Curs 7: Optimizarea modelelor, preprocesare, pipelines

Optimizarea modelelor

In 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]:
```
from sklearn.model_selection import KFold
!pip install prettytable --upgrade
from prettytable import PrettyTable
```

```
Requirement already up-to-date: prettytable in c:\anaconda3\lib\site-packages
(0.7.2)
```

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__)
```

```
numpy:  1.18.1
pandas:  1.0.3
```

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, $k \in \{1, \ldots, 31\}$
- putere corespunzatoare metricii Minkowski:

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

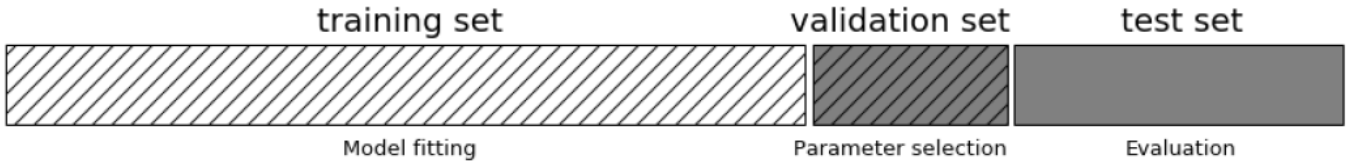
In [23]:
```
best_score = 0
for k in range(1, 31):
    for p in [1, 2, 3, 4.7]:
        model = KNeighborsClassifier(n_neighbors=k, p=p)
        score = np.mean(cross_val_score(model, X, y, cv=10))
        if score >= best_score:
            best_score = score
            best_params = {'n_neighbors':k, 'p':p}
print('Best score:', best_score)
print('Best params:', best_params)
model = KNeighborsClassifier(n_neighbors=best_params['n_neighbors'], p=best_pa
rams['p'])
model.fit(X, y)
y_predicted = model.predict(X)
print('Accuracy on whole set:', accuracy_score(y, y_predicted))
```

```
Best score: 0.9420112781954886
Best params: {'n_neighbors': 6, 'p': 1}
Accuracy on whole set: 0.9525483304042179
```

Pentru procesul de mai sus urmatoarele comentarii sunt necesare:

1. strategia implementata se numeste grid search: se cauta peste toate combinatiile de 30*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 sutn 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, 4.7]:
        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_pa
rams['p'])
model.fit(X_trainval, y_trainval)
y_predicted = model.predict(X_test)
print(accuracy_score(y_test, y_predicted))
```

```
Best score: 0.9833333333333332
Best params: {'n_neighbors': 27, 'p': 3}
0.9
```

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 partitiile *_trainvalid 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, 4.7]}
grid_search = GridSearchCV(estimator = KNeighborsClassifier(), param_grid=para
meter_grid, scoring='accuracy', cv=5,
                           return_train_score=True)
grid_search.fit(X_train, y_train)
```

Out[7]:
```
GridSearchCV(cv=5, error_score=nan,
             estimator=KNeighborsClassifier(algorithm='auto', leaf_size=30,
                                            metric='minkowski',
                                            metric_params=None, n_jobs=None,
                                            n_neighbors=5, p=2,
                                            weights='uniform'),
             iid='deprecated', n_jobs=None,
             param_grid={'n_neighbors': [1, 2, 3, 4, 5, 6, 7, 8, 9],
                         'p': [1, 2, 3, 4.7]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring='accuracy', verbose=0)
```

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

```
0.9
```

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

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

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

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
0	0.000807	0.000404	0.002391	0.001192	1	1
1	0.000603	0.000493	0.000998	0.000001	1	2
2	0.000200	0.000400	0.001590	0.000789	1	3
3	0.000601	0.000491	0.001794	0.000746	1	4.7
4	0.000399	0.000488	0.001406	0.000491	2	1

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(), para
m_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 comparbile, de ex [0, 1].

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

In [14]:
```
from sklearn.datasets import load_breast_cancer
medical = load_breast_cancer()
X, y = medical.data, medical.target
```

In [15]:
```
def print_ranges(X):
    for col_index in range(X.shape[1]):
        column = X[:, col_index]
        print(np.min(column), np.max(column))

print_ranges(X)
```

```
6.981 28.11
9.71 39.28
43.79 188.5
143.5 2501.0
0.05263 0.1634
0.01938 0.3454
0.0 0.4268
0.0 0.2012
0.106 0.304
0.04996 0.09744
0.1115 2.873
0.3602 4.885
0.757 21.98
6.802 542.2
0.001713 0.03113
0.002252 0.1354
0.0 0.396
0.0 0.05279
0.007882 0.07895
0.0008948 0.02984
7.93 36.04
12.02 49.54
50.41 251.2
185.2 4254.0
0.07117 0.2226
0.02729 1.058
0.0 1.252
0.0 0.291
0.1565 0.6638
0.05504 0.2075
```

In [16]:
```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(X)
X = scaler.transform(X)

print_ranges(X)
```

```
0.0 1.0
0.0 1.0
0.0 1.0
0.0 0.9999999999999999
0.0 1.0
0.0 0.9999999999999999
0.0 0.9999999999999999
0.0 0.9999999999999999
0.0 1.0
0.0 1.0
0.0 0.9999999999999999
0.0 1.0
0.0 1.0
0.0 1.0000000000000002
0.0 1.0
0.0 0.9999999999999999
0.0 1.0
0.0 0.9999999999999999
0.0 1.0
0.0 1.0
0.0 0.9999999999999999
0.0 1.0
0.0 1.0
0.0 1.0
0.0 0.9999999999999998
0.0 1.0
0.0 1.0
0.0 1.0
0.0 0.9999999999999999
0.0 1.0
```

De mentionat ca secventa `fit` si `transform` se poate apela intr-un singur pas:

In [17]:
```
X, y = medical.data, medical.target
X = scaler.fit_transform(X)
print_ranges(X)
```

```
0.0 1.0
0.0 1.0
0.0 1.0
0.0 0.9999999999999999
0.0 1.0
0.0 0.9999999999999999
0.0 0.9999999999999999
0.0 0.9999999999999999
0.0 1.0
0.0 1.0
0.0 0.9999999999999999
0.0 1.0
0.0 1.0
0.0 1.0000000000000002
0.0 1.0
0.0 0.9999999999999999
0.0 1.0
0.0 0.9999999999999999
0.0 1.0
0.0 1.0
0.0 0.9999999999999999
0.0 1.0
0.0 1.0
0.0 1.0
0.0 0.9999999999999998
0.0 1.0
0.0 1.0
0.0 1.0
0.0 0.9999999999999999
0.0 1.0
```

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 [18]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=1/3)
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
print_ranges(X_test)
```

```
-0.036700848829792444 0.9661449389136945
-0.028595817328211624 1.2334613743064444
-0.0293782899416703 0.9886185801678762
-0.015214205671924845 1.000861994655633
0.11853389907014528 0.8311817279046674
0.039077357217348646 0.8199496963376482
0.0 1.0009380863039399
0.0 1.0713525026624067
0.07828282828282829 0.8055555555555556
0.006333260537235521 1.0369076217514739
-0.0028689715281812918 0.8816095293434049
-0.0004200844590859870 4 0.6097636472174932
-0.00411144913205369 0.8424496477590474
-0.0008720044140400431 1.0320249408696125
0.03243022741951933 0.7348471971988986
0.005707934028299334 0.782197254183315
0.0 0.7671717171717172
0.0 0.6605417692744838
0.02331569764169529 0.7281476895367818
0.0019139615549382955 0.7588546632947778
-0.04228732207878186 1.1208209202250912
-0.06643244969658257 1.131906739061003
-0.03801401157477917 1.1851964666463597
-0.018448200544373128 1.3488020430794045
0.10942349600475473 0.9154724955424949
0.021577359296019248 0.8140117006723521
0.0 1.07008547008547
0.0 0.9470790378006874
0.05179377524352575 1.2052744119743404
0.02813852813852813 0.773711137347501
```

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` (http://scikit-learn.org/stable/modules/preprocessing.html).

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 [19]:
```
X, y = medical.data, medical.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=1/3)
```

In [20]:
```
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))
```

```
0.9789473684210527
```

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 [21]:
```
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,
4.7]}
grid = GridSearchCV(pipe, param_grid = parameter_grid, scoring = 'accuracy', c
v=5)
grid.fit(X_trainval, y_trainval)
```

Out[21]:
```
GridSearchCV(cv=5, error_score=nan,
             estimator=Pipeline(memory=None,
                                steps=[('scaler',
                                        MinMaxScaler(copy=True,
                                                     feature_range=(0, 1))),
                                       ('knn',
                                        KNeighborsClassifier(algorithm='aut
o',
                                                             leaf_size=30,
                                                             metric='minkowsk
i',
                                                             metric_params=No
ne,
                                                             n_jobs=None,
                                                             n_neighbors=5, p
=2,
                                                             weights='unifor
m'))],
                                verbose=False),
             iid='deprecated', n_jobs=None,
             param_grid={'knn__n_neighbors': [1, 2, 3, 4, 5, 6, 7, 8, 9],
                         'knn__p': [1, 2, 3, 4.7]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='accuracy', verbose=0)
```

In [22]:
```
y_predicted = grid.predict(X_test)
print(accuracy_score(y_test, y_predicted))
```

```
0.9578947368421052
```