신용카드 사기 탐지 하기

2020 산업공학 데이터 분석 대회

홍혜림

홍익대학교 대학혁신지원사업

00
목차

00
목차

#0 주제선정 배경

신용카드 결제 사기로 인한 피해 증가

- 최근 3년간 9개 카드사의 FDS가 차단한 부정 사용 시도는 약 100만건
- 더불어 정상 거래임에도 사기로 오인하여 정지 당하는 고객의 피해도 포함

- 따라서, 금융기관에서는 빅데이터, AI 등 신기술을 적극 활용하여

정확한 신용카드 부정사용 차단을 위해 힘쓰고 있음

HOME > News > 금융

'신용카드결제 사기'.. 최근 3년간 100만건, 1700억 규모

유호영 기자 | 승인 2020.07.23 17:31 | 댓글 0

카드사 FDS 통해 부정사용 사전 차단...100% 막기는 불가능
피해액 카드사 책임 지지만, 소비자도 경각심 가져야

1. 데이터 설명

- 캐글 제공 : 2013년 9월 유럽 카드 소지자의 2일 동안의 신용카드 거래 데이터
- 설명 변수
  → 개인 정보 보호로 인해 PCA로 이미 변환된 변수(V1~V28)
  → 시간(Time)과 거래 금액(Amount)
- 목표 변수 : Class(0은 정상, 1은 사기), 이진 변수

변수명	변수 설명	특징
V1~V28	개인 정보 보호로 공개 되지 않음	PCA 변환된 설명변수
Time	각 거래와 첫 거래 사이의 경과된 시간 (초)	PCA 변환X 설명 변수
Amount	거래 금액	
Class	응답 변수로 0이 정상 거래, 1이 사기	목표 변수

출처 : 캐글

2. 데이터 분석 목표

→ 신용카드 거래 데이터를 바탕으로 거래가 **정상적인 결제인지, 사기인지 분류**하는 모델 수립

3. 분석 이슈

1) 잘못된 사기 탐지의 위험성

- 정상 거래를 사기 탐지로 인식하여 해당 카드 정지를 내릴 경우 사용자는 큰 불만을 가질 것
- 이를 방지하기 위해 **정상 거래를 사기로 예측하는 경우(FP)를 최소화** 하는 것을 추구

Confusion Matrix		실제 정답	
		True(사기)	False(정상)
분류 결과	True(사기)	True Positive	False Positive
	False(정상)	False Negative	True Negative

최소화

3. 분석 이슈

2) 데이터의 불균형 문제

- 데이터에서 **사기의 비율은 단 0.172%로 매우 불균형**
- 그러므로 정확도는 **편중(bias)** 문제가 존재함
- 정상인 거래를 예측하는 성능은 높지만, 반대로 사기를 예측하는 성능은 매우 낮음

- 따라서, 이를 보완하기 위해 precision과 recall의 조화평균인 **F1-score 지표 사용**
- F1-score는 Label이 **불균형 구조**일 때, **모델의 성능을 정확하게 평가**할 수 있는 장점이 있음.

즉, 해당 문제에서 모델의 성능을 판단하는 두 가지 중요한 요소: 낮은 FP, 높은 F1 – score

$$(F1\text{-}score) = 2\times\frac{1}{\frac{1}{Precision}+\frac{1}{Recall}} = 2\times\frac{Precision \times Recall}{Precision + Recall}$$

4. 데이터 탐색

- 행과 열 개수: [284807, 31]
- 모두 수치형 변수

```
In [148]: df.shape
Out[148]: (284807, 31)
```

```
df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 284807 entries, 0 to 284806
Data columns (total 31 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   Time    284807 non-null  float64
 1   V1      284807 non-null  float64
 2   V2      284807 non-null  float64
 3   V3      284807 non-null  float64
```

5. 결측치 처리

- 결측치 없음

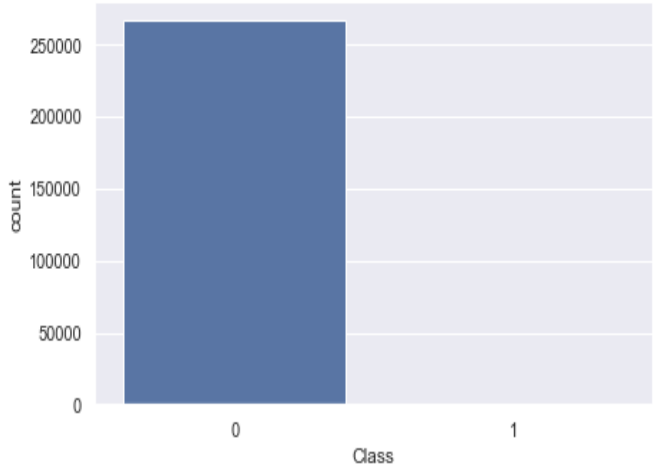
```
#결측치 확인
df.isna().sum().max()

i]: 0
```

```
#결측치 확인
df.isna().sum()

Time      0
V1        0
V2        0
V3        0
V4        0
V5        0
```

6. 목표변수 확인

- Class : 0 (정상 거래), 1 (사기)
- 목표 변수 분포
- 정상 거래 → 266937건
- 사기 → 492건
- 매우 불균형함

```
# 목표 변수의 분포 확인
df['Class'].value_counts()

0    266937
1       492
Name: Class, dtype: int64
```

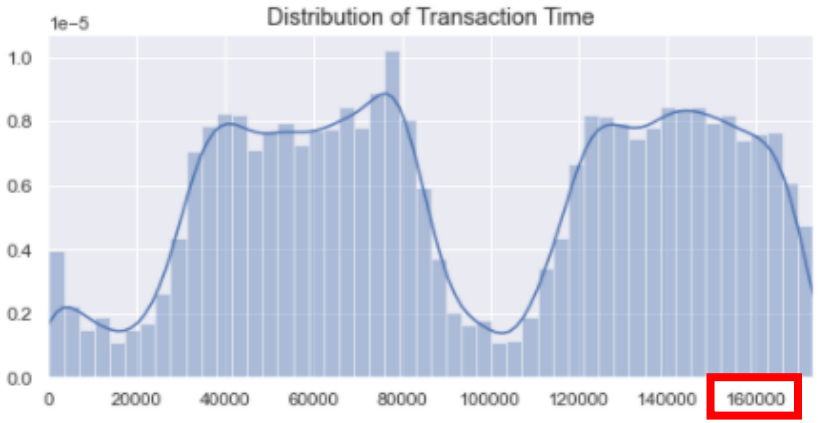


7. 정규화

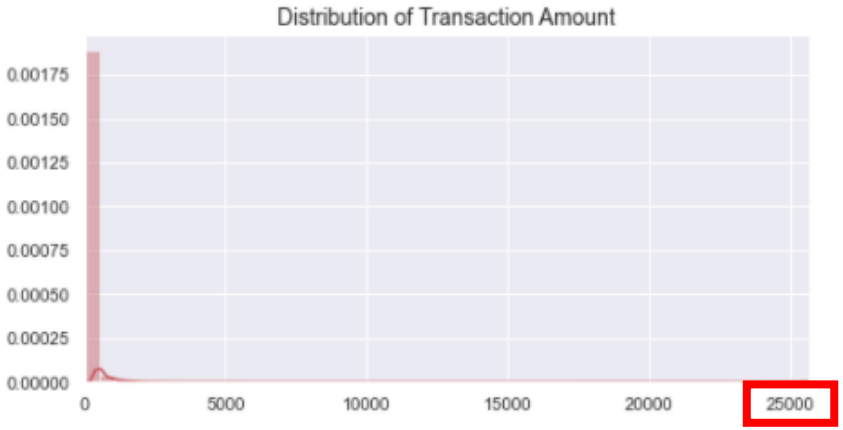
1) 정규화 전

- PCA 변환되지 않은 변수인 Time와 Amount에 대한 정규화
- Time은 0~ 160000이 넘는 값까지 분포
- Amount는 0에 굉장히 밀집했고, 최대 25000 값까지 분포

<Time의 분포>



<Amount의 분포>



2) 정규화 방법

❖ StandardScaler: 기본 스케일, 평균과 표준편차 사용

❖ RobustScaler: 중앙값과 IQR 사용, 이상치의 영향을 최소화

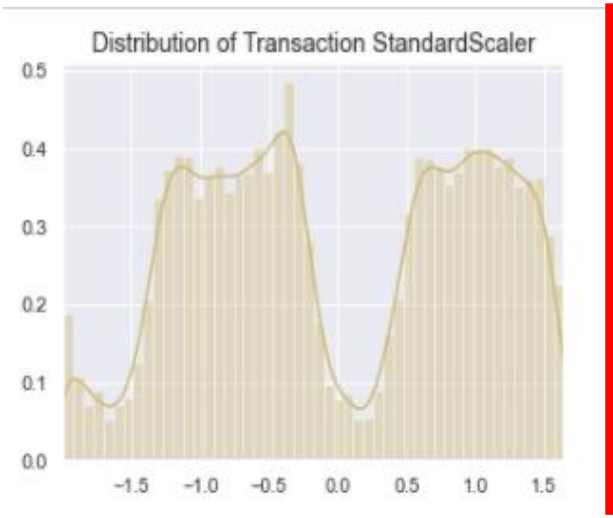
→ 두 가지 방법을 사용하여 Time과 Amount를 정규화

→ 더 작은 분포를 보인 것을 채택

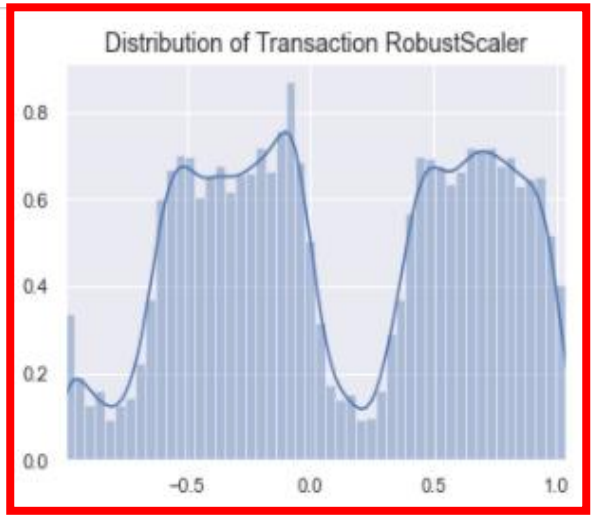
1) Time 정규화

→ 더 작은 분포 범위를 보이는 RobustScaler 채택

<StandardScaler 후 Time의 분포>



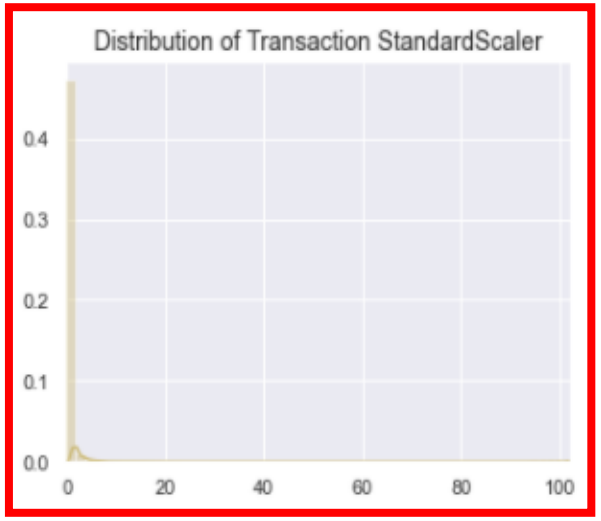
<RobustScaler 후 Time의 분포>



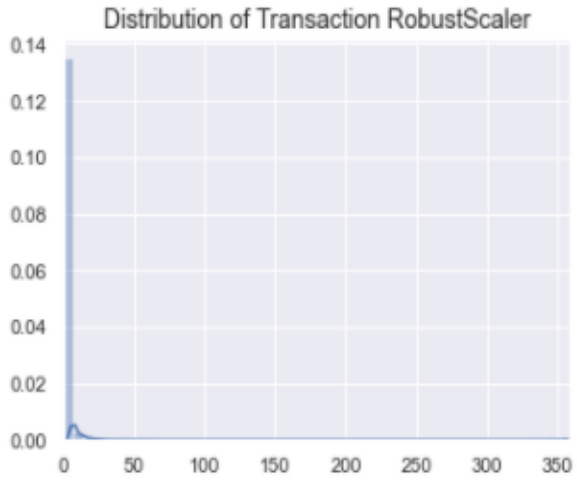
2) Amount 정규화

→ 더 작은 분포 범위를 보이는 StandardScaler 채택

<Standard Scaler 후 Amount의 분포>



<RobustScaler후 Amount의 분포>



8. 이상치 처리

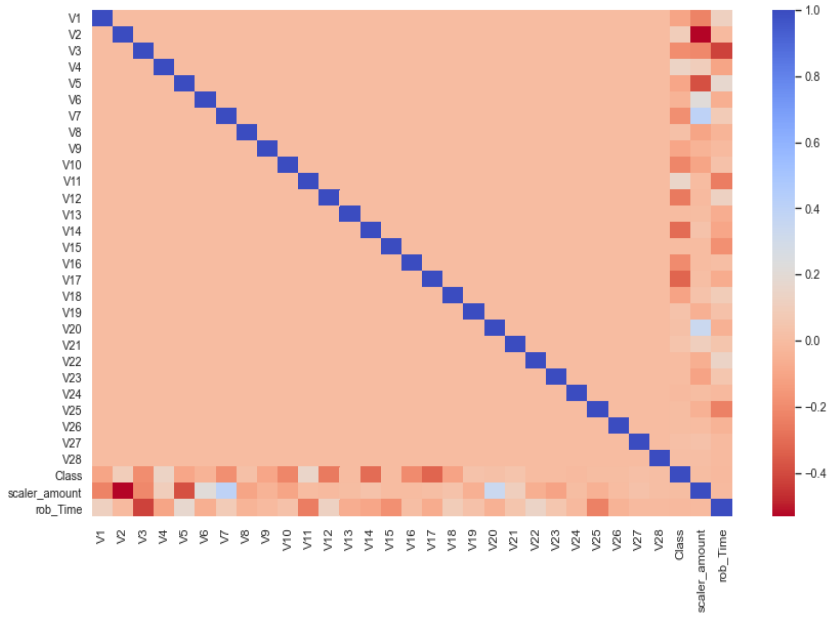
→ 모든 데이터에서 IQR을 통해 이상치 제거할 시
많은 데이터의 손실이 발생

✓ 해결 방법

- 목표 변수에 영향을 많이 미치는 변수를 상관 분석을 통해 찾고, 해당 변수에서만 이상치 제거
- Class(목표 변수)가 1(사기)인 데이터는 부족하기 때문에 Class가 0(정상)인 데이터의 이상치 제거
- 상관 분석 결과

-> 목표 변수와 연관성이 높은 변수는 V17, V14

<상관관계 그래프>

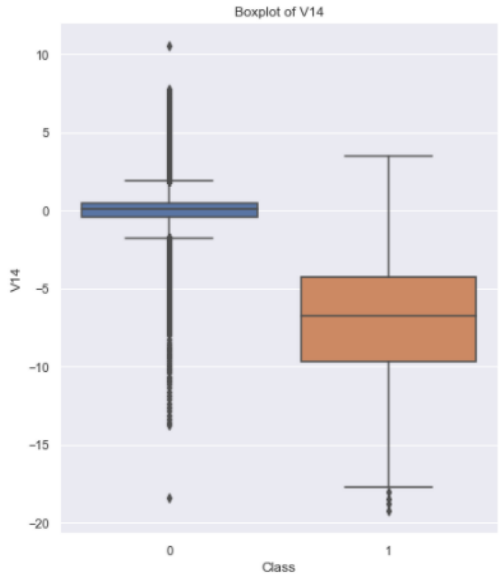


<상관관계 수치화>

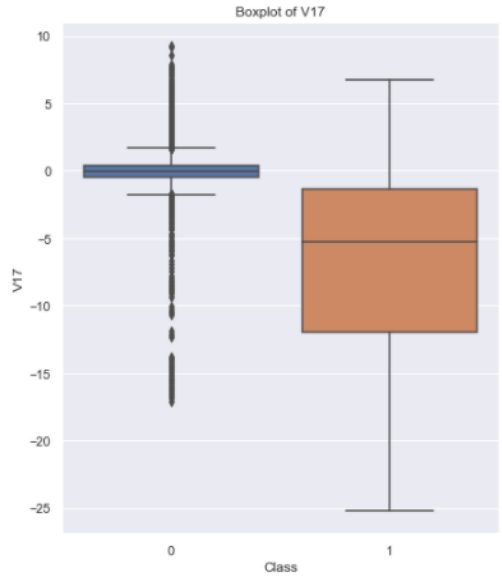
	Class
V17	-0.326481
V14	-0.302544
V12	-0.260593
V10	-0.216883
V16	-0.196539

#3 EDA

<목표 변수와 연관성이 높은 V14, V17의 Boxplot>

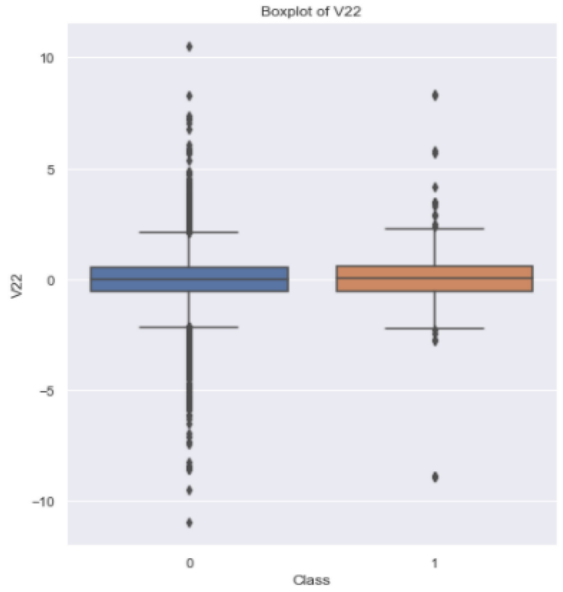


0 class 1

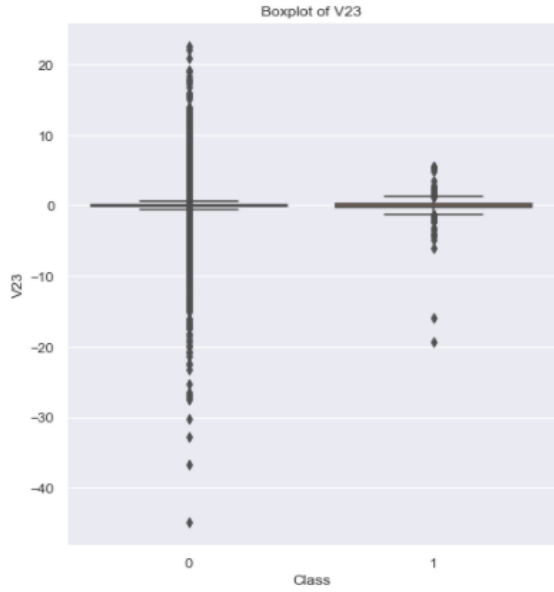


0 class 1

<목표 변수와 연관성이 낮은 V22, V23의 Boxplot>



0 class 1



0 class 1

→ Boxplot의 비교를 통하여 V14와 V17은 목표 변수가 0, 1일 때 서로 분포가 확연히 다름을 확인

→ V14와 V17에 대하여 자세히 알 수는 없지만 정상 거래와 사기를 구분하는 중요한 Column임을 재차 확인

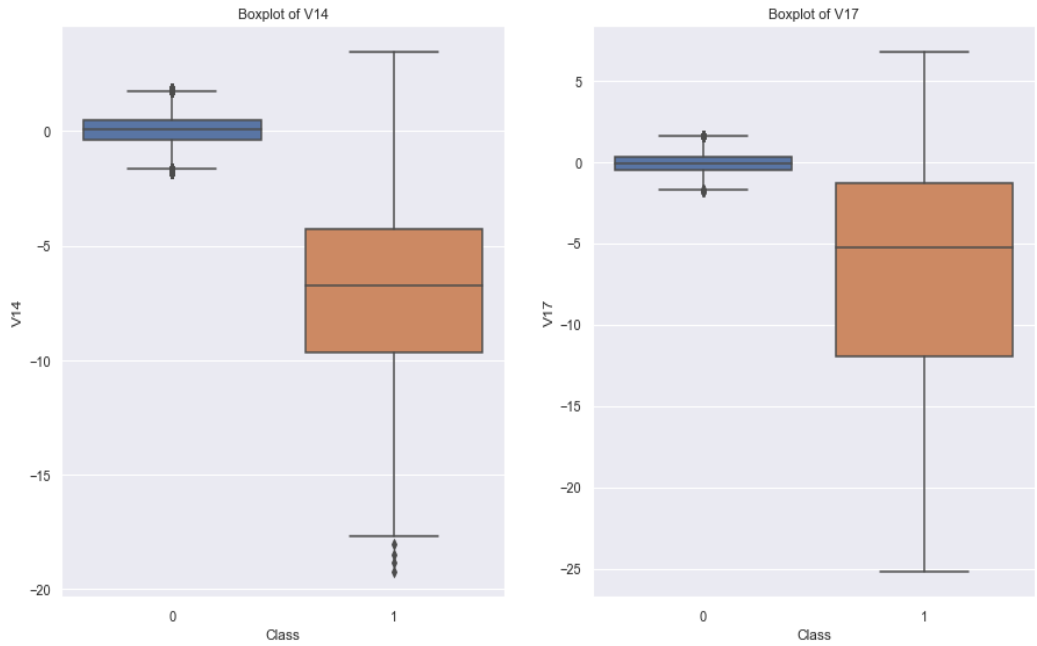
- V14, V17의 이상치 제거

```
#14, 17의 이상치 제거
def remove_outlier(d_cp, column):dusd
    fraud_column_data = d_cp[d_cp['Class']==0][column]
    quan_25 = np.percentile(fraud_column_data.values,25)
    quan_75 = np.percentile(fraud_column_data.values,75)

    iqr = quan_75-quan_25
    iqr = iqr*1.5
    lowest = quan_25 - iqr
    highest = quan_75 + iqr
    outlier_index = fraud_column_data[(fraud_column_data < lowest) | (fraud_column_data > highest)].index
    print(len(outlier_index))
    d_cp.drop(outlier_index, axis = 0, inplace = True)
    print(d_cp.shape)
    return d_cp
```

```
df = remove_outlier(df, 'V14')
df = remove_outlier(df, 'V17')
```

```
13800
(271007, 31)
3578
(267429, 31)
```

<이상치가 제거된 V14와 V17의 Boxplot>



→ 결과적으로 IQR을 통해 Class가 0 일 때의 V14, V17의 이상치 제거

→ 17378개의 이상치 제거

00
목차

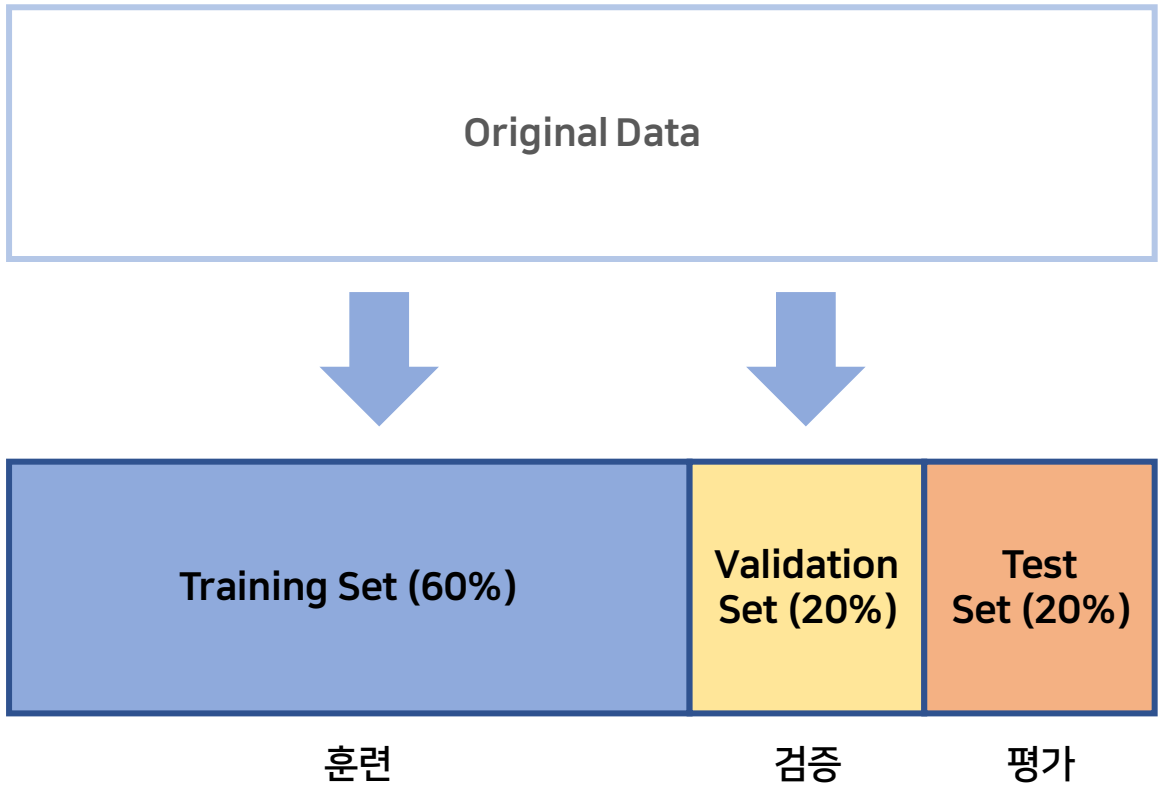
1. 과적합 방지

→ 데이터를 Training (60%), Validation (20%), Test (20%) 3개의 set으로 분할

1) Training set을 가지고 분류 모델을 학습

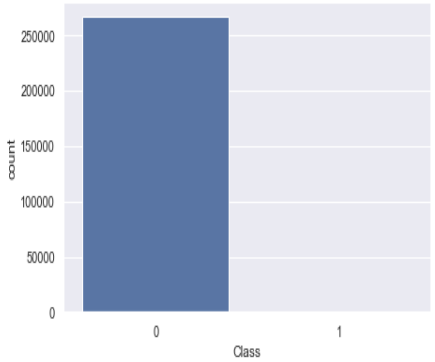
2) Validation set을 통해 훈련중인 모델이 과적합 또는 과소적합 문제에 직면했는지 검증하며 모델링 구축

3) Test set을 사용하여 최종 모델에 대해 평가

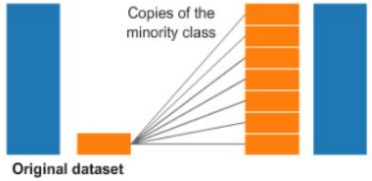


2. 모델링 방법

앞서 말했듯이 가장 큰 문제점은 목표 변수의 "데이터 불균형"



Under Sampling



Over Sampling

로지스틱 회귀
랜덤포레스트
XGBOOST
LGBM
신경망

로지스틱 회귀
랜덤포레스트
XGBOOST
LGBM
신경망

✓ 해결 방법

1) 언더 샘플링: 불균형한 데이터 셋에서 높은 비율을 차지하던 정상 거래 데이터를 줄임 → RandomUnderSampler 사용
2) 오버 샘플링: 불균형한 데이터 셋에서 낮은 비율을 차지하던 사기 데이터를 늘림 → SMOTE 사용

두 샘플링 방법을 통하여 데이터 불균형 문제를 해결하고, 이후 5개의 모델을 각각 사용하여 분류를 진행

2. 모델링 방법

1) 언더 샘플링: 불균형한 데이터 셋에서 높은 비율을 차지하던 정상 거래 데이터를 줄임 → RandomUnderSampler 사용


```
#X_samp, y_samp = RandomUnderSampler(random_state=0).fit_sample(X_imb, y_imb)
rus = RandomUnderSampler(random_state=0)
X_train_under,y_train_under = rus.fit_sample(X_train,y_train)
print('RandomUnder 적용 전 학습용 피처/레이블 데이터 세트: ', X_train.shape, y_train.shape)
print('RandomUnder 적용 후 학습용 피처/레이블 데이터 세트: ', X_train_under.shape, y_train_under.shape)
print('RandomUnder 적용 후 레이블 값 분포:₩n', pd.Series(y_train_under).value_counts())
```

```
RandomUnder 적용 전 학습용 피처/레이블 데이터 세트:  (213943, 30) (213943,)
RandomUnder 적용 후 학습용 피처/레이블 데이터 세트:  (774, 30) (774,)
RandomUnder 적용 후 레이블 값 분포:
 1    387
0    387
Name: Class, dtype: int64
```

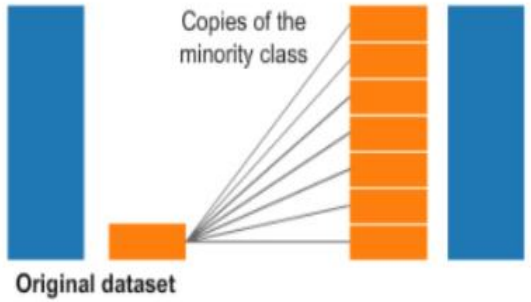
```
sns.countplot(x='Class',data=df_under)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x2136e508148>
```

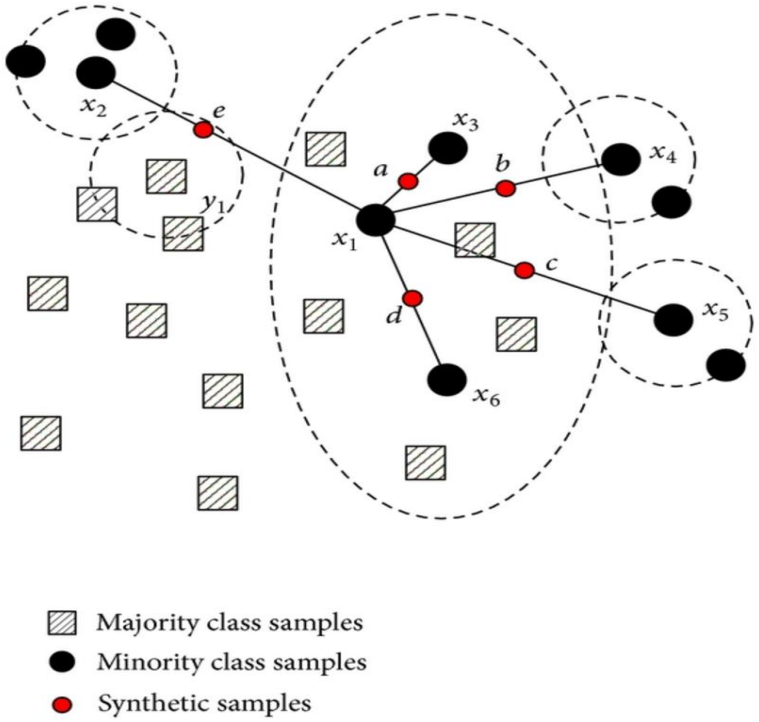
- 무작위로 데이터를 없애는 단순 샘플링 방식
- 잠재적으로 가치가 큰 데이터가 사라질 위험이 있어 정보의 손실이 큼

2. 모델링 방법

2) 오버 샘플링: 불균형한 데이터 셋에서 낮은 비율을 차지하던 사기 데이터를 늘림 → SMOTE 사용



<SMOTE 동작 방식>



- 단순한 오버샘플링 방식은 동일한 데이터의 복제로 개수만 늘리며, 오버피팅 존재
- SMOTE 알고리즘 : 합성 소수 샘플링 기술, 기존 소수 데이터를 보간하여 새로운 소수 인스턴스를 합성
- 부트스트래핑이나 KNN(최근접이웃) 모델 기법을 활용

2. 모델링 방법

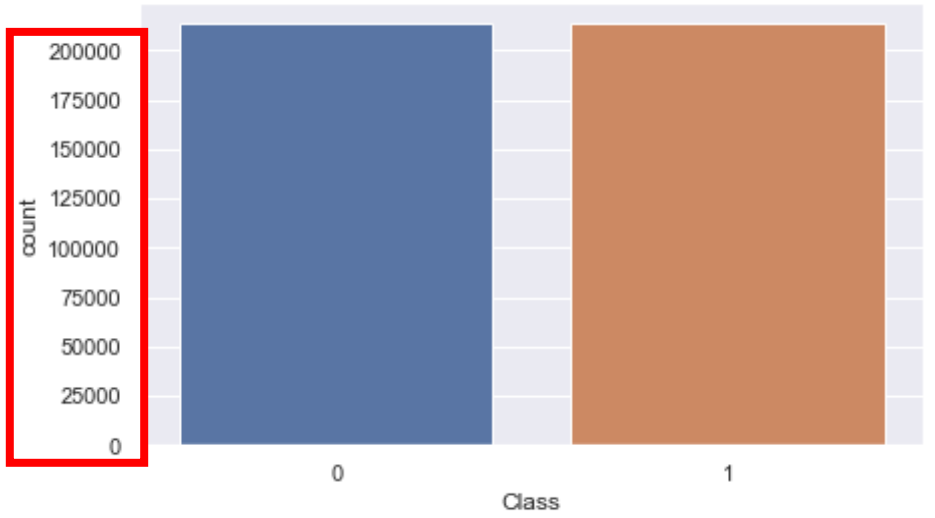
2) 오버 샘플링: 불균형한 데이터 셋에서 낮은 비율을 차지하던 사기 데이터를 늘림 → SMOTE 사용

```
from imblearn.over_sampling import SMOTE
smote = SMOTE(random_state=0)
X_train_over,y_train_over = smote.fit_sample(X_train,y_train)
print('SMOTE 적용 전 학습용 피처/레이블 데이터 세트: ', X_train.shape, y_train.shape)
print('SMOTE 적용 후 학습용 피처/레이블 데이터 세트: ', X_train_over.shape, y_train_over.shape)
print('SMOTE 적용 후 레이블 값 분포: ₩n', pd.Series(y_train_over).value_counts())
```

```
SMOTE 적용 전 학습용 피처/레이블 데이터 세트:  (213943, 30) (213943,)
SMOTE 적용 후 학습용 피처/레이블 데이터 세트:  (427094, 30) (427094,)
SMOTE 적용 후 레이블 값 분포:
 1    213547
0    213547
Name: Class, dtype: int64
```

```
sns.countplot(x='Class',data=df_over)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x2879711cf08>
```



#2 모델링 방법

3. 머신 러닝 분류 모델

```python
#모델링 함수
def modeling(model,X_train,X_val,y_train,y_val):
    model.fit(X_train,y_train)
    pred = model.predict(X_val)
    metrics(y_val,pred)
```

```python
#정확도, 정밀도, 재현율, f1-score, auroc 확인
def metrics(y_val,pred):
    accuracy = accuracy_score(y_val,pred)
    precision = precision_score(y_val,pred)
    recall = recall_score(y_val,pred)
    f1 = f1_score(y_val,pred)
    roc_score = roc_auc_score(y_val,pred,average='macro')
    print('정확도 : {0:.2f}, 정밀도 : {1:.2f}, 재현율 : {2:.2f}'.form
    print('f1-score : {0:.2f}, auc : {1:.3f}'.format(f1,roc_score)
```

→ 언더 샘플링과 오버 샘플링 동일하게 모델링 진행

```python
#로지스틱 회귀
lr = LogisticRegression(random_state =1,C=1000)
modeling(lr,X_train,X_test,y_train,y_test)
```

```
정확도 : 0.98, 정밀도 : 0.07, 재현율 : 0.94
f1-score : 0.12, auc : 0.957
```

```python
#랜덤포레스트
rfc = RandomForestClassifier(n_estimators=200,random_state=30, max_depth=8,n_jobs=-1)
modeling(rfc,X_train,X_test,y_train,y_test)
```

```
정확도 : 1.00, 정밀도 : 0.90, 재현율 : 0.91
f1-score : 0.90, auc : 0.953
```

```python
#xgb
xgb = XGBClassifier()
modeling(xgb,X_train,X_test,y_train,y_test)
```

```
정확도 : 1.00, 정밀도 : 0.86, 재현율 : 0.90
f1-score : 0.88, auc : 0.948
```

```python
#lgbm
lgb = LGBMClassifier(n_estimators=1000,num_leaves=64,n_jobs=-1,boost_from_average=False)
modeling(lgb,X_train,X_test,y_train,y_test)
```

```
정확도 : 1.00, 정밀도 : 0.97, 재현율 : 0.89
f1-score : 0.92, auc : 0.943
```

#2 모델링 방법

4. 신경망 모델

```
model = Sequential()
model.add(Dense(12, input_shape=(30, ), activation='relu'))
model.add(Dense(12, activation='relu'))
model.add(Dropout(0.3))

model.add(Dense(1, activation='sigmoid'))
optimizer = Adam()

ck = ModelCheckpoint('rnn_ksic.h5', monitor='val_loss', verbose=1, save_best_only=True)
es = EarlyStopping(monitor='val_loss', patience=3)

model.compile(optimizer, loss='binary_crossentropy', metrics=['accuracy'])
```

```
history=model.fit(X_train,y_train,epochs=20,batch_size=5,callbacks = [ck, es],validation_data=(X_val,y_val))
```

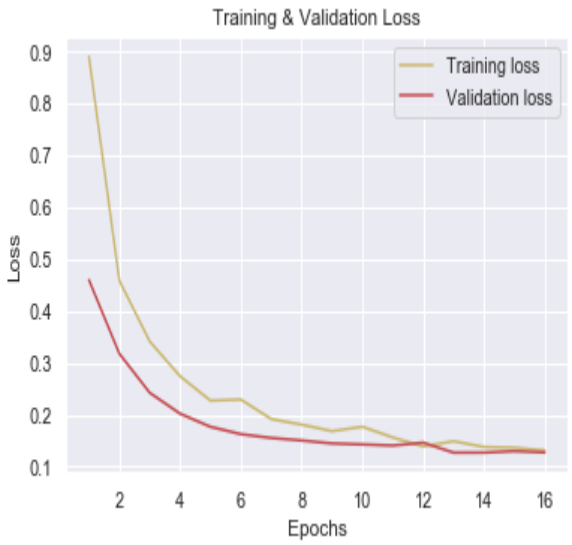
```
Epoch 1/20
116/124 [===========================>..] - ETA: 0s - loss: 0.9123 - accuracy: 0.6224
Epoch 00001: val_loss improved from inf to 0.45982, saving model to rnn_ksic.h5
124/124 [==============================] - 1s 4ms/step - loss: 0.8901 - accuracy: 0.6333 - val_loss: 0.4598 - val_accuracy: 0.8774
Epoch 2/20
 91/124 [=====================>........] - ETA: 0s - loss: 0.4755 - accuracy: 0.8044
Epoch 00002: val_loss improved from 0.45982 to 0.31794, saving model to rnn_ksic.h5
124/124 [==============================] - 0s 2ms/step - loss: 0.4587 - accuracy: 0.8110 - val_loss: 0.3179 - val_accuracy: 0.9226
Epoch 3/20
117/124 [===========================>..] - ETA: 0s - loss: 0.3445 - accuracy: 0.8718
Epoch 00003: val_loss improved from 0.31794 to 0.24251, saving model to rnn_ksic.h5
124/124 [==============================] - 0s 3ms/step - loss: 0.3415 - accuracy: 0.8772 - val_loss: 0.2425 - val_accuracy: 0.9419
Epoch 4/20
119/124 [===========================>..] - ETA: 0s - loss: 0.2759 - accuracy: 0.8958
Epoch 00004: val_loss improved from 0.24251 to 0.20222, saving model to rnn_ksic.h5
124/124 [==============================] - 0s 3ms/step - loss: 0.2743 - accuracy: 0.8950 - val_loss: 0.2022 - val_accuracy: 0.9548
Epoch 5/20
121/124 [============================>.] - ETA: 0s - loss: 0.2281 - accuracy: 0.9306
Epoch 00005: val_loss improved from 0.20222 to 0.17689, saving model to rnn_ksic.h5
124/124 [==============================] - 0s 3ms/step - loss: 0.2274 - accuracy: 0.9321 - val_loss: 0.1769 - val_accuracy: 0.9484
Epoch 6/20
100/124 [=======================>......] - ETA: 0s - loss: 0.2238 - accuracy: 0.9180
Epoch 00006: val_loss improved from 0.17689 to 0.16276, saving model to rnn_ksic.h5
124/124 [==============================] - 0s 3ms/step - loss: 0.2294 - accuracy: 0.9176 - val_loss: 0.1628 - val_accuracy: 0.9484
Epoch 7/20
116/124 [===========================>..] - ETA: 0s - loss: 0.1967 - accuracy: 0.9276
Epoch 00007: val_loss improved from 0.16276 to 0.15538, saving model to rnn_ksic.h5
124/124 [==============================] - 0s 2ms/step - loss: 0.1918 - accuracy: 0.9289 - val_loss: 0.1554 - val_accuracy: 0.9484
Epoch 8/20
117/124 [===========================>..] - ETA: 0s - loss: 0.1805 - accuracy: 0.9282
Epoch 00008: val_loss improved from 0.15538 to 0.15058, saving model to rnn_ksic.h5
124/124 [==============================] - 0s 3ms/step - loss: 0.1811 - accuracy: 0.9305 - val_loss: 0.1506 - val_accuracy: 0.9484
Epoch 9/20
```

✓ 다층 퍼셉트론 모델

- 은닉층 : 2개의 은닉층에 각각 12개의 노드 사용
- 과적합 방지 : dropout 사용
- 은닉층의 활성화 함수 : relu
- 출력층의 활성화 함수 : sigmoid
- Optimizer : Adam
- Callbacks : 성능개선이 이루어지지 않으면 멈춤
- Loss : 이진 분류(binary_ctrossentropy)
- epoch : 20

4. 신경망 모델

- 언더샘플링의 신경망 모델 학습 곡선



- epoch 16에서 더 이상 손실과 정확도가 개선되지 않아 학습을 멈춤

- 오버샘플링의 신경망 모델 학습 곡선

- epoch 7에서 더 이상 손실과 정확도가 개선되지 않아 학습을 멈춤

#3 모델링 결과

5. 언더 샘플링 모델링 결과

- Test set로 평가한 모델의 confusion matrix

로지스틱 회귀 Confusion Matrix		실제 정답	
		사기	정상
분류 결과	사기	100	1600
	정상	4	52000

랜덤포레스트 Confusion Matrix		실제 정답	
		사기	정상
분류 결과	사기	99	420
	정상	6	53000

LGBM Confusion Matrix		실제 정답	
		사기	정상
분류 결과	사기	100	750
	정상	5	53000

XGBOOST Confusion Matrix		실제 정답	
		사기	정상
분류 결과	사기	100	910
	정상	4	53000

신경망 Confusion Matrix		실제 정답	
		사기	정상
분류 결과	사기	101	1084
	정상	4	52297

5. 언더 샘플링 모델링 결과

- confusion matrix의 결과 수치화

모델	정확도	정밀도	재현율	F1-score	Auc
로지스틱 회귀	0.97	0.06	0.96	0.11	0.966
랜덤 포레스트	0.99	0.19	0.94	0.32	0.967
XGBOOST	0.98	0.10	0.96	0.18	0.972
LGBM	0.99	0.12	0.95	0.21	0.969
신경망	0.98	0.09	0.96	0.16	0.970

→ 언더 샘플링에서는 랜덤포레스트 모델이 F1-score점수가 가장 높음

→ 또한, 정상을 사기로 판단할 오류도 가장 적음

#3 모델링 결과

6. 오버 샘플링 모델링 결과

- Test set로 평가한 모델 confusion matrix

로지스틱 회귀 Confusion Matrix		실제 정답	
		사기	정상
분류 결과	사기	90	1200
	정상	6	52000

랜덤포레스트 Confusion Matrix		실제 정답	
		사기	정상
분류 결과	사기	87	17
	정상	9	53000

LGBM Confusion Matrix		실제 정답	
		사기	정상
분류 결과	사기	85	5
	정상	11	53000

XGBOOST Confusion Matrix		실제 정답	
		사기	정상
분류 결과	사기	86	10
	정상	10	53000

신경망 Confusion Matrix		실제 정답	
		사기	정상
분류 결과	사기	86	102
	정상	10	53288

6. 오버 샘플링 모델링 결과

- confusion matrix의 결과 수치화

모델	정확도	정밀도	재현율	F1-score	Auc
로지스틱 회귀	0.98	0.07	0.94	0.12	0.957
랜덤 포레스트	1.00	0.81	0.90	0.85	0.948
XGBOOST	1.00	0.83	0.90	0.86	0.948
LGBM	1.00	0.93	0.89	0.91	0.943
신경망	1.00	0.46	0.89	0.60	0.946

→ 오버 샘플링에서는 LGBM 모델이 F1-score 점수가 가장 높음

→ 또한, 정상을 사기로 판단할 오류도 가장 적음

00
목차

#1 최종 선정 모델

1. 최종 선정 모델

샘플링	모델	정확도	정밀도	재현율	F1-score	Auc
언더 샘플링	랜덤 포레스트	0.99	0.19	0.94	0.32	0.967
오버 샘플링	LGBM	1.00	0.93	0.89	0.91	0.943

✓ 최종 모델링 결과

- 오버 샘플링한 LGBM 모델이 언더 샘플링한 랜덤포레스트 보다 F1-score 점수가 높으며, 정상 거래를 사기로 탐지할 오류가 낮음

“” 최종 모델로 오버 샘플링한 LGBM 모델 선정 “”

<confusion matrix>

랜덤포레스트 Confusion Matrix		실제 정답	
		사기	정상
분류 결과	사기	99	420
	정상	6	53000

LGBM Confusion Matrix		실제 정답	
		사기	정상
분류 결과	사기	85	5
	정상	11	53000

2. 결론

- 불균형 데이터 셋에는 언더 샘플링보다 오버 샘플링 방법(SMOTE)이 더 효과적이고 분류 모델의 성능이 좋음

- 최종 모델의 LGBM은 정상을 사기로 판단할 오류가 0.03%로 높은 성과를 보이고 있음

- 금융기관에서 우리 모델을 사용해 신용카드의 정상 거래와 사기를 91%의 성능으로 탐지할 수 있음

참고 자료

- 분류성능평가 지표 – 정밀도, 재현율, 정확도

: https://sumniya.tistory.com/26

- 과적합 방지 방법

: https://bit.ly/39nP353

- 비대칭 데이터 문제

: https://bit.ly/3byjmc6

- Smote로 데이터 불균형 해결하기

: https://bit.ly/3shhHgW

참고 자료

- 실무자의 관점에서 신용카드 사기탐지에 대한 교훈

: https://bit.ly/2Xv74sE

- 신용카드 사기탐지를 위한 딥러닝 도메인 적응 기법

: https://bit.ly/3qcQvxT

- 불균형 클래스 분류

: https://dining-developer.tistory.com/27

- '신용카드 결제 사기' 최근 3년간 100만건, 1700억 규모

http://www.opinionnews.co.kr/news/articleView.html?idxno=37069

감사합니다 :)