4조 파이썬 머신러닝 완벽 가이드 발표

최유미, 엄유진, 서지민

EWHA EURON

목차

EWHA EURON

4.9 캐글 산탄데르 고객 만족 예측

EWHA EURON

#4.9 캐글 산탄데르 고객 만족 예측

#01 데이터 전처리

#02 XGBoost 모델 학습과 하이퍼 파라미터 튜닝

#03 LightGBM 모델 학습과 하이퍼 파라미터 튜닝

#01 데이터 전처리

- 클래스 레이블 명은 TARGET
- TARGET 값이 1이면 불만족, 0이면 만족한 고객

Dataset Description

You are provided with an anonymized dataset containing a large number of numeric variables. The "TARGET" column is the variable to predict. It equals one for unsatisfied customers and 0 for satisfied customers.

The task is to predict the probability that each customer in the test set is an unsatisfied customer.

- info()로 데이터 분석
- 76020개의 행, 370개의 열
- float형 111개, int형 259개의 피처 존재

```
cust_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 76020 entries, 0 to 76019
Columns: 370 entries, var3 to TARGET
dtypes: float64(111), int64(259)
memory usage: 214.6 MB
```

#01 데이터 전처리

- 레이블인 TARGET 속성의 값의 분포 확인
- 불만족인 고객이 전체의 4%이다. -> 정확도보다는 ROC-AUC가 성능 평가에 적합

```
print(cust_df['TARGET'].value_counts())
unsatisfied_cnt = cust_df[cust_df['TARGET'] == 1].TARGET.count()
total_cnt = cust_df.TARGET.count()
print('unsatisfied 비율은 {0:.2f}'.format((unsatisfied_cnt / total_cnt)))
# 불만족인 고객 4% -> ROC-AUC로 성능 평가
```

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

#01 데이터 전처리

- describe()로 각 피처의 값 분포 확인
- var3의 min 값이 -999999 -> NaN이나 특정 예외 값을 변환한 값이므로 평균값으로 변환
- 불필요한 ID 피처 드롭

```
# 각 피처의 값 분포
cust_df.describe( )
```

	ID	var3	var15
count	76020.000000	76020.000000	76020.000000
mean	75964.050723	-1523.199277	33.212865
std	43781.947379	39033.462364	12.956486
min	1.000000	-999999.000000	5.000000
25%	38104.750000	2.000000	23.000000
50%	76043.000000	2.000000	28.000000
75%	113748.750000	2.000000	40.000000
max	151838.000000	238.000000	105.000000

8 rows × 371 columns

```
# 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))
```

피처 데이터 shape:(76020, 369)

* ID와 레이블 피처를 제외하여 피처 데이터 세트의 열이 369개로 줄었다.

#01 데이터 전처리

- 80%가 학습용 데이터 세트가 되도록 데이터 분리
- 비대칭한 데이터 세트이므로 클래스인 TARGET의 분포도가 비슷한지 확인

```
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, random_state=0)
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(' 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)
# Target 값의 분포도가 학습 및 테스트 데이터 세트에 비슷하게 추출됐는지 확인
```

```
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/테스트 데이터 세트 모두 약 4%의 레이블 값 분포를 가진다.

#01 데이터 전처리

- XGBoost의 조기 중단을 위한 검증 데이터 세트 분리
- 학습용 데이터 세트를 다시 학습, 검증 데이터 세트로 분리한다.
- 학습용 데이터 세트르 70%로 분리

```
# X_train, y_train을 다시 학습과 검증 데이터 세트로 분리, (조기중단)
X_tr, X_val, y_tr, y_val = train_test_split(X_train, y_train,
                                            test_size=0.3, random_state=0)
```

#02 XGBoost 모델 학습과 하이퍼 파라미터 튜닝

- 사이킷런 래퍼 XGBClassifier를 기반으로 학습을 수행한다.
- n_estimators = 500 (반복횟수)
- early_stopping_rounds = 100 (조기 중단 조건)
- eval_metric = ‘auc’ (성능 평가 기준 ROC-AUC)
- eval_set = [(X_tr, y_tr), (X_val, y_val)] (학습/검증 데이터 세트)

- 성능 평가는 sklearn에 내장된 roc_auc_score() 메서드 이용

```
from xgboost import XGBClassifier
from sklearn.metrics import roc_auc_score

# n_estimators는 500으로, learning_rate 0.05, random state는 예제 수행 시마다 동일 예측 결과를 위해 설정.
xgb_clf = XGBClassifier(n_estimators=500, learning_rate=0.05, random_state=156)

# 성능 평가 지표를 auc로, 조기 중단 파라미터는 100으로 설정하고 학습 수행.
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))
```

```
[234]   validation_0-auc:0.91414        validation_1-auc:0.83307
[235]   validation_0-auc:0.91415        validation_1-auc:0.83305
ROC AUC: 0.8429
```

EWHA EURON

#02 XGBoost 모델 학습과 하이퍼 파라미터 튜닝

- 하이퍼 파라미터 검색 공간 설정하기
- max_depth와 min_child_weight은 간격 검색 공간을 위해 quniform() 사용
- colsample_bytree와 learning_rate는 정규 분포 값을 위해 uniform() 사용

```
from hyperopt import hp

# max_depth는 5에서 15까지 1간격으로, min_child_weight는 1에서 6까지 1간격으로
# colsample_bytree는 0.5에서 0.95사이, learning_rate는 0.01에서 0.2사이 정규 분포된 값으로 검색.

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)
}
```

- max_depth : 트리의 최대 높이(깊이)
- min_child_weight : 리프 노드에 포함되는 최소 관측지 수. 과적합 방지
- colsample_bytree : 트리마다 사용할 칼럼(피처)의 비율
- learning_rate : 학습률. 낮을수록 견고해짐

#02 XGBoost 모델 학습과 하이퍼 파라미터 튜닝

- 목적 함수 만들기
- XGBoost와 LightGBM에서는 교차 검증을 하면 조기 중단 지원이 안된다 -> KFold 방식으로 직접 구현
- 3 Fold 교차 검증을 이용해 평균 ROC-AUC 값을 반환하며, -1을 곱해 최대 ROC-AUC 값이 최소 반환값이 되게 한다.

```
# 목적 함수 설정.
# 추후 fmin()에서 입력된 search_space값으로 XGBClassifier 교차 검증 학습 후 -1* roc_auc 평균 값을 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 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 학습과 검증 데이터 세트 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]
        # early stopping은 30회로 설정하고 추출된 학습과 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)])

        # 1로 예측한 확률값 추출후 roc auc 계산하고 평균 roc auc 계산을 위해 list에 결과값 담음.
        score = roc_auc_score(y_val, xgb_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)
```

* int 형으로 형변환 주의

* 목적 함수의 반환 값이 **최소**가 될 수 있는 최적의 값을 찾으므로 -1 곱하기!

#02 XGBoost 모델 학습과 하이퍼 파라미터 튜닝

- HyperOpt에서 제공하는 fmin() 함수를 이용해 최적의 하이퍼 파라미터 도출

```
from hyperopt import fmin, tpe, Trials

trials = Trials()

# fmin()함수를 호출. max_evals지정된 횟수만큼 반복 후 목적함수의 최소값을 가지는 최적 입력값 추출.
best = fmin(fn=objective_func,
            space=xgb_search_space,
            algo=tpe.suggest,
            max_evals=50, # 최대 반복 횟수를 지정합니다.
            trials=trials, rstate=np.random.default_rng(seed=30))

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

- fn : 목적 함수
- space : 검색 공간
- algo=tpe.sugget : 베이지안 최적화 적용 알고리즘
- trials : 시도한 입력값 및 목적 함수 반환값 결과 저장
- rstate : 랜덤 seed값

```
[84]    validation_0-auc:0.92659        validation_1-auc:0.83251
[85]    validation_0-auc:0.92671        validation_1-auc:0.83250
100%|██████████| 50/50 [1:10:42<00:00, 84.85s/trial, best loss: -0.8378699584549635]
best: {'colsample_bytree': 0.5802594350237742, 'learning_rate': 0.15309606486156122, 'max_depth': 5.0, 'min_child_weight': 5.0}
```

#02 XGBoost 모델 학습과 하이퍼 파라미터 튜닝

- 도출된 하이퍼 파라미터를 기반으로 XGBClassifier 재학습 및 ROC-AUC 측정
- best에 저장된 최적 하이퍼 파라미터 사용
- n_estimaors 500으로 증가

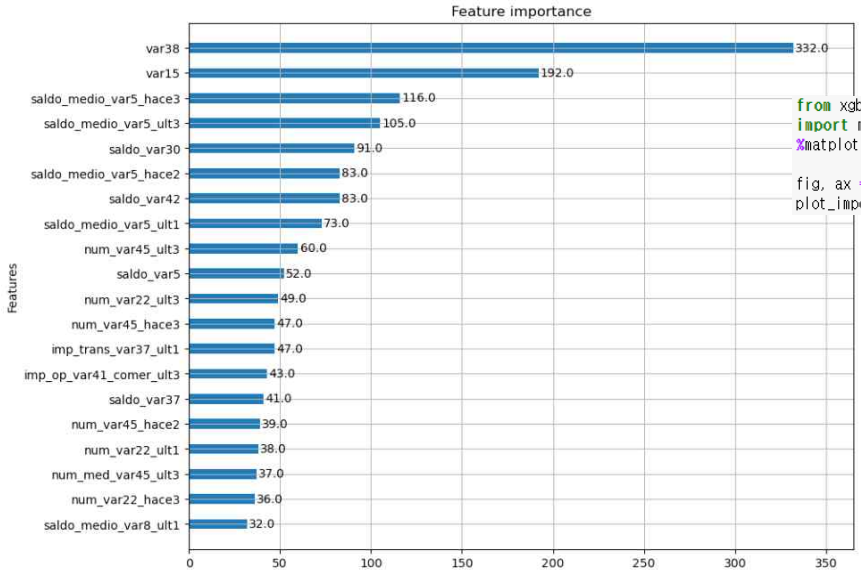
```
# n_estimators를 500증가 후 최적으로 찾은 하이퍼 파라미터를 기반으로 학습과 예측 수행.
xgb_clf = XGBClassifier(n_estimators=500, learning_rate=round(best['learning_rate'], 5),
                        max_depth=int(best['max_depth']), min_child_weight=int(best['min_child_weight']),
                        colsample_bytree=round(best['colsample_bytree'], 5)
                       )

# evaluation metric을 auc로, early stopping은 100 으로 설정하고 학습 수행.
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))
```

```
[179]   validation_0-auc:0.91062        validation_1-auc:0.83327
[180]   validation_0-auc:0.91071        validation_1-auc:0.83330
ROC AUC: 0.8460
```

- 0.8429에서 약간 개선된 결과
- XGBoost는 GBM보다는 빠르지만 여전히 수행 시간이 많이 요구된다.
- 앙상블 계열 알고리즘은 기본적으로 과적합이나 잡음에 뛰어난 알고리즘

#02 XGBoost 모델 학습과 하이퍼 파라미터 튜닝

- 각 피처의 중요도를 그래프로 확인
- xgboost 모듈의 plot_importance() 메서드 이용




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

fig, ax = plt.subplots(1,1,figsize=(10,8))
plot_importance(xgb_clf, ax=ax , max_num_features=20,height=0.4)
```

#03 LightGBM 모델 학습과 하이퍼 파라미터 튜닝

- 앞서 만들어진 데이터 세트를 기반으로 LightGBM 학습 수행 및 ROC-AUC 측정
- LightGBM은 leaf-wise로 트리를 분할하므로 XGBoost보다 학습 시간이 빠르다.
- XGBoost와 동일하게 n_estimators는 500, early_stopping_rounds는 100으로 설정

```
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))
```

```
[141]   training's auc: 0.948038        training's binary_logloss: 0.0923201    valid_1's auc: 0.829218 valid_1's binary_logloss: 0.137
468
[142]   training's auc: 0.948302        training's binary_logloss: 0.0921179    valid_1's auc: 0.829267 valid_1's binary_logloss: 0.137
482
ROC AUC: 0.8384
```

#03 LightGBM 모델 학습과 하이퍼 파라미터 튜닝

- 하이퍼 파라미터 튜닝을 위한 검색 공간 설정하기
- LightGBM은 과적합 가능성이 높으므로 주의해서 설정한다.

```
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)
                    }
```

- num_leaves : 전체 트리의 leaves 수 조절.
- num_leaves = 2^(max_depth)일 경우 depth_wise tree와 같은 수의 leaves를 가진다.
- 따라서 이보다 수를 적게 해야 과적합을 줄일 수 있다.

#03 LightGBM 모델 학습과 하이퍼 파라미터 튜닝

- 목적 함수 생성하기
- 앞 예제와 거의 같다. LGBMClassifier 객체 생성만 수정하기

```
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['subsample'],
                               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 학습과 검증 데이터 세트 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]

        # early stopping은 30회로 설정하고 추출된 학습과 검증 데이터로 XGBClassifier 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)
```

* 여기만 수정

#03 LightGBM 모델 학습과 하이퍼 파라미터 튜닝

- fmin()을 호출하여 최적 하이퍼 파라미터 도출

```
from hyperopt import fmin, tpe, Trials

trials = Trials()

# fmin()함수를 호출. max_evals지정된 횟수만큼 반복 후 목적함수의 최소값을 가지는 최적 입력값 추출.
best = fmin(fn=objective_func, space=lgbm_search_space, algo=tpe.suggest,
            max_evals=50, # 최대 반복 횟수를 지정합니다.
            trials=trials, rstate=np.random.default_rng(seed=30))

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

```
[63]    training's auc: 0.917095        training's binary_logloss: 0.110432     valid_1's auc: 0.834418 valid_1's binary_logloss: 0.137
025
[64]    training's auc: 0.91762 training's binary_logloss: 0.110182     valid_1's auc: 0.834357 valid_1's binary_logloss: 0.137029
100%|██████████████████████████████████████████| 50/50 [08:49<00:00, 10.60s/trial, best loss: -0.8357657786434084]
best: {'learning_rate': 0.08592271133758617, 'max_depth': 121.0, 'min_child_samples': 69.0, 'num_leaves': 41.0, 'subsample': 0.91489580
93027029}
```

#03 LightGBM 모델 학습과 하이퍼 파라미터 튜닝

- 도출된 파라미터를 기반으로 LightGBM 재학습 후 성능 평가하기

```
lgbm_clf =  LGBMClassifier(n_estimators=500, 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 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_val, y_val)])

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

```
[140]   training's auc: 0.947793        training's binary_logloss: 0.0930837    valid_1's auc: 0.824702 valid_1's binary_logloss: 0.138
638
[141]   training's auc: 0.947875        training's binary_logloss: 0.0929829    valid_1's auc: 0.824448 valid_1's binary_logloss: 0.138
739
ROC AUC: 0.8446
```

- 0.8384(튜닝 전)보다 약간 상승했다.
- 0.8460의 정확도를 가진 XGBoost와 비교했을 때, 큰 성능의 차이는 없지만 수행 시간은 훨씬 빨라졌다.

4.10 캐글 신용카드 사기 검출

EWHA EURON

#00 캐글 신용카드 사기 검출

- 캐글 신용카드 사기 데이터의 Class 속성은 불균형한 분포 가지고 있음
  - 0 : 정상 데이터 / 1 : 사기 트랜잭션 (전체 데이터의 0.172%)

- 이상 레이블을 가지는 데이터 건수 너무 적기 때문에 예측 성능 문제 발생 가능
  - 일방적으로 정상 레이블로 치우친 학습 -> 제대로 다양한 유형 학습이 어려움

- 지도학습에서 이런 경우(극도로 불균형한 레이블값 분포):

  오버 샘플링(Oversampling) / 언더 샘플링(Undersampling)

#00 캐글 신용카드 사기 검출

- 언더샘플링(Undersampling)

많은 레이블을 가진 데이터 세트를 적은 레이블을 가진 데이터 세트 수준으로 감소

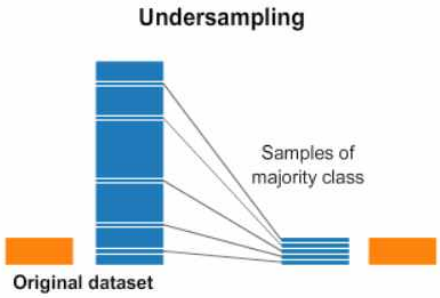
(+) 과도하게 정상 레이블로 학습/예측하는 부작용 개선

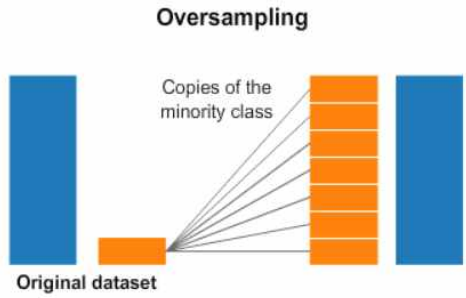
(-) 너무 많은 정상 레이블 데이터 감소 -> 정상 레이블의 경우 제대로 된 학습을 수행할 수 없는 문제

- 오버샘플링(Oversampling)

적은 레이블을 가진 데이터 세트를 많은 레이블을 가진 데이터 세트 수준으로 증식

- 원본 데이터 피처값들을 아주 약간 변경하여 증식
- 예측 성능상 유리한 경우가 많아 상대적으로 더 많이 사용



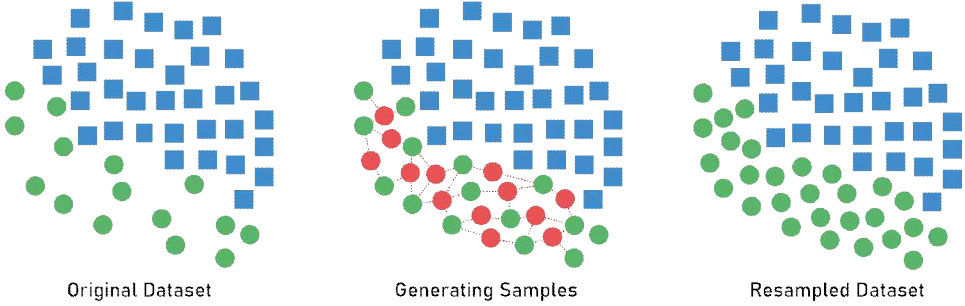


대표적인 오버샘플링 'SMOTE'

: 적은 데이터 세트의 개별 데이터들의 KNN을 찾아서, 데이터와 K개 이웃들의 차이를 일정값으로 만듦 -> 기존 데이터와 약간 차이 나는 새로운 데이터 생성!

#00 캐글 신용카드 사기 검출

- SMOTE (Synthetic Minority Over-Sampling Technique)




```
## SMOTE를 구현한 대표적인 파이썬 패키지 = imbalanced-learn
!pip install imbalanced-learn
```

#01 데이터 1차가공 및 모델 학습/예측/평가

- Time: 큰 의미X -> 제거
- Amount: 신용카드 트랜잭션 금액
- Class: 레이블 (0: 정상, 1: 사기 트랜잭션)
- 결측치 X , Class 레이블만 int, 나머지 float

```
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('/content/creditcard.csv')
card_df.head(3)
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.066928	0.128539	-0.189115	0.133558
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.101288	-0.339846	0.167170	0.125895	-0.008983
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.909412	-0.689281	-0.327642	-0.139097	-0.055353

3 rows × 31 columns

#01 데이터 1차가공 및 모델 학습/예측/평가

- get_preprocessed_df() 생성 (+ 불필요한 Time 컬럼 삭제)

```
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
```

#01 데이터 1차가공 및 모델 학습/예측/평가

- get_train_test_dataset() 생성
  - get_preprocessed_df() 호출 뒤 학습, 테스트 데이터세트 반환
  - 학습/데이터 분리 -> test_size=0.3으로 학습, 테스트 데이터 세트 레이블값 분포 동일하게 만듦

```
def get_train_test_dataset(df=None):

  ## Time feature을 삭제한 df를 받아옴
  df_copy = get_preprocessed_df(df)

  ## X, y 분리
  X_features = df_copy.iloc[:, :-1]
  y_target = df_copy.iloc[:, -1]

  ## train, test으로 데이터셋 분리
  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      ## train, test의 레이블 값 분포도를 동일하게 설정
  )

  return X_train, X_test, y_train, y_test
```

#01 데이터 1차가공 및 모델 학습/예측/평가

- 학습, 테스트 레이블값 비율

```
## train, test 비슷하게 분할 되었는지 label 비율 확인
print(y_train.value_counts() / y_train.shape[0] * 100)
print(y_test.value_counts() / y_test.shape[0] * 100)
```

```
0    99.828453
1     0.171547
Name: Class, dtype: float64
0    99.829122
1     0.170878
Name: Class, dtype: float64
```

V 큰 차이 없이 잘 분할된 것 확인

#01 데이터 1차가공 및 모델 학습/예측/평가

- 신용카드 사기여부 예측하는 모델 생성
- 1. 로지스틱 회귀 이용 (get_clf_eval()함수 불러와서 다시 사용)

```
from sklearn.metrics import f1_score, confusion_matrix, precision_recall_curve, roc_curve
from sklearn.metrics import accuracy_score, precision_score, recall_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 = 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))
```

```
from sklearn.linear_model import LogisticRegression

lr_clf = LogisticRegression(max_iter=1000)

lr_clf.fit(X_train, y_train)
lr_pred = lr_clf.predict(X_test)
lr_pred_proba = lr_clf.predict_proba(X_test)[:, 1]
```

```
오차 행렬
[[85281    14]
 [   57    91]]
정확도: 0.9992, 정밀도: 0.8667, 재현율: 0.6149, F1: 0.7194, AUC:0.9704
```

#01 데이터 1차가공 및 모델 학습/예측/평가

- 앞으로 반복적으로 모델 변경해 학습/예측/평가 => get_model_train_eval() 생성

```
def get_model_train_eval(model, ftr_train=None, ftr_test=None, 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)
```

#01 데이터 1차가공 및 모델 학습/예측/평가

- 2. LightGBM 모델 생성

```
from lightgbm import LGBMClassifier

## model load
lgbm_clf = LGBMClassifier(
    n_estimators=1000,
    num_leaves=64,
    n_jobs=-1,
    boost_from_average=False
)

## train and eval
get_model_train_eval(lgbm_clf, ftr_train=X_train, ftr_test=X_test, tgt_train=y_train, tgt_test=y_test)
```

* 극히 불균형한 레이블값 분포도 → boost_from_average = False

디폴트값이 True이므로 주의!

```
오차 행렬
[[85290     5]
 [   37   111]]
정확도: 0.9995, 정밀도: 0.9569, 재현율: 0.7500, F1: 0.8409, AUC:0.9779
```

- 로지스틱 회귀보다 높은 수치
- (참고) 로지스틱 회귀 결과:

```
오차 행렬
[[85281    14]
 [   57    91]]
정확도: 0.9992, 정밀도: 0.8667, 재현율: 0.6149, F1: 0.7194, AUC:0.9704
```

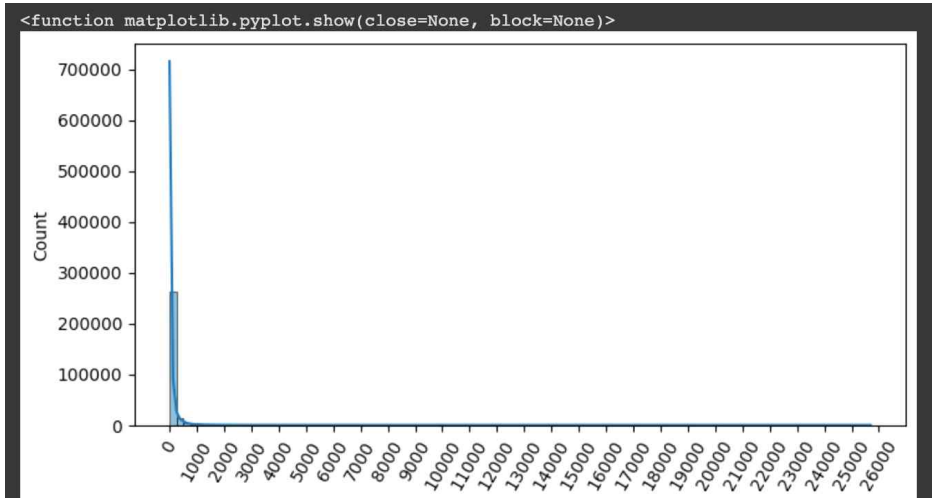
#02 데이터 분포도 변환 후 모델 학습/예측/평가

- 중요 피처값 분포도 살펴보기

* 로지스틱 회귀는 선형모델 => 정규분포형태 선호

- Amount 피처의 분포도 확인하기

```
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
```



- 카드 사용금액 $1000 이하가 대부분
- 꼬리가 긴 형태

#02 데이터 분포도 변환 후 모델 학습/예측/평가

- Amount 표준정규분포 형태로 변환 후 로지스틱 회귀 예측성능 측정

```
from sklearn.preprocessing import StandardScaler

def get_preprocessed_df(df=None):

  ## df copy
  df_copy = df.copy()

  ## Amount 피처를 표준 정규 분포 형태로 변환
  scalar = StandardScaler()
  amount_n = scalar.fit_transform(df_copy['Amount'].values.reshape(-1, 1))

  ## 변환된 피처를 새로운 열로 추가
  df_copy.insert(0, 'Amount_Scaled', amount_n)

  ## 기존의 Time, Amount 피처 삭제
  df_copy.drop(['Time', 'Amount'], axis=1, inplace=True)

  return df_copy
```

#02 데이터 분포도 변환 후 모델 학습/예측/평가

- 함수 수정 후 get_train_test_dataset() 호출 -> 학습/테스트 데이터 생성

-> get_model_train_eval()로 두개 모델 각각 학습, 예측, 평가

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

print('## logistic regression ##')
lr_clf = LogisticRegression(max_iter=1000)
get_model_train_eval(lr_clf, ftr_train=X_train, ftr_test=X_test, tgt_train=y_train, tgt_test=y_test)

print('## lightgbm ##')
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)
```

```
## logistic regression ##
오차 행렬
[[85281    14]
 [   58    90]]
정확도: 0.9992, 정밀도: 0.8654, 재현율: 0.6081, F1: 0.7143, AUC:0.9702
## lightgbm ##
오차 행렬
[[85290     5]
```

- 로지스틱회귀: 정밀도, 재현율 저하
- LightGBM: 정밀도 재현율 약간 저하(큰 성능차이X)

#02 데이터 분포도 변환 후 모델 학습/예측/평가

- 로그 변환 : 데이터 분포 심한 왜곡 시 사용 !
  - 로그값 변환으로 큰 값 => 상대적으로 작은값으로 바꿔주며 왜곡 상당수준 개선
  - numpy의 log1p() 사용

```
def get_preprocessed_df(df=None):

  ## df copy
  df_copy = df.copy()

  ## Amount 피처를 로그 변환
  amount_n = np.log1p(df_copy['Amount'])

  ## 변환된 피처를 새로운 열로 추가
  df_copy.insert(0, 'Amount_Scaled', amount_n)

  ## 기존의 Time, Amount 피처 삭제
  df_copy.drop(['Time', 'Amount'], axis=1, inplace=True)

  return df_copy
```

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

print('## logistic regression ##')
get_model_train_eval(lr_clf, ftr_train=X_train, ftr_test=X_test, tgt_train=y_train, tgt_test=y_test)
```

```
## logistic regression ##
오차 행렬
[[85283    12]
 [   59    89]]
정확도: 0.9992, 정밀도: 0.8812, 재현율: 0.6014, F1: 0.7149, AUC:0.9727
```

- Amount 피처 로그변환 뒤, 로지스틱 회귀와 LightGBM 모델 적용하여 예측 성능 확인
  - 로지스틱 회귀) 정밀도 향상 + 재현율 저하
  - LightGBM: 재현율 향상

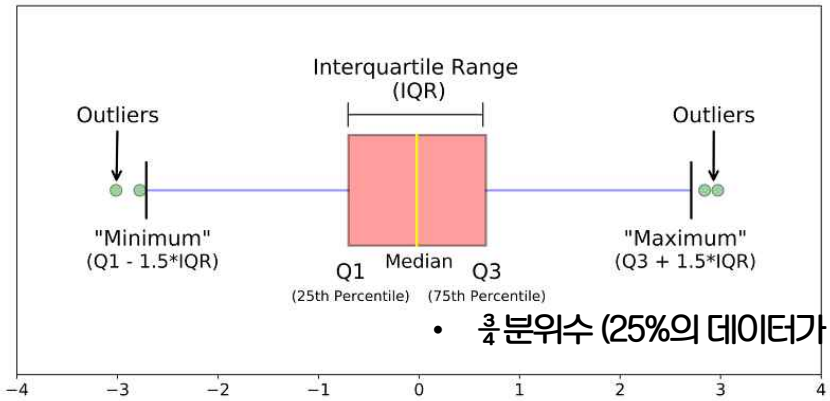
- 불균일한 레이블의 데이터셋에서 로지스틱회귀는 불안정한 성능결과 보여줌!

#03 이상치 데이터 제거 후 모델 학습/예측/평가

- 이상치 데이터(Outlier):

전체 데이터 패턴에서 벗어난 이상값을 가진 데이터, 성능에 영향 줄 수 있음

- IQR(Inter Quantile Range)



- $\frac{3}{4}$ 분위수 (25%의 데이터가 이 값보다 큼!)

#03 이상치 데이터 제거 후 모델 학습/예측/평가

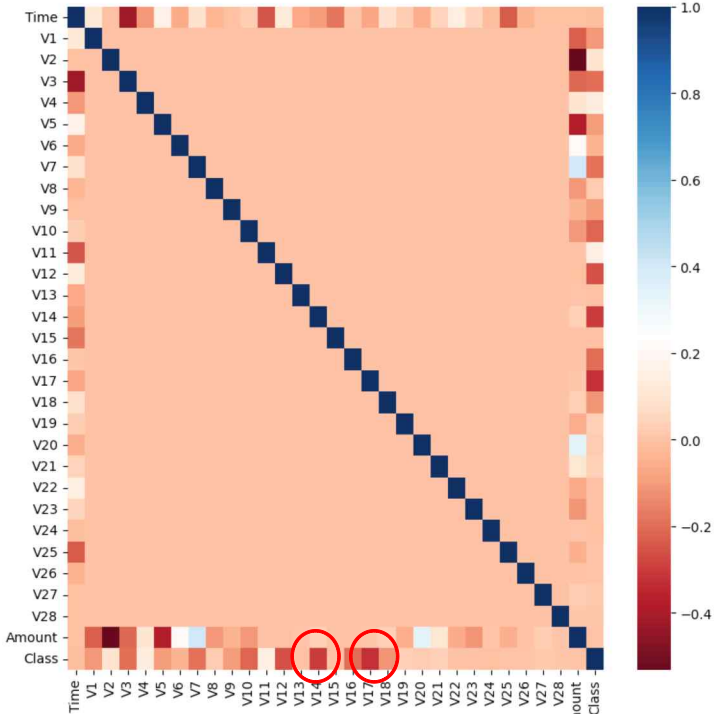
- IQR 통해 이상치 데이터 제거하기!
- 레이블과 상관성 높은 피처 위주로 이상값 검출하는 것이 좋음

(모든 이상치 검출은 시간 비효율적, 상관성 높지 않다면 크게 성능향상 기여X)

- 피처별 상관도 (DataFrame의 corr() 사용) + 시각화(Seaborn heatmap)
  - RdBu: 양의 상관관계 높을 수록 진한 파란색, 음의 상관관계 높을수록 진한 빨간색

```
import seaborn as sns

plt.figure(figsize=(9, 9))
corr = card_df.corr()           ## dataframe의 각 피처별로 상관도를 구함
sns.heatmap(corr, cmap='RdBu')  ## 상관도를 시본의 heatmap으로 시각화
```

#03 이상치 데이터 제거 후 모델 학습/예측/평가



- Class 피처와 V14, V17의 음의 상관관계 가장 높음
- V14에 대해 이상치 제거

```python
import numpy as np

def get_outlier(df=None, column=None, weight=1.5):
  fraud = df[df['Class']==1][column]
  quanitile_25 = np.percentile(fraud.values, 25)
  quanitile_75 = np.percentile(fraud.values, 75)

  ## IQR 구하기
  iqr = quanitile_75 - quanitile_25
  iqr_weight = iqr * weight

  lowest_val = quanitile_25 - iqr_weight
  highest_val = quanitile_75 + iqr_weight

  outlier_index = fraud[(fraud < lowest_val) | (fraud > highest_val)].index
  return outlier_index
```

```python
outlider_index = get_outlier(df=card_df, column='V14', weight=1.5)
print('이상치 데이터 인덱스:', outlider_index)
```

```
이상치 데이터 인덱스: Int64Index([8296, 8615, 9035, 9252], dtype='int64')
```

#03 이상치 데이터 제거 후 모델 학습/예측/평가

- 데이터 가공: get_outlier()로 이상치 추출 -> get_processed_df()에 추가
- 로지스틱회귀, LightBGM 다시 적용

```
def get_preprocessed_df(df=None):

  ## df copy
  df_copy = df.copy()

  ## Amount 피처를 로그 변환
  amount_n = np.log1p(df_copy['Amount'])

  ## 변환된 피처를 새로운 열로 추가
  df_copy.insert(0, 'Amount_Scaled', amount_n)

  ## 기존의 Time, Amount 피처 삭제
  df_copy.drop(['Time', 'Amount'], axis=1, inplace=True)

  ## 이상치 데이터는 삭제하는 로직 추가
  outlier_index = get_outlier(df_copy, 'V14', 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(card_df)
print('## logistic regression ##')
get_model_train_eval(lr_clf, ftr_train=X_train, ftr_test=X_test, tgt_train=y_train, tgt_test=y_test)

print('## lightgbm ##')
get_model_train_eval(lgbm_clf, ftr_train=X_train, ftr_test=X_test, tgt_train=y_train, tgt_test=y_test)
```

```
## logistic regression ##
오차 행렬
[[85281    14]
 [   48    98]]
정확도: 0.9993, 정밀도: 0.8750, 재현율: 0.6712, F1: 0.7597, AUC:0.9743
## lightgbm ##
오차 행렬
[[85290     5]
 [   25   121]]
정확도: 0.9996, 정밀도: 0.9603, 재현율: 0.8288, F1: 0.8897, AUC:0.9780
```

- 결과적으로 모두 예측 성능 향상
  - 로지스틱회귀) 재현율 60.14->67.12%
  - LightGBM) 재현율 76.35->82.88%

#04 SMOTE 오버샘플링 적용 후 모델 학습/예측/평가

- SMOTE 기법으로 오버샘플링 -> 예측성능 평가
- imbalanced-learn 패키지의 SMOTE 클래스 사용
- 주의) 학습 데이터 세트만 오버 샘플링!
  (검증/테스트 데이터 세트 오버 샘플링할 경우 올바른 검증/테스트 불가)
- fit_resample() 이용하여 증식 (전과 비교)
  - 2배로 데이터 증식

```
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=0)

## SMOTE 오버 샘플링 적용
X_train_over, y_train_over = smote.fit_resample(X_train, y_train)

## 데이터 확인

print('SMOTE 적용 전 학습용 데이터셋:', X_train.shape, y_train.shape)


print('SMOTE 적용 후 학습용 데이터셋:',X_train_over.shape, y_train_over.shape)


## SMOTE 적용 후 레이블 값 분포
print('SMOTE 적용 후 레이블 값 분포:\n',pd.Series(y_train_over).value_counts())
```

```
SMOTE 적용 전 학습용 데이터셋: (199362, 29) (199362,)
SMOTE 적용 후 학습용 데이터셋: (398040, 29) (398040,)
SMOTE 적용 후 레이블 값 분포:
 0    199020
1    199020
Name: Class, dtype: int64
```

#04 SMOTE 오버샘플링 적용 후 모델 학습/예측/평가

- 로지스틱 회귀 모델 성능평가

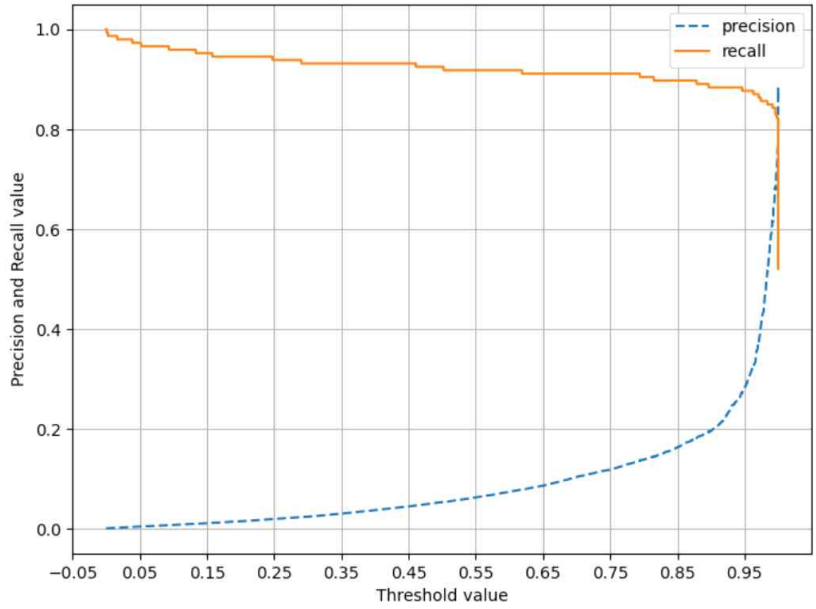
```
lr_clf = LogisticRegression(max_iter=1000)
#ftr_train과 tgt_train 인자값이 SMOTE 증식된 X_train_over와 y_train_over로 변경됨에 유의
get_model_train_eval(lr_clf, ftr_train = X_train_over, ftr_test= X_test, tgt_train = y_train_over, tgt_test = y_test)
```

```
오차 행렬
[[82937  2358]
 [   11   135]]
정확도: 0.9723, 정밀도: 0.0542, 재현율: 0.9247, F1: 0.1023, AUC:0.9737
```

- 재현율 증가하지만, 정밀도 0.0542 (급격히 저하)
  - 예측을 지나치게 class=1로 적용하여 정밀도 떨어진 것(적용 불가)

#04 SMOTE 오버샘플링 적용 후 모델 학습/예측/평가

```
def precision_recall_curve_plot(y_test, pred_proba_c1):
    # threshold ndarray와 이 threshold에 따른 정밀도, 재현율 ndarray 추출
    precisions, recalls, thresholds = precision_recall_curve(y_test, pred_proba_c1)

    # x축을 threshold 값, y축을 정밀도, 재현율로 그리기
    plt.figure(figsize=(8,6))
    thresholds_boundary = thresholds.shape[0]
    plt.plot(thresholds, precisions[0: thresholds_boundary], linestyle= '--', label='precision')
    plt.plot(thresholds, recalls[0: thresholds_boundary], label='recall')

    # threshold의 값 x축의 scale을 0.1 단위로 변경
    stard, end = plt.xlim()
    plt.xticks(np.round(np.arange(stard, end, 0.1), 2))

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

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



- 0.99 threshold 이하에서는 재현율 매우 좋고 정밀도 극단적으로 낮다가, 0.99 이상에서는 재현율 낮고 정밀도 높은 것 확인
- 로지스틱 회귀 모델에서는 SMOTE 적용 후 예측모델 성능 좋지 않음

#04 SMOTE 오버샘플링 적용 후 모델 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
[[85283    12]
 [   22   124]]
정확도: 0.9996, 정밀도: 0.9118, 재현율: 0.8493, F1: 0.8794, AUC:0.9814
```

- LightGBM에서 SMOTE로 오버샘플링했을 때 재현율은 높아졌지만 정밀도는 낮아짐 (일반적)
- 정밀도보다는 재현율을 높이는 게 머신러닝의 주요 목표인 경우 SMOTE 적용하면 좋음!

THANK YOU

EWHA EURON