수치형 데이터 처리

데이터 스케일링이란 데이터 전처리 과정의 하나입니다.

데이터 스케일링을 해주는 이유는 데이터의 값이 너무 크거나 혹은 작은 경우에 모델 알고리즘 학습과정에서 0으로 수렴하거나 무한으로 발산해버릴 수 있기 때문입니다.

따라서, scaling은 데이터 전처리 과정에서 굉장히 중요한 과정입니다.

수치형 데이터 처리

➢ 특성 스케일 변환

- 수치형 특성이 두 값의 범위 안에 놓이도록 변환
- 대부분의 알고리즘은 모든 특성이 동일한 스케일을 가지고 있다고 가정합니다.
- 일반적으로 0~1이나 -1~1 사이입니다.
- MinMaxScaler - 특성의 최솟값과 최댓값을 사용하여 일정 범위 안으로 값을 조정합니다.
  모든 feature가 0과 1사이에 위치하게 만듭니다.
  데이터가 2차원 셋일 경우, 모든 데이터는 x축의 0과 1 사이에, y축의 0과 1사이에 위치하게 됩니다.

$$x'_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

- 사이킷런의 MinMaxScaler는 특성 스케일 fit()는 특성의 최솟값과 최댓값을 계산
- transform()는 특성의 스케일을 조정
- fit_transform()는 두 연산을 한번에 처리

```
import numpy as np
from sklearn import preprocessing

feature = np.array([[-500.5], [-100.1], [0], [100.1], [900.9]])
minmax_scale = preprocessing.MinMaxScaler(feature_range=(0, 1))

scaled_feature = minmax_scale.fit_transform(feature)
scaled_feature
```

수치형 데이터 처리

➢ 특성 표준화 변환

- 특성을 평균이 0이고 표준편차가 1이 되도록 변환 $x'_i = \frac{x_i - \bar{x}}{\sigma}$
- 특성을 표준 정규분포로 근사하는 스케일링 방식
- 변환된 특성은 원본 값이 특성 평균에서 몇 표준편차만큼 떨어져 있는지로 표현(통계학에서는 z-점수)
- 주성분 분석은 표준화가 적합하지만 신경망에는 최소-최대 스케일링을 권장합니다.
- StandardScaler - 각 feature의 평균을 0, 분산을 1로 변경합니다. 모든 특성들이 같은 스케일을 갖게 됩니다.

```
import numpy as np
from sklearn import preprocessing

x= np.array([[-1000.1], [-200.2], [500.5], [600.6], [9000.9]])
scaler = preprocessing.StandardScaler()
standardized = scaler.fit_tranform(x)
Standardized

print("평균:", round(standardized.mean()))
print("표준편차:", standardized.std())
```

수치형 데이터 처리

➢ 특성 스케일 변환

- 데이터에 이상치가 많다면 특성의 평균과 표준편차에 영향을 미치기 때문에 표준화에 부정적인 효과를 끼칩니다.
- 이상치가 많은 경우 중간값과 사분위 범위를 사용하여 특성의 스케일을 조정합니다.
- RobustScaler - 모든 특성들이 같은 크기를 갖는다는 점에서 StandardScaler와 비슷하지만, 평균과 분산 대신 median과 quartile을 사용합니다.
- RobustScaler는 이상치에 영향을 받지 않습니다.
- RobustScaler는 데이터에서 중간값을 빼고 IQR로 나눕니다.

```
robust_scaler = preprocessing.RobustScaler()
robust_scaler.fit_transform(x)
```

- QunatileTransformer : 훈련 데이터를 1,000개의 분위로 나누어 0~1 사이에 고르게 분포시킴으로써 이상치로 인한 영향을 줄입니다.

```
import numpy as np
from sklearn import preprocessing

x= np.array([[-1000.1], [-200.2], [500.5], [600.6], [9000.9]])
preprocessing.QuantileTransformer().fit_transform(x)
```

수치형 데이터 처리

➢ 정규화 변환

- Normalizer 클래스 - 단위 길이의 합이 1이 되도록 개별 샘플의 값을 변환
- 예) 각 단어나 n개의 단어 그룹이 특성인 텍스트 분류와 같이) 유사한 특성이 많을 때 종종 사용
- StandardScaler, RobustScaler, MinMaxScaler가 각 columns의 통계치를 이용한다면 Normalizer는 row마다 각각 정규화됩니다. Normalizer는 유클리드 거리가 1이 되도록 데이터를 조정합니다. (유클리드 거리는 두 점 사이의 거리를 계산할 때 쓰는 방법, L2 Distance)
- 행단위로 변환되므로 fit() 계산 작업 없이 transform() 사용

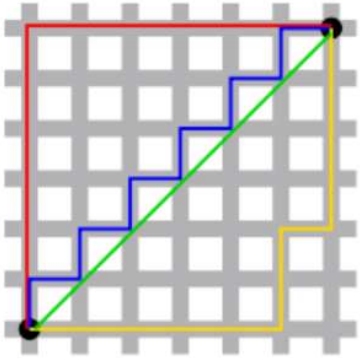
```
import numpy as np
from sklearn.preprocessing import Normalizer

features = np.array([[0.5, 0.5],  [1.1, 3.4],  [1.5, 20.2],  [1.63, 34.4],   [10.9, 3.3]])
normalizer = Normalizer(nor="12")
normalizer.transform(features)
fearues_l2_norm = Normalizer(norm="l2").tranform(features)
fearues_l2_norm
fearues_l1_norm = Normalizer(norm="l1").tranform(features)
fearues_l1_norm
print("첫 번째 샘플값의 합 : ", features_l1_norm[0, 0] + features_l1_norm[0, 1])
features / np.sum(np.abs(features), axis=1, keepdims=True)
features / np.sqrt(np.sum(np.square(features), axis=1, keepdims=True))
```

수치형 데이터 처리

➢ 정규화 변환

- Normalizer의 norm 옵션 L2는 유클리드로 기본값입니다. $\|x\|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$
- 유클리드 거리는 두 점 사이의 거리를 계산할 때 쓰이는 방법으로 녹색과 같은 최단거리이다.
- Normalizer의 norm 옵션 L1은 맨하튼 거리계산(샘플 특성 값의 합을 1로 만듬) 방식입니다
- 빨간색, 노란색, 파란색이 모두 맨하탄 거리(맨하탄의 건물들을 가로지르지 않고 도로로 갔을 경우의 거리)이다.

$\|x\|_1 = \sum_{i=1}^{n} |x_i|$



- Normalizer의 norm 매개변수에 지정 옵션 max : 각 행의 최대값으로 행의 값을 나눔

```
import numpy as np
from sklearn.preprocessing import Normalizer

features = np.array([[0.5, 0.5],  [1.1, 3.4],  [1.5, 20.2],  [1.63, 34.4],   [10.9, 3.3]])
Normalizer(norm="max").transform(features)
```

수치형 데이터 처리

➢ 다항 특성과 교차항 특성 생성

- PolynomialFeatures 클래스는 교차항을 포함합니다.
- degree 매개변수가 다항식의 최대 차수를 결정
  예) degree=2는 2의 제곱까지 새로운 특성을 만듬 x_1, x_2, x_1^2, x_2^2
  degree=3은 2제곱과 3제곱까지 새로운 특성을 만듬 $x_1, x_2, x_1^2, x_2^2, x_1^3, x_2^3$
- interaction_only를 True로 지정하면 교차항 특성만 만들 수 있습니다.
- 특성과 타깃 사이에 비선형 관계가 있다는 가정을 추가할 때 다항 특성을 듭니다.
- 예) 주요 질병에 걸릴 확률에 나이가 미치는 영향은 일정한 상숫값이 아니고 나이가 증가함에 따라 같이 증가한다는 의심을 할 수 있음

```
import numpy as np
from sklearn.preprocessing import PolynomialFeatures

features = np.array([[2, 3],      [2, 3],      [2, 3]])  # 특성 행렬을 만듭니다
# PolynomialFeatures 객체를 만듭니다.
polynomial_interaction = PolynomialFeatures(degree=2, include_bias=False)
polynomial_interaction.fit_transform(features)      # 다항 특성을 만듭니다.

interaction = PolynomialFeatures(degree=2,  interaction_only=True, include_bias=False)
interaction.fit_transform(features)
```

수치형 데이터 처리

➢ 다항 특성과 교차항 특성 생성

- 특성 x에 변동 효과를 주입하기 위해서 고차항 특성을 만들 수 있습니다.
- include_bias 매개변수의 변수값은 True (변환된 특성에 상수항 1을 추가합니다.)

```
# 상수항 1을 추가합니다.
polynomial_bias = PolynomialFeatures(degree=2, include_bias=True).fit(features)
polynomial_bias.transform(features)
polynomial_bias.get_feature_names()
```

수치형 데이터 처리

➢ 특성 변환하기

- 하나 이상의 특성에 사용자 정의 변환을 적용
- FunctionTransformer 를 사용하여 일련의 특성에 어떤 함수를 적용할 수 있습니다.

```
import numpy as np
from sklearn.preprocessing import FunctionTransformer

features = np.array([[2, 3],    [2, 3],    [2, 3]])  # 특성 행렬 데이터 생성

def add_ten(x):    #  함수를 정의합니다.
    return x + 10

ten_transformer = FunctionTransformer(add_ten)  # 변환기 객체 생성
ten_transformer.transform(features)# 특성 행렬을 변환
```

- 판다스의 apply()를 사용하여 동일한 변환을 수행할 수 있다

```
import pandas as pd
df = pd.DataFrame(features, columns=['feature_1', 'feature_2'])
df.apply(add_ten)
```

수치형 데이터 처리

➢ 특성 변환하기

- FunctionTransformer의 validate 매개변수가 True이면 입력값이 2차원 배열인지 확인하고 아닐 경우 예외를 발생시킵니다
- validate가 False이면 일차원 배열에도 적용할 수 있습니다.

```
FunctionTranformer(add_ten, validate=False).transform(np.array([1, 2, 3]))
```

- ColumnTransformer : 특성 배열이나 데이터프레임의 열마다 다른 변환을 적용할 수 있습니다.

```
from sklearn.compose import ColumnTransformer

def add_hundred(x):       # 100을 더하는 함수 정의
    return x + 100

# (이름, 변환기, 열 리스트)로 구성된 튜플의 리스트를 ColumnTransformer에 전달합니다.
ct = ColumnTransformer(
    [("add_ten", FunctionTransformer(add_ten, validate=True), ['feature_1']),
     ("add_hundred", FunctionTransformer(add_hundred, validate=True), ['feature_2'])])
ct.fit_transform(df)
```

수치형 데이터 처리

➢ 이상치 검색

- 일반적인 방법은 데이터가 정규분포를 따른다고 가정하고 이런 가정을 기반으로 데이터를 둘러싼 타원을 그립니다. 타원 안의 샘플을 정상치(레이블 1)로 분류하고, 타원 밖의 샘플은 이상치(레이블 -1)로 분류합니다.
- 이상치의 비율은 contamination 매개변수로 지정 (contamination은 데이터가 얼마나 깨끗한지 추측)
- 데이터에 이상치가 적다면 contamination을 작게 지정, 데이터에 이상치가 많다면 contamination 값을 크게 설정할 수 있습니다.

```
import numpy as np
from sklearn.covariance import EllipticEnvelope
from sklearn.datasets import make_blobs

features, _ = make_blobs(n_samples = 10,  n_features = 2,   centers = 1,  random_state = 1) # 모의 데이터 생성

features[0,0] = 10000  # 첫 번째 샘플을 극단적인 값으로 변경
features[0,1] = 10000

outlier_detector = EllipticEnvelope(contamination=.1)  # 이상치 감지 객체 생성
outlier_detector.fit(features)   # 감지 객체를 훈련시킴
outlier_detector.predict(features)  # 이상치를 예측
```

수치형 데이터 처리

➢ 이상치 검색

- 샘플을 전체적으로 보는 것보다 개별 특성에서 사분위범위(IQR)을 사용하여 극단적인 값을 구별할 수 있습니다.
- IQR은 데이터에 있는 1사분위와 3사분위 사이의 거리입니다.
- 보통 이상치는 1사분위보다 1.5 IQR 이상 작은 값이나 3사분위보다 1.5 IQR 큰 값으로 정의합니다.

```
# 하나의 특성을 만듭니다.
feature = features[:,0]

# 이상치의 인덱스를 반환하는 함수를 만듭니다.
def indicies_of_outliers(x):
    q1, q3 = np.percentile(x, [25, 75])
    iqr = q3 - q1
    lower_bound = q1 - (iqr * 1.5)
    upper_bound = q3 + (iqr * 1.5)
    return np.where((x > upper_bound) | (x < lower_bound))

# 함수를 실행합니다.
indicies_of_outliers(feature)
```

수치형 데이터 처리

➢ 이상치 처리

1. 삭제

```
import pandas as pd

houses = pd.DataFrame()
houses['Price'] = [534433, 392333, 293222, 4322032]
houses['Bathrooms'] = [2, 3.5, 2, 116]
houses['Square_Feet'] = [1500, 2500, 1500, 48000]

houses[houses['Bathrooms'] < 20]  # 샘플을 필터링
```

2. 이상치로 표시하고 이를 특성의 하나로 포함

```
import numpy as np

houses["Outlier"] = np.where(houses["Bathrooms"] < 20, 0, 1)  # 불리언 조건을 기반으로 특성을 생성
houses  # 데이터 확인
```

3. 이상치 처리 - 이상치의 영향이 줄어줄도록 특성을 변환

```
#로그 특성
houses['Log_Of_Square_Feet'] = [np.log(x) for x in houses["Square_Feet"]]
houses
```

수치형 데이터 처리

- **이상치 처리**
  - 이상치는 평균과 분산에 영향을 끼치기 때문에 이상치가 있다면 표준화가 적절하지 않습니다.
  - 이상치 처리 고려할 점

  1. 어떤 것을 이상치로 간주할 것인지?
  2. 이상치를 다루는 방법이 머신러닝의 목적에 맞아야 합니다.

수치형 데이터 처리

➢ 특성 이산화

- 이산화는 수치 특성을 범주형처럼 다루어야 할 때 유용한 전략입니다.
- 수치 특성을 개별적인 구간으로 나눌수 있습니다.
- Binarizer - 임계값에 따라 특성을 둘로 나누는 방법
- numpy.digitize() - 수치 특성을 여러 임계값에 따라 나누는 방법 (bins 매개변수의 입력값은 각 구간의 왼쪽 경계값입니다)

```
import numpy as np
from sklearn.preprocessing import Binarizer

age = np.array([[6], [12],[20],[36], [65]])
binarizer = Binarizer(18)
binarizer.fit_transform(age)

#bins 매개변수의 입력값은 각 구간의 왼쪽 경곗값입니다.
np.digitizer(age, bins=[20, 30, 64])
#right 매개변수를 True로 설정하면 이 동작을 바꿀 수 있습니다.
np.digitize(age, bins=[20, 30, 64], right=True)
np.digitize(age, bins=[18])
```

수치형 데이터 처리

➢ 특성 이산화

- KBinsDiscretizer - 연속적인 특성값을 여러 구간으로 나눔
- encode 매개변수의 기본값은 'onehot'으로 원-핫 인코딩된 희소 행렬을 반환합니다.
- onehot-dense 매개변수 값은 원-핫 인코딩된 밀집 배열을 반환합니다.
- 연속된 값을 이산화하여 원-핫 인코딩으로 만들면 범주형 특성으로 다루기 편리합니다.
- strategy 매개변수의 기본값은 'quantile'로 각 구간에 포함된 샘플 개수가 비슷하도록 만듭니다.
- strategy 매개변수의 기본값은 'uniform'은 구간의 폭이 동일하도록 만듭니다.
- 구간은 bin_edges_속성에서 확인할 수 있습니다.

```
from sklearn.preprocessing import KBinsDiscretizer

kb = KBinsDiscretizer(4, encode='ordinal', strategy='quantile') # 네 개의 구간으로 나눕니다
kb.fit_transform(age)

kb = KBinsDiscretizer(4, encode='onehot-dense', strategy='quantile')  # 원-핫 인코딩으로 반환
kb.fit_transform(age)

kb = KBinsDiscretizer(4, encode='onehot-dense', strategy='uniform') # 동일한 길이의 구간을 만듭니다
kb.fit_transform(age)
kb.bin_edges_
```

수치형 데이터 처리

➢ 군집으로 샘플을 그룹으로 묶기

- k개의 그룹이 있다는 것을 안다면 k-평균 군집(비지도 학습 알고리즘)을 사용하여 비슷한 샘플을 그룹으로 모을 수 있습니다.

```
import pandas as pd
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans

# 모의 특성 행렬 생성
features, _ = make_blobs(n_samples = 50,  n_features = 2,  centers = 3,  random_state = 1)
dataframe = pd.DataFrame(features, columns=["feature_1", "feature_2"])

clusterer = KMeans(3, random_state=0)  # k-평균 군집 모델 생성
clusterer.fit(features)  # 모델 훈련

dataframe["group"] = clusterer.predict(features)  # 그룹 소속을 예측
dataframe.head(5)  # 처음 5 개의 샘플을 조회
```

수치형 데이터 처리

➢ 누락된 값을 가진 데이터 삭제

- 넘파이를 이용하여 누락된 값이 있는 샘플을 삭제

```
import numpy as np

features = np.array([[1.1, 11.1],
                     [2.2, 22.2],
                     [3.3, 33.3],
                     [4.4, 44.4],
                     [np.nan, 55]]) # 특성 행렬을 생성

# (~ 연산자를 사용하여) 누락된 값이 없는 샘플만 남깁니다.
features[~np.isnan(features).any(axis=1)]
```

- 판다스를 사용하여 누락된 값이 있는 샘플을 삭제

```
import pandas as pd

dataframe = pd.DataFrame(features, columns=["feature_1", "feature_2"]) # 데이터 로드
dataframe.dropna()   # 누락된 값이 있는 샘플을 제거
```

수치형 데이터 처리

➢ 누락된 값 채우기

- 데이터의 양이 작으면 k-최근접 이웃(KNN) 알고리즘을 사용해 누락된 값을 예측할 수 있습니다.
- KNN은 누락된 값에 가장 가까운 샘플을 구하기 위해 누락된 값과 모든 샘플 사이의 거리를 계산하므로 작은 데이터셋에서는 수용할 만하지만 데이터넷의 샘플이 수백만 개라면 문제가 됨

```
import numpy as np
from fancyimpute import KNN
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_blobs

features, _ = make_blobs(n_samples = 1000, n_features = 2,  random_state = 1)  # 모의 특성 행렬 생성
scaler = StandardScaler()   # 특성을 표준화
standardized_features = scaler.fit_transform(features)

true_value = standardized_features[0,0]   # 첫 번째 샘플의 첫 번째 특성을 삭제
standardized_features[0,0] = np.nan

# 특성 행렬에 있는 누락된 값을 예측합니다.
features_knn_imputed = KNN(k=5, verbose=0).fit_transform(standardized_features)

print("실제 값:", true_value)   # 실제 값과 대체된 값을 비교
print("대체된 값:", features_knn_imputed[0,0])
```

수치형 데이터 처리

➢ 누락된 값 채우기

- 사이킷런의 Imputer 모듈을 사용하면 특성의 평균, 중간값, 최빈값으로 누락된 값을 채울 수 있습니다. (KNN 보다는 결과가 좋지 않습니다)
- 대용량 데이터셋에서는 누락된 값을 모두 평균값으로 채우는 것을 권장

```
from sklearn.preprocessing import Imputer

mean_imputer = Imputer(strategy="mean", axis=0) # Imputer 객체 생성
features_mean_imputed = mean_imputer.fit_transform(features) # 누락된 값을 채웁니다.

print("실제 값  True Value: ", true_value)
print("대체 값 Imputed Value :", features_mean_imputed[0, 0])
```

- SimpleImputer 는 strategy 매개변수 값 - mean, median, most_frequent, constant

```
from sklearn.impute import SimpleImputer

simple_imputer = SimpleImputer()
features_simple_imputed = simple_imputer.fit_transform(features)

print("실제 값 True Value:", true_value)  # 실제 값과 대체된 값을 비교
print("대체된 값 Imputed Value:", features_simple_imputed[0,0])
```