Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Machine Learning Algorithmen: Decision Trees

CAS Practical Machine Learning

▸ Violeta Vogel, TI BFH

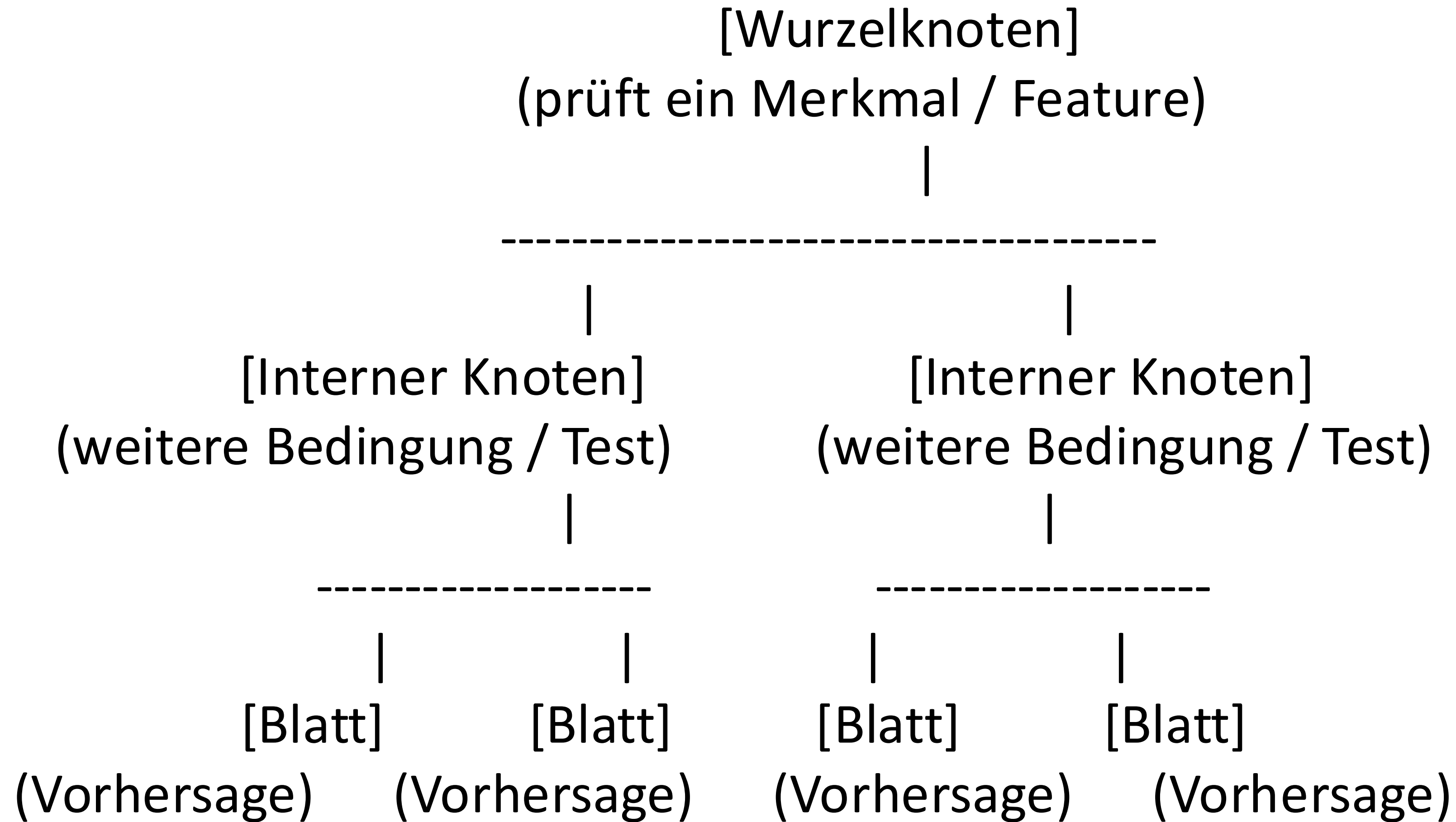
Algorithmen: Übersicht

Klasse	Beispiel-Algorithmen	Typische Einsätze
Lineare Modelle	Lineare Regression, Logistische Regression, Ridge, Lasso	Prognosen, Risikoanalyse, einfache Klassifikation, Wirtschaftsdaten
Baumverfahren	CART, C4.5, Regression Trees	Entscheidungslogik, Kreditvergabe, Medizin, Interpretierbarkeit
Ensemble-Methoden	Random Forest, Gradient Boosting, XGBoost, AdaBoost	Höchste Genauigkeit bei tabellarischen Daten, Industrie, Finance
Support Vector Machines	SVC, SVR, Kernel-SVM	Hochdimensionale Daten, Textklassifikation, Bioinformatik
Naive Bayes	Multinomial NB, Gaussian NB, Bernoulli NB	Spamfilter, Textklassifikation, Dokumentanalyse
k-Nearest Neighbors	kNN-Klassifikation, kNN-Regression	Kleine Datensätze, Empfehlungssysteme, Mustererkennung
Neuronale Netze	Feedforward-Netze, CNN, RNN, Transformer	Bilderkennung, Sprache, Zeitreihen, Deep Learning

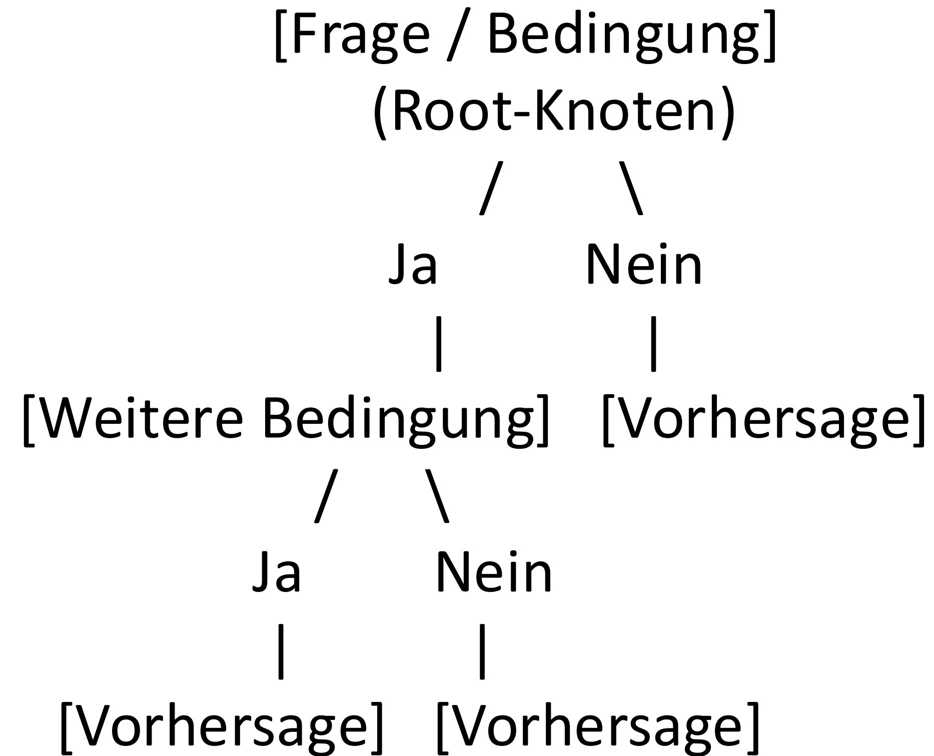
Entscheidungsbäume

- Ein Entscheidungsbaum ist ein Modell, das Daten anhand von Wenn-Dann-Regeln klassifiziert oder vorhersagt.
- Er besteht aus:
  - Wurzelknoten — der erste Entscheidungspunkt
  - Entscheidungsknoten — weitere Fragen basierend auf Merkmalen
  - Blattknoten — finale Entscheidung bzw. Klasse
  - Ästen — die Wege zwischen den Fragen
  - Der Baum versucht, die Daten so aufzuteilen, dass die Gruppen möglichst rein werden, also möglichst nur noch eine Klasse enthalten.

Entscheidungsbäume



Entscheidungsbäume



Entscheidungsbäume

- Ein Entscheidungsbaum arbeitet nach dem Prinzip Top-Down – Divide and Conquer:
  - Bestes Merkmal wählen Das Modell sucht das Merkmal, das die Daten am besten trennt (z. B. nach *Gini* oder *Entropie*).
  - Daten aufteilen Die Daten werden entlang dieses Merkmals in zwei oder mehr Gruppen geteilt.
  - Rekursiv weiter aufteilen Jeder neue Knoten wird wieder nach dem besten Merkmal gesplittet.
  - Stoppen, wenn
    - die Daten in einem Knoten homogen sind
    - keine sinnvollen Splits mehr möglich sind
    - eine maximale Tiefe erreicht ist

Entscheidungsbäume

- Wichtige Eigenschaften
  - Interpretierbarkeit: Entscheidungsbäume sind sehr gut nachvollziehbar, da man jeden Schritt sehen kann.
  - Flexibilität: Sie funktionieren mit numerischen und kategorialen Daten.
  - Geringe Anforderungen an Datenvorbereitung: Keine Normierung oder Skalierung nötig.
  - Anfällig für Überanpassung: Ohne Begrenzung wächst der Baum zu tief und passt sich zu stark an Trainingsdaten an.

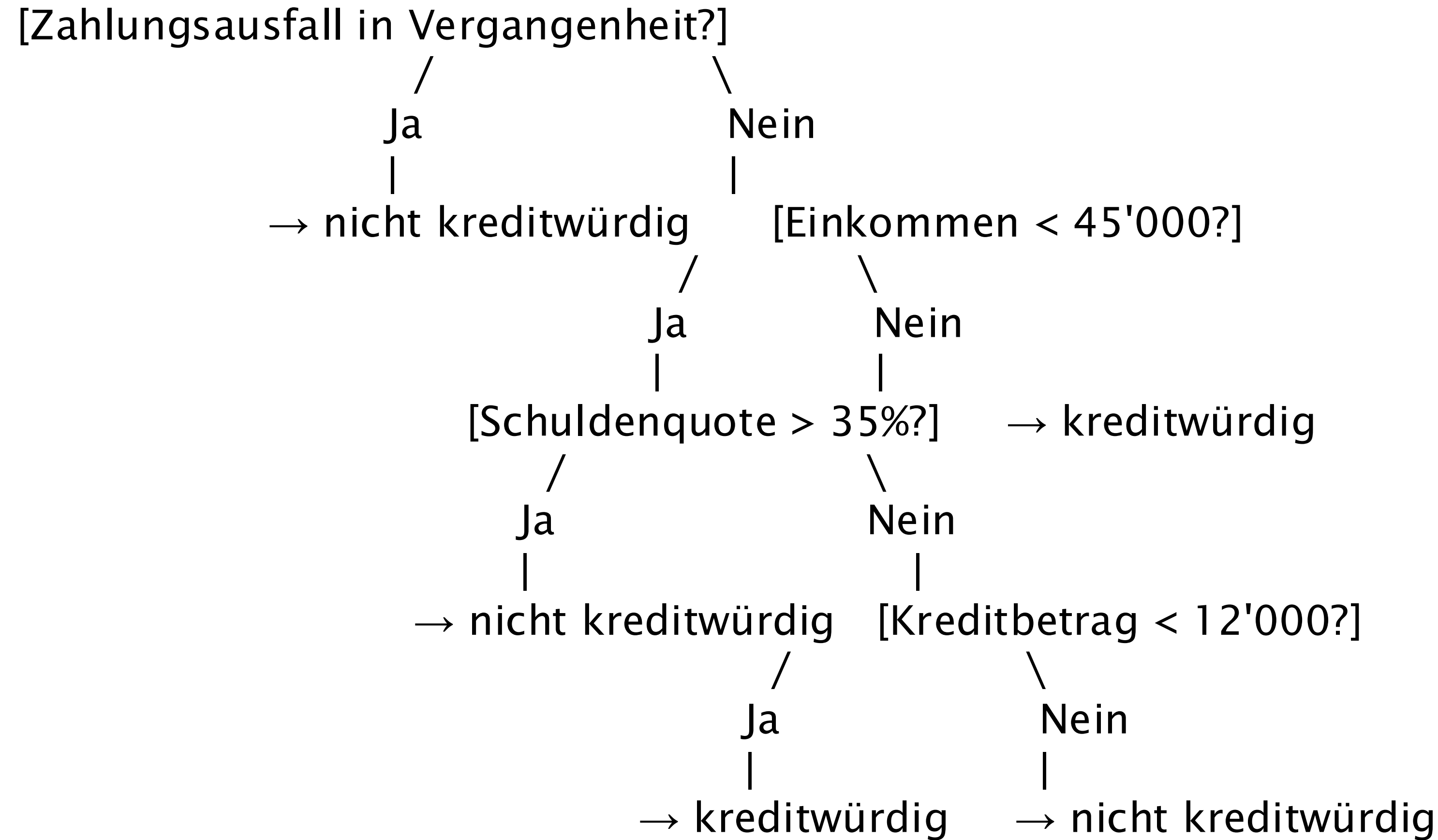
Entscheidungsbäume: Beispiel 1

- Spielt jemand bei einem bestimmten Wetter Golf? Der Baum könnte so aussehen:
  - Ist das Wetter **sonnig**?
    - Ja → Ist die **Luftfeuchtigkeit** hoch?
      - Ja → **Nein, spielt nicht**
      - Nein → **Ja, spielt**
  - Ist das Wetter **bewölkt**?
    - → **Ja, spielt**
  - Ist das Wetter **regnerisch**?
    - → Hängt vom **Wind** ab
- So entsteht ein klarer, nachvollziehbarer Entscheidungsweg.

Entscheidungsbäume: Beispiel 2

- Ein typischer Datensatz aus Finanzen enthält Merkmale wie:
  - Einkommen
  - Alter
  - Beschäftigungsdauer
  - Kreditbetrag
  - Schuldenquote
  - Vergangene Zahlungsausfälle
  - Kreditlaufzeit
  - Vermögen / Sicherheiten
- Ein Entscheidungsbaum versucht daraus abzuleiten, ob ein Kunde kreditwürdig ist oder nicht.

Entscheidungsbäume: Beispiel 2



Entscheidungsbäume: Wichtigkeit der Features

- Feature Importance misst, wie stark jedes Merkmal zur Vorhersage beiträgt.
  - Je höher der Wert, desto wichtiger ist das Feature für das Modell.
  - Ein einzelner Entscheidungsbaum bewertet Features danach, wie stark sie die Daten „rein“ machen, also wie gut sie Klassen trennen.
- Die Wichtigkeit basiert auf:
  - Gini-Reduktion
  - Entropie-Reduktion
  - oder Informationsgewinn
- Ein Feature ist wichtig, wenn es:
  - oft gesplittet wird
  - früh im Baum vorkommt
  - grosse Verbesserungen der Reinheit bringt

Entscheidungsbäume: Wichtigkeit der Features

- Warum Feature Importance wichtig ist
  - Modellverständnis: Wie das Modell denkt
  - Transparenz: Besonders wichtig für Erklärbarkeit
  - Feature Engineering: Erkennung der unnötigen oder dominanten Features
  - Bias-Erkennung: Nutzt das Modell unfaire Merkmale?

Entscheidungsbäume: Wichtigkeit der Features

- **Zwei Arten von Feature Importance**
  - **Impurity-based Importance**
    - schnell
    - Standard in scikit-learn
    - kann verzerrt sein (bevorzugt numerische Features)

  - **Permutation Importance**
    - robuster
    - misst, wie stark die Modellleistung sinkt, wenn ein Feature zufällig permutiert wird
    - bevorzugte Methode in der Praxis

Entscheidungsbäume: Impurity-based Importance

- Impurity-based Importance:
  - Ein Knoten ist rein, wenn alle Datenpunkte zur gleichen Klasse gehören
  - Ein Feature ist wichtig, wenn es beim Splitten viel Ordnung schafft – also die Klassen sauber trennt.
  - Die Wichtigkeit ergibt sich aus der Summe aller Reinheitsverbesserungen, die dieses Feature im Baum verursacht.

- Wann sollte man Impurity-based Importance nutzen?
  - Für erste Einschätzungen der Feature-Relevanz
  - Wenn man schnell eine Übersicht braucht
  - Wenn das Modell nicht zu viele korrelierte Features hat

Entscheidungsbäume: Impurity-based Importance

- Wie entsteht die Feature Importance?
  - Schritt 1: Split erzeugt Reinheitsgewinn
    - Wenn ein Feature einen Knoten teilt, wird die Unreinheit reduziert. Der Reinheitsgewinn ist:
      - Gain = $Impurity_{parent} - \sum_i \frac{n_i}{n} \cdot Impurity_{child_i}$
  - Schritt 2: Alle Gewinne pro Feature aufsummieren
    - Für jedes Feature werden alle Gains über den ganzen Baum addiert.
  - Schritt 3: Normalisieren
    - Die Summe aller Feature-Wichtigkeiten wird auf 1 normiert.

Entscheidungsbäume: Impurity-based Importance

- Vorteile
  - schnell zu berechnen
  - direkt aus dem Modell verfügbar
  - gut interpretierbar
- Nachteile
  - bevorzugt numerische Features mit vielen möglichen Splitpunkten
  - bevorzugt Features mit vielen Kategorien
  - kann bei korrelierten Features verzerrt sein
  - weniger robust als Permutation Importance

Entscheidungsbäume: Permutation Importance

- Permutation Importance misst die Bedeutung eines Features, indem es prüft, wie schlecht das Modell wird, wenn dieses Feature zerstört wird

- Wann sollte man Permutation Importance verwenden?
  - Wenn man verlässliche Feature-Wichtigkeiten braucht
  - Wenn man ein Black-Box-Modell erklärbar machen will
  - Wenn man Bias oder unerwartete Muster prüfen möchte
  - Wenn impurity-based Importance verzerrt wirkt

Entscheidungsbäume: Permutation Importance

- Wie Permutation Importance berechnet wird
  - Schritt 1.Baseline-Leistung messen
    - Zuerst misst man die Modellleistung auf den echten Daten (z. B. Accuracy, AUC).
  - Schritt 2. Ein Feature permutieren
    - Man mischt die Werte eines Features zufällig durch, sodass der Zusammenhang mit dem Ziel zerstört wird.
  - Schritt 3. Leistung erneut messen
    - Das Modell wird mit den permutierten Daten erneut evaluiert.
  - Schritt 4. Wichtigkeitswert berechnen
    - $\text{Importance} = \textit{Baseline Score} - \textit{Score nach Permutation}$
- Je grösser der Unterschied, desto wichtiger das Feature.

Entscheidungsbäume: Wichtigkeit der Features

Kriterium	**Impurity Importance**	**Permutation Importance**
Berechnung	während des Trainings	nach dem Training
Geschwindigkeit	sehr schnell	langsamer
Verzerrung	hoch (numerische Features bevorzugt)	gering
Robustheit	mittel	hoch
Modellabhängig	ja	nein
Umgang mit Korrelation	schlecht	mittel
Interpretierbarkeit	gut	sehr gut

Entscheidungsbäume: Gini-Impurity

- Die Gini-Impurity misst, wie wahrscheinlich es ist, dass zwei zufällig ausgewählte Elemente aus einem Knoten verschiedenen Klassen angehören.
  - Ein Entscheidungsbaum sucht bei jedem Split das Feature, das die grösste Reduktion der Gini-Impurity erzeugt.

- Für einen Knoten mit Klassenanteilen $p_1, p_2, \ldots, p_k$:

    - Gini $= 1 - \sum_{i=1}^{k} p_i^2$

- Gini = 0 → perfekte Reinheit (nur eine Klasse)
- Gini hoch → starke Durchmischung

Entscheidungsbäume: Gini-Impurity

- Beispiel: Angenommen, ein Knoten enthält:
  - 80 % Klasse A
  - 20 % Klasse B
- Dann:
    - $\text{Gini} = 1 - (0.8^2 + 0.2^2) = 1 - (0.64 + 0.04) = 0.32$
- Der Knoten ist relativ rein.

Entscheidungsbäume: Gini-Impurity

- Vorteile der Gini-Impurity
  - Sehr effizient Die Berechnung ist einfach (keine Logarithmen). Dadurch ist Gini schneller als Entropie und ideal für grosse Datensätze.
  - Stabile Splits Gini reagiert etwas „glatter“ als Entropie. Das führt oft zu stabileren, weniger überempfindlichen Splits.
  - Gut geeignet für Random Forests Die meisten Random-Forest-Implementationen nutzen Gini als Standard, weil es schnell und robust ist.
  - Ähnliche Resultate wie Entropie In der Praxis unterscheiden sich die resultierenden Bäume kaum, obwohl Gini schneller ist.

Entscheidungsbäume: Gini-Impurity

- Nachteile der Gini-Impurity
  - Bevorzugt Features mit vielen möglichen Splitpunkten Numerische Features oder solche mit vielen Kategorien werden oft überbewertet.
  - Kann bei unbalancierten Klassen verzerren. Wenn eine Klasse sehr selten ist, kann Gini diese weniger stark berücksichtigen als Entropie.
  - Lokales Kriterium Gini betrachtet nur die Reinheit im aktuellen Knoten, nicht die globale Baumstruktur. Das kann zu suboptimalen Splits führen.
  - Kann zu leicht anderen Splits führen als Entropie Obwohl die Resultate ähnlich sind, kann Gini in Grenzfällen andere Entscheidungen treffen, was den Baum leicht verändert.

Entscheidungsbäume: Entropie

- Entropie misst, wie viel Unsicherheit in einem Datensegment steckt. Ein Entscheidungsbaum versucht, diese Unsicherheit durch Splits zu reduzieren.
- Ein Entscheidungsbaum sucht bei jedem Split das Feature, das die grösste Reduktion der Entropie erzeugt. Diese Reduktion nennt man Informationsgewinn

- Für einen Knoten mit Klassenanteilen $p_1, p_2, \ldots, p_k$:
  - Entropie $= -\sum_{i=1}^{k} p_i \log_2 (p_i)$

- Entropie = 0 → perfekte Reinheit (nur eine Klasse)
- Entropie hoch → starke Durchmischung

Entscheidungsbäume: Entropie

- Beispiel: Ein Knoten enthält:
  - 50 % Klasse A
  - 50 % Klasse B
- Dann:
  - Entropie $= -\left(0.5 \log_2 0.5 + 0.5 \log_2 0.5\right) = 1$
- Das ist die maximale Entropie für zwei Klassen → maximale Unordnung.

Entscheidungsbäume: Entropie

- Vorteile der Entropie:
  - Hohe Sensitivität Entropie reagiert stärker auf kleine Veränderungen in der Klassenverteilung als die Gini-Impurity. Das führt oft zu präziseren Splits, besonders wenn Klassen ungleich verteilt sind.
  - Theoretisch fundiert Entropie stammt aus der Informationstheorie und misst die Unsicherheit eines Systems. Dadurch ist sie gut interpretierbar und mathematisch sauber definiert.
  - Gute Trennung bei seltenen Klassen Wenn eine Klasse selten vorkommt, erkennt Entropie deren Bedeutung oft besser als Gini.
  - Nützlich für Informationsgewinn Viele klassische Entscheidungsbaum-Algorithmen (z. B. ID3, C4.5) basieren auf dem Informationsgewinn, der direkt aus der Entropie berechnet wird.

Entscheidungsbäume: Entropie

- Nachteile der Entropie
  - Höherer Rechenaufwand Die Berechnung enthält Logarithmen, was sie langsamer macht als Gini – besonders bei grossen Datensätzen.
  - Ähnliches Verhalten wie Gini In der Praxis unterscheiden sich die resultierenden Bäume oft kaum. Der zusätzliche Aufwand bringt nicht immer einen klaren Vorteil.
  - Kann bei stark unbalancierten Daten instabil sein Wenn eine Klasse extrem selten ist, kann Entropie zu überempfindlichen Splits führen.
  - Kann zu tieferen Bäumen führen Da Entropie stärker auf kleine Unterschiede reagiert, entstehen manchmal komplexere Bäume, die anfälliger für Overfitting sind.

Entscheidungsbäume: Gini vs. Entropie

Kriterium	**Gini-Impurity**	**Entropie**
Rechenaufwand	tiefer	höher
Sensitivität	glatter	empfindlicher
Verhalten bei seltenen Klassen	weniger gut	besser
Baumkomplexität	tendenziell tiefer	tendenziell höher
Praxis	Standard in Random Forests	Standard in ID3/C4.5

Entscheidungsbäume: Splitting

- Splitting ist der Prozess, bei dem der Entscheidungsbaum die beste Frage auswählt, um die Daten zu trennen.
- Er testet viele mögliche Splits, bewertet sie mit einem Reinheitsmass und wählt den Split mit dem höchsten Informationsgewinn.

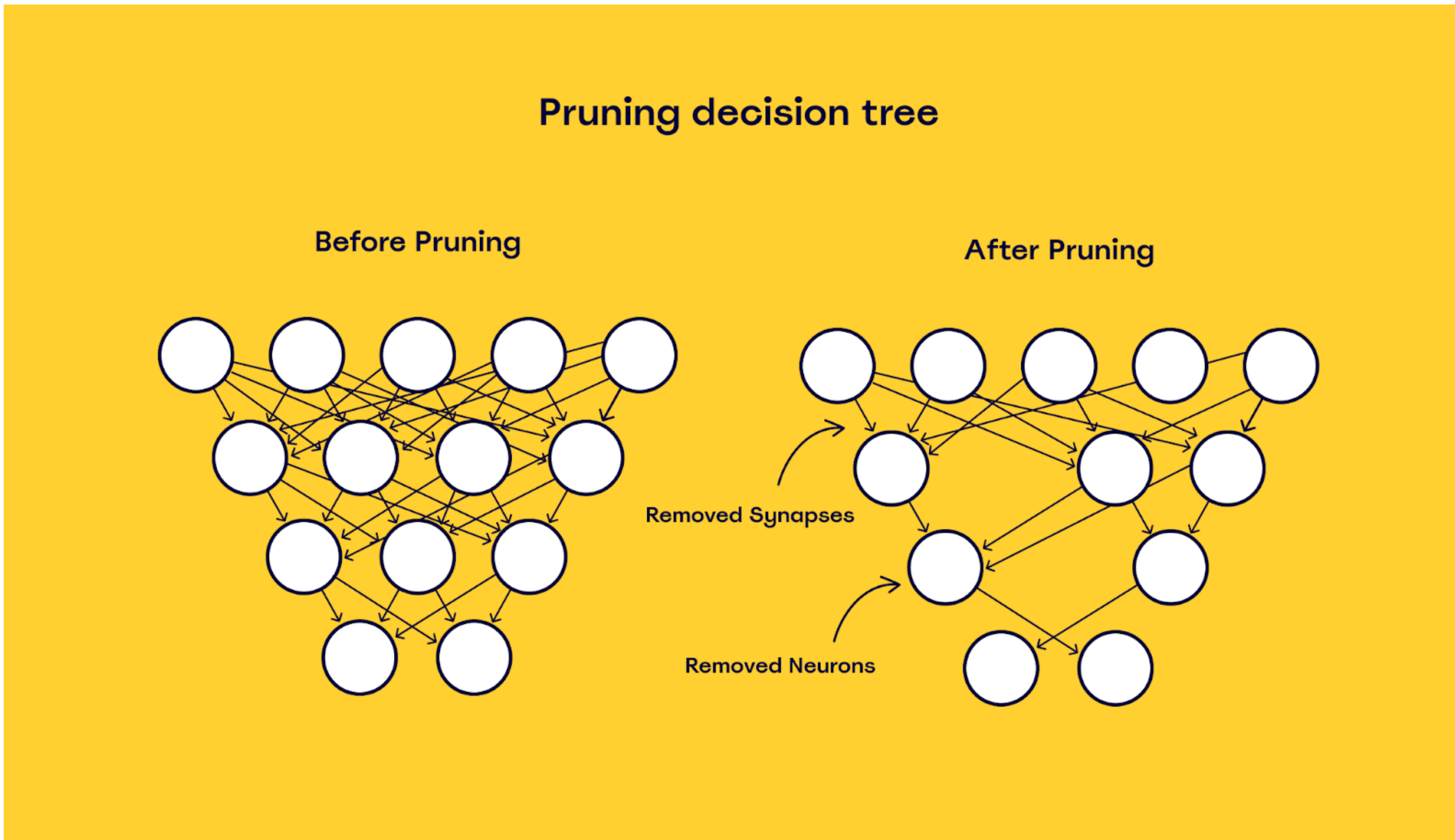
Entscheidungsbäume: Splitting

- 1. Alle Features testen
  - Für jedes Feature prüft der Baum viele mögliche Schwellenwerte.
    - Beispiel:
      - Feature „Alter“ → teste 20, 25, 30, 35 …
      - Feature „Einkommen“ → teste 40k, 50k, 60k …
- 2. Für jeden Split die Reinheit berechnen
  - Der Baum berechnet, wie „rein“ die beiden entstehenden Gruppen sind. Dazu nutzt er z. B.:
    - Gini-Impurity
    - Entropie
    - Misclassification Error

Entscheidungsbäume: Splitting

- 3. Split-Qualität bestimmen
  - Der Baum misst, wie stark sich die Reinheit verbessert:
    - $\text{Gain} = \textit{Reinheit vorher} - \textit{Reinheit nachher}$
  - Je grösser der Gain, desto besser der Split.
- 4. Den besten Split auswählen
  - Der Split mit dem höchsten Informationsgewinn wird gewählt.
- 5. Rekursiv weiter splitten
  - Der Prozess wiederholt sich für jeden neuen Knoten, bis ein Stoppkriterium erreicht ist.

Entscheidungsbäume: Pruning



Entscheidungsbäume: Pruning

- Pruning bedeutet, einen Entscheidungsbaum nach dem Training gezielt zu verkleinern, indem unnötige oder schwache Äste entfernt werden.

- Ziel ist es, Overfitting zu reduzieren und den Baum robuster, generalisierbarer und verständlicher zu machen.

- Ein ungeprunter Baum hat oft:
  - sehr tiefe Struktur
  - viele kleine Blätter
  - Splits, die nur wenige Datenpunkte betreffen
  - schlechte Generalisierung auf neue Daten
- Pruning entfernt genau diese Teile.

Entscheidungsbäume: Pruning

- Vorteile von Pruning
  - Reduziert Overfitting
  - Verbessert Generalisierung
  - Erhöht Stabilität
  - Vereinfacht Interpretierbarkeit
  - Verhindert unnötige Splits

- Nachteile von Pruning
  - Kann zu stark vereinfachen, wenn Parameter schlecht gewählt sind
  - Erfordert Validierungsdaten, um optimal zu funktionieren
  - Etwas rechenintensiver als Pre-Pruning
  - Nicht immer nötig, wenn Random Forests verwendet werden (dort übernimmt das Ensemble die Regularisierung)

Entscheidungsbäume: Pruning - Prepruning

- Pre-Pruning setzt Wachstumsgrenzen für den Baum, damit er nicht unnötig tief oder komplex wird (Stoppen während des Wachstums)
  - Der Baum wird nicht vollständig wachsen gelassen.

- Typische Kriterien:
  - maximale Tiefe (max_depth)
  - minimale Samples pro Blatt (min_samples_leaf)
  - minimale Verbesserung nötig für einen Split

Entscheidungsbäume: Pruning - Prepruning

- **Methoden:**
  - **max_depth:** Begrenzt, wie tief der Baum wachsen darf.
    - Tiefer Baum → komplex
    - Flacher Baum → generalisiert besser
  - **min_samples_split:** Ein Split wird nur durchgeführt, wenn genügend Datenpunkte vorhanden sind.
    - Verhindert Splits auf winzigen Gruppen
  - **min_samples_leaf:** Jedes Blatt muss mindestens eine bestimmte Anzahl Samples enthalten.
    - Glättet Entscheidungsgrenzen
  - **min_impurity_decrease:** Ein Split wird nur akzeptiert, wenn er die Reinheit genügend verbessert.
    - Verhindert unwichtige Splits

Entscheidungsbäume: Pruning - Prepruning

- Vorteile von Pre-Pruning
  - Schnelleres Training Der Baum wächst weniger tief → weniger Rechenaufwand.
  - Verhindert Overfitting früh Der Baum wird gar nicht erst zu komplex.
  - Einfachere Modelle Flachere Bäume sind leichter zu verstehen.
  - Weniger Speicherbedarf Kleinere Bäume benötigen weniger Ressourcen.

- Nachteile von Pre-Pruning
  - Kann gute Splits verhindern Wenn die Grenzen zu streng sind, stoppt der Baum zu früh.
  - Erfordert Hyperparameter-Tuning Falsche Werte führen zu Underfitting.
  - Weniger flexibel als Post-Pruning Post-Pruning lässt den Baum erst wachsen und schneidet dann gezielt.

Entscheidungsbäume: Pruning -Post-Pruning

- Post-Pruning (Baum erst wachsen lassen, dann schneiden)
  - Der Baum wächst vollständig und wird danach vereinfacht.
- Typische Methoden:
  - Cost-Complexity-Pruning (Standard in scikit-learn)
  - Reduced Error Pruning
- Vorteil: bessere Kontrolle, oft bessere Modelle Nachteil: etwas langsamer

Entscheidungsbäume: Pruning -Post-Pruning

- Post-Pruning schneidet einen bereits gewachsenen Baum zurück, um Overfitting zu reduzieren und die Modellqualität auf neuen Daten zu verbessern. (Baum erst wachsen lassen, dann schneiden)

- Der Prozess besteht aus drei Schritten:
  - Vollständigen Baum wachsen lassen Der Baum trennt so lange, bis alle Blätter rein sind oder keine Splits mehr möglich sind.
  - Qualität der Teilbäume prüfen Man evaluiert, ob ein Ast wirklich zur Verbesserung der Modellleistung beiträgt.
  - Unnötige Äste entfernen Wenn ein Ast die Leistung auf Validierungsdaten nicht verbessert, wird er abgeschnitten.

Entscheidungsbäume: Pruning -Post-Pruning

- Die zwei wichtigsten Post-Pruning-Methoden
  - Cost-Complexity-Pruning (Standard in scikit-learn)
    - Der Baum wird vereinfacht, indem man einen Kompromiss zwischen Fehler und Komplexität findet.
      - $R_\alpha(T) = R(T) + \alpha \cdot | T |$
    - $R(T)$:Fehler des Baums
    - $| T |$: Anzahl Blätter
    - α: Strafe für Komplexität
    - Je grösser α, desto stärker wird geschnitten.
  - Reduced Error Pruning
    - Man entfernt Äste, wenn der Baum auf Validierungsdaten nicht schlechter wird.
    - Sehr einfach, aber weniger flexibel.

Entscheidungsbäume: Pruning -Post-Pruning

- Vorteile von Post-Pruning
  - Reduziert Overfitting Der Baum wird robuster gegenüber neuen Daten.
  - Verbessert Generalisierung Weniger komplexe Modelle sind stabiler.
  - Erhöht Interpretierbarkeit Kleinere Bäume sind leichter zu verstehen.
  - Flexibler als Pre-Pruning Der Baum darf zuerst alle Muster entdecken.

- Nachteile von Post-Pruning
  - Rechenintensiver Der Baum muss zuerst vollständig wachsen.
  - Benötigt Validierungsdaten Ohne separate Daten ist die Entscheidung schwieriger.
  - Komplexere Implementierung Besonders bei Cost-Complexity-Pruning.

Entscheidungsbäume: Prepruning versus Postpruning

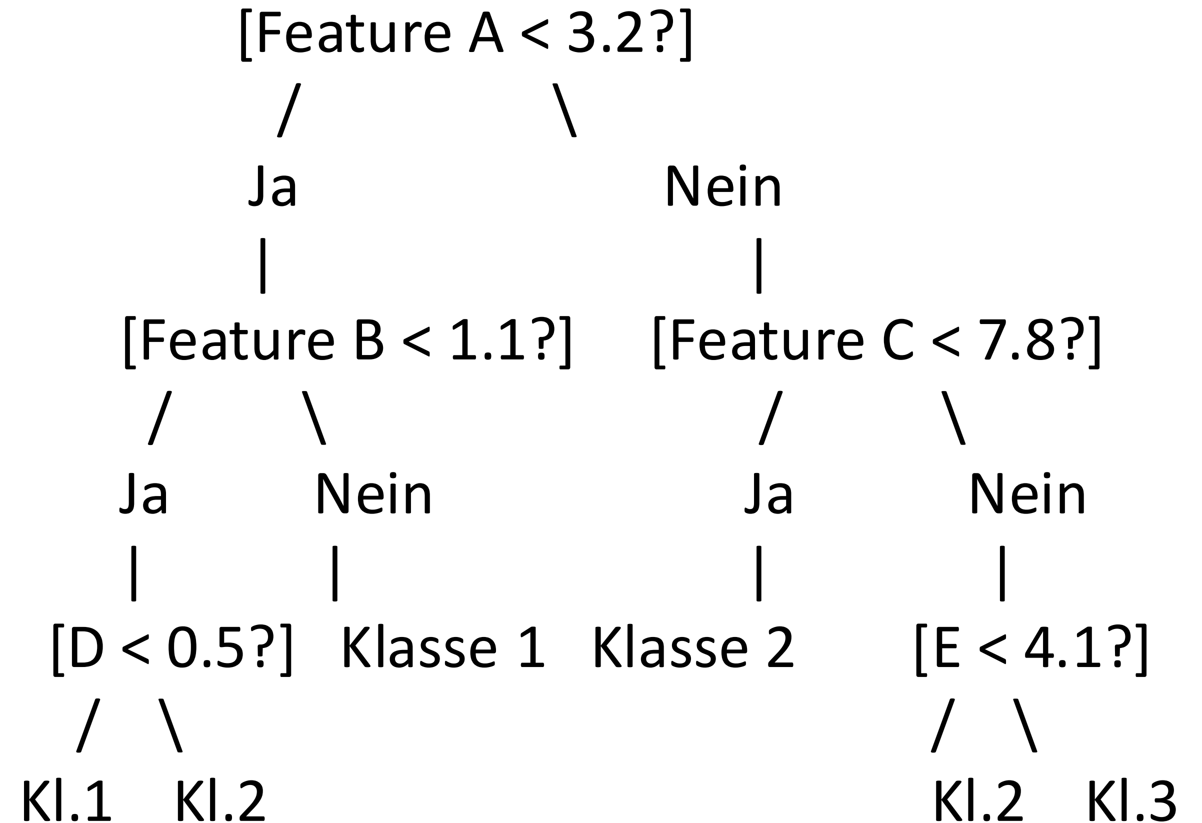
Aspekt	**Pre-Pruning**	**Post-Pruning**
Zeitpunkt	während des Wachstums	nach vollständigem Wachstum
Risiko	Underfitting	Overfitting (vor dem Schneiden)
Kontrolle	geringer	höher
Geschwindigkeit	schneller	langsamer
Modellqualität	gut	oft besser

- Post-Pruning ist die präziseste und effektivste Methode, um Entscheidungsbäume zu vereinfachen.
- Es führt fast immer zu besseren, stabileren und interpretierbareren Modellen als Pre-Pruning.

Entscheidungsbäume: Pruning Beipiel

- Vorher: Ungeprunter Entscheidungsbaum (überfit)
  - Der Baum ist zu tief, zu komplex und passt sich stark an das Rauschen der Trainingsdaten an.
- Eigenschaften
  - 7 Blätter
  - Tiefe: 5
  - Perfekte Trennung im Training
  - Schlechte Generalisierung
  - Viele Splits mit sehr wenigen Datenpunkten

Entscheidungsbäume: Pruning Beipiel



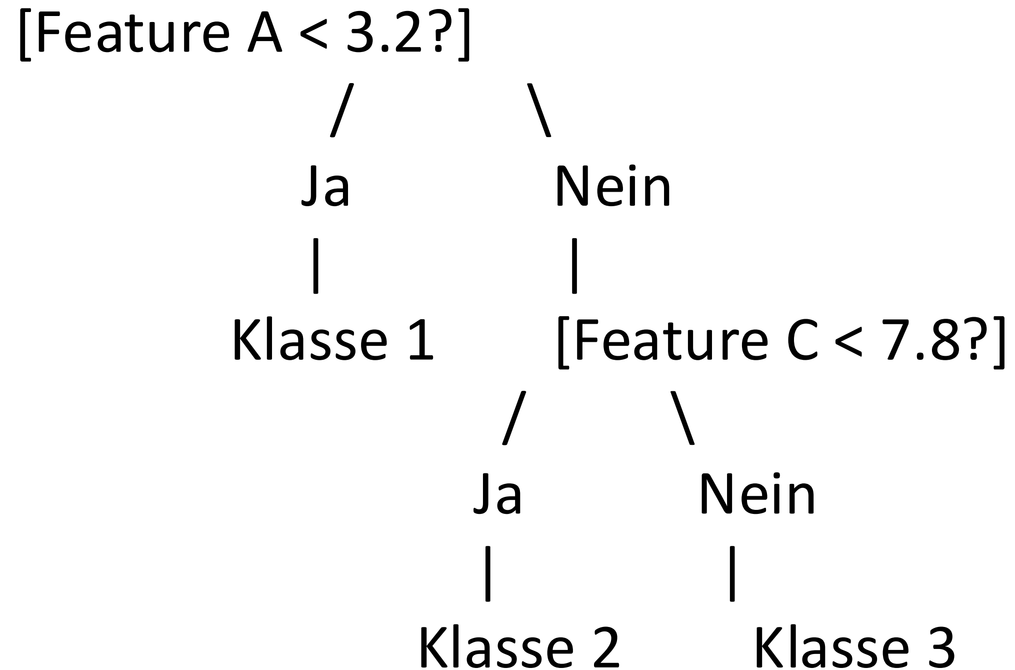
- Bewertung
  - Trainingsfehler: 0 %
  - Validierungsfehler: 14 %

- Problem: Der Baum hat Muster gelernt, die nur im Training vorkommen → Overfitting.

Entscheidungsbäume: Pruning Beipiel

- **Nachher: Geprunter Entscheidungsbaum (robust)**
  - Der Baum wurde mit **Cost-Complexity-Pruning** vereinfacht. Unnötige Äste wurden entfernt, ohne die Modellleistung zu verschlechtern.
- **Eigenschaften**
  - 3 Blätter
  - Tiefe: 2
  - Weniger Splits → weniger Varianz
  - Bessere Generalisierung
  - Deutlich einfacher zu interpretieren

Entscheidungsbäume: Pruning Beipiel



- Bewertung
  - Trainingsfehler: 4 % (leicht höher)
  - Validierungsfehler: 10 % (besser!)
  - Vorteil: Der Baum ist kleiner, stabiler und generalisiert besser.

Decision Tree Classifier

- Ein Decision Tree Classifier ist ein überwachter Machine-Learning-Algorithmus, der Daten anhand einer Reihe von regelbasierten Entscheidungen klassifiziert.
- Es erstellt ein umgedrehtes Baumdiagramm:
  - das oben mit einer Frage zu einem wichtigen Merkmal beginnt.
  - Jede Antwort führt zu einem neuen Ast mit einer weiteren Frage,
  - bis ein Blattknoten erreicht wird, der die finale Klassenzuordnung enthält.

- Wesentliche Elemente:
  - Wurzelknoten – erster Split basierend auf dem wichtigsten Merkmal
  - Entscheidungsknoten – weitere Splits basierend auf Feature-Werten
  - Blätter – finale Klassen
  - Äste – mögliche Entscheidungswege
  - Diese Struktur ist intuitiv, da sie einer Reihe von if-then-Regeln entspricht.

Decision Tree Classifier: Wie der Algorithmus arbeitet

- Der Decision Tree nutzt ein **Top-Down-Divide-and-Conquer-Verfahren**:
  - Wähle das beste Merkmal für den Split (z. B. über **Gini-Impurity** oder **Entropy**).
  - Teile die Daten in möglichst homogene Teilmengen.
  - Wiederhole den Vorgang rekursiv für jeden neuen Knoten.
  - Stoppe, wenn:
    - alle Datenpunkte in einem Knoten zur gleichen Klasse gehören
    - keine sinnvollen Splits mehr möglich sind
    - eine maximale Tiefe erreicht ist

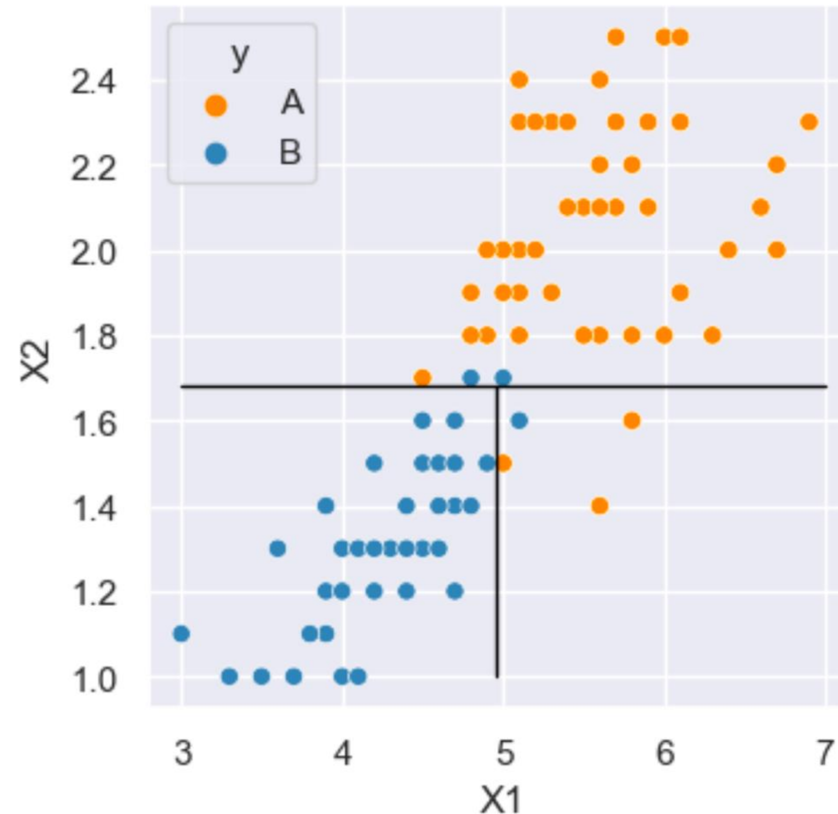
Decision Tree Classifier

- Vorteile:
  - Leicht interpretierbar
  - Funktioniert gut ohne starke Datenvorverarbeitung
  - Kann sowohl numerische als auch kategorische Daten verarbeiten
  - Schnell zu trainieren

- Nachteile:
  - Hohe Gefahr von Overfitting ohne Pruning
  - Instabil gegenüber kleinen Datenänderungen
  - Oft schlechtere Genauigkeit als Ensemble-Methoden (z. B. Random Forest)

Decision Tree Classifier: Splitting Beispiel

- Beispiel Datenset demo_data_class.csv
  - Kandidaten für ersten Split (von Auge abgeschätzt)
    - X1≈5.0
    - X2≈1.7

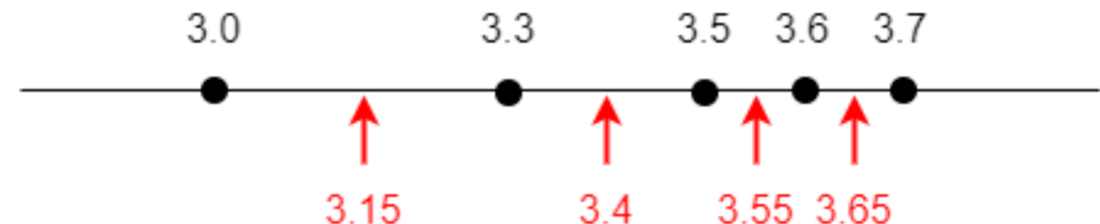


Decision Tree Classifier: Splitting Beispiel

- 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
:
```



Decision Tree Classifier: Splitting Beispiel

- 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                         cild_r
A      5                        A     40
B     34                        B      2
```

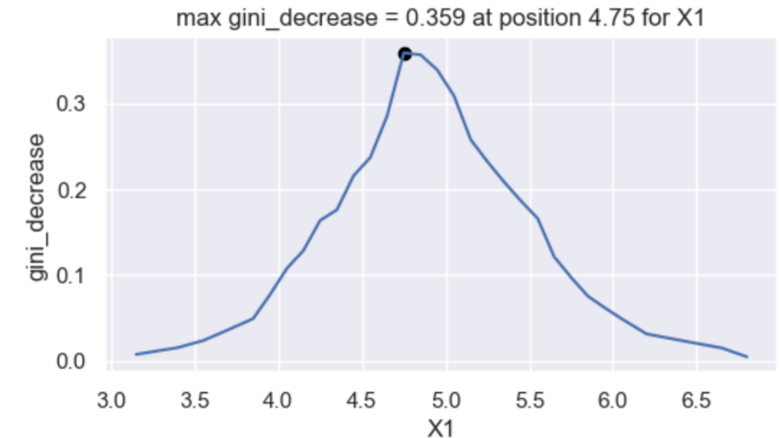
- ein Mass, welchem wir bereits begegnet sind, ist accuracy_score, d.h. die Anzahl korrekt zugeordneter gegenüber allen Instanzen, in diesem Zahlenbeispiel

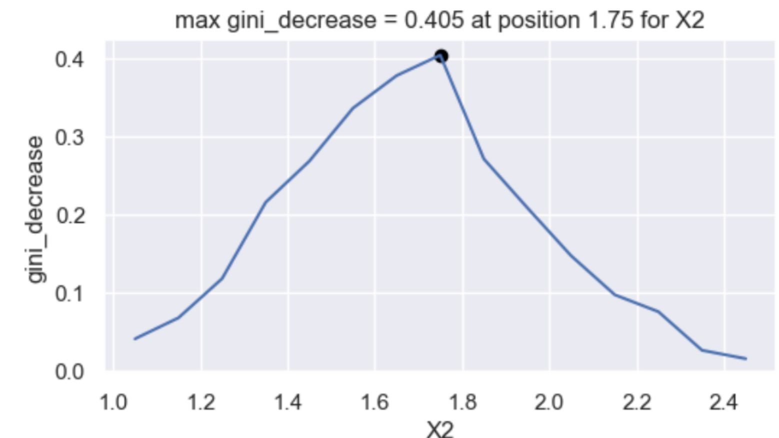
```
pred = np.where(X_demo['X1'] >= 5, 'A', 'B')
from sklearn.metrics import accuracy_score
accuracy_score(pred, y_demo)
```

```
0.9135802469135802
```

Decision Tree Classifier: Splitting Beispiel

- 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	✓

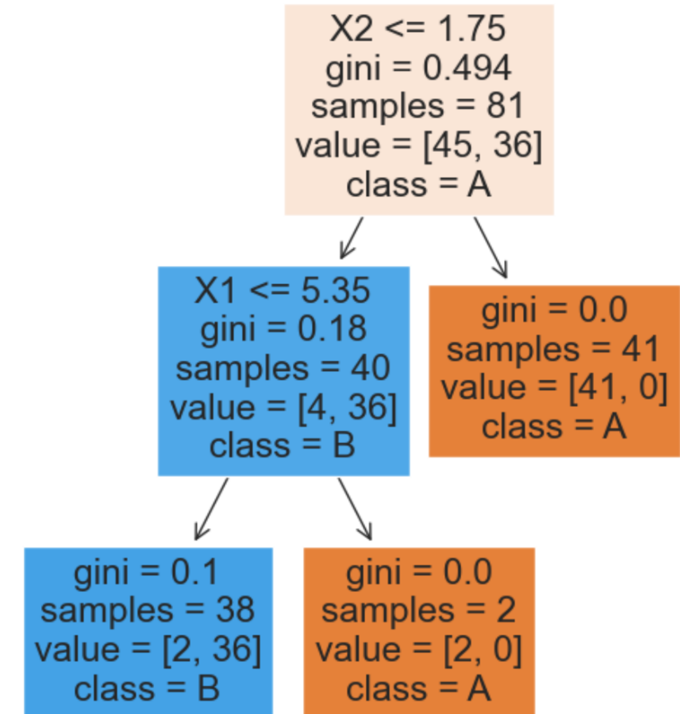
- als erster Split wird hier die Position 1.75 auf dem Feature X2 identifiziert

Decision Tree Classifier: Splitting Beispiel

- 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
```



Decision Tree Classifier: Splitting Beispiel

- 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

X2 <= 1.75
gini = 0.494
samples = 81
value = [45, 36]
class = A

Decision Tree Classifier: 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,

Decision Tree Classifier: 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
```

Decision Tree Classifier: 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
```

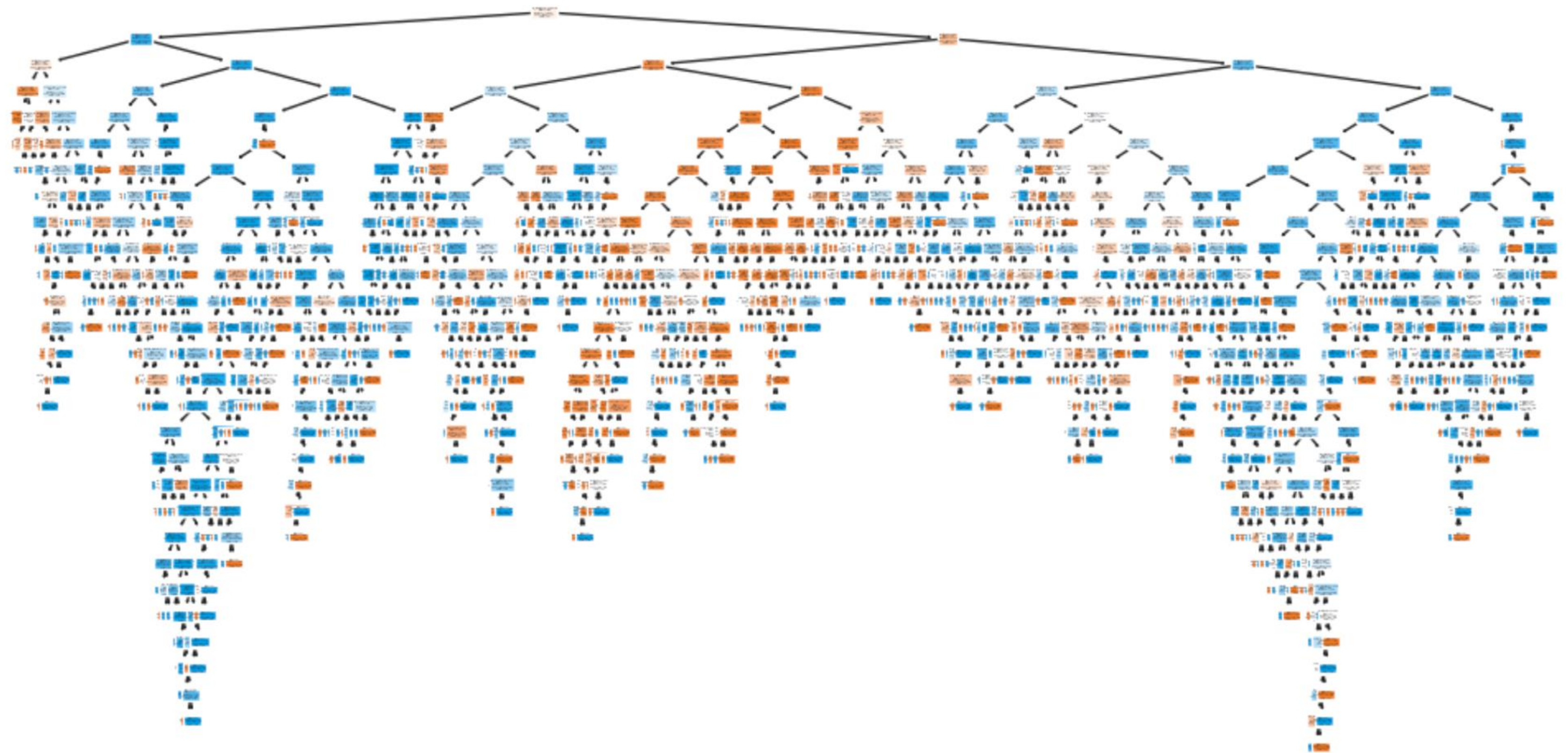
(!/?)

davon später mehr

- Accuracy ist schon klar besser als bei der vorigen Methode

Decision Tree Classifier: 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)



Decision Tree Classifier: 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)
- 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"

Decision Tree Classifier: Hyperparameter

- Ein Decision Tree hat mehrere Hyperparameter, die steuern, wie gross, tief und k omplex der Baum wird.
- Diese Parameter bestimmen direkt, ob der Baum überfitten, unterfitten oder opti mal generalisieren wird.

- **max_depth**: Begrenzt die maximale Tiefe des Baums.
  - Wirkung: Flache Bäume → weniger Overfitting; tiefe Bäume → mehr Overfitting.
  - Quelle: Tiefe beeinflusst Overfitting stark.
- **min_samples_split**: Minimale Anzahl Samples, um einen Knoten zu splitten.
  - Wirkung: Höhere Werte → weniger, aber stabilere Splits.
  - Empfehlung: 2–10, abhängig von Datengröße.

Decision Tree Classifier: Hyperparameter

- **min_samples_leaf**: Minimale Anzahl Samples in einem Blatt.
  - Wirkung: Höhere Werte → glattere Entscheidungsgrenzen, weniger Overfitting.
  - Empfehlung: 1–5.
- **min_impurity_decrease**
  - Ein Split wird nur durchgeführt, wenn er die Reinheit genügend verbessert.
  - Verhindert unwichtige Splits
  - Gut für grosse Datensätze
- **max_leaf_nodes**
  - Begrenzt die Anzahl Blätter.
  - Gut für kompakte Modelle
  - Verhindert extreme Tiefe
- **Criterion**: Metrik zur Split-Bewertung: gini, entropy.

Decision Tree Classifier: Praxis - Hyperparameter

- 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

Decision Tree Classifier: Praxis - Hyperparameter

- 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))
```

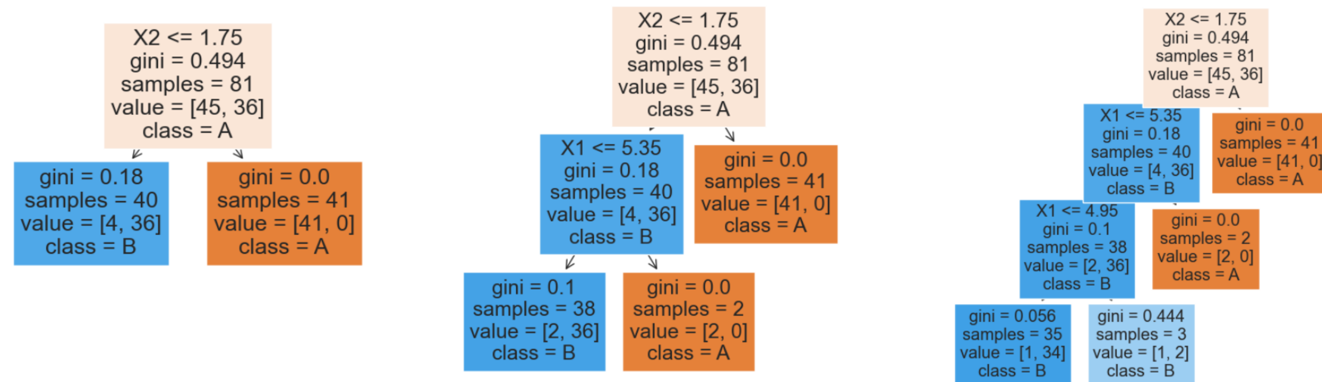
- was zu folgenden Ergebnissen führt

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

Decision Tree Classifier: Praxis – Hyperparameter max-deph

- 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)

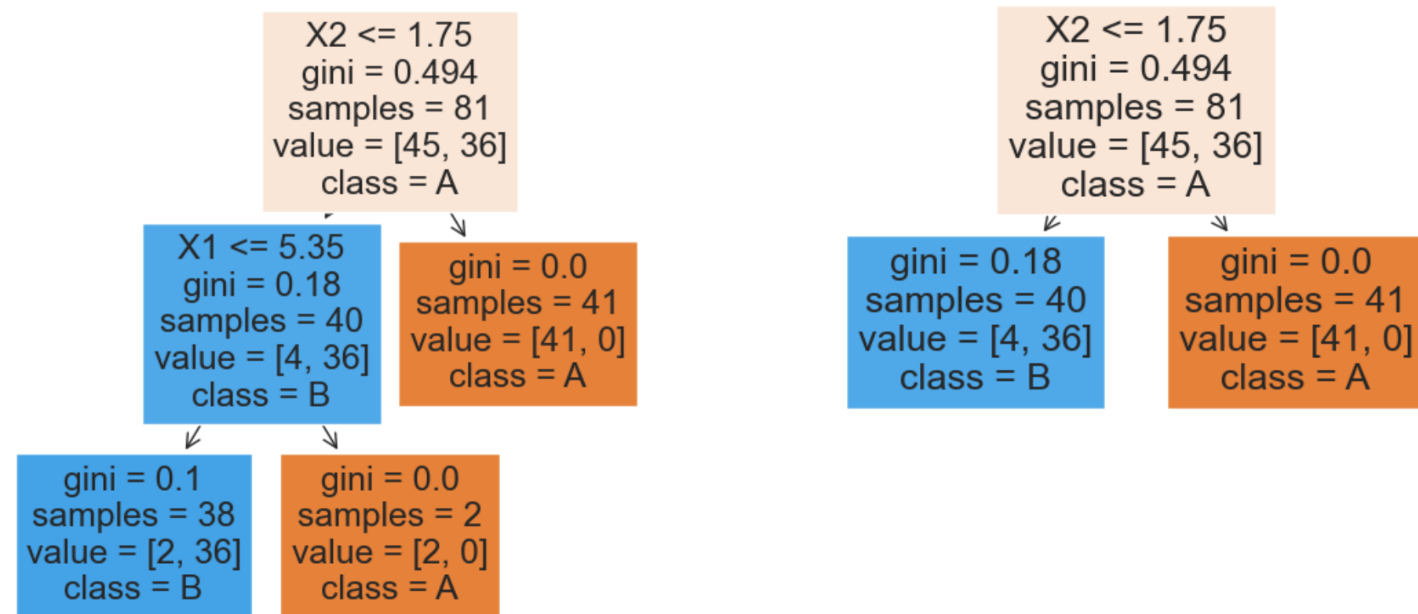
Decision Tree Classifier: Praxis – Hyperparameter max-deph

- 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

Decision Tree Classifier: Praxis – Hyperparameter 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



Decision Tree Classifier: Praxis – Hyperparameter 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

Decision Tree Classifier: Praxis – Hyperparameter 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

Decision Tree Classifier: Praxis – Hyperparameter max_leaf_nodes

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

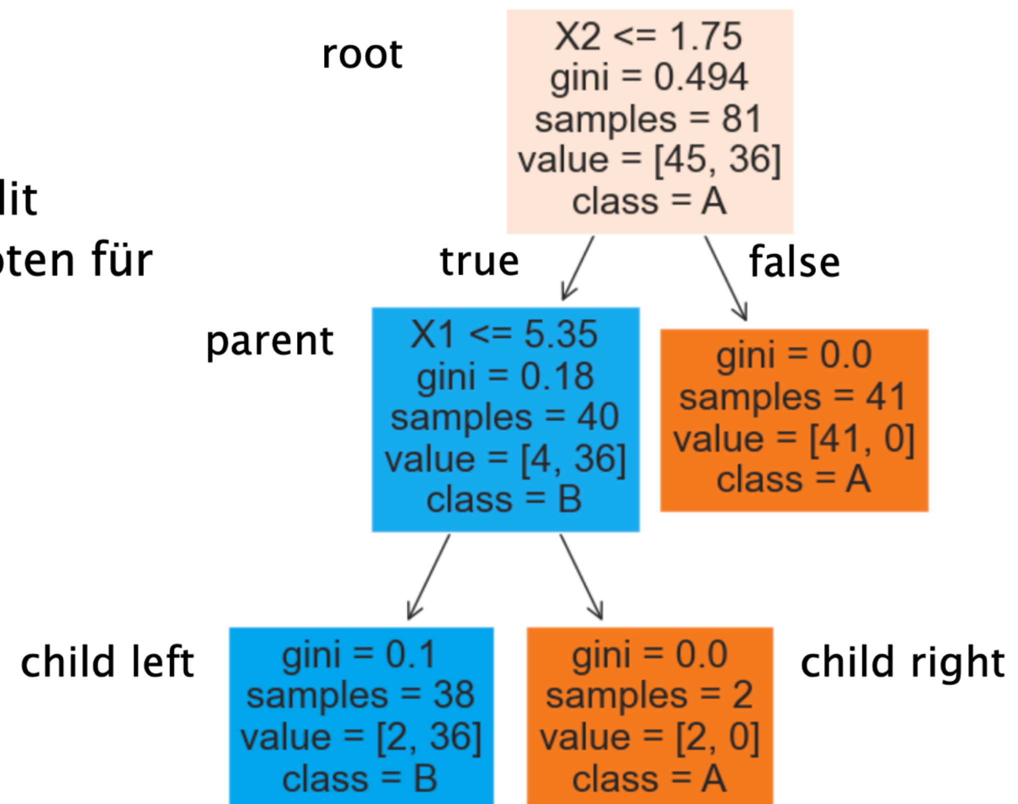
Decision Tree Classifier: Praxis – Hyperparameter min_impurity_decrease

- *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

Decision Tree Classifier: Praxis – Hyperparameter 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



Decision Tree Classifier: Praxis – Hyperparameter min_impurity_decrease

- Vorgehen Schritt für Schritt
  - berechnen des Gini-Index für den Parent-Knoten: $2*4/40*36/40=0.18$,
  - 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
  - berechnen der Differenz zwischen 1. und 2.
  - 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

Decision Tree Classifier: Parameter Tuning

- die oben eingesetzten Parameterwerte sind willkürlich
- durch systematisches Austesten kann der optimale Wert ermittelt werden
- dazu wieder etwas Code:

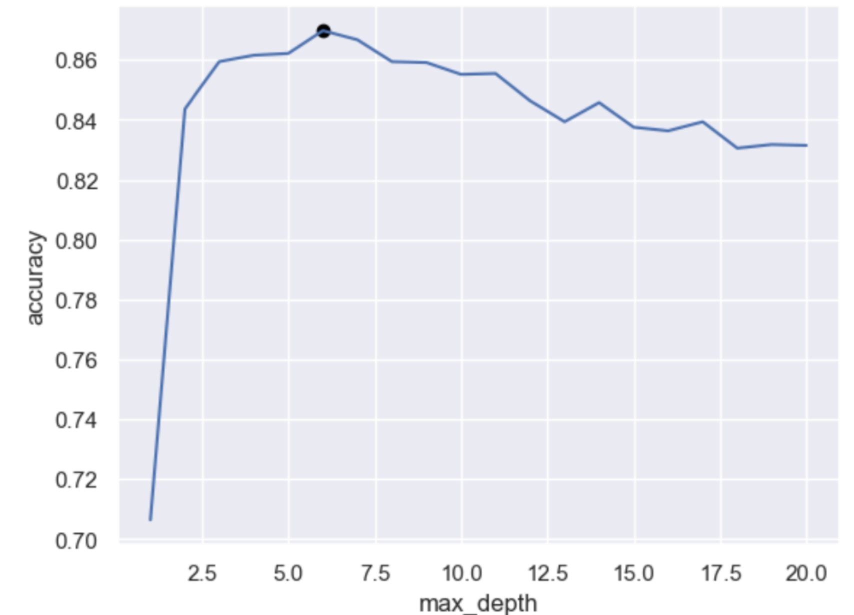
```
## prepare loop
model = DecisionTreeClassifier()
scores = [] ## empty list for collect iteration scorers
params = range(1, 21) ## define range

## 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
```

Decision Tree Classifier: 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



Decision Tree Classifier: Parameter Tuning

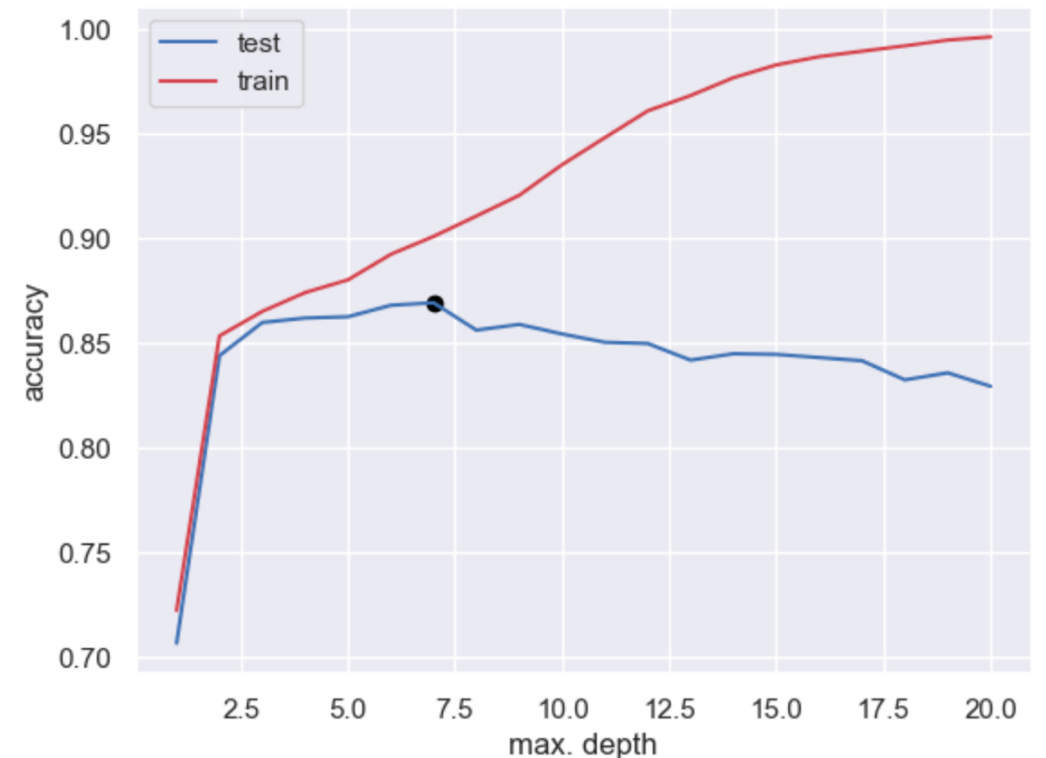
- mit dem für die Visualisierung eingesetzten Ausdruck `scores.index(max(scores))` kann ausserdem der Listenindex für den grössten Scorewert ermittelt werden, was es erlaubt, Score und Parameterwert des Maximums in der Konsole auszugeben

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

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

Decision Tree Classifier: Overfitting

- 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]



Decision Tree Classifier: Overfitting

- 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

Decision Tree Classifier: Feature Importance

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

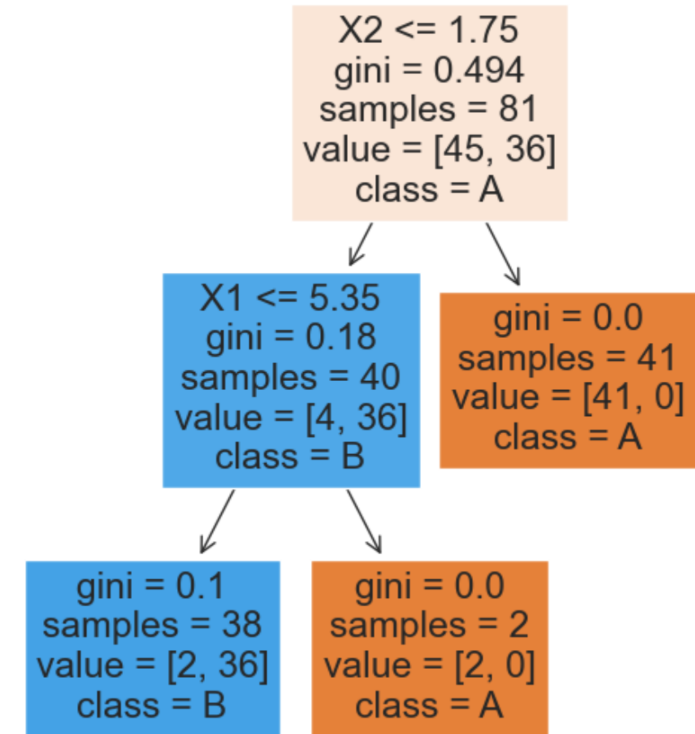
```
model = DecisionTreeClassifier()
model.fit(X_train, y_train)
```

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

Decision Tree Classifier: 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



Decision Tree Classifier: Feature Importance

- eine zentrale Rolle bei der Beurteilung von einzelnen Splits spielt bekanntlich impurity_decrease, 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

Decision Tree Classifier: Feature Importance

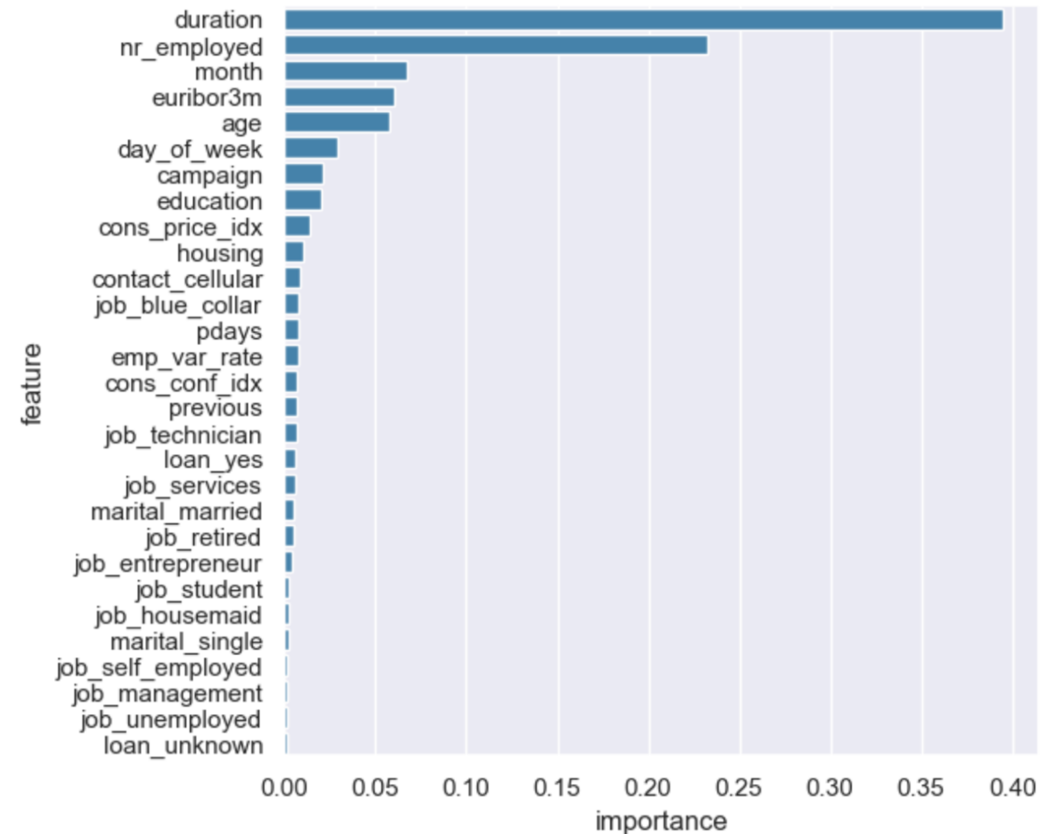
- Ausgabe von feature_importances_, kombiniert mit den Feature Namen:

```
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
:
```

Decision Tree Classifier: 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



Workshop 05

- Gruppen zu 2 bis 4, Zeit: 60-90'
- 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

RandomForest Tree

- in Random Forest ist ein Ensemble-Lernverfahren, das aus vielen einzelnen Entscheidungsbäumen besteht.
- Jeder Baum trifft eine Vorhersage, und der Random Forest kombiniert diese zu einer robusteren, stabileren und genaueren Gesamtvorhersage.
- Ein Random Forest ist eine Sammlung zufällig trainierter Entscheidungsbäume, deren gemeinsame Entscheidung fast immer besser ist als die eines einzelnen Baums.

RandomForest Tree

- 1. Bootstrap-Sampling
  - Für jeden Baum wird ein zufälliges Teil-Dataset gezogen (mit Zurücklegen). → Jeder Baum sieht leicht andere Daten.
- 2. Random Feature Selection
  - Bei jedem Split darf der Baum nur aus einer zufälligen Auswahl von Features wählen. → Bäume werden unterschiedlich und weniger korreliert.
- 3. Viele Bäume trainieren
  - Typisch: 100–1000 Bäume. → Jeder Baum ist ein schwacher Lerner, aber gemeinsam stark.
- 4. Aggregation der Vorhersagen
  - Klassifikation: Mehrheitsentscheid (Voting)
  - Regression: Durchschnitt aller Baumvorhersagen
- → Das Ensemble glättet Fehler einzelner Bäume.

RandomForest Tree: Hyperparameter

- **1. n_estimators**
  - Anzahl der Entscheidungsbäume im Wald.
  - Mehr Bäume → stabiler, genauer
  - Zu viele Bäume → langsamer, aber selten schlechter Typisch: 100–1000
- **2. max_depth**
  - Maximale Tiefe jedes einzelnen Baums.
  - Klein → verhindert Overfitting
  - Gross → jeder Baum wird komplexer Typisch: 5–30
- **3. max_features**
  - Wie viele Features dürfen pro Split betrachtet werden?
  - Klassifikation: oft "sqrt"
  - Regression: oft "log2" oder "auto" Steuert die **Zufälligkeit** und reduziert Korrelation zwischen Bäumen.

Random Tree: Hyperparameter

- **4. min_samples_split**
  - Minimale Anzahl Samples, die nötig sind, um einen Split zu machen.
  - Höher → glattere, stabilere Bäume
  - Tiefer → mehr Splits, riskanter
- **5. min_samples_leaf**
  - Minimale Anzahl Samples in einem Blatt.
  - Höher → robust gegen Rauschen
  - Tiefer → feinere, aber instabile Blätter Sehr wichtig bei unbalancierten Daten.
- **6. bootstrap**
  - Ob Bootstrap-Sampling verwendet wird.
  - True → jeder Baum sieht zufällige Daten (Standard)
  - False → alle Bäume sehen dieselben Daten (selten sinnvoll)

Random Tree: Hyperparameter

- **7. criterion**
  - Reinheitsmass für Splits:
  - Klassifikation: "gini" oder "entropy"
  - Regression: "squared_error" oder "absolute_error"
- **8. max_leaf_nodes**
  - Begrenzt die Anzahl Blätter pro Baum.
  - Gut für kompakte Modelle
  - Verhindert extreme Tiefe
- **9. oob_score**
  - Out-of-Bag-Validierung aktivieren.
  - Liefert eine eingebaute Testgenauigkeit
  - Spart einen separaten Validierungssplit

Random Tree: Hyperparameter

Hyperparameter	Wirkung	Risiko
n_estimators	mehr Bäume → stabiler	zu viele → langsam
max_depth	begrenzt Komplexität	zu klein → Underfitting
max_features	erhöht Diversität	zu klein → schlechte Splits
min_samples_leaf	glättet Modell	zu gross → zu grob
bootstrap	reduziert Korrelation	False → schlechtere Performance

RandomForest Tree: optimale Werte der Hyperparameter

- **1. n_estimators** → ***300–1000*** Mehr Bäume = stabileres Modell.
  - 300 reicht oft
  - 1000 für maximale Stabilität
  - Kaum Risiko von Overfitting, nur Rechenzeit steigt
- **2. max_depth** → ***10–30*** Begrenzt die Komplexität jedes Baums.
  - 10–20 für kleine Datensätze
  - 20–30 für grosse, komplexe Datensätze
  - Zu tief → Overfitting
  - Zu flach → Underfitting
- **3. max_features**
  - Empfohlene Werte:
    - **Klassifikation:** "sqrt"
    - **Regression:** "log2" oder "sqrt"
- Das sorgt für Diversität zwischen den Bäumen → bessere Generalisierung.

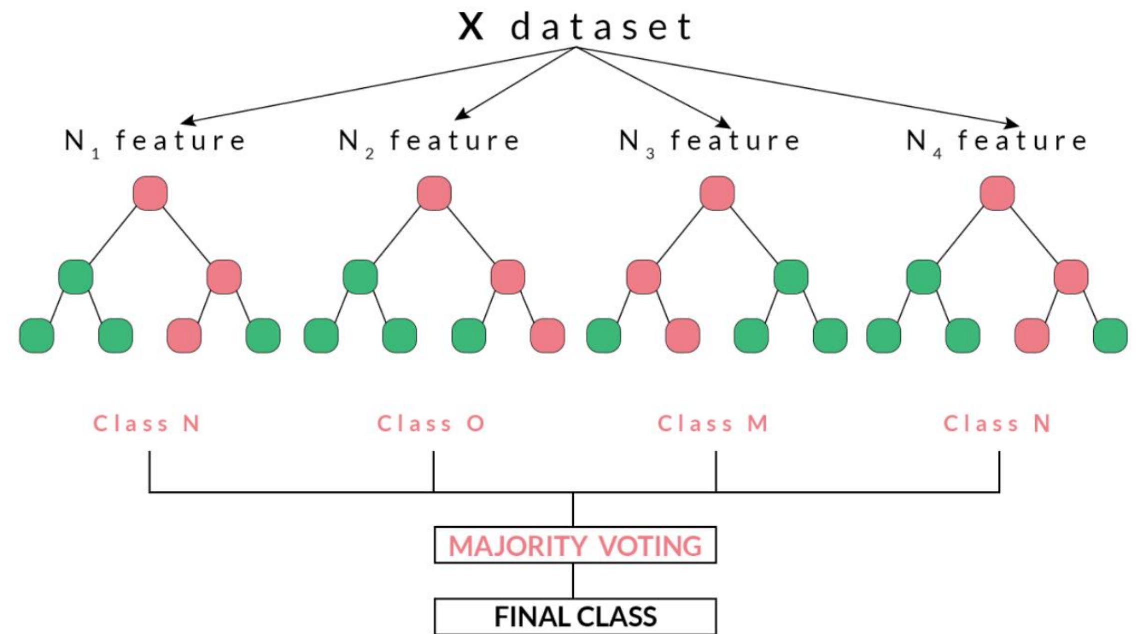
RandomForest Tree: optimale Werte der Hyperparameter

- **4. min_samples_leaf** → ***1–5*** Sehr wichtig für Stabilität.
  - 1 für grosse Datenmengen
  - 2–5 für verrauschte oder unbalancierte Daten
  - Höher = glattere Entscheidungsgrenzen
- **5. min_samples_split** → ***2–10***
  - 2 = Standard
  - 5–10 = verhindert Splits auf winzigen Gruppen
- **6. bootstrap** → ***True***
  - Standard und fast immer optimal. Sorgt für zufällige Trainingsdaten pro Baum.
- **7. ccp_alpha** → ***0***
  - Random Forests brauchen **kein Pruning**, weil das Ensemble selbst regularisiert. Nur bei extrem kleinen Datensätzen sinnvoll.

RandomForest Tree versus Entscheidungsbaum

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)



Decision Tree Classifier vs Random Forest

- Decision Tree, wenn …
  - für ein leicht erklärbares Modell (z. B. Medizin, Recht, Business).
  - die Daten einfach sind und Overfitting kontrolliert werden kann.
  - schnelle Trainings- und Vorhersagezeiten wichtig sind.

- Random Forest, wenn …
  - Für maximale Genauigkeit
  - die Daten komplex, verrauscht oder hochdimensional sind.
  - Robustheit und geringe Varianz entscheidend sind.
  - Overfitting unbedingt vermieden werden soll.

RandomForest Tree : 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,...
```

RandomForest Tree : 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

RandomForest Tree : 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
```

RandomForest Tree : Praxis

- da für ieden Baum nur ein Teil der Beobachtungen verwendet wird 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_)
```

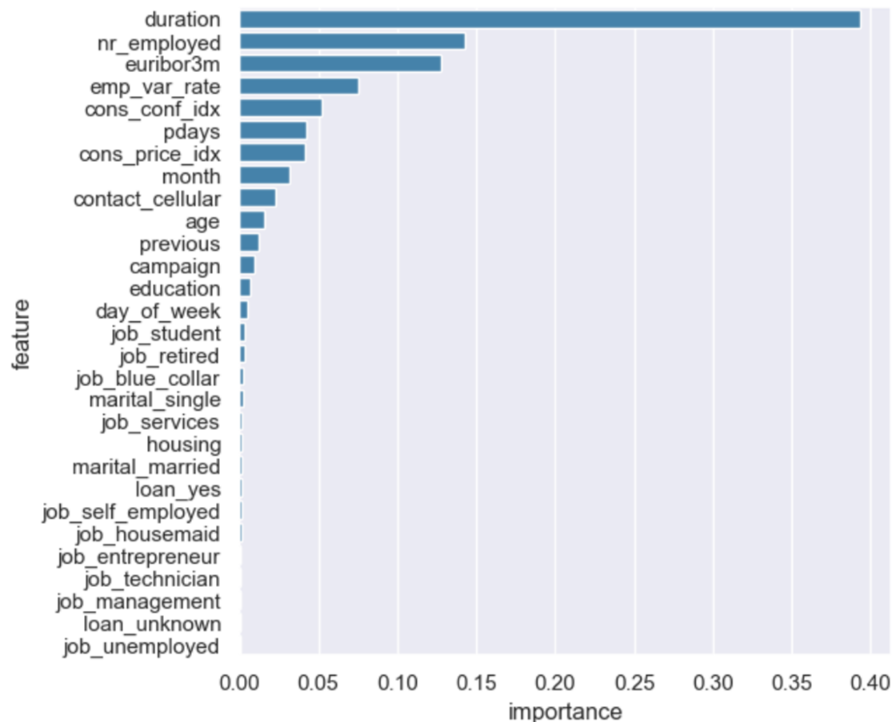
```
model.oob_score_: 0.8869618134793854
```

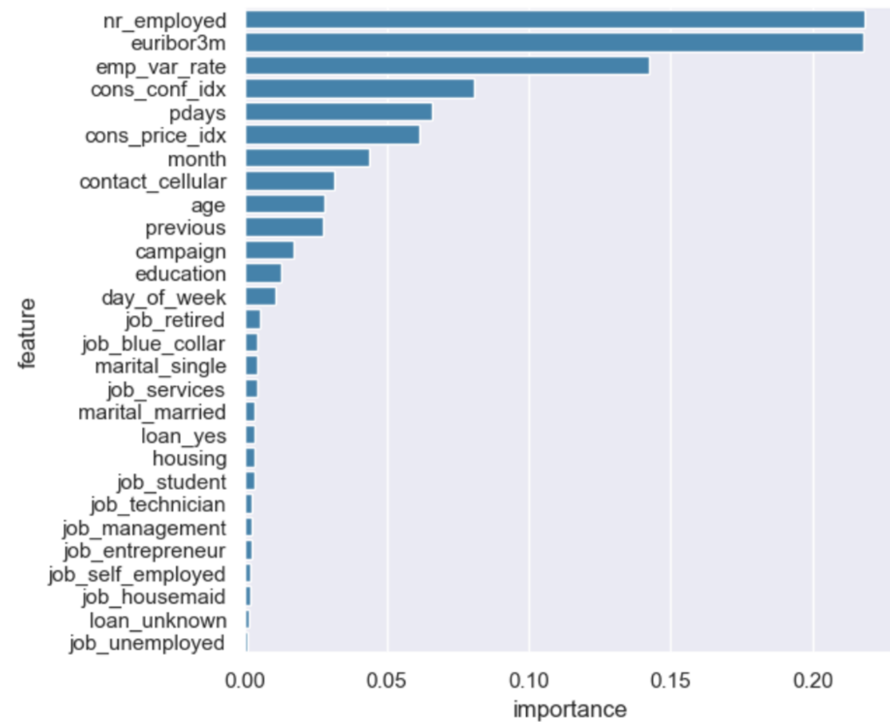
RandomForest Tree : Praxis – Parameter Tuning

- 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

RandomForest Tree : Praxis – 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





RandomForest Tree : Praxis - 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

Weitere Classifier

- AdaBoostClassifier
- GradientBoostingClassifier
- HistGradientBoostingClassifier

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),
:
]
```

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

Klassifikation - Regelbasiert - Modellvergleiche

```python
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:

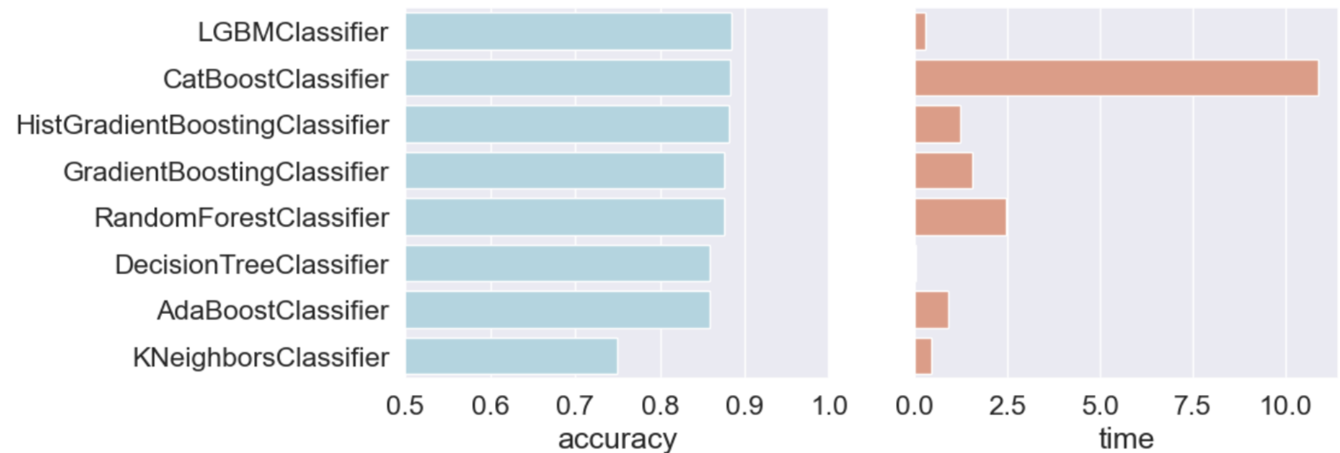
```python
print('%-32s %6.4f %6.3f' % (name, score, t)) ## console output
```

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



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 : 0.8850015211439002
used_time  : 0.31695556640625
```

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

Workshop 06

- 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