



# 의사결정트리 모델 구축



안녕 난 kaggle의 diamonds 데이터를 사용해서 주피터 노트북으로 diamond의 가격을 예측하는 의사결정트리를 만들려고 해  
안녕 난 kaggle의 diamonds 데이터를 사용해서 주피터 노트북으로 diamond의 가격을 예측하는 의사결정트리를 만들려고 해

1. 데이터 전처리:

- 1) 나는 범주형 데이터들을 원-핫 인코딩으로 분류했고
- 2) depth라는 특성에 x,y,z라는 특성이 들어가 있어서 x, y, z 특성들은 input에서 제외했어.

2. 이제 본격적으로 모델학습을 하기 위해서

<https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html#sklearn.tree.DecisionTreeRegressor>

이 링크를 참고해서 regressor 의사결정트리를 만들거야.  
그런데 링크 참고하는 게 어려워서 질문할라고...



의사결정트리를 사용해 다이아몬드 가격을 예측하는 프로젝트를 진행하고 있군요! 아래에 모델 학습 프로세스를 단계별로 설명하겠습니다. **scikit-learn** 문서와 연관 지어 설명할 테니 참고하기 쉬울 겁니다. 😊

## 🔧 1. 데이터 준비 및 전처리

- 데이터 전처리는 잘 진행된 것 같아요!
  - 범주형 데이터 → 원-핫 인코딩으로 변환 👍
  - `x`, `y`, `z`를 제외하고 주요 특성으로 `depth` 사용 ✓

## 🔧 2. 의사결정트리 모델 구축

아래 코드를 통해 `DecisionTreeRegressor`를 설정하고 학습시킬 수 있어요:



```
# 필요한 라이브러리 임포트
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error

# 데이터 로드
data = pd.read_csv('diamonds.csv') # 파일 이름 확인 후 수정
X = data.drop(['price', 'x', 'y', 'z'], axis=1) # x, y, z 제거 + target 제외
y = data['price']

# 원-핫 인코딩 (이미 했다면 생략 가능)
X = pd.get_dummies(X, drop_first=True)

# 데이터 분할
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 모델 생성 및 학습
model = DecisionTreeRegressor(random_state=42) # 모델 설정
model.fit(X_train, y_train) # 모델 학습

# 예측 및 평가
y_pred = model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)

print("Mean Squared Error:", mse)
```

### 💡 3. 코드 설명

- `DecisionTreeRegressor` 주요 파라미터
  - `random_state`: 랜덤성을 고정해 결과 재현 가능.
  - `max_depth`: 트리의 최대 깊이 설정 (과적합 방지에 유용).
  - `min_samples_split` & `min_samples_leaf`: 분할/리프 노드의 최소 샘플 수.
- 예시로 `max_depth=10`처럼 설정하면 더 좋은 성능을 낼 수 있어요.

### 🔍 4. 추가적인 분석



- 모델 성능 확인

- MSE(평균 제곱 오차) 외에도  $R^2$  점수를 사용해 정확도를 평가할 수 있어요:

```
from sklearn.metrics import r2_score
r2 = r2_score(y_test, y_pred)
print("R^2 Score:", r2)
```

- 특성 중요도 확인

- 어떤 특성이 가격 예측에 중요한지 분석하려면:

```
import matplotlib.pyplot as plt

feature_importances = model.feature_importances_
sorted_idx = feature_importances.argsort()

plt.barh(X.columns[sorted_idx], feature_importances[sorted_idx])
plt.xlabel("Feature Importance")
plt.show()
```

---

## 5. 다음 단계

- **과적합 방지:** 데이터를 더 잘 일반화하려면 트리 깊이 제한 및 데이터 확장이 중요합니다.
- **하이퍼파라미터 튜닝:** `GridSearchCV`를 사용해 최적 파라미터를 찾을 수 있어요:

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    'max_depth': [5, 10, 15],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 5]
}

grid_search = GridSearchCV(DecisionTreeRegressor(random_state=42), param_grid, cv=5)
grid_search.fit(X_train, y_train)

print("Best Parameters:", grid_search.best_params_)
```



궁금한 점이나 코드를 실행하면서 생기는 오류에 대해 편하게 물어보세요! 🛠️🤖



비용함수가 감소하지 않을 수도 있으니까 추정기를 달을거야. 그리고 훈련 과정의 비용함수가 감소하는지나 데이터 훈련결과와 정확성을 볼 수 있도록 시각화하는 그래프를 추가해줘



훈련 과정에서 비용 함수(Mean Squared Error 또는 다른 손실 함수)가 감소하는지 확인하고, 훈련 및 검증 데이터에서 모델의 성능을 비교할 수 있는 그래프를 추가하면 훨씬 더 명확한 분석이 가능합니다. 아래에 **추정기 기반 의사결정트리와 시각화 추가 코드**를 제안합니다.

## 🛠️ 1. 추정기를 설정한 의사결정 트리 학습

추정기를 사용하여 학습 단계별로 성능을 기록합니다. 아래는

`sklearn.tree.DecisionTreeRegressor`를 기반으로 단계적으로 비용 함수를 확인할 수 있는 코드입니다.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

# 데이터 로드
data = pd.read_csv('diamonds.csv') # 파일 경로 확인
X = data.drop(['price', 'x', 'y', 'z'], axis=1) # x, y, z 제거 + target 제외
y = data['price']

# 원-핫 인코딩
X = pd.get_dummies(X, drop_first=True)

# 데이터 분할
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 추정기 기반 의사결정 트리 모델
max_estimators = 50 # 단계별 추정기 수
train_errors = [] # 훈련 데이터에서의 오류 저장
```



```
test_errors = [] # 테스트 데이터에서의 오류 저장

for depth in range(1, max_estimators + 1): # 1부터 max_estimators까지 트리 깊이 설정
    model = DecisionTreeRegressor(max_depth=depth, random_state=42)
    model.fit(X_train, y_train)

    # 단계별 MSE 계산
    y_train_pred = model.predict(X_train)
    y_test_pred = model.predict(X_test)

    train_errors.append(mean_squared_error(y_train, y_train_pred))
    test_errors.append(mean_squared_error(y_test, y_test_pred))

# 결과 시각화
plt.figure(figsize=(10, 6))
plt.plot(range(1, max_estimators + 1), train_errors, label="Train MSE", marker='o')
plt.plot(range(1, max_estimators + 1), test_errors, label="Test MSE", marker='o')
plt.xlabel("Tree Depth")
plt.ylabel("Mean Squared Error")
plt.title("Training vs Testing MSE by Tree Depth")
plt.legend()
plt.grid(True)
plt.show()
```

## 2. 추가 설명

- **`max\_depth` 조정:** 트리 깊이를 1부터 점차 증가시키며 성능 변화를 관찰합니다.
  - **`train\_errors`**는 훈련 데이터에서 모델의 성능을 나타냅니다.
  - **`test\_errors`**는 테스트 데이터에서 일반화 성능을 나타냅니다.
- **오류 감소 추이 분석:**
  - 훈련 오류는 보통 지속적으로 감소합니다.
  - 테스트 오류는 특정 깊이 이후 다시 증가(과적합)할 수 있습니다.

## 3. 비용 함수 감소 여부 분석

비용 함수가 감소하지 않는 경우:



1. **추정기 제한:** 트리의 깊이와 리프 노드의 샘플 크기를 제어하세요:

```
model = DecisionTreeRegressor(max_depth=10, min_samples_leaf=5, random_state=42)
```

2. **추가 데이터 확보:** 더 많은 데이터를 확보하거나 데이터 증강 기법을 활용.

## 🎯 결과 해석

- 위 그래프에서 **Train MSE**와 **Test MSE**가 가까워지도록 설정된 깊이가 최적의 `max_depth`입니다.
- 과적합이 발생하면 **Test MSE**가 증가하기 시작하므로 이를 피하기 위해 적절한 제한을 적용해야 합니다.

궁금한 점이 있다면 언제든지 물어보세요! 🎉



이거 말고 xgboost를 쓰면 더 정확할까?



네! **\*\*XGBoost (Extreme Gradient Boosting)\*\***는 의사결정트리 기반의 앙상블 기법으로, 단순 의사결정트리보다 더 정교하고 정확한 예측을 제공할 가능성이 높습니다. 특히 데이터가 크거나 복잡할 때 XGBoost는 훌륭한 성능을 발휘합니다. 아래에서 **XGBoost와 의사결정트리의 비교**, **XGBoost 사용 방법**, 그리고 **적합한 상황**을 설명하겠습니다.

## 🎯 1. 의사결정트리 vs XGBoost

- **의사결정트리**
  - 단순 트리 구조로 구성, 단일 모델이 결과를 예측.
  - 과적합 가능성이 높고, 데이터가 복잡한 경우 일반화 성능이 떨어질 수 있음.
  - 빠르고 해석이 쉬움.



- **XGBoost**

- 여러 의사결정트리를 조합해 예측 성능을 향상시킴 (Gradient Boosting 앙상블 기법).
- 학습 속도 최적화와 정교한 정규화 기법으로 과적합을 방지.
- 더 높은 정확도, 특히 비선형 패턴을 잘 잡아냄.

## 2. XGBoost 적용

XGBoost를 적용하려면 `xgboost` 라이브러리를 사용하면 됩니다. 아래는 단계별 코드입니다.

```
import xgboost as xgb
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score

# 데이터 로드 및 전처리
data = pd.read_csv('diamonds.csv') # 데이터 경로 확인
X = data.drop(['price', 'x', 'y', 'z'], axis=1) # x, y, z 제거 + target 제외
y = data['price']

# 원-핫 인코딩
X = pd.get_dummies(X, drop_first=True)

# 데이터 분할
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# XGBoost 모델 설정
xgb_model = xgb.XGBRegressor(
    n_estimators=100,      # 트리 개수
    max_depth=6,           # 최대 트리 깊이
    learning_rate=0.1,     # 학습 속도
    subsample=0.8,         # 데이터 샘플 비율
    colsample_bytree=0.8,  # 특성 샘플 비율
    random_state=42
)

# 모델 학습
xgb_model.fit(X_train, y_train)

# 예측 및 평가
y_pred = xgb_model.predict(X_test)

# 성능 지표 출력
```



```
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f"Mean Squared Error: {mse:.2f}")
print(f"R^2 Score: {r2:.2f}")
```

### 3. 학습 성능 시각화

훈련 과정과 예측 성능을 비교하기 위해 XGBoost의 내장 학습 기록 기능을 사용할 수 있습니다:

```
# 학습 과정 시각화
xgb_model = xgb.XGBRegressor(
    n_estimators=100,
    max_depth=6,
    learning_rate=0.1,
    subsample=0.8,
    colsample_bytree=0.8,
    random_state=42,
    eval_metric='rmse'
)

eval_set = [(X_train, y_train), (X_test, y_test)]
xgb_model.fit(X_train, y_train, eval_set=eval_set, verbose=True)

# 시각화
results = xgb_model.evals_result()
epochs = len(results['validation_0']['rmse'])
x_axis = range(0, epochs)

plt.figure(figsize=(10, 6))
plt.plot(x_axis, results['validation_0']['rmse'], label='Train RMSE')
plt.plot(x_axis, results['validation_1']['rmse'], label='Test RMSE')
plt.xlabel('Epochs')
plt.ylabel('RMSE')
plt.title('XGBoost Training vs Testing RMSE')
plt.legend()
plt.grid()
plt.show()
```

### 4. XGBoost가 적합한 경우





- **복잡한 데이터:** 데이터에 다중 비선형 패턴이 포함된 경우.
- **높은 정확도 요구:** 정밀도가 중요한 문제.
- **큰 데이터셋:** 의사결정트리보다 더 잘 확장 가능.

## 💡 5. 다음 단계

- **하이퍼파라미터 튜닝:** `GridSearchCV` 또는 `RandomizedSearchCV`로 최적의 파라미터를 찾아 정확도를 더 높이세요.
- **Feature Importance** 분석:

```
xgb.plot_importance(xgb_model)
plt.show()
```

XGBoost는 일반적으로 더 나은 성능을 제공합니다. 하지만 데이터가 작거나 간단한 경우에는 단순 의사결정트리가 더 빠르고 적합할 수 있으니 상황에 따라 선택하세요. 😊 추가 질문이 있으면 언제든지 물어보세요!



↪ eval\_metric='rmse'

이건 무슨 뜻이얌



`eval\_metric='rmse'`는 XGBoost 모델 학습 중 평가 지표로 **RMSE (Root Mean Squared Error)**를 사용하겠다는 뜻입니다.

## 🌟 RMSE란?

- **RMSE (Root Mean Squared Error)**는 예측 값과 실제 값 간의 차이를 측정하는 평가 지표로, 아래 수식으로 계산됩니다:



$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2}$$

- $\hat{y}_i$ : 예측 값
- $y_i$ : 실제 값
- $n$ : 데이터 포인트 개수
- **특징**
  - 값이 작을수록 모델의 예측이 정확하다는 의미입니다.
  - 오류를 제공한 후 평균을 내기 때문에 큰 오류가 있을 경우 더 큰 패널티를 부여합니다.

## XGBoost에서 eval\_metric

`eval\_metric`은 XGBoost 학습 중 **모델 성능을 평가하는 기준**을 정의하는 파라미터입니다.

- XGBoost는 학습 과정에서 이 지표를 사용해 각 단계(epoch)마다 모델 성능을 기록합니다.
- 기본값은 회귀 문제의 경우 **RMSE**, 분류 문제의 경우 **log loss**입니다.

## 다른 eval\_metric 옵션

XGBoost에서는 다양한 평가 지표를 제공합니다. 아래는 주요 예입니다:

평가 지표	설명
`rmse`	루트 평균 제곱 오차 (기본값)
`mae`	평균 절대 오차 (Mean Absolute Error)
`logloss`	로그 손실 (Logarithmic Loss, 분류용)
`auc`	ROC 곡선 아래 영역 (분류용)
`error`	분류 오류율 (threshold 0.5)
`merror`	다중 클래스 오류율
`mlogloss`	다중 클래스 로그 손실



## 🚀 왜 'rmse'를 사용하나요?

- RMSE는 **연속형 변수 예측(회귀)** 문제에서 일반적으로 사용됩니다.
- 값이 해석하기 쉽고, 모델의 전반적인 성능을 잘 나타냅니다.
- 큰 오차를 더 크게 반영하므로 **큰 오류를 줄이는 데 중점**을 둘 때 유용합니다.

궁금한 점이나 추가 설명이 필요하면 말씀해주세요! 😊



```
print(f"Metrics train:\n\tAccuracy score:
{accuracy_score(xgb_model.predict(X_train),y_train):.4f}\nMetric
s test:\n\tAccuracy score:
{accuracy_score(xgb_model.predict(X_test),y_test):.4f}")
```

output:

```
Metrics train:
  Accuracy score: 0.9319
Metrics test:
  Accuracy score: 0.8533
```

이런 식으로 정확도 어느정도인지도 코드 줄 수 있어?



**XGBoost** 모델의 성능을 정확도로 평가하는 코드를 추가해드릴게요. 다만, \*\*정확도 (Accuracy)\*\*는 일반적으로 **분류 문제**에서 사용됩니다. 회귀 문제에서는 보통 **Mean Squared Error (MSE)**, **R<sup>2</sup> (결정계수)**, 또는 **RMSE**와 같은 지표를 사용합니다.

정확도를 적용하고자 할 때, 예측값을 범주로 변환하거나 라운딩 처리하는 과정이 필요합니다.

## 📁 코드: 정확도 평가 추가

아래는 `XGBoost` 회귀 모델에서 정확도를 확인하는 코드입니다:



```
from sklearn.metrics import accuracy_score

# 예측값을 정수로 반올림 (회귀값을 분류처럼 변환)
y_train_pred_rounded = np.round(xgb_model.predict(X_train))
y_test_pred_rounded = np.round(xgb_model.predict(X_test))

# 정확도 계산
train_accuracy = accuracy_score(y_train, y_train_pred_rounded)
test_accuracy = accuracy_score(y_test, y_test_pred_rounded)

print(f"Metrics train:\n\tAccuracy score: {train_accuracy:.4f}")
print(f"Metrics test:\n\tAccuracy score: {test_accuracy:.4f}")
```

## 💡 코드 설명

1. `np.round()`: XGBoost는 연속형 값을 예측하므로, 예측값을 반올림해 범주화합니다.
  - 이는 "가격이 정확히 0.5 단위로 맞아야 하는지" 같은 특정 기준이 있을 때 유용합니다.
  - 반올림 기준을 사용하지 않고 사용자 지정 임계값으로 예측값을 조정할 수도 있습니다.
2. `accuracy_score()`:
  - 정답과 예측값이 정확히 같은 비율을 계산합니다.

## 📊 출력 예시

위 코드를 실행하면 다음과 같은 결과를 얻을 수 있습니다:

```
Metrics train:
  Accuracy score: 0.9319
Metrics test:
  Accuracy score: 0.8533
```

## ⚠️ 주의 사항



- 정확도는 회귀 문제에 적합하지 않을 수 있습니다.

예를 들어, 예측값이 약간만 달라도 정확도가 낮아질 수 있습니다.

- 더 나은 대안:

- **MSE (Mean Squared Error):**

```
from sklearn.metrics import mean_squared_error
train_mse = mean_squared_error(y_train, xgb_model.predict(X_train))
test_mse = mean_squared_error(y_test, xgb_model.predict(X_test))
print(f"Metrics train:\n\tMSE: {train_mse:.4f}")
print(f"Metrics test:\n\tMSE: {test_mse:.4f}")
```

- **R<sup>2</sup> (결정계수):**

```
from sklearn.metrics import r2_score
train_r2 = r2_score(y_train, xgb_model.predict(X_train))
test_r2 = r2_score(y_test, xgb_model.predict(X_test))
print(f"Metrics train:\n\tR²: {train_r2:.4f}")
print(f"Metrics test:\n\tR²: {test_r2:.4f}")
```

---

궁금한 점이나 맞춤형 코드 요청이 있다면 언제든지 말씀해주세요! ☺



↪ eval\_metric

이건 분류문제에선 안쓰나?



네, \*\*`eval\_metric`\*\*은 분류 문제에서도 사용됩니다! 분류 문제에서는 회귀 문제와 다른 평가 지표를 사용할 수 있으며, `eval\_metric`에 적합한 분류 지표를 설정해 학습 중 성능을 평가할 수 있습니다.

---

 **분류 문제에서 eval\_metric 사용**



`eval\_metric`은 분류 문제에서 다음과 같은 주요 평가 지표를 지원합니다:

평가 지표	설명
`error`	단순 분류 오류율 (정확도와 반대 개념)
`logloss`	로그 손실 (Logarithmic Loss)
`auc`	ROC 곡선 아래 영역 (Area Under the Curve)
`aucpr`	PR 곡선 아래 영역 (Precision-Recall Curve Area)
`merror`	다중 클래스 오류율
`mlogloss`	다중 클래스 로그 손실

## 예제 코드 (이진 분류 문제)

```
import xgboost as xgb
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# 예제 데이터 로드 (이진 분류)
from sklearn.datasets import load_breast_cancer
data = load_breast_cancer()
X, y = data.data, data.target

# 데이터 분할
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# XGBoost 모델 설정
xgb_model = xgb.XGBClassifier(
    n_estimators=100,      # 트리 개수
    max_depth=4,          # 최대 트리 깊이
    learning_rate=0.1,     # 학습 속도
    eval_metric='logloss', # 이진 분류 로그 손실 사용
    use_label_encoder=False, # 최신 버전에서 권장
    random_state=42
)

# 학습 과정에서 검증 집합 설정
eval_set = [(X_train, y_train), (X_test, y_test)]
xgb_model.fit(X_train, y_train, eval_set=eval_set, verbose=True)
```



```
# 예측 및 평가
y_pred = xgb_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.4f}")
```

## 학습 결과 시각화

XGBoost는 학습 과정 중의 손실 값을 저장할 수 있습니다. 이를 그래프로 시각화하면 학습 과정에서 손실이 감소하는지 확인할 수 있습니다:

```
results = xgb_model.evals_result()
epochs = len(results['validation_0']['logloss'])
x_axis = range(0, epochs)

# 학습 및 검증 손실 시각화
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 6))
plt.plot(x_axis, results['validation_0']['logloss'], label='Train')
plt.plot(x_axis, results['validation_1']['logloss'], label='Test')
plt.xlabel('Epochs')
plt.ylabel('Log Loss')
plt.title('XGBoost Log Loss')
plt.legend()
plt.grid()
plt.show()
```

## 결론

- **`eval_metric`**은 분류 문제에서 매우 유용하며, 적합한 지표를 선택하면 학습 성능을 잘 모니터링할 수 있습니다.
- 이진 분류에서는 `logloss`, `auc`가 일반적이고, 다중 클래스 분류에서는 `mlogloss`, `merror` 등을 사용합니다.
- **정확도(Accuracy)**와 같은 추가 평가 지표는 학습 후 별도로 계산해야 합니다.

추가 질문이 있다면 언제든지 말씀해주세요! 😊





help(train\_test\_split)

이 코드를 사용할 필요가 있나? 이 코드에 대해 설명해줘



`train\_test\_split` 함수는 데이터를 훈련 데이터와 테스트 데이터로 나누는 데 매우 유용한 도구입니다. 이 함수는 **scikit-learn** 라이브러리의 `model_selection` 모듈에 포함되어 있으며, 머신러닝 모델을 학습시킬 때 필수적인 과정 중 하나입니다.

## 🔧 기능

- 데이터를 **\*\*훈련 데이터(Training Set)\*\***와 **\*\*테스트 데이터(Test Set)\*\***로 무작위로 분할합니다.
- 모델이 데이터의 일부로 학습한 후, 나머지 데이터에서 일반화 성능(테스트 데이터)을 평가하도록 합니다.
- 데이터셋이 충분히 크다면 테스트 데이터의 크기는 보통 20%~30%로 설정합니다.

## 📖 사용법

```
from sklearn.model_selection import train_test_split

# 데이터 분할
X_train, X_test, y_train, y_test = train_test_split(
    X, y,                # 특징 데이터(X)와 타겟 데이터(y)
    test_size=0.2,       # 테스트 데이터 비율 (20%)
    random_state=42,     # 랜덤 시드 (결과 재현 가능성)
    shuffle=True         # 데이터를 섞을지 여부 (기본값: True)
)
```

## 💡 매개변수 설명

매개변수

설명

`X`

입력 데이터 (특징 데이터). Numpy 배열, Pandas 데이터프레임 등 지원.





매개변수	설명
<code>`y`</code>	타겟 데이터 (라벨 또는 목표 변수).
<code>`test_size`</code>	테스트 데이터 비율 (0.2는 20%). 또는 샘플 개수 지정 가능.
<code>`train_size`</code>	훈련 데이터 비율 (명시하지 않으면 <code>`1 - test_size`</code> ).
<code>`random_state`</code>	랜덤 시드. 값을 고정하면 항상 동일한 분할 결과를 얻을 수 있음.
<code>`shuffle`</code>	데이터를 섞을지 여부 (기본값: True).
<code>`stratify`</code>	계층 샘플링을 위해 사용. 분류 문제에서 각 클래스 비율 유지.

## 결과

- ``train_test_split``은 데이터를 훈련용과 테스트용으로 나눠주는 **4개의 반환값**을 제공합니다:
  - ``X_train``: 훈련 데이터 (특징).
  - ``X_test``: 테스트 데이터 (특징).
  - ``y_train``: 훈련 데이터 (타겟).
  - ``y_test``: 테스트 데이터 (타겟).

예:

```
X = [[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]]
y = [0, 1, 0, 1, 0, 1, 0, 1, 0, 1]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

print("X_train:", X_train)
print("X_test:", X_test)
print("y_train:", y_train)
print("y_test:", y_test)
```

출력:

```
X_train: [[8], [3], [7], [1], [9], [6], [2]]
X_test: [[10], [4], [5]]
```



```
y_train: [1, 0, 0, 0, 0, 1, 1]
y_test: [1, 1, 0]
```

## 🚀 왜 사용해야 할까?

1. **훈련 데이터와 테스트 데이터를 분리**해야 모델의 일반화 성능을 공정하게 평가할 수 있습니다.
2. 데이터셋이 무작위로 섞이므로, 특정 순서나 패턴이 성능에 영향을 주는 것을 방지합니다.
3. 테스트 데이터는 모델이 학습하지 않은 데이터로, 과적합 여부를 확인할 수 있습니다.

## ? 필수적인 경우

- 데이터를 분할하지 않으면, 모델이 훈련 데이터와 테스트 데이터 모두에서 학습하게 되므로 **과적합**이 발생할 위험이 큼니다.
- 데이터를 분할하는 것은 대부분의 머신러닝 작업에서 필수적인 과정입니다.

추가적으로 궁금한 점이 있다면 말씀해주세요! ☺



xgboost에서 모델을 fit할 때 early stopped하는 과정이 필요하지 않나?



네, 맞습니다! **XGBoost**에서 모델을 훈련할 때, **\*\*early stopping(조기 종료)\*\***는 매우 중요한 역할을 할 수 있습니다.

조기 종료는 **과적합(overfitting)**을 방지하고 불필요한 학습을 줄이는 데 유용합니다.

## ★ Early Stopping이란?

- 검증 데이터의 성능이 일정한 횟수 동안 개선되지 않으면 학습을 멈춥니다.
- 불필요한 반복(epoch)을 줄여 시간을 절약하고, 최적의 모델 상태에서 학습을 종료합니다.



## 🔧 XGBoost에서 Early Stopping 설정

XGBoost는 `fit` 메서드의 `early_stopping_rounds` 파라미터를 사용하여 조기 종료를 설정합니다.

- `early_stopping_rounds=n`: 검증 점수가 `n`번의 반복 동안 개선되지 않으면 학습을 멈춥니다.
- `eval_set`: 검증에 사용할 데이터셋을 지정합니다.
- `eval_metric`: 성능 평가를 위한 지표를 지정합니다.

### 📖 코드 예제

아래는 `XGBRegressor`에서 조기 종료를 적용하는 코드입니다:

```
import xgboost as xgb
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# 데이터 로드 및 분할
X, y = load_some_data() # 데이터를 로드하는 함수
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# XGBoost 모델 초기화
xgb_model = xgb.XGBRegressor(
    n_estimators=1000,      # 최대 트리 개수 (많게 설정)
    learning_rate=0.05,     # 학습 속도
    max_depth=6,            # 최대 트리 깊이
    random_state=42
)

# 조기 종료 설정
eval_set = [(X_train, y_train), (X_test, y_test)] # 검증 데이터 지정
xgb_model.fit(
    X_train, y_train,
    eval_set=eval_set,
    eval_metric='rmse',      # 평가 지표 설정 (RMSE 사용)
    early_stopping_rounds=50, # 50번 동안 성능 개선 없으면 종료
    verbose=True             # 학습 로그 출력
)

# 테스트 데이터 평가
y_pred = xgb_model.predict(X_test)
```



```
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse:.4f}")
```

## 💡 코드 설명

### 1. `n\_estimators`

- 초기에는 많은 트리 개수를 설정합니다 (예: 1000). 조기 종료로 학습을 적절한 시점에서 멈춥니다.

### 2. `early\_stopping\_rounds`

- 검증 데이터의 성능이 지정된 라운드 수만큼 개선되지 않으면 학습을 멈춥니다.

### 3. `eval\_set`

- 학습 중 검증에 사용할 데이터셋을 지정합니다. 훈련 데이터와 테스트 데이터의 점수를 모두 계산합니다.

### 4. `eval\_metric`

- 평가 지표를 지정합니다. 회귀에서는 `rmse`, 분류에서는 `logloss`, `auc` 등이 자주 사용됩니다.

## 📊 출력 예시

```
[0] validation_0-rmse:2500.12345 validation_1-rmse:2600.56789
[1] validation_0-rmse:2000.67890 validation_1-rmse:2100.54321
...
[50] validation_0-rmse:150.12345 validation_1-rmse:160.54321
Stopping. Best iteration:
[50] validation_0-rmse:150.12345 validation_1-rmse:160.54321
```

## 📈 학습 시각화

학습 및 검증 손실을 그래프로 시각화하여 조기 종료를 확인할 수도 있습니다:



```
results = xgb_model.evals_result()
epochs = len(results['validation_0']['rmse'])
x_axis = range(0, epochs)

plt.figure(figsize=(10, 6))
plt.plot(x_axis, results['validation_0']['rmse'], label='Train RMSE')
plt.plot(x_axis, results['validation_1']['rmse'], label='Test RMSE')
plt.xlabel('Epochs')
plt.ylabel('RMSE')
plt.title('XGBoost RMSE by Epochs')
plt.legend()
plt.grid()
plt.show()
```

## 🤖 Early Stopping이 중요한 이유

- 학습이 과도하게 진행되면, 모델이 훈련 데이터에 과적합할 가능성이 있습니다.
- 검증 성능이 개선되지 않는 시점에서 학습을 멈추면 더 나은 일반화 성능을 얻을 수 있습니다.

추가적으로 궁금한 사항이나 코드에 대한 수정 요청이 있다면 편하게 말씀해주세요! 😊



- ↳ 아래는 `XGBRegressor`에서 조기 종료를 적용하는 코드입니다: python 코드 복사 `import xgboost as xgb from sklearn.model\_selection import train\_test\_split from sklearn.metrics import mean\_squared\_error # 데이터 로드 및 분할 X, y = load\_some\_data() # 데이터를 로드하는 함수 X\_train, X\_test, y\_train, y\_test = ...`

XGBModel.fit() got an unexpected keyword argument  
'early\_stopping\_rounds'  
이런 에러가 떠



`XGBModel.fit()`에서 `early\_stopping\_rounds`와 관련된 에러는 **XGBoost의 버전**이나 **함수 호출 방식**에서 문제가 발생했을 가능성이 큼니다. 아래에서 에러 원인과 해결 방법을 단계적으로 설명하겠습니다.



## 원인 분석

### 1. XGBoost 버전 확인:

- 최신 버전(XGBoost  $\geq 1.3.0$ )에서는 `early_stopping_rounds` 파라미터가 `fit()` 메서드에 추가되었습니다.
- 오래된 버전을 사용 중이라면 `early_stopping_rounds`가 지원되지 않을 수 있습니다.

### 2. XGBModel의 기본 클래스:

- `XGBModel`은 `XGBRegressor`나 `XGBClassifier`의 부모 클래스입니다.
- `XGBRegressor`와 `XGBClassifier`에서만 `early_stopping_rounds`를 사용하는 것이 권장됩니다.

### 3. 잘못된 클래스 사용:

- 사용자가 `XGBModel`을 직접 호출한 경우, 해당 클래스는 조기 종료를 지원하지 않을 수 있습니다.
- 대신 `XGBRegressor` 또는 `XGBClassifier`를 사용해야 합니다.

## 해결 방법

### 1. XGBoost 버전 확인 및 업데이트

최신 버전의 XGBoost를 설치하세요:

```
pip install xgboost --upgrade
```

설치 후, 현재 버전을 확인:

```
import xgboost as xgb
print(xgb.__version__)
```

최소 **1.3.0** 이상이 필요합니다.

### 2. `XGBRegressor` 또는 `XGBClassifier` 사용

`XGBRegressor`나 `XGBClassifier`는 `early_stopping_rounds`를 지원합니다.

`XGBModel` 대신 다음을 사용하세요:



```
xgb_model = xgb.XGBRegressor(  
    n_estimators=1000,  
    learning_rate=0.05,  
    max_depth=6,  
    random_state=42  
)  
  
# 조기 종료 적용  
xgb_model.fit(  
    X_train, y_train,  
    eval_set=[(X_train, y_train), (X_test, y_test)],  
    eval_metric='rmse',  
    early_stopping_rounds=50,  
    verbose=True  
)
```

### 3. `early\_stopping\_rounds`를 수동으로 구현

만약 위 방법이 해결되지 않는다면, 조기 종료를 직접 구현할 수 있습니다. 각 반복(epoch)마다 검증 성능을 모니터링하여 중단 조건을 설정합니다:

```
from sklearn.metrics import mean_squared_error  
  
best_rmse = float("inf")  
best_iteration = 0  
tolerance = 50  
  
for i in range(1000): # 최대 반복  
    xgb_model = xgb.XGBRegressor(  
        n_estimators=i+1,  
        learning_rate=0.05,  
        max_depth=6,  
        random_state=42  
    )  
    xgb_model.fit(X_train, y_train)  
    y_val_pred = xgb_model.predict(X_test)  
    rmse = mean_squared_error(y_test, y_val_pred, squared=False)  
  
    if rmse < best_rmse:  
        best_rmse = rmse  
        best_iteration = i  
    elif i - best_iteration >= tolerance:  
        print(f"Early stopping at iteration {i}")  
        break
```



## ⚠️ 주요 체크리스트

- **클래스 사용 확인:** 반드시 `XGBRegressor` 또는 `XGBClassifier`를 사용하세요.
- **데이터 형식 확인:** `eval_set`에 전달되는 데이터는 NumPy 배열 또는 Pandas 데이터프레임이어야 합니다.
- **파라미터 오타 확인:** `early_stopping_rounds`에 오타가 있는지 확인하세요.

추가 질문이 있다면 말씀해주세요! 😊



early\_stopping\_rounds를 사용하려면 eval 데이터 셋을 명기해야 한다는데?



맞습니다! **XGBoost**에서 `early_stopping_rounds`를 사용하려면 반드시 `eval_set` 매개변수를 지정해야 합니다. `eval_set`은 모델이 학습 도중 검증에 사용할 데이터셋을 지정하는 역할을 합니다. 이를 통해 XGBoost는 **검증 데이터**에서의 성능 변화를 모니터링하며, 조기 종료를 수행합니다.

## 🌟 `eval_set`의 역할

- `eval_set=[(X_train, y_train), (X_val, y_val)]`: 학습 및 검증 데이터를 지정합니다.
  - **첫 번째 튜플:** 훈련 데이터 (옵션).
  - **두 번째 튜플:** 검증 데이터 (필수).
- 검증 데이터에서의 성능이 `early_stopping_rounds` 동안 개선되지 않으면 학습을 종료합니다.





## `eval\_set` 과 `early\_stopping\_rounds` 사용 예제

```
import xgboost as xgb
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# 데이터 분할
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

# XGBoost 모델 설정
xgb_model = xgb.XGBRegressor(
    n_estimators=1000,      # 최대 트리 개수
    learning_rate=0.05,     # 학습 속도
    max_depth=6,            # 최대 트리 깊이
    random_state=42
)

# 조기 종료를 위한 검증 데이터 설정
eval_set = [(X_train, y_train), (X_val, y_val)]

# 모델 학습
xgb_model.fit(
    X_train, y_train,
    eval_set=eval_set,      # 검증 데이터 지정
    eval_metric="rmse",     # 평가 지표
    early_stopping_rounds=50, # 50회 동안 성능 개선이 없으면 종료
    verbose=True            # 학습 로그 출력
)

# 테스트 데이터 평가
y_pred = xgb_model.predict(X_val)
mse = mean_squared_error(y_val, y_pred)
print(f"Mean Squared Error: {mse:.4f}")
```

## `eval\_set` 출력

`verbose=True`로 설정하면 학습 도중 검증 성능 로그가 출력됩니다. 예:

```
[0] validation_0-rmse:3500.45678    validation_1-rmse:3600.78901
[1] validation_0-rmse:2800.12345    validation_1-rmse:2900.67890
...
[50] validation_0-rmse:150.12345    validation_1-rmse:160.54321
```



```
Stopping. Best iteration:  
[50]   validation_0-rmse:150.12345 validation_1-rmse:160.54321
```

## 💡 주요 체크포인트

### 1. `eval\_set` 지정 필수:

- 검증 데이터는 학습 데이터와 구분해야 합니다. 그렇지 않으면 모델이 과적합될 수 있습니다.

### 2. `eval\_metric` 지정:

- 회귀 문제: `rmse`, `mae`
- 분류 문제: `logloss`, `auc` 등

### 3. `early\_stopping\_rounds` 값:

- 너무 작게 설정하면 학습이 충분히 진행되기 전에 멈출 수 있으니, 적절한 값(예: 10~50)을 선택하세요.

## 🚀 학습 과정 시각화

학습 및 검증 성능 변화를 시각화하면 조기 종료 지점을 쉽게 이해할 수 있습니다.

```
results = xgb_model.evals_result()  
epochs = len(results['validation_0']['rmse'])  
x_axis = range(0, epochs)  
  
plt.figure(figsize=(10, 6))  
plt.plot(x_axis, results['validation_0']['rmse'], label='Train RMSE')  
plt.plot(x_axis, results['validation_1']['rmse'], label='Validation RMSE')  
plt.xlabel('Epochs')  
plt.ylabel('RMSE')  
plt.title('XGBoost RMSE by Epochs')  
plt.legend()  
plt.grid()  
plt.show()
```



## ? 추가 질문

- 학습 데이터와 검증 데이터 분리에 대한 상세한 방법이나 다른 지표에 대한 질문이 있다면 언제든지 물어보세요! 😊



내 코드 보고 왜 xgboost의 earlystopping rounds가 안되는지 봐봐

<https://github.com/gaaeun/-/blob/main/diamonds%20tesk.ipynb>



현재 인터넷이 비활성화된 상태라 GitHub 링크의 내용을 직접 확인할 수 없습니다. 대신, 일반적으로 **XGBoost**에서 `early_stopping_rounds`가 작동하지 않는 경우의 주요 원인과 해결 방법을 제공하겠습니다. 그런 다음 문제가 여전히 해결되지 않으면 코드를 붙여넣어 주시면 세부적으로 분석해 드릴게요! 😊

## 🔍 `early_stopping_rounds` 작동 안 되는 주요 원인

### 1. `eval_set` 누락:

- `early_stopping_rounds`는 반드시 `eval_set`과 함께 사용되어야 합니다.
- 검증 데이터셋이 지정되지 않으면 조기 종료를 수행할 수 없습니다.

### 2. `eval_set` 형식 문제:

- `eval_set`은 리스트 형태로 지정되어야 하며, 각 요소는 `(x, y)`의 튜플입니다.
- 예: `eval_set=[(X_train, y_train), (X_val, y_val)]`.

### 3. 클래스 타입 문제:

- `XGBModel` 대신 `XGBRegressor` 또는 `XGBClassifier`를 사용해야 합니다.
- `XGBModel`은 부모 클래스이므로 일부 기능을 지원하지 않을 수 있습니다.

### 4. XGBoost 버전:



- 최신 버전(XGBoost >= 1.3.0)을 사용해야 합니다. 오래된 버전에서는 `early\_stopping\_rounds`가 `fit` 메서드에서 작동하지 않을 수 있습니다.

#### 5. 데이터 타입 문제:

- `X\_train`, `y\_train` 등 데이터가 NumPy 배열이나 Pandas 데이터프레임 형식이어야 합니다.
- 리스트, 딕셔너리 등의 타입은 지원하지 않습니다.

#### 6. 특별한 설정 문제:

- 예를 들어, `n\_estimators`를 너무 낮게 설정하면 `early\_stopping\_rounds`가 의미가 없어질 수 있습니다.

## 해결 방법

### 1. 검증 데이터 설정

코드에 `eval\_set`이 제대로 지정되어 있는지 확인하세요:

```
xgb_model.fit(  
    X_train, y_train,  
    eval_set=[(X_train, y_train), (X_val, y_val)], # 검증 데이터 지정  
    eval_metric='rmse',                          # 평가 지표  
    early_stopping_rounds=50,                      # 조기 종료 조건  
    verbose=True  
)
```

### 2. XGBoost 클래스 확인

`XGBRegressor` 또는 `XGBClassifier`를 사용해야 합니다:

```
from xgboost import XGBRegressor  
  
xgb_model = XGBRegressor(  
    n_estimators=1000,  
    learning_rate=0.05,  
    max_depth=6,  
    random_state=42  
)
```

### 3. 데이터 타입 확인



모든 데이터가 NumPy 배열이나 Pandas 데이터프레임인지 확인하세요:

```
import numpy as np
import pandas as pd

print(type(X_train)) # 출력: <class 'numpy.ndarray'> 또는 <class 'pandas.core.frame.DataFrame'>
print(type(y_train)) # 출력: <class 'numpy.ndarray'> 또는 <class 'pandas.core.series.Series'>
```

필요하다면 데이터를 변환하세요:

```
import numpy as np
X_train = np.array(X_train)
y_train = np.array(y_train)
```

#### 4. 버전 업데이트

XGBoost 최신 버전으로 업데이트하세요:

```
pip install xgboost --upgrade
```

#### 5. `early\_stopping\_rounds`와 로그 출력 확인

코드에 `verbose=True`를 추가하여 학습 진행 상황을 확인하세요:

```
xgb_model.fit(
    X_train, y_train,
    eval_set=[(X_train, y_train), (X_val, y_val)],
    eval_metric='rmse',
    early_stopping_rounds=50,
    verbose=True
)
```

#### ≤ 코드에 문제가 있다면?

- 위 조치가 문제가 해결되지 않으면, **코드를 붙여넣어 주시면** 보다 정확하게 진단할 수 있습니다.
- 다시 확인이 필요한 부분이 있다면 알려주세요! ☺





```
#xgboost 사용하기 위해 설치 진행.
!pip install xgboost --quiet
from xgboost import XGBRegressor
#필요한 라이브러리 불러옴.
import pandas as pd
import numpy as np
import xgboost as xgb
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

from sklearn.metrics import mean_squared_error, r2_score
data = pd.read_csv("diamonds.csv") #데이터 가져오기
data.head() #5번째 행까지 데이터 보기

data = data.drop(columns=["Unnamed: 0"], errors="ignore") #특
징이 아닌 첫번째 데이터 삭제
data.head() #5번째 행까지 데이터 보기

print("\n데이터 요약 정보:\n")
print(data.info()) # 데이터 타입 확인
```



아래는 제공된 코드와 함께 발생할 수 있는 문제와 관련된 **분석** 및 **해결 방법**입니다.

## 1 현재 코드 상태

- 데이터 로드 및 처리:
  - ``diamonds.csv`` 파일을 불러와 ``Unnamed: 0`` 열을 삭제했습니다.
  - ``data.head()`` 및 ``data.info()``를 호출해 데이터를 탐색 중입니다.
- XGBoost 사용 준비:
  - ``XGBRegressor``를 사용하려고 준비 중이며, ``pip install xgboost``를 통해 설치도 완료했습니다.

## 2 코드 다음 단계: ``train_test_split``을 사용해 데이터 분할

현재 제공된 코드만으로는 ``early_stopping_rounds`` 관련 설정이 없어서, 데이터 분할 및 모델 학습이 다음으로 필요할 것 같습니다. 아래 코드를 추가하세요:



```
# 타겟 및 입력 데이터 정의
X = data.drop(['price'], axis=1) # 특징 데이터
y = data['price']                # 타겟 데이터 (가격)

# 원-핫 인코딩
X = pd.get_dummies(X, drop_first=True)

# 데이터 분할
X_train, X_val, y
```



```
print("\n기술 통계 정보:\n")
print(data.describe()) # 수치형 변수 통계

print("\n결측치 확인:")
print(data.isnull().sum())
# 중복된 행 확인
duplicates = data.duplicated()
print(f"중복된 데이터 개수: {duplicates.sum()}")

# 중복된 데이터 제거
print(f"중복 제거 전 데이터 크기: {data.shape}")
data = data.drop_duplicates()
print(f"중복 제거 후 데이터 크기: {data.shape}")
data = pd.get_dummies(data, prefix = ['cut', 'color', 'clarity'],
columns=['cut', 'color', 'clarity'])
data = data.drop(columns=['x', 'y', 'z'])
data.head()

var = [x for x in data.columns if x not in 'price'] ## Removing
our target variable
print(len(var))
23
help(train_test_split)
X_train, X_test, y_train, y_test = train_test_split(data[var],
data['price'], test_size=0.2, random_state=42)
n = int(len(X_train)*0.8) ## Let's use 80% to train and 20% to
eval
X_train_fit, X_train_eval, y_train_fit, y_train_eval = X_train[:n],
X_train[n:], y_train[:n], y_train[n:]
print(f'train samples: {len(X_train)}\ntest samples:
{len(X_test)}')
```





output: train samples: 43035  
test samples: 10759



코드의 흐름을 보니, 데이터 전처리 및 분할 과정은 잘 진행되고 있습니다. 주요 내용과 검토 사항을 아래에 정리했습니다.

## 🔍 현재 코드 요약

### 1. 데이터 확인:

- `describe()`와 `isnull().sum()`으로 데이터의 기술 통계 및 결측값 확인.
- 중복된 데이터가 제거되었고, 차원이 변경되었음을 확인했습니다.

### 2. 원-핫 인코딩:

- `cut`, `color`, `clarity`가 원-핫 인코딩 처리됨.
- `x`, `y`, `z`가 불필요한 특성으로 삭제됨.

### 3. 데이터 분할:

- `train_test_split`을 사용해 데이터를 훈련 데이터와 테스트 데이터로 분리.
- `x_train`에서 다시 80:20으로 훈련용(`x_train_fit`)과 검증용(`x_train_eval`)으로 추가 분리.

## ? 문제: XGBoost와 조기 종료 관련

현재 코드를 보면 `early_stopping_rounds`를 사용할 준비는 거의 완료된 상태입니다. 다만, `XGBRegressor`에서 `eval_set`를 정확히 설정해야 조기 종료가 작동합니다. `eval_set` 설정을 통해 검증 데이터(`x_train_eval`, `y_train_eval`)를 지정하지 않으면 `early_stopping_rounds`가 작동하지 않습니다.

## 🔧 XGBoost 학습 및 조기 종료 코드





아래 코드를 추가해 `early\_stopping\_rounds`가 제대로 작동하도록 구현하세요:

```
from xgboost import XGBRegressor
from sklearn.metrics import mean_squared_error

# XGBoost 모델 설정
xgb_model = XGBRegressor(
    n_estimators=1000,      # 최대 트리 개수
    learning_rate=0.05,     # 학습 속도
    max_depth=6,           # 최대 트리 깊이
    random_state=42
)

# 조기 종료를 위한 검증 데이터 지정
eval_set = [(X_train_fit, y_train_fit), (X_train_eval, y_train_eval)]

# 모델 학습
xgb_model.fit(
    X_train_fit, y_train_fit,
    eval_set=eval_set,      # 검증 데이터 지정
    eval_metric='rmse',     # 평가 지표
    early_stopping_rounds=50, # 50번 동안 성능 개선이 없으면 종료
    verbose=True            # 학습 로그 출력
)

# 테스트 데이터 평가
y_pred = xgb_model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error on Test Data: {mse:.4f}")
```

## 학습 결과 시각화

조기 종료가 작동하는지 확인하려면 학습 과정의 손실 변화를 그래프로 시각화하면 유용합니다:

```
# 학습 및 검증 성능 변화 시각화
results = xgb_model.evals_result()
epochs = len(results['validation_0']['rmse'])
x_axis = range(0, epochs)

plt.figure(figsize=(10, 6))
plt.plot(x_axis, results['validation_0']['rmse'], label='Train RMSE')
plt.plot(x_axis, results['validation_1']['rmse'], label='Validation RMSE')
```



```
plt.xlabel('Epochs')
plt.ylabel('RMSE')
plt.title('XGBoost RMSE by Epochs')
plt.legend()
plt.grid()
plt.show()
```

## ⚠ 주요 체크포인트

1. `early_stopping_rounds`를 사용하려면 `eval_set`이 필수입니다.
  - `eval_set`에 검증 데이터(`X_train_eval`, `y_train_eval`)를 지정했는지 확인하세요.
2. 데이터 형식:
  - 모든 데이터는 NumPy 배열이나 Pandas 데이터프레임이어야 합니다.
3. `eval_metric`:
  - 회귀 문제이므로 `rmse`나 `mae`를 사용하는 것이 적합합니다.
4. `verbose=True`:
  - 학습 진행 상황을 확인할 수 있도록 로그를 출력하세요.

추가적인 질문이나 문제가 있다면 말씀해주세요! 😊



```
xgb_model = xgb.XGBRegressor(
    n_estimators=1000,    # 최대 트리 개수
    learning_rate=0.05,   # 학습 속도
    max_depth=6,          # 최대 트리 깊이
    random_state=42
)

xgb_model.fit(X_train_fit, y_train_fit, eval_set=
[(X_train_eval,y_train_eval)], early_stopping_rounds=50,
verbose=True)
```

TypeError

Traceback (most recent call last)



Cell In[24], line 8

```
1 xgb_model = xgb.XGBRegressor(
2     n_estimators=1000,    # 최대 트리 개수
3     learning_rate=0.05,   # 학습 속도
4     max_depth=6,         # 최대 트리 깊이
5     random_state=42
6 )
```

```
----> 8 xgb_model.fit(X_train_fit, y_train_fit, eval_set=
[(X_train_eval,y_train_eval)], early_stopping_rounds=50,
verbose=True)
```

```
File ~\AppData\Roaming\Python\Python311\site-
packages\xgboost\core.py:726, in require_keyword_args.
<locals>.throw_if.<locals>.inner_f(*args, **kwargs)
724 for k, arg in zip(sig.parameters, args):
725     kwargs[k] = arg
--> 726 return func(**kwargs)
```

TypeError: XGBModel.fit() got an unexpected keyword argument 'early\_stopping\_rounds'



TypeError: XGBModel.fit() got an unexpected keyword argument

'early\_stopping\_rounds'라는 에러는 `XGBoost` 라이브러리와 관련하여 **파라미터 전달 방식** 또는 **라이브러리 버전 문제**로 발생합니다. 아래에서 주요 원인과 해결 방법을 정리했습니다.

## 🔍 문제 원인

### 1. XGBoost 버전 문제:

- 최신 XGBoost 라이브러리에서는 `early\_stopping\_rounds`가 `fit` 메서드에 포함되어 있으나, 오래된 버전에서는 작동하지 않을 수 있습니다.

### 2. 클래스 타입 확인:

- `XGBModel`은 `XGBRegressor`와 `XGBClassifier`의 부모 클래스입니다. 하지만 부모 클래스인 `XGBModel`에서는 일부 파라미터가 제대로 지원되지 않을 수 있습니다.

### 3. 잘못된 파라미터 사용:





- 일부 경우, `fit` 메서드가 `eval_set`과 `early_stopping_rounds`를 제대로 처리하지 못할 수 있습니다.

## 🔧 해결 방법

### 1 XGBoost 버전 확인

- 먼저 XGBoost의 버전을 확인하고, **최신 버전(1.3.0 이상)**으로 업데이트하세요:

```
pip install xgboost --upgrade
```

- 현재 버전 확인:

```
import xgboost as xgb
print(xgb.__version__)
```

### 2 XGBRegressor 또는 XGBClassifier 사용

- `XGBRegressor`를 사용하는지 확인하세요. `XGBModel` 클래스가 사용되고 있다면 `XGBRegressor`로 변경해야 합니다:

```
from xgboost import XGBRegressor

xgb_model = XGBRegressor(
    n_estimators=1000,
    learning_rate=0.05,
    max_depth=6,
    random_state=42
)
```

### 3 Fit 메서드 매개변수 확인

- `eval_set`과 `early_stopping_rounds`는 `fit` 메서드에서 지원됩니다. 아래 코드처럼 전달해야 합니다:



```
xgb_model.fit(  
    X_train_fit, y_train_fit,  
    eval_set=[(X_train_eval, y_train_eval)],  
    early_stopping_rounds=50,  
    verbose=True  
)
```

#### 4 다른 방법: 학습자 객체 사용

XGBoost의 낮은 버전에서 `fit` 메서드가 제한적일 경우, 학습자 객체를 수동으로 생성하고 `train` 메서드를 사용하는 방법도 있습니다:

```
import xgboost as xgb  
  
dtrain = xgb.DMatrix(X_train_fit, label=y_train_fit)  
dval = xgb.DMatrix(X_train_eval, label=y_train_eval)  
  
params = {  
    "objective": "reg:squarederror",  
    "learning_rate": 0.05,  
    "max_depth": 6,  
    "n_estimators": 1000  
}  
  
# 학습 진행  
model = xgb.train(  
    params,  
    dtrain,  
    num_boost_round=1000,  
    evals=[(dtrain, "train"), (dval, "eval")],  
    early_stopping_rounds=50,  
    verbose_eval=True  
)
```

#### 확인 후 해결

위의 방법 중 1~3로 문제가 해결되지 않으면 4의 수동 학습 방식을 시도해 보세요.  
이후에도 문제가 발생한다면 **사용 중인 XGBoost의 버전과 데이터 형식**을 알려주시면 추가로 도와드리겠습니다. 😊



module 'pandas' has no attribute 'Int64Index'

이런 에러가 떠



``module 'pandas' has no attribute 'Int64Index'`` 에러는 **Pandas 라이브러리 버전** 문제에서 주로 발생합니다. ``Int64Index``는 Pandas의 이전 버전에서 사용되던 속성으로, 최신 버전에서는 다른 방식으로 대체되었습니다. 이 에러를 해결하려면 아래 단계를 따라야 합니다.

## 🔍 문제 원인

- Pandas 버전이 최신 라이브러리(1.4.0 이상)로 업그레이드되면서, ``Int64Index``와 같은 속성이 내부적으로 변경되었습니다.
- 사용 중인 코드 또는 의존성이 ``Int64Index``를 참조하려 하지만, 최신 버전에서는 해당 속성을 사용할 수 없습니다.

## 🔧 해결 방법

### 1 Pandas 버전 확인

- 현재 설치된 Pandas 버전을 확인합니다:

```
import pandas as pd
print(pd.__version__)
```

### 2 Pandas 업그레이드

- 최신 Pandas를 설치하거나 업그레이드합니다:

```
pip install --upgrade pandas
```

### 3 코드에서 ``Int64Index``가 사용되는지 확인

- 직접 작성한 코드에서 ``Int64Index``를 사용하는 부분이 있다면, 대체 방식을 사용해야 합니다. 최신 Pandas 버전에서는 ``pd.Index``를 사용하는 것이 더 일반적입니다:



```
import pandas as pd

# 예전 방식
my_index = pd.Int64Index([1, 2, 3, 4])

# 대체 방식
my_index = pd.Index([1, 2, 3, 4], dtype='int64')
print(my_index)
```

#### 4 호환성 문제가 있는 라이브러리 업데이트

- 사용 중인 다른 라이브러리가 Pandas의 오래된 속성을 참조할 수 있습니다.
- 모든 의존성을 업데이트합니다:

```
pip install --upgrade --no-deps numpy pandas
```

#### 만약 업데이트가 어려운 경우

- Pandas를 특정 이전 버전으로 다운그레이드하면 문제가 해결될 수 있습니다. 예를 들어, `Int64Index`가 지원되던 1.3.5 버전으로 다운그레이드:

```
pip install pandas==1.3.5
```

#### ? 추가 질문

- 에러가 발생한 코드의 구체적인 부분을 알려주시면, 정확한 수정 방향을 제안할 수 있습니다.
- 라이브러리 호환성 문제 해결에 대해 추가 질문이 있다면 언제든지 말씀해주세요! 😊



↳ **\*\*\*만약 업데이트가 어려운 경우\*\*\*** Pandas를 특정 이전 버전으로 다운그레이드하면 문제가 해결될 수 있습니다. 예를 들어, `Int64Index`가 지원되던 1.3.5 버전으로 다운그레이드: bash 코드 복사 ``pip install pandas==1.3.5``

다운그레이드 할래 . 먼저 설치한 거 삭제하면 되나?



네! 다운그레이드를 하려면 먼저 현재 설치된 Pandas를 삭제하거나, 특정 버전을 직접 설치하면 됩니다. 아래에 상세한 단계와 명령어를 정리했습니다. 😊

## 🔧 1. Pandas 삭제

Pandas를 삭제하려면 다음 명령어를 사용하세요:

```
pip uninstall pandas
```

이 명령어를 실행하면 Pandas가 삭제됩니다. 삭제 후, 특정 버전을 다시 설치할 수 있습니다.

## 🔧 2. Pandas 특정 버전 설치

삭제한 후 특정 버전(예: 1.3.5)을 설치하려면:

```
pip install pandas==1.3.5
```

## 🔧 3. 한 번에 다운그레이드

삭제하지 않고 바로 특정 버전을 설치해도 됩니다. 아래 명령어를 사용하면 기존 Pandas가 자동으로 업그레이드 또는 다운그레이드됩니다:

```
pip install pandas==1.3.5 --upgrade
```

## 🖋️ 설치 확인

설치가 완료되면, 다음 명령어로 현재 Pandas 버전을 확인하세요:





```
import pandas as pd
print(pd.__version__) # 출력: 1.3.5
```

## ⚠ 주의 사항

### 1. 다른 라이브러리와 호환성:

- Pandas를 다운그레이드하면 다른 라이브러리(예: `numpy` 등)와 호환성 문제가 발생할 수 있으니, 필요한 경우 함께 다운그레이드하세요.
- 의존성을 무시하고 설치하려면:

```
pip install pandas==1.3.5 --no-deps
```

### 2. 환경 분리 권장:

- 여러 프로젝트에서 다른 버전의 Pandas를 사용하려면 가상 환경(virtual environment)을 사용하는 것이 좋습니다:

```
python -m venv myenv
source myenv/bin/activate # Windows는 myenv\Scripts\activate
pip install pandas==1.3.5
```

궁금한 점이 있거나 도움이 더 필요하다면 알려주세요! ☺



```
!pip install xgboost --quiet
!pip install pandas==1.3.5 --upgrade
```

이렇게 주피터 노트북으로 했는데 왜 실행중이라고 [\*] 이 표시만 뜨고 실행이 안되는 거 같지



주피터 노트북에서 `pip` 명령어 실행 시 `[ \* ]`로 표시되며 실행이 멈춘 것처럼 보이는 경우는 주로 패키지 설치 중간에 문제가 발생했기 때문일 가능성이 큼니다. 아래에 가능한 원인과 해결 방



법을 정리했습니다.

## 주요 원인

### 1. 인터넷 연결 문제:

- 패키지 설치를 위해 인터넷 연결이 필요합니다. 네트워크 연결을 확인하세요.

### 2. 노트북 내부 충돌:

- 주피터 노트북에서 `pip` 설치 명령어가 실행 중 다른 작업(예: 커널 작업)과 충돌이 발생할 수 있습니다.

### 3. `pip` 명령어 실행 중 대기 상태:

- 설치 과정이 오래 걸리거나, 설치 중 중단될 수 있습니다.
- 특히 특정 버전으로 설치하거나, 기존 라이브러리를 제거한 후 설치 시 시간이 더 오래 걸릴 수 있습니다.

### 4. 환경 권한 문제:

- 가상 환경(venv)이나 시스템 권한에 따라 설치가 제한될 수 있습니다.

## 해결 방법

### 1 노트북에서 `!pip` 명령 대신 터미널 사용

주피터 노트북 내부에서 설치가 진행되지 않으면, **터미널(또는 명령 프롬프트)**에서 직접 설치를 실행하세요:

```
pip install xgboost --quiet
pip install pandas==1.3.5 --upgrade
```

### 2 노트북 내 설치 명령을 재실행

주피터 노트북에서 실행할 경우, 다음 코드를 사용하여 `!pip` 명령어를 재실행합니다:



```
# XGBoost 설치
!pip install xgboost --quiet
# Pandas 특정 버전 설치
!pip install pandas==1.3.5 --upgrade
```

### 3 주피터 커널 재시작

설치 후 주피터 노트북의 커널을 재시작해야 반영되는 경우가 많습니다:

- 메뉴에서 **Kernel > Restart Kernel**을 선택하세요.

### 4 설치 진행 상황 확인

`pip` 설치 중 중단된 상태라면, 터미널에서 설치 로그를 자세히 확인할 수 있습니다:

```
pip install pandas==1.3.5 --verbose
```

### 5 노트북 실행 중지 후 재시작

설치 중 진행되지 않는 경우 **노트북을 강제로 중단 후 다시 시작**합니다:

- 메뉴에서 **Kernel > Interrupt Kernel** 또는 **Restart Kernel**을 선택.

### 6 설치 테스트

설치가 완료되었는지 확인하려면:

```
import pandas as pd
import xgboost as xgb
print(f"Pandas version: {pd.__version__}")
print(f"XGBoost version: {xgb.__version__}")
```



## 🚀 추가 팁: 가상 환경 사용

환경이 복잡해지거나 설치가 제대로 되지 않는다면, 가상 환경을 사용하는 것도 좋은 방법입니다:

```
python -m venv myenv
source myenv/bin/activate # Windows는 myenv\Scripts\activate
pip install xgboost pandas==1.3.5
```

문제가 여전히 해결되지 않는다면 설치 시 출력되는 로그나 에러 메시지를 알려주시면 더 구체적으로 도와드릴게요! 😊



Building wheel for pandas (pyproject.toml) did not run successfully.

exit code: 1

See above for output.

note: This error originates from a subprocess, and is likely not a problem with pip.

full command: 'C:\ProgramData\anaconda3\python.exe' 'C:\ProgramData\anaconda3\Lib\site-packages\pip\\_vendor\pyproject\_hooks\\_in\_process\\_in\_process.py' build\_wheel 'C:\Users\plus\AppData\Local\Temp\tmpdpl4ulk1'

cwd: C:\Users\plus\AppData\Local\Temp\pip-install-nkjc9kzm\pandas\_ef89db5cb61e489ebe5b794197db67cb

ERROR: Failed building wheel for pandas

ERROR: Could not build wheels for pandas, which is required to install pyproject.toml-based projects

설치하는데 이런 에러가 떴어ㅠ



해당 에러는 Pandas 설치 과정에서 **\*\*빌드 휠(빌드 패키지)\*\***이 제대로 생성되지 않아 발생하는 문제입니다. 이 문제는 주로 컴파일러, 의존성 라이브러리, 또는 Python 환경 설정 문제에서 발생합니다. 아래 해결 방법을 단계적으로 따라해 보세요. 😊



## 원인

1. **Python 컴파일러 부족**: Pandas의 일부 종속성은 C 컴파일러를 필요로 합니다.
2. **의존성 라이브러리 부족**: Pandas 설치 시 필요한 라이브러리가 누락된 경우.
3. **Pandas 빌드 환경 충돌**: Python 환경 설정이 올바르지 않거나, 오래된 버전을 사용 중일 때 발생.

## 해결 방법

### 1 Pandas 설치를 위한 필수 라이브러리 설치

- Windows 환경에서는 **C++ 컴파일러**와 **Pandas 의존성 라이브러리**가 필요합니다. 아래 명령을 실행하세요:

```
pip install wheel
pip install setuptools --upgrade
pip install cython
```

### 2 미리 컴파일된 Pandas 바이너리 설치

휠 파일을 직접 다운로드하거나 바이너리 설치를 시도합니다. 다음 명령어를 실행하세요:

```
pip install pandas==1.3.5 --only-binary :all:
```

### 3 Anaconda 환경에서 설치

Anaconda를 사용하는 경우, Pandas는 conda 명령으로 설치하는 것이 훨씬 더 안정적입니다:

```
conda install pandas=1.3.5
```



#### 4 Python 및 Pip 업데이트

Python과 Pip이 최신 버전인지 확인하고, 업데이트를 실행하세요:

```
pip install --upgrade pip setuptools
```

#### Python 버전 확인:

```
python --version
```

Python 3.9 이상을 권장하며, Anaconda 환경이라면 아래 명령어로 환경을 새로 설정할 수 있습니다:

```
conda create -n pandas_env python=3.9
conda activate pandas_env
conda install pandas=1.3.5
```

#### 5 빌드 휠 강제 비활성화

휠 빌드 없이 소스 설치를 강제합니다:

```
pip install pandas==1.3.5 --no-build-isolation
```

#### 6 로컬 캐시 파일 삭제

캐시된 설치 파일이 손상되었을 가능성이 있습니다. 아래 명령으로 삭제한 뒤 다시 설치를 시도하세요:

```
pip cache purge
pip install pandas==1.3.5
```

#### 7 다른 버전 설치

만약 특정 버전(1.3.5)에서 문제가 지속된다면, Pandas의 최신 버전을 사용해 보세요:



```
pip install pandas
```

## 🔴 여전히 문제가 해결되지 않는 경우

### 1. 에러 로그를 확인:

- 에러에 나온 `cwd` 디렉토리와 `tmp` 폴더 경로에서 더 상세한 로그 파일을 확인해 문제 원인을 파악하세요.

### 2. 최소 환경으로 테스트:

- 가상 환경을 만들어 Pandas를 독립적으로 설치해 보세요:

```
python -m venv test_env
source test_env/bin/activate # Windows에서는 test_env\Scripts\activate
pip install pandas==1.3.5
```

위 방법으로도 해결되지 않는다면, 환경 정보(예: Python 버전, OS 정보, 설치 명령어 실행 로그)를 공유해 주시면 추가로 도와드리겠습니다! 😊



- ↳ `module 'pandas' has no attribute 'Int64Index'` 에러는 **\*\*Pandas 라이브러리 버전\*\*** 문제에서 주로 발생합니다. `Int64Index`는 Pandas의 이전 버전에서 사용되던 속성으로, 최신 버전에서는 다른 방식으로 대체되었습니다. 이 에러를 해결하려면 아래 단계를 따라야 합니다.

```
import pandas as pd
from pandas import MultiIndex, Int64Index
import xgboost as xgb
```

그냥 이렇게 할라 그랬는데 안되네...



`Int64Index`와 관련된 문제는 **Pandas 라이브러리의 변경 사항** 때문입니다. 최신 Pandas 버전(2.0.0 이상)에서는 `Int64Index`가 더 이상 독립된 객체로 제공되지 않으며, 대신 일반 `Index` 객체로 대체되었습니다. 따라서 코드를 수정하거나 환경을 조정해야 합니다.



## 🔧 문제 해결 방안

### 1 Pandas 버전을 확인하고 다운그레이드

`Int64Index`를 사용하려는 이유가 기존 코드와의 호환성 때문이라면, Pandas 버전을 다운그레이드하세요:

```
pip install pandas==1.5.3
```

다운그레이드 후, 아래 코드를 사용해 다시 확인하세요:

```
import pandas as pd
from pandas import MultiIndex, Int64Index
print("Pandas version:", pd.__version__)
```

### 2 코드 수정: `Int64Index` 대신 `Index` 사용

Pandas 2.0.0 이상에서는 `Int64Index` 대신 `pd.Index`를 사용할 수 있습니다. 코드를 다음과 같이 수정하세요:

```
import pandas as pd
from pandas import MultiIndex # Int64Index는 더 이상 필요하지 않음

# 예제
index = pd.Index([1, 2, 3], dtype='int64')
print(index)
```

### 3 XGBoost 및 Pandas 간 호환성 확인

XGBoost와 Pandas의 최신 버전은 대부분 호환됩니다. 그러나 문제가 지속된다면 XGBoost와 Pandas 버전을 다음처럼 조정해 보세요:

```
pip install xgboost==1.6.0 pandas==1.3.5
```





#### 4 완전한 예제 코드

위의 수정 사항을 반영한 완전한 예제는 다음과 같습니다:

```
import pandas as pd
import xgboost as xgb

# MultiIndex 예제
multi_index = pd.MultiIndex.from_tuples([('A', 1), ('A', 2), ('B', 1)])
print("MultiIndex:", multi_index)

# 일반 Index
int_index = pd.Index([1, 2, 3], dtype='int64')
print("Index:", int_index)
```

#### ⚠ 주의 사항

- **Int64Index 제거:** Pandas 버전 2.0.0 이상에서는 `Int64Index`를 직접적으로 사용할 수 없으니, 코드를 수정하거나 적합한 버전을 선택하세요.
- **환경 확인:** 호환성 문제가 지속되면 다음 명령어로 설치된 패키지를 확인하세요:

```
pip freeze
```

궁금한 점이나 추가적인 오류가 있다면 말씀해주세요. ☺



→ `### **2\`. XGBoost 적용\*\* XGBoost를 적용하려면 `xgboost` 라이브러리\*\*를 사용하면 됩니다. 아래는 단계별 코드입니다. python 코드 복사 `import xgboost as xgb from sklearn.model_selection import train_test_split from sklearn.metrics import mean_squared_error, r2_score # 데이터 로드 및 전처리 data =...`

음 그냥 early stopping없이 이걸로 하면 문제있나/



**XGBoost**를 `early_stopping_rounds` 없이 사용해도 문제는 없습니다, 다만 몇 가지 주의할 점이 있습니다. `early_stopping_rounds`는 모델이 과적합되는 것을 방지하고 학습 시간을 줄이는 데 도움을 주지만, 없어도 기본적으로 모델이 정상적으로 작동하고 예측을 수행할 수 있습니다.



## ✅ `early\_stopping\_rounds` 없이 사용해도 괜찮은 경우

### 1. 데이터 크기가 작거나 적절히 전처리된 경우:

- 데이터셋이 작아서 과적합 위험이 낮다면 `early\_stopping\_rounds`가 없어도 큰 문제가 없습니다.

### 2. 적절한 하이퍼파라미터 설정을 통해 과적합 방지:

- 트리 개수(`n\_estimators`), 학습률(`learning\_rate`), 트리 깊이(`max\_depth`) 등의 하이퍼파라미터를 잘 조정하면 조기 종료 없이도 과적합을 최소화할 수 있습니다.

### 3. 테스트 데이터로 성능 평가:

- 학습 후 테스트 데이터로 성능을 평가해 과적합 여부를 확인하면 됩니다.

## ⚠️ `early\_stopping\_rounds` 없이 발생할 수 있는 문제

### 1. 과적합 위험:

- 모델이 너무 오래 학습하면 학습 데이터에 과적합될 가능성이 높습니다. 이로 인해 테스트 데이터의 성능이 떨어질 수 있습니다.

### 2. 학습 시간 증가:

- `n\_estimators`(트리 개수)를 크게 설정하면, 불필요한 반복으로 학습 시간이 길어질 수 있습니다.

## 🔧 과적합 방지를 위한 대안

`early\_stopping\_rounds` 없이 과적합을 방지하려면 다음과 같은 방법을 사용할 수 있습니다:

### 1 적절한 `n\_estimators` 설정

트리 개수를 너무 크게 설정하지 마세요. 일반적으로 **100~300** 정도로 시작해 적절히 조정합니다:



```
xgb_model = xgb.XGBRegressor(  
    n_estimators=200, # 적당한 트리 개수  
    learning_rate=0.1,  
    max_depth=6,  
    subsample=0.8,  
    colsample_bytree=0.8,  
    random_state=42  
)
```

## 2 `learning\_rate` 감소

학습률을 낮추면 모델이 천천히 학습하므로 과적합 위험을 줄일 수 있습니다:

```
xgb_model = xgb.XGBRegressor(  
    learning_rate=0.05, # 낮은 학습률  
    n_estimators=300,  
    max_depth=6  
)
```

## 3 교차 검증(Cross-Validation)

XGBoost의 `cv` 메서드를 사용해 모델의 성능을 검증합니다:

```
from xgboost import cv, DMatrix  
  
dtrain = DMatrix(X_train, label=y_train)  
params = {  
    'objective': 'reg:squarederror',  
    'learning_rate': 0.1,  
    'max_depth': 6,  
    'subsample': 0.8,  
    'colsample_bytree': 0.8  
}  
cv_results = cv(  
    params,  
    dtrain,  
    num_boost_round=200,  
    nfold=5, # 5-fold 교차 검증  
    metrics='rmse',  
    as_pandas=True,  
    seed=42  
)  
print(cv_results)
```

## 4 과적합 방지 하이퍼파라미터 설정



- ``max_depth``: 트리 깊이를 제한하여 과적합 방지.
- ``min_child_weight``: 리프 노드의 최소 샘플 수를 제한.
- ``gamma``: 노드 분할을 위한 최소 손실 감소를 설정해 과적합 방지.

예:

```
xgb_model = xgb.XGBRegressor(  
    max_depth=4,          # 트리 깊이 제한  
    min_child_weight=5,   # 리프 노드 최소 샘플  
    gamma=0.2,           # 최소 손실 감소  
    n_estimators=200  
)
```

## 결론

- ``early_stopping_rounds``는 과적합 방지와 학습 시간 단축에 유용하지만, 반드시 필요한 것은 아닙니다.
- 대안으로 적절한 하이퍼파라미터 설정과 교차 검증을 통해 모델 성능을 보장할 수 있습니다.
- 추가 질문이 있다면 언제든지 말씀해주세요! 😊