4. 분류 (2)

goblurry · 방금 전

데이터분석 머신러닝 파완머

캐글의 산탄데르 고객 만족 데이터 세트에 대해, XGBoost와 LightGBM을 활용해 고객 만족 여부 예측
(https://www.kaggle.com/competitions/santander-customer-satisfaction/rules)
전체 코드:

데이터 전처리

클래스 레이블 Target이 1이면 불만, 0이면 만족 고객이다. 모델의 성능 평가는 ROC-AUC로 이루어진다.
(대부분 만족일 것이기 때문에 정확도 수치보다 적합하다. '진짜 양성 비율(TPR)에 대한 거짓 양성 비율(FPR)')

먼저 cust_df 변수에 데이터셋을 로딩한다.

```
# 필요한 모듈 로딩 + 학습 데이터 DF로 로딩
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='latin-1')
print('dataset shape:', cust_df.shape)
cust_df.head(3)
```

```
dataset shape: (76020, 371)
```

	ID	var3	var15	imp_ent_var16_ult1	imp_op_var39_comer_ult1	imp_op_var39_comer_ult3	imp_op
0	1	2	23	0.0	0.0	0.0	
1	3	2	34	0.0	0.0	0.0	
2	4	2	23	0.0	0.0	0.0	

3 rows × 371 columns

```
print(cust_df['TARGET'].value_counts())
unsatisfied_cnt = cust_df[cust_df['TARGET'] == 1].TARGET.count() # target 값이 1일 때 불만족
total_cnt = cust_df.TARGET.count()
print('unsatisfied 비율은 {0:.2f}'.format((unsatisfied_cnt / total_cnt)))
```

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

Target 값이 0인 게 73012개, 1인 게 3008개이다. 예상대로 대부분이 만족(0)이며 불만족 응답을 한 고객은 4%에 불과하다.

먼저 사이킷런의 train_test_split을 활용해 학습 데이터와 테스트 데이터 세트를 분리한 뒤, Target 값(=클래스) 분포도가 비슷하게 추출되었는지 확인해야 한다. 비대칭한 데이터 세트이기 때문이다.

```
# 원본 데이터셋에서 학습/테스트 데이터셋 분리
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X_features, y_labels, test_size=0.2, r
train_cnt = y_train.count()
test_cnt = y_test.count()
print('학습 세트 Shape: {0}, 테스트 세트 Shape: {1}'.format(X_train.shape, X_test.shape))

print('\n학습 세트 레이블 값 분포 비율')
print(y_train.value_counts()/train_cnt)
print('\n테스트 세트 레이블 값 분포 비율')
print(y_test.value_counts()/test_cnt)

# 결과 ==================================================
학습 세트 Shape: (60816, 369), 테스트 세트 Shape: (15204, 369)

학습 세트 레이블 값 분포 비율
0    0.960964
1    0.039036
Name: TARGET, dtype: float64

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

학습 데이터셋과 테스트 데이터셋 모두 원본 데이터와 유사하게 전체 데이터의 약 4%가 불만족 값(Target=1)로 추출된 것을 확인할 수 있다. XGBoost의 **조기 중단** 검증 데이터셋으로 활용하기 위해 X_train과 y_train을 다시 쪼개 학습/검증 데이터셋으로 분리한다. (test_size=0.3)

XGBoost의 모델 학습과 하이퍼 파라미터 튜닝

XGBoost 모델을 생성하고 예측 결과를 ROC AUC로 평가한다. 학습 수행은 사이킷런의 XGBClassifier를 기반으로 한다.

```
# XGBoost의 학습 모델 생성 + 예측 결과를 roc auc로 평가

from xgboost import XGBClassifier # sckitlearn wrapper
from sklearn.metrics import roc_auc_score

# n_estimator=500. random states는 예제 수행 시마다 동일 예측 결과 위해 설정
xgb_clf = XGBClassifier(n_estimators=500, learning_rate=0.05, random_state=156)

# 성능 평가 지표를 auc로, 조기 중단 파라미터를 100으로 설정하고 학습 수행
xgb_clf.fit(X_tr, y_tr, early_stopping_rounds=100, eval_metric="auc", eval_set=[(X_tr, y_t
```

```
xgb_roc_score = roc_auc_score(y_test, xgb_clf.predict_proba(X_test)[:, 1])
print('ROC AUC: {0:.4f}'.format(xgb_roc_score))
```

앞서 분리한 학습 / 검증 데이터셋을 이용해 eval_set에 넣고, 조기 중단은 100회로 설정한 뒤 학습을 진행했다. 이때 예측 시 ROC AUC는 0.8429가 출력된다.

HyperOpt의 베이지안 최적화 기법을 활용한 XGBoost의 하이퍼 파라미터 튜닝

> HyperOpt: ML 모델의 하이퍼 파라미터 튜닝에 베이지안 최적화를 적용할 수 있게 제공되는 대표적인 파이썬 패키지
>
> **주요 로직**
> 입력 변수명, 입력값의 검색 공간 설정
> 목적 함수 설정
> 목적 함수의 최솟값을 가지는 최적 입력값 유추

1. 검색 공간 설정

```
# 하이퍼 파라미터 검색 공간 설정
# max_depth: 5~15 1간격, min_child_weight: 1~6 1간격
# colsample_bytree: 0.5~0.95, learning_rate: 0.01~0.2 사이 정규분포된 값으로 검색

from hyperopt import hp
xgb_search_space = {'max_depth': hp.quniform('max_depth', 5, 15, 1),
                    'min_child_weight': hp.quniform('min_child_weight', 1, 6, 1),
                    'colsample_bytree': hp.uniform('colsample_bytree', 0.5, 0.95),
                    'learning_rate': hp.uniform('learning_rate', 0.01, 0.2)}
```

2. 목적 함수 생성

```
# 목적 함수 생성 : 3폴드 교차 검증 이용해 roc-auc 평균값을 반환하되, -1 곱하기
from sklearn.model_selection import KFold
from sklearn.metrics import roc_auc_score

# fmin()에서 호출 시 search_space 값으로 XGBClassifier 교차 검증 학습 후 -1*roc_auc 평균 값 반환
def objective_func(search_space):
    xgb_clf = XGBClassifier(n_estimators=100, max_depth=int(search_space['max_depth']), #
                            min_child_weight = int(search_space['min_child_weight']),
                            colsample_bytree = search_space['colsample_bytree'],
                            learning_rate = search_space['learning_rate'])

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

    # 3개 k-fold 방식 적용
    kf = KFold(n_splits=3)
    # X_train 다시 학습/검증용 데이터로 분리
    for tr_index, val_index in kf.split(X_train):
        # kf.split(X_train)으로 추출된 학습/검증 인덱스 값으로 데이터셋 분리
        X_tr, y_tr = X_train.iloc[tr_index], y_train.iloc[tr_index]
        X_val, y_val = X_train.iloc[val_index], y_train.iloc[val_index]

        # 조기중단 30회 설정, 추출된 학습/검증 데이터로 XGBClassifier 학습 수행
        xgb_clf.fit(X_tr, y_tr, early_stopping_rounds=30, eval_metric="auc",
                eval_set=[(X_tr, y_tr), (X_val, y_val)])
```

```
        # 1(불만족)로 예측한 확률값 추출 후 roc auc 계산하고 평균 roc auc 계산을 위해 list에 결괏값 공유
        score = roc_auc_score(y_val, xgb_clf.predict_proba(X_val)[:,1])
        roc_auc_list.append(score)

    # 3개 kfold로 계산된 roc-auc 값의 평균값 반환하되, -1 곱하기
    return -1*np.mean(roc_auc_list)
```

최소 반환값을 유추하는 HyperOpt의 특성을 고려하여 -1을 곱해 준다.

3. fmin() 함수 호출해 최적 입력값 찾기

```
from hyperopt import fmin, tpe, Trials # fmin 함수 호출
trials = Trials()

# fmin 함수 호출해 max_evals 지정된 횟수(50)만큼 반복 후 목적함수의 최솟값 가지는 최적의 입력값 추출하기
best = fmin(fn=objective_func,
            space=xgb_search_space,
            algo=tpe.suggest,
            max_evals=50,
            trials=trials, rstate=np.random.default_rng(seed=30))
print('best:', best)
```

50회만큼 교차 검증을 반복해 학습과 평가를 수행한다. 교재에는 약 30분 소요된다고 되어 있으나 훨씬 더 오래 소요되었다... (약 2시간?) 그 결과 도출된 하이퍼 파라미터 값들은 다음과 같다.

```
100%|████████| 50/50 [23:49<00:00, 28.59s/trial, best loss: -0.8375277185394956]
best: {'colsample_bytree': 0.6511149462012105, 'learning_rate': 0.16991129737205532, 'max_d
epth': 5.0, 'min_child_weight': 4.0}
```

이제 이 최적 하이퍼 파라미터를 기반으로 분류 모델을 재학습시키고 테스트 데이터셋에서 ROC AUC를 측정한다. (n_estimators=500으로 증가)

```
# 최적 파라미터 기반으로 학습과 예측 수행. n_estimators=500으로 증가

xgb_clf = XGBClassifier(n_estimators=500, learning_rate=round(best['learning_rate'], 5),
                        max_depth=int(best['max_depth']),
                        min_child_weight=int(best['min_child_weight']),
                        colsample_bytree=round(best['colsample_bytree'], 5)
                       )

# 평가 지표 auc, 조기 중단 100 설정
xgb_clf.fit(X_tr, y_tr, early_stopping_rounds=100,
            eval_metric="auc", eval_set=[(X_tr, y_tr), (X_val, y_val)])

xgb_roc_score = roc_auc_score(y_test, xgb_clf.predict_proba(X_test)[:, 1])
print('ROC AUC: {0:.4f}'.format(xgb_roc_score))
```

결과로 도출되는 ROC AUC: 0.8443
파라미터 설정 전 값: 0.8429

크게 개선되지는 않았으나 치열한 순위 경쟁이 필요한 경우 이 정도 수치 개선은 도움이 된다고 한다...

XGBoost가 GBM보다는 빠르지만 수행 시간이 상당히 오래 소요된다. 이는 GBM을 기반으로 하고 있기 때문이다. **앙상블** 기반 알고리즘(배깅, 부스팅 등)에서 하이퍼 파라미터 튜닝으로 성능 수치가 급격히 개선되는 경우는 많지 않은데, 이는 앙상블 계열 알고리즘이 과적합이나 잡음에 기본적으로 우수한 알고리즘이라서

그렇다.

LightGBM을 이용한 모델 학습과 하이퍼 파라미터 튜닝을 통한 예측 성능 평가

모델만 LightGBM으로 바꾸어 동일 작업을 수행한다. 검색 공간 설정, 목적 함수 생성, fmin 호출, 성능 평가 부분 모두 핵심 로직은 전부 동일하다. 모델 객체와 하이퍼 파라미터명에 유의할 것.

```
from lightgbm import LGBMClassifier

lgbm_clf = LGBMClassifier(n_estimators=500)
eval_set = [(X_tr, y_tr), (X_val, y_val)]
lgbm_clf.fit(X_tr, y_tr, early_stopping_rounds=100, eval_metric="auc", eval_set=eval_set)

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

출력 결과: ROC AUC 0.8384

1. 검색 공간 설정

```
# HyperOpt 사용한 하이퍼 파라미터 튜닝
# 하이퍼 파라미터 검색 공간 설정

lgbm_search_space = {'num_leaves': hp.quniform('num_leaves', 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)}
```

미묘하게 XGBoost의 하이퍼 파라미터명과 차이가 있다. (min_child_weight > samples.. 처럼)

2. 목적 함수 생성

3. fmin() 호출해 최적 하이퍼 파라미터 도출

이 부분 역시 동일하다. max_evals로 정의된 최대 반복 횟수만큼 fmin 함수를 반복하며 최적의 하이퍼 파라미터 값을 추출한다. XGBoost 모델 학습시킬 때보다 확연히 빠른 속도가 체감된다...

그 결과는 다음과 같다.

```
100%|████████| 50/50 [02:05<00:00,  2.52s/trial, best loss: -0.8357657786434084]
best: {'learning_rate': 0.08592271133758617, 'max_depth': 121.0, 'min_child_samples': 69.0,
'num_leaves': 41.0, 'subsample': 0.9148958093027029}
```

4. 하이퍼 파라미터를 이용해 LightGBM 학습 후 ROC-AUC 평가

```
lgbm_clf = LGBMClassifier(n_estimators=100, num_leaves=int(best['num_leaves']),
                              max_depth=int(best['max_depth']),
                            min_child_samples = int(best['min_child_samples']),
                            subsample = round(best['subsample'],5),
                            learning_rate = round(best['learning_rate'],5)
                          )

# evaluation metric = auc, 조기중단 100
lgbm_clf.fit(X_tr, y_tr, early_stopping_rounds=100,
             eval_metric="auc", eval_set=[(X_tr, y_tr), (X_val, y_val)])
```

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

ROC AUC: 0.8384
LightGBM은 상대적으로 학습 시간이 빠르기 때문에 추가적인 하이퍼 파라미터를 적용해 튜닝을 수행해 볼 수도 있다.

goblurry

이전 포스트
4. 분류 (1)