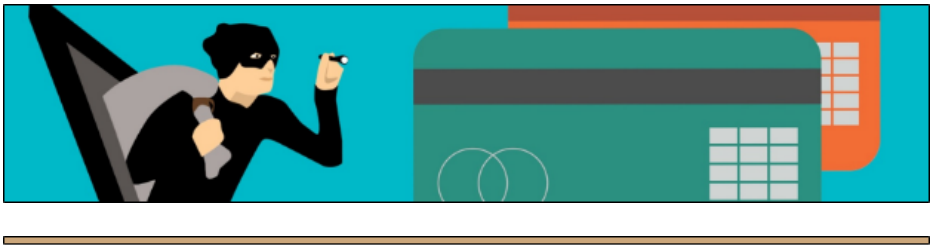
신용카드 사기거래 탐지

1. 팀 구성원 & 역할

- 성완용: EDA / 전처리 / 모델링 / 성능평가
- 심양관: EDA / 전처리 / 모델링 / 성능평가
- 안철현: EDA / 전처리 / 모델링 / 성능평가

2. 데이터 선정 배경 및 이유



- **데이터 링크:** https://www.kaggle.com/datasets/kartik2112/fraud-detection
- 개요: 2019년 1월 1일부터 2020년 12월 31일까지의 합법적 거래와 사기 거래를 포함하는 시뮬레이션된 신용카드 거래 데이터 세트이다. 800개 상인과 거래를 하는 1000명 고객의 신용카드를 다루고 있다.

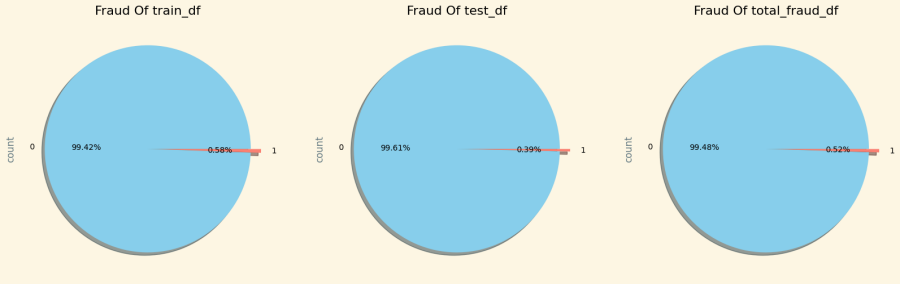
  (데이터 시뮬레이션 참고 링크: https://github.com/namebrandon/Sparkov_Data_Generation/blob/master/profiles/adults_2550_female_rural.json)
- 선정 이유
  1. 금융을 주제로 사기탐지, 탈세자 탐지와 같이 이상을 탐지하여 이진분류 하는 주제를 중심으로 탐색하여, 머신러닝 모델링 주제선정에 적합하다고 판단하였다.
  2. 훈련데이터(1296675) + 테스트데이터(555719) 으로 총 185만개가 넘고, 결측치가 없어 머신러닝 모델링하기에 충분한 데이터양을 갖고 있어 적합하다고 판단하였다.
  3. 신용카드 감지 타깃인 'is_fraud'(사기: 1, 정상:0) 종속변수가 존재하여 이진분류 머신러닝 예측에 용이할 것이라 판단하였다.
- 데이터 컬럼 및 설명

컬럼명	설명
Unnamed	데이터셋 로드 시 생긴 인덱스 컬럼. 별다른 의미는 없으며, 분석에서 제외할 수 있음.
trans_date_trans_time	거래 발생 날짜와 시간. 각 거래가 이루어진 시점.
cc_num	신용카드 번호. 거래에 사용된 신용카드 번호로, 실제 번호 대신 해시 처리된 값.
merchant	가맹점 이름. 거래가 이루어진 가맹점(또는 판매업체)의 이름.
category	가맹점의 카테고리. 해당 가맹점이 속한 업종을 나타내며, 예를 들어 grocery_pos는 식료품 관련 업종을 의미.
amt	거래 금액. 거래에서 사용된 금액. 달러($)로 예상됨.
first	카드 소유자의 이름. 이름 값이 일부 포함되어 있으며, 개인정보 보호를 위해 일부 데이터는 삭제 또는 해시된 것으로 추정.
last	카드 소유자의 성. 마찬가지로, 개인정보 보호를 위해 처리된 것으로 추정.
gender	카드 소유자의 성별. F는 여성, M은 남성을 나타냄.
street	카드 소유자의 주소 (거리). 거래를 수행한 카드 소유자의 주소 일부를 나타냄.
city	카드 소유자가 거주하는 도시.

state	카드 소유자가 거주하는 주.
zip	카드 소유자가 거주하는 지역의 우편번호.
lat	카드 소유자의 위도. 카드 소유자가 위치한 위도.
long	카드 소유자의 경도. 카드 소유자가 위치한 경도.
city_pop	도시 인구. 카드 소유자가 위치한 도시의 인구.
job	카드 소유자의 직업. 개인정보를 보호하기 위해 직업 이름 일부는 해시 처리된 것으로 추정.
dob	'date of birth'의 약자로 카드 소유자의 생년월일. 개인정보 보호를 위해 일부 변경된 것으로 추정.
trans_num	거래 번호. 각 거래의 고유 식별자.
unix_time	거래 발생 시간의 Unix 타임스탬프. trans_date_trans_time과 같은 정보이지만, Unix 타임스탬프로 변환된 형태.
merch_lat	가맹점의 위도. 거래가 이루어진 가맹점의 위치 위도.
merch_long	가맹점의 경도. 거래가 이루어진 가맹점의 위치 경도.
is_fraud	거래 사기 여부. 0은 정상 거래, 1은 사기 거래를 의미.

3. 탐색적 데이터 분석(EDA)

- 불균형 데이터(Imbalanced Data)



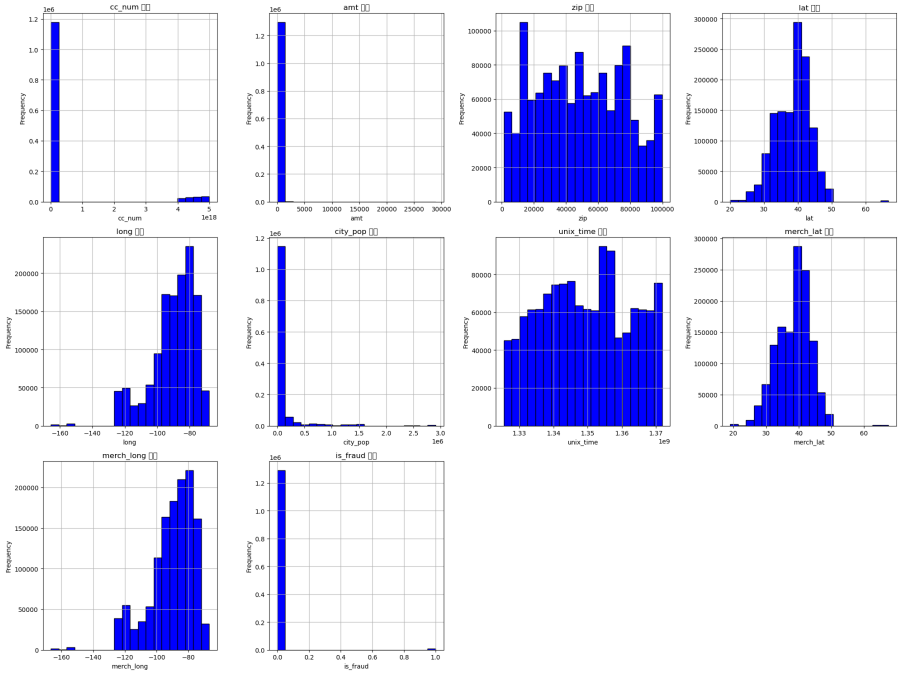
  - 전체 row 의 수(1852394) 중 신용카드 사기거래('is_fraud:1) 의 수(10747)의 비율은 약 0.5%로 매우 작은 비율의 불균형한 데이터(imbalanced data)의 특징을 가지고 있다.

- 변수 데이터 타입 확인

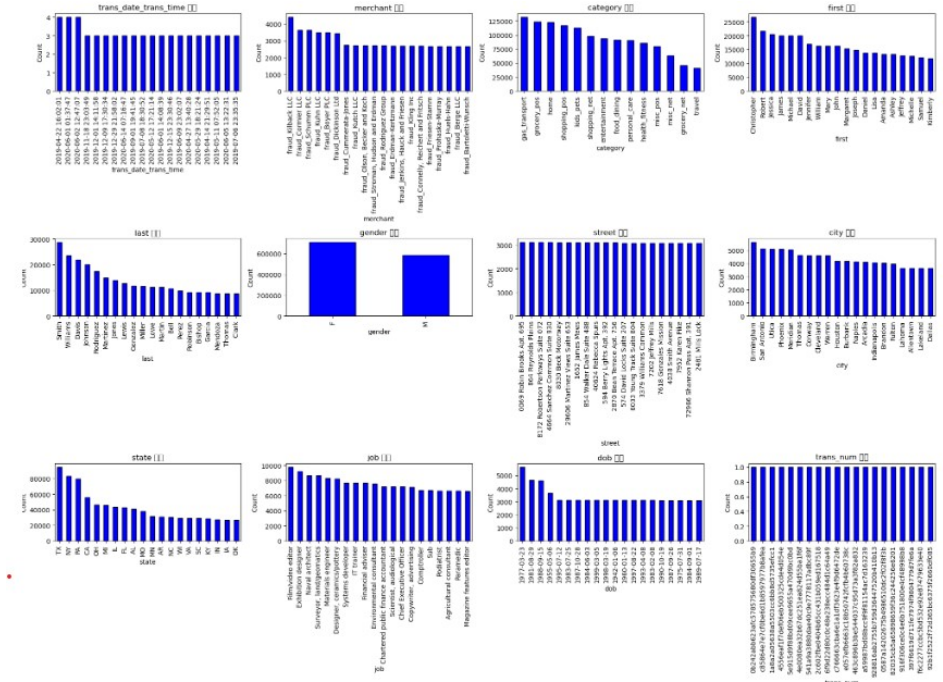
```
Unnamed: 0                int64
trans_date_trans_time    object
cc_num                    int64
merchant                 object
category                 object
amt                     float64
first                    object
last                     object
gender                   object
street                   object
city                     object
state                    object
zip                       int64
lat                     float64
long                    float64
city_pop                  int64
job                      object
```

```
dob                      object
trans_num                object
unix_time                 int64
merch_lat               float64
merch_long              float64
is_fraud                  int64
dtype: object
```

- 수치형 변수 시각화



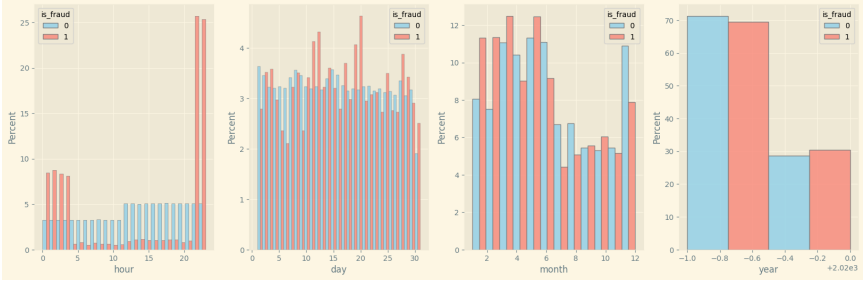
- 범주형 변수에 대한 시각화



- 거래날짜시간(trans_date_trans_time)
  - 시간대 분석을 위해 `trans_date_trans_time` 컬럼에서 hour, day, month, year 를 추출하고 `is_fraud` 를 가져와서 time_df 데이터프레임을 생성.

	hour	day	month	year	is_fraud
0	0	1	1	2019	0
1	0	1	1	2019	0
2	0	1	1	2019	0
3	0	1	1	2019	0
4	0	1	1	2019	0
...	...	...	...	...	...
1296670	12	21	6	2020	0
1296671	12	21	6	2020	0
1296672	12	21	6	2020	0
1296673	12	21	6	2020	0
1296674	12	21	6	2020	0

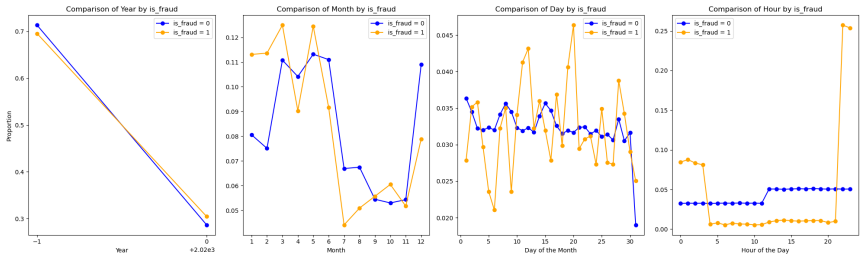
  - hour, day, month, year 별 is_fraud 비율 시각화



  - 관찰:
    - 시간별 정상거래는 12이후 거래량이 증가했으며, 사기거래는 0-4시, 23-24시에 많이 나타났다.

- 날짜별 정상거래는 전반적으로 균등하였으나, 말일(31일)에 거래량이 다소 감소하였다. 사기거래는 월초, 월말 보다는 중순에 거래량이 다소 증가세를 보였다.
- 월별는 상반기에 사기거래가 정상거래보다 높은 양상을 보이며 하반기에는 비슷한 양상을 보이다 12월 사기거래는 정상거래보다 현저하게 적다.
- 년도별 정상거래와 사기거래는 차별성을 보이지 않지만, 2019년에 비해 2020년에는 절반이상 감소하였다. (covid19 로 전체 거래량이 감소한것으로 추정된다.)

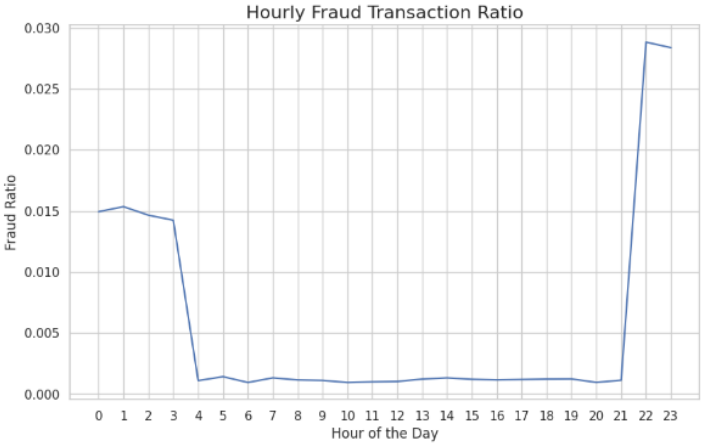
◦ 정상거래 사기거래에 따른 거래시간(year,month,day,hour) 비율 비교

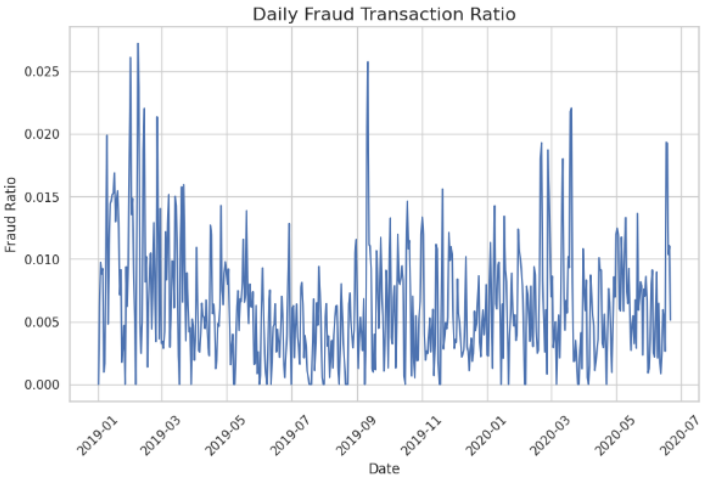


◦ 관찰:

- Year(년도) : 19년도 대비 20년대에 거래(정상,사기)비율이 50% 정도 떨어진 것을 확인할 수 있다.
- Month(월) : 정상거래 비율은 봄에 많이 일어나고 사기거래 비율은 겨울에 많이 일어남을 확인할 수 있다.
- Day(일) : 정상거래 비율은 비슷한 수치로 일어나다 말일에 거래율이 떨어지고 사기거래 비율은 날마다 차이가 나타나며 6일에 가장 적게 일어나고 20일에 가장 많이 일어남을 확인할 수 있다.
- Hour(시간) : 정상거래율은 큰 차이가 나타나지 않지만 사기거래율은 22-3시(늦은 밤시간)에 많이 발생한다는 것을 알 수 있다.

◦ 사기 거래와 시간/날짜 별 상관관계





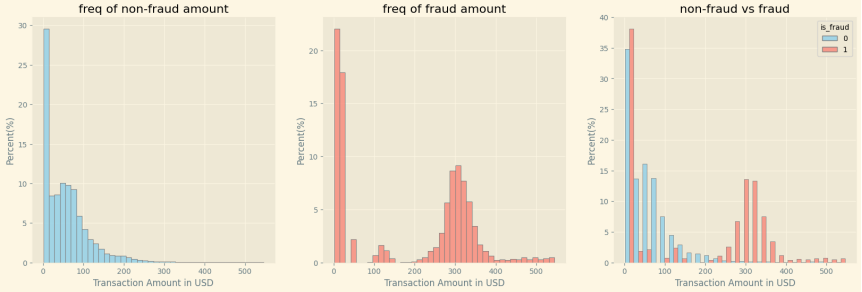
- 관찰:
  - 시간 별 사기거래 비율을 보면 21~4시까지의 비율이 타 시간대의 비율보다 높다. 이를 통해 늦은 저녁부터 새벽까지 거래를 사기거래로 의심해볼 수 있으며 시간을 주요한 피처로 볼 수 있다.
  - 날짜 별 사기거래를 보면 특정 날짜에 높은 사기 거래 비율이 높은 것을 볼 수 있다. 이를 반영하여 해달 날짜들의 거래를 의심해 볼 수 있지만 날짜를 주요한 피처로 보는 것은 힘들 것으로 보인다.

- 거래금액(amt)
  - 사기거래와 거래량(amt)

  - 관찰:

- 정상 거래(0)의 분포: 사기거래가 아닌 경우, 대부분의 거래 금액은 낮은 값에 집중되어 있으며, 많은 이상치(outliers)가 위쪽으로 나타나 있다. 이는 정상 거래에서 고액의 거래가 드물지만, 일부 높은 금액의 거래가 존재한다는 것을 의미한다.
- 사기 거래(1)의 분포: 사기거래인 경우, 박스가 상대적으로 정상 거래보다 높게 나타나며, 사기 거래의 중앙값이 더 높다. 이는 사기 거래에서 평균적인 거래 금액이 정상 거래보다 높을 수 있음을 말한다.
- 전체적인 패턴: 두 그룹 간 박스의 위치가 다르다는 점에서 거래 금액(amt)이 사기 거래 여부를 예측하는 데 유의미한 특징이 될 수 있음을 알 수 있다. 정상 거래의 경우 금액 분포가 넓고, 사기 거래의 경우 분포가 비교적 좁기 때문에 금액에 따른 구분이 가능하다. 이를 통해, 모델링 시 amt 변수를 중요한 피처로 고려할 수 있다.

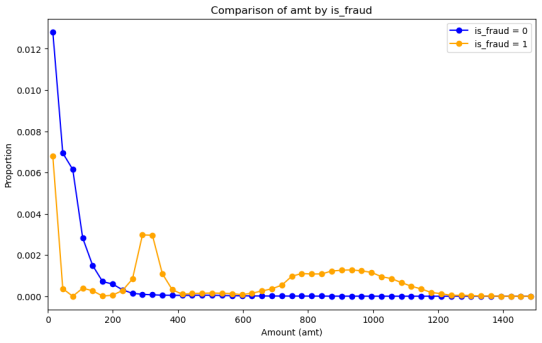
◦ Amount 와 is_fraud 시각화



▪ 관찰:

- 정상거래는 대다수가 아주 적은 금액.
- 사기거래는 아주작은 금액도 있지만, 200 ~ 400 달러 금액대에서 사기거래가 많은 양상을 보임.

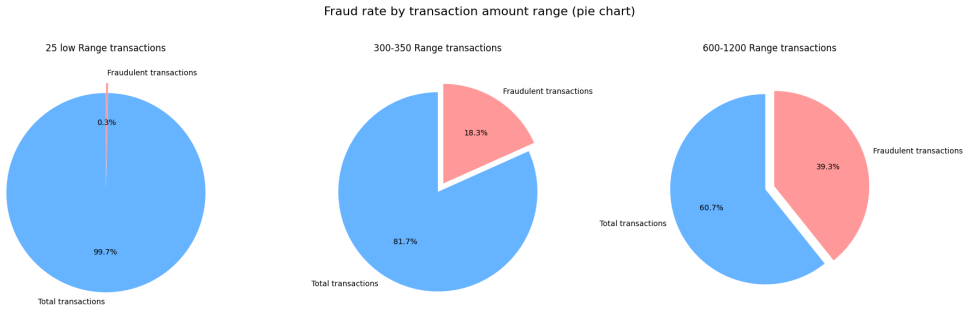
◦ 정상거래 사기거래에 따른 거래금액(amt) 비율 비교



▪ 관찰:

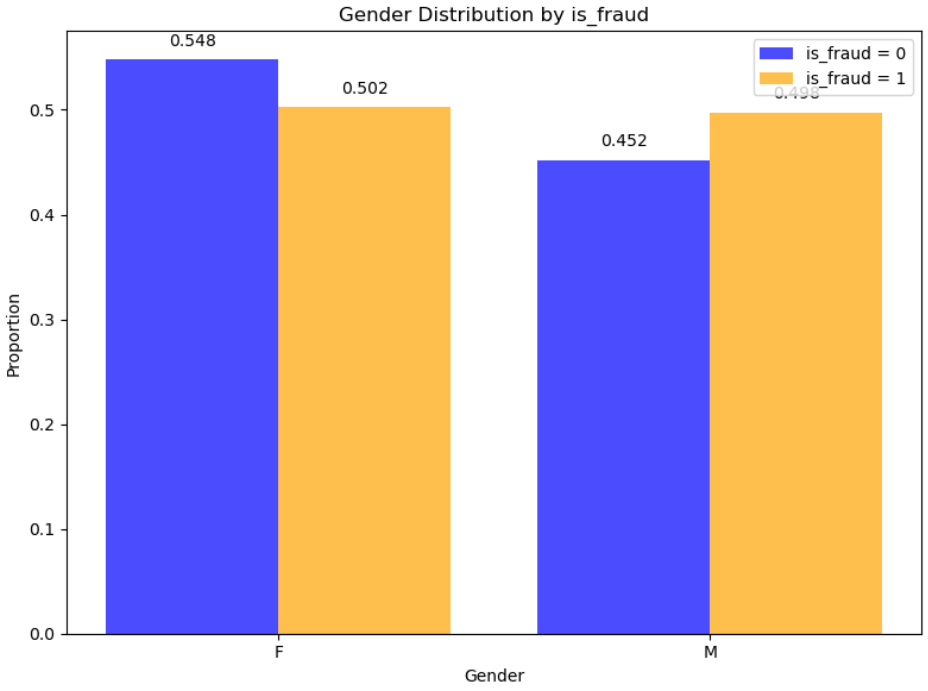
- 정상거래는 주로 200달러이하의 저가거래로 많이 발생함을 알 수 있다.
- 사기거래는 비교적 고가거래에서도 많이 발생함을 확인할 수 있다.

◦ 25$이하, 300-350$, 600-1200$ 사기확률/전체확률



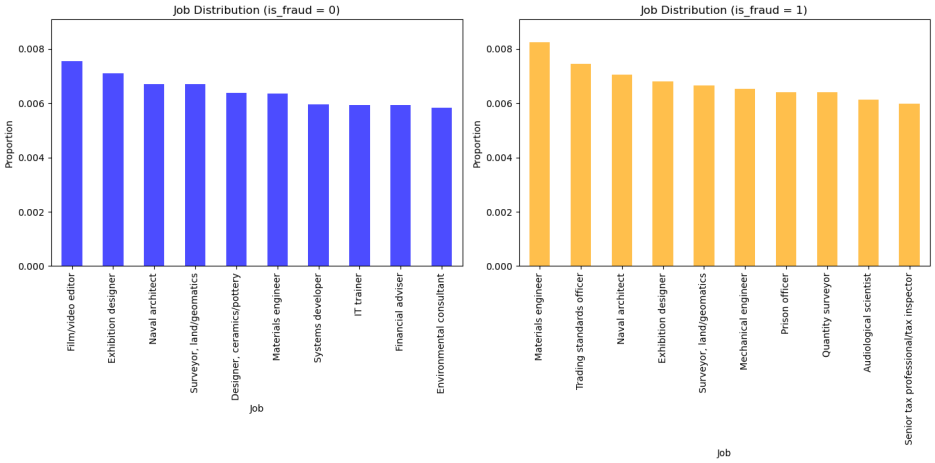
- 관찰:
  - 거래액이 증가할수록 전체확률에서 사기거래 확률이 점점 증가함을 확인할 수 있다.

- 성별(gender)



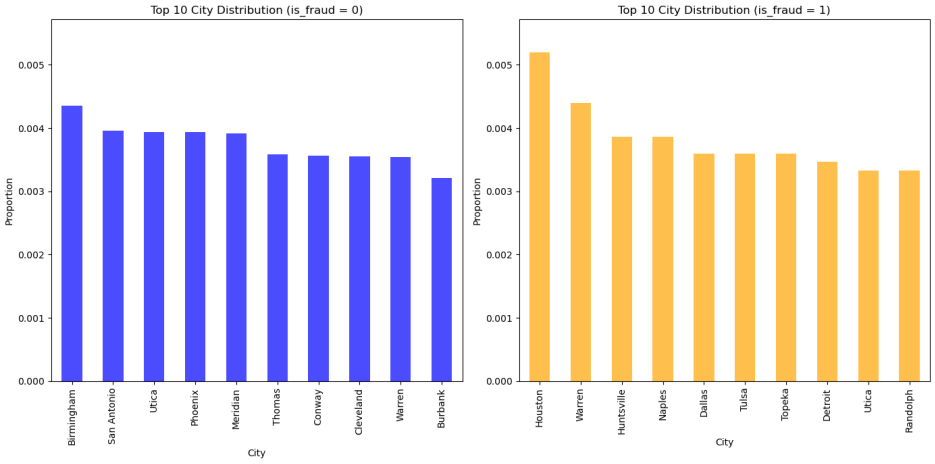
  - 관찰:
    - 정상거래에서는 여성의 거래가 0.1% 정도 더 많고 사기거래에서는 남여 비율 차이가 거의 나지 않는다는 것을 알 수 있다.

- 직업(job)



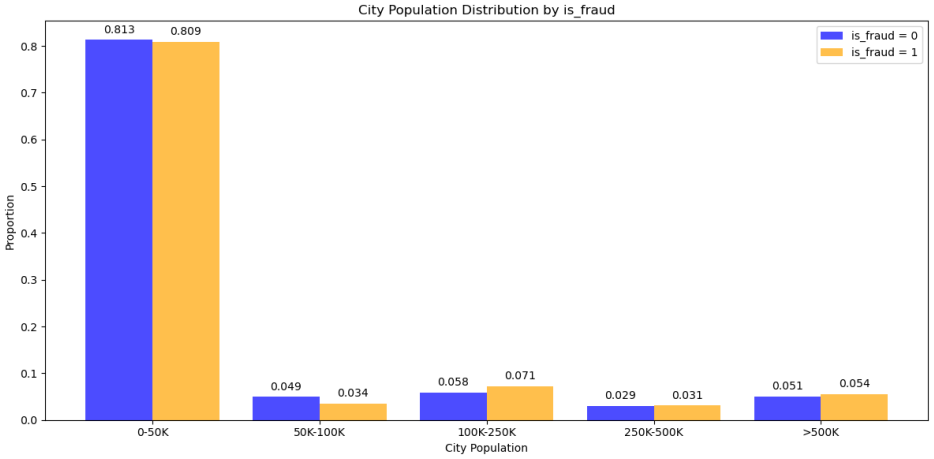
- 관찰:
  - 직업별 정상, 사기거래의 분포가 다양한 것을 알 수 있고 특정 직업에 대한 거래비율이 높지 않다는 것을 확인할 수 있다.

- 도시(city)



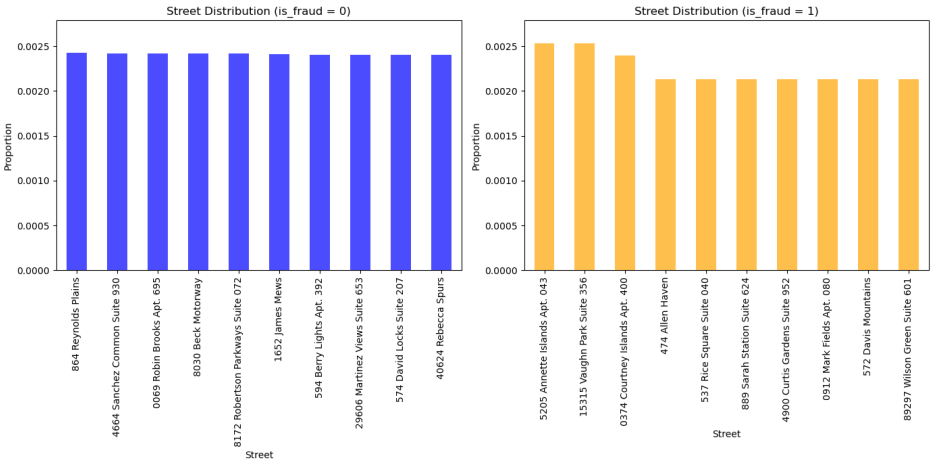
- 관찰:
  - 도시별 정상, 사기거래 비율은 서로 상이하다는 것을 알 수 있으며 비율 또한 특정 도시에 집중되어 있지 않고 다양하게 분포되어 있음을 확인할 수 있다.

- 도시인구(city_pop)



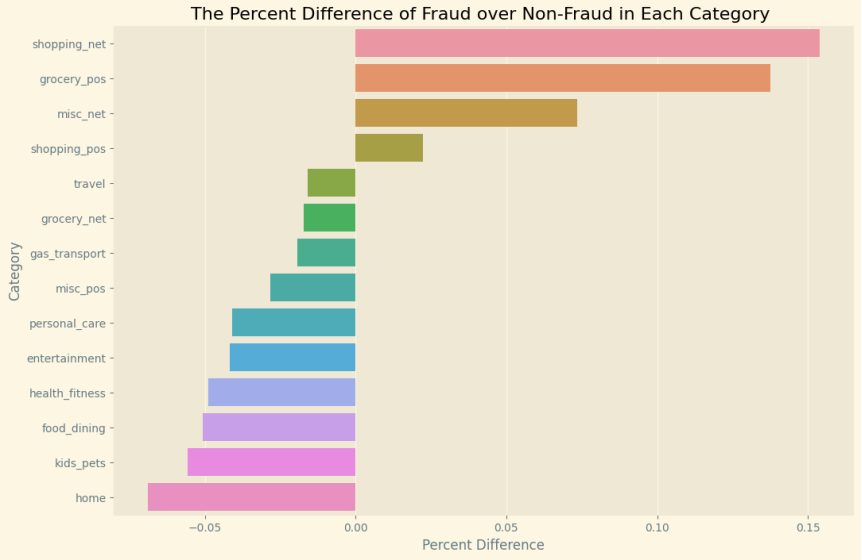
- 관찰:
  - 정상거래, 사기거래 모두 소도시에 발생하는 비율이 압도적으로 높은 것을 확인할 수 있다.

- 거리주소(street)



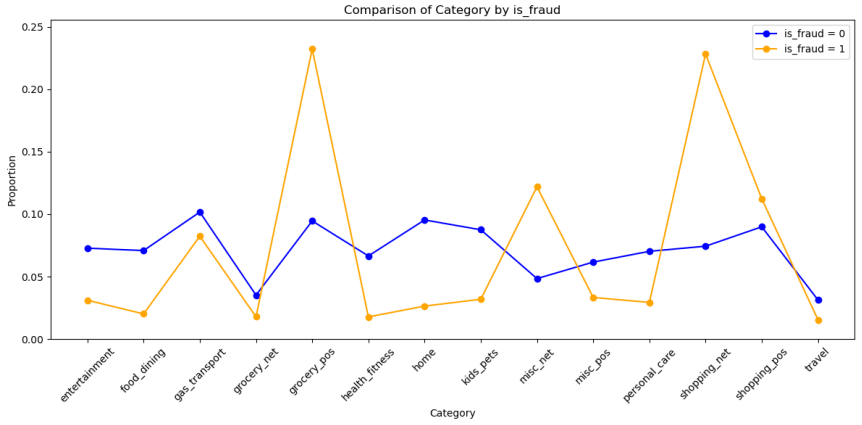
- 관찰:
  - 정상거래, 사기거래의 거리주소 분포가 매우 다양하다는 것을 그래프를 통해 확인할 수 있었다.

- 거래 점포의 카테고리(category)
  - 거래된 점포에서 단순거래량으로 체크하면 특정한 패턴을 찾기 어렵다.
  - 정상거래율과 사기거래율의 차이로 특정한 패턴을 파악한다.
  - 차이('diff') = 사기거래율 - 정상거래율
  - 카테고리(category) 차이('diff') 시각화



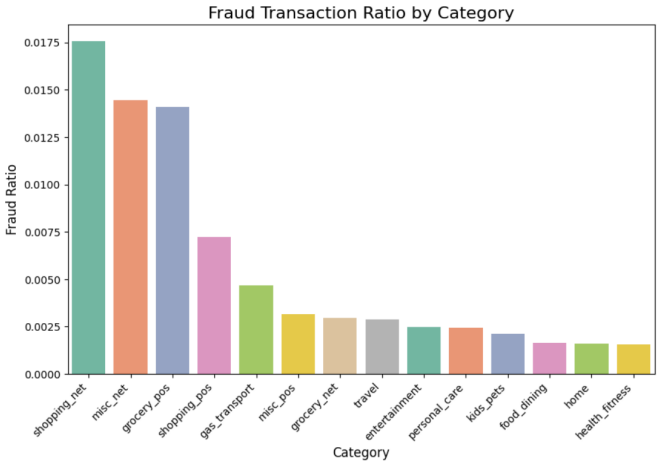
- 관찰:
  - **shopping_net, grocery_pos, misc_net, shopping_pos** 에서 사기거래 비율이 높은 반면, **home, kids_pets, food_dining** 은 정상거래 비율이 더 높다. 참고(_net: 온라인, _pos: 오프라인)

- 정상거래,사기거래에 따른 카테고리 종류 비율 비교



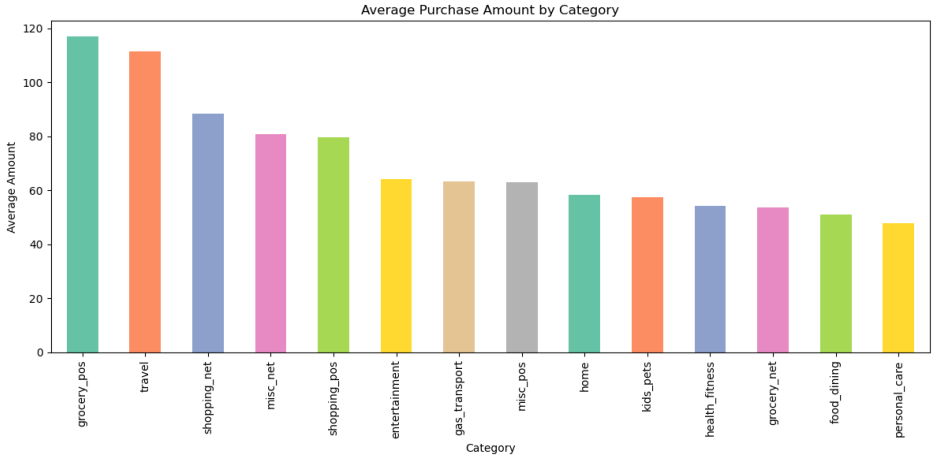
- 관찰:
  - 정상거래에서는 각 카테고리별 균등한 분포를 띄고 있으며 사기거래에서는 grocery_pos와 shopping_net 비율이 비교적 높게 나오는 것을 확인할 수 있다.

- 카테고리 별 사기 거래



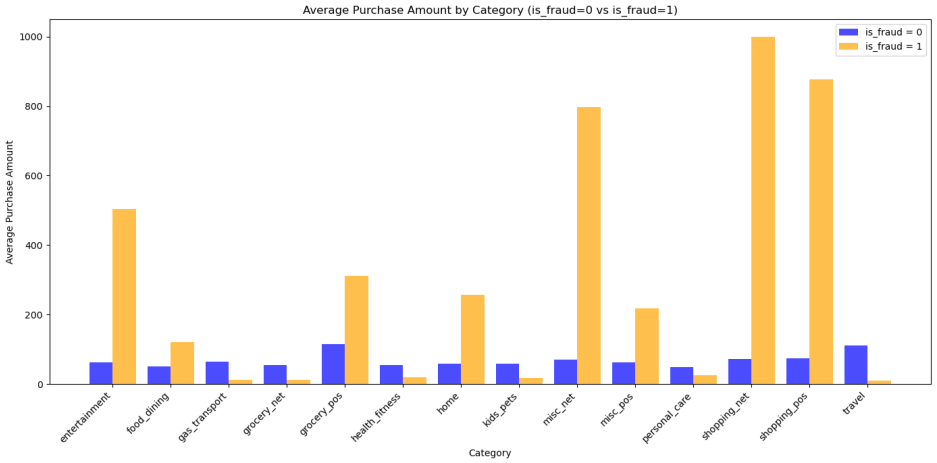
- 관찰:
  - 사기 거래 비율에 있어서 Shopping_net, Misc_net, Grocery_pos이 타 카테고리에 비해서 높은 비율을 갖고 있는 것을 볼 수 있다. 위 그래프를 보면 Shopping, Misc, Grocery 카테고리를 유의해서 봐야 할 필요가 있다는 것을 알 수 있다. 이를 통해 카테고리 또한 유의미한 피처가 될 수 있다고 볼 수 있다.

- 카테고리별 정상+사기거래 평균 구매액 비교



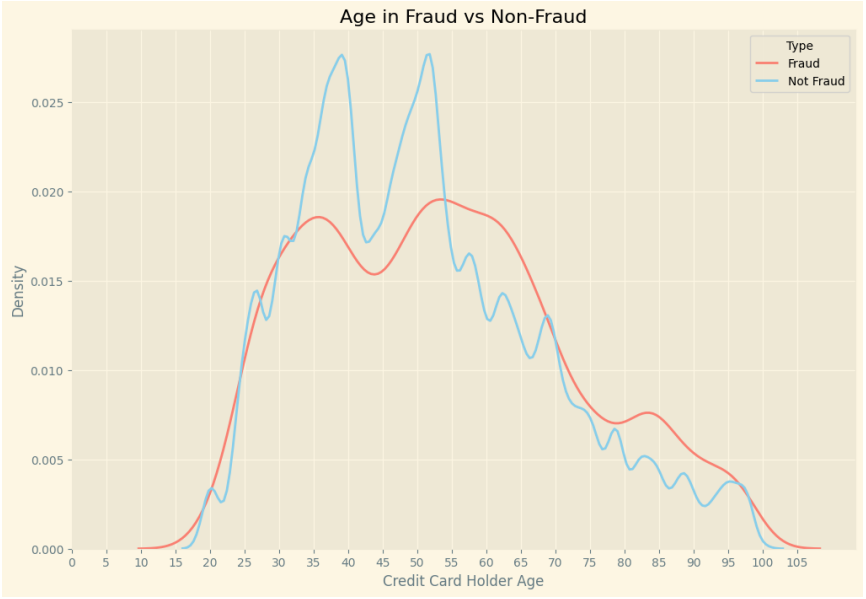
- 관찰:
  - grocery_pos, travel의 평균 구매액이 100달러 이상으로 타 카테고리들에 비해 높은 것을 확인 할 수 있다.

- 카테고리별 정상, 사기거래 평균 구매액 비교



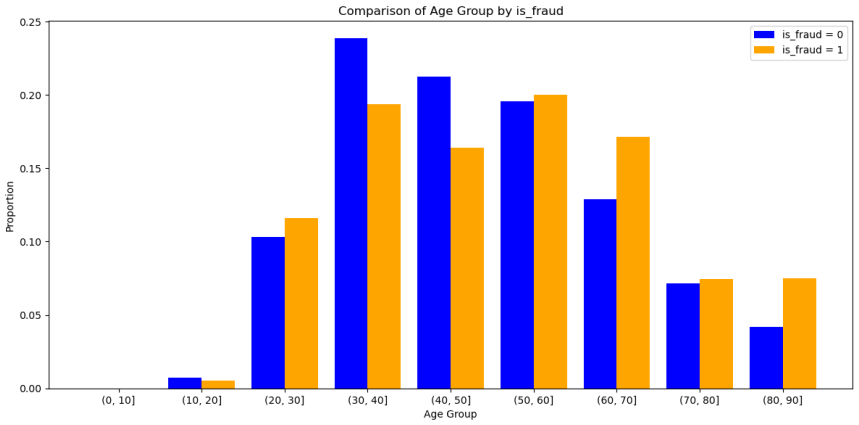
- 관찰:
  - 정상거래에서는 평균 구매액에서 큰 차이를 보이지 않음을 확인할 수 있다.
  - 사기거래에서는 쇼핑쪽에서 800달러 이상의 고가의 거래가 발생하며 그 다음으로 misc_net(잡화점)의 순위가 높았다.

- 생일(dob)
  - 나이(age) : 생일(dob) 에서 추출
  - 나이(age) 와 is_fraud 시각화



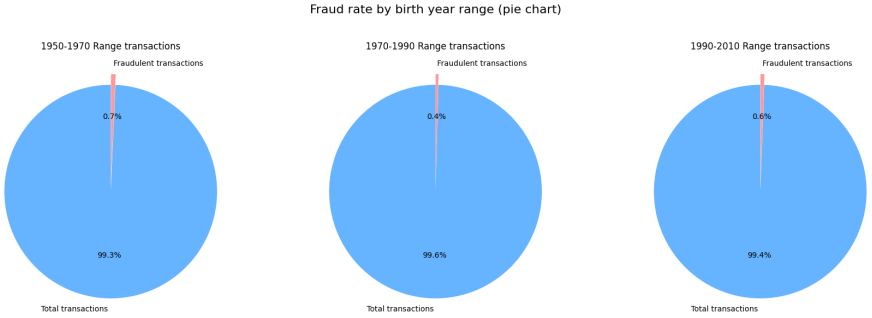
- 관찰:
  - 정상거래에서는 37-38세, 49-50세에서 정상을 가리키고 있고, 그와 동반하여 사기거래도 그보다더 적지만 높은 양상을 가지고 있다.
  - 전반적으로 거래량이 많을 수록 사기거래도 많지만, 약 54세 이하대비 이상에서는 정상거래보다 사기거래가 많다. 결론적으로 **고령층에서 사기거래율**이 높다.

  - 정상거래, 사기거래에 따른 dob변수를 활용한 나이대별 비율 비교



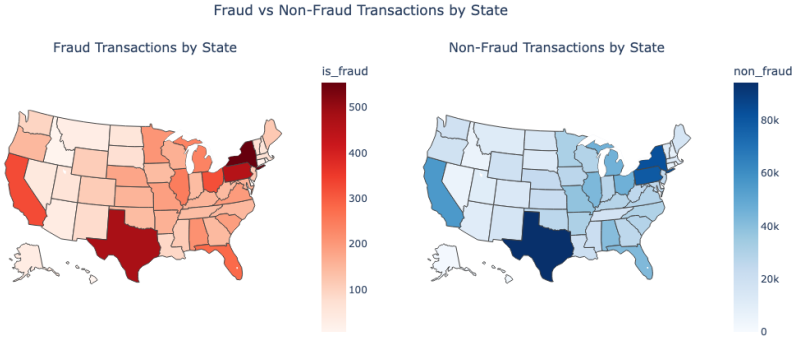
- 관찰:
  - 정상거래, 사기거래 비율이 30-60살 사이의 사람들에게 모두 높게 나타난 것을 확인할 수 있다.
  - 사기거래는 고령층에서 정상거래 대비 높게 나타난 것을 확인할 수 있었다.

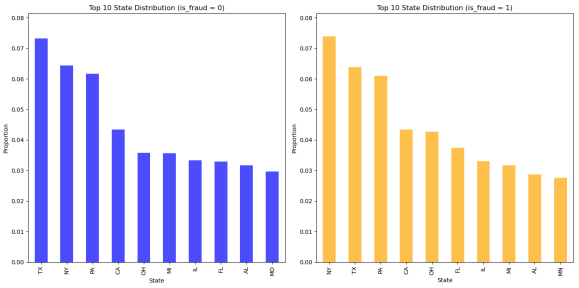
- 50-70년대, 70-90년대, 90-10년대 사기확률/전체확률



- 관찰:
  - 50-70년대(장년층), 90-10(청년층), (70-90)중년층 순으로 사기거래비율이 높게 나타났다.

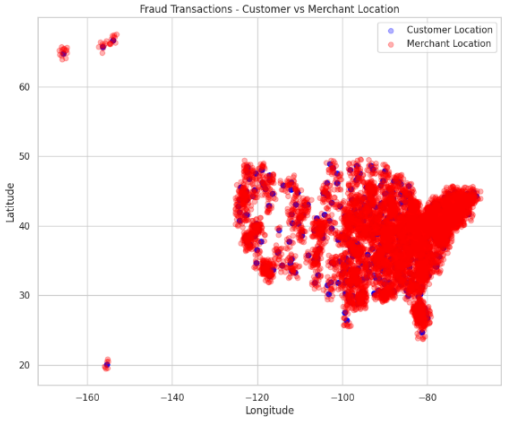
- 지역(state)
  - 지역(state) 별 is_fraud 시각화





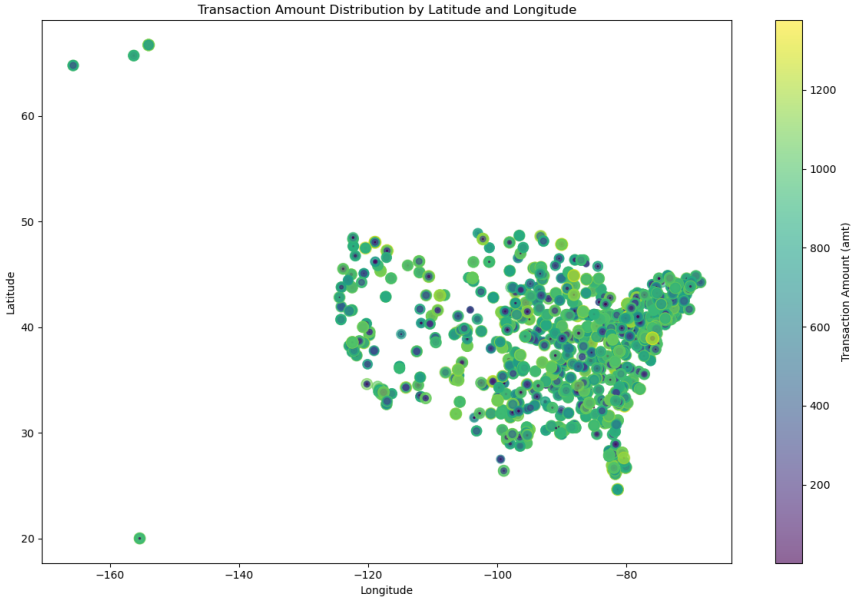
- 관찰:
  - 지역(주:state)별 정상거래와 사기거래량은 전반적으로 비슷한 양상을 보인다.
  - 하지만 정상거래는 텍사스(94397), 뉴욕(82946), 펜실베니아(79389) 순으로 나타난 반면, 사기거래는 뉴욕(555), 텍사스(479), 펜실베니아(458) 순으로 나타났다. --> **뉴욕**이 정상거래량 대비 **사기거래량 비율**이 더 높았다.

- 위치(위도, 경도) 별 사기 거래 분포



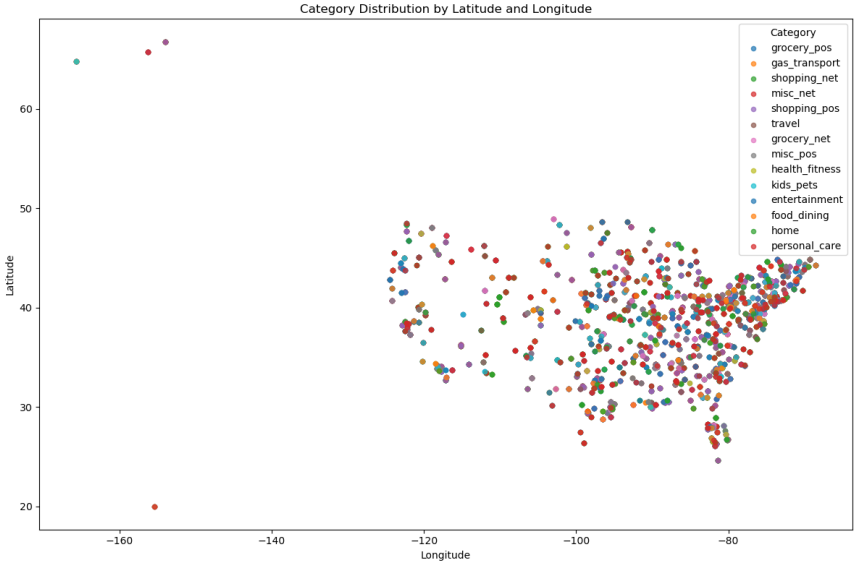
- 관찰:
  - 사기 거래에 대한 소비자 위치와 상점 위치를 각각 파란 점과 빨간 점으로 위도, 경도에 따라 표기한 시각화 지표이다. 전체 사기거래에 있어서 대부분의 사기거래들은 위도 30~50, 경도 -120~-80의 위치에서 주로 밀집된 것을 알 수 있다. 이를 통해 위치 또한 하나의 주요한 피처가 될 수 있다는 것을 알 수 있다.

- 위치(위도, 경도) 별 사기 거래의 거래액 분포



- 관찰:
  - 사기거래가 일어나는 위치에서의 거래금액을 시각화 해본결과 미국 동남부쪽의 고가의 사기거래가 많이 일어나고 있음을 확인할 수 있다.

- 위치(위도, 경도) 별 사기 거래의 카테고리 분포



- 관찰:
  - shopping_net,misc_net,grocery_pos 순으로 색상이 분포하고 있으며 거래분포는 미국 동남부쪽에 집중적으로 분포되어 있음을 확인할 수 있다.

4. 데이터 전처리(Data Preprocessing)

팀원1

4.1 피처 엔지니어링(Feature Engineering)

- 아래 모델링 중 class weights, SMOTE oversampling, NO Under/Oversampling, Random Undersampling 에 적용된 피처 엔지니어링
  - **hour, day, month, year** 추출

```
# time_df 생성
hour = pd.to_datetime(df['trans_date_trans_time']).dt.hour
day = pd.to_datetime(df['trans_date_trans_time']).dt.day
month = pd.to_datetime(df['trans_date_trans_time']).dt.month
year = pd.to_datetime(df['trans_date_trans_time']).dt.year
is_fraud = df['is_fraud']
time_df = pd.DataFrame({'hour': hour, 'day': day, 'month': month, 'year': year, 'is_

# df 에 병합
df[['hour', 'day', 'month', 'year']] = time_df[['hour', 'day', 'month', 'year']]
```

    - trans_date_trans_time 으로 부터 hour, day, month, year 를 추출하여 time_df 데이터 프레임을 생성하여 df 에 병합하였다.
  - **age** 추출

```
# 'age' 생성
import datetime as dt

df['age'] = dt.date.today().year - pd.to_datetime(df['dob']).dt.year
```

    - 오늘(2024년)에서 'dob'의 년도를 빼서 'age' 생성 및 df에 병합하였다.

  - 컬럼 선택(Collecting Columns)

```
# 컬럼 선택
collected_columns = ['is_fraud', 'hour', 'day', 'month', 'year', 'category', 'amt',
collected_df = df[collected_columns]
```

    - EDA 를 통해 유의미한 판단이 되는 컬럼들을 선택하였다.

4.2 인코딩(Encoding)

- 아래 모델링 중 class weights, SMOTE oversampling, NO Under/Oversampling, Random Undersampling 에 적용된 인코딩
  - 라벨인코딩(LabelEncoding)

```
# object 컬럼들의 unique 값 확인
print(f"unique number of category: {collected_df['category'].nunique()}")
print(f"unique number of city: {collected_df['city'].nunique()}")
print(f"unique number of state: {collected_df['state'].nunique()}")
print(f"unique number of job: {collected_df['job'].nunique()}")

# unique number of category: 14
# unique number of city: 906
# unique number of state: 51
# unique number of job: 497

# LabelEncoding
from sklearn.preprocessing import LabelEncoder
```

```
le = LabelEncoder()

le.fit(collected_df['category'])
encoded_category = le.transform(collected_df['category'])
collected_df['enc_category'] = encoded_category

le.fit(collected_df['city'])
encoded_city = le.transform(collected_df['city'])
collected_df['enc_city'] = encoded_city

le.fit(collected_df['state'])
encoded_state = le.transform(collected_df['state'])
collected_df['enc_state'] = encoded_state

le.fit(collected_df['job'])
encoded_job = le.transform(collected_df['job'])
collected_df['enc_job'] = encoded_state
```

- dtype 이 object인 컬럼들의 unique 값을 확인해서 라벨인코딩(LabelEncoding) 을 사용하기로 결정하였다.
- sckitlearn 의 LabelEncoder 를 임포트 하여 **'category', 'city', 'state', 'job'** object 인 컬럼들의 값을 인코딩(str → integer)하였다.

- 상관관계(Correlation) 확인

	is_fraud	hour	day	month	year	amt	zip	city_pop	unix_time	age	enc_category	enc_city	enc_state	enc_job
is_fraud	1.000000	0.013196	-0.000131	-0.016417	-0.006022	0.209308	-0.002190	0.000325	-0.013329	0.011103	0.019278	-0.001107	-0.000252	-0.000252
hour	0.013196	1.000000	-0.000155	-0.000986	-0.000329	-0.024891	0.005947	0.019949	0.000571	-0.173053	0.157991	0.006995	0.007033	0.007033
day	-0.000131	-0.000155	1.000000	0.010817	0.000743	0.000069	-0.000442	0.000654	0.048073	-0.000556	-0.000283	0.000768	-0.000741	-0.000741
month	-0.016417	-0.000986	0.010817	1.000000	0.000730	-0.002593	0.001160	-0.000510	0.496730	0.000039	0.001071	-0.000024	-0.000280	-0.000280
year	-0.006022	-0.000329	0.000743	0.000730	1.000000	-0.001266	0.000522	-0.001660	0.867243	-0.004624	0.000357	0.000722	-0.000195	-0.000195
amt	0.209308	-0.024891	0.000069	-0.002593	-0.001266	1.000000	0.001979	0.004921	-0.002411	-0.010662	0.029665	-0.000240	0.000433	0.000433
zip	-0.002190	0.005947	-0.000442	0.001160	0.000522	0.001979	1.000000	0.077601	0.001017	0.010347	0.003013	0.075212	-0.116563	-0.116563
city_pop	0.000325	0.019949	0.000654	-0.000510	-0.001660	0.004921	0.077601	1.000000	-0.001636	-0.090867	0.009318	0.034824	-0.012633	-0.012633
unix_time	-0.013329	0.000571	0.048073	0.496730	0.867243	-0.002411	0.001017	-0.001636	1.000000	-0.004249	0.001043	0.000657	-0.000331	-0.000331
age	0.011103	-0.173053	-0.000556	0.000039	-0.004624	-0.010662	0.010347	-0.090867	-0.004249	1.000000	-0.003262	-0.018322	-0.050456	-0.050456
enc_category	0.019278	0.157991	-0.000283	0.001071	0.000357	0.029665	0.003013	0.009318	0.001043	-0.003262	1.000000	-0.000045	0.000164	0.000164
enc_city	-0.001107	0.006995	0.000768	-0.000024	0.000722	-0.000240	0.075212	0.034824	0.000657	-0.018322	-0.000045	1.000000	-0.042611	-0.042611
enc_state	-0.000252	0.007033	-0.000741	-0.000280	-0.000195	0.000433	-0.116563	-0.012633	-0.000331	-0.050456	0.000164	-0.042611	1.000000	1.000000
enc_job	-0.000252	0.007033	-0.000741	-0.000280	-0.000195	0.000433	-0.116563	-0.012633	-0.000331	-0.050456	0.000164	-0.042611	1.000000	1.000000

- is_fraud 와 가장 상관관계가 높은 컬럼은 'amt' 로 0.209308 가 나왔다.
- is_fraud 와 2번째로 상관관계가 높은 컬럼은 'age' 로 0.011103 가 나왔다.
- 'day', 'month', 'year', 'zip', 'unix_time', 'enc_city', 'enc_state', 'enc_job' 은 음의 상관관계가 나왔다.

4.3 학습, 테스트 데이터 분리(Train & Test Split)

- 아래 모델링 중 class weights, SMOTE oversampling, NO Under/Oversampling, Random Undersampling 에 적용된 학습, 테스트 데이터 분리
  - 데이터 분리(Split Train, Test)

```
# X: 독립 변수(특징 데이터), y: 종속 변수(레이블 데이터)
X = collected_df[['hour', 'day', 'month', 'amt', 'zip', 'city_pop', 'unix_time', 'ag
y = collected_df['is_fraud']

print(X.shape)
print(y.shape)

# (1852394, 12)
```

```
# (1852394,)

# train test 데이터 분리
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_stat
```

- X: 독립 변수(특징 데이터), y: 종속 변수(레이블 데이터) 로 분리하고, 8:2 의 비율로 Train 데이터와 Test 데이터를 분리하였다.

4.4 스케일링(Scaling)

- 아래 모델링 중 class weights, SMOTE oversampling, NO Under/Oversampling, Random Undersampling 에 적용된 스케일링
  - 스탠다드스케일링(StandardScaler)

```
from sklearn.preprocessing import StandardScaler

# scaler 생성
scaler = StandardScaler()

# train 데이터 fit
scaler.fit(X_train)

# fit 된 스케일러로 transform
X_train_scaled = scaler.transform(X_train)
# train 데이터로 fit 된 스케일러로 test 데이터 transform (일관성 유지)
X_test_scaled = scaler.transform(X_test)
```

- StandardScaler 를 임포트 하여 fit 하고 일관성 유지를 위해 train, test 데이터에 transform 하였다.

팀원2

4.1 피처 엔지니어링(Feature Engineering)

```
import pandas as pd

# 불필요한 열 제거 (df1)
df1_cleaned = df1.drop(columns=['Unnamed: 0', 'first', 'last', 'street', 'dob'])

# trans_date_trans_time을 datetime 형식으로 변환 (df1)
df1_cleaned['trans_date_trans_time'] = pd.to_datetime(df1_cleaned['trans_date_trans_time']

# 새로운 피처 생성 (df1)
df1_cleaned['transaction_date'] = df1_cleaned['trans_date_trans_time'].dt.date
df1_cleaned['transaction_time'] = df1_cleaned['trans_date_trans_time'].dt.time
df1_cleaned['transaction_hour'] = df1_cleaned['trans_date_trans_time'].dt.hour
df1_cleaned['transaction_day'] = df1_cleaned['trans_date_trans_time'].dt.day
df1_cleaned['transaction_month'] = df1_cleaned['trans_date_trans_time'].dt.month
df1_cleaned['transaction_year'] = df1_cleaned['trans_date_trans_time'].dt.year
df1_cleaned['transaction_dayofweek'] = df1_cleaned['trans_date_trans_time'].dt.dayofweek

# 수치형 변수 목록
numeric_cols = ['amt', 'lat', 'long', 'city_pop']

# X와 y 데이터 분리 (df1)
```

```
X = df1_encoded.drop(columns=['is_fraud', 'trans_date_trans_time', 'transaction_date', 'tr
y = df1_encoded['is_fraud']

# 수치형 열만 선택 (df1)
X_numeric = X.select_dtypes(include=['float64', 'int64'])

# --------- df2에 대해 동일한 전처리 진행 (SMOTE는 제외) ---------

# 불필요한 열 제거 (df2)
df2_cleaned = df2.drop(columns=['Unnamed: 0', 'first', 'last', 'street', 'dob'])

# trans_date_trans_time을 datetime 형식으로 변환 (df2)
df2_cleaned['trans_date_trans_time'] = pd.to_datetime(df2_cleaned['trans_date_trans_time']

# 새로운 피처 생성 (df2)
df2_cleaned['transaction_date'] = df2_cleaned['trans_date_trans_time'].dt.date
df2_cleaned['transaction_time'] = df2_cleaned['trans_date_trans_time'].dt.time
df2_cleaned['transaction_hour'] = df2_cleaned['trans_date_trans_time'].dt.hour
df2_cleaned['transaction_day'] = df2_cleaned['trans_date_trans_time'].dt.day
df2_cleaned['transaction_month'] = df2_cleaned['trans_date_trans_time'].dt.month
df2_cleaned['transaction_year'] = df2_cleaned['trans_date_trans_time'].dt.year
df2_cleaned['transaction_dayofweek'] = df2_cleaned['trans_date_trans_time'].dt.dayofweek

# X 데이터 분리 (df2)
X_test = df2_encoded.drop(columns=['is_fraud', 'trans_date_trans_time', 'transaction_date'

# 수치형 열만 선택 (df2)
X_test_numeric = X_test.select_dtypes(include=['float64', 'int64'])
```

	cc_num	amt	zip	lat	long	city_pop	unix_time	merch_lat	merch_long
0	2703186189652095	-0.407826	28654	-0.484420	0.657620	-0.282589	1325376018	36.011293	-82.048315
1	630423337322	0.230039	99160	2.039120	-2.033870	-0.293670	1325376044	49.159047	-118.186462
2	38859492057661	0.934149	83252	0.717754	-1.601537	-0.280406	1325376051	43.150704	-112.154481
3	3534093764340240	-0.158132	59632	1.515617	-1.590766	-0.287742	1325376076	47.034331	-112.561071
4	375534208663984	-0.177094	24433	-0.023035	0.782279	-0.293835	1325376186	38.674999	-78.632459
...	...	...	...	...	...	...	...	...	...
2578333	60427851591	-0.104104	73624	-0.628161	-0.651328	-0.287070	1334553134	35.943771	-99.285763
2578334	2242542703101233	0.286575	40077	-0.008968	0.346966	-0.292295	1337998863	38.337766	-85.365101
2578335	3519607465576254	0.726324	95537	0.431297	-2.470448	-0.293249	1334022860	40.414796	-124.571076
2578336	4481172224716138496	5.390737	28119	-0.726293	0.743061	-0.284887	1342129753	34.972175	-79.350016
2578337	3542826960473004	0.485086	53559	0.911378	0.083656	-0.274084	1358934145	42.559753	-89.114198

- 날짜와 시간 부분을 활용하기 위해서 새로운 피처를 생성하였습니다.
- 이후에 수치 위주의 분석을 위해서 수치형 변수로 활용할 열을 가져오고 이에 대해서 X에는 특성을 나타내는 값들을, Y에는 사기 여부 값을 넣고 수치형 데이터들을 X에 넣었다.

4.2 인코딩(One-Hot Encoding)

```
# One-Hot Encoding (df1)
df1_encoded = pd.get_dummies(df1_cleaned, columns=['category', 'job'], drop_first=True)

# One-Hot Encoding (df2)
df2_encoded = pd.get_dummies(df2_cleaned, columns=['category', 'job'], drop_first=True)
```

- 인코딩 방식에 있어서는 category, job과 같은 카테고리형 변수(명목형 데이터)에 맞게 Label Encoding과 같이 순서나 크기에 대한 의미를 크게 두지 않는 One-Hot Encoding을 사용하였다. 이를 통해 카테고리형 변수를 이진 벡터로 변환하여 데이터의 순

서 의미를 배제하여 인코딩하였다.

4.3 학습, 테스트 데이터 분리(Train & Test Split)

- 데이터 셋 (Train = df1, Test = df2)

```
from imblearn.over_sampling import SMOTE


# SMOTE 적용 (df1)
smote = SMOTE()
X_smote, y_smote = smote.fit_resample(X_numeric, y)

# 결과 확인 (df1)
print(X_smote.shape, y_smote.shape)

# --------- df2에 대해 동일한 전처리 진행 (SMOTE는 제외) ---------

# 결과 확인 (df2)
print(X_test_numeric.shape)
```

기존 데이터 셋에 있던 Train data set과 Test data set을 그대로 사용하였다. 다만 사기 여부를 감지하는 과정에서 Train data set에서 사기 거래 비율이 0.58%로 낮다는 점으로 인하여 Smote를 적용하여 그 차이를 사기 거래에 대한 충분한 머신러닝을 하려 했다.

```
fraud_count = y_smote.sum()
print(f"사기(1)인 데이터의 개수: {fraud_count}")

non_fraud_count = (y_smote == 0).sum()
print(f"사기 아닌 (0) 데이터의 개수: {non_fraud_count}")
```

```
사기(1)인 데이터의 개수: 1289169
사기 아닌 (0) 데이터의 개수: 1289169
```

이 과정에서 Train data set의 크기가 약 2배로 늘어났고 결과적으로 Train data set과 test data set의 비율이 2578338 : 555719 ⇒ 대략적으로 8.8 : 1.2 혹은 9 : 1로 볼 수 있다. 이에 대해서 데이터의 절대적인 수가 많기 때문에 모델 성능 평가에 충분하다 판단하여 이대로 진행하였다.

4.4 스케일링(Scaling)

- 스탠다드 스케일링(StandardScaler)을 사용

```
from sklearn.preprocessing import StandardScaler

# StandardScaler 적용 (df1)
scaler = StandardScaler()
X_numeric[numeric_cols] = scaler.fit_transform(X_numeric[numeric_cols])

# df1에서 fit한 scaler로 변환 (df2)
X_test_numeric[numeric_cols] = scaler.transform(X_test_numeric[numeric_cols])
```

- 피처 스케일링 방식으로는 MinMaxScaler와 StandardScaler 중에서 StandardScaler를 선택하여 사용하였다. 거래 금액의 변차가 큰 거래가 있어서 MinMaxScaler와 같이 데이터의 최대값과 최소값에 민감한 스케일링과 맞지 않는다고 판단하였고, StandardScaler은 모델 학습할 때 가중치를 동일하게 반영해 적합하다고 판단했다. StandardScaler를 통해서 수치형 피처를 평균 0, 표준편차 1로 변환하여 모델 학습 시 데이터 분포를 반영하였다.

팀원3

4.1 피처 엔지니어링(Feature Engineering)

- OverSampling, Random undersampling, SMOTETomek에 적용된 피처 엔지니어링 변수들
  - trans_date_trans_time 변수에서 year, month, day, hour 추출

```
# 1. trans_date_trans_time 변수를 datetime 형식으로 변환 (errors='coerce' 추가)
df_cleaned['trans_date_trans_time'] = pd.to_datetime(df_cleaned['trans_date_trans_time'

# 2. trans_date_trans_time 변수에서 연도, 월, 일, 시간 추출
df_cleaned['year'] = df_cleaned['trans_date_trans_time'].dt.year
df_cleaned['month'] = df_cleaned['trans_date_trans_time'].dt.month
df_cleaned['day'] = df_cleaned['trans_date_trans_time'].dt.day
df_cleaned['hour'] = df_cleaned['trans_date_trans_time'].dt.hour

# 3. 인코딩 후 원래 trans_date_trans_time 열을 제거
df_cleaned = df_cleaned.drop(columns=['trans_date_trans_time'])
```

  - dob변수에서 age_encoded 추출

```
#  1. dob 변수를 나이로 변환 (2024년을 기준으로 나이 계산)
df_cleaned['dob'] = pd.to_datetime(df_cleaned['dob'])
df_cleaned['age'] = 2024 - df_cleaned['dob'].dt.year

# 2. 나이를 범주로 구분 (예: 연령대별로 나누기)
df_cleaned['age_category'] = pd.cut(df_cleaned['age'],
                                    bins=[0, 18, 30, 50, 100],
                                    labels=['Teen', 'Young Adult', 'Adult', 'Senior'])

# 3. age_category에 맞춰 직접 인코딩 값 할당
age_mapping = {'Teen': 1, 'Young Adult': 2, 'Adult': 3, 'Senior': 4}
df_cleaned['age_encoded'] = df_cleaned['age_category'].map(age_mapping)
```

- amt변수에서 amt_encoded 추출

```
# 1. 구간 설정 시 amt_max 값이 10000보다 큰지 확인하여 구간 설정
if amt_max <= 10000:
    bins = [0, 100, 500, 1000, 5000, 10000]
    labels = ['Very Low', 'Low', 'Medium', 'High', 'Very High']
else:
    bins = [0, 100, 500, 1000, 5000, 10000, amt_max + 1]  # amt_max가 10000보다 클 때 추가
    labels = ['Very Low', 'Low', 'Medium', 'High', 'Very High', 'Ultra']

# 2. 거래 금액을 구간별로 범주화 (비닝)
df_cleaned['amt_category'] = pd.cut(df_cleaned['amt'], bins=bins, labels=labels)

# 3. 레이블을 1, 2, 3, 4, 5, 6에 맞춰서 직접 할당
label_mapping = {'Very Low': 1, 'Low': 2, 'Medium': 3, 'High': 4, 'Very High': 5, 'Ultr

# 4. amt_category 값을 amt_encoded로 변환
df_cleaned['amt_encoded'] = df_cleaned['amt_category'].map(label_mapping)
```

- city_pop변수에서 city_pop_encoded 추출

```
# 1. 구간 설정 시 city_pop_max 값에 따라 구간 설정
if city_pop_max <= 500000:
    bins = [0, 10000, 50000, 100000, 500000]
    labels = ['Very Small', 'Small', 'Medium', 'Large']  # bins가 5개면 labels는 4개여야 힘
else:
    bins = [0, 10000, 50000, 100000, 500000, city_pop_max + 1]  # city_pop_max가 500000보
    labels = ['Very Small', 'Small', 'Medium', 'Large', 'Very Large']  # bins가 6개면 lab

# 2. city_pop 변수를 구간별로 범주화 (비닝)
df_cleaned['city_pop_category'] = pd.cut(df_cleaned['city_pop'], bins=bins, labels=labe

# 3. 레이블을 1, 2, 3, 4, 5, 6에 맞춰서 직접 할당
label_mapping = {'Very Small': 1, 'Small': 2, 'Medium': 3, 'Large': 4, 'Very Large': 5}

# 4. city_pop_category 값을 city_pop_encoded로 변환
df_cleaned['city_pop_encoded'] = df_cleaned['city_pop_category'].map(label_mapping)
```

머신러닝모델(사기거래예측)에 정확성을 높이기 위해 이와 관련되어 영향을 줄 수 있을거 같은 변수들
(trans_date_trans_time,dob,amt,city_pop)을 종합해 피처엔지니어링을 진행하였다.

4.2 인코딩(Encoding)

- OverSampling, Random undersampling, SMOTETomek에 적용된 인코딩 변수들
  - category변수에서 category_encoded 추출

```
# 1. LabelEncoder 객체 생성
label_encoder = LabelEncoder()

# 2. category 변수 레이블 인코딩 적용
df_cleaned['category_encoded'] = label_encoder.fit_transform(df_cleaned['category']) +

# 3. category와 category_encoded의 고유 값만 추출 후 오름차순 정렬
category_mapping = pd.DataFrame({
    'category': df_cleaned['category'],
    'category_encoded': df_cleaned['category_encoded']
}).drop_duplicates().sort_values(by='category_encoded')
```

  - city변수에서 city_encoded 추출

```
# 1. LabelEncoder 객체 생성
label_encoder = LabelEncoder()

# 2. city 변수 레이블 인코딩 적용
df_cleaned['city_encoded'] = label_encoder.fit_transform(df_cleaned['city']) + 1  # 1부

# 3. city와 city_encoded의 고유 값만 추출 후 오름차순 정렬
city_mapping = pd.DataFrame({
    'city': df_cleaned['city'],
    'city_encoded': df_cleaned['city_encoded']
}).drop_duplicates().sort_values(by='city_encoded')
```

- state변수에서 state_encoded추출

```
from sklearn.preprocessing import LabelEncoder
import pandas as pd

# 1. LabelEncoder 객체 생성
label_encoder = LabelEncoder()

# 2. state 변수 레이블 인코딩 적용
df_cleaned['state_encoded'] = label_encoder.fit_transform(df_cleaned['state']) + 1  # 1

# 3. state와 state_encoded의 고유 값만 추출 후 오름차순 정렬
state_mapping = pd.DataFrame({
    'state': df_cleaned['state'],
    'state_encoded': df_cleaned['state_encoded']
}).drop_duplicates().sort_values(by='state_encoded')
```

- job변수에서 job_encoded추출

```
from sklearn.preprocessing import LabelEncoder

# LabelEncoder 객체 생성
label_encoder = LabelEncoder()

# job 변수에 레이블 인코딩 적용
df_cleaned['job_encoded'] = label_encoder.fit_transform(df_cleaned['job'])+1

# job_encoded 열을 기준으로 오름차순 정렬
df_cleaned_sorted = df_cleaned[['job', 'job_encoded']].sort_values(by='job_encoded')
```

- 피처엔지니어링, 인코딩된 변수들로 데이터프레임 생성

```
df_encoded = df_cleaned[['is_fraud','year', 'month', 'day', 'hour', 'category_encoded', 'a
                         'state_encoded', 'job_encoded', 'age_encoded']]
```

머신러닝모델(사기거래예측)에 정확성을 높이기 위해 이와 관련되어 영향을 줄 수 있을거 같은 변수들(category,city,state,job)을 종합해 인코딩을 진행하였고 앞서 진행한 피처엔지니어링과 인코딩한 변수들로 새로운 데이터 프레임을 생성하였다.

4.3 학습, 테스트 데이터 분리(Train & Test Split)

```
from sklearn.model_selection import train_test_split

# df_encoded를 8:2로 분할
train_df, test_df = train_test_split(df_encoded, test_size=0.2, random_state=42)

# 분할된 데이터프레임의 크기 확인
```

```
print(f"훈련 데이터 행 수: {train_df.shape[0]}")
print(f"테스트 데이터 행 수: {test_df.shape[0]}")
```

새로운 데이터 프레임으로 학습,테스트 데이터 분리를 하였다. 비율은 훈련:학습 8:2 비율로 진행하였고 진행한 후 훈련데이터 행 수는 877851개, 테스트데이터 행 수는 219463개가 나왔다.

4.4 스케일링(Scaling)

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# 1. PCA를 적용하기 전에 데이터 스케일링 (표준화)
scaler = StandardScaler()
train_scaled = scaler.fit_transform(train_df)
test_scaled = scaler.transform(test_df)

# 2. PCA 객체 생성 (예: 2차원으로 축소)
pca = PCA(n_components=2)

# 3. 훈련 데이터와 테스트 데이터에 PCA 적용
train_pca = pca.fit_transform(train_scaled)
test_pca = pca.transform(test_scaled)
```

데이터 분포 정규화와 모델 학습성능을 개선시키기 위해 스탠다드스케일링과 PCA를 진행하였다.

5. 예측 모델 구현(Modeling) 및 예측(Predict)

팀원1

- **No Under/Oversampling** 모델 구현 및 예측
  - 교차 검증(Cross Validation)

```
# 교차 검증 객체 생성
from sklearn.model_selection import StratifiedKFold, cross_val_score
from sklearn.ensemble import RandomForestClassifier

# n_splits=5 로 하여 5개의 Fold 설정
kf = StratifiedKFold(n_splits=5, shuffle=False)

# 이진분류 모델로 RandomForestClassifier 를 선택, n_jobs=-1 로 하여 가능한 cpu 모두를 사용
rf = RandomForestClassifier(n_estimators=100, random_state=23, n_jobs=-1)

# scoring='recall' 로 설정하여 교차검증을 재현율(recall)중심으로 실행
score = cross_val_score(rf, X_train_scaled, y_train, cv=kf, scoring='recall')

print("Cross Validation Recall scores are: {}".format(score))
print("Average Cross Validation Recall score: {}".format(score.mean()))

# Cross Validation Recall scores are: [0.76074407 0.76972418 0.74166667 0.76346154 0
# Average Cross Validation Recall score: 0.7545551882370356
```

    - 교차 검증을 위하여 StratifiedFold 를 사용하여 5개의 Fold 를 생성하여 RandomForest 를 구동하였다.

- cross_val_score 를 사용하여 재현율(Recall) 을 중심으로 5개의 점수를 추출하고, 평균을 구했다.

- 하이퍼파라미터 튜닝(GridSearchCV)

```
from sklearn.model_selection import GridSearchCV

# 선정해야 할 parameter 후보 생성
params = {
    'n_estimators': [50, 100, 200],
    'max_depth': [4, 6, 10, 12],
    'random_state': [23]
}

# 모델 학습
grid_rf = GridSearchCV(rf,
                       param_grid=params,
                       cv=kf,
                       scoring='recall').fit(X_train_scaled, y_train)

print('Best parameters:', grid_rf.best_params_)
print('Best score:', grid_rf.best_score_)

# Best parameters: {'max_depth': 12, 'n_estimators': 100, 'random_state': 23}
# Best score: 0.6520914129701814
```

  - 하이퍼파라미터 튜닝을 위하여 최적의 값들을 찾기 위하여 GridSearchCV 를 사용하여 'max_depth': 12, 'n_estimators': 100 를 찾았으며 scoring='recall' 로 설정하여 가장 좋은 점수가 약 재현율(Recall) = 0.65 임을 알게 되었다.

- 모델 예측 및 성능평가

```
# 모델 예측
y_pred = grid_rf.predict(X_test_scaled)

# 성능 평가
from sklearn.metrics import confusion_matrix, recall_score, precision_score, f1_scor
cm = confusion_matrix(y_test, y_pred)

rf_Recall = recall_score(y_test, y_pred)
rf_Precision = precision_score(y_test, y_pred)
rf_f1 = f1_score(y_test, y_pred)
rf_accuracy = accuracy_score(y_test, y_pred)

print(cm)

# [[368585     41]
#  [   644   1209]]

classificationReport = classification_report(y_test, y_pred, target_names=["non_frau
print(classificationReport)

#               precision    recall  f1-score   support
#
#    non_fraud       1.00      1.00      1.00    368626
```

```
#        fraud       0.97      0.65      0.78      1853
#
#     accuracy                           1.00    370479
#    macro avg       0.98      0.83      0.89    370479
# weighted avg       1.00      1.00      1.00    370479

ndf = [(rf_Recall, rf_Precision, rf_f1, rf_accuracy)]

rf_score = pd.DataFrame(data = ndf, columns=['Recall','Precision','F1 Score', 'Accur
rf_score.insert(0, 'Random Forest with', 'No Under/Oversampling')
rf_score
```

	Random Forest with	Recall	Precision	F1 Score	Accuracy
0	No Under/Oversampling	0.652455	0.9672	0.779246	0.998151

- 모델을 예측(.predict)하고 confusion matrix 를 사용해서 Recall, Precision, F1 Score, Accuracy 를 도출하였다.
- 재현율(Recall) 을 기준으로 파라미터를 튜닝하고 학습, 예측 하였지만 재현율(Recall)은 정확도(Precision) 에 비해 다소 낮다.

- **Random Undersampling** 모델 구현 및 예측
  - (정상거래 데이터 축소)Undersampling

```
from imblearn.under_sampling import RandomUnderSampler
# undersampling 생성
rus = RandomUnderSampler(random_state=23)

# 리샘플링하여 X_under, y_under 생성
X_under, y_under = rus.fit_resample(X_train_scaled, y_train)

print('non-Frauds:', y_under.value_counts()[0], '/', round(y_under.value_counts()[0]
print('Frauds:', y_under.value_counts()[1], '/',round(y_under.value_counts()[1]/len(

# non-Frauds: 7798 / 50.0 % of the dataset
# Frauds: 7798 / 50.0 % of the dataset
```

- 정상거래 수를 사기거래 수와 동일하게 7798 개로 리샘플링 하였다.

  - 파이프라인(Pipeline)을 통한 교차 검증(Cross Validation)

```
from imblearn.pipeline import Pipeline, make_pipeline

random_unders_pipeline = make_pipeline(
                                                        RandomUnderS
                                    RandomForestClassifier(n_estimators=100,
                                    random_state=23,
                                    n_jobs=-1))

score2 = cross_val_score(random_unders_pipeline, X_train_scaled, y_train, scoring='r
print("Cross Validation Recall Scores are: {}".format(score2))
print("Average Cross Validation Recall score: {}".format(score2.mean()))
```

```
# Cross Validation Recall Scores are: [0.96792816 0.96151379 0.97115385 0.96474359 0
# Average Cross Validation Recall score: 0.9657601848653805
```

- 교차 검증을 위하여 RandomForest 를 파이프라인(Pipeline)을 사용하여 구동하였다.
- cross_val_score 를 사용하여 재현율(Recall) 을 중심으로 5개의 점수를 추출하고, StratifiedFold 를 사용하여 5개의 Fold 를 생성하고 평균을 구했다.

- 하이퍼파라미터 튜닝(GridSearchCV)

```
# params 생성
new_params = {'randomforestclassifier__' + key: params[key] for key in params}

# 모델 생성
grid_under_rf = GridSearchCV(random_unders_pipeline,
                             param_grid=new_params,
                             cv=kf,
                             scoring='recall',
                             return_train_score=True,
                             n_jobs=-1)
# 모델 학습
grid_under_rf.fit(X_train_scaled, y_train)

print('Best parameters:', grid_under_rf.best_params_)
print('Best score:', grid_under_rf.best_score_)

# Best parameters: {'randomforestclassifier__max_depth': 12, 'randomforestclassifier
# Best score: 0.961399976974063
```

- 하이퍼파라미터 튜닝을 위하여 최적의 값들을 찾기 위하여 GridSearchCV 를 사용하여 'max_depth': 12, 'n_estimators': 200 를 찾았으며 scoring='recall' 로 설정하여 가장 좋은 점수가 약 재현율(Recall) = 0.96 임을 알게 되었다.

- 모델 예측 및 성능평가

```
y_pred = grid_under_rf.best_estimator_.named_steps['randomforestclassifier'].predict

cm = confusion_matrix(y_test, y_pred)

under_rf_Recall = recall_score(y_test, y_pred)
under_rf_Precision = precision_score(y_test, y_pred)
under_rf_f1 = f1_score(y_test, y_pred)
under_rf_accuracy = accuracy_score(y_test, y_pred)

print(cm)

# [[358556  10070]
#  [    59   1794]]

classificationReport = classification_report(y_test, y_pred, target_names=["non_frau
print(classificationReport)

#               precision    recall  f1-score   support
#
#    non_fraud       1.00      0.97      0.99    368626
```

```
#        fraud       0.15      0.97      0.26      1853
#
#     accuracy                           0.97    370479
#    macro avg       0.58      0.97      0.62    370479
# weighted avg       1.00      0.97      0.98    370479

ndf = [(under_rf_Recall, under_rf_Precision, under_rf_f1, under_rf_accuracy)]

under_rf_score = pd.DataFrame(data = ndf, columns=['Recall','Precision','F1 Score',
under_rf_score.insert(0, 'Random Forest with', 'Random Undersampling')
under_rf_score
```

	Random Forest with	Recall	Precision	F1 Score	Accuracy
0	Random Undersampling	0.96816	0.151214	0.261573	0.97266

- **SMOTE Oversampling** 모델 구현 및 예측
  - SMOTE(Synthetic Minority Oversampling Technique) 는 신용카드 사기거래와 같이 사기거래의 데이터양이 정상거래에 비해서 현저하게 작은 양을 가질 때 작은양의 데이터를 오버샘플링하여 학습시키는 이론이자 imblearn 의 라이브러이다.
  - 파이프라인(Pipeline)을 통한 교차 검증(Cross Validation)

```
from imblearn.over_sampling import SMOTE

smote_pipeline = make_pipeline(
                                                    SMOTE(random_state=23),
                              RandomForestClassifier(n_estimators=100,
                              random_state=23,
                              n_jobs=-1))

score3 = cross_val_score(smote_pipeline, X_train_scaled, y_train, scoring='recall',
print("Cross Validation Recall Scores are: {}".format(score3))
print("Average Cross Validation Recall score: {}".format(score3.mean()))

# Cross Validation Recall Scores are: [0.82168056 0.82809493 0.80448718 0.82371795 0
# Average Cross Validation Recall score: 0.8170063814739891
```

    - 교차 검증을 위하여 SMOTE 를 적용, RandomForest 를 파이프라인(Pipeline)을 사용하여 구동하였다.
    - cross_val_score 를 사용하여 재현율(Recall) 을 중심으로 5개의 점수를 추출하고, StratifiedFold 를 사용하여 5개의 Fold 를 생성하고 평균을 구했다.

  - 하이퍼파라미터 튜닝(GridSearchCV)

```
new_params = {'randomforestclassifier__' + key: params[key] for key in params}
smote_rf = GridSearchCV(smote_pipeline,
                        param_grid=new_params,
                        cv=kf,
                        scoring='recall',
                        return_train_score=True
                        )
smote_rf.fit(X_train_scaled, y_train)

print('Best parameters:', smote_rf.best_params_)
```

```
print('Best score:', smote_rf.best_score_)

# Best parameters: {'randomforestclassifier__max_depth': 10, 'randomforestclassifier
# Best score: 0.9170312165918324
```

- 하이퍼파라미터 튜닝을 위하여 최적의 값들을 찾기 위하여 GridSearchCV 를 사용하여 'max_depth': 10, 'n_estimators': 100 를 찾았으며 scoring='recall' 로 설정하여 가장 좋은 점수가 약 재현율(Recall) = 0.92 임을 알게 되었다.

◦ 모델 예측 및 성능평가

```
y_pred = smote_rf.best_estimator_.named_steps['randomforestclassifier'].predict(X_te

cm = confusion_matrix(y_test, y_pred)

smote_rf_Recall = recall_score(y_test, y_pred)
smote_rf_Precision = precision_score(y_test, y_pred)
smote_rf_f1 = f1_score(y_test, y_pred)
smote_rf_accuracy = accuracy_score(y_test, y_pred)

print(cm)

# [[361320   7306]
#  [   124   1729]]

classificationReport = classification_report(y_test, y_pred, target_names=["non_frau
print(classificationReport)

#               precision    recall  f1-score   support
#
#    non_fraud       1.00      0.98      0.99    368626
#        fraud       0.19      0.93      0.32      1853
#
#     accuracy                           0.98    370479
#    macro avg       0.60      0.96      0.65    370479
# weighted avg       1.00      0.98      0.99    370479

ndf = [(smote_rf_Recall, smote_rf_Precision, smote_rf_f1, smote_rf_accuracy)]

smote_rf_score = pd.DataFrame(data = ndf, columns=['Recall','Precision','F1 Score',
smote_rf_score.insert(0, 'Random Forest with', 'SMOTE Oversampling')
smote_rf_score
```

	Random Forest with	Recall	Precision	F1 Score	Accuracy
0	SMOTE Oversampling	0.933081	0.191367	0.317597	0.979945

- **Undersampling using Tomek Links** 샘플링
  - Tomek 은 1976년 Ivan Tomek 에 의해 고안된 under-sampling 테크닉이다. 근접한(CNN) 데이터에서 유클리디어 기법을 통하여 다수의 데이터를 제거하여 다수의 데이터를 적게 샘플링하는 테크닉이다.
  - Tomek 을 통한 리샘플링과 학습

```
from imblearn.under_sampling import TomekLinks

# undersampling 정의
tomekU = TomekLinks(sampling_strategy='auto', n_jobs=-1)
# tomekU = TomekLinks()

# 모델 리샘플링
X_underT, y_underT = tomekU.fit_resample(X_train_scaled, y_train)

print('non-Frauds:', y_underT.value_counts()[0], '/', round(y_underT.value_counts()[
print('Frauds:', y_underT.value_counts()[1], '/',round(y_underT.value_counts()[1]/le

# non-Frauds: 1473429 / 99.47 % of the dataset
# Frauds: 7798 / 0.53 % of the dataset
```

- Tomek 의 테크닉은 데이터를 감소시키는 기법으로 샘플링만 진행해보고, 본 프로젝트에서는 학습 및 모델링을 진행하지 않았다.

- **Class Weights** 모델 구현 및 예측
  - 교차 검증(Cross Validation)

```
# class_weight='balanced' 로 세팅하여
rfb = RandomForestClassifier(n_estimators=100, random_state=23, class_weight="balanc

score5 = cross_val_score(rfb, X_train_scaled, y_train, cv=kf, scoring='recall')
print("Cross Validation Recall scores are: {}".format(score5))
print("Average Cross Validation Recall score: {}".format(score5.mean()))

# Cross Validation Recall scores are: [0.75817832 0.76844131 0.73717949 0.76217949 0
# Average Cross Validation Recall score: 0.7523752076446111
```

    - 교차 검증을 위하여 StratifiedFold 를 사용하여 5개의 Fold 를 생성하여 RandomForest 를 구동하였다.
    - cross_val_score 를 사용하여 재현율(Recall) 을 중심으로 5개의 점수를 추출하고, 평균을 구했다.
  - 하이퍼파라미터 튜닝(GridSearchCV)

```
grid_rfb = GridSearchCV(rfb,
                        param_grid=params,
                        cv=kf,
                        scoring='recall').fit(X_train_scaled, y_train)
```

    - 하이퍼파라미터 튜닝을 위하여 최적의 값들을 찾기 위하여 GridSearchCV 를 사용하였다.
  - 모델 예측 및 성능평가

```
y_pred = grid_rfb.predict(X_test_scaled)

cm = confusion_matrix(y_test, y_pred)

grid_rfb_Recall = recall_score(y_test, y_pred)
grid_rfb_Precision = precision_score(y_test, y_pred)
grid_rfb_f1 = f1_score(y_test, y_pred)
grid_rfb_accuracy = accuracy_score(y_test, y_pred)

print(cm)
```

```
# [[362009   6617]
#  [   119   1734]]

classificationReport = classification_report(y_test, y_pred, target_names=["non_frau
print(classificationReport)

#               precision    recall  f1-score   support
#
#    non_fraud       1.00      0.98      0.99    368626
#        fraud       0.21      0.94      0.34      1853
#
#     accuracy                           0.98    370479
#    macro avg       0.60      0.96      0.67    370479
# weighted avg       1.00      0.98      0.99    370479

ndf = [(grid_rfb_Recall, grid_rfb_Precision, grid_rfb_f1, grid_rfb_accuracy)]

grid_rfb_score = pd.DataFrame(data = ndf, columns=['Recall','Precision','F1 Score',
grid_rfb_score.insert(0, 'Random Forest with', 'Class weights')
grid_rfb_score
```

	Random Forest with	Recall	Precision	F1 Score	Accuracy
0	Class weights	0.93578	0.20764	0.339867	0.981818

팀원2

- Smote_RandomForest/XgBoost Model

```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.metrics import confusion_matrix, classification_report

# RandomForest 및 XGBoost 모델 학습 및 테스트
def train_and_evaluate(X_train, y_train, X_test, y_test):
    # RandomForestClassifier (병렬 처리 사용, n_jobs=-1)
    rf_model = RandomForestClassifier(random_state=42, n_jobs=-1)
    rf_model.fit(X_train, y_train)
    rf_pred = rf_model.predict(X_test)

    print("Random Forest 성능 평가:")
    print("Confusion Matrix:\n", confusion_matrix(y_test, rf_pred))
    print("\nClassification Report:\n", classification_report(y_test, rf_pred))

    # XGBoostClassifier (병렬 처리 사용, n_jobs=-1)
    xgb_model = XGBClassifier(random_state=42, use_label_encoder=False, eval_metric='lo
    xgb_model.fit(X_train, y_train)
    xgb_pred = xgb_model.predict(X_test)

    print("XGBoost 성능 평가:")
    print("Confusion Matrix:\n", confusion_matrix(y_test, xgb_pred))
    print("\nClassification Report:\n", classification_report(y_test, xgb_pred))
```

```
# X_train, y_train 데이터셋 (df1)
X_train = X_smote  # SMOTE 적용된 X 데이터
y_train = y_smote  # SMOTE 적용된 y 데이터

# X_test, y_test 데이터셋 (df2)
X_test = X_test_numeric  # 전처리된 df2 데이터
y_test = df2_encoded['is_fraud']  # df2의 실제 라벨

# 모델 학습 및 평가
train_and_evaluate(X_train, y_train, X_test, y_test)
```

- x,y 학습 데이터 셋으로 Smote로 오버샘플링 된 데이터를 사용하여 Random Forest와 XGBoost 모델링 방법을 선택하여 df2 데이터를 예측값을 만들기로 라다. 여기서 두 모델링을 선택한 이유는 다음과 같다. 랜텀 포레스트 같은 경우 여러 개의 의사결정 트리를 결합해 예측 성능을 향상시켜 칼럼이 다양할 수록 유리하다 생각하였고, 데이터 불균현 문제에 있어도 유리하다 생각하여 선택하였다. XGBoost의 경우에는 사기 예측에 자주 사용되며 데이터 셋 크기에 맞춰 적용할 수 있다는 점에서 선택하였다.

- Smote_RandomForest/XgBoost_Hyperparameter Model

```
# 하이퍼파라미터 튜닝(Grid Search)을 통한 개선
import numpy as np
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier

# Random Forest 하이퍼파라미터 튜닝
rf_param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'class_weight': ['balanced', None]  # 클래스 불균형을 고려
}

rf_grid_search = GridSearchCV(RandomForestClassifier(random_state=42, n_jobs=-1),
                              param_grid=rf_param_grid,
                              scoring='f1',  # f1-score 사용
                              cv=5,  # 5-fold cross-validation
                              verbose=2,
                              n_jobs=-1)

rf_grid_search.fit(X_train, y_train)

# 최적의 하이퍼파라미터와 성능 평가
print("최적의 하이퍼파라미터 (Random Forest):", rf_grid_search.best_params_)
print("최고 F1-Score (Random Forest):", rf_grid_search.best_score_)

# XGBoost 하이퍼파라미터 튜닝
xgb_param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [3, 5, 7],
    'learning_rate': [0.01, 0.1, 0.2],
    'subsample': [0.5, 0.7, 1.0],
    'colsample_bytree': [0.5, 0.7, 1.0],
    'class_weight': ['balanced', None]  # 클래스 불균형을 고려
}
```

```
xgb_grid_search = GridSearchCV(XGBClassifier(random_state=42, use_label_encoder=False,
                               param_grid=xgb_param_grid,
                               scoring='f1',  # f1-score 사용
                               cv=5,  # 5-fold cross-validation
                               verbose=2,
                               n_jobs=-1)

xgb_grid_search.fit(X_train, y_train)

# 최적의 하이퍼파라미터와 성능 평가
print("최적의 하이퍼파라미터 (XGBoost):", xgb_grid_search.best_params_)
print("최고 F1-Score (XGBoost):", xgb_grid_search.best_score_)
```

- 위의 모델을 발전시키기 위해 모델의 하이퍼 파라미터를 조정하여 성능을 최적화하는 코드를 만들었고, 이를 통해서 모델의 성능을 향상시킴과 동시에 더 나은 성능 평가 결과를 얻을 수 있을 것이라 예측했다.

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint, uniform

# Random Forest 하이퍼파라미터 범위 축소 및 무작위 탐색
rf_param_dist = {
    'n_estimators': randint(100, 300),
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': randint(2, 10),
    'class_weight': ['balanced', None]
}

rf_random_search = RandomizedSearchCV(RandomForestClassifier(random_state=42, n_jobs=-1
                                      param_distributions=rf_param_dist,
                                      n_iter=10,  # 무작위로 20번 시도
                                      scoring='f1',
                                      cv=3,  # 3-fold 교차 검증
                                      verbose=2,
                                      n_jobs=-1)

rf_random_search.fit(X_train, y_train)

print("최적의 하이퍼파라미터 (Random Forest):", rf_random_search.best_params_)
print("최고 F1-Score (Random Forest):", rf_random_search.best_score_)

# XGBoost 하이퍼파라미터 범위 축소 및 무작위 탐색
xgb_param_dist = {
    'n_estimators': randint(100, 300),
    'max_depth': randint(3, 7),
    'learning_rate': uniform(0.01, 0.2),
    'subsample': uniform(0.5, 1.0),
    'colsample_bytree': uniform(0.5, 1.0)
}

xgb_random_search = RandomizedSearchCV(XGBClassifier(random_state=42, use_label_encoder
                                       param_distributions=xgb_param_dist,
                                       n_iter=20,  # 무작위로 20번 시도
                                       scoring='f1',
                                       cv=3,
                                       verbose=2,
```

```
                                    n_jobs=-1)

xgb_random_search.fit(X_train, y_train)

print("최적의 하이퍼파라미터 (XGBoost):", xgb_random_search.best_params_)
print("최고 F1-Score (XGBoost):", xgb_random_search.best_score_)
```

- 위의 grid_search 방식과 cv = 5라는 설정은 캐글 노트북 환경에서는 시간이 너무 오래 걸려 결과를 확인하기 어려웠기에 이후에 시간 단축을 위해서 random_search 방식을 활용하였고 n_iter과 cv를 축소하였다. 그 결과는 아래 주석과 같다.

```
# Fitting 3 folds for each of 10 candidates, totalling 20 fits
# [CV] END class_weight=balanced, max_depth=10, min_samples_split=4, n_estimators=14
# [CV] END class_weight=balanced, max_depth=10, min_samples_split=2, n_estimators=19
# [CV] END class_weight=balanced, max_depth=20, min_samples_split=5, n_estimators=15
# [CV] END class_weight=balanced, max_depth=20, min_samples_split=8, n_estimators=24
# [CV] END class_weight=balanced, max_depth=10, min_samples_split=4, n_estimators=14
# [CV] END class_weight=balanced, max_depth=10, min_samples_split=2, n_estimators=19
# [CV] END class_weight=balanced, max_depth=20, min_samples_split=5, n_estimators=15
# [CV] END class_weight=balanced, max_depth=20, min_samples_split=8, n_estimators=24
# [CV] END class_weight=balanced, max_depth=20, min_samples_split=9, n_estimators=29
# [CV] END class_weight=balanced, max_depth=20, min_samples_split=9, n_estimators=29
# [CV] END class_weight=None, max_depth=30, min_samples_split=7, n_estimators=179; t
# [CV] END class_weight=balanced, max_depth=20, min_samples_split=5, n_estimators=29
# [CV] END class_weight=balanced, max_depth=None, min_samples_split=9, n_estimators=
# [CV] END class_weight=balanced, max_depth=30, min_samples_split=7, n_estimators=16
# [CV] END class_weight=balanced, max_depth=30, min_samples_split=6, n_estimators=23
# [CV] END class_weight=None, max_depth=30, min_samples_split=7, n_estimators=179; t
# [CV] END class_weight=balanced, max_depth=20, min_samples_split=5, n_estimators=29
# [CV] END class_weight=balanced, max_depth=None, min_samples_split=9, n_estimators=
# [CV] END class_weight=balanced, max_depth=30, min_samples_split=7, n_estimators=16
# [CV] END class_weight=balanced, max_depth=30, min_samples_split=6, n_estimators=23
# 최적의 하이퍼파라미터 (Random Forest): {'class_weight': 'balanced', 'max_depth': None,
# 최고 F1-Score (Random Forest): 0.6662714576933472

# Fitting 3 folds for each of 20 candidates, totalling 60 fits
# [CV] END colsample_bytree=1.331476447381445, learning_rate=0.05205062831980455, ma
# [CV] END colsample_bytree=1.331476447381445, learning_rate=0.05205062831980455, ma
# [CV] END colsample_bytree=0.8046170495009265, learning_rate=0.16706467591103843, m
# [CV] END colsample_bytree=1.368050519359643, learning_rate=0.08207068052219425, ma
# [CV] END colsample_bytree=1.331476447381445, learning_rate=0.05205062831980455, ma
# [CV] END colsample_bytree=0.8046170495009265, learning_rate=0.16706467591103843, m
# [CV] END colsample_bytree=1.368050519359643, learning_rate=0.08207068052219425, ma
# [CV] END colsample_bytree=1.4284930526021637, learning_rate=0.19043335228867095, m
# [CV] END colsample_bytree=0.8046170495009265, learning_rate=0.16706467591103843, m
# [CV] END colsample_bytree=1.368050519359643, learning_rate=0.08207068052219425, ma
# [CV] END colsample_bytree=1.4284930526021637, learning_rate=0.19043335228867095, m
# [CV] END colsample_bytree=1.4284930526021637, learning_rate=0.19043335228867095, m
# [CV] END colsample_bytree=0.6366086066896268, learning_rate=0.054782820793783, max
# [CV] END colsample_bytree=0.6366086066896268, learning_rate=0.054782820793783, max
# [CV] END colsample_bytree=0.6366086066896268, learning_rate=0.054782820793783, max
# [CV] END colsample_bytree=1.195527446734801, learning_rate=0.19756935082973992, ma
# [CV] END colsample_bytree=1.195527446734801, learning_rate=0.19756935082973992, ma
# [CV] END colsample_bytree=1.195527446734801, learning_rate=0.19756935082973992, ma
# [CV] END colsample_bytree=1.2774451270562759, learning_rate=0.09928137004251979, m
# [CV] END colsample_bytree=1.2774451270562759, learning_rate=0.09928137004251979, m
# [CV] END colsample_bytree=1.2774451270562759, learning_rate=0.09928137004251979, m
```

```
# [CV] END colsample_bytree=1.4233200307078724, learning_rate=0.10531116285700982, m
# [CV] END colsample_bytree=1.4233200307078724, learning_rate=0.10531116285700982, m
# [CV] END colsample_bytree=0.609197548201356, learning_rate=0.054003691683521766, m
# [CV] END colsample_bytree=1.4233200307078724, learning_rate=0.10531116285700982, m
# [CV] END colsample_bytree=0.9169062231007818, learning_rate=0.03022791016070145, m
# [CV] END colsample_bytree=0.9169062231007818, learning_rate=0.03022791016070145, m
# [CV] END colsample_bytree=0.9169062231007818, learning_rate=0.03022791016070145, m
# [CV] END colsample_bytree=1.365918889120294, learning_rate=0.14230727532724297, ma
# [CV] END colsample_bytree=1.365918889120294, learning_rate=0.14230727532724297, ma
# [CV] END colsample_bytree=1.365918889120294, learning_rate=0.14230727532724297, ma
# [CV] END colsample_bytree=0.9539016584790856, learning_rate=0.1692443093528138, ma
# [CV] END colsample_bytree=0.9539016584790856, learning_rate=0.1692443093528138, ma
# [CV] END colsample_bytree=0.9539016584790856, learning_rate=0.1692443093528138, ma
# [CV] END colsample_bytree=0.5343164772659842, learning_rate=0.13358038660131966, m
# 최적의 하이퍼파라미터 (XGBoost): {'colsample_bytree': 0.8046170495009265, 'learning_ra
# 최고 F1-Score (XGBoost): 0.6550819172450099
```

팀원3

- Oversampling 모델구현 및 예측

```
from sklearn.preprocessing import StandardScaler

# SMOTE 적용 (훈련 데이터에만 적용)
smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)

# 결과 확인 (클래스 분포)
print(f"오버샘플링 전 클래스 분포:\n{y_train.value_counts()}")
print(f"오버샘플링 후 클래스 분포:\n{y_train_resampled.value_counts()}")

오버샘플링 전 클래스 분포:
is_fraud
0    875111
1      2740
Name: count, dtype: int64
오버샘플링 후 클래스 분포:
is_fraud
0    875111
1    875111
Name: count, dtype: int64
```

- 로지스틱 회귀모델 학습

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, accuracy_score

# 1. 로지스틱 회귀 모델 학습 (균형 조정된 데이터로)
logreg = LogisticRegression(random_state=42)
logreg.fit(X_train_resampled, y_train_resampled)

# 2. 테스트 데이터로 예측
y_pred = logreg.predict(X_test)

# 3. 성능 평가
```

```
print("로지스틱 회귀 성능 평가:")
print(classification_report(y_test, y_pred))
```

- 랜덤 포레스트 모델 학습

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report

# 랜덤 포레스트 모델 학습
rf = RandomForestClassifier(random_state=42)
rf.fit(X_train_resampled, y_train_resampled)

# 테스트 데이터로 예측
y_pred_rf = rf.predict(X_test)

# 성능 평가
print("랜덤 포레스트 성능 평가:")
print(classification_report(y_test, y_pred_rf))
```

- XGBoost 모델 학습

```
import xgboost as xgb
from sklearn.metrics import classification_report, accuracy_score

# 1. XGBoost 모델 학습
xgb_model = xgb.XGBClassifier(random_state=42, enable_categorical=True)
xgb_model.fit(X_train_resampled, y_train_resampled)

# 2. 테스트 데이터로 예측
y_pred_xgb = xgb_model.predict(X_test)

# 3. 성능 평가
print("XGBoost 성능 평가:")
print(classification_report(y_test, y_pred_xgb))
```

- Randomundersampling 모델구현 및 예측

```
from imblearn.under_sampling import RandomUnderSampler

# 1. 언더샘플링 적용
undersampler = RandomUnderSampler(random_state=42)
X_train_resampled, y_train_resampled = undersampler.fit_resample(X_train, y_train)

# 2. 결과 확인 (클래스 분포)
print(f"언더샘플링 전 클래스 분포:\n{y_train.value_counts()}")
print(f"언더샘플링 후 클래스 분포:\n{y_train_resampled.value_counts()}")

언더샘플링 전 클래스 분포:
is_fraud
0    875111
1      2740
Name: count, dtype: int64
언더샘플링 후 클래스 분포:
is_fraud
0    2740
```

```
1    2740
Name: count, dtype: int64
```

- 로지스틱 회귀모델 학습

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, accuracy_score

# 1. 로지스틱 회귀 모델 학습 (균형 조정된 데이터로)
logreg = LogisticRegression(random_state=42)
logreg.fit(X_train_resampled, y_train_resampled)

# 2. 테스트 데이터로 예측
y_pred = logreg.predict(X_test)

# 3. 성능 평가
print("로지스틱 회귀 성능 평가:")
print(classification_report(y_test, y_pred))
```

- 랜덤 포레스트 모델 학습

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report

# 랜덤 포레스트 모델 학습
rf = RandomForestClassifier(random_state=42)
rf.fit(X_train_resampled, y_train_resampled)

# 테스트 데이터로 예측
y_pred_rf = rf.predict(X_test)

# 성능 평가
print("랜덤 포레스트 성능 평가:")
print(classification_report(y_test, y_pred_rf))
```

- XGBoost 모델 학습

```
import xgboost as xgb
from sklearn.metrics import classification_report, accuracy_score

# 1. XGBoost 모델 학습
xgb_model = xgb.XGBClassifier(random_state=42, enable_categorical=True)
xgb_model.fit(X_train_resampled, y_train_resampled)

# 2. 테스트 데이터로 예측
y_pred_xgb = xgb_model.predict(X_test)

# 3. 성능 평가
print("XGBoost 성능 평가:")
print(classification_report(y_test, y_pred_xgb))
```

- SMOTETomek 모델 구현 예측

```
from imblearn.combine import SMOTETomek

# 1. SMOTETomek 적용
```

```
smote_tomek = SMOTETomek(random_state=42)
X_train_resampled, y_train_resampled = smote_tomek.fit_resample(X_train, y_train)

# 2. 결과 확인 (클래스 분포)
print(f"SMOTETomek 적용 전 클래스 분포:\n{y_train.value_counts()}")
print(f"SMOTETomek 적용 후 클래스 분포:\n{y_train_resampled.value_counts()}")

SMOTETomek 적용 전 클래스 분포:
is_fraud
0    875111
1      2740
Name: count, dtype: int64
SMOTETomek 적용 후 클래스 분포:
is_fraud
0    875048
1    875048
Name: count, dtype: int64
```

- 로지스틱 회귀 모델 학습

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, accuracy_score

# 1. 로지스틱 회귀 모델 학습 (균형 조정된 데이터로)
logreg = LogisticRegression(random_state=42)
logreg.fit(X_train_resampled, y_train_resampled)

# 2. 테스트 데이터로 예측
y_pred = logreg.predict(X_test)

# 3. 성능 평가
print("로지스틱 회귀 성능 평가:")
print(classification_report(y_test, y_pred))
print(f"Accuracy: {accuracy_score(y_test, y_pred)}")
```

- 랜덤포레스트 모델 학습

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report

# 랜덤 포레스트 모델 학습
rf = RandomForestClassifier(random_state=42)
rf.fit(X_train_resampled, y_train_resampled)

# 테스트 데이터로 예측
y_pred_rf = rf.predict(X_test)

# 성능 평가
print("랜덤 포레스트 성능 평가:")
print(classification_report(y_test, y_pred_rf))
```

- XGBoost 모델 학습

```
import xgboost as xgb
from sklearn.metrics import classification_report, accuracy_score
```

```
# 1. XGBoost 모델 학습
xgb_model = xgb.XGBClassifier(random_state=42, enable_categorical=True)
xgb_model.fit(X_train_resampled, y_train_resampled)

# 2. 테스트 데이터로 예측
y_pred_xgb = xgb_model.predict(X_test)

# 3. 성능 평가
print("XGBoost 성능 평가:")
print(classification_report(y_test, y_pred_xgb))
```

6. 예측 모델의 성능 평가 (이미지 첨부)

팀원1

- 모델 성능 평가 비교

Random Forest with	Recall	Precision	F1 Score	Accuracy
Class weights	0.935780	0.207640	0.339867	0.981818
SMOTE Oversampling	0.933081	0.191367	0.317597	0.979945
No Under/Oversampling	0.652455	0.967200	0.779246	0.998151
Random Undersampling	0.96816	0.151214	0.261573	0.97266

  - 4개의 모델 모두 RandomForest 를 사용하였으며, 불균형한 데이터를 어떻게 가공하여 피처(features) 로 사용하였는지에 따라 모델의 이름을 정하였다.
  - 본 주제는 신용카드 거래 다수의 정상거래 중에 사기거래를 찾아내는 것으로, 정밀도(Precision) 보다는 **재현율(Recall)**에 중점을 두어 모델을 선택해야 한다. 예를 들어 의사가 암환자를 진단하는 상황과 유사하다고 볼 수 있다. 의사가 진단한 암양성자 중에 실제 암양성자의 비율(Precision)보다 실제 암양성중 의사가 진단한 암양성자의 비율(Recall)이 중요하다.
  - 다시 신용카드 사기거래 탐지에서 살펴보면 고객의 신용카드가 사기거래에 사용되면 심각한 상황이 연출될 수 있으므로, 정밀도(Precision)가 높은것보다 재현율(Recall)이 높은것으로 선택해야 한다. 하지만, 재현율(Recall)이 높은것만 고려하면 상대적으로 사기거래가 아닌 거래도 사기거래로 탐지하여 고객에게 스트레스를 줄 수도 있다. 결과적으로 정밀도 보다는 재현율에 중점을 두되 정밀도를 완전히 포기할 수 없으므로, 조화평균인 F1 Score 를 고려해야 한다.
  - 위 상황을 종합적으로 고려해 볼때 재현율이 다소 낮더라도 F1 Score 가 가장 높은 No **Under/Oversampling** 을 모델 선택의 우선순위로 둘 수 있다. 한편 고객의 신용카드 사기거래를 엄격하게 막기위해 재현율이 높은 모델을 선택하려면 **Random Undersampling** 을 선택할 수 있다.

팀원2

- Smote_RandomForest/XgBoost Model

```
Random Forest 성능 평가:
Confusion Matrix:
 [[552240   1334]
 [  2140      5]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    553574
           1       0.00      0.00      0.00      2145

    accuracy                           0.99    555719
   macro avg       0.50      0.50      0.50    555719
weighted avg       0.99      0.99      0.99    555719

XGBoost 성능 평가:
Confusion Matrix:
 [[544816   8758]
 [  1868    277]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.98      0.99    553574
           1       0.03      0.13      0.05      2145

    accuracy                           0.98    555719
   macro avg       0.51      0.56      0.52    555719
weighted avg       0.99      0.98      0.99    555719
```

- 랜덤 포레스트의 경우에는 정확도가 매우 높지만, 사기 클래스에 대해 모델이 거의 예측하지 못하고, 이로 인해 정밀도와 재현율이 0가 나온다.
- XGBoost는 랜덤 포레스트보다 약간 낮은 정확도를 가지지만, 사기 클래스에 대해 몇 가지를 식별할 수 있다. 하지만 여전히 사기 클래스에 대한 정밀도가 낮고, F1 Score가 매우 낮다.

- Smote_RandomForest/XgBoost_Hyperparameter Model.
  1. n_iter = 72, cv = 5, **Grid Search**

```
Fitting 5 folds for each of 72 candidates, totalling 360 fits
[CV] END class_weight=balanced, max_depth=None, min_samples_split=2, n_estimators=100; total time=42.9min
[CV] END class_weight=balanced, max_depth=None, min_samples_split=2, n_estimators=100; total time=45.4min
[CV] END class_weight=balanced, max_depth=None, min_samples_split=2, n_estimators=100; total time=43.7min
[CV] END class_weight=balanced, max_depth=None, min_samples_split=2, n_estimators=200; total time=87.0min
[CV] END class_weight=balanced, max_depth=None, min_samples_split=2, n_estimators=100; total time=44.3min
[CV] END class_weight=balanced, max_depth=None, min_samples_split=2, n_estimators=200; total time=88.4min
[CV] END class_weight=balanced, max_depth=None, min_samples_split=2, n_estimators=100; total time=44.9min
[CV] END class_weight=balanced, max_depth=None, min_samples_split=2, n_estimators=200; total time=88.0min
[CV] END class_weight=balanced, max_depth=None, min_samples_split=2, n_estimators=200; total time=87.0min
[CV] END class_weight=balanced, max_depth=None, min_samples_split=2, n_estimators=300; total time=132.9min
[CV] END class_weight=balanced, max_depth=None, min_samples_split=2, n_estimators=200; total time=89.8min
[CV] END class_weight=balanced, max_depth=None, min_samples_split=2, n_estimators=300; total time=129.0min
[CV] END class_weight=balanced, max_depth=None, min_samples_split=2, n_estimators=300; total time=132.8min
[CV] END class_weight=balanced, max_depth=None, min_samples_split=5, n_estimators=100; total time=44.1min
[CV] END class_weight=balanced, max_depth=None, min_samples_split=5, n_estimators=100; total time=44.8min
[CV] END class_weight=balanced, max_depth=None, min_samples_split=2, n_estimators=300; total time=125.3min
[CV] END class_weight=balanced, max_depth=None, min_samples_split=2, n_estimators=300; total time=134.3min
+ Code  + Markdown
```

     - 위의 하이퍼파라미터 튜닝의 경우, 소프트웨어적 한계로 인하여 결과를 얻을 수 없었다.
  2. n_iter = 20, cv = 3, **RandomizedSearch**

```
# 최적의 하이퍼파라미터 (Random Forest): {'class_weight': 'balanced', 'max_depth': None,
# 최고 F1-Score (Random Forest): 0.6662714576933472

# 최적의 하이퍼파라미터 (XGBoost): {'colsample_bytree': 0.8046170495009265, 'learning_ra
# 최고 F1-Score (XGBoost): 0.6550819172450099
```

     - 위의 경우도 시간은 오래 결렸지만 결과를 얻을 수 있었다.

```
Random Forest 성능 평가:
Confusion Matrix:
 [[530735  22839]
 [   534   1611]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.96      0.98    553574
           1       0.07      0.75      0.12      2145

    accuracy                           0.96    555719
   macro avg       0.53      0.85      0.55    555719
weighted avg       1.00      0.96      0.98    555719

XGBoost 성능 평가:
Confusion Matrix:
 [[511667  41907]
 [   554   1591]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.92      0.96    553574
           1       0.04      0.74      0.07      2145

    accuracy                           0.92    555719
   macro avg       0.52      0.83      0.51    555719
weighted avg       1.00      0.92      0.96    555719
```

- 위의 결과에서 얻을 수 있는 것은 절대적으로 보면 성능적으로 2145개의 사기 거래 중 1591개만 정확히 예측했으며, 554개는 비사기로 잘못 예측하는 등의 아직 부족한 성능을 보이지만, 이전 성능 평가와 비교해보면 정확도, 재현율, 정밀도, 재현율 모두 높아졌다.

팀원3

1. Oversampling

```
로지스틱 회귀 성능 평가:
              precision    recall  f1-score   support

           0       1.00      0.61      0.76    218771
           1       0.01      0.71      0.01       692

    accuracy                           0.61    219463
   macro avg       0.50      0.66      0.38    219463
weighted avg       1.00      0.61      0.75    219463
```

- 모델이 정상거래는 잘 예측하지만 사기거래에 대해서는 예측률이 매우 낮다.
- 재현율은 사기거래에서 상대적으로 높지만 이는 사기거래로 예측한 값들이 많다는 것을 뜻할 수 있겠다.

```
랜덤 포레스트 성능 평가:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    218771
           1       0.52      0.46      0.48       692

    accuracy                           1.00    219463
   macro avg       0.76      0.73      0.74    219463
weighted avg       1.00      1.00      1.00    219463
```

- 정상거래를 잘 예측하며 사기거래에 대한 예측 성능이 로지스틱 회귀에 비해 개선되었다.
- 사기거래의 F1-score도 큰 향상을 보인다.

XGBoost 성능 평가:

	precision	recall	f1-score	support
0	1.00	0.94	0.97	218771
1	0.04	0.84	0.08	692
accuracy			0.94	219463
macro avg	0.52	0.89	0.53	219463
weighted avg	1.00	0.94	0.97	219463

- 사기거래 Precision값은 0.04로 매우 낮고 Recall은 0.84로 높은 값을 보인다.
- 정상거래의 성능은 매우 우수하지만 사기거래에 대한 F1-score는 0.08로 낮게 나타난다.

2. Randomundersampling

로지스틱 회귀 성능 평가:

	precision	recall	f1-score	support
0	1.00	0.58	0.73	218771
1	0.01	0.71	0.01	692
accuracy			0.58	219463
macro avg	0.50	0.64	0.37	219463
weighted avg	1.00	0.58	0.73	219463

- 정상거래는 정밀도가 매우 높지만, 사기거래에 대한 성능은 재현율을 제외하고는 낮은 수치를 보인다.

랜덤 포레스트 성능 평가:

	precision	recall	f1-score	support
0	1.00	0.86	0.92	218771
1	0.02	0.89	0.04	692
accuracy			0.86	219463
macro avg	0.51	0.87	0.48	219463
weighted avg	1.00	0.86	0.92	219463

- 정상거래의 성능은 높으며, 사기거래에 대한 재현율(Recall)이 크게 증가했다.
- 로지스틱 회귀보다 더 좋은 성능을 보이지만, 사기거래의 예측에 있어서 여전히 개선이 필요한 것으로 보인다.

XGBoost 성능 평가:

	precision	recall	f1-score	support
0	1.00	0.88	0.93	218771
1	0.02	0.89	0.04	692
accuracy			0.88	219463
macro avg	0.51	0.88	0.49	219463
weighted avg	1.00	0.88	0.93	219463

- 랜덤 포레스트와 유사한 성능을 보이며 사기거래에 대한 재현율이 높다.

3. SMOTETomek

로지스틱 회귀 성능 평가:

	precision	recall	f1-score	support
0	1.00	0.61	0.75	218771
1	0.01	0.68	0.01	692
accuracy			0.61	219463
macro avg	0.50	0.65	0.38	219463
weighted avg	1.00	0.61	0.75	219463

- 정상거래에 대한 성능은 높지만 사기거래에 대한 예측 성능이 여전히 매우 낮다.
- 사기거래의 재현율은 0.68로 비교적 높은 값을 보이지만, 정밀도가 낮아 실질적인 성능이 저조한 것을 볼 수 있다.

랜덤 포레스트 성능 평가:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	218771
1	0.50	0.45	0.47	692
accuracy			1.00	219463
macro avg	0.75	0.72	0.74	219463
weighted avg	1.00	1.00	1.00	219463

- 사기거래에 대한 Precision은 0.50, Recall은 0.45로 이전보다 나아졌다. 그러나 여전히 사기거래 예측 성능이 완벽하지 않으며 추가적인 조정이 필요해 보인다.

XGBoost 성능 평가:

	precision	recall	f1-score	support
0	1.00	0.94	0.97	218771
1	0.04	0.84	0.09	692
accuracy			0.94	219463
macro avg	0.52	0.89	0.53	219463
weighted avg	1.00	0.94	0.97	219463

- 사기거래에 대한 재현율이 매우 높지만 Precision이 0.04로 매우 낮아 사기거래로 예측된 값들 중 실제로 사기거래인 비율이 거의 없다 볼 수 있겠다.

7. 프로젝트 결과 및 최종 결론

팀원1

EDA 를 통해 관찰 된 내용 중 가장 관건이 되었던 것은 불균형 데이터(Imbalanced Data)였다. 일반적인 머신러닝 알고리즘으로 학습과 예측을 할 경우, 머신러닝은 모든 데이터를 정상거래로 판단할 것임이 자명하였다. (그렇게 해도 99.5% 의 성능을 낼 수 있기 때문이다. 필자는 본 보고서에 작성하지 않았지만, LogisticRegression 으로 불균형 데이터를 학습하고 예측 했을 때 모두 정상건이라고 예측하고 사기건은 찾아내지 못했다.)

그래서 해결 방법으로 SMOTE(Synthetic Minority Oversampling Technique)을 사용해서 테스트 데이터 중 불균형한 극소량의 데이터를 오버샘플링 하였다. 또한 두번째 방법으로 교차검증을 사용해서 재현율(Recall) 중심으로 5개의 점수를 찾고 평균을 냈다. 세번째 GridSearchCV 를 통해서 하이퍼파라미터 튜닝의 최적값을 찾아 max_depth 와 n_estimator 를 설정했다. 모델로는 DecisionTree 의 Ensemble 모델인 RandomForestClassifier 를 사용하였다. 이와 같은 결과로 '6. 모델 성능 평가'와 같은 결과를 얻게 되었다.(위 6. 모델 성능 평가 참조)

이 모든 과정의 최종 목적은 비지니스에 어떻게 연결 시키느냐 일것이다. 신용카드 사기거래 탐지의 경우 고객의 신용카드가 사기거래에 사용 되는 것을 막기 위해서는 재현율(Recall)에 중점을 두어야 하지만, 동시에 낮은 정밀도(Precision)에 의해 정상거래 고객에게 스트레스를 주어서는 안 될 것이다.

결론적으로 **재현율**에 중점을 두되 **F1 Score**(재현율과 정밀도의 조화평균)과 함께 고려하여 비지니스 상황에 맞춰 의사결정에 반영해야 할 것이다. 클라이언트에게 솔루션을 제공해줄 때 한가지의 모델링을 주는 것 보다 몇가지 모델을 제시하고 재현율, 정밀도, 정확도, F1 Score 을 설명하여 비지니스에 최적화하도록 하는 것이 현명한 방법이 될 것이다.

덧붙이는 말로 필자는 여러 모델을 오버샘플링하고 교차검증, 하이퍼파라미터 튜닝을 하느라 컴퓨팅 파워의 한계를 확실하게 경험했다. (한개의 셀이 완료되기까지 3시간 이상넘어, 결국에는 오버플로우되어 커널이 재시작되는 경험을 반복했다.) K-ICT빅데이터 센터에서 인프라를 제공받아 cpu:40, memory=128GB 에서 실행하였지만, 여전히 한계를 극복하지 못했다. 이제 마지막 대안으로 spark + hadoop 이 될 것 같다. 금융분야 그리고 불균형한 이상탐지 데이터를 분석하기 위해서는 분산 병렬 데이터 분석 플랫폼인 spark + hadoop 을 추천한다.

팀원2

EDA부터 성능평가까지의 과정을 요약해보면 다음과 같다. 우선 전체적인 EDA를 통해 데이터의 특성을 찾았다. 이 과정에서 수치적 데이터에 집중하여 전처리와 인코딩, 스케일링, 그리고 데이터 불균형 때문에 Smote를 이용하였다. 다음으로 이를 랜덤포레스트 방식과 XGBoost 방식으로 예측 모델링을 하였다. 그리고 이중 결과만 놓고 보면 랜덤포레스트 방식이 약간 더 우세한 것으로 나왔다. 이후에 이 결과를 더 발전시키는 방향으로 수치적 데이터 뿐만 아니라 추가적인 정성적 데이터를 추가하고 모델링 이후에 하이퍼파라미터 튜닝 과정에서 튜닝 수치를 높이면 더 나은 결과를 얻을 수 있을 것으로 생각된다. 다만 튜닝 과정에서 시간 대비 발전 기대값이 높진 않다는 한계가 있다.

팀원3

EDA를 통한 유의미한 특징과 그에 따른 변수들을 추출,인코딩한후 여러가지방법으로 데이터수를 조정하고 이를 다양한모델로 성능평가를 해본 결과 smotetomek의 XGBoost모델이 시도한 평가중 가장 나은 성능을 보인 것으로 확인되었다.일련의 과정을 통해 많이 배울 수 있었고 그 과정속에서 머신러닝 모델학습에 대한 전반적인 흐름을 알 수 있었던 거 같다.

참조

- 불균형 데이터를 처리하여 모델링 하는 해결방법 : https://www.kaggle.com/code/marcinrutecki/best-techniques-and-metrics-for-imbalanced-dataset