# 6장 K-최근접 이웃(KNN) : 와인 등급 데이터셋

#### 학습 목표

K-최근접 이웃K Nearest Neighbors, KNN 모델을 구현하여 결과물을 뽑아내고, 작동 원리를 이해합니다. 여기에서는 종속변수가 카테고리값이며, 카테고리값 개수가 3개 이상인 다중분류Multicalssification를 다룹니다.

#### 학습 순서



#### K-최근접 이웃 소개

K-최근접 이웃은 거리 기반 모델입니다. 지금까지 다룬 알고리즘들과는 달리 선형 관계를 전제로 하지 않습니다. 즉 각 데이터 간의 거리를 활용해서 새로운 데이터를 예측하는 모델입니다. 이때 가까이에 있는 데이터를 고려하여 예측값이 결정됩니다. K Nearest Neighbors라는 이름은 이를 잘 반영하고 있는데, K개의 가장 가까운 이웃 데이터에 의해 예측된다는 의미입니다.



#### 장단점

| **장점** | **단점** |
| --- | --- |
| * 수식에 대한 설명이 필요 없을 만큼 직관적이고 간단합니다. * 선형 모델과 다르게 별도의 가정assumption이 없습니다(예를 들어 선형 회귀는 독립변수와 종속변수의 선형 관계를 가정하고 있기 때문에, 이 가정이 들어맞지 않는 데이터에 취약하나, KNN은 이러한 가정이 없어서 더 자유롭습니다). | * 데이터가 커질수록 상당히 느려질 수 있습니다. * 아웃라이어에 취약합니다. |

<용어/>

**아웃라이어)**

평균치에서 크게 벗어나는 데이터를 의미합니다. 이상치라고도 합니다. 예를 들어 한국 축구 선수들의 연봉에 대한 데이터가 있을 때, 평균 연봉은 약 2억이고 스타 플레이어의 연봉이 150억 이상인 경우 스타 플레이어의 연봉을 아웃라이어라고 부릅니다.

</>

#### 유용한 곳

* 주로 분류Classification에서 사용되며, 로지스틱 회귀Logistic Regression로 해결할 수 없는 3개 이상의 목표 변수들도 분류할 수 있습니다.
* 작은 데이터셋에 적합합니다.

#### TOP 10 선정 이유

* 다중분류 문제에서 가장 간편히 적용할 수 있는 알고리즘입니다. 물론 최신 알고리즘들도 다중분류 문제에 사용하나, 데이터가 크지 않고 예측이 까다롭지 않은 상황에서 KNN을 사용하면 신속하고 쉽게 예측 모델을 구현할 수 있습니다. 이러한 특성으로 베이스라인 모델로도 사용합니다.

## 6.1 문제 정의 : 한눈에 보는 분석 목표

<금토끼의 문제 정의> 금토끼는 요새 와인에 빠져 있습니다만, 아직 좋은 와인을 구분할 줄 몰랐습니다. 다양한 와인을 맛보는 것만으로는 한계가 있는 것 같았습니다. ‘좀 더 과학적인 방법이 필요해’라는 생각이 들었습니다. 알코올 도수, 말산, 마그네슘, 색조 같은 정보가 들어있는 와인 데이터를 구해 와인 등급을 예측해보기로 했습니다.

| **난이도** | ⭐☆☆ | | |
| --- | --- | --- | --- |
| **알고리즘** | KNN | | |
| **데이터셋 파일명** | wine.csv | **종속 변수** | class(등급) |
| **데이터셋 소개** | 와인에 대한 데이터입니다. 총 3가지 목푯값으로 이루어진 카테고리형 변수이기 때문에 다중분류multiclassification 문제에 적합합니다. 알코올, 말산, 마그네슘, 색조 등이 독립변수로, 와인 등급인 class를 종속변수로 사용합니다. | | |
| **문제 유형** | 분류 | **평가지표** | 정확도 |
| **사용한 모델** | KNeighborsClassifier | | |
| **사용 라이브러리** | * numpy (numpy==1.19.5) * pandas (pandas==1.3.5) * seaborn (seaborn==0.11.2) * matplotlib (matplotlib==3.2.2) * sklearn (scikit-learn==1.0.2 | | |
| **예제 코드 노트북** | 위치 : <https://github.com/musthave-ML10/notebooks/>  파일 : 06\_KNN.ipynb | | |

## 6.2 라이브러리 및 데이터 불러오기

판다스, 넘파이, 맷플롯립, 시본 라이브러리를 불러오겠습니다. 넘파이는 아직 사용하지 않았고 이 장에서도 사용할 일은 없지만, 이 4가지 라이브러리는 머신러닝에 있어서 필수입니다. 이 장에서부터는 습관적으로 불러오겠습니다.

| import pandas as pd # 판다스 라이브러리 임포트 import numpy as np # 넘파이 라이브러리 임포트 import matplotlib.pyplot as plt # 맷플롭립 라이브러리 임포트 import seaborn as sns # 시본 라이브러리 임포트  file\_url = 'https://media.githubusercontent.com/media/musthave-ML10/data\_source/main/wine.csv' data = pd.read\_csv(file\_url) # 데이터셋 읽기 |
| --- |

## 6.3 데이터 확인하기

우선 head() 함수를 호출해 데이터가 어떻게 생겼는지 확인합니다.

| data.head() # 상위 5행 출력 |
| --- |

|  | **alcohol** | **malic\_acid** | **ash** | **alcalinity\_of\_ash** | **magnesium** | **total\_phenols** | **flavanoids** | **nonflavanoid\_phenols** | **proanthocyanins** | **color\_intensity** | **hue** | **od280/od315\_of\_diluted\_wines** | **proline** | **class** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 14.23 | 1.71 | 2.43 | 15.6 | 127 | 2.80 | 3.06 | 0.28 | 2.29 | 5.64 | 1.04 | 3.92 | 1065 | 0 |
| **1** | NaN | 1.78 | 2.14 | 11.2 | 100 | 2.65 | 2.76 | 0.26 | 1.28 | 4.38 | 1.05 | 3.4 | 1050 | 0 |
| **2** | 13.16 | 2.36 | 2.67 | 18.6 | 101 | 2.80 | 3.24 | 0.3 | 2.81 | 5.68 | 1.03 | 3.17 | 1185 | 0 |
| **3** | 14.37 | 1.95 | 2.50 | 16.8 | 113 | 3.85 | 3.49 | 0.24 | 2.18 | 7.8 | 0.86 | 3.45 | 1480 | 0 |
| **4** | 13.24 | 2.59 | 2.87 | 21.0 | 118 | 2.80 | 2.69 | 0.39 | 1.82 | 4.32 | 1.04 | 2.93 | 735 | 0 |



가장 우측의 class가 목표 변수이며 나머지 12개 변수는 독립변수입니다. 와인의 성분/성질에 대한 정보임을 알 수 있습니다. 여기서는 각 변수에 대한 설명이 크게 중요하지 않아서 생략하고 넘어가겠습니다.

한눈에 봐도 숫자로만 구성된 데이터지만 info() 함수를 사용하여 살펴보겠습니다.

| data.info() # 변수 특징 출력 |
| --- |



모두 숫자로 된 변수이기 때문에 Dtype은 예상대로 int형 아니면 float형입니다. 그리고 전체 데이터 길이는 178로, 상당히 작습니다. 그런데 Non-Null Count값이 다른 변수가 있습니다. alcohol은 178이 아닌 176입니다. 이 변수에 2개의 결측치Missing value가 있다는 의미입니다. nonflavanoid\_phenols는 173이므로 결측치가 5개입니다.

결측치는 쉽게 말해서 값이 비어 있다는 의미입니다. 다음과 같이 엑셀 파일에서 특정셀의 값이 비어 있는 모습을 생각하면 됩니다.



파이썬에서는 이러한 결측치를 Null, na, NaN 등으로 표현합니다. 사용하는 모듈마다 다른 방식을 사용합니다.

마지막으로 통계적 정보를 살펴봅시다.

| data.describe() # 통계적 정보 출력 |
| --- |

|  | **alcohol** | **malic\_acid** | **ash** | **alcalinity\_of\_ash** | **magnesium** | **total\_phenols** | **flavanoids** | **nonflavanoid\_phenols** | **proanthocyanins** | **color\_intensity** | **hue** | **od280/od315\_of\_diluted\_wines** | **proline** | **class** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **count** | 176.000000 | 178.000000 | 178.000000 | 178.000000 | 178.000000 | 178.000000 | 178.000000 | 173 | 178 | 178 | 178 | 178 | 178 | 178 |
| **mean** | 12.989091 | 2.336348 | 2.366517 | 19.494944 | 99.741573 | 2.295112 | 2.029270 | 0.36237 | 1.590899 | 5.05809 | 0.957449 | 2.611685 | 746.893258 | 0.938202 |
| **std** | 0.804431 | 1.117146 | 0.274344 | 3.339564 | 14.282484 | 0.625851 | 0.998859 | 0.126153 | 0.572359 | 2.318286 | 0.228572 | 0.70999 | 314.907474 | 0.775035 |
| **min** | 11.030000 | 0.740000 | 1.360000 | 10.600000 | 70.000000 | 0.980000 | 0.340000 | 0.13 | 0.41 | 1.28 | 0.48 | 1.27 | 278 | 0 |
| **25%** | 12.355000 | 1.602500 | 2.210000 | 17.200000 | 88.000000 | 1.742500 | 1.205000 | 0.26 | 1.25 | 3.22 | 0.7825 | 1.9375 | 500.5 | 0 |
| **50%** | 13.050000 | 1.865000 | 2.360000 | 19.500000 | 98.000000 | 2.355000 | 2.135000 | 0.34 | 1.555 | 4.69 | 0.965 | 2.78 | 673.5 | 1 |
| **75%** | 13.672500 | 3.082500 | 2.557500 | 21.500000 | 107.000000 | 2.800000 | 2.875000 | 0.45 | 1.95 | 6.2 | 1.12 | 3.17 | 985 | 2 |
| **max** | 14.750000 | 5.800000 | 3.230000 | 30.000000 | 162.000000 | 3.880000 | 5.080000 | 0.66 | 3.58 | 13 | 1.71 | 4 | 1680 | 2 |



여기에서도 count를 통해 alcohol과 nonflavanoid\_phenols 변수에 결측치가 있음을 다시 한번 확인해볼 수 있습니다.

여기에서 강조하고 싶은 부분은 크게 두 가지입니다. 첫째는 변수마다 값의 범위가 상당히 다르다는 겁니다. 예를 들어 nonflavanoid\_phenols는 최솟값이 0.13이고 최댓값은 0.66인데, proline은 최솟값이 278이고 최댓값은 1680입니다. 이런 경우를 두고 변수의 스케일이 다르다고 하는데, 이는 거리 기반 알고리즘인 KNN에서는 문제가 될 수 있기 때문에 잠시 후 이 문제를 다루는 스케일링scaling을 할 예정입니다.

<용어/>

**스케일링**

독립 변수의 범위를 동일한 수준으로 만드는 데 사용되는 방법

</>

두 번째는 아웃라이어입니다. 몇몇 변수에서 75%와 max값의 차이가 유독 두드러진 경우가 있습니다. 예를 들어 color\_intensity는 min부터 75%까지는 비교적 고른 패턴으로 증가를 하다가 75%와 max를 비교해보면 6.2에서 13으로 갑자기 2배 이상 증가합니다. 이러한 아웃라이어는 경우에 따라 모델링에 영향을 미칠 수 있으므로 확인해두시는 것이 좋습니다. 지금은 아웃라이어가 있을 수 있다는 점만 명심해두고 넘어가겠습니다.

## 6.4 목푯값에서 고윳값 확인하기

이전 장에서는 목푯값의 값을 별도로 확인하지 않았으나, 이번에는 확인하고 넘어가겠습니다. 목푯값의 특성에 따라, 즉 연속형 변수인지, 0과 1로 이루어진 이진binary 변수인지, 아니면 3개 이상으로 된 카테고리 변수인지에 따라 적합한 알고리즘이 다르기 때문에 필요에 따라 확인하면 됩니다.

예를 들어 연속형 변수는 head() 함수로 출력해 한눈에 파악이 가능하기 때문에 별도의 확인 작업이 필요 없습니다. 명백하게 목표 변수가 0과 1로 이루어졌다는 것을 이미 알고 있는 데이터라면 역시 확인 작업이 불필요합니다.

다만 지금처럼 새로운 데이터를 다루고 있고, 목표 변수가 어떤 값들로 구성되어 있는지 알지 못할 때는 다음과 같은 방법으로 확인할 수 있습니다.

| data['class'].unique() # 목표 변수의 고윳값 출력 |
| --- |

array([0, 1, 2])

data에서 class라는 변수를 인덱싱하여 뒤에 unique() 함수를 사용하면 어떤 값들로 구성된 변수인지를 확인할 수 있습니다. class에는 0, 1, 2라는 고윳값 3가지가 있습니다. 즉 와인을 세 등급으로 나누고 있습니다. 고윳값의 개수를 확인하려는 목적은 이미 달성했지만, 이와 관련된 코드를 몇 가지 더 실습합니다.

| data['class'].nunique() # 고윳값 가짓수 출력 |
| --- |

3

unique는 어떤 값이 있는지 하나하나 보여주지만, nunique는 고윳값 개수를 바로 알려줍니다. 가령 고윳값들이 너무 많아서 unique를 통해 하나하나 세어보기 어려울 때, nunique를 쓰면 쉽게 확인할 수 있습니다.

다음은 각 고윳값이 몇 개씩 있는지 확인하는 코드입니다.

| data['class'].value\_counts() # 각 고윳값에 해당하는 개수 출력 |
| --- |

1 71

0 59

2 48

Name: class, dtype: int64

결괏값은 수가 많은 것부터 순차적으로 나열되는데, 이 데이터에서는 class 1이 71개, class 0이 59개, 그리고 class 2가 48개입니다.

unique, nunique, value\_counts는 서로 비슷한 정보를 제공합니다. 예를 들어 이 데이터에서 value\_counts() 함수를 가장 먼저 확인했다면 고윳값이 3개이고, 각각 몇 개씩인지도 단번에 확인할 수 있습니다. 그러면 굳이 unique()나 nunique() 함수를 또 사용할 필요가 없습니다. 조금 더 구체적이냐, 더 요약된 정보이냐의 차이이기 때문입니다. 목적에 따라 필요한 함수만 사용하면 됩니다.

마지막으로 value\_counts() 함수를 호출해 확인한 값을 막대 그래프Barplot로 그려보겠습니다. 그래프를 그리는 데 시본seaborn의 barplot() 함수를 사용합니다.

| sns.barplot(x = data['class'].value\_counts().index, y = data['class'].value\_counts()) # 막대 그래프 생성 및 출력 |
| --- |

  
코드는 위와 같이 x축, y축에 해당하는 데이터를 각각 지정해주면 됩니다. value\_counts() 함수의 결괏값은 한 줄짜리 데이터인 판다스 시리즈Series 형태입니다. 여기서 class라는 변수의 값(0, 1, 2)을 x축으로, 각 클래스에 해당하는 값의 개수(59, 71, 48)를 y축으로 보내주면 됩니다.

x값을 입력할 때 data['class'].value\_counts() 끝에 .index를 붙였습니다. value\_counts() 함수의 출력이 한 줄짜리 데이터입니다(인덱스와 해당하는 값을 모두 출력합니다).

1 71

0 59

2 48

Name: class, dtype: int64

판다스 시리즈나 데이터 프레임에 이처럼 .index를 사용하면 인덱스만을 따로 뽑아낼 수 있습니다. 정말 그런지 다음과 같이 .index를 추가해보세요.

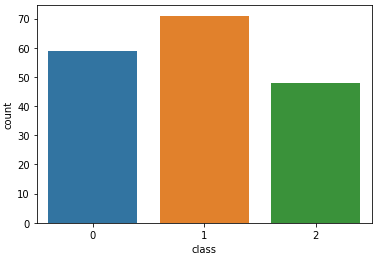
| data['class'].value\_counts().index # 인덱스만 출력 |
| --- |

Int64Index([1, 0, 2], dtype='int64')

단순한 그래프 하나 그리는 데 코드가 복잡해보이죠? 구조만 잘 이해하면 복잡하지 않으니 항상 데이터의 구조를 이해하는 습관을 들이기 바랍니다.

사실 이 그래프는 value\_counts()함수와 상관 없이, sns의 countplot()을 이용하면 더욱 쉽게 구현할 수 있습니다. 괄호안에 원하는 데이터를 입력해주면, 위와 동일한 그래프를 그려줍니다.

| sns.countplot(data['class']) |
| --- |



## 6.5 전처리 : 결측치 처리하기

6.3절 ‘데이터 확인하기’에서 info()와 describe() 함수를 호출해 일부 변수에 결측치가 있다는 사실을 확인했습니다. 이번에는 결측치를 처리하는 방법을 알아봅니다. 그에 앞서 결측치를 확인하는 또다른 방법부터 배워보겠습니다.

### 6.5.1 결측치를 쉽게 확인하는 방법

우선 결측치 유무를 확인하는 가장 직접적인 방법은 isna()[[1]](#footnote-0) 함수를 쓰는 겁니다.

| data.isna() # 값을 결측치 여부에 따라 TRUE/FALSE로 변환 |
| --- |

|  | **alcohol** | **malic\_acid** | **ash** | **alcalinity\_of\_ash** | **magnesium** | **total\_phenols** | **flavanoids** | **nonflavanoid\_phenols** | **proanthocyanins** | **color\_intensity** | **hue** | **od280/od315\_of\_diluted\_wines** | **proline** | **class** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | False | False | False | False | False | False | False | FALSE | FALSE | FALSE | FALSE | FALSE | FALSE | FALSE |
| **1** | True | False | False | False | False | False | False | FALSE | FALSE | FALSE | FALSE | FALSE | FALSE | FALSE |
| **2** | False | False | False | False | False | False | False | FALSE | FALSE | FALSE | FALSE | FALSE | FALSE | FALSE |
| **3** | False | False | False | False | False | False | False | FALSE | FALSE | FALSE | FALSE | FALSE | FALSE | FALSE |
| **4** | False | False | False | False | False | False | False | FALSE | FALSE | FALSE | FALSE | FALSE | FALSE | FALSE |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **173** | False | False | False | False | False | False | False | FALSE | FALSE | FALSE | FALSE | FALSE | FALSE | FALSE |
| **174** | False | False | False | False | False | False | False | FALSE | FALSE | FALSE | FALSE | FALSE | FALSE | FALSE |
| **175** | False | False | False | False | False | False | False | FALSE | FALSE | FALSE | FALSE | FALSE | FALSE | FALSE |
| **176** | False | False | False | False | False | False | False | FALSE | FALSE | FALSE | FALSE | FALSE | FALSE | FALSE |
| **177** | False | False | False | False | False | False | False | FALSE | FALSE | FALSE | FALSE | FALSE | FALSE | FALSE |

178 rows × 14 columns



출력 결과를 보면 알 수 있듯이 각 값이 결측치인지 아닌지를 True/False로 나타내줍니다. alcohol 컬럼의 1라인은 True입니다. 결측치가 들어있다는 의미입니다. head() 함수에서는 결측치를 NaN으로 표시했죠. 이런 식으로 하나하나 확인하는 방식은 비효율적이기 때문에 또다른 함수과 조합을 시도해야 합니다.

우선 판다스에서 컬럼별 합과 평균을 구하는 방법을 알아보고 조합해봅시다. 다음과 같이 sum() 함수를 쓰면 각 컬럼별 합을 보여줍니다.

| data.sum() # 변수별 합 출력 |
| --- |

alcohol 2286.080000

malic\_acid 415.870000

ash 421.240000

alcalinity\_of\_ash 3470.100000

magnesium 17754.000000

total\_phenols 408.530000

flavanoids 361.210000

nonflavanoid\_phenols 62.690000

proanthocyanins 283.180000

color\_intensity 900.339999

hue 170.426000

od280/od315\_of\_diluted\_wines 464.880000

proline 132947.000000

class 167.000000

dtype: float64

isna()와 sum() 함수를 조합하여 결측치를 확인합니다. 다음과 같이 마침표를 사용하여 함수를 붙여나가면 됩니다.

| data.isna().sum() # 값을 TRUE/FALSE로 변환 후 합 |
| --- |

alcohol 2

malic\_acid 0

ash 0

alcalinity\_of\_ash 0

magnesium 0

total\_phenols 0

flavanoids 0

nonflavanoid\_phenols 5

proanthocyanins 0

color\_intensity 0

hue 0

od280/od315\_of\_diluted\_wines 0

proline 0

class 0

dtype: int64

결괏값을 보면 alcohol 컬럼에는 결측치가 2개, nonflavanoid\_phenols 컬럼에는 5개 있네요. 어떻게 이런 결과를 얻을 수 있었던 걸까요? 이 코드를 이해하려면 파이썬에서 True와 False의 특징을 알아야 합니다. True/False은 불리언boolean 자료형입니다. 파이썬에서는 이러한 불리언을 연산할 때 False는 0, True는 1로 자동으로 인식합니다. 그래서 처음 isna() 함수만으로 확인했을 때는 True/False와 같은 형식으로 결괏값이 나타났지만, sum()으로 계산할 때는 자동적으로 0과 1로 인식되어 계산이 된 겁니다.



mean() 함수는 컬럼별 평균값을 보여줍니다.

| data.mean() # 평균 출력 |
| --- |

alcohol 12.989091

malic\_acid 2.336348

ash 2.366517

alcalinity\_of\_ash 19.494944

magnesium 99.741573

total\_phenols 2.295112

flavanoids 2.029270

nonflavanoid\_phenols 0.362370

proanthocyanins 1.590899

color\_intensity 5.058090

hue 0.957449

od280/od315\_of\_diluted\_wines 2.611685

proline 746.893258

class 0.938202

dtype: float64

이번에는 isna()와 mean()을 결합하여 확인합니다.

| data.isna().mean() # 값을 TRUE/FALSE로 변환 후 평균 출력 |
| --- |

alcohol 0.011236

malic\_acid 0.000000

ash 0.000000

alcalinity\_of\_ash 0.000000

magnesium 0.000000

total\_phenols 0.000000

flavanoids 0.000000

nonflavanoid\_phenols 0.028090

proanthocyanins 0.000000

color\_intensity 0.000000

hue 0.000000

od280/od315\_of\_diluted\_wines 0.000000

proline 0.000000

class 0.000000

dtype: float64

이번에도 역시 alcohol과 nonflavanoid\_phenols 컬럼에만 0이 아닌 숫자가 보입니다. 0과 1로 구성된 값들을 평균냈기 때문에, 이 값은 해당 컬럼에 몇 퍼센트나 결측치가 있는지를 보여줍니다. alcohol에 약 1.12%, nonflavanoid\_phenols에 약 2.81%가 결측치군요.



이처럼 isna()와 mean() 두 함수의 조합으로 결측치를 비율을 매우 효율적으로 확인할 수 있습니다. 이어서 결측치 비율을 고려하여 결측치를 처리하는 방법이 어떻게 달라지는지 알아보겠습니다.

### 6.5.2 결측치를 처리하는 방법

결측치를 처리하는 결측치 행 제거, 결측치 컬럼 제거, 결측치 채우기 방법을 알아보겠습니다. 가장 손쉬운 해결 방법은 결측치가 있는 줄을 지워버리는 겁니다.

#### 방식 1 결측치 행 제거하기 : dropna()

결측치가 있는 행을 제거해보겠습니다. 현재 데이터에 결측치 5개 있으므로 해당 5줄만 지우면 결측치 문제가 해결되는 거죠. dropna() 함수를 사용해 해당 줄을 제거할 수 있습니다.

| data.dropna() # 결측치가 있는 행 제거 |
| --- |

|  | alcohol | malic\_acid | ash | alcalinity\_of\_ash | magnesium | total\_phenols | flavanoids | nonflavanoid\_phenols | proanthocyanins | color\_intensity | hue | od280/od315\_of\_diluted\_wines | proline | class |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 14.23 | 1.71 | 2.43 | 15.60 | 127.00 | 2.80 | 3.06 | 0.28 | 2.29 | 5.64 | 1.04 | 3.92 | 1065 | 0 |
| 2 | 13.16 | 2.36 | 2.67 | 18.60 | 101.00 | 2.80 | 3.24 | 0.30 | 2.81 | 5.68 | 1.03 | 3.17 | 1185 | 0 |
| 3 | 14.37 | 1.95 | 2.50 | 16.80 | 113.00 | 3.85 | 3.49 | 0.24 | 2.18 | 7.80 | 0.86 | 3.45 | 1480 | 0 |
| 4 | 13.24 | 2.59 | 2.87 | 21.00 | 118.00 | 2.80 | 2.69 | 0.39 | 1.82 | 4.32 | 1.04 | 2.93 | 735 | 0 |
| 5 | 14.20 | 1.76 | 2.45 | 15.20 | 112.00 | 3.27 | 3.39 | 0.34 | 1.97 | 6.75 | 1.05 | 2.85 | 1450 | 0 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 173 | 13.71 | 6 | 2.45 | 20.5 | 95 | 1.68 | 0.61 | 0.52 | 1.06 | 7.7 | 0.64 | 1.74 | 740 | 2 |
| 174 | 13.40 | 4 | 2.48 | 23.0 | 102 | 1.8 | 0.75 | 0.43 | 1.41 | 7.3 | 0.7 | 1.56 | 750 | 2 |
| 175 | 13.27 | 4 | 2.26 | 20 | 120 | 1.59 | 0.69 | 0.43 | 1.35 | 10.2 | 0.59 | 1.56 | 835 | 2 |
| 176 | 13.17 | 3 | 2.37 | 20 | 120 | 1.65 | 0.68 | 0.53 | 1.46 | 9.3 | 0.6 | 1.62 | 840 | 2 |
| 177 | 14.13 | 4 | 2.74 | 24.5 | 96 | 2.05 | 0.76 | 0.56 | 1.35 | 9.2 | 0.61 | 1.6 | 560 | 2 |

171 rows × 14 columns

data()를 그냥 호출했을 때와 비슷해보이지만 맨 아래를 보면 총 데이터 수가 ‘171 rows’로 표시되네요. 즉 7줄이 삭제되었다는 이야깁니다. 왜 5줄 아니고 7줄이 삭제되었을까요? alcohol의 결측치와 nonflavanoid\_phenols의 결측치가 각각 다른 줄에 있었기 때문입니다. 결측치 일부가 같은 줄에 있다면 그만큼 덜 삭제되겠죠? 우선 결측치가 정말 사라졌는지 isna()와 mean() 함수를 덧붙여 확인해봅시다.

| data.dropna().isna().mean() |
| --- |

alcohol 0.0

malic\_acid 0.0

ash 0.0

alcalinity\_of\_ash 0.0

magnesium 0.0

total\_phenols 0.0

flavanoids 0.0

nonflavanoid\_phenols 0.0

proanthocyanins 0.0

color\_intensity 0.0

hue 0.0

od280/od315\_of\_diluted\_wines 0.0

proline 0.0

class 0.0

dtype: float64

모든 컬럼에서 결측치가 0%를 차지합니다. 여기서 dropna() 함수는 결측치를 처리한 결과를 보여주기만 할 뿐이지, 아직 data에 그 결과를 저장하지는 않았다는 점에 유의합시다. 결괏값으로 업데이트하고 싶다면 다음과 같이 data라는 이름으로 (혹은 원하시는 다른 이름으로) 할당하거나, inplace 매개변수를 사용하면 됩니다. 다만, 여기서는 dropna()의 결괏값으로 업데이트하지 않을 것이기 때문에 아래 코드는 참고만 하고 실행은 하지 마시기 바랍니다(둘 중 어느 방법을 써도 결과는 같습니다).

| data = data.dropna() # 참고만 하고 실행하지 말아주세요 |
| --- |

| data.dropna(inplace = True) # 참고만 하고 실행하지 말아주세요 |
| --- |

결측치를 가진 변수를 일관된 방식으로 처리할 수도 있지만, 필요에 따라 변수별로 다른 방식을 적용해 처리해야 할 때도 있습니다. 따라서 dropna()를 데이터 전체가 아닌, 특정 변수에 한정하여 적용시키는 방법도 알아야 합니다. subset 매개변수를 사용하면 특정 컬럼에만 적용할 수 있습니다. alcohol 변수에 dropna()를 적용시켜보겠습니다.

| data.dropna(subset=['alcohol']) # 지정된 변수의 결측치 행만 제거하기 |
| --- |

|  | alcohol | malic\_acid | ash | alcalinity\_of\_ash | magnesium | total\_phenols | flavanoids | nonflavanoid\_phenols | proanthocyanins | color\_intensity | hue | od280/od315\_of\_diluted\_wines | proline | class |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 14.23 | 1.71 | 2.43 | 15.60 | 127.00 | 2.80 | 3.06 | 0.28 | 2.29 | 5.64 | 1.04 | 3.92 | 1065 | 0 |
| 2 | 13.16 | 2.36 | 2.67 | 18.60 | 101.00 | 2.80 | 3.24 | 0.30 | 2.81 | 5.68 | 1.03 | 3.17 | 1185 | 0 |
| 3 | 14.37 | 1.95 | 2.50 | 16.80 | 113.00 | 3.85 | 3.49 | 0.24 | 2.18 | 7.80 | 0.86 | 3.45 | 1480 | 0 |
| 4 | 13.24 | 2.59 | 2.87 | 21.00 | 118.00 | 2.80 | 2.69 | 0.39 | 1.82 | 4.32 | 1.04 | 2.93 | 735 | 0 |
| 5 | 14.20 | 1.76 | 2.45 | 15.20 | 112.00 | 3.27 | 3.39 | 0.34 | 1.97 | 6.75 | 1.05 | 2.85 | 1450 | 0 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 173 | 13.71 | 6 | 2.45 | 20.5 | 95 | 1.68 | 0.61 | 0.52 | 1.06 | 7.7 | 0.64 | 1.74 | 740 | 2 |
| 174 | 13.40 | 4 | 2.48 | 23.0 | 102 | 1.8 | 0.75 | 0.43 | 1.41 | 7.3 | 0.7 | 1.56 | 750 | 2 |
| 175 | 13.27 | 4 | 2.26 | 20 | 120 | 1.59 | 0.69 | 0.43 | 1.35 | 10.2 | 0.59 | 1.56 | 835 | 2 |
| 176 | 13.17 | 3 | 2.37 | 20 | 120 | 1.65 | 0.68 | 0.53 | 1.46 | 9.3 | 0.6 | 1.62 | 840 | 2 |
| 177 | 14.13 | 4 | 2.74 | 24.5 | 96 | 2.05 | 0.76 | 0.56 | 1.35 | 9.2 | 0.61 | 1.6 | 560 | 2 |

176 rows × 14 columns

alcohol 변수에만 적용되었기 때문에 2개의 행만 제거되어 ‘176 rows’가 나왔습니다. isna()를 사용하여 더 확실하게 확인합니다.

| data.dropna(subset=['alcohol']).isna().mean() |
| --- |



예상한 대로 alcohol은 0이므로 결측치가 없습니다. 반면 nonflavanoid\_phenols은 0이 아니므로 여전히 결측치가 있네요.

#### 방식 2 결측 변수 제거하기 : drop()

두 번째 방법은 변수 자체를 제거하는 겁니다. 2개 변수에만 결측치가 있으니 해당 변수들을 제거해 결측치가 없는 데이터로 만들겠습니다.

| data.drop(['alcohol','nonflavanoid\_phenols'], axis = 1) # 변수를 제거하기 |
| --- |

|  | malic\_acid | ash | alcalinity\_of\_ash | magnesium | total\_phenols | flavanoids | proanthocyanins | color\_intensity | hue | od280/od315\_of\_diluted\_wines | proline | class |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1.71 | 2.43 | 15.60 | 127.00 | 2.80 | 3.06 | 2.29 | 5.64 | 1.04 | 3.92 | 1,065.00 | 0.00 |
| 1 | 1.78 | 2.14 | 11.20 | 100.00 | 2.65 | 2.76 | 1.28 | 4.38 | 1.05 | 3.40 | 1,050.00 | 0.00 |
| 2 | 2.36 | 2.67 | 18.60 | 101.00 | 2.80 | 3.24 | 2.81 | 5.68 | 1.03 | 3.17 | 1,185.00 | 0.00 |
| 3 | 1.95 | 2.50 | 16.80 | 113.00 | 3.85 | 3.49 | 2.18 | 7.80 | 0.86 | 3.45 | 1,480.00 | 0.00 |
| 4 | 2.59 | 2.87 | 21.00 | 118.00 | 2.80 | 2.69 | 1.82 | 4.32 | 1.04 | 2.93 | 735.00 | 0.00 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 173 | 5.65 | 2 | 20.5 | 95 | 1.68 | 0.61 | 1.06 | 7.7 | 0.64 | 1.74 | 740 | 2 |
| 174 | 3.91 | 2 | 23.0 | 102 | 1.8 | 0.75 | 1.41 | 7.3 | 0.7 | 1.56 | 750 | 2 |
| 175 | 4.28 | 2 | 20 | 120 | 1.59 | 0.69 | 1.35 | 10.2 | 0.59 | 1.56 | 835 | 2 |
| 176 | 2.59 | 2 | 20 | 120 | 1.65 | 0.68 | 1.46 | 9.3 | 0.6 | 1.62 | 840 | 2 |
| 177 | 4.1 | 3 | 24.5 | 96 | 2.05 | 0.76 | 1.35 | 9.2 | 0.61 | 1.6 | 560 | 2 |

178 rows × 12 columns

이 방법은 제가 제안하는 최종 해법이 아니므로 data를 업데이트하지 않고 넘어가겠습니다.

#### 방식 3 결측값 채우기 : fillna()

마지막으로 결측치에 값을 채워넣는 방법을 알아봅시다. 일반적으로 평균값이나 중간값을 이용합니다. 우선 코드를 이해하는 차원에서 평균값이나 중간값이 아닌 임의의 숫자 -99를 채워보겠습니다.

| data.fillna(-99) # 결측치를 -99로 변경하기 |
| --- |

|  | **alcohol** | **malic\_acid** | **ash** | **alcalinity\_of\_ash** | **magnesium** | **total\_phenols** | **flavanoids** | **nonflavanoid\_phenols** | **proanthocyanins** | **color\_intensity** | **hue** | **od280/od315\_of\_diluted\_wines** | **proline** | **class** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 14.23 | 1.71 | 2.43 | 15.6 | 127 | 2.80 | 3.06 | 0.28 | 2.29 | 5.64 | 1.04 | 3.92 | 1065 | 0 |
| **1** | -99.00 | 1.78 | 2.14 | 11.2 | 100 | 2.65 | 2.76 | 0.26 | 1.28 | 4.38 | 1.05 | 3.4 | 1050 | 0 |
| **2** | 13.16 | 2.36 | 2.67 | 18.6 | 101 | 2.80 | 3.24 | 0.3 | 2.81 | 5.68 | 1.03 | 3.17 | 1185 | 0 |
| **3** | 14.37 | 1.95 | 2.50 | 16.8 | 113 | 3.85 | 3.49 | 0.24 | 2.18 | 7.8 | 0.86 | 3.45 | 1480 | 0 |
| **4** | 13.24 | 2.59 | 2.87 | 21.0 | 118 | 2.80 | 2.69 | 0.39 | 1.82 | 4.32 | 1.04 | 2.93 | 735 | 0 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **173** | 13.71 | 5.65 | 2.45 | 20.5 | 95 | 1.68 | 0.61 | 0.52 | 1.06 | 7.7 | 0.64 | 1.74 | 740 | 2 |
| **174** | 13.40 | 3.91 | 2.48 | 23.0 | 102 | 1.80 | 0.75 | 0.43 | 1.41 | 7.3 | 0.7 | 1.56 | 750 | 2 |
| **175** | 13.27 | 4.28 | 2.26 | 20.0 | 120 | 1.59 | 0.69 | 0.43 | 1.35 | 10.2 | 0.59 | 1.56 | 835 | 2 |
| **176** | 13.17 | 2.59 | 2.37 | 20.0 | 120 | 1.65 | 0.68 | 0.53 | 1.46 | 9.3 | 0.6 | 1.62 | 840 | 2 |
| **177** | 14.13 | 4.10 | 2.74 | 24.5 | 96 | 2.05 | 0.76 | 0.56 | 1.35 | 9.2 | 0.61 | 1.6 | 560 | 2 |

178 rows × 14 columns



❶ alcohol 1열을 볼까요? NaN값이 -99로 변경되었습니다. 나머지 결측치들도 -99로 변경되었습니다. 이번에는 평균값인 mean()을 사용하여 결측치를 채워보겠습니다.

| data.fillna(data.mean()) # 평균값으로 결측치 채우기 |
| --- |

|  | **alcohol** | **malic\_acid** | **ash** | **alcalinity\_of\_ash** | **magnesium** | **total\_phenols** | **flavanoids** | **nonflavanoid\_phenols** | **proanthocyanins** | **color\_intensity** | **hue** | **od280/od315\_of\_diluted\_wines** | **proline** | **class** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 14.230000 | 1.71 | 2.43 | 15.6 | 127 | 2.80 | 3.06 | 0.28 | 2.29 | 5.64 | 1.04 | 3.92 | 1065 | 0 |
| **1** | 12.989091 | 1.78 | 2.14 | 11.2 | 100 | 2.65 | 2.76 | 0.26 | 1.28 | 4.38 | 1.05 | 3.4 | 1050 | 0 |
| **2** | 13.160000 | 2.36 | 2.67 | 18.6 | 101 | 2.80 | 3.24 | 0.3 | 2.81 | 5.68 | 1.03 | 3.17 | 1185 | 0 |
| **3** | 14.370000 | 1.95 | 2.50 | 16.8 | 113 | 3.85 | 3.49 | 0.24 | 2.18 | 7.8 | 0.86 | 3.45 | 1480 | 0 |
| **4** | 13.240000 | 2.59 | 2.87 | 21.0 | 118 | 2.80 | 2.69 | 0.39 | 1.82 | 4.32 | 1.04 | 2.93 | 735 | 0 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **173** | 13.710000 | 5.65 | 2.45 | 20.5 | 95 | 1.68 | 0.61 | 0.52 | 1.06 | 7.7 | 0.64 | 1.74 | 740 | 2 |
| **174** | 13.400000 | 3.91 | 2.48 | 23.0 | 102 | 1.80 | 0.75 | 0.43 | 1.41 | 7.3 | 0.7 | 1.56 | 750 | 2 |
| **175** | 13.270000 | 4.28 | 2.26 | 20.0 | 120 | 1.59 | 0.69 | 0.43 | 1.35 | 10.2 | 0.59 | 1.56 | 835 | 2 |
| **176** | 13.170000 | 2.59 | 2.37 | 20.0 | 120 | 1.65 | 0.68 | 0.53 | 1.46 | 9.3 | 0.6 | 1.62 | 840 | 2 |
| **177** | 14.130000 | 4.10 | 2.74 | 24.5 | 96 | 2.05 | 0.76 | 0.56 | 1.35 | 9.2 | 0.61 | 1.6 | 560 | 2 |

178 rows × 14 columns



fillna() 안에 data.mean()을 넣어 평균값으로 결측치를 채웠습니다. 별도로 컬럼명을 지정하지 않더라도 각각 컬럼에 대한 평균값을 구해 결측치를 채웠습니다(즉, alcohol의 평균치를 구하여 alcohol의 결측치를 채우고, nonflavanoid\_phenols의 평균을 구하여 nonflavanoid\_phenols의 결측치를 채웠습니다).

### 6.5.3 결측치 처리 방식 선택하기

지금까지 다룬 3가지 방법 중 무엇이 가장 좋을까요? 일반적인 방법은 dropna()를 사용하여 결측치 행을 지우는 겁니다. 평균 등을 이용하여 결측치를 채우면, 아무리 비슷한 값을 채운다고 할지라도 실젯값과 정확하게 일치할 가능성이 매우 낮기 때문에 오차의 원인이 될 수밖에 없습니다. 즉, 데이터에 노이즈가 더해진 효과를 내게 됩니다.

그런데 dropna()를 사용한 결측치 행 제거 방법에는 큰 단점이 있습니다. 경우에 따라서는 너무 과도하게 많은 데이터가 삭제될 수 있습니다. 지금은 단 7행만 삭제되지만, 만약 특정 변수의 90%가 결측치이면 90% 데이터가 삭제됩니다. 그래서 행을 지우는 방식을 사용하려면 결측치 비중이 매우 낮아야 하고, 데이터 크기도 충분히 커야 합니다. 그래야 결측치를 삭제해도 큰 영향을 미치지 않게 됩니다. 지금 우리가 다루는 데이터는 전체 178개의 데이터밖에 없으므로 매우 작습니다. 여기서는 단 7행 조차도 아쉬운 상황이니 행 삭제 방식을 사용하지 않겠습니다.

변수 자체를 drop()으로 제거하는 방식은 어떤가요? 매우 무모해보입니다. 머신러닝에서는 변수 하나하나가 중요합니다. 변수 자체를 없애버리는 방식은 모델링에 도움이 되지 않는 경우가 많습니다. 하지만 만약 특정 변수의 99%가 결측치라면 어떨까요? 이때는 행 삭제 방식을 적용하면 너무 많은 데이터가 사라져 버리고, 평균값으로 채워봤자 의미있는 변수라고 보기 어렵습니다. 이럴 때는 변수 자체를 제거하는 방식이 합리적입니다. 어느 정도 비중으로 결측치가 있을 때 drop()을 적용하는 것이 좋은가에 대한 기준은 상당히 주관적입니다. **통상적으로 50% 이상이면 drop()을 고려해볼 만하고, 70~80% 이상이면 가급적 drop() 적용하는 것이 좋습니다. 하지만 경우에 따라서는 90%가 결측치라고 해도, 해당 변수가 프로젝트에서 매우 중요한 역할을 할 거라 예상된다면 어떻게든 활용 방법을 찾는 것이 좋습니다.** 8장 ‘결정 트리 : 연봉 데이터셋’에서 문자형 변수의 결측치를 다룰 때 이부분을 설명합니다(이 장에서는 숫자형 변수에 대한 결측치 처리를 다루고 있습니다).

평균값 등으로 결측치를 채워주는 방식은 무난합니다. 여기에서는 단순히 평균값으로 처리하는 방법만 다루었지만, 평균보다 더 복잡하게 결측치를 채워넣는 방법이 다양하게 존재합니다. 다만, 어떤 방법일지라도 결국 추정치이기 때문에 노이즈를 완전히 피할 수는 없습니다., 결측치를 채우려는 노력과 시간에 비해서 평균으로 처리하는 것보다 월등히 나은 결과를 보여주지 않는 경우가 많기 때문에, 이 책에서는 평균으로 처리하는 수준까지만 다룹니다.

### 6.5.4 드디어 결측치 처리하기

이제 결측치를 어떻게 처리할지 결론을 말씀드려야겠군요. fillna()를 사용하여 결측치를 채우되, 아웃라이어에 조금 덜 민감한 중간값을 사용합니다.

| data.fillna(data.median(), inplace= True) # 결측치를 중간값으로 채우기 |
| --- |

이번에는 inplace 매개변수를 사용하여 data를 결측치 채운 결과로 data를 업데이트했습니다. 결측치 여부를 확인해봅시다. 모두 0.0으로 출력되면 결측치가 없는 겁니다.

| data.isna().mean() # 결측치 확인하기 |
| --- |

alcohol 0.0

malic\_acid 0.0

ash 0.0

alcalinity\_of\_ash 0.0

magnesium 0.0

total\_phenols 0.0

flavanoids 0.0

nonflavanoid\_phenols 0.0

proanthocyanins 0.0

color\_intensity 0.0

hue 0.0

od280/od315\_of\_diluted\_wines 0.0

proline 0.0

class 0.0

dtype: float64

## 6.6 스케일링

스케일링scaling은 데이터의 스케일scale을 맞추는 작업입니다. 스케일을 맞춘다니 무슨 말인지 잘 모르겠죠? 앞서 describe()를 호출해 데이터를 살펴보았을 때, 각 컬럼들마다 값들의 범위가 다양하다는 점을 확인했습니다. K-최근접 이웃은 거리 기반의 알고리즘이기 때문에, 이러한 스케일 문제가 안 좋은 결과를 초래할 수 있습니다. 즉, alcohol(최솟값 11.03, 최댓값 14.75)에서의 1과 magnesium(최솟값 70, 최댓값 162)에서의 1은 완전히 다른 영향을 미치기 때문에, KNN 알고리즘이 왜곡된 예측을 할 수 있습니다(왜 그런지는 6.9절 ‘이해하기’에서 더 구체적으로 살펴보겠습니다). 스케일링은 이러한 문제를 해결할 목적으로 인위적으로 각 컬럼이 비슷한 범위를 가지도록 만드는 작업입니다.



스케일링에는 표준화 스케일링Standarad Scaling, 로버스트 스케일링Robust Scaling, 최소-최대 스케일링Min-Max Scaling, 정규화 스케일링Normalizer Scaling이 있습니다.

▽스케일링 종류

| **종류** | **설명** |
| --- | --- |
| 표준화 스케일링 | 평균이 0이 되고, 표준편차가 1이 되도록 데이터를 고르게 분포시키는 데 사용 |
| 로버스트 스케일링 | 데이터에 아웃라이어가 존재하고, 그 영향력을 그대로 유지하고 싶을 때 사용 |
| 최소-최대 스케일링 | 데이터 분포의 특성을 최대한 그대로 유지하고 싶을 때 사용 |
| 정규화 스케일링 | 행 기준의 스케일링이 필요할 때 사용하나, 실제로 거의 사용하지 않음 |

이제부터 표준화 스케일링, 로버스트 스케일링, 최소-최대 스케일링 방식을 다룹니다.

#### 라이브러리 임포트하기

스케일링은 사이킷런 라이브러리의 preprocessing 모듈을 사용합니다. 우리가 사용할 3가지 스케일러를 임포트합시다.

| from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler # 한 라이브러리에서 여러 모듈 임포트 |
| --- |

import 뒤에 쉼표를 사용하여 3가지 모듈을 한 번에 불러왔습니다. from에 지정해준 라이브러리 안에서 여러 모듈을 불러올 때는 이처럼 간단하게 처리할 수 있으니 기억해두세요.

### 6.6.1 표준화 스케일링

수학적인 얘기는 잠시 뒤로 미뤄두고, 먼저 코드로 스케일링을 만나보겠습니다. 스케일러를 사용하는 방법은 모델링 코드와 비슷합니다. 우선 StandardScaler()를 호출해 스케일러(st\_scaler)를 지정합니다.

| st\_scaler = StandardScaler() # 스케일러 지정 |
| --- |

fit() 함수로 우리가 가진 데이터를 학습시킵니다. 이 단계에서는 스케일링을 위해 필요한 정보(평균, 표준편차)가 학습됩니다.

| st\_scaler.fit(data) # 학습 |
| --- |

학습이 되었으면 transform()으로 연산해줄 텐데, 이 단계에서는 학습에서 얻은 정보로 계산하게 됩니다. 결괏값을 st\_scaled라는 이름으로 저장합니다.

| st\_scaled = st\_scaler.transform(data) # 학습에서 얻은 정보 계산 |
| --- |

<함수>

| **함수명** | **설명** |
| --- | --- |
| transform() | 스케일러가 fit()을 통해 학습한 정보를 통해 데이터를 변환, 즉 스케일링하는 함수입니다. |

</>

결과를 확인하기 위해 st\_scaled를 출력하겠습니다.

| st\_scaled |
| --- |

array([[ 1.55484903, -0.5622498 , 0.23205254, ..., 1.84791957,

1.01300893, -1.21394365],

[ 0.07550273, -0.49941338, -0.82799632, ..., 1.1134493 ,

0.96524152, -1.21394365],

[ 0.21340789, 0.02123125, 1.10933436, ..., 0.78858745,

1.39514818, -1.21394365],

...,

[ 0.35131305, 1.74474449, -0.38935541, ..., -1.48544548,

0.28057537, 1.37386437],

[ 0.22594472, 0.22769377, 0.01273209, ..., -1.40069891,

0.29649784, 1.37386437],

[ 1.4294807 , 1.58316512, 1.36520822, ..., -1.42894777,

-0.59516041, 1.37386437]])

결과물의 형태가 조금 어색해 보일 겁니다. 그동안 판다스 데이터프레임 혹은 판다스 시리즈를 다뤘습니다. array가 앞에 붙은 이번 데이터는 넘파이 배열Array입니다. 넘파이는 계산을 더 효율적으로 수행할 목적으로 컬럼명 없이 데이터를 한 줄로 쭉 이어서 표현합니다. 반면 판다스 데이터프레임에서는 엑셀처럼 테이블이 갖춰져서 한 줄한 줄 정리된 데이터를 보여줬죠?



아무래도 사람이 읽기에는 넘파이보단 판다스가 편하기 때문에, 해당 데이터를 판다스 형태로 변경합니다.

| pd.DataFrame(st\_scaled) |
| --- |

|  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 1.554849 | -0.562250 | 0.232053 | -1.169593 | 1.913905 | 0.808997 | 1.034819 | -0.658865 | 1.224884 | 0.251717 | 0.362177 | 1.84792 | 1.013009 | -1.213944 |
| **1** | 0.075503 | -0.499413 | -0.827996 | -2.490847 | 0.018145 | 0.568648 | 0.733629 | -0.820072 | -0.544721 | -0.293321 | 0.406051 | 1.113449 | 0.965242 | -1.213944 |
| **2** | 0.213408 | 0.021231 | 1.109334 | -0.268738 | 0.088358 | 0.808997 | 1.215533 | -0.497658 | 2.135968 | 0.26902 | 0.318304 | 0.788587 | 1.395148 | -1.213944 |
| **3** | 1.730365 | -0.346811 | 0.487926 | -0.809251 | 0.930918 | 2.491446 | 1.466525 | -0.981279 | 1.032155 | 1.186068 | -0.427544 | 1.184071 | 2.334574 | -1.213944 |
| **4** | 0.313703 | 0.227694 | 1.840403 | 0.451946 | 1.281985 | 0.808997 | 0.663351 | 0.227773 | 0.401404 | -0.319276 | 0.362177 | 0.449601 | -0.037874 | -1.213944 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **173** | 0.902934 | 2.974543 | 0.305159 | 0.301803 | -0.332922 | -0.985614 | -1.424900 | 1.275618 | -0.930179 | 1.142811 | -1.392758 | -1.231206 | -0.021952 | 1.373864 |
| **174** | 0.514292 | 1.412609 | 0.414820 | 1.052516 | 0.158572 | -0.793334 | -1.284344 | 0.550187 | -0.31695 | 0.969783 | -1.129518 | -1.485445 | 0.009893 | 1.373864 |
| **175** | 0.351313 | 1.744744 | -0.389355 | 0.151661 | 1.422412 | -1.129824 | -1.344582 | 0.550187 | -0.422075 | 2.224236 | -1.612125 | -1.485445 | 0.280575 | 1.373864 |
| **176** | 0.225945 | 0.227694 | 0.012732 | 0.151661 | 1.422412 | -1.033684 | -1.354622 | 1.356221 | -0.229346 | 1.834923 | -1.568252 | -1.400699 | 0.296498 | 1.373864 |
| **177** | 1.429481 | 1.583165 | 1.365208 | 1.502943 | -0.262708 | -0.392751 | -1.274305 | 1.598032 | -0.422075 | 1.791666 | -1.524378 | -1.428948 | -0.59516 | 1.373864 |

178 rows × 14 columns



이제 훨씬 보기 편해졌으나, 컬럼명이 숫자로 바뀌었네요. 넘파이는 컬럼명을 가지지 않기 때문에 데이터프레임으로 변경하면 이런 식으로 표현됩니다. 컬럼명이 있어야 다루기 편하므로 원래 다룬 데이터 테이블인 data에서 컬럼명을 가져와서 지정해주겠습니다.

| # 컬럼명을 지정하여 데이터 프레임으로 변환  st\_scaled = pd.DataFrame(st\_scaled, columns = data.columns) |
| --- |

data.columns 명령으로 data의 컬럼명을 가져와 결과물을 st\_scaled에 업데이트했습니다.

이제 st\_scaled를 확인하면 다음과 같은 결과물을 볼 수 있습니다.

|  | alcohol | malic\_acid | ash | alcalinity\_of\_ash | magnesium | total\_phenols | flavanoids | nonflavanoid\_phenols | proanthocyanins | color\_intensity | hue | od280/od315\_of\_diluted\_wines | proline | class |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1.55 | -0.56 | 0.23 | -1.17 | 1.91 | 0.81 | 1.03 | -0.66 | 1.22 | 0.25 | 0.36 | 1.85 | 1.013009 | -1.213944 |
| 1 | 0.08 | -0.50 | -0.83 | -2.49 | 0.02 | 0.57 | 0.73 | -0.82 | -0.54 | -0.29 | 0.41 | 1.11 | 0.965242 | -1.213944 |
| 2 | 0.21 | 0.02 | 1.11 | -0.27 | 0.09 | 0.81 | 1.22 | -0.50 | 2.14 | 0.27 | 0.32 | 0.79 | 1.395148 | -1.213944 |
| 3 | 1.73 | -0.35 | 0.49 | -0.81 | 0.93 | 2.49 | 1.47 | -0.98 | 1.03 | 1.19 | -0.43 | 1.18 | 2.334574 | -1.213944 |
| 4 | 0.31 | 0.23 | 1.84 | 0.45 | 1.28 | 0.81 | 0.66 | 0.23 | 0.40 | -0.32 | 0.36 | 0.45 | -0.037874 | -1.213944 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 173 | 0.902934 | 3 | 0.305159 | 0.301803 | -0.332922 | -0.985614 | -1.4249 | 1.275618 | -0.930179 | 1.142811 | -1.392758 | -1.231206 | -0.021952 | 1.373864 |
| 174 | 0.514292 | 1 | 0.414820 | 1.052516 | 0.158572 | -0.793334 | -1.284344 | 0.550187 | -0.31695 | 0.969783 | -1.129518 | -1.485445 | 0.009893 | 1.373864 |
| 175 | 0.351313 | 2 | -0.389355 | 0.151661 | 1.422412 | -1.129824 | -1.344582 | 0.550187 | -0.422075 | 2.224236 | -1.612125 | -1.485445 | 0.280575 | 1.373864 |
| 176 | 0.225945 | 0 | 0.012732 | 0.151661 | 1.422412 | -1.033684 | -1.354622 | 1.356221 | -0.229346 | 1.834923 | -1.568252 | -1.400699 | 0.296498 | 1.373864 |
| 177 | 1.429481 | 2 | 1.365208 | 1.502943 | -0.262708 | -0.392751 | -1.274305 | 1.598032 | -0.422075 | 1.791666 | -1.524378 | -1.428948 | -0.59516 | 1.373864 |

178 rows × 14 columns

#### 이해하기 : 표준화 스케일링

코드로 결과물을 얻는 방법을 배웠으니, 이제 이론적인 이야기를 합니다. 표준화 스케일링은 데이터를 표준화된 정규분포Standard Normal Distribution으로 만들어주는 방법인데, 다음과 같은 공식으로 계산이 됩니다.



각 변수를 기준으로 하여, 각 데이터에 해당 변수의 평균만큼을 빼주고 이를 표준편차로 나눕니다. 이 수식을 거치면 모든 컬럼이 표준정규분포의 형태를 따르게 됩니다.

우리가 fit() 함수로 학습을 시켜주는 과정에서 각 컬럼의 평균과 표준편차가 st\_scaler에 기억되고, transform()을 적용하면 그 값들을 이용하여 위의 수식으로 연산을 하는 겁니다.

만약 여러분이 정규분포에 대해 기억한다면 표준화 스케일링은 매우 이해하기 쉬울 겁니다. 데이터를 표준화하면 평균은 0이고 분산은 1인 형태로 데이터가 변경됩니다. 소수점 2자리까지 표시하도록 describe()를 호출해봅시다.

| round(st\_scaled.describe(), 2) # 소수점 2번째 자리까지 출력  반올림 |
| --- |

|  | alcohol | malic\_acid | ash | alcalinity\_of\_ash | magnesium | total\_phenols | flavanoids | nonflavanoid\_phenols | proanthocyanins | color\_intensity | hue | od280/od315\_of\_diluted\_wines | proline | class |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| count | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 |
| mean | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | -0.00 | -0.00 |
| std | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| min | -2.46 | -1.43 | -3.68 | -2.67 | -2.09 | -2.11 | -1.70 | -1.87 | -2.07 | -1.63 | -2.09 | -1.90 | -1.49 | -1.21 |
| 25% | -0.79 | -0.66 | -0.57 | -0.69 | -0.82 | -0.89 | -0.83 | -0.74 | -0.60 | -0.80 | -0.77 | -0.95 | -0.78 | -1.21 |
| 50% | 0.08 | 0 | -0.02 | 0.00 | -0.12 | 0.1 | 0.11 | -0.18 | -0.06 | -0.16 | 0.03 | 0.24 | -0.23 | 0.08 |
| 75% | 0.84 | 1 | 0.70 | 0.60 | 0.51 | 0.81 | 0.85 | 0.61 | 0.63 | 0.49 | 0.71 | 0.79 | 0.76 | 1.37 |
| max | 2.21 | 3 | 3.16 | 3.15 | 4.37 | 2.54 | 3.06 | 2.4 | 3.49 | 3.44 | 3.3 | 1.96 | 2.97 | 1.37 |

모든 컬럼에서 평균과 표준편차가 각각 0과 1임을 볼 수 있습니다. min과 max값은 컬럼마다 각기 다르지만, 처음보다는 훨씬 차이가 줄었습니다. 이로써 데이터가 더 동등한 수준에서 연산될 수 있습니다.

### 6.6.2 로버스트 스케일링

이번에는 로버스트 스케일링을 사용할텐데, 이번에는 더 효율적인 학습/연산 코드를 소개합니다. RobustScaler()를 호출해 로버스트 스케일링에 사용할 객체를 생성합니다.

| rb\_scaler = RobustScaler() # 로버스트 스케일링에 사용할 객체를 생성 |
| --- |

그러고 나서 fit()과 transform()이 합쳐진 fit\_transform() 함수를 사용해 스케일링합니다.

| rb\_scaled = rb\_scaler.fit\_transform(data) # 로버스트 스케일링 rb\_scaled = pd.DataFrame(rb\_scaled, columns = data.columns) # 데이터프레임으로 변형 |
| --- |

fit\_transform()는 fit()으로 학습한 값으로 transform()까지 해주는 함수입니다. 결과물을 rb\_scaled로 저장하고, 데이터 프레임으로 변형했습니다.

이번에도 describe()를 호출해 결과물을 확인해봅시다.

| round(rb\_scaled.describe(), 2) # 소수점 2번째 자리까지 출력 |
| --- |

|  | alcohol | malic\_acid | ash | alcalinity\_of\_ash | magnesium | total\_phenols | flavanoids | nonflavanoid\_phenols | proanthocyanins | color\_intensity | hue | od280/od315\_of\_diluted\_wines | proline | class |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| count | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 |
| mean | -0.05 | 0.32 | 0.02 | 0.00 | 0.09 | -0.06 | -0.06 | 0.13 | 0.05 | 0.12 | -0.02 | -0.14 | 0.15 | -0.03 |
| std | 0.62 | 0.75 | 0.79 | 0.78 | 0.75 | 0.59 | 0.60 | 0.74 | 0.82 | 0.78 | 0.68 | 0.58 | 0.65 | 0.39 |
| min | -1.55 | -0.76 | -2.88 | -2.07 | -1.47 | -1.30 | -1.07 | -1.25 | -1.64 | -1.14 | -1.44 | -1.23 | -0.82 | -0.50 |
| 25% | -0.53 | -0.18 | -0.43 | -0.53 | -0.53 | -0.58 | -0.56 | -0.42 | -0.44 | -0.49 | -0.54 | -0.68 | -0.36 | -0.50 |
| 50% | 0.00 | 0 | 0.00 | 0.00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 75% | 0.47 | 1 | 0.57 | 0.47 | 0.47 | 0.42 | 0.44 | 0.58 | 0.56 | 0.51 | 0.46 | 0.32 | 0.64 | 0.5 |
| max | 1.31 | 3 | 2.50 | 2.44 | 3.37 | 1.44 | 1.76 | 1.91 | 2.89 | 2.79 | 2.21 | 0.99 | 2.08 | 0.5 |

#### 이해하기 : 로버스트 스케일링

표준화 스케일링와 달리 로버스트 스케일링은 평균과 표준편차 대신 사분위값을 이용하여 계산됩니다. 때문에 이번에는 평균값이 0에 가깝지만 완전 0은 아니고, 분산 또한 표준 스케일링과는 다르게 1로 고정되지 않았습니다.



여기서 Q1, Q2, Q3는 각각 데이터의 25%, 50%, 75% 지점의 값입니다. 특히 50% 지점인 Q2는 중위값Median이라고도 부릅니다. 즉, 수식에 따르면 각 값에 중위값을 빼주고, 75% 지점에서 25% 지점까지 범위만큼으로 나누어주는 겁니다. 물론 여기에서 Q1, Q2, Q3 값은 데이터 전체가 아닌 각 컬럼에서 구해진 값들입니다.

### 6.6.3 최소-최대 스케일링

마지막으로 최소-최대 스케일링입니다. 이 모듈 또한 동일한 방식으로 해주시면 됩니다. MinMaxScaler() 함수를 사용해주세요.

| mm\_scaler = MinMaxScaler() # 최소-최대 스케일링 객체 생성 mm\_scaled = mm\_scaler.fit\_transform(data) # 최소-최대 스케일링 mm\_scaled = pd.DataFrame(mm\_scaled, columns = data.columns) # 데이터프레임으로 변형 round(mm\_scaled.describe(), 2) # 소수점 2째 자리까지 출력 |
| --- |

|  | alcohol | malic\_acid | ash | alcalinity\_of\_ash | magnesium | total\_phenols | flavanoids | nonflavanoid\_phenols | proanthocyanins | color\_intensity | hue | od280/od315\_of\_diluted\_wines | proline | class |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| count | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 | 178.00 |
| mean | 0.53 | 0.32 | 0.54 | 0.46 | 0.32 | 0.45 | 0.36 | 0.44 | 0.37 | 0.32 | 0.39 | 0.49 | 0.33 | 0.47 |
| std | 0.22 | 0.22 | 0.15 | 0.17 | 0.16 | 0.22 | 0.21 | 0.23 | 0.18 | 0.20 | 0.19 | 0.26 | 0.22 | 0.39 |
| min | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 25% | 0.36 | 0.17 | 0.45 | 0.34 | 0.20 | 0.26 | 0.18 | 0.26 | 0.26 | 0.17 | 0.25 | 0.24 | 0.16 | 0.00 |
| 50% | 0.54 | 0 | 0.53 | 0.46 | 0.3 | 0.47 | 0.38 | 0.4 | 0.36 | 0.29 | 0.39 | 0.55 | 0.28 | 0.5 |
| 75% | 0.71 | 0 | 0.64 | 0.56 | 0.4 | 0.63 | 0.53 | 0.58 | 0.49 | 0.42 | 0.52 | 0.7 | 0.5 | 1 |
| max | 1.00 | 1 | 1.00 | 1.00 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

#### 이해하기 : 최소-최대 스케일링

최소-최대 스케일링의 특징은 모든 컬럼에서 최댓값이 1, 최솟값이 0인 형태로 변환된다는 겁니다. 적용 수식은 다음과 같습니다.



각 값에서 최솟값을 빼주고, 최댓값과 최솟값의 차이만큼으로 나누어주면 위와 같은 최소-최대 스케일링 결괏값이 나옵니다.

### 6.6.4 스케일링 방식 선택하기

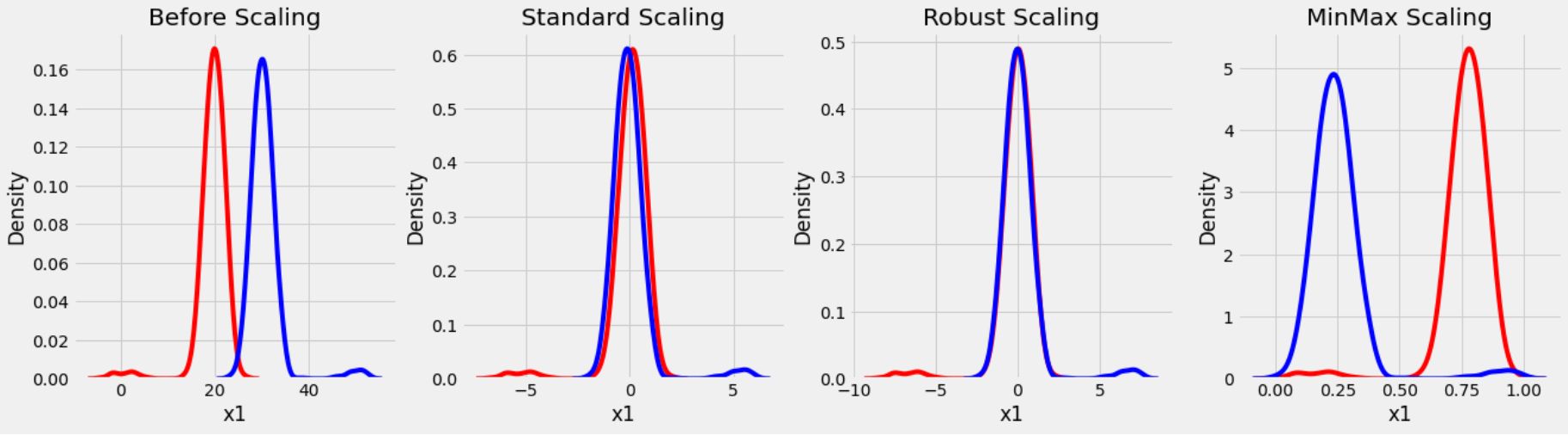
그렇다면 언제 어떤 방법을 사용하는 것이 좋을까요? 우선 아웃라이어의 유무에 따라 판단하는데, 아웃라이어의 영향이 큰 데이터이고 이를 피하고 싶다면 로버스트 스케일링이 적합합니다. 표준화 스케일링은 평균과 표준편차를 이용하고 최소-최대 스케일링은 최대/최솟값을 이용하는데, 이들은 모두 아웃라이어가 있을 때 민감하게 반응하기 때문입니다. 반면 로버스트 스케일링은 사분위값을 사용하기 때문에 아웃라이어가 있어도 영향을 덜 받습니다.

때에 따라서는 아웃라이어 존재 자체가 중요할 수도 있습니다. 이 경우에는 데이터의 기존 분포를 최대한 유지시켜야 하므로 최소-최대 스케일링이 적합합니다. 표준화 스케일링는 모든 데이터를 표준정규분포 형태, 즉 좌우 대칭의 종 모양으로 변경하기 때문에, 아웃라이어의 영향을 받으면서도 기존의 데이터 분포에 대한 특징은 상실하게 됩니다. 반면 최소-최대 스케일링은 최댓값 1과 최솟값 0의 범위에서, 기존 데이터의 분포를 최대한 그대로 옮겨담아 냅니다. 그래서 아웃라이어가 있다면 그 분포 모습 그대로 결괏값에도 나타납니다.

표준화 스케일링은 기존 데이터가 정규분포를 따르고 있고 아웃라이어가 없는 상황에서 무난하게 사용됩니다. 데이터의 분포가 정규분포와 상당히 거리가 있거나 아웃라이어가 상당수 있는 경우에 표준화 스케일링을 사용하면 기존 데이터의 특징을 상당히 잃어버릴 수 있으니 주의해야 합니다.

▽ 스케일링별 특징

| **구분** | **결과물의 특징** |
| --- | --- |
| 표준화 스케일링 | 데이터에 아웃라이어가 존재할 때 아웃라이어의 영향을 감소시켜줍니다. 평균 0, 분산 1이 되게끔 분포시키기 때문에, 데이터의 기존 분포 형태가 사리지고 정규 분포를 따르는 결과물을 가져옵니다. |
| 로버스트 스케일링 | 데이터에 아웃라이어가 존재할 때, 아웃라이어의 영향력을 그대로 유지시켜줍니다. 변환된 데이터의 범위는 표준화 스케일링이나 최소-최대 스케일링보다 넓게 나타납니다. |
| 최소-최대 스케일링 | 로버스트 스케일링과 마찬가지로 아웃라이어의 영향력을 유지시켜줍니다. 위의 두 스케일러와 비교했을 때, 데이터의 기존 분포를 가장 있는 그대로 담아내며 스케일만 변화시킵니다. 데이터의 범위는 0~1로 나타납니다. |

▽ 스케일링별 분포  
  
위 그래프는 아웃라이어가 존재하는 임의의 데이터를 사용하여 만든 스케일링 종류별 분포입니다. 아웃라이어에 따라 표준화 스케일링과 로버스트 스케일링 간에 약간의 차이가 보이며, 최소-최대 스케일링에서는 기존의 분포처럼 빨간색 변수와 파란색 변수의 분포가 서로 멀찌감치 떨어져 있습니다(두 선의 위치가 기존 분포와 반대이나, 이는 해당 공식을 대입해보면 왜 이런 반전이 생기는지 쉽게 이해할 수 있습니다).

머신러닝에 스케일링을 적용할 때 주의하여야 할 점이 있습니다.

1. **스케일링 대상에서 종속변수를 제외해야 합니다.** 앞서 스케일링 연습 시, 결과물에서 class가 0, 1, 2가 아닌 다른 값들로 전부 바뀌었는데, 우리는 class라는 컬럼을 예측해야 하기 때문에 이 변수는 그대로 남겨두어야 합니다.
2. **스케일링 전에 훈련셋과 시험셋을 나누어야 합니다.** 훈련셋에서 fit()으로 스케일링을 위한 값을 학습시키고, 이 값을 활용하여 훈련셋과 시험셋을 변환해야 합니다.

이 과정을 표준화 스케일링으로 예를 들겠습니다. 일단 훈련셋에 스케일러의 fit() 함수를 사용하여 평균과 표준편차 값을 학습합니다. 평균과 표준편차 값을 사용하여 transform()으로 훈련셋과 시험셋을 각각 변환시켜줍니다. 즉, 시험셋에서는 fit()을 사용해 평균과 표준편차를 구하지 않는다는 겁니다.

만약 훈련셋과 시험셋을 나누기 전에 스케일링을 사용하면 어떨까요? 전체 데이터에서 fit()을 사용하여 평균과 표준편차를 구하고, transform()으로 변환까지 완료한 뒤에 데이터를 나누게 될 겁니다. 얼핏 생각하면 문제가 없어 보입니다. 그렇다면 훈련셋과 시험셋을 나누는 이유는 무얼까요? 시험셋은 앞으로 예측할 진짜 새로운 데이터를 대비한 가상의 새로운 데이터입니다. 따라서 훈련셋과 시험셋을 합쳐서 평균과 표준편차를 구한다는 행위는 실제로 새로운 데이터를 받았을 때 기존 데이터와 합쳐서 평균과 표준편차를 구하는 행위와 같습니다. 새 데이터가 올 때마다 합쳐서 다시 학습을 하는 방식은 상당한 비용이 드는 불편한 프로세스입니다.

그럼 기존 데이터와 굳이 합치지 않고, 새로운 데이터에 대해서만 fit()을 사용하여 평균과 표준편차를 구할 수도 있지 않나 생각할 수도 있습니다. 하지만 이것은 가장 현실적이지 못한 방법입니다. 만약 여러분의 모델이 실시간으로 예측하는 모델이라서 새로운 데이터가 한 건씩 들어온다고 가정합시다. 데이터 한 건에 대한 평균과 표준편차는 무의미합니다. 따라서 모델링에 사용한 데이터의 평균과 표준편차를 그대로 활용하여 새로운 데이터에 transform()을 적용해주는 방식이 합당합니다. 기존 데이터가 훈련셋 역할이고, 새로운 데이터가 시험셋 역할입니다.

### 6.6.5 스케일링 적용하기

그럼 본격적으로 스케일링을 하기 전에 데이터부터 나누겠습니다.

| from sklearn.model\_selection import train\_test\_split X\_train, X\_test, y\_train, y\_test = train\_test\_split(data.drop('class', axis=1),data['class'], test\_size=0.2, random\_state=100) # 학습셋과 시험셋 분리 |
| --- |

X\_train

X\_test

y\_train

y\_test

<note/>

이번에는 X와 y를 따로 지정하지 않고 train\_test\_split 안에 바로 넣어주었습니다. 위 코드가 복잡해보인다면 다음과 같이 X와 y를 따로 지정해도 좋습니다.

| X = data.drop('class', axis=1) # 독립변수만 대입  y = data['class'] # 종속변수만 대입 X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.2, random\_state = 100) # 학습셋과 시험셋 분리 |
| --- |

</>

세 가지 스케일링 기법 중 데이터 특성을 그대로 보존하는 최대-최소 스케일러를 사용하겠습니다.

| mm\_scaler = MinMaxScaler() # 최대-최소 스케일러 객체 생성 mm\_scaler.fit(X\_train) # 학습 |
| --- |

우선 X\_train에 fit()을 적용하여 스케일링을 위한 값들을 학습시켰습니다. 이제 mm\_scaler로 X\_train과 X\_test를 변환합니다.

| X\_train\_scaled = mm\_scaler.transform(X\_train) # 학습셋 트랜스폼 X\_test\_scaled = mm\_scaler.transform(X\_test) # 시험셋 트랜스폼 |
| --- |

fit\_transform()을 사용하면 다음과 같이 간단하게 만들 수 있습니다.

| mm\_scaler = MinMaxScaler() # 최대-최소 스케일러 객체 생성 X\_train\_scaled = mm\_scaler.fit\_transform(X\_train) # 학습셋 학습 및 트랜스폼 X\_test\_scaled = mm\_scaler.transform(X\_test) # 시험셋 학습 및 트랜스폼 |
| --- |

fit\_transform() 함수는 fit()과 transform()을 동시에 처리해 fit()으로 학습된 정보를 mm\_scaler에 저장합니다. 따라서 세 번째 줄부터는 fit() 과정을 다시 거칠 필요 없어 바로 transform()을 사용해 X\_test를 스케일링했습니다.

앞서 주의점으로 목푯값을 스케일링에서 제외해야 한다고 말씀드렸습니다. train\_test\_split()을 통해 독립변수(X)와 종속변수(y) 또한 분류했으므로 X\_train과 X\_test에는 자연스럽게 목푯값이 제외되어 있습니다. 이번에는 굳이 데이터프레임으로 변형시키지 않고, 넘파이 형태 그대로 모델링에 적용시키겠습니다.

## 6.7 모델링 및 예측/평가하기

KNN 모델 역시 sklearn 라이브러리에 있습니다. 다음과 같이 불러옵니다.

| from sklearn.neighbors import KNeighborsClassifier |
| --- |

사용 방법은 회귀 알고리즘의 과정과 동일합니다.

| knn = KNeighborsClassifier() # KNN 모델 생성 knn.fit(X\_train\_scaled,y\_train) # 학습 pred = knn.predict(X\_test\_scaled) # 예측 |
| --- |

KNeighborsClassifier() 함수를 호출해 알고리즘의 속성을 KNN으로 부여했고, fit()으로 학습시킨 후에 predict()로 예측까지 간단하게 완료했습니다. 예측된 값을 pred 변수에 저장했습니다. 그럼 어떤 형태로 저장되어 있는지 pred를 출력해봅시다.

| pred |
| --- |

array([1, 2, 0, 1, 2, 2, 1, 2, 1, 0, 2, 0, 2, 2, 2, 0, 2, 0, 1, 0, 2, 0,

2, 1, 0, 0, 1, 1, 1, 2, 2, 1, 0, 1, 2, 2]

0, 1, 2 중 어디에 해당하는지 예측한 값이 저장되어 있습니다. 그럼 우리가 알고 있는 정답인 y\_test와 예측값인 pred가 얼마나 일치하는지 accuracy\_score를 사용해 확인해봅시다.

| from sklearn.metrics import accuracy\_score accuracy\_score(y\_test, pred) |
| --- |

0.8888888888888888

약 89%의 정확도를 보여줍니다. 이정도면 괜찮은 수준의 예측이라 할 수 있습니다.

## 6.8 하이퍼파라미터 튜닝하기

KNN 알고리즘에는 아주 중요한 매개변수가 하나 있습니다. 바로 n\_neighbors라는 이름의 매개변수인데, 예측에 가까운 이웃을 몇 개나 고려할지를 정합니다. 이 부분은 코딩 파트 이후에 알고리즘 부분에서 더 자세히 설명드리겠습니다.

우리가 knn = KNeighborsClassifier()로 속성을 부여할 때 별도로 매개변수를 지정해주지 않았는데, 별도로 지정해주지 않으면 기본값이 적용됩니다.

<함수/>

| **함수** | **설명** |
| --- | --- |
| KNeighborsClassifier() | KNN은 분류와 회귀 문제 모두 지원합니다. KNeighborsClassifier는 분류문제를 위한 알고리즘이며, 회귀문제에는 KNeighborsRegressor를 사용합니다.  주요 파라미터는 다음과 같습니다.   * n\_neighbors=5 : 예측에 참고할 이웃 수 * weights='uniform' : 예측에 사용되는 가중치 함수로 기본값인 uniform은 모든 포인트에 동일한 가중치가 부여됩니다. - 'uniform','distance', 사용자 정의 함수 * metric='minkowski' : 거리 측정 기준, 사용 가능한 매개변수는 sklearn의 DistanceMetric 문서 참조[[2]](#footnote-1) * n\_jobs=None : 실행할 병렬 작업 수 |

= 뒤에 정의된 값이 기본값입니다. n\_neighbors는 기본값이 5입니다. 5가 아닌 7로 설정하여 다시 한번 모델링하고 평가해봅시다.

| knn = KNeighborsClassifier(n\_neighbors=7) # KNN 모델 생성 knn.fit(X\_train\_scaled,y\_train) # 학습 pred = knn.predict(X\_test\_scaled) # 예측 accuracy\_score(y\_test, pred) # 정확도 계산 |
| --- |

0.9166666666666666

n\_neighbors를 7로 설정하여 모델링해보니 약 92%로 예측 정확도가 높아졌습니다. 이번에는 3으로 설정하여 모델을 돌려보겠습니다.

| knn = KNeighborsClassifier(n\_neighbors=3) # KNN 모델 생성 knn.fit(X\_train\_scaled,y\_train) # 학습  pred = knn.predict(X\_test\_scaled) # 예측 accuracy\_score(y\_test, pred) # 정확도 계산 |
| --- |

0.8888888888888888

이번에는 약 89%로 기본값 5일 때와 같은 결과입니다. 이렇게 n\_neighbors에 들어가는 숫자가 달라지면 예측 결과 또한 달라질 수 있는데, 어떤 숫자로 설정했을 때 가장 좋을지는 데이터에 따라 매번 달라지기 때문에 일일이 확인하는 방법밖에는 없습니다. 하나하나 수많은 숫자를 수작업으로 확인해보는 방식은 상당히 번거롭습니다. 적당한 범위에서 반복되도록 프로그램을 수정해봅시다.

우선 매개변수를 1부터 20까지, 총 20개의 숫자를 넣어볼 for문을 만들어 예측해보겠습니다. 마지막 숫자인 21은 포함되지 않기때문에, 1부터 20까지의 숫자가 i에 적용될 겁니다.

| for i in range(1, 21): # ❶ 20번 반복  knn = KNeighborsClassifier(n\_neighbors=i) # KNN 모델 생성  knn.fit(X\_train\_scaled,y\_train) # 학습  pred = knn.predict(X\_test\_scaled) # 예측  print(accuracy\_score(y\_test, pred)) # ❷ 정확도 출력 |
| --- |

0.9166666666666666

0.8888888888888888

0.8888888888888888

0.9166666666666666

0.8888888888888888

0.9166666666666666

0.9166666666666666

0.8888888888888888

0.8888888888888888

0.8888888888888888

0.9166666666666666

0.9166666666666666

0.9722222222222222

0.9444444444444444

0.9444444444444444

0.9444444444444444

0.9166666666666666

0.9722222222222222

0.9444444444444444

0.9722222222222222

❶ n\_neighbors가 i로 지정되어 있기 때문에 해당 코드를 실행하면 매개변수가 1부터 20까지, 총 20번의 모델링을 거치게 됩니다. ❷에서 각 정확도를 출력합니다.

(그래프를 그리는 용도 등으로) 정확도를 활용할 수 있도록 리스트 형태로 저장해봅시다. 우선 정확도를 저장할 빈 리스트를 scores라는 이름으로 만듭니다.

| scores = [ ] # 빈 리스트 생성 |
| --- |

그리고 위의 for문과 동일한 코드를 작성하되, 맨 마지막의 print() 대신에 append()를 사용하여 scores에 값을 저장합니다.

| for i in range(1, 21): # 1부터 20까지 반복  knn = KNeighborsClassifier(n\_neighbors=i) # KNN 모델 생성  knn.fit(X\_train\_scaled,y\_train) # 학습  pred = knn.predict(X\_test\_scaled) # 예측  acc = accuracy\_score(y\_test, pred) # 정확도 계산  scores.append(acc) # ❶ 정확도 저장 |
| --- |

이 코드를 실행하면 역시 총 20번 반복하며 모델링 및 예측을 하고 ❶ scores 리스트 안에 정확도를 저장됩니다. 이제 scores를 출력해보면 총 20개의 accuracy score를 확인할 수 있습니다.

| scores |
| --- |

[0.9166666666666666,

0.8888888888888888,

0.8888888888888888,

0.9166666666666666,

0.8888888888888888,

0.9166666666666666,

0.9166666666666666,

0.8888888888888888,

0.8888888888888888,

0.8888888888888888,

0.9166666666666666,

0.9166666666666666,

0.9722222222222222,

0.9444444444444444,

0.9444444444444444,

0.9444444444444444,

0.9166666666666666,

0.9722222222222222,

0.9444444444444444,

0.9722222222222222]

리스트로 저장했으니 활용해보겠습니다. 매개변수의 변화에 따른 예측력의 변화를 한눈에 보여줄 선그래프를 그려보겠습니다.

<함수/>

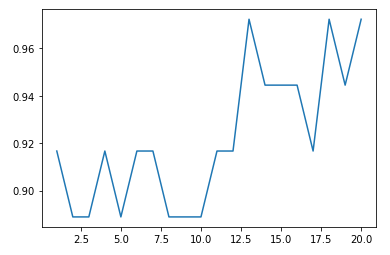
| **함수** | **설명** |
| --- | --- |
| lineplot() | 선형 그래프를 그리는 함수로 아래와 같이 사용합니다.  <코드/>  sns.lineplot(data = {데이터 이름}, x = {x축에 넣을 변수}, y = {y축에 넣을 변수})  sns.lineplot(x = {x축에 넣을 데이터}, y = {y축에 넣을 데이터})  </> |

</>

여기에서 x값은 for문에 넣었던 1부터 20까지의 숫자, y값은 리스트에 저장된 예측도가 되겠습니다.

| sns.lineplot(x=range(1,21), y=scores) # 그래프 생성 및 출력 |
| --- |

<matplotlib.axes.\_subplots.AxesSubplot at 0x7fbe92ce7d50



결과를 보면 대체로 매개변수 값이 클수록 더욱 나은 예측을 보여주지만, 13 이후로는 딱히 더 나은 개선은 보이지 않습니다. 결과가 같다면 연산을 더 많이 하는 더 큰 값을 매개변수로 사용할 필요는 없습니다. 따라서 여기서는 13이 합리적인 선택입니다. 이제 가장 좋은 매개변수를 확인했으니 n\_neighbors를 13으로 지정하면 됩니다.

| knn = KNeighborsClassifier(n\_neighbors=13) # KNN 모델 생성 knn.fit(X\_train\_scaled,y\_train) # 학습 pred = knn.predict(X\_test\_scaled) # 예측 accuracy\_score(y\_test, pred) # 정확도 출력 |
| --- |

0.9722222222222222

## 6.9 이해하기 : K-최근접 이웃

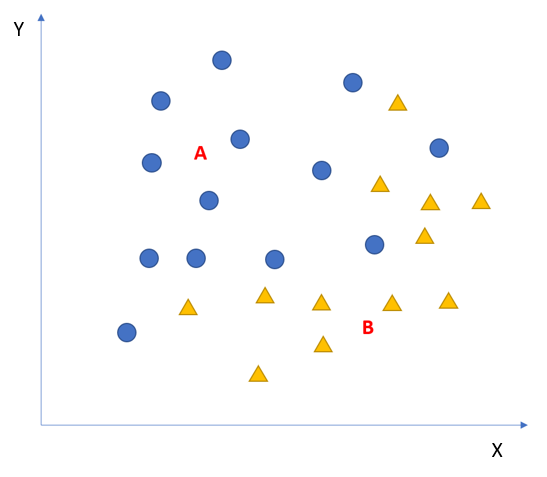
KNN 알고리즘은 상당히 간단한 원리로 작동합니다. 새로운 데이터를 예측할 때, 거리를 기반으로 하여 인접한 데이터과 같은 종류로 분류해내는 기법입니다(예를 들어 인접한 데이터가 모두 파랑이라면 나도 파랑). 거리 기반과 스케일링의 의미를 파악해 KNN을 이해해봅시다.

### 6.9.1 거리 기반의 의미 파악하기

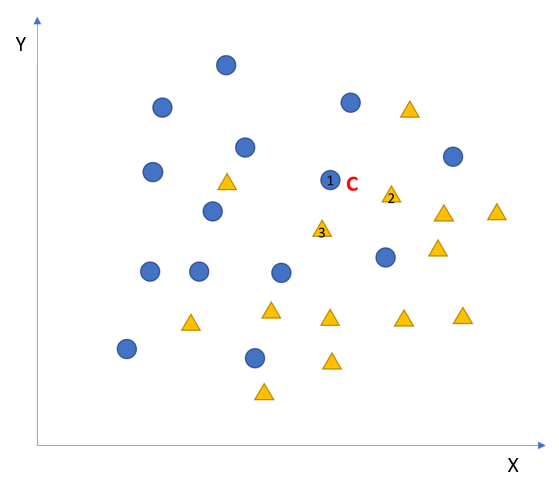
그림으로 간단한 예를 들겠습니다.



위의 그림은 우리가 모델링을 위해 가지고 있는 데이터의 예시입니다. 독립변수는 X, Y로 2개가 있고, 종속변수는 도형의 모양입니다. 즉 동그라미인지 세모인지를 예측해야 하는 상황입니다. 이때 새로운 데이터 A와 B가 다음 위치에 놓여있다고 가정합니다.



이런 상황에서 여러분들은 A와 B를 각각 어떤 도형이라고 예측하겠습니까? 직관적으로 A는 원 주위에 있기 때문에 원형으로, 같은 이유로 B는 세모로 예측할 수 있습니다. 이를 수학적으로 다룬다면 A와 B 각각에서 다른 데이터와의 거리를 계산하고, 각각에 가까이에 있는 데이터를 참고하여 결정하게 됩니다. 이번에는 조금 복잡한 그림을 보겠습니다.

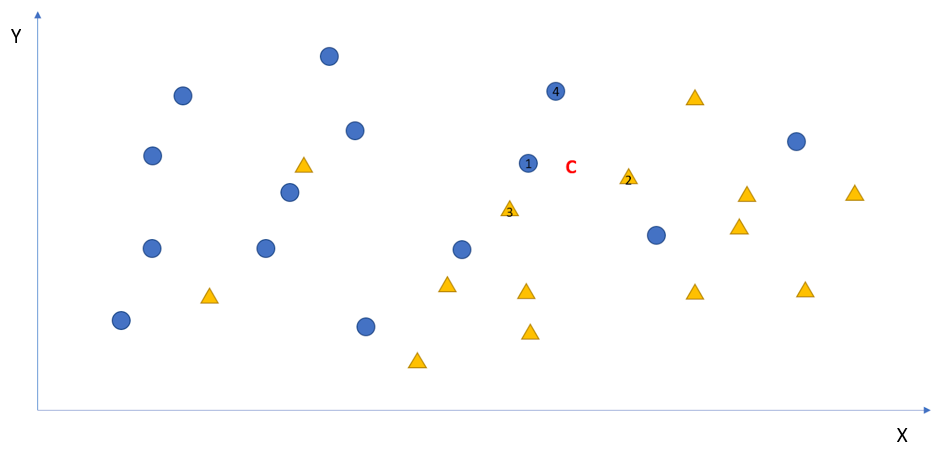


C를 어느 쪽으로 분류하는 게 좋을까요? 원과 세모가 뒤섞인 한 가운데에 있기 때문에 좀처럼 결정하기가 쉽지 않습니다. KNN 알고리즘에서는 이런 모호함을 정리하는 데 특정 이웃의 개수를 지정하여 판단합니다. 인접한 도형 3개는 1, 2, 3번입니다. 가장 가까운 이웃 하나만으로 판단하면 가장 가까운 이웃은 원형(1)이기 때문에 C는를 원형으로 분류됩니다. 인접한 이웃 2개로 고려한다면 어떨까요? 1은 원형이고 2는 세모이기 때문에 동점이 되는데, 사이킷런 라이브러리에서는 이런 경우 랜덤으로 분류합니다. 고려하는 이웃을 3개로 늘린다면 원형이 1개, 세모가 2개로 세모로 분류하게 됩니다.

이렇게 판단에 반영할 가까운 이웃 수를 지정해주어야 하는데, 이것이 KNeighborsClassifier() 함수의 매개변수 n\_neighbors입니다. 이제는 K-최근접 이웃이 왜 거리 기반 모델인지 명확히 이해가 되었을 겁니다.

### 6.9.2 스케일링

이번에는 스케일링과 관련된 상황을 살펴보겠습니다. 위의 그림들은 X와 Y의 스케일이 비슷합니다. 만약 X의 스케일이 Y보다 2배가량 크다면, 다음과 같은 그림이 될 겁니다.



기존의 그림을 가로로 2배 늘려봤습니다. 이 상태에서 C와 가까운 이웃을 따져보면 앞에서와는 다른 결과가 나옵니다. 두 번째로 가까웠던 세로(2)가 멀어지면서 앞에서는 사용하지 않았던 동그라미(4)가 더 가깝습니다. C을 중심으로 근접 이웃 3개는 1(동그라미), 3(세모), 4(동그라미)이므로 이제 C는 원형으로 분류됩니다. 이러한 이유로 KNN에서는 각 변수의 스케일이 매우 중요합니다. 그래서 일관적인 크기로 통일시키는 스케일링을 사용하는 겁니다.

### 6.9.3 동점일 때 처리

동점을 알아봅시다. 2개 이웃을 고려할 때 세모와 원형이 1개씩으로 동점일 때 최종 예측값은 랜덤으로 결정된다고 했습니다. 사실 동점 상황에서 랜덤으로 결정이 되어도 대세에는 큰 지장이 없는 경우가 대부분이나, 이런 동점 문제를 해결하려면 몇 가지 방법이 있습니다.

1. 고려할 이웃의 수를 항상 홀수로 유지하는 겁니다. 우리는 매개변수 튜닝 파트에서 1~20까지의 숫자를 사용했지만, 이를 1, 3, 5, 7, 9, ...처럼 홀수 개의 이웃만 고려한다면 동점 문제를 해결할 수 있습니다.
2. 가중치를 주는 방법입니다. KNN에서는 weights라는 매개변수가 있는데, 이를 이용하면 동점일 때 그 거리가 더 가까운 쪽으로 결정하도록 설정할 수 있습니다.

### 6.9.4 K-최근접 이웃 알고리즘의 계산 특징

마지막으로 K-최근접 이웃 알고리즘의 계산에 관련된 특징을 하나 말씀드리겠습니다. K-최근접 이웃은 예측할 데이터와 기존 데이터의 거리만 계산하면 되기 때문에, 모델링에서 fit() 과정은 기존의 데이터 위치를 스크린샷 하는 정도일뿐, 별다른 학습을 하지는 않습니다. 예를 들어 선형 회귀나 로지스틱 회귀는 fit() 함수를 사용하여 모델을 학습시키면, 선형 모델에 대한 수식을 만들어내 예측을 사용합니다. 하지만 KNN은 이러한 과정이 필요 없습니다. 새 데이터가 오기 전까지는 굳이 거리를 계산할 일이 없기 때문에, fit() 함수를 사용하는 것은 단지 트레이닝 데이터의 위치를 스크린샷하는 정도의 역할 뿐입니다. 그리고 predict() 함수를 사용하여 예측을 하면, 이때 새로운 데이터와 트레이닝 데이터 (학습셋과 시험셋) 사이의 거리들을 계산하고 이를 기반으로 예측하는 겁니다. 그래서 fit() 과정은 순식간에 끝나지만 데이터 규모가 클 때는 predict()에서 상당한 시간이 걸릴 수도 있습니다.

## 학습 마무리

#### 되짚어보기

6.1 와인에 대한 정보를 사용하여 와인의 등급을 예측하는 모델을 만들어봅니다.

6.2 판다스, 넘파이, 맷플롯립, 시본 라이브러리를 임포트했습니다. 프로젝트에 쓸 예제 데이터셋을 불러옵니다.

6.3 데이터에서 결측치가 있음을 확인했습니다. 그 밖에 통계적인 정보도 간단히 살펴보았습니다. KNN에서는 변수의 스케일이 중요하게 작용하는데, 데이터셋에서 변수의 스케일들이 상당히 다르다는 점을 확인할 수 있었습니다.

6.4 와인이 몇 가지 등급을 갖는지 확인했습니다.

6.5 중위값으로 결측치를 처리했습니다

6.6 거리 기반 모델을 사용하기 때문에 스케일링을 통해 변수들의 스케일을 맞춰주었습니다

6.7 K-최근접 이웃 알고리즘을 사용하여 와인을 3개의 등급으로 분류하는 모델을 만들었습니다. 그 결과 88% 정확도를 얻었습니다.

6.8 K-최근접 이웃은 이웃을 몇 개로 고려하냐에 따라 결과가 달라집니다. 여기서는 최적의 이웃 수를 찾는 방법을 학습했습니다.



타이타닉 데이터를 KNN으로 분류하여 결과를 비교해보세요. 다른 스케일러를 사용하여 결과가 어떻게 달라지는지도 비교해봅시다.

#### 핵심 용어 정리

1. **K-최근접 이웃** : 거리 기반으로 데이터를 분류하는 알고리즘으로, 3개 이상의 분류도 가능합니다.
2. **아웃라이어** : 평균치에서 크게 벗어나는 데이터를 의미합니다.
3. **스케일링** : 독립 변수의 범위를 동일한 수준으로 만드는 데 사용되는 방법입니다.

| 표준화 스케일링 | 평균이 0이되고, 표준편차가 1이 되도록 데이터를 고르게 분포시키는 데 사용 |
| --- | --- |
| 로버스트 스케일링 | 데이터에 아웃라이어가 존재하고, 그 영향력을 그대로 유지하고 싶을 때 사용 |
| 최소-최대 스케일링 | 데이터 분포의 특성을 최대한 그대로 유지하고 싶을 때 사용 |
| 정규화 스케일링 | 행 기준의 스케일링이 필요할 때 사용하나, 실제로 거의 사용하지 않음 |

#### 새로운 함수와 라이브러리

1. **pandas.Series.value\_counts()** : 고윳값별 출현 횟수 확인
2. **pandas.Series.unique()** : 고윳값 확인
3. **seaborn.barplot()** : 막대그래프 그리기
4. **pandas.isna()** : 결측치 여부 확인
5. **pandas.DataFrame.sum()** : 컬럼별 합 구하기
6. **pandas.DataFrame.mean()** : 컬럼별 평균 구하기
7. **pandas.DataFrame.dropna()** : 결측치가 있는 행 삭제
8. **스케일러.transform()** : 스케일러에서 학습된 정보를 사용하여 데이터 변환
9. **스케일러.fit\_transform()** : 스케일러 학습 및 변환을 한번에 실행
10. **sklearn.neighbors.KNeighborsClassifier()** : KNN의 분류 알고리즘 객체 생성
11. **seaborn.lineplot()** : 선형 그래프 그리기

## 연습문제

1. 다음 중 결측치에 대한 설명으로 방법으로 옳지 않은 것은?

① 결측치는 데이터가 비어있는 값을 의미한다.

② 결측치는 비어있는 값이므로 숫자 0과 같다.

③ 결측치가 너무 많은 변수는 변수 자체를 제거할 수 있다.

④ 결측치가 많지 않은 경우는 결측치가 있는 행을 제거할 수 있다.

2. 다음 중 스케일링 결과가 0~1 사이로 표현되는 방법은?

① Standard Scaling

② Robust Scaling

③ Min-Max Scaling

3. 다음은 KNN에 대한 설명 중 옳지 않은 것은?

① 예측하려는 데이터 주위에 있는 데이터들을 활용하는 예측 기법이다.

② 모델링 과정에서 참고하려는 이웃 수 K를 설정할 수 있다.

③ 이웃 수 K는 많을수록 좋은 결과를 보여준다.

④ 3개 이상의 다중분류에 활용할 수 있다.

#### 정답 및 해설

1. 2

② 결측치는 비어있는 값이므로 숫자 0과 같다. ← 0도 하나의 입력값입니다. 결측치는 0인지 아닌지도 모르는, 말 그대로 빈 값이므로 0과 같다고 할 수 없습니다.

2. 3

① Standard Scaling ← 평균값(mean) 0을 중심으로 +- 양쪽으로 분포합니다.

② Robust Scaling ← 중위값(median) 0을 중심으로 +- 양쪽으로 분포합니다.

③ Min-Max Scaling ← 최솟값 0, 최댓값 1 사이의 숫자로 분포합니다.

3. 3

③ 이웃 수 K는 많을수록 좋은 결과를 보여준다. ← 이웃 수 K가 너무 많으면 너무 멀리까지 있는 데이터까지 고려하기 때문에, 오히려 예측력이 떨어질 수 있습니다.

1. isnull()도 사용 가능합니다. [↑](#footnote-ref-0)
2. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.DistanceMetric.html#sklearn.neighbors.DistanceMetric [↑](#footnote-ref-1)