08_grid_search

February 28, 2024

1 Efficiently searching for optimal tuning parameters

Lesson 8 from Introduction to Machine Learning with scikit-learn

Note: This notebook uses Python 3.9.1 and scikit-learn 0.23.2. The original notebook (shown in the video) used Python 2.7 and scikit-learn 0.16.

1.1 Agenda

- How can K-fold cross-validation be used to search for an **optimal tuning parameter**?
- How can this process be made **more efficient**?
- How do you search for **multiple tuning parameters** at once?
- What do you do with those tuning parameters before making **real predictions**?
- How can the **computational expense** of this process be reduced?

1.2 Review of K-fold cross-validation

Steps for cross-validation:

- Dataset is split into K "folds" of **equal size**
- Each fold acts as the **testing set** 1 time, and acts as the **training set** K-1 times
- **Average testing performance** is used as the estimate of out-of-sample performance

Benefits of cross-validation:

- More **reliable** estimate of out-of-sample performance than train/test split
- Can be used for selecting **tuning parameters**, choosing between **models**, and selecting **features**

Drawbacks of cross-validation:

- Can be computationally **expensive**

1.3 Review of parameter tuning using `cross_val_score`

Goal: Select the best tuning parameters (aka "hyperparameters") for KNN on the iris dataset

[1]:
```
# added empty cell so that the cell numbering matches the video
```

[2]:
```
from sklearn.datasets import load_iris
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score
```

```
import matplotlib.pyplot as plt
%matplotlib inline
```

[3]:
```
# read in the iris data
iris = load_iris()

# create X (features) and y (response)
X = iris.data
y = iris.target
```

[4]:
```
# 10-fold cross-validation with K=5 for KNN (the n_neighbors parameter)
knn = KNeighborsClassifier(n_neighbors=5)
scores = cross_val_score(knn, X, y, cv=10, scoring='accuracy')
print(scores)
```

```
[1.         0.93333333 1.         1.         0.86666667 0.93333333
 0.93333333 1.         1.         1.        ]
```

[5]:
```
# use average accuracy as an estimate of out-of-sample accuracy
print(scores.mean())
```

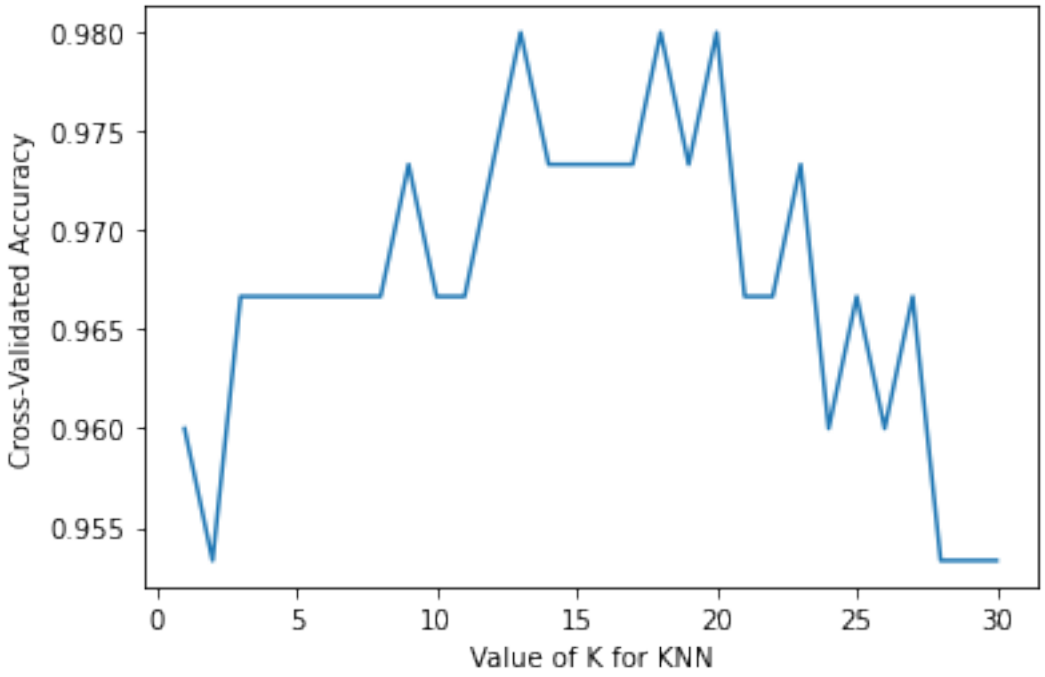
```
0.9666666666666668
```

[6]:
```
# search for an optimal value of K for KNN
k_range = list(range(1, 31))
k_scores = []
for k in k_range:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, X, y, cv=10, scoring='accuracy')
    k_scores.append(scores.mean())
print(k_scores)
```

```
[0.96, 0.9533333333333334, 0.9666666666666666, 0.9666666666666666,
0.9666666666666668, 0.9666666666666668, 0.9666666666666668, 0.9666666666666668,
0.9733333333333334, 0.9666666666666668, 0.9666666666666668, 0.9733333333333334,
0.9800000000000001, 0.9733333333333334, 0.9733333333333334, 0.9733333333333334,
0.9733333333333334, 0.9800000000000001, 0.9733333333333334, 0.9800000000000001,
0.9666666666666666, 0.9666666666666666, 0.9733333333333334, 0.96,
0.9666666666666666, 0.96, 0.9666666666666666, 0.9533333333333334,
0.9533333333333334, 0.9533333333333334]
```

[7]:
```
# plot the value of K for KNN (x-axis) versus the cross-validated accuracy␣
↪(y-axis)
plt.plot(k_range, k_scores)
plt.xlabel('Value of K for KNN')
plt.ylabel('Cross-Validated Accuracy')
```

[7]:
```
Text(0, 0.5, 'Cross-Validated Accuracy')
```



1.4 More efficient parameter tuning using `GridSearchCV`

Allows you to define a **grid of parameters** that will be **searched** using K-fold cross-validation

[8]:
```
from sklearn.model_selection import GridSearchCV
```

[9]:
```
# define the parameter values that should be searched
k_range = list(range(1, 31))
print(k_range)
```

```
[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, 30]
```

[10]:
```
# create a parameter grid: map the parameter names to the values that should be␣
↪searched
param_grid = dict(n_neighbors=k_range)
print(param_grid)
```

```
{'n_neighbors': [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, 30]}
```

[11]:
```
# instantiate the grid
grid = GridSearchCV(knn, param_grid, cv=10, scoring='accuracy')
```

- You can set **`n_jobs = -1`** to run computations in parallel (if supported by your computer and OS)

[12]:
```
# fit the grid with data
grid.fit(X, y)
```

[12]:
```
GridSearchCV(cv=10, estimator=KNeighborsClassifier(n_neighbors=30),
             param_grid={'n_neighbors': [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, 30]},
             scoring='accuracy')
```

[13]:
```
# view the results as a pandas DataFrame
import pandas as pd
pd.DataFrame(grid.cv_results_)[['mean_test_score', 'std_test_score', 'params']]
```

[13]:

	mean_test_score	std_test_score	params
0	0.960000	0.053333	{'n_neighbors': 1}
1	0.953333	0.052068	{'n_neighbors': 2}
2	0.966667	0.044721	{'n_neighbors': 3}
3	0.966667	0.044721	{'n_neighbors': 4}
4	0.966667	0.044721	{'n_neighbors': 5}
5	0.966667	0.044721	{'n_neighbors': 6}
6	0.966667	0.044721	{'n_neighbors': 7}
7	0.966667	0.044721	{'n_neighbors': 8}
8	0.973333	0.032660	{'n_neighbors': 9}
9	0.966667	0.044721	{'n_neighbors': 10}
10	0.966667	0.044721	{'n_neighbors': 11}
11	0.973333	0.032660	{'n_neighbors': 12}
12	0.980000	0.030551	{'n_neighbors': 13}
13	0.973333	0.044222	{'n_neighbors': 14}
14	0.973333	0.032660	{'n_neighbors': 15}
15	0.973333	0.032660	{'n_neighbors': 16}
16	0.973333	0.032660	{'n_neighbors': 17}
17	0.980000	0.030551	{'n_neighbors': 18}
18	0.973333	0.032660	{'n_neighbors': 19}
19	0.980000	0.030551	{'n_neighbors': 20}
20	0.966667	0.033333	{'n_neighbors': 21}
21	0.966667	0.033333	{'n_neighbors': 22}
22	0.973333	0.032660	{'n_neighbors': 23}
23	0.960000	0.044222	{'n_neighbors': 24}
24	0.966667	0.033333	{'n_neighbors': 25}
25	0.960000	0.044222	{'n_neighbors': 26}
26	0.966667	0.044721	{'n_neighbors': 27}
27	0.953333	0.042687	{'n_neighbors': 28}
28	0.953333	0.042687	{'n_neighbors': 29}
29	0.953333	0.042687	{'n_neighbors': 30}

[14]:
```
# examine the first result
print(grid.cv_results_['params'][0])
```

```
print(grid.cv_results_['mean_test_score'][0])
```

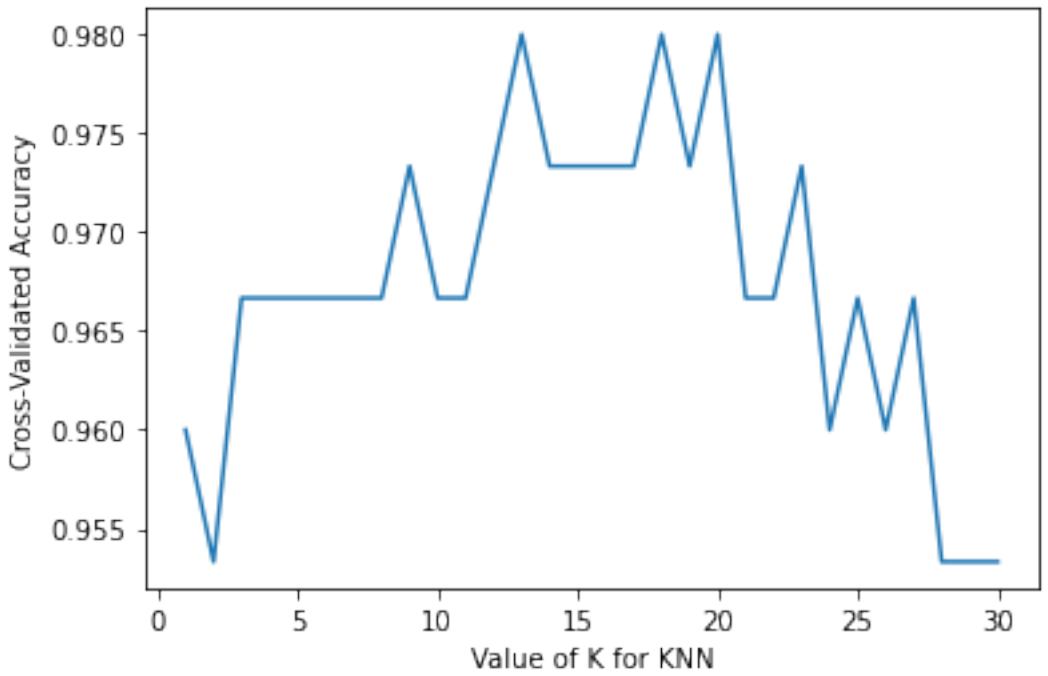
```
{'n_neighbors': 1}
0.96
```

[15]:
```
# print the array of mean scores only
grid_mean_scores = grid.cv_results_['mean_test_score']
print(grid_mean_scores)
```

```
[0.96       0.95333333 0.96666667 0.96666667 0.96666667 0.96666667
 0.96666667 0.96666667 0.97333333 0.96666667 0.96666667 0.97333333
 0.98       0.97333333 0.97333333 0.97333333 0.97333333 0.98
 0.97333333 0.98       0.96666667 0.96666667 0.97333333 0.96
 0.96666667 0.96       0.96666667 0.95333333 0.95333333 0.95333333]
```

[16]:
```
# plot the results
plt.plot(k_range, grid_mean_scores)
plt.xlabel('Value of K for KNN')
plt.ylabel('Cross-Validated Accuracy')
```

[16]:
```
Text(0, 0.5, 'Cross-Validated Accuracy')
```



[17]:
```
# examine the best model
print(grid.best_score_)
print(grid.best_params_)
```

```
print(grid.best_estimator_)
```

```
0.9800000000000001
{'n_neighbors': 13}
KNeighborsClassifier(n_neighbors=13)
```

1.5 Searching multiple parameters simultaneously

- **Example:** tuning `max_depth` and `min_samples_leaf` for a `DecisionTreeClassifier`
- Could tune parameters **independently**: change `max_depth` while leaving `min_samples_leaf` at its default value, and vice versa
- But, best performance might be achieved when **neither parameter** is at its default value

[18]:
```
# define the parameter values that should be searched
k_range = list(range(1, 31))
weight_options = ['uniform', 'distance']
```

[19]:
```
# create a parameter grid: map the parameter names to the values that should be␣
↪searched
param_grid = dict(n_neighbors=k_range, weights=weight_options)
print(param_grid)
```

```
{'n_neighbors': [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, 30], 'weights': ['uniform',
'distance']}
```

[20]:
```
# instantiate and fit the grid
grid = GridSearchCV(knn, param_grid, cv=10, scoring='accuracy')
grid.fit(X, y)
```

[20]:
```
GridSearchCV(cv=10, estimator=KNeighborsClassifier(n_neighbors=30),
             param_grid={'n_neighbors': [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, 30],
                         'weights': ['uniform', 'distance']},
             scoring='accuracy')
```

[21]:
```
# view the results
pd.DataFrame(grid.cv_results_)[['mean_test_score', 'std_test_score', 'params']]
```

[21]:

	mean_test_score	std_test_score	\
0	0.960000	0.053333	
1	0.960000	0.053333	
2	0.953333	0.052068	
3	0.960000	0.053333	
4	0.966667	0.044721	
5	0.966667	0.044721	
6	0.966667	0.044721	
7	0.966667	0.044721	

8	0.966667	0.044721
9	0.966667	0.044721
10	0.966667	0.044721
11	0.966667	0.044721
12	0.966667	0.044721
13	0.966667	0.044721
14	0.966667	0.044721
15	0.966667	0.044721
16	0.973333	0.032660
17	0.973333	0.032660
18	0.966667	0.044721
19	0.973333	0.032660
20	0.966667	0.044721
21	0.973333	0.032660
22	0.973333	0.032660
23	0.973333	0.044222
24	0.980000	0.030551
25	0.973333	0.032660
26	0.973333	0.044222
27	0.973333	0.032660
28	0.973333	0.032660
29	0.980000	0.030551
30	0.973333	0.032660
31	0.973333	0.032660
32	0.973333	0.032660
33	0.980000	0.030551
34	0.980000	0.030551
35	0.973333	0.032660
36	0.973333	0.032660
37	0.980000	0.030551
38	0.980000	0.030551
39	0.966667	0.044721
40	0.966667	0.033333
41	0.966667	0.044721
42	0.966667	0.033333
43	0.966667	0.044721
44	0.973333	0.032660
45	0.973333	0.032660
46	0.960000	0.044222
47	0.973333	0.032660
48	0.966667	0.033333
49	0.973333	0.032660
50	0.960000	0.044222
51	0.966667	0.044721
52	0.966667	0.044721
53	0.980000	0.030551
54	0.953333	0.042687

```
55         0.973333        0.032660
56         0.953333        0.042687
57         0.973333        0.032660
58         0.953333        0.042687
59         0.966667        0.033333
```

```
                                        params
0     {'n_neighbors': 1, 'weights': 'uniform'}
1    {'n_neighbors': 1, 'weights': 'distance'}
2     {'n_neighbors': 2, 'weights': 'uniform'}
3    {'n_neighbors': 2, 'weights': 'distance'}
4     {'n_neighbors': 3, 'weights': 'uniform'}
5    {'n_neighbors': 3, 'weights': 'distance'}
6     {'n_neighbors': 4, 'weights': 'uniform'}
7    {'n_neighbors': 4, 'weights': 'distance'}
8     {'n_neighbors': 5, 'weights': 'uniform'}
9    {'n_neighbors': 5, 'weights': 'distance'}
10    {'n_neighbors': 6, 'weights': 'uniform'}
11   {'n_neighbors': 6, 'weights': 'distance'}
12    {'n_neighbors': 7, 'weights': 'uniform'}
13   {'n_neighbors': 7, 'weights': 'distance'}
14    {'n_neighbors': 8, 'weights': 'uniform'}
15   {'n_neighbors': 8, 'weights': 'distance'}
16    {'n_neighbors': 9, 'weights': 'uniform'}
17   {'n_neighbors': 9, 'weights': 'distance'}
18   {'n_neighbors': 10, 'weights': 'uniform'}
19  {'n_neighbors': 10, 'weights': 'distance'}
20   {'n_neighbors': 11, 'weights': 'uniform'}
21  {'n_neighbors': 11, 'weights': 'distance'}
22   {'n_neighbors': 12, 'weights': 'uniform'}
23  {'n_neighbors': 12, 'weights': 'distance'}
24   {'n_neighbors': 13, 'weights': 'uniform'}
25  {'n_neighbors': 13, 'weights': 'distance'}
26   {'n_neighbors': 14, 'weights': 'uniform'}
27  {'n_neighbors': 14, 'weights': 'distance'}
28   {'n_neighbors': 15, 'weights': 'uniform'}
29  {'n_neighbors': 15, 'weights': 'distance'}
30   {'n_neighbors': 16, 'weights': 'uniform'}
31  {'n_neighbors': 16, 'weights': 'distance'}
32   {'n_neighbors': 17, 'weights': 'uniform'}
33  {'n_neighbors': 17, 'weights': 'distance'}
34   {'n_neighbors': 18, 'weights': 'uniform'}
35  {'n_neighbors': 18, 'weights': 'distance'}
36   {'n_neighbors': 19, 'weights': 'uniform'}
37  {'n_neighbors': 19, 'weights': 'distance'}
38   {'n_neighbors': 20, 'weights': 'uniform'}
39  {'n_neighbors': 20, 'weights': 'distance'}
```

```
 40   {'n_neighbors': 21, 'weights': 'uniform'}
 41  {'n_neighbors': 21, 'weights': 'distance'}
 42   {'n_neighbors': 22, 'weights': 'uniform'}
 43  {'n_neighbors': 22, 'weights': 'distance'}
 44   {'n_neighbors': 23, 'weights': 'uniform'}
 45  {'n_neighbors': 23, 'weights': 'distance'}
 46   {'n_neighbors': 24, 'weights': 'uniform'}
 47  {'n_neighbors': 24, 'weights': 'distance'}
 48   {'n_neighbors': 25, 'weights': 'uniform'}
 49  {'n_neighbors': 25, 'weights': 'distance'}
 50   {'n_neighbors': 26, 'weights': 'uniform'}
 51  {'n_neighbors': 26, 'weights': 'distance'}
 52   {'n_neighbors': 27, 'weights': 'uniform'}
 53  {'n_neighbors': 27, 'weights': 'distance'}
 54   {'n_neighbors': 28, 'weights': 'uniform'}
 55  {'n_neighbors': 28, 'weights': 'distance'}
 56   {'n_neighbors': 29, 'weights': 'uniform'}
 57  {'n_neighbors': 29, 'weights': 'distance'}
 58   {'n_neighbors': 30, 'weights': 'uniform'}
 59  {'n_neighbors': 30, 'weights': 'distance'}
```

[22]:
```
# examine the best model
print(grid.best_score_)
print(grid.best_params_)
```

```
0.9800000000000001
{'n_neighbors': 13, 'weights': 'uniform'}
```

1.6 Using the best parameters to make predictions

[23]:
```
# train your model using all data and the best known parameters
knn = KNeighborsClassifier(n_neighbors=13, weights='uniform')
knn.fit(X, y)

# make a prediction on out-of-sample data
knn.predict([[3, 5, 4, 2]])
```

[23]:
```
array([1])
```

[24]:
```
# shortcut: GridSearchCV automatically refits the best model using all of the␣
  ↪data
grid.predict([[3, 5, 4, 2]])
```

[24]:
```
array([1])
```

1.7 Reducing computational expense using `RandomizedSearchCV`

- Searching many different parameters at once may be computationally infeasible

- `RandomizedSearchCV` searches a subset of the parameters, and you control the computational "budget"

[25]:
```
from sklearn.model_selection import RandomizedSearchCV
```

[26]:
```
# specify "parameter distributions" rather than a "parameter grid"
param_dist = dict(n_neighbors=k_range, weights=weight_options)
```

- **Important:** Specify a continuous distribution (rather than a list of values) for any continous parameters

[27]:
```
# n_iter controls the number of searches
rand = RandomizedSearchCV(knn, param_dist, cv=10, scoring='accuracy',␣
 ↪n_iter=10, random_state=5)
rand.fit(X, y)
pd.DataFrame(rand.cv_results_)[['mean_test_score', 'std_test_score', 'params']]
```

[27]:

	mean_test_score	std_test_score	params
0	0.973333	0.032660	{'weights': 'distance', 'n_neighbors': 16}
1	0.966667	0.033333	{'weights': 'uniform', 'n_neighbors': 22}
2	0.980000	0.030551	{'weights': 'uniform', 'n_neighbors': 18}
3	0.966667	0.044721	{'weights': 'uniform', 'n_neighbors': 27}
4	0.953333	0.042687	{'weights': 'uniform', 'n_neighbors': 29}
5	0.973333	0.032660	{'weights': 'distance', 'n_neighbors': 10}
6	0.966667	0.044721	{'weights': 'distance', 'n_neighbors': 22}
7	0.973333	0.044222	{'weights': 'uniform', 'n_neighbors': 14}
8	0.973333	0.044222	{'weights': 'distance', 'n_neighbors': 12}
9	0.973333	0.032660	{'weights': 'uniform', 'n_neighbors': 15}

[28]:
```
# examine the best model
print(rand.best_score_)
print(rand.best_params_)
```

```
0.9800000000000001
{'weights': 'uniform', 'n_neighbors': 18}
```

[29]:
```
# run RandomizedSearchCV 20 times (with n_iter=10) and record the best score
best_scores = []
for _ in range(20):
    rand = RandomizedSearchCV(knn, param_dist, cv=10, scoring='accuracy',␣
 ↪n_iter=10)
    rand.fit(X, y)
    best_scores.append(round(rand.best_score_, 3))
print(best_scores)
```

```
[0.98, 0.98, 0.98, 0.98, 0.973, 0.98, 0.973, 0.98, 0.98, 0.98, 0.973, 0.98,
0.98, 0.973, 0.973, 0.98, 0.98, 0.973, 0.973, 0.98]
```

1.8 Resources

- scikit-learn documentation: Grid search, GridSearchCV, RandomizedSearchCV
- Timed example: Comparing randomized search and grid search
- scikit-learn workshop by Andreas Mueller: Video segment on randomized search (3 minutes), related notebook
- Paper by Yoshua Bengio: Random Search for Hyper-Parameter Optimization

1.9 Comments or Questions?

- Email: `kevin@dataschool.io`
- Website: https://www.dataschool.io
- Twitter: @justmarkham