

# 2장 머신러닝 프로젝트 처음부터 끝까지 (2부)

# 개요

1. 실전 데이터 활용
2. 큰 그림 그리기
3. 데이터 훑어보기
4. 데이터 탐색과 시각화
5. **데이터 준비**
6. 모델 선택과 훈련
7. 모델 미세 조정
8. 최적 모델 저장과 활용



## 2.5. 데이터 준비: 정제와 전처리

## 데이터 정제

- 결측치 처리, 이상치 및 노이즈 데이터 제거
- 캘리포니아 주택가격 데이터셋: 구역별 총 침실 수(`total_bedrooms`) 특성에 결측치 포함됨

# 데이터 전처리

- 범주형 특성 전처리 과정
  - 원-핫-인코딩
- 수치형 특성에 대한 전처리
  - 특성 스케일링
  - 특성 조합

## 캘리포니아 주택가격 데이터셋 대상 전처리

- 비율 특성 추가
  - 침실 비율 특성
  - 가구당 방 수 특성
  - 가구당 평균 가구원수 특성
- 로그 변환: "total\_bedrooms", "total\_rooms", "population", "households", "median\_income"
- 구역 군집 특성 추가: 위도와 경도, 주택 중위가격 정보를 활용하여 유사한 구역끼리 구성된 군집으로 분류
- 해안 근접도 특성에 대한 원-핫-인코딩 적용: 5개의 범주로 구분된 특성을 총 5개의 특성으로 구성된 수치형 데이터로 변환

## 데이터 준비 자동화

- 사이킷런 라이브러리에서 제공하는 API 활용
- 모든 전처리 과정을 **파이프라인**<sub>pipeline</sub>을 이용하여 자동화
- 파이프라인
  - 일반적으로 여러 과정을 한 번에 수행하는 기능을 지원하는 도구를 의미함.
  - 여기서는 사이킷런 API를 순차적으로 적용하는 기능을 지원하는 기능을 가리킴.

## API란?

- Application Programming Interface(응용 프로그래밍 인터페이스)의 줄임말
- 간단하게 말해 응용 프로그램을 가리킴.
- 프로그래밍 분야에서 API: 함수, 클래스, 모듈 등 프로그램 구현에 유용한 도구 총칭
- 여기서는 사이킷런 라이브러리에 포함된 클래스, 메서드, 함수를 가리키는 용어

### 2.5.1. 사이킷런 API

사이킷런의 API는 일반적으로 다음 세 클래스의 인스턴스로 생성됨.

- 추정기
- 변환기
- 예측기

## 추정기(estimator)

- `fit()` 메서드를 지원하는 클래스의 인스턴스
- 일반적으로 변환기와 예측기 둘 중의 하나임.

## 변환기(transformer)

- `fit()` 메서드와 `transform()` 메서드를 함께 지원하는 클래스의 인스턴스
- 일반적으로 데이터 정제와 전처리 과정에서 주로 사용됨.
- `fit()` 메서드: 데이터 변환에 필요한 정보 계산
- `transform()` 메서드: 데이터 변환 실행
- `fit_transform()` 메서드도 함께 지원: `fit()` 메서드와 `transform()` 메서드를 연속으로 호출.

## 예측기(predictor)

- `fit()` 메서드와 `predict()` 메서드를 함께 지원하는 클래스의 인스턴스
- 일반적으로 **모델**이라 불림.
- `fit()` 메서드: 모델의 훈련 관장
- `predict()` 메서드: 모델의 훈련이 종료 된 후 실전에서 예측값을 계산할 활용
- `predict()` 메서드가 예측한 값의 성능을 측정하는 `score()` 메서드도 일반적으로 함께 지원됨.
- 일부 예측기는 예측값의 신뢰도를 평가하는 기능도 함께 제공.

## 2.5.2. SimpleImputer 변환기: 결측치 처리

`total_bedrooms` 특성에 207개 구역에 대한 값이 `NaN` (Not a Number), 즉 결측치로 지정되어 있음.

```
null_rows_idx = housing.isnull().any(axis=1)  
housing.loc>null_rows_idx].head()
```

	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	375.0	183.0	9.8020	<1H OCEAN

## 결측치 처리 방법

- 방법 1: 해당 구역 샘플 제거
- 방법 2: 해당 특성 삭제
- 방법 3: 평균값, 중위수, 최빈값, 0, 또는 주변에 위치한 값 등 특정 값으로 결측치 채우기

## 방법 3 적용: 중위수로 대체

```
from sklearn.impute import SimpleImputer

# 수치형 특성들만 선택
housing_num = housing.select_dtypes(include=[np.number])
imputer = SimpleImputer(strategy="median")
X = imputer.fit_transform(housing_num) ..... # 결과는 np.array 자료형

# 아래 테이블을 보여주기 위해 데이터프레임으로 변환.
housing_tr = pd.DataFrame(X, columns=housing_num.columns,
                           index=housing_num.index)
housing_tr.loc>null_rows_idx].head()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	
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	
11889	-118.05	34.04	33.0	1348.0	434.0	1098.0	257.0	4.2917	
20325	-118.88	34.17	15.0	4260.0	434.0	1701.0	669.0	5.1033	
14360	-117.87	33.62		8.0	1266.0	434.0	375.0	183.0	9.8020

### 2.5.3. 입력 데이터셋과 타깃 데이터셋

- 데이터 전처리를 진행하기 전에 먼저 훈련셋을 다시 입력 데이터셋과 타깃 데이터셋으로 구분
- 이유: 입력 데이터셋과 타깃 데이터셋에 대한 전처리 과정이 일반적으로 다름
- 여기서는 입력 데이터셋에 대해서만 전처리 실행
- 타깃 데이터셋: 일반적으로 전처리 대상이 아니지만 경우에 따라 변환이 요구될 수 있음.
  - 타깃 데이터셋의 두터운 꼬리 분포를 따르는 경우 로그 함수를 적용하여 데이터의 분포가 보다 균형잡히도록 하는 것이 권장됨.

## 타깃 데이터셋

- 모델이 훈련을 통해 최대한 정확하게 예측해야 하는 값으로 구성됨.
- 여기서는 구역별 주택 중위가격을 타깃으로 지정
- 즉, 앞으로 다를 모델은 주어진 구역의 주택 중위가격을 최대한 정확하게 예측하도록 훈련됨.

## 입력 데이터셋

- 타깃으로 지정된 값을 예측하는 데에 필요한 정보로 구성된 데이터셋
- 여기서는 구역별 주택 중위가격을 제외한 나머지 특성들로 구성된 데이터셋
- 따라서 주택 중위가격이 제외된 구역의 다른 정보가 입력되면 해당 구역의 주택 중위가격을 예측하도록 모델이 훈련됨.

## 입력/타깃 데이터셋 구분

- 입력 데이터셋: 주택 중위가격 특성이 제거된 훈련셋

```
housing = strat_train_set.drop("median_house_value", axis=1)
```

- 타깃 데이터셋: 주택 중위가격 특성으로만 구성된 훈련셋

```
housing_labels = strat_train_set["median_house_value"].copy()
```

## 2.5.4. OneHotEncoder 변환기: 범주형 특성 전처리

- 해안 근접도 특성 `ocean_proximity`: 5 개의 범주를 나타내는 문자열을 값으로 사용
- 사이킷런의 머신러닝 모델은 일반적으로 문자열과 같은 텍스트 데이터를 다루지 못 함.

## 정수로의 변환

- 가장 단순한 해결책으로 5 개의 범주를 정수로 변환할 수 있음.
- 주의사항: 모델 학습 과정에서 큰 수가 작은 수보다 더 영향을 줄 수 있음.
- 해안 근접도는 단순히 구분을 위해 사용. 해안에 근접하고 있다 해서 주택 가격이 기본적으로 더 비싸지 않음.

범주	숫자
<1H OCEAN	0
INLAND	1
ISLAND	2
NEAR BAY	3
NEAR OCEAN	4

## 원-핫 인코딩 one-hot encoding

- 수치화된 범주들 사이의 크기 비교를 피하기 위해 더미(dummy) 특성을 추가하여 활용
- 해안 근접도 특성 대신에 다섯 개의 범주 전부를 새로운 특성으로 추가한 후 각각의 특성값을 아래처럼 지정
  - 해당 카테고리의 특성값: 1
  - 나머지 카테고리의 특성값: 0
- 예제: INLAND 특성을 갖는 구역은 길이가 5인 다음 어레이로 특성으로 대체됨.

```
[0.0, 1.0, 0.0, 0.0, 0.0]
```

- 리스트에 포함된 다섯 개 각각의 값은 차례대로 다음 특성에 해당하는 값을 가리킴.

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

# OneHotEncoder 변환기

```
from sklearn.preprocessing import OneHotEncoder  
  
housing_cat = housing[["ocean_proximity"]]  
  
cat_encoder = OneHotEncoder(sparse_output=False)  
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)  
  
housing_cat_onehot = pd.DataFrame(housing_cat_1hot,  
                                   columns=cat_encoder.get_feature_names_out(),  
                                   index=housing_cat.index)
```

housing\_cat\_onehot

	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
...	...	...	...	...	...
14207	1.0	0.0	0.0	0.0	0.0
13105	0.0	1.0	0.0	0.0	0.0
19301	0.0	0.0	0.0	0.0	1.0
19121	1.0	0.0	0.0	0.0	0.0
19888	0.0	0.0	0.0	0.0	1.0

## 2.5.5. MinMaxScaler 와 StandardScaler 변환기: 수치형 특성 스케일링

- 머신러닝 모델은 입력 데이터셋의 특성값들의 스케일 scale이 비슷할 때 보다 잘 훈련됨
- 특성에 따라 다루는 수치형 값의 스케일이 다를 때 통일된 스케일링 scaling 필요
- 아래 두 가지 방식이 일반적으로 사용됨.
  - min-max 스케일링(정규화)
  - 표준화

## 정규화: min-max 스케일링

- 아래 식을 이용하여 모든 특성값  $x$ 를 0에서 1 사이의 값으로 변환.
- $max$  와  $min$  은 각각 해당 특성값들의 최댓값과 최솟값을 가리킴.

$$x \longmapsto \frac{x - \min}{\max - \min}$$

- 주의사항: 이상치가 매우 크면 분모가 매우 커져서 변환된 값이 0 근처에 몰릴 수 있음
- 사이킷런의 `MinMaxScaler` 변환기 활용 가능

```
from sklearn.preprocessing import MinMaxScaler  
  
min_max_scaler = MinMaxScaler(feature_range=(0, 1))  
housing_num_min_max_scaled = min_max_scaler.fit_transform(housing_num)
```

## 표준화

standardization

- 아래식을 이용하여 특성값  $x$ 를 변환함.  $\mu$  와  $\sigma$  는 각각 해당 특성값들의 평균값과 표준편차를 가리킴.

$$x \longmapsto \frac{x - \mu}{\sigma}$$

- 변환된 데이터들이 **표준정규분포**에 가까워 지며, 이상치에 상대적으로 영향을 덜 받음.
- 사이킷런의 `StandardScaler` 변환기 활용 가능

```
from sklearn.preprocessing import StandardScaler  
  
std_scaler = StandardScaler()  
housing_num_std_scaled = std_scaler.fit_transform(housing_num)
```

## 2.5.6. FunctionTransformer 변환기

- `MinMaxScaler` 또는 `Standardscaler` 클래스의 `fit()` 메서드 활용
  - min-max 스케일링을 위해 먼저 각 특성의 최댓값과 최솟값 확인
  - 표준화를 위해 먼저 각 특성의 평균값과 표준편차 확인
- 반면에 경우에 따라 어떤 정보도 필요 없이 바로 데이터 변환을 진행할 수도 있음.
  - 로그 변환과 비율 계산
  - `fit()` 메서드를 사용할 필요 없음.
- `fit()` 메서드 호출이 필요 없는 변화기: `FunctionTransformer` 클래스 활용

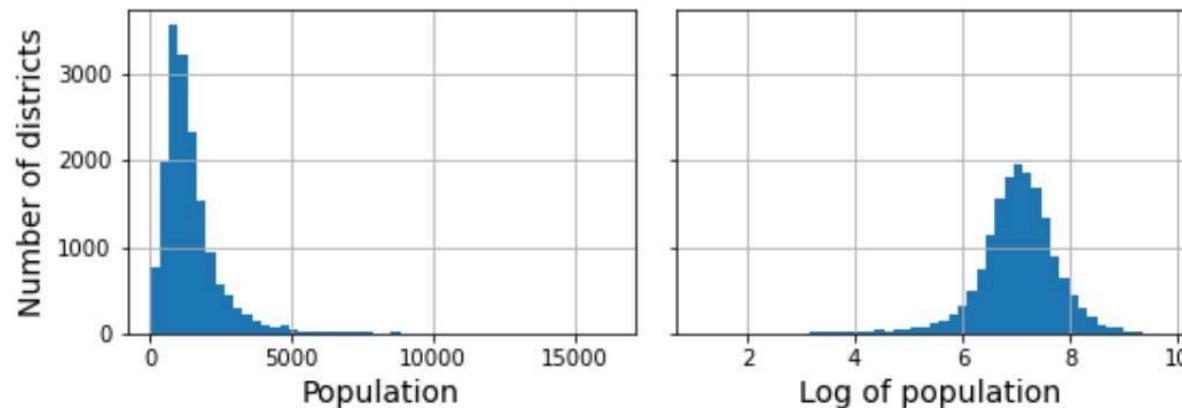
## 로그 변환기

- 데이터셋이 두터운 꼬리 분포를 따르는 경우 적용 추천
- 아래 형식의 로그 변환기 적용

```
FunctionTransformer(np.log)
```

- 적용대상 특성

```
"total_bedrooms", "total_rooms", "population", "households",  
"median_income"
```



## 비율 계산 변환기

- 두 개의 특성 사이의 비율을 계산하여 새로운 특성을 생성하는 변환기

```
FunctionTransformer(lambda X: X[:, [0]] / X[:, [1]])
```

- 비율 계산 변환기를 이용하여 아래 특성을 새로 생성 가능
  - 침실 비율(bedrooms for room): `housing['total_bedrooms'] / housing['total_rooms']`
  - 가구당 방 수(rooms for household): `housing['total_bedrooms'] / housing['households']`
  - 가구당 평균 가구원수(population per household):  
`housing['population'] / housing['households']`

## 2.5.7. 군집 변환기: 사용자 정의 변환기

- 캘리포니아 주 2만 여개의 구역을 서로 가깝게 위치한 구역들로 묶어 총 10개의 군집으로 구분하는 변환기 클래스 선언
- 사이킷런의 다른 변환기와 호환이 되도록 하기 위해 `fit()`, `transform()`,  
`get_feature_names_out()` 선언 필요

```
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.cluster import KMeans
from sklearn.metrics.pairwise import rbf_kernel

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

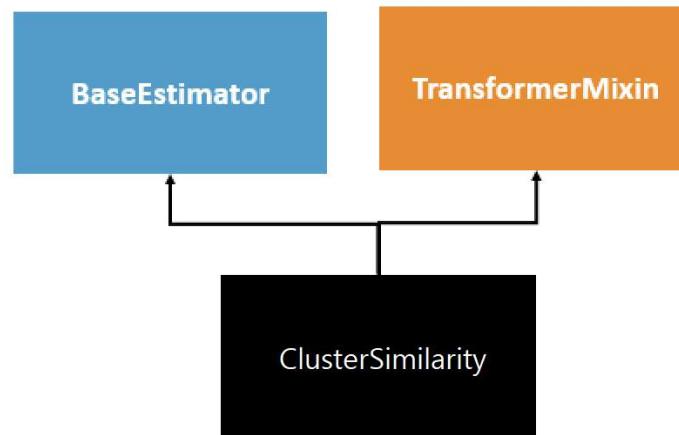
    def fit(self, X, y=None, sample_weight=None):
        self.kmeans_ = KMeans(self.n_clusters, random_state=self.random_state)
        self.kmeans_.fit(X, sample_weight=sample_weight)
        return self # 항상 self 반환

    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 선언 주의사항

- 첫째, `BaseEstimator` 와 `TransformerMixin` 클래스 상속
- 둘째, `get_feature_names_out()` 메서드를 재정의해서 변환기에 의해 새로 생성된 특성들의 이름 지정.



## ClusterSimilarity 변환기 적용 결과

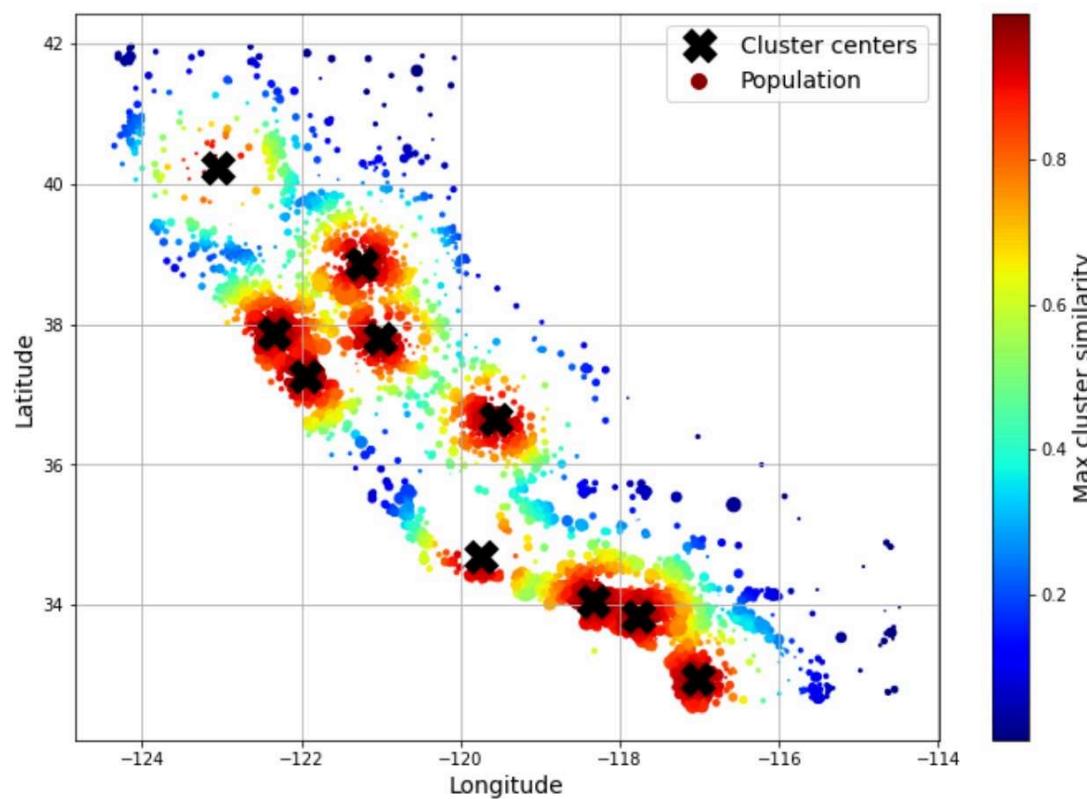
- 모든 구역을 10개의 군집으로 분류.
- `transform()` 메서드: 각 샘플에 대해 10개의 센트로이드와의 유사도 점수를 계산.
- `transform()` 메서드: 각 샘플에 대해 10개의 센트로이드와의 유사도 점수 계산. 유사도 계산을 위해 `fit()` 메서드가 알아낸 10개의 센트로이드 정보 이용.
- `fit()` 메서드: 위도와 경도 정보뿐만 아니라 중위소득 또한 참고. 즉, 중위소득이 비슷한 구역들의 거리가 보다 가까워지도록 계산.

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

```
similarities[:5].round(2)
```

```
array([[0.0, 0.14, 0.0, 0.0, 0.0, 0.08, 0.0, 0.97, 0.0, 0.61],  
       [0.6, 0.0, 0.99, 0.0, 0.0, 0.0, 0.03, 0.0, 0.12, 0.0],  
       [0.0, 0.29, 0.0, 0.0, 0.01, 0.45, 0.0, 0.74, 0.0, 0.31],  
       [0.68, 0.0, 0.21, 0.0, 0.0, 0.0, 0.52, 0.0, 0.0, 0.0],  
       [0.82, 0.0, 0.89, 0.0, 0.0, 0.0, 0.13, 0.0, 0.03, 0.0]])
```

- $\times$  표시: 각 군집의 센트로이드 centroid, 즉 각 군집의 중심 구역을 표시
- 색상: 센트로이드 구역과의 유사도를 가리킴.
- 빨강색의 구역이 센트로이드 구역과의 유사도가 1에 가까움.



## 2.6. 변환 파이프라인

- 모든 전처리 단계가 정확한 순서대로 진행되어야 함
- 사이킷런에서 제공하는 파이프라인 관련 주요 API
  - `Pipeline` 클래스
  - `make_pipeline()` 함수
  - `ColumnTransformer` 클래스
  - `make_column_selector()` 함수
  - `make_column_transformer()` 함수

## 2.6.1. Pipeline 클래스

- 아래 코드: 수치형 특성을 대상으로 결측치를 중위수로 채우는 정제와 표준화를 연속적으로 실행하는 파이프라인
  - Pipeline 인스턴스 생성: 추정기명과 추정기로 이루어진 쌍들의 리스트 이용
  - 마지막 추정기를 제외한 나머지 추정기는 모두 fit\_transform() 메서드를 지원하는 변환기이어야 함.
  - 파이프라인으로 정의된 추정기의 유형은 마지막 추정기의 유형과 동일. 즉, num\_pipeline 은 변환기임.
  - num\_pipeline.fit() 호출
    - 마지막 추정기 이전까지의 변환기: fit\_transform() 메소드 연속 호출
    - 마지막 변환기: fit() 메서드 호출

## make\_pipeline() 함수

- 파이프라인에 포함되는 변환기의 이름이 중요하지 않을 때 사용.
- 아래 코드: 위 파이프라인과 동일한 파이프라인 객체 생성

```
from sklearn.pipeline import make_pipeline  
  
num_pipeline = make_pipeline(SimpleImputer(strategy="median"),  
                             StandardScaler())
```

## 2.6.2. ColumnTransformer 클래스

- 특성별로 파이프라인 변환기 지정
- 아래 코드: 수치형 특성엔 num\_pipeline 변환기를, 범주형 특성엔 OneHotEncoder 변환기를 적용

```
# 수치형 특성 리스트 지정
num_attributes = ["longitude", "latitude", "housing_median_age", "total_rooms",
                  "total_bedrooms", "population", "households", "median_income"]

# 범주형 특성 리스트 지정
cat_attributes = ["ocean_proximity"]

# 범주형 특성 변환 파이프라인
cat_pipeline = make_pipeline(
    SimpleImputer(strategy="most_frequent"),
    OneHotEncoder(handle_unknown="ignore"))

# 전체 특성 변환기
preprocessing = ColumnTransformer([
    ("num", num_pipeline, num_attributes),
    ("cat", cat_pipeline, cat_attributes)])
```

## make\_column\_selector() 함수

- 지정된 자료형을 사용하는 특성들만을 선택해줌
- `make_column_selector(dtype_include=np.number)`: 수치형 특성 모두 선택
- `make_column_selector(dtype_include=object)`: 범주형 특성 모두 선택

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

## `make_column_transformer()` 함수

- `ColumnTransformer` 인스턴스에 포함되는 파이프라인의 이름이 중요하지 않을 때 사용
- 아래 코드: `preprocessing` 변환기를 아래와 같이 정의 가능

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

### 2.6.3. 캘리포니아 데이터셋 변환 파이프라인

- `ColumnTransformer` 클래스와 `Pipeline` 클래스 이용
- 캘리포니아 주택가격 데이터의 입력 데이터셋을 한꺼번에 변환

## (1) 비율 변환기

- 가구당 방 수, 침실 비율, 가구당 평균 가구원수 특성 추가 생성 용도

```
def column_ratio(X):
    ... return X[:, [0]] / X[:, [1]] # 1번 특성에 대한 0번 특성의 비율을

def ratio_name(function_transformer, feature_names_in):
    ... return ["ratio"] # 새로 생성되는 특성값들의 특성명

ratio_pipeline = make_pipeline(
    SimpleImputer(strategy="median"),
    FunctionTransformer(column_ratio, feature_names_out=ratio_name),
    StandardScaler()
)
```

## (2) 로그 변환기

- 데이터 분포가 두터운 꼬리를 갖는 특성을 대상으로 로그 함수를 적용하는 변환기 지정
- `feature_names_out="one-to-one"` : 로그 변환되어 생성되는 특성값들의 특성명을 이전 특성명과 동일하게 지정

```
log_pipeline = make_pipeline(  
    ... SimpleImputer(strategy="median"),  
    ... FunctionTransformer(np.log, feature_names_out="one-to-one"),  
    ... StandardScaler())
```

### (3) 군집 변환기

- 구역의 위도와 경도를 이용하여 구역들의 군집 정보를 새로운 특성으로 추가
- 이전과는 달리 군집 분류를 위해 경도, 위도 정보만 이용

```
cluster_simil = ClusterSimilarity(n_clusters=10, gamma=1., random_state=42)
```

#### (4) 기본 변환기

- 특별한 변환이 필요 없는 경우에도 기본적으로 결측치 문제 해결과 스케일을 조정하는 변환기를 사용

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

## 종합

```
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) # 주택 중위연령 대상
```

## 데이터 정제/전처리 결과

```
housing_prepared = preprocessing.fit_transform(housing)
```

변환된 데이터셋의 특성은 총 24개이며 다음과 같다.

- 비율 변환기 적용: 3개의 새로운 특성 생성.
- 위도와 경도: 10개의 특성으로 변환. 2개의 특성을 빼고 10개 특성 새로 추가.
- 해안근접도: 5개의 특성으로 변환. 1개의 특성을 빼고 10개 특성 추가.
- 나머지 특성은 새로운 특성을 추가로 생성하지는 않음.

## 새로 생성된 24개 특성

```
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 1 similarity', 'geo_Cluster 2 similarity',
       'geo_Cluster 3 similarity', 'geo_Cluster 4 similarity',
       'geo_Cluster 5 similarity', 'geo_Cluster 6 similarity',
       'geo_Cluster 7 similarity', 'geo_Cluster 8 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)
```

## 데이터프레임으로 보기

아래 그림은 24개 중에 6개의 특성만 보여준다.

```
housing_prepared_df = pd.DataFrame(housing_prepared,  
                                     columns=preprocessing.get_feature_names_out(),  
                                     index=housing.index)  
  
housing_prepared_df.head()
```

	bedrooms__ratio	rooms_per_house__ratio	people_per_house__ratio	log__total_bedrooms	log__total_rooms	log__population
13096	1.846624	-0.866027	-0.330204	1.324114	0.637892	0.456906
14973	-0.508121	0.024550	-0.253616	-0.252671	-0.063576	-0.711654
3785	-0.202155	-0.041193	-0.051041	-0.925266	-0.859927	-0.941997
14689	-0.149006	-0.034858	-0.141475	0.952773	0.943475	0.670700
20507	0.963208	-0.666554	-0.306148	1.437622	1.003590	0.719093

5 rows × 24 columns