3️⃣

분류2

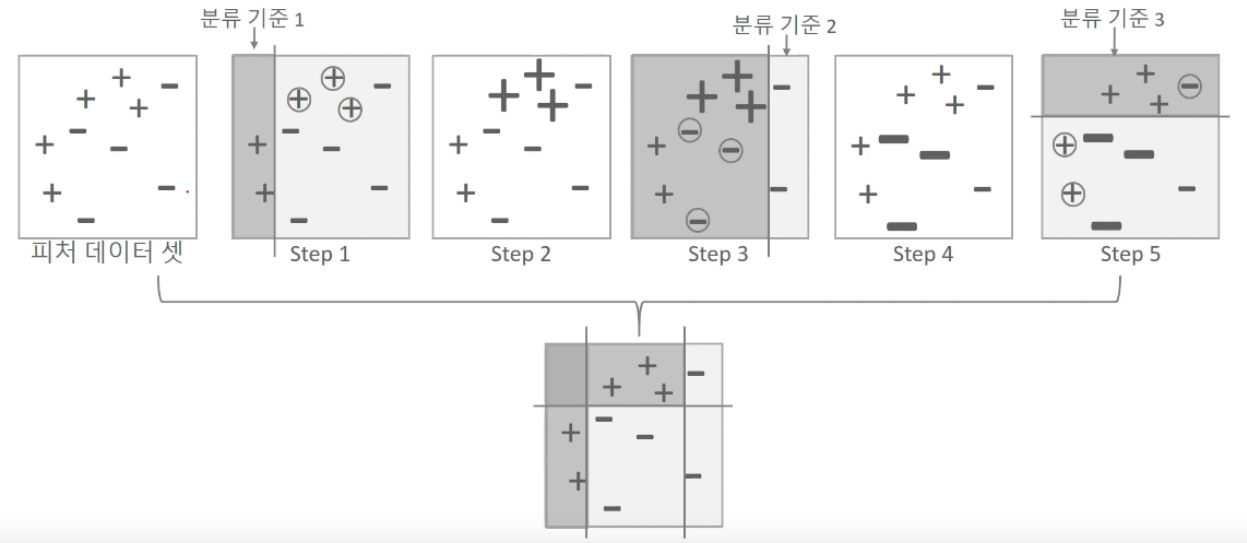
05. GBM(Gradient Boosting Machine)

GBM의 개요 및 실습

부스팅 알고리즘은 여러 개의 약한 학습기를 순차적으로 학습-예측 하면서 잘못 예측한 데이터에 가중치 부여를 통해 오류를 개선해 나가는 학습 방식.

→ 부스팅의 대표적인 구현: AdaBoost, 그래디언트 부스트

- 에이다 부스트: 오류 데이터에 가중치를 부여하면서 부스팅 수행하는 대표적인 알고리즘



https://asthtls.tistory.com/1275?category=904088

맨 왼쪽 그림과 같이 +와 -로 된 피처 데이터 세트가 있다면

1. 첫 번째 약한 학습기가 분류 기준 1로 +와 -를 분류함. 동그라미로 표시된 + 데이터는 + 데이터가 잘못 분류된 오류 데이터.
2. 이 오류 데이터에 대해 가중치 값 부여. 가중치가 부여된 오류 + 데이터는 다음 약한 학습기가 더 잘 분류할 수 있도록 크기가 커짐.
3. 두 번째 약한 학습기가 분류 기준 2로 + 와 - 분류. 마찬가지로 동그라미 - 데이터는 오류 데이터.

4. 잘못 분류된 - 오류 데이터에 대해 다음 약한 학습기가 잘 분류할 수 있게 더 큰 가중치 부여. 오류 - 데이터의 크기 커짐.
5. 세 번째 약한 학습기가 기준 3으로 + 와 - 분류하고 오류 데이터 찾음. **에이다부스트**는 이렇게 약한 학습기가 순차적으로 오류 값에 대해 가중치를 부여한 예측 결정 기준을 모두 결합해 예측 수행함.
6. 맨 아래에는 첫, 두, 세번째 약한 학습기를 모두 결합한 결과 예측. 개별 약한 학습기보다 정확도가 훨씬 높아짐.

- GBM도 에이다부스트와 유사하나, 가중치 업데이트를 **경사 하강법**을 이용하는 것이 큰 차이.
  - 오류 값 = 실제값 - 예측값
  - 실제값을 y, 피처를 x1 x2 ... xn, 이 피처에 기반한 예측 함수를 F(x)라 하면
    → 오류식: h(x) = y - F(x) ⇒ 최소화하는 방향성을 가지고 반복적으로 가중치 값 업데이트: 경사 하강법 ('반복 수행을 통해 오류를 최소화할 수 있도록 가중치의 업데이트 값을 도출하는 기법' 정도로 알아두기)
  - 분류와 회귀 모두 가능
  - `GradientBoostingClassifier`

```
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))  # GBM 정확도: 0.9376
print("GBM 수행 시간: {0:.1f} 초 ".format(time.time() - start_time))  # GBM 수행 시간: 160.2
 초
```

일반적으로 GBM이 랜덤포레스트보다는 예측 성능이 조금 뛰어난 경우가 많음. 그러나 수행 시간이 오래 걸리고, 하이퍼 파라미터 튜닝 노력도 더 필요함. `GradientBoostingClassifier` 은 약한

학습기의 순차적인 예측 오류 보정을 통해 학습을 수행하기에, 멀티 CPU 코어 시스템을 사용하더라도 병렬 처리가 지원되지 않아 시간 오래 걸림. 반면, 랜덤포레스트는 상대적으로 빠른 수행 시간 보장해 더 쉽게 예측 결과 도출 가능.

GBM 하이퍼 파라미터 및 튜닝

`loss`	- 경사 하강법에서 사용할 **비용 함수** 지정 - 특별한 이유 없으면 기본값 deviance 그대로 적용
`learning_rate`	- GBM이 학습을 진행할 때마다 적용하는 학습률 - 약한 분류기가 순차적으로 **오류 값을 보정해나가는데 적용하는 계수** - 0~1 사이의 값, 기본값 0.1 - 작은 값 → 업데이트 값 작아져 최소 오류 값을 찾아 예측 성능 높아짐 // 큰 값 → 최소 오류 찾지 못해 예측 성능 떨어짐 - 너무 작으면 수행 시간 오래 걸리고, 모든 약학습기 반복 완료돼도 최소 오류 찾지 못할 수 있음 - `learning_rate` 와 `n_estimators` 상호 보완적 사용
`n_estimators`	- 약한 학습기 개수 - 약한 학습기 순차적으로 오류 보정하므로 개수 많을수록 예측 성능 일정 수준까지는 좋아짐, but 시간 오래 걸림 - 기본값 100
`subsamples`	- 약한 학습기가 학습에 사용하는 데이터의 샘플링 비율 - 기본값 1 , 전체 학습 데이터를 기반으로 학습한다는 의미 - 과적합이 걱정되면 이 값을 1보다 작은 값으로 설정

`GridSearchCV` 하이퍼파라미터 최적화

```
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_)  # 최적 하이퍼 파라미터: {'learning_rate':
 0.05, 'n_estimators': 500}
print('최고 예측 정확도: {0:.4f}'.format(grid_cv.best_score_))  # 최고 예측 정확도: 0.9010
```

```
# GridSearchCV를 이용하여 최적으로 학습된 estimator로 predict 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))  # GBM 정확도: 0.9410
```

⇒ GBM: 과적합에도 강한 뛰어난 예측 성능 가짐, 하지만 수행 시간 오래 걸림

06. XGBoost(eXtra Gradient Boost)

XGBoost 개요

XGBoost는 트리 기반 앙상블 학습에서 가장 각광받고 있는 알고리즘 중 하나. 분류에 있어 일반적으로 다른 머신러닝보다 뛰어난 예측 성능을 나타냄. XGBoost는 GBM에 기반하지만, GBM의 단점인 느린 수행 시간 및 과적합 규제 부재 등의 문제를 해결해 각광받음. 특히 XGBoost는 병렬 CPU 환경에서 병렬 학습이 가능해 빠른 학습 가능함.

항목	
뛰어난 예측 성능	일반적으로 분류와 회귀 영역에서 뛰어난 예측 성능 발휘
GBM 대비 빠른 수행 시간	병렬 수행 및 다양한 기능으로 GBM에 비해 빠른 수행 성능 보장. (GBM보다 빠른거지 다른 머신러닝 알고리즘에 비해 빠른건 아님)
과적합 규제 (Regularization)	XGBoost 자체에 과적합 규제 기능으로 과적합에 좀 더 강한 내구성 가짐
Tree pruning(나무 가지치기)	일반적으로 GBM은 분할 시 부정 손실이 발생하면 분할을 더 이상 수행하지 않지만, 이러한 방식도 자칫 지나치게 많은 분할 발생할 수 있음. XGBoost도 max_depth 파라미터로 분할 깊이 조정하긴 하지만, tree pruning으로 더 이상 긍정 이득이 없는 분할을 가지치기 해서 분할 수를 더 줄이는 추가적인 장점 가짐.
자체 내장된 교차 검증	반복 수행 시마다 내부적으로 학습 데이터 세트와 평가 데이터 세트에 대한 교차 검증 수행해 최적화된 반복 수행 횟수 가질 수 있음. 지정된 반복 횟수가 아니라 교차 검증을 통해 평가 데이터 세트의 평가 값이 최적화 되면 반복을 중간에 멈출 수 있는 조기 중단 기능이 있음.
결손값 자체 처리	결손값 자체 처리 기능 가짐.

초기의 독자적인 XGBoost 프레임워크 기반의 XGBoost → 파이썬 래퍼 XGBoost 모듈

사이킷런과 연동되는 모듈 → 사이킷런 래퍼 XGBoost 모듈

XGBoost 설치하기

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, regularization 등과 관련 파라미터 등을 지칭

- 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 gbtree이다
- silent: 디폴트는 0이며, 출력메시지를 나타내고 싶지않을 경우 1로 설정
- nthred: CPU의 실행 스레드개 수를 조정하며, 디폴트는 CPU의 전체 스레드를 다 사용하는 것. 멀티 코어/스레드 CPU 시스템에서 전체 CPU를 사용하지 않고 일부 CPU만 사용해 ML 애플리케이션을 구동하는 경우에 변경

주요 부스터 파라미터

- eta [default=0.3, alias: learning_rate] : GBM의 학습률(learning rate)과 같은 파라미터이다. 0에서 1 사이의 값을 지정하며 부스팅 스텝을 반복적으로 수행할 때 업데이트되는 학습률 값, 파이썬 래퍼 기반의 xgboost를 이용할 경우 default는 0.3이고 사이킷런 래퍼 클래스를 이용할 경우 eta는 learning_rate 파라미터로 대체되며, default는 0.1이다. 보통은 0.01 ~ 0.2 사이의 값을 선호한다.
- num_boost_rounds : GBM의 n_estimators와 같은 파라미터이다.
- min_child_weight [default=1] : 트리에서 추가적으로 가지를 나눌지를 결정하기 위해 필요한 데이터들의 weight 총합, min_child_weight가 클수록 분할을 자제한다. 과적합을 조절하기 위해 사용된다.
- gamma [default=0, alias: min_split_loss] : 트리의 리프 노드를 추가적으로 나눌지를 결정할 최소 손실 감소 값이다. 해당 값보다 큰 손실(loss)이 감소된 경우에 리프 노드를 분리한다. 값이 클수록 과적합 감소 효과가 있다.
- max_depth [default=6] : 트리 기반 알고리즘의 max_depth와 같다. 0을 지정하면 깊이에 제한이 없다. max_depth가 높으면 특정 피쳐 조건에 특화되어 룰 조건이 만들어지므로 과적합 가능성이 높아지며 보통은 3~10 사이의 값을 적용한다.
- sub_sample [default=1] : GBM의 subsample과 동일하다. 트리가 커져서 과적합되는 것을 제어하기 위해 데이터를 샘플링하는 비율을 지정한다. sub_sample=0.5로 지정하면 전체 데이터의 절반을 트리를 생성하는 데 사용한다. 0에서 1사이의 값이 가능하나 일반적으로 0.5 ~ 1 사이의 값을 사용한다.

- colsample_bytree [default=1] : GBM의 max_feature와 유사하다. 트리 생성에 필요한 피처(column)를 임의로 샘플링하는 데 사용된다. 매우 많은 피처가 있는 경우 과적합을 조정하는 데 적용한다.
- lambda [default=1, alias: reg_lambda] : L2 Regularation 적용 값이다. 피처 개수가 많을 경우 적용을 검토하며 값이 클수록 과적합 감소 효과가 있다.
- alpha [default=0, alias: reg_alpha] : L1 Regularization 적용 값이다. 피처 개수가 많을 경우 적용을 검토하며 값이 클수록 과적합 감소 효과가 있다.
- scale_pos_weight [default=1] : 특정 값으로 치우친 비대칭한 클래스로 구성된 dataset의 균형을 유지하기 위한 파라미터이다.

학습 태스크 파라미터

- objective : 최솟값을 가져야할 손실 함수를 정의합니다. XGBoost는 많은 유형의 손실함수를 사용할 수 있습니다. 주로 사용되는 손실함수는 이진 분류인지 다중 분류인지에 따라 달라진다.
- binary:logitic : 이진 분류일 때 적용한다.
- multi:softmax : 다중 분류일 때 적용한다. 손실함수가 multi:softmax일 경우에는 label 클래스의 개수인 num_class parameter를 지정해야 한다.
- multi:softprob : multi:softmax와 유사하나 개별 label 클래스의 해당되는 예측 확률을 반환한다.
- eval_metric : 검증에 사용되는 함수를 정의한다. defalut는 회귀인 경우는 rmse, 분류일 경우에는 error이다. 다음은 eval_metric의 값 유형들이다.
    - rmse : Root Mean Square Error
    - mae : Mean Absolute Error
    - logloss : Negative log-likelihood
    - error : Binary classification error rate (0.5 threshold)
    - merror : Multiclass classification error rate
    - mloglose : Multiclass logloss
    - auc : Area under the curve

과적합 문제가 심각하다면 다음과 같이 적용할 것을 고려할 수 있다.

- eta 값을 낮춘다.(0.01 ~ 0.1) 그리고 eta 값을 낮추면 num_round(또는 n_estimator)는 반대로 높여준다.
- max_depth 값을 낮춘다.
- min_child_weight 값을 높인다.
- gamma 값을 높인다.
- 또한 subsample과 colsample_bytree를 조정하는 것도 트리가 너무 복잡하게 생성되는 것을 막아 과적합 문제에 도움이 될 수 있다.

XGBoost 자체적으로 교차 검증, 성능 평가, 피처 중요도 등의 시각화 기능 가지고 있음. 또한, GBM에서 부족한 다른 여러 가지 성능 향상 기능이 있음. 수행 속도 향상을 위한 대표적인 기능으로 조기 중단(Early Stopping) 기능이 있음. 기본 GBM의 경우, n_estimators(또는 num_boost_rounds)에 지정된 횟수만큼 반복적으로 학습 오류를 감소시켜 학습을 진행하면서 중간에 반복을 멈출 수 없고 n_estimators에 지정된 횟수를 다 완료해야 함.

XGBoost와 LightGBM 모두 조기중단 기능이 있어 n_estimators에 지정한 부스팅 반복 횟수에 도달하지 않더라도 예측 오류가 더이상 개선되지 않으면 반복을 끝까지 수행하지 않고 중지해 수행 시간을 개선할 수 있음.

예를 들어, n_estimators를 200으로 설정하고 조기 중단 파라미터 값을 50으로 설정하면, 1부터 200회까지 부스팅을 반복하다가 50회를 반복하는 동안 학습 오류가 감소하지 않으면 더 이상 부스팅을 진행하지 않고 종류함.

파이썬 래퍼 XGBoost 적용 - 위스콘신 유방암 예측

XGBoost는 병렬 처리와 조기 중단 등으로 빠른 수행시간 처리가 가능함.

위스콘신 유방암 데이터 세트는 종양의 크기, 모양 등의 다양한 속성값을 기반으로 악성 종양(malignant)인지 양성 종양(benign)인지를 분류한 데이터 세트임. 종양은 양성 종양과 악성 종양으로 구분할 수 있으며, 악성 종양이 더 위험함.
→ 종양의 다양한 피처에 따라 악성 종양인지 일반 양성종양인지를 XGBoost를 이용해 예측하겠음.

```
import xgboost as xgb
from xgboost import plot_importance  # 피처 중요도 시각화
import pandas as pd
import numpy as np
from sklearn.datasets import load_breast_cancer  # 사이킷런 내장 데이터
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)#df로 로드
cancer_df['target']= y_label
cancer_df.head(3)
```

종양의 크기와 모양 관련된 많은 속성이 숫자형 값으로 돼 있음. 타깃 레이블 값의 종류는 악성(malignant)이 0, 양성(benign)이 1로 되어있음.

```
# 레이블 값의 분포
print(dataset.target_names)  # ['malignant' 'benign']
print(cancer_df['target'].value_counts())
'''
1    357
0    212
Name: target, dtype: int64
'''
```

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

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

→ 파이썬 래퍼 XGBoost는 사이킷런과 다르게, 학습용과 테스트용 데이터 세트를 위해 별도의 객체인 DMatrix를 생성함. DMatrix는 주로 넘파이 입력 파라미터를 받아서 만들어지는 XGBoost만의 전용 데이터 세트이다. DMatrix의 주요 입력 파라미터는 data와 label이다. data는 피처 데이터 세트, label은 레이블 데이터(회귀-숫자형 종속값 데이터 세트)

파이썬 래퍼 XGBoost를 이용해 학습을 수행하기 전에 먼저 XGBoost의 하이퍼 파라미터 설정함, 주로 딕셔너리 형태로 입력.

- max_depth(트리 최대 깊이) 3
- 학습률 eta 0.1(XGBClassifier이용시 learning_rate)

- 예제 데이터가 0또는 1인 이진 분류이므로 목적합수는 이직 로지스틱(binary:logistic)
- 오류 함수의 평가 성능 지표를 logloss
- num_rounds(부스팅 반복 횟수)는 400회

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

지정된 파라미터로 모델 학습시키는데, 사이킷런과 다르게 train() 함수에 파라미터로 전달함. + 조기중단은 train()함수에 early_stopping_rounds 파라미터를 입력해 설저앟ㅁ.

- 반드시 eval_set와 eval_metirc이 함께 설정돼야 함
- 반복마다 eval_set으로 지정된 데이터 세트에서 eval_metirc의 지정된 평가 지표로 예측 오류 측정

eval 파라미터에 학습 데이터 세트와 eval 데이터 세트를 명기해주면 평가를 eval 데이터 세트에 수행하면서 조기 중단을 적용할 수 있음. 조기 중단을 수행하려면 반드시 evals 파라미터에 eval 데이터 세트를 명기해줘야 함.

학습을 수행하면, 반복 시마다 evals에 표시된 데이터 세트에 대해 평가 지표 결과가 출력됨. train()은 학습이 완료된 모델 객체 반환.

```
# train 데이터 셋은 'train' , evaluation(test) 데이터 셋은 'eval' 로 명기
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 )
```

→ train으로 학습을 수행하면서 반복시 train_error와 eval-logloss가 지속적으로 감소함.

모델 학습 완료 후 이를 이용해 테스트 데이터 세트에 예측 수행.

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

# 예측 확률이 0.5 보다 크면 1 , 그렇지 않으면 0 으로 예측값 결정하여 List 객체인 preds에 저장preds= [ 1i
```

```
f x> 0.5else 0for xin pred_probs ]
print('예측값 10개만 표시:',preds[:10])
```

```
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, recall, f1, roc_auc))
```

```
# 예측 성능 평가
get_clf_eval(y_test , preds, pred_probs)
```

```
# 시각화 기능 수행 - 피처 중요도
# f0, f1와 같이 피처 순서별로 f 뒤에 숫자 붙여 x축에 피처들 나열함
import matplotlib.pyplot as plt
%matplotlib inline

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

사이킷런 래퍼 XGBoost의 개요 및 적용

다른 Estimator와 동일하게 fit()과 predictI()만으로 학습과 예측이 가능하고, 사이킷런의 다른 유틸리티를 그대로 사용할 수 있음.

`XGBoostClassifier` `XGBoostRegressor`

- eta → `learning_rate`
- sub_sample → `subsample`

- lambda → `reg_lambda`
- alpha → `reg_alpha`
- `n_estimators`

위스콘신 대학병원의 유방암 데이터 세트 분류를 사이킷런 래퍼 XGBoost를 이용해 예측하겠음. 모델의 하이퍼 파라미터, 학습 데이터와 테스트 데이터는 이전과 동일하게 적용하겠음.

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

# 학습과 예측 수행
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]
```

```
# 모델의 예측 성능 평가 (3장 평가 p158에서 생성한 함수)
get_clf_eval(y_test , w_preds, w_pred_proba)  # w_preds = 예측 레이블
'''
오차 행렬
[[35  2]
 [ 1 76]]
정확도: 0.9737, 정밀도: 0.9744, 재현율: 0.9870,    F1: 0.9806, AUC:0.9951
'''
```

조기중단 수행 가능. 조기 중단 파라미터 fit( )에 입력하면 됨.

- early_stopping_rounds = 평가 지표가 향상될 수 있는 반복 횟수 정의
- eval_metrix = 조기 중단을 위한 평가 지표
- eval_est = 성능 평가 수행할 데이터 세트 (학습 데이터 아닌 별도의 데이터 세트) → 여기서는 테스트 데이터 세트 설정(하지만 완전히 알려지지 않은 데이터 사용해야 함)

```
from xgboost import XGBClassifier

xgb_wrapper = XGBClassifier(n_estimators=400, learning_rate=0.1, max_depth=3)
evals = [(X_test, y_test)]
xgb_wrapper.fit(X_train, y_train, early_stopping_rounds=100, eval_metric="logloss",
                eval_set=evals, verbose=True)

ws100_preds = xgb_wrapper.predict(X_test)
ws100_pred_proba = xgb_wrapper.predict_proba(X_test)[:, 1]
```

n_estimators를 400으로 설정해도 400번 반복수행하지 않고 311변 반복한 후 학습 완료함. 211번 반복 시 logloss가 0.055938이고 311번 반복시 logloss가 0.08594인데, 211번에서 311번까지 early_stopping_rounds=100으로 지정된 100번의 반복 동안 성능 평가 지수가 향상되지 않았기 때문에 더이상 반복하지 않고 멈춤.

```
# 조기 중단으로 학습된 XGBClassifier의 예측 성능
get_clf_eval(y_test , ws100_preds, ws100_pred_proba)
'''
오차 행렬
[[34  3]
 [ 1 76]]
정확도: 0.9649, 정밀도: 0.9620, 재현율: 0.9870,    F1: 0.9744, AUC:0.9954
'''
```

→ 조기 중단 적용되지 않은 결과보다 약간 저조하지만 큰 차이는 아님.

하지만 조기 중단값을 너무 급격하게 줄이면 예측 성능이 저하될 우려가 있음. early_stopping_rounds=10으로 설정하면, 성능 향상될 여지가 있음에도 불구하고 10번 반복 동안 성능 평가 지표가 향상되지 않으면 반복이 멈춰 버려 충분한 학습이 되지 않아 예측 성능이 나빠질 수 있음.

```
# early_stopping_rounds를 10으로 설정하고 재 학습.
xgb_wrapper.fit(X_train, y_train, early_stopping_rounds=10,
                eval_metric="logloss", eval_set=evals,verbose=True)

ws10_preds = xgb_wrapper.predict(X_test)
ws10_pred_proba = xgb_wrapper.predict_proba(X_test)[:, 1]
get_clf_eval(y_test , ws10_preds, ws10_pred_proba)
'''
오차 행렬
[[34  3]
 [ 2 75]]
정확도: 0.9561, 정밀도: 0.9615, 재현율: 0.9740,    F1: 0.9677, AUC:0.9947
'''
```

```
# 피처의 중요도 시각화 모듈.
from xgboost import plot_importance
import matplotlib.pyplot as plt
%matplotlib inline

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

```
# 사이킷런 래퍼 클래스를 입력해도 무방.
plot_importance(xgb_wrapper, ax=ax)
```

07. LightGBM

XGBoost 여전히 학습 시간 오래 걸림, GridSearchCV로 하이퍼 파라미터 튜닝할 때 특히 오래 걸림.

LightGBM의 장점은 학습 시간이 훨씬 적고, 메모리 사용량이 상대적으로 적다는 것이다. 또한, XGBoost와 성능 차이가 크게 없고, 기능상의 다양성도 많음. 하지만, 적은 데이터 세트(일반적으로 10,000건 이하의 데이터)에 적용할 경우 과적합 발생하기 쉬움.

- 일반 GBM 계열의 트리 분할 방법과 다르게, 리프 중심 트리 분할(Leaf Wise) 방식 사용함
    - 균형 트리 분할(Level Wise): 기존 대부분의 트리 알고리즘, 트리의 깊이를 효과적으로 줄이기 위해 사용 → 최대한 균형 잡힌 트리 유지하면서 분할 → 트리의 깊이 최소화
    ⇒ 오버피팅에 강함, but 균형 잡기 위한 시간 소요
    - 리프 중심 트리 분할(Leaf Wise): 트리의 균형 맞추지 않고, 최대 손실 값을 가지는 리프 노드를 지속적으로 분할하면서 트리의 깊이가 깊어지고 비대칭적 규칙 트리 생성 → 예측 오류 손실 최소화

LightGBM의 XGBoost 대비 장점

- 더 빠른 학습과 예측 수행 시간
- 더 적은 메모리 사용량
- 카테고리형 피처의 자동 변환과 최적 분할(원-핫 인코딩 등을 사용하지 않고도 카테고리형 피처를 최적으로 변환하고 이에 따른 노드 분할 수행)

+) 둘다 대용량 데이터에 대한 뛰어난 예측 성능 및 병렬 컴퓨팅 기능, GPU 지원 추가

LightGBM 설치

LightGBM 하이퍼 파라미터

LightGBM의 XGBoost 파라미터 많은 부분 유사. 하지만, 주의할 점은 LightGBM은 XGBoost와 다르게 리프 노드가 계속 분할되면서 트리가 깊어지므로 이러한 트리 특성에 맞는 하이퍼 파라미터 설정이 필요함(e.g. max_depth를 매우 크게 가짐)

주요 파라미터

표 안에 넣은건 사이킷런 랩퍼 LightGBM 파라미터

- 로 시작하는건 파이썬 랩퍼 파라미터
- boosting [default=gbdt] : 부스팅 트리를 생성하는 알고리즘 (gbdt:일반적인 그래디언트 부스팅, rf:랜덤포레스트)

`n_estimators`	- 반복 수행하기 위한 트리의 개수 지정 - 크게 지정할수록 예측 성능이 높아질 수 있지만 과적합 문제 가능성 또한 높아짐
`learning_rate`	- 0~1 사이의 값을 지정해 부스팅 스텝을 반복적으로 수행할 때 업데이트 되는 학습률 값 -일반적으로 n_estimators를 크게하고 learning_rate을 작게 해서 예측 성능 향상 가능 - 과적합 이슈 동반, 학습 시간 길어질 수 있음
`max_depth`	- 0보다 작은 값 지정하면 깊이에 제한 없음 - LightGBM은 Leaf Wise 기반이므로 깊이가 상대적으로 더 깊음
`min_child_samples`	- 최종 결정 클래스인 리프 노드가 되기 위해서 최소한으로 필요한 레코드 수 - 과적합 제어 파라미터
`num_leaves`	- 하나의 트리가 가질 수 있는 최대 리프 개수
`subsample`	- 트리가 커져서 과적합되는 것을 제어하기 위해 데이터를 샘플링하는 비율 지정
`colsample_bytree`	- 개별 트리를 학습할 때마다 무작위로 선택하는 피처의 비율 - 과적합 막기 이ㅜ해 사용함
`reg_lambda`	- L2 regulation 제어를 위한 값 - 피처 개수가 많을 경우 적용 검토, 값이 클수록 과적합 감소 효과
`reg_alpha`	- L1 regulation 제어를 위한 값 - 과적합 제어를 위한 것

Learning Task 파라미터

- objective: 최솟값을 가져야 할 손실함수 정의, 애플리케이션 유형, 즉 회귀, 분류(다중, 이진) 문제인지에 따라 손실함수 지정

하이퍼 파라미터 튜닝 방안

`num_leaves` 개수 중심으로 `min_child_samples`, `max_depth` 를 함께 조정하면서 모델의 복잡도를 줄이는 것이 기본 튜닝 방안.

- **`num_leaves`** 개별 트리가 가질 수 있는 최대 리프의 개수를 지정. LightGBM 복잡도를 제어하는 주 파라미터, num_leaves가 클수록 모델의 깊이가 깊어지므로 모델 정확도가 높아질 수 있지만 과적합 문제가 발생할 확률 또한 높아짐

- `min_child_samples` 과적합 개선을 위해 사용되는 주 파라미터, 큰 값을 설정하는 경우 트리가 깊어지는 것을 방지함
- `max_depth` 명시적으로 트리의 깊이를 제어하는 파라미터

learning_rate를 작게하면서 n_estimators를 크게 하는 것은 부스팅 계열 튜닝에서 가장 기본적인 튜닝 방안이므로 이를 적용하는 것도 좋음. 하지만 n_estimators를 너무 크게 하는 것은 과적합으로 인한 성능 저하될 수 있음.

파이썬 래퍼 LightGBM과 사이컷런 래퍼 XGBoost, LightGBM 하이퍼 파라미터 비교

사이킷런 래퍼 LightGBM	사이킷런 래퍼 XGBoost
`n_estimators`	`n_estimators`
`learning_rate`	`learning_rate`
`max_depth`	`max_depth`
`min_child_samples`	N/A
`subsample`	`subsample`
`colsample_bytree`	`colsample_bytree`
`reg_lambda`	`reg_lambda`
`reg_alpha`	`reg_alpha`
`early_stopping_rounds`	`early_stopping_rounds`
`num_leaves`	N/A
`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()
ftr = dataset.data
target = dataset.target

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

```
# 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]
```

→ 조기 중단으로 147번 반복까지만 수행하고 학습 종료함. 이제 학습된 LightGBM으로 예측 성능 평가.

```
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, recall, f1, roc_auc))
```

```
get_clf_eval(y_test, preds, pred_proba)
'''
오차 행렬
[[33  4]
 [ 2 75]]
정확도: 0.9474, 정밀도: 0.9494, 재현율: 0.9740,    F1: 0.9615, AUC:0.9926
'''
```

```
# 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)
```

08. 분류 실습 - 캐글 산탄데르 고객 만족 예측

데이터 전처리

XGBoost 모델 학습과 하이퍼 파라미터 튜닝

LightGBM 모델 학습과 하이퍼 파라미터 튜닝

09. 분류 실습 - 캐글 신용카드 사기 검출

언더 샘플링과 오버 샘플링의 이해

데이터 일차 가공 및 모델 학습/예측/평가

데이터 분포도 변환 후 모델 학습/예측/평가

이상치 데이터 제거 후 모델 학습/예측/평가

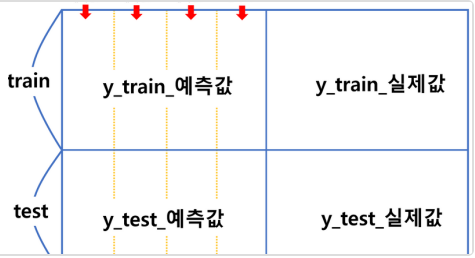
SMOTE 오버 샘플링 적용 후 모델 학습/예측/평가

10. 스태킹 앙상블

[ML] 스태킹(Stacking) 완벽 정리

이 포스팅만 읽으면 스태킹을 쉽게 이해할 수 있도록 정리해봤습니다 :) 천천히 읽어볼까요? 아직 잘 와닿지가 않나요? 간단한 예시를 아래 그림과 함께 들어보겠습니다. 저는 knn, logistic regression, randomforest,

https://hwi-doc.tistory.com/entry/%EC%8A%A4%ED%83%9C%ED%82%B9Stacking-%EC%99%84%EB%B2%BD-%EC%A0%95%EB%A6%AC



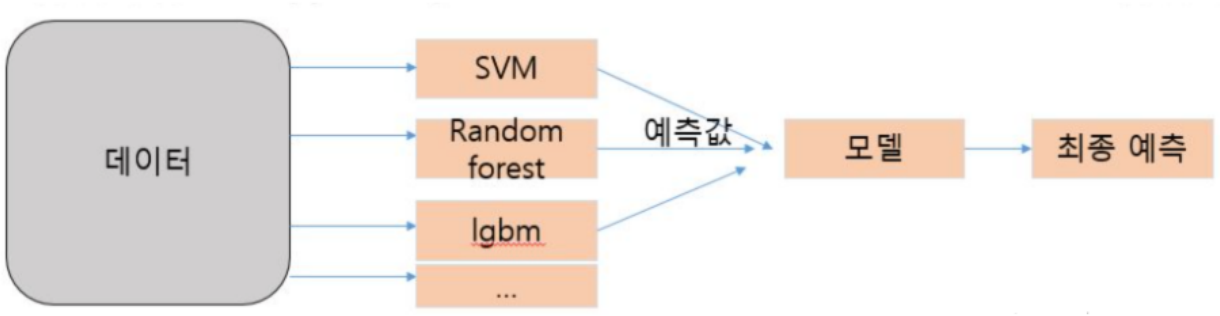
⇒여러 가지 모델들의 예측값을 (하나의 데이터 프레임으로 만들어) 최종 모델의 학습 데이터로 사용하는 예측 방식.

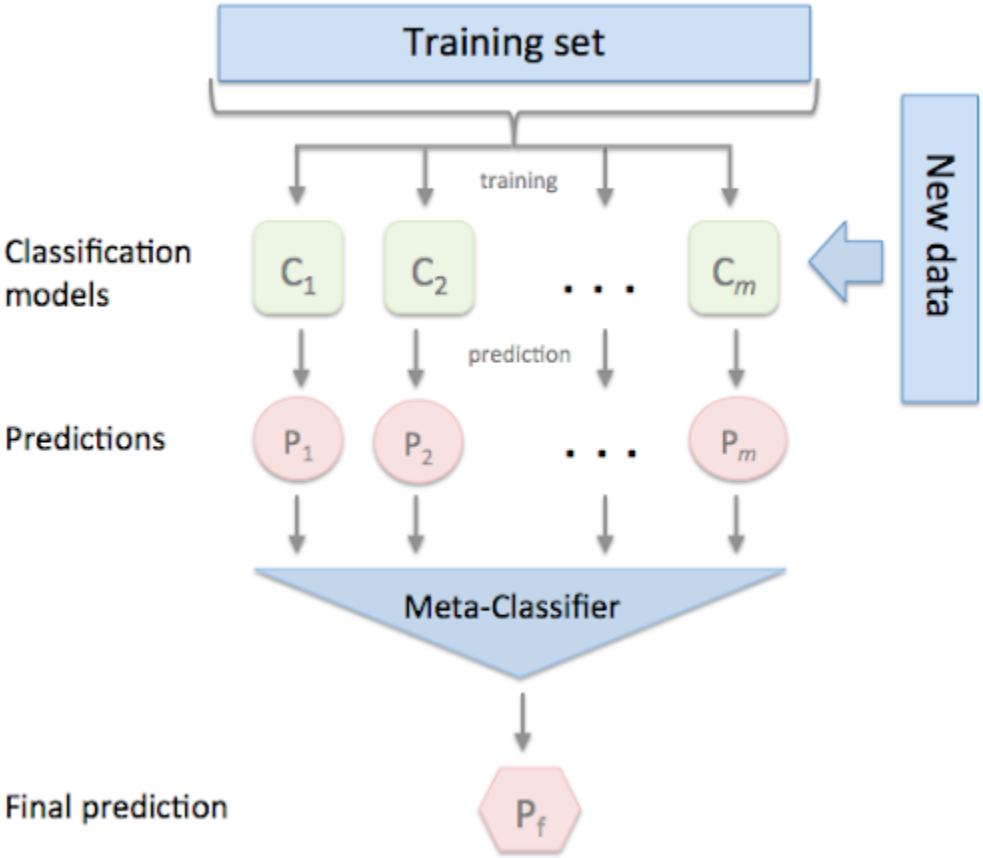
스태킹: 개별적인 여러 알고리즘을 서로 결합해 예측 결과를 도출함(배깅, 부스팅과 공통) 차이는, 개별 알고리즘으로 예측한 데이터를 기반으로 다시 예측을 수행
⇒ 즉, 개별 알고리즘의 예측 결과 데이터 세트를 최종적인 메타 데이터 세트로 만들어 별도의 ML 알고리즘으로 최종 학습을 수행하고 테스트 데이터를 기반으로 다시 최종 예측을 수행하는 방식.
(이렇게 개별 모델의 예측된 데이터 세트를 다시 기반으로 하여 학습하고 예측하는 방식을 메타 모델이라고 함)

1) 개별적인 기반 모델

2) (개별 기반 모델의 예측 데이터를 // 학습 데이터로 만들어서 학습하는) 최종 메타 모델

⇒ 여러 개별 모델의 예측 데이터를, 각각 스태킹 형태로 결합해, 최종 메타 모델의 학습 피처 데이터 세트와 테스트용 피처 데이터 세트를 만드는 것.
(일반적으로 많은 개별 모델이 필요하고, 스태킹 반드시 성능 향상 보장x, 일반적으로 성능 비슷한 모델을 결합해 좀 더 나은 성능 향상 도출 위해 적용됨)





https://kimdingko-world.tistory.com/186

⇒ 그림 설명: 여러 개의 모델에 대한 예측값을 합한 후, 즉 스태킹 형태로 쌓은 뒤, 이에 대한 예측을 다시 수행

1. M개의 로우, N개의 피처(칼럼)을 가진 데이터 세트에 스태킹 앙상블을 적용한다고 가정. 학습에 사용할 ML 알고리즘은 모두 3개.
2. 먼저 모델별로 각각 학습을 시킨 뒤 예측을 수행하면서 각각 M개의 로우를 가진 1개의 레이블 값을 도출할 것.
3. 모델별로 도출된 예측 레이블 값을 다시 합해서(스태킹) 새로운 데이터 세트를 만들고 이렇게 스태킹된 데이터 세트에 대해 최종 모델을 적용해 최종 예측을 하는 것이 스태킹 앙상블 모델.

기본 스태킹 모델

```
# 위스콘신 암 데이터 세트
import numpy as np
```

```
import pandas as pd

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)
```

```
# 개별 ML 모델을 위한 Classifier 생성.
knn_clf  = KNeighborsClassifier(n_neighbors=4)
rf_clf = RandomForestClassifier(n_estimators=100, random_state=0)
dt_clf = DecisionTreeClassifier()
ada_clf = AdaBoostClassifier(n_estimators=100)

# 최종 Stacking 모델을 위한 Classifier생성. -> 로지스틱 회귀 (예측 결과를 합한 데이터 세트로 학습/예측하는
최종 모델)
lr_final = LogisticRegression(C=10)
```

```
# 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))) # KNN 정확도: 0.9211
print('랜덤 포레스트 정확도: {0:.4f}'.format(accuracy_score(y_test, rf_pred)))  # 랜덤 포레스트
 정확도: 0.9649
print('결정 트리 정확도: {0:.4f}'.format(accuracy_score(y_test, dt_pred))) # 결정 트리 정확도:
 0.9123
print('에이다부스트 정확도: {0:.4f}'.format(accuracy_score(y_test, ada_pred)))  # 에이다# 부스트
 정확도: 0.9561
```

```
# 개별 알고리즘으로부터 예측된 예측값을 칼럼 레벨 옆으로 붙여 피처 값으로 만듦 -> 최종 메타 모델인 로지스틱 회귀
에서의 학습데이터로 다시 사용
```

```
# 반환된 예측 데이터 셋은 1차원 형태의 ndarray이르모 먼저 반환된 예측 결과를 행 형태로 붙임
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)
```

```
# 예측 데이터로 생성된 데이터 셋을 기반으로 최종 메타 모델인 로지스틱 회귀 학습, 예측 정확도 구함
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)))  #0.9737
```

CV 세트 기반의 스태킹

과적합 개선을 위해 최종 메타 모델을 위한 데이터 세트를 만들 때, **교차 검증 기반으로 예측된 결과 데이터 세트**를 이용함.

앞 예제에서 마지막에 메타 모델인 로지스틱 회귀 모델 기반에서 최종 학습할 때 레이블 데이터 세트로 학습 데이터가 아닌 테스트용 레이블 데이터 세트를 기반으로 학습해서 과적합 문제가 발생할 수 있음.

CV 세트 기반의 스태킹은 이에 대한 개선을 위해 개별 모델들이 각각 교차 검증으로 메타 모델을 위한 학습용 스태킹 데이터 생성과 예측을 위한 테스트용 스태킹 데이터를 생성한 뒤 이를 기반으로 메타 모델이 학습과 예측을 수행함.

1. 각 모델별로 원본 학습/테스트 데이터를 예측한 결과 값을 기반으로 메타 모델을 위한 학습용/테스트용 데이터를 생성함.(개별 모델에서 메타 모델에서 사용될 학습용 데이터와 테스트용 데이터를 교차 검증을 통해 생성하는 것!)
2. 1단계에서 개별 모델들이 생성한 학습용 데이터를 모두 스태킹 형태로 합쳐서 메타 모델이 학습할 최종 학습용 데이터 세트를 생성함. 마찬가지로 각 모델들이 생성한 테스트용 데이터를 모두 스태킹 형태로 합쳐서 메타 모델이 예측할 최종 테스트 데이터 세트를 생성함. 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 평가함.

11. 정리

앙상블 기법 - 결정 트리 기반의 다수의 약한 학습기를 결합해 변동성을 줄여 예측 오류를 줄이고 성능 개선함

- 결정 트리: 정보의 균일도에 기반한 규칙 트리 만들어 예측 수행
- 배깅: 학습 데이터의 중복을 허용하면서 다수의 세트로 샘플링하여 이를 다수의 약한 학습기가 학습한 뒤 최종 결과 결합해 예측 - 랜덤 포레스트
- 부스팅: 학습기들이 순차적으로 학습 진행하면서 예측이 틀린 데이터에 대해 가중치 부여해 다음번 학습기가 학습할 때에는 이전에 예측이 틀린 데이터에 대해서는 보다 높은 정확도로 예측할 수 있도록 해줌 - GBM XGBoost, LightGBM
- 스태킹: 여러 개별 모델들이 생성한 예측 데이터를 기반으로 최종 메타 모델이 학습할 별도의 학습 데이터 세트와 예측할 테스트 데이터 세트를 재생성하는 기법