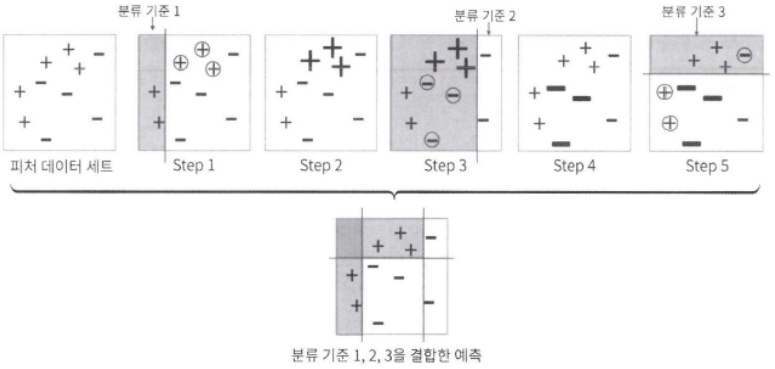
분류 Part2

5. GBM (Gradient Boosting Machine)

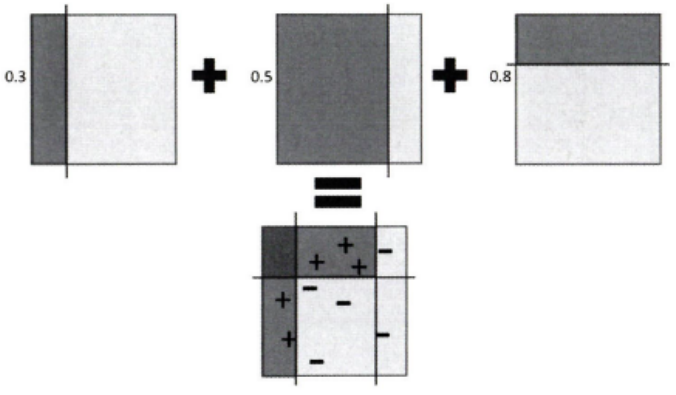
GBM 개요 및 실습

- **부스팅** 알고리즘
  - 약한 학습기 차례로 학습/예측 → 잘못된 데이터에 가중치 부여함

1. **에이다 부스트 (AdaBoost)**
   a. 약한 학습기로 데이터 분류 (학습/예측)
   b. 오류 데이터에 가중치 값 부여
      ⇒ 다음 학습기가 더 잘 분류하게 됨
   c. 약한 학습기를 모두 결합해서 결과를 예측함



- 개별 약한 학습기에 가중치 부여



2. **그래디언트 부스트(GBM)**
   - 경사 하강법으로 가중치 업데이트함
     - 경사 하강법: 오류식을 최소화하는 방향성으로 가중치 업데이트
       (y: 실제 결괏값, F(x): 피처에 기반한 예측 함수, 오류식: h(x)=y-F(x))

- GradientBoostingClassifier: 분류 클래스
- 랜덤 포레스트보다 예측 성능 뛰어남

  but 느림 (⇒ XGBoost), 하이퍼 파라미터 튜닝 노력 필요함

GBM 하이퍼 파라미터 및 튜닝

이름	설명
loss	- 경사 하강법 비용 함수 - deviance(디폴트)
learning_rate	- 학습률: 약한 학습기가 오류값 보정할 때 적용하는 계수 - 0~1 (디폴트: 0.1) - 학습률 작을 때: 최소 오류 값 찾을 수 있음 (예측 성능 good) but 수행 시간 길어짐, 최소 오류 값 찾지 못할 수도 있음 - 학습률 클 때: 수행 시간 짧아짐 but 최소 오류 값 지나칠 수 있음 (예측 성능 bad)
n_estimators	- weak learner 개수 - 개수 많을 수록 성능 좋아질 수 있음 but 시간 오래 걸림
subsample	- 학습에 사용하는 데이터의 샘플링 비율 - 1(디폴트): 전체 데이터, 0.5: 학습 데이터의 50% - subsample<1 → 과적합 방지

6. XGBoost

XGBoost 개요

- 장점

항목	설명
1. 뛰어난 성능	
2. GBM에 비해 빠른 수행 시간	- 병렬로 수행함 ◦ GBM: 순차적인 진행 → 느림
3. 과적합 규제	- 자체 과적합 규제 기능 ◦ GBM: 과적합 규제 기능 없음
4. 나무 가지치기	긍정 이득이 없는 분할을 가지치기 함 → 분할 수를 줄임
5. 자체 내장된 교차 검증	- 교차 검증 ⇒ 최적화된 반복 수행 횟수 얻음 - 조기 중단 기능: 평가 값 최적화됨 → 반복 중간에 중단함
6. 결손값 자체 처리	

- 파이썬 래퍼 XGBoost 모듈
  - 독자적인 XGBoost 프레임워크 기반
- 사이킷런 래퍼 XGBoost 모듈
  - 사이킷런과 연동

XGBoost 설치하기

```
conda install -c anaconda py-xgboost
```

```
import xgboost as xgb
from xgboost import XGBClassifier
```

파이썬 래퍼 XGBoost 하이퍼 파라미터

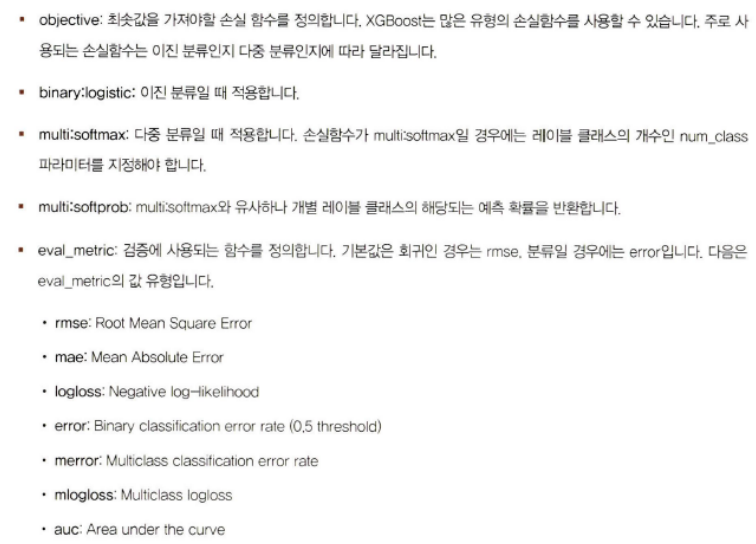
- 하이퍼 파라미터 유형
  1. 일반 파라미터

이름	설명
booster	gbtree(tree based model) (디폴트) gblinear(linear model)
slient	- 0 (디폴트) - 1: 출력 메시지 나타내고 싶지 않을 때
nthread	CPU 실행 스레드 수 - (디폴트) 전체 사용

  2. 부스터 파라미터
     - 트리 최적화, 부스팅, regularization 관련

이름	설명
eta [default=0.3, alias: learning_rate]	학습률
num_boost_rounds	GBM의 n_estimators
min_child_weight [default=1]	클수록 분할 자제 ⇒ 과적합 규제
gamma [default=0, alias: min_split_loss]	트리 리프 노드 추가로 나눌지 결정할 최소 손실 감소 값
max_depth	
sub_sample	데이터 샘플링 비율 지정 ⇒ 과적합 규제
colsample_bytree	GBM의 max_feature
lambda	L2 Regularization 적용 값
alpha	L1 Regularization 적용 값
scale_pos_weight	비대칭한 클래스로 구성된 데이터 세트의 균형을 유지함

  3. 학습 태스크 파라미터
     - 학습 수행 시 객체 함수, 평가 지표 설정



- 과적합 문제 해결 tip
  - eta 값 낮추기
    - num_round (=n_estimators) 값 높여줘야 함
  - max_depth 값 낮추기
  - min_child_weight 값 높이기
  - gamma 값 높이기
  - subsample, colsample_bytree 조정하기
- 조기 중단 기능
  - n_estimators 반복 횟수에 도달하지 않았지만 예측 오류가 더 이상 개선되지 않음 → 중지!

    ⇒ 시간 단축됨
    - ex. 조기 중단 파라미터 = 50
      : 50회 반복하는 동안 오류 감소하지 않음 → 종료

사이킷런 래퍼 XGBoost 개요 및 적용

- XGBClassifier: 분류

  XGBRegressor: 회귀
- 하이퍼 파라미터명

사이킷런 래퍼 XGBoost 이름	파이썬 래퍼 XGBoost 이름
learning_rate	eta
subsample	sub_sample
reg_lambda	lambda
reg_alpha	alpha
n_estimators	num_boost_round

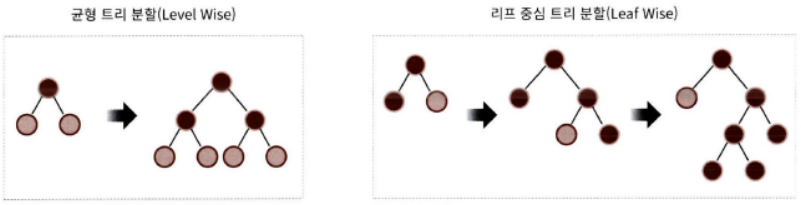
7. LightGBM

- 특징

장점	단점
1. XGBoost보다 학습 시간이 짧음	1. if 적은 데이터 사용 → 과적합 발생하기 쉬움
2. 메모리 사용량 적음	
3. 카테고리형 피처 자동변환, 최적 분할 - 원-핫 인코딩 사용하지 않고 노드 분할 수행 가능함	

- **리프 중심 트리 분할** 사용함
  - cf. 대부분의 트리 기반 알고리즘은 균형 트리 분할 방식 사용함

이름	설명	장점	단점
리프 중심 트리 분할	트리의 균형을 맞추지 않고 최대 손실 값을 갖는 리프 노드를 계속 분할함	예측 오류 손실 최소화함	- 깊이가 깊어짐 - 비대칭적인 트리 생성됨
균형 트리 분할	균형 잡힌 트리 유지하면서 분할함	트리의 깊이 최소화 됨 ⇒ 과적합에 강함	시간이 걸림



LightGBM 설치

```
conda install -c conda-forge lightgbm
```

LightGBM 하이퍼 파라미터

이름	설명
num_iterations	반복 수행하는 트리 개수
learning_rate	- 학습률: 약한 학습기가 오류값 보정할 때 적용하는 계수 - 0~1 (디폴트: -1) - 학습률 작을 때: 최소 오류 값 찾을 수 있음 (예측 성능 good) but 수행 시간 길어짐, 최소 오류 값 찾지 못할 수도 있음 - 학습률 클 때: 수행 시간 짧아짐 but 최소 오류 값 지나칠 수 있음 (예측 성능 bad)
max_depth	- <0: 깊이에 제한이 없음
min_data_in_leaf	= min_samples_leaf - LightGBMClassifier: min_child_samples - 리프 노드 되기 위해 필요한 최소 레코드 수
num_leaves	하나의 트리가 갖는 최대 리프 개수
boosting	부스팅의 트리 생성하는 알고리즘을 기술함
bagging_fraction	- 데이터 샘플링 비율 지정 ⇒ 과적합 방지
feature_fraction	- 개별 트리 학습할 때 무작위로 선택하는 피처 비율 - GBM: max_feature

이름	설명
	- XGBClassifier: colsample_bytree
lambda_l2	L2 Regularization 제어 값
lambda_l1	L1 Regularization 제어 값
objective	최솟값을 가져야 할 손실함수

하이퍼 파라미터 튜닝 방안

- 모델 복잡도 감소 tip
  - num_leaves 줄이기
    - num_leaves 높임
      → 정확도 높아짐
      → but 트리 깊이 깊어짐, 복잡도 커짐
  - min_data_in_leaf (=min_child_samples) 큰 값으로 설정함
  - max_depth로 깊이 제한함
  - learning_rate 작게, n_estimators 크게 설정함

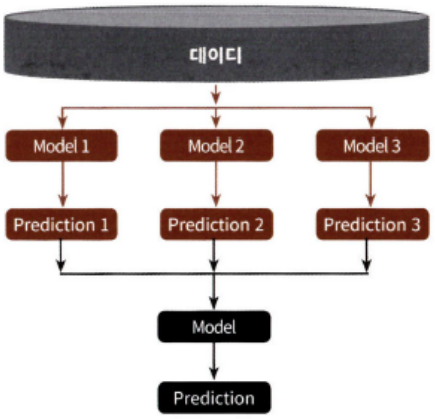
파이썬 래퍼/사이킷런 래퍼 LightGBM, 사이킷런 래퍼 XGBoost 하이퍼 파라미터 비교

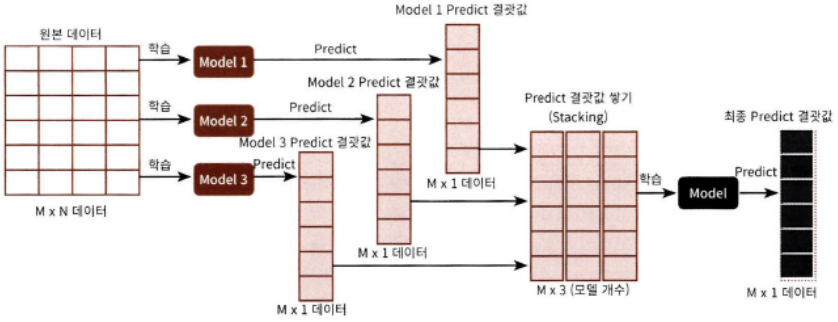
유형	파이썬 래퍼 LightGBM	사이킷런 래퍼 LightGBM	사이킷런 래퍼 XGBoost
파라미터명	num_iterations	n_estimators	n_estimators
	learning_rate	learning_rate	learning_rate
	max_depth	max_depth	max_depth
	min_data_in_leaf	min_child_samples	N/A
	bagging_fraction	subsample	subsample
	feature_fraction	colsample_bytree	colsample_bytree
	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

10. 스태킹 앙상블

- **메타 모델**: 개별 모델의 예측된 데이터 세트를 기반으로 다시 학습, 예측하는 방식
- 수행 절차
  1. **개별적인 기반 모델**로 예측한 결과 값으로 학습/테스트용 데이터 세트를 생성함
  2. 각 모델에서 생성된 학습/테스트용 데이터를 스태킹 형태로 합침 (메타 데이터 세트)
  3. ML 알고리즘으로 최종 학습 수행함 (**최종 메타 모델**)
  4. 테스트용 데이터 (메타 데이터 세트) 기반으로 최종 예측 수행함
  5. 원본 테스트 데이터 기반으로 평가함

- 성능 비슷한, 많은 개별 모델이 필요함 → 성능 높임

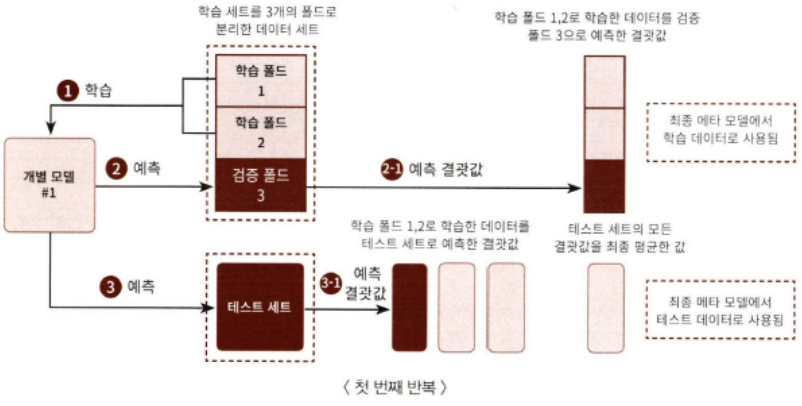


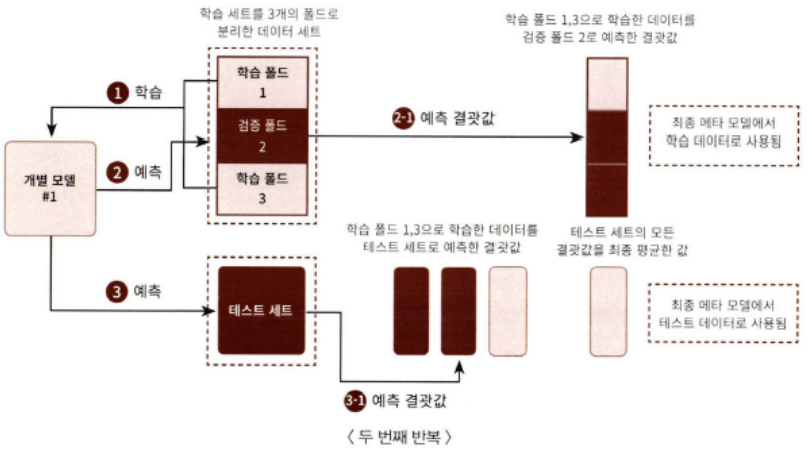


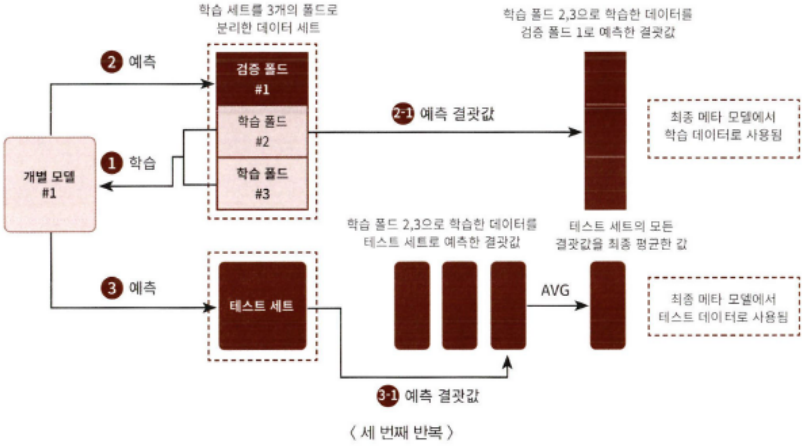
기본 스태킹 모델

CV 세트 기반의 스태킹

- 최종 메타 모델을 위한 데이터 세트 만들 때 **교차 검증 기반**으로 예측함 → 과적합 개선!
- 수행 절차







1. 학습용 데이터를 N개의 폴드로 나눔

   → (N-1)개 학습용 데이터 폴드, 1개 검증용 데이터 폴드

2. 학습용 데이터 폴드로 개별 모델 학습시킴

   a. 검증 폴드 1개로 예측하고 결과 저장함

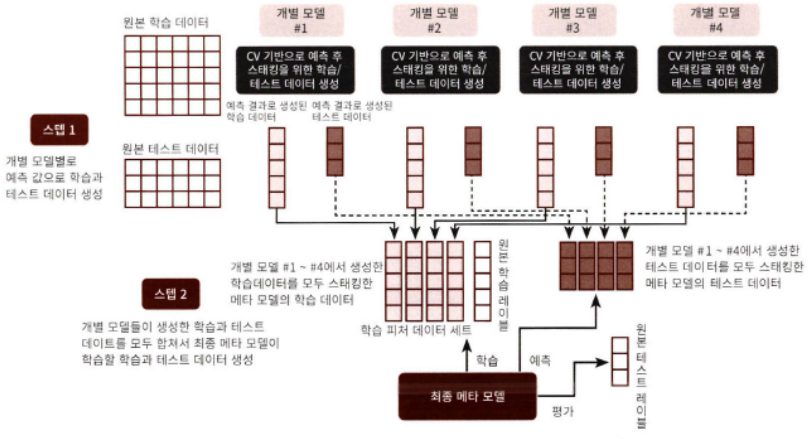
   b. 1~3을 N번 반복함

3. (N-1)개 폴드로 학습한 모델로 원본 테스트 데이터 예측함

   a. N번 반복함

   b. 최종 결괏값 = 예측값의 평균

      → 메타 모델의 테스트 데이터



4. 학습/테스트용 데이터 모두 합침
5. 최종 학습 데이터와 원본 데이터의 레이블 데이터 합침
6. 메타 모델 학습함
    a. 최종 테스트 데이터로 예측함
    b. 최종 예측 결과를 원본 테스트 데이터의 레이블과 비교함 (평가)

[파머완 2판] 베이지안 최적화 기반 HyperOpt로 하이퍼 파라미터 튜닝

- Grid Search 방식을 이용한 하이퍼 파라미터 튜닝
    - 하이퍼 파라미터 수 많음 → 최적화 시간 오래 걸림

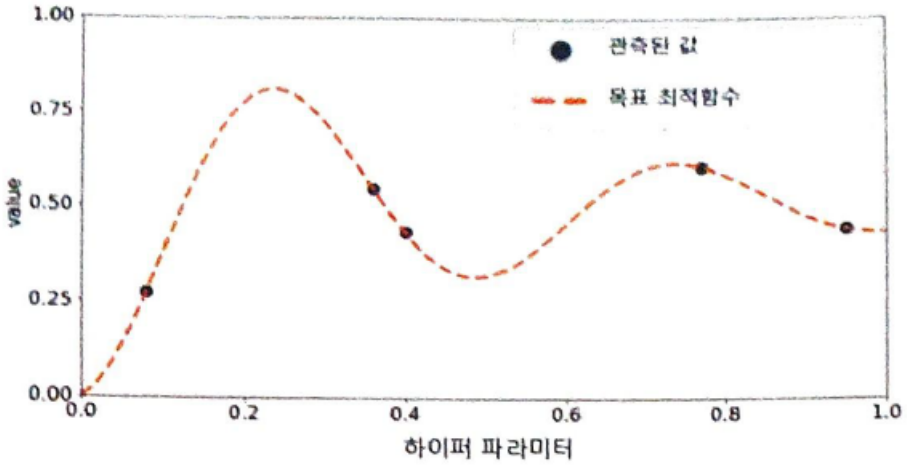
      ⇒ **베이지안 최적화 기법**

베이지안 최적화 개요

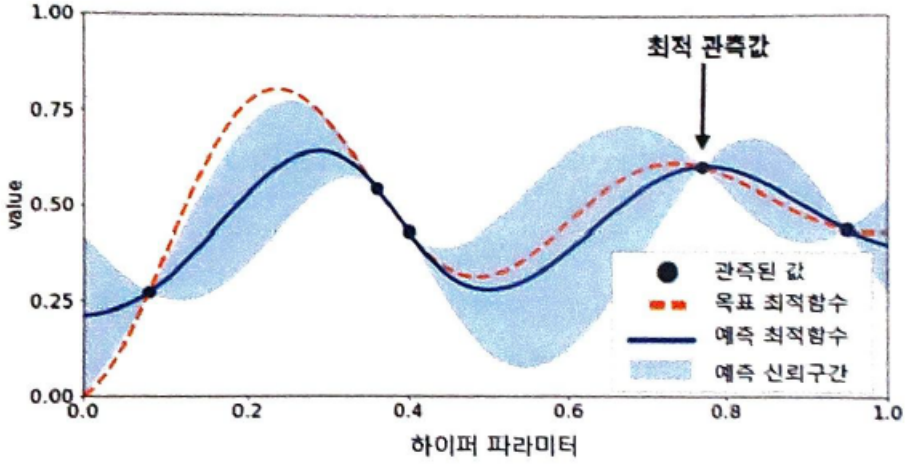
- 목적 함수 식을 알 수 없을 때 최적 입력값을 최소한의 시도로 찾아주는 방식
    - 목적 함수: 어떤 목적 (ex. 최대, 최솟값) 위해 사용하는 함수
- 새로운 데이터 입력 받음 → 최적 함수 예측하는 사후 모델 개선 → 최적 함수 모델 생성
    - 베이지안 확률 기반 최적화 기법
        - 베이지안 확률: 새로운 사건의 관측, 데이터 기반으로 사후 확률 개선함

구성 요소	설명
대체 모델(Surrogate Model)	획득 함수로부터 입력값 추천 받음 → 최적 함수 모델 개선
획득 함수(Acquisition Function)	개선된 모델 기반으로 최적 입력값 계산함

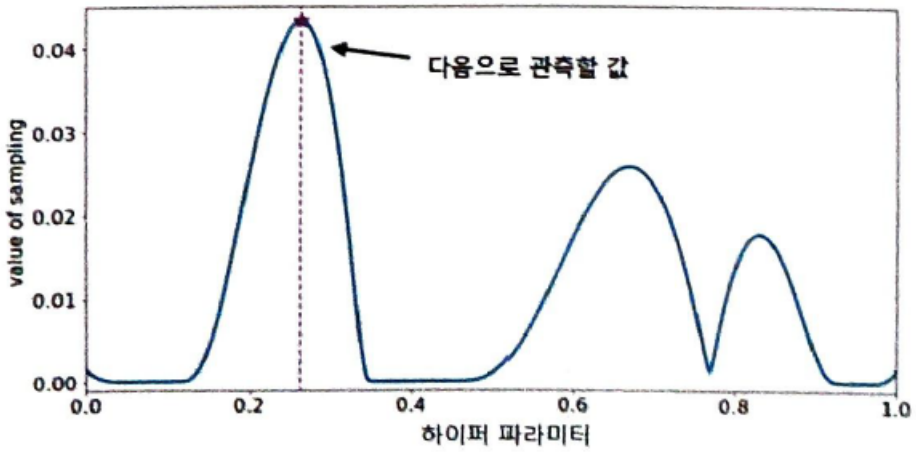
- 최적화 단계
    1. 랜덤 하이퍼 파라미터를 입력하여 성능 결과 관측함



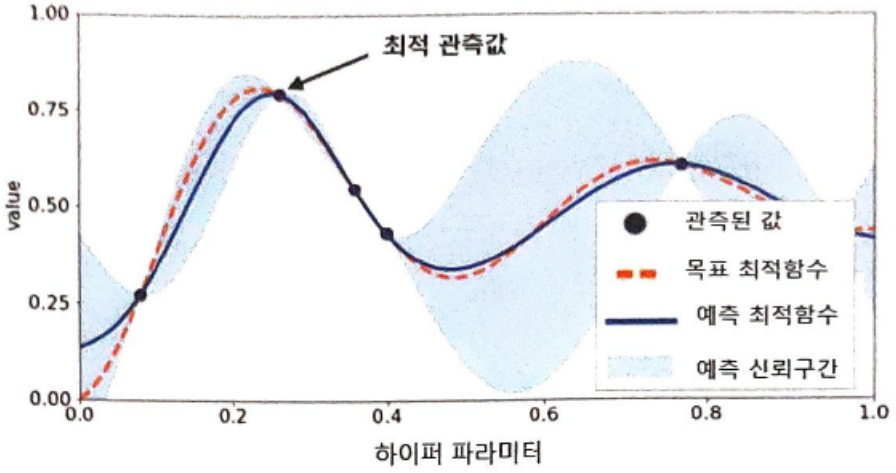
2. 관측값 기반으로 최적 함수 추정함
    - 파란색 실선: 대체 모델이 추정한 최적 함수
    - 옅은 파란색: 예측된 함수의 신뢰 구간, 추정된 함수의 결괏값 오류 편차 (=추정 함수의 불확실성)
    - 최적 관측값: y축(value)에서 가장 높은 값 가질 때의 하이퍼 파라미터



3. 획득 함수가 추정된 최적 함수 기반으로 하이퍼 파라미터 값을 계산함
    - 이전 최적 관측값보다 더 큰 최댓값을 가질 가능성이 있는 지점 찾음 → 그 지점을 대체 모델에 전달함
      (왜 최댓값을 갖는 지점 찾아야 함?)



4. 전달 받은 하이퍼 파라미터 수행해서 관측한 값 기반으로 모델 개선함 → 최적 함수 다시 예측 추정함



HyperOpt 사용하기

- HyperOpt 사용 로직
  1. 입력 변수명, 입력값의 검색 공간 설정하기
     - 검색 공간 제공하는 함수
       - label: 입력 변수명
       - low: 최솟값, high: 최댓값, q: 간격

함수	설명
hp.quniform(label, low, high, q)	입력값 변수 검색 공간을 최솟값에서 최댓값까지 q 간격으로 설정
hp.uniform(label, low, high)	최솟값에서 최댓값까지 정규 분포 형태의 검색 공간 설정
hp.randint(label, upper)	0부터 최댓값(upper)까지 random 정숫값으로 검색 공간 설정
hp.loguniform(label, low, high)	exp(uniform(low,high)) 반환 - log(반환 값)이 정규 분포 형태인 검색 공간 설정
hp..choice(label, options)	검색 값이 문자열 or 문자열+숫자값

  2. 목적 함수 설정하기

- 인자: 변숫값, 검색 공간 갖는 딕셔너리
- 반환값: 숫자형 or dictionary
    - if dictionary → {'loss': retval, 'status':STATUS_OK}

3. 목적 함수의 반환 최솟값 갖는 최적 입력값 유추하기
    - **fmin**(objective, space, algo, max_evals, trials) 함수

이름	설명
fn	목적 함수
space	(= search_space) 검색 공간 딕셔너리
algo	베이지안 최적화 알고리즘
max_evals	입력값 시도 횟수
trials	최적 입력값 찾기 위해 시도한 입력값 및 목적 함수 반환값 결과를 저장함
rstate	random seed

    - **Trials** 객체 중요 속성
        1. results
            - 함수 반복 수행 시, 반환되는 반환값
            - result- list 형태
              result 개별 원소- dictionary 형태: {'loss': retval, 'status':STATUS_OK}
        2. vals
            - 입력되는 입력 변숫값
            - dictionary 형태: {'입력 변수명': 개별 수행 시마다 입력된 값의 리스트}

HyperOpt 이용한 XGBoost 하이퍼 파라미터 최적화

- 주의점
    - 특정 하이퍼 파라미터들은 정숫값만 입력 받음 ⇒ 입력 시 형변환!
    - 목적 함수는 최솟값 반환하도록 최적화됨 ⇒ 값이 클수록 좋은 성능 지표일 경우 (-1) 곱하기!