데이터 분할 (Data Splitting)

- 머신러닝 모델을 구축하고 평가하기 위해, 전체 데이터셋을 두 개 이상의 하위 집합(subset)으로 나누는 과정
- 가장 일반적인 분할 : 훈련 세트(Training Set)와 테스트 세트(Test Set)로 나누는 것
- **훈련 세트 (Training Set)**
  - 모델을 학습시키는 데 사용되는 데이터
  - 모델은 이 데이터의 패턴과 관계를 학습하여 예측 규칙을 생성
- **테스트 세트 (Test Set)**
  - 학습이 완료된 모델의 성능을 평가하는 데 사용되는 데이터
  - 이 데이터는 모델 학습 과정에 전혀 사용되지 않아야 함
  - 모델이 본 적 없는 새로운 데이터에 대해 얼마나 잘 일반화되는지를 측정하는 척도
- **검증 세트 (Validation Set)**
  - 모델의 하이퍼파라미터를 튜닝하거나, 여러 모델 중 최적의 모델을 선택하기 위해 사용되는 데이터
  - 훈련 세트의 일부를 다시 분할하여 생성
  - 교차 검증(Cross-Validation)을 사용하면 별도의 검증 세트 없이 이 과정을 수행 가능

적용 가능한 상황

- 모든 지도 학습(Supervised Learning) 모델링 과정에서 필수적으로 수행
- 모델이 훈련 데이터에만 과적합(Overfitting)되는 것을 방지
- 일반화 성능을 객관적으로 평가하기 위해 반드시 필요

구현 방법

`scikit-learn`의 `train_test_split` 함수를 사용하는 것이 가장 일반적이고 편리한 방법입니다.

주의사항

- **데이터 셔플링 (Shuffling)**
  - `train_test_split`은 기본적으로 데이터를 분할하기 전에 무작위로 섞습니다.
  - 만약 데이터가 특정 순서(e.g. 서열 순)로 정렬되어 있다면, 셔플링을 통해 각 세트에 데이터가 편향되지 않고 고르게 분포되도록 하는 것이 중요합니다.
  - 시계열 데이터와 같이 **순서가 중요한 경우는 `shuffle=False`로 설정**해야 합니다.
    - 이 경우 그냥 `.iloc` 등을 활용하여 최근 데이터를 테스트셋, 나머지 과거 데이터를 훈련용 데이터셋으로 활용할 수도 있습니다.
- **층화 샘플링 (Stratified Sampling)**
  - 분류 문제에서 클래스 비율이 불균형할 경우, `stratify` 옵션을 사용하는 것이 매우 중요합니다.
  - 이 옵션은 원본 데이터의 클래스 비율을 훈련 세트와 테스트 세트 모두에 동일하게 유지시켜 줍니다.

    - 이를 통해 모델이 소수 클래스에 대해서도 충분히 학습하고, 더 신뢰성 있는 평가를 할 수 있습니다.
- `random_state` **설정**
    - `random_state`를 특정 정수 값으로 고정하면, 코드를 실행할 때마다 항상 동일한 방식으로 데이터가 분할됩니다.
    - 이는 분석 결과의 재현성(reproducibility)을 확보하기 위해 필수적입니다.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris

# 1. 데이터 준비
iris = load_iris()
X = iris.data
y = iris.target

# 특성(X)과 타겟(y)을 함께 DataFrame으로 만들어 확인
df = pd.DataFrame(data=np.c_[X, y], columns=iris.feature_names + ['target'])
print("원본 데이터 클래스 비율:\n", df['target'].value_counts(normalize=True))
'''
원본 데이터 클래스 비율:
 2.0    0.333333
1.0    0.333333
0.0    0.333333
Name: target, dtype: float64
'''
# 2. 데이터 분할 (기본)
# train_test_split 하이퍼파라미터
# *arrays: 분할할 배열들 (X, y 등)
# test_size: 테스트 세트의 비율 (0.0 ~ 1.0 사이의 float) 또는 개수 (int). (기본값
=0.25)
# train_size: 훈련 세트의 비율 또는 개수. test_size와 둘 중 하나만 지정하는 것이 일반
적.
# random_state: 재현성을 위한 시드 값.
# shuffle: 분할 전 데이터를 섞을지 여부. (기본값=True)
# stratify: 층화 샘플링을 위한 기준 배열 (보통 y를 지정). (기본값=None)
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.3,
    random_state=42,
    shuffle=True
)

print("\n--- 기본 분할 후 클래스 비율 ---")
print("훈련 세트:\n", pd.Series(y_train).value_counts(normalize=True))
print("테스트 세트:\n", pd.Series(y_test).value_counts(normalize=True))
# 클래스 비율이 원본과 약간 다를 수 있음
'''
훈련 세트:
2    0.352381
1    0.352381
```

```
0    0.295238
dtype: float64
테스트 세트:
0    0.422222
2    0.288889
1    0.288889
dtype: float64
'''

# 3. 층화 샘플링을 사용한 데이터 분할
X_train_strat, X_test_strat, y_train_strat, y_test_strat = train_test_split(
    X, y,
    test_size=0.3,
    random_state=42,
    shuffle=True,
    stratify=y # y(타겟)의 클래스 비율을 유지하도록 지정
)

print("\n--- 층화 분할 후 클래스 비율 ---")
print("훈련 세트:\n", pd.Series(y_train_strat).value_counts(normalize=True))
print("테스트 세트:\n", pd.Series(y_test_strat).value_counts(normalize=True))
# 클래스 비율이 원본과 거의 동일하게 유지됨
'''
훈련 세트:
2    0.333333
1    0.333333
0    0.333333
dtype: float64
테스트 세트:
2    0.333333
1    0.333333
0    0.333333
dtype: float64
'''

print("\n--- 분할된 데이터 shape ---")
print("X_train shape:", X_train.shape)
print("X_test shape:", X_test.shape)
print("y_train shape:", y_train.shape)
print("y_test shape:", y_test.shape)
'''
X_train shape: (105, 4)
X_test shape: (45, 4)
y_train shape: (105,)
y_test shape: (45,)
'''
```

결과 해석 방법

- `train_test_split` 함수는 분할된 배열들을 튜플 형태로 반환합니다. 일반적으로 `X_train, X_test, y_train, y_test` 순서로 변수에 할당받아 사용합니다.
- 분할된 각 세트의 `shape` 속성을 통해 데이터가 의도한 비율대로 잘 나뉘었는지 확인할 수 있습니다.

- 분류 문제의 경우, 각 세트의 클래스 분포(`value_counts(normalize=True)`)를 확인하여 `stratify` 옵션이 잘 적용되었는지 검토하는 것이 좋습니다.

장단점 및 대안

- **장점**:
  - 사용법이 매우 간단하고 직관적입니다.
  - 무작위 분할과 층화 분할을 유연하게 선택할 수 있습니다.
- **단점**:
  - 데이터를 한 번만 분할하기 때문에, 특정 데이터가 우연히 훈련 또는 테스트 세트에 편중될 수 있습니다. 이로 인해 모델 성능 평가가 불안정할 수 있으며, 특히 데이터 양이 적을 때 이 문제가 두드러집니다.
- **대안**:
  - **교차 검증 (Cross-Validation)**: 이 단점을 보완하기 위해 교차 검증을 사용하는 것이 강력히 권장됩니다. 교차 검증은 데이터를 여러 번 다른 방식으로 분할하고 평가하여, 모델 성능에 대한 더 안정적이고 신뢰도 높은 추정치를 제공합니다. (e.g., K-Fold, Stratified K-Fold)