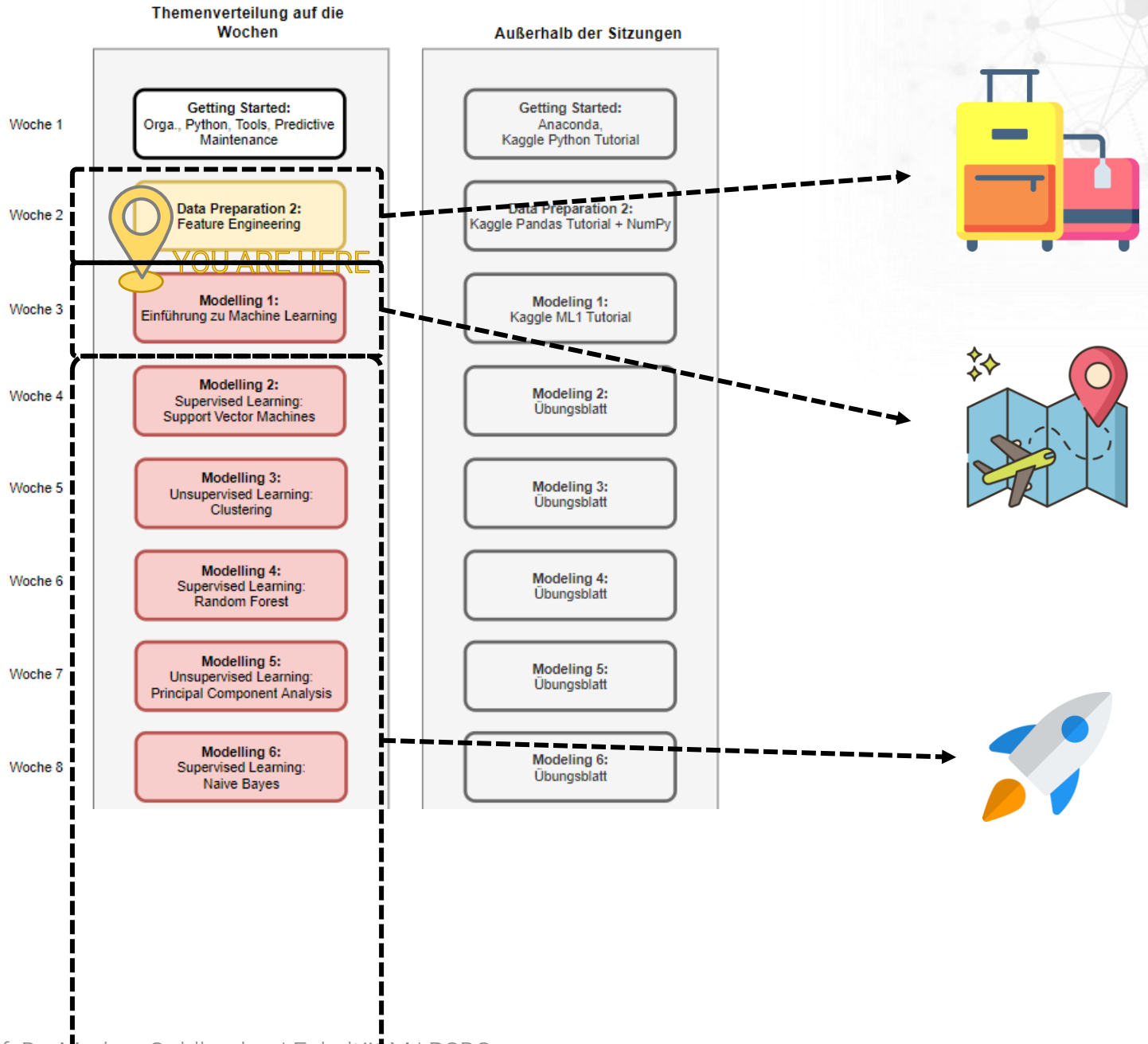
Modeling

Einführung in Machine Learning

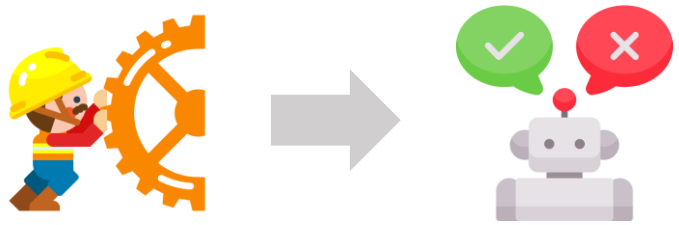
Agenda

1. Was ist Machine Learning?
2. Kategorien des Machine Learning und qualitative Beispiele
3. Machine Learning in Python: `scikit-learn`
4. Hyperparameter und Model Validation
5. Bias-Variance Trade-off
6. Cross-Validation und Generalisierung
7. Kostenfunktionen
8. Machine Learnining Pipelines (evtl. in der letzten Sitzung bzw. extra Foliensatz erstellen)

Wo sind wir?


Themenverteilung auf die Wochen
Außerhalb der Sitzungen
Woche 1
Getting Started: Orga., Python, Tools, Predictive Maintenance
Getting Started: Anaconda, Kaggle Python Tutorial
Woche 2
Data Preparation 2: Feature Engineering
YOU ARE HERE
Data Preparation 2: Kaggle Pandas Tutorial + NumPy
Woche 3
Modelling 1: Einführung zu Machine Learning
Modeling 1: Kaggle ML1 Tutorial
Woche 4
Modelling 2: Supervised Learning: Support Vector Machines
Modeling 2: Übungsblatt
Woche 5
Modelling 3: Unsupervised Learning: Clustering
Modeling 3: Übungsblatt
Woche 6
Modelling 4: Supervised Learning: Random Forest
Modeling 4: Übungsblatt
Woche 7
Modelling 5: Unsupervised Learning: Principal Component Analysis
Modeling 5: Übungsblatt
Woche 8
Modelling 6: Supervised Learning: Naive Bayes
Modeling 6: Übungsblatt

Ausgangssituation



Wo befinden wir uns jetzt?

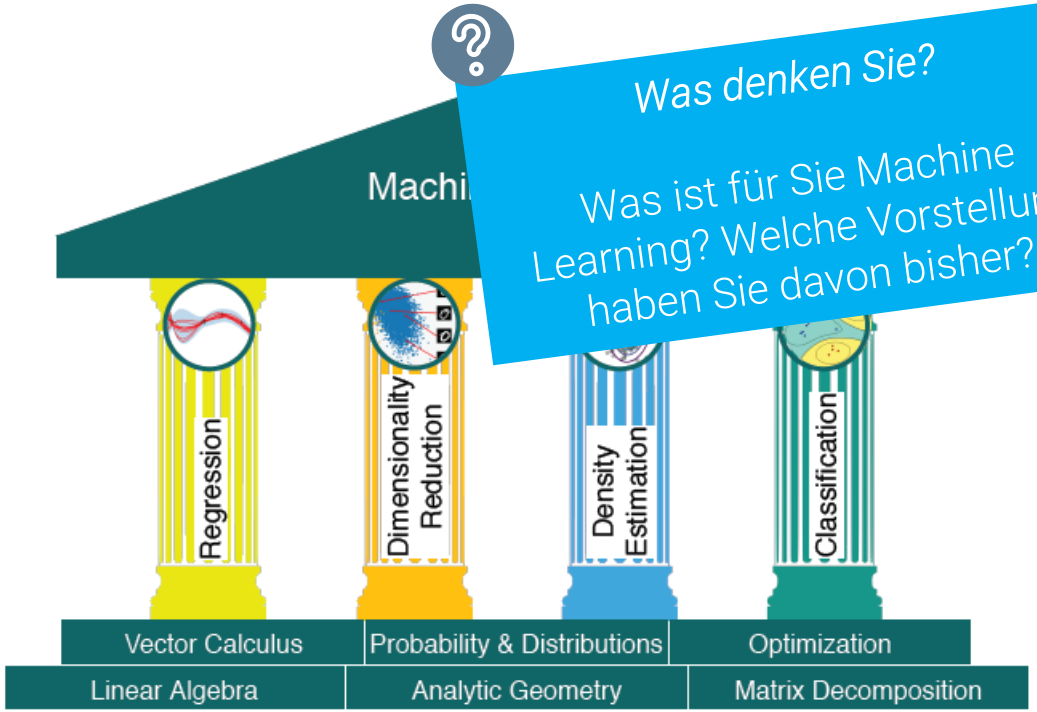
- Wir wissen jetzt was ein Feature ist
- Wir haben eine Vorstellung von einem Feature-Raum
- Wir wissen, wie Daten in einem Feature-Raum repräsentiert werden
- Wir wissen wie man Daten exploriert und **„manuell“** Schlüsse zieht (deskriptiv und inferenzstatistisch)

→ Nun wollen wir, dass Algorithmen für uns Schlüsse ziehen und Urteile fällen!

So what?

Interessanter Aspekt vorab: im Machine Learning bringen wir Algorithmen bei *wissenschaftlich zu Denken bzw. zu arbeiten*

Was ist Machine Learning?



Aus Deisenroth et al. „Mathematics for Machine Learning“

Was denken Sie?

Was ist für Sie Machine Learning? Welche Vorstellung haben Sie davon bisher?

Machine Learning == „Building Models of Data“

- Machine Learning zeichnet sich dadurch aus, dass mathematische Modelle entstehen, um zugrundeliegende **Daten** zu **verstehen**
- Diese Modelle werden an die **Daten angepasst**, damit sie diese so optimal wie möglich repräsentieren
- Diese Anpassung kann stattfinden, da Machine Learning Modelle „**tunable parameters**“ haben
  - → Anpassung dieser auf die Daten
  - → Das Modell **lernt** aus den Daten
- Nach diesem Lernprozess können trainierte Modelle dazu verwendet werden, um **Labels** ungesehener Daten vorherzusagen

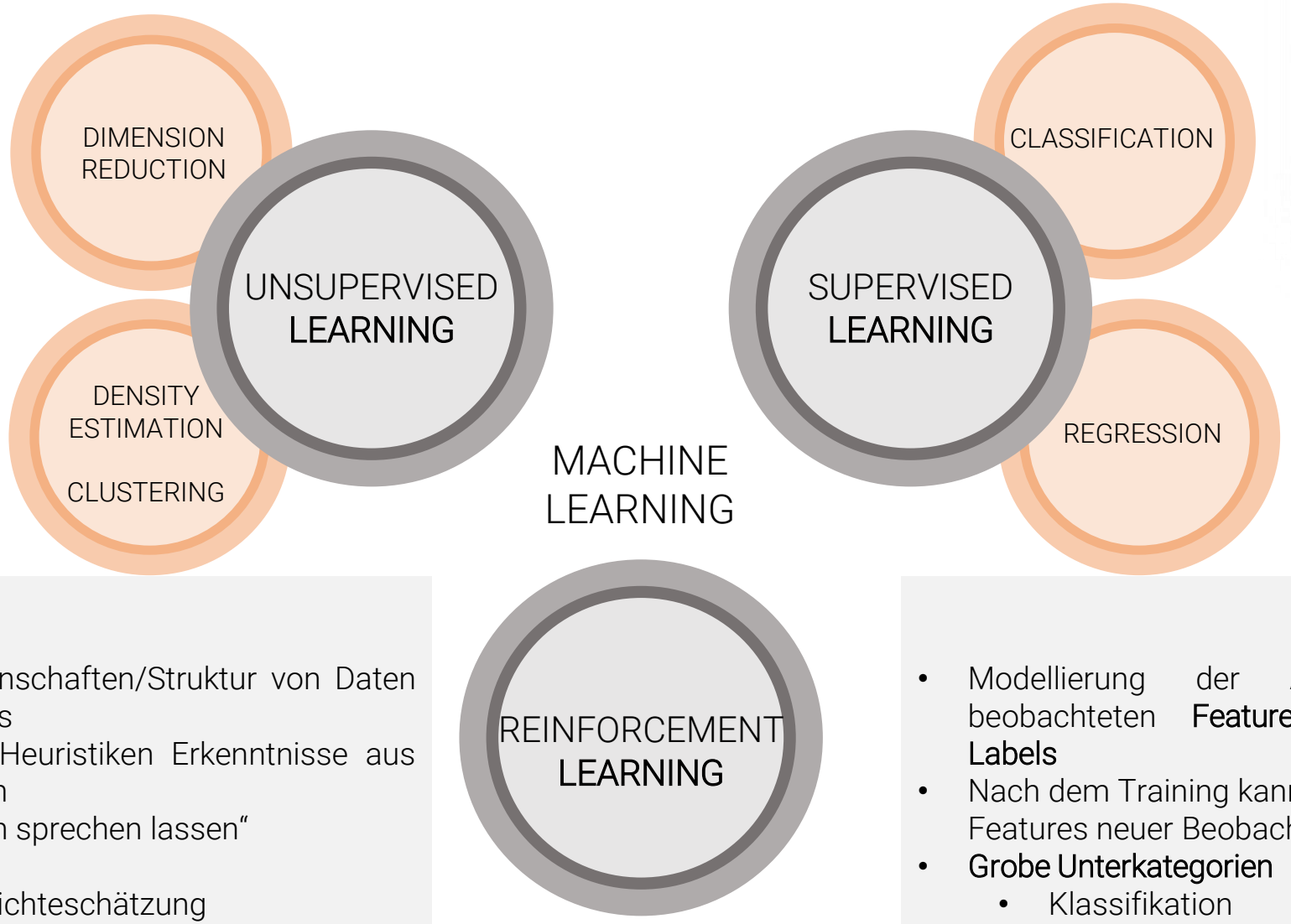
So what?

Wir führen in dieser Vorlesung die wichtigsten Konzepte des Machine Learning anhand einfacher und qualitativer Beispiele ein

Was denken Sie?

Was suchen also Machine Learning Modelle?

Kategorien des Machine Learning: Überblick



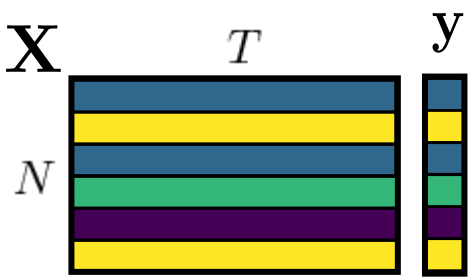
Unsupervised Learning

- Modellierung der Eigenschaften/Struktur von Daten **ohne** zugehörige Labels
- Man versucht durch Heuristiken Erkenntnisse aus den Daten zu gewinnen
- „Den Datensatz für sich sprechen lassen“
- **Grobe Unterkategorien**
  - Clustering und Dichteschätzung
  - Dimensionsreduktion

Supervised Learning

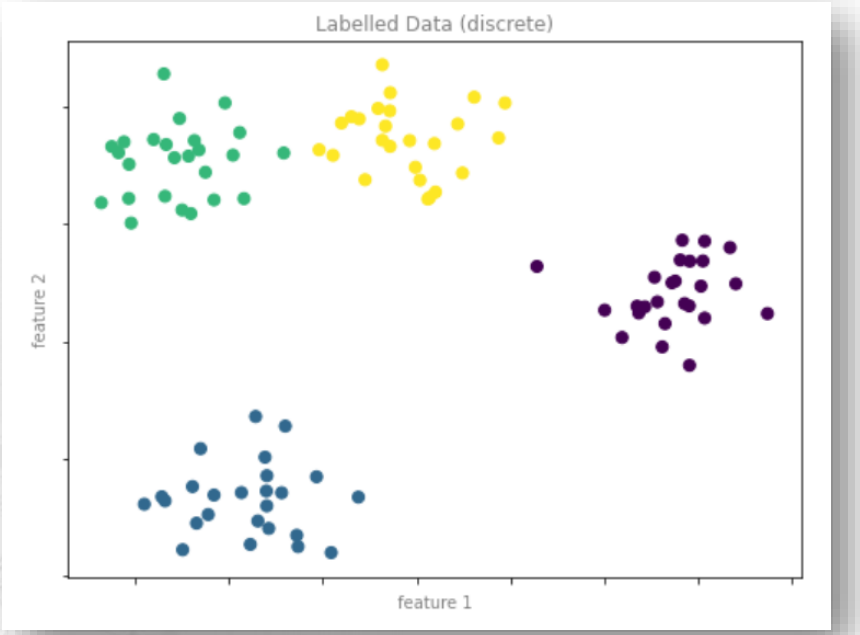
- Modellierung der Abhängigkeiten von beobachteten **Features** und zugehörigen **Labels**
- Nach dem Training kann das Modell Labels zu Features neuer Beobachtungen zuordnen
- **Grobe Unterkategorien**
  - Klassifikation
  - Regression

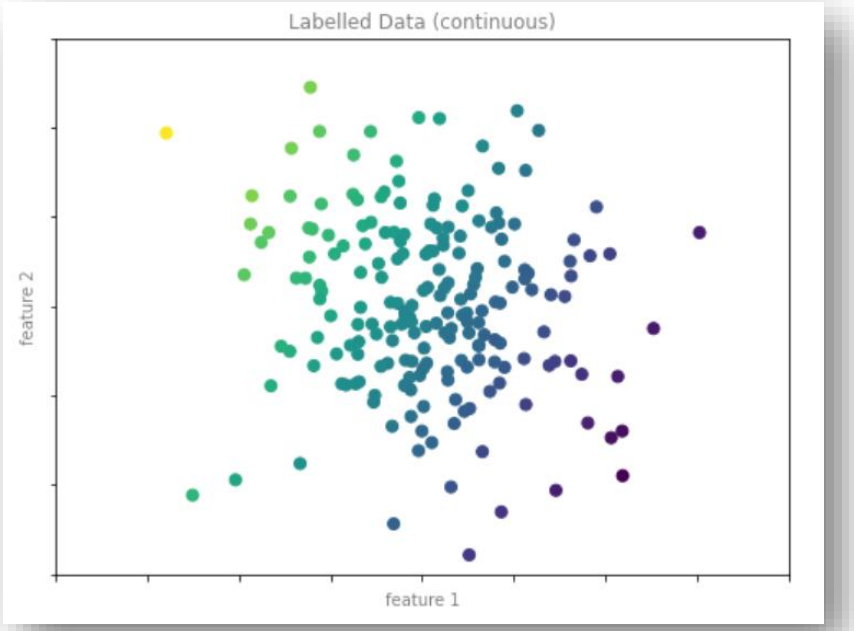
Supervised Learning: Datensicht



Labels bzw. Targets

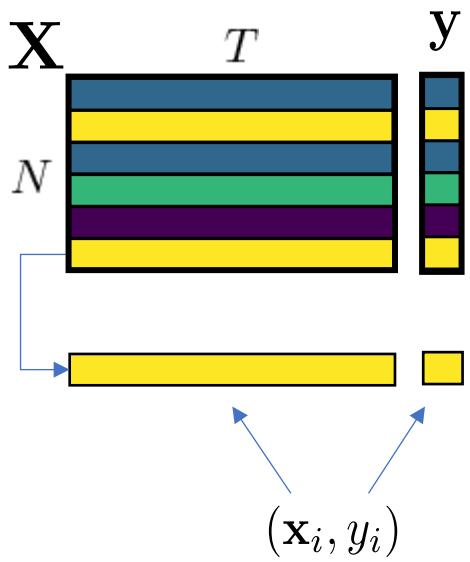
- Wir kennen schon unsere Feature-Matrix: nun assoziieren wir einen neuen Begriff mit dieser – ein **Label** oder **Target**
- Bei Labels handelt es sich um eine Variable, die **abhängige Variable**, die man aus den Features, den **unabhängigen Variablen**, vorhersagen will
- Labels können unterschiedliche Ausprägungen haben
  - **Klassifikation**: Labels haben eine **diskrete** Ausprägung
  - **Regression**: Labels haben eine **kontinuierliche** Ausprägung



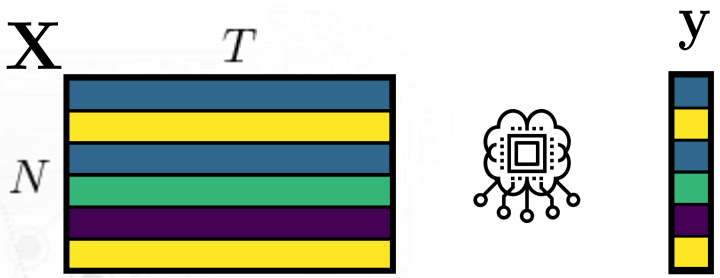


In den Abbildungen:
Farbe == Label

Supervised Learning: mathematische Notation



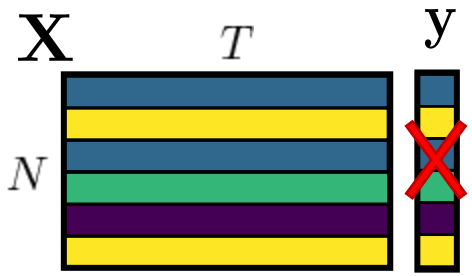
- Unsere Feature-Matrix beschreiben wir – wie gewohnt – durch ein $\mathbf{X}$ mit den Dimensionen *N x T*
- Das zugehörige Label ist ein *N*-dimensionaler Spaltenvektor $\mathbf{y}$
- Im sog. **Trainingsdatensatz** existiert zu jeder Beobachtung (Zeile der Feature-Matrix) ein Eintrag im Label-Vektor
- Wenn wir die Feature-Matrix als eine Menge an gestapelten Zeilenvektoren $\mathbf{x}_i$ betrachten, dann liegt also folgende Assoziation vor $(\mathbf{x}_i, y_i)$

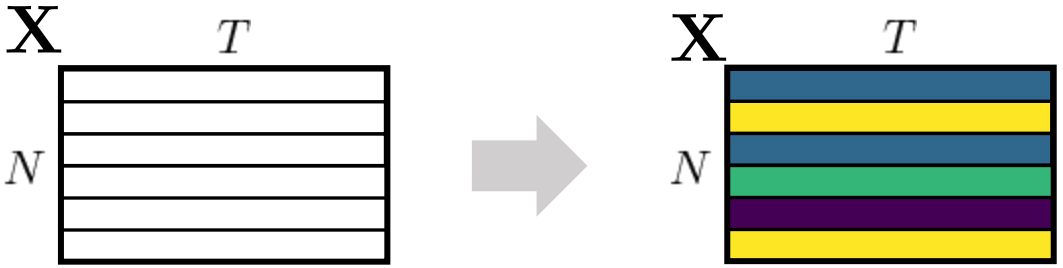


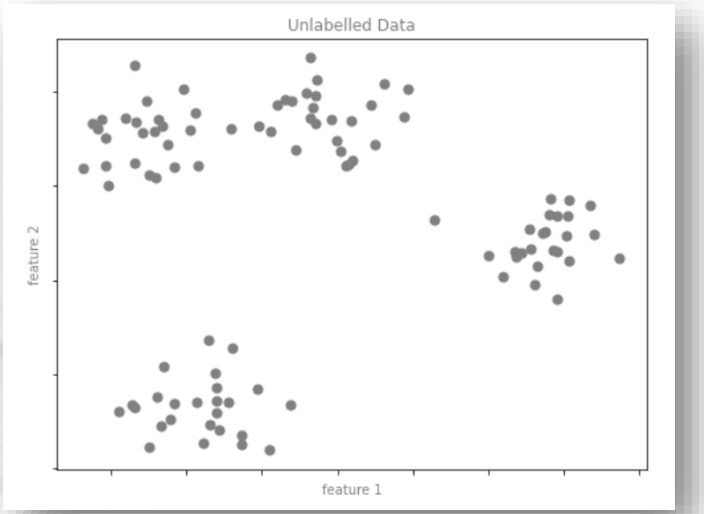
Trainingsdatensatz

- Unter diesem Begriff versteht man die Daten (und Labels), anhand derer das Machine Learning Modell **trainiert** wird
- An diesem wird also die Abhängigkeit der Labels von den Features geschätzt

Unsupervised Learning: Datensicht



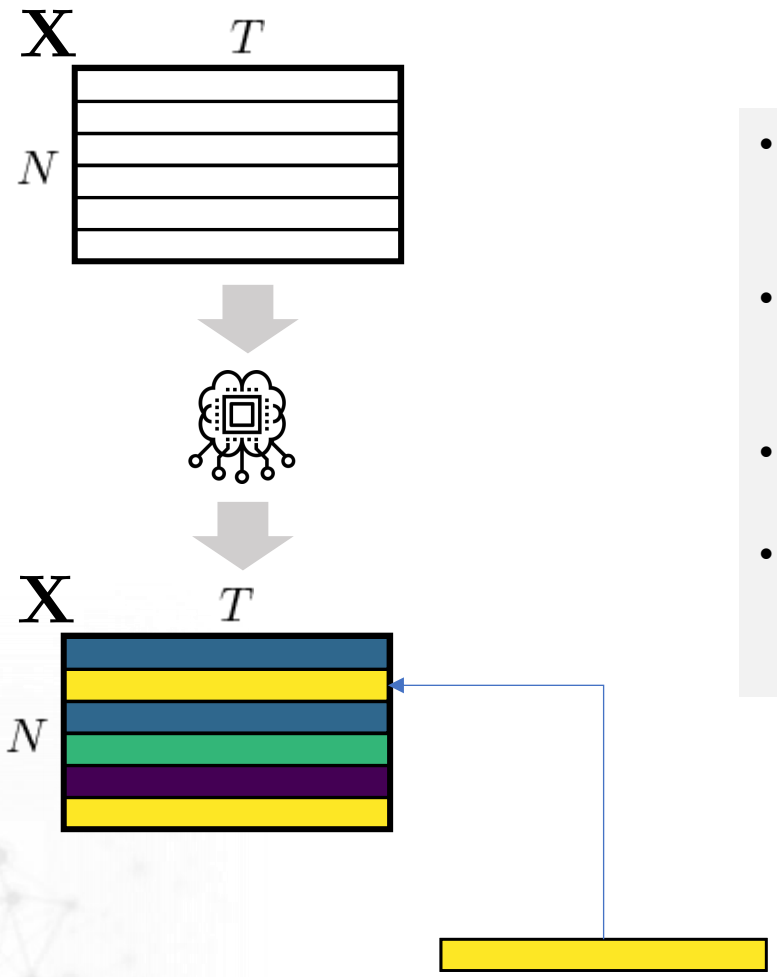




Unsupervised Learning == Keine Labels!

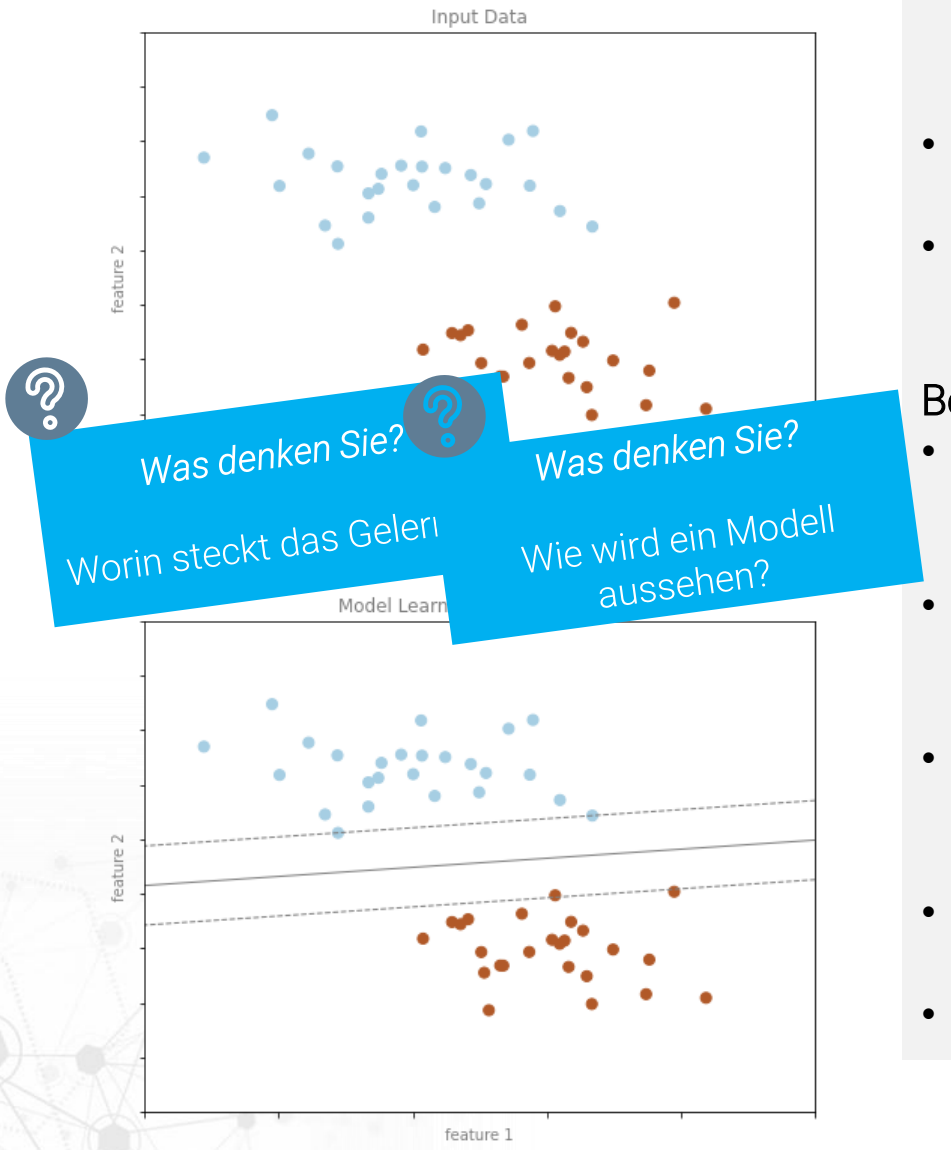
- Beim Unsupervised Learning liegen uns keine Labels vor
- Die, dem Datensatz zugrundeliegende Struktur, ist für uns (anfänglich) nicht ersichtlich bzw. markiert
- Wir haben es also mit einer „nicht eingefärbten“ Feature-Matrix zu tun
- Durch unsere Unsupervised Learning Modelle versuchen wir die Einfärbung zu schätzen

Unsupervised Learning: mathematische Notation



- Wir haben also im Unsupervised Learning Fall (zuerst) nur eine Feature-Matrix $\mathbf{X}$
- Das Machine Learning Modell versucht in diesem Fall Strukturen bzw. Muster in den Daten zu entdecken
- Wir können diese Muster dann auch mit Labels $\mathbf{y}$ versehen
- Nachdem das Modell trainiert wurde, können auch im Unsupervised Learning Fall neue, ungesehene Daten den entdeckten Strukturen bzw. Muster zugeordnet werden

Supervised Learning: Klassifikation



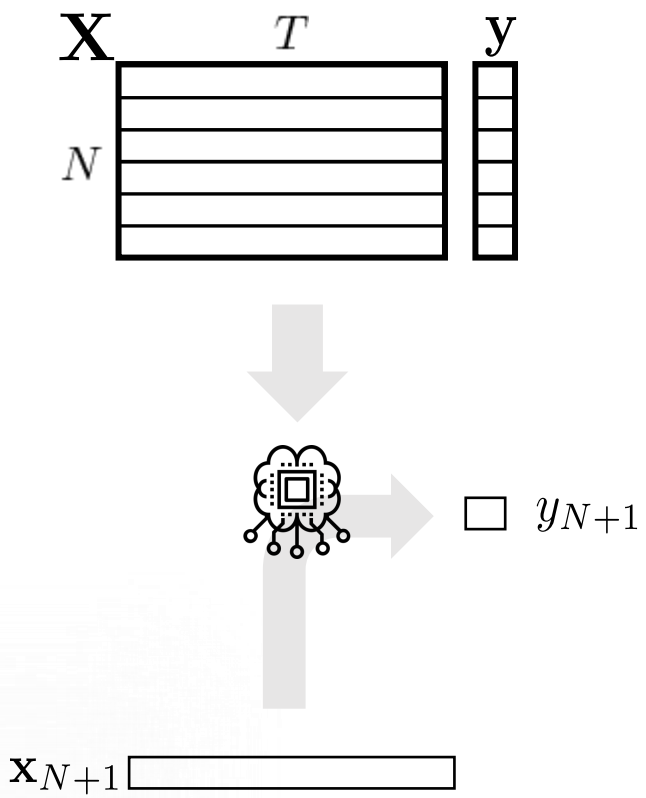
- Unsere Aufgabe bei einer Klassifikation ist: wir haben einen gelabelten Datensatz vorliegen (Trainingsdatensatz)
- Daran trainieren wir unser Machine Learning Modell
- Unser Modell soll dann **ungelabelte** Datenpunkte **klassifizieren** → Labels zuweisen

Beispiel

- In unserem Beispiel liegen zweidimensionale Daten vor, die auf zwei Kategorien aufgeteilt sind (blau und rot)
- Unser Modell soll soz. eine „Trennlinie“ finden, die die beiden Klassen aufteilt
- Neue Daten können dann anhand dieser Trennlinie bewertet werden, ob sie in die blaue oder rote Kategorie fallen
- Die **Modellparameter** wären in diesem Fall die Koeffizienten der Geraden
- Diese Modellparameter werden aus den Daten **gelernt**

So what?
„Learning is finding parameters.“

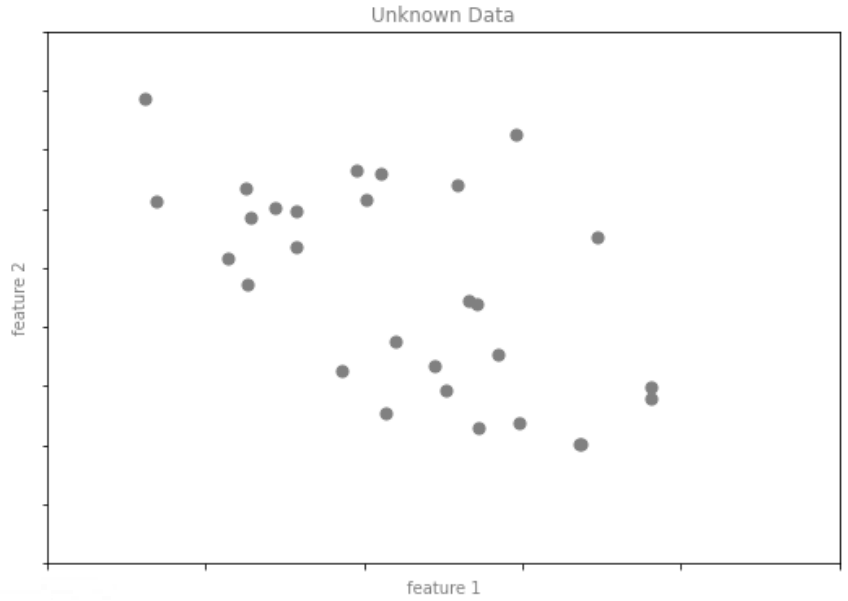
Konzepte des Machine Learning: Scoring bzw. Prediction

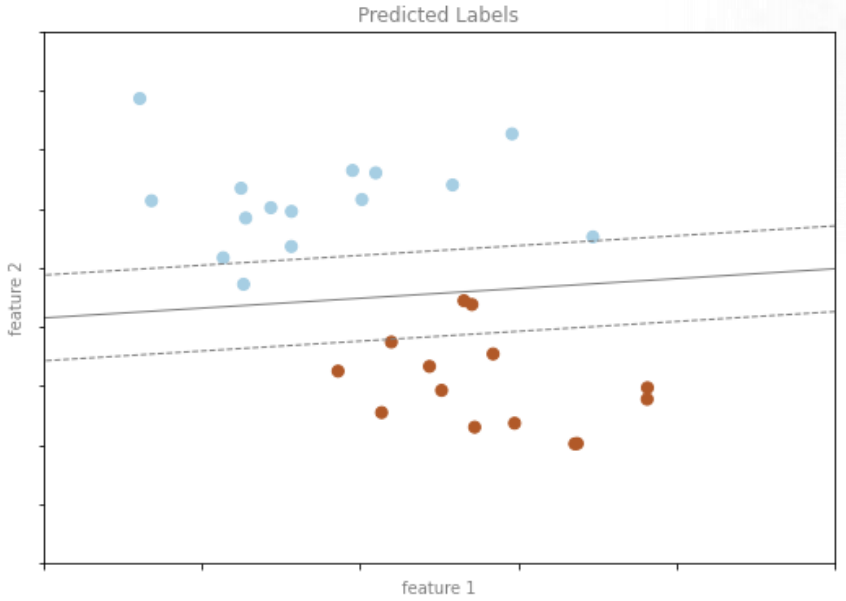


- Wir haben nun schon kennengelernt, dass aus dem Training ein Modell entsteht, das die Beziehung zwischen Features und Labels abbildet
- Ziel beim Machine Learning ist dieses gelernte Wissen auf **neue**, vorher **ungesehene** Daten, anzuwenden
- Man führt dem Modell neue Beobachtungen zu und dieses bewertet diese dann
  → Das Modell weist den neuen Beobachtungen Labels zu

Diese Zuweisung von Labels auf neue Beobachtungen nennt man Scoring bzw. Prediction

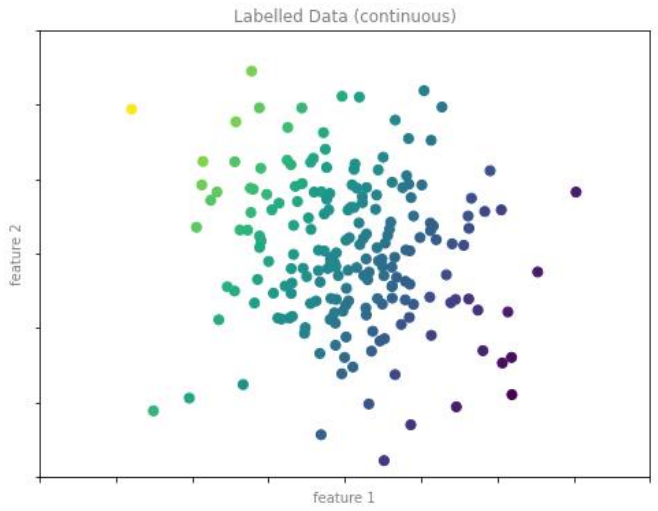
Supervised Learning: Klassifikation

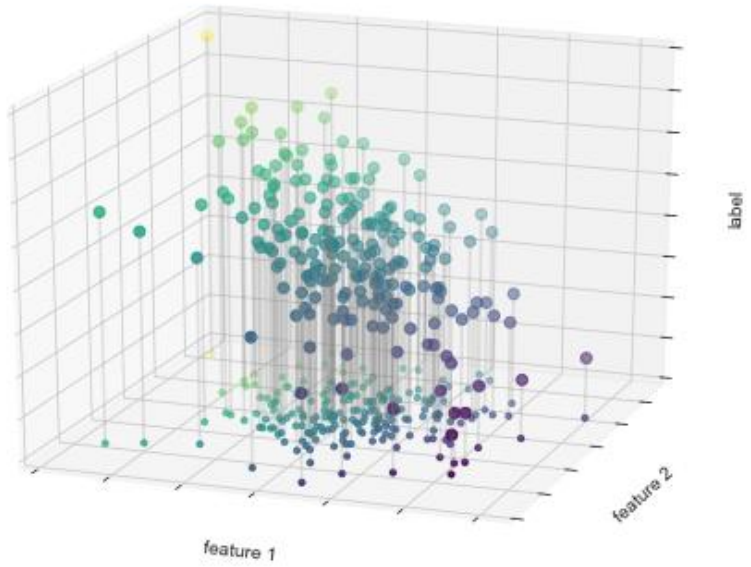




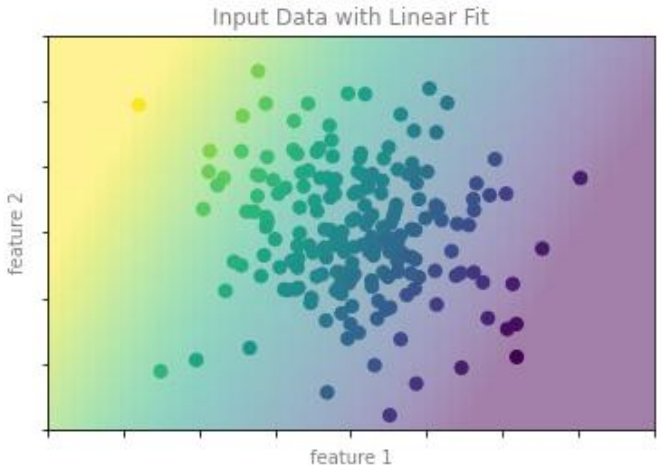
- Scoring bzw. Prediction wird in unserem Beispiel so durchgeführt, dass neue, unbekannte Daten mit dem Modell (unsere gelernte Gerade) **verglichen** werden
- Je nachdem auf welcher Seite der Gerade die Datenpunkte liegen, werden die entsprechenden **diskreten** Labels (rot oder blau) zugeordnet

Supervised Learning: Regression

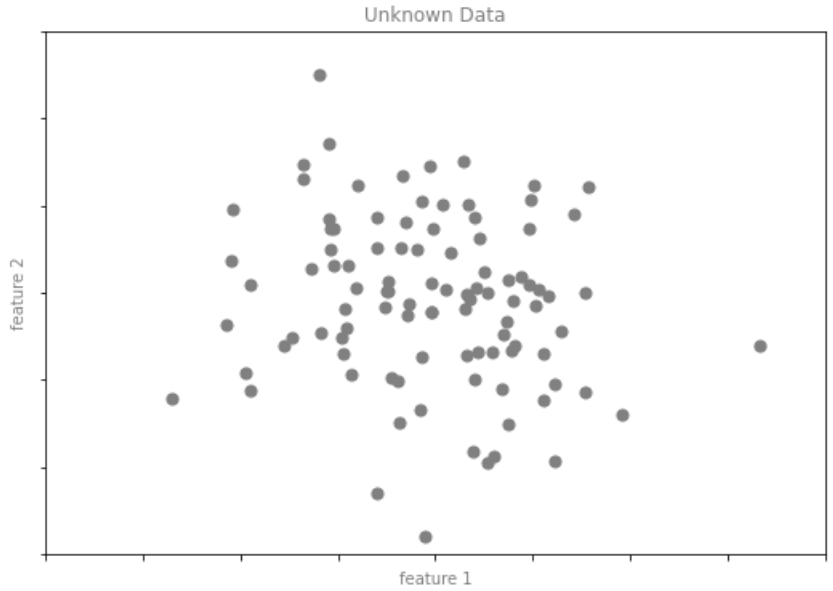


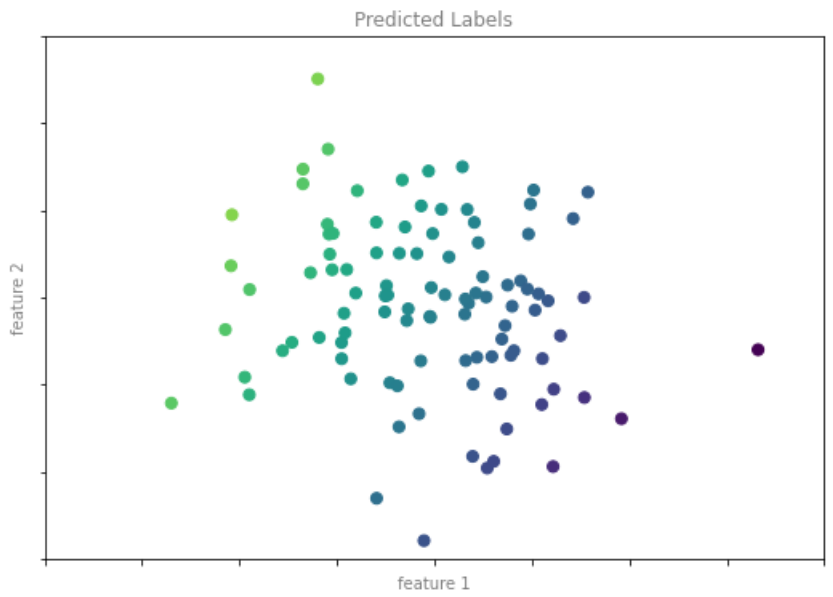


- Im Gegensatz zur Klassifikation liegen uns bei der Regression **kontinuierliche** Labels vor
- In unserem zweidimensionalen Feature-Raum können wir uns das Label dann als eine **dritte Dimension** vorstellen
- Bei einer einfachen Regression würde man dann versuchen eine **Ebene** auf diese Daten zu fitten



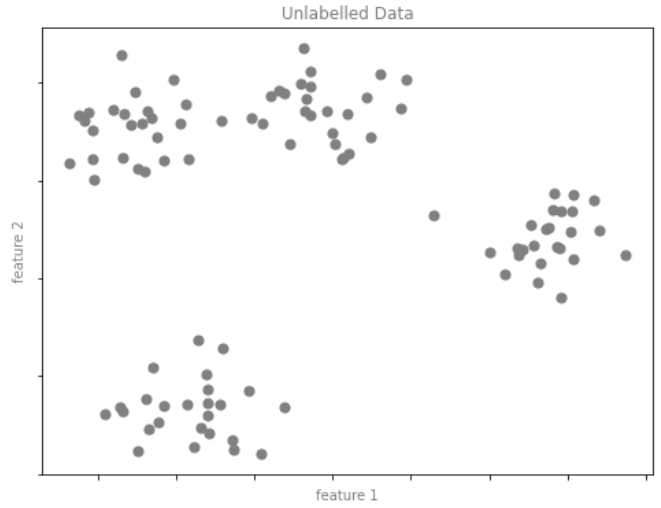
Supervised Learning: Regression: Scoring

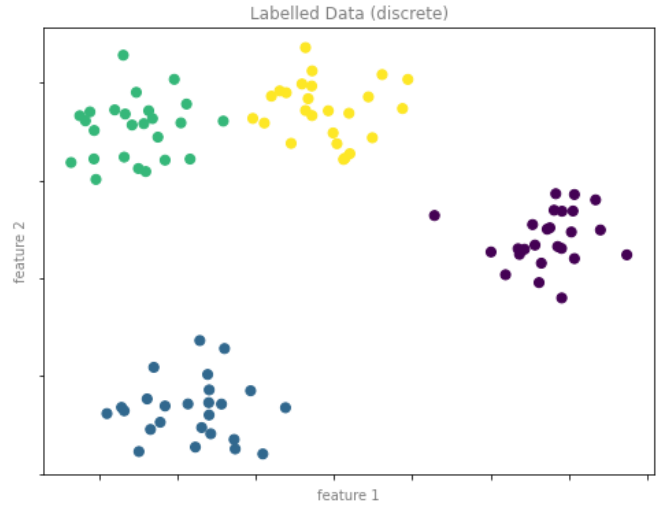




- Beim Scoring im Regressionsfall werden die unbekannten Daten soz. auf die geschätzte Ebene **projiziert**
- Dies liefert uns dann die **neuen**, kontinuierlichen **Labels** für die ungelabelten, neuen Daten

Unsupervised Learning: Clustering





Was denken Sie?

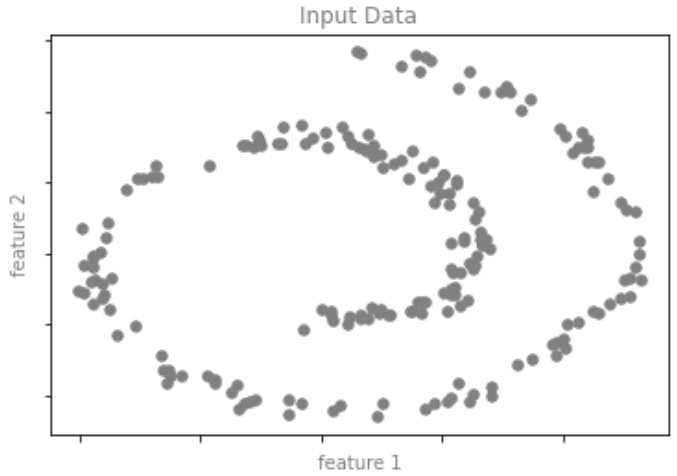
Wie würden Sie abschätzen, ob ein Datenpunkt zu einem bestimmten Cluster gehört? Welches Maß würden Sie verwenden?

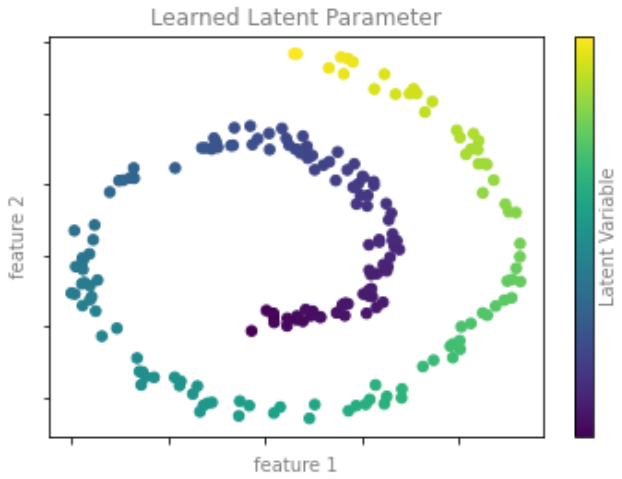
- Ein typisches Unsupervised Learning Problem ist **Clustering**
- Beim Clustering werden die ungelabelten Daten **diskreten Gruppen bzw. Labels** zugeordnet
- Clustering ist eng verwandt mit der **Dichteschätzung** der zugrundeliegenden Daten
- Neue Datenpunkte können dann z.B. durch **Abstandsbetrachtungen** einem bestimmten Cluster zugewiesen werden

Was denken Sie?

Welches Problem sehen Sie hier?

Unsupervised Learning: Dimensionsreduktion





- Ein weiteres Problem des Unsupervised Learnings ist die Dimensionsreduktion
- Ziel ist eine **niedrigdimensionale** Repräsentation der Daten, die die meisten/wichtigsten Eigenschaften dieser erhält
- In unserem Beispiel sehen wir schon visuell, dass eine bestimmte Struktur in den Daten vorhanden ist
- Diese Daten liegen auf einer Spirale – sind also intrinsisch **eindimensional**!
- Die Farbe entspricht einer sog. **latenten Variable** → Eine neue Koordinatenachse, die das Modell entdeckt/gelernt hat

Machine Learning in Python

`scikit-Learn`

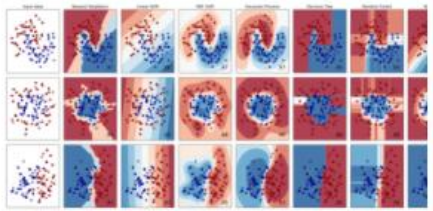
Scikit-Learn: Überblick

https://scikit-learn.org/stable/index.html

Classification

Identifying which category an object belongs to.

Applications: Spam detection, image recognition.
Algorithms: SVM, nearest neighbors, random forest, and more...

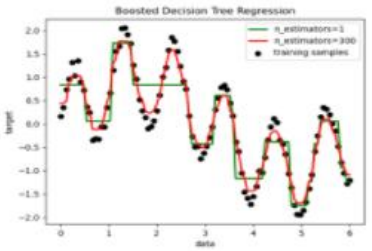


Examples

Regression

Predicting a continuous-valued attribute associated with an object.

Applications: Drug response, Stock prices.
Algorithms: SVR, nearest neighbors, random forest, and more...

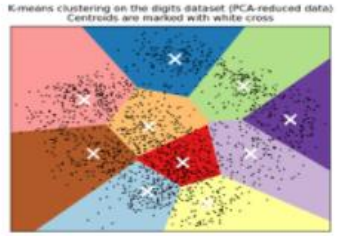


Examples

Clustering

Automatic grouping of similar objects into sets.

Applications: Customer segmentation, Grouping experiment outcomes
Algorithms: k-Means, spectral clustering, mean-shift, and more...

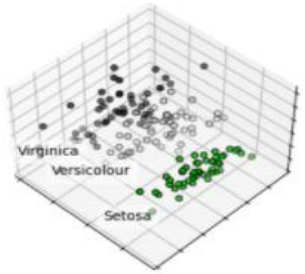


Examples

Dimensionality reduction

Reducing the number of random variables to consider.

Applications: Visualization, Increased efficiency
Algorithms: k-Means, feature selection, non-negative matrix factorization, and more...

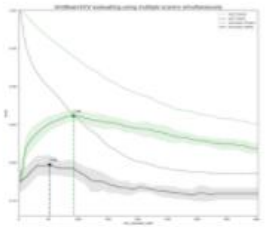


Examples

Model selection

Comparing, validating and choosing parameters and models.

Applications: Improved accuracy via parameter tuning
Algorithms: grid search, cross validation, metrics, and more...

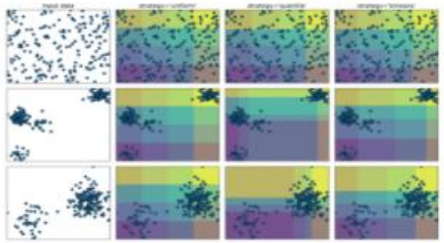


Examples

Preprocessing

Feature extraction and normalization.

Applications: Transforming input data such as text for use with machine learning algorithms.
Algorithms: preprocessing, feature extraction, and more...



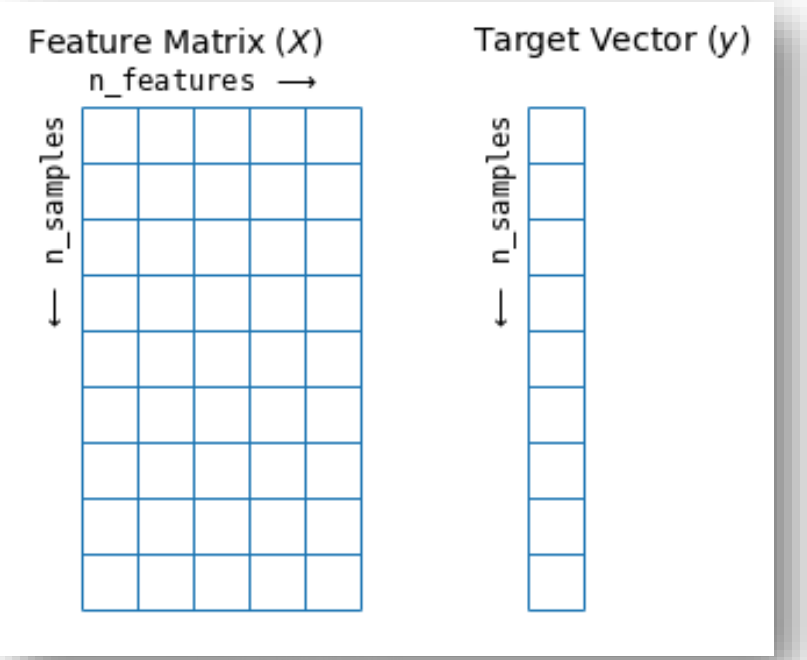
Examples

Scikit-Learn: generischer Aufbau

- `sklearn` hat einen sehr zugänglichen und konsistenten Aufbau
- Jeder Machine Learning Algorithmus hat in `sklearn` den gleichen Aufbau – z.B. im Sinne von Methoden
- Das führt dazu, dass man sogar typische Schritte für die Anwendung von Machine Learning Algorithmen mittels `sklearn` beschreiben kann:
  1. Wähle Modell und importiere entsprechende Klasse aus `sklearn`
  2. Wähle Hyperparameter durch entsprechende Instanziierung
  3. Trainiere/fitte das Modell auf die Daten mittels der `.fit()`-Methode der Modellinstanz
  4. Wende das Modell auf neue Daten an:
     a. Supervised Learning: Labels für neue Daten durch die `.predict()`-Methode vorhersagen
     b. Unsupervised Learning: neue Daten transformieren oder zu Strukturen zuweisen mittels `.transform()`- und `.predict()`-Methode

So what?
Wenn Sie ein Modell anwenden können – können Sie jedes anwenden – zumindest bzgl. der Syntax :)

Wie werden die Daten erwartet?

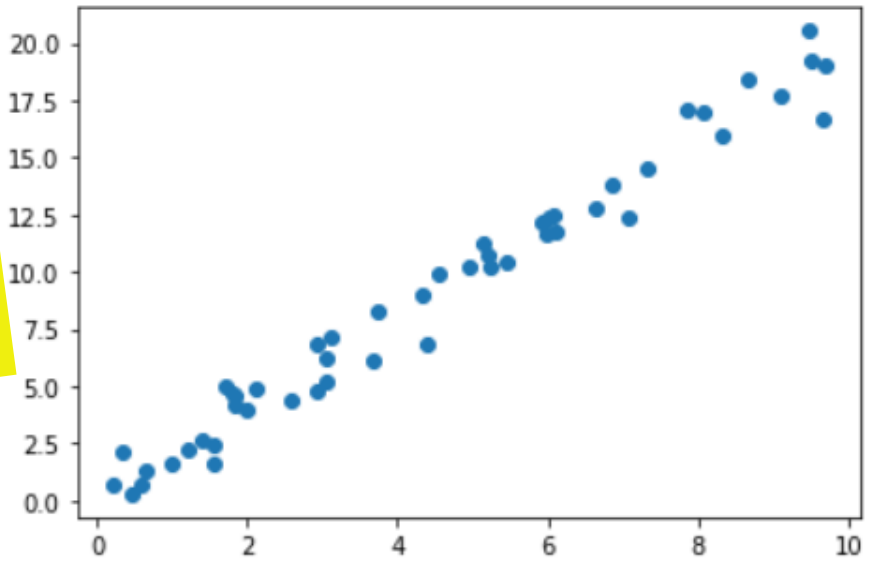


- `sklearn` erwartet die Feature-Matrix und den zugehörigen Target- bzw. Label-Vektor im links abgebildeten Format (das wir schon kennen)
- Die Feature Matrix liegt üblicherweise als NumPy-Array oder Pandas-DataFrame vor
- Der Target-Vektor als NumPy-Array oder Pandas Series

Beispiel: Linear Regression as Exemplary `sklearn` Process

Im Themenblock "Data Understanding: Pair-wise und Multivariate Explorations" haben wir schon ein einfaches Beispiel eines Supervised Learning Modells kennengelernt - die einfache lineare Regression. Wir gehen in diesem Beispiel nun den typischen Ablauf eines Machine Learning Prozesses in `sklearn` durch. Hierzu generieren wir uns folgende Daten.

```
1  rng = np.random.RandomState(42)
2  X = 10 * rng.rand(50, 1)
3  y = 2 * X + rng.randn(50, 1)
4  plt.scatter(X, y);
```



Was denken Sie?

Jetzt haben wir unser erstes exemplarisches Modell trainiert: sind wir schon fertig?

Demo

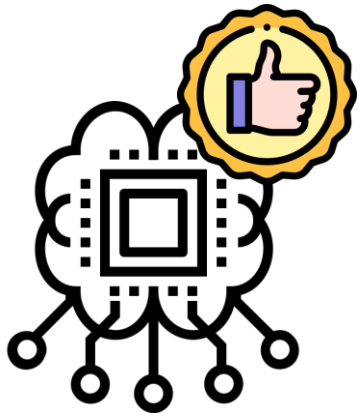
Nutzen Sie `.predict()`, um die Regressionsgerade einzuzeichnen

So what?

Bei Bedarf wiederholen wir wie eine lineare Regression funktioniert

Hyperparameter und Model Validation

Model Validation



Typischer ML Ablauf:

1. Wähle Modell und importiere entsprechende Klasse aus `sklearn`
2. Wähle Hyperparameter durch entsprechende Instanziierung
3. Trainiere/fitte das Modell auf die Daten
4. Wende das Modell auf neue Daten an

Was denken Sie?

Wie würden Sie intuitiv an die Modellvalidierung herangehen?

- Die Punkte 1. und 2. sind von entscheidender Bedeutung
- Um eine objektive Wahl für das Modell und die zugehörigen Hyperparameter treffen zu können, müssen wir das verwendete Modell validieren
- Das Modell und die gewählten Hyperparameter müssen die **zugrundeliegenden Daten** gut **repräsentieren**

Model Validation: the „not so right“ way

```python
# Get some data
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
y = iris.target
```

```python
# Get some model and hyperparameter
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_neighbors=1)
```

```python
# Train the model to our data
model.fit(X, y)
y_model = model.predict(X)
```

```python
# Test on the training set
from sklearn.metrics import accuracy_score
accuracy_score(y, y_model)
```

Bauchgefühl

Intuitiver Ansatz:

Nachdem wir das Modell mit den ausgesuchten Hyperparametern trainiert haben, wenden wir es auf einen Teil dieser Daten an und vergleichen den vorhergesagten Wert mit dem tatsächlichen.

- Wir erhalten ein Modell, das zu 100% genau die Labels der Daten vorhersagen kann
- Problem: wir trainieren UND validieren das Modell an denselben Daten!

Was denken Sie?

Was tun wir jetzt, um dieses Problem zu beheben?

Model Validation: Einschub - Accuracy und Model-Score

```
# Fit the model on one set of data
model.fit(X_train, y_train)

# Evaluate the model on the second set of data
y_predicted = model.predict(X_test)
accuracy_score(y_test, y_predicted)
```

```
0.96
```

```
model.score(X_test, y_test)
```

```
0.96
```

- In den vorhergehenden Folien haben wir kennengelernt, wie man die Vorhersage- bzw. Klassifikationsgenauigkeit eines Modells berechnet
- Hierzu gibt es die Funktion `accuracy_score` aus dem Modul `sklearn.metrics`

  `accuracy_score(ytest, ypredicted)`
- Weiterhin haben Sie die Möglichkeit diese Berechnung mittels der Methode `.score()`, die sich in den meisten Modellen befindet, durchzuführen
  → Diese Methode nimmt dann folgende Eingabeargumente auf

  `model.score(Xtest, ytest)`

Model Validation: the right way: Houldout Sets

```
1  X.shape
```

```
(150, 4)
```

```
1  y.shape
```

```
(150,)
```

```
1  # Import
2  from sklearn.model_selection import train_test_split
3
4  # split the data in train/test sets
5  X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.5)
```

```
1  print(X_train.shape)
2  print(X_test.shape)
3  print(y_train.shape)
4  print(y_test.shape)
```

Was denken Sie?

Sie rufen die `train_test_split`-Funtion mehrmals auf: bekommen Sie jedes Mal dasselbe Ergebnis?

- Um die Model Performance nicht an Datensätzen zu testen, die das Modell schon „gesehen“ hat
  → **Holdout Sets**
- Der Datensatz wird in **Training- und Test-Sets** aufgeteilt
- Am Training-Set wird das Modell trainiert bzw. angelernt bzw. gefittet
- Am Test-Set wird die Genauigkeit des trainierten Modells überprüft
  → Man sagt: ein genaues Modell kann gut **generalisieren**

```
1  # Fit the model on one set of data
2  model.fit(X_train, y_train)
3
4  # Evaluate the model on the second set
5  y_predicted = model.predict(X_test)
6  accuracy_score(y_test, y_predicted)
```

```
0.96
```

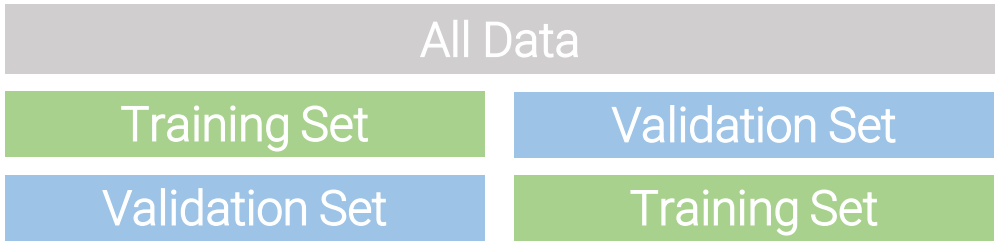
Was denken Sie?

Wie könnten wir diesen Gedanken nun weiterdenken? Was ist ein Nachteil hierbei?

`sklearn` hat hierzu eine vorimplementierte Funktion

```
train_test_split(X, y, train_size=<ratio>)
```

Model Validation: Cross-Validation




```python
# split the data in train/test sets
X1, X2, y1, y2 = train_test_split(X, y, train_size=0.5)

# Two-fold Cross-Validation
y2_model = model.fit(X1, y1).predict(X2)
y1_model = model.fit(X2, y2).predict(X1)

```

- Nachteil Holdout Set: Man **„verliert" Daten**, an denen Man das Modell trainieren kann
- Daher, in obigem Fall: nutze **jeden Teil** des Datensatzes je einmal als **Training- und Test-Set**
- Jeder Teil des Datensatzes wird einmal als Houldout-Set verwendet
- Das führt im Endeffekt zu **zwei Genauigkeitswerten** des Modells
- Diese kann man dann **aggregieren** – z.B. mitteln – und dann hat man eine „globale" Performance-Bewertung des Modells
- Diese Art der **Cross-Validation** heißt two-fold cross validation

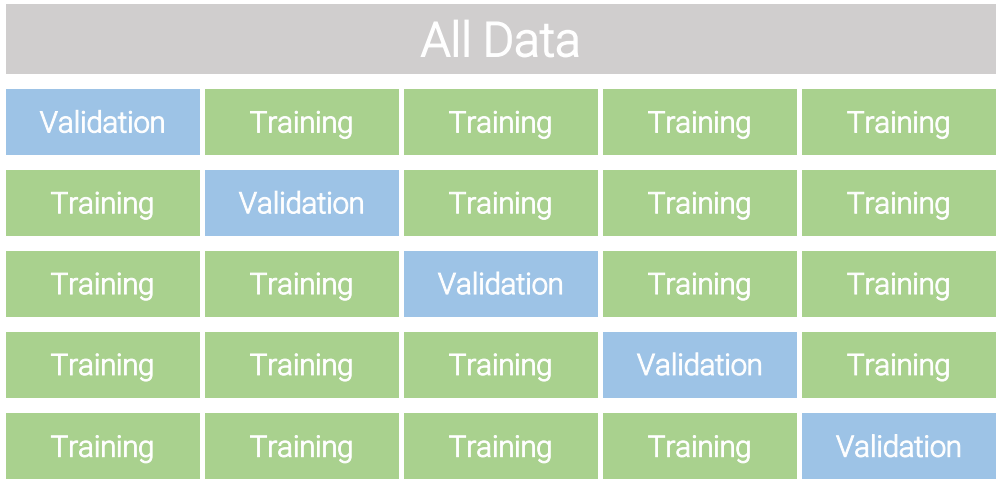
Was denken Sie?

Wie geht es wohl nun konzeptuell weiter? Was wird eine weitere Prämisse für uns sein?

Was denken Sie?

Was tun wir mit diesen Genauigkeitswerten?

Model Validation: Cross-Validation




```
1  from sklearn.model_selection import cross_val_score
2  cross_val_score(model, X, y, cv=5)
```

```
array([0.96666667, 0.96666667, 0.93333333, 0.93333333, 1.        ])
```

```
1  from sklearn.model_selection import LeaveOneOut
2  scores = cross_val_score(model, X, y, cv=LeaveOneOut())
3  scores
```

```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 0., 1., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
```

- Diesen Gedanken können wir weiterverfolgen
  → Mehr Splits
- Hierzu hat `sklearn` eine Funktion, um Cross-Validation mit einer beliebigen Anzahl an Splits zu verwirklichen

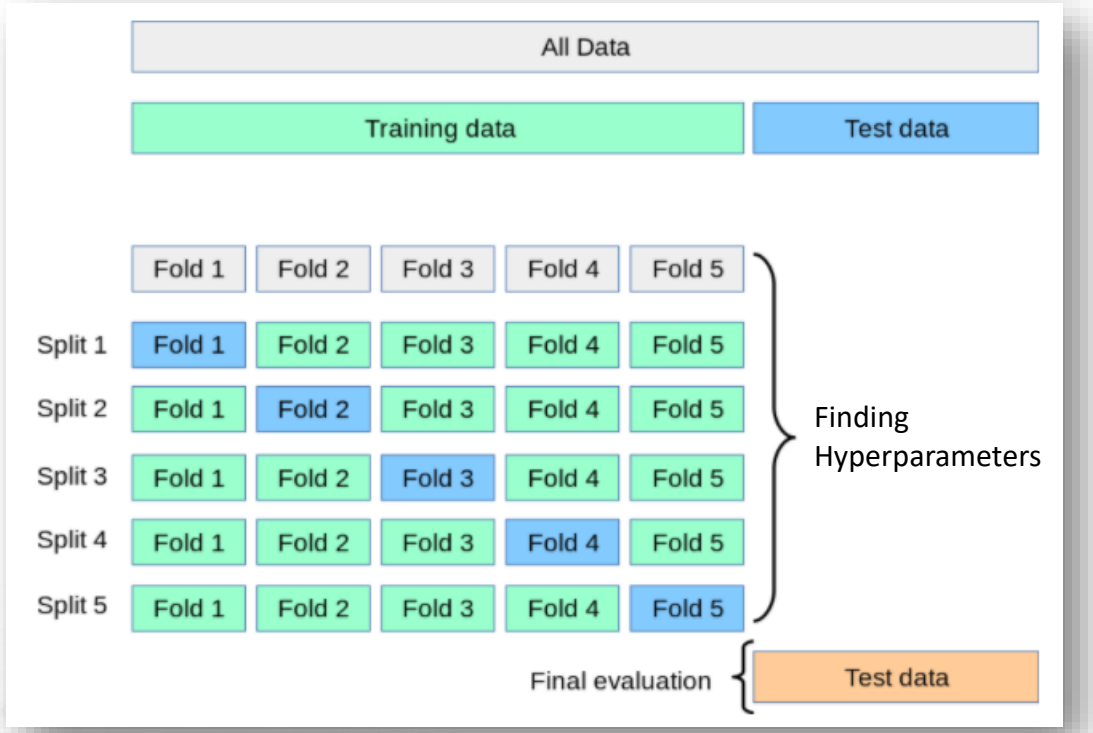
  `cross_val_score(<model>, X, y, cv=<number_of_splits>)`
- Diese Funktion führt im Grunde Training, Prediction und Accuracy Score auf einmal aus – für jeden der Splits!
- Die Extremform dieses Konzepts nennt man Leave-one-out Cross-Validation
  → Das Validation-Set besteht nur noch aus einem Datenpunkt
  → Es werden so viele Splits durchgeführt wie Datenpunkte vorhanden sind

Was denken Sie?

Wie sieht die Extremform dieses Konzepts aus?

Train, Test, Validation Split

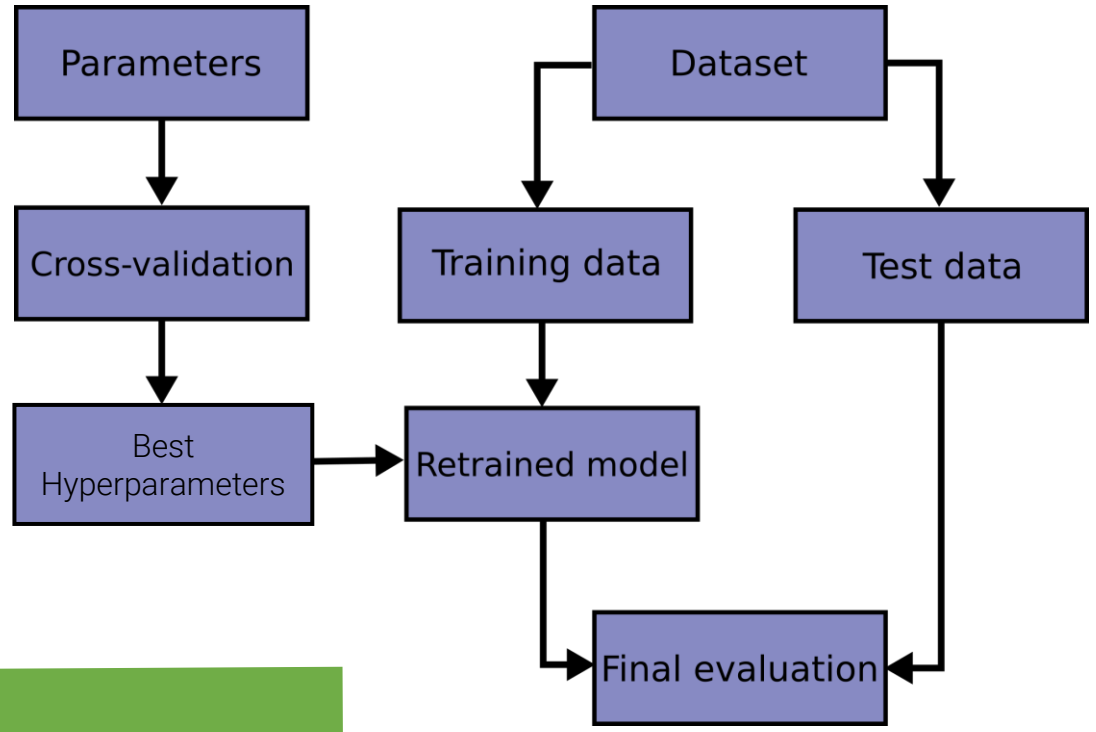
https://scikit-learn.org/stable/modules/cross_validation.html



- Jetzt haben wir gelernt wie wir die **(Generalisierungs-) Performance** eines Machine Learning Modells bewerten können
- Der Ansatz mit Cross-Validation birgt jedoch immer noch ein Problem: die Parameter können theoretisch **zu genau** an die Trainings- und Testdatensätze **angepasst** werden
- Daher ist der State-of-the-Art die Aufteilung der Daten in:
  - **Test-Set**: wird für die abschließende Bewertung der Modell-Performance genutzt
  - **Validation-Sets**
  - **Training-Sets**
- Man nutzt soz. die Training- und Validation-Sets, um die **optimalen Hyperparameter** des Modells zu finden
- Und das eigentliche Test-Set, um die Modell-Performance **realistisch** bewerten zu können

Model Training Workflow

https://scikit-learn.org/stable/modules/cross_validation.html



So what?

Achtung: was hier evtl. verwirrend sein könnte: es gibt einen Unterschied zwischen den **Hyperparametern** und den **Parametern** eines Modells. :)

Repeated k-fold Cross Validation

Was denken Sie?

Treiben Sie den Ansatz auf die Spitze: wo/wie können Sie noch Artefakte vermeiden?

https://scikit-learn.org/stable/modules/cross_validation.html#repeated-k-fold

Nested k-fold Cross Validation

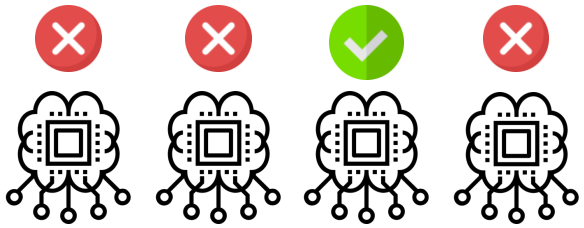
Grafik hierzu einfügen – gibt eine schöne in einer Abschlussarbeit, die ich betreut habe – welche?

https://machinelearningmastery.com/repeated-k-fold-cross-validation-with-python/

Hyperparameter: Selecting the Best Model

Definition

Wir nehmen uns schon mal mit: der Wert eines *Hyperparameters*, im Gegensatz zu dem eines *Parameters*, eines Modells wird nicht durch das Training bestimmt, sondern vorab festgelegt.



- Jetzt haben wir Cross-Validation als eine Methode kennengelernt, um sog. *Hyperparameter* von Modellen bewerten zu können
- **Leitfrage: wie sollen wir weitermachen, falls unser Modell eine zu niedrige Genauigkeit aufweist?**
- **Mehrere Möglichkeiten**:
  - Modellkomplexität erhöhen
  - Modellkomplexität verringern
  - Mehr Trainingsdaten
  - Andere/mehr Features

  → Hyperparameter-Tuning

→ Das, was wir in den kommenden Folien erfahren werden, scheint – zumindest teilweise – kontraintuitiv zu sein

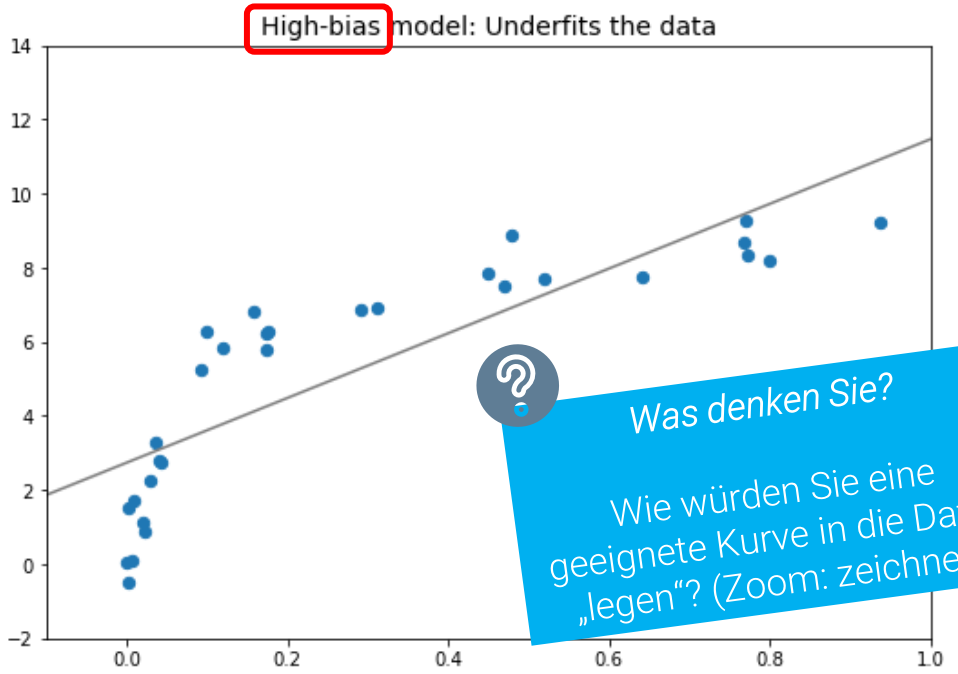
So what?

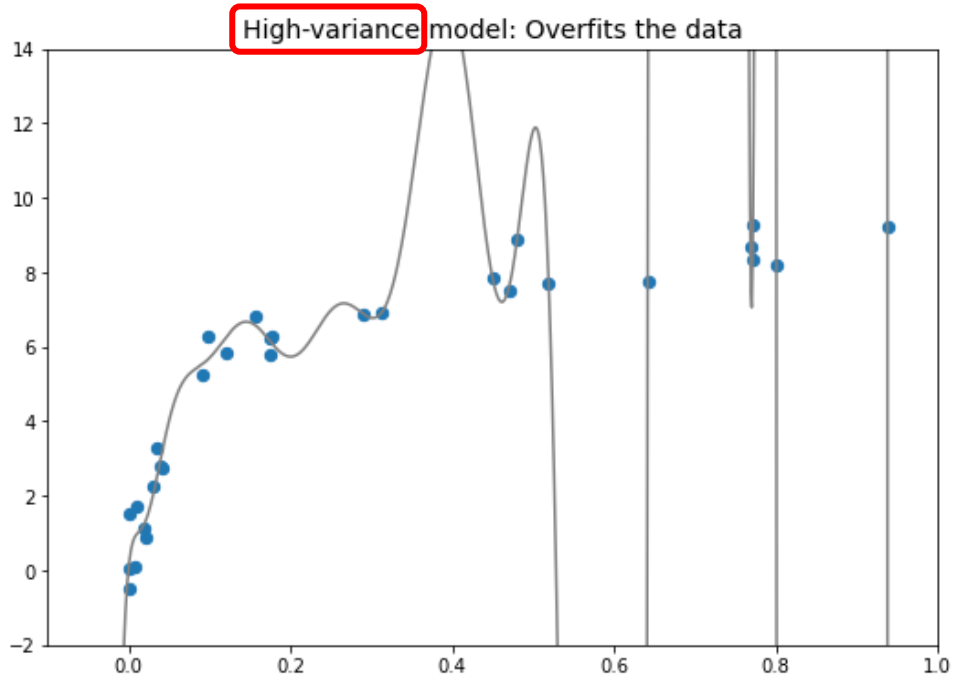
Die Wahl des besten bzw. optimalen Modells anhand des sog. *Hyperparameter-Tunings* ist einer der wichtigsten Aspekte des Machine Learning

Was denken Sie?

Wie würden Sie weitermachen?

Modellkomplexität: der Bias-Variance Trade-off





Was denken Sie?

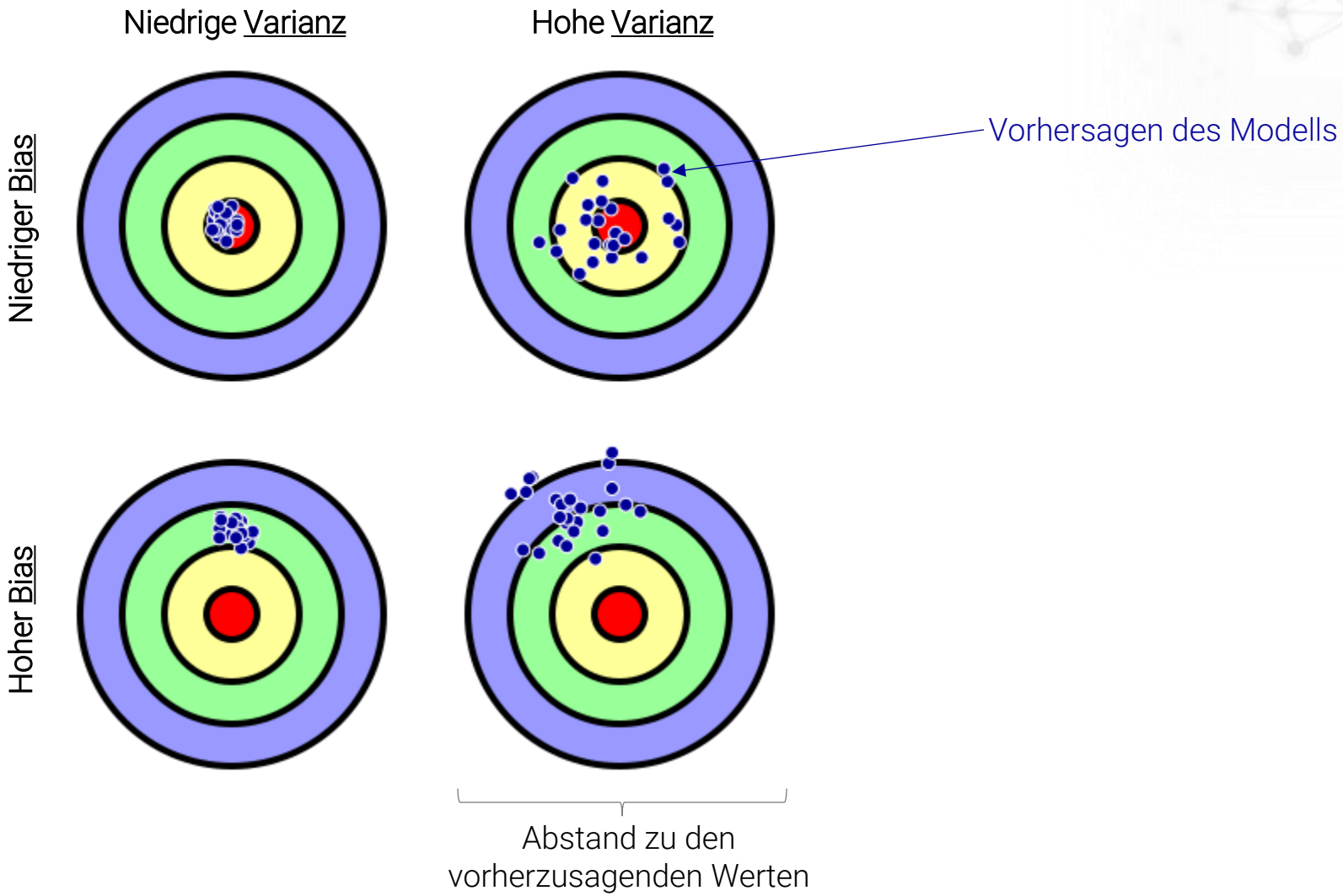
Wie würden Sie eine geeignete Kurve in die Daten „legen“? (Zoom: zeichnen)

- Das optimale Modell zu finden == Balance zwischen dem sog. **Bias** und der **Varianz** eines Modells zu finden
- Wir nutzen den regressiven Fit einer **Polynomfunktion** auf vorhandene Trainingsdaten als Beispiel eines Modells
- Unsere beiden Beispiele für einen Fit scheinen ungeeignet zu sein → der gesuchte Fit scheint also „zwischen“ diesen beiden Extrema zu liegen
- **High-bias** model → Daten werden „**underfitted**“
- **High-variance** model → Daten werden „**overfitted**“

Was denken Sie?

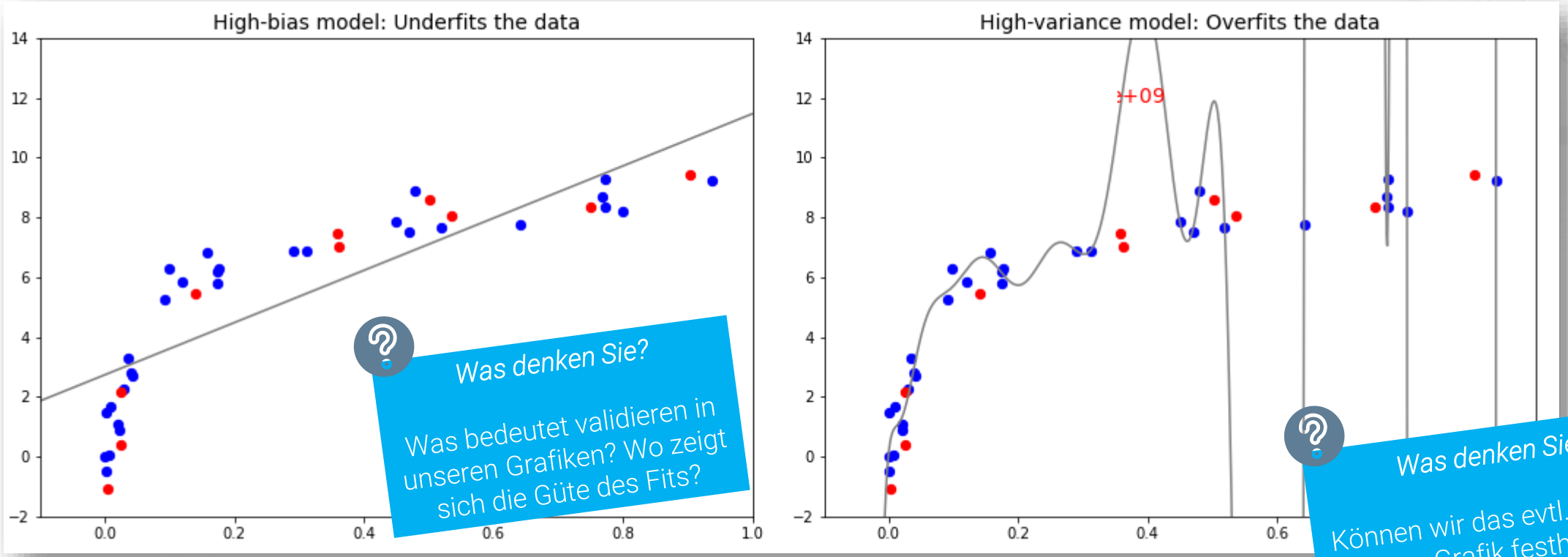
Wie überprüfen wir den Fit? Da fehlt doch was, oder?

Der Bias-Variance Trade-off: eine andere Perspektive



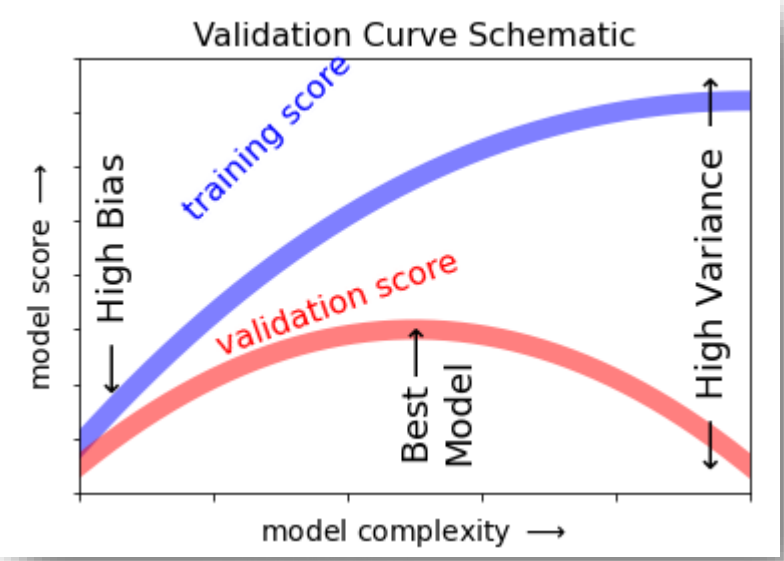
So what?
Quantitative Methoden: Bias und Variance haben einen Bezug zu Validität und Reliabilität!

Modellkomplexität: der Bias-Variance Trade-off



- Erinnern Sie sich an die Methode der Cross-Validation: was passiert, wenn wir das Modell an **Testdaten** validieren?
- Unser *Gütemaß* der **Validierung** befindet sich bei den **underfitted** Daten in einem **vergleichbaren** (schlechten) Wertebereich, wie beim **Training** – bei den **overfitted** Daten ist unser Gütemaß deutlich **kleiner** in der **Validierung** als beim **Training**
  → Das ist ein generelles Muster!

Modellkomplexität: der Bias-Variance Trade-off



Was denken Sie?

Welche Charakteristiken sehen Sie? Und warum?

- Wir stellen uns nun vor, dass wir die Modellkomplexität **tunen** bzw. **sweepen** können
- Würden wir für jede erzeugte Modellkomplexität einen Trainings- und Validierungs-Score berechnen, dann würde man gegenüberliegende Grafik erwarten
  → eine Validierungskurve
- Charakteristiken:
  - Der Trainings-Score ist immer höher als der Validierungs-Score
  - Niedrige Modellkomplexität (High-Bias): Underfitting der Trainingsdaten und schlechte Schätzung ungesehener Daten
  - Hohe Modellkomplexität (High-Variance): Overfitting der Trainingsdaten → d.h. die Trainingsdaten werden (zu) gut abgebildet – die Testdaten jedoch zu schlecht
  - Es gibt ein Optimum: das Maximum des Validation-Scores → bester Trade-off zwischen Bias und Varianz eines Modells

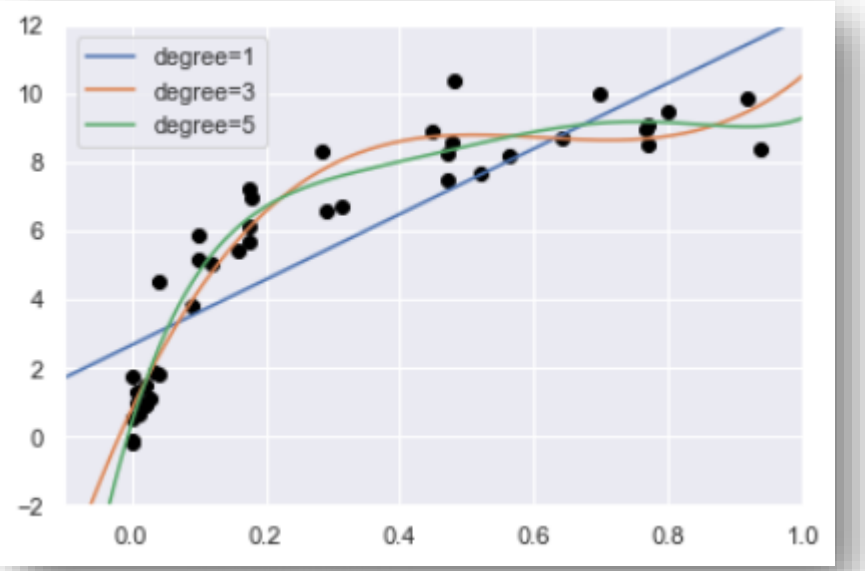
Einschub: Polynomiale Regression

Was denken Sie?

Was bedeutet „angepasst“ bzgl. der Formel?

Was denken Sie?

Was sind hier unsere Hyperparameter?



Was denken Sie?

Was ist hier ein High-Bias und High-Variance Model?

- Unser Repräsentant eines Machine Learning Modells soll nun eine Polynomfunktion beliebigen Grades sein

$$f(x) = a_n x^n + \cdots + a_1 x + a_0$$

- Diese wird mittels einer sog. Polynomial Regression an uns vorliegenden Daten angepasst
- **Durch die Anpassung werden die Koeffizienten geschätzt!**
- Die Hyperparameter sind in diesem Fall die Anzahl der Summanden bzw. der Grad des Polynoms → **repräsentiert die Modellkomplexität!**
- Diese werden getuned bzw. gesweept: d.h. zuerst wird ein Polynom des 1. Grades, dann des 2. Grades, etc. an die Daten angepasst

$$f(x) = a_1 x + a_0$$
$$f(x) = a_2 x^2 + a_1 x + a_0$$
$$\vdots$$

- Vorhergehende Folien: ein High-Bias Model hatte also einen **niedrigen Grad** – ein High-Variance Model einen **hohen Grad**

Beispiel: Polynomiale Regression

In diesem Beispiel schauen wir uns eine Polynomfunktion in Python näher an. Hierzu müssen wir auf die `PolynomialFeatures` und die `LinearRegression` Klassen zugreifen, die wir in `sklearn` finden. Schauen wir uns den Docstring der `PolynomialFeatures`

```
Docstring:
Generate polynomial and interaction features.

Generate a new feature matrix consisting of all polynomial combinations
of the features with degree less than or equal to the specified degree.
For example, if an input sample is two dimensional and of the form
[a, b], the degree-2 polynomial features are [1, a, b, a^2, ab, b^2].
```

und der `LinearRegression`

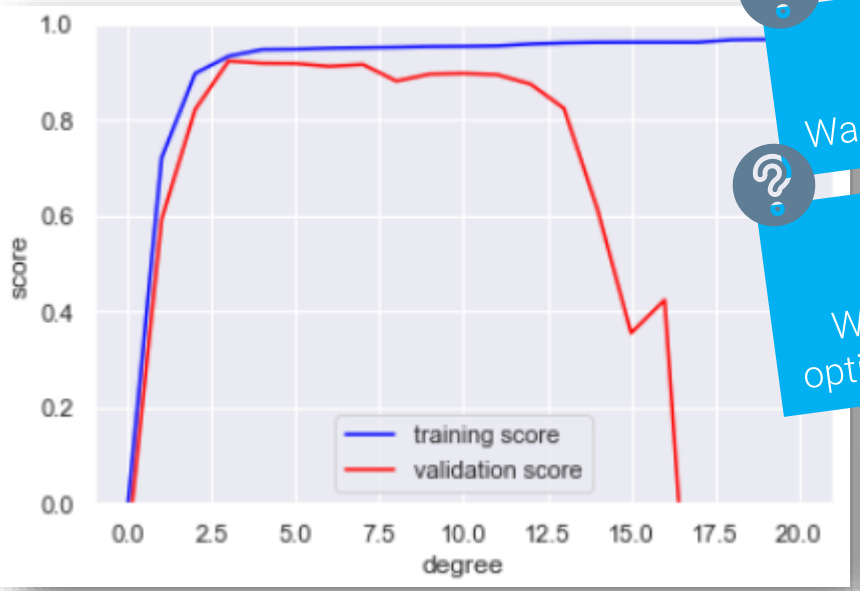
```
LinearRegression fits a linear model with coefficients w = (w1, ..., wp)
to minimize the residual sum of squares between the observed targets in
the dataset, and the targets predicted by the linear approximation.
```

an. Was müssen wir wohl tun, um eine *polynomiale Regression* durchzuführen?

So what?

Für jeden Fit gibt es also auch ein Gütemaß. Dieses Gütemaß kann man sowohl für den Trainings-, als auch Testdatensatz berechnen.
→ Daraus ergeben sich dann die *Validierungskurven*

Validierungskurven in `sklearn`

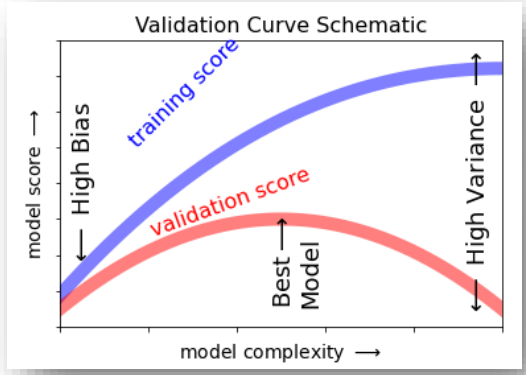


Was denken Sie?

Was ist nun das Optimum?

Was denken Sie?

Wie findet man nun den optimalen Hyperparameter?

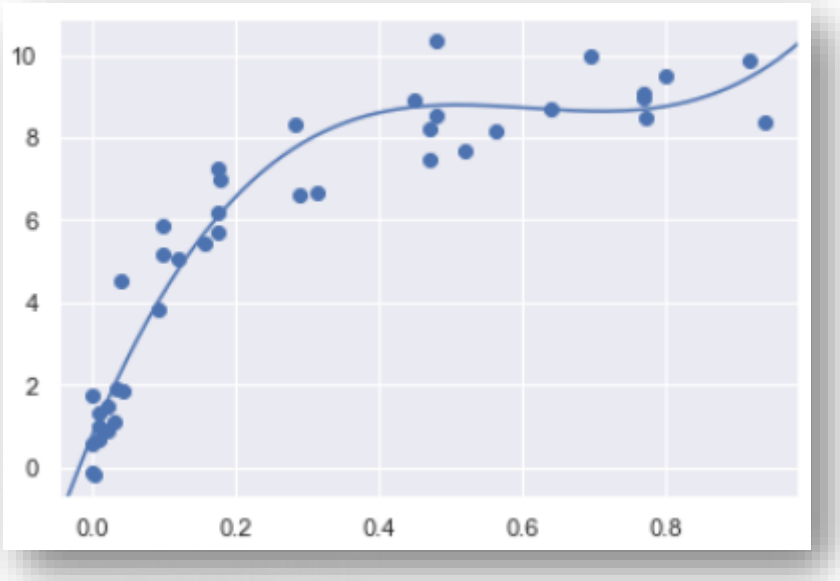



```python
from sklearn.model_selection import validation_curve
degree = np.arange(0, 21)
train_score, val_score = validation_curve(PolynomialRegression(), X, y,
                                          param_name='polynomialfeatures__degree', param_range=degree, cv=7)
```

- Wir visualisieren uns die Validierungskurven: d.h. wir plotten den Trainings- und Validierungs-Score für jeden untersuchten Polynomgrad
- Hierzu gibt es in sklearn eine Funktion

```
validation_curve(<model>, X, y, param_name=<hyperparameter>,
param_range=<hyperparameter_range>, cv)
```

Validierungskurven in `sklearn`



- Unser Optimaler Hyperparameter liegt also bei einem Polynomgrad von 3
- Hier hat unser Validierungs-Score das Maximum

So what?

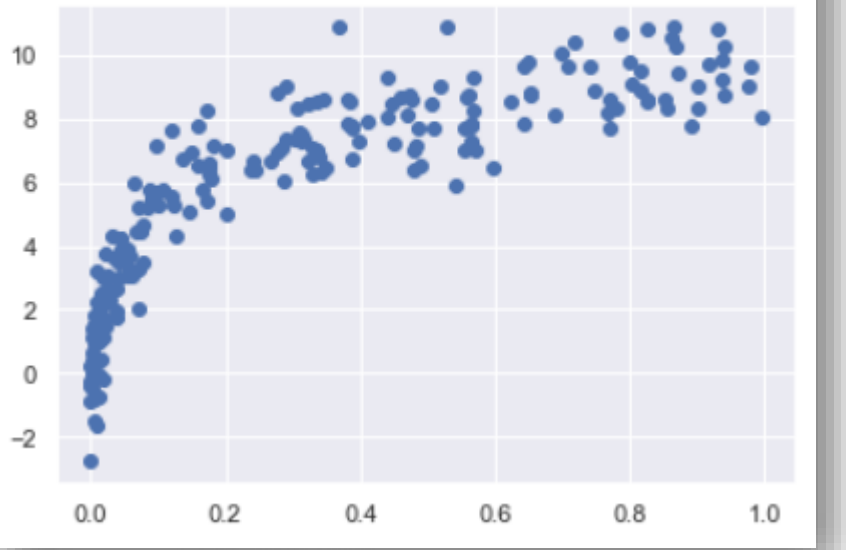
- Wir hätten dieses Ergebnis auch nur durch die Betrachtung bzw. die Maximierung des Validierungs-Scores erhalten können
- Aber die Visualisierung beider Kurven kann uns weitere Einsichten in das Modell geben

Learning Curves

Was denken Sie?

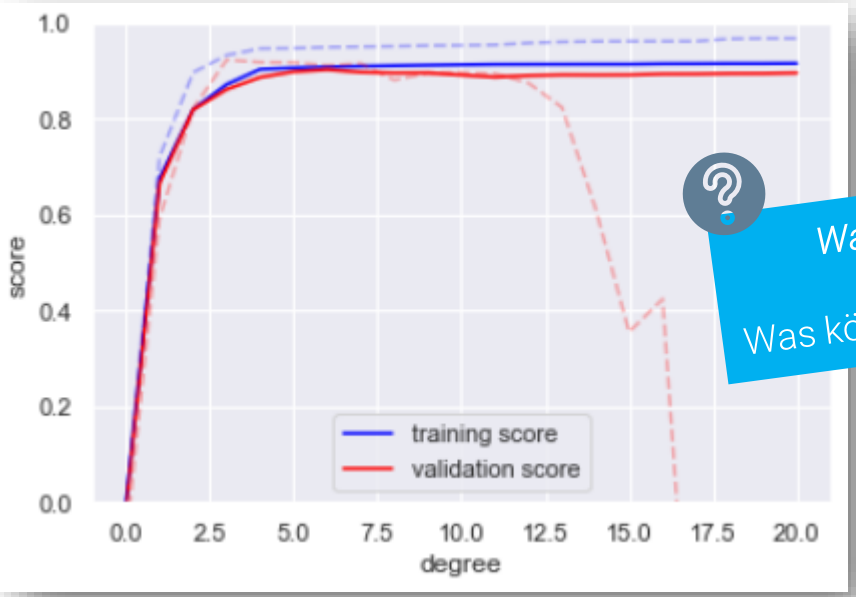
Wie können wir unser Modell noch verbessern?

```
1 X2, y2 = make_data(200)
2 plt.scatter(X2.ravel(), y2);
```



- Mehr Daten!
- Mittels sog. *Learning Curves* kann man den Einfluss der Datensatzgröße auf die Modell Performance untersuchen
- Hierzu wählt man eine Modellkomplexität aus und trägt den Validierungs- und Trainings-Score gegen die Datensatzgröße auf

Learning Curves



Was denken Sie?

Was können wir erkennen?

- Bevor wir das tun, vergleichen wir unsere ursprüngliche Validierungskurve mit der neuen – bzgl. mehr Daten
- Bei einem **größeren Datensatz** kann man anscheinend eine **höhere Modellkomplexität** verwenden
- Die Validierungskurve hängt also nicht nur von der Modellkomplexität ab, sondern auch von der Menge der Daten!

→ Aus der Untersuchung des Einflusses der Datensatzgröße auf die Validierungskurve resultieren die *Learning Curves*

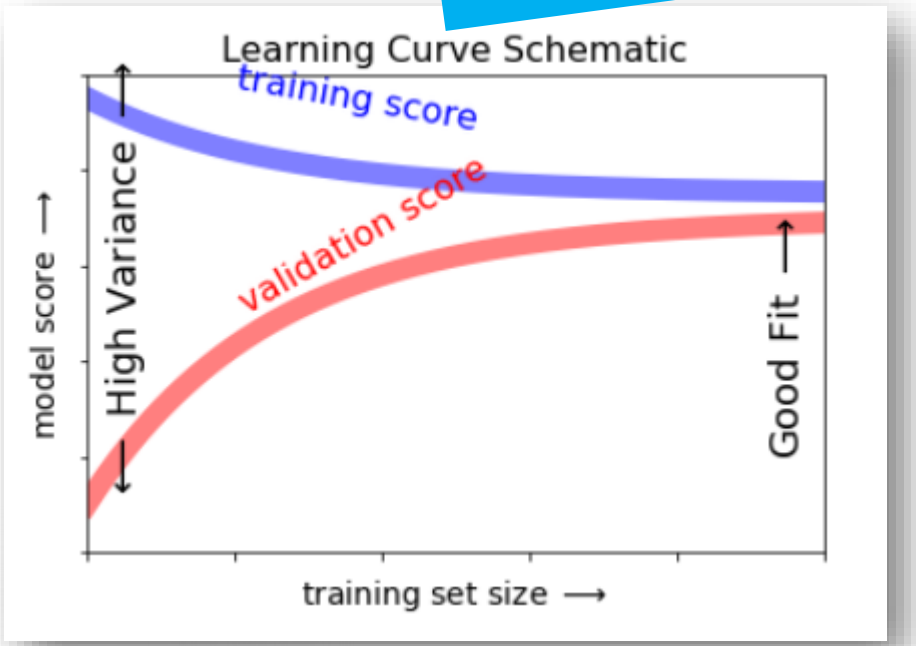
```
degree = np.arange(21)
train_score2, val_score2 = validation_curve(PolynomialRegression(), X2, y2,
                                            param_name='polynomialfeatures__degree', param_range=degree, cv=7)

plt.plot(degree, np.median(train_score2, 1), color='blue', label='training score')
plt.plot(degree, np.median(val_score2, 1), color='red', label='validation score')
plt.plot(degree, np.median(train_score, 1), color='blue', alpha=0.3, linestyle='dashed')
plt.plot(degree, np.median(val_score, 1), color='red', alpha=0.3, linestyle='dashed')
plt.legend(loc='lower center')
plt.ylim(0, 1)
plt.xlabel('degree')
plt.ylabel('score');
```

Learning Curves

Was denken Sie?

Bei fester Modellkomplexität: welches Verhalten würden wir mit zunehmender Datensatzgröße erwarten?



Allgemeine Charakteristiken von Learning Curves:

- Bei gegebener Modellkomplexität und kleinem Datensatz kommt es zu einem *Overfitting*
- ~~Bei gegebener Modellkomplexität und großem Datensatz kommt es zu einem „*Underfitting*“~~
- Der Validierungs-Score wird nie – außer durch Zufall – größer sein als der Trainings-Score

So what?

- Es kommt zu einer Konvergenz mit zunehmender Datensatzgröße!
- Ab einem gewissen Punkt bringt die Vergrößerung des Datensatzes keinen Mehrwert mehr!

Was denken Sie?

Wodurch können wir nun nur noch die Modell Performance erhöhen?

Was denken Sie?

Was passiert also mit zunehmender Datensatzgröße?

Learning Curves in `sklearn`

Demo
Ausführung im
Jupyter Notebook

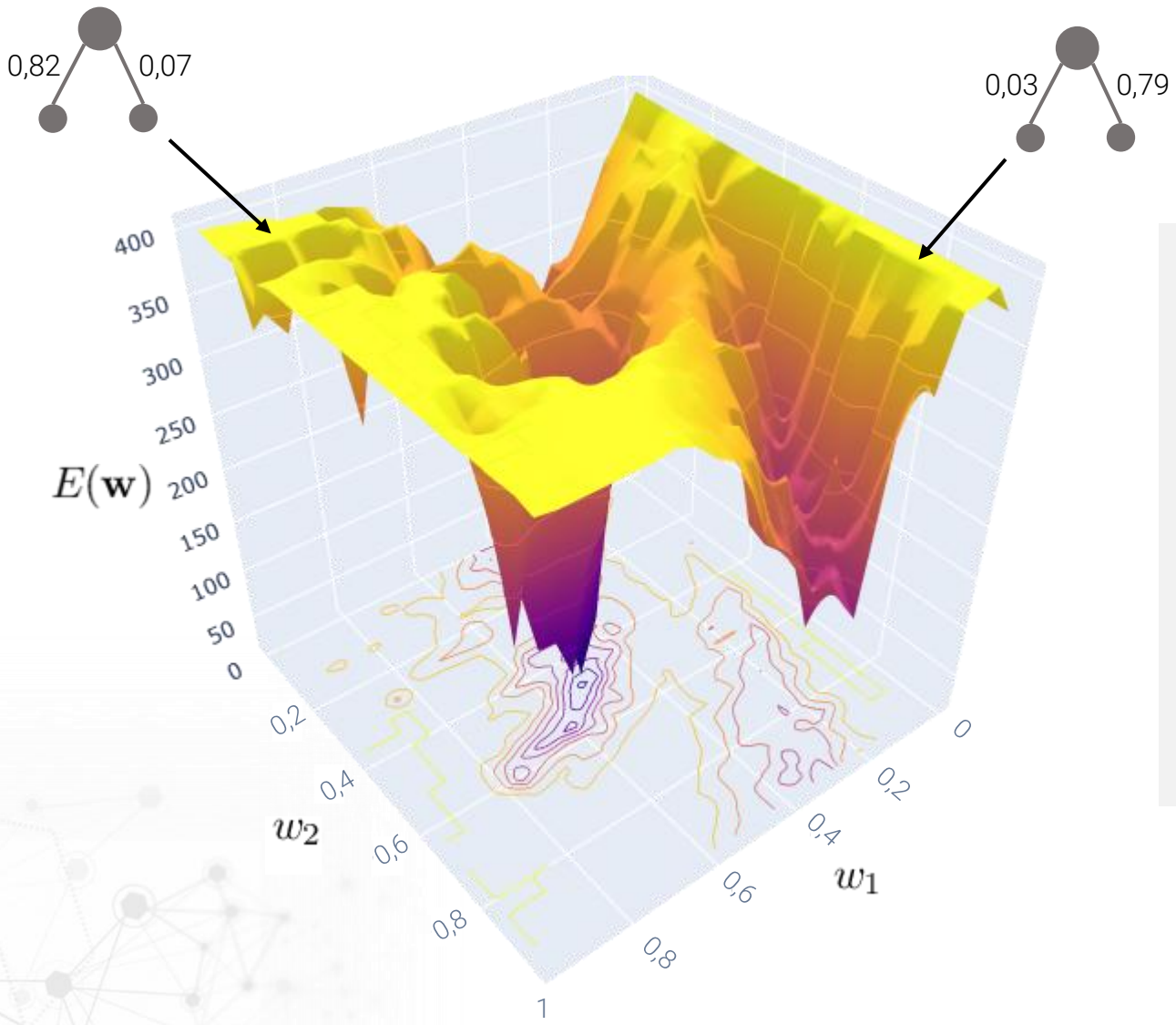
Auch hier liefert `sklearn` eine Funktion, um Learning Curves zu implementieren:

```
learning_curve(<model>, X, y, cv=<cv>, train_sizes=<train_sizes>)
```

Was denken Sie?

Was zeigt uns das Konvergenzverhalten?

Validierung in der Praxis: Grid Search



Was denken Sie?

Was liegt uns jetzt vor?

- Validierungs- und Lernkurven haben uns einen Einblick in das Trainieren von Machine Learning Algorithmen verschafft
- In der Praxis haben Machine Learning Modelle mehr als einen Hyperparameter
- Dadurch werden aus Validierungskurven **mehrdimensionale** „Flächen“
- Daher untersucht man in der Praxis nur den Validierungs-Score → Optimum == globales Maximum

Validierung in der Praxis: Grid Search

Was denken Sie?

Wieso sagt man Grid Search?

```
1  from sklearn.model_selection import GridSearchCV
2
3  param_grid = {'polynomialfeatures__degree': np.arange(21),
4                'linearregression__fit_intercept': [True, False],
5                'linearregression__normalize': [True, False]}
6
7  grid = GridSearchCV(PolynomialRegression(), param_grid, cv=7)
```

```
1  grid
```

```
GridSearchCV(cv=7,
             estimator=Pipeline(steps=[('polynomialfeatures',
                                        PolynomialFeatures()),
                                       ('linearregression',
                                        LinearRegression())]),
             param_grid={'linearregression__fit_intercept': [True, False],
                         'linearregression__normalize': [True, False],
                         'polynomialfeatures__degree': array([ 0,  1,  2,  3,
       17, 18, 19, 20])})
```

- Beim Grid Search gibt man vorab bestimmte **Bereiche** der **Hyperparameter** des jeweiligen Modells vor
- Diese Bereiche werden „abgefahren“ bzw. „gesweeped“ bzw. „getuned“ – d.h. für **jede Parameterkombination** wird **ein Modell** trainiert
- Jedes dieser Modelle resultiert in einen **Validierungs-Score**
- Die Aufgabe des Grid Search ist dann das **Optimum** bzw. Maximum des Validierungs-Scores in Abhängigkeit von der Parametereinstellung zu finden
- In sklearn gibt es hierzu eine Klasse

  `GridSearchCV(<model>, param_grid, cv=<cv>)`

Validierung in der Praxis: Grid Search

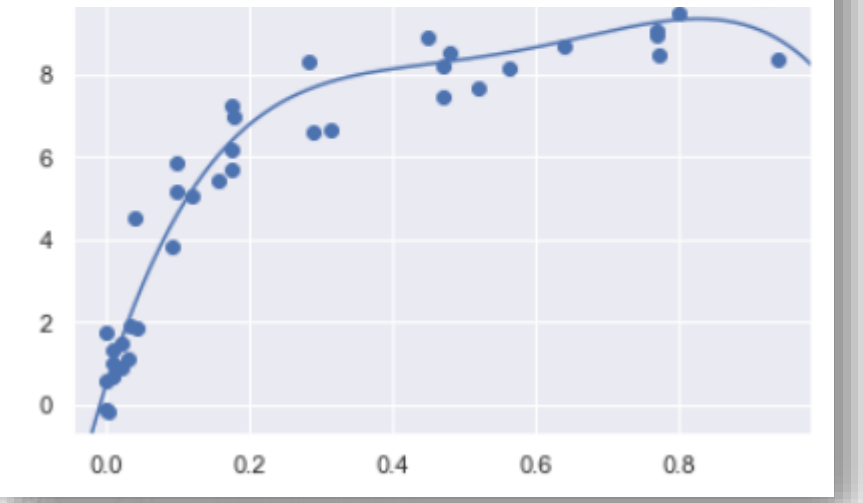
```
1 grid.fit(X, y);
```

```
1 grid.best_params_
```

```
{'linearregression__fit_intercept': False,
 'linearregression__normalize': True,
 'polynomialfeatures__degree': 4}
```

```
model = grid.best_estimator_

plt.scatter(X.ravel(), y)
lim = plt.axis()
X_new = np.linspace(0, 1, 100).reshape(-1, 1)
y_new = model.fit(X, y).predict(X_new)
plt.plot(X_new, y_new);
plt.axis(lim);
```



- Wie ein normales Machine Learning Modell auch, können wir die Grid Search Instanz auf unsere Daten anpassen – und zwar mit der `.fit()`-Methode
- Wir können uns dann die beste Hyperparameter-Kombination ausgeben lassen
- Und diese dann nutzen

Too much information…

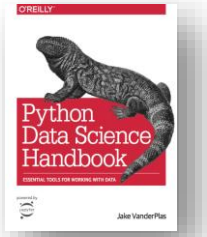
#toomuchinformation

- Jetzt haben wir uns **viele** Informationen zu Gemüte geführt
- Haben viele **generische** Machine Learning **Konzepte** kennengelernt
- Und das auch noch mit `sklearn` verknüpft
- Jetzt wird es Zeit für konkrete Algorithmen und **anwendungsorientierte** Aufgaben

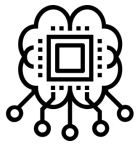
So what?

- Die Aspekte, die wir in diesem Themenblock kennengelernt haben, werden häufig nur implizit vermittelt – parallel zur Einführung einzelner ML-Modelle
- Sehen Sie das hier auch als Referenz: wir werden immer wieder auf diesen Foliensatz zurückkommen

Quellen



Jake VanderPlas. *Python Data Science Handbook: Essential Tools for working with Data*. O'Reilly UK Ltd. 2016.
→ https://github.com/jakevdp/PythonDataScienceHandbook



Icon made by https://www.freepik.com from https://www.flaticon.com/

Icon made by https://www.flaticon.com/authors/vectors-market Vectors Market from https://www.flaticon.com/

Icon made by https://www.freepik.com from https://www.flaticon.com/

Icon made by https://www.freepik.com from https://www.flaticon.com/

Quellen

https://media.giphy.com/media/xUA7b1XxX2owHd3Vw4/giphy.gif?cid=ecf05e47li0ps41v3k5nuewpfzlao1p8l1xgeorrcq5t0izg&rid=giphy.gif&ct=g

Icons made by https://www.freepik.com from https://www.flaticon.com/

https://giphy.com/embed/E5jmd4OQ7PSo7JKlAR