데이터 전처리

구글 코랩에서 실행하기

넘파이로 데이터 준비하기

```
#세련된 방법으로 생선 데이터를 준비

fish_length = [25.4, 26.3, 26.5, 29.0, 29.0, 29.7, 29.7, 30.0, 30.0, 30.7, 31.0, 31.0,  31.5, 32.0, 32.0, 32.0, 33.0, 33.0, 33.5, 33.5, 34.0, 34.0, 34.5, 35.0, 35.0, 35.0, 35.0,
fish_weight = [242.0, 290.0, 340.0, 363.0, 430.0, 450.0, 500.0, 390.0, 450.0, 500.0, 475.0, 500.0,  500.0, 340.0, 600.0, 600.0, 700.0, 700.0, 610.0, 650.0, 575.0, 685.0, 620.0,

import numpy as np

#column_stack() 함수는 전달받은 리스트를 일렬로 세운 다음 차례대로 나란히 연결함
#예를 들면 다음과 같은 간단한 2개의 리스트를 연결할 리스트는 파이썬 튜플 tuple로 전달
#세로(열)로 쌓아주는 column_stack() 대신 row_stack()은 가로(행)로 쌓아줌
np.column_stack(([1,2,3], [4,5,6]))
```

```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

---

fish_length와 fish_weight를 합치기고 확인하기

```
fish_data = np.column_stack((fish_length, fish_weight))
print(fish_data[:5])
#넘파이 배열을 출력하면 리스트처럼 한 줄로 길게 출력되지 않고 행과 열(5행*2열)을
#맞추어 가지런히 정리된 모습으로 보여 줌
```

```
[[ 25.4 242. ]
 [ 26.3 290. ]
 [ 26.5 340. ]
 [ 29.  363. ]
```

```
       [ 29.  430. ]]
```

```
#넘파이의 np.ones()와 np.zeros() 함수는 각각 원하는 개수의 1과 0을 채운 배열을 만들어 줌
print(np.ones(5))
```

```
[1. 1. 1. 1. 1.]
```

```
#1이 35개인 배열과 0이 14개인 배열을 간단히 만들 수 있음
#np.concatenate()를 이용하여 두 배열을 그대로 연결
#np.full((2, 3), 9)는 2행 3열로 9가 채워짐
fish_target = np.concatenate((np.ones(35), np.zeros(14)))
print(fish_target)
```

```
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0.]
```

사이킷런으로 훈련 세트와 테스트 세트 나누기

```
#넘파이 배열의 인덱스를 직접 섞어서 훈련 세트와 테스트 세트로 나누는 대신
#사이킷런은 유틸리티 도구 train_test_split() 함수를 이용하여 전달되는 리스트나 배열을
#비율에 맞게 훈련 세트와 테스트 세트로 나누어 줌

from sklearn.model_selection import train_test_split

#fish_data와 fish_target 2개의 배열을 전달했으므로 2개씩 나뉘어 총 4개의 배열이 반환됨
#fish_data → train_input, test_input / fish_target → train_target, test_target
#np.random.seed() 함수를 사용해 출력 결과가 같아지도록 함
train_input, test_input, train_target, test_target = train_test_split(fish_data, fish_target, random_state=42)
print(train_input.shape, test_input.shape)
```

```
(36, 2) (13, 2)
```

```
#train_test_split는 기본적으로 25%를 테스트 세트로 떼어 냄
print(train_target.shape, test_target.shape) #13/49 = 0.265
```

```
(36,) (13,)
```

```
#도미와 빙어가 잘 섞였는지 테스트 데이터를 출력
#테스트 세트 13개중 도미가 10개, 빙어가 3개로 빙어의 비율이 적음
#원래 도미와 빙어의 개수가 35개와 14개이므로 두 생선의 비율은 2.5:1임
#하지만 이 테스트 세트의 도미와 빙어의 비율은 3.3:1이기 때문에 샘플링 편향이 조금 나타남

print(test_target)
```

```
[0. 0. 1. 0. 1. 0. 1. 1. 1. 1. 1. 1. 1.]
```

```
#또한 무작위로 데이터를 나누었을 때 샘플이 골고루 섞이지 않을 수 있으며 특히 일부 클래스의 개수가 적으면 이런 일이 발생
#그러므로 stratify 매개 변수에(☞)타깃 데이터를 전달하면 타깃 비율을 보고 클래스 비율에 맞게 골고루 섞이도록 데이터를 나눠줌
#이는 훈련 데이터가 작거나 특정 클래스의 샘플 개수가 적을 때 특히 유용

train_input, test_input, train_target, test_target = train_test_split(fish_data, fish_target, stratify=fish_target, random_state=42)
print(test_target)
```

```
[0. 0. 1. 0. 1. 0. 1. 1. 1. 1. 1. 1. 1.]
```

수상한 도미 한마리 - 준비한 데이터로 k-최근접 이웃 모델을 훈련하고 평가하기

```
from sklearn.neighbors import KNeighborsClassifier

kn = KNeighborsClassifier()
```

```
kn.fit(train_input, train_target)
kn.score(test_input, test_target)  #테스트 세트의 도미와 빙어를 모두 올바르게 분류함
```

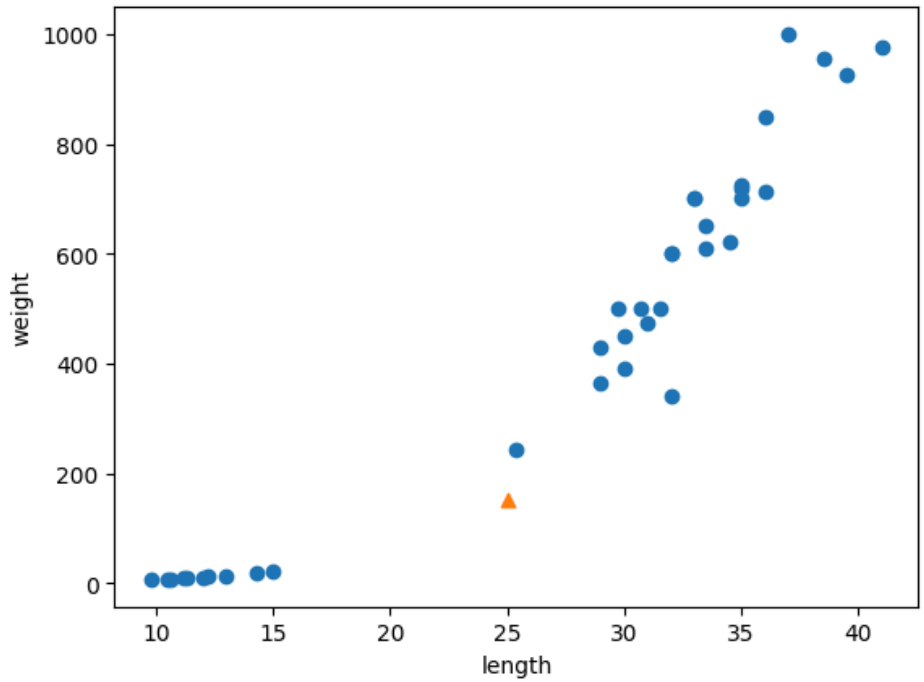
```
1.0
```

```
#주어진 도미 데이터(25, 150)를 넣고 결과를 확인
print(kn.predict([[25, 150]])) #잘못된 예측 발생(빙어로 예측) - 정말 이렇게 큰 빙어가 있는 걸까요?
```

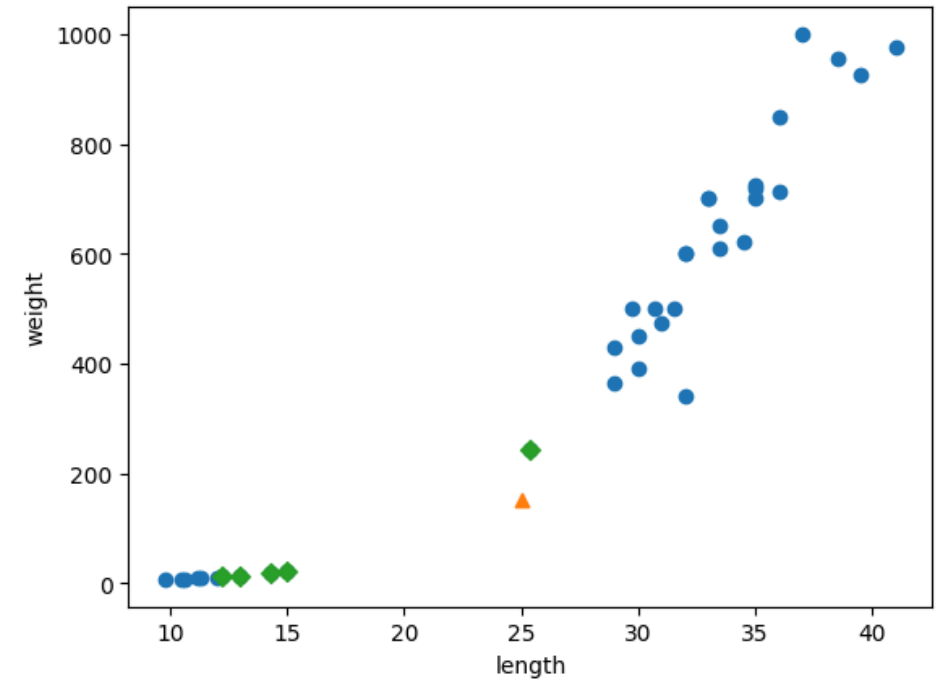
```
[0.]
```

산점도 그리기

```
import matplotlib.pyplot as plt
plt.scatter(train_input[:,0], train_input[:,1])
# marker 매개변수는 모양을 지정
plt.scatter(25, 150, marker='^')
plt.xlabel('length')
plt.ylabel('weight')
plt.show()
```




```
#이유 - k-최근접 이웃은 주변의 샘플 중에서 다수인 클래스를 예측으로 사용

#KNeighborsClassifier 클래스는 주어진 샘플에서 가장 가까운 이웃을 찾아 주는 kneighbors() 메서드를 제공
#이 메서드는 이웃까지의 거리와 이웃 샘플의 인덱스를 반환
#KNeighborsClassifier 클래스의 이웃 개수인 n_neighbors의 기본값은 5이므로 5개의 이웃이 반환
distances, indexes = kn.kneighbors([[25, 150]])

plt.scatter(train_input[:,0], train_input[:,1])
plt.scatter(25, 150, marker='^')

#marker='D'로 지정하면 산점도를 마름모로 그림
plt.scatter(train_input[indexes,0], train_input[indexes,1], marker='D')
plt.xlabel('length')
plt.ylabel('weight')
plt.show()
```




```
print(train_input[indexes])
```

```
[[[ 25.4 242. ]
  [ 15.   19.9]
  [ 14.3  19.7]
  [ 13.   12.2]
  [ 12.2  12.2]]]
```

```
print(train_target[indexes])
```

```
[[1. 0. 0. 0. 0.]]
```

```
print(distances)
```

```
[[ 92.00086956 130.48375378 130.73859415 138.32150953 138.39320793]]
```

기준을 맞춰라

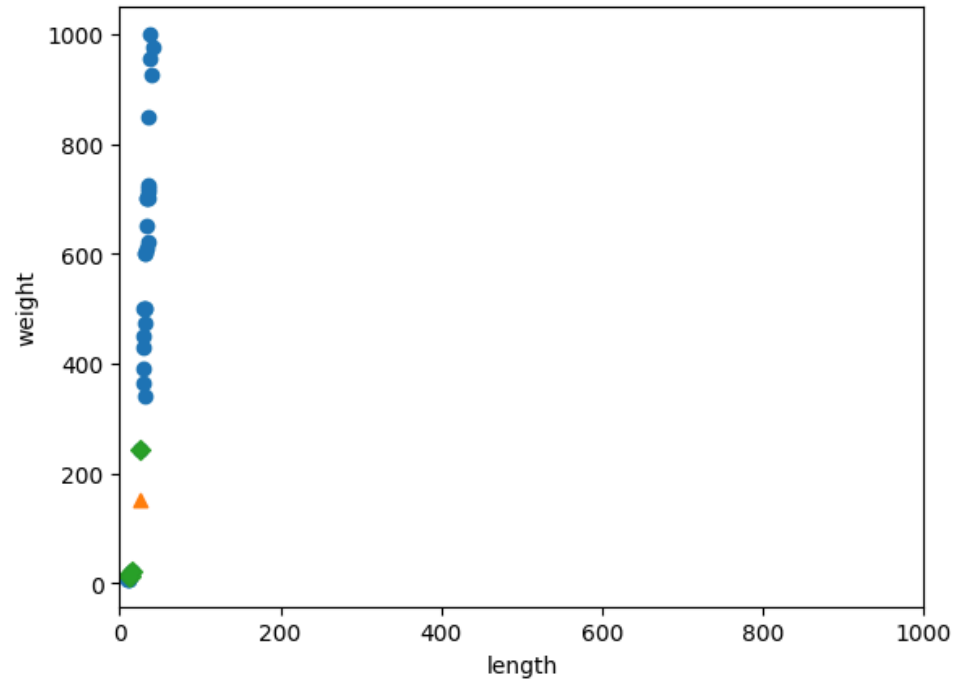
- 산점도 살펴보기

> - 가장 가까운 첫 번째 샘플까지의 거리는 92이고, 그외 가장 가까운 샘플들은 모두 130, 138임
> - 거리가 92와 130이라고 했을 때 그래프에 나타난 거리 비율이 이상한 것을 알 수 있음
> - x축(10-40)은 범위가 좁고 y축(0-1000)은 범위가 넓기 때문에 스케일이 맞지 않아 y축에서 조금만 멀어져도 거리가 아주 큰 값으로 계산되게 되며 이 때문에 오른쪽 위의 도미 샘플이 이웃으로 선택되지 못했던 것임

스케일 조정하기

- Matplotlib에서 x축 범위를 지정하려면 xlim() 함수를 사용
- y축 범위를 지정하려면 ylim() 함수를 사용

```
#x축의 범위를 동일하게 0~1000으로 맞추기 위해 xlim() 함수를 이용
#x축과 y축의 범위를 동일하게 맞추었더니 모든 데이터가 수직으로 늘어선 형태가 되었음
#이런 데이터라면 생선의 길이(x축)는 가장 가까운 이웃을 찾는데 크게 영향을 끼치지 않고 오로지 생선의 무게(y축)만 고려 대상
#무게가 생선을 구분하는데 큰 영향력이 있고, 길이는 거의 미미하므로
#두 특성의 값이 놓인 범위가 매우 달라 두 특성의 스케일링이 달라 발생하는 문제라고 할 수 있음
#스케일이 다른 일은 매우 흔한데 예를들어 어떤 사람이 방의 넓이를 재는데 세로는 cm로, 가로는 inch로 쟀다면 정사각형인 방도 직사각형처럼 보일 것임

plt.scatter(train_input[:,0], train_input[:,1])
plt.scatter(25, 150, marker='^')
plt.scatter(train_input[indexes,0], train_input[indexes,1], marker='D')

plt.xlim((0, 1000))

plt.xlabel('length')
plt.ylabel('weight')
plt.show()
```



데이터 전처리하기

- 샘플 간의 거리에 영향을 많이 받으므로 제대로 사용하려면 특성값을 일정한 기준으로 맞춰주는 데이터 전처리 작업이 필요
- 데이터를 표현하는 기준이 다르면 스케일이 큰 특성에 절대적으로 영향을 받으므로 올바르게 예측할 수 없음(거리 기반인 k-최근접 이웃도 포함됨)

표준점수(standard score)

- 가장 널리 사용하는 전처리 방법 중 하나
- z 점수라고도 부름
- 각 특성값이 평균에서 표준편차의 몇 배만큼 떨어져 있는지를 나타냄

- 이를 통해 실제 특성값의 크기와 상관없이 동일한 조건으로 비교할 수 있음

```
#평균과 편차 구하기

#np.mean() 함수는 평균을 계산
mean = np.mean(train_input, axis=0)

#np.std() 함수는 표준편차를 계산
#특성마다 값의 스케일이 다르므로 평균과 표준편차는 각 특성별로 계산해야 함
#이를 위해 axis=0으로 지정 - 행을 따라 각 열의 통계 값을 계산
std = np.std(train_input, axis=0)

#train_input은 (36, 2) 크기의 배열
print(mean, std)
```

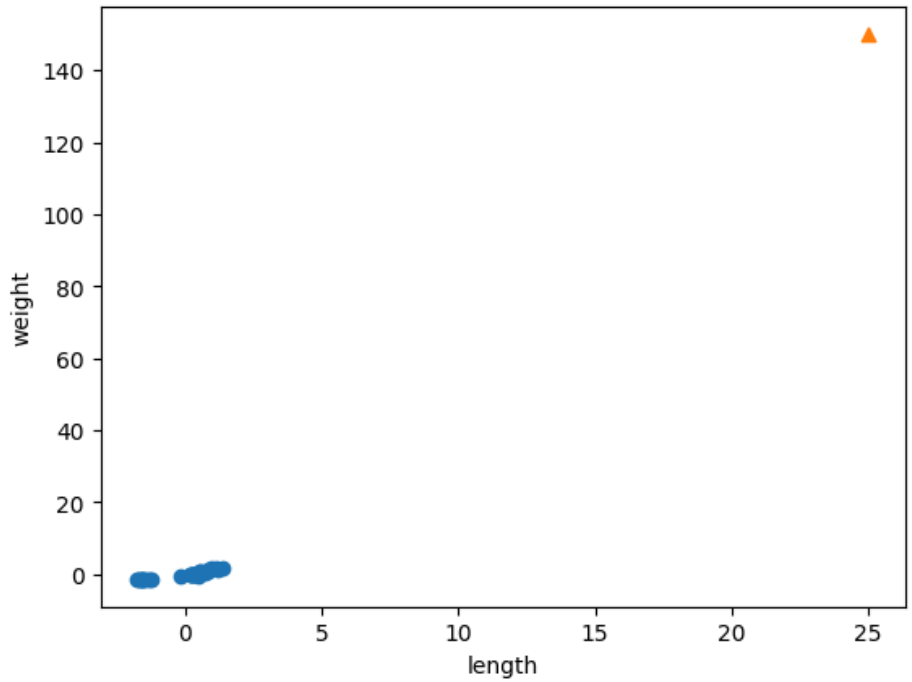
```
[ 27.29722222 454.09722222] [  9.98244253 323.29893931]
```

```
#넘파이 기능 브로드캐스팅(broadcasting)을 이용 - 넘파이에서 서로 다른 모양(shape)의 배열도 일정 조건을 만족하면 연산할 수 있음

#(3, 3)배열과 (1, 3) 배열은 서로 짝이 맞고, 하나의 배열이
#1차원 배열이므로 브로드캐스팅이 가능
arr1 = np.array([[0, 0, 0], [1, 1, 1], [2, 2, 2]])
arr2 = np.array([5, 6, 7])
print(arr1 + arr2)


#a(1, 3)배열과 (3, 1)배열은 서로 짝이 맞고, 하나의 배열이 1차원 배열이므로 #브로드캐스팅이 가능
arr3 = np.array([1, 1, 1])
arr4 = np.array([[0], [1], [2]])
print(np.add(arr3, arr4))

#표준 점수로 변환
train_scaled = (train_input - mean) / std
```

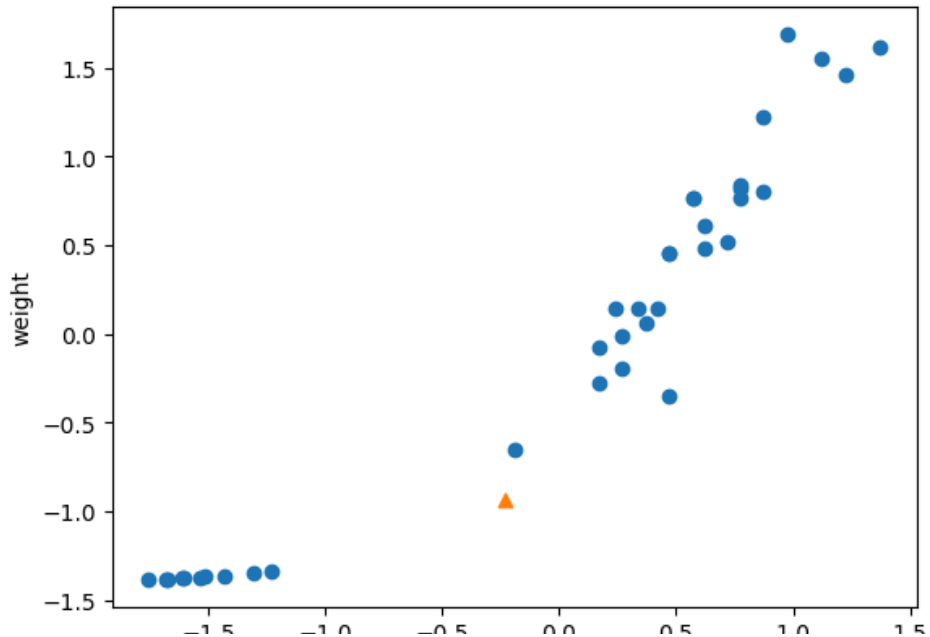
```
[[5 6 7]
 [6 7 8]
 [7 8 9]]
[[1 1 1]
 [2 2 2]
 [3 3 3]]
```

전처리 데이터로 모델 훈련하기

```
#표준점수로 변환한 train_scaled를 활용하여 산점도 그리기

plt.scatter(train_scaled[:,0], train_scaled[:,1])
plt.scatter(25, 150, marker='^')
plt.xlabel('length')
plt.ylabel('weight')
plt.show()
```




```
#표준점수로 변환한 train_scaled를 활용하여 산점도 그리기
new = ([25, 150] - mean) / std
plt.scatter(train_scaled[:,0], train_scaled[:,1])
plt.scatter(new[0], new[1], marker='^')
plt.xlabel('length')
plt.ylabel('weight')
plt.show()
```




```
kn.fit(train_scaled, train_target)
test_scaled = (test_input - mean) / std
kn.score(test_scaled, test_target) #모든 테스트 세트의 샘플을 완벽하게 분류
```

```
1.0
```

```
print(kn.predict([new]))  #도미 new = ([25, 150]를 1로 예측
```

```
[1.]
```

```
distances, indexes = kn.kneighbors([new])
plt.scatter(train_scaled[:,0], train_scaled[:,1])
plt.scatter(new[0], new[1], marker='^')
plt.scatter(train_scaled[indexes,0], train_scaled[indexes,1], marker='D')
plt.xlabel('length')
plt.ylabel('weight')
plt.show()
```