# 8장 결정 트리 : 연봉 데이터셋

#### 학습 목표

트리 모델 계열의 근간인 결정 트리를 학습하고, 모델링에서 발생하는 중요한 이슈인 오버피팅 개념과 해결 방법을 알아봅니다.

#### 학습 순서



#### 결정 트리 소개

결정 트리Decision Tree는 관측값과 목푯값을 연결시켜주는 예측 모델로서 나무 모양으로 데이터를 분류합니다. 수많은 트리의 기본 모델based model이 되는 중요 모델입니다. 트리 기반의 모델은 선형 모델과는 전혀 다른 특징을 가지는데, 선형 모델이 각 변수에 대한 기울기값들을 최적화하여 모델을 만들어나갔다면, 트리 모델에서는 각 변수의 특정 지점을 기준으로 데이터를 분류해가며 예측 모델을 만듭니다. 예를 들어 성별이 남자/여자로 나눠서 각 목푯값 평균치를 나눈다거나, 나이를 30세 이상/미만인 두 부류로 나눠서 평균치를 계산하는 방식으로 데이터를 무수하게 쪼개어 나가고, 각 그룹에 대한 예측치를 만들어냅니다.



#### 장단점

| **장점** | **단점** |
| --- | --- |
| * 데이터에 대한 가정이 없는 모델입니다(Non-parametric Model). 예를 들어 선형 모델은 정규분포에 대한 가정이나 독립변수와 종속변수의 선형 관계 등을 가정으로 하는 모델인 반면, 결정 트리는 데이터에 대한 가정이 없으므로 어디에나 자유롭게 적용할 수 있습니다. * 아웃라이어에 영향을 거의 받지 않습니다. * 트리 그래프를 통해서 직관적으로 이해하고 설명할 수 있습니다. 즉 시각화에 굉장히 탁월합니다. | * 트리가 무한정 깊어지면 오버피팅 문제를 야기할 수 있습니다. * 앞으로 배울 발전된 트리 기반 모델들에 비하면 예측력이 상당히 떨어집니다. |

#### 유용한 곳

* 종속변수가 연속형 데이터와 범주형 데이터 모두에 사용할 수 있습니다.
* 모델링 결과를 시각화할 목적으로 가장 유용합니다.
* 아웃라이어가 문제될 정도로 많을 때 선형 모델보다 좋은 대안이 될 수 있습니다.

#### TOP 10 선정 이유

* 예측력과 성능으로만 따지면 결정 트리 모델을 사용할 일은 없습니다. 시각화가 매우 뛰어나다는 유일한 장점이 있을 뿐입니다. 하지만 앞으로 배울 다른 트리 기반 모델을 설명하려면 결정 트리를 알아야 합니다. 트리 기반 모델은 딥러닝을 제외하고는 현재 가장 유용하고 많이 쓰이는 트렌드이기 때문에 트리 모델을 필수로 알아둬야 합니다.

## 

| **예측력과 설명력**  예측력이란 모델 학습을 통해 얼마나 좋은 예측치를 보여주는가를 의미하며, 설명력은 학습된 모델을 얼마나 쉽게 해석할 수 있는지를 뜻합니다. 알고리즘의 복잡도가 증가할수록 예측력은 좋아지나 설명력은 다소 떨어지는 반비례 관계를 보여줍니다. 즉, 단순한 알고리즘일수록 예측력이 상대적으로 떨어질 수 있으나 해석에 용이하며, 복잡한 알고리즘은 예측력이 뛰어난만큼 해석은 어렵습니다.  결정트리와 회귀 분석은 상대적으로 해석이 쉬워 설명력이 높다고 할 수 있으며, 9장부터 배울 알고리즘들은 복잡도가 증가하여 예측력이 높지만 해석이 어렵습니다. 딥러닝 또한 매우 복잡한 알고리즘으로 해석이 어려워서 이를 블랙박스에 비유하기도 합니다.  예측력과 설명력 중 어느 쪽을 택해야 하는지는 상황에 따라 다릅니다. 예를 들어 의학 계열에서 특정 질병의 발병률에 대한 예측모델을 만들 때는, 발병률을 높이거나 억제하는 중요한 요인을 밝히는 데는 설명력이 좋은 알고리즘이 적합할 수 있습니다. 다른 예로 사기거래를 예측하는 모델에서는 요인보다는 더 정확하게 사기거래를 잡아낼 수 있어야 하므로 예측력이 높은 알고리즘이 더 적합할 수 있니다. |
| --- |

## 

## 

## 8.1 문제 정의 : 한눈에 보는 분석 목표

<금토끼의 문제 정의> 하와이 와이키키 해변에서 바라보는 바다는 참으로 아름다웠습니다. 오늘날 이렇게 멋진 곳에서 휴가를 보낼 수 있었던 데에는 아버지의 말씀 덕분입니다. “네게 정말 필요한 공부를 하려무나, 그러면 꼭 꿈을 달성할 수 있을 거야.” 금토끼는 열심히 공부해 오늘날 데이터 분석가가 되었답니다. 뉴욕에서 데이터 분석가로 활약하는 내게 지금 연봉이 적당한지 문득 궁금해졌습니다. 나이, 교육 수준, 결혼상태, 직업, 인종 성별 등 항목에 따른 연봉을 예측하는 알고리즘을 개발해 궁금증을 풀어보기로 했습니다.

| **난이도** | ⭐☆☆ | | |
| --- | --- | --- | --- |
| **알고리즘** | 결정 트리(Decision Tree) | | |
| **데이터셋 파일명** | salary.csv | **종속 변수** | class(연봉 등급) |
| **데이터셋 소개** | 이번 장에서는 연봉 데이터를 사용합니다. 연봉이 $50,000 이상인지 이하인지를 예측하는 것이 목표이며, 종속변수는 class, 독립변수로는 학력, 교육 연수, 혼인 상태, 직업 등이 있습니다. | | |
| **문제 유형** | 분류 | **평가지표** | 정확도 |
| **사용한 모델** | DecisionTreeClassifier | | |
| **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 : 08\_Decision Tree.ipynb | | |

## 8.2 라이브러리 및 데이터 불러오기, 데이터 확인하기

4가지 필수 모듈과 데이터(salary.csv) 파일을 불러오겠습니다.

| import pandas as pd import numpy as np import matplotlib.pyplot as plt import seaborn as sns  file\_url = 'https://media.githubusercontent.com/media/musthave-ML10/data\_source/main/salary.csv' data = pd.read\_csv(file\_url, skipinitialspace = True) # ❶ 데이터셋 읽기 |
| --- |

❶ csv 파일을 불러옵니다. 매개변수 skipinitialspace은 각 데이터의 첫 자리에 있는 공란을 자동 제거합니다. 예를 들어 ‘ Male’를 ‘Male’로 정리합니다. 이번에 사용할 csv 파일에 불필요한 공란이 많아서 적용했습니다.

그럼 head() 함수를 실행해 데이터의 전반적인 모습을 살펴봅시다.

| data.head() # 상위 5행 출력 |
| --- |



종속변수는 class입니다. unique() 함수로 몇 가지 값이 있나 확인하겠습니다.

| data['class'].unique() # 고윳값 확인 |
| --- |

array(['<=50K', '>50K'], dtype=object)

50K 이하와 초과 두 가지만 있습니다. 50K에서 K는 천 단위를 뜻합니다. 즉 $50,000을 기준으로 나뉘어진 데이터입니다.

다음은 info() 함수로 변수별 형태를 보겠습니다.

| data.info() # 변수 특징 출력 |
| --- |

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 48842 entries, 0 to 48841

Data columns (total 14 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 age 48842 non-null int64

1 workclass 46043 non-null object

2 education 48842 non-null object

3 education-num 48842 non-null int64

4 marital-status 48842 non-null object

5 occupation 46033 non-null object

6 relationship 48842 non-null object

7 race 48842 non-null object

8 sex 48842 non-null object

9 capital-gain 48842 non-null int64

10 capital-loss 48842 non-null int64

11 hours-per-week 48842 non-null int64

12 native-country 47985 non-null object

13 class 48842 non-null object

dtypes: int64(5), object(9)

memory usage: 5.2+ MB



❶ object형, 즉 텍스트로 구성된 카테고리형 변수가 많습니다. ❷ Non-Null Count를 보면 결측치가 있는 변수도 몇몇 보입니다. 이번 데이터에서는 이런 카테고리형 변수들과 결측치가 있는 변수들에 대한 처리를 해줄 게 많아 보입니다.

마지막으로 describe() 함수로 통계적 정보를 확인합니다.

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

|  | age | education-num | capital-gain | capital-loss | hours-per-week |
| --- | --- | --- | --- | --- | --- |
| count | 48,842.00 | 48,842.00 | 48,842.00 | 48,842.00 | 48,842.00 |
| mean | 38.64 | 10.08 | 1,079.07 | 87.50 | 40.42 |
| std | 13.71 | 2.57 | 7,452.02 | 403.00 | 12.39 |
| min | 17.00 | 1.00 | 0.00 | 0.00 | 1.00 |
| 25% | 28.00 | 9.00 | 0.00 | 0.00 | 40.00 |
| 50% | 37.000000 | 10 | 0.000000 | 0.000000 | 40 |
| 75% | 48.000000 | 12 | 0.000000 | 0.000000 | 45 |
| max | 90.000000 | 16 | 99999.000000 | 4356.000000 | 99 |

object형의 변수가 많다 보니 변수가 5개만 보입니다. 기본적으로 describe()는 object형의 데이터를 제거하고 통계적 수치를 보여주지만, 매개변수를 이용하여 object형의 데이터까지 보이게 하는 방법도 있습니다.

| data.describe(include = 'all') # object형이 포함된 통계정보 출력 |
| --- |

|  | age | workclass | education | education-num | marital-status | occupation | relationship | race | sex | capital-gain | capital-loss | hours-per-week | native-country | class |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| count | 48,842.00 | 46,043.00 | 48,842.00 | 48,842.00 | 48,842.00 | 46,033.00 | 48,842.00 | 48,842.00 | 48,842.00 | 48,842.00 | 48,842.00 | 48,842.00 | 47985 | 48842 |
| unique | NaN | 8.00 | 16.00 | NaN | 7.00 | 14.00 | 6.00 | 5.00 | 2.00 | NaN | NaN | NaN | 41 | 2 |
| top | NaN | Private | HS-grad | NaN | Married-civ-spouse | Prof-specialty | Husband | White | Male | NaN | NaN | NaN | United-States | <=50K |
| freq | NaN | 33,906.00 | 15,784.00 | NaN | 22,379.00 | 6,172.00 | 19,716.00 | 41,762.00 | 32,650.00 | NaN | NaN | NaN | 43832 | 37155 |
| mean | 38.64 | NaN | NaN | 10.08 | NaN | NaN | NaN | NaN | NaN | 1,079.07 | 87.50 | 40.42 | NaN | NaN |
| std | 13.710510 | NaN | NaN | 2.570973 | NaN | NaN | NaN | NaN | NaN | 7452.019058 | 403.004552 | 12.391444 | NaN | NaN |
| min | 17.000000 | NaN | NaN | 1.000000 | NaN | NaN | NaN | NaN | NaN | 0 | 0 | 1 | NaN | NaN |
| 25% | 28.000000 | NaN | NaN | 9.000000 | NaN | NaN | NaN | NaN | NaN | 0 | 0 | 40 | NaN | NaN |
| 50% | 37 | NaN | NaN | 10 | NaN | NaN | NaN | NaN | NaN | 0 | 0 | 40 | NaN | NaN |
| 75% | 48 | NaN | NaN | 12 | NaN | NaN | NaN | NaN | NaN | 0 | 0 | 45 | NaN | NaN |
| max | 90 | NaN | NaN | 16 | NaN | NaN | NaN | NaN | NaN | 99999 | 4356 | 99 | NaN | NaN |



include 매개변수를 사용해 object형까지도 출력했습니다. ❶ unique, top, freq 행이 추가되었습니다. 새로 추가된 행들은 오로지 object형의 변수들만을 위한 것이고, 기존의 숫자형 변수들는 NaN으로 처리되어 있으며, 반대로 object형의 변수들은 mean, std와 같은 기존의 통계적 정보가 모두 NaN으로 처리되어 있습니다. unique는 각 변수에서 가지고 있는 고유한 value의 숫자입니다. unique() 함수를 사용했을 때와 같은 수치를 보여줍니다. top은 각 변수별로 가장 많이 등장하는 value가 무엇인지를 보여주며, freq는 top에 나와있는 value가 해당 변수에서 총 몇 건인지를 보여줍니다. 예를 들어 workclass 변수는 해당 변수에는 고유한 value가 8 종류이며, Private이 총 33,906 등장합니다(object 변수들을 이런 식으로 살펴보는 방식은 분석에서 크게 유의미하지는 않으니 참고로만 알아두세요). 잠시 후에 각 카테고리형 변수들을 더욱 자세히 확인한 뒤에 전처리하겠습니다.

## 8.3 전처리 : 범주형 데이터

가장 먼저 처리할 변수는 종속변수인 class입니다. 차후 해석에 혼선이 없도록 50K 이하를 0, 초과를 1로 변경하겠습니다.

| data['class'] = data['class'].map({'<=50K': 0, '>50K': 1}) # 숫자로 변환 |
| --- |

<note>

사실 이 데이터의 원본에는 ‘<=50K’의 맨 앞부분에 다음과 같이 공란이 하나 포함되어 있습니다. 그래서 위의 코드가 제대로 작동하지 않을 수 있는데, 앞서 데이터를 불러오는 과정에서 맨 앞자리의 공란을 모두 제거해주었기 때문에 아무런 이슈 없이 처리할 수 있습니다.

</>

### 8.3.1 object형의 변수 정보 확인하기

다음은 독립변수의 범주형 데이터를 다루겠습니다. 이 데이터에는 변수는 13개뿐입니다. 그래서 어떤 변수가 object형인지 쉽게 눈으로 확인할 수 있으나, 변수가 100개 이상이 된다면 다른 접근 방법이 필요합니다. 범주형 데이터가 얼마나 있는지 확인하는 방법을 알아보겠습니다.

변수의 자료형을 확인하는 코드는 다음과 같습니다.

| data['age'].dtype |
| --- |

dtype('int64')

age 변수는 int64형의 데이터입니다(object형은 object로 출력됩니다). 모든 변수의 자료형을 확인하겠습니다.

| for i in data.columns: # ❶ 순회  print(i, data[i].dtype) # ❷ 변수명과 데이터 타입 출력 |
| --- |

age int64

workclass object

education object

education-num int64

marital-status object

occupation object

relationship object

race object

sex object

capital-gain int64

capital-loss int64

hours-per-week int64

native-country object

class object

❶ data.columns으로 변수명을 지정하고 ❷ 해당 변수의 자료형을 dtype으로 확인을 했습니다. 출력 결과는 아직까지 info() 함수와 크게 다르지 않습니다.

object형 변수 이름들을 별도의 리스트로 모아보겠습니다.

| obj\_list = [] # 빈 리스트 for i in data.columns: # 순회  if data[i].dtype == 'object': # ❶ 데이터타입이 object이면  obj\_list.append(i) # ❷ 리스트에 변수 이름을 추가 |
| --- |

❶ dtype이 object인지 아닌지를 확인합니다. object가 맞으면 ❷ 리스트(obj\_list)에 추가합니다.

그럼 obj\_list를 확인하겠습니다.

| obj\_list # 모아진 변수 확인 |
| --- |

['workclass',

'education',

'marital-status',

'occupation',

'relationship',

'race',

'sex',

'native-country']

### 8.3.2 전처리할 변수 선별하기

이번에는 각 변수의 고윳값 개수를 nunique() 함수를 이용하여 확인하겠습니다.

| for i in obj\_list: # 순회  print(i, data[i].nunique()) # ❶ 변수 이름과 고윳값 개수 확인 |
| --- |

workclass 8

education 16

marital-status 7

occupation 14

relationship 6

race 5

sex 2

native-country 41

❶ 변수의 이름과 값의 개수를 같이 볼 수 있도록 print() 안에 둘 다 넣어 출력했습니다.

describe()로 본 정보와 같습니다. 변수 개수가 상당히 많을 때는 describe() 함수 대신 이와 같은 방법을 사용하면 간결하게 확인할 수 있답니다.

범주형 변수를 더미 변수로 바꾸어 활용하는 방법을 기본으로 생각하는 게 좋습니다. 이전 장에서 배웠다시피, 더미 변수를 사용하면 값의 종류만큼 새로운 변수들이 생겨나기 때문에, 값의 종류가 수백 개면 그만큼 많은 변수가 생겨납니다. 필요하다면 수백 개의 변수를 감수하고서 모델링을 해야 하지만 꼭 필요하지 않다면 변수 수를 줄일 방법을 강구할 필요가 있습니다.

값의 종류가 10개 미만인 변수는 그대로 두고, 10개 이상인 변수만 확인하여 조치할지를 검토하겠습니다.

위의 코드에 if절을 추가해 unique가 10개 이상인 변수들만 다시 추려보겠습니다.

| for i in obj\_list: # 순회  if data[i].nunique() >= 10: # 변수의 고윳값이 10보다 크거나 같으면  print(i, data[i].nunique()) # 변수명과 고윳값 개수 출력 |
| --- |

education 16

occupation 14

native-country 41

세 가지 변수(education, occupation, native-country)가 출력되었군요. 각각을 살펴보고 각기 다른 방법으로 처리하는 연습을 하겠습니다.

### 8.3.3 education 변수 처리

우선 education 변수의 정보를 value\_counts()로 살펴봅시다.

| data['education'].value\_counts() # 고윳값 출현빈도 확인 |
| --- |

HS-grad 15784

Some-college 10878

Bachelors 8025

Masters 2657

Assoc-voc 2061

11th 1812

Assoc-acdm 1601

10th 1389

7th-8th 955

Prof-school 834

9th 756

12th 657

Doctorate 594

5th-6th 509

1st-4th 247

Preschool 83

Name: education, dtype: int64

미국의 교육 시스템에 대한 정보라 완전하게 이해하기는 어려울 겁니다. 초중고에 속하는 1~12학년까지의 값과, Bachelors나 Masters 같은 학위 이름들이 있습니다. 이 경우는 범주형 변수지만 서열화가 가능하기 때문에 비교적 다루기 쉬운 편입니다. 예를 들어 초등학교를 가장 낮은 숫자로, 박사학위를 가장 높은 숫자로 나타낼 수 있습니다.

기본적으로 범주형 변수를 숫자로 대체시킬 때는 주의를 해야 합니다. 5.4절 ‘전처리 : 카테고리 변수 변환하기(더미 변수와 원-핫 인코딩)’에서 더미 변수를 설명할 때 말씀드렸죠? 하지만 지금과 같은 경우는 각 값들의 서열이 명백하기 때문에, 숫자로 바꿔주어도 전혀 문제가 없습니다. 숫자로 바꾸는 방법은 앞에서 class를 0과 1로 바꾸어주었을 때처럼 map() 함수를 활용하여 변경하면 되는데, 공교롭게도 해당 데이터에는 이에 대한 변수가 이미 준비되어 있습니다. 잠시 head() 함수로 확인한 결과물을 다시 보겠습니다.

|  | age | workclass | education | education-num | marital-status | occupation | relationship | race | sex | capital-gain | capital-loss | hours-per-week | native-country | class |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 25.00 | Private | 11th | 7.00 | Never-married | Machine-op-inspct | Own-child | Black | Male | 0.00 | 0.00 | 40.00 | United-States | <=50K |
| 1 | 38.00 | Private | HS-grad | 9.00 | Married-civ-spouse | Farming-fishing | Husband | White | Male | 0.00 | 0.00 | 50.00 | United-States | <=50K |
| 2 | 28.00 | Local-gov | Assoc-acdm | 12.00 | Married-civ-spouse | Protective-serv | Husband | White | Male | 0.00 | 0.00 | 40.00 | United-States | >50K |
| 3 | 44.00 | Private | Some-college | 10.00 | Married-civ-spouse | Machine-op-inspct | Husband | Black | Male | 7,688.00 | 0.00 | 40.00 | United-States | >50K |
| 4 | 18.00 | NaN | Some-college | 10.00 | Never-married | NaN | Own-child | White | Female | 0.00 | 0.00 | 30.00 | United-States | <=50K |



❶ education-num 변수를 보니 education 정보를 숫자로 표현한 것 같습니다. 기대하는 바처럼 서열순인지 확인은 해봐야겠죠?

education-num에 어떤 숫자들이 들어있는지 unique() 함수를 써서 알아봅니다. 변수가 제대로 서열화되었는지를 넘파이의 sort()를 사용하여 오름차순으로 정리해 확인하겠습니다.

| np.sort(data['education-num'].unique()) # 고윳값을 오름차순으로 확인 |
| --- |

array([ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16],

dtype=int64)

1에서부터 16까지의 숫자가 있습니다. 각 숫자가 어떤 education값과 매칭되는지를 확인할 차례입니다. 기대하는 바는 일대일 매핑인데, 실제로 그렇지 않을 가능성도 염두에 두고 확인을 해야 합니다.

1에 대한 매핑을 확인합시다. education-num이 1인 데이터만 모아서 그중 education 변수에 어떤 값들이 있는지를 확인하면 됩니다. 한 번에 한 줄씩 코드를 늘려가며 최대한 쉽게 설명하겠습니다.

우선 education-num이 1인지 확인합니다.

| data['education-num'] == 1 # 값이 1인지 확인 - True/False로 출력됨 |
| --- |

0 False

1 False

2 False

3 False

4 False

...

48837 False

48838 False

48839 False

48840 False

48841 False

Name: education-num, Length: 48842, dtype: bool

1이면 True, 아니면 False입니다. 이 정보를 가지고 인덱싱을 하여, True인 경우만 불러오겠습니다(즉 1인 경우).

| data[data['education-num'] == 1] # 값이 1인 (True인) 행만 필터링 |
| --- |

|  | age | workclass | education | education-num | marital-status | occupation | relationship | race | sex | capital-gain | capital-loss | hours-per-week | native-country | class |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 779 | 64.00 | Private | Preschool | 1.00 | Married-civ-spouse | Handlers-cleaners | Husband | Asian-Pac-Islander | Male | 0.00 | 0.00 | 40.00 | Philippines | 0 |
| 818 | 21.00 | Private | Preschool | 1.00 | Never-married | Farming-fishing | Not-in-family | White | Male | 0.00 | 0.00 | 25.00 | Mexico | 0 |
| 1029 | 57.00 | NaN | Preschool | 1.00 | Separated | NaN | Not-in-family | White | Male | 0.00 | 0.00 | 40.00 | United-States | 0 |
| 1059 | 31.00 | Private | Preschool | 1.00 | Never-married | Handlers-cleaners | Not-in-family | Amer-Indian-Eskimo | Male | 0.00 | 0.00 | 25.00 | United-States | 0 |
| 1489 | 19.00 | Private | Preschool | 1.00 | Never-married | Farming-fishing | Not-in-family | White | Male | 0.00 | 0.00 | 36.00 | Mexico | 0 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 48079 | 31 | State-gov | Preschool | 1 | Never-married | Other-service | Not-in-family | White | Male | 0 | 0 | 24 | United-States | 0 |
| 48316 | 40 | Private | Preschool | 1 | Married-civ-spouse | Other-service | Husband | White | Male | 0 | 1672 | 40 | Mexico | 0 |
| 48505 | 40 | Private | Preschool | 1 | Never-married | Other-service | Not-in-family | White | Female | 0 | 0 | 20 | United-States | 0 |
| 48640 | 46 | Private | Preschool | 1 | Married-civ-spouse | Machine-op-inspct | Other-relative | Black | Male | 0 | 0 | 75 | Dominican-Republic | 0 |
| 48713 | 36 | Private | Preschool | 1 | Divorced | Other-service | Not-in-family | Other | Male | 0 | 0 | 72 | Mexico | 0 |

83 rows × 14 columns

data['education-num'] 결과는 0아니면 1입니다. 그 결과를 1과 비교해 참인 경우, 즉 education-num이 1인 경우만 불러왔습니다. 이게 판다스에서 가장 기본적으로 필터링하는 방법이오니 잘 알아둡시니다.

다음은 이 안에서 education이 어떤 종류가 있는지를 확인하기 위하여 해당 변수를 인덱싱하고 unique() 함수를 쓰겠습니다.

| data[data['education-num'] == 1]['education'].unique() # education-num이 1인 데이터들의 education 고윳값 확인 |
| --- |

array(['Preschool'], dtype=object)

결과를 보니 education-num이 1이면 모두 Preschool입니다. 위 코드는 다음과 같이 해석하면 됩니다.



이번에는 모든 숫자에 대해 확인하겠습니다.

| for i in np.sort(data['education-num'].unique()): # 순회  print(i, data[data['education-num'] == i]['education'].unique()) # education-num의 고윳값 별 education의 고윳값 확인 |
| --- |

1 ['Preschool']

2 ['1st-4th']

3 ['5th-6th']

4 ['7th-8th']

5 ['9th']

6 ['10th']

7 ['11th']

8 ['12th']

9 ['HS-grad']

10 ['Some-college']

11 ['Assoc-voc']

12 ['Assoc-acdm']

13 ['Bachelors']

14 ['Masters']

15 ['Prof-school']

16 ['Doctorate']

그리고 print 부분에는 해당 education-num의 숫자와, 그에 응하는 education 고윳값들을 동시에 보여주기 위해서 이 둘을 같이 넣어줍니다.

완전히 우리가 기대했던 형태의 결과입니다. 낮은 숫자부터 높은 숫자까지, 점점 고학력순으로 나열되어 있습니다. 우리가 하려던 작업이 이미 별도의 변수로 마련이 되어 있습니다. (중복 정보이므로) 별도의 작업 없이 기존 변수를 drop() 함수로 제거시킵니다.

| data.drop('education', axis=1, inplace= True) # 변수 제거 |
| --- |

### 8.3.4 occupation 변수 처리

다음은 occupation 변수를 살펴보겠습니다.

| data['occupation'].value\_counts() # 고윳값 출현 빈도 확인 |
| --- |

Prof-specialty 6172

Craft-repair 6112

Exec-managerial 6086

Adm-clerical 5611

Sales 5504

Other-service 4923

Machine-op-inspct 3022

Transport-moving 2355

Handlers-cleaners 2072

Farming-fishing 1490

Tech-support 1446

Protective-serv 983

Priv-house-serv 242

Armed-Forces 15

Name: occupation, dtype: int64

다양한 직업군이 표기되는데, 이미 비슷한 직업군끼리는 묶인 상태로 정리된 데이터로 보입니다. 예를 들어 Farming과 fishing이 하나의 이름으로 묶여있습니다. 유사한 직업군끼리 묶여있지 않았다면 이를 묶는 작업을 하려했으나 이미 정리가 되어 있고, 각 직업 간의 서열이라고 할 만한 부분도 딱히 정의할 수가 없습니다. 더미 변수로 변환을 하면 변수 14개가 생기겠군요. 그정도면 감당할 수 있는 수준이니 별도의 작업을 하지 않고 나중에 더미 변수로 처리하겠습니다.

### 8.3.5 native-country 변수 처리

마지막으로 native-country 변수를 살펴보겠습니다.

| data['native-country'].value\_counts() # 고윳값 출현 빈도 확인 |
| --- |

United-States 43832

Mexico 951

Philippines 295

Germany 206

Puerto-Rico 184

Canada 182

El-Salvador 155

India 151

Cuba 138

England 127

China 122

South 115

Jamaica 106

Italy 105

Dominican-Republic 103

Japan 92

Guatemala 88

Poland 87

Vietnam 86

Columbia 85

Haiti 75

Portugal 67

Taiwan 65

Iran 59

Nicaragua 49

Greece 49

Peru 46

Ecuador 45

France 38

Ireland 37

Thailand 30

Hong 30

Cambodia 28

Trinadad&Tobago 27

Outlying-US(Guam-USVI-etc) 23

Yugoslavia 23

Laos 23

Scotland 21

Honduras 20

Hungary 19

Holand-Netherlands 1

Name: native-country, dtype: int64

수많은 값이 있습니다. 그리고 특이사항으로 United-States가 압도적으로 큰 비중을 차지합니다. 그럼 해당 변수를 처리하는 여러 방법을 살펴봅시다. 우선 이 경우는 United-States가 약 90%를 차지하고 있기 때문에, 아주 단순하게는 United-States 이외의 국가들을 하나로 묶어 Others 같은 이름으로 변경하는 방식도 가능합니다. 데이터가 간소화되는 장점이 있으나, 그만큼 정보가 줄어드는 단점이 있습니다. 만약 해당 예측 모델에서 United-States가 아닌 국가 사이에 큰 차이가 없다면 이 방법을 써도 무방합니다. 또 다른 방법으로는 비슷한 값들끼리 묶는 겁니다. 예를 들어 지역별로 묶는 거죠. 즉, North America, South America, Asia 등과 같이 구분할 수 있습니다. 이 방법 또한 해당 지역에 속한 국가끼리 어느 정도 유사성을 보여야 무리가 없습니다. 여기서 말하는 유사성은, 종속 변수의 값에 대한 유사성을 의미합니다. 즉 국가별로 class에 대한 평균값을 내었을 때, 만약 Asia 국가들이 다소 비슷한 수치를 보여준다면 이렇게 묶는 데에 큰 무리가 없을 겁니다. 유사성은 처음에 말한 United-States 이외의 국가들끼리 묶는 때도 필요하니 국가별로 class값의 평균을 확인해봅시다.

국가별 평균을 확인해야 하기 때문에 groupby() 함수를 사용하여 국가별로 묶어주겠습니다.

<함수/>

groupby() 함수

| data.groupby('native-country').연산() |
| --- |

groupby 뒤의 괄호 안에 묶어줄 변수 이름을 넣으시면 되는데, 여기까지만 쓰면 아무런 아웃풋도 없습니다. 반드시 그 뒤에 추가 연산을 붙여야 합니다. 판다스에서 제공하는 sum(), mean(), std() 같은 명령어를 사용할 수 있습니다.

</>

우리는 국가별로 class의 평균값을 볼 것이기 때문에 mean() 함수를 사용하겠습니다.

| # 아래 코드는 참고용으로, 실행할 필요가 없습니다.  data.groupby('native-country').mean() # 그룹별 평균 계산 |
| --- |

여기까지만 코드를 작성해도 원하는 아웃풋을 볼 수 있으나, 국가가 너무 많아서 쉽게 눈에 들어오는 상황은 아닙니다. 같은 지역에 있는 국가들끼리 비슷한 수준의 class 평균값을 가지는지를 확인하는 것이 목적이므로 지역 혹은 class 기준으로 정렬시키겠습니다. 이 데이터에는 지역에 대한 변수가 없어, class 기준으로 정렬시키겠습니다.

| data.groupby('native-country').mean().sort\_values('class') # 그룹별 평균 계산 후 class 기준으로 오름차순 정렬 |
| --- |

| **native-country** | **age** | **education-num** | **capital-gain** | **capital-loss** | **hours-per-week** | **class** |
| --- | --- | --- | --- | --- | --- | --- |
| **Holand-Netherlands** | 32.000000 | 10.000000 | 0.000000 | 2205.000000 | 40.000000 | 0.000000 |
| **Guatemala** | 32.090909 | 6.306818 | 167.875000 | 18.113636 | 38.715909 | 0.034091 |
| **Outlying-US(Guam-USVI-etc)** | 38.826087 | 10.043478 | 0.000000 | 76.608696 | 41.347826 | 0.043478 |
| **Columbia** | 39.458824 | 9.258824 | 125.364706 | 65.247059 | 39.929412 | 0.047059 |
| **Dominican-Republic** | 37.970874 | 7.320388 | 1064.456311 | 39.029126 | 41.621359 | 0.048544 |
| **Mexico** | 33.635121 | 6.026288 | 415.954784 | 32.656151 | 40.213460 | 0.049422 |
| **Nicaragua** | 36.285714 | 9.000000 | 138.653061 | 69.938776 | 36.938776 | 0.061224 |
| **El-Salvador** | 33.380645 | 6.722581 | 392.761290 | 36.367742 | 36.361290 | 0.070968 |
| **Trinadad&Tobago** | 39.259259 | 8.962963 | 116.185185 | 156.518519 | 38.888889 | 0.074074 |
| **Vietnam** | 34.616279 | 9.616279 | 604.802326 | 86.372093 | 37.976744 | 0.081395 |
| **Laos** | 35.217391 | 8.826087 | 125.434783 | 75.652174 | 39.391304 | 0.086957 |
| **Peru** | 36.434783 | 9.826087 | 39.804348 | 40.173913 | 36.543478 | 0.086957 |
| **Honduras** | 35.050000 | 8.350000 | 75.300000 | 152.000000 | 35.650000 | 0.100000 |
| **Puerto-Rico** | 39.864130 | 8.331522 | 335.347826 | 104.777174 | 39.016304 | 0.108696 |
| **Haiti** | 38.600000 | 8.533333 | 297.626667 | 66.200000 | 36.920000 | 0.120000 |
| **Ecuador** | 37.666667 | 9.244444 | 328.288889 | 0.000000 | 39.266667 | 0.133333 |
| **Jamaica** | 37.141509 | 9.811321 | 495.915094 | 17.801887 | 39.160377 | 0.141509 |
| **Scotland** | 46.761905 | 10.190476 | 246.571429 | 0.000000 | 41.666667 | 0.142857 |
| **Thailand** | 37.666667 | 10.833333 | 243.266667 | 49.500000 | 44.700000 | 0.166667 |
| **South** | 38.095652 | 10.765217 | 1678.208696 | 134.269565 | 42.852174 | 0.173913 |
| **Portugal** | 41.238806 | 6.761194 | 203.402985 | 0.000000 | 42.238806 | 0.179104 |
| **Poland** | 42.758621 | 10.068966 | 471.919540 | 70.390805 | 37.689655 | 0.195402 |
| ❶ **United-States** | 38.698690 | 10.168667 | 1089.626529 | 88.789743 | 40.440774 | 0.243977 |
| **Cuba** | 46.355072 | 9.391304 | 470.543478 | 62.717391 | 40.101449 | 0.246377 |
| **Hong** | 34.233333 | 10.433333 | 500.800000 | 255.466667 | 40.266667 | 0.266667 |
| **Germany** | 38.601942 | 10.927184 | 1062.058252 | 61.456311 | 40.815534 | 0.281553 |
| **Philippines** | 39.633898 | 10.722034 | 1508.823729 | 88.522034 | 39.620339 | 0.288136 |
| **China** | 41.852459 | 11.262295 | 1407.737705 | 170.655738 | 38.262295 | 0.295082 |
| **Ireland** | 38.486486 | 9.783784 | 494.783784 | 96.189189 | 42.432432 | 0.297297 |
| **Hungary** | 50.368421 | 10.947368 | 490.315789 | 87.789474 | 37.947368 | 0.315789 |
| **Cambodia** | 36.892857 | 9.392857 | 697.464286 | 194.821429 | 42.035714 | 0.321429 |
| **Italy** | 45.419048 | 8.790476 | 797.895238 | 52.638095 | 40.942857 | 0.323810 |
| **Canada** | 44.049451 | 10.692308 | 1233.505495 | 131.159341 | 40.406593 | 0.346154 |
| **Japan** | 37.358696 | 11.423913 | 1874.586957 | 59.445652 | 42.282609 | 0.347826 |
| **Yugoslavia** | 40.478261 | 9.391304 | 916.391304 | 0.000000 | 40.217391 | 0.347826 |
| **Greece** | 45.836735 | 9.755102 | 1131.448980 | 174.857143 | 46.897959 | 0.367347 |
| **England** | 40.527559 | 11.110236 | 1076.551181 | 108.913386 | 41.937008 | 0.370079 |
| **Iran** | 38.372881 | 12.186441 | 1490.169492 | 124.406780 | 42.949153 | 0.372881 |
| **Taiwan** | 34.184615 | 13.276923 | 1948.600000 | 121.476923 | 39.400000 | 0.400000 |
| **India** | 38.364238 | 12.536424 | 3196.390728 | 115.417219 | 41.423841 | 0.410596 |
| ❷ **France** | 40.315789 | 12.000000 | 417.473684 | 37.052632 | 42.789474 | 0.421053 |

우선 다른 국가들을 Others로 묶어줄 수 있는지 ❶ United-States를 먼저 찾아보겠습니다. class 평균값은 약 0.24입니다. 그리고 다른 국가들은 United-States의 평균값보다 크거나 작아서 전혀 비슷하지 않은 양상입니다. Others로 묶는 방법은 좋지 않아 보입니다. 그럼 지역별로 묶어줄 수 있는지 보겠습니다. class 평균이 가장 높은 ❷ France 중심으로 유럽 국가를 찾아보겠습니다. 극단적으로 다른 경우만 몇 개 발견하면 해당 방법이 좋지 않음이 증명되기 때문에, 수치가 낮은 쪽에 유럽 국가가 있는지를 보는 게 빠릅니다. 가장 낮게는 Holand가 있지만 관측치가 1개밖에 안 되므로 고려하지 않아도 될 것 같고, 그다음으로는 Portual이 0.18 정도로 France와 상당한 차이를 보입니다. 다른 지역의 예로 Cuba와 Dominican Republic도 각각 0.25와 0.05로 큰 차이를 보입니다. 물론 다른 많은 나라는 지역별로 비슷한 양상을 보이기도 하지만 큰 차이를 보는 경우가 많으므로 지역별로 묶는 방법은 적합하지 않습니다.

그럼 더미 변수를 사용하지 않고 각 국가명을 숫자로 변환하여 하나의 변수를 그대로 유지하게 하는 건 어떨까요? 제가 처음 더미 변수를 설명할 때, 카테고리 데이터를 무작정 숫자로 치환하여 모델링하는 방법은 결코 좋지 않다고 설명드렸습니다. 그러나 이러한 방법이 허용되는 경우가 있는데, 바로 트리 기반의 모델을 사용할 때입니다. 트리 기반의 모델은 연속된 숫자들도 연속적으로 받아들이기보다 일정 구간을 나누어 받아들이기 때문에, 트리가 충분히 깊어지면 카테고리 변수를 숫자로 바꾼다고해도 큰 문제가 없습니다.

카테고리 데이터를 숫자로 치환하는 여러 방법이 있습니다. 기본적으로는 랜덤하게 번호를 부여하는 겁니다. 예를 들어 United-States는 1, Peru는 2, Guatemala는 3 등 임의의 번호를 붙여주는 거죠. 단순한 라벨링으로 생각하면 됩니다. 또 다른 방법은 우리가 value\_counts() 함수로 확인한 숫자들을 부여하는 방법입니다. 이 방법은 종속 변수인 class가 이민자 수가 많은 국가인지 적은 국가인지에 따라 유의미한 차이를 보인다면 더 유용한 변수로 활용이 될 것이고, 그렇지 못하더라도 최소한 라벨링 효과는 가지고 가게 됩니다. 단, 라벨링의 목적만으로 이 방법을 사용할 때에는 같은 값을 가지는 국가가 없는지 확인해야 하며, 이 데이터의 경우는 동수인 국가들이 있어서 적합하지 않습니다.

마지막은 우리가 groupby()로 확인했던 class의 평균값을 넣어주는 방법입니다. 이는 조금 극단적인 방법인데, 예상하려고 하는 목푯값을 독립변수의 일환으로 반영하기 때문입니다. 즉, 답을 거의 밀어넣다 시피해서 모델링하는지라 오버피팅 문제가 발생할 수 있습니다. 오버피팅 문제는 잠시 후 매개변수 튜닝에서 더 구체적으로 다루겠습니다.

‘native-country’별로 class의 평균값을 구하는 코드를 작성하겠습니다.

| country\_group = data.groupby('native-country').mean()['class'] # 그룹별 class의 평균값을 계산하여 저장 |
| --- |

앞서 작성한 코드에서 정렬에 사용한 sort\_values() 함수를 빼고, 인덱싱을 추가해 class값만 취했습니다. 그리고 아웃풋을 country\_group이라는 이름으로 저장했습니다.

이렇게 평균값을 구해놓은 판다스 시리즈를 기존 데이터에 새로운 변수로 붙여넣어야 합니다. join()이나 merge()를 사용할 수 있는데 이번에는 merge()를 사용해서 데이터를 합치겠습니다.

<함수/>

| 함수 | 설명 |
| --- | --- |
| merge() | a.merge(b) # 데이터 a와 데이터 b를 합침 |

</>

국가 이름이 키값이 되어야 하는데 현재 country\_group에서 국가 이름은 변수가 아닌 인덱스에 자리하고 있습니다. reset\_index()를 써서 변수로 뺄 수 있습니다.

<함수/>

| **함수** | **설명** |
| --- | --- |
| reset\_index() | 판다스 데이터 프레임과 시리즈에서, 인덱스에 들어있는 정보를 별도의 변수로 빼옵니다.  sample = pd.DataFrame({'var\_1':[3,5,7], 'var\_2':[2,4,6]}, index = ['a','b','c']) # ❶  sample.reset\_index() # ❷  var\_A var\_B var\_C  0 a 10 2  1 b 30 7  2 c 50 8  3 d 70 9  4 e 90 0  ❶ 임의의 데이터 프레임 2개를 만들었습니다.. ❷ merge()를 사용하여 데이터를 합쳤고, 자동적으로 두 데이터프레임에 공통으로 존재하는 변수인 var\_A를 기준으로 합쳐집니다. |

</>

reset\_index()를 써서 인덱스를 변수로 빼주겠습니다.

| country\_group = country\_group.reset\_index() # 인덱스를 변수로 불러냄 |
| --- |

이제 country\_group()을 확인하면 다음과 같은 데이터프레임을 확인할 수 있습니다.

| country\_group # 데이터 확인 |
| --- |

|  | native-country | class |
| --- | --- | --- |
| 0 | Cambodia | 0.321429 |
| 1 | Canada | 0.346154 |
| 2 | China | 0.295082 |
| 3 | Columbia | 0.047059 |
| 4 | Cuba | 0.246377 |
| 5 | Dominican-Republic | 0.048544 |
| 6 | Ecuador | 0.133333 |
| 7 | El-Salvador | 0.070968 |
| 8 | England | 0.370079 |
| 9 | France | 0.421053 |
| 10 | Germany | 0.281553 |
| 11 | Greece | 0.367347 |
| 12 | Guatemala | 0.034091 |
| 13 | Haiti | 0.120000 |
| 14 | Holand-Netherlands | 0.000000 |
| 15 | Honduras | 0.100000 |
| 16 | Hong | 0.266667 |
| 17 | Hungary | 0.315789 |
| 18 | India | 0.410596 |
| 19 | Iran | 0.372881 |
| 20 | Ireland | 0.297297 |
| 21 | Italy | 0.323810 |
| 22 | Jamaica | 0.141509 |
| 23 | Japan | 0.347826 |
| 24 | Laos | 0.086957 |
| 25 | Mexico | 0.049422 |
| 26 | Nicaragua | 0.061224 |
| 27 | Outlying-US(Guam-USVI-etc) | 0.043478 |
| 28 | Peru | 0.086957 |
| 29 | Philippines | 0.288136 |
| 30 | Poland | 0.195402 |
| 31 | Portugal | 0.179104 |
| 32 | Puerto-Rico | 0.108696 |
| 33 | Scotland | 0.142857 |
| 34 | South | 0.173913 |
| 35 | Taiwan | 0.400000 |
| 36 | Thailand | 0.166667 |
| 37 | Trinadad&Tobago | 0.074074 |
| 38 | United-States | 0.243977 |
| 39 | Vietnam | 0.081395 |
| 40 | Yugoslavia | 0.347826 |

이제 merge()를 사용하여 두 데이터를 합치겠습니다.

| data = data.merge(country\_group, on = 'native-country', how='left') # data와 country\_group을 native\_country 기준으로 결합 (left join) |
| --- |

data 국가 이름은 기준으로 하여 country\_group을 붙이는 형태이기 때문에 data를 메인으로 삼아 merge()를 실행했습니다. 키값은 native-country, 결합 형태는 left 조인(3.1.8절 ‘데이터프레임 합치기’)으로 처리해줍니다.

그럼 데이터가 잘 합쳐졌는지 data를 불러서 확인해봅시다.

| data # 데이터 확인 |
| --- |

|  | **age** | **workclass** | **education-num** | **marital-status** | **occupation** | **relationship** | **race** | **sex** | **capital-gain** | **capital-loss** | **hours-per-week** | **native-country** | **class\_x** | **class\_y** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 25 | Private | 7 | Never-married | Machine-op-inspct | Own-child | Black | Male | 0 | 0 | 40 | United-States | 0 | 0.243977 |
| **1** | 38 | Private | 9 | Married-civ-spouse | Farming-fishing | Husband | White | Male | 0 | 0 | 50 | United-States | 0 | 0.243977 |
| **2** | 28 | Local-gov | 12 | Married-civ-spouse | Protective-serv | Husband | White | Male | 0 | 0 | 40 | United-States | 1 | 0.243977 |
| **3** | 44 | Private | 10 | Married-civ-spouse | Machine-op-inspct | Husband | Black | Male | 7688 | 0 | 40 | United-States | 1 | 0.243977 |
| **4** | 18 | NaN | 10 | Never-married | NaN | Own-child | White | Female | 0 | 0 | 30 | United-States | 0 | 0.243977 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **48837** | 27 | Private | 12 | Married-civ-spouse | Tech-support | Wife | White | Female | 0 | 0 | 38 | United-States | 0 | 0.243977 |
| **48838** | 40 | Private | 9 | Married-civ-spouse | Machine-op-inspct | Husband | White | Male | 0 | 0 | 40 | United-States | 1 | 0.243977 |
| **48839** | 58 | Private | 9 | Widowed | Adm-clerical | Unmarried | White | Female | 0 | 0 | 40 | United-States | 0 | 0.243977 |
| **48840** | 22 | Private | 9 | Never-married | Adm-clerical | Own-child | White | Male | 0 | 0 | 20 | United-States | 0 | 0.243977 |
| **48841** | 52 | Self-emp-inc | 9 | Married-civ-spouse | Exec-managerial | Wife | White | Female | 15024 | 0 | 40 | United-States | 1 | 0.243977 |
| 48842 rows × 14 columns | | | | | | | | | | | | | | |

합쳐지기는 했는데 문제가 하나 발생했군요, class\_x와 class\_y라는 변수가 생겼습니다. 기존의 data에 class 변수가 있고, country\_group에도 class 변수가 있다 보니, 컬럼 이름이 겹치는 바람에 판다스에서 자동적으로 변경시켜버렸습니다. 우리는 class\_y를 국가명 대신 사용해줄 것이기 때문에 기존 native-country를 삭제하고 ➝ class\_y를 해당 이름(native-country)으로 바꾸고 ➝ class\_x를 class로 다시 변경해주겠습니다.

| data.drop('native-country', axis=1, inplace=True) # 변수 제거 data = data.rename(columns= {'class\_x': 'class', 'class\_y': 'native-country'}) # 변수 이름 변환 |
| --- |

여기서 컬럼 이름을 바꾸는데 rename() 함수를 사용했습니다. rename() 함수는 위에서 보시는 것처럼 변경할 대상(columns 혹은 index)을 지정해주시고 구체적인 변경 사항은 딕셔너리 형태로 매핑해주면 됩니다.

<함수/>

| 함수명 | 설명 |
| --- | --- |
| rename() | 데이터프레임의 변수나 인덱스 이름을 변경할 수 있습니다. 변경하려는 이름과 변경 후의 이름을 딕셔너리 형태로 넣어주시면 됩니다.  <코드/>  # ❶ 데이터프레임 생성  sample = pd.DataFrame({'var\_1':[3,5,7], 'var\_2':[2,4,6]}, index = ['a','b','c'])  sample.rename(columns = {'var\_1':'var\_new'}) # ❷ 이름 변경  </>  <출력/>  var\_new var\_2  a 3 2  b 5 4  c 7 6  </>  ❶ 변수이름이 var\_1, var\_2인 데이터프레임을 만듭니다.  ❷ var\_1을 var\_new로 변경합니다. 이때 변경 전후의 이름은 딕셔너리 형태로 입력합니다. 변수가 아닌 인덱스 이름을 변경하려면 columns 대신 index를 사용합니다. |

</>

## 8.4 전처리 : 결측치 처리 및 더미 변수 변환

결측치 처리합시다. 먼저 결측치가 있는 변수들과 그 비율을 확인하겠습니다.

| data.isna().mean() # 결측치 비율 확인 |
| --- |

age 0.000000

workclass 0.057307

education-num 0.000000

marital-status 0.000000

occupation 0.057512

relationship 0.000000

race 0.000000

sex 0.000000

capital-gain 0.000000

capital-loss 0.000000

hours-per-week 0.000000

native-country 0.017546

class 0.000000

dtype: float64

총 3개의 변수에서 결측치가 보입니다. workclass와 occupation은 약 5%, native-country는 1.7% 정도입니다. 우선 native-country는 각 국가별 class의 평균값으로 대체한 상태입니다. 원래는 카테고리형 변수이기 때문에 평균치로 채우는 것이 불가능하지만 지금은 숫자로 바뀌었고 그 숫자가 마침 class의 평균값이기 때문에, mean()이나 median()으로 결측치를 채우는 것도 가능합니다. 또는 완전 별개의 숫자를 지정하여 채워주는 방법도 가능합니다. 보통 결측치를 숫자로 채워줄 때 -9나 -99와 같은 임의의 숫자를 사용합니다. 이렇게 임의의 숫자를 채워주는 것은 트리 기반 모델에서는 큰 문제가 없으나 선형 모델에서는 데이터의 왜곡을 불러오니 주의해야 합니다. 지금은 두 가지 방법 모두 가능한데, 여기에서는 -99를 채우겠습니다.

| data['native-country'] = data['native-country'].fillna(-99) # 결측치를 -99로 대체 |
| --- |

workclass와 occupation 변수는 모두 카테고리형 변수이기 때문에 평균치로 해결할 수가 없습니다. 이 경우는 특정 텍스트를 채워주거나, dropna()로 해당 라인을 제거해야 하는데, 우선 각 컬럼의 value\_counts() 출력물을 보고 판단하겠습니다.

| data['workclass'].value\_counts() # 고윳값별 출현빈도 확인 |
| --- |

Private 33906 ❶

Self-emp-not-inc 3862

Local-gov 3136

State-gov 1981

Self-emp-inc 1695

Federal-gov 1432

Without-pay 21

Never-worked 10

Name: workclass, dtype: int64

workclass에서는 ❶ Private 비율이 압도적입니다. 특정 값이 대부분을 차지하는 경우라면 해당 값으로 결측치를 채워주는 방법도 무난합니다. Private이 70% 정도라서 조금 아쉬운 부분은 있지만, 연습 차원에서 이 값으로 결측치를 채워넣겠습니다.

| data['workclass'] = data['workclass'].fillna('Private') # 결측치를 Private으로 대체 |
| --- |

다음으로 occupation 컬럼을 확인하겠습니다.

| data['occupation'].value\_counts() # 고윳값별 출현빈도 확인 |
| --- |

Prof-specialty 6172

Craft-repair 6112

Exec-managerial 6086

Adm-clerical 5611

Sales 5504

Other-service 4923

Machine-op-inspct 3022

Transport-moving 2355

Handlers-cleaners 2072

Farming-fishing 1490

Tech-support 1446

Protective-serv 983

Priv-house-serv 242

Armed-Forces 15

Name: occupation, dtype: int64

이번에는 어떤 특정값이 압도적으로 많다고 하기가 어렵습니다. 이 경우는 위와 같은 방법은 알맞지 않습니다. 별도의 텍스트 ‘Unknown’으로 채워보겠습니다.

| data['occupation'] = data['occupation'].fillna('Unknown') # 결측치를 Unknown으로 대체 |
| --- |

이제 모든 결측치를 해결했으니, 카테고리 데이터를 더미 변수로 변환하겠습니다.

| data = pd.get\_dummies(data, drop\_first=True) # 더미 변수로 변환 |
| --- |

## 8.5 모델링 및 평가하기

모델링에 앞서 이번에도 데이터를 훈련셋과 시험셋으로 나누어보겠습니다.

| from sklearn.model\_selection import train\_test\_split # 임포트 X\_train, X\_test, y\_train, y\_test = train\_test\_split(data.drop('class', axis=1), data['class'], test\_size = 0.4, random\_state = 100) # 훈련셋/시험셋 분리 |
| --- |

이번에는 데이터가 비교적 크기 때문에 test\_size를 0.4로 높여서 나누었습니다. 결정 트리 모델은 사이킷런의 tree에서 불러올 수 있는데, 분석 목적에 따라 연속형 변수를 위한 DecisionTreeRegressor와 범주형 변수를 위한 DecisionTreeClassifier로 나뉘어졌습니다.



여기서는 0과 1을 예측해야 하므로 DecisionTreeClassifier를 사용하겠습니다.

| from sklearn.tree import DecisionTreeClassifier # 임포트 |
| --- |

모델링 및 예측 방법은 이전과 같습니다.

| model = DecisionTreeClassifier() # 모델 객체 생성 model.fit(X\_train, y\_train) # 학습 pred = model.predict(X\_test) # 예측 |
| --- |

X\_train과 y\_train으로 학습해, X\_test에 대한 예측값을 만들었습니다. 그럼 이 예측값에 대한 정확도를 보겠습니다.

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

0.8134309259354047

약 81%의 예측률을 보여줍니다. 그럼 이제 결정 트리가 어떤 원리로 작동되는지 살펴보고, 이를 기반으로 매개변수를 튜닝해 더 나은 예측값을 얻을 수 있는지 보겠습니다.

## 8.6 이해하기 : 결정 트리

결정 트리는 특정 변수에 대한 특정 기준값으로 데이터를 계속 분류해가면서 유사한 그룹으로 묶어내어 예측값을 만드는 알고리즘입니다. 예를 들어 다음과 같은 데이터가 있다고 가정해봅시다.

| 지름 (독립변수) | 종류 (종속변수) |
| --- | --- |
| 6 | 사과 |
| 4.5 | 귤 |
| 4 | 귤 |
| 7 | 사과 |
| 7.5 | 사과 |

사과인지 귤인지를 예측하는 모델을 만든다면, “지름이 5보다 큰가? 혹은 작은가?”라는 질문 하나로 이 둘을 명확하게 구분해 낼 수 있을 겁니다. 트리 모델은 이런 식으로 특정 기준으로 계속 분류해나가는데, 실제 데이터에서는 이보다 훨씬 많은 독립변수가 있고, 대부분은 종속변수를 완전하게 분류해낼 수 있는 값은 없기 때문에, 수많은 분류의 분류를 거듭하게 됩니다.

이번에는 조금 더 복잡한 데이터로 예를 들겠습니다.

| 지름 (독립변수 1) | 무게 (독립변수 2) | 종류 (종속변수) |
| --- | --- | --- |
| 7.5 | 320 | 사과 |
| 5 | 240 | 사과 |
| 7 | 300 | 사과 |
| 6.5 | 230 | 복숭아 |
| 7 | 250 | 복숭아 |
| 6 | 200 | 복숭아 |

이 데이터에서 사과와 복숭아를 구분하는 모델을 고려해봅시다. 우선 지름과 무게 모두 중첩되는 부분이 있기 때문에 아까처럼 손쉽게 둘을 구분하는 기준점은 찾기 어렵습니다만, 몇 번의 분류 작업을 거치면 이 또한 잘 구분해낼 수 있습니다.



위의 그림과 같이 지름 7을 기준으로 2개의 그룹으로 나누면 “사과 2, 복숭아 1”, “사과 1, 복숭아 2”처럼 뒤섞여있습니다. 이를 다시 무게를 기준으로 나누면 사과와 복숭아를 완전히 분리해낼 수 있습니다.

이와 같은 원리 때문에 좋은 모델을 만들려면 첫 번째 분류에 사용할 변수를 선정과 기준점 정하기가 정말 중요합니다. 그리고 두 번째 분류 과정에서도 각 상황에서 최적의 변수와 기준점을 채택해야 합니다. 그럼 결정 트리는 어떤 기준으로 평가하여 찾아내는지 알아보겠습니다.

### 8.6.1 분류 결정 트리

분류와 회귀는 각기 다른 로직이 적용되는데, 우선 분류부터 설명하겠습니다. 위의 예시 다이어그램에서 보이는 각 회색 박스를 노드Node라고 부르는데, DecisionTreeClassifier는 각 노드의 순도가 가장 높은 방향으로 분류를 합니다. 순도는 한 노드 안에 여러 종류가 아닌 한 종류의 목푯값만 있는 상태에 대한 지표입니다. 예를 들어 노드 안에 사과 3개와 복숭아 3개가 있으면 두 과일이 반씩 있기 때문에 순도가 매우 낮은 반면, 한 노드 안에 사과 3개와 복숭아 0개가 있다면 순도가 높다고 볼 수 있습니다. 결정 트리는 이러한 순도를 체크하여 가지를 뻗어 나가는데, 이 순도를 평가하는 지표로는 크게 지니 인덱스Gini Index와 교차 엔트로피Cross Entropy가 있습니다.

<용어/>

**지니 인덱스와 교체 엔트로피**

노드의 순도를 평가하는 방법. 노드의 순도가 높을수록 지니 및 엔트로피 값은 낮아집니다.

</>

#### 지니 인덱스

우선 지니 인덱스의 수식은 다음과 같습니다.

지니 인덱스는 각 노드에 대해서 계산되며, p는 노드 안에 특정 아이템의 비율입니다. 예시를 통해 위의 수식을 적용하겠습니다.  
  
‘사과 2, 복숭아 2’일 때를 계산해보겠습니다.

* 각각 50% 이므로, 수식의 우측부분에 대한 계산은 0.52  + 0.52 = 0.5
* 수식 왼쪽 부분을 함께 계산하면 1 - 0.5이므로 0.5

‘사과 1, 복숭아 3’일 때를 계산해보겠습니다.

* 각각 25%, 75%. 수식 우측부분은 0.252 + 0.752 = 0.625
* 수식 왼쪽 부분을 함께 계산하면 1 - 0625이므로 0.375

‘사과 0, 복숭아 4’일 때를 계산해보겠습니다.

* 각각 0%, 100%이므로 우측 수식부분은 02 + 12 = 1
* 수식 왼쪽 부분을 함께 계산하면 1-1 이므로 0

이를 표로 정리하면 다음과 같습니다.

| **예시** | **비율** |  | **지니 인덱스** |
| --- | --- | --- | --- |
| 사과 2개, 복숭아 2개 | 각각 50% | 0.52  + 0.52 = 0.5 | 1 - 0.5이므로 0.5 |
| 사과 1개, 복숭아 3개 | 25%, 75% | 0.252 + 0.752 = 0.625 | 1 - 0625이므로 0.375 |
| 사과 0개, 복숭아 4개 | 0%, 100% | 02 + 12 = 1 | 1-1 이므로 0 |

노드가 한쪽 아이템으로 완전히 분류가 잘된 경우(사과 0, 복숭아 4)는 지니 인덱스가 0이며, 전혀 분류가 안 되어서 반반이 들어 있으면(사과 2, 복숭아 2)는 지니 인덱스가 0.5입니다. 즉, 순도가 높을수록 지니 인덱스는 낮은 값을 보이며, 최댓값은 0.5 최솟값은 0이 나올 수 있습니다. **결정 트리에서는 지니 인덱스가 가장 낮은 값이 나오는 특정 변수의 특정 값을 기준으로 노드를 분류해갑니다.**

#### 교차 엔트로피

다음은 교차 엔트로피 수식입니다.

몇 가지 예를 들어보겠습니다.

‘사과 2, 복숭아 2’일 때를 계산해보겠습니다.

* 사과 50%에 대한 부분 : 0.5 \* Log2(0.5) = 0.5 \* -1 = -0.5
* 복숭아 50%에 대한 부분 : 0.5 \* Log2(0.5) = 0.5 \* -1 = -0.5
* 시그마는 위의 두 부분에 대한 합이므로 -0.5 + -0.5 = -1
* 마지막으로 맨 앞의 - 부호를 반영하면 교차 엔트로피는 1

‘사과 1, 복숭아 3’일 때를 계산해보겠습니다.

* 사과 25%에 대한 부분 : 0.25 \* Log2(0.25) = 0.25 \* -2 = -0.5
* 복숭아 75%에 대한 부분 : 0.75 \* Log2(0.75) = 0.75 \* -0.415037... = 약 -0.31
* 시그마는 위의 두 부분에 대한 합이므로 -0.5 + -0.31 = -0.81
* 마지막으로 맨 앞의 - 부호를 반영하면 교차 엔트로피는 약 0.81

‘사과 0, 복숭아 4’일 때를 계산해보겠습니다.

* 사과 0%에 대한 부분 : 0 \* Log2(0) = 0
* 복숭아 100%에 대한 부분 : 1 \* Log2(1) = 1 \* 0 = 0
* 시그마는 위의 두 부분에 대한 합이므로 -0.5 + -0.5 = 0
* 마지막으로 맨 앞의 - 부호를 반영하면 교차 엔트로피는 0

이를 표로 정리해보겠습니다.

| **예시** | **비율** | **중간 계산** | **교차 엔트로피** |
| --- | --- | --- | --- |
| 사과 2개, 복숭아 2개 | 각각 50% | * 사과 50% : 0.5 \* Log2(0.5) = 0.5 \* -1 = -0.5 * 복숭아 50% : 0.5 \* Log2(0.5) = 0.5 \* -1 = -0.5 * 시그마는 위 두값 대한 합 : -0.5 + -0.5 = -1 | 1 |
| 사과 1개, 복숭아 3개 | 25%, 75% | * 사과 25% : 0.25 \* Log2(0.25) = 0.25 \* -2 = -0.5 * 복숭아 75% : 0.75 \* Log2(0.75) = 0.75 \* -0.415037... = 약 -0.31 * 시그마는 위 두값 대한 합 : -0.5 + -0.31 = -0.81 | 약 0.81 |
| 사과 0개, 복숭아 4개 | 0%, 100% | * 사과 0% : 0 \* Log2(0) = 0 * 복숭아 100% : 1 \* Log2(1) = 1 \* 0 = 0 * 시그마는 위 두값 대한 합 : -0.5 + -0.5 = 0 | 0 |

교차 엔트로피도 마찬가지로 순도가 높을수록 낮은 값을 보이나, 최댓값은 1까지 나올 수 있다는 차이점이 있습니다.

지니 인덱스와 교차 엔트로피 중 어느 로직을 사용하는 것이 더 좋은가에 대한 부분은 딱히 승자가 없이, 둘 다 결정 트리에서 비슷한 성능을 보여줍니다. 사이킷런의 결정 트리에서는 기본값으로 지니 인덱스를 사용합니다(매개변수를 통해 교차 엔트로피로 지정해줄 수 있습니다).

### 5.6.2 회귀 결정 트리

회귀는 우리가 이전에 배운 MSE를 평가 기준으로 이용합니다. 회귀는 연속형 변수를 대상으로 하는 모델이기 때문에, 새로운 데이터를 예로 들겠습니다.

| x | y |
| --- | --- |
| 10 | 98 |
| 20 | 0 |
| 35 | 6 |
| 5 | 44 |
| 15 | 88 |

위의 데이터를 대상으로 결정 트리의 첫 번째 노드가 x >15를 기준으로 나눈다고 한다면, 다음과 같은 두 노드로 분류될 겁니다.

| x | y |  | x | y |
| --- | --- | --- | --- | --- |
| 10 | 98 |  | 20 | 0 |
| 5 | 44 |  | 35 | 6 |
| 15 | 88 |  |  |  |

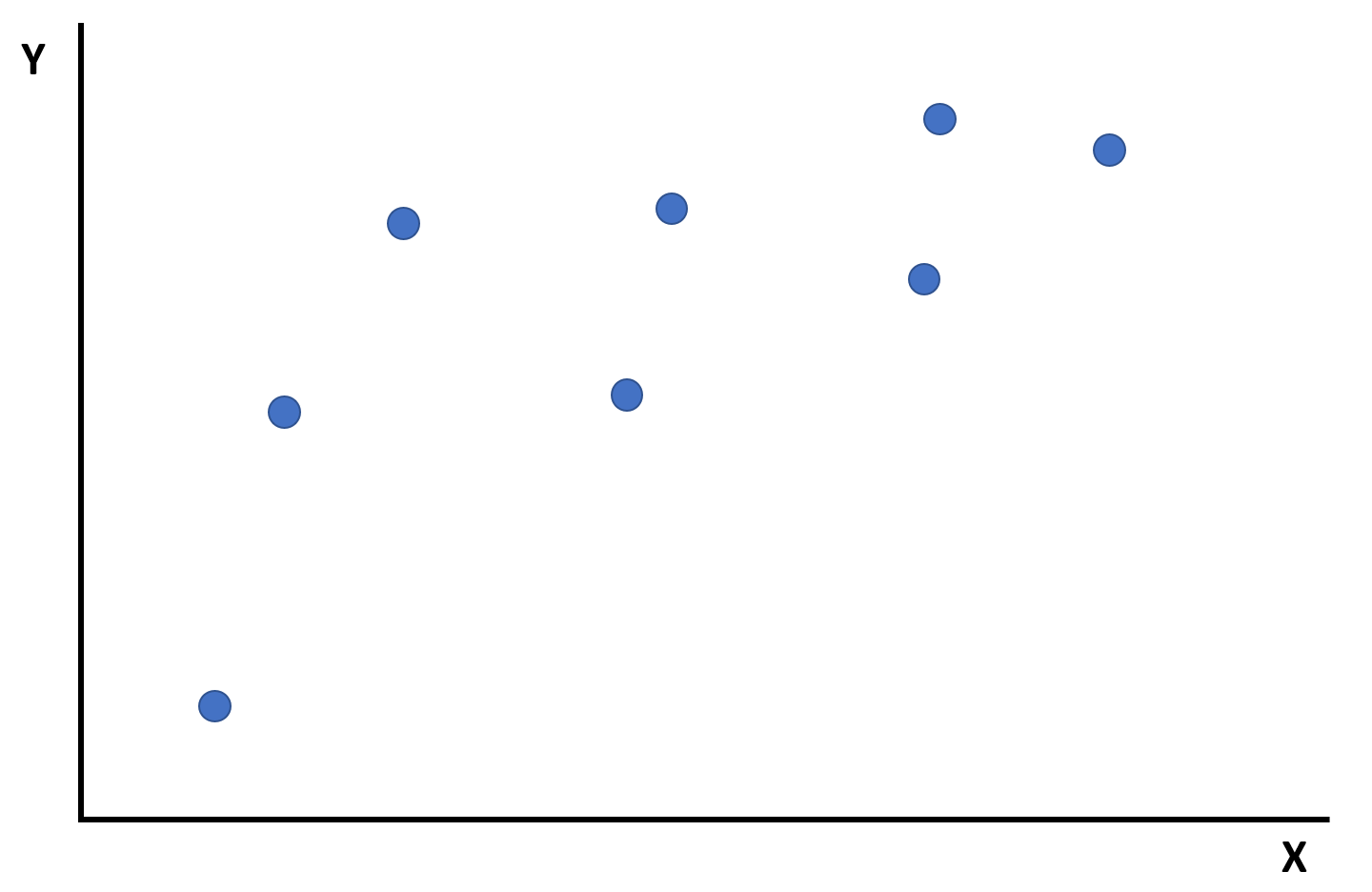
간단하게 오른쪽 노드에 대해서 MSE를 계산하겠습니다. 배웠다시피 MSE는 실젯값과 예측값의 차이에 대한 계산입니다. 여기에서의 실젯값은 y값이 되고, 예측값은 오른쪽 노드에서 y의 평균이 됩니다.

* 우선 y의 평균을 구합니다 : 3
* 평균과 각 값의 차를 구합니다 : 0 - 3 = -3, 6 - 3 = 3
* 위의 값에 대한 제곱 후 합을 구합니다 : -32 + 32 = 18
* 위의 값을 관측치 개수만큼 나누어줍니다 : 18 / 2= 9

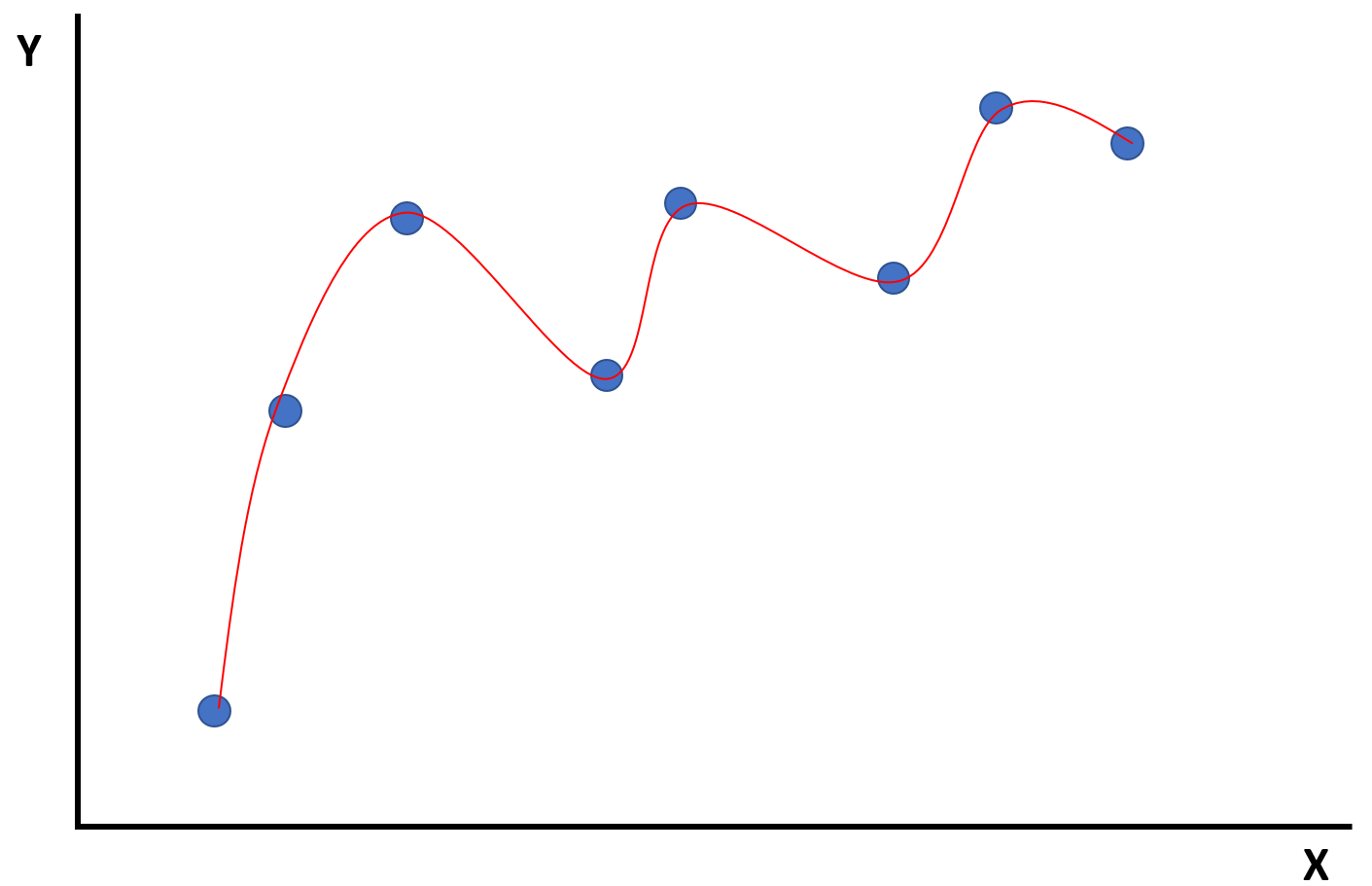
이와 같은 방식으로 MSE를 구하며, **결정 트리 회귀는 가장 낮은 MSE값이 나오도록 노드를 분류해나갑니다**. MSE는 사이킷런의 결정 트리 모델에서 기본값으로 설정된 평가 기준이고, 필요에 따라 매개변수를 이용하여 MSE대신 MAE나 Poisson 등으로 설정할 수도 있습니다.

## 8.7 오버피팅 문제

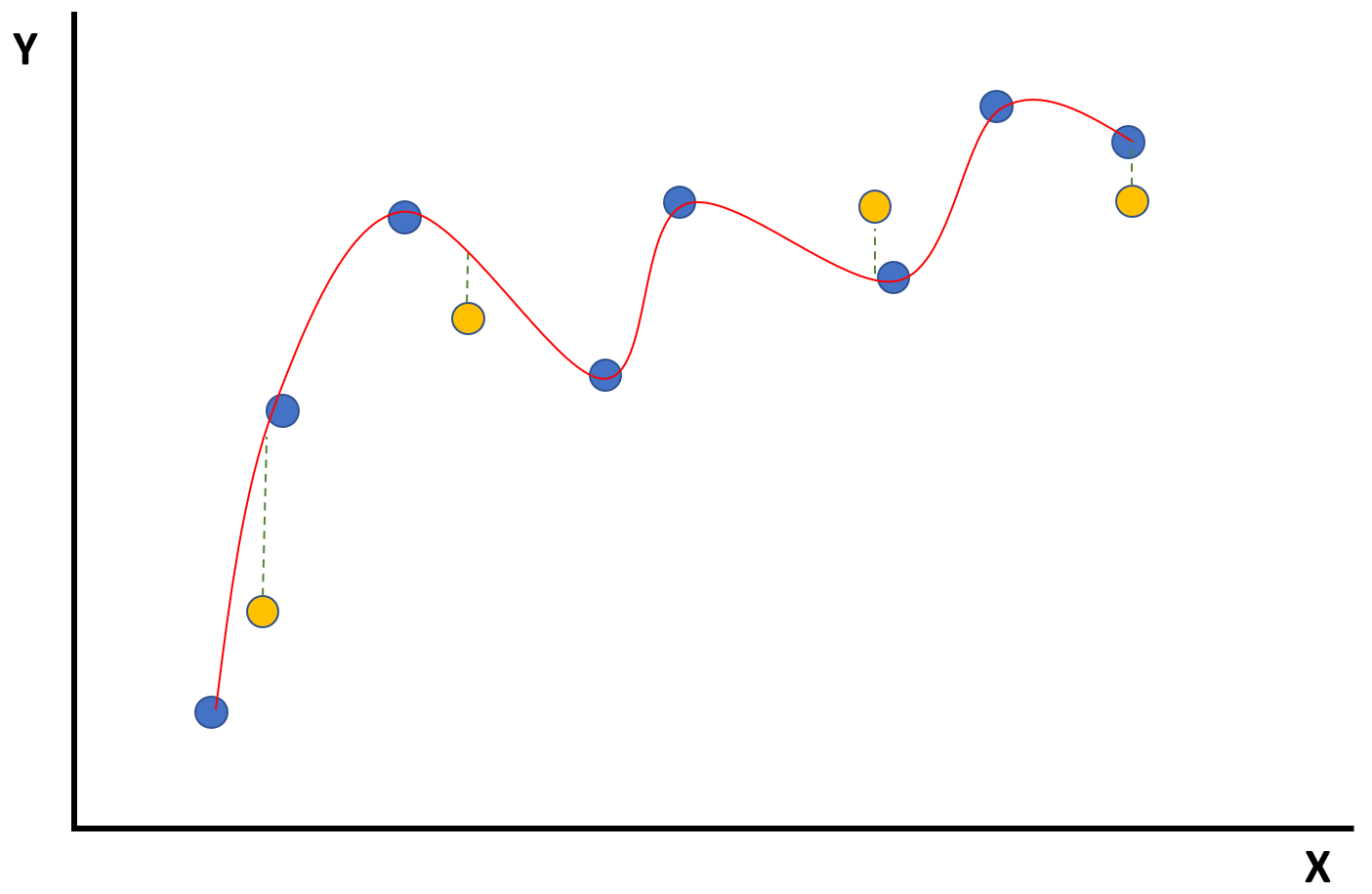
결정 트리는 최대한 정확하게 분류해내기까지 수많은 가지를 뻗어나갈 수 있습니다. 각 마지막 노드에 1개의 관측치만 들어갈 정도로 세밀하게 분류해냈다면 100%의 정확도로 분류가 가능할 것인데, 매우 정확해 보이지만 이런 모델을 좋은 모델이라고 말하기는 어렵습니다. 우리가 훈련셋을 사용하여 지나치게 깊은 트리 모델을 만들면, 시험셋에 대한 예측은 오히려 떨어지는 경향이 있습니다. 이 문제는 비단 결정 트리에서만의 문제가 아니라, 모든 머신러닝 모델에서 발생하는 문제입니다. 아래의 그림을 예로 들어 설명하겠습니다.



독립변수가 1개인 데이터이고, 위의 파란 점들은 훈련셋의 데이터입니다. 만약 위의 데이터를 가지고 아주 정확한 예측 모델을 만든다면 아래의 같은 빨간 곡선 형태가 예측값이 될 겁니다.

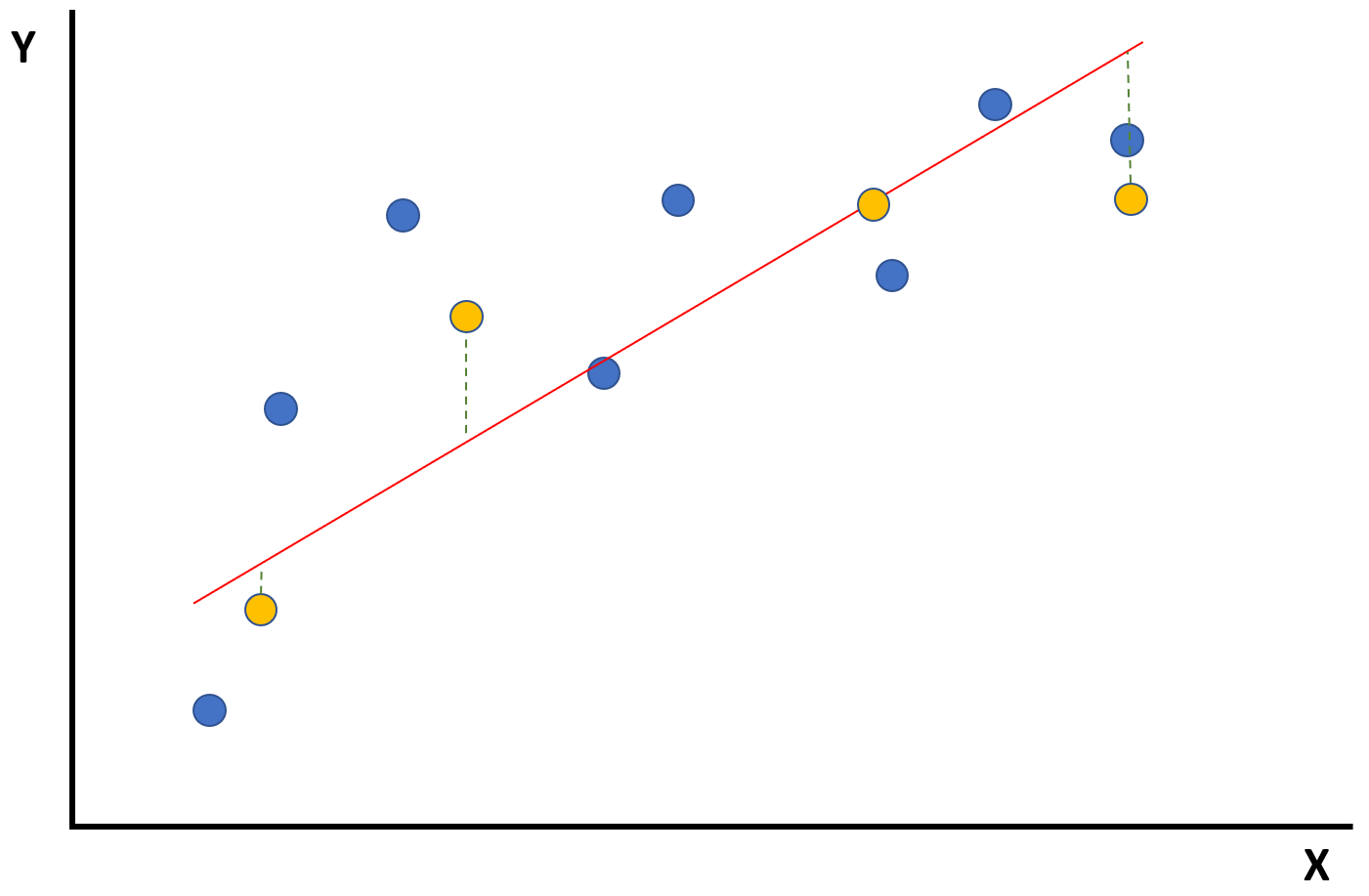


그럼 이 예측 모델을 가지고 시험셋을 예측하면 어떻게 되는지도 보겠습니다.



위 그래프에서의 노란색 원들은 시험셋의 데이터이고 녹색 점선은 예측치와의 실젯값 사이의 오차를 의미합니다. 예측 모델이 훈련셋을 지나치게 잘 예측한다면 위와 같이 새로운 데이터를 예측할 때 큰 오차를 유발할 수 있습니다. 이러한 상황을 바로 오버피팅overfitting, 과적합이라고 합니다.

현실 데이터에는 어느 정도의 오차 범위가 있고 때로는 아웃라이어도 있기 때문에, 훈련셋에 아주 딱 떨어지는 예측 모델을 만드는 것보다는 어느 정도 두루뭉실한 수준으로 예측하는 것이 오히려 새로운 데이터를 예측에는 더 나을 수도 있습니다. 이번에는 동일한 훈련셋으로 만든 아주 단순한 예측 모델을 살펴봅시다.



직선으로 예측하는 모델입니다. 시험셋의 실젯값과 예측 모델에서 상당한 오차가 발생합니다. 이러한 경우를 언더피팅underfitting, 과소적합이라고 합니다. 오버피팅과 언더피팅은 모두 피해야 하는 문제이며, 언더피팅이 더 안 좋은 모델이라 볼 수 있습니다. 시험셋의 예측이 상대적으로 괜찮다고 해도 훈련셋조차 제대로 예측하지 못한다면 그 결과를 신뢰하기가 어렵기 때문입니다. 따라서 언더피팅은 무조건 피해야 하며, 데이터와 모델이 괜찮다면 잘 발생하지도 않습니다. 반면 오버피팅은 머신러닝 알고리즘이 복잡해짐에 따라 빈번하게 발생하며, 완전한 해결도 어렵습니다. 그래서 오버피팅을 완전히 피한다기보다는 줄여나간다는 개념으로 접근하는 것이 좋습니다.

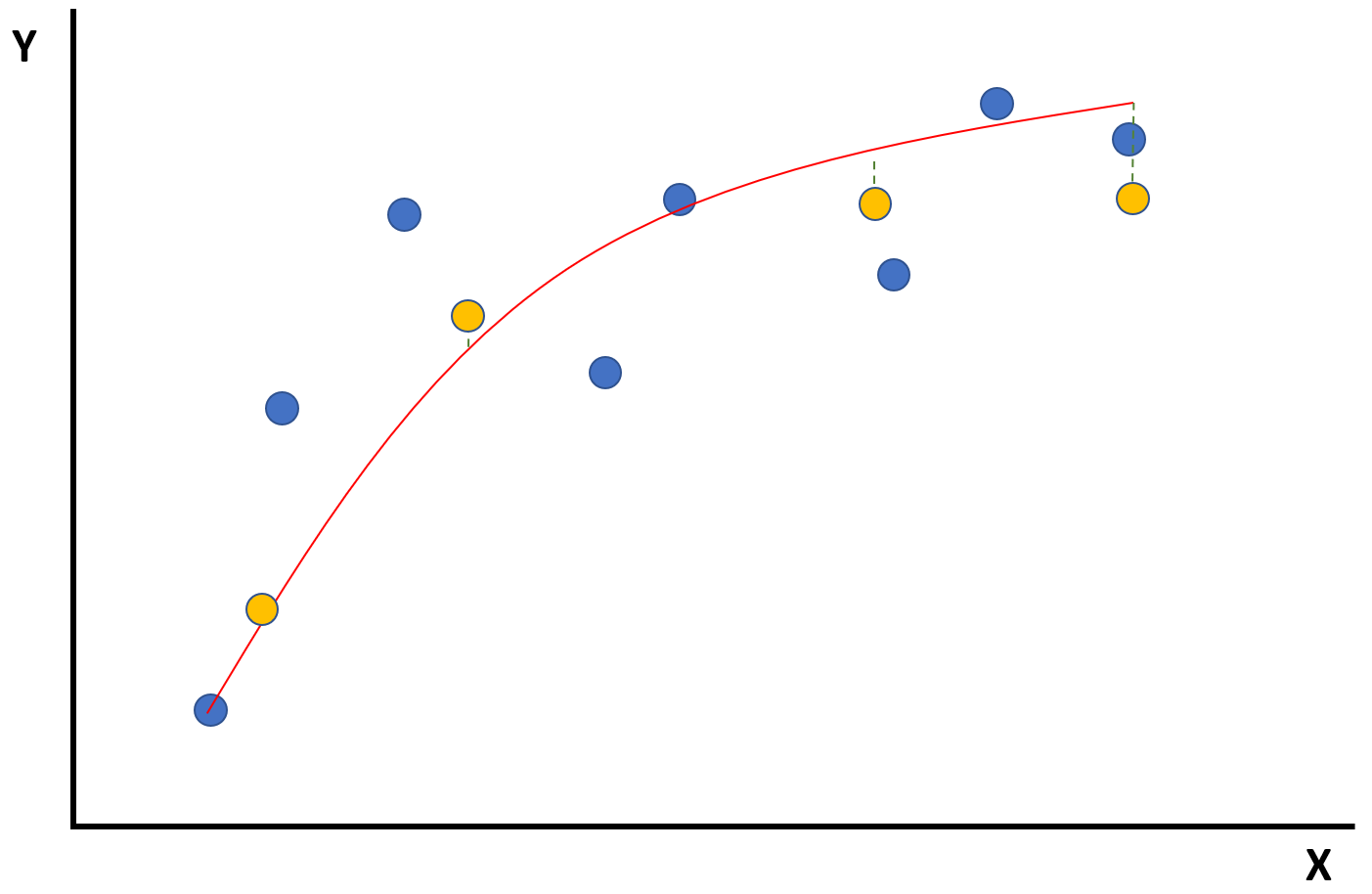
<용어/>

**언더피팅**

과소적합이라고도 하며, 모델이 충분히 학습되지 않아 훈련셋에 대해서도 좋은 예측력을 내지 못하는 상황

</>

그럼 오버피팅과 언더피팅을 해결한 예시도 보겠습니다.



이렇게 너무 과하지도 부족하지도 않은 수준의 예측 모델을 만들어서 훈련셋과 시험셋 모두에서 적절한 수준의 오차를 가져가는 형태가 좋은 모델입니다. 따라서 훈련셋을 지나치게 잘 예측하는 모델의 정확도를 일부러 낮춰 오버피팅을 예방할 필요가 있습니다.

이러한 상황을 설명하는 개념으로 편향-분산 트레이드오프bias-variance tradeoff가 있습니다. 우선 편향은 독립변수와 종속변수를 모델링한 알고리즘이 적절치 못하거나 중요한 부분을 놓쳐서 제대로 된 예측을 하지 못할 때 높아집니다. 예를 들어 앞에서 봤던 언더피팅 예시가 편향이 높은 상황입니다. 반면 분산은 훈련셋에 있는 데이터의 노이즈에 의해 발생하는 오차입니다. 예를 들어 앞서 살펴본 오버피팅은 알고리즘이 훈련셋을 지나치게 정확하게 따라가면서 모든 노이즈를 모델에 포함해 높은 분산이 발생할 수 있습니다. 즉 높은 편향는 언더피팅 문제를, 높은 분산는 오버피팅을 문제를 불러올 수 있습니다. 가장 이상적이게는 편향과 분산 모두가 낮은 모델이겠지만, 현실에서는 기대하기 어렵습니다. 현실에서는 아래 그래프와 같은 형태를 띄는 것이 일반적입니다.



분산이 낮으면 편향이 올라가게 되고, 반대로 편향이 낮아질수록 분산이 올라갑니다. 이 각 오차들의 합이 전체 모델의 에러가 되기 때문에 이 둘의 합이 가장 낮은 중간 지점이 우리가 찾아야 할 최적점입니다. 이를 두고 편향-분산 트레이드오프라 하며, 그래프에서 우측이 오버피팅, 왼쪽이 언더피팅입니다.

## 8.8 매개변수 튜닝

**결정 트리에서는 트리 깊이가 깊어질수록, 즉 수없이 많은 노드를 분류하여 모델을 만들수록 오버피팅 발생 가능성이 높습니다.** 이 문제를 해결할 목적으로 결정 트리에서는 트리의 깊이를 제한하는 매개변수를 제공합니다. 기본값이 None이므로 매개변수를 지정해주지 않으면 최대한 깊은 수준으로 트리를 만들어냅니다. 우선 기본값으로 다시 모델링하고, 훈련셋과 시험셋 각 정확도를 살펴보겠습니다.

| model = DecisionTreeClassifier() # 모델 객체 생성 model.fit(X\_train, y\_train) # 학습 train\_pred = model.predict(X\_train) # 훈련셋 예측 test\_pred = model.predict(X\_test) # 시험셋 예측 print('Train score:', accuracy\_score(y\_train, train\_pred), 'Test score:', accuracy\_score(y\_test, test\_pred)) # 훈련셋, 시험셋의 정확도 평가 |
| --- |

Train score: 0.9780242279474493 Test score: 0.8155806930439679

훈련셋에서 정확도(Train score)는 약 0.98, 시험셋에서 정확도(Test score)는 약 0.81입니다. 훈련셋에서는 정확도가 매우 높지만 시험셋에서는 상대적으로 낮은 수준입니다. 이정도의 격차라면 오버피팅이 발생했다고 볼 수 있고, 이 둘의 격차를 낮추면서 시험셋에서 정확도를 올리는 방향으로 매개변수를 설정해보겠습니다. 트리의 깊이를 지정하는 매개변수는 max\_depth입니다. 우선 5로 지정하여 결과를 보겠습니다.

| model = DecisionTreeClassifier(max\_depth=5) # 모델 객체 생성 model.fit(X\_train, y\_train) # 학습 train\_pred = model.predict(X\_train) # 훈련셋 예측 test\_pred = model.predict(X\_test) # 시험셋 예측 print('Train score:', accuracy\_score(y\_train, train\_pred), 'Test score:', accuracy\_score(y\_test, test\_pred)) # 훈련셋, 시험셋의 정확도 평가 |
| --- |

Train score: 0.8540180856509129 Test score: 0.8499769667809797

앞선 결과보다 Train score가 확 낮아졌고 Test score는 살짝 올라갔습니다. 둘 사이의 차이는 매우 낮아졌습니다. 마지막으로 7로 변경하여 한 번 더 테스트하겠습니다.

| model = DecisionTreeClassifier(max\_depth=7) # 모델 객체 생성 model.fit(X\_train, y\_train) # 학습 train\_pred = model.predict(X\_train) # 훈련셋 예측 test\_pred = model.predict(X\_test) # 시험셋 예측 print('Train score:', accuracy\_score(y\_train, train\_pred), 'Test score:', accuracy\_score(y\_test, test\_pred)) # 훈련셋, 시험셋의 정확도 평가 |
| --- |

Train score: 0.8598532673605187 Test score: 0.8542765009981061

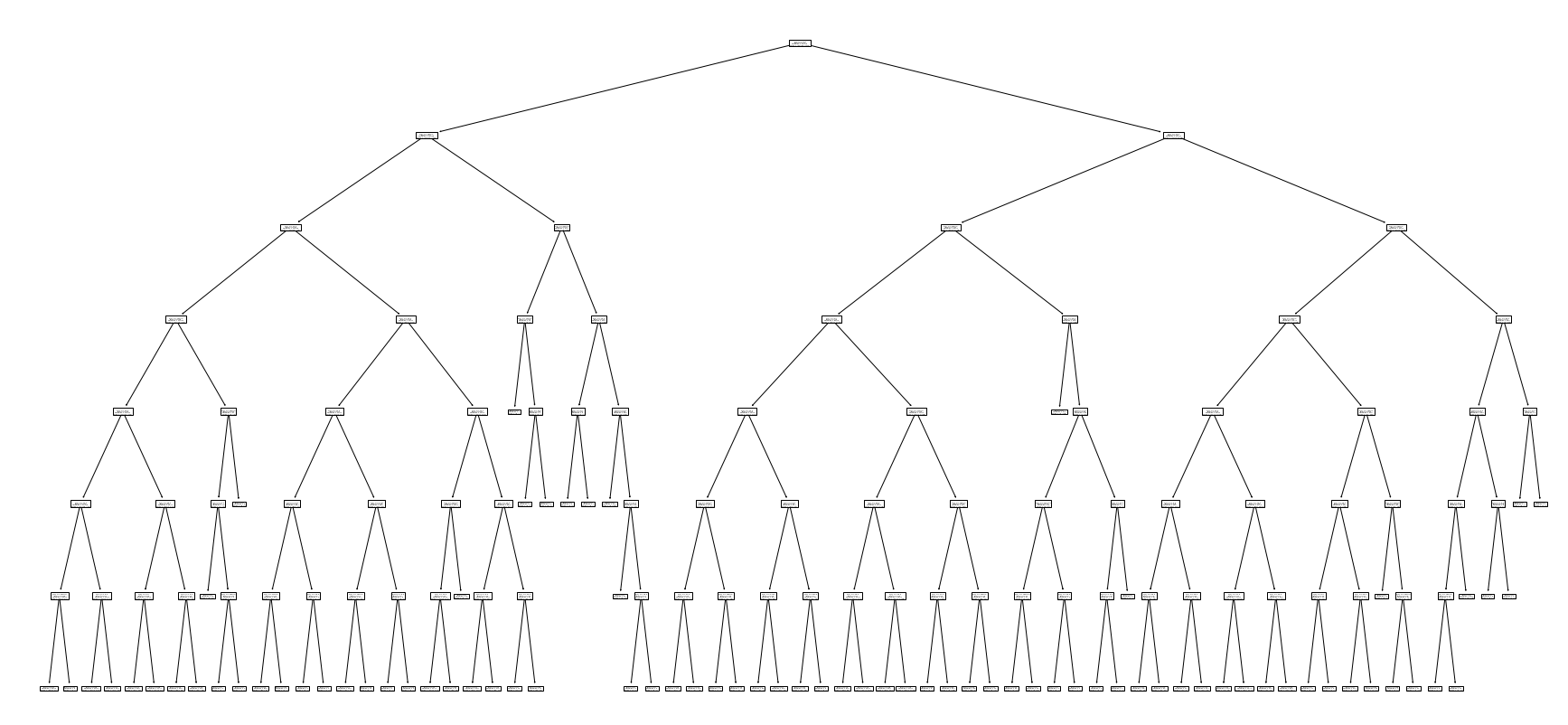
위의 결과와 거의 비슷하지만 전반적으로 조금 더 정확해졌습니다. 이상적인 max\_depth의 값은 주어진 데이터에 따라서 다르기 때문에 직접 다양한 시도를 하면서 오버피팅을 낮추는 노력을 해야 합니다.

## 8.9 트리 그래프

8.6절 ‘이해하기 : 결정 트리’에서 다룬 예시를 그래프로 그려봅시다. 이 그래프는 다른 그래프들과 다르게 맷플롯립matplotlib이나 시본seaborn이 아닌, 사이킷런의 tree 안에서 불러옵니다.

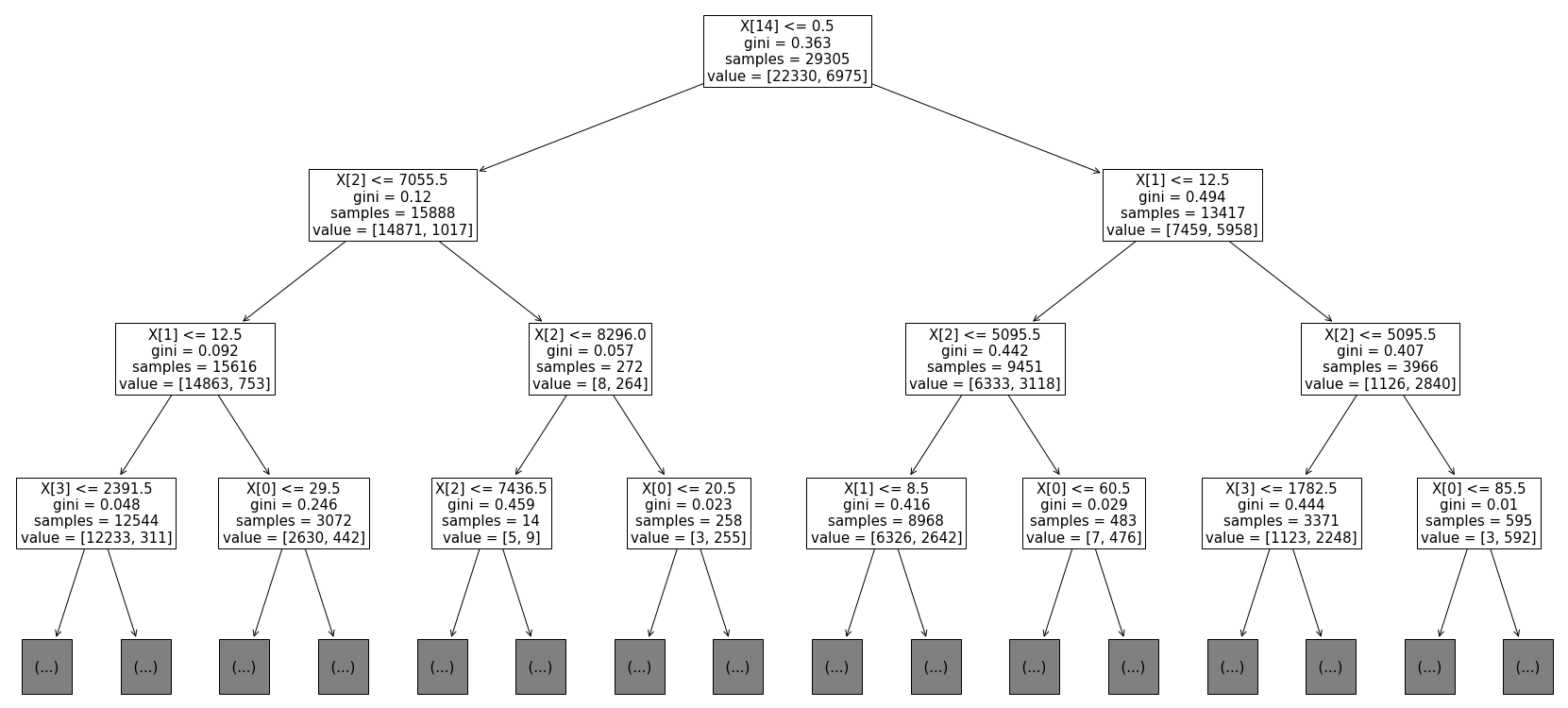
| from sklearn.tree import plot\_tree # 임포트  plt.figure(figsize=(30,15)) # ❶ 그래프 크기 설정  plot\_tree(model) # ❷ 트리 그래프 출력  plt.show() # ❸ 불필요한 문자가 나오지 않게 출력 |
| --- |

❶ 그래프 크기를 조정합니다. figsize에 입력되는 두 값은 각각 가로, 세로 길이입니다. 단위는 인치입니다. ❷ plot\_tree 안에 위에서 훈련시킨 모델의 이름을 넣어주면 그래프를 그립니다. ❸ 안써도 상관없으나, 불필요한 텍스트들까지 아웃풋으로 출력되는 걸 원하지 않는다면 가급적 사용하기 바랍니다.



이 모델은 우리가 마지막에 훈련시킨 max\_depth 7의 결과물입니다. 7으로 제한했음에도 너무 거대한 트리라서 내용을 확인할 수 없을 정도로 작게 표현되었습니다. 따라서 이 트리의 전체를 보기보다는 중요하다고 여겨지는 윗 부분(3단계까지)만을 확대해서 보겠습니다. 그 밖에 폰트 크기를 조정하는 매개변수도 추가하겠습니다.

| plt.figure(figsize=(30,15)) # 그래프 크기 지정 plot\_tree(model, max\_depth=3, fontsize=15) # 깊이와 폰트 크기 조정하여 트리 그래프 출력 plt.show() # 불필요한 문자가 나오지 않게 출력 |
| --- |



깊이를 낮추고 폰트를 키워서 이전보다 보기 수월해졌습니다만, 노드 안의 분류 기준이 X[1], X[2]와 같이 되어 있습니다. 이 숫자는 독립변수를 의미하는데, 훈련셋의 변수 순서와 같습니다. 이 부분도 매개변수를 통해서 원래 변수 이름으로 바꿔보겠습니다.

| plt.figure(figsize=(30,15)) # 그래프 크기 지정  plot\_tree(model, max\_depth=3, fontsize=15, feature\_names=X\_train.columns) # ❶ 변수 이름을 추가하여 그래프 출력 plt.show() # 불필요한 문자가 나오지 않게 출력 |
| --- |



❶ feature\_names 매개변수에 변수 이름이 담긴 리스트를 입력합니다. X\_train.columns를 활용하면 간단하게 변수 이름을 불러 올 수 있습니다.

❶ 첫 번째 노드에는 분류 기준, 지니 인덱스(gini), 총 데이터 수(samples)가 표시됩니다. value는 목푯값 0과 1이 각각 몇 개씩인지를 보여줍니다. 분류 기준인 첫 줄에서는 marital-status\_Married-civ-spouse ≦ 0.5를 기준으로 나누었습니다. 해당 변수는 기존 marital-status에서 더미 변수로 파생된 것으로, 결혼상태가 Married-civ-spouse에 해당하는지 아닌지가 가장 중요한 변수로 꼽혔습니다. 노드의 분류는 조건에 맞으면 왼쪽, 아니면 오른쪽으로 분류되는데, 여기에서는 왼쪽으로 뻗은 노드가 Married-civ-spouse에 해당하지 않는 경우가 되겠습니다.

❷ 두 번째 노드의 지니 인덱스는 0.12로 꽤 많이 낮아졌고, value를 보면 0인 경우가 훨씬 많도록 분류되었습니다. 이와 같이 트리 그래프를 해석할 수 있으며, 이는 비전문이에게도 설명이 용이하기 때문에 유용하게 사용됩니다.

## 학습 마무리

#### 되짚어보기

8.1 나이, 교육연수, 가족관계 등의 다양한 변수를 사용하여 연봉을 예측하는 모델을 만들어봅니다.

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

8.3 더미 변수로 추가하고, 카테고리별 종속변수의 평균값을 치환해 범주형 데이터를 전처리했습니다.

8.4 결측치를 특정값으로 치환하는 방법으로 처리했습니다. 아직까지 남아있는 범주형 데이터를 더미 변수로 변환해주었습니다.

8.5 결정 트리를 사용해 예측 모델을 만든 결과, 약 81%의 정확도를 얻었습니다.



#### 과제

‘native-country’를 다양한 방법으로 처리하여 결과를 비교해봅시다. 8.3절에 소개된 방법을 이용해도 좋고, 더미 변수를 사용하여 수많은 컬럼으로 만들어도 좋습니다. 4장에서 사용한 타이타닉 데이터를 결정 트리로 모델링해봅시다.

#### 핵심 용어 정리

1. **결정 트리** : 트리 모델의 가장 기본 형태입니다. 데이터의 특성을 고려하여 데이터를 분류해나가는 방식입니다.
2. **지니 인덱스와 교체 엔트로피** : 노드의 순도를 평가하는 방법. 노드의 순도가 높을수록 지니 및 엔트로피 값은 낮아집니다.
3. **오버피팅** : 모델이 학습셋에 지나치게 잘 맞도록 학습되어서 새로운 데이터에 대한 예측력이 떨어지는 현상을 의미합니다. 8장에서 더 자세하게 다룰 예정입니다.
4. **언더피팅** : 과소적합이라고도 하며, 모델이 충분히 학습되지 않아 훈련셋에 대해서도 좋은 예측력을 내지 못하는 상황입니다.

#### 새로운 함수와 라이브러리

* **pandas.DataFrame.merge()** : 두개의 데이터프레임을 합칩니다.
* **pandas.DataFrame.reset\_index()** : 판다스 데이터 프레임과 시리즈에서 인덱스에 들어있는 정보를 별도의 컬럼으로 빼옵니다.
* **pandas.DataFrame.rename()** : 데이터프레임의 컬럼이나 인덱스 이름을 변경할 수 있습니다.
* **pandas.DataFrame.sort\_values()** : 데이터를 특정 컬럼 기준 오름차순 혹은 내림차순으로 정렬합니다.

## 연습문제

1. 다음 중 범주형 데이터의 결측치를 처리하는 방법으로 옳지 않은 것은?

① 특정 값이 압도적으로 많이 등장할 경우, 해당값으로 결측치를 대체할 수 있다.

② 경우에 따라 임의의 텍스트(unknown) 등으로 대체할 수 있다.

③ 평균값으로 결측치를 대체하는 것이 가장 무난하다.

④ 결측치가 많지 않으면 결측치 행을 제거할 수 있다.

2. 다음 중 결정트리가 노드를 분류할 때 사용하는 기준은?

① 지니 인덱스

② RMSE

③ R2

④ 사후 확률

3. 다음 중 오버피팅에 대한 설명으로 옳지 않은 것은?

① 모델이 학습 데이터에 지나치게 편향적으로 예측할 경우 발생한다.

② 반대 개념으로 언더피팅이 있다.

③ 오버피팅보다는 언더피팅이 발생하는 것이 더 나은 모델이다.

④ 모델의 학습을 의도적으로 제한하여 오버피팅을 줄일 수 있다.

#### 정답 및 해설

## 

1. 3

③ 평균값으로 결측치를 대체하는 것이 가장 무난하다. ← 범주형 데이터에서는 평균값을 구할 수가 없습니다.

2. 1

① 지니 인덱스

② RMSE ← 연속형 변수 예측의 평가지표입니다.

③ R2 ← 선형 회귀 분석의 평가지표입니다.

④ 사후 확률 ← 사건 A와 B가 있을 때, 사건 A가 발생한 상황에서 사건 B가 발생할 확률로, 나이브베이즈 알고리즘에서 사용됩니다.

3. 3

③ 오버피팅보다는 언더피팅이 발생하는 것이 더 나은 모델이다. ← 오버피팅은 학습이 지나치게 잘되어서 문제인 반면, 언더피팅은 학습이 애초에 잘 되지 않은 것이므로 언더피팅이 더 안 좋은 상황입니다.