

# 파머완 4장 Part2

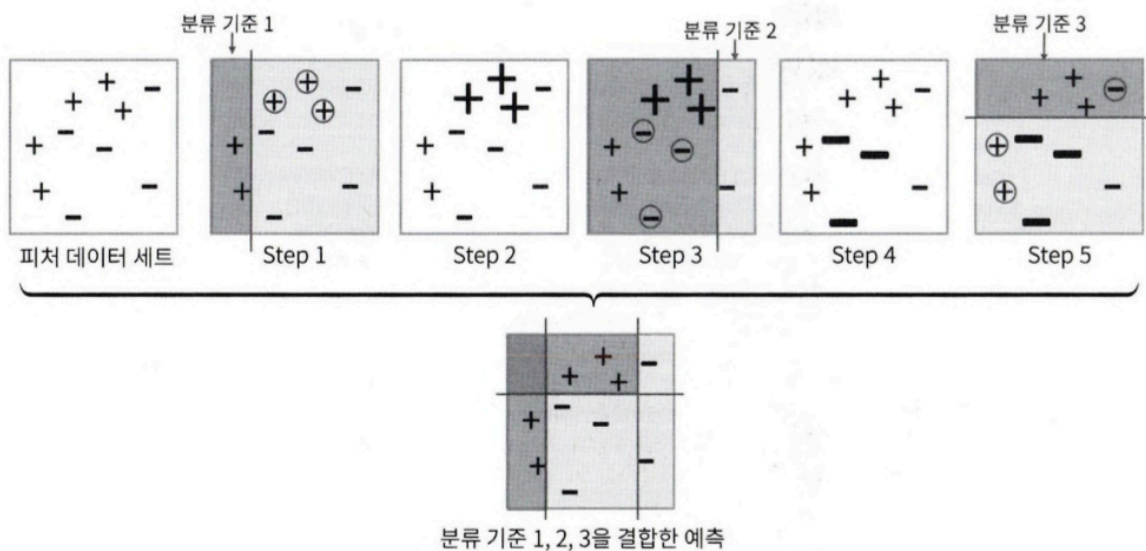
## 05. GBM(Gradient Boosting Machine)

Boosting 알고리즘 : 약한 학습기를 순차적으로 학습,

예측하면서 잘못 예측한 데이터에 가중치 부여, 오류 개선

### 종류

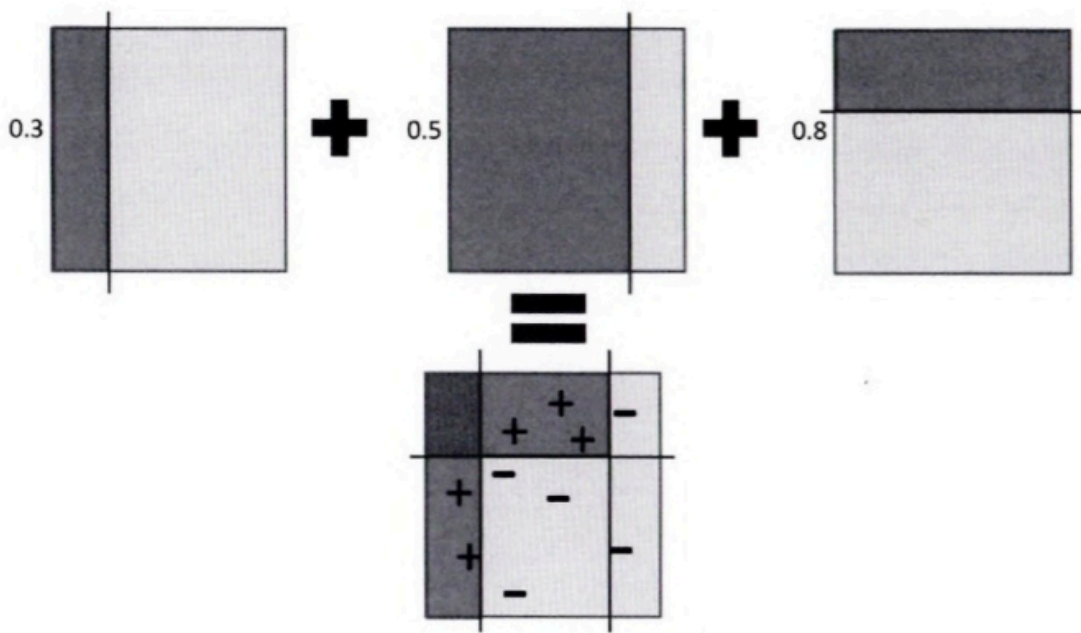
#### 1) AdaBoost



+, - 로 가중치 데이터 분류

- step1, step2 > 1번 weak learner가 분류기준 1 학습, 오류에 대해 가중치 부여
- step3, step4 > 2번 weak learner가 분류기준 2 학습, 오류에 대해 가중치 부여
- step5 > 3번 weak learner가 분류기준 3 학습, 오류에 대해 가중치 부여
- weak learner 1번 + 2번 + 3번 = 최종  
✓ weak learner들보다 향상된 성능

#### 2) Gradient Boost




- AdaBoost와의 차이점 : 가중치 업데이트에 경사하강법(Gradient Descent)를 이용.
- 오류 = ( 실제값 - 예측값)  
 $h(x) = y - F(x)$  {  $x$ : 피쳐,  $F(x)$ : 예측함수,  $y$ : 클래스(정답) }
- 과적합에 강함, 수행시간이 오래 걸림.

## GBM vs RandomForest

→ GBM이 예측 성능이 조금 더 좋지만 수행 시간이 오래걸림  
 사이킷런의 GBM은 병렬 처리 지원X,

## GBM 하이퍼파라미터

1. loss : loss function 지정  
 [ default = 'deviance' ]
2. learning\_rate :  
 학습률, weak learner 오류값 보정 계수  
 0~1 사이값, [ default = 0.1 ]  
 너무 작으면 업데이트값 ⬇️ 최소 오류값 찾기 수월, 성능 up  
 너무 크면 최소 오류값 못 찾을 수 있음 / 빠른 수행
3. n\_estimators :  
 weak learner의 개수, [ default = 100 ]

일정 수준까지는 많아질수록 예측 성능 , 시간 오래걸림  
learning rate를 낮추고 n\_estimators 높이면 성능이 좋아지는 경향이 있지만 한계점이 있고 수행시간 대비 성능이 크게 좋아지지는 않음.

- 4. subsample :  
weak learner가 학습에 사용하는 데이터 샘플링 비율  
[ default = 1( 100%, 전체 학습 데이터 기반) ]  
overfitting 피하기 위해서는  $< 1$


## 06. XGBoost

트리 기반 앙상블 학습 알고리즘 중 각광받는 알고리즘

GBM 기반 & 보완

핵심 라이브러리 - C/C++ , 파이썬 패키지 제공

xgboost 사이킷런 프레임워크 기반X

→ predict(), fit()/ 유틸리티함수 

✓ sklearn과 연동되는 wrapper class,  
XGBClassifier, XGBRegressor 사용

### 장점

1. 분류, 회귀에서 뛰어난 예측 성능
2. GBM 대비 빠른 수행시간 :  
병렬 수행 등 다양한 기능 / 절대적으로 빠르진 않음
3. 과적합 규제 :  
GBM에는 없음
4. Tree pruning(나무 가지치기) :  
GBM - 분할시 부정손실 발생하면 분할 더 X,  
지나치게 많은 분할 발생 가능  
XGBoost - 긍정 손실 발생하지 않는 분할 가지치기,  
분할 수 줄임
5. 교차 검증 자체 내장 :  
반복시 자체적으로 train set/test set 교차검증 수행,  
평가값이 최적화되면 반복을 중간에 멈출수있음

6. 결손값 자체 처리 가능

7. 시각화 가능

## 하이퍼 파라미터

뛰어난 알고리즘일수록 튜닝 ⬇️

### 1. 일반 파라미터 : default 변동 거의 X

- **booster** - 'gbtree' / 'gblinear'
- **silent** - [default = 0], 출력 메시지 끄기 = 1
- **nthread** - CPU 실행 스레드 개수 조정  
default = 다실행  
멀티코어/스레드 CPU 시스템에서 일부만 사용해  
ML application 구동 시 조정

### 2. 부스터 파라미터 : 트리 최적화, 부스팅, regularization 관련

- **eta** - [default = 0.3, alias: learning\_rate]  
( sklearn wrapper 기반 xgboost learning\_rate 대체, default 0.1 )
- **num\_boost\_rounds** - GBM n\_estimators
- **min\_child\_weight** - 트리에서 가지를 분할할지 결정하기 위해 필요한 데이터들의  $\Sigma$  (weight)  
클수록 분할 자제(overfitting 조절)
- **gamma** - [default = 0, alias: min\_split\_loss ]  
트리의 리프노드 추가 분할을 결정하는 최소손실 감소 값  
감소한 loss값 > gamma값 : 리프노드 분리,  
클수록 분할 자제
- **max\_depth** - [default = 6, 0(무제한) / 보통 3~ 10 ]  
너무 높으면 특정 피처에만 특화된 조건 생성, overfitting 위험 ⬆️
- **sub\_sample** - [default = 1 / 보통 0.5 ~ 1]  
GBM subsample
- **colsample\_bytree** - GBM max\_features  
트리 생성에 필요한 피처(칼럼) 임의로 샘플링

피쳐 너무 많으면 과적합 조정하는데 사용

- **lambda** - [default = 1, alias : reg\_lambda]  
L2 Regularization 적용값  
피쳐 많을 때 적용 검토, 클수록 overfitting ⬇
- **alpha** - [default = 1, alias = reg\_alpha]  
L1 Regularization 적용값  
피쳐 많을 때 적용 검토, 클수록 overfitting ⬇
- **scale\_pos\_weight** - [default = 1]  
클래스 분포가 비대칭적일 때 데이터셋 균형 유지

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

- **objective** - 최솟값을 가져야 할 손실 함수 ( binary / multiclass )
- **binary:logistic** - 이진 분류
- **multi:softmax** - 다중 분류  
( 손실함수로 사용시 num\_class = 레이블 클래스 수 )
- **mult:softprob** - 개별 클래스에 해당하는 예측 확률 return
- **eval\_metric** - 검증 함수 정의  
default = 'rmse'(회귀) , 'error'(분류)
  - rmse : Root Mean Square Error
  - mae : Mean Absolute Error
  - logloss : Negative log- likelihood
  - error : Binary Classification error rate (0.5 threshold)
  - merror : Multiclass classification error rate
  - mlogloss : Multiclass logloss
  - auc : Area under the curve

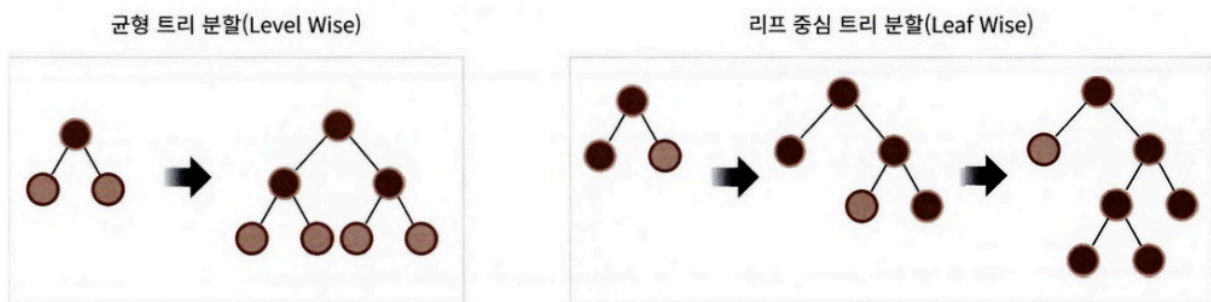
### # Overfitting 심할 때

- eta(learning\_rate) 0.01 ~ 0.1 + num\_round(n\_estimators) ⬆
- max\_depth ⬇
- min\_child\_weight ⬆

- gamma ↑
- subsample, colsample\_bytree 조정

## 07. LightGBM

※파머완에서는 LightGBM 3.3.2 버전에서 진행  
최신 버전에서는 바뀐 파라미터들 존재



일반

GBM 계열의 트리 분할 방법(Level Wise) 과 다르게 리프중심 트리분할(Leaf Wise) 방식

Level Wise - 균형 잡힌 트리 유지, 깊이 최소화, 오버피팅에 강함

Leaf Wise - 트리 균형 유지❌, 최대 손실값(max delta loss) 가지는 리프노드 분할,  
깊이 ↑, 비대칭적 트리 모양  
학습을 반복할 수록 예측 오류 손실 최소화

파이썬 패키지(래퍼) - lightgbm

```
import lightgbm as lgb
```

✅사이킷런 래퍼 - LGBM Classifier, LGBMRegressor

```
from lightgbm import LGBM
```

## # XGBoost 대비 장점

1. 더 빠른 학습과 예측 수행 시간 : 높은 병렬도,
2. 적은 메모리 사용량
3. 카테고리형 피처 자동 변환, 최적 분할 - 원 핫 인코딩 없이도 가능

## 하이퍼 파라미터

### 1. 주요 하이퍼 파라미터

- `num_iterations` - [ default = 100 , Sklearn Wrapper : `n_estimators` ]  
반복 수행하려는 트리 개수  
클수록 예측 성능 ⬆, 너무 크면 overfitting으로 성능 ⬇
- `learning_rate` - [ default = 0.1 ]  
작을수록 예측 성능 ⬆, 너무 작으면 성능 ⬇, 학습 시간 ⬆
- `max_depth` - [ default = -1 ]  
0보다 작으면 무제한, LightGBM은 leaf wise 기반, 더 깊음.
- `min_data_in_leaf` - [ default = 20 , Sklearn Wrapper : `min_child_samples` ]  
⇒ 결정트리 `min_samples_leaf`  
리프 노드가 되기 위해서 최소한으로 필요한 레코드 수  
overfitting 제어
- `num_leaves` - [ default = 31 ]  
하나의 트리가 가질 수 있는 최대 리프 개수
- `boosting` - [ default = `gbdt` ]
  - `gbdt` : 일반적인 GraDient BoosTing 결정트리
  - `rf` : RandomForest
- `bagging_fraction` - [ default = 1.0 , Sklearn Wrapper : `subsample` ]  
데이터 샘플링 비율 지정  
트리 커져서 overfitting되는 것 제어
- `feature_fraction` - [ default = 1.0, Sklearn Wrapper : `colsample_bytree` ]  
GBM `max_features`  
개별 트리를 학습할 때마다 무작위로 선택하는 피처 비율

- `lambda_l2` - [ default = 0.0, Sklearn Wrapper : `reg_lambda` ]  
L2 regulation 제어  
피처 개수 많을 때 적용 검토  
값이 클수록 overfitting ⬇️
- `lambda_l1` - [ default = 0.0, Sklearn Wrapper : `reg_alpha` ]  
L1 regulation 제어  
피처 개수 많을 때 적용 검토  
값이 클수록 overfitting ⬇️

## 2. Learning Task 파라미터

- `objective` : XGBoost objective  
최솟값 가질 loss function ( 회귀, 다중클래스/이진 분류 )

## 하이퍼 파라미터 튜닝 방안

- 부스팅 계열 튜닝 : `learning_rate` ⬇️, `n_estimators` ⬆️ ( 적당히 )
- `num_leaves` ⬆️ : 정확도 ⬆️, 트리 깊이 ⬆️, overfitting 영향도 ⬆️
- `min_data_in_leaf` ( `min_child_samples` ) ⬆️ : 트리 깊이 ⬆️❌
- `max_depth` 제한
- regularization 적용 : `reg_lambda`, `reg_alpha`
- 피처 개수/ 데이터 샘플링 레코드 수 ⬇️ : `colsample_bytree`, `subsample`

## 파이썬 래퍼 LightGBM vs 사이킷런 XGBoost/LightGBM 파라미터



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

## 08. 베이지안 최적화 기반의 HyperOpt를 이용한 하이퍼 파라미터 튜닝

대용량 학습 데이터 → GridSearch는 최적화 시간이 너무 오래 걸림

### # 베이지안 최적화

베이지안 확률에 기반을 두는 최적화 기법

베이지안 확률 - 새로운 사건의 관측/샘플 데이터 ⇒ 사후 확률 개선

베이지안 최적화 - 새로운 데이터 ⇒ 최적 함수 예측 사후 모델 개선

$$f(x,y) = 2x - 3y$$

$f(x,y)$  최소/최대로 하는  $(x,y)$  찾기

→  $x$ 는 클수록,  $y$ 는 0일 때 최대

#### 1. 대체 모델(Surrogate Model)

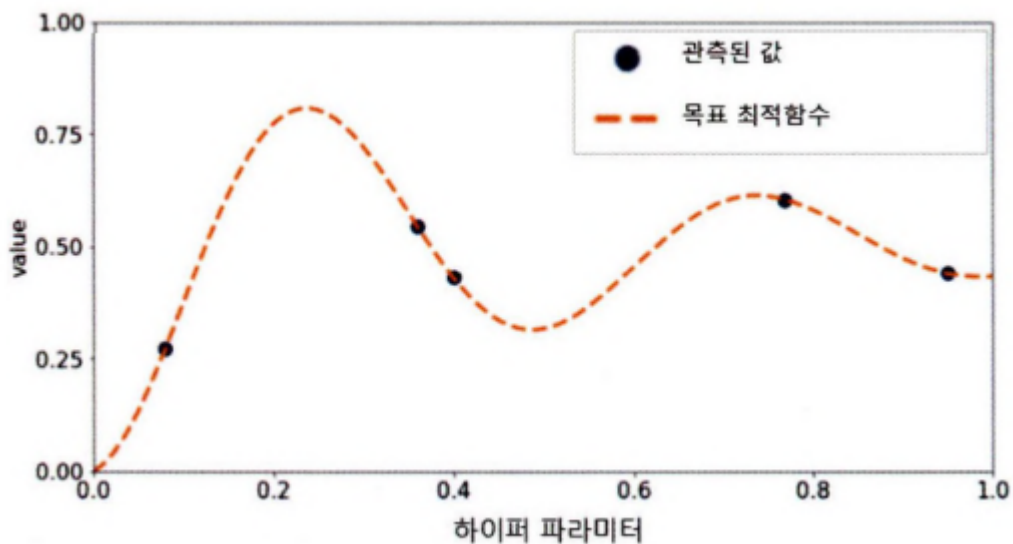
: 획득 함수 → 최적 함수 예측 입력값 → 최적함수 모델 개선

- 가우시안 프로세스(Gaussian Process, 일반적), 트리 파르젠 Estimator(TPE)

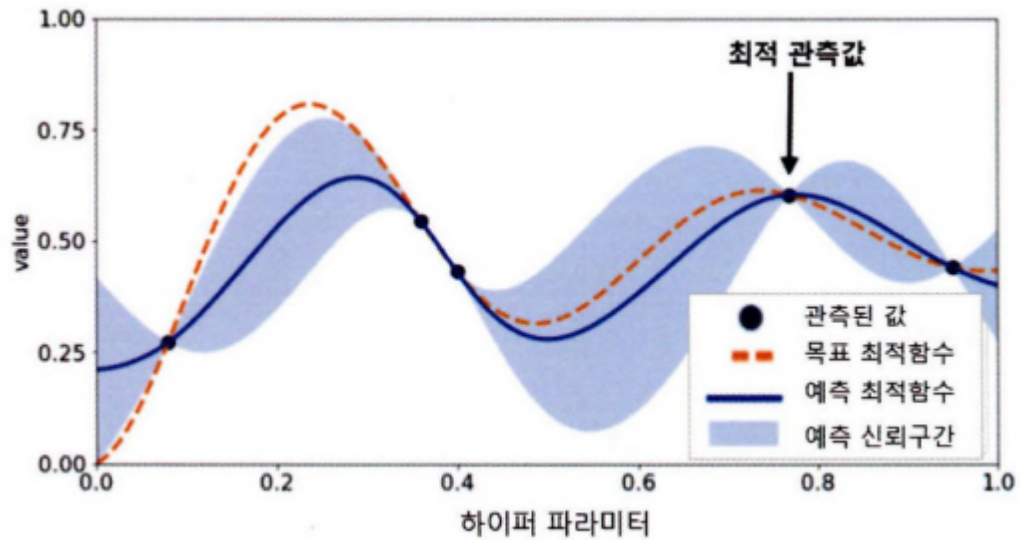
## 2. 획득 함수(Acquisition Function) : 개선된 대체 모델 → 최저 입력값 벅계산

### 베이지안 최적화 단계

- 1 랜덤하게 하이퍼 파라미터 샘플링  
검은색 원 = 특정 하이퍼 파라미터 입력시 성능 지표 결과값  
주황색 사선 = goal

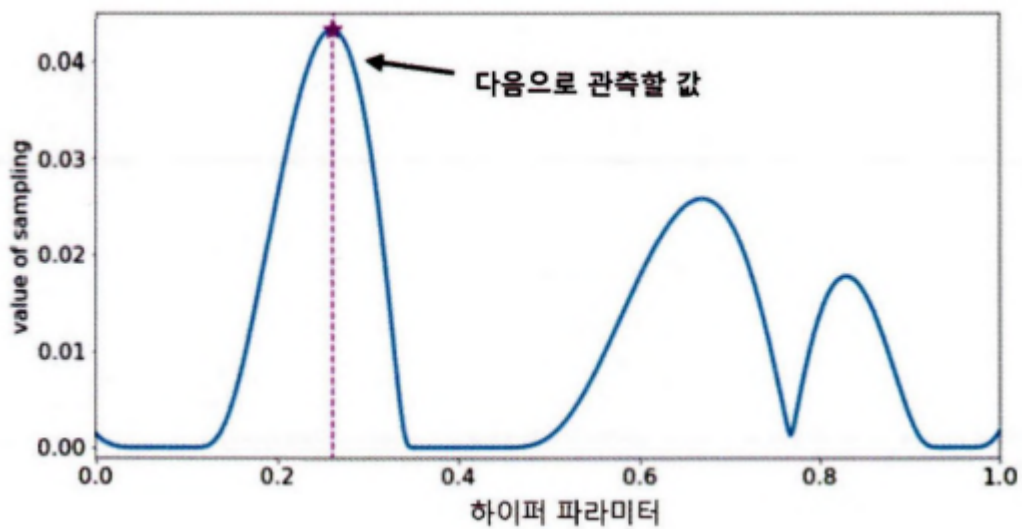


- 2 파란색 = **대체 모델(Surrogate Model)**이 추정한 최적 함수  
파란색 영역 : 예측된 함수의 신뢰구간/ 오류 편차 / 불확실성  
최적의 관측값 : y축 value가 가장 높은 값을 가질 때의 하이퍼 파라미터



**3 획득함수(Acquisition Function)**이 다음으로 관측할 하이퍼 파라미터 값 계산  
이전 최적 관측값보다 더 큰 최댓값을 가질 가능성이 높은 지점(하이퍼 파라미터  
값) 찾기

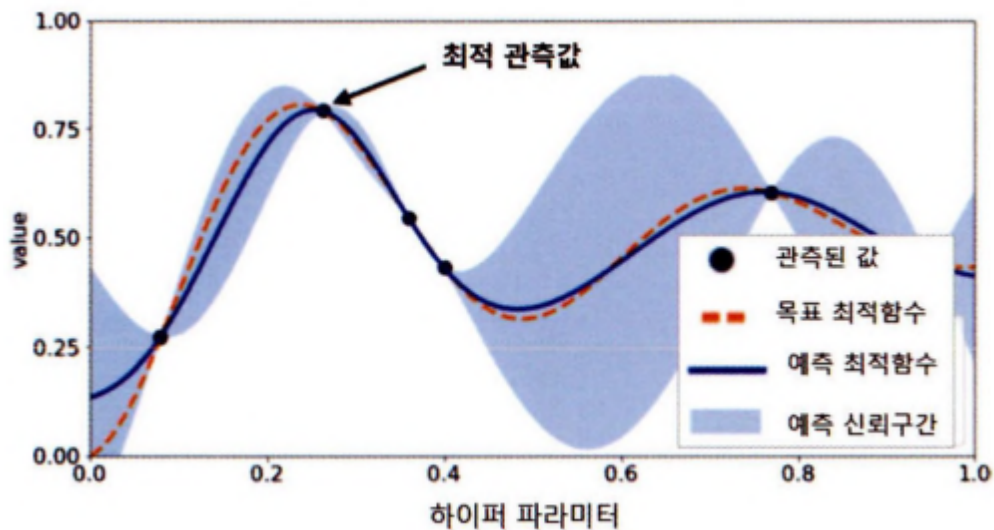
→ 그 값을 **대체 모델**에 전달



#### 4 대체 모델 갱신, 다시 최적 함수 예측 추정

⇒ 3 ⇒ 4 ⇒ 3 ⇒ ....

특정횟수만큼 반복 : 불확실성 개선



## HyperOpt 사용하기

### 로직

1. 입력 변수명, 입력값의 검색공간 (Search Space) 설정
2. 목적함수 (Objective Function)
3. 목적함수의 반환 최솟값을 가지는 최적 입력값 유추

### ✓ Search Space



딕셔너리형으로 저장

{ '입력변수 x' : [입력변수의 검색 공간] }

## 사용 함수들

- `hp.quniform`(입력변수의 검색공간, 최솟값, 최댓값, 간격)
- `hp.uniform`(입력변수의 검색공간, 최솟값, 최댓값) - random한 정수값, 정규분포
- `hp.randint`(입력변수의 검색공간, 최댓값) - random한 정수값
- `hp.loguniform`(입력변수의 검색공간, 최솟값, 최댓값) - `exp(uniform(최소, 최대))` 반환,  
그 로그변환값 정규분포
- `hp.choice`(label, options) : 검색 값이 문자열 또는 문자열과 숫자값이 섞여 있을 경우 설정  
option은 튜플/리스트, 엔트로피 지수/지니 지수 등 설정 가능

## ✓ Objective Function



input : 변수값, 검색 공간을 가지는 딕셔너리(Search space)

return : 특정 값 > 최솟값 반환하게 최적화 (ex : \* (-1) )

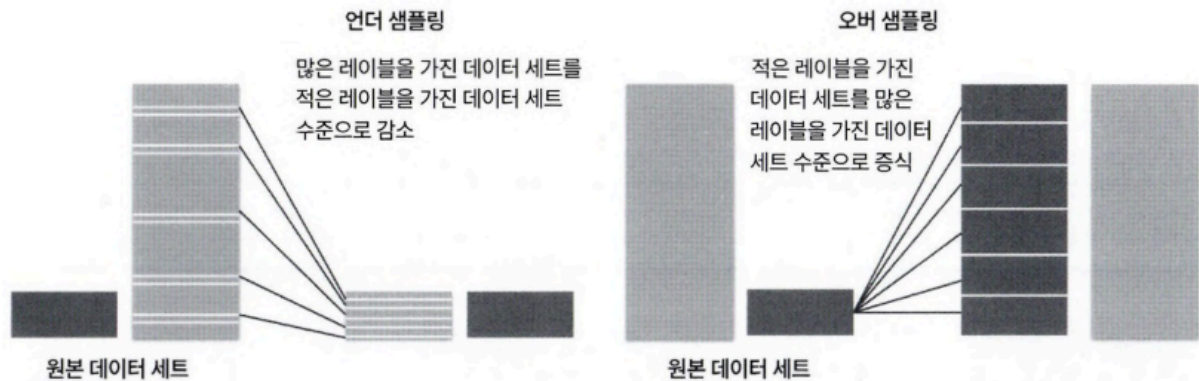
**`fmin( objective, space, algo, max_evals, trials )`**

### 주요 인자들 ( 모두 실수형, 정수형 변환 필요할 수 있음 )

- `fn`
- `space` - `search_space` 같은 검색 공간 딕셔너리
- `algo` - 베이지안 최적화 적용 알고리즘, default = `tpe.suggest`
- `max_evals` - 최적 입력값 찾기 위한 입력값 시도 횟수
- `trials` - 최적 입력값을 찾기 위해 시도한 입력값 / 해당 입력값의 목적 함수 반환값 저장  
← `Trials( )`  
`.vals` = 딕셔너리, {'입력변수명': 개별 수행시마다 입력된 값 리스트}
- `rstate` - `fmin()` 수행시 쓰는 랜덤 시드값

## 10. 분류 실습 - 캐글 산탄데르 고객 만족 예측

## 언더 샘플링/ 오버 샘플링



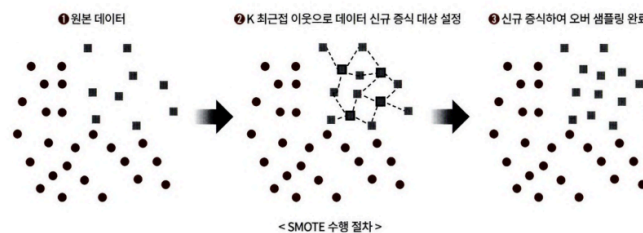
### 언더 샘플링

- 많은 데이터를 적은 데이터셋 기준으로 감소
- 100/10000 → 100/100
- 과도하게 정상레이블에 맞춰 학습 **✗**
- 정상 레이블 제대로 학습 힘들 수 있음

### 오버 샘플링

- 적은 데이터셋을 증식
- 원본 데이터의 피쳐값 약간 변경 **(SMOTE)**
- 데이터 단순 증식 시 overfitting → 무의미

## SMOTE( Synthetic Minority Over-sampling Technique )



- 개별 데이터들의 K 최근접 이웃을 찾음  
→ 데이터와 이웃들의 차를 일정 값으로 만들어 기존 데이터와 약간 차이 나는 새 데이터 생성

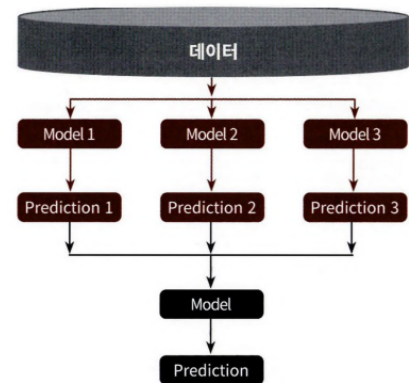
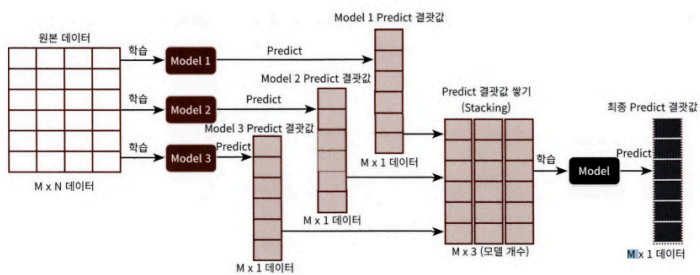
## 11. 스택킹 앙상블



## 개별적 기반 모델 + 최종 메타 모델

여러 개별 모델의 예측 데이터(y\_pred)를 스택킹 형태로 결합

→ 최종 메타 모델의 training set으로 사용



## CV 세트 기반의 스택킹



## Step1) 각 모델의 pred값으로 메타 모델을 위한 X\_test, X\_train 생성

각 모델은 X\_train, y\_train을 3개의 폴드로 나눈 후 kFold 학습 진행

→ 교차 검증된 pred값이 나옴 = 모델의 X\_train, X\_test

# kfold 교차검증

```
for folder_counter , (train_index, valid_index) in  
enumerate(kf.split(X_train)):
```

```
    print('\t 폴드 세트: ',folder_counter,' 시작 ')
```

```
    X_tr = X_train[train_index]
```

```
    y_tr = y_train[train_index]
```

```
X_te = X_train[valid_index]
```

```
    model.fit(X_tr , y_tr)
```

```
train_fold_pred[valid_index, :] = model.predict(X_te).reshape(-1,1)
```

```
test_pred[:, folder_counter] = model.predict(X_test)
```

```
# pred 평균
```

```
test_pred_mean = np.mean(test_pred, axis=1).reshape(-1,1)
```

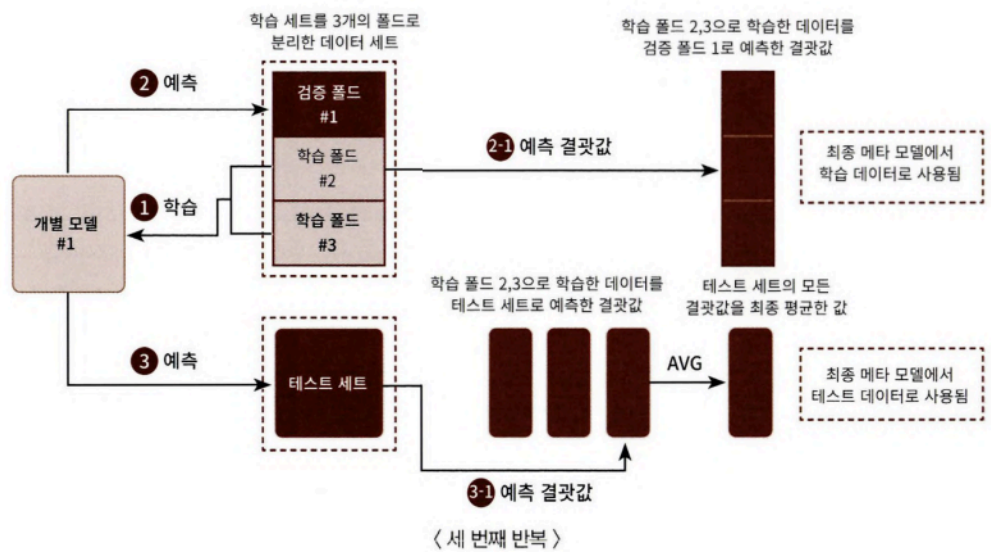
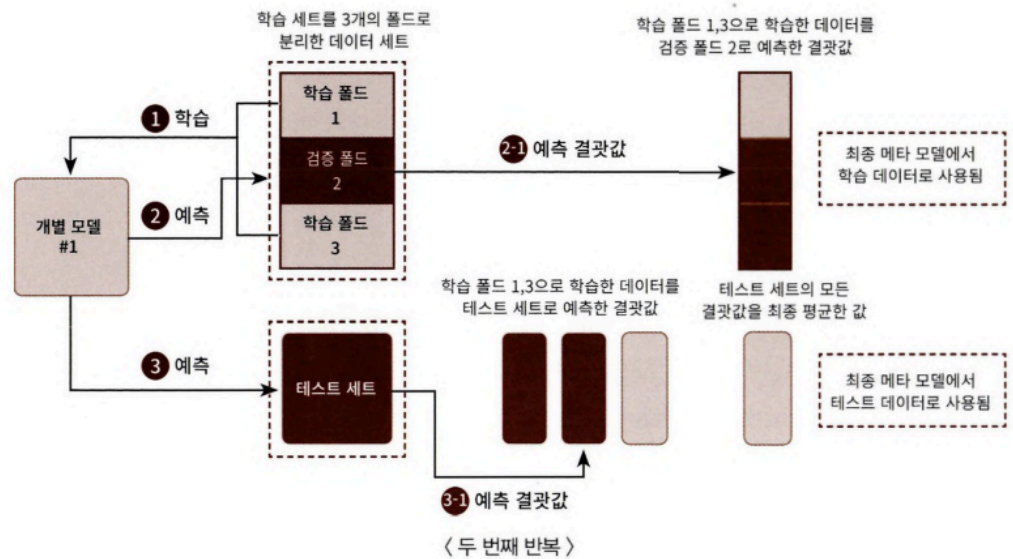
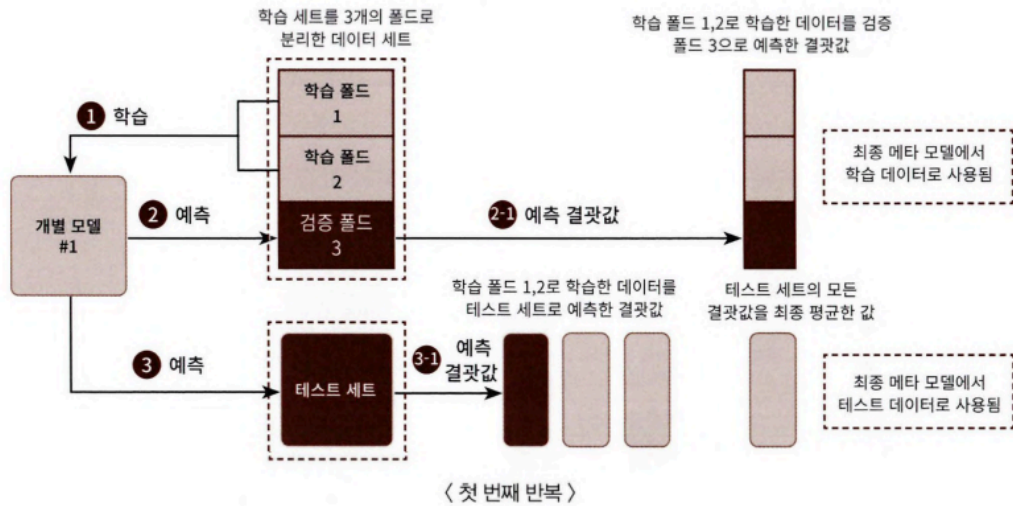
```
# train_fold_pred는 최종 메타 모델이 사용하는 학습 데이터,
```

```
# test_pred_mean은 테스트 데이터
```

```
return
```

```
train_fold_pred , test_pred_mean
```







## Step2) 각 모델이 생성한 Train Set/ Test Set를 다 합쳐서 최종 데이터셋 생성

```
메타모델_X_train = np.concatenate(knn_train, rf_train, dt_train,  
ada_train), axis=1)
```

```
메타모델_X_test = np.concatenate((knn_test, rf_test, dt_test, ada_test),  
axis=1)
```

```
lr_final.fit(Stack_final_X_train, y_train) # y_train은 원본 학습 레이블
```

```
stack_final = lr_final.predict(Stack_final_X_test)
```

```
accuracy_score(y_test, stack_final) # y_test는 원본 테스트 레이블
```

