TSRI-12

July 23, 2024

1 Übung 12

Gruppenname: TSRI

- Christian Rene Thelen @cortex359
- Leonard Schiel @leo_paticumbum
- Marine Raimbault @Marine Raimbault
- Alexander Ivanets @sandrium

1.0.1 In dieser Übung ...

... werden wir uns mit daten-getriebener Modellierung mithilfe von Nächste Nachbarn Modellen beschäftigen. Nächste Nachbarn Modelle gehören zu den einfachsten Machine Learning Modellen. Viele weitere Modelle können Sie im Bachelor Kurs "Machine Learning" kennenlernen. Ich lade Sie herzlich dazu ein, sich für den Kurs Bachelorkurs "Machine Learning" anzumelden, sollten Sie Interesse haben.

1.0.2 12.1 Weinqualitäten (Multiklassen-Klassifikation, Feature Engineering)

- Diese Übung kennen Sie in abgewandelter Form aus der optionalen Übung 11.3. Falls Sie Übung 11.3 schon bearbeitet haben, können Sie sich von dort für eine Lösung hier inspirieren lassen.

In dieser Übung werden wir ein Nächste Nachbarn (NN) Modell erstellen, mit dem wir die Klasse eines Weines (d.h. die Kultursorte) aus den Eigenschaften vorhersagen können. Wir werden mit PCA-transformierten Merkmalen arbeiten.

Ihre Daten

Zu Beginn der 90er Jahre wurden verschiedene Weinproben in einer Region Italiens untersucht. Die Weine stammen von drei verschiedenen Kultursorten. Diese Kultursorten werden im unten hinterlegten Datensatz als Klasse 1, 2 und 3 (*class labels*) bezeichnet. Unter den 13 untersuchten Merkmalen finden Sie neben chemischen Eigenschaften (Alkoholgehalt, Säuregehalt) auch physikalische Eigenschaften (Farbintensität, etc).

Ich habe Ihnen den Weindatensatz in zwei Teile geteilt: Der erste Datensatz ist der sogenannte Trainingsdatensatz, mithilfe dessen Sie ihr Modell bauen werden. Der zweite Datensatz ist der sogenannte Testdatensatz, auf dem Sie ihr Modell anwenden und testen werden.

[1]:
```
import numpy as np
import pandas as pd
```

```
from matplotlib import pyplot as plt
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split

# import data
column_names = ['Class label', 'Alcohol', 'Malic acid', 'Ash', 'Alcalinity of␣
 ↪ash', 'Magnesium', 'Total phenols', 'Flavanoids',
                'Nonflavanoid phenols', 'Proanthocyanins', 'Color intensity',␣
 ↪'Hue', 'OD280/OD315 of diluted wines', 'Proline']
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/
 ↪wine/wine.data', header=None)
df.columns = column_names

# preprocess data
X, y = df.iloc[:, 1:].values, df.iloc[:, 0].values

X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.4, random_state=10)

# apply PCA
pca = PCA(n_components=2)
X_train_p = pca.fit_transform(X_train)
X_test_p = pca.transform(X_test)
```

Ihre Aufgaben

(1) Importieren Sie die Daten, indem Sie die obere Code-Zelle ausführen.

(2) Vergegenwärtigen Sie sich die Eigenschaften des Wein-Datensatzes: Untersuchen Sie dazu das Pandas-Objekt. Welche Eigenschaften wurden für die Weine erfasst? Wie viele Weinproben wurden genommen?

[2]:
```
print(f"Erfasste Eigenschaften von {df.shape[0]} Weinproben: {', '.join(list(df.
 ↪columns[1:]))}")
```

```
Erfasste Eigenschaften von 178 Weinproben: Alcohol, Malic acid, Ash, Alcalinity
of ash, Magnesium, Total phenols, Flavanoids, Nonflavanoid phenols,
Proanthocyanins, Color intensity, Hue, OD280/OD315 of diluted wines, Proline
```

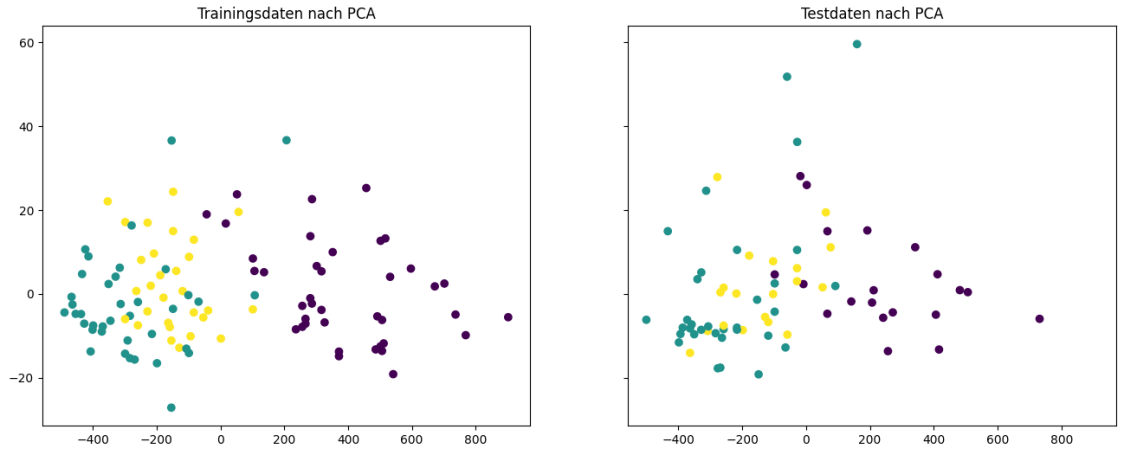
(3) Lesen Sie den Code ab dem Kommentar `# preprocess data`. Was passiert in diesen Code-Zeilen? Welches Feature-Engineering findet statt, bis wir zu den Daten `X_train_p`, `X_test_p` angekommen sind, mit denen Sie dann arbeiten werden? Wie viele Features (Merkmale) haben die transformierten Daten?

Es wurde eine PCA der Daten von 14 auf 2 Dimensionen durchgeführt.

(4) Visualisieren Sie in zwei Scatterplots den PCA-transformierten Trainingsdatensatz sowie den Testdatensatz. Färben Sie die Punkte in beiden Plots ein gemäß der Klassenzugehörigkeit, wie Sie in Ihrem Vektor `y_train` bzw. `y_test` kodiert ist. Die Einträge in `y_train` und `y_test` werden auch *Labels* genannt.

- Hinweis: Vielleicht wollen Sie sich an Ihrem Code aus früheren Übungen bedienen, wo Sie eine ähnliche Aufgabe schon einmal gelöst haben.

[3]:
```
fix, axs = plt.subplots(1, 2, figsize=(16, 6), sharey=True)
axs[0].scatter(X_train_p[:, 0], X_train_p[:, 1], c=y_train)
axs[0].set_title("Trainingsdaten nach PCA")
axs[1].scatter(X_test_p[:, 0], X_test_p[:, 1], c=y_test)
axs[1].set_title("Testdaten nach PCA")
axs[1].set_xlim(*axs[0].get_xlim())
plt.show()
```



(5) Wir werden jetzt ein NN-Modell (Nächste Nachbarn Modell) erstellen. Schreiben Sie dazu eine Funktion mit dem Namen `NN`, die die Trainingsdaten (`X_train_p` und `y_train`) sowie die Testdaten (`X_test_p`) entgegen nimmt und die vorhergesagten Klassen (Labels) für die Testdaten als Vektor (`y_pred`) zurückgibt. Schlagen Sie in den Vorlesungsfolien nach, wie ein NN Modell definiert ist und nutzen Sie für die Implementierung numpy Funktionen. Broadcasting kann Ihnen ebenfalls sehr hilfreich sein.

[4]:
```
def NN(X_train: np.ndarray, y_train: np.ndarray, X: np.ndarray) -> np.ndarray:
    y2 = np.sum(X_train**2, axis=1)
    x2 = np.sum(X**2, axis=1).reshape(-1, 1)
    distances = np.sqrt(x2 - 2*(X @ X_train.T) + y2)
    return y_train[np.argmin(distances, axis=1)]
```

(6) Sie haben in Schritt (5) ein einfaches Machine Learning Modell implementiert, einen Klassifikator. Sagen Sie mithilfe Ihrer Funktion die Labels (Klassen) der Weinproben des Testdatensatzes voraus und speichern Sie die Voraussage im Vektor `y_pred`.

[5]:
```
y_pred: np.ndarray = NN(X_train_p, y_train, X_test_p)
```

(7) Bestimmen Sie die *Accuracy* Ihrer Vorhersage, also den Anteil der korrekt vorhergesagten Klassen dividiert durch die Gesamtanzahl aller Vorhersagen. Beurteilen Sie anhand der *Accuracy*, ob Ihr Modell die Klassen des Testdatensatzes gut vorhersagen kann.

[6]:
```
def Accuracy(y_test: np.ndarray, y_pred: np.ndarray) -> float:
    return (y_pred == y_test).sum() / y_pred.size

Accuracy(y_test, y_pred)
```

[6]:
```
np.float64(0.6944444444444444)
```

Mit einer Accuracy von ACC $\approx 69,4\%$ ist die Vorhersage eher schlecht.

(8) Wiederholen Sie Schritt (5), allerdings für Trainingsdaten, deren Features Sie auf Mittelwert 0 und Varianz 1 normiert haben:

- Normieren Sie also die ursprünglichen Trainingsdaten **vor der PCA**, führen Sie dann die PCA mit den normierten Daten durch und klassifizieren Sie die aus der PCA resultierenden Daten. Welche Genauigkeit (Accuracy) erhalten Sie nun auf den Trainingsdaten? Wie erklären Sie sich diese Veränderung im Vergleich zu dem Ergebnis aus Schritt (7)?
- Normieren Sie anschließend auch die Testdaten, und zwar so, dass diese mit denselben Parametern wie die Trainingsdaten normiert werden. Dies bedeutet: Fitten Sie den StandardScaler auf die Trainingsdaten (`.fit`) und wenden Sie diese Transformation nun auch auf die Testdaten an (`.transform`).

[7]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split

# preprocess data
X, y = df.iloc[:, 1:].values, df.iloc[:, 0].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4,␣
 ↪random_state=10)

pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('pca', PCA(n_components=2))
])

X_train_p = pipe.fit_transform(X_train)
X_test_p = pipe.transform(X_test)

y_pred = NN(X_train_p, y_train, X_test_p)

Accuracy(y_test, y_pred)
```

[7]:
```
np.float64(0.9583333333333334)
```

(9) Wir werden nun untersuchen, wie sich die *Accuracy* mit der Anzahl k der nächsten Nachbarn ändert. Nutzen Sie die Implementierung des kNN Modells von scikit learn, um die Accuracy als Funktion von k zu ermitteln ($k \in \{1, \dots, 50\}$).

Hinweise

- Trainieren Sie Ihr Modell auf dem Trainingsset.
- Ermitteln Sie die Accuracies auf den Testsets.

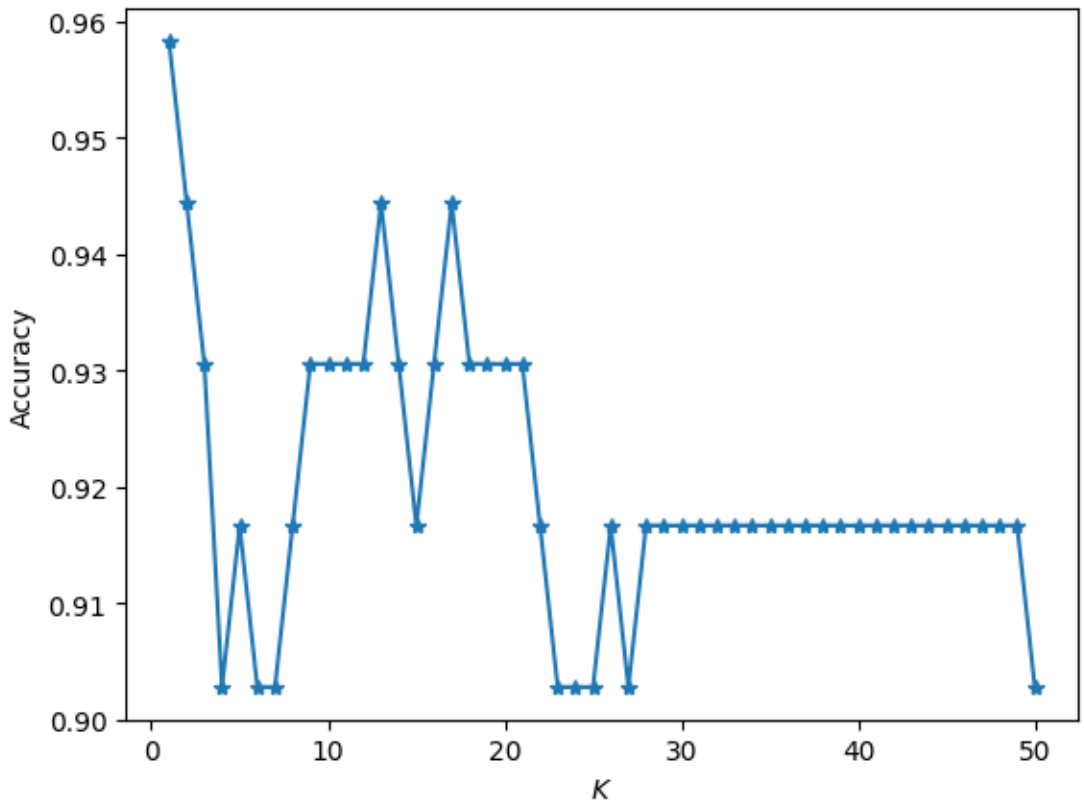
[8]:
```
from sklearn.neighbors import KNeighborsClassifier

accuracies = []
for k in range(1, 51):
    knn_classifier = KNeighborsClassifier(n_neighbors=k).fit(X_train_p, y_train)
    y_pred = knn_classifier.predict(X_test_p)
    accuracy = (y_pred == y_test).sum() / y_pred.size
    accuracies.append(accuracy)
    print(f"k={k:2d} ACC = {accuracy}")
```

```
k= 1 ACC = 0.9583333333333334
k= 2 ACC = 0.9444444444444444
k= 3 ACC = 0.9305555555555556
k= 4 ACC = 0.9027777777777778
k= 5 ACC = 0.9166666666666666
k= 6 ACC = 0.9027777777777778
k= 7 ACC = 0.9027777777777778
k= 8 ACC = 0.9166666666666666
k= 9 ACC = 0.9305555555555556
k=10 ACC = 0.9305555555555556
k=11 ACC = 0.9305555555555556
k=12 ACC = 0.9305555555555556
k=13 ACC = 0.9444444444444444
k=14 ACC = 0.9305555555555556
k=15 ACC = 0.9166666666666666
k=16 ACC = 0.9305555555555556
k=17 ACC = 0.9444444444444444
k=18 ACC = 0.9305555555555556
k=19 ACC = 0.9305555555555556
k=20 ACC = 0.9305555555555556
k=21 ACC = 0.9305555555555556
k=22 ACC = 0.9166666666666666
k=23 ACC = 0.9027777777777778
k=24 ACC = 0.9027777777777778
k=25 ACC = 0.9027777777777778
k=26 ACC = 0.9166666666666666
k=27 ACC = 0.9027777777777778
k=28 ACC = 0.9166666666666666
k=29 ACC = 0.9166666666666666
k=30 ACC = 0.9166666666666666
k=31 ACC = 0.9166666666666666
k=32 ACC = 0.9166666666666666
k=33 ACC = 0.9166666666666666
k=34 ACC = 0.9166666666666666
```

```
k=35 ACC = 0.9166666666666666
k=36 ACC = 0.9166666666666666
k=37 ACC = 0.9166666666666666
k=38 ACC = 0.9166666666666666
k=39 ACC = 0.9166666666666666
k=40 ACC = 0.9166666666666666
k=41 ACC = 0.9166666666666666
k=42 ACC = 0.9166666666666666
k=43 ACC = 0.9166666666666666
k=44 ACC = 0.9166666666666666
k=45 ACC = 0.9166666666666666
k=46 ACC = 0.9166666666666666
k=47 ACC = 0.9166666666666666
k=48 ACC = 0.9166666666666666
k=49 ACC = 0.9166666666666666
k=50 ACC = 0.9027777777777778
```

(10) Visualisieren Sie die Accuracy als Funktion von k. Bestimmen Sie den besten Wert k.

[9]:
```
plt.plot(range(1, 51), accuracies, "-*")
plt.xlabel("$K$")
plt.ylabel("Accuracy")
plt.show()
```



Bei $k = 1$ haben wir die höchste Accuracy auf den Testdaten erzielt.

Damit darf ich Ihnen gratulieren. Sie haben ein k-Nächste Nachbarn Modell an einen Datensatz angepasst.

1.0.3 12.2 k-nächste-Nachbarn Klassifikation

In dieser kurzen Übungsaufgabe werden Sie sich weiter mit dem kNN-Klassifikator vertraut machen und die Wirkung des Parameters k untersuchen.

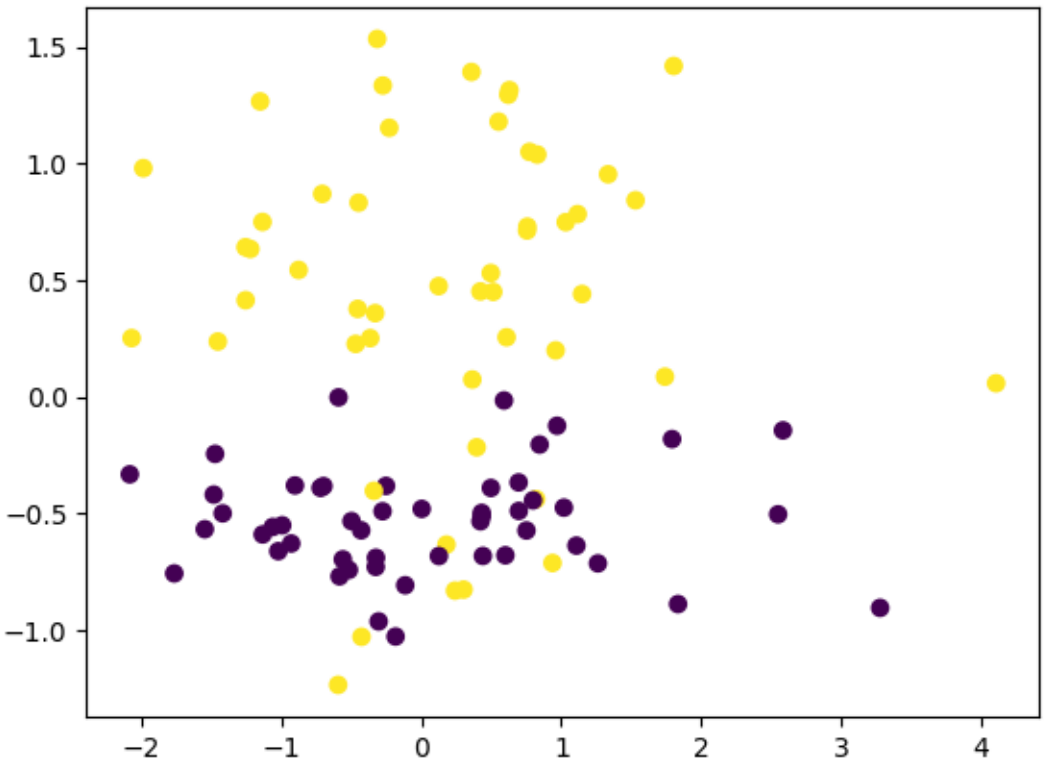
Ihre Aufgaben

(1) Visualisieren Sie die weiter unten erzeugten synthetischen Daten.

[10]:
```
import numpy as np
from matplotlib import pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.datasets import make_classification
from sklearn.neighbors import KNeighborsClassifier
%matplotlib inline

# helper function to plot decision boundaries
def plot_decision_regions(X, y, classifier, axis=plt, resolution=0.02):

    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    axis.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)

    if not hasattr(axis, "set_xlim"):
        axis = plt.gca()
    axis.set_xlim(xx1.min(), xx1.max())
    axis.set_ylim(xx2.min(), xx2.max())

    # plot class samples
    for idx, cl in enumerate(np.unique(y)):
        axis.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                     alpha=0.8, c=np.expand_dims(cmap(idx), 0),
                     marker=markers[idx], label=cl)
```

```
# generate 100 data points: features X, labels y
X, y = make_classification(n_features=2, n_redundant=0, n_informative=1,
                           n_clusters_per_class=1, class_sep=0.5,␣
 ↪random_state=2)
```

[11]:
```
plt.scatter(X[:, 0], X[:, 1], c=y)
plt.show()
```



(2) Fitten Sie einen kNN-Klassifikator an die Daten mit $k = 1$ und visualisieren Sie mit der Hilfsfunktion `plot_decision_regions` die entstandenen Entscheidungsflächen.

[12]:
```
knn_classifier = KNeighborsClassifier(n_neighbors=1).fit(X, y)
plot_decision_regions(X, y, knn_classifier)
```

(3) Wiederholen Sie Schritt (2) für einige wenige Werte von k. Variieren Sie dabei k zwischen 1 und 50. Wie verändern sich die Entscheidungsflächen? Wie verändert sich, Ihrer Meinung nach, die Komplexität des kNN-Modells mit dem Wert k?

[13]:
```
k_list = [1, 2, 3, 10, 15, 20, 30, 40, 50]
fig, axs = plt.subplots(len(k_list) // 3, 3, figsize=(16, 16))
for i, k in enumerate(k_list):
    knn_classifier = KNeighborsClassifier(n_neighbors=k).fit(X, y)
```

```
    axs.flat[i].set_title(f"$k={k}$")
    plot_decision_regions(X, y, knn_classifier, axs.flat[i])
```

Je kleiner k ist, desto größer ist die Komplexität des kNN-Modells.

(4) Machen Sie Ihre Visualisierung aus Schritt (3) interaktiv, indem Sie ipywidgets benutzen, und über einen Slider Werte für k auswählen können.

[14]:
```
from ipywidgets import interact
import ipywidgets as widgets

def knn_instance(k: int):
    knn_classifier = KNeighborsClassifier(n_neighbors=k).fit(X, y)
    plot_decision_regions(X, y, knn_classifier)

interact(knn_instance, k=widgets.IntSlider(min=1, max=60, step=1))
```

```
interactive(children=(IntSlider(value=1, description='k', max=60, min=1),␣
↪Output()), _dom_classes=('widget-int…
```

[14]:
```
<function __main__.knn_instance(k: int)>
```