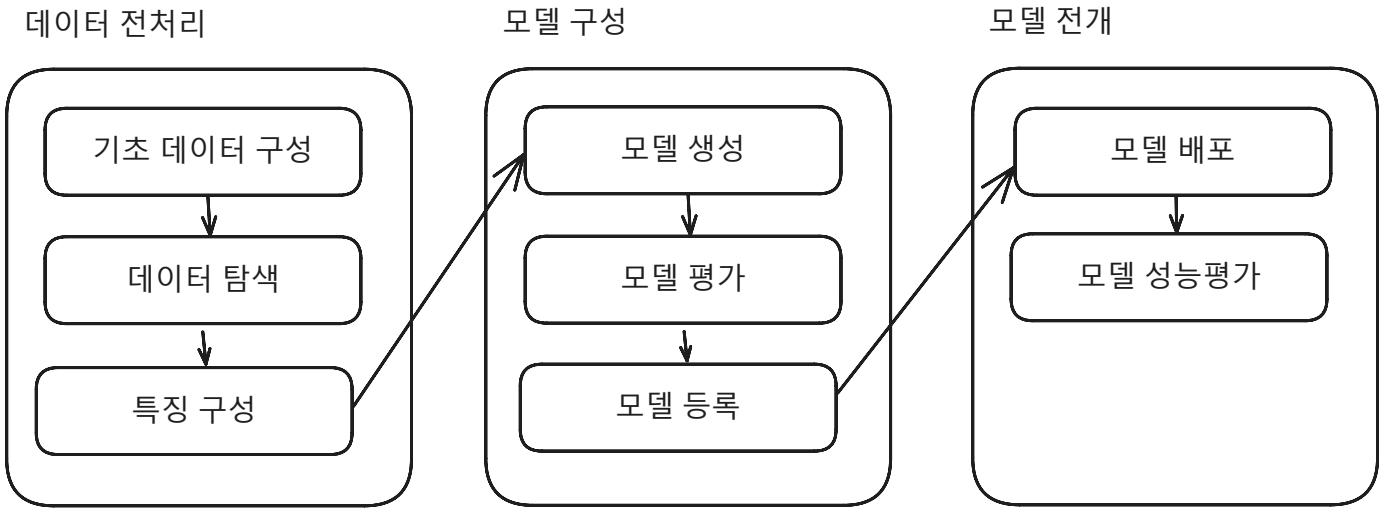
ML 기본 프로세스


데이터 전처리
기초 데이터 구성
데이터 탐색
특징 구성
모델 구성
모델 생성
모델 평가
모델 등록
모델 전개
모델 배포
모델 성능평가

```python
import importlib.metadata

# Python 3.8 이상에서는 importlib.metadata.distributions()를 이용하여 설치된 패키지 정보를 얻을 수 있습니다.
installed_packages = importlib.metadata.distributions()

# 각 패키지의 이름과 버전을 문자열 리스트로 생성합니다.
package_list = sorted([f"{dist.metadata['Name']}=={dist.version}" for dist in installed_packages])

# 목록 출력
for package in package_list:
    print(package)
```

1. 데이터 레이아웃

컬럼 이름	컬럼 내용	컬럼 내용(영문)
BAD	대출 상환 실패 여부를 나타내는 타겟 변수(1이면 디폴트대출자의, 0이면 상환 완료)	The target variable indicating whether the borrower defaulted on the loan (1 for default, 0 for non-default).
LOAN	대출 요청 금액	The amount of the loan request.
MORTDUE	현재 주택 담보 대출의 잔액	The outstanding balance of the mortgage.
VALUE	현재 자산의 가치	The current value of the property.
REASON	대출을 요청한 이유(DebtCon 부채 조정, HomeImp 주택 개선 등)	The reason for the loan request (DebtCon for debt consolidation, HomeImp for home improvement, etc.).
JOB	대출자의 직업	The borrower's job category.
YOJ	현재 직장에서의 근속 연수	Years at the current job.
DEROG	주요 신용 문제 보고의 수	Number of major derogatory reports.
DELINQ	연체된 신용 계좌의 수	Number of delinquent credit lines.
CLAGE	최장 계좌 연령(월)	Age of oldest credit line in months.
NINQ	최근의 신용 문의 수	Number of recent credit inquiries.
CLNO	신용 계좌의 수	Number of credit lines.
DEBTINC	부채 소득 비율	Debt-to-income ratio.

2. 데이터 불러오기

- 데이터 저장소 만들기

```
import os

# 현재 작업 디렉토리 경로 가져오기
current_dir = os.getcwd()
# data 디렉토리 경로 생성
data_dir = os.path.join(current_dir, 'data')

# 디렉토리가 존재하지 않으면 생성
if not os.path.exists(data_dir):
    os.mkdir(data_dir)
    print(f"{data_dir} 디렉토리가 생성되었습니다.")
else:
    print(f"{data_dir} 디렉토리가 이미 존재합니다.")
```

- 데이터 가져오기

```
# 필요한 패키지
import pandas as pd
import numpy as np
import os

import joblib



#
# 데이터 읽어오기
#

# 데이터 경로
```

```
data_url = "https://github.com/bong-ju-kang/data/raw/master/hmeq.csv"



# 데이터 읽어오기
df_raw = pd.read_csv(data_url)

# 데이터 저장
joblib.dump(df_raw, "data/hmeq.pkl")

# 데이터 불러오기
df_raw = joblib.load( "data/hmeq.pkl")
```

변수 정의

```
# 반응변수
target = 'BAD'

# 특징변수
features = df_raw.columns.drop(target)

# 범주형/숫자형 변수 정의
cats = df_raw[features].select_dtypes(include=['object']).columns
nums = df_raw[features].select_dtypes(include=['number']).columns
```

#실습

- 층화 추출을 해보자. 단, 평가 데이터는 20%, 나머지는 훈련 데이터로 한다. 씨앗값은 42으로 정한다.
- 차후 이용을 위하여 df_train.pkl, df_test.pkl 데이터로 저장한다. joblib.dump, joblib.load 를 이용한다.

1. 탐색 기준표

정책	적용 변수 유형	기본값	비고
missing	all	5, 25	결측값 비율(%)
cardinality	nominal	lowMediumCutoff=32, mediumHighCutoff=64	작은 경계값(low-medium), 큰 경계값(medium-high)
entropy	nominal	0.25, 0.75	표준화 엔트로피 기준으로 정의. 표준화 엔트로피는 0과 1사이의 값이며, 값이 크다는 것은 랜덤 값에 가까움을 의미함
IQV(index of qualitative variation)	nominal	highVariationRatio=0.5, highTopBottom=100, highTopTwo=10	변동비, 빈도 최상/최하 비율, 빈도 상위/차상위 비율. 디폴트는 변동비로 판정
cv	interval	lowMoment=1, lowRobust=1	기본값은 경계값(low-high)의 백분위수를 의미함. 해당 값보다 크면 모두 3등급으로 처리 |CV|*100의 경계값. 값이 클수록 평균대비 편차가 크다는 것을 의미
skewness	Interval	적률(2, 10), 분위수(0.75, 2)	적률기준, 분위수 기준 작은/큰 경계값. 정규분포는 0에 가까움. 양쪽 모두 평가 후 높은 등급으로 판정
kurtosis	interval	적률(5, 10), 분위수(2, 3)	적률기준, 분위수 기준 작은/높은 경계값. 정규분포는 0에 가까움. . 양쪽 모두 평가 후 높은 등급으로 판정
outlier	interval	z(1, 2.5), IQR(1, 2.5)	z, IQR기준 이상값 백분위수. 양쪽 모두 평가 후 높은 등급으로 판정

2. 결측값

#실습 결측 비율을 계산하는 프로그램을 작성해보자.

3. 카디널리티(cardinality, 유일 건수)

#실습 유일건수를 계산하는 프로그램을 작성해보자.

4. 엔트로피(entropy)

- 정규화 샤논 엔트로피(Shannon entropy)
  - $\frac{1}{\log_2 c}\left(-\sum_j p_j \log_2 p_j\right)$, c : 범주의 개수
- 정규화 지니 불순도(Gini impurity)
  - $\frac{c}{c-1}\left(1-\sum_j p_j^2\right)$

#실습 샤논 엔트로피를 계산하는 프로그램을 작성해보자.

5. 정성 변동 지수(IQV, index of qualitative variation)

- 변동비(varation ratio)
  - $1 - \frac{f_{mode}}{n}$
  - f_{mode} : 최대 빈도
  - n : 데이터 건수(결측값 제외)
- 빈도 최상/최하 비율
- 빈도 상위/차상위 비율

#실습 변동비를 계산하는 프로그램을 작성해보자.
#실습 상위 2개간의 비율을 계산해보자.

6. 변동 계수(CV: coefficient of variation)

- 변동 계수
  - $CV = \frac{s}{\bar{x}}$
  - s : 표본 표준편차
  - $\bar{x}$:
- 로버스트 변동 계수
  - $CV_{IQR} = \frac{IQR}{m}$
  - $IQR = Q_3 - Q_1$: 사분위수 범위
  - m : 중간값
- MAD 기반 로버스트 변동 계수
  - $CV_{MAD} = 1.4826 \times \frac{MAD}{m}$
  - MAD

```
MAD = (np.abs(df[nums] - df[nums].median())).median()
```

#실습 변동계수를 계산하는 프로그램을 작성해보자.
#실습 로버스트 변동계수를 계산하는 프로그램을 작성해보자.

7. 비대칭도(skewness)

- 불편 비대칭도(unbiased skewness)
  - $Skew = \frac{n}{(n-1)(n-2)} \sum_i (\frac{x_i - \bar{x}}{s})^3$
- 로버스트 비대칭도(Bowly-Galton skewness)
  - $Skew_{BG} = \frac{(Q_3 - Q_2) - (Q_2 - Q_1)}{Q_3 - Q_1}$
- 로버스트 비대칭도(Hogg skewness)
  - $Skew_H = \frac{U_5 - M_{25}}{M_{25} - L_5}$
  - U_5 : 상위 5%의 데이터 평균
  - M_{25} : 25% 절사 평균

```
M_25 = df[nums].apply(lambda x: np.mean(x[(x > x.quantile(0.25)) & (x < x.quantile(0.75))]), axis=0)
U_5 = df[nums].apply(lambda x: np.mean(x[x > x.quantile(0.95)]), axis=0)
L_5 = df[nums].apply(lambda x: np.mean(x[x < x.quantile(0.05)]), axis=0)
```

#실습 비대칭도를 계산하는 프로그램을 작성해보자.
#실습 로버스트 비대칭도를 계산하는 프로그램을 작성해보자.

8. 첨도(kurtosis)

- 불편 첨도(unbiased kurtosis)
  - $Kurt = \frac{n(n+1)}{(n-1)(n-2)(n-3)} \sum_i (\frac{x_i - \bar{x}}{s})^4 - \frac{3(n-1)^2}{(n-2)(n-3)}$
- 로버스트 첨도(Hogg kurtosis)
  - $Kurt_H = \frac{U_{20} - L_{20}}{U_{50} - L_{50}}$
  - U_{50} : 상위 50%의 데이터 평균
  - U_{20} : 상위 20%의 데이터 평균
  - L_{50} : 하위 50%의 데이터 평균
  - L_{20} : 하위 20%의 데이터 평균

```
U_20 = df[nums].apply(lambda x: np.mean(x[x > x.quantile(0.8)]), axis=0)
L_20 = df[nums].apply(lambda x: np.mean(x[x < x.quantile(0.2)]), axis=0)
U_50 = df[nums].apply(lambda x: np.mean(x[x > x.quantile(0.5)]), axis=0)
L_50 = df[nums].apply(lambda x: np.mean(x[x < x.quantile(0.5)]), axis=0)
```

#실습 첨도를 계산하는 프로그램을 작성해보자.
#실습 로버스트 첨도를 계산하는 프로그램을 작성해보자.

9. 이상값(outlier)

- z 점수를 이용하는 방법
  - $|z| > 2.5$ 인 비율 계산
  - $Z = \frac{x_i - \bar{x}}{s}$
- IQR 을 이용하는 방법
  - $x < Q_1 - 1.5 \times IQR$
  - $x > Q_3 + 1.5 \times IQR$
  - 2개의 비율을 합산

#실습 이상값를 계산하는 프로그램을 작성해보자.
#실습 로버스트 이상값을 계산하는 프로그램을 작성해보자.

데이터 전처리

정책	적용 변수 유형	처리 방향	비고
missing	all	5, 25	결측값 비율(%)
cardinality	nominal	lowMediumCutoff=32, mediumHighCutoff=64	작은 경계값(low-medium), 큰 경계값(medium-high)
entropy	nominal	0.25, 0.75	표준화 엔트로피 기준으로 정의. 표준화 엔트로피는 0과 1사이의 값이며, 값이 크다는 것은 랜덤 값에 가까움을 의미함
IQV(index of qualitative variation)	nominal	highVariationRatio=0.5, highTopBottom=100, highTopTwo=10	변동비, 빈도 최상/최하 비율, 빈도 상위/차상위 비율. 디폴트는 변동비로 판정
cv	interval	lowMoment=1, lowRobust=1	기본값은 경계값(low-high)의 백분위수를 의미함. 해당 값보다 크면 모두 3등급으로 처리 |CV|*100의 경계값. 값이 클수록 평균대비 편차가 크다는 것을 의미
skewness	Interval	적률(2, 10), 분위수(0.75, 2)	적률기준, 분위수 기준 작은/큰 경계값. 정규분포는 0에 가까움. 양쪽 모두 평가 후 높은 등급으로 판정
kurtosis	interval	적률(5, 10), 분위수(2, 3)	적률기준, 분위수 기준 작은/높은 경계값. 정규분포는 0에 가까움. . 양쪽 모두 평가 후 높은 등급으로 판정
outlier	interval	z(1, 2.5), IQR(1, 2.5)	z, IQR기준 이상값 백분위수. 양쪽 모두 평가 후 높은 등급으로 판정

1. 결측값 처리

결측값을 처리하는 주요 방법은 삭제(Removal)와 대체(Imputation)입니다. 데이터의 특성과 결측값의 분포에 따라 적절한 방법을 선택해야 합니다.

1.1.결측값 삭제

1.1.1. 행 삭제 (`dropna`)

데이터프레임에서 결측값이 있는 **전체 행**을 삭제합니다.

```
# 결측값이 있는 행 삭제
df_dropped = df.dropna()
print(df_dropped.sample(5))
```

```
>>> print(df_dropped.sample(5))
       LOAN   MORTDUE     VALUE   REASON      JOB   YOJ  DEROG  DELINQ       CLAGE  NINQ  CLNO    DEBTINC  BAD
3330  17700  107395.0  133433.0  DebtCon  ProfExe   6.0    0.0     0.0   96.157562   0.0  13.0  27.592277    0
4037  21100   81389.0  123842.0  DebtCon    Other   9.0    0.0     0.0  285.046916   1.0  17.0  37.139068    0
5778  46300  120636.0  178823.0  DebtCon     Self  11.0    0.0     0.0  188.043619   0.0   3.0  40.915141    0
5127  27500   69579.0  117799.0  DebtCon    Other   5.0    0.0     0.0   84.373124   2.0  32.0  38.862121    0
5092  27200   61118.0   35184.0  DebtCon    Sales  10.0    4.0     2.0  134.648904   7.0  23.0  45.255290    1
```

1.1.2. 열 삭제

특정 열에 결측값이 많을 경우, 해당 **전체 열**을 삭제할 수 있습니다.

```
# 결측값이 있는 열 삭제
drop_cols = df.columns[df.isnull().sum()/len(df) * 100 > 20]
df_dropped_cols = df.drop(drop_cols, axis=1)
print(f"삭제된 열: {drop_cols}")
print(f"삭제후 결과: {df_dropped_cols.shape}")
```

```
>>> print(f"삭제된 열: {drop_cols}")
삭제된 열: Index(['DEBTINC'], dtype='object')
>>> print(f"삭제후 결과: {df_dropped_cols.shape}")
삭제후 결과: (4768, 12)
```

2.1. 결측값 대체(Imputation)

2.1.1. 단순 대체

가장 일반적인 방법으로, 결측값을 특정 값(평균, 중앙값, 최빈값 등)으로 대체합니다.

- **평균으로 대체** (Mean Imputation)

```
# 평균값 대체
df[nums] = df[nums].fillna(df[nums].mean())
```

- 최빈값으로 대체 (Mode Imputation)

```
# 최빈값 대체
df[cats] = df[cats].fillna(df[cats].mode())
```

- 범주형 변수의 원핫 인코딩

```
# 범주형 데이터인 경우
from sklearn.preprocessing import OneHotEncoder

# 범주형 변수 인코딩:넘파이 배열로 반환
encoder = OneHotEncoder(handle_unknown='ignore', sparse_output=False)
encoded_features = encoder.fit_transform(df[cats])

# 인코딩된 데이터프레임 생성
encoded_df = pd.DataFrame(encoded_features, columns=encoder.get_feature_names_out(cats))

# 수치형 변수와 결합
df_combined = pd.concat([df[nums.tolist()+[target]], encoded_df], axis=1)

df_combined.isnull().sum()
print("\n인코딩된 데이터:")
print(df_combined)
```

2.1.2. Scikit-learn의 `SimpleImputer` 사용

Scikit-learn의 `SimpleImputer`를 사용하면 보다 체계적으로 결측값을 대체할 수 있습니다.

```
# SimpleImputer 이용
from sklearn.impute import SimpleImputer

# 수치형 변수
num_imputer = SimpleImputer(strategy='mean')
df_impute_nums = num_imputer.fit_transform(df[nums])
```

```
# 범주형 변수
cat_imputer = SimpleImputer(strategy='most_frequent')
df_impute_cats = cat_imputer.fit_transform(df[cats])
```

2.1.3. 고급 대체 방법

단순 대체 외에도 **예측 모델 기반 대체**나 **다중 대체**(Multiple Imputation) 등의 고급 방법을 사용할 수 있습니다.

- **K-최근접 이웃**(KNN) **기반 대체**

```
# 필요한 패키지
from sklearn.impute import KNNImputer

# KNNImputer 객체 생성
knn_imputer = KNNImputer(n_neighbors=5)

# 수치형 변수만 추출
df_num = df[nums]

# KNNImputer 적용
df_num_imputed = knn_imputer.fit_transform(df_num)

# 결측값 개수 확인
np.isnan(df_num_imputed).sum()

# 데이터프레임으로 변환
df_num_imputed = pd.DataFrame(df_num_imputed, columns=df_num.columns)
```

```
# 결측값 개수 확인
df_num_imputed.isnull().sum()
```

- **반복 대체 (Iterative Imputation)**
  - `IterativeImputer`를 사용하여 여러 번의 예측을 통해 결측값을 대체합니다.
  - 초기값을 평균값 등으로 처리
    - $X_{ij}^{(0)} = \begin{cases} X_{ij}, & i \in \mathcal{O}_j \\ \text{초기 대체값}, & i \in \mathcal{M}_j \end{cases}$
  - 회귀 모델 학습
    - 각 반복 $t = 0, 1, 2, \cdots$ 에 대하여 다음 절차를 수행합니다.
      - 목적함수: $\hat{w}_j = \arg\min_w \{ \sum_{i \in \mathcal{O}_j} (X_{ij} - [X_{i,-j}]^T w)^2 + \lambda ||w||^2 \}$
      - 결측값 업데이트
        - $X_{ij}^{(t+1)} = \begin{cases} X_{ij}, & i \in \mathcal{O}_j \\ [X_{i,-j}^{(t)}]^T \hat{w}_j, & i \in \mathcal{M}_j \end{cases}$
      - 이러한 과정을 모든 변수에 대하여 반복
  - 반복(convergence)
    - 이러한 각 변수에 대한 업데이트를 모든 변수에 대해 수행한 후, 전체 행렬 $X^{(t+1)}$가 생성됩니다.
    - 이 과정은 아래와 같이 반복됩니다:
      - $X^{(0)} \to X^{(1)} \to X^{(2)} \to \cdots \to X^{(T)}$
    - 반복을 종료하는 조건은 미리 정한 최대 반복 횟수 `max_iter`에 도달하거나, consecutive iteration 간 업데이트된 값의 변화(예, Frobenius norm 차이가 `tol` 이하)가 매우 작을 때입니다.

```
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
```

```
df.isnull().sum()
# Iterative Imputer 초기화
iter_imputer = IterativeImputer(max_iter=10, random_state=0)

# 수치형 변수에 Iterative Imputer 적용
df_nums_mi= iter_imputer.fit_transform(df[nums])
print(df_nums_mi)

# 데이터프레임으로 변환
df_nums_mi = pd.DataFrame(df_nums_mi, columns=nums)

# 범주형 변수와 결합
df_mi = pd.concat([df_nums_mi, df[cats], df[target]], axis=1)

# 결과 확인
df_mi.isnull().sum()
```

```
>>> df_mi.isnull().sum()
LOAN         0
MORTDUE      0
VALUE        0
YOJ          0
DEROG        0
DELINQ       0
CLAGE        0
NINQ         0
CLNO         0
DEBTINC      0
```

```
REASON     195
JOB        230
BAD          0
dtype: int64
```

- 다중 대체(mulitple imputation)
  - MICE(Multiple Imputation by Chained Equations):연쇄 방정식을 이용한 다중 대체법으로 각 변수의 결측치를 다른 변수와의 관계를 활용하여 반복적으로 추정하는 방법으로, 여러 번의 예측을 통해 다양한 완전한 데이터셋을 만들어 결측값 문제를 해결
  - 기본 아이디어
    - 데이터 $X \in \mathbb{R}^p \sim \mathcal{N}(\mu, \Sigma)$
    - X_{obs} : 관측된 부분
    - X_{mis} : 결측된 부분
    - 다변량 정규분포의 특성에 따라, 전체 변수에 대해 추정된 평균 μ와 공분산 Σ를 이용하여 결측값의 조건부 분포(conditional distribution)를 구할 수 있습니다.
  - 조건부 분포
    - 데이터를 두 부분, 즉 결측 부분과 관측 부분으로 나누어
    - $X = \begin{pmatrix} X_{mis} \\ X_{obs} \end{pmatrix},\ \mu = \begin{pmatrix} \mu_{mis} \\ \mu_{obs} \end{pmatrix},\ \Sigma = \begin{pmatrix} \Sigma_{mis,mis} & \Sigma_{mis,obs} \\ \Sigma_{obs,mis} & \Sigma_{obs,obs} \end{pmatrix}$
    - 정리하면
    - $X_{mis}|X_{obs} \sim \mathcal{N}\big(\mu_{mis} + \Sigma_{mis,obs}\Sigma_{obs,obs}^{-1}(X_{obs} - \mu_{obs}),\ \Sigma_{mis,obs}\Sigma_{obs,obs}^{-1}\Sigma_{obs,mis}\big)$
    - 와 같이 주어집니다.
    - 이를 통해, 관측된 값에 조건부하여 결측값을 샘플링할 수 있습니다.
  - 다중 대체 절차
    - 단계 1: 파라미터 추정

        - **목표**: 데이터의 완전성(결측치 제거된 데이터)을 가정하고 평균 μ와 공분산 Σ를 추정합니다.
        - 일반적으로 **EM 알고리즘**(Expectation Maximization)을 사용하여 결측값을 임시 대체한 후, 평균 μ와 공분산 Σ를 추정합니다.
    - 단계 2: 대체
        - 추정된 $\hat{\mu}$와 $\hat{\Sigma}$를 기반으로 각 결측치를 위에서 설명한 조건부 분포로부터 샘플링합니다.
        - 이때, 단일 대체가 아니라 여러 개의 (예: `m`) 대체 데이터를 생성하여, 데이터의 불확실성을 반영합니다. 즉,
            - $\{X_{mis}^{(1)}, X_{mis}^{(2)}, \cdots, X_{mis}^{(m)}\}$
        - 와 같이 `m` 개의 대체를 데이터셋을 생성합니다.
    - 단계 3: 반복 및 분석
    - 각각의 대체 데이터셋에 대해 분석(예: 회귀분석, 분류 등)을 수행하고, 결과를 결합(combine)하여 최종 추정값 및 불확실성을 평가합니다.

```python
import statsmodels.api as sm
from statsmodels.imputation.mice import MICEData

# 연속형 데이터 추출
df_num = df[nums]

# MICE 초기화
mice_data = MICEData(df_num)

# 예를 들어, 5번의 임퓨테이션 수행
m = 5
for i in range(m):
    imputed_df = mice_data.next_sample()
    print(f"Imputation {i+1}:\n{imputed_df}\n")
```

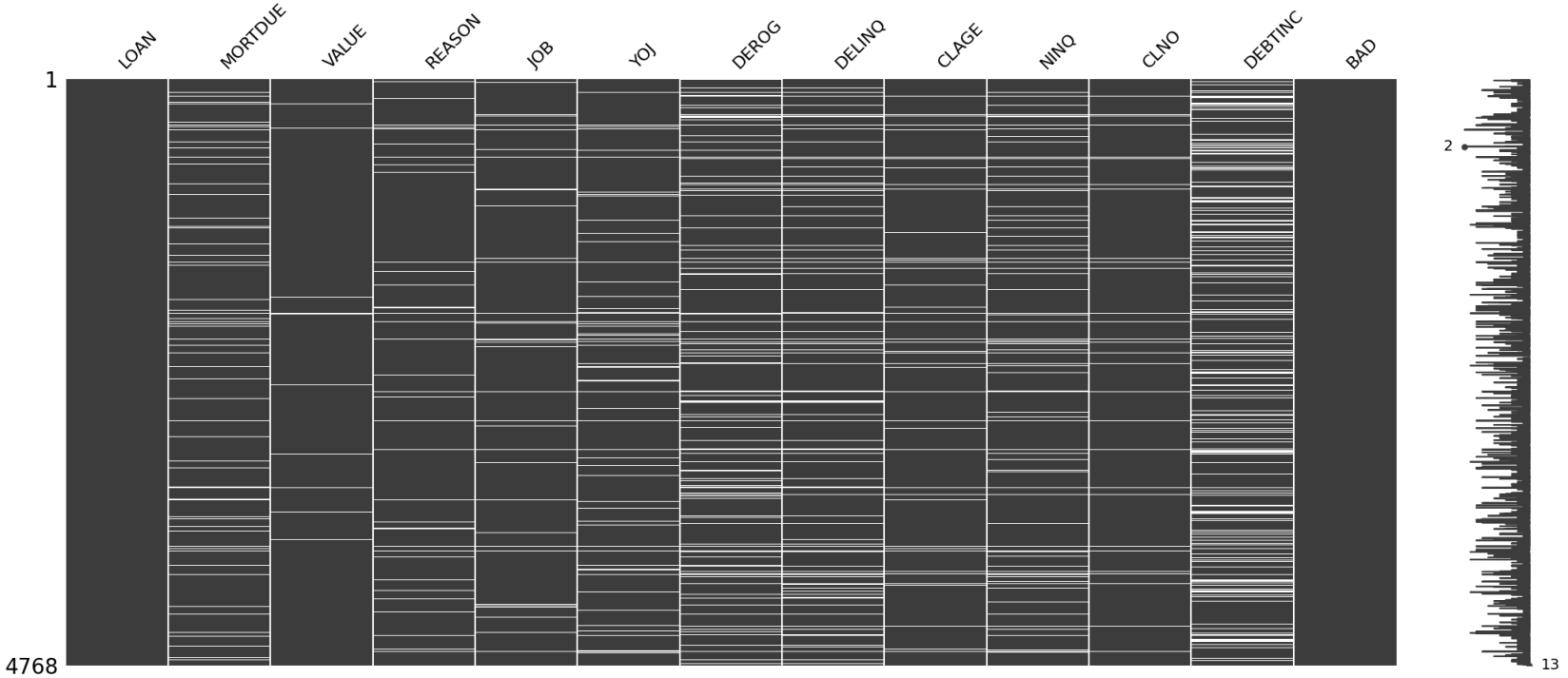
```
Imputation 1:
       LOAN    MORTDUE     VALUE   YOJ  DEROG  DELINQ       CLAGE  NINQ  CLNO    DEBTINC
0      7700   70451.00   81862.0   3.0    0.0     0.0  141.268671   0.0  31.0  31.681943
1     21000   48735.00   71694.0   8.0    0.0     0.0   48.498971   3.0  10.0  32.921401
2     11500   63136.00   81099.0   3.0    1.0     2.0  149.064745   0.0  35.0  28.975940
3      9900   55342.00   72357.0   7.0    0.0     3.0  111.998855   1.0  11.0  39.870340
4      9000   47350.86  105000.0   6.0    0.0     1.0  227.266667   0.0  10.0  26.853259
...     ...        ...       ...   ...    ...     ...         ...   ...   ...        ...
4763  17500   68000.00  133015.0  19.0    1.0     1.0  177.500000   5.0  21.0  40.430695
4764  22400   51470.00   68139.0   9.0    0.0     0.0   31.168696   2.0   8.0  37.952180
4765  15800   31807.00   44144.0   0.0    0.0     0.0  315.812632   1.0   9.0  25.650384
4766  10000  147234.00  175300.0   6.0    0.0     0.0  192.766667   1.0  56.0  40.921433
4767   1500   13500.00   16700.0   4.0    0.0     0.0  149.466667   1.0  10.0  38.739219

[4768 rows x 10 columns]
...
```

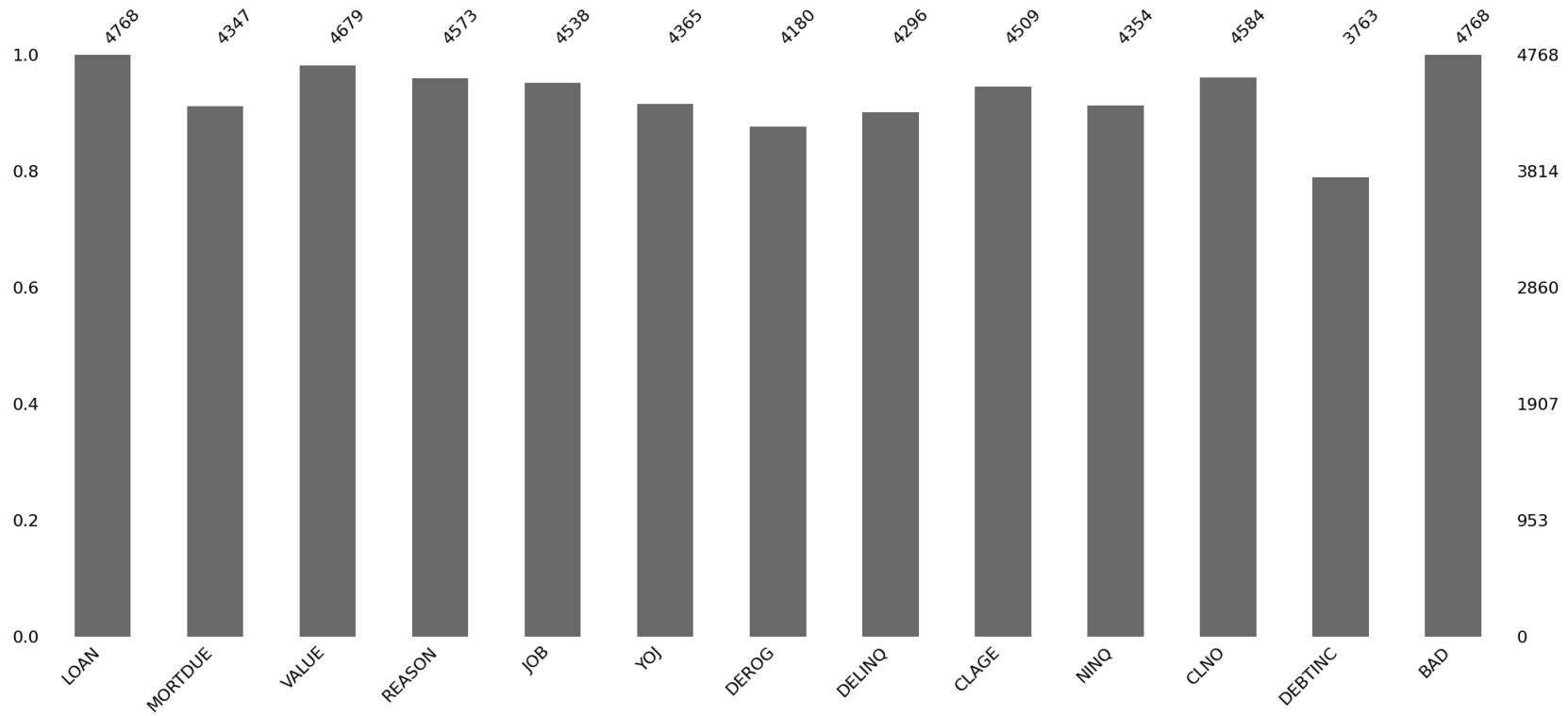
3. 결측값 시각화

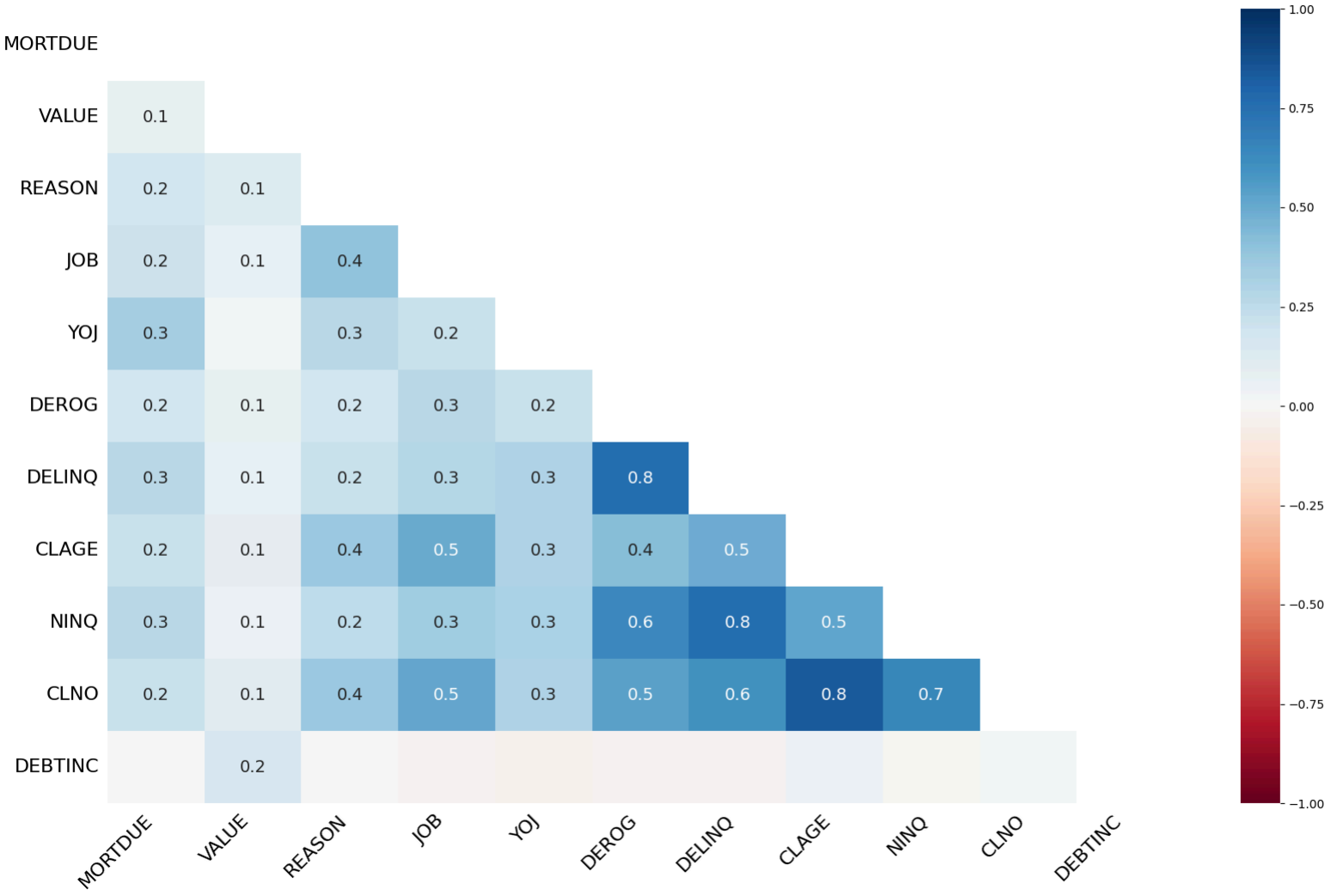
결측값의 분포와 패턴을 시각화하면 데이터의 결측값 특성을 더 잘 이해할 수 있습니다. Seaborn과 Missingno 라이브러리를 사용하면 효과적으로 시각화할 수 있습니다.

```python
import missingno as msno

# 결측값 행렬 시각화
```

```
msno.matrix(df)
plt.show()

# 결측값 열별 시각화
msno.bar(df)
plt.show()

# 결측값 상관관계 시각화
msno.heatmap(df)
plt.show()
```




4768
4347
4679
4573
4538
4365
4180
4296
4509
4354
4584
3763
4768
1.0
0.8
0.6
0.4
0.2
0.0
4768
3814
2860
1907
953
0
LOAN
MORTDUE
VALUE
REASON
JOB
YOJ
DEROG
DELINQ
CLAGE
NINQ
CLNO
DEBTINC
BAD


MORTDUE
VALUE 0.1
REASON 0.2 0.1
JOB 0.2 0.1 0.4
YOJ 0.3 0.3 0.2
DEROG 0.2 0.1 0.2 0.3 0.2
DELINQ 0.3 0.1 0.2 0.3 0.3 0.8
CLAGE 0.2 0.1 0.4 0.5 0.3 0.4 0.5
NINQ 0.3 0.1 0.2 0.3 0.3 0.6 0.8 0.5
CLNO 0.2 0.1 0.4 0.5 0.3 0.5 0.6 0.8 0.7
DEBTINC 0.2
MORTDUE VALUE REASON JOB YOJ DEROG DELINQ CLAGE NINQ CLNO DEBTINC
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

4. 결측값 여부 변수 생성

```
# 결측값 있는 변수만 추출
missing_vars = df[features].columns[df[features].isnull().any()]

# 결측값 변수 이름 지정
missing_vars_indicators = [f'M_{col}' for col in missing_vars]

# 결측값 여부 변수 생성: 결측값이면 1, 아니면 0
for col in missing_vars:
    df[f'M_{col}'] = df[col].isnull().astype(int)
```

- 결측값 여부 변수와 목표 변수와의 상호정보 등의 값을 구한 후 영향력 분석 가능

```
# 타겟변수와 결측값 변수간의 상호정보, 표준화 상호정보, 대칭불확실도 계산
df[missing_vars_indicators].apply(calc_mi, y=df[target], axis=0)

# 표준화 상호정보
df[missing_vars_indicators].apply(calc_normalized_mi, y=df[target], axis=0)

# 대칭불확실도
df[missing_vars_indicators].apply(calc_su, y=df[target], axis=0)
```

1. 숫자형 변수 변환

1.1. Box-Cox, Yeo-Johnson 변환

- 박스-칵스 변환
  - $y(\lambda) = \begin{cases} \frac{(x^\lambda - 1)}{\lambda}, & x > 0, \lambda \neq 0 \\ \ln(x), & x > 0, \lambda = 0 \end{cases}$
- 여-존슨 변환
  - $y(\lambda) = \begin{cases} \frac{((x+1)^\lambda - 1)}{\lambda}, & x \geq 0, \lambda \neq 0 \\ \ln(x+1), & x \geq 0, \lambda = 0 \\ \frac{((-x+1)^{2-\lambda} - 1)}{2-\lambda}, & x < 0, \lambda \neq 2 \\ -\ln(-x+1), & x < 0, \lambda = 2 \end{cases}$
- 최적의 람다는 정규분포에 가까운 정도로 파악

```
# 변수 변환
from sklearn.preprocessing import PowerTransformer

# 여-존슨 변환
pt = PowerTransformer(method='yeo-johnson')
df_nums_array = pt.fit_transform(df[nums])

# 데이터프레임으로 변환
df_nums = pd.DataFrame(df_nums_array, columns=nums)
```

```python
# 적용된 람다 정보: 변수이름: 람다값 형식으로 출력
for i, col in enumerate(df[nums].columns):
    print(f"{col}: {pt.lambdas_[i]:.4f}")
```

```
LOAN: 0.1372
MORTDUE: 0.3270
VALUE: 0.0446
YOJ: 0.2772
DEROG: -7.3870
DELINQ: -3.8925
CLAGE: 0.4642
NINQ: -0.7008
CLNO: 0.6091
DEBTINC: 0.7787
```

1. 2 분위수 변환(quantile transformation)

- 누적분포 함수
  - 데이터 $x_1, \cdots, x_n$ 에 대해 경험적 분포함수 $\hat{F}(x)$ 를 정의합니다.
  - 예를 들어 $\hat{F}(x)$ 는 x 이하의 값이 전체 데이터에서 차지하는 비율입니다.
- 균등 분포
  - $x_{uniform} = \hat{F}(x)$
- 정규 분포
  - $x_{normal} = \Phi^{-1}(\hat{F}(x))$

- `QuantileTransformer`는 데이터의 각 값을 해당 데이터셋 내에서의 상대적인 순위(누적 분포 값, percentile)를 계산한 후, 그 순위를 지정한 목표 분포(예, 균등분포나 정규분포)상의 값으로 매핑합니다. 이 과정은 데이터의 절대적 스케일이나 평균, 표준편차와 무관하게 이루어집니다.

```
# 분위수 변환
from sklearn.preprocessing import QuantileTransformer

# 분위수 변환 적용
qt = QuantileTransformer(output_distribution='normal', random_state=123)
df_nums_array = qt.fit_transform(df[nums])

# 데이터프레임으로 변환
df_nums = pd.DataFrame(df_nums_array, columns=nums)
```

1.3 사전 정의된 변환

방법	식	접두사
Centering	$x_i - \bar{x}$	CNTR_
Exponential	$\begin{cases} e^{x_i}, & \max\{x_i\} \leq 100 \\ e^{x_i - \max\{x_i\}+1}, & else \end{cases}$	EXP_
Inverse	$\text{if } x_i \geq 0 \text{ then } \frac{1}{x_i+1} \text{ else} \frac{1}{x_i-1}$	INV_
Inverse square	$\frac{1}{x_i^2+1}$	INVSQR_
Inverse square root	$\text{if } x_i \geq 0 \text{ then } \frac{1}{\sqrt{x_i+1}} \text{ else} \frac{1}{\sqrt{x_i+\lvert\min\{x_i\}\rvert+1}}$	INVSQRT_

방법	식	접두사
Log	$\text{if } x_i \geq 0 \text{ then } \log x_i + 1 \text{ else } \log \sqrt{x_i + \lvert \min\{x_i\} \rvert + 1}$	LOG_
Log 10	$\text{if } x_i \geq 0 \text{ then } \log_{10} x_i + 1 \text{ else } \log_{10} \sqrt{x_i + \lvert \min\{x_i\} \rvert + 1}$	LOG10_
Square	x_i^2	SQR_
Square root	$\text{if } x_i \geq 0 \text{ then } \sqrt{x_i + 1} \text{ else} \sqrt{x_i + \lvert \min\{x_i\} \rvert + 1}$	SQRT_
Standardization	$\frac{x_i - \bar{x}}{s}$	STD_
Range Standardization	$\frac{x_i - \min\{x_i\}}{\max\{x_i\} - \min\{x_i\}}$	RNG_

1.3.1 중심화 변환

```python
import numpy as np

def center_data(x):
    x = np.asarray(x)
    return x - np.nanmean(x)

# 사용 예
data = np.array([1, 2, 3, 4])
centered = center_data(data)
print("Centered:", centered)
```

#실습 결측값이 입력된다는 가정하에 exp_transform 함수를 작성해보자.

- 데이터

```
data = np.array([10, 20, np.nan, 30])
```

- 지수변환 프로그램 정의

```
def exp_transform(x):
    pass
```

1.3.2 역 변환

```
# 역 변환
def inverse_transform(x):
    x = np.asarray(x, dtype=float)
    return np.where(x>=0, 1/(x+1), np.where(x<0, 1/(x-1), np.nan))
```

#실습 역 제곱 변환 프로그램을 작성해보자. inverse_square_trasnform

1.3.3 역 제곱근 변환

```
# 역 제곱근 변환
def inverse_sqrt_transform(x):
    x = np.asarray(x, dtype=float)
    result = np.where(x>=0, 1/np.sqrt(x+1),
                      np.where(x<0, 1/np.sqrt(x+abs(np.nanmin(x))+1), np.nan))
    return result
```

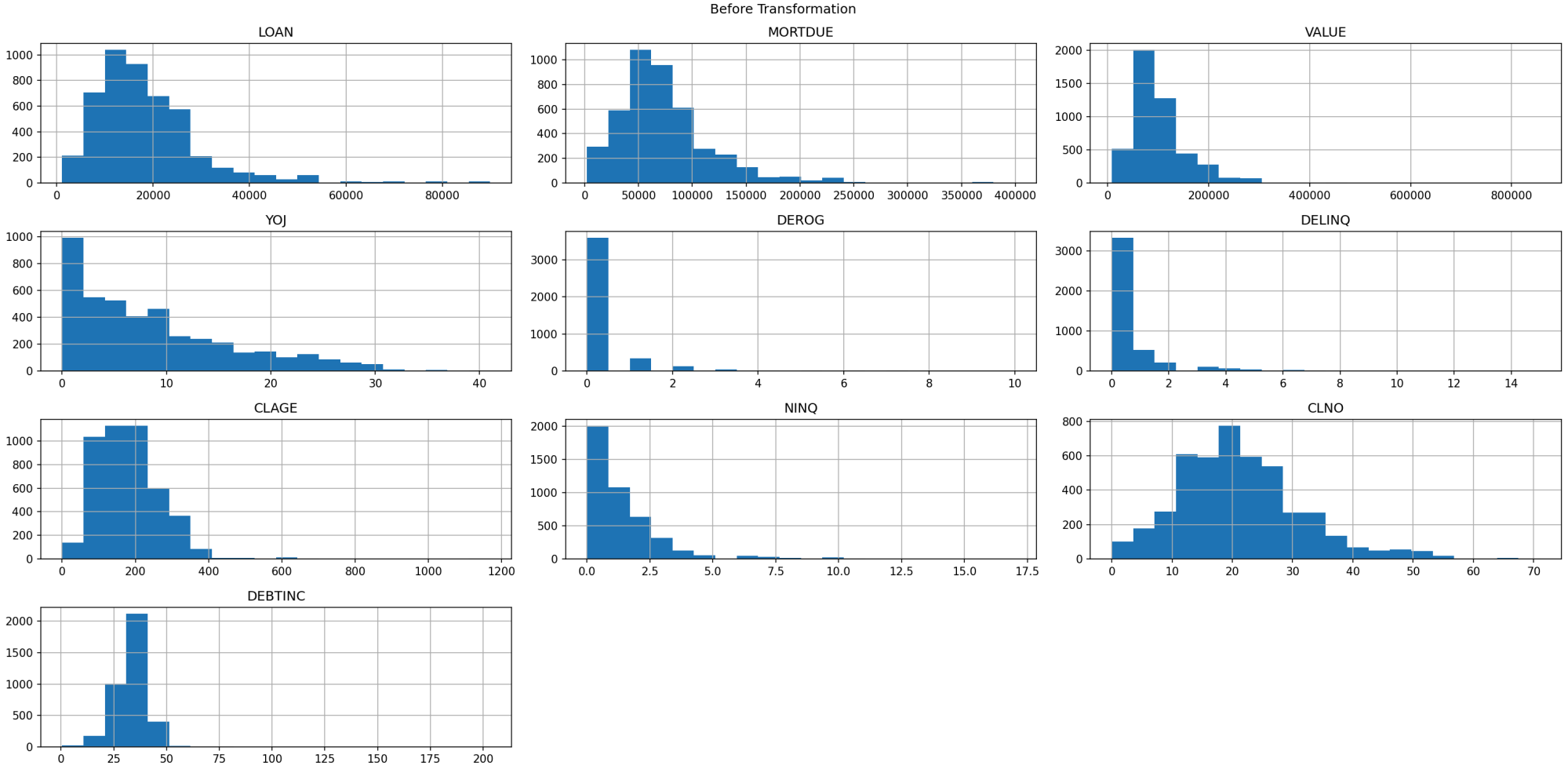
```
# 사용 예
data = np.array([-3, 0, 1, np.nan, 4])
print("Inverse Square Root:", inverse_sqrt_transform(data))
```

#실습

1. 로그 변환 프로그램을 작성해보자. log_transform
2. 상용 로그 변환 프로그램을 작성해보자. log10_transform
3. 제곱 변환 프로그램을 작성해보자. square_transform
4. 제곱근 변환 프로그램을 작성해보자. sqrt_transform
5. 표준화 변환 프로그램을 작성해보자. standardize_transform
6. 범위 표준화 변환 프로그램을 작성해보자. range_standardize_transform

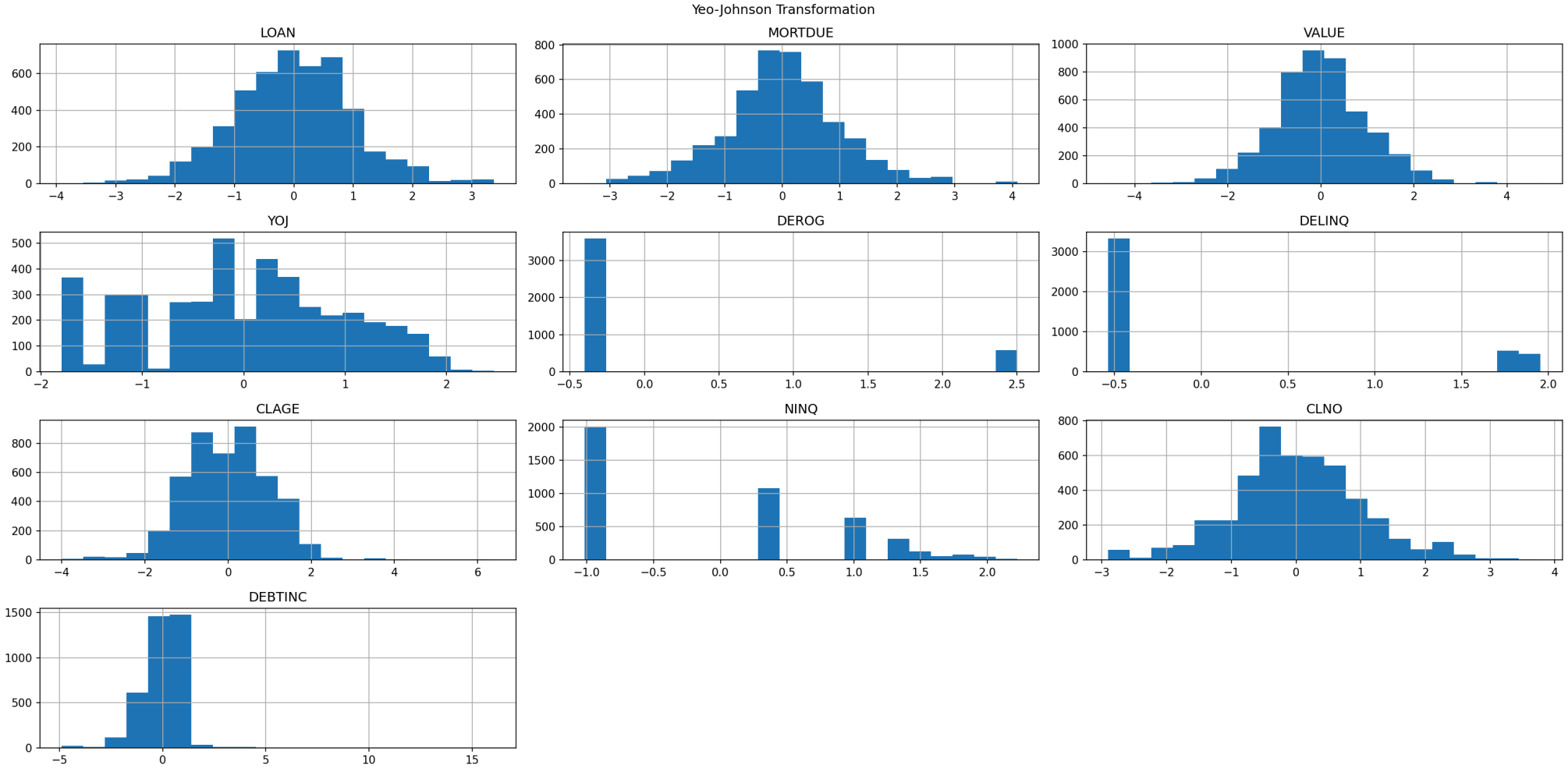
1.4 변환 전/후 그래프 확인

```
# 변환 전 데이터프레임
df_nums = df[nums]

# 히스토그램
df_nums.hist(bins=20, figsize=(20, 10))
plt.suptitle("Before Transformation")
plt.tight_layout()
plt.show(block=False)
```




```
# 여-존슨 변환 적용
pt = PowerTransformer(method='yeo-johnson')
df_nums_array = pt.fit_transform(df[nums])

# 데이터프레임으로 변환
df_nums_yj = pd.DataFrame(df_nums_array, columns=nums)

# 히스토그램
```

```
df_nums_yj.hist(bins=20, figsize=(20, 10))
plt.suptitle("Yeo-Johnson Transformation")
plt.tight_layout()
plt.show(block=False)
```



#실습 분위수 변환 후 그 결과를 그래프로 비교해보자.

1.5 결정나무 기반 구간화

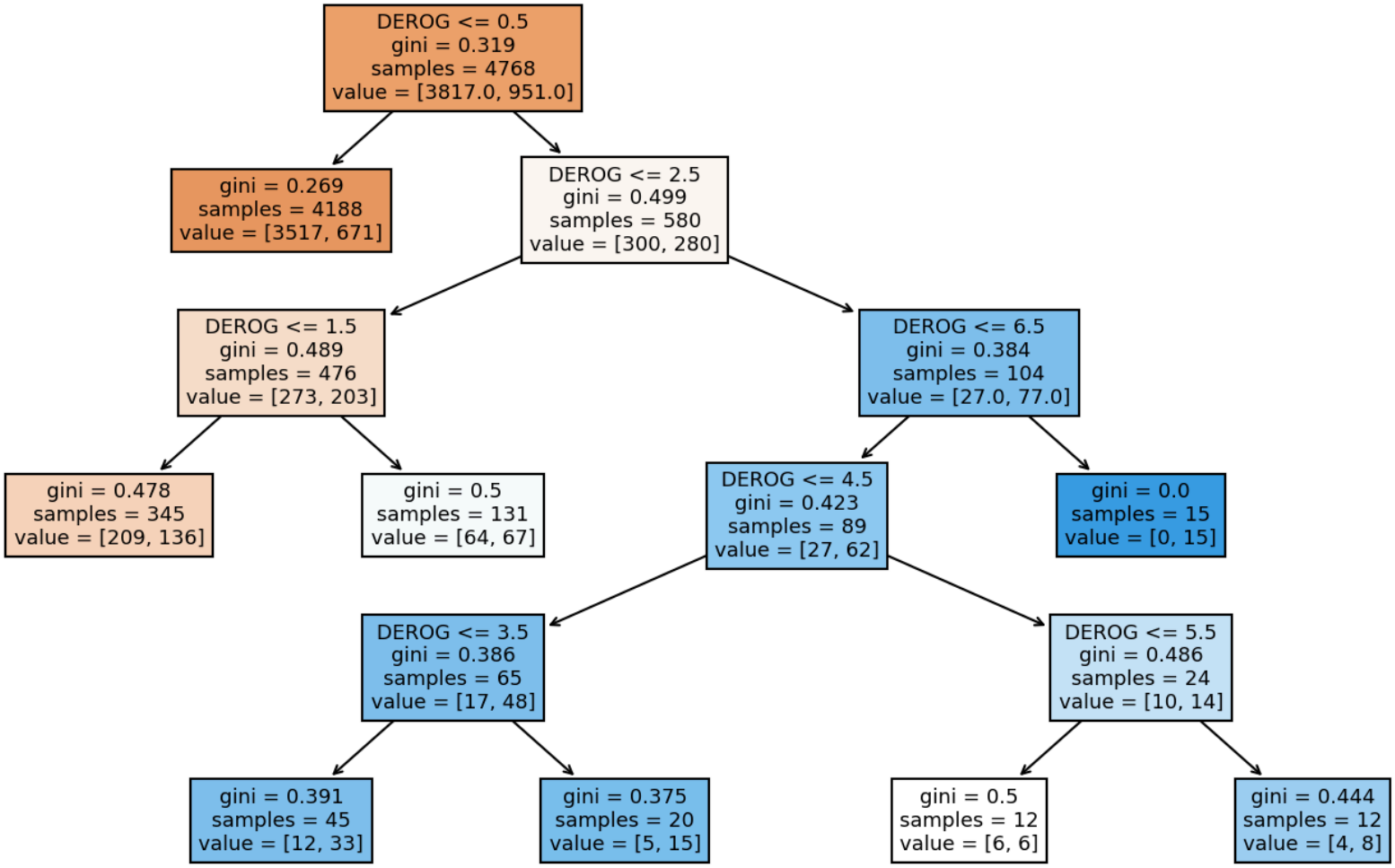
```
def tree_based_binning(series, target, type='C', max_leaf_nodes=4, random_state=42):
    """
    결정트리 기반 구간화를 수행하는 함수입니다.

    Parameters
    ----------
    series : pandas.Series
        구간화를 적용할 연속형 변수 (결측값은 그대로 유지됩니다).
    target : pandas.Series
        구간화의 기준이 될 목표 변수 (범주형 또는 연속형).
    type : str, optional (default='C')
        목표 변수의 유형을 지정합니다.
        'C'인 경우 분류 문제, 'R'인 경우 회귀 문제로 간주합니다.
    max_leaf_nodes : int, optional (default=4)
        결정트리에서 최대 리프 노드 개수를 지정합니다.
        (리프 노드의 개수가 N이면, 구간의 개수는 N이고 분할 기준은 N-1 개가 됩니다.)
    random_state : int, optional (default=42)
        결정트리 모델 재현을 위한 랜덤 시드.

    Returns
    -------
    pd.Series
        원래 시리즈와 동일한 인덱스를 가지며, 각 값에 대해 결정트리 기반 구간 번호를 리턴합니다.
        결측값은 그대로 np.nan으로 유지됩니다.

    Notes
    -----
```

```
    입력된 target 변수 값을 이용하여 결정트리 모델을 학습하고,
    이를 통해 연속형 변수의 분포에 따른 최적의 구간(빈)을 생성합니다.
    """

    from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
    import numpy as np
    import pandas as pd

    # 원본 시리즈 복사 (결측값 유지)
    series = series.copy()

    # 결측값 인덱스 저장
    mask_na = series.isna()

    # 결측값이 아닌 값들로부터 학습 데이터 준비
    non_na_series = series[~mask_na]

    # 입력 feature: 2차원 배열, 타깃: target 변수의 값
    X = non_na_series.values.reshape(-1, 1)
    y = target[~mask_na].values

    # 결정트리 모델 학습: 분류 문제인 경우 DecisionTreeClassifier, 회귀 문제인 경우 DecisionTreeRegressor 사용
    if type == 'C':
        tree = DecisionTreeClassifier(max_leaf_nodes=max_leaf_nodes, random_state=random_state)
    else:
        tree = DecisionTreeRegressor(max_leaf_nodes=max_leaf_nodes, random_state=random_state)
    tree.fit(X, y)

    # 트리의 분할 임계값 추출: tree.tree_.threshold
    # 리프 노드는 -2가 저장되므로, 실제 분할 기준만 추출
```

```
    thresholds = tree.tree_.threshold
    thresholds = thresholds[thresholds > 0]
    thresholds = np.sort(thresholds)

    # 구간 할당 함수: 정렬된 threshold에 따라 구간 번호 부여
    def assign_bin(x, ths):
        for i, thr in enumerate(ths):
            if x < thr:
                return i
        return len(ths)

    # 결측값이 아닌 데이터에 대해 구간 번호를 계산
    binned_non_na = non_na_series.apply(lambda x: assign_bin(x, thresholds))

    # 전체 시리즈와 동일한 인덱스를 갖도록 결합: 결측값은 그대로 np.nan
    binned_series = pd.Series(np.nan, index=series.index)
    binned_series.update(binned_non_na)

    return binned_series

# 사용 예
df_nums = df[nums]
df_nums_binned = df_nums.apply(lambda x: tree_based_binning(x, df[target], max_leaf_nodes=10))
```

1.5.1 결정나무 기반 구간화 예시

- 10개의 구간으로 나누는 경우




```
# 나무 시각화
tree = DecisionTreeClassifier(max_leaf_nodes=10, random_state=42)
```

```
X = df_nums["DEROG"].values.reshape(-1, 1)
y = df[target].values
tree.fit(X, y)

plt.figure(figsize=(10, 5))
plot_tree(tree, feature_names=["DEROG"], filled=True)
plt.show(block=False)
```

2. 범주형 변수 변환

2.1. 변환 방법

방법	식	접두사
Bin rare nominal levels	희귀 범주 비율 이하는 모두 “*OTHER*” 으로 분류	BIN_ , 희귀 범주의 기준, 즉, 비율은 지정 가능함
Level encoding	1, 2, ⋯, K 으로 인코딩	LEVENC_
Level frequency encoding	범주 별 빈도로 인코딩	LEVFRQ_
Level proportion encoding	범주 별 비율로 인코딩	LEVPRP_
One-hot encoding		variable name + category name
Target encoding	범주 별 목표변수의 평균값으로 인코딩	TARGENC_
WOE encoding	증거가중값(weight of evidence) 으로 인코딩	WOEENC_

2.1.1. 명목형 희귀 범주 변환(Bin Rare Nominal Levels)

- 결측값은 그대로

```
def bin_rare_categories(series, threshold=0.05):
    """
    series: 범주형 변수 (pandas Series)
    threshold: 각 범주의 비율이 전체의 threshold 미만이면 "_OTHER_" 처리함.
    결측값은 그대로 유지.
    """
    # 결측값 제외한 값들만 사용해서 비율 계산
    non_na = series.dropna()
    freq = non_na.value_counts(normalize=True)
    # 희귀 범주(비율 < threshold) 확인
    rare_levels = freq[freq < threshold].index
    # 결측값이 아닌 경우 희귀 범주 치환, 결측은 그대로
    return series.apply(lambda x: "_OTHER_" if pd.notnull(x) and x in rare_levels else x)

# 사용 예
df_cats = df[cats]
df_cats_rare = df_cats.apply(bin_rare_categories, threshold=0.05)

# 이름 변경
df_cats_rare.columns = [f"BIN_{col}" for col in df_cats_rare.columns]

# 결과 확인
print(df_cats_rare.isnull().sum() )
print(df_cats_rare.head())
```

```
>>> print(df_cats_rare.isnull().sum() )
BIN_REASON    195
BIN_JOB       230
dtype: int64
>>> print(df_cats_rare.head())
  BIN_REASON  BIN_JOB
0    DebtCon  _OTHER_
1    DebtCon    Other
2    DebtCon    Other
3    DebtCon      Mgr
4    DebtCon  ProfExe
```

2.1.2. 수준 숫자 인코딩(Level encoding)

- 범주를 숫자 또는 다른 문자로 인코딩
- 결측값은 그대로

```python
from sklearn.preprocessing import LabelEncoder

def level_encoding(series):
    # 원본 시리즈의 복사본 생성
    temp = series.copy()
    mask_na = temp.isna()
    # 결측값은 제외한 나머지 값에 대해 LabelEncoder 적용
    temp_nonan = temp[~mask_na]
    le = LabelEncoder()
    encoded_nonan = pd.Series(le.fit_transform(temp_nonan.astype(str)), index=temp_nonan.index)
```

```
    # 결과를 원본 시리즈에 결측값은 그대로 둔 채 결합
    encoded = pd.Series(np.nan, index=series.index)
    encoded.update(encoded_nonan)
    print(f"Encoded {series.name}: {le.classes_}")
    return encoded, le.classes_

# 사용 예
df_cats = df[cats]
df_cats_level_encoded, mapping_rules = zip(*[level_encoding(df_cats[col]) for col in df_cats.columns])

# 튜플을 데이터프레임으로 변환
df_cats_level_encoded = pd.concat(df_cats_level_encoded, axis=1)

# 이름 변경
df_cats_level_encoded.columns = [f"LEVENC_{col}" for col in df_cats.columns]

# 결측값 유지 여부 확인
print(df_cats_level_encoded.isnull().sum())

# 매핑 정보 확인
print(mapping_rules)
```

2.1.3. 수준 빈도 인코딩 (Level frequency encoding)

- 각 범주를 그 범주가 전체에서 나타나는 빈도로 매핑
- 결측값은 그대로

```
# 사용 예시:
# map 함수는 각 범주에 대해 빈도수를 매핑. 결측값은 그대로 유지
```

```
df_cats = df[cats]
df_cats_freq_encoded = df_cats.apply(lambda x: x.map(x.value_counts()))

# 이름 변경
df_cats_freq_encoded.columns = [f"LEVFRQ_{col}" for col in df_cats_freq_encoded.columns]

# 결측값 유지 여부 확인
print(df_cats_freq_encoded.isnull().sum())
```

2.1.4. Level proportion encoding

- 각 범주를 그 범주가 전체에서 나타나는 비율로 매핑
- 결측값은 그대로

```
# 사용 예시:
# map 함수는 각 범주에 대해 비율을 매핑. 결측값은 그대로 유지
df_cats = df[cats]
df_cats_prop_encoded = df_cats.apply(lambda x: x.map(x.value_counts(normalize=True)))

# 이름 변경
df_cats_prop_encoded.columns = [f"LEVPRP_{col}" for col in df_cats_prop_encoded.columns]

# 결측값 유지 여부 확인
print(df_cats_prop_encoded.isnull().sum())
```

2.1.5. One-hot encoding

- 각 범주를 더미 변수(이진 변수)로 변환합니다.

- 기본적으로 `pd.get_dummies()` 는 결측값(NaN)을 무시합니다.
- **결측값을 그대로 두려면**, 결측값은 그대로 둔 후 별도로 처리하거나, 결측값을 다른 열로 연결하지 않는 방식으로 진행합니다.

```
# 사용 예
df_cats = df[cats]

# 결측값을 "Missing"으로 처리
df_cats.fillna("Missing_", inplace=True)

# 결측값을 "Missing"으로 처리한 후 one-hot encoding
df_cats_onehot_encoded = pd.get_dummies(df_cats, prefix=cats).astype(int)
```

```
>>> df_cats_onehot_encoded.head()
   REASON_DebtCon  REASON_HomeImp  REASON_Missing_  JOB_Mgr  JOB_Missing_  JOB_Office  JOB_Other  JOB_ProfExe  JOB_Sales
JOB_Self
0               1               0                0        0             0           0          0            0          1
0
1               1               0                0        0             0           0          1            0          0
0
2               1               0                0        0             0           0          1            0          0
0
3               1               0                0        1             0           0          0            0          0
0
4               1               0                0        0             0           0          0            1          0
0
```

2.1.6. Target encoding

- 각 범주를 대상(타깃) 변수의 평균값으로 매핑합니다.
- 계산 시 결측값은 제외하고, 매핑 후 결측값은 그대로 둡니다.

```
def target_encoding(train_series, target):
    """
    train_series: 범주형 변수 (pandas Series)
    target: 목표 변수 (pandas Series)
    결측값은 계산 및 매핑에서 제외하고, 최종적으로는 결측값 그대로 둔다.
    """
    df_temp = pd.DataFrame({"category": train_series, "target": target})
    # 결측값이 아닌 경우에 대해서만 평균 계산
    target_mean = df_temp.loc[df_temp["category"].notna()].groupby("category")["target"].mean()
    # map: 결측값은 그대로
    return train_series.map(target_mean)


# 사용 예
df_cats = df[cats]
df_cats_target_encoded = df_cats.apply(lambda x: target_encoding(x, df[target]))

# 이름 변경
df_cats_target_encoded.columns = [f"TARGENC_{col}" for col in df_cats_target_encoded.columns]

# 결측값 유지 여부 확인
print(df_cats_target_encoded.isnull().sum())
```

2.1.6. WOE encoding

- 각 범주 내의 이벤트 건수, 비 이벤트 건수를 각각 e_i, h_i 이고 전체 이벤트 건수, 비 이벤트 건수를 각각 e, h 이라고 할 때
- $WOE_i = \log \frac{\frac{h_i}{h}}{\frac{e_i}{e}} = \log \frac{q_i}{p_i}$
- 이 때 p_i, q_i 중 하나라도 0이면 $WOE_i = \log \frac{\frac{h_i + woeAdj}{h}}{\frac{e_i + woeAdj}{e}}$ 와 같이 수정되면 기본 $woeAdj = \frac{1}{2}$ 이다.
- 계산 시 결측값은 제외하고, 매핑 후 결측값은 그대로 둡니다.

```
def woe_encoding(train_series, target, epsilon=0.5):
    """
    train_series: 범주형 변수 (pandas Series)
    target: 이진 타깃 (0,1) (pandas Series)
    epsilon: 0으로 나누는 경우 방지를 위한 작은 값.
    결측값은 계산 시 제외하고, 매핑 후에는 그대로 NaN으로 유지.
    """
    df_temp = pd.DataFrame({"category": train_series, "target": target.astype(float)})
    # 결측값을 제외한 경우에 대해 전체 Good/Bad 계산
    total_good = (df_temp.loc[df_temp["category"].notna(), "target"] == 1).sum()
    total_bad = (df_temp.loc[df_temp["category"].notna(), "target"] == 0).sum()

    # 각 범주별로 계산 (결측값은 제외됨)
    stats = df_temp.loc[df_temp["category"].notna()].groupby("category")["target"].agg(
        good=lambda x: (x == 1).sum().astype(float),
        bad=lambda x: (x == 0).sum().astype(float)
    )

    # good, bad  둘 중에 하나라도 0인 경우에는 log(0)이 되므로, 둘 다 동시에 epsilon으로 대체
    # 둘 중에 하나라도 0인 경우 검사
    stats.loc[(stats["good"] == 0.0) | (stats["bad"] == 0.0), ["good", "bad"]] = epsilon
```

```
    # WOE 계산
    stats["WOE"] = np.log((stats["good"] / total_good) / (stats["bad"] / total_bad))
    # map: 결측값은 그대로 남음. 단 해당 결측값을 None으로 처리. 결측값에 한해서
    return train_series.map(stats["WOE"].to_dict()).where(train_series.notna(), None)
```

3. 변환 적용 방법 예시

문제	처방	비고
큰 왜도	Box-Cox 변환	숫자형 변수
결측값	중간값 대체	숫자형 변수
결측값 비율 높음	결측값을 새로운 범주로 두고 나머지 범주들에는 희귀범주 처리 적용(missing level and group rare)	범주형 변수
결측값 비율 높음	결측값을 새로운 범주로 두고 나머지 범주들과 함께 레벨 인코딩 적용(missing level and level encoding)	범주형 변수
결측값	결측 표시자(missing indicator) 변수 생성	공통
결측값 비율 작음	최빈값 대체 후 희귀범주 처리 적용(mode imputation and group rare)	범주형 변수
결측값 비율 작음	최빈값 대체 후 레벨 인코딩 적용(mode imputation and level encoding)	범주형 변수
심각한 비대칭도, 첨도, 이상치	결측값 구간을 갖는 분위수 기반 구간화(Quantile binning with missing bins)	숫자형 변수
심각한 비대칭도, 첨도, 이상치. 범주형인 경우에 집합크기가 클 때 적용	목표 변수가 숫자형인 경우에 회귀나무를 이용한 구간화 (Regression tree binning)	공통

문제	처방	비고
심각한 비대칭도, 첨도, 이상치. 범주형인 경우에 집합크기가 클 때 적용	목표 변수가 범주형인 경우 분류나무를 이용한 구간화(Decision tree binning)	공통
범주형인 경우에 집합크기가 클 때	타겟 인코딩. 회귀 문제인 경우는 평균값, 최소값, 최대값을 고려. 분류 문제인 경우는 상대빈도, 이벤트 확률, 증거가중값 고려. 목표변수 개수 인코딩(label count encoding) 등은 회귀, 분류 문제 동시에 가능	범주형 변수
큰 첨도	여-존슨 변환	숫자형 변수

- 선별 정책

정책	기본값	비고
constant	True	해당 변수는 제거 대상
groupRareLevels	True(〈1%)	희귀범주값을 갖는 범주형변수 색출. 범주 제거 또는 병합 필요
leakagePerentThreshold	〉90	누수율(엔트로피감소율)= ((H(Y)-H(Y｜X))/H(Y) ×100). 경계값 보다 큰 경우 제거 대상. 100% 설명되면 H(Y|X)=0.
lowCv	True(〈0.1)	CV 값이 작은 변수 색출. 경계값 보다 작으면 제거 대상
lowMutualInformation	≤0.05	상호정보값이 작은 변수 색출, 표준화 상호정보는 0과 1사이의 값을 가짐
missingIndicatorPercent	〉75	결측률이 기준보다 큰 변수는 결측표시자 변수를 생성하여 보존하고 기존 변수는 제거
missingPercentThreshold	〉90	결측률이 기준보다 큰 변수는 삭제
redundant	=1	대칭 불확실도(SU=2×I(X;Y)/(H(X)+H(Y)))가 높은 중복 변수 색출 임계값이 1인 경우는 두 변수가 100% 일치한다는 의미이며 중복검사를 실시하지 않음

0. 결측값 처리

```
# 결측값 처리: 단순 함수
def simple_imputer(df, target, nums, cats):
    df_pre = df.drop(target, axis=1).copy()
    df_pre[nums] = df[nums].fillna(df[nums].mean().to_dict())
    df_pre[cats] = df[cats].fillna(df[cats].mode().iloc[0].to_dict())
    df_pre[target] = df[target]
    return df_pre

# FunctionTransformer로 변환
from sklearn.preprocessing import FunctionTransformer

# 변환기 생성
ft = FunctionTransformer(simple_imputer, kw_args={'target':target,  'nums': nums, 'cats': cats})

# 적용
df_pre = ft.fit_transform(df)

# 결과 확인
df_pre.isnull().sum()
```

```
>>> df_pre.isnull().sum()
LOAN       0
MORTDUE    0
VALUE      0
```

```
REASON     0
JOB        0
YOJ        0
DEROG      0
DELINQ     0
CLAGE      0
NINQ       0
CLNO       0
DEBTINC    0
BAD        0
dtype: int64
```

1. 상수값 변수

- 분산이 0인 변수

```
# 상수 변수 판별 함수
def find_constant_vars(df, target):
    constant_vars = []
    for col in df.drop(target, axis=1).columns:
        # dropna()를 사용하여 결측값 제외
        if df[col].dropna().nunique() <= 1:
            constant_vars.append(col)
    return constant_vars

# 적용
constant_vars = find_constant_vars(df_pre, target)

# 결과 확인
print("상수 변수:", constant_vars)
```

```
>>> print("상수 변수:", constant_vars)
상수 변수: []
```

2. groupRareLevels (희귀 범주값 처리)

- 범주형 변수에서, 전체 데이터에서 각 범주의 비율이 매우 낮은(기본값: 1% 미만) 범주가 존재하는 경우를 식별합니다.
  - 이러한 변수는 희귀 범주를 제거하거나 "*OTHER*"로 병합하는 등의 후처리가 필요합니다.

```
# 정책을 적용하는 함수 (희귀 범주 비율이 threshold 미만인 범주가 있으면 후보로 선택)
def find_group_rare_vars(df, cats, threshold=0.01):
    rare_vars = []
    for col in df[cats].columns:
        counts = df[col].value_counts(normalize=True, dropna=True)
        if len(counts) > 0 and counts.min() < threshold:
            rare_vars.append(col)
    return rare_vars

# 적용
rare_vars = find_group_rare_vars(df_pre, cats)

# 결과 확인
print("희귀 범주 비율 변수:", rare_vars)
```

```
>>> print("희귀 범주 비율 변수:", rare_vars)
희귀 범주 비율 변수: []
```

3. leakagePerentThreshold (누수율, 엔트로피 감소율)

- 엔트로피
  - $H(X) = -\sum_{i} p(x_i) \log_2 p(i)$

```
# 엔트로피 계산
def calc_entropy_x(df: pd.DataFrame, x: str)-> float:
    p_x = df[x].value_counts(normalize=True)
    entropy = -np.sum(p_x * np.log2(p_x))
    return entropy
```

#실습

1. 시리즈 엔트로피 함수를 모든 범주형 컬럼에 대하여 적용하는 함수를 적용해보자. map 함수 이용.

- 결합 엔트로피
  - $H(X, Y) = -\sum_{x,y} p(x, y) \log_2 p(x, y)$

```
# 결합 엔트로피 계산
def calc_entropy_joint_xy(df: pd.DataFrame, x, y):
    p_xy = df.groupby([x, y]).size() / len(df)
    entropy = -np.sum(p_xy * np.log2(p_xy))
    return entropy

# 변수 리스트에 대한 결합 엔트로피 계산
```

```python
def calc_entropy_joint(df: pd.DataFrame, cats: list)-> pd.Series:
    # 모든 변수의 조합에 대한 결합 엔트로피 계산
    entropies = np.zeros((len(cats), len(cats)))
    for x in cats:
        for y in cats:
            entropy = calc_entropy_joint_xy(df, x, y)
            entropies[cats.index(x), cats.index(y)] = entropy
    return pd.DataFrame(entropies, index=cats, columns=cats)

# 적용
entropy_joint = calc_entropy_joint(df_pre, cats.tolist()+[target])

# 결과 확인
print(f"결합 엔트로피:\n{entropy_joint}")
```

```
>>> print(f"결합 엔트로피:\n{entropy_joint}")
결합 엔트로피:
          REASON       JOB       BAD
REASON  0.885002  2.927256  1.604780
JOB     2.927256  2.054786  2.765743
BAD     1.604780  2.765743  0.720836
```

- 조건부 엔트로피
  - $H(X|Y) = -\sum_{y\in\mathcal{Y}} p(y) \sum_{x\in\mathcal{X}} p(x|y)\log_2 p(x|y) = -\sum_{y\in\mathcal{Y}} p(y) H(X|y)$

```python
# 조건부 엔트로피 계산: H(x|y)
def calc_entropy_conditional_xy(df, x, y):
    entropy = calc_entropy_joint_xy(df, x, y) - calc_entropy_x(df, y)
    return entropy

# 조건부 엔트로피 계산: 변수 리스트
def calc_entropy_conditional(df: pd.DataFrame, cats: list)-> pd.DataFrame:
    entropies = np.zeros((len(cats), len(cats)))
    for x in cats:
        for y in cats:
            entropy = calc_entropy_conditional_xy(df, x, y)
            entropies[cats.index(x), cats.index(y)] = entropy
    return pd.DataFrame(entropies, index=cats, columns=cats)


# 적용
entropy_cond = calc_entropy_conditional(df_pre, cats.tolist()+[target])

# 결과 확인
print(f"조건부 엔트로피:\n{entropy_cond}")
```

```
>>> print(f"조건부 엔트로피:\n{entropy_cond}")
조건부 엔트로피:
          REASON           JOB       BAD
REASON  0.000000  8.724702e-01  0.883944
JOB     2.042254  4.440892e-16  2.044907
BAD     0.719778  7.109579e-01  0.000000
```

- 상호 정보 (mutual information)
  - $I(X;Y) = H(X) - H(X|Y)$

```
# 상호정보 계산
def calc_mutual_info_xy(df, x, y):
    mi = calc_entropy_x(df, x) - calc_entropy_conditional_xy(df, x, y)
    return mi

# 상호정보 계산: 변수 리스트
def calc_mutual_info(df: pd.DataFrame, cats: list)-> pd.Series:
    entropies = np.zeros((len(cats), len(cats)))
    for x in cats:
        for y in cats:
            mi = calc_mutual_info_xy(df, x, y)
            entropies[cats.index(x), cats.index(y)] = mi
    return pd.DataFrame(entropies, index=cats, columns=cats)

# 적용
mi = calc_mutual_info(df_pre, cats.tolist()+[target])

# 결과 확인
print(f"상호정보:\n{mi}")
```

```
>>> print(f"상호정보:\n{mi}")
상호정보:
          REASON       JOB       BAD
REASON  0.885002  0.012532  0.001058
```

```
JOB     0.012532  2.054786  0.009878
BAD     0.001058  0.009878  0.720836
```

- 표준화 상호 정보(normalized MI)
  - $\sqrt{1 - e^{-2I(X;Y)}}$

```python
# 표준화 상호 정보 계산
def calc_mutual_info_std_xy(df, x, y):
    mi = calc_mutual_info_xy(df, x, y)
    mi_std = np.sqrt(1-np.exp(-2*mi))
    return mi_std)

# 표준화 상호 정보 계산: 변수 리스트
def calc_mutual_info_std(df: pd.DataFrame, cats: list)-> pd.DataFrame:
    entropies = np.zeros((len(cats), len(cats)))
    for x in cats:
        for y in cats:
            mi = calc_mutual_info_std_xy(df, x, y)
            entropies[cats.index(x), cats.index(y)] = mi
    return pd.DataFrame(entropies, index=cats, columns=cats)

# 적용
mi_std = calc_mutual_info_std(df_pre, cats.tolist()+[target])

# 결과 확인
print(f"표준화 상호정보:\n{mi_std}")
```

```
>>> print(f"표준화 상호정보:\n{mi_std}")
표준화 상호정보:
          REASON       JOB       BAD
REASON  0.910861  0.157327  0.045969
JOB     0.157327  0.991759  0.139866
BAD     0.045969  0.139866  0.873767
```

- 누수율
  - $LKR = \frac{H(X)-H(X|Y)}{H(Y)} \times 100 = \frac{I(X;Y)}{H(Y)} \times 100$

```python
# 누수율(Leakage Rate) 계산
def calc_leakage_rate_xy(df, x, y):
    mi = calc_mutual_info_xy(df, x, y)
    entropy = calc_entropy_x(df, y)
    lkg = mi / entropy * 100
    return lkg

# 누수율 계산: 변수 리스트
def calc_leakage_rate(df: pd.DataFrame, cats: list)-> pd.Series:
    entropies = np.zeros((len(cats), len(cats)))
    for x in cats:
        for y in cats:
            lkg = calc_leakage_rate_xy(df, x, y)
            entropies[cats.index(x), cats.index(y)] = lkg
    return pd.DataFrame(entropies, index=cats, columns=cats)

# 적용
```

```
leakage_rate = calc_leakage_rate(df_pre, cats.tolist()+[target])

# 결과 확인
print(f"누수율:\n{leakage_rate}")
```

```
누수율:
            REASON         JOB         BAD
REASON  100.000000    0.609874    0.146732
JOB       1.415997  100.000000    1.370386
BAD       0.119514    0.480743  100.000000
```

- 누수율 감지

```
# 누수율 감지
threshold = 90
leakage_vars = leakage_rate[target][leakage_rate[target] > threshold][:-1].tolist()

# 결과 확인
print("누수율 변수:", leakage_vars)
```

```
>>> print("누수율 변수:", leakage_vars)
누수율 변수: []
```

4. lowCv (계수변동, CV)

- 입력 변수의 계수변동(CV = 표준편차/평균)이 너무 낮으면(기본값: CV 〈 0.1) 정보 변화가 거의 없다고 판단해 제거 대상
  - $CV = \frac{s}{\bar{x}}$

```
# low CV 변수 찾기
def find_low_cv_vars(df, nums, threshold=0.1):
    low_cv_vars = []
    for col in nums:
        cv = df[col].std() / df[col].mean()
        if cv < threshold:
            low_cv_vars.append(col)
    return low_cv_vars

# 적용
low_cv_vars = find_low_cv_vars(df_pre, nums)

# 결과 확인
print("low CV 변수:", low_cv_vars)
```

```
>>> print("low CV 변수:", low_cv_vars)
low CV 변수: []
```

5. lowMutualInformation (상호정보량)

- 목표 변수와의 상호정보량(MI)이 너무 낮은 경우 (기본값: ≤ 0.05) 해당 변수는 목표 변수를 설명하는 데 거의 기여하지 않는다고 판단하여 제거 대상

```
# low MI 변수 찾기
def find_low_mi_vars(df, cats, target, threshold=0.05):
    low_mi_vars = cats[[calc_mutual_info_xy(df, col, target) < threshold for col in cats]].tolist()
    return low_mi_vars

# 적용
low_mi_vars = find_low_mi_vars(df_pre, cats, target)

# 결과 확인
print("low MI 변수:", low_mi_vars)
```

```
>>> print("low MI 변수:", low_mi_vars)
low MI 변수: ['REASON', 'JOB']
```

6. missingIndicatorPercent (결측률이 높은 경우)

- 변수의 결측값 비율이 지정된 기준(기본값: 75%)을 초과할 경우,
- 해당 변수는 단순히 제거하기보다는 결측 표시자를 별도로 생성한 후, 원래 변수는 제거하는 방향을 고려합니다.

```
# 결측값이 임계값보다 크면 표시자 변수 생성
def missing_indicator_df(df, target, threshold=20):
    df_temp = df.copy()
    missing_vars = df_temp.drop(target, axis=1).isnull().mean() * 100
    missing_vars = missing_vars[missing_vars > threshold].index.tolist()
    # 표시자 변수 생성
    for col in missing_vars:
        df_temp["M_" + col ] = df_temp[col].isnull().astype(int)
        # 해당 변수 drop
        df_temp = df_temp.drop(col, axis=1)
    return df_temp

# 적용
df_missing = missing_indicator_df(df, target)

# 결과 확인
print(df_missing.info())
```

```
>>> print(df_missing.info())
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4768 entries, 0 to 4767
Data columns (total 13 columns):
```

```
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   LOAN       4768 non-null   int64
 1   MORTDUE    4347 non-null   float64
 2   VALUE      4679 non-null   float64
 3   REASON     4573 non-null   object
 4   JOB        4538 non-null   object
 5   YOJ        4365 non-null   float64
 6   DEROG      4180 non-null   float64
 7   DELINQ     4296 non-null   float64
 8   CLAGE      4509 non-null   float64
 9   NINQ       4354 non-null   float64
 10  CLNO       4584 non-null   float64
 11  BAD        4768 non-null   int64
 12  M_DEBTINC  4768 non-null   int32
dtypes: float64(8), int32(1), int64(2), object(2)
memory usage: 465.8+ KB
```

7. missingPercentThreshold (결측률이 매우 높은 경우 삭제)

- 변수의 결측률이 너무 높아(기본값: 90%) 복구가 어려운 경우, 해당 변수를 완전히 삭제합니다.

```
# 결측률이 임계값 이상이면 삭제
def drop_missing_vars(df, target, threshold=20):
    missing_vars = df.drop(target, axis=1).isnull().mean() * 100
    missing_vars = missing_vars[missing_vars > threshold].index.tolist()
    df_temp = df.drop(missing_vars, axis=1)
    return df_temp, missing_vars

# 적용
df_drop_missing, missing_vars = drop_missing_vars(df, target)

# 결과 확인
print("결측값 삭제 변수:", missing_vars)
print("삭제후 변수 정보")
print(df_drop_missing.info())
```

```
>>> print("결측값 삭제 변수:", missing_vars)
결측값 삭제 변수: ['DEBTINC']
>>> print("삭제후 변수 정보")
삭제후 변수 정보
>>> print(df_drop_missing.info())
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4768 entries, 0 to 4767
Data columns (total 12 columns):
```

```
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   LOAN     4768 non-null   int64  
 1   MORTDUE  4347 non-null   float64
 2   VALUE    4679 non-null   float64
 3   REASON   4573 non-null   object 
 4   JOB      4538 non-null   object 
 5   YOJ      4365 non-null   float64
 6   DEROG    4180 non-null   float64
 7   DELINQ   4296 non-null   float64
 8   CLAGE    4509 non-null   float64
 9   NINQ     4354 non-null   float64
 10  CLNO     4584 non-null   float64
 11  BAD      4768 non-null   int64  
dtypes: float64(8), int64(2), object(2)
memory usage: 447.1+ KB
```

8. redundant (중복 변수 제거, 대칭 불확실도 SU)

- 두 변수 간 비대칭 불확실도(SU, Symmetric Uncertainty)가 1이면(즉, 두 변수가 100% 일치하면) 중복 변수로 판단합니다.
  - $SU = \frac{2I(X;Y)}{H(X)+H(Y)}$

```
# 대칭 블확실도 계산
def calc_su_xy(df, x, y):
    mi = calc_mutual_info_xy(df, x, y)
    h_x = calc_entropy_x(df, x)
    h_y = calc_entropy_x(df, y)
    su = 2 * mi / (h_x + h_y)
    return su

# 대칭 불확실도 계산: 변수 리스트
def calc_su(df: pd.DataFrame, cats: list)-> pd.DataFrame:
    entropies = np.zeros((len(cats), len(cats)))
    for x in cats:
        for y in cats:
            su = calc_su_xy(df, x, y)
            entropies[cats.index(x), cats.index(y)] = su
    return pd.DataFrame(entropies, index=cats, columns=cats)

# 적용
su = calc_su(df_pre, cats.tolist())

# 결과 확인
print(f"SU:\n{su}")
```

```
>>> print(f"SU:\n{su}")
SU:
          REASON       JOB
REASON  1.000000  0.008526
JOB     0.008526  1.000000
```

입력 변수 영향도 분석

1. 상호정보 기반 중요도

- 상호정보를 적용하기 위해서는 숫자형 변수의 범주화가 필요

```
# 구간화 정의: type은 'qcut' 또는 'cut' 즉, '등분절단' 또는 '등간절단', 원래 변수는 삭제
def quantile_transform(df, nums, bins=10, type='qcut')-> pd.DataFrame:
    df_pre = df[nums].copy()
    for col in nums.tolist():
        # 10개 구간으로 변환
        if type == 'qcut':
            df_pre[f"BIN_"+col] = pd.qcut(df_pre[col], bins, duplicates='drop')
        else:
            df_pre[f"BIN_"+col] = pd.cut(df_pre[col], bins, duplicates='drop')
        df_pre[f"BIN_"+col] = pd.qcut(df_pre[col], bins, duplicates='drop')
        df_pre = df_pre.drop(col, axis=1)
    return df_pre
```

```
def mi_importance(df, nums: list, cats:list, target: str)-> pd.DataFrame:

    # 숫자형 변수를 적절히 구간화 작업 실행
    df_quantile = quantile_transform(df_pre, nums)
    df_quantile = pd.concat([df_quantile, df_pre[cats], df_pre[target]], axis=1)

    # 상호 정보 계산
    mi_importance = calc_mutual_info(df_quantile, df_quantile.columns.tolist()).sort_values(target, ascending=False)
```

```
[target].drop(target)
    mi_importance = pd.DataFrame(mi_importance).reset_index()
    mi_importance.columns = ['Feature', 'Importance']

    return mi_importance

# 적용
mi_importance = mi_importance(df_pre, nums, cats, target)

# 결과 확인
print(f"변수 중요도:\n {mi_importance}")
```

1. 모델 기반 중요도 (Feature Importance)

1.1. 트리 기반 모델의 중요도

- **방법:**
  결정트리, 랜덤 포레스트, Gradient Boosting 등의 트리 기반 모델은 각 분할 시 사용된 변수의 기여도(예: 불순도 감소량)를 계산합니다.
- **분석:**
  모델 학습 후 `feature_importances_` 속성을 확인하여, 변수별로 상대적 중요도를 산출할 수 있습니다.

```
# 랜덤 포레스트 중요도 계산
def rf_importance(df, nums: list, cats: list, target: str, type='C', **kwargs)-> pd.DataFrame:

    from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier

    # 범주형 변수를 원핫인코딩
    df_ohe = pd.get_dummies(df[cats]).astype(int)

    # 수치형 변수와 결합
    df_combined = pd.concat([df[nums], df_ohe, df[target]], axis=1)

    # target 이외의 변수 선택
    features_combined = df_combined.columns.drop(target)

    # 기본 파라미터 설정
    params = {'n_estimators': 10, 'random_state': 42}
    params.update(kwargs)  # kwargs로 전달된 값이 있으면 덮어씌움
```

```
    # 모델 학습
    if type == 'C':
        rf = RandomForestClassifier(**params)
    else:
        rf = RandomForestRegressor(**params)
    rf.fit(df_combined[features_combined], df_combined[target])



    # 변수 중요도 확인
    importances = rf.feature_importances_
    importance_df = pd.DataFrame({'Feature': features_combined, 'Importance': importances})
    importance_df = importance_df.sort_values(by='Importance', ascending=False)

    return importance_df
```

- 시각화

```
# 시각화
def plot_importance(importance_df, feature_col: str ='Feature', importance_col: str='Importance', figsize=(12, 4),
dpi=100)-> None:
    plt.figure(figsize=figsize, dpi=dpi)
    plt.bar(importance_df[feature_col], importance_df[importance_col])
    plt.xticks(rotation=45)
    plt.xlabel('Feature')
    plt.ylabel('Feature Importance')
    plt.title('랜덤 포레스트 변수 중요도')
    # 위/아래 여백을 줌
    plt.subplots_adjust(top=0.9)
    plt.subplots_adjust(bottom=0.2)
```

```
    # 레이블이 겹치지 않도록 조정
    plt.tight_layout()
    plt.show(block=False)
```

전처리 파이프라인 구성하기

시나리오 1: 범주형 변수는 최빈값, 숫자형 변수는 중간값으로 치환 후 숫자형 변수는 표준화 변환, 범주형 변수는 원핫인코딩을 하는 파이프라인 구성하기

- 데이터 불러오기
- 변수 정의
- simple_imputer 만들기
- FunctionTransformer 를 이용한 결측값 대체. 나머지 변수는 통과
- ColumnTransformer 를 이용한 표준화, 원핫인코딩 실시
- Pipeline를 이용한 전체 파이프라인 구성
- 데이터 프레임 구성
- 데이터 저장
- df_test 에 적용해보기

```
# 데이터 불러오기
df = joblib.load(os.path.join(data_path, "df_train.pkl"))
df.reset_index(drop=True, inplace=True)

# 메타 정보
df.info()

# 변수 정의
target = 'BAD'
features = df.columns.drop(target)
cats = df[features].select_dtypes(include=['object']).columns
```

```
nums = df[features].select_dtypes(include=['number']).columns


# 1. 결측값 처리를 위한 FunctionTransformer 생성
imputer_transformer = FunctionTransformer(
    simple_imputer,
    kw_args={'target': target, 'nums': nums, 'cats': cats},
    validate=False
)

# 2. 숫자형 변수에 대해 StandardScaler를 적용하고,
#    범주형 변수는 OneHotEncoder로 처리하며, target 변수는 그대로 전달하도록 ColumnTransformer 구성
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), nums),
        ('cat', OneHotEncoder(sparse=False, handle_unknown='ignore'), cats)
    ],
    remainder='passthrough'  # target 변수 (및 nums, cats에 포함되지 않은 변수)는 그대로 전달
)

# 3. 전체 파이프라인 구성:
#    - 먼저 simple_imputer 함수로 결측값 처리를 수행한 후,
#    - ColumnTransformer를 통해 숫자형 표준화 및 범주형 변수 원핫인코딩 적용
pipeline = Pipeline(steps=[
    ('imputation', imputer_transformer),
    ('preprocessing', preprocessor)
])

# 파이프라인 적용: 최종 결과는 numpy 배열로 반환됨
df_processed_array = pipeline.fit_transform(df)
```

```python
# 결과 확인을 위해, 최종 DataFrame의 컬럼명은 ColumnTransformer에서 생성된 결과 순서를 파악하여 재구성합니다.
# 숫자형 변수의 컬럼명는 그대로 사용되고,
# 범주형 변수의 경우 OneHotEncoder가 내부적으로 생성한 이름은 get_feature_names_out()를 활용할 수 있습니다.
# 그리고 remainder='passthrough' 옵션으로 전달된 target 변수는 마지막에 위치할 것입니다.

# 먼저, 추출된 숫자형 컬럼 이름은 그대로 nums 리스트를 사용합니다.
num_cols = nums.tolist()

# 범주형 컬럼 이름은 OneHotEncoder의 get_feature_names_out() 함수를 이용합니다.
# preprocessor.named_transformers_['cat']는 학습된 OneHotEncoder 객체입니다.
cat_cols = list(preprocessor.named_transformers_['cat'].get_feature_names_out(cats))

# remainder 부분에 해당하는 컬럼: 여기서는 target 변수만 남았다고 가정
remainder_cols = [target]

# 최종 컬럼 리스트 (ColumnTransformer의 transformers 순서는: num, cat, 그리고 remainder)
result_columns = num_cols + cat_cols + remainder_cols

# 최종 결과 DataFrame 생성
df_processed = pd.DataFrame(df_processed_array, columns=result_columns)

print("원본 데이터:")
print(df)
print("\n전처리 후 데이터:")
print(df_processed)
```

#실습

시나리오 2: 시나리오 1에 이어서 랜덤포레스트 모델을 적용해보기