모델 선택

최선의 학습 알고리즘을 선택하는 것
최선의 하이퍼파라미터를 선택하는 것
효율적으로 최선의 모델을 선택하는 기법

모델 선택

➢ 완전 탐색을 사용해 최선의 모델 선택

- 하이퍼파라미터 범위를 검사하여 최선의 모델을 선택하려면 사이킷런의 GridSearchCV를 사용합니다.
- 사용자는 하나 이상의 하이퍼파라미터에 대해 가능성이 있는 값을 정의합니다.
- GridSearchCV는 모든 값의 조합에 대해 모델을 훈련하고 최고 성능 점수를 내는 모델이 최선의 모델로 선택됩니다.
- GridSearchCV는 교차검증을 사용하여 모델을 선택하는 브루트포스(brute-force)한 방법입니다.

```
#실습 : 로지스틱 회귀에서 C와 규제 페널티 값의 각 조합에 대해 모델을 훈련하고 k-폴드 교차검증으로 평가합니다.
#C의 값이 10개이고 규제 페널티는 두 개, 폴드 수는 5입니다. 총 10X2X5=100개의 모델 후보 중에서 가장 좋은 것을 선택합니다.
import numpy as np
from sklearn import linear_model, datasets
from sklearn.model_selection import GridSearchCV

iris = datasets.load_iris()                                  # 데이터 로드
features = iris.data
target = iris.target

logistic = linear_model.LogisticRegression()                 # 로지스틱 회귀 모델 객체 생성
penalty = ['l1', 'l2']                                       # 페널티(penalty) 하이퍼파라미터 값의 후보를 만듭니다.
C = np.logspace(0, 4, 10)                                    # 규제 하이퍼파라미터 값의 후보 범위를 만듭니다.
hyperparameters = dict(C=C, penalty=penalty)                 # 하이퍼파라미터 후보 딕셔너리를 만듭니다.
```

모델 선택

➢ **완전 탐색을 사용해 최선의 모델 선택**

- verbose 매개변수는 탐색 시간이 긴 경우에 잘 진행되는지 확인할 수 있는 옵션입니다.
- verbose 매개변수는 탐색 과정에서 출력되는 메시지의 양을 결정합니다.
- 0은 아무것도 출력하지 않고 1에서 3까지 갈수록 자세한 메시지를 출력합니다.

```
gridsearch = GridSearchCV(logistic, hyperparameters, cv=5, verbose=0)  # 그리드 서치 객체 생성
best_model = gridsearch.fit(features, target)                          # 그리드 서치 수행 .

np.logspace(0, 4, 10)

# 최선의 하이퍼파라미터를 확인합니다.
print('가장 좋은 페널티:', best_model.best_estimator_.get_params()['penalty'])
print('가장 좋은 C 값:', best_model.best_estimator_.get_params()['C'])

best_model.predict(features)   #타깃 벡터 예측
```

모델 선택

➢ 랜덤 서치를 사용해 최선의 모델 선택

- 사이킷런의 RandomizedSearchCV는 완전 탐색보다 최선의 모델을 선택하는데 계산 비용이 적게 듭니다.
- RandomizedSearchCV는 사용자가 제공한 분포(예: 정규분포나 균등 분포)에서 랜덤한 하이퍼파라미터 조합을 지정된 횟수만큼 추출하여 조사하는 것입니다
- RandomizedSearchCV에 분포를 지정하면 이 분포에서 중복을 허용하지 않도록 하이퍼파라미터 값을 랜덤하게 샘플링합니다.

```
from scipy.stats import uniform
from sklearn import linear_model, datasets
from sklearn.model_selection import RandomizedSearchCV

iris = datasets.load_iris()  # 데이터 로드
features = iris.data
target = iris.target

logistic = linear_model.LogisticRegression()   # 로지스틱 회귀 모델 생성
penalty = ['l1', 'l2']    # 페널티 하이퍼파라미터 후보를 만듭니다.
C = uniform(loc=0, scale=4)    # 규제 하이퍼파라미터 값의 후보를 위한 분포를 만듭니다.
hyperparameters = dict(C=C, penalty=penalty)    # 하이퍼파라미터 옵션을 만듭니다.

randomizedsearch = RandomizedSearchCV(          # 랜덤 서치 객체 생성
    logistic, hyperparameters, random_state=1, n_iter=100, cv=5, verbose=0,      n_jobs=-1)
best_model = randomizedsearch.fit(features, target)  # 랜덤 서치  수행
```

모델 선택

➢ **랜덤 서치를 사용해 최선의 모델 선택**

- n_iter 매개변수는 샘플링한 하이퍼파라미터 조합의 횟수 (훈련할 후보 모델의 개수)를 지정합니다.

```
# 0~4 사이의 균등 분포를 정의하고 10개의 값을 샘플링합니다.
uniform(loc=0, scale=4).rvs(10)

# 최선의 하이퍼파라미터를 확인
print('가장 좋은 페널티:', best_model.best_estimator_.get_params()['penalty'])
print('가장 좋은 C 값:', best_model.best_estimator_.get_params()['C'])

# 타깃 벡터 예측
best_model.predict(features)
```

모델 선택

➢ **여러 학습 알고리즘에서 최선의 모델 선택**

- 다양한 학습 알고리즘과 각각의 하이퍼파라미터를 탐색하여 최선의 모델을 선택하려면 후보 학습 알고리즘과 이에 해당하는 하이퍼파라미터의 딕셔너리를 만듭니다.

```
import numpy as np
from sklearn import datasets
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline

np.random.seed(0)                          # 랜덤 시드  설정
iris = datasets.load_iris()                # 데이터 로드
features = iris.data
target = iris.target
pipe = Pipeline([("classifier", RandomForestClassifier())])      # 파이프라인을 만듭니다.
# 후보 학습 알고리즘과 하이퍼파라미터로 딕셔너리를 만듭니다.
search_space = [ { "classifier": [LogisticRegression()],
                   "classifier__penalty": ['l1', 'l2'],
                   "classifier__C": np.logspace(0, 4, 10)},
                 { "classifier": [RandomForestClassifier()],
                   "classifier__n_estimators": [10, 100, 1000],
                   "classifier__max_features": [1, 2, 3]}]
```

모델 선택

➢ **여러 학습 알고리즘에서 최선의 모델 선택**

- 다양한 학습 알고리즘과 각각의 하이퍼파라미터를 탐색하여 최선의 모델을 선택하려면 후보 학습 알고리즘과 이에 해당하는 하이퍼파라미터의 딕셔너리를 만듭니다.

```
gridsearch = GridSearchCV(pipe, search_space, cv=5, verbose=0)        # 그리드 서치 객체 생성
best_model = gridsearch.fit(features, target)    # 그리드 서치를 수행

best_model.best_estimator_.get_params()["classifier"]          # 최선의 모델을 확인
best_model.predict(features)      # 타깃 벡터를 예측
```

모델 선택

➢ 전처리와 함께 최선의 모델 선택

- 모델 선택 과정에 전처리를 포함하려면 전처리 단계와 필요한 매개변수를 포함한 파이프라인을 만듭니다.
- FeatureUnion을 사용하면 여러 전처리 단계를 적절하게 연결할 수 있습니다.

```
import numpy as np
from sklearn import datasets
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

np.random.seed(0)   # 랜덤 시드 설정
iris = datasets.load_iris()   # 데이터 로드
features = iris.data
target = iris.target

# StandardScaler와 PCA를 포함한 전처리 객체를 만듭니다.
preprocess = FeatureUnion([("std", StandardScaler()), ("pca", PCA())])
# 파이프라인을 만듭니다.
pipe = Pipeline([("preprocess", preprocess),  ("classifier", LogisticRegression())])
```

모델 선택

➢ 전처리와 함께 최선의 모델 선택

- GridSearchCV는 교차검증을 사용하여 가장 높은 성능을 내는 모델을 고릅니다.
- 교차 검증에서 제외된 폴드는 본 적 없는 테스트 세트와 같은 역할을 하므로 어떤 전처리 단계에도 포함되어서는 안되므로 먼저 데이터를 전처리하고 GridSearchCV를 실행할 수 없습니다.
- 대신 전처리 단계를 GridSearchCV가 수행하는 일련의 작업 중 하나로 포함시켜야 합니다.

```
# 후보 값을 정의합니다.
#최선의 모델을 만드는 주성분이 하나인지 두개, 세 개인지를 탐색하도록 지시합니다.
search_space = [{"preprocess__pca__n_components": [1, 2, 3],
                 "classifier__penalty": ["l1", "l2"],
                 "classifier__C": np.logspace(0, 4, 10)}]

clf = GridSearchCV(pipe, search_space, cv=5, verbose=0, n_jobs=-1)   # 그리드 서치 객체 생성
best_model = clf.fit(features, target)  # 그리드 서치 수행
best_model.best_estimator_.get_params()['preprocess__pca__n_components']   # 최선의 주성분 개수를 확인
clf.best_score_       #GridSearchCV가 수행한 교차검증에서 최상의 점수가 저장되는 속성
clf.best_estimator_.named_steps["preprocess"].transform(features[0:1])
#memory 매개변수에 전처리 데이터를 임시 저장할 디렉토리 이름을 전달하면 하이퍼파라미터 탐색 과정에서
중복으로 전처리 과정을 수행하지 않습니다
pipe = Pipeline([("std", StandardScaler()),
                 ("pca", PCA()),
                 ("classifier", LogisticRegression())],
                 memory='cache')
```

모델 선택

➢ 전처리와 함께 최선의 모델 선택

- FeatureUnion 클래스는 전처리 단계를 병렬로 연결합니다.

```
# 후보 값을 정의합니다.
search_space = [{"pca__n_components": [1, 2, 3],
                 "classifier__penalty": ["l1", "l2"],
                 "classifier__C": np.logspace(0, 4, 10)}]

clf = GridSearchCV(pipe, search_space, cv=5, verbose=0, n_jobs=-1)   # 그리드 서치 객체생성

best_model = clf.fit(features, target)    # 그리드 서치 수행

clf.best_score_

clf.best_estimator_.get_params()['pca__n_components']  # 최선의 주성분 개수를 확인

clf.best_estimator_.named_steps["pca"].transform(features[0:1])
```

모델 선택

➢ 병렬화로 모델 선택 속도 향상

- 모델 선택의 처리 속도를 높이려면 병렬화로 수행합니다.
- 사이킷런은 컴퓨터에 있는 코어 개수만큼 동시에 모델을 훈련할 수 있습니다.
- n_jobs 매개변수에 병렬로 훈련할 모델의 개수를 정의합니다.

```
import numpy as np
from sklearn import linear_model, datasets
from sklearn.model_selection import GridSearchCV

iris = datasets.load_iris()  # 데이터를 로드
features = iris.data
target = iris.target

logistic = linear_model.LogisticRegression()  # 로지스틱 회귀 모델 객체 생성
penalty = ["l1", "l2"]    # 규제 페널티의 후보를 만듭니다.
C = np.logspace(0, 4, 1000)      # C 값의 후보 범위를 만듭니다.
hyperparameters = dict(C=C, penalty=penalty)    # 하이퍼파라미터 옵션을 만듭니다.

gridsearch = GridSearchCV(logistic, hyperparameters, cv=5, n_jobs=-1, verbose=1)
best_model = gridsearch.fit(features, target)
# 하나의 코어만 사용하는 그리드 서치 객체를 만듭니다.
clf = GridSearchCV(logistic, hyperparameters, cv=5, n_jobs=1, verbose=1)
best_model = clf.fit(features, target)    # 그리드 서치를 수행
```

모델 선택

- **알고리즘에 특화된 기법을 사용하여 모델 선택 수행 속도 향상**
  -

```
from sklearn import linear_model, datasets

iris = datasets.load_iris()   # 데이터 로드
features = iris.data
target = iris.target

logit = linear_model.LogisticRegressionCV(Cs=100)      # 교차검증 로지스틱 회귀 모델 생성
logit.fit(features, target)   # 모델  훈련
```

모델 선택

➢ 모델 선택 후 성능 평가상

- k-폴드 교차검증은 데이터 중 k-1개의 폴드에서 모델을 훈련하고 이 모델을 사용해 남은 폴드에서 예측을 만듭니다.
- 그 다음 모델의 예측과 정답을 비교하여 모델이 얼마나 잘 예측하는지 평가합니다.
- GridSearchCV와 RandomizedSearch에서는 최선의 하이퍼파라미터 값을 찾기 위해 데이터를 사용했기 때문에 동일한 데이터로 모델의 성능을 평가할 수 없습니다.
- 해결 방법은 모델 탐색을 위해 사용한 교차검증을 다른 교차검증으로 감싸는 것입니다.
- 중첩 교차검증에서 안쪽의 교차검증이 최선의 모델을 찾고 바깥쪽의 교차검증이 편향되지 않은 모델의 성능을 평가합니다.

```
import numpy as np
from sklearn import linear_model, datasets
from sklearn.model_selection import GridSearchCV, cross_val_score

iris = datasets.load_iris() # 데이터  로드
features = iris.data
target = iris.target

logistic = linear_model.LogisticRegression(solver='liblinear', multi_class='auto')     # # 로지스틱 회귀 모델
C = np.logspace(0, 4, 20) # 하이퍼 파라미터 범위  설정
hyperparameters = dict(C=C) # 하이퍼파라미터 옵션 설정
hyperparameters = dict(C=C)                # 하이퍼파라미터 옵션을 만듭니다
```

모델 선택

➢ 모델 선택 후 성능 평가상

- 중첩 교차검증에서 바깥쪽 교차검증이 데이터를 두 부분(훈련 세트, 테스트 세트)으로 나눕니다.
- 안쪽 그리드 서치는 이 훈련 세트를 다시 훈련 세트와 검증 세트로 나누어 교차검증을 수행합니다.
- 바깥쪽 교차검증을 반복하여 편향되지 않은 평균 성능 점수를 얻을 수 있습니다.
- #경고를 출력하지 않기 위해 solver='liblinear', multi_class='auto'로 지정합니다.
- GridSearchCV의 iid 매개변수가 True이면 독립 동일 분포라고 가정하고 테스트 세트의 샘플 수로 폴드의 점수를 가중 평균합니다.
- iid 매개변수가 False 이면 단순한 폴드 점수의 평균입니다. (기본 교차검증과 동작 방식이 같습니다.)

```
gridsearch = GridSearchCV(logistic, hyperparameters, cv=5, n_jobs=-1, verbose=0, iid=False)
# 중첩 교차검증을 수행하고 평균 점수를 출력
cross_val_score(gridsearch, features, target, cv=3).mean()

gridsearch = GridSearchCV(logistic, hyperparameters, cv=5, verbose=1, iid=False)
best_model = gridsearch.fit(features, target)
# 출력 결과 내용:  안쪽 교차검증이 20개의 후보 모델을 다섯 번씩 총 100개의 모델을 훈련
scores = cross_val_score(gridsearch, features, target, cv=3)
# 출력 결과 내용:  안쪽 교차검증이 20개의 후보 모델을 다섯 번씩 총 100개의 모델을 훈련
# 이 모델이 바깥쪽 3-폴드 교차검증을 사용해 평가
# 총 300개의 모델이 훈련
```