

1. Objektorientierung in Python

2. Klassifikationsmetriken

3. Regression im Entscheidungsbaum

## 1.1 Klassen

Klassen in Python sind einfach zu definieren:

```
In [6]: ┌ class Greeter:  
      """Dokumentation der Klasse erfolgt über 3-fache Anführungszeichen"""  
  
      # Konstruktor  
      def __init__(self, name):      # self ist eine Konvention und dient zur Referenzierung der Instanz  
          self.name = name          # Instanz-Variablen  
  
      # Instanz-Methode  
      def greet(self, loud=False):  
          if loud:  
              print(f"HELLO, {self.name.upper()}")  
          else:  
              print(f"Hello, {self.name}!")
```

```
In [178]: ┌ g = Greeter("Fred")    # Erzeuge eine Instanz der Greeter-Klasse  
g.greet()           # Call an instance method; prints "Hello, Fred"  
g.greet(loud=True)  # Call an instance method; prints "HELLO, FRED!"
```

```
Hello, Fred!  
HELLO, FRED
```

```
In [7]: ┌ # Ruft Dokumentationsstring auf
?Greeter
```

```
In [180]: ┌ # Ein Objekt hat standardmäßig keine "Länge"
len(g)
```

```
-----  
TypeError                                     Traceback (most recent call last)
Cell In[180], line 2
      1 # Ein Objekt hat standardmäßig keine "Länge"
----> 2 len(g)
```

TypeError: object of type 'Greeter' has no len()

```
In [9]: ┌ # Klassen können nachträglich verändert werden.
# Implementiere die sog. "Dunder-Methode" __len__, die von der "Funktion" len aufgerufen wird.

Greeter.__len__ = lambda self: len(self.name)
h = Greeter("Fred")
len(h)
```

Out[9]: 4

## 1.2 Dunder-Methoden

```
In [4]: ┌─ print("Hello " + 5)
```

```
-----  
-----  
TypeError  
aceback (most recent call last)  
Cell In[4], line 1  
----> 1 print("Hello " + 5)
```

```
TypeError: can only concatenate str (not "in  
t") to str
```

```
In [5]: ┌─ print("Hello " + str(5))
```

```
Hello 5
```

Implementierung von `str()` und `repr()`:

- `str(x)` ruft die `__str__`-Methode des Objekts `x` auf (`__<name>__` sind sog. *Dunder-Methoden*)
- `repr(x)` ruft die `__repr__`-Methode des Objekts `x` auf
- `str(x)` soll eine Menschen-lesbare Beschreibung des Objekts liefern.
- `repr(x)` soll eine vollständige Beschreibung liefern, mit der das Objekt neu erstellt werden kann.

## Implementierung von `str()` und `repr()`:

- `str(x)` ruft die `__str__`-Methode des Objekts `x` auf (`__<name>__` sind sog. *Dunder-Methoden*)
- `repr(x)` ruft die `__repr__`-Methode des Objekts `x` auf
- `str(x)` soll eine Menschen-lesbare Beschreibung des Objekts liefern.
- `repr(x)` soll eine vollständige Beschreibung liefern, mit der das Objekt neu erstellt werden kann.

```
In [13]: ┌ class Person:  
    def __init__(self, age):      # Konstruktor  
        self.age = age           # das erste Argument "self" ist ein Platzhalter für das Objekt ("this")  
  
        def __str__(self):  
            return "Eine Person, die {0} Jahre alt ist".format(self.age)  
  
        def __repr__(self):  
            return "Person({0})".format(self.age)  
  
p = Person(42)  
  
print ( str(p) )  
print ( repr(p) )
```

Eine Person, die 42 Jahre alt ist  
Person(42)

## 1.3 Datenklassen

Datenklassen sind praktisch, um einfach Daten abzulegen, ohne z.B. den Klassenkontruktor zu verwenden.

```
In [198]: ┶ from dataclasses import dataclass

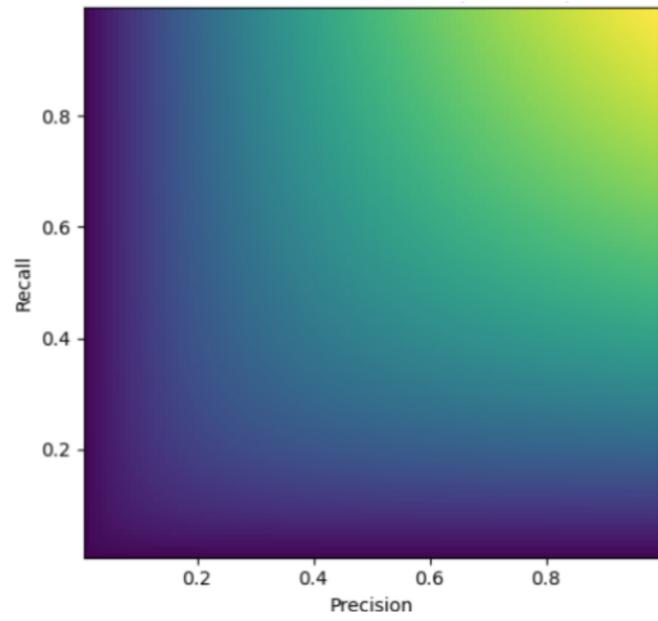
@dataclass
# sog. "Decorator": Ein Wrapper um eine Funktion/Klasse, der deren Funktionalität erweitert
class InventoryItem:
    """Class for keeping track of an item in inventory."""

    name: str          # Beispiele für Type Annotations, diese sind meist optional
    unit_price: float
    quantity_on_hand: int = 0 # default value

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand

item = InventoryItem(name="Schrank", unit_price=120.50, quantity_on_hand=10)
print(item.total_cost())
```

1205.0



1. Objektorientierung in Python
2. Klassifikationsmetriken
3. Regression im Entscheidungsbaum

Daten einlesen:

```
In [21]: ➜ import pandas as pd  
df = pd.read_csv("VL02_Material/heart.csv")  
  
# mit .head() zeigt man nur die ersten Zeilen an  
df.head()  
  
# mit .tail() werden die letzten Zeilen  
# angezeigt.  
  
# Mit dem Parameter n (default n=5)  
# wird die Anzahl der Zeilen spezifiziert.
```

Out[21]:

|   | Age | Sex | ChestPainType | RestingBP | Cholesterol | FastingBS | RestingECG |
|---|-----|-----|---------------|-----------|-------------|-----------|------------|
| 0 | 40  | M   | ATA           | 140       | 289         | 0         | Normal     |
| 1 | 49  | F   | NAP           | 160       | 180         | 0         | Normal     |
| 2 | 37  | M   | ATA           | 130       | 283         | 0         | ST         |
| 3 | 48  | F   | ASY           | 138       | 214         | 0         | Normal     |
| 4 | 54  | M   | NAP           | 150       | 195         | 0         | Normal     |

Holdout-Methode: Split der Daten in Trainings/Validierungs/Testdaten:

```
In [22]: ┌─┐ x = df.iloc[:, :-1]  
y = df.iloc[:, -1] # Zielgröße HeartDisease
```

Verwende `train_test_split`, um Trainingsdaten abzuseparieren:

```
In [23]: ┌─┐ from sklearn.model_selection import train_test_split  
  
# train_size als float (=Anteil der Gesamtmenge)  
# oder als int (=Anzahl Datenpunkte)  
X_train, X_rest, y_train, y_rest = train_test_split(  
    X, y, train_size=1/3, random_state=123 )  
  
len(X_train), len(X_rest), \  
    y_train.mean(), y_rest.mean()  
# 60% der Personen in den Trainingsdaten sind krank
```

Out[23]: (306, 612, 0.6013071895424836, 0.5294117647058824)

```
In [24]: ┌─┐ # Argument "stratify", um die Werte angegebener  
# Spalten gleichmäßig auf die Zielmengen zu verteilen  
X_train, X_rest, y_train, y_rest = train_test_split(  
    X, y, train_size=1/3, random_state=123, stratify=y )  
  
len(X_train), len(X_rest), \  
    y_train.mean(), y_rest.mean()  
# je 55% der Trainings- UND Testpatienten sind krank
```

Out[24]: (306, 612, 0.5522875816993464, 0.553921568627451)



Erneuter Split, um weiter in Validierungs- und Testdaten aufzuteilen mit `train_test_split(..., train_size=?)`:

In [25]:

```
x_val, x_test, y_val, y_test = train_test_split (
    X_rest,y_rest,train_size=1/2,random_state=456 )

len(X_train), len(X_val), len(X_test),\
    y_train.sum(), y_val.sum(), y_test.sum()
```

Out[25]: (306, 306, 306, 169, 178, 161)

In [26]:

```
# erneut mit Stratifizierung:
x_val, x_test, y_val, y_test = train_test_split (
    X_rest,y_rest,train_size=1/2,random_state=456,
    stratify=y_rest )

len(X_train), len(X_val), len(X_test),\
    y_train.sum(), y_val.sum(), y_test.sum()
```

Out[26]: (306, 306, 306, 169, 169, 170)

## 2.1 Ein einfaches Klassifikationsmodell

```
In [27]: # Daten in einem 1/3-1/3-1/3-Split;  
# die Variable y beinhaltet die Zielgröße  
# HeartDisease.  
  
len(X_train), len(X_val), \  
y_train.sum(), y_val.sum()
```

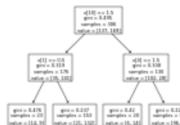
Out[27]: (306, 306, 169, 169)

```
In [28]: # quick and dirty preprocessing (besser: One-Hot-Enc.)  
from sklearn.preprocessing import OrdinalEncoder  
enc = OrdinalEncoder(  
    handle_unknown="use_encoded_value",  
    unknown_value=-1)  
  
X_train = pd.DataFrame (  
    enc.fit_transform(X_train.values),  
    columns = X_train.columns )  
X_val = pd.DataFrame (  
    enc.transform(X_val.values), # not fit_transform  
    columns = X_val.columns )
```

Training eines einfachen Modells:

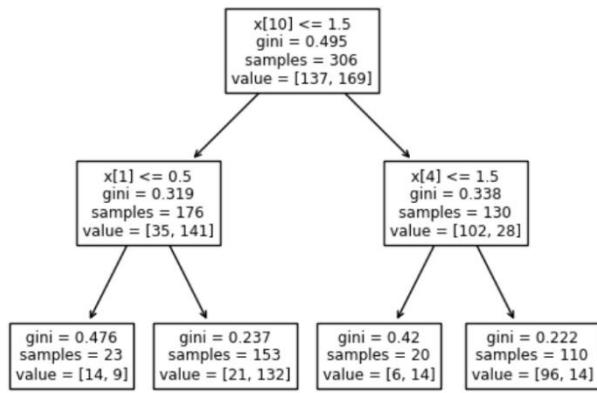
```
In [29]: from sklearn import tree  
clf = tree.DecisionTreeClassifier ( max_depth=2,  
                                    random_state=0 )  
  
clf.fit ( X_train, y_train );
```

```
In [30]: import matplotlib.pyplot as plt  
tree.plot_tree ( clf )  
plt.gcf().set_dpi(30);
```





In [53]: `tree.plot_tree ( clf );`



z.B. value=[6,14] bedeutet: In diesem Blatt gibt es in den Trainingsdaten 6 Gesunde und 14 Kranke.

Daher:

`clf.predict_proba(x_new) -> [[6/20, 14/20]],`

wenn ein neuer Datenpunkt  $x_{new}$  so den Baum durchläuft, dass diese Person in diesem Blatt landet.

Bei einem *Threshold* von 0.5 würde diese Person als "krank" klassifiziert werden, weil der Score 14/20 größer als 0.5 ist.





In [39]: x\_train.iloc[:1,:]

Out[39]:

|   | Age  | Sex | ChestPainType | RestingBP | Cholesterol | FastingBS | RestingECG | MaxHR | ExerciseAngina | Oldpeak | ST_Slope |
|---|------|-----|---------------|-----------|-------------|-----------|------------|-------|----------------|---------|----------|
| 0 | 34.0 | 1.0 | 0.0           | 44.0      | 87.0        | 1.0       | 1.0        | 23.0  | 1.0            | 26.0    | 1.0      |

In [54]:   
# Links: Die finale Vorhersage  
# Rechts: Der Zwischenschritt der "Wahrscheinlichkeiten"  
# (hier heißt der Wert 0.86 "Score")  
clf.predict ( X\_train.iloc[:1,:] ), \  
clf.predict\_proba ( X\_train.iloc[:1,:] )

Out[54]: (array([1], dtype=int64), array([[0.1372549, 0.8627451]]))

In [55]:   
# Um von Scores zu Predictions zu kommen, muss der Score mit einem Threshold verglichen werden (standardmäßig 0.5):  
clf.predict\_proba ( X\_train.iloc[:1,:] )[0,1] > 0.5

Out[55]: True





Berechnung der Konfusionsmatrix:

```
In [145]: # predictions  
pred_val = clf.predict(X_val)
```

```
In [85]: # Rechts oben sollte "False Positives" sein:  
  
( (y_val == 0) & (pred_val == 1) ).sum()  
  
# Frage: Wo ist der Fehler?
```

```
Out[85]: 48
```

```
In [89]: # Lieber nochmal überprüfen: "True Positives"  
# sollte Links oben sein:  
  
( (y_val == 1) & (pred_val == 1) ).sum()  
  
# Frage: ???
```

```
Out[89]: 149
```

```
In [87]: from sklearn.metrics import confusion_matrix  
confusion_matrix( pred_val, y_val )
```

```
Out[87]: array([[ 89,  20],  
[ 48, 149]], dtype=int64)
```

```
In [88]: # Dokumentation: confusion_matrix(y_true, y_pred, ...)  
# => Die Reihenfolge ist wichtig, eine Vertauschung  
#     spiegelt die Matrix.  
  
confusion_matrix( y_val, pred_val )
```

```
Out[88]: array([[ 89,  48],  
[ 20, 149]], dtype=int64)
```

```
In [91]: # Dokumentation: "other references may use  
# a different convention for axes"  
TN, FP, FN, TP = confusion_matrix(y_val,  
                                 pred_val).flatten()  
  
import numpy as np  
np.array ( [[TP,FP],[FN,TN]] )
```

```
Out[91]: array([[149,  48],  
[ 20,  89]], dtype=int64)
```





## 2.2 Metriken für eine Binäre Klassifikation

| Wahre Klasse:         | positiv (krank)  | negativ (gesund) |
|-----------------------|------------------|------------------|
| Vorhergesagte Klasse: | positiv (krank)  | 149 (TP)         |
|                       | negativ (gesund) | 89 (TN)          |

In [92]: `# Accuracy:  
(TP+TN) / (TP+TN+FP+FN)`

Out[92]: 0.7777777777777778

In [96]: `# Precision:  
# Wenn mein Testergebnis 1 ist, wie wahrscheinlich ist  
# es, dass ich krank bin?  
TP / (TP+FP)`

Out[96]: 0.7563451776649747

In [98]: `# Sensitivität (=Recall):  
TP / (TP+FN)`

Out[98]: 0.8816568047337278

In [95]: `# Erinnerung: 55% der Daten sind 1er, 45% sind 0er  
y_val.mean()`

Out[95]: 0.5522875816993464

In [97]: `# Recall:  
# Von allen tatsächlich kranken,  
# wie viele wurden als krank erkannt?  
TP / (TP+FN)`

Out[97]: 0.8816568047337278

In [99]: `# Spezifität:  
# von allen tatsächlich gesunden,  
# wie viele wurden als gesund erkannt?  
TN / (TN+FP)`

Out[99]: 0.6496350364963503

Frage: Wann sollte welche Metrik verwendet werden?





Frage: Wann sollte welche Metrik verwendet werden?

In [100]: # Accuracy:  
 $(TP+TN) / (TP+TN+FP+FN)$

Out[100]: 0.7777777777777778

In [101]: # Precision, Recall  
 $TP / (TP+FP)$ ,  $TP / (TP+FN)$

Out[101]: (0.7563451776649747, 0.8816568047337278)

In [102]: # Sensitivität, Spezifität  
 $TP / (TP+FN)$ ,  $TN / (TN+FP)$

Out[102]: (0.8816568047337278, 0.6496350364963503)

| Wahre Klasse:         | positiv | negativ          |
|-----------------------|---------|------------------|
| Vorhergesagte Klasse: | positiv | 149 (TP) 48 (FP) |
|                       | negativ | 20 (FN) 89 (TN)  |

Die Konfusionsmatrix liefert die vollständige Information. Die Accuracy ist eine Komprimierung auf eine Zahl, die sinnvoll verwendet werden kann bei balancierten Daten.

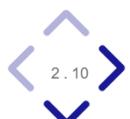
Das Begriffspaar Precision/Recall ist maßgeschneidert auf den "typischen" Fall, dass "Positives" selten sind.

- Precision: Wie groß ist die WSK, dass ich krank bin, wenn ich einen positiven Test habe?
- Recall: Wie viele von den "Positives" in der Gesamtpopulation werden erkannt?

Das Paar Precision/Recall kann im sog. F1-Score zusammengefasst werden.

Sensitivität/Spezifität ist symmetrisch formuliert für Positives und Negatives.

Können, noch bevor ein Threshold Scores in Predictions übersetzt, im sog. (ROC-)AUC kombiniert werden; nützliches frühes Optimierungsziel.





### 3.3 Precision-Recall-Tradeoff

```
In [105]: # Erinnerung: Klassifikationsmodelle geben i.A.  
# scores aus, aus denen durch Prüfung auf  
# Schwellenwertüberschreitung  
# Vorhersagen generiert werden.  
  
scores = clf.predict_proba ( X_val )  
scores[:20,:]
```

```
Out[105]: array([[0.87272727, 0.12727273],  
[0.87272727, 0.12727273],  
[0.1372549 , 0.8627451 ],  
[0.1372549 , 0.8627451 ],  
[0.87272727, 0.12727273],  
[0.3      , 0.7      ],  
[0.87272727, 0.12727273],  
[0.1372549 , 0.8627451 ],  
[0.87272727, 0.12727273],  
[0.87272727, 0.12727273],  
[0.1372549 , 0.8627451 ],  
[0.1372549 , 0.8627451 ],  
[0.87272727, 0.12727273],  
[0.87272727, 0.12727273],  
[0.60869565, 0.39130435],  
[0.1372549 , 0.8627451 ],  
[0.1372549 , 0.8627451 ],  
[0.60869565, 0.39130435],  
[0.60869565, 0.39130435],  
[0.87272727, 0.12727273]])
```

```
In [107]: # typischerweise versteht man unter Scores den Wert  
# für die Klasse 1  
scores = scores[:,1]  
scores[:5]
```

```
Out[107]: array([0.12727273, 0.12727273, 0.8627451 , 0.8627451 ,  
0.12727273])
```

```
In [108]: # Generierung von binären Vorhersagen  
threshold = 0.5  
pred_val2 = (scores > threshold)  
pred_val2[:5]
```

```
Out[108]: array([False, False, True, True, False])
```

```
In [109]: # Stimmen unsere Vorhersagen überein mit den  
# Vorhersagen von clf.predict()?  
pred_val = clf.predict(X_val)  
(pred_val == pred_val2).mean()
```

```
Out[109]: 1.0
```

```
In [112]: # Veränderung des Thresholds  
threshold = 0.1  
pred_val2 = (scores > threshold)  
pred_val2[:5]
```

```
Out[112]: array([ True,  True,  True,  True,  True])
```





## 2.4 Zusammenfassung

Ein niedriger Threshold würde zu vielen 1-Vorhersagen führen (niedrige Precision, hoher Recall).

Ein hoher Threshold würde zu vielen 0-Vorhersagen führen (hohe Precision, niedrige Recall).

Der **Precision-Recall Tradeoff** besagt, dass durch die Wahl des Thresholds nicht Precision und Recall gleichzeitig maximiert werden können. Der Threshold wird unter Berücksichtigung der "Kosten" für False Positives und False Negatives für den konkreten Anwendungsfall optimiert.

**Beispiel:** In einem Krebstest möchte man vielleicht eine hohe Sensitivität (=Recall) erreichen und nimmt eine niedrige Precision in Kauf, da man "False Positives" als weniger schlimm bewertet als "False Negatives".

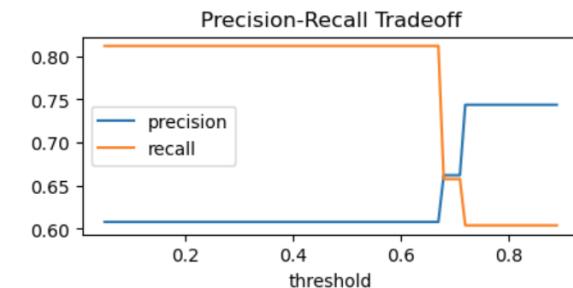
In [448]:

```
from sklearn.metrics import precision_score, \
                           recall_score

# generiere Predictions für verschiedene Thresholds
thresholds = np.arange(0.05,0.9,0.01)
preds      = [(scores > t) for t in thresholds]

# Berechne jeweils Precision und Recall
precs     = [precision_score(y_val,p) for p in preds]
recalls   = [recall_score(y_val,p)      for p in preds]

# Visualisierung
fig,ax = plt.subplots ( figsize=(5,2) )
ax.plot ( thresholds, precs,   label="precision" )
ax.plot ( thresholds, recalls, label="recall"   )
ax.set_title("Precision-Recall Tradeoff")
ax.set_xlabel("threshold")
ax.legend();
```

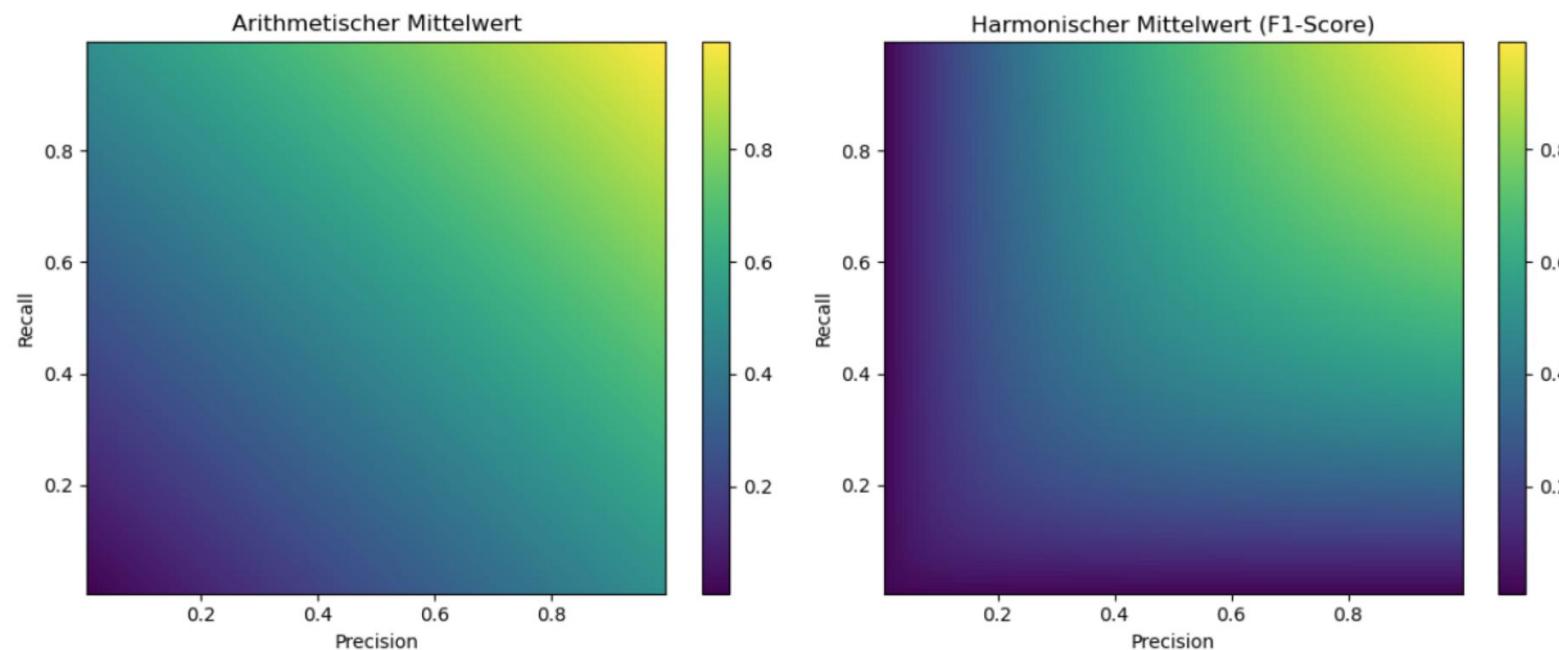




## 2.6 F1-Score als harmonischer Mittelwert von Precision und Recall

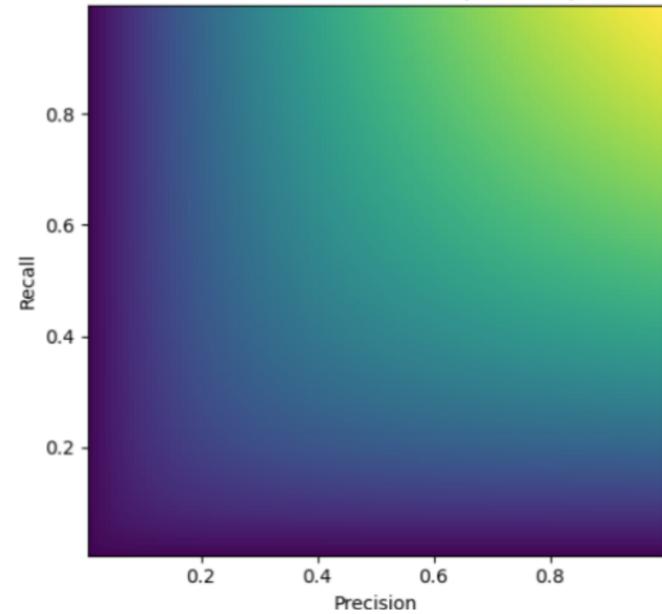
Um eine eindimensionale Zielgröße zu haben, wird oft der F1-Score verwendet:

$$F1 = \frac{2}{1/\text{Precision} + 1/\text{Recall}}$$



Der harmonische Mittelwert erzwingt, dass der Mittelwert 0 ist, wenn einer der beiden Werte 0 ist.





1. Objektorientierung in Python
2. Klassifikationsmetriken
3. Regression im Entscheidungsbaum





## 3.1 Daten mit einer kontinuierlichen Zielgröße

In [146]:

```
# Load the diabetes dataset
from sklearn.datasets import load_diabetes
X = load_diabetes(as_frame=True)

# unpack values
X,y = X[["data"]], X[["target"]]
df = pd.concat([X,y],axis=1)
df
```

Out[146]:

|     | age       | sex       | bmi       | bp        | s1        | s2        | s3        | s4        | s5        | s6        | target |
|-----|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|--------|
| 0   | 0.038076  | 0.050680  | 0.061696  | 0.021872  | -0.044223 | -0.034821 | -0.043401 | -0.002592 | 0.019907  | -0.017646 | 151.0  |
| 1   | -0.001882 | -0.044642 | -0.051474 | -0.026328 | -0.008449 | -0.019163 | 0.074412  | -0.039493 | -0.068332 | -0.092204 | 75.0   |
| 2   | 0.085299  | 0.050680  | 0.044451  | -0.005670 | -0.045599 | -0.034194 | -0.032356 | -0.002592 | 0.002861  | -0.025930 | 141.0  |
| 3   | -0.089063 | -0.044642 | -0.011595 | -0.036656 | 0.012191  | 0.024991  | -0.036038 | 0.034309  | 0.022688  | -0.009362 | 206.0  |
| 4   | 0.005383  | -0.044642 | -0.036385 | 0.021872  | 0.003935  | 0.015596  | 0.008142  | -0.002592 | -0.031988 | -0.046641 | 135.0  |
| ... | ...       | ...       | ...       | ...       | ...       | ...       | ...       | ...       | ...       | ...       | ...    |
| 437 | 0.041708  | 0.050680  | 0.019662  | 0.059744  | -0.005697 | -0.002566 | -0.028674 | -0.002592 | 0.031193  | 0.007207  | 178.0  |
| 438 | -0.005515 | 0.050680  | -0.015906 | -0.067642 | 0.049341  | 0.079165  | -0.028674 | 0.034309  | -0.018114 | 0.044485  | 104.0  |
| 439 | 0.041708  | 0.050680  | -0.015906 | 0.017293  | -0.037344 | -0.013840 | -0.024993 | -0.011080 | -0.046883 | 0.015491  | 132.0  |
| 440 | -0.045472 | -0.044642 | 0.039062  | 0.001215  | 0.016318  | 0.015283  | -0.028674 | 0.026560  | 0.044529  | -0.025930 | 220.0  |
| 441 | -0.045472 | -0.044642 | -0.073030 | -0.081413 | 0.083740  | 0.027809  | 0.173816  | -0.039493 | -0.004222 | 0.003064  | 57.0   |

442 rows × 11 columns





In [79]:

```
# df.info() für einen ersten Überblick.  
# Ergebnis: Wir haben keine fehlenden Werte.  
# Alle Daten sind numerisch.  
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 442 entries, 0 to 441  
Data columns (total 11 columns):  
 #   Column  Non-Null Count  Dtype     
---    
 0   age      442 non-null   float64  
 1   sex      442 non-null   float64  
 2   bmi      442 non-null   float64  
 3   bp       442 non-null   float64  
 4   s1       442 non-null   float64  
 5   s2       442 non-null   float64  
 6   s3       442 non-null   float64  
 7   s4       442 non-null   float64  
 8   s5       442 non-null   float64  
 9   s6       442 non-null   float64  
 10  target    442 non-null   float64  
dtypes: float64(11)  
memory usage: 38.1 KB
```

Features s1-s6 sind diverse Blutwerte:

| Spalte | Beschreibung  |
|--------|---|
| age    | age in years  |
| sex    | sex   |
| bmi    | body mass index   |
| bp     | average blood pressure  |
| s1     | tc, total serum cholesterol   |
| s2     | ldl, low-density lipoproteins   |
| s3     | hdl, high-density lipoproteins  |
| s4     | tch, total cholesterol / HDL  |
| s5     | ltg, possibly log of serum triglycerides level                        |
| s6     | glu, blood sugar level  |
| target | a quantitative measure of disease progression one year after baseline |

[https://scikit-learn.org/stable/datasets/toy\\_dataset.html#diabetes-dataset](https://scikit-learn.org/stable/datasets/toy_dataset.html#diabetes-dataset)





In [80]: `# df.describe() für eine Beschreibung der numerischen Spalten durch diverse "Statistiken" (=Kenngrößen).  
df.describe()`

Out[80]:

|       | age           | sex           | bmi           | bp            | s1            | s2            | s3            | s4            | s5            | s6            | target     |
|-------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|------------|
| count | 4.420000e+02  | 442.000000 |
| mean  | -2.511817e-19 | 1.230790e-17  | -2.245564e-16 | -4.797570e-17 | -1.381499e-17 | 3.918434e-17  | -5.777179e-18 | -9.042540e-18 | 9.293722e-17  | 1.130318e-17  | 152.133484 |
| std   | 4.761905e-02  | 77.093005  |
| min   | -1.072256e-01 | -4.464164e-02 | -9.027530e-02 | -1.123988e-01 | -1.267807e-01 | -1.156131e-01 | -1.023071e-01 | -7.639450e-02 | -1.260971e-01 | -1.377672e-01 | 25.000000  |
| 25%   | -3.729927e-02 | -4.464164e-02 | -3.422907e-02 | -3.665608e-02 | -3.424784e-02 | -3.035840e-02 | -3.511716e-02 | -3.949338e-02 | -3.324559e-02 | -3.317903e-02 | 87.000000  |
| 50%   | 5.383060e-03  | -4.464164e-02 | -7.283766e-03 | -5.670422e-03 | -4.320866e-03 | -3.819065e-03 | -6.584468e-03 | -2.592262e-03 | -1.947171e-03 | -1.077698e-03 | 140.500000 |
| 75%   | 3.807591e-02  | 5.068012e-02  | 3.124802e-02  | 3.564379e-02  | 2.835801e-02  | 2.984439e-02  | 2.931150e-02  | 3.430886e-02  | 3.243232e-02  | 2.791705e-02  | 211.500000 |
| max   | 1.107267e-01  | 5.068012e-02  | 1.705552e-01  | 1.320436e-01  | 1.539137e-01  | 1.987880e-01  | 1.811791e-01  | 1.852344e-01  | 1.335973e-01  | 1.356118e-01  | 346.000000 |

Lagemaße mean und 50%-Quantil (=Median):

Wo liegen die Daten etwa?

Streumaße std (=Standardabweichung) oder

Inter Quartile Range = 75%- minus 25%-Quantil:

Wie sehr streuen die Daten?

In [85]: `df["sex"].head()`

Out[85]:

```
0    0.050680
1   -0.044642
2    0.050680
3   -0.044642
4   -0.044642
Name: sex, dtype: float64
```

In [88]: `# Documentation: Each of these 10 feature variables
# have been mean centered and scaled by the standard
# deviation times the square root of n_samples
df["sex"].mean(), df["sex"].std()*np.sqrt(len(df))`

Out[88]:

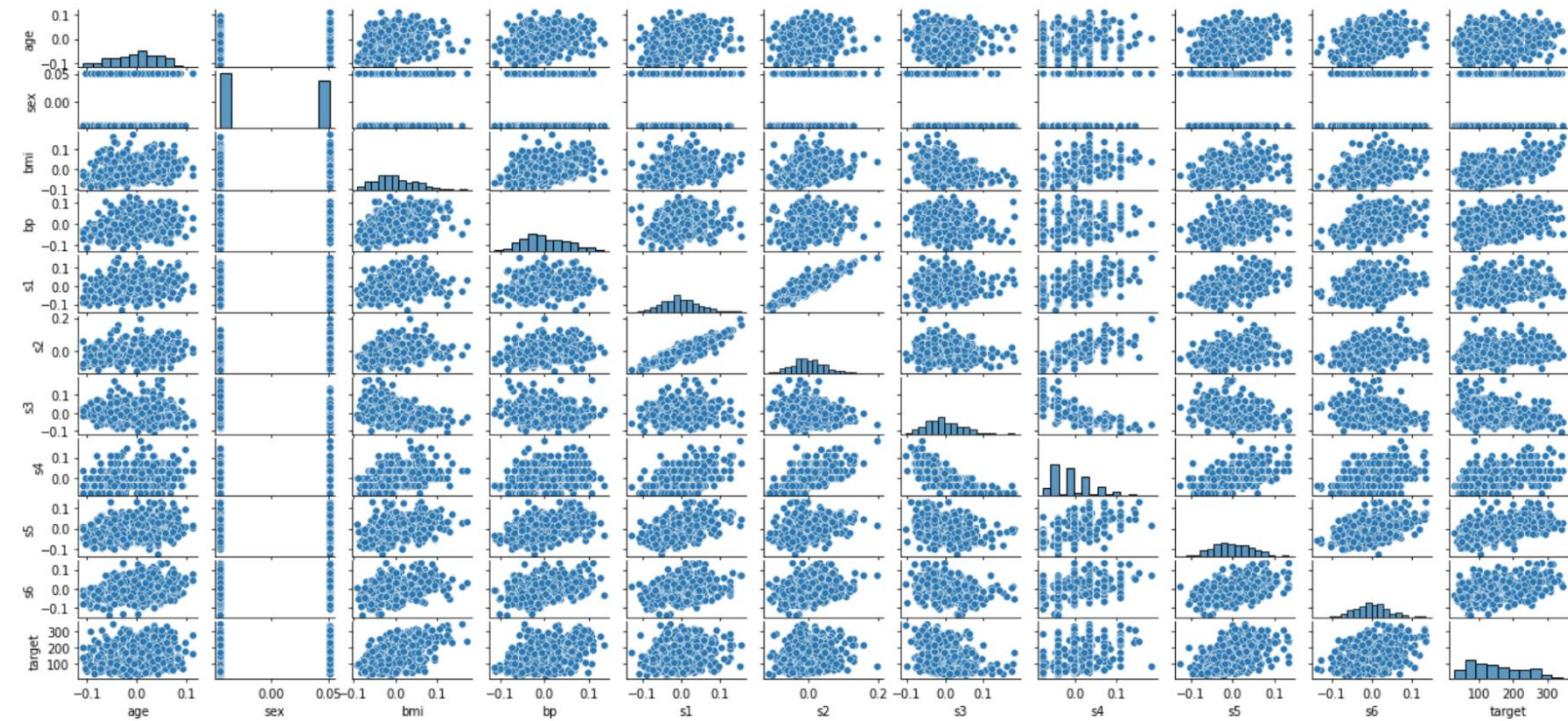
```
(1.2307902309192911e-17, 1.00113314483946)
```





Abschließend: Überblick verschaffen.

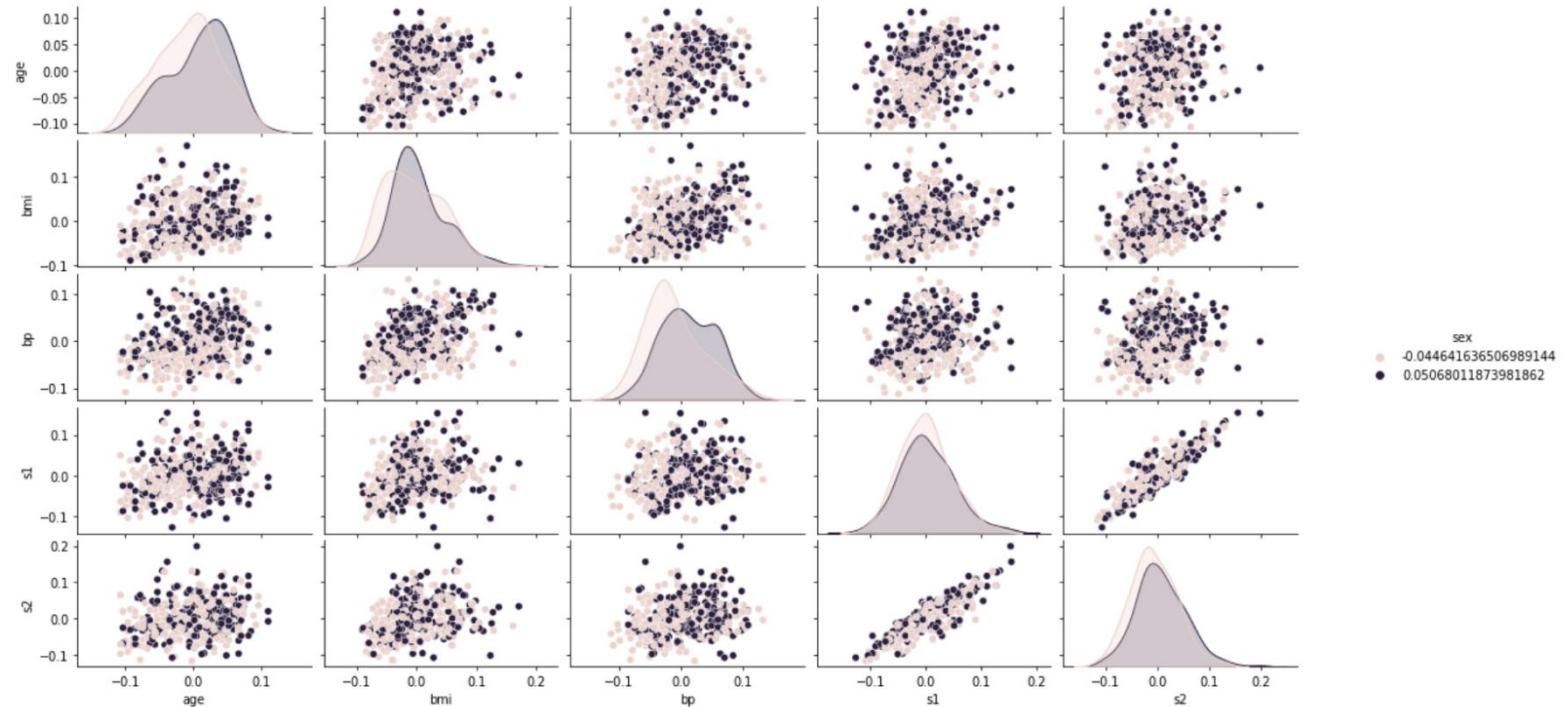
In [95]: # 11 Spalten sind vielleicht doch zu viel?  
import seaborn as sns; sns.pairplot ( df )  
plt.gcf().set\_size\_inches ( 18, 8 ); plt.gcf().set\_dpi(70);





In [96]:

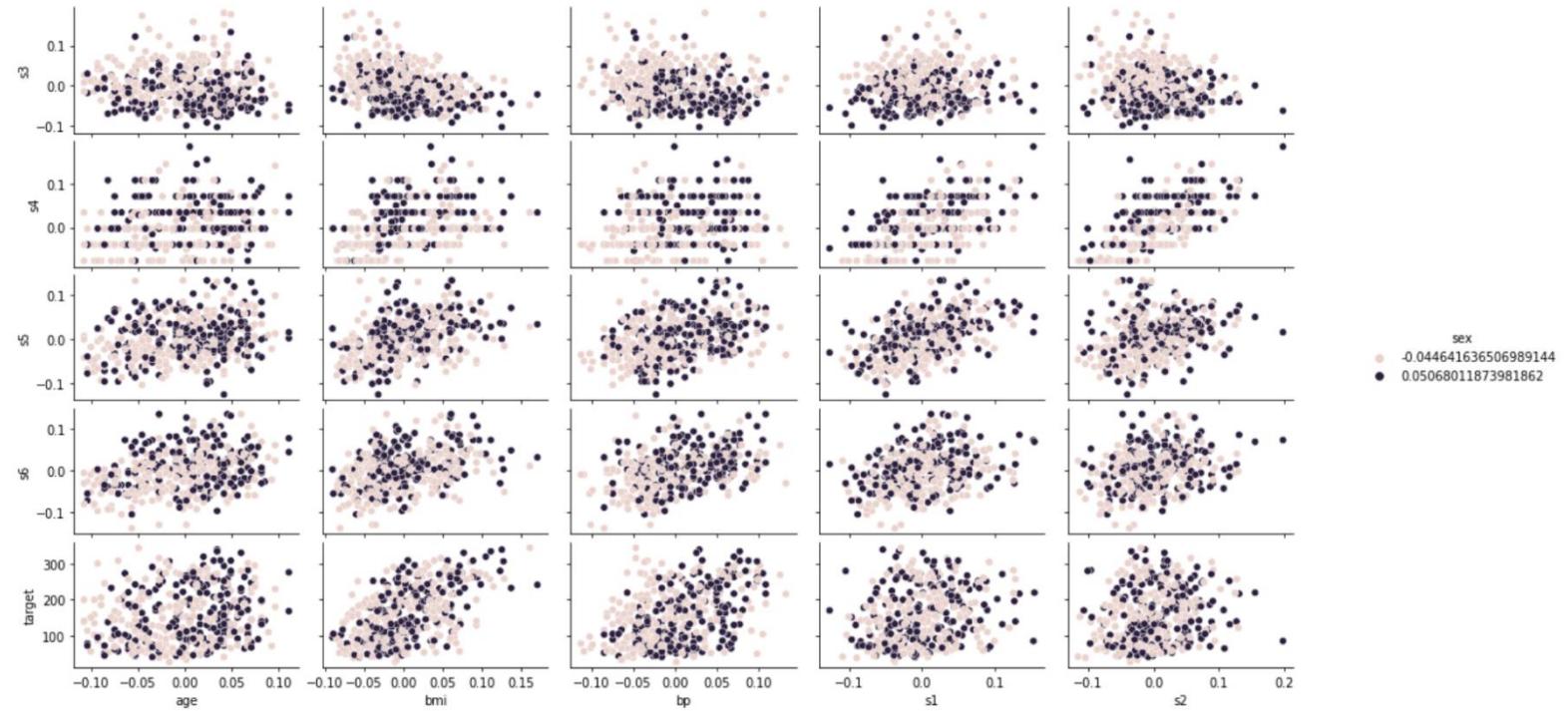
```
# Plot 1 von 4
cols = [c for c in df.columns if c != "sex"]
sns.pairplot ( df, x_vars=cols[:5], y_vars=cols[:5], hue="sex" )
plt.gcf().set_size_inches ( 18, 8 ); plt.gcf().set_dpi(70);
```





In [98]:

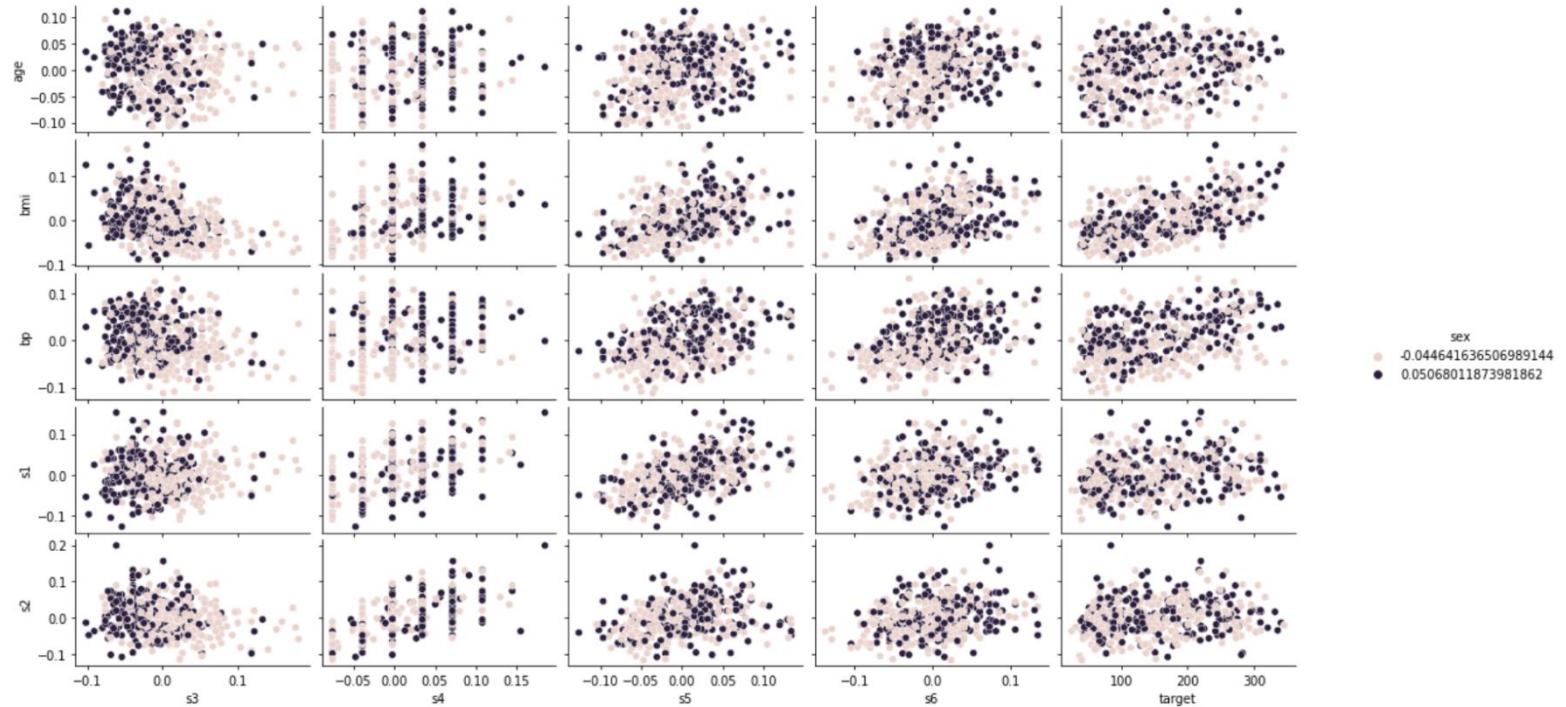
```
# Plot 2 von 4
sns.pairplot ( df, x_vars=cols[:5], y_vars=cols[5:], hue="sex" )
plt.gcf().set_size_inches ( 18, 8 ); plt.gcf().set_dpi(70);
```





In [99]:

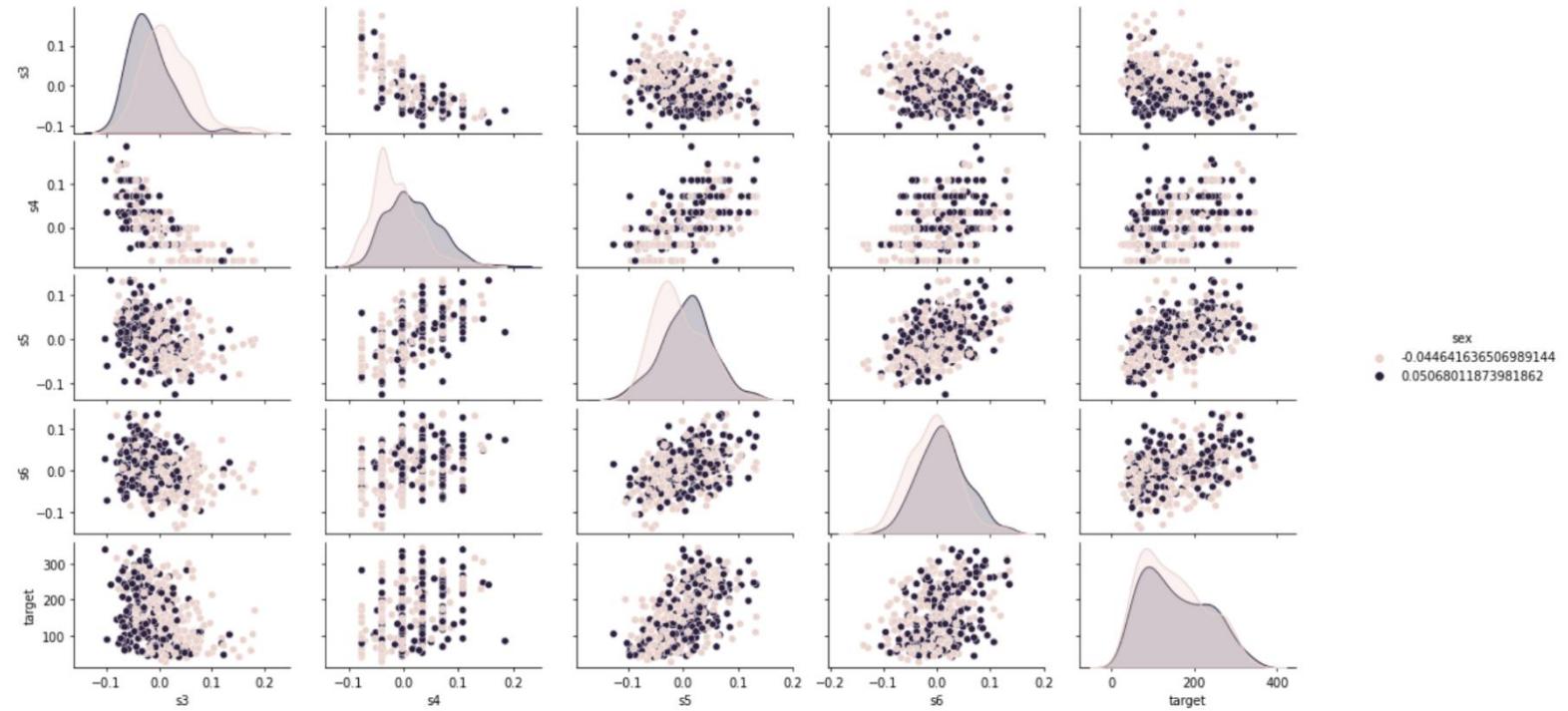
```
# Plot 3 von 4
sns.pairplot ( df, x_vars=cols[5:], y_vars=cols[:5], hue="sex" )
plt.gcf().set_size_inches ( 18, 8 ); plt.gcf().set_dpi(70);
```





In [100]:

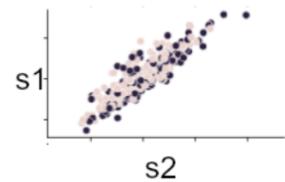
```
# Plot 4 von 4
sns.pairplot ( df, x_vars=cols[5:], y_vars=cols[5:], hue="sex" )
plt.gcf().set_size_inches ( 18, 8 ); plt.gcf().set_dpi(70);
```



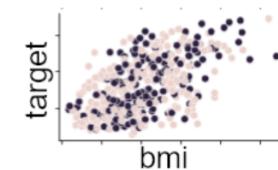


### Ergebnisse (z.B.):

Blutwerte sind z.T. hochkorreliert, hier `s1` (=total serum cholesterol) und `s2` (=low-density lipoproteins).



Die Zielgröße `target` zeigt z.B. mit der Variable `bmi` eine hohe Korrelation.



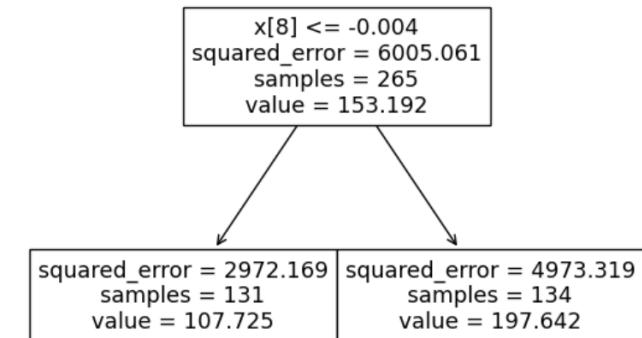


## 3.2 Train-Test-Split und Training eines Entscheidungsbaums

```
In [116]: # Verzichte der Einfachheit halber auf einen  
# zusätzlichen Testdatensatz  
# (wir tun so, als hätten wir diesen bereits  
# zur Seite gelegt)  
from sklearn.model_selection import train_test_split  
X_train, X_test, y_train, y_test = \  
    train_test_split(X, y, random_state=42,  
                      train_size=0.6, stratify=X[\"sex\"])  
  
len(X_train), len(X_test)
```

Out[116]: (265, 177)

```
In [117]: # Trainiere einen Entscheidungsbaum  
# auf den Regressions-Task, die kontinuierliche  
# Zielgröße "target" vorherzusagen  
from sklearn.tree import DecisionTreeRegressor  
reg = DecisionTreeRegressor( max_depth=1 )  
reg.fit ( X_train, y_train )  
  
from sklearn.tree import plot_tree  
plot_tree ( reg );
```

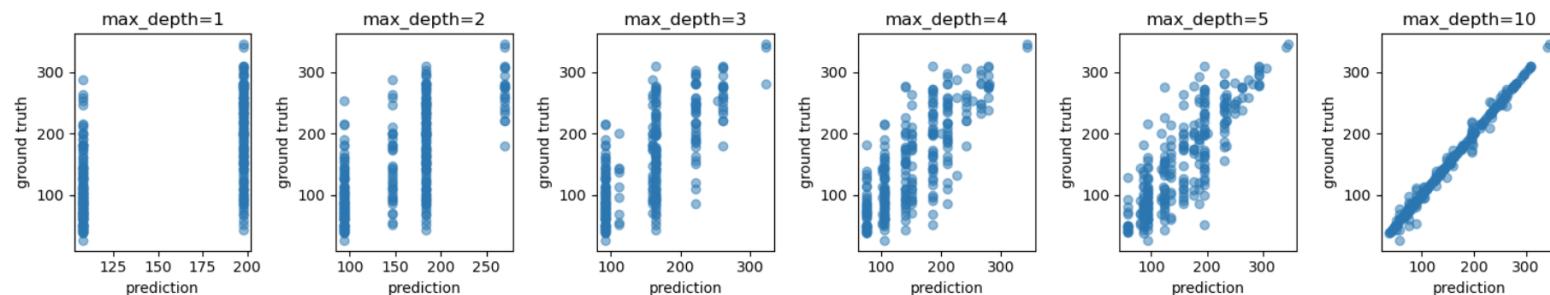




### 3.3 Predictions für verschiedene Baumtiefen

```
In [121]: fig, axes = plt.subplots ( figsize=(15,3), ncols=6 )
for max_depth, ax in zip ( [1,2,3,4,5,10], axes ):
    # train
    reg = DecisionTreeRegressor( max_depth=max_depth, random_state=0 )
    reg.fit ( X_train, y_train )

    # plot the predictions on the training set
    ax.scatter ( reg.predict(X_train), y_train, alpha=0.5 )
    ax.set_xlabel ( "prediction" )
    ax.set_ylabel ( "ground truth" )
    ax.set_title ( "max_depth="+str(max_depth) )
plt.tight_layout()
```



Frage: Wie viele Werte kann das Modell annehmen?

Bei Baumtiefe  $k$  können  $2^k$  Werte angenommen werden.





## 3.4 Algorithmus des Entscheidungsbaums für eine Regression

Modifikation des CART-Algorithmus: <https://scikit-learn.org/stable/modules/tree.html#tree-mathematical-formulation>; siehe auch ID3, C4.5 und C5.0 für alternative Implementationen.

1. Für jedes Feature  $j \in \{1, \dots, n\}$  und jeden möglichen Splitwert  $t$ , zerlege die Daten in Mengen  $Q^{\text{left}}(j, t)$  und  $Q^{\text{right}}(j, t)$ , definiert via

$$Q^{\text{left}}(j, t) = \{(x, y) \in Q \mid x_j \leq t\}$$

$$Q^{\text{right}}(j, t) = Q \setminus Q^{\text{left}}(j, t)$$

2. Die Qualität eines Splits  $\theta = (j, t)$  wird bewertet durch eine Impurity-Funktion  $H$  (bei einer Regression:  $H=\text{squared\_error}$ ):

$$\text{gemittelte Impurity nach dem Split: } G(Q, \theta) = \frac{N^{\text{left}}}{N} H(Q^{\text{left}}(\theta)) + \frac{N^{\text{right}}}{N} H(Q^{\text{right}}(\theta))$$

Hierbei ist  $N = |Q|$  und  $N^{\text{left}} = |Q^{\text{left}}(\theta)|$ ,  $N^{\text{right}} = |Q^{\text{right}}(\theta)|$ .

3. Wähle den Split  $\theta^*$ , der die niedrigste Impurity  $G(Q, \theta)$  erzeugt:  $\theta^* = \operatorname{argmin}_{\theta} G(Q, \theta)$ .

4. Fahre rekursiv fort auf  $Q^{\text{left}}(\theta^*)$  und  $Q^{\text{right}}(\theta^*)$ , bis eine geeignete Abbruchbedingung erreicht ist: "until max\_depth is reached,  $N < \text{min\_samples}$  or  $N = 1$ ".

**Impurity-Funktion für die Regression:** Wir betrachten wie oben eine Teilmenge  $Q$  der Daten.

1. Berechne den Mittelwert der Zielgröße  $\bar{y}_Q = \frac{1}{|Q|} \sum_{(x,y) \in Q} y$

2. Die Impurity ist die gemittelte quadratische Abweichung vom Mittelwert (=Varianz<sup>\*</sup>):

$$H(Q) = \frac{1}{|Q|} \sum_{(x,y) \in Q} (y - \bar{y}_Q)^2$$

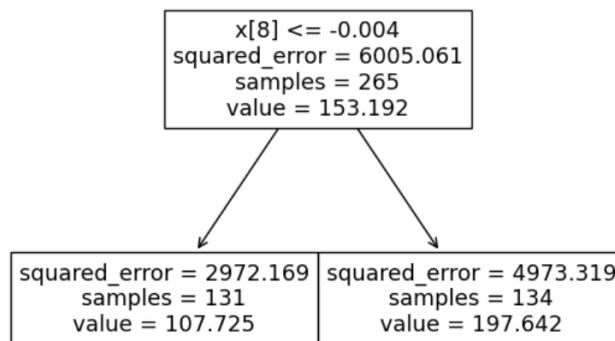
\* ... bis auf Verwendung des Normierungsfaktors  $1/|Q|$  statt  $1/(|Q|-1)$ , der üblicherweise für die empirische Varianz verwendet wird.





### 3.5 Vorgehen für die Inference-Phase (Vorhersage des kontinuierlichen Zielwerts auf neuen Daten)

```
In [117]: # Trainiere einen Entscheidungsbaum  
# auf den Regressions-Task, die kontinuierliche  
# Zielgröße "target" vorherzusagen  
from sklearn.tree import DecisionTreeRegressor  
reg = DecisionTreeRegressor( max_depth=1 )  
reg.fit ( X_train, y_train )  
  
from sklearn.tree import plot_tree  
plot_tree ( reg );
```



Vorgehen für neue Daten:

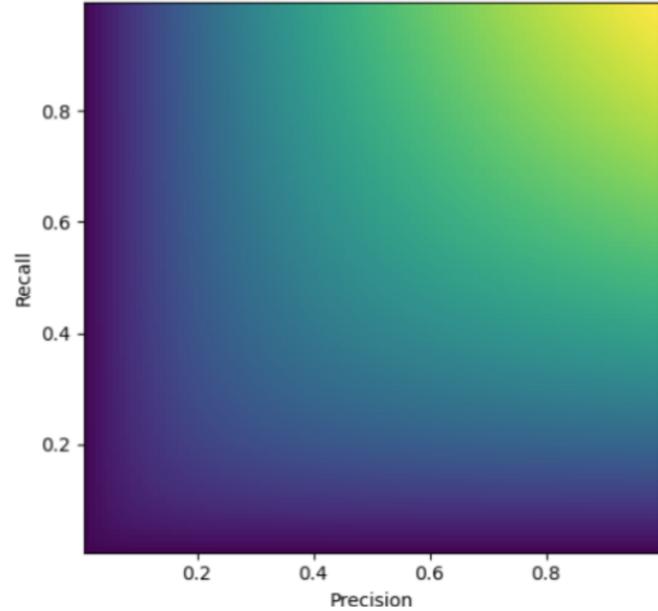
- Während des Trainings wird die Trainingsmenge in kleinere Teile unterteilt entsprechend der Baumstruktur.
- Für jedes Blatt im Baum wird der Mittelwert über die kontinuierliche Zielgröße berechnet ("value" in der Grafik links).
- Dieser Mittelwert des betreffenden Blatts wird als Vorhersagewert verwendet, wenn der neue Datenpunkt in diesem Blatt landet. (Daher gibt es max.  $2^{\text{Baumtiefe}}$  Vorhersagewerte.)

$$\text{prediction}(x_{\text{new}}) = \frac{1}{n_{\text{Blatt}(x_{\text{new}})}} \sum_{i \in \text{Blatt}(x_{\text{new}})} y_i$$

Frage: Wie messen wir die Qualität der Vorhersage?

Wir errechnen den "Abstand" der Vorhersage von der Ground Truth. (RMSE oder  $R^2$ )





1. Objektorientierung in Python
2. Klassifikationsmetriken
3. Regression im Entscheidungsbaum

**Vielen Dank für Ihre Aufmerksamkeit!**

