MYONGJI UNIVERSITY

# 머신러닝 프로젝트 처음부터 끝까지
## - Part2 -

# 4. 머신러닝 알고리즘을 위한 데이터 준비

# 데이터 준비 과정 자동화

- 머신러닝 모델 재훈련이 필요할 때마다 호출해서 사용할 수 있게끔 데이터를 준비하는 일련의 과정을 함수로 구현하여 자동화하는 것이 바람직함

- 데이터 준비 과정 자동화의 장점
  - 향후 **업데이트 된 데이터셋에 대해서도 동일한 데이터 변환(transformation) 과정을 손쉽게 반복 가능**
  - **데이터 변환 과정 수행을 위한 함수들의 라이브러리 구축**하여 향후 다른 프로젝트에서 재사용
  - 실제 라이브 시스템에서 새로운 데이터를 머신러닝 알고리즘에 주입하기 전에 데이터 변환 과정 수행을 위해 이 함수를 사용할 수 있음
  - 가능한 **여러 데이터 변환 방법을 시도**해봄으로써 **어떤 조합이 가장 적합한지 확인이 용이함**

# 훈련셋 준비

- 예측에 활용할 9개 입력특성들(predictors)과 타깃 특성(label)을 분리하여 사본 생성

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

타겟 특성인 medain_house_value만 제외해서 따로 취함.

  - housing 에는 median_house_value 열이 포함되지 않음
  - housing_labels 에는 median_house_value 열의 값들만 포함
  - 원본 strat_train_set 에 대해서는 변경 없음

# 데이터 전처리(Data Preprocessing)

- 주어진 데이터셋을 머신러닝 알고리즘에서 사용할 수 있도록 하기 위해 필요한 변환 작업을 수행하는 것을 의미

- 수치형 특성에 대한 전처리 과정
  - 데이터 정제(data cleaning): e.g., 결측치 처리, 이상치 및 노이즈 데이터 제거
  - 주어진 기존 특성 조합을 통해 새로운 특성 추가
  - 특성 스케일링

- 범주형 특성에 대한 전처리 과정
  - 원-핫 인코딩(one-hot encoding)

```
housing["ocean_proximity"].value_counts()
```

```
<1H OCEAN      9136
INLAND         6551
NEAR OCEAN     2658
NEAR BAY       2290
ISLAND            5
Name: ocean_proximity, dtype: int64
```

# 데이터 정제(Data Cleaning)

- **결측치 처리, 이상치 및 노이즈 데이터 제거** 등의 작업을 포함
- 캘리포니아 주택 가격 데이터셋의 구역별 **'침실 총 개수'(total_bedrooms) 특성에 결측치 존재**

| | longitude | latitude | housing_median_age | total_rooms | total_bedrooms | population | households | median_income | ocean_proximity |
|---|---|---|---|---|---|---|---|---|---|
| **14452** | -120.67 | 40.50 | 15.0 | 5343.0 | NaN | 2503.0 | 902.0 | 3.5962 | INLAND |
| **18217** | -117.96 | 34.03 | 35.0 | 2093.0 | NaN | 1755.0 | 403.0 | 3.4115 | <1H OCEAN |
| **11889** | -118.05 | 34.04 | 33.0 | 1348.0 | NaN | 1098.0 | 257.0 | 4.2917 | <1H OCEAN |
| **20325** | -118.88 | 34.17 | 15.0 | 4260.0 | NaN | 1701.0 | 669.0 | 5.1033 | <1H OCEAN |
| **14360** | -117.87 | 33.62 | 8.0 | 1266.0 | NaN | 375.0 | 183.0 | 9.8020 | <1H OCEAN |

- **머신러닝 알고리즘은 결측치가 포함된 데이터를 다루지 못하므로** 이에 대한 처리 반드시 필요

# 데이터 정제(Data Cleaning)

```
housing.dropna(subset=["total_bedrooms"], inplace=True)  # option 1

housing.drop("total_bedrooms", axis=1)  # option 2

median = housing["total_bedrooms"].median()  # option 3
housing["total_bedrooms"].fillna(median, inplace=True)
```

- 결측치 처리 방법
  - 옵션1: 해당 구역 샘플들 제거
  - 옵션2: 결측치가 존재하는 특성(e.g., total_bedrooms)의 column 전체를 삭제
  - 옵션3: 해당 특성의 전체 값들의 평균값(average), 중간값(median), 0 등의 특정 값으로 결측치 채우기

# 데이터 정제(Data Cleaning)

```
median = housing["total_bedrooms"].median()  # option 3
housing["total_bedrooms"].fillna(median, inplace=True)
```

- 여기서는 옵션3을 이용하여 total_bedrooms 특성의 결측치를 중간값(median)으로 채움

| | longitude | latitude | housing_median_age | total_rooms | total_bedrooms | population | households | median_income | ocean_pro |
|---|---|---|---|---|---|---|---|---|---|
| **14452** | -120.67 | 40.50 | 15.0 | 5343.0 | 434.0 | 2503.0 | 902.0 | 3.5962 | |
| **18217** | -117.96 | 34.03 | 35.0 | 2093.0 | 434.0 | 1755.0 | 403.0 | 3.4115 | <1H |
| **11889** | -118.05 | 34.04 | 33.0 | 1348.0 | 434.0 | 1098.0 | 257.0 | 4.2917 | <1H |
| **20325** | -118.88 | 34.17 | 15.0 | 4260.0 | 434.0 | 1701.0 | 669.0 | 5.1033 | <1H |
| **14360** | -117.87 | 33.62 | 8.0 | 1266.0 | 434.0 | 375.0 | 183.0 | 9.8020 | <1H |

# SimpleImputer를 활용한 결측치 처리

- 옵션3 구현에 이용 가능한 사이킷런 변환기 클래스(Scikit-Learn transformer class)

```
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy="median")
```

  - 각 특성별 결측치를 해당 특성값들의 median으로 채우기 위한 SimpleImputer 객체 생성

```
housing_num = housing.select_dtypes(include=[np.number])
```

  - median strategy는 수치형 특성에 대해서만 사용 가능하므로 텍스트 타입의 범주형 특성인 ocean_proximity를 제거한 데이터 사본(housing_num) 생성

    simpleImputer는 수치형 특성에서만 적용 가능.

# SimpleImputer를 활용한 결측치 처리

- 옵션3 구현에 이용 가능한 사이킷런 변환기 클래스(Scikit-Learn transformer class)

```
imputer.fit(housing_num)
```

```
SimpleImputer(strategy='median')
```

```
imputer.statistics_
```

```
array([-118.51  ,   34.26  ,   29.    , 2125.    ,  434.    , 1167.    ,
        408.    ,    3.5385])
```

  - imputer.fit(..) 함수 실행을 통해 housing_num 내 각 특성별 median 값이 계산되며, statistics_ 인스턴스 변수에 저장됨

# SimpleImputer를 활용한 결측치 처리

Colab 확인

- 옵션3 구현에 이용 가능한 사이킷런 변환기 클래스(Scikit-Learn transformer class)

```
X = imputer.transform(housing_num)
```

  - imputer.transform(..) 함수 실행을 통해 각 특성별 median 값을 학습한 imputer가 housing_num 내 결측치를 해당 특성의 median 값으로 채우게 됨

- SimpleImputer의 가능한 strategy
  - strategy="mean": 각 특성별 mean 값으로 결측치를 채움
    - 수치형 특성에 대해서만 적용 가능
  - strategy="most_frequent": 각 특성별 most freq 값으로 결측치를 채움
  - strategy="constant", fill_value=..: fill_value에 명시된 특정 값으로 결측치를 채움
    - 수치형 특성 외에도 적용 가능

# 사이킷런(Scikit-Learn)에서 제공하는 3가지 클래스(객체) 유형

# Estimator(추정기)

- 주어진 데이터셋을 기반으로 모델 파라미터들을 추정(estimate)하는 객체를 estimator 라고 칭함
  - E.g., 앞서 사용했던 SimpleImputer 객체가 estimator에 해당함
  - 주어진 housing_num 데이터셋에 대해 각 특성별 median 값을 추정
  - 결과적으로 각 특성 별 median 값이 데이터셋으로부터 SimpleImputer가 학습한 파라미터 값에 해당함
- estimator 객체의 fit(..) 함수 호출을 통해 파라미터 추정 작업이 실행됨
  - E.g., imputer.fit(housing_num)
- (estimator의 모델 파라미터가 아닌) 추정 과정이 어떻게 실행되어야 하는지를 가이드하기 위한 파라미터들은 하이퍼파라미터(hyperparameter)라고 칭함
  - E.g., SimpleImputer 객체 생성 시 설정한 strategy가 하이퍼파리미터에 해당함
- 하이퍼파라미터는 estimator 객체의 인스턴스 변수로 저장됨

```
imputer.strategy
```

```
'median'
```

# Transformer(변환기)

- estimator 객체들 중 일부는 주어진 데이터에 대한 변환 작업도 수행할 수 있으며, 이런 것들을 transformer 라고 칭함
  - E.g., 결과적으로 앞서 사용한 SimpleImputer 객체는 estimator이면서 동시에 transformer에도 해당함
- transformer 객체의 transform(..) 함수 호출을 통해 주어진 데이터에 대한 변환 작업이 실행되며, 변환된 데이터셋이 함수 실행 결과로 반환됨
  - E.g., X = imputer.transform(housing_num)
- 데이터 변환 작업은 SimpleImputer 객체의 경우와 같이 학습된 모델 파라미터 (e.g., 특성별 median)에 기반해서 이루어짐
- 모든 transformer는 fit()과 transform()이 순차적으로 호출되도록 하는 fit_transform() 함수 또한 제공함
  - Fit_transform()이 최적화가 적용되서 더 효율적임

# Predictor(예측기)

- 일부 estimator는 새롭게 주어진 데이터 샘플에 대해 예측을 만들 수 있으며, 이런 것들을 predictor 라고 칭함
  - E.g., 앞서 사용했던 LinearRegression 객체가 predictor에 해당함
  - 어떤 국가의 1인당 GDP 값이 주어지면, 그것을 기반으로 삶의 만족도 값을 예측
    즉 estimator이면서 predictor

- Predictor 객체의 **predict(..)** 함수
  - 입력: 예측하고자 하는 샘플 데이터(e.g., Cyprus의 1인당 GDP)
  - 출력: 입력으로 주어진 샘플에 대해 만들어진 예측(e.g., Cyprus의 삶의 만족도 지수 예측값)

- Predictor 객체의 **score(..)** 함수
  - 주어진 데이터셋(e.g., 테스트셋)을 이용하여 predictor의 예측 성능을 측정하기 위한 함수

# 기타 사이킷런 관련

- Estimator 객체의 하이퍼파라미터는 객체의 public 인스턴스 변수를 통해 직접 접근 가능함
  - E.g., imputer.strategy

- Estimator 객체가 학습한 모델 파라미터도 객체의 public 인스턴스 변수를 통해 직접 접근 가능함
  - E.g., imputer.statistics_ (인스턴스 변수명 마지막에 밑줄 추가됨)

    객체를 학습시켰다면 모델 파라미터 (즉 어떤 데이터가 들어갔을 시라고 보면 될 듯?)

# 범주형 특성(Categorical Features) 다루기

# 범주형 특성 다루기

- 지금까지는 수치형 특성만 다루었음
- 주어진 데이터셋에 텍스트나 범주형 특성 또한 포함될 수 있음
  - E.g., ocean_proximity(해안 근접도) 특성이 범주형 특성에 해당. 특성 값으로 총 5가지 범주(category)가 포함되었음

'<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'

```
>>> housing_cat = housing[["ocean_proximity"]]
>>> housing_cat.head(8)
       ocean_proximity
13096         NEAR BAY
14973        <1H OCEAN
3785            INLAND
14689           INLAND
20507       NEAR OCEAN
1286            INLAND
18078        <1H OCEAN
4396          NEAR BAY
```

- **대부분의 머신러닝 알고리즘은 숫자만을 다룰 수 있음**. 따라서 **ocean_proximity 특성의 텍스트 값을 숫자 값으로 변환**해주어야 함

# OrdinalEncoder를 이용한 단순 수치화

- 사이킷런의 OrdinalEncoder 클래스를 이용하여 ocean_proximity 특성을 다음과 같이 수치화 가능

수치값과 매칭을 해서 수치화로 사용가능

| 범주 | 수치값 |
|---|---|
| <1H OCEAN | 0 |
| INLAND | 1 |
| ISLAND | 2 |
| NEAR BAY | 3 |
| NEAR OCEAN | 4 |

- 문제점
  - **ML 알고리즘은 가까운 두 수치값이 상대적으로 떨어져 있는 두 값보다 더 비슷하다고 간주**
    - E.g., bad, avg, good, excellent와 같이 순서가 있는 범주들의 경우 문제되지 않음
  - 하지만 ocean_proximity 특성의 경우 범주 0(<1H OCEAN)과 1(INLAND) 보다 범주 0과 4(NEAR OCEAN)가 더 의미적으로 비슷함 ➔ **모델 훈련 과정에서 숫자값들의 크기 때문에 잘못된 학습이 이루어질 수 있음** 가까운 값들을 비슷하다 간주하는데 이것때문에 차이가 날 수 있음. (기계는 의미를 해석할 수 없으므로)

# 원-핫 인코딩(One-hot Encoding)

- 범주형 특성을 구성하는 값들의 순서가 의미를 갖지 않는 상황에서 단순 수치화 했을 때 수치값 간의 크기 비교에서 야기되는 문제를 피하기 위한 방안
- 범주형 특성을 범주 수 만큼의 더미(dummy) 특성으로 대체
  - E.g., ocean_proximity 특성에 원-핫 인코딩을 적용하면 5개의 더미 특성으로 대체됨

| | ocean_proximity |
|---|---|
| 13096 | NEAR BAY |
| 14973 | <1H OCEAN |
| 3785 | INLAND |
| 14689 | INLAND |
| 20507 | NEAR OCEAN |
| 1286 | INLAND |
| 18078 | <1H OCEAN |
| 4396 | NEAR BAY |

해당하는 것만 1.0으로 채우고 나머진 0으로 채움.

| | ocean_proximity_<1H OCEAN | ocean_proximity_INLAND | ocean_proximity_ISLAND | ocean_proximity_NEAR BAY | ocean_proximity_NEAR OCEAN |
|---|---|---|---|---|---|
| 13096 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 |
| 14973 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 3785 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 |
| 14689 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 |
| 20507 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 |
| 1286 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 |
| 18078 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 4396 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 |

# 사이킷런의 OneHotEncoder Transformer(변환기)

- OneHotEncoder 변환기(transformer)를 이용하여 범주형 특성에 대한 원-핫 인코딩 가능
- OneHotEncoder 객체 생성 시 sparse 인자 설정을 통해 변환 결과 형태 지정 가능
  - **기본값은 sparse=True** 이며, SciPy sparse matrix 형태로 원-핫 인코딩 결과를 리턴
    - Sparse matrix는 대부분 0으로 채워진 행렬을 효율적으로 표현(행렬 내 소수의 1의 위치만 저장)
    - 굉장히 많은 수의 범주값들로 구성된 범주형 특성을 원-핫 인코딩 할 때 효과적(그만큼 행렬 내 많은 수의 0이 존재할 것이기 때문)

```
from sklearn.preprocessing import OneHotEncoder

cat_encoder = OneHotEncoder()
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
```

```
housing_cat_1hot
```

```
<16512x5 sparse matrix of type '<class 'numpy.float64'>'
        with 16512 stored elements in Compressed Sparse Row format>
```

# 사이킷런의 OneHotEncoder Transformer(변환기)

- OneHotEncoder 변환기를 이용하여 범주형 특성에 대한 원-핫 인코딩 가능
- OneHotEncoder 객체 생성 시 sparse 인자 설정을 통해 변환 결과 형태 지정
  - **sparse=False로 지정**할 경우 **NumPy 이차원 배열 형태로 결과 리턴** 사람이 확인가능한 이차원 배열형태로 전환됨. (False입력시)

```
cat_encoder = OneHotEncoder(sparse=False)
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
housing_cat_1hot
```

```
array([[0., 0., 0., 1., 0.],
       [1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       ...,
       [0., 0., 0., 0., 1.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1.]])
```

# 사이킷런의 OneHotEncoder Transformer(변환기)

Colab 확인

- fit()(또는 fit_transform()) 함수 실행을 통해 OneHotEncoder 객체는 주어진 데이터셋의 범주형 특성에 어떤 범주들이 포함되는지를 학습하게 됨
  - OneHotEncoder 객체의 categories_ public 인스턴스 변수에 학습된 범주들에 대한 정보가 저장됨

```
cat_encoder.categories_
```

```
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

MYONGJI UNIVERSITY

# 특성 스케일링(FEATURE SCALING)

MYONGJI UNIVERSITY

# 특성 스케일링(Feature Scaling)

- **수치형 특성에 대한 전처리 과정임**
- 머신러닝 알고리즘은 입력 데이터셋의 각 특성 값들의 스케일(scale, 범위)가 다르면 제대로 작동하지 않음
  - e.g., total number of rooms 특성 값들의 범위는 6~39320 인데 반해 median_income 특성 값들의 범위는 0~15 로 스케일 차이가 큼 ➔ total number of rooms 특성에 초점을 맞추도록 머신러닝 모델이 편향될 수 있음
- 따라서 모든 특성값들의 스케일을 통일시키는 특성 스케일링(feature scaling) 작업이 반드시 필요

- 다음 두가지 스케일링 방식이 일반적으로 사용됨
  - **Min-max 스케일링**
  - **표준화(standardization)**

- 타깃(레이블) 특성에 대해서는 스케일링 필요 없음

# Min-max 스케일링

- 다음 식을 통해 각 특성 값 x를 변환하는 것이며, 정규화(normalization)라고도 불림
  - $\frac{x-min}{max-min}$ max/min: x가 속한 특성의 값들 중 최대값/최소값
- 사이킷런의 MinMaxScaler 변환기를 이용하여 가능
  - Min-max 스케일링 결과 값의 범위를 0~1이 아닌 다른 범위로 하고자 할 경우 feature_range 하이퍼파라미터를 통해 지정 가능

```
from sklearn.preprocessing import MinMaxScaler

min_max_scaler = MinMaxScaler(feature_range=(-1, 1))
housing_num_min_max_scaled = min_max_scaler.fit_transform(housing_num)
```

- 이상치에 매우 민감
  - E.g., 매우 큰 이상치가 포함된 경우 분자에 비해 분모가 훨씬 크게 되어 min-max 스케일링을 통해 변환된 값들이 0 근처에 몰리게 됨

# 표준화(standardization)

Colab 확인

- 다음 식을 통해 각 특성 값 x를 변환

$$-\frac{x-\mu}{\sigma}$$ $\mu/\sigma$ : x가 속한 특성 값들의 평균/표준편차

- 표준화 스케일링이 적용된 특성 값들의 분포는 평균값 0, 표준편차 1인 표준정규 분포를 따르며, 이상치에 상대적으로 덜 민감함

- 사이킷런의 StandardScaler 변환기를 이용하여 가능

```
from sklearn.preprocessing import StandardScaler

std_scaler = StandardScaler()
housing_num_std_scaled = std_scaler.fit_transform(housing_num)
```

# 변환기 관련 주의사항

- fit(): 오직 훈련셋에 대해서만 호출해야 함(테스트셋에 대해서는 사용해서는 안됨)
- transform(): 테스트셋 포함 모든 데이터셋에 대해 사용 가능
  - fit() 함수 실행을 통해 변환기가 훈련셋을 학습하고 나면 훈련을 마친 변환기에 대한 transform() 함수는 테스트셋에 대해서도 사용 가능함.

- 훈련셋을 이용하여 변환기를 fit 한 다음(훈련셋을 분석하여 특성 값 변환에 필요한 파라미터들을 학습), 그 변환기를 이용하여 전체 데이터셋(훈련셋 + 테스트셋)을 변환

# 사용자 정의 변환기(Custom Transformers)

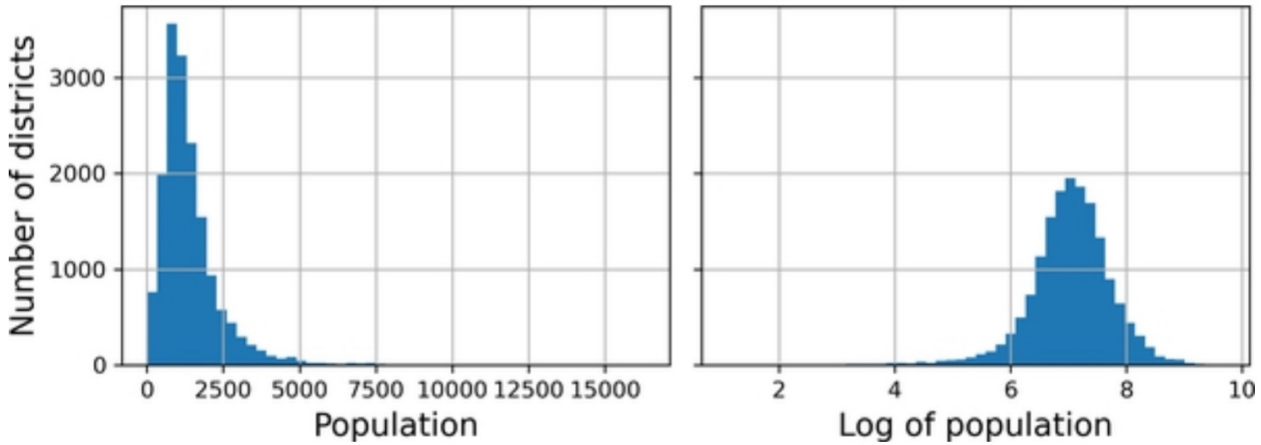
- 필요에 따라서 개발자가 필요한 변환기를 직접 구현할 수 있음

- 방법1: 사이킷런의 **FuctionTransformer 클래스를 이용**하여 커스텀 변환기 생성
  - fit() 함수 우선 호출 없이 transform() 함수를 바로 적용해도 되는 변환기의 경우는 사이킷런의 FuctionTransformer 클래스를 이용하여 간단히 생성 가능

- 방법2: **사용자 정의 변환기 클래스 구현**
  - fit() 함수 우선 실행을 통해 변환에 필요한 정보 학습 후 transform() 함수 호출이 가능한 변환기를 직접 구현하고자 할 경우 변환기 클래스를 직접 구현해야 함

# 방법1 예: 로그 함수 적용 변환

- 특성 값들이 heavy tail 분포를 따르는 경우(아래 왼쪽) 스케일링(e.g., min-max 스케일링)을 적용하기 전에 우선 특성 값들에 로그 함수를 적용하여 좌우 균형 잡힌 분포(아래 오른쪽)로 변환하는 것이 바람직함.




- 이를 위한 커스텀 변환기 생성 방법

```
from sklearn.preprocessing import FunctionTransformer

log_transformer = FunctionTransformer(np.log, inverse_func=np.exp)
log_pop = log_transformer.transform(housing[["population"]])
```

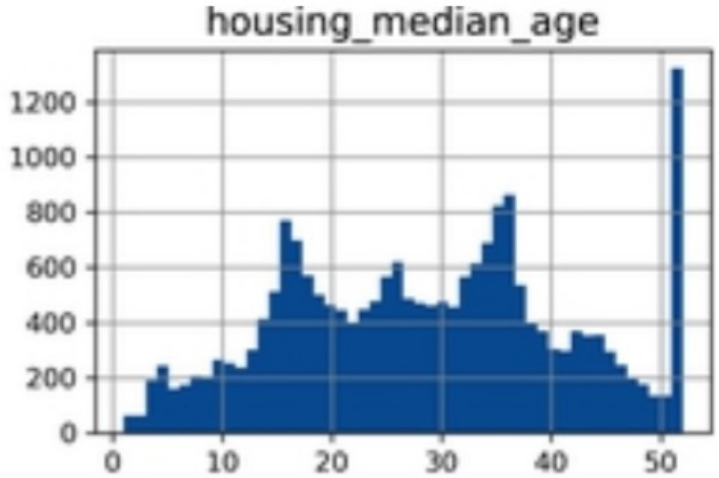
# 방법1 예: 비율 계산 커스텀 변환기

- 두 특성 간 비율을 계산하여 새로운 특성을 생성하는 커스텀 변환기
  - 데이터셋에 대한 학습이 필요치 않은 경우에 해당함

```
>>> ratio_transformer = FunctionTransformer(lambda X: X[:, [0]] / X[:, [1]])
>>> ratio_transformer.transform(np.array([[1., 2.], [3., 4.]]))
array([[0.5 ],
       [0.75]])
```

# 방법1 예: Multimodal 분포를 따르는 특성값 전처리

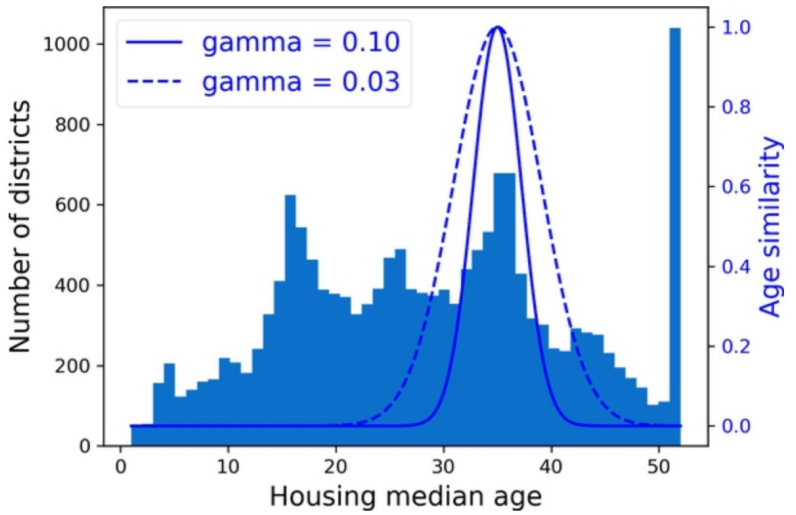
- 특성 값들이 multimodal 분포를 따르는 경우 각 특성 값과 특정 mode 간 유사도를 나타내는 새로운 특성을 추가
  - Multimodal 분포는 아래 그림의 막대 그래프(housing_median_age 특성 값들의 분포)와 같이 봉우리가 여러 개인 분포를 의미하며 각 봉우리를 mode 라고 부름




- 주어진 입력 값과 특정 고정 지점 간의 유사도 측정을 위해 RBF(radial basis function) 함수를 이용함

# 방법1 예: Multimodal 분포를 따르는 특성값 전처리

- **Gaussian RBF 함수: p는 특정 지점**을 가리키며, **입력 값 x가 p에서 조금만 멀어져도 함수 결과값이 급격히 작아짐**
  - $\phi(\boldsymbol{x}, \boldsymbol{p}) = \exp(-\gamma \|\boldsymbol{x} - \boldsymbol{p}\|^2)$
  - **하이퍼파라미터 γ(gamma)**는 x가 p로부터 멀어질 수록 함수 결과값이 얼마나 빠르게 감소되도록 할지를 결정. γ 값이 클 수록 좁은 종 모양의 그래프가 그려지게 됨.




- rbf_kernel()(Guassian RBF 함수)를 이용하여 housing_median_age 특성의 각각의 값과 35(특정 지점) 간의 유사도를 나타내는 새로운 특성 생성

```
from sklearn.metrics.pairwise import rbf_kernel

age_simil_35 = rbf_kernel(housing[["housing_median_age"]], [[35]], gamma=0.1)
```

# 방법1 FunctionTransformer 이용한 변환기 예: 유사도 특성 추가

- rbf_kernel()을 이용하여 **각 구역(district)과 샌프란시스코(sf_coords) 간 지리적 유사도를 나타내는 새로운 특성(sf_simil) 추가**하기 위한 FunctionTransformer 객체 생성 예

```
sf_coords = 37.7749, -122.41
sf_transformer = FunctionTransformer(rbf_kernel,
                                     kw_args=dict(Y=[sf_coords], gamma=0.1))
sf_simil = sf_transformer.transform(housing[["latitude", "longitude"]])
```

# 방법2: 사용자 정의 변환기 클래스

- 결측치를 채우기 위한 SimpleImputer 변환기 경우처럼 fit() 함수 우선 실행을 통해 각 특성 별 mean, median 등 학습 후 transform() 함수 적용이 가능한 변환기를 직접 구현하고자 할 경우 변환기 클래스를 직접 구현해야 함.

- 사이킷런의 다른 변환기와의 호환을 위해 fit(), transform() 등 메서드 구현 필요

# 방법2: 사용자 정의 변환기 클래스 예

- 캘리포니아 주택 가격 데이터에서 지리적으로 서로 근접한 샘플들의 클러스터를 확인 후
- 각 샘플(district)과 n_clusters개 클러스터 각각의 center 와의 유사도 특성을 추가하기 위한 커스텀 변환기 클래스
  - **fit() 함수**에서 **K-means 클러스터링 알고리즘을 이용하여 훈련셋 내 근접한 구역들의 군집을 알아내며**, 총 몇개의 군집들로 나눌 것인지는 **하이퍼파라미터 n_clusters**를 통해 지정
  - **transform() 함수**에서 **rbf_kernel() 이용하여 주어진 샘플 X와 각 군집 center와의 유사도를 측정하여 리턴**. 결과적으로 **샘플 X에 대해 n_clusters개 유사도 특성이 추가됨**

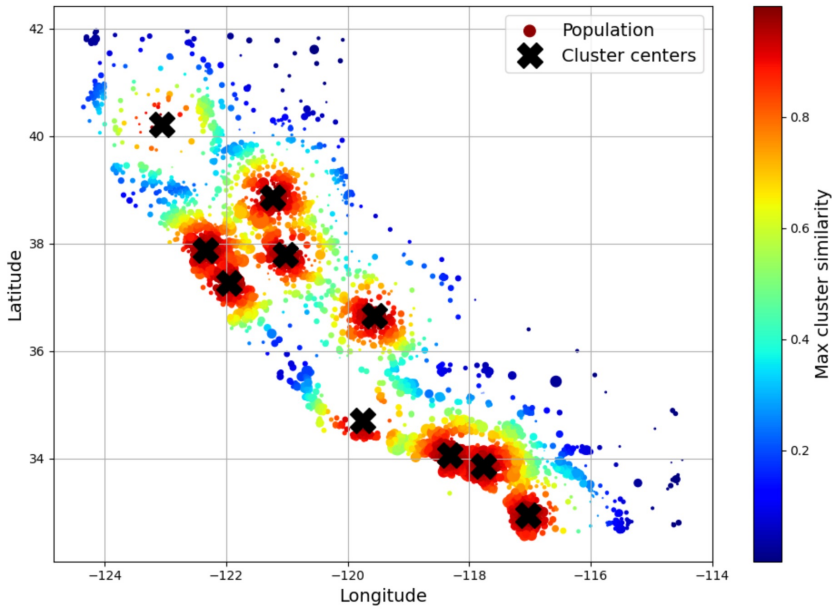
```python
from sklearn.cluster import KMeans

class ClusterSimilarity(BaseEstimator, TransformerMixin):
    def __init__(self, n_clusters=10, gamma=1.0, random_state=None):
        self.n_clusters = n_clusters
        self.gamma = gamma
        self.random_state = random_state

    def fit(self, X, y=None, sample_weight=None):
        self.kmeans_ = KMeans(self.n_clusters, random_state=self.random_state)
        self.kmeans_.fit(X, sample_weight=sample_weight)
        return self  # always return self!

    def transform(self, X):
        return rbf_kernel(X, self.kmeans_.cluster_centers_, gamma=self.gamma)

    def get_feature_names_out(self, names=None):
        return [f"Cluster {i} similarity" for i in range(self.n_clusters)]
```



# 사용자 정의 변환기 클래스 예

Colab 확인

- ClusterSimilarity 커스텀 변환기 클래스를 이용하여 housing 데이터셋의 모든 샘플들에 대해서 n_clusters개 유사도 특성을 생성하는 코드 예

```
cluster_simil = ClusterSimilarity(n_clusters=10, gamma=1., random_state=42)
similarities = cluster_simil.fit_transform(housing[["latitude", "longitude"]],
                                           sample_weight=housing_labels)
```

```
from sklearn.cluster import KMeans

class ClusterSimilarity(BaseEstimator, TransformerMixin):
    def __init__(self, n_clusters=10, gamma=1.0, random_state=None):
        self.n_clusters = n_clusters
        self.gamma = gamma
        self.random_state = random_state

    def fit(self, X, y=None, sample_weight=None):
        self.kmeans_ = KMeans(self.n_clusters, random_state=self.random_state)
        self.kmeans_.fit(X, sample_weight=sample_weight)
        return self  # always return self!

    def transform(self, X):
        return rbf_kernel(X, self.kmeans_.cluster_centers_, gamma=self.gamma)

    def get_feature_names_out(self, names=None):
        return [f"Cluster {i} similarity" for i in range(self.n_clusters)]
```

# 변환 파이프라인(Transformation Pipelines)

- 데이터 전처리 과정은 여러 세부 변환 단계들로 이루어지며 올바른 순서로 진행되어야 함

- 사이킷런의 Pipeline 클래스를 이용하여 여러 변환기들에 의한 일련의 데이터 변환 단계들이 순서대로 실행되도록 하는 데이터 변환 파이프라인 생성 가능

# 수치형 특성에 대한 변환 파이프라인 예

```
from sklearn.pipeline import Pipeline

num_pipeline = Pipeline([
    ("impute", SimpleImputer(strategy="median")),
    ("standardize", StandardScaler()),
])
```

- Pipeline 객체 생성 시 각 데이터 변환 단계을 위해 적용할 객체 및 식별 이름 쌍들의 리스트를 입력으로 받는다.
  - E.g., num_pipeline은 "impute" 변환기와 "standardize" 변환기로 이루어진 변환 파이프라인
- 파이프라인의 마지막 객체를 제외한 모든 객체는 fit_transform()을 지원하는 변환기(transformer) 객체이어야 함
  - 파이프라인의 마지막 객체로는 transformer, predictor, or estimator 객체 모두 가능함
- 정의된 파이프라인이 estimator, transformer, predictor 중 어느 유형에 해당하는지는 파이프라인의 마지막 객체 유형에 의해 결정됨
  - e.g., num_pipeline의 마지막 객체가 StandardScaler() 변환기이므로 num_pipeline도 변환기에 해당한다.
  - 따라서 num_pipeline.transform(), num_pipeline.fit_transform() & num_pipeline.fit() 모두 가능함

# 수치형 특성에 대한 변환 파이프라인 예

```
from sklearn.pipeline import Pipeline

num_pipeline = Pipeline([
    ("impute", SimpleImputer(strategy="median")),
    ("standardize", StandardScaler()),
])
```

- num_pipeline.fit() 호출 시
  - 파이프라인의 마지막 객체 이전 각 변환기에 대해서 fit_transform() 함수가 순서대로 호출됨
  - 파이프라인의 마지막 객체에 대해서는 대해서는 fit() 함수가 최종 호출됨

- num_pipeline.fit_transform() 호출 시
  - 파이프라인의 마지막 객체를 포함된 모든 변환기 각각에 대해서 fit_transform() 함수가 순서대로 호출됨

# 수치형 특성에 대한 변환 파이프라인 예

```
from sklearn.pipeline import make_pipeline

num_pipeline = make_pipeline(SimpleImputer(strategy="median"), StandardScaler())
```

- 파이프라인에 포함되는 객체의 식별 이름이 필요치 않다면 make_pipeline() 함수를 이용하여 pipeline 객체 생성 가능

- 파이프라인을 구성하는 각 객체의 이름은 자동 부여됨
  - SimpleImputer(..) 변환기의 이름은 “simpleimputer”로 자동 부여됨
  - StandardScaler() 변환기의 이름은 “standardscaler”로 자동 부여됨

# ColumnTransformer 클래스

- ColumnTransformer 클래스를 이용하여 특성 별로 다른 전처리 과정을 지정할 수 있다.
- 수치형 특성과 범주형 특성을 구분해서 서로 다른 전처리 과정이 적용되도록 하는 통합 파이프라인 구성 가능
  - 수치형 특성들에 대해서는 앞서 이미 정의한 num_pipeline 변환기 적용
  - 범주형 특성에 대해서는 cat_pipeline(SimpleImputer와 OneHotEncoder 순으로 구성) 변환기 적용

```
num_attribs = ["longitude", "latitude", "housing_median_age", "total_rooms",
               "total_bedrooms", "population", "households", "median_income"]
cat_attribs = ["ocean_proximity"]

cat_pipeline = make_pipeline(
    SimpleImputer(strategy="most_frequent"),
    OneHotEncoder(handle_unknown="ignore"))

preprocessing = ColumnTransformer([
    ("num", num_pipeline, num_attribs),
    ("cat", cat_pipeline, cat_attribs),
])
```

# make_column_selector() 함수

- ColumnTransformer 이용하여 변환 파이프라인 구성 시 각 변환기를 적용할 특성을 일일이 나열하기는 것이 번거로움
- 이때 make_column_selector() 함수를 이용하여 지정된 자료형의 특성들만을 선별 가능

```
preprocessing = ColumnTransformer([
    ("num", num_pipeline, make_column_selector(dtype_include=np.number)),
    ("cat", cat_pipeline, make_column_selector(dtype_include=object)
])
```

  - make_column_selector(dtype_include=np.number) 실행 결과로 np.number 타입의 수치형 특성들에 대한 리스트가 리턴되며, 따라서 수치형 특성들에 대해서 num_pipeline 변환기가 적용됨
  - object 타입의 범주형 특성에 대해서는 cat_pipeline 변환기 적용

# make_column_transformer() 함수

- make_pipeline() 함수와 유사
- ColumnTransformer 파이프라인에 포함되는 각 객체에게 특정 이름을 부여할 필요가 없다면 make_column_transformer() 함수를 이용할 수 있음

```
from sklearn.compose import make_column_selector, make_column_transformer

preprocessing = make_column_transformer(
    (num_pipeline, make_column_selector(dtype_include=np.number)),
    (cat_pipeline, make_column_selector(dtype_include=object)),
)
```

  - num_pipeline에 대해서는 "pipeline-1"로 이름 자동 부여됨
  - cat_pipeline에 대해서는 "pipeline-2"로 이름 자동 부여됨

# make_column_selector() 함수

Colab 확인

- make_column_transformer() 함수를 이용하여 파이프라인 구성 시에도 make_column_selector() 함수 사용이 가능함

```
from sklearn.compose import make_column_selector, make_column_transformer

preprocessing = make_column_transformer(
    (num_pipeline, make_column_selector(dtype_include=np.number)),
    (cat_pipeline, make_column_selector(dtype_include=object)),
)
```

  - np.number 타입의 수치형 특성들에 대해서는 num_pipeline 변환기 적용
  - object 타입의 범주형 특성들에 대해서는 cat_pipeline 변환기 적용

```
preprocessing = ColumnTransformer([
    ("num", num_pipeline, make_column_selector(dtype_include=np.number)),
    ("cat", cat_pipeline, make_column_selector(dtype_include=object)
])
```

# 캘리포니아 주택 가격 데이터셋 전처리를 위한 변환 파이프라인 구성

- 캘리포니아 주 주택 가격 데이터셋에 적용할 모든 전처리 과정들을 변환 파이프라인으로 구성해보자.

- 결측치 처리(imputation)
  - 수치형 특성의 결측치는 각 특성 별 median 값으로 채움(i.e., SimpleImputer(strategy="median"))
  - 범주형 특성의 결측치는 most frequent 값으로 채움(i.e., SimpleImputer(strategy="most_frequent"))
- 범주형 특성에 대해 원-핫 인코딩 적용 (i.e., OneHotEncoder(handle_unknown="ignore"))
- 다음 3가지 ratio 특성들을 계산해서 추가
  - bedrooms_ratio(= total_bedrooms/total_rooms)
  - rooms_per_house(= total_rooms/households)
  - people_per_house(= population/households)

# 캘리포니아 주택 가격 데이터셋 전처리를 위한 변환 파이프라인 구성

- 캘리포니아 주 주택 가격 데이터셋에 적용할 다음 전처리 과정들을 변환 파이프라인으로 구성해보자.

- 샘플들(district)의 longitude, latitude 특성 값에 대해서 앞서 구현한 ClusterSimilarity 커스텀 변환기 적용
  - 샘플들 간 지리적 근접도를 기준으로 샘플들을 총 10개 클러스터로 나누고, 각 클러스터 별 center에 해당하는 (longitude, latitude) 값 조합을 찾음(K-means 클러스터링 이용)
  - 샘플 $x^{(i)}$와 10개 클러스터 center 각각과의 유사도에 대한 특성 추가(rbf_kernel() 이용하여 유사도 계산)
  - 결과적으로 기존 longitude, latitude 2개 특성이 cluster0_similarity부터 cluster9_similarity까지 10개의 특성으로 대체되도록 구현

# 캘리포니아 주택 가격 데이터셋 전처리를 위한 변환 파이프라인 구성

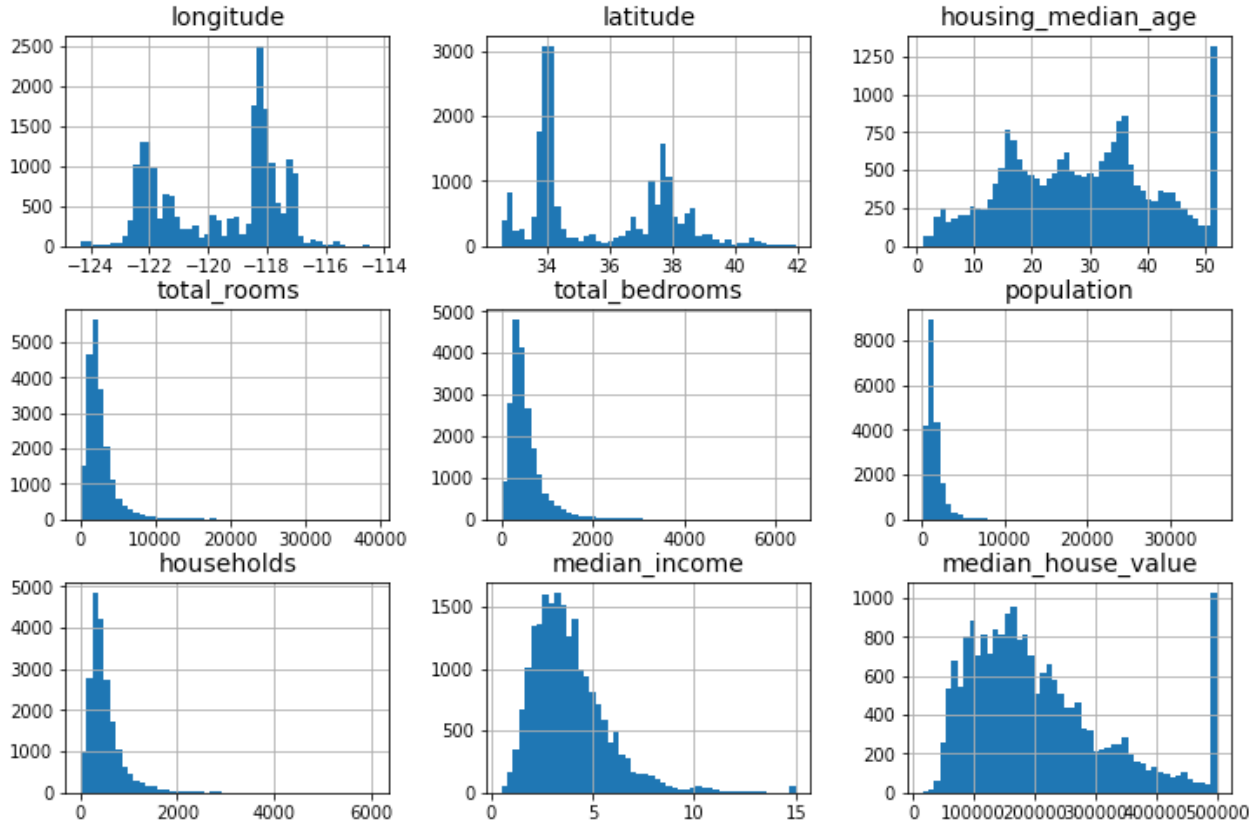
- 캘리포니아 주 주택 가격 데이터셋에 적용할 다음 전처리 과정들을 변환 파이프라인으로 구성해보자.

- Heavy tail 분포를 보이는 다음 5개 특성들에 대해서는 로그 함수 변환 적용 후 StandardScaler()(표준화 스케일링) 적용
  - e.g., total_bedrooms, total_rooms, population, households, median_income

- 그 외 수치형 특성에 대해서는 결측치 처리 후 StandardScaler()만 적용

# 9개 수치형 특성에 대한 히스토그램

```
housing.hist(bins=50, figsize=(12, 8))
# save_fig("attribute_histogram_plots")  # 그림 저장용
plt.show()
```



# 캘리포니아 주택 가격 데이터셋에 대한 변환 파이프라인

```
def column_ratio(X):
    return X[:, [0]] / X[:, [1]]

def ratio_name(function_transformer, feature_names_in):
    return ["ratio"]  # feature names out

def ratio_pipeline():
    return make_pipeline(
        SimpleImputer(strategy="median"),
        FunctionTransformer(column_ratio, feature_names_out=ratio_name),
        StandardScaler())

log_pipeline = make_pipeline(
    SimpleImputer(strategy="median"),
    FunctionTransformer(np.log, feature_names_out="one-to-one"),
    StandardScaler())
cluster_simil = ClusterSimilarity(n_clusters=10, gamma=1., random_state=42)
default_num_pipeline = make_pipeline(SimpleImputer(strategy="median"),
                                     StandardScaler())
preprocessing = ColumnTransformer([
        ("bedrooms", ratio_pipeline(), ["total_bedrooms", "total_rooms"]),
        ("rooms_per_house", ratio_pipeline(), ["total_rooms", "households"]),
        ("people_per_house", ratio_pipeline(), ["population", "households"]),
        ("log", log_pipeline, ["total_bedrooms", "total_rooms", "population",
                               "households", "median_income"]),
        ("geo", cluster_simil, ["latitude", "longitude"]),
        ("cat", cat_pipeline, make_column_selector(dtype_include=object)),
    ],
    remainder=default_num_pipeline)  # one column remaining: housing_median_age
```

# 캘리포니아 주택 가격 데이터셋에 대한 변환 파이프라인

- preprocessing 변환 파이프라인을 이용한 훈련셋 전처리

```
>>> housing_prepared = preprocessing.fit_transform(housing)
>>> housing_prepared.shape
(16512, 24)
>>> preprocessing.get_feature_names_out()
array(['bedrooms__ratio', 'rooms_per_house__ratio',
       'people_per_house__ratio', 'log__total_bedrooms',
       'log__total_rooms', 'log__population', 'log__households',
       'log__median_income', 'geo__Cluster 0 similarity', [...],
       'geo__Cluster 9 similarity', 'cat__ocean_proximity_<1H OCEAN',
       'cat__ocean_proximity_INLAND', 'cat__ocean_proximity_ISLAND',
       'cat__ocean_proximity_NEAR BAY', 'cat__ocean_proximity_NEAR OCEAN',
       'remainder__housing_median_age'], dtype=object)
```

  - original 데이터셋인 housing 데이터프레임의 특성 수는 총 9개 였음

```
[135] preprocessing.feature_names_in_

     array(['longitude', 'latitude', 'housing_median_age', 'total_rooms',
            'total_bedrooms', 'population', 'households', 'median_income',
            'ocean_proximity'], dtype=object)
```

# 캘리포니아 주택 가격 데이터셋에 대한 변환 파이프라인

Colab 확인

- preprocessing 변환 파이프라인을 이용한 훈련셋 전처리

```
>>> housing_prepared = preprocessing.fit_transform(housing)
>>> housing_prepared.shape
(16512, 24)
>>> preprocessing.get_feature_names_out()
array(['bedrooms__ratio', 'rooms_per_house__ratio',
       'people_per_house__ratio', 'log__total_bedrooms',
       'log__total_rooms', 'log__population', 'log__households',
       'log__median_income', 'geo__Cluster 0 similarity', [...],
       'geo__Cluster 9 similarity', 'cat__ocean_proximity_<1H OCEAN',
       'cat__ocean_proximity_INLAND', 'cat__ocean_proximity_ISLAND',
       'cat__ocean_proximity_NEAR BAY', 'cat__ocean_proximity_NEAR OCEAN',
       'remainder__housing_median_age'], dtype=object)
```

  - preprocessing 변환 파이프라인을 통한 전처리 결과
    - 3가지 ratio 특성들이(bedrooms__ratio, rooms_per_house__ratio, people_per_house__ratio) 추가됨
    - longitude & latitude ➔ geo__Cluster 0 ~ Cluster 9 similarity 10개 특성들로 대체
    - ocean_proximity ➔ cat__ocean_proximity_<1H OCEAN ~ NEAR OCEAN 5개 특성들로 대체
    - 따라서 전처리 결과 특성 수 3+10+5+ (5(log__*) + 1(remainder__*))=24개