Data Science 2

Regressionsverfahren

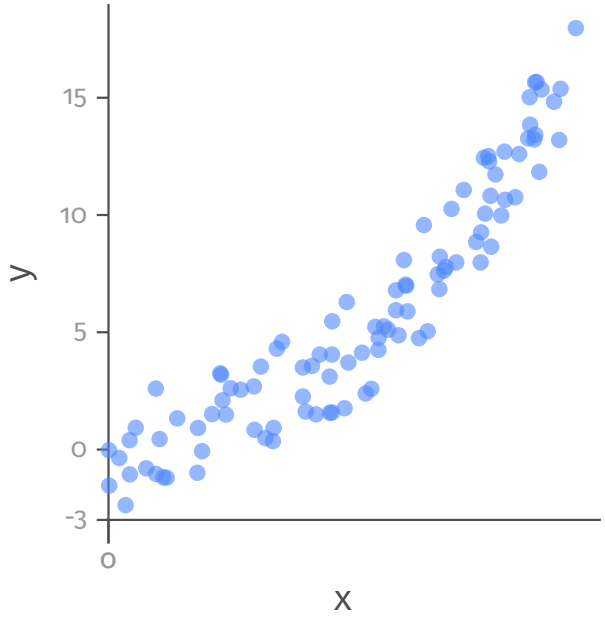
Prof. Dr. Christian Bockermann

Hochschule Bochum

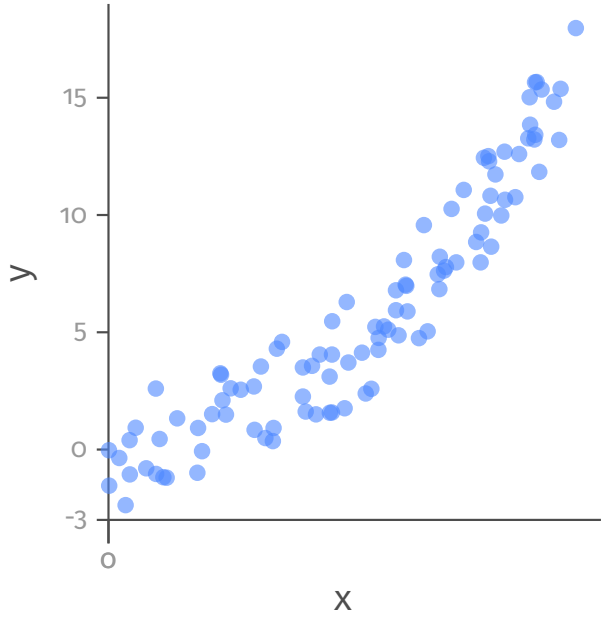
Wintersemester 2023 / 2024

Inhalt

Was ist, wenn die Daten nicht linear erklärbar sind?



Was ist, wenn die Daten nicht linear erklärbar sind?



Funktionsklasse linearer Funktionen
reicht nicht immer aus!

Idee: Erweiterung auf komplexere Funktionen

Polynom d-ten Grades, ein-dimensionale Daten $\mathbf{X}$:

$$\begin{aligned} p(x) &= a_0 + a_1 x + a_2 x^2 + \ldots + a_d x^d \\ &= \sum_{i=0}^{d} a_i x_i^i = \mathbf{a}^T \mathbf{x} \end{aligned}$$

Idee: Erweiterung auf komplexere Funktionen

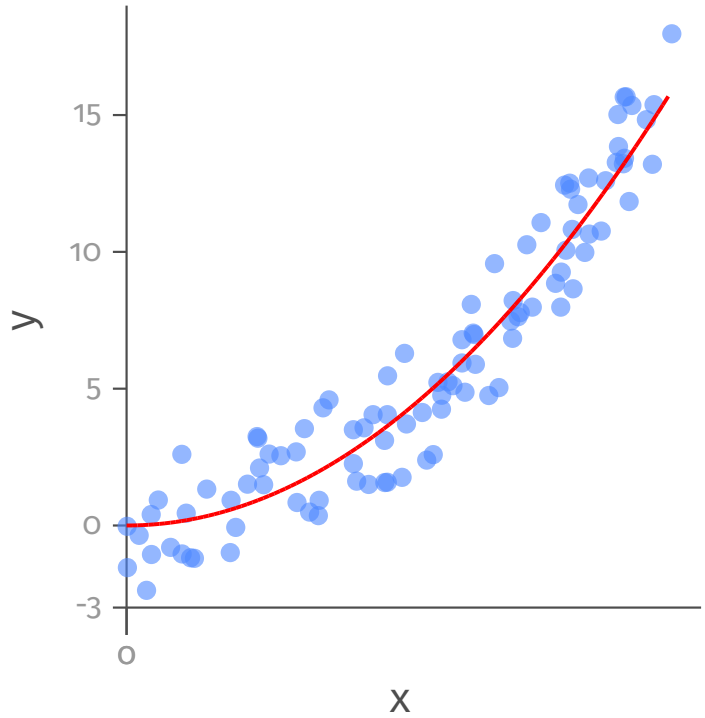
Polynom d-ten Grades, ein-dimensionale Daten $\mathbf{X}$:

$$\begin{aligned} p(x) &= a_0 + a_1 x + a_2 x^2 + \ldots + a_d x^d \\ &= \sum_{i=0}^{d} a_i x_i^i = \mathbf{a}^T \mathbf{x} \end{aligned}$$

Beispiel: Polynom q mit Parametern $\mathbf{a}^T = (17, 3, 2)$ ergibt

$$q(x) = 17 + 3x + 2x^2$$

Polynom 2-ten Grades



Generierte Daten und das Polynom $q(x) = 4x^2$

Polynome für Multiple Regression

Erweiterung auf mehrere Merkmale:

$$x_1, x_2 \Rightarrow x_1, x_2, x_1x_2, x_1^2, x_2^2$$

x_1	x_2
0.445	0.259
0.158	0.528
0.487	0.561
0.755	0.884
0.495	0.312

x_1	x_2	x_1x_2	x_1^2	x_2^2
0.445	0.259	0.115	0.198	0.067
0.158	0.528	0.083	0.025	0.278
0.487	0.561	0.274	0.237	0.315
0.755	0.884	0.668	0.571	0.781
0.495	0.312	0.154	0.245	0.097

Polynomielle Regression für mehrere Variablen/Merkmale

- Berechnung zusätzlicher Merkmale (Feature-Berechnung)
- Funktionsklasse linear in der Anzahl der Merkmale
- Optimierung wie bei normaler Regression

Polynomielle Regression für mehrere Variablen/Merkmale

- Berechnung zusätzlicher Merkmale (Feature-Berechnung)
- Funktionsklasse linear in der Anzahl der Merkmale
- Optimierung wie bei normaler Regression

Hinweis: So etwas ähnliches hatten wir schonmal – bei der SVM wurden die Daten nicht explizit transformiert, sondern lediglich das erforderliche Skalarprodukt in einem höherdimensionalen Raum berechnet (Kernel-Trick).

SciKit Learn enthält Funktion zur Merkmalsberechnung

```
from sklearn.preprocessing import PolynomialFeatures

X = ... # read DataFrame
pre = PolynomialFeatures(degree=2)

# berechne neue Merkmale auf X:
Xpoly = pre.fit_transform(X)
```

Wichtig:

- Preprocessor: `fit`, `transform`, `fit_transform`
- SciKit Learn benutzt numpy statt Pandas DataFrames

SciKit Learn enthält Funktion zur Merkmalsberechnung

```
from sklearn.preprocessing import PolynomialFeatures

X = ... # read DataFrame
pre = PolynomialFeatures(degree=2)

# berechne neue Merkmale auf X:
Xpoly = pre.fit_transform(X)
```

Wichtig:

- Preprocessor: `fit`, `transform`, `fit_transform`
- SciKit Learn benutzt numpy statt Pandas DataFrames

Wie funktioniert Preprocessing in SciKit-Learn?

SciKit Learn enthält Funktion zur Merkmalsberechnung

```
from sklearn.preprocessing import PolynomialFeatures

X = ... # read DataFrame
pre = PolynomialFeatures(degree=2)

# berechne neue Merkmale auf X:
Xpoly = pre.fit_transform(X)
```

Wichtig:

- Preprocessor: `fit`, `transform`, `fit_transform`
- SciKit Learn benutzt numpy statt Pandas DataFrames

Wie funktioniert Preprocessing in SciKit-Learn? Exkursion!

Exkursion: Pre-Processing mit SciKit Learn

Vorverarbeitung - Data Pre-Processing

Pre-Processing dient der Vorbereitung von Daten z.B. für die Modellierung:

- Normalisierung, Filtern von Daten
- Ersetzen fehlender Werte, Umrechnungen
- Berechnung zusätzlicher Merkmale
- Binning, Encoding nicht-numerischer Werte

Vorverarbeitung - Data Pre-Processing

Pre-Processing dient der Vorbereitung von Daten z.B. für die Modellierung:

- Normalisierung, Filtern von Daten
- Ersetzen fehlender Werte, Umrechnungen
- Berechnung zusätzlicher Merkmale
- Binning, Encoding nicht-numerischer Werte

SciKit Learn enthält eine Reihe von Preprocessing Klassen, z.B.

- MinMaxScaler, StandardScaler
- PolynomialFeatures, OrdinalEncoder
- KBinsDiscretizer

Preprocessing Module in SciKit Learn haben drei Funktionen:

- `fit(data)`
- `transform(data)`
- `fit_transform(data)`

Mit `fit(data)` werden die Parameter für ein Modul bestimmt.

Preprocessing Module in SciKit Learn haben drei Funktionen:

- `fit(data)`
- `transform(data)`
- `fit_transform(data)`

Mit `fit(data)` werden die Parameter für ein Modul bestimmt.

Mit `transform(data)` werden die Daten transformiert.

Preprocessing Module in SciKit Learn haben drei Funktionen:

- `fit(data)`
- `transform(data)`
- `fit_transform(data)`

Mit `fit(data)` werden die Parameter für ein Modul bestimmt.

Mit `transform(data)` werden die Daten transformiert.

`fit_transform(data)` macht beides in einem Schritt.

Beispiel: Min-Max Skalierung

Alle numerischen Attribute werden auf das Interval $[0, 1]$ skaliert.

```
# Parameter bestimmen (fit):
x_min = min(df['x'])
x_max = max(df['x'])

# Daten transformieren (transform):
df['x'] = (df['x'] - x_min) / (x_max - x_min)
```

Data Science 1: Übungsblatt 6, Aufgabe 2 und Foliensatz 6, Folien 26+27

Beispiel: Min-Max Skalierung

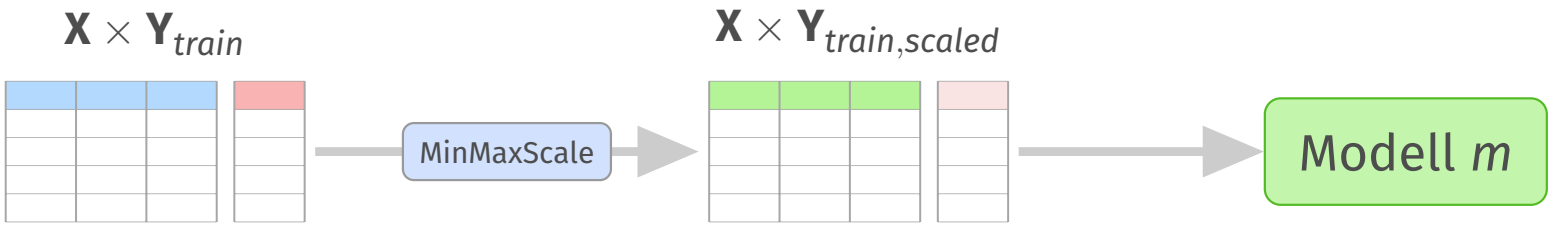
Alle numerischen Attribute werden auf das Interval $[0, 1]$ skaliert.

```
# Parameter bestimmen (fit):
x_min = min(df['x'])
x_max = max(df['x'])

# Daten transformieren (transform):
df['x'] = (df['x'] - x_min) / (x_max - x_min)
```

Data Science 1: Übungsblatt 6, Aufgabe 2 und Foliensatz 6, Folien 26+27

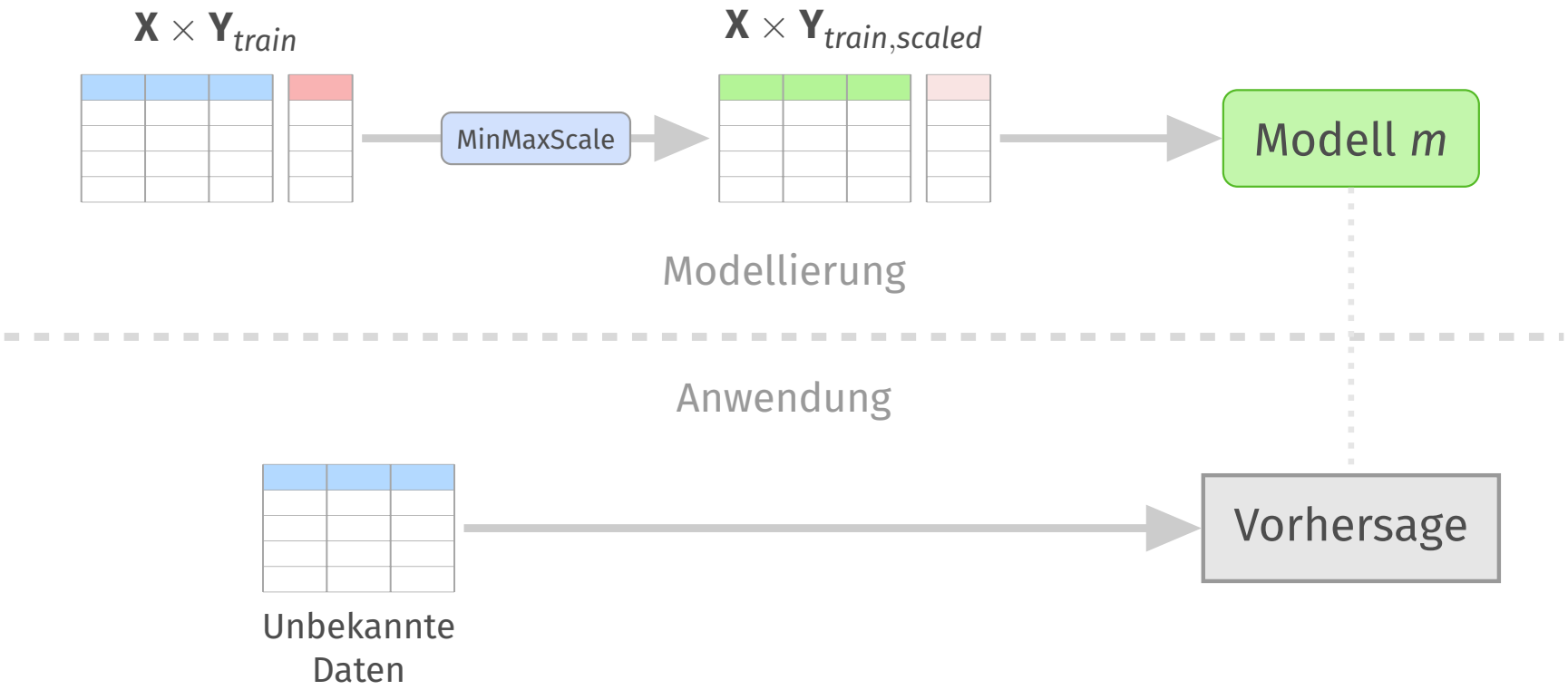
Die Parameter brauchen wir, wenn wir später Daten nochmal auf die gleiche Weise vorverarbeiten müssen!

Beispiel: Min-Max Skalierung

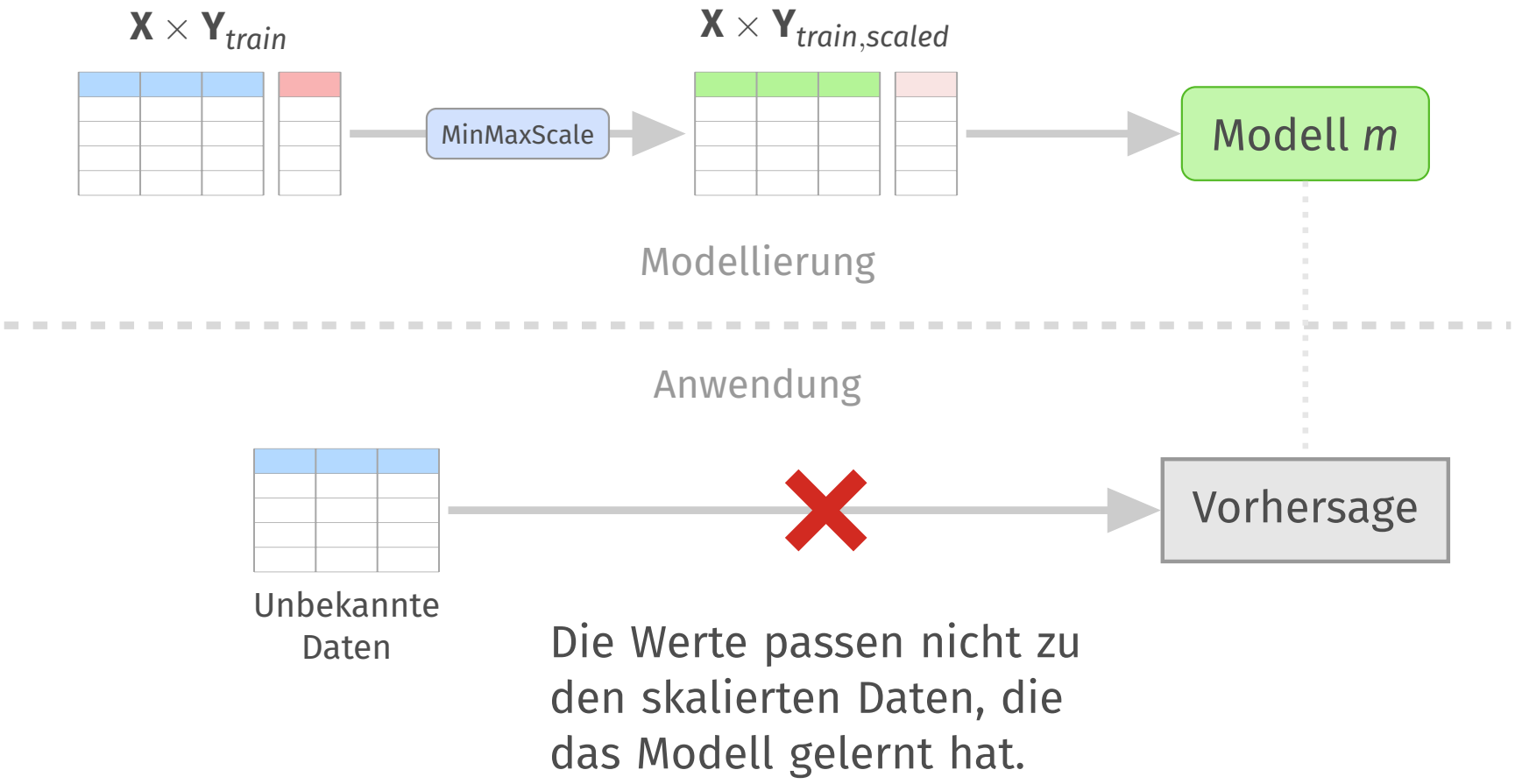


Modellierung

Beispiel: Min-Max Skalierung

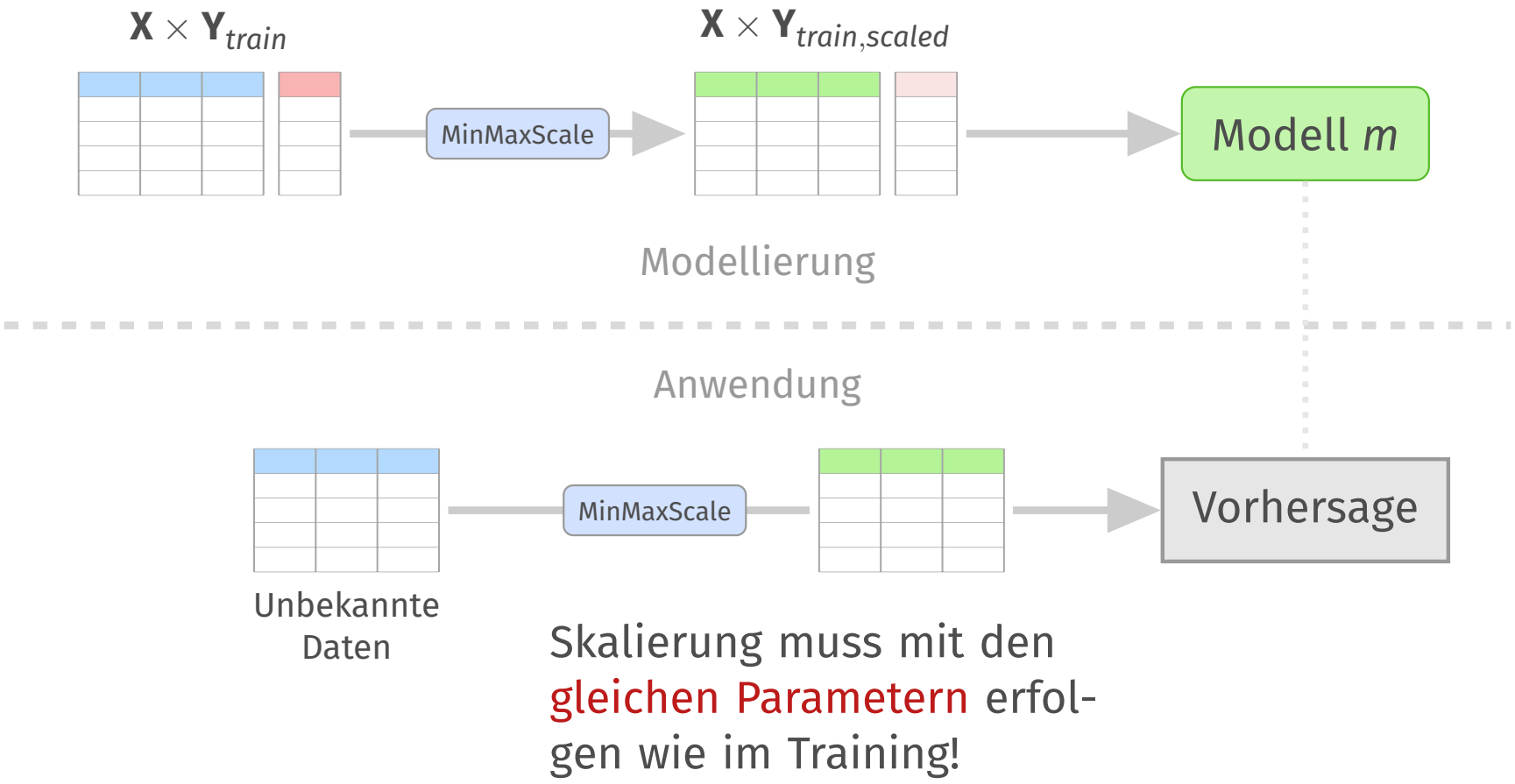


Beispiel: Min-Max Skalierung



Die Werte passen nicht zu den skalierten Daten, die das Modell gelernt hat.

Beispiel: Min-Max Skalierung



Skalierung muss mit den gleichen Parametern erfolgen wie im Training!

Min-Max Skalierung mit SciKit Learn

```
from sklearn.preprocessing import MinMaxScaler

df = ... # read DataFrame

scaler = MinMaxScaler()
scaler.fit(df)

# scaler hat jetzt die min/max Werte gelernt:
scaler.data_max_
scaler.data_min_
```

Problem: SciKit Learn arbeitet auf NumPy Arrays

NumPy Arrays sind effiziente Datenstruktur aus numpy:

- Liste von Zeilen, jede Zeile ist Liste von Werten
- nur numerische Werte erlaubt
- Keine Verwaltung von “Spaltennamen” wie bei DataFrame
- Aus NumPy Array lässt sich leicht ein DataFrame machen

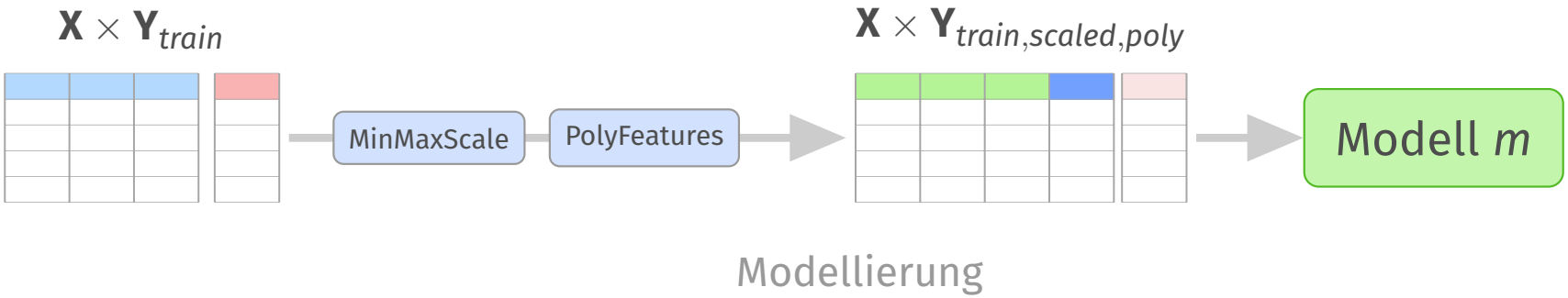
```
tf = scaler.transform(df)
# ergibt:
# array( [[0.347, 0.613], [0.292, 0.586],...])
```

Min-Max Skalierung mit DataFrame Output

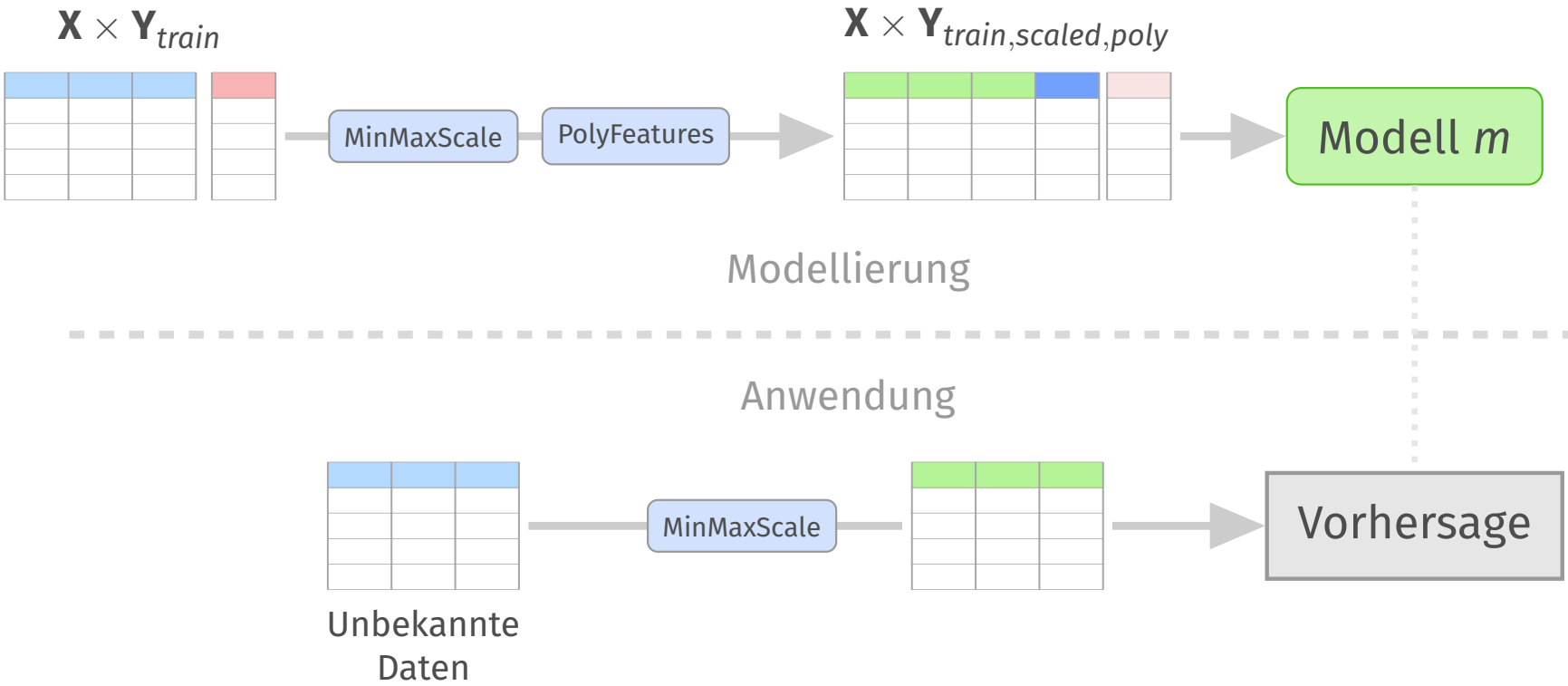
Im Folgenden sei `df` ein DataFrame mit numerischen Spalten:

```
# MinMax Scaler anpassen/trainieren
scaler = MinMaxScaler()
scaler.fit(df)

# Daten transformieren
norm_data = scaler.transform(df)

# DataFrame aus numpy-Array erzeugen - mit
# den Spaltennamen von df
norm_df = pd.DataFrame(norm_data, columns=df.columns)
```

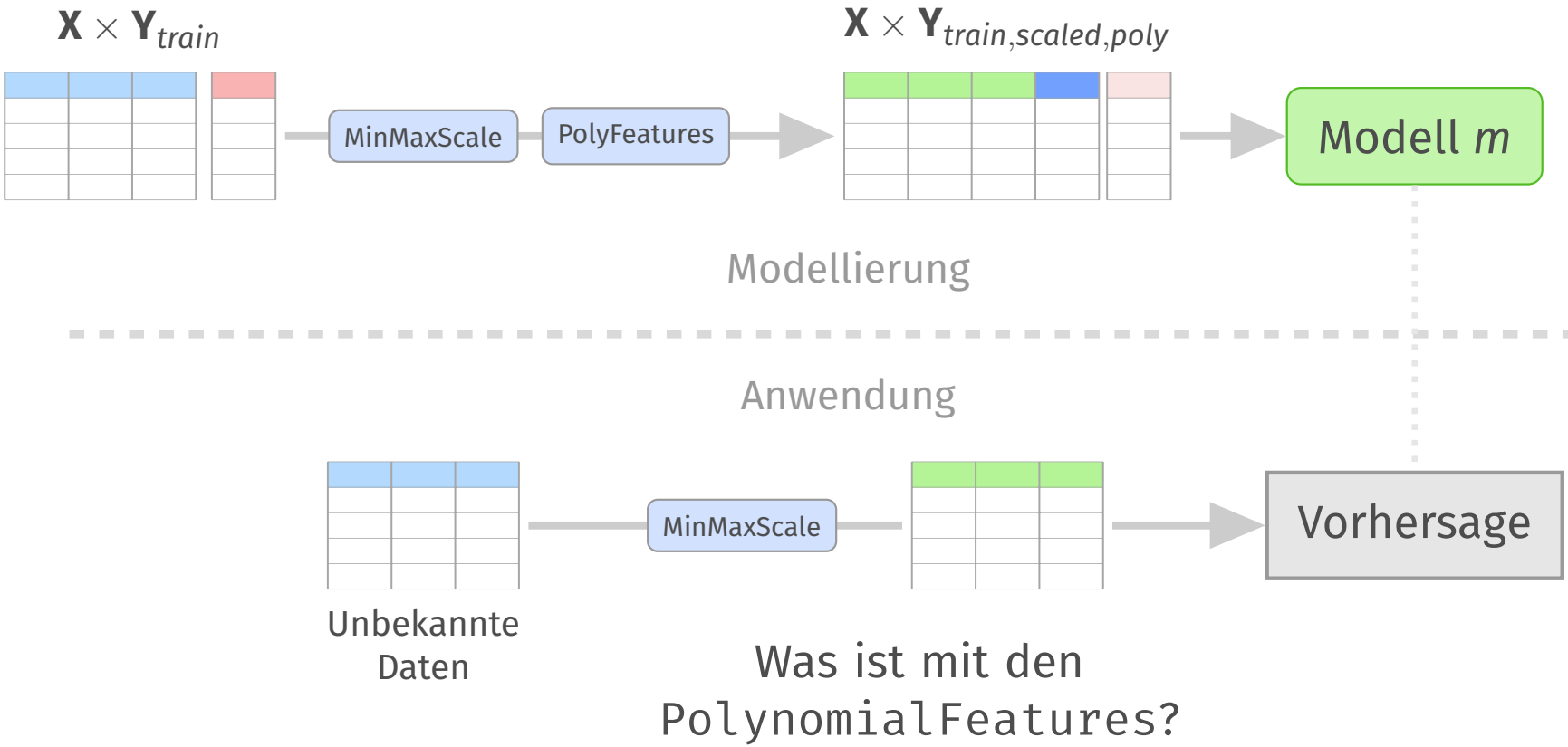
Beispiel: Min-Max Skalierung, Polynomial Regression



Beispiel: Min-Max Skalierung, Polynomial Regression



Beispiel: Min-Max Skalierung, Polynomial Regression



Pipeline: Mehrere Pre-Processing Schritte zusammengefasst

```
from sklearn.preprocessing ...
from sklearn.pipeline import Pipeline
from sklearn.pipeline import make_pipeline

# Pipeline mit 2 Schritten:
scaler = MinMaxScaler()
poly = PolynomialFeatures(degree=4)

pipeline = Pipeline([('Scaling', scaler), ('Features'
                                 ,poly)])

# kurz:
pipeline = make_pipeline(scaler, poly)
```

Pipeline ist selbst wieder Preprocessor

Pipeline definieren und Schritte anpassen/anwenden:

```
# alle Schritte anpassen
pipeline.fit(data)

# alle Schritte anwenden
transformed = pipeline.transform(data)
```

◀ Probieren Sie es im Notebook aus!

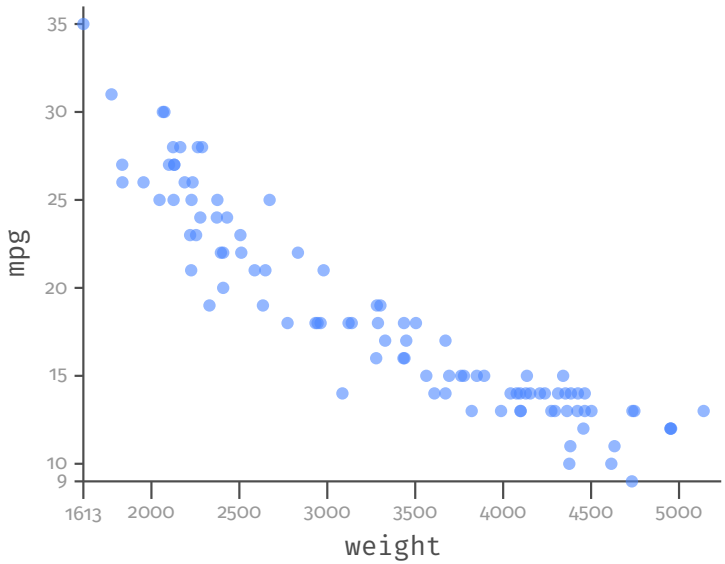
Notebook: Vorlesung/V2-2-MinMaxScaling_sklearn.ipynb

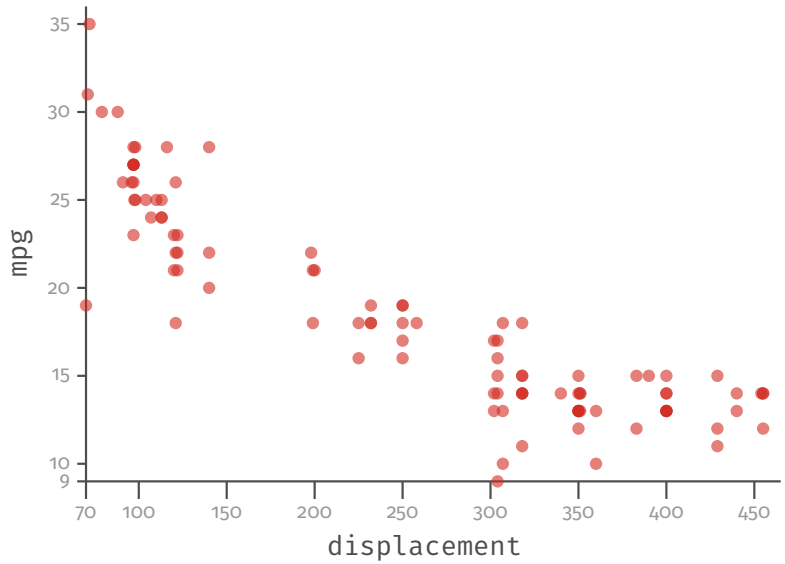
Ende der Exkursion

Zurück zur Polynomiellen Regression

Beispiel: Auto MPG Datensatz

Kraftstoffverbrauch von Autos (in *miles per gallon*)





Lineare Regression auf $x_{weight}, x_{displacement}$ führt zu

$$Err_{mae} \approx 3.28, \quad R^2\text{-Score} \approx 0.572$$

Beispiel: Auto MPG Datensatz

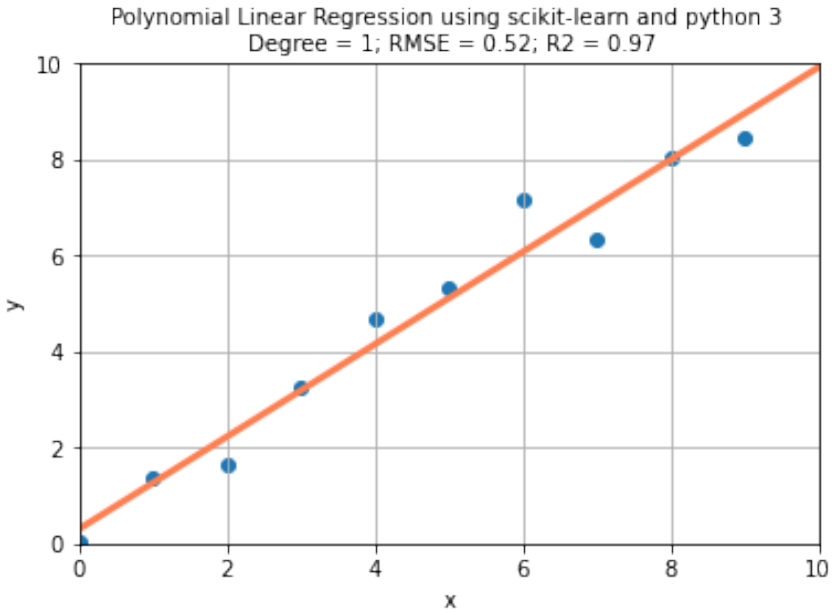
```
df = pd.read_csv(...)

X = df[['displacement', 'weight']]
Y = df[['mpg']]

poly = PolynomialFeatures(degree=2)
Xtrans = poly.fit_transform(X)

print(poly.get_feature_names())
#  ['1', 'x0', 'x1', 'x0^2', 'x0 x1', 'x1^2']

Xtrans = pd.DataFrame(X, poly.get_feature_names())

model = LinearRegression()
model.fit(Xtrans, Y)
```

Hinweis:

- Konvertierung in DataFrame `Xtrans` nicht zwingend notwendig
- SciKit-Learn Modelle arbeiten auch mit `numpy` Arrays
- DataFrame wird hier verwendet um die bekannten Methoden auf den transformierten Daten weiterhin benutzen zu können.

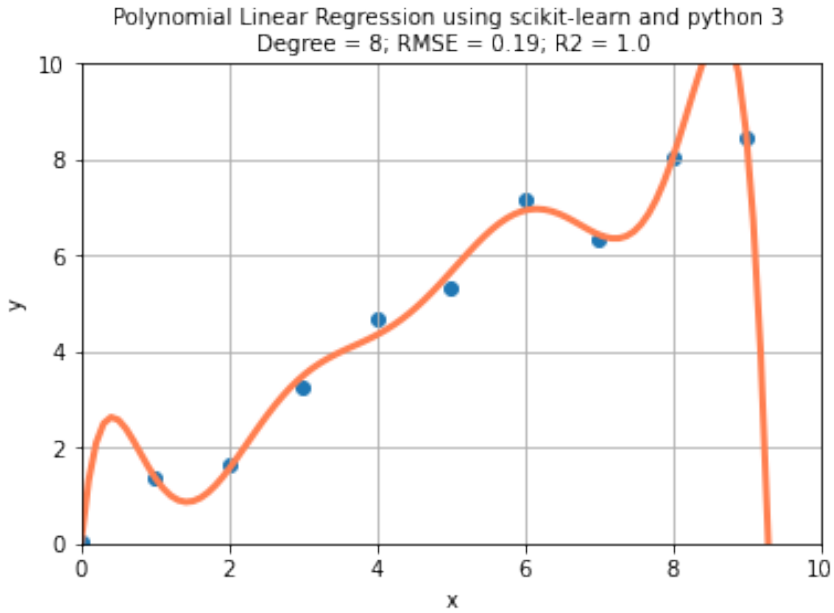
Overfitting + Risiko-Minimierung

Optimierung auf geringen Trainingsfehler



$Err_{rmse} = 0.52$

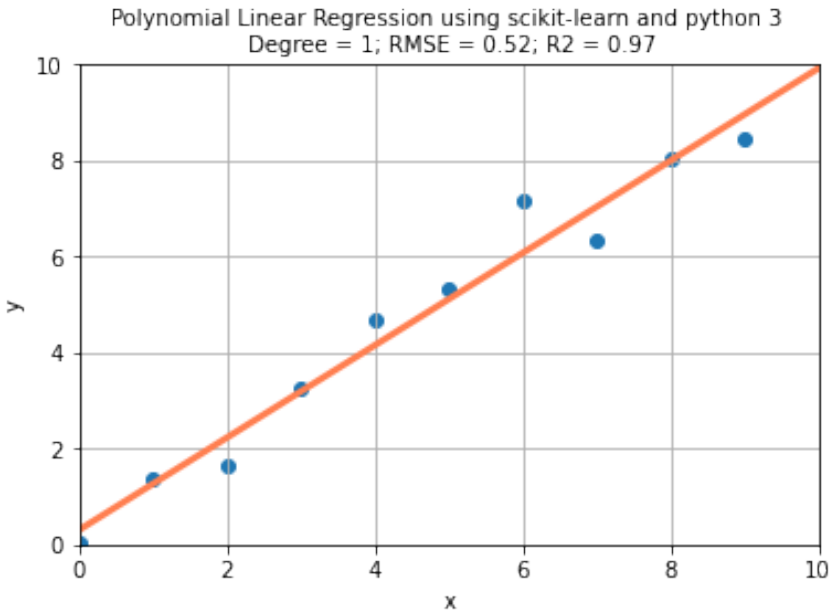
Niedrige Komplexität



$Err_{rmse} = 0.19$

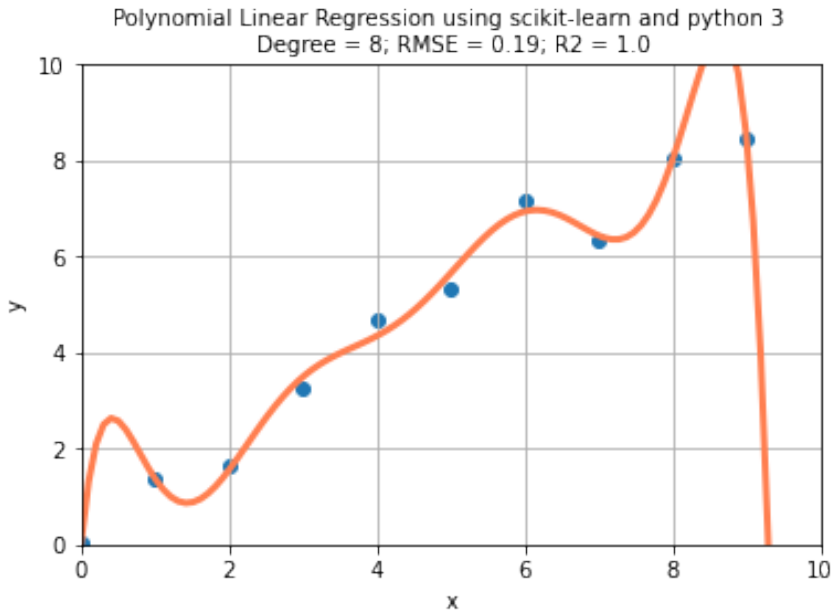
Hohe Komplexität

Optimierung auf geringen Trainingsfehler



$Err_{rmse} = 0.52$

Niedrige Komplexität



$Err_{rmse} = 0.19$

Hohe Komplexität

Frage: Welches ist das bessere Modell?

Expected Prediction Error

Gehen wir davon aus, dass es die wahre Funktion f gibt, die unsere Daten erzeugt:

$$Y = f(X) + \epsilon$$

ε beschreibt Rauschen in den Daten.

Der erwartete Fehler unseres Modells $\hat{f}$ ist nun

$$Err_{exp}(x) = E\left[(Y - \hat{f}(x))^2\right]$$

Fehler besteht aus Verzerrung und Varianz

Der erwartete Fehler läßt sich aufteilen in Verzerrung (Bias) und Varianz:

$$
\begin{aligned}
Err_{exp}(x) &= E\left[(Y - \hat{f}(x))^2\right] \\
&= \left(E\left[\hat{f}(x)\right] - f(x)\right)^2 + E\left[\left(\hat{f}(x) - E[\hat{f}(x)]\right)^2\right] + \sigma_\epsilon^2 \\
&= \text{Bias}(\hat{f})^2 + \text{Var}(\hat{f}) + \text{unvermeidbarer Fehler}
\end{aligned}
$$

Bias und Varianz

Bias von $\hat{f}$

- beschreibt den durchschnittlichen Abstand zur Wahrheit
- Wie gut passt $\hat{f}$ im Mittel zum wahren Wert?
- Kann unser Modell die Wahrheit überhaupt erreichen?

Varianz von $\hat{f}$

- Beschreibt die Streuung unserers Modells $\hat{f}$
- Wie konstant sagt $\hat{f}$ das richtige y vorher?

Bias und Varianz

Bias von $\hat{f}$

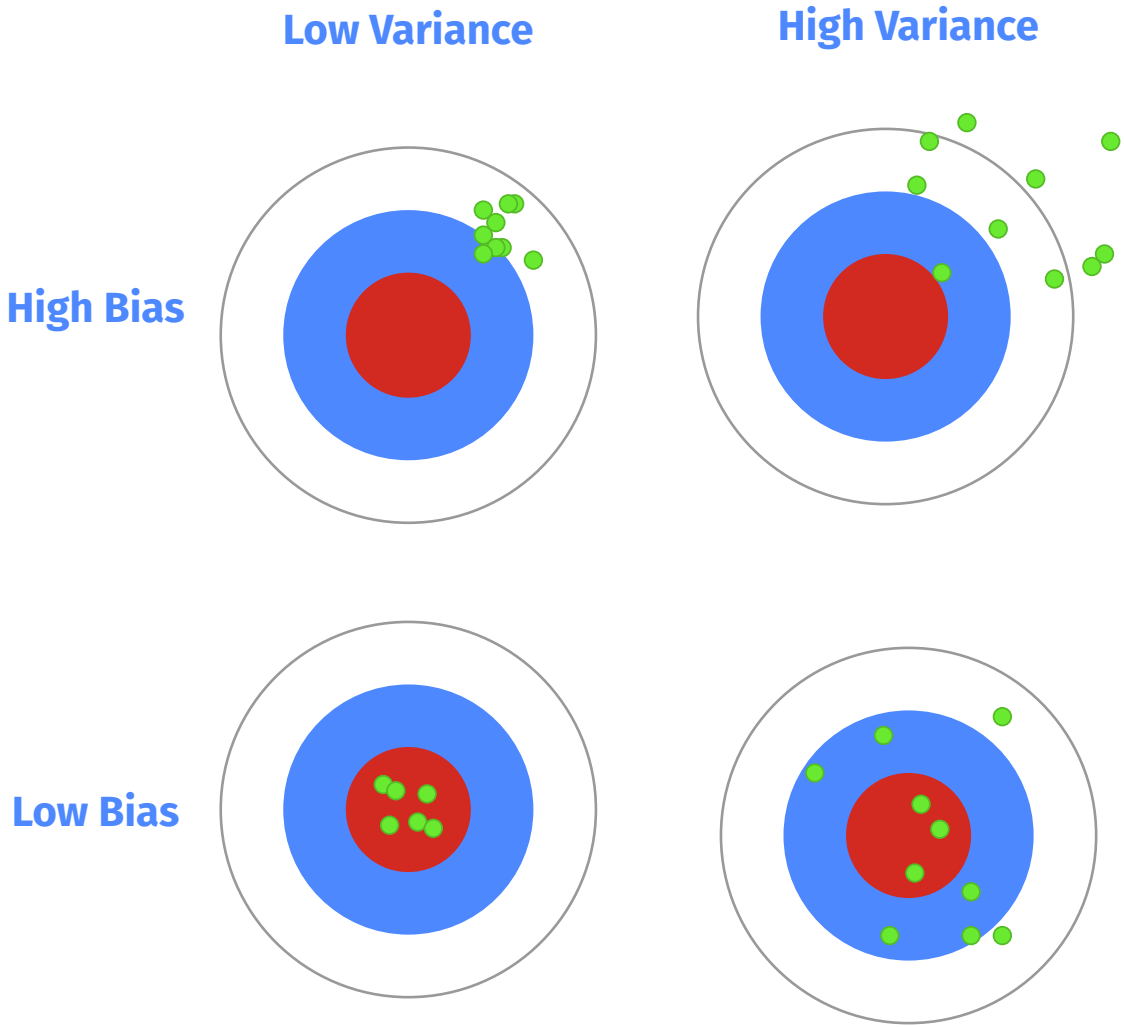
- beschreibt den durchschnittlichen Abstand zur Wahrheit
- Wie gut passt $\hat{f}$ im Mittel zum wahren Wert?
- Kann unser Modell die Wahrheit überhaupt erreichen?

Varianz von $\hat{f}$

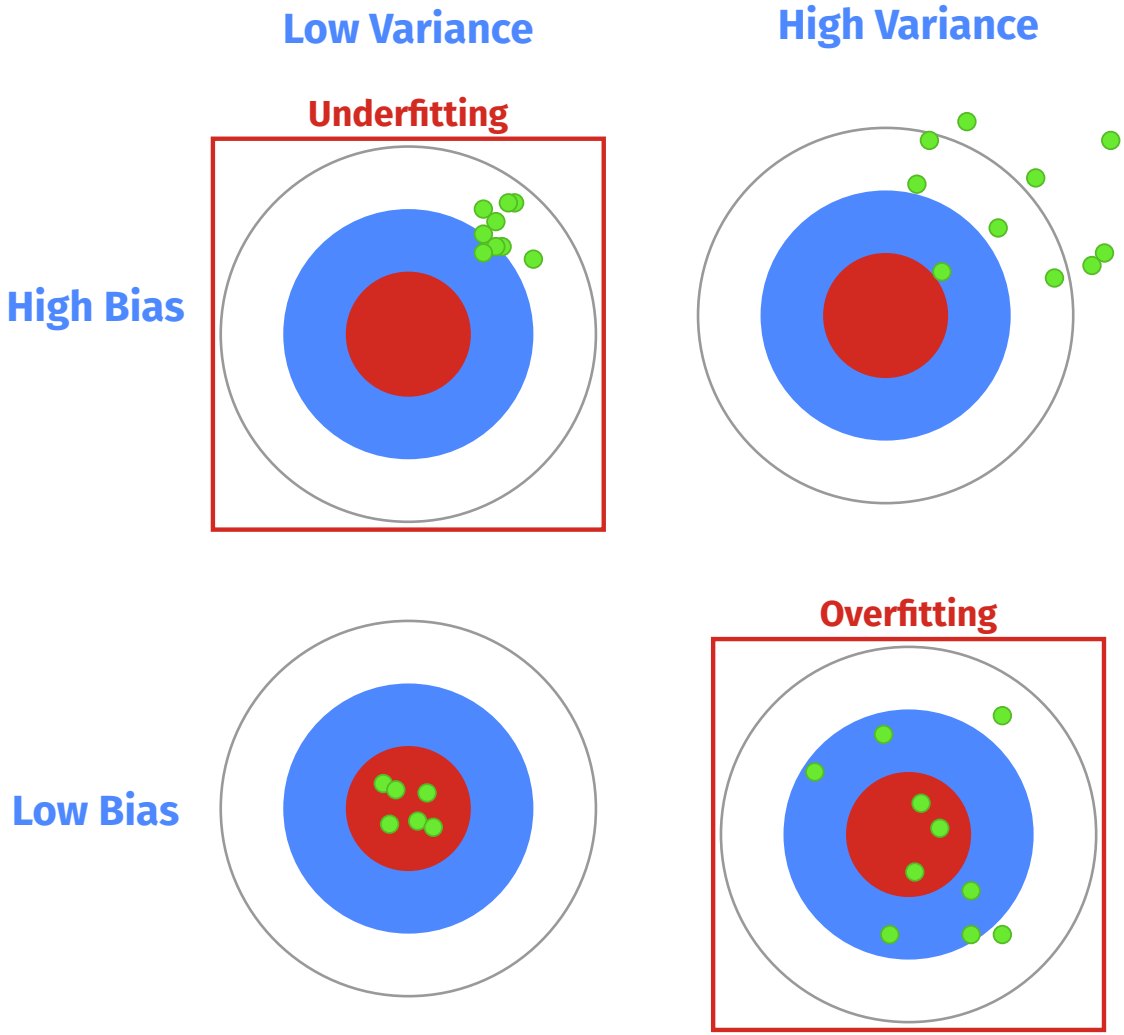
- Beschreibt die Streuung unserers Modells $\hat{f}$
- Wie konstant sagt $\hat{f}$ das richtige y vorher?

Wir suchen eine gute Balance zwischen Bias und Varianz.

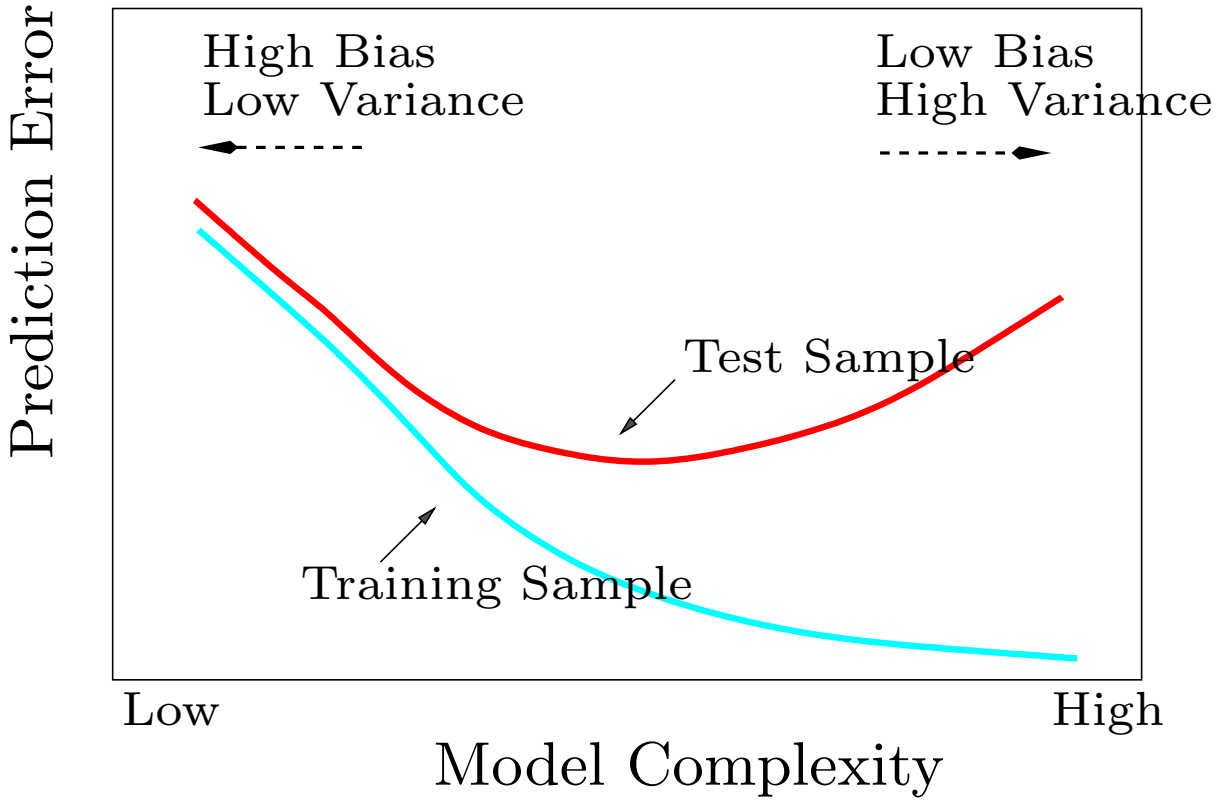
Bias / Variance Decomposition



Bias / Variance Decomposition



Bias / Variance Decomposition



Wie kontrollieren wir die Balance?

- Hohe Modell-Komplexität führt zu niedrigem Bias, hoher Varianz
- Niedrige Modell-Komplexität zu hohem Bias, niedriger Varianz

Modell-Komplexität

Modell-Komplexität ergibt sich z.B. aus

- dem Grad der Polynome (Regression)
- der Tiefe von Bäumen

Wie kontrollieren wir die Modell-Komplexität?

Maß für Modell-Komplexität in Fehler-Funktion aufnehmen:

$$\arg\min_{\mathbf{w}} \underbrace{\sum_{\mathbf{x},\mathbf{y}} (y - f_{\mathbf{w}}(\mathbf{x}))^2}_{\text{Fehler}} + \lambda \underbrace{C(f_{\mathbf{w}})}_{\text{Komplexität}}$$

Parameter λ steuert Schwerpunkt (Fehler vs. Modellkomplexität)

Idee: Wenn w viele Nullen hat $\rightarrow$ geringe Komplexität

Wir können die Modellkomplexität an **w** "ablesen":

$$\mathbf{w}' = \begin{pmatrix} 4 \\ 0 \\ 1 \\ 8 \\ 31 \\ 19 \end{pmatrix} \qquad \mathbf{w}'' = \begin{pmatrix} 0 \\ 0 \\ 4 \\ 0 \\ 0 \\ 19 \end{pmatrix}$$

$$4 + x^2 + 8x^3 + 31x^4 + 19x^5 \qquad 4x^2 + 19x^5$$

Idee: Wenn w viele Nullen hat $\rightarrow$ geringe Komplexität

Wir können die Modellkomplexität an **w** "ablesen":

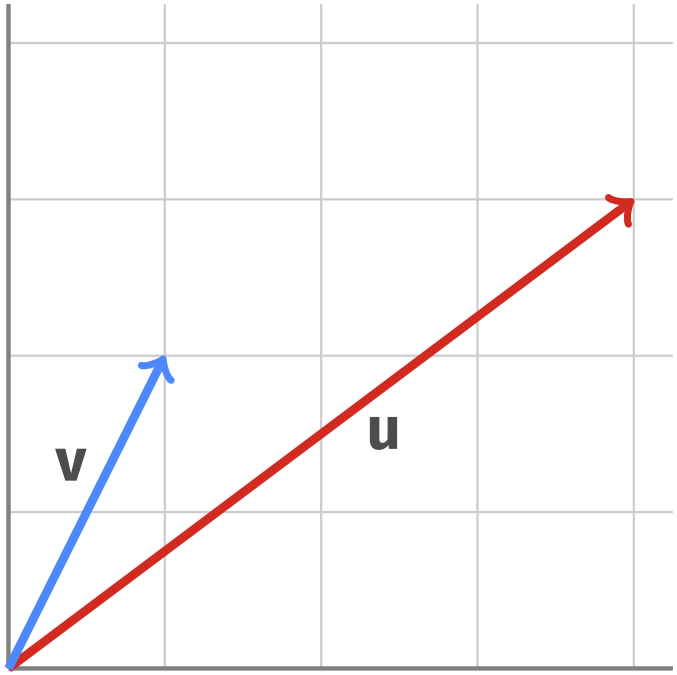
$$\mathbf{w}' = \begin{pmatrix} 4 \\ 0 \\ 1 \\ 8 \\ 31 \\ 19 \end{pmatrix} \qquad \mathbf{w}'' = \begin{pmatrix} 0 \\ 0 \\ 4 \\ 0 \\ 0 \\ 19 \end{pmatrix}$$

$$4 + x^2 + 8x^3 + 31x^4 + 19x^5 \qquad 4x^2 + 19x^5$$

Statt Nullen sind auch viele kleine Werte gut, dann beeinflussen viele Komponenten die Funktion nicht so stark.

Frage: Wann hat ein Vektor w viele kleine Werte?

Norm eines Vektors **v** entspricht der Länge von **v**



$||v|| = 2.235$

$||u|| = 5.0$

Data Science 1: Foliensatz 7, Folie 9 und Übungsblatt 7

Regularisierung der Modellkomplexität

Modellkomplexität von $f_{\mathbf{w}}$ über Parametervektor $\mathbf{w}$ messen:

$$||\mathbf{w}||_2 = \sqrt{\sum_{i=1}^{p} w_i^2}$$

Regularisierung der Modellkomplexität

Modellkomplexität von $f_{\mathbf{w}}$ über Parametervektor $\mathbf{w}$ messen:

$$||\mathbf{w}||_2 = \sqrt{\sum_{i=1}^{p} w_i^2}$$

Beispiel: Ridge Regression

Optimierungsfunktion der Ridge Regression:

$$\arg\min_{\mathbf{w}} \left\{ \sum_{i=1}^{n} (y_i - \mathbf{w}^T \mathbf{x}_i)^2 + \lambda \sum_{i=1}^{p} w_i^2 \right\}$$

$||\mathbf{w}||_2^2$ als Komplexitätsmaß (quadrierte L_2-Norm)

Weiteres Beispiel: Lasso Regression

L_1-Norm (=Betrag) als Modellkomplexität

$$L_1(\mathbf{w}) = ||\mathbf{w}||_1 = \sum_{i=1}^{p} |w_i|$$

führt zu folgender Optimierungsfunktion:

$$\arg\min_{\mathbf{w}} \left\{ \sum_{i=1}^{n} (y_i - \mathbf{w}^T \mathbf{x}_i)^2 + \lambda \sum_{i=1}^{p} |w_i| \right\}$$