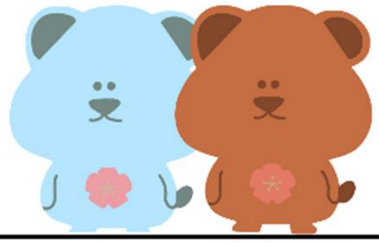
4장. 분류-Part 2

ML세션 팀 4 김남우, 도연수, 이덕주

EWHA EURON

목차



EWHA EURON

4.5 GBM

EWHA EURON

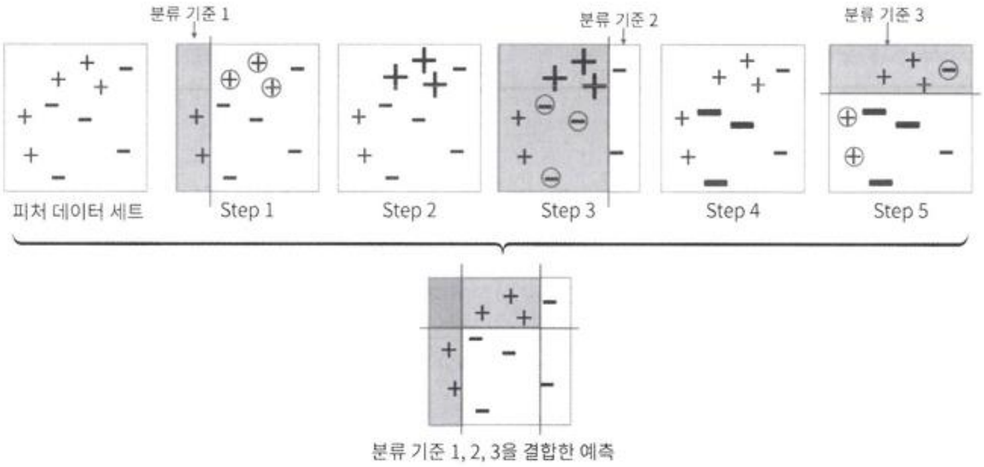
#4.5 GBM

Boosting Algorithm

약한 학습기(weak learner)를 순차적으로 학습, 예측하며 잘못 예측한 데이터에 가중치 부여를 통해 오류를 개선하는 방식

1. **AdaBoost (Adaptive boosting)**

2. Gradient Boost



#4.5 GBM

Boosting Algorithm

1. AdaBoost (Adaptive boosting)

2. **Gradient Boost**
   Gradient Descent를 사용한 Optimization

Loss Function

$$h(x) = y - f(x)$$

#4.5 GBM

Boosting Algorithm

2. Gradient Boost

```
from sklearn.ensemble import GradientBoostingClassifier
import time
import warnings
warnings.filterwarnings('ignore')

X_train, X_test, y_train, y_test = get_human_dataset()

# GBM 수행 시간 측정을 위함. 시작 시간 설정.
start_time = time.time()
```

```
gb_clf = GradientBoostingClassifier(random_state=0)
gb_clf.fit(X_train, y_train)
gb_pred = gb_clf.predict(X_test)
gb_accuracy = accuracy_score(y_test, gb_pred)

print('GBM 정확도: {0:.4f}'.format(gb_accuracy))
print("GBM 수행 시간: {0:.1f} 초 ".format(time.time() - start_time))
```

【Output】

```
GBM 정확도: 0.9376
GBM 수행 시간: 176.2 초
```

#4.5 GBM

하이퍼파라미터 튜닝

- loss: 경사 하강법에서 사용할 비용 함수를 지정합니다. 특별한 이유가 없으면 기본값인 'deviance'를 그대로 적용합니다.
- learning_rate: GBM이 학습을 진행할 때마다 적용하는 학습률입니다. Weak learner가 순차적으로 오류 값을 보정해 나가는 데 적용하는 계수입니다. 0~1 사이의 값을 지정할 수 있으며 기본값은 0.1입니다. 너무 작은 값을 적용하면 업데이트되는 값이 작아져서 최소 오류 값을 찾아 예측 성능이 높아질 가능성이 높습니다. 하지만 많은 weak learner는 순차적인 반복이 필요해서 수행 시간이 오래 걸리고, 또 너무 작게 설정하면 모든 weak learner의 반복이 완료돼도 최소 오류 값을 찾지 못할 수 있습니다. 반대로 큰 값을 적용하면 최소 오류 값을 찾지 못하고 그냥 지나쳐 버려 예측 성능이 떨어질 가능성이 높아지지만, 빠른 수행이 가능합니다.
- n_estimators: weak learner의 개수입니다. weak learner가 순차적으로 오류를 보정하므로 개수가 많을수록 예측 성능이 일정 수준까지는 좋아질 수 있습니다. 하지만 개수가 많을수록 수행 시간이 오래 걸립니다. 기본값은 100입니다.
- subsample: weak learner가 학습에 사용하는 데이터의 샘플링 비율입니다. 기본값은 1이며, 이는 전체 학습 데이터를 기반으로 학습한다는 의미입니다(0.5이면 학습 데이터의 50%). 과적합이 염려되는 경우 subsample을 1보다 작은 값으로 설정합니다.

#4.5 GBM

하이퍼파라미터 튜닝

```
from sklearn.model_selection import GridSearchCV

params = {
    'n_estimators':[100, 500],
    'learning_rate' : [ 0.05, 0.1]
}
grid_cv = GridSearchCV(gb_clf, param_grid=params, cv=2, verbose=1)
grid_cv.fit(X_train, y_train)
print('최적 하이퍼 파라미터:\n', grid_cv.best_params_)
print('최고 예측 정확도: {0:.4f}'.format(grid_cv.best_score_))
```

【Output】

```
최적 하이퍼 파라미터:
 {'learning_rate': 0.05, 'n_estimators': 500}
최고 예측 정확도: 0.9010
```

▲ **Hyperparameter Tuning 후 Training accuracy**

```
# GridSearchCV를 이용해 최적으로 학습된 estimator로 예측 수행.
gb_pred = grid_cv.best_estimator_.predict(X_test)
gb_accuracy = accuracy_score(y_test, gb_pred)
print('GBM 정확도: {0:.4f}'.format(gb_accuracy))
```

【Output】

```
GBM 정확도: 0.9410
```

▲ **Hyperparameter Tuning 후 Test accuracy**

4.6 XGBoost

EWHA EURON

#4.6 XGBoost

XGBoost의 장단점

장점
- Parallel 학습으로 예측 성능이 뛰어나다.
-다른 GBM 대비 수행 시간이 빠르다.

단점
- 다른 학습 모델에 비해 수행 시간이 빠른 것은 아니다.

항목	설명
뛰어난 예측 성능	일반적으로 분류와 회귀 영역에서 뛰어난 예측 성능을 발휘합니다.
GBM 대비 빠른 수행 시간	일반적인 GBM은 순차적으로 Weak learner가 가중치를 증감하는 방법으로 학습하기 때문에 전반적으로 속도가 느립니다. 하지만 XGBoost는 병렬 수행 및 다양한 기능으로 GBM에 비해 빠른 수행 성능을 보장합니다. 아쉽게도 XGBoost가 일반적인 GBM에 비해 수행 시간이 빠르다는 것이지, 다른 머신러닝 알고리즘(예를 들어 랜덤 포레스트)에 비해서 빠르다는 의미는 아닙니다.

항목	설명
과적합 규제 (Regularization)	표준 GBM의 경우 과적합 규제 기능이 없으나 XGBoost는 자체에 과적합 규제 기능으로 과적합에 좀 더 강한 내구성을 가질 수 있습니다.
Tree pruning (나무 가지치기)	일반적으로 GBM은 분할 시 부정 손실이 발생하면 분할을 더 이상 수행하지 않지만, 이러한 방식도 자칫 지나치게 많은 분할을 발생할 수 있습니다. 다른 GBM과 마찬가지로 XGBoost도 max_depth 파라미터로 분할 깊이를 조정하기도 하지만, tree pruning으로 더 이상 긍정 이득이 없는 분할을 가지치기 해서 분할 수를 더 줄이는 추가적인 장점을 가지고 있습니다.
자체 내장된 교차 검증	XGBoost는 반복 수행 시마다 내부적으로 학습 데이터 세트와 평가 데이터 세트에 대한 교차 검증을 수행해 최적화된 반복 수행 횟수를 가질 수 있습니다. 지정된 반복 횟수가 아니라 교차 검증을 통해 평가 데이터 세트의 평가 값이 최적화 되면 반복을 중간에 멈출 수 있는 조기 중단 기능이 있습니다.
결손값 자체 처리	XGBoost는 결손값을 자체 처리할 수 있는 기능을 가지고 있습니다.

#4.6 XGBoost

XGBoost 하이퍼파라미터

- 일반 파라미터 : 일반적으로 실행 시 스레드의 개수나 silent 모드 등의 선택을 위한 파라미터로서 값을 바꾸는 경우는 거의 없음

- 부스터 파라미터 : 트리 최적화, 부스팅, regularization 등과 관련된 파라미터

- 학습 태스크 파라미터 : 학습 수행 시의 객체 함수, 평가를 위한 지표 등을 설정하는 파라미터

주요 일반 파라미터

- booster: gbtree(tree based model) 또는 gblinear(linear model) 선택. 디폴트는 gbtree입니다.
- silent: 디폴트는 0이며, 출력 메시지를 나타내고 싶지 않을 경우 1로 설정합니다.
- nthread: CPU의 실행 스레드 개수를 조정하며, 디폴트는 CPU의 전체 스레드를 다 사용하는 것입니다. 멀티 코어/스레드 CPU 시스템에서 전체 CPU를 사용하지 않고 일부 CPU만 사용해 ML 애플리케이션을 구동하는 경우에 변경합니다.

#4.6 XGBoost

주요 부스터 파라미터

- eta [default=0.3, alias: learning_rate]: GBM의 학습률(learning rate)과 같은 파라미터입니다. 0에서 1 사이의 값을 지정하며 부스팅 스텝을 반복적으로 수행할 때 업데이트되는 학습률 값. 파이썬 래퍼 기반의 xgboost를 이용할 경우 디폴트는 0.3. 사이킷런 래퍼 클래스를 이용할 경우 eta는 learning_rate 파라미터로 대체되며, 디폴트는 0.1입니다. 보통은 0.01 ~ 0.2 사이의 값을 선호합니다.
- num_boost_rounds: GBM의 n_estimators와 같은 파라미터입니다.
- min_child_weight[default=1]: 트리에서 추가적으로 가지를 나눌지를 결정하기 위해 필요한 데이터들의 weight 총합. min_child_weight이 클수록 분할을 자제합니다. 과적합을 조절하기 위해 사용됩니다.
- gamma [default=0, alias: min_split_loss]: 트리의 리프 노드를 추가적으로 나눌지를 결정할 최소 손실 감소 값입니다. 해당 값보다 큰 손실(loss)이 감소된 경우에 리프 노드를 분리합니다. 값이 클수록 과적합 감소 효과가 있습니다.
- max_depth[default=6]: 트리 기반 알고리즘의 max_depth와 같습니다. 0을 지정하면 깊이에 제한이 없습니다. Max_depth가 높으면 특정 피처 조건에 특화되어 룰 조건이 만들어지므로 과적합 가능성이 높아지며 보통은 3~10 사이의 값을 적용합니다.
- sub_sample[default=1]: GBM의 subsample과 동일합니다. 트리가 커져서 과적합되는 것을 제어하기 위해 데이터를 샘플링하는 비율을 지정합니다. sub_sample=0.5로 지정하면 전체 데이터의 절반을 트리를 생성하는 데 사용합니다. 0에서 1사이의 값이 가능하나 일반적으로 0.5 ~ 1 사이의 값을 사용합니다.

#4.6 XGBoost

XGBoost 하이퍼파라미터

Overfitting 방지를 위해 하이퍼파라미터를 조정할 때

<감소시킬 하이퍼파라미터 >
eta, max_depth

<증가시킬 하이퍼파라미터>
min_child_weight, gamma

<그 외 >
subsample, colsample_bytree 조정하여 트리 복잡도 낮추기

#4.6 XGBoost

위스콘신 유방암 예측 실습 (파이썬 Wrapper)

Step 1. 파이썬 패키지에서 xgboost 와 필요한 모듈 임포트, 데이터 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')

dataset = load_breast_cancer()
X_features= dataset.data
y_label = dataset.target
```

```
cancer_df = pd.DataFrame(data=X_features, columns=dataset.feature_names)
cancer_df['target']= y_label
cancer_df.head(3)
```

【Output】

	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	c
0	17.99	10.38	122.8	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	0.07871	...	17.33	184.6	2019.0	0.1622	
1	20.57	17.77	132.9	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	0.05667	...	23.41	158.8	1956.0	0.1238	
2	19.69	21.25	130.0	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	0.05999	...	25.53	152.5	1709.0	0.1444	

#4.6 XGBoost

위스콘신 유방암 예측 실습 (파이썬 Wrapper)

Step 2. Train data, Test data 나누기

```
# 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 )
print(X_train.shape, X_test.shape)
```

【Output】

```
(455, 30) (114, 30)
```

#4.6 XGBoost

위스콘신 유방암 예측 실습 (파이썬 Wrapper)

Step 3. Dmatrix 객체 생성

```
dtrain = xgb.DMatrix(data=X_train, label=y_train)
dtest = xgb.DMatrix(data=X_test, label=y_test)
```

Step 4. 하이퍼파라미터 설정

```
params = { 'max_depth':3,
           'eta': 0.1,
           'objective':'binary:logistic',
           'eval_metric':'logloss',
           'early_stoppings':100
        }
num_rounds = 400
```

#4.6 XGBoost

위스콘신 유방암 예측 실습 (파이썬 Wrapper)

Step 5. 학습 및 결과 확인

```
# train 데이터 세트는 'train', evaluation(test) 데이터 세트는 'eval'로 명기합니다.
wlist = [(dtrain, 'train'), (dtest, 'eval') ]
# 하이퍼 파라미터와 early stopping 파라미터를 train() 함수의 파라미터로 전달
xgb_model = xgb.train(params = params, dtrain=dtrain, num_boost_round=num_rounds, \
                      early_stopping_rounds=100, evals=wlist )
```

【Output】

```
[0] train-logloss:0.609688 eval-logloss:0.61352
Multiple eval metrics have been passed: 'eval-logloss' will be used for early stopping.

Will train until eval-logloss hasn't improved in 100 rounds.
[1] train-logloss:0.540803 eval-logloss:0.547842
[2] train-logloss:0.483753 eval-logloss:0.494247
.....
[311] train-logloss:0.00545 eval-logloss:0.085948
Stopping. Best iteration:
[211] train-logloss:0.006413 eval-logloss:0.085593
```

```
# 예측 확률이 0.5보다 크면 1, 그렇지 않으면 0으로 예측값 결정해 리스트 객체인 preds에 저장
preds = [ 1 if x > 0.5 else 0 for x in pred_probs ]
print('예측값 10개만 표시:', preds[:10])
```

【Output】

```
predict( ) 수행 결과값을 10개만 표시, 예측 확률 값으로 표시됨
[0.934 0.003 0.91  0.094 0.993 1.    1.    0.999 0.997 0.   ]
예측값 10개만 표시: [1, 0, 1, 0, 1, 1, 1, 1, 1, 0]
```

```
get_clf_eval(y_test , preds, pred_probs)
```

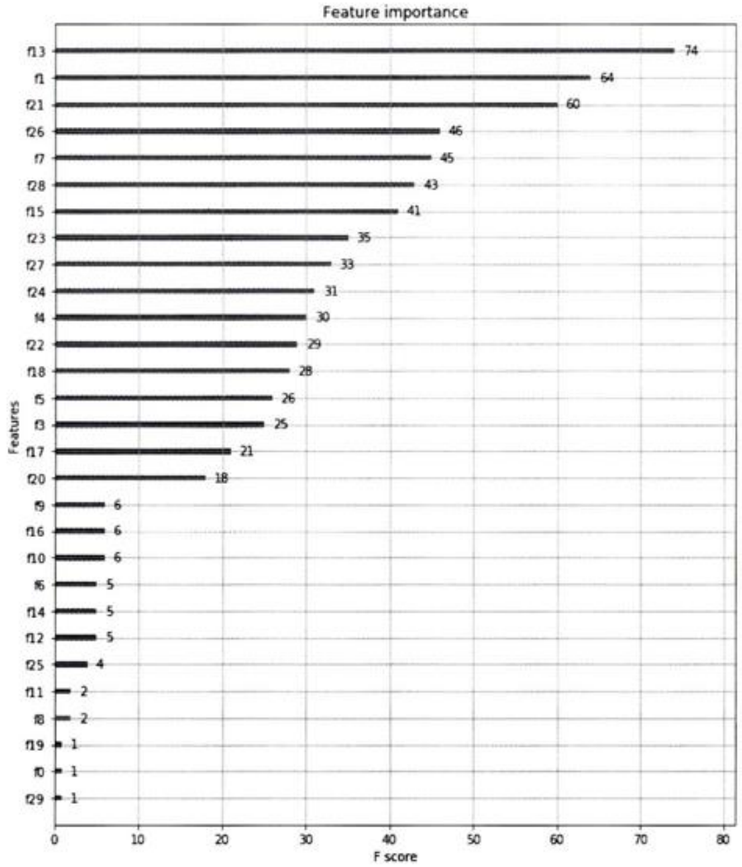
【Output】

```
오차 행렬
[[35  2]
 [ 1 76]]
정확도: 0.9737, 정밀도: 0.9744, 재현율: 0.9870, F1: 0.9806, AUC:0.9951
```

#4.6 XGBoost

```
from xgboost import plot_importance
import matplotlib.pyplot as plt
%matplotlib inline

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



#4.6 XGBoost

위스콘신 유방암 예측 실습 (사이킷런 Wrapper)

파이썬 XGBoost vs 사이킷런 XGBoost

📌 하이퍼파라미터가 다르니 주의

- eta → learning_rate
- sub_sample → subsample
- lambda → reg_lambda
- alpha → reg_alpha

#4.6 XGBoost

위스콘신 유방암 예측 실습 (사이킷런 Wrapper)

학습 및 결과 확인 -> 파이썬 Wrapper 와 결과 동일

```
# 사이킷런 래퍼 XGBoost 클래스인 XGBClassifier 임포트
from xgboost import XGBClassifier

xgb_wrapper = XGBClassifier(n_estimators=400, learning_rate=0.1, max_depth=3)
xgb_wrapper.fit(X_train, y_train)
w_preds = xgb_wrapper.predict(X_test)
w_pred_proba = xgb_wrapper.predict_proba(X_test)[:, 1]
```

```
get_clf_eval(y_test, w_preds, w_pred_proba)
```

【Output】

```
오차 행렬
[[35  2]
 [ 1 76]]
정확도: 0.9737, 정밀도: 0.9744, 재현율: 0.9870, F1: 0.9806, AUC:0.9951
```

4.7 LightGBM

EWHA EURON

#4.7 LightGBM

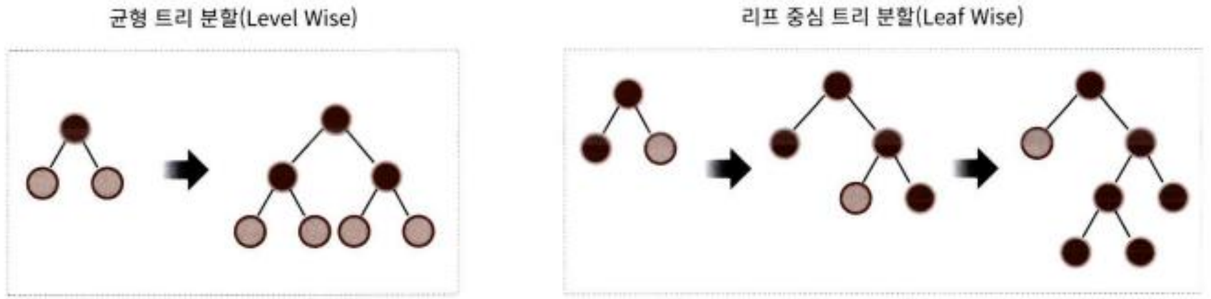
LightGBM

- XGBoost와 함께 부스팅 계열 알고리즘에서 가장 각광받고 있는 알고리즘

<장단점>

- XGBoost보다 학습 시간이 짧고 메모리 사용량도 적지만 성능은 별다른 차이가 없다
- 적은 데이터 세트(10,000건 이하)에 적용하면 과적합이 발생하기 쉽다
- 카테고리형 피처의 자동 변환과 최적 분할

- 리프 중심 트리 분할(Leaf Wise) 방식 사용



#4.7 LightGBM

LightGBM 하이퍼 파라미터

주요 파라미터

· **num_iterations**[default = 100] : 반복 수행하려는 트리의 개수를 지정하는 파라미터로 이 값을 크게 지정할수록 예측 성능이 높아질 수 있지만 너무 크게 지정하면 과적합으로 성능이 저하될 수 있음

· **learning_rate** [default = 0.1] : 0에서 1 사이의 값을 지정하며 부스팅 스텝을 반복적으로 수행할 때 업데이트 되는 학습률 값

· **max_depth** [default = 1] : 트리 기반 알고리즘의 max_depth와 같은 파라미터로 깊이에 제한을 둠

· **min_data_in_leaf** [default = 20] : 최종 결정 클래스인 리프 노드가 되기 위한 최소한의 레코드 수로 과적합 제어

· **num_leaves** [default = 31] : 하나의 트리가 가질 수 있는 최대 리프 개수

· **boosting [default** = gbdt] : 부스팅의 트리를 생성하는 알고리즘 기술
- gbdt : 일반적인 그래디언트 부스팅 결정 트리 / rf : 랜덤 포레스트

#4.7 LightGBM

주요 파라미터

· **bagging_fraction** [default = 1.0] : 데이터 샘플링 비율

· **feature_fraction** [default = 1.0] : 개별 트리를 학습할 때마다 무작위로 선택하는 피처의 비율

· **lambda_l2** [default = 0.0] : L2 regulation 제어, 피처 개수가 많을 경우 적용을 검토하며 값이 클수록 과적합 감소 효과가 있음

· **lambda_l1** [default = 0.0] : L1 regulation 제어

Learning Task 파라미터

· **objective** : 최솟값을 가져야 할 손실함수 정의, 애플리케이션 유형(회귀, 다중 클래스 분류, 이진 분류)에 따라 손실함수가 지정됨

4.7 LightGBM

하이퍼 파라미터 튜닝 방안

기본 튜닝 방안

num_leaves의 개수를 중심으로 min_child_samples(min_data_in_leaf), max_depth를 함께 조정하며 모델의 복잡도를 줄인다

Learning_rate를 작게 하면서 n_estimators를 크게 한다

📍 **n_estimators**를 너무 크게 하면 과적합으로 성능이 떨어질 수 있다

이밖에도 reg_lambda, reg_alpha와 같은 regularization을 적용하거나 학습 데이터에 사용할 피처의 개수나 데이터 샘플링 레코드 개수를 줄이는 파라미터를 적용하여 과적합을 제어할 수 있다

#4.7 LightGBM

파이썬 래퍼 LightGBM과 사이킷런 래퍼 XGBoost, LightGBM 하이퍼 파라미터 비교

- LightGBM이 사이킷런과 호환하기 위해 분류를 위한 LGBMClassifier와 회귀를 위한 LGBMRegressor 클래스를 래퍼 클래스로 생성함
- LightGBM과 XGBoost는 많은 유사한 기능이 있었기 때문에 둘의 사이킷런 래퍼 클래스는 많은 하이퍼 파라미터가 동일함

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

#4.7 LightGBM

LightGBM 실습 - 위스콘신 유방암 예측

Step 1. 파이썬 패키지 lightgbm에서 LGBMClassifier를 임포트

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

Step 2. breast_cancer 데이터셋 load 후 target 분리

```
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()
ftr = dataset.data
target = dataset.target
```

#4.7 LightGBM

LightGBM 실습 - 위스콘신 유방암 예측

Step 3. 학습용 데이터 80%, 테스트용 데이터 20% 추출

```
X_train, X_test, y_train, y_test=train_test_split(ftr, target, test_size=0.2, random_state=156 )
```

Step 4. n_estimators = 400 으로 설정하여 모델 학습 및 예측 수행

```
# 앞서 XGBoost와 동일하게 n_estimators는 400 설정.
lgbm_wrapper = LGBMClassifier(n_estimators=400)

# LightGBM도 XGBoost와 동일하게 조기 중단 수행 가능.
evals = [(X_test, y_test)]
lgbm_wrapper.fit(X_train, y_train, early_stopping_rounds=100, eval_metric="logloss",
                 eval_set=evals, verbose=True)
preds = lgbm_wrapper.predict(X_test)
pred_proba = lgbm_wrapper.predict_proba(X_test)[:, 1]
```

【Output】

```
[1] valid_0's binary_logloss: 0.565079      valid_0's binary_logloss: 0.565079
Training until validation scores don't improve for 100 rounds.
[2] valid_0's binary_logloss: 0.507451      valid_0's binary_logloss: 0.507451
.....
[147]   valid_0's binary_logloss: 0.192769   valid_0's binary_logloss: 0.192769
Early stopping, best iteration is:
[47]    valid_0's binary_logloss: 0.126108   valid_0's binary_logloss: 0.126108
```

EWHA EURON

#4.7 LightGBM

LightGBM 실습 - 위스콘신 유방암 예측

Step 5. get_clf_eval( ) 함수를 이용해 예측 성능 평가

```
get_clf_eval(y_test, preds, pred_proba)
```

【Output】

```
오차 행렬
[[33  4]
 [ 2 75]]
정확도: 0.9474, 정밀도: 0.9494, 재현율: 0.9740, F1: 0.9615, AUC:0.9926
```

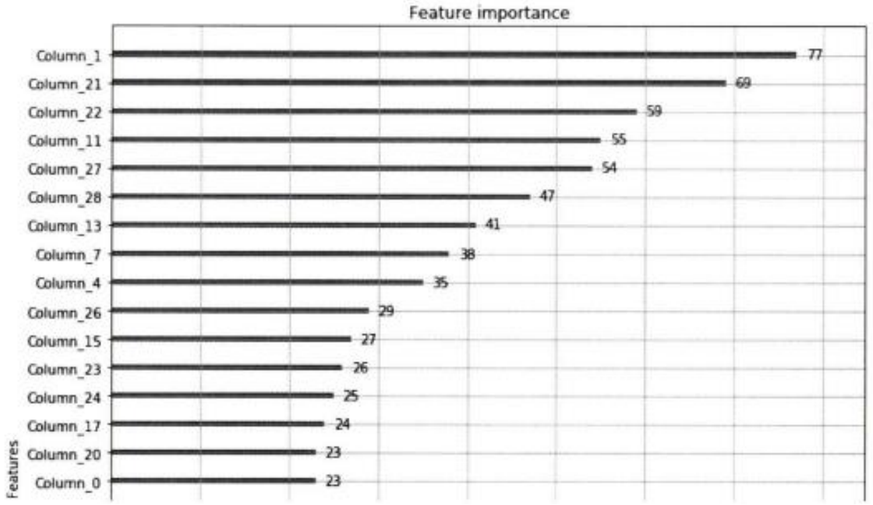
#4.7 LightGBM

LightGBM 실습 - 위스콘신 유방암 예측

Step 6. plot_importance( )로 피처 중요도 시각화

```
# plot_importance( )를 이용해 피처 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)
```



4.10 스태킹 앙상블

#4.10 스태킹 앙상블

배깅(Bagging)

- 여러 개의 분류기가 투표를 통해 최종 예측 결과 결정 방식
- 각각의 분류기가 모두 같은 유형의 알고리즘 기반이지만, 데이터 샘플링을 서로 다르게 가져가면서 학습을 수행해 보팅을 수행

부스팅(Boosting)

- 여러 개의 분류기가 순차적으로 학습
- 앞에서 학습한 분류기가 예측이 틀린 데이터에 대해 다음 분류기에 가중치(weight) 부여

스태킹(Stacking)

- 개별적인 여러 알고리즘을 서로 결합해 예측 결과를 도출
- 배깅 및 부스팅과 다르게 개별 알고리즘으로 예측한 데이터를 기반으로 다시 예측을 수행

→ 개별 알고리즘의 예측 결과 데이터 세트를 최종 메타 데이터 세트로 만들어 별도의 ML 알고리즘으로 최종 학습 수행 후 테스트 데이터를 기반으로 다시 최종 예측 수행

#4.10 스태킹 앙상블

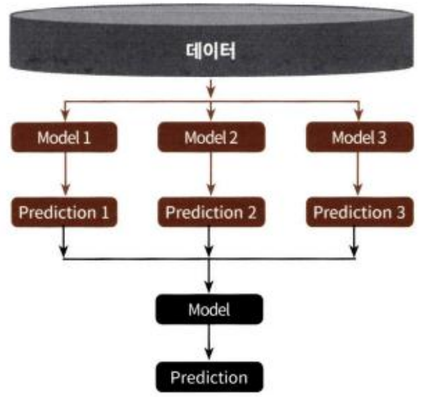
스태킹 모델에 필요한 두 모델

1. 개별적인 기반 모델
2. 최종 메타 모델

* 메타 모델 : 개별 모델의 예측된 데이터 세트를 다시 기반으로 하여 학습하고 예측하는 방식

- 여러 개별 모델의 예측 데이터를 각각 스태킹 형태로 결합해 최종 메타 모델의 학습용 피처 데이터 세트와 테스트용 피처 데이트 세트를 만드는 것이 핵심

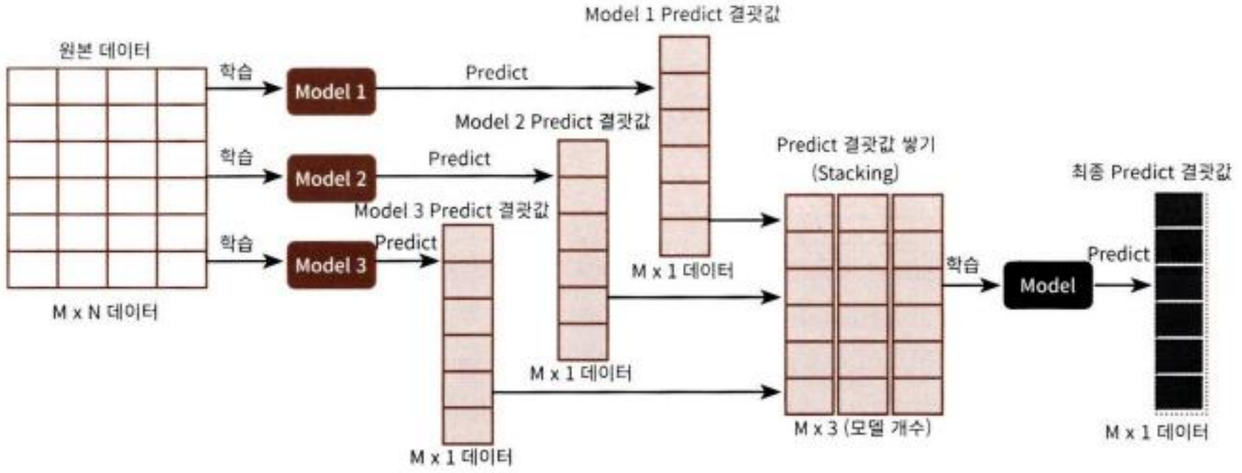
- 여러 개의 모델에 대한 예측값을 합한 후, 이에 대한 예측을 다시 수행



#4.10 스태킹 앙상블

스태킹 앙상블 모델

- M x N 데이터셋
- 3개의 모델에 학습 후 예측
- M x 1 Predict 결괏값 도출
- 각 모델의 Predict 결괏값을 하나의 M x 3(모델 개수)로 다시 합해서(스태킹) 새로운 데이터 세트 생성
- 최종 모델에 이 데이터 세트를 적용해 최종 예측



#4.10 스태킹 앙상블

기본 스태킹 모델

Step 1. 위스콘신 암 데이터 세트를 로딩 후 학습/테스트 데이터 세트로 분리

```
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_data, y_label, test_size=0.2, random_state=0)
```

Step 2. 스태킹에는 여러 모델이 필요하므로 사용될 머신러닝 알고리즘 클래스 생성

개별 모델 : KNN, 랜덤 포레스트, 결정 트리, 에이다부스트

최종 모델 : 로지스틱 회귀

```
# 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)

# 스태킹으로 만들어진 데이터 세트를 학습, 예측할 최종 모델
lr_final = LogisticRegression(C=10)
```

#4.10 스태킹 앙상블

기본 스태킹 모델

Step 3. 개별 모델 학습 및 예측

```
# 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 세트를 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)
gbm_pred = gbm_clf.predict(X_test)
```

Step 4. 예측 정확도 측정

```
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)))
```

【Output】

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

#4.10 스태킹 앙상블

기본 스태킹 모델

Step 5. 반환된 예측 데이터 세트를 행 형태로 붙인 후 numpy의 transpose( )를 이용해 행과 열 위치를 바꾼 ndarray로 변환

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

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

【Output】

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

Step 6. 예측 데이터로 생성된 데이터 세트로 최종 메타 모델(로지스틱 회귀) 학습 및 예측 정확도 측정

```
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)))
```

【Output】

```
최종 메타 모델의 예측 정확도: 0.9737
```

EWHA EURON

#4.10 스태킹 앙상블

CV 세트 기반의 스태킹

- 과적합을 개선하기 위해 최종 메타 모델을 위한 데이터 세트를 만들 때 **교차 검증 기반**으로 예측된 결과 데이터 세트를 이용

Step 1. 각 모델별로 원본 학습/테스트 데이터를 예측한 결과 값을 기반으로 메타 모델을 위한 학습용/테스트용 데이터를 생성

Step 2. Step 1에서 개별 모델들이 생성한 학습용/테스트용 데이터를 각각 모두 스태킹 형태로 합쳐 메타 모델이 학습할 최종 학습용/테스트용 데이터 세트 생성

#4.10 스태킹 앙상블

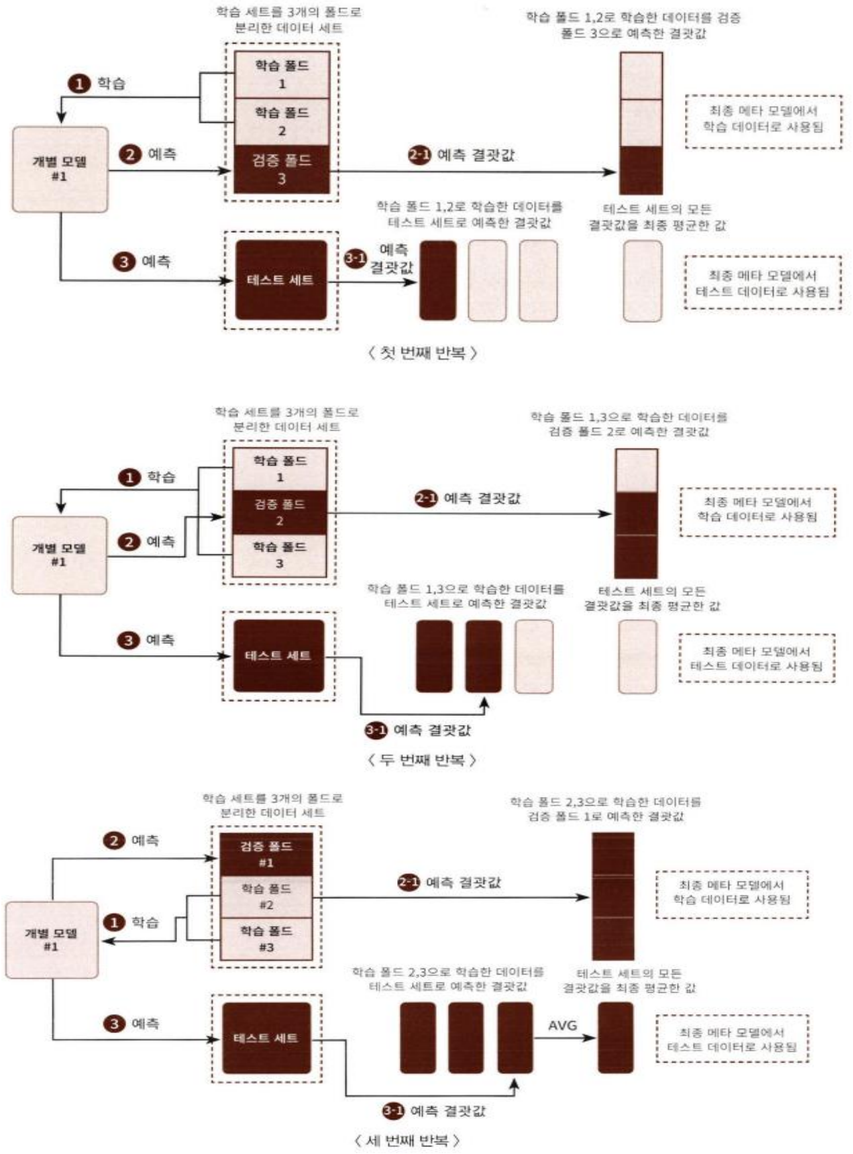
Step 1

1. 학습용 데이터를 N개의 폴드(Fold)로 나눈다.
   (학습용 : N-1, 검증용 : 1)

2. 나뉜 학습 데이터를 기반으로 개별 모델 학습 후 검증 폴드 1개 데이터로 예측하고 그 결과 저장 -> N번 반복

3. 이 결과 생성된 예측 데이터는 메타 모델의 학습 데이터로 사용

4. N-1개의 폴드 데이터로 학습된 개별 모델은 원본 테스트 데이터를 예측하여 예측값을 생성하는데 이 예측을 N번 반복하며 예측값의 평균으로 최종 결괏값을 생성하고 메타 모델을 위한 테스트 데이터로 사용

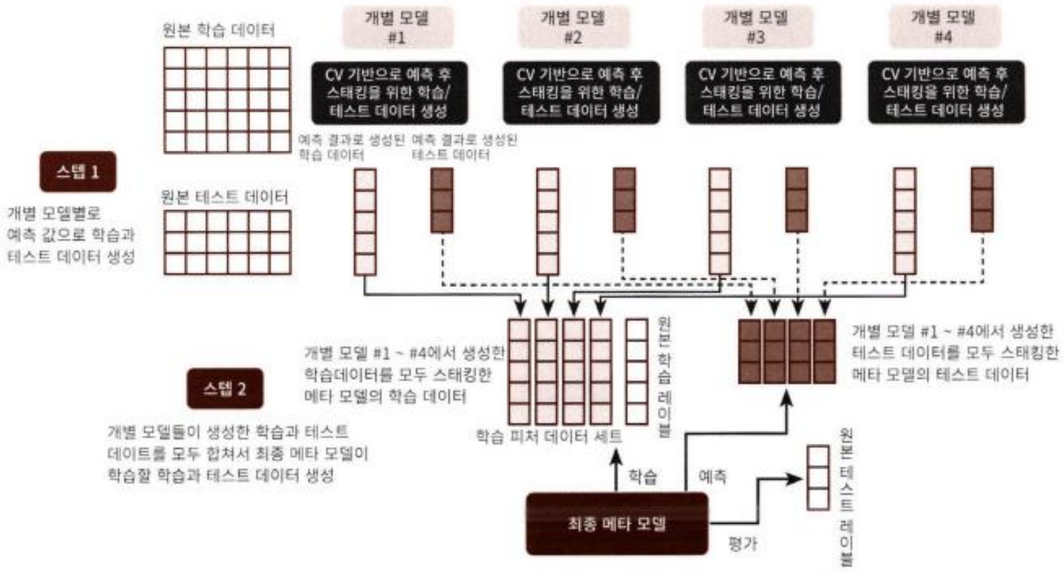


#4.10 스태킹 앙상블

Step 2

1. 각 모델들이 step 1로 생성한 학습과 테스트 데이터를 모두 합쳐 메타 모델이 사용할 학습 데이터와 테스트 데이터를 생성

2. 생성한 데이터로 메타 모델을 학습 및 예측을 수행한 후 최종 예측 결과를 원본 테스트 데이터의 레이블 데이터와 비교해 평가



#4.10 스태킹 앙상블

CV 세트 기반의 스태킹 코드 구현

Step 1

- get_stacking_base_datasets( ) 함수 생성

- parameters : 개별 모델의 Classifier 객체, 원본 학습용 피처 데이터, 원본 학습용 레이블 데이터, 원본 테스트 피처 데이터, 폴드 개수

- 폴드의 개수만큼 반복을 수행하며 학습용 데이터로 학습 후 예측 결괏값을 기반으로 메타 모델을 위한 학습/테스트용 데이터 생성

```
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, random_state=0)
    # 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 세트 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
```

#4.10 스태킹 앙상블

- 개별 모델별로 각각 메타 모델이 추후에 사용할 학습용/테스트용 데이터 세트 반환

```
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)
```

Step 2

- numpy의 concatenate( )를 이용해 하나의 학습용/테스트용 데이터로 합침

```
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)
```

【Output】

```
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)
```

#4.10 스태킹 앙상블

- Stack_final_X_train : 메타 모델이 학습할 학습용 피처 데이터 세트
- Stack_final_X_test : 메타 모델이 예측할 테스트용 피처 데이터 세트

- 최종 메타 모델은 스태킹된 학습용 데이터 세트와 원본 학습 레이블 데이터로 학습한 후, 스태킹된 테스트 데이터 세트로 예측하고 예측 결과를 원본 테스트 레이블 데이터와 비교하여 정확도 측정

```
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)))
```

Output】

```
최종 메타 모델의 예측 정확도: 0.9737
```

4.8 HyperOpt

EWHA EURON

베이지안 최적화

```
params={
    'max_depth'= [10,20,30,40,50],
    'num_leaves'= [35,45,55,65],
    'colsample_bytree'= [0.5,0.6,0.7,0.8,0.9,1.0],
    'subsample'= [0.5,0.6,0.7,0.8,0.9,1.0],
    'max_bin'= [100,200,300,400]
    'min_child_weight'= [10,20,30,40]
}
```

5*4*6*6*4*3= 11520

LightGBM의 6가지 하이퍼 파라미터를 최적화하려는 시도

GridSearchCV

수행 시간이 너무 오래 걸림
개별 하이퍼 파라미터들을 Grid 형태로 지정하는 것은 한계

RandomizedSearch

GridSearchCV를 랜덤하게 수행-> 수행 시간은 줄여줌
Random한 선택으로 최적 하이퍼 파라미터 검출에 태생적 제약

베이지안 최적화 필요!

베이지안 최적화

베이지안 최적화

- 베이지안 확률에 기반을 두고 있는 최적화 기법
- 최적 함수를 예측하는 사후 모델을 개선해 나가며 사후 모델을 만듦

주요 요소

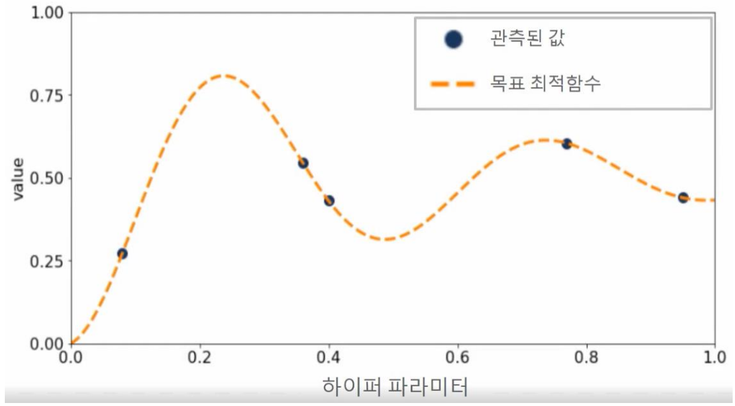
대체 모델(Surrogate Model)

획득 함수로부터
최적 함수를 예측할 수 있는 입력값을 추천받고
이를 기반으로 최적 함수 모델 개선

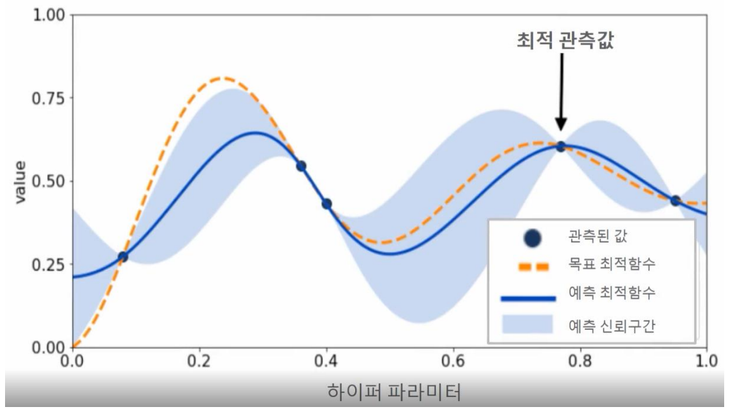
획득 함수(Acquisition Function)

개선된 대체 모델을 기반으로
최적 입력값을 계산

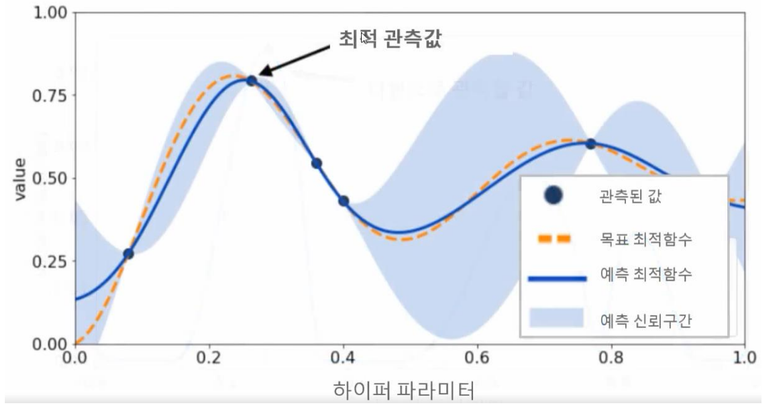
베이지안 최적화



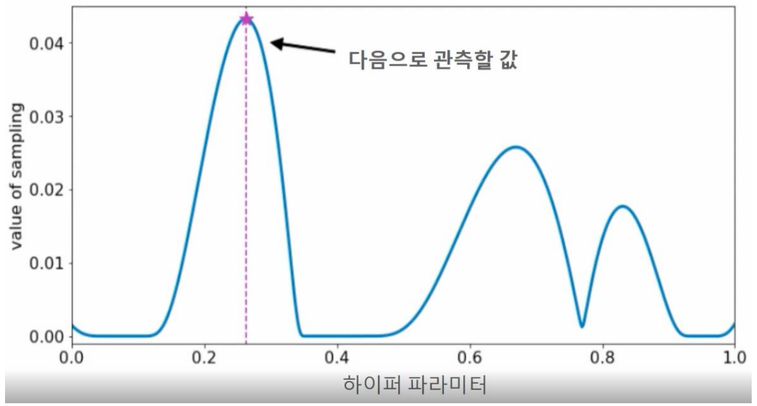
Step.1
랜덤하게 하이퍼 파라미터를 샘플링하고 성능 결과 관측



Step.2
관측된 값을 기반으로 대체 모델은 최적 함수 및 신뢰 구간을 추정



Step.3
추정된 최적 함수를 기반으로 획득 함수에서
다음으로 관측할 하이퍼 파라미터 추천



Step.4
획득 함수로부터 전달된 하이퍼 파라미터를 수행하여
관측된 값을 기반으로 대체 모델 다시 갱신
→ 다시 최적 함수를 예측 추정

HyperOpt

베이지안 최적화를 머신러닝 모델의 하이퍼 파라미터 튜닝에 적용할 수 있게 제공되는 파이썬 패키지 중 하나

설치: pip install hyperopt

검색 공간 설정

```
from hyperopt import hp

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

hp.quniform(label, low, high, q)	label로 지정된 입력 변수명 검색 공간을 최솟값 low에서 최대값 high까지 q의 간격을 가지고 설정
hp.uniform(label, low, high)	최솟값 low에서 최대값 high까지 정규 분포 형태의 검색 공간 설정
hp.randint(label, upper)	0부터 최대값 upper까지 random한 정수값으로 검색 공간 설정
hp.loguniform(label, low, high)	exp(uniform(low, high)값을 반환하며, 반환 값의 log 변환된 값은 정규 분포 형태를 가지는 검색 공간 설정
hp.choice(label, options)	검색 값이 문자열 또는 문자열과 숫자값이 섞여 있을 경우 설정. options는 리스트나 튜플 형태로 제공되며 hp.choice('tree_criterion', ['gini', 'entropy'])와 같이 설정하면 입력 변수 tree_criterion의 값을 'gini'와 'entropy'로 설정하여 입력함

HyperOpt

목적 함수 설정

```
from hyperopt import STATUS_OK

## 목적 함수 생성. 변수값과 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 # retval = x**2-20*y로 계산된 값을 반환하게 설정한 이유는 예제를
위해서 임의로 만든것


  return retval
```

HyperOpt

최적 입력 값 찾기

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

fn	위에서 생성한 objective_func와 같은 목적 함수
space	위에서 생성한 search_space와 같은 검색 공간 딕셔너리
algo	베이지안 최적화 적용 알고리즘
max_evals	최적 입력값을 찾기 위한 입력값 시도 횟수
trials	최적 입력값을 찾기 위해 시도한 입력값 및 해당 입력값의 목적 함수 반환값 결과를 저장하는데 사용. Trials 클래스를 객체로 생성한 변수명을 입력함
Rstate	fmin()을 수행할 때마다 동일한 결과값을 가질 수 있도록 설정하는 랜덤 시드 값

HyperOpt

최적 입력 값 찾기

```
from hyperopt import fmin, tpe, Trials
import numpy as np

# 입력값 및 해당 입력값의 목적 함수 반환값 결과를 저장할 Trials 객체값 생성
trial_val = Trials()

# 목적 함수의 반환값이 최솟값을 가지는 최적 입력값을 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, 174.63trial/s, best loss: -224.0]
best: {'x': -4.0, 'y': 12.0}
```

```
trial_val = Trials()

# max_evals를 20회로 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, 557.03trial/s, best loss: -296.0]
best: {'x': 2.0, 'y': 15.0}
```

HyperOpt

*Trials 객체의 중요 속성

```
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, 'stat
us': 'ok'}, {'loss': 64.0, 'status': 'ok'}, {'loss': 100.0, 'status': 'ok'}, {'loss': 60.0, 'status': 'ok'}, {'loss': -39.0, 's
tatus': 'ok'}, {'loss': 1.0, 'status': 'ok'}, {'loss': -164.0, 'status': 'ok'}, {'loss': 21.0, 'status': 'ok'}, {'loss': -56.0,
'status': 'ok'}, {'loss': 284.0, 'status': 'ok'}, {'loss': 176.0, 'status': 'ok'}, {'loss': -171.0, 'status': 'ok'}, {'loss':
0.0, 'status': 'ok'}]
```

```
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]}
```

```
import pandas as pd

# result에서 loss 키값에 해당하는 value를 추출하여 list로 생성
losses = [loss_dict['loss'] for loss_dict in trial_val.results]

# DataFrame으로 생성
result_df = pd.DataFrame({'x':trial_val.vals['x'], 'y':trial_val.vals['y'], 'losses':los
ses})
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
5	2.0	15.0	-296.0
6	10.0	7.0	-40.0
7	-9.0	-10.0	281.0
8	-8.0	0.0	64.0
9	-0.0	-5.0	100.0
10	-0.0	-3.0	60.0
11	1.0	2.0	-39.0
12	9.0	4.0	1.0
13	6.0	10.0	-164.0
14	9.0	3.0	21.0
15	2.0	3.0	-56.0
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

HyperOpt를 이용한 XGBoost 하이퍼 파라미터 튜닝

주의!

- 정수형 하이퍼 파라미터 입력 시 형변환 필요
- 정확도와 같이 값이 클수록 좋은 성능 지표일 경우, -1을 곱한 뒤 반환
  회귀 함수의 경우 값이 작을수록 좋은 성능 지표이므로 곱하지 않음

필요한 패키지 및 모듈 불러오기

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split

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

데이터 로드

```
# 유방암 데이터셋
dataset = load_breast_cancer()

cancer_df = pd.DataFrame(data=dataset.data, columns=dataset.feature_names)
cancer_df['target'] = dataset.target
cancer_df.head()
```

HyperOpt를 이용한 XGBoost 하이퍼 파라미터 튜닝

데이터 분리

```
# 피처와 타겟 분리
X_features = cancer_df.iloc[:, :-1]
y_label = cancer_df.iloc[:, -1]
```

```
# 전체 데이터 중 80%는 학습용 데이터, 20%는 테스트용 데이터 분리
X_train, X_test, y_train, y_test = train_test_split(X_features, y_label, test_size=0.2,
random_state=156)

# 학습 데이터를 다시 학습과 검증 데이터로 분리(학습:9 / 검증:1)
X_tr, X_val, y_tr, y_val = train_test_split(X_train, y_train, test_size=0.1, random_stat
e=156)
```

모델 성능 평가 함수 선언

```
from sklearn.metrics import
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))
```

HyperOpt를 이용한 XGBoost 하이퍼 파라미터 튜닝

검색 공간 설정

```
# HyperOpt 검색 공간 설정

from hyperopt import hp

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)}
```

목적 함수 설정

```
# HyperOpt 목적 함수 설정
from sklearn.model_selection import cross_val_score

# 교차검증
from xgboost import XGBClassifier
from hyperopt import STATUS_OK

def objective_func(search_space):
    xgb_clf = XGBClassifier(
        n_estimators=100,
        max_depth=int(search_space['max_depth']),# 정수형 하이퍼 파라미터 형변환 필요:int형
        min_child_weight=int(search_space['min_child_weight']),
        # 정수형 하이퍼 파라미터 형변환 필요:int형
        learning_rate=search_space['learning_rate'],
        colsample_bytree=search_space['colsample_bytree'],
        eval_metric='logloss')

# 목적 함수의 반환값은 교차검증 기반의 평균 정확도 사용
    accuracy = cross_val_score(xgb_clf, X_train, y_train, scoring='accuracy', cv=3)

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

HyperOpt를 이용한 XGBoost 하이퍼 파라미터 튜닝

fmin()을 이용해 최적 하이퍼 파라미터 도출

```
# HyperOpt fmin()을 이용해 최적 하이퍼 파라미터 도출
from hyperopt import fmin, tpe, Trials

trial_val = Trials()

best = fmin(fn=objective_func,
            space=xgb_search_space,
            algo=tpe.suggest,
            max_evals=50,# 입력값 시도 횟수 지정
            trials=trial_val, rstate=np.random.default_rng(seed=9))

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

```
100%|██████████████████████████████████████████████████| 50/50 [00:11<00:00,  4.45trial/s, best loss: -0.967047170907401]
best: {'colsample_bytree': 0.5818531121748992, 'learning_rate': 0.15280987770873294, 'max_depth': 5.0, 'min_child_weight': 2.0}
```

HyperOpt를 이용한 XGBoost 하이퍼 파라미터 튜닝

XGBoost의 인자로 입력하여 학습 및 평가

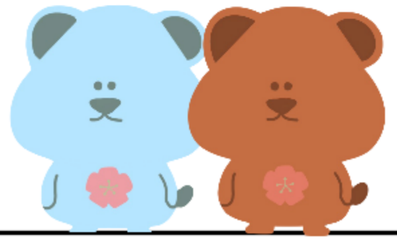
추출된 최적 하이퍼 파라미터를 이용
*정수형 하이퍼 파라미터(max_depth, min_child_weight) 값이 실수형으로 도출됨을 유의

```
# 도출된 최적 하이퍼 파라미터를 이용하여 모델 선언
xgb_wrapper = XGBClassifier(n_estimators=400,
                            learning_rate=round(best['learning_rate'], 5),
                            max_depth=int(best['max_depth']),# 형변환
                            min_child_weight=int(best['min_child_weight']),# 형변환
                            colsample_bytree=round(best['colsample_bytree'], 5)
                            )
# 모델 학습: 조기 중단(early stopping) - 50
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)

# 예측
preds = xgb_wrapper.predict(X_test)
pred_proba = xgb_wrapper.predict_proba(X_test)[:, 1]

# 모델 평가
get_clf_eval(y_test, preds, pred_proba)
```

THANK YOU



EWHA EURON