# Data Mining und Maschinelles Lernen

**Prof. Kristian Kersting**
**Steven Lang**
**Johannes Czech**

TECHNISCHE UNIVERSITÄT DARMSTADT

Sommersemester 2021
22. April 2021
Übungsblatt 2

Diese Übung wird am **29.04.2021** um **13:30 Uhr** besprochen und **nicht** bewertet.

**Benötigte Dateien**
Alle benötigten Datensätze und Skriptvorlagen finden Sie in unserem Moodle-Kurs:
https://moodle.informatik.tu-darmstadt.de/course/view.php?id=1058

## 2.1 Kreuzvalidierung

Bei der Evaluierung eines Modells im maschinellen Lernen sind wir daran interessiert, wie gut das Modell auf ungesehenen Daten funktioniert, wenn wir es zur Vorhersage neuer Stichproben verwenden wollen. Daher ist es notwendig, die uns bereits vorliegenden Daten in einen Trainingssatz und einen Testsatz aufzuteilen. Das Modell wird auf dem Trainingssatz trainiert und auf dem Testsatz ausgewertet. Dies kann auf verschiedene Weise erfolgen, wie im Folgenden erläutert wird.

Das Problem bei der einfachen Train-Test-Aufteilung besteht darin, dass sie nur einen Bruchteil von $1 - p$ des vollständigen Datensatzes zur Bewertung der Modellleistung verwendet. Dies könnte problematisch werden, wenn der Datensatz selbst bereits recht klein ist. Ein kleinerer Testsatz erhöht die Bewertungsvarianz und kann leichter durch Ausreißer beeinflusst werden. Die Lösung für dieses Problem ist die so genannte k-fache Kreuzvalidierung.

a) **Train-Test-Aufteilung**

Implementieren Sie die Methode `train_test_split` und teilen Sie den Datensatz zu 70 % in eine Trainingsmenge und zu 30 % in eine Testmenge auf.

Sie können dabei die Methoden `numpy.random.seed`, `numpy.random.shuffle`, oder `numpy.random.choice` verwenden.

Sofern Sie diese Aufgabe nicht lösen können, verwenden Sie `train_test_split` von *sklearn*.

```python
def train_test_split(X: np.ndarray, y: np.ndarray, test_size: float, random_state: int)\
        -> [np.array, np.array, np.array, np.array]:
    """Splits the given dataset into random train and test subsets.
     The ‘test_size‘ defines the proportion of the test set."""
    assert(len(X) == len(y))
    assert(test_size >= 0)
    assert(test_size <= 1)
    np.random.seed(random_state)
    nb_test_samples = round(len(X) * test_size)
    test_indices = np.zeros(len(X), dtype=np.bool)
    test_indices[:nb_test_samples] = True
    np.random.shuffle(test_indices)
    train_indices = ~test_indices

    return X[train_indices], X[test_indices], y[train_indices], y[test_indices]
```

```python
def train_test_split_v2(X: np.ndarray, y: np.ndarray, test_size: float, random_state: int) \
        -> [np.array, np.array, np.array, np.array]:
    """Splits the given dataset into random train and test subsets.
     The ‘test_size‘ defines the proportion of the test set."""
    assert (len(X) == len(y))
    assert (test_size >= 0)
```

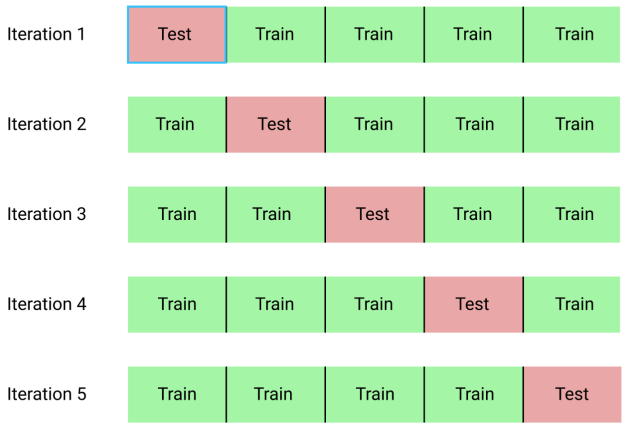


Abbildung 1: Übersicht der k-fachen Kreuzvalidierung für *k* = 5

```
    assert (test_size <= 1)
    test_indices = np.random.choice([True, False], size=len(X), p=[test_size, 1-test_size])
    train_indices = ~test_indices

    return X[train_indices], X[test_indices], y[train_indices], y[test_indices]
```

b) **Beschreibung der k-fachen Kreuzvalidierung**

Beschreiben Sie die Methodik der k-fachen Kreuzvalidierung (engl. k-fold Cross-Validation) und warum diese eine verlässlichere Validierung als ein einfacher Train-Test-Split ermöglicht. Ein Skizze ist hierbei hilfreich.

*Bei dieser Einstellung wird der Datensatz zufällig in k disjunkte Gruppen* $\{\mathbf{X}_0, \ldots, \mathbf{X}_{k-1}\}, \mathbf{X}_0 \cap \cdots \cap \mathbf{X}_{k-1} = \emptyset$ *aufgeteilt. Für jede* $\mathbf{X}_i$ *der Untermengen werden wir ein separates Modell trainieren und evaluieren mit* $\mathbf{X}_{train} = \mathbf{X} \backslash \mathbf{X}_i$ *und* $\mathbf{X}_{test} = \mathbf{X}_i$ *(Hinweis: Es ist üblich, ausgeglichene Aufteilungen zu verwenden, d.h. sicherzustellen, dass jede Aufteilung die gleichen Klassenprioren enthält wie die ursprüngliche Datenmenge). Die resultierende Auswertung kann dann über alle Aufteilungen gemittelt werden. Siehe Abbildung 1 für eine Einteilung mit* $k = 5$ [2].

c) **Weitere Methoden zur Kreuzvalidierung**

Neben der üblichen k-fachen Kreuzvalidierung gibt es auch die folgenden Verfahren: Wiederholte k-fache Kreuzvalidierung (engl. Repeated k-Fold), Eins auslassen (engl. Leave One Out), P% auslassen (engl. Leave P Out) und Gemischte Trennung (engl. Shuffle Split). Beschreiben Sie kurz jeder dieser Techniken.

***Wiederholte k-fache Kreuzvalidierung****: Wiederholen Sie die k-fache Kreuzvalidierung n-mal mit zufälligen Permutationen bei jeder Wiederholung.*

***Eins auslassen****: Jeder Datenpunkt wird als Testsatz verwendet, und ein Modell wird auf allen anderen Datenpunkten trainiert. Dieses Verfahren ist sehr rechenintensiv.*

***P% auslassen****: Jede Gruppe von P-Proben wird als Testsatz verwendet. Für eine Gruppe von n Proben ergibt dies* $\binom{n}{p}$ *Trainings/Test-Splits. Dieses Verfahren ist sehr rechenintensiv.*

***Gemischte Trennung****: Der Datensatz wird nach dem Zufallsprinzip gemischt und in einen Trainings und einen Testsatz aufgeteilt. Dies kommt einem wiederholten Holdout-Verfahren gleich.*

[2] Quelle: `https://towardsdatascience.com/cross-validation-explained-evaluating-estimator-performance-e51e5430ff85`

d) **Durchführung der Kreuzvalidierung**

Implementieren Sie die Funktion `acc_kfold_cross_val`, um die gegebene Kreuzvalidierungsmethode durchzuführen und die mittlere Trainings- und Testgenauigkeit zu errechnen. Verwenden Sie dabei die Methode `sklearn.model_selection.cross_validate`.

```python
def acc_kfold_cross_val(clf: sklearn.base.BaseEstimator, cv: sklearn.model_selection._split._BaseKFold,
                        X: np.ndarray, y: np.ndarray) -> (float, float):
    """Runs cross validation for the given classifier and cross validation method and
     returns the mean train and test accuracy."""
    scores = cross_validate(
        clf,
        X,
        y,
        cv=cv,
        n_jobs=-1,
        scoring=make_scorer(accuracy_score),
        return_train_score=True,
    )
    mean_acc_train = scores['train_score'].mean()
    mean_acc_test = scores['test_score'].mean()
    return mean_acc_train, mean_acc_test
```

e) **Kreuzvalidierung bei großen Datenmengen**

Beurteilen Sie die Verwendung von Kreuzvalidierung bei Datenmengen mit 100, 1000 und einer Millionen Datenpunkten.

*Die Ergebnisse sind je nach der gewählten Kreuzvalidierungsmethode sehr unterschiedlich. Dies wird bei größeren Datensätzen weniger problematisch, da genügend Trainings- und Testdaten zur Verfügung stehen, um den Fehler bei ungesehenen Daten korrekt zu approximieren.*

*Bei kleinen Datenmengen wie 100 ist die Kreuzvalidierung essentiell, da die Varianz der Ergebnisbewertungen sehr hoch ist. Für 1000 Datenpunkte nähern sich die Ergbenisse unterschiedlicher Kreuzvalidierungsmethodiken einander an. Bei sehr großen Datensätzen (z.B. mehr als eine Millionen Dateneinträge) ist die Durchführung einer Kreuzvalidierung meist nicht mehr praktikabel. Allerdings ist hier auch der Testdatensatz größer und damit aussagekräftiger.*

## 2.2 K nächste Nachbarn (kNN)

Die nachbarschaftsbasierte Klassifikation ist eine Art des instanzbasierten oder nicht-generalisierenden Lernens: Es wird nicht versucht, ein allgemeines internes Modell zu konstruieren, sondern es werden lediglich Instanzen der Trainingsdaten gespeichert. Die Klassifikation wird aus einer einfachen Mehrheitsentscheidung der nächsten Nachbarn jedes Punktes berechnet: Einem Abfragepunkt wird die Datenklasse zugewiesen, die innerhalb der nächsten Nachbarn des Punktes die meisten Vertreter hat.

a) **Vorverarbeitung des Datensatzes**

In dieser Aufgabe verwenden wir einen Datensatz über Schwertlilien, bekannt als Iris-Dataset, welcher von dem Biologen Ronald Fisher in dem Jahre 1936 veröffentlicht wurde [1]. Wählen Sie die ersten beiden Merkmale (engl. features): Kelchblattbreite (engl. sepal width) und Kelchblattlänge (engl. sepal length) über die Methode `load_reduced_iris_dataset` aus. Den vollständigen Iris-Datensatz können Sie über die Funktion `sklearn.datasets.load_iris` laden.

```python
def load_reduced_iris_dataset() -> (np.ndarray, np.ndarray):
    """Loads the iris dataset reduced to its first two features (sepal width, sepal length)."""
    iris = datasets.load_iris()
    X = iris.data[:, :2]
    y = iris.target
    return X, y
```

b) **Evaluierung unterschiedlicher k-Werte**

Der einzige und Hauptparameter von kNN ist die Anzahl, wie viele nächste Nachbarn eines Datenpunktes entscheiden, welchem Label der Datenpunkt zugeordnet wird. Evaluieren Sie die Trainings- und Testgenauigkeit für alle k-Werte $k \in [1, 3, 5, \ldots, 99]$ mittels der Funktion `evaluate_ks`. Verwenden Sie dabei die Methode `utils.evaluate` um eine Kreuzvalidierung mit 10 Splits für jeden K-Wert durchzuführen.

```python
def evaluate_ks(ks: range, X: np.ndarray, y: np.ndarray) -> (np.ndarray, np.ndarray):
    """Evaluates each k-value of the ks-array and returns their respective training and test accuracy
    ."""
    accuracies = np.array([utils.evaluate(k, X, y) for k in ks])
    acc_train = accuracies[:, 0]
    acc_test = accuracies[:, 1]
    return acc_train, acc_test
```

c) **Visualisierung der k-Werte**

Visualisieren Sie die k-Werte gegenüber der Traings- und Testgenauigkeit in der Funktion `plot_k_to_acc`. Der k-Wert wird auf die X-Achse und die Genauigkeit auf die Y-Achse abgebildet. Beschriften Sie ebenfalls beide Achsen.

```python
def plot_k_to_acc(ks: range, acc_train: np.ndarray, acc_test: np.ndarray) -> None:
    """Plots the k-values in relation to their respective training and test accuracy."""
    plt.figure()
    plt.scatter(ks, acc_train, label="Accuracy Train", marker="x")
    plt.scatter(ks, acc_test, label="Accuracy Test", marker="^")
    plt.legend()
    plt.xlabel("$k$ Neighbors")
    plt.ylabel("Accuracy")
    plt.title("KNN: Accuracy vs. Number of Neighbors")
    plt.show()
```

d) **Bester k-Wert**

Bestimmen Sie anhand der zuvor erhaltenen Zwischenergebnisse den besten k-Wert über die Funktion `get_best_k`.

```python
def get_best_k(ks: range, acc_test: np.ndarray) -> int:
    """Returns the best value for k based on the highest test accuracy."""
    best_k = ks[acc_test.argmax()]
    return best_k
```

e) **Entscheidungsregionen**

Wir können auch die Entscheidungsregionen visualisieren, die ein kNN-Klassifikator für gegebene Werte von $k$ erstellt. Erzeugen Sie einen kNN-Klassifikator und stellen Sie dessen Entscheidungsregionen visuell in der Methode `plot_decision_boundary_for_k` dar.

```python
def plot_decision_boundary_for_k(k: int, X: np.ndarray, y: np.ndarray) -> None:
    """Creates and fits a KNN model with value k and plots its decision boundary."""
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X, y)
    utils.plot_decision_boundary(knn, X, y)
```

## 2.3 Regression (Fortsetzung)

Die lineare Regression ist die Aufgabe, eine Linearkombination der Eingabe $X$ zu finden, um das Ziel $y$ zu schätzen. Wir nehmen an, dass die Daten $\mathbf{x_i}$ linear mit dem Ziel $y_i$ korreliert sind, unter Verwendung der Koeffizienten $\beta_0, \ldots, \beta_K \in \mathbb{R}$:

$$y_i = \beta_0 + \beta_1 x_i^1 + \beta_2 x_i^2 + \cdots + \beta_K x_i^K + \varepsilon_i \,, \tag{1}$$

wobei $\varepsilon_i$ einen Störungsterm modelliert und der Superskript den Eingangsvariablen-Index entspricht.

Die lineare Regression lässt sich am besten an einem einfachen zweidimensionalen Beispiel demonstrieren. Hier reduziert sich diese auf den klassischen Fall $y_i = b + mx_i$.

a) **Robustheit**

Bewerten Sie, ob sich die lineare Regression robust gegenüber Ausreißer verhält und warum.

*Die Modellkoeffizienten $m$ und $b$ werden mit der Methode der gewöhnlichen kleinsten Quadrate (engl. Ordinary Least Squares) berechnet, die von Gauß-verteilten Residuen $|\hat{y}_i - y_i|$ ausgeht. Da die Gaußsche Methode schwache Außenseiten hat, weist sie Ausreißern eine geringe Wahrscheinlichkeit zu. Daher reagiert die lineare Regression sehr empfindlich auf Ausreißer.*

b) **Polynomiale Regression**

Wir können die lineare Regression leicht auf nicht-lineare Eingangsvariablen-Zielvariablen-Korrelationen ausdehnen. Daher ist es nur notwendig, unsere Eingabe in einen Merkmalsraum abzubilden, in dem die Merkmale unserer Meinung nach linear mit der Zielvariablen korreliert sind. Ein gutes Beispiel ist die Polynomiale Regression, die den Eingabedatenpunkt $x$ nimmt und ihn auf den Satz polynomialer Merkmale $\left(x^0, x^1, x^2, \ldots, x^d\right)$ abbildet, wobei $d$ der Grad der Polynomtransformation ist.

Implementieren Sie die Funktion `map_polynomial`, welche diese Operation durchführt.

```python
def map_polynomial(x: np.ndarray, degree: int) -> np.ndarray:
    """Create polynomial transformation of degree d:
     Maps each datapoint x_i, x_(i+1), ..., x_k to its polynomial:
      [x_i, x_(i+1), ..., x_k] -> [[x_i^0,     ..., x_i^d],
                                   [x_(i+1)^0, ..., x_(i+1)^d],
                                   ....
                                   [x_k^0,     ..., x_k^d]]."""
    x_poly = []
    # For each datapoint
    for xi in x.reshape(-1):
        xi_poly = [xi ** d for d in range(degree + 1)]
        x_poly.append(xi_poly)
    return np.array(x_poly)
```

# Literatur

[1] Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2):179–188, 1936.