하이퍼파라미터 튜닝

- 머신러닝 모델의 학습 과정에 사용자가 직접 설정해야 하는 값으로, 모델의 성능에 큰 영향을 미칩니다. 하이퍼파라미터 튜닝은 모델이 최적의 성능을 발휘할 수 있도록 다양한 하이퍼파라미터 조합을 탐색하고 최적의 값을 찾는 과정을 의미합니다. 이 과정은 데이터에 가장 적합한 모델을 구축하기 위한 핵심 단계 중 하나입니다.

적용 가능한 상황

- 모델의 초기 학습 결과가 만족스럽지 않을 때 성능을 개선하고자 할 경우
- 여러 모델 후보군 중에서 특정 모델을 선택한 후, 해당 모델의 성능을 극한으로 끌어올리고자 할 경우
- 데이터의 특성이 변경되어 기존에 사용하던 하이퍼파라미터가 더 이상 최적이 아닐 것으로 예상될 경우
- 안정적이고 일반화 성능이 높은 모델을 구축하고자 할 경우 (교차 검증을 통해 검증)

1. GridSearchCV

- 사용자가 지정한 하이퍼파라미터 값들의 모든 조합에 대해 교차 검증을 수행하여 가장 좋은 성능을 내는 조합을 찾는 방법
- 격자(Grid)처럼 촘촘하게 모든 경우의 수를 탐색하므로, 탐색 범위 내에서는 최적의 조합을 반드시 탐색 가능

주의사항

- 탐색할 하이퍼파라미터의 종류와 범위가 넓어질수록 경우의 수가 기하급수적으로 증가하여 매우 오랜 시간이 소요될 수 있습니다.
- 제한된 자원으로 인해 탐색 범위를 좁게 설정할 경우, 전역 최적해(Global Optimum)가 아닌 지역 최적해(Local Optimum)에 머무를 수 있습니다.

하이퍼파라미터 설명

- `estimator`: 탐색을 적용할 머신러닝 모델 객체입니다.
- `param_grid`: 딕셔너리 또는 리스트 형태로, 탐색할 하이퍼파라미터의 이름(문자열)을 키로, 탐색할 값들의 리스트를 값으로 가집니다.
- `scoring`: 모델 성능 평가에 사용할 지표입니다. (예: 'accuracy', 'f1', 'roc_auc', 'neg_mean_squared_error')
- `cv`: 교차 검증을 위해 데이터를 나눌 폴드(fold)의 개수입니다.
- `n_jobs`: 동시에 실행할 작업의 수입니다. -1로 설정하면 모든 CPU 코어를 사용합니다.
- `verbose`: 탐색 과정의 상세 출력 여부를 결정합니다. 숫자가 클수록 더 자세한 정보를 출력합니다.
- `refit`: `True`로 설정 시(기본값), 전체 학습 데이터셋을 사용하여 최적의 하이퍼파라미터로 모델을 다시 학습시킵니다.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# 1. 데이터 준비
```

```
# 예시 데이터 생성 (실제 사용 시에는 데이터를 불러와야 함)
from sklearn.datasets import make_classification
X, y = make_classification(n_samples=1000, n_features=20, n_informative=10,
n_redundant=5, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# 2. 모델 및 하이퍼파라미터 그리드 설정
model = RandomForestClassifier(random_state=42)
param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [10, 20, None],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2]
}

# 3. GridSearchCV 객체 생성 및 학습
# cv=5는 5-fold cross-validation을 의미
# n_jobs=-1은 사용 가능한 모든 CPU 코어를 사용하라는 의미
grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=5,
n_jobs=-1, verbose=2, scoring='accuracy')
grid_search.fit(X_train, y_train)

# 4. 결과 확인
print(f"최적의 하이퍼파라미터: {grid_search.best_params_}")
# {'max_depth': 20, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators':
200}
print(f"최고 교차 검증 점수: {grid_search.best_score_:.4f}") # 0.9387

# 5. 최적 모델로 예측 및 평가
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)
print(f"테스트 세트 정확도: {accuracy_score(y_test, y_pred):.4f}") # 0.9250

# GridSearchCV 결과 전체 확인
cv_results_df = pd.DataFrame(grid_search.cv_results_)
print(cv_results_df[['param_n_estimators', 'param_max_depth', 'mean_test_score',
'rank_test_score']].sort_values('rank_test_score').head())
'''
   param_n_estimators param_max_depth  mean_test_score  rank_test_score
17                200            None          0.93875                1
9                 200              20          0.93875                1
21                200            None          0.93750                3
13                200              20          0.93750                3
15                200              20          0.93250                5
'''
```

결과 해석 방법

- `grid_search.best_params_`: 가장 높은 `scoring` 점수를 기록한 하이퍼파라미터 조합을 딕셔너리 형태로 반환합니다.
- `grid_search.best_score_`: 최적의 하이퍼파라미터 조합으로 얻은 교차 검증 점수의 평균값입니다.

- `grid_search.best_estimator_`: `refit=True`일 때, 전체 학습 데이터로 재학습된 최적의 모델입니다.
- `grid_search.cv_results_`: 교차 검증의 모든 중간 결과를 딕셔너리 형태로 저장하고 있으며, `pandas.DataFrame`으로 변환하여 분석하면 각 조합의 성능을 상세히 비교할 수 있습니다.

---

2. RandomizedSearchCV

- 지정된 횟수(`n_iter`)만큼 하이퍼파라미터 값의 조합을 랜덤하게 추출하여 탐색하는 방법
- 계산 비용을 절약하면서도 준수한 최적해를 효율적으로 탐색 가능

주의사항

- `n_iter` 횟수만큼만 탐색하므로, 최적의 조합을 놓칠 가능성이 있습니다.
- `n_iter` 값이 클수록 최적해를 찾을 확률이 높아지지만, 그만큼 탐색 시간이 길어집니다.
- 하이퍼파라미터의 값 범위를 연속적인 분포(예: `scipy.stats.uniform`)로 지정할 수 있어, `GridSearchCV`보다 더 넓고 세밀한 탐색이 가능합니다.

하이퍼파라미터 설명

- `param_distributions`: `param_grid` 대신 사용되며, 리스트(이산형) 또는 분포 객체(연속형, 예: `scipy.stats.uniform`, `scipy.stats.randint`)를 값으로 가집니다.
- `n_iter`: 탐색을 시도할 횟수입니다. 이 값이 클수록 더 좋은 조합을 찾을 가능성이 높아집니다.
- `random_state`: 랜덤 샘플링의 재현성을 위해 설정하는 시드(seed) 값입니다.
- 나머지 인자(`estimator`, `scoring`, `cv`, `n_jobs`, `verbose`, `refit`)는 `GridSearchCV`와 동일합니다.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split, RandomizedSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from scipy.stats import randint

# 1. 데이터 준비 (GridSearchCV 예시와 동일)
from sklearn.datasets import make_classification
X, y = make_classification(n_samples=1000, n_features=20, n_informative=10,
n_redundant=5, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# 2. 모델 및 하이퍼파라미터 분포 설정
model = RandomForestClassifier(random_state=42)
param_dist = {
    'n_estimators': randint(100, 500), # 100에서 500 사이의 정수
    'max_depth': [10, 20, 30, None],
    'min_samples_split': randint(2, 11), # 2에서 10 사이의 정수
    'min_samples_leaf': randint(1, 5) # 1에서 4 사이의 정수
}

# 3. RandomizedSearchCV 객체 생성 및 학습
# n_iter=100은 100개의 조합을 랜덤하게 테스트하라는 의미
random_search = RandomizedSearchCV(estimator=model,
```

```
param_distributions=param_dist, n_iter=100,
                                    cv=5, n_jobs=-1, verbose=2, scoring='accuracy',
random_state=42)
random_search.fit(X_train, y_train)

# 4. 결과 확인
print(f"최적의 하이퍼파라미터: {random_search.best_params_}")
print(f"최고 교차 검증 점수: {random_search.best_score_:.4f}")

# 5. 최적 모델로 예측 및 평가
best_model = random_search.best_estimator_
y_pred = best_model.predict(X_test)
print(f"테스트 세트 정확도: {accuracy_score(y_test, y_pred):.4f}")
```

결과 해석 방법

- `GridSearchCV`와 동일하게 `best_params_`, `best_score_`, `best_estimator_`, `cv_results_` 속성을 통해 결과를 확인 가능

---

3. 베이지안 최적화 (Bayesian Optimization)

- 사전 지식(prior)을 바탕으로 목적 함수(여기서는 모델의 성능)의 형태를 추정하고, 이를 기반으로 다음 탐색할 하이퍼파라미터 지점을 '지능적으로' 결정하는 방법
- 이전 탐색 결과를 활용하여 불필요한 탐색은 줄이고, 성능이 좋을 것으로 기대되는 영역을 집중적으로 탐색하여 `GridSearchCV`나 `RandomizedSearchCV`보다 더 적은 횟수로 최적해를 찾는 것이 목표

주의사항

- `scikit-learn`에 내장되어 있지 않아 `scikit-optimize`, `Hyperopt`, `Optuna` 등 별도의 라이브러리 설치가 필요합니다.
- 내부적으로 확률 모델(주로 가우시안 프로세스)을 사용하므로, 탐색 과정이 앞선 두 방법보다 복잡하게 느껴질 수 있습니다.
- 탐색 횟수가 매우 적을 경우, 오히려 랜덤 탐색보다 성능이 안 좋을 수도 있습니다.

하이퍼파라미터 설명 (`scikit-optimize`의 `BayesSearchCV`)

- `search_spaces`: `param_grid`나 `param_distributions` 대신 사용되며, `skopt.space`의 `Integer`, `Real`, `Categorical` 등을 사용하여 각 하이퍼파라미터의 탐색 공간을 정의합니다.
- `n_iter`: 탐색 횟수입니다.
- `BayesSearchCV`는 `scikit-learn`의 `GridSearchCV` 등과 API 호환성이 높아 사용법이 매우 유사합니다.

```
# scikit-optimize 라이브러리 설치 필요
# pip install scikit-optimize

import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
```

```
from sklearn.metrics import accuracy_score
from skopt import BayesSearchCV
from skopt.space import Real, Categorical, Integer

# 1. 데이터 준비 (GridSearchCV 예시와 동일)
from sklearn.datasets import make_classification
X, y = make_classification(n_samples=1000, n_features=20, n_informative=10,
n_redundant=5, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# 2. 모델 및 하이퍼파라미터 탐색 공간(Search Space) 설정
model = RandomForestClassifier(random_state=42)
search_spaces = {
    'n_estimators': Integer(100, 500),
    'max_depth': Integer(10, 50),
    'min_samples_split': Integer(2, 10),
    'min_samples_leaf': Integer(1, 5),
    'max_features': Real(0.1, 1.0, prior='uniform') # 0.1에서 1.0 사이의 실수
}

# 3. BayesSearchCV 객체 생성 및 학습
# n_iter=50은 50개의 조합을 테스트하라는 의미
bayes_search = BayesSearchCV(estimator=model, search_spaces=search_spaces,
n_iter=50,
                             cv=5, n_jobs=-1, verbose=2, scoring='accuracy',
random_state=42)
bayes_search.fit(X_train, y_train)

# 4. 결과 확인
print(f"최적의 하이퍼파라미터: {bayes_search.best_params_}")
print(f"최고 교차 검증 점수: {bayes_search.best_score_:.4f}")

# 5. 최적 모델로 예측 및 평가
best_model = bayes_search.best_estimator_
y_pred = best_model.predict(X_test)
print(f"테스트 세트 정확도: {accuracy_score(y_test, y_pred):.4f}")
```

결과 해석 방법

- `GridSearchCV`와 동일하게 `best_params_`, `best_score_`, `best_estimator_`, `cv_results_` 속성을 통해 결과를 확인할 수 있습니다. `best_params_`는 `OrderedDict` 형태로 반환됩니다.

장단점 및 대안

방법	장점	단점
GridSearchCV	- 지정된 범위 내의 최적해를 반드시 찾음 - 이해하고 사용하기 쉬움	- 탐색 공간이 커지면 시간이 매우 오래 걸림 - 연속형 하이퍼파라미터 탐색에 비효율적

방법	장점	단점
RandomizedSearchCV	- GridSearchCV보다 탐색 속도가 빠름 - 예상치 못한 좋은 조합을 발견할 수 있음 - 넓은 탐색 공간에서 효율적	- 최적해를 찾는다는 보장이 없음 - 랜덤성에 의존하므로 결과의 편차가 있을 수 있음
베이지안 최적화	- 이전 탐색 결과를 활용하여 지능적으로 탐색 - 적은 시도 횟수로 높은 성능의 조합을 찾을 가능성이 높음	- 구현이 상대적으로 복잡하고 별도 라이브러리 필요 - 내부 메커니즘이 직관적이지 않음

대안

- **Hyperopt, Optuna**: 베이지안 최적화를 포함한 고급 최적화 기법을 제공하는 파이썬 라이브러리들입니다. `BayesSearchCV`보다 더 유연하고 강력한 기능을 제공하며, 특히 딥러닝 모델 튜닝에 많이 사용됩니다.
- **수동 튜닝 (Manual Tuning)**: 분석가의 직관과 경험을 바탕으로 직접 하이퍼파라미터를 조정하는 방법입니다. 시간과 노력이 많이 들지만, 데이터와 모델에 대한 깊은 이해가 있다면 효과적일 수 있습니다.