Aichemist Session

CHAP 04 분류(2)

CONTENTS

분류

01. 결정 트리

02. 앙상블 학습

03. 랜덤 포레스트

04. GBM

4주차는 여기부터!

05. XGBoost

06. LightGBM

07. 베이지안 최적화 기반의 HyperOpt를 이용한 하이퍼 파라미터 튜닝

08. 스태킹 앙상블

05.

XGBoost

XGBoost

: GBM 기반, 단점을 보완하여 각광받는 알고리즘

분류에서 일반적으로 다른 ML보다 뛰어난 예측 성능 가짐

- 주요 장점

- 분류, 회귀 영역에서 뛰어난 예측 성능
- 병렬 수행 및 병렬 수행 등 다양한 기능으로 GBM에 비해 빠른 수행 속도
- 과적합 규제 기능
- 나무 가지치기 : 긍정 이득이 없는 분할 줄임
- 조기 중단 가능: 최적화된 반복 수행 횟수를 채웠거나 최적화가 이뤄지면 반복 중단
- 결손값 자체 처리

XGBoost 패키지

파이썬 래퍼 XGBoost	사이킷런 래퍼 XGBoost
- 사이킷런과 호환되지 않는 초기의 독자적인 XGBoost 전용 패키지 - 고유의 API와 하이퍼 파라미터 사용 - 주로 XGBoost의 고급 기능과 하이퍼파라미터 튜닝을 수행할 때 사용	- 사이킷런과 연동되어 표준 사이킷런 개발 프로세스 및 유틸리티 사용 가능 - 클래스 : XGBClassifier, XGBRegressor

파이썬 래퍼 XGBoost 하이퍼 파라미터

- 일반 파라미터 : 스레드의 개수나 silent 모드 등 선택, 일반적으로 default값 유지
- 부스터 파라미터 : 트리 최적화, 부스팅, regularization(정규화) 관련
- 학습 태스크 파라미터 : 학습에 사용되는 객체 함수, 평가 지표 등 설정

- 주요 일반 파라미터

booster	gbtree(트리 기반 모델, 디폴트) / gblinear(선형 모델)
silent	0(출력 메시지 x, 디폴트) / 1(출력 메시지 o)
nthread	CPU의 실행 스레드 개수(디폴트: 전체 스레드 사용)

파이썬 래퍼 XGBoost 하이퍼 파라미터

- 주요 일반 파라미터

alias : 별칭. 이 이름으로 사용해도 동일한 설정 적용

eta	GBM의 learning rate와 같은 역할, 학습 반영률 0~1(디폴트: 0.3, 보통: 0.01~0.2, alias: learning_rate)
num_boost_rounds	GBM의 n_estimator와 같은 역할, weak learner의 개수
min_child_weight	트리 분할을 결정하는 데이터들의 weight 총합 (디폴트: 1) 클수록 분할 자제, 과적합 제어
gamma	트리의 리프 노드를 추가적으로 나눌지 결정하는 최소 손실 감소 값 (디폴트: 0, alias: min_split_loss) 값이 클수록 과적합 제어
max_depth	트리 기반 알고리즘의 max_depth와 같은 역할, 과적합 제어 (디폴트: 6, 보통: 3~10)
sub_sample	GBM의 subsample과 같은 역할, 데이터의 샘플링 비율 0~1(디폴트: 1, 보통: 0.5~1)
colsample_bytree	GBM의 max_features와 같은 역할, 반영할 피처 개수 (디폴트: 1)

파이썬 래퍼 XGBoost 하이퍼 파라미터

- 주요 부스터 파라미터

lambda	L2 Regulation 적용 값, 피처 많을 때 사용 검토 (디폴트: 1, alias: reg_lambda)
alpha	L1 Regulation 적용 값, 피처 많을 때 사용 검토 (디폴트: 0, alias: reg_alpha)
scale_pos_weight	비대칭한 클래스로 구성된 데이터 세트의 균형을 위한 파라미터 (디폴트: 1)

파이썬 래퍼 XGBoost 하이퍼 파라미터

- 주요 학습 태스크 파라미터

objective	최솟값을 가져야 할 손실 함수를 정의 binary:logistic : 이진 분류 / multi:softmax : 다중 분류 / multi:softprob : 개별 레이블 클래스에 해당되는 예측 확률 반환
eval_matric	검증에 사용되는 함수 rmse(회귀 디폴트) / mae / logloss / error(분류 디폴트) / merror / mlogloss / auc

- 과적합 제어

- eta↓, num_round↑
- max_depth ↓
- min_child_weight↑
- gamma↑
- subsample, colsample_bytree 조절

실습 - 위스콘신 유방암 데이터 세트

(1) 모듈 임포트, 데이터 로드 및 확인

(2) 데이터 분리

(3) XGBoost 모델 학습/예측/평가 (조기 중단 기능)

(4) 피처 중요도 시각화

파이썬 래퍼 실습 - 위스콘신 유방암 데이터 세트

(1) 모듈 임포트, 데이터 로드 및 확인

```
# 데이터 로드 및 확인
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')

# 데이터 로드
dataset = load_breast_cancer()
features = dataset.data
labels = dataset.target

# DF 만들고 살펴보기
cancer_df = pd.DataFrame(data=features, columns=dataset.feature_names)
cancer_df['target'] = labels
cancer_df.head(3)
```

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	mean fractal dimension	...	target
0	17.99	10.38	122.8	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	0.07871	...	0
1	20.57	17.77	132.9	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	0.05667	...	0
2	19.69	21.25	130.0	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	0.05999	...	0

3 rows × 31 columns

```
# 레이블 값 분포 확인
print(dataset.target_names)
print(cancer_df['target'].value_counts())
```

```
['malignant' 'benign']
1    357
0    212
Name: target, dtype: int64
```

타깃 레이블

- 악성(malignant): 0
- 양성(benign): 1

실습 - 위스콘신 유방암 데이터 세트

(2) 데이터 분리

- 피처 / 레이블 데이터 세트 분리, 학습 / 테스트 / 검증 데이터 세트 분리

```
# cacer_df에서 feature용 DataFrame과 Label용 Series 객체 추출
# 맨 마지막 칼럼이 Label. Feature용 DataFrame은 cancer_df의 첫번째 칼럼에서 맨 마지막 두번째 칼럼까지를 :-1 슬라이싱으로 추출
# 검증용 데이터 분할하여 XGBoost의 검증 성능 평가, 조기 중단 수행할 예정

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)

# 나눠진 데이터 수 확인
print(X_train.shape, X_test.shape)
print(X_tr.shape, X_val.shape)
```

train_test_split

X_train y_train

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

- DataFrame을 XGBoost 전용 데이터 객체 Dmatrix로 변경

```
# XGBoost만의 전용 데이터 객체 DMatrix 사용 => Numpy, Pandas 데이터 세트를 Dmatrix로 생성해 모델에 입력해야함!!
# 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)
```

실습 - 위스콘신 유방암 데이터 세트

(3) XGBoost 모델 학습/예측/평가 (조기 중단 기능)

- 하이퍼 파라미터 설정

```
# XGBoost 하이퍼 파라미터 설정 (주로 딕셔너리 형태로 입력)
params = {'max_depth': 3,
          'eta': 0.05,
          'objective': 'binary:logistic',
          'eval_metric': 'logloss'
         }
num_rounds = 400
```

eval_metric = logloss 지정

- 평가용 데이터 세트 설정, 학습 수행 (조기 중단)

```
# 평가용 데이터 세트 설정
# 학습 데이터 셋은 'train' 또는 평가 데이터 셋은 'eval'로 명기
eval_list = [(dtr, 'train'),(dval, 'eval')] # 또는 eval_list = [(dval,'eval')] 만 명기해도 무방

# 학습
# 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)
```

```
[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

[174]   train-logloss:0.01278   eval-logloss:0.26229
[175]   train-logloss:0.01267   eval-logloss:0.26086
[176]   train-logloss:0.01258   eval-logloss:0.26103
```

평가용 데이터 세트 지정:

학습용, 검증용 DMatrix

train() 으로 학습

조기 중단 기능: train() 함수에 early_stopping_rounds 입력하여 설정,

더 이상 지표 개선 없을 경우 num_boost_round 횟수 채우지 않고 빠져나옴

반드시 (평가용 데이터 세트 지정)과 eval_metric 함께 설정해야

⇒[126] 이후 early_stopping_rounds=50 동안 지표 개선x, 빠져나옴

실습 - 위스콘신 유방암 데이터 세트

(3) XGBoost 모델 학습/예측/평가 (조기 중단 기능)

- 예측

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

# 예측 결괏값이 아닌 예측 확률 값 반환, 예측값 결정 로직 추가
# 예측 확률이 0.5보다 크면 1, 그렇지 않으면 0으로 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])
```

```
predict() 수행 결괏값을 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]
```

predict() 로 예측,

예측 확률 값 반환

- 평가

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, roc_auc_score
from sklearn.metrics import f1_score, confusion_matrix, precision_recall_curve, roc_curve
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 = roc_auc_score(y_test, pred_proba)
    print('오차 행렬')
    print(confusion)
    print('정확도: {0:.4f}, 정밀도: {1:.4f}, 재현율: {2:.4f},₩
 F1: {3:.4f}, AUC: {4:.4f}'.format(accuracy, precision, recall, f1, roc_auc))

# 예측 성능 평가
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
```

3장의 get_clf_eval() 함수 적용

실습 - 위스콘신 유방암 데이터 세트

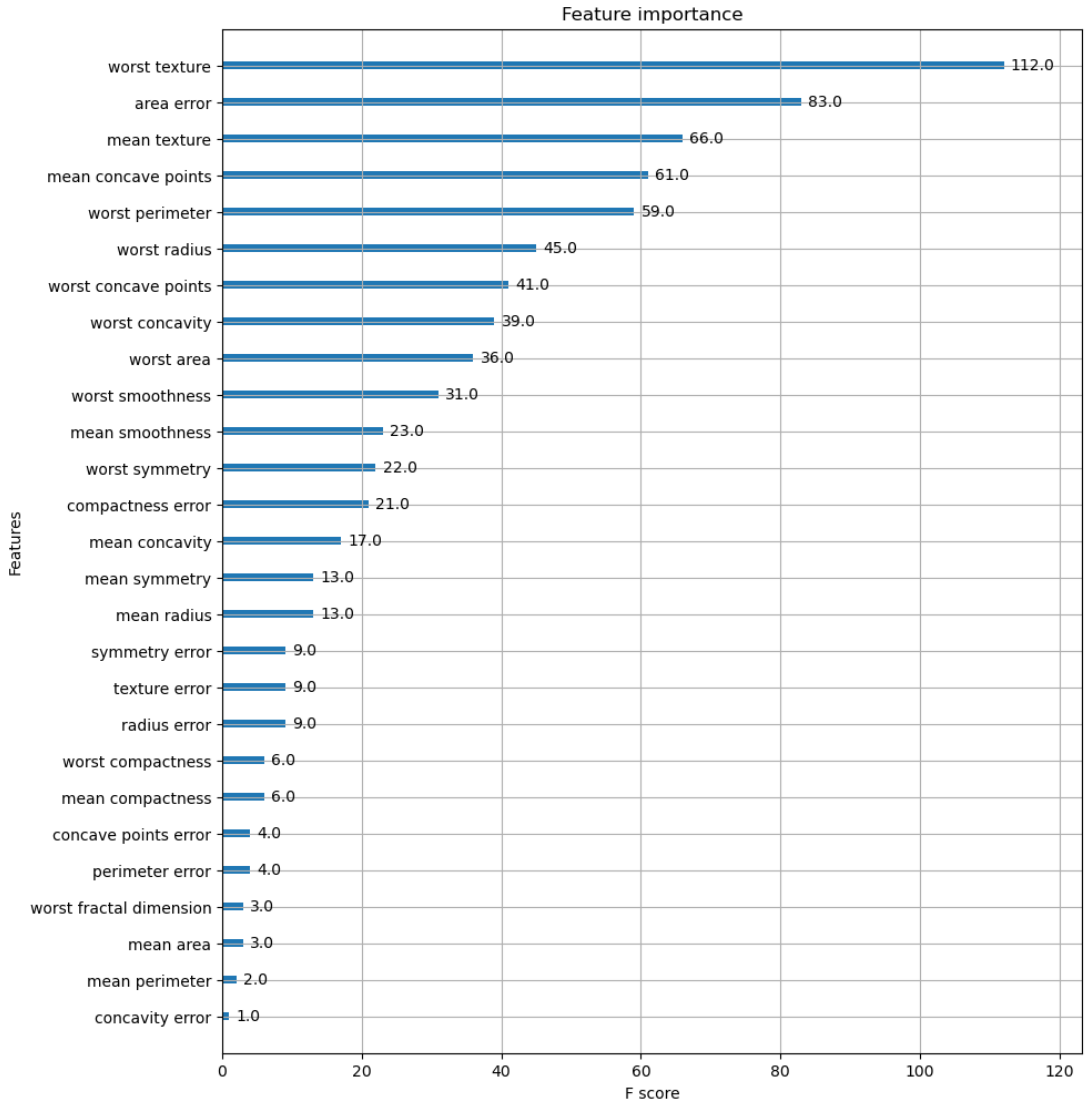
(4) 피처 중요도 시각화

- plot_importance()

```
# plot_importance() 로 피처 중요도 시각화
import matplotlib.pyplot as plt
%matplotlib inline

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

plot-importance



평가 지표: f 스코어

해당 피처가 트리 분할 시 얼마나 자주

사용되었는지 지표로 나타냄

to_graphviz():

xgboost에서 트리 기반 규칙 구조 시각화 가능

파이썬 래퍼 XGBoost 교차 검증

xgboost.cv(params, dtrain, num_boost_round, nfold, stratified, folds, metrics, obj,

feval, maximize, early_stopping_rounds, fpreproc, as_pandas, verbose_eval, show_stdv,

seed, call_back, shuffle)

사이킷런 GridSearchCV와 유사

데이터 세트에 대한 교차 검증 수행 후 최적 파라미터 구할 수 있음

이름	설명
params (dict)	부스터 파라미터
dtrain (DMatrix)	학습 데이터
num_boost_round (int)	부스팅 반복 횟수
nfold (int)	CV 폴드 개수
stratified (bool)	CV 수행 시 층화 표본 추출(Stratified sampling) 수행 여부
metrics (string or list of string)	CV 수행 시 모니터링할 성능 평가 지표
early_stopping_rounds (int)	조기 중단을 활성화시킴, 반복 횟수 지정

사이킷런 래퍼 XGBoost 하이퍼 파라미터

- 파이썬 - 사이킷런 파라미터명 차이

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

- 조기 중단 파라미터

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 - 학습 데이터가 아닌 별도의 데이터 세트여야 한다

사이킷런 래퍼 실습 - 위스콘신 유방암 데이터 세트

(1) 기본 학습/예측/평가

(2) 조기 중단 학습/예측/평가 (early_stopping_rounds=50)

(3) 조기 중단 학습/예측/평가 (early_stopping_rounds=10)

(4) 피처 중요도 시각화

파이썬 래퍼 실습 - 위스콘신 유방암 데이터 세트

(1) 기본 학습/예측/평가

```
# 파이썬 래퍼 XGBoost 실습 예제 노트북 이어서 사용
# 사이킷런 래퍼 XGBoost 클래스인 XGBClassifier 임포트
from xgboost import XGBClassifier

# Warning 메세지를 없애기 위해 eval_metric 값을 XGBClassifier 생성 인자로 입력.
# 모델 생성
xgb_wrapper = XGBClassifier(n_estimators=400, learning_rate=0.05, max_depth=3, eval_metric='logloss')
# 학습
xgb_wrapper.fit(X_train, y_train, verbose=True)
# 예측
w_preds = xgb_wrapper.predict(X_test)
w_pred_proba = xgb_wrapper.predict_proba(X_test)[:, 1]
# 평가
get_clf_eval(y_test, w_preds, w_pred_proba)
```

```
오차 행렬
[[34  3]
 [ 1 76]]
정확도: 0.9649, 정밀도: 0.9620, 재현율: 0.9870, F1: 0.9744, AUC: 0.9954
```

데이터 세트가 작아서 검증 데이터 분리 또는 교차 검증 등을 적용할 때 성능 수치가 불안정함

파이썬 래퍼 실습 - 위스콘신 유방암 데이터 세트

(2) 조기 중단 학습/예측/평가 (early_stopping_rounds=50)

```
# 조기 중단 수행
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)] # 파이썬 래퍼와 다르게 맨 앞이 학습용, 뒤 튜플이 검증용 데이터로 자동 인식
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]
```

early_stopping_rounds

```
[125]   validation_0-logloss:0.01998    validation_1-logloss:0.25714
[126]   validation_0-logloss:0.01973    validation_1-logloss:0.25587
[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
```

파이썬 래퍼의 조기 중단과 동일하게 학습 마무리

```
# 평가
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
```

파이썬 래퍼의 조기 중단과 성능 동일

파이썬 래퍼 실습 - 위스콘신 유방암 데이터 세트

(3) 조기 중단 학습/예측/평가 (early_stopping_rounds=10)

```
# early_stopping_rounds를 10으로 설정하고 재학습
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)
```

```
[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
```

조기 중단값이 너무 작아서 아직 성능이 향상될 여지가 있음에도 학습이 멈춰 예측 성능 저하

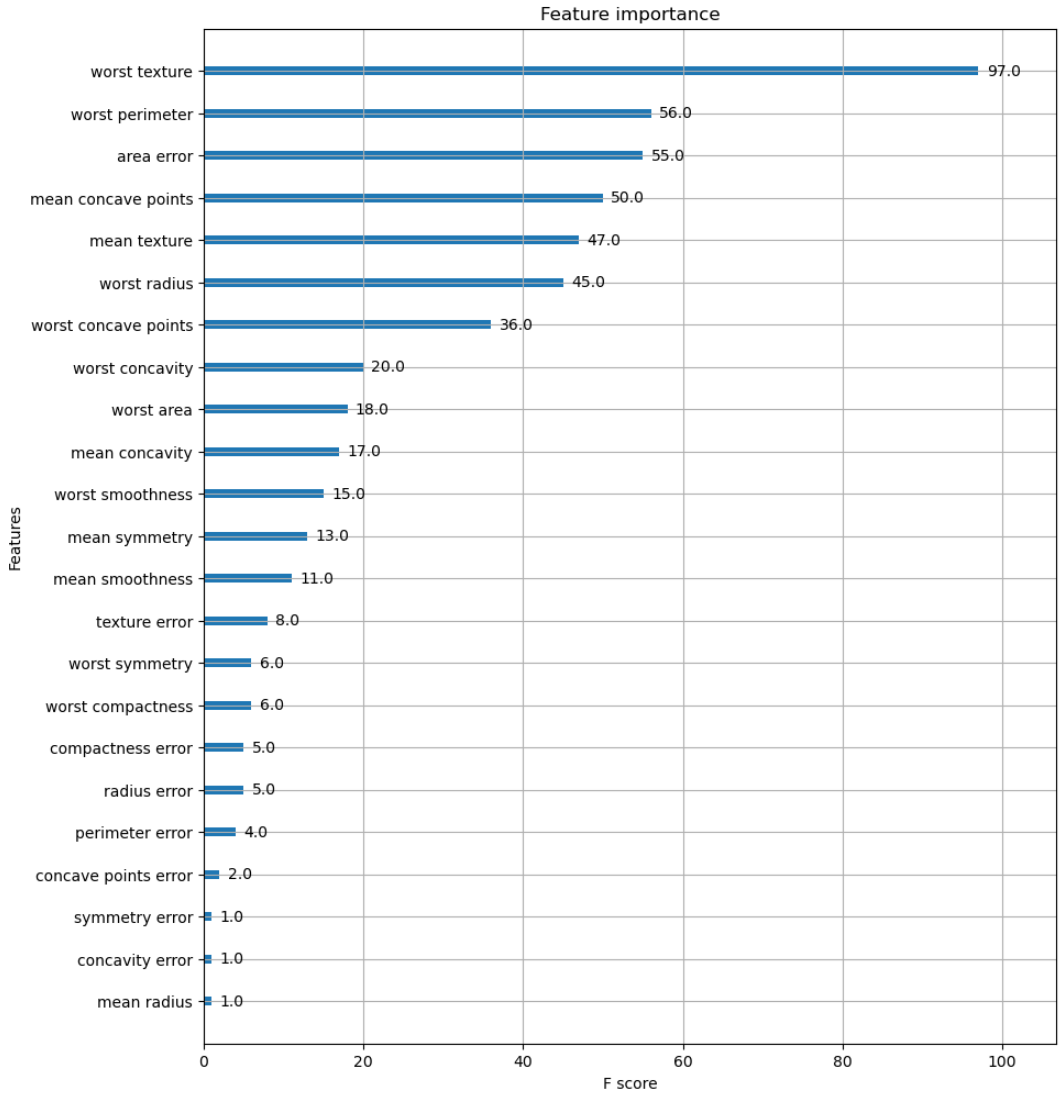
파이썬 래퍼 실습 - 위스콘신 유방암 데이터 세트

(4) 피처 중요도 시각화

```
# 피처 중요도 시각화
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)
```

파이썬 래퍼 클래스와 결과 동일



06.

LightGBM

LightGBM

: GBM 기반, XGBoost 장점 계승, 단점 보완한 알고리즘

- XGBoost 대비 장점

- 빠른 학습과 예측 수행 시간, 더 작은 메모리 사용량

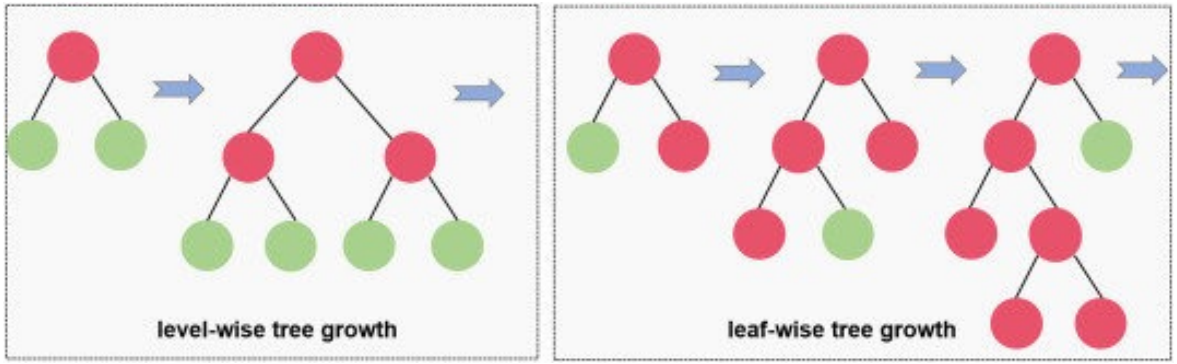
- XGBoost와 예측 성능은 비슷하지만 기능상 더 다양

- 카테고리형 피처의 자동 변환과 최적 분할

-리프중심 트리 분할(Leaf Wise) : 일반적인 트리 분할 방식처럼 트리의 균형을 맞추는 대신

-리프 노드를 지속적으로 분할하면서 트리의 깊이가 깊어지고 비대칭적인 규칙 트리가 생성

➡ 단점 : 과적합 가능성 증가



LightGBM 하이퍼 파라미터

- 주요 파라미터

num_iterations	반복 수행하려는 트리의 개수 일정 수준까지 증가 시 예측 성능 향상 (디폴트: 100, 사이킷런 클래스: n_estimator)
learning_rate	부스팅 스텝을 반복적으로 수행할 때 업데이트되는 학습률 0~1(디폴트: 0.1)
max_depth	트리 기반 알고리즘의 max_depth < 0 이면 깊이 제한X (디폴트: -1)
min_data_in_leaf	결정 트리의 min_samples_leaf와 같은 역할 (디폴트: 20, 사이킷런 클래스: min_child_samples)
num_leaves	하나의 트리가 가질 수 있는 최대 리프 개수 (디폴트: 31)
boosting	부스팅의 트리를 생성하는 알고리즘 gbdt(디폴트) : GBM / rf : 랜덤 포레스트

LightGBM 하이퍼 파라미터

bagging_fraction	데이터 샘플링 비율 지정, 과적합 제어 (디폴트: 1, 사이킷런 클래스: 동일)
feature_fraction	개별 트리를 학습할 때마다 무작위로 선택하는 feature의 비율 (디폴트: 1, 사이킷런 클래스: 동일)
lambda_l2	L2 regulation 제어, 과적합 제어(피처 많을 때 적용) (디폴트: 0, 사이킷런 클래스: reg_lambda)
lambda_l1	L1 regulation 제어, 과적합 제어(피처 많을 때 적용) (디폴트: 0, 사이킷런 클래스: reg_alpha)

- 학습 태스크 파라미터

objective	최솟값 가져야 할 손실 함수 정의 binary:logistic : 이진 분류 / multi:softmax : 다중 분류 / multi:softprob : 개별 레이블 클래스에 해당되는 예측 확률 반환

실습 - 위스콘신 유방암 데이터 세트

(1) LGBMClassifier 학습/예측/평가

(2) 피처 중요도 시각화

파이썬 래퍼 실습 - 위스콘신 유방암 데이터 세트

(1) LGBMClassifier 학습/예측/평가

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

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

# 데이터 로드 및 피처/레이블 분리
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]

# 학습/테스트/검증용 데이터 분리
# 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 학습, 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)

# 모델 생성
# 앞서 XGBoost와 동일하게 n_estimators는 400
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)]
# 학습
lgbm_wrapper.fit(X_tr, y_tr, early_stopping_rounds=50, eval_metric="logloss", eval_set=evals, verbose=True)
# 예측
preds = lgbm_wrapper.predict(X_test)
pred_proba = lgbm_wrapper.predict_proba(X_test)[:, 1]
```

```
[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

[111]   training's binary_logloss: 0.00850714   valid_1's binary_logloss: 0.280894
```

XGBoost와 동일하게 진행

- 모듈 임포트
- 데이터 로드
- 피처/레이블 데이터 분리
- 학습/테스트/검증용 데이터 분리
- 모델 생성
- 평가용 데이터 지정
- 학습
- 예측
- 평가

```
# 예측 성능 평가
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
```

3장 평가 함수 사용

파이썬 래퍼 실습 - 위스콘신 유방암 데이터 세트

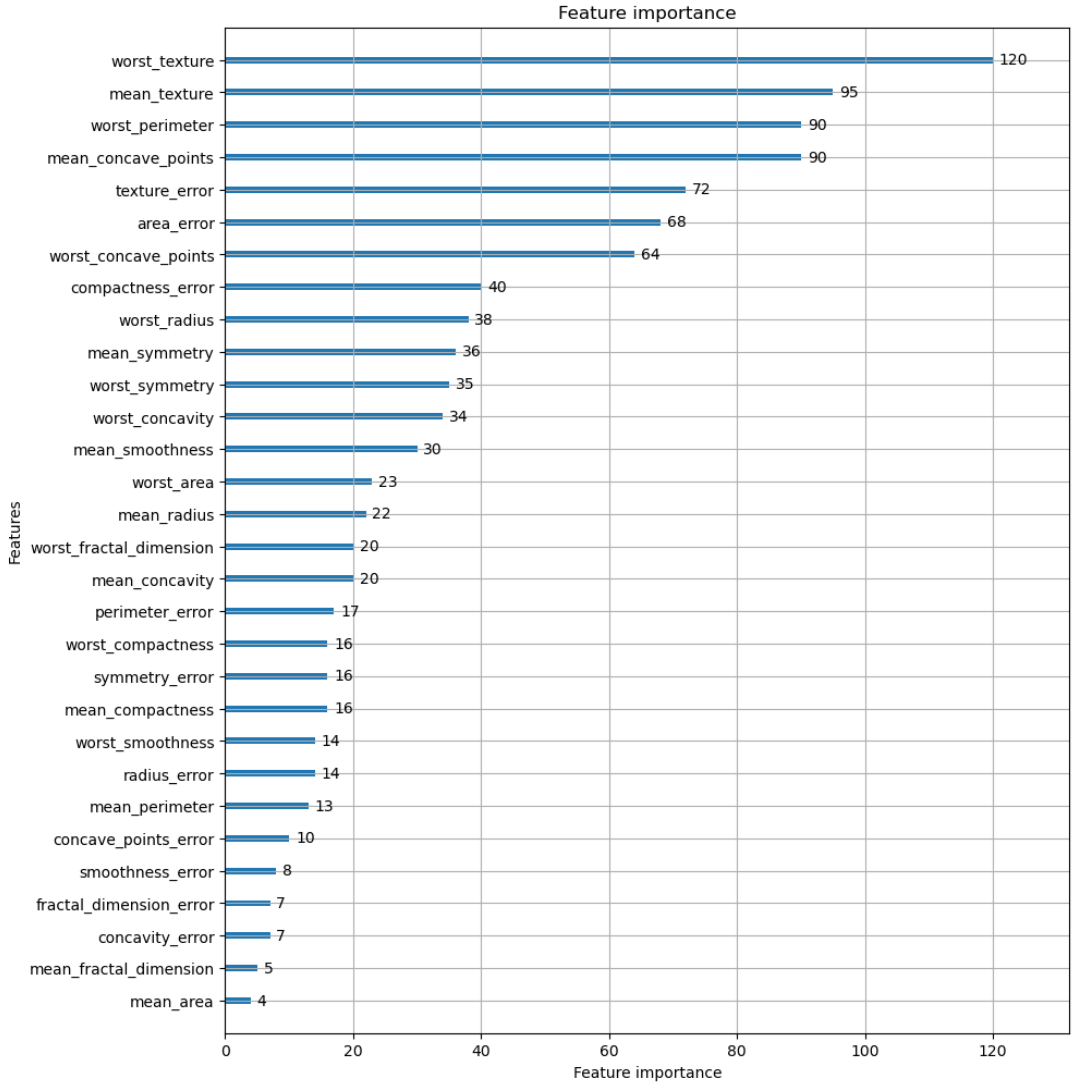
(2) 피처 중요도 시각화

```
# plot_importance()를 이용해 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)
```

xgboost와 동일하게

피처 중요도 시각화 내장 API 제공



07.

베이지안 최적화 기반의 HyperOpt를 이용한 하이퍼 파라미터 튜닝

베이지안 최적화 기반의 HyperOpt를 이용한 하이퍼 파라미터 튜닝

부스팅 알고리즘 모델은 하이퍼 파라미터가 많아 Grid Search로 튜닝 시,

대용량 데이터의 경우 기하급수적으로 최적화 시간이 증가

다른 방식 적용 ⇒ 베이지안 최적화 기법

- 하이퍼 파라미터 최적화 방식

Manual Search : 직관, 노하우 등에 의존하여 최적 하이퍼 파라미터를 직접 탐색하는 방법

Grid Search : 특정 탐색 구간의 하이퍼 파라미터 값들을 일정 간격으로 선정하여 탐색하는 방법

Random Search : 특정 탐색 구간의 하이퍼 파라미터 값들을 랜덤 샘플링으로 선정하여 탐색하는 방법

Bayesian optimization

...

베이지안 최적화

: 목적 함수 식을 제대로 알 수 없는 블랙 박스 형태의 함수에서

최대 또는 최소 함수 반환 값을 만드는 최적 입력값을 효과적으로 찾아주는 방식

최적함수

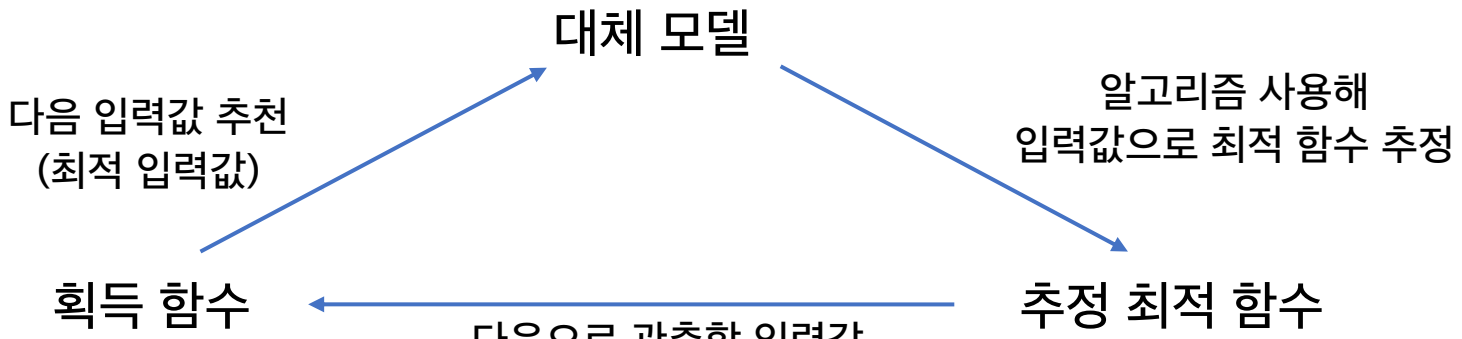
- 베이지안 확률 기반 최적화 기법

- 필수 구성 요소:

① 대체 모델: 획득 함수의 입력값 바탕으로, 미지의 목적 함수의 형태에 대해 확률적인 추정을 하는 모델

② 획득 함수: 대체모델 이 목적 함수에 대해 확률적으로 추정한 결과를 바탕으로,

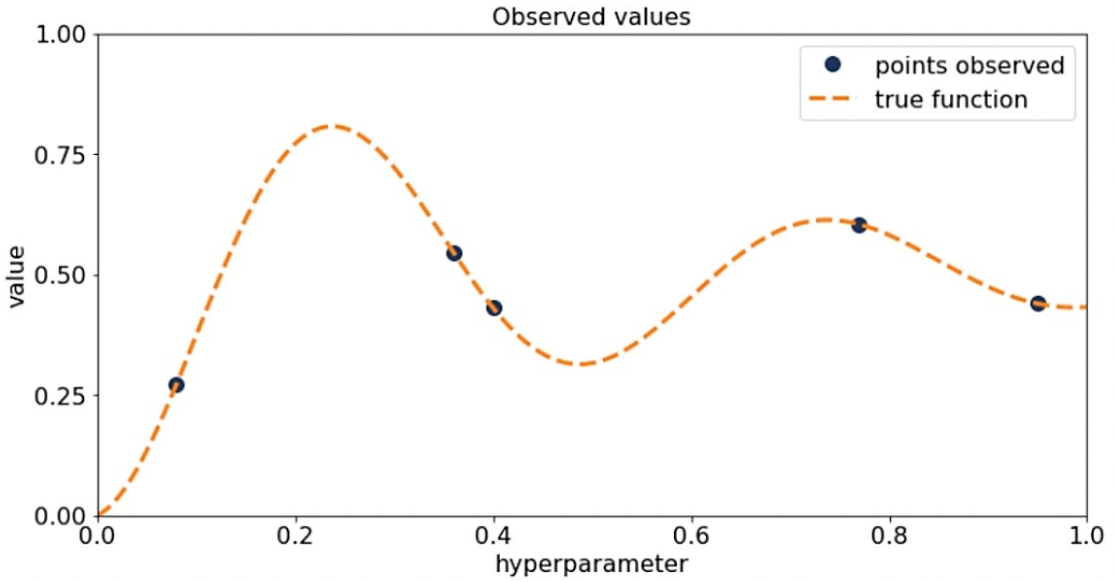
최적 입력값을 찾는 데 있어 가장 유용할 만한 다음 입력값 추천하는 함수



하이퍼 파라미터 튜닝 시,
입력값 = 하이퍼 파라미터

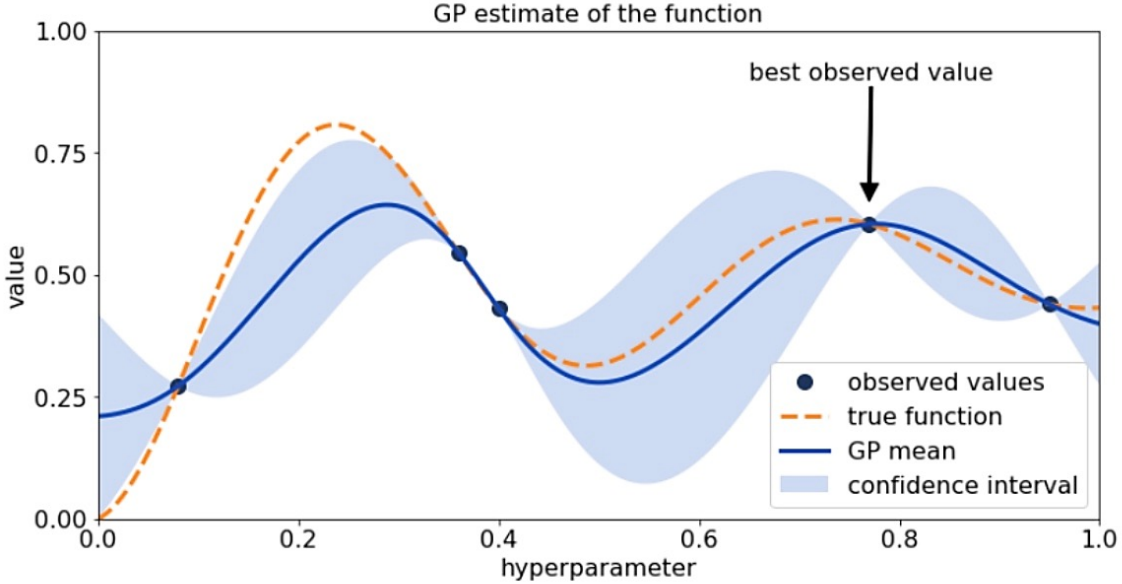
베이지안 최적화

Step 1



- 최초에는 랜덤하게 하이퍼 파라미터들을 샘플링하고 성능 관찰
- 주황색 실선 : 목표 실제함수
- 검은색 점 : 특정 하이퍼 파라미터 값에 대한 관측된 성능 지표 결괏값

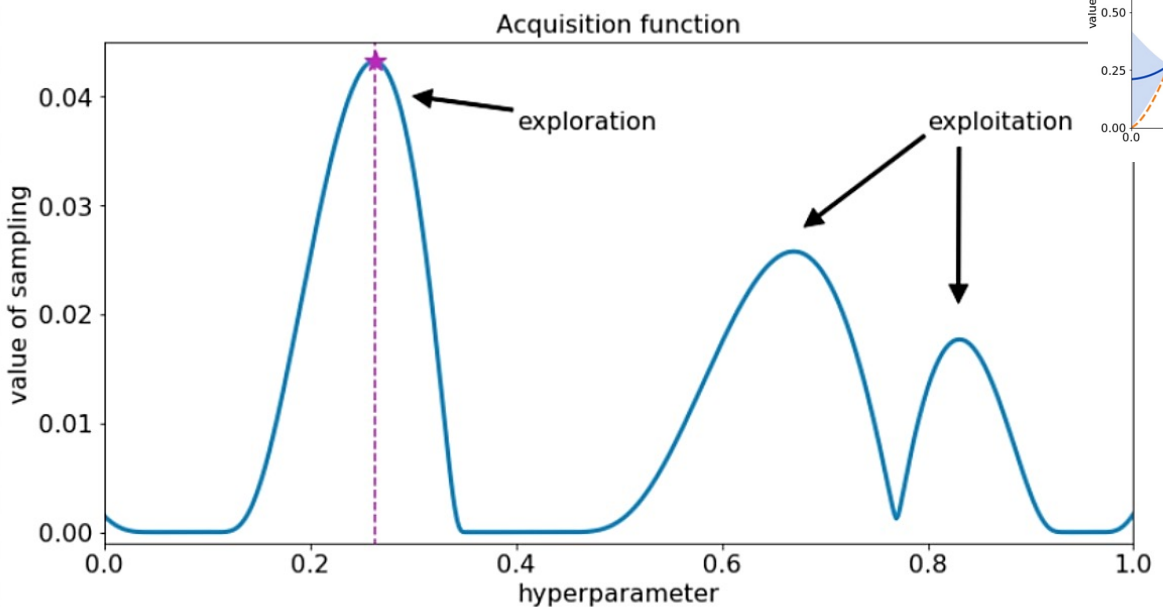
Step 2

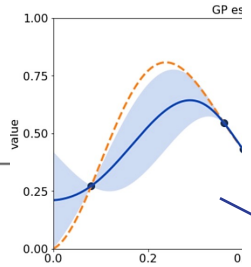


- 관측된 값을 기반으로 대체 모델은 실제함수 를 추정
- 파란색 영역 : 예측된 함수의 신뢰구간 분산
- 최적 관측값은 Y축 최댓값 가질 때 의 하이퍼 파라미터

베이지안 최적화

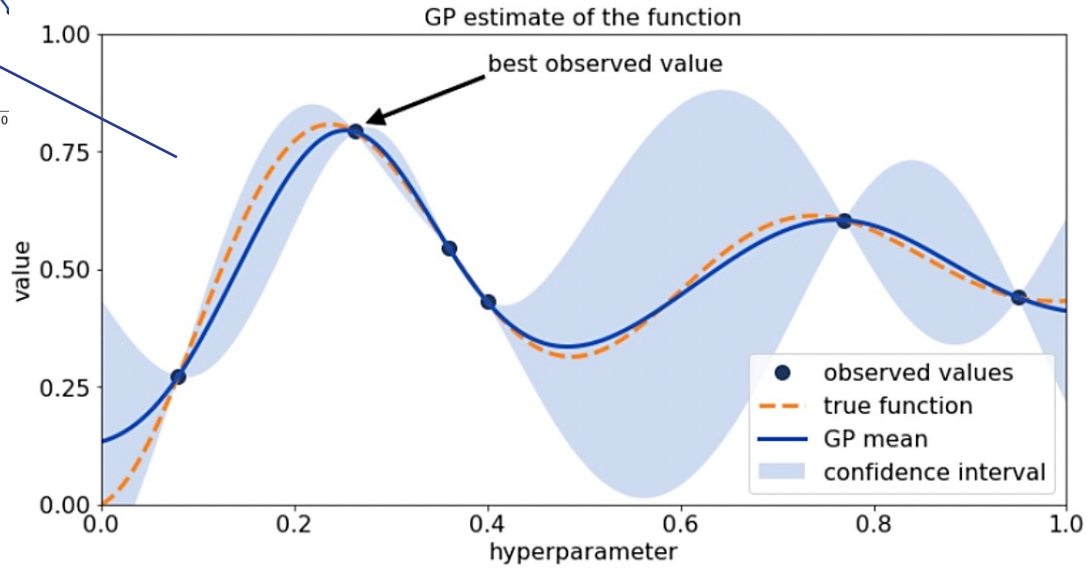
Step 3





신뢰구간 줄어듦 → 불확실성 개선
→ 최적 함수 추정 개선

Step 4



- 획득 함수는 다음으로 관측할 하이퍼 파라미터 값 계산
- 획득 함수는 이전의 최적 관측값보다 더 큰 최댓값을 가질 가능성이 높은 지점을 찾아서 대체 모델에 전달

- 획득 함수로부터 전달된 하이퍼 파라미터를 수행하여 관측된 값을 기반으로 대체 모델은 갱신되어 다시 최적함수 예측

HyperOPT

: 베이지안 최적화를 하이퍼 파라미터 튜닝에 적용할 수 있도록 제공되는 파이썬 패키지 중 하나

대체 모델이 최적 함수를 추정할 때 트리 파르젠 사용

목적 함수 반환값의 최솟값을 가지는 최적 입력값 유추

- 활용 로직

① 입력 변수명, 입력값의 검색 공간 설정

② 목적 함수 설정

③ 목적 함수의 반환값이 최솟값을 가지게 하는 최적 입력값 유추

HyperOPT

- 입력값의 검색 공간 제공 함수

hyperopt의 hp 모듈: from hyperopt import hp

함수명	설명
hp.quniform()	검색 공간을 범위를 이용해서 설정 매개 변수 : label(파라미터명), low(최솟값), high(최댓값), q(간격)
hp.uniform()	검색 공간을 정규 분포 형태로 설정 매개 변수 : : label(파라미터명), low(최솟값), high(최댓값)
hp.randint()	검색 공간을 일정 범위 내에서 랜덤한 정숫값으로 설정 매개 변수 : label(파라미터명), upper(최댓값)
hp.loguniform()	반환값 : exp(uniform(low, high)) 반환값의 log 변환된 값이 정규 분포를 띄도록 검색 공간 설정
hp.choice()	검색값에 문자열이 포함되어 있을 경우 설정

HyperOPT

- 목적 함수의 최적 입력값 유추 함수 fmin()

hyperopt의 fmin 모듈: from hyperopt import fmin

fmin(fn, space, algo, max_evals, trials)

파라미터명	설명
fn	목적 함수
space	search_space와 같은 검색 공간 딕셔너리
algo	베이지안 최적화 적용 알고리즘 - tpe.suggest (TPE)
max_evals	최적 입력값을 찾기 위한 입력값 시도 횟수
trials	최적 입력값을 찾기 위해 시도한 입력값 및 해당 입력값의 목적 함수 반환값 저장에 사용
rstate	fmin()을 수행할 때마다 동일한 결괏값을 가질 수 있도록 설정하는 랜덤 시드

실습1 - 기본 로직

(1) 입력 변수명, 입력값의 검색 공간 설정

(2) 목적 함수 생성

(3) 목적 함수의 반환값이 최솟값을 가지도록 하는 최적 입력값 유추

(4) Trials 객체 results, vals 속성

실습1 - 기본 로직

(1) 입력 변수명, 입력값의 검색 공간 설정

```
# 입력 변수명과 입력값의 검색 공간 설정
from hyperopt import hp

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

quniform

(2) 목적 함수 생성

```
from hyperopt import STATUS_OK

# 목적 함수 생성 변숫값과 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 # 목적 함수식

    return retval
```

목적 함수 = x^2 - 20y

현 입력값 검색 공간에서 x=0, y=15에 가까울수록 최소 반환값 근사

실습1 - 기본 로직

(3) 목적 함수의 반환값이 최솟값을 가지도록 하는 최적 입력값 유추

- fmin(), max_evals=5

```
from hyperopt import fmin, tpe, Trials
import numpy as np
# 입력 결괏값을 저장한 Trials 객체값 생성
trial_val = Trials()

# 목적 함수의 최솟값을 반환하는 최적 입력 변숫값을 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, 238.16trial/s, best loss: -224.0]
best: {'x': -4.0, 'y': 12.0}
```

- fmin(), max_evals=20

```
trial_val = Trials()

# max_evals=20 로 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, 499.78trial/s, best loss: -296.0]
best: {'x': 2.0, 'y': 15.0}
```

x=2, y=15로 최소 반환값 가지도록 하는 최적 입력값에 근사함

실습1 - 기본 로직

(4) Trials 객체 results, vals 속성

- results 확인

```
# Trials 객체의 result 속성에 파이썬 리스트로 목적 함수 반환값들이 저장됨
# 리스트 내부의 개별 원소는 {'loss': 함수 반환값, 'status': 반환 상태값}와 같은 딕셔너리
print(trial_val.results)
```

```
[{'loss': -64.0, 'status': 'ok'}, {'loss': -184.0, 'status': 'ok'}, {'loss': 56.0, 'status': 'ok'}, {'loss': -224.0, 'status': 'ok'},
{'loss': 61.0, 'status': 'ok'}, {'loss': -296.0, 'status': 'ok'}, {'loss': -40.0, 'status': 'ok'}, {'loss': 281.0, 'status': 'ok'},
{'loss': 64.0, 'status': 'ok'}, {'loss': 100.0, 'status': 'ok'}, {'loss': 60.0, 'status': 'ok'}, {'loss': -39.0, 'status': 'ok'}, {'l
oss': 1.0, 'status': 'ok'}, {'loss': -164.0, 'status': 'ok'}, {'loss': 21.0, 'status': 'ok'}, {'loss': -56.0, 'status': 'ok'}, {'los
s': 284.0, 'status': 'ok'}, {'loss': 176.0, 'status': 'ok'}, {'loss': -171.0, 'status': 'ok'}, {'loss': 0.0, 'status': 'ok'}]
```

fmin()이 max_evals=20으로 20회 반복 수행 → 20개 딕셔너리를 개별 원소로 가지는 리스트

loss: 함수반환값, status: 반환상태값 (ok-평가 성공적으로 완료, fail-평가 중 오류 발생)

results: [{'loss': 함수 반환값, 'status': 반환 상태값}, … 반복 수행 시마다 반환]

- vals 확인

```
# Trials 객체의 vals 속성에 {'입력변수명': 개별 수행 시마다 입력된 값의 리스트} 형태로 저장됨
print(trial_val.vals)
```

```
{'x': [-6.0, -4.0, 4.0, -4.0, 9.0, 2.0, 10.0, -9.0, -8.0, -0.0, -0.0, 1.0, 9.0, 6.0, 9.0, 2.0, -2.0, -4.0, 7.0, -0.0], 'y': [5.0, 10.
0, -2.0, 12.0, 1.0, 15.0, 7.0, -10.0, 0.0, -5.0, -3.0, 2.0, 4.0, 10.0, 3.0, 3.0, -14.0, -8.0, 11.0, -0.0]}
```

20회 반복 수행 → x, y 각 수행 시마다 사용된 입력값들 리스트

vals: {'입력변수명1': 개별 수행시마다 입력된 값 리스트, …입력 변수마다}

실습1 - 기본 로직

(4) Trials 객체 results, vals 속성

- results, vals 이용한 최적화 경과 보기

```
# Trials 객체의 results, vals 속성 이용해 최적화 경과 보기
# 각 회마다 x, y, loss 값 보여줌
import pandas as pd

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

# DF으로 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
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

Trials 객체의 속성을 이용해

최적화 경과를 직관적으로 볼 수 있음

실습2 - XGBoost 하이퍼 파라미터 최적화 (위스콘신 데이터)

(1) 데이터 로드 및 분리

(2) 입력 변수명, 입력값의 검색 공간 설정

(3) 목적 함수 생성

(4) 목적 함수의 반환값이 최솟값을 가지도록 하는 최적 입력값 유추

(5) 최적 하이퍼 파라미터로 XGBClassifier 재학습/예측/평가

실습1과 같은 흐름,

주의점

① 정수형 하이퍼 파라미터 입력시, HyperOPT 반환값이 실수형이면 형변환하여 입력

② HyperOPT의 목적함수는 최솟값 반환하도록 최적화하므로 정확도처럼 클수록 좋은 지표는 -1 을 곱해야

실습2 - XGBoost 하이퍼 파라미터 최적화 (위스콘신 데이터)

(1) 데이터 로드 및 분리

```
# 데이터 로드 및 분리
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')

dataset = load_breast_cancer()

cancer_df = pd.DataFrame(data=dataset.data, columns=dataset.feature_names)
cancer_df['target']= dataset.target
# 피처, 레이블 분리
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 )

# 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=156 )
```

XGBoost, LightGBM 예제에서 사용한 코드

실습2 - XGBoost 하이퍼 파라미터 최적화 (위스콘신 데이터)

(2) 입력 변수명, 입력값의 검색 공간 설정

```
# 입력 변수명, 입력값의 검색 공간 설정
from hyperopt import hp

# max_depth는 5에서 20까지 1간격으로, min_child_weight는 1에서 2까지 1간격으로 (정수)
# colsample_bytree는 0.5에서 1사이, learning_rate는 0.01에서 0.2 사이 정규 분포된 값으로 검색 (실수)
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),
                   }
```

정수형: quniform()

실수형: uniform() 사용

실습2 - XGBoost 하이퍼 파라미터 최적화 (위스콘신 데이터)

(3) 목적 함수 생성

```
# 목적 함수 생성
from sklearn.model_selection import cross_val_score
from xgboost import XGBClassifier
from hyperopt import STATUS_OK

# !주의1! fmin()에서 입력된 search_space 값으로 입력된 모든 값은 실수형
# XGBClassifier의 정수형 하이퍼 파라미터는 정수형 변환 필요

# !주의2! 정확도는 높을수록 더 좋은 수치. -1 * 정확도를 곱해서 큰 정확도 값일수록 최소가 되도록 변환
def objective_func(search_space):
    # 수행 시간 절약을 위해 nestimators는 100으로 축소
    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)

    # accuracy는 cv=3 개수만큼 roc-auc 결과를 리스트로 가짐. 이를 평균해서 반환하되 -1을 곱함.
    return {'loss': -1 * np.mean(accuracy), 'status': STATUS_OK}
```

XGBClassifier 생성, 교차검증

목적 함수 반환값:

교차 검증 기반의 평균 정확도

정수형 파라미터 명시적 형변환

최종 반환값은 평균 정확도* -1

실습2 - XGBoost 하이퍼 파라미터 최적화 (위스콘신 데이터)

(4) 목적 함수의 반환값이 최솟값을 가지도록 하는 최적 입력값 유추

```
# 최적 입력값 유추
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 횟수 50 지정
            trials=trial_val, rstate=np.random.default_rng(seed=9))
print('best:', best)
```

```
100%|██████████████████████████████████████████████████| 50/50 [00:29<00:00,  1.70trial/s, best loss: -0.9670616939700244]
best: {'colsample_bytree': 0.5424149213362504, 'learning_rate': 0.12601372924444681, 'max_depth': 17.0, 'min_child_weight': 2.0}
```

정수형 하이퍼 파라미터가 실수형 값으로 도출됨 주의

```
# 실수형 소수점 5자리까지, 정수형 변환하여 확인
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:0.54241, learning_rate:0.12601, max_depth:17, min_child_weight:2
```

실습2 - XGBoost 하이퍼 파라미터 최적화 (위스콘신 데이터)

(5) 최적 하이퍼 파라미터로 XGBClassifier 재학습/예측/평가

```
# 최적 하이퍼 파라미터로 XGBClassifier 재학습/예측/평가
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)
```

```
[0]     validation_0-logloss:0.58942    validation_1-logloss:0.62048
[1]     validation_0-logloss:0.50801    validation_1-logloss:0.55913
[2]     validation_0-logloss:0.44160    validation_1-logloss:0.50928

[234]   validation_0-logloss:0.01324    validation_1-logloss:0.22773
[235]   validation_0-logloss:0.01322    validation_1-logloss:0.22743
[236]   validation_0-logloss:0.01320    validation_1-logloss:0.22713
오차 행렬
[[35  2]
 [ 2 75]]
정확도: 0.9649, 정밀도: 0.9740, 재현율: 0.9740,    F1: 0.9740, AUC:0.9944
```

get_clf_eval() 3장 평가 함수 사용

앞에서 튜닝하지 않은 결과보다는 약간 좋지만,

데이터 세트가 작기 때문에 불안정한 성능 결과

08.

스태킹 앙상블

스태킹 앙상블

: 개별 알고리즘으로 예측한 데이터 기반으로 다시 예측 수행하는 방식

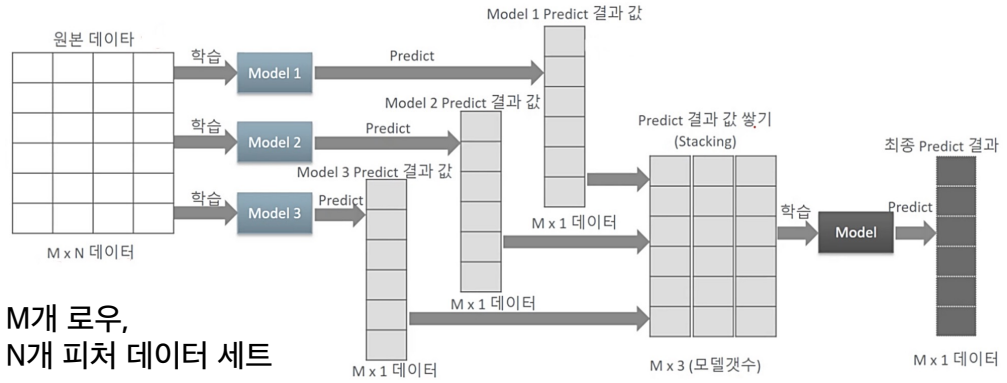
- 스태킹 앙상블 모델

- 구성모델

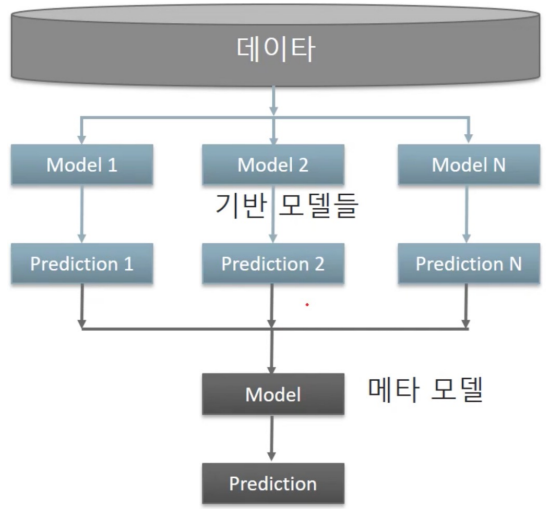
①개별기반모델: 많은 개별 모델 필요

②최종메타모델: 개별 기반 모델의 예측 데이터를 스태킹 형태로 결합해 학습 데이터로 만들어 학습

메타모델: 개별 모델의 예측된 데이터 세트를 기반으로 학습하고 예측하는 방식



M개 로우,
N개 피처 데이터 세트



기본 스태킹 모델 실습 - 위스콘신 유방암 데이터 세트

(1) 모듈 임포트, 데이터 준비

(2) 개별, 최종 모델 Classifier 생성

(3) 개별 모델 학습/예측/평가

(4) 최종 메타 모델 학습 데이터 생성

(5) 최종 메타 모델 학습/예측/평가

기본 스태킹 모델 실습 - 위스콘신 유방암 데이터 세트

(1) 모듈 임포트, 데이터 준비

```
# 모듈 임포트
import numpy as np

from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression

from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# 데이터 로드
cancer_data = load_breast_cancer()

X_data = cancer_data.data
y_label = cancer_data.target

# 학습/테스트용 데이터 셋 분리
X_train , X_test , y_train , y_test = train_test_split(X_data , y_label , test_size=0.2 , random_state=0)
```

기본 스태킹 모델 실습 - 위스콘신 유방암 데이터 세트

(2) 개별, 최종 모델 Classifier 생성

```
# 개별 ML 모델 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 모델 생성
lr_final = LogisticRegression(C=10)
```

개별 모델: KNN, 랜덤 포레스트, 결정 트리, 에이다부스트

최종 모델: 로지스틱 회귀

(3) 개별 모델 학습/예측/평가

```
# 개별 모델들 학습
knn_clf.fit(X_train, y_train)
rf_clf.fit(X_train , y_train)
dt_clf.fit(X_train , y_train)
ada_clf.fit(X_train, y_train)

# 학습된 개별 모델들 각자 예측 데이터 셋을 생성, 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: 0.9035
에이다부스트 정확도: 0.9561 :
```

기본 스태킹 모델 실습 - 위스콘신 유방암 데이터 세트

(4) 최종 메타 모델 학습 데이터 생성

```
# 최종 메타 모델의 학습 데이터 생성
# 개별 알고리즘의 예측값을 옆으로 붙여 피처값으로 만듦

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

# transpose를 이용해 행과 열의 위치 교환. 컬럼 레벨로 각 알고리즘의 예측 결과를 피처로 만듦.
pred = np.transpose(pred)
print(pred.shape)
```

```
(4, 114)
(114, 4)
```

(5) 최종 메타 모델 학습/예측/평가

```
# 최종 메타 모델 학습/예측/평가
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
```

개별 모델 정확도보다 향상됨

(무조건 좋아지는 것은 아님)

```
KNN 정확도: 0.9211
랜덤 포레스트 정확도: 0.9649
결정 트리 정확도: 0.9035
에이다부스트 정확도: 0.9561 :
```

CV 세트 기반 스태킹

: 과적합 개선을 위해 개별 모델이 각각 교차검증 으로 메타 모델을 위한 학습용 스태킹 데이터 생성과

예측을 위한 테스트용 스태킹 데이터를 생성해 이를 기반으로 메타 모델이 학습, 예측하는 방식

- 단계

① 개별 기반 모델별로 학습/예측 수행

- 학습 데이터를 K개의 fold로 나눔

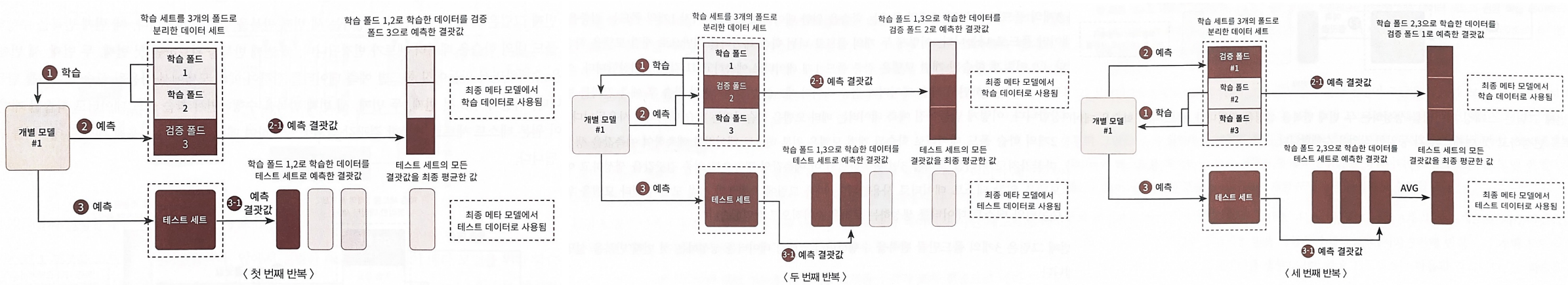
- K-1 개의 fold를 학습 데이터로 하여 base 모델 학습 (K번 반복)

  * 검증 fold 1개를 예측한 결과 (K fold) -> 최종 meta 모델의 학습데이터

  * 테스트 데이터를 예측한 결과의 평균 -> 최종 meta 모델의 테스트데이터

CV 세트 기반 스태킹

① 개별 기반 모델별로 학습/예측 수행



CV 세트 기반 스태킹

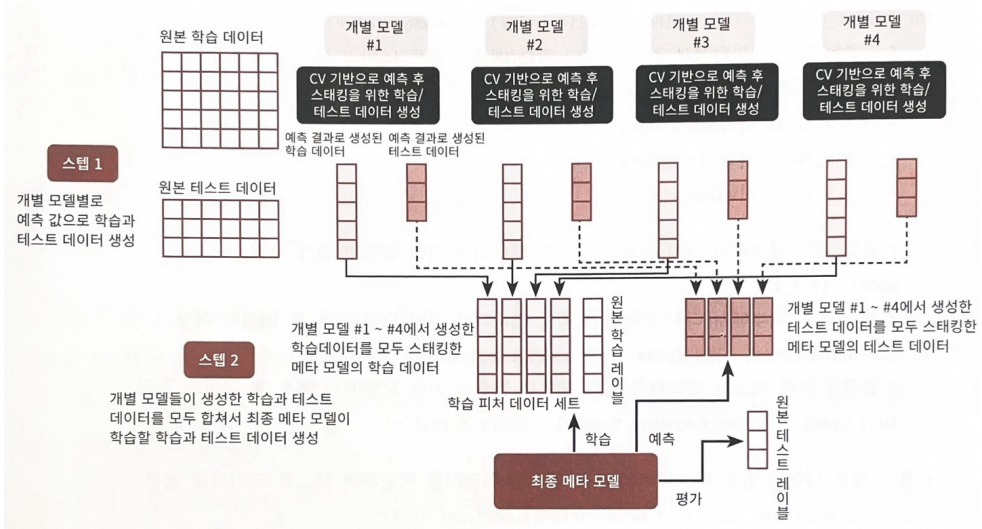
② 최종 메타 모델 학습/예측/평가

- 각 base 모델이 생성한 학습용 데이터를 stacking -> 최종 meta 모델의 학습용 데이터 세트

- 각 base 모델이 생성한 테스트용 데이터를 stacking -> 최종 meta 모델의 테스트용 데이터 세트

- 최종 학습용 데이터 + 원본 학습 레이블 데이터로 학습

- 최종 테스트용 데이터로 예측 -> 원본 테스트 레이블 데이터로 평가



실습 - CV 세트 기반 스태킹 구현

(1) 최종 메타 모델 학습/테스트용 데이터 생성 함수 정의

(2) 개별 모델로 최종 메타 모델 학습/테스트용 데이터 생성

(3) 스태킹 학습/테스트용 데이터 생성

(4) 최종 메타 모델 학습/예측/평가

실습 - CV 세트 기반 스태킹 구현

(1) 최종 메타 모델 학습/테스트용 데이터 생성 함수 정의

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

# 개별 기반 모델에서 최종 메타 모델의 학습 및 테스트용 데이터 생성 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('₩t 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
```

실습 - CV 세트 기반 스태킹 구현

(2) 개별 모델로 최종 메타 모델 학습/테스트용 데이터 생성

```
# 각 개별 모델로 최종 메타 모델의 학습/테스트 데이터 생성
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)
```

```
KNeighborsClassifier  model 시작
        폴드 세트:  0  시작
        폴드 세트:  1  시작
        폴드 세트:  2  시작
        폴드 세트:  3  시작
        폴드 세트:  4  시작
        폴드 세트:  5  시작
        폴드 세트:  6  시작
RandomForestClassifier  model 시작
        폴드 세트:  0  시작
        폴드 세트:  1  시작
        폴드 세트:  2  시작
        폴드 세트:  3  시작
        폴드 세트:  4  시작
        폴드 세트:  5  시작
        폴드 세트:  6  시작
DecisionTreeClassifier  model 시작
        폴드 세트:  0  시작
        폴드 세트:  1  시작
        폴드 세트:  2  시작
        폴드 세트:  3  시작
        폴드 세트:  4  시작
        폴드 세트:  5  시작
        폴드 세트:  6  시작
AdaBoostClassifier  model 시작
        폴드 세트:  0  시작
        폴드 세트:  1  시작
        폴드 세트:  2  시작
        폴드 세트:  3  시작
        폴드 세트:  4  시작
        폴드 세트:  5  시작
        폴드 세트:  6  시작
```

(3) 스태킹 학습/테스트용 데이터 생성

```
# 각 학습/테스트 데이터 합치기
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 피처 데이터 Shape:',X_train.shape, '원본 테스트 피처 Shape:',X_test.shape)
print('스태킹 학습 피처 데이터 Shape:', Stack_final_X_train.shape,
      '스태킹 테스트 피처 데이터 Shape:',Stack_final_X_test.shape)
```

```
원본 학습 피처 데이터 Shape: (455, 30) 원본 테스트 피처 Shape: (114, 30)
스태킹 학습 피처 데이터 Shape: (455, 4) 스태킹 테스트 피처 데이터 Shape: (114, 4)
```

(4) 최종 메타 모델 학습/예측/평가

```
# 개별 모델의 학습/테스트 데이터로 최종 메타 모델 학/예/평
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: 0.9737
```

수고하셨습니다