

# 채무불이행 분류 예측 모델

스탁캐스터골드버튼가시조

이름 : 2019147018 김희연  
2019147025 신재욱  
2020147003 정혜진

## Contents

- I. 데이터 탐색
- II. 전처리 방법
- III. 모델 소개
- IV. 분류 예측
  - I. 전처리 적용
  - II. 모델 학습
  - III. 최종 예측

# 데이터 탐색

## 01. EDA

### (1) 데이터 불러오기

- EDA 및 전처리 과정 용이하게 하기 위해 train data(train\_x + train\_y)와 test data(test\_x + test\_y)를 생성

```
In [1]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

In [2]: train_x = pd.read_csv('data/train/train.csv')
train_y = pd.read_csv('data/train/train_label.csv')
test_x = pd.read_csv('data/test/test.csv')
test_y = pd.read_csv('data/test/test_label.csv')

In [3]: # EDA 및 전처리 과정 용이하게 하기 위해 합침
train = pd.merge(train_x, train_y)

In [4]: train.shape
Out[4]: (13228, 20)

In [5]: # EDA 및 전처리 과정 용이하게 하기 위해 합침
test = pd.merge(test_x, test_y)

In [6]: test.shape
Out[6]: (13229, 20)
```

# 데이터 탐색

## 01. EDA

### (2) train 데이터 파악 - train.head()

- train 데이터셋의 상위 5개 데이터 파악

In [7]: `train.head()`

Out[7]:

	index	gender	car	reality	child_num	income_total	income_type	edu_type	family_type	house_type	DAY_S_BIRTH	DAY_S_EMPLOYED	FLAG_MOBIL	work_phone	phone	email	occyp_type	family_size	begin_month	credit
0	0	F	Y	Y	0	202500.0	Pensioner	Secondary / secondary special	Married	House / apartment	-19031	365243	1	0	0	0	Nan	2	-53	1
1	1	F	N	N	1	157500.0	Working	Higher education	Married	House / apartment	-15773	-309	1	0	1	0	Sales staff	3	-26	0
2	2	M	Y	N	0	135000.0	Working	Secondary / secondary special	Married	House / apartment	-13483	-1816	1	1	1	0	Laborers	2	-9	1
3	3	F	Y	N	2	112500.0	Working	Secondary / secondary special	Married	House / apartment	-12270	-150	1	0	1	0	Security staff	4	-12	1
4	4	M	Y	Y	1	225000.0	Working	Secondary / secondary special	Married	House / apartment	-16175	-2371	1	0	0	0	Drivers	3	-3	1

▲ train data

# 데이터 탐색

## 01. EDA

### (3) 데이터 type 확인 - train.info()

- train data의 dtype 확인

```
In [8]: # train data의 dtype 확인  
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>  
Int64Index: 13228 entries, 0 to 13227  
Data columns (total 20 columns):  
 #   Column           Non-Null Count  Dtype     
---  --  
 0   index            13228 non-null   int64    
 1   gender           13228 non-null   object   
 2   car              13228 non-null   object   
 3   reality          13228 non-null   object   
 4   child_num        13228 non-null   int64    
 5   income_total     13228 non-null   float64  
 6   income_type      13228 non-null   object   
 7   edu_type         13228 non-null   object   
 8   family_type      13228 non-null   object   
 9   house_type       13228 non-null   object   
 10  DAYS_BIRTH       13228 non-null   int64    
 11  DAYS_EMPLOYED    13228 non-null   int64    
 12  FLAG_MOBIL       13228 non-null   int64    
 13  work_phone       13228 non-null   int64    
 14  phone             13228 non-null   int64    
 15  email             13228 non-null   int64    
 16  occyp_type        9096 non-null   object   
 17  family_size       13228 non-null   int64    
 18  begin_month       13228 non-null   int64    
 19  credit            13228 non-null   int64    
dtypes: float64(1), int64(11), object(8)  
memory usage: 2.1+ MB
```

▲ train data

# 데이터 탐색

## 01. EDA

### (4) null값 확인 - train.isnull().sum()

- train data의 null값 존재 여부 확인
- 'occyp\_type' null값 존재, 전처리 필요

```
In [9]: # train data의 null값 존재 여부 확인 -> 'occyp_type' null값 존재 -> 전처리 불가피  
train.isnull().sum()
```

```
Out[9]: index          0  
gender          0  
car             0  
reality         0  
child_num       0  
income_total    0  
income_type     0  
edu_type        0  
family_type     0  
house_type      0  
DAYS_BIRTH      0  
DAYS_EMPLOYED   0  
FLAG_MOBIL      0  
work_phone      0  
phone           0  
email           0  
occyp_type      4132  
family_size     0  
begin_month     0  
credit          0  
dtype: int64
```

▲ train data

# 데이터 탐색

## 01. EDA

### (5) 데이터 분포 확인 - train.describe()

- train data의 수치형 column에 대한 분포 확인

```
In [10]: # train data의 수치형 column에 대한 분포 확인
train.describe()
```

Out[10]:

	index	child_num	income_total	DAY_S_BIRTH	DAY_S_EMPLOYED	FLAG_MOBIL	work_phone	phone	email	family_size	begin_month	credit
count	13228.000000	13228.000000	1.322800e+04	13228.000000	13228.000000	13228.0	13228.000000	13228.000000	13228.000000	13228.000000	13228.000000	13228.000000
mean	6613.500000	0.428107	1.888212e+05	-15958.143408	59854.037496	1.0	0.230269	0.296946	0.089507	2.192773	-26.198292	0.878213
std	3818.739015	0.740691	1.041743e+05	4199.720373	138166.424477	0.0	0.421021	0.456930	0.285485	0.910463	16.579604	0.327052
min	0.000000	0.000000	2.700000e+04	-25152.000000	-15713.000000	1.0	0.000000	0.000000	0.000000	1.000000	-60.000000	0.000000
25%	3306.750000	0.000000	1.215000e+05	-19406.500000	-3153.000000	1.0	0.000000	0.000000	0.000000	2.000000	-40.000000	1.000000
50%	6613.500000	0.000000	1.575000e+05	-15521.500000	-1539.000000	1.0	0.000000	0.000000	0.000000	2.000000	-24.000000	1.000000
75%	9920.250000	1.000000	2.250000e+05	-12454.000000	-401.750000	1.0	0.000000	1.000000	0.000000	3.000000	-12.000000	1.000000
max	13227.000000	14.000000	1.575000e+06	-7705.000000	365243.000000	1.0	1.000000	1.000000	1.000000	15.000000	0.000000	1.000000

▲ train data

# 데이터 탐색

## 01. EDA

### (6) [credit] EDA - 신용도

- y value: 사용자의 신용카드 대금 연체를 기준으로 한 신용도 (0: 낮은 신용도, 1: 높은 신용도)
- 0과 1의 label 값의 비율 확인 → 1:7 의 class imbalance 상태, 추후 전처리 과정 필요

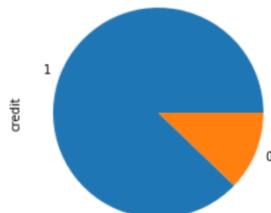
train

```
In [11]: # 0과 1의 label 값의 비율 확인 -> 1:7 의 class imbalance 상태 -> 추후 전처리 과정 불가피  
train['credit'].value_counts()
```

```
Out[11]: 1    11617  
0     1611  
Name: credit, dtype: int64
```

```
In [12]: # 데이터 분포도  
train['credit'].value_counts().plot.pie()
```

```
Out[12]: <AxesSubplot: ylabel='credit'>
```



test : null 값

▲ train data

# 데이터 탐색

## 01. EDA

### (7) [gender] EDA - 성별

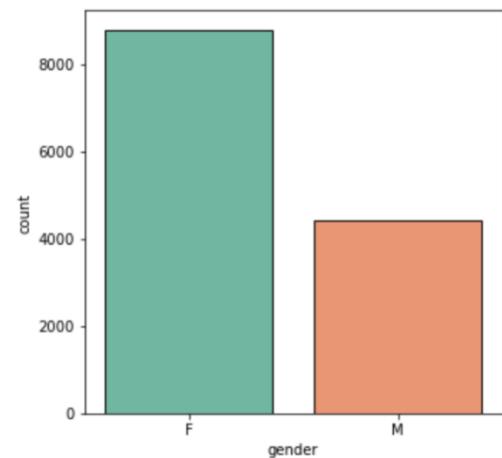
- train과 test 데이터의 countplot 분포가 비슷함
- train과 test 모두 Female의 비율이 높음

```
train

In [13]: train['gender'].value_counts()
Out[13]: F    8795
          M    4433
          Name: gender, dtype: int64

In [14]: # countplot 시각화
          fig, axes = plt.subplots(1, 1, figsize=(5, 5), sharey=True)
          sns.countplot(x='gender', data=train, palette="Set2", edgecolor='black')
          plt.suptitle('train data [gender] column distribution', fontsize=15, fontweight='bold', x=0.13, y=1.0, ha='left')
          plt.tight_layout()
          plt.show()
```

train data [gender] column distribution



▲ train data

# 데이터 탐색

## 01. EDA

### (7) [gender] EDA - 성별

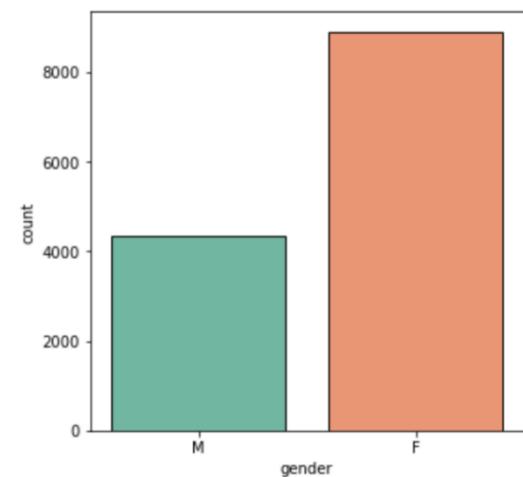
- train과 test 데이터의 countplot 분포가 비슷함
- train과 test 모두 Female의 비율이 높음

```
test

In [15]: test['gender'].value_counts()
Out[15]: F    8902
          M    4327
          Name: gender, dtype: int64

In [16]: # countplot 시각화
          fig, axes = plt.subplots(1, 1, figsize=(5, 5), sharey=True)
          sns.countplot(x='gender', data=test, palette="Set2", edgecolor='black')
          plt.suptitle('test data [gender] column distribution', fontsize=15, fontweight='bold', x=0.13, y=1.0, ha='left')
          plt.tight_layout()
          plt.show()
```

test data [gender] column distribution



▲ test data

# 데이터 탐색

## 01. EDA

### (8) [car] EDA - 차량 소유 여부

- train과 test 데이터의 countplot 분포가 비슷함
- train과 test 모두 N의 비율이 높음

```
train

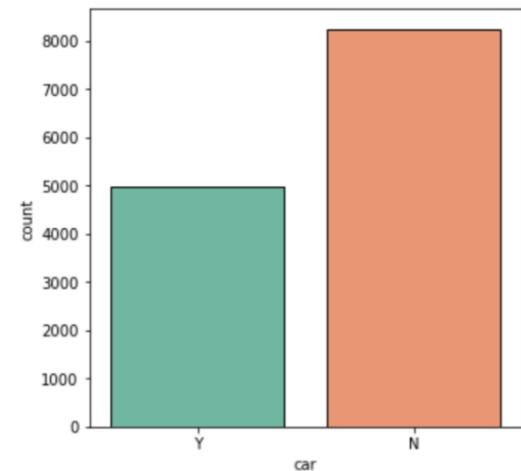
In [17]: train['car'].value_counts()
Out[17]: N    8251
          Y    4977
          Name: car, dtype: int64

In [18]: # countplot 시각화
fig, axes = plt.subplots(1, 1, figsize=(5, 5), sharey=True)

sns.countplot(x='car', data=train, palette="Set2", edgecolor='black')
plt.suptitle('train data [car] column distribution', fontsize=15, fontweight='bold', x=0.13, y=1.0, ha='left')

plt.tight_layout()
plt.show()
```

train data [car] column distribution



▲ train data

# 데이터 탐색

## 01. EDA

### (8) [car] EDA - 차량 소유 여부

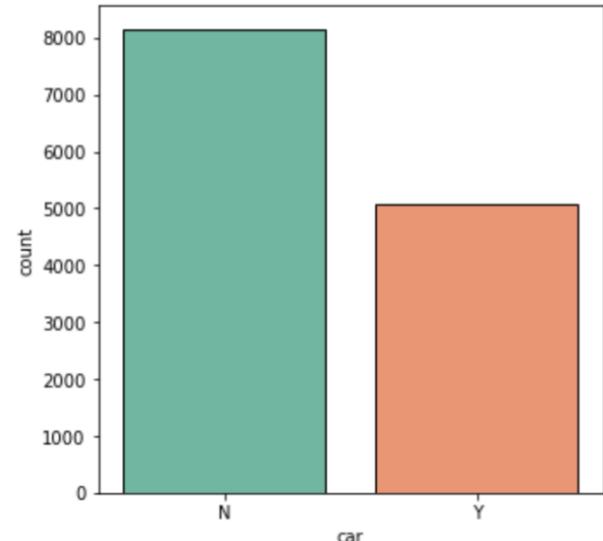
- train과 test 데이터의 countplot 분포가 비슷함
- train과 test 모두 N의 비율이 높음

```
test

In [19]: test['car'].value_counts()
Out[19]: N    8159
          Y    5070
          Name: car, dtype: int64

In [20]: # countplot 시각화
          fig, axes = plt.subplots(1, 1, figsize=(5, 5), sharey=True)
          sns.countplot(x='car', data=test, palette="Set2", edgecolor='black')
          plt.suptitle('test data [car] column distribution', fontsize=15, fontweight='bold', x=0.13, y=1.0, ha='left')
          plt.tight_layout()
          plt.show()
```

test data [car] column distribution



▲ test data

# 데이터 탐색

## 01. EDA

### (9) [reality] EDA - 부동산 소유 여부

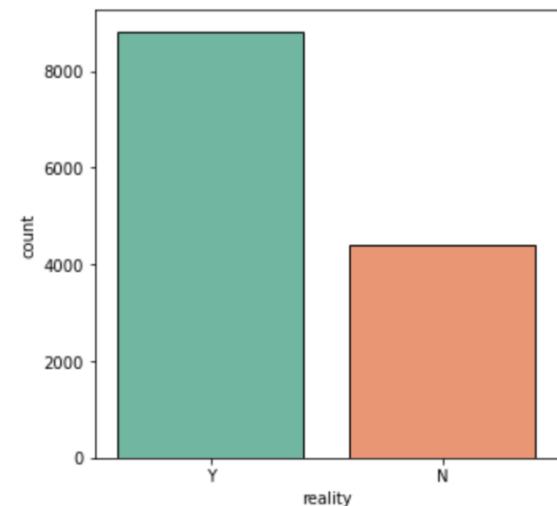
- train과 test 데이터의 countplot 분포가 비슷함
- 모두 Y의 비율이 더 높음

```
train

In [21]: train['reality'].value_counts()
Out[21]: Y    8823
          N    4405
          Name: reality, dtype: int64

In [22]: # countplot 시각화
          fig, axes = plt.subplots(1, 1, figsize=(5, 5), sharey=True)
          sns.countplot(x='reality', data=train, palette="Set2", edgecolor='black')
          plt.suptitle('train data [reality] column distribution', fontsize=15, fontweight='bold', x=0.13, y=1.0, ha='left')
          plt.tight_layout()
          plt.show()
```

train data [reality] column distribution



▲ train data

# 데이터 탐색

## 01. EDA

### (9) [reality] EDA - 부동산 소유 여부

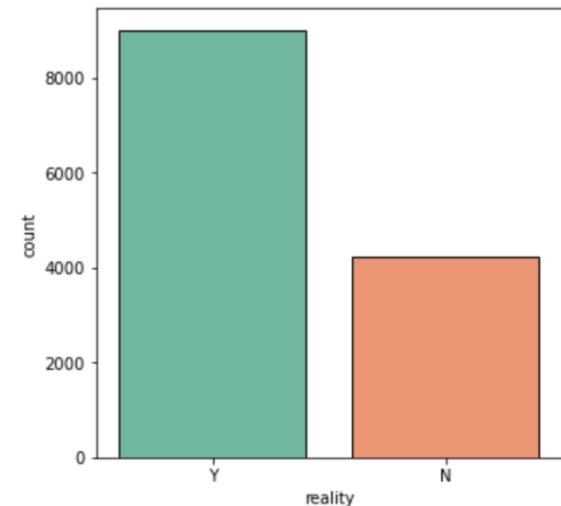
- train과 test 데이터의 countplot 분포가 비슷함
- 모두 Y의 비율이 더 높음

```
test  
In [23]: test['reality'].value_counts()
```

```
Out[23]: Y    9007  
          N    4222  
          Name: reality, dtype: int64
```

```
In [24]: # countplot 시각화  
  
fig, axes = plt.subplots(1, 1, figsize=(5, 5), sharey=True)  
sns.countplot(x='reality', data=test, palette="Set2", edgecolor='black')  
plt.suptitle('test data [reality] column distribution', fontsize=15, fontweight='bold', x=0.13, y=1.0, ha='left')  
plt.tight_layout()  
plt.show()
```

test data [reality] column distribution



▲ test data

# 데이터 탐색

## 01. EDA

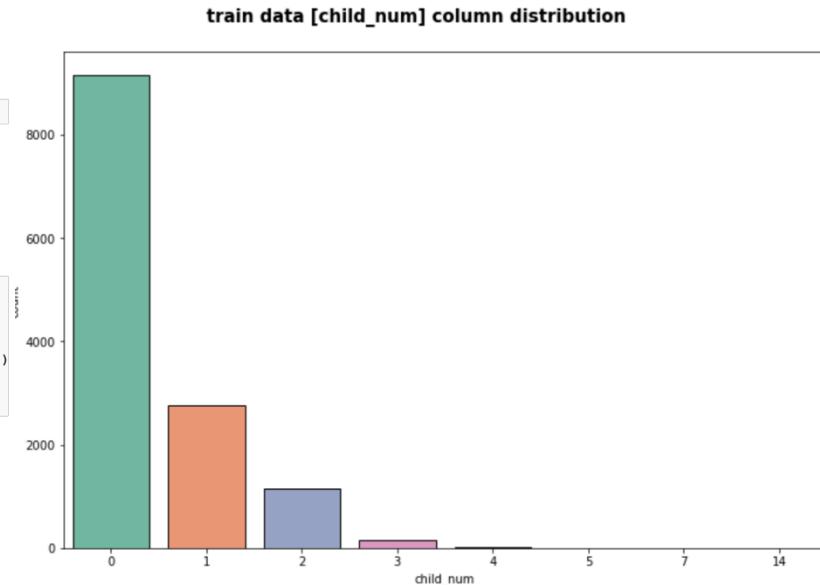
### (10) [child\_num] EDA - 자녀 수

- train과 test 데이터의 countplot 분포가 비슷함
- child가 적은 비율이 높음

```
train

In [25]: train['child_num'].value_counts()
Out[25]: 0    9144
1    2751
2    1150
3    148
4     27
5      5
14     2
7      1
Name: child_num, dtype: int64

In [26]: # countplot 시각화
fig, axes = plt.subplots(1,1, figsize=(10, 7), sharey=True)
sns.countplot(x='child_num', data=train, palette="Set2", edgecolor='black')
plt.suptitle('train data [child_num] column distribution', fontsize=15, fontweight='bold', x=0.25, y=1.0, ha='left')
plt.tight_layout()
plt.show()
```



▲ train data

# 데이터 탐색

## 01. EDA

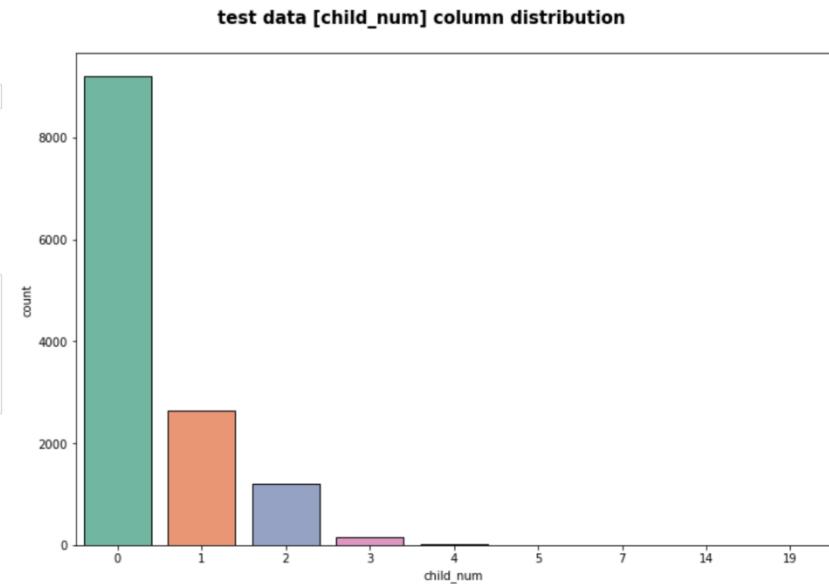
### (10) [child\_num] EDA - 자녀 수

- train과 test 데이터의 countplot 분포가 비슷함
- child가 적은 비율이 높음

```
test

In [27]: test['child_num'].value_counts()
Out[27]: 0    9196
1    2635
2    1212
3     158
4      20
5       5
7       1
19      1
14      1
Name: child_num, dtype: int64

In [28]: # countplot 시각화
fig, axes = plt.subplots(1,1, figsize=(10, 7), sharey=True)
sns.countplot(x='child_num', data=test, palette="Set2", edgecolor='black')
plt.suptitle('test data [child_num] column distribution', fontsize=15, fontweight='bold', x=0.25, y=1.0, ha='left')
plt.tight_layout()
plt.show()
```



▲ test data

# 데이터 탐색

## 01. EDA

### (11) [income\_total] EDA - 연간 소득

- train과 test 데이터의 boxplot 분포가 비슷함

train

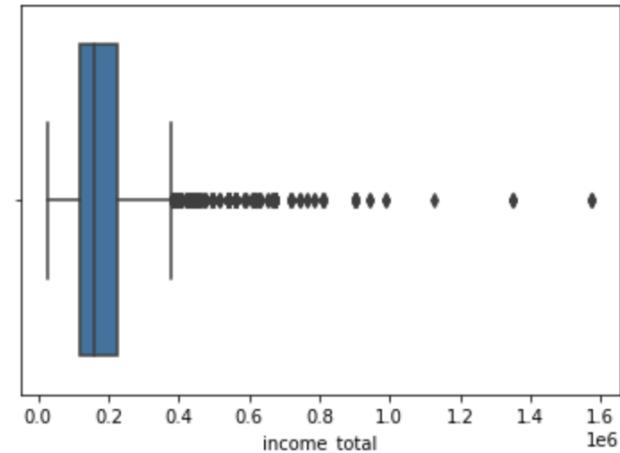
```
In [29]: train['income_total'].value_counts()
```

```
Out[29]:
```

135000.0	1578
157500.0	1123
225000.0	1114
112500.0	1100
180000.0	1058
...	
215100.0	1
240750.0	1
46948.5	1
177750.0	1
661500.0	1

Name: income\_total, Length: 216, dtype: int64

```
Out[30]: <AxesSubplot:xlabel='income_total'>
```



```
In [30]: # boxplot 시각화  
sns.boxplot(train['income_total'])
```

▲ train data

# 데이터 탐색

## 01. EDA

### (11) [income\_total] EDA - 연간 소득

- train과 test 데이터의 boxplot 분포가 비슷함

test

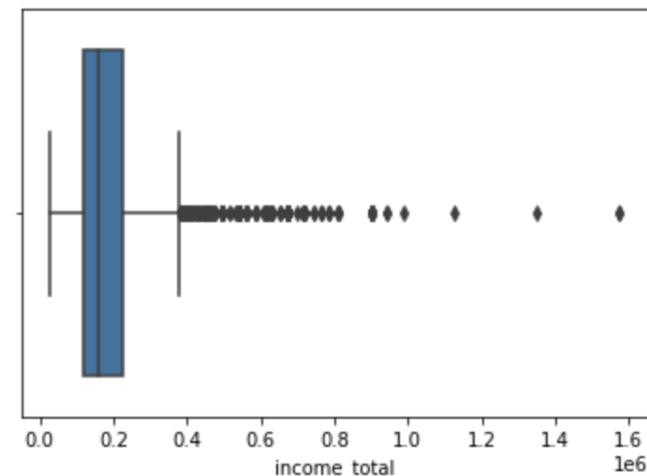
In [31]: `test['income_total'].value_counts()`

Out [31]:

135000.0	1586
180000.0	1167
157500.0	1110
112500.0	1078
225000.0	1056
...	
69372.0	1
101250.0	1
227250.0	1
60376.5	1
91530.0	1

Name: income\_total, Length: 218, dtype: int64

Out [32]: <AxesSubplot:xlabel='income\_total'>



▲ test data

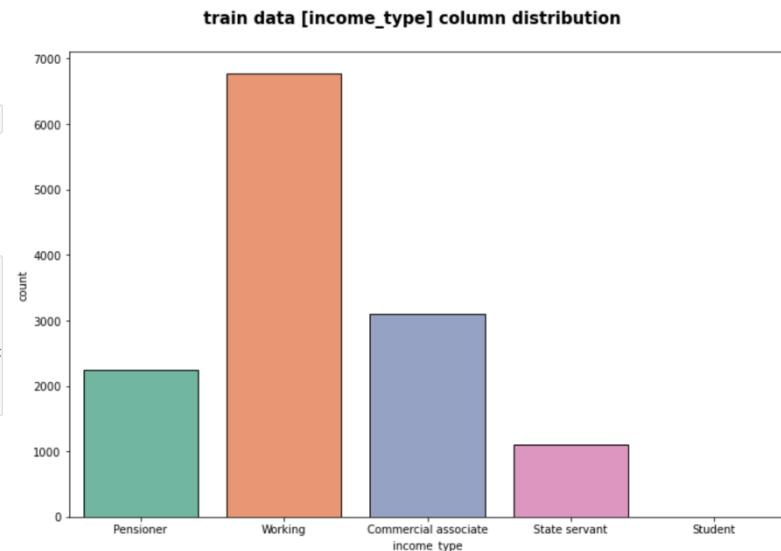
# 데이터 탐색

## 01. EDA

### (12) [income\_type] EDA - 소득 분류

- train과 test 데이터의 countplot 분포가 비슷함
- train과 test 모두 'working'의 비율이 높음

```
train  
  
In [33]: train['income_type'].value_counts()  
Out[33]: Working      6766  
Commercial associate  3108  
Pensioner           2249  
State servant        1103  
Student              2  
Name: income_type, dtype: int64  
  
In [34]: # countplot 시각화  
fig, axes = plt.subplots(1,1, figsize=(10, 7), sharey=True)  
sns.countplot(x='income_type', data=train, palette="Set2", edgecolor='black')  
plt.suptitle('train data [income_type] column distribution', fontsize=15, fontweight='bold', x=0.25, y=1.0, ha='left')  
plt.tight_layout()  
plt.show()
```



▲ train data

# 데이터 탐색

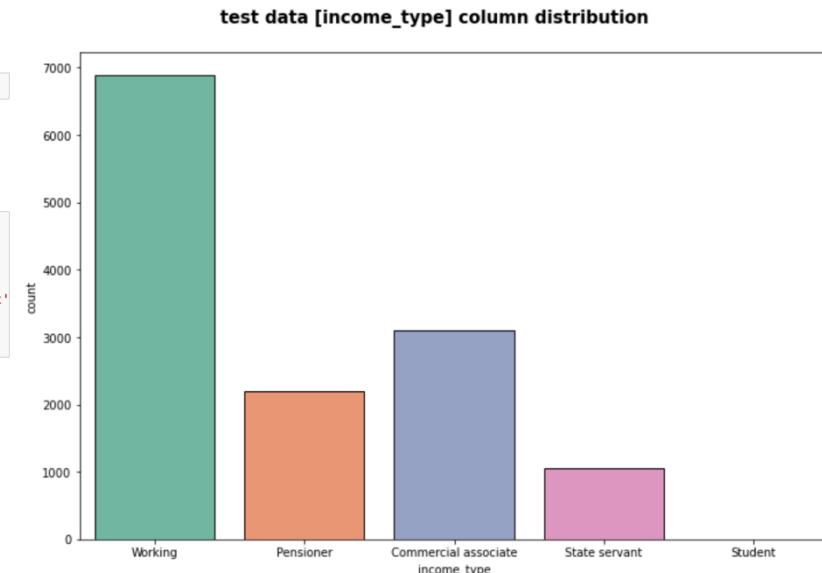
## 01. EDA

### (12) [income\_type] EDA - 소득 분류

- train과 test 데이터의 countplot 분포가 비슷함
- train과 test 모두 'working'의 비율이 높음

```
test
In [35]: test['income_type'].value_counts()
Out[35]: Working      6879
          Commercial associate    3094
          Pensioner        2200
          State servant     1051
          Student            5
          Name: income_type, dtype: int64

In [36]: # countplot 시각화
fig, axes = plt.subplots(1,1, figsize=(10, 7), sharey=True)
sns.countplot(x='income_type', data=test, palette="Set2", edgecolor='black')
plt.suptitle('test data [income_type] column distribution', fontweight='bold', x=0.25, y=1.0, ha='left')
plt.tight_layout()
plt.show()
```



▲ test data

# 데이터 탐색

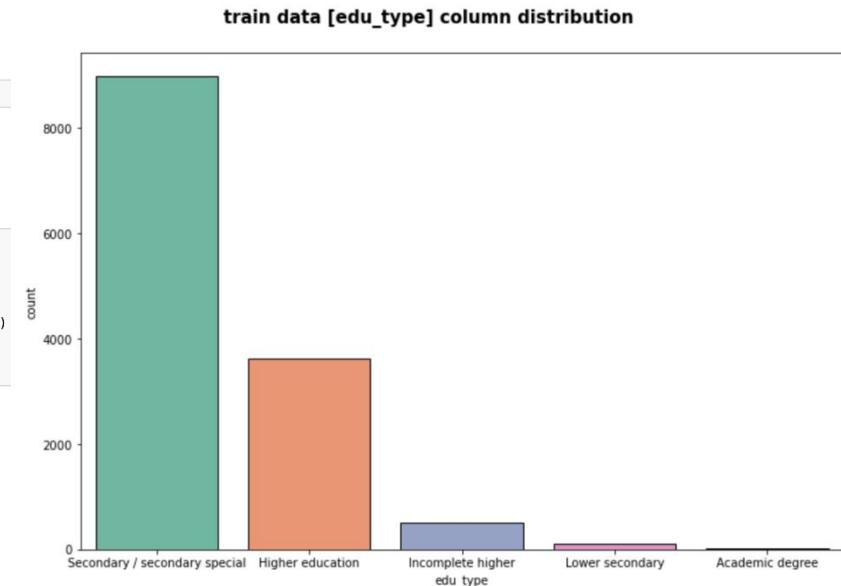
## 01. EDA

### (13) [edu\_type] EDA - 교육 수준

- train과 test 데이터의 countplot 분포가 비슷함
- 'Secondary / secondary special'의 비율이 가장 높음

```
train
In [37]: train['edu_type'].value_counts()
Out[37]: Secondary / secondary special    8972
Higher education                      3626
Incomplete higher                      499
Lower secondary                          118
Academic degree                         13
Name: edu_type, dtype: int64

In [38]: # countplot 시각화
fig, axes = plt.subplots(1,1, figsize=(10, 7), sharey=True)
sns.countplot(x='edu_type', data=train, palette="Set2", edgecolor='black')
plt.suptitle('train data [edu_type] column distribution', fontsize=15, fontweight='bold', x=0.25, y=1.0, ha='left')
plt.tight_layout()
plt.show()
```



▲ train data

# 데이터 탐색

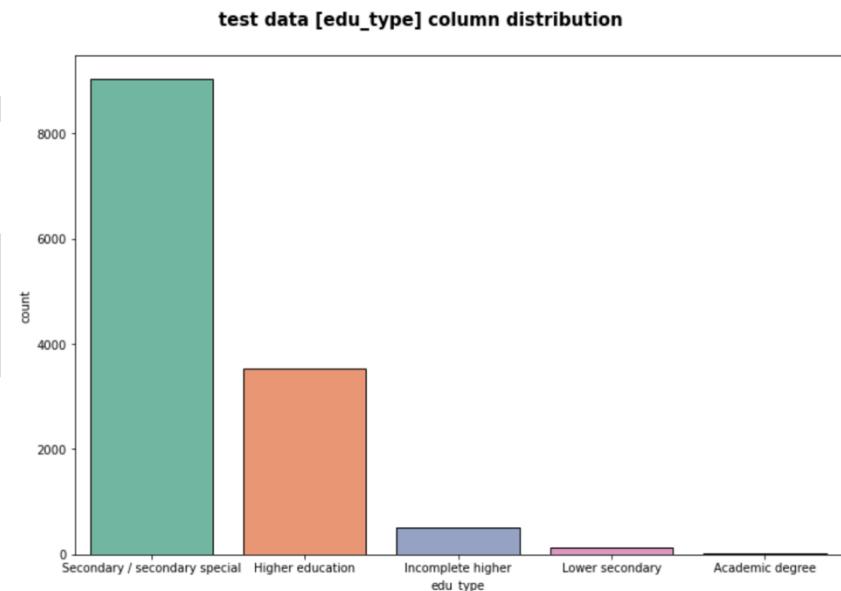
## 01. EDA

### (13) [edu\_type] EDA - 교육 수준

- train과 test 데이터의 countplot 분포가 비슷함
- 'Secondary / secondary special'의 비율이 가장 높음

```
test
In [39]: test['edu_type'].value_counts()
Out[39]:
Secondary / secondary special    9023
Higher education                  3536
Incomplete higher                 521
Lower secondary                   139
Academic degree                  10
Name: edu_type, dtype: int64

In [40]: # countplot 시각화
fig, axes = plt.subplots(1,1, figsize=(10, 7), sharey=True)
sns.countplot(x='edu_type', data=test, palette="Set2", edgecolor='black')
plt.suptitle('test data [edu_type] column distribution', fontsize=15, fontweight='bold', x=0.25, y=1.0, ha='left')
plt.tight_layout()
plt.show()
```



▲ test data

# 데이터 탐색

## 01. EDA

### (14) [family\_type] EDA - 결혼 여부

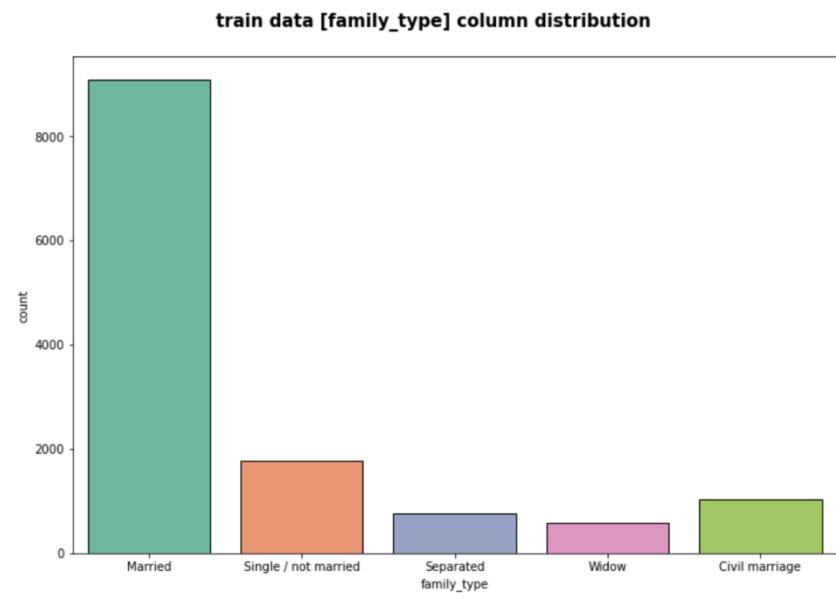
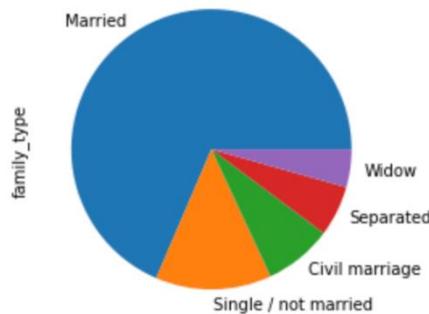
- train과 test 데이터의 countplot 분포가 비슷함
- 'Married'의 비율이 가장 높음

```
train
In [41]: train['family_type'].value_counts()
Out[41]: Married          9068
Single / not married    1770
Civil marriage           1041
Separated                 771
Widow                      578
Name: family_type, dtype: int64

In [42]: # countplot 시각화
fig, axes = plt.subplots(1,1, figsize=(10, 7), sharey=True)
sns.countplot(x='family_type', data=train, palette="Set2", edgecolor='black')
plt.suptitle('train data [family_type] column distribution', fontsize=15, fontweight='bold', x=0.25, y=1.0, ha='left')
plt.tight_layout()
plt.show()
```

```
In [43]: # 데이터 분포도
train['family_type'].value_counts().plot.pie()
```

```
Out[43]: <AxesSubplot:ylabel='family_type'>
```



▲ train data

# 데이터 탐색

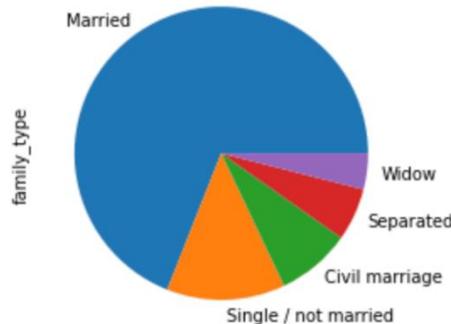
## 01. EDA

### (14) [family\_type] EDA - 결혼 여부

- train과 test 데이터의 countplot 분포가 비슷함
- 'Married'의 비율이 가장 높음

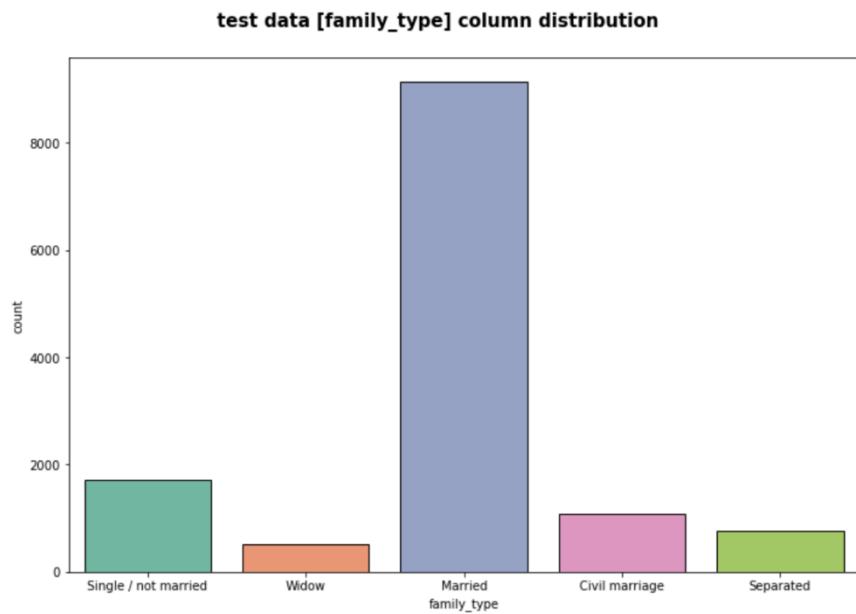
```
In [46]: # 데이터 분포도
test['family_type'].value_counts().plot.pie()
```

Out[46]: <AxesSubplot:ylabel='family\_type'>



```
test
In [44]: test['family_type'].value_counts()
Out[44]: Married      9128
Single / not married    1726
Civil marriage        1082
Separated            768
Widow                525
Name: family_type, dtype: int64

In [45]: # countplot 시각화
fig, axes = plt.subplots(1,1, figsize=(10, 7), sharey=True)
sns.countplot(x='family_type', data=test, palette="Set2", edgecolor='black')
plt.suptitle('test data [family_type] column distribution', fontsize=15, fontweight='bold', x=0.25, y=1.0, ha='left')
plt.tight_layout()
plt.show()
```



▲ test data

# 데이터 탐색

## 01. EDA

### (15) [house\_type] EDA - 생활 방식

- train과 test 데이터의 countplot 분포가 비슷함
- train과 test 모두 'House / apartment'의 비율이 가장 높음



▲ train data

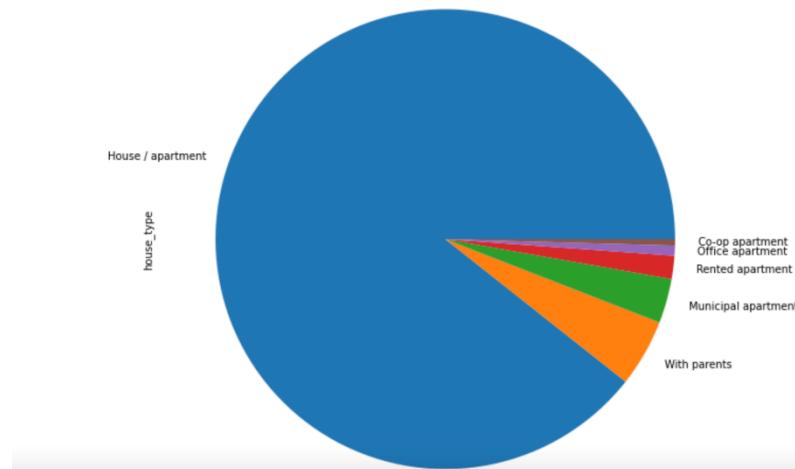
# 데이터 탐색

## 01. EDA

### (15) [house\_type] EDA - 생활 방식

- train과 test 데이터의 countplot 분포가 비슷함
- train과 test 모두 'House / apartment'의 비율이 가장 높음

```
In [49]: # 데이터 분포도  
fig, axes = plt.subplots(1, 1, figsize=(10,10), sharey=True)  
train['house_type'].value_counts().plot.pie()  
  
Out[49]: <AxesSubplot:ylabel='house_type'>
```



▲ train data

# 데이터 탐색

## 01. EDA

### (15) [house\_type] EDA - 생활 방식

- train과 test 데이터의 countplot 분포가 비슷함
- train과 test 모두 'House / apartment'의 비율이 가장 높음



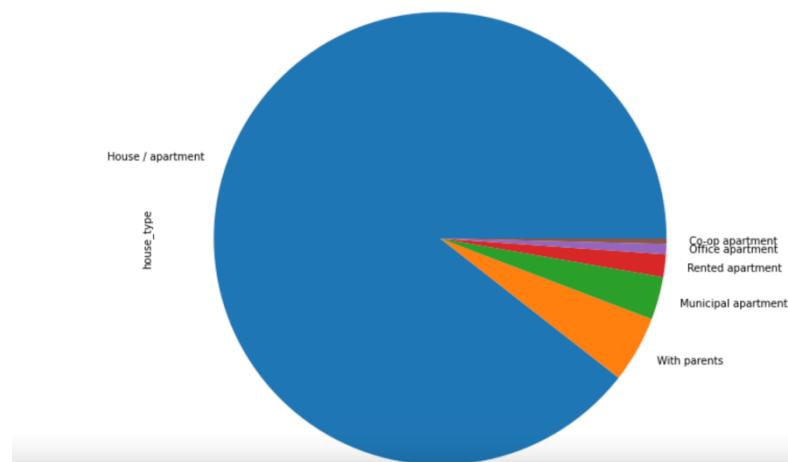
# 데이터 탐색

## 01. EDA

### (15) [house\_type] EDA - 생활 방식

- train과 test 데이터의 countplot 분포가 비슷함
- train과 test 모두 'House / apartment'의 비율이 가장 높음

```
In [52]: # 데이터 분포도  
fig, axes = plt.subplots(1, 1, figsize=(10,10), sharey=True)  
test['house_type'].value_counts().plot.pie()  
  
Out[52]: <AxesSubplot:ylabel='house_type'>
```



▲ test data

# 데이터 탐색

## 01. EDA

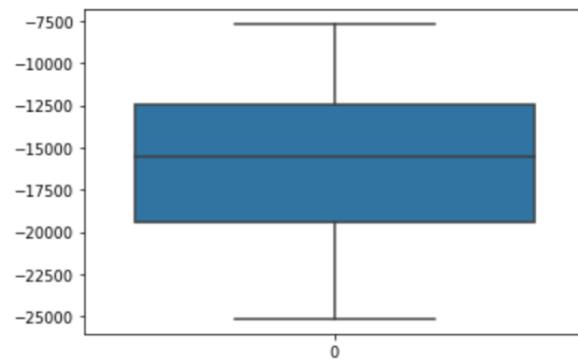
### (16) [DAYS\_BIRTH] EDA - 출생일

- 데이터 수집 당시 (0)부터 역으로 셈. 즉, -1은 데이터 수집일 하루 전에 태어났음을 의미
- train과 test 데이터의 boxplot 분포가 비슷함
- 모두 outlier로 판단되는 데이터 없음

train

```
In [53]: # boxplot 시각화  
sns.boxplot(train['DAYS_BIRTH'])
```

```
Out[53]: <AxesSubplot:>
```

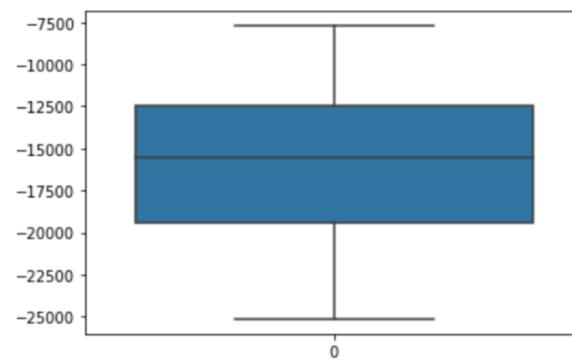


▲ train data

test

```
In [54]: # boxplot 시각화  
sns.boxplot(test['DAYS_BIRTH'])
```

```
Out[54]: <AxesSubplot:>
```



▲ test data

# 데이터 탐색

## 01. EDA

### (17) [DAYS\_EMPLOYED] EDA - 업무 시작일

- 데이터 수집 당시 (0)부터 역으로 셈. 즉, -1은 데이터 수집일 하루 전부터 일을 시작함을 의미 양수 값은 고용되지 않은 상태를 의미함
- train과 test 데이터의 boxplot 분포가 비슷함
- 양수 값 : 크기에 상관없이 고용되지 않은 상태를 의미 → 양수 값에 대한 전처리 과정 필요

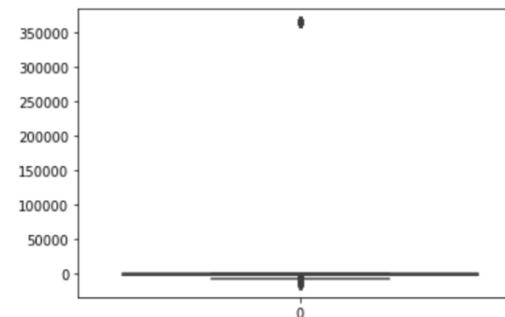
train

In [55]: `train['DAYS_EMPLOYED']`

```
Out[55]: 0      365243
         1      -309
         2      -1816
         3      -150
         4     -2371
         ..
        13223    -5637
        13224   -3482
        13225   -7827
        13226   -2326
        13227   -1621
Name: DAYS_EMPLOYED, Length: 13228, dtype: int64
```

In [57]: `# boxplot 시각화``sns.boxplot(train['DAYS_EMPLOYED'])`

Out[57]: &lt;AxesSubplot:&gt;



▲ train data

# 데이터 탐색

## 01. EDA

### (17) [DAYS\_EMPLOYED] EDA - 업무 시작일

- 데이터 수집 당시 (0)부터 역으로 셈. 즉, -1은 데이터 수집일 하루 전부터 일을 시작함을 의미 양수 값은 고용되지 않은 상태를 의미함
- train과 test 데이터의 boxplot 분포가 비슷함
- 양수 값 : 크기에 상관없이 고용되지 않은 상태를 의미 → 양수 값에 대한 전처리 과정 필요

		In [56]:	train[train['DAYS_EMPLOYED'] > 0]	Out [56]:	
0	0	F	Y	Y	0 202500.0 Pensioner Secondary / secondary special Married House / apartment -19031 365243 1 0 0 0 NaN 2 -53 1
6	6	F	N	N	0 157500.0 Pensioner Secondary / secondary special Married House / apartment -21253 365243 1 0 1 0 NaN 2 -10 1
7	7	M	Y	Y	1 270000.0 Pensioner Secondary / secondary special Married House / apartment -18948 365243 1 0 0 0 NaN 3 -52 1
12	12	F	N	Y	0 112500.0 Pensioner Secondary / secondary special Married House / apartment -22361 365243 1 0 0 0 NaN 2 -30 1
14	14	F	Y	N	0 225000.0 Pensioner Secondary / secondary special Married House / apartment -20460 365243 1 0 0 0 NaN 2 -20 1
...	...	...	...	...	...
13191	13191	F	N	Y	0 121500.0 Pensioner Secondary / secondary special Married House / apartment -23784 365243 1 0 0 0 NaN 2 -13 1
13193	13193	F	N	Y	0 135000.0 Pensioner Higher education Married House / apartment -21226 365243 1 0 0 0 NaN 2 -13 1
13195	13195	F	N	Y	0 112500.0 Pensioner Secondary / secondary special Separated House / apartment -23632 365243 1 0 0 0 NaN 1 -18 1
13207	13207	F	N	Y	0 360000.0 Pensioner Higher education Married House / apartment -22295 365243 1 0 0 0 NaN 2 -17 1
13221	13221	F	N	Y	0 157500.0 Pensioner Secondary / secondary special Widow House / apartment -23596 365243 1 0 0 0 NaN 1 -28 1

2247 rows × 20 columns

▲ train data

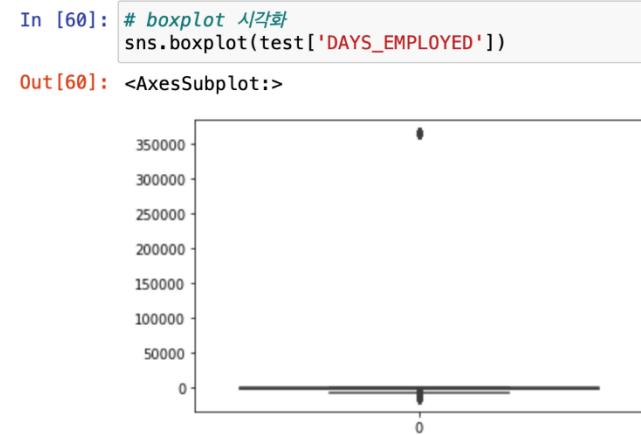
# 데이터 탐색

## 01. EDA

### (17) [DAYS\_EMPLOYED] EDA - 업무 시작일

- 데이터 수집 당시 (0)부터 역으로 셈. 즉, -1은 데이터 수집일 하루 전부터 일을 시작함을 의미 양수 값은 고용되지 않은 상태를 의미함
- train과 test 데이터의 boxplot 분포가 비슷함
- 양수 값 : 크기에 상관없이 고용되지 않은 상태를 의미 → 양수 값에 대한 전처리 과정 필요

```
test
In [58]: test['DAYS_EMPLOYED']
Out[58]: 0      -1101
1      365243
2     -1389
3     -4019
4     -2879
...
13224   -2057
13225   -2049
13226   -5420
13227   -4781
13228   -962
Name: DAYS_EMPLOYED, Length: 13229, dtype: int64
```



▲ test data

# 데이터 탐색

## 01. EDA

### (17) [DAYS\_EMPLOYED] EDA - 업무 시작일

- 데이터 수집 당시 (0)부터 역으로 셈. 즉, -1은 데이터 수집일 하루 전부터 일을 시작함을 의미 양수 값은 고용되지 않은 상태를 의미함
- train과 test 데이터의 boxplot 분포가 비슷함
- 양수 값 : 크기에 상관없이 고용되지 않은 상태를 의미 → 양수 값에 대한 전처리 과정 필요

In [59]:		test[test['DAYS_EMPLOYED'] > 0]																			
Out [59]:																					
		index	gender	car	reality	child_num	income_total	income_type	edu_type	family_type	house_type	days_birth	days_employed	flag_mobil	work_phone	phone	email	occyp_type	family_size	begin_month	credit
1	1	1	F	N	Y	0	157500.0	Pensioner	Secondary / secondary special	Widow	House / apartment	-24340	365243	1	0	1	0	NaN	1	-62	NaN
6	6	6	F	N	Y	0	46000.0	Pensioner	Secondary / secondary special	Married	House / apartment	-21740	365243	1	0	0	0	NaN	2	-6	NaN
8	8	8	F	N	N	0	135000.0	Pensioner	Secondary / secondary special	Married	House / apartment	-23056	365243	1	0	1	0	NaN	2	-11	NaN
19	19	19	F	N	Y	0	90000.0	Pensioner	Secondary / secondary special	Married	House / apartment	-22123	365243	1	0	0	1	NaN	2	-31	NaN
24	24	24	F	N	Y	0	90000.0	Pensioner	Secondary / secondary special	Married	House / apartment	-20718	365243	1	0	0	0	NaN	2	-6	NaN
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
13201	13201	13201	F	N	Y	0	157500.0	Pensioner	Secondary / secondary special	Married	House / apartment	-23129	365243	1	0	0	0	NaN	2	-60	NaN
13205	13205	13205	F	N	Y	0	67500.0	Pensioner	Secondary / secondary special	Married	House / apartment	-20575	365243	1	0	1	0	NaN	2	-28	NaN
13207	13207	13207	F	N	Y	0	225000.0	Pensioner	Higher education	Married	House / apartment	-23868	365243	1	0	0	0	NaN	2	-20	NaN
13211	13211	13211	M	N	Y	0	46948.5	Pensioner	Secondary / secondary special	Single / not married	House / apartment	-21167	365243	1	0	0	0	NaN	1	-5	NaN
13223	13223	13223	F	N	Y	0	81000.0	Pensioner	Secondary / secondary special	Separated	House / apartment	-23137	365243	1	0	0	0	NaN	1	-31	NaN

2191 rows × 20 columns

▲ test data

# 데이터 탐색

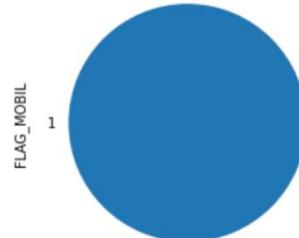
## 01. EDA

### (18) [FLAG\_MOBIL] EDA - 핸드폰 소유 여부

- train과 test 데이터의 countplot 분포가 비슷함
- 모두 1이라는 단일 값만을 가짐 → 분류 모델에 영향이 없을 것이라는 판단 → 변수 축소 전처리 필요

train

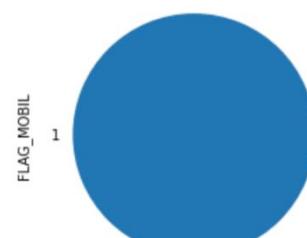
```
In [61]: train['FLAG_MOBIL'].value_counts()  
Out[61]: 1    13228  
Name: FLAG_MOBIL, dtype: int64  
  
In [62]: # 데이터 분포도  
train['FLAG_MOBIL'].value_counts().plot.pie()  
Out[62]: <AxesSubplot:ylabel='FLAG_MOBIL'>
```



▲ train data

test

```
In [63]: test['FLAG_MOBIL'].value_counts()  
Out[63]: 1    13229  
Name: FLAG_MOBIL, dtype: int64  
  
In [64]: # 데이터 분포도  
test['FLAG_MOBIL'].value_counts().plot.pie()  
Out[64]: <AxesSubplot:ylabel='FLAG_MOBIL'>
```



▲ test data

# 데이터 탐색

## 01. EDA

### (19) [work\_phone] EDA - 업무용 전화 소유 여부

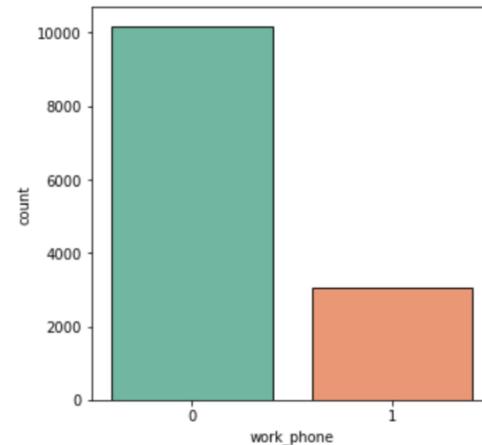
- train과 test 데이터의 countplot 분포가 비슷함
- train과 test 모두 0의 비율이 높음

```
train

In [65]: train['work_phone'].value_counts()
Out[65]: 0    10182
          1     3046
Name: work_phone, dtype: int64

In [66]: # countplot 시각화
    fig, axes = plt.subplots(1, 1, figsize=(5, 5), sharey=True)
    sns.countplot(x='work_phone', data=train, palette="Set2", edgecolor='black')
    plt.suptitle('train data [work_phone] column distribution', fontsize=15, fontweight='bold', x=0.1, y=1.0, ha='left')
    plt.tight_layout()
    plt.show()
```

train data [work\_phone] column distribution



▲ train data

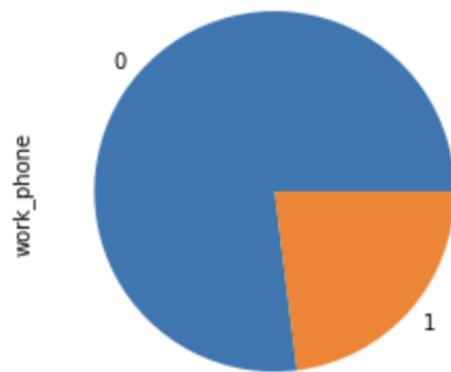
# 데이터 탐색

## 01. EDA

### (19) [work\_phone] EDA - 업무용 전화 소유 여부

- train과 test 데이터의 countplot 분포가 비슷함
- train과 test 모두 0의 비율이 높음

```
In [67]: # 데이터 분포도  
train['work_phone'].value_counts().plot.pie()  
  
Out[67]: <AxesSubplot:ylabel='work_phone'>
```



▲ train data

# 데이터 탐색

## 01. EDA

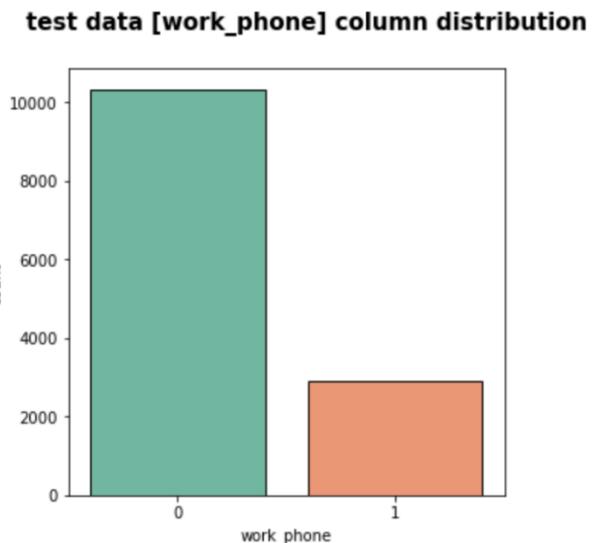
### (19) [work\_phone] EDA - 업무용 전화 소유 여부

- train과 test 데이터의 countplot 분포가 비슷함
- train과 test 모두 0의 비율이 높음

```
test

In [68]: test['work_phone'].value_counts()
Out[68]: 0    10329
          1     2900
Name: work_phone, dtype: int64

In [69]: # countplot 시각화
    fig, axes = plt.subplots(1, 1, figsize=(5, 5), sharey=True)
    sns.countplot(x='work_phone', data=test, palette="Set2", edgecolor='black')
    plt.suptitle('test data [work_phone] column distribution', fontsize=15, fontweight='bold', x=0.1, y=1.0, ha='left')
    plt.tight_layout()
    plt.show()
```



▲ test data

# 데이터 탐색

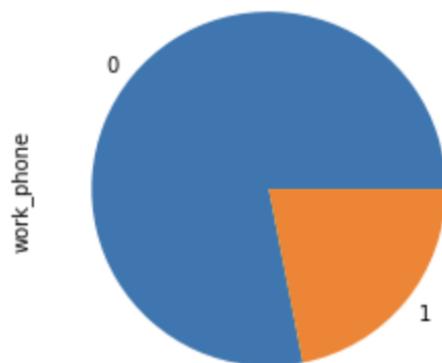
## 01. EDA

### (19) [work\_phone] EDA - 업무용 전화 소유 여부

- train과 test 데이터의 countplot 분포가 비슷함
- train과 test 모두 0의 비율이 높음

```
In [70]: # 데이터 분포도  
test['work_phone'].value_counts().plot.pie()
```

```
Out[70]: <AxesSubplot:ylabel='work_phone'>
```



▲ test data

# 데이터 탐색

## 01. EDA

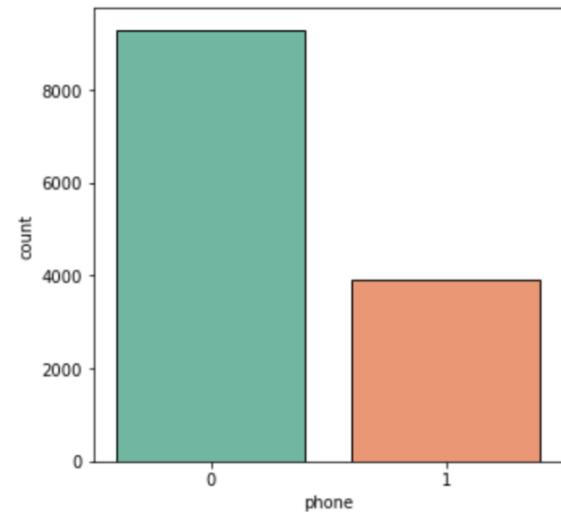
### (20) [phone] EDA - 전화 소유 여부

- train과 test 데이터의 countplot 분포가 비슷함
- train과 test 모두 0의 비율이 높음

```
train  
  
In [71]: train['phone'].value_counts()  
Out[71]: 0    9300  
1    3928  
Name: phone, dtype: int64
```

```
In [72]: # countplot 시각화  
  
fig, axes = plt.subplots(1, 1, figsize=(5, 5), sharey=True)  
  
sns.countplot(x='phone', data=train, palette="Set2", edgecolor='black')  
plt.suptitle('train data [phone] column distribution', fontsize=15, fontweight='bold', x=0.13, y=1.0, ha='left')  
  
plt.tight_layout()  
plt.show()
```

train data [phone] column distribution



▲ train data

# 데이터 탐색

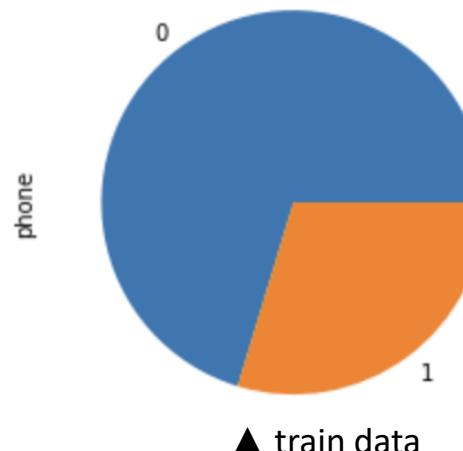
## 01. EDA

### (20) [phone] EDA - 전화 소유 여부

- train과 test 데이터의 countplot 분포가 비슷함
- train과 test 모두 0의 비율이 높음

```
In [73]: # 데이터 분포도  
train['phone'].value_counts().plot.pie()
```

```
Out[73]: <AxesSubplot:ylabel='phone'>
```



# 데이터 탐색

## 01. EDA

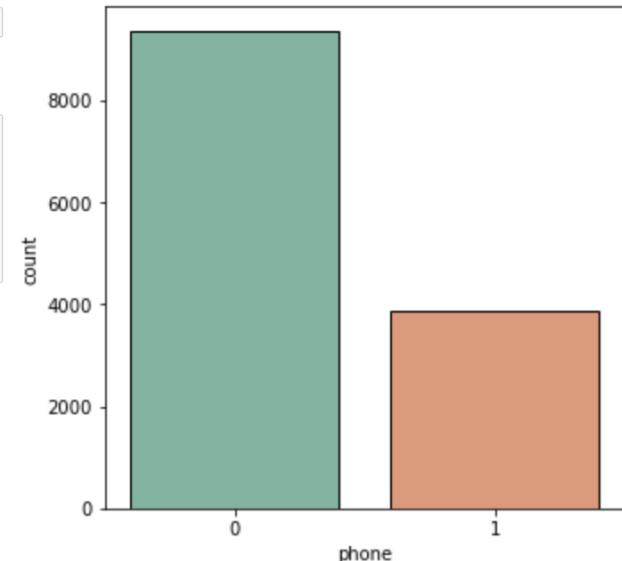
### (20) [phone] EDA - 전화 소유 여부

- train과 test 데이터의 countplot 분포가 비슷함
- train과 test 모두 0의 비율이 높음

```
test
In [74]: test['phone'].value_counts()
Out[74]: 0    9372
          1    3857
          Name: phone, dtype: int64

In [75]: # countplot 시각화
          fig, axes = plt.subplots(1, 1, figsize=(5, 5), sharey=True)
          sns.countplot(x='phone', data=test, palette="Set2", edgecolor="black")
          plt.suptitle('test data [phone] column distribution', fontsize=15, fontweight='bold', x=0.13, y=1.0, ha='left')
          plt.tight_layout()
          plt.show()
```

test data [phone] column distribution



▲ test data

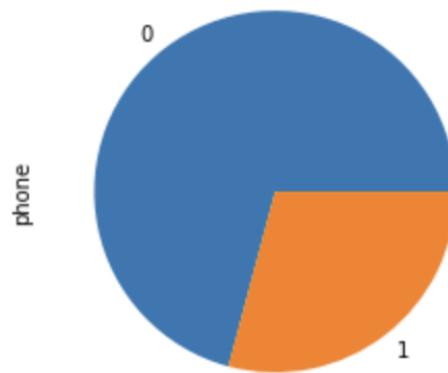
# 데이터 탐색

## 01. EDA

### (20) [phone] EDA - 전화 소유 여부

- train과 test 데이터의 countplot 분포가 비슷함
- train과 test 모두 0의 비율이 높음

```
In [76]: # 데이터 분포도  
test['phone'].value_counts().plot.pie()  
  
Out[76]: <AxesSubplot: ylabel='phone'>
```



▲ test data

# 데이터 탐색

## 01. EDA

### (21) [email] EDA - 이메일 소유 여부

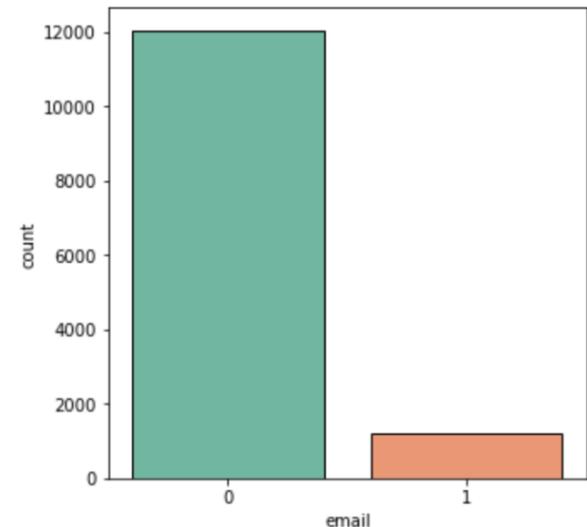
- train과 test 데이터의 countplot 분포가 비슷함
- train과 test 모두 0의 비율이 높음

```
train

In [77]: train['email'].value_counts()
Out[77]: 0    12044
          1    1184
          Name: email, dtype: int64

In [78]: # countplot 시각화
          fig, axes = plt.subplots(1, 1, figsize=(5, 5), sharey=True)
          sns.countplot(x='email', data=train, palette="Set2", edgecolor='black')
          plt.suptitle('train data [email] column distribution', fontsize=15, fontweight='bold', x=0.13, y=1.0, ha='left')
          plt.tight_layout()
          plt.show()
```

train data [email] column distribution



▲ train data

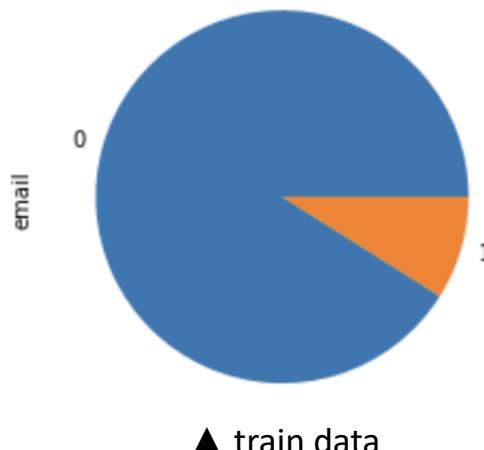
# 데이터 탐색

## 01. EDA

### (21) [email] EDA - 이메일 소유 여부

- train과 test 데이터의 countplot 분포가 비슷함
- train과 test 모두 0의 비율이 높음

```
In [79]: # 데이터 분포도  
train['email'].value_counts().plot.pie()  
  
Out[79]: <AxesSubplot:ylabel='email'>
```



# 데이터 탐색

## 01. EDA

### (21) [email] EDA - 이메일 소유 여부

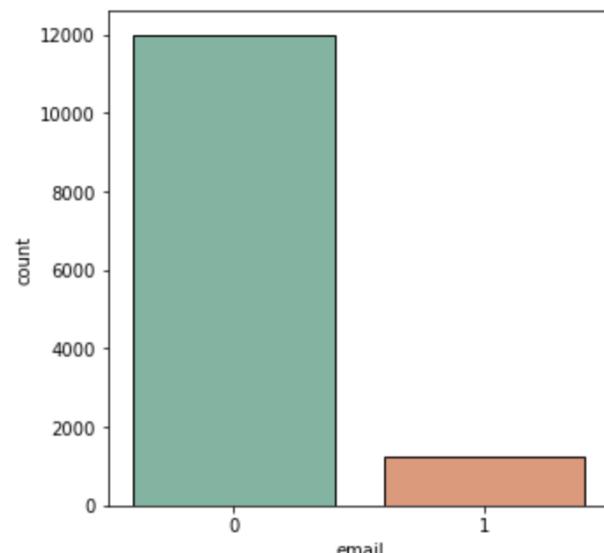
- train과 test 데이터의 countplot 분포가 비슷함
- train과 test 모두 0의 비율이 높음

```
test

In [80]: test['email'].value_counts()
Out[80]: 0    11998
          1    1231
Name: email, dtype: int64

In [81]: # countplot 시각화
          fig, axes = plt.subplots(1, 1, figsize=(5, 5), sharey=True)
          sns.countplot(x='email', data=test, palette="Set2", edgecolor='black')
          plt.suptitle('test data [email] column distribution', fontsize=15, fontweight='bold', x=0.13, y=1.0, ha='left')
          plt.tight_layout()
          plt.show()
```

test data [email] column distribution



▲ test data

# 데이터 탐색

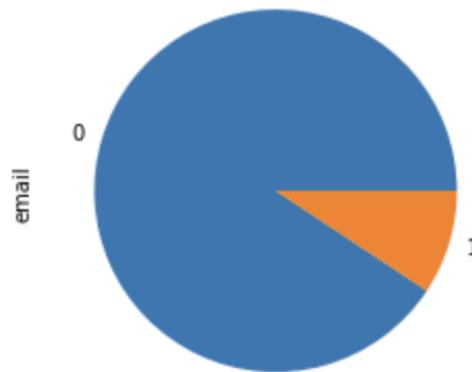
## 01. EDA

### (21) [email] EDA - 이메일 소유 여부

- train과 test 데이터의 countplot 분포가 비슷함
- train과 test 모두 0의 비율이 높음

```
In [82]: # 데이터 분포도  
test['email'].value_counts().plot.pie()
```

```
Out[82]: <AxesSubplot: ylabel='email'>
```



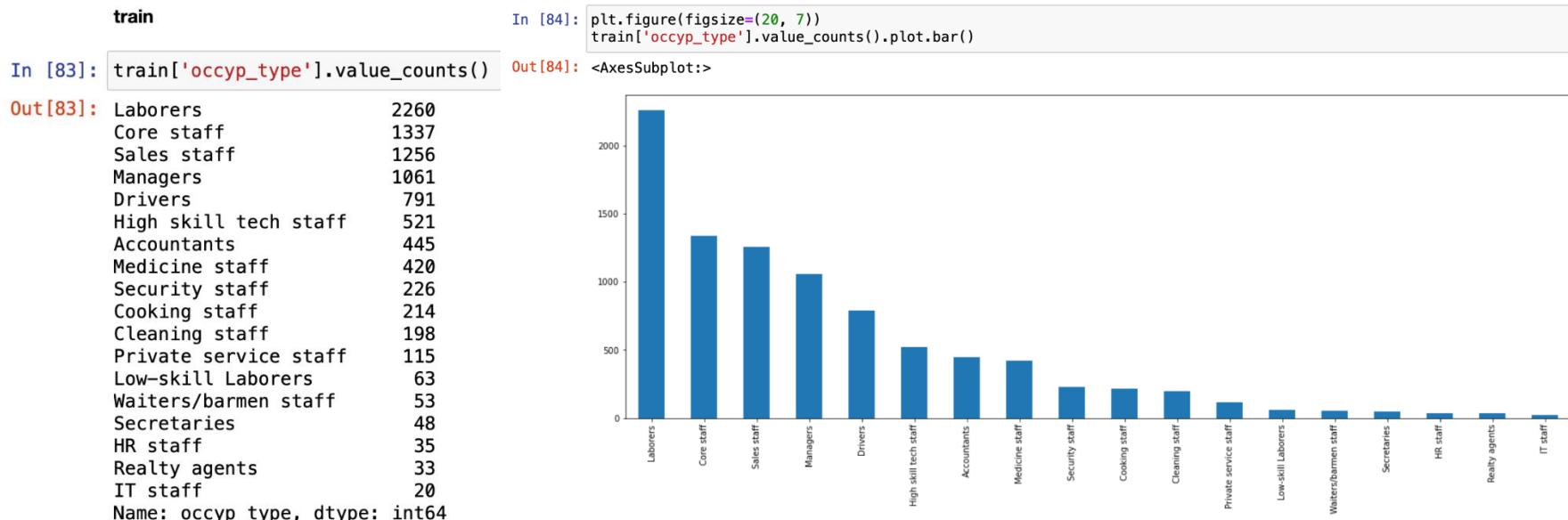
▲ test data

# 데이터 탐색

## 01. EDA

### (22) [occyp\_type] EDA - 직업 유형

- train과 test 데이터의 plot 분포가 비슷함
- train과 test 모두 'Laborers'의 비율이 가장 높음



# 데이터 탐색

## 01. EDA

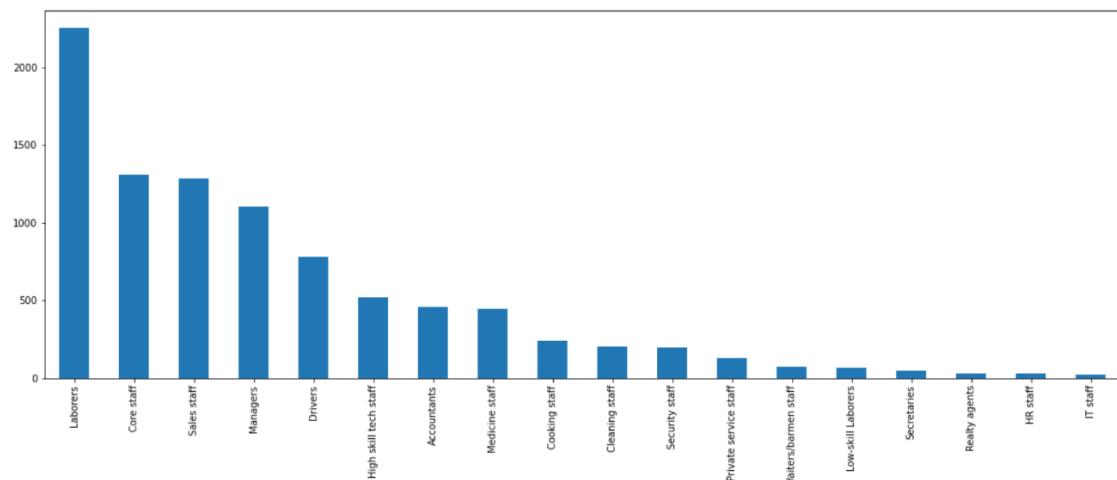
### (22) [occyp\_type] EDA - 직업 유형

- train과 test 데이터의 plot 분포가 비슷함
- train과 test 모두 'Laborers'의 비율이 가장 높음

```
test  
In [85]: test['occyp_type'].value_counts()  
Out[85]:
```

occyp_type	Count
Laborers	2252
Core staff	1309
Sales staff	1283
Managers	1106
Drivers	784
High skill tech staff	519
Accountants	457
Medicine staff	444
Cooking staff	243
Cleaning staff	205
Security staff	198
Private service staff	128
Waiters/barmen staff	71
Low-skill Laborers	64
Secretaries	49
Realty agents	30
HR staff	27
IT staff	21
Name: occyp_type, dtype: int64	

```
In [86]: plt.figure(figsize=(20, 7))  
test['occyp_type'].value_counts().plot.bar()  
Out[86]: <AxesSubplot:>
```



▲ test data

# 데이터 탐색

## 01. EDA

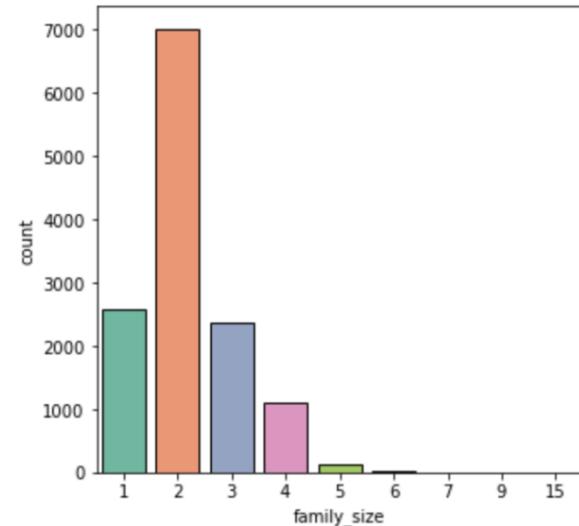
### (23) [family\_size] EDA - 가족 규모

- train과 test 데이터의 countplot 분포가 비슷함
- train과 test 모두 2의 비율이 높음

```
train  
  
In [87]: train['family_size'].value_counts()  
  
Out[87]: 2    7016  
1    2576  
3    2374  
4    1092  
5     138  
6      24  
7      5  
15     2  
9      1  
Name: family_size, dtype: int64
```

```
In [88]: # countplot 시각화  
  
fig, axes = plt.subplots(1, 1, figsize=(5, 5), sharey=True)  
  
sns.countplot(x='family_size', data=train, palette="Set2", edgecolor='black')  
plt.suptitle('train data [family_size] column distribution', fontsize=15, fontweight='bold', x=0.1, y=1.0, ha='left')  
plt.tight_layout()  
plt.show()
```

train data [family\_size] column distribution



▲ train data

# 데이터 탐색

## 01. EDA

### (23) [family\_size] EDA - 가족 규모

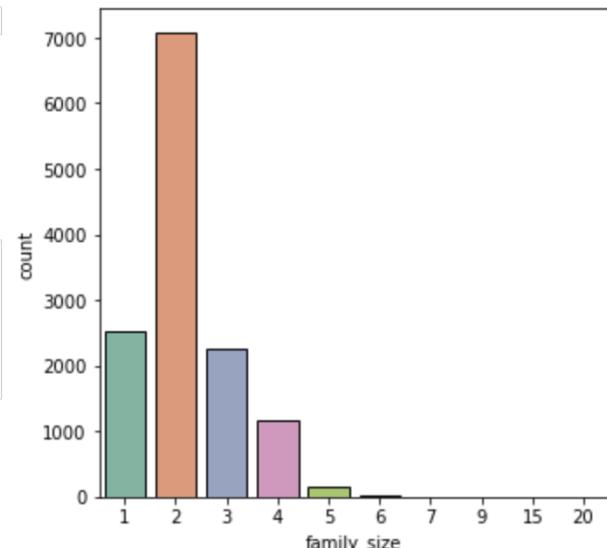
- train과 test 데이터의 countplot 분포가 비슷함
- train과 test 모두 2의 비율이 높음

```
test

In [89]: test['family_size'].value_counts()
Out[89]: 2    7090
1    2533
3    2258
4    1168
5    153
6     20
7      4
9      1
20     1
15     1
Name: family_size, dtype: int64

In [90]: # countplot 시각화
fig, axes = plt.subplots(1, 1, figsize=(5, 5), sharey=True)
sns.countplot(x='family_size', data=test, palette="Set2", edgecolor='black')
plt.suptitle('test data [family_size] column distribution', fontsize=15, fontweight='bold', x=0.1, y=1.0, ha='left')
plt.tight_layout()
plt.show()
```

test data [family\_size] column distribution



▲ test data

# 데이터 탐색

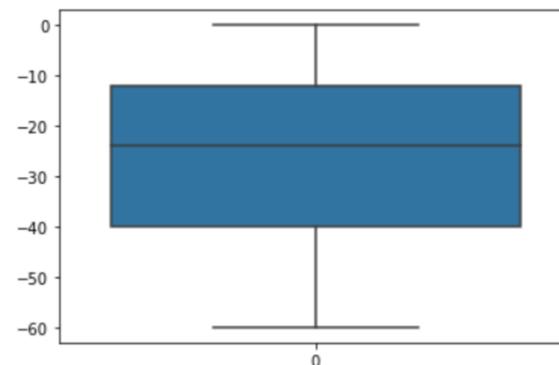
## 01. EDA

### (24) [begin\_month] EDA - 신용카드 발급 월

- 데이터 수집 당시 (0)부터 역으로 셈, 즉, -1은 데이터 수집일 한 달 전에 신용카드를 발급함을 의미
- train과 test 데이터의 boxplot 분포가 비슷함
- 모두 outlier로 판단되는 데이터 없음

```
train  
  
In [91]: train['begin_month']  
  
Out[91]: 0      -53  
1      -26  
2      -9  
3     -12  
4      -3  
..  
13223   -43  
13224   -53  
13225   -34  
13226   -16  
13227    -4  
Name: begin_month, Length: 13228, dtype: int64
```

```
In [92]: # boxplot 시각화  
sns.boxplot(train['begin_month'])  
  
Out[92]: <AxesSubplot:>
```



▲ train data

# 데이터 탐색

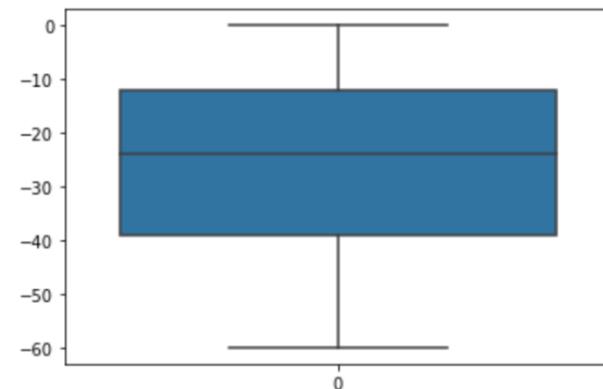
## 01. EDA

### (24) [begin\_month] EDA - 신용카드 발급 월

- 데이터 수집 당시 (0)부터 역으로 셈, 즉, -1은 데이터 수집일 한 달 전에 신용카드를 발급함을 의미
- train과 test 데이터의 boxplot 분포가 비슷함
- 모두 outlier로 판단되는 데이터 없음

```
test
In [93]: test['begin_month']
Out[93]: 0      -10
          1      -52
          2      -15
          3      -24
          4      -54
          ..
          13224   -30
          13225   -24
          13226   -26
          13227   -30
          13228   -53
Name: begin_month, Length: 13229, dtype: int64
```

```
In [94]: # boxplot 시각화
sns.boxplot(test['begin_month'])
Out[94]: <AxesSubplot:>
```



▲ test data

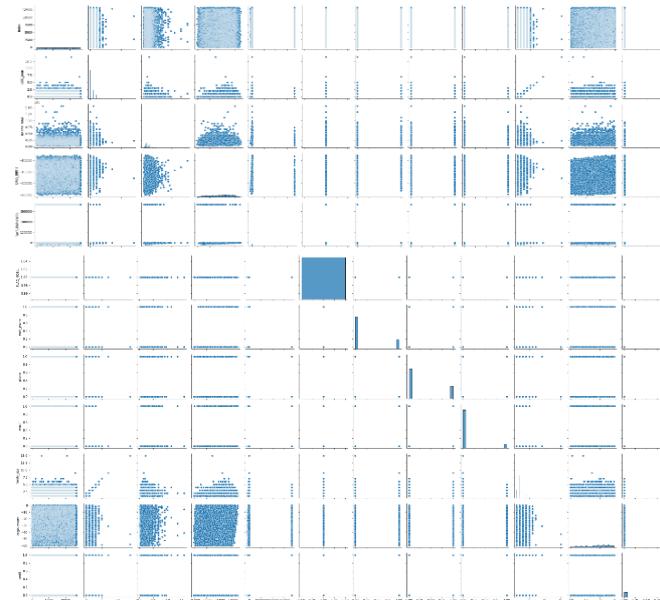
# 데이터 탐색

## 01. EDA

### (25) pairplot 생성 - sns.pairplot()

- 지나치게 선형인 그래프를 통해 데이터 전처리를 통한 변수 제거가 불가피함을 확인

```
In [97]: plt.figure(figsize=(8,8))
sns.pairplot(data=train)
plt.show()
```



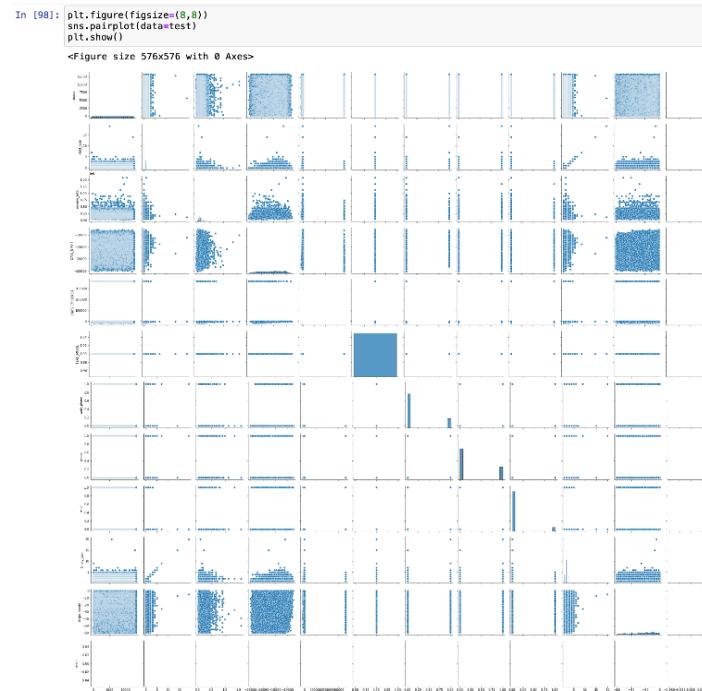
▲ train data

# 데이터 탐색

## 01. EDA

### (25) pairplot 생성 - sns.pairplot()

- 지나치게 선형인 그래프를 통해 데이터 전처리를 통한 변수 제거가 불가피함을 확인



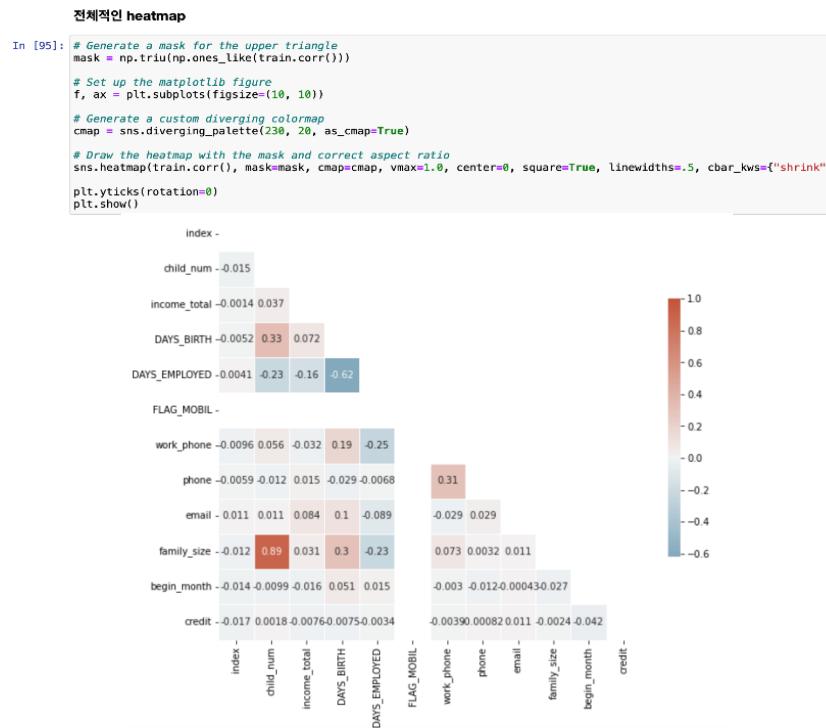
▲ test data

# 데이터 탐색

## 02. 통계적 분석

### (1) heatmap 생성 - sns.heatmap()

- 상관 관계가 높은 경우가 존재하며, 데이터 전처리를 통한 변수 제거가 불가피함을 확인



▲ train data

# 데이터 탐색

## 02. 통계적 분석

### (1) heatmap 생성 - sns.heatmap()

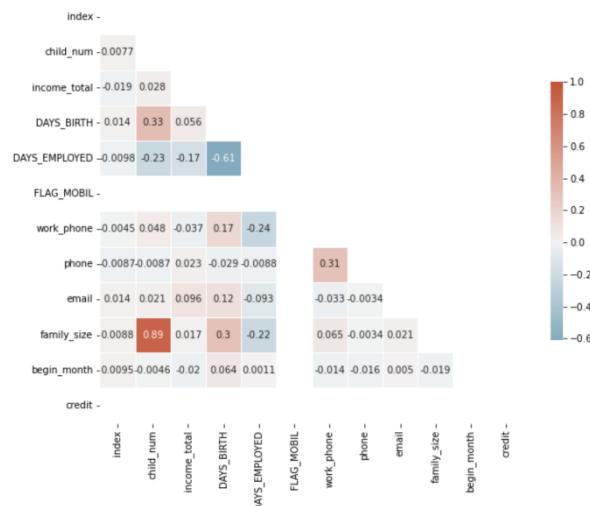
- 상관 관계가 높은 경우가 존재하며, 데이터 전처리를 통한 변수 제거가 불가피함을 확인

```
In [96]: # Generate a mask for the upper triangle
mask = np.triu(np.ones_like(test.corr()))

# Set up the matplotlib figure
f, ax = plt.subplots(figsize=(10, 10))

# Generate a custom diverging colormap
cmap = sns.diverging_palette(230, 20, as_cmap=True)

# Draw the heatmap with the mask and correct aspect ratio
sns.heatmap(test.corr(), mask=mask, cmap=cmap, vmax=1.0, center=0, square=True, linewidths=.5, cbar_kws={"shrink": .9})
plt.yticks(rotation=0)
plt.show()
```



▲ test data

# 전처리 방법

## 01. 변수 축소 및 조정

### (1) 사용하지 않는 변수 축소

- 의미가 없는 변수 및 단일 값들 갖는 변수를 `data.drop()` 함수를 활용하여 변수 축소
- ▶ 변수를 축소함으로써 차원의 저주 해결 및 연산량이 감소하여 모델의 속도가 빨라짐, 또한 시각화에도 용이

### (2) 결측치 확인 및 변수 조정

- `data.isnull()` 함수를 활용하여 결측치 확인, 결측치에 대체 값 삽입
- 변수가 갖고 있는 의미에 따라 변수 값 임의 조정
- ▶ 결측치를 제거 및 변수를 조정함으로써 데이터의 정제 효과로 모델의 성능 개선에 도움이 될 수 있음

### (3) 범주형 변수 더미화

- 범주형 변수를 더미화하여 0 또는 1의 값만 가지도록 조정
- 어떠한 특성이 존재하는가 존재하지 않는가를 표시
- ▶ 범주형 변수를 수치로 변경해 해석함으로써 모델이 해당 변수를 반영 가능

# 전처리 방법

## 02. Class Imbalance 문제 해결

### (1) Class Imbalance의 문제점

- Class Imbalance 문제가 발생한다면 단순히 우세한 클래스를 택하는 모형의 정확도가 높아지는 문제점이 발생
- 정확도(accuracy)가 높아도 재현율(recall-rate)이 작아지게 됨
- oversampling 기법을 활용해 이 문제를 해결 가능

### (2) SMOTETomek 기법

- SMOTETomek : SMOTE(Synthetic Minority Over-sampling Technique) 방법 + 토멕 링크 방법
- SMOTE : 낮은 비율로 존재하는 클래스의 데이터를 k-NN 알고리즘을 활용하여 새롭게 생성하는 방법
- 토멕링크 : 서로 다른 클래스에 속하는 한 쌍의 데이터를 찾은 다음, 그 중에서 다수 클래스에 속하는 데이터를 제거하는 방법

▶ Class Imbalance 문제를 해결하고 일반적 성능이 좋은 모델을 도출 가능

# 전처리 방법

## 03. Scaling

### (1) Scaling의 정의 및 중요성

- Scaling : 데이터의 범위를 임의조정하는 것
- StandardScaling, MinmaxScaling, RobustScaling 등을 시도함
- 성과가 가장 좋았던 **StandardScaling**을 사용
- ▶ Scaling 과정을 통해 독립된 여러 개의 변수들의 중요도가 달라지는 것을 방지함

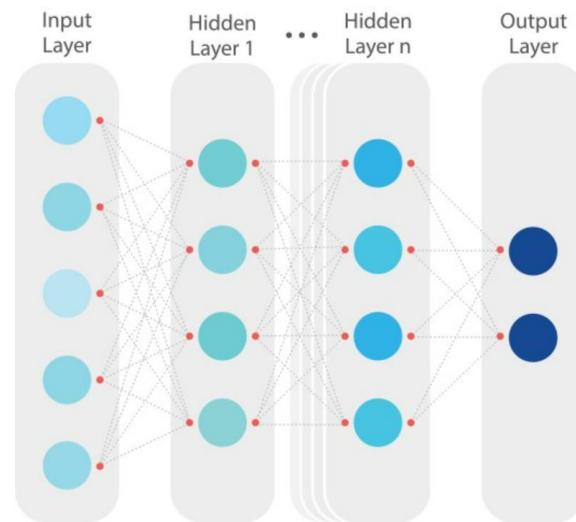
### (2) StandardScaling

- StandardScaling : 데이터의 평균이 0, 분산이 1이 되도록 Scaling하는 방법
- ▶ **StandardScaling**을 통해 데이터를 정규분포화 하여, 각 변수의 중요도가 달라지는 것을 방지

# 모델 소개

## 01. DNN(Deep Neural Network)

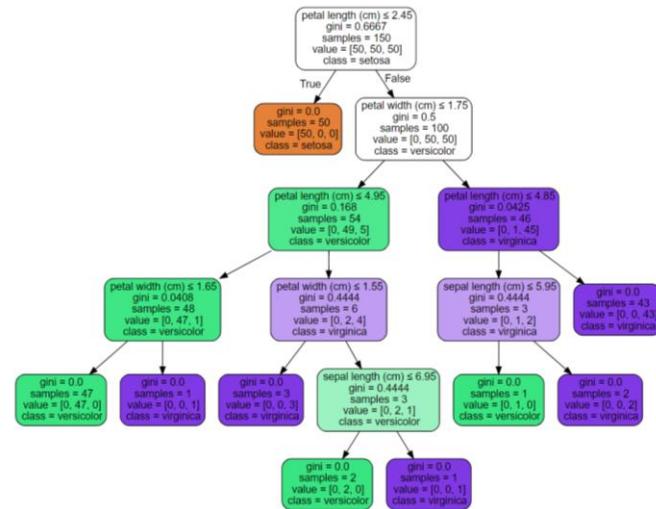
- DNN은 입력층과 출력층 사이에 여러 은닉층이 존재하는 ANN. 다양한 딥러닝 기법 중 하나로 볼 수 있음
- DNN 역시 Neural Network이기 때문에 비선형적 관계의 모델링이 가능하다는 것이 특징
- BPN(Back-Propagation) 알고리즘을 이용해 모델을 학습시킬 수 있음. Sigmoid 함수가 많이 사용됨. 출력과 기대했던 값을 비교해 차이를 줄이는 방향으로 가중치를 조절
- 학습 시 크기(계층 수, 노드 수), 학습률, 초기 가중치 등 많은 수의 파라미터들이 고려되어야 함
- 장점 : ANN보다 더 적은 수의 노드로도 복잡한 데이터를 모델링할 수 있음
- 단점 : 과적합과 높은 시간 복잡도가 발생할 수 있음. 이론적 기반과 설명력이 완전하지 않을 수 있음



# 모델 소개

## 02. Decision Tree

- Decision tree는 의사결정 규칙을 나무구조로 도표화해 분류, 예측을 수행하는 방법. Inductive rule에 따름. CART, CHAID, C4.5, C5.0 등의 알고리즘이 많이 활용됨
- 어떤 변수가 어떤 분류 상태에 큰 영향을 주는지 사람이 쉽게 파악할 수 있게 결과를 줌
- 과정 : Decision tree 형성 → 가지치기 → 타당성 평가 → 해석 및 예측 → 의사결정
- 부모마디의 순수도보다 자식마디의 순수도가 증가하도록 형성
- 장점 : 설명력이 좋음. 직관적인 이해가 가능하고, 속도가 빠른 편
- 단점 : 종속변수가 연속형일 때는 쓸 수 없음. 데이터가 추가되면 Decision Tree 자체가 바뀔 수 있으며 이 방법 자체로 예측력을 가지기는 어려움

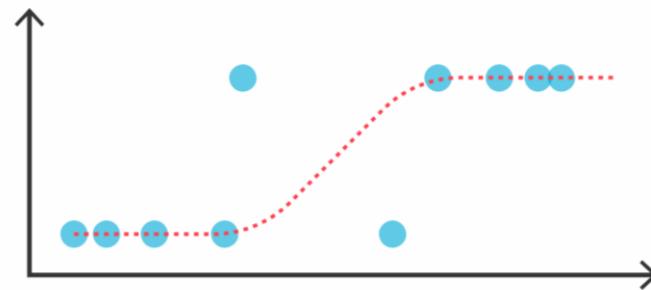


# 모델 소개

## 03. Logistic Regression

- 독립 변수의 선형 결합을 이용해 사건의 발생 가능성을 예측하는 데 사용되는 통계 기법. 선택확률이 로지스틱 함수 형태
- 어느 집단에 속하는지를 판단할 때 사용되는 방법
- 선형 회귀 분석과 비슷한 점도 있지만 Logistic Regression은 종속 변수가 범주형 자료라는 차이점이 있음. 특히 종속변수의 유효한 범주 개수가 2개인 경우에 많이 사용됨.
- 장점 : 독립변수는 정성적, 정량적 변수가 모두 가능, 일반적인 선형 회귀분석보다 특이한 데이터 값이 모수에 미치는 영향력이 작음
- 단점 : 효율적인 분석이 가능하지만, 가중치 해석이 쉽지 않으며 과적합 발생 우려도 존재

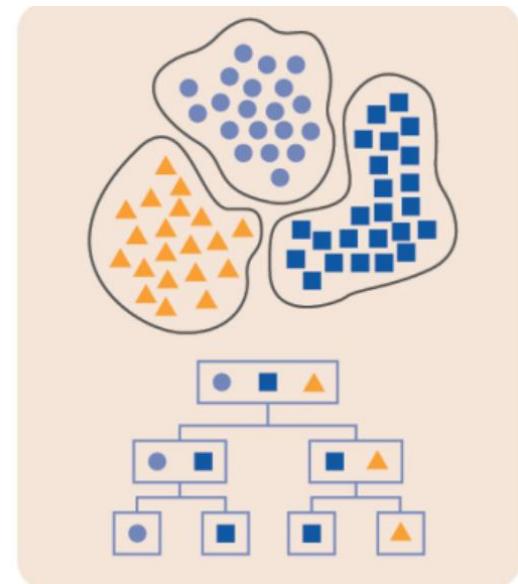
$$g(x) = \frac{e^x}{1 + e^x}$$



# 모델 소개

## 04. Clustering Analysis

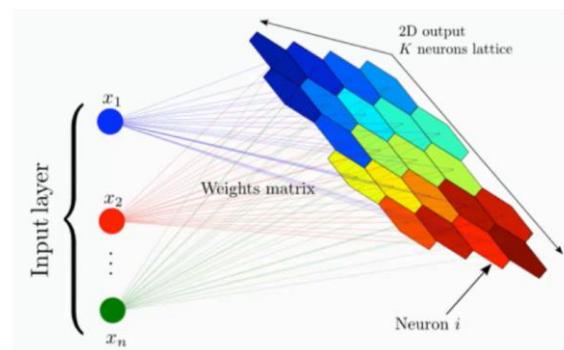
- 주어진 데이터들의 특성을 고려해 Cluster를 정의하고 중심점을 찾는 방법
- 클러스터 : 비슷한 특성을 가진 데이터들의 집단  
(데이터가 가지는 Self-similarity에 의해 서로 다른 클러스터로 나뉘며 각 클러스터는 하나의 데이터로 대표될 수 있음)
- 거리에 근거한 클러스터링 방법인 K-Means Method이 대표적  
(가까운 곳에 위치한 데이터들끼리 묶음. K는 군집 수를 뜻하는데, K값, 초기의 군집 중심 등에 따라 결과 변화)
- 장점 : 미리 클래스를 정의하거나 분류 규칙을 학습시키지 않고 의미 있는 결과를 얻을 수 있음. 또한 다양한 종류의 데이터에 적용할 수 있는 방법
- 단점 : 가중치를 정하는 것이 어려우며, 초기 군집의 수를 정해주는 것이 정확하지 않음. 또한 변수들에 대한 역할 정의가 없기 때문에 결과 해석이 분명치 않음



# 모델 소개

## 05. SOM(Self-Organizing Map)

- 비지도 학습에 의한 클러스터링 방법론. 대뇌 피질의 시각 피질을 모형화한 인공신경망이기도 함
- 승자전취 방식을 따르는 경쟁 학습 알고리즘이며, 자율적으로 학습 하게 함. 입력층과 경쟁층으로 구성
- 초기 Input vectors를 무작위로 선정하고 노드 간 weight를 조정하여 클러스터링 수행
- 매핑 최적화에 시간 투자가 필요하며 임의 구성 요소(초기화, 데이터 표시 순서 등)가 있기 때문에 동일한 토폴로지, 트레이닝 설정을 가진 맵들이 다른 결과를 낼 수도 있음
- 장점 : 높은 차원의 데이터를 낮은 차원으로 변환해 보는 데 유용. 또한 데이터 이해, 유사성 관찰, 해석에 용이, 대규모 데이터를 처리할 수 있으며 feedforward flow를 사용. 입력 데이터의 통계적 분포가 변하면 이에 적응할 수 있음
- 단점 : 필요한 수준의 데이터가 충분한 양이 요구되는데, 현실적으로 어려운 경우도 많음. 그룹화가 맵 내에서 고유한 경우 완벽한 매핑을 얻기가 어려움



# 분류 예측 - 전처리 적용

## 01. train, test 공통 전처리

### (1) 데이터 불러오기

- 전처리 과정을 위해 데이터를 merge시킴

```
In [1]: ┌─▶ import numpy as np
      import pandas as pd
      import matplotlib.pyplot as plt
      import seaborn as sns
      pd.options.display.max_columns = 100

      import warnings
      warnings.filterwarnings('ignore')
```

```
In [2]: ┌─▶ train_x = pd.read_csv('data/train/train.csv')
      train_y = pd.read_csv('data/train/train_label.csv')
      test_x = pd.read_csv('data/test/test.csv')
      test_y = pd.read_csv('data/test/test_label.csv')
```

```
In [3]: ┌─▶ # EDA 및 전처리 과정 용이하게 하기 위해 합침
      train = pd.merge(train_x, train_y)
```

```
In [4]: ┌─▶ # EDA 및 전처리 과정 용이하게 하기 위해 합침
      test = pd.merge(test_x, test_y)
```

# 분류 예측 - 전처리 적용

## 01. train, test 공통 전처리

### (2) train 데이터 파악 - train.head()

- train 데이터셋의 상위 5개 데이터 파악

In [5]: train.head()

	index	gender	car	reality	child_num	income_total	income_type	edu_type	family_type	house_type	DAY_S_BIRTH	DAY_S_EMPLOYED	FLAG_MOBIL
0	0	F	Y	Y	0	202500.0	Pensioner	Secondary / secondary special	Married	House / apartment	-19031	365243	1
1	1	F	N	N	1	157500.0	Working	Higher education	Married	House / apartment	-15773	-309	1
2	2	M	Y	N	0	135000.0	Working	Secondary / secondary special	Married	House / apartment	-13483	-1816	1
3	3	F	Y	N	2	112500.0	Working	Secondary / secondary special	Married	House / apartment	-12270	-150	1
4	4	M	Y	Y	1	225000.0	Working	Secondary / secondary special	Married	House / apartment	-16175	-2371	1

# 분류 예측 - 전처리 적용

## 01. train, test 공통 전처리

### (3) test 데이터 파악 - test.head()

- test 데이터셋의 상위 5개 데이터 파악

```
In [6]: test.head()
```

```
Out[6]:
```

	index	gender	car	reality	child_num	income_total	income_type	edu_type	family_type	house_type	DAY_S_BIRTH	DAY_S_EMPLOYED	FLAG_MOBIL
0	0	M	N	Y	0	211500.0	Working	Secondary / secondary special	Single / not married	House / apartment	-10072	-1101	1
1	1	F	N	Y	0	157500.0	Pensioner	Secondary / secondary special	Widow	House / apartment	-24340	365243	1
2	2	F	N	Y	0	45000.0	Commercial associate	Secondary / secondary special	Married	House / apartment	-15724	-1389	1
3	3	M	N	Y	2	270000.0	Working	Secondary / secondary special	Single / not married	House / apartment	-11505	-4019	1
4	4	F	N	Y	0	202500.0	Working	Secondary / secondary special	Married	House / apartment	-15929	-2879	1

# 분류 예측 - 전처리 적용

## 01. train, test 공통 전처리

### (4) index, DAYS\_EMPLOYED 전처리

- index column은 영향력 없다고 판단해 drop
- DAYS\_EMPLOYED의 양수 값은 고용되지 않은 상태를 의미 → 0으로 값 변경

```
In [7]: train = train.drop(['index'],axis=1)
```

```
In [8]: test = test.drop(['index'],axis=1)
```

```
In [9]: # 기준에 'DAYS_EMPLOYED'가 0의 값을 가지는 경우 -> 존재하지 않을  
train[train['DAYS_EMPLOYED']==0]
```

```
Out[9]: gender car reality child_num income_total income_type edu_type family_type house_type DAYS_BIRTH DAYS_EMPLOYED FLAG_MOBIL work_p
```

```
In [10]: # 'DAYS_EMPLOYED'값이 0보다 큰 경우 모두 0으로 지정  
train.loc[train['DAYS_EMPLOYED']>0, 'DAYS_EMPLOYED'] = 0
```

```
In [11]: # 'DAYS_EMPLOYED'값이 0보다 큰 경우 모두 0으로 지정  
test.loc[test['DAYS_EMPLOYED']>0, 'DAYS_EMPLOYED'] = 0
```

# 분류 예측 - 전처리 적용

## 01. train, test 공통 전처리

### (5) occyp\_type 전처리

- train과 test의 NaN 값 확인 및 다른 column과의 연관성 파악
- isnull() 이용

train

```
In [12]: # train에서 'occyp_type'가 NaN 값인 크기  
train[train['occyp_type'].isnull()].shape  
Out[12]: (4132, 19)
```

```
In [13]: # train에서 'DAYS_EMPLOYED'가 0인 값의 크기  
a = train[train['DAYS_EMPLOYED']==0]  
a.shape  
Out[13]: (2247, 19)
```

```
In [14]: # train에서 'DAYS_EMPLOYED'가 0인 것들 중에 'occyp_type'가 NaN 값인 크기  
a_1 = a[a['occyp_type'].isnull()]  
a_1.shape  
Out[14]: (2247, 19)
```

```
In [15]: # train에서 'DAYS_EMPLOYED'가 0인 'occyp_type'은 NaN 값을 가짐  
a_1.equals(a)  
Out[15]: True
```

# 분류 예측 - 전처리 적용

## 01. train, test 공통 전처리

### (5) occyp\_type 전처리

- [DAYS\_EMPLOYED==0]이면 occyp\_type은 NaN 값임을 확인  
→ 무직 상태이기 때문에 결측값을 가짐 → None으로 대체

test

```
In [16]: # test에서 'occyp_type'가 NaN 값인 크기  
test[test['occyp_type'].isnull()].shape
```

```
Out[16]: (4039, 19)
```

```
In [17]: # test에서 'DAYS_EMPLOYED'가 0인 값의 크기  
b = test[test['DAYS_EMPLOYED']==0]  
b.shape
```

```
Out[17]: (2191, 19)
```

```
In [18]: # test에서 'DAYS_EMPLOYED'가 0인 것들 중에 'occyp_type'가 NaN 값인 크기  
b_1 = b[b['occyp_type'].isnull()]  
b_1.shape
```

```
Out[18]: (2191, 19)
```

```
In [19]: # test에서 'DAYS_EMPLOYED'가 0이면 'occyp_type'은 NaN 값을 가짐  
b_1.equals(b)
```

```
Out[19]: True
```

# 분류 예측 - 전처리 적용

## 01. train, test 공통 전처리

### (5) occyp\_type 전처리

- DAYS\_EMPLOYED 값이 0이 아닌데 occyp\_type이 NaN인 경우 → 'Extra'(기타)로 대체

```
In [20]: train.loc[(train['DAYS_EMPLOYED'] == 0) & (train['occyp_type'].isnull()), 'occyp_type'] = 'None'
```

```
In [21]: train.loc[(train['DAYS_EMPLOYED'] != 0) & (train['occyp_type'].isnull()), 'occyp_type'] = 'Extra'
```

```
In [22]: test.loc[(test['DAYS_EMPLOYED'] == 0) & (test['occyp_type'].isnull()), 'occyp_type'] = 'None'
```

```
In [23]: test.loc[(test['DAYS_EMPLOYED'] != 0) & (test['occyp_type'].isnull()), 'occyp_type'] = 'Extra'
```

```
In [24]: # 결측치 해결  
train.isnull().sum()
```

```
Out[24]: gender      0  
car          0  
reality      0  
child_num    0  
income_total  0  
income_type   0  
edu_type     0  
family_type   0  
house_type    0  
DAYS_BIRTH    0  
DAYS_EMPLOYED 0  
FLAG_MOBIL    0  
work_phone    0  
phone         0  
email         0  
occyp_type    0  
family_size   0  
begin_month   0  
credit        0  
dtype: int64
```

```
In [25]: # 결측치 해결(실제 예측해야 할 credit 결측치)  
test.isnull().sum()
```

```
Out[25]: gender      0  
car          0  
reality      0  
child_num    0  
income_total  0  
income_type   0  
edu_type     0  
family_type   0  
house_type    0  
DAYS_BIRTH    0  
DAYS_EMPLOYED 0  
FLAG_MOBIL    0  
work_phone    0  
phone         0  
email         0  
occyp_type    0  
family_size   0  
begin_month   0  
credit        13229  
dtype: int64
```

# 분류 예측 – 전처리 적용

## 01. train, test 공통 전처리

### (6) FLAG\_MOBIL 전처리와 범주형 변수 더미화

- FLAG\_MOBIL은 모두 1의 값을 가짐 → 분류 모델에 크게 영향이 없을 것이라는 판단 → 변수 축소(drop)
- train, test 모두 범주형 변수 더미화 진행(pandas.get\_dummies() 이용)

```
In [26]: train = train.drop(['FLAG_MOBIL'],axis=1)
```

```
In [27]: test = test.drop(['FLAG_MOBIL'],axis=1)
```

```
In [28]: # train 더미화  
train_dum = pd.get_dummies(train, columns = ['gender','car','reality','income_type','edu_type','family_type','house_type','occyp_type'])
```

```
In [29]: # test 더미화  
test_dum = pd.get_dummies(test, columns = ['gender','car','reality','income_type','edu_type','family_type','house_type','occyp_type'])
```

# 분류 예측 – 전처리 적용

## 02. train 데이터셋 분리 및 추가 전처리

### (1) class imbalance 문제 해결

- ① train 데이터셋의 label 값의 비율 확인 → 1:7의 class imbalance 상태임을 확인, 전처리 필요
- ② train : test = 7 : 3으로 y 값 비율에 맞춰 분리(stratify=y)
- ③ class imbalance 해결을 위해 SMOTETomek을 이용한 복합 oversampling 진행
- ④ StandardScaler을 이용해 데이터셋 정규화

```
In [30]: # 0과 1의 label 값의 비율 확인 -> 1:7 의 class imbalance 상태 -> 전처리
train_dum['credit'].value_counts()

Out[30]:
1    11617
0     1611
Name: credit, dtype: int64

In [31]: # train_dum -> train:test = 7:3으로 y값의 비율 맞춰(stratify) 분리
from sklearn.model_selection import train_test_split
y = train_dum['credit']
X = train_dum.drop(['credit'],axis=1)
train_X, val_X, train_y, val_y = train_test_split(X, y, test_size=0.3, random_state=42, stratify=y)

In [32]: # class imbalance 해결을 위해 SMOTETomek을 이용한 복합 oversampling 진행
# over_X : oversampling된 train_X
# over_y : oversampling된 train_y
from imblearn.combine import *
over_X, over_y = SMOTETomek(random_state=42).fit_resample(train_X, train_y)

In [33]: # StandardScaler를 이용해 데이터셋 정규화
from sklearn.preprocessing import StandardScaler
sd=StandardScaler()
sd.fit(over_X)
over_X_sd = sd.transform(over_X)
val_X_sd = sd.transform(val_X)
```

# 분류 예측 - 전처리 적용

## 03. Clustering을 위한 추가 전처리

### (1) 데이터셋 관리

- ① train, test 데이터셋을 합쳐 clustering을 진행하고자 함
- ② 두 데이터셋 concatenation 후 index 재지정(concat, reset\_index 이용)
- ③ 데이터셋에서 예측해야 하는 credit 제외 및 분리(drop)

```
In [30]: # data : train_dum과 test_dum을 위아래로 합친 df
data = pd.concat([train_dum, test_dum])
data
```

Out[30]:

	child_num	income_total	DAY_S_BIRTH	DAY_S_EMPLOYED	work_phone	phone	email	family_size	begin_month	credit	gender_F	gender_M	car_
0	0	202500.0	-19031	0	0	0	0	2	-53	1.0	1	0	
1	1	157500.0	-15773	-309	0	1	0	3	-26	0.0	1	0	
2	0	135000.0	-13483	-1816	1	1	0	2	-9	1.0	0	1	
3	2	112500.0	-12270	-150	0	1	0	4	-12	1.0	1	0	
4	1	225000.0	-16175	-2371	0	0	0	3	-3	1.0	0	1	
...	...	...	...	...	...	...	...	...	...	...	...	...	...
13224	0	202500.0	-12347	-2057	0	0	0	2	-30	NaN	1	0	
13225	0	148500.0	-9382	-2049	0	1	1	1	-24	NaN	0	1	
13226	0	270000.0	-14896	-5420	0	0	1	2	-26	NaN	0	1	
13227	0	405000.0	-15881	-4781	1	0	0	2	-30	NaN	0	1	
13228	0	292500.0	-10375	-962	0	0	0	2	-53	NaN	0	1	

26457 rows × 57 columns

```
In [31]: # index 재지정
data = data.reset_index(drop=True)
data
```

Out[31]:

	child_num	income_total	DAY_S_BIRTH	DAY_S_EMPLOYED	work_phone	phone	email	family_size	begin_month	credit	gender_F	gender_M	car_
0	0	202500.0	-19031	0	0	0	0	2	-53	1.0	1	0	
1	1	157500.0	-15773	-309	0	1	0	3	-26	0.0	1	0	
2	0	135000.0	-13483	-1816	1	1	0	2	-9	1.0	0	1	
3	2	112500.0	-12270	-150	0	1	0	4	-12	1.0	1	0	
4	1	225000.0	-16175	-2371	0	0	0	3	-3	1.0	0	1	
...	...	...	...	...	...	...	...	...	...	...	...	...	...
26452	0	202500.0	-12347	-2057	0	0	0	2	-30	NaN	1	0	
26453	0	148500.0	-9382	-2049	0	1	1	1	-24	NaN	0	1	
26454	0	270000.0	-14896	-5420	0	0	1	2	-26	NaN	0	1	
26455	0	405000.0	-15881	-4781	1	0	0	2	-30	NaN	0	1	
26456	0	292500.0	-10375	-962	0	0	0	2	-53	NaN	0	1	

26457 rows × 57 columns

```
In [32]: # data_1 : data에서 'credit'을 뺀, 즉 라벨값 제거 df
# data_2 : data의 'credit' column, 즉 라벨값
data_1 = data.drop(['credit'],axis=1)
data_2 = data['credit']
```

# 분류 예측 - 전처리 적용

## 03. Clustering을 위한 추가 전처리

### (2) Scaling

- ① 라벨값이 포함되지 않은 데이터인 data\_1에 대해 MinMaxScaling 진행 → scaled\_df 생성  
 - sklearn.preprocessing의 MinMaxScaler 활용

```
In [33]: # 라벨값이 포함되지 않은 데이터인 data_1에 대해 minmaxscaling 진행 -> scaled_df 생성
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
scaled_df = pd.DataFrame(data=scaler.fit_transform(data_1), columns=data_1.columns)
scaled_df.shape
scaled_df
```

Out [33]:

	child_num	income_total	DAY_S_BIRTH	DAY_S_EMPLOYED	work_phone	phone	email	family_size	begin_month	gender_F	gender_M	car_N	car_
0	0.000000	0.113372	0.350834	1.000000	0.0	0.0	0.0	0.052632	0.116667	1.0	0.0	0.0	1
1	0.052632	0.084302	0.537571	0.980335	0.0	1.0	0.0	0.105263	0.566667	1.0	0.0	1.0	0
2	0.000000	0.069767	0.668826	0.884427	1.0	1.0	0.0	0.052632	0.850000	0.0	1.0	0.0	1
3	0.105263	0.055233	0.738350	0.990454	0.0	1.0	0.0	0.157895	0.800000	1.0	0.0	0.0	1
4	0.052632	0.127907	0.514530	0.849106	0.0	0.0	0.0	0.105263	0.950000	0.0	1.0	0.0	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...
26452	0.000000	0.113372	0.733937	0.869089	0.0	0.0	0.0	0.052632	0.500000	1.0	0.0	1.0	0
26453	0.000000	0.078488	0.903880	0.869598	0.0	1.0	1.0	0.000000	0.600000	0.0	1.0	1.0	0
26454	0.000000	0.156977	0.587837	0.655063	0.0	0.0	1.0	0.052632	0.566667	0.0	1.0	1.0	0
26455	0.000000	0.244186	0.531381	0.695730	1.0	0.0	0.0	0.052632	0.500000	0.0	1.0	1.0	0
26456	0.000000	0.171512	0.846965	0.938777	0.0	0.0	0.0	0.052632	0.116667	0.0	1.0	1.0	0

26457 rows × 56 columns

# 분류 예측 - 전처리 적용

## 04. SOM 모델을 위한 추가 전처리

### (1) SOM 진행을 위한 데이터셋 전처리

- ① train\_dum 과 test\_dum을 합친 data 생성
- ② data의 인덱스 재지정, y값인 'credit'을 drop
- ③ 라벨값이 포함되지 않은 데이터인 data\_1에 대해 StandardScaling 진행 → scaled\_df 생성

```
In [30]: # data : train_dum과 test_dum을 위하여 합친 df
data = pd.concat([train_dum, test_dum])
data
```

```
Out[30]:
```

	child_num	income_total	DAY_S_BIRTH	DAY_S_EMPLOYED	work_phone	phone	email	family_size	begin_month	credit	gender_F	gender_M	car_N_c
0	0	202500.0	-19031	0	0	0	0	2	-53	1.0	1	0	0
1	1	157500.0	-15773	-309	0	1	0	3	-26	0.0	1	0	1
2	0	135000.0	-13483	-1816	1	1	0	2	-9	1.0	0	1	0
3	2	112500.0	-12270	-150	0	1	0	4	-12	1.0	1	0	0
4	1	225000.0	-16175	-2371	0	0	0	3	-3	1.0	0	1	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
13224	0	202500.0	-12347	-2057	0	0	0	2	-30	NaN	1	0	1
13225	0	148500.0	-9382	-2049	0	1	1	1	-24	NaN	0	1	1
13226	0	270000.0	-14896	-5420	0	0	1	2	-26	NaN	0	1	1
13227	0	405000.0	-15881	-4781	1	0	0	2	-30	NaN	0	1	1
13228	0	292500.0	-10375	-962	0	0	0	2	-53	NaN	0	1	1

26457 rows × 57 columns

```
In [31]: # index 재지정
data = data.reset_index(drop=True)
data
```

```
Out[31]:
```

	child_num	income_total	DAY_S_BIRTH	DAY_S_EMPLOYED	work_phone	phone	email	family_size	begin_month	credit	gender_F	gender_M	car_N_c
0	0	202500.0	-19031	0	0	0	0	2	-53	1.0	1	0	0
1	1	157500.0	-15773	-309	0	1	0	3	-26	0.0	1	0	1
2	0	135000.0	-13483	-1816	1	1	0	2	-9	1.0	0	1	0
3	2	112500.0	-12270	-150	0	1	0	4	-12	1.0	1	0	0
4	1	225000.0	-16175	-2371	0	0	0	3	-3	1.0	0	1	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
26452	0	202500.0	-12347	-2057	0	0	0	2	-30	NaN	1	0	1
26453	0	148500.0	-9382	-2049	0	1	1	1	-24	NaN	0	1	1
26454	0	270000.0	-14896	-5420	0	0	1	2	-26	NaN	0	1	1
26455	0	405000.0	-15881	-4781	1	0	0	2	-30	NaN	0	1	1
26456	0	292500.0	-10375	-962	0	0	0	2	-53	NaN	0	1	1

26457 rows × 57 columns

```
In [32]: # data에서 'credit'을 뺀, 즉 라벨값 제거 df
data_1 = data.drop(['credit'], axis=1)
data_2 = data['credit']
```

```
In [33]: # 라벨값이 포함되지 않은 데이터인 data_1에 대해 StandardScaling 진행 -> scaled_df 생성
from sklearn.preprocessing import StandardScaler
```

```
scaler = StandardScaler()
scaled_df = pd.DataFrame(data=scaler.fit_transform(data_1), columns=data_1.columns)
scaled_df.shape
scaled_df
```

```
Out[33]:
```

	child_num	income_total	DAY_S_BIRTH	DAY_S_EMPLOYED	work_phone	phone	email	family_size	begin_month	gender_F	gender_M	car_N_c
0	-0.573599	0.149136	-0.731391	0.976162	-0.538417	-0.645705	-0.316937	-0.214735	-1.623064	0.703562	-0.703562	-1.278011
1	0.784529	-0.292575	0.044045	0.797238	-0.538417	1.546693	-0.316937	0.876135	0.007446	0.703562	-0.703562	0.782461
2	-0.573599	-0.513431	0.589087	0.161398	1.857295	1.546693	-0.316937	-0.214735	1.034063	-1.421339	1.421339	-1.278011
3	2.102658	-0.734287	0.877793	0.864324	-0.538417	1.546693	-0.316937	1.967005	0.852895	0.703562	-0.703562	-1.278011
4	0.774529	0.369992	-0.051635	-0.072769	-0.538417	-0.645705	-0.316937	0.876135	1.396399	-1.421339	1.421339	-1.278011
...	...	...	...	...	...	...	...	...	...	...	...	...
26452	-0.573599	0.149136	0.859466	0.059715	-0.538417	-0.645705	-0.316937	-0.214735	-0.234111	0.703562	-0.703562	0.782461
26453	-0.573599	-0.380918	1.565165	0.063090	-0.538417	1.546693	3.155199	-1.305605	0.128224	-1.421339	1.421339	0.782461
26454	-0.573599	0.811704	0.252779	-1.359215	-0.538417	-0.645705	3.155199	-0.214735	0.007446	-1.421339	1.421339	0.782461
26455	-0.573599	2.136838	0.018340	-1.086806	1.857295	-0.645705	-0.316937	-0.214735	-0.234111	-1.421339	1.421339	0.782461
26456	-0.573599	1.032559	1.328821	0.521721	-0.538417	-0.645705	-0.316937	-0.214735	-1.623064	-1.421339	1.421339	0.782461

26457 rows × 56 columns

# 분류 예측 - 모델 학습

## 01. DNN, DT, LR 모델 학습

### (1) Deep Neuron Network(DNN) - MLPClassifier 사용

- ① MLPClassifier 사용하여 DNN 진행
- ② 그리드 서치(GridSearchCV 사용)를 통한 하이퍼파라미터 튜닝 진행
- ③ 최적 하이퍼파라미터 → activation function으로 tanh 함수 사용
- ④ 검증 데이터의 accuracy 확인 : 약 **0.8360**의 정확도

```
In [34]: from sklearn.neural_network import MLPClassifier
mlp = MLPClassifier(random_state=42)
```

```
In [35]: # 하이퍼파라미터 탐색
mlp.get_params().keys()
```

```
Out[35]: dict_keys(['activation', 'alpha', 'batch_size', 'beta_1', 'beta_2', 'early_stopping', 'epsilon', 'hidden_layer_sizes', 'learning_rate', 'learning_rate_init', 'max_fun', 'max_iter', 'momentum', 'n_iter_no_change', 'nesterovs_momentum', 'power_t', 'random_state', 'shuffle', 'solver', 'tol', 'validation_fraction', 'verbose', 'warm_start'])
```

```
In [36]: # 그리드 서치 -> 하이퍼파라미터 튜닝 진행
from sklearn.model_selection import GridSearchCV
```

```
mlp = MLPClassifier(random_state=42)
para1 = {'activation': ['tanh', 'relu']}
mlp = GridSearchCV(estimator = mlp, param_grid = para1, cv=5, scoring='accuracy')
mlp.fit(over_X_sd, over_y)
```

```
Out[36]: GridSearchCV(cv=5, estimator=MLPClassifier(random_state=42),
param_grid={'activation': ['tanh', 'relu']}, scoring='accuracy')
```

```
In [37]: # 최적 하이퍼파라미터 확인
mlp.best_params_
```

```
Out[37]: {'activation': 'tanh'}
```

```
In [38]: # 검증 데이터 accuracy 확인
from sklearn.metrics import accuracy_score
```

```
mlp = MLPClassifier(activation='tanh', random_state=42).fit(over_X_sd, over_y)
pred = mlp.predict(val_X_sd)
accuracy_score(val_y, pred)
```

```
Out[38]: 0.8359788359788359
```

# 분류 예측 - 모델 학습

## 01. DNN, DT, LR 모델 학습

### (1) Deep Neuron Network(DNN) - 직접 구현

#### ① 모델 구조 생성

- Sequential 지정 후 model.add()를 통해 은닉층 추가
- 출력 뉴런과 연결된 Dense 층을 마지막에 추가

#### ② Compile

- 모델 학습 전 모델 학습 환경 설정
- optimizer로 rmsprop을 사용
- 손실함수 loss로 binary\_crossentropy 사용
- metrics로 acc 사용

#### ③ Early Stopping 생성, Fitting

- epoch와 batch\_size를 지정 후 모델을 학습

```
In [39]: # 모델 구현
from tensorflow import keras
from tensorflow.keras import models
from tensorflow.keras import layers

model = models.Sequential()
model.add(layers.Dense(4, activation='relu', input_shape=(56,)))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

In [40]: # compiling
model.compile(optimizer = 'rmsprop', loss = 'binary_crossentropy', metrics = 'acc')

In [41]: # earlystopping 생성
early_stopping = keras.callbacks.EarlyStopping(patience=10, min_delta=0.001, restore_best_weights=True)

history = model.fit(over_X_sd, over_y,
                     validation_data=(val_X_sd, val_y),
                     batch_size=32, epochs=100, callbacks=[early_stopping])

Epoch 1/100
472/472 [=====] - 3s 5ms/step - loss: 0.6148 - acc: 0.7477 - val_loss: 0.4733 - val_acc: 0
.8738
Epoch 2/100
472/472 [=====] - 2s 4ms/step - loss: 0.5082 - acc: 0.8540 - val_loss: 0.4143 - val_acc: 0
.8763
Epoch 3/100
472/472 [=====] - 2s 4ms/step - loss: 0.4497 - acc: 0.8662 - val_loss: 0.4180 - val_acc: 0
.8748
Epoch 4/100
472/472 [=====] - 2s 4ms/step - loss: 0.4094 - acc: 0.8724 - val_loss: 0.4073 - val_acc: 0
.8758
Epoch 5/100
472/472 [=====] - 2s 4ms/step - loss: 0.3814 - acc: 0.8764 - val_loss: 0.4115 - val_acc: 0
.8748
Epoch 6/100
472/472 [=====] - 2s 4ms/step - loss: 0.3622 - acc: 0.8789 - val_loss: 0.4175 - val_acc: 0
.8733
Epoch 7/100
472/472 [=====] - 2s 4ms/step - loss: 0.3520 - acc: 0.8809 - val_loss: 0.4200 - val_acc: 0
.8729
Epoch 8/100
472/472 [=====] - 2s 4ms/step - loss: 0.3418 - acc: 0.8827 - val_loss: 0.4220 - val_acc: 0
.8719
Epoch 9/100
472/472 [=====] - 2s 4ms/step - loss: 0.3316 - acc: 0.8845 - val_loss: 0.4238 - val_acc: 0
.8709
Epoch 10/100
472/472 [=====] - 2s 4ms/step - loss: 0.3214 - acc: 0.8863 - val_loss: 0.4256 - val_acc: 0
.8699
Epoch 11/100
472/472 [=====] - 2s 4ms/step - loss: 0.3112 - acc: 0.8881 - val_loss: 0.4274 - val_acc: 0
.8689
Epoch 12/100
472/472 [=====] - 2s 4ms/step - loss: 0.3010 - acc: 0.8899 - val_loss: 0.4292 - val_acc: 0
.8679
Epoch 13/100
472/472 [=====] - 2s 4ms/step - loss: 0.2908 - acc: 0.8917 - val_loss: 0.4310 - val_acc: 0
.8669
Epoch 14/100
472/472 [=====] - 2s 4ms/step - loss: 0.2806 - acc: 0.8935 - val_loss: 0.4328 - val_acc: 0
.8659
Epoch 15/100
472/472 [=====] - 2s 4ms/step - loss: 0.2704 - acc: 0.8953 - val_loss: 0.4346 - val_acc: 0
.8649
Epoch 16/100
472/472 [=====] - 2s 4ms/step - loss: 0.2602 - acc: 0.8971 - val_loss: 0.4364 - val_acc: 0
.8639
Epoch 17/100
472/472 [=====] - 2s 4ms/step - loss: 0.2500 - acc: 0.8989 - val_loss: 0.4382 - val_acc: 0
.8629
Epoch 18/100
472/472 [=====] - 2s 4ms/step - loss: 0.2400 - acc: 0.9000 - val_loss: 0.4400 - val_acc: 0
.8619
Epoch 19/100
472/472 [=====] - 2s 4ms/step - loss: 0.2300 - acc: 0.9010 - val_loss: 0.4418 - val_acc: 0
.8609
Epoch 20/100
472/472 [=====] - 2s 4ms/step - loss: 0.2200 - acc: 0.9020 - val_loss: 0.4436 - val_acc: 0
.8599
Epoch 21/100
472/472 [=====] - 2s 4ms/step - loss: 0.2100 - acc: 0.9030 - val_loss: 0.4454 - val_acc: 0
.8589
Epoch 22/100
472/472 [=====] - 2s 4ms/step - loss: 0.2000 - acc: 0.9040 - val_loss: 0.4472 - val_acc: 0
.8579
Epoch 23/100
472/472 [=====] - 2s 4ms/step - loss: 0.1900 - acc: 0.9050 - val_loss: 0.4490 - val_acc: 0
.8569
Epoch 24/100
472/472 [=====] - 2s 4ms/step - loss: 0.1800 - acc: 0.9060 - val_loss: 0.4508 - val_acc: 0
.8559
Epoch 25/100
472/472 [=====] - 2s 4ms/step - loss: 0.1700 - acc: 0.9070 - val_loss: 0.4526 - val_acc: 0
.8549
Epoch 26/100
472/472 [=====] - 2s 4ms/step - loss: 0.1600 - acc: 0.9080 - val_loss: 0.4544 - val_acc: 0
.8539
Epoch 27/100
472/472 [=====] - 2s 4ms/step - loss: 0.1500 - acc: 0.9090 - val_loss: 0.4562 - val_acc: 0
.8529
Epoch 28/100
472/472 [=====] - 2s 4ms/step - loss: 0.1400 - acc: 0.9100 - val_loss: 0.4580 - val_acc: 0
.8519
Epoch 29/100
472/472 [=====] - 2s 4ms/step - loss: 0.1300 - acc: 0.9110 - val_loss: 0.4598 - val_acc: 0
.8509
Epoch 30/100
472/472 [=====] - 2s 4ms/step - loss: 0.1200 - acc: 0.9120 - val_loss: 0.4616 - val_acc: 0
.8499
Epoch 31/100
472/472 [=====] - 2s 4ms/step - loss: 0.1100 - acc: 0.9130 - val_loss: 0.4634 - val_acc: 0
.8489
Epoch 32/100
472/472 [=====] - 2s 4ms/step - loss: 0.1000 - acc: 0.9140 - val_loss: 0.4652 - val_acc: 0
.8479
Epoch 33/100
472/472 [=====] - 2s 4ms/step - loss: 0.0900 - acc: 0.9150 - val_loss: 0.4670 - val_acc: 0
.8469
Epoch 34/100
472/472 [=====] - 2s 4ms/step - loss: 0.0800 - acc: 0.9160 - val_loss: 0.4688 - val_acc: 0
.8459
Epoch 35/100
472/472 [=====] - 2s 4ms/step - loss: 0.0700 - acc: 0.9170 - val_loss: 0.4706 - val_acc: 0
.8449
Epoch 36/100
472/472 [=====] - 2s 4ms/step - loss: 0.0600 - acc: 0.9180 - val_loss: 0.4724 - val_acc: 0
.8439
Epoch 37/100
472/472 [=====] - 2s 4ms/step - loss: 0.0500 - acc: 0.9190 - val_loss: 0.4742 - val_acc: 0
.8429
Epoch 38/100
472/472 [=====] - 2s 4ms/step - loss: 0.0400 - acc: 0.9200 - val_loss: 0.4760 - val_acc: 0
.8419
Epoch 39/100
472/472 [=====] - 2s 4ms/step - loss: 0.0300 - acc: 0.9210 - val_loss: 0.4778 - val_acc: 0
.8409
Epoch 40/100
472/472 [=====] - 2s 4ms/step - loss: 0.0200 - acc: 0.9220 - val_loss: 0.4796 - val_acc: 0
.8399
Epoch 41/100
472/472 [=====] - 2s 4ms/step - loss: 0.0100 - acc: 0.9230 - val_loss: 0.4814 - val_acc: 0
.8389
Epoch 42/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9240 - val_loss: 0.4832 - val_acc: 0
.8379
Epoch 43/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9250 - val_loss: 0.4850 - val_acc: 0
.8369
Epoch 44/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9260 - val_loss: 0.4868 - val_acc: 0
.8359
Epoch 45/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9270 - val_loss: 0.4886 - val_acc: 0
.8349
Epoch 46/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9280 - val_loss: 0.4904 - val_acc: 0
.8339
Epoch 47/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9290 - val_loss: 0.4922 - val_acc: 0
.8329
Epoch 48/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9300 - val_loss: 0.4940 - val_acc: 0
.8319
Epoch 49/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9310 - val_loss: 0.4958 - val_acc: 0
.8309
Epoch 50/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9320 - val_loss: 0.4976 - val_acc: 0
.8299
Epoch 51/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9330 - val_loss: 0.4994 - val_acc: 0
.8289
Epoch 52/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9340 - val_loss: 0.5012 - val_acc: 0
.8279
Epoch 53/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9350 - val_loss: 0.5030 - val_acc: 0
.8269
Epoch 54/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9360 - val_loss: 0.5048 - val_acc: 0
.8259
Epoch 55/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9370 - val_loss: 0.5066 - val_acc: 0
.8249
Epoch 56/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9380 - val_loss: 0.5084 - val_acc: 0
.8239
Epoch 57/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9390 - val_loss: 0.5102 - val_acc: 0
.8229
Epoch 58/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9400 - val_loss: 0.5120 - val_acc: 0
.8219
Epoch 59/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9410 - val_loss: 0.5138 - val_acc: 0
.8209
Epoch 60/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9420 - val_loss: 0.5156 - val_acc: 0
.8199
Epoch 61/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9430 - val_loss: 0.5174 - val_acc: 0
.8189
Epoch 62/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9440 - val_loss: 0.5192 - val_acc: 0
.8179
Epoch 63/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9450 - val_loss: 0.5210 - val_acc: 0
.8169
Epoch 64/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9460 - val_loss: 0.5228 - val_acc: 0
.8159
Epoch 65/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9470 - val_loss: 0.5246 - val_acc: 0
.8149
Epoch 66/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9480 - val_loss: 0.5264 - val_acc: 0
.8139
Epoch 67/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9490 - val_loss: 0.5282 - val_acc: 0
.8129
Epoch 68/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9500 - val_loss: 0.5300 - val_acc: 0
.8119
Epoch 69/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9510 - val_loss: 0.5318 - val_acc: 0
.8109
Epoch 70/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9520 - val_loss: 0.5336 - val_acc: 0
.8099
Epoch 71/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9530 - val_loss: 0.5354 - val_acc: 0
.8089
Epoch 72/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9540 - val_loss: 0.5372 - val_acc: 0
.8079
Epoch 73/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9550 - val_loss: 0.5390 - val_acc: 0
.8069
Epoch 74/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9560 - val_loss: 0.5408 - val_acc: 0
.8059
Epoch 75/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9570 - val_loss: 0.5426 - val_acc: 0
.8049
Epoch 76/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9580 - val_loss: 0.5444 - val_acc: 0
.8039
Epoch 77/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9590 - val_loss: 0.5462 - val_acc: 0
.8029
Epoch 78/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9600 - val_loss: 0.5480 - val_acc: 0
.8019
Epoch 79/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9610 - val_loss: 0.5498 - val_acc: 0
.8009
Epoch 80/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9620 - val_loss: 0.5516 - val_acc: 0
.7999
Epoch 81/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9630 - val_loss: 0.5534 - val_acc: 0
.7989
Epoch 82/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9640 - val_loss: 0.5552 - val_acc: 0
.7979
Epoch 83/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9650 - val_loss: 0.5570 - val_acc: 0
.7969
Epoch 84/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9660 - val_loss: 0.5588 - val_acc: 0
.7959
Epoch 85/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9670 - val_loss: 0.5606 - val_acc: 0
.7949
Epoch 86/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9680 - val_loss: 0.5624 - val_acc: 0
.7939
Epoch 87/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9690 - val_loss: 0.5642 - val_acc: 0
.7929
Epoch 88/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9700 - val_loss: 0.5660 - val_acc: 0
.7919
Epoch 89/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9710 - val_loss: 0.5678 - val_acc: 0
.7909
Epoch 90/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9720 - val_loss: 0.5696 - val_acc: 0
.7899
Epoch 91/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9730 - val_loss: 0.5714 - val_acc: 0
.7889
Epoch 92/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9740 - val_loss: 0.5732 - val_acc: 0
.7879
Epoch 93/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9750 - val_loss: 0.5750 - val_acc: 0
.7869
Epoch 94/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9760 - val_loss: 0.5768 - val_acc: 0
.7859
Epoch 95/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9770 - val_loss: 0.5786 - val_acc: 0
.7849
Epoch 96/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9780 - val_loss: 0.5804 - val_acc: 0
.7839
Epoch 97/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9790 - val_loss: 0.5822 - val_acc: 0
.7829
Epoch 98/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9800 - val_loss: 0.5840 - val_acc: 0
.7819
Epoch 99/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9810 - val_loss: 0.5858 - val_acc: 0
.7809
Epoch 100/100
472/472 [=====] - 2s 4ms/step - loss: 0.0000 - acc: 0.9820 - val_loss: 0.5876 - val_acc: 0
.7799
In [42]: history_dict = history.history
history_dict.keys()

Out[42]: dict_keys(['loss', 'acc', 'val_loss', 'val_acc'])
```

# 분류 예측 - 모델 학습

## 01. DNN, DT, LR 모델 학습

### (1) Deep Neuron Network(DNN) - 직접 구현

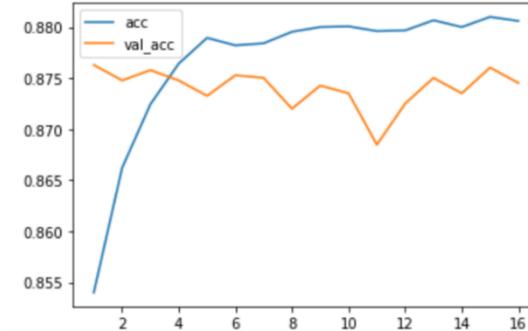
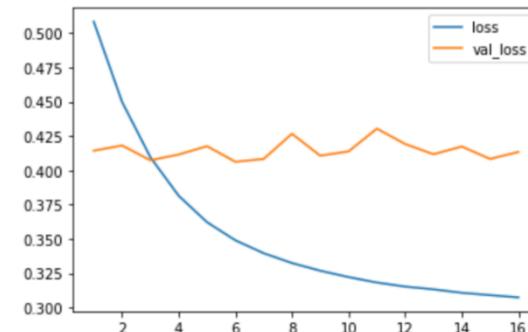
#### ④ loss, accuracy 시각화

- Best Validation Loss : 0.40618
- Best Validation Accuracy : **0.87629**

```
In [43]: # loss, accuracy 시각화
history_df = pd.DataFrame(history.history)
history_df.loc[1:, ['loss', 'val_loss']].plot()
history_df.loc[1:, ['acc', 'val_acc']].plot()

print(("Best Validation Loss: {:.5f}" + \
"\nBest Validation Accuracy: {:.5f}")\
.format(history_df['val_loss'].min(), history_df['val_acc'].max()))
```

Best Validation Loss: 0.40618  
Best Validation Accuracy: 0.87629



# 분류 예측 - 모델 학습

## 01. DNN, DT, LR 모델 학습

### (2) Decision Tree(DT)

- ① DecisionTreeClassifier 사용하여 DT 진행
- ② 그리드 서치(GridSearchCV 사용)를 통한 하이퍼파라미터 튜닝 진행
- ③ 최적 하이퍼파라미터 → criterion = 'entropy', max\_depth = 20
- ④ 검증 데이터의 accuracy 확인 : 약 **0.8133**의 정확도

```
In [44]: # decision tree 클래스 지정
from sklearn.tree import DecisionTreeClassifier
dt = DecisionTreeClassifier(random_state=42)

In [45]: # 하이퍼파라미터 탐색
dt.get_params().keys()

Out[45]: dict_keys(['ccp_alpha', 'class_weight', 'criterion', 'max_depth', 'max_features', 'max_leaf_nodes', 'min_impurity_decrease', 'min_samples_leaf', 'min_samples_split', 'min_weight_fraction_leaf', 'random_state', 'splitter'])

In [46]: # 그리드 서치 -> 하이퍼파라미터 튜닝 진행
dt = DecisionTreeClassifier(random_state=42)
para2 = {'criterion':['gini', 'entropy'], 'max_depth':[5, 10, 15, 20]}

dt = GridSearchCV(estimator = dt, param_grid = para2, cv=5, scoring='accuracy')
dt.fit(over_X_sd, over_y)

Out[46]: GridSearchCV(cv=5, estimator=DecisionTreeClassifier(random_state=42),
param_grid={'criterion': ['gini', 'entropy'],
'max_depth': [5, 10, 15, 20]},
scoring='accuracy')

In [47]: # 최적 하이퍼파라미터 확인
dt.best_params_

Out[47]: {'criterion': 'entropy', 'max_depth': 20}

In [48]: # 검증 데이터 accuracy 확인
dt = DecisionTreeClassifier(criterion='entropy', max_depth=20, random_state=42)
dt.fit(over_X_sd, over_y)
pred = dt.predict(val_X_sd)
print(accuracy_score(val_y, pred))

0.8133030990173847
```

# 분류 예측 - 모델 학습

## 01. DNN, DT, LR 모델 학습

### (3) Logistic Regression(LR)

- ① LogisticRegression 사용하여 LR 진행
- ② 그리드 서치(GridSearchCV 사용)를 통한 하이퍼파라미터 튜닝 진행
- ③ 최적 하이퍼파라미터 → C = 1, penalty = 'l2'
- ④ 검증 데이터의 accuracy 확인 : 약 **0.8778**의 정확도

```
In [49]: # logistic regression 클래스 지정
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression(random_state=42)

In [50]: # 하이퍼파라미터 탐색
lr.get_params().keys()

Out[50]: dict_keys(['C', 'class_weight', 'dual', 'fit_intercept', 'intercept_scaling', 'l1_ratio', 'max_iter', 'multi_class',
       'n_jobs', 'penalty', 'random_state', 'solver', 'tol', 'verbose', 'warm_start'])

In [51]: # 그리드 서치 -> 하이퍼파라미터 튜닝 진행
lr = LogisticRegression(random_state=42)
para3 = {'C': [0.001, 0.01, 0.1, 1, 10, 100], 'penalty': ['l1', 'l2']}
lr = GridSearchCV(estimator=lr, param_grid=para3, cv=5, scoring='accuracy')
lr.fit(over_X_sd, over_y)

Out[51]: GridSearchCV(cv=5, estimator=LogisticRegression(random_state=42),
param_grid={'C': [0.001, 0.01, 0.1, 1, 10, 100],
'penalty': ['l1', 'l2']},
scoring='accuracy')

In [52]: # 최적 하이퍼파라미터 확인
lr.best_params_

Out[52]: {'C': 1, 'penalty': 'l2'}

In [53]: # 검증 데이터 accuracy 확인
lr = LogisticRegression(C=1, penalty='l2', random_state=42)
lr.fit(over_X_sd, over_y)
pred = lr.predict(val_X_sd)
print(accuracy_score(val_y, pred))

0.8778029730410682
```

# 분류 예측 - 모델 학습

## 01. DNN, DT, LR 모델 학습

### (4) 모델별 성능 비교

- DNN(MLPClassifier) : 0.8359788359788359
- DNN(직접 구현) : 0.87629
- Decision Tree : 0.8133030990173847
- Logistic Regression : 0.8778029730410682

❖ 군집화를 진행하지 않은 일반적인 DNN, Decision Tree, Logistic Regression 중에서는 LR이 약 **0.8778**으로 성능이 가장 우수

# 분류 예측 - 모델 학습

## 02. Clustering Analysis + DNN, DT, LR 모델 학습

### (1) Clustering Analysis(K-means)

- ① sklearn.cluster의 KMeans, sklearn.metrics의 silhouette\_samples, silhouette\_score 이용
- ② 함수를 작성하여 군집 개수에 따른 실루엣 계수 확인 및 실루엣 계수 시각화
- ③ K-means 클러스터링을 진행. 실루엣 스코어, 개별 데이터의 실루엣 값 계산

```
def visualize_silhouette(cluster_lists, X_features):
    from sklearn.datasets import make_blobs
    from sklearn.cluster import KMeans
    from sklearn.metrics import silhouette_samples, silhouette_score

    import matplotlib.pyplot as plt
    import matplotlib.cm as cm
    import math

    # 입력값으로 클러스터링 갯수들을 리스트로 받아서, 각 갯수별로 클러스터링을 적용하고 실루엣 개수를 구함
    n_cols = len(cluster_lists)

    # plt.subplots()으로 리스트에 기재된 클러스터링 갯수들의 sub_figures를 가지는 axs 생성
    fig, axs = plt.subplots(figsize=(4*n_cols, 4), nrows=1, ncols=n_cols)

    # 리스트에 기재된 클러스터링 갯수들을 차례로 iteration 수행하면서 실루엣 개수 시각화
    for ind, n_cluster in enumerate(cluster_lists):

        # KMeans 클러스터링 수행하고, 실루엣 스코어와 개별 데이터의 실루엣 값 계산.
        clusterer = KMeans(n_clusters = n_cluster, max_iter=500, random_state=42)
        cluster_labels = clusterer.fit_predict(X_features)

        sil_avg = silhouette_score(X_features, cluster_labels)
        sil_values = silhouette_samples(X_features, cluster_labels)

        y_lower = 10
        axs[ind].set_title('Number of Cluster : ' + str(n_cluster)+ '\n' +
                           'Silhouette Score : ' + str(round(sil_avg,3)) )
        axs[ind].set_xlabel("The silhouette coefficient values")
        axs[ind].set_ylabel("Cluster label")
        axs[ind].set_xlim([-0.1, 1])
        axs[ind].set_ylim([0, len(X_features) + (n_cluster + 1) * 10])
        axs[ind].set_yticks([])
        # Clear the xaxis labels / ticks
        axs[ind].set_xticks([0, 0.2, 0.4, 0.6, 0.8, 1])

        # 클러스터링 갯수별로 fill_betweenx( )형태의 막대 그래프 표현.
        for i in range(n_cluster):
            ith_cluster_sil_values = sil_values[cluster_labels==i]
            ith_cluster_sil_values.sort()

            size_cluster_i = ith_cluster_sil_values.shape[0]
            y_upper = y_lower + size_cluster_i

            color = cm.nipy_spectral(float(i) / n_cluster)
            axs[ind].fill_betweenx(np.arange(y_lower, y_upper), 0, ith_cluster_sil_values, #
                                  facecolor=color, edgecolor=color, alpha=0.7)
            axs[ind].text(-0.05, y_lower + 0.5 * size_cluster_i, str(i))
            y_lower = y_upper + 10

        axs[ind].axvline(x=sil_avg, color="red", linestyle="--")
```

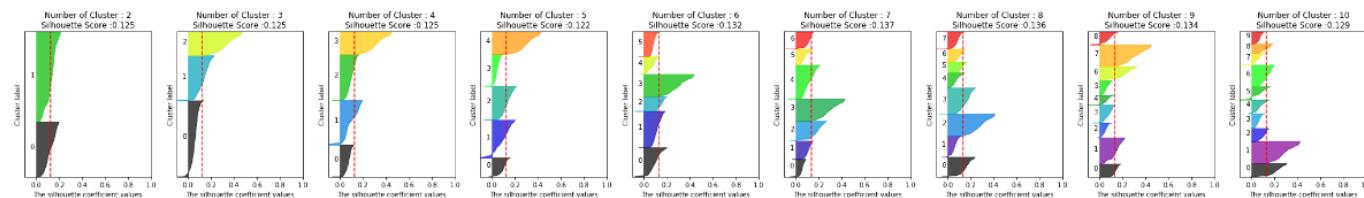
# 분류 예측 - 모델 학습

## 02. Clustering Analysis + DNN, DT, LR 모델 학습

### (1) Clustering Analysis(K-means)

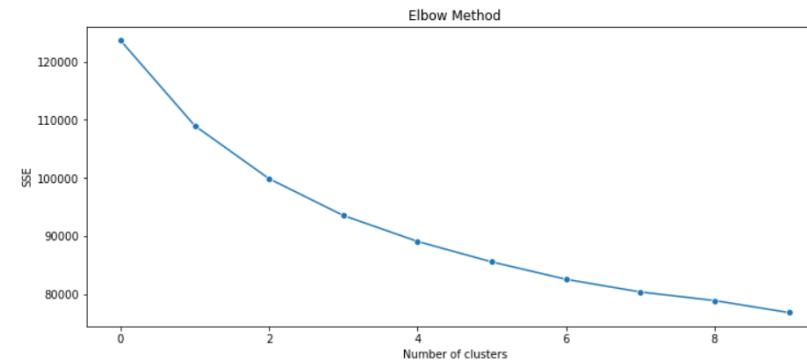
- ④ 함수 적용 결과 cluster 개수는 2가 최적인 것으로 판단
- ⑤ 최적 군집 수 결정을 위해 elbow method도 함께 활용

In [35]: # 실루엣 계수 시각화 → cluster 개수 2가 최적으로 판단  
visualize\_silhouette([2,3,4,5,6,7,8,9,10], scaled\_df)



In [36]: # elbow method 사용  
from sklearn.cluster import KMeans  
import seaborn as sns  
  
SSE = []  
for i in range(1, 11):  
 kmeans = KMeans(n\_clusters = i, random\_state = 42)  
 kmeans.fit(scaled\_df)  
 SSE.append(kmeans.inertia\_)

plt.figure(figsize=(12,5))  
sns.lineplot(SSE, marker='o')  
plt.title('Elbow Method')  
plt.xlabel('Number of clusters')  
plt.ylabel('SSE')  
plt.show()



# 분류 예측 - 모델 학습

## 02. Clustering Analysis + DNN, DT, LR 모델 학습

### (1) Clustering Analysis(K-means)

⑥ sklearn.cluster의 KMeans 활용

⑦ 앞서 결정한 최적 군집 수 2에 따라 분석 진행하였으며, random\_state = 42로 값 통일 적용

```
In [37]: # KMeans -> n_clusters=2로 진행
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=2, random_state=42)
clusters = kmeans.fit(scaled_df)

# 클러스터링 변수인 clusters 값을 원본 데이터의 'scaled_df'에 넣기
scaled_df['cluster'] = clusters.labels_
scaled_df.head()
```

```
Out[37]:
child_num income_total DAYS_BIRTH DAYS_EMPLOYED work_phone phone email family_size begin_month gender_F gender_M car_N car_Y
0 0.000000 0.113372 0.350834 1.000000 0.0 0.0 0.0 0.052632 0.116667 1.0 0.0 0.0 1.0
1 0.052632 0.084302 0.537571 0.980335 0.0 1.0 0.0 0.105263 0.566667 1.0 0.0 0.0 1.0
2 0.000000 0.069767 0.668826 0.884427 1.0 1.0 0.0 0.052632 0.850000 0.0 1.0 0.0 1.0
3 0.105263 0.055233 0.738350 0.990454 0.0 1.0 0.0 0.157895 0.800000 1.0 0.0 0.0 1.0
4 0.052632 0.127907 0.514530 0.849106 0.0 0.0 0.0 0.105263 0.950000 0.0 1.0 0.0 1.0
```

# 분류 예측 - 모델 학습

## 02. Clustering Analysis + DNN, DT, LR 모델 학습

### (1) Clustering Analysis(K-means)

- ⑧ 각 클러스터를 기준으로 데이터 수 카운트하며 확인
- ⑨ 클러스터별 특징 파악 : 클러스터, column별 평균값 비교

```
In [38]: # cluster를 기준으로 데이터 개수 세기  
scaled_df.groupby('cluster').count()
```

```
Out[38]:  
child_num  income_total  DAYS_BIRTH  DAYS_EMPLOYED  work_phone  phone  email  family_size  begin_month  gender_F  gender_M  car_N  ca  
cluster  
0          10047       10047       10047       10047       10047       10047       10047       10047       10047       10047       10047       10047       10047       10047       10047       10047  
1          16410       16410       16410       16410       16410       16410       16410       16410       16410       16410       16410       16410       16410       16410       16410       16410
```

```
In [39]: # 그룹별 특징을 알아보기(그룹별 평균값)  
scaled_df.groupby('cluster').mean()
```

```
Out[39]:  
child_num  income_total  DAYS_BIRTH  DAYS_EMPLOYED  work_phone  phone  email  family_size  begin_month  gender_F  gender_M  car_N  ca  
cluster  
0          0.027680     0.121509     0.575247     0.859244     0.238678     0.286255     0.098238     0.072208     0.554046     0.450980     0.549020     0.0  
1          0.019427     0.092566     0.497403     0.860595     0.216210     0.299147     0.087020     0.057349     0.571081     0.802316     0.197684     1.0
```

# 분류 예측 - 모델 학습

## 02. Clustering Analysis + DNN, DT, LR 모델 학습

### (1) Clustering Analysis(K-means)

- ⑩ sklearn.decomposition의 PCA를 import하여 PCA 진행. 데이터의 수많은 정보를 축약해 높은 차원의 데이터 값 변화를 효과적으로 설명
- ⑪ 가독성을 높이기 위해 데이터프레임으로 변환, clustering 결과 추가

```
In [40]: # PCA 진행
from sklearn.decomposition import PCA
X = scaled_df.copy()

pca = PCA(n_components=2)

pca.fit(X)
x_pca = pca.transform(X)
x_pca
```

```
Out[40]: array([[ 0.23879565, -1.4330941 ],
[-0.47228634,  1.07511336],
[ 1.58687387,  1.05515635],
...,
[-0.06633981, -0.30853223],
[ 0.22130603,  0.37761757],
[-0.06035094, -0.19428901]])
```

```
In [41]: # x_pca를 보기 쉽게 데이터프레임으로 변환
pca_df = pd.DataFrame(x_pca)
pca_df['cluster'] = scaled_df['cluster']
pca_df.head()
```

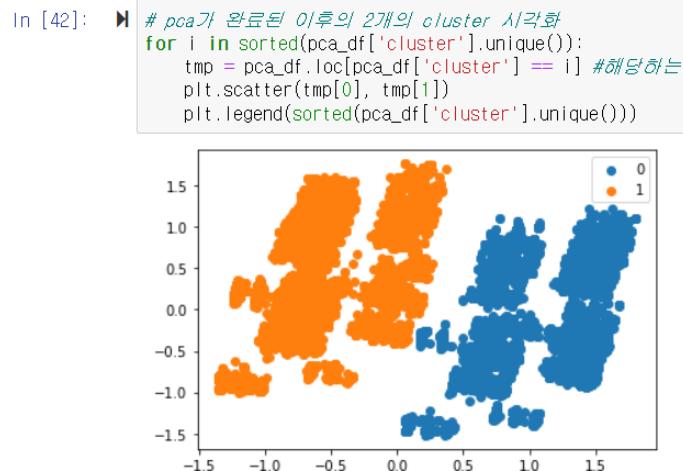
	0	1	cluster
0	0.238796	-1.433094	0
1	-0.472286	1.075113	1
2	1.586874	1.055156	0
3	0.814643	0.596772	0
4	1.411477	-0.444249	0

# 분류 예측 - 모델 학습

## 02. Clustering Analysis + DNN, DT, LR 모델 학습

### (1) Clustering Analysis(K-means)

⑫ PCA 진행 후 2개의 클러스터를 시각화



	child_num	income_total	DAY_S_BIRTH	DAY_S_EMPLOYED	work_phone	phone	email	family_size	begin_month	gender_F	gender_M	car_N	car_
0	0.00000	0.113372	0.350834	1.000000	0.0	0.0	0.0	0.052632	0.116667	1.0	0.0	0.0	1
1	0.052632	0.084302	0.537571	0.980335	0.0	1.0	0.0	0.105263	0.566667	1.0	0.0	1.0	0
2	0.00000	0.069767	0.668826	0.884427	1.0	1.0	0.0	0.052632	0.850000	0.0	1.0	0.0	1
3	0.105263	0.055233	0.738350	0.990454	0.0	1.0	0.0	0.157895	0.800000	1.0	0.0	0.0	1
4	0.052632	0.127907	0.514530	0.849106	0.0	0.0	0.0	0.105263	0.950000	0.0	1.0	0.0	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...
26452	0.000000	0.113372	0.733937	0.869089	0.0	0.0	0.0	0.052632	0.500000	1.0	0.0	1.0	0
26453	0.000000	0.078488	0.903880	0.869598	0.0	1.0	1.0	0.000000	0.600000	0.0	1.0	1.0	0
26454	0.000000	0.156977	0.587837	0.655063	0.0	0.0	1.0	0.052632	0.566667	0.0	1.0	1.0	0
26455	0.000000	0.244186	0.531381	0.695730	1.0	0.0	0.0	0.052632	0.500000	0.0	1.0	1.0	0
26456	0.000000	0.171512	0.846965	0.938777	0.0	0.0	0.0	0.052632	0.116667	0.0	1.0	1.0	0

26457 rows × 13 columns

# 분류 예측 - 모델 학습

## 02. Clustering Analysis + DNN, DT, LR 모델 학습

### (1) Clustering Analysis(K-means)

- ⑬ K-means 분석 결과를 원 데이터셋에 반영
- ⑭ isnull(), notnull()을 이용해 train, test 셋 분리

```
In [44]: # data에 'cluster' column 추가
data['cluster'] = scaled_df['cluster']
data
```

Out[44]:

	child_num	income_total	DAY_S_BIRTH	DAY_S_EMPLOYED	work_phone	phone	email	family_size	begin_month	credit	gender_F	gender_M	car_
0	0	202500.0	-19031	0	0	0	0	2	-53	1.0	1	0	
1	1	157500.0	-15773	-309	0	1	0	3	-26	0.0	1	0	
2	0	135000.0	-13483	-1816	1	1	0	2	-9	1.0	0	1	
3	2	112500.0	-12270	-150	0	1	0	4	-12	1.0	1	0	
4	1	225000.0	-16175	-2371	0	0	0	3	-3	1.0	0	1	
...	...	...	...	...	...	...	...	...	...	...	...	...	...
26452	0	202500.0	-12347	-2057	0	0	0	2	-30	NaN	1	0	
26453	0	148500.0	-9382	-2049	0	1	1	1	-24	NaN	0	1	
26454	0	270000.0	-14896	-5420	0	0	1	2	-26	NaN	0	1	
26455	0	405000.0	-15881	-4781	1	0	0	2	-30	NaN	0	1	
26456	0	292500.0	-10375	-962	0	0	0	2	-53	NaN	0	1	

26457 rows × 14 columns

```
In [45]: # 다시 원래대로 train(credit 값 존재)과 test(credit 값 NaN) 분리
train_data = data.loc[data['credit'].notnull()]
test_data = data.loc[data['credit'].isnull()]
```

# 분류 예측 - 모델 학습

## 02. Clustering Analysis + DNN, DT, LR 모델 학습

### (2) Deep Neuron Network(DNN) - MLPClassifier 사용

- ① MLPClassifier 사용하여 DNN 진행
- ② 그리드 서치(GridSearchCV 사용)를 통한 하이퍼파라미터 튜닝 진행
- ③ 최적 하이퍼파라미터 : activation function으로 tanh 함수 사용
- ④ 검증 데이터의 accuracy 확인 : 약 **0.8445**의 정확도

```
In [50]: # from sklearn.neural_network import MLPClassifier
mlp = MLPClassifier(random_state=42)

In [51]: # 하이퍼파라미터 풀기
mlp.get_params().keys()

Out[51]: dict_keys(['activation', 'alpha', 'batch_size', 'beta_1', 'beta_2', 'early_stopping', 'epsilon', 'hidden_layer_sizes', 'learning_rate', 'learning_rate_init', 'max_fun', 'max_iter', 'momentum', 'n_iter_no_change', 'nesterovs_momentum', 'power_t', 'random_state', 'shuffle', 'solver', 'tol', 'validation_fraction', 'verbose', 'warm_start'])

In [52]: # 그리드 서치 -> 하이퍼파라미터 풀기 진행
from sklearn.model_selection import GridSearchCV

mlp = MLPClassifier(random_state=42)
para1 = {'activation': ['tanh', 'relu']}

mlp = GridSearchCV(estimator = mlp, param_grid = para1, cv=5, scoring='accuracy')
mlp.fit(over_X_sd, over_y)

Out[52]: GridSearchCV(cv=5, estimator=MLPClassifier(random_state=42),
param_grid={'activation': ['tanh', 'relu']}, scoring='accuracy')

In [53]: # 최적 하이퍼파라미터 확인
mlp.best_params_

Out[53]: {'activation': 'tanh'}
```

# 분류 예측 - 모델 학습

## 02. Clustering Analysis + DNN, DT, LR 모델 학습

### (2) Deep Neuron Network(DNN) - 직접 구현

#### ① 모델 구조 생성

- Sequential 지정 후 model.add()를 통해 은닉층 추가
- 출력 뉴런과 연결된 Dense 층 마지막에 추가

#### ② Compile

- 모델 학습 전 모델 학습 환경 설정
- optimizer로 rmsprop을 사용
- 손실함수 loss로 binary\_crossentropy 사용
- metrics로 acc 사용

#### ③ Early Stopping 생성, Fitting

- epoch와 batch\_size를 지정 후 모델을 학습

```

In [55]: from tensorflow import keras
         from tensorflow.keras import models
         from tensorflow.keras import layers

model = models.Sequential()
model.add(layers.Dense(4, activation='relu', input_shape=(57,)))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

In [56]: # Compiling
         model.compile(optimizer = 'rmsprop', loss = 'binary_crossentropy', metrics = 'acc')

In [57]: # earlystopping 설정
         early_stopping = keras.callbacks.EarlyStopping(patience=10, min_delta=0.001, restore_best_weights=True)

         history = model.fit(over_X_sd, over_y,
                             validation_data=(val_X_sd, val_y),
                             batch_size=32, epochs=100, callbacks=[early_stopping])

91/91 [=====] - 2s 4ms/step - loss: 0.2940 - acc: 0.8833 - val_loss: 0.3934 - val_acc: 0.8748
Epoch 12/100
91/91 [=====] - 2s 4ms/step - loss: 0.2929 - acc: 0.8838 - val_loss: 0.3906 - val_acc: 0.8758
Epoch 13/100

```

# 분류 예측 - 모델 학습

## 02. Clustering Analysis + DNN, DT, LR 모델 학습

### (2) Deep Neuron Network(DNN) - 직접 구현

#### ④ loss, accuracy 시각화

- Best Validation Loss : 0.3884
- Best Validation Accuracy : **0.8758**

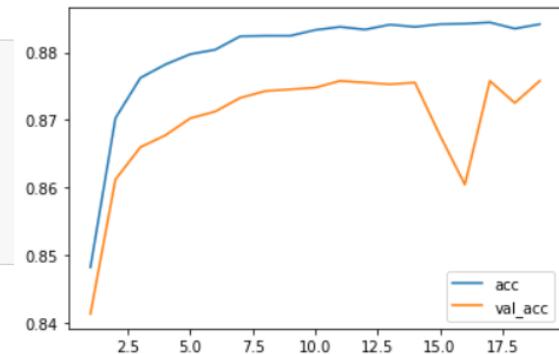
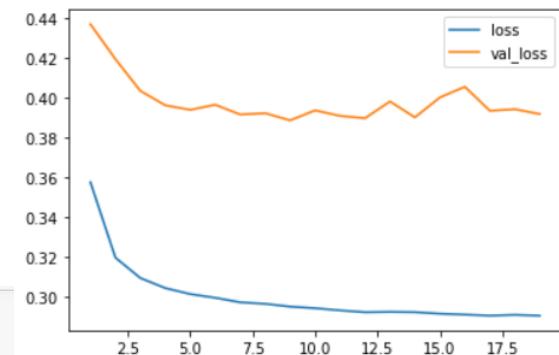
```
In [58]: history_dict = history.history
history_dict.keys()

Out[58]: dict_keys(['loss', 'acc', 'val_loss', 'val_acc'])
```

```
In [59]: # loss, accuracy 시각화
history_df = pd.DataFrame(history.history)
history_df.loc[1:, ['loss', 'val_loss']].plot()
history_df.loc[1:, ['acc', 'val_acc']].plot()

print(("Best Validation Loss: {:.4f}" +#
      "#\nBest Validation Accuracy: {:.4f}").format(
          history_df['val_loss'].min(), history_df['val_acc'].max()))
```

Best Validation Loss: 0.3884  
 Best Validation Accuracy: 0.8758



# 분류 예측 - 모델 학습

## 02. Clustering Analysis + DNN, DT, LR 모델 학습

### (3) Decision Tree(DT)

- ① DecisionTreeClassifier 사용하여 DT 진행
- ② 그리드 서치(GridSearchCV 사용)를 통한 하이퍼파라미터 튜닝 진행
- ③ 최적 하이퍼파라미터 → criterion = 'entropy', max\_depth = 20
- ④ 검증 데이터의 accuracy 확인 : 약 **0.8110**의 정확도

```
In [60]: # decision tree 클래스 지정
from sklearn.tree import DecisionTreeClassifier
dt = DecisionTreeClassifier(random_state=42)
```

```
In [61]: # 하이퍼파라미터 설정
dt.get_params().keys()
```

```
Out[61]: dict_keys(['ccp_alpha', 'class_weight', 'criterion', 'max_depth', 'max_features', 'max_leaf_nodes', 'min_impurity_decrease', 'min_samples_leaf', 'min_samples_split', 'min_weight_fraction_leaf', 'random_state', 'splitter'])
```

```
In [62]: # 그리드 서치 -> 하이퍼파라미터 튜닝 진행
dt = DecisionTreeClassifier(random_state=42)
para2 = {'criterion': ['gini', 'entropy'], 'max_depth': [5, 10, 15, 20]}

dt = GridSearchCV(estimator = dt, param_grid = para2, cv=5, scoring='accuracy')
dt.fit(over_X_sd, over_y)
```

```
Out[62]: GridSearchCV(cv=5, estimator=DecisionTreeClassifier(random_state=42),
param_grid={'criterion': ['gini', 'entropy'],
'max_depth': [5, 10, 15, 20]},
scoring='accuracy')
```

```
In [63]: # 최적 하이퍼파라미터 확인
dt.best_params_
```

```
Out[63]: {'criterion': 'entropy', 'max_depth': 20}
```

```
In [64]: # 검증 데이터 accuracy 확인
dt = DecisionTreeClassifier(criterion='entropy', max_depth=20, random_state=42)
dt.fit(over_X_sd, over_y)
pred = dt.predict(val_X_sd)
print(accuracy_score(val_y, pred))
```

```
0.8110355253212396
```

# 분류 예측 - 모델 학습

## 02. Clustering Analysis + DNN, DT, LR 모델 학습

### (4) Logistic Regression(LR)

- ① LogisticRegression 사용하여 LR 진행
- ② 그리드 서치(GridSearchCV 사용)를 통한 하이퍼파라미터 튜닝 진행
- ③ 최적 하이퍼파라미터 → C = 1, penalty = 'l2'
- ④ 검증 데이터의 accuracy 확인 : 약 **0.8778**의 정확도

```
In [65]: # logistic regression 클래스 지정
from sklearn.linear_model import LogisticRegression

lr = LogisticRegression(random_state=42)
```

```
In [66]: # 하이퍼파라미터 품색
lr.get_params().keys()
```

```
Out[66]: dict_keys(['C', 'class_weight', 'dual', 'fit_intercept', 'intercept_scaling', 'l1_ratio', 'max_iter', 'multi_class', 'n_jobs', 'penalty', 'random_state', 'solver', 'tol', 'verbose', 'warm_start'])
```

```
In [67]: # 그리드 서치 -> 하이퍼파라미터 튜닝 진행
lr = LogisticRegression(random_state=42)
para3 = {'C': [0.001, 0.01, 0.1, 1, 10, 100], 'penalty': ['l1', 'l2']}

lr = GridSearchCV(estimator = lr, param_grid = para3, cv=5, scoring='accuracy')
lr.fit(over_X_sd, over_y)
```

```
Out[67]: GridSearchCV(cv=5, estimator=LogisticRegression(random_state=42),
param_grid={'C': [0.001, 0.01, 0.1, 1, 10, 100],
'penalty': ['l1', 'l2']},
scoring='accuracy')
```

```
In [68]: # 최적 하이퍼파라미터 확인
lr.best_params_
```

```
Out[68]: {'C': 1, 'penalty': 'l2'}
```

```
In [69]: # 검증 데이터 accuracy 확인
lr = LogisticRegression(C=1, penalty='l2', random_state=42)
lr.fit(over_X_sd, over_y)
pred = lr.predict(val_X_sd)
print(accuracy_score(val_y, pred))
```

0.8778029730410682

# 분류 예측 - 모델 학습

## 02. Clustering Analysis + DNN, DT, LR 모델 학습

### (5) 모델별 성능 비교

- DNN(MLPClassifier) : 0.8445452254976065
- DNN(직접 구현) : 0.8758
- Decision Tree : 0.8110355253212396
- Logistic Regression : 0.8778029730410682

❖ Kmeans method Clustering을 적용한 후의 DNN, Decision Tree, Logistic Regression 중에서는 **Kmeans + LR**이 약 **0.8778**으로 성능이 가장 우수

# 분류 예측 - 모델 학습

## 03. SOM + DNN, DT, LR 모델 학습

### (1) SOM - 모델 생성 및 비교

- ① MiniSom 사용하여 SOM 진행
- ② 원하는 파라미터 조합하여 리스트화함
- ③ 모든 조합에 대해 모델 생성 및 랜덤으로 초기값을 설정하는 경우, pca로 초기값을 설정하는 경우의 qe, te값 계산
- ④ 결과 데이터프레임 생성, sorting, 시각화 진행

```
In [34]: from minisom import MiniSom
s_time = pd.Timestamp.now()
print('시작시간:', s_time, '\n')

# 원하는 파라미터 조합 리스트화
map_n = [n for n in range(2,6)]
para_sigma = [np.round(sigma*0.1,2) for sigma in range(1,10)]
para_learning_rate = [np.round(learning_rate*0.1,2) for learning_rate in range(1,10)]

# 결과 값을 담을 리스트 res 생성
res = []
# 모든 조합에 대해 모델 생성 및 qe,te값 계산
for n in map_n:
    for sigma in para_sigma:
        for lr in para_learning_rate:
            try:
                # 랜덤으로 초기값을 설정하는 경우
                estimator = MiniSom(n,56,sigma = sigma, learning_rate = lr, topology='hexagonal',random_seed=42)
                estimator.random_weights_init(scaled_df.values)
                estimator.train(scaled_df.values,1000,random_order=True)
                qe = estimator.quantization_error(scaled_df.values)
                # te = estimator.topographic_error(scaled_df.values)
                winner_coordinates = np.array([estimator.winner(x) for x in scaled_df.values]).T
                cluster_index = np.ravel_multi_index(winner_coordinates,(n,n))

                res.append([str(n)+x+str(sigma),str(lr),'random_init',qe,len(np.unique(cluster_index))])

            except ValueError as e:
                print(e)

# pca로 초기값을 설정하는 경우
estimator = MiniSom(n,56,sigma = sigma, learning_rate = lr,topology='hexagonal', random_seed=42)
estimator.pca_weights_init(scaled_df.values)
estimator.train(scaled_df.values,1000,random_order=True)
qe = estimator.quantization_error(scaled_df.values)
# te = estimator.topographic_error(scaled_df.values)
winner_coordinates = np.array([estimator.winner(x) for x in scaled_df.values]).T
cluster_index = np.ravel_multi_index(winner_coordinates,(n,n))

res.append([str(n)+x+str(sigma),str(lr),'pca_init',qe,len(np.unique(cluster_index))])
```

```
except ValueError as e:
    print(e)

# 결과 데이터프레임 생성 및 sorting
df_res = pd.DataFrame(res,columns=['map_size','sigma','learning_rate','init_method','qe','n_cluster'])
df_res.shape
df_res.sort_values(by=['qe'],ascending=True,inplace=True,ignore_index=True)
df_res.head(10)

# 시각화를 위한 Lineplot 생성
plt.figure(figsize=(20,10))
sns.lineplot(data = df_res)

e_time = pd.Timestamp.now()
print('\n종료시간:',e_time, '\n총 소요시간:',e_time-s_time)

시작시간: 2022-11-27 17:12:35.244168
종료시간: 2022-11-27 17:12:35.244168
총 소요시간: 0:00:00.000000

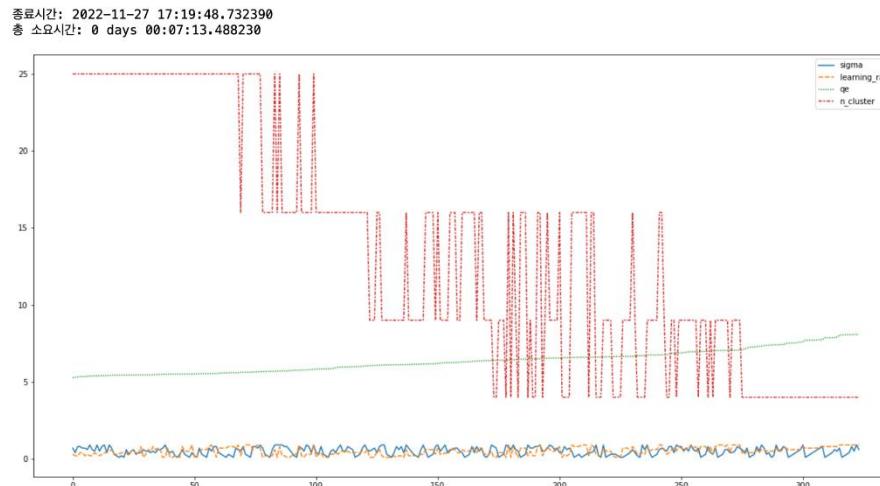
Received 56 features, expected 4.
```

# 분류 예측 - 모델 학습

## 03. SOM + DNN, DT, LR 모델 학습

### (1) SOM - 모델 생성 및 비교

- ① MiniSom 사용하여 SOM 진행
- ② 원하는 파라미터 조합하여 리스트화함
- ③ 모든 조합에 대해 모델 생성 및 랜덤으로 초기값을 설정하는 경우, pca로 초기값을 설정하는 경우의 qe, te값 계산
- ④ 결과 데이터프레임 생성, sorting, 시각화 진행



# 분류 예측 - 모델 학습

## 03. SOM + DNN, DT, LR 모델 학습

### (1) SOM - 모델 생성 및 비교

- SOM 결과 데이터프레임 정보는 다음과 같음

In [35]: df\_res

Out[35]:

	map_size	sigma	learning_rate	init_method	qe	n_cluster
0	5x5	0.7	0.3	random_init	5.270559	25
1	5x5	0.4	0.2	random_init	5.305664	25
2	5x5	0.8	0.2	random_init	5.315885	25
3	5x5	0.8	0.4	random_init	5.337061	25
4	5x5	0.7	0.4	random_init	5.337695	25
...	...	...	...	...	...	...
319	2x2	0.4	0.9	random_init	8.056978	4
320	2x2	0.8	0.9	random_init	8.066916	4
321	2x2	0.5	0.9	random_init	8.073240	4
322	2x2	0.9	0.9	random_init	8.075346	4
323	2x2	0.6	0.9	random_init	8.078343	4

324 rows × 6 columns

In [38]: # cluster 총 개수별 군집화 종류 개수  
df\_res['n\_cluster'].value\_counts()

Out[38]:

25	81
16	81
9	81
4	81

Name: n\_cluster, dtype: int64

In [39]: df\_res[df\_res['n\_cluster']==4]

Out[39]:

	map_size	sigma	learning_rate	init_method	qe	n_cluster
173	2x2	0.8	0.1	random_init	6.391071	4
174	2x2	0.7	0.1	random_init	6.395825	4
178	2x2	0.9	0.1	random_init	6.413493	4
180	2x2	0.6	0.1	random_init	6.417253	4
183	2x2	0.5	0.1	random_init	6.472109	4
...	...	...	...	...	...	...
319	2x2	0.4	0.9	random_init	8.056978	4
320	2x2	0.8	0.9	random_init	8.066916	4
321	2x2	0.5	0.9	random_init	8.073240	4
322	2x2	0.9	0.9	random_init	8.075346	4
323	2x2	0.6	0.9	random_init	8.078343	4

81 rows × 6 columns

# 분류 예측 - 모델 학습

## 03. SOM + DNN, DT, LR 모델 학습

### (1) SOM - 모델 파라미터 조정

- ① qe 값이 최소인 하이퍼파라미터 기준으로 SOM 진행
- ② 모델의 초기값 설정 및 모델 평가
- ③ 4개의 cluster에 배정된 cluster별 data 개수 count

```
In [40]: # qe값이 최소인 하이퍼파라미터 기준 SOM 진행
som_b2 = MiniSom(2,2,56,sigma=0.8,learning_rate=0.1,topology='hexagonal',neighborhood_function='gaussian',activation
```

```
# 초기값설정
som_b2.pca_weights_init(scaled_df.values)
som_b2.train(scaled_df.values,1000,random_order=True)
```

```
# 평가
som_b2.quantization_error(scaled_df.values)
#som_b2.topographic_error(data.values)
```

```
Out[40]: 6.42072234413208
```

```
In [41]: winner_coordinates = np.array([som_b2.winner(x) for x in scaled_df.values]).T
cluster_index = np.ravel_multi_index(winner_coordinates,(n,n))
```

```
In [42]: # 4개의 cluster에 배정된 cluster별 data 개수
pd.DataFrame(cluster_index).value_counts()
```

```
Out[42]: 5    9645
0    7335
6    5038
1    4439
dtype: int64
```

# 분류 예측 - 모델 학습

## 03. SOM + DNN, DT, LR 모델 학습

### (1) SOM - 시각화

- ① RegularPolygon, cm, make\_axes\_locatable, ColorbarBase를 import하여 시각화 진행

```
In [43]: from matplotlib.patches import RegularPolygon
from matplotlib import cm
from mpl_toolkits.axes_grid1 import make_axes_locatable
from matplotlib.colorbar import ColorbarBase
```

```
In [44]: xx, yy = som_b2.get_euclidean_coordinates()
umatrix = som_b2.distance_map()
weights = som_b2.get_weights()

f = plt.figure(figsize=(10,10))
ax = f.add_subplot(111)
ax.set_aspect('equal')

# iteratively add hexagons
# plotting the distance map as background
# 해당 결과 다른 이웃들 간 거리를 표현, 밝을수록 가깝고, 어두울수록 멀다
for i in range(weights.shape[0]):
    for j in range(weights.shape[1]):
        wy = yy[[i, j]] * 2 / np.sqrt(3) * 3 / 4
        hex = RegularPolygon((xx[[i, j]], wy),
                             numVertices=6,
                             radius=.95 / np.sqrt(3),
                             facecolor=cm.Blues(umatrix[i, j]),
                             alpha=.4,
                             edgecolor='gray')
        plot = ax.add_patch(hex)

#output노드에 해당하는 클러스터 종류 및 밀도 확인
cnt=[]
for c in np.unique(cluster_index):
    x_= [som_b2.convert_map_to_euclidean(som_b2.winner(x))[0] + (2*np.random.rand(1)[0]-1)*0.4 for x in scaled_d]
    y_= [som_b2.convert_map_to_euclidean(som_b2.winner(x))[1] + (2*np.random.rand(1)[0]-1)*0.4 for x in scaled_d]
    y_= [(i* 2 / np.sqrt(3) * 3 / 4) for i in y_]

    plot = sns.scatterplot( x = x_, y = y_ ,label='cluster='+str(c),alpha=.7)
```

```
#클러스터에 속한 데이터 개수 데이터프레임으로 출력
cnt.append([c,len(x_)])
```

```
#클러스터별 개수를 표 형태로 출력
df_cnt = pd.DataFrame(cnt,columns=['cluster이름', '개수'])

#x축,y축 간격 설정
xrange = np.arange(weights.shape[0])
yrange = np.arange(weights.shape[1])
plot = plt.xticks(xrange-.5, xrange)
plot = plt.yticks(yrange * 2 / np.sqrt(3) * 3 / 4, yrange)

#차트 우측에 color bar생성
divider = make_axes_locatable(plt.gca())
ax_cb = divider.new_horizontal(size="5%", pad=0.05)
cb1 = ColorbarBase(ax_cb, cmap=cm.Blues,
                    orientation='vertical', alpha=.4)
cb1.ax.get_yaxis().labelpad = 16
plot = cb1.ax.set_ylabel('distance from neurons in the neighborhood',
                        rotation=270, fontsize=16)
plot = plt.gcf().add_axes(ax_cb)

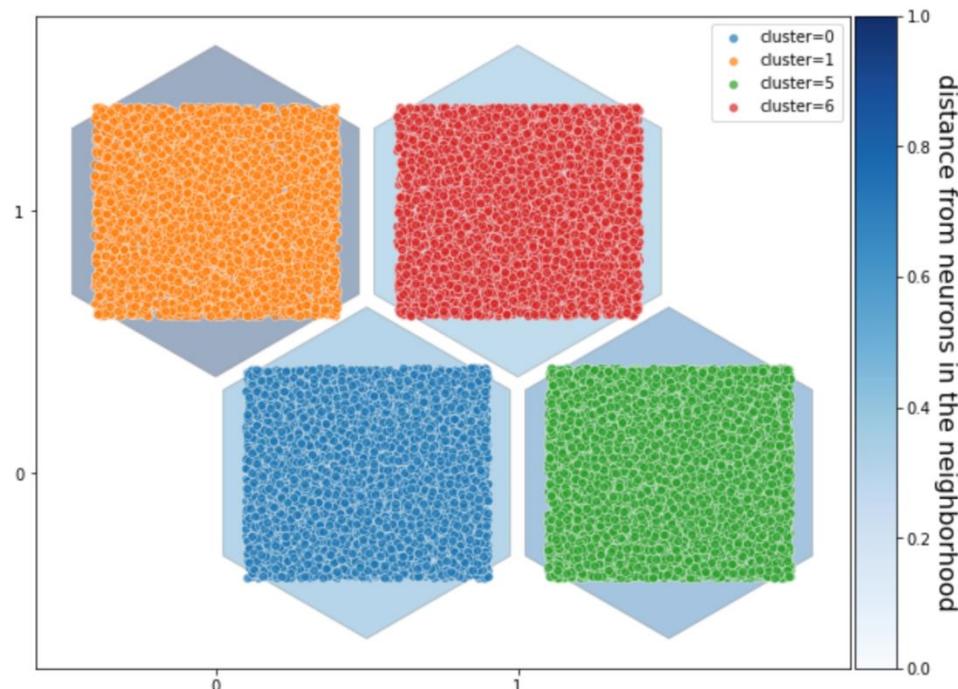
#이미지 저장
plt.savefig('som_seed_hex.png')
```

# 분류 예측 - 모델 학습

## 03. SOM + DNN, DT, LR 모델 학습

### (1) SOM - 시각화

- 시각화의 결과는 다음과 같음



# 분류 예측 - 모델 학습

## 03. SOM + DNN, DT, LR 모델 학습

### (1) SOM - 시각화 후 데이터 처리

- ① 클러스터링 변수인 clusters 값을 원본 데이터인 'scaled\_df'내에 넣음
- ② cluster를 기준으로 데이터 개수를 셈 결과, 0번 cluster에는 7335개, 1번 cluster에는 4439개, 5번 cluster에는 9645개, 6번 cluster에는 5038개의 데이터가 있는 것을 확인할 수 있음
- ③ 그룹별 평균값을 통해 그룹별 특징을 확인해봄

```
In [45]: #클러스터링 변수인 clusters 값을 원본 데이터인 'scaled_df' 내에 넣기
scaled_df['cluster'] = cluster_index
scaled_df.head()
```

```
Out[45]:
child_num income_total DAYS_BIRTH DAYS_EMPLOYED work_phone phone email family_size begin_month gender_F gender_M car_N
0 -0.573599 0.149136 -0.731391 0.927612 -0.538417 -0.645705 -0.316937 -0.214735 -1.623064 0.703562 -0.703562 -1.278015 1
1 0.764529 -0.292575 0.044045 0.797238 -0.538417 1.548696 -0.316937 0.876135 0.007446 0.703562 -0.703562 0.782463 -0
2 -0.573599 -0.513431 0.589087 0.161398 1.857295 1.548696 -0.316937 -0.214735 1.034063 -1.421339 1.421339 -1.278015 1
3 2.102658 -0.734287 0.877793 0.864324 -0.538417 1.548696 -0.316937 1.967005 0.852895 0.703562 -0.703562 -1.278015 1
4 0.764529 0.369992 -0.051635 -0.072769 -0.538417 -0.645705 -0.316937 0.876135 1.396399 -1.421339 1.421339 -1.278015 1
```

```
In [46]: # cluster를 기준으로 데이터 개수 세기
scaled_df.groupby('cluster').count()
```

```
Out[46]:
child_num income_total DAYS_BIRTH DAYS_EMPLOYED work_phone phone email family_size begin_month gender_F gender_M car_N car_Y
cluster
0 7335 7335 7335 7335 7335 7335 7335 7335 7335 7335 7335 7335 7335
1 4439 4439 4439 4439 4439 4439 4439 4439 4439 4439 4439 4439 4439
5 9645 9645 9645 9645 9645 9645 9645 9645 9645 9645 9645 9645 9645
6 5038 5038 5038 5038 5038 5038 5038 5038 5038 5038 5038 5038 5038
```

```
In [47]: #그룹별 특징을 알아보기(그룹별 평균값)
scaled_df.groupby('cluster').mean()
```

```
Out[47]:
child_num income_total DAYS_BIRTH DAYS_EMPLOYED work_phone phone email family_size begin_month gender_F gender_M car_
cluster
0 0.029516 0.420785 0.396421 -0.022516 0.168048 -0.020742 0.053708 0.103082 -0.009366 -1.325161 1.325161 -0.71705
1 -0.510295 -0.370167 -1.383632 0.927529 -0.538417 -0.015414 -0.204902 -0.498327 0.015064 0.390020 -0.390020 0.35727
5 0.379947 -0.110744 0.314321 -0.215606 0.129501 0.058460 0.060695 0.537507 -0.008120 0.560800 0.560800 0.12041
6 -0.320741 -0.074467 0.040209 -0.371702 -0.018189 -0.068139 -0.014383 -0.740033 0.015908 0.512076 -0.512076 0.49862
```

# 분류 예측 - 모델 학습

## 03. SOM + DNN, DT, LR 모델 학습

### (1) SOM - 시각화 후 PCA 진행

- ① PCA 라이브러리 활용하여 PCA 진행
- ② PCA가 완료된 이후의 2개의 cluster 시각화
- ③ 데이터에 SOM 분석 결과 column을 추가함으로써 군집 분석 결과 반영
- ④ 다시 원래대로 train(credit 값 존재)과 test(credit 값 NaN) 분리

```
In [48]: # PCA 진행
from sklearn.decomposition import PCA
X = scaled_df.copy()

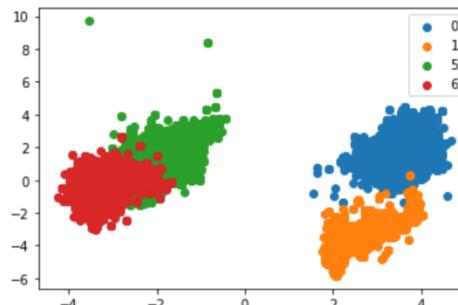
pca = PCA(n_components=2)
pca.fit(X)
x_pca = pca.transform(X)
x_pca
```

```
Out[48]: array([[ 2.77498254, -2.89177319],
 [-2.38412371,  0.75560108],
 [ 3.83805293,  2.03475776],
 ...,
 [ 3.53365153,  0.73092056],
 [ 3.19694863,  1.51047434],
 [ 3.33853694,  0.52548338]])
```

```
In [49]: # x_pca를 보기 쉽게 데이터프레임으로 변환
pca_df = pd.DataFrame(x_pca)
pca_df['cluster'] = scaled_df['cluster']
pca_df.head()
```

```
Out[49]:   0      1  cluster
0  2.774983 -2.891773      1
1 -2.384124  0.755601      5
2  3.838053  2.034758      0
3 -1.568297  1.948583      5
4  4.234423  1.858138      0
```

```
In [50]: # pca가 완료된 이후의 2개의 cluster 시각화
for i in sorted(pca_df['cluster'].unique()):
    tmp = pca_df.loc[pca_df['cluster'] == i] # 해당하는 클러스터 번호일 때 그림을 그리고, for문 실행하며 위에 덧그림
    plt.scatter(tmp[0], tmp[1])
plt.legend(sorted(pca_df['cluster'].unique()))
```



# 분류 예측 - 모델 학습

## 03. SOM + DNN, DT, LR 모델 학습

### (1) SOM - 시각화 후 PCA 진행

- ① PCA 라이브러리 활용하여 PCA 진행
- ② PCA가 완료된 이후의 2개의 cluster 시각화
- ③ 데이터에 SOM 분석 결과 column을 추가함으로써 군집 분석 결과 반영
- ④ 다시 원래대로 train(credit 값 존재)과 test(credit 값 NaN) 분리

In [51]: scaled\_df

Out[51]:

	child_num	income_total	DAYS.BIRTH	DAYS_EMPLOYED	work_phone	phone	email	family_size	begin_month	gender_F	gender_M	car_N
0	-0.573599	0.149136	-0.731391	0.927612	-0.538417	-0.645705	-0.316937	-0.214735	-1.623064	0.703562	-0.703562	-1.27801
1	0.764529	-0.292575	0.044045	0.797238	-0.538417	1.548696	-0.316937	0.876135	0.007446	0.703562	-0.703562	0.78246
2	-0.573599	-0.513431	0.589087	0.161398	1.857295	1.548696	-0.316937	-0.214735	1.034063	-1.421339	1.421339	-1.27801
3	2.102658	-0.734287	0.877793	0.864324	-0.538417	1.548696	-0.316937	1.967005	0.852895	0.703562	-0.703562	-1.27801
4	0.764529	0.369992	-0.051635	-0.072769	-0.538417	-0.645705	-0.316937	0.876135	1.396399	-1.421339	1.421339	-1.27801
...	...	...	...	...	...	...	...	...	...	...	...	...
26452	-0.573599	0.149136	0.859466	0.059715	-0.538417	-0.645705	-0.316937	-0.214735	-0.234111	0.703562	-0.703562	0.78246
26453	-0.573599	-0.380918	1.565165	0.063090	-0.538417	1.548696	3.155199	-1.305605	0.128224	-1.421339	1.421339	0.78246
26454	-0.573599	0.811704	0.252779	-1.359215	-0.538417	-0.645705	3.155199	-0.214735	0.007446	-1.421339	1.421339	0.78246
26455	-0.573599	2.136838	0.018340	-1.089606	1.857295	-0.645705	-0.316937	-0.214735	-0.234111	-1.421339	1.421339	0.78246
26456	-0.573599	1.032559	1.326821	0.521721	-0.538417	-0.645705	-0.316937	-0.214735	-1.623064	-1.421339	1.421339	0.78246

26457 rows × 57 columns

In [52]: # data에 'cluster' column 추가  
data['cluster'] = scaled\_df['cluster']  
data

Out[52]:

	child_num	income_total	DAYS.BIRTH	DAYS_EMPLOYED	work_phone	phone	email	family_size	begin_month	credit	gender_F	gender_M	car_N	c
0	0	202500.0	-19031	0	0	0	0	2	-53	1.0	1	0	0	
1	1	157500.0	-15773	-309	0	1	0	3	-26	0.0	1	0	1	
2	0	135000.0	-13483	-1816	1	1	0	2	-9	1.0	0	1	0	
3	2	112500.0	-12270	-150	0	1	0	4	-12	1.0	1	0	0	
4	1	225000.0	-16176	-2371	0	0	0	3	-3	1.0	0	1	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
26452	0	202500.0	-12347	-2057	0	0	0	2	-30	NaN	1	0	1	
26453	0	148500.0	-9382	-2049	0	1	1	1	-24	NaN	0	1	1	
26454	0	270000.0	-14896	-5420	0	0	1	2	-26	NaN	0	1	1	
26455	0	405000.0	-15881	-4781	1	0	0	2	-30	NaN	0	1	1	
26456	0	292500.0	-10375	-962	0	0	0	2	-53	NaN	0	1	1	

26457 rows × 58 columns

In [53]: # 다시 원래대로 train(credit 값 존재)과 test(credit 값 NaN) 분리  
train\_data = data.loc[data['credit'].notnull()]  
test\_data = data.loc[data['credit'].isnull()]

# 분류 예측 - 모델 학습

## 03. SOM + DNN, DT, LR 모델 학습

### (2) Deep Neuron Network(DNN) - MLPClassifier 사용

- ① MLPClassifier 사용하여 DNN 진행
- ② 그리드 서치(GridSearchCV 사용)를 통한 하이퍼파라미터 튜닝 진행
- ③ 최적 하이퍼파라미터 : activation function으로 tanh 함수 사용
- ④ 검증 데이터의 accuracy 확인 : 약 **0.8438**의 정확도

```
In [58]: from sklearn.neural_network import MLPClassifier
mlp = MLPClassifier(random_state=42)

In [59]: # 하이퍼파라미터 탐색
mlp.get_params().keys()

Out[59]: dict_keys(['activation', 'alpha', 'batch_size', 'beta_1', 'beta_2', 'early_stopping', 'epsilon', 'hidden_layer_sizes', 'learning_rate', 'learning_rate_init', 'max_fun', 'max_iter', 'momentum', 'n_iter_no_change', 'nesterovs_momentum', 'power_t', 'random_state', 'shuffle', 'solver', 'tol', 'validation_fraction', 'verbose', 'warm_start'])

In [60]: # 그리드 서치 -> 하이퍼파라미터 튜닝 진행
from sklearn.model_selection import GridSearchCV

mlp = MLPClassifier(random_state=42)
para1 = {'activation': ['tanh', 'relu']}

mlp = GridSearchCV(estimator = mlp, param_grid = para1, cv=5, scoring='accuracy')
mlp.fit(over_X_sd, over_y)

Out[60]: GridSearchCV(cv=5, estimator=MLPClassifier(random_state=42),
param_grid={'activation': ['tanh', 'relu']}, scoring='accuracy')

In [61]: # 최적 하이퍼파라미터 확인
mlp.best_params_

Out[61]: {'activation': 'tanh'}

In [62]: # 검증 데이터 accuracy 확인
from sklearn.metrics import accuracy_score

mlp = MLPClassifier(activation='tanh', random_state=42).fit(over_X_sd, over_y)
pred = mlp.predict(val_X_sd)
accuracy_score(val_y, pred)

Out[62]: 0.8437893675988914
```

## 분류 예측 - 모델 학습

## 03. SOM + DNN, DT, LR 모델 학습

## (2) Deep Neuron Network(DNN) - 직접 구현

## ① 모델 구조 생성

- Sequential 지정 후 model.add()를 통해 은닉층 추가
  - 출력 뉴런과 연결된 Dense 층을 마지막에 추가

## ② Compile

- 모델 학습 전 모델 학습 환경 설정
  - optimizer로 rmsprop을 사용
  - 손실함수 loss로 binary\_crossentropy 사용
  - metrics로 acc 사용

### ③ Early Stopping 생성, Fitting

- epoch와 batch\_size를 지정 후 모델을 학습

```
In [63]: # 모델 구조
from tensorflow import keras
from tensorflow.keras import models
from tensorflow.keras import layers

model = models.Sequential()
model.add(layers.Dense(4, activation='relu', input_shape=(57,)))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

```
In [64]: # Compiling
model.compile(optimizer = 'rmsprop', loss = 'binary_crossentropy', metrics = 'acc')
```

```
In [65]: # earlystopping 설정  
early_stopping = keras.callbacks.EarlyStopping(patience=10, min_delta=0.001, restore_best_weights=True)  
  
history = model.fit(over_X_sd, over_y,  
                    validation_data=(val_X_sd, val_y),  
                    batch_size=32, epochs=100, callbacks=[early_stopping])
```

```
Epoch 1/100  
472/472 [=====] - 4s 5ms/step - loss: 0.5637 - acc: 0.7356 - val_loss: 0.5443 - val_acc: 0  
.8325  
Epoch 2/100  
472/472 [=====] - 2s 4ms/step - loss: 0.3666 - acc: 0.8592 - val_loss: 0.4357 - val_acc: 0  
.8686  
Epoch 3/100  
472/472 [=====] - 2s 4ms/step - loss: 0.3253 - acc: 0.8713 - val_loss: 0.4216 - val_acc: 0  
.8667  
Epoch 4/100  
472/472 [=====] - 2s 4ms/step - loss: 0.3134 - acc: 0.8746 - val_loss: 0.4026 - val_acc: 0  
.8702  
Epoch 5/100  
472/472 [=====] - 2s 4ms/step - loss: 0.3075 - acc: 0.8758 - val_loss: 0.3983 - val_acc: 0  
.8720  
Epoch 6/100  
472/472 [=====] - 2s 4ms/step - loss: 0.3035 - acc: 0.8783 - val_loss: 0.4104 - val_acc: 0  
.8584  
Epoch 7/100
```

```
In [66]: history_dict = history.history  
history_dict.items()
```

```
Out[66]: dict_keys(['loss', 'acc', 'val_loss', 'val_acc'])
```

# 분류 예측 - 모델 학습

## 03. SOM + DNN, DT, LR 모델 학습

### (2) Deep Neuron Network(DNN) - 직접 구현

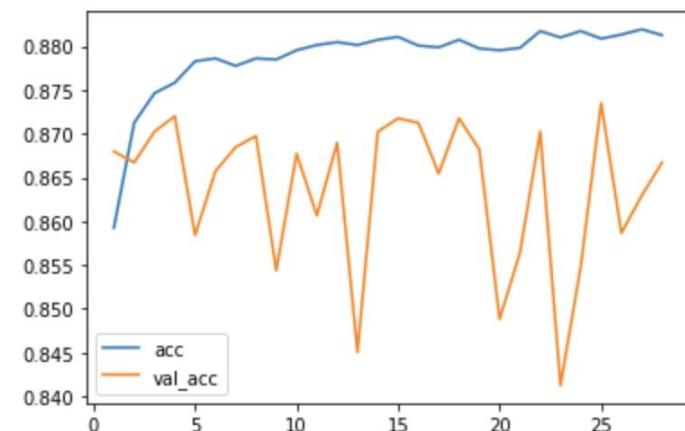
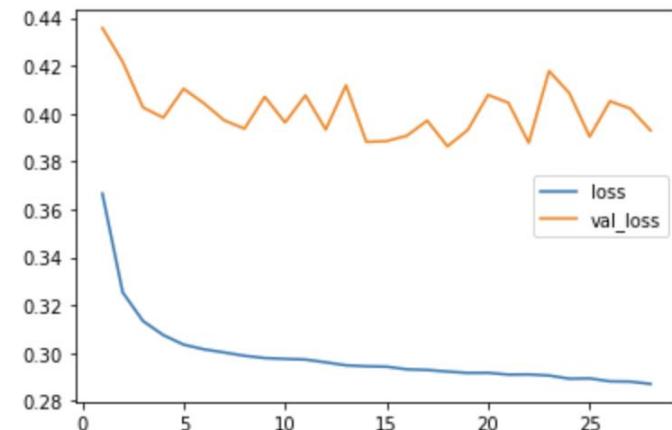
#### ④ loss, accuracy 시각화

- Best Validation Loss : 0.3863
- Best Validation Accuracy : **0.8735**

```
In [67]: # loss, accuracy 시각화
history_df = pd.DataFrame(history.history)
history_df.loc[1:, ['loss', 'val_loss']].plot()
history_df.loc[1:, ['acc', 'val_acc']].plot()

print(("Best Validation Loss: {:.4f}" +\
      "\nBest Validation Accuracy: {:.4f}") +\
      .format(history_df['val_loss'].min(), history_df['val_acc'].max()))

Best Validation Loss: 0.3863
Best Validation Accuracy: 0.8735
```



# 분류 예측 - 모델 학습

## 03. SOM + DNN, DT, LR 모델 학습

### (3) Decision Tree(DT)

- ① DecisionTreeClassifier 사용하여 DT 진행
- ② 그리드 서치(GridSearchCV 사용)를 통한 하이퍼파라미터 튜닝 진행
- ③ 최적 하이퍼파라미터 → criterion = 'gini', max\_depth = 20
- ④ 검증 데이터의 accuracy 확인 : 약 **0.8100**의 정확도

```
In [68]: # decision tree 클래스 지정
from sklearn.tree import DecisionTreeClassifier
dt = DecisionTreeClassifier(random_state=42)

In [69]: # 하이퍼파라미터 탐색
dt.get_params().keys()

Out[69]: dict_keys(['ccp_alpha', 'class_weight', 'criterion', 'max_depth', 'max_features', 'max_leaf_nodes', 'min_impurity_decrease', 'min_samples_leaf', 'min_samples_split', 'min_weight_fraction_leaf', 'random_state', 'splitter'])

In [70]: # 그리드 서치 -> 하이퍼파라미터 튜닝 진행
dt = DecisionTreeClassifier(random_state=42)
para2 = {'criterion':['gini','entropy'], 'max_depth':[5, 10, 15, 20]}

dt = GridSearchCV(estimator = dt, param_grid = para2, cv=5, scoring='accuracy')
dt.fit(over_X_sd, over_y)

Out[70]: GridSearchCV(cv=5, estimator=DecisionTreeClassifier(random_state=42),
param_grid={'criterion': ['gini', 'entropy'],
'max_depth': [5, 10, 15, 20]},
scoring='accuracy')

In [71]: # 최적 하이퍼파라미터 확인
dt.best_params_

Out[71]: {'criterion': 'gini', 'max_depth': 20}

In [72]: # 검증 데이터 accuracy 확인
dt = DecisionTreeClassifier(criterion='gini', max_depth=20, random_state=42)
dt.fit(over_X_sd, over_y)
pred = dt.predict(val_X_sd)
print(accuracy_score(val_y, pred))

0.8100277147896195
```

# 분류 예측 - 모델 학습

## 03. SOM + DNN, DT, LR 모델 학습

### (4) Logistic Regression(LR)

- ① LogisticRegression 사용하여 LR 진행
- ② 그리드 서치(GridSearchCV 사용)를 통한 하이퍼파라미터 튜닝 진행
- ③ 최적 하이퍼파라미터 → C = 1, penalty = 'l2'
- ④ 검증 데이터의 accuracy 확인 : 약 **0.8786**의 정확도

```
In [73]: # logistic regression 클래스 지정
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression(random_state=42)

In [74]: # 하이퍼파라미터 탐색
lr.get_params().keys()
Out[74]: dict_keys(['C', 'class_weight', 'dual', 'fit_intercept', 'intercept_scaling', 'l1_ratio', 'max_iter', 'multi_class',
       'n_jobs', 'penalty', 'random_state', 'solver', 'tol', 'verbose', 'warm_start'])

In [75]: # 그리드 서치 -> 하이퍼파라미터 튜닝 진행
lr = LogisticRegression(random_state=42)
para3 = {'C': [0.001, 0.01, 0.1, 1, 10, 100], 'penalty': ['l1', 'l2']}
lr = GridSearchCV(estimator = lr, param_grid = para3, cv=5, scoring='accuracy')
lr.fit(over_X_sd, over_y)

Out[75]: GridSearchCV(cv=5, estimator=LogisticRegression(random_state=42),
           param_grid={'C': [0.001, 0.01, 0.1, 1, 10, 100],
                       'penalty': ['l1', 'l2']},
           scoring='accuracy')

In [76]: # 최적 하이퍼파라미터 확인
lr.best_params_
Out[76]: {'C': 1, 'penalty': 'l2'}

In [77]: # 검증 데이터 accuracy 확인
lr = LogisticRegression(C=1, penalty='l2', random_state=42)
lr.fit(over_X_sd, over_y)
pred = lr.predict(val_X_sd)
print(accuracy_score(val_y, pred))
0.8785588309397834
```

# 분류 예측 - 모델 학습

## 03. SOM + DNN, DT, LR 모델 학습

### (5) 모델별 성능 비교

- DNN(MLPClassifier) : 0.8437893675988914
- DNN(직접 구현) : 0.8735
- Decision Tree : 0.8100277147896195
- Logistic Regression : 0.8785588309397834

❖ SOM 이후의 DNN, Decision Tree, Logistic Regression 중에서는 '**SOM + LR**'이 약 **0.8785**으로 성능이 가장 우수

# 분류 예측 - 모델 학습

## 04. 모델별 성능 비교

### (1) 모델별 성능 비교

	DNN (MLPClassifier)	DNN(직접 구현)	Decision Tree	Logistic Regression
DNN, DT, LR	0.8360	0.8763	0.8133	0.8778
Clustering Analysis + DNN, DT, LR	0.8445	0.8758	0.8110	0.8778
SOM + DNN, DT, LR	0.8438	0.8735	0.8100	<b>0.8785</b>

전체 12가지 모델 중에서는 '**SOM + LR**'이 약 **0.8785**으로 성능이 가장 우수

## 분류 예측 - 최종 예측

## 01. 최종 예측

## (1) SOM

- ① MiniSom 사용하여 SOM 진행
  - ② 원하는 파라미터 조합하여 리스트화함
  - ③ 모든 조합에 대해 모델 생성 및 랜덤으로 초기값을 설정하는 경우, pca로 초기값을 설정하는 경우의  $qe$ ,  $te$ 값 계산
  - ④ 결과 데이터프레임 생성, sorting, 시각화 진행

```

In [34]: from minisom import MiniSom
s_time = pd.Timestamp.now()
print('시작시간:', s_time, '\n')

# 원하는 파라미터 조합 리스트화
map_n = [n for n in range(2,6)]
para_sigma = [np.round(sigma*0.1,2) for sigma in range(1,10)]
para_learning_rate = [np.round(learning_rate*0.1,2) for learning_rate in range(1,10)]

# 결과 값을 담을 리스트 res 생성
res = []
# 모든 조건에 대해서 모델 생성 및 qe, te값 계산
for n in map_n:
    for sigma in para_sigma:
        for lr in para_learning_rate:
            try:
                # 랜덤으로 초기값을 설정하는 경우
                estimator = MiniSom(n,n,4,sigma=sigma, learning_rate = lr, topology='hexagonal', random_seed=42)
                estimator.random_weights_init(scaled_df.values)
                estimator.train(scaled_df.values, 1000, random_order=True)
                qe = estimator.quantization_error(scaled_df.values)
                # te = estimator.topographic_error(scaled_df.values)
                winner_coordinates = np.array([estimator.winner(x) for x in scaled_df.values]).T
                cluster_index = np.ravel_multi_index(winner_coordinates, (n,n))

                res.append([str(n)+'*'+str(sigma)+'*'+str(lr), 'random_init', qe, len(np.unique(cluster_index))])

                # pca로 초기값을 설정하는 경우
                estimator = MiniSom(n,n,4,sigma=sigma, learning_rate = lr,topology='hexagonal', random_seed=42)
                estimator.pca_weights_init(scaled_df.values)
                estimator.train(scaled_df.values, 1000, random_order=True)
                qe = estimator.quantization_error(scaled_df.values)
                # te = estimator.topographic_error(scaled_df.values)
                winner_coordinates = np.array([estimator.winner(x) for x in scaled_df.values]).T
                cluster_index = np.ravel_multi_index(winner_coordinates, (n,n))

                res.append([str(n)+'*'+str(sigma)+'*'+str(lr), 'pca_init', qe, len(np.unique(cluster_index))])
            except:
                pass

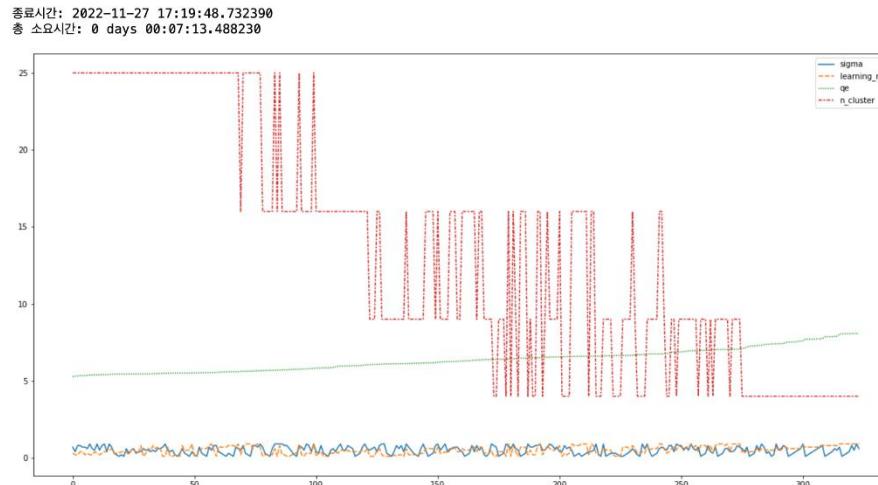
```

# 분류 예측 - 최종 예측

## 01. 최종 예측

### (1) SOM

- ① MiniSom 사용하여 SOM 진행
- ② 원하는 파라미터 조합하여 리스트화함
- ③ 모든 조합에 대해 모델 생성 및 랜덤으로 초기값을 설정하는 경우, pca로 초기값을 설정하는 경우의 qe, te값 계산
- ④ 결과 데이터프레임 생성, sorting, 시각화 진행



# 분류 예측 - 최종 예측

## 01. 최종 예측

### (1) SOM

- SOM 결과 데이터프레임 정보는 다음과 같음

In [35]: df\_res

Out[35]:

	map_size	sigma	learning_rate	init_method	qe	n_cluster
0	5x5	0.7	0.3	random_init	5.270559	25
1	5x5	0.4	0.2	random_init	5.305664	25
2	5x5	0.8	0.2	random_init	5.315885	25
3	5x5	0.8	0.4	random_init	5.337061	25
4	5x5	0.7	0.4	random_init	5.337695	25
...	...	...	...	...	...	...
319	2x2	0.4	0.9	random_init	8.056978	4
320	2x2	0.8	0.9	random_init	8.066916	4
321	2x2	0.5	0.9	random_init	8.073240	4
322	2x2	0.9	0.9	random_init	8.075346	4
323	2x2	0.6	0.9	random_init	8.078343	4

324 rows × 6 columns

In [38]: # cluster 총 개수별 군집화 종류 개수  
df\_res['n\_cluster'].value\_counts()

Out[38]:

25	81
16	81
9	81
4	81

Name: n\_cluster, dtype: int64

In [39]: df\_res[df\_res['n\_cluster']==4]

Out[39]:

	map_size	sigma	learning_rate	init_method	qe	n_cluster
173	2x2	0.8	0.1	random_init	6.391071	4
174	2x2	0.7	0.1	random_init	6.395825	4
178	2x2	0.9	0.1	random_init	6.413493	4
180	2x2	0.6	0.1	random_init	6.417253	4
183	2x2	0.5	0.1	random_init	6.472109	4
...	...	...	...	...	...	...
319	2x2	0.4	0.9	random_init	8.056978	4
320	2x2	0.8	0.9	random_init	8.066916	4
321	2x2	0.5	0.9	random_init	8.073240	4
322	2x2	0.9	0.9	random_init	8.075346	4
323	2x2	0.6	0.9	random_init	8.078343	4

81 rows × 6 columns

# 분류 예측 - 최종 예측

## 01. 최종 예측

### (1) SOM - 모델 파라미터 조정

- ① qe 값이 최소인 하이퍼파라미터 기준으로 SOM 진행
- ② 모델의 초기값 설정 및 모델 평가
- ③ 4개의 cluster에 배정된 cluster별 data 개수 count

```
In [40]: # qe값이 최소인 하이퍼파라미터 기준 SOM 진행
som_b2 = MiniSom(2,2,56,sigma=0.8,learning_rate=0.1,topology='hexagonal',neighborhood_function='gaussian',activation
```

```
# 초기값설정
som_b2.pca_weights_init(scaled_df.values)
som_b2.train(scaled_df.values,1000,random_order=True)
```

```
# 평가
som_b2.quantization_error(scaled_df.values)
#som_b2.topographic_error(data.values)
```

```
Out[40]: 6.42072234413208
```

```
In [41]: winner_coordinates = np.array([som_b2.winner(x) for x in scaled_df.values]).T
cluster_index = np.ravel_multi_index(winner_coordinates,(n,n))
```

```
In [42]: # 4개의 cluster에 배정된 cluster별 data 개수
pd.DataFrame(cluster_index).value_counts()
```

```
Out[42]: 5    9645
0    7335
6    5038
1    4439
dtype: int64
```

# 분류 예측 - 최종 예측

## 01. 최종 예측

### (1) SOM - 시각화

- ① RegularPolygon, cm, make\_axes\_locatable, ColorbarBase를 import하여 시각화 진행

```
In [43]: from matplotlib.patches import RegularPolygon
from matplotlib import cm
from mpl_toolkits.axes_grid1 import make_axes_locatable
from matplotlib.colorbar import ColorbarBase
```

```
In [44]: xx, yy = som_b2.get_euclidean_coordinates()
umatrix = som_b2.distance_map()
weights = som_b2.get_weights()

f = plt.figure(figsize=(10,10))
ax = f.add_subplot(111)
ax.set_aspect('equal')

# iteratively add hexagons
# plotting the distance map as background
# 해당 결과 다른 이웃들 간 거리를 표현, 밝을수록 가깝고, 어두울수록 멀다
for i in range(weights.shape[0]):
    for j in range(weights.shape[1]):
        wy = yy[[i, j]] * 2 / np.sqrt(3) * 3 / 4
        hex = RegularPolygon((xx[[i, j]], wy),
                             numVertices=6,
                             radius=.95 / np.sqrt(3),
                             facecolor=cm.Blues(umatrix[i, j]),
                             alpha=.4,
                             edgecolor='gray')
        plot = ax.add_patch(hex)

#output노드에 해당하는 클러스터 종류 및 밀도 확인
cnt=[]
for c in np.unique(cluster_index):
    x_= [som_b2.convert_map_to_euclidean(som_b2.winner(x))[0] + (2*np.random.rand(1)[0]-1)*0.4 for x in scaled_d]
    y_= [som_b2.convert_map_to_euclidean(som_b2.winner(x))[1] + (2*np.random.rand(1)[0]-1)*0.4 for x in scaled_d]
    y_= [(i* 2 / np.sqrt(3) * 3 / 4) for i in y_]

    plot = sns.scatterplot( x = x_, y = y_ ,label='cluster='+str(c),alpha=.7)
```

#클러스터에 속한 데이터 개수 데이터프레임으로 출력  
cnt.append([c,len(x\_)])

```
#클러스터별 개수를 표 형태로 출력
df_cnt = pd.DataFrame(cnt,columns=['cluster이름', '개수'])

#x축,y축 간격 설정
xrange = np.arange(weights.shape[0])
yrange = np.arange(weights.shape[1])
plot = plt.xticks(xrange-.5, xrange)
plot = plt.yticks(yrange * 2 / np.sqrt(3) * 3 / 4, yrange)

#차트 우측에 color bar생성
divider = make_axes_locatable(plt.gca())
ax_cb = divider.new_horizontal(size="5%", pad=0.05)
cb1 = ColorbarBase(ax_cb, cmap=cm.Blues,
                    orientation='vertical', alpha=.4)
cb1.ax.get_yaxis().labelpad = 16
plot = cb1.ax.set_ylabel('distance from neurons in the neighborhood',
                        rotation=270, fontsize=16)
plot = plt.gcf().add_axes(ax_cb)

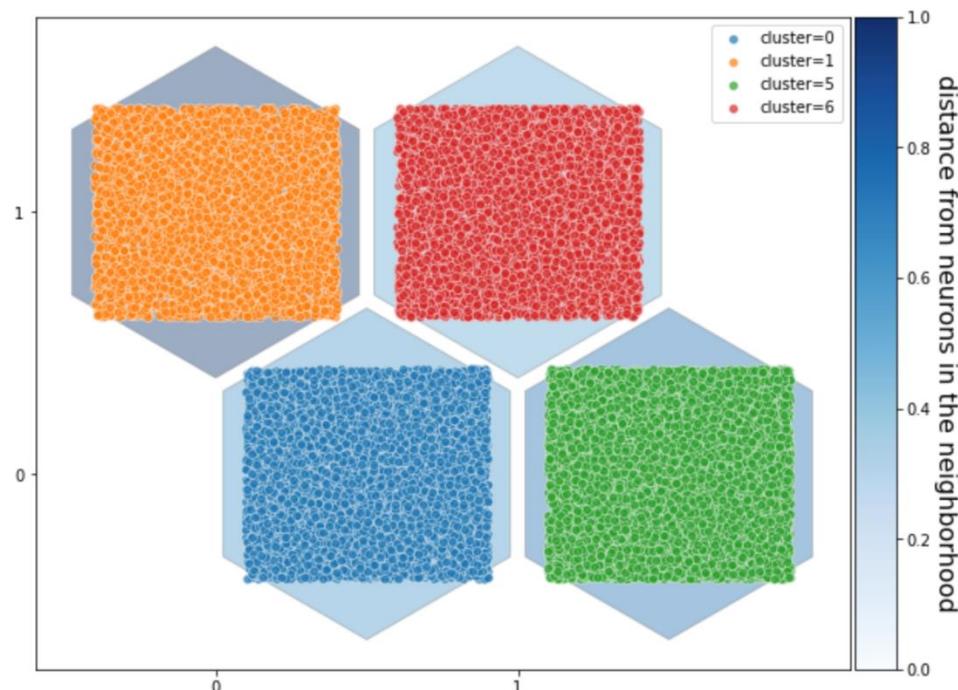
#이미지 저장
plt.savefig('som_seed_hex.png')
```

# 분류 예측 - 최종 예측

## 01. 최종 예측

### (1) SOM - 시각화

- 시각화의 결과는 다음과 같음



# 분류 예측 - 최종 예측

## 01. 최종 예측

### (1) SOM - 시각화 후 데이터 처리

- ① 클러스터링 변수인 clusters 값을 원본 데이터인 'scaled\_df'내에 넣음
- ② cluster를 기준으로 데이터 개수를 셈 결과, 0번 cluster에는 7335개, 1번 cluster에는 4439개, 5번 cluster에는 9645개, 6번 cluster에는 5038개의 데이터가 있는 것을 확인할 수 있음
- ③ 그룹별 평균값을 통해 그룹별 특징을 확인해봄

```
In [45]: #클러스터링 변수인 clusters 값을 원본 데이터인 'scaled_df' 내에 넣기
scaled_df['cluster'] = cluster_index
scaled_df.head()
```

```
Out[45]:
child_num income_total DAYS_BIRTH DAYS_EMPLOYED work_phone phone email family_size begin_month gender_F gender_M car_N
0 -0.573599 0.149136 -0.731391 0.927612 -0.538417 -0.645705 -0.316937 -0.214735 -1.623064 0.703562 -0.703562 -1.278015 1
1 0.764529 -0.292575 0.044045 0.797238 -0.538417 1.548696 -0.316937 0.876135 0.007446 0.703562 -0.703562 0.782463 -0
2 -0.573599 -0.513431 0.589087 0.161398 1.857295 1.548696 -0.316937 -0.214735 1.034063 -1.421339 1.421339 -1.278015 1
3 2.102658 -0.734287 0.877793 0.864324 -0.538417 1.548696 -0.316937 1.967005 0.852895 0.703562 -0.703562 -1.278015 1
4 0.764529 0.369992 -0.051635 -0.072769 -0.538417 -0.645705 -0.316937 0.876135 1.396399 -1.421339 1.421339 -1.278015 1
```

```
In [46]: # cluster를 기준으로 데이터 개수 세기
scaled_df.groupby('cluster').count()
```

```
Out[46]:
child_num income_total DAYS_BIRTH DAYS_EMPLOYED work_phone phone email family_size begin_month gender_F gender_M car_N car_Y
cluster
0 7335 7335 7335 7335 7335 7335 7335 7335 7335 7335 7335 7335 7335
1 4439 4439 4439 4439 4439 4439 4439 4439 4439 4439 4439 4439 4439
5 9645 9645 9645 9645 9645 9645 9645 9645 9645 9645 9645 9645 9645
6 5038 5038 5038 5038 5038 5038 5038 5038 5038 5038 5038 5038 5038
```

```
In [47]: #그룹별 특징을 알아보기(그룹별 평균값)
scaled_df.groupby('cluster').mean()
```

```
Out[47]:
child_num income_total DAYS_BIRTH DAYS_EMPLOYED work_phone phone email family_size begin_month gender_F gender_M car_
cluster
0 0.029516 0.420785 0.396421 -0.022516 0.168048 -0.020742 0.053708 0.103082 -0.009366 -1.325161 1.325161 -0.71706
1 -0.510295 -0.370167 -1.383632 0.927529 -0.538417 -0.016414 -0.204902 -0.498327 0.015064 0.390020 -0.390020 0.35727
5 0.379947 -0.110744 0.314321 -0.215606 0.129501 0.058460 0.060695 0.537507 -0.008120 0.560800 0.560800 0.12041
6 -0.320741 -0.074467 0.040209 -0.371702 -0.018189 -0.068139 -0.014383 -0.740033 0.015908 0.512076 -0.512076 0.49862
```

# 분류 예측 - 최종 예측

## 01. 최종 예측

### (1) SOM - 시각화 후 PCA 진행

- ① PCA 라이브러리 활용하여 PCA 진행
- ② PCA가 완료된 이후의 2개의 cluster 시각화
- ③ 데이터에 SOM 분석 결과 column을 추가함으로써 군집 분석 결과 반영
- ④ 다시 원래대로 train(credit 값 존재)과 test(credit 값 NaN) 분리

```
In [48]: # PCA 진행
from sklearn.decomposition import PCA
X = scaled_df.copy()

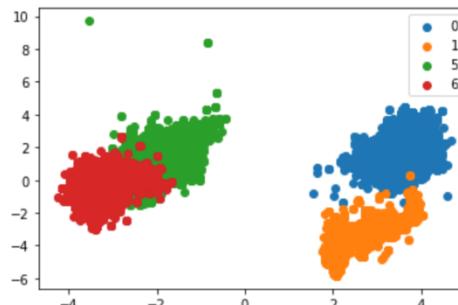
pca = PCA(n_components=2)
pca.fit(X)
x_pca = pca.transform(X)
x_pca
```

```
Out[48]: array([[ 2.77498254, -2.89177319],
 [-2.38412371,  0.75560108],
 [ 3.83805293,  2.03475776],
 ...,
 [ 3.53365153,  0.73092056],
 [ 3.19694863,  1.51047434],
 [ 3.33853694,  0.52548338]])
```

```
In [49]: # x_pca를 보기 쉽게 데이터프레임으로 변환
pca_df = pd.DataFrame(x_pca)
pca_df['cluster'] = scaled_df['cluster']
pca_df.head()
```

```
Out[49]:   0      1  cluster
0  2.774983 -2.891773      1
1 -2.384124  0.755601      5
2  3.838053  2.034758      0
3 -1.568297  1.948583      5
4  4.234423  1.858138      0
```

```
In [50]: # pca가 완료된 이후의 2개의 cluster 시각화
for i in sorted(pca_df['cluster'].unique()):
    tmp = pca_df.loc[pca_df['cluster'] == i] # 해당하는 클러스터 번호일 때 그림을 그리고, for문 실행하며 위에 덧그림
    plt.scatter(tmp[0], tmp[1])
plt.legend(sorted(pca_df['cluster'].unique()))
```



# 분류 예측 - 최종 예측

## 01. 최종 예측

### (1) SOM - 시각화 후 PCA 진행

- ① PCA 라이브러리 활용하여 PCA 진행
- ② PCA가 완료된 이후의 2개의 cluster 시각화
- ③ 데이터에 SOM 분석 결과 column을 추가함으로써 군집 분석 결과 반영
- ④ 다시 원래대로 train(credit 값 존재)과 test(credit 값 NaN) 분리

In [51]: scaled\_df

Out[51]:

	child_num	income_total	DAYS.BIRTH	DAYS_EMPLOYED	work_phone	phone	email	family_size	begin_month	gender_F	gender_M	car_N
0	-0.573599	0.149136	-0.731391	0.927612	-0.538417	-0.645705	-0.316937	-0.214735	-1.623064	0.703562	-0.703562	-1.27801
1	0.764529	-0.292575	0.044045	0.979238	-0.538417	1.548696	-0.316937	0.876135	0.007446	0.703562	-0.703562	0.78246
2	-0.573599	-0.513431	0.589087	0.161398	1.857295	1.548696	-0.316937	-0.214735	1.034063	-1.421339	1.421339	-1.27801
3	2.102658	-0.734287	0.877793	0.864324	-0.538417	1.548696	-0.316937	1.967005	0.852895	0.703562	-0.703562	-1.27801
4	0.764529	0.369992	-0.051635	-0.072769	-0.538417	-0.645705	-0.316937	0.876135	1.396399	-1.421339	1.421339	-1.27801
...	...	...	...	...	...	...	...	...	...	...	...	...
26452	-0.573599	0.149136	0.859466	0.059715	-0.538417	-0.645705	-0.316937	-0.214735	-0.234111	0.703562	-0.703562	0.78246
26453	-0.573599	-0.380918	1.565165	0.063090	-0.538417	1.548696	3.155199	-1.305605	0.128224	-1.421339	1.421339	0.78246
26454	-0.573599	0.811704	0.252779	-1.359215	-0.538417	-0.645705	3.155199	-0.214735	0.007446	-1.421339	1.421339	0.78246
26455	-0.573599	2.136838	0.018340	-1.089606	1.857295	-0.645705	-0.316937	-0.214735	-0.234111	-1.421339	1.421339	0.78246
26456	-0.573599	1.032559	1.326821	0.521721	-0.538417	-0.645705	-0.316937	-0.214735	-1.623064	-1.421339	1.421339	0.78246

26457 rows × 57 columns

In [52]: # data에 'cluster' column 추가  
data['cluster'] = scaled\_df['cluster']  
data

Out[52]:

	child_num	income_total	DAYS.BIRTH	DAYS_EMPLOYED	work_phone	phone	email	family_size	begin_month	credit	gender_F	gender_M	car_N	c
0	0	202500.0	-19031	0	0	0	0	2	-53	1.0	1	0	0	
1	1	157500.0	-15773	-309	0	1	0	3	-26	0.0	1	0	1	
2	0	135000.0	-13483	-1816	1	1	0	2	-9	1.0	0	1	0	
3	2	112500.0	-12270	-150	0	1	0	4	-12	1.0	1	0	0	
4	1	225000.0	-16175	-2371	0	0	0	3	-3	1.0	0	1	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
26452	0	202500.0	-12347	-2057	0	0	0	2	-30	NaN	1	0	1	
26453	0	148500.0	-9382	-2049	0	1	1	1	-24	NaN	0	1	1	
26454	0	270000.0	-14896	-5420	0	0	1	2	-26	NaN	0	1	1	
26455	0	405000.0	-15881	-4781	1	0	0	2	-30	NaN	0	1	1	
26456	0	292500.0	-10375	-962	0	0	0	2	-53	NaN	0	1	1	

26457 rows × 58 columns

In [53]: # 다시 원래대로 train(credit 값 존재)과 test(credit 값 NaN) 분리  
train\_data = data.loc[data['credit'].notnull()]  
test\_data = data.loc[data['credit'].isnull()]

# 분류 예측 - 최종 예측

## 01. 최종 예측

### (2) SOM + LR

- ① LogisticRegression 사용하여 LR 진행
- ② 그리드 서치(GridSearchCV 사용)를 통한 하이퍼파라미터 튜닝 진행
- ③ 최적 하이퍼파라미터 → C = 1, penalty = 'l2'
- ④ 검증 데이터의 accuracy 확인 : 약 **0.8786**의 정확도

```
In [73]: # logistic regression 클래스 지정
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression(random_state=42)

In [74]: # 하이퍼파라미터 탐색
lr.get_params().keys()
Out[74]: dict_keys(['C', 'class_weight', 'dual', 'fit_intercept', 'intercept_scaling', 'l1_ratio', 'max_iter', 'multi_class',
       'n_jobs', 'penalty', 'random_state', 'solver', 'tol', 'verbose', 'warm_start'])

In [75]: # 그리드 서치 -> 하이퍼파라미터 튜닝 진행
lr = LogisticRegression(random_state=42)
para3 = {'C': [0.001, 0.01, 0.1, 1, 10, 100], 'penalty': ['l1', 'l2']}
lr = GridSearchCV(estimator = lr, param_grid = para3, cv=5, scoring='accuracy')
lr.fit(over_X_sd, over_y)

Out[75]: GridSearchCV(cv=5, estimator=LogisticRegression(random_state=42),
param_grid={'C': [0.001, 0.01, 0.1, 1, 10, 100],
            'penalty': ['l1', 'l2']},
scoring='accuracy')

In [76]: # 최적 하이퍼파라미터 확인
lr.best_params_
Out[76]: {'C': 1, 'penalty': 'l2'}

In [77]: # 검증 데이터 accuracy 확인
lr = LogisticRegression(C=1, penalty='l2', random_state=42)
lr.fit(over_X_sd, over_y)
pred = lr.predict(val_X_sd)
print(accuracy_score(val_y, pred))
0.8785588309397834
```

# 분류 예측 - 최종 예측

## 01. 최종 예측

### (2) SOM + LR

- 가장 성능이 좋았던 SOM+LR 모델로 최종 예측 진행 후 결과 처리(03.SOM-(4)LR 모델 채택)

① test 데이터셋에서 credit column 분리

② 인덱스 재지정

```
In [63]: test_y = test_data['credit']
test_X = test_data.drop(['credit'], axis=1)

In [64]: # index 재지정
test_X = test_X.reset_index(drop=True)
test_X
```

Out[64]:

	child_num	income_total	DAY_S_BIRTH	DAY_S_EMPLOYED	work_phone	phone	email	family_size	begin_month	gender_F	gender_M	car_N	car_
0	0	211500.0	-10072	-1101	1	1	0	1	-10	0	1	1	1
1	0	157500.0	-24340	0	0	1	0	1	-52	1	0	1	1
2	0	45000.0	-15724	-1389	1	1	0	2	-15	1	0	1	1
3	2	270000.0	-11505	-4019	0	0	0	3	-24	0	1	1	1
4	0	202500.0	-15929	-2879	0	1	0	2	-54	1	0	1	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...
13224	0	202500.0	-12347	-2057	0	0	0	2	-30	1	0	1	1
13225	0	148500.0	-9382	-2049	0	1	1	1	-24	0	1	1	1
13226	0	270000.0	-14896	-5420	0	0	1	2	-26	0	1	1	1
13227	0	405000.0	-15881	-4781	1	0	0	2	-30	0	1	1	1
13228	0	292500.0	-10375	-962	0	0	0	2	-53	0	1	1	1

13229 rows × 57 columns

# 분류 예측 - 최종 예측

## 01. 최종 예측

### (2) SOM + LR

- ③ test\_X 데이터를 StandardScaling 정규화
- ④ test\_label.csv 확인

```
In [65]: test_y = pd.DataFrame(test_y).reset_index(drop=True)  
test_y
```

```
Out[65]:
```

	credit
0	NaN
1	NaN
2	NaN
3	NaN
4	NaN
...	...
13224	NaN
13225	NaN
13226	NaN
13227	NaN
13228	NaN

13229 rows × 1 columns

```
In [66]: test_X_sd = sd.transform(test_X)
```

```
In [67]: test_label = pd.read_csv('data/test/test_label.csv')  
test_label
```

```
Out[67]:
```

index	credit
0	0
1	1
2	2
3	3
4	4
...	...
13224	13224
13225	13225
13226	13226
13227	13227
13228	13228

13229 rows × 2 columns

# 분류 예측 - 최종 예측

## 01. 최종 예측

### (2) SOM + LR

- ⑤ LR을 이용한 model predict, 예측값 확인
- ⑥ test\_label의 credit column에 예측값 기입, test\_label.csv로 데이터 내보내기

```
In [68]: # model.predict()
pred_label = lr.predict(test_X_sd)
pred_label = pd.DataFrame(pred_label)
pred_label
```

```
Out[68]:
0
0 1.0
1 1.0
2 1.0
3 1.0
4 1.0
...
13224 1.0
13225 1.0
13226 1.0
13227 1.0
13228 1.0
```

```
In [69]: # test_label의 credit 값 채우기
test_label['credit'] = pred_label
test_label
```

```
Out[69]:
index credit
0 0 1.0
1 1 1.0
2 2 1.0
3 3 1.0
4 4 1.0
...
13224 13224 1.0
13225 13225 1.0
13226 13226 1.0
13227 13227 1.0
13228 13228 1.0
```

```
In [70]: # dtype float -> int로 변경
test_label['credit'] = test_label['credit'].apply(np.int64)
test_label
```

```
Out[70]:
index credit
0 0 1
1 1 1
2 2 1
3 3 1
4 4 1
...
13224 13224 1
13225 13225 1
13226 13226 1
13227 13227 1
13228 13228 1
```

13229 rows × 2 columns

13229 rows × 1 columns

13229 rows × 2 columns

```
In [71]: # test_label.csv로 내보내기
test_label.to_csv('data/test/test_label.csv', index = None) #
```

감사합니다