Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

2024 FS CAS PML - Supervised Learning
2 Klassifikation
2.2 Regelbasiert

Werner Dähler 2024

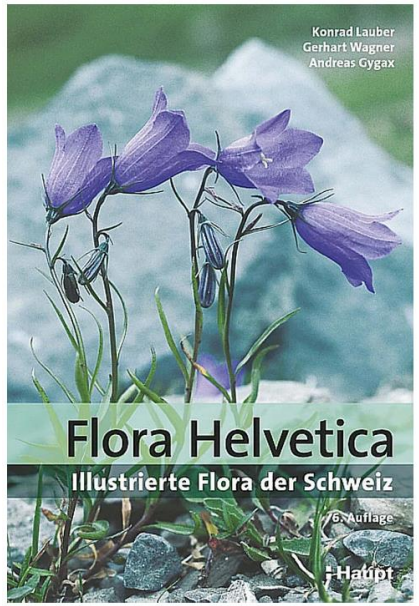
2 Klassifikation - AGENDA

21. Instanzbasierte Modelle
22. Regelbasierte Modelle
    - 221. DecisionTreeClassifier
    - 222. RandomForestClassifier
    - 223. AdaBoostClassifier
    - 224. GradientBoostingClassifier
    - 225. Modellvergleiche
23. Mathematische Modelle
24. Neuronale Netze
25. Multiklass Klassifikation

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.1 Theorie

- typische Anwendung von Entscheidungsbäumen (ausserhalb ML): Bestimmungsschlüssel
- zur Illustration ein Ausschnitt aus dem Bestimmungsschlüssel der Flora Helvetica von Lauber und Wagner [2018] zur Bestimmung der Arten der Gattung Iris in der Schweiz:



Gattung *Iris* Schwertlilie

1	Äussere Perigonb. innen bärtig.	
–	Äussere Perigonb. innen nicht bärtig.	
2	St. 1–2blütig.	*I. x lutescens* Nr. 292
–	St. mehrblütig.	
3	Innere Perigonb. reingelb.	*I.variegata* Nr. 292
–	Innere Perigonb. nicht reingelb.	
4	Äussere Perigonb. nur am Grunde mit dunklen Adern.	*I. x germanica* Nr. 292
–	Äussere Perigonb. bis zum Rande mit dunklen Adern.	
5	Innere Perigonb. graublau bis violett.	*I. x sambucina* Nr. 292
–	Innere Perigonb. gelblich, violett überlaufen.	*I. x squalens* Nr. 292
6	(1) B. schwertfg., 1–3 cm breit.	
–	B. lineal, höchstens 1 cm breit.	
7	Blüten gelb.	*I.pseudacorus* Nr. 292
–	Äussere Perigonb. blau bis rötlich.	*I.foetidissima* Nr. 292
8	St. rund. B. höchstens so lang wie der St.	*I.sibirica* Nr. 292
–	St. kantig. B. viel länger als der St.	*I.graminea* Nr. 292

- dabei wird jeweils aufgrund eines einzelnen Merkmals eine binäre Entscheidung getroffen, um die Menge an möglichen Antworten möglichst schnell (und zuverlässig) einzuschränken

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.1 Theorie

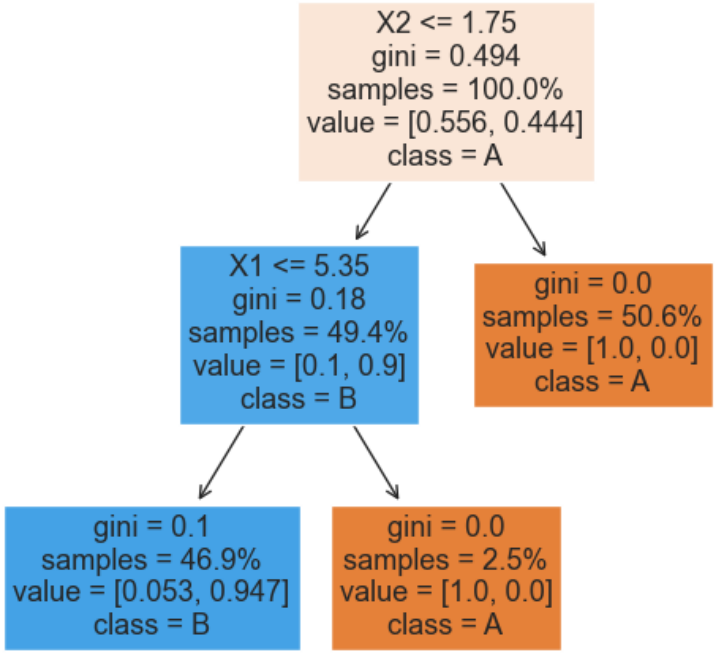
- die ermittelten Regeln (resp. Fragen) können entweder als Regelset oder als Entscheidungsbaum dargestellt werden (hier beispielhaft an Demo-Dataset Klassifikation)

Beispiel als Regelset

Regel Nr.	Frage	Antwort	Entscheid
1	X2 <= 1.75	ja	→ Regel 2
		nein	A
2	X1 <= 5.35	ja	B
		nein	A

- Konvention zur Richtung der Verzweigung
  - Antwort ja: nach links
  - Antwort nein: nach rechts

Beispiel als Entscheidungsbaum



2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.1 Theorie

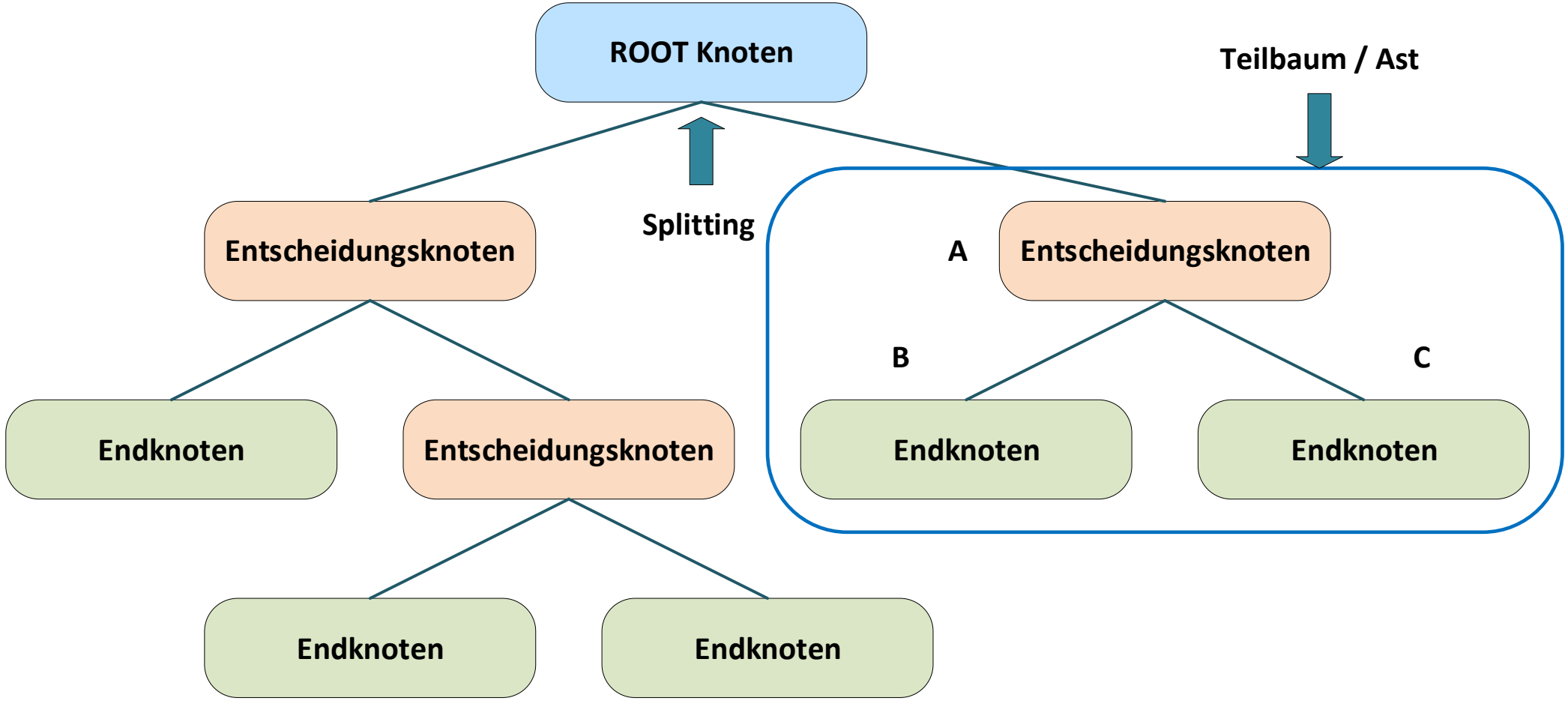
- jede Regel (Frage / Split) bezieht sich auf genau **ein** Feature
- die Antwort ist generell binär, d.h. ja oder nein (true oder false, 1 oder 0)
- die Antwort jeder Regel verweist auf
  - Blatt / Endknoten (farbig), oder
  - eine nächste Frage (Regel)

- zu den weiteren Informationen in den Knoten des dargestellten Baumes (gini, samples, value, class), vgl. Kap. 2.2.1.2)

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.1 Theorie

- einige Begriffe



2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.1 Theorie

ROOT Knoten	Wurzelknoten, umfasst alle Instanzen des Trainingsset
Splitting	Regel, welche die Gesamtmenge der Instanzen in zwei Teilmengen unterteilt
Entscheidungsknoten	Knoten, welcher zu einem weiteren Splitting führt
Endknoten	Knoten, welcher ein Resultat zurückgibt
Teilbaum / Ast	Ast (Teilbaum), welcher durch Splitting entsteht und zu einem weiteren Entscheidungsknoten führt

(A wird als Elternknoten von B und C bezeichnet, resp. B und C als Kindknoten von A)

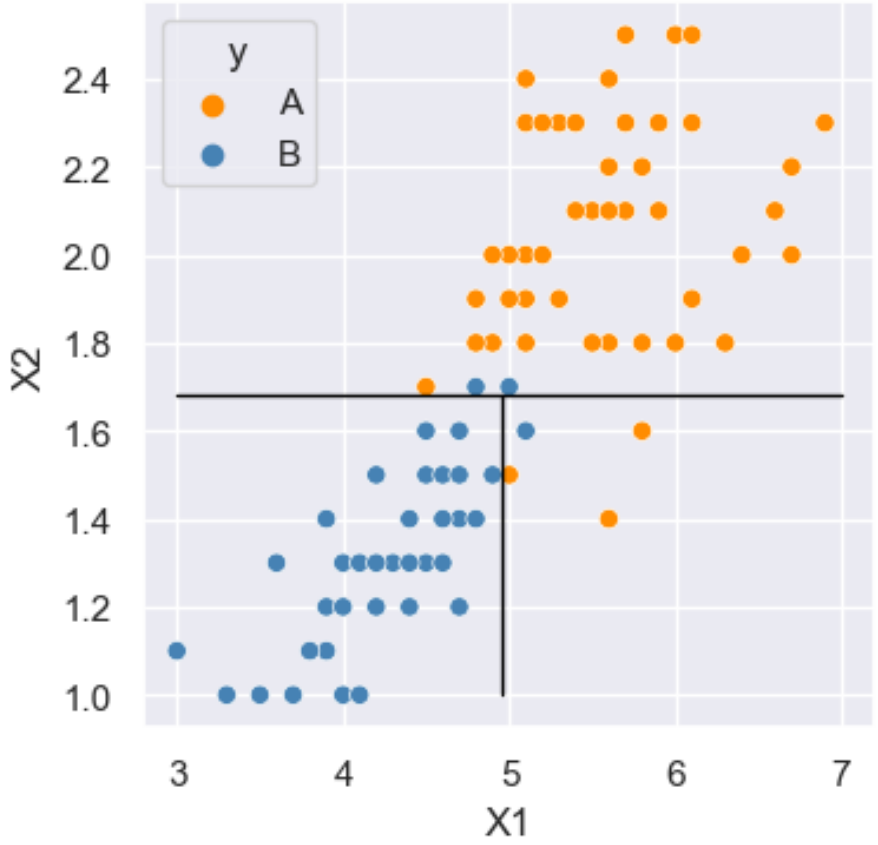
- der komplette Aufbau des Baums erfolgt **rekursiv** durch weiteres analoges splitten aller entstandenen Teilbäume

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.1 Theorie

- Suche nach dem ersten Split
- Beispiel Dataset (demo_data_class.csv)
  (vgl. 2.1.1.1)

- Kandidaten für ersten Split (von Auge abgeschätzt)
  - X1 ≈ 5.0
  - X2 ≈ 1.7



2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.1 Theorie

- Vorgehen: für alle (hier beide) Features wird unabhängig von den anderen die optimale Position für den Split gesucht
- die Split-Positionen aller (beider) Features werden anschliessend verglichen und die beste ausgewählt, d.h. die Datenmenge in entsprechend dem Split in zwei Teilmengen unterteilt
- in den Teilmengen wird wiederum der nächste Split identifiziert (gleiches Verfahren)
  - bis alle Instanzen eindeutig zugeordnet sind
  - oder ein anderes Abbruchkriterium erreicht ist

- dabei stellen sich folgende Fragen
  - welche Split-Positionen sollen untersucht werden?
  - wie kann ein Split quantitativ beurteilt werden?
  - wozu überhaupt ein Abbruchkriterium?
- diesen Fragen soll im Folgenden etwas vertiefter nachgegangen werden

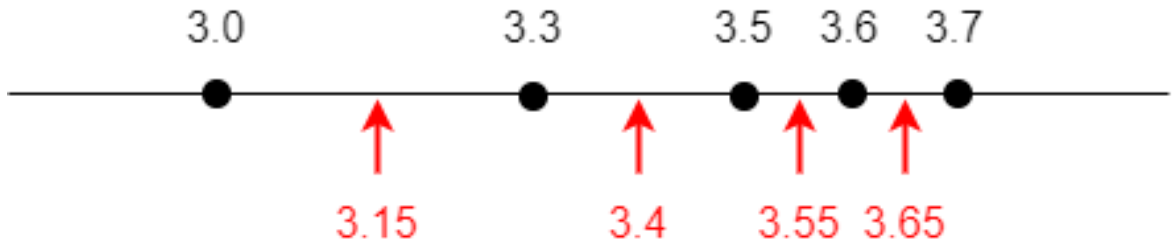
2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.2 Splitting Kriterien

- zu untersuchende Split-Positionen:
- np.unique() gibt einen sortierten Array der Einzelwerte eines Arrays zurück
- mit einem Loop über alle Elemente dieses Arrays, ohne dem letzten, können die Mittelwerte aller benachbarten Einzelwerte als potenzielle Split-Kandidaten ermittelt

```
idx = np.unique(X_demo.X1)
for i in range(len(idx) - 1):
    print('%2i %4.1f %4.1f %5.2f' % (
        i, idx[i], idx[i+1], (idx[i] + idx[i+1]) / 2))
```

```
 0  3.0  3.3  3.15
 1  3.3  3.5  3.40
 2  3.5  3.6  3.55
 3  3.6  3.7  3.65
 4  3.7  3.8  3.75
 :
```



2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.2 Splitting Kriterien

- ein Split im Demo Dataset an der Position X1 = 5 (oder knapp darunter) führt bei der Verteilung der Gruppen (y) zu folgenden Mengenverhältnissen:

```
child_l = y_demo[X_demo.X1 < 5]
child_r = y_demo[X_demo.X1 >= 5]
print(pd.Series(child_l).value_counts(sort = False))
print(pd.Series(child_r).value_counts(sort = False))
```

```
child_l
A     5
B    34
```

```
cild_r
A    40
B     2
```

- ein Mass, welchem wir bereits begegnet sind, ist accuracy_score, d.h. die Anzahl korrekt zugeordneter gegenüber allen Instanzen, in diesem Zahlenbeispiel

```
pred = np.where(X_demo['X1'] >= 5, 'A', 'B')
from sklearn.metrics import accuracy_score
accuracy_score(pred, y_demo)
```

```
0.9135802469135802
```

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.2 Splitting Kriterien

- in der Literatur werden verschiedene Splitting Kriterien diskutiert:
  - Gini Index (wird bei DecisionTreeClassifier als Default verwendet)
  - Information Gain (entropy, kann optional auch bei DecisionTreeClassifier eingesetzt werden)
  - 1R algorithmus (vergleichbar mit accuracy_score, quantifiziert die Fehlerrate (bei den Klassifikatoren von scikit-learn aber nicht direkt verfügbar, aber vgl. mlxtend)

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.2 Splitting Kriterien

Gini Index

- "Corrado Gini... entwickelte unter anderem den nach ihm benannten Gini-Koeffizienten, mit dem er die Ungleichverteilung der Einkommen in einer Volkswirtschaft darstellte..."
- dabei wird die "Reinheit" resp. Homogenität von Mengen untersucht und beurteilt, wie stark ein Split die Gesamthomogenität eines Knotens nach dem Aufteilen in zwei Untermengen verbessert
- Gini formal (zwei Gruppen):

  wobei

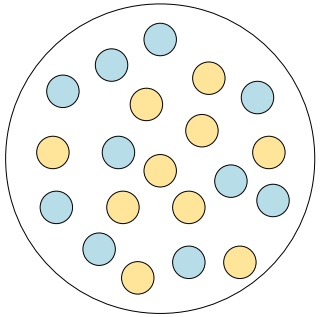
$$I_g = p_1(1 - p_1) + p_2(1 - p_2) = 2 * p_1 * p_2$$

  - p_1 : relativer Anteil der ersten Gruppe
  - p_2 : relativer Anteil der zweiten Gruppe
- der Index erreicht sein Minimum (0) wenn alle Beobachtungen in dieselbe Klasse fallen, sein Maximum (0.5) wenn zwei gleich grosse Gruppen vorliegen

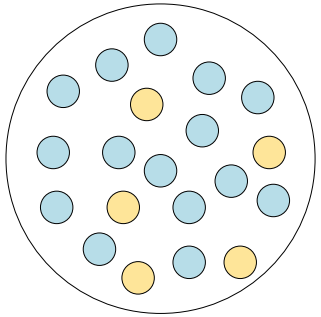
2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.2 Splitting Kriterien

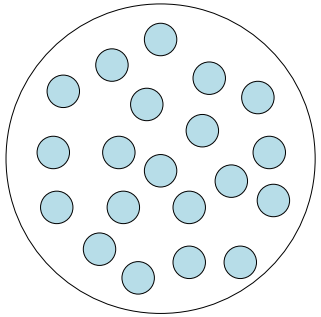
- zur Illustration einige Beispiele (für jeweils zwei Gruppen in einer Menge)



A



B



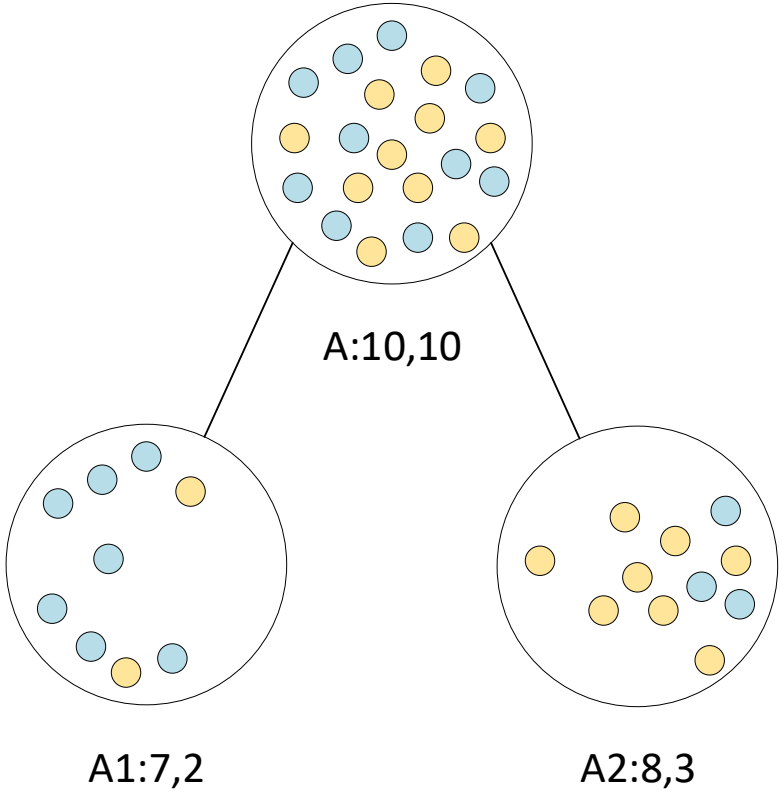
C

	blau	gelb	p_1	p_2	$2*p_1*p_2$
A	10	10	0.50	0.50	0.500
B	15	5	0.75	0.25	0.375
C	20	0	1.00	0.00	0.000

- um die Veränderung des Gini-Koeffizienten durch ein Split zu berechnen wird die Differenz des Koeffizienten vor dem Split und der Summe der gewichteten Koeffizienten nach dem Split berechnet

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.2 Splitting Kriterien



	blau	gelb	p_1	p_2	$2*p_1*p_2$	Gewicht	g(gewichtet)
A	10	10	1/2	1/2	0.5		
A1	7	2	7/9	2/9	0.34568	9/20	0.15556
A2	3	8	3/11	8/11	0.39669	11/20	0.21818
Total							0.37374

- Gini Impurity ist auch für mehr als zwei Gruppen (Kategorien) anwendbar

$$I_g = \sum_{i=1}^{n} p_i(1 - p_i)$$

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.2 Splitting Kriterien

- ImpurityDecrease ist somit ein Mass zur Quantifizierung der "Entmischung" durch den Split,
  im obigen Zahlenbeispiel:

$$ImpurityDecrease = Gini_{parent} - GewichteteSummeGinis_{childeren} = 0.5 - 0.373 = 0.127$$

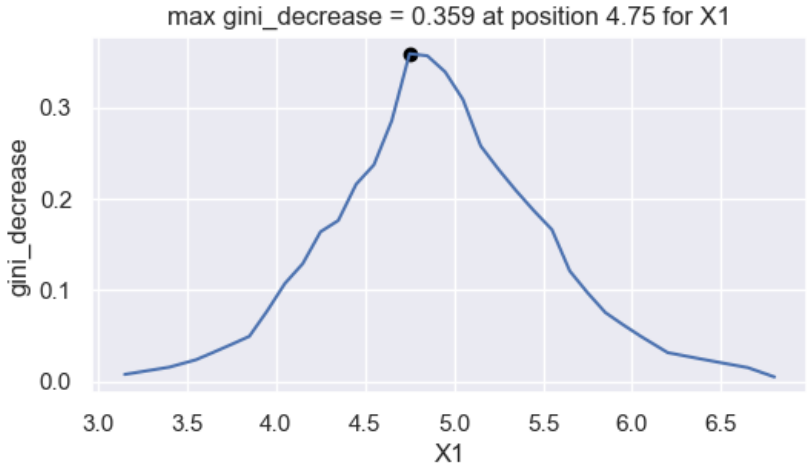
- ausserdem wird dieser Wert noch in Beziehung gesetzt zum Gewicht des fraglichen Elternknotens in Bezug zum ganzen Baum
- wenn also der Elternknoten n_p Instanzen enthält, und der Rootknoten n_r, wird ImpurityDecrease anschliessend mit dem Faktor $\frac{n_p}{n_r}$ multipliziert

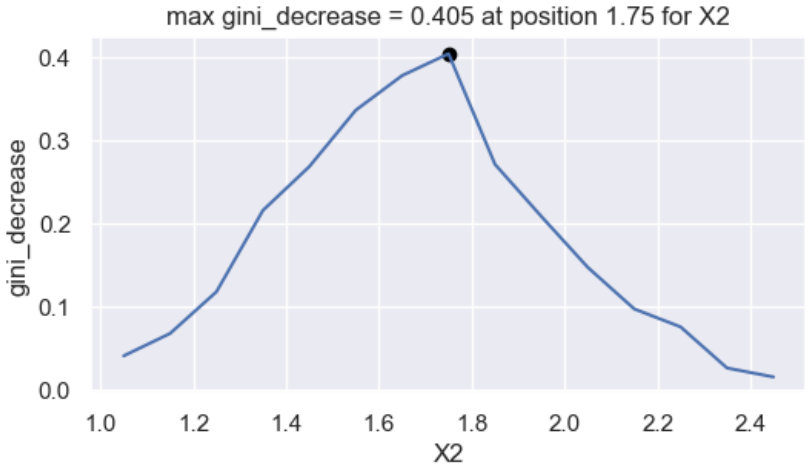
- beim weiteren Aufbau des Baumes wird dieser Wert noch weiter eine wichtige Rolle spielen (vgl. Abbruchkriterien: min_impurity_decrease, Kap. 2.2.1.4.5, sowie zur Bestimmung der Feature Importance, Kap. 2.2.1.7)

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.2 Splitting Kriterien

- Anwendung des ImpurityDecrease auf das Beispieldataset (demo_data_class.csv)
  - im nebenstehenden Beispiel werden die beiden Features X1 und X2 auf die optimale Splitt Positionen untersucht
  - für beide Features wird dabei für alle (potentiellen) Positionen der Gini Index bestimmt und entlang der Werteachse des jeweiligen Features aufgetragen (vgl. extra_get_dt_split_positions.ipynb)





- Fazit:

Feature	Position	ImpurityDecrease	Wahl
X1	4.75	0.359	
X2	1.75	0.405	✓

- als erster Split wird hier die Position 1.75 auf dem Feature X2 identifiziert

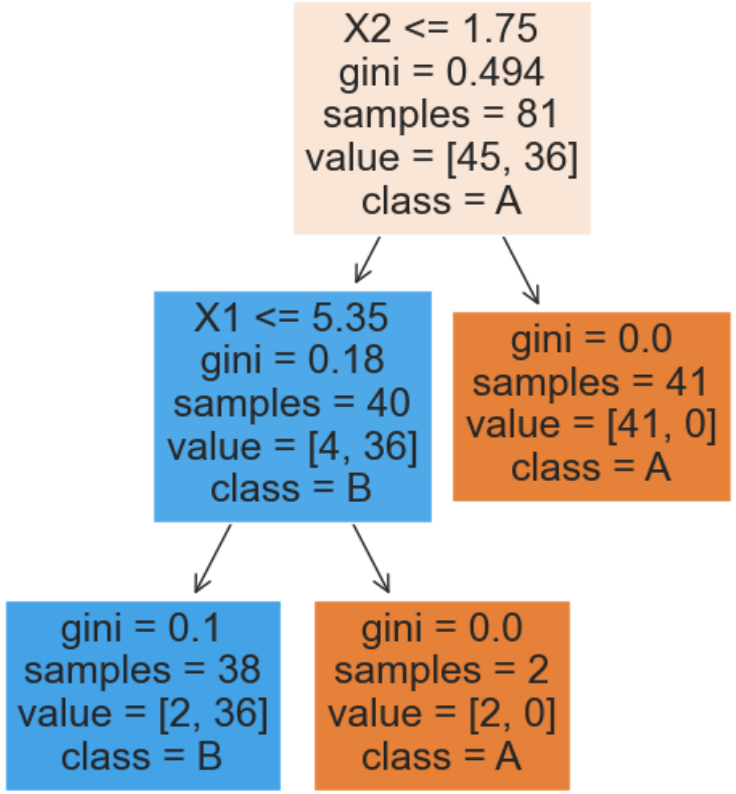
2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.2 Splitting Kriterien

- Kontrolle mit DecisionTreeClassifier
  - trainiert mit Standard Parametern (beinahe)
  - Regeln ausgegeben mit export_text()

```
:
model.fit(X_, y_)
from sklearn.tree import export_text
print(export_text(
    model, feature_names=list(X_.columns)))
```

```
|--- X2 <= 1.75
|   |--- X1 <= 5.35
|   |   |--- class: B
|   |--- X1 >  5.35
|   |   |--- class: A
|--- X2 >  1.75
|   |--- class: A
```



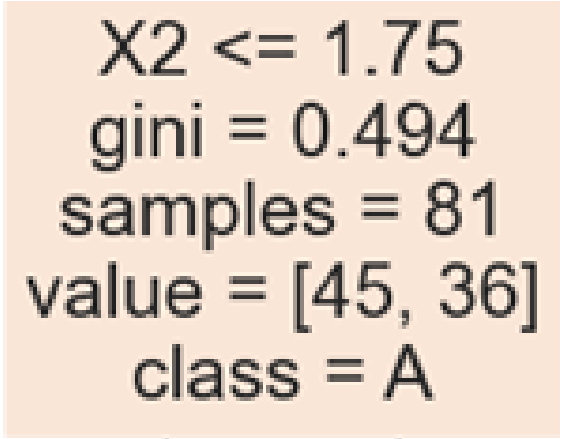
2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.2 Splitting Kriterien

- zu den Informationen in den einzelnen Knoten des oben dargestellten Baums:

  - X2 ≤ ...: Regel für den ermittelten Split auf dem Knoten (Feature und Bedingung)
  - gini: Gini Index für den Knoten (vor dem Split)
  - samples: Anzahl Beobachtungen total
  - value: Anzahl Beobachtungen nach Klassen
  - class: vorhergesagte Klasse (majority: Modalwert)

  - die Wahrscheinlichkeit für die jeweilige Voraussage der Knoten wird ausserdem durch Farbtöne angezeigt



2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.2 Splitting Kriterien

- als alternatives Kriterium steht für den Parameter criterion auch noch "entropy" zur Verfügung
- Entropie ist ein Mass aus der Informationstheorie und geht zurück auf Claude Shannon
- wie Gini quantifiziert dieses Mass die "Unordnung" von Gruppenzugehörigkeiten in einer Menge
- Formulierung für 2 Klassen (links) und n Klassen (rechts):

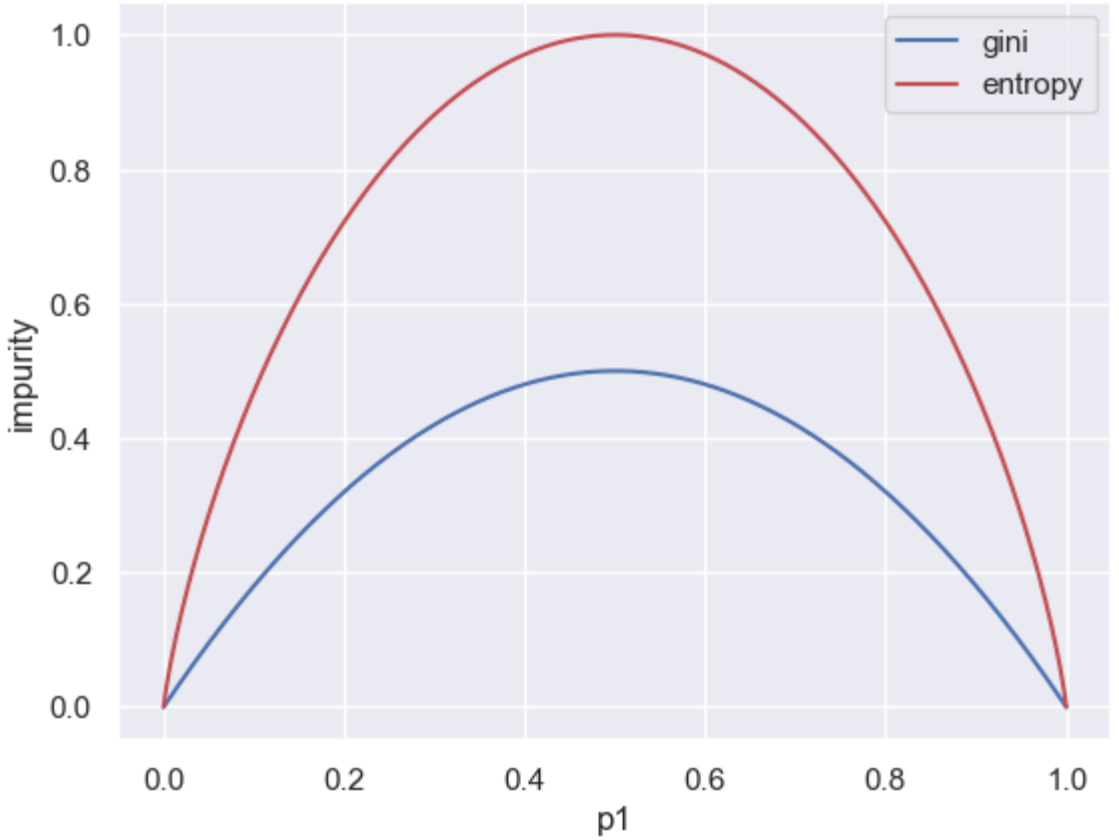
$$I_e = -\, p_1 \cdot log_2(p_1) - p_2 \cdot log_2(p_2) \qquad I_e = -\sum p_i \cdot log_2(p_i)$$

- es wird der Logarithmus zur Basis 2 verwendet
- da p_i immer kleiner als 1 sind, resultieren negative Logarithmen, daher das ganze am Ende negieren (was wäre vorzukehren, wenn ein p-Wert 0 ist?)
- wegen dem Logarithmieren ist dieses Mass etwas zeitaufwändiger als Gini
- eine vertieftere Diskussion zu den Unterschieden zwischen Gini Info Gain und Entropy Info Gain findet sich z.B. in Towards Data Science

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.2 Splitting Kriterien

- eine Gegenüberstellung des Verhaltens von Gini-Index und Entropie für zwei-Klassen Fragestellungen
  - Minimum für p1 = 0 oder p2 = 0
  - Maximum für p1 = p2 = 0.5
- einziger Unterschied, das Maximum
  - Gini-Index: 0.5
  - Entropie: 1.0
- beide Kriterien können auch für Multiklass-Fragestellungen eingesetzt werden



2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.3 Praxis

- Voraussetzungen
  - die mittels Feature Engineering aufbereiteten Daten sind geladen
  - ausserdem
    - features - target - split
    - train - test - split
  - dazu Import und Aufruf der in Modul bfh_cas_plm hinterlegten Funktion prep_data()

```
from bfh_cas_pml import prep_data
X_train, X_test, y_train, y_test = prep_data(
    'bank_data_prep.csv', 'y', seed = 1234)
```

- dieser Vorbereitungsschritt wird im Folgenden bei allen Praxisbeispielen eingesetzt und daher in der Präsentation nicht mehr wiederholt,
  vgl. Kap. 2.1.3

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.3 Praxis

- Modell instanziieren, definieren und trainieren

```
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier(random_state=1234)
model.fit(X_train, y_train)
```

  random_state ist optional, wird im Folgenden aber immer gesetzt zwecks Reproduzierbarkeit der Ergebnissen gesetzt

- anwenden des trainierten Models auf die Trainingsdaten

```
y_pred = model.predict(X_test)
```

- bewerten des trainierten Modells anhand der Testdaten, mit modellinternem Scorer (accuracy)

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

```
0.8296318831761484
```

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.3 Praxis

- sowie noch einige weitere Merkmale (interne Informationen) zum trainierten Modell

```
print('depth:', model.get_depth())
print('n_leaves:', model.get_n_leaves())
print('score on train:', model.score(X_train, y_train))
print('score on test:', model.score(X_test, y_test))
```

```
depth: 28
n_leaves: 777
score on train: 1.0
score on test: 0.8296318831761484
```

(!/?)

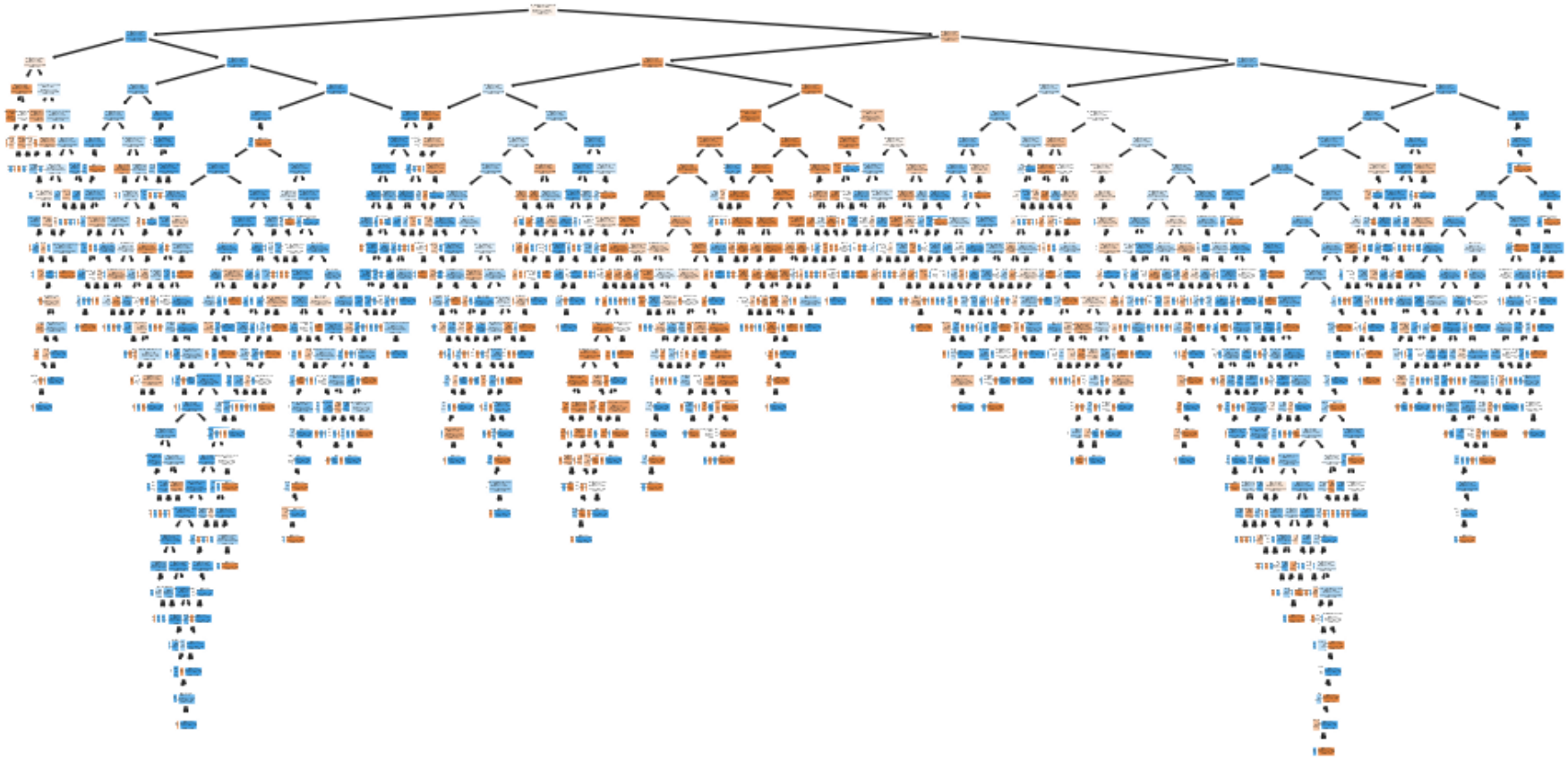
davon später mehr

- Accuracy ist schon klar besser als bei der vorigen Methode

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.3 Praxis

- der trainierte Baum hat eine Tiefe von 28 und 777 Endknoten, und würde zu einer Visualisierung wie unten führen (was aber kaum mehr interpretierbar ist)



2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.3 Praxis

- die Endknoten sind alle rein (score on train = 1.0 im Gegensatz zu 0.830 auf test)

- der Baum ist voll ausgebildet und damit überbestimmt (overfitted, mehr dazu später in Kap. 2.2.1.6)
- voll ausgebildete Bäume widerspiegeln ausschliesslich das Verhalten der Trainingsdaten
- sind für die Voraussage von neuen Daten (Testdaten) zu wenig generalisiert
- Abhilfe: Beschneiden des Baums → "Pruning" (vgl. nächstes Kapitel)

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.4 Abbruchkriterien

- dazu zwei mögliche Strategien
  - Postpruning: der Baum wird vollständig aufgebaut, und danach so lange zurückgeschnitten (pruned), bis sich das Lernverhalten merklich verschlechtert
  - Präpruning: bereits beim Aufbau wird bei jedem Split geprüft, ob dieser noch zu einer wesentlichen Verbesserung des Baumes führt
- Postpruning wird heute kaum mehr verwendet, da viel rechenintensiver
- im Folgenden werden einige Parameter vorgestellt, welche bei diesem Learner zu einem sogenannten "early stop" führen
  - max_depth
  - min_samples_split
  - min_samples_leaf
  - max_leaf_nodes
  - min_impurity_decrease

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.4 Abbruchkriterien

- um im Folgenden die Auswirkungen einzelner Parameterwerte auf die entstehenden Bäume beurteilen zu können, wird im Modul `bfh_cas_pml.py` eine Funktion definiert welche
  - interne Informationen zurückgibt
  - den trainierten Baum visualisiert

```
def inspect_decision_tree_model(model_def, features, target, figsize=(6, 6)):
    :
```

- sie nimmt als Parameter entgegen:
  - Modelldefinition (parametrisiertes Objekt)
  - die Feature Matrix (train)
  - den Target Vektor (train)
  - ausserdem optional die Grösse der zu erstellende Visualisierung

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.4 Abbruchkriterien

- nach dem Import ...

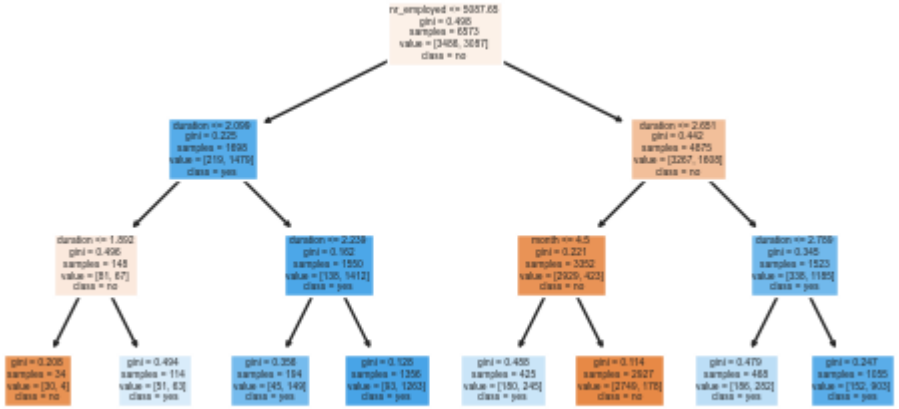
```
from bfh_cas_pml import inspect_decision_tree_model
```

- kann die Funktion wie folgt aufgerufen werden (hier exemplarisch)

```
inspect_decision_tree_model (
    DecisionTreeClassifier(max_depth = 3), X_train, y_train, figsize = (6, 3))
```

- was zu folgenden Ergebnissen führt

```
TREE DIAGNOSTICS:
depth  : 3
leaves : 8
score  : 0.8647497337593184
```

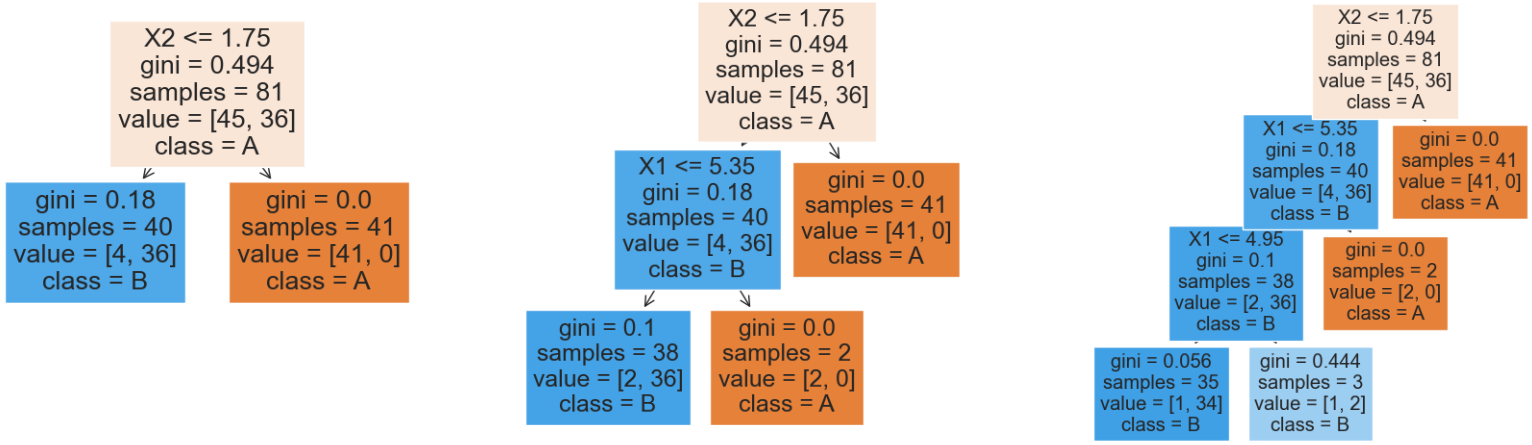


2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.4.1 Abbruchkriterien - max_depth

- maximale Tiefe des zu bildenden Baumes
  - ein Baum der Tiefe 0 besteht nur aus dem Root Knoten
  - ein Baum der Tiefe 1 aus einem einzigen Split, usw.

  - nebenstehend drei Bäume auf den Demodaten mit Tiefen 1, 2, 3



- es findet kein weiterer internen Test statt
- default = None (keine Beschränkung in dieser Hinsicht)

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.4.1 Abbruchkriterien - max_depth

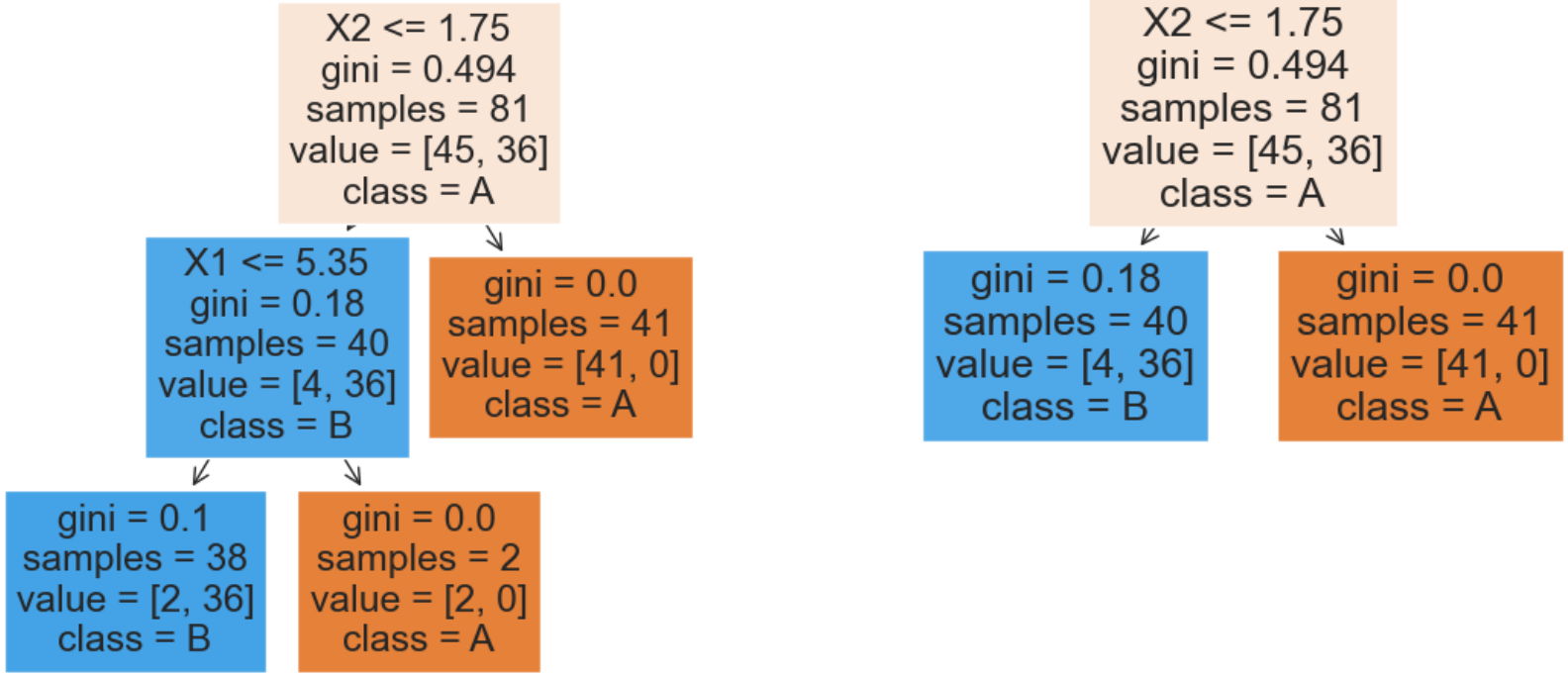
- eine Schwäche zeigt sich allerdings bereits bei Verwendung dieses Parameters beim praxisnahen Dataset (Bankkunden)

- das strikte Beschneiden des Baums auf eine einheitliche maximale Tiefe hindert Äste (Teilbäume) daran, weiter ausgebildet zu werden, auch wenn dort noch Potential vorhanden sein könnte

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.4.2 Abbruchkriterien - min_samples_split

- die minimale Anzahl Beobachtungen, welche ein Knoten aufweisen muss, um überhaupt noch weiter gesplittet zu werden
- im untenstehenden Beispiel wird bei einem Wert von 40 der Kind-Knoten links noch gesplittet (links), bei 41 dagegen nicht mehr (rechts)
- default: 2



2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.4.3 Abbruchkriterien - min_samples_leaf

- die minimale Anzahl Beobachtungen für jeden Endknoten
- im Gegensatz zu min_samples_split (s.o.) muss dieser Split provisorisch durchgeführt werden, um das Ergebnis zu beurteilen
- ist die Bedingung nicht erfüllt (mindestens einer der Kindknoten hat weniger Beobachtungen) wird der Split zurückgenommen
- default = 1
- vgl. [ipynb]

2.2.1.4.4 Abbruchkriterien - max_leaf_nodes

- maximale Anzahl Endknoten
- default: None, d.h. keine Beschränkung in diesem Sinne
- vgl. [ipynb]

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.4.5 Abbruchkriterien - min_impurity_decrease

- *minimale Verminderung der Unreinheit des gesamten Baums durch erfolgten Split auf dem jeweiligen Knoten*
- defalt: 0

- Vorgehen dazu:
  1) berechnen von Impurity (Gini-Index, vgl. Kap. 2.2.1.2) vor dem Split
  2) berechnen der gewichteten (!) Impurity der beiden Kindknoten nach dem Split
  3) berechnen der Differenz zwischen 1) und 2) (diese muss kleiner werden)
  4) gewichten der Differenz aus 3) gemäss dem relativen Anteil des untersuchten Eltern-Knotens in Bezug auf den ganzen Baum

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.4.5 Abbruchkriterien - min_impurity_decrease

- formal:

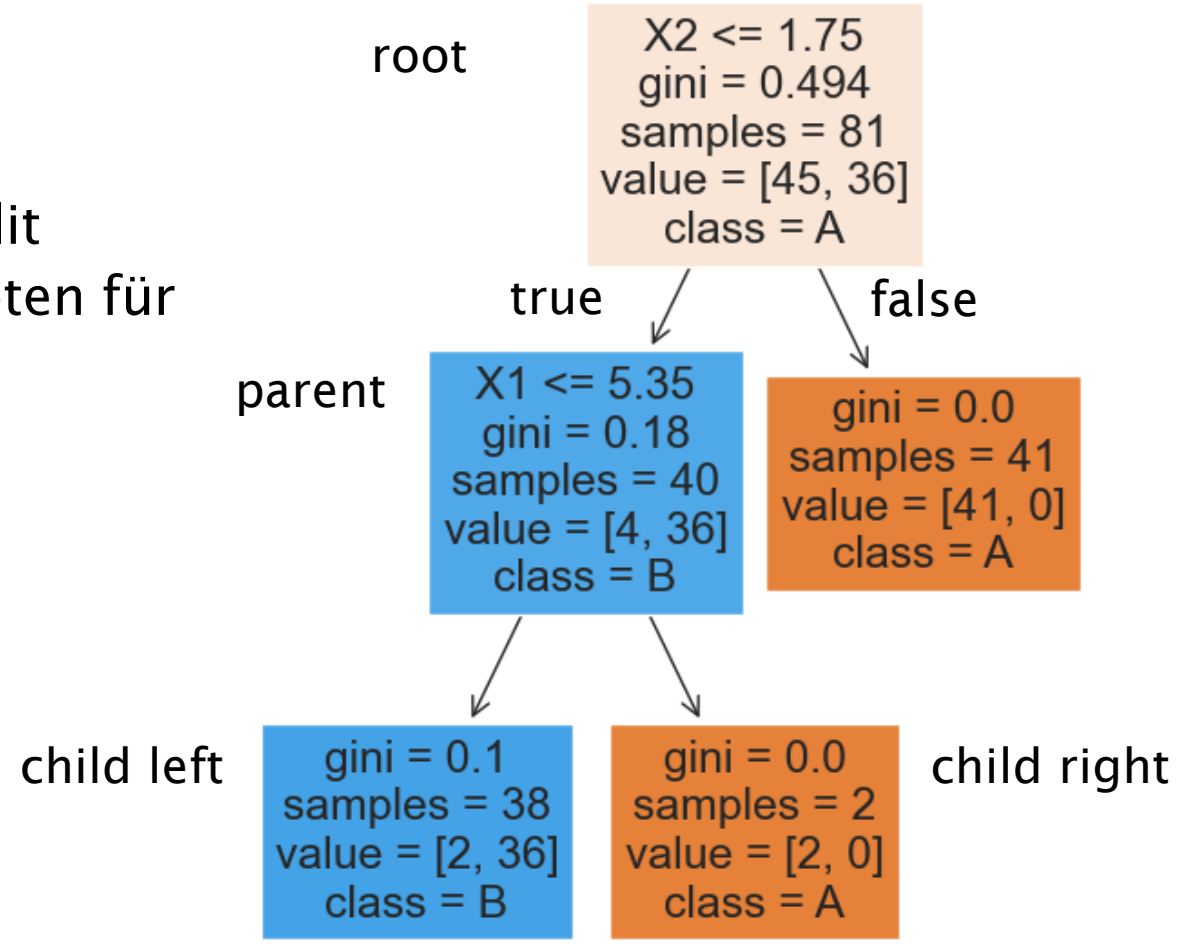
$$impurity\_decrease = \frac{N_p}{N_r} \cdot \left(I_p - \frac{N_{cl}}{N_p} \cdot I_{cl} - \frac{N_{cr}}{N_p} \cdot I_{cr} \right)$$

- wobei
  N : Anzahl Beobachtungen im jeweiligen Knoten
  I : Impurity im jeweiligen Knoten
  und die Indices
  r: Root
  p: Parent
  cl: Child left
  cr: Child right
- Impurity kann dabei je nach Parametrisierung eine der beiden Funktionen sein:
  - gini (default)
  - entropy

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.4.5 Abbruchkriterien - min_impurity_decrease

- Rechenbeispiel auf oberstem Kindknoten links im obigen Baum:
- dabei bedeuten:
  - **Root**: Wurzelknoten des gesamten Baums
  - **Parent**: Untersuchter Knoten für diesen Split
  - **Child left, Child right**: die beiden Kindknoten für den untersuchten Split



2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.4.5 Abbruchkriterien - min_impurity_decrease

- Vorgehen Schritt für Schritt
  1. berechnen des Gini-Index für den Parent-Knoten: 2∗4/40∗36/40=0.18, (kann auch direkt abgelesen werden, vgl. auch Kap. 2.2.1.2)
  2. berechnen des gewichteten Gini-Index für die beiden Kindknoten, dabei werden die Indices unabhängig bestimmt und mit dem Gewicht (relativer Anteil der betroffenen Beobachtungen) addiert
  3. berechnen der Differenz zwischen 1. und 2.
  4. gewichten dieser Differenz in Bezug auf den ganzen Baum, d.h. korrigieren mit dem Verhältnis der Beobachtungen im Parent-Knoten gegenüber den Beobachtungen im ganzen Baum (Root)

  das ergibt für dieses Beispiel einen Wert von 0.0421
- für Visualisierungen: vgl. [ipynb]
- zum Finden von optimalen Parameterwerten: vgl. Kap. 2.2.1.5 Parameter Tuning

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.5 Parameter Tuning

- die oben eingesetzten Parameterwerte sind willkürlich
- durch systematisches Austesten kann der optimale Wert ermittelt werden (vgl. auch Kap. 2.1.1.3, Parameter Tuning bei KNeighborsClassifier)
- dazu wieder etwas Code:

```
## prepare loop
model = DecisionTreeClassifier()
scores = [] ## empty list for collect iteration scorers
params = range(1, 21) ## define range

## iteration over params for
param in params:
    model.set_params(max_depth = param)
    model.fit(X_train, y_train)
    score = model.score(X_test, y_test)
    scores.append(score) ## add score to list
```

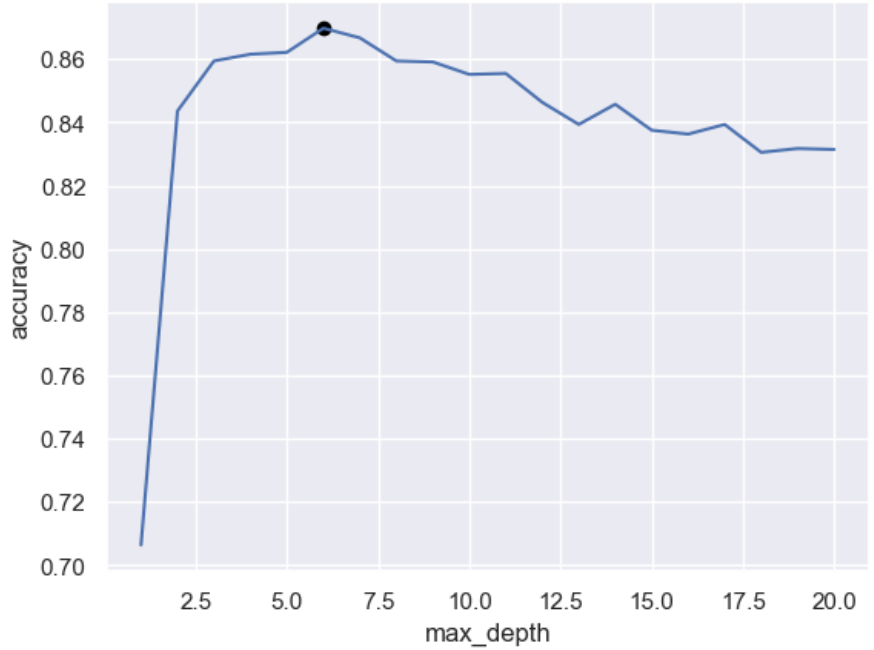
2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.5 Parameter Tuning

- auswerten des Loops

```
fig = sns.lineplot(x=params, y=scores)
plt.scatter(x=params[scores.index(max(scores))], y=max(scores), color="black")
plt.xlabel('max_depth')
plt.ylabel('accuracy');
```

- Fazit
  - Accuracy steigt zu Beginn rasch an, erreicht ihr Maximum und sinkt dann wieder ab



2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.5 Parameter Tuning

- mit dem für die Visualisierung eingesetzten Ausdruck `scores.index(max(scores))` kann ausserdem der Listenindex für den grössten Scorewert ermittelt werden, was es erlaubt, Score und Parameterwert des Maximums in der Konsole auszugeben

```
print('best_param :', params[scores.index(max(scores))])
print('best_score :', max(scores))
```

```
best_param : 6
best_score : 0.8697900821417706
```

- ein Codebeispiel für einen geschachtelten Loop über Bereiche von zwei Parametern findet sich im [ipynb],
  mehr dazu in Kap. 4.3.2 Validierung und mehr - Grid Search mit CV

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.6 Overfitting, ein Ausblick

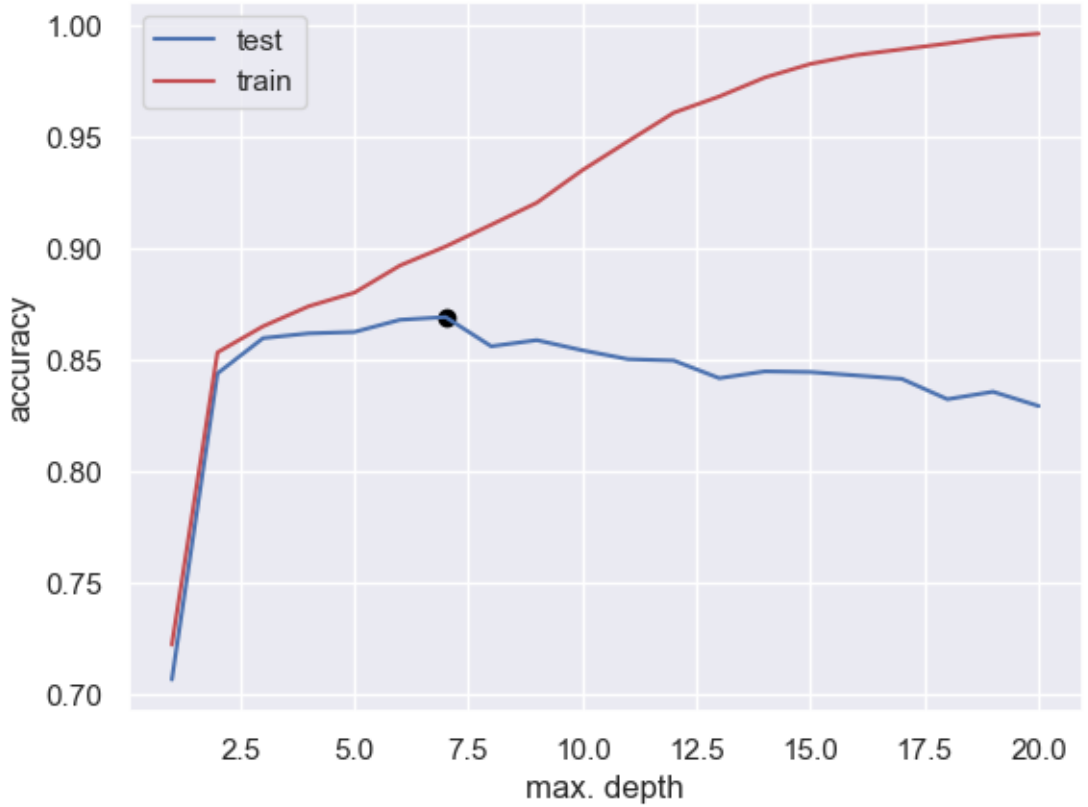
(oder von guten und schlechten Bäumen)

- **schlechtester Baum**:
  - eigentlich kein Baum
  - besteht nur aus Root Knoten (Tiefe = 0, das Ergebnis ist die häufigste Klasse der Trainingsdaten)
  - beste Generalisierung
- **bester Baum** (aber nur auf den ersten Blick):
  - der Baum ist vollständig aufgebaut
    - alle Endknoten sind rein: enthalten generell nur noch Beobachtungen derselben Klasse (mögliche Ausnahmen: sind Situationen denkbar, wo dies nicht zutrifft?)
  - reinster Baum / schlechteste Generalisierung
- **optimaler Baum**:
  - ein Kompromiss zwischen schlechtestem und bestem Baum
    - möglichst "rein" **und** möglichst generalisiert

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.6 Overfitting, ein Ausblick

- ein Beispiel zur Illustration:
  - Learner: DecisionTreeClassifier
  - Tuning Parameter: max_depth von 1 bis 20
  - Performance Metrik: accuracy
  - Trainingsset: X_train, y_train aus Bankkunden Dataset
  - Predict und Test vergleichen
    - mit Trainingsset
    - mit Testset
- Code: vgl. [ipynb]



2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.6 Overfitting, ein Ausblick

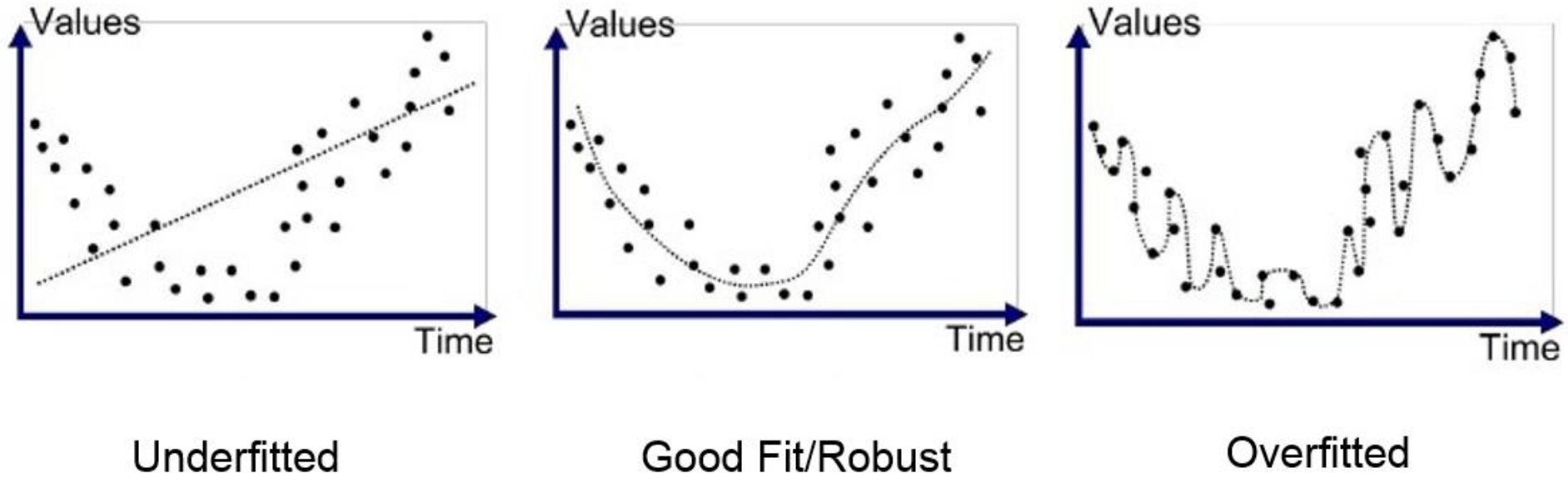
- accuracy in Bezug auf die Trainingsdaten (rot)
  - nimmt kontinuierlich zu
  - erreicht einen Wert von 1 sobald der Baum vollständig aufgebaut ist
- accuracy in Bezug auf Testdaten (blau)
  - nimmt anfangs stark zu
  - erreicht ein Maximum (bei 7) und nimmt dann wieder (leicht) ab

- **Underfitting** (Unteranpassung): Bereich von max_depth, wo beide accuracy Werte deutlich kleiner sind als das Maximum der accuracy auf Training
- **Overfitting** (Überanpassung): Bereich von max_depth, wo die beiden accuracy Werte mehr und mehr auseinanderklaffen
  - liefert einen "Tunnelblick" auf die Trainingsdaten
  - nicht mehr generalisiert

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.6 Overfitting, ein Ausblick

- kann auch bei Regressionsmodellen (kontinuierlich numerisches Target) auftreten



[Quelle]

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.7 Feature Importance

- wie im Rahmen von Feature Engineering (1.5.1.2 Modellbasierte Feature Selection) erwähnt, liefern verschiedene Modellierungstechniken Hilfestellungen, um die Wichtigkeit einzelner Features mit Sicht auf die Modellierung zu beurteilen
- zu diesen gehört auch DecisionTreeClassifier (sowie die meisten anderen Regelbasierten Learner)

- im unten stehenden Beispiel wird ein Modell auf dem vollständigen Bankkunden Dataset (also ohne Train - Test - Split) erstellt, mit Default Parametern

```
model = DecisionTreeClassifier()
model.fit(X_train, y_train)
```

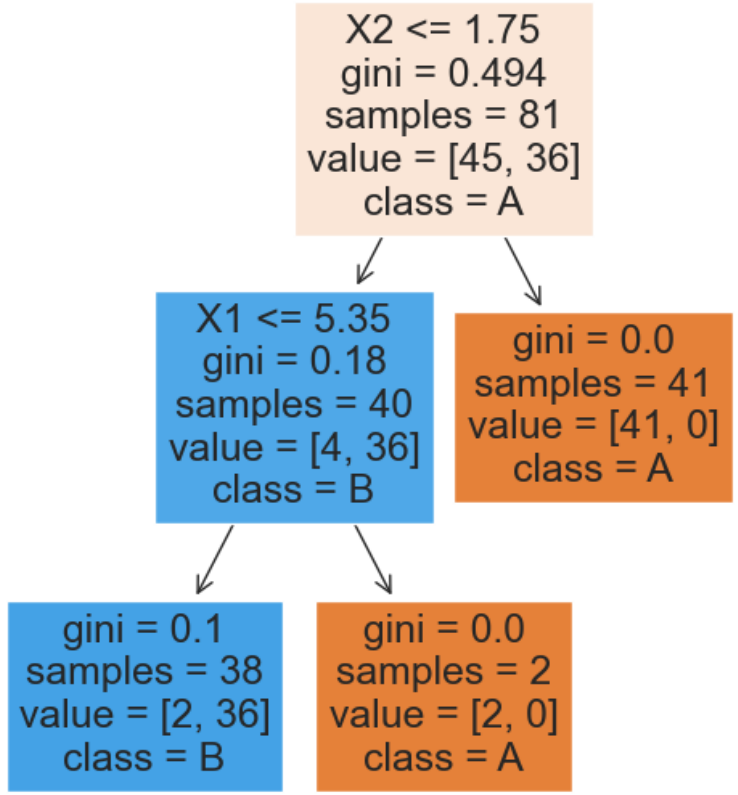
- das trainierte Modell bietet unter andere das Attribut **feature_importances_** an

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.7 Feature Importance

zur Bestimmung von Feature Importance bei DecisionTreeClassifier:

- zur Erinnerung, bei jedem Split werden unter anderem folgende Merkmale verwendet und hinterlegt (!):
  - Split-Regel mit Bedingung auf einem Feature
  - Anzahl Instanzen pro Klasse im entsprechenden Knoten
- aus diesen Informationen können nach Aufbau des Baumes folgende Informationen zu den Wichtigkeiten der einzelnen Features extrahiert werden
  1. Features, welche überhaupt im Baum eine Rolle spielen
  2. Features, welche häufiger im Baum erscheinen
  3. Features, auf welchen Splits für den gesamten Baum bedeutsamer sind als andere



2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.7 Feature Importance

- eine zentrale Rolle bei der Beurteilung von einzelnen Splits spielt bekanntlich impurity_decrease (vgl: Kap. 2.2.1.4.5), unabhängig vom eingesetzten Split-Kriterium
- für den ganzen Baum werden die entsprechenden Werte gesammelt und pro Feature aufsummiert
- abschliessend werden die Summen normiert, d.h. derart linear transformiert, dass deren Gesamtsumme 1 ergibt

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.7 Feature Importance

- Ausgabe von feature_importances_, kombiniert mit den Feature Namen:

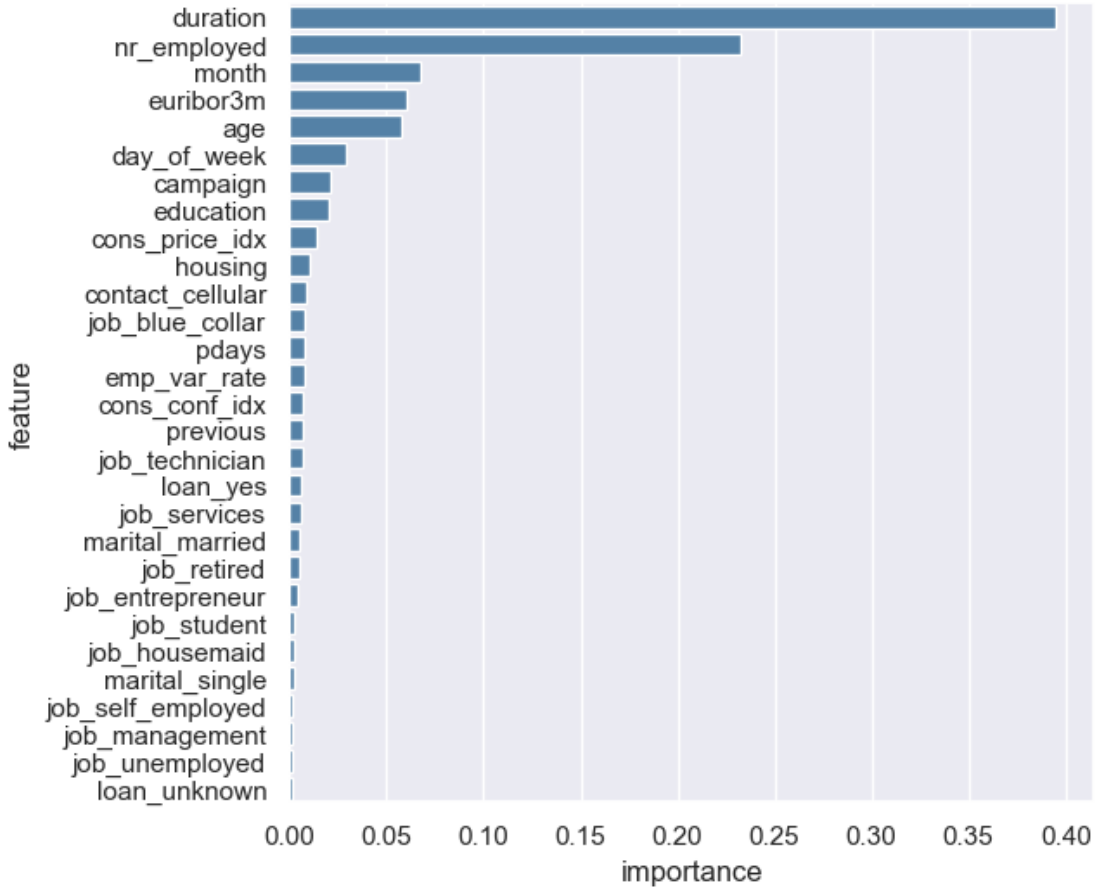
```
print(X_train.columns)
print(model.feature_importances_)
```

```
Index(['age', 'education', 'housing', 'contact_cellular', 'month',
       'day_of_week', 'duration', 'campaign', 'pdays', 'previous',
       'emp_var_rate', 'cons_price_idx', 'cons_conf_idx', 'euribor3m',
:
[0.05733455 0.02018367 0.01058606 0.00849656 0.06758796 0.02891144
 0.39505179 0.02137808 0.00731779 0.00718283 0.00728213 0.0140668
 0.00721146 0.06012356 0.23292918 0.0076316  0.00396511 0.00239168
:
```

2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

2.2.1.7 Feature Importance

- da obiger Output nur schwer zu interpretieren ist, drängt sich eine Visualisierung auf (vgl. [ipynb])
- die "üblichen Verdächtigen" sind hier recht gut erkennbar
- aus den für die nebenstehende Visualisierung aufbereiteten Daten kann relativ einfach ein Filter erstellt werden, um anschliessend die n "besten" Features auszuwählen
- weitergehendes zu automatischer Feature-Selektion wird im nachgeschobenen Kapitel 1.7.2.1.1 diskutiert



2.2.1 Klassifikation - Regelbasiert - DecisionTreeClassifier

Workshop 05

Gruppen zu 2 bis 4, Zeit: 30'

- untersuchen Sie verschiedene Werte von `min_impurity_decrease` bei `DecisionTreeClassifier` auf die erreichbare Performance (Accuracy)
- grenzen Sie dabei den zu untersuchenden Wertebereich schrittweise ein
- stellen Sie dazu die Ergebnisse wie folgt dar
  - grafisch als Liniendiagramm
  - in der Konsole mit bestem Score und entsprechendem Parameterwert

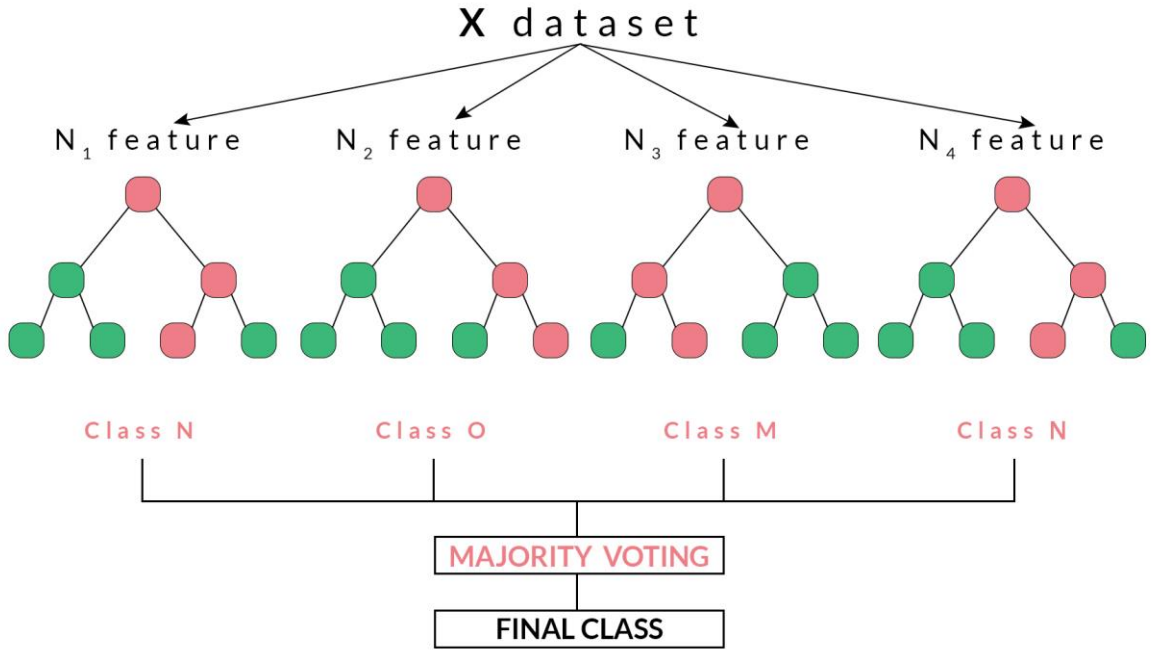
- Hinweis
  - `range()`: erstellt einen Bereich von Ganzzahligen Werten mit identischer Schrittweite
  - `np.arange()`: (Funktion von numpy) erstellt mit analoger Parametrisierung einen Bereich mit Gleitkommawerten

2.2.2 Klassifikation - Regelbasiert - RandomForestClassifier

2.2.2.1 Theorie

Unterschiede gegenüber Entscheidungsbäumen

1. trainieren von vielen Bäumen (oft einige 100)
2. jeder Baum basiert auf einer Zufallsstichprobe der Trainingsdaten
3. jeder Split-Versuch basiert auf einem zufällig ausgewählten Teil der Features
4. Bäume werden aber voll ausgebildet
5. Vorhersage: alle Bäume werden ausgewertet → Majority Vote und die Ergebnisse konsolidiert (aggregiert)



2.2.2 Klassifikation - Regelbasiert - RandomForestClassifier

2.2.2.1 Theorie

1. viele Bäume:
   - basiert auf der Idee von Schwarmintelligenz (Kollektive Intelligenz)
   - viele "lokale" Experten erreichen gemeinsam unter Umständen zuverlässigere Voraussagen als ein globaler Superexperte (z.B. lokale Wettervorhersagen), und die Voraussagen von weniger kompetenten Experten neutralisieren sich gegenseitig (vgl. Gesetz der Grossen Zahlen), wenn davon ausgegangen wird, dass deren Voraussagen zufällig und unabhängig sind
2. unabhängige Bäume:
   - die einzelnen Bäume müssen aber möglichst unterschiedlich sein (keine Monokulturen)
   - wird erreicht durch Stichproben derselben Grösse wie die jeweilige Population (Trainingsset), aber mit Zurücklegen (Bootstrap Sampling, mehr Details dazu in Kap. 4.2.3 Bootstrap Validierung)
   - Instanzen, welche nicht in der Stichprobe vorhanden sind, können zum internen Validieren verwendet werden (Out of Bag, OOB)

2.2.2 Klassifikation - Regelbasiert - RandomForestClassifier

2.2.2.1 Theorie

3. split-Versuche auf Teil der Features:
   - für jeden Split (!) wird nur eine Zufallsstichprobe der vorhandenen Features berücksichtigt, dies erhöht die Vielfalt der Bäume zusätzlich
   - weitere Vorteile
     - die einzelen Bäume (und damit der ganze Wald) werden schneller aufgebaut
     - gibt nach dem Trainieren zuverlässigere Information zur Wichtigkeit der einzelnen Features
   - Anzahl Features pro Split, Default :

$$int(\sqrt{Anz\_Features})$$

4. vollständige Ausbildung der Bäume:
   - durch den Einsatz vieler Bäume, basierend auf Samples, entfällt hier das Risiko von Overfitting
   - bestimmen der Veränderung von min_impurity_decrease (vgl. DecisionTreesClassifier) entfällt per Default, daher weiterer Zeitgewinn beim Trainieren

2.2.2 Klassifikation - Regelbasiert - RandomForestClassifier

2.2.2.1 Theorie

5. Majority vote:
   - die Gesamtheit der Vorhersagen wird konsolidiert
   - differenziertere Rückgabe als
     - wahrscheinlichste Klasse als Kategorie
     - Wahrscheinlichkeitswerte für jede Klasse als Realwert (vgl. Kap. 4.4.1.4)

- weitere Informationen können aus dem trainierten Modell extrahiert werden wie z.B. die Wichtigkeit der Features (vgl. Kap. 2.2.2.4)

2.2.2 Klassifikation - Regelbasiert - RandomForestClassifier

2.2.2.2 Praxis

- importieren der benötigten Funktion (Klasse), wie beinahe alles aus scikit-learn

```
from sklearn.ensemble import RandomForestClassifier
```

- definieren des Modells und trainieren, mit den schon vorbereiteten Daten, vorerst mit den Default Parametern (ausser random_state)

```
model = RandomForestClassifier(random_state = 1234)
model.fit(X_train, y_train)
```

```
RandomForestClassifier(1234)
```

- .get_params() gibt alle für das trainieren wirksamen Parameter zurück

```
print(model.get_params())
```

```
{'bootstrap': True, 'ccp_alpha': 0.0, 'class_weight': None, 'criterion': 'gini',
'max_depth': None, 'max_features': 'auto', 'max_leaf_nodes': None, 'max_samples':
None, 'min_impurity_decrease': 0.0, 'min_impurity_split': None,...
```

2.2.2 Klassifikation - Regelbasiert - RandomForestClassifier

2.2.2.2 Praxis

- einige davon sind bereits von DecisionTreeClassifier her bekannt und wirken analog
  - einige sind spezifisch für diesen Klassifikator und werden unten detaillierter vorgestellt
- abrufen von accuracy mit der klasseneigenen Methode

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

```
0.8749619714024947
```

- Fazit: noch etwas besser als DecisionTreeClassifier

2.2.2 Klassifikation - Regelbasiert - RandomForestClassifier

2.2.2.2 Praxis

- weitere Informationen zum trainierten Modell welche abgerufen werden können (die hier auskommentierten generieren zu langen Output und sind auch nicht zweckdienlich interpretierbar)

```
print('model.n_classes_:', model.n_classes_)
print('model.classes_:', model.classes_)
print('model.n_features_in_:', model.n_features_in_)
print('model.n_outputs_:', model.n_outputs_)
#print(model.base_estimator_)
#print(model.estimators_)
```

```
model.n_classes_: 2
model.classes_: ['no' 'yes']
model.n_features_ 29
model.n_outputs_: 1
```

2.2.2 Klassifikation - Regelbasiert - RandomForestClassifier

2.2.2.2 Praxis

- da für jeden Baum nur ein Teil der Beobachtungen verwendet wird (Bootstrap Sample, vgl. Kap. 4.2.3), sind beste Voraussetzungen gegeben, dass gleich eine **interne** Validierung durchgeführt werden kann
- dazu werden einfach jene Beobachtungen als Testset verwendet, welche nicht für den Aufbau des Baums benutzt worden waren
- wird in der Modelldefinition der Parameter oob_score=True gesetzt, kann aus dem trainierten Modell mit .oob_score_ die (interne) Performance des gesamten Waldes abgefragt werden (oob steht für Out Of the Bag)

```
model = RandomForestClassifier(random_state = 1234, oob_score = True)
model.fit(X_train, y_train)
print('model.oob_score_:', model.oob_score_)
```

```
model.oob_score_: 0.886961813479385 4
```

2.2.2 Klassifikation - Regelbasiert - RandomForestClassifier

2.2.2.3 Parameter Tuning

- wie gesehen, sind die meisten Parameter für RandomForestClassifier bereits von DecisionTreeClassifier her bekannt, und haben auch dieselbe Wirkungsweise (vgl. Doku)
- einige davon kommen hier neu dazu (oder werden erst hier von Bedeutung)
- die wichtigsten
  - **n_estimators** (default=100): Anzahl Bäume, die für den Wald gebildet werden sollen
  - **max_features** (default=Auto, d.h. sqrt(Anzahl Features)):
    Anzahl Features, welche für jeden Split zu berücksichtigen sind
    - diese werden zufällig ausgewählt
    - steht auch für DecisionTreeClassifier zur Verfügung, wird aber dort kaum eingesetzt
  - **oob_score** (default=False): bestimmt, ob mit den OOB Samples eine interne Validierung durchgeführt werden soll
    - kein Tuning Parameter

2.2.2 Klassifikation - Regelbasiert - RandomForestClassifier

2.2.2.3 Parameter Tuning

- weitere, welche aber hier nicht vertiefter erläutert werden:
  - warm_start
  - max_samples
  - bootstrap

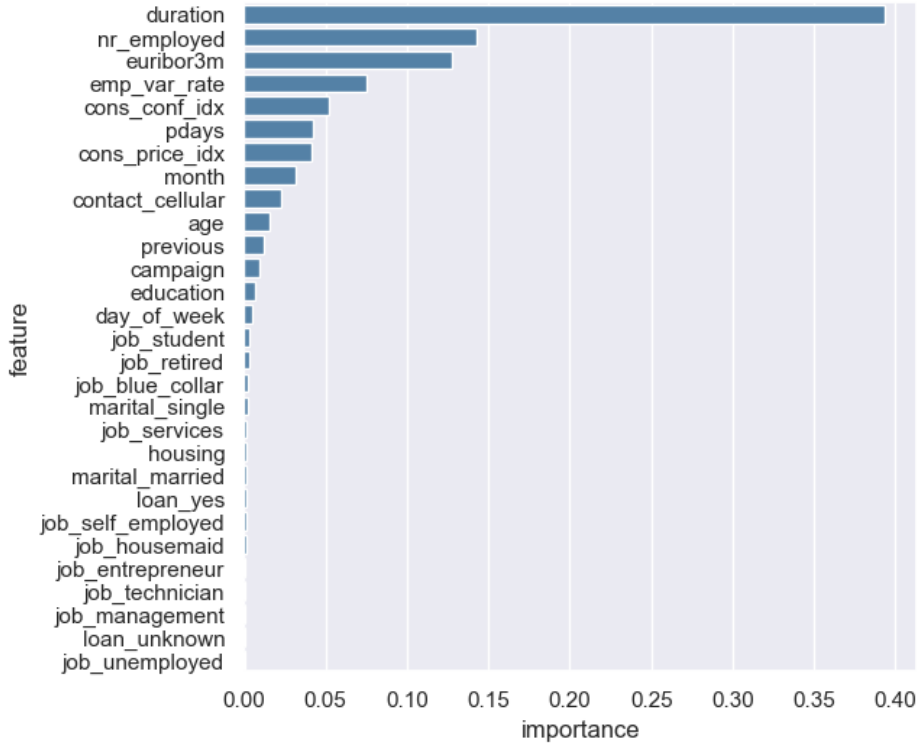
  Details dazu in der Online Dokumentation

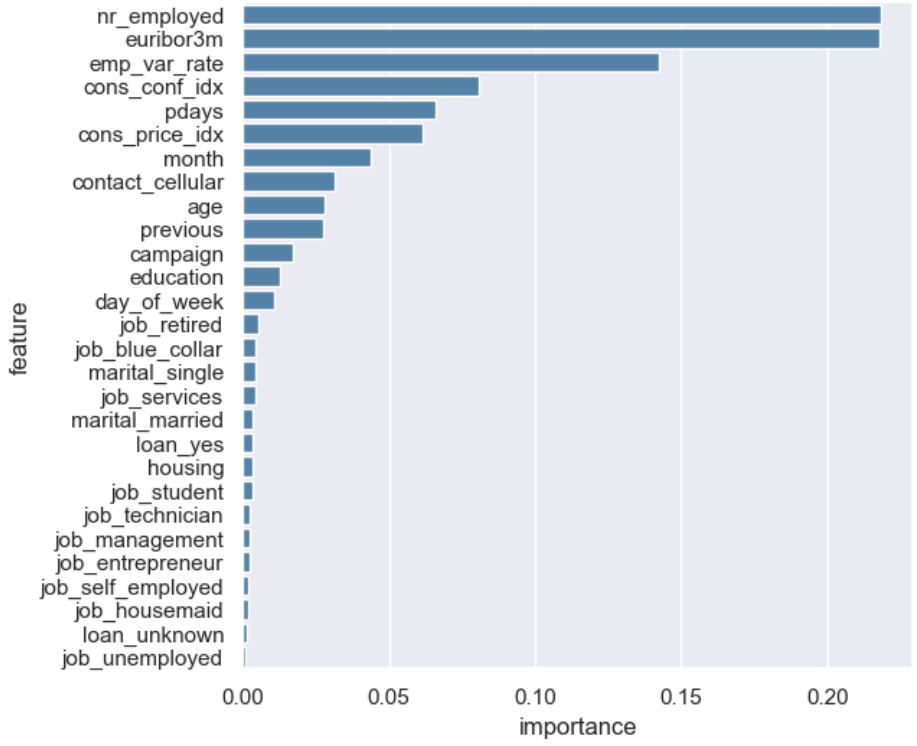
- für das konkrete Tuning der Parameter kann auf dieselbe Weise vorgegangen werden wie bei DecisionTreeClassifier (sowie bei allen andern Klassifikatoren)
- auf ein entsprechendes Codebeispiel wird hier daher verzichtet

2.2.2 Klassifikation - Regelbasiert - RandomForestClassifier

2.2.2.4 Feature Importance

- analog zu DecisionTreeClassifier verfügt auch RandomForestClassifier über ein Attribut, welches die Feature Importance zurückgibt: **.feature_importances_**
- diese wird auf dieselbe Weise aufgerufen (vgl. [ipynb])
- unten werden die Auswertungen mit und ohne "duration" einander gegenübergestellt





2.2.2 Klassifikation - Regelbasiert - RandomForestClassifier

2.2.2.5 Parallelisierung

- model.get_params() zeigt unter anderem folgende Einstellung: 'n_jobs': None
- dieser Parameter steuert gemäss Dokumentation eine allfällig mögliche Parallelisierung einiger Methoden (falls dies die verwendete Infrastruktur zulässt), bei RandomForestClassifier z.B.
  - fit
  - predict
  - decision_path
  - apply
- Einstellungen:
  - none (default): 1 Core
  - n: n Cores
  - -1: alle verfügbaren Cores
- ein Vergleich der bnötigten Zeiten für verschieden Werte von n_jobs ergibt in etwa nebenstehende Tabelle

n_jobs	time[s]
None	1.367
1	1.448
-1	0.810

2.2.2 Klassifikation - Regelbasiert - RandomForestClassifier

2.2.2.5 Parallelisierung

- der Defaultwert für n_jobs kann je nach Infrastruktur global gesetzt werden
  vgl. joblib.parallel_backend
- weitere wichtige Funktionen / Klassenmethoden, welche parallelisiert ausgeführt werden können:
  - Klassifikation (Kap. 2)
    - KNeighborsClassifier
    - LogisticRegression
  - Regression (Kap. 3)
    - LinearRegression
    - KNeighborsRegressor
    - RandomForestRegressor
  - Validierung (Kap. 4)
    - GridSearchCV
    - RandomizedSearchCV

2.2.3 Klassifikation - Regelbasiert - AdaBoostClassifier

2.2.3.1 Theorie

AdaBoost (Adaptive Boosting)

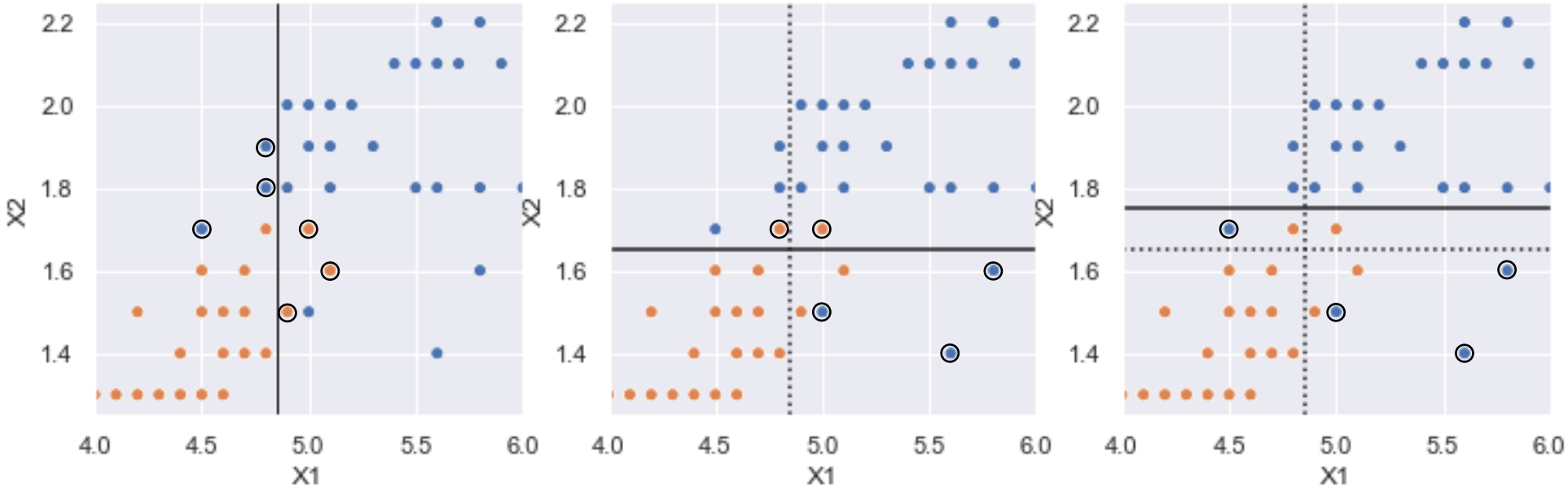
- Adaptiv: anpassungsfähig
- Booster: (Wirkungs-) Verstärker

- ist ein Meta-Learner, mit welchem unterschiedliche Klassifikatoren optimiert werden können
- trainiert einen ersten Klassifikator auf den Ausgangsdaten
- modifiziert danach Kopien des Klassifikators auf demselben Dataset, bei welchem aber die Gewichte falsch klassierter Beobachtungen derart angepasst werden, so dass nachfolgende Klassifikatoren diese mehr berücksichtigen (und damit auf schwieriger zu klassifizierende Beobachtungen fokussieren)
- für jeden Klassifikator werden "schwache" Learner verwendet, z.B. max_depth=1 bei DecisionTreeClassifier

2.2.3 Klassifikation - Regelbasiert - AdaBoostClassifier

2.2.3.1 Theorie

- es werden so lange neue Klassifikatoren aufgebaut, bis die maximale Anzahl (n_estimators) erreicht ist, bei perfektem Fit wird der Aufbau vorher abgebrochen
- im hier visualisierten Beispiel aus dem Demo Datensatz werden bloss drei Bäume aufgebaut, welche jeweils auch nur über einen einzigen Knoten verfügen
- die markierten Beobachtungen werden also beim nachfolgenden Baum stärker gewichtet



2.2.3 Klassifikation - Regelbasiert - AdaBoostClassifier

2.2.3.2 Praxis

- in aller Kürze: das Einzige, was sich von den bisherigen Klassifikatoren unterscheidet: die verwendete Funktion, vorerst mit Standard-Parametrisierung (beinahe)

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import accuracy_score
:
model = AdaBoostClassifier(algorithm='SAMME')
:
```

- die modell-interne Funktion (Methode) .score() liefert Accuracy direkt

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

```
0.8585336172801947
```

2.2.3 Klassifikation - Regelbasiert - AdaBoostClassifier

2.2.3.2 Praxis

- die verwendete Default Parametrisierung:

```
print(model.get_params())
```

```
{'algorithm': 'SAMME.R', 'base_estimator': 'deprecated', 'estimator': None,
'learning_rate': 1.0, 'n_estimators': 50, 'random_state': None}
```

die Modellparameter:

- **estimator**=None (DecisionTreeClassifier(max_depth=1)), wenn None (default) wird DecisionTreeClassifier mit max_depth=1 verwendet
  - es können folgende Modifikationen vorgenommen werden:
    - anderer Classifier (aber nicht alle)
    - modifizieren der Parameter des Classifiers
- **n_estimators**=50: Anzahl zu trainierender Teilmodelle

2.2.3 Klassifikation - Regelbasiert - AdaBoostClassifier

2.2.3.2 Praxis

- **learning_rate**=1.0: Lernrate, gemäss Dokumentation gibt es einen Kompromiss zwischen learning_rate und n_estimators (ansonsten schweigt sie sich vornehm aus)
- **algorithm**='SAMME.R'
  - Alternative: SAMME
  - SAMME.R kann nur verwendet werden, wenn der Klassifikator Wahrscheinlichkeiten für die vorausgesagten Klassen bestimmen kann
    - konvergiert schneller

- `base_estimator` wird hier als "deprecated" bezeichnet, weshalb wohl? recherchieren Sie kurz

2.2.3 Klassifikation - Regelbasiert - AdaBoostClassifier

2.2.3.2 Praxis

- weitere Informationen zum trainierten Modell:

```
print(model.estimator_)
```

```
DecisionTreeClassifier(max_depth=1)
```

- zeigt die Parametereinstellungen des Basis-Klassifikators an
- einzige Modifikation gegenüber Standard-Parametrisierung von diesem:
  max_depth=1 (→ schwacher Learner)

2.2.4 Klassifikation - Regelbasiert - GradientBoostingClassifier

2.2.4.1 Theorie

- wie bei AdaBoost werden auch bei GradientBoosting Bäume (oder andere Learner) sequenziell aufgebaut, indem jeder Baum versucht, die Fehler des Vorgängers zu korrigieren
- im Gegensatz zu AdaBoost werden aber nicht die Gewichte der im Vorgänger falsch klassierten Instanzen verstärkt, sondern die Vorhersagefehler (Residuen) des Vorgängers vorausgesagt, um diese schrittweise zu minimieren, resp. zu korrigieren
- gemäss Literatur ist GradientBoosting robuster, d.h. weniger anfällig auf Extremwerte
- mehr Details z.B. unter [towards data science]

2.2.4 Klassifikation - Regelbasiert - GradientBoostingClassifier

2.2.4.2 Praxis

```
from sklearn.ensemble import GradientBoostingClassifier
model = GradientBoostingClassifier()
model.fit(X_train, y_train)
print(model.score(X_test, y_test))
```

```
0.8767873440827503
```

- **Parameter**: einige davon sind schon von DecisionTreeClassifier her bekannt (z.B. min_impurity_decrease)

2.2.5 Klassifikation - Regelbasiert - HistGradientBoostingClassifier

- relativ neu in scikit-learn: HistGradientBoostingClassifier
  - lernt schneller als GradientBoostingClassifier
  - kann intern mit NAs umgehen

```
from sklearn.ensemble import HistGradientBoostingClassifier
model = HistGradientBoostingClassifier()
model.fit(X_train, y_train)
print(model.score(X_test, y_test))
```

```
0.8819592333434743
```

2.2.6 Klassifikation - Regelbasiert - Weitere sklearn fremde Methoden

- die beiden hier noch aufgeführten Klassifikatoren stammen nicht aus scikit-learn und müssen gegebenenfalls separat installiert werden (vgl. [ipynb])
- sie weisen aber analoge Schnittstellen auf, so dass sie leicht in entsprechende Modellvergleiche integriert werden können (vgl. Kap. 2.2.7)

catboost

- gute Performance
- kann automatisch mit kategorialen Features umgehen
- kann in unterschiedlichsten ML Umgebungen integriert werden
- weist über 100 Parameter auf!
- braucht aber relativ viel Zeit

lightgbm

- schnelle Trainingszeit und gute Performance
- weniger Speicherbedarf
- Unterstützung von parallelem, verteilten und GPU Learning
- geeignet für sehr grosse Datenmengen

2.2.7 Klassifikation - Regelbasiert - Modellvergleiche

- analog dem systematischen Vergleich verschiedener Tuning Parameter können auch unterschiedliche Lernmethoden mit Hilfe eines Loops miteinander verglichen werden
- vorab muss jeweils sichergestellt sein, dass
  - die Klassen importiert sind
  - die verschiedenen zu vergleichenden Modellobjekte instanziiert und parametrisiert sind
  - letztere können dann in einer Liste als Iterator zusammengestellt werden:

```
models = [
    KNeighborsClassifier(n_neighbors=5),
    DecisionTreeClassifier(min_impurity_decrease=0.002),
    RandomForestClassifier(n_estimators =200),
:
]
```

2.2.7 Klassifikation - Regelbasiert - Modellvergleiche

- je nach Anforderungen werden ausserhalb des Loops noch verschiedene leere Listen zum Sammeln der Iterations-Resultate bereitgestellt, z.B.

```
scores = []
used_times = []
model_names = []
```

  - score: sammelt accuracy_score (interner Scorer aller Methoden)
  - used_time: für die Zeitmessung jedes einzelnen Modells
  - model_name: wird verwendet, um die jeweiligen ermittelten Werte dem entsprechenden Modell zuordnen zu können

- im Loop selber werden dann alle Modelle der Liste
  - trainiert
  - evaluiert
  - und die Score Werte gesammelt

2.2.7 Klassifikation - Regelbasiert - Modellvergleiche

```
for model in models:
    start_time = time.time()                ## start timer
    model.fit(X_train, y_train)             ## train
    name = model.__class__.__name__         ## pick model name (for output only)
    score = model.score(X_test, y_test)     ## calculate score
    t = time.time() - start_time            ## calculate used time
    ## collect iteration results
    scores.append(score)
    used_times.append(t)
    model_names.append(name)
```

- als Fortschrittsbericht innerhalb des Loops auch gleich die Resultate ausgeben:

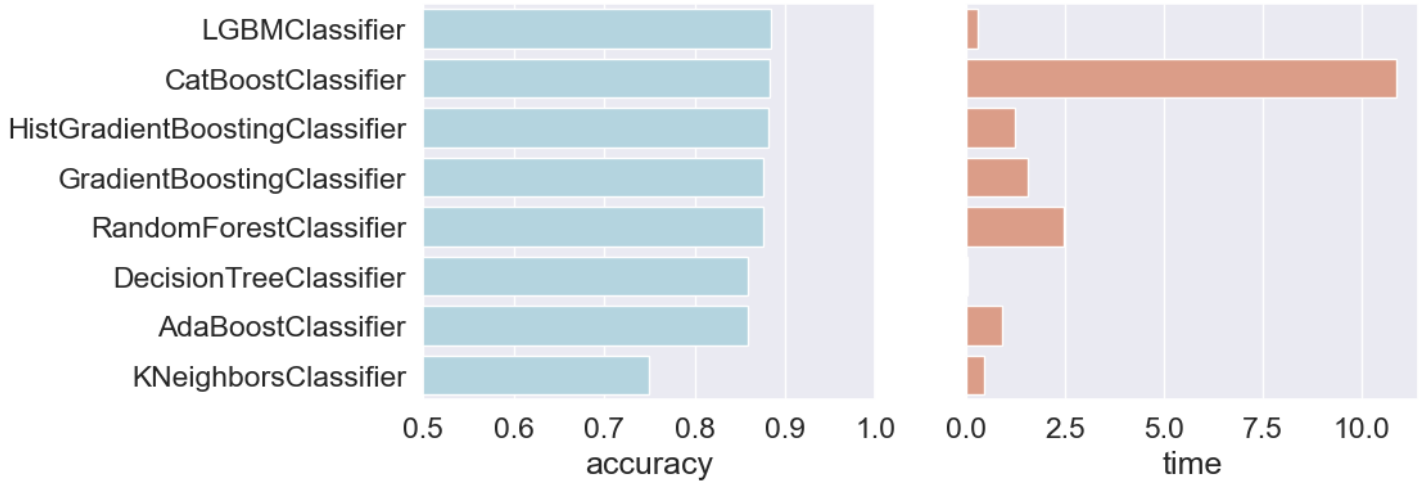
```
print('%-32s %6.4f %6.3f' % (name, score, t)) ## console output
```

2.2.7 Klassifikation - Regelbasiert - Modellvergleiche

- der Konsolenoutput

```
Classifier                          Score    Time
================================================
KNeighborsClassifier                0.7493    0.472
DecisionTreeClassifier              0.8588    0.047
RandomForestClassifier              0.8780    2.703
AdaBoostClassifier                  0.8585    0.928
GradientBoostingClassifier          0.8765    1.619
HistGradientBoostingClassifier      0.8820    1.462
CatBoostClassifier                  0.8829    9.491
LGBMClassifier                      0.8850    0.317
```

- sowie die Visualisierungen



2.2.7 Klassifikation - Regelbasiert - Modellvergleiche

- **Evaluation**

```
print('best_model :', model_names[scores.index(max(scores))])
print('best_score :', max(scores))
print('used_time  :', used_times[scores.index(max(scores))])
```

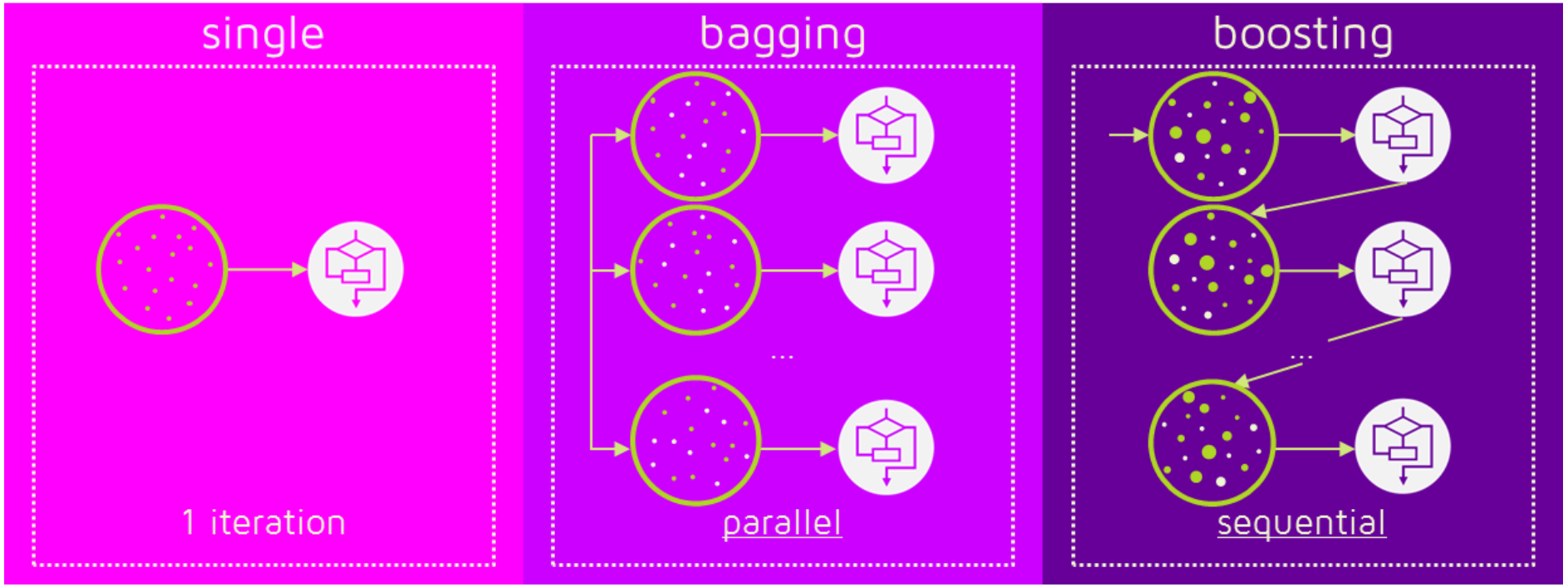
```
best_model : LGBMClassifier
best_score : 0.885001521143900 2
used_time  : 0.31695556640625
```

- **Fazit:**
  - beste Performance
    - LGBMClassifier
    - CatBoostClassifier
    - HistGradientBoostingClassifier
  - schnellster Classifier: DecisionTreeClassifier

2.2 Klassifikation - Regelbasiert

Rekap Bagging und Boosting

- abschliessend noch einmal eine Gegenüberstellung der wichtigsten Regelbasierten Methodengruppen (Quelle)
  - DecisionTreeClassifier (single)
  - RandomForestClassifier (bagging)
  - boosted Methodes (boosting)



2.2 Klassifikation - Regelbasiert

Workshop 06

Gruppen zu 2 bis 4, Zeit: 30'

- untersuchen Sie die folgenden Tuning-Parameter von RandomForestClassifier in Bezug auf die erreichte Performance (accuracy_score) mit dem vorbereiteten Dataset:
  - n_estimators als range(100, 500, 50)
  - max_features als range(1, 11)
  - min_impurity_decrease als np.arange(0, 0.1, 0.01)
- wie wirkt sich der random_state aus?

- Zusatzfrage: welche der ausserdem zur Verfügung stehenden Parameter sind keine Tuning Parameter? Konsultieren Sie dazu die (Online-) Dokumentation