신용카드 사기 검출

Credit Card Fraud Detection

#컴퓨터공학전공 #2017108262 김희범

Credit Card Fraud

신용카드 사기 데이터셋은 284807 rows x 31 columns 의 구조를 가지고 있습니다.
특이사항으로는 전체에서 정상과 사기의 비율이 99.83% : 0.17% 으로 굉장히 불균형적인 구조를 하고 있습니다. 그 외에도 신용카드 거래 내역이기 때문에 데이터 셋의 31개의 컬럼에서 3개를 제외하고는 V1 ~ V28으로 이름이 숨겨져 있습니다. 우리는 사기 여부인 Class, 금액인 Amount만 정확히 무엇인지 알 수 있습니다. Class가 0은 정상, 1은 사기입니다.

이렇게 불균형한 데이터셋으로 만든 모델은 대부분 정상 레이블 쪽으로 예측하게 됩니다.
이럴 때 쓰이는 방법에는 두가지가 있습니다. 높은 비율을 차지하던 클래스의 데이터를 줄이는 언더샘플링과, 낮은 비율의 클래스의 데이터를 높이는 오버샘플링입니다.

언더 샘플링은 학습에 사용되는 전체 데이터가 감소해 성능이 떨어질 수도 있습니다.
오버 샘플링은 새로운 데이터를 어떻게 만들어내느냐가 문제입니다.

여기서는 SMOTE (Synthetic Minority Over-sampling Technique) 방법을 사용할 예정입니다. 이는 최근접 이웃 (K-Nearest Neighbor) 으로 데이터와 이웃들의 차이를 값으로 만들어 새로운 데이터를 생성하는 방식입니다.

#01

데이터셋 소개 및 전처리

```
df = pd.read_csv('../input/creditcardfraud/creditcard.csv')
df.head()
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Class
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	-0.021053	149.62	0
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	0.014724	2.69	0
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	-0.059752	378.66	0
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274	-0.190321	-1.175575	0.647376	-0.221929	0.062723	0.061458	123.50	0
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.798278	-0.137458	0.141267	-0.206010	0.502292	0.219422	0.215153	69.99	0

```
print('전체에서 정상 결제는', round(df['Class'].value_counts()[0]/len(df) * 100,2), '%')
print('전체에서 사기는', round(df['Class'].value_counts()[1]/len(df) * 100,2), '%')

df.shape
```

```
전체에서 정상 결제는 99.83 %
전체에서 사기는 0.17 %
[6]: (284807, 31)
```

```
from sklearn.model_selection import train_test_split

# 필요없는 Time 칼럼 삭제 및 전처리
def preprocessed_df(df=None):
    df_copy = df.copy()
    df_copy.drop('Time', axis=1, inplace=True)
    X_features = df_copy.iloc[:, :-1] # 맨 마지막 열만 선택
    y_target = df_copy.iloc[:, -1] # 맨 마지막 열만 선택
    X_train, X_test, y_train, y_test = train_test_split(X_features, y_target, test_size=0.2, random_state=0, stratify=y_target)

    return X_train, X_test, y_train, y_test

X_train, X_test, y_train, y_test = preprocessed_df(df)
```

+ Code + Markdown

```
print('학습 데이터 레이블 값 비율')
print(y_train.value_counts()/y_train.shape[0] * 100)
print('테스트 데이터 레이블 값 비율')
print(y_test.value_counts()/y_test.shape[0] * 100)
```

```
학습 데이터 레이블 값 비율
0    99.827075
1     0.172925
Name: Class, dtype: float64
테스트 데이터 레이블 값 비율
0    99.827955
1     0.172045
Name: Class, dtype: float64
```

#02

모델 학습/예측/평가

LogisticRegression, LightGBM 모델을 이용하여 진행합니다.

정확도: 예측이 정답과 얼마나 정확한지
정밀도: 예측 중 정답의 비율
재현율:찾아야 할 것 중 실제로 찾은 비율
F1 스코어: 정밀도와 재현율의 평균
AUC: ROC 곡선 아래의 넓이

F1 스코어와 AUC가 높으면 높을수록 성능이 높다고 할 수 있습니다.

```
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score, 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 = roc_auc_score(y_test, pred)

    print('오차행렬')
    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))
```

오차행렬이란

True Positive, False Negative

Flase Positive, True Negative

예측과 실제 결과가 부합한지를 알려주는 행렬입니다.

```
from sklearn.linear_model import LogisticRegression

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

get_clf_eval(y_test, lr_pred, lr_pred_proba)
```

```
오차행렬
[[56853    11]
 [   36    62]]
정확도: 0.9992, 정밀도: 0.8493, 재현율: 0.6327, F1: 0.7251, AUC:0.8162
```

Logistic Regression 결과로

정확도: 0.9992

정밀도: 0.8493

재현율: 0.6327

F1 스코어 : 0.7251

AUC : 0.8162

```
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)
```

계속 재사용하기 위해 함수로 만들어줍니다.

```
# 불균형한 레이블 값 분포도를 가지므로 LGBMClassifier에서 boost_from_average=False로 설정해야한다.
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)
```

```
오차행렬
[[56860     4]
 [   24    74]]
정확도: 0.9995, 정밀도: 0.9487, 재현율: 0.7551, F1: 0.8409, AUC:0.8775
```

LightGBM 결과로
정확도: 0.9995
정밀도: 0.9487
재현율: 0.7551
F1 스코어 : 0.8409
AUC : 0.8775

LightGBM이 LogisticRegression보다 더 높은 성능을 보입니다.
F1 Score: 0.8409 vs 0.7251
AUC: 0.8162 vs 0.8775

신용카드 사기 검출

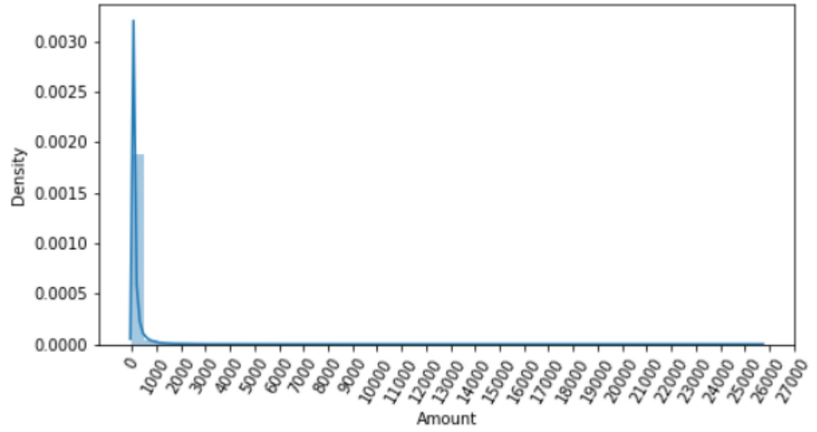
```python
from sklearn.preprocessing import StandardScaler
# 사이킷런의 StandardScaler를 이용해 정규 분포 형태로 Amount 피처 값 변환
def preprocessed_df(df=None):
    df_copy = df.copy()
    scaler = StandardScaler()
    amount_n = scaler.fit_transform(df_copy['Amount'].values.reshape(-1, 1))
    df_copy.insert(0, 'Amount_Scaled', amount_n)
    # 기존 Time, Amount 피처 삭제
    df_copy.drop(['Time', 'Amount'], axis=1, inplace=True)
    X_features = df_copy.iloc[:, :-1] # 맨 마지막 열만 선택
    y_target = df_copy.iloc[:, -1] # 맨 마지막 열만 선택
    X_train, X_test, y_train, y_test = train_test_split(X_features, y_target, test_size=0.2, random_state=0, stratify=y

    return X_train, X_test, y_train, y_test
```

+ Code + Markdown

```python
X_train, X_test, y_train, y_test = preprocessed_df(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('\n### LightGBM 예측 성능 ###')
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)
```

```
### 로지스틱 회귀 예측 성능 ###
오차행렬
[[56853    11]
 [   36    62]]
정확도: 0.9992, 정밀도: 0.8493, 재현율: 0.6327, F1: 0.7251, AUC:0.8162

### LightGBM 예측 성능 ###
오차행렬
[[56860     4]
 [   24    74]]
정확도: 0.9995, 정밀도: 0.9487, 재현율: 0.7551, F1: 0.8409, AUC:0.8775
```

0 ~ 1000불 이하의 데이터가 대부분이기에 정규분포로 변환



Logistic Regression 결과로	LightGBM 결과로
정확도: 0.9992	정확도: 0.9995
정밀도: 0.8493	정밀도: 0.9487
재현율: 0.6327	재현율: 0.7551
F1 스코어 : 0.7251	F1 스코어 : 0.8409
AUC : 0.8162	AUC : 0.8775

차이가 있을 것 같았는데, 차이가 발생하지 않았습니다.

```
def preprocessed_df_log(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)
    X_features = df_copy.iloc[:, :-1]
    y_target = df_copy.iloc[:, -1]
    # stratify = y_target 으로 stratified 기반 분할
    X_train, X_test, y_train, y_test = train_test_split(X_features, y_target, test_size=0.2, random_state=0, stratify=y

    return X_train, X_test, y_train, y_test
```

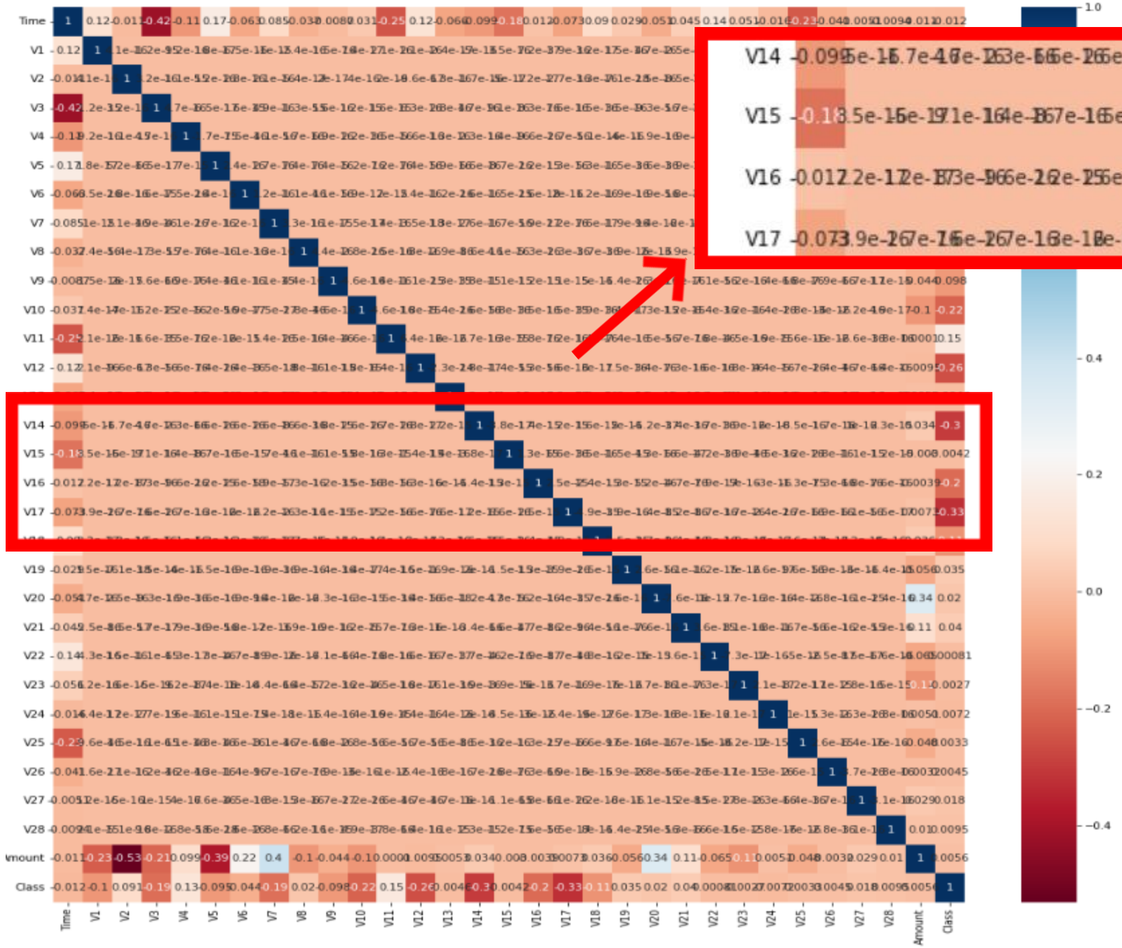
넘파이의 log1p 함수를 이용해서 로그 변환 후
모델을 적용했습니다.

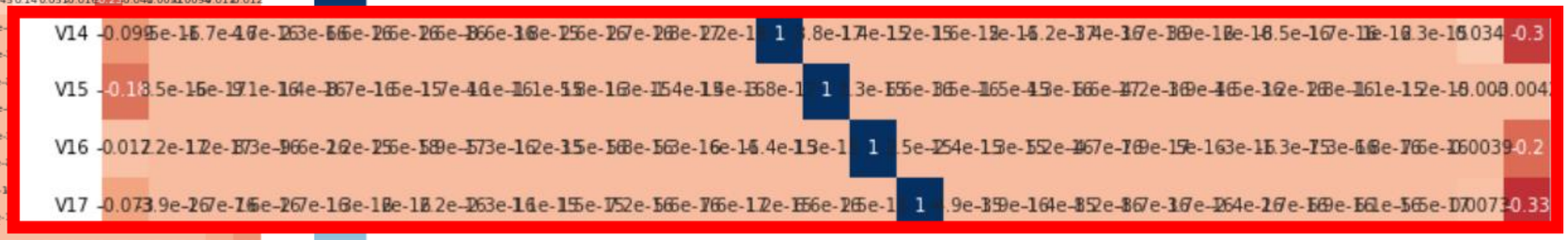
```
X_train, X_test, y_train, y_test = preprocessed_df_log(df)

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('\n### 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 ###
오차행렬
[[56854    10]
 [   36    62]]
정확도: 0.9992, 정밀도: 0.8611, 재현율: 0.6327, F1: 0.7294, AUC:0.8162

### LightGBM 예측 성능 ###
오차행렬
[[56860     4]
 [   23    75]]
정확도: 0.9995, 정밀도: 0.9494, 재현율: 0.7653, F1: 0.8475, AUC:0.8826
```

Logistic Regression 결과로	LightGBM 결과로
정확도: 0.9992	정확도: 0.9995
정밀도: 0.8611	정밀도: 0.9494
재현율: 0.6327	재현율: 0.7653
F1 스코어 : 0.7294	F1 스코어 : 0.8475
AUC : 0.8162	AUC : 0.8826

로지스틱 회귀 같은 경우는 정밀도와 F1 스코어에서 약간의 변화 발생
LightGBM은 정밀도, 재현율을 포함한 F1, AUC 모두 성능 소폭 상승을 보입니다.





클래스와 음의 상관관계가 가장 높은 두 피쳐는 V14와 V17
그 중에서 V14에 대해서 이상치를 찾아 제거해봤습니다.
이상치 인덱스 8296, 8615, 9035, 9252을 추출하고 삭제하는 로직을 생성 후 모델을 돌려보았습니다.

```
def get_outlier(df=None, column=None, weight=1.5):
    fraud = df[df['Class'] == 1][column]
    q25 = np.percentile(fraud.values, 25)
    q75 = np.percentile(fraud.values, 75)
    iqr = q75 - q25
    iqr_weight = iqr * weight
    low_val = q25 - iqr_weight
    high_val = q75 + iqr_weight
    outlier_index = fraud[(fraud < low_val) | (fraud > high_val)].index
    return outlier_index
```

+ Code + Markdown

```
outlier_index = get_outlier(df=df, column='V14', weight=1.5)
print(f'이상치 데이터 인덱스: {outlier_index}')
```

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

신용카드 사기 검출

```python
def preprocessed_df_ext(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)
    outlier_index = get_outlier(df=df_copy, column='V14', weight=1.5)
    df_copy.drop(outlier_index, axis=0, inplace=True)
    X_features = df_copy.iloc[:, :-1]
    y_target = df_copy.iloc[:, -1]

    X_train, X_test, y_train, y_test = train_test_split(X_features, y_target, test_size=0.2, random_state=0, stratify=y_target)

    return X_train, X_test, y_train, y_test

X_train, X_test, y_train, y_test = preprocessed_df_ext(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('\n### 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 ###
오차행렬
[[56853    10]
 [   34    64]]
정확도: 0.9992, 정밀도: 0.8649, 재현율: 0.6531, F1: 0.7442, AUC:0.8264

### LightGBM 예측 성능 ###
오차행렬
[[56859     4]
 [   19    79]]
정확도: 0.9996, 정밀도: 0.9518, 재현율: 0.8061, F1: 0.8729, AUC:0.9030
```

Logistic Regression 결과로
정확도: 0.9992
정밀도: 0.8649
재현율: 0.6531
F1 스코어 : 0.7449
AUC : 0.8264

LightGBM 결과로
정확도: 0.9996
정밀도: 0.9518
재현율: 0.8061
F1 스코어 : 0.8729
AUC : 0.9030

두 모델 모두 전반적으로 크게 향상되었다고 할 수 있습니다.

SMOTE 적용 후 LogisticRegression 모델 적용

```
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=0)
X_train_over, y_train_over = smote.fit_resample(X_train, y_train)
print(f'SMOTE 적용 전 학습용 피처/레이블 데이터 세트: {X_train.shape, y_train.shape}')
print(f'SMOTE 적용 후 학습용 피처/레이블 데이터 세트: {X_train_over.shape, y_train_over.shape}')
print(f'SMOTE 적용 후 레이블 값 분포: \n{pd.Series(y_train_over).value_counts()}')
```

```
SMOTE 적용 전 학습용 피처/레이블 데이터 세트: ((227842, 29), (227842,))
SMOTE 적용 후 학습용 피처/레이블 데이터 세트: ((454904, 29), (454904,))
SMOTE 적용 후 레이블 값 분포:
0    227452
1    227452
Name: Class, dtype: int64
```

```
lr_clf = LogisticRegression()
# ftr_train 과 tgt_train 인자 값이 SMOTE 증식된 X_train_over 와 y_train_over로 변경
get_model_train_eval(lr_clf, ftr_train=X_train_over, ftr_test=X_test, tgt_train=y_train_over,
```

```
오차행렬
[[55324  1539]
 [   10    88]]
정확도: 0.9728, 정밀도: 0.0541, 재현율: 0.8980, F1: 0.1020, AUC:0.9354
```

맨 초기 Logistic Regression 결과는

정확도: 0.9992

정밀도: 0.8493

재현율: 0.6327

F1 스코어 : 0.7251

AUC : 0.8162

재현율은 0.8980으로 향상되었지만,

정밀도가 0.0541로 말도 안 되게 하락했습니다.

SMOTE 적용 후 LightGBM 모델 적용

```
lgbm_clf = LGBMClassifier(n_estimators=1000, num_leaves=64, n_jobs=-1, boost_from_average=Fal
get_model_train_eval(lgbm_clf, ftr_train=X_train_over, ftr_test=X_test, tgt_train=y_train_ove
```

```
오차행렬
[[56856     7]
 [   17    81]]
정확도: 0.9996, 정밀도: 0.9205, 재현율: 0.8265, F1: 0.8710, AUC:0.9132
```

맨 초기 LightGBM 결과는
정확도: 0.9995
정밀도: 0.9487
재현율: 0.7551
F1 스코어 : 0.8409
AUC : 0.8775

재현율은 0.8265로 향상되었지만,
정밀도가 0.9205로 하락했습니다.

SMOTE 방법을 쓰면 재현율은 높아지지만, 정밀도가 하락하게 됩니다.
이는 초기 결과와 비교하면 높은 성능 향상을 보여주었지만, 이상치만을 제거한 경우
와는 성능면에서 큰 차이가 발생하지 않았습니다.

THANK YOU ®

Q&A

#컴퓨터공학전공 #2017108262 김희범

인공지능
변영철 교수님