Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

# 2024 FS CAS PML - Supervised Learning
# 3 Regression
# 3.3 ML Methoden

Werner Dähler 2024

# 3 Regression - AGENDA

31. Einleitung
32. Regression klassisch (OLS)
33. Regression mit ML
    331. KNeighborsRegressor
    332. DecisionTreeRegressor
    333. RandomForestRegressor
    334. SVR
    335. MLPRegressor
34. Vergleiche über alle Modelle

# 3.3 Regression - ML Methoden

- die meisten der unter Klassifikation vorgestellten Methoden (Klassen) haben "Geschwister", welche auch für Regressions-Modelle eingesetzt werden können
- eine Übersicht:

| Modul | Klassifikation | Regression |
|---|---|---|
| sklearn.linear_model | (kein analoger Klassifikator) | LinearRegression |
| sklearn.neighbors | KNeighborsClassifier | KNeighborsRegressor |
| sklearn.tree | DecisionTreeClassifier | DecisionTreeRegressor |
| sklearn.ensemble | RandomForestClassifier | RandomForestRegressor |
| sklearn.ensemble | AdaBoostClassifier | AdaBoostRegressor |
| sklearn.ensemble | GradientBoostingClassifier | GradientBoostingRegressor |
| sklearn.ensemble | HistGradientBoostingClassifier | HistGradientBoostingRegressor |
| sklearn.svm | SVC | SVR |
| sklearn.neural_network | MLPClassifier | MLPRegressor |
| catboost | CatBoostClassifier | CatBoostRegressor |
| lightgbm.sklearn | LGBMClassifier | LGBMRegressor |

- sie heissen nicht nur (fast) gleich, auch die Tuning-Parameter sind mehrheitlich dieselben

# 3.3 Regression - ML Methoden

## Vorbereitungen

Voraussetzungen: die üblichen Libraries sind importiert und die notwendigen Daten geladen (vgl. 3.1.4)

1. ein synthetisches Testset
   - für die folgenden Methoden wird im Theorieteil ein synthetisches Testset verwendet werden , um die Funktionsweise beim Trainieren mit den Demo-Daten zu visualisieren
   - der untenstehende Code erstellt einen Array über den gesamten Wertebereich von X_demo mit 100 gleich grossen Schritten

```
X_synth = np.linspace(X_demo.min(), X_demo.max(), 100).reshape(-1,1)
```

   - das Modell wird jeweils mit X_demo trainiert und dann eine Prediction mit X_synth erstellt
   - diese wird danach für eine Visualisierung verwendet (jeweils orange "Kurve")
   - eine Validierung mit einer Performance Metrik unterbleibt hier, da sie hier nicht relevant ist (kein Train - Test - Split), diese wird im Praxis-Teil auf dem Melbourne Housing Dataset durchgeführt und diskutiert
   - vgl. Funktion `show_pred_on_synth()` in Modul `bfh_cas_pml.py`

# 3.3 Regression - ML Methoden

## Vorbereitungen

2. Vorbereitung für Praxis:
   - da im Folgenden für jede Regressionsklasse derselbe Ablauf zur Anwendung kommt, wird hier eine Funktion definiert, die das parametrisierte Modell sowie die Daten entgegennimmt, und dann folgendes ausführt:
     - train
     - predict
     - berechnen und ausgeben der Metriken
     - erstellen Scatterplot von y_pred vs. y_test (optional, default=True)
   - die Funktion gibt das trainierte Modell zurück, um nach deren Aufruf dort bei Bedarf noch einige Interna (Modellattribute) untersuchen zu können, ausserdem wird r2_score in der Konsole ausgegeben

# 3.3 Regression - ML Methoden

## Vorbereitungen

- die Funktionsdefinition (ein Ausschnitt davon, vgl. Modul bfh_cas_pml):

```
def test_regression_model(
    model, X_train, y_train, X_test, y_test, show_plot=True):

    from sklearn.metrics import r2_score
    from sklearn.metrics import mean_squared_error

    model = model model.fit(X_train, y_train)
    y_pred = model.predict(X_test)

    print('R2 = %0.4f' %(r2_score(y_test, y_pred)))
    if show_plot == True:
        :

    return (model)
```

# 3.3 Regression - ML Methoden

**Vorbereitungen**

- vor Aufruf der Funktion muss sichergestellt sein, dass die Regressionsklasse bereits importiert ist
- als Parameter werden das parametrisierte Objekt sowie alle notwendigen Daten übergeben,
  dazu kann mit dem optionalen Parameter `show_plot` gesteuert werden, ob der Scatterplot ausgegeben werden soll
- ausserdem wird das trainierte Modell (Rückgabe der Funktion) als Objekt zurückgegeben, um dies bedarfsweise nach Aufruf der Funktion weiter untersuchen zu können
- ein beispielhafter Aufruf mit LinearRegression()

```
from bfh_cas_pml import test_regression_model
from sklearn.linear_model import LinearRegression
this_model = test_regression_model(
    LinearRegression(), X_train, y_train, X_test, y_test, show_plot=False)
```

```
R2 = 0.5601
```

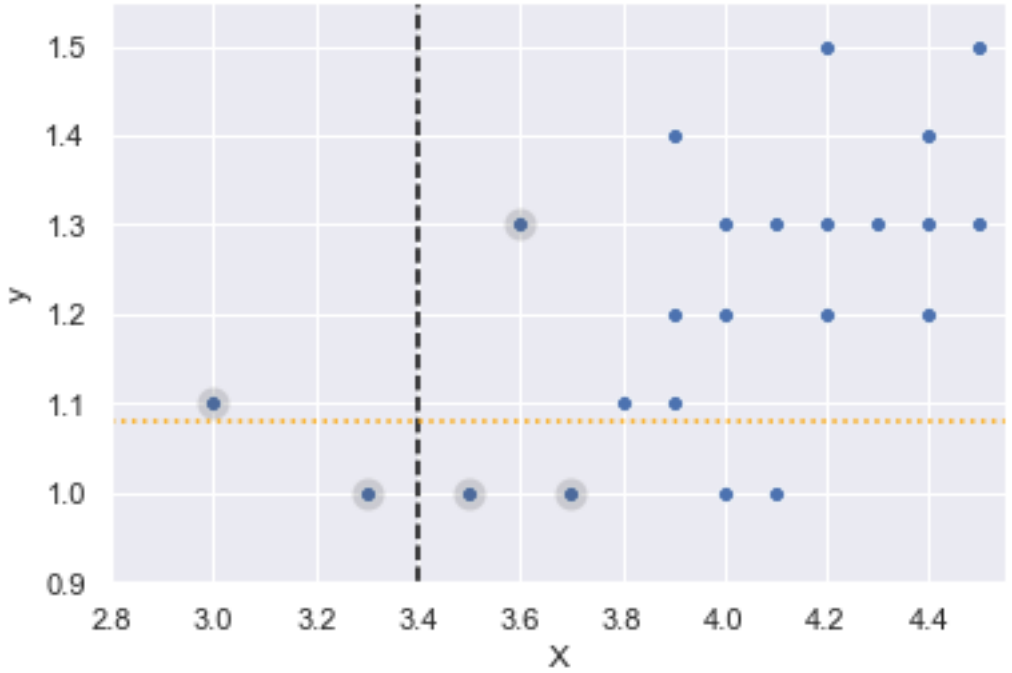
# 3.3.1 Regression - ML Methoden - KNeighborsRegressor

## 3.3.1.1 Theorie

- analog `KNeighborsClassifier` werden die k ähnlichsten Beobachtungen im Trainingsset in Bezug auf Feature Werte berücksichtigt
- ein Zahlenbeispiel zur Illustration:
  - ein Modell basierend auf demo_data_regr.csv soll auf eine neue Test-Beobachtung mit einem Wert X = 3.4 angewendet werden (vertikale Linie)
  - die 5 nächsten Nachbarn dieser Beobachtungen weisen für y folgende Werte auf: 1.1, 1.0, 1.0, 1.3, 1.0 (speziell ausgezeichnet)
  - als Prediction wird der Mittelwert von y dieser 5 Beobachtungen zurückgegeben:

$$\frac{1}{5} \cdot (1.1 + 1.0 + 1.0 + 1.3 + 1.0) = 1.08$$

(horizontale Linie)



# 3.3.1 Regression - ML Methoden - KNeighborsRegressor

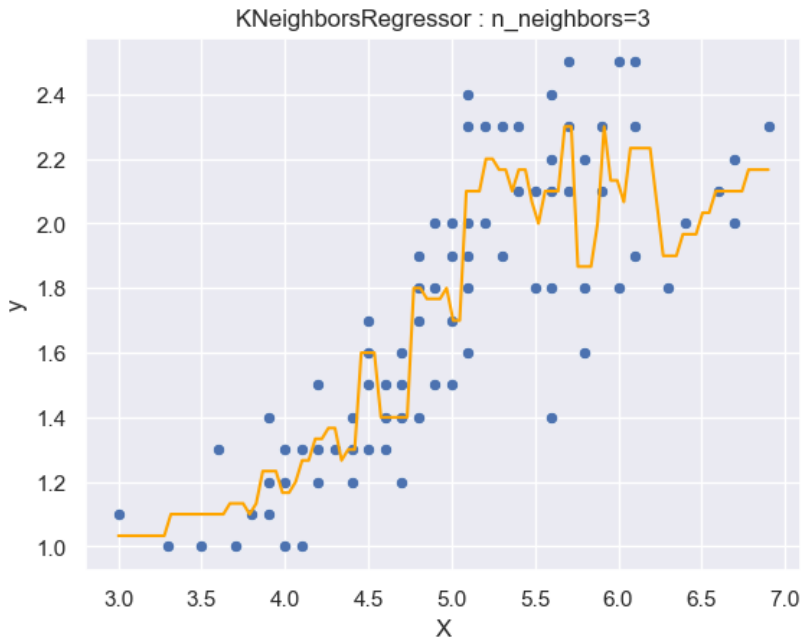
## 3.3.1.1 Theorie

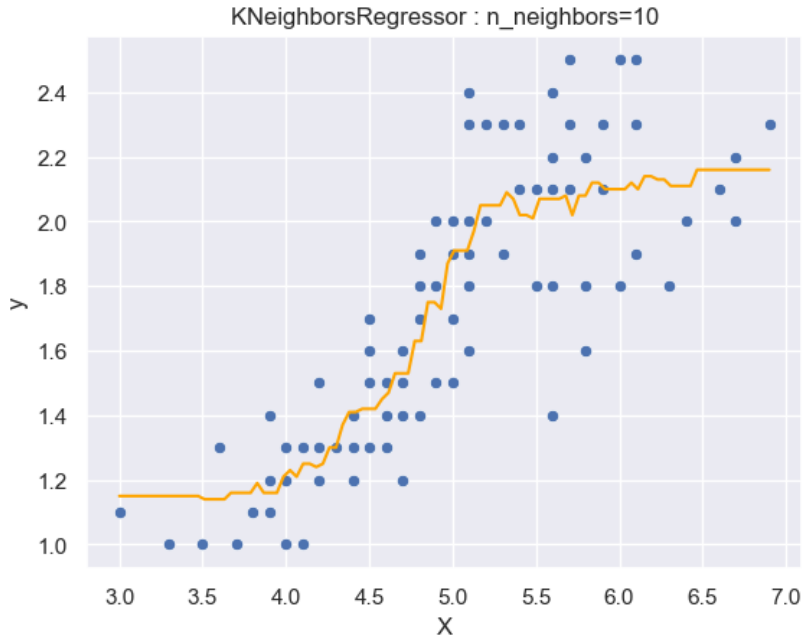
- Analogie zu `KNeighborsClassifier`
  - `KNeighborsClassifier`: Prediction ist der **Modalwert** des Labels der k nächsten Beobachtungen
  - `KNeighborsRegressor`: Prediction ist der **Mittelwert** des Labels der k nächsten Beobachtungen

# 3.3.1 Regression - ML Methoden - KNeighborsRegressor

## 3.3.1.1 Theorie

- die untenstehende Darstellung zeigt den Vergleich der Predictions (orange) bei 3 resp. 10 nächsten Nachbarn
  - blau: Demo Daten
  - orange: Prediction des trainierten Modells auf das synthetische Testset mit unterschiedlichen Werten für n_neighbors




- provisorisches Fazit: kleine Werte von n_neighbors führen tendenziell zu Overfitting

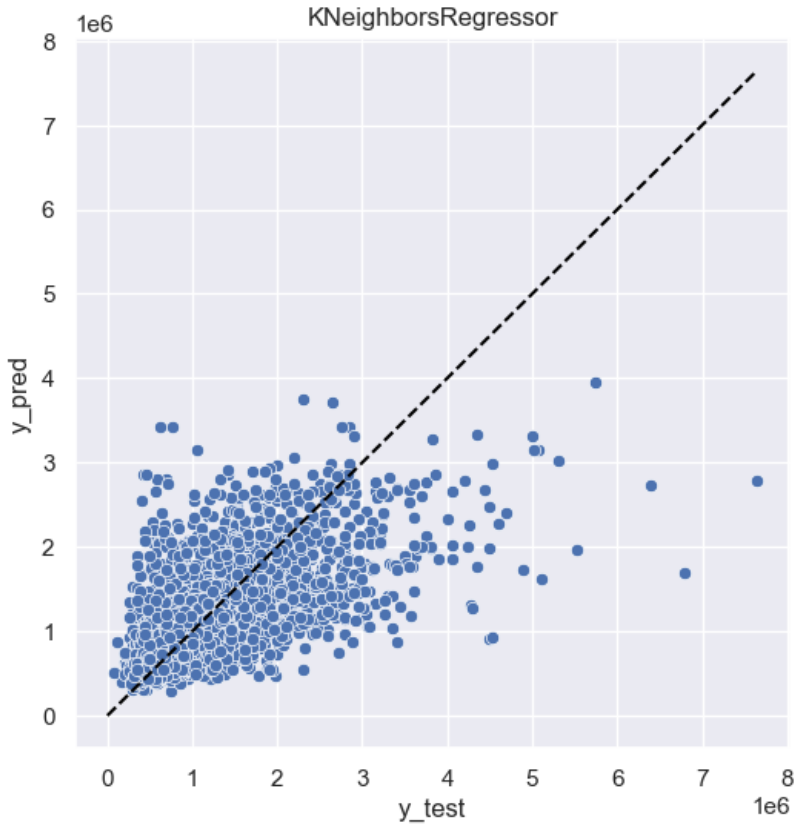
# 3.3.1 Regression - ML Methoden - KNeighborsRegressor

## 3.3.1.2 Praxis

```
from sklearn.neighbors import KNeighborsRegressor
this_model = test_regression_model(
    KNeighborsRegressor(), X_train, y_train, X_test, y_test)
```

```
R2 = 0.4493
```

- wichtigste Parameter:
  - `n_neighbors=5`: selbstsprechend (vgl. Klassifikator)
  - `metric='minkowski'`
  - `p=2`: zusammen mit metrics → Euklidisch Distanz
  - weitere: scikit-learn Dokumentation
- Fazit
  - ist zwar deutlich schlechter als OLS (0.5601), aber negative Vorhersagen und Nichtlinearität sind bereinigt



# 3.3.2 Regression - ML Methoden - DecisionTreeRegressor

## 3.3.2.1 Theorie

- Aufbau des Regressionsbaums analog Klassifikationsbaum
  - für jedes Feature wird die optimale Splitposition gesucht
  - das Feature mit dem besten Splitverhalten wird ausgewählt und der Split durchgeführt
- Split Kriterium: MSE (Mean squared error regression loss) anstelle von gini oder entropy

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

dabei bedeuten

$n$: Anzahl Beobachtungen der Testdaten

$y_i$: wahrer Targetwert der i-ten Beobachtung der Testdaten

$\hat{y}_i$: Mittelwert aller Targetwerte der Testdaten

entspricht somit der Varianz (Quadrat der Standardabweichung, vgl. Kap. 1.2.3.1)

(Ausblick: MSE werden wir auch leicht modifiziert als Performance-Metrik wieder sehen, vgl. Kap. 4.4.2.2)

# 3.3.2 Regression - ML Methoden - DecisionTreeRegressor

## 3.3.2.1 Theorie

- Abbruchkriterium: standardmässig wird der Baum komplett aufgebaut, da aber verschiedene Beobachtungen denselben X Wert aufweisen können, kann es in den Endblättern mehrere Beobachtungen mit unterschiedlichem Predict-Werten haben - dort wird dann der Mittelwert als Prediction zurückgegeben (vgl. `KNeighborsRegressor`)
- die Parameter sind mehrheitlich dieselben wie beim analogen Klassifikator

  - Berechnung analog Klassifikation
    - für jeden Split wird die Impurity (hier also MSE) vor und nach dem Split berechnet
    - vor dem Split MSE gegenüber dem Mittelwert aller Beobachtungen
    - nach dem Split der gewichtete Mittelwert der beiden MSE Werte der Kindknoten
    - die Differenz dann wiederum gewichtet am Anteil des untersuchten Knotens in Bezug auf Root-Knoten (vgl. DecisionTreeClassifier)

# 3.3.2 Regression - ML Methoden - DecisionTreeRegressor

## 3.3.2.1 Theorie

- Prediction vor einem ersten Split: Mittelwert über alle y (schwarze horizontale Linie)
- ein (hypothetischer) Split an der Stelle X=4.95 (vertikale Linie) teilt die Instanzen in zwei Teilmengen (orange und blau)
- für jede dieser Teilmengen wird der Mittelwert von y bestimmt (horizontale Linien) und für die jeweilige Teilmenge als Prediction betrachtet
- darauf basierend wird für beide Teilmengen MSE gegenüber dem jeweiligen Mittelwert berechnet
- zur Bestimmung von MSE nach dem Split werden beide MSE gewichtet gemittelt

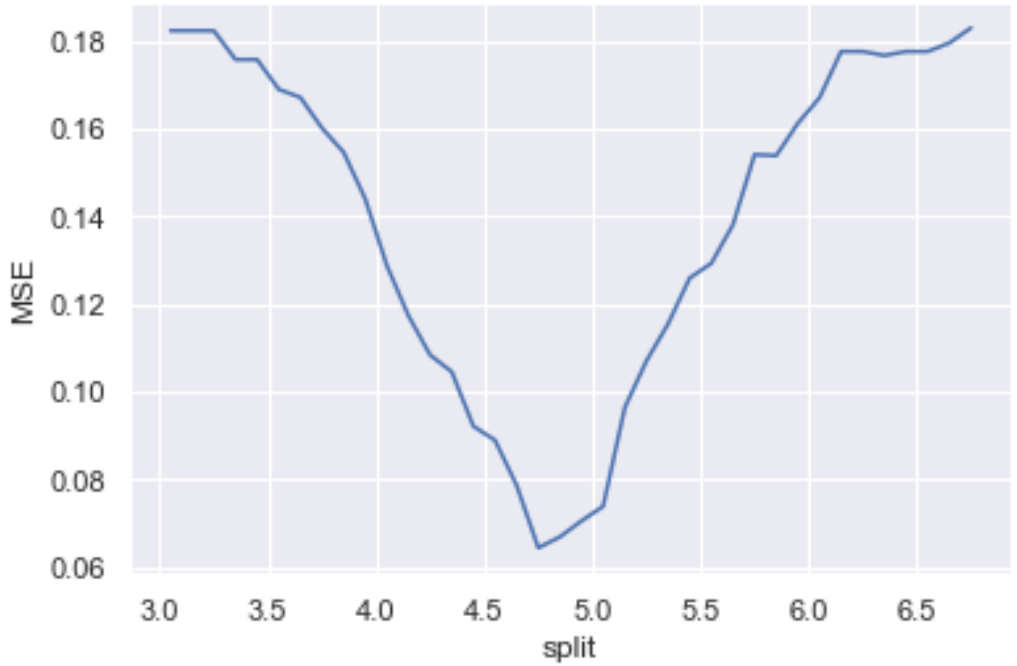
# 3.3.2 Regression - ML Methoden - DecisionTreeRegressor

## 3.3.2.1 Theorie

- analog DecisionTreeClassifier wird für jedes Feature der gesamte Bereich gescannt, um die jeweils beste Split Position zu ermitteln
- hier abgekürztes Verfahren: da auch hier die maximale Verminderung der "impurity" herzugezogen wird, genügt es, die gewichteten MSE-Werte nach dem Split zu vergleichen und das Minimum zu ermitteln
- am Demo Datensatz ergeben sich dabei die folgenden Ergebnisse

```
min_MSE   : 0.0644
min_split : 4.7500
```

- der hier dargestellte Wert min_MSE entspricht dem gewichteten Mittelwert der entsprechenden Kindknoten nach dem Split (vgl. Baumdarstellung unten)



# 3.3.2 Regression - ML Methoden - DecisionTreeRegressor

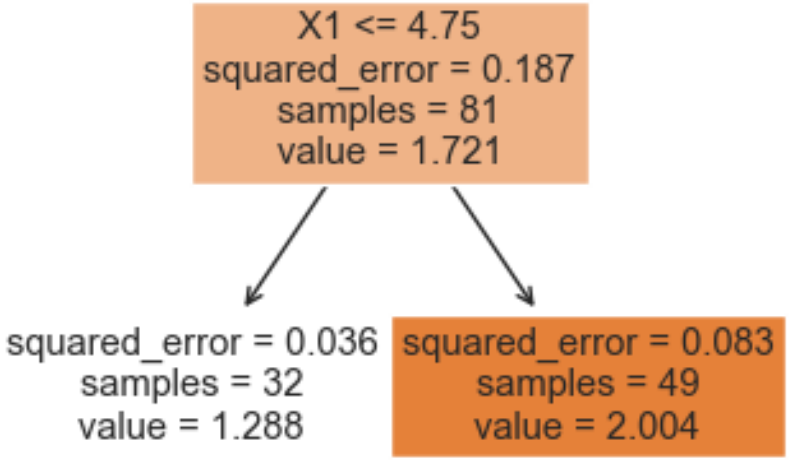
## 3.3.2.1 Theorie

- Kontrolle mit DecisionTreeRegressor

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.tree import export_text
model = DecisionTreeRegressor(max_depth=1)
model.fit(X_demo, y_demo)
print(export_text(model))
```

```
|--- feature_0 <= 4.75
|   |--- value: [1.29]
|--- feature_0 >  4.75
|   |--- value: [2.00]
```

- für dieses Beispiel wurde `max_depth=1` gesetzt, Vorgabe ist aber 'None', d.h. der Baum würde viel detaillierter aufgebaut (mehr dazu unter Praxis)

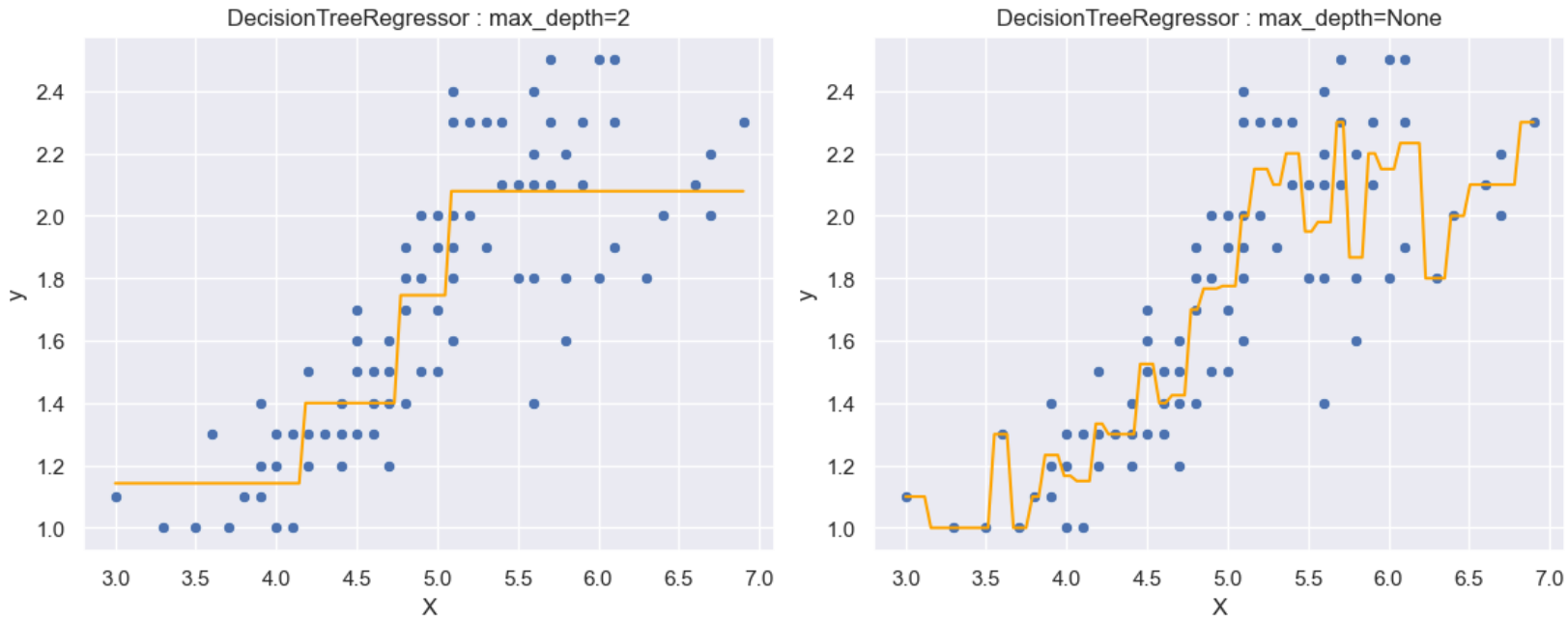


# 3.3.2 Regression - ML Methoden - DecisionTreeRegressor

## 3.3.2.1 Theorie

- wie schon bei DecisionTreeClassifier ist max_depth nicht unbedingt ein idealer Parameter, wurde hier nur zu Demozwecken verwendet
- ein Beispiel mit zwei unterschiedlichen Werten für max_depth

- auch hier gilt:
  mittels Parameter Tuning
  die besten Parameter
  sowie deren Werte
  experimentell zu ermitteln



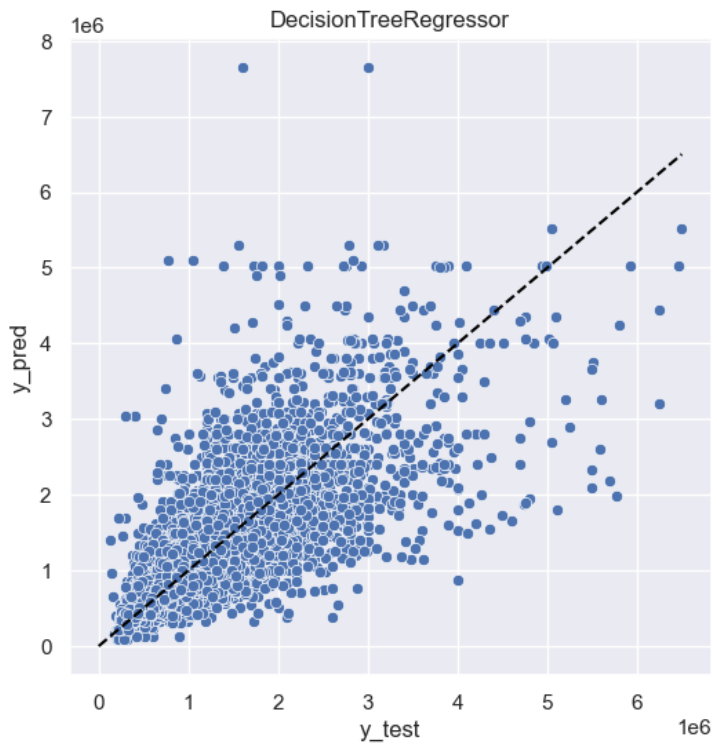
# 3.3.2 Regression - ML Methoden - DecisionTreeRegressor

## 3.3.2.2 Praxis

- Anwendung der eingangs definierten Funktion:

```
from sklearn.tree import DecisionTreeRegressor
this_model = test_regression_model(DecisionTreeRegressor(random_state=1234),
    X_test, y_test, X_train, y_train)
```

```
R2   = 0.5502
```




- Fazit, Performance vergleichbar mit OLS, ausserdem keine negativen Voraussagen und keine Nichtlinearität
- einige interne Methoden:

```
print('get_depth()    : ', this_model.get_depth())
print('get_n_leaves() : ', this_model.get_n_leaves())
```

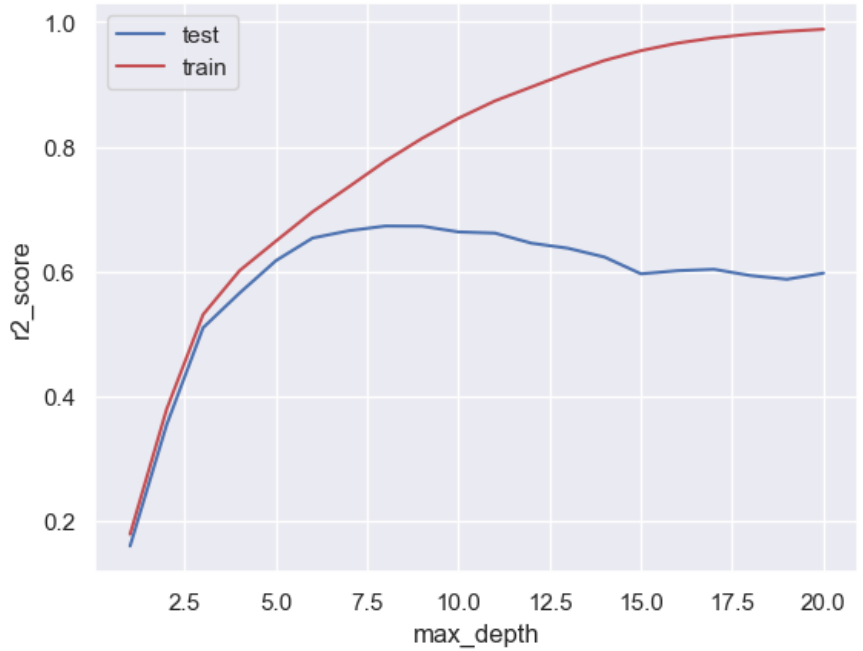
```
get_depth()    :  33
get_n_leaves() :  5947
```

# 3.3.2 Regression - ML Methoden - DecisionTreeRegressor

## 3.3.2.2 Praxis

Parameter-Tuning: max_depth

- ein Vergleich von r2_score für Train und Testdaten über einen Bereich von max_depth zeigt
  - r2 für train steigt kontinuierlich an und erreicht bei max_depth ≈ 20 ein Plateau nahe bei 1
  - r2 für test erreicht das Maximum im Bereich zwischen 7 und 9 und nimmt dann wieder ab
    Vorschlag: `max_depth=8`



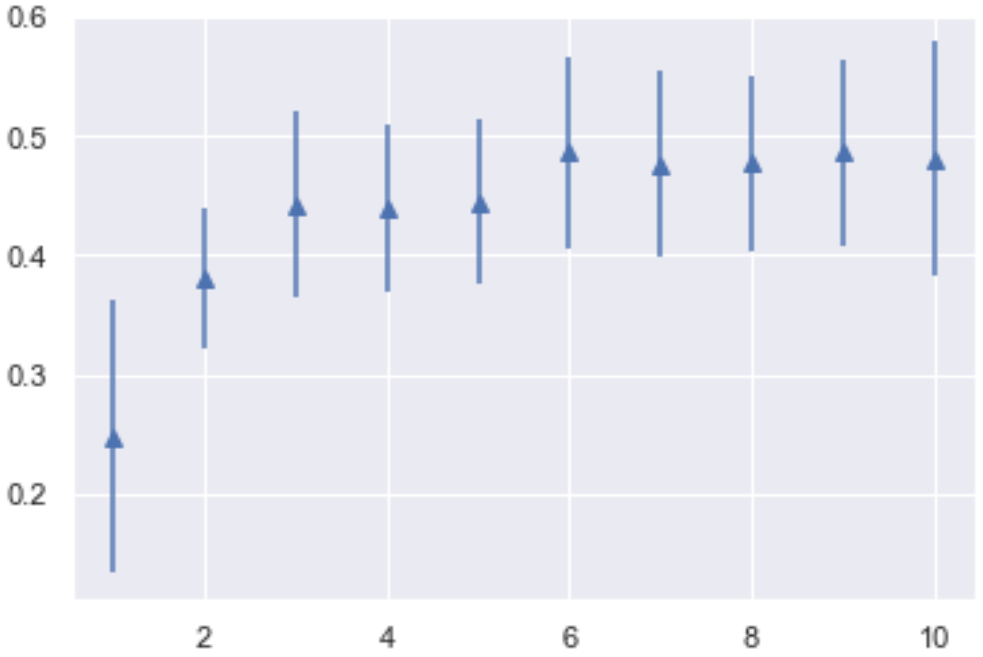
# 3.3.2 Regression - ML Methoden - DecisionTreeRegressor

## 3.3.2.2 Praxis

Ausblick auf Grid Search und Kreuzvalidierung (vgl. Kap. 4.3)

- es werden für min_samples_leaf die Werte von 1 bis 10 untersucht
- für jeden Wert wird eine 5-fache Kreuzvalidierung (default) durchgeführt
- die Resultate werden anschliessend mit sogenannten Error-Bars dargestellt

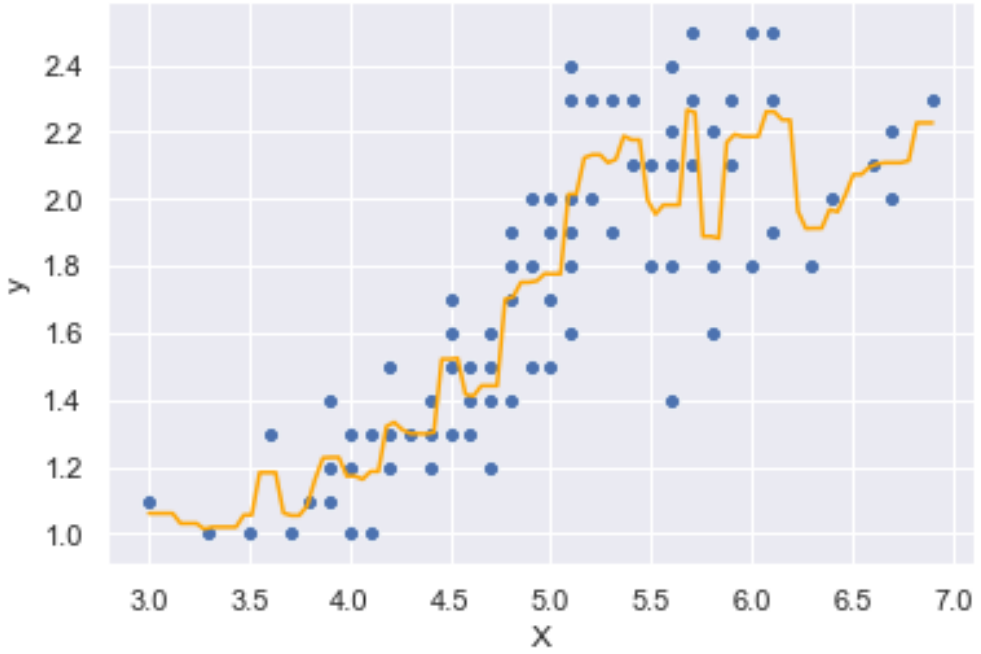
- hier zeigt sich, dass bei 3 ein erstes Plateau erreicht wird, ein zweites dann noch bei 6 (vgl. [ipynb])



# 3.3.3 Regression - ML Methoden - RandomForestRegressor

## 3.3.3.1 Theorie

- das Verfahren entspricht weitgehend dem RandomForestClassifier
- es wird eine vorgegebene Anzahl Regressionsbäume auf unterschiedlichen Subsets der Trainingsdaten erstellt, mit den schon bekannten Parametern
  - `n_estimators=100`
  - `max_features='auto'`, im Gegensatz zu RandomForestClassifier n_features (und nicht sqrt(n_features))
- Training auf den Demodaten mit Standard Parametern und Predict auf synthetische Testdaten zeigt nebenstehendes Bild

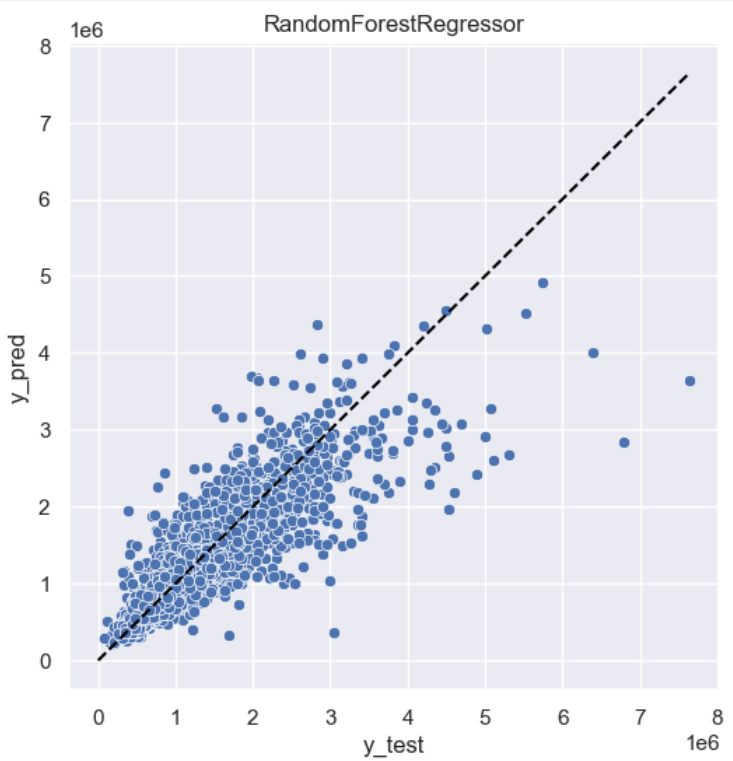


# 3.3.3 Regression - ML Methoden - RandomForestRegressor

## 3.3.3.2 Praxis

```
from sklearn.ensemble import RandomForestRegressor
this_model = test_regression_model(
    RandomForestRegressor(n_estimators=100, random_state=1234),
    X_train, y_train, X_test, y_test)
```

```
R2   = 0.7779
```




Fazit:

- beste Performance bisher

# 3.3.4 Regression - ML Methoden - Weitere Ensemble Regressoren

- analog der Klassifikation existieren auch für Regression weitere Learner für Regressionsfragestellungen:
  - sklearn.ensemble
    - AdaBoostRegressor
    - GradientBoostingRegressor
    - HistGradientBoostingRegressor
  - catboost
    - CatBoostRegressor
  - lightgbm.sklearn
    - LGBMRegressor
- während die Learner-Klassen von sklearn in der Anaconda-Distribution vorliegen, müssen jene für catboost und lightgbm explizit nachinstalliert werden (falls dies nicht bereits für die entsprechenden Klassifikatoren geschehen ist, vgl. Kap. 2.2.7)

# 3.3.4 Regression - ML Methoden - Weitere Ensemble Regressoren

- hier exemplarisch zusammengestellt, zusammen mit den bisher behandelten, jeweils für Standard-Parametrisierung, vgl. Code in `extra_3.3.4_weitere_ensemble_regressoren.ipynb`

| Regressor | R2 | weitere | sklearn-extern |
|---|---|---|---|
| sklearn.linear_model.LinearRegression | 0.5601 | | |
| sklearn.neighbors.KNeighborsRegressor | 0.4493 | | |
| sklearn.tree.DecisionTreeRegressor | 0.5502 | | |
| sklearn.ensemble.RandomForestRegressor | 0.7779 | | |
| sklearn.ensemble.AdaBoostRegressor | -0.3023 | ja | |
| sklearn.ensemble.GradientBoostingRegressor | 0.7250 | ja | |
| sklearn.ensemble.HistGradientBoostingRegressor | 0.7855 | ja | |
| catboost.CatBoostRegressor | 0.8003 | ja | ja |
| lightgbm.LGBMRegressor | 0.7882 | ja | ja |

- Fazit:
  - AdaBoostRegressor ist mit Standard-Parametrisierung untauglich, Potential bei Parameter-Tuning?

# 3.3.4 Regression - ML Methoden


**Workshop 09**

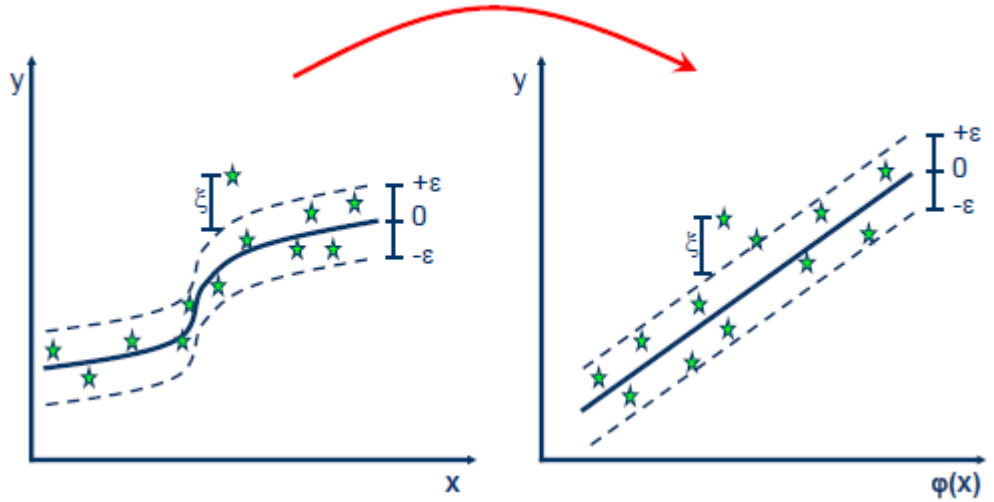
Gruppen zu 2 bis 4, Zeit: 30'

- es wurde festgestellt, dass z.B. `AdaBoostRegressor` unter Standard-Parametrisierung ein unbrauchbares Ergebnis liefert
- untersuchen Sie das Potential von Parameter-Tuning für diesen Regressor
- konzentrieren Sie sich auf folgende Parameter
  - `learning_rate`, Parameter von `AdaBoostRegressor`
  - `max_depth`, interner Parameter des Basis-Estimators, hier `DecisionTreeRegressor`
- falls Zeit übrig, untersuchen Sie noch andere Regressoren Ihrer Wahl dahingehend

# 3.3.5 Regression - ML Methoden - SVR

## 3.3.5.1 Theorie

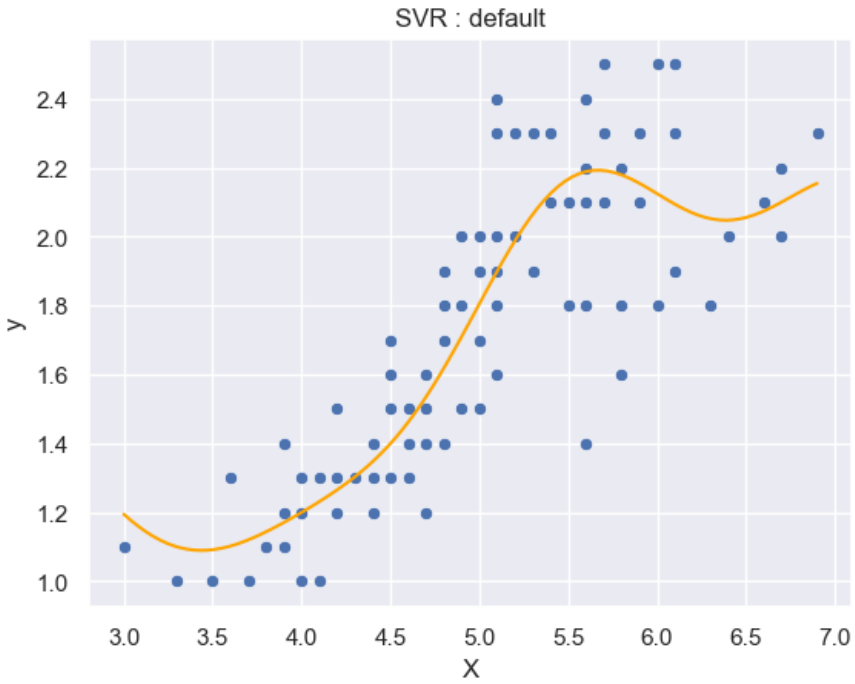
- grundsätzlich gleiches Verfahren wie SVC ausser
  - es wird eine Hyperebene gesucht, welche die Daten möglichst gut abbildet
  - vorab kann ein maximaler Fehler ε (L2-Form) festgelegt werden (Hyperparameter)
- vgl.
  - kaggle
  - Towards Data Science



# 3.3.5 Regression - ML Methoden - SVR

## 3.3.5.1 Theorie

- ein Vergleich mit den Demodaten und Prediction auf synthetische Daten ergibt folgendes Bild
- im Gegensatz zu den Regelbasierten Regressionsmodellen ist folgendes zu erkennen:
  - die Prediction des Modells führt nicht mehr zu einer abgestuften Vorhersage, sondern zu einer geglättet eingepassten Kurve (smoothed curve)
  - dieses Verhalten wird auch bei den folgenden mathematischen ML Methoden zu beobachten sein



# 3.3.5 Regression - ML Methoden - SVR

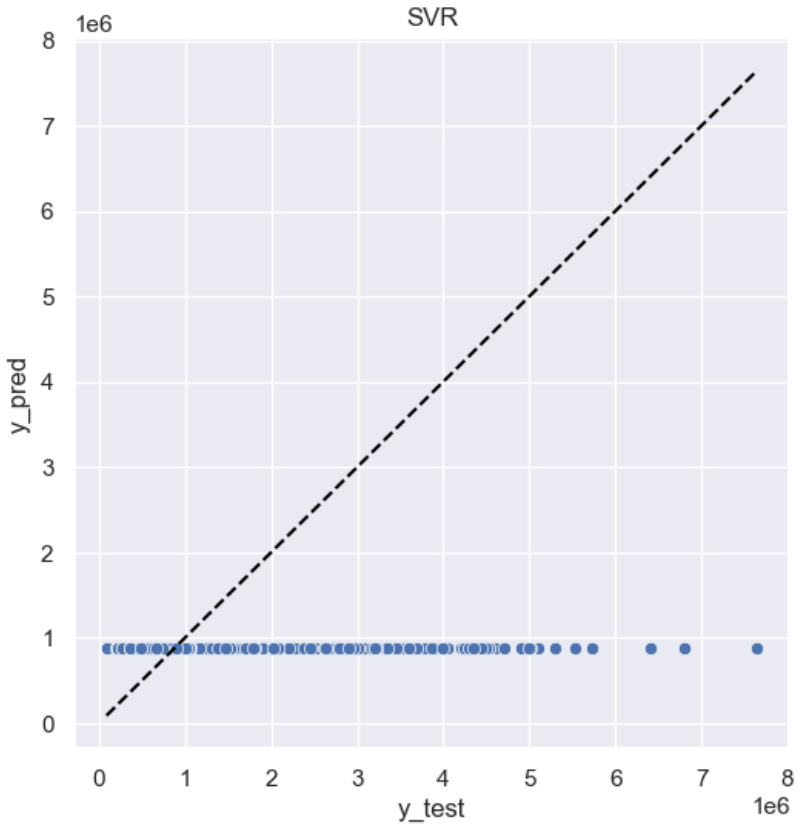
## 3.3.5.2 Praxis

```
from sklearn.svm import SVR
test_regression_model(SVR())
```

```
R2 = -0.0773
```

- suboptimal:
  - die Prediction ist ausschliesslich der Mittelwert von y_train
  - r2 ist negativ (!)

- mögliche Probleme bei dieser Methode:
  - verlangt vorgängiges standardisieren der Features Verteilung des Target ist rechts-schief logarithmieren)

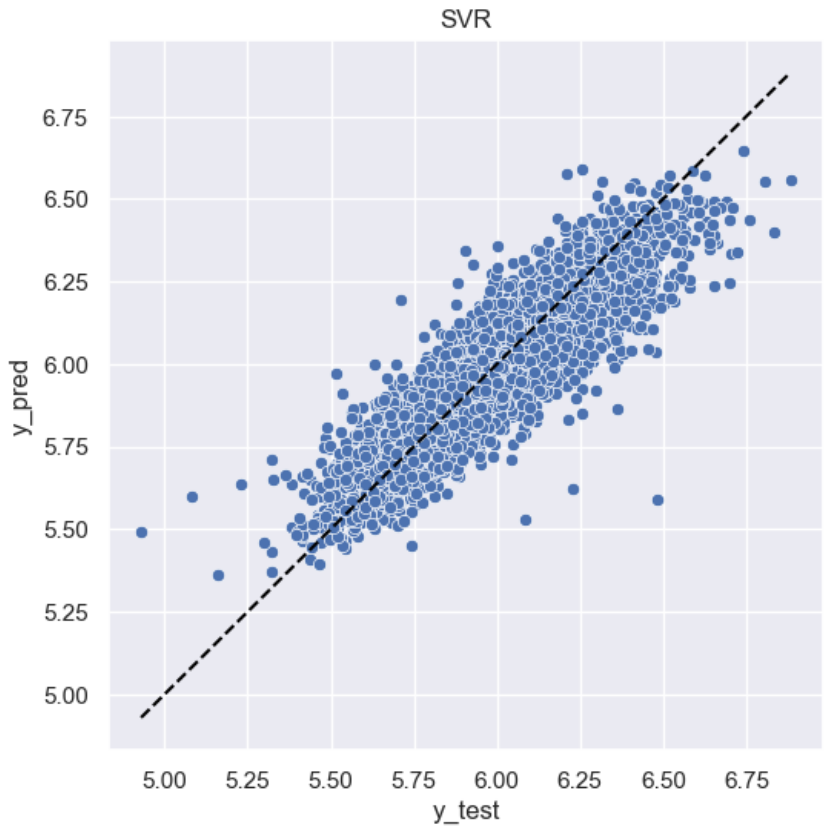


# 3.3.5 Regression - ML Methoden - SVR

## 3.3.5.2 Praxis

- Parameter-Tuning für diesen Regressor auf den vorliegenden Daten ist wenig zielführend und wegen der langen Rechenzeit auch sehr aufwändig
- daher wurde der Einfluss unterschiedlichen Preprocessings einander gegenübergestellt vgl. `extra_3.3.5.2_variants_of_SVR.ipynb`
- Ergebnisse

| scale features | log target | r2_score |
|---|---|---|
| | | -0.0773 |
| X | | -0.0768 |
| | X | 0.0687 |
| X | X | 0.8054 |




- aber achtung, da die Targetwerte durch Logarithmieren viel kleiner werden, wird R2 gezwungernermassen grösser
- für eine faire Beurteilung müssten letztere zur Berechnung von r2 wiederum exponentiell dargestellt werden
- r2 wäre dann 0.7310, was immerhin annehmbar ist

# 3.3.5 Regression - ML Methoden - SVR

### 3.3.5.3 Skalieren und Trainieren in einer Pipeline (Ausblick)

- sklearn.pipeline bietet die Möglichkeit, sequentielle Schritte mit einem abschliessenden Learner in einem Aufruf zu kombinieren (eine Abkürzung)
- im untenstehende Code Beispiel werden `StandardScaler()` und `SVR()` in einer solchen Sequenz zusammengestellt und danach .fit() und .score() auf der Pipeline aufgerufen

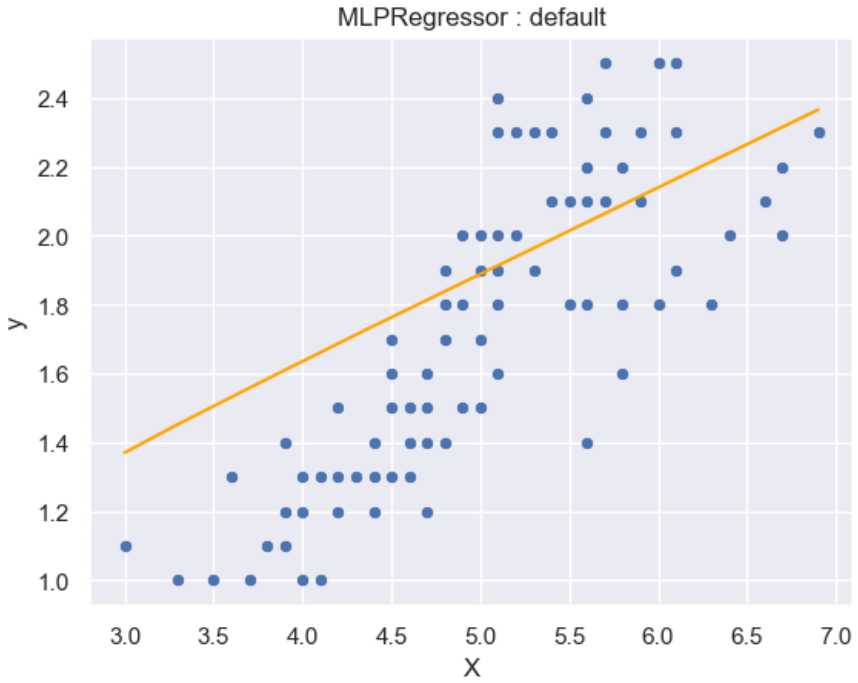
```
from sklearn.pipeline import Pipeline
pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('svr', SVR())])
pipe.fit(X_train, y_train)
```

```
score = 0.5726
```

# 3.3.6 Regression - ML Methoden - MLPRegressor

## 3.3.6.1 Theorie

- die primäre Prediction bei MLPClassifier ist ein numerischer Wert zwischen 0 und 1 (allenfalls -1 und +1), welcher dann aber in die wahrscheinlichste Klasse transformiert wird
- die Modifikation von MLPRegressor besteht hauptsächlich darin, dass am Output Port gerade der Wert des Targets vorausgesagt werden soll
- eine Train - Predict Sequenz mit den Demodaten und Standard Parametrisierung zeigt erst einmal nebenstehendes Bild




- MLPRegressor verfügt (wie auch MLPClassifier) über umfangreiche Möglichkeiten zur Parametrisierung, auf die aber hier nicht weiter eingegangen werden soll
  - vgl. Kursteil Neuronale Netze

# 3.3.6 Regression - ML Methoden - MLPRegressor

## 3.3.6.2 Praxis

```
from sklearn.neural_network import MLPRegressor
    test_regression_model(
    MLPRegressor(random_state=1234),
    X_train, y_train, X_test, y_test,
    show_plot=False)
```

```
... ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached
and the optimization hasn't converged yet.
```

```
R2 = 0.0258
```

- die angezeigte Warnung signalisiert, dass die per Default gewählte Anzahl maximaler Iterationen noch nicht zu Konvergenz geführt hat
- Abhilfe: Erhöhen des Parameterwertes `max_iter`, was allerdings noch nicht zu einer Verbesserung des unbrauchbaren Score-Wertes führt!

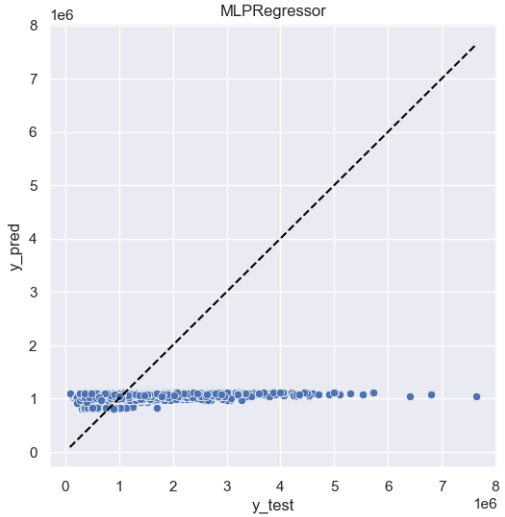
# 3.3.6 Regression - ML Methoden - MLPRegressor

## 3.3.6.2 Praxis

- Versuche mit alternativem Preprocessing (analog SVC) führen dagegen zu folgenden Ergebnissen (vgl. `extra_3.3.6.2_variants_of_MLPRegressor.ipynb`)

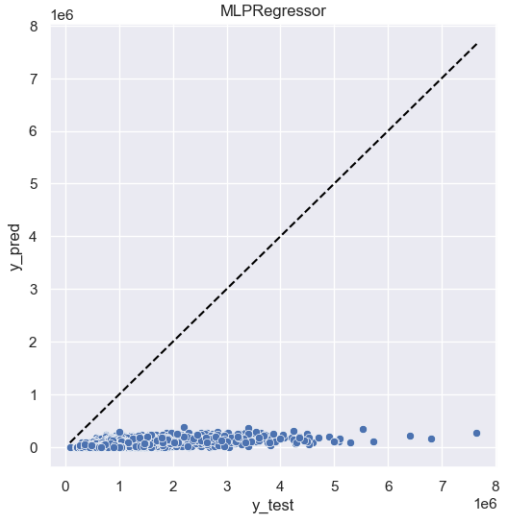
kein Preprocessing

`R2 = 0.0258`



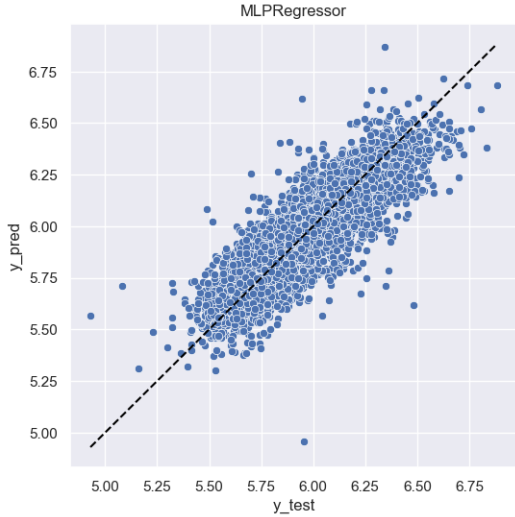

skalieren der Features

`R2 = -2.4330`




skalieren der Features **und** logarithmieren des Targets

`R2 = 0.7205`




- allerdings ist auch hier zu beachten, dass nach einer Rücktransformation des Targets die Performance wieder etwas schlechter wird (0.6553)