eigenen logistischen Regressions-Algorithmus.



## 2.6.1. Was ist die logistische Regression?

Es mag paradox erscheinen, dass der nächste Machine-Learning-Algorithmus aus dem Bereich der Klassifikationen das Wort „Regression“ enthält, aber hier sind wir. Die logistische Regression ist strenggenommen eine Regression, da sie numerische Werte vorhersagt. Allerdings werden beim Machine-Learning-Algorithmus die numerischen Werte zur Klassifikation benutzt. Da es eine Regression ist, haben wir es auch hier wieder mit Prädiktoren und einem Kriterium zu tun. Im Fall der logistischen Regression ist das Kriterium eine kategoriale Variable. Die logistische Regression wird immer dann zur Klassifikation genutzt, wenn die unabhängige Variable nur ein paar wenige gleichrangige Ausprägungen hat. Was bedeutet das konkret?

Nehmen wir als Beispiel an, wir würden vor zwei Klassifizierungsproblemen stehen. Beim ersten Problem verkaufen wir in unserem Online-Store einen Rucksack, den wir in zwei Farben – rot und grün – anbieten. Wir wollen abhängig von verschiedenen Kundendaten – Geschlecht, Alter und Herkunftsregion – vorhersagen können, welchen Rucksack sie kaufen. Abhängig von ihren Grunddaten wollen wir sie in eine der Kategorien „rot“ oder „grün“ packen.

Beim zweiten Problem stehen wir vor der gleichen Herausforderung, allerdings ist das Produkt diesmal ein anderes. Es geht um eine Tasse, die wir in 27 verschiedenen Ausführungen anbieten. Abhängig von Geschlecht, Alter und Herkunftsregion wollen wir nun vorhersagen, welche Ausführung gekauft wird. Der Kunde bzw. die Kundin kann also in eine von 27 Kategorien fallen.

Beim ersten Problem würde der Ansatz einer logistischen Regression funktionieren. Da das Kriterium – die Farbe des Rucksacks – nur zwei Ausprägungen hat, spricht man von einer binären logistischen Regression, der am häufigsten eingesetzten Form der logistischen Regression.



Wenn das Kriterium dagegen mehr als zwei Ausprägungen hat, spricht man von einer multinominellen logistischen Regression. Allerdings würde bei unserem Beispiel-Problem Nummer 2 die logistische Regression nicht mehr so gut bei der Klassifikation funktionieren.

Welche weiteren Anwendungsgebiete lassen sich denn für die logistische Regression finden? Prinzipiell lässt sich sagen, dass sich die binäre Regression hervorragend bei Problemen eignet, bei welchen ein binäres Kriterium vorhergesagt werden soll. Ein weiteres Beispiel wäre die Vorhersage, ob ein\*e Studierende\*r die Aufnahmeprüfung schafft, wobei man dessen IQ als Grundlage für die Vorhersage verwendet. Der/Die Studierende kann die Aufnahmeprüfung entweder bestehen oder nicht – das Kriterium hat also zwei Ausprägungen. Ein weiteres häufiges Anwendungsgebiet der logistischen Regression ist in der Medizin bei der Vorhersage von Krankheiten. Hierbei möchte der/die Mediziner\*in in Abhängigkeit von den Parametern eines Patienten/einer Patientin – wie beispielsweise dem Geschlecht, Alter und Raucherstatus – wissen, ob diese Person anfällig für eine bestimmte Krankheit sein könnte. Hierbei kann man die Anzahl der Parameter natürlich noch erhöhen und spezifischere Angaben miteinbeziehen, um die Klassifizierung zu verbessern. Der Einsatz in der Medizin hat seinen Grund vor allem in der probabilistischen Natur der logistischen Regression. Das bedeutet, dass die logistische Regression eigentlich Wahrscheinlichkeiten vorhersagt.

## 2.6.2. Logistische Regression und Wahrscheinlichkeit

Bei der logistischen Regression werden zwar numerische Werte vorhergesagt, anders als bei der linearen oder der polynominalen Regression aber nicht die numerischen Werte des Kriteriums. Stattdessen wird die Wahrscheinlichkeit vorhergesehen, dass das Kriterium in Abhängigkeit von den Prädiktoren in eine der Kategorien fällt. Wenn wir wieder das Beispiel mit der Aufnahmeprüfung nehmen, dann würden wir



schätzen, wie wahrscheinlich es ist, dass ein\*e Studierende\*r mit einem IQ von 112 die Prüfung besteht. Dieses Beispiel ist auch das Einfachste für die Anwendung einer logistischen Regression, weil wir es zum einen mit einem binären Kriterium zu tun haben und zum anderen nur einen Prädiktor zur Klassifikation verwenden. Im Fall einer binären Regression ordnet man praktischerweise der einen Ausprägung die Zahl 0 und der anderen die Zahl 1 zu. Im Fall der Abschlussprüfung wäre dann eine 1 mit dem Status „aufgenommen“ zu assoziieren, während die 0 für den Status „durchgefallen“ steht. Der binären Variable die Werte 0 und 1 zuzuordnen, bietet sich an, da wir es bei Wahrscheinlichkeitswerten immer mit Werten zwischen 0 und 1 zu tun haben. Wie genau berechnet die logistische Regression denn nun die Wahrscheinlichkeiten? Dafür müssen wir zurück auf die lineare Regression schauen.

Schauen wir uns zunächst noch einmal die Gleichung der linearen Regression an:

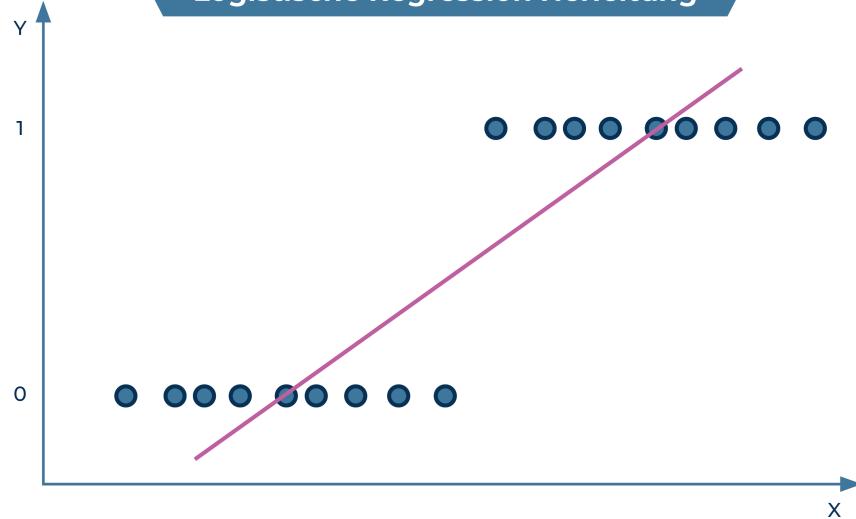
$$y = b_1 \cdot x_1 + b_2 \cdot x_2 + \dots + b_k \cdot x_k + a$$

Da wir es hier mit k unabhängigen Variablen zu tun haben, haben wir es mit einer multiplen Regression zu tun. Das ist einfach die allgemeine Form der Gleichung. Wenn wir k=1 setzen, erhalten wir die normale Geraden-Gleichung einer zweidimensionalen linearen Funktion. Überlegen wir uns mal, wie die Anwendung der linearen Regression bei einem binären Klassifizierungs-Problem aussehen würde. Der Einfachheit halber nehmen wir an, es gäbe nur eine unabhängige Variable zur Beschreibung.

Graphisch würde das Beschriebene folgendermaßen aussehen:



### Logistische Regression Herleitung

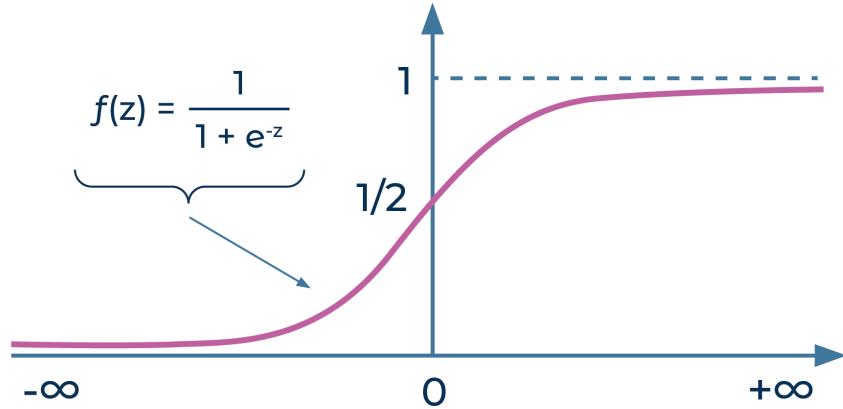


In dem Streudiagramm können wir deutlich eine binäre Aufteilung der Datenpunkte erkennen. Wir wollen also die Variable  $y$  in Abhängigkeit von der Variable  $x$  klassifizieren. Wie wir sehen können, eignet sich die lineare Regression überhaupt nicht zur Vorhersage der Kriteriumswerte, ist also zur Klassifizierung völlig ungeeignet. Die lineare Regression nimmt Werte zwischen minus unendlich und plus unendlich an. Da wir allerdings Wahrscheinlichkeiten vorhersagen wollen, benötigen wir eine Funktion, die Werte zwischen 0 und 1 vorhersagt. Ziel ist es also, den Wertebereich einer – bis jetzt noch hypothetischen logistischen Regressions-Funktion – auf das Intervall zwischen 0 und 1 zu beschränken. Um das zu erreichen, bietet sich die sogenannte Sigmoid-Funktion an.

Die Sigmoid-Funktion ist eine Funktion, die uns im Bereich des Machine Learning noch häufiger begegnen wird. Sie wird auch logistische Funktion genannt. Das besondere an der logistischen Funktion ist, dass sie für alle möglichen  $x$ -Werte nur Werte zwischen 0 und 1 ausspuckt. Tatsächlich erhält man nie als Ergebnis den Wert 0 oder 1 selbst, die Funktion schmiegt sich lediglich im Unendlichen an diese Werte an, erreicht sie allerdings nie. Mathematisch sagt man, die Sigmoid-Funktion konvergiert gegen 0 für den Weg nach minus unendlich und konvergiert gegen 1 für den Weg nach plus.



unendlich. Der Hauptbestandteil der logistischen Funktion ist eine Exponentialfunktion, die sich im Nenner eines Quotienten befindet. Graphisch und mit Formel sieht die logistische Funktion folgendermaßen aus:

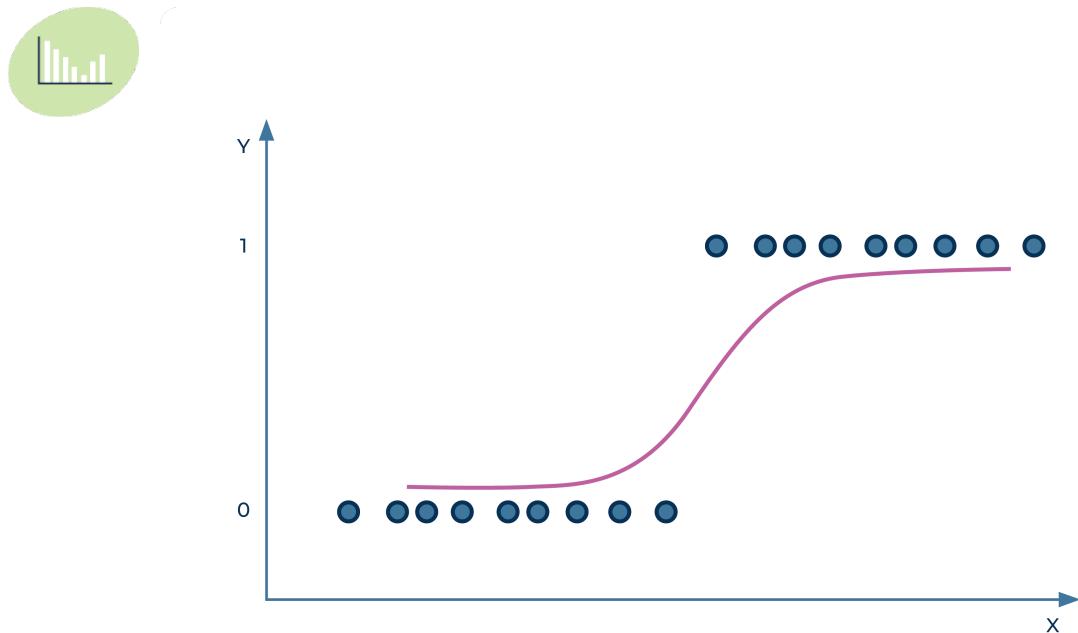


Wie verwenden wir jetzt allerdings die ganzen unabhängigen Variablen zusammen mit der logistischen Funktion zur Vorhersage von Wahrscheinlichkeiten? Um das zu bewerkstelligen, verwenden wir die

$$f(z) = \frac{1}{1 + e^{-z}} = \frac{1}{1 + e^{-(b_1 \cdot x_1 + \dots + b_k \cdot x_k + a)}}$$

Gleichung für die lineare Regression als Argument für die Exponentialfunktion. Das sieht dann so aus:

Wir haben nun also das  $z$  durch die lange lineare Gleichung ersetzt, die sämtliche Prädiktoren zur Vorhersage enthält. Egal in welchem Bereich sich die Werte der Prädiktoren befinden, die Funktion wird stets einen Wert zwischen 0 und 1 vorhersagen. Und diese Werte können wir als Wahrscheinlichkeit dafür nutzen, dass das Kriterium in eine der beiden Kategorien fällt. Zusammen mit Datenpunkten könnte der Graph folgendermaßen aussehen:



In der Praxis berechnet man meistens immer die Wahrscheinlichkeit, dass das Kriterium in die Kategorie 1 fällt. Ist die Wahrscheinlichkeit größer als 0.5, dann ist TRUE, andernfalls FALSE das Ergebnis. Möchte man die Wahrscheinlichkeit berechnen, dass das Kriterium in die Kategorie 0 fällt, dann zieht man die Wahrscheinlichkeit für die Kategorie 1 von 1 ab.

Wie würde diese Formel jetzt beispielsweise im Fall einer medizinischen Diagnose aussehen, bei welcher eine Ärztin in Abhängigkeit von Geschlecht, Alter und Raucherstatus vorhersagen möchte, ob der Patient erkranken wird oder nicht?

$$P(\text{ist krank}) = \frac{1}{1 + e^{-(b_1\text{Alter} + b_2\text{Geschlecht} + b_3\text{Raucherstatus} + a)}}$$

In diesem Fall würde man dem Zustand „krank“ die Zahl 1 zuordnen und dem Zustand „gesund“ die Zahl 0. Die Formel enthält dann die drei Prädiktoren als Argumente in der Exponentialfunktion. Jetzt wissen wir also, welche mathematische Funktion verwendet wird, um die Daten zu beschreiben, aber wie läuft eigentlich das Training ab? Immerhin ist ja die



logistische Regression anscheinend ein Supervised-Learning-Algorithmus, benötigt also Training, bevor er Vorhersagen treffen kann.

Das Training bei der logistischen Regression ist das Anpassen der Parameter in der Exponentialfunktion. Jeder Prädiktor hat einen Parameter  $b$  und zu guter Letzt ist da noch der Parameter  $a$ , der sogenannte y-Achsenabschnitt – der y-Wert, bei dem die Gerade die y-Achse schneidet. Diese Parameter müssen angepasst werden, denn je nachdem welche Werte diese Parameter haben, beschreibt die Funktion die Daten besser oder schlechter. Die Frage, die wir uns jetzt aber natürlich stellen, ist: Wie finden wir eigentlich diese optimalen Parameter?

### 2.6.3. Die Maximum-Likelihood-Methode

Um eine Funktion an die Daten anzupassen, hatten wir bereits die Methode der kleinsten Quadrate kennengelernt, bei welcher wir den quadratischen Abstand zwischen den Datenpunkten und den von der Funktion vorhergesagten Funktion minimieren wollten. Diese Methode wird bei den Regressions-Algorithmen angewandt, die wir uns zuletzt angesehen hatten. Im Fall der logistischen Regression wäre diese Methode auch denkbar, allerdings hat sich in der Praxis ein anderer Ansatz bewährt: die sogenannte Maximum-Likelihood-Methode.

Die gute Neuigkeit vorweg: Die Maximum-Likelihood-Methode ist zwar etwas komplexer, allerdings läuft dieser Algorithmus bei den gängigsten Machine-Learning-Bibliotheken im Hintergrund. In diesem Abschnitt soll es nur um ein Verständnis für dessen Funktionsweise gehen.

Wie funktioniert sie also, die Maximum-Likelihood-Methode?



Zunächst braucht es eine sogenannte Likelihood-Funktion. Wir nennen diese mal L. Diese Likelihood-Funktion L ist abhängig von den Parametern im Modell, also:

$$L(b_1, b_2, \dots, b_n, a)$$

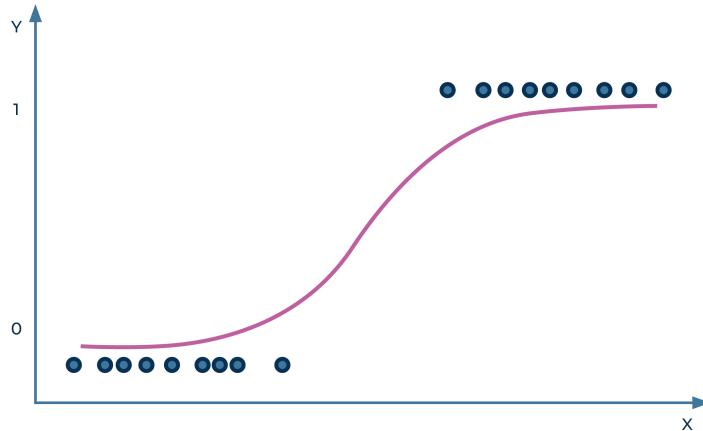
Der Einfachheit halber fassen wir all diese Parameter in einer einzigen Variablen zusammen:

$$L(b_1, b_2, \dots, b_n, a) = L(\theta)$$

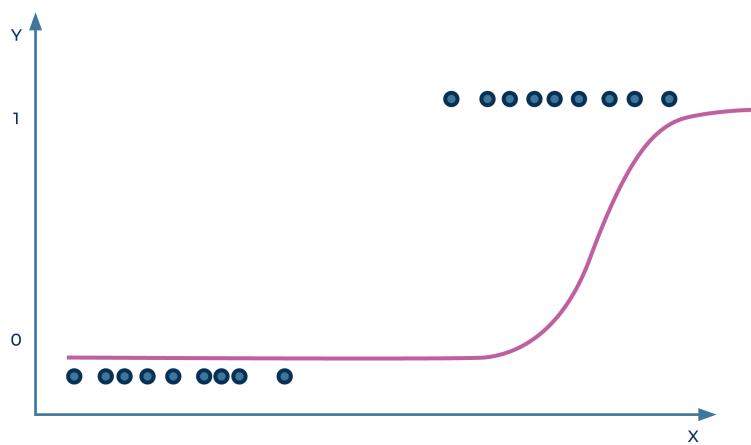
Was macht jetzt diese Likelihood-Funktion eigentlich? Vereinfacht gesagt gibt sie an, wie wahrscheinlich es ist, dass die logistische Funktionsgleichung mit einem Satz an Parametern die beobachteten Datenpunkte vorhersagt. Ändern sich die Parameter, ändert sich auch die Wahrscheinlichkeit, dass die Funktionsgleichung die beobachteten Daten vorhersieht. Visuell kann man sich das ungefähr so vorstellen:



Mit der gegebenen **logistischen Funktion**, ist die Wahrscheinlichkeit „**groß**“, dass die gegebenen Punkte auftreten → Der Wert von  $L(0)$  ist „**groß**“



Mit der gegebenen **logistischen Funktion**, ist die Wahrscheinlichkeit „**klein**“, dass die gegebenen Punkte auftreten → Der Wert von  $L(0)$  ist „**klein**“



Die Kurve der Sigmoid-Funktion passt sich also durch eine Variation der Parameter immer besser an die Datenpunkte an. Im Fall der rechten Kurve besteht keine besonders gute Anpassung. Für viele der beobachteten



Datenpunkte aus der Kategorie 1 würde die Funktion eine Klassifizierung in die andere Kategorie vorschlagen.

Jetzt haben wir die Likelihood-Funktion und verstehen, wie sie funktioniert. Wie können wir diese jetzt also nutzen, um die optimalen Parameter für die logistische Funktion zu bestimmen?

Dafür nehmen wir den Logarithmus der logistischen Funktion, was wir mathematisch folgendermaßen schreiben:

$$LL(\theta) = \log(L(\theta))$$

Diese Log-Likelihood-Funktion soll nun maximiert werden. Es sollen also die Parameter gefunden werden, für die der Logarithmus der Likelihood-Funktion seinen maximalen Wert annimmt. Daher kommt auch der Name Maximum-Likelihood-Funktion. Um dieses Maximum zu bestimmen, wird ein numerisches Verfahren verwendet, welches wir noch eingehender im Kapitel über neuronale Netze kennenlernen werden.

## 2.6.4. Interpretation der Parameter

Bei der linearen Regression waren die Parameter der Gleichung recht einfach zu verstehen. War der Parameter beispielsweise 2, so hatte das direkt eine Verdopplung beim Kriterium zur Folge. Stieg der Parameter, stieg auch der Wert des Kriteriums. Und umgekehrt: Wenn der Parameter negativ war, bedeutet das ein Sinken des Kriteriums-Wertes. Die Parameter bei der linearen Regression sind also recht einfach zu interpretieren. Bei der logistischen Regression sind die Parameter, verschachtelt in einem Quotienten und einer Exponentialfunktion, nicht so einfach zu interpretieren. Dennoch gilt noch die sogenannte „Vorzeicheninterpretation“. Was bedeutet das?



Wenn der Parameter  $b_k$ , der zur unabhängigen Variable  $x_k$  gehört, ein positives Vorzeichen hat, hat ein Anstieg der Variablen  $x_k$  auch einen Anstieg des Kriteriums zur Folge. Umgekehrt gilt das natürlich auch. Wenn der Parameter ein negatives Vorzeichen hat, führt ein Sinken des Parameter-Wertes zu einem Sinken des Kriteriums-Wertes.

Im speziellen Fall der logistischen Regression mit einer probabilistischen Interpretation kann man also eine positive oder negative Korrelation der Parameter mit der Wahrscheinlichkeit erkennen.

Wie allerdings können wir nun den Einfluss der Parameter auf die Werte der logistischen Funktion quantitativ untersuchen? Dafür führen wir das Konzept der sogenannten Odds ein – der Chancen, dass ein Ereignis eintritt. Diese Odds bilden die Grundlage für die sogenannten Logits, dem Logarithmus der Odds. Mit diesem wird es uns möglich, eine lineare Beziehung zu den Parametern herzustellen. Schauen wir uns das mal im Detail an.

Beginnen müssen wir mit der Definition der Odds. Wie oben bereits kurz erwähnt, handelt es sich bei den Odds um das Chancenverhältnis. Dabei wird die Wahrscheinlichkeit, dass ein Ereignis eintritt, ins Verhältnis zu einem Nicht-Eintreten gesetzt. Mathematisch wird das über den folgenden Quotienten erreicht:

$$Odds = \frac{P(y \text{ trifft ein})}{P(y \text{ trifft nicht ein})} = \frac{P(y \text{ trifft ein})}{1 - P(y \text{ trifft ein})}$$

Wenn die Odds  $> 1$ , dann bedeutet das, dass es wahrscheinlicher ist, dass das Ereignis eintritt, als dass es nicht eintritt.

Wenn die Odds  $= 1$ , dann bedeutet das, dass sowohl ein Eintreten als auch ein Nicht-Eintreten des Ereignisses gleich wahrscheinlich sind.



Sind die Odds  $< 1$ , dann bedeutet das, dass es wahrscheinlicher ist, dass das Ereignis nicht eintritt, als dass es eintritt.

Die Odds können also Werte zwischen 0 und unendlich annehmen.

Wenn wir jetzt davon den Logarithmus nehmen, dann bekommen wir eine Funktion, die Werte zwischen minus und plus unendlich ausgeben kann – also perfekt geeignet ist, um einen linearen Zusammenhang zu den Parametern herzustellen. Der Logarithmus der Odds sieht dann folgendermaßen aus:

$$\text{logits} = \log\left(\frac{p}{1-p}\right)$$

Wie hängt das jetzt mit den Parametern der logistischen Funktion zusammen? Hierfür müssen wir in die obige Gleichung für die Wahrscheinlichkeit  $p$  die Formel der logistischen Regression einsetzen. Dann erhalten wir für den Quotienten Folgendes:

$$\frac{1-p}{p} = \frac{1}{\exp(\beta_0 + \beta_1 x_1 + \dots + \beta_k x_k)}$$

Wenn man von dem Inversen dieser Gleichung den Logarithmus nimmt, erhält man:

$$\text{logit}(p) = \log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_1 + \dots + \beta_k x_k$$

Wie können wir nun damit quantitativ den Einfluss der Parameter auf den Ausgang des Ergebnisses untersuchen? Schauen wir uns dafür an, was

$$\frac{\text{odds}_{x_j+1}}{\text{odds}} = \frac{\exp(\theta_0 + \theta_1 x_1 + \dots + \theta_j(x_j + 1) + \dots + \theta_p x_p)}{\exp(\theta_0 + \theta_1 x_1 + \dots + \theta_j x_j + \dots + \theta_p x_p)}$$



passiert, wenn wir den Wert einer der unabhängigen Variablen um 1 erhöhen:

In diesem Beispiel erhöhen wir die Variable  $x_j$  um 1. Das Verhältnis der Odds ist ein Bruch mit zwei Exponentialfunktionen. Mit der Regel

$$\frac{\exp(a)}{\exp(b)} = \exp(a - b)$$

erhalten wir:

$$\frac{odds_{x_j+1}}{odds} = \exp(\theta_j(x_j + 1) - \theta_j x_j) = \exp(\theta_j)$$

Am Ende haben wir also einen recht simplen quantitativen Zusammenhang zwischen einem Parameter und dem Verhältnis der Odds feststellen können. Eine Änderung eines der Features um 1 führt also zu einer exponentiellen Änderung in dem Verhältnis der Odds. Dennoch bleibt die Interpretation der Parameter in der logistischen Regression nicht ganz so einfach.

## 2.6.5. Multinomische logistische Regression

Wir wissen jetzt, wie die logistische Regression für eine Kriteriums-Variable funktioniert, die nur zwei Ausprägungen hat, also eine binäre logistische Regression. Wie allerdings können wir die obigen Methoden nutzen, um beispielsweise eine Klassifizierung in drei Kategorien vornehmen zu können?

Die Antwort in einfacher Form vorweg: Wir führen mehrere binäre logistische Regressionen aus.

Schauen wir uns das anhand des Beispiels mit drei Merkmalsausprägungen an:

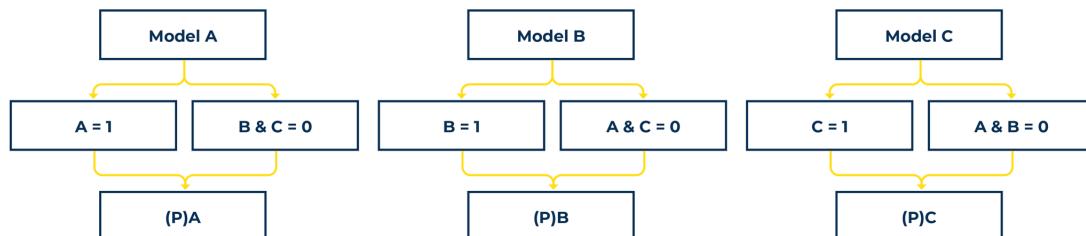


Nennen wir unsere 3 Kategorien A, B und C. Wir werden für jede dieser Kategorien ein eigenes binäres logistisches Regressions-Modell erstellen. Im ersten Modell wird es die Kategorie A gegen die Kategorien B&C sein. Was bedeutet das?

Im ersten Modell bestimmen wir eine logistische Funktion, die die Wahrscheinlichkeit für den Ausgang A oder Nicht-A vorhersagt. Und Nicht-A ist einfach der Ausgang B&C.

Analog ist das zweite Modell für B gegen A&C und das dritte Modell für C gegen A&B.

Im ersten Modell ist A=1, im zweiten B=1 und im dritten C=1. Für jedes dieser Modelle bestimmen wir die optimalen Parameter und damit die optimale logistische Funktion.



Die Klassifizierung am Ende findet über die Wahrscheinlichkeitsfunktionen statt, die wir in jedem der drei Modelle bestimmt haben. Wenn wir eine Klassifizierung vornehmen, dann wählen wir jene Klasse, für welche die Wahrscheinlichkeit am höchsten ist. Wenn also  $P(A)>P(B)$  und  $P(A)>P(C)$ , dann gehört das Kriterium in die Kategorie A.

Hier kann man auch sehen, wieso das Kriterium nicht zu viele Ausprägungen haben sollte. Die Wahrscheinlichkeiten für die einzelnen Ausprägungen werden dann immer kleiner und es wird schwieriger, eindeutig zwischen diesen differenzieren zu können.



## 2.6.6. Programmierung der logistischen Regression

In diesem Abschnitt wollen wir uns endlich der Praxis widmen und einen logistischen Regressions-Algorithmus in Python programmieren. Dafür nehmen wir an, wir hätten einen kleinen Datensatz, der uns den gemessenen Durchmesser von entdeckten Tumoren angibt und ob diese bösartig sind oder nicht. Ziel dieser logistischen Regression wird es also sein, anhand des Prädiktors „Durchmesser“ eine Wahrscheinlichkeit für ein Eintreten des Kriteriums „Bösartig“ zu berechnen. Dafür werden wir wieder die scikit-learn-Bibliothek verwenden. Fangen wir aber erst einmal damit an, die Daten vorzubereiten.

```
import numpy
import matplotlib.pyplot as plt
from sklearn import linear_model

# X repräsentiert den Durchmesser eines Tumors in Zentimeter
X = numpy.array([3.78, 2.44, 2.09, 0.14, 1.72, 1.65, 4.92, 4.37, 4.96, 4.52, 3.69, 5.88]).reshape(-1,1)

# X "re-shaped" werden für die LogisticRegression()-Funktion:
# von einer Zeile in eine Spalte.

#y repräsentiert, ob der Tumor bösartig ist oder nicht (0 für "Nein", 1 für "Ja").
y = numpy.array([0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1])

plt.scatter(X,y)
plt.show()
```

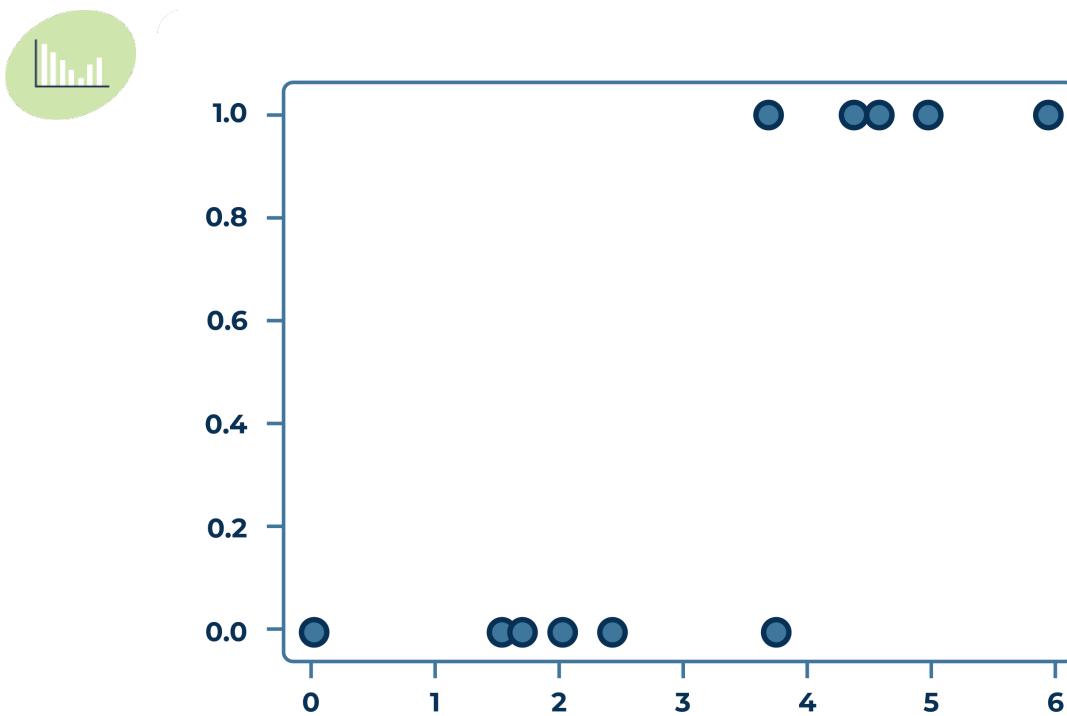
Zu Beginn importieren wir zuallererst alle Bibliotheken und Module, die wir in diesem Code benötigen werden. Die numpy- und die matplotlib-Bibliothek dürfen in einem Data-Science- Projekt eigentlich nie fehlen. Dann importieren wir noch aus der sklearn-Bibliothek das Modul linear\_model. Dieses wird eine Klasse mit dem Namen LogisticRegression() enthalten, mit der wir ein logistisches Regressions-Objekt erzeugen können. Aber zunächst einmal zu den Daten.

Den Durchmesser des Tumors haben wir in ein numpy-Array gepackt und anschließend „re- shaped“: von einer Zeile wird es in eine Spalte



transformiert. Dieser Schritt ist notwendig, damit die sklearn-Funktion später damit arbeiten kann.

Das Kriterium  $y$  ist ganz einfach ein Array mit den Werten 0 und 1. Die 0 steht hierbei für einen gutartigen Tumor und die 1 für einen bösartigen Tumor. Zu guter Letzt wollen wir uns anschauen, wie das Ganze graphisch aussehen wird:



Das Streudiagramm zeigt uns wieder deutlich, dass hier eine gewöhnliche lineare Regression oder selbst eine Polynom-Regression nichts mehr bringen würden. Auf der x-Achse sind die Durchmesser der Tumore aufgetragen, die y-Achse gibt die Zuordnung gut- oder bösartig an. Nun wollen wir eine logistische Regression für diese Daten durchführen. Dafür hängen wir an den bereits geschriebenen Code den folgenden ran:



```
#Erzeuge ein logistische Regressions-Objekt
logr = linear_model.LogisticRegression()
logr.fit(X,y)

#Klassifikation an einem Test-Tumor
predicted = logr.predict(numpy.array([1.46]).reshape(-1,1))
print(predicted)
```

Mit der LogisticRegression()-Klasse erzeugen wir ein logistisches Regressions-Objekt. Dieses enthält die bekannte fit()-Methode, mit der wir das Objekt an den Datensatz (X,y) anpassen. Anschließend nutzen wir die predict()-Funktion des Objekts, um für einen Tumordurchmesser von 1.46 Zentimetern vorherzusagen, ob er bösartig sein wird oder nicht. Das Ergebnis wird in der Variable predicted gespeichert und mit dem print()-Befehl ausgegeben.

Das Ergebnis ist: [0]

Das bedeutet, dass der Tumor der Klasse „gutartig“ zugeordnet wird.

Möchte man sich die Odds ausgeben lassen, dann geht das mit dem coefficient-Attribut, das das logistische Regressions-Objekt hat. Mit dem Befehl log\_odds = logr.coef\_ speichert man in der log\_odds-Variable die Logits, die ja der Logarithmus der Odds sind. Also müssen wir das anschließend noch in eine Exponentialfunktion packen, um die Odds zu erhalten.

```
log_odds = logr.coef_
odds = numpy.exp(log_odds)

print(odds)
```

Das Ergebnis im obigen Fall wäre: [4.03541657]



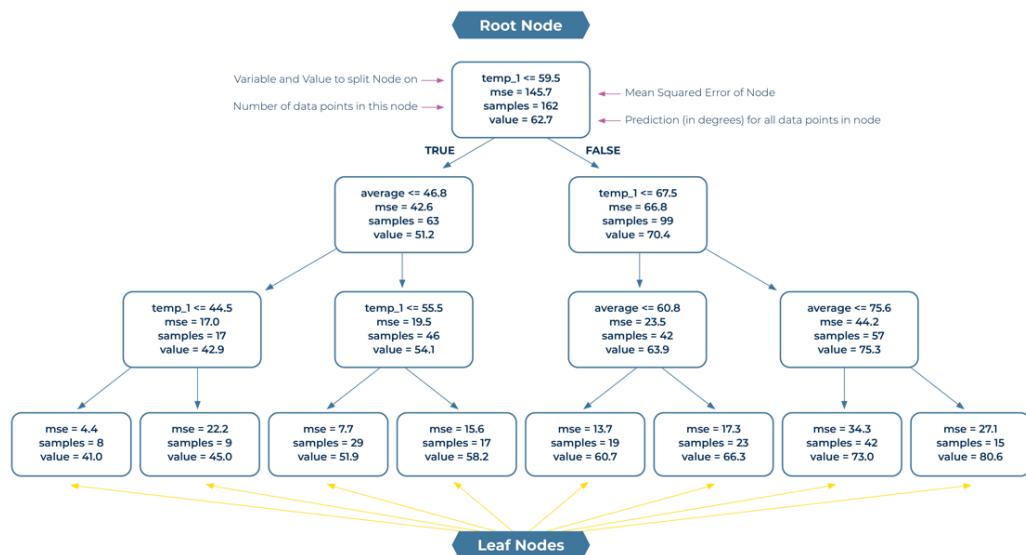
Das bedeutet, wenn man den Tumor um einen Millimeter vergrößert, erhöhen sich die Chancen (die Odds) einer Bösartigkeit auf das Vierfache.

## 2.7. Decision Trees

In diesem Abschnitt werden wir die Klassifizierungsmethode Decision Tree kennenlernen. Dabei schauen wir uns zunächst an, was ein Decision Tree ist und wie man ihn aufbaut bzw. trainiert. Bei diesem Prozess wird uns der Information Gain aus dem Kapitel über Informationstheorie wieder begegnen, wir werden aber noch weitere Methoden wie die Gain Ratio und die Gini Impurity kennenlernen. Anschließend lernen wir die sogenannten Regression Trees kennen, eine Art von Decision Tree, die zur Vorhersage von numerischen Werten – also zur Regression – verwendet werden kann. Dann schauen wir uns an, wie man einen Decision Tree mit scikit-learn programmieren kann und welche Vor- und Nachteile in der Nutzung entstehen.

### 2.7.1. Was ist ein Decision Tree?

Der nächste Klassifizierungsalgorithmus aus dem Bereich des Supervised Learning, den wir kennenlernen werden, ist der sogenannte Decision-Tree-Algorithmus. Auf Deutsch bedeutet das Entscheidungsbaum. Decision Trees werden zwar hauptsächlich zur Klassifikation genutzt, können allerdings auch zur Regression verwendet werden, also zur Vorhersage von numerischen Werten. Der Name Decision Tree leitet sich aus dessen Erscheinungsbild ab. Ein Beispiel ist in der unteren Grafik zu sehen.



Decision Trees werden von oben nach unten gelesen, das bedeutet, dass die Klassifikation von oben nach unten erfolgt. Es beginnt beim Root Node, wo die erste TRUE/FALSE-Frage gestellt wird und der Baum sich in zwei neue Nodes aufspaltet. An jedem einzelnen dieser gibt es dann wieder eine binäre Verästelung. Von jedem der Nodes im Decision Tree geht eine binäre Entscheidung aus.

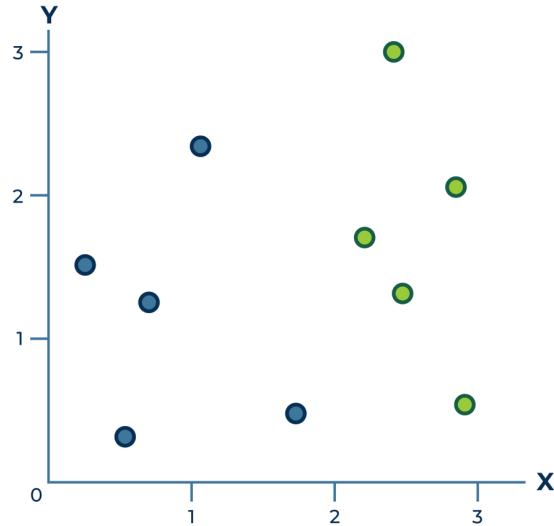
Da Decision Trees ein sehr anschauliches Instrument aus dem Machine Learning sind, werden sie gerne verwendet, um die Entscheidungsfindung bei einem Klassifizierungsproblem besser zu verstehen und auch zu visualisieren. Außerdem lassen sich beliebig viele Entscheidungsebenen im Algorithmus implementieren. Decision Trees werden beispielsweise von Banken eingesetzt, um anhand bestimmter Kriterien die Kreditwürdigkeit von Kunden/Kundinnen vorherzusagen. Außerdem können Decision Trees auch beispielsweise bei der Analyse von undurchsichtigen Kostenstrukturen helfen. Dadurch lässt sich dann deutlich machen, welche Entscheidungen zu höheren Kosten führen.

Um aber genau zu verstehen, wie Decision Trees funktionieren und zur Klassifikation genutzt werden können, sollten wir uns beispielhaft die Entwicklung eines einfachen Decision Trees anschauen.

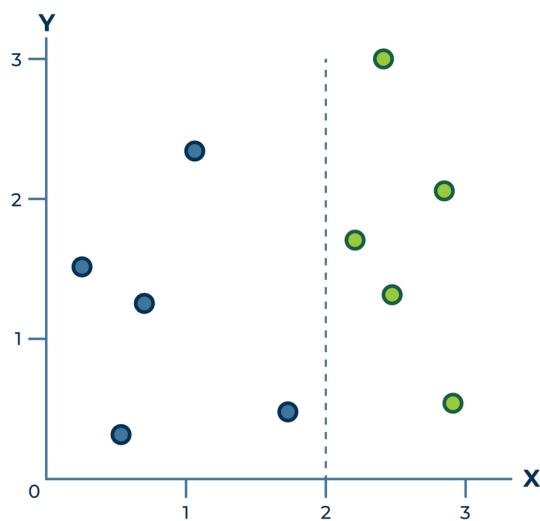


## 2.7.2. Aufbau eines Decision Trees

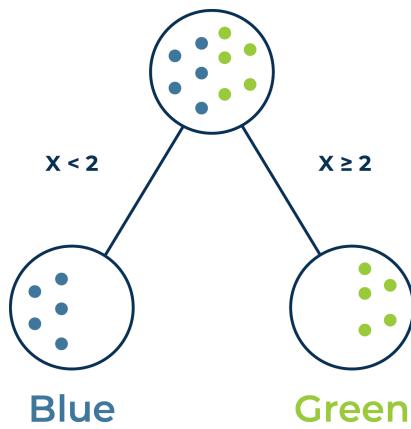
Fangen wir mit folgenden Daten in einem Streudiagramm dargestellt an:



Es besteht eine deutliche Aufteilung zwischen den blauen und den grünen Punkten. Würden wir jetzt einen Punkt in die Ebene tun, der die x-Koordinate 1 hätte, würden wir ihm die Farbe Blau geben. Wir würden ihn also als blau klassifizieren. Die blauen und grünen Punkte werden durch die senkrechte Linie, die durch  $x=2$  geht, sauber voneinander getrennt.

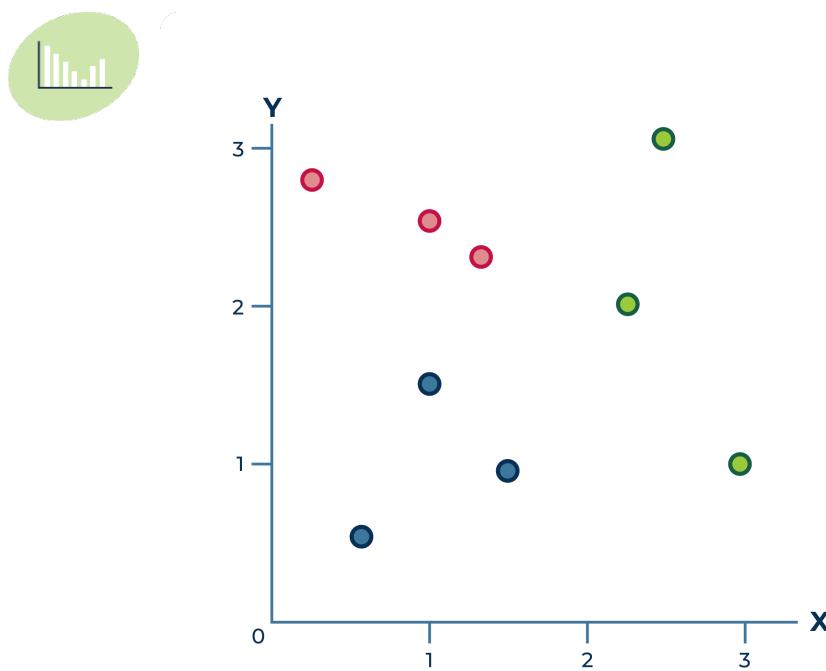


Diese Entscheidung können wir als Decision Tree darstellen:

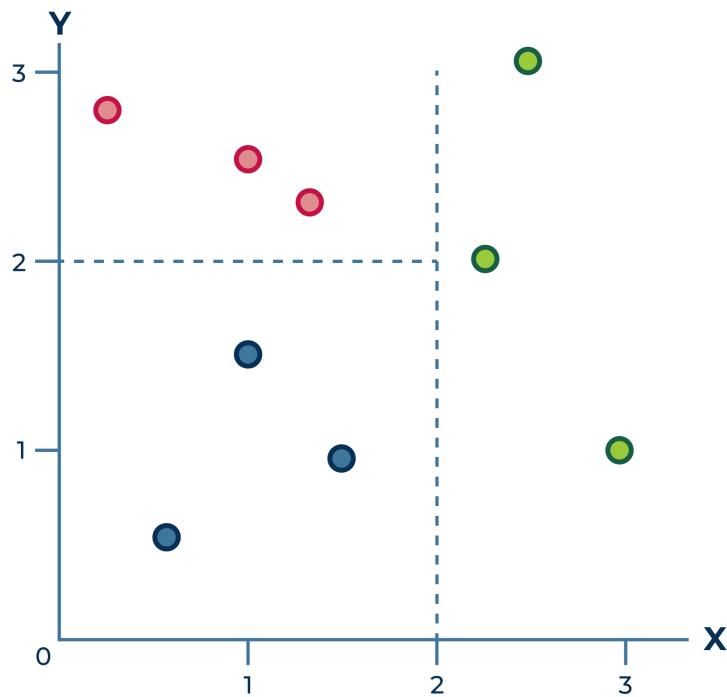


Wenn die x-Koordinate eines Datenpunktes kleiner als 2 ist, dann wird ein Datenpunkt als blau klassifiziert, wenn sie dagegen größer oder gleich 2 ist, wird der Datenpunkt als grün klassifiziert.

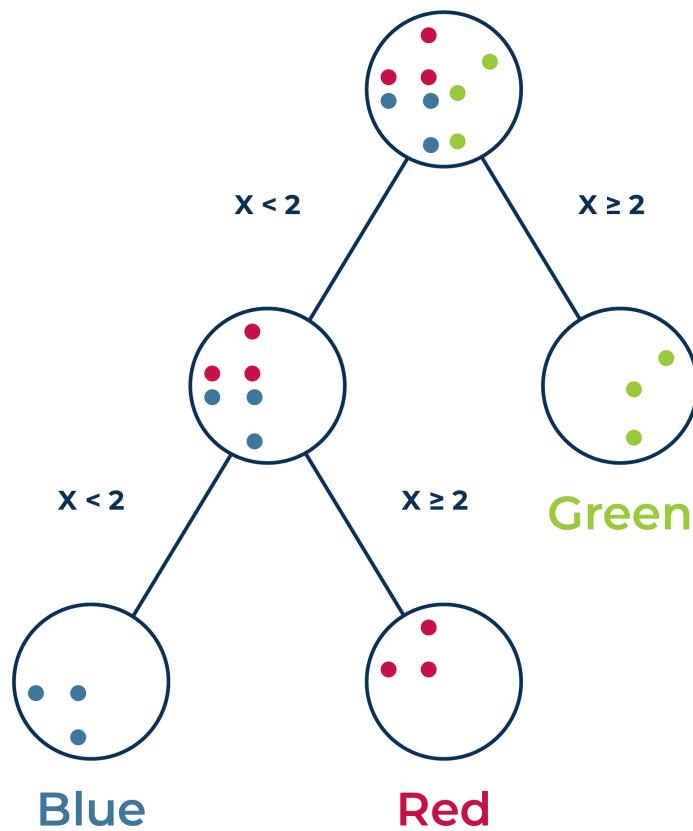
Machen wir das Ganze etwas komplizierter und fügen noch eine dritte Klasse an Datenpunkten hinzu – die Roten.



Wenn wir jetzt die gleiche Klassifizierungsmethode wie zu Beginn verwenden, werden die Grünen immer noch korrekt klassifiziert. Jeder Datenpunkt, für den die x-Koordinate größer oder gleich 2 ist, ist immer noch grün. Auf der linken Seite sieht das Ganze aber anders aus. Die Datenpunkte mit einer x-Koordinate kleiner als 2 können sowohl rot als auch grün sein. Wir müssen also unseren Decision Tree um eine weitere Entscheidungsebene erweitern, um eine passendere Klassifizierung vornehmen zu können.



Zusätzlich zu der Bedingung  $x < 2$  kommt nun für eine Differenzierung zwischen rot und blau noch die Bedingung  $y > 2$  hinzu. Wenn  $x < 2$  und  $y > 2$ , dann handelt es sich um einen Datenpunkt der Klasse Rot. Wenn  $x < 2$  und  $y < 2$ , dann handelt es sich um einen Datenpunkt der Klasse Blau. Als Entscheidungsbaum mit zusätzlicher Entscheidungsebene sieht das dann folgendermaßen aus:



### 2.7.3. Training des Decision Trees

Wir haben jetzt also gesehen, wie ein Decision Tree prinzipiell aufgebaut ist und wie er zur Klassifizierung von Daten verwendet werden kann. Die Frage, die wir uns jetzt aber stellen müssen, ist: Wie trainiert man einen Decision Tree? Immerhin soll es sich ja um einen Supervised-Learning-Algorithmus handeln.

Brechen wir diese Frage weiter runter. Woher weiß der Algorithmus, welche Entscheidungsfragen er in welcher Reihenfolge stellen muss?

Nehmen wir wieder obiges Beispiel mit den drei Klassen von Daten: rot, grün und blau. Woher weiß der Algorithmus, ob es besser ist, zuerst die



grünen Datenpunkte von allen übrigen zu trennen oder doch erst mit einer Separation der roten Punkte anzufangen?

Welches Entscheidungskriterium eignet sich also am besten, um damit anzufangen und den Root Node zu konstruieren?

Prinzipiell gibt es drei Möglichkeiten, um die optimalen Split-Kriterien zu bestimmen. Eine dieser Methoden haben wir bereits kennengelernt, als wir uns im ersten Kapitel mit den Grundlagen der Informationstheorie auseinandergesetzt haben.

Die Reinheit der Daten bestimmt die Qualität eines Split-Kriteriums. Diese haben wir über die Entropie definiert und mit der Entropie konnten wir den Begriff des Informationsgewinns definieren.

Schauen wir uns mal die anderen beiden Methoden an.

### **Gain Ratio**

Eines der Probleme bei der Verwendung des Informationsgewinns zur Bestimmung des optimalen Split-Kriteriums ist dessen Voreingenommenheit. Was das bedeutet? Je höher die Anzahl der Elemente in einem Split, umso größer der Informationsgewinn. Um dieses Problem zu vermeiden, bezieht man zusätzlich einfach die Anzahl der Werte eines Splits mit ein. So definiert sich die sogenannte Gain Ratio mit folgender Formel:

$$\text{GainRatio} = \frac{IG}{SplitInfo}$$



Bei der Gain Ratio handelt es sich also um das Verhältnis vom Informationsgewinn zur sogenannten Split-Info. Diese Split-Info ist die Information, die aus dem Split hervorgeht, und ist einfach die Entropie des Features:

$$\text{SplitInfo} = - \sum_{i=1}^n \frac{N_i}{N} \cdot \log_2\left(\frac{N_i}{N}\right)$$

Um besser zu verstehen, was die Formeln bedeuten, wollen wir sie mal in der Anwendung sehen. Schauen wir uns dafür wieder den Wetter-Datensatz an, mit dem wir im ersten Kapitel gearbeitet haben.

Tag	Wetteraussicht	Temperatur	Luftfeuchtigkeit	Wind	Aussenverkauf
1	sonnig	heiss	hoch	schwach	nein
2	sonnig	heiss	hoch	stark	nein
3	bewölkt	heiss	hoch	schwach	ja
4	regnerisch	mild	hoch	schwach	ja
5	regnerisch	kalt	normal	schwach	ja
6	regnerisch	kalt	normal	stark	nein
7	bewölkt	kalt	normal	stark	ja
8	sonnig	mild	hoch	schwach	nein
9	sonnig	kalt	normal	schwach	ja
10	regnerisch	mild	normal	schwach	ja
11	sonnig	mild	normal	stark	ja
12	bewölkt	mild	hoch	stark	ja
13	bewölkt	heiss	normal	schwach	ja
14	regnerisch	mild	hoch	stark	nein

Nehmen wir mal die vorletzte Spalte mit der Benennung „Wind“. Diese Spalte hat die beiden Ausprägungen „Weak“ (schwach) und „Strong“ (stark). Wir wollen jetzt für diese Spalte beispielhaft die Gain Ratio berechnen.

Für die Gain Ratio benötigen wir zunächst einmal den Informationsgewinn, den wir bereits im ersten Kapitel für die einzelnen Spalten dieses Datensatzes berechnet haben. Das Ergebnis für die Spalte „Wind“ war:

$$\text{IG(Wind)} = 0.048$$



Nun müssen wir noch die Split-Info für die Spalte „Wind“ berechnen. Wir wollen also wissen, wie viel Information wir aus dem Split zwischen „Weak“ und „Strong“ gewinnen können. Wenn wir die entsprechenden Werte in die obige Formel einsetzen, erhalten wir:

$$\text{SplitInfo} = -\frac{8}{14} \cdot -\log_2\left(\frac{8}{14}\right) - \frac{6}{14} \cdot -\log_2\left(\frac{6}{14}\right) = 0.985$$

Daraus können wir dann ganz einfach die Gain Ratio schlussfolgern:

$$\text{GainRatio} = \frac{0.048}{0.985} = 0.049$$

### Gini Impurity

Eine weitere Methode, die besonders gerne bei Decision Trees eingesetzt wird, ist die sogenannte Gini Impurity eines Features in einem Datensatz. Wie kann man diese neue Art der Unreinheit eines Datensatzes nun am besten verstehen?

Die Gini Impurity ist vereinfacht gesagt die Wahrscheinlichkeit, dass man ein zufälliges Element in einem Datensatz falsch klassifiziert.

Als Formel sieht das Ganze so aus:

$$G = \sum_{i=1}^C p(i) * (1 - p(i))$$

In dieser Formel steht C für die Anzahl der Klassen, die das Feature hat. Der Ausdruck  $p(i)$  steht für die Wahrscheinlichkeit, zufällig ein Element der Klasse i aus dem Datensatz zu fischen. Beim Training eines Decision Trees findet man den besten Split, indem man versucht, den sogenannten Gini-Gain zu maximieren. Diesen berechnet man, indem man die gewichteten Unreinheiten der Äste des Baums von der ursprünglichen Unreinheit abzieht. Schauen wir uns das Ganze mal Schritt für Schritt an einem Beispiel an.

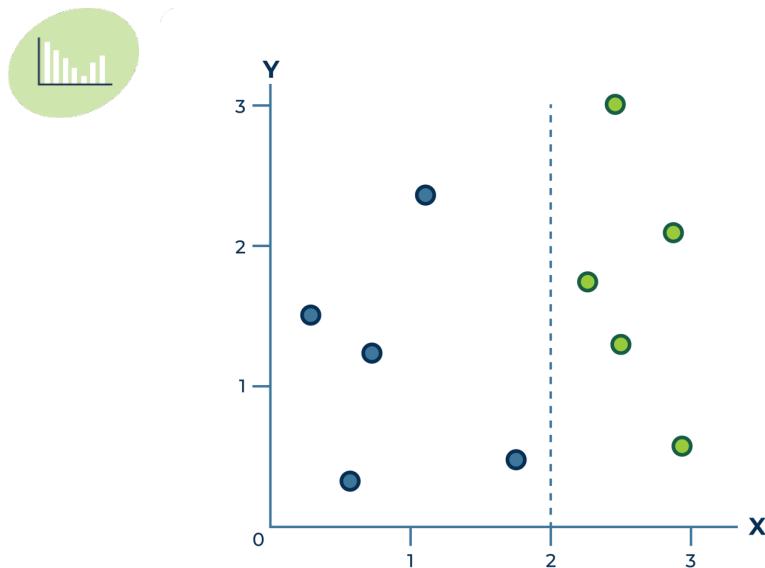


Fangen wir wieder mit dem ursprünglichen Datensatz aus grünen und blauen Punkten an. Wir haben 5 blaue und 5 grüne Punkte, es besteht also eine gleichmäßige Aufteilung. Wir haben 2 Klassen und wissen eigentlich alles, um die Gini Impurity dieses Datensatzes zu berechnen. Die Wahrscheinlichkeit, einen blauen Punkt zufällig zu wählen, ist nämlich 0.5, genauso wie die Wahrscheinlichkeit, einen grünen Punkt zu wählen. Also erhalten wir für die Gini Impurity – wenn wir alles in die Formel einsetzen – folgenden Wert:

$$\begin{aligned}G &= p(1) * (1 - p(1)) + p(2) * (1 - p(2)) \\&= 0.5 * (1 - 0.5) + 0.5 * (1 - 0.5) \\&= \boxed{0.5}\end{aligned}$$

Jetzt wollen wir uns anschauen, wie sich die Unreinheit der Daten verändert, wenn wir verschiedene Splits vornehmen.

Fangen wir mit einem perfekten Split bei  $x=2$  an, der die Datenpunkte einwandfrei voneinander differenziert.





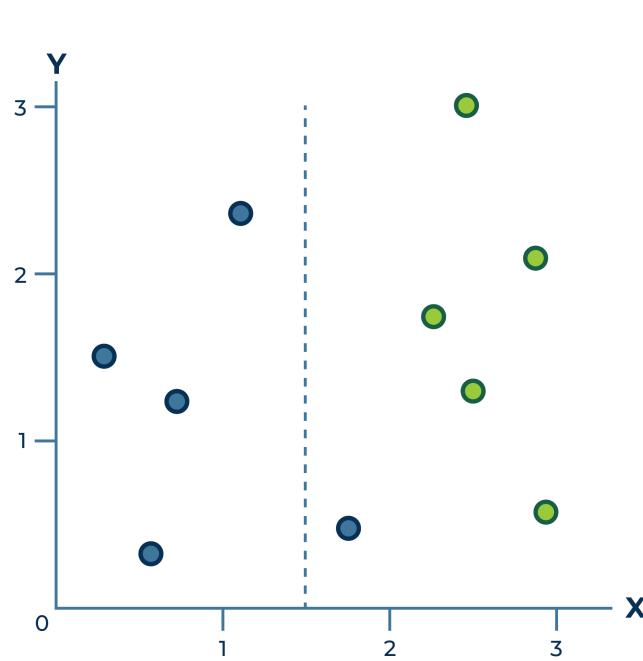
Berechnen wir mal die Gini Impurities für beide Seiten dieses Splits. Auf der linken Seite gilt:

$$G_{left} = 1 * (1 - 1) + 0 * (1 - 0) = \boxed{0}$$

Für die rechte Seite gilt analog:

$$G_{right} = 0 * (1 - 0) + 1 * (1 - 1) = \boxed{0}$$

Beide Seiten haben jetzt also eine Gini Impurity von 0! Das ist der niedrigste mögliche Wert, den die Gini Impurity annehmen kann, und es ist auch der beste. Eine solche Gini Impurity bekommt man nur, wenn sich in einem Datensatz lediglich Daten einer einzigen Klasse befinden. Wir haben es also nun mit einem Split bei  $x=2$  geschafft, dass sich die Gini Impurity von 0.5 auf 0 verringert. Wie schaut das Ganze eigentlich aus, wenn wir den Split an einer anderen Stelle setzen? Beispielsweise bei  $x=1.5$ ?





Da die linke Seite immer noch nur blaue Datenpunkte enthält, bleibt die Gini Impurity auf dieser Seite 0. Wie sieht es aber mit der rechten Seite aus? Da hat sich zusätzlich zu den 5 grünen Punkten noch ein blauer eingeschlichen. Indem wir alles in die Formel einsetzen, erhalten wir für die Gini Impurity:

$$\begin{aligned}
 G_{right} &= \frac{1}{6} * \left(1 - \frac{1}{6}\right) + \frac{5}{6} * \left(1 - \frac{5}{6}\right) \\
 &= \frac{5}{18} \\
 &= \boxed{0.278}
 \end{aligned}$$

So, jetzt wissen wir, wie wir die Gini Impurity vor und nach einem Split berechnen können, aber was sagt uns das jetzt über die Qualität eines Splits aus? Das ist ja genau das, was wir für unseren Decision Tree bestimmen können müssen. Hier kommt dann der sogenannte Gini-Gain ins Spiel.

Für den Gini-Gain bestimmen wir zunächst die gewichtete Summe der Impurities der einzelnen Splits. Wir teilen also die Datenpunkte in zwei Bereiche auf. Für jeden dieser Bereiche berechnen wir die Impurity, multiplizieren sie aber mit der Anzahl der Elemente, die sich in diesem Bereich befinden. Für unseren letzten Split, in welchem der linke Bereich 4 Datenpunkte und der rechte 6 Datenpunkte enthalten, ist die gewichtete Summe der Impurities Folgende:

$$(0.4 * 0) + (0.6 * 0.278) = 0.167$$

Der Gini-Gain ist jetzt also als der Gewinn an Reinheit definiert. Wie viel Reinheit gewinnt der Datensatz durch die Durchführung eines Splits? Diesen Gini-Gain erhält man, indem man die gewichtete Summe von der

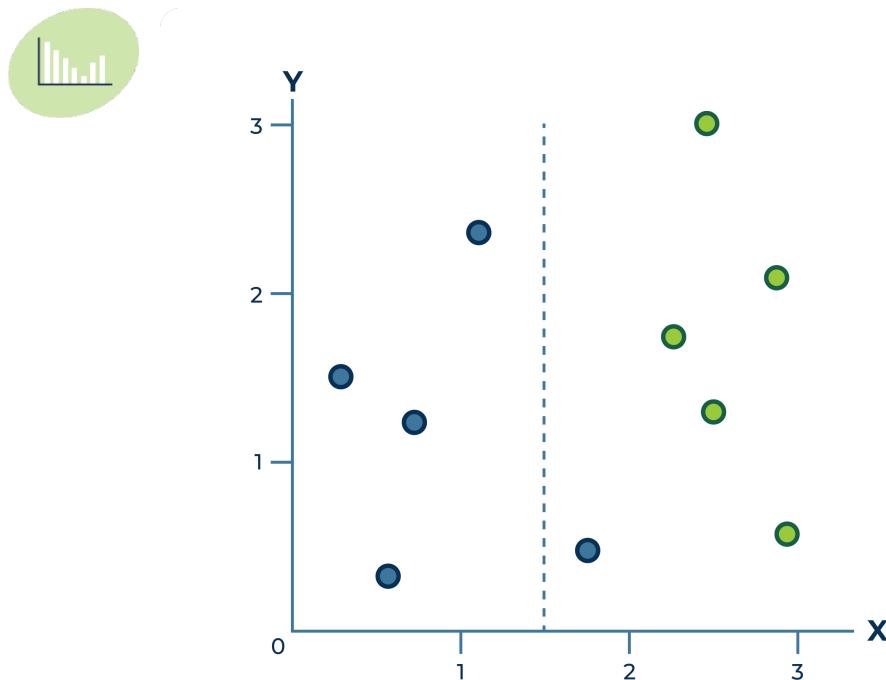
$$0.5 - 0.167 = \boxed{0.333}$$

ursprünglichen Gini Impurity abzieht. Im obigen Fall erhält man dann:



Wir haben also einen Gini-Gain von 0.333. Je höher der Gini-Gain, desto besser der Split. Bei unserem ersten Split bei  $x=2$  hatten beide Bereiche am Ende eine Impurity von 0, was zu einem Gini-Gain von 0.5 führt. Dementsprechend ist der erste Split bei  $x=2$  besser als der zweite bei  $x=1.5$ .

Schauen wir uns nun an, wie die Gini-Gain-Berechnung bei einem etwas komplexeren Beispiel mit drei Klassen an Datenpunkten aussieht. Gehen wir wieder zurück zu unserem anfänglichen Beispiel mit roten, grünen und blauen Datenpunkten.



Als erstes wollen wir die Gini Impurity der Daten berechnen, bevor wir irgendeinen Split durchführen. Dabei erhalten wir den folgenden Wert:

$$\begin{aligned}G_{initial} &= \sum_{i=1}^3 p(i) * (1 - p(i)) \\&= 3 * \left(\frac{1}{3} * \frac{2}{3}\right) \\&= \boxed{\frac{2}{3}}\end{aligned}$$



Nehmen wir nun an, dass wir einen Split bei  $x=0.4$  durchführen. Dann erhalten wir eine senkrechte Linie, die die Datenpunkte in zwei Bereiche teilt: den linken mit nur einem roten Datenpunkt und den rechten mit roten, grünen und blauen Datenpunkten. Um am Ende den Gini-Gain zu bestimmen, müssen wir jetzt die Gini Impurity vom linken und vom rechten Bereich berechnen. Da sich im linken Bereich nur ein roter Datenpunkt

$$G_{left} = 0 * 1 + 1 * 0 + 0 * 1 = \boxed{0}$$

$$\begin{aligned} G_{right} &= \frac{3}{8} * \frac{5}{8} + \frac{2}{8} * \frac{6}{8} + \frac{3}{8} * \frac{5}{8} \\ &= \boxed{\frac{21}{32}} \end{aligned}$$

befindet, ist dessen Gini Impurity offensichtlich 0. Für die rechte Seite sieht das etwas komplizierter aus:

Jetzt haben wir alles, um den Gini-Gain dieses Splits bei  $x=0.4$  zu berechnen. Dafür ziehen wir nur von der ursprünglichen Unreinheit die gewichteten Impurities der beiden Bereiche ab. Damit erhalten wir als Gini-Gain:

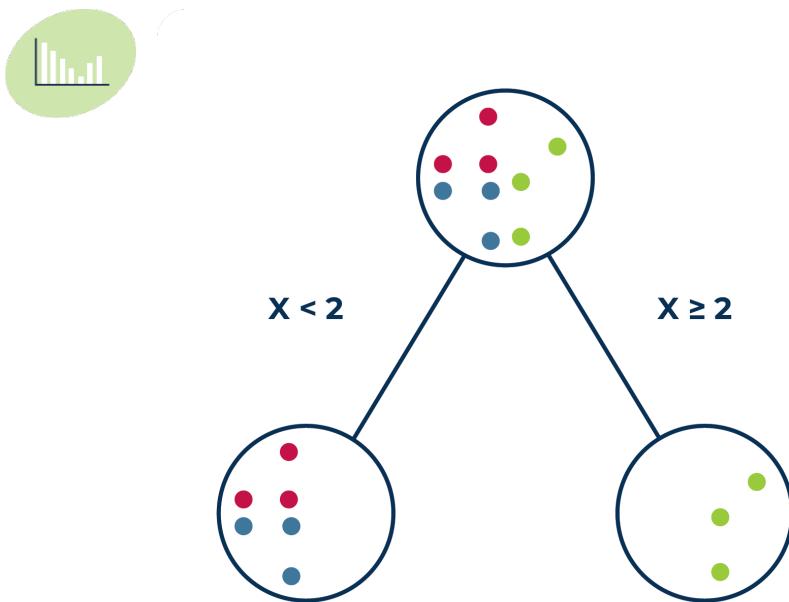
$$\begin{aligned} \text{Gain} &= G_{initial} - \frac{1}{9}G_{left} - \frac{8}{9}G_{right} \\ &= \frac{2}{3} - \frac{1}{9} * 0 - \frac{8}{9} * \frac{21}{32} \\ &= \boxed{0.083} \end{aligned}$$

Der Gini-Gain hat einen Wert von 0.083, ist also nicht besonders hoch. Er sagt uns, dass sich die Unreinheit durch die Durchführung des Splits um 0.083 verringert hat. In der folgenden Übung schauen wir uns an, wie sich die Gini-Gains je nach Wahl des Splits ändern.

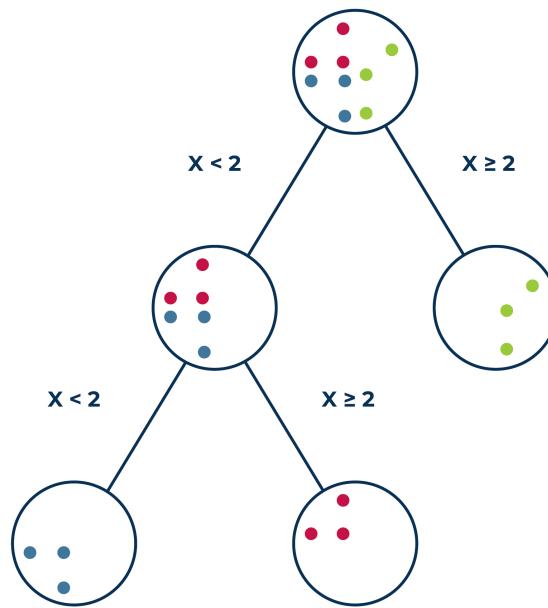
## Training des Decision Trees – Fortsetzung



Wir haben nun herausgefunden, dass der optimale Split für die obigen Datenpunkte bei  $x=2$  war, denn der Gini-Gain für einen Split bei diesem Wert war am höchsten. Da wir ja die Gini Impurity und den Gini-Gain letztendlich dafür verwenden wollen, um unseren Decision Tree zu trainieren, wollen wir uns nun dem widmen. Wir können dank des optimalen Splits bei  $x=2$  nun den Root-Node des Decision Trees aufzeichnen:



Wie geht es jetzt weiter? Was machen wir mit den Blättern des Baumes? Nun gehen wir die ganze Prozedur für die nächsten Nodes noch einmal durch. Da der rechte Node bereits eine Impurity von 0 vorweisen kann, lässt sich die Unreinheit hier durch kein denkbare Splitting-Kriterium noch weiter verringern. Deshalb wollen wir uns dem linken Node widmen. Wir haben 6 Datenpunkte – 3 rote und 3 blaue – und wir wollen nun das beste Split-Kriterium für diese finden. Dafür müssen wir genauso wie beim Root-Node vorgehen, jeden möglichen Split ausprobieren und den Gini-Gain berechnen. In diesem Fall wird die Prozedur zum Ergebnis haben, dass ein Splitting bei  $y=2$  die optimale Wahl ist. Somit sieht der Decision Tree am Ende folgendermaßen aus:



Wie viele Schritte gehen wir nun insgesamt? Wie oft wollen wir die Daten noch weiter splitten? Wie viele Entscheidungsebenen wollen wir also noch in den Decision Tree einbauen?

Die Antwort ist allgemeiner gehalten: Es hängt davon ab.

Lässt sich die Unreinheit durch einen weiteren Split noch weiter verringern?

Schauen wir uns die beiden Blatt-Nodes ganz unten an: einer mit 3 nur blauen Datenpunkten und einer mit 3 nur roten Datenpunkten. Die Impurity für die beiden liegt bereits bei 0, das bedeutet, jede weitere Entscheidungsebene kann die Unreinheit nicht mehr weiter verringern. Insofern ist der Decision Tree nun fertig trainiert und kann zur Vorhersage und Klassifizierung verwendet werden.



## 2.7.4. Regression Trees

Wir wissen jetzt, dass ein Decision Tree ein Klassifizierungs-Algorithmus ist, und wie er prinzipiell funktioniert. Was also hat es mit Regression Trees auf sich? Wie der Name schon sagt, handelt es sich um eine Methode zur Regression, also zur Vorhersage von numerischen Werten. Wie kann etwas mit einer so diskreten Struktur wie ein Entscheidungsbaum zur Vorhersage numerischer Werte verwendet werden?

Ein Regression Tree ist prinzipiell nicht anders aufgebaut als ein normaler Decision Tree, nur der Aufbau läuft etwas anders.

Erinnern wir uns dafür kurz daran, wie ein Decision Tree aufgebaut wird. Es werden die richtigen Fragen an den richtigen Nodes gestellt, um die Daten immer weiter zu splitten und akkurate Klassifizierungen vorzunehmen. Um die optimalen Splits zu bestimmen, nutzen wir Metriken wie die Gain Ratio oder vor allem die Gini Impurity.

Das macht aber bei einer Regression nicht mehr so viel Sinn, denn für numerische Werte lassen sich die obigen Metriken nicht anwenden. Stattdessen verwendet man bei Regression Trees den mittleren quadratischen Fehler MQF:

$$MQF = \frac{\sum(y_i - f(x_i))^2}{n}$$

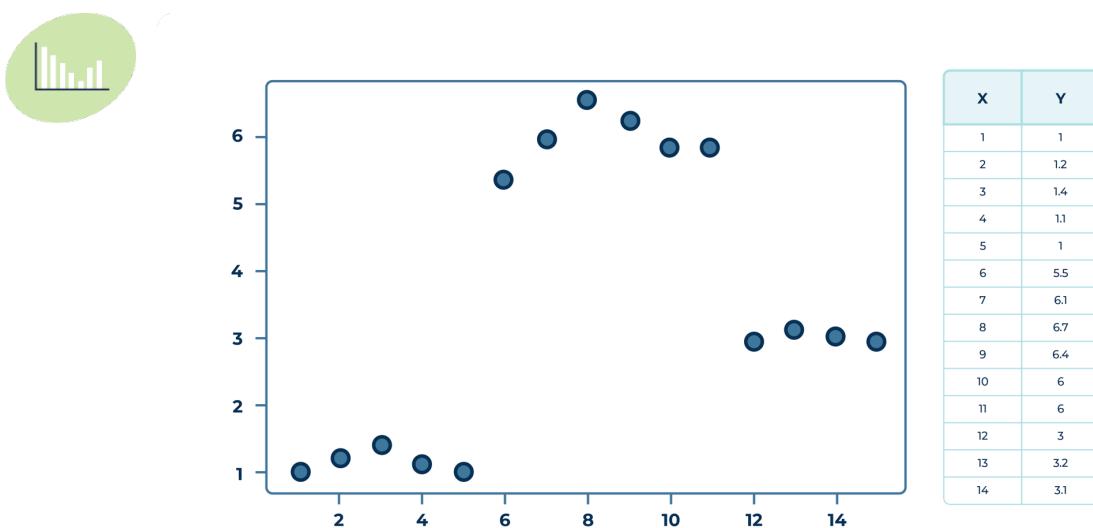
Hierbei ist  $y_i$  der eigentliche Datenwert und  $f(x_i)$  der vom Regression Tree vorhergesagte Wert. Die Vorgehensweise beim Regression Tree ist nun analog zu einem Decision Tree.

An jedem Children Node versucht man, den mittleren quadratischen Fehler zu reduzieren.



Schauen wir uns das Ganze an einem Beispiel an.

Zunächst einmal benötigen wir einen Datensatz; wir nehmen einen einfachen mit 2 Variablen – X und Y. In einem Streudiagramm können wir erkennen, dass die Datenpunkte sich in 3 Cluster unterteilen lassen. Die Datenpunkte in den 3 Clustern haben annähernd ähnliche y-Werte – mit leichten Schwankungen.



Unser Ziel ist es nun, anhand des x-Wertes den y-Wert vorherzusehen. Einen Regression Tree aufzubauen, der diese Aufgabe erfüllt, erfolgt in drei Schritten:

### **Schritt 1:**

Zuerst müssen wir die x-Werte sortieren. Im Fall des Beispiel-Datensatzes sind sie das schon. Dann nehmen wir die ersten beiden x-Werte und bilden den Durchschnitt aus beiden. In unserem Fall ist  $x1=1$  und  $x2=2$ , was zu einem Durchschnitt von 1.5 führt.

Dann werden die Datenpaare ( $x,y$ ) sortiert, und zwar so, dass alle Datenpaare mit  $x < 1.5$  in eine Gruppe mit Namen A kommen und alle mit  $x > 1.5$  in die



andere mit Namen B. Das war der „Tree“-Teil. Jetzt kommt der Regressions-Teil:

Denn in diesen zwei Bereichen A und B wird nun der Durchschnitt der y-Werte berechnet. Dieser Durchschnitt ist dann auch die Prognose, die der Algorithmus für die beiden Bereiche A und B liefert.

Der Durchschnitt in A beispielsweise wäre 1 (es gibt nur ein Datenpaar mit  $x < 1.5$  und das ist (1,1)). Das bedeutet, die Prognose für einen Datenpunkt, der in die Kategorie A fällt, wäre immer 1.

Jetzt haben wir alles, um den mittleren quadratischen Fehler zu berechnen. Wir haben die Prognose (über den Durchschnitt definiert) und den eigentlichen Datenpunkt ( $x, y$ ). Diesen notieren wir.

## Schritt 2:

Wir haben jetzt den MQF für einen möglichen Split der Daten berechnet, aber ist das dann gleich der optimale? Nein, den müssen wir noch finden. Dafür wiederholen wir die Prozedur aus Schritt 1, doch diesmal für den 2. und 3. x-Wert im Datensatz.

Wir berechnen den Durchschnitt aus beiden, was uns 2.5 liefert. Dann teilen wir die Datenpaare wieder in zwei Kategorien A und B: A enthält alle Datenpunkte mit  $x < 2.5$  und B enthält alle Datenpunkte mit  $x > 2.5$ . Für diese beiden Bereiche berechnen wir wieder die durchschnittlichen y-Werte und nutzen diese Prognose zur Berechnung des MQF.

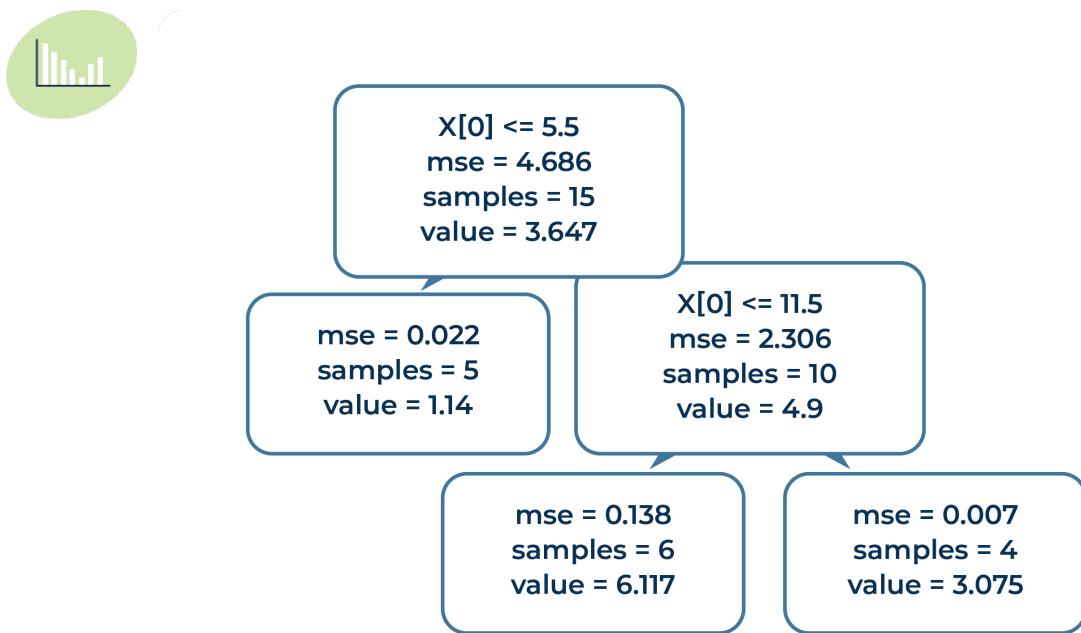
Die Abfolge von Schritten führen wir fort für den 3. und 4. x-Wert, den 4. und 5. x-Wert und so weiter, bis wir  $n-1$  MQF's berechnet haben (wobei  $n$  die Anzahl der Datenpaare im Datensatz ist).



### Schritt 3:

Nun, da wir die MQFs für alle möglichen Splits berechnet haben, wählen wir den Split mit dem geringsten mittleren quadratischen Fehler. Diesen verwenden wir als Entscheidungsgrundlage für den Root Node des Regression Trees. In unserem Fall werden die Datenpaare bei  $x=5.5$  geteilt. Der Teil A wird alle Datenpunkte mit  $x < 5.5$  enthalten, der Teil B alle übrigen.

Beim Regression-Tree-Algorithmus geht es also darum, den Punkt in der unabhängigen Variable zu finden, der den Datensatz so aufteilt, dass der mittlere quadratische Fehler minimal ist. Je mehr Entscheidungsebenen man zu dem Baum hinzufügt, umso mehr passt sich die Entscheidungsstruktur an die Daten an (was schnell zu Overfitting führen kann). Für den obigen Datensatz haben wir erst nur den Root Node bestimmt. Die Unterteilung kann aber noch weiter gehen. Das ist graphisch besonders gut zu erkennen.



Im Root Node sehen wir, dass die Aufteilung bei  $x=5.5$  erfolgt. Die Variable samples gibt hier die Gesamtzahl der Datenpaare an. Die Variable mse gibt

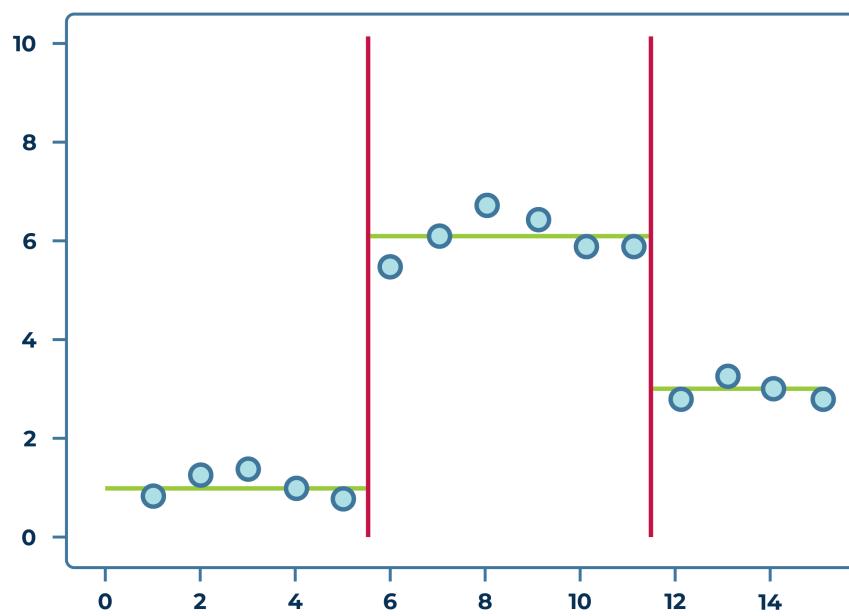


den mittleren quadratischen Fehler an, der in diesem Fall aus dem Durchschnitt der y-Werte resultiert, der in der Variable `value` gespeichert ist. Der linke Node hat bereits einen geringen quadratischen Fehler, weshalb dieser ein Blatt wird, und kein weiteres Splitting stattfindet.

Der rechte Node hat einen MQF von 2.306: Da besteht noch Verbesserungsbedarf. Deswegen teilt sich hier der Baum nochmal in zwei Kategorien auf: die Datenpaare mit  $x < 11.5$  und die Datenpaare mit  $x > 11.5$ .

Dadurch kann der MQF noch einmal erheblich gesenkt werden, was vor allem beim rechten Blatt mit einem Wert von 0.007 sehr gut zu sehen ist.

Graphisch ist die Entscheidungsstruktur des Baumes in folgendem Streudiagramm visualisiert. Die roten vertikalen Linien stehen für die Klassifizierungen des Decision Trees in die drei Kategorien  $x < 5.5$ ,  $x < 11.5$  und  $x > 11.5$ . Die grünen Linien repräsentieren die Durchschnitte der y- Werte und damit die Prognosen des Regression Trees. Wir sehen, dass diese Prognosen gerade im linken und rechten Bereich besonders gut funktionieren, nur in der Mitte gibt es einige Ausreißer.



Wie wir also sehen konnten, lässt sich die diskrete Entscheidungsstruktur eines Decision Trees auch für die Vorhersage von numerischen Werten einsetzen. Das nennt sich dann Regression Tree. Allerdings ist ein Regression Tree sehr anfällig für Overfitting: es kann schnell passieren, dass sich das Modell durch zahlreiche Entscheidungsebenen zu gut an die Daten anpasst und keine treffenden Vorhersagen mehr treffen kann.

## 2.7.5. Programmierung eines Decision Trees

Schauen wir uns nun nach all der Theorie mal an, wie man denn einen Decision Tree in Python am besten umsetzt. Wie auch zuvor werden wir hierbei Gebrauch von der scikit-learn- Bibliothek machen. Diese enthält das Sub-Modul tree, das wiederum die Klasse DecisionTreeClassifier() enthält. Mit dieser können wir ein Classifier-Objekt erzeugen, das einen Decision Tree mit der fit()-Methode trainiert. Gehen wir einfach mal Schritt für Schritt durch, wie wir einen Decision Tree erstellen, visualisieren und nutzen können.



Fangen wir mit den Daten an. Wir stellen uns vor, wir hätten Daten zu Bewerbern bzw. Bewerberinnen: Alter (Age), Berufserfahrung in Jahren (Experience), ihr Rang auf einer Job-Plattform (Rank), die Uni, an der sie studiert haben, (Uni: Oxford, Harvard, ETH) und ob sie eingestellt wurden oder nicht (Hire: yes or no).

Die Daten haben wir als CSV-File; sie sehen in tabellarischer Form folgendermaßen aus:

Age	Experience	Rank	Uni	Hire
36	10	9	Oxford	NO
42	12	4	Harvard	NO
23	4	6	ETH	NO
52	4	4	Harvard	NO
43	21	8	Harvard	YES
44	14	5	Oxford	NO
66	3	7	ETH	YES
35	14	9	Oxford	YES
52	13	7	ETH	YES
35	5	9	ETH	YES
24	3	5	Harvard	NO
18	3	7	Oxford	YES
45	9	9	Oxford	YES

Als erstes müssen wir also die Daten einlesen. Dafür müssen wir natürlich die pandas-Bibliothek importieren. Zusätzlich importieren wir aus der sklearn-Bibliothek das Tree-Modul und anschließend die DecisionTreeClassifier()-Klasse.

```
import pandas
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier

# Einlesen der Daten
df = pandas.read_csv("DecisionTree.csv")
```



Als nächstes müssen wir die Daten vorbereiten. Die Klassen im Feature „Uni“ beispielsweise müssen durch numerische Werte repräsentiert werden. Die Klasse „Oxford“ bekommt als Zahl die 0, „Harvard“ bekommt die 1 und die „ETH“ bekommt die 2. Ähnlich muss es auch bei der „Hire“-Spalte gemacht werden. Das lässt sich mit der map()-Funktion erreichen:

```
# Vorbereiten der Daten

d = {'Oxford':0, 'Harvard':1, 'ETH':2}
df['Uni'] = df['Uni'].map(d)

d = {'YES':1, 'NO':0}
df['Hire'] = df['Hire'].map(d)
```

Am Ende sieht dann das Dataframe mit den neukodierten Spalten so aus:

	Age	Experience	Rank	Uni	Hire
0	36	10	9	0	0
1	42	12	4	1	0
2	23	4	6	2	0
3	52	4	4	1	0
4	43	21	8	1	1
5	44	14	5	0	0
6	66	3	7	2	1
7	35	14	9	0	1
8	52	13	7	2	1
9	35	5	9	2	1
10	24	3	5	1	0
11	18	3	7	0	1
12	45	9	9	0	1

Jetzt müssen wir nur noch für den Algorithmus nachher spezifizieren, welche Spalten die Features sind und welche das Kriterium, also die Target-Spalte. Was wollen wir am Ende eigentlich vorhersagen?

Wir wollen anhand der Daten Alter, Erfahrung, Rank und Uni klassifizieren, ob ein\*e Bewerber\*in eingestellt wird oder nicht.

Das heißt: Alter, Erfahrung, Rank und Uni sind die Features. Und die „Hire“-Spalte ist die Target-Spalte.



Das zeigen wir dem Algorithmus mit den folgenden Zeilen:

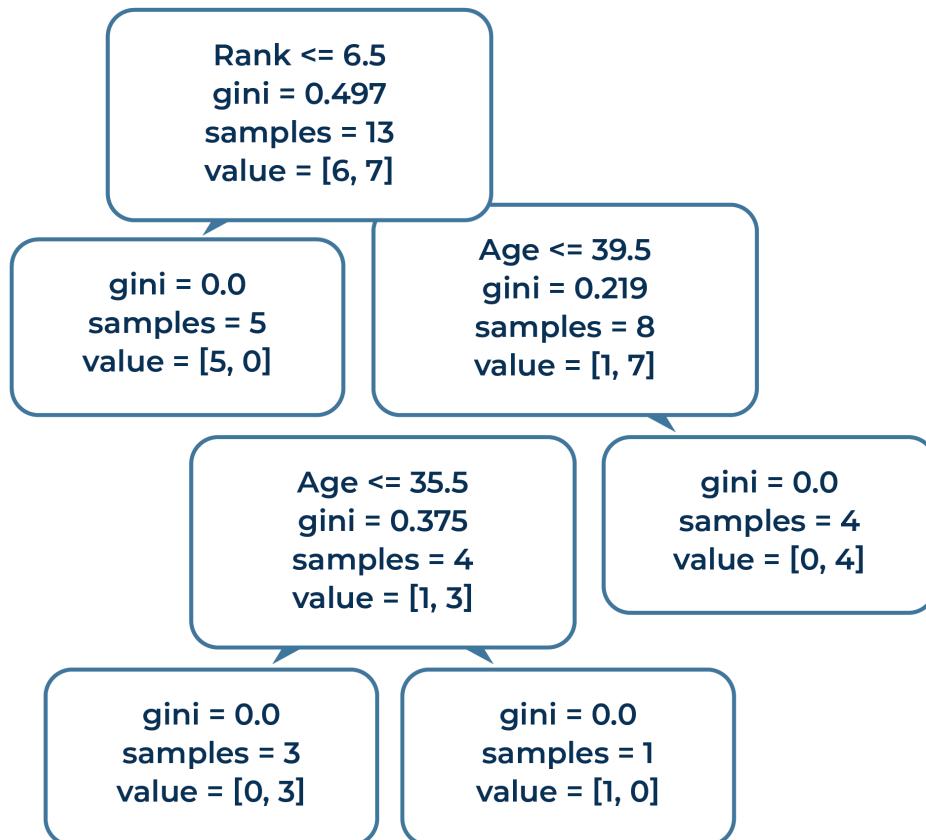
```
# Trennung der Feature Spalten von der Target-Spalte  
features = ['Age', 'Experience', 'Rank', 'Uni']  
  
X = df[features]  
y = df['Hire']
```

Jetzt können wir endlich mit der eigentlichen Decision-Tree-Klassifizierung beginnen. Wir erzeugen mit der `DecisionTreeClassifier()`-Klasse ein Objekt

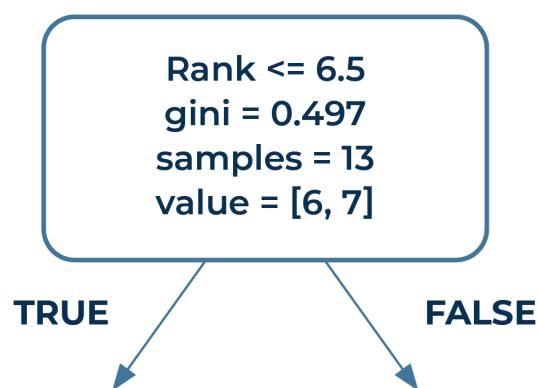
```
# Jetzt kommt der eigentliche Decision Tree Klassifizierer  
dtree = DecisionTreeClassifier()  
dtree.fit(X,y)  
  
tree.plot_tree(dtree, feature_names=features)
```

mit dem Namen `dtree`. Dieses Objekt trainieren wir dann auf die Daten und stellen anschließend den Entscheidungsbaum visuell mit der `plot_tree()`-Funktion dar:

Der Baum, den wir dann am Ende bekommen, ist der Folgende:



Schauen wir uns mal an, was der Inhalt in den Nodes des Decision Trees eigentlich bedeutet. Beginnen wir mit dem Root-Node:





In diesem Root-Node ist der Rank das optimale Splitting-Kriterium. Jede\*r Bewerber\*in mit einem Rank kleiner oder gleich 6.5 wird nach links in Richtung TRUE wandern. Die übrigen gehen nach rechts in Richtung FALSE. Die Variable gini steht für den Wert der Gini Impurity bei diesem Split.

Die Variable samples gibt an, wie viele Datenpunkte an diesem Punkt im Baum gesplittet werden. In diesem Fall sind es noch alle 13 Datenpunkte.

Die Variable value gibt an, wie viele der Samples ein NO und wie viele ein YES als Antwort bekommen. So geht das dann weiter in der zweiten Entscheidungsebene des Decision Trees.

Wie können wir jetzt aber Vorhersagen treffen? Oder neue Datenpunkte klassifizieren? Wie so üblich in der scikit-learn-Bibliothek funktioniert das mit der predict()-Funktion. Klassifizieren wir beispielsweise eine Bewerberin, die 40 Jahre alt ist, 10 Jahre Berufserfahrung, einen Rank von 7 und in Harvard studiert hat. Der Code dafür sieht folgendermaßen aus:

```
print(dtree.predict([[40, 10, 7, 1]]))
```

Das Ergebnis in diesem Fall ist die Klasse 1, also ein JA zur Bewerbung der Kandidatin. Wenn wir jetzt den Rank der Bewerberin auf 6 ändern, erhalten wir eine andere Klassifizierung: 0. Der Rank scheint also das stärkste Kriterium zur Klassifizierung der Bewerber\*innen zu sein.

## 2.7.6. Vor- und Nachteile eines Decision Trees

Nachdem wir Decision Trees nun im Detail kennengelernt haben, befassen wir uns jetzt mit den Vor- und Nachteilen, die mit der Verwendung dieses Algorithmus einhergehen. Fangen wir mit den Vorteilen an.

### Vorteile



Einer der Vorteile eines Decision Trees, die einem sofort ins Auge springen, ist dessen Anschaulichkeit. Die Baumstruktur macht die Entscheidungsfindung bei der Klassifizierung einfach zu verstehen und auch die Hierarchie wird ersichtlich.

Weitere Vorteile zeigen sich bei der Vorbereitung der Daten. Während die meisten Machine-Learning-Algorithmen eine Vorbereitung in Form einer Skalierung der Daten benötigen, ist das beim Decision-Tree-Algorithmus nicht notwendig. Zusätzlich haben fehlende Werte in den Daten keinen Einfluss auf das Training und damit die Konstruktion des Baums. Andere Fehler in den Daten führen allerdings zu Problemen.

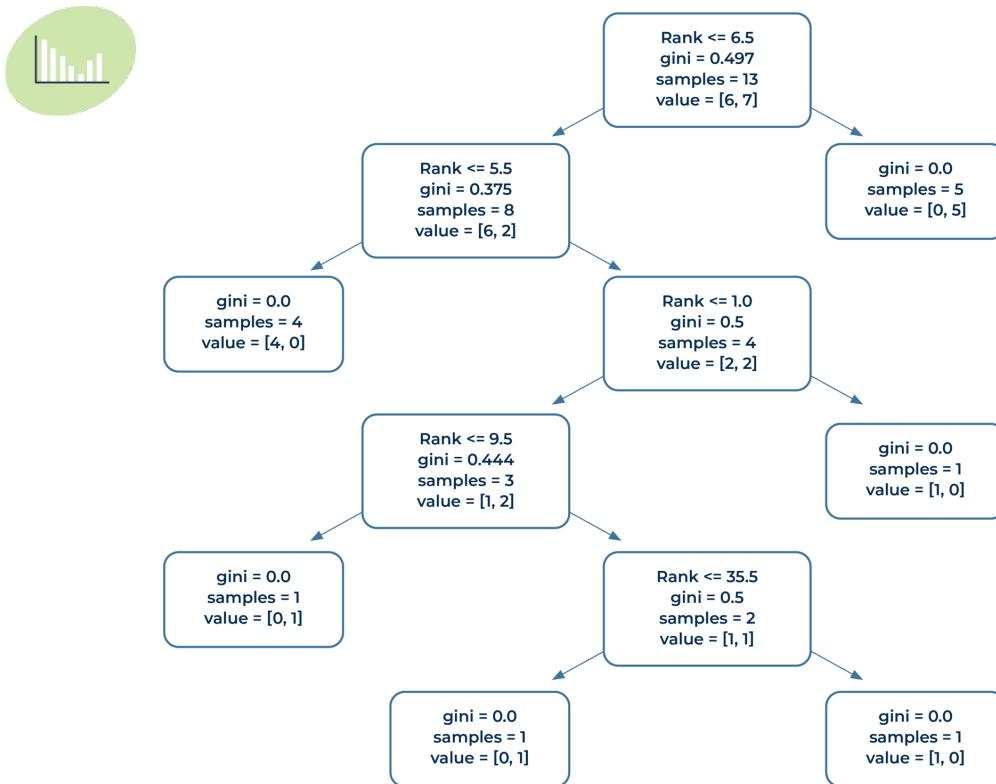
### Nachteile

Einer der größten Nachteile eines Decision Trees ist dessen Instabilität in Bezug auf die Trainingsdaten. Wenn man diese nur minimal ändert, hat dies einen immensen Einfluss auf die Gestalt des Decision Trees. Nehmen wir als Beispiel den Bewerber-Datensatz, den wir zuvor benutzt haben. Wir ändern ihn ein klein wenig:

Age	Experience	Rank	Uni	Hire
36	10	9	Oxford	NO
42	12	4	Harvard	NO
23	4	6	ETH	NO
52	4	4	Harvard	NO
43	21	8	Harvard	YES
44	14	5	Oxford	NO
66	3	7	ETH	YES
35	14	9	Oxford	YES
52	13	7	ETH	YES
35	5	9	ETH	YES
24	3	5	Harvard	NO
18	3	7	Oxford	YES
45	9	9	Oxford	YES



In den farbig markierten Zeilen haben wir lediglich den Rank der Bewerber\*innen von 9 auf 6 heruntergesetzt. Dadurch ändert sich der trainierte Decision Tree folgendermaßen:



Ein weiteres schwerwiegendes Problem beim Einsatz von Decision Trees ist, dass die Berechnung der einzelnen Entscheidungsebenen mit der Zeit sehr komplex wird. Das wiederum bedeutet, dass das Training eines Decision Trees sehr rechen- und zeitintensiv werden kann.

## 2.8. Bagging und Boosting

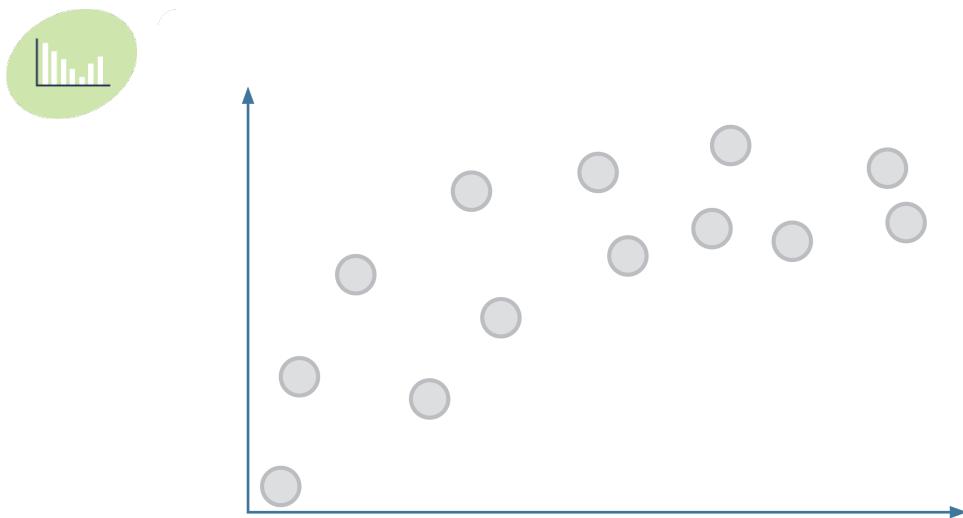
Wie wir im Abschnitt über Decision Trees lernen konnten, reagiert dieser Klassifizierungsalgorithmus sehr sensitiv auf eine Änderung der Trainingsdaten. Kleine Abweichungen können bereits zu einem völlig neuen Entscheidungsbaum und damit auch zu einer neuen Klassifizierungsvorschrift führen. Dieses Problem ist allerdings nicht nur auf Decision Trees beschränkt. Viele Machine-Learning-Algorithmen reagieren



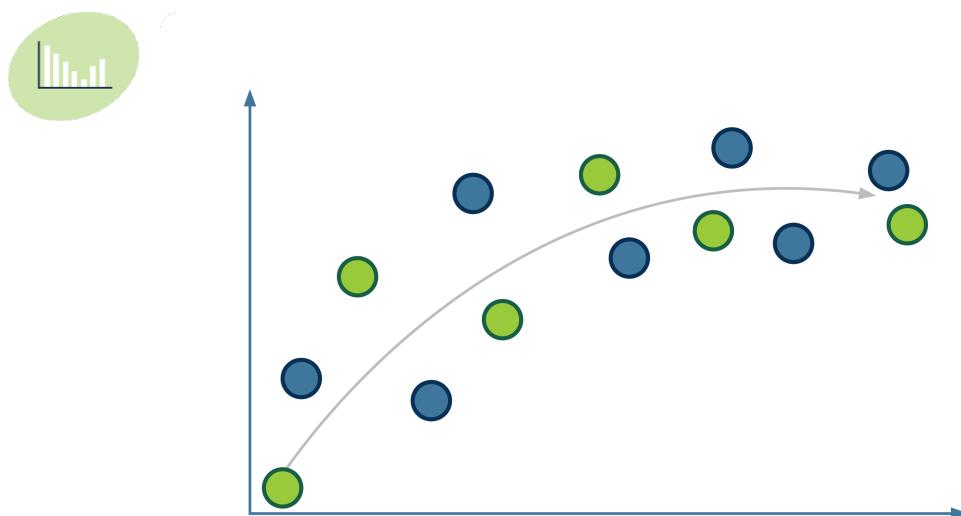
sensibel auf Datenrauschen. Einen solchen Algorithmus nennt man auch „schwachen Lerner“. Um dieses Problem zu lösen, ohne sich gleich einen komplexeren Algorithmus ausdenken zu müssen, verwendet man sogenannte Ensemble-Methoden. Diese vereinen mehrere schwache Lerner zu einem starken Lerner. Die am häufigsten verwendeten Ensemble-Methoden sind Bagging und Boosting, die vor allem gerne im Bereich der Decision Trees eingesetzt werden. Bevor wir uns aber auf diese stürzen können, müssen wir uns mit den Begriffen Bias und Varianz auseinandersetzen – wo wir wieder auf die bekannten Begriffe Overfitting und Underfitting stoßen werden.

### 2.8.1. Bias und Varianz

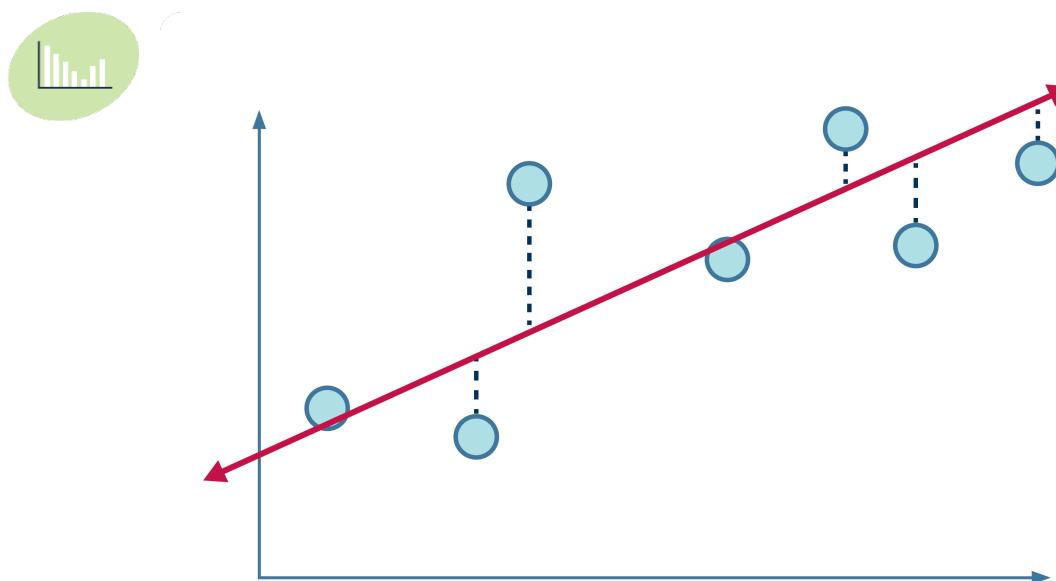
Um die Begriffe Bias und Varianz zu verstehen, fangen wir wie immer mit einem Datensatz an. Schauen wir uns die grauen Datenpunkte an, die eine leichte Krümmung haben.



Nun teilen wir diese Daten in einen Trainings-Datensatz und einen Test-Datensatz ein. Die blauen Punkte sind die Trainingsdaten und die grünen Punkte die Testdaten.

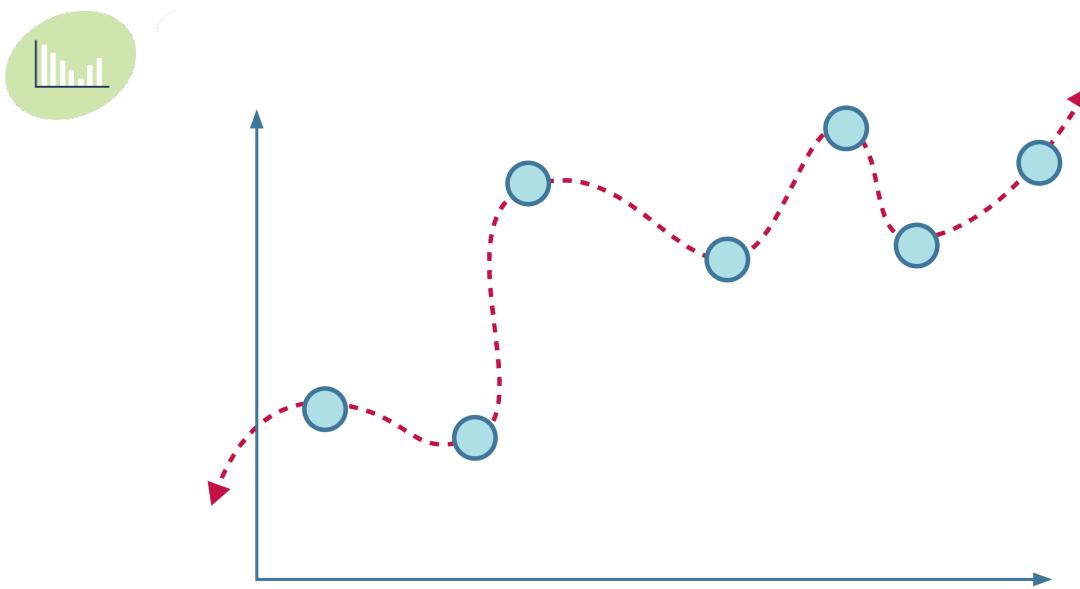


Auf die Testdaten wenden wir jetzt zwei verschiedene Machine-Learning-Methoden an. Zunächst einmal eine simple lineare Regression und dann einen Algorithmus, der sich perfekt an die Testdaten anpasst. Fangen wir mit der linearen Regression an.





Wie wir sehen können, kann eine Gerade – egal wie wir sie im Raum verschieben oder die Steigung ändern – niemals den Bogen in den Daten akkurat beschreiben. Die Gerade wird also niemals die wahre Beziehung zwischen den Features einfangen können. Anders sieht es dagegen aus, wenn wir einen Machine-Learning-Algorithmus anwenden, der eine Schlangenlinie an die Daten anpasst, sodass diese durch jeden Datenpunkt hindurchgeht:



Wie wir an der Grafik sehen können, werden die Testdaten nun perfekt durch das Machine- Learning-Modell beschrieben. Damit haben wir auch schon die Grundlage für die Erklärung des ersten wichtigen Begriffs geschaffen: der Bias.

Wenn man im Machine Learning von Bias spricht, dann meint man die Unfähigkeit eines Machine-Learning-Algorithmus, die wahre Beziehung zwischen den Datenvariablen einzufangen. Je größer der Bias eines Modells ist, umso schlechter wird die Beziehung zwischen den Daten erfasst. Im obigen Beispiel hat die lineare Regression einen sehr hohen Bias, während die Schlangenlinie einen niedrigen hat.

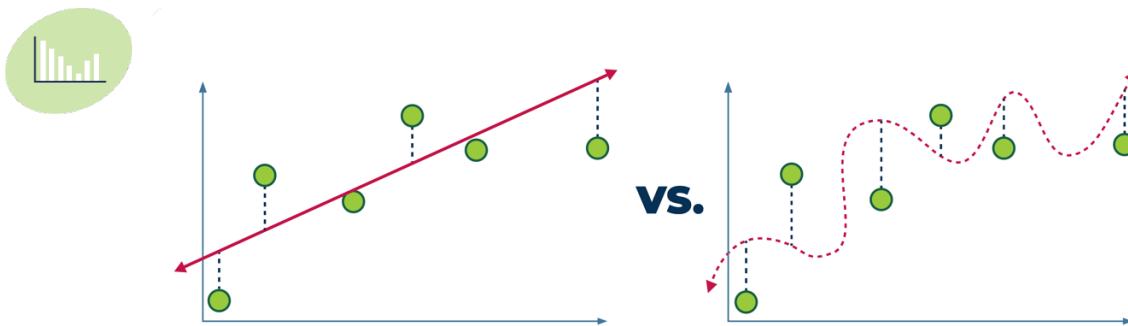


Um genau zu sein, hat die Schlangenlinie überhaupt keinen Bias, weil sie die Testdaten perfekt beschreibt.

Quantitativ ausgedrückt handelt es sich beim Bias um die Differenz der durchschnittlichen Vorhersage eines Modells und dem eigentlichen korrekten Wert, den wir versuchen vorherzusagen. Deshalb hat die Schlangenlinie auch überhaupt keinen Bias, weil der Abstand zwischen vorhergesagten und tatsächlichen Werten 0 ist.

Machine-Learning-Modelle mit einem hohen Bias achten weniger auf die Trainingsdaten und tendieren dazu, die Beziehung zwischen den Variablen zu stark zu vereinfachen. Ein hoher Bias führt immer zu einem großen Fehler sowohl bei den Test- als auch bei den Trainingsdaten.

Jetzt sollten wir uns aber mal anschauen, was passiert, wenn wir beide Modelle auf die Testdaten loslassen. Diesmal nämlich scheint die Gerade die Testdaten besser vorhersagen zu können als die Schlangenlinie!



Auch wenn die Schlangenlinie ein perfektes Modell für die Trainingsdaten war, so weicht sie jetzt mit ihren Vorhersagen stärker von den Testdaten ab, als es die Vorhersagen der linearen Regression tun. Und damit haben wir auch schon die Grundlage für den zweiten wichtigen Begriff: die Varianz.



Mit der Varianz eines Machine-Learning-Modells ist die Differenz der Abweichungen zwischen Trainings- und Testdaten gemeint. Was heißt das nun genau?

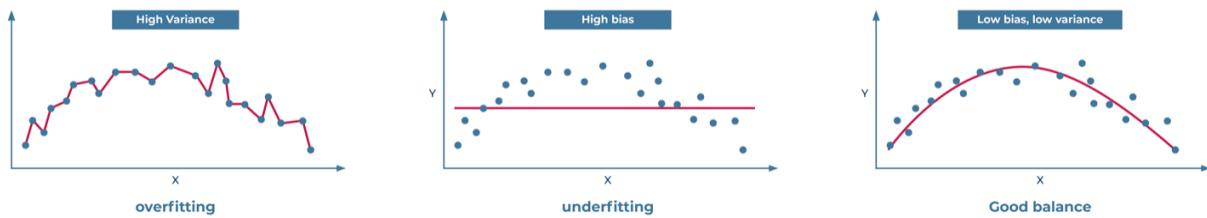
Die lineare Regression weicht zwar mit ihrem Modell von den Trainingsdaten ab, allerdings bewegt sich die Abweichung von den Testdaten in der gleichen Größenordnung. Deshalb ist die Varianz des linearen Regressions-Modells niedrig.

Anders sieht es bei der Schlangenlinie aus. Bei den Trainingsdaten war die Abweichung noch 0, doch dann springen die Abweichungen hoch, wenn das Modell auf die Testdaten angewendet wird. Die Varianz der Schlangenlinie ist also groß und größer als sie bei der linearen Regression ist.

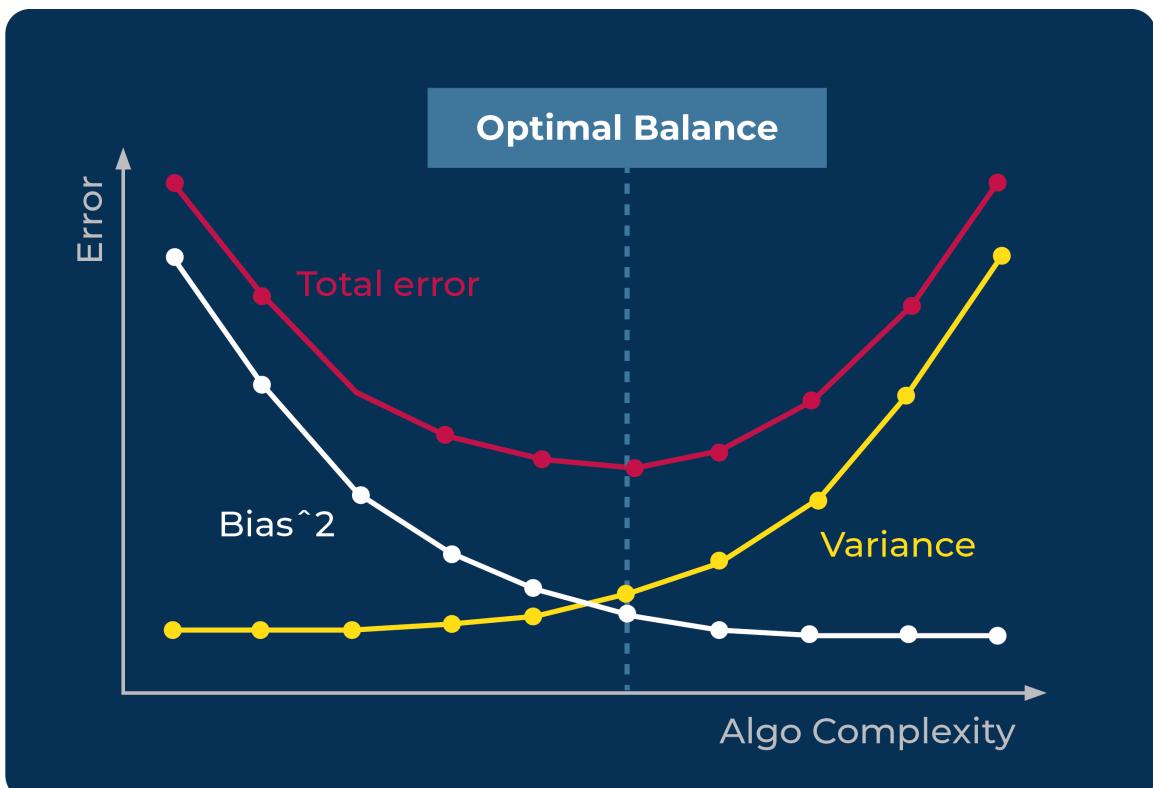
Machine-Learning-Algorithmen mit einer hohen Varianz achten zu stark auf die Trainingsdaten und verallgemeinern den Zusammenhang zwischen den Variablen nicht stark genug. Dadurch funktionieren solche Algorithmen nicht besonders gut bei ihnen völlig unbekannten Daten. Eine hohe Varianz bedeutet immer, dass das Modell einen geringen Fehler bei den Trainingsdaten hat und einen großen Fehler bei den Testdaten.

Durch die Begriffe Bias und Varianz werden auch die Begriffe Overfitting und Underfitting klarer. Ein Modell mit hoher Varianz hat sich an die Trainingsdaten überangepasst.

Ein Modell mit hohem Bias hat sich an die Daten unterangepasst. Ziel ist aber natürlich, ein Machine-Learning-Modell für die Daten zu haben, dass einen niedrigen Bias und eine niedrige Varianz hat.



Dieses Ziel ist allerdings schwierig zu erreichen, da zwei widersprüchliche Bedingungen in Einklang zu bringen sind. Wenn wir ein simples Modell mit wenigen Parametern haben, hat es einen hohen Bias und eine geringe Varianz. Wenn wir allerdings unser Modell zu komplex gestalten und zu viele Parameter verwenden, wird es zwar einen niedrigen Bias, dafür aber eine hohe Varianz haben. Unsere Aufgabe ist es also, ein Gleichgewicht zwischen Über- und Unteranpassung zu finden, sodass der gesamte Fehler des Machine-Learning-Modells minimal wird. Hier kommen dann die Ensemble-Methoden wie Bagging und Boosting ins Spiel.



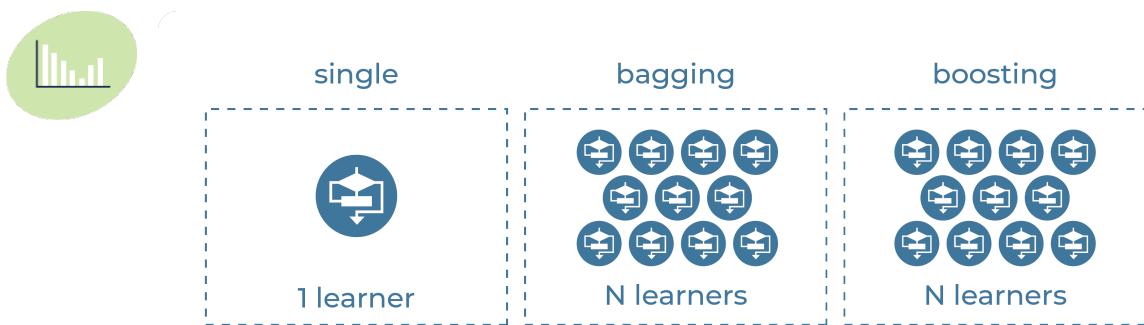


## 2.8.2. Ensemble-Methoden

Wie wir vorher in der Einleitung schon erfahren haben, handelt es sich beim Bagging und Boosting um sogenannte Ensemble-Methoden. Beim Ensemble Learning werden schwache Lernalgorithmen kombiniert und erschaffen gemeinsam einen starken Lernalgorithmus, der Probleme wie Under- und Overfitting sowie Datenrauschen unterdrücken kann. Der wesentliche Unterschied zwischen Bagging und Boosting liegt in der Anwendung.

Bagging hilft dabei, die Varianz eines Modells zu senken, also ein Overfitting zu vermeiden.

Boosting hilft dabei, den Bias eines Modells zu senken, also ein Underfitting der Trainingsdaten zu vermeiden. Bevor wir uns im Detail anschauen können, wie Bagging und Boosting genau funktionieren, schauen wir uns noch kurz das Konzept des Bootstrapping an.

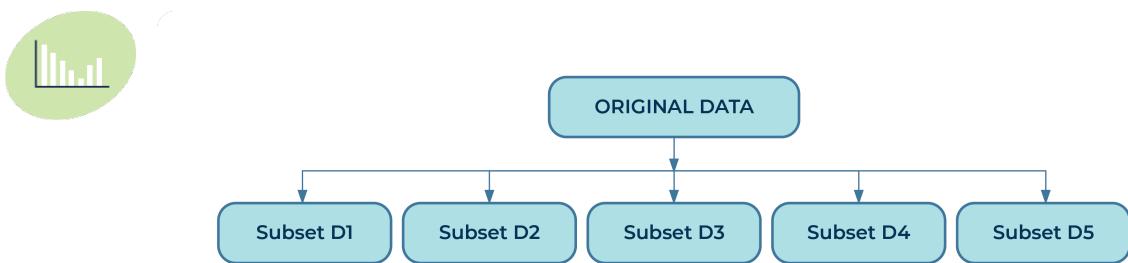


### Bootstrapping

Beim Bootstrapping geht es darum, die Daten auf den Bagging- und Boosting-Prozess vorzubereiten. Hierbei wählt man zunächst eine Teilmenge des originalen Datensatzes, die kleiner als dieser sein kann. Beispielsweise wählen wir aus einem Datensatz mit 100 Einträgen 10 aus.



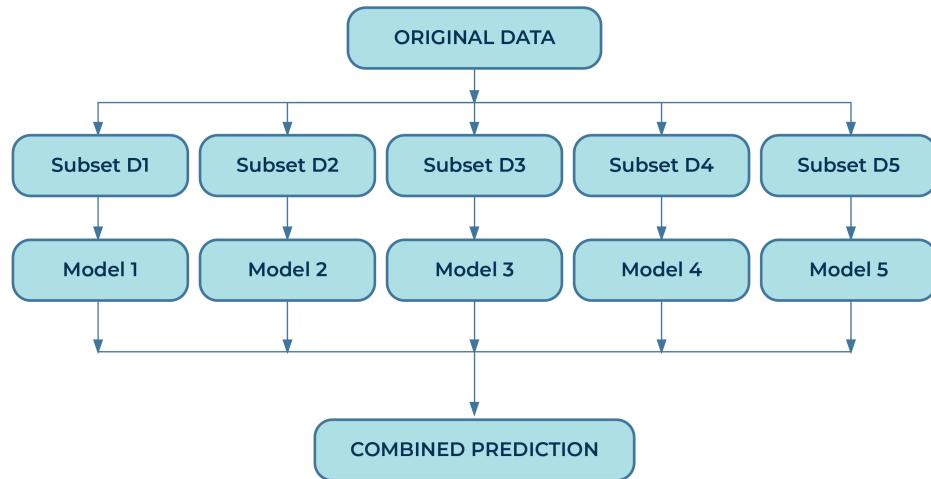
Um dann wieder einen Datensatz mit 100 Einträgen zu erhalten, wählen wir die restlichen 90 Einträge zufällig aus dem originalen Datensatz aus. Dadurch entsteht eine neue Mischung aus Daten. Diesen Prozess führen wir mehrere Male durch, bis wir mehrere unterschiedliche Datensätze haben, die aus dem originalen entstanden sind und für das Training verwendet werden können.



## Bagging

Die Idee beim Bagging ist, mehrere schwache Lerner parallel laufen zu lassen (beispielsweise mehrere Decision Trees) und anschließend die Ergebnisse zu kombinieren, um ein allgemeineres und vorhersagekräftigeres Modell zu erhalten. Hierfür verwenden wir die Bootstrapping-Methode.

Durch das Bootstrapping erzeugen wir mehrere ähnliche Datensätze, mit denen wir die schwachen Lerner parallel trainieren. Dabei sind die herauskommenden Modelle alle unabhängig voneinander. Das finale Modell erhält man als Kombination der schwachen Lerner. Im Fall von Decision Trees nennt man die Menge an schwachen Modellen einen Random Forest.



## Boosting

Anders als beim Bagging handelt es sich beim Boosting nicht um einen parallelen, sondern einen sequentiellen Prozess. Hierbei möchte man bei jedem Boosting-Schritt die Fehler des vorherigen Modells ausmerzen. Jedes Modell baut also auf dem vorherigen auf und ist von diesem abhängig. Schauen wir uns mal an, wie genau das funktioniert.

Zuerst gibt man jedem Datenpunkt im Datensatz das gleiche Gewicht. Dann wählt man wieder eine Teilmenge aus dem Datensatz aus und trainiert den Algorithmus mit diesen Daten. Das dabei entstehende Modell wird verwendet, um Vorhersagen für die Daten aus dem ursprünglichen Datensatz zu treffen.

Die dabei beobachteten Abweichungen zwischen vorhergesagten Werten durch das Modell und den tatsächlichen Datenwerten nutzt man, um die Gewichte der Datenpunkte anzupassen.

Datenpunkte, für die die Fehler groß waren, bekommen eine höhere Gewichtung als die anderen.

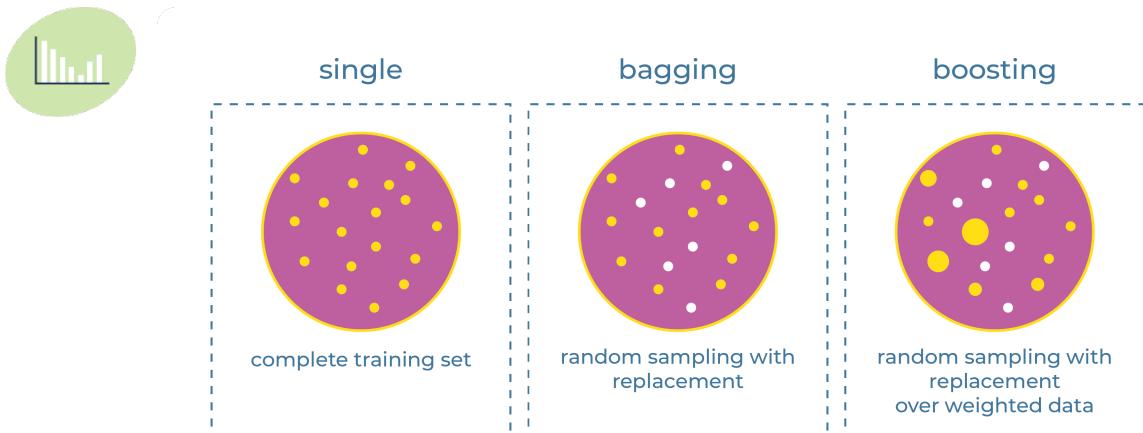


Nun trainiert man den Algorithmus auf dem angepassten Datensatz mit den gewichteten Datenpunkten und erzeugt ein neues Modell. Dieses überprüft man wieder mit dem originalen Datensatz, berechnet die Abweichungen und passt die Gewichtung der Datenpunkte an.

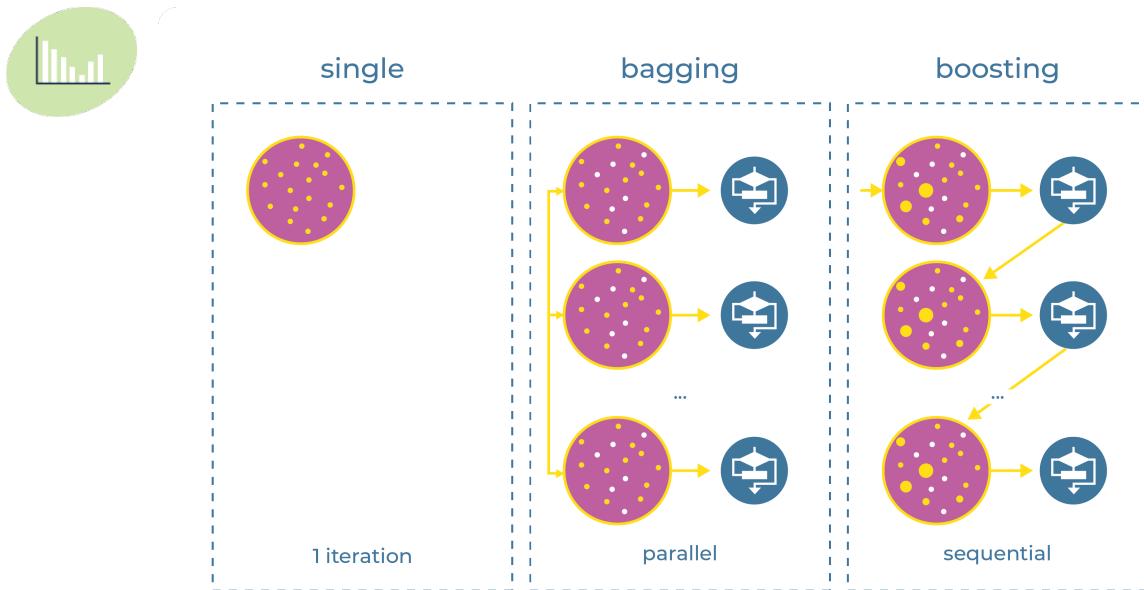
Durch Fortführung dieser Abfolge von Schritten minimiert man den Fehler immer weiter, wobei man einzelne schwache Lerner sukzessiv zu einem starken Lerner aufbaut.

### Unterschiede von Bagging und Boosting

Zusammengefasst verwendet man also beim Bagging eine zufällige Aufteilung der Daten und beim Boosting eine zufällige und gewichtete Aufteilung der Daten.



Beim Bagging werden die schwachen Lernalgorithmen parallel trainiert und sind unabhängig voneinander. Beim Boosting werden die schwachen Lerner nacheinander trainiert und bauen aufeinander auf.



Beim Bagging werden die Trainings-Datensätze zufällig zusammengestellt, während beim Boosting jeder neue Datensatz sein Augenmerk vor allem auf die Elemente richtet, die im vorherigen Schritt nicht korrekt klassifiziert worden sind.

### 2.8.3. Programmierung eines Random Forest

Wie wir vorher kurz erwähnt haben, erhält man einen sogenannten Random Forest, wenn man Decision Trees mit der Bagging-Methode kombiniert. Ein Random Forest lässt sich recht simpel mit der scikit-learn-Bibliothek implementieren.

Im Allgemeinen nutzt man für Ensemble-Methoden das `sklearn.ensemble`-Modul. Dieses enthält die `RandomForestClassifier()`-Klasse. Die wichtigsten Parameter dieser Klasse sind:

- **n\_estimators: int, default=100**



Hiermit ist die Anzahl der Decision Trees gemeint, die der Random Forest haben soll. Per Default wird ein Random Forest mit 100 Decision Trees erstellt.

- **criterion{"gini", "entropy", "log\_loss"}, default="gini"**

Hier legt man fest, welches Kriterium man zum Splitten der Features in den Decision Trees verwenden möchte. Per Default ist das auf die Gini Impurity gesetzt, man kann aber auch mit der Entropie oder dem logarithmischen Verlust arbeiten.

- **bootstrap: bool, default=True**

Dieser Parameter hat einen Booleschen Wert und ist per Default auf TRUE gesetzt. Hier legt man fest, ob man die Daten per Bootstrapping aufteilen möchte. Wenn der Wert auf FALSE gesetzt wird, wird der ursprüngliche Datensatz verwendet.

- **n\_jobs: int, default=None**

Hier legt man fest, wie viele Decision Trees gleichzeitig trainiert werden sollen.

Trainiert wird ein RandomForestClassifier()-Objekt mit der fit(X,y)-Funktion, wobei mit X die Trainingsdaten und mit y die entsprechenden Labels gemeint sind. Mit der predict()-Funktion lässt sich dann ein unbekannter Datensatz klassifizieren.

## 2.9. K-Means Clustering

In diesem Abschnitt verabschieden wir uns von den Supervised-Machine-Learning-Algorithmen, widmen uns den Unsupervised-Learning-Methoden und lernen, wie der K-Means-Clustering- Algorithmus funktioniert. Zunächst lernen wir, was es mit Clustering-Algorithmen auf sich hat und schließlich, wie genau der K-Means-Algorithmus funktioniert. Da man diesem von Beginn an vorgeben muss, wie viele Cluster er in einem



Datensatz erkennen soll, werden wir uns mit Methoden befassen, um diese optimale Cluster-Anzahl zu bestimmen. Die gängigsten Methoden, die in der Praxis einander ergänzend eingesetzt werden, sind die Elbow-Methode und der Silhouetten-Koeffizient. Zum Schluss vertiefen wir das Gelernte mit einer Programmieraufgabe.

## **2.9.1. Unsupervised-Learning-und-Clustering-Algorithmen**

Bis jetzt haben wir uns ausschließlich mit Algorithmen aus dem Bereich des Supervised Learning auseinandergesetzt. Sowohl Regressions- als auch Klassifizierungs-Algorithmen benötigen einen Datensatz zum Training. Ohne Training ist es einem Supervised-Learning-Algorithmus nicht möglich, in einem Datensatz einen Sinn zu erkennen und richtige Vorhersagen für zukünftige Daten zu treffen.

Anders ist das beim Unsupervised Learning. Algorithmen von dieser Sorte können selbstständig Muster und Informationen in den Daten entdecken, die vorher noch gar nicht bekannt waren. Dadurch wird es einem Data Scientist möglich, sich mit komplexeren Problemstellungen auseinanderzusetzen als es mit Supervised-Learning-Methoden möglich gewesen wäre.

Ein alltägliches Beispiel, um den Unterschied zwischen Supervised und Unsupervised Learning anschaulicher zu machen, ist der Lernprozess eines Babys.

Nehmen wir an, ein Baby sieht einen Hund zum ersten Mal und speichert bestimmte Merkmale wie die Ohren, die Pfoten oder die Schnauze ab. Eine Woche später begegnet es wieder einem Hund und erkennt die Ähnlichkeit zu der Erfahrung der vorherigen Woche. Also ordnet das Baby den Hund der gleichen Kategorie zu wie den Hund, den es in der Woche zuvor gesehen hat. Das ist Unsupervised Learning. Würden wir dem Baby jedes Mal sagen,



dass es sich um einen Hund handelt, dann würden wir das Baby trainieren, indem wir die Daten „labeln“. Durch dieses Labeln entsteht ein Supervised-Learning-Prozess.

Das bedeutet, dass man mit Unsupervised-Learning-Methoden Datensätze analysieren kann, die keinerlei Labels besitzen.

Zu den gängigen Formen von Unsupervised-Learning-Algorithmen gehören Clustering- Algorithmen und neuronale Netze. Wobei an dieser Stelle gesagt werden muss, dass neuronale Netze sowohl zum Supervised als auch zum Unsupervised Learning eingesetzt werden können. Damit werden wir uns aber erst im nächsten Kapitel beschäftigen.

Clustering-Methoden finden in zahlreichen Geschäftsbereichen eine Anwendung.

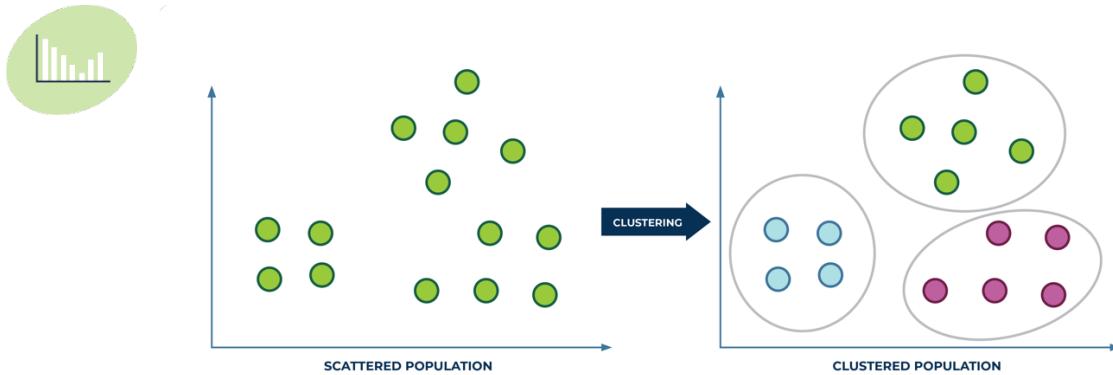
Im Marketing werden sie genutzt, um in bereits existierenden Kundendaten verschiedene Gruppen zu identifizieren, die bis dahin noch unerkannt blieben. Dadurch können die Marketingmaßnahmen auf die jeweilige Gruppe maßgeschneidert werden, was maßgeblich zu einem Wachstum der Conversion Rate beitragen kann.

Ein weiterer Anwendungsbereich liegt bei Buchhandlungen und Büchereien. Die steigende Anzahl an Büchern in zahlreichen Sprachen macht es schwierig, sie zu verkaufen. Vieles geht unter, weil die Genre-Zugehörigkeit nicht erkannt wurde. Buchhandlungen können durch Clustering einen besseren Überblick über ihren Bestand bekommen und so auch Bestellungs- und Liefervorgänge optimieren.

Clustering-Algorithmen können auch dazu verwendet werden, um Daten überhaupt erst einmal zu labeln und damit für einen Supervised-Learning-Prozess vorzubereiten. Beispielsweise kann man Textdokumente vom



Clustering-Algorithmus in Gruppen aufteilen, ohne dass man sie händisch labeln muss.



In diesem Abschnitt werden wir uns im Speziellen mit dem K-Means-Clustering-Algorithmus beschäftigen.

## 2.9.2. K-Means-Clustering

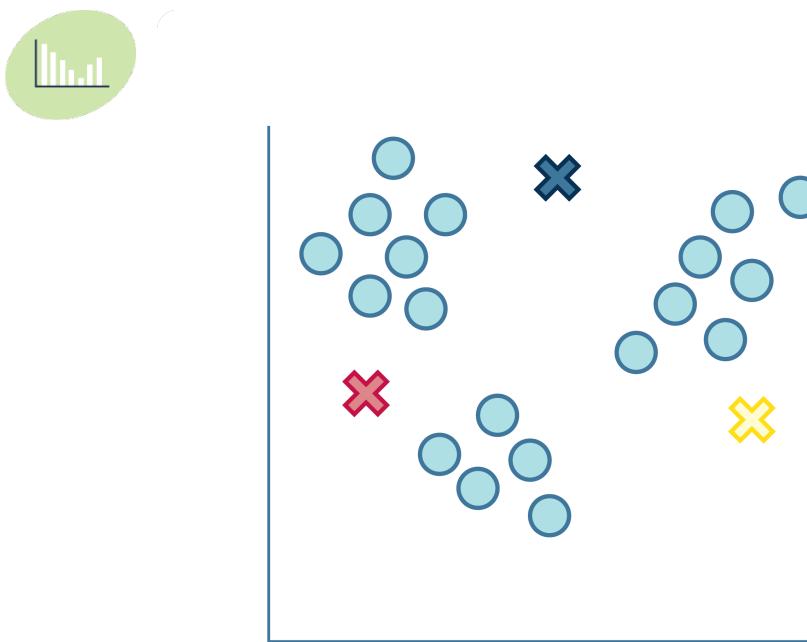
Die Idee für den K-Means-Clustering-Algorithmus schwirrt schon seit 1957 herum und stammt ursprünglich von Hugo Steinhaus. Veröffentlicht wurde er allerdings erst 1982. Heute ist der K-Means-Algorithmus mittlerweile einer der am häufigsten verwendeten Clustering-Algorithmen, vor allem weil er sich aufgrund seiner effizienten Berechnungsmethoden und seines geringen Speicherbedarfs hervorragend für die Analyse von großen Datenmengen (Big Data) eignet. Seine Laufzeit wächst linear mit der Anzahl der Datenpunkte, und der Algorithmus besteht nur aus einer einfachen Abfolge von Schritten. Schauen wir uns diese mal im Detail an.

Nehmen wir dafür obige Grafik links, ein 2-dimensionales Streudiagramm mit Datenpunkten. Der K-Means-Algorithmus teilt die Daten folgendermaßen in Cluster ein:

1. Wahl von K Punkten als Anfangszentren der Berechnung:

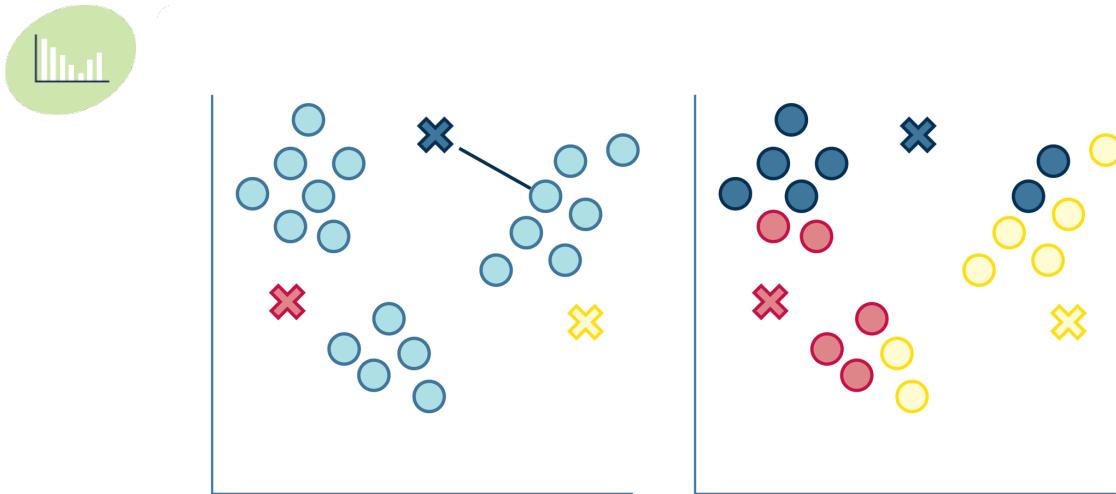


Damit wird auch klar, wofür der Buchstabe K im K-Means-Algorithmus steht. Zu Beginn müssen wir dem Algorithmus sagen, wie viele Cluster er in den Datenpunkten erkennen soll. Dann wählt er willkürlich K Punkte im Raum aus, die er als Zentren der Cluster definiert.



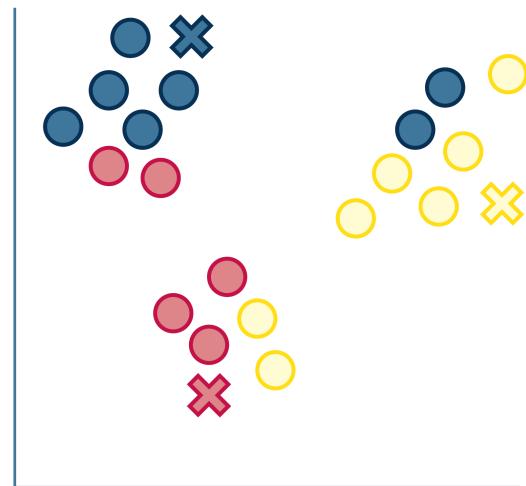
## 2. Zuordnung der Datenpunkte:

Nun werden die einzelnen Datenpunkte basierend auf ihrem Abstand zu den willkürlich in Schritt 1 gewählten Zentren zu den jeweiligen Clustern zugewiesen. Das bedeutet, der Algorithmus misst für jeden Datenpunkt den Abstand zu den K Zentren und wählt dann das Zentrum mit dem geringsten Abstand als Zuordnung aus. Als Abstandsmetrik bietet sich hier wieder die bekannte euklidische Metrik an.



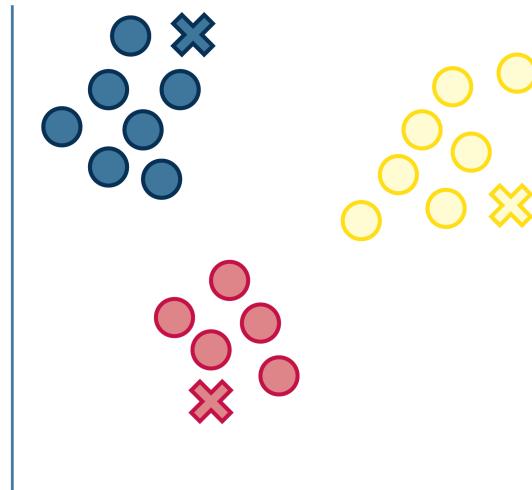
### 3. Neuberechnung der Zentren:

Nun, da die Datenpunkte jeweils einem der zufällig ausgewählten Zentren zugeordnet wurden, beginnt der Algorithmus, die Zentren der Cluster neu zu berechnen. Dafür wählt er als neues Zentrum für die blauen Datenpunkte den Durchschnitt aus allen blauen Datenpunkten. Das macht er für jede Farbe. Nachdem für jeden Cluster die Zentren neu berechnet wurden, werden die Datenpunkte wieder einzeln basierend auf ihrem Abstand einem der Cluster zugeordnet.



#### 4. Wiederholung:

Der Algorithmus wiederholt die Neuberechnung und erneute Zuordnung der Datenpunkte zu den neuen Clustern immer und immer wieder, bis sich die Position der Zentren nicht mehr ändert. In den ersten Schleifen-Durchläufen werden sich die Cluster-Zentren noch sehr stark verschieben, doch mit jeder Wiederholung werden die Veränderungen immer kleiner.



Einer der wichtigsten Faktoren für einen effizienten Ablauf des Algorithmus ist die Wahl der Anfangszentren. Wenn diese optimal gewählt werden, reduziert sich dadurch die Anzahl der Wiederholungen und damit dann auch die Laufzeit.

Im klassischen K-Means-Algorithmus werden die Cluster-Zentren zu Beginn zufällig gewählt. So kann es passieren, dass einer der Punkte viel zu weit weg von den Daten gewählt wird, wodurch die Neuberechnung des richtigen Zentrums verlangsamt wird.

Eine Erweiterung des K-Means-Algorithmus ist der K-Means++-Algorithmus, welcher 2007 von David Arthur und Sergei Vassilvitskii vorgeschlagen wurde, um die Startwerte für die Cluster- Zentren besser als mit dem Zufallsprinzip wählen zu können. Durch die Anwendung des K-Means++ kann der Algorithmus beinahe doppelt so schnell den Cluster-Prozess durchführen als bei einer zufälligen Wahl der Anfangszentren.



### 2.9.3. Anwendungen des K-Means-Algorithmus

Wie wir in der Einführung bereits gesehen haben, wird der K-Means-Algorithmus aufgrund seines geringen Speicherbedarfs gerne im Big-Data-Umfeld verwendet. Aber in welchem Bereich lässt sich der K-Means-Algorithmus nun konkret anwenden?

Der Klassiker wäre hier im Sales und Marketing bei der Analyse von Kundendaten. Durch das effiziente Clustern von riesigen Datenmengen, die von den Kunden und Kundinnen generiert werden (Uhrzeit des Kaufes, Verweildauer des Produkts im Warenkorb, verwandte Produkte, ... – die Liste könnte man ewig fortführen), lassen sich Marketingmaßnahmen auf die jeweilige Kundengruppe spezifizieren.

Ein weiterer Anwendungsfall für den K-Means-Algorithmus findet sich aber auch in der Bildverarbeitung. Hier wird der Algorithmus gerne zur Segmentierung von Bilddaten eingesetzt, womit beispielsweise die Trennung von Farben oder von Vorder- und Hintergrund gemeint ist. Dadurch wird es unter anderem möglich, Objekte in Bildern zu erkennen.

### 2.9.4. Programmierung des K-Means-Algorithmus

Wie bei den anderen Machine-Learning-Algorithmen auch lässt sich der K-Means mit Hilfe der scikit-learn-Bibliothek implementieren. Dafür verwenden wir das sklearn.cluster-Modul und importieren daraus die KMeans-Klasse.

```
from sklearn.cluster import KMeans
```

Als nächstes erzeugen wir zwei Listen mit zufällig gewählten Zahlen. Diese beiden Listen stellen unseren Datensatz (mit zwei Features) dar und werden



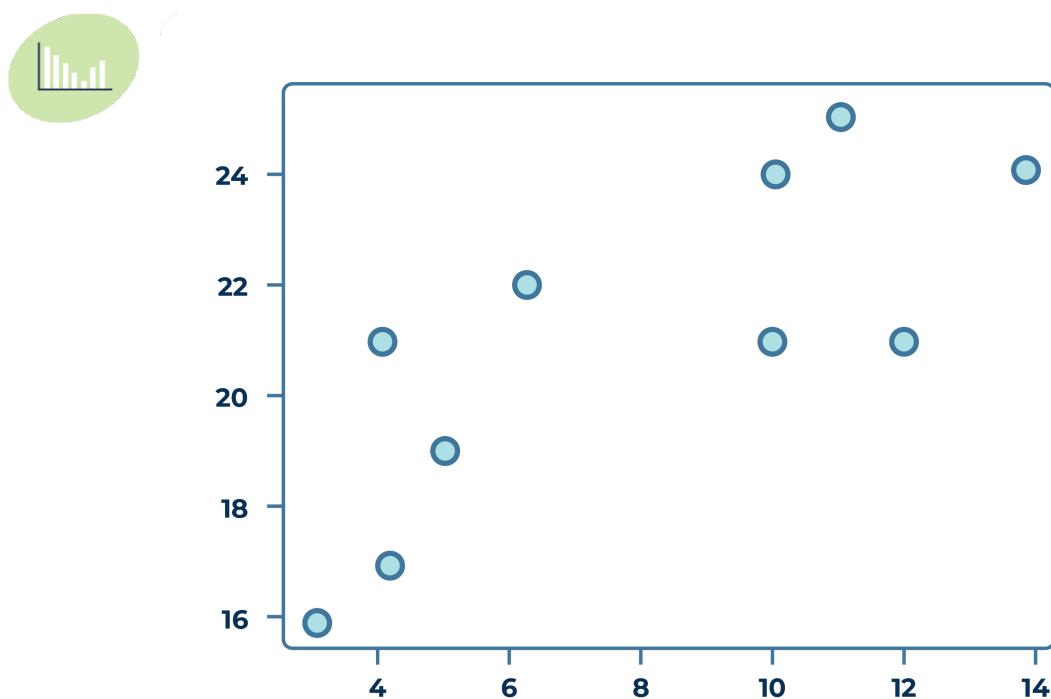
in einem Streudiagramm dargestellt. Der vollständige Code sieht folgendermaßen aus:

```
from sklearn.cluster import KMeans

# Daten visualisieren und vorbereiten
x = [4, 5, 10, 4, 3, 11, 14, 6, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]

plt.scatter(x,y)
plt.show()
```

Und das Streudiagramm mit den Daten:



Durch bloßes Betrachten der Verteilung der Datenpunkte im obigen Streudiagramm könnten wir schon annehmen, dass sich die Daten in zwei Gruppen einteilen lassen. Wir könnten also einen K-Means-Algorithmus mit



K=2 laufen lassen. Damit die KMeans-Klasse mit unserem Datensatz arbeiten kann, müssen wir diesen allerdings noch etwas bearbeiten.

## 2.9.5. Die richtige Anzahl an Clustern bestimmen

Es gibt im Prinzip zwei Methoden, um die richtige Anzahl an Clustern zu bestimmen. Die beiden Methoden konkurrieren nicht, sondern werden in der Praxis ergänzend zueinander eingesetzt. Es handelt sich bei den beiden Methoden um

- die Elbow-Methode und
- den Silhouetten-Koeffizient.

Schauen wir uns die beiden Methoden mal im Detail an.

### Die Elbow-Methode

Bei der Elbow-Methode beginnt man damit, den K-Means-Algorithmus für verschiedene K- Werte auszuführen. Für jeden K-Wert berechnet man den sogenannten Inertia-Wert und hält diesen fest. Inertia ist kurz gesagt ein Maß dafür, wie sehr die Daten innerlich kohärent sind. Mathematisch ist das Inertia als die Summe über die quadratischen Abstände der Datenpunkte zu den Cluster-Zentren definiert. Das war jetzt kompliziert formuliert, deshalb hier noch einmal als Formel ausgeschrieben:

$$\sum_{i=0}^n \min_{\mu_j \in C} (\|x_i - \mu_j\|^2)$$

Natürlich möchte man jetzt diese Summe der quadratischen Abstände möglichst kleinhalten. Wir möchten also die Anzahl an Clustern bestimmen, für die der Inertia-Wert am kleinsten ist. Wir führen also den K-Means-Algorithmus für verschiedene K-Werte durch und halten jedes Mal das



Inertia fest. Das erreichen wir am einfachsten durch die Verwendung einer for-Schleife.

## Der Silhouetten-Koeffizient

Nicht konkurrierend, sondern ergänzend zu der Elbow- Methode bestimmt man in der Praxis den sogenannten Silhouetten-Koeffizienten. Vereinfacht gesagt quantifiziert diese Zahl, wie gut ein Datenpunkt in den Cluster passt, dem er zugeordnet wurde. Diese Quantifizierung geschieht in Abhängigkeit von zwei Faktoren:

- Wie nah der betrachtete Datenpunkt den anderen Datenpunkten im Cluster ist
- Wie weit der betrachtete Datenpunkt von Datenpunkten aus anderen Clustern entfernt ist

Dabei nimmt der Silhouetten-Koeffizient Werte zwischen -1 und 1 an. Große Werte (nahe 1) bedeuten, dass der betrachtete Datenpunkt den Punkten seines Clusters näher ist als den Punkten der anderen Cluster. Berechnen lässt sich der Silhouetten-Koeffizient mit der folgenden Formel:

$$s = \frac{b - a}{\max(a, b)}$$

Hierbei ist a der durchschnittliche Abstand zwischen dem betrachteten Datenpunkt und allen Datenpunkten im eigenen Cluster.

Die Variable b steht für den durchschnittlichen Abstand des betrachteten Datenpunktes zu allen Punkten in den anderen Clustern.

Möchte man den Silhouetten-Koeffizient für eine ganze Menge von Datenpunkten bestimmen, berechnet man den Durchschnitt der Silhouetten-Koeffizienten der einzelnen Datenpunkte.



Diesen Gesamt-Silhouetten-Koeffizienten kann man sich mit der silhouette\_score()-Funktion ausgeben lassen. Dafür muss man diese erst aus dem sklearn.metrics-Modul importieren:

```
from sklearn.metrics import silhouette_score
```

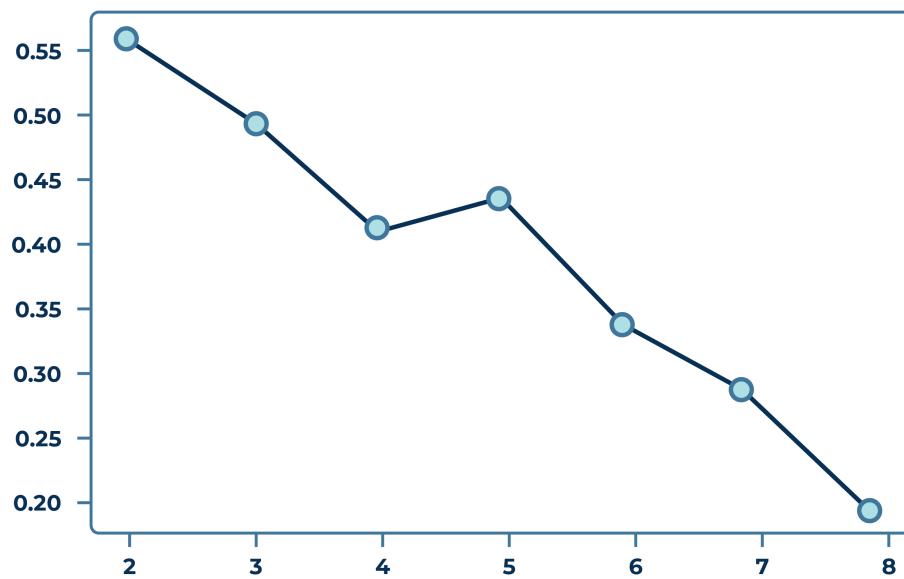
Die Argumente für die silhouette\_score()-Funktion sind die Daten und die Labels, die die Daten nach einer K-Means-Clustering-Analyse bekommen haben. Wir wollen jetzt – analog wie bei der Elbow-Methode – den K-Means-Algorithmus für verschiedene K-Werte laufen lassen und für jeden Wert den Gesamt-Silhouetten-Koeffizient berechnen. Die Koeffizienten tragen wir dann wieder auf der y-Achse gegen die Anzahl der Cluster auf der x-Achse auf.

```
for i in range(2,9):

    km = KMeans(n_clusters=i)
    km.fit(data)
    score = silhouette_score(data, km.labels_)
    silhouette_coefficients.append(score)

pass

plt.plot(range(2,9),silhouette_coefficients,marker='o')
```



Anhand des Plots können wir erkennen, dass der Silhouetten-Koeffizient für zwei Cluster am größten ist, was bedeutet, dass dies die optimale Cluster-Anzahl ist. Wir sehen also, dass die Berechnung des Silhouetten-Koeffizienten unsere Analyse mit der Elbow-Methode bestätigt hat.

## 2.10. Hierarchisches Clustering

Der nächste Unsupervised-Algorithmus, mit dem wir uns jetzt befassen, ist der sogenannte hierarchische Clustering-Algorithmus. Diese Art von Clustering-Algorithmus lässt sich in zwei Kategorien einteilen:

- Agglomeratives Clustering
- Divisive Clustering

Vereinfacht gesagt handelt es sich beim agglomerativen Clustering um den sogenannten Bottom-Up-Ansatz und beim Divisive Clustering um den Top-Down-Ansatz. In der Praxis wird hauptsächlich der erste der beiden Algorithmen eingesetzt. Schauen wir uns dennoch die Funktionsweise von



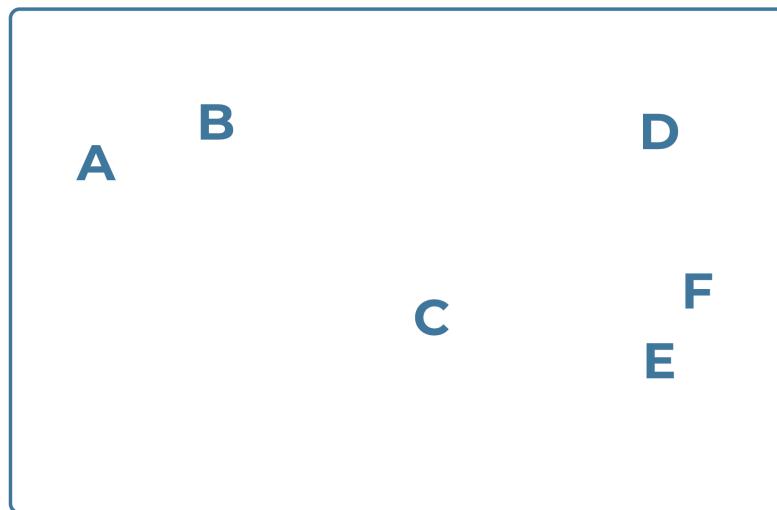
beiden Algorithmen an (wobei der Fokus auf dem agglomerativen Clustering liegen wird).

### **2.10.1. Agglomeratives Clustering**

Wie wir oben bereits erwähnt haben, handelt es sich bei dieser Art des hierarchischen Clustering um den Bottom-Up-Ansatz. Man fängt bei den einzelnen Datenpunkten eines Datensatzes an und betrachtet jeden einzelnen von ihnen als separaten Cluster. Dann werden im ersten Schritt die beiden Datenpunkte zu einem Cluster zusammengefasst, die am nächsten zueinander sind. Diese Prozedur führt man immer weiter fort, wobei immer größere Cluster entstehen. Am Ende erhält man dann einen einzigen großen Cluster.

Um die Vorgehensweise besser zu verstehen, schauen wir uns diese anhand eines einfachen Beispiels an.

Nehmen wir an, wir haben einen Datensatz mit sechs Datenpunkten {A,B,C,D,E,F}. Zu Beginn betrachten wir jeden einzelnen Datenpunkt als eigenen Cluster und berechnen alle Abstände zwischen ihnen.



Die beiden Punkte, die am nächsten zueinander sind, sind E und F. Diese fassen wir zum Cluster (E,F) zusammen. Im nächsten Schritt stellen wir fest, dass A und B am nächsten zueinander sind und fassen diese zum Cluster (A,B) zusammen. Die Punkte C und D werden an dieser Stelle noch als separate Cluster betrachtet.

Im nächsten Schritt allerdings fügen wir dem (E,F)-Cluster den Punkt D zu und erhalten den Cluster (D,E,F). Der Punkt C bleibt weiterhin ein eigener Cluster und (A,B) bleibt unverändert.

Im vorletzten Schritt fügen wir den Punkt C zum Cluster (D,E,F) hinzu und kommen bei nur noch den beiden Clustern (A,B) und (C,D,E,F) raus. Diese beiden letzten Cluster fassen wir dann in einem letzten großen Cluster zusammen, der alle Datenpunkte enthält.



## 2.10.2. Divisive Clustering

Der Divisive-Clustering-Algorithmus ist – wie oben bereits erwähnt – das Gegenteil zum agglomerativen Clustering und wird auch als Top-Down-Ansatz bezeichnet. Da er in der Praxis kaum zur Anwendung kommt, werden wir ihn hier nur kurz behandeln. Bei dieser Art des hierarchischen Clustering werden alle Datenpunkte zusammen am Anfang als ein einziger Cluster angesehen. Dieser Ur-Cluster wird dann bei jedem Schritt in weitere kleinere Cluster aufgeteilt, wobei immer die Datenpunkte von einem Cluster getrennt werden, die am wenigsten zu den anderen passen. Diese Prozedur der Aufteilung der Datenpunkte ist auch der Grund, wieso der Name des Algorithmus das Wort „Divisive“ enthält. Die Schritte zur Trennung der Datenpunkte werden so lange fortgeführt, bis man eine zu Beginn festgelegte Anzahl an Clustern erreicht hat.

## 2.10.3. Vor- und Nachteile des hierarchischen Clustering

Der wesentliche Vorteil des hierarchischen Clustering liegt in seiner Flexibilität, da der Algorithmus außer der Distanzfunktion (wie beispielsweise der bekannten euklidischen Metrik) und der angewandten Fusionierungsmethode (von denen wir die wichtigsten im nächsten Abschnitt kennenlernen werden) keine eigenen Parameter hat. Außerdem erhält man beim hierarchischen Clustering nicht nur eine Aufteilung der Daten in Cluster, sondern eine ganze Cluster-Hierarchie, die auch Unterstrukturen erlaubt. Dadurch erhält man eine gute Zusammenfassung des qualitativen Verhaltens des Clustering, was einem einen besseren Einblick in das Verhalten der Daten liefert.

Der Nachteil des hierarchischen Clustering liegt allerdings in dessen Analyse-Aufwand. Verfahren wie das K-Means-Clustering liefern eine einzelne Partitionierung der Daten, während das hierarchische Clustering



mehrere solcher Aufteilungen erstellt. Die Entscheidung, welche Aufteilung zur Analyse der Daten verwendet werden soll, liegt bei den Anwendenden. Ein weiterer Nachteil des hierarchischen Clustering zeigt sich bei Ausreißern in den Daten. Diese werden nämlich als eigene Cluster angesehen, was dazu führt, dass am Ende Cluster mit nur wenigen Datenpunkten herauskommen. Diese Mini-Cluster müssten dann im Nachhinein nachträglich analysiert werden, um ein adäquates Bild von den Daten zu erhalten.

## 2.10.4. Die Fusionierungsalgorithmen

Nun widmen wir uns dem eigentlich wichtigsten Teil des hierarchischen Clustering – der Fusion der einzelnen Teilcluster. Um die einzelnen Teilcluster zusammenbringen zu können, müssen wir die Ähnlichkeit zwischen diesen bestimmen, wofür es verschiedene Methoden gibt. Welche Methode letztendlich die richtige ist, hängt von den vorliegenden Daten ab. Schauen wir uns mal anhand eines einfachen Beispiels an, wie das genau ablaufen würde.

Nehmen wir an, ein Lehrer hätte 5 Schüler\*innen (um nicht allzu viele Datenpunkte zu haben), und jede\*n Schüler\*in hat er für eine bestimmte Aufgabe mit Punkten bewertet. Die Bewertungen sind in der folgenden Tabelle aufgeführt.



Student_ID	Marks
1	10
2	7
3	28
4	20
5	35

Um die Schüler\*innen nun in Gruppen einzuteilen, stellen wir die sogenannte Distanzmatrix auf, die die Abstände zwischen den Datenpunkten enthält. Da wir 5 Schüler\*innen haben, werden wir am Ende eine 5x5-Matrix erhalten. Je mehr Datenpunkte ein Datensatz enthält, umso größer wird auch die Matrix.

Um die Abstände zwischen den Datenpunkten zu berechnen, nutzen wir die euklidische Metrik. Beispielsweise wäre der Abstand zwischen Schüler\*in 1 und 2 folgender:

$$D = (\sqrt{10 - 7})^2 = 3$$

Auf diese Weise werden die Abstände zwischen allen Punkten berechnet und in die Matrix eingetragen. Das sieht dann am Ende so aus:

ID	1	2	3	4	5
1	0	3	18	10	25
2	3	0	21	13	28
3	18	21	0	8	7
4	10	13	8	0	15
5	25	28	7	15	0



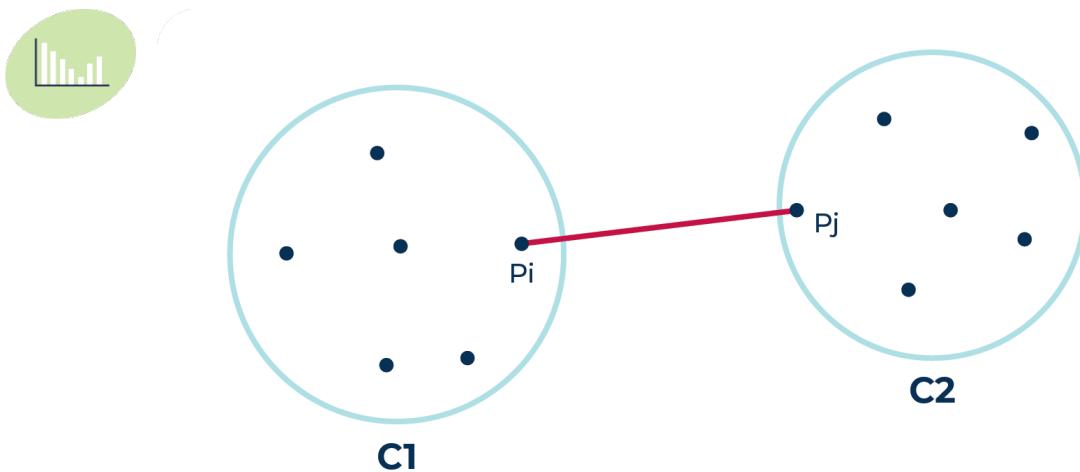
Auffallend ist hier, dass die Diagonal-Elemente alle Null sind. Das liegt daran, dass der Abstand eines Punktes zu sich selbst natürlich Null ist. Diese Distanzmatrix zu berechnen, kann für viele Datenpunkte sehr aufwändig werden, doch zum Glück gibt es eine `scipy`-Bibliothek, mit der man das ganz einfach hinbekommt.

### Single-Linkage oder Minimum

Nehmen wir an, wir hätten zwei Cluster – C1 und C2 –, deren Ähnlichkeit wir bestimmen wollen, um auf dessen Basis zu entscheiden, ob wir die Cluster zusammenführen sollen oder nicht. Die Punkte in Cluster C1 bezeichnen wir mit  $P_i$  und die Punkte in Cluster C2 mit  $P_j$ . Die Ähnlichkeit zwischen den beiden Clustern definieren wir dann als den kleinsten Abstand, der zwischen zwei Punkten aus den beiden Clustern herrscht. Für die Ähnlichkeit der beiden Cluster schreiben wir  $\text{Sim}(C1, C2)$  und für den Abstand zwischen zwei Punkten x und y schreiben wir  $D(x, y)$ . Mathematisch ausgedrückt sieht die Ähnlichkeit bei der Single-Linkage-Methode folgendermaßen aus:

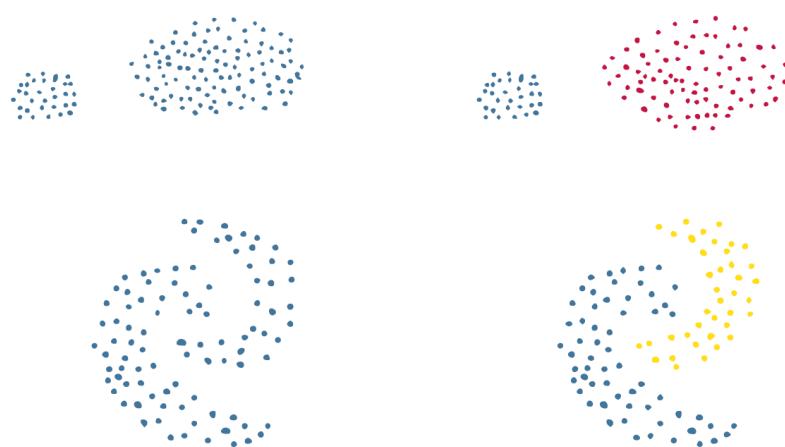
$$\text{Sim}(C1, C2) = \min(D(P_i, P_j))$$

In einem Satz gesagt berechnen wir alle Abstände zwischen den Punkten aus C1 und C2 und wählen die Punkte mit dem minimalen Abstand aus, um die Ähnlichkeit der beiden Cluster zu quantifizieren.





Der Vorteil der Single-Linkage-Methode liegt bei Daten, die keine elliptische Form haben und bei denen der Abstand zwischen den Clustern nicht zu groß ist.



Original data vs Clustered data using MIN approach

Der Nachteil dieser Methode zeigt sich bei Clustern, zwischen denen ein Datenrauschen herrscht. Dadurch können die Cluster nicht mehr adäquat voneinander getrennt werden.



Original data vs Clustered data using MIN approach

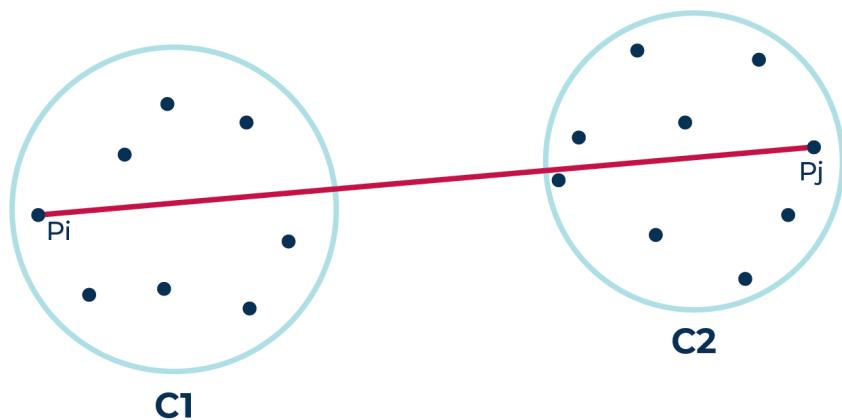


### Complete Linkage oder Maximum

Die Complete-Linkage-Methode ist das komplette Gegenteil zum Single-Linkage-Ansatz. Hier wird die Ähnlichkeit zweier Cluster über den maximalen Abstand zwischen zwei Punkten  $P_i$  aus  $C_1$  und  $P_j$  aus  $C_2$  definiert. Mathematisch ausgedrückt sieht das folgendermaßen aus:



$$\text{Sim}(C_1, C_2) = \max(D(P_i, P_j))$$



Der Vorteil dieser Methode zeigt sich bei Clustern, zwischen denen ein Datenrauschen besteht. Hier lassen sich die Cluster besser trennen als bei der Single-Linkage-Methode.



Original data vs Clustered data using MAX approach



Der Nachteil der Complete-Linkage-Methode liegt allerdings darin, dass diese Cluster mit einer sphärischen Form bevorzugt. Außerdem tendiert dieser Ansatz dazu, größere Cluster aufzubrechen.

### Group Average

Anstatt dass man nur einen Datenpunkt pro Cluster verwendet, um die Ähnlichkeit zwischen zwei Gruppen zu bestimmen, nutzt man bei dieser Methode alle im Cluster vorhandenen Datenpunkte. Die Ähnlichkeit zwischen zwei Clustern ist als durchschnittlicher Abstand zwischen allen Datenpunkten definiert, wobei man diesen Durchschnitt zur Normierung durch das Produkt aus der Anzahl der Cluster in C1 und C2 teilt. Als Formel ausgeschrieben sieht das dann so aus:

$$Sim(C1, C2) = \sum_i \frac{D(P_i, P_j)^2}{|C1| \cdot |C2|}$$

Diese Methode funktioniert ganz gut, wenn zwischen den Daten kein Datenrauschen herrscht, allerdings bevorzugt auch diese Methode sphärische Cluster.

### Die Ward-Methode

Die letzte und häufig verwendete Methode ist die sogenannte Ward-Linkage-Methode. Im Prinzip funktioniert diese genauso wie die Group-Average-Methode, allerdings nutzt man hier statt dem einfachen Abstand zwischen den Punkten die quadratische Distanz und dividiert diese wieder

$$Sim(C1, C2) = \sum_i \frac{D(P_i, P_j)^2}{|C1| \cdot |C2|}$$



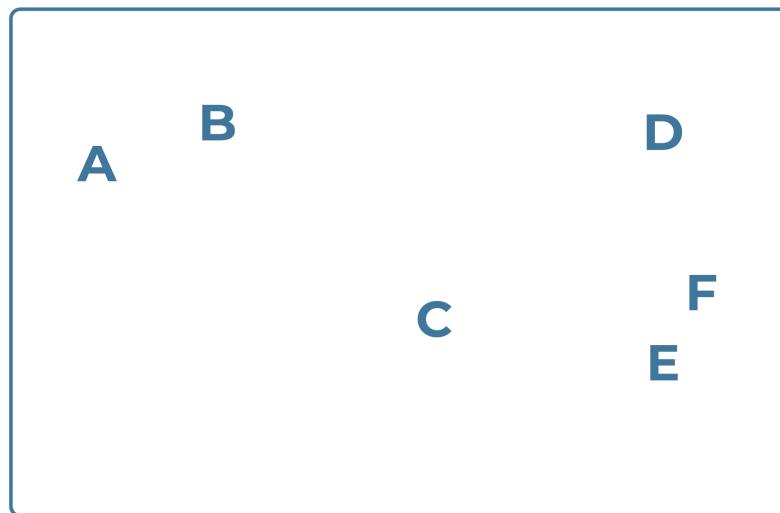
durch das Produkt aus der Anzahl der Punkte in C1 und C2. Als Formel ausgeschrieben haben wir dann:

Anders als die Group-Average-Methode eignet sich die Ward-Linkage-Methode gut, um Cluster voneinander zu trennen, zwischen denen ein Datenrauschen herrscht. Allerdings liegt auch hier der Nachteil in der Bevorzugung von sphärischen Clustern.

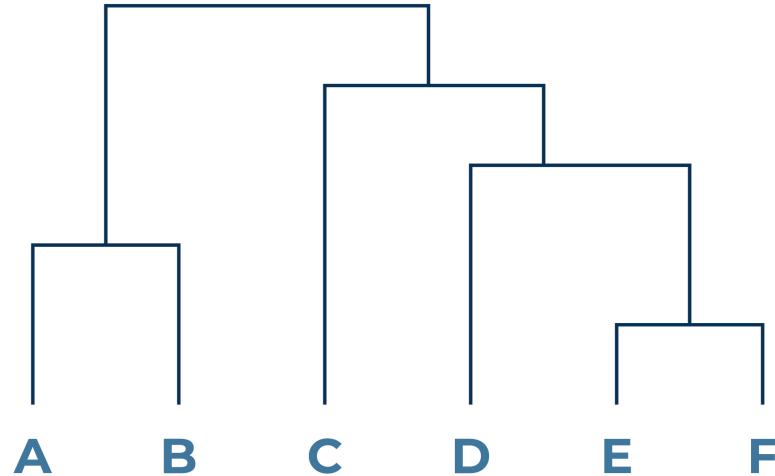
## 2.10.5. Dendrogramme

Um die hierarchische Beziehung zwischen den Datenpunkten und Clustern zu veranschaulichen, verwendet man beim hierarchischen Clustering sogenannte Dendrogramme. Der Nutzen von Dendrogrammen liegt darin herauszufinden, welches der beste Weg ist, um die Datenpunkte zu Clustern zusammenzuführen. Gleichzeitig wird durch die Visualisierung des Clustering- Prozesses auch ein besseres Verständnis für die Natur der Daten geschaffen. Schauen wir uns mal im Detail an, wie man ein solches Dendrogramm erstellt.

Beginnen wir wieder mit einem einfachen Streudiagramm von sechs Datenpunkten A, B, C, D, E, F.



Der hierarchische Clustering-Algorithmus würde als erstes die Punkte E und F zu einem Cluster zusammenfassen, im zweiten Schritt dann A und B. Während dieser ersten beiden Schritte bleiben C und D eigene Cluster. Im dritten Schritt wird dann Punkt D dem Cluster (E,F) zugeordnet, im vierten Schritt kommt dann noch der Punkt C dazu. Im letzten Schritt führen wir den Cluster (A,B) mit dem Cluster (C,D,E,F) zu einem großen Cluster zusammen. Das Dendrogramm für den Clustering-Prozess sieht folgendermaßen aus:



Das Dendrogramm zeigt uns, wann welche Datenpunkte im Clustering-Prozess zusammengeführt werden. Um das Dendrogramm zu lesen, müssen wir auf die Höhe achten, auf welcher Datenpunkte zusammengeführt werden. Wir sehen, dass die Höhe, auf der E und F zusammenkommen, niedriger ist, als für A und B, was uns zeigt, dass diese Datenpunkte als erstes einen Cluster bilden.

## 2.10.6. Programmierung des hierarchischen Clustering

Nun können wir uns endlich anschauen, wie wir mit Hilfe der scikit-learn-Bibliothek einen hierarchischen Clustering-Algorithmus implementieren können. Zunächst einmal müssen wir dafür alle wichtigen Bibliotheken wie matplotlib und das scipy.cluster.hierarchy-Modul importieren, welches wir für die Funktionen dendrogram() und linkage() benötigen. Vor allem aber benötigen wir das sklearn.cluster-Modul, um die AgglomerativeClustering-Klasse verwenden zu können.



Die AgglomerativeClustering-Klasse enthält verschiedene wichtige Parameter, mit denen wir Eigenschaften des Clustering-Objekts festlegen können:

- n\_clusters: int or None, default=2

Hiermit ist die Anzahl an Clustern gemeint, die der hierarchische Clustering-Algorithmus identifizieren soll. Per default ist der Wert hierfür 2.

- linkage{'ward', 'complete', 'average', 'single'}, default='ward' Hier geht es darum, welches Linkage-Kriterium man verwendet. 'ward': hierbei wird die Varianz minimiert  
'average': nutzt den durchschnittlichen Abstand  
'complete' oder 'maximum': nutzen die maximalen Abstände  
'single': nutzt das Minimum

Anschließend erzeugen wir simple Testdaten in Form von zwei Listen, die wir dann mit der zip()- Funktion in einer Liste von zwei Tupeln zusammenfassen.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.cluster import AgglomerativeClustering

x = [4, 6, 9, 4, 3, 11, 12, 6, 10, 12]
y = [22, 18, 25, 16, 16, 24, 24, 22, 21, 21]

data = list(zip(x,y))
```

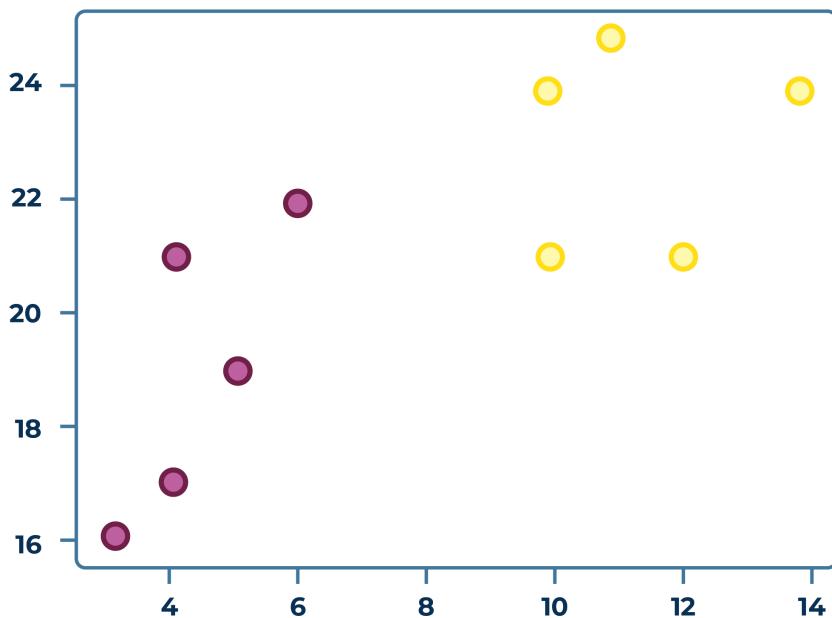


Um schließlich den Algorithmus auf den Daten laufen lassen zu können, müssen wir ein Objekt erzeugen, das die Daten in Cluster einteilt. Mit der `fit_predict()`-Funktion können wir uns dann die Cluster in einem Streudiagramm durch Farben getrennt darstellen lassen.

```
hierarchical_cluster = AgglomerativeClustering(n_clusters=2, affinity='euclidean', linkage='ward')
labels = hierarchical_cluster.fit_predict(data)

plt.scatter(x, y, c=labels)
plt.show()
```

Das Ergebnis als Streudiagramm sieht dann so aus:





### 3. Neural Networks

#### Lernziele für dieses Kapitel:

**Grobziel:** Die Lernenden können selbstständig ein neuronales Netz konstruieren, es trainieren und die Ergebnisse auswerten.

Feinziele (ca. 3-5)	Inhalte
1. Die Lernenden können selbstständig ein neuronales Netz zum Supervised Learning entwerfen.	<ul style="list-style-type: none"><li>• Signalübertragung im Neural Network</li><li>• MNIST-Datensatz</li><li>• Vorbereitung der Daten und Training</li></ul>
2. Die Lernenden können die Ergebnisse des Trainings eines Neural Networks auswerten.	<ul style="list-style-type: none"><li>• Performance-Metrik</li><li>• Untersuchung des Einflusses von Lernraten und Epochen</li></ul>
3. Die Lernenden können selbstständig mit Hilfe der PyTorch-Bibliothek ein neuronales Netz entwerfen und auswerten.	<ul style="list-style-type: none"><li>• PyTorch-Grundlagen (Tensoren, DataLoaders, Klassen)</li><li>• Aktivierungsfunktionen</li><li>• Loss Functions</li><li>• Hyperparameter</li><li>• Speichern von Modellen</li></ul>

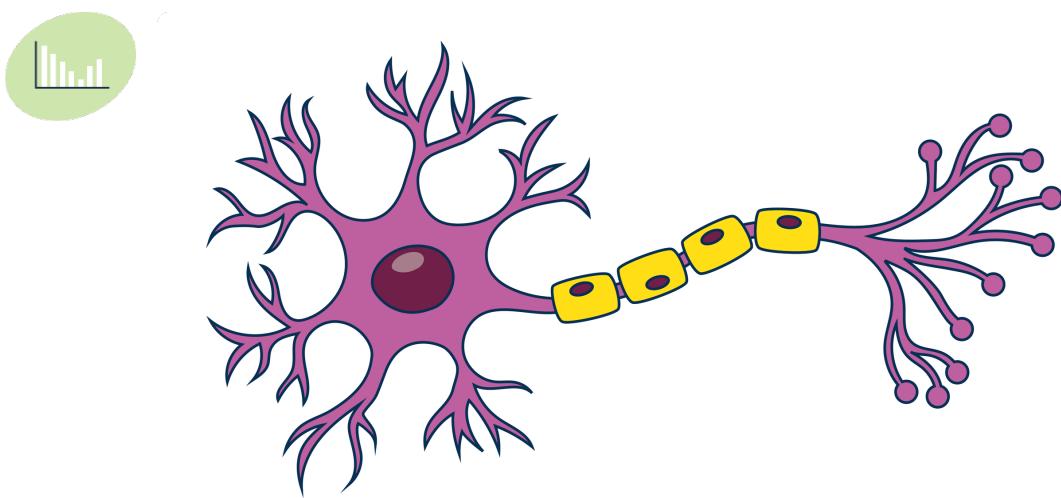
Mit Neural Networks (neuronalen Netzwerken) sind wir nun in einem eigenen Bereich des Machine Learning angekommen. Bisher hatten wir die Machine-Learning-Algorithmen in Supervised- und Unsupervised-



Algorithmen unterscheiden können. Neural Networks dagegen können sowohl für geführtes Training als auch für eigenständige Mustererkennung eingesetzt werden – einer der Gründe, wieso Neural Networks ein sehr mächtiges Werkzeug zur Datenanalyse sind. In diesem Kapitel werden wir uns im Detail anschauen, was ein Neural Network ist und wie man eines von der Pike auf programmieren kann. Anschließend befassen wir uns mit der sehr nützlichen Bibliothek PyTorch, mit welcher die Implementierung von komplexen Neural Networks möglich wird. Beginnen werden wir beim biologischen Vorbild der künstlichen Neural Networks: den Neuronen.

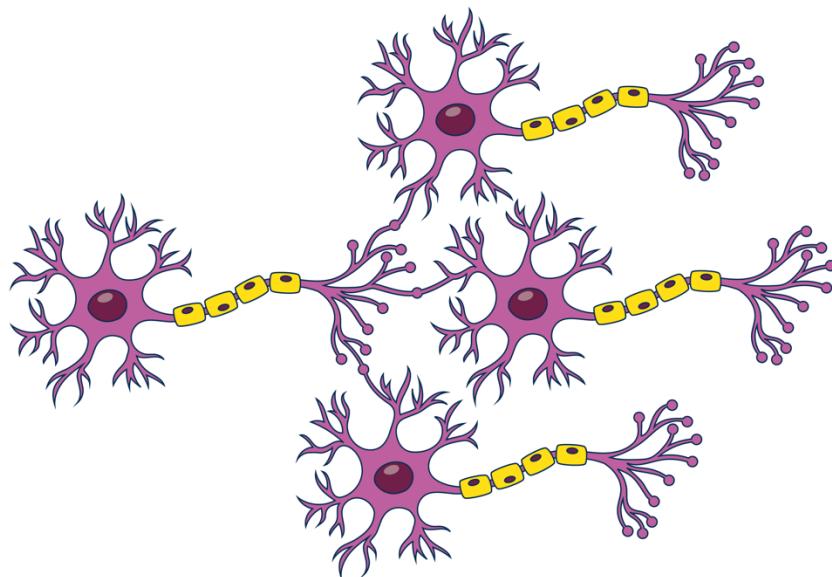
### 3.1. Neuronen

Aus der Biologie wissen wir, dass in tierischen und menschlichen Gehirnen die Informationsverarbeitung über die Verwendung sogenannter Neuronen stattfindet. Neuronen lassen sich als die Grundeinheit der Informationsverarbeitung im Gehirn sehen. Üblicherweise wird ein solches Neuron graphisch so veranschaulicht:





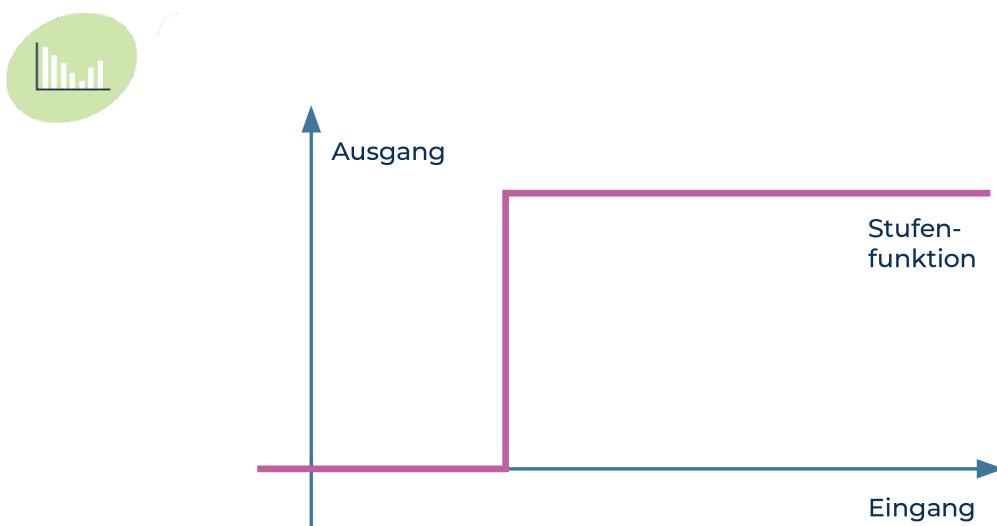
Jeder einzelne Ast in der großen orangenen Verästelung links wird Dendrit genannt, während die kleinere orangene Verästelung auf der rechten Seite die Terminalen sind. Der gelbe Übergang zwischen den Dendriten ist das Axon. Wenn Information im Gehirn verarbeitet wird, dann werden elektrische Signale durch ein Netzwerk aus Neuronen geleitet. Die Terminalen sind mit Dendriten von anderen Neuronen verbunden.



Das Neuron übernimmt über die Dendriten ein elektrisches Eingangssignal und gibt ein anderes elektrisches Signal aus. Aus Beobachtungen hat man entnehmen können, dass Neuronen ein elektrisches Signal nicht sofort weitertransportieren. Stattdessen wird die Eingabe unterdrückt, bis sie ausreichend groß ist, um die Weiterleitung des Signals auszulösen. Es muss also ein bestimmter Schwellwert erreicht werden, bevor irgendein Signal weitergeleitet wird. Dadurch wird es dem natürlichen Neural Network möglich, Rauschsignale zu unterdrücken und nur gewollte, starke Signale in den Verarbeitungsprozess miteinzubeziehen.

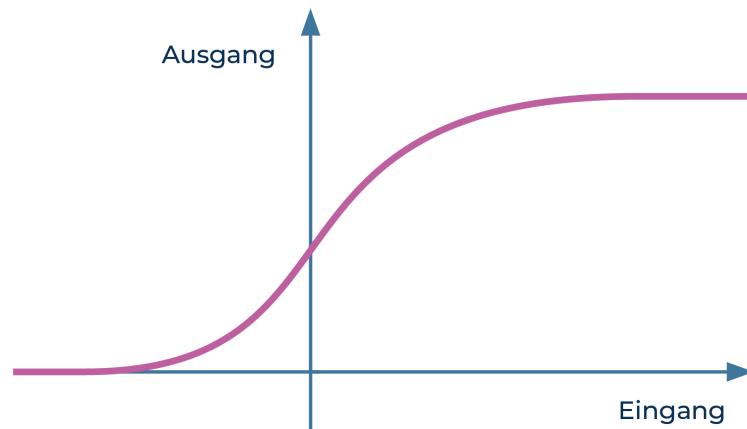


Da wir das Ziel haben, die Prinzipien der natürlichen Neural Networks in einem Python-Code zu kopieren, schauen wir uns an, wie diese Rauschunterdrückung mathematisch ausgedrückt aussehen würde. Wir sprechen hier von einer sogenannten Aktivierungsfunktion, einer Funktion, die ein Eingangssignal übernimmt und unter Berücksichtigung eines Schwellwertes ein Ausgangssignal generiert. Es gibt viele Arten von Aktivierungsfunktionen. Die einfachste ist die sogenannte Stufenfunktion:



Aus dem Diagramm kann man ablesen, dass für kleine Eingabesignale das Ausgangssignal 0 ist. Nachdem aber eine Schwelle erreicht ist, springt der Ausgabewert hoch. Man sagt, dass das Neuron „feuert“, wenn das Eingabesignal den Schwellenwert erreicht.

Eine etwas mehr an das natürliche Vorbild angepasste Aktivierungsfunktion ist die sogenannte Sigmoid-Funktion. Sie verläuft ein wenig sanfter als die abrupte Stufenfunktion und hat die Form des Buchstabens S. Diese Sigmoid-Funktion werden wir sehr häufig als Aktivierungsfunktion bei unseren Neural Networks nutzen.



Mathematisch wird die Sigmoid-Funktion durch diese Vorschrift beschrieben:

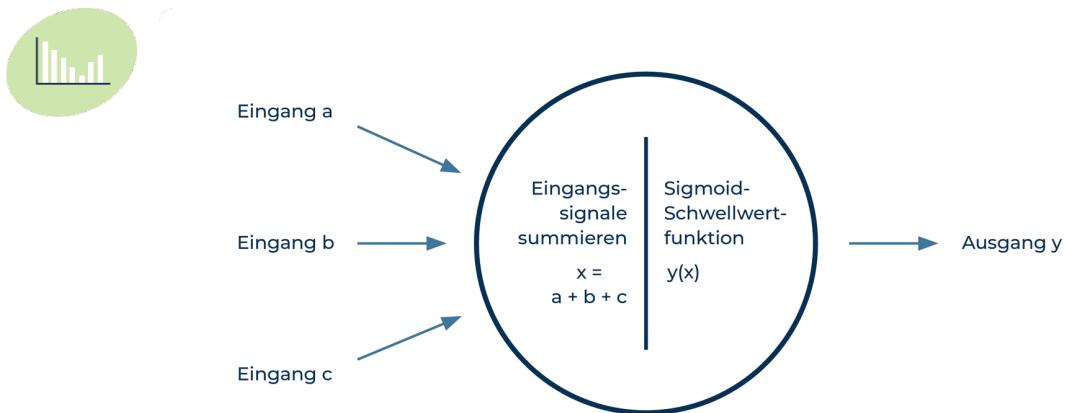
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Ein weiterer Vorteil der Sigmoid-Funktion besteht darin, dass es leichter ist, mit ihr zu rechnen. Gerade die Stufenfunktion würde später bei beispielsweise dem Gradienten-Verfahren zu Problemen führen.

In einem künstlichen Neuron gehen also Signale in ein Neuron hinein und addieren sich so lange, bis die Aktivierungsfunktion aktiviert wird und das Neuron feuert. Wenn das kombinierte Signal aus verschiedenen Eingängen nicht groß genug ist, unterdrückt die sigmoidale Schwellwertfunktion das Ausgangssignal. Interessant ist hier, dass verschiedene Kombinationen der Stärken der Eingangssignale ein Feuern des Neurons auslösen können. Wenn beispielsweise nur eines der Eingangssignale groß ist und die anderen klein sind und nur einen geringen Beitrag leisten, können diese kleinen Beiträge reichen, um das Neuron zum Feuern zu bringen. Genauso kann das Neuron feuern, wenn die Eingänge zwar alle für sich genommen



nicht ausreichend groß sind, aber in der Summe ein Signal erzeugen, das ausreicht, um den Schwellwert zu überschreiten. Hier zeigt sich dann auch ein enormer Vorteil gegenüber der üblichen Vorgehensweise eines Computers bei der Lösung von Problemen.

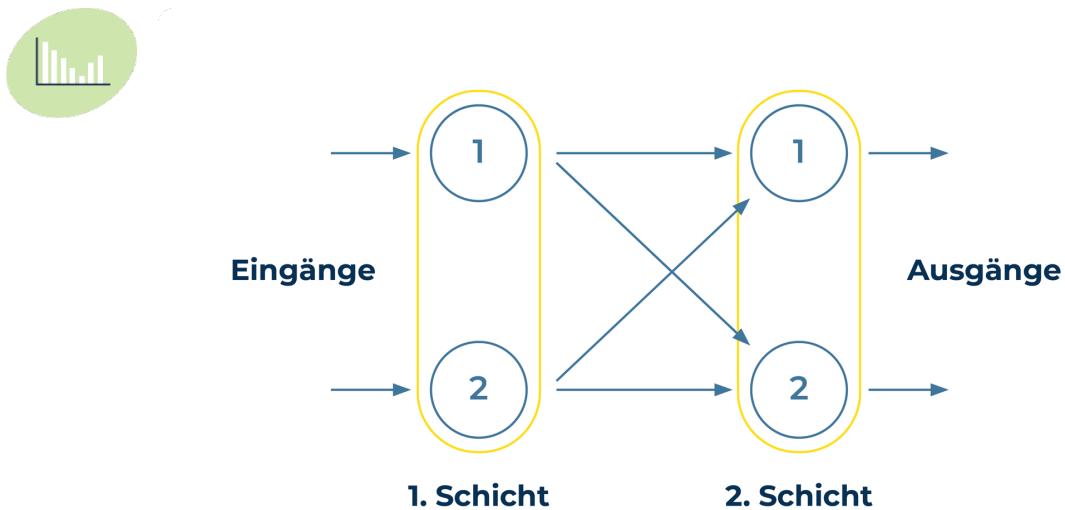


Herkömmliche Computer verarbeiten Daten sequentiell und befolgen dabei sehr konkrete Vorschriften. Bei diesen strikten Berechnungen gibt es weder Unschärfe (Fuzziness) noch Mehrdeutigkeit. Bei Neural Networks allerdings werden die eingehenden Daten parallel verarbeitet und Fuzziness spielt in dieser Verarbeitung eine große Rolle. Das macht Neural Networks zu einem mächtigeren Werkzeug als die Machine-Learning-Algorithmen, die wir bisher kennengelernten.

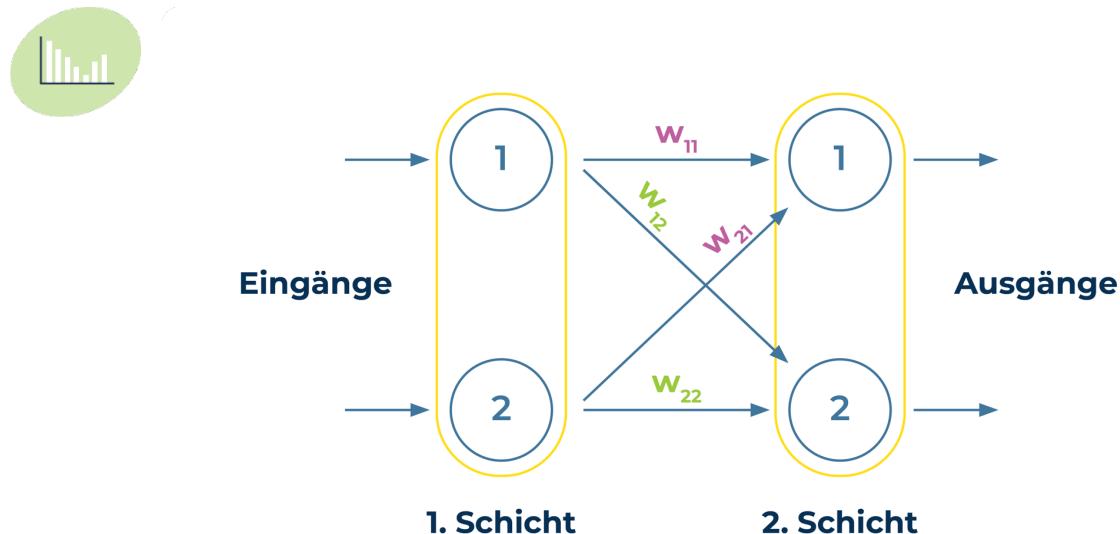
Ein künstliches Neural Network besteht also – wie sein natürliches Vorbild – aus Neuronen, die miteinander verbunden sind. Jedes Neuron kann mehrere Signale als Eingang einnehmen und diese kombiniert zu einem Ausgangssignal umwandeln – wenn der Schwellwert erreicht ist. Wir werden bei künstlichen Neural Networks die Neuronen in Schichten organisieren. In der Grafik unten ist ein einfaches Neural Network mit zwei Schichten und jeweils zwei Neuronen pro Schicht dargestellt. Jedes Neuron aus der ersten Schicht ist mit jedem Neuron aus der zweiten Schicht



verbunden. Es bestehen also vier Verbindungen zwischen den beiden Schichten. Die linke Schicht nennen wir die Eingabeschicht, die rechte Schicht ist die Ausgabeschicht. Wir können auch Neural Networks mit mehreren Schichten zwischen der Eingabe- und Ausgabeschicht entwerfen: Diese nennt man Hidden Layers.



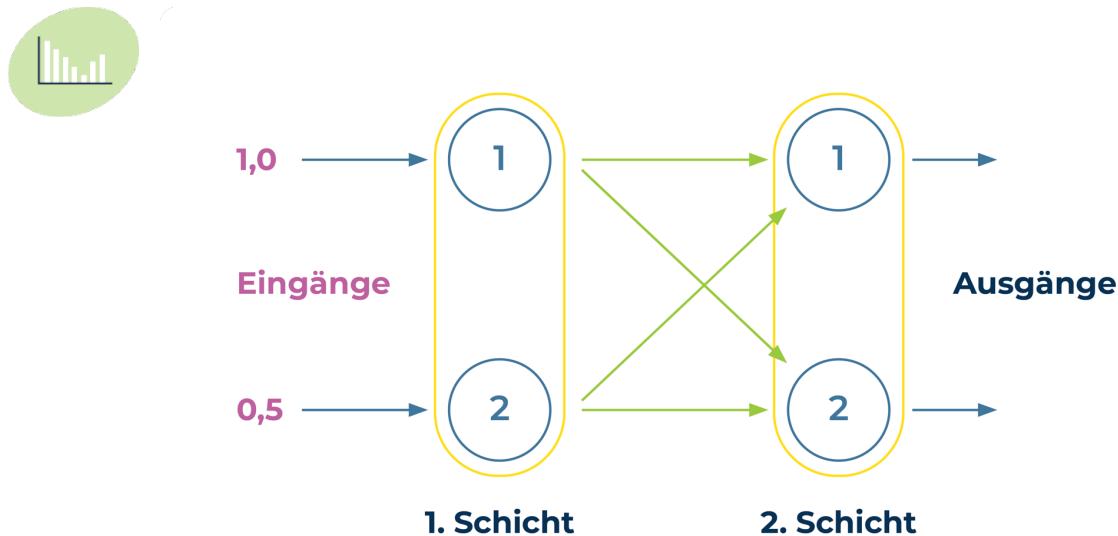
Wenn ein Signal von links in die ersten beiden Neuronen kommt, dann werden diese – wenn sie den Schwellwert überschritten haben – über die vier Verbindungen zu den Neuronen der nächsten Schicht geleitet. Die Frage an dieser Stelle aber ist: Wie stark wird ein Signal durch eine der Leitungen gejagt? Die Stärke, mit der ein Signal weitergeleitet wird, legt man über das Gewicht der Verbindung fest. Jede der Verbindungen zwischen den Schichten bekommt ein Gewicht.



Hierbei steht der Wert  $w_{12}$  beispielsweise für das Gewicht der Verbindung zwischen dem ersten Neuron in der linken Schicht und dem zweiten Neuron in der rechten Schicht.

### 3.2. Signalübertragung in einem Neural Network

Um genauer zu verstehen, wie ein Neural Network Informationen verarbeitet, schauen wir uns jetzt mal im Detail an, wie ein Signal von der Eingabeschicht in die Ausgabeschicht weitergeleitet wird. Die dabei beschriebene Methodik lässt sich auch auf Neural Networks mit mehr als zwei Schichten anwenden. Beginnen wir mit einem einfachen Neural Network aus zwei Schichten mit zwei Neuronen pro Schicht. In Neuron 1 in der Eingabeschicht soll ein Signal der Stärke 1.0 ankommen, im Neuron 2 ein Signal der Stärke 0.5.



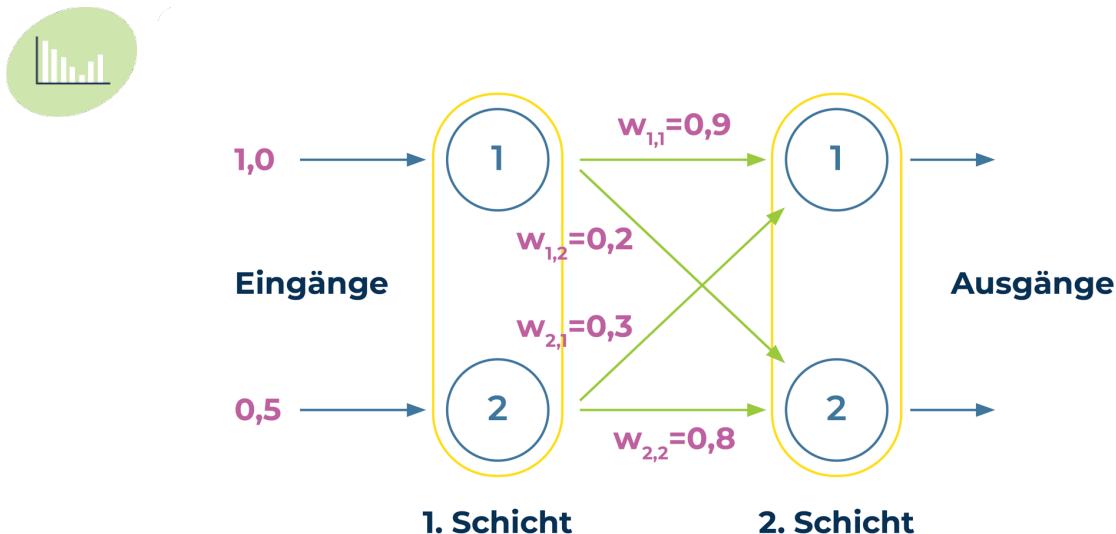
Die erste Schicht ist einfach nur die Eingabeschicht: In ihr wird keine Aktivierungsfunktion angewendet. Dass das so gehandhabt wird, hängt mit der Entwicklungsgeschichte von Neural Networks zusammen und hat keinen tieferen Grund. Diese Signale werden nun über die vier gewichteten Verbindungen zur zweiten Schicht weitergeleitet. Die Gewichte der einzelnen Verbindungen setzen wir zufällig fest:

$$w_{1,1} = 0.9$$

$$w_{1,2} = 0.2$$

$$w_{2,1} = 0.3$$

$$w_{2,2} = 0.8$$



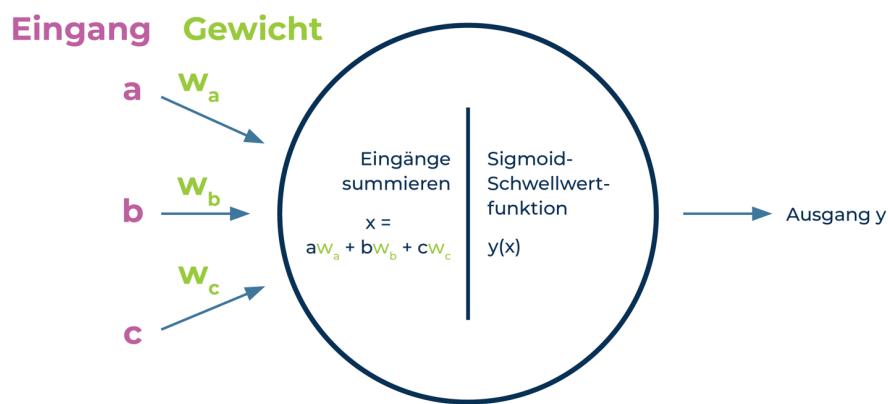
Wie verarbeiten jetzt die einzelnen Neuronen die einkommenden Signale aus der vorherigen Schicht? Entsprechend ihrer Gewichte werden die Eingänge summiert. Anschließend wendet man die Sigmoid-Funktion auf diese Summe an und erhält den Ausgang. Dieser ist in unserem einfachen Fall eines zwei-schichtigen Neural Networks dann die Ausgabe des Netzes, würde aber bei weiteren Schichten das Signal sein, das gewichtet in den Neuronen der nächsten Schicht ankommt.

Um die Prozedur der Signalverarbeitung in den Neuronen zu verstehen, rechnen wir das Ganze einfach mal durch. Konzentrieren wir uns hierfür zunächst auf den ersten Knoten in der zweiten Schicht. Beide Knoten aus der Eingabeschicht sind mit ihm verbunden. Diese Eingabesignale haben die Werte 1.0 und 0.5. Der Verknüpfung vom ersten Knoten in der Eingabeschicht zum ersten Knoten in der Ausgabeschicht ( $w_{11}$ ) ist ein Gewicht von 0.9 zugeordnet. Die Verknüpfung zwischen dem zweiten Knoten in der Eingabeschicht und dem ersten Knoten in der Ausgabeschicht ( $w_{21}$ ) hat den Wert 0.3. Daraus ergibt sich dann für die gewichtete Summe als Eingang im Neuron:



$x = (\text{Ausgabe vom \textbf{ersten} Knoten} \times \text{Verknüpfungsgewicht}) + (\text{Ausgabe vom \textbf{zweiten} Knoten} \times \text{Verknüpfungsgewicht})$

$$x = (0.9 \cdot 1.0) + (0.5 \cdot 0.3) = \\ 0.9 + 0.15 = 1.05$$



Diesen Wert müssen wir nun in die Sigmoid-Funktion ( $\sigma(x) = 1 / (1 + e^{-x})$ ) als Argument eingeben, um  $1/(1+e^{-x})$  die Ausgabe zu erhalten:

$$\sigma(x) = \frac{1}{1 + 0.3499} = \frac{1}{1.3499} = 0.7408$$

Das Gleiche müssen wir jetzt auch für den zweiten Knoten in der Ausgabeschicht durchführen.



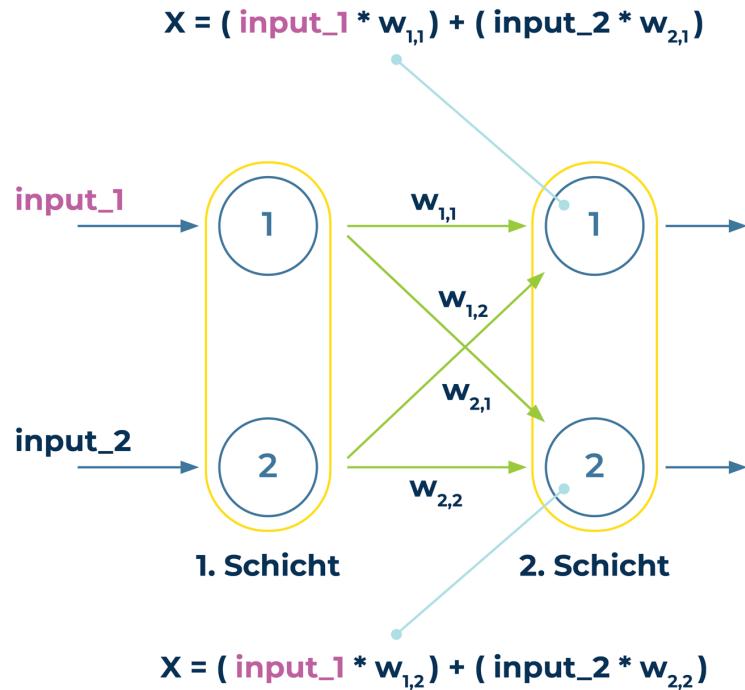
### 3.3. Signalübertragung in einem drei-schichtigen Neural Network

Die vielen Berechnungen, die bei einem größeren Neural Network durchgeführt werden müssten, lassen sich prägnanter durch Matrixmultiplikation ausdrücken. Dadurch lassen sich die Verknüpfungen zwischen den Schichten übersichtlicher strukturieren.

Wir beginnen damit, die Eingangssignale in einem Vektor zusammenzufassen. Im Beispiel des zwei-schichtigen Neural Networks von vorher wäre dann der Eingangssignal-Vektor:

$$\begin{pmatrix} \text{input1} \\ \text{input2} \end{pmatrix}$$

Um nun die Signalübertragung in die nächste Schicht zu berechnen, multipliziert man diesen Vektor mit der 2x2-Gewichte-Matrix. Als Ergebnis erhält man dann wieder einen 2-dimensionalen Vektor.



Wir können allgemein für ein beliebiges Neural Network schreiben:

$$X = W I$$

wobei W die Gewichtsmatrix ist, I der Input-Vektor und X das Ergebnis. Je mehr Knoten eine Schicht hat, umso größer werden die Vektoren und die Gewichtsmatrix. Interessant ist an dieser Stelle, dass die Gewichtsmatrix nicht zwingend symmetrisch sein muss. Wie wir später noch bei der Programmierung sehen werden, muss die Anzahl der Neuronen in den Schichten nicht unbedingt gleich sein. Um letztendlich aus der obigen Matrixmultiplikation das Ausgangssignal zu erhalten, müssen wir noch die Sigmoid-Funktion darauf anwenden. Der Output O wird dann geschrieben als:



$$\mathbf{O} = \text{sigmoid}(\mathbf{X})$$

wobei  $\text{sigmoid}(\mathbf{X})$  einfach bedeutet, dass die Sigmoid-Funktion auf jede Komponente des Vektors  $\mathbf{X}$  angewendet wird. Auf dieser Basis können wir uns nun die Signalübertragung in einem Neural Network mit drei Schichten und jeweils drei Neuronen anschauen. Die beiden obigen Ausdrücke gelten für die Signalübertragung zwischen einer Schicht und der nächsten. Bei einer zusätzlichen Schicht führen wir einfach die Matrixmultiplikation und die Anwendung der Sigmoid-Funktion noch einmal aus – diesmal jedoch wird eine andere Gewichtsmatrix verwendet.

### 3.4. Training des Neural Networks

Wir wissen jetzt, wie ein Signal in einem Neural Network übertragen wird, aber wie wird es denn eigentlich trainiert? Der Grundgedanke bei einem Neural Network ist ja, dass man die Eingabeschicht mit Daten füttert und die Ausgabeschicht dann das richtige Ergebnis (wie zum Beispiel die richtige Klassifizierung eines Bildes) ausspuckt. Diese sogenannte Signalübertragung von Daten - beginnend bei der Eingabeschicht über die versteckte(n) Schicht(en) - wird nachher in der Programmierung durch die `feedforward()` - Funktion implementiert. Wenn wir die Gewichte zu Beginn des Trainings zufällig wählen, wird uns das Neural Network wahrscheinlich nicht gleich das richtige Ergebnis ausgeben. Wir müssen also die Gewichte anpassen. Mit jedem neuen Datenelement, das in die Eingabeschicht gegeben wird, beginnt ein neuer Durchlauf im Training. Das vom Neural Network ausgespuckte Ergebnis wird mit dem eigentlich erwarteten Ergebnis verglichen. Der Fehler wird verwendet, um die Gewichte zwischen den Schichten anzupassen. Um die Gewichte also anpassen zu können, müssen wir erst verstehen, wie genau man den Fehler an der Ausgabeschicht berechnet und wie er sich auf die Gewichte zwischen den Schichten auswirkt. Das wird als Fehler-Backpropagierung bezeichnet.

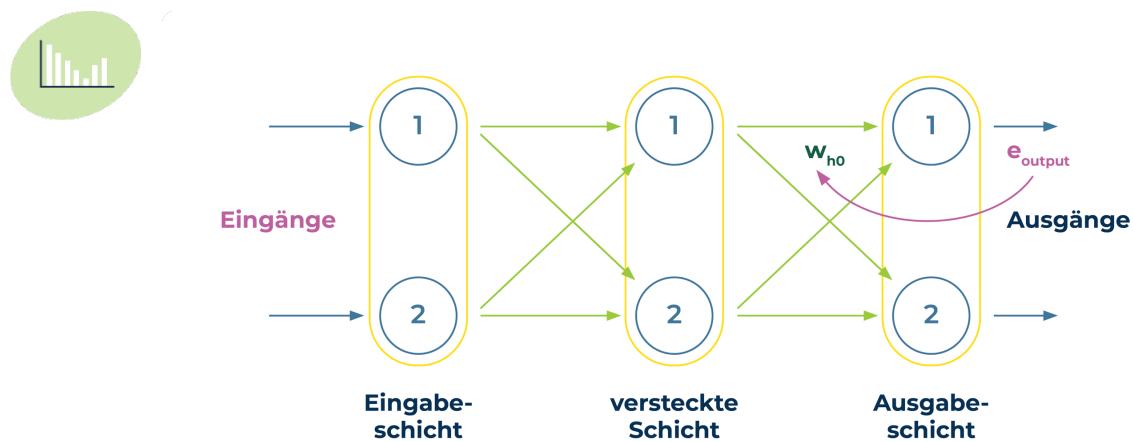


### 3.4.1. Fehler-Backpropagierung

Beginnen wir für unsere Überlegungen damit, uns ein drei-schichtiges Neural Network mit jeweils zwei Neuronen pro Schicht vorzustellen. Die Ausgabeschicht liefert also zwei Ausgänge. Jeder dieser Ausgänge kann einen Fehler haben, eine Abweichung zur eigentlich erwarteten Aussage. Wenn wir den Fehler  $e$  (error), die Ausgabe des Netzes  $o$  (Output) und den eigentlichen Zielwert  $t$  (Target) nennen, dann können wir den Fehler einfach über die Differenz zwischen Target und Output definieren:

$$e = t - o$$

Der Fehler der einzelnen Ausgabewerte soll nun zurückgeführt werden auf die vorherigen Schichten, um die Gewichte zwischen den Schichten anzupassen. Dafür müssen wir herausfinden, welchen Anteil die Neuronen der Eingabe- und der versteckten Schicht am Fehler an der Ausgabeschicht haben.

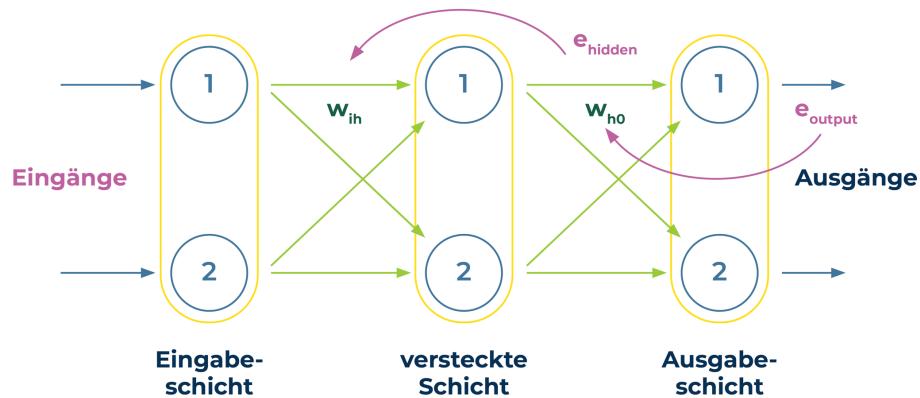


Dafür teilen wir die Fehler entsprechend der Gewichte auf.

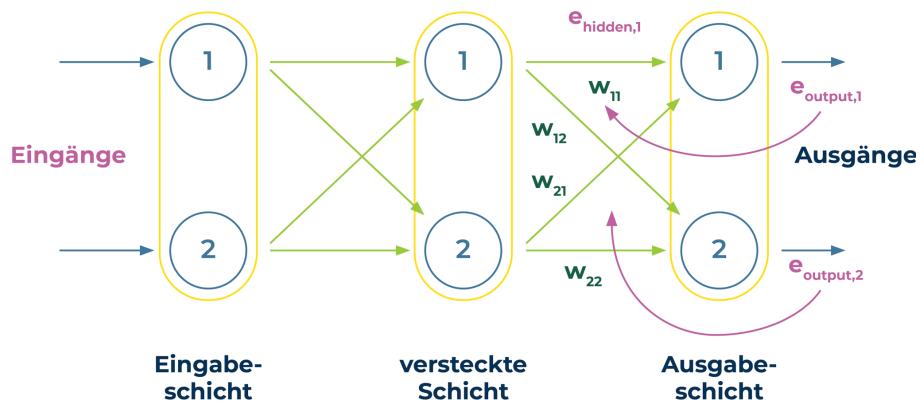
Wir müssen also die Fehler in der Eingabeschicht entsprechend der Verknüpfungsgewichte so aufteilen, dass wir den Fehler im ersten und im



zweiten Neuron der versteckten Schicht erhalten. Diesen Fehler können wir dann analog auf die Eingabeschicht zurückführen.



Schauen wir uns nun an, wie wir die Fehler-Backpropagierung berechnen können. Dafür nennen wir den Fehler am ersten Neuron der Ausgabeschicht  $e_{output,1}$  und den Fehler am zweiten Neuron der Ausgabeschicht  $e_{output,2}$ . Dieser Fehler soll nun auf einen Fehler in den Neuronen der versteckten Schicht zurückgeführt werden. Die Fehler in dieser Schicht bezeichnen wir analog zur Ausgabeschicht mit  $e_{hidden,1}$  und  $e_{hidden,2}$ .





Das erste Neuron in der versteckten Schicht erhält seinen Fehler  $e_{hidden,1}$  aus den beiden Fehlern  $e_{output,1}$  und  $e_{output,2}$ , anteilig der Verknüpfungsgewichte.

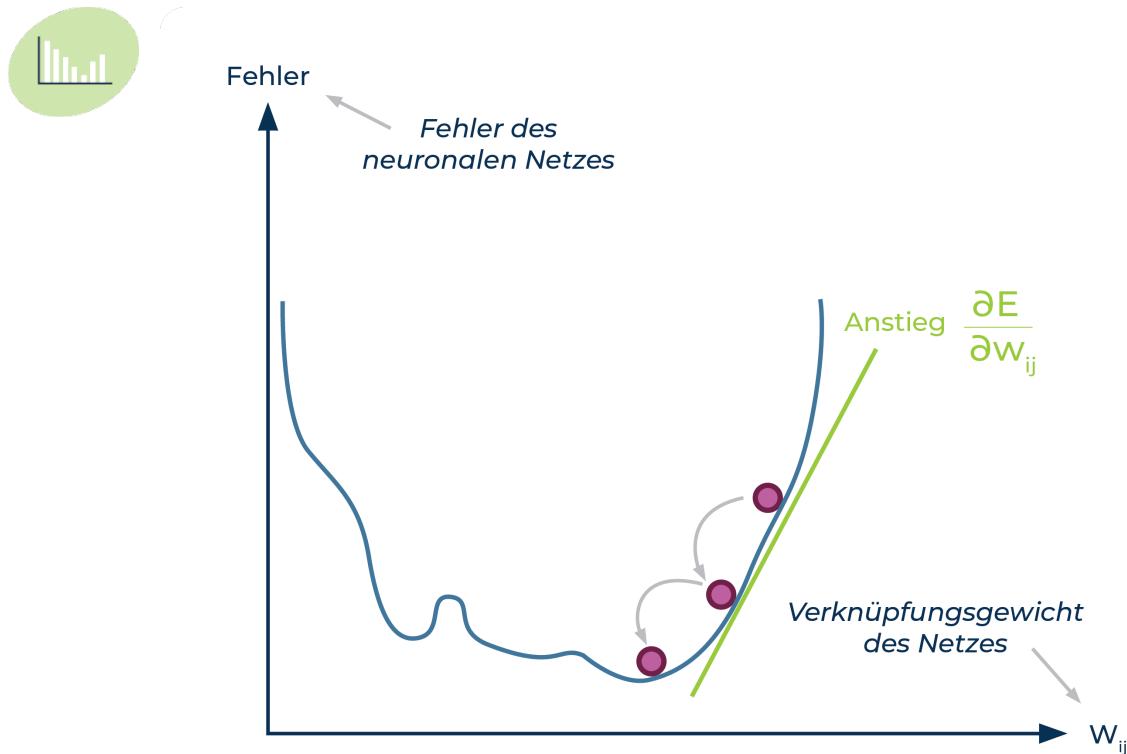
Der Fehler  $e_{output,1}$  teilt sich auf die Gewichte  $w_{11}$  und  $w_{21}$  auf. Analog teilt sich der Fehler am zweiten Neuron  $e_{output,2}$  auf die Gewichte  $w_{12}$  und  $w_{22}$  auf. Der Fehler am ersten Neuron der versteckten Schicht  $e_{hidden,1}$  ist also die Summe der anteiligen Fehler in allen Verknüpfungen, die nach vorn vom selben Knoten ausgehen. Mathematisch ausgeschrieben sieht der Fehler für  $e_{hidden,1}$  folgendermaßen aus:

$$e_{hidden,1} = e_{output,1} \cdot \frac{w_{11}}{w_{11} + w_{21}} + e_{output,2} \cdot \frac{w_{12}}{w_{12} + w_{22}}$$

### 3.4.2. Gradienten-Verfahren und Fehlerfunktion

Der erste Teil des Trainings war, den Fehler zu bestimmen und auf die einzelnen Schichten und deren Neuronen zurückzuführen. Dabei haben wir gesehen, dass sich die Fehler anteilig der Gewichte auf die Verknüpfungen aufteilen. Wie allerdings können wir diese Fehler dazu nutzen, um die Verknüpfungsgewichte zu aktualisieren – also das Neural Network zu trainieren?

Dafür müssen wir unsere Fragestellung etwas verfeinern. Denn was uns interessiert, ist, den Fehler beim nächsten Trainings-Durchlauf verkleinert zu haben. Wir müssen uns also fragen, welchen Einfluss ein einzelnes Gewicht auf den Fehler hat und für welchen Wert der Fehler minimal wird. Stellen wir uns dafür den Fehler in Abhängigkeit vom Gewicht  $w_{ij}$  als Kurve in einem 2D- Koordinatensystem vor.



Ziel ist es, das Minimum dieser Funktion zu bestimmen, also den Gewichtswert, für den der Fehler minimal wird. Das Minimum einer Funktion bestimmen wir normalerweise mit Hilfe einer simplen Ableitung, aber in unserem Fall ist das nicht möglich. Wir wissen nämlich nur, dass der Fehler vom jeweils betrachteten Gewicht abhängt, nicht allerdings wie er von diesem abhängt. Um also das Minimum dieser unbekannten Funktion zu bestimmen, müssen wir ein anderes Verfahren anwenden: das sogenannte Gradienten-Verfahren

Das Gradienten-Verfahren bezeichnet man auch als Verfahren des steilsten Abstiegs. Eine beliebte Analogie zur Erklärung des Gradienten-Verfahrens spielt in den Bergen – und sollte in der Praxis nicht so gemacht werden.

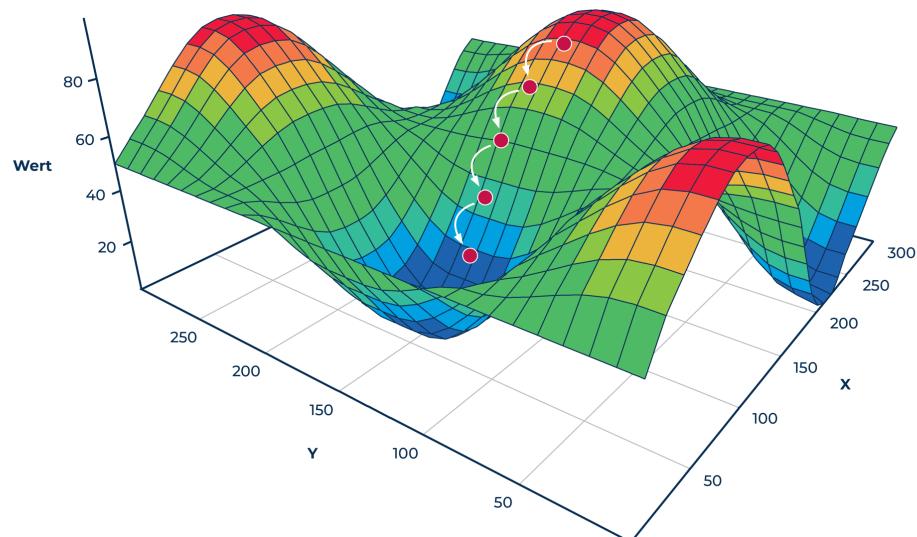
Stellen wir uns vor, wir stehen am Berg und suchen nach dem schnellsten Weg runter. Intuitiv bewegen wir uns also dorthin, wo es am steilsten



runtergeht, wo also die Steigung am höchsten ist (mathematisch: die Ableitung).

Wir beginnen damit, uns umzuschauen, und erkennen: „Ah, da links gehts steil runter.“ Dann landen wir in einer Senke und schauen uns noch mal nach dem steilsten Abschnitt um.

Genauso funktioniert das Gradienten-Verfahren in der Mathematik. Stellen wir uns eine Funktion vor, die von zwei Variablen abhängt und in einem 3-dimensionalen Koordinatensystem als wellenartiges Gebilde dargestellt wird:



Beim Gradienten-Verfahren startet man oberhalb eines Minimums, so wie in der Abbildung der rote Punkt im roten Bereich. Dieser möchte jetzt möglichst schnell nach unten in das blaue Tal. Der rote Punkt tastet sich Schritt für Schritt nach vorne, bis er im Tal – im Minimum – angekommen ist. Solche mathematischen Schritt-für-Schritt-Verfahren zur Lösung von



Problemstellungen bezeichnet man auch als numerische Verfahren. Das Gradienten-Verfahren eignet sich nicht nur hervorragend zur Bestimmung der Minima von komplexen Funktionen, sondern auch für die Bestimmung der Minima von Funktionen, die von sehr vielen Parametern abhängen. Und die Fehlerfunktion in einem Neural Network ist eine solche Funktion. Selbst wenn die Funktions-Vorschrift für die Abhängigkeit des Fehlers von den einzelnen Gewichten einfach wäre, würde die immense Anzahl an Abhängigkeiten es dennoch unmöglich machen, das Minimum mit rein analytischen Methoden zu bestimmen. Hier kommt dann die Numerik zur Rettung. Indem wir also das Gradienten-Verfahren auf die Fehlerfunktion anwenden, können wir in jedem Durchlauf die Gewichte sukzessive aktualisieren und damit das Netzwerk trainieren. Die Frage, die wir uns jetzt stellen, ist: wie sieht denn die Fehlerfunktion eigentlich aus?

$$E = (Output - Target)^2$$

Der Fehler soll ja die Abweichung von der Ausgabe des Neural Networks zur tatsächlich erwarteten Ausgabe quantifizieren. Da das Neural Network uns einen Output-Vektor ausspuckt, bietet es sich an, den erwarteten Output (Target) ebenfalls als Vektor zu schreiben. Oben haben wir bereits den Fehler als bloße Differenz zwischen dem Output und dem Target definiert. Das ist zwar besonders einfach, allerdings auch sehr fehleranfällig. Da man nämlich über die Fehler summiert, kann es passieren, dass sich zwei Summanden aufheben und dann einen Fehler von 0 suggerieren. Und das obwohl an zwei Neuronen eigentlich ein massiver Fehler aufgetreten ist! Sie waren einfach nur von entgegengesetztem Vorzeichen.

Eine weitere Möglichkeit für eine Fehlerfunktion ist der Absolutbetrag. In mathematischer Schreibweise würde der Fehler dann folgendermaßen aussehen:

$$E = |Output - Target|$$



Damit hätte man das Problem der bloßen Differenz gelöst, weil sich hier Fehlerwerte nicht einfach aufheben können. Allerdings ist bei der Betragsfunktion die Differenzierbarkeit ein Problem. Für  $x=0$  lässt sich nämlich keine Ableitung für die Betragsfunktion berechnen, da sie an dieser Stelle einen unstetigen Knick macht. Besser geeignet dagegen ist das Quadrat der Differenz:

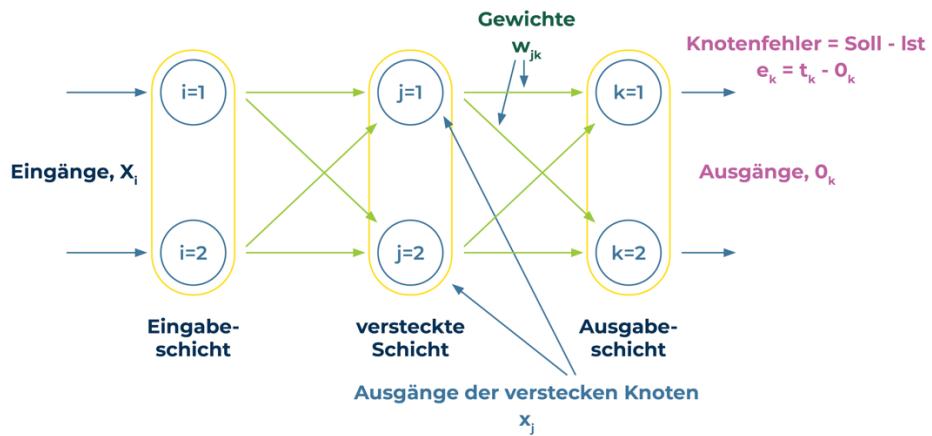
Es gibt mehrere Gründe, wieso diese Art der Fehlerfunktion besonders gut geeignet ist:

- Die Funktion ist stetig, wodurch das Gradienten-Verfahren besser funktioniert.
- Die erforderliche Mathematik zur Differentiation ist noch bedeutend einfacher als bei anderen, komplexeren Fehlerfunktionen.
- In der Nähe des Minimums wird der Gradient kleiner, sodass sich die Wahrscheinlichkeit, im richtigen Minimum zu landen, erhöht.

Nun haben wir mit der Fehler-Backpropagierung, dem Gradienten-Verfahren und der richtigen Fehlerfunktion alle Grundlagen, um die Aktualisierung der Gewichte in eine anschauliche Formel zu packen.

### 3.4.3. Aktualisierung der Gewichte

Um die Gewichte nun zu aktualisieren, müssen wir in jedem Trainings-Schritt die Ableitung der Fehlerfunktion in Abhängigkeit vom Gewicht der Verknüpfung bestimmen. Nehmen wir zur Veranschaulichung wieder an, wir hätten ein drei-schichtiges Neural Network mit zwei Neuronen pro Schicht. Die Eingabeschicht bekommt den Index  $i$ , die versteckte Schicht den Index  $j$  und die Ausgabeschicht den Index  $k$ . Als erstes sind wir daran interessiert, die Gewichte zwischen der versteckten und der Ausgabeschicht zu aktualisieren, also die Gewichte  $w_{jk}$ .



Dafür müssen wir die Abhängigkeit des Fehlers – der quadratischen Differenz zwischen Target und Output – vom Gewicht  $w_{jk}$  bestimmen:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} (t_k - o_k)^2$$

Diesen Ausdruck können wir zunächst einmal mit der Kettenregel vereinfachen:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial o_k} \frac{\partial o_k}{\partial w_{jk}}$$

Diese angewandt erhalten wir dann als Ergebnis für die Gleichung:

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot \frac{\partial o_k}{\partial w_{jk}}$$

Um also die Ableitung der Fehlerfunktion mit Bezug auf das Gewicht  $w_{jk}$  bestimmen zu können, müssen wir die Ableitung des Outputs  $o_k$  berechnen. Für diesen gilt die Vorschrift:

$$o_k = \text{sigmoid}(\sum_j w_{jk} o_j)$$



In dieser Formel ist mit  $o_j$  der Output aus der versteckten Schicht gemeint.  
Für die Ableitung der Sigmoid-Funktion gibt es eine einfache Formel:

$$\frac{\partial}{\partial x} \text{sigmoid}(x) = \text{sigmoid}(x)(1 - \text{sigmoid}(x))$$

Wenn wir das oben einsetzen, erhalten wir:

$$\begin{aligned} \frac{\partial E}{\partial w_{jk}} &= -2(t_k - o_k) \cdot (\text{sigmoid}(\sum_j w_{jk} o_j)) \cdot (1 - (\text{sigmoid}(\sum_j w_{jk} o_j))) \cdot \\ &\quad \frac{\partial}{\partial w_{jk}} (\sum_j w_{jk} o_j) \end{aligned}$$

Für den letzten Faktor gilt:

$$\frac{\partial}{\partial w_{jk}} (\sum_j w_{jk} o_j) = o_j$$

Da die Outputs der versteckten Schicht nicht von den Gewichten zwischen der versteckten und der Ausgabeschicht abhängen, vereinfacht sich die Formel für die Ableitung der Fehlerfunktion zu:

$$\frac{\partial E}{\partial w_{jk}} = 2e_k \cdot (\text{sigmoid}(\sum_j w_{jk} o_j)) \cdot (1 - (\text{sigmoid}(\sum_j w_{jk} o_j))) \cdot o_j$$

wobei:

$$e_k = (t_k - o_k)$$



Um die Gewichte zwischen der Eingabeschicht und der versteckten Schicht zu aktualisieren, benötigen wir die backpropagierten Fehler in der versteckten Schicht. Die obige Formel passt sich dann folgendermaßen an:

$$\frac{\partial E}{\partial w_{ij}} = 2e_j \cdot (\text{sigmoid}(\sum_i w_{ij}o_i)) \cdot (1 - (\text{sigmoid}(\sum_i w_{ij}o_i))) \cdot o_i$$

Das aktualisierte Gewicht erhalten wir dann mit der folgenden Formel:

$$w_{jk,\text{neu}} = w_{jk,\text{alt}} - \alpha \frac{\partial E}{\partial w_{jk}}$$

Wir subtrahieren also vom alten Gewicht die Ableitung des Fehlers, multipliziert mit einem Faktor, den wir die Lernrate des Neural Networks

$$\Delta W_{jk} = \alpha \cdot E_k \cdot O_k (1 - O_k) \cdot O_j^T$$

nennen. Der Fehler-Term wird hier subtrahiert, weil der Wert des Gewichts bei einem positiven Anstieg der Ableitung verringert und bei einem negativen Anstieg vergrößert werden soll. Den Ausdruck für die Ableitung des Fehler-Terms können wir noch übersichtlicher in Matrixform schreiben:

$$\Delta W_{jk} = \alpha \cdot E_k \cdot O_k (1 - O_k) \cdot O_j^T$$

Hierbei ist  $E_k$  der Fehler-Vektor an der Ausgabeschicht,  $O_k$  der Output-Vektor und der transponierte Vektor  $O_j$  steht für den Output aus der versteckten Schicht. Diese Formel werden wir später zum Training des Neural Networks verwenden.

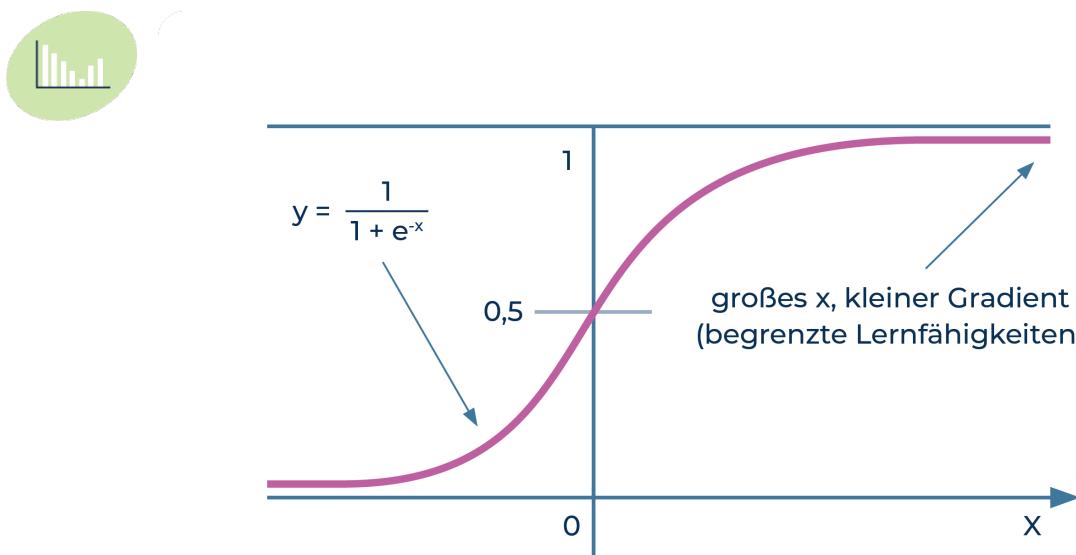


### 3.4.4. Vorbereitung der Daten

Bevor wir uns nun an die eigentliche Programmierung des Neural Networks machen können, müssen wir uns noch damit beschäftigen, wie wir die Daten für das Training vorbereiten. Um genauer zu sein, müssen wir die Eingabe- und die Ausgabedaten skalieren und die anfänglichen Werte für die Verknüpfungsgewichte festlegen. Diese Vorbereitung der Daten ist wichtig, damit der Trainingsprozess gute Aussichten auf eine erfolgreiche Durchführung hat. Neural Networks funktionieren dann am besten, wenn Eingabe-, Ausgabe- und Gewichtsdaten auf das Netz-Design und das zu lösende Problem abgestimmt sind. Schauen wir uns die drei Komponenten der Reihe nach an.

#### Eingaben

Um zu verstehen, wie wir die Eingabedaten vorbereiten sollten, werfen wir noch einmal einen Blick auf die Sigmoid-Funktion.



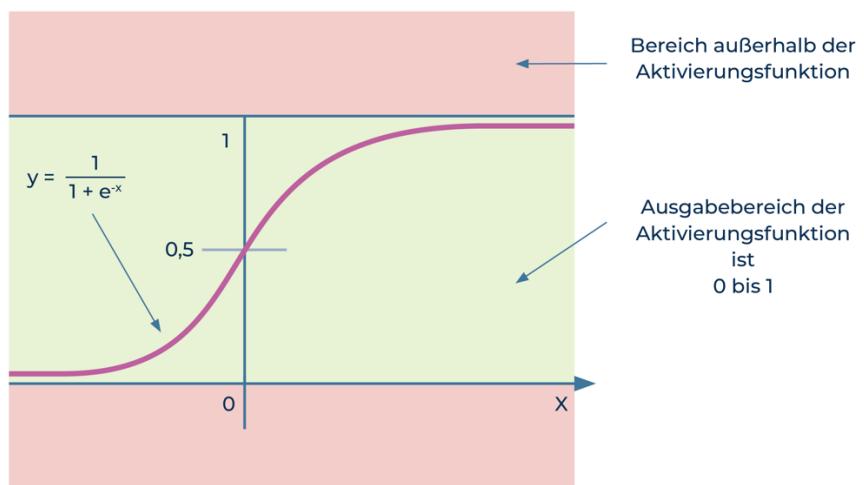
Anhand des Graphen können wir zunächst einmal erkennen, dass sich die Werte der Sigmoid-Funktion zwischen 0 und 1 bewegen. Zusätzlich können



wir erkennen, dass die Kurve für sehr große (und auch für sehr kleine) Werte abflacht. Eine flache Aktivierungsfunktion würde sich aber beim Einsatz des Gradienten-Verfahrens als problematisch herausstellen. Denn wenn wir uns noch einmal den Ausdruck für die Aktualisierung der Gewichte anschauen, können wir sehen, dass die Ableitung von der Aktivierungsfunktion abhängt. Wenn die Ableitung klein ist, bedeutet das, dass man die Lernfähigkeit des Neural Network begrenzt hält. Wir sollten also die Eingabewerte möglichst klein halten.

Allerdings sollte man die Eingabewerte auch nicht zu klein skalieren, da je nach Datentyp bei der Verarbeitung sehr kleiner Zahlen Genauigkeit verloren gehen kann. Idealerweise wählt man für die Sigmoid-Funktion einen Skalenbereich zwischen 0 und 1. Da Nulleingaben allerdings störend für das Netz sein und dessen Lernfähigkeit abwürgen können, beginnt man die Skala üblicherweise bei 0.01.

## Ausgaben





Mit den Ausgaben eines Neural Networks sind die Signale gemeint, die aus den Knoten der letzten Schicht austreten. Da unsere Aktivierungsfunktion keine Werte liefern kann, die größer als 1 sind, sollten wir die Trainingsdaten auch nicht mit Zielwerten größer als 1 einspeichern. Würden wir die Target-Werte in einen Bereich außerhalb des Intervalls [0,1] legen, würde das Training des Netzes zu immer größeren Gewichten führen, da es immer größere Ausgabewerte erzeugen will. Da die Aktivierungsfunktion diese Werte allerdings niemals liefern kann, ist das Netz gesättigt und funktioniert damit schlecht.

Um das zu vermeiden, müssen wir also auch die Target-Werte auf einen Bereich zwischen 0 und 1 skalieren. Allerdings konzentriert man sich hier in der Praxis auf das Intervall [0.01,0.99], da sowohl 0 als auch 1 unmögliche Target-Werte sind. Diese in die Berechnungen miteinzubeziehen, würde das Risiko mit sich bringen, die Gewichte zu stark zu vergrößern.

### Zufällige Anfangswerte für die Gewichte

Hier greift das gleiche Argument wie bei den Eingaben und Ausgaben. Große Anfangsgewichte sollten wir vermeiden, weil dadurch große Signale die Aktivierungsfunktion schnell zur Sättigung bringen können. Eine erste Idee wäre, die Anfangsgewichte aus einem Bereich von -1.0 bis +1.0 zufällig und gleichverteilt zu wählen. Mathematiker\*innen und Informatiker\*innen haben sich tatsächlich aber eine noch bessere Idee ausgedacht.

Ihre Faustregel besagt, dass sich die Gewichte bei der Initialisierung des Trainings in einem Bereich befinden sollen, der in etwa dem Kehrwert aus der Quadratwurzel der Anzahl der Verknüpfungen zu einem Neuron entspricht.

Das klang jetzt unheimlich kompliziert, dröseln wir das also auf.



Zuerst einmal schauen wir auf den Ausdruck „Anzahl der Verknüpfungen zu einem Neuron“. Damit ist die Anzahl der Verknüpfungen gemeint, die in ein Neuron hineingehen. Wenn wir also in einem drei-schichtigen Neural Network mit drei Neuronen pro Schicht unterwegs sind, dann gehen in jedes Neuron aus der versteckten und der Ausgabeschicht drei Verknüpfungen aus der jeweils vorherigen Schicht ein.

Diese Anzahl der Verknüpfungen soll nun irgendwie mit der Quadratwurzel und dem Kehrwert verwendet werden, um den optimalen Wertebereich für die Gewichtsmatrizen zu bestimmen. Wir nehmen also für das drei-schichtige Neural Network den Bereich

$$\left[-\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right]$$

oder in Zahlen ausgedrückt [-0.577,0.577].



Wichtig bei der Wahl der Anfangsgewichte ist allerdings, sie nicht auf 0 zu setzen! Nullgewichte blockieren das Eingangssignal, weswegen die Funktion zur Gewichtsaktualisierung – die von den Eingangssignalen abhängt – nur noch Nullwerte liefern würde. Somit wäre es nicht mehr möglich, die Gewichte zu aktualisieren.



### 3.5. Programmierung eines Neural Networks

Dieses Kapitel enthält ausschließlich Übungsaufgaben. Diese sind in der Kursstrecke auf der Lernplattform direkt hinterlegt.

### 3.6. Neural Networks mit PyTorch

Bis jetzt haben wir gelernt, wie Neural Networks im Prinzip funktionieren und wie wir ein einfaches Netz in Python mit relativ wenig Code implementieren können. Allerdings wird es mit dem Code, den wir geschrieben haben, schwieriger, komplexe Netze mit mehreren Schichten aufzubauen oder verschiedene Parameter beim Training zu tunen. Daher verwendet man in der Praxis vorprogrammierte Bibliotheken, deren Module man übernehmen kann, um seine eigenen Netze für die eigenen Projekte zu konstruieren. Es gibt verschiedene Python-Bibliotheken, um das zu machen, die bekanntesten sind wohl TensorFlow und PyTorch. Wir werden uns nun mit letzterer beschäftigen und lernen, wie wir komplexe Neural Networks konstruieren können. Um das Gerüst für ein Neural Network mit PyTorch aufzubauen, müssen wir folgende Schritte ausführen:

1. Importieren der relevanten Bibliotheken
2. Erstellen einer Neural-Network-Klasse
3. Festlegen der Hyperparameter
4. Importieren der Daten
5. Initialisierung des Netzes
6. Festlegen der Loss-Function und des Optimierungsverfahrens
7. Training des Neural Networks
8. Testen des Neural Networks
9. Speichern des trainierten Modells



Der Workflow ist bei fast jedem Data-Science-Projekt, welches Neural Networks einsetzt, der gleiche. Schauen wir uns also jeden einzelnen Schritt nun im Detail an.

### Importieren der relevanten Bibliotheken

Wie bei jedem Projekt müssen wir als erstes die Bibliotheken importieren, deren Funktionen wir im Laufe des Codes verwenden wollen. PyTorch enthält zahlreiche Bibliotheken und Unter-Bibliotheken, die zur Konstruktion und zur Optimierung von Neural Networks verwendet werden oder um PyTorch-interne Daten zu importieren, die zum Training genutzt werden können. Bevor wir allerdings all diese Bibliotheken nutzen können, müssen wir PyTorch auf unserem Rechner installieren. Das geht wieder recht simpel mit dem pip-Befehl, mit welchem wir die torch- und die torchvision-Bibliothek importieren.

```
# Python 3.x
pip3 install torch torchvision
```

Weitere Informationen zur Installation von PyTorch lassen sich auf der Website <https://pytorch.org/get-started/locally/> finden. Nachdem der Installationsprozess durchgelaufen ist, können wir mit dem Importieren starten.

```
# Importieren der wichtigen Module
import time
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
import torchvision
import torchvision.transforms as transforms
import numpy as np
import matplotlib.pyplot as plt
```



Wir können sehen, dass wir eine Menge Bibliotheken importieren. Nicht alle Module stammen ursprünglich von PyTorch. Gehen wir die Liste einfach der Reihe nach durch.

Das erste importierte Modul ist das time-Modul. Mit diesem werden wir die Zeit messen, die das Skript zur Ausführung des Codes benötigt. Dafür schreiben wir zu Beginn des Codes die folgende Zeile:

```
# Run Time messen  
start = time.time()
```

Am Ende des Skripts schreiben wir dann noch mal die folgenden zwei Zeilen:

```
# Runtime messen  
end = time.time()  
print("Runtime: ", end-start, " s")
```

Sowohl zu Beginn als auch am Ende des Skripts wird die jeweilige Zeit in einer Variable gespeichert und die Differenz zwischen den beiden ist dann die Runtime des Skripts in Sekunden.

Nach der time-Bibliothek kommt dann endlich alles, was mit PyTorch zusammenhängt. Als erstes importieren wir das torch-Modul, das eigentlich alles enthält, was wir brauchen. Doch um uns später im Code Schreibarbeit zu sparen, importieren wir auch noch explizit ein paar Unter-Module, von denen wir wissen, dass wir sie in jedem Projekt nutzen werden. Deshalb wird im nächsten Schritt das Unter-Modul torch.nn importiert, wobei wir dieses später im Code mit nn abkürzen werden. Dieses Modul ist wichtig, um nachher die NeuralNetwork-Klasse zu definieren. Die Klasse, die wir



definieren, wird nämlich alle Eigenschaften vom nn-Modul erben, sodass wir nur noch ein paar grundlegende Spezifikationen für unser eigenes Projekt festlegen müssen.

Als nächstes importieren wir das torch.optim-Modul, welches die Optimierungs-Algorithmen enthält. Im letzten Abschnitt haben wir für die Backpropagierung der Fehler das Gradienten- Verfahren als Optimierungsalgorithmus verwendet. Tatsächlich gibt es noch eine Reihe weiterer Optimierungsalgorithmen, die häufig in der Praxis eingesetzt werden und im torch.optim-Modul integriert sind.

Im nächsten Schritt importieren wir die beiden Module DataLoader und Dataset, die beide Bestandteile des torch.utils.data-Moduls sind. Das Dataset-Modul werden wir benötigen, um unsere Daten (ob CSV- oder andere Typen von Files) in den Code zu integrieren. Das DataLoader- Modul brauchen wir dann, um die integrierten Daten in ein für PyTorch verständliches Format zu packen – dem sogenannten DataLoader.

Nachdem diese Bibliotheken alle importiert sind, kommt die torchvision-Bibliothek und mit ihr das Modul torchvision.transforms, welches wir brauchen, um die importierten Daten in sogenannte Tensoren zu verwandeln. Bei Tensoren handelt es sich um eine spezielle Form von Array, welches im PyTorch-Umfeld genutzt wird, um die Berechnungen in Neural Networks effizienter zu gestalten.

Damit haben wir alle PyTorch-Module importiert, jetzt kommen zu guter Letzt nur noch numpy und matplotlib hinzu, die wir bereits aus den vorherigen Kapiteln kennen.

## Die Neural-Network-Klasse



Nun, da wir alle erforderlichen Bibliotheken importiert haben, können wir mit der Definition der Neural-Network-Klasse beginnen. Schauen wir uns

```
# Erstellen einer Neural Network Klasse

# Die Klasse NN erbt von nn.Module die Eigenschaften
class NN(nn.Module):
    def __init__(self):
        super(NN, self).__init__()
        self.flatten = nn.Flatten()
        self.network = nn.Sequential(
            nn.Linear(784, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.network(x)
        return logits
```

zunächst einmal den fertigen Code an und gehen diesen dann Zeile für Zeile durch, um ihn zu verstehen:

In der ersten Zeile definieren wir die Klasse NN (kurz für NeuralNetwork) und spezifizieren in der Klammer, dass diese Klasse die Methoden und Attribute von nn.Module übernehmen soll. In dieser Klasse werden zwei Methoden definiert: die `__init__()`-Methode und die `forward()`- Methode. Erstere setzt das Neural Network auf: wie viele Schichten es haben soll und wie viele Knoten pro Schicht existieren. Mit der `forward()`-Methode wird dann die Signalübertragung von der Eingabeschicht zur Ausgabeschicht ausgeführt. Schauen wir uns beide Methoden mal genauer an.

Die `__init__()`-Methode hat als Argument zunächst einmal nur `self`, sonst keine weiteren Parameter. Man könnte der `__init__()`-Methode auch die Spezifikationen der Schichten als Parameter geben (Anzahl der Knoten in der Eingabe- und Ausgabeschicht), allerdings legen wir in diesem Beispiel die Parameter manuell fest. Das Erste, was die `__init__()`-Methode macht, ist



die sogenannte `super()`-Funktion aufzurufen, welche der neuen Klasse `NN` die Parameter und Methoden der Elternklasse `nn.Module` übergibt: Die `super()`-Methode ist also die Ausführung des Erbprozesses. Als Rückgabewert erhält man ein Objekt, welches die Elternklasse repräsentiert.

In der nächsten Zeile definieren wir die `Flatten()`-Funktion, die ebenfalls ein Bestandteil des `nn`-Moduls ist. Die `Flatten()`-Funktion wird eingesetzt, um die Bilder, welche als Zahlenwerte im 28x28-Format kommen, in ein einzelnes Array von 784 Werten umzuwandeln.

Nun können wir das Netz konstruieren. Dafür verwenden wir den sogenannten `nn.Sequential`-Container. Dieser Container enthält alle wichtigen Module, und die Daten werden der Reihe nach durch diesen Container geleitet. Wir können also mit `Sequential()` recht simpel und schnell ein Netz mit Schichten und Aktivierungsfunktionen konstruieren. Genau das passiert in den folgenden Zeilen.

Mit `nn.Linear(784, 512)` legen wir die Beziehung zwischen der Eingabeschicht und der versteckten Schicht fest. Die Eingabeschicht hat 784 Neuronen (entsprechend der 784 Pixel in den Daten) und die versteckte Schicht soll 512 Neuronen haben. Da beim Übergang von einer Schicht zur nächsten die Daten mittels Matrixmultiplikation verarbeitet werden, spricht man von einer linearen Transformation – weswegen die Methode auch `nn.Linear` heißt.

In der nächsten Zeile legen wir die Aktivierungsfunktion für die versteckte Schicht fest. In diesem Fall wählen wir die `ReLU()`-Funktion, eine Alternative zur Sigmoid-Funktion, die besonders gerne wegen ihrer simplen Form in Neural-Network-Projekten verwendet wird. Wie diese Aktivierungsfunktion genau funktioniert und welche anderen Funktionen es in PyTorch gibt, werden wir uns später ansehen.



Nachdem wir die Aktivierungsfunktion festgelegt haben, kommt der Übergang zur nächsten versteckten Schicht, welche wieder 512 Neuronen haben soll. Dann wird wieder die ReLU()-Aktivierungsfunktion angewendet und anschließend konstruiert man den Übergang von der letzten versteckten Schicht zur Ausgangsschicht.

Damit haben wir nun alle wichtigen Parameter für das Neural Network festgelegt und können uns nun der forward()-Methode widmen. In dieser wenden wir zunächst self.flatten() auf die Eingabedaten an und lassen schließlich die Daten durch die einzelnen Schichten im Sequential()-Container laufen. Das Ergebnis (der Output des Netzes) wird in der Variable logits gespeichert und zurückgegeben. Damit haben wir die Klasse für das Neural Network definiert und können nachher eine Instanz davon erzeugen.

### Hyperparameter festlegen

Bevor wir die Daten importieren können, müssen wir ein paar Hyperparameter für das Neural- Network-Modell formulieren. Das geht recht fix in wenigen Zeilen. Wir können – wenn wir die input\_size und die output\_size in dem Sequential()-Container definieren – die ersten Zeilen weglassen. Danach kommt lediglich eine Deklarierung der Lernrate und der Anzahl der Epochen, die das Training laufen soll. Später werden wir auch noch einen weiteren Hyperparameter – die sogenannte Batch Size – kennenlernen.

```
# Hyperparameter festlegen

input_size = 784 #Anzahl der Pixel eines jeden Bildes
output_size = 10 #Die Daten lassen sich in 10 Klassen einteilen
learning_rate = 0.0001
num_epochs = 5
```

### Importieren der Daten



Zunächst müssen wir die Daten aus dem CSV-File importieren. Hierfür können wir die pandas- Bibliothek verwenden, wir können aber auch zur Abwechslung mal das Ganze mit der numpy- Bibliothek durchziehen. Hierfür schreiben wir folgende Zeile in unser Skript:

```
# Daten importieren
xy = np.loadtxt("mnist_data.csv",
                 delimiter=",", dtype=np.float32, skiprows=1)
```

Mit dem Parameter delimiter legen wir fest, dass die einzelnen Datenwerte durch ein Komma getrennt sind. Mit dem nächsten Parameter legen wir fest, dass es sich bei den Werten um Gleitkommazahlen handeln soll. Mit dem letzten Parameter sagen wir dem Skript, dass die erste Zeile beim Import ausgelassen werden soll, weil diese nur die Spaltennamen enthält. Diese Zeile können wir nachher in der Data-Klasse verwenden, um die Daten zu importieren und in ein für PyTorch verständliches Objekt zu verwandeln. Der Code für die Klassendefinition sieht folgendermaßen aus:

```
class Data(Dataset):
    def __init__(self):
        xy = np.loadtxt("mnist_data.csv",
                       delimiter=",", dtype=np.float32, skiprows=1)
        self.x = torch.from_numpy(xy[:, 1:])
        self.y = torch.from_numpy(xy[:, 0])
        self.n_samples = xy.shape[0]

    def __getitem__(self, index):
        return self.x[index], self.y[index]

    pass

    def __len__(self):
        return self.n_samples
```

Bei der obigen Klassendefinition können wir auch gleich sehen, wieso wir zu Beginn des Skripts die Klasse Dataset aus torch.utils.data importiert haben. Die Data()-Klasse soll nämlich von dieser PyTorch-Klasse alle Parameter und Methoden erben. Somit müssen wir nur noch drei neue Methoden in der



Klasse formulieren: die `__init__()`-Methode, die `__getitem__()`-Methode und die `__len__()`-Methode. Schauen wir uns an, was diese Methoden genau machen.

Die `__init__()`-Methode ist wieder da, um die grundlegenden Eigenschaften der Klasse zu initialisieren. Dafür importieren wir als erstes die Daten aus dem CSV-File in ein numpy-Array. Dieses teilen wir dann in Features und Labels auf. Dafür müssen wir das Array `xy` in einen Tensor umwandeln, wobei wir den Befehl `torch.from_numpy()` anwenden. Was genau Tensoren sind und wie sie eingesetzt werden, sehen wir später. Um den Code zum Laufen zu bringen, reicht an dieser Stelle erst einmal zu wissen, dass die Daten in einen Tensor umgewandelt werden müssen. Mit der folgenden Zeile sagen wir, dass die Variable `x` alle Spalten ab der ersten enthalten soll (es wird bei 0 mit dem Zählen angefangen). Das heißt die Variable `x` wird die Features enthalten.

```
self.x = torch.from_numpy(xy[:, 1:])
```

Mit der nächsten Zeile wählen wir die Labels als die erste Spalte des Tensors aus:

```
self.y = torch.from_numpy(xy[:, 0])
```

Daraufhin legen wir die Anzahl der enthaltenen Daten über die `shape()`-Funktion fest und speichern sie in der Variable `n_samples`. Damit sind alle wichtigen Attribute für die Daten festgelegt.

Die nächsten beiden Methoden sind wichtig, um mit den Daten hantieren zu können. Mit `__getitem__()` lässt man sich den Datenwert an einem bestimmten Index ausgeben, weswegen die Funktion auch zusätzlich zu `self` noch den `index` als Parameter enthält. Die Methode `__len__()` soll lediglich die Länge des Datensatzes als Rückgabewert liefern. Mit diesen drei Methoden haben wir die `Data()`-Klasse vollständig formuliert und

```
dataset = Data()
dataloader = DataLoader(dataset=dataset, shuffle = True)
```



können nun weiter fortfahren, sie in einen sogenannten DataLoader zu packen:

Zuerst erzeugen wir eine Instanz der Data()-Klasse und speichern das Objekt in der Variable dataset. Dieses Objekt (das unsere Daten enthält) packen wir dann in den DataLoader, welcher das abstrakte Datenobjekt iterierbar und somit verwendbar für das Training später macht. Der zweite Parameter shuffle=True legt fest, dass die Daten gemischt werden sollen, um später beim Training einen möglichen Bias aus dem Weg zu räumen. Nun haben wir praktisch alles an Vorarbeit erledigt, um das Netz trainieren zu können. Das Letzte, was wir vor einer Initialisierung des Netzes noch erledigen müssen, ist, den Datensatz in ein Trainings- und ein Testset aufzuteilen.

## Training des Neural Networks

Nachdem wir nun fast sämtliche Vorarbeiten erledigt haben, können wir uns dem Training des Neural Networks widmen. Dafür werden wir zuerst das Netz initialisieren müssen, welches wir mit der folgenden Zeile erreichen:

```
# Initialisierung des neuronalen Netzes  
model = NN(input_size, output_size)
```

Wir erzeugen also ein Objekt der Klasse NN mit den Parametern input\_size und output\_size. Dieses Objekt ist unser Neural Network. Damit wir es auch adäquat trainieren können, müssen wir noch die Fehlerfunktion (Loss Function) und das Optimierungsverfahren festlegen.



```
# Loss Function und Optimierungsverfahren festlegen
# Vorher war es die quadratische Abweichung und das Gradientenverfahren
loss_function = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
```

Als Fehlerfunktion verwenden wir diesmal nicht den quadratischen Fehler, sondern den Cross Entropy Loss, den wir bereits im Kapitel über Machine Learning kennengelernt haben. Die `CrossEntropyLoss()`-Funktion ist in der `nn`-Klasse implementiert und kann somit in einer simplen Zeile aufgerufen werden. Als Optimierungsverfahren verwenden den sogenannten Stochastic Gradient Descent, eine Weiterentwicklung des Gradientenverfahrens, welches wir bereits kennengelernt und eingesetzt haben. Die Stochastic Gradient Descent Funktion wird durch `SGD()` abgekürzt und ist im `optim`-Modul implementiert. Die Funktion benötigt vom NN-Objekt die Model-Parameter und natürlich auch die Lernrate, um Schritt für Schritt die Gewichte zwischen den Schichten zu aktualisieren. Jetzt kann das Training losgehen, wofür wir zwei ineinander verschachtelte `for`-Schleifen verwenden werden:

```
# Training des neuronalen Netzes
for epoch in range(num_epochs):
    for idx, (data,labels) in enumerate(train_loader):
        labels = labels.type(torch.LongTensor)

        # Daten durchs Netz jagen
        outputs = model(data)
        loss = loss_function(outputs,labels)

        # Backpropagierung
        optimizer.zero_grad() #Am Beginn jedes Schleifendurchlaufs Gradienten auf 0
        loss.backward()
        optimizer.step() # Gradientenverfahren

        if idx%100 == 0:
            loss, current = loss.item(), idx * len(data)
            print(f"loss:{loss} Durchlauf: {current}")
```

Mit der ersten `for`-Schleife legen wir den Durchlauf für jede Epoche fest. Die zweite `for`-Schleife ist etwas komplizierter: hier haben wir plötzlich die 3 Variablen `idx`, `data` und `labels`, wobei wir die letzten beiden im Tupel `(data, labels)` zusammenfassen. Und dann kommt da noch die `enumerate()`-



Funktion mit dem Trainingsdatensatz trainloader hinzufügen. Was hat das zu bedeuten?

Fangen wir mit der enumerate()-Funktion und den Trainingsdaten an. Unsere Trainingsdaten haben wir ja vorher in einen DataLoader gepackt, weil dieser ein iterierbares Objekt ist. Da sich allerdings die Daten im DataLoader mit jedem Durchlauf ändern, wollen wir ihnen eine Nummerierung geben. Und das machen wir mit der enumerate()-Funktion. Damit erklärt sich dann auch die Verwendung der Schleifen-Variable idx, welche für Index steht. Dank der enumerate()-Funktion können wir nämlich genau nachvollziehen, welcher Index gerade vom Training behandelt wird. Das Tupel (data, labels) steht für die Daten und ihre Klassifizierungen. Mit der zweiten Schleife gehen wir also jedes einzelne Daten-Sample im Trainingsdatensatz durch und trainieren mit diesem das Netz. Schauen wir uns nun an, wie das Training genau abläuft.

Als erstes müssen wir den Datentyp der Labels in Long transformieren, wofür wir die folgende Zeile schreiben:

```
labels = labels.type(torch.LongTensor)
```

Dann können wir für die einzelnen Daten im Datensatz den Output der Signalübertragung berechnen:

```
outputs = model(data)
```



Anschließend berechnen wir den Fehler mit der oben definierten Fehlerfunktion, welche die Outputs aus der Signalübertragung und die entsprechenden Labels benötigt. Nachdem die Fehler berechnet wurden, beginnt die Backpropagierung der Fehler. Diese teilt sich in 3 Schritte auf.

- Zu Beginn von jedem Schleifen Durchlauf soll der Gradient auf Null gesetzt werden, um das Training mit den folgenden Daten nicht zu beeinflussen.
- Im 2. Schritt wenden wir die backward()-Funktion an, die eine Methode des loss-Objekts ist. Damit verfolgen wir die Fehler durch die Schichten zurück.
- Im letzten Schritt wenden wir das ausgewählte Optimierungsverfahren an.

Im letzten Schritt lassen wir uns jeden hundertsten Durchlauf den jeweiligen Fehler und die Durchlauf-Nummer ausgeben. Damit können wir während des Trainings bereits beobachten, wie sich der Fehler verändert (und hoffentlich kleiner wird).

## Tensoren

Nun, da wir das Neural Network zum Laufen gebracht haben, können wir uns mit den Details der einzelnen PyTorch-Funktionalitäten auseinandersetzen. Als erstes werden wir uns mit sogenannten Tensoren beschäftigen. Bei Tensoren handelt es sich um eine spezielle Datenstruktur, die im Prinzip wie Arrays oder Matrizen funktionieren. Tensoren werden in PyTorch verwendet, um die Eingänge, die Ausgänge und auch die Parameter des trainierten Modells in eine einheitliche Form zu bringen. Der wesentliche Vorteil von Tensoren gegenüber Arrays liegt in ihrer Fähigkeit,



auch auf GPUs anstatt klassisch auf den CPUs laufen zu können, wodurch die Effizienz des Trainings gesteigert wird. Gleichzeitig ist dieser Datentyp optimiert für die Berechnung von Ableitungen, was vor allem bei Optimierungsverfahren wie dem Gradient Descent von unglaublichem Nutzen ist. Schauen wir uns einfach mal anhand ein paar praktischer Beispiele an, wie wir Tensoren initialisieren können, welche Eigenschaften sie haben und welche Operationen man auf ihnen ausführen kann.

Beginnen wir mit der Initialisierung eines Tensors. Hier gibt es prinzipiell drei Möglichkeiten:

- Wir erzeugen einen Tensor direkt aus den Daten, die uns vorliegen.
- Wir transformieren ein numpy-Array in einen Tensor.
- Wir erzeugen einen Tensor mit zufälligen oder konstanten Werten.

Um einen Tensor zu erzeugen, müssen wir natürlich die torch-Bibliothek importieren. Für die Umwandlung eines numpy-Arrays in einen Tensor brauchen wir auch noch die numpy-Bibliothek:

```
import torch  
import numpy as np
```

Nehmen wir beispielsweise an, dass die Daten in der folgenden recht simplen Form vorliegen:

```
data = [[1, 2], [3, 4]]
```

Möchten wir diese Daten in einen Tensor verwandeln, dann schaffen wir das mit der `torch.tensor()`-Methode, welche als Argument die folgenden Daten bekommen wird:

```
x_data = torch.tensor(data)
```



Transformieren wir nun die obigen Daten in ein numpy-Array:

```
np_array = np.array(data)
```

Wenn wir diese in einen Tensor transformieren wollen, benötigen wir die `torch.from_numpy()`-Methode:

```
x_np = torch.from_numpy(np_array)
```

Eine andere Möglichkeit, um einen Tensor zu erzeugen, ist die Verwendung von zufälligen oder konstanten Werten. Um einen solchen Tensor zu erzeugen, müssen wir erst die Form (shape) des Tensors festlegen. Für unser Beispiel wählen wir die folgende Form:

```
shape = (2,3,)
```

Das bedeutet, dass der Tensor die Form einer 2x3-Matrix annimmt. Wollen wir nun einen Tensor mit zufälligen Werten haben, verwenden wir die `torch.rand()`-Methode, welche als Argument die Form des Tensors bekommt, die wir oben in der Variable `shape` deklariert haben:

```
rand_tensor = torch.rand(shape)
```

Das Ergebnis könnte dann so aussehen:

```
tensor([[0.8029, 0.2701, 0.3133],  
        [0.0959, 0.5825, 0.5655]])
```

Möchten wir aber den Tensor nicht mit zufälligen Werten, sondern mit konstanten Werten wie zum Beispiel 0 oder 1 füllen, dann erreichen wir das mit den Methoden `torch.ones()` und `torch.zeros()`:

```
ones_tensor = torch.ones(shape)  
zeros_tensor = torch.zeros(shape)
```



Wenn wir uns beide Tensoren mit `print()` ausgeben lassen, erhalten wir:

```
tensor([[1., 1., 1.],  
       [1., 1., 1.]])  
  
tensor([[0., 0., 0.],  
       [0., 0., 0.]])
```

Tensoren können drei verschiedene Eigenschaften haben:

- Ihre Form (shape)
- Der enthaltene Datentyp
- Der Speicherort (GPU oder CPU)

Um zu sehen, wie wir diese Eigenschaften ausgeben können, werfen wir wieder einen Blick auf den oben definierten Tensor mit zufälligen Zahlen. Um die obigen drei Eigenschaften ausgeben zu lassen, schreiben wir folgenden Code und erhalten als Ergebnis:

```
print(rand_tensor.shape)  
print(rand_tensor.dtype)  
print(rand_tensor.device)
```

```
torch.Size([2, 3])  
torch.float32  
cpu
```

Wir sehen also, dass die Form des Tensors (2,3,) ist, die enthaltenen Daten Float-Zahlen sind und dass der Tensor auf der CPU gespeichert wird.

Nachdem wir nun die Grundlagen der Tensoren kennen, können wir uns anschauen, welche Operationen wir auf ihnen ausführen können. Und die erste wird sein, einen Tensor von der CPU auf die GPU zu bewegen. Hierfür müssen wir erst mit einer if-Bedingung überprüfen, ob die GPU überhaupt

```
if torch.cuda.is_available():  
    tensor = tensor.to("cuda")
```



verfügbar ist. Dann schieben wir mit der `to()`-Funktion den Tensor von der CPU auf die GPU:

Möchten wir beispielsweise Operationen nur auf bestimmten Elementen des Tensors anwenden, machen wir das ähnlich zur Indexierung bei numpy-Arrays. Erzeugen wir hierfür mal einen (4,4)-Tensor mit zufälligen Zahlen:

```
shape=(4,4)
tensor = torch.rand(shape)
print(tensor)
```

```
tensor([[0.2543, 0.3113, 0.4210, 0.0386],
       [0.5620, 0.7463, 0.0855, 0.1143],
       [0.0321, 0.3054, 0.7844, 0.5063],
       [0.2798, 0.7729, 0.7739, 0.1091]])
```

Die einzelnen Elemente lassen sich wie bei Arrays über die eckigen Klammern aufrufen. Bei unserem Tensor mit der Form (4,4,) sind die Parameter des Tensors die Zeilen und Spalten: `tensor[zeile, spalte]`

Die erste Zeile dieses Tensors können wir uns mit dem folgenden Code ausgeben lassen:

```
#Erste Zeile
print(tensor[0])
```

```
tensor([0.2543, 0.3113, 0.4210, 0.0386])
```

Die erste Spalte dieses Tensors erhalten wir folgendermaßen:

```
#Erste Spalte
print(tensor[:,0])
```

```
tensor([0.2543, 0.5620, 0.0321, 0.2798])
```

## Aktivierungsfunktionen

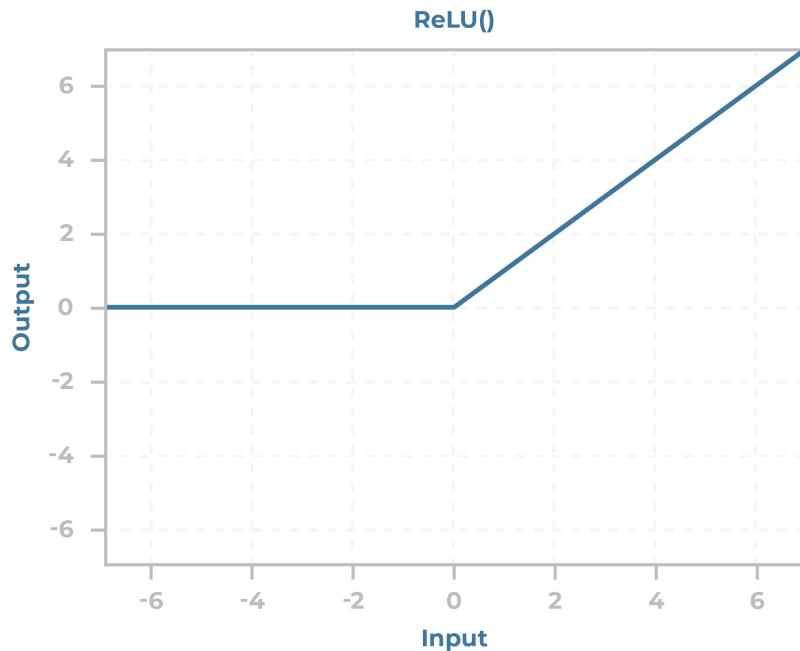
Wie wir bereits zu Beginn dieses Kapitels gesehen haben, ist die Verwendung von Aktivierungsfunktionen unerlässlich für den Erfolg von Neural Networks, weil sie für die nötige Unschärfe sorgen, die sich auch im biologischen Vorbild finden lässt. Bisher lernten wir die Sigmoid-Funktion kennen, haben aber nun bei PyTorch eine ganz andere Funktion verwendet: die sogenannte ReLU-Aktivierungsfunktion, die sehr gerne bei Neural Network Projekten eingesetzt wird. Wir wollen uns also in diesem Abschnitt



verschiedene Alternativen zur Sigmoid-Funktion anschauen und welchen Einfluss sie auf die Performance eines Neural Networks haben. Fangen wir also gleich mit der ReLU-Funktion an.

ReLU ist die Abkürzung für Rectified Linear Unit und ist folgendermaßen definiert:

$$f(x) = \begin{cases} 0, & \text{if } x < 0 \\ x, & \text{if } x \geq 0 \end{cases}$$



Wenn also der Wert des Inputs negativ ist, wird die ReLU-Funktion 0 als Wert liefern, nur wenn der Wert des Inputs 0 oder positiv ist, wird der entsprechende Wert zurückgegeben.

Der Vorteil gegenüber der Sigmoid-Funktion liegt in der Rechenkapazität. Dadurch, dass für negative Werte der Output einfach auf 0 gesetzt wird, erspart sich der Algorithmus einiges an Rechenaufwand, was das Training wiederum beschleunigt. Ein weiterer Grund ist das sogenannte Vanishing-



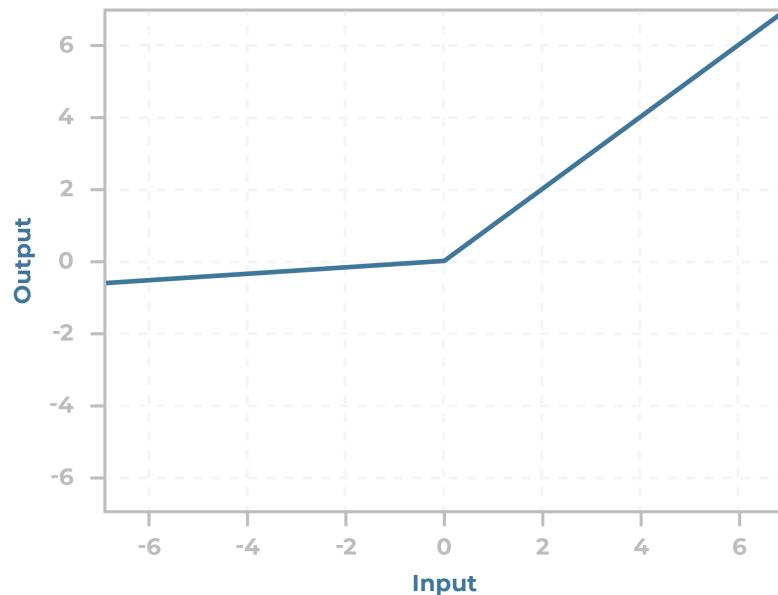
Gradient-Problem. Bei Funktionen wie der Sigmoid-Funktion, welche asymptotisch im Unendlichen sind, konvergiert der Gradient bei einer zunehmenden Anzahl an Schichten gegen 0, was den Output beim Testen negativ beeinflussen kann (da ein verschwindender Gradient bedeutet, dass kein weiteres Training mehr möglich ist). Daher wird ReLU gerne im Deep Learning eingesetzt, wo Neural Networks mit zahlreichen versteckten Schichten trainiert werden.

Die ReLU-Funktion mag zwar das Vanishing-Gradient-Problem lösen, allerdings nur für positiven Input. Da die Funktion für negativen Input immer 0 ist, wird auch der Gradient hier verschwinden. Daher gibt es eine Erweiterung zu der ReLU-Funktion, welche als Leaky ReLU bekannt ist. Hier wird der negative Input mit einem Vorfaktor multipliziert, um so eine leichte Steigung ebenfalls im negativen Bereich zu haben. Definiert ist die Leaky-ReLU-Funktion folgendermaßen:

$$\text{LReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$



LeakyReLU(negative\_slope=0.1)



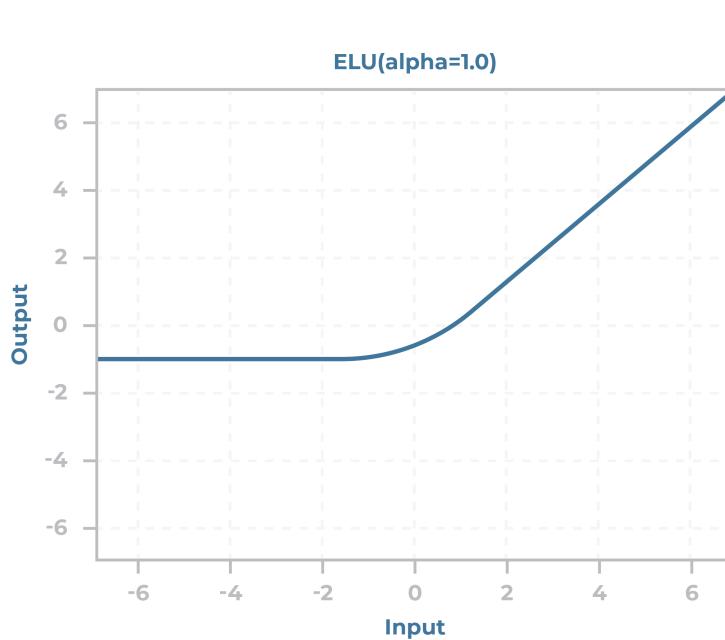
Einer der Nachteile der LReLU-Funktion liegt in der Stelle  $x=0$ . Dort ist die Funktion nicht differenzierbar, was zu einem Hin- und Herspringen des Gradienten führen kann. Ein weiterer Nachteil ist die Wahl des Vorfaktors für den negativen Bereich, da dieser das Training maßgeblich beeinträchtigen kann. Um dieses Problem zu lösen, kann man die sogenannte PReLU-Funktion anwenden, die für Parametrized Rectified Linear Unit steht. Diese ist analog zur LReLU-Funktion definiert, allerdings wird der Vorfaktor während des Trainings kontinuierlich angepasst und optimiert.

Um das Problem mit der vollständigen Differenzierbarkeit zu lösen, wendet man die ELU-Funktion an, was für Exponential Linear Unit steht. Sie ist folgendermaßen definiert:

$$\text{ELU}(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha * (\exp(x) - 1), & \text{if } x \leq 0 \end{cases}$$



Für positiven Input liefert die ELU-Funktion also einen linearen Output, für negativen Input wird ein Ausdruck mit einer Exponentialfunktion angewandt, welcher mit einem Vorfaktor alpha multipliziert wird. In der Praxis wird dieser in der Regel bei ca. 1.0 gewählt. Graphisch sieht die ELU-Funktion folgendermaßen aus (mit einem Vorfaktor von 1):



Wie wir bereits erwähnt haben, löst die ELU-Funktion das Problem der Differenzierbarkeit bei  $x=0$ . Allerdings führt die komplexere Form der Aktivierungsfunktion dazu, dass die Laufzeit des Algorithmus und somit des Trainings erhöht wird. Gleichzeitig wird der Vorfaktor nicht kontinuierlich während des Trainings angepasst, wodurch ebenfalls höhere Abweichungen beim Testen entstehen können.

Pauschal lässt sich also nicht wirklich sagen, welche Aktivierungsfunktion die richtige ist. Das hängt immer sehr stark vom jeweiligen Projekt ab und man muss deshalb mit verschiedenen Parametern experimentieren. Und genau das machen wir jetzt.



## 4. Data-Science-Methoden

### 4.1. Erzeugung von synthetischen Daten

Wir wissen nun, wie wir uns Daten aus dem Internet besorgen können und welche Methoden es gibt, um diese Datensätze zu analysieren. Ein weiterer wichtiger Bestandteil der Arbeit eines Data Scientist ist allerdings nicht nur die Arbeit mit realen Daten, sondern auch mit sogenannten synthetischen Daten. Bei dieser Form von Daten handelt es sich um vom Computer produzierte Datensätze, die ähnlich wie reale Datensätze aufgebaut sind. Prinzipiell gibt es drei Möglichkeiten zur Erzeugung synthetischer Daten:

- Durch statistische Methoden, die auf realen Daten basieren
- Durch semantische Methoden zur Erzeugung von Texten
- Durch Generative Adversial Networks (GANs): Hierbei handelt es sich um ein Machine- Learning-Modell, welches Daten generiert. Dabei arbeiten zwei konkurrierende Neural Networks miteinander. Eines erzeugt echt wirkende Daten, das andere hat die Aufgabe, die Daten als echt oder künstlich zu klassifizieren.

Die Verwendung von synthetischen Daten spielt vor allem eine Rolle beim Schutz der privaten Nutzerdaten.

Aber nicht nur der Schutz persönlicher Daten bringt Unternehmen zukünftig mehr in Bedrängnis, ihre Big-Data-Strategien zu überdenken und anzupassen. Das Sammeln von großen Datenmengen und das anschließende Bereinigen ist sehr zeit- und kostenintensiv. Vor allem, wenn man noch am Anfang von großen Datenanalysen steht und noch nicht weiß, welche Analyse-Methode die beste sein wird. Daher bietet es sich an, künstliche Datensätze zu erstellen, um seine Machine-Learning-Algorithmen zu trainieren oder andere Analyse-Methoden zu testen. Bereits jetzt sind über die Hälfte der Daten, welche zum Training von Machine-Learning- Algorithmen verwendet werden, synthetische Daten und der



Trend geht weiter nach oben. Besonders in Bereichen, in denen es schwierig ist, reale Daten zu bekommen (wie beispielsweise für Bankbetrug, Malware-Attacken oder selbstfahrende Autos) werden synthetische Daten immer mehr zur Effizienzsteigerung der Algorithmen beitragen.

Den ersten wichtigen Grund für die Verwendung von synthetischen Daten haben wir oben bereits kurz erwähnt: Datenschutz. Anstatt reale und personenbezogene Daten zu verwenden, werden Namen, E-Mail-Adressen und / oder reale Adressen einfach ersetzt. Durch die Verwendung von synthetischen Daten – manchmal auch in Kombination mit realen Daten – werden Cyber-Attacken schwieriger. Die Tatsache, dass die Erzeugung von synthetischen Daten wesentlich kosteneffizienter ist, macht die Analysen sehr viel flexibler. Dadurch, dass neue Daten schnell generiert werden können, kann man sich schneller an neue Situationen anpassen.

Die Verwendung synthetischer Daten hat aber auch einen erheblichen Einfluss auf die Performance von Machine-Learning-Modellen. Nehmen wir beispielsweise die Bilderkennung: Hier werden synthetische Daten durch das Drehen, Verzerren oder die Verschiebung von Trainingsbildern erzeugt. Dadurch entsteht ein abwechslungsreicher Datensatz, der zu einer besseren Modell-Performance führt. Das geschieht unter anderem durch die Vermeidung des Overfitting-Effekts, da der Einsatz von synthetischen Daten die Varianz in den Trainingsdaten erhöht.

Die Vorteile, die durch die Verwendung synthetischer Daten für Unternehmen entstehen, werden mit der Zeit immer deutlicher. Schon heute können Firmen dadurch sensible Daten mit Dritten teilen, ohne damit Datenschutzrichtlinien zu verletzen. Die synthetischen Daten werden einfach basierend auf den realen Daten erzeugt.

Schauen wir uns also mal genauer an, wie wir eigentlich synthetische Daten mit einem Python- Skript generieren können.



### 4.1.1. Python Faker

Mit dem Wissen, worum es sich bei synthetischen Daten handelt und wie sie im Alltag eines Data Scientist eingesetzt werden, können wir uns nun einer Implementierung in Python widmen. Dafür werden wir die sogenannte Faker-Bibliothek nutzen – ein Open-Source-Python-Modul zur Erzeugung von synthetischen Datensätzen. Als erstes müssen wir dafür das Modul mit dem pip- Befehl in der Kommandozeile unseres Editors installieren:

```
pip install faker
```

Um diese Bibliothek in unserem Skript nutzen zu können, müssen wir aus der Faker-Bibliothek das Faker-Modul importieren und mit diesem ein Faker()-Objekt erzeugen (welches wir in der Variable fake speichern werden):

```
from faker import Faker  
fake = Faker()
```

Das Faker-Modul enthält einige grundlegende Funktionen, mit denen wir schnell einen Datensatz erzeugen und diesen an unsere Bedürfnisse anpassen können. Schauen wir uns ein paar dieser Funktionen an.

- name()** : Diese Funktion erzeugt einen vollen künstlichen Namen.
- credit\_card\_full()** : Hiermit erzeugt man eine Kreditkartennummer mit Ablaufdatum und CVV.
- email()** : Erzeugung einer E-Mail-Adresse
- url()** : Erzeugung einer URL-Adresse
- phone\_number()** : Erzeugung einer Telefonnummer mit Landes-Vorwahl
- address()** : Erzeugung einer vollständigen Adresse
- license\_plate()** : Erzeugung eines Autokennzeichens
- local\_latlng()** : Erzeugung von Koordinaten (Breitengrad und Längengrad) zusammen mit einem Land oder einer Gegend



**text()** : Erzeugung eines kurzen, synthetischen Textes

**company()** : Erzeugung eines Firmennamens

## 4.2. Datenvisualisierung – Geographische Daten

### 4.2.1. Grundlagen

Da es nicht selten vorkommt, dass man es bei Data-Science-Projekten mit geographischen Daten zu tun hat, wollen wir uns in diesem Abschnitt anschauen, wie wir diese mit Hilfe der matplotlib- Bibliothek visualisieren können. Hierfür werden wir auf eines der zahlreichen Toolkits der matplotlib-Bibliothek zugreifen – das sogenannte Basemap Toolkit –, welche wir im Modul mpl\_toolkits finden. Zunächst einmal müssen wir das basemap-Modul mit einem Befehl in der Kommandozeile installieren:

```
conda install -c anaconda basemap
```

Nun können wir die Bibliotheken importieren, die wir im Laufe dieses Abschnitts benötigen werden:

```
import numpy as np
import matplotlib.pyplot as plt
import mpl_toolkits
from mpl_toolkits.basemap import Basemap
```

Nun, da wir das basemap-Modul installiert und die wichtigen Bibliotheken importiert haben, können wir unsere erste Karte plotten. Hierbei ist zu

```
fig = plt.figure(figsize=(8, 8))
m = Basemap(projection='lcc', resolution='i',
            width=8E6, height=8E6,
            lat_0=45, lon_0=10,)
m.etopo(alpha = 0.5)

# Map (long, lat) to (x, y) for plotting
x, y = m(10, 47.6)
plt.plot(x, y, 'ok', markersize=5)
plt.text(x, y, 'Ort in den Alpen', fontsize=12)
```



beachten, dass es viele verschiedene Möglichkeiten gibt, um die 3-dimensionalen geographischen Daten auf einer 2-dimensionalen Fläche wie einem Bildschirm darzustellen. Diese Möglichkeiten werden als Projektionen bezeichnet: Die 3-dimensionale gekrümmte Oberfläche der Erde wird auf eine 2-dimensionale Fläche projiziert. Als erstes werden wir uns eine Karte in der sogenannten Lambert-Conformal- Projektion anschauen, welche in Basemap mit „lcc“ abgekürzt wird:

Zunächst erzeugen wir mit der Zeile `fig = plt.figure(figsize=(10, 10))` eine Abbildung. Dann erzeugen wir das Basemap-Objekt, welches die folgenden Parameter hat:

- `projection`: Hiermit ist die Art der Projektion gemeint. In diesem Beispiel haben wir die Lambert-Conformal-Projektion gewählt – „lcc“.
- `resolution`: Hiermit ist die Auflösung der Karte gemeint. Zur Auswahl stehen "c" for crude, „l“ für low, „i“ für intermediate, „h“ für high und „f“ für full".
- `width, height`: Hiermit ist die Breite und die Höhe der Karte in Metern gemeint. Wir bilden im obigen Beispiel also ein Quadrat von 8000 Kilometern Seitenlänge ab.
- `lat_0, lon_0`: Das sind die Breitengrad- und Längengrad-Angaben des Punktes, der in der Mitte der Karte sein soll.

Im nächsten Schritt legen wir mit der `etopo()`-Funktion fest, dass wir eine topografische Karte haben wollen. Der `alpha`-Parameter gibt die Transparenz der Karte an (oder auch wie intensiv die Farben zu sehen sind).

Diese ersten Zeilen reichen aus, um eine Karte zu plotten. Wir sind allerdings nicht nur daran interessiert, eine Karte zu plotten, sondern wir wollen in den Karten auch Daten abbilden. Das kommt in den nächsten Zeilen.



Dort geben wir in der Zeile `x, y = m(10, 47.6)` die Koordinaten des Punktes an, den wir auf der Karte markieren wollen. Mit der `plot()`-Funktion plotten wir diesen Punkt in der Karte und geben ihm mit der `text()`-Funktion noch einen Namen. Das Ergebnis sollte dann so aussehen:



## 4.2.2. Projektionen

Wir haben bereits gesehen, dass wir bei der Initialisierung des `Basemap()`-Objekts als Parameter angeben können, welche Art der Projektion wir für unsere Karte wünschen. Über die letzten Jahrhunderte wurden viele

```
# Lambert-
# Projektion
fig = plt.figure(figsize=(10, 10))
m = Basemap(projection='lcc', resolution='i',
            width=2E6, height=2E6,
            lat_0=65, lon_0=18,
            etopo(alpha = 0.5)
```

unterschiedliche Möglichkeiten zur Projektion der 3-dimensionalen Erdoberfläche auf 2-dimensionale Karten entwickelt und mit der `Basemap`-Bibliothek können wir einige Dutzend dieser Projektionen auch in unseren Projekten anwenden. Hier wollen wir uns die am häufigsten benutzten Projektionen anschauen und wie wir sie in Python implementieren können. Oben haben wir bereits kurz die sogenannte Lambert-Projektion kennengelernt. Diese gehört zu den sogenannten konischen Projektionen. Das bedeutet, dass die Karte auf einen Kegel projiziert wird, welcher dann entrollt wird. Konische Projektionen bieten sich an, um kleine Teilbereiche des Globus abzubilden. Der Nachteil dieser Projektion besteht darin, dass die Orte am Rand der Karte sehr verzerrt werden können. Dennoch liefert sie eine sehr gute Visualisierung für den Fokuspunkt. Die Python-Implementierung haben wir auch schon kennengelernt:



Weitere konische Projektionen sind die „Equidistant-Conic-Projektion“ (`projection='eqdc'`) und die „Albers-Equal-Area-Projektion“ (`projection='aea'`).

Die einfachste Form der Projektion ist die sogenannte zylindrische Projektion. Hierbei projiziert man die 3-dimensionale Kugeloberfläche auf die Oberfläche eines Zylinders, welchen man anschließend wieder entrollt. Umsetzen können wir das mit dem folgenden Code:

```
fig = plt.figure(figsize=(8, 6), edgecolor='w')
m = Basemap(projection='cyl', resolution=None,
            llcrnrlat=-90, urcrnrlat=90,
            llcrnrlon=-180, urcrnrlon=180, )

m.shadedrelief()
```

Hier kommen jetzt bei der Initialisierung des `Basemap()`-Objekts einige neue Parameter vor, die wir uns anschauen wollen:

- `llcrnrlat=-90` – lower left corner latitude: Der Breitengrad der unteren linken Ecke
- `llcrnrlon=-180` – lower left corner longitude: Der Längengrad der unteren linken Ecke
- `urcrnrlat=90` – upper right corner latitude: Der Breitengrad der oberen rechten Ecke
- `urcrnrlon=180` – upper right corner longitude: Der Längengrad der oberen rechten Ecke

Mit diesen Parametern können wir eingrenzen, welchen Bereich der Erdkugel wir abbilden wollen. Im diesem Beispiel wählen wir die gesamte Erde und das Ergebnis sieht dann folgendermaßen aus:



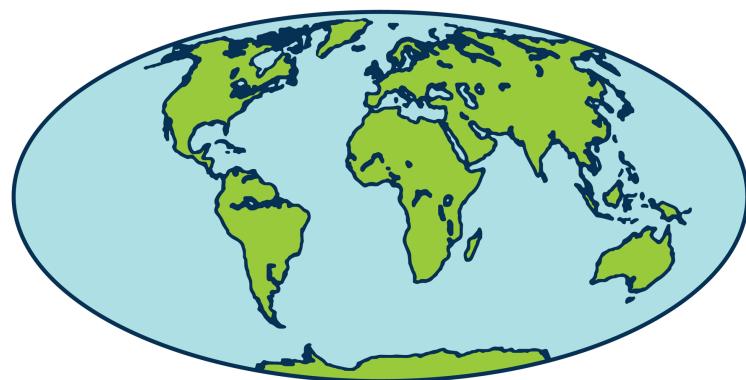
Eine Erweiterung der zylindrischen Projektion ist die pseudo-zylindrische Projektion. Diese zeichnet sich dadurch aus, dass sämtliche Längengrade in den Polen – also einem Punkt oben und einem unten – enden. Ein Beispiel für eine pseudo-zylindrische Projektion ist die Mollweide- Projektion (abgekürzt mit `projection='moll'`). Wollen wir eine Mollweide-Projektion in Python implementieren, benötigen wir folgenden Code:

```
fig = plt.figure(figsize=(8, 6), edgecolor='w')
m = Basemap(projection='moll', lon_0=0, resolution='c')

m.drawcoastlines()
m.fillcontinents(color='#2ECC71', lake_color='#AED6F1')
m.drawmapboundary(fill_color='#AED6F1')

plt.show()
```

Das Ergebnis sieht dann folgendermaßen aus:





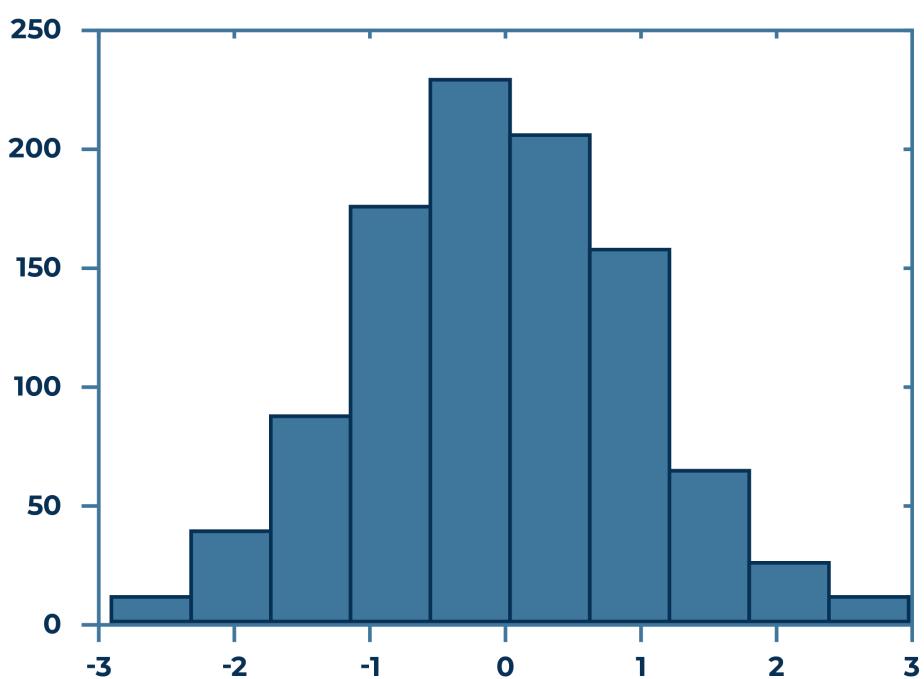
Hierbei haben wir nach der Initialisierung des Basemap()-Objekts die Funktion `m.drawcoastlines()` aufgerufen, um die Küstenlinien der Kontinente einzulegen. Anschließend haben wir die Kontinente und den Kartenrand mit Farben gefüllt.

### 4.3. Matplotlib Stylesheets

Als wir im ersten Kapitel die matplotlib-Bibliothek kennengelernten, haben wir gelernt, wie wir verschiedene Arten von Plots recht simpel in Python erstellen können. Allerdings spielt bei der Visualisierung der Daten nicht nur der Plot selbst eine Rolle, sondern auch wie visuell ansprechend dieser ist. Normalerweise werden beim Plotten immer die sogenannten default-Einstellungen übernommen, die in der Regel nicht allzu ästhetisch sind. Schauen wir uns dafür einfach dieses einfache Histogramm einer Zufallsverteilung an:

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

x = np.random.randn(1000)
plt.hist(x)
```



Mit der `randn()`-Funktion haben wir 1000 zufällige aber normalverteilte Zahlen erzeugt und in einem Histogramm dargestellt. Das Histogramm sieht jetzt nicht besonders schön aus, deshalb wollen wir es etwas aufwerten.

#### 4.4. 3-dimensionale Plots

Ursprünglich wurde die `matplotlib`-Bibliothek nur zur Darstellung von 2-dimensionalen Plots entworfen, allerdings hat man es als Data Scientist häufig mit Situationen zu tun, in denen ein 3- dimensionaler Plot viel besser zur Visualisierung der Daten beitragen würde. Solche 3D-Plots können wir durch das `mplot3d-toolkit` realisieren, welches wir aus dem `mpl_toolkits`-Modul importieren:

```
from mpl_toolkits import mplot3d
```



Ein 3-dimensionales Koordinatensystem können wir dann durch den Parameter `projection='3d'` implementieren:

```
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
fig = plt.figure()
ax = plt.axes(projection='3d')
```

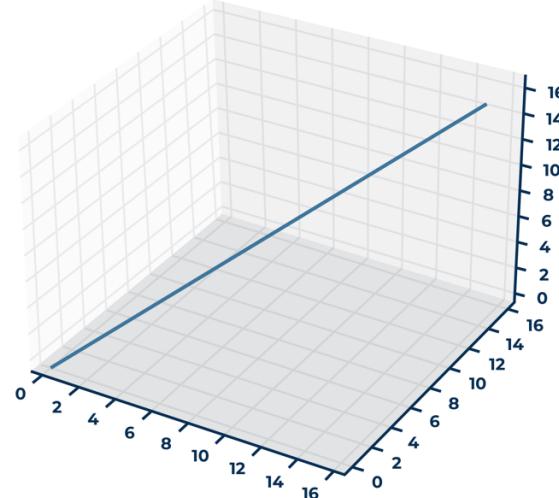


Möchten wir beispielsweise einfach nur eine 3-dimensionale Ursprungsgerade plotten, bekommen wir das mit den folgenden Zeilen hin:

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits import mplot3d

fig = plt.figure()
ax = plt.axes(projection='3d')

# 3D Linie
zline = np.linspace(0, 15, 1000)
xline = zline
yline = zline
ax.plot3D(xline, yline, zline, 'blue')
```



Anstatt der klassischen `plot()`-Funktion verwenden wir hier die `plot3D()`-Funktion. Analog können wir auch einen Scatter-Plot von Punkten erstellen:

```
# 3D Scatter Plot
zdata = 15 * np.random.random(100)
xdata = zdata
ydata = zdata
ax.scatter3D(xdata, ydata, zdata, c=zdata, cmap='Greens');
```

