강원지역혁신플랫폼

기계학습

Machine Learning

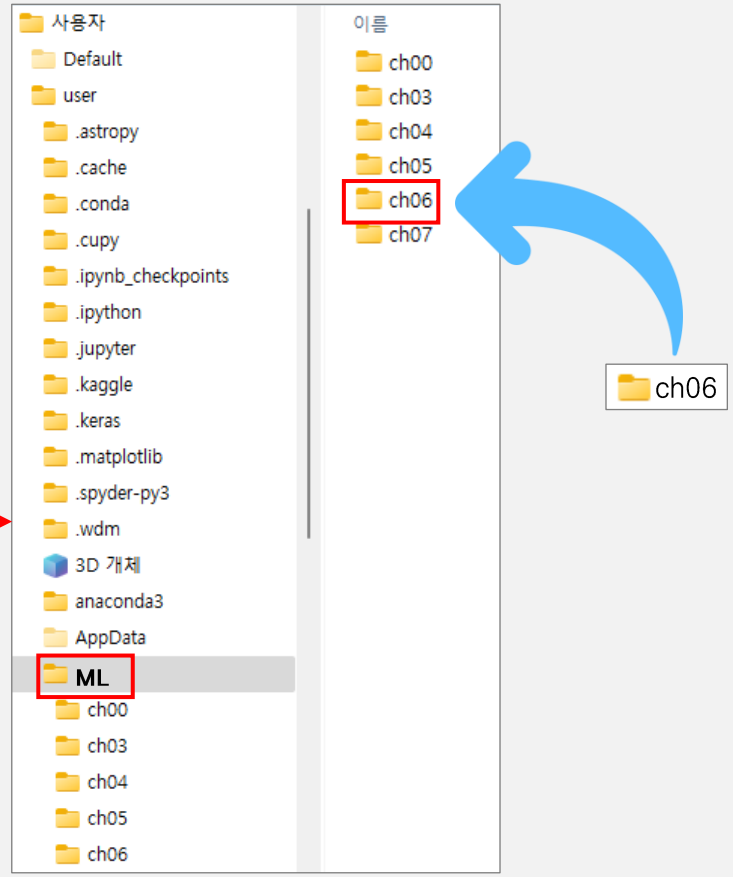
모델 테스트 및 검증

학습목표

- 모델 테스트 및 검증 개념을 이해하고 설명할 수 있습니다.

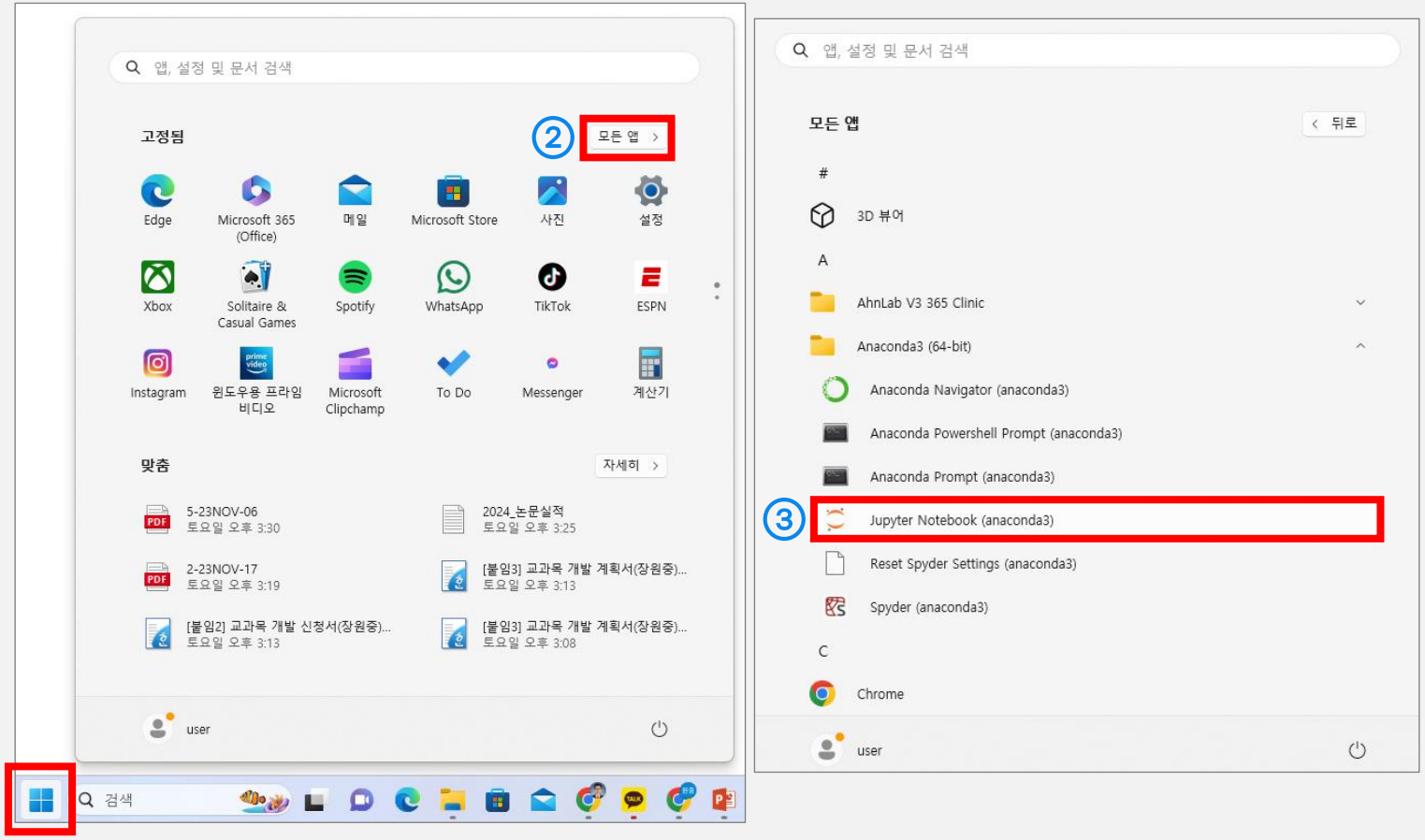
01 | 6주차 실습코드 복사하기

- (권장) 아래와 같은 경로에 실행 소스가 존재하면 환경 구축 완료
  - 6주차 실습코드 다운로드 → 압축해제 → ch06 폴더를 ML 하위 폴더로 복사
  - c:₩Users₩user>ML> 컴퓨터이름 또는 사용자계정



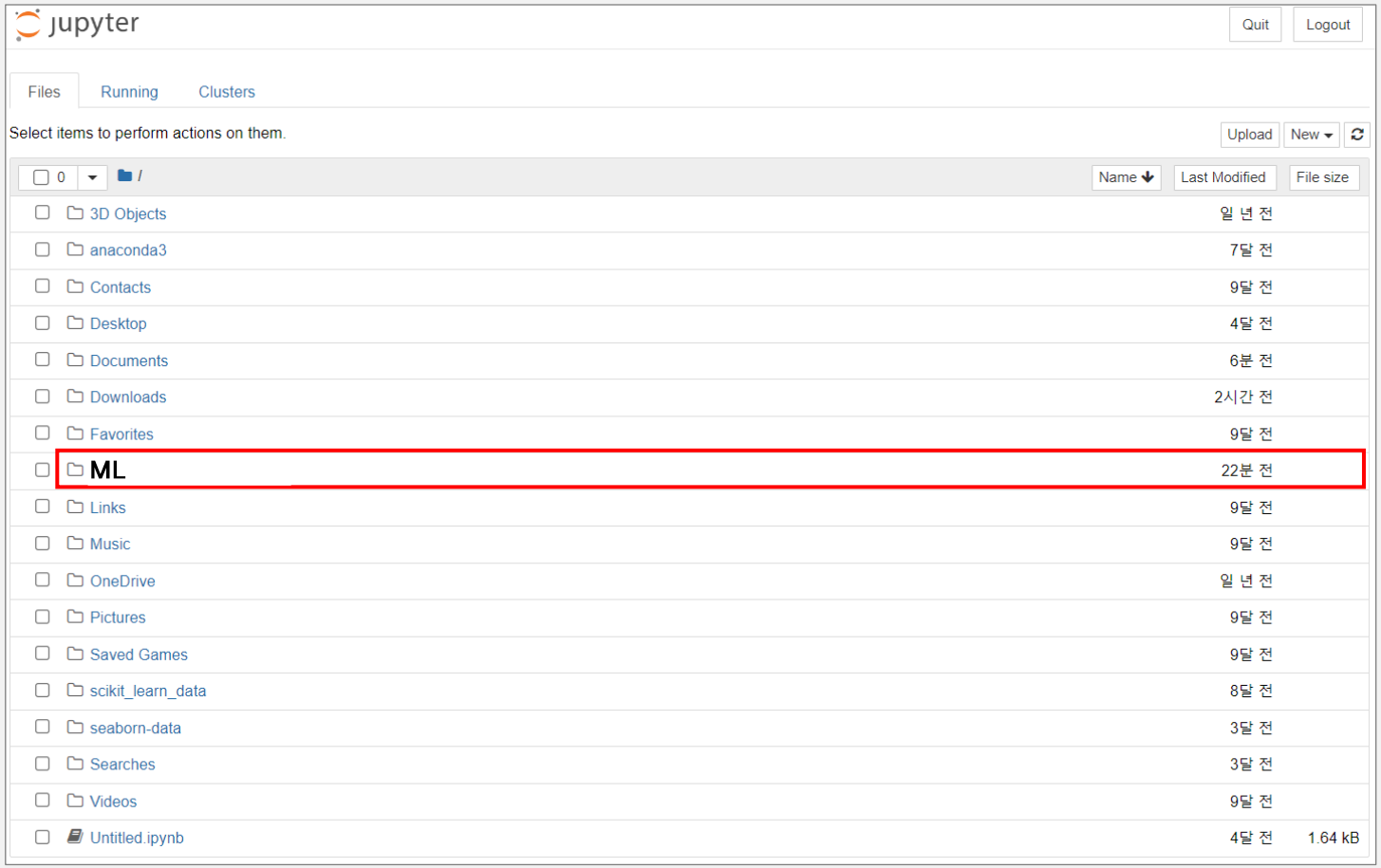
02 | Jupyter Notebook 실행하기

✦ ①시작 메뉴 클릭 > ②모든 앱 버튼 클릭 > ③Anaconda3(64-bit)
> “Jupyter Notebook (anaconda)” 메뉴 클릭하기



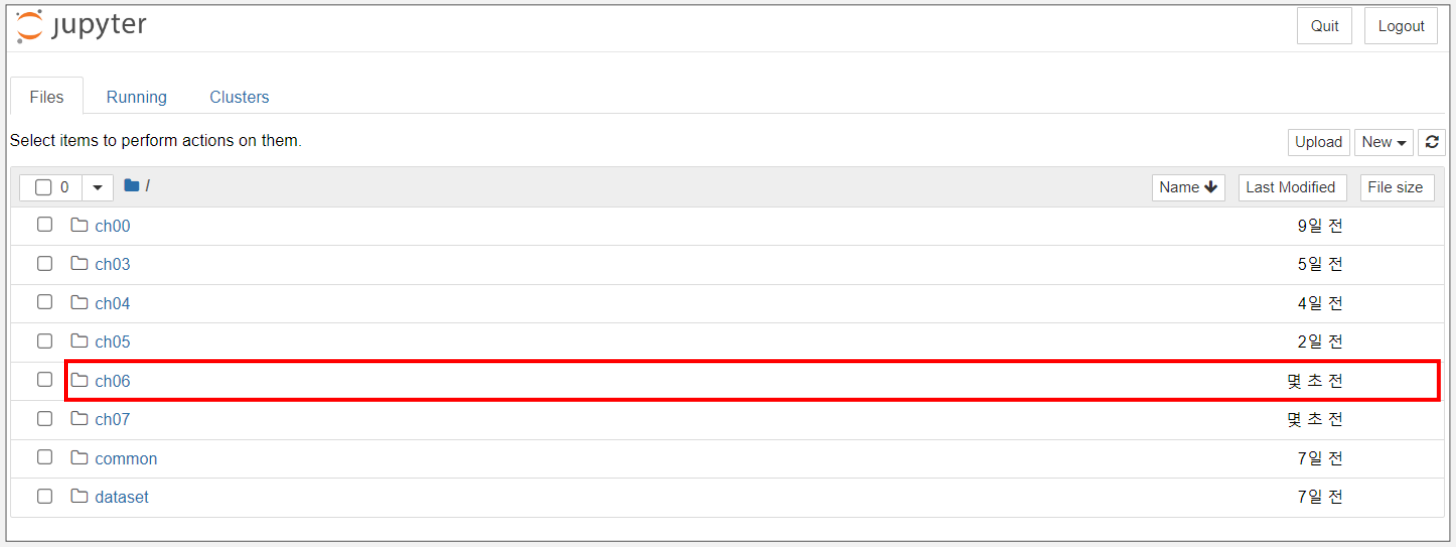
03 | ML 폴더

ML 폴더를 클릭하기



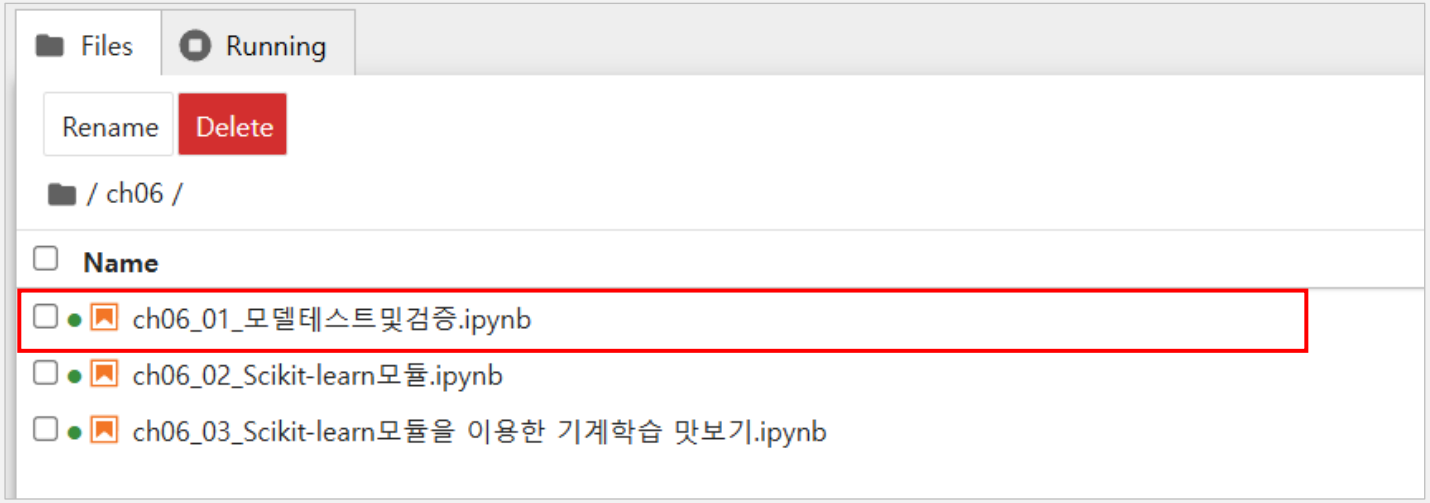
04 | ch06 폴더

ch06 폴더 클릭하기



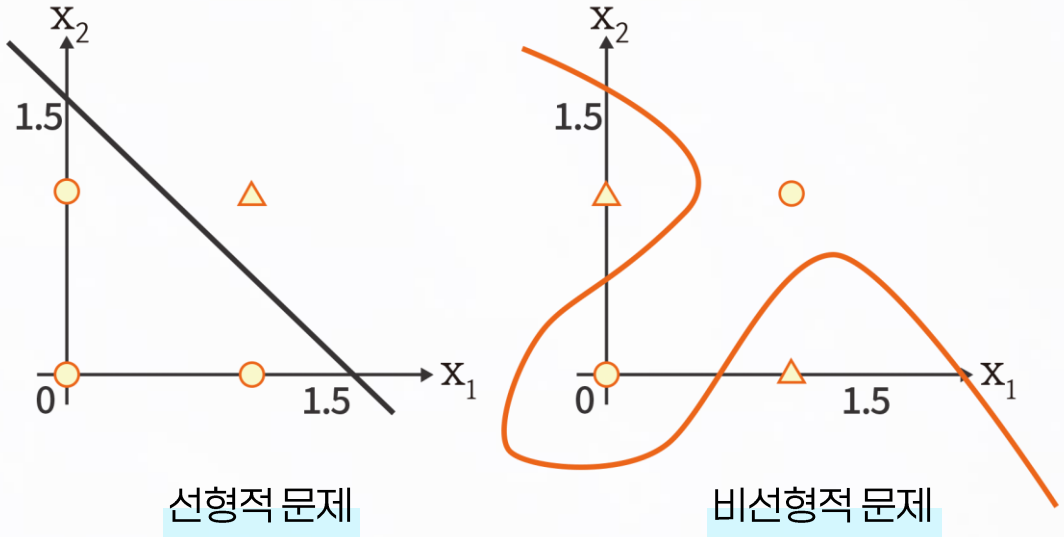
05 | ch06_01_모델테스트및검증.ipynb

✦ ch06_01_모델테스트및검증.ipynb 파일 클릭하기



모델 테스트 및 검증

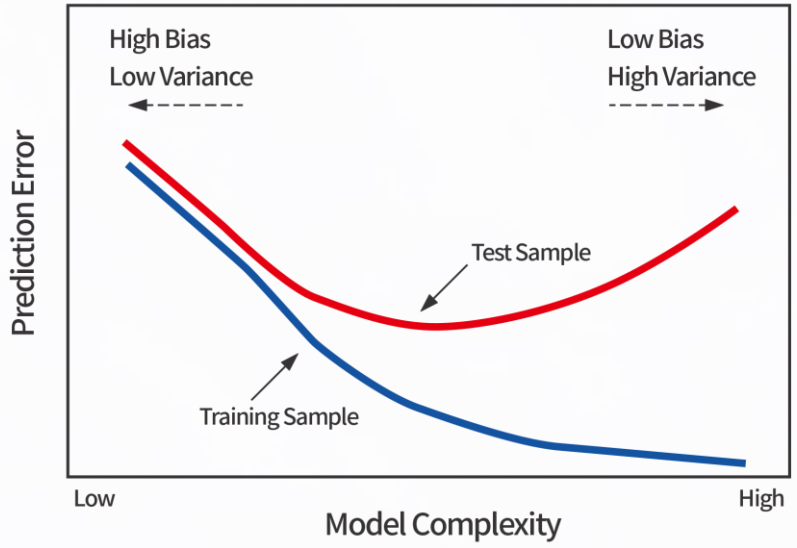
- 모델 테스트와 검증은 모델이 새로운 샘플에 얼마나 잘 일반화될지 아는 방법임
  - 새로운 샘플에 실제로 테스트 하는 방법
    - 실제 서비스에 모델을 넣고 잘 동작하는지 모니터링 하는 것임



선형적 문제 비선형적 문제

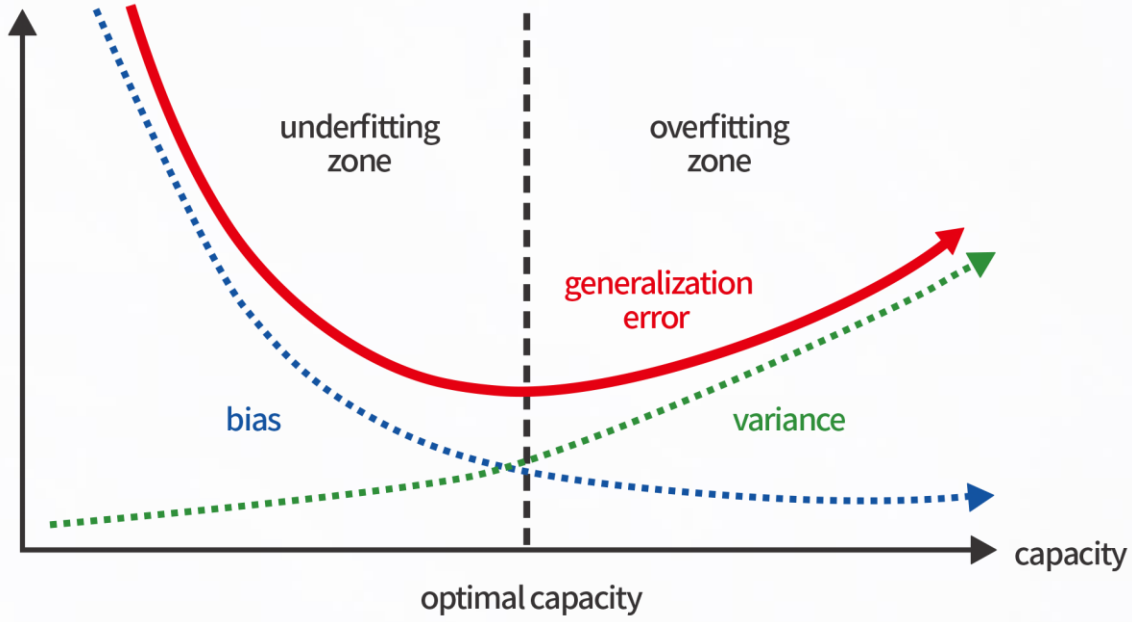
06 | 모델 테스트 및 검증

- 훈련 데이터를 훈련 세트와 테스트 세트로 나누어 테스트 하는 방법
  - 새로운 샘플에 대한 오류 비율을 일반화 오차(Generalization Error)라고 하며, 테스트 세트에서 모델을 평가함으로써 이 오차에 대한 추정값(Estimation)을 얻음
    - 이 값은 새로운 샘플에 모델이 얼마나 잘 작동하는지 알려줌
      - 훈련 오차가 낮고 일반화 오차가 높다면, 이는 모델이 훈련에 의해 과적합 되었음을 뜻한다는 것을 알 수 있음



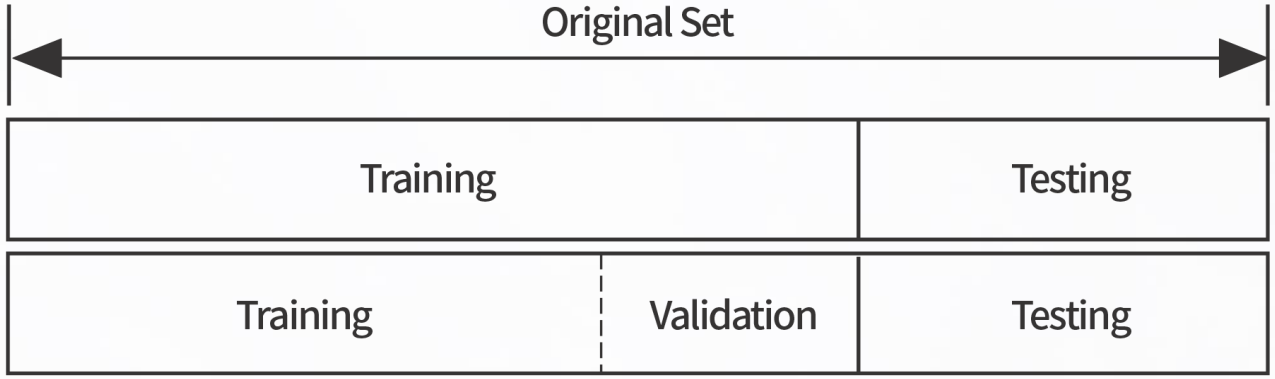
하이퍼파라미터 튜닝과 모델 선택

- 모델을 구축할 때, 훈련 데이터와 테스트 데이터만으로도 훈련의 척도를 판단할 수 있음
  - 하지만, 훈련 데이터에 대한 학습만을 바탕으로 모델의 하이퍼파라미터(Hyperparameter)를 튜닝하게 되면 과대적합(overfitting)이 일어날 가능성이 매우 큼



07 | 하이퍼파라미터 튜닝과 모델 선택

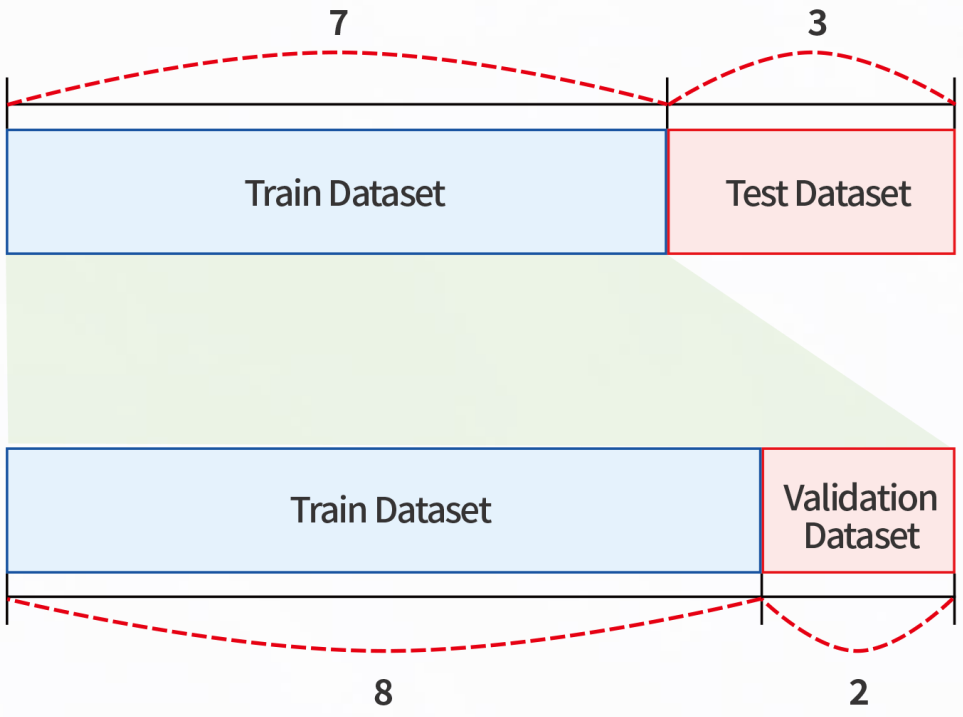
- 또한, 테스트 데이터는 학습에서 모델에 간접적으로라도 영향을 미치면 안 되기 때문에 테스트 데이터로 검증을 해서는 안됨
  - 그래서 검증(validation) 데이터셋을 따로 두어 매 훈련마다 검증 데이터셋에 대해 평가하여 모델을 튜닝함 (하이퍼파라미터 조정)
    - 즉, 훈련 세트, 검증 세트, 테스트 세트로 나눔
      - 검증 세트로 하이퍼파라미터를 조정함



08 | 홀드아웃 검증

홀드아웃 검증(Hold-out validation)

- 기본적인 검증 방법으로 단순히 훈련 데이터와 테스트 데이터로 나눔
  - 나눠진 훈련 데이터에서 다시 검증 데이터셋을 따로 떼어내는 방법임



08 | 홀드아웃 검증

- 이 방식의 문제점은 데이터가 적을 때, 각 데이터셋이 전체 데이터를 통계적으로 대표하지 못할 가능성이 높음
  - 즉, 하나의 데이터셋에 다양한 특징을 지닌 데이터들이 포함되지 않을 수 있다는 것임
    - 이를 확인하는 방법은 새롭게 데이터를 셔플링하여 다시 모델을 학습시켰을 때 모델의 성능이 많이 차이가 나면 이 문제라고 볼 수 있음

09 | 홀드아웃 검증: 실습

홀드아웃 검증: 실습

- 보스턴 주택 가격 데이터(Boston Housing Price Data)
  - 보스턴 시의 주택 가격에 대한 데이터임
    - 주택의 여러 가지 요건들이 가격 정보에 포함되어 있음
    - 주택의 가격에 영향을 미치는 요소를 분석하고자 하는 목적으로 사용될 수 있음
    - 보스턴 주택 데이터는 여러 개의 측정지표를 포함하고 있고 보스턴 인근의 주택 가격의 중앙값(median value)임
    - 회귀분석 등으로 주택 가격 예측에 활용될 수 있음

09 | 홀드아웃 검증: 실습

보스턴 주택 가격 데이터는 14개 속성으로 구성되어 있고, 속성에 대한 설명은 아래와 같음

- 관측치 수는 506개임

변수	변수 설명	변수	변수 설명
CRIM	자치 시(town) 별 1인당 범죄율	DIS	5개의 보스턴 직업센터까지의 접근성 지수
ZN	25,000 평방미터를 초과하는 거주지역의 비율	RAD	방사형 도로까지의 접근성 지수
INDUS	비소매 상업지역이 점유하고 있는 토지의 비율	TAX	10,000 달러 당 재산세율
CHAS	찰스강에 대한 더미변수 (강의 경계에 위치한 경우 1, 아니면 0)	PTRATIO	자치 시(town)별 학생/교사 비율
NOX	10ppm 당 농축 일산화질소	B	1000(Bk-0.62)^2, 여기서 Bk는 자치 시별 흑인의 비율을 의미
RM	주택 1가구당 평균 방의 개수	LSTAT	모집단의 하위계층의 비율(%)
AGE	1940년 이전에 건축된 소유주택의 비율	MEDV	본인 소유의 주택가격(중앙값) (단위:$1,000)

09 | 홀드아웃 검증: 실습

다음은 보스턴 주택 가격 데이터(Boston Housing Price Data)셋을 읽어오는 코드이다.

- 아래와 같이 데이터 형상이 (506, 14)인 것을 알 수 있음

```
# UCI 머신러닝 리포지토리에서 데이터셋 불러오기
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data"
column_names = [ "CRIM", "ZN", "INDUS", "CHAS", "NOX", "RM", "AGE", "DIS", "RAD",
   "TAX", "PTRATIO", "B", "LSTAT", "MEDV" ]
df = pd.read_csv(url, delim_whitespace=True, names=column_names)
print(df.shape)  # (506, 14)
df
```

```
(506, 14)
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222.0	18.7	396.90	5.33	36.2
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
501	0.06263	0.0	11.93	0	0.573	6.593	69.1	2.4786	1	273.0	21.0	391.99	9.67	22.4
502	0.04527	0.0	11.93	0	0.573	6.120	76.7	2.2875	1	273.0	21.0	396.90	9.08	20.6
503	0.06076	0.0	11.93	0	0.573	6.976	91.0	2.1675	1	273.0	21.0	396.90	5.64	23.9
504	0.10959	0.0	11.93	0	0.573	6.794	89.3	2.3889	1	273.0	21.0	393.45	6.48	22.0
505	0.04741	0.0	11.93	0	0.573	6.030	80.8	2.5050	1	273.0	21.0	396.90	7.88	11.9

506 rows × 14 columns

09 | 홀드아웃 검증: 실습

- 다음은 보스턴 주택 가격 데이터를 **훈련 데이터**(70%)와 **테스트 데이터**(30%) **분리**하는 코드이다.
  - 아래와 같이 훈련 데이터(354개)와 테스트 데이터(152개)가 **7:3 비율**로 **잘 분리된 것**을 볼 수 있음
    - 여기서는 훈련 데이터와 테스트 데이터에 **독립 변수**와 **종속 변수**가 **모두 포함**되어 있음

```
# 훈련 데이터 70%, 테스트 데이터 30% 분할
df_train, df_test = train_test_split(df, test_size=0.3, random_state=0)
df_train.shape, df_test.shape        # ((354, 14), (152, 14))
```

09 | 홀드아웃 검증: 실습

- 다음은 보스턴 주택 가격 데이터를 **훈련 데이터**(70%)와 **테스트 데이터**(30%) **분리**하는 코드이다.
  - 아래와 같이 훈련 데이터(354개)와 테스트 데이터(152개)가 **7:3 비율**로 **잘 분리된 것**을 볼 수 있음
    - 여기서는 훈련 데이터와 테스트 데이터에 **독립 변수**(13개)와 **종속 변수**(1개)가 **나누어져** 있음

```
dfX_train, dfX_test, dfy_train, dfy_test = train_test_split(df.iloc[:, 0:13], df.loc[:,'MEDV'],
                                                            test_size=0.3, random_state=0)
dfX_train.shape, dfy_train.shape, dfX_test.shape, dfy_test.shape
# ((354, 13), (354,), (152, 13), (152,))
```

09 | 홀드아웃 검증: 실습

다음은 회귀 분석 모델로 보스턴 주택 가격 데이터셋에서 홀드아웃 검증을 수행하는 코드이다.

- 회귀분석의 결정계수(R^2)는 훈련 데이터로 학습하는 경우 0.764, 테스트 데이터로 평가한 경우 0.673인 것을 알 수 있음

```
feature_names = [ "CRIM", "ZN", "INDUS", "CHAS", "NOX", "RM", "AGE", "DIS", "RAD",
                  "TAX", "PTRATIO", "B", "LSTAT"]

# 선형 회귀분석
model = sm.OLS.from_formula("MEDV ~ " + "+".join(feature_names), data=df_train)
result = model.fit()

pred = result.predict(df_test)
rss = ((df_test.MEDV - pred) ** 2).sum()
tss = ((df_test.MEDV - df_test.MEDV.mean())** 2).sum()
rsquared = 1 - rss / tss

print("학습 R2 = {:.8f}, 검증 R2 = {:.8f}".format(result.rsquared, rsquared))
# 학습 R2 = 0.76454510, 검증 R2 = 0.67338255
```

10 | K-겹 교차 검증

K-겹 교차 검증(K-fold cross-validation)

- K-겹 교차 검증은 홀드아웃에 비해 훈련 세트의 분할에 덜 민감한 성능 추정을 얻을 수 있음
  - 중복을 허락하지 않고 훈련 데이터 셋을 K개의 폴더로 랜덤하게 나눈 다음
    - K-1개의 폴더로 모델을 훈련하는 용도로 사용함
    - 나머지 하나의 폴더로 성능을 평가하는 용도로 사용함

Train	Train	Train	Validation
Train	Train	Validation	Train
Train	Validation	Train	Train
Validation	Train	Train	Train

K-Fold

10 | K-겹 교차 검증

- 아래 그림의 경우 4번(K=4)반복하여 4개의 모델과 성능 추정을 얻음
  - 만족할만한 성능이 나온 하이퍼파라미터를 찾은 후에는 전체 훈련 세트를 사용하여 모델을 다시 훈련함
    - 그리고, 독립적인 테스트 세트를 이용하여 최종 성능 추정을 수행함
    - 모델의 총 검증 점수(validation score)는 각 훈련의 검증 점수의 평균임

Train	Train	Train	Validation
Train	Train	Validation	Train
Train	Validation	Train	Train
Validation	Train	Train	Train

K-Fold

10 | K-겹 교차 검증

K-겹 교차 검증을 사용하는 경우를 생각해 보자.

- 주로 회귀 문제에 활용함
  - 회귀의 결정값은 이산값 형태의 레이블이 아니라 연속된 숫자값이기 때문에 결정값 별로 분포를 정하는 의미가 없기 때문임
  - K-겹 교차 검증은 학습 데이터 세트와 검증 데이터 세트를 점진적으로 변경하면서 마지막 K번째까지 학습과 검증을 수행하는 것임

Train	Train	Train	Validation
Train	Train	Validation	Train
Train	Validation	Train	Train
Validation	Train	Train	Train

K-Fold

10 | K-겹 교차 검증

데이터를 나눌 시 주의점은 다음과 같음

1. 대표성
   - 훈련 데이터 셋과 테스트 데이터 셋은 전체 데이터에 대한 대표성을 띄고 있어야 함
2. 시간의 방향
   - 과거 데이터로부터 미래 데이터를 예측하고자 할 경우에는 데이터를 섞어서는 안 됨
   - 이런 문제는 훈련 데이터 셋에 있는 데이터보다 테스트 데이터 셋의 모든 데이터가 미래의 것이어야 함
3. 데이터 중복
   - 각 훈련, 검증, 테스트 데이터 셋에는 데이터 포인트의 중복이 있어서는 안 됨
   - 데이터가 중복되면 올바른 평가를 할 수 없기 때문임

10 | K-겹 교차 검증

다음은 선형 회귀 모델로 K-5 교차 검증을 수행하는 코드이다.

- 여기서는 결정계수를 직접 계산함
  - 실행 결과에서 결정계수(R^2)의 평균은 0.708인 것을 볼 수 있음

```
scores = np.zeros(5)
cv = KFold(5, shuffle=True, random_state=0)

for i, (idx_train, idx_test) in enumerate(cv.split(df)):
    df_train = df.iloc[idx_train]
    df_test = df.iloc[idx_test]

    # 선형 회귀분석
    model = sm.OLS.from_formula("MEDV ~ " + "+".join(feature_names), data=df_train)
    result = model.fit()

    pred = result.predict(df_test)
    rss = ((df_test.MEDV - pred) ** 2).sum()
    tss = ((df_test.MEDV - df_test.MEDV.mean())** 2).sum()
    rsquared = 1 - rss / tss
    scores[i] = rsquared
    print("학습 R2 = {:.8f}, 검증 R2 = {:.8f}".format(result.rsquared, rsquared))
#학습 R2 = 0.77301356, 검증 R2 = 0.58922238
#학습 R2 = 0.72917058, 검증 R2 = 0.77799144
#학습 R2 = 0.74897081, 검증 R2 = 0.66791979
#학습 R2 = 0.75658611, 검증 R2 = 0.66801630
#학습 R2 = 0.70497483, 검증 R2 = 0.83953317
print("검증 R2 평균 = {:.8f}".format(scores.mean())) # 검증 R2 평균 = 0.70853662
```

10 | K-겹 교차 검증

- 다음은 선형 회귀 모델로 K-5 교차 검증을 수행하는 코드이다.
  - 여기서는 결정계수를 r2_score() 함수로 계산함
    - 실행 결과에서 결정계수(R^2)의 평균은 0.708인 것을 볼 수 있음

```
scores = np.zeros(5)
cv = KFold(5, shuffle=True, random_state=0)

for i, (idx_train, idx_test) in enumerate(cv.split(df)):
    df_train = df.iloc[idx_train]
    df_test = df.iloc[idx_test]

    model = sm.OLS.from_formula("MEDV ~ " + "+".join(feature_names),
data=df_train)
    result = model.fit()

    pred = result.predict(df_test)
    rsquared = r2_score(df_test.MEDV, pred)
    scores[i] = rsquared              # 검증 결과 차례로 저장

print(scores)        # [0.58922238 0.77799144 0.66791979 0.6680163
0.83953317]
print("검증 R2 평균 = {:.8f}".format(scores.mean()))  # 검증 R2 평균 = 0.70853662
```

10 | K-겹 교차 검증

다음은 선형 회귀 모델로 K-5 교차 검증을 수행하는 코드이다.

- 여기서는 결정계수를 cross_val_score()함수로 계산함
  - 실행 결과에서 결정계수(R^2)의 평균은 0.708인 것을 볼 수 있음

```
class StatsmodelsOLS(BaseEstimator, RegressorMixin):
  def __init__(self, formula):
    self.formula = formula
    self.model = None
    self.data = None
    self.result = None

  def fit(self, dfX, dfy):
    self.data = pd.concat([dfX, dfy], axis=1)
    self.model = smf.ols(self.formula, data=self.data)
    self.result = self.model.fit()

  def predict(self, new_data):
    return self.result.predict(new_data)

model = StatsmodelsOLS("MEDV ~ " + "+".join(boston.feature_names))
cv = KFold(5, shuffle=True, random_state=0)
scores = cross_val_score(model, df.iloc[:, 0:13], df.loc[:,'MEDV'], scoring="r2", cv=cv)

print(scores)          # [0.58922238 0.77799144 0.66791979 0.6680163  0.83953317]
print("검증 R2 평균 = {:.8f}".format(scores.mean())) # 검증 R2 평균 = 0.70853662
```

11 | 계층별 K-폴더 검증

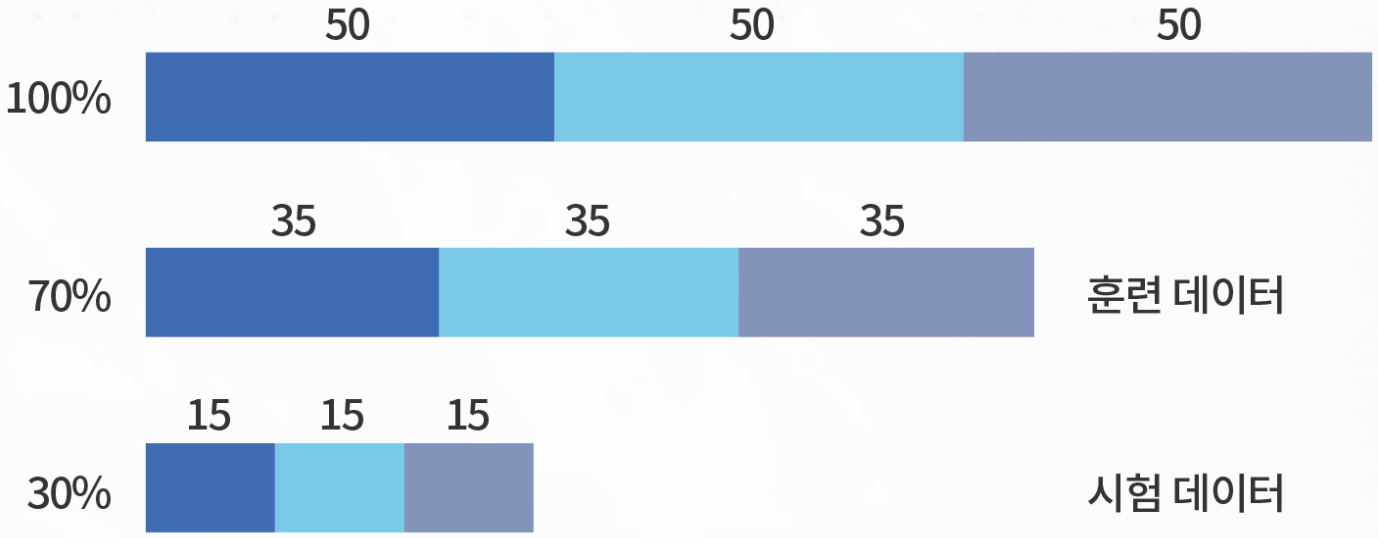
계층별(Stratified) K-폴더 검증

- 훈련 데이터 세트와 검증 데이터 세트를 무작위로 나누기 때문에 홀드아웃(hold-out)할 때 타겟 클래스가 일정하지 않을 수도 있음
  - 이 경우, 훈련 데이터 세트와 검증 데이터 세트의 데이터 분포가 달라지므로 학습에도 영향을 미침
    - 기계학습은 학습 데이터의 분포와 현실 세계 데이터의 분포가 동일하다는 전제가 있음
      - 기본 전제가 무너지면 학습 모델의 성능이 떨어지게 되는 것임

11 | 계층별 K-폴더 검증

훈련 데이터 세트와 검증 데이터 세트의 타겟 클래스 비율을 일정하게 나누어주는 것을 계층별(stratified)기법이라고 함

✦ 아래의 그림처럼 계층별 기법은 타겟 클래스의 비율을 7:3으로 유지하는 것을 볼 수 있음



계층별 K-폴더 기법은 타겟 클래스의 비율을 일정하게 유지함

11 | 계층별 K-폴더 검증

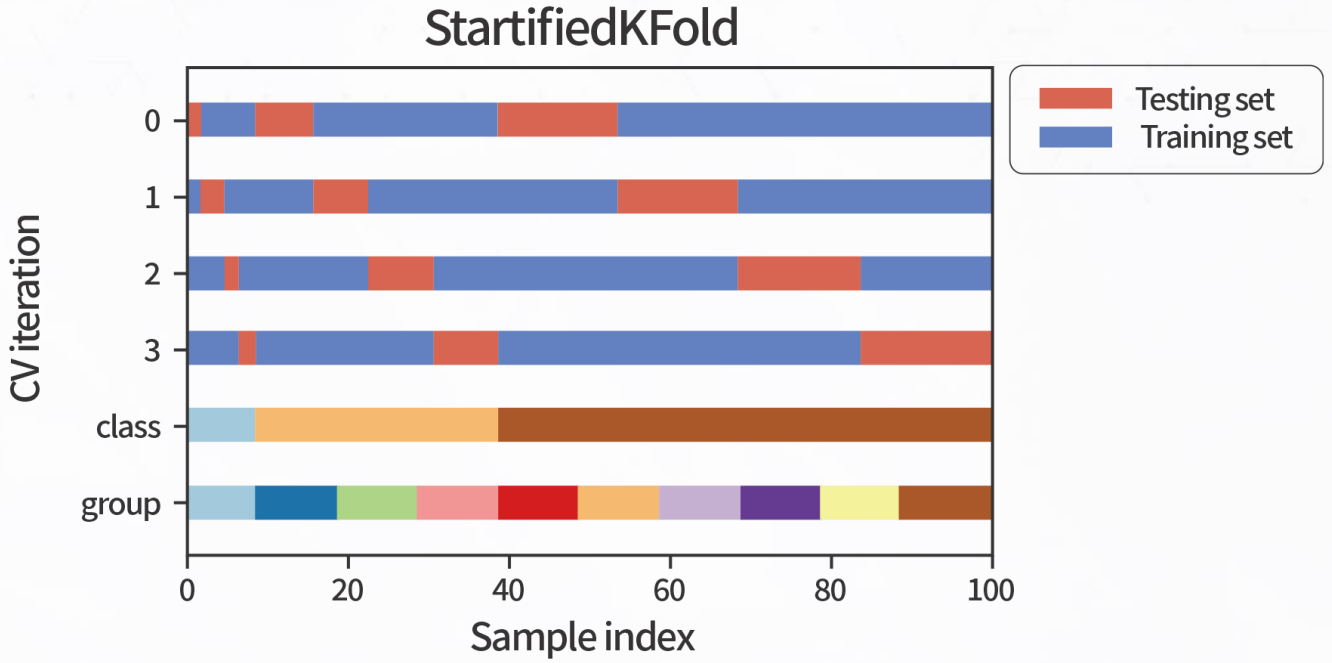
계층별 K-폴더를 사용하는 경우를 생각해 보자.

- 레이블 데이터가 왜곡된 경우 반드시 사용함
  - 즉, 특정 레이블 값이 특이하게 많거나, 매우 적어서 값의 분포가 한쪽으로 치우치는 경우에 주로 사용함

- 일반적으로 분류에서의 교차 검증을 하는 경우 사용함

11 | 계층별 K-폴더 검증

- 계층별 K-폴더는 **불균형한 분포도**를 가진 **레이블 데이터 집합**을 **위한 KFold 방식**임
  - 아래 그림과 같이 **레이블**이 **불균형한 분포도**인 경우 **주로 사용**함



11 | 계층별 K-폴더 검증

아이리스(IRIS, 붓꽃) 데이터 설명

- 아이리스 데이터 셋은 꽃잎의 각 부분의 너비와 길이 등을 측정한 데이터임
  - 관측치는 150개, 속성은 6개로 구성되어 있음
  - 아이리스 꽃은 아래 그림과 같음

열이름	설명
Caseno	일련번호
Sepal Length	꽃받침의 길이 정보
Sepal Width	꽃받침의 너비 정보
Petal Length	꽃잎의 길이 정보
Petal Width	꽃잎의 너비 정보
Species	꽃의 종류 정보 (setosa, versicolor,virginica)

11 | 계층별 K-폴더 검증

다음은 아이리스 데이터셋을 읽어오는 코드이다.

- 아래와 같이 데이터 형상이 (150, 5)인 것을 알 수 있음

```
data = load_iris()
iris_feature = pd.DataFrame(data=data.data,
columns=data.feature_names)
iris_target = pd.Series(data.target, dtype="category")
iris_target = iris_target.cat.rename_categories(data.target_names)
iris = pd.concat([iris_feature, iris_target], axis=1)
print(iris.shape)
Iris
```

```
(150, 5)
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	0
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	virginica
146	6.3	2.5	5.0	1.9	virginica
147	6.5	3.0	5.2	2.0	virginica
148	6.2	3.4	5.4	2.3	virginica
149	5.9	3.0	5.1	1.8	virginica

150 rows × 5 columns

11 | 계층별 K-폴더 검증

다음은 아이리스 데이터셋의 속성명을 변경하는 코드이다.

아래와 같이 독립 변수와 종속 변수의 속성명이 변경된 것을 볼 수 있음

```
iris.rename({"sepal length (cm)":"sepal_length", "sepal width (cm)":
            "sepal_width", "petal length (cm)":"petal_length",
            "petal width (cm)":"petal_width", 0:"species"}, axis=1, inplace=True)
iris
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	virginica
146	6.3	2.5	5.0	1.9	virginica
147	6.5	3.0	5.2	2.0	virginica
148	6.2	3.4	5.4	2.3	virginica
149	5.9	3.0	5.1	1.8	virginica

150 rows × 5 columns

11 | 계층별 K-폴더 검증

다음은 아이리스 데이터셋을 층을 고려하지 않고 훈련 데이터와 시험 데이터로 7:3 비율로 분리하는 코드이다.

실행결과 종속 변수의 각 레이블 개수가 7:3 비율로 나누어 지지 않은 것을 볼 수 있음

```
X_train, X_test, y_train, y_test = train_test_split(iris.iloc[:, :-1], iris.iloc[:, -1],
                                                    test_size=0.3, shuffle=True,
                                                    random_state=42)
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
print(y_train.value_counts())
print(y_test.value_counts())
```

```
(105, 4) (45, 4) (105,) (45,)
versicolor    37
virginica     37
setosa        31
Name: species, dtype: int64
setosa        19
versicolor    13
virginica     13
Name: species, dtype: int64
```

11 | 계층별 K-폴더 검증

- 다음은 아이리스 데이터셋을 층을 고려하여 훈련 데이터와 시험 데이터로 7:3 비율로 분리하는 코드이다.
  - 실행결과 종속 변수의 각 레이블 개수가 7:3 비율로 정확하게 나누어진 것을 볼 수 있음

```
X_train, X_test, y_train, y_test = train_test_split(iris.iloc[:, :-1], iris.iloc[:, -1],
                                                    test_size=0.3, shuffle=True,
                                                    stratify=iris['species'],
                                                    random_state=42)
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
print(y_train.value_counts())
print(y_test.value_counts())
```

```
setosa        35
versicolor    35
virginica     35
Name: species, dtype: int64
setosa        15
versicolor    15
virginica     15
Name: species, dtype: int64
```

11 | 계층별 K-폴더 검증

다음은 10-겹 KFold를 사용한 아이리스 데이터셋을 분리하는 코드이다.

- 아래와 같이 4번째 데이터 분리에서 검증 레이블 데이터 분포를 보면, virginica=0, versicolor=10, setosa=5로 분리된 것을 볼 수 있음

```
kfold = KFold(n_splits=10)
n_iter = 0

for train_index, test_index in kfold.split(iris):
    n_iter += 1
    label_train = iris['species'].iloc[train_index]
    label_test = iris['species'].iloc[test_index]
    print('## 교차검증: {0}'.format(n_iter))
    print('학습 레이블 데이터 분포: \n', count_frequency(label_train))
    print('검증 레이블 데이터 분포: \n', count_frequency(label_test))
    print('-----------------------------------------------------')
```

```
## 교차검증: 4
학습 레이블 데이터 분포:
 [('virginica', 50), ('setosa', 45), ('versicolor', 40)]
검증 레이블 데이터 분포:
 [('versicolor', 10), ('setosa', 5)]
```

11 | 계층별 K-폴더 검증

- 다음은 10-겹 StratifiedKFold를 사용한 아이리스 데이터셋을 분리하는 코드이다.
  - 아래와 같이 모든 데이터 분리에서 검증 레이블 데이터 분포를 보면, virginica=5, versicolor=5, setosa=5로 분리된 것을 볼 수 있음

```
skf = StratifiedKFold(n_splits=10)
n_iter = 0

for train_index, test_index in skf.split(iris, iris['species']):
    n_iter += 1
    label_train = iris['species'].iloc[train_index]
    label_test = iris['species'].iloc[test_index]
    print('## 교차검증: {0}'.format(n_iter))
    print('학습 레이블 데이터 분포: \n', count_frequency(label_train))
    print('검증 레이블 데이터 분포: \n', count_frequency(label_test))
    print('-----------------------------------------------------')
```

```
## 교차검증: 10
학습 레이블 데이터 분포:
 [('setosa', 45), ('versicolor', 45), ('virginica', 45)]
검증 레이블 데이터 분포:
 [('setosa', 5), ('versicolor', 5), ('virginica', 5)]
```

11 | 계층별 K-폴더 검증

다음은 의사결정나무 모델로 10-겹 KFold 검증을 수행하는 코드이다.

- 실행결과 평균 검증 정확도는 약 95%인 것을 볼 수 있음

```
cv = KFold(n_splits=10)  # K=10
cv_accuracy=[]          # KFold 별 정확도 저장
n_iter = 0              # 반복횟수
for train_index, test_index in cv.split(iris):
   X_train = iris_feature.iloc[train_index]
   X_test = iris_feature.iloc[test_index]
   y_train = iris['species'].iloc[train_index]
   y_test = iris['species'].iloc[test_index]
   model.fit(X_train, y_train)              # 학습 및 예측
   pred = model.predict(X_test)
   n_iter += 1                              # 반복횟수
   label_train = iris['species'].iloc[train_index]
   label_test = iris['species'].iloc[test_index]
   print("n_iter=",n_iter,"\n",count_frequency(label_train), count_frequency(label_test))
   accuracy = np.round(accuracy_score(y_test, pred), 4)
   train_size = X_train.shape[0]
   test_size = X_test.shape[0]
   print('교차 검증 정확도 :{1}, 학습 데이터 크기: {2}, 검증 데이터 크기: {3}'.format(n_iter, accuracy,train_size,test_size))
   print('검증 세트 인덱스 :{1}'.format(n_iter, test_index))
   print('-----------------------------------------------------------')
   cv_accuracy.append(accuracy)
print('\n## 평균 검증 정확도:', np.mean(cv_accuracy))  # 평균 검증 정확도: 0.95333
```

```
(135,) (15,)
n_iter= 10
 [('setosa', 50), ('versicolor', 50), ('virginica', 35)] [('virginica', 15)]
교차 검증 정확도 :1.0, 학습 데이터 크기: 135, 검증 데이터 크기: 15
검증 세트 인덱스 :[135 136 137 138 139 140 141 142 143 144 145 146 147 148 149]
-----------------------------------------------------------

## 평균 검증 정확도: 0.95333
```

11 | 계층별 K-폴더 검증

다음은 의사결정나무 모델로 10-겹 Stratified KFold 검증을 수행하는 코드이다.

실행결과 평균 검증 정확도는 약 96%인 것을 볼 수 있음

```
skf = StratifiedKFold(n_splits=10)
cv_accuracy=[]                  # KFold 별 정확도 저장
n_iter = 0                      # 반복횟수
for train_index, test_index in skf.split(iris.iloc[:,:-1],iris['species']):
    X_train = iris_feature.iloc[train_index]
    X_test = iris_feature.iloc[test_index]
    y_train = iris['species'].iloc[train_index]
    y_test = iris['species'].iloc[test_index]
    model.fit(X_train, y_train)
    pred = model.predict(X_test)
    n_iter += 1                 # 반복횟수
    label_train = iris['species'].iloc[train_index]
    label_test = iris['species'].iloc[test_index]
    print("n_iter=",n_iter,"\n",count_frequency(label_train), count_frequency(label_test))
    accuracy = np.round(accuracy_score(y_test, pred), 4)
    train_size = X_train.shape[0]
    test_size = X_test.shape[0]
    print('교차 검증 정확도 :{1}, 학습 데이터 크기: {2}, 검증 데이터 크기: {3}'.format(n_iter,accuracy,train_size,test_size))
    print('검증 세트 인덱스 :{1}'.format(n_iter, test_index))
    print('-----------------------------------------------------------')
    cv_accuracy.append(accuracy)
print('\n## 평균 검증 정확도:', np.mean(cv_accuracy))  # 평균 검증 정확도: 0.95999
```

```
(135,) (15,)
n_iter= 10
 [('setosa', 45), ('versicolor', 45), ('virginica', 45)] [('setosa', 5), ('versicolor', 5), ('virginica', 5)]
교차 검증 정확도 :1.0, 학습 데이터 크기: 135, 검증 데이터 크기: 15
검증 세트 인덱스 :[ 45  46  47  48  49  95  96  97  98  99 145 146 147 148 149]
-----------------------------------------------------------
검증 정확도
 [1.0, 0.9333, 1.0, 0.9333, 0.9333, 0.8667, 0.9333, 1.0, 1.0, 1.0]

## 평균 검증 정확도: 0.95999
```