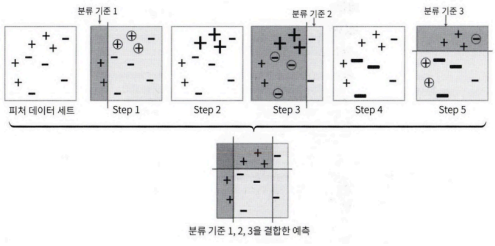
파머완 4장 part 2(4.5장 ~ 4.8장, 4.10장 ~ 4.11장)

05. GBM (Gradient Boosting Machine)

- 여러 개의 약한 학습기를 순차적으로 학습 - 예측하면서 잘못 예측한 데이터에 가중치 부여를 통해 오류를 개선해 나가면서 학습해 나가는 방식
- AdaBoost, Gradient Boost

AdaBoost

- 개별 약한 학습기는 각각 가중치를 부여해 결합함



GBM (Gradient Boost Machine)

- 가중치 업데이트를 경사 하강법을 이용함
- 오류식 : h(x) =y - F(x)
- 오류식을 최소화하는 방향성을 가지고 반복적으로 가중치 값을 업데이트
- GradientBoostClassifier 클래스
- 일반적으로 GBM이 랜덤 포레스트보다는 예측 성능이 조금 뛰어난 경우가 많음

↔ 긴 수행 시간, 하이퍼 파라미터 튜닝 노력 필요

GBM 하이퍼 파라미터 소개

loss	경사 하강법에서 사용할 비용 함수
learning_rate	GBM이 학습을 진행할 때마다 적용하는 학습률
n_estimators	weak learner의 개수

subsample	weak learner가 학습에 사용하는 데이터의 샘플링 비율

- GBM은 과적합에도 뛰어난 예측 성능을 가진 알고리즘이지만, 수행시간이 오래 걸린다는 단점 존재

06. XGBoost(eXtra Gradient Boost)

XGBoost

- GBM에 기반하고 있지만, 느린 수행 시간과 과적합 규제 부재를 해결함
- 뛰어난 예측 성능, GBM 대비 빠른 수행 시간, 과적합 규제 (regularization), tree pruning, 자체 내장된 교차 검증, 결손값 자체 처리 등의 장점
- XGboost 설치

```
pip install xgboost = 1.5.0
```

- 파이썬 래퍼 XGBoost 모듈과 사이킷런 래퍼 XGBoost 모듈의 하이퍼 파라미터는 약간 다름

파이썬 래퍼 XGBoost 하이퍼 파라미터

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, reg나arization 등과 관련 파라미터 등을 지칭
학습 태스크 파라미터	학습 수행 시의 객체 함수, 평가를 위한 지표 등을 설정하는 파라미터

주요 부스터 파라미터 : eta, num_boost_rounds, min_child_weight, gamma , max_depth, sub_sample, colsample_bytree, lambda, alpha, scale_pos_weight 등

학습 태스크 파라미터 : objective, binary:logistic, multi:softmax, multi:softprob, eval_metric (rmse, mae, logloss, error, merror, mlogloss, auc)

- 뛰어난 알고리즘일수록 파라미터를 튜닝할 필요가 적음
- 과적합 문제가 심각하다면 : eta 값을 낮추거나, max_depth 값을 낮추거나, min_child_weight 값을 높이거나, gamma 값을 높이거나, sub_sample과 colsample_bytree를 조정한다.
- XGBoost : 교차 검증, 성능 평가, 피처 중요도 등 시가고하 기능

- 조기 중단 (Early stopping) : 조기 중단 기능이 있어서 n_estimators에 지정한 부스팅 반복 횟수에 도달하지 않더라도 예측 오류가 더 이상 개선되지 않으면 반복을 끝까지 수행하지 않고 중지해 수행 시간을 개선

파이썬 래퍼 XGBoost 적용 - 위스콘신 유방암 예측

- XGBoost만의 전용 데이터 객체인 DMatrix를 사용

DataFrame 기반 학습 데이터 세트, 테스트 데이터 세트 → DMatrix 변환 예제

```
#만약 구버전 XGBoost에서 DataFrame으로 DMatrix 생성이 안될경우 X_train.values
#학습, 검증, 테스트용 DMatrix를 생성
dtr = xgb.DMatrix(data= X_tr, label = y_tr)
dval = vgb.DMatrix(data = X_val, label = y_val)
dtest = cgb.DMatrix(data = X_test, label = y_test)
```

- 더 이상 지표 개선이 없을 경우에 num_boost_round 횟수를 채우지 않고 중간에 반복을 빠져나올 수 있도록
- 조기 중단의 성능 평가 : 별도의 검증 데이터 세트
- xgboost의 train() 함수에 early_stopping_rounds 파라미터를 입력하여 설정, eval_metric과 평가용 데이터 세트 지정 필요
- 평가용 데이터 세트 설정 : 학습용 DMatrix인 dtr과 검증용 DMatrix인 dval로 설정한 뒤 train() 함수의 evals 인자값으로 입력
- xgboost 를 이용해 모델 학습 완료 → 테스트 데이터 세트에 예측 수행
- train() 함수, predict() 메서드 이용
- xgboost 의 predict()는 예측 결과를 추정할 수 있는 확률 값을 반환
- xgboost 패키지에 내장된 시각화 기능 : plot_importance() API - 피처의 중요도를 막대그래프 형식으로 나타냄
- plot_importance() 사용 시 xgboost를 넘파이 기반의 피처 데이터로 학습 시에는 넘파이에서 피처명을 제대로 알 수 없으므로 Y축 피처명을 나열 시 f0,f1과 같이 피처 순서별로 f자 뒤에 순서를 붙여서 피처명을 나타냄
- xgboost 모듈의 to_graphviz() API를 이용하면 규칙 트리 구조 그릴 수 있음

```
 xgboost.cv(params, dtrain, num_boost_round=10, nfold=3, stratified=False, fo
obj=None, feval=None, maximize=False, early_stopping_rounds=None, fprepr
verbose_eval=None, show_stdv=True, seed=0, callbacks=None, shuffle=True)
```

사이킷런 래퍼 XGBoost의 개요 및 적용

- 분류를 위한 래퍼 클래스인 XGBClassifier, 회귀를 위한 래퍼 클래스인 XGBRegressor
- xgboost의 n_estimators와 num_boost_round 하이퍼 파라미터 동일
- 사이킷런 래퍼 XGBoost가 더 좋은 평가 결과 나옴 (위스콘신 데이터 세트의 개수가 작아서)
- 조기 중단 : fit()에 관련 파라미터 입력 - early_stopping_rounds, eval_metric, eval_set
-

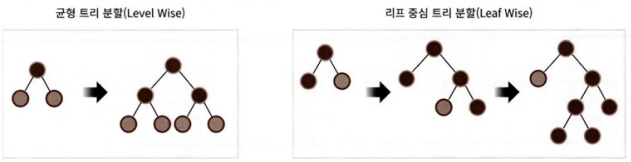
```
from xgboost import XGBClassifier

xgb_wrapper = XGBClassifier(n_estimators = 400, learning_rate = 0.05, max_d
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]
```

→ 400번 반복 X, 176번째 반복에서 학습을 마무

- 조기중단값을 너무 줄이면 예측 성능이 저하될 우려가 있음
- plot_importance() API에 사이킷런 래퍼 클래스를 입력해도 파이썬 래퍼 클래스를 입력한 결과와 똑같이 시각화 결과를 도출

07. LightGBM

- LightGBM은 XGBoost보다 학습에 걸리는 시간이 훨씬 적고, 메모리 사용량도 적음
- 일반 GBM 계열의 트리 분할 방법과 다르게 리프 중심 트리 분할(Leaf Wise) 방식을 사용 → 최대한 균형 잡힌 트리를 유지하면서 분할하기 때문에 트리의 깊이가 최소화, overfitting에 강함



- 패키지명 : 'lightgbm'
- 사이킷런 래퍼 LightGBM 클래스는 분류를 위한 LGBMClassifier 클래스와 회귀를 위한 LGBMRegressor 클래스

LightGBM 설치

```
pip install lightgbm == 3.3.2
import lightgbm
from lightgbm import LGBMClassifier
```

LightGBM 하이퍼 파라미터

- 주요 파라미터

num_iterations	반복 수행하려는 트리의 개수를 지정
learning_rate	0에서 1 사이의 값을 지정하며 부스팅 스텝을 반복적으로 수행할 때 업데이트되는 학습률 값
max_depth	트리 기반 알고리즘의 max_depth과 동일
min_data_in_leaf	최종 결정 클래스 인 리프 노드가 되기 위해서 최소한으로 필요한 레코드 수이며, 과적합을 제어하기 위한 파라미터
num_leaves	하나의 트리가 가질 수 있는 최대 리프 개수
boosting	부스팅의 트리를 생성하는 알고리즘을 기술
bagging_fraction	트리가 커져서 과적합되는 것을 제어하기 위해서 데이터를 샘플링하는 비율을 지정
featurejraction	개별 트리를 학습할 때마다 무작위로 선택하는 피처의 비율
lambda_l2	L2 regulation 제어를 위한 값
lambda_l1	L1 regulation 제어를 위한 값

하이퍼 파라미터 튜닝 방안

- num_leaves의 개수를 중심으로min_child_samples(min_data_in_leaf), max_depth를 함께 조정하면서 모델의 복잡도를 줄이는 것
- learning_rate를 작게 하면서 n_estimators를 크게 하는 것

파이썬 래퍼 LightGBM과 사이킷런 래퍼 XGBoost, LightGBM 하이퍼 파라미터 비교

- 사이킷런 래퍼 LightGBM 클래스와 사이킷런 래퍼 XGBoost 클래스는 많은 하이퍼 파라미터가 동일

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

LightGBM 적용 - 위스콘신 유방암 예측

```
#LightGBM의 파이썬 패키지인 lightbgm에서 LGBMClassifier 임포트
from lightbgm 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_na
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 = 

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

#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]
```

- 피처 중요도 시각화 API : plot_importance()

08. 베이지안 최적화 기반의 HyperOpt를 이용한 하이퍼 파라미터 튜닝

- 그동안 GridSearch를 적용했으나, 튜닝해야 할 하이퍼 파라미터 개수가 많을 경우 최적화 수행 시간이 오래 걸린다는 단점

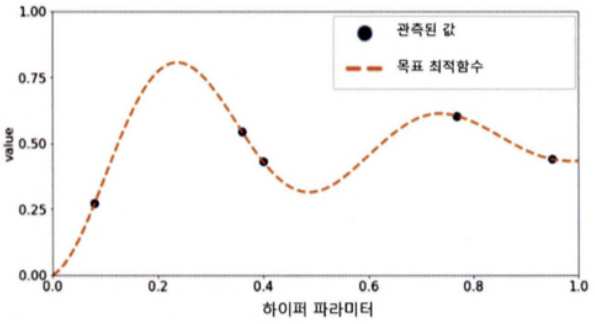
베이지안 최적화 개요

- 베이지안 최적화란?

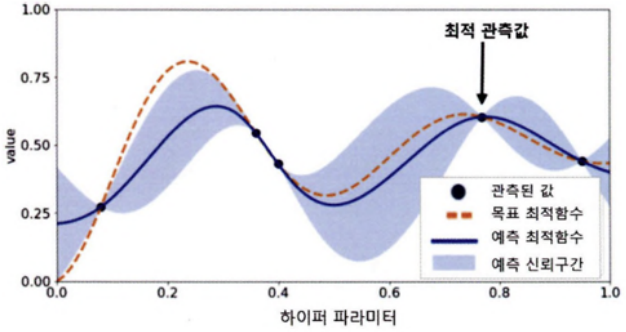
→ 목적 함수 식을 제대로 알 수 없는 블랙 박스 형태의 함수에서 최대 또는 최소 함수 반환 값을 만드는 최적 입력값을 가능한 적은 시도를 통해 빠르고 효과적으로 찾아주는 방식

- 베이지안 확률에 기반하여 베이지안 최적화는 새로운 데이터를 입력받았을 때 최적 함수를 예측하는 사후 모델을 개선해 나가면서 최적 함수 모델을 만들어 냄
- 대체 모델 (Surrogate Model) : 획득 함수로부터 최적 함수를 예측할 수 있는 입력값을 추천 받은 뒤 이를 기반으로 최적 함수 모델을 개선
- 획득 함수 : 개선된 대체 모델을 기반으
  로 최적 입력값을 계산
- 대체 모델은 획득 함수가 계산한 하이퍼 파라미터를 입력받으면서 점차적으로 개선되며, 개선된 대체 모델을 기반으로 획득 함수는 더 정확한 하이퍼 파라미터를
  계산할 수 있게 됨
- 베이지안 최적화 단계

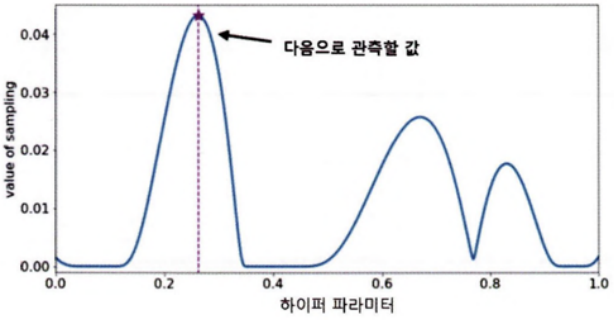
step 1. 최초에는 랜덤하게 하이퍼 파라미터들을 샘플링하고 성능 결과를 관측



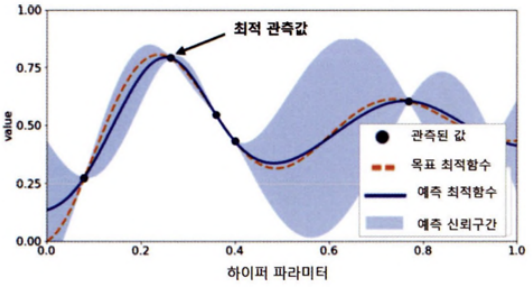
step 2. 관측된 값을 기반으로 대체 모델은 최적 함수를 추정



step 3. 추정된 최적 함수를 기반으로 획득 함수(Acquisition Function)는 다음으로 관측할 하이퍼 파라미터 값을 계산



step 4. 획득 함수로부터 전달된 하이퍼 파라미터를 수행하여 관측된 값을 기반으로 대체 모델은 갱신되어 다시 최적 함수를 예측 추정함



→ step 3, step 4를 특정 횟수만큼 반복하면 대체 모델의 불확실성이 개선되고, 점차 정확한 최적 함수 추정이 가능하게 됨

Hyper Opt 사용하기

- HyperOpt : 입력 변수명과 입력 값의 검색 공간 설정, 목적 함수 설정, 목적 함수의 반환 최솟값을 가지는 최적 입력값을 유추
- 입력 변수명과 입력값 검색 공간은 파이썬 딕셔너리 형태로 설정
- 입력값 검색공간 제공 함수들 : hp.quniform(label, low, high, q), hp.uniform(label, low, high), hp.randint(label, upper), hp.loguniform(label, low, high), hp.choice(label, options)
- 목적 함수 생성 : 변숫값과 검색 공간을 가지는 딕셔너리를 인자로 받고, 특정 값을 반환하는 구조
- fmin(objective, space, algo, max_evals, trials) 함수 : 목적 함수의 반환값이 최소가 될 수 있는 최적의 입력값을 베이지안 최적화 기법에 기반하여 찾아줌
- HyperOpt 의 버전이 달라지면 아래 실행 결과도 달라질 수 있음
- Trials 객체의 중요 속성 : results, vals
- result: 함수 반복 수행 시마다 반환되는 반환값

→ {'loss' : 함수 반환값, 'status' : 반환 상태값}

- vals : 함수의 반복 수행 시마다 입력되는 입력 변숫값 , 딕셔너리 형태로 값을 가짐

→ {'입력변수명' : 개별 수행 시마다 입력된 값의 리스트}

- Trials의 result와 vals 속성값들을 데이터프레임으로 만들어서 확인할 수 있

HyperOpt를 이용한 XGBoost 하이퍼 파라미터 최적화 (p.280)

- HyperOpt 이용하여 하이퍼 파라미터 최적화 : 적용해야 할 하이퍼 파라미터와 검색 공간을 설정, 목적 함수에서 XGBoost 학습 후에 예측 성능 결과를 반환 값으로 설정
- 위스콘신 유방암 데이터 세트

1. 검색 공간에서 목적 함수로 입력되는 모든 인자들은 실수형 값이므로 이를 XGBoostClassifier 의 정수형 하이퍼 파라미터 값으로 설정할 때는 정수형으로 형변환 필요
2. HyperOpt의 목적 함수는 최솟값을 반환할 수 있도록 최적화해야 함 → 값이 클수록 좋은 성능 지표일 경우 -1을 곱한 뒤 반환하여, 더 큰 성능 지표가 더 작은 반환값이 되도록 만들어줘야 함

- 목적함수 objective_func()

- 최적 하이퍼 파라미터들을 이용해서 재학습

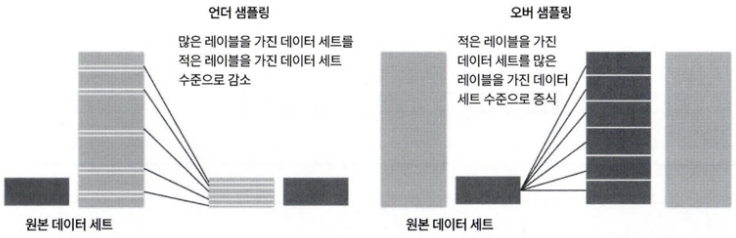
10. 분류 실습 - 캐글 신용카드 사기 검출

언더 샘플링과 오버 샘플링의 이해

- 이상 레이블을 가지는 데이터 건수는 매우 적음 → 다양한 유형을 제대로 학습 X
- 정상 레이블을 가지는 데이터 건수는 매우 많음 → 일방적으로 정상 레이블로 치우친 학습 수행

→ 제대로 된 이상 데이터 검출이 어려워지기 쉬움

↔ 적절한 학습 데이터 확보 필요 : Oversampling, Undersampling



- 언더 샘플링 : 많은 데이터 세트를 적은 데이터 세트 수준으로 감소시키는 방법

→ 정상 레이블 데이터를 이상 레이블 데이터 수준으로 줄여버린 상태에서 학습을 수행하면 과도하게 정상 레이블로 학습/예측하는 부작용을 개선할 수 있지만, 너무 많은 정상 레이블 데이터를 감소시켜 정상 레이블의 경우 제대로 된 학습을 수행할 수 없는 문제가 발생할 수 있음

- 오버 샘플링 : 이상 데이터와 같이 적은 데이터 세트를 증식하여 학습을 위한 충분한 데이터를 확보하는 방법 (SMOTE : 적은 데이터 세트에 있는 개별 데이터들의 K 최근접 이웃을 찾기 → 이 데이터와 K개 이웃들의 차이를 일정 값으로 만들어서 기존 데이터와 약간 차이가 나는 새로운 데이터들을 생성)

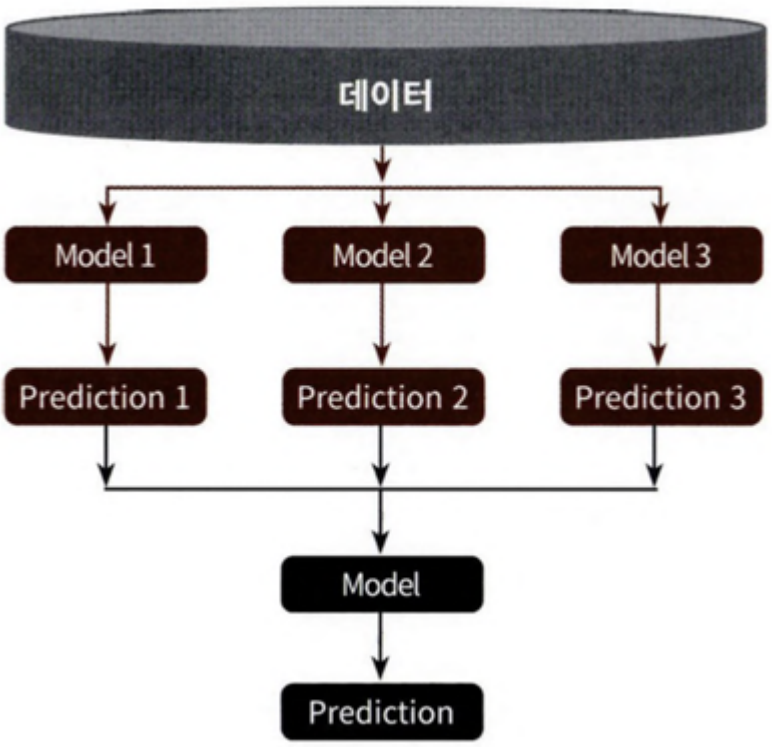
11. 스태킹 앙상블

- 스태킹 (Stacking) : 개별적인 여러 알고리즘을 서로 결합해 예측 결과를 도출, 개별 알고리즘으로 예측한 데이터를 기반으로 다시 예측 수행

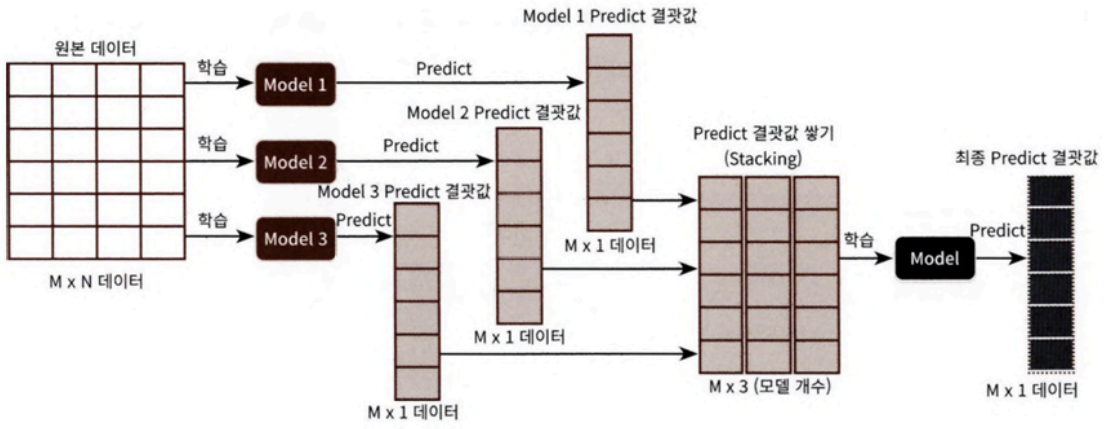
→ 개별 알고리즘의 예측 결과 데이터 세트를 최종적인 메타 데이터 세트로 만들어 별도의 ML 알고리즘으로 최종 학습을 수행하고, 테스트 데이터를 기반으로 다시 최종 예측을 수행하는 방식

2가지 모델 필요 : 1) 개별적인 기반 모델 2) 최종 메타 모델

→ 여러 개별 모델의 예측 데이터를 각각 스태킹 형태로 결합해 최종 메타 모델의 학습용 피처 데이터 세트와 테스트용 피처 데이터 세트를 만드는 것



- 모델별로 각각 학습을 시킨 뒤 예측을 수행 → 모델별로 도출된 예측 레이블 값을 다시 합해서 새로운 데이터 세트를 만듦 → 스태킹된 데이터 세트에 최종 모델을 적용해 최종 예측을 하는 것



기본 스태킹 모델

- 위스콘신 암 데이터 세트
- 스태킹에 사용될 머신러닝 알고리즘 클래스 생성 : KNN, 랜덤 포레스트, 결정 트리, 에이다 부스트, 로지스틱 회귀
- 개별 알고리즘으로부터 예측된 예측값을 칼럼 레벨로 옆으로 붙여서 피처 값으로 만들어, 최종 메타 모델인 로지스틱 회귀에서 학습 데이터로 다시 사용

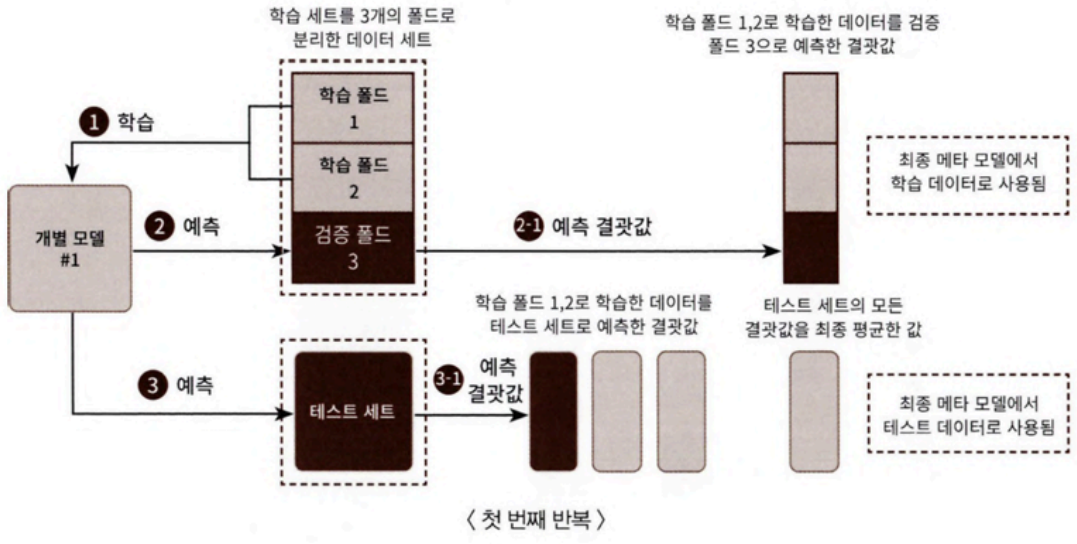
CV 세트 기반의 스태킹

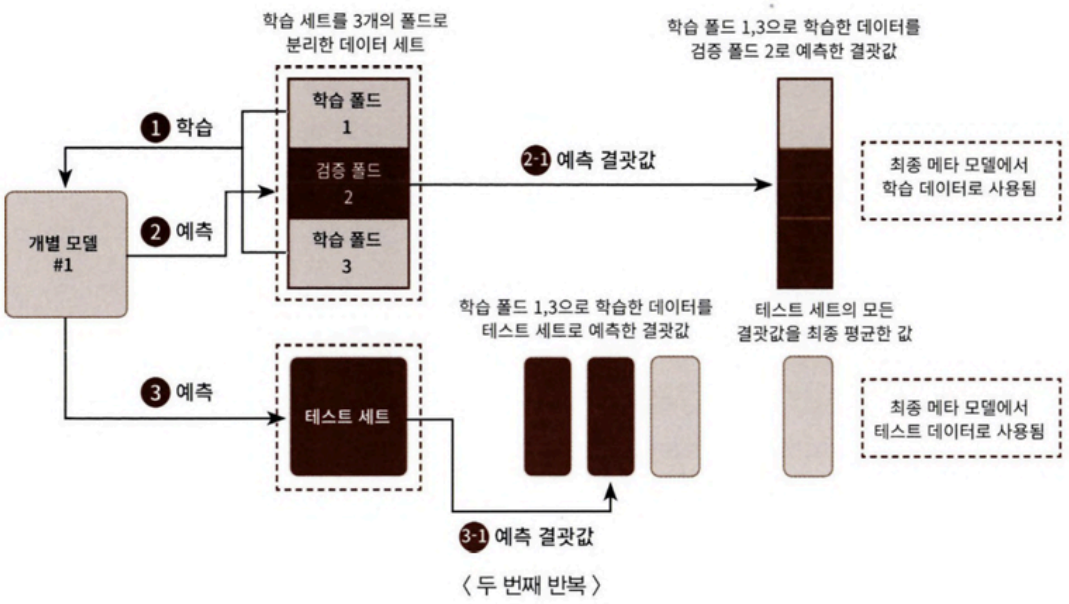
- 과적합을 개선하기 우해 최종 메타 모델을 위한 데이터 세트를 만들 때 교차 검증 기반으로 예측된 결과 데이터 세트를 이용

1. step1)각 모델별로 원본 학습/테스트 데이터를 예측한 결과 값을 기반으로 메타 모델을 위한 학습용/테스트용 데이터를 생성
2. step2) 스텝 1에서 개별 모델들이 생성한 학습용 데이터를 모두 스태킹 형태로 합쳐서 메타 모델이 학습할 최종 학습용 데이터 세트를 생성. → 모두 합쳐서 최종 테스트 데이터 세트 생성 → 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 평가

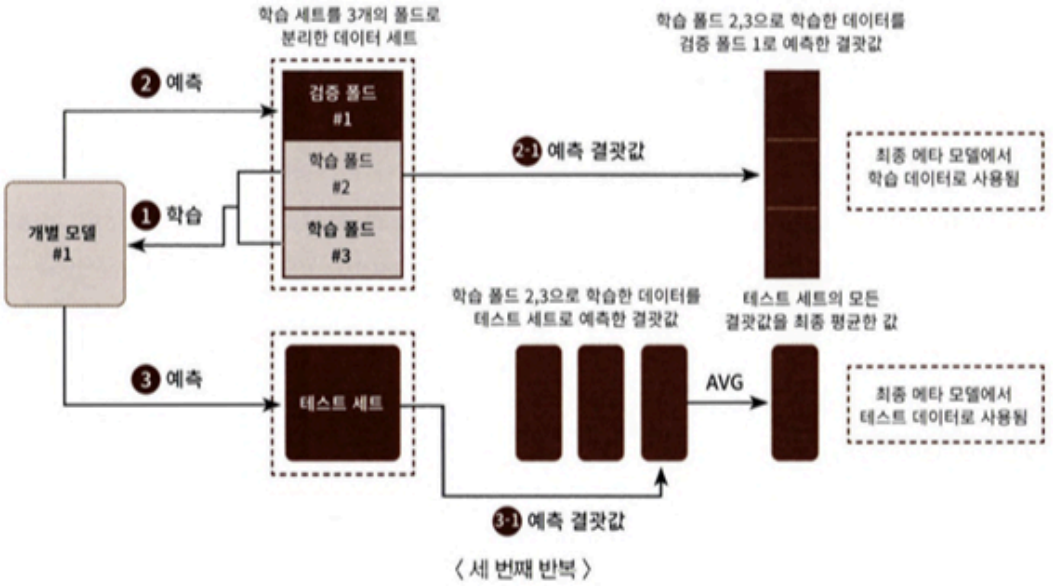
step 1)

- 학습용 데이터를 N개의 폴드로 나눔, 유사한 반복 작업 수행 → 마지막 반복에서 개별 모델의 예측값으로 학습 데이터와 테스트 데이터 생성



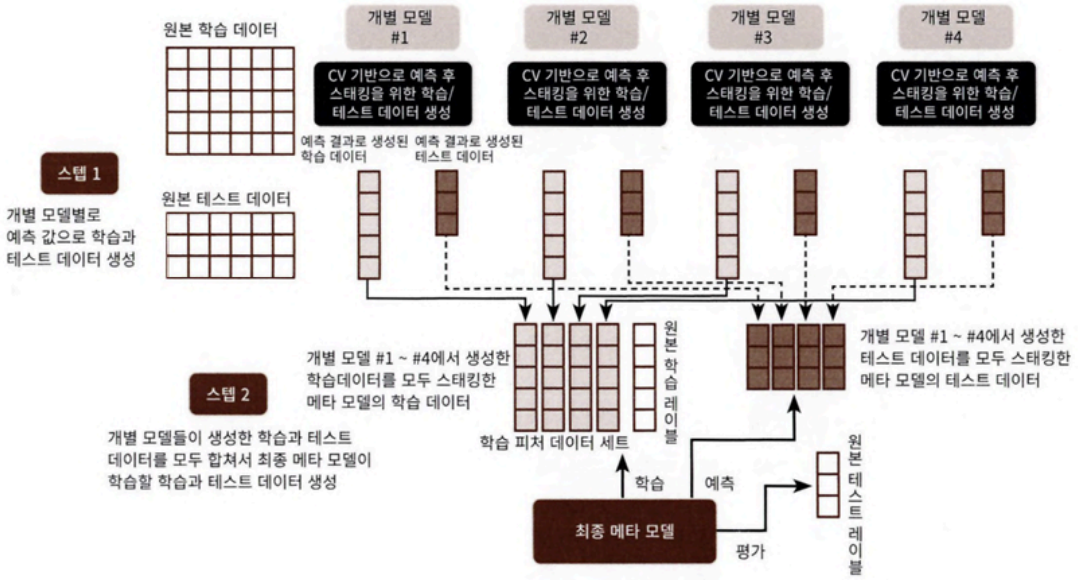


→ 폴드 내 학습용 데이터 세트를 변경하고 첫번째 그림과 동일한 작업을 수행



→ 스태킹 데이터 생성

step2) 각 모델들이 스텝 1로 생성한 학습과 테스트 데이터를 모두 합쳐서 최종적으로 메타 모델이 사용할 학습 데이터와 테스트 데이터를 생성


원본 학습 데이터
개별 모델 #1
개별 모델 #2
개별 모델 #3
개별 모델 #4
CV 기반으로 예측 후 스태킹을 위한 학습/테스트 데이터 생성
CV 기반으로 예측 후 스태킹을 위한 학습/테스트 데이터 생성
CV 기반으로 예측 후 스태킹을 위한 학습/테스트 데이터 생성
CV 기반으로 예측 후 스태킹을 위한 학습/테스트 데이터 생성
예측 결과로 생성된 학습 데이터
예측 결과로 생성된 테스트 데이터
스텝 1
개별 모델별로 예측 값으로 학습과 테스트 데이터 생성
원본 테스트 데이터
개별 모델 #1 ~ #4에서 생성한 학습데이터를 모두 스태킹한 메타 모델의 학습 데이터
원본 학습 레이블
개별 모델 #1 ~ #4에서 생성한 테스트 데이터를 모두 스태킹한 메타 모델의 테스트 데이터
스텝 2
개별 모델들이 생성한 학습과 테스트 데이터를 모두 합쳐서 최종 메타 모델이 학습할 학습과 테스트 데이터 생성
학습 피처 데이터 세트
학습
예측
원본 테스트 레이블
최종 메타 모델
평가