파이썬 머신러닝 완벽 가이드 - 분류 (2)

📌 목차

📌 07. LightGBM

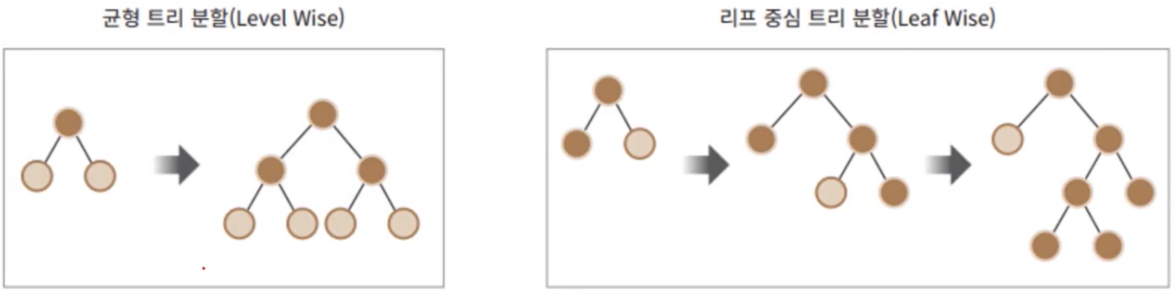
LightGBM이란?

- XGBoost와 함께 가장 각광 받는 부스팅 계열 알고리즘

LightGBM 장점 및 특징은?

- (장점) XGBoost와 예측 성능에 차이를 보이지 않지만, 학습 시간이 훨씬 적음
- (장점) 메모리 사용량도 상대적으로 적음
- (특징) 리프 중심 트리 분할(Leaf Wise) 방식을 사용함
    - 기존의 대부분 트리 기반 알고리즘은 트리의 깊이를 효과적으로 줄이기 위해 균형 트리 분할(Level Wise) 방식을 사용함. 즉, 최대한 균형 잡힌 트리를 유지하면서 분할하기 때문에 트리의 깊이가 최소화될 수 있음
    - 장점 : 오버피팅에 보다 더 강한 구조를 가질 수 있음
    - 단점 : 균형을 맞추기 위한 시간이 오래 걸림

- LightGBM에서는 최대 손실 값(max delta loss)을 가지고 리프 노드를 지속적으로 분할하면서 트리의 깊이가 깊어지고 비대칭적인 규칙 트리가 생김
- 장점 : 최대 손실값을 가지는 리프 노드를 지속적으로 분할해 생성된 규칙 트리는 학습을 반복할수록 결국 균형 트리 방식보다 예측 오류 손실을 최소화할 수 있음



LightGBM 주요 하이퍼 파라미터

- 튜닝 방안 : num leaves 개수를 중심으로 min_data_in_leaf, max_depth 함께 조정하며 모델의 복잡도를 줄이는 것이 기본 방안
- num leaves [default = 31]
  - 하나의 트리가 가질 수 있는 최대 리프 개수
  - LightGBM 모델의 복잡도를 제어하는 주요 파라미터로, num_leaves 개수를 높이면 정확도가 높아지지만, 반대로 트리의 깊이가 깊어지고 모델이 복잡도가 커져 과적합 영향도가 커짐
- min_data_in_leaf [default = 20]
  - 최종 결정 클래스인 리프 노드가 되기 위해서 최소한으로 필요한 레코드 수이며, 과적합을 제어하기 위한 파라미터
  - 과적합 개선하기 위해 중요한 파라미터로, 보통 큰 값을 설정하면 트리가 깊어지는 것을 방지함
- max_depth [default = 1]
  - 깊이의 크기를 제안함

파이썬 래퍼 LightGBM과 사이킷런 래퍼 XGBoost, LightGBM 하이퍼 파라미터 비교

- 기본적으로 사이킷런 래퍼 LightGBM은 사이킷런 래퍼 XGBoost 파라미터명을 따라가되, 없을 경우 파이썬 래퍼 LightGBM 파라미터명을 참조

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

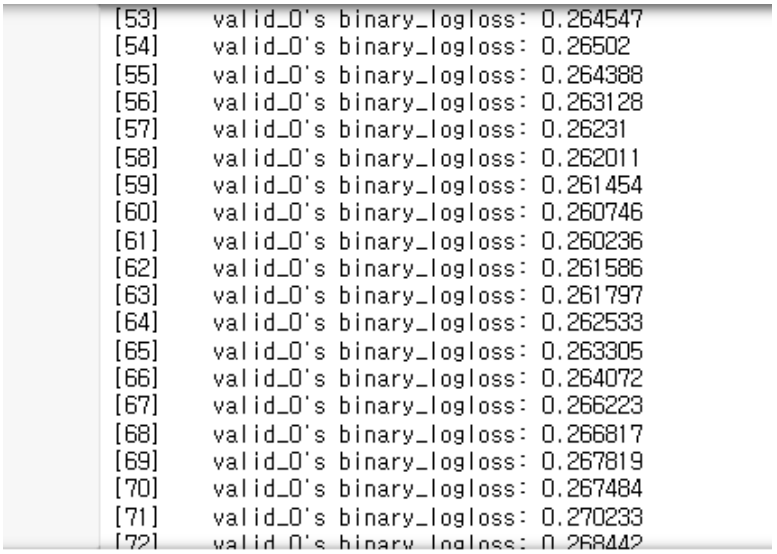
```
# LightGBM의 파이썬 패키지인 lightgbm에서 LGBMClassifier 임포트
from lightgbm import LGBMClassifier

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.f
cancer_df['target']= dataset.target
X_features = cancer_df.iloc[:, :-1]
y_label = cancer_df.iloc[:, -1]

cancer_df.head()
```

```
#X_features.head()
#y_label.head()
```

	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	com
0	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	0.07871	...	17.33	184.60	2019.0	0.1622	
1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	0.05667	...	23.41	158.80	1956.0	0.1238	
2	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	0.05999	...	25.53	152.50	1709.0	0.1444	
3	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.2597	0.09744	...	26.50	98.87	567.7	0.2098	
4	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0.1809	0.05883	...	16.67	152.20	1575.0	0.1374	

```
# 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,

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

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

# LightGBM에서 조기 중단을 위해 valid_sets와 early_stopping_rounds
evals = [(X_tr, y_tr), (X_val, y_val)]
lgbm_wrapper.fit(X_tr, y_tr, eval_set=evals, early_stopping_r
preds = lgbm_wrapper.predict(X_test)
pred_proba = lgbm_wrapper.predict_proba(X_test)[:, 1]
```

```
[101]   valid_0's binary_logloss: 0.276695
[102]   valid_0's binary_logloss: 0.278488
[103]   valid_0's binary_logloss: 0.278932
[104]   valid_0's binary_logloss: 0.280997
[105]   valid_0's binary_logloss: 0.281454
[106]   valid_0's binary_logloss: 0.282058
[107]   valid_0's binary_logloss: 0.279275
[108]   valid_0's binary_logloss: 0.281427
[109]   valid_0's binary_logloss: 0.280752
[110]   valid_0's binary_logloss: 0.282152
[111]   valid_0's binary_logloss: 0.280894
C:₩ProgramData₩anaconda3₩Lib₩site-packages₩lightgbm₩sklearn.py:726: UserWarning: 'early_stopping_rounds' argument is deprecated and will be removed in a future release of LightGBM. Pass 'early_stopping()' callback via 'callbacks' argument instead.
  _log_warning("'early_stopping_rounds' argument is deprecated and will be removed in a future release of LightGBM. "
C:₩ProgramData₩anaconda3₩Lib₩site-packages₩lightgbm₩sklearn.py:736: UserWarning: 'verbose' argument is deprecated and will be removed in a future release of LightGBM. Pass 'log_evaluation()' callback via 'callbacks' argument instead.
  _log_warning("'verbose' argument is deprecated and will be removed in a future release of LightGBM. "
```

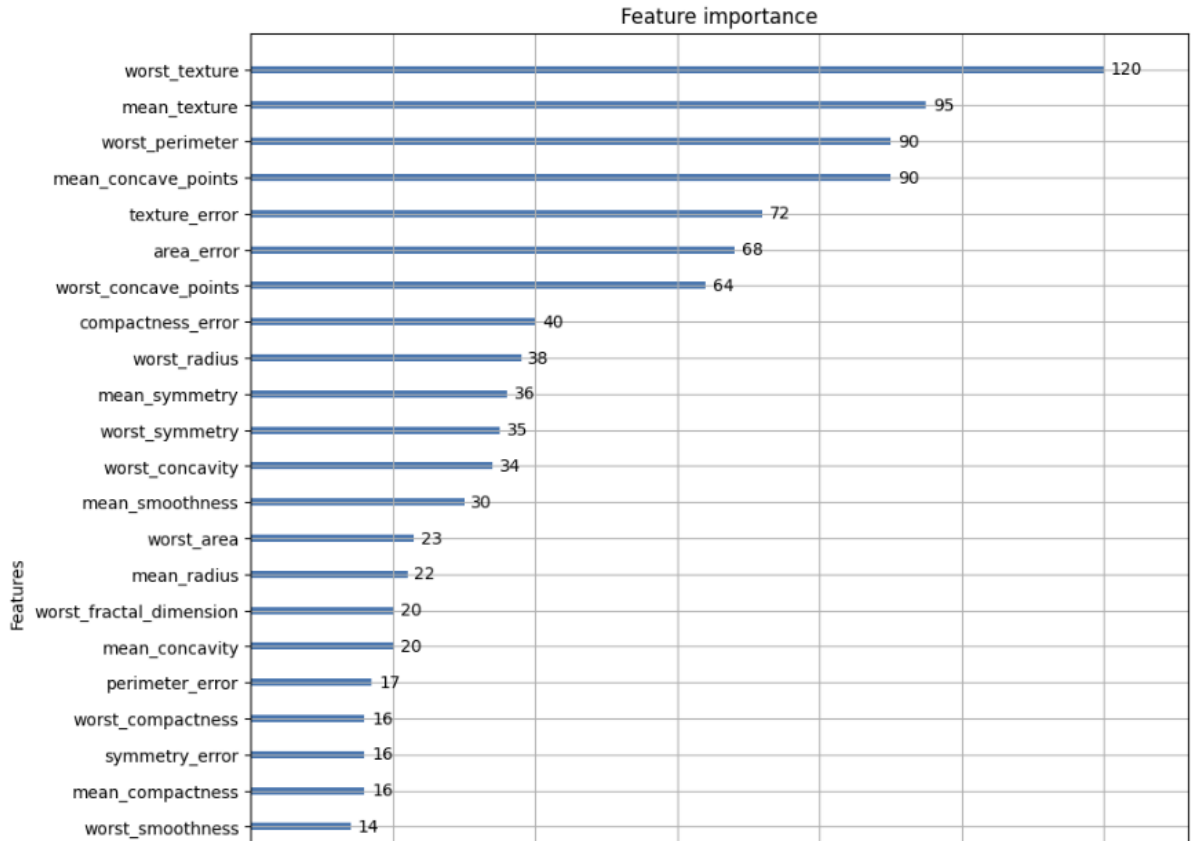


```
[53]    valid_0's binary_logloss: 0.264547
[54]    valid_0's binary_logloss: 0.26502
[55]    valid_0's binary_logloss: 0.264388
[56]    valid_0's binary_logloss: 0.263128
[57]    valid_0's binary_logloss: 0.26231
[58]    valid_0's binary_logloss: 0.262011
[59]    valid_0's binary_logloss: 0.261454
[60]    valid_0's binary_logloss: 0.260746
[61]    valid_0's binary_logloss: 0.260236
[62]    valid_0's binary_logloss: 0.261586
[63]    valid_0's binary_logloss: 0.261797
[64]    valid_0's binary_logloss: 0.262533
[65]    valid_0's binary_logloss: 0.263305
[66]    valid_0's binary_logloss: 0.264072
[67]    valid_0's binary_logloss: 0.266223
[68]    valid_0's binary_logloss: 0.266817
[69]    valid_0's binary_logloss: 0.267819
[70]    valid_0's binary_logloss: 0.267484
[71]    valid_0's binary_logloss: 0.270233
[72]    valid_0's binary_logloss: 0.268442
```

```
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, rec
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
```

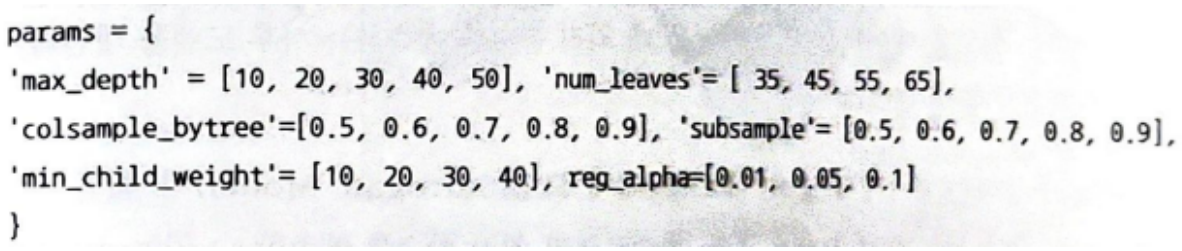
```
# plot_importance( )를 이용하여 feature 중요도 시각화
from lightgbm import plot_importance
import matplotlib.pyplot as plt
%matplotlib inline

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



📌 08. 베이지안 최적화 기반의 HyperOpt를 이용한 하이퍼 파라미터 튜닝

Grid Search 단점

- 지금까지 하이퍼 파라미터 튜닝을 위해 사이킷런에서 제공하는 'Grid Search' 방식을 적용함
- Grid Search 주요 단점은 튜닝해야 할 하이퍼 파라미터 개수가 많을 경우 최적화 수행 기간이 오래 걸림
- 개별 하이퍼 파라미터 값의 범위가 넓거나 학습 데이터가 대용량일 경우 최적화 시간이 더욱 늘어나게 됨
- 예를 들어, 아래의 경우 (5 x 4 × 5 × 5 × 4 × 3) = 6,000 회에 걸쳐서 반복적으로 학습과 평가를 수행해야 하므로 수행 시간이 오래 걸릴 수 밖에 없음



```
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]
}
```

- 캐글에서는 정해진 시간 안에 조금이라도 모델 성능을 향상시켜야 Grid Search는 수행 시간이 오래 걸려 시간적 제약이 있을 수 밖에 없음 → 실무의 대용량 학습 데이터에 XGboost, Light GBM 수행 시 사용하는 '베이지안 최적화 기법' 설명

베이지안 최적화란?

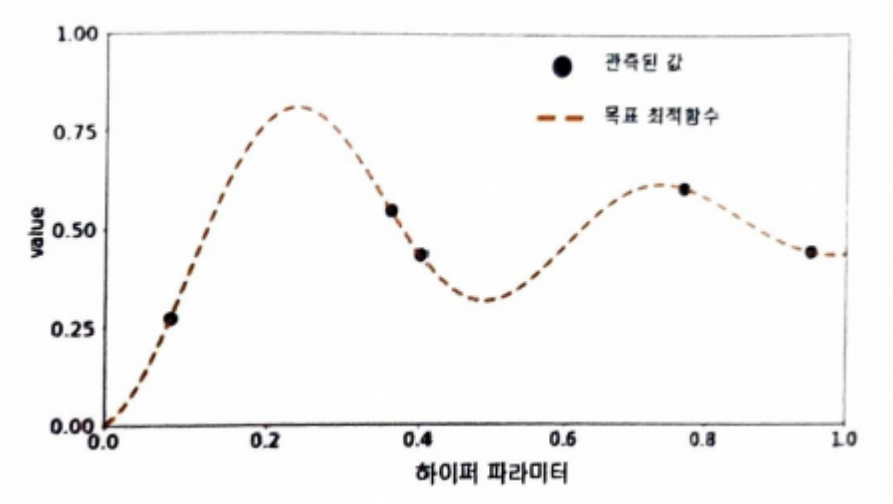
- **(1) 목적 함수 식을 제대로 알 수 없는** 블랙 박스 형태의 함수에서 최대 또는 최소 함수 반환 값을 만드는 **(2)최적 입력값을** **(3)가능한 적은 시도를 통해** 빠르고 효과적으로 찾아주는 방식

베이지안 최적화를 구성하는 두 가지 중요 요소

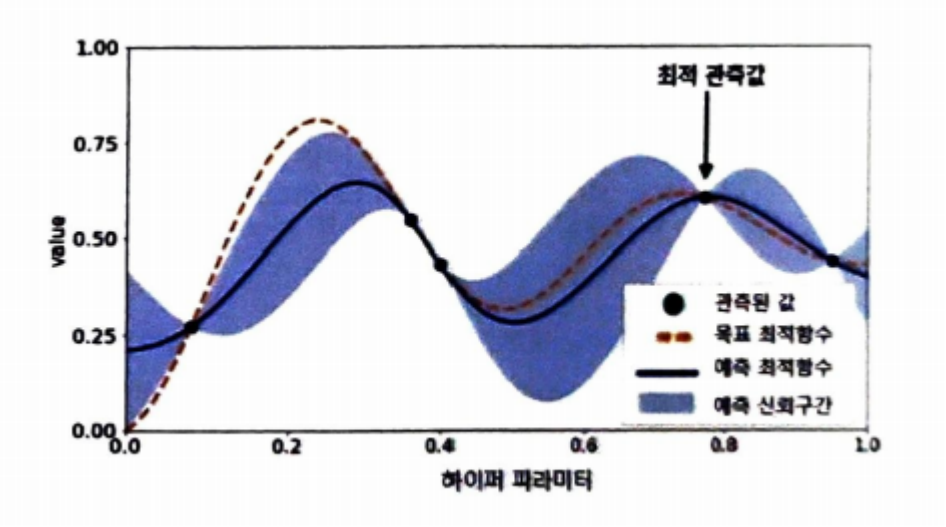
- 대체 모델 (Surrogate Model)
- 획득 함수 (Acquisition Function)

베이지안 최적화의 3단계

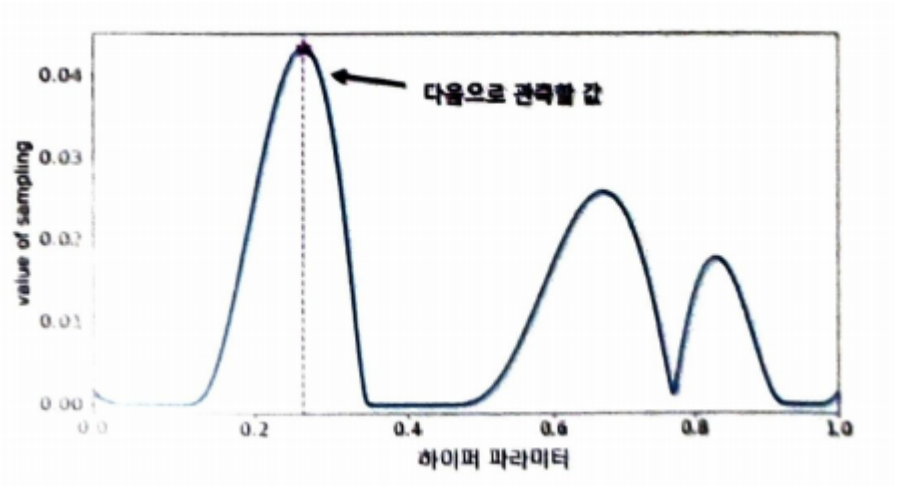
- Step1 : 최초에는 랜덤하게 하이퍼 파라미터를 샘플링하고 성능 결과를 관측함. 아래 그림에서 검은색 원은 특정 하이퍼 파라미터가 입력되었을 때 관측된 성능 지표 결괏값을 뜻하며 주황색 사선은 찾아야 할 목표 최적함수임



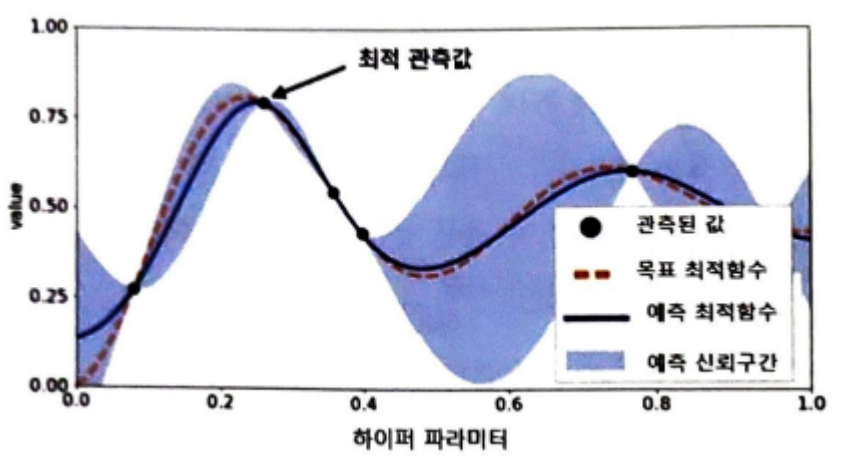
- Step2 : 관측된 값을 기반으로 대체 모델은 최적 함수를 추정함. 아래 그림에서 파란색 실선은 대체 모델이 추정한 최적 함수임. 옅은 파란색으로 되어 있는 영역은 예측된 함수의 신뢰 구간임. 추정된 함수의 결괏값 오류 편차를 의미하며 추적 함수의 불확실성을 나타냄. 최적 관측값은 y축 value에서 가장 높은 값을 가질 때의 하이퍼 파라미터임.



- Step3: 추정된 최적 함수를 기반으로 획득 함수(Acquisition Function)는 다음으로 관측할 하이퍼 파라미터 값을 계산함. 획득 함수는 이전의 최적 관측값보다 더 큰 최댓값을 가질 가능성이 높은 지점을 찾아서 다음에 관측할 하이퍼 파라미터를 대체 모델에 전달함



- Step4: 획득 함수로부터 전달된 하이퍼 파라미터를 수행하여 관측된 값을 기반으로 대체 모델은 갱신되어 다시 최적 함수를 예측 추정함



HyperOpt란?

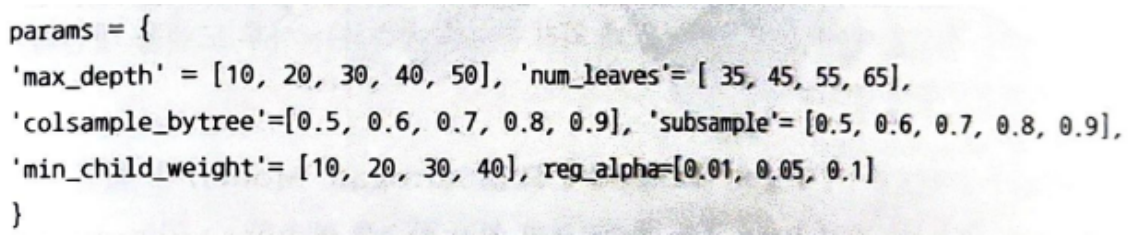
- 베이지안 최적화를 머신러닝 모델의 하이퍼 파라미터 튜닝에 적용할 수 있게 제공되는 파이썬 패키지 중 하나
- 대표적으로는 HyperOpt, Basyesian Optimization, Optuna 등이 있음
- HyperOpt는 목적 함수의 반환 최대값이 아닌 최소값을 유추함 (특징)

HyperOpt 사용법

- (1) Search Space 설정 (2) 목표 함수 설정 (3) 목적 함수의 반환 최소값을 가지는 최적 입력값 유추

- 검색 공간(Search Space) 설정
  - 입력 변수명과 입력값의 검색 공간 설정할 때 딕셔녀리 형태로 설정하며, 키(key)값으로 입력 변수명, 밸류(value) 값으로 해당 입력 변수의 검색 공간이 주어짐
  - hp.quniform(label, low, high, q) : label로 지정된 입력값 변수 검색 공간을 최솟값 low에서 최댓값 high까지 q의 간격을 가지고 설정

```
from hyperopt import hp

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



- 목적 함수 생성
  - 목적 함수는 반드시 변숫값과 검색 공간을 가지는 딕셔너리를 인자로 받고, 특정 값을 반환하는 구조로 만들어져야 함
  - 아래 예제는 search_space로 지정된 딕셔너리에서 x 입력 변숫값과 y 입력 변숫값을 추출하여 retval = x**2 - 20*y로 계산된 값을 반환함

```
from hyperopt import STATUS_OK

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

    return retval
```

- 목적 함수의 반환 최소값을 가지는 최적 입력값 유추
    - fmin(objective, space, algo, max_evals, trials) 함수를 통해 최소값을 찾음
    - 위에서 설정한 searh_space, objective func을 넣고 max_evals 설정함

```
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(ma
best_01 = fmin(fn=objective_func, space=search_space, algo
               , trials=trial_val, rstate=np.random.defaul
print('best:', best_01)
```

```
100%|██████████████████████████████████████████████████| 5/5 [00:00<00:00, 624.99trial/s, best loss: -224.0]
best: {'x': -4.0, 'y': 12.0}
```

HyperOpt를 이용한 XGBoost 하이퍼 파라미터 최적화

```
# 1. 학습용 / 테스트용 데이터 추출

import pandasas pd
import numpyas np
from sklearn.datasetsimport load_breast_cancer
from sklearn.model_selectionimport 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
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_feature
s, 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_trai
n, test_size=0.1, random_state=156 )
```

```
# 2. 검색 공간 설정

from hyperopt import hp

# max_depth는 5에서 20까지 1간격으로, min_child_weight는 1에서 2
까지 1간격으로
# colsample_bytree는 0.5에서 1사이, learning_rate는 0.01에서
0.2 사이 정규 분포된 값으로 검색.
xgb_search_space = {'max_depth': hp.quniform('max_depth',
5, 20, 1),
                    'min_child_weight': hp.quniform('min_ch
ild_weight', 1, 2, 1),
                    'learning_rate': hp.uniform('learning_r
ate', 0.01, 0.2),
                    'colsample_bytree': hp.uniform('colsamp
le_bytree', 0.5, 1),
                   }
```

```
# 3. 목적 함수 생성

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

# 유의사항 2가지
# 1. 검색 공간 설정값을 실수형 → 정수형 변환 작업
# fmin()에서 입력된 search_space 값으로 입력된 모든 값은 실수형임.
```

```
# XGBClassifier의 정수형 하이퍼 파라미터는 정수형 변환을 해줘야 함.
# 그러므로 int 사용함

#2. HyperOpt의 목적 함수는 최솟값을 반환할 수 있도록 최적화해야 하기 때문에 정확도와 같이 값이 클수록 좋은 성능 지표일 경우 -1을 곱한 뒤 반환하여, 더 큰 성능 지표가 더 작은 반환값이 되도록 만들어야 함.
# 예를 들어, 목적 함수의 반환값을 정확도록 한다면, 정확도는 값이 클수록 좋은 성증 지표이므로 0.8(80%) 보다는 0.9(90%)가 더 좋은 지표임. 그러나 fmin()함수는 최솟값을 최적화하므로 0.8을 더 좋은 최적화로 판단함. 이런 경우 정확도에 -1을 곱해주게 되면 -0.9가 -0.8보다 더 작은 값이므로 fmin() 함수는 -0.9를 더 좋은 최적화로 판단함

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}
```

```
# 4. 목적함수의 최소값 유추(fmin)

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 지정함
            trials=trial_val, rstate=np.random.default_rng
(seed=9))
print('best:', best)
```

```
100%|██████████| 50/50 [00:10<00:00,  4.80trial/s, best loss: -0.9670616939700244]
best: {'colsample_bytree': 0.6844417779397407, 'learning_rate': 0.1475201153968472, 'max_depth': 9.0, 'min_child_weight': 2.
0}
```

```
# 5. 성능평가

print('colsample_bytree:{0}, learning_rate:{1}, max_depth:
{2}, min_child_weight:{3}'.format(
    round(best['colsample_bytree'], 5), round(best['learnin
g_rate'], 5),
    int(best['max_depth']), int(best['min_child_weight'])))
from sklearn.metrics import confusion_matrix, accuracy_scor
e
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, r
ecall, f1, roc_auc))
xgb_wrapper = XGBClassifier(n_estimators=400,
                            learning_rate=round(best['learn
```

```
ing_rate'], 5),
                            max_depth=int(best['max_dept
h']),
                            min_child_weight=int(best['min_
child_weight']),
                            colsample_bytree=round(best['co
lsample_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)
```

```
[66]    validation_0-logloss:0.02022    validation_1-logloss:0.25985
[67]    validation_0-logloss:0.02010    validation_1-logloss:0.26161
[68]    validation_0-logloss:0.01998    validation_1-logloss:0.26049
[69]    validation_0-logloss:0.01987    validation_1-logloss:0.26093
[70]    validation_0-logloss:0.01977    validation_1-logloss:0.25877
[71]    validation_0-logloss:0.01966    validation_1-logloss:0.26048
[72]    validation_0-logloss:0.01956    validation_1-logloss:0.26022
[73]    validation_0-logloss:0.01945    validation_1-logloss:0.25763
[74]    validation_0-logloss:0.01935    validation_1-logloss:0.25927
[75]    validation_0-logloss:0.01926    validation_1-logloss:0.25970
[76]    validation_0-logloss:0.01915    validation_1-logloss:0.25861
[77]    validation_0-logloss:0.01906    validation_1-logloss:0.25966
오차 행렬
[[35  2]
 [ 4 73]]
정확도: 0.9474, 정밀도: 0.9733, 재현율: 0.9481,    F1: 0.9605, AUC:0.9933
```

📌 09. 분류 실습 - 캐글 산탄데르 고객 만족 예측

- 370개의 피처로 이루어진 데이터 세트 기반으로 산탄데르 은행 고객 만족 여부를 예측함
- 피처 이름은 모두 익명 처리돼 이름만 가지고 어떤 속성인지는 추정할 수 없음

- 클래스 레이블 명은 TARGET이며, 이 값이 1이면 불만을 가진 고객, 0이면 만족한 고객
- 모델의 성능 평가는 ROC-AUC(ROCE 곡선 영역)으로 평가함
- 대부분 만족이고 불만족인 데이터는 일부일 것이기 때문에 정확도 수치보다는 ROC-AUC가 더 적합함

데이터 전처리

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib
import warnings
warnings.filterwarnings('ignore')

cust_df = pd.read_csv("./train_santander.csv", encoding='la
tin-1')
print('dataset shape:', cust_df.shape)
cust_df.head(3)
```

o_var33_ult1	saldo_medio_var33_ult3	saldo_medio_var44_hace2	saldo_medio_var44_hace3	saldo_medio_var44_ult1	saldo_medio_var44_ult3	var38	TARGET
0.0	0.0	0.0	0.0	0.0	0.0	39205.17	0
0.0	0.0	0.0	0.0	0.0	0.0	49278.03	0
0.0	0.0	0.0	0.0	0.0	0.0	67333.77	0

```
cust_df.info()

# 111개의 피쳐가 float, 260개 피쳐가 int형으로 모든 피쳐가 숫자형, n
ull 값은 없음
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 76020 entries, 0 to 76019
Columns: 371 entries, ID to TARGET
dtypes: float64(111), int64(260)
memory usage: 215.2 MB
```

```
# 불만족 비율 확인
```

```
print(cust_df['TARGET'].value_counts())
unsatisfied_cnt = cust_df[cust_df['TARGET'] == 1].TARGET.co
unt()
total_cnt = cust_df.TARGET.count()
print('unsatisfied 비율은 {0:.2f}'.format((unsatisfied_cnt /
total_cnt)))
```

```
TARGET
0    73012
1     3008
Name: count, dtype: int64
unsatisfied 비율은 0.04
```

```
cust_df.describe( )
```

	ID	var3	var15	imp_ent_var16_ult1	imp_op_var39_comer_ult1	imp_op_var39_comer_ult3	imp_op_var40_comer_ult1	imp_
count	76020.000000	76020.000000	76020.000000	76020.000000	76020.000000	76020.000000	76020.000000	
mean	75964.050723	-1523.199277	33.212865	86.208265	72.363067	119.529632	3.559130	
std	43781.947379	39033.462364	12.956486	1614.757313	339.315831	546.266294	93.155749	
min	1.000000	-999999.000000	5.000000	0.000000	0.000000	0.000000	0.000000	
25%	38104.750000	2.000000	23.000000	0.000000	0.000000	0.000000	0.000000	
50%	76043.000000	2.000000	28.000000	0.000000	0.000000	0.000000	0.000000	
75%	113748.750000	2.000000	40.000000	0.000000	0.000000	0.000000	0.000000	
max	151838.000000	238.000000	105.000000	210000.000000	12888.030000	21024.810000	8237.820000	

```
cust_df['var3'].value_counts()
```

```
var3
2          74165
8            138
-999999      116
9            110
3            108
           ...
231            1
188            1
168            1
135            1
87             1
```

```
# var3 피처 값 대체 및 ID 피처 드롭
cust_df['var3'].replace(-999999, 2, inplace=True)
cust_df.drop('ID', axis=1, inplace=True)

# 피처 세트와 레이블 세트분리. 레이블 컬럼은 DataFrame의 맨 마지막에
위치해 컬럼 위치 -1로 분리
X_features = cust_df.iloc[:, :-1]
y_labels = cust_df.iloc[:, -1]
print('피처 데이터 shape:{0}'.format(X_features.shape))
```

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X_featu
res, y_labels,
                                                    test_si
ze=0.2, random_state=0)
train_cnt = y_train.count()
test_cnt = y_test.count()
print('학습 세트 Shape:{0}, 테스트 세트 Shape:{1}'.format(X_tra
in.shape , X_test.shape))

print(' 학습 세트 레이블 값 분포 비율')
print(y_train.value_counts()/train_cnt)
print('\n 테스트 세트 레이블 값 분포 비율')
print(y_test.value_counts()/test_cnt)
```

```
학습 세트 Shape:(60816, 369), 테스트 세트 Shape:(15204, 369)
 학습 세트 레이블 값 분포 비율
TARGET
0    0.960964
1    0.039036
Name: count, dtype: float64

 테스트 세트 레이블 값 분포 비율
TARGET
0    0.9583
1    0.0417
Name: count, dtype: float64
```

```
# X_train, y_train을 다시 학습과 검증 데이터 세트로 분리.
X_tr, X_val, y_tr, y_val = train_test_split(X_train, y_trai
n,
                                                    test_si
ze=0.3, random_state=0)
```

LightGBM 모델 학습과 하이퍼 파라미터 튜닝

```
from lightgbm import LGBMClassifier
from sklearn.metrics import roc_auc_score

# n_estimators는 500으로, learning_rate 0.05, random state는
예제 수행 시마다 동일 예측 결과를 위해 설정.
lgbm_clf = LGBMClassifier(n_estimators=500)

eval_set=[(X_tr, y_tr), (X_val, y_val)]

# 성능 평가 지표를 auc로, 조기 중단 파라미터는 100으로 설정하고 학습
수행.
lgbm_clf.fit(X_tr, y_tr, early_stopping_rounds=100, eval_me
tric="auc", eval_set=eval_set)

lgbm_roc_score = roc_auc_score(y_test, lgbm_clf.predict_pro
ba(X_test)[:,1])
print('ROC AUC: {0:.4f}'.format(lgbm_roc_score))
```

```
ogloss: 0.137391
[135]   training's auc: 0.946253        training's binary_logloss: 0.0933204    valid_1's auc: 0.829006 valid_1's binary_l
ogloss: 0.13739
[136]   training's auc: 0.946589        training's binary_logloss: 0.0931143    valid_1's auc: 0.829054 valid_1's binary_l
ogloss: 0.137403
[137]   training's auc: 0.946772        training's binary_logloss: 0.092979     valid_1's auc: 0.828949 valid_1's binary_l
ogloss: 0.137447
[138]   training's auc: 0.946832        training's binary_logloss: 0.0929083    valid_1's auc: 0.828907 valid_1's binary_l
ogloss: 0.137476
[139]   training's auc: 0.947105        training's binary_logloss: 0.0927328    valid_1's auc: 0.829034 valid_1's binary_l
ogloss: 0.137463
[140]   training's auc: 0.94779 training's binary_logloss: 0.0924716    valid_1's auc: 0.829175 valid_1's binary_logloss:
0.137451
[141]   training's auc: 0.948038        training's binary_logloss: 0.0923201    valid_1's auc: 0.829218 valid_1's binary_l
ogloss: 0.137468
[142]   training's auc: 0.948302        training's binary_logloss: 0.0921179    valid_1's auc: 0.829267 valid_1's binary_l
ogloss: 0.137482
ROC AUC: 0.8384
```

HyperOpt를 활용하여 최적의 파라미터 값 찾기

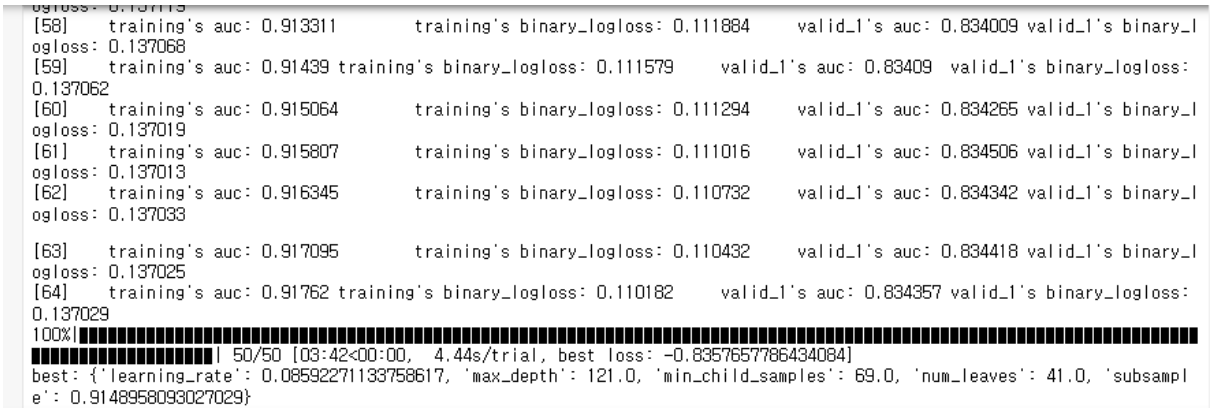
```
# 1. 검색 공간 설정

from hyperopt import hp

lgbm_search_space = {'num_leaves': hp.quniform('num_leave
s', 32, 64, 1),
                     'max_depth': hp.quniform('max_depth',
100, 160, 1),
                     'min_child_samples': hp.quniform('min_
child_samples', 60, 100, 1),
                     'subsample': hp.uniform('subsample',
0.7, 1),
                     'learning_rate': hp.uniform('learning_
rate', 0.01, 0.2)
                    }
```

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


# 2. 목적 함수 구하기

# 추후 fmin()에서 입력된 search_space값으로 LGBMCclassifier 교차
검증 학습 후 -1* roc_auc 평균 값을 반환

def objective_func(search_space):
    lgbm_clf =  LGBMClassifier(n_estimators=100, num_leaves
=int(search_space['num_leaves']),
                               max_depth=int(search_space
['max_depth']),
                               min_child_samples=int(search
_space['min_child_samples']),
                               subsample=search_space['subs
ample'],
                               learning_rate=search_space
['learning_rate'])
```

```
    # 3개 k-fold 방식으로 평가된 roc_auc 지표를 담는 list
    roc_auc_list = []

    # 3개 k-fold방식 적용
    kf = KFold(n_splits=3)
    # X_train을 다시 학습과 검증용 데이터로 분리
    for tr_index, val_index in kf.split(X_train):
        # kf.split(X_train)으로 추출된 학습과 검증 index값으로 학
습과 검증 데이터 세트 분리
        X_tr, y_tr = X_train.iloc[tr_index], y_train.iloc[t
r_index]
        X_val, y_val = X_train.iloc[val_index], y_train.ilo
c[val_index]

        # early stopping은 30회로 설정하고 추출된 학습과 검증 데이
터로 LGBMClassifier 학습 수행.
        lgbm_clf.fit(X_tr, y_tr, early_stopping_rounds=30,
eval_metric="auc",
           eval_set=[(X_tr, y_tr), (X_val, y_val)])

        # 1로 예측한 확률값 추출후 roc auc 계산하고 평균 roc auc
계산을 위해 list에 결과값 담음.
        score = roc_auc_score(y_val, lgbm_clf.predict_proba
(X_val)[:, 1])
        roc_auc_list.append(score)

    # 3개 k-fold로 계산된 roc_auc값의 평균값을 반환하되,
    # HyperOpt는 목적함수의 최소값을 위한 입력값을 찾으므로 -1을 곱한
뒤 반환.
    return -1*np.mean(roc_auc_list)
```

```
from hyperopt import fmin, tpe, Trials

trials = Trials()

# fmin()함수를 호출. max_evals지정된 횟수만큼 반복 후 목적함수의 최소
값을 가지는 최적 입력값 추출.
best = fmin(fn=objective_func, space=lgbm_search_space, alg
```

```
o=tpe.suggest,
            max_evals=50, # 최대 반복 횟수를 지정합니다.
            trials=trials, rstate=np.random.default_rng(see
d=30))

print('best:', best)
```




```
# n_estimators를 500증가 후 최적으로 찾은 하이퍼 파라미터를 기반으로
학습과 예측 수행.
lgbm_clf =  LGBMClassifier(n_estimators=500, num_leaves=int
(best['num_leaves']),
                           max_depth=int(best['max_dept
h']),
                           min_child_samples=int(best['min_
child_samples']),
                           subsample=round(best['subsampl
e'], 5),
                           learning_rate=round(best['learni
ng_rate'], 5)
                          )

# evaluation metric을 auc로, early stopping은 100 으로 설정하고
학습 수행.
lgbm_clf.fit(X_tr, y_tr, early_stopping_rounds=100,
            eval_metric="auc",eval_set=[(X_tr, y_tr), (X_va
l, y_val)])

lgbm_roc_score = roc_auc_score(y_test, lgbm_clf.predict_pro
```

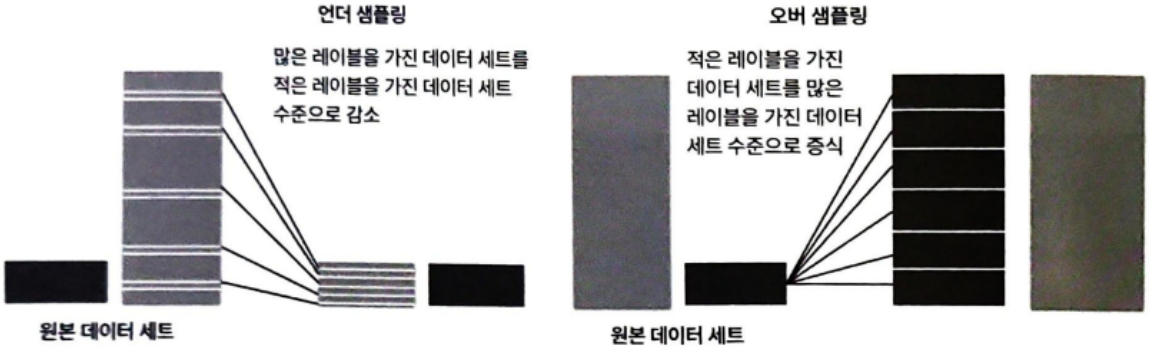
```
ba(X_test)[:,1])
print('ROC AUC: {0:.4f}'.format(lgbm_roc_score))
```

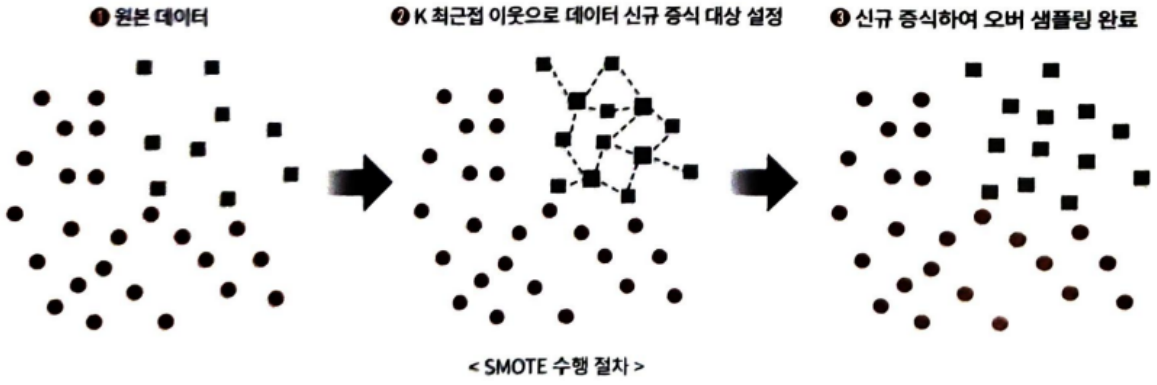
```
[133]   training's auc: 0.946517        training's binary_logloss: 0.0940387    valid_1's auc: 0.825927 valid_1's binary_l
ogloss: 0.138201
[134]   training's auc: 0.946629        training's binary_logloss: 0.0939145    valid_1's auc: 0.825757 valid_1's binary_l
ogloss: 0.138251
[135]   training's auc: 0.946881        training's binary_logloss: 0.0937509    valid_1's auc: 0.825661 valid_1's binary_l
ogloss: 0.138318
[136]   training's auc: 0.946978        training's binary_logloss: 0.0936331    valid_1's auc: 0.825493 valid_1's binary_l
ogloss: 0.138375
[137]   training's auc: 0.947196        training's binary_logloss: 0.0934537    valid_1's auc: 0.825437 valid_1's binary_l
ogloss: 0.138394
[138]   training's auc: 0.947281        training's binary_logloss: 0.0933505    valid_1's auc: 0.825157 valid_1's binary_l
ogloss: 0.138466
[139]   training's auc: 0.947514        training's binary_logloss: 0.0931953    valid_1's auc: 0.824876 valid_1's binary_l
ogloss: 0.138568
[140]   training's auc: 0.947793        training's binary_logloss: 0.0930837    valid_1's auc: 0.824702 valid_1's binary_l
ogloss: 0.138638
[141]   training's auc: 0.947875        training's binary_logloss: 0.0929829    valid_1's auc: 0.824448 valid_1's binary_l
ogloss: 0.138739
ROC AUC: 0.8446
```

📌 10. 분류 실습 - 캐글 신용카드 사기 검출

- 신용카드 데이터셋을 이용해 신용카드 사기 검출 분류 실습을 수행
- 데이터셋의 레이블인 Class 속성은 매우 불균형한 분포를 가지고 있음 (사기 검출이나 이상 검출의 특성)
  - Class의 0은 정상, 1은 사기 트랜잭션을 의미
  - 284,807 건 중 492건이 사기 (약 0.172%)
- 0으로만 예측해도 정확도가 매우 높아지므로 재현율 지표를 높이는 것이 주요 목표가 될 수 있겠음
- 데이터 전처리/재가공 실습 多
  - 데이터 분포도 변환(StadndardScaler 클래스(정규분포), 로그변환)
  - 이상치 데이터 제거 (IQR 방식)
  - 오버 샘플링 적용 (SMOTE 기법)
    - 이상 레이블을 가지는 데이터 건수가 적으면 다양한 유형을 학습하지 못하고 정상 레이블이 많으면 일방적으로 정상 레이블로 치우친 학습을 할 수 있어 예측 성능에 문제가 발생함. 이를 해결하기 위해 언더샘플링, 오버샘플링 방식이 있고 일반적으로 오버 샘플링 방식이 예측 성능상 조금 유리한 경우가 많음

- SMOTE 방식 : 적은 데이터 세트를 증식시켜 학습을 위한 충분한 데이터를 확보하는 방법. 동일한 데이터를 단순히 증식하면 과적합 되기 때문에 원본 데이터의 피쳐 값들을 아주 조금씩 변경해줌. **SMOTE는 적은 데이터셋에 있는 개별 데이터들의 K 최근접 이웃(K Nearest Neighbor) 찾아서 이 데이터와 K개 이웃들의 차이를 일정 값으로 만들어서 기존 데이터와 약간 차이가 나는 새로운 데이터를 생성하는 방식.**





< SMOTE 수행 절차 >

- 실습 모델 : 로지스틱 회귀 분석, LightGBM

데이터 일차 가공 및 모델 학습/예측/평가

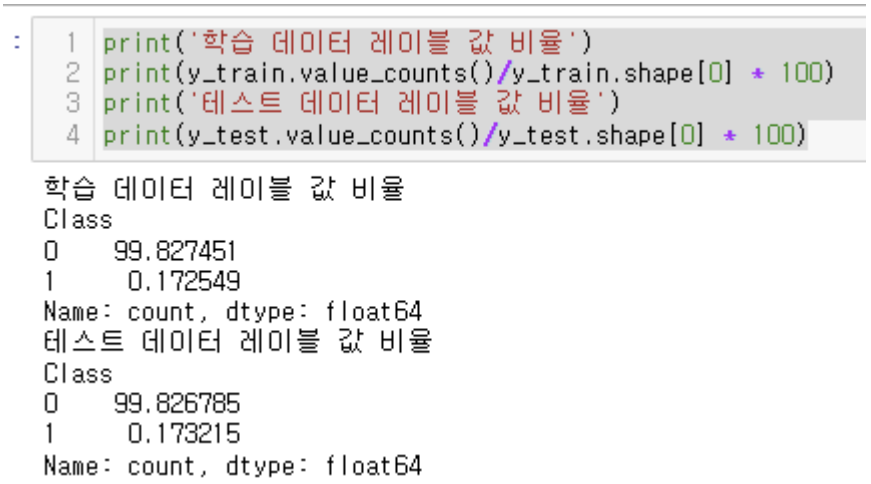
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")
%matplotlib inline

card_df = pd.read_csv('./creditcard.csv')
card_df.head(3)
```

V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Class
-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.066928	0.128539	-0.189115	0.133558	-0.021053	149.62	0
0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.101288	-0.339846	0.167170	0.125895	-0.008983	0.014724	2.69	0
-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.909412	-0.689281	-0.327642	-0.139097	-0.055353	-0.059752	378.66	0

```
from sklearn.model_selection import train_test_split

# 인자로 입력받은 DataFrame을 복사 한 뒤 Time 컬럼만 삭제하고 복사된 DataFrame 반환
def get_preprocessed_df(df=None):
    df_copy = df.copy()
    df_copy.drop('Time', axis=1, inplace=True)
    return df_copy
```

```
# 사전 데이터 가공 후 학습과 테스트 데이터 세트를 반환하는 함수.
def get_train_test_dataset(df=None):
    # 인자로 입력된 DataFrame의 사전 데이터 가공이 완료된 복사 DataFrame 반환
    df_copy = get_preprocessed_df(df)

    # DataFrame의 맨 마지막 컬럼이 레이블, 나머지는 피처들
    X_features = df_copy.iloc[:, :-1]
    y_target = df_copy.iloc[:, -1]

    # train_test_split( )으로 학습과 테스트 데이터 분할. stratify=y_target으로 Stratified 기반 분할
    X_train, X_test, y_train, y_test = \
    train_test_split(X_features, y_target, test_size=0.3, random_state=0, stratify=y_target)

    # 학습과 테스트 데이터 세트 반환
    return X_train, X_test, y_train, y_test

X_train, X_test, y_train, y_test = get_train_test_dataset(card_df)
```

```
print('학습 데이터 레이블 값 비율')
print(y_train.value_counts()/y_train.shape[0] * 100)
print('테스트 데이터 레이블 값 비율')
print(y_test.value_counts()/y_test.shape[0] * 100)
```




```
# 모델 평가

from sklearn.metrics import confusion_matrix, accuracy_scor
e, precision_score, recall_score, f1_score
from sklearn.metrics import 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, r
ecall, f1, roc_auc))
```

```
# Logistic Regression 모델

from sklearn.linear_model import LogisticRegression

lr_clf = LogisticRegression()
lr_clf.fit(X_train, y_train) #학습
lr_pred = lr_clf.predict(X_test) # 예측 값
lr_pred_proba = lr_clf.predict_proba(X_test)[:, 1] #1일 때
확률

# 3장에서 사용한 get_clf_eval() 함수를 이용하여 평가 수행.
get_clf_eval(y_test, lr_pred, lr_pred_proba)
```

```
오차 행렬
[[85281    14]
 [   56    92]]
정확도: 0.9992, 정밀도: 0.8679, 재현율: 0.6216,    F1: 0.7244, AUC:0.9590
```

```
# 앞서 수행할 예제 코드에서 반복적으로 모델을 변경해 학습/예측/평가할 것
이므로 이를 위한 별도의 함수 생성

# 인자로 사이킷런의 Estimator객체와, 학습/테스트 데이터 세트를 입력 받
아서 학습/예측/평가 수행.

def get_model_train_eval(model, ftr_train=None, ftr_test=No
ne, tgt_train=None, tgt_test=None):
    model.fit(ftr_train, tgt_train)
    pred = model.predict(ftr_test)
    pred_proba = model.predict_proba(ftr_test)[:, 1]
    get_clf_eval(tgt_test, pred, pred_proba)
```

```
# LightGBM 모델

from lightgbm import LGBMClassifier

lgbm_clf = LGBMClassifier(n_estimators=1000, num_leaves=64,
n_jobs=-1, boost_from_average=False)
```

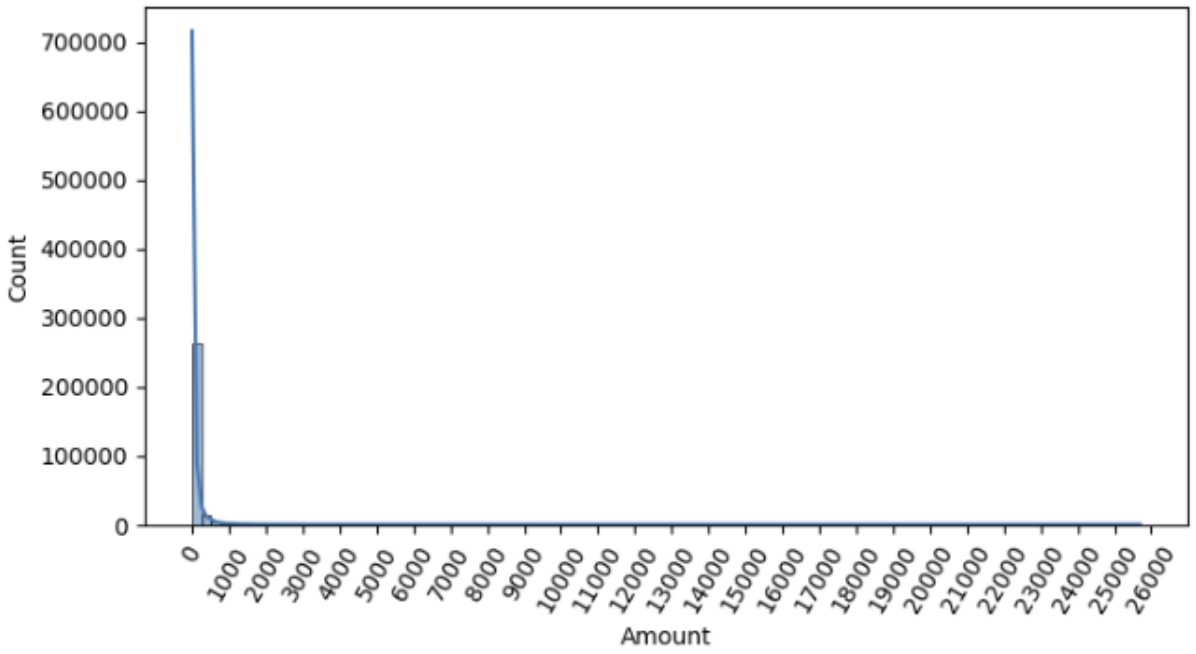
```
get_model_train_eval(lgbm_clf, ftr_train=X_train, ftr_test=
X_test, tgt_train=y_train, tgt_test=y_test)
```

```
오차 행렬
[[85290     5]
 [   36   112]]
정확도: 0.9995, 정밀도: 0.9573, 재현율: 0.7568,    F1: 0.8453, AUC:0.9790
```

데이터 분포도 변환 후 모델 학습/예측/평가 (StandardScaler, 로그 변환)

- 왜곡된 분포도를 가진 데이터를 재가공하며 StandScaler, 로그 변환 실습
- StandardScaler(정규분포변환)
  - 로지스틱 회귀는 선형 모델로, 대부분의 선형 모델은 중요 피처들의 값이 정규 분포 형태를 유지하는 것을 선호함. Amount 피처는 신용카드 사용 금액으로 정상/사기 트랜잭션을 결정하는 매우 중요한 속성일 가능성이 높음.
  - 분포도 확인해 보면, Amount 가 1,000불 이하인 데이터가 대부분이며 꼬리가 긴 형태의 분포 곡선을 가짐
  - Amount를 표준 정규 분포 형태로 변환하여 로지스틱 회귀 예측 성능을 측정할 것이며, 사이킷런의 StandardScaler 클래스 이용
- 로그 변환
  - 원래 값을 log 값으로 변환해 원래 큰 값을 상대적으로 작은 값으로 변환하기 때문에 분포도의 왜곡을 상당 수준 개선
  - 넘파이의 log1p( ) 함수를 사용 (자세한 내용은 5장 참조)

```
import seaborn as sns

plt.figure(figsize=(8, 4))
plt.xticks(range(0, 30000, 1000), rotation=60)
sns.histplot(card_df['Amount'], bins=100, kde=True)
plt.show()
```



StandardScaler 클래스 실습

```
from sklearn.preprocessing import StandardScaler
# 사이킷런의 StandardScaler를 이용하여 정규분포 형태로 Amount 피처값
변환하는 로직으로 수정.
def get_preprocessed_df(df=None):
    df_copy = df.copy()
    scaler = StandardScaler()
    amount_n = scaler.fit_transform(df_copy['Amount'].value
s.reshape(-1, 1))
    # 변환된 Amount를 Amount_Scaled로 피처명 변경후 DataFrame맨
앞 컬럼으로 입력
    df_copy.insert(0, 'Amount_Scaled', amount_n)
    # 기존 Time, Amount 피처 삭제
    df_copy.drop(['Time','Amount'], axis=1, inplace=True)
    return df_copy
```

```
# Amount를 정규분포 형태로 변환 후 로지스틱 회귀 및 LightGBM 수행.
X_train, X_test, y_train, y_test = get_train_test_dataset(c
ard_df)

print('### 로지스틱 회귀 예측 성능 ###')
lr_clf = LogisticRegression()
```

```
get_model_train_eval(lr_clf, ftr_train=X_train, ftr_test=X_
test, tgt_train=y_train, tgt_test=y_test)

print('### LightGBM 예측 성능 ###')
lgbm_clf = LGBMClassifier(n_estimators=1000, num_leaves=64,
n_jobs=-1, boost_from_average=False)
get_model_train_eval(lgbm_clf, ftr_train=X_train, ftr_test=
X_test, tgt_train=y_train, tgt_test=y_test)
```

```
오차 행렬
[[85281    14]
 [   56    92]]
정확도: 0.9992, 정밀도: 0.8679, 재현율: 0.6216,    F1: 0.7244, AUC:0.9590
```

```
오차 행렬
[[85290     5]
 [   36   112]]
정확도: 0.9995, 정밀도: 0.9573, 재현율: 0.7568,    F1: 0.8453, AUC:0.9790
```

```
### 로지스틱 회귀 예측 성능 ###
오차 행렬
[[85281    14]
 [   55    93]]
정확도: 0.9992, 정밀도: 0.8692, 재현율: 0.6284,    F1: 0.7294, AUC:0.9706
### LightGBM 예측 성능 ###
오차 행렬
[[85290     5]
 [   37   111]]
정확도: 0.9995, 정밀도: 0.9569, 재현율: 0.7500,    F1: 0.8409, AUC:0.9779
```

로그 변환

```
def get_preprocessed_df(df=None):
    df_copy = df.copy()
    # 넘파이의 log1p( )를 이용하여 Amount를 로그 변환
    amount_n = np.log1p(df_copy['Amount'])
    df_copy.insert(0, 'Amount_Scaled', amount_n)
    df_copy.drop(['Time','Amount'], axis=1, inplace=True)
    return df_copy
```

```
X_train, X_test, y_train, y_test = get_train_test_dataset(c
ard_df)

print('### 로지스틱 회귀 예측 성능 ###')
get_model_train_eval(lr_clf, ftr_train=X_train, ftr_test=X_
test, tgt_train=y_train, tgt_test=y_test)

print('### LightGBM 예측 성능 ###')
```

```
get_model_train_eval(lgbm_clf, ftr_train=X_train, ftr_test=
X_test, tgt_train=y_train, tgt_test=y_test)
```

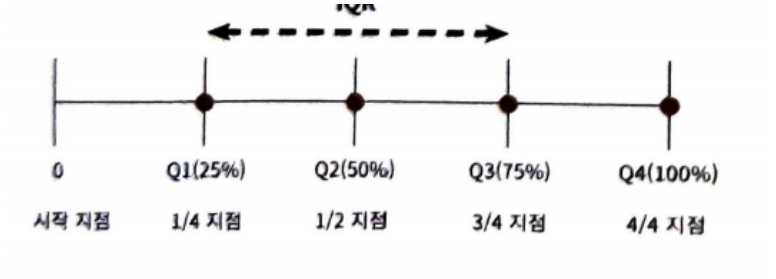
```
오차 행렬
[[85281    14]
 [   56    92]]
정확도: 0.9992, 정밀도: 0.8679, 재현율: 0.6216,    F1: 0.7244, AUC:0.9590
```

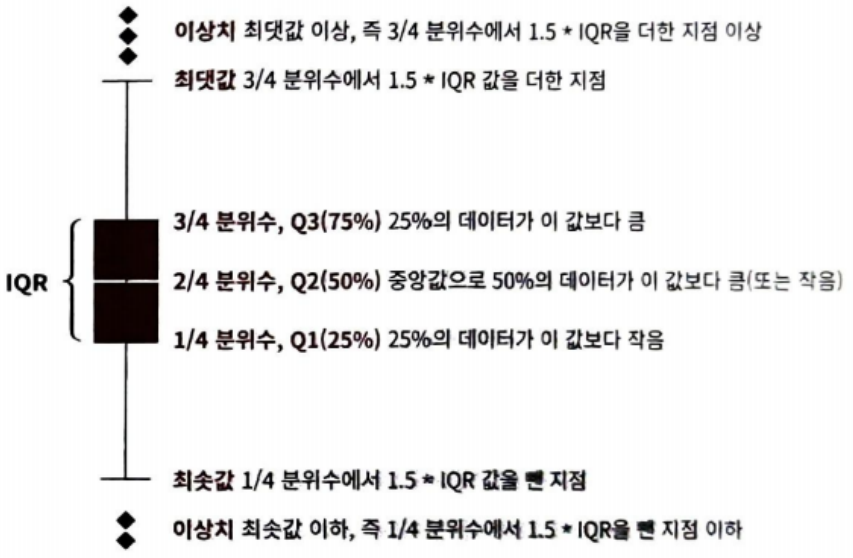
```
오차 행렬
[[85290     5]
 [   36   112]]
정확도: 0.9995, 정밀도: 0.9573, 재현율: 0.7568,    F1: 0.8453, AUC:0.9790
```

```
### 로지스틱 회귀 예측 성능 ###
오차 행렬
[[85282    13]
 [   59    89]]
정확도: 0.9992, 정밀도: 0.8725, 재현율: 0.6014,    F1: 0.7120, AUC:0.9734
### LightGBM 예측 성능 ###
오차 행렬
[[85290     5]
 [   35   113]]
정확도: 0.9995, 정밀도: 0.9576, 재현율: 0.7635,    F1: 0.8496, AUC:0.9796
```

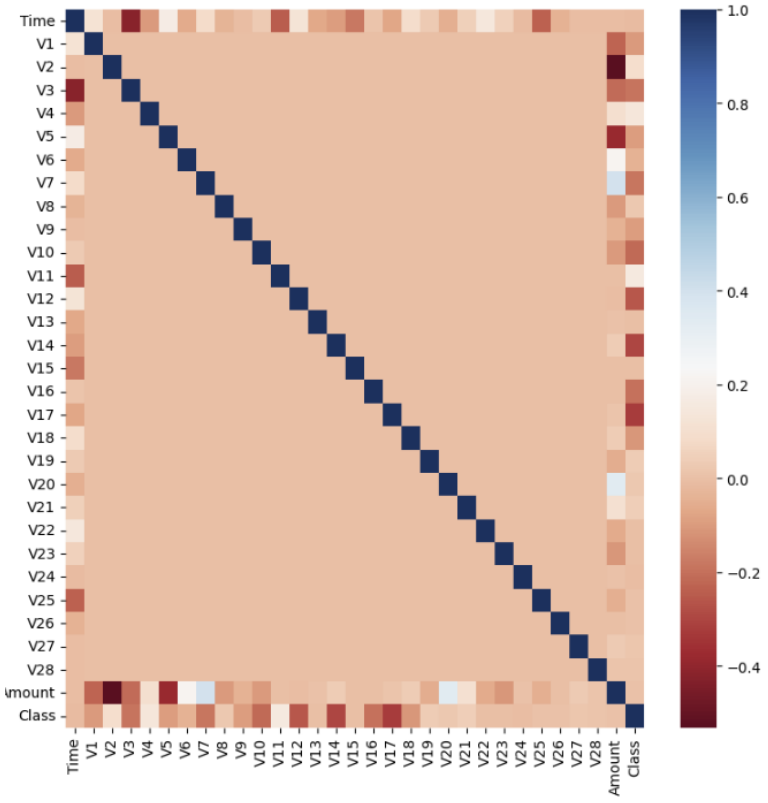
이상치 데이터 제거 후 모델 학습/예측/평가

- 이상치 데이터는 전체 데이터의 패턴에서 벗어난 이상 값을 가진 데이터이며, 아웃라이어라고도 불림
- 매우 많은 피처가 있을 경우, 이들 중 결정값(즉 레이블)과 가장 상관성이 높은 피처들을 위주로 이상치를 검출하는 것이 좋음





```
import seaborn as sns

plt.figure(figsize=(9, 9))
corr = card_df.corr()
sns.heatmap(corr, cmap='RdBu') #v14, v17
```




```
import numpy as np

def get_outlier(df=None, column=None, weight=1.5):
    # fraud에 해당하는 column 데이터만 추출, 1/4 분위와 3/4 분위
지점을 np.percentile로 구함.
    fraud = df[df['Class']==1][column]
    quantile_25 = np.percentile(fraud.values, 25)
    quantile_75 = np.percentile(fraud.values, 75)
    # IQR을 구하고, IQR에 1.5를 곱하여 최대값과 최소값 지점 구함.
    iqr = quantile_75 - quantile_25
    iqr_weight = iqr * weight
    lowest_val = quantile_25 - iqr_weight
```

```
    highest_val = quantile_75 + iqr_weight
    # 최대값 보다 크거나, 최소값 보다 작은 값을 아웃라이어로 설정하고 D
ataFrame index 반환.
    outlier_index = fraud[(fraud < lowest_val) | (fraud > h
ighest_val)].index
    return outlier_index
```

```
outlier_index = get_outlier(df=card_df, column='V14', weigh
t=1.5)
print('이상치 데이터 인덱스:', outlier_index)
```

```
이상치 데이터 인덱스: Index([8296, 8615, 9035, 9252], dtype='int64')
```

```
# get_processed_df( )를 로그 변환 후 V14 피처의 이상치 데이터를 삭
제하는 로직으로 변경.
def get_preprocessed_df(df=None):
    df_copy = df.copy()
    amount_n = np.log1p(df_copy['Amount'])
    df_copy.insert(0, 'Amount_Scaled', amount_n)
    df_copy.drop(['Time','Amount'], axis=1, inplace=True)
    # 이상치 데이터 삭제하는 로직 추가
    outlier_index = get_outlier(df=df_copy, column='V14', w
eight=1.5)
    df_copy.drop(outlier_index, axis=0, inplace=True)
    return df_copy

X_train, X_test, y_train, y_test = get_train_test_dataset(c
ard_df)
print('### 로지스틱 회귀 예측 성능 ###')
get_model_train_eval(lr_clf, ftr_train=X_train, ftr_test=X_
test, tgt_train=y_train, tgt_test=y_test)
print('### LightGBM 예측 성능 ###')
get_model_train_eval(lgbm_clf, ftr_train=X_train, ftr_test=
X_test, tgt_train=y_train, tgt_test=y_test)
```

```
오차 행렬
[[85281    14]
 [   56    92]]
정확도: 0.9992, 정밀도: 0.8679, 재현율: 0.6216,    F1: 0.7244, AUC:0.9590
```

```
오차 행렬
[[85290     5]
 [   36   112]]
정확도: 0.9995, 정밀도: 0.9573, 재현율: 0.7568,    F1: 0.8453, AUC:0.9790
```

```
### 로지스틱 회귀 예측 성능 ###
오차 행렬
[[85280    15]
 [   48    98]]
정확도: 0.9993, 정밀도: 0.8673, 재현율: 0.6712,    F1: 0.7568, AUC:0.9725
### LightGBM 예측 성능 ###
오차 행렬
[[85290     5]
 [   25   121]]
정확도: 0.9996, 정밀도: 0.9603, 재현율: 0.8288,    F1: 0.8897, AUC:0.9780
```

```
outlier_index = get_outlier(df=card_df, column='V14', weigh
t=1.5)
print('이상치 데이터 인덱스:', outlier_index)
```

SMOTE 오버 샘플링 적용 후 모델 학습/예측/평가

- 유의사항 : 반드시 학습 데이터 세트만 오버 샘플링을 해야 함. 검증 데이터 셋이나 테스트 데이터 세트를 오버 샘플링할 경우 결국 원본 데이터 세트가 아닌 데이터 세트에서 검증 또는 테스트를 수행하기 때문에 올바른 검증/테스트가 될 수 없음

```
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=0)
X_train_over, y_train_over = smote.fit_resample(X_train, y_
train)
print('SMOTE 적용 전 학습용 피처/레이블 데이터 세트: ', X_train.sh
ape, y_train.shape)
print('SMOTE 적용 후 학습용 피처/레이블 데이터 세트: ', X_train_ov
er.shape, y_train_over.shape)
print('SMOTE 적용 후 레이블 값 분포: \n', pd.Series(y_train_ove
r).value_counts())
```

```
SMOTE 적용 전 학습용 피처/레이블 데이터 세트:  (199362, 29) (199362,)
SMOTE 적용 후 학습용 피처/레이블 데이터 세트:  (398040, 29) (398040,)
SMOTE 적용 후 레이블 값 분포:
 Class
0    199020
1    199020
Name: count, dtype: int64
```

```
lr_clf = LogisticRegression()
# ftr_train과 tgt_train 인자값이 SMOTE 증식된 X_train_over와 y_
```

```
train_over로 변경됨에 유의
get_model_train_eval(lr_clf, ftr_train=X_train_over, ftr_te
st=X_test, tgt_train=y_train_over, tgt_test=y_tes
```

```
오차 행렬
[[85281    14]
 [   56    92]]
정확도: 0.9992, 정밀도: 0.8679, 재현율: 0.6216,    F1: 0.7244, AUC:0.9590
```

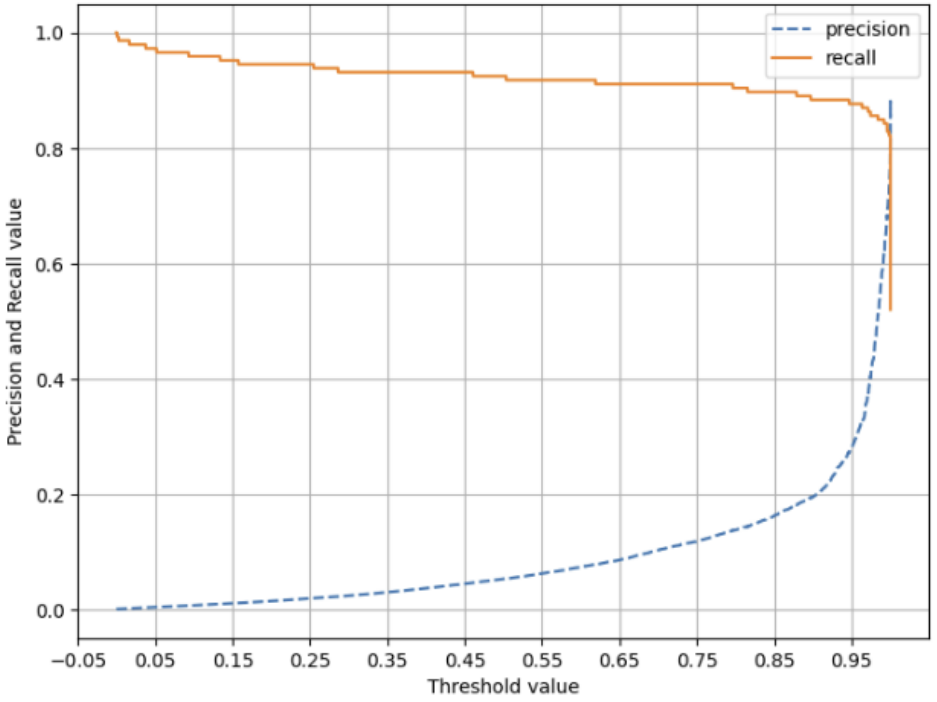
```
오차 행렬
[[82933  2362]
 [   11   135]]
정확도: 0.9722, 정밀도: 0.0541, 재현율: 0.9247,    F1: 0.1022, AUC:0.9736
```

- 재현율이 92%로 크게 증가했으나 정밀도가 5.4%로 급격히 감소함. 이는 로지스틱 회귀 모델이 오버 샘플링으로 인해 실제 원본 데이터의 유형보다 너무나 많은 Class=1(사기) 데이터를 학습하면서 실제 테스트 데이터 세트에서 예측을 지나치게 Class=1(사기)로 적용해 정밀도가 떨어짐.
- 분류 결정 임곗값에 따른 정밀도와 재현율 곡선을 통해 SMOTE로 학습된 로지스틱 회귀 모델에 어떤 문제가 발생했는지 시각적으로 확인 필요

```
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
from sklearn.metrics import precision_recall_curve
%matplotlib inline

def precision_recall_curve_plot(y_test , pred_proba_c1):
    # threshold ndarray와 이 threshold에 따른 정밀도, 재현율 nda
rray 추출.
    precisions, recalls, thresholds = precision_recall_curv
e( y_test, pred_proba_c1)

    # X축을 threshold값으로, Y축은 정밀도, 재현율 값으로 각각 Plot
수행. 정밀도는 점선으로 표시
    plt.figure(figsize=(8,6))
    threshold_boundary = thresholds.shape[0]
    plt.plot(thresholds, precisions[0:threshold_boundary],
linestyle='--', label='precision')
    plt.plot(thresholds, recalls[0:threshold_boundary],labe
l='recall')

    # threshold 값 X 축의 Scale을 0.1 단위로 변경
    start, end = plt.xlim()
```

```
    plt.xticks(np.round(np.arange(start, end, 0.1),2))

    # x축, y축 label과 legend, 그리고 grid 설정
    plt.xlabel('Threshold value'); plt.ylabel('Precision an
d Recall value')
    plt.legend(); plt.grid()
    plt.show()
```

```
precision_recall_curve_plot( y_test, lr_clf.predict_proba(X
_test)[:, 1] )
```



LightGBM 모델에 SMOTE 오버 샘플링 데이터 세트 학습/예측/평가

```
lgbm_clf = LGBMClassifier(n_estimators=1000, num_leaves=64,
n_jobs=-1, boost_from_average=False)
get_model_train_eval(lgbm_clf, ftr_train=X_train_over, ftr_
test=X_test,
                     tgt_train=y_train_over, tgt_test=y_test)
```

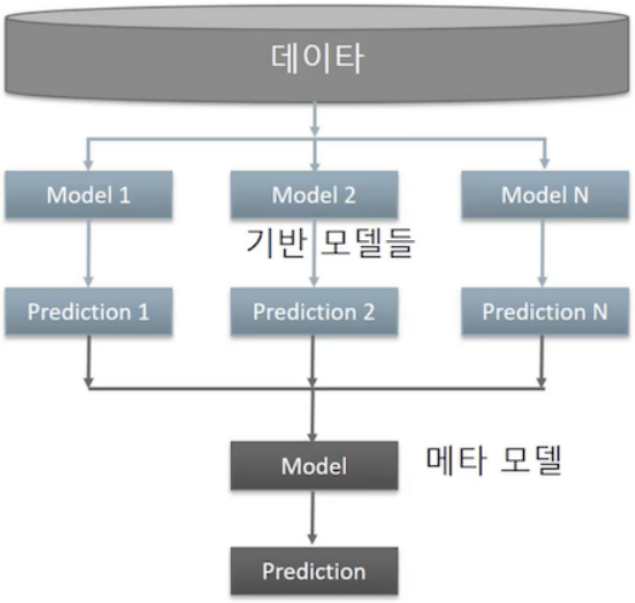
```
오차 행렬
[[85290     5]
 [   36   112]]
정확도: 0.9995, 정밀도: 0.9573, 재현율: 0.7568,    F1: 0.8453, AUC:0.9790
```

```
오차 행렬
[[85283    12]
 [   22   124]]
정확도: 0.9996, 정밀도: 0.9118, 재현율: 0.8493,    F1: 0.8794, AUC:0.9814
```

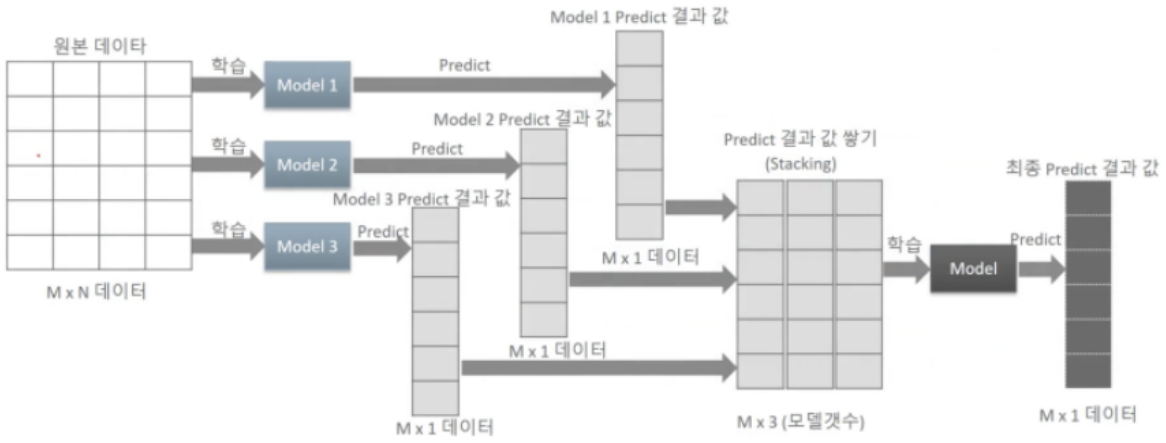
📌 11. 스태킹 앙상블

기본 스태킹

- 특징 : 개별적인 여러 알고리즘을 서로 결합해 예측 결과를 도출하는 점은 배깅 및 부스팅과 유사하지만, **큰 차이점은 개별 알고리즘으로 예측한 데이터를 기반으로 다시 예측을 수행한다. (메타 모델)**



- 과정
  - M x N 데이터 세트에 스캐팅 앙상블을 적용한다고 가정합니다.
  - 모델별 각각 학습을 시킨 뒤 예측을 수행하면 각각 M x 1 레이블 값을 도출합니다.
  - 모델별로 도출된 예측 레이블 값을 다시 합해서(스태킹) 새로운 데이터 세트를 만들고 이렇게 스태킹된 데이터 세트에 대해 최종 모델을 적용해 최종 예측합니다.



- 실습
  - 유방암 데이터 로딩 및 학습/테스트 데이터 나누기

```
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

cancer_data = load_breast_cancer()

X_data = cancer_data.data
y_label = cancer_data.target

X_train , X_test , y_train , y_test = train_test_split(X_da
ta , y_label , test_size=0.2 , random_state=0)
```

```
# 개별 ML 모델을 위한 Classifier 생성.
knn_clf  = KNeighborsClassifier(n_neighbors=4)
rf_clf = RandomForestClassifier(n_estimators=100, random_st
ate=0)
```

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

# 최종 Stacking 모델을 위한 Classifier생성.
lr_final = LogisticRegression(C=10)
```

```
# 개별 모델들을 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 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 :
```

```
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)
```

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

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

CV 세트 기반의 스태킹

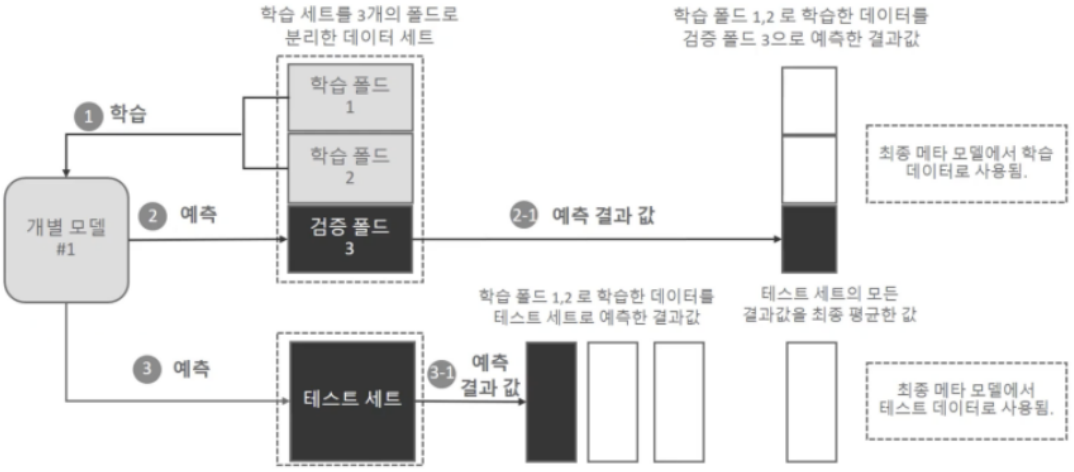
- 특징 : 앞선 기본 스태킹은 최종 학습할 때 레이블 데이터 세트로 학습 데이터가 아닌 테스트용 레이블 데이터 세트의 기반으로 학습했기에 과적합 문제가 발생할 수 있음. 그러므로 CV 세트 기반의 스태킹은 최종 메타 모델을 위한 데이터 세트를 만들 때 **교차 검증 기반으로 예측된 결과 데이터 세트를 이용함 (학습용 데이터 따로, 테스트용 데이터 따로 쌓는다)**

- 과정 :
  - 스텝1 : 각 모델별로 원본 학습/테스트 데이터를 예측한 결과 값을 기반으로 메타 모델을 위한 학습용/테스트용 데이터를 생성
  - 스텝2 :
    - 스텝 1에서 개별 모델들이 생성한 학습용 데이터를 모두 스태킹 형태로 합침
    - 메타 모델이 학습하도록 최종 학습용 데이터 세트를 생성
    - 각 모델이 생성한 테스트용 데이터를 모두 스태킹 형태로 합침
    - 메타 모델이 예측할 최종 테스트 데이터 세트를 생성
    - 메타 모델은 최종적으로 생성된 학습 데이터와 원본 학습 데이터의 레이블을 기반으로 학습

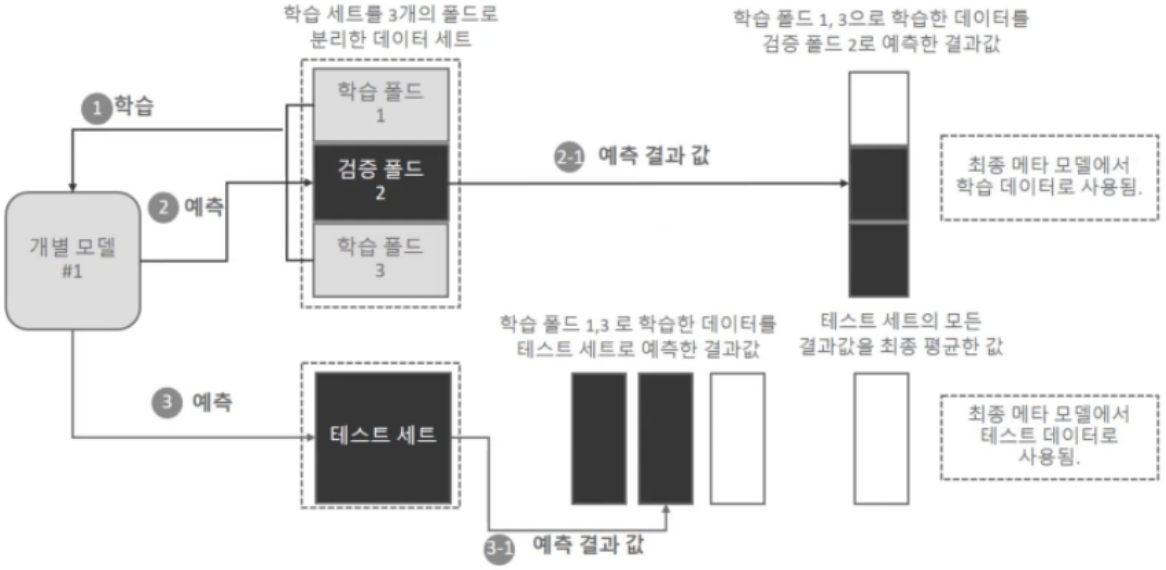
- 최종적으로 생성된 테스트 데이터 세트를 예측하고, 원본 테스트 데이터의 레이블 데이터를 기반으로 평가

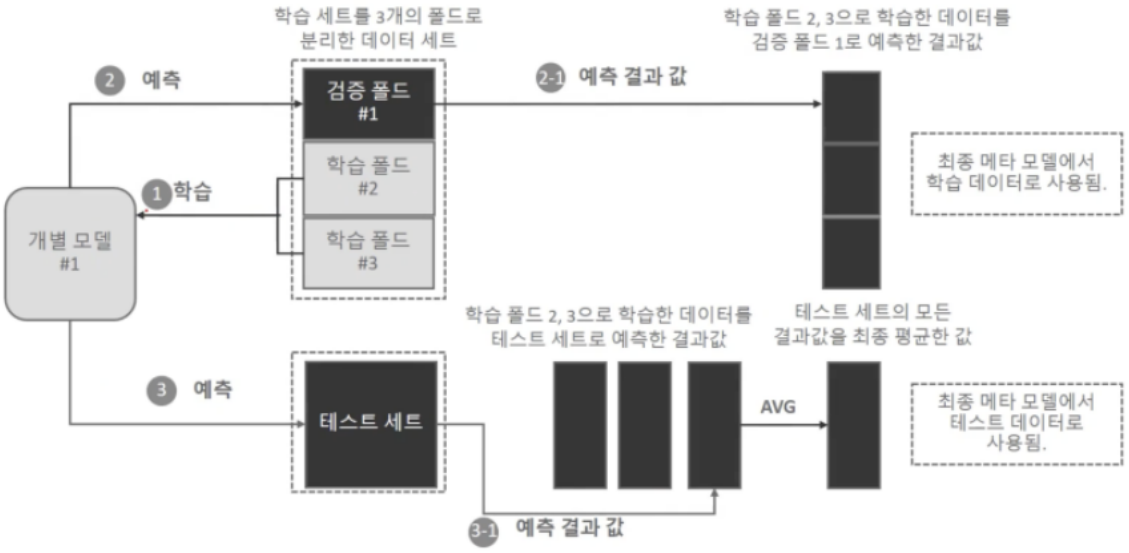
개별 모델#1 기반의 과정 설명

- <첫 번째 반복>
  - 학습할 데이터를 3개의 폴드로 나눔. 2개의 폴드는 학습용, 나머지 1개의 폴드는 검증을 위한 데이터 폴드로 구분
  - 학습할 데이터를 3개의 폴드로 나눔. 2개의 폴드는 학습용, 나머지 1개의 폴드는 검증을 위한 데이터 폴드로 구분
  - 2-1) 이렇게 학습된 개별 모델은 검증 폴드 1개 데이터로 예측하고 그 결과를 저장
  - 3-1) 학습된 개별 모델은 원본 테스트 데이터를 예측하여 예측값을 생성

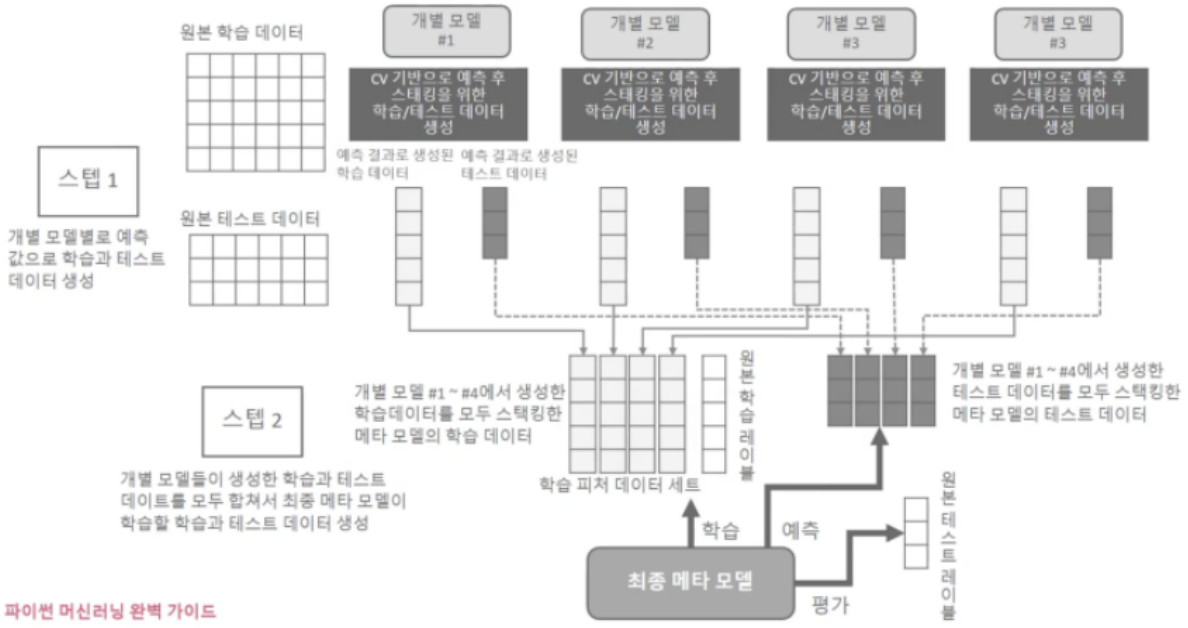


- <N 번째 반복>
  - 학습할 데이터를 3개의 폴드로 나눔. 2개의 폴드는 학습용, 나머지 1개의 폴드는 검증을 위한 데이터 폴드로 구분
  - 학습할 데이터를 3개의 폴드로 나눔. 2개의 폴드는 학습용, 나머지 1개의 폴드는 검증을 위한 데이터 폴드로 구분
  - 2-1) 이렇게 학습된 개별 모델은 검증 폴드 1개 데이터로 예측하고 그 결과를 저장
  - 3-1) 학습된 개별 모델은 원본 테스트 데이터를 예측하여 예측값을 생성





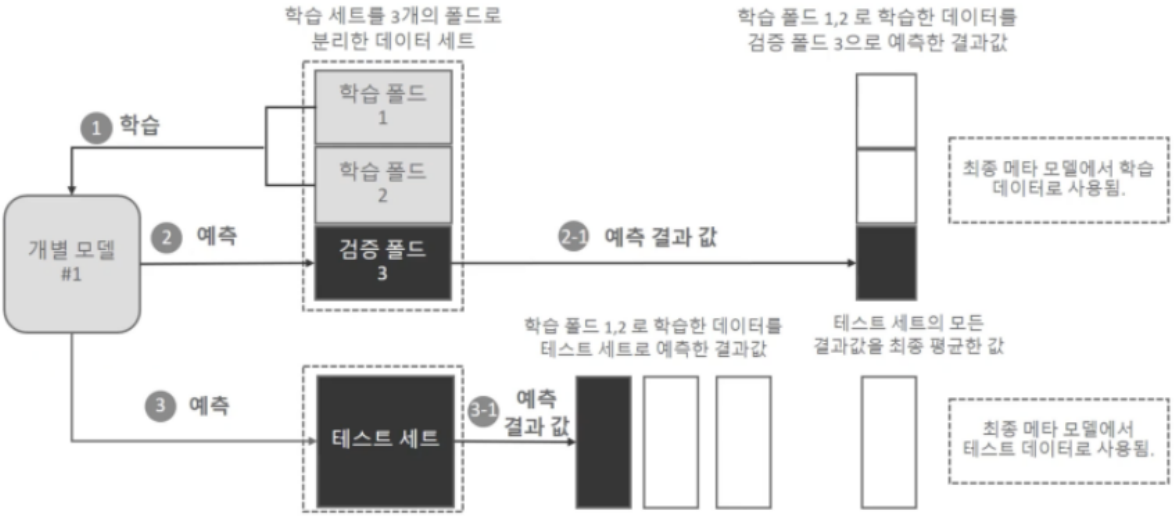
- <최종 메타 모델>
  - 메타 모델이 사용할 최종 학습 데이터 / 원본 데이터의 레이터의 레이블 데이터를 합쳐서 메타 모델을 학습시킨 후 최종 테스트 데이터로 예측을 수행한 뒤, 최종 예측 결과를 원 테스트 데이터의 레이블 데이터와 비교해 평가함



실습

```
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('\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] # 검증 데이터

        #폴드 세트 내부에서 다시 만들어진 학습 데이터로 기반 모델의 학
습 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_t
e).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 테
스트 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
```



```
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_tr
ain, y_train, X_test, 7)
dt_train, dt_test = get_stacking_base_datasets(dt_clf, X_tr
ain, 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
C:₩ProgramData₩anaconda3₩Lib₩site-packages₩sklearn₩ensemble₩_weight_boosting.py:519: FutureWarning: The SAMME.R algorithm (t
he default) is deprecated and will be removed in 1.6. Use the SAMME algorithm to circumvent this warning.
  warnings.warn(
        폴드 세트:  1  시작
C:₩ProgramData₩anaconda3₩Lib₩site-packages₩sklearn₩ensemble₩_weight_boosting.py:519: FutureWarning: The SAMME.R algorithm (t
he default) is deprecated and will be removed in 1.6. Use the SAMME algorithm to circumvent this warning.
  warnings.warn(
        폴드 세트:  2  시작
```

```
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.sha
pe,
```

```
      '스태킹 테스트 피처 데이터 Shape:',Stack_final_X_test.shap
e)
```

```
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)
```

```
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_sc
ore(y_test, stack_final)))
```

```
최종 메타 모델의 예측 정확도: 0.9737
```