결정 트리

로지스틱 회귀로 와인 분류하기

In [1]:
```python
import pandas as pd
wine=pd.read_csv("wine.csv")
wine.head()
```

Out[1]:

	alcohol	sugar	pH	class
0	9.4	1.9	3.51	0.0
1	9.8	2.6	3.20	0.0
2	9.8	2.3	3.26	0.0
3	9.8	1.9	3.16	0.0
4	9.4	1.9	3.51	0.0

In [2]:
```python
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
```

In [3]:
```python
wine.describe()
```

Out[3]:

	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

In [4]:
```python
data=wine[["alcohol","sugar","pH"]].to_numpy()
target=wine["class"].to_numpy()
```

In [5]:
```python
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)
```

In [6]:
```python
print(train_input.shape,test_input.shape)
```

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

In [7]:
```python
from sklearn.preprocessing import StandardScaler
ss=StandardScaler()
ss.fit(train_input)
train_scaled=ss.transform(train_input)
test_scaled=ss.transform(test_input)
```

In [8]:
```python
from sklearn.linear_model import LogisticRegression
lr=LogisticRegression()
lr.fit(train_scaled,train_target)
print(lr.score(train_scaled,train_target))
print(lr.score(test_scaled,test_target))
```

```
0.7808350971714451
0.7776923076923077
```

설명하기 쉬운 모델과 어려운 모델

- 로지스틱 회귀로 분류하는 것은 결과를 설명하기 어려운 단점이 있음

In [9]:
```python
print(lr.coef_,lr.intercept_)
```

```
[[ 0.51270274  1.6733911  -0.68767781]] [1.81777902]
```

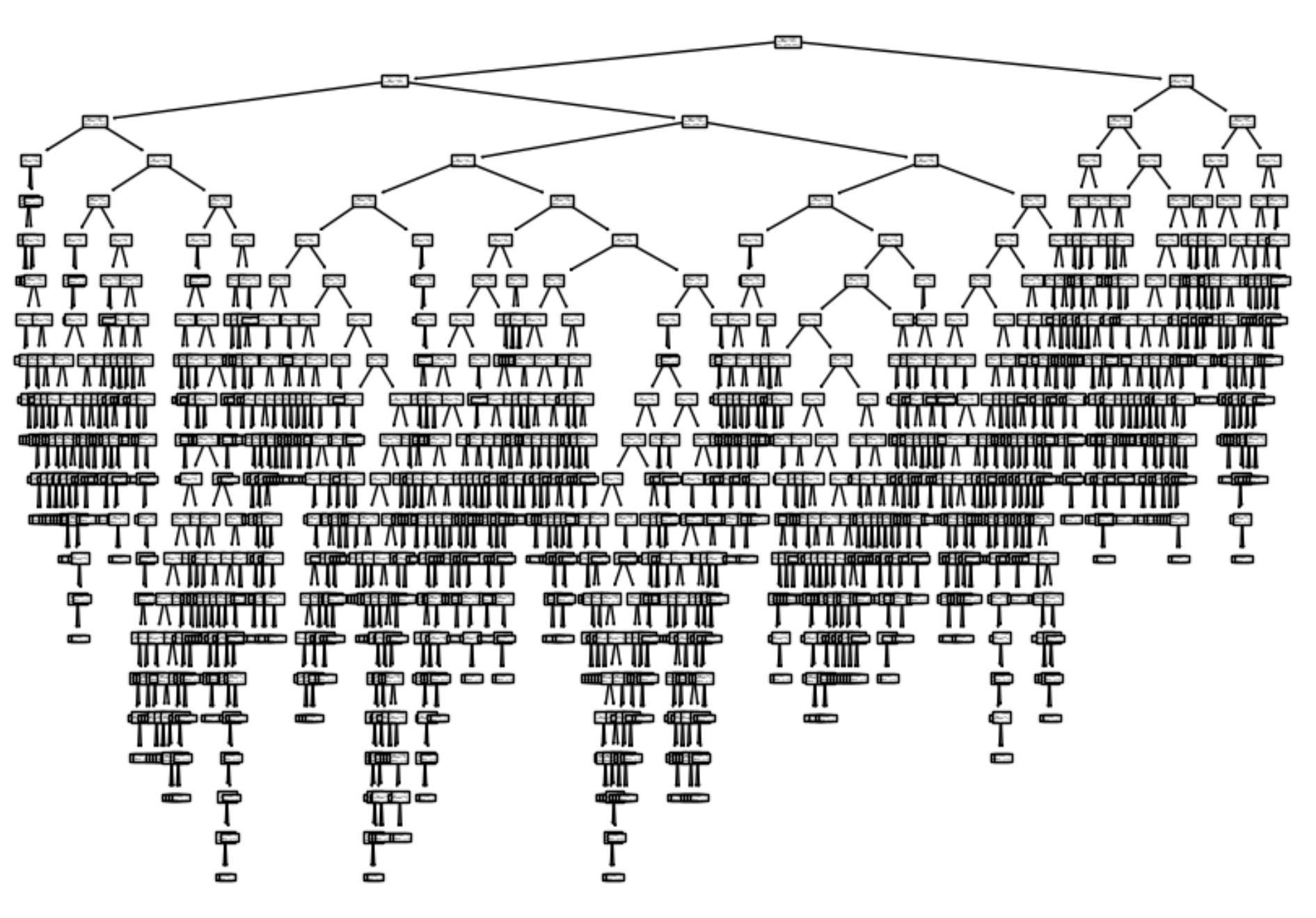
결정 트리

- 루프 노드: 맨 위의 노드
- 리프 노드: 맨 아래 끝에 달린 노드
- 노드: 훈련 데이터의 특성에 대한 테스트를 표현
- 표준화 전처리를 할 필요가 없음

In [10]:
```python
from sklearn.tree import DecisionTreeClassifier
dt=DecisionTreeClassifier(random_state=42)
dt.fit(train_scaled,train_target)
print(dt.score(train_scaled,train_target))
print(dt.score(test_scaled,test_target))
```

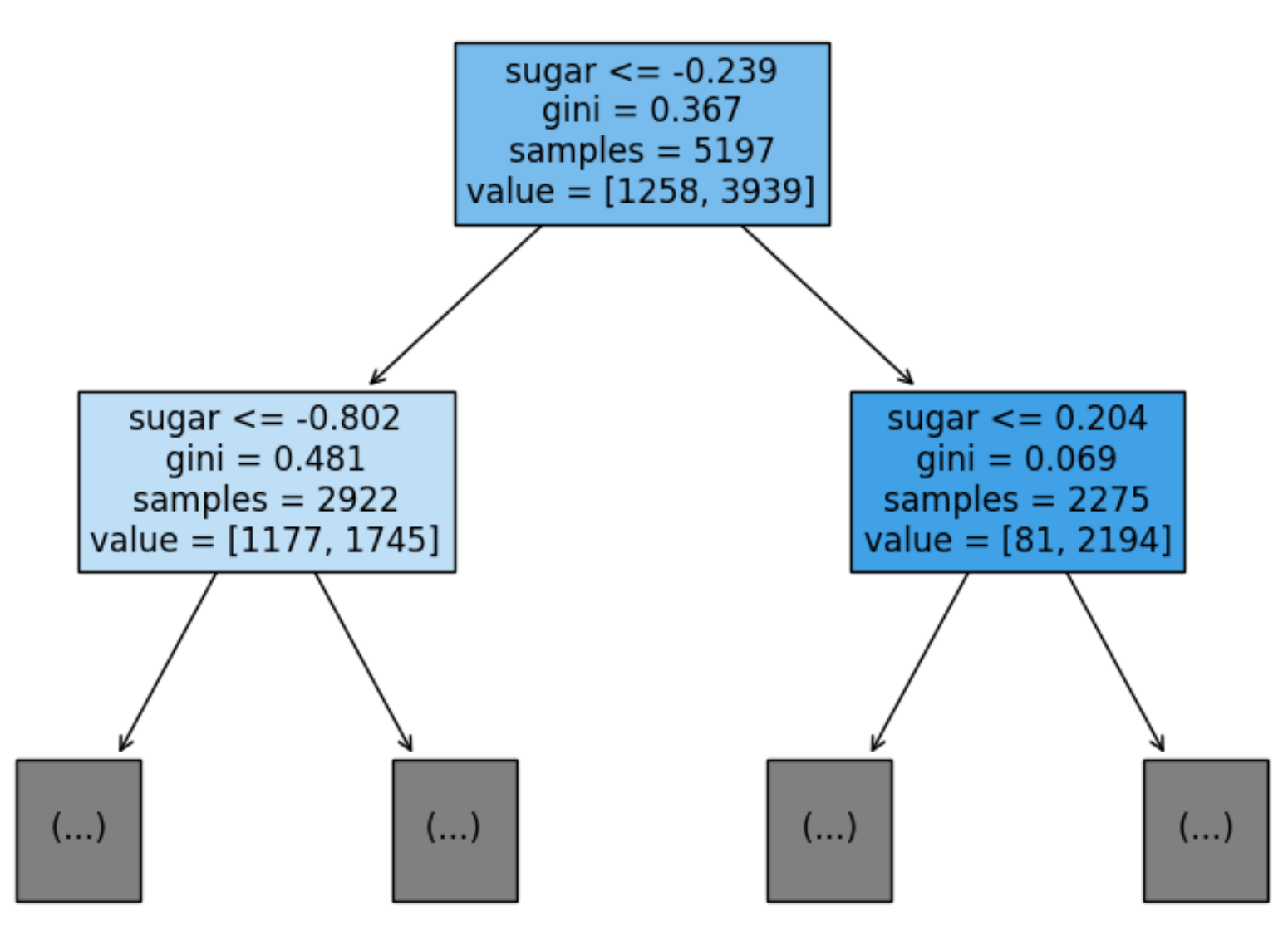
```
0.996921300750433
0.8592307692307692
```

In [11]:
```python
import matplotlib.pyplot as plt
from sklearn.tree import plot_tree
plt.figure(figsize=(10,7))
plot_tree(dt)
plt.show()
```



In [12]:
```python
plt.figure(figsize=(10,7))

# max_depth 매개변수는 트리의 깊이를 제한할 수 있음
# filled 매개변수는 노드의 색을 칠할 수 있음
# feature_names 매개변수는 특성의 이름을 전달함
# 노드의 설명은 순서대로 테스트 조건, 불순도, 총 샘플 수, 클래스별 샘플 수 임
# 왼쪽이 yes, 오른쪽이 no임
plot_tree(dt,max_depth=1, filled=True, feature_names=["alcohol","sugar","pH"])
plt.show()
```



불순도

- $\text{지니불순도} = (1 - (\text{음성클래스비율}^2 + \text{양성클래스비율}^2))$
- 지니 불순도가 0.5일 때 최악이고, 0에 가까울 수록 좋음. 0인 노드를 순수 노드라 함
- 결정 트리 모델은 부모 노드와 자식 노드의 불순도 차이가 가능한 크도록 트리를 성장시킴
- $\text{불순도차이(정보이득)} = (\text{왼쪽노드샘플수}/\text{부모의샘플수}) \times \text{왼쪽노드불순도} - (\text{오른쪽노드샘플수}/\text{부모의샘플수}) \times \text{오른쪽노드불순도}$
- 엔트로피 불순도도 있음

In [13]:
```python
print(1-(1258/5197)**2-(3939/5197)**2)
```

```
0.3669367279393918
```

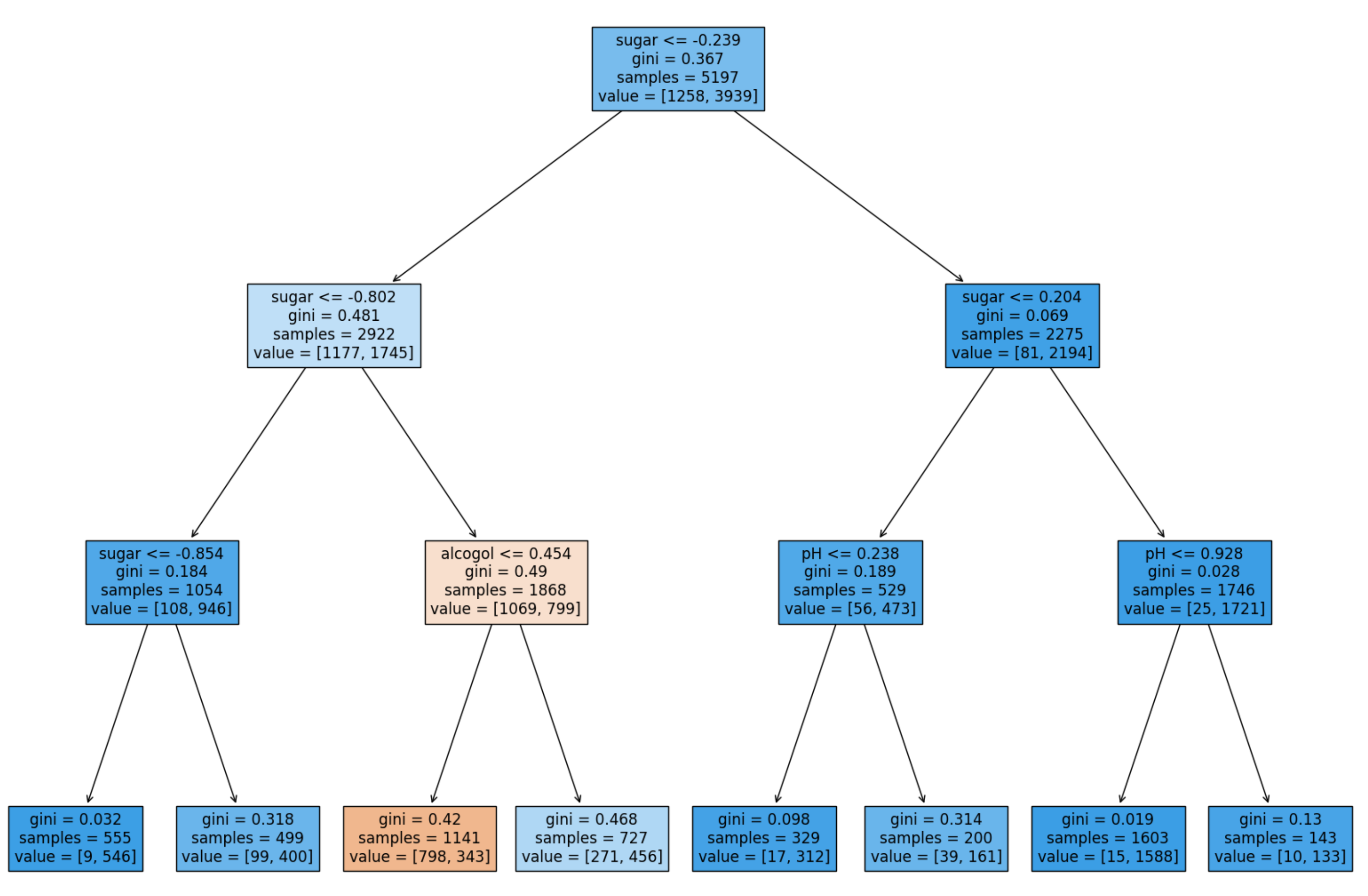
가지치기

- 가지치기를 하지 않으면 과대적합되기 쉬움

In [14]:
```python
dt=DecisionTreeClassifier(max_depth=3,random_state=42)
dt.fit(train_scaled,train_target)
print(dt.score(train_scaled,train_target))
print(dt.score(test_scaled,test_target))
```

```
0.8454877814123533
0.8415384615384616
```

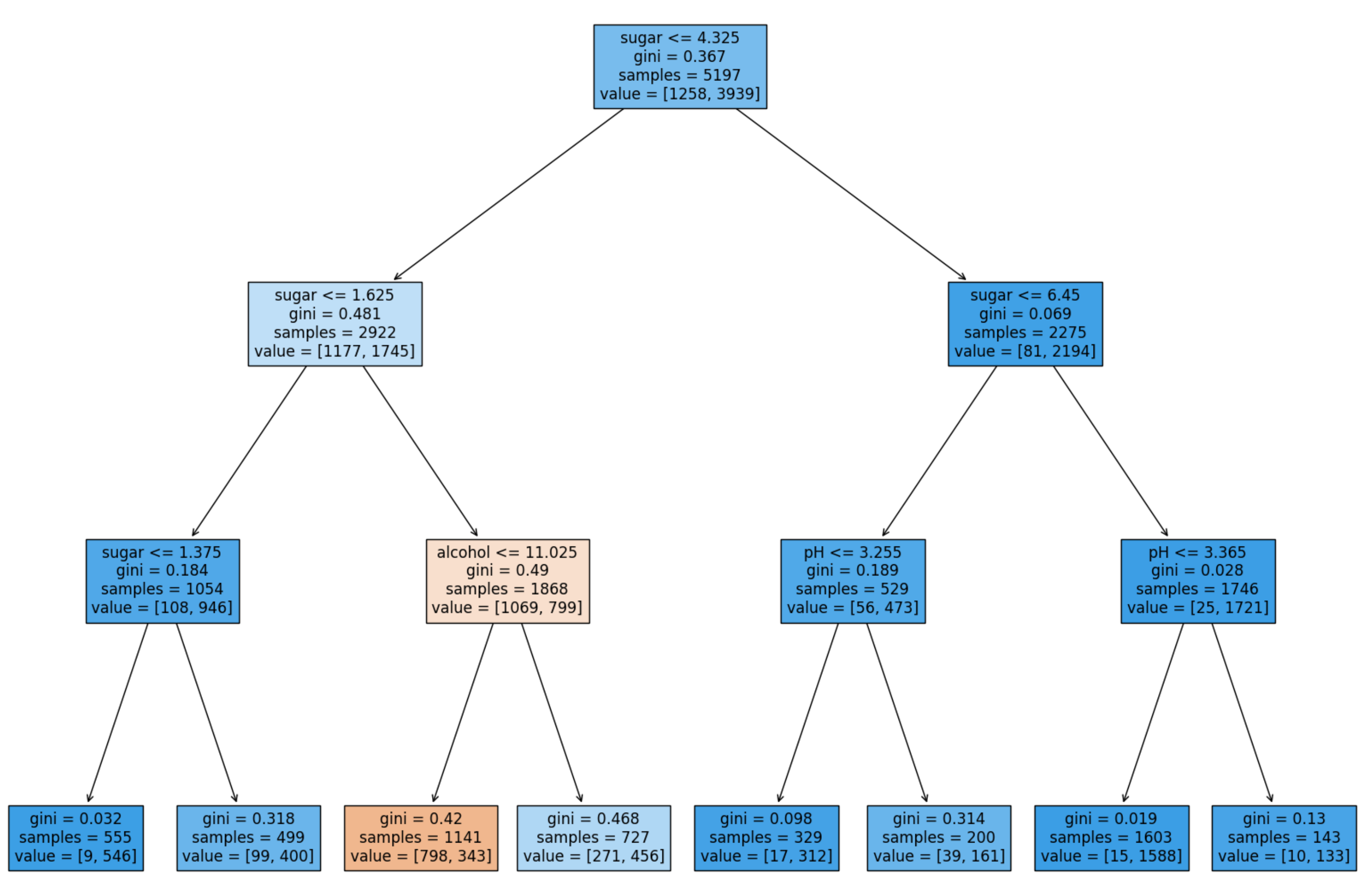
In [15]:
```python
plt.figure(figsize=(20,15))
plot_tree(dt,filled=True, feature_names=["alcogol","sugar","pH"])
plt.show()
```



In [16]:
```python
dt.fit(train_input,train_target)
print(dt.score(train_input,train_target))
print(dt.score(test_input,test_target))
```

```
0.8454877814123533
0.8415384615384616
```

In [17]:
```python
plt.figure(figsize=(20,15))
plot_tree(dt,filled=True,feature_names=["alcohol","sugar","pH"])
plt.show()
```



In [18]:
```python
print(dt.feature_importances_) # 어떤 튺성이 가장 유형한지 알려줌
```

```
[0.12345626 0.86862934 0.0079144 ]
```

교차 검증과 그리드 서치

- 테스트 세트를 많이 사용하면, 테스트 세트로 일반화 성능을 예측하는 꼴이 됨

검증 세트

- 검증 세트를 활용해 가장 좋은 모델을 고르고, 마지막에 테스트 세트에서 최종 점수를 평가함

In [19]:
```python
import pandas as pd
wine=pd.read_csv("wine.csv")
data=wine[["alcohol","sugar","pH"]].to_numpy()
target=wine["class"].to_numpy()
```

In [20]:
```python
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)
```

In [21]:
```python
sub_input,val_input,sub_target,val_target=train_test_split(train_input,train_target,test_size=0.2,random_state=42)
```

In [22]:
```python
print(sub_input.shape,val_input.shape,test_input.shape)
```

```
(4157, 3) (1040, 3) (1300, 3)
```

In [23]:
```python
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
```

교차 검증

- 검증 세트를 떼어 내어 평가하는 과정을 여러 번 반복하고 이 점수들을 평균하여 최종 검증 점수를 얻음
- train_test_split() 함수는 전체 데이터를 섞은 후 나눠주기 때문에, 교차 검증시 훈련 세트를 섞을 필요가 없음
- 그러나 만약 교차 검증시 훈련 세트를 섞으려면 분할기를 지정해야함

In [24]:
```python
# croee_validate() 함수는 기본적으로 5-폴드 교차 검증을 함
# fit_time,score_time,test_score(검증 폴드의 점수)을 반환함
from sklearn.model_selection import cross_validate
scores=cross_validate(dt,train_input,train_target)
print(scores)
```

```
{'fit_time': array([0.01322246, 0.01569223, 0.01927447, 0.00588369, 0.02796197]), 'score_time': array([0.0037272 , 0.00299025, 0.00216866, 0.        , 0.00600
362]), 'test_score': array([0.86923077, 0.84615385, 0.87680462, 0.84889317, 0.83541867])}
```

In [25]:
```python
import numpy as np
print(np.mean(scores["test_score"]))
```

```
0.855300214703487
```

In [26]:
```python
from sklearn.model_selection import StratifiedKFold
# 회귀 모델일 경우 KFold 분할기를 사용하고, 분류 모델일 경우 StratifiedKFole 분할기를 사용함

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

```
0.855300214703487
```

In [27]:
```python
splitter=StratifiedKFold(n_splits=10, shuffle=True,random_state=42) # 10-폴드 교차 검증 실시
scores=cross_validate(dt,train_input,train_target,cv=splitter)
print(np.mean(scores["test_score"]))
```

```
0.8574181117533719
```

하이퍼파라미터 튜닝

- 그리드 서치 사용
- GridSearchCV 클래스는 하이퍼파라미터 탐색과 교차 검증을 한 번에 수행함

In [28]:
```python
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) # n_jobs는 사용할 CPU 코어 수를 지정함. -1이면 모든 코어 사
```

In [29]:
```python
gs.fit(train_input,train_target)
# 실행한 모든 모델 중에서 검증 점수가 가장 높은 모델의 매개변수 조합으로 전체 훈련 세트에서 자동으로 다시 모델을 훈련함
dt=gs.best_estimator_
print(dt.score(train_input,train_target))
print(gs.best_params_)
```

```
0.9615162593804117
{'min_impurity_decrease': 0.0001}
```

In [30]:
```python
print(gs.cv_results_["mean_test_score"])
```

```
[0.86819297 0.86453617 0.86492226 0.86780891 0.86761605]
```

In [31]:
```python
best_index=np.argmax(gs.cv_results_["mean_test_score"])
print(gs.cv_results_["params"][best_index])
```

```
{'min_impurity_decrease': 0.0001}
```

1. 먼저 탐색할 매개변수를 지정함
2. 그다음 훈련 세트에서 그리드 서치를 수행하여 최상의 평균 검증 점수가 나오는 매개변수 조합을 찾음. 이 조합은 그리드 서치 객체에 저장됨
3. 그리드 서치는 최상의 매개변수에서 전체 훈련 세트를 사용해 최종 모델을 훈련함. 이 모델도 그리드 서치 객체에 저장됨

In [32]:
```python
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)}

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

```
{'max_depth': 14, 'min_impurity_decrease': 0.0004, 'min_samples_split': 12}
```

In [33]:
```python
print(np.max(gs.cv_results_["mean_test_score"]))
```

```
0.8683865773302731
```

랜덤 서치

- 매개변수의 값의 범위나 간격을 미리 정하기 어려움
- 랜덤 서치에는 매개변수 값의 목록을 전달하는 것이 아니라 매개변수를 샘플링할 수 있는 확률 분포 객체를 전달함

In [34]:
```python
from scipy.stats import uniform,randint
rgen=randint(0,10)
rgen.rvs(10)
```

Out[34]:
```
array([6, 8, 2, 2, 6, 8, 2, 3, 6, 4], dtype=int64)
```

In [35]:
```python
np.unique(rgen.rvs(1000),return_counts=True)
```

Out[35]:
```
(array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=int64),
 array([104, 112, 101, 100,  99,  93,  99,  97, 109,  86], dtype=int64))
```

In [36]:
```python
ugen=uniform(0,1)
ugen.rvs(10)
```

Out[36]:
```
array([0.11325921, 0.71383778, 0.34450042, 0.23339286, 0.97805413,
       0.95720056, 0.68797648, 0.81293759, 0.35504935, 0.59273014])
```

In [37]:
```python
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)}
```

In [38]:
```python
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_)
print(np.max(gs.cv_results_["mean_test_score"]))
```

```
{'max_depth': 39, 'min_impurity_decrease': 0.00034102546602601173, 'min_samples_leaf': 7, 'min_samples_split': 13}
0.8695428296438884
```

In [39]:
```python
dt=gs.best_estimator_
print(dt.score(test_input,test_target))
```

```
0.86
```

트리의 앙상블

- 정형 데이터: 어떤 구조로 되어 있다는 뜻. csv, 데이터베이스, 엑셀에 저장하기 쉬움
- 비정형 데이터: 데이터베이스나 엑셀로 표현하기 어려운 것들
- 앙상블 학습: 정형 데이터를 다루는 데 가장 뛰어난 성과를 내는 알고리즘

랜덤 포레스트

- 앙상블 학습의 대표 주자이며 안정적인 성능을 가짐
- 결정 트리를 랜덤하게 만들어 결정 트리의 숲을 만듬. 그리고 각 결정 트리의 예측을 사용해 최종 예측을 만듬
- 부트스트램 샘플: 중복적으로 샘플을 뽑아서 만든 샘플
- 각 노드를 분할할 때 전체 특성 중에서 일부 특성을 무작위로 고른 다음 최선의 분할을 찾음
- 과대적합되는 것을 막아주고 검증 세트와 테스트 세트에서 안정적인 성능을 얻을 수 있음
- RandomForestClassifier 클래스는 기본적으로 100개의 결정 트리를 사용함

In [40]:
```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
wine=pd.read_csv("wine.csv")
data=wine[["alcohol","sugar","pH"]].to_numpy()
target=wine["class"].to_numpy()
train_input,test_input,train_target,test_target=train_test_split(data,target,test_size=0.2,random_state=42)
```

In [41]:
```python
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
```

In [42]:
```python
rf.fit(train_input, train_target)
print(rf.feature_importances_)
```

```
[0.23167441 0.50039841 0.26792718]
```

- OOB(부트스트랩 샘플에 포함되지 않은 샘플)샘플로 결정 트리를 평가할 수 있음. 검증 세트의 역할을 함

In [43]:
```python
rf=RandomForestClassifier(oob_score=True,n_jobs=-1,random_state=42)
rf.fit(train_input,train_target)
print(rf.oob_score_)
```

```
0.8934000384837406
```

엑스트라 트리

- 랜덤 포레스트와 매우 비슷하게 동작
- 랜덤 포레스트와의 차이점은 부트스트랩 샘플을 사용하지 않고 전체 훈련 세트를 사용함
- 노드를 분할할 때 가장 좋은 분할을 찾는 것이 아니라 무작위로 분할함 -> 더 많은 트리가 필요함
- 성능이 낮아지겠지만 많은 트리를 앙상블 하기 때문에 과대적합을 막고 검증 세트의 점수를 높이는 효과가 있음

In [44]:
```python
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
```

In [45]:
```python
et.fit(train_input,train_target)
print(et.feature_importances_)
```

```
[0.20183568 0.52242907 0.27573525]
```

그레디언트 부스팅

- 깊이가 얕은 결정 트리를 사용하여 이전 트리의 오차를 보완하는 방식으로 앙상블 하는 방법
- GradientBoostingClassifier은 기본적으로 깊이가 3인 결정 트리를 100개 사용함
- 경사 하강법을 사용하여 트리를 앙상브레 추가함
- 분류에서는 로지스틱 손실 함수를 사용하고 회귀에서는 평균 제곱 오차 함수를 사용함
- 트리의 개수를 늘려도 과대적합에 매우 강함
- 학습률의 기본값은 0.1임. 클 수록 복잡하고 훈련 세트에 과대적합된 모델을 얻을 수 있음

In [46]:
```python
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
```

In [47]:
```python
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
```

In [48]:
```python
gb.fit(train_input,train_target)
print(gb.feature_importances_)
```

```
[0.15853457 0.68010884 0.1613566 ]
```

히스토그램 기반 그레이디언트 부스팅

- 정형 데이터를 다루는 머신러닝 알고리즘 중에 가장 인기가 높은 알고리즘
- 입력 특성을 256개의 구간으로 나눔 -> 노드를 분할할 때 최적의 분할을 매우 빠르게 찾을 수 있음
- 입력에 누락된 특성이 있더라도 이를 따로 전처리할 필요가 없음
- n_estimators 대신에 max_iter 매개변수를 사용함
- 과대적합을 잘 억제하면서 그레이디언트 부스팅보다 조금 더 높은 성능을 제공함

In [49]:
```python
from sklearn.ensemble import HistGradientBoostingClassifier
hgb=HistGradientBoostingClassifier(random_state=42)
scores=cross_validate(hgb,train_input,train_target,return_train_score=True)
print(np.mean(scores["train_score"]),np.mean(scores["test_score"]))
```

```
0.9321723946453317 0.8801241948619236
```

In [50]:
```python
# 특성 중요도 확인
from sklearn.inspection import permutation_importance

hgb.fit(train_input,train_target)

# 특성을 하나씩 랜덤하게 섞어서 모델의 성능이 변화하는지를 관찰하여 어떤 특성이 중요한지 계산함
# n_repeats 매개변수는 랜덤하게 섞을 횟수를 지정함. 기본값은 5임
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]
```

In [51]:
```python
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     ]
```

In [52]:
```python
hgb.score(test_input,test_target)
```

Out[52]:
```
0.8723076923076923
```

- 그레이디언트 부스팅 알고리즘을 구현한 대표적인 라이브러리는 XGBoost임