Modellevaluierung und Grid Search

June 7, 2021

1 Modellevaluierung und Grid Search

1.1 K-Fold-Cross Validation

Bisher haben wir einen Datensatz in einen Trainings- und einen Testdatensatz aufgeteilt. Doch es besteht die Möglichkeit, dass wir einen "ungünstigen" Trainings- bzw. Testdatensatz erhalten! Außerdem wissen wir nicht, wie "stabil" unser Modell ist. Schießlich wollen wir auch noch die Hyperparameter optimieren, also Parameter, die wir *vor* dem Trainieren unseres Modells bestimmen müssen. Um diese Hyperparameter zu optimieren und jeweils die Qualität des Modells zu bestimmen verwenden man häufig die sog. K-Fold-Cross-Validation.

Im diesem Beispiel wollen wir die handschriftlich erstellten Ziffern erkennen und verwenden hierfür den K-Nearest-Neighbors-Klassifizierer. Dem Konstruktor der Klasse *KNeighborsClassifier* können wir 2 Hyperparameter als Argumente übergeben:

- n_neighbors: Entspricht dem "K" in K-Nearest-Neighbors
- weights: *uniform* oder *distance*, bei *uniform* werden die Distanzen nicht gewichtet, bei *distance* werden sie gewichtet

1.2 Grid Search

Um die besten Hyperparameter zu bestimmen, gehen wir nun wie folgt vor:

- Wir erstellen ein Dictionary mit diesen Parametern. Für K versuchen wir die Werte 1, 3, 5, 7, 9 und 11
- Für jedes dieser Kombinationen, also insgesamt 2 * 6 = 12 Kombinationen, werden Modelle erstellt
- Wir führen mit K-Fold-Cross-Validation jeweils ein Modell. Da wir im Beispiel 5 Fold erstellen, werden also jeweils 5 Modelle für jedes der Kombinationen erstellt (also 5 * 12 = 60).
- Für jede Kombination wird jeweils der Mittelwert des Scores besimmt (hier die Accuracy).
- Das Objekt der Klasse GridSearchCV kann uns nun die Parameter zurückgeben, die die beste durchschnittliche Accuracy lieferte.
- Als letztes erstellen wir ein Objekt der Klasse *KNeighborsClassifier*, trainieren das Modell mit dem Test-Datensatz und testen mit dem ursprünglich erstellen Testdaten. **Wichtig**: Diese Testdaten wurden nicht bei der Cross-Validation verwendet! Das Modell sieht diese Testdaten also zum ersten Mal!

[9]:
```
from sklearn.datasets import load_digits
from sklearn.model_selection import GridSearchCV, train_test_split
```

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, plot_confusion_matrix

digits = load_digits()
X = digits.data
y = digits.target

# Aufteilen in Test- und Trainingsdaten. Die Testdaten werden für die␣
 ↪abschließende
# Evaluation verwendet (nicht bei der Cross-Validation!). Wir halten uns dafür
# 20% der Daten zurück.

X_train, X_test, y_train, y_test = train_test_split(X, y, shuffle=True,␣
 ↪random_state=42, test_size=0.2)

# Wir definieren die Hyperparameter, die wir testen wollen, insgesamt also 12␣
 ↪Modelle!
# Die Keys im Dictionary müssen exakt den Argumenten der jeweiligen Klasse␣
 ↪lauten!
parameter_grid = {"weights" : ["uniform", "distance"],
                  "n_neighbors" : [1, 3, 5, 7, 9, 11]}


# Wir erstellen ein Objekt der Klasse GridSearchCV, übergeben ein Objekt der␣
 ↪Klasse KNeighborsClassifier,
# unser Grid mit den Hyperparametern, für das Scoring definieren wir die␣
 ↪Accuracy,
# wir bestimmen mit cv=5 dass wir 5 Folds erstellen wollen (K-Folds).
# "Verbose=3" bestimmt, dass wir einige Ausgaben während des Trainings erhalten,
# "n_jobs=-1" bestimmt, dass wir alle Prozessoren für Threads nutzen wollen.

grid = GridSearchCV(KNeighborsClassifier(), parameter_grid, scoring="accuracy",␣
 ↪cv=5, verbose = 3, n_jobs=-1)

# Nun trainieren wir insgesamt 60 Modelle (2 * 6 * 5)
grid.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 12 candidates, totalling 60 fits
```

[9]:
```
GridSearchCV(cv=5, estimator=KNeighborsClassifier(), n_jobs=-1,
             param_grid={'n_neighbors': [1, 3, 5, 7, 9, 11],
                         'weights': ['uniform', 'distance']},
             scoring='accuracy', verbose=3)
```

[10]:
```
# Wir können nun die Parameter ausgeben, die die beste Performance lieferten:
print(grid.best_params_)
```

```
{'n_neighbors': 3, 'weights': 'distance'}
```

[11]:
```
# Wir können nun mit den bisher unverwendeten Testdaten ein Modell erstellen und
# das Modell evaluieren. Die Methode predict verwendet das beste gefundene␣
 ↪Modell!

grid_predictions = grid.predict(X_test)
print(classification_report(y_test, grid_predictions))
```

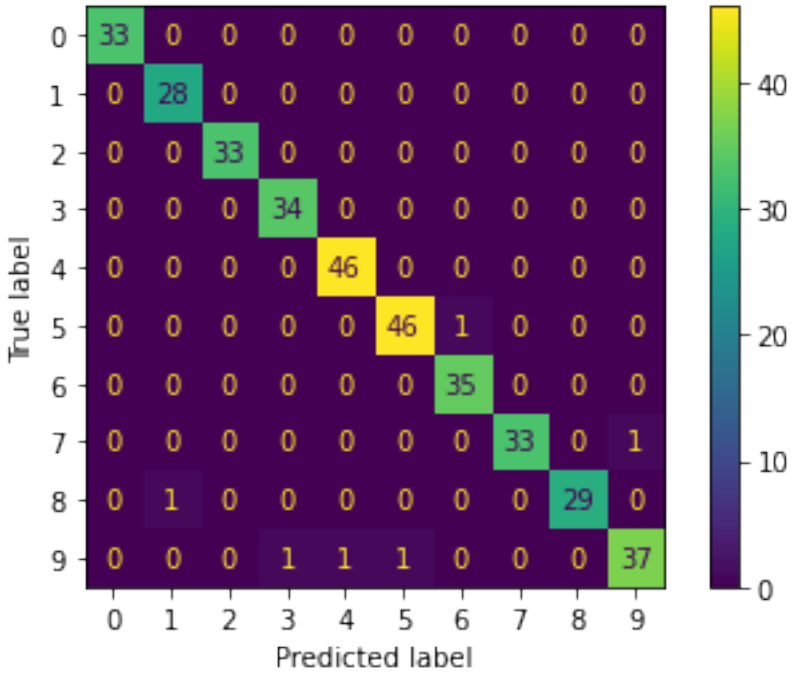
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        33
           1       0.97      1.00      0.98        28
           2       1.00      1.00      1.00        33
           3       0.97      1.00      0.99        34
           4       0.98      1.00      0.99        46
           5       0.98      0.98      0.98        47
           6       0.97      1.00      0.99        35
           7       1.00      0.97      0.99        34
           8       1.00      0.97      0.98        30
           9       0.97      0.93      0.95        40

    accuracy                           0.98       360
   macro avg       0.98      0.98      0.98       360
weighted avg       0.98      0.98      0.98       360
```

[12]:
```
# Schließlich können wir auch noch eine Confusion Matrix ausgeben
plot_confusion_matrix(grid, X_test, y_test)
```

[12]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x1fd1d2f77f0>
```



1.3 RandomizedSearchCV

Wir haben für den Hyperparameter K bei K-Nearest-Neighbors vorab definierte K´s vorgegeben (1, 3, 5, 7, 9, und 11). Wir können aber auch zufällig aus einer Range definierte Werte für K Modelle trainieren. Vielleicht ist ja ein K das "beste" K, das wir vorab gar nicht in Betracht ziehen!? Die Klasse *RandomizedSearchCV* wählt zufällige Kombinationen aller vorgegebenen Parameter aus und führt schließlich wieder mit jedes dieser Kombinationen eine K-Fold-Cross-Validation aus.

Statt der vorgegebenen Liste für die K´s wird hier also eine Bereich definiert, im Beispiel Werte für K von 1 bis 12.

Hinweis: Im gegebenen Beispiel macht die Verwendung eines *RandomizedSearchCV* wenig Sinn, da wir nur wenige Parameter verwenden. Je mehr Parameter ein Modell verwendet, insbesondere wenn diese *float*-Werte enthalten, desto sinnvoller ist der Einsatz eines *RandomizedSearchCV*.

[13]:
```
from sklearn.model_selection import RandomizedSearchCV

parameter_grid = {"weights" : ["uniform", "distance"],
                  "n_neighbors" : range(1,13)}

rgrid = RandomizedSearchCV(KNeighborsClassifier(),␣
 ↪param_distributions=parameter_grid, n_iter=10, scoring="accuracy", n_jobs=-1,
                           verbose=3)

rgrid.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
```

[13]:
```
RandomizedSearchCV(estimator=KNeighborsClassifier(), n_jobs=-1,
                   param_distributions={'n_neighbors': range(1, 13),
                                        'weights': ['uniform', 'distance']},
                   scoring='accuracy', verbose=3)
```

[14]:
```
print(rgrid.best_params_)
```

```
{'weights': 'uniform', 'n_neighbors': 1}
```

1.4 Daten standardisieren / normieren

Vielleicht ist es jemandem aufgefallen: Wir haben den Klassifikator K-Nearest-Neighbors verwendet. Wir haben bereits gelernt, dass man u.a. bei Machine-Learning-Modellen, die mit Abstandsberechnungen arbeiten, die Daten vorher skalieren sollte.

Nun haben wir aber ein kleines Problem: Die Standardisierung der Daten muss für jeden Fold für sich berechnet werden, schließlich verfügt jedes Fold über einen Mittelwert und Standardabweichung (bzw. jeweils eigene Minima und Maxima, sollte man den Min-Max-Scaler verwenden). Wie können wir nun die Daten für jeden Fold für sich standardardisieren?

Hier hilft uns **Pipelining** weiter: Wir definieren in einer Pipeline eine Folge von Preprocessing-Schritten, der letzte Schritt enthält die *fit*-Methode für das Erstellen des jeweiligen Modells. Hier möchten wir also **vor** dem Aufruf der *fit*-Methode die Daten skalieren, hier mit Hilfe der Standardskalierung (Z-Werte).

Dazu erstellen wir ein Objekt der Klasse *Pipeline* und übergeben ein Dictionary. Jedem Schlüssel weisen wir einen beliebigen String als Key zu, im Beispiel *scaler* und *kn* (für "K-Neighbors").

Anschließßend führen wir den Grid-Search durch und übergeben die jeweiligen Parameter. Den Parametern wird jeweils der Key aus dem Dictionary, gefolgt durch zwei Underscores vorangestellt. Dadurch kann GridSearch den jeweiligen Parameter korrekt zuordnen.

[15]:
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler



pipeline = Pipeline([( "scaler" , StandardScaler()),
                     ("kn",KNeighborsClassifier())])

parameter_grid = {"kn__weights" : ["uniform", "distance"],
                  "kn__n_neighbors" : [1, 3, 5, 7, 9, 11]}



rgrid = GridSearchCV(pipeline, parameter_grid, cv=10)
rgrid.fit(X_train, y_train)
```

```
print(rgrid.best_params_)
```

```
{'kn__n_neighbors': 3, 'kn__weights': 'uniform'}
```

[16]:
```
grid_predictions = grid.predict(X_test)
print(classification_report(y_test, grid_predictions))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        33
           1       0.97      1.00      0.98        28
           2       1.00      1.00      1.00        33
           3       0.97      1.00      0.99        34
           4       0.98      1.00      0.99        46
           5       0.98      0.98      0.98        47
           6       0.97      1.00      0.99        35
           7       1.00      0.97      0.99        34
           8       1.00      0.97      0.98        30
           9       0.97      0.93      0.95        40

    accuracy                           0.98       360
   macro avg       0.98      0.98      0.98       360
weighted avg       0.98      0.98      0.98       360
```