**RandomForest Hyperparameter tuning guide**

하이퍼파라미터 튜닝은 이론보다 실험 결과에 더 의존하므로 최적 설정을 결정하는 가장 좋은 방법은 여러 조합으로 각 모형의 성능을 평가하는 것이다. 그러나 각 모델을 교육 세트에 대해서만 평가하면 기계 학습에서 가장 근본적인 문제 중 하나인 과적합 문제가 발생할 수 있다.

교육 데이터에 맞게 모델을 최적화하면 모델은 교육 세트에서 매우 좋은 점수를 얻을 수 있지만 테스트 세트와 같은 새로운 데이터로 일반화할 수는 없다.

이것이 교차검증을 해야하는 이유이다.

교차검증은 가장 일반적인 Kfold를 사용할 수 있다. 기계 학습 문제에 접근할 때, 우린느 반드시 데이터를 훈련 세트와 테스트 세트로 나눈다. kfold에서 우리는 우리의 훈련세트를 폴드라고 불리는 k개의 하위 집합으로 나눈다. 그 다음 모델을 k번 반복적으로 학습 후 평가한다.

하이퍼파라미터 튜닝을 위해, 우리는 매번 다른 모델 설정을 사용하여 전체 폴드 프로세스를 여러 번 반복한다. 그 다음 모든 모델을 비교하고, 가장 적합한 모델을 선택하고, 전체 교육세트에서 교육한 다음, 테스트 세트에서 평가한다. 서로 다른 하이퍼 파라미터 세트를 평가하려고 할때할 때 교육 데이터를 k개의 폴드로 나누고 k번 교육해야 한다. 예를 들어 10개의 하이퍼 파라미터 조합이 있고 k가 5인 경우 50번 학습하고 평가해야 한다. 이 과정은 사이킷런의 GridSearchCV와 RandomizedSearchCV로 작업할 수 있다.

일반적으로 최적의 하이퍼 파라미터에 대한 막연한 아이디어만 가지고 있으므로 검색 범위를 좁히는 최선의 방법은 각 하이퍼 파라미터에 대한 광범위한 값을 평가하는 것이다. 사이킷런의 RandomizedSearchCV는 하이퍼 파라미터 범위의 그리드를 정의하고 그리드에서 무작위로 샘플을 추출하여 각 값의 조합으로 kfold를 수행한다.

랜덤포레스트에서 가장 중요한 파라미터는 n\_estimators와 max\_features이다.

n\_estimators: 평가 나무의 개수

max\_features: 부트스트랩 시 추출된 데이터셋의 최대 특징 개수

max\_depth: 나무의 최대 깊이

min\_sample\_split: 노드가 분할되기 전 노드에 존재해야 하는 최소 포인트 개수

min\_samples\_leaf: 리프 노드에 포함되어야 하는 최소 포인트 개수

bootstrap: 부트스트랩 방식

criterion: 분할 방식 (범주형: gini, entropy, 연속형: f검정, 분산감소량)

**예시**

**{'bootstrap': [True, False],  
'max\_depth': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, None],  
'max\_features': ['auto', 'sqrt'],  
'min\_samples\_leaf': [1, 2, 4],  
'min\_samples\_split': [2, 5, 10],  
'n\_estimators': [200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000]}**

랜덤서치에서 가장 중요한 파라미터는 시도할 여러 조합의 수를 제어하는 n\_iter와 교차검증 방식인 cv이다. 반복 횟수가 많을수록 검색 공간이 넓어지고 cv가 많을수록 과접합 가능성은 감소하지만 실행 시간이 증가한다.

# Use the random grid to search for best hyperparameters  
# First create the base model to tune  
rf = RandomForestRegressor()  
# Random search of parameters, using 3 fold cross validation,   
# search across 100 different combinations, and use all available cores  
rf\_random = RandomizedSearchCV(estimator = rf, param\_distributions = random\_grid, n\_iter = 100, cv = 3, verbose=2, random\_state=42, n\_jobs = -1)# Fit the random search model  
rf\_random.fit(train\_features, train\_labels)

1차 서치 결과

rf\_random.best\_params\_**{'bootstrap': True,  
 'max\_depth': 70,  
 'max\_features': 'auto',  
 'min\_samples\_leaf': 4,  
 'min\_samples\_split': 10,  
 'n\_estimators': 400}**

베이스 모델과 성능 비교

def evaluate(model, test\_features, test\_labels):  
 predictions = model.predict(test\_features)  
 errors = abs(predictions - test\_labels)  
 mape = 100 \* np.mean(errors / test\_labels)  
 accuracy = 100 - mape  
 print('Model Performance')  
 print('Average Error: {:0.4f} degrees.'.format(np.mean(errors)))  
 print('Accuracy = {:0.2f}%.'.format(accuracy))  
   
 return accuracybase\_model = RandomForestRegressor(n\_estimators = 10, random\_state = 42)  
base\_model.fit(train\_features, train\_labels)  
base\_accuracy = evaluate(base\_model, test\_features, test\_labels)**Model Performance  
Average Error: 3.9199 degrees.  
Accuracy = 93.36%.**best\_random = rf\_random.best\_estimator\_  
random\_accuracy = evaluate(best\_random, test\_features, test\_labels)**Model Performance  
Average Error: 3.7152 degrees.  
Accuracy = 93.73%.**print('Improvement of {:0.2f}%.'.format( 100 \* (random\_accuracy - base\_accuracy) / base\_accuracy))**Improvement of 0.40%.**

위 예제는 랜덤서치 후 베이스 모델보다 0.4% 성능이 개선됨을 보여준다.

랜덤서치로 넒은 범위를 빠르게 찾았다면, 보다 좁아진 영역에서 그리드 서치를 진행한다.

# Fit the grid search to the data  
grid\_search.fit(train\_features, train\_labels)grid\_search.best\_params\_**{'bootstrap': True,  
 'max\_depth': 80,  
 'max\_features': 3,  
 'min\_samples\_leaf': 5,  
 'min\_samples\_split': 12,  
 'n\_estimators': 100}**best\_grid = grid\_search.best\_estimator\_  
grid\_accuracy = evaluate(best\_grid, test\_features, test\_labels)**Model Performance  
Average Error: 3.6561 degrees.  
Accuracy = 93.83%.**print('Improvement of {:0.2f}%.'.format( 100 \* (grid\_accuracy - base\_accuracy) / base\_accuracy))**Improvement of 0.50%.**