

# Curs 8: Optimizarea modelelor, preprocesare, pipelines

## Optimizarea modelelor

În cursul anterior s-a aratat cum se poate folosi k-fold cross validation pentru estimarea performantei unui model. Totodata, s-a aratat o maniera simpla de cautare a valorilor celor mai potrivite pentru hiperparametri - in cazul respectiv, valoarea adecvata a numarului de vecini pentru un model de KNN.

Vom continua aceasta idee pentru mai multi hiperparametri, apoi folosim facilitatile bibliotecii `sklearn` pentru automatizarea procesului.

K-fold cross validation asigura ca fiecare din cele k partitii ale setului de date initial este pe rand folosit ca subset de testare:

```
In [1]: import sklearn
print(f'sklearn version: {sklearn.__version__}')
from sklearn.model_selection import KFold

!pip install prettytable --upgrade
from prettytable import PrettyTable
```

```
sklearn version: 0.24.1
Requirement already satisfied: prettytable in c:\anaconda3\envs\ids\lib\site-packages (2.1.0)
Requirement already satisfied: wcwidth in c:\anaconda3\envs\ids\lib\site-packages (from prettytable) (0.2.5)
```

```
In [2]: kf = KFold(n_splits=3)
splits = kf.split(range(30))
t = PrettyTable(['Iter', 'Train', 'Test'])
t.align = 'l'
for i, data in enumerate(splits):
    t.add_row([i+1, data[0], data[1]])
print(t)
```

```
+-----+-----+-----+-----+
+-----+
| Iter | Train                                     | Test
+-----+-----+-----+-----+
| 1    | [10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29] | [0 1 2 3 4 5 6 7 8 9]
| 2    | [ 0 1 2 3 4 5 6 7 8 9 20 21 22 23 24 25 26 27 28 29] | [10 11 12 13 14 15 16 17 18 19]
| 3    | [ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19] | [20 21 22 23 24 25 26 27 28 29]
+-----+-----+-----+-----+
```

Folosim k-fold cross validation pentru a face evaluarea de modele pentru diferite valori ale hiperparametrilor.

```
In [3]: import numpy as np
import pandas as pd
print('numpy: ', np.__version__)
print('pandas: ', pd.__version__)

!pip install tqdm
from tqdm import tqdm
```

```
numpy: 1.20.2
pandas: 1.2.3
Requirement already satisfied: tqdm in c:\anaconda3\envs\ids\lib\site-packages (4.59.0)
```

```
In [4]: from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

from sklearn.datasets import load_iris

iris = load_iris()
X = iris.data
y = iris.target
```

Pentru k-nearest neighbors vom cauta valorile optime pentru:

- numarul de vecini,  $k \in \{1, \dots, 31\}$
- putere corespunzatoare metricii Minkowski:

$$d(\mathbf{x}, \mathbf{y}) = \left( \sum_{i=1}^n |x_i - y_i|^p \right)^{1/p}$$

```
In [5]: best_score = 0
for k in range(1, 31):
    for p in [1, 2, 3, 4.7]:
        model = KNeighborsClassifier(n_neighbors=k, p=p)
        score = np.mean(cross_val_score(model, X, y, cv=10))
        if score >= best_score:
            best_score = score
            best_params = {'n_neighbors':k, 'p':p}
print('Best score:', best_score)
print('Best params:', best_params)
model = KNeighborsClassifier(n_neighbors=best_params['n_neighbors'], p=best_params['p'])
model.fit(X, y)
y_predicted = model.predict(X)
print('Accuracy on whole set:', accuracy_score(y, y_predicted))
```

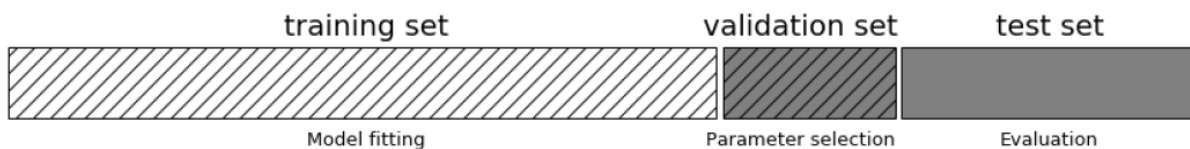
```
Best score: 0.9800000000000001
Best params: {'n_neighbors': 20, 'p': 2}
Accuracy on whole set: 0.98
```

Pentru procesul de mai sus urmatoarele comentarii sunt necesare:

1. strategia implementata se numeste grid search: se cauta peste toate combinatiile de  $30 \times 4$  variante si sa retine cea mai buna; este consumatoare de resurse, dar o prima varianta de lucru acceptabila

2. am dori sa avem o modalitate automatizata de considerare a tuturor combinatiilor de parametri din multimea de valori candidat. Codul devine greu de scris cand sunt multi hiperparametri, fiecare cu multimea proprie de valori candidat
3. estimarea efectuata in final este de cele mai multe ori optimista: optimizarea parametrilor s-a facut peste niste date, care date in final sunt cele folosite pentru evaluarea finala; am ajuns practic sa facem evaluare pe setul de antrenare, ceea ce e o idee proasta. Estimarea finala a performantelor modelului trebuie facuta peste un set de date aparte, care nu a fost folosit nici pentru antrenare, nici pentru validarea modelelor candidat. Altfel, modelul poate face overfitting (invatare foarte buna a datelor de pe setul de antrenare, dar generalizare foarte slaba = rezultatul de pe un set de testare separat sunt foarte slabe).

Pentru ultimul punct se recomanda ca setul sa fie impartit ca mai jos:



Ca atare, va trebui sa rescriem codul astfel:

```
In [6]: X_trainval, X_test, y_trainval, y_test = train_test_split(X, y, test_size=1/5)
best_score = 0
for k in range(1, 31):
    for p in [1, 2, 3, 10]:
        model = KNeighborsClassifier(n_neighbors=k, p=p)
        score = np.mean(cross_val_score(model, X_trainval, y_trainval, cv=10))
        if score >= best_score:
            best_score = score
            best_params = {'n_neighbors':k, 'p':p}
print('Best score:', best_score)
print('Best params:', best_params)

model = KNeighborsClassifier(n_neighbors=best_params['n_neighbors'], p=best_params['p'])
model.fit(X_trainval, y_trainval)
y_predicted = model.predict(X_test)
print(f'accuracy on test ste: {accuracy_score(y_test, y_predicted)}')
```

```
Best score: 0.9833333333333332
Best params: {'n_neighbors': 19, 'p': 1}
accuracy on test ste: 0.9666666666666667
```

Desigur, si implementarea de mai sus e criticabila: s-a facut evaluare pe un singur set de testare, anume cel rezultat dupa impartirea initiala in partiile `*_trainval` si `*_test`. Este totusi o estimare mai corect facuta decat cea precedenta. In realitate, acest stil de lucru este frecvent intalnit: exista un set de testare unic, dar necunoscut la inceput. Singurele date disponibile sunt impartite in *training set* si *validation set* (eventual mai multe) pentru a obtine un model care se spera ca generalizeaza bine = se comporta bine pe setul de testare.

Varianta anterioara se numeste **grid search with cross validation**. Exista clasa `sklearn.model_selection.GridSearchCV` care automatizeaza procesul:

```
In [7]: from sklearn.model_selection import GridSearchCV
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=1/5)
parameter_grid = {'n_neighbors': list(range(1, 10)), 'p': [1, 2, 3, 10]}
```

```
grid_search = GridSearchCV(estimator = KNeighborsClassifier(), param_grid=parameter_grid,
                           return_train_score=True)
grid_search.fit(X_train, y_train)
```

```
Out[7]: GridSearchCV(cv=5, estimator=KNeighborsClassifier(),
                    param_grid={'n_neighbors': [1, 2, 3, 4, 5, 6, 7, 8, 9],
                                'p': [1, 2, 3, 10]},
                    return_train_score=True, scoring='accuracy')
```

```
In [8]: y_estimated = grid_search.predict(X_test)
        print(accuracy_score(y_test, y_estimated))
```

0.9

Valorile optime ale hiperparametrilor sunt retinute in atributul `best_params_` :

```
In [9]: print(grid_search.best_params_)
```

```
{'n_neighbors': 2, 'p': 3}
```

In codul anterior denumirile cheilor din dictionarul `parameter_grid` nu sunt intamplatoare: ele coincid cu numele parametrilor modelului vizat. Instantierea `estimator = KNeighborsClassifier()` se face cu valorile implicite ale parametrilor, apoi insa se ruleaza metode de tip `set_` care seteaza parametrii dati in dictionarul `parameter_grid`.

Pentru cei interesati, valorile de performanta pentru fiecare fold se pot inspecta. Pentru ca acestea sa fie disponibile, este obligatorie setarea parametrului `return_train_score=True` din clasa `GridSearchCV`.

```
In [10]: df_grid_search = pd.DataFrame(grid_search.cv_results_)
        df_grid_search.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 36 entries, 0 to 35
Data columns (total 22 columns):
#   Column                Non-Null Count  Dtype
---  -
0   mean_fit_time          36 non-null    float64
1   std_fit_time           36 non-null    float64
2   mean_score_time        36 non-null    float64
3   std_score_time         36 non-null    float64
4   param_n_neighbors      36 non-null    object
5   param_p                36 non-null    object
6   params                 36 non-null    object
7   split0_test_score      36 non-null    float64
8   split1_test_score      36 non-null    float64
9   split2_test_score      36 non-null    float64
10  split3_test_score      36 non-null    float64
11  split4_test_score      36 non-null    float64
12  mean_test_score        36 non-null    float64
13  std_test_score         36 non-null    float64
14  rank_test_score        36 non-null    int32
15  split0_train_score     36 non-null    float64
16  split1_train_score     36 non-null    float64
17  split2_train_score     36 non-null    float64
18  split3_train_score     36 non-null    float64
19  split4_train_score     36 non-null    float64
20  mean_train_score       36 non-null    float64
21  std_train_score        36 non-null    float64
```

```
dtypes: float64(18), int32(1), object(3)
memory usage: 6.2+ KB
```

```
In [11]: df_grid_search.head()
```

```
Out[11]:
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_n_neighbors	param_p	param_max_iter
0	0.001597	0.000795	0.003988	0.001411	1	1	{'n_neighbors': 1, 'p': 1, 'max_iter': 1000}
1	0.001397	0.000488	0.002991	0.000631	1	2	{'n_neighbors': 1, 'p': 2, 'max_iter': 1000}
2	0.000996	0.000002	0.003391	0.000798	1	3	{'n_neighbors': 1, 'p': 3, 'max_iter': 1000}
3	0.000798	0.000399	0.002793	0.000747	1	10	{'n_neighbors': 1, 'p': 10, 'max_iter': 1000}
4	0.000599	0.000489	0.002393	0.000490	2	1	{'n_neighbors': 2, 'p': 1, 'max_iter': 1000}

5 rows × 22 columns

Pentru situatia in care se doreste evaluarea nu doar pe un singur set de testare, ci in stil cross-validation, se poate face un *nested cross-validation*:

```
In [12]: scores = cross_val_score(GridSearchCV(estimator = KNeighborsClassifier(),
                                                param_grid=parameter_grid,
                                                scoring='accuracy', cv=5), X, y, cv=10)
```

```
In [13]: print(scores.mean())
```

```
0.9666666666666668
```

## Metode de preprocesare

Uneori, inainte de aplicarea vreunui model, este nevoie ca datele de intrare sa fie supuse unor transformari. De exemplu, daca pentru algoritmul k-NN vreuna din trasaturi (fie ea  $F$ ) are valori de ordinul sutelor si celelalte de ordinul unitatilor, atunci distanta dintre doi vectori ar fi dominata de diferenta pe dimensiunea  $F$ ; celelalte dimensiuni nu ar conta prea mult.

Intr-o astfel de situatie se recomanda sa se faca o scalare in prealabil a datelor la intervale comparabile, de ex  $[0, 1]$ .

```
In [ ]: from sklearn.datasets import load_breast_cancer
medical = load_breast_cancer()
X, y = medical.data, medical.target
```

```
In [15]: from typing import List
def get_ranges(X: np.ndarray, columns: List[str] = medical.feature_names) -> pd.DataFrame:
```

```
df = pd.DataFrame(X, columns=columns)
return df.describe().loc[['min', 'max'], :]
```

În modulul `sklearn.preprocessing` se afla clasa `MinMaxScaler` care permite scalarea independenta a trasaturilor. Îl vom demonstra pe un set de date care are trasaturi cu marimi disproporționate.

In [16]: `get_ranges(X)`

Out[16]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	dim
<b>min</b>	6.981	9.71	43.79	143.5	0.05263	0.01938	0.0000	0.0000	0.106	
<b>max</b>	28.110	39.28	188.50	2501.0	0.16340	0.34540	0.4268	0.2012	0.304	

2 rows × 30 columns

In [17]:

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(X)
X = scaler.transform(X)

get_ranges(X)
```

Out[17]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	dim
<b>min</b>	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
<b>max</b>	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	

2 rows × 30 columns

De menționat că secvența `fit` și `transform` se poate apela într-un singur pas:

In [18]:

```
X, y = medical.data, medical.target
X = scaler.fit_transform(X)
get_ranges(X)
```

Out[18]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	dim
<b>min</b>	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
<b>max</b>	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	

2 rows × 30 columns

De regula, setul de date se imparte in doua (in modul naiv): set de antrenare si set de testare. Se presupune ca setul de testare este cunoscut mult mai tarziu decat cel de antrenare. Ca atare, doar cel de antrenare se trece prin preprocesor, iar valorile 'invatate' via `fit` se pastreaza (obiectul de tip `MinMaxScaler` are stare). Ele vor fi folosite pentru scalarea setului de test:

```
In [19]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=1/3, random_state=1)
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
get_ranges(X_test)
```

```
Out[19]:
```

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry
<b>min</b>	0.057504	0.022658	0.054730	0.024772	0.107310	-0.022327	0.000000	0.000000	0.080800
<b>max</b>	0.835297	0.815015	0.841061	0.720042	1.203106	0.787394	0.825445	0.805169	0.775250

2 rows × 30 columns

Se remarca faptul ca, folosindu-se parametrii de scalare din setul de antrenare, nu se poate garanta ca setul de testare este cuprins de asemenea in hipercubul unitate  $[0, 1]^{X.shape[1]}$

Exista si alte metode de preprocesare in modulul `sklearn.preprocessing`.

Sa vedem care e efectul aplicarii scalarii asupra datelor, in contextul KNN:

## KNN fara scalare

```
In [28]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=1/3, random_state=3)
model = KNeighborsClassifier()
model.fit(X_train, y_train)
y_hat_unscaled = model.predict(X_test)
acc_unscaled = accuracy_score(y_test, y_hat_unscaled)
print(f'Acuratete, fara scalare: {acc_unscaled}')
```

Acuratete, fara scalare: 0.9315789473684211

## KNN cu scalare

```
In [29]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=1/3, random_state=3)
model = KNeighborsClassifier()
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
model.fit(X_train, y_train)
y_hat_unscaled = model.predict(X_test)
acc_unscaled = accuracy_score(y_test, y_hat_unscaled)
print(f'Acuratete, fara scalare: {acc_unscaled}')
```

Acuratete, fara scalare: 0.9736842105263158

Desigur, rezultatul de mai sus este dat pentru un singur test, s-ar impune in mod normal mai multe

incercari (cross validation). Folosind pipelines, acest lucru se face usor.

## Pipelines

Se prefera inlantuirea intr-un proces a pasilor: preprocesare si aplicare de model. Exemplificam pentru cazul simplu in care exista un set de antrenare si unul de testare:

```
In [20]: X, y = medical.data, medical.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=1/3)
```

```
In [21]: from sklearn.pipeline import Pipeline
pipe = Pipeline([('scaler', MinMaxScaler()), ('knn', KNeighborsClassifier())])
pipe.fit(X_train, y_train)
y_predicted = pipe.predict(X_test)
print(accuracy_score(y_test, y_predicted))
```

```
0.9736842105263158
```

Pentru cazul in care se vrea k-fold cross validation pentru determinarea valorilor optime pentru hiperparametri, urmata de testare pe un set de testare:

```
In [22]: X_trainval, X_test, y_trainval, y_test = train_test_split(X, y, test_size=1/3)
parameter_grid = {'knn__n_neighbors': list(range(1, 10)),
                  'knn__p': [1, 2, 3, 10]}
grid = GridSearchCV(pipe, param_grid = parameter_grid, scoring = 'accuracy',
                   cv=5, n_jobs=4)
grid.fit(X_trainval, y_trainval)
```

```
Out[22]: GridSearchCV(cv=5,
                    estimator=Pipeline(steps=[('scaler', MinMaxScaler()),
                                              ('knn', KNeighborsClassifier())]),
                    n_jobs=4,
                    param_grid={'knn__n_neighbors': [1, 2, 3, 4, 5, 6, 7, 8, 9],
                               'knn__p': [1, 2, 3, 10]},
                    scoring='accuracy')
```

```
In [23]: y_predicted = grid.predict(X_test)
print(accuracy_score(y_test, y_predicted))
```

```
0.9631578947368421
```

```
In [ ]:
```