

## ▼ 분류 실습 1 - 캐글 산탄데르 고객 만족 예측

370개의 피처로 주어진 캐글 산탄데르 데이터 세트에서 고객 만족 여부를 예측하기

- TARGET 값이 1이면 불만을 가진 고객, 0이면 만족한 고객
- 모델의 성능 평가는 ROC-AUC(ROC 곡선 영역)로 평가(대부분이 만족이고 불만족 데이터를 일부이기에 정확도 수치 사용 X)
- train\_santander.csv 파일 사용

### • 데이터 전처리

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

cust_df = pd.read_csv('train_santander.csv', encoding='latin-1')
print('데이터셋의 크기:', cust_df.shape)
cust_df.head(3)
cust_df.info()
```

- 데이터셋의 크기: (76020, 371)
- 타겟값을 포함한 피처 371개 존재
- 111개의 피처가 float형, 260개의 피처가 int형 (모든 피처가 숫자형)
- NULL값은 없음

```
cust_df[cust_df['TARGET']==1].TARGET.count()
print('불만족 고객의 비율:', cust_df[cust_df['TARGET']==1].TARGET.count()/cust_df.TARGET.count())
```

```
3008
불만족 고객의 비율: 0.0395685345961589
```

- TARGET 값의 분포는 대부분이 만족이며, 불만족인 고객은 4%에 불과

+cust\_df.describe() 를 해서 각 피처의 값 분포를 확인해보았을 때, var3 피처의 경우 min 값이 -999999가 나오고, 이는 NaN 이나 특정 예외 값을 변경한 것으로 보임

- -999999값이 116개 존재, 다른 값들에 비해 너무 편차가 심하므로 최빈값인 2로 변경

```
cust_df['var3'].replace(-999999, 2, inplace=True)
#ID 피처는 단순 식별자이므로 피처 Drop
cust_df.drop('ID', axis=1, inplace=True)

#피처 데이터 세트와 target 값 분리
X_features = cust_df.drop('TARGET', axis=1)
y_labels = cust_df['TARGET']
```

+train\_test\_split()으로 학습, 테스트 데이터 세트 분리

- 만족이 많고 불만족이 적은 비대칭적인 데이터셋이므로 비율 확인 필요!

```
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=123)
print('학습 세트 shape:', X_train.shape, '테스트 세트 shape:', X_test.shape)
print('학습 세트 target 값 분포 비율')
print(y_train.value_counts()/y_train.count())
print('테스트 세트 target 값 분포 비율')
print(y_test.value_counts()/y_test.count())
```

```
학습 세트 shape: (60816, 369) 테스트 세트 shape: (15204, 369)
학습 세트 target 값 분포 비율
0    0.960652
1    0.039348
Name: TARGET, dtype: float64
테스트 세트 target 값 분포 비율
0    0.95955
1    0.04045
Name: TARGET, dtype: float64
```

⇒ train, test 데이터 세트 모두 TARGET 값의 분포가 원본 데이터와 비슷하게 4%정도로 만들어짐

#### • XGBoost 모델 학습과 하이퍼 파라미터 튜닝

- 사이킷런 래퍼 XGBoost 기반으로 학습 모델 생성 및 예측 결과를 ROC, AUC로 평가
- n\_estimators=500, early\_stopping\_rounds=100, eval\_metric='auc'로 설정
- 학습 모델을 평가하는 평가 데이터 세트를 원래 새로운 데이터 세트를 사용해야 하지만 여기에선 그냥 test 데이터 세트 사용 (과적합 위험 있음), eval\_set=[(X\_train, y\_train), (X\_test, y\_test)]

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

xgb_clf = XGBClassifier(n_estimators=500, random_state=156)

#성능 평가 지표를 auc로, 조기 중단 파라미터는 100으로 설정하고 학습 수행
xgb_clf.fit(X_train, y_train, early_stopping_rounds=100, eval_metric='auc', eval_set=[(X_train, y_train), (X_test, y_test)])
xgb_roc_score = roc_auc_score(y_test, xgb_clf.predict_proba(X_test)[:,:1], average='macro')
print('ROC AUC', xgb_roc_score)
```

ROC AUC 0.8277512236360283

? eval\_set는 언제는 [(X\_train, y\_train), (X\_test, y\_test)]로 하고 언제는 [(X\_test, y\_test)]로 하는가

eval\_set은 evaluation 세트, 즉 검증 세트를 지정하는 것입니다. fit() 수행시 학습하면서 반복적으로 예측 오류값을 줄일 수 있도록 학습이 진행되는데 이때 학습은 학습 데이터로 하되, 예측 오류값 평가는 eval\_set로 지정된 검증 세트로 평가하는 방식입니다. 학습 데이터로만 예측 오류값을 줄이게 되면 오버피팅 우려가 높아서 별도의 검증 세트를 지정하여 수행합니다.

eval\_set=[(X\_train, y\_train), (X\_test, y\_test)] 는 학습은 (X\_train, y\_train), 검증은 (X\_test, y\_test) 데이터 세트를 이용하라는 것입니다. 하지만 eval\_set=[(X\_test, y\_test)]으로 하는게 더 명확한 설정인데 eval\_set=[(X\_train, y\_train), (X\_test, y\_test)]로 해서 오히려 이해를 더 어렵게 만든것 같습니다.

그리고 eval\_set=[(X\_test, y\_test)]로 테스트 데이터 세트를 설정하였는데, 원칙적으로는 이렇게 설정하면 안됩니다. 원래는 별도의 validation 데이터 세트를 넣어야 하는데, 데이터가 많지 않아서 test dataset을 설정하였으니 양해 부탁드립니다. 물론 eval\_set=[(X\_train, y\_train)] 로 설정하실 필요도 마찬가지 이유로 없습니다.

피처의 개수가 많으므로 과적합의 가능성 가정

→ max\_depth, min\_child\_weight, colsample\_bytree 하이퍼 파라미터만 일차 튜닝 대상



학습 시간이 많이 필요한 ML 모델의 하이퍼 파라미터 튜닝 요령

1. 먼저 2~3개 정도의 파라미터를 결합해 최적 파라미터 찾아내기
2. 이 최적 파라미터를 기반으로 다시 1~2개 파라미터를 결합해 파라미터 튜닝 수행

하이퍼 파라미터 튜닝을 통해 GridSearchCV에서 재학습된 Estimator에서 ROC-AUC 수치가 얼마나 향상되는지 check!

```
from sklearn.model_selection import GridSearchCV

#하이퍼 파라미터 테스트의 수행 속도를 향상시키기 위해 n_estimators를 100으로 감소 시키기
xgb_clf = XGBClassifier(n_estimators=100, random_state=123)

#colsample_bytree는 트리 생성에 필요한 피처를 임의로 샘플링하는데 사용(매우 많은 피처가 있는 경우 과적합을 조정하는데 이용)
params = {'max_depth':[5,7], 'min_child_weight':[1,3], 'colsample_bytree':[0.5,0.75]}

#cv는 3으로 지정
gridcv = GridSearchCV(xgb_clf, param_grid = params, cv=3)
gridcv.fit(X_train, y_train, early_stopping_rounds=30, eval_metric='auc', eval_set=[(X_train, y_train), (X_test, y_test)])
print('GridSearchCV 최적 파라미터:', gridcv.best_params_)

xgb_roc_score = roc_auc_score(y_test, gridcv.predict_proba(X_test)[:,-1], average='macro')
print('ROC AUC:', xgb_roc_score)
```

```
GridSearchCV 최적 파라미터: {'colsample_bytree': 0.5, 'max_depth': 5, 'min_child_weight': 3}
ROC AUC: 0.8282308142842892
```

→ 약간 개선됨

→ 주어진 하이퍼파라미터 + 다양한 하이퍼파라미터 추가를 통해 학습 모델 생성

```
#n_estimators는 1000으로 증가, learning_rate=0.2로 감소, reg_alpha=0.03 추가
xgb_clf = XGBClassifier(n_estimators=1000, learning_rate=0.2, reg_alpha=0.03,
                        colsample_bytree=0.5, max_depth=5, min_child_weight=3, random_state=123)

xgb_clf.fit(X_train, y_train, early_stopping_rounds=200,
            eval_metric='auc', eval_set=[(X_test, y_test)])
xgb_roc_score = roc_auc_score(y_test, xgb_clf.predict_proba(X_test)[:,-1], average='macro')
print('ROC AUC:', xgb_roc_score)
```

```
ROC AUC: 0.8313533918806184
```

→ 앙상블 계열 알고리즘에서 하이퍼 파라미터 튜닝으로 성능 수치 개선이 급격하게 되는 경우는 많지 않음 (과적합이나 잡음에 기본적으로 뛰어나기 때문)

#### • LightGBM 모델 학습과 하이퍼 파라미터 튜닝

XGBoost와 동일하게 n\_estimators=500, early\_stopping\_rounds=100으로 설정

```
#LGBM
from lightgbm import LGBMClassifier

lgbm_clf = LGBMClassifier(n_estimators=500, random_state=123)
evals=[(X_test, y_test)]
lgbm_clf.fit(X_train, y_train, eval_metric='auc', eval_set=evals, early_stopping_rounds=100, verbose=True)
```

```
lgbm_roc_score = roc_auc_score(y_test, lgbm_clf.predict_proba(X_test)[: ,1], average='macro')
print('ROC AUC:', lgbm_roc_score)
```

ROC AUC: 0.832072443488161

→ XGBoost보다 학습에 걸리는 시간이 단축됨

→ 더 다양한 하이퍼 파라미터를 튜닝해보자! (num\_leaves, max\_depth, min\_child\_samples, subsample)

```
from sklearn.model_selection import GridSearchCV

lgbm_clf = LGBMClassifier(n_estimators=200, random_state=123)
params = {'num_leaves': [32, 64], 'max_depth': [128, 160], 'min_child_samples': [60, 100],
          'subsample': [0.8, 1]}
gridcv = GridSearchCV(lgbm_clf, param_grid=params, cv=3)
gridcv.fit(X_train, y_train, early_stopping_rounds=30, eval_metric='auc', eval_set=evals)

print('GridSearchCV 최적 하이퍼 파라미터:', gridcv.best_params_)
lgbm_roc_score = roc_auc_score(y_test, gridcv.predict_proba(X_test)[: ,1], average='macro')
print('ROC AUC:', lgbm_roc_score)
```

GridSearchCV 최적 하이퍼 파라미터: {'max\_depth': 128, 'min\_child\_samples': 60, 'num\_leaves': 32, 'subsample': 0.8}  
ROC AUC: 0.8328069316062274

→ 해당 파라미터로 LGBM 학습모델 생성하자

```
lgbm_clf = LGBMClassifier(n_estimators=1000, max_depth=128, min_child_samples=60, num_leaves=32, subsample=0.8)

lgbm_clf.fit(X_train, y_train, eval_metric='auc', eval_set=evals, early_stopping_rounds=100, verbose=True)
lgbm_roc_score = roc_auc_score(y_test, lgbm_clf.predict_proba(X_test)[: ,1], average='macro')
print('ROC AUC:', lgbm_roc_score)
```

ROC AUC: 0.8328069316062274

⚙️ lgbm을 fit하는 단계에서 쓰는 verbose는 만약 verbose=10으로 설정한다면 n\_estimators를 기준으로 10번 마다 결과 값을 출력하는 옵션

## ▼ 분류 실습 2 - 캐글 신용카드 사기 검출

### 신용카드 사기 검출 분류

→ Class는 0과 1로 분류 (0은 사기가 아닌 정상, 1은 신용카드 사기 트랜잭션)

→ 전체 데이터의 약 0.172%만이 사기 트랜잭션(매우 불균형한 분포)

#### • 언더샘플링과 오버 샘플링

레이블이 불균형한 분포를 가진 데이터 세트를 학습할 때 예측 성능에 문제 발생

(이상 레이블을 가지는 데이터 건수가 정상 레이블을 가진 데이터 건수에 비해 너무 적기 때문)

→ 오버 샘플링(Oversampling), 언더 샘플링(Undersampling)으로 적절한 학습 데이터 확보

→ 오버 샘플링이 예측 성능상 더 유리한 경우가 많음



### 1. 언더 샘플링

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

### 2. 오버 샘플링

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

→ 언더 샘플링은 과도하게 정상 레이블로 학습/예측하는 부작용을 개선할 순 있지만 너무 많은 정상 레이블 데이터를 감소시키기에 오히려 제대로 된 학습을 수행할 수 없을 수 있음(잘 적용x)

→ 오버 샘플링은 원본 데이터의 피쳐 값들을 아주 약간 변경하여 증식(SMOTE 방법)

## +SMOTE 방법이란?

: 적은 데이터 세트에 있는 개별 데이터들의 **K 최근접 이웃(K Nearest Neighbor)**을 찾아서 이 데이터와 k개의 이웃들의 차이를 일정한 값으로 만들어 기존 데이터와 약간 차이가 나는 새로운 데이터를 생성하는 방법

→ 대표적인 파이썬 패키지: imbalanced-learn

## • 데이터 일차 가공 및 모델 학습/예측/평가

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

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

→ Time 칼럼은 큰 의미가 없는 피쳐, 제거해야 함

→ 전체 284,807 레코드에서 null 값은 없음, class 레이블만 int형, 나머지 피쳐는 float형

### 1. get\_preprocessed\_df(), get\_train\_test\_dataset() 함수 만들기

→ 먼저 인자로 입력된 DataFrame을 복사한 뒤, 이를 가공하여 반환(get\_preprocessed\_df())

→ get\_preprocessed\_df()를 호출한 뒤 학습 피쳐/레이블 데이터 세트와 테스트 피쳐/레이블 데이터 세트를 반환(get\_train\_test\_dataset())

```
from sklearn.model_selection import train_test_split

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

#사전 데이터 가공 후 학습과 테스트 데이터 세트를 반환하는 함수
def get_train_test_dataset(df=None):
    #인자로 입력된 DataFrame의 사전 데이터 가공이 완료된 복사 DataFrame 반환
    df_copy = get_preprocessed_df(df)
    #DataFrame의 맨 마지막 칼럼이 레이블, 나머지는 피쳐들임
    X_features = df_copy.drop('Class', axis=1)
    y_target = df_copy['Class']
    #train_test_split()으로 학습과 테스트 데이터 분할
    #stratify=y_target으로 Stratified 기반 분할
    X_train, X_test, y_train, y_test = train_test_split(X_features, y_target, test_size=0.3, random_state=123,
                                                        stratify=y_target)

    #학습, 테스트 데이터세트 반환
    return X_train, X_test, y_train, y_test

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

```
#비슷하게 분류되었는지 check
print('학습 데이터 레이블 값 비율')
print(y_train.value_counts()/y_train.count())
print('테스트 데이터 레이블 값 비율')
print(y_test.value_counts()/y_test.count())
```

```
학습 데이터 레이블 값 비율
0    0.998275
1    0.001725
Name: Class, dtype: float64
테스트 데이터 레이블 값 비율
0    0.998268
1    0.001732
Name: Class, dtype: float64
```

⇒ train, test 데이터 세트 모두 target 값의 분포가 원본 데이터와 비슷하게 0.172%정도로 만들어짐

## 2. 모델 만들기(Logistic Regression, LightGBM)

```
#로지스틱 회귀
from sklearn.linear_model import LogisticRegression

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

#3장에서 이용한 get_clf_eval() 함수 이용
get_clf_eval(y_test, lr_pred, lr_pred_proba)
```

```
오차 행렬
[[85278   17]
 [   65   83]]
정확도: 0.9990, 정밀도: 0.8300, 재현율: 0.5608,   F1: 0.6694, AUC:0.7803
```

앞으로 수행할 예제 코드에서 반복적으로 모델을 변경하며 학습, 예측, 평가를 할 것이므로 이를 위한 별도의 함수 생성 (get\_model\_train\_eval())

→ 인자로 사이킷런의 Estimator 객체와 학습, 테스트 데이터 세트를 입력 받아서 학습, 예측, 평가 수행

```
#인자로 사이킷런의 Estimator 객체와 학습, 테스트 데이터 세트를 입력 받아서 학습, 예측, 평가 수행
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)
```

🤖 본 데이터 세트는 극도로 불균형한 레이블 값 분포도를 갖고 있으므로 LGBMClassifier 객체 생성 시 **boost\_from\_average = False**로 파라미터 설정

(디폴트가 True로 되어있는데, 불균형한 데이터 셋에서 그대로 적용하면 ROC-AUC 성능을 크게 저하시킴)

```
#LightGBM 모델 학습
from lightgbm import LGBMClassifier

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

```
오차 행렬
[[85292 3]
 [ 39 109]]
정확도: 0.9995, 정밀도: 0.9732, 재현율: 0.7365, F1: 0.8385, AUC:0.8682
```

→ 앞서 했던 로지스틱 회귀보다 높은 수치를 나타냄

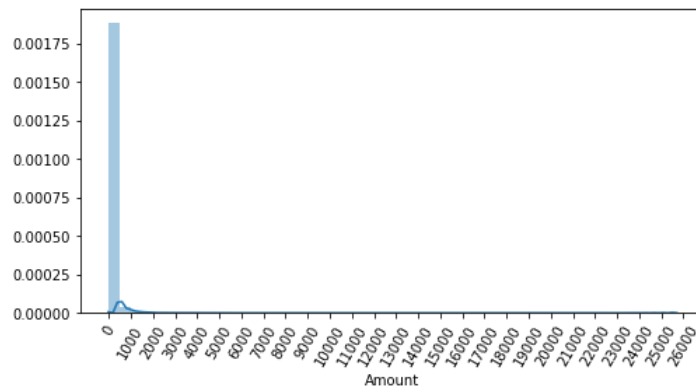
### • 데이터 분포도 변환 후 모델 학습/예측/평가

→ 왜곡된 분포도를 가지는 데이터를 재가공한 뒤에 모델을 다시 테스트

→ 로지스틱 회귀는 선형모델, 대부분의 선형모델은 **중요 피쳐들의 값이 정규 분포 형태를 유지하는 것**을 선호

⇒ Amount 피쳐의 분포도를 살펴보고 **표준 정규 분포 형태로 변환**하자!

```
#데이터 분포도 변환 후 모델 학습/예측/평가
import seaborn as sns
plt.figure(figsize=(8,4))
plt.xticks(range(0,30000,1000), rotation=60)
sns.distplot(card_df['Amount'])
```



→ 카드 사용 금액이 1000불 이하인 데이터가 대부분(긴 꼬리 형태의 분포)

→ **표준 정규 분포 형태로 변환**하고 로지스틱 회귀의 예측 성능을 측정해보자!

(사이킷런의 StandardScaler 클래스 이용해 Amount 피쳐를 정규 분포 형태로 변환)

(앞서 정의했던 get\_preprocessed\_df()함수 수정)

```
from sklearn.preprocessing import StandardScaler
#사이킷런의 StandardScaler를 이용해 정규 분포 형태로 Amount 피쳐값을 변환하는 로직으로 수정
def get_preprocessed_df(df=None):
    df_copy = df.copy()
    scaler = StandardScaler()
    amount_n = scaler.fit_transform(df_copy['Amount'].values.reshape(-1,1))
    #변환된 Amount를 Amount_Scaled로 피쳐명 변경 후 DataFrame 맨 앞 칼럼으로 입력
    df_copy.insert(0, 'Amount_Scaled', amount_n)
    # 기존 Time, Amount 피쳐 삭제
    df_copy.drop(['Time', 'Amount'], axis=1, inplace=True)
    return df_copy
```

get\_model\_train\_eval()을 이용해 로지스틱 회귀와 LightGBM 모델 학습,예측,평가


```
#Amount를 정규 분포 형태로 변환 후 로지스틱 회귀 및 LightGBM 수행
X_train, X_test, y_train, y_test = get_train_test_dataset(card_df)

print('로지스틱 회귀 예측 성능')
lr_clf = LogisticRegression()
get_model_train_eval(lr_clf, ftr_train=X_train, ftr_test=X_test, tgt_train=y_train, tgt_test=y_test)
```

```
print('LightGBM 예측 성능')
lgbm_clf = LGBMClassifier()
get_model_train_eval(lgbm_clf, ftr_train=X_train, ftr_test=X_test, tgt_train=y_train, tgt_test=y_test)
```

```
로지스틱 회귀 예측 성능
오차 행렬
[[85281    14]
 [    60    88]]
정확도: 0.9991, 정밀도: 0.8627, 재현율: 0.5946,    F1: 0.7040, AUC:0.7972
LightGBM 예측 성능
오차 행렬
[[85145    150]
 [    64    84]]
정확도: 0.9975, 정밀도: 0.3590, 재현율: 0.5676,    F1: 0.4398, AUC:0.7829
```

⇒ 두 모델 모두 변환 이전과 비교해 성능이 크게 개선되지는 않음

 StandardScaler가 아닌 **로그 변환**을 수행해보자!

- 로그 변환은 데이터 분포도가 심하게 왜곡되어 있을 경우 적용하는 중요 기법 중 하나
- 원래 값을 log 값으로 변환해 큰 값을 상대적으로 작은 값으로 변환(왜곡을 상당수준 개선)

(get\_preprocessed\_df())를 또 수정하자)

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

```
#Amount를 로그 변환 후 로지스틱 회귀 및 LightGBM 수행
X_train, X_test, y_train, y_test = get_train_test_dataset(card_df)

print('로지스틱 회귀 예측 성능')
lr_clf = LogisticRegression()
get_model_train_eval(lr_clf, ftr_train=X_train, ftr_test=X_test, tgt_train=y_train, tgt_test=y_test)
print('LightGBM 예측 성능')
lgbm_clf = LGBMClassifier()
get_model_train_eval(lgbm_clf, ftr_train=X_train, ftr_test=X_test, tgt_train=y_train, tgt_test=y_test)
```

```
로지스틱 회귀 예측 성능
오차 행렬
[[85282    13]
 [    60    88]]
정확도: 0.9991, 정밀도: 0.8713, 재현율: 0.5946,    F1: 0.7068, AUC:0.7972
LightGBM 예측 성능
오차 행렬
[[85168    127]
 [    52    96]]
정확도: 0.9979, 정밀도: 0.4305, 재현율: 0.6486,    F1: 0.5175, AUC:0.8236
```

⇒ 두 모델 모두 약간씩 성능 개선

#### • 이상치 데이터 제거 후 모델 학습/예측/평가

- 이상치(outlier)로 인해 머신러닝 모델의 성능에 영향을 받는 경우가 발생하기 쉬움
- 이상치를 찾는 방법은 여러가지, 그 중 **IQR(사분위 값의 편차를 이용하는 기법)**이 대표적
- $IQR = Q3(3/4지점) - Q1(1/4지점)$
- 보통 **IQR에 1.5**를 곱해서 생성된 범위를 이용해 최댓값과 최솟값을 결정한 뒤 **최댓값을 초과하거나 최솟값에 미달하는 데이터를 이상치로 간주**



→ **최댓값:  $Q3 + 1.5 \times IQR$  / 최솟값:  $Q1 - 1.5 \times IQR$**  (1.5가 아닌 다른 수를 곱해도 됨, 일반적으로는 1.5)

1. 결정값(레이블)과 **가장 상관성이 높은 피쳐들**을 골라내자

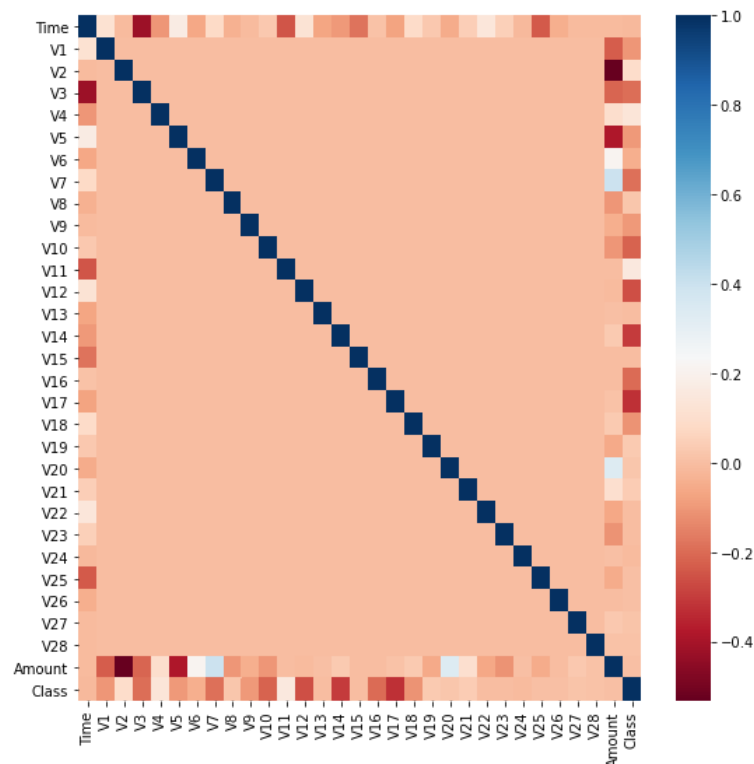
→ 모든 피쳐들의 이상치를 검출하는 것은 시간이 많이 소모됨.

→ 결정값과 상관성이 높지 않은 피쳐들의 경우는 이상치를 제거하더라도 크게 성능 향상이 되지 않음

→ DataFrame의 `corr()`을 이용해 각 피쳐별로 상관도를 구하자

```
import seaborn as sns

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



→ 양의 상관관계가 높을수록 진한 파란색, 음의 상관관계가 높을수록 진한 빨간색

→ 결정 레이블인 Class 피쳐와 음의 상관관계가 가장 높은 피쳐는 **V14, V17**

(이 중 V14에 대해서만 이상치를 찾아서 제거해보자!)

2. **IQR을 이용해** 이상치를 검출하는 함수 생성 및 이상치 삭제 (`get_outlier()`)

→ 인자로 DataFrame과 이상치를 검출한 칼럼을 입력받음

→ 해당 이상치가 있는 DataFrame Index를 반환하자

```
import numpy as np

def get_outlier(df=None, column=None, weight=1.5):
    #fraud(사기)에 해당하는 column 데이터만 추출, Q1과 Q3 지점을 np.percentile로 구함
    fraud = df[df['Class']==1][column] #왜 사기에 해당하는 column 데이터만 추출?
    quantile_25 = np.percentile(fraud.values, 25)
    quantile_75 = np.percentile(fraud.values, 75)
    #IQR을 구하고 IQR에 1.5를 곱해 최댓값과 최솟값 지점 구하기
    iqr = quantile_75 - quantile_25
```

```

iqr_weight = iqr*weight
lowest_val = quantile_25 - iqr_weight
highest_val = quantile_75 + iqr_weight
#최대값보다 크거나 최소값보다 작은 값을 이상치 데이터로 설정하고 DataFrame index 반환
outlier_index = fraud[(fraud < lowest_val) | (fraud > highest_val)].index
return outlier_index

outlier_index = get_outlier(df=card_df, column='V14', weight=1.5)
print('이상치의 인덱스:', outlier_index)

```

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

→ 총 4개의 데이터인 8296, 8615, 9035, 9252번 index가 이상치로 추출됨

→ 이를 이용해 이상치를 추출하고 삭제하는 로직을 get\_preprocessed\_df() 함수에 추가하자

```

#get_preprocessed_df()를 로그 변환 후 V14 피쳐의 이상치 데이터를 삭제하는 로직
def get_preprocessed_df(df=None):
    df_copy = df.copy()
    amount_n = np.log1p(df_copy['Amount'])
    df_copy.insert(0, 'Amount_Scaled', amount_n)
    df_copy.drop(['Time', 'Amount'], axis=1, inplace=True)
    #이상치 데이터를 삭제하는 로직 추가
    outlier_index = get_outlier(df=df_copy, column='V14', weight=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('## 로지스틱 회귀 예측 성능 ##')
get_model_train_eval(lr_clf, ftr_train=X_train, ftr_test=X_test, tgt_train=y_train, tgt_test=y_test)
print('## LightGBM 예측 성능 ##')
get_model_train_eval(lgbm_clf, ftr_train=X_train, ftr_test=X_test, tgt_train=y_train, tgt_test=y_test)

```

```

## 로지스틱 회귀 예측 성능 ##
오차 행렬
[[85283   12]
 [   57   89]]
정확도: 0.9992, 정밀도: 0.8812, 재현율: 0.6096,   F1: 0.7206, AUC:0.9839
## LightGBM 예측 성능 ##
오차 행렬
[[85004   291]
 [   49   97]]
정확도: 0.9960, 정밀도: 0.2500, 재현율: 0.6644,   F1: 0.3633, AUC:0.8038

```

→ LGBM 정밀도가 왜이렇게 낮게 나오지..? ㅏ

→ LGBM 모델 객체 생성할 때 하이퍼 파라미터 지정 안해줘서인듯!

(n\_estimators=1000, num\_leaves=64, n\_jobs=-1, boost\_from\_average=False)

(파라미터 지정하고 다시 돌린 결과)

```

## 로지스틱 회귀 예측 성능 ##
오차 행렬
[[85283   12]
 [   57   89]]
정확도: 0.9992, 정밀도: 0.8812, 재현율: 0.6096,   F1: 0.7206, AUC:0.9839
## LightGBM 예측 성능 ##
오차 행렬
[[85292     3]
 [   26  120]]
정확도: 0.9997, 정밀도: 0.9756, 재현율: 0.8219,   F1: 0.8922, AUC:0.9838

```

## • SMOTE 오버 샘플링 적용 후 모델 학습/예측/평가

⚠ 반드시 학습 데이터 세트에만 오버 샘플링 적용!

```

from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state = 123)

```

```
X_train_over, y_train_over = smote.fit_sample(X_train, y_train)
print('SMOTE 적용 전 학습용 피쳐, 레이블 데이터 세트:', X_train.shape, y_train.shape)
print('SMOTE 적용 후 학습용 피쳐, 레이블 데이터 세트:', X_train_over.shape, y_train_over.shape)
print('SMOTE 적용 후 레이블 값 분포')
print(y_train_over.value_counts())
```

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

→ 2배 가까이 데이터 증식

→ 레이블 값이 0과 1의 분포가 동일하게 199,020건으로 생성

```
#로지스틱 회귀
lr_clf = LogisticRegression()
#ftr_train과 tgt_train이 SMOTE 증식된 X_train_over와 y_train_over로 변경됨
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
[[83063  2232]
 [   9   137]]
정확도: 0.9738, 정밀도: 0.0578, 재현율: 0.9384,    F1: 0.1089, AUC:0.9833
```

→ 로지스틱 회귀의 경우 재현율이 크게 증가 BUT **정밀도가 급격하게 저하**

→ 모델이 오버 샘플링으로 인해 실제 원본 데이터의 유형보다 너무 많은 Class=1 데이터를 학습하면서 실제 테스트 데이터 세트에서 예측을 지나치게 Class=1으로 적용해 정밀도가 떨어짐

⇒ 올바른 예측 모델이 생성되지 못함!

```
#LGBM
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
[[85282   13]
 [  21  125]]
정확도: 0.9996, 정밀도: 0.9058, 재현율: 0.8562,    F1: 0.8803, AUC:0.9927
```

⇒ SMOTE를 적용하면 재현율은 높아지나 정밀도는 낮아지는 것이 일반적