1. 그리드 서치

1 그리드 서치 개요

개요

그리드 서치는 각 결정 변수의 실행 가능 공간에서 탐색할 값 일부를 선택하여 구성한 새로운 탐색 공간인 그리드 안에 있는 모든 해를 비교하는 방법입니다.

예제: k-최근접 이웃의 하이퍼파라미터를 튜닝

- 이웃 수(n_neighbors): {3, 5, 7}
- 거리 척도(metric): {Manhattan, Euclidean}
- 평가 척도: 정확도

그리드

		n_neighbors		
		3	5	7
metric	manhattan	(manhattan, 3)	(manhattan, 5)	(manhattan, 7)
	euclidean	(euclidean, 3)	(euclidean, 5)	(euclidean, 7)

→

탐색 결과

		n_neighbors		
		3	5	7
metric	manhattan	0.75	0.65	0.70
	euclidean	0.80	0.70	0.60

↓

(euclidean, 3)을 사용

특징 및 장단점

- 매우 직관적이고 구현이 쉬워, 자주 사용됨
- 실험과 경험 등을 통해 그리드를 잘 설계한다면 좋은 해를 찾을 수 있음
- 그리드를 잘 설계하기가 매우 어려우며, 데이터에 따른 적절한 그리드가 다를 수 있음
- 그리드가 복잡할수록 소요되는 시간이 기하급수적으로 증가함

1. 그리드 서치

2 최댓값 및 최솟값 탐색 알고리즘

최댓값과 최솟값 탐색 알고리즘: 필요성

그리드 서치를 비롯한 하이퍼파라미터 튜닝 해법 대부분은 여러 하이퍼파라미터를 평가하고 비교해서 최적의 하이퍼파라미터를 찾습니다. 이때, 하이퍼파라미터와 점수를 전부 저장하면 메모리 관리 측면에서 매우 비효율적입니다.

탐색 결과를 활용한 최적 하이퍼 파라미터 선택

$$\boldsymbol{h}^* = \underset{i=1,2,\cdots,n}{\operatorname{argmax}} s^{(i)}$$

- 점수가 가장 큰 하이퍼 파라미터를 $\boldsymbol{h}^*$에 저장함
- $\mathrm{H} = \left(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)}, \cdots, \boldsymbol{h}^{(n)}\right)$: 탐색한 하이퍼파라미터 집합
- $\mathrm{S} = \left(s^{(1)}, s^{(2)}, \cdots, s^{(n)}\right)$: 평가 결과

탐색 결과를 활용한 최적 하이퍼 파라미터 선택 예제

```
1  import numpy as np
2  def find_optimal_h(H, S):
3      idx = np.argmax(S)
4      return H[idx]
5
6  H = ["H1", "H2", "H3", "H4", "H5"]
7  S = [0.8, 0.7, 0.9, 0.6, 0.7]
8  print(find_optimal_h(H, S))
```

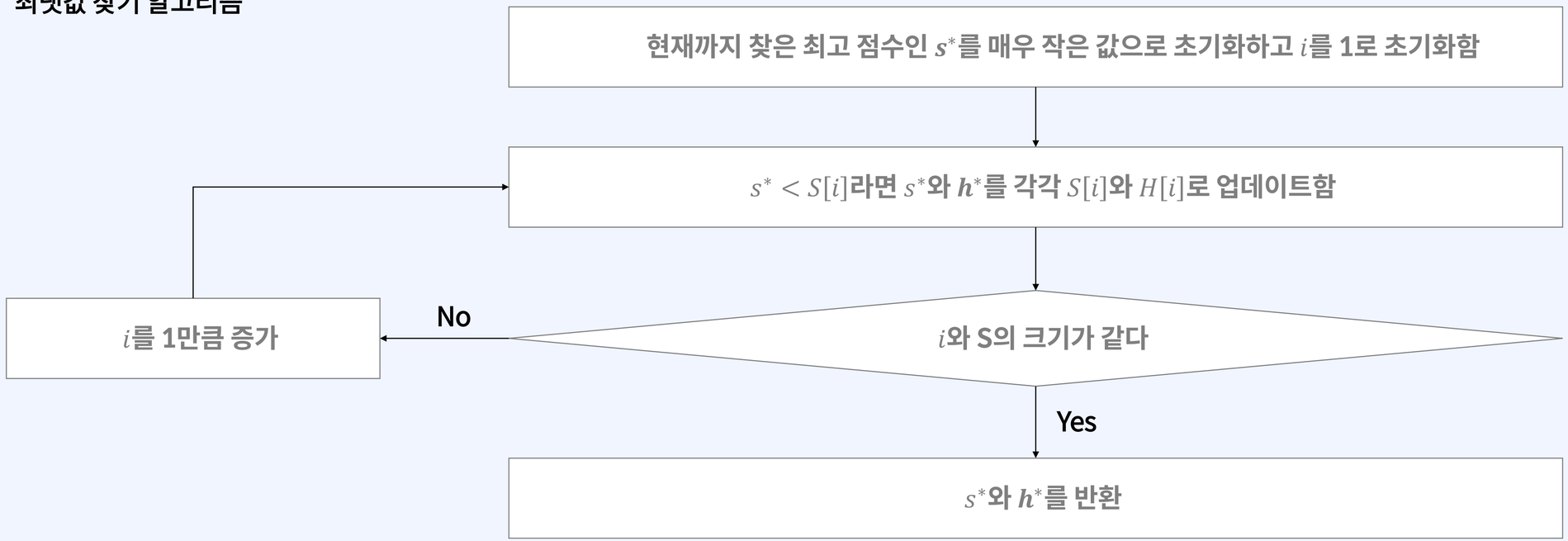
- **라인 2:** 탐색한 하이퍼파라미터 목록 H와 점수 목록 S를 입력받습니다.
- **라인 3:** np.argmax 함수를 사용해 S의 최댓값의 인덱스를 idx에 저장합니다.
- **라인 4:** S의 최댓값의 인덱스를 H의 인덱스로 사용합니다.

```
H3
```

최댓값과 최솟값 탐색 알고리즘

효율적인 메모리 관리를 위해, 매 이터레이션마다 최종 점수와 하이퍼파라미터를 업데이트해야 합니다.

최댓값 찾기 알고리즘



최댓값과 최솟값 탐색 알고리즘: 예시

S: [0.8, 0.7, 0.9, 0.6, 0.7]

H: [H1, H2, H3, H4, H5]

이터레이션 (i)	$S[i]$	비교 연산 결과	s^*	h^*
0	-	-	-inf	-
1	0.8	-inf < 0.8	0.8	H1
2	0.7	0.8 > 0.7	0.8	H1
3	0.9	0.8 < 0.9	0.9	H3
4	0.6	0.9 > 0.6	0.9	H3
5	0.7	0.9 > 0.7	0.9	H3

(H3, 0.9)

최댓값과 최솟값 탐색 알고리즘: 파이썬 구현

최댓값 탐색 알고리즘 예제

```
1  def find_optimal_h_update(H, S):
2      current_max_value = -np.inf
3      for h, s in zip(H, S):
4          if s > current_max_value:
5              current_max_value = s
6              h_star = h
7      return h_star
8
9  H = ["H1", "H2", "H3", "H4", "H5"]
10 S = [0.8, 0.7, 0.9, 0.6, 0.7]
11 print(find_optimal_h_update(H, S))
```

- **라인 2:** 현재까지 찾은 최댓값을 음의 무한대로 초기화합니다.
- **라인 3:** H와 S를 각각 h와 s로 순회합니다.
- **라인 4~6:** s가 current_max_value보다 크다면 current_max_value를 s로, h_star를 h로 업데이트합니다.
- **라인 9~11:** find_optimal_h_update를 사용해 최적의 하이퍼파라미터를 찾습니다.

1. 그리드 서치

3 GridSearchCV를 이용한 그리드 서치

예제 데이터 불러오기

그리드 서치를 실습할 데이터를 불러옵니다. 이때, k-겹 교차 검증을 사용해 해를 평가할 예정이므로 train_test_split으로 학습 데이터와 평가 데이터로 분리하지는 않았습니다.

예제 데이터 불러오기

```
1  import pandas as pd
2  df1 = pd.read_csv("../../data/classification/optdigits.csv")
3  df2 = pd.read_csv("../../data/regression/baseball.csv")
4  X1 = df1.drop('y', axis = 1)
5  y1 = df1['y']
6  X2 = df2.drop('y', axis = 1)
7  y2 = df2['y']
```

GridSearchCV 클래스
: 주요 인자

사이킷런의 model_selection.GridSearchCV 클래스를 이용하면 손쉽게 하이퍼파라미터 튜닝을 위한 그리드 서치를 구현할 수 있습니다. 이 클래스는 주어진 그리드에 속하는 모든 해를 k-겹 교차 검증 방식으로 평가하여 가장 좋은 하이퍼 파라미터를 찾습니다.

주요 인자

인자	설명
estimator	분류 및 회귀 모델 인스턴스
param_grid	파라미터 그리드 (사전 자료형)
cv	폴드 개수
scoring	평가 척도
refit	가장 좋은 성능의 하이퍼파라미터를 갖는 모델을 전체 데이터로 재학습할지 여부

- param_grid의 키는 estimator의 인자이고 값은 해당 인자가 갖는 값으로 구성된 배열임

(예시) grid = {"n_neighbors": [3, 5, 7], "metric":["euclidean", "manhattan"]}

- scoring은 각 하이퍼파라미터를 평가하는 데 사용하는 평가 척도로 'accuracy', 'f1', 'neg_mean_absolute_error'와 같이 문자열로 입력함. 여기서 neg_mean_absolute_error는 다른 지표처럼 값이 크면 클수록 좋다고 일관되게 평가할 수 있도록 MAE에 마이너스 부호를 붙인 것에 불과함

GridSearchCV 클래스
: 주요 메서드 및 속성

주요 메서드 및 속성

메서드/속성	설명
fit	입력한 그리드 내의 모든 하이퍼 파라미터를 평가
predict	가장 우수한 하이퍼 파라미터를 갖는 모델로 예측을 수행
cv_results_	그리드 서치를 이용한 탐색 결과
best_estimator_	점수가 가장 높은 모델 인스턴스
best_score_	최고 점수
best_params_	점수가 가장 높은 하이퍼파라미터

그리드 서치

GridSearchCV를 이용해 k-최근접 이웃(분류)의 하이퍼파라미터를 튜닝해보겠습니다.

GridSearchCV를 이용한 하이퍼파라미터 튜닝 예제

```
1  from sklearn.model_selection import GridSearchCV
2  from sklearn.neighbors import KNeighborsClassifier
3  clf = GridSearchCV(estimator = KNeighborsClassifier(),
4                     cv = 5,
5                     param_grid = grid,
6                     scoring = "accuracy").fit(X1, y1)
7  result = pd.DataFrame(clf.cv_results_)
8  display(result[['params', 'mean_test_score', 'mean_fit_time']])
```

	params	mean_test_score	mean_fit_time
0	{'metric': 'euclidean', 'n_neighbors': 3}	0.982918	0.009826
1	{'metric': 'euclidean', 'n_neighbors': 5}	0.982562	0.010027
2	{'metric': 'euclidean', 'n_neighbors': 7}	0.983452	0.009425
3	{'metric': 'manhattan', 'n_neighbors': 3}	0.978470	0.008423
4	{'metric': 'manhattan', 'n_neighbors': 5}	0.978648	0.010027
5	{'metric': 'manhattan', 'n_neighbors': 7}	0.978292	0.009227

- params: param_grid의 하이퍼파라미터
- mean_test_score: k-겹 교차 검증에서 k번 평가한 결과의 평균값
- mean_fit_time: 평균 학습 시간

그리드 서치
(계속)

GridSearchCV를 이용해 k-최근접 이웃(분류)의 하이퍼파라미터를 튜닝해보겠습니다.

GridSearchCV를 이용한 하이퍼파라미터 튜닝 예제

```
1  print(clf.best_estimator_)
2  print(clf.best_score_)
3  print(clf.best_params_)
```

```
KNeighborsClassifier(metric='euclidean', n_neighbors=7)
0.9834519572953736
{'metric': 'euclidean', 'n_neighbors': 7}
```

그리드 서치
(계속)

GridSearchCV를 이용해 k-최근접 이웃(회귀)의 하이퍼파라미터를 튜닝해보겠습니다.

GridSearchCV를 이용한 하이퍼파라미터 튜닝 예제

```
1  from sklearn.neighbors import KNeighborsRegressor
2  clf = GridSearchCV(estimator = KNeighborsRegressor(),
3                     cv = 5,
4                     param_grid = grid,
5                     scoring = "neg_mean_absolute_error").fit(X2, y2)
6
7  result = pd.DataFrame(clf.cv_results_)
8  display(result[['params', 'mean_test_score', 'mean_fit_time']])
```

	params	mean_test_score	mean_fit_time
0	{'metric': 'euclidean', 'n_neighbors': 3}	-666.301580	0.006816
1	{'metric': 'euclidean', 'n_neighbors': 5}	-651.092379	0.006016
2	{'metric': 'euclidean', 'n_neighbors': 7}	-653.397034	0.005615
3	{'metric': 'manhattan', 'n_neighbors': 3}	-693.410097	0.005214
4	{'metric': 'manhattan', 'n_neighbors': 5}	-655.554548	0.005414
5	{'metric': 'manhattan', 'n_neighbors': 7}	-644.461514	0.006216

- "neg_mean_absolute_error"는 MAE에 마이너스 부호만 붙인 것임
- mean_test_score가 -644.461514로 가장 큰 5번 행의 파라미터가 가장 좋다고 할 수 있음

1. 그리드 서치

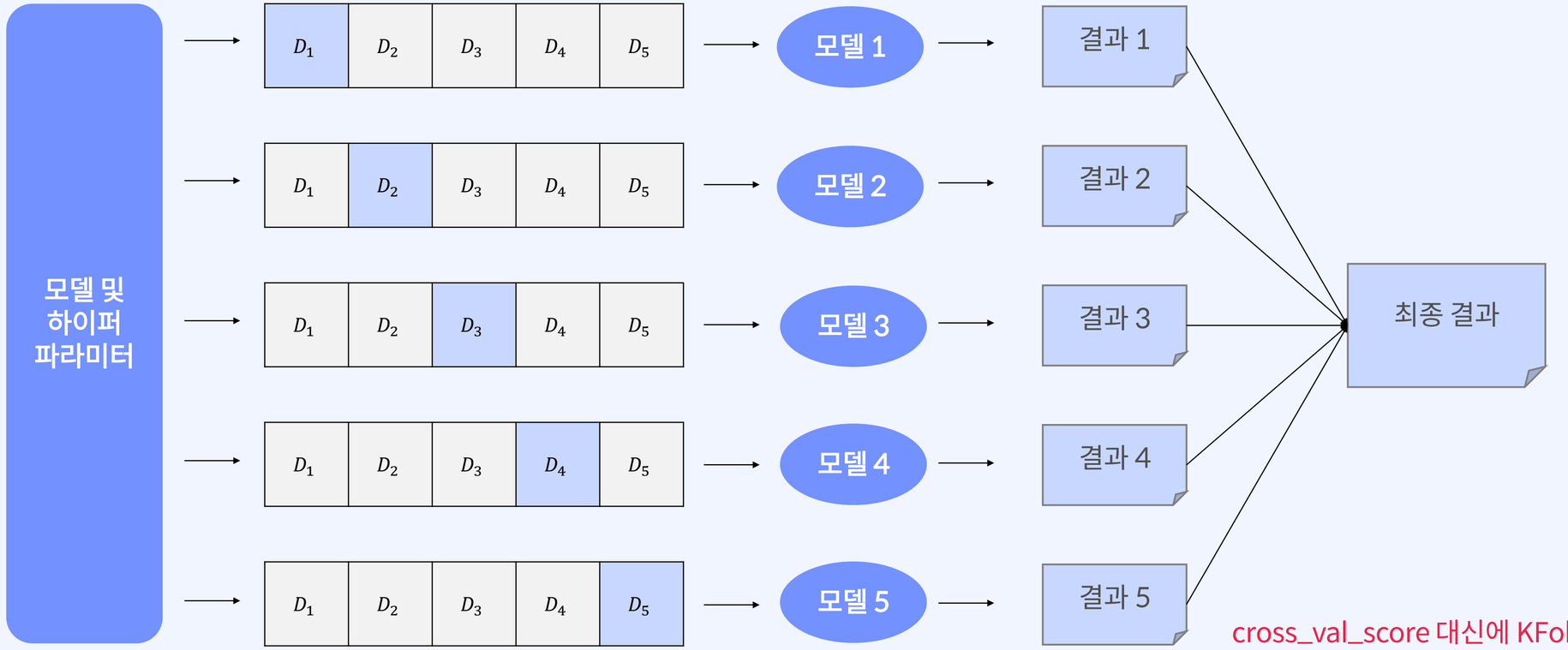
4 ParameterGrid를 이용한 그리드 서치

GridSearchCV의 문제점

GridSearchCV 클래스는 사용하기 매우 간편하지만, 학습 데이터로 전처리 모델을 학습하고 전체 데이터에 적용하는 등 적절하게 데이터를 전처리하기 어렵습니다.

GridSearchCV를 이용하는 경우

모델별 적절한 전처리 불가 (예: 재샘플링 절대 불가)



cross_val_score 대신에 KFold 클래스를 사용하는 것과 같은 이유로 GridSearchCV를 실무에서 잘 사용하지 않음

ParameterGrid 함수

ParameterGrid 함수는 사전 형태의 하이퍼파라미터 그리드를 입력받아 그리드의 모든 해를 순회하는 이터레이터를 반환합니다.

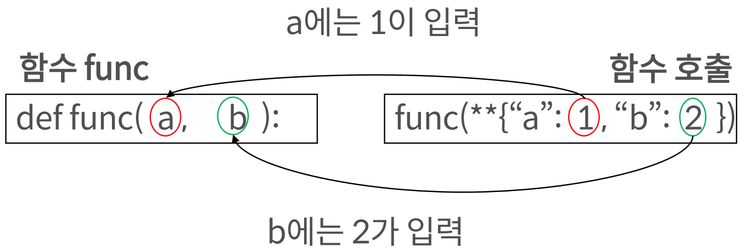
ParameterGrid 함수 예시

```
1  from sklearn.model_selection import ParameterGrid
2  for param in ParameterGrid(grid):
3          print(param)
```

```
{'metric': 'euclidean', 'n_neighbors': 3}
{'metric': 'euclidean', 'n_neighbors': 5}
{'metric': 'euclidean', 'n_neighbors': 7}
{'metric': 'manhattan', 'n_neighbors': 3}
{'metric': 'manhattan', 'n_neighbors': 5}
{'metric': 'manhattan', 'n_neighbors': 7}
```

**kwargs 문법

사전 자료형으로 클래스나 함수의 인자를 설정하는 데 필요한 파이썬 문법으로 **kwargs가 있습니다.



- ParameterGrid 인스턴스를 순회하는 변수 param은 사전 자료형으로 키가 모델 인스턴스의 인자이고 값이 해당 인자의 값임
- {"인자": "값"} 형태로 구성된 사전 자료형에 **를 붙이고 입력하면 대응되는 인자에 해당 값이 입력됨

**kwargs 문법 예시

```
1  param = {"metric": "euclidean", "n_neighbors": 3}
2  KNeighborsClassifier(**param)
```

```
KNeighborsClassifier(metric='euclidean', n_neighbors=3)
```

ParameterGrid와 KFold를 이용한 그리드 서치

ParameterGrid와 KFold를 이용한 그리드 서치 예시 (분류)

```
1  from sklearn.metrics import *
2  from sklearn.model_selection import KFold
3  grid = ParameterGrid(grid)
4  kf = KFold(n_splits = 5)
5  best_score = -1
6  for param in grid:
7      total_score = 0
8      for train_index, test_index in kf.split(X1):
9          X1_train = X1.loc[train_index]
10         X1_test = X1.loc[test_index]
11         y1_train = y1.loc[train_index]
12         y1_test = y1.loc[test_index]
13
14         model = KNeighborsClassifier(**param).fit(X1_train, y1_train)
15         y1_pred = model.predict(X1_test)
16         score = accuracy_score(y1_test, y1_pred)
17         total_score += score / 5
18     if total_score > best_score:
19         best_score = total_score
20         best_parameter = param
```

- **라인 3:** 사전 자료형인 grid를 ParameterGrid를 사용해 변환합니다.
- **라인 5:** 최고 점수 best_score를 -1로 초기화합니다. 평가 지표인 정확도는 아무리 작아도 0이므로 –1로 초기화하더라도 무방합니다.
- **라인 6~7:** grid에 있는 하이퍼파라미터를 순회하면서 평가 점수 total_score를 0으로 초기화합니다.
- **라인 8~17:** KFold 클래스를 사용해 5-겹 교차 검증을 수행합니다. score는 i(i= 0, 1, 2, 3, 4)번째 폴드로 평가한 정확도이며, total_score에 score/5를 더함으로써 평균 정확도를 계산합니다.
- **라인 18~20:** 현재 탐색 중인 파라미터의 점수인 total_score가 현재까지 찾은 최고 점수인 best_score보다 크다면 best_score와 best_parameter를 업데이트합니다.

ParameterGrid와 KFold를 이용한 그리드 서치 (계속)

ParameterGrid와 KFold를 이용한 그리드 서치 예시 (분류)

```
1  print(best_parameter, best_score)
```

```
{'metric': 'euclidean', 'n_neighbors': 7}  0.9830960854092526
```

ParameterGrid와 KFold를 이용한 그리드 서치 (계속)

ParameterGrid와 KFold를 이용한 그리드 서치 예시 (회귀)

```
1   kf = KFold(n_splits = 5)
2   best_score = np.inf
3   for param in grid:
4       total_score = 0
5       for train_index, test_index in kf.split(X2):
6           X2_train = X2.loc[train_index]
7           X2_test = X2.loc[test_index]
8           y2_train = y2.loc[train_index]
9           y2_test = y2.loc[test_index]
10
11          model = KNeighborsClassifier(**param).fit(X2_train, y2_train)
12          y2_pred = model.predict(X2_test)
13          score = mean_absolute_error(y2_test, y2_pred)
14          total_score += score / 5
15      if total_score < best_score:
16          best_score = total_score
17          best_parameter = param
18  print(best_parameter, best_score)
```

- **라인 2:** MAE는 작으면 작을수록 좋으므로 best_score를 무한대로 초기화합니다.
- **라인 15:** 현재 탐색 중인 파라미터의 MAE인 total_score가 지금까지 찾은 최저 MAE인 best_score보다 작다면, best_score와 best_parameter를 업데이트합니다.

```
{'metric': 'euclidean', 'n_neighbors': 3}    801.970237050044
```

모델 선택과 하이퍼 파라미터 최적화 문제로 확장

모델 선택과 하이퍼 파라미터 최적화 문제로 확장 예시: 탐색 공간 정의

```
1  from sklearn.ensemble import RandomForestClassifier as RFC
2  from sklearn.neural_network import MLPClassifier as MLP
4  rf_grid = {"n_estimators":[20, 50, 100, 200],
5             "max_depth": [3, 4, 5, 6, 7]}
6
7  nn_grid = {"hidden_layer_sizes": [(10, 10), (20, 20), (30, 30), (20, 20, 20, 20)],
8             "max_iter": [1000, 10000]}
9  model_parameter_dict = {RFC:ParameterGrid(rf_grid), MLP:ParameterGrid(nn_grid)}
```

- **라인 9:** RandomForestClassifier 클래스를 키로, rf_grid를 값으로 하는 사전과 MLPClassifier 클래스를 키로, nn_grid를 값으로 하는 사전을 정의합니다. 이때, rf_grid와 nn_grid는 ParameterGrid로 변환합니다.

모델 선택과 하이퍼 파라미터 최적화 문제로 확장 (계속)

모델 선택과 하이퍼 파라미터 최적화 문제로 확장 예시

```
1  kf = KFold(n_splits = 5)
2  best_score = -1
3  for model_class in model_parameter_dict.keys():
4      parameter_grid = model_parameter_dict[model_class]
5      for param in parameter_grid:
6          total_score = 0
7          for train_index, test_index in kf.split(X1):
8              X1_train = X1.loc[train_index]
9              X1_test = X1.loc[test_index]
10             y1_train = y1.loc[train_index]
11             y1_test = y1.loc[test_index]
12             model = model_class(**param).fit(X1_train, y1_train)
13             y1_pred = model.predict(X1_test)
14             score = accuracy_score(y1_test, y1_pred)
15             total_score += score / 5
16         if total_score > best_score:
17             best_score = total_score
18             best_parameter = param
19             best_model = model_class
```

- **라인 3~4:** model_parameter_dict의 키를 model_class로 순회하면서 parameter_grid를 정의합니다. model_class가 RFC라면 parameter_grid는 rf_grid가 되며, MLP라면 nn_grid가 됩니다.
- **라인 5~19:** 하이퍼파라미터를 튜닝합니다. 모델 클래스가 주어진 상태이므로 이 라인의 코드는 이전에 사용했던 코드와 같습니다. 단, 모델도 선택해야 하므로 라인 19에서 best_model에 model_class를 저장합니다.

모델 선택과 하이퍼 파라미터 최적화 문제로 확장 (계속)

모델 선택과 하이퍼 파라미터 최적화 문제로 확장 예시

```
1  print(best_model, best_parameter, best_score)
```

```
<class 'sklearn.neural_network._multilayer_perceptron.MLPClassifier'>  {'hidden_layer_sizes': (30, 30), 'max_iter': 1000}  0.9731316725978649
```