머신러닝

트리 알고리즘
앙상블 모델

로지스틱 회귀로 와인 분류

➢ 데이터 불러오기, 확인

```
import pandas as pd

wine = pd.read_csv('https://bit.ly/wine-date')
```

```
wine.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6497 entries, 0 to 6496
Data columns (total 4 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   alcohol  6497 non-null   float64
 1   sugar    6497 non-null   float64
 2   pH       6497 non-null   float64
 3   class    6497 non-null   float64
dtypes: float64(4)
memory usage: 203.2 KB
```

```
wine.describe()
```

	alcohol	sugar	pH	class
count	6497.000000	6497.000000	6497.000000	6497.000000
mean	10.491801	5.443235	3.218501	0.753886
std	1.192712	4.757804	0.160787	0.430779
min	8.000000	0.600000	2.720000	0.000000
25%	9.500000	1.800000	3.110000	1.000000
50%	10.300000	3.000000	3.210000	1.000000
75%	11.300000	8.100000	3.320000	1.000000
max	14.900000	65.800000	4.010000	1.000000

로지스틱 회귀로 와인 분류

- 학습데이터 나누기

```
data = wine[['alcohol', 'sugar', 'pH']].to_numpy()
target = wine['class'].to_numpy()
```

```
from sklearn.model_selection import train_test_split

train_input, test_input, train_target, test_target = train_test_split(
    data, target, test_size=0.2, random_state=42)
```

```
print(train_input.shape, test_input.shape)
```

```
(5197, 3) (1300, 3)
```

로지스틱 회귀로 와인 분류

- 데이터 전처리(표준화)

```
from sklearn.preprocessing import StandardScaler

ss = StandardScaler()
ss.fit(train_input)

train_scaled = ss.transform(train_input)
test_scaled = ss.transform(test_input)
```

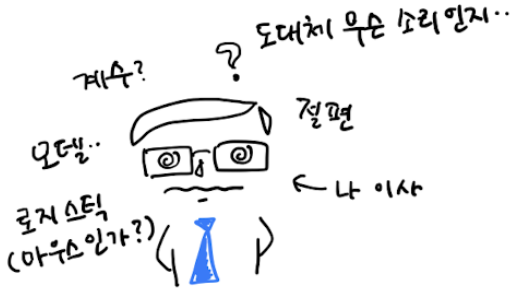
로지스틱 회귀로 와인 분류

➢ LogisticRegression 모델 적용

```
from sklearn.linear_model import LogisticRegression

lr = LogisticRegression()
lr.fit(train_scaled, train_target)

print(lr.score(train_scaled, train_target))
0.7808350971714451
print(lr.score(test_scaled, test_target))
0.7776923076923077

print(lr.coef_, lr.intercept_)
[[ 0.51270274  1.6733911  -0.68767781]] [1.81777902]
```

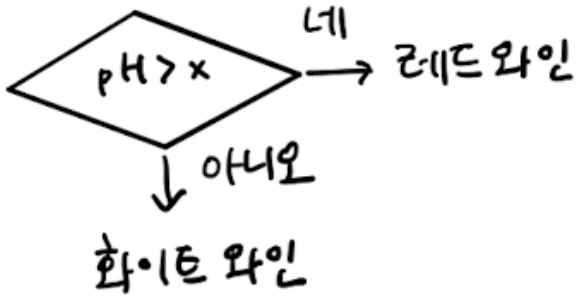
➢ 보고서 작성

0.51270274*알코올 도수 + 1.6733911 * 당도 + -0.68767781*pH + 1.81777902 계산 결과가 0보다 크면 화이트와인, 작으면 레드화인입니다. 분류 정확도는 약 77%입니다.

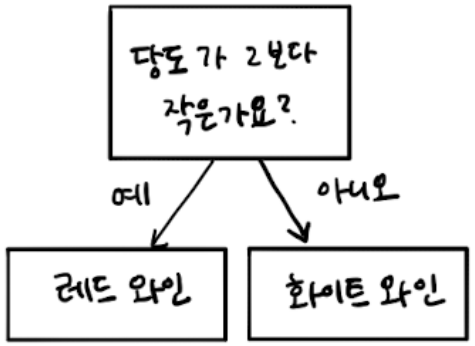


로지스틱 회귀로 와인 분류

- 의사 결정에 도움되는 보고서 만들기



- 결정 트리 알고리즘
  - 데이터의 특성을 기반으로 트리 구조의 예측 모델을 만든다.
  - 데이터를 잘 나눌 수 있는 질문을 찾아 분류해 나가는 방법



결정트리(DecisionTree) 알고리즘

➢ 결정 트리의 주요 개념

1. 노드 (Node): 트리의 각 분기점. 노드는 내부 노드와 리프 노드로 구분

1. **내부 노드 (Internal Node)**: 자식 노드를 가지고 있으며, 데이터를 분할하는 기준.
2. **리프 노드 (Leaf Node)**: 최종 출력 값을 가지며, 분류 문제에서는 클래스 레이블을, 회귀 문제에서는 실수를 출력.

2. 루트 노드 (Root Node): 트리의 최상위 노드로, 전체 데이터의 분할이 시작되는 곳.

3. 분할 (Split): 노드를 두 개 이상의 하위 노드로 나누는 과정. 각 분할은 데이터의 특정 특성을 기준으로 이루어진다.

4. 가지치기 (Pruning): 트리의 과적합을 방지하기 위해 불필요하거나 덜 중요한 분할을 제거하는 과정.

➢ 결정 트리의 장점과 단점

장점 – 사람이 쉽게 이해하고 해석할 수 있다.

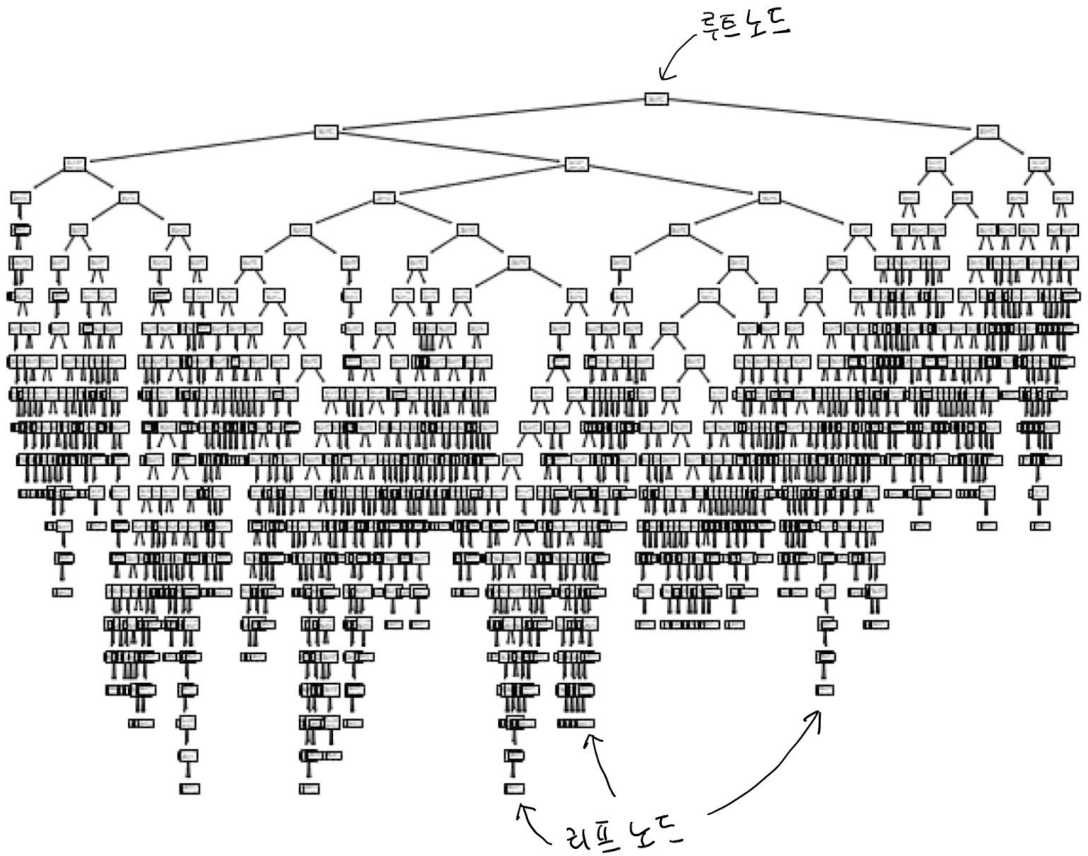
- 다른 알고리즘에 비해 특성 스케일링이나 정규화가 덜 요구됨
- 범주형 데이터와 연속형 데이터를 모두 처리 가능

단점 - 트리가 너무 복잡해지면 학습 데이터에 과적합될 수 있다.

- 작은 데이터 변화에도 트리 구조가 크게 변할 수 있다.
- 매우 큰 데이터셋에서는 트리의 깊이가 깊어져 학습 속도가 느려질 수 있다.

결정트리(DecisionTree) 알고리즘

```
from sklearn.tree import DecisionTreeClassifier

dt = DecisionTreeClassifier(random_state=42)
dt.fit(train_scaled, train_target)

print(dt.score(train_scaled, train_target))
0.996921300750433
print(dt.score(test_scaled, test_target))
0.8592307692307692


import matplotlib.pyplot as plt
from sklearn.tree import plot_tree

plt.figure(figsize=(10,7))
plot_tree(dt)
plt.show()
```

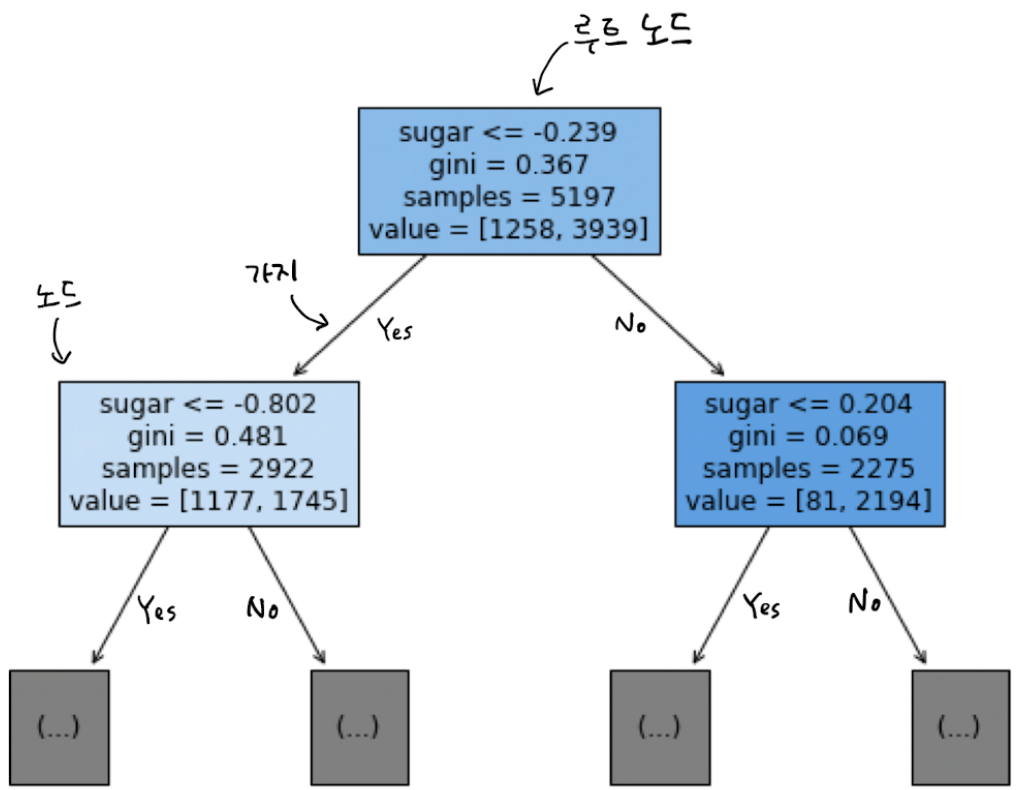


결정트리(DecisionTree) 알고리즘

➢ 결정트리 시각화 함수 plot_tree() 매개변수

1. max_depth=1
   루트노드를 제외하고 하나의 노드를 더 확장하여 그림
2. filled=True
   클래스에 맞게 노드 색상 부여
3. feature_names=['alcohol', 'sugar', 'pH']
   특성 이름 전달

```
plt.figure(figsize=(10,7))
plot_tree(dt, max_depth=1, filled=True,
          feature_names=['alcohol', 'sugar', 'pH'])
plt.show()
```



결정트리(DecisionTree) 알고리즘

➢ 지니 불순도 (Gini Impurity) - **데이터 분할** 기준

- 노드의 혼합도를 측정하여 노드에 속한 데이터가 얼마나 잘 분리되었는지를 나타냄
- 노드가 순수할수록(하나의 클래스에 속할수록) 지니 불순도는 낮아지며, 여러 클래스가 섞여 있을수록 지니 불순도는 높아진다.

$$\text{Gini}(D) = 1 - \sum_{k=1}^{C}(p_k)^2$$

D: 현재 노드의 데이터 집합
C: 클래스 개수
p_k: 클래스 k에 속한 데이터 포인트의 비율

➢ 두 클래스 A와 B가 있는 데이터 집합의 지니 불순도를 계산 예시

1) 노드에 있는 데이터 중 클래스 A가 60%, 클래스 B가 40% 일 때,

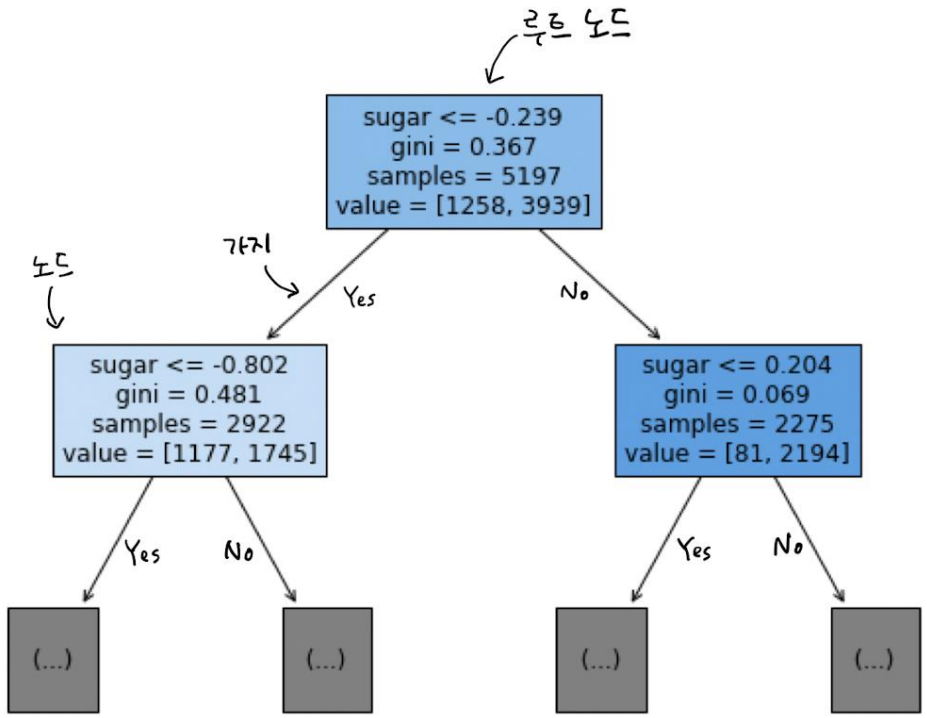
$$\text{Gini} = 1 - (0.6^2 + 0.4^2) = 1 - (0.36 + 0.16) = 1 - 0.52 = 0.48$$

2) 노드에 있는 데이터 중 클래스 A가 100%, 클래스 B가 0% 일 때,(노드가 완전히 순수함)

$$\text{Gini} = 1 - (1.0^2) = 0$$

결정트리(DecisionTree) 알고리즘

➢ 지니 불순도 계산 예



$$\text{Gini}(D) = 1 - \sum_{k=1}^{C}(p_k)^2$$

지니불순도 = 1 - (음성클래스 비율^2 + 양성클래스 비율^2)

$1 - \left(\left(\frac{1258}{5197}\right)^2 + \left(\frac{3939}{5197}\right)^2\right) = 0.367$

$1 - \left(\left(\frac{50}{100}\right)^2 + \left(\frac{50}{100}\right)^2\right) = 0.5$

$1 - \left(\left(\frac{0}{100}\right)^2 + \left(\frac{100}{100}\right)^2\right) = 0$

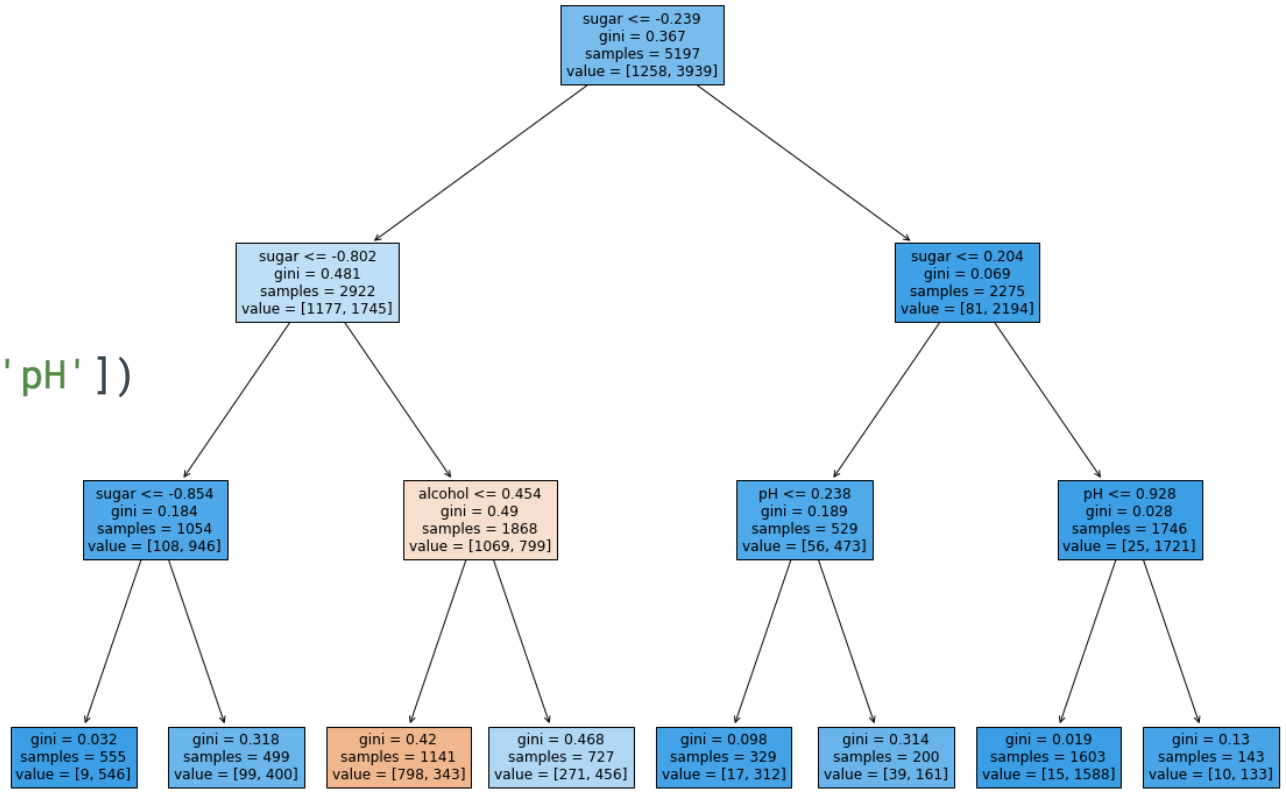
결정트리모델은 (부모노드와 자식노드의 불순도 차이)가 가능한 크도록 트리를 성장시킨다.

$$\text{부모의 불순도} - \frac{\text{왼쪽 노드의 샘플수}}{\text{부모의 샘플수}} \times \text{왼쪽노드의 불순도} - \frac{\text{오른쪽 노드의 샘플수}}{\text{부모의 샘플수}} \times \text{오른쪽 노드의 불순도}$$

결정트리(DecisionTree) 알고리즘

- 가지치기
  - 트리의 과적합을 방지 방법의 하나, 트리 최대 깊이(max_depth, 하이퍼파라미터) 지정

```
dt = DecisionTreeClassifier(max_depth=3, random_state=42)
dt.fit(train_scaled, train_target)

print(dt.score(train_scaled, train_target))
0.8454877814123533
print(dt.score(test_scaled, test_target))
0.8415384615384616

plt.figure(figsize=(20,15))
plot_tree(dt, filled=True,
          feature_names=['alcohol', 'sugar', 'pH'])
plt.show()
```



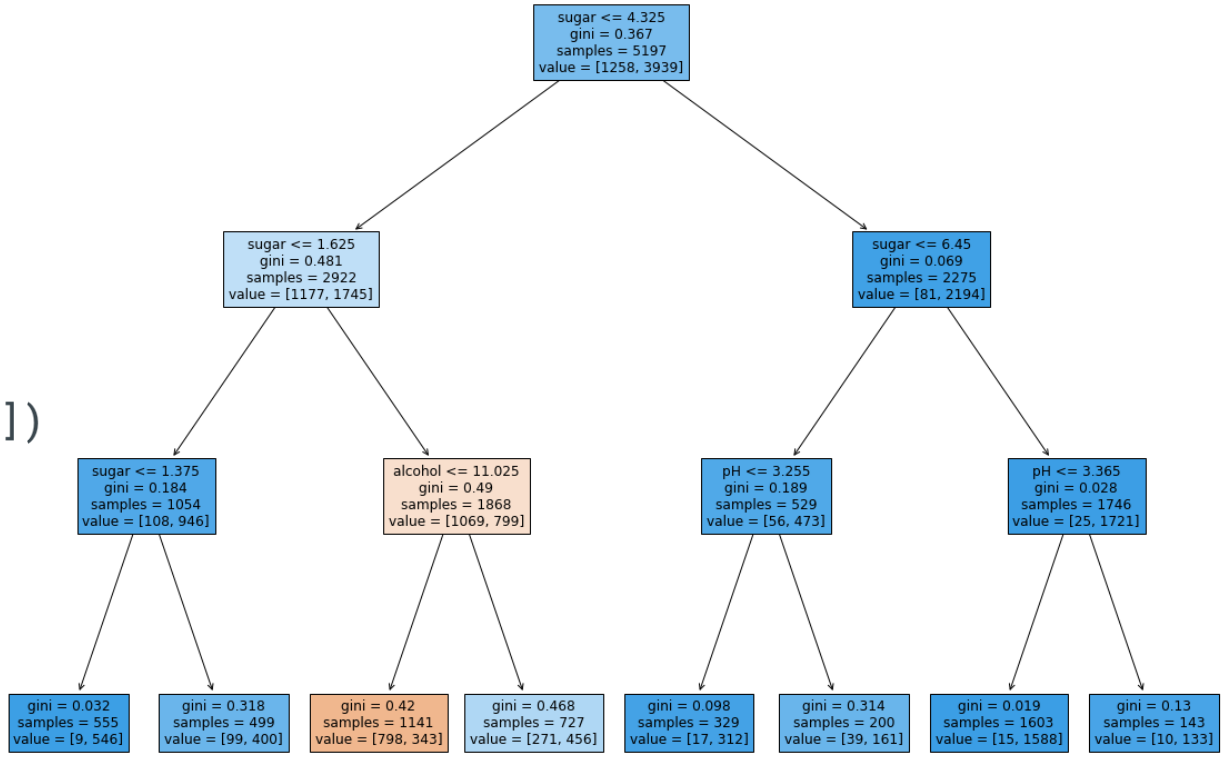
결정트리(DecisionTree) 알고리즘

➢ 결정트리모델을 생성하는 과정에서 훈련데이터 값의 표준화(전처리)는 필요한가?

```
dt = DecisionTreeClassifier(max_depth=3, random_state=42)
dt.fit(train_input, train_target)

print(dt.score(train_input, train_target))
0.8454877814123533
print(dt.score(test_input, test_target))
0.8415384615384616

plt.figure(figsize=(20,15))
plot_tree(dt, filled=True,
          feature_names=['alcohol', 'sugar', 'pH'])
plt.show()
```

```
print(dt.feature_importances_)
[0.12345626 0.86862934 0.0079144 ]
```

** 결정트리의 특성 중요도로 특성 선택 가능



머신러닝모델 성능 검증

➢ 머신러닝 모델 검증 방법

1. 데이터 분할:
   - 데이터를 훈련 데이터(training data)와 검증 데이터(validation data), 테스트 데이터 (test data)로 나눔.
   - 일반적으로 60-70%는 학습 세트로, 10-20%는 검증 세트로, 나머지 10~20% 정도는 테스트 세트로 사용

2. 교차 검증:
   - 훈련 데이터의 검증 부분을 여러 부분으로 나누어 모델을 훈련하고 검증.
   예를 들어, k-폴드 교차 검증에서는 데이터를 k개의 폴드(fold)로 나누어, 각 폴드를 한 번씩 검증 데이터로 사용하고 나머지를 훈련 데이터로 사용하여 k번 모델을 훈련 및 평가.

3. 평가 지표 계산:
   - 모델의 성능을 평가하기 위해 정확도(accuracy), 정밀도(precision), 재현율(recall), F1 스코어(F1 score), AUC-ROC(Area Under the Receiver Operating Characteristic curve) 등 다양한 지표 사용.

4. 테스트 데이터 평가(모델의 일반화 성능 평가):
   - 모델의 최종 성능을 확인하기 위해 테스트 데이터 사용.
   - 모델이 한 번도 본 적 없는 데이터를 테스트 데이터로 한다.

와인 결정트리모델 검증

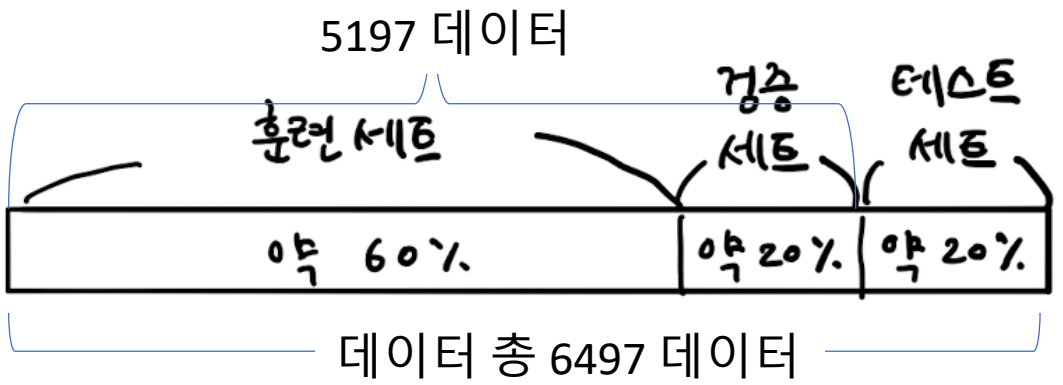
➢ 검증 세트 만들기

```
from sklearn.model_selection import train_test_split

train_input, test_input, train_target, test_target = train_test_split(
    data, target, test_size=0.2, random_state=42)

sub_input, val_input, sub_target, val_target = train_test_split(
    train_input, train_target, test_size=0.2, random_state=42)
```

➢ 훈련세트를 train_test_split() 입력으로 하여 훈련세트와 검증 세트로 분할한다.

```
print(sub_input.shape, val_input.shape)
```

```
(4157, 3) (1040, 3)
```



와인 결정트리모델 검증

- 검증 세트로 모델 확인

```
from sklearn.tree import DecisionTreeClassifier

dt = DecisionTreeClassifier(random_state=42)
dt.fit(sub_input, sub_target)

print(dt.score(sub_input, sub_target))
print(dt.score(val_input, val_target))
```

```
0.9971133028626413
0.864423076923077
```

- 검증세트로 모델 검증 결과 과대적합됨

머신러닝모델 성능 검증(교차검증)

- 교차 검증(cross-validation)은 데이터 과적합(overfitting)을 방지하고 모델의 일반화 성능을 평가하기 위해 사용.
- 즉, 모델이 새로운 데이터에서도 잘 작동할 수 있도록 교차 검증 수행함.
- k-폴드 교차 검증의 예시

- 훈련 데이터셋이 1200개의 샘플에 대하여 k=3인 3-폴드 교차 검증 수행한다면,

1.데이터를 3개의 폴드로 나눈다 (각 폴드는 400개의 샘플로 구성).

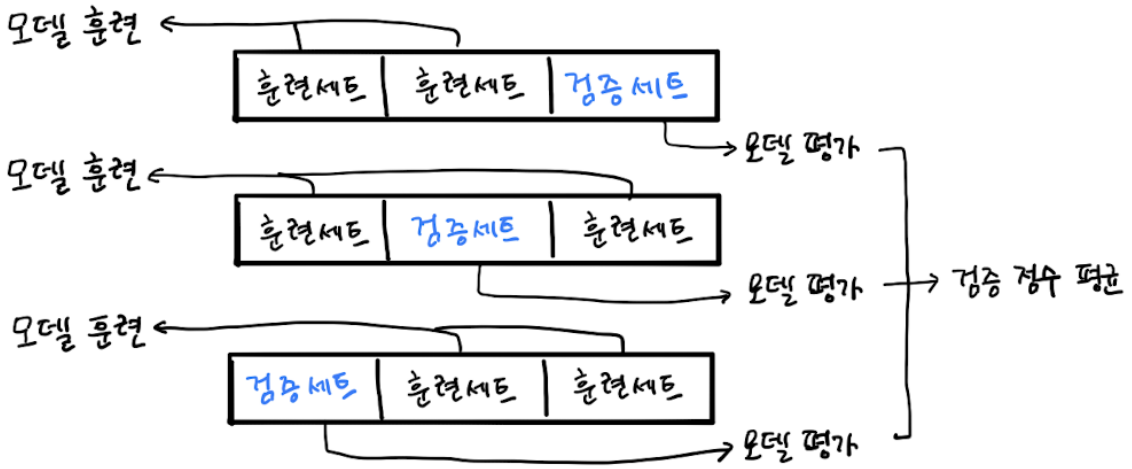
2. 첫 번째 폴드를 검증 데이터로, 나머지 2개의 폴드를 훈련 데이터로 사용해 모델을 훈련

3. 훈련된 모델을 첫 번째 폴드로 평가

4. 두 번째 폴드를 검증 데이터로, 나머지 폴드를 훈련 데이터로 사용해 모델을 다시 훈련하고 평가

5. 이 과정을 3번 반복한다.

6. 3번의 평가 결과를 평균 내어 최종 성능을 산출한다.



머신러닝모델 성능 검증(교차검증)

➢ Scikit-Learn의 교차검증 지원하는 함수: cross_validate()

from sklearn.model_selection import cross_validate

cross_validate(**estimator**, X, y=None, *, scoring=None, **cv=None**, n_jobs=None, return_train_score=False, return_estimator=False, error_score=nan)

- **estimator**: 평가할 모델(예: LinearRegression(), DecisionTreeClassifier() 등).
- X: 특징 행렬(입력 데이터).
- y: 타겟 벡터(레이블 데이터).
- scoring: 평가 지표 혹은 지표들의 리스트(예: accuracy, roc_auc 등).
- **cv**: 교차 검증 분할 전략. 기본값은 5로, 5-폴드 교차 검증을 의미, KFold(), StratifiedFold() 적용
- n_jobs: 병렬 처리에 사용할 CPU 코어 수. -1로 설정하면 모든 가용 코어를 사용합니다.
- return_train_score: 훈련 세트에 대한 점수를 반환할지 여부. 기본값은 False입니다.
- return_estimator: 각 분할의 훈련된 모델을 반환할지 여부. 기본값은 False입니다.
- error_score: 계산 중 에러가 발생했을 때의 점수. 기본값은 nan입니다.

머신러닝모델 성능 검증(교차 검증)

➢ cross_validate()로 모델 교차 검증

```
from sklearn.model_selection import cross_validate

scores = cross_validate(dt, train_input, train_target)
```

1. train_input, train_target 데이터를 5개의 폴드로 나눈다
2. 첫 번째 폴드를 검증 데이터로, 나머지 4개의 폴드를 훈련 데이터로 사용해 모델 훈련
3. 모델을 첫 번째 폴드로 평가
4. 두 번째 폴드를 검증 데이터로, 나머지 폴드를 훈련 데이터로 사용해 모델 훈련하고 평가
5. 이 과정을 5번 반복

```
print(scores)
{'fit_time': array([0.00725031, 0.00697041, 0.00710249, 0.00712824, 0.00681305]),
 'score_time': array([0.00077963, 0.00055647, 0.0005784 , 0.00052595, 0.00059152]),
 'test_score': array([0.86923077, 0.84615385, 0.87680462, 0.84889317, 0.83541867])}
```

검증폴드 점수

```
import numpy as np

print(np.mean(scores['test_score']))
0.855300214703487
```

6. 5번의 평가 결과를 평균으로 최종 성능 산출

머신러닝모델 성능 검증(교차 검증)

- cross_validate() 함수의 cv 매개변수
  - 교차 검증 분할 전략.
  - **cv=5**
    - 기본값, KFold(n_splits=5)와 동일한 의미
    - 데이터를 5개의 폴드로 나누고, 각 폴드를 한 번씩 검증 데이터로 사용
    - 데이터는 기본적으로 섞이지 않는다. (shuffle=False).
  - cv=KFold(n_splits=5, shuffle=True, random_state=42)
    - 데이터 섞기(shuffle) 및 무작위 시드(random_state)를 설정
  - cv=StratifiedKFold(n_splits=5, shuffle=True)
    - 분류 문제에 주로 사용.
    - 분류 문제에서는 종종 클래스 불균형을 포함함
    - 예) wine 데이터의 경우 red wine(0)의 개수는 1599, white wine(1) 개수는 4898로 불균형함
    - StratifiedKFold는 각 폴드에 대해 전체 데이터셋의 클래스 분포를 유지하므로, 모델이 모든 폴드에서 균형 잡힌 학습 및 평가를 할 수 있다.

머신러닝모델 성능 검증(교차 검증)

- cross_validate() 함수는 폴드 구성시 훈련세트를 섞지 않음 -> 분할기로 훈련세트 섞은 후 사용하는 것이 바람직함.

- 분할기를 사용한 교차 검증

```
from sklearn.model_selection import StratifiedKFold

scores = cross_validate(dt, train_input, train_target, cv=StratifiedKFold())
print(np.mean(scores['test_score']))
0.855300214703487

splitter = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
scores = cross_validate(dt, train_input, train_target, cv=splitter)
print(np.mean(scores['test_score']))
0.8574181117533719
```

하이퍼파라미터 튜닝

- 모델이 학습하는 파라미터: 모델파라미터
- 사용자 지정 파라미터: 하이퍼파라미터

예) DecisionTreeClassifier 클래스의 경우, max_depth, min_samples_split, min_samples_leaf 등의 매개변수
DecisionTreeClassifier(criterion='entropy', max_depth=5, min_samples_split=4, min_samples_leaf=2)

- 최적의 하이퍼파라미터를 찾을 수 있도록 Scikit-Learn에서 지원하는 클래스

▪ 그리드 서치 (Grid Search)

- 하이퍼파라미터의 가능한 모든 조합을 시도하여 최적의 조합을 찾는 방법

- GridSearchCV 클래스

  ▪ 하이퍼파라미터 탐색과 교차검증을 동시에 수행함

```
param_grid = { 'criterion': ['gini', 'entropy'],
               'max_depth': [None, 10, 20, 30],
               'min_samples_split': [2, 5, 10],
               'min_samples_leaf': [1, 2, 4],
               'max_features': [None, 'sqrt', 'log2'] }

clf = DecisionTreeClassifier(random_state=42)
grid_search = GridSearchCV(estimator=clf, param_grid=param_grid, cv=5, n_jobs=-1)
```

최적의 하이퍼파라미터 찾기: 그리드 서치

```
from sklearn.model_selection import GridSearchCV

params = {'min_impurity_decrease': [0.0001, 0.0002, 0.0003, 0.0004, 0.0005]}

gs = GridSearchCV(DecisionTreeClassifier(random_state=42), params, n_jobs=-1)
gs.fit(train_input, train_target)

dt = gs.best_estimator_    # 검증 점수가 가장 높은 모델 저장된 속성
print(dt.score(train_input, train_target))
0.9615162593804117

print(gs.best_params_)  # 검증 점수가 가장 높은 매개변수-값 저장
{'min_impurity_decrease': 0.0001}

print(gs.cv_results_['mean_test_score'])   # 5번의 교차검증으로 얻은 점수
[0.86819297 0.86453617 0.86492226 0.86780891 0.86761605]
```

0.0001 일 때 교차검증 점수

- gs.fit() 메소드에서 훈련되는 모델 수: 25개
- min_impurity_decrease 매개값 5개
- 교차검증 cv=5 기본값 사용

최적의 하이퍼파라미터 찾기

- 다음의 매개변수 조합으로 훈련되는 모델 개수는?

```
params = {'min_impurity_decrease': np.arange(0.0001, 0.001, 0.0001),
          'max_depth': range(5, 20, 1),
          'min_samples_split': range(2, 100, 10)
          }
```

- 최상의 매개변수 조합은?

{ ‘max_depth’: 14, ‘min_impurity_decrease’: 0.0004, ‘min_samples_split’: 12 }

- 최상의 교차검증점수?

{ ‘max_depth’: 14, ‘min_impurity_decrease’: 0.0004, ‘min_samples_split’: 12 }

랜덤서치

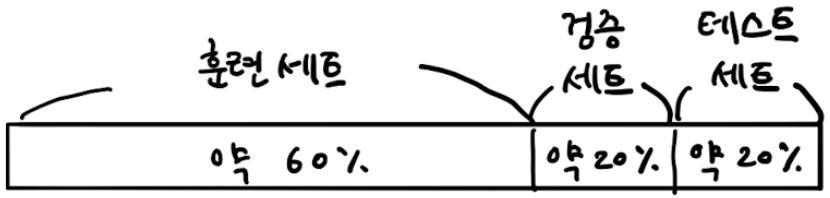
- 그리드 서치는 최적의 하이퍼파라미터 조합을 찾기 위한 수행 시간 오래 걸림
- 랜덤서치(RandomSearch)는 매개변수 조건이 많거나, 매개변수 값이 연속된 수치로 범위나 간격 등을 지정하기 어려울 경우 사용
  - 랜덤서치는 매개변수를 샘플링할 수 있는 **확률분포객체**를 전달받아 수행함

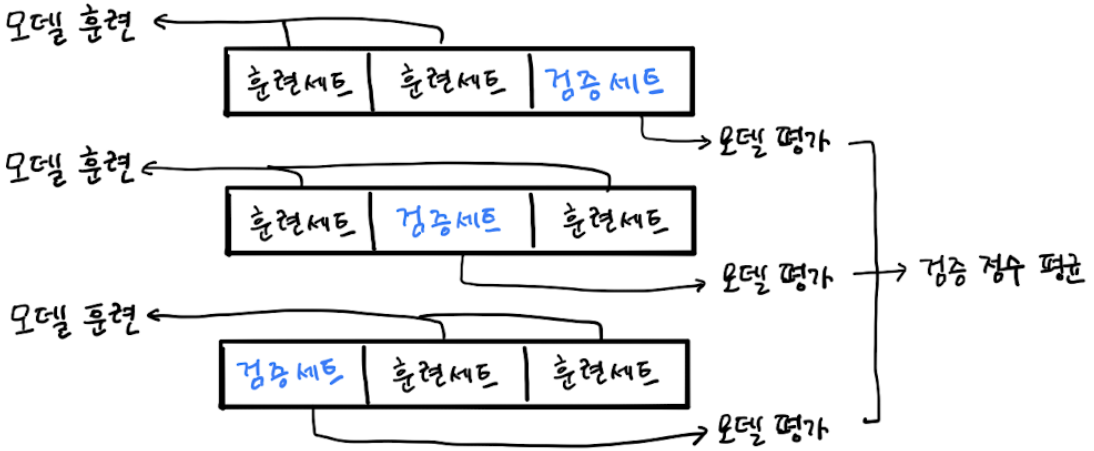
```
params = {'min_impurity_decrease': uniform(0.0001, 0.001),
          'max_depth': randint(20, 50),
          'min_samples_split': randint(2, 25),
          'min_samples_leaf': randint(1, 25),
          }

gs = RandomizedSearchCV(DecisionTreeClassifier(random_state=42), params,
                        n_iter=100, n_jobs=-1, random_state=42)
```

최적의 하이퍼파라미터 찾기: 랜덤 서치

```
params = {'min_impurity_decrease': uniform(0.0001, 0.001),
          'max_depth': randint(20, 50),
          'min_samples_split': randint(2, 25),
          'min_samples_leaf': randint(1, 25),
          }

from sklearn.model_selection import RandomizedSearchCV

gs = RandomizedSearchCV(DecisionTreeClassifier(random_state=42), params,
                        n_iter=100, n_jobs=-1, random_state=42)
gs.fit(train_input, train_target)

print(gs.best_params_)
{'max_depth': 39, 'min_impurity_decrease': 0.00034102546602601173,
 'min_samples_leaf': 7, 'min_samples_split': 13}

print(np.max(gs.cv_results_['mean_test_score']))
0.8695428296438884

dt = gs.best_estimator_
print(dt.score(test_input, test_target))
0.86
```

지난 시간에...



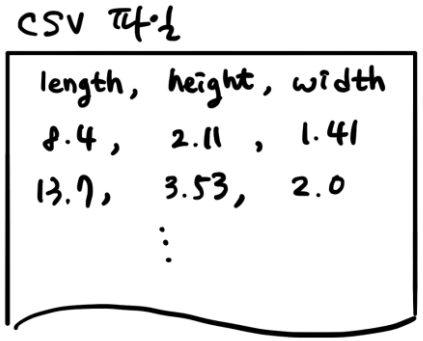



```
from sklearn.model_selection import GridSearchCV

params = {'min_impurity_decrease': [0.0001, 0.0002, 0.0003, 0.0004, 0.0005]}

gs = GridSearchCV(DecisionTreeClassifier(random_state=42), params, n_jobs=-1)
gs.fit(train_input, train_target)

dt = gs.best_estimator_
print(dt.score(train_input, train_target))
0.9615162593804117

print(gs.best_params_)
{'min_impurity_decrease': 0.0001}
```

앙상블 알고리즘

정형 데이터와 비정형 데이터

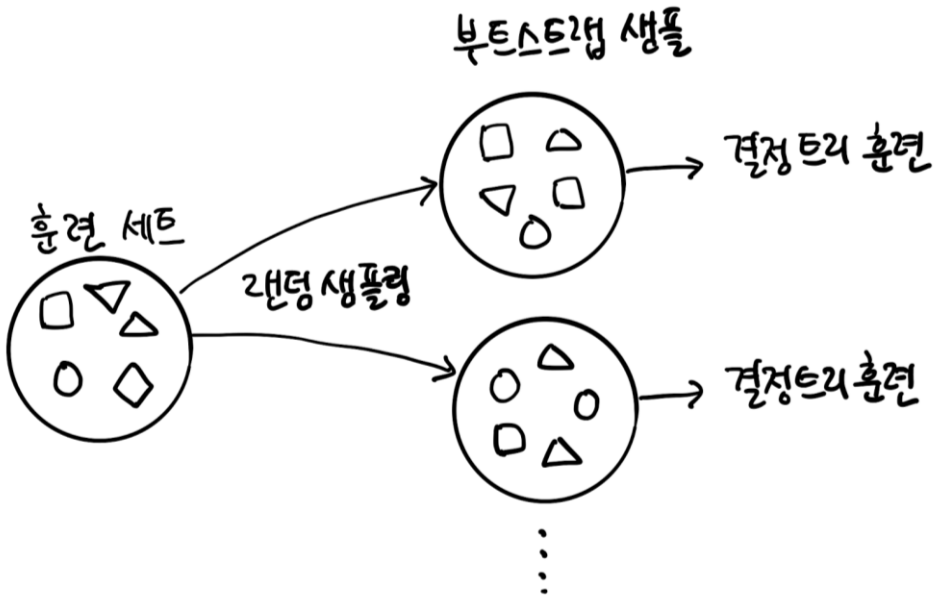
특징	정형데이터	비정형데이터
구조	고정된 스키마	유연하거나 고정된 구조 없음
저장 방식	테이블 형식 (행과 열)	파일 시스템, NoSQL 데이터베이스, 클라우드 저장소 등
분석 용이성	SQL 등을 사용한 쉬운 분석	데이터 마이닝, NLP, 이미지 처리 등 다양한 기술 필요
일관성	높음	낮음
예시	관계형 데이터베이스, 엑셀 시트, CSV 파일	이메일, 소셜 미디어 게시물, 이미지, 비디오, 오디오 파일 등



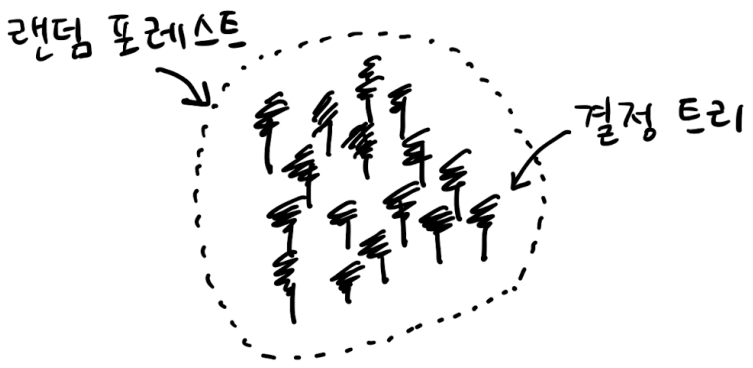
앙상블 모델

- 더 나은 예측 성능을 얻기 위해 여러 개의 모델을 결합하여 만든 모델
  - 단일 모델보다 더 높은 정확도와 안정성 제공, 과적합 방지에 유용함

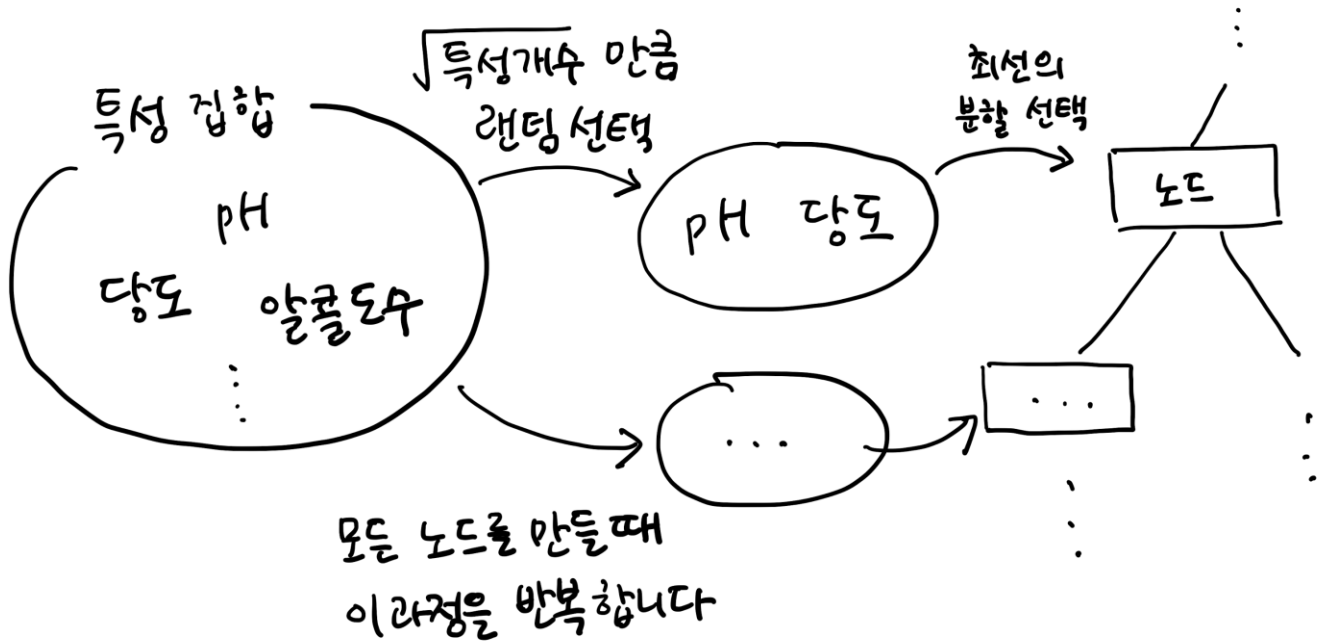
1. 배깅(Bagging) 적용한 기법
- Bagging: **Bootstrap** Aggregating 줄임말
- 여러 **데이터 샘플을 무작위로 추출**하여 각기 다른 모델을 학습한 후, 이 모델들의 예측을 결합하는 기법. 예) 랜덤 포레스트(Random Forest)



앙상블 모델-랜덤포레스트(Random Forest)



- 부트스트랩 방식으로 만들어진 훈련데이터로 결정 트리 만듬
- 여러 개의 결정 트리를 학습하고, 이 트리들의 예측을 평균내거나(회귀) 다수결 투표 방식으로 결합(분류)하여 최종 예측을 도출
- 결정트리의 노드 분할시, 전체 특성중 일부 특성을 무작위 (random)로 선택함
- from sklearn.ensemble
  import **RandomForestClassifier(n_estimators=100, random_state=42)**



앙상블 모델-랜덤포레스트(Random Forest)

```
from sklearn.model_selection import cross_validate
from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(n_jobs=-1, random_state=42)
scores = cross_validate(rf, train_input, train_target,
return_train_score=True, n_jobs=-1)

print(np.mean(scores['train_score']), np.mean(scores['test_score']))
0.997354196512243 1 0.8905151032797809

rf.fit(train_input, train_target)
print(rf.feature_importances_)    # 특성중요도 속성
[0.23167441 0.50039841 0.26792718]

rf = RandomForestClassifier(oob_score=True, n_jobs=-1, random_state=42)
rf.fit(train_input, train_target)
print(rf.oob_score_)
0.893400038483740 6
```

Out Of Bag 부트스트랩에 포함되지 않고 남는 샘플
oob를 검증세트로 할 경우 oob_score =True

앙상블 모델 - 엑스트라 트리(Extra Tree)

- 랜덤포레스트의 동작 원리와 비슷함
- 기본 100개의 결정트리 훈련
- 배깅 방식으로 훈련세트를 만들지 않고 전체 훈련세트 사용함
- 노드 분할시 가장 좋은 분할이 아닌 랜덤 분할함-속도가 랜덤포레스트보다 빠름

```
from sklearn.ensemble import ExtraTreesClassifier

et = ExtraTreesClassifier(n_jobs=-1, random_state=42)
scores = cross_validate(et, train_input, train_target,
return_train_score=True, n_jobs=-1)

print(np.mean(scores['train_score']), np.mean(scores['test_score']))
0.9974503966084433 0.8887848893166506

et.fit(train_input, train_target)
print(et.feature_importances_)
[0.20183568 0.52242907 0.27573525]
```

앙상블 모델

2. 부스팅(Boosting) 적용한 기법

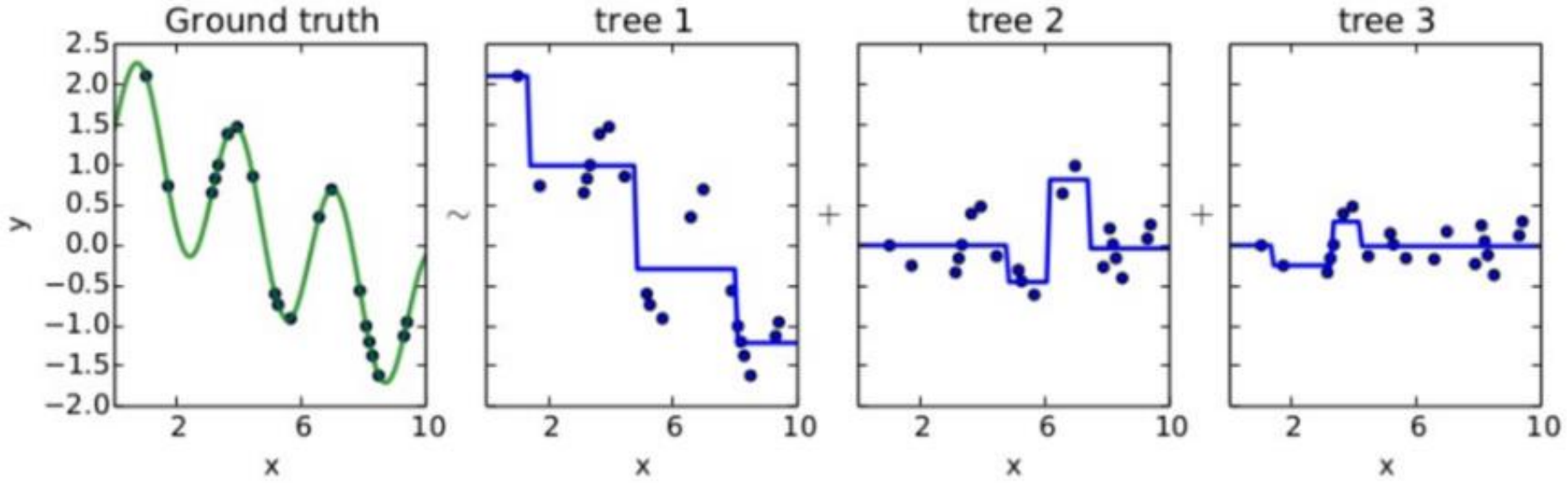
- Boosting: 모델을 순차적으로 학습시키는 기법
- 각 모델이 이전 모델의 오류를 줄이기 위해 학습

예) 그레디언트 부스팅 (Gradient Boosting)

: 깊이가 얕은 결정트리(depth=3)를 사용하여 **이전 트리의 오차를 보완**하는 방식으로 앙상블하는 방법

: 즉, 손실함수를 정의하고 손실함수의 최저점을 찾는 경사하강법 적용하여 결정트리를 추가해 나감

: 손실함수-분류에서는 로지스틱 함수, 회귀에서는 평균제곱오차함수를 손실함수



앙상블모델-그레이디언트 부스팅

```
from sklearn.ensemble import GradientBoostingClassifier

gb = GradientBoostingClassifier(random_state=42)   # n_estimators=100,learning_rate=0.1 (기본값)
scores = cross_validate(gb, train_input, train_target, return_train_score=True, n_jobs=-1)
print(np.mean(scores['train_score']), np.mean(scores['test_score']))
0.888108689215256 3 0.8720430147331015

gb = GradientBoostingClassifier(n_estimators=500, learning_rate=0.2, random_state=42)
scores = cross_validate(gb, train_input, train_target, return_train_score=True, n_jobs=-1)
print(np.mean(scores['train_score']), np.mean(scores['test_score']))
0.9464595437171814 0.8780082549788999

gb.fit(train_input, train_target)
print(gb.feature_importances_)
[0.15872278 0.68010884 0.16116839]
```

앙상블모델-히스토그램 기반 그레이디언트 부스팅

- 입력 데이터의 연속형 특성값을 256개의 히스토그램 빈(bin)으로 구분하여 사용함
- 대규모 데이터셋을 효율적으로 처리하면서 높은 성능을 유지하는데 유리한 알고리즘

```
from sklearn.experimental import enable_hist_gradient_boosting
from sklearn.ensemble import HistGradientBoostingClassifier

hgb = HistGradientBoostingClassifier(random_state=42)
scores = cross_validate(hgb, train_input, train_target, return_train_score=True, n_jobs=-1)

print(np.mean(scores['train_score']), np.mean(scores['test_score']))
0.9321723946453317 0.8801241948619236
```

모델의 특성 중요도 측정 함수-permutation_importance()

- 히스토그램 기반 그래디언트부스팅의 특성 중요도와 관련한 feature_importances_ 속성 지원하지 않음.
- permutation_importance(): 특성 중요도 계산하기 위한 함수
  동작원리: 특성을 하나씩 랜덤하게 섞어서 모델의 성능이 변화하는지 관찰, 어떤 특성이 중요한지를 계산함

```
from sklearn.inspection import permutation_importance
hgb.fit(train_input, train_target)
result = permutation_importance(hgb, train_input, train_target, n_repeats=10,
                                random_state=42, n_jobs=-1)
print(result.importances_mean)
[0.08876275 0.23438522 0.08027708] # sugar 특성값을 섞어서 모델 평가하면, 0.234 정도 정확도가 떨어진다.

result = permutation_importance(hgb, test_input, test_target, n_repeats=10,
                                random_state=42, n_jobs=-1)
print(result.importances_mean)
[0.05969231 0.20238462 0.049     ]

hgb.score(test_input, test_target)
0.8723076923076923
```