# 11장 LightGBM : 카드 거래 내역 데이터셋

#### 학습 목표

가장 최신 부스팅 기법에 속하는 LightGBM을 활용하여 이상거래를 예측하는 모델을 만들어봅니다. 하이퍼파라미터 튜닝에 사용하는 랜덤 그리드 서치와 모델 평가에 사용할 AUCarea under the curve 개념을 학습합니다.

#### 학습 순서



#### LightGBM 소개

XGBoost 이후로 나온 최신 부스팅 모델입니다. LightGBM이 등장하기 전까지는 XGBoost가 가장 인기있는 부스팅 모델이였지만, 점점 LightGBM이 XGBoost와 비슷한 수준 혹은 그 이상으로 활용되는 추세입니다. 캐글 컴피티션에서도 좋은 퍼포먼스를 많이 보여주어서 그 성능을 인정받았습니다. 리프 중심 트리 분할 방식을 사용합니다.

<그림/>



리프 중심 트리 분할

</>

#### 장단점

| **장점** | **단점** |
| --- | --- |
| XGBoost보다도 빠르고 높은 정확도를 보여주는 경우가 많습니다. | 복잡한 모델인 만큼, 해석에 어려움이 있습니다. |
| 예측에 영향을 미친 변수의 중요도를 확인할 수 있습니다. | 하이퍼파라미터 튜닝이 까다롭습니다. |
| 변수 종류가 많고 데이터가 클수록 상대적으로 뛰어난 성능을 보여줍니다. |  |

#### 유용한 곳

* 종속변수가 연속형 데이터인 경우든 범주형 데이터인 경우든 모두 사용할 수 있습니다.
* 이미지나 자연어가 아닌 표로 정리된 데이터라면 거의 모든 상황에서 활용할 수 있습니다.

#### TOP 10 선정 이유

* 표로 정리된 데이터tabular data에서 캣부스트, XGBoost와 함께 가장 좋은 성능을 보여주는 알고리즘입니다. 성능을 비교하는 자료를 보면 대체로 LightGBM와 캣부스트가 XGBoost보다 나은 성능을 보여주며, LightGBM과 캣부스트는 상황에 따라 다른 결과를 보여주어 우열을 가리기 어렵습니다. 다만, 사람들에게 활용/언급되는 빈도는 아직까지는 LightGBM이 우위를 보입니다.

## 11.1 문제 정의 : 한눈에 보는 분석 목표

<금토끼의 문제 정의> “삐리릭~ 결제가 완료되었습니다”. 잠시 후 “금토끼 님”을 부르는 소릴 듣고 커피를 받아들고 회사로 향했습니다. 모니터 앞에 앉자마자 메일을 살펴보니 본부장에게서 메일이 왔군요. 최근에 남의 카드를 도용해 고가의 물건을 구매하는 일의 비율이 높아져서 우리 회사의 손해가 이만저만이 아니니 이상거래를 탐지해 방지하는 모델을 만들어 달라는 요청이 적혀 있습니다. ‘이상거래를 탐지하려면 평소 구매하던 패턴을 분석해야겠군. 지리적인 위치도 고려해야겠어요.’ 남은 커피 한 모금을 마저 마시고, 메일에 첨부된 데이터셋 데이터 분석에 돌입했습니다.

| **난이도** | ⭐⭐⭐ | | |
| --- | --- | --- | --- |
| **알고리즘** | LightGBM(LightGBM) | | |
| **데이터셋 파일명** | fraud.csv | **종속 변수** | is\_fraud(이상거래) |
| **데이터셋 소개** | 이상거래에 관련된 데이터입니다. 이상거래라 함은 카드값을 지불하지 않을 의도를 가지고서 결제를 하거나, 도난된 카드를 가지고 결제를 하는 등의 거래를 의미합니다. 종속 변수는 이상거래 여부이고, 독립변수는 거래 시간, 거래 금액, 고객 성별, 상점 카테고리 등입니다. | | |
| **문제 유형** | 분류 | **평가지표** | 정확도, 혼동 행렬, 분류 리포트, ROC AUC 점수 |
| **사용한 모델** | LGBMClassifier, train | | |
| **사용 라이브러리** | * numpy (numpy==1.19.5) * pandas (pandas==1.3.5) * seaborn (seaborn==0.11.2) * matplotlib (matplotlib==3.2.2) * sklearn (scikit-learn==1.0.2) * lightgbm (lightgbm==2.2.3) | | |
| **예제 코드 노트북** | 위치 : <https://github.com/musthave-ML10/notebooks/>  파일 : 11\_LightGBM.ipynb | | |

## 11.2 라이브러리 및 데이터 불러오기, 데이터 확인하기

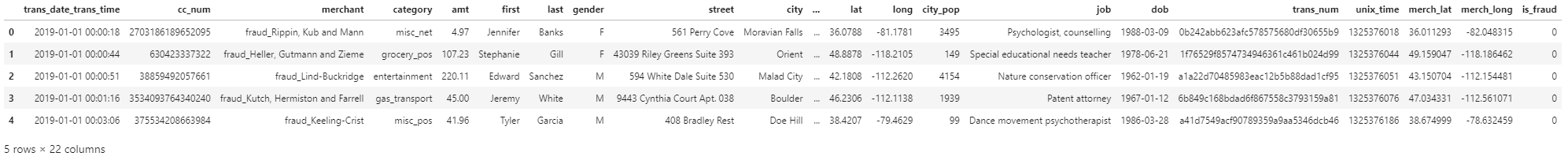
이번에 사용할 데이터는 fraud.csv로 저장되어 있습니다. 해당 파일과 함께 기본 라이브러리를 불러오겠습니다.

| import pandas as pd import numpy as np import matplotlib.pyplot as plt import seaborn as sns  file\_url = 'https://media.githubusercontent.com/media/musthave-ML10/data\_source/main/fraud.csv' data = pd.read\_csv(file\_url) # 데이터셋 읽기 |
| --- |

head() 함수를 사용하여 데이터가 어떻게 생겼는지 간단히 살펴보겠습니다.

| data.head() # 상위 5행 출력 |
| --- |

|  | trans\_date\_trans\_time | cc\_num | merchant | category | amt | first | last | gender | street | city | state | zip | lat | long | city\_pop | job | dob | trans\_num | unix\_time | merch\_lat | merch\_long | is\_fraud |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 2019-01-01 00:00:18 | 2703186189652095 | fraud\_Rippin, Kub and Mann | misc\_net | 4.97 | Jennifer | Banks | F | 561 Perry Cove | Moravian Falls | NC | 28654 | 36.0788 | -81.1781 | 3495 | Psychologist, counselling | 1988-03-09 | 0b242abb623afc578575680df30655b9 | 1325376018 | 36.011293 | -82.048315 | 0 |
| 1 | 2019-01-01 00:00:44 | 630423337322 | fraud\_Heller, Gutmann and Zieme | grocery\_pos | 107.23 | Stephanie | Gill | F | 43039 Riley Greens Suite 393 | Orient | WA | 99160 | 48.8878 | -118.2105 | 149 | Special educational needs teacher | 1978-06-21 | 1f76529f8574734946361c461b024d99 | 1325376044 | 49.159047 | -118.186462 | 0 |
| 2 | 2019-01-01 00:00:51 | 38859492057661 | fraud\_Lind-Buckridge | entertainment | 220.11 | Edward | Sanchez | M | 594 White Dale Suite 530 | Malad City | ID | 83252 | 42.1808 | -112.262 | 4154 | Nature conservation officer | 1962-01-19 | a1a22d70485983eac12b5b88dad1cf95 | 1325376051 | 43.150704 | -112.154481 | 0 |
| 3 | 2019-01-01 00:01:16 | 3534093764340240 | fraud\_Kutch, Hermiston and Farrell | gas\_transport | 45.0 | Jeremy | White | M | 9443 Cynthia Court Apt. 038 | Boulder | MT | 59632 | 46.2306 | -112.1138 | 1939 | Patent attorney | 1967-01-12 | 6b849c168bdad6f867558c3793159a81 | 1325376076 | 47.034331 | -112.561071 | 0 |
| 4 | 2019-01-01 00:03:06 | 375534208663984 | fraud\_Keeling-Crist | misc\_pos | 41.96 | Tyler | Garcia | M | 408 Bradley Rest | Doe Hill | VA | 24433 | 38.4207 | -79.4629 | 99 | Dance movement psychotherapist | 1986-03-28 | a41d7549acf90789359a9aa5346dcb46 | 1325376186 | 38.674999 | -78.632459 | 0 |

  
  
  
변수는 총 22개로, 20개가 넘어서 state와 zip이 생략된 상태로 표시되었습니다. 변수에 대한 설명은 다음과 같습니다.

* trans\_date\_trans\_time : 거래 시간
* cc\_num : 카드 번호. 고윳값이기 때문에 여기서는 id처럼 활용할 수 있습니다.
* merchant : 거래 상정
* category : 거래 상점의 카테고리(애완용품, 여행, 엔터테인먼트 등)
* amt : 거래 금액
* first / last : 이름
* gender : 성별
* street / state / zip : 고객 거주지 정보
* lat / long : 고객 주소에 대한 위도 및 경도
* city\_pop : 고객의 zipcode에 속하는 인구 수
* job : 직업
* dob : 생년월일
* trans\_num : 거래번호
* unix\_time : 거래 시간(유닉스 타임스탬프 형식)
* merch\_lat / merch\_long : 상점 위치에 대한 위도 및 경도
* is\_fraud : 사기거래 여부 (종속 변수)

다음은 info() 함수를 사용해 살펴보겠습니다.

| data.info() # 변수 특징 출력 |
| --- |

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 1852394 entries, 0 to 1852393

Data columns (total 22 columns):

# Column Dtype

--- ------ -----

0 trans\_date\_trans\_time object

1 cc\_num int64

2 merchant object

3 category object

4 amt float64

5 first object

6 last object

7 gender object

8 street object

9 city object

10 state object

11 zip int64

12 lat float64

13 long float64

14 city\_pop int64

15 job object

16 dob object

17 trans\_num object

18 unix\_time int64

19 merch\_lat float64

20 merch\_long float64

21 is\_fraud int64

dtypes: float64(5), int64(5), object(12)

memory usage: 310.9+ MB

이번에는 평소와는 다르게 Non-null count가 나오지 않았습니다. 판다스에서는 데이터가 너무 클 때는 Non-null count를 생략하고 보여줍니다. 이를 강제로 보이게 하는 데 show\_counts 매개변수를 사용합니다.

| data.info(show\_counts=True) # Non-Null Count 보이도록 출력 |
| --- |

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 1852394 entries, 0 to 1852393

Data columns (total 22 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 trans\_date\_trans\_time 1852394 non-null object

1 cc\_num 1852394 non-null int64

2 merchant 1852394 non-null object

3 category 1852394 non-null object

4 amt 1852394 non-null float64

5 first 1852394 non-null object

6 last 1852394 non-null object

7 gender 1852394 non-null object

8 street 1852394 non-null object

9 city 1852394 non-null object

10 state 1852394 non-null object

11 zip 1852394 non-null int64

12 lat 1852394 non-null float64

13 long 1852394 non-null float64

14 city\_pop 1852394 non-null int64

15 job 1852394 non-null object

16 dob 1852394 non-null object

17 trans\_num 1852394 non-null object

18 unix\_time 1852394 non-null int64

19 merch\_lat 1852394 non-null float64

20 merch\_long 1852394 non-null float64

21 is\_fraud 1852394 non-null int64

dtypes: float64(5), int64(5), object(12)

memory usage: 310.9+ MB

결측치가 있는 변수는 없습니다. 자료형에 하나 특이사항이 보입니다. trans\_date\_trans\_time은 날짜/시간 형태의 데이터인데, 자료형이 object형입니다. 날짜/시간 데이터는 datetime형이어야 날짜/시간 관련 함수에 적용할 수 있으므로, 잠시 후에 이 부분을 수정해주겠습니다.

다음은 describe()로 호출한 정보입니다.

| round(data.describe(), 2) # 통계 정보 확인 |
| --- |

|  | cc\_num | amt | zip | lat | long | city\_pop | unix\_time | merch\_lat | merch\_long | is\_fraud |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| count | 1852394.0 | 1852394.0 | 1852394.0 | 1852394.0 | 1852394.0 | 1852394.0 | 1852394.0 | 1852394.0 | 1852394.0 | 1852394.0 |
| mean | 4.1738603839400646e+17 | 70.06 | 48813.26 | 38.54 | -90.23 | 88643.67 | 1358674218.83 | 38.54 | -90.23 | 0.01 |
| std | 1.3091152653180204e+18 | 159.25 | 26881.85 | 5.07 | 13.75 | 301487.62 | 18195081.39 | 5.11 | 13.76 | 0.07 |
| min | 60416207185.0 | 1.0 | 1257.0 | 20.03 | -165.67 | 23.0 | 1325376018.0 | 19.03 | -166.67 | 0.0 |
| 25% | 180042946491150.0 | 9.64 | 26237.0 | 34.67 | -96.8 | 741.0 | 1343016823.75 | 34.74 | -96.9 | 0.0 |
| 50% | 3521417320836166.0 | 47.45 | 48174.0 | 39.35 | -87.48 | 2443 | 1357089331 | 39.37 | -87.44 | 0 |
| 75% | 4642255475285942.0 | 83.1 | 72042.0 | 41.94 | -80.16 | 20328 | 1374581485 | 41.96 | -80.25 | 0 |
| max | 4.992346398065154e+18 | 28948.9 | 99921.0 | 66.69 | -67.95 | 2906700 | 1388534374 | 67.51 | -66.95 | 1 |

❶ amt와 ❷ city\_pop에서 max값이 급격하게 높아져서 아웃라이어를 의심해볼 수 있는데, 변수 특성상 충분히 일어날 수 있으므로 별도로 처리하지는 않겠습니다. ❸ is\_fraud의 mean값을 보면 0.01로 매우 낮습니다. 즉, fraud인 경우가 1%로 매우 낮기 때문에 이점을 유의해서 분석 및 해석해야 합니다. 이런 때는 정확도가 99%이더라도 좋은 모델이라고 보장할 수 없습니다. 이렇게 한쪽으로 치우친 데이터를 비대칭 데이터imbalanced data라고 부르는데, 때에 따라서 오버샘플링을 해주어 예측 정확도를 높일 수도 있습니다.

## 11.3 전처리 : 데이터 클리닝

가장 먼저 불필요한 변수들을 제외시켜야 합니다. 종속 변수의 특성을 고려했을 때, 이 변수는 없어도 될 것 같다는 확신이 들거나 굳이 이용하지 않겠다고 생각한 변수들을 추려봅시다.

우선 카드 이름을 통해서 이상거래(fraud)를 예측한다는 것은 합리적이기 않기 때문에, 이름 관련 변수를 제외할 수 있습니다. street, city, state, zip의 경우 위도와 경도 정보가 있어 사용하지 않겠습니다. 만약 특정 도시/장소에서 이상거래가 더 빈번하다면 가치가 있겠으나 여기에서는 제외하겠습니다. trans\_num은 거래번호이고 중복되는 값이 없는 거래에 대한 id이기 때문에 필요하지 않습니다. unix\_time은 거래 시간에 대한 정보로서, trans\_date\_trans\_time을 쓰면 되므로 제외하겠습니다. job은 종류가 많지 않으면 더미 변수로 활용해보겠으나, 종류가 많기 때문에 굳이 여기서는 활용하지 않겠습니다. 마지막으로 merchant 또한 제외 대상에 포함하겠습니다. 특정 상점에서의 이상거래가 더 높을 수도 있겠으나, category 변수로 어느 정도 대체가 되는 부분입니다. 또한 카드 정보를 획득한 사람이 특정 물건을 사기 위해 해당 카테고리 내에서 어떤 상점이든 이용할 수 있으니, 미래의 정보를 고려할 때 merchant 변수는 자칫 오버피팅을 야기할 수도 있습니다.

| data.drop(['first','last','street','city','state','zip','trans\_num','unix\_time','job','merchant'], axis=1, inplace= True) # 변수 제거 |
| --- |

다음은 trans\_date\_trans\_time 변수를 datetime형으로 수정해주겠습니다.

| data['trans\_date\_trans\_time'] = pd.to\_datetime(data['trans\_date\_trans\_time']) # 날짜 형식으로 변환 |
| --- |

이제 info() 함수를 호출해 자료형을 다시 확인하면 object가 아닌 datetime64[ns]로 변경되었습니다.

| data.info() # 변수 특징 출력 |
| --- |

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 1852394 entries, 0 to 1852393

Data columns (total 13 columns):

# Column Dtype

--- ------ -----

0 trans\_date\_trans\_time datetime64[ns]

1 cc\_num int64

2 category object

3 amt float64

4 gender object

5 lat float64

6 long float64

7 city\_pop int64

8 job object

9 dob object

10 merch\_lat float64

11 merch\_long float64

12 is\_fraud int64

dtypes: datetime64[ns](1), float64(5), int64(3), object(4)

memory usage: 183.7+ MB

## 11.4 전처리 : 피처 엔지니어링

이상거래 감지의 기본적인 아이디어는 해당 고객의 기존 거래 패턴에서 벗어나는 경우를 감지하는 겁니다. 예를 들어 서울에서만 카드를 사용하던 사람이 갑자기 런던에서 고액의 물건을 구매했다던가, 한 번도 주류매장을 이용한 적이 없던 고객이 주류매장에서 고액의 결제가 일어났다든가 하는 경우입니다. 실제로 많은 카드사들이 이런 식으로 이상거래를 감지하고, 이상거래가 감지되면 해당 결제를 잠시 보류하고 고객에게 문자/메일로 확인받는 시스템을 갖추고 있습니다.

### 11.4.1 결제 금액

첫 번째로 다룰 피처 엔지니어링은 결제 금액에 대한 부분입니다. A라는 고객이 평소에 $100, $120, $80, $30 정도의 금액을 사용하다가 갑자기 $2,000의 금액을 사용하면 평소와 다르다고 볼 수 있습니다. 반면 B라는 고객의 평소 소비 패턴이 $10, $3,000, $300, $1,300과 같은 식으로 들쭉날쭉하다면 어느날 $5,000의 거래가 발생해도 크게 이상해보이지는 않을 겁니다. 이러한 패턴을 파악하기 위해 Z 점수(Z-Score, z값, 표준값, 표준 점수라고도 씁니다)를 사용할 겁니다. 표준 점수는 평균과 표준편차를 이용하여 특정값이 정규분포 범위에서 어느 수준에 위치하는지를 나타냅니다. 수식은 다음과 같습니다.

<용어/>

**Z 점수**

평균과 표준편차를 이용하여 특정값이 정규분포 범위에서 어느 수준에 위치하는지를 나타냅니다.

</>

여기서 평균과 표준편차는 각 개인의 거래내역에 대한 값이므로, cc\_num 기준으로 groupby를 사용하여 구할 수 있습니다.

| amt\_info = data.groupby('cc\_num').agg(['mean','std'])['amt'].reset\_index() # cc\_num별 amt 평균과 표준편차 계산 |
| --- |

head() 함수를 사용하여 amt\_info 확인하겠습니다.

| amt\_info.head() # 상위 5행 확인 |
| --- |

|  | cc\_num | mean | std |
| --- | --- | --- | --- |
| 0 | 60416207185 | 59.257796 | 142.869746 |
| 1 | 60422928733 | 65.483159 | 92.042844 |
| 2 | 60423098130 | 96.376084 | 1,000.693872 |
| 3 | 60427851591 | 107.487550 | 131.014534 |
| 4 | 60487002085 | 64.096925 | 153.207660 |

카드번호별로 거래금액의 평균과 표준편차가 계산되어 있습니다. cc\_num을 키값으로 하여 기존 데이터에 이 데이터를 left join으로 붙이면 원하는 형태의 데이터를 얻을 수 있습니다.

| data = data.merge(amt\_info, on ='cc\_num', how='left') # 데이터 합치기 |
| --- |

이제 기존 데이터에 특정값(amt), 평균(mean), 표준편차(std)가 모두 모였으니 Z 점수를 계산하겠습니다.

| data['amt\_z\_score'] = (data['amt'] - data['mean']) / data['std'] # z-score 계산 |
| --- |

Z 점수에 대한 해석에 사용할 amt, mean, std, amt\_z\_score 변수만 head()로 불러보겠습니다.

| data[['amt','mean','std','amt\_z\_score']].head() # 상위 5행 확인 |
| --- |

|  | amt | mean | std | amt\_z\_score |
| --- | --- | --- | --- | --- |
| 0 | 4.97 | 89.408743 | 127.530101 | -0.662108 |
| 1 | 107.23 | 56.078113 | 159.201852 | 0.321302 |
| 2 | 220.11 | 69.924272 | 116.688602 | 1.287064 |
| 3 | 45.0 | 80.090040 | 280.077880 | -0.125287 |
| 4 | 41.96 | 95.341146 | 94.322842 | -0.565941 |

인덱스 2행을 확인하면 평균 거래 금액이 약 $69.9 수준인데 해당 지출금액은 $220으로 상당한 차이를 보입니다. 표준편차는 116으로, 이에 대한 Z 점수는 1.28이 나왔습니다. 위에 나타난 다른 거래 건들과 비교하면 상대적으로 특이합니다.

mean과 std는 Z 점수를 구하는 데 필요한 값일뿐 예측에는 필요하지 않으니 제거합니다.

| data.drop(['mean','std'], axis=1, inplace=True) # 변수 제거 |
| --- |

### 11.4.2 카테고리

다음으로 처리할 피처 엔지니어링도 Z 점수인데, 이번에는 카테고리 관점까지 파고들어서 더욱 디테일하게 만들겠습니다. 개인의 소비 패턴도 다르지만, 각 개인마다 어떤 카테고리에 얼마만큼의 금액을 쓰는지 또한 다를 겁니다. 이에 대한 Z 점수도 구하려면 위와 같은 작업을 하되, 평균과 표준편차를 구하는 groupby 과정에서 cc\_num과 category를 함께 적용합니다.

| category\_info = data.groupby(['cc\_num','category']).agg(['mean','std'])['amt'].reset\_index() # cc\_num과 category 기준으로 amt의 평균, 표준편차 계산 |
| --- |

merge()를 사용해서 데이터를 합칠 때도 키값에 cc\_num과 category를 함께 넣어줘야 합니다.

| data = data.merge(category\_info, on =['cc\_num','category'], how='left') # 데이터 합치기 |
| --- |

Z 점수를 구하고 불필요해진 mean과 std 변수를 제거합니다.

| data['cat\_z\_score'] = (data['amt'] - data['mean']) / data['std'] # z-score 계산 data.drop(['mean','std'], axis=1, inplace=True) # 변수 제거 |
| --- |

### 11.4.3 거리

다음 피처 엔지니어링은 거리입니다. 이 데이터에는 고객의 위치와 상점의 위치가 있기 때문에 이 둘 사이의 거리를 계산할 수 있습니다. 거리에 대한 Z 점수를 활용하면 기존 패턴에서 벗어난 거래를 감지할 수 있습니다. 예를 들어 고객이 주로 주소지 부근 100km 반경에서만 거래를 하다가 3,000km 떨어진 지역에서 거래가 발생했다면, 카드 정보가 해킹되어 타인이 사용한 것은 아닌지 의심해볼 여지가 있습니다.

두 지점의 거리 계산에 geopy 라이브러리를 사용할 수 있습니다. 해당 라이브러리의 distance 모듈을 불러오겠습니다.

| import geopy.distance # 임포트 |
| --- |

distance 모듈은 두 지점에 대한 위도와 경도를 튜플 형태로 받아서 계산됩니다.

| geopy.distance.distance((위도1, 경도1), (위도2, 경도2)) |
| --- |

지금 우리가 가진 데이터는 각 위도와 경도가 각각 별도의 변수로 존재하기 때문에, zip을 활용하여 하나로 묶어주고, 이를 판다스 시리즈 형태로 정의하여 변수로 활용할 수 있습니다.

| data['merch\_coord'] = pd.Series(zip(data['merch\_lat'], data['merch\_long'])) # 위도, 경도 한 변수로 합치기 data['cust\_coord'] = pd.Series(zip(data['lat'], data['long'])) # 위도, 경도 한 변수로 합치기 |
| --- |

이제 merch\_coord에는 상점에 대한 위치, cust\_coord에는 고객 주소에 대한 위치가 저장되었습니다. 이제 distance()를 사용하여 거리를 계산할 수 있는데, 이 함수는 시리즈 형태가 아니라 한 건씩 데이터를 처리하기 때문에 apply()를 활용하여 적용시켜야 합니다.

| data['distance'] = data.apply(lambda x: geopy.distance.distance(x['merch\_coord'], x['cust\_coord']).km, axis=1) # 거리 계산 |
| --- |

이 코드를 실행시키면 상당히 오랜 시간이 걸립니다.

<글상자/>

**코스가 실행된 시간을 측정하기**

이렇게 시간이 오래 걸릴 때는 시간을 측정해두어 추후 코드를 수정할 때 참고하는 것이 좋습니다. 진행 시간을 확인하는 다양한 방법이 있는데, 여기서는 time 라이브러리를 소개하겠습니다. 우선 time 라이브러리를 불러옵니다.

| import time |
| --- |

그리고 time.time()을 실행시켜보면 현재 시간이 표시됩니다.

| time.time() |
| --- |

이 기능을 활용하여 distance 코드 실행 전과 실행 후의 현재 시간을 빼는 방식으로 총 얼마나 걸렸는지 확인할 수 있습니다.

| start\_time = time.time() data['distance'] = data.apply(lambda x: geopy.distance.distance(x['merch\_coord'], x['cust\_coord']).km, axis=1) end\_time = time.time() print(end\_time-start\_time) |
| --- |

564.754515171051

해당 정보는 초 단위로 표시됩니다. 저는 약 564초가 걸렸습니다.

</>

distance를 구했으면 이전과 같은 방법으로 Z 점수를 구해줍니다.

| distance\_info = data.groupby('cc\_num').agg(['mean','std'])['distance'].reset\_index() # cc\_num 별, 거리 정보 계산 data = data.merge(distance\_info, on ='cc\_num', how='left') # 데이터 합치기 data['distance\_z\_score'] = (data['distance'] - data['mean']) / data['std'] # z-score 계산 data.drop(['mean','std'], axis=1, inplace=True) # 변수 제거  data.head() # 상위 5행 출력 |
| --- |

|  | trans\_date\_trans\_time | cc\_num | category | amt | gender | lat | long | city\_pop | dob | merch\_lat | merch\_long | is\_fraud | amt\_z\_score | cat\_z\_score | merch\_coord | cust\_coord | distance | distance\_z\_score |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 43,466 | 2703186189652095 | misc\_net | 5 | F | 36.07880 | -81.18 | 3,495.00 | 32211 | 36.01 | -82.05 | 0.00 | -0.66 | -0.688297 | (36.011293, -82.048315) | (36.0788, -81.1781) | 78.773821 | 0.030974 |
| 1 | 43,466 | 630,423,337,322 | grocery\_pos | 107 | F | 48.88780 | -118.2 | 149.0 | 28662 | 49.16 | -118.19 | 0.00 | 0.32 | 0.317631 | (49.159047, -118.186462) | (48.8878, -118.2105) | 30.216618 | -1.475224 |
| 2 | 43,466 | 38,859,492,057,661 | entertainment | 220 | M | 42.18080 | -112.26 | 4,154.00 | 22665 | 43.15 | -112.15 | 0.00 | 1.29 | 2.872509 | (43.150704, -112.154481) | (42.1808, -112.262) | 108.102912 | 1.160572 |
| 3 | 43,466 | 3,534,093,764,340,240 | gas\_transport | 45 | M | 46.23060 | -112.11 | 1,939.00 | 24484 | 47.03 | -112.56 | 0.00 | -0.13 | -1.050197 | (47.034331, -112.561071) | (46.2306, -112.1138) | 95.685115 | 0.818063 |
| 4 | 43,466 | 375,534,208,663,984 | misc\_pos | 42 | M | 38.42070 | -79.46 | 99.00 | 31499 | 38.67 | -78.63 | 0.00 | -0.57 | 1.312866 | (38.674999, -78.632459) | (38.4207, -79.4629) | 77.702395 | 0.064037 |

### 11.4.4 나이 구하기

생년월일을 정보를 활용해 나이를 구해봅시다. 판다스 시리즈에서 dt 함수를 사용하면 해당 값에 대한 년, 월, 일 등을 따로 분리해낼 수 있습니다. 다음과 같이 dt.year를 활용하여 출생년도를 계산하고 2021년 기준으로 나이를 구하겠습니다.

| data['age'] = 2021 - pd.to\_datetime(data['dob']).dt.year # 나이 계산 |
| --- |

그리고 불필요한 변수들은 모두 제거해줍니다. 카드번호나 위도, 경도 정보는 피처 엔지니어링에만 필요하고 예측하는 데는 사용할 필요가 없습니다.

| data.drop(['cc\_num','lat','long', 'merch\_lat', 'merch\_long' ,'dob','merch\_coord','cust\_coord'], axis = 1, inplace=True) # 변수 제거  data.head() # 상위 5행 출력 |
| --- |

|  | trans\_date\_trans\_time | category | amt | gender | city\_pop | is\_fraud | amt\_z\_score | cat\_z\_score | distance | distance\_z\_score | age |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 43,466 | misc\_net | 4.970 | F | 3,495 | 0.00000 | -0.66 | -0.69 | 78.77 | 0.03 | 33.00 |
| 1 | 43,466 | grocery\_pos | 107.230 | F | 149 | 0.00000 | 0.3 | 0.3 | 30.22 | -1.48 | 43.00 |
| 2 | 43,466 | entertainment | 220.110 | M | 4,154 | 0.00000 | 1.29 | 2.87 | 108.10 | 1.16 | 59.00 |
| 3 | 43,466 | gas\_transport | 45.000 | M | 1,939 | 0.00000 | -0.13 | -1.05 | 95.69 | 0.82 | 54.00 |
| 4 | 43,466 | misc\_pos | 41.960 | M | 99 | 0.00000 | -0.57 | 1.31 | 77.70 | 0.06 | 35.00 |

### 11.4.5 새 변수 만들기

이제 기존 변수에서 새로운 변수를 만들어내는 작업은 모두 끝났습니다. 남아있는 object 변수인 category와 gender는 더미 변수로 변환하여 활용하겠습니다.

| data = pd.get\_dummies(data, columns = ['category','gender'], drop\_first = True) # 더미 변수 변환 |
| --- |

trans\_date\_trans\_time은 예측에 필요하지는 않으나, 훈련셋을 분리시키는 데 활용하려고 아직 제외시키지 않았습니다. 그래도 모델링하는 데는 활용하지 않을 것이기 때문에 index에 들어가도록 처리해줍니다.

| data.set\_index('trans\_date\_trans\_time', inplace=True) # 인덱스 설정  data.head() # 상위 5행 출력 |
| --- |

| trans\_date\_trans\_time | amt | city\_pop | is\_fraud | amt\_z\_score | cat\_z\_score | distance | distance\_z\_score | age | category\_food\_dining | category\_gas\_transport | ... | category\_health\_fitness | category\_home | category\_kids\_pets | category\_misc\_net | category\_misc\_pos | category\_personal\_care | category\_shopping\_net | category\_shopping\_pos | category\_travel | gender\_M |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2019-01-01 00:00:18 | 5 | 3,495 | 0.000 | -1 | -1 | 78.77382 | 0.0 | 33.0 | 0.00 | 0.00 | ... | 0.00 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2019-01-01 00:00:44 | 107 | 149 | 0.000 | 0 | 0 | 30.21662 | -1.48 | 43.00 | 0.00 | 0.00 | ... | 0.00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2019-01-01 00:00:51 | 220 | 4,154 | 0.000 | 1 | 3 | 108.10291 | 1.16 | 59.00 | 0.00 | 0.00 | ... | 0.00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 2019-01-01 00:01:16 | 45 | 1,939 | 0.000 | 0 | -1 | 95.68512 | 0.82 | 54.00 | 0.00 | 1.00 | ... | 0.00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 2019-01-01 00:03:06 | 42 | 99 | 0.000 | -1 | 1 | 77.70240 | 0.064037 | 35 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |

## 11.5 모델링 및 평가하기

신용카드의 이상거래를 감지하는 상황을 상상하면 지금까지 발생한 거래 데이터를 기반으로 모델을 학습시키고, 그 모델을 이용하여 앞으로 일어나는 거래에 대한 이상 여부를 예측해야 합니다. 이러한 이유로 이번에는 train\_test\_split()을 사용하지 않고, 특정 날짜를 기준으로 하여 훈련셋과 시험셋을 나누겠습니다. 이 데이터는 2019년 1월부터 2020년 12월까지의 데이터를 담고 있는데, 이중 2020년 7월~12월 데이터를 시험셋으로 활용하겠습니다.

| train = data[data.index < '2020-07-01'] # 훈련셋 설정 test = data[data.index >= '2020-07-01'] # 시험셋 설정 |
| --- |

이렇게 시험셋을 정의하면 시험셋의 비율은 다음과 같이 약 28%입니다.

| len(test) / len(data) # 시험셋 비율 확인 |
| --- |

0.2837738623640543

이제 각 셋에서 종속변수를 분리하여 X와 y가 붙은 훈련셋과 시험셋을 만듭니다.

| X\_train = train.drop('is\_fraud', axis = 1) # X\_train 설정 X\_test = test.drop('is\_fraud', axis = 1) # X\_test 설정 y\_train = train['is\_fraud'] # y\_train 설정 y\_test = test['is\_fraud'] # y\_test 설정 |
| --- |

이제 모델링에 사용할 LightGBM 라이브러리를 lgb라는 별칭으로 불러옵니다.

| import lightgbm as lgb # 임포트 |
| --- |

XGBoost와 마찬가지로 LightGBM 학습을 시키는 함수로는 회귀, 분류, train() 함수(11.7절 참고)가 있습니다. 가장 먼저 하이퍼파라미터 튜닝 없이 기본값으로 분류 함수를 이용하여 모델링하겠습니다. 방법은 기존과 동일하니 예측하는 코드까지 한 번에 처리하겠습니다.

| model\_1 = lgb.LGBMClassifier(random\_state = 100) # 모델 객체 생성  model\_1.fit(X\_train, y\_train) # 학습  pred\_1 = model\_1.predict(X\_test) # 예측 |
| --- |

그리고 이번에는 이전에 사용했던 평가 방법인 정확도(accuracy\_score), 혼동 행렬(confusion\_matrix), 분류 리포트(classification\_report)에 추가로, ROC AUC 점수(roc\_auc\_score)까지 활용하겠습니다.

| from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report, roc\_auc\_score # 임포트 |
| --- |

가장 먼저 accuracy\_score() 함수로 정확도를 확인해봅시다.

| accuracy\_score(y\_test, pred\_1) # 정확도 확인 |
| --- |

0.9964749144410561

약 99.6%로 매우 정확한 예측률을 보여줍니다. 하지만 10장에서 설명했던 바처럼 기존 종속 변수의 구성이 한쪽으로 지나치게 치우친 비대칭 데이터에서는 이런 높은 예측률이 큰 의미는 없습니다. 이미 is\_fraud가 0인 경우가 약 99%이기 때문에, 이보다 작은 정확도가 나온다면 무언가 크게 잘못된 겁니다.

그럼 다음으로 confusion\_matrix() 함수를 이용해서 혼동 행렬을 출력해 어떤 오류가 많이 발생했는지, 참 양성True Positive은 얼마나 많은지 확인하겠습니다.

| print(confusion\_matrix(y\_test, pred\_1)) # 혼동행렬 확인 |
| --- |

[[522626 1023]

[ 830 1182]]

❶ 참 양성은 1182건입니다. 1종 오류와 2종 오류가 비슷한 수준으로 발생했습니다. ❷ 거짓 음성False Negative(2종 오류)이 830건으로, 절반이 좀 안 되는 이상거래를 잡아내지 못했습니다. 반대로 정상거래 건이 이상거래로 잘못 감지된 ❸ 거짓 양성False Positive(1종 오류)은 1023건입니다.

다음은 classification\_report() 함수로 분류 리포트를 확인해봅시다.

| print(classification\_report(y\_test, pred\_1)) # 분류 리포트 확인 |
| --- |

precision recall f1-score support

0 1.00 1.00 1.00 523649

1 0.54 0.59 0.56 2012

accuracy 1.00 525661

macro avg 0.77 0.79 0.78 525661

weighted avg 1.00 1.00 1.00 525661

정밀도(precision)는 0.54, 재현율(recall)은 0.59, F-1 점수(f1-score)는 0.56입니다. 이상거래 감지에서는 셋 중 어떤 값이 가장 중요할까요? 지나치게 예민하게 반응하여 많은 이상거래를 감지해낸다면 실제로 이상거래를 감지하는 경우도 많아지겠지만, 동시에 정상거래를 이상거래로 인식하고 거래를 지연시키고 고객에게 확인하는 일 역시 많아질 겁니다. 반대로 덜 예민하게 반응하는 모델이라면 정상 거래 건을 안정적으로 처리하는 경우가 많아지겠지만, 그중 이상거래가 포함되는 경우 또한 늘게 됩니다. 둘 중 한쪽을 선택해야 한다면, 정상 거래를 한 고객이 확인 절차를 거치는 불편함이 조금 더 발생하더라도 이상거래를 조금 더 포착하는 게 중요할 겁니다. 즉, 실제 이상거래를 얼마나 많이 예측해냈는지를 의미하는 재현율이 더 중요합니다.

여기서 잠깐 모델이 예민하게 반응한다는 것의 의미를 조금 구체적으로 살펴보겠습니다. 로지스틱 회귀에서부터 부스팅 모델에 이르기까지, 모든 이진분류Binary Classification에서 예측값은 0부터 1까지에 해당하는 숫자였습니다. 지금까지 보아온 예측값들은 0과 1이었지만, 실제로는 0.2322, 0.678221 같은 소수점 형태입니다. 다만 0.5를 기준으로 반올림하여 0과 1로 보여준 겁니다. 0과 1로 나누는 기준을 0.5가 아닌 다른 값으로 적용하면 예측 결과 또한 달라지며, 이 기준점이 민감도를 결정합니다. 예를 들어 0.2를 기준으로 하면 0.2보다 큰 값은 모두 1로 분류되어 매우 민감하게 반응합니다. 반면 기준점을 0.8로 하면 0.8 이하는 모두 0으로 분류되어 상대적으로 둔감한 예측 결과를 보일 겁니다. 예측하는 코드에서 predict() 대신 predict\_proba()를 사용하면 0과 1이 아닌 소수점 형태의 결과를 얻을 수 있습니다(이는 다른 알고리즘에서도 동일하게 적용됩니다).

| proba\_1 = model\_1.predict\_proba(X\_test) # 예측 |
| --- |

proba\_1을 출력하면 다음과 같이 2열 X 데이터수만큼의 소수점값이 나옵니다.

| proba\_1 # 예측 결과 출력 |
| --- |

array([[9.99944636e-01, 5.53639581e-05],

[9.99912715e-01, 8.72847343e-05],

[1.00000000e+00, 0.00000000e+00],

...,

[9.99890545e-01, 1.09455330e-04],

[9.99624117e-01, 3.75882664e-04],

[9.99868882e-01, 1.31118203e-04]])

각 행은 각 데이터에 대한 예측값이며, 그중 첫 번째 값은 0에 대한 예측값, 두 번째 값은 1에 대한 예측값입니다. 즉 둘의 합은 1입니다. 우리가 알고 싶은 것은 1일 가능성을 예측하는 것이기 때문에 두 번째 값들만 나타나도록 다음과 같이 인덱싱할 수 있습니다.

| proba\_1[:,1] # 1에 대한 예측 결과만 출력 |
| --- |

array([5.53639581e-05, 8.72847343e-05, 0.00000000e+00, ...,

1.09455330e-04, 3.75882664e-04, 1.31118203e-04])

이 부분(1일 가능성 예측값)만 사용할 것이기 때문에 proba\_1에 덮어쓰겠습니다.

| proba\_1 = proba\_1[:,1] # 예측 결과 재설정 |
| --- |

이제 이 값을 0.2과 0.8을 기준으로 하여 0과 1로 나누어보고 혼동 행렬과 분류 리포트를 비교하겠습니다.

| proba\_int1 = (proba\_1 > 0.2).astype('int') # 0.2 기준으로 분류 proba\_int2 = (proba\_1 > 0.8).astype('int') # 0.8 기준으로 분류 |
| --- |

>를 사용하여 특정 기준보다 더 큰지를 확인하면 True/False로 결과가 나오므로, astype('int')를 붙여서 0과 1로 변환했습니다.

이제 각각을 confusion\_matrix()와 classification\_report() 함수에 적용해보겠습니다.

| print(confusion\_matrix(y\_test, proba\_int1)) # 혼동행렬 확인 |
| --- |

[[522007 1642]

[ 657 1355]]

| print(classification\_report(y\_test, proba\_int1)) # 분류 리포트 확인 |
| --- |

precision recall f1-score support

0 1.00 1.00 1.00 523649

1 0.45 0.67 0.54 2012

accuracy 1.00 525661

macro avg 0.73 0.84 0.77 525661

weighted avg 1.00 1.00 1.00 525661

0.2를 기준점으로 한 결과입니다. 기존 결과(0.5 기준)보다 훨씬 많은 1355건의 참 양성이 보이면서, 동시에 거짓 양성 또한 1642건으로 늘었습니다. 그렇다보니 정밀도는 기존보다 떨어지고 재현율은 많이 개선된 모습입니다.

다음은 0.8을 기준으로 한 결과입니다.

| print(confusion\_matrix(y\_test, proba\_int2)) # 혼동행렬 확인 |
| --- |

[[522910 739]

[ 1007 1005]]

| print(classification\_report(y\_test, proba\_int2)) # 분류 리포트 확인 |
| --- |

precision recall f1-score support

0 1.00 1.00 1.00 523649

1 0.58 0.50 0.54 2012

accuracy 1.00 525661

macro avg 0.79 0.75 0.77 525661

weighted avg 1.00 1.00 1.00 525661

0.8을 기준으로 하면 ❶ 참 양성이 1005건으로 조금 줄었지만 ❷ 거짓 양성은 739건으로 줄어 크게 개선되었고, 반대로 ❸ 거짓 음성이 1007로 많아졌습니다. 즉, 정상거래 건을 이상거래로 잘못 의심하는 경우는 줄었으나, 반대로 이상거래를 놓치는 경우가 너무 많아졌습니다. 때문에 정밀도가 높아진 반면 재현율은 다소 떨어집니다. 이 분석의 목적에 따라 재현율을 가장 중요하게 보아야겠지만 그렇다고 정밀도가 지나치게 낮아지는 것도 조심해야 하기 때문에 적절한 기준점을 잡아야 합니다.

만약 여러 모델을 만들고 그 성능을 비교한다면 주로 어떤 값을 살펴보는 게 좋을까요? 여기서는 재현율을 가장 우선시해서 봐야겠지만, 기준점에 따라 재현율이 달라질 수 있고 최적의 기준점은 모델에 따라 달라진다는 문제가 있습니다. 예를 들어 모델 A에서는 0.4에서 높은 재현율과 적절한 정밀도를 얻을 수 있는 반면, 모델 B에서는 0.3이 최적의 기준점이 될 수 있습니다. 그래서 재현율을 기준으로 모델간의 성능 비교는 아주 명쾌하지는 않습니다. AUC라는 또 다른 지표를 활용하면 이러한 문제를 쉽게 해결할 수 있습니다. 우선 AUC를 이해하려면 ROC 곡선ROC Curve을 알아야 합니다. ROC 곡선은 민감도와 특이도 개념을 활용합니다.

<용어/>

**ROC 곡선과 AUC**

이진분류 모델을 평가하는 방법으로, 기준점에 영향을 받지 않기 때문에 여러 모델을 비교할 때 요긴하게 사용됩니다. AUC는 ROC 곡선의 아래 면적을 의미하고, 0.5~1 사이의 값을 지니며 높을수록 좋은 모델입니다.

</>

* : 민감도라고 부릅니다(재현율과 수식이 같습니다). 다시 한번 개념을 설명하자면, 실제 1인 것 중 얼마만큼 제대로(1로) 예측되었는지를 뜻합니다. 1에 가까울수록 좋은 수치입니다.
* : 특이도라고 부릅니다. 분모는 실젯값이 0인 경우이고, 분자는 잘못 예측된 1입니다. 즉, 실제 0인 것 중 얼마만큼이 1로 잘못 예측되었는지를 말합니다. 0에 가까울수록 좋은 수치입니다.

만약 모델이 100% 정확하게 예측했다면, 즉 실제 1인 것을 모두 1이라고 예측하고, 실제 0인 것 중 1이라고 잘못 예측한 것이 없다면, TPR(민감도)은 1, FPR(특이도)은 0이 됩니다. 아래 그림에서 왼쪽 상단의 점이 이에 해당합니다.

<그림/>



</>

하지만 현실적으로 이런 상황은 보기 어려우며, 주로 ❶ ❷ ❸ 곡선 같은 모습을 보입니다. ❷번 곡선을 예로 들어 그래프를 해석해보자면, FPR이 0.1인 지점에서는 TPR이 0.4 정도입니다. 즉 잘못 1로 분류한 경우가 별로 없지만, 제대로 1로 예측하는 경우도 40% 정도입니다. 조금 더 오른쪽으로 움직여서 FPR이 0.6인 지점은 TPR이 0.9입니다. 실젯값 1에 대해 90%를 정확히 예측해냈지만, 잘못 예측된 실젯값 0들이 60%나 나왔습니다. FPR을 희생하면 더 나은 TPR을 얻을 수 있고, 반대로 TPR을 포기하면 안정적인(0에 가까운) FPR을 얻을 수 있습니다. 그리고 한 모델에 대해서 이렇게 다양한 값이 나올 수 있는 것은 위에서 설명한 기준점을 달리할 수 있기 때문입니다. 즉, 이 그래프는 모든 가능한 기준점들에 대한 TPR과 FPR에 대한 결과를 보여줍니다. 좋은 모델의 ROC 곡선은 ❶번처럼 FPR을 아주 조금 희생해도 쉽게 높은 TPR을 얻을 수 있어야 합니다. 반대로 ❸번 곡선은 모델의 FPR을 아무리 높여봐야 TPR이 별로 높아지지 않아 안 좋은 모델이라 볼 수 있습니다. ROC 곡선에서 가장 안 좋은 경우는 빨간 점선의 형태로, 전혀 예측을 하지 못하는 경우입니다. ROC 곡선은 빨간 점선보다 더 아래쪽으로는 만들어지지 않습니다.

AUC는 Area Under the ROC Curve의 약자로, ROC 곡선의 아래쪽에 해당하는 면적을 의미합니다. 아래의 예시 그림에서 파란색 선이 ROC 곡선라면, 그아래 색이 칠해진 부분이 AUC입니다.



사각형 전체의 면적을 1로 두었을 때, AUC의 최댓값은 1이며, 최솟값은 빨간 점선 아랫부분인 0.5가 됩니다.

이번에는 AUC를 조금 다른 각도에서 설명하겠습니다. 만약 다음과 같은 모델이 있다고 합시다. X 축은 기준점Threshold이고 y축은 해당 기준점에 대한 데이터 개수입니다. ❷번 선은 1에 해당하는 데이터를 나타내고 ❶번 선은 0에 해당하는 데이터를 의미합니다.

<그림/>



참 양성(TP), 참 음성(TN),

거짓 음성(FN), 거짓 양성(FP)

기준선

</>

이 그림에서는 예측값을 0.5를 기준으로 0과 1을 나누면 참 양성(TP), 참 음성(TN), 거짓 음성(FN), 거짓 양성(FP)이 어떻게 나누어지는지를 보여줍니다. 0.5를 기준으로 오른쪽은 1로 예측되기 때문에, 0.5 우측에 해당하는 연두색선의 데이터는 잘못 분류된, 즉 거짓 양성에 해당하며, 반대로 0.5의 왼쪽에 존재하는 빨간색선의 데이터는 거짓 음성이 되는 겁니다. 대략 이정도로 구분할 수 있는 모델의 AUC를 0.7이라고 하겠습니다. 그렇다면 아주 완벽하게 예측하는 모델의 그림도 살펴봅시다.

<그림/>



참 음성 / 참 양성

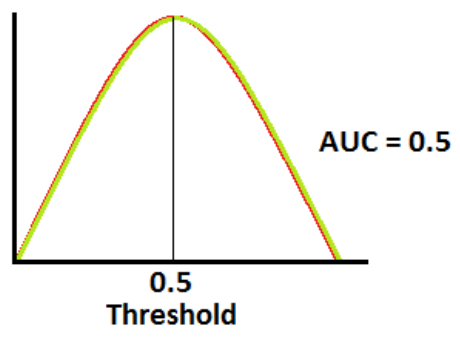
AUC = 1

기준점

</>

여기에서는 ❷번 선과 ❶번 선이 겹치는 지점이 없기 때문에, 적절한 기준점을 잡는다면(여기에서는 0.5) 아주 완벽하게 예측할 수 있습니다. 이 경우는 AUC가 1이 됩니다. 마지막으로 가장 나쁜 모델의 예도 살펴보겠습니다.

<그림/>



기준점

</>

두 선이 겹쳐서, 사실상 아무런 예측력도 가지지 못하는 모델입니다. 기준점을 어느 쪽으로 움직이든 도무지 유의미한 예측이 불가능합니다. 이때 AUC는 최솟값인 0.5가 나오게 됩니다.

즉, AUC는 모델이 얼마나 0과 1을 잘 분리하는지 기준점에 상관없이 보여주고 있기 때문에, 여러 모델을 비교하기에 아주 객관적인 지표로 사용될 수 있습니다. 그럼 위에서 구한 예측값들을 사용하여 AUC를 구하겠습니다. AUC는 다른 평가지표와는 다르게 0과 1로 나누어진 값이 아닌 소수점 값, 즉 predict\_proba()로 예측된 값을 그대로 사용해야 합니다.

| roc\_auc\_score(y\_test, proba\_1) # 정확도 확인 |
| --- |

0.9031275287192038

약 0.9031 정도로 상당히 높은 AUC가 나왔습니다. 보통 0.8 이상이면 상당히 높은 편입니다만, AUC 또한 정확도와 마찬가지로 종속변수가 한쪽으로 편향된 때 자연스럽게 높게 나오는 경향이 있으므로 0.9031이라고 해서 반드시 매우 좋다고 해석할 수는 없습니다.

5장에서부터 지금까지 다양한 이진분류에 대한 평가지표를 알아보았습니다. 전체적으로 복습하는 차원에서 다시 한번 간단하게 살펴봅시다.

▽ 이진분류 평가 지표

| 지표 | 설명 | 다루는 장 |
| --- | --- | --- |
| 정확도 | 전체 예측값 중 몇 %나 정확하게 맞추었는지에 대한 지표 | 5장 |
| 오차행렬 | 실제 참/거짓, 예측 참/거짓을 2X2 테이블로 표현한 것 | 7장 |
| 정밀도 | 양성으로 예측된 것 중 참 양성의 비율 | 10장 |
| 재현율 | 실제 양성 중 참 양성으로 예측 한 비율 | 10장 |
| F1 스코어 | 정밀도와 재현율의 조화평균 | 10장 |
| 민감도 | 참 양성과 거짓 음성으로 예측된 것 중 참 양성의 비율 | 11장 |
| 특이도 | 거짓 양성과 참 음 음성으로 예측된 것 중 참 음성의 비율 | 11장 |
| AUC | 민감도와 특이도를 사용하여 그린 ROC 커브의 아래면적으로, 모델이 데이터를 얼마나 명료하게 분류하는지를 나타내는 지표 | 11장 |

## 11.6 하이퍼파라미터 튜닝 : 랜덤 그리드 서치

랜덤 그리드 서치를 활용하여 하이퍼파라미터 튜닝을 하겠습니다. 랜덤 그리드 서치는 기존 그리드 서치와 유사하나, 주어진 하이퍼파라미터값들의 모든 조합을 시도하지 않고 랜덤으로 일부만 선택하여 모델링합니다. 모든 조합에 대해 모델링하면 너무 많은 시간이 소요될 수 있기 때문에, 랜덤 그리드 서치를 통해 더 넓은 영역의 하이퍼파라미터 값을 더 짧은 시간에 다양하게 활용할 수 있는 장점이 있습니다. 물론 하이퍼파라미터 값들의 조합 전부를 확인하는 것보다는 덜 좋은 모델이 나올 수 있으니 시간적인 부분을 고려한다면 장점이 더 크다고 볼 수 있습니다. 랜덤 그리드 서치도 사이킷런의 model\_selection에서 불러올 수 있습니다.

| from sklearn.model\_selection import RandomizedSearchCV |
| --- |

이번에는 더 많은 하이퍼파라미터를 활용하겠습니다.

| params = {  'n\_estimators': [100, 500, 1000], # 반복횟수   'learning\_rate': [0.01, 0.05, 0.1, 0.3], # 러닝 레이트   'lambda\_l1': [0,10,20,30,50], # L1 정규화   'lambda\_l2': [0,10,20,30,50], # L2 정규화  'max\_depth': [5,10,15,20], # 최대 깊이  'subsample': [0.6, 0.8, 1]} # 서브샘플 비율 |
| --- |

새로 등장한 하이퍼파라미터는 lambda\_l1과 lambda\_l2입니다. 이들은 각각 L1 정규화와 L2 정규화를 의미합니다. 이는 LightGBM뿐만 아니라 XGBoost에서도 사용할 수 있으며, 이 개념을 선형 회귀에 접목한 라쏘 회귀lasso regression(L1 정규화 적용), 릿지 회귀ridge regression(L2 정규화 적용)도 있습니다. 선형 회귀의 경우를 들어 L1 정규화와 L2 정규화의 개념을 간단히 알아보겠습니다.

<용어/>

**L1 정규화과 L2 정규화**

둘 다 매개변수에 패널티를 가해서 그 영향력을 감소시키는 방법으로, 오버피팅을 방지하는 목적으로 쓰입니다. L21은 일부 매개변수가 0이 되도록 패널티를 가할 수 있기 때문에 피처 셀렉션 효과도 있습니다.

</>

| **피처 셀렉션**  피처 셀렉션은 머신러닝 학습에 사용할 피처를 선택하는 것을 의미합니다. 기본적으로 많은 피처(독립변수)가 있는 것이 풍부한 데이터이기 때문에 머신러닝 학습에 있어서 좋다고 볼 수 있지만, 언제나 많은 피처가 좋은 것은 아닙니다. 적절하지 못한 피처가 섞인 경우는 오히려 예측 결과가 안 좋아질 수 있기 때문에, 특정 피처만을 선택하여 모델링했을 때 더 좋은 결과가 나오기도 합니다. 여기서 다루는 L1 정규화는 피처 셀렉션의 역할을 해주기 때문에 불필요한 변수들을 자동으로 학습에서 배제합니다. 반면, 회귀 분석은 모든 피처를 다 사용하기 때문에 피처 셀렉션을 지원하는 라쏘 회귀를 사용하거나, 포워드 셀렉션(forward selection), 백워드 제거(backward elimination), 재귀적 피처 제거(recursive reature elimination)과 같은 방법을 사용할 수도 있습니다. |
| --- |

예를 들어 선형 회귀 모델을 만들면 각 변수에 대한 기울기, 즉 계수가 구해집니다. L1 정규화과 L2 정규화는 이 계수에 패널티를 부가하여 너무 큰 계수가 나오지 않도록 강제하는 방법입니다. 이는 각 변수에 대한 계수를 조금 더 작게 반영함으로써 오버피팅을 방지하는 데 유용합니다. 우선 선형 회귀의 오찻값에 대한 수식을 살펴보겠습니다.

좌항의 는 실젯값, 는 예측값으로, 이들의 차를 제곱한 값을 오차로 정의합니다. 이를 풀어보면 우항과 같은데, 는 좌항과 동일하고 - 이후의 값은 예측값을 풀어낸 모습입니다. x는 각 변수에 들어가는 값이고, w는 해당 변수에 대한 기울기인 계수입니다. 즉 변수에 들어가는 값과 기울깃값을 곱한 것이 임을 의미합니다. 아래의 단순한 1차 방정식과 비교하면 더 쉽게 이해할 수 있습니다. Y = AX일 때, Y는 예측값인 이고 기울기인 A는 위 수식에서의 w, X는 위 수식에서의 x와 같습니다.

L1 정규화과 L2 정규화는 이 수식에 패널티를 주는 항목을 추가합니다.

L1 정규화 =

맨 우측에 추가된 항을 보면 기울기인 w에 절댓값이 붙어진 채로 더해집니다. 이 수식이 오차에 대한 수식임을 생각하면, 기울기의 절댓값만큼 오차에 더해진다는 의미입니다. 즉, 큰 기울깃값들이 많으면 오차가 더 큰 것으로 간주하여 최적의 모델을 찾을 때 자연스럽게 배제됩니다. 그리고 시그마 앞에 붙은 (람다라고 읽음)는 이 패널티에 대한 가중치를 의미합니다. 이는 우리가 직접 정의할 수 있는 하이퍼파라미터로, 높은 값을 넣으면 그 패널티를 더 크게 만들고, 작은 값을 넣으면 패널티의 역할 또한 작아집니다. L2 정규화는 L1 정규화과 매우 유사한데 추가된 항의 w에 절댓값 대신 제곱을 사용하여 기울기의 마이너스 부호를 해결합니다.

L2 정규화 =

L1 정규화과 L2 정규화는 모두 값이 커질수록 강한 패널티를 부과하기 때문에, 그 크기가 커질수록 변수들에 대한 계수의 절댓값이 작은 모델이 나오게 됩니다. 그러나 이 둘 사이에 결정적인 차이가 있는데, 가 커짐에 따라 계수의 절댓값이 감소하는 방식이 다르게 적용됩니다. 아래의 예시는 가 변함에 따라서 계수(w)가 어떻게 달라지는가를 보여줍니다. x축이 1/로 되어 있기 때문에 왼쪽으로 갈 수록 큰 를 의미합니다. 각 선들은 각기 다른 변수들에 대한 계수를 나타냅니다.

<그림/>



L2 정규화 L1 정규화

w

1/ 1/

</>

가 커질수록, 즉 x축의 왼쪽으로 이동할수록 모든 계수가 0에 가까워지는 것을 볼 수 있는데, L2 정규화에서는 계수의 절댓값들이 0에 가깝게 수렴하지만 0이 되지는 않습니다. 반면 L1 정규화에서는 특정 값에서 특정 변수들의 계수는 0이 되어버립니다. 계수가 0이 된다는 것은 해당 변수의 영향력이 사라짐을 의미합니다. 때문에 L1 정규화는 를 조절함으로써 몇몇 불필요한 변수를 제거해버리는 피처 셀렉션의 효과를 가져올 수 있습니다. L2 정규화의 경우, 계수가 0에 가깝게 수렴하지만 미미하게나마 그 변수의 영향력이 존재하기 때문에 모든 변수들이 모델에 반영됩니다.

이제 다시 랜덤 그리드 서치로 돌아와서 모델링을 하겠습니다.

| model\_2 = lgb.LGBMClassifier(random\_state = 100) # 모델 객체 생성 rs = RandomizedSearchCV(model\_2, param\_distributions=params, n\_iter= 30, scoring = 'roc\_auc', random\_state=100, n\_jobs=-1) # 랜덤그리드서치 객체 생성 |
| --- |

대부분 그리드 서치와 동일하나 scoring에 roc\_auc를 지정해주었으며, n\_iter라는 하이퍼파라미터를 추가했습니다. n\_iter는 몇 번을 반복할 것인가를 정의하는데 이 숫자는 곧 전체 하이퍼파라미터의 조합 중 몇 개를 사용할 것인지를 정합니다. 여기서는 단 30개의 하이퍼파라미터 조합만을 사용하여 모델링해보았습니다. rs를 가지고 모델을 훈련시키며, 동시에 소요 시간을 확인하는 코드도 함께 넣었습니다.

| start = time.time() # 시작시간 설정 rs.fit(X\_train, y\_train) # 학습 print(time.time() - start) # 총 소요 시간 확인 |
| --- |

6099.671524763107

약 100분만에 모든 모델링이 끝났습니다. 우선 어떤 하이퍼파라미터 조합이 가장 좋았는지부터 확인하겠습니다.

| rs.best\_params\_ # 최적 하이퍼파라미터 확인 |
| --- |

{'lambda\_l1': 0,

'lambda\_l2': 20,

'learning\_rate': 0.05,

'max\_depth': 15,

'n\_estimators': 1000,

'subsample': 1}

우선 subsample은 1로, 일부가 아닌 전체를 다 사용했으며, lambda\_l1 또한 0으로 L1 정규화가 적용되지 않았습니다. 해당 모델을 가지고 predict\_proba로 예측 후 AUC를 확인하겠습니다.

| rs\_proba = rs.predict\_proba(X\_test) # 예측 roc\_auc\_score(y\_test, rs\_proba[:,1]) # 정확도 확인 |
| --- |

0.9976715031170279

약 0.995로 하이퍼파라미터 튜닝 이전인 0.9366보다 훨씬 높아진 모습을 보여줍니다. 이제 0.2를 기준으로 하여 0과 1로 분류한 뒤 confusion\_matrix와 classification\_report를 확인하겠습니다.

| rs\_proba\_int = (rs\_proba[:,1] > 0.2).astype('int') # 0.2 기준으로 분류 print(confusion\_matrix(y\_test, rs\_proba\_int)) # 혼동행렬 확인 |
| --- |

[[522514 1135]

[ 508 1504]]

참 양성과 거짓 음성은 이전보다 조금 나아진 수준이고, 거짓 양성는 훨씬 많이 좋아졌습니다. 즉, 더 많은 이상거래 건을 감지해 내면서, 정상거래를 이상거래로 잘못 판단하는 경우 또한 많이 줄었다는 것을 의미합니다. classification\_report() 함수로 확인하면 이를 더 명료하게 볼 수 있습니다.

| print(classification\_report(y\_test, rs\_proba\_int)) # 분류 리포트 확인 |
| --- |

precision recall f1-score support

0 1.00 1.00 1.00 523649

1 0.57 0.75 0.65 2012

accuracy 1.00 525661

macro avg 0.78 0.87 0.82 525661

weighted avg 1.00 1.00 1.00 525661

재현율이 0.75까지 높아졌으면서도 F1-점수는 0.65입니다. 하이퍼파라미터 튜닝 이전의 모델에서 구할 수 있는 그 어떤 F1-점수보다도 높은 값이 나왔습니다.

## 11.7 LightGBM의 train() 함수 사용하기

XGBoost와 LightGBM에서는 기본적으로 회귀와 분류의 fit() 함수를 활용하여 분석 목적에 맞게 모델링할 수 있습니다. 여기에 한 가지 더하여 train() 함수를 활용해서도 모델링할 수 있습니다. train()이 기본이 되는 알고리즘이고, 회귀와 분류의 fit() 함수는 train() 함수를 활용하는 형태라고 이해하면 됩니다. 이 둘의 주요 차이점은 다음과 같습니다.

▽ train()와 LGBMRegressor.fit()/LGBMClassifier.fit() 차이점

|  | lgb.train() | lgb.LGBMRegressor.fit()  lgb.LGBMClassifier.fit() |
| --- | --- | --- |
| 검증셋 | 모델링 과정에 검증셋 지원 | 모델링에 검증셋 포함하지 않음 |
| 데이터셋 | 데이터프레임을 별도의 포맷으로 변환 필요 | 별도의 포맷 필요 없이 자동적으로 처리됨 |
| 하이퍼파라미터 | 무조건 지정해야 함 | 기본값으로도 모델링 가능 |
| 사이킷런과 연동 (그리드 서치, CV 등) | 불가 | 가능 |

위 내용은 이제 코드를 통해 하나씩 더 구체적으로 확인하게 될 것이니 지금 당장 100% 이해하지 못하셔도 괜찮습니다.

우선 train() 함수에서는 검증셋을 함께 사용할 수 있습니다. 부스팅 모델을 학습할 때 수많은 트리를 계속해서 만들어나가는데, 일반적으로 훈련셋에 대한 특정 metrics(예를 들어 auc나 rmse)을 기준으로 평가하며 다음 트리를 개선해나갑니다. 검증셋을 사용한다는 것은, 트리를 계속해서 만들어나갈 때 훈련셋이 아닌 훈련에 사용되지 않은 검증셋으로 평가를 해가며 모델을 개선해나가는 겁니다. 그리고 이렇게 모델링이 완료되면, 아직 전혀 사용하지 않은 시험셋으로 최종 예측 및 평가를 합니다. 분할 과정을 그림으로 나타내면 다음과 같습니다.

<그림/>

▽ 훈련셋, 검증셋, 시험셋 분할 과정



</>

기존에 훈련셋과 시험셋을 약 8:2~6:4 수준으로 나누었는데, 검증셋은 일반적으로 훈련셋의 일부를 취하여 만들어냅니다. train\_test\_split() 함수를 두 번 사용하여 위의 세 가지 셋을 분리해낼 수 있는데, 여기서는 날짜 기준으로 훈련셋과 시험셋을 나누었으니 검증셋도 다음과 같은 날짜 기준으로 분리하겠습니다.

* 훈련셋 : 2019년 1월 ~ 2019년 12월
* 검증셋 : 2020년 1월 ~ 2020년 6월
* 시험셋 : 2020년 7월 ~ 2020년 12월

| train = data[data.index < '2020-01-01'] # 훈련셋 설정 val = data[(data.index >= '2020-01-01') & (data.index < '2020-07-01')] # 검증셋 설정 test = data[data.index >= '2020-07-01'] # 시험셋 설정 |
| --- |

그리고 각 셋에서 독립변수와 종속변수를 분리하겠습니다.

| X\_train = train.drop('is\_fraud', axis = 1) # X\_train 설정 X\_val = val.drop('is\_fraud', axis = 1) # X\_val 설정 X\_test = test.drop('is\_fraud', axis = 1) # X\_test 설정 y\_train = train['is\_fraud'] # y\_train 설정 y\_val = val['is\_fraud'] # y\_val 설정  y\_test = test['is\_fraud'] # y\_test 설정 |
| --- |

이제 위의 비교표에서 두 번째 차이점인 데이터셋 부분을 다룰 차례입니다. train() 함수로 학습하려면 LightGBM에서 제시하는 고유한 데이터셋 형태를 취해야 합니다. 데이터셋 형태를 변환하는 것은 LightGBM의 Dataset() 함수로 처리할 수 있습니다. 여기에는 독립변수 데이터프레임와 종속변수 데이터를 다음과 같이 입력해주면 됩니다. 시험셋은 모델링할 때 사용하지 않으니 훈련셋과 검증셋에 대해서만 처리해줍니다.

| d\_train = lgb.Dataset(X\_train, label=y\_train) # 데이터 타입 변환  d\_val = lgb.Dataset(X\_val, label=y\_val) # 데이터 타입 변환 |
| --- |

다음은 세 번째 차이점인 하이퍼파라미터 지정입니다. train() 함수는 하이퍼파라미터가 입력되지 않으면 에러가 나기 때문에 반드시 하이퍼파라미터를 입력해야 합니다. 우선 우리가 위의 그리드 서치에서 확보한 가장 좋은 하이퍼파라미터셋을 그대로 이용하고, 여기에 ‘metric’을 auc로 지정하여 다음과 같이 추가해줍니다.

| params\_set = rs.best\_params\_ # 최적 파라미터 설정 params\_set['metrics'] = 'auc' # 평가 기준 추가 |
| --- |

어떤 하이퍼파라미터들이 들어 있는지 다시 한번 확인하고 가겠습니다.

| params\_set # 하이퍼파라미터 확인 |
| --- |

{'lambda\_l1': 0, L1 정규화

'lambda\_l2': 20, L1 정규화

'learning\_rate': 0.05, 러닝 레이터

'max\_depth': 15, 최대 깊이

'metrics': 'auc', 평가 기준

'n\_estimators': 1000, 반복 횟수

'subsample': 1} 서브샘플 비율

이제 train() 함수로 훈련해주고, 그 결과를 model\_3에 저장하겠습니다. 모델 객체를 생성한 뒤에 fit() 함수로 훈련시키는 LGBMClassifier/LGBMRegressor와는 달리 train() 함수는 객체 생성 시점에 한 번에 학습까지 완료됩니다.

| model\_3 = lgb.train(params\_set, d\_train, valid\_sets=[d\_val], early\_stopping\_rounds=100, verbose\_eval=100) |
| --- |

Training until validation scores don't improve for 100 rounds

[100] valid\_0's auc: 0.97237

[200] valid\_0's auc: 0.988574

[300] valid\_0's auc: 0.993031

[400] valid\_0's auc: 0.992582

Early stopping, best iteration is:

[327] valid\_0's auc: 0.993459

이 함수에서는 각 트리가 생성될 때마다 검증셋에 대한 결과물을 출력해줍니다. 여기에서는 n\_estimators가 1000으로 들어갔기 때문에 총 1000개의 결과물이 출력되어야 합니다. 1000줄은 불필요하게 길기 때문에 verbose\_eval를 사용해 줄였습니다. verbose\_eval은 출력물을 특정 간격으로 보여줍니다. 100으로 설정하여 100번째, 200번째, 300번째 등의 결과만 보여줍니다.

불필요한 학습 시간을 제한하는 하이퍼파라미터로 early\_stopping\_rounds가 있습니다. early\_stopping\_rounds에 해당하는 숫자만큼 진행했음에도 더는 개선이 보이지 않으면 굳이 그 이상의 학습을 진행하지 않는 겁니다. 개선 여부에 대한 평가는 검증셋에 대한 AUC로 결정됩니다. 여기서는 100을 설정해주었기 때문에 100개의 트리가 더 만들어졌음에도 개선이 되지 않으면 중지합니다. 출력 결과에서 제일 마지막 두 줄을 살펴볼까요?

<실행결과/>

Early stopping, best iteration is:

[327] valid\_0's auc: 0.993459

</>

327번째 트리에서 가장 좋은 auc를 보여주었고, 이 뒤로 100개의 트리를 더 시도했으나 개선되지 않아서 여기서 멈췄다는 의미입니다. 이제 model\_3를 활용하여 X\_test에 대한 예측을 하겠습니다.

| pred\_3 = model\_3.predict(X\_test) # 예측 |
| --- |

train() 함수로 훈련된 모델은 predict\_proba() 역할을 predict()가 대신합니다. 즉, predict()를 사용하여 예측하여도, 0과 1이 아닌 소수점값들을 얻을 수 있습니다. 그럼 이 예측값에 대한 AUC를 확인하겠습니다.

| roc\_auc\_score(y\_test, pred\_3) # 정확도 확인 |
| --- |

0.991352108964131

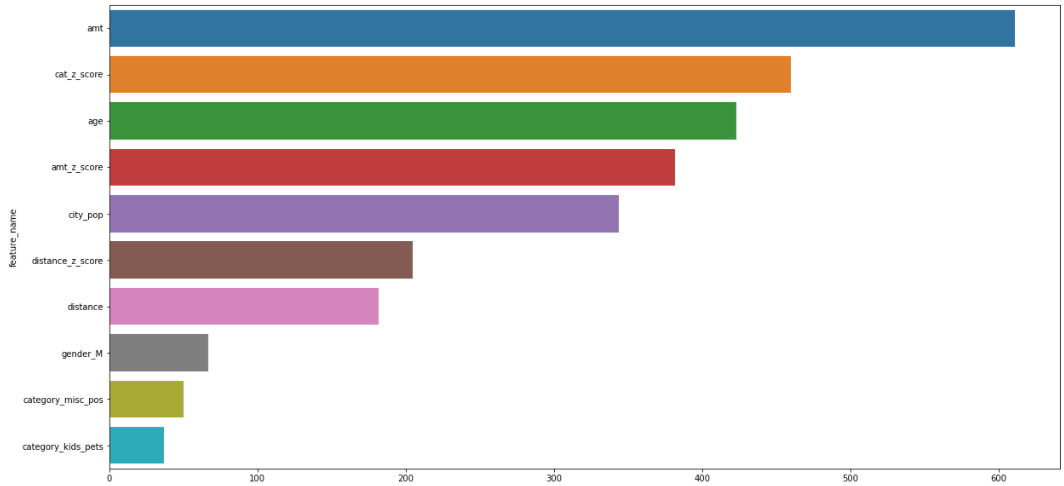
약 0.9913으로, LGBMClassifier를 사용했을 때 보다는 조금 낮은 값이 나왔습니다. 훈련에 사용된 데이터셋의 범위가 달라졌기 때문에 차이가 발생한 부분도 있고, LGBMClassifier에 기본적으로 설정된 하이퍼파라미터의 값과 train() 함수에 반영된 기본 하이퍼파라미터 값들의 차이에 의한 것이기도 합니다.

장단점을 비교해보자면 LGBMClassifier/LGBMRegressor를 사용하면 사이킷런의 그리드 서치와 연동되고 데이터 포맷을 변경할 필요가 없기 때문에 훨씬 편리합니다. train() 함수를 사용하면 검증셋을 활용할 수 있어서 조금 더 신뢰할 만한 결과를 보여주기도 합니다.

또 하나의 사소한 차이점으로는 feature importance를 확인할 때 코드가 약간 다릅니다. LGBMClassifier/LGBMRegressor으로 훈련된 모델에서는 feature\_importances\_()를 사용해야 하며, train() 함수로 훈련된 모델에서는 feature\_importance()를 사용해야 합니다.

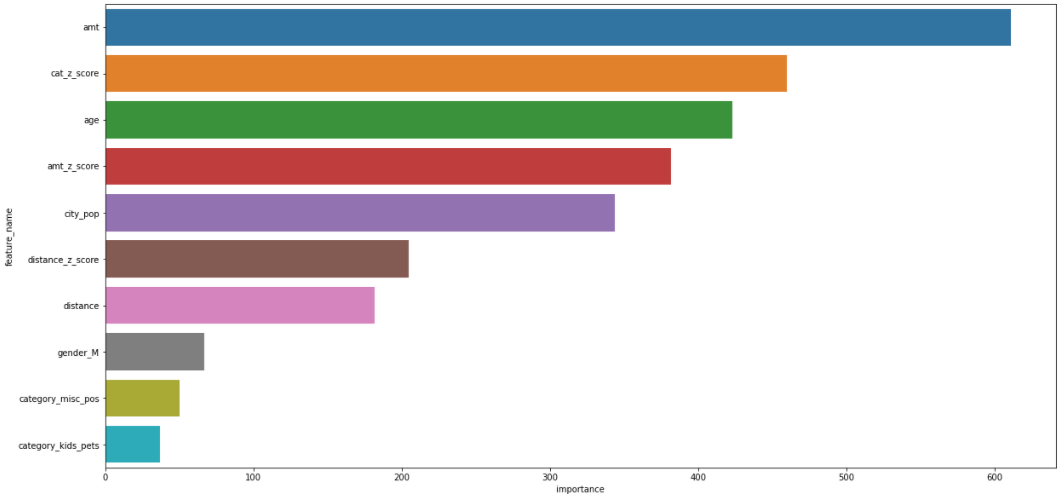
LGBMClassifier/LGBMRegressor의 feature\_importances\_()를 model\_1에 사용해보겠습니다.

| feature\_imp = pd.DataFrame({'feature\_name': X\_train.columns, 'importance': model\_1.feature\_importances\_}).sort\_values('importance', ascending = False) # 중요 변수 정리 plt.figure(figsize=(20, 10)) # 그래프 크기 정의 sns.barplot(x="importance", y="feature\_name", data=feature\_imp.head(10)) plt.show() # 바그래프 출력 |
| --- |



이번에는 train의 feature\_importance()를 model\_3에 사용해보겠습니다.

| feature\_imp\_3 = pd.DataFrame(sorted(zip(model\_3.feature\_importance(), X\_train.columns)), columns=['Value','Feature']) # 중요 변수 정리  plt.figure(figsize=(20, 10)) # 그래프 크기 정의 sns.barplot(x="importance", y="feature\_name", data=feature\_imp.head(10)) plt.show() # 바그래프 출력 |
| --- |



두 함수는 본질적으로 같은 알고리즘이기 때문에 이 둘을 별도로 사용할 필요는 없습니다. 여기서 소개하는 이유는 어디까지나 학습의 목적이기 때문에, 실제로 사용할 때는 둘 중 하나만 사용하면 됩니다. 사이킷런과의 호환성을 고려하면 LGBMClassifier/LGBMRegressor가 더 편리합니다.

## 11.8 이해하기 : LightGBM

LightGBM은 XGBoost와 마찬가지로 트리 기반 모델의 최신 알고리즘 중 하나이므로, 지난 장에서 설명한 내용을 거의 포함합니다. 그래서 여기에서는 XGBoost와 비교하는 것으로 간단하게 LightGBM을 설명할 수 있습니다.

둘의 가장 큰 차이점은 ‘트리의 가지를 어떤 식으로 뻗어나가는가’입니다. XGBoost는 균형 분할Level-wise tree growth 방식으로, 각 노드에서 같은 깊이를 형성하도록 한층 한층 밑으로 내려옵니다.

▽ XGBoost(균형 분할 방식)



반면 LightGBM은 이러한 전제를 거부하고, 특정 노드에서 뻗어나가는 가지가 모델의 개선에 더 도움이 된다면 계속하여 다음과 같은 그림처럼 진행될 수 있습니다.

▽ LightGBM(리프 중심 트리 분할 방식)



그래서 LightGBM은 속도가 훨씬 빠르게 진행될 수 있으며, 복잡성은 더 증가하고, 오버피팅 문제를 야기할 가능성 또한 더 높습니다. 물론 이러한 문제점은 하이퍼파라미터 튜닝으로 어느 정도 극복할 수 있습니다.

또 다른 차이는 속도면에서 LightGBM이 XGBoost에 비해 월등히 앞선다는 겁니다. 그러나 이것은 어디까지나 모델링에 CPU를 활용하는 것을 전제로 합니다. 만약 GPU를 사용한다면 XGBoost가 더 빠른 속도를 보이고, 사실 LightGBM의 기본 패키지에서는 GPU를 지원하지 않아 별도의 추가 패키지를 설치해야 하는 번거로움 마저 있습니다. 그러나 딥러닝 알고리즘이 아닌 때는 CPU를 사용하여 모델링하는 경우가 많기 때문에, 전반적으로 LightGBM이 XGBoost보다 나은 성능을 보여준다고 볼 수 있습니다.

XGBoost 대비 LightGBM의 장점은 다음과 같습니다.

* 빠른 학습 및 예측
* 더 적은 메모리 사용
* 데이터셋 자동 변환 및 최적 분할

## 학습 마무리

#### 되짚어보기

11.1 카드 거래 내역 데이터셋를 LightGBM로 학습해 이상거래를 예측했습니다.

11.2 판다스, 넘파이, 맷플롯립, 시본 라이브러리를 임포트했습니다. 프로젝트에 쓸 예제 데이터셋을 불러옵니다.

11.3 불필요한 변수를 삭제하고, 변수의 속성을 적절하게 변경해주었습니다.

11.4 결제 금액, 카테고리, 거리, 나이 등을 기반으로 새로운 변수를 만들었습니다.

11.5 LightGBM으로 모델링하여 0.90이라는 AUC를 얻었습니다. 일반적으로 좋은 수치라 볼 수 있습니다.

11.6 최적의 하이퍼파라미터를 찾는 데 랜덤 그리드 서치를 사용했습니다. 그 결과 AUC를 0.99까지 높였습니다.

11.7 train()는 LightGBM의 모델 훈련 역할을 하는 근본적인 함수입니다. 알고리즘 자체는 LGBMClassifier/LGBMRegressor와 차이가 없습니다.



#### 과제

L1, L2 정규화는 선형 회귀와도 결합되어 사용됩니다. sklearn.linear\_model에서 임포트할 수 있으니, 4장에서 사용한 보험 데이터셋을 대상으로 해당 모델도 사용해보세요. train() 함수는 XGBoost에서도 지원합니다. 10장을 train() 함수로도 연습해봅시다. 10장에서 다룬 내용을 XGBoost를 사용해 모델링하고, LightGBM과 학습 속도, 예측률을 비교해봅시다.

#### 유의할 점

연습 목적이므로 데이터를 훈련셋, 검증셋, 시험셋으로 나누기 전에 피처 엔지니어링을 수행했습니다. 그렇지만 실제 비즈니스에 적용하려면 훈련셋에 대해서 모든 피처 엔지니어링을 마친 뒤에 동일한 기준으로 검증셋과 시험셋에도 피처 엔지니어링을 적용해야 합니다. 예를 들어 금액에 대한 Z점수를 구하는 것은 훈련셋에 한해서 계산되어야 하며, 이때 각 고객의 사용금액에 대한 정보(평균, 표준편차 등)를 별도의 테이블로 데이터 서버에 저장한 후, 새로운 데이터는 기존의 고객별 평균, 표준편차 등을 사용하여 Z점수를 계산해야 합니다.

#### 관련 모델

1. **CatBoost**  
   패키지: from catboost import CatBoostRegressor, CatBoostClassifier)  
   XGBoost, LightGBM과 같은 경사 부스팅 기법을 사용한 모델로, 카테고리형 데이터에 대한 우수한 처리 기술을 가지고 있습니다. 구글 코랩에서 기본적으로 지원하는 알고리즘은 아니기 때문에, 아래의 코드를 사용해 설치해야 합니다.  
   !pip install catboost

#### 핵심 용어 정리

1. **LightGBM** : XGBoost보다 한단계 더 진화한 형태로, 리프 중심 트리 분할을 사용하여 더 빠르고 정확한 예측을 보여줍니다.
2. **리프 중심 트리 분할**leaf-wise tree growth : XGBoost와 LightGBM의 중요한 차이점으로, 동일한 레벨로 노드를 확장하지 않고 불규칙적으로 노드를 뻗어나가기 때문에 더욱 빠르고 높은 예측율을 보이나 오버피팅을 유의해야합니다.
3. **L1, L2 정규화** : 둘 다 매개변수에 패널티를 가해서 그 영향력을 감소시키는 방법으로, 오버피팅을 방지하는 목적으로 쓰입니다. L1 정규화는 일부 매개변수가 0이 되도록 패널티를 가할 수 있기 때문에 피처 셀렉션 효과도 있습니다.
4. **ROC 곡선과 AUC** : 이진분류 모델을 평가하는 방법으로, 기준점에 영향을 받지 않기 때문에 여러 모델을 비교할 때 요긴하게 사용됩니다. AUC는 ROC 곡선의 아래 면적을 의미하고, 0.5~1 사이의 값을 지니며 높을수록 좋은 모델입니다.

#### 새로운 함수와 라이브러리

* **time.time()** : 현재 시간을 호출합니다.
* **lightgbm.DatasetDataset()** : LightGBM에서 학습 가능한 형태의 데이터로 변환해주는 함수로, train()을 통해 학습할 때 필요합니다.
* **LightGBM모델.feature\_importances\_()** : 학습된 LightGBM에서 변수의 중요도를 보여줍니다.

## 연습문제

1. 다음 AUC의 설명 중 옳지 않은 것은?

① 이진분류의 예측 결과에 사용된다.

② AUC가 높을수록 더 잘 분류된 것이다.

③ 최댓값은 1, 최솟값은 0까지 나올 수 있다.

④ Area Under the Curve의 약자로, ROC 커브 아래의 면적이다.

2. 다음 중 이진분류의 예측 결과를 위한 평가 지표가 아닌 것은?

① AUC

② Recall

③ F1 Score

④ RMSE

3. 다음 L1, L2 정규화에 대한 설명 중 옳지 않은 것은?

① L1, L2 정규화는 모두 패널티를 가해서 변수의 영향력을 감소시킨다.

② 오버피팅을 방지하기 위한 목적으로 사용한다.

③ 선형회귀에도 적용할 수 있으며, L1 정규화가 적용된 것은 라쏘 회귀이다.

④ L2 정규화는 피처 셀렉션의 효과도 있다.

#### 정답 및 해설

1. 3

③ 최댓값은 1, 최솟값은 0까지 나올 수 있다. ← 최솟값은 0.5입니다.

2. 4

④ RMSE ← 연속형 변수의 예측 결과를 위한 평가지표입니다.

3. 4

④ L2 정규화는 피처 셀렉션의 효과도 있다. ← 피처 셀렉션의 효과가 있는 것은 L1 입니다.