Woche 07: Programmierung – *k*-nächste Nachbarn (*k*-NN) mit Scikit-learn

Skript

Erarbeitet von
Ludmila Himmelspach

Lernziele

- *k*-nächste-Nachbarn-Modell für einen gegebenen Datensatz erstellen können
- Ein trainiertes Modell nutzen können, um für Beobachtungen der Testmenge die Klasse zu prognostizieren
- Die Güte eines Klassifikationsmodells mit Accuracy bewerten können

Inhalt

Einstieg

Unter *k*-nearest Neighbors oder auf Deutsch *k*-nächste-Nachbarn (*k*-NN) versteht man ein Klassifikationsverfahren, das nach dem Prinzip arbeitet: „Sag mir, wer deine Nachbarn sind und ich sage dir, wer du bist“. Denn der k-nächste-Nachbarn-Algorithmus ordnet eine neue Beobachtung der Klasse zu, zu der die meisten ihrer *k* Nachbarn mit bekannter

Projekt „KI für alle“
HeiCAD – Heine Center for Artificial Intelligence and Data Science

Klassenzugehörigkeit gehören. Dabei wird die Nachbarschaft durch den Abstand zwischen den Datenpunkten bestimmt.

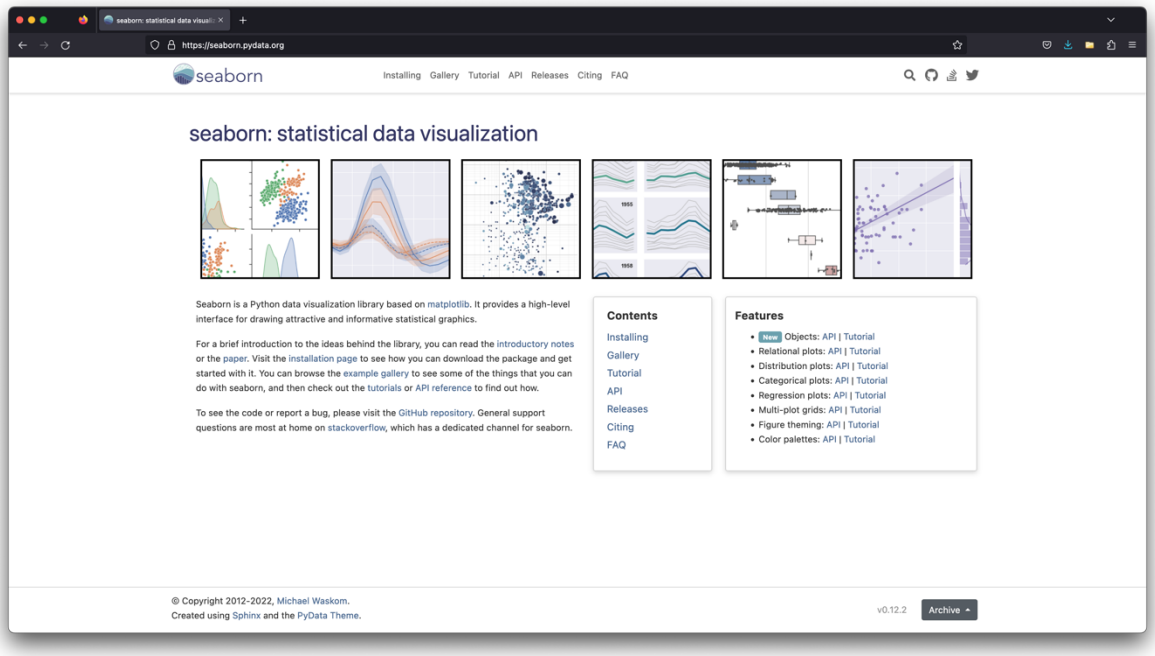
Einblendung *k*-NN-Animation

Vorbereitung des Datensatzes

Am Anfang importieren wir alle Module, die wir in unserem Programm brauchen werden.

```
# Importiere die nötigen Module
import seaborn as sns
import numpy as np
import plotly.express as px
import plotly.offline as pyo
import plotly.graph_objects as go
from sklearn import neighbors
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

Als erstes siehst du ein neues Python-Modul namens *seaborn*. *Seaborn* ist eigentlich ein Visualisierungsmodul zur Erstellung statistischer Grafiken.



Einblendung seaborn-Webseite (seaborn.pydata.org)

Quelle [1]

Es bietet aber die Funktion *load_dataset()* an, mit deren Hilfe man einige Referenzdatensätze aus einem Online-Repository laden kann.

Quelle [2]

In diesem Video arbeiten wir mit dem *Palmer Archipelago (Antarctica) Penguin* Datensatz.

Einblendung Pinguine (Quelle [8])

Dieser Datensatz enthält Informationen über 344 Pinguine, die auf drei Inseln des Palmer-Archipels in der Antarktis beheimatet sind. Die Merkmale des Datensatzes enthalten verschiedene Messdaten, die über die Pinguine gesammelt wurden, wie Länge und Höhe des Schnabels, Länge der Flosse und die Körpermasse bzw. das Gewicht. Das Zielmerkmal beschreibt die entsprechende Pinguin-Art. Dabei gehören die Beobachtungen bzw. die Pinguine im Datensatz zu einer der drei Arten: *Adelie*, *Chinstrap* (deutsch: Zügelpinguin) und *Gentoo* (deutsch: Eselspinguin).

Einblendung Pinguine (Quelle [8])

Quelle [3]

Wir laden den Datensatz und speichern ihn in der Variable *penguins*. Der Datensatz wird in einer Datenstruktur namens *DataFrame* gespeichert. Wir gehen auf diese Datenstruktur nicht weiter ein, lassen uns aber mit der Funktion *head()* die ersten fünf Zeilen des Datensatzes mit den Merkmalsbezeichnungen anzeigen.

```
# Lade den Palmer Archipelago (Antarctica) penguin Datensatz
penguins = sns.load_dataset("penguins")

# Gebe die ersten fünf Zeilen des Datensatzes mit den
# Bezeichnungen der Merkmale aus
print(penguins.head())
```

In der Ausgabe sehen wir bereits, dass einige Beobachtungen fehlende Werte enthalten. Dazu kommen wir später nochmal.

Out[2]:

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0	Male
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0	Female
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0	Female
3	Adelie	Torgersen	NaN	NaN	NaN	NaN	NaN
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0	Female

Einblendung Ausgabe

Zuerst wandeln wir die Datenstruktur *DataFrame*, in der unser Datensatz momentan vorliegt, in die Datenstruktur *ndarray* um. Dafür verwenden wir die Funktion *to_numpy()*.

Quelle [4]

```
# Wandle das DataFrame in ein NumPy-Array um
penguins_arr = penguins.to_numpy()
```

Für die weitere Analyse konzentrieren wir uns nur auf die Merkmale Schnabellänge, Schnabelhöhe, Flossenlänge und die Körpermasse. Also wählen wir diese aus dem Array *penguins_arr* aus und speichern die Werte im *ndarray X_raw* ab. Die erste Spalte des Arrays *penguins_arr* enthält die entsprechenden Pinguin-Arten. Also wählen wir diese aus und speichern die Pinguinarten im Zielarray *y_raw* ab.

```
# Speichere das dritte (Schnabellänge), das vierte (Schnabelhöhe),
# das fünfte (Flossenlänge) und das sechste (Körpermasse)
# Merkmal in die Datenmatrix
X_raw = np.array(penguins_arr[:,2:6]).astype(float)

# Speichere das erste Merkmal als Zielmerkmal
y_raw = penguins_arr[:,0]
```

Wie wir vorhin gesehen haben, fehlen bei einigen Beobachtungen die Werte. Genau genommen betrifft es zwei Beobachtungen. Da man zwischen diesen und den vollständigen Beobachtungen keine Distanzen berechnen kann, bekommt der Klassifikationsalgorithmus Probleme. Deswegen entfernen wir die unvollständigen Beobachtungen aus dem Datensatz. Mach dir keine Sorgen, wenn du die Befehle dafür nicht verstehst. Das wird von dir an dieser Stelle nicht erwartet.

```
# Lösche alle Zeilen aus der Datenmatrix, die
# fehlende Wert enthalten
X = X_raw[~np.isnan(X_raw).any(axis=1)]
y = y_raw[~np.isnan(X_raw).any(axis=1)]
```

Nach Bereinigung verbleiben im Datensatz noch 342 Beobachtungen.

Um sich ein besseres Bild über die Verteilung der Beobachtungen zu machen, visualisieren wir sie in einem Diagramm. Da es schwierig ist, die Datenpunkte durch alle vier Merkmale grafisch darzustellen, beschränken wir uns auf die Darstellung der Beobachtungen durch die Werte der Merkmale Schnabellänge, Schnabelhöhe und die Flossenlänge in einem 3-dimensionalen Diagramm. Dabei markieren wir die Zugehörigkeit zur Pinguinart durch unterschiedliche Farben der Datenpunkte. Die Adelie-Pinguine werden durch blaue, die Chinstrap-Pinguine durch rote und die Gentoo-Pinguine durch grüne Punkte dargestellt.

```
pyo.init_notebook_mode()
fig1 = px.scatter_3d(x=X[:,0], y=X[:,1], z=X[:,2], color=y)
fig1.show()
```

Im Diagramm sehen wir, dass die Adelie-Pinguine zwar durchschnittlich einen längeren Schnabel als die Chinstrap-Pinguine haben, aber eine klare Trennung zwischen diesen beiden Pinguin-Arten anhand der drei gewählten Merkmale nicht zu beobachten ist. Dagegen lassen sich die Gentoo-Pinguine durch eine längere Flosse von den anderen Arten besser trennen.

Bevor wir ein *k*-nächste-Nachbarn-Modell erstellen, zerlegen wir die Datenmatrix in eine Trainings- und eine Testmenge. Dafür benutzen wir die Funktion *train_test_split()* des Untermoduls *Model Selection* von *Scikit-learn*.

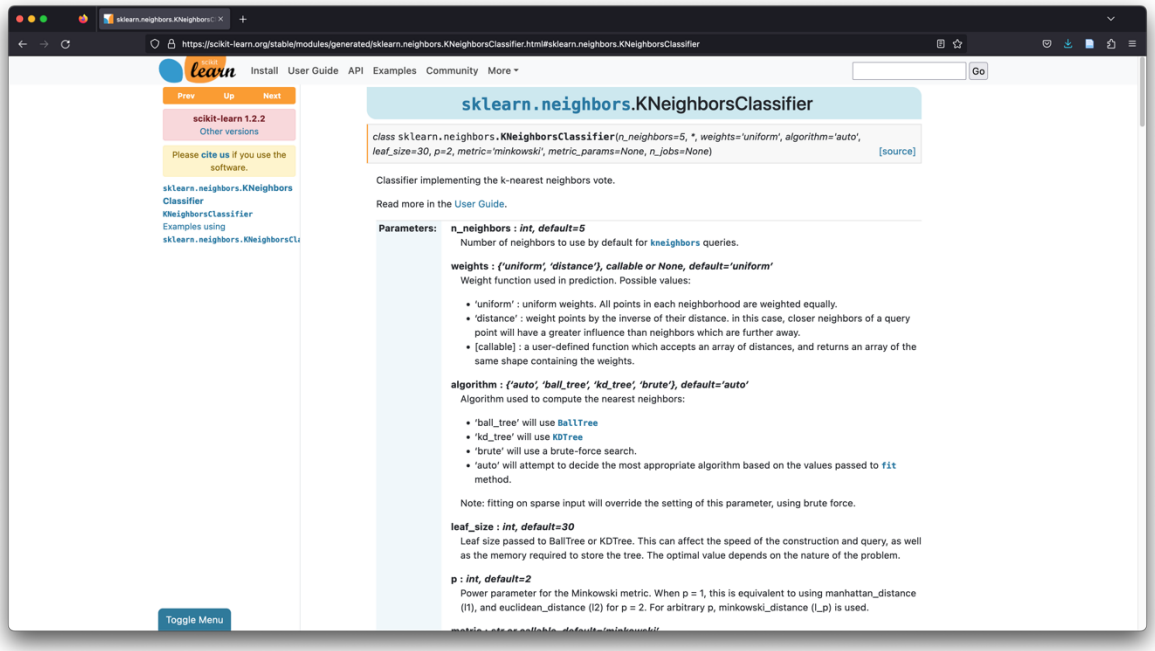
Quelle [5]

Diese Funktion erwartet als Eingabe die Datenmatrix mit dem entsprechenden Zielmerkmal. Mit dem Wert 0,1 für den Parameter *test_size* legen wir fest, dass die Testmenge 10 % aller Beobachtungen enthalten soll. In den Voreinstellungen der Funktion ist festgelegt, dass die Beobachtungen vor der Zerlegung gemischt werden. Für diesen Datensatz ist es besonders wichtig, diese Voreinstellung beizubehalten, weil die Beobachtungen in der ursprünglichen Datenmatrix nach dem Zielmerkmal sortiert sind. Würden wir auf das Mischen verzichten, wäre die resultierende Trainingsmenge für die gesamte Datenmenge nicht repräsentativ, da eine Art der Pinguine entweder unterrepräsentiert oder gar nicht in der Trainingsmenge vertreten wäre. Mit der Wertzuweisung des Parameters *random_state* stellen wir aber sicher, dass die Beobachtungen im Datensatz auf eine bestimmte Weise durchmischt werden. Damit sichern wir auch die Reproduzierbarkeit unserer Zerlegung.

```
# Teile den Datensatz in die Trainings- und die Testmenge auf
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.1, random_state=45)
```

k-nächste-Nachbarn-Modell

Beim *Palmer Archipelago (Antarctica) Penguin* Datensatz besteht unsere Aufgabe darin, ein *k*-nächste-Nachbar-Modell zu entwickeln, das anhand der Merkmale eines Pinguins prognostizieren kann, zu welcher Pinguinart dieser gehört. Ein *k*-nächste-Nachbar-Modell wird mit der Funktion *KNeighborsClassifier()* des *Scikit-learn*-Untermoduls *Nearest Neighbors* erstellt. Mit dem Parameter *n_neighbors* wird die Anzahl der nächsten Nachbarn festgelegt, deren Klassenzugehörigkeit bei der Zuordnung einer neuen Beobachtung herangezogen wird. Wir speichern den Wert dieses Parameters in der Variable *k* und weisen ihren Wert dem Parameter *n_neighbors* zu. Auf diese Weise findest du die Stelle im Programm schneller, die verändert werden muss, wenn du den *k*-nächste-Nachbarn-Klassifikator für einen anderen Wert von *k* ausprobieren möchtest.



Einblendung User Guide

Quelle [6]

Neben der Nachbarzahl kannst du durch einige andere Parameter der Funktion die Distanzfunktion vordefinieren, mit deren Hilfe die Nachbarn bestimmt werden. In den Voreinstellungen ist die Euklidische Distanzfunktion, die du sicherlich noch aus der Schule kennst, festgelegt.

Das Anpassen des Modells an die Trainingsdaten erfolgt mit der Funktion *fit()*. Bei diesem Modell findet kein richtiges Training statt. Stattdessen werden die Beobachtungen der Trainingsmenge intern in einer speziellen Datenstruktur gespeichert, die das schnelle Finden der *k*-nächsten-Nachbarn für eine gegebene Distanzfunktion ermöglicht.

```
# In der Variable k wird die Nachbaranzahl gespeichert
k = 3

# Erstelle das knn-Modell
knn_classifier = neighbors.KNeighborsClassifier(n_neighbors=k)

# Trainiere das Modell mit der Trainingsmenge
knn_classifier.fit(X_train, y_train)
```

Mit der Funktion *predict()* kannst du die Klasse für neue Beobachtungen prognostizieren. Wir prognostizieren die Klassenzuordnung für alle Beobachtungen der Testmenge.

```
# Erstelle die Vorhersage für die Testmenge
y_pred = knn_classifier.predict(X_test)
```

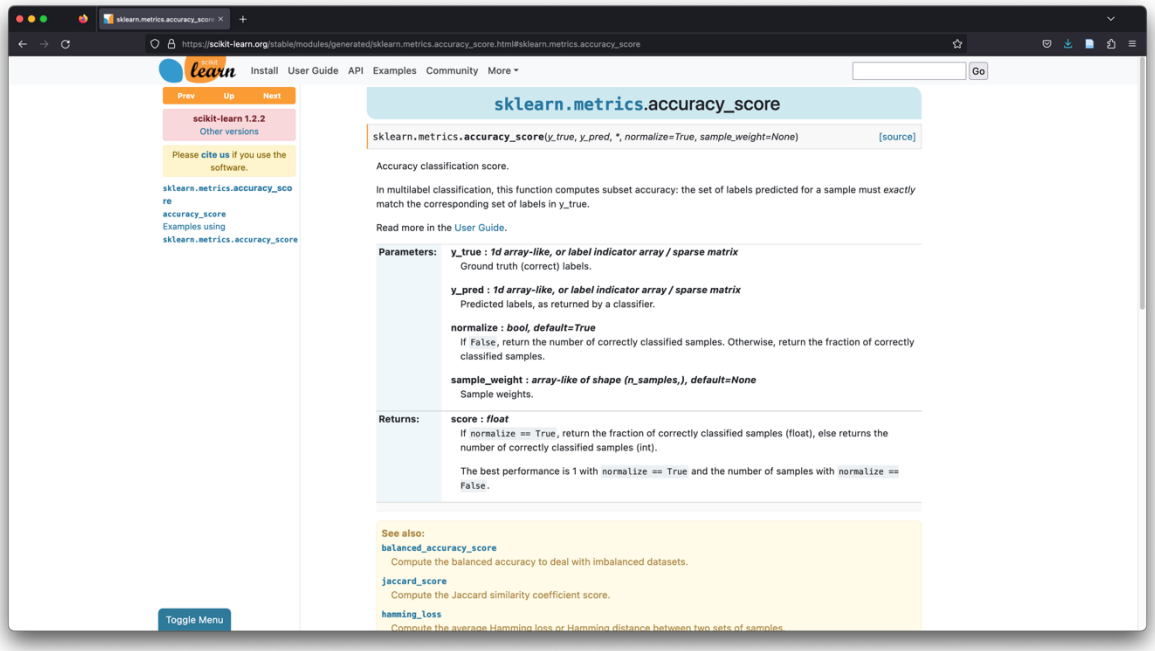
Um sehen zu können, wie gut das *k*-nächste-Nachbarn-Modell die Klasse für neue Beobachtungen schätzen kann, geben wir exemplarisch für die erste Beobachtung der Testmenge die tatsächliche und die durch das *k*-NN Klassifikationsmodell geschätzte Klasse bzw. Pinguinart aus.

```
# Gebe den tatsächlichen Wert des Zielmerkmals für
# die erste Beobachtung der Testmenge aus
print("Die tatsächliche Pinguin-Art der "
      "ersten Beobachtung aus der Testmenge:", y_test[0])

# Gebe den geschätzten Wert des Zielmerkmals für
# die erste Beobachtung der Testmenge aus
print("Der geschätzte Pinguin-Art der "
      "ersten Beobachtung aus der Testmenge:", y_pred[0])
```

In der Ausgabe sehen wir, dass die durch das Klassifikationsmodell prognostizierte Klasse mit der tatsächlichen Klasse für diese eine Beobachtung übereinstimmt.

Um die Leistung eines Klassifikationsmodells zu bewerten, reicht es normalerweise nicht aus, die tatsächliche und die prognostizierte Klasse für eine Beobachtung miteinander zu vergleichen. Vielmehr werden diese beiden Werte für alle Beobachtungen der Testmenge miteinander verglichen und zu einem Gütemaß zusammengefasst. Eins dieser Gütemaße ist *Accuracy*, die auf Deutsch auch *Genauigkeit* oder *Korrektklassifizierungsrate* genannt wird. *Accuracy* misst die Anzahl der richtigen Vorhersagen im Verhältnis zur Gesamtzahl aller gemachten Vorhersagen. Im Untermodul *Metrics* von *Scikit-learn* steht die Funktion *accuracy_score()* für die Berechnung der *Accuracy* zur Verfügung. Diese Funktion erwartet zwei 1-dimensionale Arrays als Eingabe. Dabei soll das erste Array die tatsächlichen und das zweite Array die geschätzten Klassen der Beobachtungen enthalten. Als Ergebnis liefert die Funktion den Genauigkeitswert für die Schätzung.



Einblendung User Guide

Quelle [7]

Die tatsächlichen Klassenzugehörigkeiten der Beobachtungen der Testmenge sind im Array *y_test* gespeichert, die prognostizierten Klassen haben wir im Array *y_pred* abgelegt. Wir übergeben diese beiden Arrays der Funktion *accuracy_score()*, um *Accuracy* für unser *k*-nächste-Nachbarn-Modell mit *k = 3* zu berechnen.

```
# Berechne die Accuracy
accuracy = accuracy_score(y_test, y_pred)
print(accuracy)
```

Wie du in der Ausgabe sehen kannst, erreicht unser Klassifikationsmodell einen Accuracy-Wert von 0.77, d. h. für 77 % der Beobachtungen wurde die Klasse richtig prognostiziert.

Wir erhöhen jetzt den Wert für *k* und schauen, wie sich die Accuracy verändert. Für *k = 5* fällt die Accuracy auf 0.66 runter, für *k = 9* steigt die Accuracy wieder auf 0.77. Du kannst selbst ausprobieren, wie sich die Accuracy für unterschiedliche Werte von *k* verändert. Beim *k*-nächste-Nachbarn-Klassifikator findet zwar keine Anpassung des Modells auf die Trainingsdaten statt, dafür kannst du durch Ausprobieren unterschiedlicher Werte für den Parameters *k* und der Wahl der Distanzfunktion dein Klassifikationsmodell für einen bestimmten Datensatz selbst optimieren.

Zum Schluss stellen wir noch die Beobachtungen der Testmenge in einem 3-dimensionalen Diagramm grafisch dar. Dafür verwenden wir wieder nur die ersten drei Merkmale: die Schnabellänge, die Schnabelhöhe und die Flossenlänge. Wir markieren die Pinguinarten der Beobachtungen durch unterschiedliche Farben. Dabei stellen wir die tatsächlichen Klassen der Beobachtungen durch Diamanten und die prognostizierten durch Kreuze dar.

```
# Visualisiere die tatsächliche Zugehörigkeit als Diamant
# und die geschätzte als Kreuz
fig3 = px.scatter_3d(x=X_test[:,0], y=X_test[:,1], z=X_test[:,2],
                     color=y_test, opacity=0.8,
                     symbol_sequence=["diamond"])
fig4 = px.scatter_3d(x=X_test[:,0], y=X_test[:,1], z=X_test[:,2],
                     color=y_pred, symbol_sequence=["cross"])
fig5 = go.Figure(data=fig3.data + fig4.data)
fig5.show()
```

Im Diagramm kannst du sehen, welche Beobachtungen richtig und welche falsch klassifiziert wurden. Bei den richtig klassifizierten Beobachtungen stimmen die Farben der Diamanten mit den Farben der Kreuze überein. Bei den falsch klassifizierten Beobachtungen sind Diamanten und Kreuze unterschiedlich eingefärbt. Die meisten Fehler passierten bei den Beobachtungen, die zwischen den Beobachtungen unterschiedlicher Klassen liegen.

Take-Home Message

In diesem Video hast du gelernt, wie man ein k-nächste-Nachbarn-Klassifikationsmodell erstellt und mit seiner Hilfe die Klassenzugehörigkeit für neue Beobachtungen prognostiziert.

Außerdem weißt du jetzt, wie man das Gütemaß *Accuracy* für ein Klassifikationsmodell berechnet, um die Leistungsfähigkeit des Modells zu bewerten.

Quellen

Quelle [1] Waskom, M. L. (2021). seaborn: statistical data visualization. Journal of Open Source Software, 6(60), 3021. https://doi.org/10.21105/joss.03021

Quelle [2] Waskom, M. (2022). seaborn.load_dataset. In *seaborn API Reference, Release 0.12.2*
https://seaborn.pydata.org/generated/seaborn.load_dataset.html

Quelle [3] Horst, A. M., Hill, A. P., & Gorman, K. B. (2020). *palmerpenguins: Palmer Archipelago (Antarctica) penguin data*. *R package version 0.1.0.*
https://allisonhorst.github.io/palmerpenguins/
https://doi.org/10.5281/zenodo.3960218

Quelle [4] NumFOCUS, Inc. (2022). DataFrame. In *pandas API Reference, Release 1.5.2*
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_numpy.html

Quelle [5] Scikit-learn (2022). Model Selection. In *Scikit-learn Reference, Release 1.2.1*
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

Quelle [6] Scikit-learn (2022). Nearest Neighbors. In *Scikit-learn Reference, Release 1.2.1*
https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

Quelle [7] Scikit-learn (2022). Metrics. In *Scikit-learn Reference, Release 1.2.1*
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html

Quelle [8] Artwork by @allison_horst
https://allisonhorst.github.io/palmerpenguins/articles/art.html

Weiterführendes Material

Nguyen, C. N., & Zeigermann, O. (2021). *Machine Learning -- kurz & gut: Eine Einführung mit Python, Pandas und Scikit-Learn*. O'Reilly

Disclaimer

Transkript zu dem Video „Woche 07: Programmierung – *k*-nächste Nachbarn (*k*-NN) mit Scikit-learn“, Ludmila Himmelspach.