

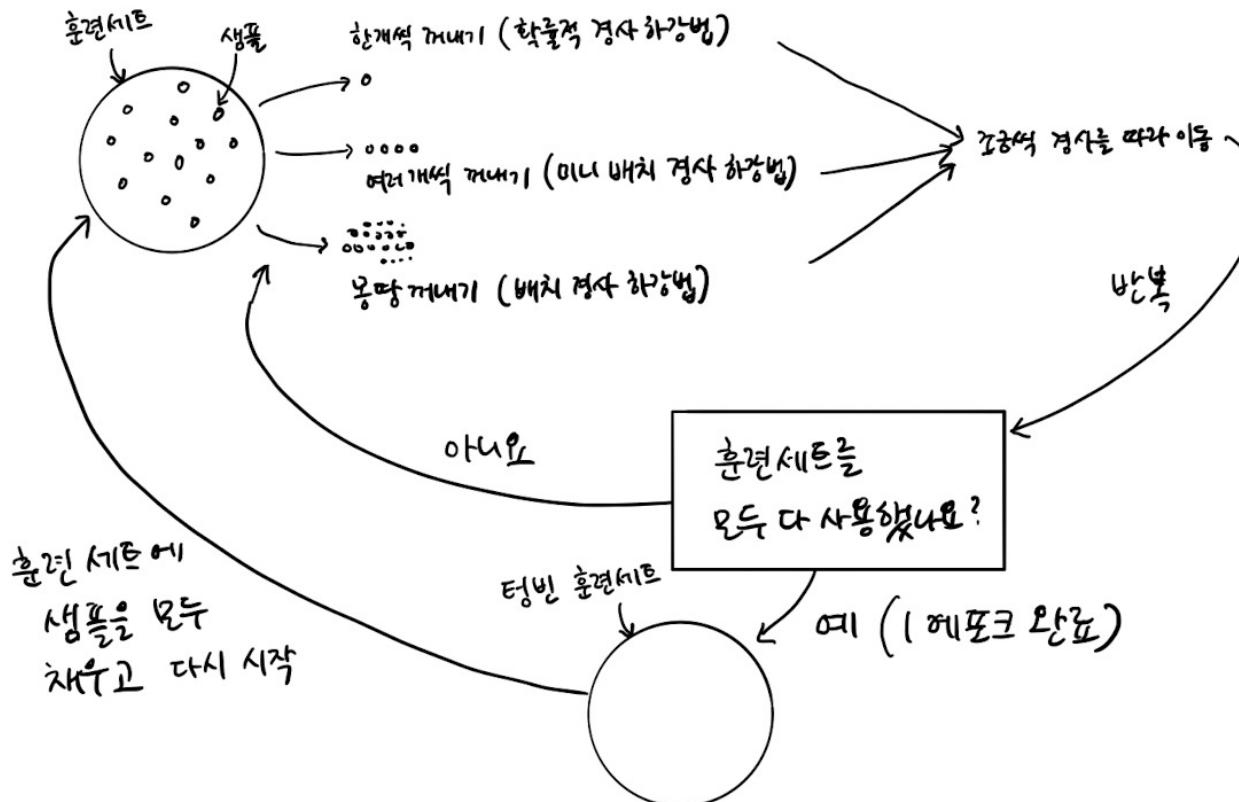
트리 알고리즘과 앙상블: 의사결정나무, 랜덤포레스트, 그레이디언트 부스팅

임 경 태

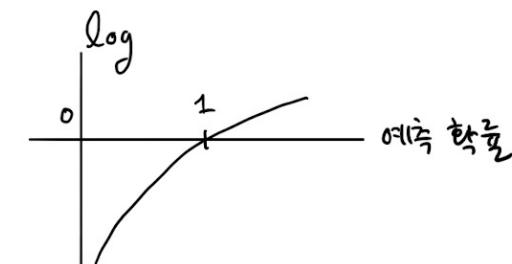
CONTENTS

- 1 복습**
- 2 결정트리
- 3 교차검증과 그리드서치
- 4 앙상블 (Bagging)
- 5 앙상블 (Boosting)

Logistic Regression 복습

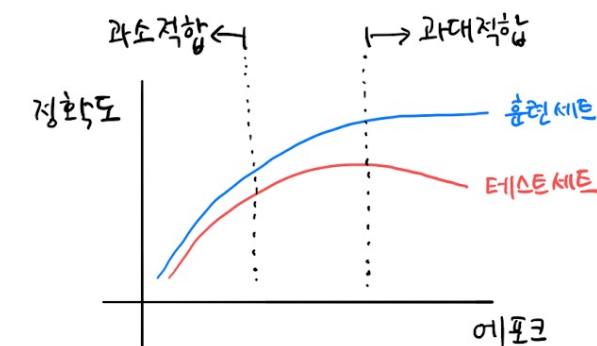


예측	정답(타깃)	
0.9	x 1	→ -0.9 ← 낮은 손실
0.3	x 1	→ -0.3 ←
0.2 → 0.8	x 1	→ -0.8 ← 높은 손실
0.8 → 0.2	x 1	→ -0.2 ←



타깃 = 1 일 때
 $\rightarrow -\log(\text{예측 확률})$

타깃 = 0 일 때
 $\rightarrow -\log(1 - \text{예측 확률})$



새로운 문제!

✎ 당도와 도수를 이용해 화이트/레드 와인 예측하기

- 입력: 당도, 도수, pH값
- 출력: 화이트/레드 와인
- 모델: ?



데이터 구조 살피기 및 전처리

- Train/Test 데이터 분할하기
- StandardScaler를 이용한 학습 데이터 표준화

```
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
```

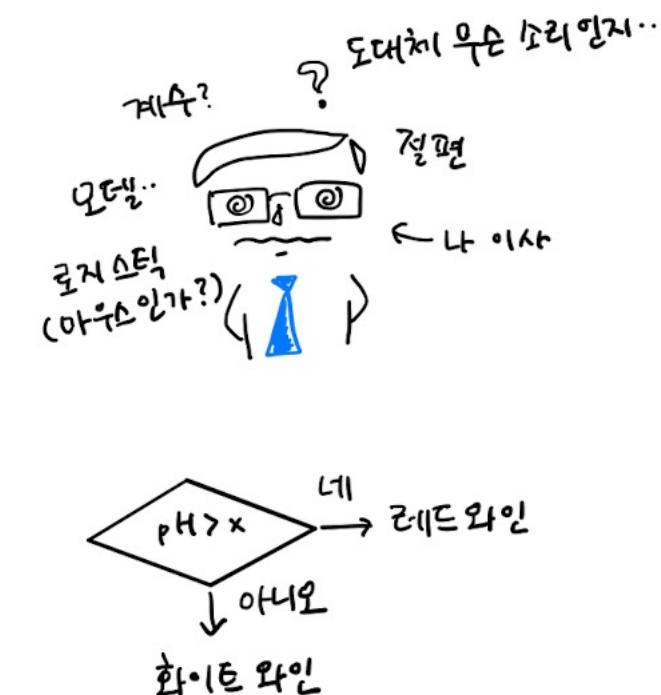
wine.describe()

	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

학습하기

- 회귀 모델로 나온 회귀계수는 예측 결과에 대한 이유를 설명하기 어려움
 - 예) 도수*0.51 + 당도*1.6 + ph*-0.68 + 1.81의 결과가 0보다 크면 화이트와인

```
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]
```



CONTENTS

- 1 복습
- 2 결정트리
- 3 교차검증과 그리드서치
- 4 앙상블 (Bagging)
- 5 앙상블 (Boosting)

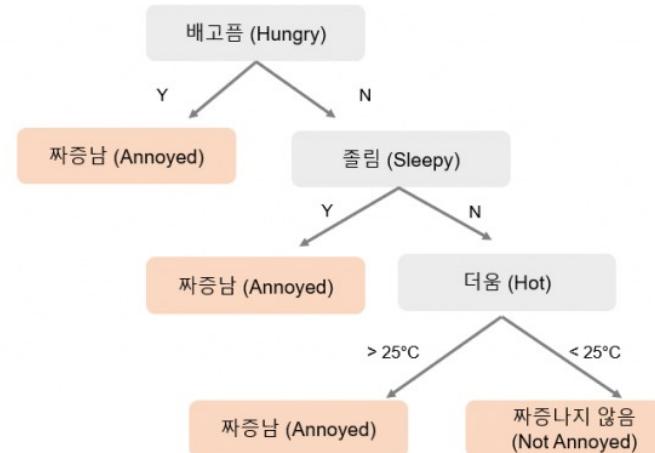
결정트리 (Decision Tree)

결정트리는 특정 기준에 의해 훈련 데이터를 '나무' 구조로 나눠가는 모델

- 결정트리는 예측 결과의 인과성을 설명하기 적합함. 또한, 열정적 모델로 학습 시 미리 학습된 계수를 이용해 예측이 빠름, 반면 kNN은 테스트 데이터를 받은 후 학습데이터를 이용해서 이웃을 구함 (예측이 느림, 게으른 모델)
- 결정트리 (=의사결정나무)는 동일한 데이터를 어떤 규칙과 순서로 나누는지에 따라 결과가 달라짐

의사결정 나무 예시

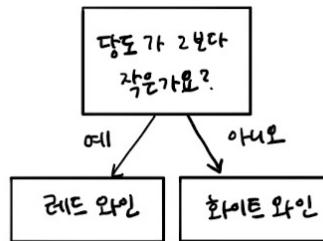
훈련 데이터(Training Data)				
	배고픔 (Hungry)	졸림 (Sleepy)	더움 (Hot)	짜증 (Annoyed)
1	Y	N	35°C	Y
2	Y	Y	20°C	Y
3	N	N	28°C	Y
4	N	N	18°C	N
5	N	Y	30°C	Y



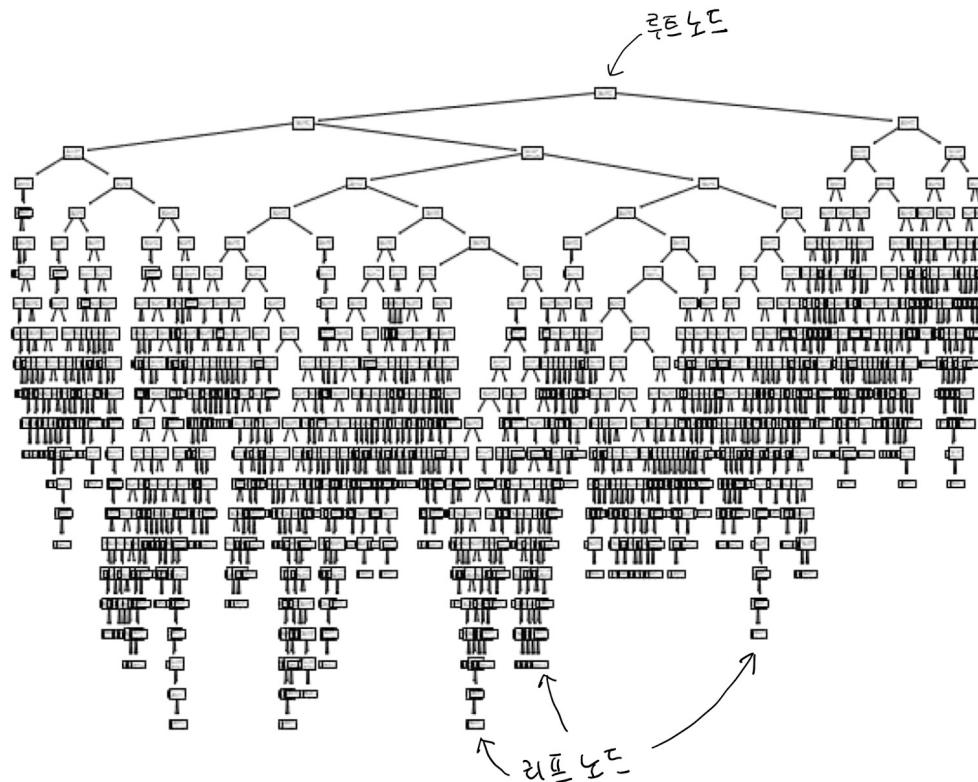
결정트리 (Decision Tree)

결정트리는 특정 기준에 의해 훈련 데이터를 '나무' 구조로 나눠가는 모델

- 학습 시 트리 구조를 확인할 수 있음



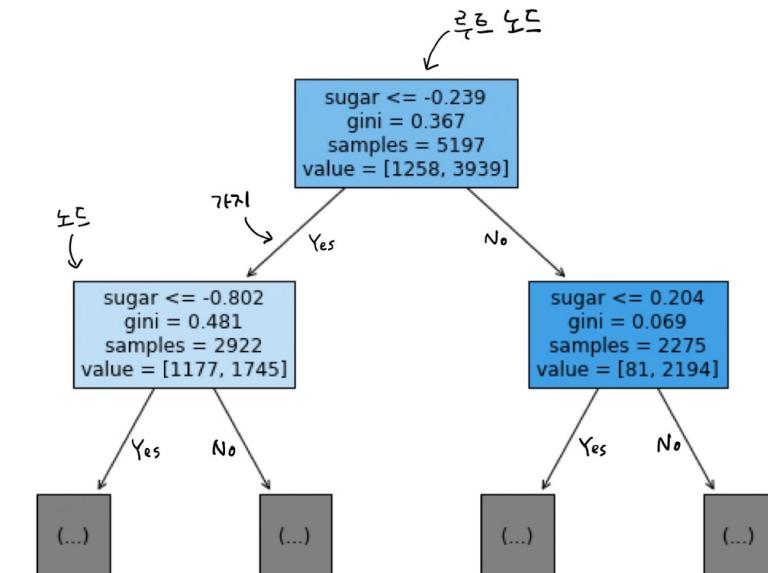
```
from sklearn.tree import DecisionTreeClassifier  
  
dt = DecisionTreeClassifier(random_state=42)  
dt.fit(train_scaled, train_target)  
  
print(dt.score(train_scaled, train_target))  
0.996921300750433  
print(dt.score(test_scaled, test_target))  
0.8592307692307692  
  
import matplotlib.pyplot as plt  
from sklearn.tree import plot_tree  
  
plt.figure(figsize=(10,7))  
plot_tree(dt)  
plt.show()
```



결정트리를 나누는 특정 기준이란?

- 결정트리는 학습 데이터를 이용해 데이터를 최적으로 분류해주는 질문들을 학습하는 머신러닝
- 스무고개를 생각해보자 질문을 잘해야 정답을 제대로 찾을 수 있겠지? 결정트리도 분류를 나누는 질문이 중요
- 그렇다면 어떤 질문이 분류를 나누기에 중요한 질문인가? 질문에 대해 분류가 최대한 명확하게 나뉠 때
 - **불순도**: 데이터가 분류되지 않고 섞여 있는 정도
 - **불순도가 낮음**: 분류가 명확하게 나뉨

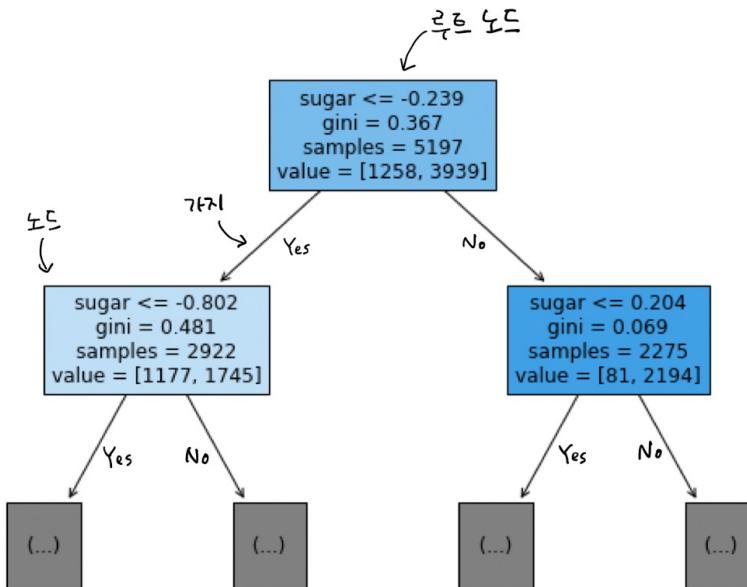
```
plt.figure(figsize=(10,7))
plot_tree(dt, max_depth=1, filled=True,
          feature_names=['alcohol', 'sugar', 'pH'])
plt.show()
```



결정트리 (Decision Tree)

✎ 지니 계수: 경제적 불평등을 표현하는 방법으로 0에 가까울 수록 평등함

- 결정 트리는 지니 계수를 이용해 노드를 나눔.



$$GINI(S) = 1 - \sum_{c=1}^C (p_i)^2, \text{ where } S = \text{tree}$$

```
def gini(x):  
    return 1 - ((x / x.sum())**2).sum()
```

$$\text{지니불순도} = 1 - (\frac{\text{음성클래스 비율}}{\text{전체}}^2 + \frac{\text{양성클래스 비율}}{\text{전체}}^2)$$

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

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

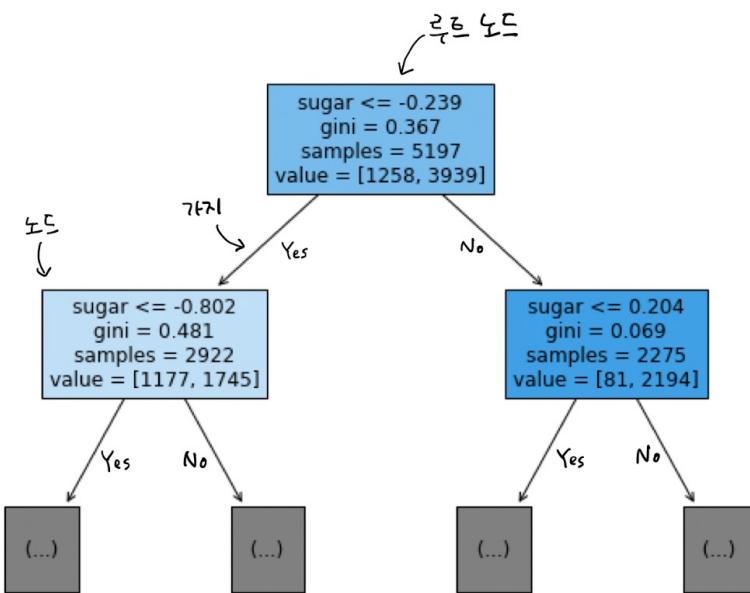
$$1 - \left(\left(\frac{0}{100} \right)^2 + \left(\frac{100}{100} \right)^2 \right) = 0$$

- 그렇다면 불순도가 어떤 상태 일 때 노드를 나눌까?

결정트리 (Decision Tree)

정보이득: 부모와 자식 노드 사이의 불순도 차이

- 결정트리는 정보 이득이 최대가 되도록 데이터를 나눔
 - 자식노드의 데이터 불순도가 작으면 작을 수록 커지게 됨



$$\text{부모의 불순도} - \frac{\text{왼쪽 노드의 생蟠率}}{\text{부모의 생蟠率}} \times \text{왼쪽 노드의 불순도} - \frac{\text{오른쪽 노드의 생蟠率}}{\text{부모의 생蟠率}} \times \text{오른쪽 노드의 불순도}$$

$$0.36 - (2922/5197)*0.481 - (2275/5197)*0.069 = 0.066$$

$$\text{지니불순도} = 1 - (\text{음성클래스 비율}^2 + \text{양성클래스 비율}^2)$$
$$1 - \left(\left(\frac{1258}{5197}\right)^2 + \left(\frac{3939}{5197}\right)^2 \right) = 0.367$$
$$1 - \left(\left(\frac{50}{100}\right)^2 + \left(\frac{50}{100}\right)^2 \right) = 0.5$$
$$1 - \left(\left(\frac{0}{100}\right)^2 + \left(\frac{100}{100}\right)^2 \right) = 0$$

결정트리 (Decision Tree)

결정트리 계산예제

- 다음 두 개의 의사결정트리 1과 2에 분류 A와 B가 존재할 때,

개별 노드의 gini계수 기반 불순도를 구하고 정보이득에 따라 비교하시오

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

$$\text{지니불순도} = 1 - (\text{음성클래스 비율}^2 + \text{양성클래스 비율}^2)$$

$$1 - \left(\left(\frac{159}{519} \right)^2 + \left(\frac{393}{519} \right)^2 \right) = 0.367$$

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

$$1 - \left(\left(\frac{90}{100} \right)^2 + \left(\frac{100}{100} \right)^2 \right) = 0$$

의사결정트리 1

A:40, B:40

A:30, B:10

A:10, B:30

의사결정트리 2

A:40, B:40

A:20, B:40

A:20, B:0

결정트리 (Decision Tree)

☞ 가지치기: 노드 깊이를 제한하여 결정트리가 Overfit 되는 것을 방지

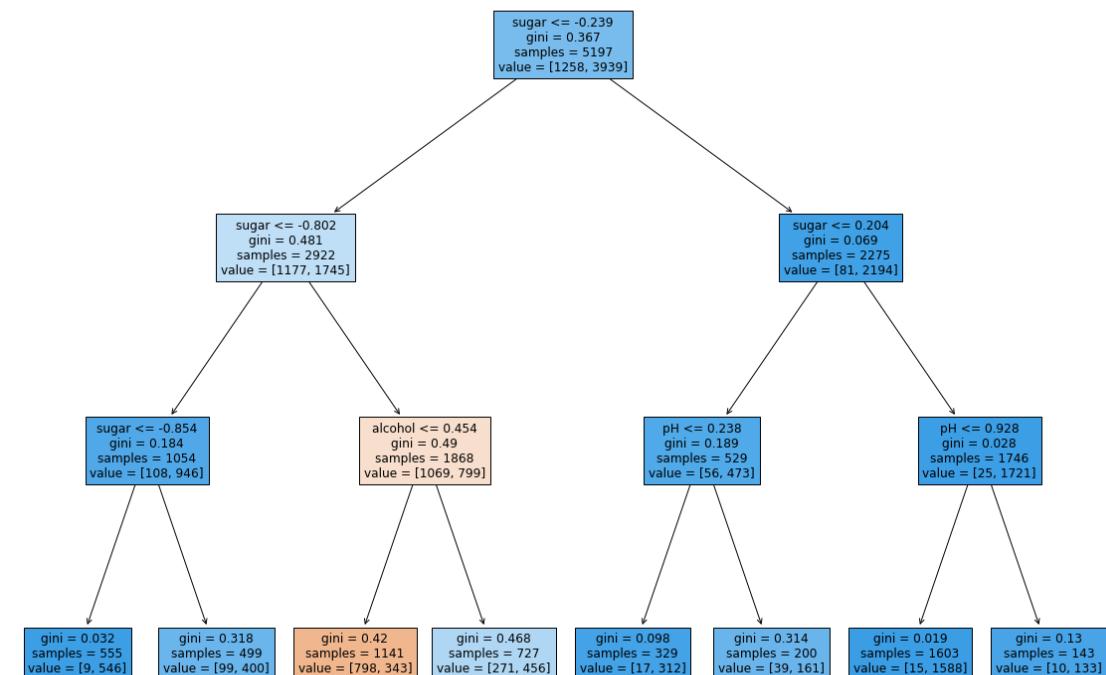
- 학습 데이터를 기준으로 매우 깊은 (분류를 위한 모든 경우의 수를 고려) 트리를 만들 경우

학습데이터에만 최적화된 학습 결과가 나올 수 있음. 이를 방지하기 위해 노드의 깊이를 제한

```
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 (Decision Tree)

표준화 하지 않은 데이터로 학습하기

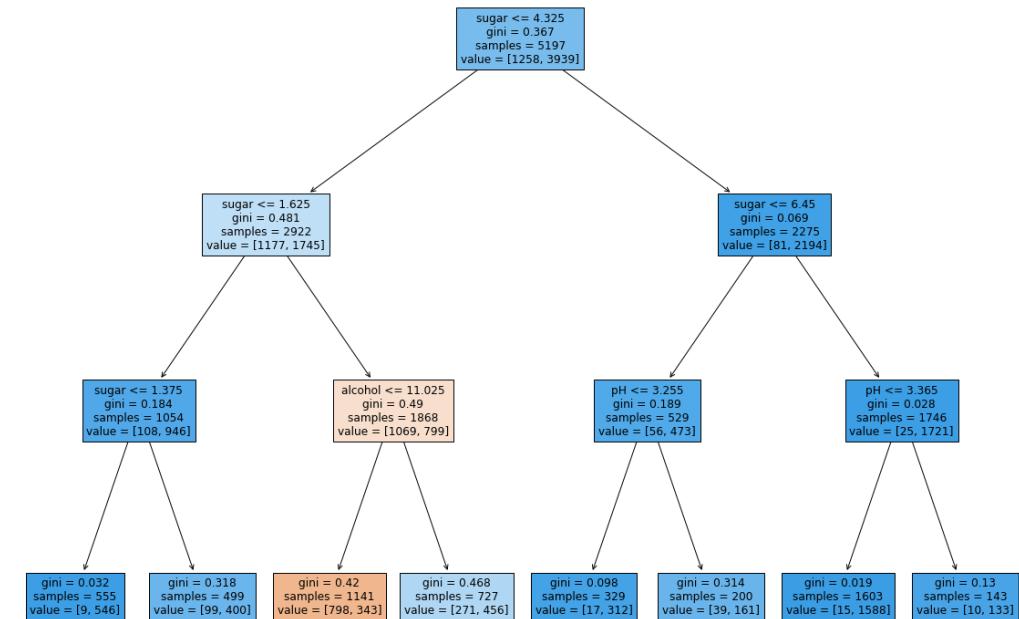
- 결정트리는 Value값 자체만으로 예측하기 때문에, 입력데이터를 표준화 하지 않아도 됨

```
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()

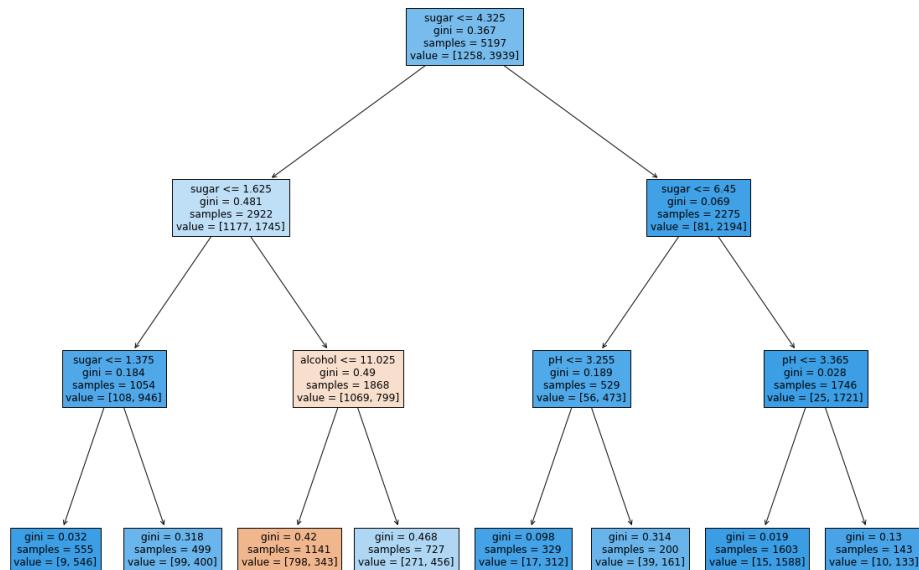
print(dt.feature_importances_)
[0.12345626 0.86862934 0.0079144 ]
```



그렇다면 마지막 남은 의문점... 결정트리의 최적의 depth는 어떤 값일까?

결정트리의 depth를 변경하여 최적의 세팅 찾기

- Test data에 가장 성능이 높은 depth값을 직접 변경해보고 가장 좋은 성능을 보이는 값을 찾자!
 - 이 경우 test data에는 성능이 높지만 실제 사용자가 넣은 새로운 데이터에 성능이 낮을 수 있지않을까?



어떻게 해결할 수 있을까...

CONTENTS

- 1 복습
- 2 결정트리
- 3 교차검증과 그리드서치
- 4 앙상블 (Bagging)
- 5 앙상블 (Boosting)

✎ 검증데이터 (Validation Data)

- 데이터를 학습/테스트에서 학습/검증/테스트로 3분할하여 테스트 데이터의 독립성을 유지
 - 검증 데이터는 최적의 depth를 찾는데 활용됨
 - 사용자가 신규로 입력하는 순수한 데이터로 (가정)활용 (훈련, 검증 과정에 활용되지 않아야 함)



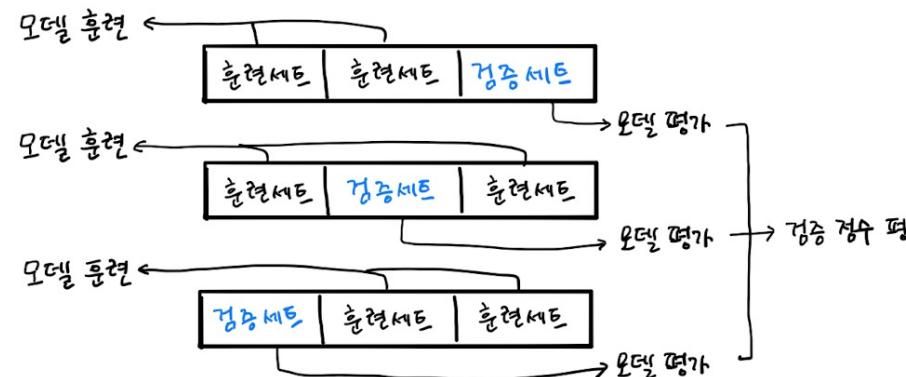
```
from sklearn.model_selection import train_test_split

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)
```

교차검증

- 학습/검증/테스트로 3분할하면 학습데이터 양이 작아져서 성능이 낮아짐, 이를 해결하기위해 활용
 - 학습데이터에서 소량의 검증세트를 떼어내어 모델을 여러개 만들고 평가하는 과정을 반복
 - 보통 K-fold cross validation이라 부르며 아래 그림은 3-fold 교차검증의 예제

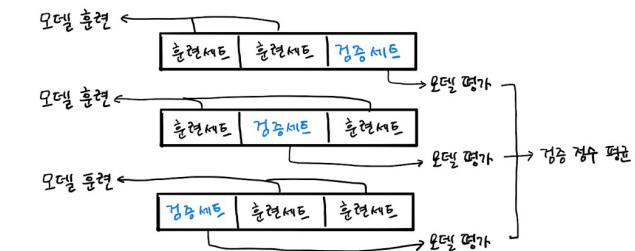


교차검증

- 일반적인 교차검증 K=5: 회귀모델은 KFold분할기 사용, 분류모델은 StratifiedKFold 사용

```
from sklearn.model_selection import cross_validate  
  
scores = cross_validate(dt, train_input, train_target)  
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])}  
  
import numpy as np  
  
print(np.mean(scores['test_score']))  
0.855300214703487  
  
from sklearn.model_selection import StratifiedKFold  
  
scores = cross_validate(dt, train_input, train_target, cv=StratifiedKFold())  
print(np.mean(scores['test_score']))  
0.855300214703487
```

두 개가 같음



- 분할기를 활용한 교차검증

```
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
```

성능이 약간 향상됨

- 총 10개의 fold를 이용: 데이터를 10개로 나누면
- 1) 검증데이터의 수가 1/10으로 적어지고 상대적으로
 - 2) 학습데이터의 수가 9/10으로 많아짐
 - 3) 따라서 검증데이터 분리로 인한 성능 하락을 감소시키고
 - 4) 총 10번의 평가를 진행 함으로써 general한 성능을 확인

그리드서치 (Grid Search)와 하이퍼파라미터

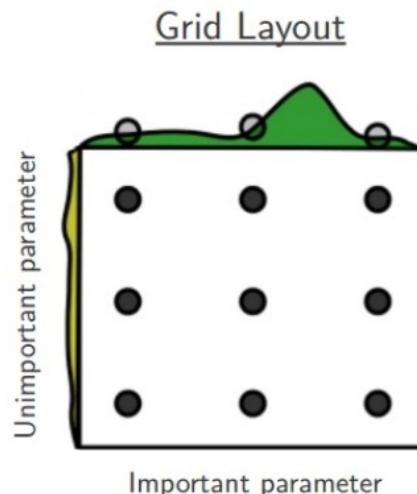
- **하이퍼파라미터 (Hyperparameter)**: 사용자 지정 파라미터 (max_depth, min_samples_split, num_epoch ...)
- **모델 파라미터**: 모델이 학습하는 파라미터(값) (Weight (W), bias (b))
- 그렇다면 의사결정 나무의 Hyperparameter인 max_depth의 **최적값은** 어떻게 찾을 수 있을까?
 - Hyperparameter의 값을 변경해보면서 실험하면 되겠지? == **하이퍼파라미터 튜닝**

그리드서치 (Grid Search)와 하이퍼파라미터

- **하이퍼파라미터 튜닝의 난점:** 최적의 max_depth를 수작업으로 찾는다 하여도 다른 하이퍼파라미터 min_samples_split의 설정 값에 따라 max_depth의 최적값이 변화됨.. 따라서 n^2 의 경우의 수를 탐험 해야함
- **하이퍼파라미터 튜닝의 해결:** for 반복문 활용 혹은 그리드서치, 랜덤서치 등을 이용

그리드서치 (Grid Search)와 하이퍼파라미터

- 그리드서치 (격자탐색): 특정 간격에서 모델에게 가장 적합한 하이퍼파라미터를 찾기



```
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}

print(gs.cv_results_['mean_test_score']) 5개 간격에 대한 5-fold의 평균 성능
[0.86819297 0.86453617 0.86492226 0.86780891 0.86761605]
```

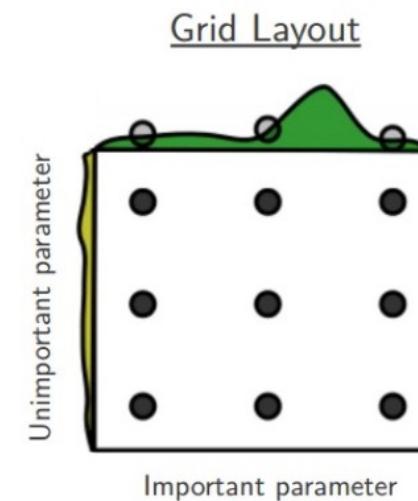
5개의 간격에 대해
디폴트: 5-fold로 설정
즉 5개 간격 * 5-fold = 25개의 모델 학습

```
params = {'min_impurity_decrease': np.arange(0.0001, 0.001, 0.0001),
          'max_depth': range(5, 20, 1), 0.0001 ~ 0.0010 | 될 때 까지 0.0001씩 더한 집합 (9개)
          'min_samples_split': range(2, 100, 10) 5 ~ 20까지 1씩 더한 집합 (15개)
        }
          2 ~ 100까지 10씩 더한 집합 (10개)
```

따라서 9 * 15 * 10 개의 교차검증 수행 여기서 5-fold로 나누었으니 *5 = 모델 수는 6750

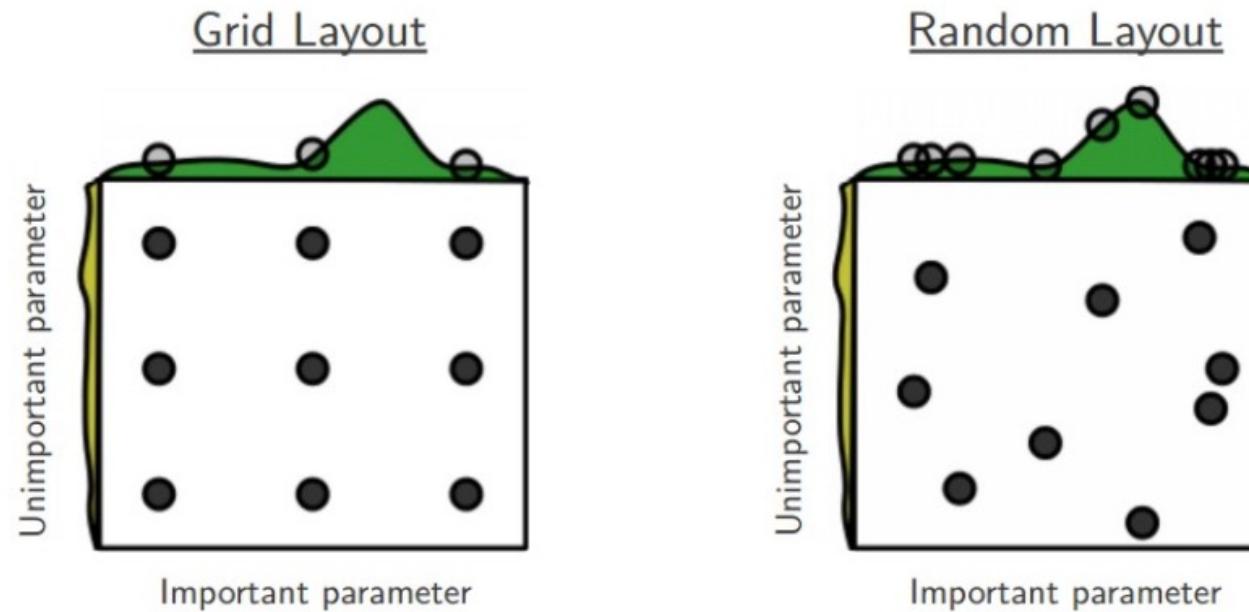
❶ 랜덤서치 (Random Search)와 확률분포

- 그리드서치 (격자탐색)의 문제점:
 - 하이퍼파라미터가 추가될 때마다 학습 해야할 모델 수가 n 승으로 상승
 - 사람이 개입해서 하이퍼파라미터의 값의 범위를 직접 지정해 줘야함
- 그리드서치의 해결을 위해 **랜덤서치** 활용 가능



❸ 랜덤서치 (Random Search)와 확률분포

- **랜덤서치:** 그리드서치처럼 하이퍼파라미터 값을 직접 전달하지 않고 확률 분포를 전달해 그중 랜덤으로 샘플링
 - 그렇다면 확률분포를 어떻게 전달할까?



❸ 랜덤서치 (Random Search)와 확률분포

- **랜덤서치:** 그리드서치처럼 하이퍼파라미터 값을 직접 전달하지 않고 확률 분포를 전달해 그중 랜덤으로 샘플링
 - 그렇다면 확률분포를 어떻게 전달할까? Scipy를 이용해 확률분포 생성!

```
from scipy.stats import uniform, randint

ugen = uniform(0, 1)      0~1 사이 실수 값의 범위를 갖는 분포 생성
ugen.rvs(10)             0~1 사이의 범위를 갖는 분포에서 10개를 랜덤으로 생성
array([ 0.67694587,  0.77912183,  0.73608526,  0.64430581,  0.8250335 ,
       0.45253031,  0.47240473,  0.81925782,  0.95971199,  0.75004125])
```

```
rgen = randint(0, 10)    0~10 사이 정수 값의 범위를 갖는 분포 생성
rgen.rvs(10)             0~10 사이의 범위를 갖는 분포에서 10개를 랜덤으로 생성
array([ 9,  2,  1,  8,  6,  4,  5,  6,  2,  6])

np.unique(rgen.rvs(1000), return_counts=True)
(array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9]),
 array([ 95,   90,   90,  115,   97,   96,  108,  101,  113,   95]))
```

✎ 랜덤서치 (Random Search)와 확률분포

- **랜덤서치:** 그리드서치처럼 하이퍼파라미터 값을 직접 전달하지 않고 확률 분포를 전달해 그중 랜덤으로 샘플링

```
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_ 랜덤서치에서 찾은 최적 모델(최적의 하이퍼파라미터가 사용된)을 dt에 저장
print(dt.score(test_input, test_target))
0.86
```

CONTENTS

- 1 복습
- 2 결정트리
- 3 교차검증과 그리드서치
- 4 앙상블 (Bagging)
- 5 앙상블 (Boosting)

정형 데이터와 비 정형 데이터

- 정형데이터 (Structured data): 일반적인 tabular 데이터 (엑셀, DB 등)
- 비정형데이터 (Unstructured data): Image, Video, Time Series, Text 등

Tabular (정형)

A	B	C	D	E	F	G	H	I	J	K	L
Passenger	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	0	3	Braund, Mr. male		22	1	0	A/5 21171	7.25		S
2	1	1	Cumings, female		38	1	0	PC 17599	71.2833	C85	C
3	2	1	Heikkinen, male		26	0	0	STON/O2.	7.925		S
4	3	1	Futrelle, Male		35	1	0	113803	53.1	C123	S
5	4	1	Allen, Mr. male		35	0	0	373450	8.05		S
6	5	0	Moran, M. male		35	0	0	330877	8.4583		Q
7	6	0	McCarthy, male		54	0	0	17463	51.8625	E46	S
8	7	0	Palsson, Mr. male		2	3	1	349909	21.075		S
9	8	0	Johnson, female		27	0	2	347742	11.1333		S
10	9	1									

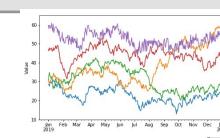
지금까지 우리가 다룬 데이터
전통적인 머신러닝이 해결

Non-tabular (비정형)

Image & Video



Time Series



Text

binary code with which the present is represented. It has the property that the symbol (or character) representing each number or symbol is the same as the symbol representing the next higher number (or character). Because this code in its primary form is not very useful, it undergoes a sort of reflection process and is converted into a similar form, which has as yet not been named. In this specification and is the "reflected binary code." a receiver station, reflected binary

딥러닝 기반의
모던 기계학습이 해결

그렇다면 정형데이터의 끝판왕 모델은? 앙상블



앙상블이란?

- 모델 하나를 학습하기보다 다수의 모델을 학습하여 최종 의사결정(예측)을 진행하는 모델
 - 일반적인 의미의 앙상블엔 투표 방법이 가장 일반적이며, 예측 확률을 Weight Sum 하는 방법 등 다양함.

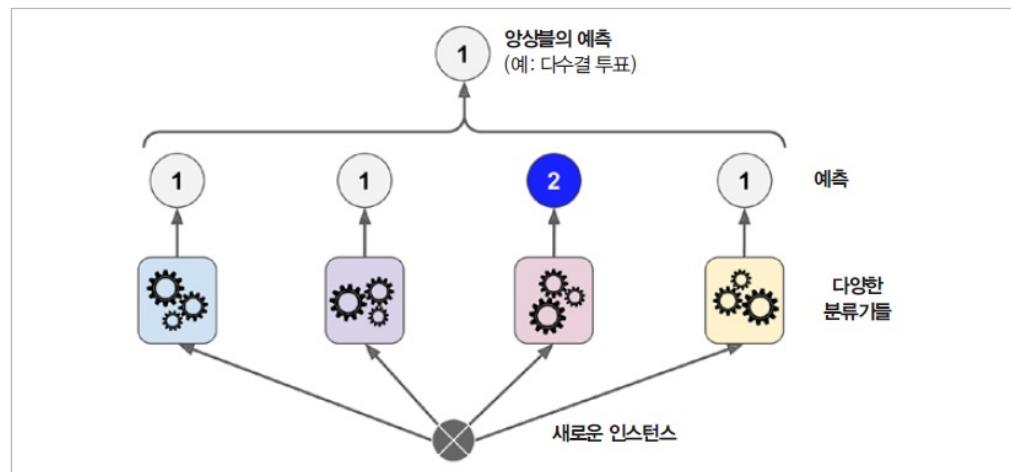


그림 7-2 직접 투표 분류기의 예측

```
from sklearn.metrics import accuracy_score

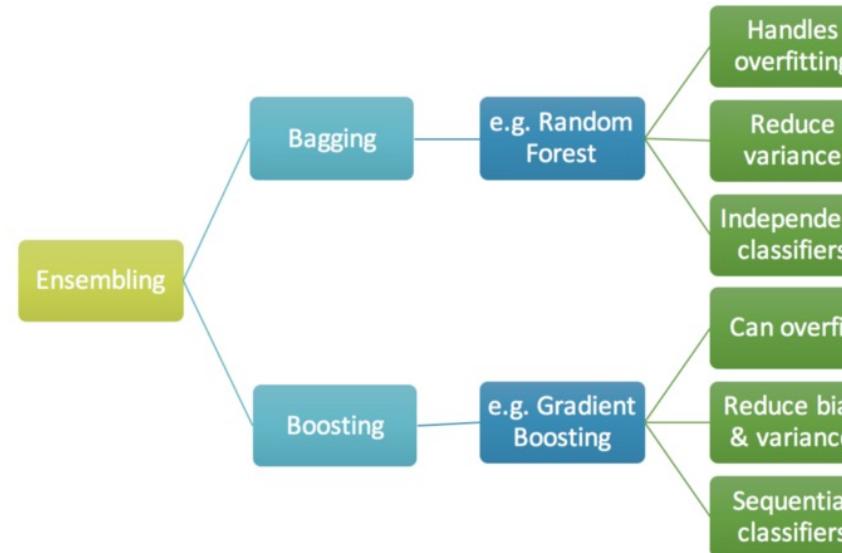
for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
    clf.fit(trainData, trainLabel)
    predict = clf.predict(testData)
    print(clf.__class__.__name__, accuracy_score(testLabel, predict))

Result
LogisticRegression 0.864
RandomForestClassifier 0.872
SVC 0.888
VotingClassifier 0.896
```

그렇다면 의사결정나무에 앙상블을 적용하면? **랜덤포레스트**

❶ 앙상블의 분류?

- **Bagging:** 같은 알고리즘을 사용하여 훈련 세트의 서브셋을 무작위로 구성하여 분류기를 각기 다르게 학습
- **Boosting:** 약한 학습기를 여러 개 연결하여 강한 학습기를 만드는 앙상블 방법



✎ Bagging 기반의 양상을 방법

- 같은 알고리즘을 사용하여 훈련 세트의 서브셋을 무작위로 구성하여 분류기를 각기 다르게 학습

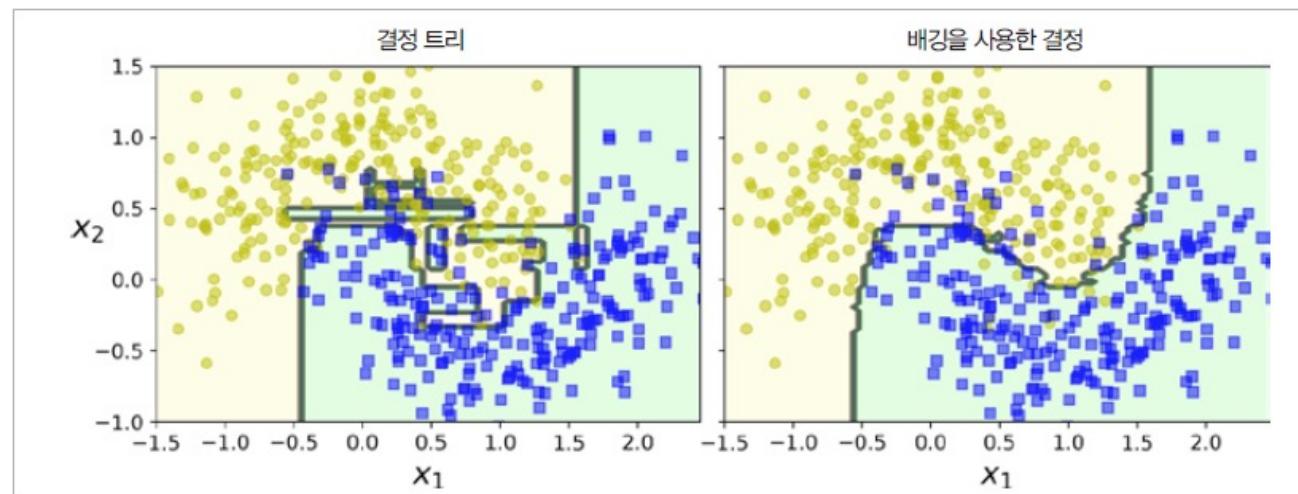
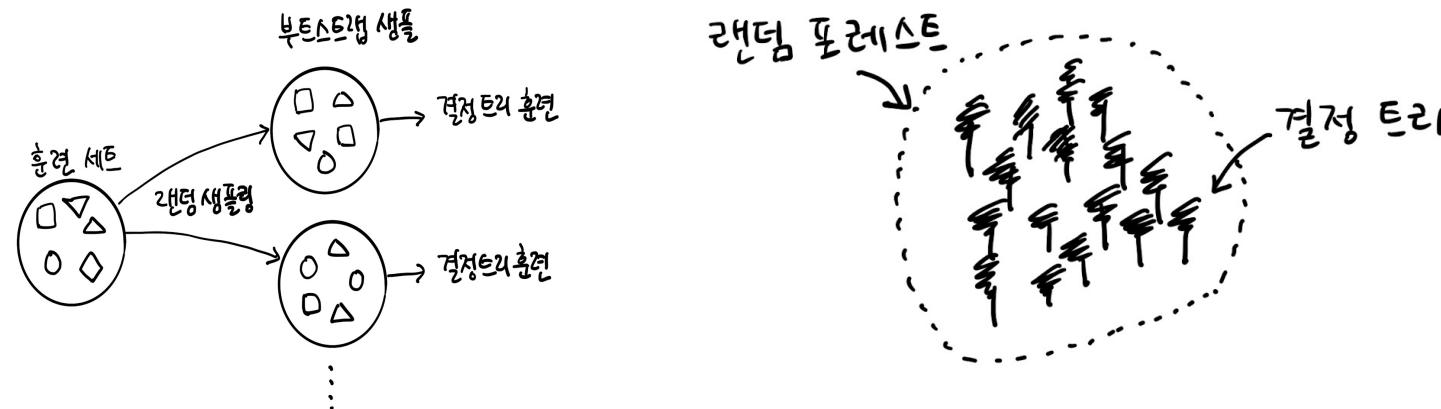


그림 7-5 단일 결정 트리(왼쪽)와 500개 트리로 만든 배깅 양상을(오른쪽) 비교

그렇다면 의사결정나무에 배깅 양상을 적용하면? **랜덤포레스트**

❶ 랜덤포레스트 (Random Forest)

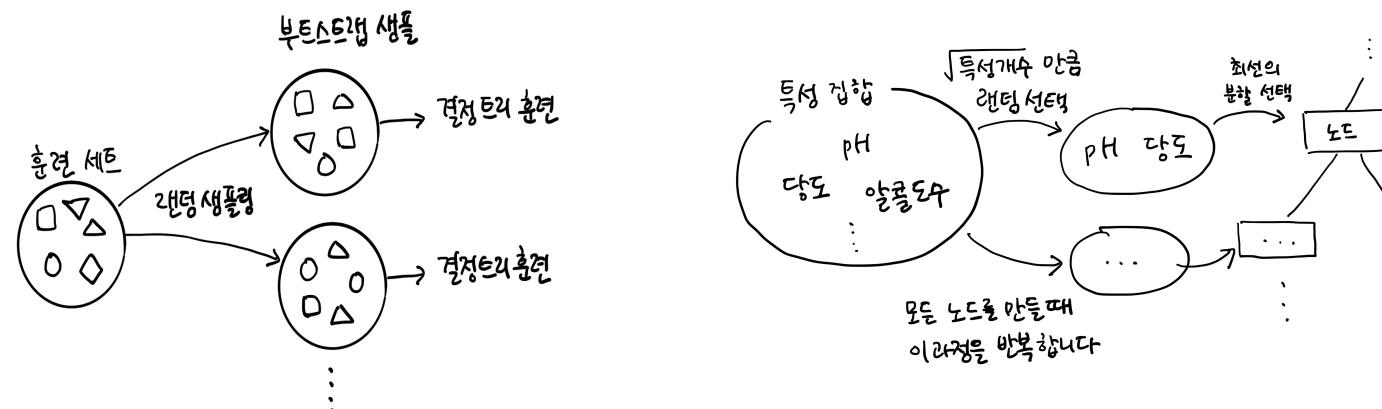
- 정의: 결정트리를 랜덤하게 여러개 만들어 결정트리의 숲을 만드는 방법
- 결정트리를 랜덤하게 만드는 방법: 부스트랩 샘플
 - (랜덤 데이터) 학습데이터로부터 중복을 허용하여 랜덤하게 데이터를 샘플링하는 방법



양상을 학습방법 (Bagging)

랜덤포레스트 (Random Forest)

- 정의: 결정트리를 랜덤하게 여러개 만들어 결정트리의 숲을 만드는 방법
- 결정트리를 랜덤하게 만드는 방법: 부스트랩 샘플
 - (랜덤 데이터) 학습데이터로부터 중복을 허용하여 랜덤하게 데이터를 샘플링하는 방법
 - (랜덤 특성) 노드를 분할 할때 특성 중 일부 특성을 무작위로 고르는 무작위성을 더함



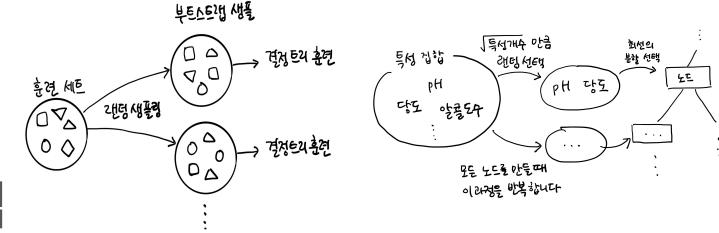
앙상블 학습방법 (Bagging)



랜덤포레스트 (Random Forest)

- 정의: 결정트리를 랜덤하게 여러개 만들어 결정트리의 숲을 만드는 방법

- 기본적으로 100개의 결정트리를 학습해서 하나의 랜덤포레스트를 만듬



```
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']))
0.9973541965122431 0.8905151032797809

rf.fit(train_input, train_target)
print(rf.feature_importances_)          결정트리로 모델링했을 때의 값 [0.12, 0.86, 0.007] 보다 당도에 대한 의존도가 낮아짐, 이 말은
[0.23167441 0.50039841 0.26792718] 랜덤으로 특성을 활용했기 때문에 하나의 특성에 과도하게 집중하지 않아 과적합 줄이고 일반화에 좋음
    알콜도수,      당도,      pH

rf = RandomForestClassifier(oob_score=True, n_jobs=-1, random_state=42)
rf.fit(train_input, train_target)      Out of Bag == 복원 추출 랜덤샘플링에서 한번도 포함되지 않은 샘플을 이용해 자동으로 성능 검증
print(rf.oob_score_)
0.8934000384837406
```

앙상블 학습방법 (Bagging)

❶ 엑스트라트리 (Extra Trees)

- 정의: 부트스트랩 샘플을 사용하지 않고 결정트리의 특성을 랜덤하게, 노드 분할 기준도 랜덤하게 만든 모델

- 장점: 모든 학습데이터가 전혀 누락되지 않고 사용됨, 특성을 무작위로 선택하기 때문에 과대적합을 막음
랜덤하게 노드를 분할하여 속도가 빠름 (기존 결정트리는 노드 분할 기준이 “best” 이걸 랜덤으로)
- 단점: 랜덤하게 노드를 분할하여 무작위성이 크기 때문에 더 많은 결정트리를 훈련 해야함

```
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]
```

CONTENTS

- 1 복습
- 2 결정트리
- 3 교차검증과 그리드서치
- 4 앙상블 (Bagging)
- 5 앙상블 (Boosting)

Boosting 기반의 양상을 방법

- Boosting: 약한 학습기를 여러 개 연결하여 강한 학습기를 만드는 양상을 방법
- 에이다부스트: 이전 예측기를 보완하는 새로운 예측기를 만드는 방법
 - 알고리즘이 기반이 되는 첫 번째 분류기(예를 들면 결정 트리)를 훈련 세트에서 훈련시키고 예측
 - 알고리즘이 잘못 분류된 훈련 샘플의 가중치를 상대적으로 높임 → 두 번째 분류기는 업데이트된 가중치를 사용해 훈련 세트에서 훈련하고 다시 예측을 만듦 → 다시 가중치를 업데이트하는 식으로 계속 진행

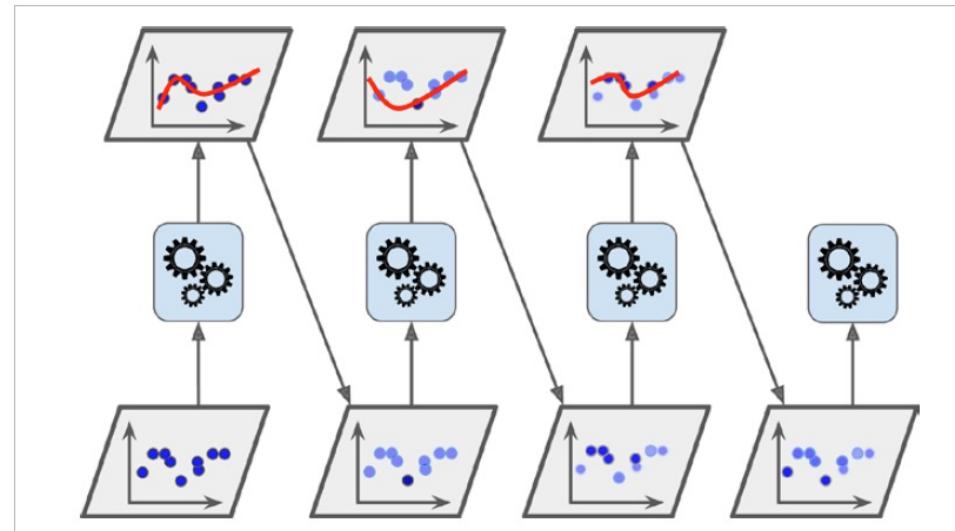


그림 7-7 샘플의 가중치를 업데이트하면서 순차적으로 학습하는 에이다부스트

그래디언트 부스팅 (Gradient Boosting)

- 정의: 에이다부스트처럼 반복마다 샘플의 가중치를 수정하는 대신 이전 예측기가 만든 잔여 오차에 새로운 예측기를 학습

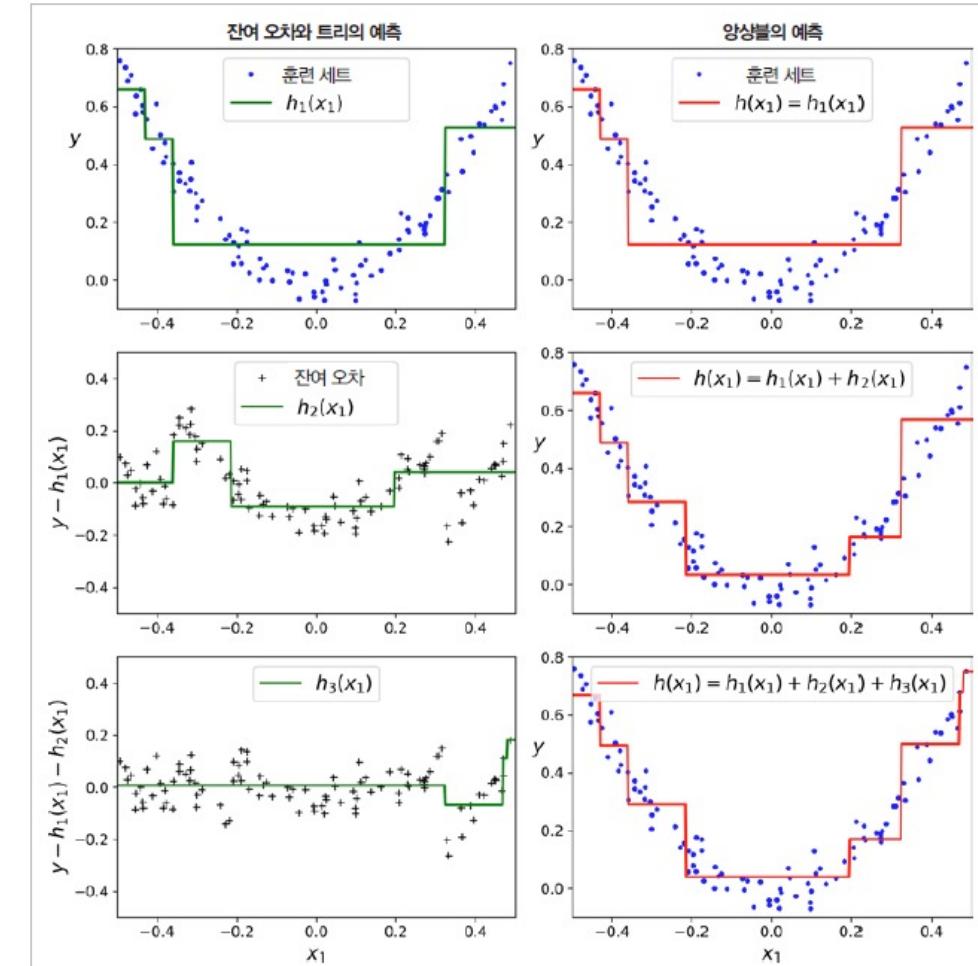


그림 7-9 이 그레이디언트 부스팅 그래프에서 첫 번째 예측기(왼쪽 위)가 평소와 같이 훈련됩니다. 그다음 연이은 예측기(왼쪽 중간, 왼쪽 아래)가 이전의 예측기의 잔여 오차에서 훈련됩니다. 오른쪽 열은 만들어진 양상블의 예측을 보여줍니다.

GRAPHIC PEN 그레디언트 부스팅 (Gradient Boosting)

- 정의: 에이다부스트처럼 반복 마다 샘플의 가중치를

수정하는 대신 이전 예측기가 만든 잔여 오차에

새로운 예측기를 학습

```
from sklearn.ensemble import GradientBoostingClassifier

gb = GradientBoostingClassifier(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']))
0.8881086892152563 0.8720430147331015

활용할 결정나무의 개수
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]
```

앙상블 학습방법 (Boosting)

✎ 다양한 부스팅 기반의 앙상블 방법

- 히스토그램 기반 그래디언트 부스팅 / XGBoost / LightGBM

```
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

from xgboost import XGBClassifier

xgb = XGBClassifier(tree_method='hist', random_state=42)
scores = cross_validate(xgb, train_input, train_target, return_train_score=True, n_jobs=-1)

print(np.mean(scores['train_score']), np.mean(scores['test_score']))
0.8824322471423747 0.8726214185237284

from lightgbm import LGBMClassifier

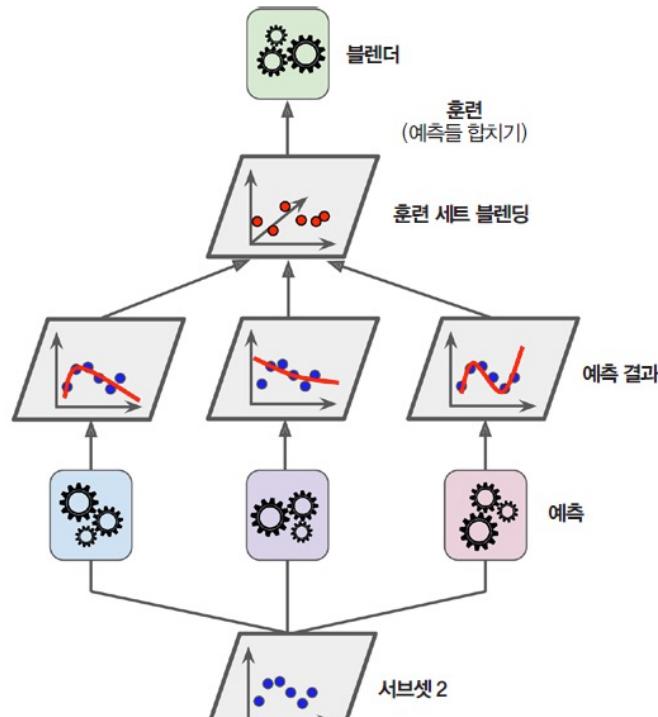
lgb = LGBMClassifier(random_state=42)
scores = cross_validate(lgb, train_input, train_target, return_train_score=True, n_jobs=-1)

print(np.mean(scores['train_score']), np.mean(scores['test_score']))
0.9338079582727165 0.8789710890649293
```

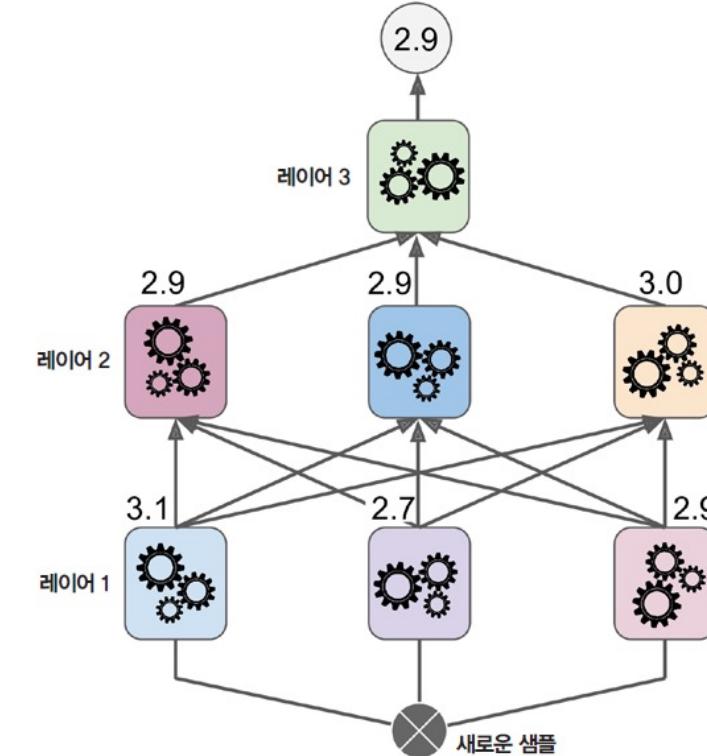
감사합니다.

▶ 스태킹

- 앙상블에 속한 모든 예측기의 예측을 취합하는 간단한 함수 (예를 들면 투표)를 사용하는 대신 취합한 모델의 결과에서 최적의 예측을 만드는 모델을 학습시키면 안될까?



▲ 그림 7-14 블렌더 훈련



▲ 그림 7-15 멀티 레이어 스태킹 앙상블의 예측