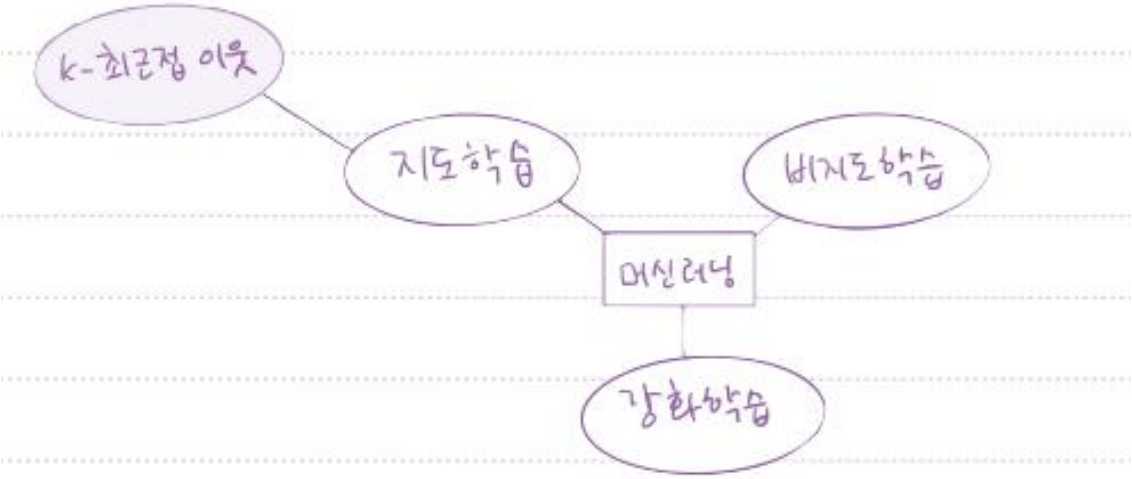
ch2 02-1 훈련 세트와 테스트 세트

- 지도 학습과 비지도 학습



지도학습
입력(데이터)과 **타깃**(정답)으로 이뤄진 훈련 데이터가 필요하며 새로운 데이터를 예측하는 데 활용함. (ch1에서 사용한 k-최근접 이웃이 지도 학습 알고리즘)

비지도학습
타깃 데이터 없이 입력 데이터만 있을 때 사용. 이런 종류의 알고리즘은 정답을 사용하지 않으므로 무언가를 맞힐 수가 없는 대신 데이터를 잘 파악하거나 변형하는데 도움을 줌

- 훈련 세트와 테스트 세트

훈련 세트 : 모델을 훈련할 때 사용되는 데이터
훈련 데이터 : 지도학습의 경우 필요한 입력(데이터)와 타깃(정답)을 합쳐 놓은 것

태스트 세트 : 모델을 평가할 때 사용하는 데이터. 테스트 세트는 전체 데이터에서 20~30%

```
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, 36.0, 36.0, 37.0, 38.5, 38.5, 39.5, 41.0, 41.0, 9.8,
               10.5, 10.6, 11.0, 11.2, 11.3, 11.8, 11.8, 12.0, 12.2, 12.4, 13.0, 14.3, 15.0]
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, 680.0,
```

```
                700.0,  725.0,  720.0,  714.0,  850.0,  1000.0,  920.0,  955.0,  925.0,
975.0,  950.0,  6.7,
                7.5,  7.0,  9.7,  9.8,  8.7,  10.0,  9.9,  9.8,  12.2,  13.4,  12.2,  19.7,  19.9]
```

```
# 물고기 데이터는 총 49개의 샘플로 구성되어 있으며,
# 각 샘플은 '길이'와 '무게' 두 가지 특성을 가진다.
# fish_length와 fish_weight를 같은 인덱스끼리 묶어
# [길이, 무게] 형태의 입력 데이터(X)를 생성한다.
fish_data = [[l, w] for l, w in zip(fish_length, fish_weight)]

# 전체 49개 샘플 중
# 처음 35개는 훈련 세트(training set),
# 나머지 14개는 테스트 세트(test set)로 사용한다.
# 훈련 세트의 타깃 값은 1(예: 도미)이고,
# 테스트 세트의 타깃 값은 0(예: 빙어)이다.
fish_target = [1]*35 + [0]*14
```

```
# 사이킷런(sklearn)의 이웃 기반 분류 알고리즘인
# KNeighborsClassifier를 불러온다
from sklearn.neighbors import KNeighborsClassifier

# K-최근접 이웃 분류 모델 객체를 생성
kn = KNeighborsClassifier()
```

```
# 전체 입력 데이터(fish_data) 중
# 처음 35개 샘플을 훈련 입력 데이터로 사용한다
train_input = fish_data[:35]

# 전체 타깃 데이터(fish_target) 중
# 처음 35개 샘플을 훈련 타깃 데이터로 사용한다
train_target = fish_target[:35]

# 전체 입력 데이터(fish_data) 중
# 35번째 인덱스부터 끝까지를 테스트 입력 데이터로 사용한다
test_input = fish_data[35:]

# 전체 타깃 데이터(fish_target) 중
# 35번째 인덱스부터 끝까지를 테스트 타깃 데이터로 사용한다
test_target = fish_target[35:]
#테스트 세트로 평가하기
from sklearn.neighbors import KNeighborsClassifier
```

```
kn = KNeighborsClassifier()
kn = kn.fit(train_input, train_target)

kn.score(test_input, test_target)
#-> 0.0, test sample에 있는 14개의 데이터를 다 맞추지 못함.
#[:35]로 나누었는데, 훈련 데이터는 도미 35마리, 테스트 데이터는 빙어 14마리의 정보가 입력되었기 때문!!!
```

슬라이싱 : 리스트(또는 문자열, 배열)의 일부분을 잘라서 가져오는 방법

기본 형태
리스트 [시작:끝] //시작 : 인덱스를 포함, 끝 : 인덱스 미포함

ex)
a = [0, 1, 2, 3, 4]

코드	결과
a[:3]	[0, 1, 2]
a[2:]	[2, 3, 4]
a[1:4]	[1, 2, 3]

```
# 훈련 입력 데이터(train_input)와
# 훈련 타깃 데이터(train_target)를 사용해
# K-최근접 이웃 분류 모델을 학습시킨다
kn.fit(train_input, train_target)

# 테스트 입력 데이터(test_input)와
# 테스트 타깃 데이터(test_target)를 사용해
# 학습된 모델의 정확도를 평가한다
# 반환값은 0~1 사이의 정확도(accuracy)이다
kn.score(test_input, test_target)
```

.fit : 모델 학습(훈련 데이터 저장)
.score : 테스트 데이터로 성능 평가(정확도 반환)
정확도 = (맞힌 개수) / (전체 테스트 샘플 수)
ex) 테스트 샘플 14개 중 14개 맞춤 → 1.0
10개 맞춤 → 10 / 14 ≈ 0.71

- 샘플링 평향

```
kn.score(test_input, test_target)
```

-> 0.0

test sample에 있는 14개의 데이터를 다 맞추지 못함.
[:35]로 나누었는데, 훈련 데이터는 도미 35마리, 테스트 데이터는 빙어 14마리의 정보가 입력되었기 때문

샘플링 편향
훈련 세트와 테스트 세트에 샘플이 고르게 섞여 있지 않을 때 즉 모델이 한쪽 클래스만 학습하게 되는 현상

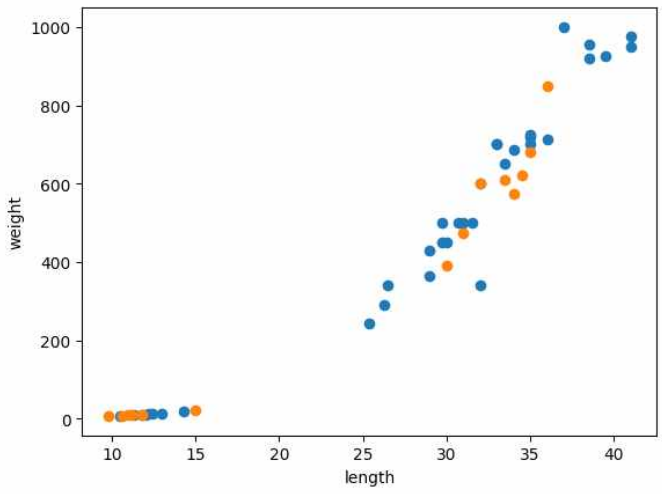
- 넘파이
넘파이 : 파이썬에서 수치 계산과 배열 처리를 빠르고 쉽게 해주는 라이브러리

```
import numpy as np

# 파이썬 리스트로 되어 있는 입력 데이터와 타깃 데이터를
# NumPy 배열로 변환한다
input_arr = np.array(fish_data)
target_arr = np.array(fish_target)

# 입력 데이터 배열 출력 (확인용)
print(input_arr)

# 데이터 섞기 (샘플링 편향 방지)
# 인덱스가 담긴 배열을 넣어서 원소를 선택하는 방식

# 난수 생성을 위한 시드(seed) 설정
# 같은 난수가 나오도록 하여 결과를 재현할 수 있게 한다
np.random.seed(42)

# 0부터 48까지 1씩 증가하는 정수 배열
index = np.arange(49)

# 인덱스 배열을 무작위로 섞는다
np.random.shuffle(index)

# 입력 데이터와 타깃 데이터는 서로 짝을 이루고 있으므로
# 각각을 따로 섞으면 정답이 뒤섞여 지도 학습이 불가능해진다
# 같은 인덱스를 사용해 입력과 타깃을 함께 섞어야 한다

# 섞인 인덱스 중 처음 35개를 사용해 훈련 세트를 만든다
train_input = input_arr[index[:35]]
```

```
train_target = target_arr[index[:35]]

# 나머지 14개를 사용해 테스트 세트를 만든다
test_input = input_arr[index[35:]]
test_target = target_arr[index[35:]]
```

훈련세트와 테스트 세트에 도미와 빙어가 잘 섞여 있는지 산점도로 확인

```
import matplotlib.pyplot as plt

# 훈련 세트의 산점도 그리기
# train_input[:, 0] → 모든 샘플의 첫 번째 특성(길이)
# train_input[:, 1] → 모든 샘플의 두 번째 특성(무게)
plt.scatter(train_input[:, 0], train_input[:, 1])

# 테스트 세트의 산점도 그리기
# 훈련 세트와 구분하기 위해 다른 색으로 표시됨
plt.scatter(test_input[:, 0], test_input[:, 1])

# x축 이름 설정 (물고기 길이)
plt.xlabel('length')

# y축 이름 설정 (물고기 무게)
plt.ylabel('weight')

# 그래프 화면에 출력
plt.show()
```



파랑 : 훈련세트
주황 : 테스트 세트

- 두 번째 머신러닝 프로그램

```
kn = kn.fit(train_input, train_target)
kn.score(test_input, test_target) #-> 1.0
kn.predict(test_input) #-> array([0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0])
```

모델 훈련에 참여하지 않은 샘플로 테스트를 했기 때문에 올바르게 평가

02-2 데이터 전처리
02-1과 다른 방식으로 데이터를 랜덤하게 나눔

넘파이로 데이터 준비하기

```
import numpy as np

# 물고기 길이(length)와 무게(weight) 데이터를
# 열(column) 방향으로 나란히 붙여 2차원 배열을 만든다
# 한 행(row)은 물고기 1마리(샘플)를 의미하고,
# 한 열(column)은 하나의 특성(길이 또는 무게)을 의미한다
# 결과 배열의 형태(shape)는 (49, 2)가 된다
fish_data = np.column_stack((length, weight))



# 물고기 종류를 나타내는 타깃 데이터 생성
# 앞의 35개는 값이 1 → 도미
# 뒤의 14개는 값이 0 → 빙어
# concatenate는 여러 배열을 순서대로 이어 붙이는 함수이다
# 최종적으로 총 49개의 타깃 데이터를 만든다
fish_target = np.concatenate((np.ones(35), np.zeros(14)))



# np.ones((행, 열)) :
# 지정한 크기만큼 모든 값이 1인 배열 생성
# 예) np.ones((2, 3))
# → [[1, 1, 1],
#    [1, 1, 1]]

# np.full((행, 열), 값) :
# 지정한 크기만큼 모든 값이 같은 배열 생성
# 예) np.full((2, 3), 9)
# → [[9, 9, 9],
#    [9, 9, 9]]
```

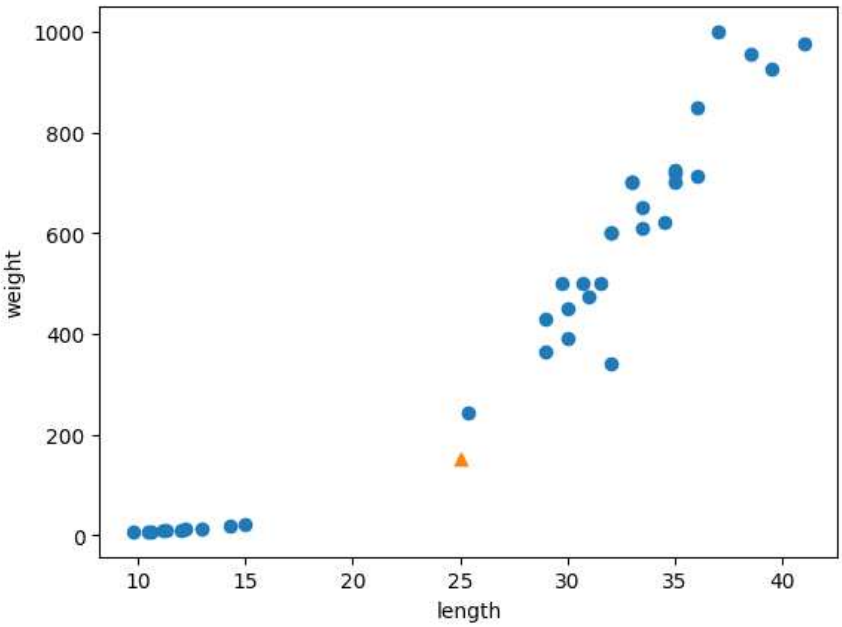
```
from sklearn.model_selection import train_test_split

# 입력 데이터(fish_data)와 타깃 데이터(fish_target)를
# 훈련 세트와 테스트 세트로 무작위 분할한다
#
# stratify=fish_target :
# 타깃 데이터의 클래스 비율(도미 : 빙어)을
# 훈련 세트와 테스트 세트에서 동일하게 유지하도록 한다
# → 샘플링 편향을 방지하기 위해 사용

# random_state=42 :
# 난수 생성 기준값을 고정하여
# 실행할 때마다 같은 결과가 나오도록 한다 (재현성)
train_input, test_input, train_target, test_target = train_test_split(
    fish_data, fish_target, stratify=fish_target, random_state=42
)



# 테스트 세트에 들어 있는 타깃 데이터 출력
# 0(빙어)와 1(도미)이 섞여 있는지 확인하여
# 데이터가 제대로 분할되었는지 점검한다
print(test_target)

# 출력 예:
# [0. 0. 1. 0. 1. 0. 1. 1. 1. 1. 1. 1. 1.]
```

수상한 도미 한 마리

```
from sklearn.neighbors import KNeighborsClassifier

# K-최근접 이웃(KNN) 분류 알고리즘을 사용하기 위해
# KNeighborsClassifier 클래스를 불러온다
kn = KNeighborsClassifier()

# 훈련 입력 데이터(train_input)와
# 훈련 타깃 데이터(train_target)를 사용해
# KNN 모델을 학습시킨다
# KNN은 훈련 데이터를 그대로 저장해 두었다가
```

```
# 새로운 데이터가 들어오면 주변 이웃과 비교하여 분류한다
kn.fit(train_input, train_target)

# 테스트 입력 데이터(test_input)와
# 테스트 타깃 데이터(test_target)를 사용해
# 모델의 성능을 평가
kn.score(test_input, test_target)    # → 1.0

# 새로운 물고기 데이터에 대해 예측을 수행
# [25, 150]은 길이가 25, 무게가 150인 물고기를 의미
# 결과가 [0.]이면 빙어(0)로 예측했다는 뜻
print(kn.predict([[25, 150]]))        # → [0.]
```

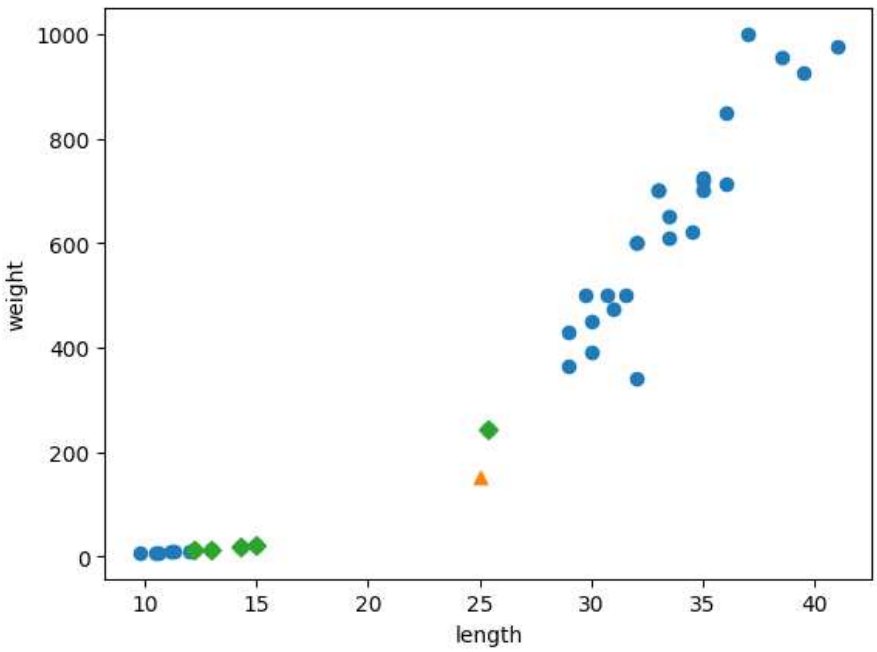
```
import matplotlib.pyplot as plt

# 훈련 데이터의 산점도를 그린다
# train_input[:, 0] → 모든 훈련 샘플의 첫 번째 특성(길이)
# train_input[:, 1] → 모든 훈련 샘플의 두 번째 특성(무게)
# 점 하나는 물고기 한 마리(샘플)를 의미
plt.scatter(train_input[:, 0], train_input[:, 1])



# 새로 예측하려는 물고기 데이터 표시
# x축 값 25 → 길이
# y축 값 150 → 무게
# marker='^'는 삼각형 모양으로 표시하겠다는 의미
# 기존 훈련 데이터와 구분하기 위해 다른 모양을 사용
plt.scatter(25, 150, marker='^')

# x축 이름 설정 (물고기의 길이)
plt.xlabel('length')

# y축 이름 설정 (물고기의 무게)
plt.ylabel('weight')

# 그래프를 화면에 출력
plt.show()
```



왜 이 모델은 빙어 데이터에 가깝다고 판단한 걸까?
[[25, 150]] 샘플에 가까운 다섯 개의 데이터를 추출

```
# 새로운 물고기 데이터 [25, 150]에 대해
# KNN 모델이 참고한 이웃들의 거리와 인덱스를 구한다
# 기본값으로 가장 가까운 5개의 이웃을 반환
distances, indexes = kn.kneighbors([[25, 150]])

# 훈련 데이터 전체를 산점도로 표시
# 점 하나는 물고기 한 마리(샘플)를 의미
plt.scatter(train_input[:, 0], train_input[:, 1])

# 예측하려는 새로운 물고기 데이터 표시
# 삼각형(^) 모양으로 구분
plt.scatter(25, 150, marker='^')

# KNN이 실제로 참고한 가장 가까운 이웃들 표시
# indexes에 들어 있는 인덱스를 사용해
# 훈련 데이터 중에서 이웃 샘플만 선택한다
# marker='D'는 마름모 모양으로 표시
plt.scatter(train_input[indexes, 0],
            train_input[indexes, 1],
            marker='D')

# x축과 y축 이름 설정
plt.xlabel('length')
```

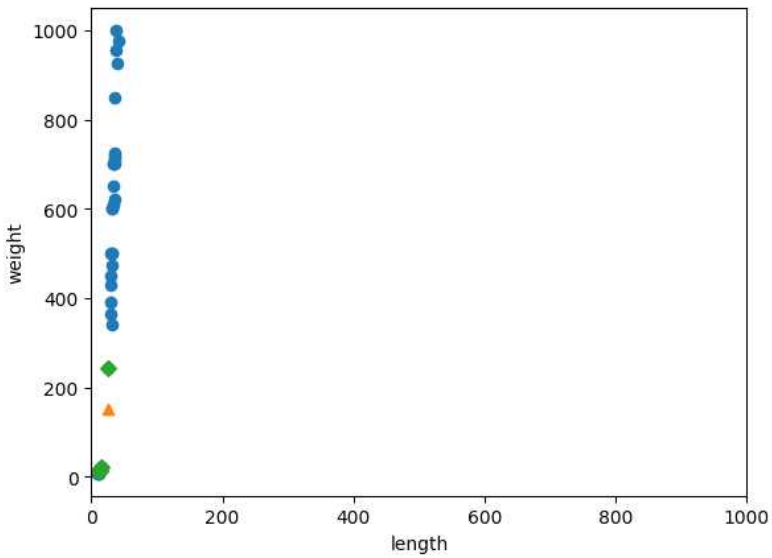
```
plt.ylabel('weight')

# 새로운 물고기와 이웃들 사이의 거리 출력
# 각 값은 유클리디안 거리이며,
# 값이 작을수록 더 가까운 이웃
print(distances)
# 출력 예:
# [[ 92.00086956 130.48375378 130.73859415 138.32150953 138.39320793]]
```



- 기준을 맞춰라

```
# 훈련 데이터 전체를 산점도로 표시
# x축: 물고기 길이, y축: 물고기 무게
plt.scatter(train_input[:, 0], train_input[:, 1])

# 예측하려는 새로운 물고기 데이터 표시
# 길이 25, 무게 150인 물고기를
# 삼각형(^) 모양으로 표시해 훈련 데이터와 구분한다
plt.scatter(25, 150, marker='^')

# KNN이 실제로 참고한 가장 가까운 이웃들 표시
# indexes를 사용해 선택된 이웃 샘플만 표시한다
# 마름모(D) 모양으로 표시한다
```

```
plt.scatter(train_input[indexes, 0],
            train_input[indexes, 1],
            marker='D')

# x축의 범위를 0부터 1000까지로 강제로 설정
# 길이와 무게의 스케일 차이로 인해
# 그래프가 한쪽으로 치우쳐 보이는 문제를 확인하기 위함
plt.xlim((0, 1000))

# x축과 y축 이름 설정
plt.xlabel('length')
plt.ylabel('weight')

# 그래프를 화면에 출력
plt.show()
```



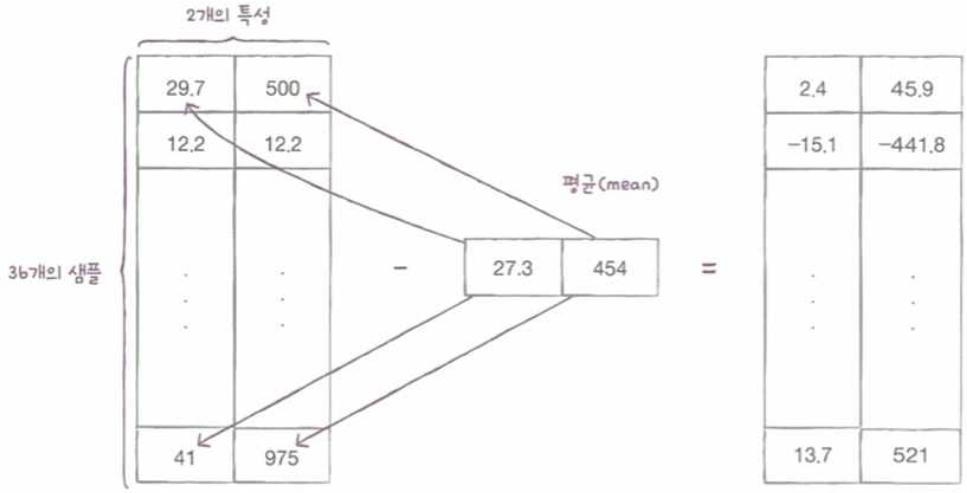

```
# 훈련 데이터의 각 특성(길이, 무게)에 대한 평균을 계산한다
# axis=0 → 행 방향으로 계산하여 열(특성)별 평균을 구함
mean = np.mean(train_input, axis=0)

# 훈련 데이터의 각 특성(길이, 무게)에 대한 표준편차를 계산한다
# axis=0 → 열(특성)별 표준편차 계산
std = np.std(train_input, axis=0)

# 계산된 평균과 표준편차 출력
```

```
# mean  → [길이 평균, 무게 평균]
# std   → [길이 표준편차, 무게 표준편차]
print(mean, std)
# 출력 예:
# [ 27.29722222 454.09722222]  [  9.98244253 323.29893931]

# 데이터 표준화(Standardization)

# 훈련 데이터를 표준화한다
# (각 값 - 평균) / 표준편차
# → 평균 0, 표준편차 1인 데이터로 변환
train_scaled = (train_input - mean) / std

# 테스트 데이터도 훈련 데이터의 평균과 표준편차를 사용해 표준화한다
# 테스트 데이터의 정보가 학습에 새어 들어가는 것을 방지하기 위함
test_scaled = (test_input - mean) / std
```

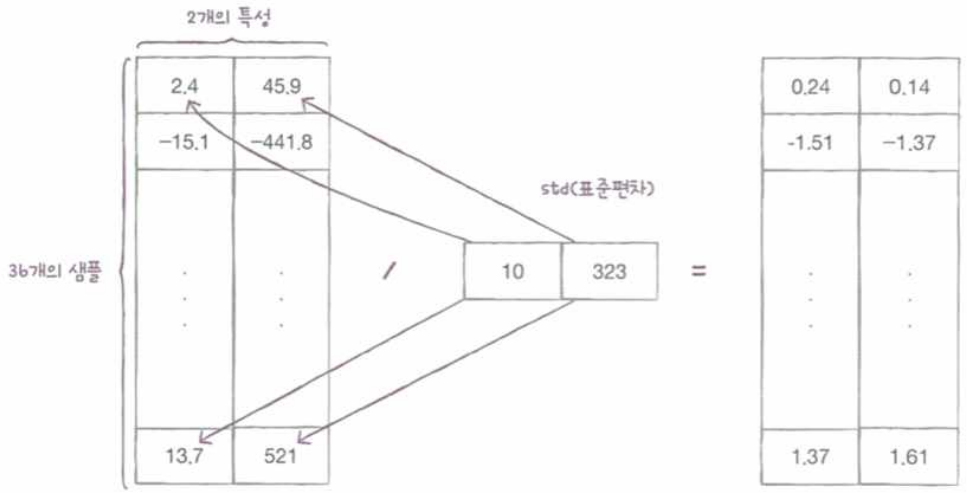
표준 점수를 활용하여 각 특성값이 0에서 표준편차의 몇 배 만큼 떨어져 있는지 나타내기위한 코드

브로드캐스팅(Broadcasting) : 크기가 다른 넘파이 배열에서 자동으로 사칙 연산을 모든 행이나 열로 확장하여 수행하는 기능

mean(평균) 빼기



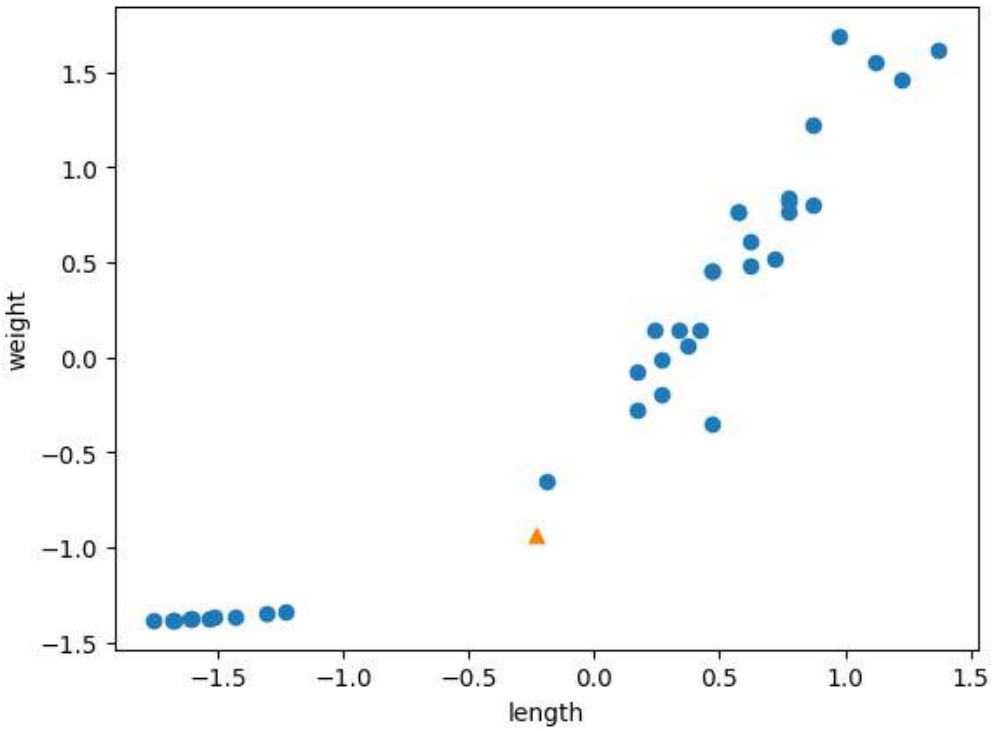
std(표준편차) 나누기



- 전처리 데이터로 모델 훈련하기

새로운 데이터를 훈련 데이터와 같은 기준으로 전처리해서 비교

```
# 새로운 데이터 표준화

# 예측하려는 새로운 물고기 데이터 (길이 25, 무게 150)
# 이 값도 반드시 훈련 데이터 기준(mean, std)으로 표준화해야 한다
# (값 - 평균) / 표준편차 → 표준점수(Z-score)
new = ([25, 150] - mean) / std

# 표준화된 훈련 데이터 시각화

# 표준화된 훈련 데이터 산점도
# train_scaled[:, 0] → 길이(표준화된 값)
# train_scaled[:, 1] → 무게(표준화된 값)
plt.scatter(train_scaled[:,0], train_scaled[:,1])

# 표준화된 새로운 데이터 표시

# 표준화된 새로운 샘플을 그래프에 표시
# marker='^' → 삼각형 모양으로 표시하여 훈련 데이터와 구분
plt.scatter(new[0], new[1], marker='^')

# x축 이름 지정 (길이 특성)
plt.xlabel('length')

# y축 이름 지정 (무게 특성)
plt.ylabel('weight')

# 그래프 출력
plt.show()
```



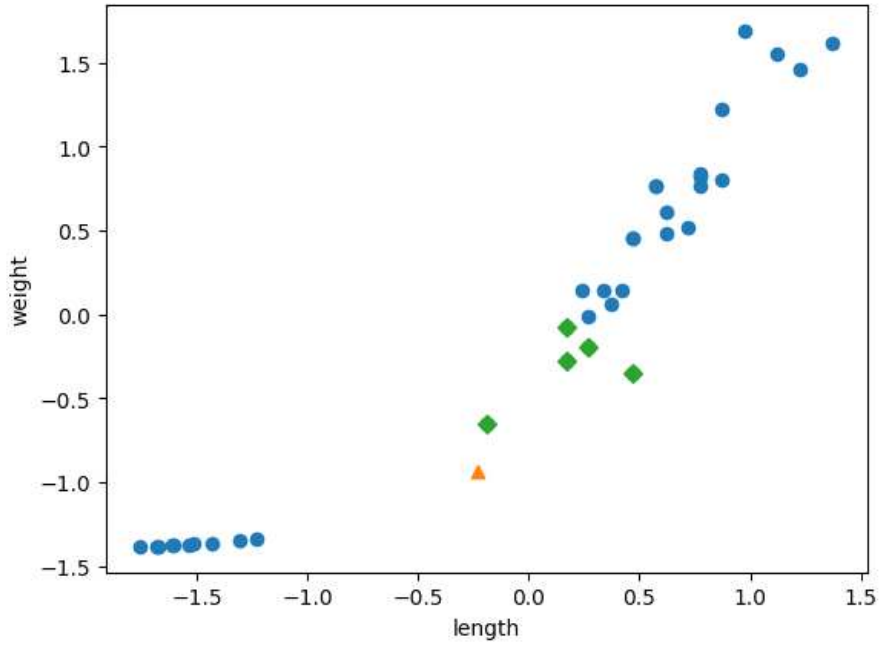
x축과 y툭의 범위가 -1.5~1.5 사이로 바뀜

전처리된 데이터로 다시 k-최근접 이웃 모델을 학습

```
kn.fit(train_scaled, train_target)
kn.score(test_scaled, test_target)
print(kn.predict([new])) #-> [1.] 도미로 예측
```

전처리후 가까운 데이터 시각화

```
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()
```



-> 특성값의 스케일에 민감하지 않고 안정적인 예측을 할 수 있는 모델