



3조 파이썬 머신러닝 완벽 가이드 발표

황재령, 김가은, 조승연, 민소연

목차

#01 GBM

#02 XGBoost

#03 LightGBM, HyperOpt 최적화

#04 앙상블 스택킹, 캣부스트



4.5 GBM



4.5 GBM

#1 앙상블 기법 정리

– Bagging, Boosting, Stacking

#2 GBM(Gradient Boosting Machine)_개요 및 실습

#3 GBM 하이퍼 파라미터

앙상블 기법 정리

Ensemble, Hybrid Method

앙상블 기법은 동일한 학습 알고리즘을 사용해서 여러 모델을 학습하는 개념

Weak learner를 결합한다면, Single learner보다 더 나은 성능을 얻을 수 있다

= Bagging 과 Boosting

동일한 학습 알고리즘을 사용하는 방법을 앙상블이라고 한다면,

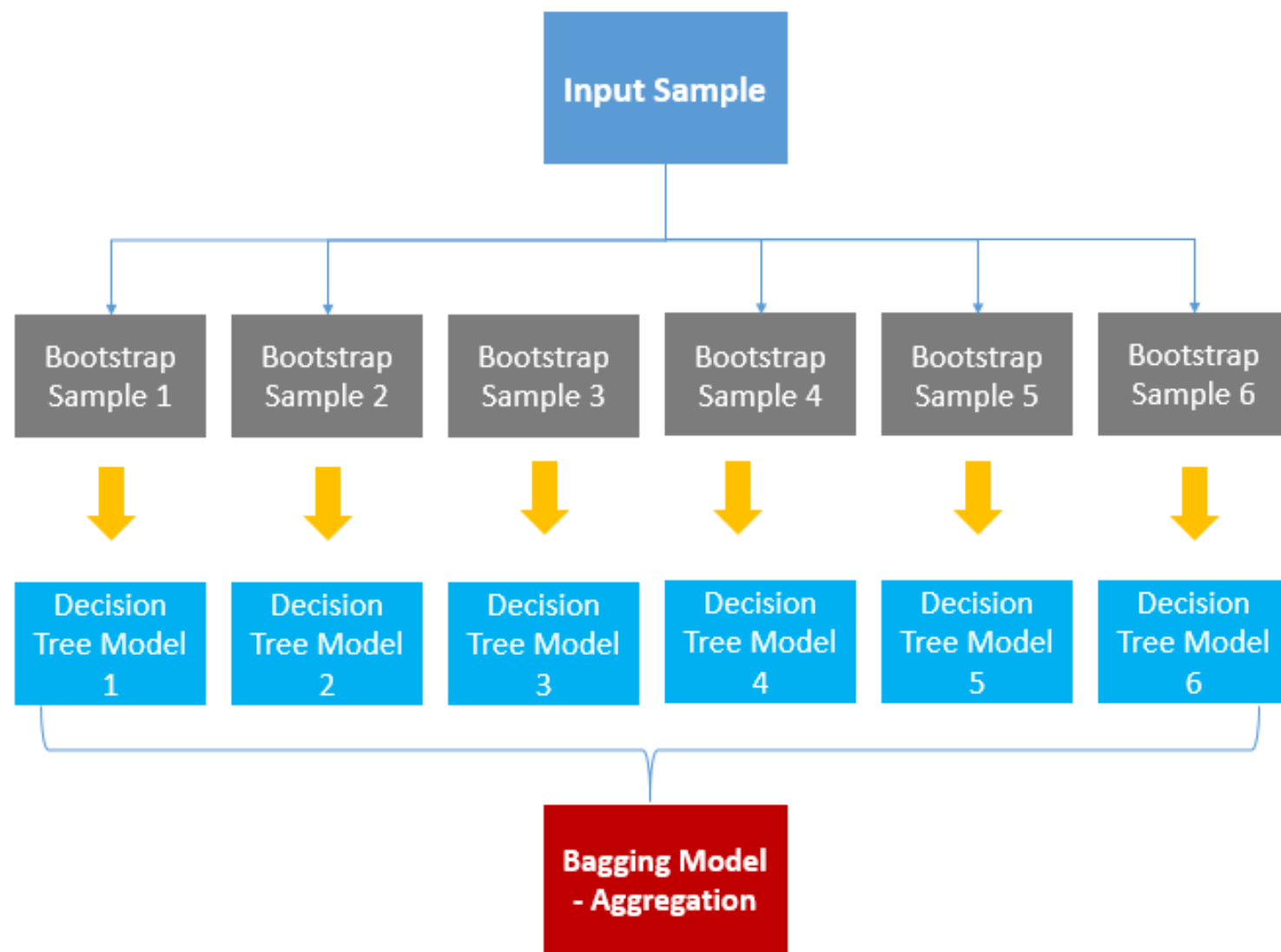
서로 다른 모델을 결합하여 새로운 모델을 만들어내는 방법도 있다.

= Stacking

앙상블 기법 정리

#1 Bagging

샘플을 여러 번 뽑아 각 모델을 학습시켜 결과를 집계(Aggregating) 하는 방법



- 1) 먼저 대상 데이터로부터 복원 랜덤 샘플링
 - 2) 이렇게 추출한 데이터가 일종의 표본 집단이 된다.
 - 3) 여기에 동일한 모델을 학습시킴
 - 4) 학습된 모델의 예측변수들을 집계하여 그 결과로 모델을 생성
- = Bootstrap Aggregating

앙상블 기법 정리

1. 높은 bias로 인한 Underfitting
2. 높은 Variance로 인한 Overfitting

Bagging은 각 샘플에서 나타난 결과를 일종의 중간값으로 맞추어 주기 때문에, Overfitting을 피할 수 있다.

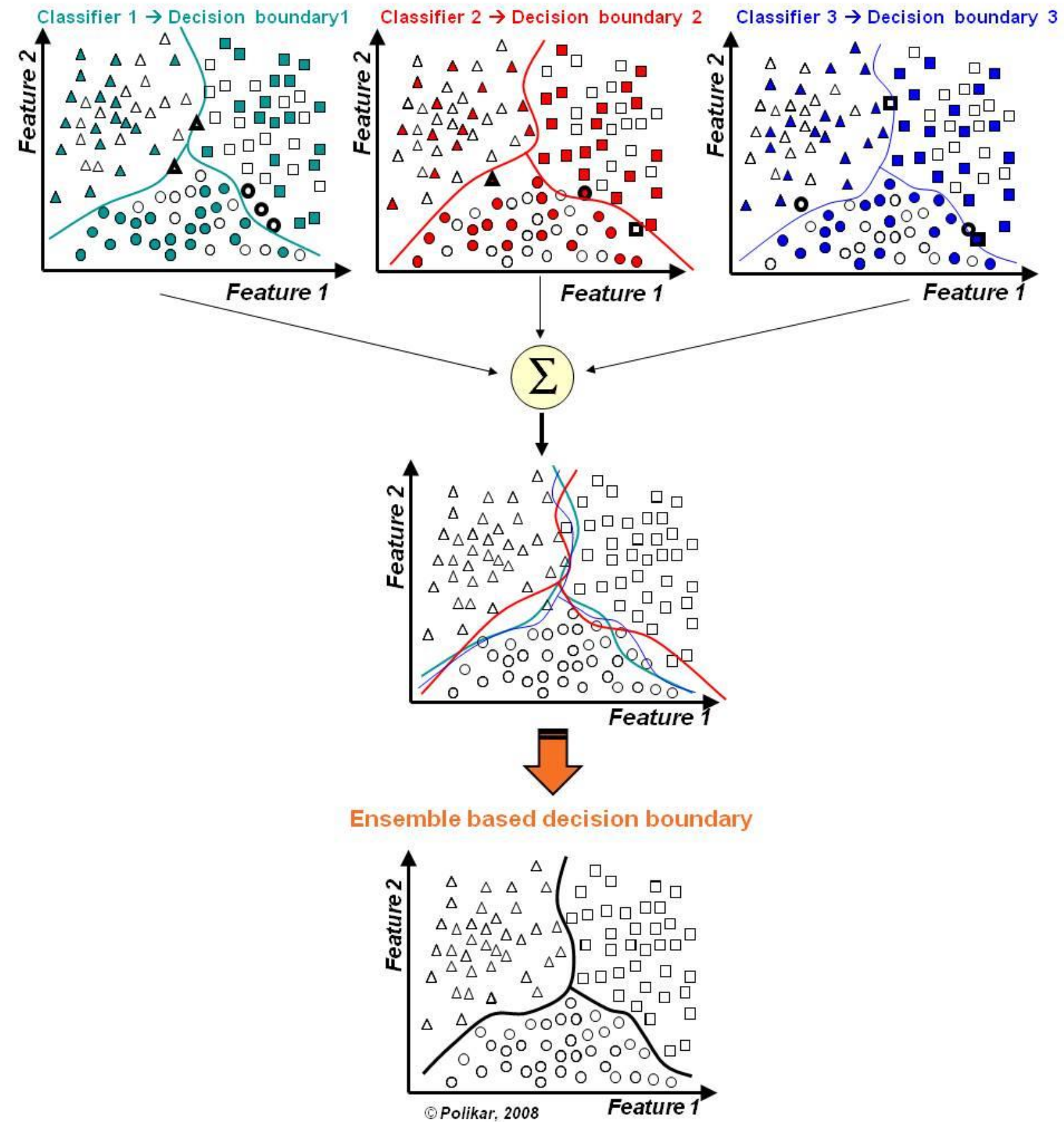
일반적으로 Categorical Data인 경우, 투표 방식 (Voting)으로 집계하며 Continuous Data인 경우,
평균 (Average)으로 집계

대표적인 Bagging 알고리즘으로 RandomForest 모델이 있다.

원래 단일 DecisionTree 모델은 boundary가 discrete 한 모양일 수 밖에 없지만,

RandomForest는 여러 트리 모델을 결합하여 이를 넘어설 수 있게 되었다.

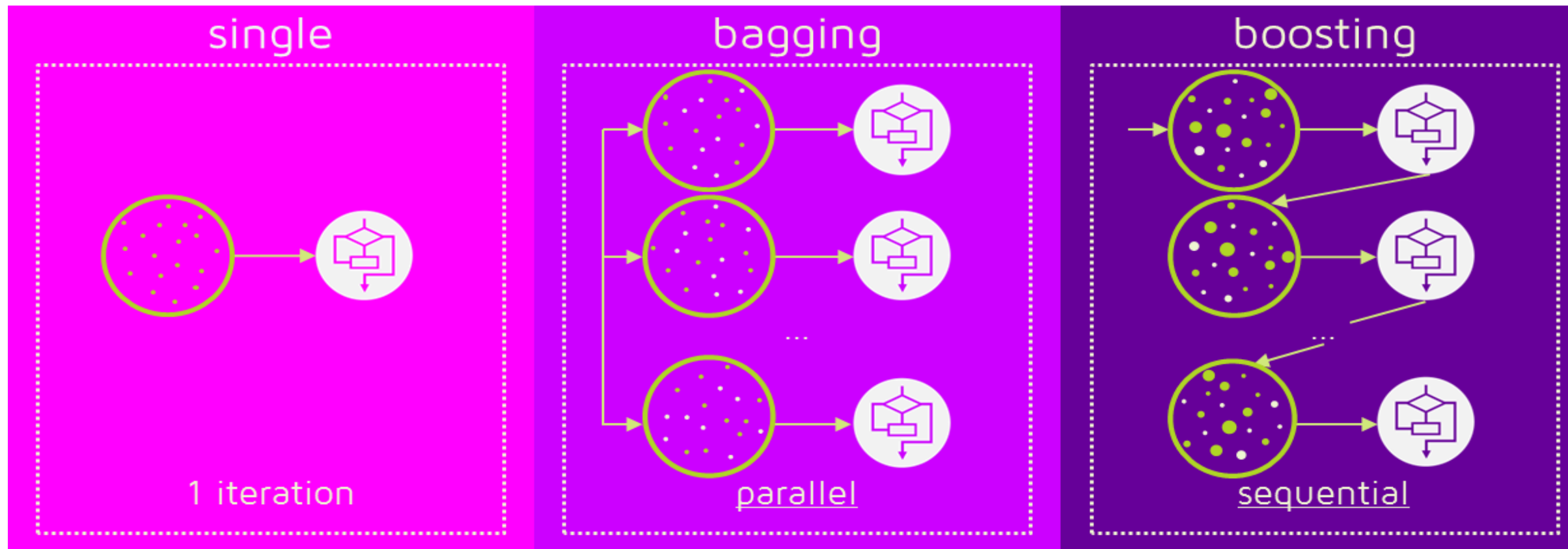
앙상블 기법 정리



앙상블 기법 정리

#2 Boosting

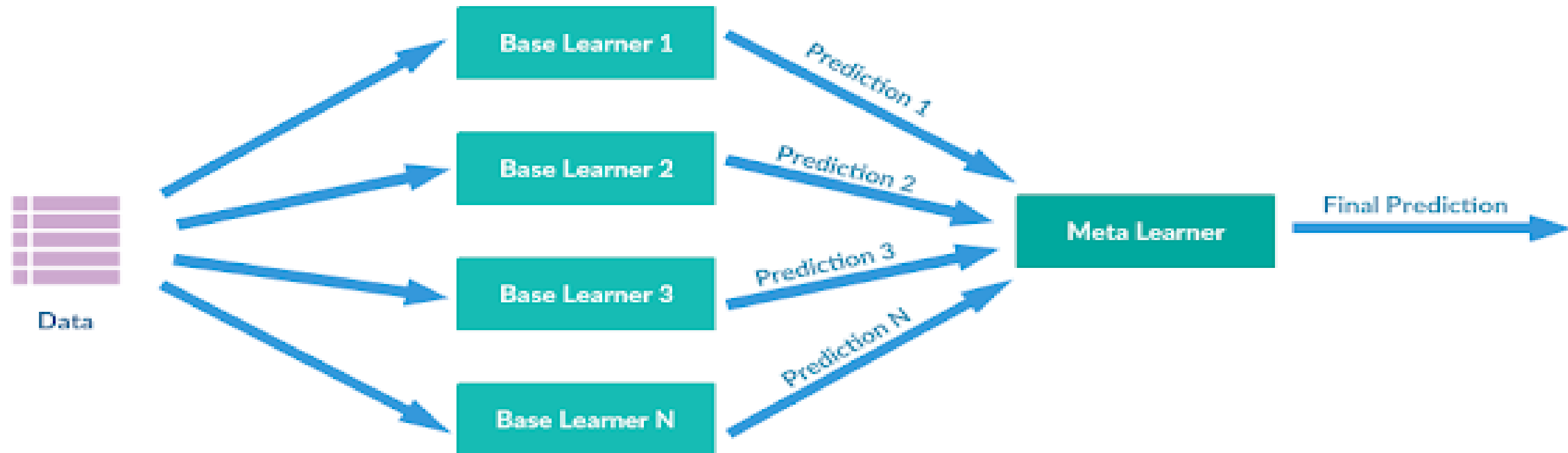
Bagging이 일반적인 모델을 만드는데 집중되어 있다면, Boosting은 맞추기 어려운 문제를 맞추는데 초점이 맞춰져 있다.



- Boosting도 Bagging과 동일하게 복원 랜덤 샘플링을 하지만, 가중치를 부여한다는 차이점
- Bagging이 병렬로 학습하는 반면, Boosting은 순차적으로 학습시킴 -> 학습이 끝나면 결과에 따라 가중치 재분배
- 오답에 대해 높은 가중치, 정답에 대해 낮은 가중치를 부여하기 때문에 오답에 더욱 집중
- Boosting 기법의 경우, 정확도 ↑, 그만큼 Outlier에 취약

앙상블 기법 정리

#3 Stacking

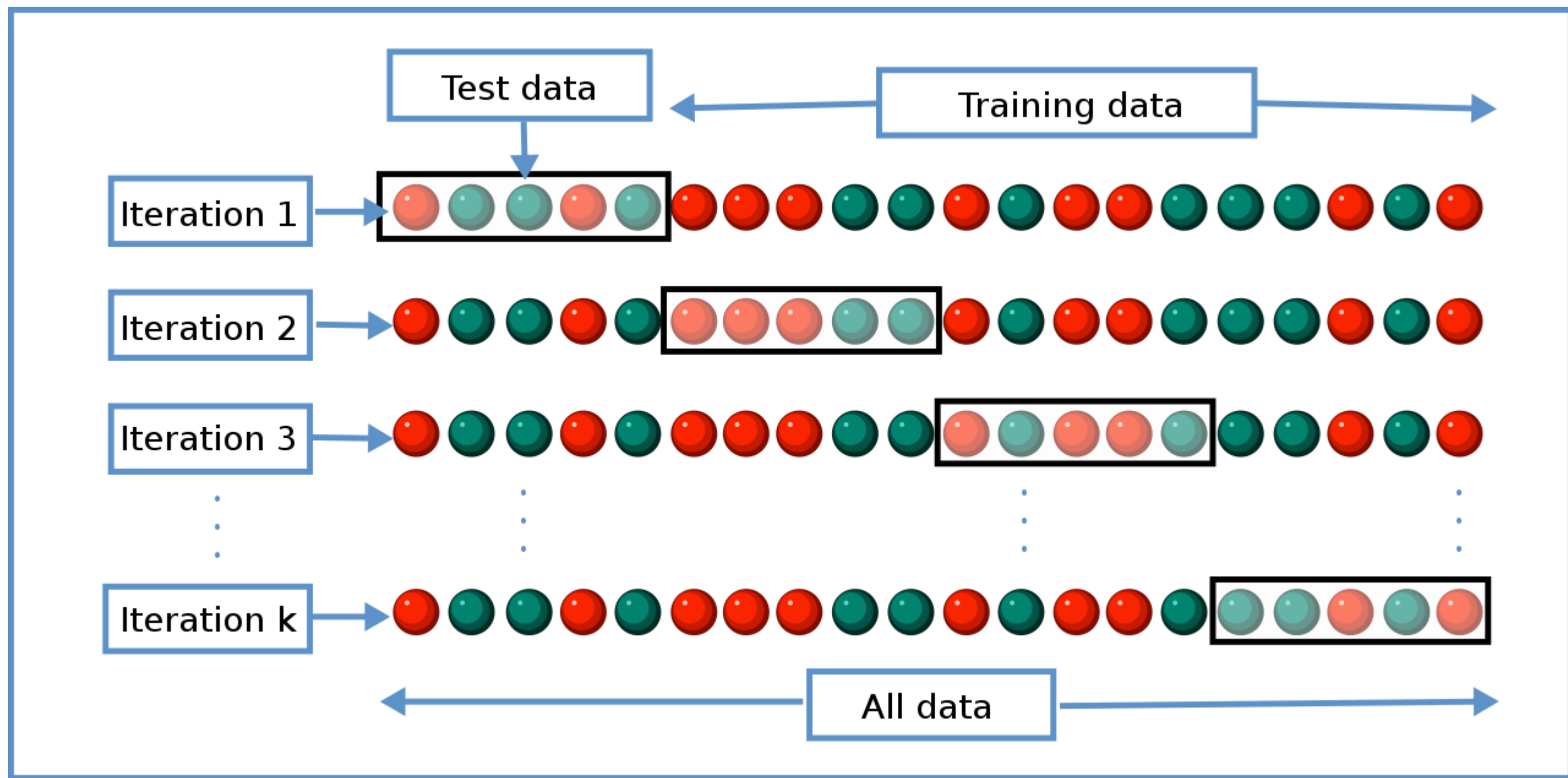


- 크로스 벨리데이션(Cross Validation) 기반으로 서로 상이한 모델들을 조합
- 개별 모델이 예측한 데이터를 다시 meta data set으로 사용해서 학습한다는 컨셉
- 2가지 개념의 모델 : 개별 모델들(그림에서의 Base Learner) & 최종 모델(그림에서의 Meta Learner)
- 기본적인 Stacking 방법의 경우 : overfitting 문제가 발생
- Bagging과 Boosting에서는 bootstrap(데이터를 random sampling) 과정을 통해 overfitting을 효과적으로 방지한 것과는 대조적인 모습

앙상블 기법 정리

#3 Stacking

따라서 Stacking에서도 비슷한 방식을 도입 : 크로스 벨리데이션(Cross Validation)으로 데이터를 쪼개는 것



CV 기반으로 Stacking을 적용함으로써 overfitting은 피하며 meta 모델은 '특정 형태의 샘플에서 어떤 종류의 단일 모델이 어떤 결과를 가지는지' 학습할 수 있게 된다.

GBM(Gradient Boosting Machine)_개요 및 실습

부스팅 알고리즘은 위에서도 언급했듯이 여러 개의 약한 학습기를 순차적으로
학습/예측을 하면서 잘못 예측한 데이터에 가중치를 부여해서
오류를 개선해나가는 학습방식

대표적인 알고리즘 : AdaBoost 와 그래디언트 부스트

-> 둘의 가장 큰 차이점은 그래디언트 부스트의 가중치 업데이트를
경사 하강법 을 이용해서 한다는 것

GBM(Gradient Boosting Machine)_개요 및 실습

경사 하강법

분류의 실제 결과값을 y , 피처를 x_1, x_2, \dots, x_n 피처에 기반한 예측 함수를 $f(x)$ 라고 할 때,
오류식 $h(x) = y - f(x)$ 을 최소화하는 방향성을 가지고
반복적으로 가중치 값을 업데이트 하는 것 (오류값은 실제값 - 예측값)

GBM(Gradient Boosting Machine)_개요 및 실습

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score
import time
import warnings
warnings.filterwarnings('ignore')

X_train, X_test, y_train, y_test = get_human_dataset()

start_time = time.time()
gb_clf = GradientBoostingClassifier(random_state=0)
gb_clf.fit(X_train, y_train)
gb_pred = gb_clf.predict(X_test)
gb_accuracy = accuracy_score(y_test, gb_pred)

print('GBM 정확도: {0:.4f}'.format(gb_accuracy))
print('GBM 수행 시간: {0:.1f} 초'.format(time.time() - start_time))
```

GBM 정확도: 0.9386
GBM 수행 시간: 1048.1 초

기본 하이퍼 파라미터만으로도 93.86%의 예측 정확도

일반적으로 GBM이 랜덤 포레스트보다는 예측 성능이 뛰어난 경우가 많지만
수행시간이 오래 걸리고, 하이퍼 파라미터 튜닝 노력도 더 필요하다.

GBM 하이퍼 파라미터

- **loss**: 경사 하강법에서 사용할 비용 함수를 지정한다. 디폴트는 *deviance* 이고 일반적으로 디폴트를 그대로 사용한다.
- **learning_rate**: GBM이 학습을 진행할 때마다 적용하는 함수이다. 0~1 사이의 값을 지정할 수 있으며, 디폴트는 0.1이다. 너무 작은 값을 설정하면 예측 성능이 높아지지만 수행 시간이 오래걸린다. 반대로 너무 높은 값을 설정하면 예측 성능이 낮아질 가능성이 높지만 수행 시간이 짧아진다.
- **n_estimators**: weak learner의 갯수이다. 개수가 많을수록 예측 성능이 일정 수준까지는 좋아지지만 수행시간이 오래걸린다. 디폴트는 100이다.
- **subsample**: weak learner가 학습에 사용하는 데이터 샘플링 비율이다. 디폴트는 1이고, 예를 들어 0.5로 설정하면 학습데이터의 50%를 기반으로 학습한다는 뜻이다. 과적합이 염려되는 경우 1보다 작은 값으로 설정한다.

GBM 하이퍼 파라미터

```
params = {'n_estimators': [100, 500],
          'learning_rate': [0.05, 0.1]}
grid_cv = GridSearchCV(gb_clf, param_grid=params, cv=2, verbose=1)
grid_cv.fit(X_train, y_train)
print(f'최적 하이퍼 파라미터:\n{grid_cv.best_params_}')
print(f'최고 예측 정확도: {np.round(grid_cv.best_score_, 4)}')
```

Fitting 2 folds for each of 4 candidates, totalling 8 fits

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  8 out of  8 | elapsed:  3.8s finished
```

최적 하이퍼 파라미터:

```
{'learning_rate': 0.1, 'n_estimators': 500}
```

최고 예측 정확도: 0.9517

*learning_rate=0.1, n_estimators=500*일 때 95.17%의
정확도를 보였다. 이를 테스트 데이터 세트에 적용해보자.

GBM 하이퍼 파라미터

```
gb_pred = grid_cv.best_estimator_.predict(X_test)
gb_accuracy = accuracy_score(y_test, gb_pred)
print(f'GBM 정확도: {np.round(gb_accuracy, 4)}')
```

GBM 정확도: 0.9737

97.37%의 높은 정확도가 나왔다.

XGBoost



#4.6 XGBoost

#1 XGBoost(eXtra Gradient Boost) 개요

#2 XGBoost 하이퍼 파라미터

#3 파이썬 래퍼 XGBoost 적용

#4 사이킷런 래퍼 XGBoost 개요 및 적용

#4.6 XGBoost 개요

XGBoost(eXtra Gradient Boost)

- 트리 기반의 앙상블 학습에서 가장 각광받고 있는 알고리즘
- 분류에 있어 일반적으로 다른 머신러닝보다 뛰어난 예측 성능
- GBM에 기반, GBM의 단점인 느린 수행 시간 및 과적합 규제 부재 등의 문제 해결
- 병렬 CPU 환경에서 병렬 학습이 가능해 기존 GBM보다 빠르게 학습 가능

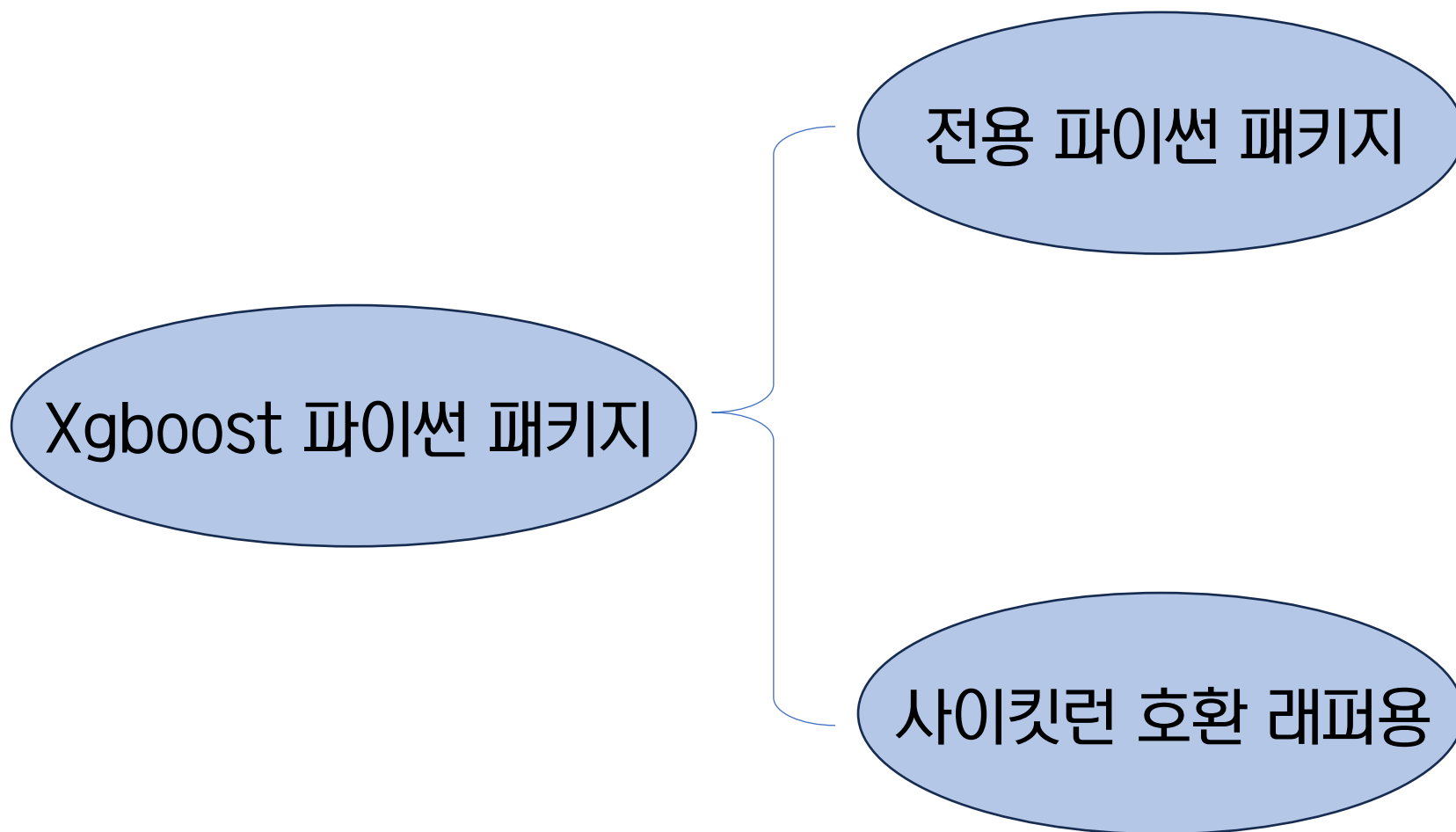
항목	설명
뛰어난 예측 성능	일반적으로 분류와 회귀 영역에서 뛰어난 예측 성능을 발휘합니다.
GBM 대비 빠른 수행 시간	일반적인 GBM은 순차적으로 Weak learner가 가중치를 증감하는 방법으로 학습하기 때문에 전반적으로 속도가 느립니다. 하지만 XGBoost는 병렬 수행 및 다양한 기능으로 GBM에 비해 빠른 수행 성능을 보장합니다. 아쉽게도 XGBoost가 일반적인 GBM에 비해 수행 시간이 빠르다는 것이지, 다른 머신러닝 알고리즘(예를 들어 랜덤 포레스트)에 비해서 빠르다는 의미는 아닙니다.
과적합 규제 (Regularization)	표준 GBM의 경우 과적합 규제 기능이 없으나 XGBoost는 자체에 과적합 규제 기능으로 과적합에 좀 더 강한 내구성을 가질 수 있습니다.
Tree pruning (나무 가지치기)	일반적으로 GBM은 분할 시 부정 손실이 발생하면 분할을 더 이상 수행하지 않지만, 이러한 방식도 자칫 지나치게 많은 분할을 발생할 수 있습니다. 다른 GBM과 마찬가지로 XGBoost도 max_depth 파라미터로 분할 깊이를 조정하기도 하지만, tree pruning으로 더 이상 긍정 이득이 없는 분할을 가지치기 해서 분할 수를 더 줄이는 추가적인 장점을 가지고 있습니다.
자체 내장된 교차 검증	XGBoost는 반복 수행 시마다 내부적으로 학습 데이터 세트와 평가 데이터 세트에 대한 교차 검증을 수행해 최적화된 반복 수행 횟수를 가질 수 있습니다. 지정된 반복 횟수가 아니라 교차 검증을 통해 평가 데이터 세트의 평가 값이 최적화 되면 반복을 중간에 멈출 수 있는 조기 중단 기능이 있습니다.
결손값 자체 처리	XGBoost는 결손값을 자체 처리할 수 있는 기능을 가지고 있습니다.

표1 XGBoost의 주요 장점, 교재 226pg 참고

#4.6 XGBoost 개요

XGBoost(eXtra Gradient Boost)

- 핵심 라이브러리 : C++
- 파이썬에서도 구동 가능 -> C++ 라이브러리 호출 : “xgboost”



초기 / 파이썬 래퍼 XGBoost 모듈

- 사이킷런과 호환X
- XGBoost 고유의 프레임워크를 파이썬 언어 기반에서 구현
- 별도의 API -> fit(), predict() 메서드 및 다양한 유틸리티 적용 불가
- 고유의 API와 하이퍼 파라미터 이용

현재 / 사이킷런 래퍼 XGBoost 모듈

- 사이킷런과 연동 가능한 래퍼 클래스(Wrapper class) 제공
- XGBClassifier / XGBRegressor
- 표준 사이킷런 개발 프로세스 및 다양한 유틸리티 활용 가능
- 다른 Estimator와 같은 사용법

#4.6 파이썬 래퍼 XGBoost 하이퍼 파라미터

파이썬 래퍼 XGBoost 하이퍼 파라미터

- XGBoost는 GBM과 유사한 하이퍼 파라미터를 동일하게 가짐
- 조기 중단(early stopping), 좌적합을 규제하기 위한 파라미터 추가

파이썬 래퍼 XGBoost 하이퍼 파라미터를 유형별로 나누면 다음과 같음

- 일반 파라미터 : 일반적으로 실행 시 스레드의 개수나 silent 모드 등의 선택을 위한 파라미터로서 디폴트 파라미터 값을 바꾸는 경우는 거의 없음
- 부스터 파라미터 : 트리 최적화, 부스팅, regularization 등과 관련 파라미터 등을 지칭
- 학습 태스크 파라미터 : 학습 수행 시의 객체 함수, 평가를 위한 지표 등을 설정하는 파라미터

#4.6 파이썬 래퍼 XGBoost 하이퍼 파라미터

파이썬 래퍼 XGBoost 하이퍼 파라미터

대부분의 하이퍼 파라미터는 Booster 파라미터에 속함

주요 일반 파라미터

- `booster`: `gbtree`(tree based model) 또는 `gblinear`(linear model) 선택. 디폴트는 `gbtree`입니다.
- `silent`: 디폴트는 0이며, 출력 메시지를 나타내고 싶지 않을 경우 1로 설정합니다.
- `nthread`: CPU의 실행 스레드 개수를 조정하며, 디폴트는 CPU의 전체 스레드를 다 사용하는 것입니다. 멀티 코어/스레드 CPU 시스템에서 전체 CPU를 사용하지 않고 일부 CPU만 사용해 ML 애플리케이션을 구동하는 경우에 변경합니다.

주요 부스터 파라미터

- `eta` [default=0.3, alias: `learning_rate`]: GBM의 학습률(learning rate)과 같은 파라미터입니다. 0에서 1 사이의 값을 지정하며 부스팅 스텝을 반복적으로 수행할 때 업데이트되는 학습률 값. 파이썬 래퍼 기반의 `xgboost`를 이용할 경우 디폴트는 0.3, 사이킷런 래퍼 클래스를 이용할 경우 `eta`는 `learning_rate` 파라미터로 대체되며, 디폴트는 0.1입니다. 보통은 0.01 ~ 0.2 사이의 값을 선호합니다.
- `num_boost_rounds`: GBM의 `n_estimators`와 같은 파라미터입니다.
- `min_child_weight` [default=1]: 트리에서 추가적으로 가지를 나눌지를 결정하기 위해 필요한 데이터들의 `weight` 총합. `min_child_weight`이 클수록 분할을 자제합니다. 과적합을 조절하기 위해 사용됩니다.
- `gamma` [default=0, alias: `min_split_loss`]: 트리의 리프 노드를 추가적으로 나눌지를 결정할 최소 손실 감소 값입니다. 해당 값보다 큰 손실(loss)이 감소된 경우에 리프 노드를 분리합니다. 값이 클수록 과적합 감소 효과가 있습니다.
- `max_depth` [default=6]: 트리 기반 알고리즘의 `max_depth`와 같습니다. 0을 지정하면 깊이에 제한이 없습니다. `Max_depth`가 높으면 특정 피쳐 조건에 특화되어 룰 조건이 만들어지므로 과적합 가능성이 높아지며 보통은 3~10 사이의 값을 적용합니다.

- `sub_sample` [default=1]: GBM의 `subsample`과 동일합니다. 트리가 커져서 과적합되는 것을 제어하기 위해 데이터를 샘플링하는 비율을 지정합니다. `sub_sample=0.5`로 지정하면 전체 데이터의 절반을 트리를 생성하는 데 사용합니다. 0에서 1 사이의 값이 가능하나 일반적으로 0.5 ~ 1 사이의 값을 사용합니다.
- `colsample_bytree` [default=1]: GBM의 `max_features`와 유사합니다. 트리 생성에 필요한 피쳐(칼럼)를 임의로 샘플링하는 데 사용됩니다. 매우 많은 피쳐가 있는 경우 과적합을 조정하는 데 적용합니다.
- `lambda` [default=1, alias: `reg_lambda`]: L2 Regularization 적용 값입니다. 피쳐 개수가 많을 경우 적용을 검토하며 값이 클수록 과적합 감소 효과가 있습니다.
- `alpha` [default=0, alias: `reg_alpha`]: L1 Regularization 적용 값입니다. 피쳐 개수가 많을 경우 적용을 검토하며 값이 클수록 과적합 감소 효과가 있습니다.
- `scale_pos_weight` [default=1]: 특정 값으로 치우친 비대칭한 클래스로 구성된 데이터 세트의 균형을 유지하기 위한 파라미터입니다.

#4.6 파이썬 래퍼 XGBoost 하이퍼 파라미터

파이썬 래퍼 XGBoost 하이퍼 파라미터

뛰어난 알고리즘일수록 파라미터를 튜닝할 필요가 적음

파라미터를 튜닝하는 경우의 수는 여러 가지 상황에 따라 달라짐

과적합 문제가 심각하다면 다음과 같이 적용할 것을 고려할 수 있음

- eta 값을 낮춘다(0.01~0.1) eta 값을 낮출 경우 num_round(또는 n_estimators)는 반대로 높여줘야 함
- max_depth 값을 낮춘다
- min_child_weight 값을 높인다
- gamma 값을 높인다
- 또한 subsample과 colsample_bytree를 조정하는 것도 트리가 너무 복잡하게 생성되는 것을 막아 과적합 문제에 도움이 될 수 있음

#4.6 파이썬 래퍼 XGBoost 하이퍼 파라미터

파이썬 래퍼 XGBoost 하이퍼 파라미터

조기 중단(Early stopping)

기존 GBM

- n_estimators(또는 num_boost_rounds)에 지정된 횟수만큼 반복적으로 학습 오류를 감소
- 학습을 진행하면서 중간에 반복을 멈출 수 없고 n_estimators에 지정된 횟수를 다 완료해야함

조기중단 기능

- XGBoost, LightGBM에 탑재
- n_estimators에 지정한 부스팅 반복 횟수에 도달하지 않더라도 예측 오류가 더 이상 개선되지 않으면 반복을 끝까지 수행하지 않고 중지해 수행 시간을 개선할 수 있음

현재 업그레이드 버전인 LightGBM 4.0.0에서는 early stopping 파라미터를 지원하지 않음

-> 3.3.2로 다운그레이드하여 수행 가능

#4.6 파이썬 래퍼 XGBoost 적용 – 위스콘신 유방암 예측

위스콘신 유방암 데이터 세트는 종양의 크기, 모양 등의 다양한 속성값을 기반으로 악성 종양(malignant)인지 양성 종양(benign)인지를 분류한 데이터 세트

위스콘신 유방암 데이터 세트에 기반해 종양의 다양한 피처에 따라 악성종양인지 양성종양인지를 XGBoost를 이용해 예측

```
In [2]: import xgboost as xgb
from xgboost import plot_importance
import pandas as pd
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
import warnings
warnings.filterwarnings('ignore')
```

Plot_importance : 피처의 중요도를 시각화

사이킷런에 내장된 데이터 세트 호출

```
dataset = load_breast_cancer()
X_features= dataset.data
y_label = dataset.target

cancer_df = pd.DataFrame(data=X_features, columns=dataset.feature_names)
cancer_df['target']= y_label
cancer_df.head(3)
```

종양의 크기와 모양에 관련된 많은 속성이 숫자형 값으로 출력

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	mean fractal dimension	...	worst texture	worst perimeter	worst area	worst smoothness	comp
0	17.99	10.38	122.8	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	0.07871	...	17.33	184.6	2019.0	0.1622	
1	20.57	17.77	132.9	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	0.05667	...	23.41	158.8	1956.0	0.1238	
2	19.69	21.25	130.0	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	0.05999	...	25.53	152.5	1709.0	0.1444	

#4.6 파이썬 래퍼 XGBoost 적용 – 위스콘신 유방암 예측

타겟 레이블 값의 종류는 악성이 0, 양성이 1

```
In [5]: print(dataset.target_names)
print(cancer_df['target'].value_counts())
```

```
['malignant' 'benign']
```

```
1    357
```

```
0    212
```

```
Name: target, dtype: int64
```

1 값인 양성이 357개, 0 값인 악성이 212개

데이터 세트의 80%는 학습용, 20%는 테스트용으로 추출 후 80%의 학습용 데이터에서 90%는 최종 학습용, 10%는 검증용으로 분할

```
In [6]: # cancer_df에서 feature용 DataFrame과 Label용 Series 객체 추출
# 맨 마지막 칼럼이 Label임. Feature용 DataFrame은 cancer_df의 첫번째 칼럼에서 맨 마지막 두번째 칼럼까지를 :-1 슬라이싱으로 추출.
X_features = cancer_df.iloc[:, :-1]
y_label = cancer_df.iloc[:, -1]

# 전체 데이터 중 80%는 학습용 데이터, 20%는 테스트용 데이터 추출
X_train, X_test, y_train, y_test = train_test_split(X_features, y_label,
                                                    test_size=0.2, random_state=156)

# 위에서 만든 X_train, y_train을 다시 쪼개서 90%는 학습과 10%는 검증용 데이터로 분리
X_tr, X_val, y_tr, y_val = train_test_split(X_train, y_train, test_size=0.1, random_state=156)
print(X_train.shape, X_test.shape)
print(X_tr.shape, X_val.shape)
```

```
(455, 30) (114, 30)
(409, 30) (46, 30)
```

전체 569개의 데이터 세트 중 최종 학습용 409개, 검증용 46개, 테스트용 114개 추출

#4.6 파이썬 래퍼 XGBoost 적용 – 위스콘신 유방암 예측

파이썬 래퍼 XGBoost와 사이킷런의 차이

전용 데이터 객체인 DMatrix 사용

- Numpy 또는 Pandas로 되어 있는 학습용, 검증, 테스트용 데이터 세트를 모두 전용의 데이터 객체인 DMatrix로 생성하여 모델에 입력
- 초기엔 주로 넘파이를 입력 받았지만 현재에는 DataFrame과 Series 기반으로도 DMatrix 생성 가능
- 주요 입력 파라미터는 data(피쳐 데이터 세트)와 label(분류 : 레이블 데이터 세트, 회귀 : 숫자형 종속값 데이터 세트)
- libsvm txt 포맷 파일, xgboost 이전 버퍼 파일도 파라미터로 입력 받아 변환 가능
- 과거 버전 XGBoost에서 판다스의 DataFrame과 호환되지 않아 DMatrix 생성 시 오류 발생할 경우 DataFrame.values를 이용해 넘파이로 일차 변환한 뒤 이를 이용해 DMatrix 변환을 적용해야 함

```
In [7]: # 만약 구버전 XGBoost에서 DataFrame으로 DMatrix 생성이 안될 경우 X_train.values로 넘파이 변환.  
# 학습, 검증, 테스트용 DMatrix를 생성.  
dtr = xgb.DMatrix(data=X_tr, label=y_tr)  
dval = xgb.DMatrix(data=X_val, label=y_val)  
dtest = xgb.DMatrix(data=X_test, label=y_test)
```

#4.6 파이썬 래퍼 XGBoost 적용 – 위스콘신 유방암 예측

XGBoost의 하이퍼 파라미터 설정 -> 주로 딕셔너리 형태로 입력

- max_depth(트리 최대 깊이)는 3
- 학습률 eta는 0.1(XGBClassifier를 사용할 경우 eta가 아니라 learning_rate)
- 예제 데이터가 0 또는 1 이진 분류이므로 목적함수(objective)는 이진 로지스틱(binary:logistic)
- 오류 함수의 평가 성능 지표는 logloss
- num_rounds(부스팅 반복 횟수)는 400회

```
In [8]: params = { 'max_depth':3,
                  'eta': 0.05,
                  'objective':'binary:logistic',
                  'eval_metric':'logloss'
                }
num_rounds = 400
```

#4.6 파이썬 래퍼 XGBoost 적용 – 위스콘신 유방암 예측

파이썬 래퍼 XGBoost는 하이퍼 파라미터를 xgboost 모듈의 train() 함수에 파라미터로 전달
조기 중단의 성능 평가는 주로 별도의 검증 데이터 세트를 이용

➔ xgboost의 train() 함수에 early_roudns 파라미터 입력하여 설정

➔ 평가용 데이터 세트 지정과 eval_metric 함께 설정해야함

학습용 DMatirx인 dtr과 검증용 DMatirx 인 dval로 설정

```
In [9]: # 학습 데이터 셋은 'train' 또는 평가 데이터 셋은 'eval' 로 명기합니다.  
eval_list = [(dtr, 'train'), (dval, 'eval')] # 또는 eval_list = [(dval, 'eval')] 만 명기해도 무방.  
  
# 하이퍼 파라미터와 early stopping 파라미터를 train( ) 함수의 파라미터로 전달  
xgb_model = xgb.train(params = params , dtrain=dtr , num_boost_round=num_rounds , #  
                      early_stopping_rounds=50, evals=eval_list )
```

train() 함수의 evals 인자값으로 입력
eval_metirc은 params 딕셔너리로 지정

[0]	train-logloss:0.65016	eval-logloss:0.66183
[1]	train-logloss:0.61131	eval-logloss:0.63609
[2]	train-logloss:0.57563	eval-logloss:0.61144
[125]	train-logloss:0.01998	eval-logloss:0.25714
[126]	train-logloss:0.01973	eval-logloss:0.25587
[127]	train-logloss:0.01946	eval-logloss:0.25640
[128]	train-logloss:0.01927	eval-logloss:0.25685
[172]	train-logloss:0.01297	eval-logloss:0.26157
[173]	train-logloss:0.01285	eval-logloss:0.26253
[174]	train-logloss:0.01278	eval-logloss:0.26229
[175]	train-logloss:0.01267	eval-logloss:0.26086

검증 데이터에 대한 logloss 값이 0.25587로 가장 낮음

early_stopping_rounds로 지정된 50회 동안 logloss값이 향상되지 않음

num_boost_round 400회로 설정했지만 175번째 반복에서 완료

#4.6 파이썬 래퍼 XGBoost 적용 – 위스콘신 유방암 예측

```
In [10]: pred_probs = xgb_model.predict(dtest)
print('predict( ) 수행 결과값을 10개만 표시, 예측 확률 값으로 표시됨')
print(np.round(pred_probs[:10],3))

# 예측 확률이 0.5 보다 크면 1, 그렇지 않으면 0 으로 예측값 결정하여 List 객체인 preds에 저장
preds = [ 1 if x > 0.5 else 0 for x in pred_probs ]
print('예측값 10개만 표시:',preds[:10])
```

예측을 위해 predict() 메서드 이용

predict() 수행 결과값을 10개만 표시, 예측 확률 값으로 표시됨
[0.845 0.008 0.68 0.081 0.975 0.999 0.998 0.998 0.996 0.001]
예측값 10개만 표시: [1, 0, 1, 0, 1, 1, 1, 1, 1, 0]

사이킷런의 predict() : 예측 결과 클래스 값(0, 1)을 반환

xgboost의 predict() : 예측 결과를 추정할 수 있는 확률 값 반환

-> 예측 확률이 0.5보다 크면 1, 그렇지 않으면 0으로 예측 값을 결정하는 로직 추가

#4.6 파이썬 래퍼 XGBoost 적용 – 위스콘신 유방암 예측

```
In [12]: from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.metrics import precision_score, recall_score
from sklearn.metrics import f1_score, roc_auc_score

def get_clf_eval(y_test, pred=None, pred_proba=None):
    confusion = confusion_matrix(y_test, pred)
    accuracy = accuracy_score(y_test, pred)
    precision = precision_score(y_test, pred)
    recall = recall_score(y_test, pred)
    f1 = f1_score(y_test, pred)
    # ROC-AUC 추가
    roc_auc = roc_auc_score(y_test, pred_proba)
    print('오차 행렬')
    print(confusion)
    # ROC-AUC print 추가
    print('정확도: {0:.4f}, 정밀도: {1:.4f}, 재현율: {2:.4f}, F1: {3:.4f}, AUC: {4:.4f}'.format(accuracy, precision, recall, f1, roc_auc))
```

3장 평가에서 생성한 get_clf_eval() 함수 불러오기

```
In [13]: get_clf_eval(y_test, preds, pred_probs)
```

오차 행렬
[[34 3]
 [2 75]]

정확도: 0.9561, 정밀도: 0.9615, 재현율: 0.9740, F1: 0.9677, AUC:0.9937

테스트 실제 레이블 값
예측 레이블
예측 확률

#4.6 파이썬 래퍼 XGBoost 적용 – 위스콘신 유방암 예측

시각화 기능 수행

`plot_importance()` : 피처의 중요도를 막대그래프 형식으로 나타냄

기본 평가 지표로 f스코어를 기반으로 해당 피처의 중요도를 나타냄

f스코어 : 해당 피처가 트리 분할 시 얼마나 자주 사용되었는지 지표로 나타낸 값

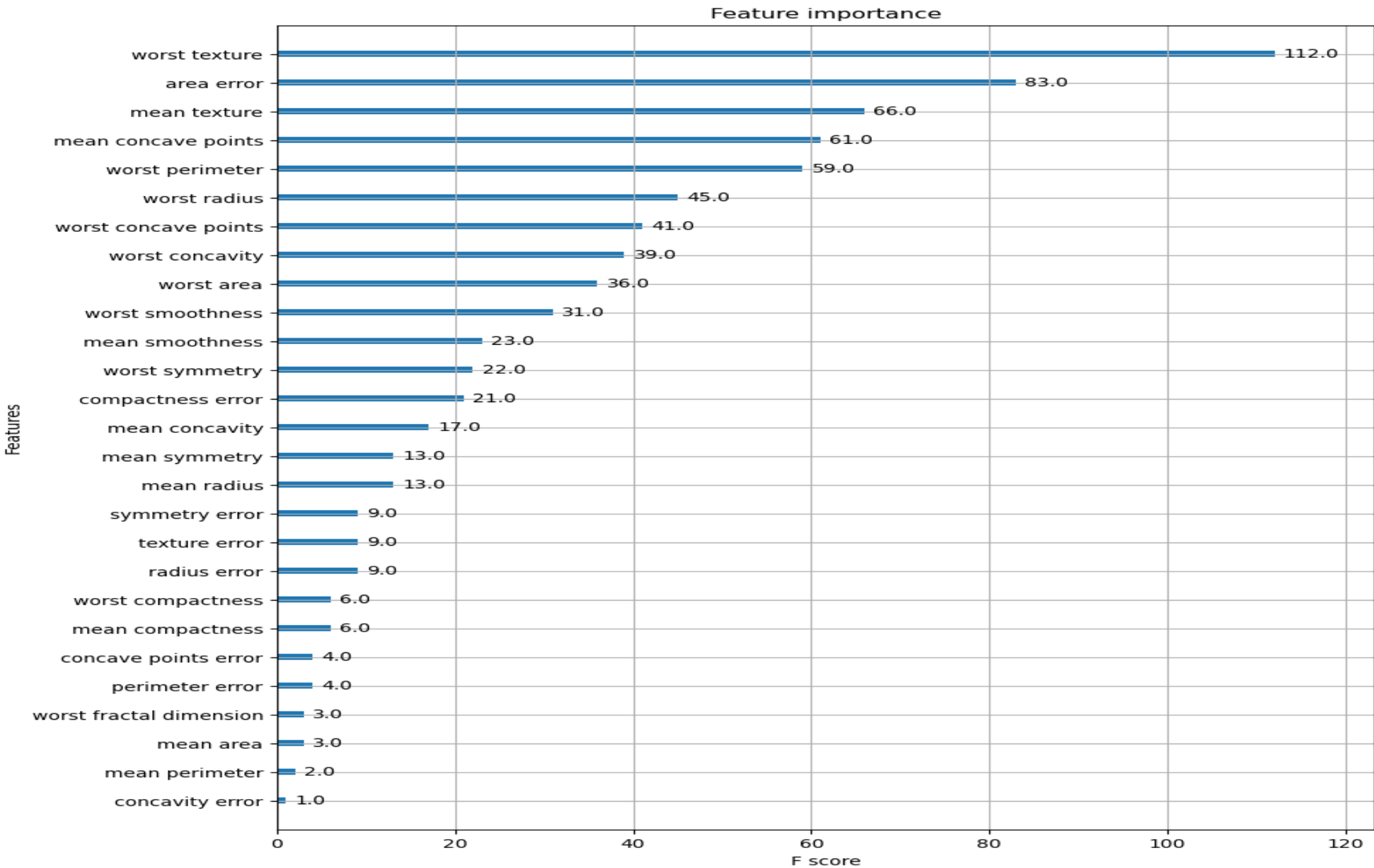
- 사이킷런 : Estimator 객체의 `feature_importances_` 속성 이용해 직접 시각화 코드 작성
- Xgboost 패키지 : `plot_importance()`를 이용해 바로 피처 중요도 시각화, 호출 시 파라미터로 앞서 학습이 완료된 모델 객체 및 맷플롯립의 `ax` 객체 입력

#4.6 파이썬 래퍼 XGBoost 적용 – 위스콘신 유방암 예측

```
In [14]: import matplotlib.pyplot as plt
%matplotlib inline

fig, ax = plt.subplots(figsize=(10, 12))
plot_importance(xgb_model, ax=ax)
```

Out [14]: <Axes: title={'center': 'Feature importance'}, xlabel='F score', ylabel='Features'>



#4.6 파이썬 래퍼 XGBoost 적용 – 위스콘신 유방암 예측

xgboost에서 트리 기반 규칙 구조는 to_graphviz() 이용해 시각화 가능

Xgboost.to_graphviz() 내에 파라미터로 학습이 완료된 모델 객체와 Graphviz가 참조할 파일명 입력

cv() API를 통해 데이터 세트에 대한 교차 검증 수행 후 최적의 파라미터를 구할 수 있음

Xgboost.cv(params, dtrain, num_boost_round=10, nfold=3, stratifield=False, folds=None, metrics(), obj=None, feval=None, maximize=False, early_stopping_rounds=None, fpreproc=None, as_pandas=True, verbose_eval=None, show_stdv=True, seed=0, callbacks=None, shuffle=True)

- params (dict) : 부스터 파라터
- dtrain (DMatrix) : 학습 데이터
- num_boost_round (int) : 부스팅 반복 횟수
- nfold (int) : CV 폴드 개수
- stratifield (bool) : CV 수행 시 층화 표본 추출(strarified sampling) 수행 여부
- metrics (string or list of strings) : CV 수행 시 모니터링 할 성능 평가 지표
- early_stoppint_rounds (int) : 조기 중단을 활성화시킴. 반복 횟수 지정

#4.6 사이킷런 래퍼 XGBoost 개요 및 적용

사이킷런 전용의 XGBoost 래퍼 클래스

- 다른 Estimator와 동일하게 fit()과 predict()만으로 학습과 예측 가능
- GridSearchCV, Pipeline 등 사이킷런의 다른 유틸리티 그대로 사용 가능
- XGBClassifier(분류), XGBRegressor(회귀)로 나뉨

호환성을 위해 기존의 xgboost 모듈에서 사용하던 네이티브 하이퍼 파라미터 몇 개를 다음과 같이 변경

- eta -> learning_rate
- sub_sample -> subsample
- lambda -> reg_lambda
- alpha -> reg_alpha

xgboost의 n_estimator와 num_boost_round 하이퍼 파라미터는 서로 동일한 파라미터

파이썬 래퍼 XGBoost API에서는 num_boost_round 적용

사이킷런 래퍼 XGBoost 클래스에서는 n_estimators 파라미터 적용

#4.6 사이킷런 래퍼 XGBoost 개요 및 적용

위스콘신 유방암 분류를 위한 래퍼 클래스 XGBClassifier 이용

```
In [16]: # 사이킷런 래퍼 XGBoost 클래스인 XGBClassifier 임포트
from xgboost import XGBClassifier
```

#Warning 메시지를 없애기 위해 eval_metric 값을 XGBClassifier 생성 인자로 입력

```
xgb_wrapper = XGBClassifier(n_estimators=400, learning_rate=0.1, max_depth=3, eval_metric='logloss')
```

```
xgb_wrapper.fit(X_train, y_train, verbose=True)
```

```
w_preds = xgb_wrapper.predict(X_test)
```

```
w_pred_proba = xgb_wrapper.predict_proba(X_test)[:, 1]
```

앞의 파이썬 래퍼 XGBoost와 동일하게 설정

학습 데이터 : 검증 데이터로 분할되기 이전의 X_train과 y_train 이용

테스트 데이터는 그대로 사용

get_clf_eval()를 이용해 사이킷런 래퍼 XGBoost로 만들어진 모델의 예측 성능 평가

```
In [17]: get_clf_eval(y_test, w_preds, w_pred_proba)
```

오차 행렬

```
[[35  2]
```

```
 [ 1 76]]
```

정확도: 0.9737, 정밀도: 0.9744, 재현율: 0.9870, F1: 0.9806, AUC:0.9951

앞 예제보다 좋은 평가 결과

∴ 최종 학습 데이터 건수가 작아지기 때문에 발생

위스콘신 데이터 세트가 작아 성능 수치가 불안정한 모습을 보임

그러나 데이터 건수가 많은 경우 일반적으로 과적합을 개선할 수 있어 모델 성능이 향상될 수 있음

#4.6 사이킷런 래퍼 XGBoost 개요 및 적용

사이킷런 래퍼XGBoost에서 조기 중단 수행

fit()에 조기 중단 관련 파라미터 입력

early_stopping_rounds : 평가 지표가 향상될 수 있는 반복 횟수 정의

eval_metric : 조기 중단을 위한 평가 지표

eval_set : 성능 평가를 수행할 데이터 세트

```
In [22]: from xgboost import XGBClassifier

xgb_wrapper = XGBClassifier(n_estimators=400, learning_rate=0.05, max_depth=3)
evals = [(X_tr,y_tr), (X_val,y_val)]
xgb_wrapper.fit(X_tr, y_tr, early_stopping_rounds=50, eval_metric="logloss",
                eval_set=evals, verbose=True)

ws50_preds = xgb_wrapper.predict(X_test)
ws50_pred_proba = xgb_wrapper.predict_proba(X_test)[: , 1]
```

맨 앞의 튜플이 학습용 데이터, 뒤의 튜플이 검증용 데이터로 자동 인식

검증을 의미하는 문자열을 넣지 않아도 됨

[0]	validation_0-logloss:0.65016	validation_1-logloss:0.66183	
[1]	validation_0-logloss:0.61131	validation_1-logloss:0.63609	
[2]	validation_0-logloss:0.57563	validation_1-logloss:0.61144	
[125]	validation_0-logloss:0.01998	validation_1-logloss:0.25714	
[126]	validation_0-logloss:0.01973	validation_1-logloss:0.25587	Validation_1-logloss가 0.25587로 가장 낮음
[127]	validation_0-logloss:0.01946	validation_1-logloss:0.25640	
[174]	validation_0-logloss:0.01278	validation_1-logloss:0.26229	
[175]	validation_0-logloss:0.01267	validation_1-logloss:0.26086	
[176]	validation_0-logloss:0.01258	validation_1-logloss:0.26103	n_estimators가 4000이지만 400번 반복하지 않고 176번째에서 학습 마무리

50번 반복까지 더 이상 성능이 향상되지 않음

#4.6 사이킷런 래퍼 XGBoost 개요 및 적용

조기 중단으로 학습된 XGBClassifier의 예측 성능

```
In [23]: get_clf_eval(y_test , ws50_preds, ws50_pred_proba)
```

오차 행렬

[[34 3]

[2 75]]

정확도: 0.9561, 정밀도: 0.9615, 재현율: 0.9740, F1: 0.9677, AUC:0.9933

- 결과는 파이썬 래퍼의 조기 중단 성능과 동일
- 위스콘신 데이터 세트가 작아, 조기 중단을 위한 검증 데이터 분리 시 검증 데이터가 없는 학습 데이터를 사용했을 때보다 성능이 약간 저조함

#4.6 사이킷런 래퍼 XGBoost 개요 및 적용

early_stopping_rounds를 10으로 설정

```
In [24]: # early_stopping_rounds를 10으로 설정하고 재 학습.
xgb_wrapper.fit(X_tr, y_tr, early_stopping_rounds=10,
                eval_metric="logloss", eval_set=evals, verbose=True)

ws10_preds = xgb_wrapper.predict(X_test)
ws10_pred_proba = xgb_wrapper.predict_proba(X_test)[:, 1]
get_clf_eval(y_test, ws10_preds, ws10_pred_proba)
```

[0]	validation_0-logloss:0.65016	validation_1-logloss:0.66183
[1]	validation_0-logloss:0.61131	validation_1-logloss:0.63609
[2]	validation_0-logloss:0.57563	validation_1-logloss:0.61144
[92]	validation_0-logloss:0.03152	validation_1-logloss:0.25918
[93]	validation_0-logloss:0.03107	validation_1-logloss:0.25864
[94]	validation_0-logloss:0.03049	validation_1-logloss:0.25951
[101]	validation_0-logloss:0.02751	validation_1-logloss:0.25955
[102]	validation_0-logloss:0.02714	validation_1-logloss:0.25901
[103]	validation_0-logloss:0.02668	validation_1-logloss:0.25991
오차 행렬		
[[34 3]		
[3 74]]		
정확도: 0.9474, 정밀도: 0.9610, 재현율: 0.9610, F1: 0.9610, AUC:0.9933		

early_stopping_rounds=50일 때의 약 0.9561보다 낮음

10번 반복하는 동안 성능 평가 지수가 향상되지 못해 더 이상 수행하지 않고 학습 종료

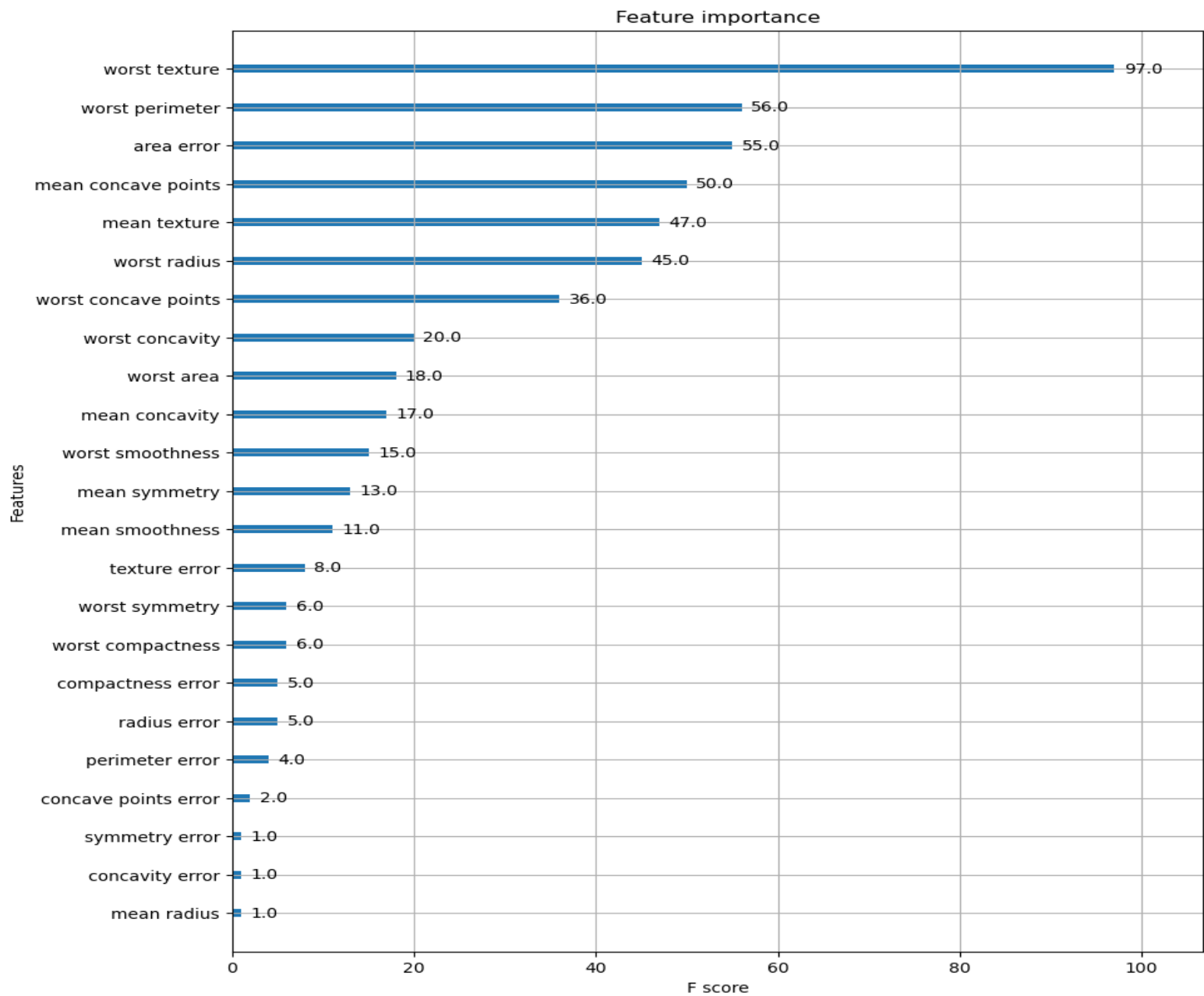
조기 중단값을 급격하게 줄일 시, 아직 성능이 향상될 여지가 있음에도 불구하고 10번 반복하는 동안 성능 평가지표가 향상되지 않으면 반복이 멈춰 버려 충분한 학습이 되지 않아 예측 성능이 나빠질 수 있음

#4.6 사이킷런 래퍼 XGBoost 개요 및 적용

피처의 중요도를 시각화하는 plot_importance()

```
In [25]: from xgboost import plot_importance
import matplotlib.pyplot as plt
%matplotlib inline

fig, ax = plt.subplots(figsize=(10, 12))
# 사이킷런 래퍼 클래스를 입력해도 무방.
plot_importance(xgb_wrapper, ax=ax)
```



앞에서 파이썬 래퍼 클래스를 입력한
결과와 똑같은 시각화 결과 도출

4.7 LightGBM



LightGBM 개요

LightGBM

: XGBoost와 더불어 가장 각광 받는 트리 기반의 부스팅 계열 알고리즘

장점+

- 타 알고리즘에 비해 빠른 학습, 예측 속도
- 적은 메모리 사용량
- XGBoost와 비슷한 예측 성능
- 카테고리형 피처의 자동 변환과 최적 분할
- 병렬 컴퓨팅 기능 제공

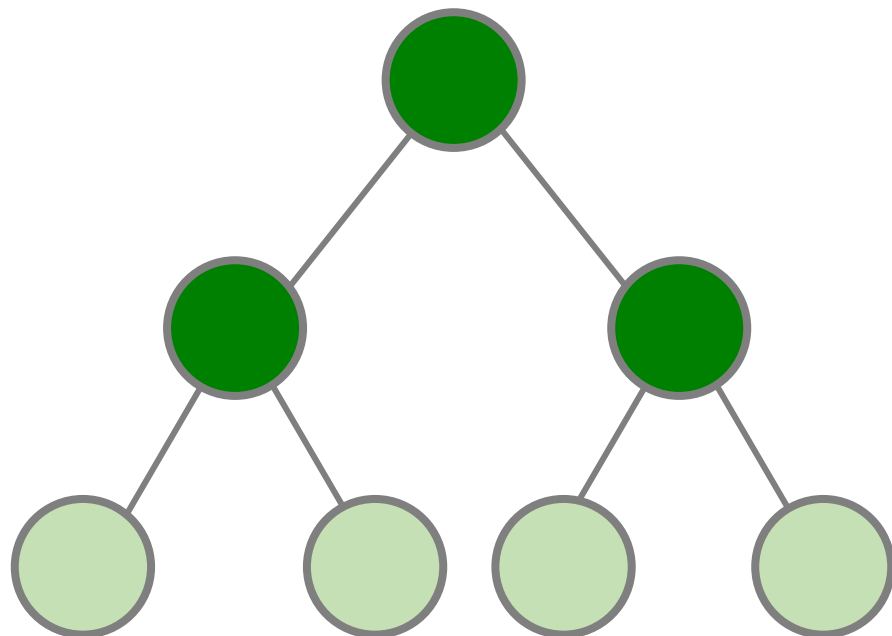
단점-

- 적은 데이터 세트에 적용할 경우 과적합이 발생하기 쉬움.(10000건 이하)

트리 분할 방식

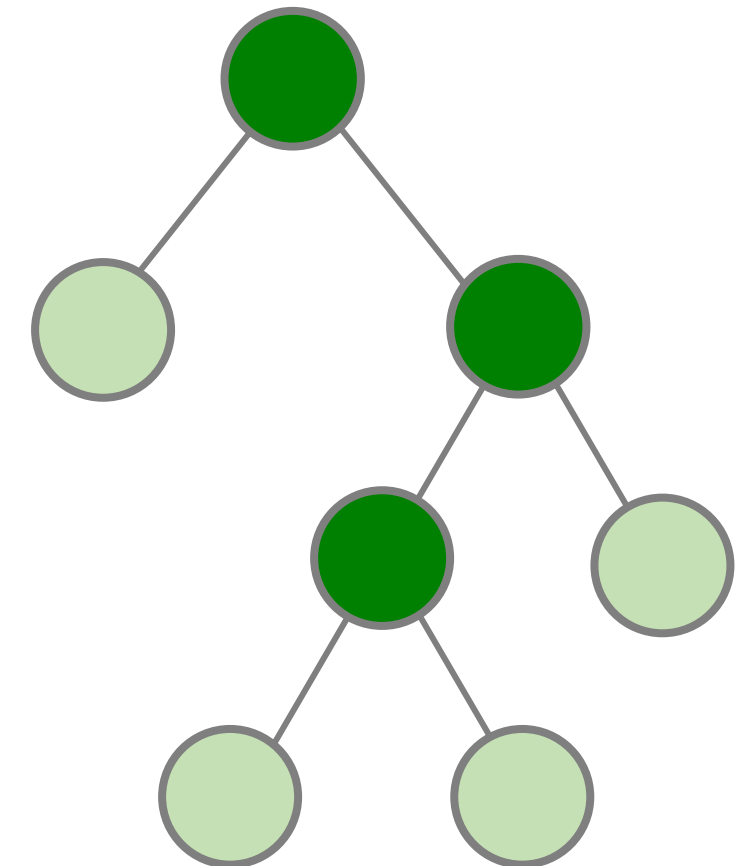
균형 분할 방식(Level Wise)

- 트리 양쪽의 균형을 맞추는 걸 중시
- 트리의 깊이를 효율적으로 줄임
(과적합에 강한 구조를 갖기 위함)
- 균형을 맞추기 위한 시간 필요
- 일반 GBM 계열(XGBoost)



리프 분할 방식(Leaf Wise)

- 최대 손실값을 가지는 리프노드를 지속적으로 분할
- 비대칭적 규칙 트리 생성
- 최대손실값 분할 -> 예측 오류 손실 최소화
- LightGBM의 특징



LightGBM 하이퍼 파라미터

XGBoost의 하이퍼 파라미터와 많은 부분이 유사

XGBoost와 달리 트리의 깊이가 깊어지므로 특성에 맞는 하이퍼 파라미터 설정이 필요

[주요 파라미터]

- **num_iterations**

: 반복 수행하려는 트리의 개수 지정

클수록 예측 성능 증가, 과하면 과적합으로 이어질 수 있음 [default = 100]

- **learning_rate**

: 부스팅 스텝을 반복적으로 수행할 때 업데이트되는 학습률

작을수록 예측 성능 증가, 너무 작으면 그에 따른 수행 시간 증가 [default = 0.1]

- **max_depth**

: 트리의 깊이, 0보다 작은 값 -> 깊이에 제한 X

앞선 level wise 방식들과는 다르게 LightGBM은 leaf wise 방식이기 때문에 깊이가 상대적으로 더 깊음

-> 과적합을 막기 위해 max_depth 조절이 필요

LightGBM 하이퍼 파라미터

- **min_data_in_leaf**
: 결정 트리의 min_samples_leaf와 같은 파라미터
최종 결정 클래스인 리프 노드가 되기 위해서 최소한으로 필요한 레코드 수 [default = 20]
- **num_leaves**
: 하나의 트리가 가질 수 있는 최대 리프 개수 [default = 31]
- **boosting**
: 부스팅의 트리를 생성하는 알고리즘을 기술 [default = gbdl]
 - gbdl : 일반적인 그래디언트 부스팅 결정 트리
 - rf : 랜덤 포레스트
- **bagging_fraction**
: 트리가 커져서 과적합되는 것을 제어하기 위해 데이터를 샘플링하는 비율 지정 [default = 1.0]

LightGBM 하이퍼 파라미터

- **feature_fraction**
: 개별 트리를 학습할 때마다 무작위로 선택하는 피터의 비율
과적합 제어를 위해 사용, GBM의 max_features와 유사 [default = 1.0]
- **lambda_l2**
: L2 regulation 제어를 위한 값 [default = 0.0]
- **lambda_l1**
: L1 regulation 제어를 위한 값 [default = 0.0]

[Learning Task 파라미터]

- **objective**
: 최솟값을 가져야 할 손실함수를 정의
회귀, 다중 클래스 분류, 이진 분류인지에 따라서 objective 손실함수 지정

LightGBM 하이퍼 파라미터

모델의 복잡도를 줄이는 것이 기본 튜닝 방안

-> 과적합 개선

- num_leaves 리프노드의 개수 ↓
- min_data_in_leaf 리프노드의 최소 데이터 개수 -> 높일 수록 트리 깊이 제어
- max_depth 트리 깊이 -> 명시적으로 트리 깊이 제한
- learning_rate를 작게 하고 n_estimator를 크게 하는 것 또한 기본적 튜닝 방안
- 해당 파라미터들을 결합해 과적합 개선!

하이퍼 파라미터 비교

유형	파이썬 래퍼 LightGBM	사이킷런 래퍼 LightGBM	사이킷런 래퍼 XGBoost
파라미터명	num_iterations	n_estimator	n_estimator
	learning_rate	learning_rate	learning_rate
	max_depth	max_depth	max_depth
	min_data_in_leaf	min_child_samples	N/A
	bagging_fraction	subsample	subsample
	feature_fraction	colsample_bytree	colsample_bytree
	lambda_l2	reg_lambda	reg_lambda
	lambda_l1	reg_alpha	reg_alpha
	early_stopping_round	early_stopping_rounds	early_stopping_rounds
	num_leaves	num_leaves	N/A
	min_sum_hessian_in_leaf	min_child_weight	min_child_weight

=> 사이킷런 래퍼들끼리 겹치는 게 많다!

LightGBM 위스콘신 유방암 예측

```
#LightGBM의 파이썬 패키지인 lightgbm에서 LGBMClassifier 임포트
from lightgbm import LGBMClassifier

import pandas as pd
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split

dataset = load_breast_cancer()

cancer_df = pd.DataFrame(data=dataset.data, columns=dataset.feature_names)
cancer_df['target'] = dataset.target
X_features = cancer_df.iloc[:, :-1]
y_label = cancer_df.iloc[:, -1]

#전체 데이터 중 80%는 학습용 데이터, 20%는 테스트용 데이터 추출
X_train, X_test, y_train, y_test=train_test_split(X_features, y_label, test_size=0.2, random_state=156 )

#위에서 만든 X_train, y_train을 다시 쪼개서 90%는 학습과 10%는 검증용 데이터로 분리
X_tr, X_val, y_tr, y_val= train_test_split(X_train, y_train, test_size=0.1, random_state=156 )

#앞서 XGBoost와 동일하게 n_estimators는 400 설정.
lgbm_wrapper = LGBMClassifier(n_estimators=400, learning_rate=0.05)

#LightGBM도 XGBoost와 동일하게 조기 중단 수행 가능.
evals = [(X_tr, y_tr), (X_val, y_val)]
lgbm_wrapper.fit(X_tr, y_tr, early_stopping_rounds=50, eval_metric="logloss", eval_set=evals, verbose=True)
preds = lgbm_wrapper.predict(X_test)
pred_proba = lgbm_wrapper.predict_proba(X_test)[:, 1]
```

LightGBM 위스콘신 유방암 예측

[1]	training's binary_logloss: 0.625671	valid_1's binary_logloss: 0.628248
[2]	training's binary_logloss: 0.588173	valid_1's binary_logloss: 0.601106
[3]	training's binary_logloss: 0.554518	valid_1's binary_logloss: 0.577587
[4]	training's binary_logloss: 0.523972	valid_1's binary_logloss: 0.556324
[5]	training's binary_logloss: 0.49615	valid_1's binary_logloss: 0.537407
[6]	training's binary_logloss: 0.470108	valid_1's binary_logloss: 0.519401
[58]	training's binary_logloss: 0.0591944	valid_1's binary_logloss: 0.262011
[59]	training's binary_logloss: 0.057033	valid_1's binary_logloss: 0.261454
[60]	training's binary_logloss: 0.0550801	valid_1's binary_logloss: 0.260746
[61]	training's binary_logloss: 0.0532381	valid_1's binary_logloss: 0.260236
[62]	training's binary_logloss: 0.0514074	valid_1's binary_logloss: 0.261586
[63]	training's binary_logloss: 0.0494837	valid_1's binary_logloss: 0.261797
[109]	training's binary_logloss: 0.00915965	valid_1's binary_logloss: 0.280752
[110]	training's binary_logloss: 0.00882581	valid_1's binary_logloss: 0.282152
[111]	training's binary_logloss: 0.00850714	valid_1's binary_logloss: 0.280894

LightGBM 위스콘신 유방암 예측

```
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.metrics import precision_score, recall_score
from sklearn.metrics import f1_score, roc_auc_score

def get_clf_eval(y_test, pred=None, pred_proba=None):
    confusion = confusion_matrix(y_test, pred)
    accuracy = accuracy_score(y_test, pred)
    precision = precision_score(y_test, pred)
    recall = recall_score(y_test, pred)
    f1 = f1_score(y_test, pred)
    #ROC-AUC 추가
    roc_auc = roc_auc_score(y_test, pred_proba)
    print('오차 행렬')
    print(confusion)
    #ROC-AUC print 추가
    print('정확도: {0:.4f}, 정밀도: {1:.4f}, 재현율: {2:.4f}, F1: {3:.4f}, AUC: {4:.4f}'.format(accuracy, precision, recall, f1, roc_auc))
```

```
get_clf_eval(y_test, preds, pred_proba)
```

```
오차 행렬
[[34  3]
 [ 2 75]]
```

```
정확도: 0.9561, 정밀도: 0.9615, 재현율: 0.9740, F1: 0.9677, AUC: 0.9877
```

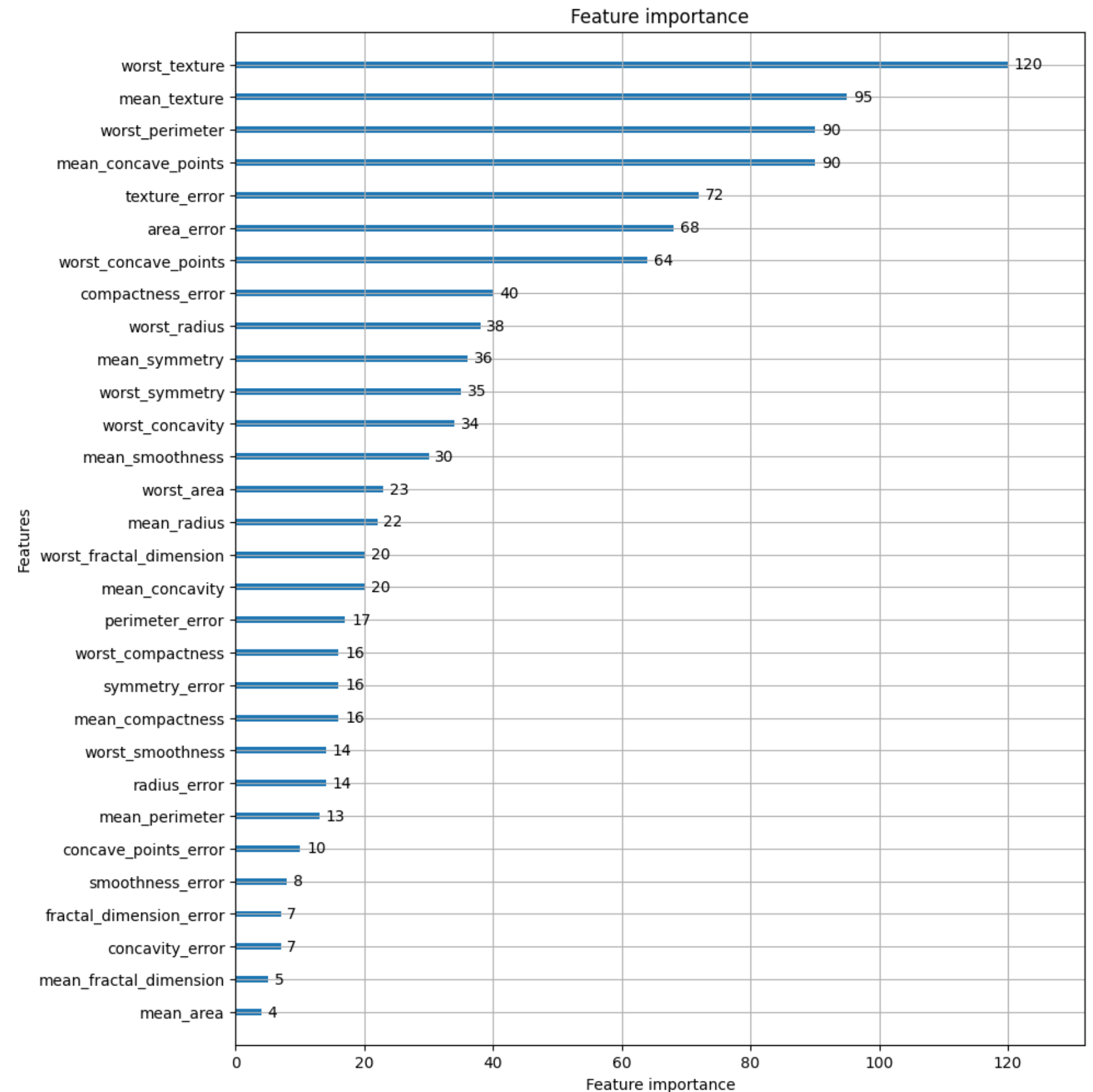
→ 앞선 XGBoost만큼의 성능

LightGBM 위스콘신 유방암 예측

```
#plot_importance( )를 이용하여 feature 중요도 시각화
from lightgbm import plot_importance
import matplotlib.pyplot as plt
%matplotlib inline

fig, ax = plt.subplots(figsize=(10, 12))
plot_importance(lgbm_wrapper, ax=ax)
plt.savefig('lightgbm_feature_importance.tif', format='tif', dpi=300, bbox_inches='tight')
```

→ 각 변수별 중요도 확인 가능



4.8 HyperOpt 최적화



하이퍼 파라미터 튜닝

Grid Search 방식

- : 가능한 모든 하이퍼 파라미터 조합을 학습/평가하여 최적화
- > 경우의 수가 너무 많은 경우 시간이 오래 걸리는 문제

```
[ ] params = {  
    'max_depth' = [10, 20, 30, 40, 50], 'num_leaves' = [35, 45, 55, 65],  
    'colsample_bytree' = [0.5, 0.6, 0.7, 0.8, 0.9], 'subsample' = [0.5, 0.6, 0.7, 0.8, 0.9],  
    'min_child_weight' = [10, 20, 30, 40], 'reg_alpha' = [0.01, 0.05, 0.1]  
}
```

-> $5*4*5*5*4*3 = 6000$

최적화를 위해 학습/평가를 6000번 해야하는 상황

시간 때문에 하이퍼 파라미터 개수를 줄이거나 범위를 줄이게 됨

->> 베이지안 최적화로 해결

베이즈 정리

사전 확률

: (사건 B가 발생하기 전에 가지고 있던) A의 확률 = $P(A)$

조건부 확률

: 원인 A에 의한 사건 B의 발생 확률(A: 원인, B: 관측)

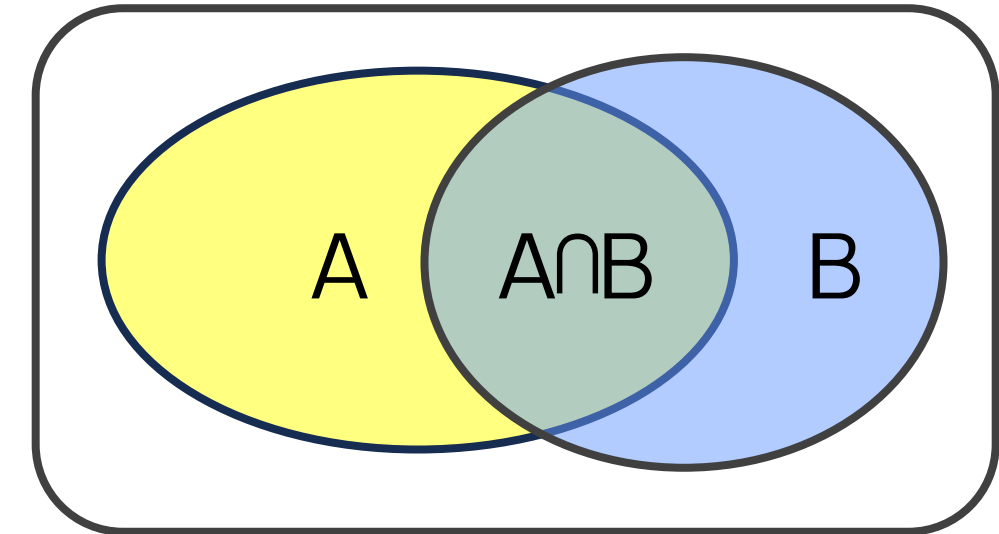
$$P(B|A) = \frac{P(A \cap B)}{P(A)}$$

사후 확률

: 관측 B가 발생한 것을 알 때 그것의 원인이 A일 확률

$$P(A|B) = \frac{P(B \cap A)}{P(B)} = \frac{P(A)P(A \cap B)}{P(A)P(B|A) + P(A')P(B|A')}$$

=> 베이즈 정리 통해 $P(B|A)$ 를 알 때 $P(A|B)$ 를 구할 수 있다



베이즈 정리

베이즈 정리는 새로운 증거나 추가 증거가 주어지면 기존 예측이나 확률을 수정하는 방법을 제공

$$P(H|E) = \frac{P(E|H)P(H)}{P(E)}$$

$P(H)$: 가설 H가 성립할 확률

$P(H|E)$: 사건 E가 일어났을 때 가설 H가 성립할 확률

⇔ 새로운 증거 혹은 사건이 일어났을 때 해당 가설이 성립할 확률

베이지안 최적화 개요

베이지안 최적화

: 목적 함수 식을 제대로 알 수 없는 블랙 박스 형태의 함수에서 최대/최소
함수 반환값을 만드는 최적 입력값을 가능한 적은 시도로 찾는 방법

↓
모델 성능 평가 지표

↘
하이퍼 파라미터

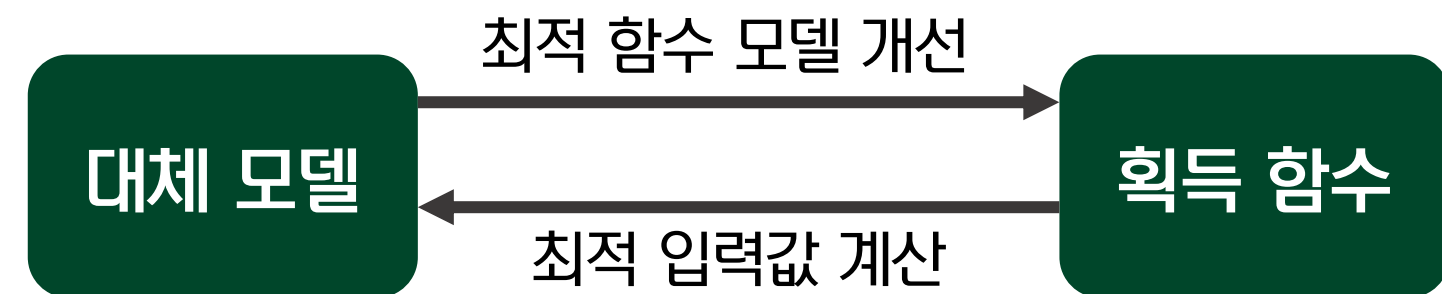
*블랙 박스 : 입력값과 출력값은 알지만 내부 작업은 명확히 알 수 없는 것

대체 모델

: 획득 함수로부터 최적 함수를 예측할 수 있는 입력값을 추천받음
→ 이를 기반으로 최적 함수 모델 개선

획득 함수

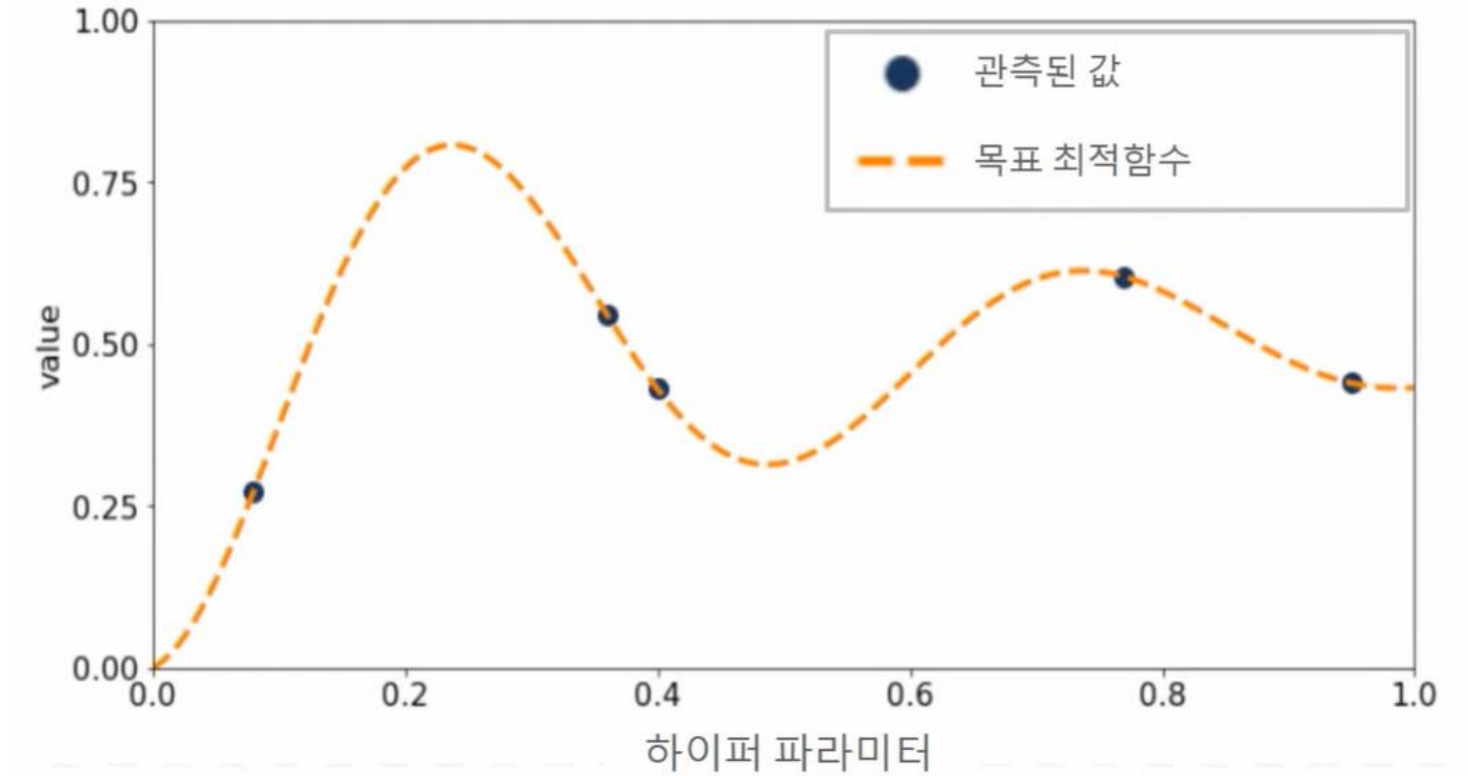
: 개선된 대체 모델 기반으로 최적 입력값 계산



베이지안 최적화 개요

#Step1

랜덤하게 하이퍼 파라미터 샘플링, 성능 결과 관측

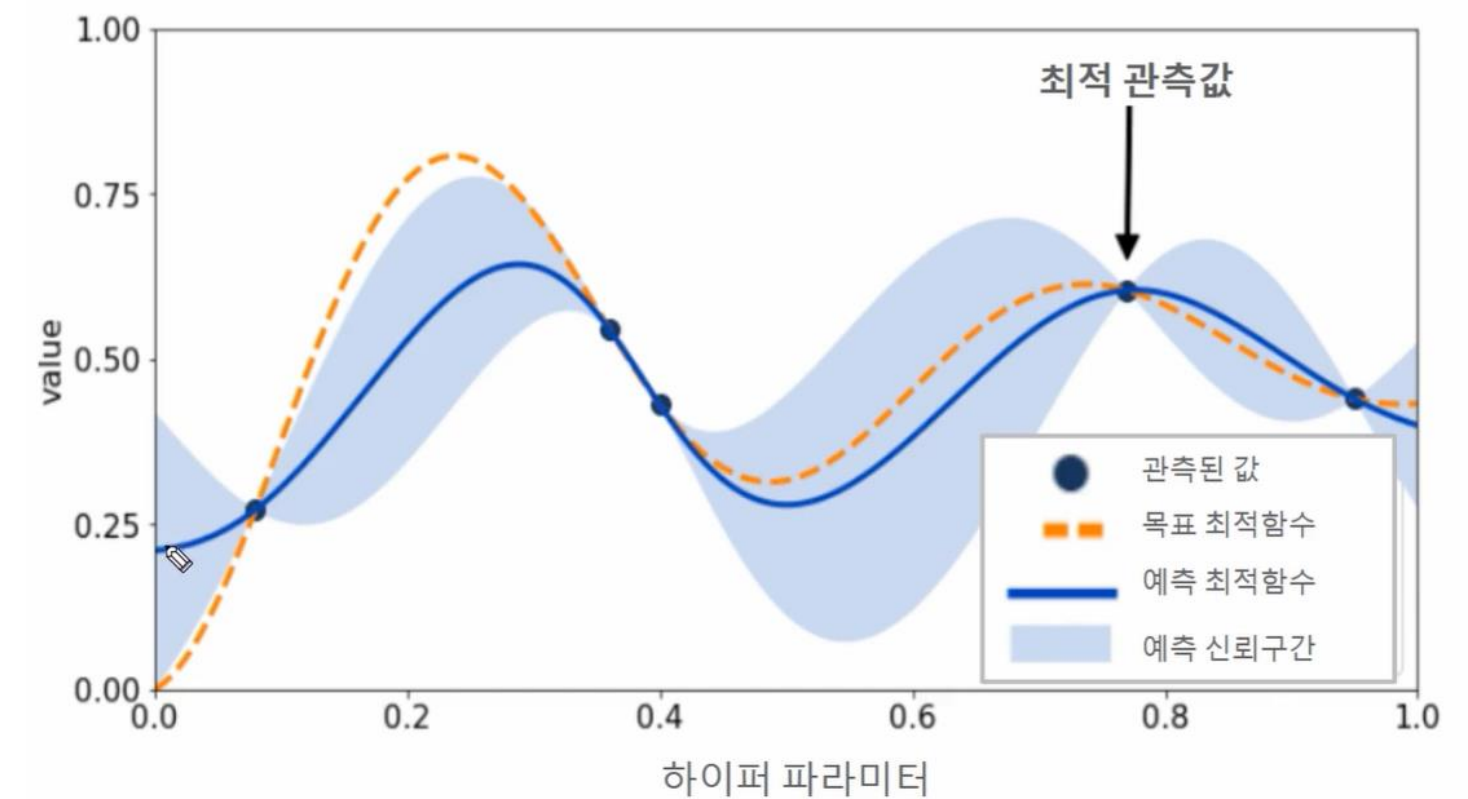


#Step2

관측된 값 기반으로 대체모델이 최적 함수 추정

최적 관측값 : 관측된 값들 중 y축 value에서 가장 높은 값을 가질 때의 하이퍼 파라미터

예측 신뢰 구간 : 추정된 함수의 결과값 오류 편차, 불확실성

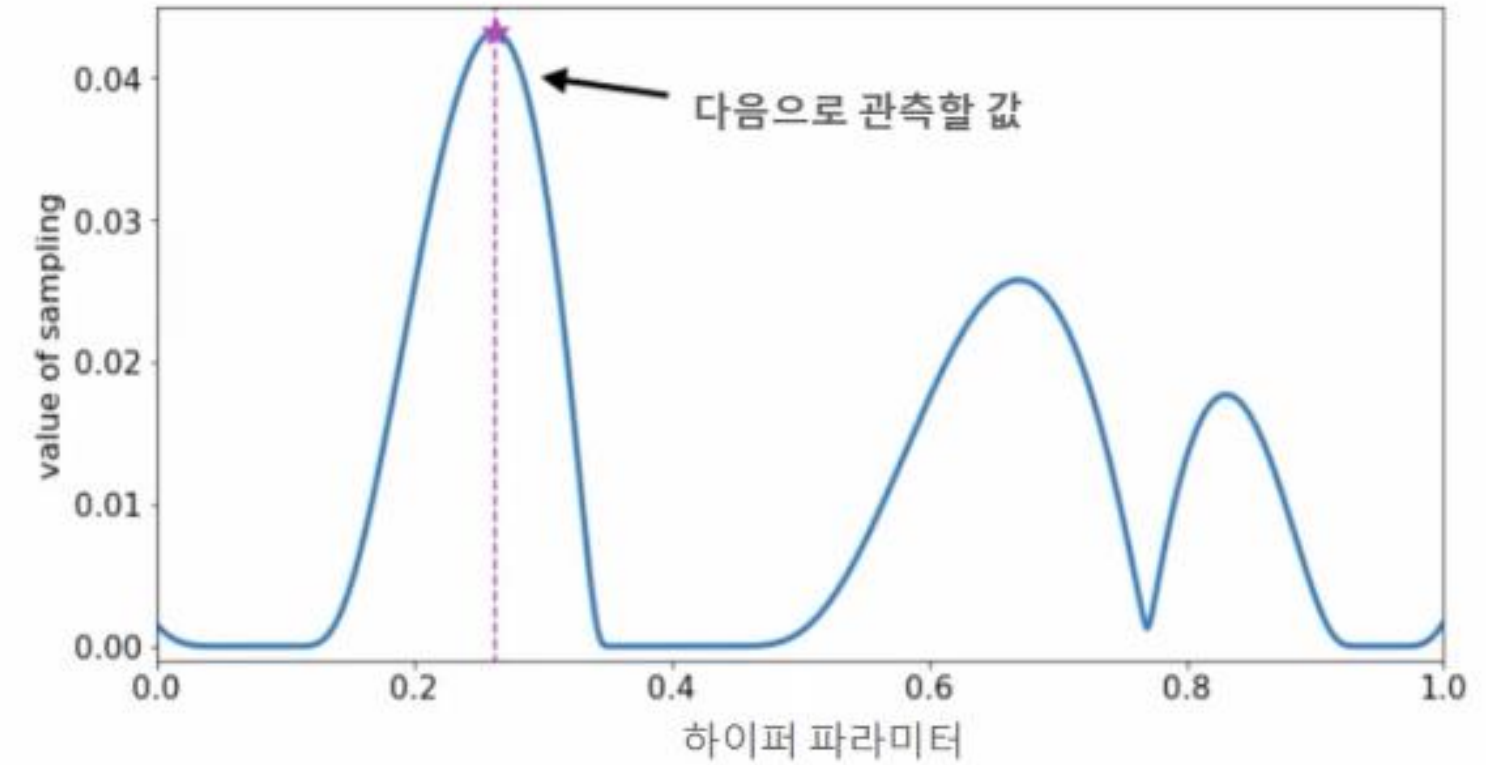


베이지안 최적화 개요

#Step3

획득 함수가 이전의 최적 관측값보다 더 큰 최댓값을 가질 가능성이 높은 지점을 찾음

->다음으로 관측할 하이퍼 파라미터 값

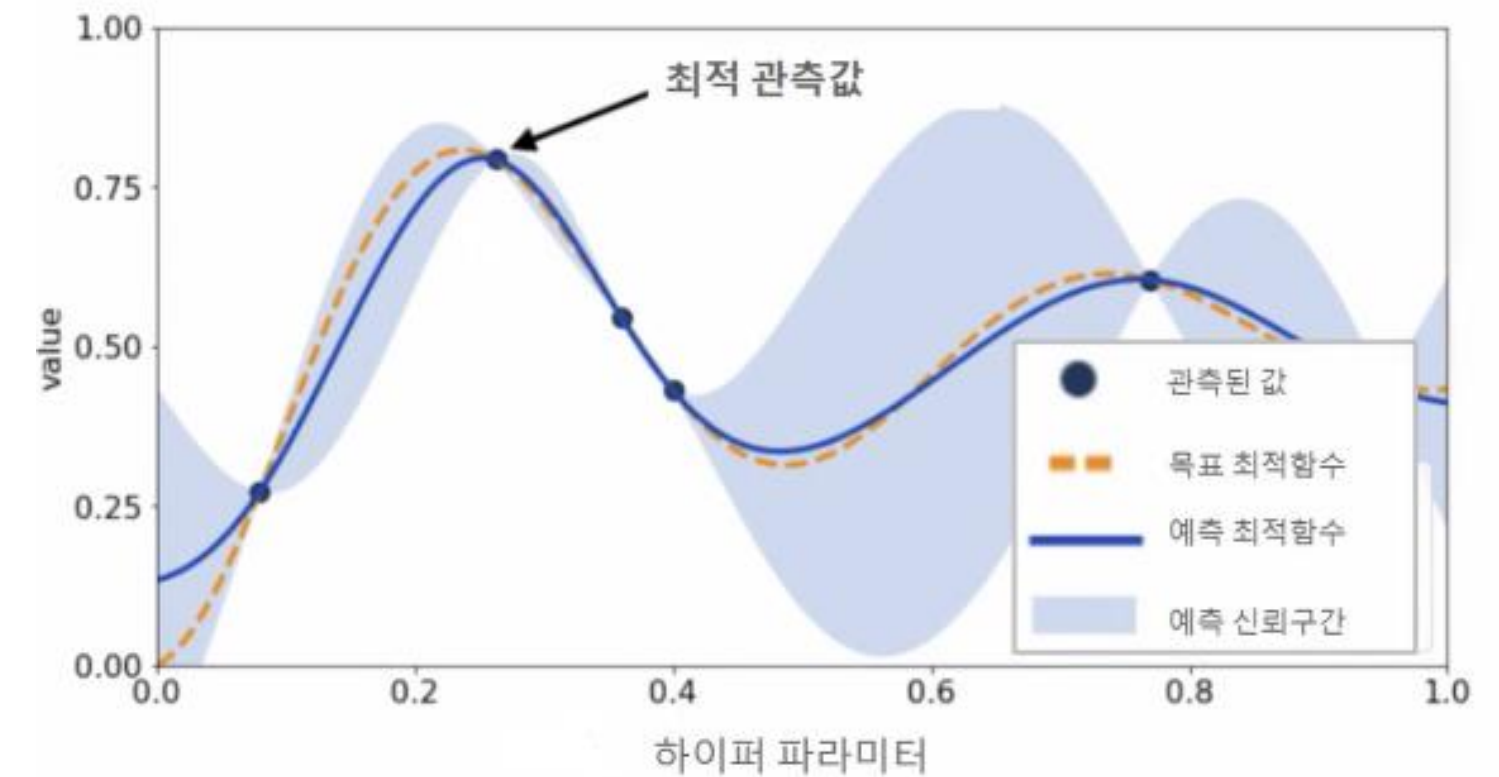


#Step4

Step3에서 얻은 하이퍼 파라미터 값으로 성능 결과 관측

->해당 값 기반으로 대체 모델 갱신

-> 최적 함수 예측 추정



Step3, 4 반복 -> 대체 모델 불확실성 개선

HyperOpt

HyperOpt

: 베이지안 최적화 기반 하이퍼 파라미터 튜닝 패키지

다른 패키지들관 달리 목적 함수 반환 값이 최댓값이 아닌 **최솟값**

#1 입력 변수명과 입력값의 검색공간 설정

#2 목적함수의 설정

#3 목적 함수의 반환 최솟값을 갖는 최적 입력값 유추

HyperOpt

#1 검색공간 설정

-> 검색공간은 딕셔너리 형태로 설정 {입력 변수명 : 해당 변수의 검색공간}

```
from hyperopt import hp

# -10 ~ 10까지 1간격을 가지는 입력 변수 x와 -15 ~ 15까지 1간격으로 입력 변수 y 설정.
search_space = {'x': hp.quniform('x', -10, 10, 1), 'y': hp.quniform('y', -15, 15, 1) }
```

함수명	기능
hp.quniform(label, low, high, q)	label에 최소 low부터 최대 high까지 q의 간격을 갖는 검색공간 설정
hp.uniform(label, low, high)	최소 low부터 최대 high까지 정규분포 형태의 검색 공간 설정
hp.radiant(label, upper)	0부터 최대 upper까지 랜덤한 정숫값으로 검색 공간 설정
hp.loguniform(label, low, high)	exp(uniform(low, high))값 반환, 반환 값의 log값이 정규분포 형태인 검색공간 설정
hp.choice(label,options)	검색 값에 문자열이 있는 경우 설정, options는 리스트, 튜플로 설정

HyperOpt

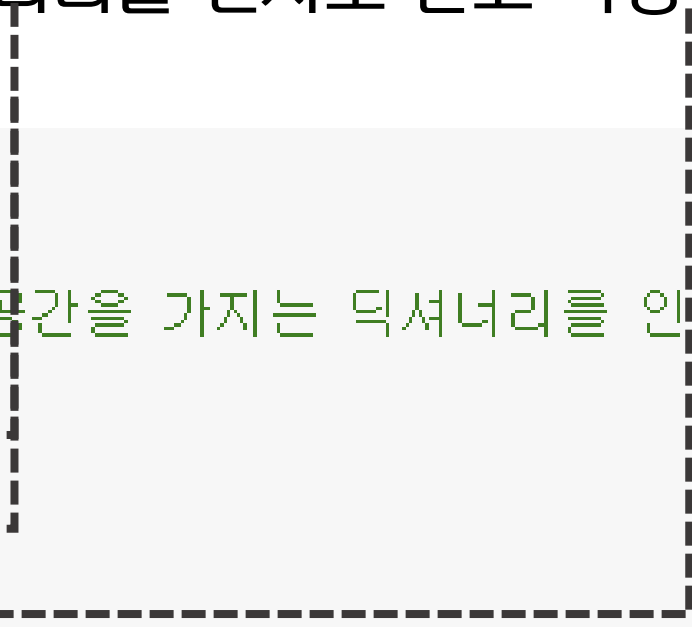
#2 목적 함수 설정

-> 변수값과 검색공간을 가지는 딕셔너리를 인자로 받고 특정 값을 반환

```
from hyperopt import STATUS_OK

# 목적 함수를 생성. 변수값과 변수 검색 공간을 가지는 딕셔너리를 인자로 받고, 특정 값을 반환
def objective_func(search_space):
    x = search_space['x']
    y = search_space['y']
    retval = x**2 - 20*y

    return retval
```



HyperOpt

#3 최적 입력값 탐색

-> `fmin(fn, space, algo, max_evals, trials)`

- `fn` : 목적 함수
- `space` : 검색 공간 디렉터리
- `algo` : 베이지안 최적화 적용 알고리즘 (default = `tpe.suggest`)
- `max_evals` : 입력값 시도 횟수
- `trials` : 최적 입력값을 찾을 때 시도한 입력값, 해당 입력값의 목적 함수 반환값 저장(Trials 클래스)
- `rstate` : 함수 수행마다 동일한 결과값을 갖도록 설정하는 시드값

HyperOpt

#3 최적 입력값 탐색

```
from hyperopt import fmin, tpe, Trials
import numpy as np

# 입력 결과값을 저장한 Trials 객체값 생성.
trial_val = Trials()

# 목적 함수의 최솟값을 반환하는 최적 입력 변수값을 5번의 입력값 시도(max_evals=5)로 찾아냄.
best_01 = fmin(fn=objective_func, space=search_space, algo=tpe.suggest, max_evals=5
               , trials=trial_val, rstate=np.random.default_rng(seed=0))
print('best:', best_01)
```

```
100%|██████████| 5/5 [00:00<00:00, 325.42trial/s, best loss: -224.0]
best: {'x': -4.0, 'y': 12.0}
```

```
trial_val = Trials()

# max_evals를 20회로 늘려서 재테스트
best_02 = fmin(fn=objective_func, space=search_space, algo=tpe.suggest, max_evals=20
               , trials=trial_val, rstate=np.random.default_rng(seed=0))
print('best:', best_02)
```

```
100%|██████████| 20/20 [00:00<00:00, 457.10trial/s, best loss: -296.0]
best: {'x': 2.0, 'y': 15.0}
```

주어진 검색공간에서

$x^2 - 20y$ 값의 최솟값:

X=0, y=15일 때, -300

-> max_evals가 5에서 20으로 늘어남에 따라 best loss 값이 더 작아지고 실제 최솟값에 가까워지는 것을 볼 수 있음

*해당 시드에선 137번째에서 -300이 나옴

HyperOpt

trial_val.results

-> { 'loss' : 함수 반환값, 'status' : 반환 상태값 } 저장

trial_val.vals

-> { '입력변수명' : 개별 수행 시마다 입력된 값 리스트 } 저장

-> 해당 속성들로 DataFrame 생성

```
import pandas as pd

# results에서 loss 키값에 해당하는 밸류들을 추출하여 list로 생성.
losses = [loss_dict['loss'] for loss_dict in trial_val.results]

# DataFrame으로 생성.
result_df = pd.DataFrame({'x': trial_val.vals['x'], 'y': trial_val.vals['y'], 'losses': losses})
result_df
```

	x	y	losses
0	-6.0	5.0	-64.0
1	-4.0	10.0	-184.0
2	4.0	-2.0	56.0
3	-4.0	12.0	-224.0
4	9.0	1.0	61.0
5	2.0	15.0	-296.0
6	10.0	7.0	-40.0
		⋮	
13	6.0	10.0	-164.0
14	9.0	3.0	21.0
15	2.0	3.0	-56.0
16	-2.0	-14.0	284.0
17	-4.0	-8.0	176.0
18	7.0	11.0	-171.0
19	-0.0	-0.0	0.0

HyperOpt를 이용한 XGBoost 최적화

```
# 전체 데이터 중 80%는 학습용 데이터, 20%는 테스트용 데이터 추출
X_train, X_test, y_train, y_test=train_test_split(X_features, y_label, test_size=0.2, ra

# 앞에서 추출한 학습 데이터를 다시 학습과 검증 데이터로 분리
X_tr, X_val, y_tr, y_val= train_test_split(X_train, y_train, test_size=0.1, random_state

from hyperopt import hp

xgb_search_space = {'max_depth': hp.quniform('max_depth', 5, 20, 1),
                    'min_child_weight': hp.quniform('min_child_weight', 1, 2, 1),
                    'learning_rate': hp.uniform('learning_rate', 0.01, 0.2),
                    'colsample_bytree': hp.uniform('colsample_bytree', 0.5, 1),
                    }

from sklearn.model_selection import cross_val_score
from xgboost import XGBClassifier
from hyperopt import STATUS_OK

def objective_func(search_space):
    xgb_clf = XGBClassifier(n_estimators=100, max_depth=int(search_space['max_depth']),
                           min_child_weight=int(search_space['min_child_weight']),
                           learning_rate=search_space['learning_rate'],
                           colsample_bytree=search_space['colsample_bytree'],
                           eval_metric='logloss')
    accuracy = cross_val_score(xgb_clf, X_train, y_train, scoring='accuracy', cv=3)

    return {'loss':-1 * np.mean(accuracy), 'status': STATUS_OK}
```

'max_depth': [5.0, 6.0, 7.0, ..., 20.0]
'min_child_weight': [1.0, 2.0]
'learning_rate': [0.01에서 0.2까지 정규분포된 값]
'colsample_bytree': [0.5에서 1까지 정규분포된 값]

해당 파라미터엔 정수형이 입력되어야 하므로
실수형을 정수형으로 형변환해서 입력

Accuracy는 더 높을 수록 좋은 평가지표
-> -1을 곱해서 음수로 만들어줌
-> HyperOpt는 최솟값을 반환할 수 있도록
최적화하기 때문

HyperOpt를 이용한 XGBoost 최적화

```
from hyperopt import fmin, tpe, Trials

trial_val = Trials()
best = fmin(fn=objective_func,
           space=xgb_search_space,
           algo=tpe.suggest,
           max_evals=50, # 최대 반복 횟수를 지정합니다.
           trials=trial_val, rstate=np.random.default_rng(seed=9))
print('best:', best)
```

```
100%|██████████| 50/50 [00:42<00:00, 1.18trial/s, best loss: -0.9670616939700244]
best: {'colsample_bytree': 0.9599446282177103, 'learning_rate': 0.15480405522751015, 'max_depth': 6.0, 'min_child_weight': 2.0}
```

```
print('colsample_bytree:{0}, learning_rate:{1}, max_depth:{2}, min_child_weight:{3}'.format(
    round(best['colsample_bytree'], 5), round(best['learning_rate'], 5),
    int(best['max_depth']), int(best['min_child_weight'])))
```

```
colsample_bytree:0.95994, learning_rate:0.1548, max_depth:6, min_child_weight:2
```

-> colsample_bytree:0.95994, learning_rate:0.1548,
max_depth:6, min_child_weight:2 에서 가장 높은 정확도를 가짐
해당 하이퍼 파라미터 설정 시의 정확도 : 0.96706169

HyperOpt를 이용한 XGBoost 최적화

```
xgb_wrapper = XGBClassifier(n_estimators=400,
                             learning_rate=round(best['learning_rate'], 5),
                             max_depth=int(best['max_depth']),
                             min_child_weight=int(best['min_child_weight']),
                             colsample_bytree=round(best['colsample_bytree'], 5)
                             )

evals = [(X_tr, y_tr), (X_val, y_val)]
xgb_wrapper.fit(X_tr, y_tr, early_stopping_rounds=50, eval_metric='logloss',
                eval_set=evals, verbose=True)

preds = xgb_wrapper.predict(X_test)
pred_proba = xgb_wrapper.predict_proba(X_test)[:, 1]

get_clf_eval(y_test, preds, pred_proba)
```

오차 행렬

[[34 3]

[2 75]]

정확도: 0.9561, 정밀도: 0.9615, 재현율: 0.9740, F1: 0.9677, AUC:0.9895



XGBoost : 0.9474

HyperOpt 적용 XGBoost : 0.9561

정확도가 약간 향상됨

⚠ 해당 데이터는 데이터 건수도 작기 때문에 rstate가 바뀔 경우엔 성능 결과가 불안정해질 수 있음

4.11 스테킹 앙상블



#4.11 스택킹 앙상블

스택킹 앙상블

- 개별 알고리즘의 예측 결과 데이터 세트를 최종적인 메타 데이터 세트로 만들어 별도의 ML 알고리즘으로 최종 학습을 수행하고 테스트 데이터를 기반으로 다시 최종 예측을 수행하는 방식
- **메타 모델** : 개별 모델의 예측된 데이터 세트를 다시 기반으로 하여 학습하고 예측하는 방식

배깅(Bagging), 부스팅(Boosting)과 비교

공통점	개별적인 여러 알고리즘을 서로 결합해 예측 결과를 도출함
차이점	개별 알고리즘으로 예측한 데이터를 기반으로 다시 예측을 수행함

#4.11 스택킹 앙상블

스택킹 앙상블

- 스택킹 앙상블은 두 종류의 모델이 필요
 - 개별적인 기반 모델
 - 개별 기반 모델의 예측 데이터를 학습 데이터로 만들어서 학습하는 최종 메타 모델

스택킹 모델의 핵심

여러 개별 모델의 예측 데이터를 각각 스택킹 형태로 결합해

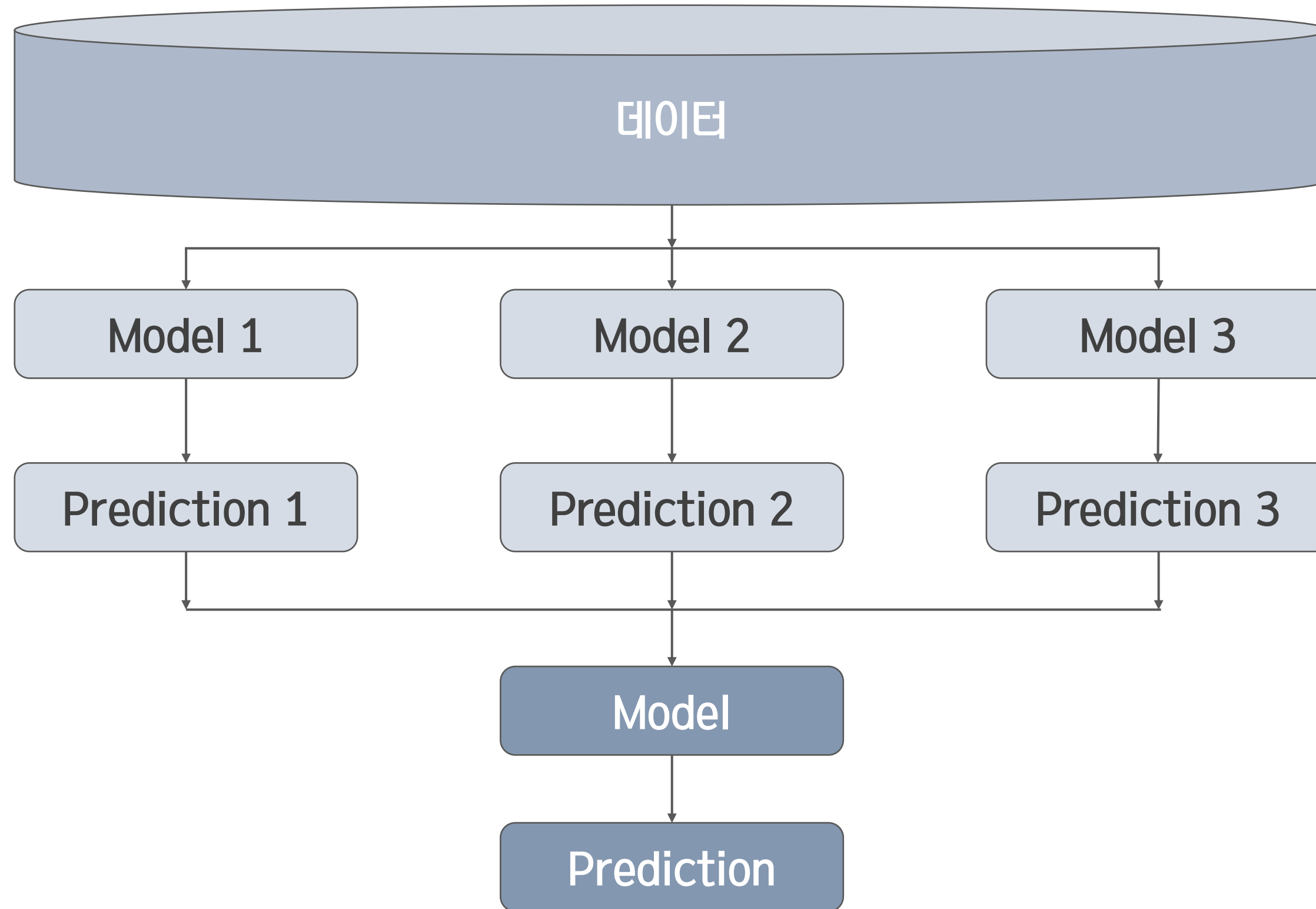
최종 메타모델의 학습용 피쳐 데이터 세트와 테스트용 피쳐 데이터 세트를 만드는 것

#4.11 스택킹 앙상블

스택킹 앙상블의 특징

- 현실 모델에 적용하는 경우는 많지 않지만 대회에서 높은 순위를 차지하기 위해 성능 수치를 조금이라도 높여야 할 경우 자주 사용됨
- 많은 개별 모델이 필요함
 - 2~3개의 개별 모델만을 결합해서는 쉽게 예측 성능을 향상시킬 수 없기 때문
 - 또한 스택킹을 적용한다고 해서 반드시 성능 향상이 되리라는 보장도 없음
- 일반적으로 성능이 비슷한 모델을 결합해 좀 더 나은 성능 향상을 도출하기 위해 적용됨

#4.11 스택킹 앙상블



#4.11 스택킹 앙상블

기본 스택킹 모델 적용

- 개별 모델 : KNN, 랜덤 포레스트, 결정 트리, 에이다부스트 모델
- 최종 모델 : 로지스틱 회귀 모델



개별 ML 모델을 위한 Classifier 생성.

```
knn_clf = KNeighborsClassifier(n_neighbors=4)
```

```
rf_clf = RandomForestClassifier(n_estimators=100, random_state=0)
```

```
dt_clf = DecisionTreeClassifier()
```

```
ada_clf = AdaBoostClassifier(n_estimators=100)
```

최종 Stacking 모델을 위한 Classifier 생성.

```
lr_final = LogisticRegression(C=10)
```

#4.11 스택킹 앙상블

기본 스택킹 모델 적용

- 개별 모델의 예측 데이터 세트를 반환하고 각 모델의 예측 정확도 측정

```
# 학습된 개별 모델들이 각자 반환하는 예측 데이터 셋을 생성하고 개별 모델의 정확도 측정 .
knn_pred = knn_clf.predict(X_test)
rf_pred = rf_clf.predict(X_test)
dt_pred = dt_clf.predict(X_test)
ada_pred = ada_clf.predict(X_test)

print('KNN 정확도: {0:.4f}'.format(accuracy_score(y_test, knn_pred)))
print('랜덤 포레스트 정확도: {0:.4f}'.format(accuracy_score(y_test, rf_pred)))
print('결정 트리 정확도: {0:.4f}'.format(accuracy_score(y_test, dt_pred)))
print('에이다부스트 정확도: {0:.4f}'.format(accuracy_score(y_test, ada_pred)))
```

```
KNN 정확도: 0.9211
랜덤 포레스트 정확도: 0.9649
결정 트리 정확도: 0.9123
에이다부스트 정확도: 0.9561
```

#4.11 스택킹 앙상블

기본 스택킹 모델 적용

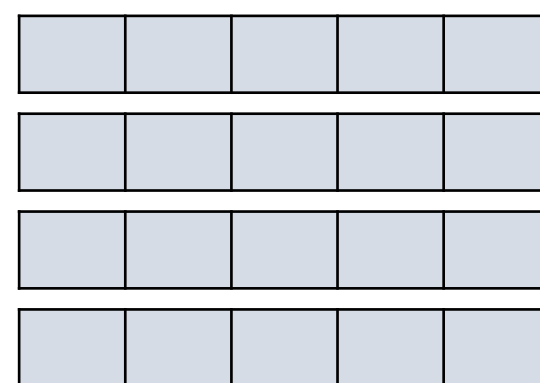
- 개별 알고리즘으로부터 예측된 예측값을 칼럼 레벨로 옆으로 붙여서 피쳐 값으로 만들기
- 반환된 1차원 형태의 ndarray를 행 형태로 붙인 뒤, 넘파이의 transpose()를 이용해 행/열 전환

```
=== KNN 예측값
[0 1 1 0 1 1 1 1 1 1 0 1 1 0 0 0 1 0 0 0 0 0 1 1 0 1 1 1 1 0]
=== 랜덤 포레스트 예측값
[0 1 1 1 1 1 1 1 1 1 0 1 1 0 0 0 1 0 0 0 0 0 1 1 0 1 1 0 1 0]
=== 결정 트리 예측값
[0 1 1 1 1 1 1 1 1 1 0 1 1 0 0 0 1 0 0 0 0 0 1 1 0 1 1 0 1 0]
=== 에이다부스트 예측값
[0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 0 0 0 0 0 1 1 0 1 1 0 1 0]
```

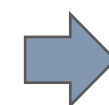
```
pred = np.array([knn_pred, rf_pred, dt_pred, ada_pred])
```

행 형태로 붙여서 피쳐 값으로 만들기

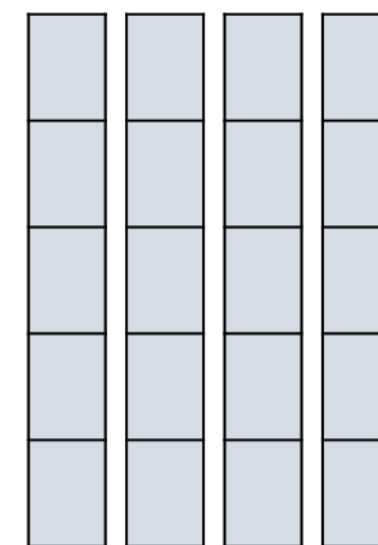
```
[[0 1 1 0 1 1 1 1 1 1 0 1 1 0 0 0 1 0 0 0 0 0 1 1 0 1 1 1 1 0]
 [0 1 1 1 1 1 1 1 1 1 0 1 1 0 0 0 1 0 0 0 0 0 1 1 0 1 1 0 1 0]
 [0 1 1 1 1 1 1 1 1 1 0 1 1 0 0 0 1 0 0 0 0 0 1 1 0 1 1 0 1 0]
 [0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 0 0 0 0 0 1 1 0 1 1 0 1 0]]
```



(4, 114)



transpose()



(114, 4)

- KNN
- 랜덤 포레스트
- 결정 트리
- 에이다부스트

(이해를 돕기 위한 예측값 ndarray의 일부입니다. [:30]개만 가져옴)

#4.11 스택킹 앙상블

기본 스택킹 모델 적용

- 최종 메타 모델에서 스택킹으로 재구성한 개별 모델 예측 데이터를 학습/예측한 결과 정확도가 97.37%로 개별 모델 정확도보다 향상되었음
- 하지만 스택킹 기법으로 예측을 한다고 무조건 개별 모델보다 좋아진다는 보장은 없음

```
lr_final.fit(pred, y_test)
final = lr_final.predict(pred)

print('최종 메타 모델의 예측 정확도: {0:.4f}'.format(accuracy_score(y_test , final)))
```

최종 메타 모델의 예측 정확도: 0.9737

KNN 정확도: 0.9211
랜덤 포레스트 정확도: 0.9649
결정 트리 정확도: 0.9123
에이다부스트 정확도: 0.9561

#4.11 스택킹 앙상블

CV 세트 기반의 스택킹 모델 적용

- 과적합을 개선하기 위한 모델
- 최종 메타 모델을 위한 데이터 세트를 만들 때 교차 검증(Cross Validation, CV) 기반으로 예측된 결과 데이터 세트를 이용함
- 앞의 예제에서 메타 모델인 로지스틱 회귀 모델 기반에서 최종 학습할 때 레이블 데이터 세트로 학습 데이터 세트가 아닌 테스트용 레이블 데이터 세트를 기반으로 학습했기에 과적합 문제가 발생할 수 있음

```
lr_final.fit(pred, y_test)
final = lr_final.predict(pred)
```

- 다시 말해, 위의 코드를 보면 pred 데이터셋으로 학습(fit)을 수행하고
- 똑같이 pred 데이터셋으로 예측(predict)을 수행하기 때문에 과적합 문제가 발생할 수 있음

#4.11 스택킹 앙상블

CV 세트 기반의 스택킹 모델 적용

- CV 세트 기반의 스택킹은 과적합 문제를 개선하기 위해 개별 모델들이 각각 교차 검증으로 메타 모델을 위한 학습용 스택킹 데이터 생성과 예측을 위한 테스트용 스택킹 데이터를 생성한 뒤 이를 기반으로 메타 모델이 학습과 예측을 수행함
- 기존 스택킹 방식 : 메타 모델의 학습과 예측을 위한 스택킹 데이터가 동일함 → 과적합 문제 발생
- CV 세트 기반 스택킹 방식 : 메타 모델을 위한 학습용 스택킹 데이터와 테스트용 스택킹 데이터를 분리

#4.11 스택킹 앙상블

CV 세트 기반의 스택킹 모델 적용

- 2단계의 스텝으로 구성됨

스텝 1	각 모델별로 원본 학습/테스트 데이터를 예측한 결과 값을 기반으로 메타 모델을 위한 학습용/테스트용 데이터를 생성함
스텝 2	스텝 1에서 개별 모델들이 생성한 학습용 데이터를 모두 스택킹 형태로 합쳐서 메타 모델이 학습할 최종 학습용 데이터 세트를 생성함 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 평가함

#4.11 스택킹 앙상블

CV 세트 기반의 스택킹 모델 적용

- 핵심 : 개별 모델에서 메타 모델인 2차 모델에서 사용될 학습용 데이터와 테스트용 데이터를 교차 검증을 통해 생성하는 것
- 개별 모델 레벨에서 여러 개의 개별 모델 모두 동일하게 스텝 1을 수행함

#4.11 스택킹 앙상블

CV 세트 기반의 스택킹 모델 적용

< 스텝 1 설명 전에 K-Fold 교차 검증 방식 복습 >

- K개의 데이터 폴드 세트를 만들어서 K번만큼 각 폴드 세트에 학습과 검증 평가를 반복적으로 수행하는 방법
- 예시 : K=4일 때, 총 4개의 폴드 세트에 4번의 학습 및 검증 평가 반복

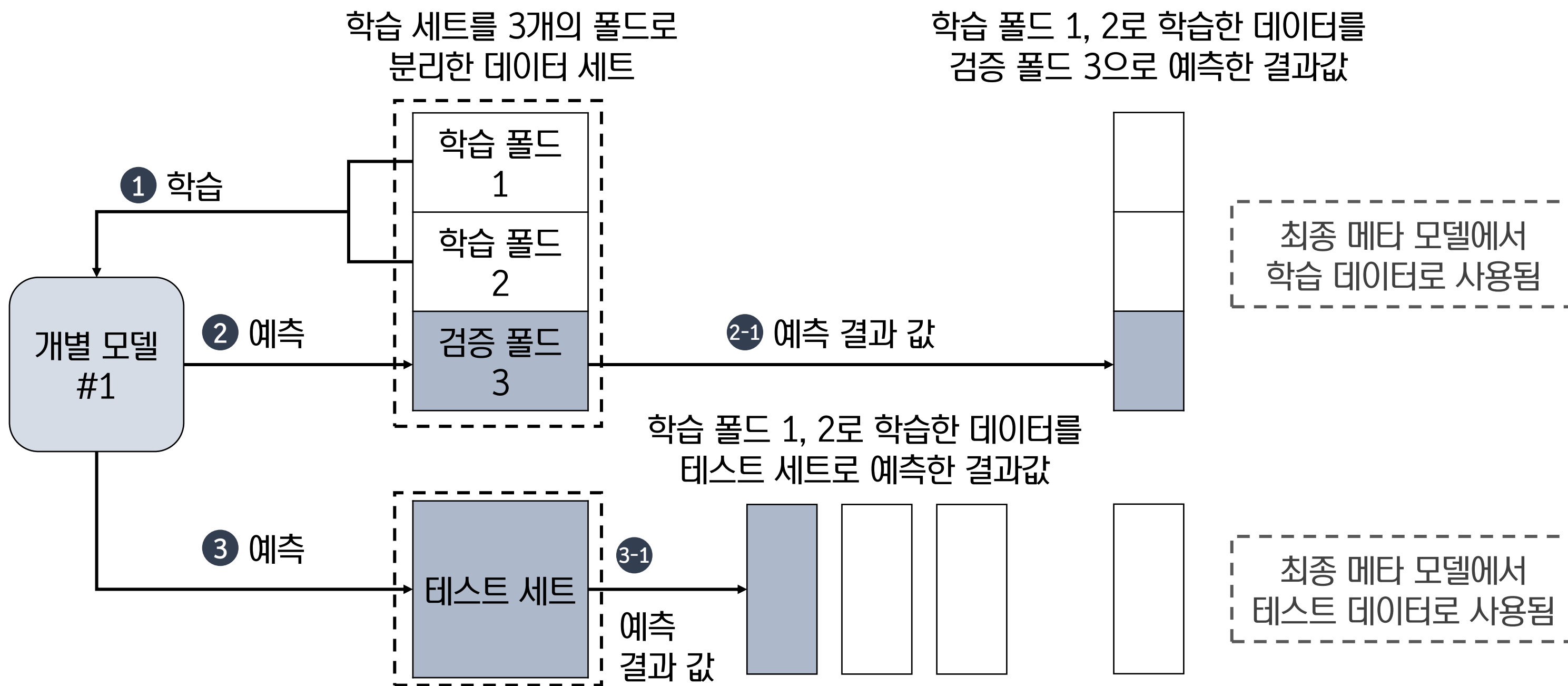


학습 데이터 세트와
검증 데이터 세트를
점진적으로 변경하면서
마지막 4번째(K번째)까지
학습과 검증을 수행한다

#4.11 스택킹 앙상블

CV 세트 기반의 스택킹 모델 적용

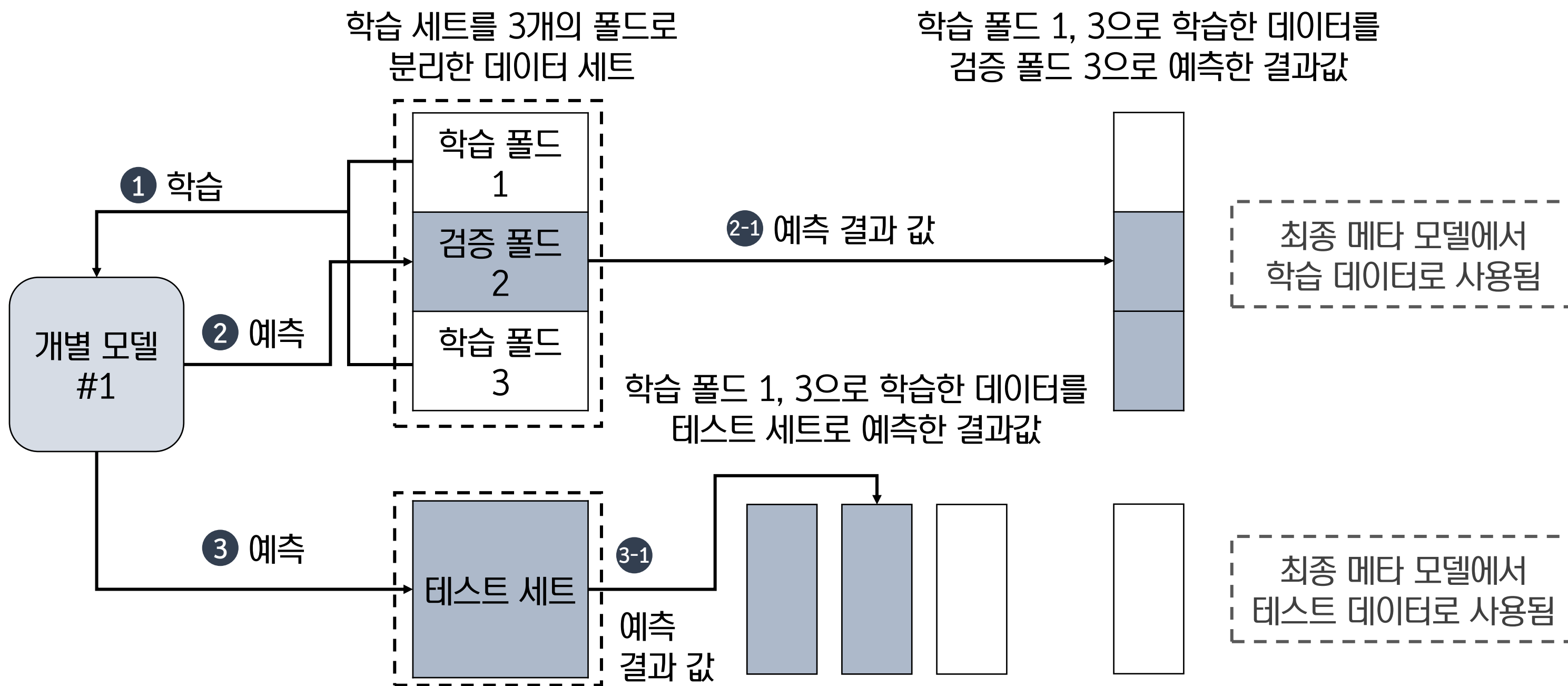
- 스텝 1 - 첫 번째 반복



#4.11 스택킹 앙상블

CV 세트 기반의 스택킹 모델 적용

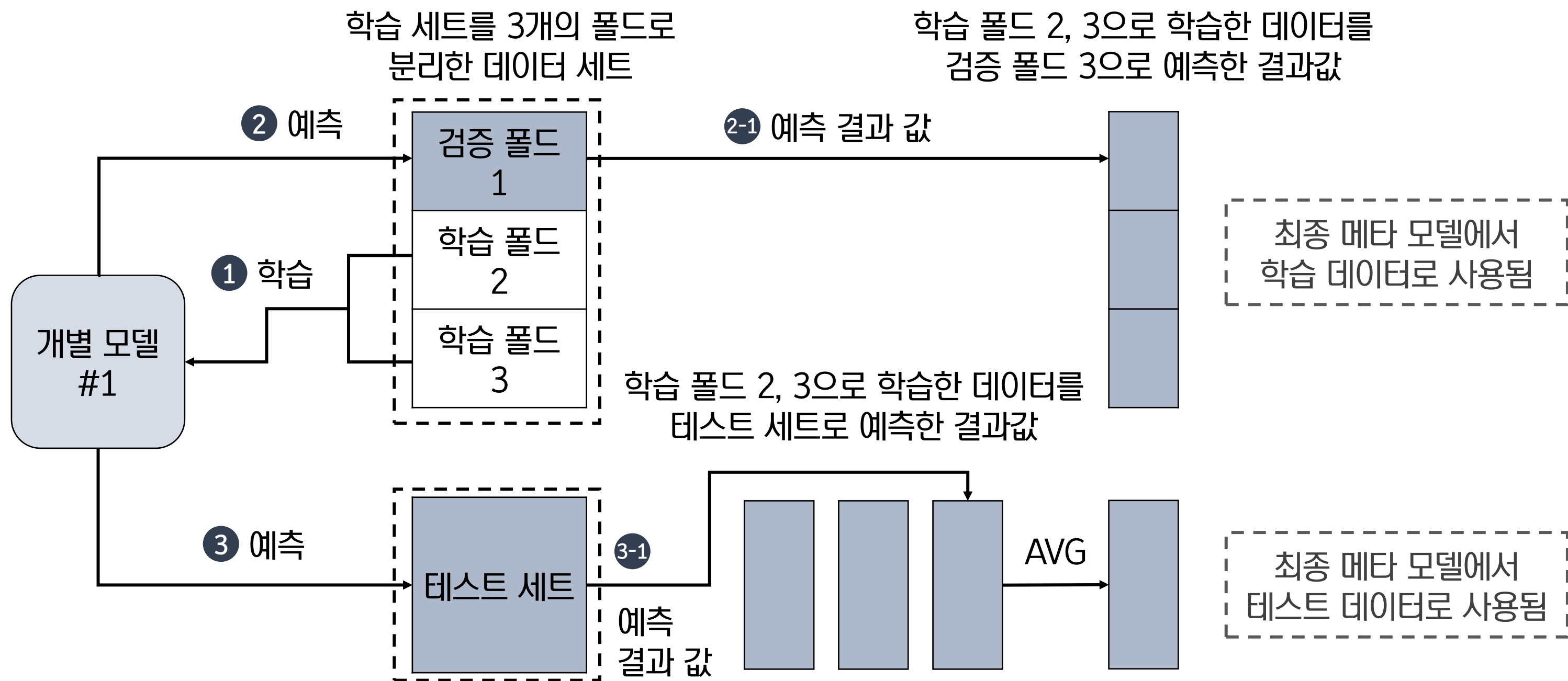
- 스텝 1 - 두 번째 반복



#4.11 스택킹 앙상블

CV 세트 기반의 스택킹 모델 적용

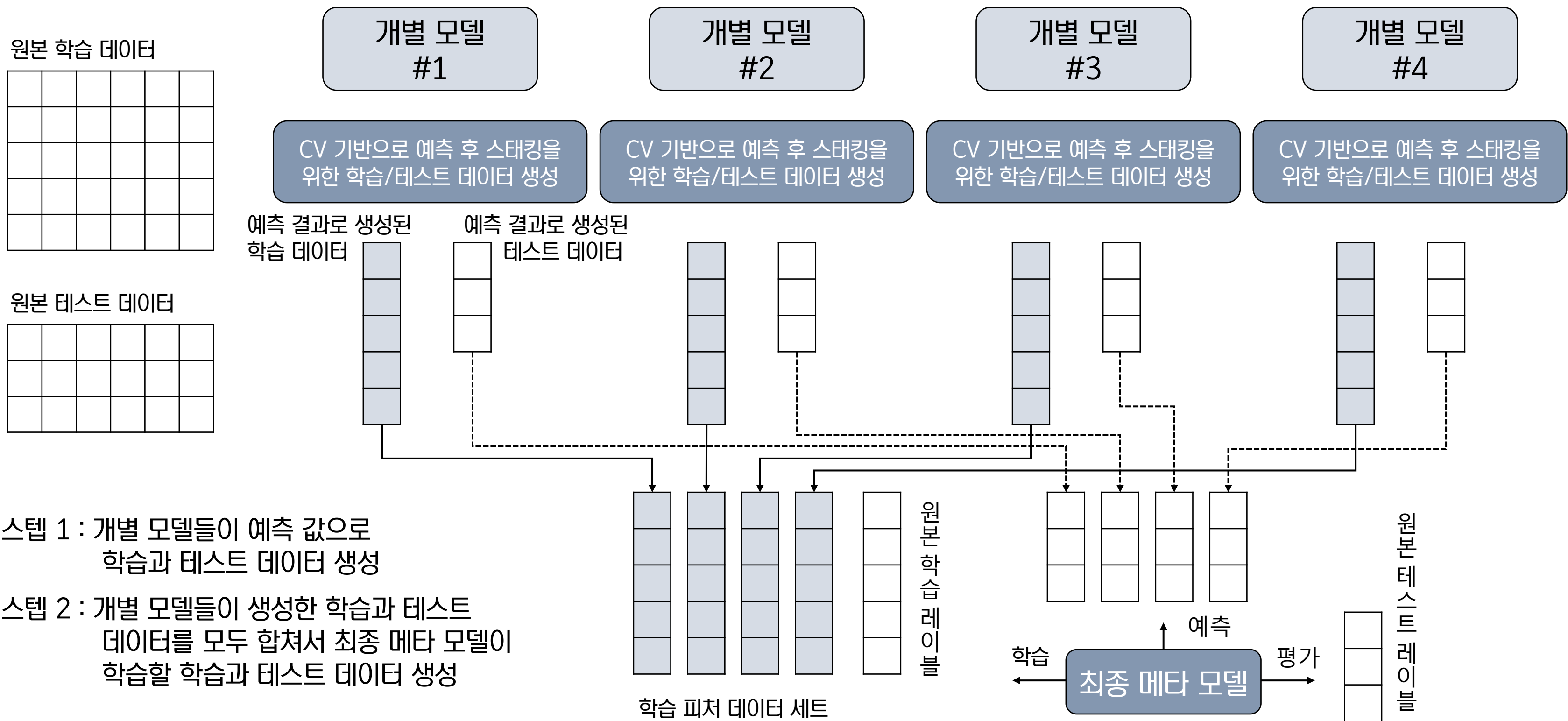
- 스텝 1 - 세 번째 반복



#4.11 스택킹 앙상블

CV 세트 기반의 스택킹 모델 적용

- 모델 전체 도식화



#4.11 스택킹 앙상블

CV 세트 기반의 스택킹 모델 적용

- 스텝 1 코드 구현
- 파라미터
 - 개별 모델의 Classifier 객체
 - 원본 학습 피쳐 데이터
 - 원본 학습 레이블 데이터
 - 원본 테스트 피쳐 데이터
 - K폴드 인자

```
from sklearn.model_selection import KFold
from sklearn.metrics import mean_absolute_error

# 개별 기반 모델에서 최종 메타 모델이 사용할 학습 및 테스트용 데이터를 생성하기 위한 함수.
def get_stacking_base_datasets(model, X_train_n, y_train_n, X_test_n, n_folds):
    # 지정된 n_folds값으로 KFold 생성.
    kf = KFold(n_splits=n_folds, shuffle=False)
    #추후에 메타 모델이 사용할 학습 데이터 반환을 위한 넘파이 배열 초기화
    train_fold_pred = np.zeros((X_train_n.shape[0], 1))
    test_pred = np.zeros((X_test_n.shape[0], n_folds))
    print(model.__class__.__name__, ' model 시작 ')

    for folder_counter, (train_index, valid_index) in enumerate(kf.split(X_train_n)):
        #입력된 학습 데이터에서 기반 모델이 학습/예측할 폴드 데이터 셋 추출
        print('### 폴드 세트: ', folder_counter, ' 시작 ')
        X_tr = X_train_n[train_index]
        y_tr = y_train_n[train_index]
        X_te = X_train_n[valid_index]

        #폴드 세트 내부에서 다시 만들어진 학습 데이터로 기반 모델의 학습 수행.
        model.fit(X_tr, y_tr)
        #폴드 세트 내부에서 다시 만들어진 검증 데이터로 기반 모델 예측 후 데이터 저장.
        train_fold_pred[valid_index, :] = model.predict(X_te).reshape(-1, 1)
        #입력된 원본 테스트 데이터를 폴드 세트내 학습된 기반 모델에서 예측 후 데이터 저장.
        test_pred[:, folder_counter] = model.predict(X_test_n)

    # 폴드 세트 내에서 원본 테스트 데이터를 예측한 데이터를 평균하여 테스트 데이터로 생성
    test_pred_mean = np.mean(test_pred, axis=1).reshape(-1, 1)

    #train_fold_pred는 최종 메타 모델이 사용하는 학습 데이터, test_pred_mean은 테스트 데이터
    return train_fold_pred, test_pred_mean
```

#4.11 스택킹 앙상블

CV 세트 기반의 스택킹 모델 적용

- 개별 분류 모델 별 학습 수행

```
knn_train, knn_test = get_stacking_base_datasets(knn_clf, X_train, y_train, X_test, 7)
rf_train, rf_test = get_stacking_base_datasets(rf_clf, X_train, y_train, X_test, 7)
dt_train, dt_test = get_stacking_base_datasets(dt_clf, X_train, y_train, X_test, 7)
ada_train, ada_test = get_stacking_base_datasets(ada_clf, X_train, y_train, X_test, 7)
```

- 스텝 2 코드 구현 – concatenate() : 여러 개의 넘파이 배열을 칼럼 또는 로우 레벨로 합쳐줌

```
Stack_final_X_train = np.concatenate((knn_train, rf_train, dt_train, ada_train), axis=1)
Stack_final_X_test = np.concatenate((knn_test, rf_test, dt_test, ada_test), axis=1)
print('원본 학습 피쳐 데이터 Shape:', X_train.shape, '원본 테스트 피쳐 Shape:', X_test.shape)
print('스택킹 학습 피쳐 데이터 Shape:', Stack_final_X_train.shape,
      '스택킹 테스트 피쳐 데이터 Shape:', Stack_final_X_test.shape)
```

원본 학습 피쳐 데이터 Shape: (455, 30) 원본 테스트 피쳐 Shape: (114, 30)

스택킹 학습 피쳐 데이터 Shape: (455, 4) 스택킹 테스트 피쳐 데이터 Shape: (114, 4)

#4.11 스택킹 앙상블

CV 세트 기반의 스택킹 모델 적용

- 최종 메타 모델 학습/예측 및 정확도 측정

```
lr_final.fit(Stack_final_X_train, y_train)
stack_final = lr_final.predict(Stack_final_X_test)

print('최종 메타 모델의 예측 정확도: {0:.4f}'.format(accuracy_score(y_test, stack_final)))
```

최종 메타 모델의 예측 정확도: 0.9825

- 최종 메타 모델의 예측 정확도 약 98.25%
- 스택킹을 이루는 모델은 최적으로 파라미터를 튜닝한 상태에서 스택킹 모델을 만드는 것이 일반적
- 스택킹 모델의 파라미터 튜닝은 일반적으로 개별 알고리즘 모델의 파라미터를 최적으로 튜닝하는 것
- 스택킹 모델은 분류(Classification)뿐만 아니라 회귀(Regression)에도 적용 가능

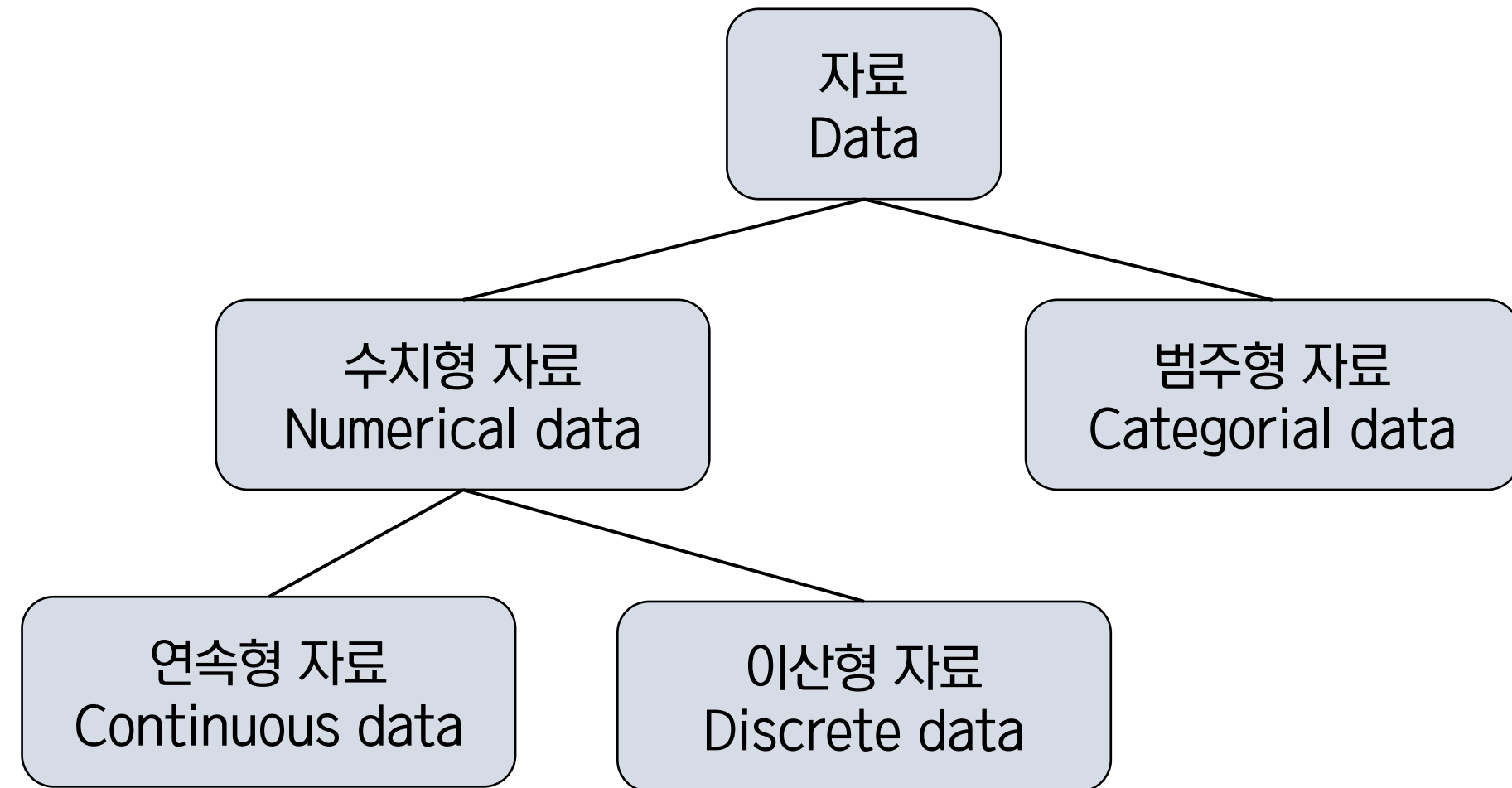
변외. CatBoost



#번외. CatBoost

CatBoost(캣부스트) Classifier

- 자료의 형태 분류
- 수치형 자료
 - 관측된 값이 수치로 측정되는 자료
 - 키, 몸무게, 시험 성적, 자동차 사고 건수
- 수치형 자료는 관측되는 값의 성질에 따라 연속형 자료, 이산형 자료로 나뉨
- 연속형 자료
 - 키, 몸무게 값처럼 연속적인 자료
- 이산형 자료
 - 자동차 사고 건수와 같이 셀 수 있는 자료

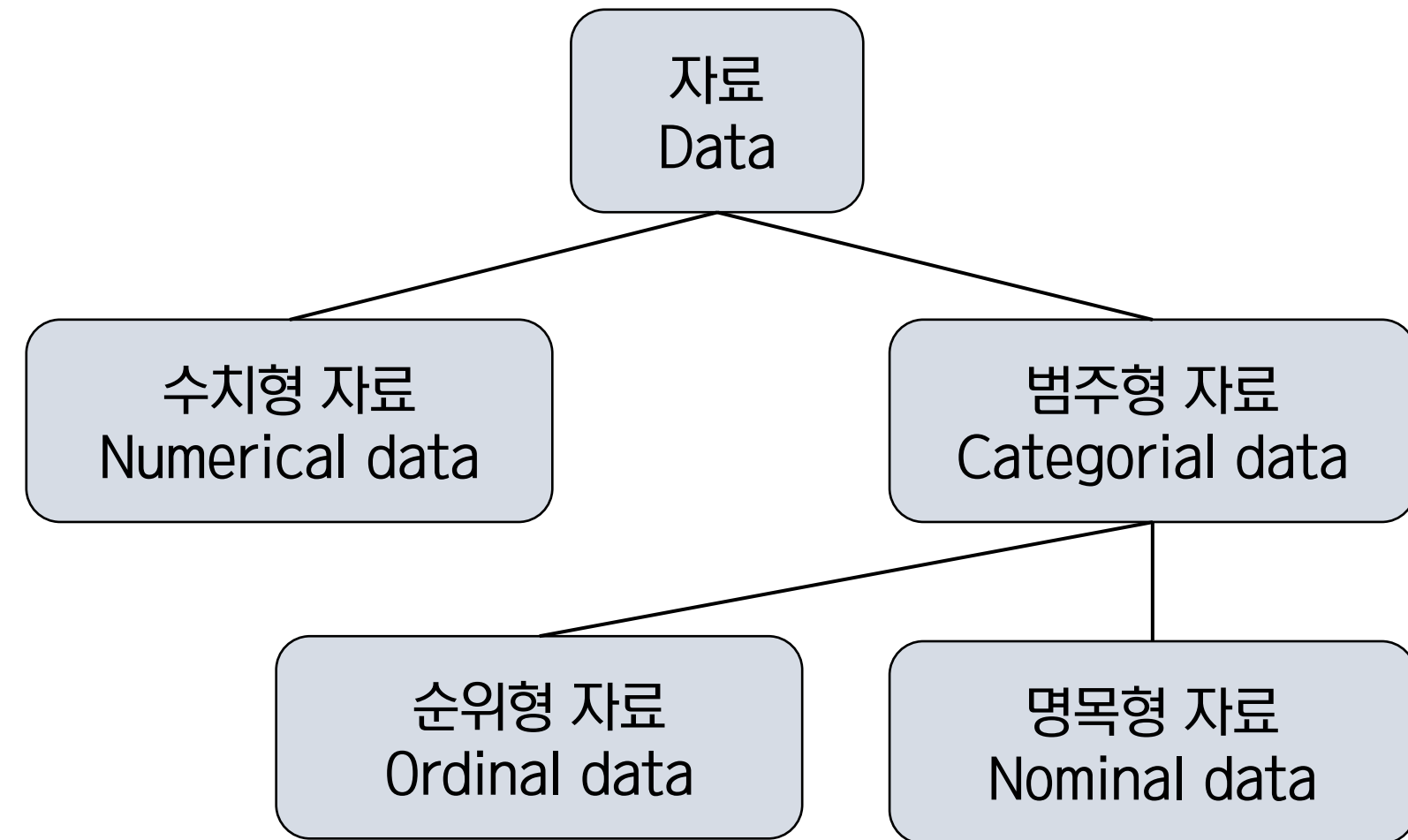


그러나, 연속형 자료라 할지라도 기록의 편리함이나 측정기구의 한계 때문에 반올림되거나 가장 가까운 눈금의 값을 얻게 되어 실제로는 이산형인 값을 갖게 된다.
예를 들면) 키를 잴 때, 센티미터 단위까지만 기록하고 그 이하의 단위는 반올림되므로 기록된 값은 이산의 형태를 갖게 된다.
이때, 기록된 값은 이산의 형태이지만 실제로 관측 가능한 값은 연속적인 척도로 주어지기 때문에 연속형 자료라 한다.

#번외. CatBoost

CatBoost(캣부스트) Classifier

- 범주형 자료
 - 관측 결과가 몇 개의 범주 또는 항목의 형태로 나타나는 자료
 - 성별(남, 여) 선호도(좋다, 보통, 싫다), 혈액형(A, B, O, AB) 등이 있음
 - 수치형 자료처럼 남자=1, 여자=0으로 표현할 수도 있지만 비교우위를 나타내는 것 X
- 순위형 자료
 - 범주 간에 순서의 의미가 있는 자료 (ex. 선호도)
- 명목형 자료
 - 범주 간에 순서의 의미가 없는 자료 (ex. 혈액형)



#번외. CatBoost

CatBoost(캣부스트) Classifier

- Category(범주) + Boosting(부스팅)
- 대부분이 범주형 피처인 자료에서 우수한 성능을 보임
- 의사결정 트리의 그래디언트 부스팅을 위한 알고리즘
- Yandex 연구원들과 엔지니어가 개발했으며, 검색, 추천 시스템, 날씨 예측 등의 작업에 많이 사용됨
- 동급의 알고리즘과 비교해 최고의 성능을 자랑함
- 범주형(categorical) 데이터를 처리하는 새로운 방법인 Ordered TS를 제시함
- 기존의 그래디언트 부스팅 알고리즘을 조작하여 타겟 누수(target leakage)를 개선함
 - 타겟 누수(target leakage) : 예측 시점에서 사용할 수 없는 데이터가 데이터셋에 포함되는 오류
 - 모델이 독립변수들인 x만을 활용하여 종속변수인 y를 예측해야 하는데, y에 대한 정보가 x에 포함되어 있는 것이 타겟 누수 -> 학습 데이터에 정답이 사용되어서 나중에 테스트 시 문제 발생

#번외. CatBoost

CatBoost(캣부스트) 특징

1. Level-wise 방식의 모델 생성

2. Ordered boosting 방식으로 부스팅

- 일반적인 boosting 방식은 전체적인 오차에 대한 값으로 모델을 앙상블하지만, Ordered Boosting은 부분의 오차에 대해 모델을 생성하고 데이터들을 늘려나가는 식입니다.
- 이 방식으로 과적합을 줄이게 됩니다.

3. Ordered Target Encoding(Mean Encoding) 범주형 변수 변환

- 독립변수의 범주형 변수가 종속 변수의 평균으로 인코딩하는 것입니다. 이렇게 했을 때 독립변수가 일반적인 0,1 인코딩했을 때보다 data와의 연관성이 생기게 됩니다.

4. Categorical Feature Combinations 범주형 변수 결합

- 연관된 범주형 변수를 자동으로 결합시켜 줌으로써 변수를 줄입니다.

5. Optimized Parameter tuning

- Catboost 는 기본 파라미터가 최적화가 잘 되어있어 파라미터 튜닝에 크게 신경쓰지 않아도 됩니다.

#번외. CatBoost

CatBoost(캣부스트) 특징

- Ordered boosting방식으로 부스팅

time	datapoint	class label
12:00	x1	10
12:01	x2	12
12:02	x3	9
12:03	x4	4
12:04	x5	52
12:05	x6	22
12:06	x7	33
12:07	x8	34
12:08	x9	32
12:09	x10	12

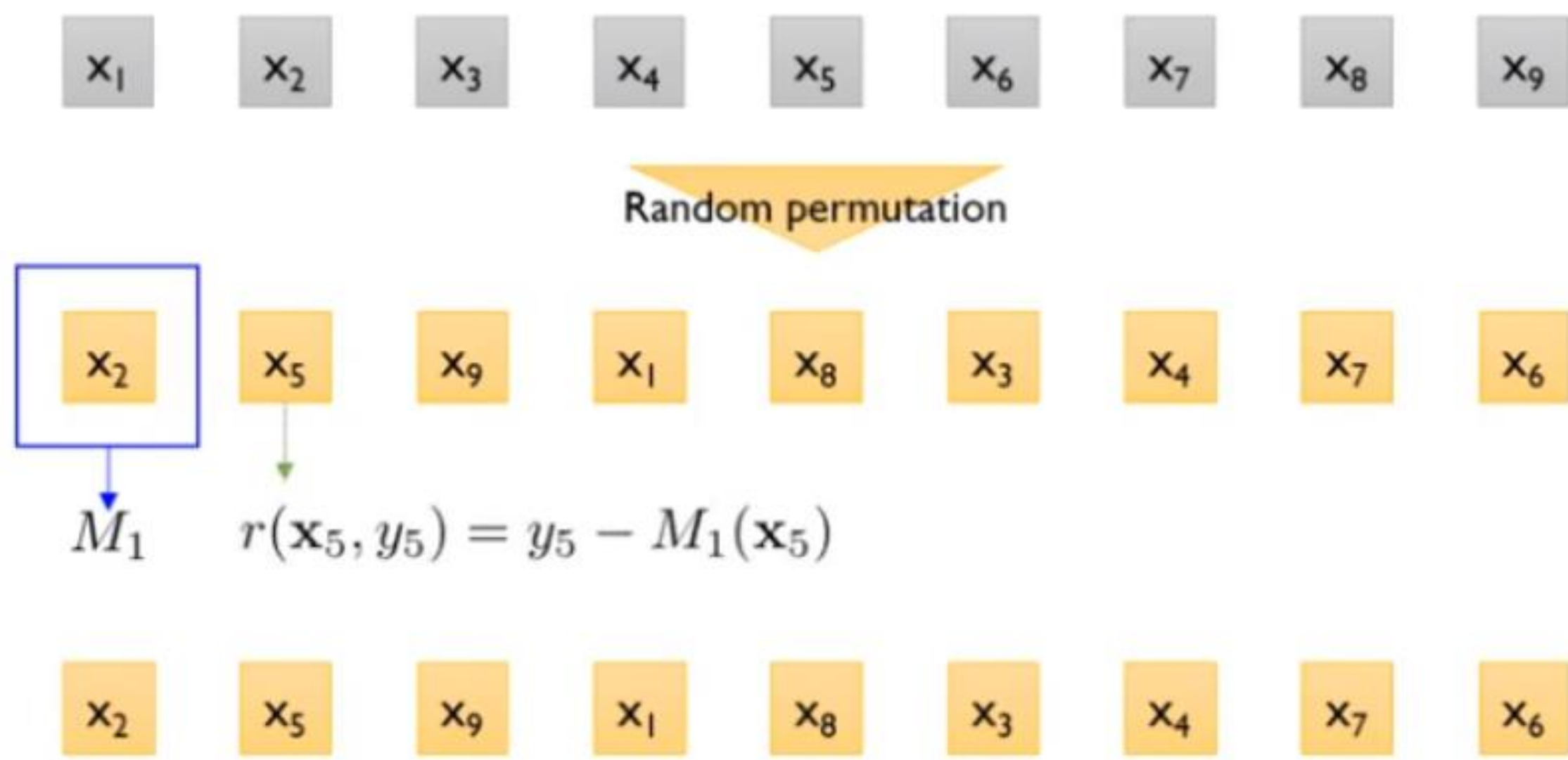
- 일반적인 boosting방식은 전체적인 오차에 대한 값으로 모델을 앙상블하지만, Ordered Boosting은 부분의 오차에 대해 모델을 생성하고 데이터들을 늘려나가는 식입니다.
- 기존 부스팅 방식 : 모든 데이터 (x1-x10) 까지의 잔차를 일괄 계산한다.
- Ordered Boosting
 - x1의 잔차만 계산하고 이를 기반으로 모델을 만든다.
 - x2의 잔차를 이 모델로 예측한다.
 - x1, x2의 잔차로 모델을 만든다.
 - 이를 기반으로 x3, x4의 잔차를 모델로 예측한다.

#번외. CatBoost

CatBoost(캣부스트) 특징

- Ordered boosting 방식으로 부스팅

잔차를 구할 때 y_5, x_5 를 이용하는데 이는 첫번째 모델의 학습 데이터로 사용되어지지 않았기 때문에 두 번째 데이터가 첫번째 모델에 전혀 영향을 미치지 못한다.

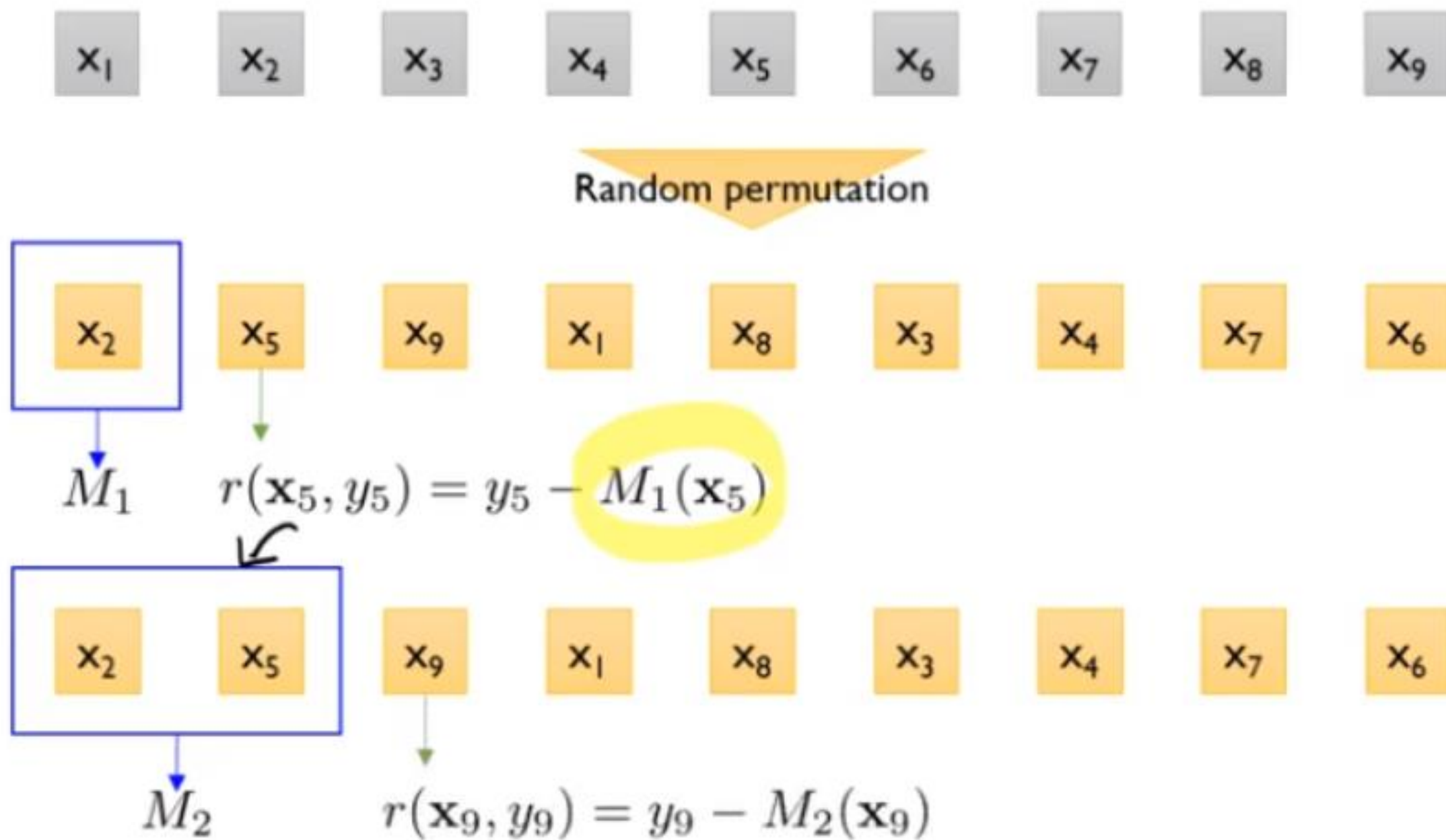


#번외. CatBoost

CatBoost(캣부스트) 특징







- Ordered boosting 방식으로 부스팅

잔차를 구할 때 사용된 y_9, x_9 이 모두 M_2 의 학습에 사용되지 않았기 때문에 세번째 데이터가 모델 M_2 에 전혀 영향을 미치지 않는다.



#번외. CatBoost

CatBoost(캣부스트) 특징

	CatBoost		LightGBM		XGBoost		H2O	
	Tuned	Default	Tuned	Default	Tuned	Default	Tuned	Default
 Adult	0.26974	0.27298 +1.21%	0.27602 +2.33%	0.28716 +6.46%	0.27542 +2.11%	0.28009 +3.84%	0.27510 +1.99%	0.27607 +2.35%
 Amazon	0.13772	0.13811 +0.29%	0.16360 +18.80%	0.16716 +21.38%	0.16327 +18.56%	0.16536 +20.07%	0.16264 +18.10%	0.16950 +23.08%
 Click prediction	0.39090	0.39112 +0.06%	0.39633 +1.39%	0.39749 +1.69%	0.39624 +1.37%	0.39764 +1.73%	0.39759 +1.72%	0.39785 +1.78%
 KDD appetency	0.07151	0.07138 -0.19%	0.07179 +0.40%	0.07482 +4.63%	0.07176 +0.35%	0.07466 +4.41%	0.07246 +1.33%	0.07355 +2.86%
 KDD churn	0.23129	0.23193 +0.28%	0.23205 +0.33%	0.23565 +1.89%	0.23312 +0.80%	0.23369 +1.04%	0.23275 +0.64%	0.23287 +0.69%
 KDD internet	0.20875	0.22021 +5.49%	0.22315 +6.90%	0.23627 +13.19%	0.22532 +7.94%	0.23468 +12.43%	0.22209 +6.40%	0.24023 +15.09%

위의 비교는 테스트 데이터에 대한 로그 손실 값을 나타내며 대부분의 경우 CatBoost의 경우 가장 낮다.
이는 CatBoost가 대부분 튜닝된 모델과 기본 모델 모두에서 더 우수한 성능을 보임을 명확하게 의미한다.
이 외에도 캣부스트는 XGBoost, LightGBM 등 특정 포맷으로 데이터 셋을 변환할 필요가 없다.

THANK YOU

