Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

2024 FS CAS PML - Supervised Learning
2 Klassifikation
2.1 Instanzbasiert

Werner Dähler 2024

2 Klassifikation

Zweck: trainieren eines möglichst zuverlässigen Vorhersagemodells für ein kategorial skaliertes Target (Label)

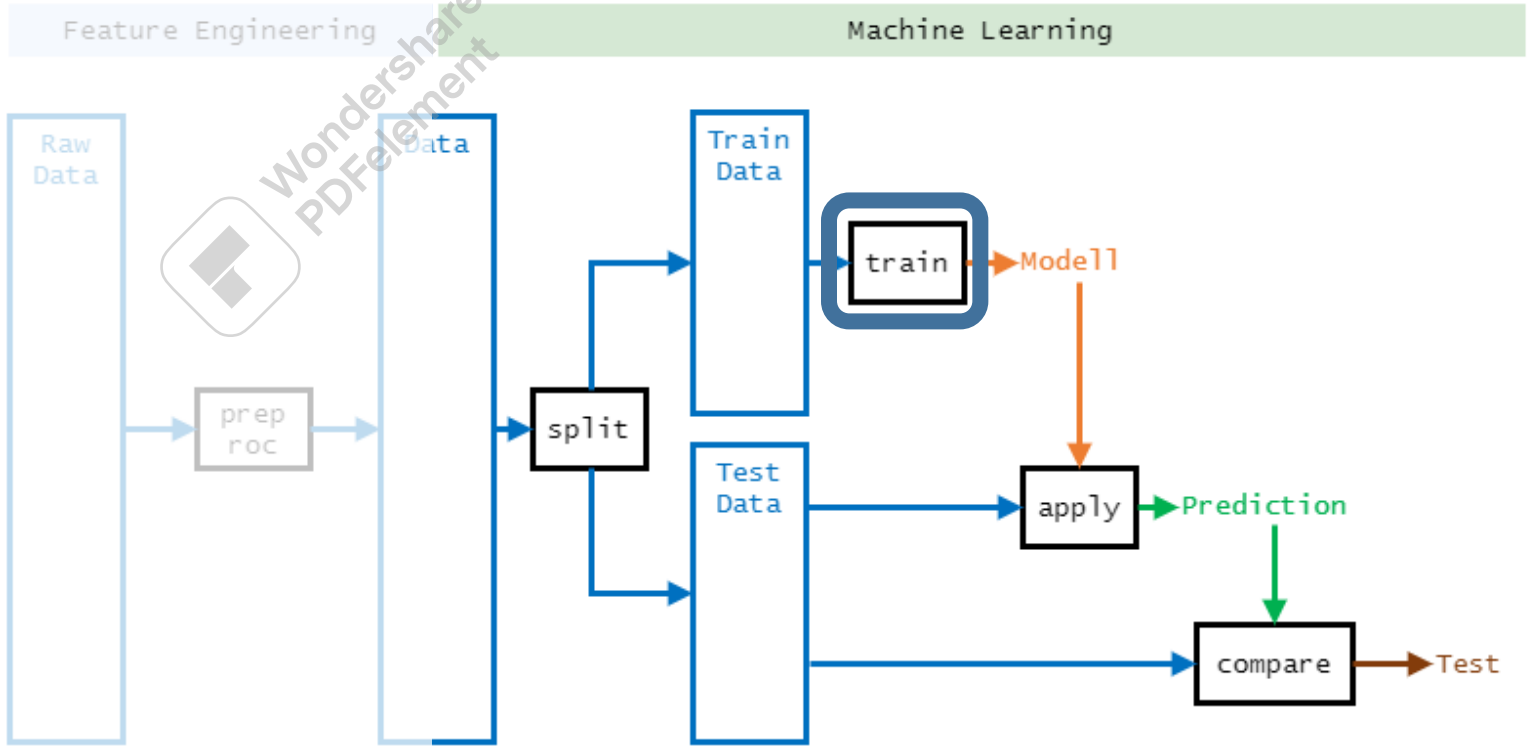
- das Target kann dabei numerisch oder nicht numerisch sein
- die Features müssen den Anforderungen der einzusetzenden Trainingsmethoden entsprechen (alle numerisch codiert, keine NAs), was mit vorgängigem Feature Engineering sicherzustellen ist (vgl. Kap. 1)

- neben dem Trainieren wird im Folgenden auch das Validieren eine wichtige Rolle spielen, d.h. beurteilen, wie "gut" die jeweiligen Lernmethoden sind
(Details zu Validierung dann aber im Kapitel 4)

2 Klassifikation

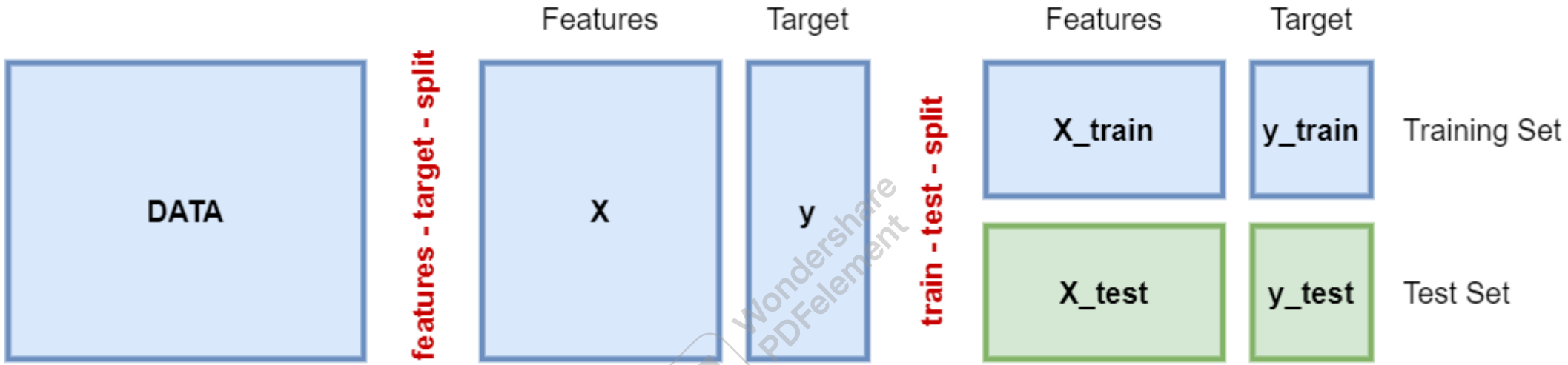
Darstellung einer typischen ML Sequenz aus der Einführung (Kap. 1.1.7)

- Voraussetzung: die Daten wurden mittels Feature Engineering für Machine Learning (insb. Supervised Learning) aufbereitet - und da setzten wir an
- im Zentrum der folgenden Betrachtungen steht dabei insbesondere die Tätigkeit "train", für welche von scikit-learn eine beachtliche Anzahl an Methoden zur Verfügung gestellt wird
- "split", "apply", "compare" kommen natürlich auch zum Einsatz, sind aber in der Anwendung unabhängig von der jeweiligen train-Methode - und werden daher jeweils auf dieselbe Weise eingesetzt



2 Klassifikation

- tatsächlich sind beim Supervised Learning (Klassifikation und Regression) mit scikit-learn zwei aufeinander folgende Splits notwendig

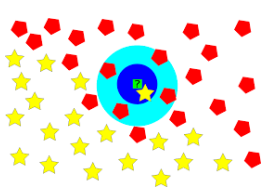


- features - target - split: eine Spezialität von scikit-learn, die Features werden als Matrix (X) erwartet, die Target-Werte als Vektor (y), andere ML Tools (z.B. R) gehen zum Identifizieren von Features und Target andere Wege
- train - test - split: tool-unabhängiges Vorgehen, um sicherzustellen, dass die trainierten Modelle **nicht mit denselben Daten** getestet werden, mit welchen sie trainiert worden waren (vgl. Kap. 4)

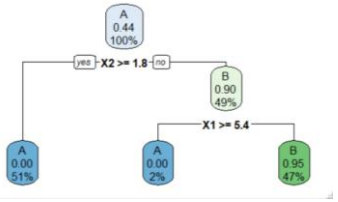
2 Klassifikation

eine Übersicht über die wichtigsten Klassifikationsmethoden (willkürliche Auswahl):

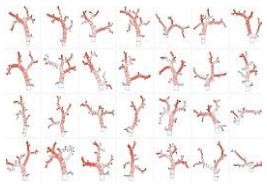
#	Bezeichnung	sklearn.<module>.<funktion>	Typ
1	k-Nächste Nachbarn	neighbors.KNeighborsClassifier	instanzbasiert
2	Entscheidungsbäume	tree.DecisionTreeClassifier	regelbasiert
3	Random Forest	ensemble.RandomForestClassifier	
4	Diskriminanzanalyse	discriminant_analysis.LinearDiscriminantAnalysis	mathematisch
5	Support Vektor Maschinen	svm.SVC	
6	Naive Bayes	naive_bayes.GaussianNB	
7	Logistische Regression	linear_model. LogisticRegression	
8	Neuronale Netze	neural_network.MLPClassifier	neuralnet



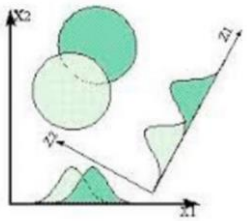
1



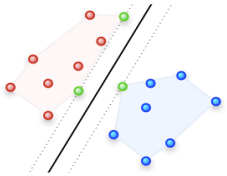
2



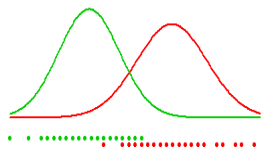
3



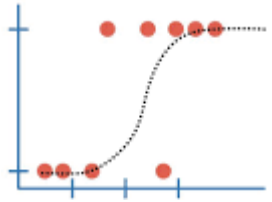
4



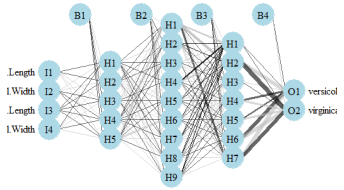
5



6



7



8

2 Klassifikation - AGENDA

21. Instanzbasierte Modelle
    211. KNeighborsClassifier
    212. Weitere Instanzbasierte Methoden
    213. Data Preparation für weitere Klassifikatoren (und Regressoren)
22. Regelbasierte Modelle
23. Mathematische Modelle
24. Neuronale Netze
25. Multiklass Klassifikation

die hinterlegten Links wurden am 27.05.2024 aufgerufen

2.1.1 Klassifikation - Instanzbasiert - KNeighborsClassifier

2.1.1.1 Theorie

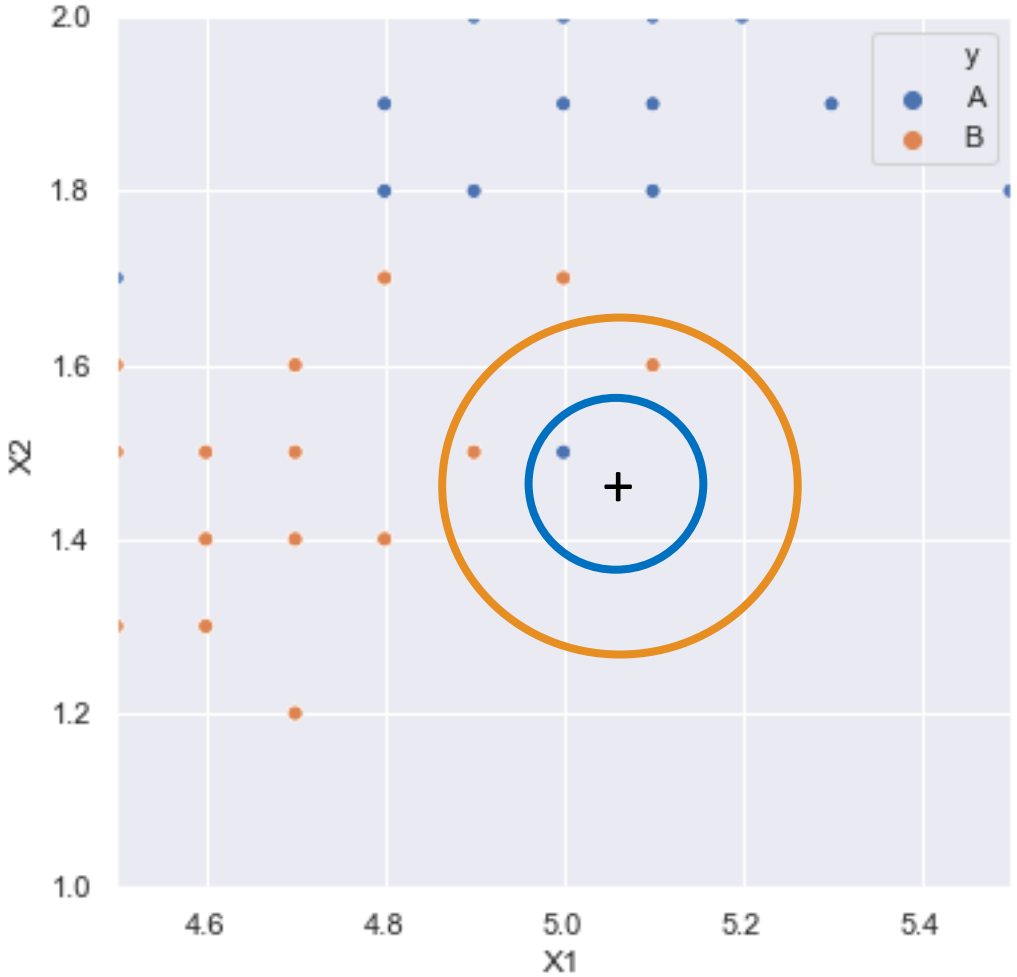
- Instanz basiertes Lernen / Lazy learning: intuitiv verständliche Methode
- im Gegensatz zu den anderen (noch folgenden) Trainingsmethoden erstellt die Methode .fit() neighbors.KNeighborsClassifier kein eigentliches Modell, sondern verwendet die Trainingsdaten direkt für die Vorhersage, das eigentliche Modell sind somit die Trainingsdaten selber
- bei der Anwendung des Modells (prediction) wird jedes Element der Testdaten mit allen Elementen der Trainingsdaten verglichen und die jeweilige Klasse der k ähnlichsten ausgewertet (k wird vorher festgelegt)
- die Klasse, welche in den Trainingsdaten am häufigsten angesprochen wird, "gewinnt" (bei Patt wird hier die erste Instanz aus den Trainingsdaten berücksichtigt, andere Tools wie R geben eine zufällig ausgewählte Klasse zurück)
- als Kriterium für die (multivariate) Ähnlichkeit wird meist die Euklidische Distanz verwendet - eine mehrdimensionale Verallgemeinerung des Satzes von Pythagoras auf n Dimensionen

2.1.1 Klassifikation - Instanzbasiert - KNeighborsClassifier

2.1.1.1 Theorie

nebenstehende Darstellung visualisiert das Vorgehen:

- Trainingsdaten mit
  - zwei Features: X1, X2
  - Target als Klassenbezeichnung "A" oder "B"
- Testdaten (hier nur eine Beobachtung "+")
- aus den Trainingsdaten wird die ähnlichste Beobachtung gesucht und deren Gruppenzugehörigkeit ermittelt
- es können eine oder mehrere "Nächste Nachbarn" aus den Trainingsdaten berücksichtigt werden



2.1.1 Klassifikation - Instanzbasiert - KNeighborsClassifier

2.1.1.1 Theorie

Exkurs: Multidimensionale Ähnlichkeiten

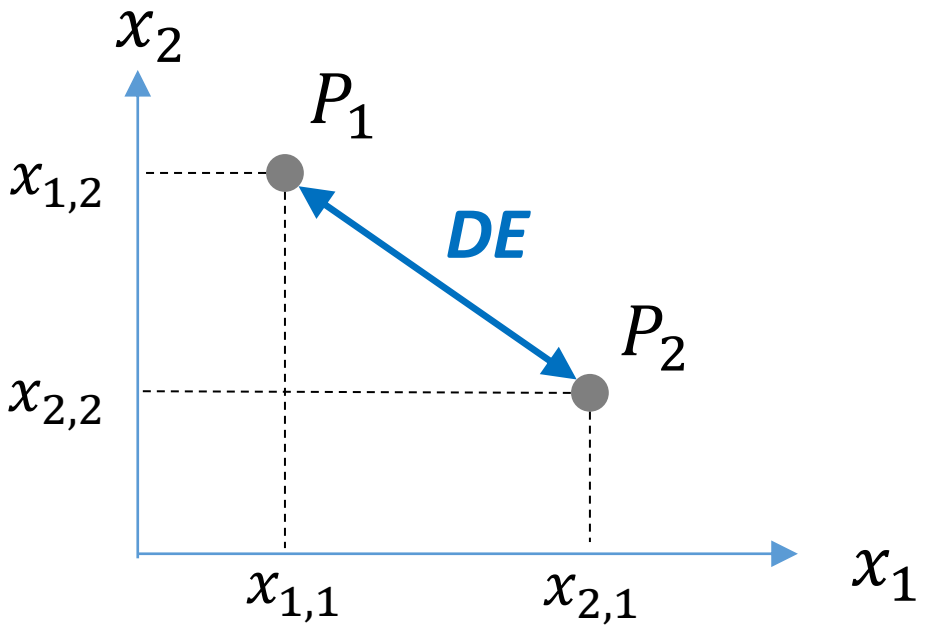
- Analogie aus der Geometrie: die Features werden als Dimensionen in einem höher dimensionalen Raum aufgefasst
- Ähnlichkeiten zwischen Objekten werden als multidimensionale Distanzen von Punktepaaren dargestellt
- am gebräuchlichsten:
  - Euklidische Distanz
  - Manhattan Distanz
- für bis zu zwei (allenfalls drei) Dimensionen ist dies noch verständlich darstellbar, für höhere Dimensionen dagegen nur noch formal

2.1.1 Klassifikation - Instanzbasiert - KNeighborsClassifier

2.1.1.1 Theorie

Multidimensionale Ähnlichkeiten

- Euklidische Distanz
  - zweidimensionale Situation
- für die nebenan dargestellte Situation errechnet sich die Euklidische Distanz DE also nach der Formel: (vgl. Pythagoras)



$$DE_{1,2} = \sqrt{(x_{1,1} - x_{2,1})^2 + (x_{1,2} - x_{2,2})^2}$$

- verallgemeinert auf n-dimensionale Situation:

$$DE_{1,2} = \sqrt[2]{\sum_{i=1}^{n} (x_{1,i} - x_{2,i})^2}$$

2.1.1 Klassifikation - Instanzbasiert - KNeighborsClassifier

2.1.1.1 Theorie

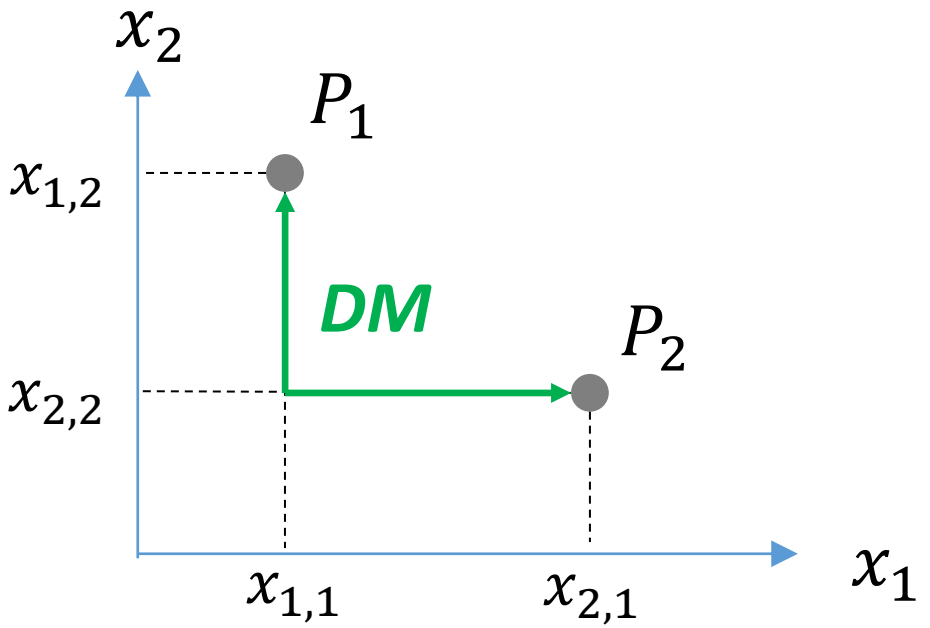
Multidimensionale Ähnlichkeiten

- Manhattan Distanz
  - summiert die **Absolutwerte** der Distanzen über alle Dimensionen

$$DM_{1,2} = |x_{1,1} - x_{2,1}| + |x_{1,2} - x_{2,2}|$$

  - verallgemeinert auf mehrdimensionale Situation:

$$DM_{1,2} = \sum_{i=1}^{n} |x_{1,i} - x_{2,i}|$$



- Verallgemeinerung: Minkovsky-Distanz:
  - p=1 Manhattan Distanz
  - p=2 Euklidische Distanz
  - p muss nicht ganzzahlig sein, aber >= 1

$$d = \sqrt[p]{\sum_{i=1}^{n} \left(|x_{1,i} - x_{2,i}|\right)^p}$$

2.1.1 Klassifikation - Instanzbasiert - KNeighborsClassifier

2.1.1.2 Praxis

- 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

2.1.1 Klassifikation - Instanzbasiert - KNeighborsClassifier

2.1.1.2 Praxis

- 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)

2.1.1 Klassifikation - Instanzbasiert - KNeighborsClassifier

2.1.1.2 Praxis

- ein Blick auf die Online Dokumentation einer solchen Klasse zeigt folgende Elemente
  - **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.)
    dadurch können vergleichbare Basismethoden für alle Learner über denselben Namen angesprochen werden (vgl. Polymorphie, Programmierung)

2.1.1 Klassifikation - Instanzbasiert - KNeighborsClassifier

2.1.1.2 Praxis

- 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

2.1.1 Klassifikation - Instanzbasiert - KNeighborsClassifier

2.1.1.2 Praxis

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)

2.1.1 Klassifikation - Instanzbasiert - KNeighborsClassifier

2.1.1.2 Praxis

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

2.1.1 Klassifikation - Instanzbasiert - KNeighborsClassifier

2.1.1.2 Praxis

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()
```

2.1.1 Klassifikation - Instanzbasiert - KNeighborsClassifier

2.1.1.2 Praxis

- .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
- weitere Informationen zu den Parametern
  - Online Dokumentation
  - Jupyter Notebook: Shift-Tab mit Cursor auf der Klasse zeigt den Docstring der Funktion resp. Klasse
  - JupyterLab: Ctrl-I

2.1.1 Klassifikation - Instanzbasiert - KNeighborsClassifier

2.1.1.2 Praxis

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
```

2.1.1 Klassifikation - Instanzbasiert - KNeighborsClassifier

2.1.1.2 Praxis

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

2.1.1 Klassifikation - Instanzbasiert - KNeighborsClassifier

2.1.1.2 Praxis

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
```

2.1.1 Klassifikation - Instanzbasiert - KNeighborsClassifier

2.1.1.2 Praxis

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

2.1.1 Klassifikation - Instanzbasiert - KNeighborsClassifier

2.1.1.2 Praxis

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 (mehr dazu dann unter dem Thema Validierung - weitere Performance Metriken in Kap. 4.4.1)

2.1.1 Klassifikation - Instanzbasiert - KNeighborsClassifier

2.1.1.2 Praxis

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

2.1.1 Klassifikation - Instanzbasiert - KNeighborsClassifier

2.1.1.3 Parameter Tuning

- 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
```

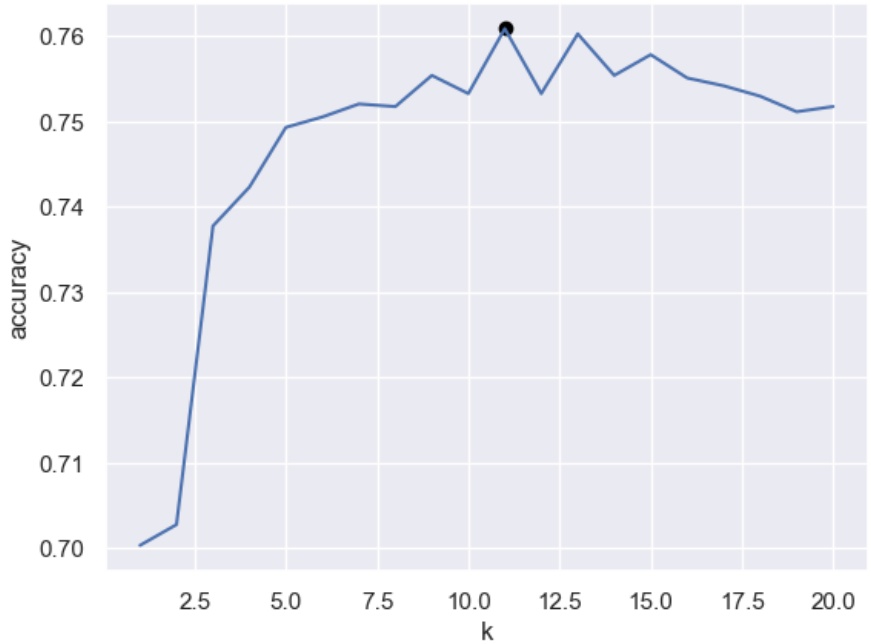
2.1.1 Klassifikation - Instanzbasiert - KNeighborsClassifier

2.1.1.3 Parameter Tuning

- 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)

2.1.1 Klassifikation - Instanzbasiert - KNeighborsClassifier

2.1.1.4 Standardisieren von gesplitteten Daten

- auf Standardisieren (resp. Normalisieren) wurde in Kap. 1.3.3.2 hingewiesen
- 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:

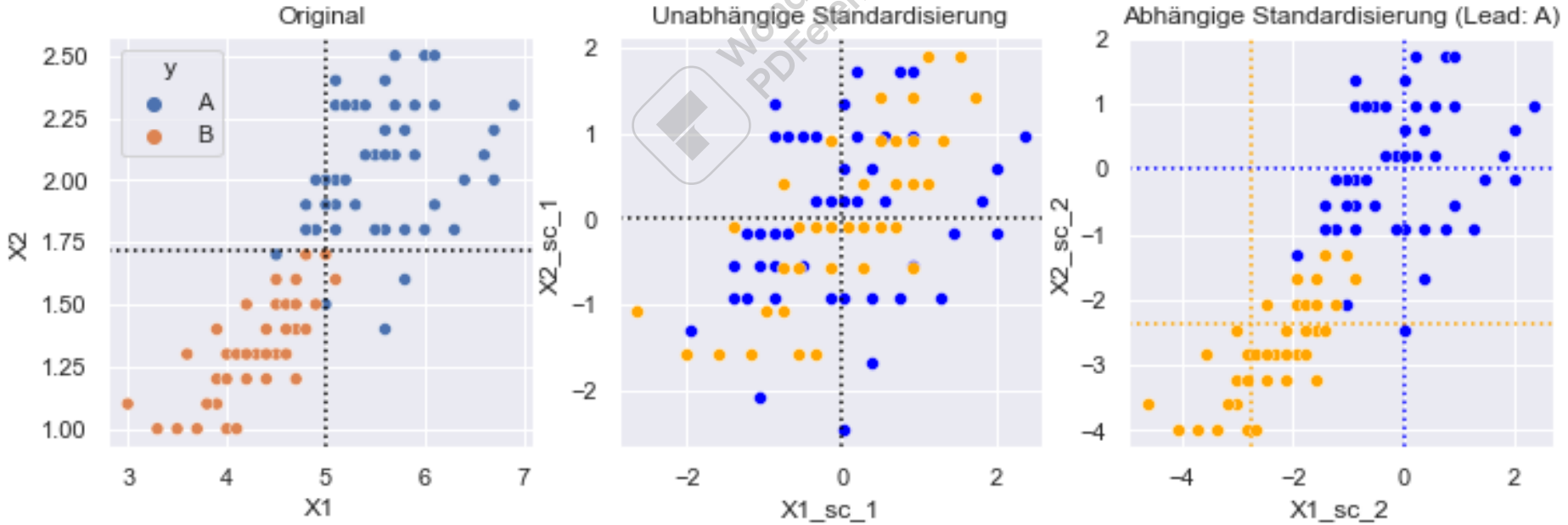
- auf der Basis von demo_data_class.csv
  - splitten in zwei separate Data Frames für die jeweiligen Klassen
  - Standardisieren der Features im Data Frame der Klasse A
  - Übertragen derselben Standardisierung auf die Features im Data Frame der Klasse B

2.1.1 Klassifikation - Instanzbasiert - KNeighborsClassifier

2.1.1.4 Standardisieren von gesplitteten Daten

- 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



2.1.1 Klassifikation - Instanzbasiert - KNeighborsClassifier

2.1.1.4 Standardisieren von gesplitteten Daten

- 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

2.1.2 Klassifikation - Instanzbasiert - Weitere Instanzbasierte Methoden

- neighbors.RadiusNeighborsClassifier()
  - eine Erweiterung von KNeighborsClassifier
  - berücksichtigt für die Prediction alle Instanzen innerhalb eines gewählten Abstandes (Radius) von der neuen Instanz
  - ist daher eher geeignet für dünnbesetzte (sparse) Daten
- neighbors.NearestCentroid()
  - jede Klasse in den Trainingsdaten wird durch deren Centroid repräsentiert (Mittelwert für jedes Feature)
  - weniger anfällig auf Extremwerte
  - sehr schnelle Prediction
- neighbors.KNeighborsRegressor()
  - instanzbasierte Methoden sind auch für Regressionsfragestellungen geeignet, vgl. KNeighborsRegressor in Kap. 3.3.1

2.1.3 Klassifikation - Instanzbasiert - Data Preparation für weitere Klassifikatoren (und Regressoren)

- die drei ersten Schritte zur Datenaufbereitung
  1. laden der Daten
  2. features - target - split
  3. train - test - split

  sind für alle folgenden Klassifikatoren (und Regressoren) identisch
- sie wurde daher in das Notebook übergreifende Modul bfh_cas_pml.py als Funktionen ausgelagert, von wo sie bedarfsweise wie folgt importiert und aufgerufen werden kann

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

- Parameter: dataset name, target name, seed (optional)
- Rückgabewerte: X_train, X_test, y_train, y_test

2.1.3 Klassifikation - Instanzbasiert - Data Preparation für weitere Klassifikatoren (und Regressoren)

- auch für die Aufbereitung der Demodaten steht im Modul eine Funktion zur Verfügung, allerdings ohne den train - test - split:
- importieren und aufrufen wie folgt

```
from bfh_cas_pml import prep_demo_data
X_demo, y_demo = prep_demo_data('demo_data_class.csv', 'y')
```

- Parameter: dataset name, target name
- Rückgabewerte: X_demo, y_demo

- (die beiden Funktionen könnten mit geeigneter Parametrisierung auch in einer zusammengefasst werden, was aber aus Gründen der Transparenz hier unterbleibt)

2.1.1 Klassifikation - Instanzbasiert - KNeighborsClassifier

Workshop 04

Gruppen zu 2 bis 4, Zeit: 30'

- 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