

Chapter 7. 양상분류 학습과 랜덤 포레스트

7.0 양상분류 학습 (Ensemble learning)

대중의 지혜(wisdom of the crowd)

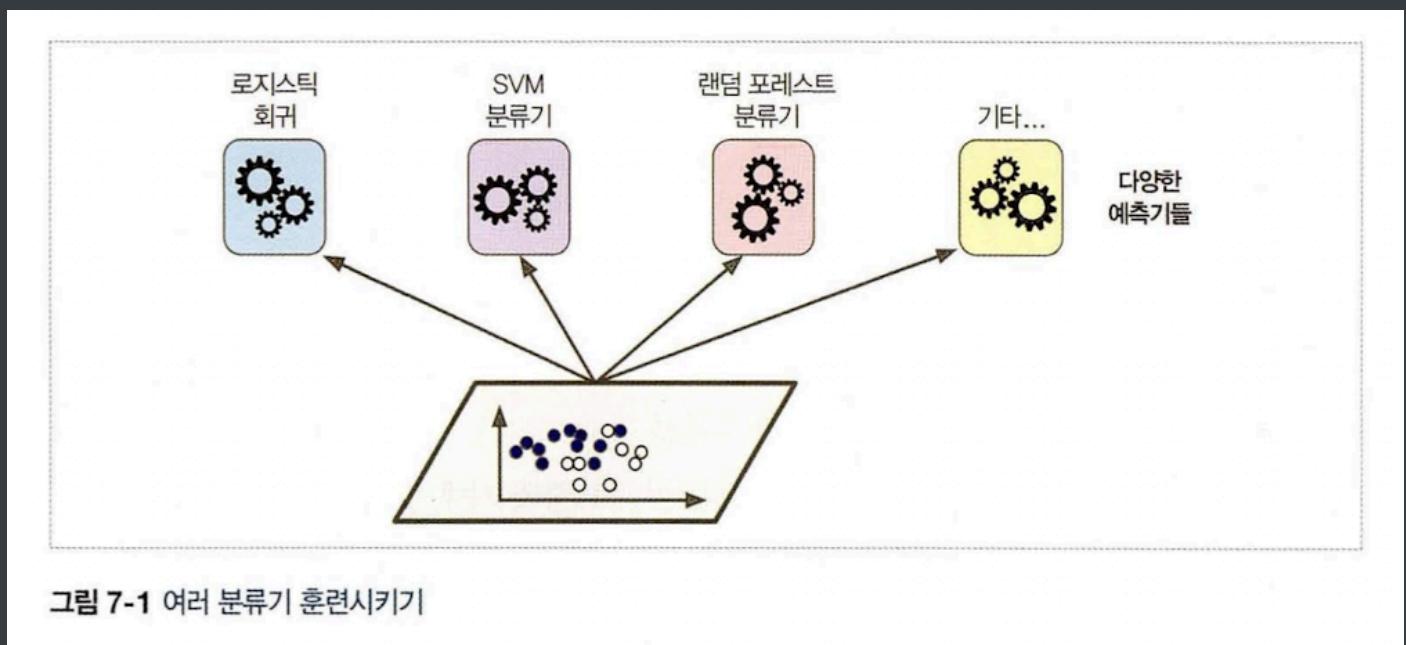
다양한 예측기(분류 혹은 회귀 모델)로부터 예측을 수집해 예측의 성능을 높임

보팅, 배깅, 부스팅, 스태킹

7.1 투표 기반 분류기

hard voting(직접 투표) 분류

- 더 좋은 분류기를 만드는 매우 간단한 방법: 각 분류기의 예측을 모아, 가장 많이 선택된 클래스로 결정



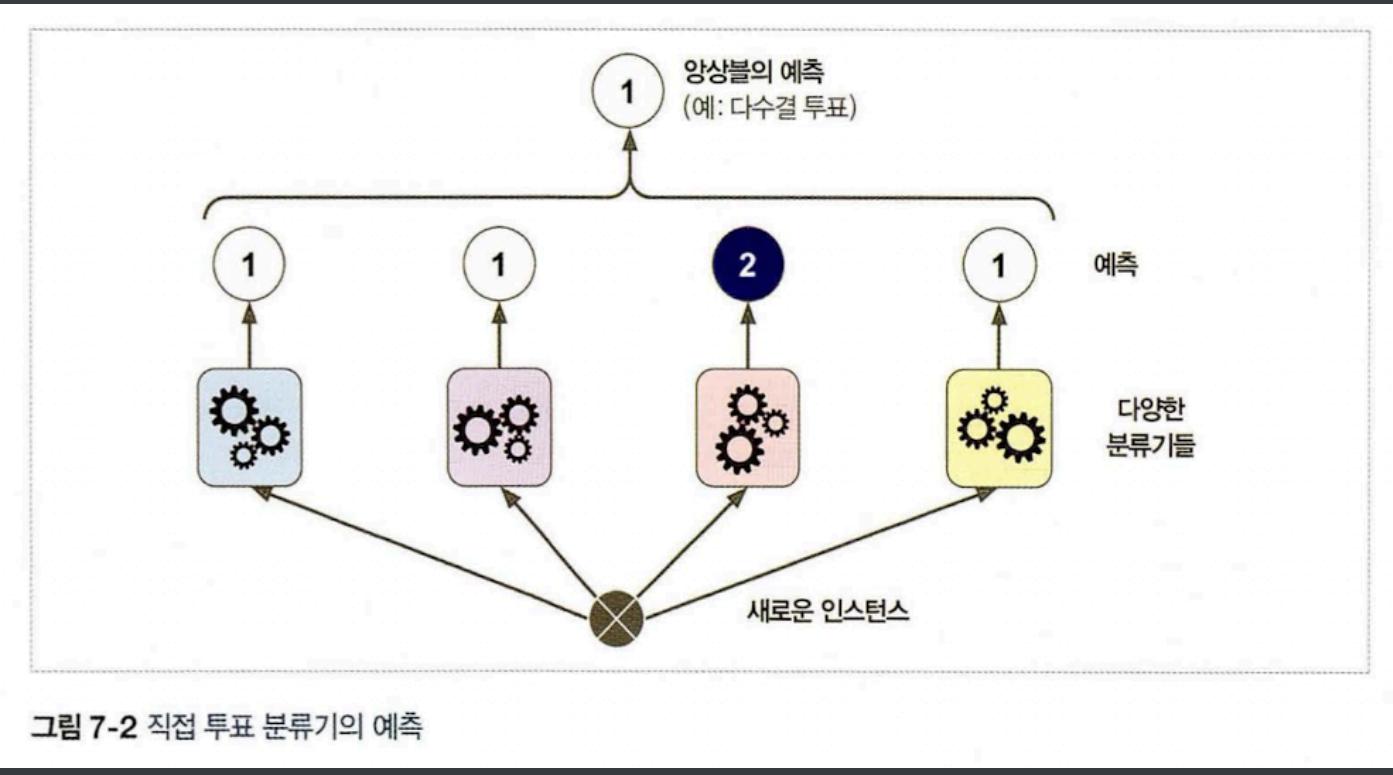


그림 7-2 직접 투표 분류기의 예측

- 개별 분류기 중 가장 뛰어난 것보다 정확도가 높은 경우가 대다수
 - 학습기가
 1. 충분하게 많고
 2. 다양하면서
 3. 가능한 서로 독립적일 때
 - 양상불을 통해 높은 정확도를 도출해낼 수 있음
 - 큰 수의 법칙(law of large numbers)
 - 사이킷런의 투표 기반 분류기(VotingClassifier)를 활용한 예시 (hard voting)

```
# 로지스틱 회귀 모델
log_clf = LogisticRegression(solver="lbfgs", random_state=42)

# 랜덤 포레스트 모델
rnd_clf = RandomForestClassifier(n_estimators=100, random_state=42)

# svm 모델
svm_clf = SVC(gamma="scale", random_state=42)

# 투표 기반 분류기 생성
voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')
```

```

# fitting
voting_clf.fit(X_train, y_train)

# 분류 시행
from sklearn.metrics import accuracy_score

for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
'''

결과
LogisticRegression 0.864
RandomForestClassifier 0.896
SVC 0.896
VotingClassifier 0.912 # 투표 기반 분류기의 성능이 제일 높음
'''
```

soft voting(간접 투표) 분류

- 모든 분류기가 클래스의 확률을 예측할 수 있으면(predict_proba() method 포함)
- 개별 분류기의 예측을 평균으로 내어 확률이 가장 높은 클래스를 예측 가능
- 직접 투표 방식보다 높은 성능
- 더 느린 훈련 속도
- 사이킷런의 투표 기반 분류기(VotingClassifier)를 활용한 예시 (soft voting)

```

log_clf = LogisticRegression(solver="lbfgs", random_state=42)
rnd_clf = RandomForestClassifier(n_estimators=100, random_state=42)
# svm 모델에 probability=True로 매개변수 설정
svm_clf = SVC(gamma="scale", probability=True, random_state=42)

# 간접 투표 분류기 생성
soft_voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
```

```

voting='soft')

# fitting
soft_voting_clf.fit(X_train, y_train)

# 분류 시행
from sklearn.metrics import accuracy_score

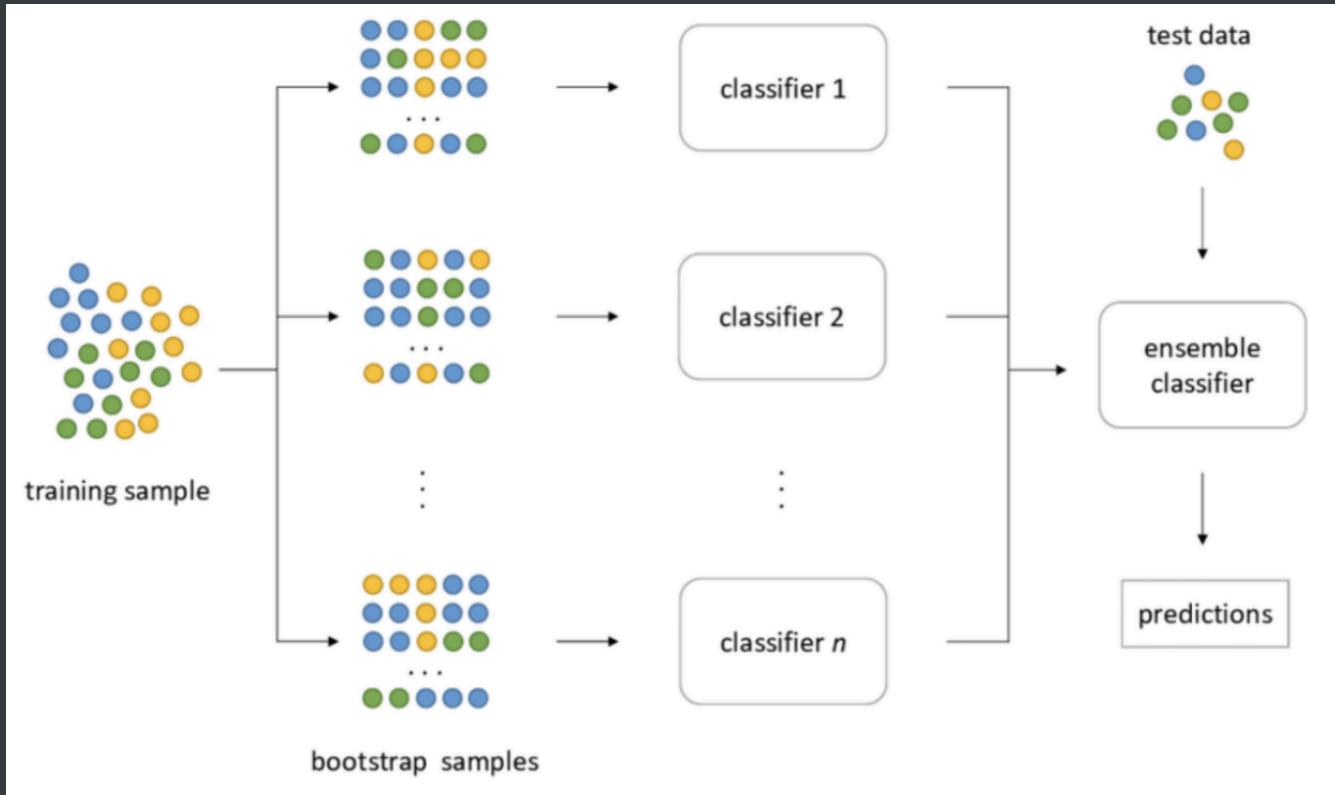
for clf in (log_clf, rnd_clf, svm_clf, soft_voting_clf):
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(clf.__class__.__name__, accuracy_score(y_test, y_pred))

...
결과
LogisticRegression 0.864
RandomForestClassifier 0.896
SVC 0.896
VotingClassifier 0.92 # 향상된 성능
...

```

7.2 배깅과 페이스팅

- 다양한 분류기를 만드는 또다른 방법: 같은 알고리즘을 사용하고 훈련 세트의 서브셋을 무작위로 구성해 분류기를 각기 다르게 학습 시키는 것
- 같은 훈련 샘플을 여러 개의 예측기에 걸쳐 사용 가능
- 샘플링(sampling): 다양한 sampling dataset으로 다양한 분류기 생성
- 배깅(bagging- bootstrap aggregating의 줄임말)



- 훈련 세트에서 중복을 허용하여 샘플링하는 방식
- 한 예측기를 위해 같은 훈련 샘플을 여러 번 샘플링 가능

- 페이스팅(pasting)
 - 중복을 허용하지 않고 샘플링하는 방식

- 모든 예측기가 훈련을 마치면 양상들은 모든 예측기의 예측을 모아 새로운 샘플에 대한 예측 생성
- 분류의 경우, 하드 보팅의 경우와 같이 가장 많은 예측 결과가 나온 모델 선택
- 회귀의 경우 평균값 계산
- 각 개별 예측기는 원본 훈련세트보다 더 크게 편향되어있지만,
- aggregation시 편향과 분산 모두 감소 => 하나의 예측기로 예측할때보다 분산 줄어듦 (편향 비슷)
- 사이킷런에서의 배깅과 페이스팅(BaggingClassifier, BaggingRegressor) 예시

```

from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

# 배깅 분류기 생성
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(random_state=42), n_estimators=500,
    max_samples=100, bootstrap=True, random_state=42)
bag_clf.fit(X_train, y_train)

```

```
y_pred = bag_clf.predict(X_test)

from sklearn.metrics import accuracy_score
print(accuracy_score(y_test, y_pred))
```

'''

결과

0.904 # 배깅의 정확도가 더 높음

'''

비교대상: 결정트리

```
tree_clf = DecisionTreeClassifier(random_state=42)
tree_clf.fit(X_train, y_train)
y_pred_tree = tree_clf.predict(X_test)
print(accuracy_score(y_test, y_pred_tree))
```

```
tree_clf = DecisionTreeClassifier(random_state=42)
tree_clf.fit(X_train, y_train)
y_pred_tree = tree_clf.predict(X_test)
print(accuracy_score(y_test, y_pred_tree))
```

'''

결과

0.856

'''

배깅 분류기를 사용했을 때 더 작은 분산을 가짐

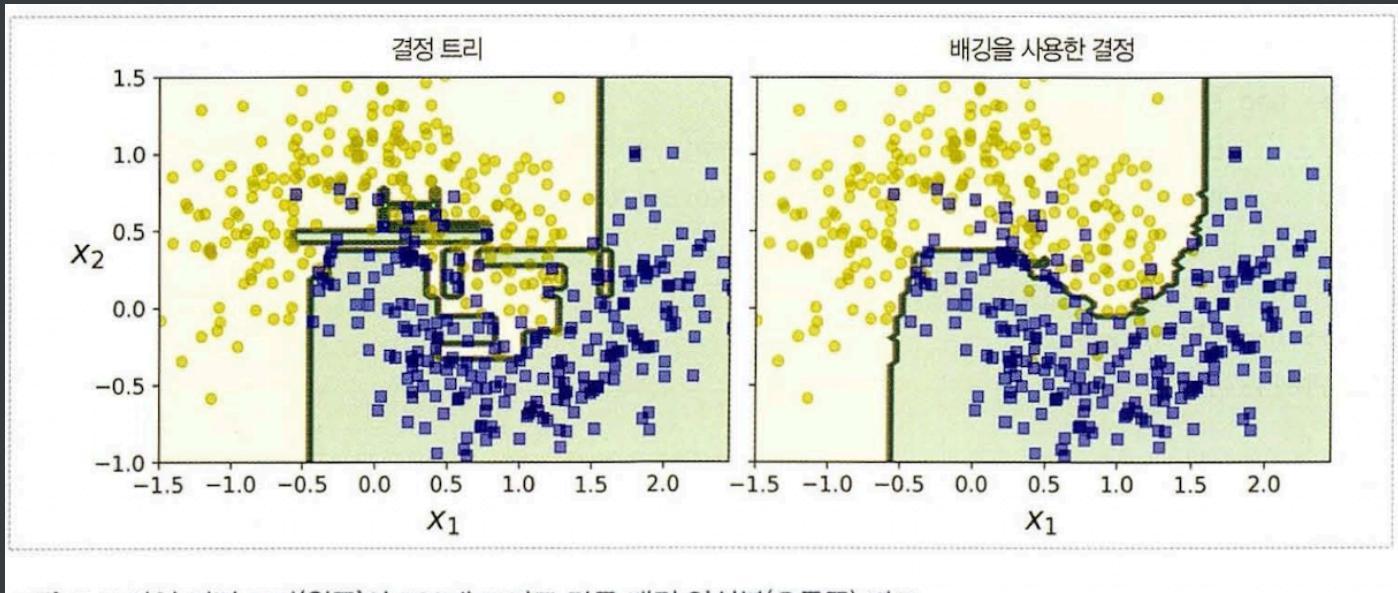


그림 7-5 단일 결정 트리(왼쪽)와 500개 트리로 만든 배깅 양상별(오른쪽) 비교

배깅 vs 페이스팅

- 부트스트래핑: 각 예측기가 학습하는 서브셋에 다양성을 증대
 - 배깅이 페이스팅보다 편향이 조금 더 높음
 - 다양성을 추가해 예측기 간 상관관계를 줄이므로 분산을 감소
- 전반적으로 배깅을 더 선호
- 여유가 있을 때 교차검증으로 배깅과 페이스팅 모두 평가해도 좋을듯

oob 평가

- BaggingClassifier는 기본값으로 중복을 허용(bootstrap=True)
- 평균적으로 각 예측기의 63% 정도만 샘플링
- 선택되지 않은 훈련 샘플 37%의 예측기: oob(out of bag) 샘플
- 사이킷런에서의 oob 평가

```
# oob_score=True

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    bootstrap=True, n_jobs=-1, oob_score=True)

bag_clf.fit(X_train, y_train)
```

```
bag_clf.obb_score_
...
결과
0.90133333333333 # 비슷하지만 테스트 결과에 못미침
...

from sklearn.metrics import accuracy_score
y_pred = bag_clf.predict(X_test)
accuracy_score(y_test, y_pred)

...
결과
0.912222222222222
...
```

7.3 랜덤 패치와 랜덤 서브 스페이스 – 참고

- BaggingClassifier는 feature 샘플링도 지원
 - max_features, bootstrap_features 두 하이퍼파라미터로 조절
 - sample이 아닌 feature에 대한 샘플링 => 각 예측기는 무작위로 선택한 입력 feature의 일부 분으로 훈련
- 매우 고차원의 데이터 셋을 다룰 때 유용 (이미지 등)
- 랜덤 패치 방식: 훈련 feature와 샘플을 모두 샘플링하는 방식
- 랜덤 서브스페이스 방식: 훈련 샘플은 모두 사용하고 feature만 샘플링하는 방식
- feature 샘플링은 더 다양한 예측기를 만들고 분산을 낮추지만 편향을 늘린다.

7.4 랜덤 포레스트

배깅 방법(또는 페이스팅)을 적용한 결정 트리의 양상을

부트스트랩 샘플과 랜덤한 후보 특성을 사용해 여러 결정 트리를 양상을

과대 적합을 방지, 모델의 높은 일반화 성능

max_samples 하이퍼파라미터를 활용, 훈련세트의 크기 지정

- 사이킷런에서의 랜덤 포레스트(RandomForestClassifier) 활용 예시

```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500,
max_leaf_nodes=16, random_state=42)
rnd_clf.fit(X_train, y_train)

y_pred_rf = rnd_clf.predict(X_test)

'''  
결과  
0.912  
'''
```

7.4.1 엑스트라 트리(익스트림 랜덤 트리) – 참고

- 랜덤포레스트는 트리를 만들 때 각 노드는 무작위로 특성의 서브셋을 만들어 분할에 사용
- 더욱 무작위하게 만들기 위해 최적의 임곗값을 찾는 대신,
- 후보 특성을 사용해 무작위로 분할한 다음 그 중 최상의 분할을 선택하는 변종(?)
- 사이킷런의 ExtraTreeClassifier 활용

7.4.2 feature 중요도

- 랜덤포레스트는 feature의 상대적 중요도를 측정하기 쉬움
- 특성을 선택할 때 어떤 특성이 중요한지 빠르게 확인 가능
- 사이킷런은 가중치 평균을 내 feature의 중요도를 측정

```
# feature importance
from sklearn.datasets import load_iris
iris = load_iris()
rnd_clf = RandomForestClassifier(n_estimators=500, random_state=42)
rnd_clf.fit(iris["data"], iris["target"])
```

```
for name, score in zip(iris["feature_names"],
rnd_clf.feature_importances_):
    print(name, score)

rnd_clf.feature_importances_
...
결과
sepal length (cm) 0.11249225099876375
sepal width (cm) 0.02311928828251033
petal length (cm) 0.4410304643639577
petal width (cm) 0.4233579963547682

array([0.11249225, 0.02311929, 0.44103046, 0.423358])
...
# petal length, petal width의 중요도가 훨씬 높음
```

7.5 부스팅

약한 학습기를 여러 개 연결하여 강한 학습기를 만드는 앙상블 방법

앞의 모델을 보완해 나가면서 예측기의 성능을 점진적으로 향상시키는 것

에이다 부스트, 그래디언트 부스팅

7.5.1 에이다 부스트

- 이전 모델이 과소적합했던 훈련 샘플의 가중치를 높여 예측기를 보완
- 보완된 예측기는 학습하기 어려운 샘플에 점점 fitting
- 경사하강법과 유사
- 앙상블에 예측기를 지속적으로 추가
- 모든 예측기가 훈련을 마치면 배깅과 페이스팅과 비슷한 방식으로 예측을 만듦
- 앞서 각 훈련 샘플에 다른 가중치를 부여 => 정확도에 따라 예측기마다 다른 가중치 적용

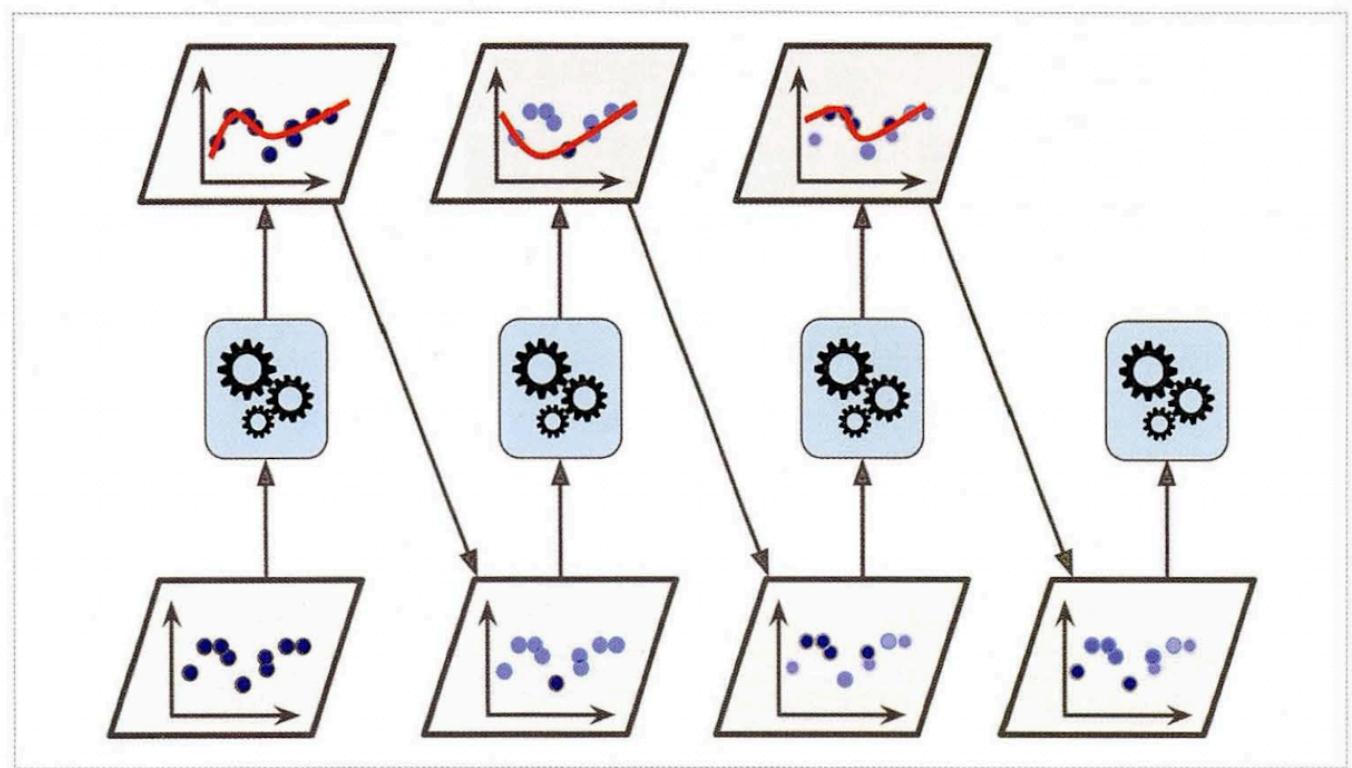


그림 7-7 샘플의 가중치를 업데이트하면서 순차적으로 학습하는 에이다부스트

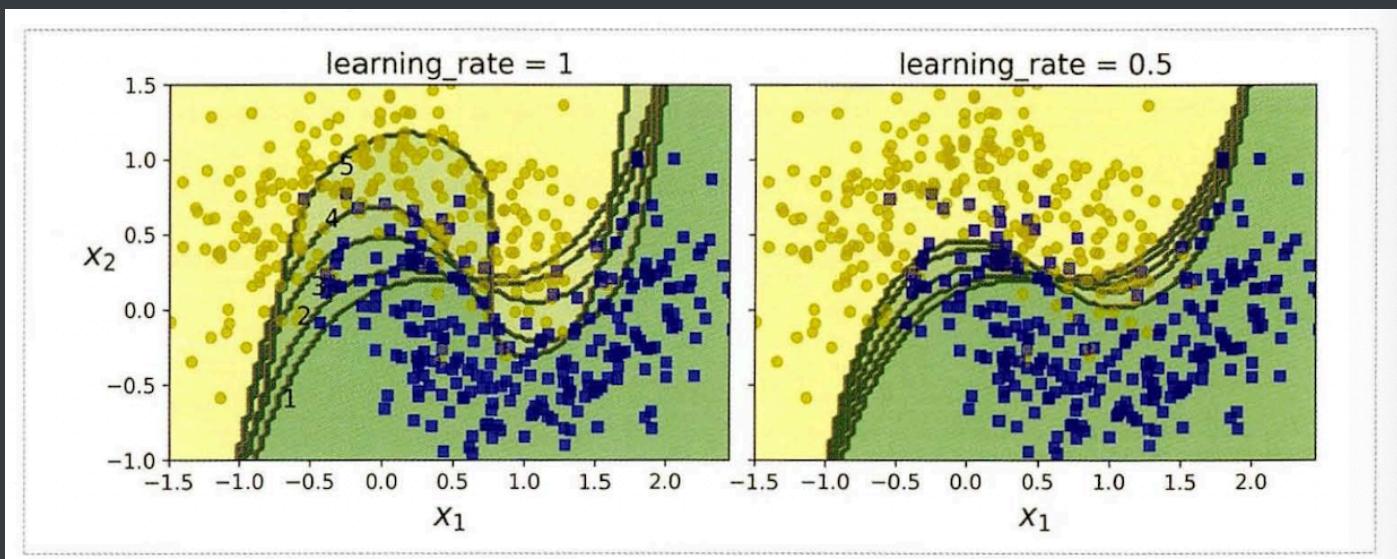


그림 7-8 연속된 예측기의 결정 경계²²

- 연속된 학습이 진행되는 만큼 병렬화(분할) 불가 => 배깅과 달리 낮은 확장성

식 7-1 j 번째 예측기의 가중치가 적용된 에러율

$$r_j = \frac{\sum_{i=1}^m w^{(i)}}{\sum_{i=1}^m \hat{y}_j^{(i)} \neq y^{(i)}} \quad \text{여기서 } \hat{y}_j^{(i)} \text{는 } i \text{번째 샘플에 대한 } j \text{ 번째 예측기의 예측}$$

예측기의 가중치 α_j 는 [식 7-2]를 사용해 계산됩니다. 여기서 η 는 학습률 하이퍼파라미터입니다(기본값 1).²³ 예측기가 정확할수록 가중치가 더 높아지게 됩니다. 만약 무작위로 예측하는 정도라면 가중치가 0에 가까울 것입니다. 그러나 그보다 나쁘면(즉, 무작위 추측보다 정확도가 낮으면) 가중치는 음수가 됩니다.²⁴

식 7-2 예측기 가중치

$$\alpha_j = \eta \log \frac{1 - r_j}{r_j}$$

그다음 에이다부스트 알고리즘이 [식 7-3]을 사용해 샘플의 가중치를 업데이트합니다. 즉, 잘못 분류된 샘플의 가중치가 증가됩니다.

식 7-3 가중치 업데이트 규칙

$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \hat{y}_j^{(i)} = y^{(i)} \text{일 때} \\ w^{(i)} \exp(\alpha_j) & \hat{y}_j^{(i)} \neq y^{(i)} \text{일 때} \end{cases}$$

여기서 $i = 1, 2, \dots, m$

그런 다음 모든 샘플의 가중치를 정규화합니다(즉, $\sum_{i=1}^m w^{(i)}$ 으로 나눕니다).

- 지정된 예측기 수에 도달하거나 완벽한 예측기가 만들어졌을 때 훈련 중지
- 가중치 합이 가장 큰 클래스의 예측 결과 선정

식 7-4 에이다부스트 예측

$$\hat{y}(\mathbf{x}) = \operatorname{argmax}_k \sum_{\substack{j=1 \\ \hat{y}_j(\mathbf{x})=k}}^N \alpha_j$$

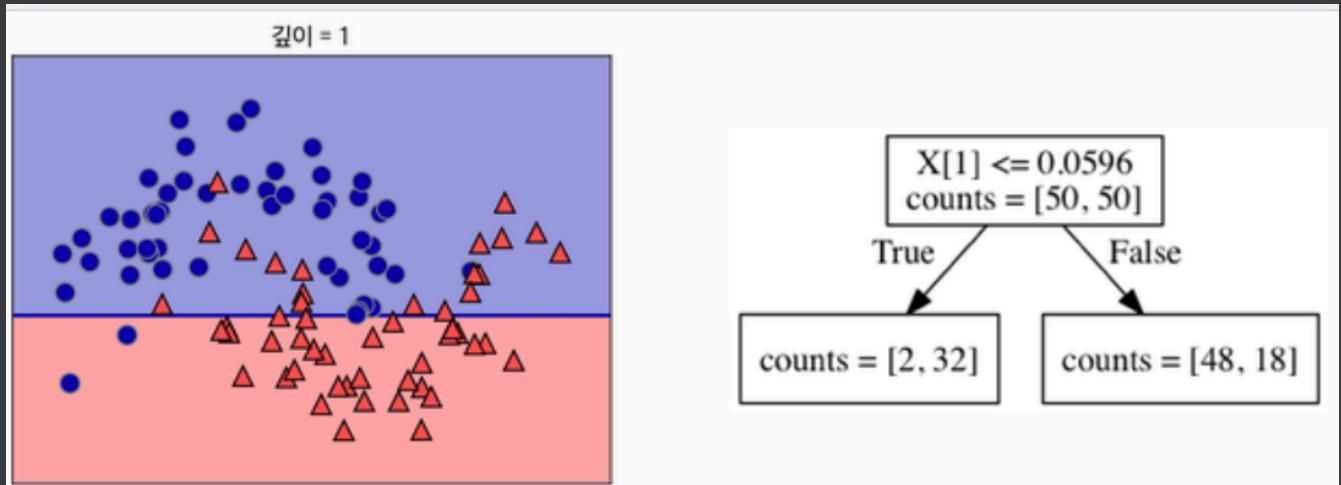
여기서 N 은 예측기 수

- 사이킷런에서의 에이다 부스트(AdaBoostClassifier) 활용 예시(Regressor또한 존재)

```
# max_depth=1의 결정트리
# 결정 노드 하나와 리프 두개로 이뤄짐
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm="SAMME.R", learning_rate=0.5, random_state=42)
ada_clf.fit(X_train, y_train)
accuracy(y_test, y_pred)

...
결과
0.904
...
```



7.5.2 그레디언트 부스팅(gradient boosting, GBM)

- 에이다 부스트처럼 양상블에 이전까지의 오차를 보정하도록 예측기를 순차적으로 추가
- 반복마다 샘플의 가중치를 수정하는 대신, 예측기가 만든 잔여 오차(residual error)에 새로운 예측기를 학습
- 사이킷런에서 gradient tree boosting(gradient boosted regression tree) 원리 파악

```
# DecisionTreeRegressor를 훈련 세트에 학습

from sklearn.tree import DecisionTreeRegressor

tree_reg1 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg1.fit(X, y)

# 예측 성공한 데이터를 빼준 잔여 오차 y2를 대상으로 회귀 모델 훈련
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg2.fit(X, y2)

# 반복 시행
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg3.fit(X, y3)

# 세 개의 트리를 포함하는 양상블 모델: 앞의 모든 트리의 예측의 합
```

```
y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2,
tree_reg3))
```

- 사이킷런의 GradientBoostingRegressor를 활용한 GRBT 앙상블 예시

```
from sklearn.ensemble import GradientBoostingRegressor

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3,
learning_rate=1.0, random_state=42)
gbrt.fit(X, y)
```

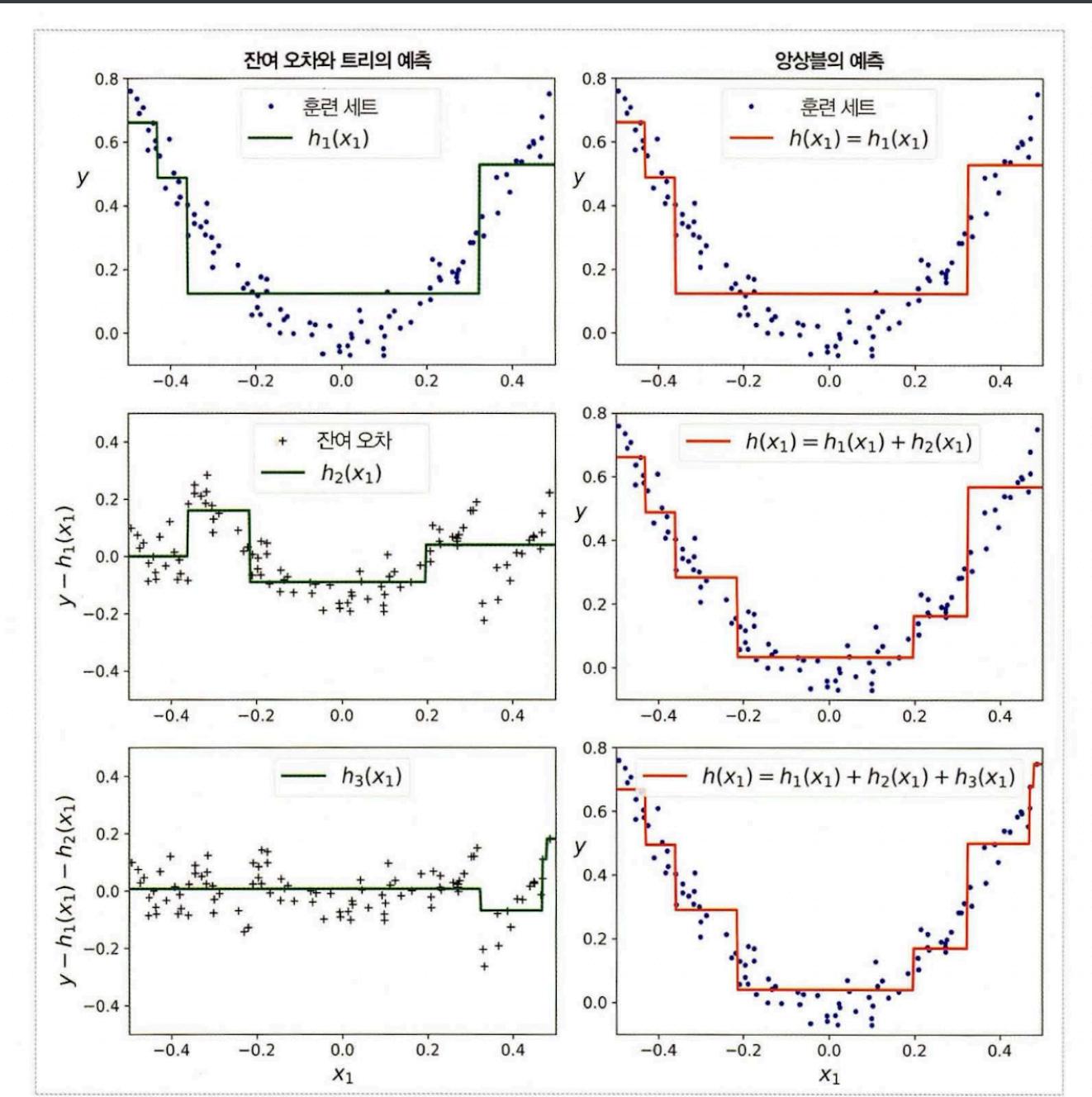
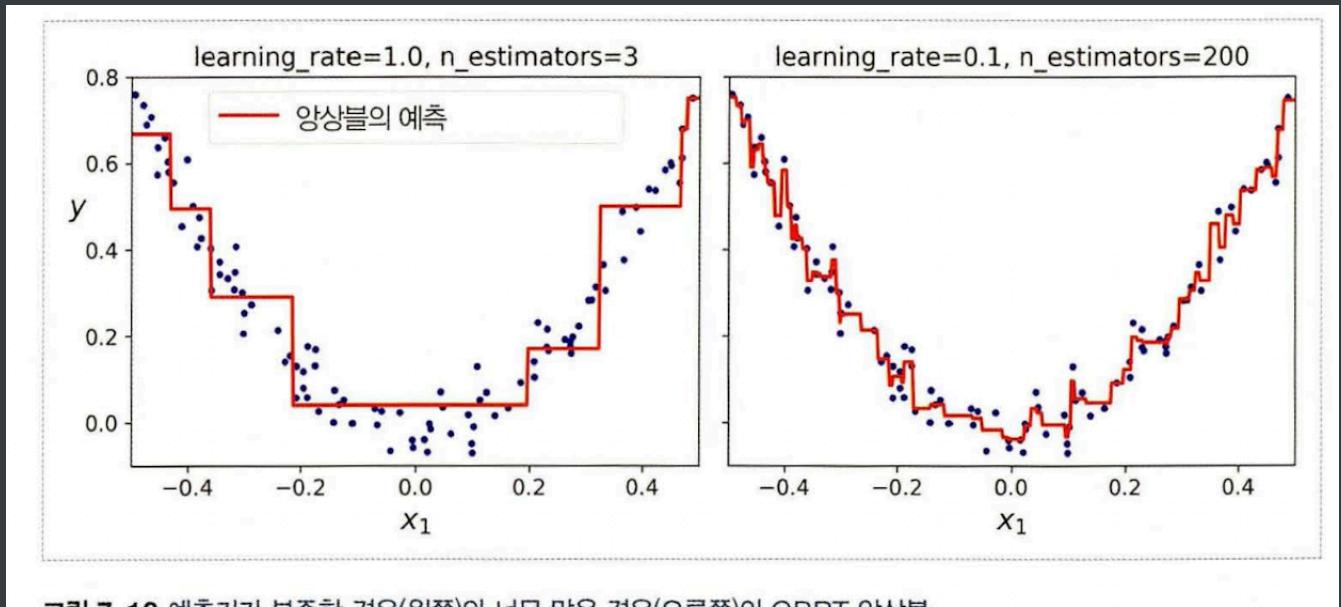


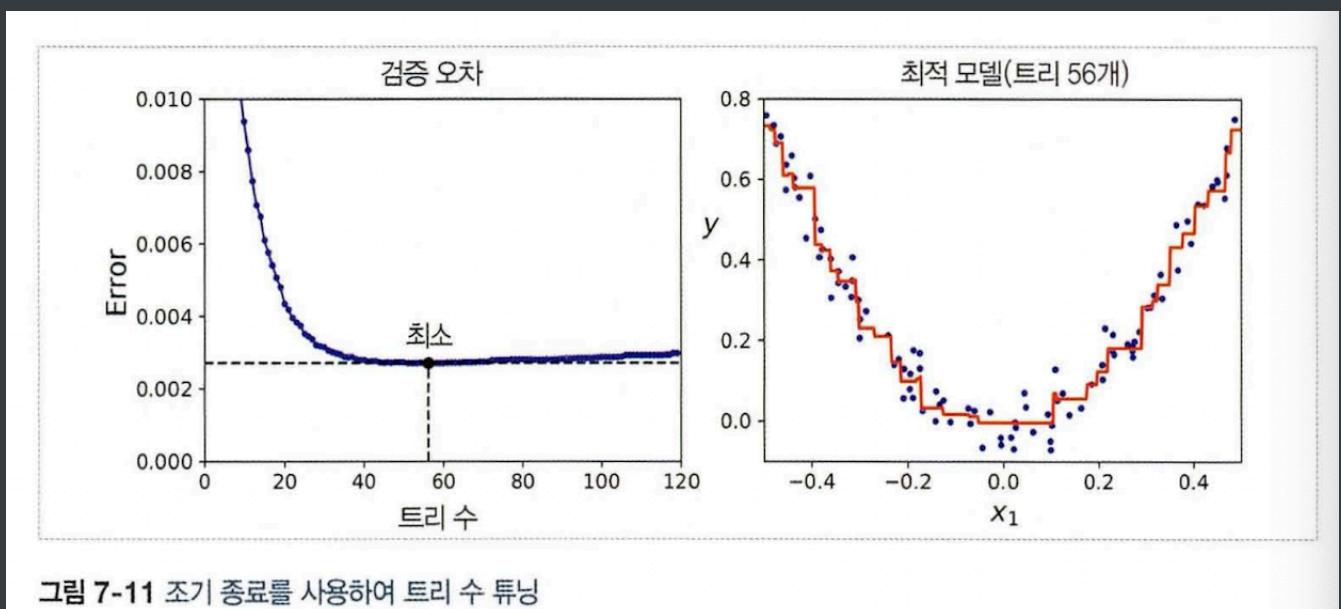
그림 7-9 이 그레이디언트 부스팅 그래프에서 첫 번째 예측기(왼쪽 위)가 평소와 같이 훈련됩니다. 그다음 연이은 예측기(왼쪽 중간, 왼쪽 아래)가 이전의 예측기의 잔여 오차에서 훈련됩니다. 오른쪽 열은 만들어진 양상불의 예측을 보여줍니다.

- n_estimators: 트리의 수
 - default=100
 - 트리가 너무 많아지면 훈련 세트에 과대 적합 가능성
- learning_rate 하이퍼파라미터로 각 트리의 기여 정도를 조절
 - default=0.1
 - 이전에 만든 트리의 오류에 기반해, 얼마나 많이 수정해나갈지
 - 낮게 설정시: 많은 트리 동반돼야 => 시간 지연, 좋은 성능 ; 축소(shrinkage)

- 높게 설정시: 예측 성능이 떨어지지만 빠른 수행 가능
- $n_{estimators}$ 와 $learning_rate$ 는 상호 보완 관계, 조합해서 사용
 - $learning_rate$ 를 작게하고 $n_{estimators}$ 를 크게 한다면 더 이상 성능이 증가하지 않는 한계점까지 성능이 조금씩 좋아질 수 있음
 - 시간이 너무 오래 걸리며, 성능에 큰 변화는 없음



- 최적의 트리 수를 찾기 위해 조기 종료 기법 사용



확률적 그래디언트 부스팅

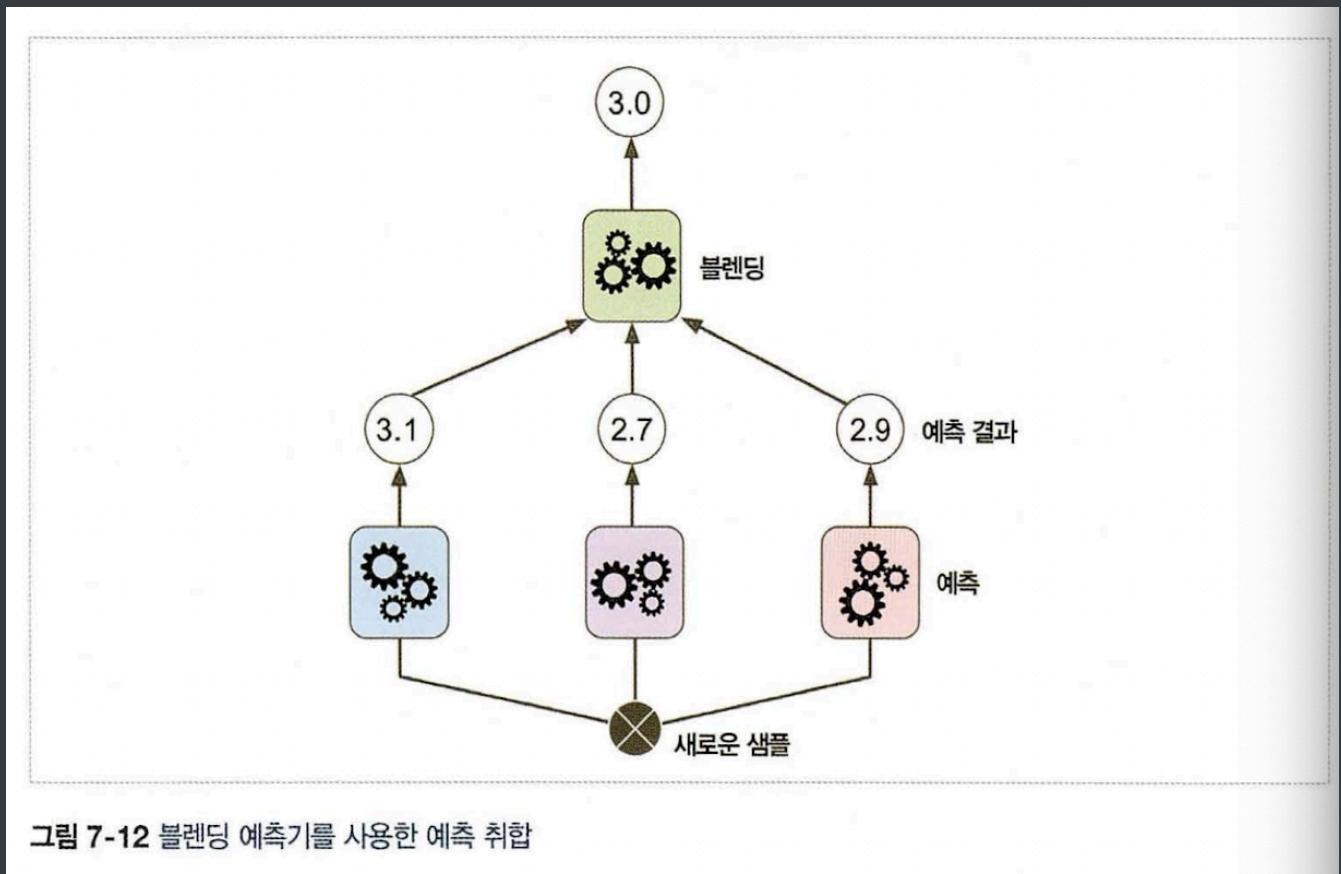
- 각 트리가 훈련할 때 사용할 훈련 샘플의 비율 설정 가능
- $subsample=\#\#$
- 설정시 편향이 높아지는 대신 분산이 낮아지게 됨, 훈련 속도 증가

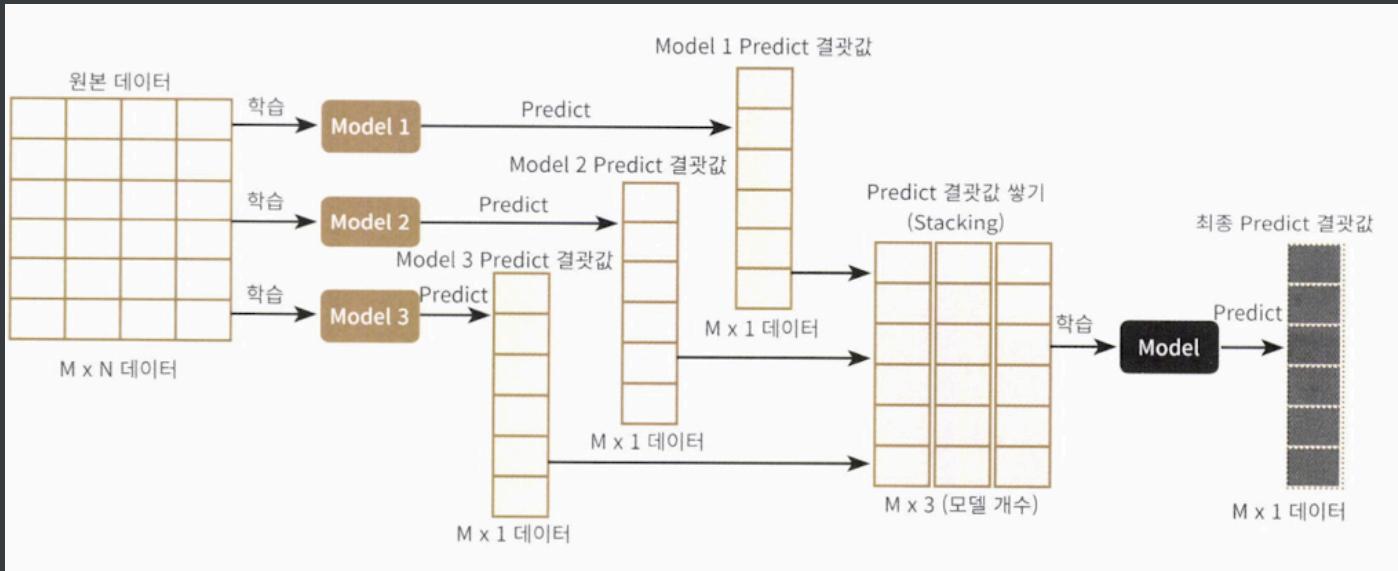
XGBoost(익스트림 그래디언트 부스팅)

- 빠른속도, 확장성, 이식성
- GBM의 단점인 느린 수행 시간 및 과적합 규제 부재 등의 문제를 해결
- 병렬 CPU 환경에서 병렬 학습이 가능해 기존 GBM보다 빠른 학습 수행 가능

7.6 스태킹(stacking)

- 양상블에 속한 예측기의 예측을 취합하는 모델을 훈련시킬 수 없을까?
- 각 예측기는 서로 다른 값을 예측하고 마지막 예측기가 예측값을 입력받아 최종 예측을 수행(마지막 예측기: 블렌더, 메타 학습기)





- 블렌더를 학습시키는 일반적인 방법: 홀드아웃(hold-out) 세트 활용

- 훈련데이터와 테스트 데이터로 나눈 뒤, 훈련 데이터에서 다시 검증 데이터 셋을 따로 떼어 냄
- 학습에 사용하지 않은 데이터에 대해 예측을 시행하므로 일반화 능력 평가에 도움
- 예시

1. 훈련 세트를 두 개의 서브셋으로 나눈다.
2. 첫번째 서브셋은 첫번째 레이어의 예측을 훈련시키기 위해 사용
3. 첫번째 레이어의 예측기를 사용해 두번째 세트에 대한 예측 수행
4. 예측한 값들을 합쳐서 최종 예측

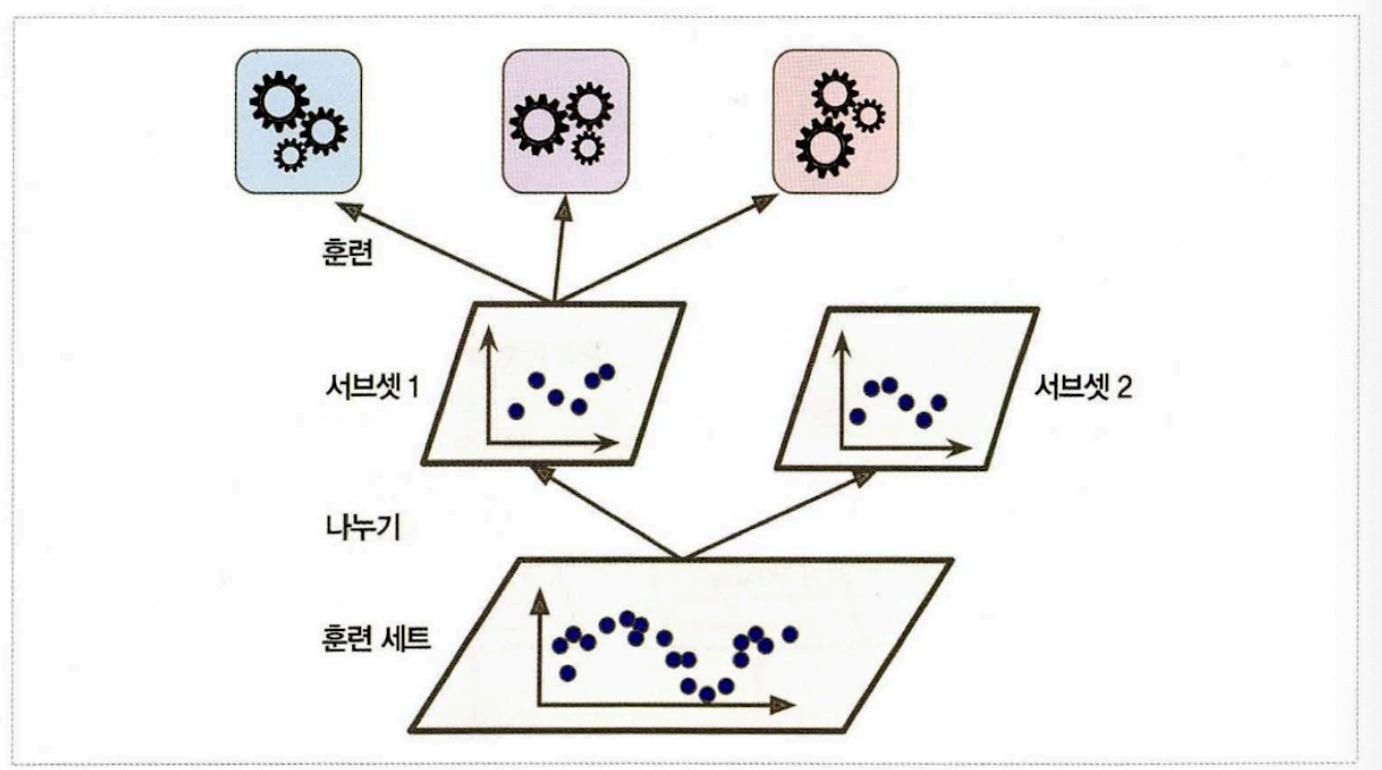


그림 7-13 첫 번째 레이어 훈련하기

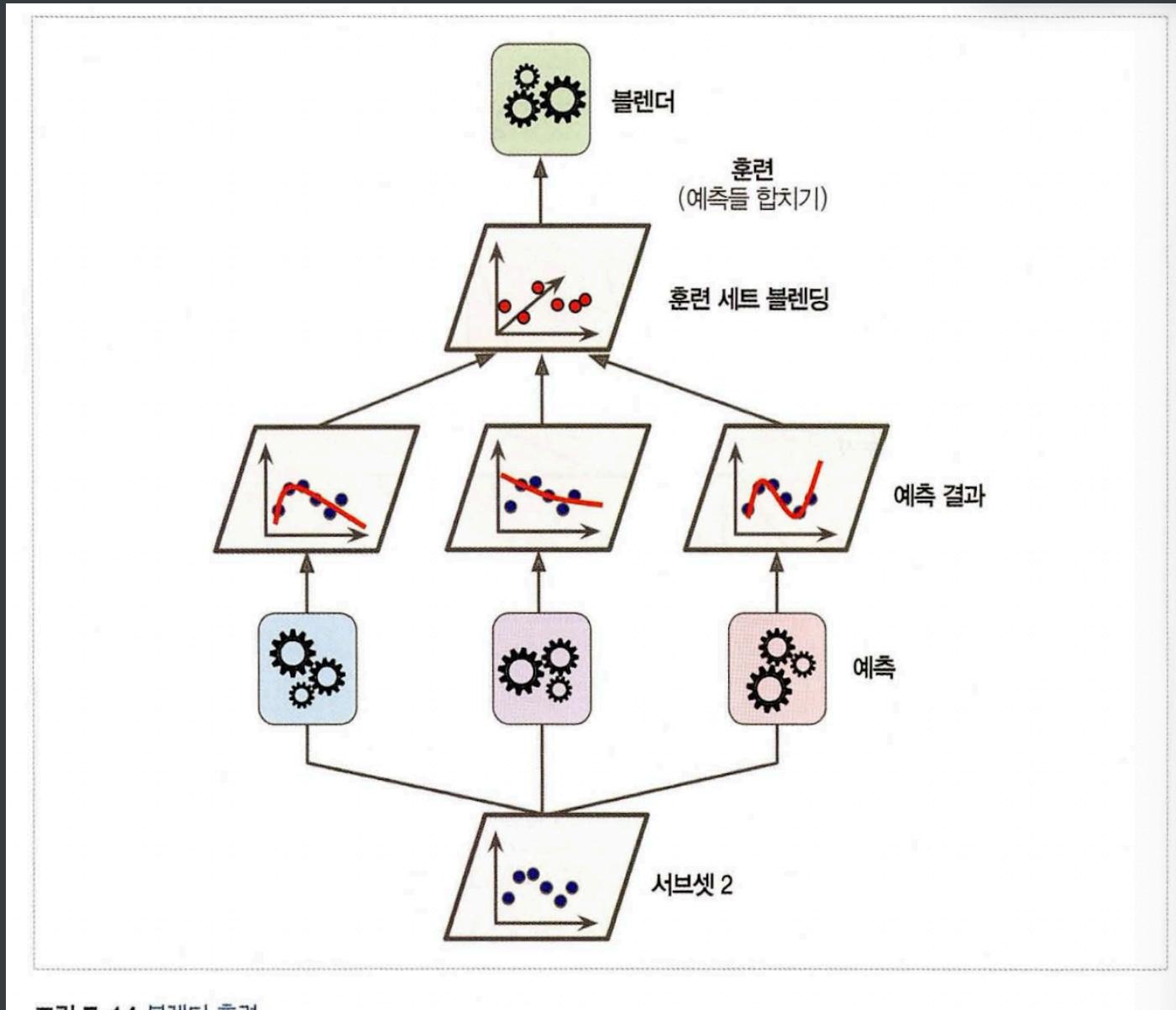


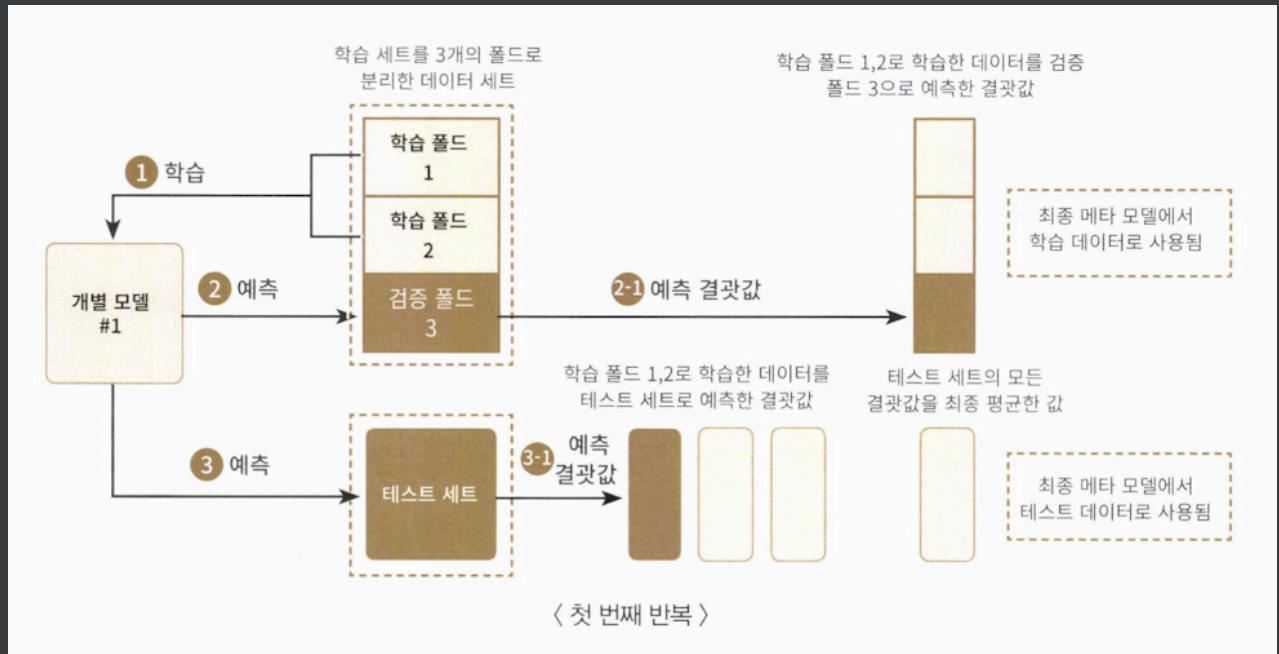
그림 7-14 브레디 증례

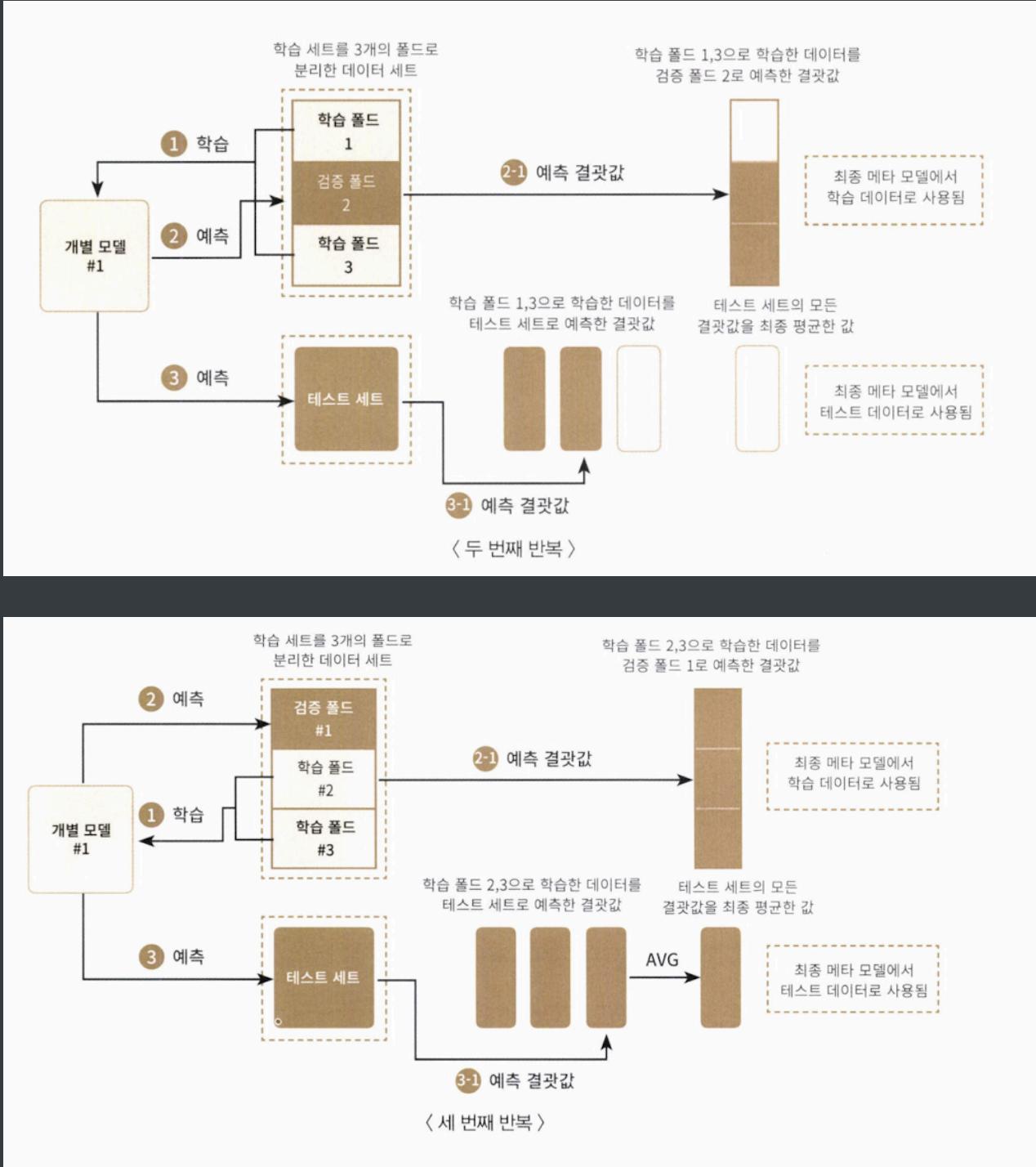
- 기본적인 스태킹 방법은 과적합 문제로 거의 사용되지 않음
- 주로 CV(교차 검증) 기반의 스태킹 사용
- 예시
 - 학습용 데이터를 3개의 폴드로 나눔
 - 3번의 유사한 반복 작업을 수행하고 마지막 3번째에서 개별 모델의 예측값으로 학습 데이터와 테스트 데이터를 생성

- 스텝 1: 각 모델별로 원본 학습/테스트 데이터를 예측한 결과 값을 기반으로 메타 모델을 위한 학습용/테스트용 데이터를 생성합니다.
- 스텝2: 스텝 1에서 개별 모델들이 생성한 학습용 데이터를 모두 스태킹 형태로 합쳐서 메타 모델이 학습할 최종 학습용 데이터 세트를 생성합니다. 마찬가지로 각 모델들이 생성한 테스트용 데이터를 모두 스태킹 형태로 합쳐서 메타 모델이 예측할 최종 테스트 세트를 생성합니다. 메타 모델은 최종적으로 생성된 학습 데이터 세트와 원본 학습 데이터의 레이블 데이터를 기반으로 학습한 뒤, 최종적으로 생성된 테스트 데이터 세트를 예측하고, 원본 테스트 데이터의 레이블 데이터를 기반으로 평가합니다

■ 스텝 1

1. 학습용 데이터를 3개의 폴드로 나눈 뒤 개별 모델을 학습시킨다
 - 2개의 폴드는 학습을 위한, 나머지 1개는 검증을 위한
2. 학습된 개별 모델은 검증 폴드 1개로 데이터를 예측한다.
3. 로직을 총 3번 반복하며 학습 데이터와 검증 데이터 세트를 변경해 가면서 학습 후 예측 결과를 저장
4. 2개의 학습 폴드 데이터로 학습된 개별 모델은 원본 테스트 데이터를 예측하여 예측값을 생성
5. 이러한 로직을 3번 반복하여 나온 3개의 예측값의 평균으로 최종 결과값을 생성하고 최종 메타 모델을 위한 테스트 데이터로 사용





■ 스텝 2

- 각 모델들이 스텝 1을 통해 생성한 학습과 테스트 데이터를 모두 합쳐 최종적으로 메타 모델이 사용할 학습 데이터와 테스트 데이터를 생성
- 메타 모델이 사용할 최종 학습 데이터와 원본 데이터의 레이블 데이터를 합쳐 메타모델을 학습
- 최종 테스트 데이터로 예측을 수행한 뒤, 최종 예측 결과를 원본 테스트 데이터의 레이블 데이터와 비교해 평가

