Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Machine Learning Algorithmen: Grundlagen

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

kNN: k Nearest Neighbors

k-Nearest Neighbors (kNN) ist einer der intuitivsten Algorithmen im überwachten Lernen. Wie funktioniert kNN?

1. Distanz messen
   - Für einen neuen Punkt wird die Distanz zu allen Trainingspunkten berechnet. Typische Distanzmasse:
     - Euklidische Distanz
     - Manhattan-Distanz
     - Kosinus-Distanz (oft bei Texten)
2. k nächste Nachbarn auswählen. Beispiel: k = 5 → die 5 ähnlichsten Punkte werden berücksichtigt.
3. Entscheidung treffen
   - Klassifikation: Mehrheit gewinnt
   - Regression: Durchschnitt der Nachbarn

kNN: k Nearest Neighbors

- Mini-Beispiel
  - Wir wollen wissen, ob ein Kunde kauft (1) oder nicht (0).
  - Die 5 nächsten Nachbarn haben die Labels: 1, 1, 1, 0, 1
- Mehrheit = 1 → **kNN sagt: Kunde kauft.**

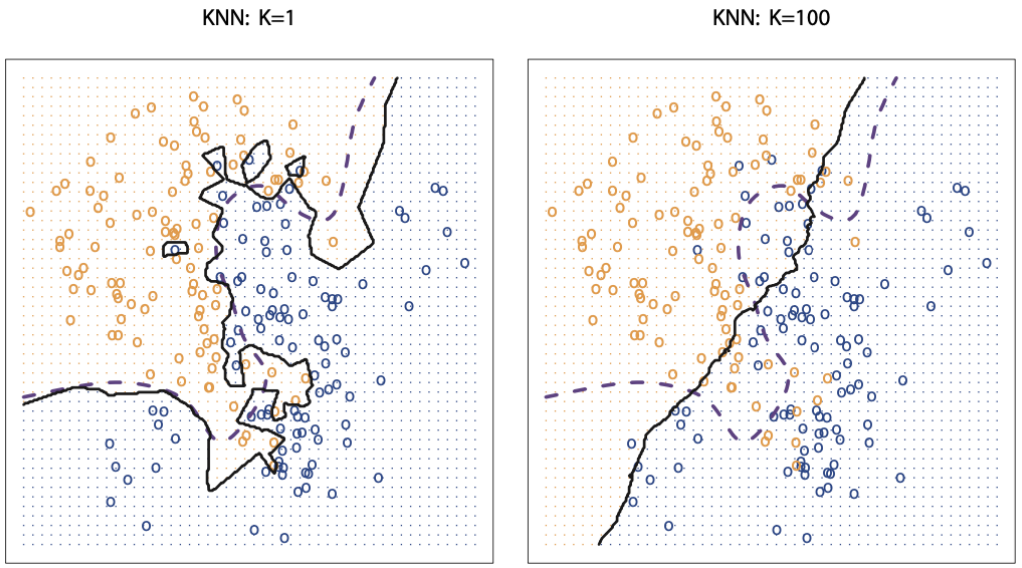
kNN: k Nearest Neighbors

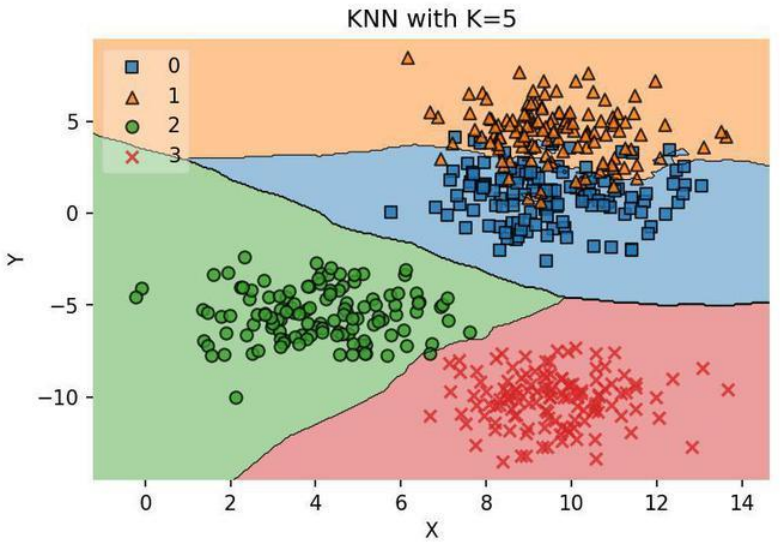
- Die Wahl von k ist einer der wichtigsten Schritte beim k-Nearest-Neighbors-Algori thmus.
- Ein gutes k sorgt dafür, dass kNN stabil, genau und robust funktioniert.
- Ein schlechtes k führt dagegen schnell zu Overfitting oder Underfitting.

- **Grundprinzip**
  - **Kleines k** → Modell reagiert stark auf einzelne Punkte → **Overfitting**
  - **Grosses k** → Modell wird zu grob → **Underfitting**

kNN: k Nearest Neighbors

- **k = 1**: extrem zackig, folgt jedem Ausreisser
- **k = 5**: gute Balance
- **k = 100**: sehr glatt, verliert Details





kNN: k Nearest Neighbors

- k abhängig von Datenmenge
- Grobe Fausregel

Anzahl Datenpunkte	Sinnvoller k-Bereich
< 100	1–5
100–1000	5–15
> 1000	10–50

kNN: k Nearest Neighbors

- Vorteile
  - Einfach zu verstehen
  - Kein Training nötig → Modell = gespeicherte Daten
  - Funktioniert gut bei kleinen Datensätzen
  - Flexibel für Klassifikation und Regression
- Nachteile
  - Langsam bei grossen Datensätzen (muss alle Distanzen berechnen)
  - Sensibel gegenüber irrelevanten Features
  - Funktioniert schlecht in hohen Dimensionen (Curse of Dimensionality)
- kNN ist ein:
  - Lazy Learner (lernt erst bei der Vorhersage)
  - Nicht-parametrisches Modell
  - Algorithmus für Klassifikation und Regression

kNN: k Nearest Neighbors

- kNN wird vor allem dort eingesetzt, wo Aehnlichkeit zwischen Datenpunkten eine grosse Rolle spielt.

Einsatzbereich	Beschreibung	Typische Beispiele
Klassifikation	kNN weist einem neuen Punkt die Klasse seiner k nächsten Nachbarn zu	Handschriftenerkennung, Diagnose (gutartig/boesartig), Kundenklassifikation
Regression	kNN schaetzt einen Wert als Durchschnitt der Nachbarn	Immobilienpreise, Temperaturprognosen, Nachfragevorhersagen
Empfehlungssysteme	Aehnliche Nutzer oder Produkte werden über Distanzmass gefunden	Produkt- oder Filmempfehlungen, Aehnlichkeitsanalysen
Anomalieerkennung	Punkte, die weit weg von allen anderen liegen, gelten als Ausreisser	Kreditkartenbetrug, Netzwerkangriffe, Sensorfehler
Clustering-nahe Aufgaben	kNN nutzt lokale Distanzen, um Gruppenstrukturen zu erkennen	Kundensegmente, geografische Muster
Mustererkennung	Aehnliche Muster werden über Distanzmass identifiziert	Bilder, einfache Formen, Signale

kNN: k Nearest Neighbors

- Er funktioniert am besten, wenn die Daten übersichtlich, niedrigdimensional und gut skaliert sind

Bedingung	Warum geeignet
Kleine bis mittlere Datensaetze	Wenig Rechenaufwand, schnelle Distanzberechnung
Niedrige Dimensionen	Distanzen bleiben aussagekraeftig
Gut skalierte Daten	Verhindert, dass einzelne Features dominieren
Klare Cluster oder Gruppen	kNN nutzt lokale Strukturen optimal
Wenig Rauschen	Weniger Risiko, dass Ausreisser die Nachbarn verfälschen

kNN: k Nearest Neighbors

- Er funktioniert am schechtestens

Problem	Grund
Grosse Datensaetze	Jede Vorhersage braucht Distanz zu allen Punkten → langsam
Hohe Dimensionen	Curse of Dimensionality → Distanzen verlieren Bedeutung
Viele irrelevante Features	kNN ist empfindlich gegenüber Rauschen
Unskalierte Daten	Ein Feature kann die Distanz dominieren

PRAXIS: k Nearest Neighbors

- im Folgenden das Vorgehen mit scikit-learn,
  generell für alle folgenden Klassifikatoren (und Regressoren)
- die einzelnen Schritte
  1. laden der Daten
  2. auftrennen in Feature-Matrix (X) und Target-Vektor (y): Features - Target - Split
  3. auftrennen von X und y in Trainingsset (X_train, y_train) und Testset (X_test, y_test)
  4. Importieren der Trainingsfunktion (aus dem entsprechenden Modul), hier KNeighborsClassifier aus dem Modul neighbors
  5. definieren des zu lernenden Modells mit der gewünschten Parametrisierung

     eigentlich sind das zwei Schritte: Instanziieren und Parametrisieren, welche aber normalerweise in einer Anweisung kombiniert werden (Ausnahme z.B. bei Loops, Parametertuning)
  6. trainieren des Modells
  7. anwenden des Modells auf Testdaten zum Erstellen der Vorhersagen für das Target
  8. evaluieren der Performance

PRAXIS: k Nearest Neighbors

- bei den weiteren Methoden ändern sich dann nur noch
  - der verwendete Klassifikator
  - die anzuwendende Parametrisierung

- aus scikit-learn werden bedarfsweise nur noch die benötigten Klassen resp. Funktionen importiert, und nicht mehr die ganze Library (wie bisher mit pandas, seaborn, etc.)
- streng genommen handelt es sich bei den Learnern (Klassifikatoren und Regressoren) um Klassen im Sinne der Objektorientierten Programmierung (Vorsicht Homonym: nicht zu verwechseln mit den Kategorien im Target, welche auch als "Klassen" bezeichnet werden)
- daher generell folgendes Vorgehen
  - importieren der Klasse
  - instanziieren eines entsprechenden Objekts
  - parametrisieren des Objektes (wobei instanziieren und parametrisieren normalerweise in einer einzigen Anweisung erfolgen)

PRAXIS: k Nearest Neighbors

- **Parameters**: Einstellungen zum Steuern des Trainings (z.B. n_neighbors)
- **Attributes**: Ergebnisse, welche direkt aus dem Modell abgerufen werden können (z.B. .classes_),
  diese stehen aber erst zur Verfügung, nachdem das Modell mit der Methode .fit() trainiert worden ist
- **Methods**: Funktionen, welche auf das Objekt angewendet werden können (z.B. .fit(), .predict(), score(), etc.)

PRAXIS: k Nearest Neighbors

- in der aktuellen Version von scikit-learn (1.4.2) verursacht KNeighborsClassifier möglicherweise eine Warnung, im Sinne von
"Could not find the number of physical cores.."
- diese kann mit folgendem Code unterdrückt werden:

```
import os
os.environ['LOKY_MAX_CPU_COUNT'] = '4'
```

- dieser Code ist in den betreffenden Notebooks gleich am Anfang als Raw eingetragen

PRAXIS: k Nearest Neighbors

1. laden der Daten
   Voraussetzungen:
   - pandas ist als pd importiert
   - der Datenpfad zeigt auf das Verzeichnis, in welchem das Dataset vorliegt

```
bank_df = pd.read_csv('bank_data_prep.csv')
```

2. features - target - split

```
X = bank_df.drop('y', axis=1)
y = bank_df['y']
```

   (auf die hier verwendete Namensgebung wurde schon in der Einführung hingewiesen, trotzdem: X ist eine Matrix, y ein Vektor gemäss Konventionen der Linearen Algebra)

PRAXIS: k Nearest Neighbors

3. train - test - split, unter Verwendung der von scikit-learn zur Verfügung gestellten Funktion
   - dabei wird für die Trainingsdaten ein Anteil von 2/3 festgelegt (train_size) und für die Testdaten die verbleibenden 1/3

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X,                  ## features
    y,                  ## target
    train_size=2/3,     ## proportion of trainint data 2/3
    random_state=1234) ## for reproducibility (optional)
```

   - die Zuordnung der Instanzen erfolgt mittels eines Zufallsprozesses, das optionale Argument random_state sorgt für Reproduzierbarkeit, und ist nur für Lern- oder Demozwecke dienlich
   - bei train_test_split() handelt es sich um eine Funktion, die mehrere Objekte zurückgibt, nämlich doppelt so viele wie sie **Positionsargumente** (hier X und y) übernimmt

PRAXIS: k Nearest Neighbors

4. importieren der Trainingsfunktion (eigentlich Klasse)

```
from sklearn.neighbors import KNeighborsClassifier
```

5. definieren des zu lernenden Modells mit der gewünschten Parametrisierung
   - hier werden vorerst alle Standardparameter des Klassifikators übernommen (z.B. n_neighbors=5, d.h. für die Prediction werden somit 5 nächste Nachbarn berücksichtigt), daher noch keine explizite Parametrisierung

```
model = KNeighborsClassifier()
```

6. trainieren des oben definierten Modells mit den Trainingsdaten
   - in der Konsole wird der Name der Trainingsfunktion angezeigt

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

```
KNeighborsClassifier()
```

PRAXIS: k Nearest Neighbors

- .get_params() zeigt die für dieses Training wirksamen Parameterwerte (Defaultwerte)

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

```
{'algorithm': 'auto', 'leaf_size': 30, 'metric': 'minkowski', 'metric_params':
None, 'n_jobs': None, 'n_neighbors': 5, 'p': 2, 'weights': 'uniform'}
```

- hier von Interesse sind vorläufig
  - n_neighbors:5
  - metric: minkowski
  - p: 2, Euklidische Distanz

PRAXIS: k Nearest Neighbors

7. anwenden des trainierten Modells auf die Testdaten
   - dabei wird die Vorhersage als Vektor-Objekt hinterlegt, um sie anschliessend mit den wahren Werten der Testdaten vergleichen zu können

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

   - obige Anweisung verursacht in der aktuellen Konfiguration möglicherweise einen Fehler: `"'NoneType' object has no attribute 'split'"`
   - dieser kann behoben werden, indem die Library `threadpoolctl` auf die Version 3.1 angehoben wird:

```
!pip install threadpoolctl==3.1.0
```

PRAXIS: k Nearest Neighbors

8. Vergleichen Vorhersage (predictions) mit wahren Targetwerten der Testdaten
   - zuerst als Wahrheitstabelle (confusion matrix)

```
print(pd.crosstab(y_pred, y_test))
```

```
y          no   yes
row_0
no       1384   473
yes       351  1079
```

- Konvention für diesen Kurs: Predictions auf Zeilen, wahre Werte auf Spalten

PRAXIS: k Nearest Neighbors

a. ausgehend von obiger Wahrheitstabelle kann die relative Anzahl der korrekt klassierten Instanzen ermittelt werden

```
print(np.diag(pd.crosstab(y_pred, y_test)).sum() / y_test.size)
```

```
0.7493154852449042
```

b. dasselbe mit importierten Performance Metrik aus scikit-learn:

```
from sklearn.metrics import accuracy_score
print(accuracy_score(y_pred, y_test))
```

```
0.7493154852449042
```

c. und mit der modellinternen Scorer-Methode, welche im Folgenden gleich standardmässig eingesetzt werden soll:

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

```
0.7493154852449042
```

PRAXIS: k Nearest Neighbors

eine Nachbemerkung zu Parametrisieren

- im obigen Codebeispiel wurde für den Klassifikator die Default-Parametrisierung übernommen

```
model = KNeighborsClassifier()
```

- soll dagegen ein anderer Parameterwert verwendet werden, kann dies gleich an dieser Stelle geschehen

```
model = KNeighborsClassifier(n_neighbors=7)
```

- tatsächlich handelt es sich aber um zwei einzelne Anweisungen, welche auch wie folgt formuliert werden können

```
model = KNeighborsClassifier()
model.set_params(n_neighbors=7)
```

- letzteres kann vor allem dann angebracht sein, wenn in einer Iteration unterschiedliche Parameterwerte desselben Modells ausgetestet werden sollen, da parametrisieren weniger ressourcenhungrig ist als instanziieren

PRAXIS: k Nearest Neighbors

Ausblick auf das Thema Validierung - Weitere Performance Metriken

- accuracy_score ist der Standard Scorer bei allen Klassifikatoren von scikit-learn
- soll eine andere Performance Metrik verwendet werden, muss sie aus dem Modul sklearn.metrics dazu importiert und angewendet werden (vgl. oben)
- bei KNeighborsClassifier, aber auch bei allen anderen noch vorzustellenden Methoden, wird mit der Funktion .predict() ein Array mit den vorausgesagten Klassen zurückgegeben
- alternativ kann für jede Methode mit .predict_proba() eine Matrix mit den Wahrscheinlichkeitswerten der jeweiligen Klassen abgerufen werden

PRAXIS: k Nearest Neighbors

Fazit

- die erreichte Performance (ca. 76%) ist noch nicht berauschend
- Verbesserungsmöglichkeiten:
  - die Anzahl Nächste Nachbarn wurde willkürlich auf 5 festgelegt - könnte ev. andere Werte bessere Performance liefern?
  - könnte allenfalls eine Modifikation des Parameters p eine Verbesserung der Performance bringen?
  - die Features wurden (noch) nicht standardisiert

PRAXIS: k Nearest Neighbors

- das systematische Durchtesten von alternativen Parameterwerten bezeichnet man als **Parameter Tuning**
  - auch als Hyperparameter Tuning bezeichnet, da die eingestellten Werte das Lernverhalten der Funktion ändern, und im Idealfall verbessern können
- mit Hilfe eines Loop-Konstrukts (Iterators) können verschiedene Parameterwerte bezüglich deren Auswirkung auf die Performance untersucht werden
- in diesem Beispiel werden k-Werte von 1 bis 10 durchgetestet, die Ergebnisse der einzelnen Iteration werden für anschliessende Analysen in einer Liste "scores" gesammelt

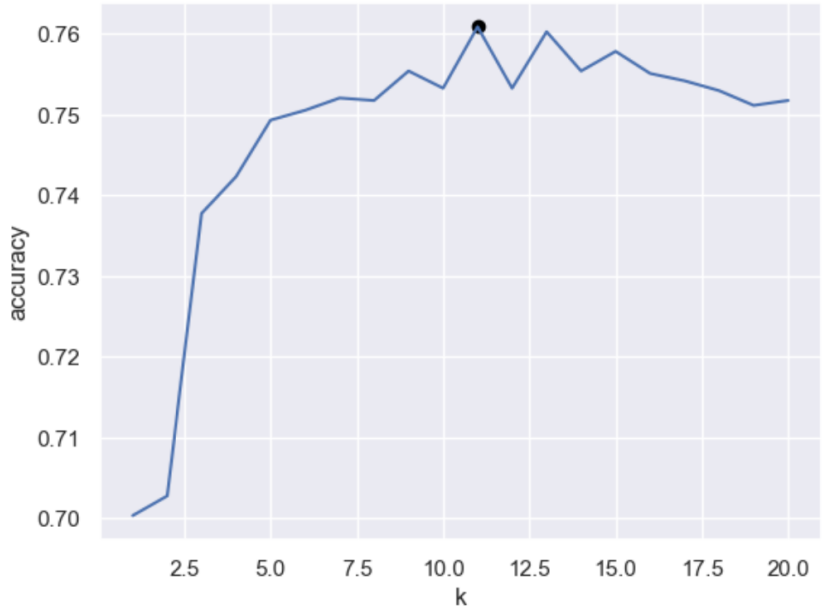
```
model = KNeighborsClassifier()
params = range(1, 21) ## k values as range from 1 to 20 by 1
scores = []           ## empty list for collecting score results by iteration
for param in params:
    model.set_params(n_neighbors=param)
    model.fit(X_train, y_train)
    scores.append(model.score(X_test, y_test))
    print(param, model.score(X_test, y_test)) ## for trace progress only
```

PRAXIS: k Nearest Neighbors

- die letzten Anweisungen im obigen Code können auch in einer kombiniert werden - hier aus praktischen Gründen: Fortschrittsanzeige in der Konsole

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

```
1 0.7003346516580469
2 0.7027684818983876
3 0.7377547916032857
4 0.7423182233039245
5 0.7493154852449042
6 0.7505324003650745
:
```



- Fazit: 11 scheint hier der beste Wert zu sein (jedenfalls im Moment)

PRAXIS: k Nearest Neighbors

- bei gesplitteten Daten stellt sich zusätzlich die Herausforderung, dass beide Teile nach denselben Kriterien standardisiert (oder normalisiert) werden müssen
- dazu zwei Möglichkeiten
  - standardisieren des Ausgangs Dataframes vor dem train - test - split
  - standardisieren der Trainingsdaten und anschliessend anwenden derselben Standardisierung auf die Testdaten
    das wäre dann auch gleich der Weg, wenn mit neuen Daten gearbeitet werden soll

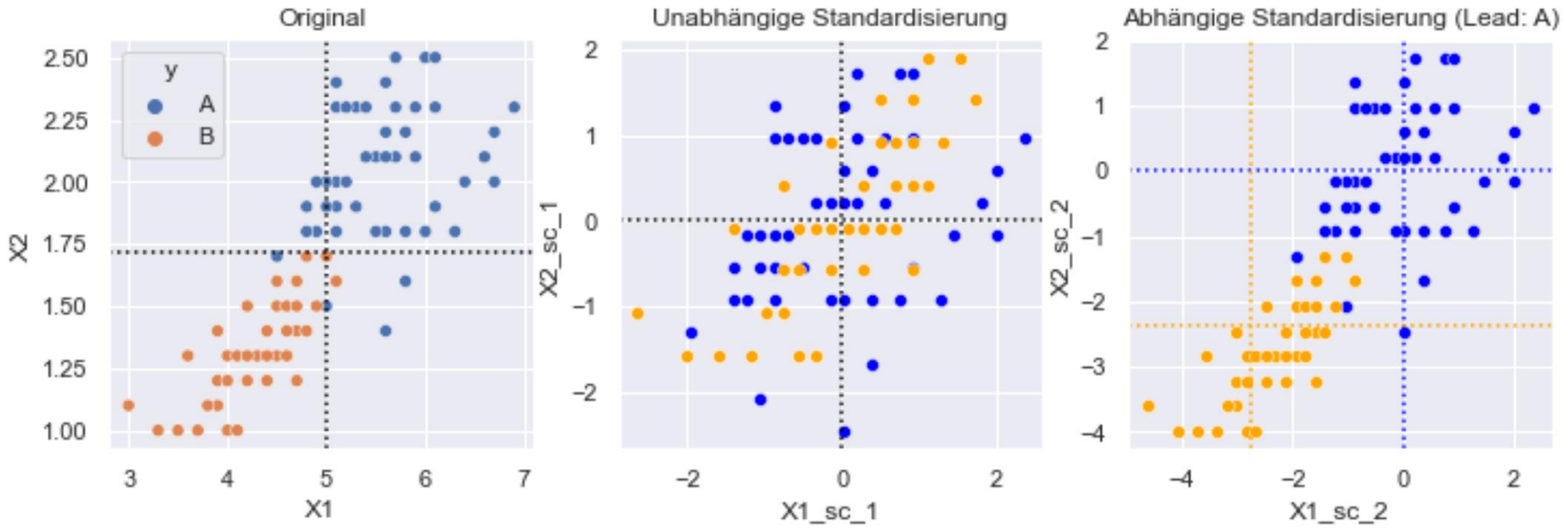
die untenstehende Darstellung demonstriert folgendes:

- auf der Basis von demo_data_class.csv
  - splitten in zwei separate Data Frames für die jeweiligen Klassen
  - Standardisieren der Features im Data Frame der Klasse A
  - Übertragen derselben Standardisierung auf die Features im Data Frame der Klasse B

PRAXIS: k Nearest Neighbors

- Visualisiert wird folgendes (von links nach rechts)
  - das Original
  - Überlagerung nach unabhängigen Standardisierungen
  - Überlagerung nach Standardisieren beider Subsets nach der Vorlage der Gruppe A

die punktierten Linien markieren jeweils die Mittelwerte



PRAXIS: k Nearest Neighbors

- standardisieren mit sklearn.preprocessing.StandardScaler

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler().set_output(transform='pandas')
scaler.fit(X_train)
X_train_sc = scaler.transform(X_train)
X_test_sc = scaler.transform(X_test)
```

- StandardScaler ermittelt für jedes Feature
  - mean_: Mittelwert
  - scale_: Standardabweichung

  mit diesen können dann bei Bedarf neue Daten (mit derselben Anordnung der Features!) nach dem hinterlegten "Rezept" standardisiert werden

PRAXIS: k Nearest Neighbors

Workshop 04

- Gruppen zu 2 bis 4, Zeit: 60'
  - standardisieren Sie die Features von Trainings- und Testdaten mit Hilfe von sklearn.preprocessing.StandardScaler
  - ermitteln Sie anschliessend die besten Parameterwerte für KNeighborsClassifier
    - n_neighbors (1-10)
    - p (z.B. 1, 2, 3)
  - vergleichen Sie die Ergebnisse ohne und mit Standardisieren