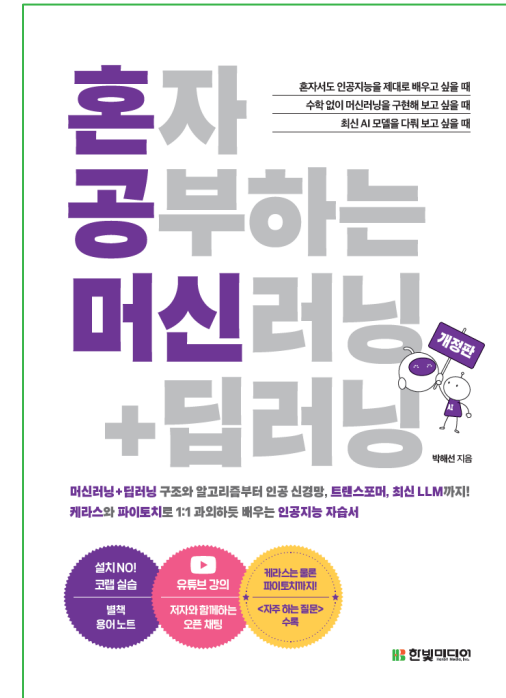


# 혼자 공부하는 머신러닝+딥러닝 (개정판)



# 학습 로드맵



## 머신러닝편

01~06장

딥러닝만 먼저 배우고  
싶다면 01~04장을 읽은 후  
07장으로 건너뛰어도 좋습니다.

START

01

나의 첫 머신러닝



02

데이터 다루기



03

회귀 알고리즘과 모델 규제



2번 보기

04

다양한 분류 알고리즘



05

트리 알고리즘



06

비지도 학습



07

딥러닝을 시작합니다



08

이미지를 위한 인공 신경망



09

텍스트를 위한 인공 신경망

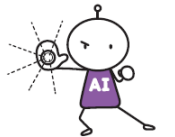


10

언어 모델을 위한 신경망



GOAL



## 딥러닝편

07~10장

07장을 읽은 후 08장과 09장은  
순서대로 읽지 않아도 괜찮습니다. 10  
장을 읽기 전에 07장과 09장을  
읽는 것이 좋습니다.

난이도



# 이 책의 학습 목표

## • CHAPTER 01: 나의 첫 머신러닝

- 인공지능, 머신러닝, 딥러닝의 차이점을 이해합니다.
- 구글 코랩 사용법을 배웁니다.
- 첫 번째 머신러닝 프로그램을 만들고 머신러닝의 기본 작동 원리를 이해합니다.

## • CHAPTER 02: 데이터 다루기

- 머신러닝 알고리즘에 주입할 데이터를 준비하는 방법을 배웁니다.
- 데이터 형태가 알고리즘에 미치는 영향을 이해합니다.

## • CHAPTER 03: 회귀 알고리즘과 모델 규제

- 지도 학습 알고리즘의 한 종류인 회귀 알고리즘에 대해 배웁니다.
- 다양한 선형 회귀 알고리즘의 장단점을 이해합니다.

## • CHAPTER 04: 다양한 분류 알고리즘

- 로지스틱 회귀, 확률적 경사 하강법과 같은 분류 알고리즘을 배웁니다.
- 이진 분류와 다중 분류의 차이를 이해하고 클래스별 확률을 예측합니다.

## • CHAPTER 05: 트리 알고리즘

- 성능이 좋고 이해하기 쉬운 트리 알고리즘에 대해 배웁니다.
- 알고리즘의 성능을 최대화하기 위한 하이퍼파라미터 튜닝을 실습합니다.
- 여러 트리를 합쳐 일반화 성능을 높일 수 있는 앙상블 모델을 배웁니다.

# 이 책의 학습 목표

## • CHAPTER 06: 비지도 학습

- 타깃이 없는 데이터를 사용하는 비지도 학습과 대표적인 알고리즘을 소개합니다.
- 대표적인 군집 알고리즘인 K-평균과 DBSCAN을 배웁니다.
- 대표적인 차원 축소 알고리즘인 주성분 분석(PCA)을 배웁니다.

## • CHAPTER 07: 딥러닝을 시작합니다

- 딥러닝의 핵심 알고리즘인 인공 신경망을 배웁니다.
- 대표적인 인공 신경망 라이브러리인 텐서플로와 케라스를 소개합니다.
- 인공 신경망 모델의 훈련을 돕는 도구를 익힙니다.

## • CHAPTER 08: 이미지를 위한 인공 신경망

- 이미지 분류 문제에 뛰어난 성능을 발휘하는 합성곱 신경망의 개념과 구성 요소에 대해 배웁니다.
- 케라스 API로 합성곱 신경망을 만들어 패션 MNIST 데이터에서 성능을 평가해 봅니다.
- 합성곱 층의 필터와 활성화 출력을 시각화하여 합성곱 신경망이 학습한 내용을 고찰해 봅니다.

## • CHAPTER 09: 텍스트를 위한 인공 신경망

- 텍스트와 시계열 데이터 같은 순차 데이터에 잘 맞는 순환 신경망의 개념과 구성 요소에 대해 배웁니다.
- 케라스 API로 기본적인 순환 신경망에서 고급 순환 신경망을 만들어 영화 감상평을 분류하는 작업에 적용해 봅니다.
- 순환 신경망에서 발생하는 문제점과 이를 극복하기 위한 해결책을 살펴봅니다.

## CHAPTER 03 회귀 알고리즘과 모델 규제

SECTION 3-1 k-최근접 이웃 회귀

SECTION 3-2 선형 회귀

SECTION 3-3 특성 공학과 규제



# CHAPTER 03 회귀 알고리즘과 모델 규제

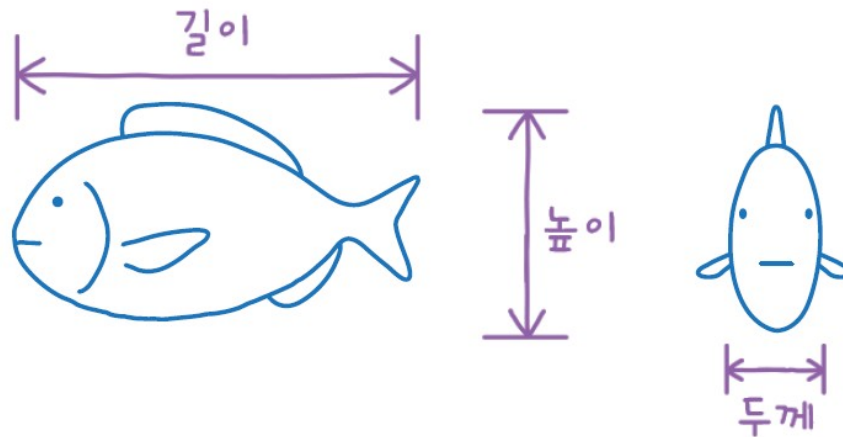
농어의 무게를 예측하라!

## 학습목표

- 지도 학습 알고리즘의 한 종류인 회귀 알고리즘에 대해 배웁니다.
- 다양한 선형 회귀 알고리즘의 장단점을 이해합니다.

## SECTION 3-1 k-최근접 이웃 회귀(1)

- 농어 샘플 56개의 높이, 길이 등 수치로 무게를 예측하기
- 지도 학습 알고리즘은 크게 분류(classification)와 회귀(regression)로 나뉨
  - 분류: 샘플을 몇 개의 클래스 중 하나로 분류
  - 회귀: 클래스 중 하나로 분류하는 것이 아니라 임의의 어떤 숫자를 예측
    - 예) 내년도 경제 성장률을 예측하거나 배달이 도착할 시간을 예측
    - 회귀는 정해진 클래스가 없고 임의의 수치를 출력

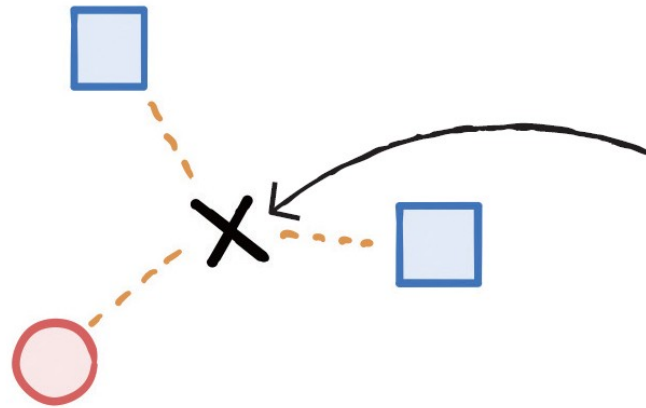


## SECTION 3-1 k-최근접 이웃 회귀(2)

### - k-최근접 이웃 분류 알고리즘

- 예측하려는 샘플에 가장 가까운 샘플  $k$ 개를 선택
- 이 샘플들의 클래스를 확인하여 다수 클래스를 새로운 샘플의 클래스로 예측
- 다음 그림의 왼쪽에 k-최근접 이웃 분류가 잘 나타남
- $k = 3$ (샘플이 3개)이라 가정하면 사각형이 2개로 다수이기 때문에 새로운 샘플의 클래스는 사각형

k-최근접 이웃 분류



최근접 이웃은

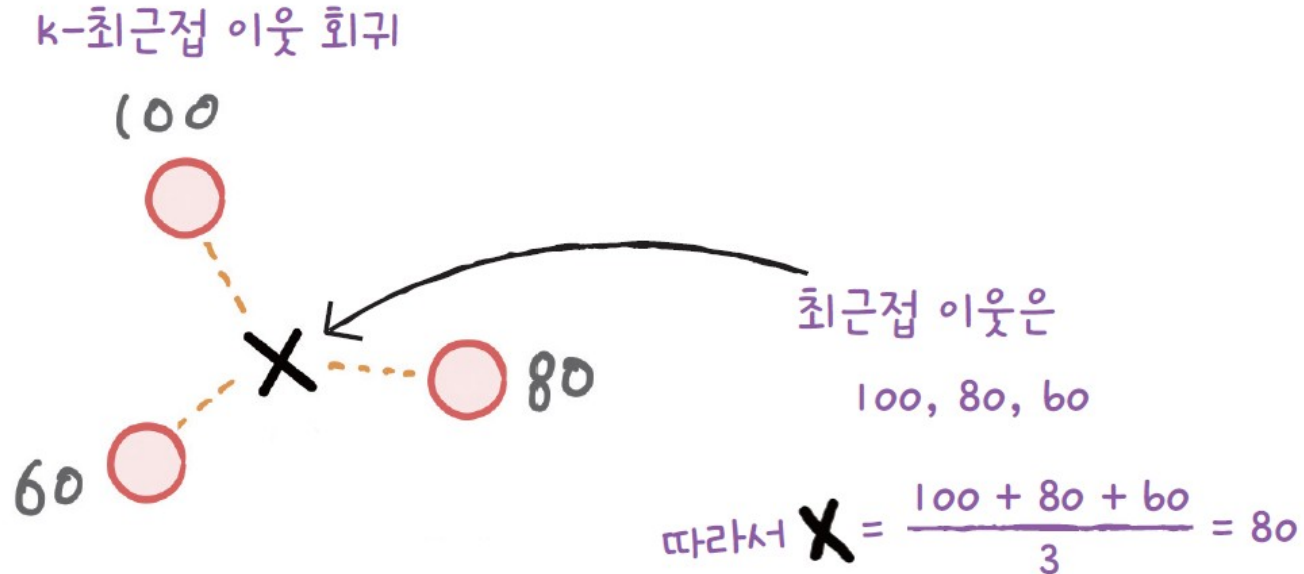
 2개  1개  
따라서 X의 클래스는 



## SECTION 3-1 k-최근접 이웃 회귀(3)

### - k-최근접 이웃 회귀 알고리즘

- 분류와 똑같이 예측하려는 샘플에 가장 가까운 샘플  $k$ 개를 선택
- 회귀이기 때문에 이웃한 샘플의 타깃은 어떤 클래스가 아니라 임의의 수치
- 이웃 샘플의 수치를 사용해 새로운 샘플의 타깃을 예측하기 위해 이 수치들의 평균을 구함
- 그림에서 이웃한 샘플의 타깃값이 각각 100, 80, 60이고 이를 평균하면 샘플의 예측 타깃값은 80



# SECTION 3-1 k-최근접 이웃 회귀(4)

- 데이터 준비

- 훈련 데이터 준비

- 농어의 길이만 있어도 무게를 잘 예측할 수 있다고 가정 (농어의 길이가 특성, 무게가 타겟)
    - 넘파이 배열로 만들기 (소스: [http://bit.ly/perch\\_data](http://bit.ly/perch_data)에서 복사)

```
import numpy as np
perch_length = np.array(
[ 8.4, 13.7, 15.0, 16.2, 17.4, 18.0, 18.7, 19.0, 19.6, 20.0,
 21.0, 21.0, 21.0, 21.3, 22.0, 22.0, 22.0, 22.0, 22.0, 22.5,
 22.5, 22.7, 23.0, 23.5, 24.0, 24.0, 24.6, 25.0, 25.6, 26.5,
 27.3, 27.5, 27.5, 27.5, 28.0, 28.7, 30.0, 32.8, 34.5, 35.0,
 36.5, 36.0, 37.0, 37.0, 39.0, 39.0, 39.0, 40.0, 40.0, 40.0,
 40.0, 42.0, 43.0, 43.0, 43.5, 44.0]
)
perch_weight = np.array(
[ 5.9, 32.0, 40.0, 51.5, 70.0, 100.0, 78.0, 80.0, 85.0,
 85.0,
 110.0, 115.0, 125.0, 130.0, 120.0, 120.0, 130.0, 135.0,
 110.0,
 130.0, 150.0, 145.0, 150.0, 170.0, 225.0, 145.0, 188.0,
 180.0,
 197.0, 218.0, 300.0, 260.0, 265.0, 250.0, 250.0, 300.0,
 320.0,
 514.0, 556.0, 840.0, 685.0, 700.0, 700.0, 690.0, 900.0,
 650.0,
```

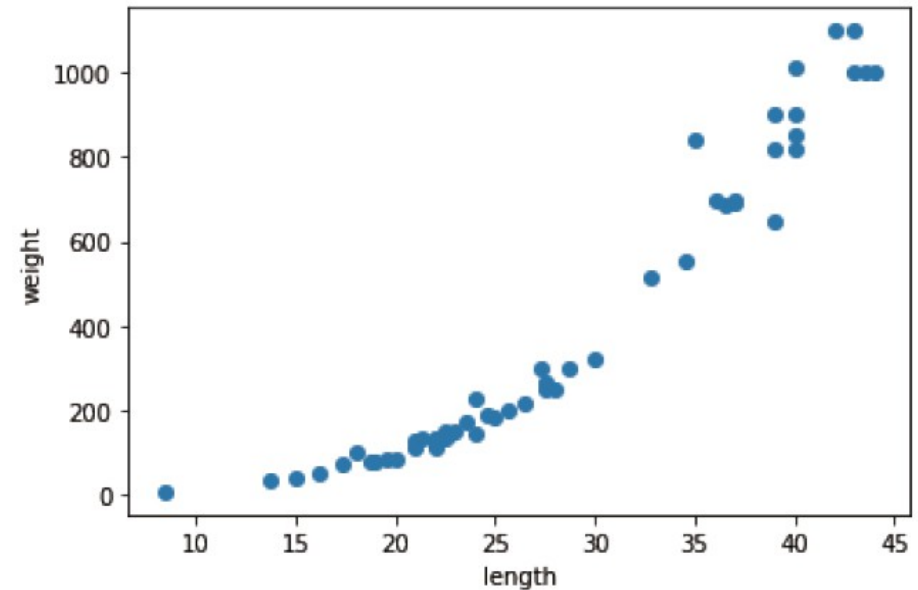
# SECTION 3-1 k-최근접 이웃 회귀(5)

## ○ 데이터 준비

### - 훈련 데이터 준비

- 데이터가 형태 파악을 위해 산점도 그리기
  - 하나의 특성을 사용하기 때문에 특성 데이터를 X축, 타겟 데이터를 Y축
- 맷플롯립을 임포트하고 scatter( ) 함수를 사용하여 산점도 그리기

```
import matplotlib.pyplot as plt
plt.scatter(perch_length, perch_weight)
plt.xlabel('length')
plt.ylabel('weight')
plt.show()
```



# SECTION 3-1 k-최근접 이웃 회귀(6)


## ○ 데이터 준비

### - 훈련 세트와 테스트 세트로 나누기

- 사이킷런의 `train_test_split()` 함수를 사용
- 책과 결과를 동일하게 유지하기 위해 `random_state=42`로 지정

```
from sklearn.model_selection import train_test_split
train_input, test_input, train_target, test_target =
train_test_split(
    perch_length, perch_weight, random_state=42)
```

- 사이킷런에 사용할 훈련 세트는 2차원 배열이어야 함
- `perch_length`가 1차원 배열이기 때문에 이를 나눈 `train_input`과 `test_input`도 1차원 배열
  - 이런 1차원 배열을 1개의 열이 있는 2차원 배열로 바꿔야 함
- 파이썬에서 1차원 배열의 크기는 원소가 1개인 튜플
  - 예를 들어 `[1, 2, 3]`의 크기는 `(3, )`
  - 이를 2차원 배열로 만들기 위해 역지로 하나의 열을 추가하여 배열의 크기가 `(3, 1)`이 됨
  - 배열을 나타내는 방식만 달라졌을 뿐 배열에 있는 원소의 개수는 동일하게 3개

`[[1],`  
`[1, 2, 3]` ~~`[2],`~~   
`[3]]`  
크기 : `(3, )`      크기 : `(3, 1)`

# SECTION 3-1 k-최근접 이웃 회귀(7)

## ○ 데이터 준비

### - 훈련 세트와 테스트 세트로 나누기

- 크기를 바꿀 수 있는 reshape( ) 메서드
  - 2장에서는 2개의 특성을 사용했기 때문에 자연스럽게 열이 2개인 2차원 배열을 사용
  - 이번 예제에서는 특성을 1개만 사용하므로 수동으로 2차원 배열을 만들어 줌
- (4, ) 배열을 (2, 2) 크기로 바꾸기

```
test_array = np.array([1,2,3,4])  
print(test_array.shape)
```

→ (4,)



```
test_array = test_array.reshape(2, 2)  
print(test_array.shape)
```

→ (2, 2)



지정한 크기와 원본 배열의 원소 개수가 달라도 되나?

- reshape() 메서드는 크기가 바뀐 새로운 배열을 반환할 때 지정한 크기가 원본 배열에 있는 원소의 개수와 다르면 에러가 발생
- 예를 들어 다음과 같이 (4, ) 크기의 배열을 (2, 3)으로 바꾸려고 하면 원본 배열의 원소는 4개인데  $2 \times 3 = 6$ 개로 바꾸려고 하기 때문에 에러 발생

## SECTION 3-1 k-최근접 이웃 회귀(8)

### ○ 데이터 준비

#### - 훈련 세트와 테스트 세트로 나누기

- reshape( ) 메서드를 사용해 train\_input과 test\_input을 2차원 배열로 변환
  - train\_input의 크기는 (42, )
  - 이를 2차원 배열인 (42, 1)로 바꾸려면 train\_input.reshape(42, 1)과 같이 사용
- 넘파이의 배열의 크기를 자동으로 지정하는 기능
  - 크기에 -1을 지정하면 나머지 원소 개수로 모두 채우라는 의미
  - 첫 번째 크기를 나머지 원소 개수로 채우고, 두 번째 크기를 1로 하려면 train\_input.reshape(-1, 1)처럼 사용
- reshape( ) 메서드로 배열의 크기 변경하기

```
train_input = train_input.reshape(-1, 1)
test_input = test_input.reshape(-1, 1)
print(train_input.shape, test_input.shape)
```

→ (42, 1) (14, 1)

## SECTION 3-1 k-최근접 이웃 회귀(9)

### ○ 결정계수( $R^2$ )

- 사이킷런에서 k-최근접 이웃 회귀 알고리즘을 구현한 클래스는 KNeighborsRegressor
- 클래스의 사용법은 KNeighborsClassifier와 매우 비슷함
- 객체를 생성하고 fit( ) 메서드로 회귀 모델을 훈련하기

```
from sklearn.neighbors import  
KNeighborsRegressor
```

```
knr = KNeighborsRegressor()
```

```
# k-최근접 이웃 회귀 모델을 훈련합니다  
knr.fit(train_input, train_target)
```

- 테스트

```
print(knr.score(test_input, test_target))
```



0.9928094061010639

- 테스트 세트의 점수는 분류에서는 정확도, 회귀에서는 결정계수( $R^2$ )
  - 만약 타겟의 평균 정도를 예측하는 수준이라면 (즉, 분자와 분모가 비슷해져)  $R^2$ 는 0에 가까워지고, 예측이 타겟에 아주 가까워지면 (분자가 0에 가까워지기 때문에) 1에 가까운 값이 됨

$$R^2 = 1 - \frac{(\text{타겟} - \text{예측})^2 \text{의 합}}{(\text{타겟} - \text{평균})^2 \text{의 합}}$$

## SECTION 3-1 k-최근접 이웃 회귀(10)

### ◦ 결정계수( $R^2$ )

- 타겟과 예측한 값 사이의 차이를 구하여 어느 정도 예측이 벗어났는지 평가
- 사이킷런 sklearn.metrics 패키지의 mean\_absolute\_error: 타겟과 예측의 절대값 오차를 평균하여 반환

```
from sklearn.metrics import mean_absolute_error
```

```
# 테스트 세트에 대한 예측을 만듭니다
```

```
test_prediction = knr.predict(test_input)
```

```
# 테스트 세트에 대한 평균 절대값 오차를 계산합니다
```

```
mae = mean_absolute_error(test_target, test_prediction)  
print(mae)
```



19.157142857142862

- 결과에서 예측이 평균적으로 19g 정도 타겟값과 차이가 나타남



# SECTION 3-1 k-최근접 이웃 회귀(11)

- 과대적합 VS 과소적합

- 앞에서 훈련한 모델을 사용해 훈련 세트의  $R^2$  점수를 확인

```
print(knr.score(train_input, train_target))
```



0.9698823289099255

- 모델을 훈련 세트와 테스트 세트에서 평가하면 두 값 중 보통 훈련 세트의 점수가 조금 더 높게 나옴
- 과대적합(overfitting)
  - 훈련 세트에서 점수가 굉장히 좋았는데 테스트 세트에서는 점수가 굉장히 나쁜 경우
  - 즉, 훈련 세트에만 잘 맞는 모델이라 테스트 세트와 나중에 실전에 투입하여 새로운 샘플에 대한 예측을 만들 때 잘 동작하지 않을 것임
- 과소적합(underfitting)
  - 반대로 훈련 세트보다 테스트 세트의 점수가 높거나 두 점수가 모두 너무 낮은 경우
  - 즉, 모델이 너무 단순하여 훈련 세트에 적절히 훈련되지 않은 경우

## SECTION 3-1 k-최근접 이웃 회귀(12)

### ○ 과대적합 VS 과소적합

- 앞의 k-최근접 이웃 회귀로 평가한 훈련 세트와 테스트 세트의 점수는 훈련 세트보다 테스트 세트의 점수가 높은 과소적합
- 문제 해결
  - 모델을 조금 더 복잡하게(즉, 훈련 세트에 더 잘 맞게) 만들면 테스트 세트의 점수는 조금 낮아질 것임
  - 이웃의 개수  $k$ 를 줄여 k-최근접 이웃 알고리즘으로 모델을 더 복잡하게 만들기
  - 이웃의 개수를 줄이면 훈련 세트에 있는 국지적인 패턴에 민감해지고, 이웃의 개수를 늘리면 데이터 전반에 있는 일반적인 패턴을 따르게 됨

## SECTION 3-1 k-최근접 이웃 회귀(13)

- 과대적합 VS 과소적합

- 사이킷런의 k-최근접 이웃 알고리즘의 기본 k 값은 5를 3으로 낮추기

- n\_neighbors 속성값 변경

```
# 이웃의 개수를 3으로 설정합니다
```

```
knr.n_neighbors = 3
```

```
# 모델을 다시 훈련합니다
```

```
knr.fit(train_input, train_target)
```

```
print(knr.score(train_input, train_target))
```

→ 0.9804899950518966

- 테스트 세트의 점수 확인

```
print(knr.score(test_input, test_target))
```

→ 0.974645996398761

# SECTION 3-1 k-최근접 이웃 회귀(14)

## ○ 회귀 문제 다루기(문제해결 과정)

### - 문제

- 농어의 높이, 길이 등의 수치로 무게를 예측하기(회귀는 임의의 수치를 예측)

### - 해결

- k-최근접 이웃 회귀 모델은 분류와 동일하게 가장 먼저 가까운 k개의 이웃을 찾아 이웃 샘플의 타깃값을 평균하여 이 샘플의 예측값으로 사용
- 사이킷런은 회귀 모델의 점수로  $R^2$ , 즉 결정계수 값을 반환. 이 값은 1에 가까울수록 좋음
- 정량적인 평가는 사이킷런에서 제공하는 다른 평가 도구를 사용할 수 있음(대표적으로 절대값 오차)
- 모델을 훈련하고 나서 훈련 세트와 테스트 세트에 대해 모두 평가 점수를 구할 수 있음
  - 훈련 세트의 점수와 테스트 세트의 점수 차이가 크면 좋지 않음
  - 일반적으로 훈련 세트의 점수가 테스트 세트보다 조금 더 높음
- 과대적합: 만약 테스트 세트의 점수가 너무 낮다면 모델이 훈련 세트에 과도하게 맞춰짐
- 과소적합: 테스트 세트 점수가 너무 높거나 두 점수가 모두 낮은 경우
- 과대적합일 경우 모델을 덜 복잡하게 만들어야 함(k-최근접 이웃의 경우 k 값을 늘림)
- 과소적합일 경우 모델을 더 복잡하게 만들어 줌(k-최근접 이웃의 경우 k 값을 줄임)

## SECTION 3-1 마무리(1)

- 키워드로 끝나는 핵심 포인트

- 회귀는 임의의 수치를 예측하는 문제. 따라서 타깃값도 임의의 수치
- k-최근접 이웃 회귀는 k-최근접 이웃 알고리즘을 사용해 회귀 문제 해결
  - 가장 가까운 이웃 샘플을 찾고 이 샘플들의 타깃값을 평균하여 예측
- 결정계수( $R^2$ )는 대표적인 회귀 문제의 성능 측정 도구
  - 1에 가까울수록 좋고, 0에 가깝다면 성능이 나쁜 모델
- 과대적합은 모델의 훈련 세트 성능이 테스트 세트 성능보다 훨씬 높을 때 발생
  - 모델이 훈련 세트에 너무 집착해서 데이터에 내재된 거시적인 패턴을 감지하지 못함
  - 과소적합은 이와 반대로 훈련 세트와 테스트 세트 성능이 모두 동일하게 낮거나 테스트 세트 성능이 오히려 더 높을 때 발생. 이런 경우 더 복잡한 모델을 사용해 훈련 세트에 잘 맞는 모델을 만들어야 함

## SECTION 3-1 마무리(2)

- 핵심 패키지와 함수
  - scikit-learn
    - KNeighborsRegressor: k-최근접 이웃 회귀 모델을 만드는 사이킷런 클래스
    - mean\_absolute\_error( ): 회귀 모델의 평균 절대값 오차를 계산
  - numpy
    - reshape( ): 배열의 크기를 바꾸는 메서드

## SECTION 3-1 확인 문제

앞서 만든 k-최근접 이웃 회귀 모델의 k 값을 1, 5, 10으로 바꿔가며 훈련하고, 농어의 길이를 5에서 45까지 바꿔가며 예측을 만들어 그래프로 나타내기.  $n$ 이 커짐에 따라 모델이 단순해지는 것을 볼 수 있는가? [노트] 맷플롯립의 plot() 함수는 x축과 y축의 값을 받아 선 그래프를 그려줌

134쪽 참고하여  
1, 2, 3 번 문제  
추가해 주세요.

```
# k-최근접 이웃 회귀 객체를 만듭니다
knr = KNeighborsRegressor()
# 5에서 45까지 x 좌표를 만듭니다
x = np.arange(5, 45).reshape(-1, 1)
# n = 1, 5, 10일 때 예측 결과를 그래프로 그립니다
for n in [1, 5, 10]:
    # 모델을 훈련합니다
    knr.n_neighbors =  # 이 라인의 코드를 완성해 보세요
    knr.fit(train_input, train_target)
    # 지정한 범위 x에 대한 예측을 구합니다
    prediction =  # 이 라인의 코드를 완성해 보세요
    # 훈련 세트와 예측 결과를 그래프로 그립니다
    plt.scatter(train_input, train_target)
    plt.plot(x, prediction)
    plt.show()
```

## SECTION 3-2 선형 회귀(1)

- 50cm 농어의 무게를 예측
  - 앞서 만든 모델을 사용해 이 농어의 무게를 예측하니, 저울에 나온 농어의 무게와 너무 차이가?





## SECTION 3-2 선형 회귀(2)

- k-최근접 이웃의 한계
  - 문제를 재현하기 위해 먼저 1절에서 사용한 데이터와 모델을 준비(소스 [http://bit.ly/perch\\_data](http://bit.ly/perch_data))

```
import numpy as np
perch_length = np.array(
    [ 8.4, 13.7, 15.0, 16.2, 17.4, 18.0, 18.7, 19.0, 19.6, 20.0,
      21.0, 21.0, 21.0, 21.3, 22.0, 22.0, 22.0, 22.0, 22.0, 22.5,
      22.5, 22.7, 23.0, 23.5, 24.0, 24.0, 24.6, 25.0, 25.6, 26.5,
      27.3, 27.5, 27.5, 27.5, 28.0, 28.7, 30.0, 32.8, 34.5, 35.0,
      36.5, 36.0, 37.0, 37.0, 39.0, 39.0, 39.0, 40.0, 40.0, 40.0,
      40.0, 42.0, 43.0, 43.0, 43.5, 44.0]
)
perch_weight = np.array(
    [ 5.9, 32.0, 40.0, 51.5, 70.0, 100.0, 78.0, 80.0, 85.0, 85.0,
      110.0, 115.0, 125.0, 130.0, 120.0, 120.0, 130.0, 135.0, 110.0,
      130.0, 150.0, 145.0, 150.0, 170.0, 225.0, 145.0, 188.0, 180.0,
      197.0, 218.0, 300.0, 260.0, 265.0, 250.0, 250.0, 300.0, 320.0,
      514.0, 556.0, 840.0, 685.0, 700.0, 700.0, 690.0, 900.0, 650.0,
      820.0, 850.0, 900.0, 1015.0, 820.0, 1100.0, 1000.0, 1100.0,
      1000.0, 1000.0]
)
```

## SECTION 3-2 선형 회귀(3)

- k-최근접 이웃의 한계

- 데이터를 훈련 세트와 테스트 세트로 나누고, 특성 데이터는 2 차원 배열로 변환

```
from sklearn.model_selection import train_test_split
# 훈련 세트와 테스트 세트로 나눕니다
train_input, test_input, train_target, test_target = train_test_split(
    perch_length, perch_weight, random_state=42)
# 훈련 세트와 테스트 세트를 2차원 배열로 바꿉니다
train_input = train_input.reshape(-1, 1)
test_input = test_input.reshape(-1, 1)
```

- 최근접 이웃 개수를 3으로 하는 모델을 훈련

```
from sklearn.neighbors import KNeighborsRegressor
knr = KNeighborsRegressor(n_neighbors=3)
# k-최근접 이웃 회귀 모델을 훈련합니다
knr.fit(train_input, train_target)
```

- 이 모델을 사용해 길이가 50cm인 농어의 무게를 예측

```
print(knr.predict([[50]]))
```

 → [1033.33333333]

- 50cm 농어의 무게를 1,033g 정도로 예측. 그런데 실제 이 농어의 무게는 훨씬 더 많다고 함

## SECTION 3-2 선형 회귀(4)

- k-최근접 이웃의 한계

- 사이킷런의 k-최근접 이웃 모델의 `kneighbors()` 메서드를 사용하여, 훈련 세트와 50cm 농어 그리고 이 농어의 최근접 이웃을 산점도에 표시

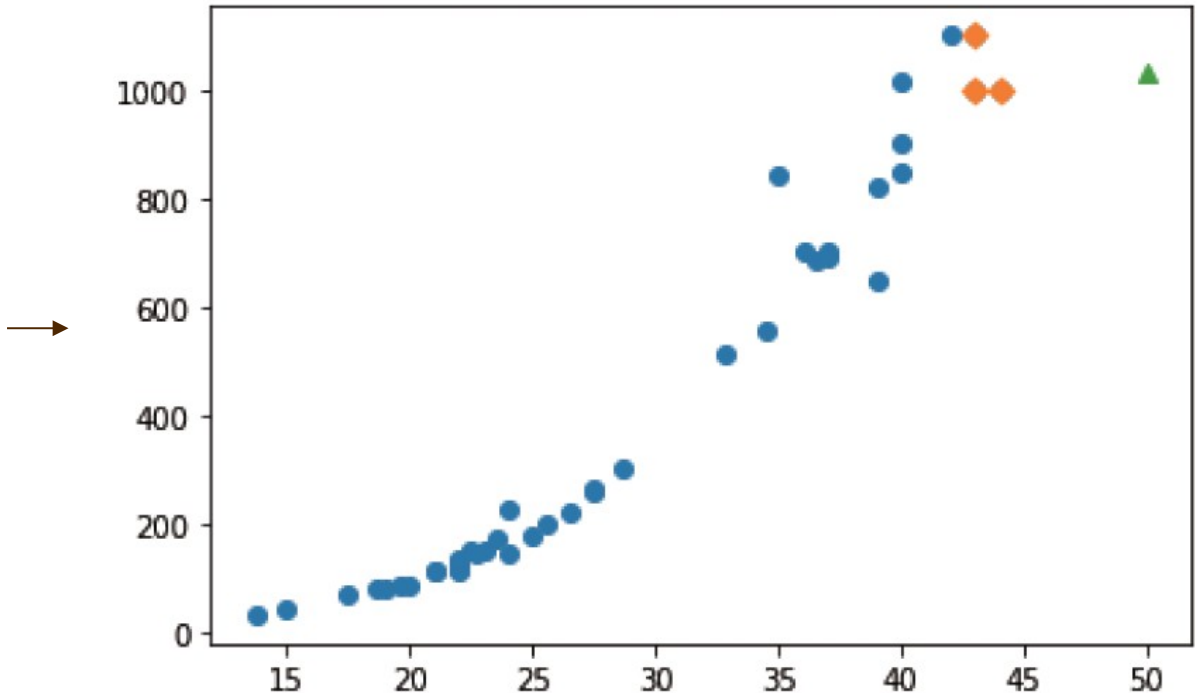
```
import matplotlib.pyplot as plt

# 50cm 농어의 이웃을 구합니다
distances, indexes = knr.kneighbors([[50]])

# 훈련 세트의 산점도를 그립니다
plt.scatter(train_input, train_target)

# 훈련 세트 중에서 이웃 샘플만 다시 그립니다
plt.scatter(train_input[indexes],
            train_target[indexes], marker='D')

# 50cm 농어 데이터
plt.scatter(50, 1033, marker='^')
plt.show()
```



- 산점도를 보면 길이가 커질수록 농어의 무게가 증가하는 경향이 보임
- 하지만 50cm 농어에서 가장 가까운 것은 45cm 근방이기 때문에 k-최근접 이웃 알고리즘은 이 샘플들의 무게를 평균

## SECTION 3-2 선형 회귀(5)

### ○ k-최근접 이웃의 한계

- 이웃 샘플의 타깃 평균

```
print(np.mean(train_target[indexes]))
```



1033.3333333333333

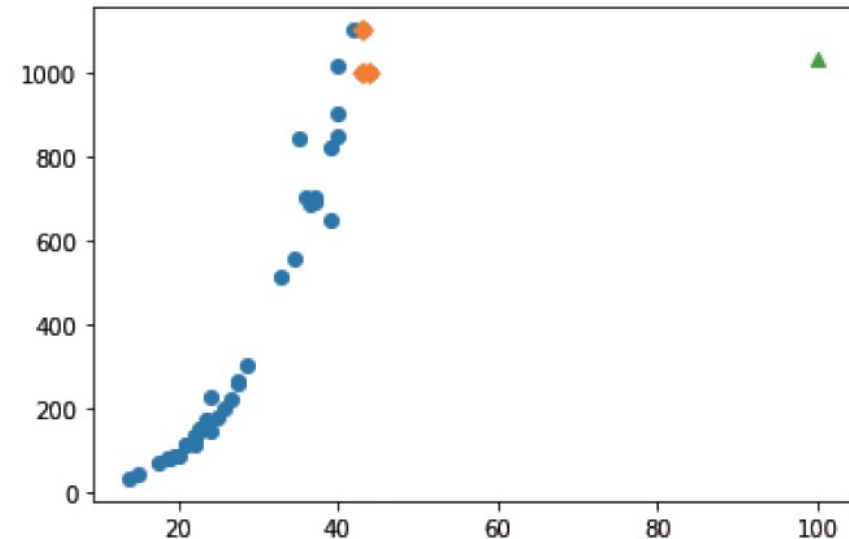
- 모델이 예측했던 값과 정확히 일치
- k-최근접 이웃 회귀는 가장 가까운 샘플들을 찾아 타깃을 평균  
따라서 새로운 샘플이 훈련 세트의 범위를 벗어나면 엉뚱한 값을 예측할 수 있음
- 예를 들어 길이가 100cm인 농어도 여전히 1,033g으로 예측

```
print(knr.predict([[100]]))
```



[1033.33333333]

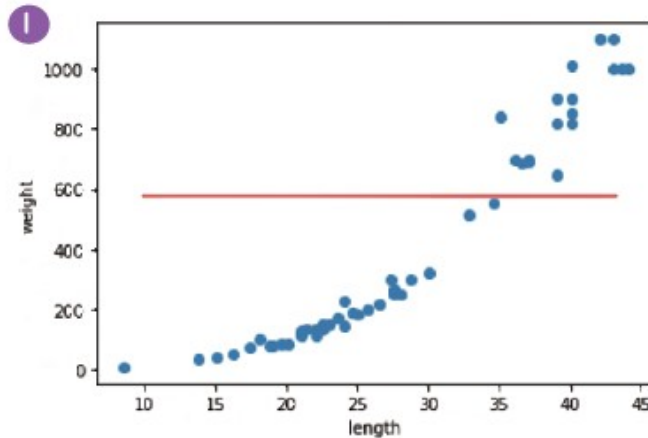
```
# 100cm 농어의 이웃을 구합니다
distances, indexes = knr.kneighbors([[100]])
# 훈련 세트의 산점도를 그림니다
plt.scatter(train_input, train_target)
# 훈련 세트 중에서 이웃 샘플만 다시 그림니다
plt.scatter(train_input[indexes],
            train_target[indexes], marker='D')
# 100cm 농어 데이터
plt.scatter(100, 1033, marker='^')
plt.show()
```



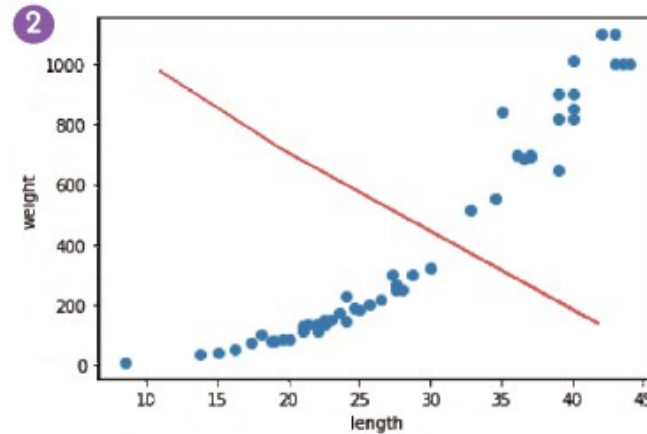
## SECTION 3-2 선형 회귀(6)

### ○ 선형 회귀(linear regression)

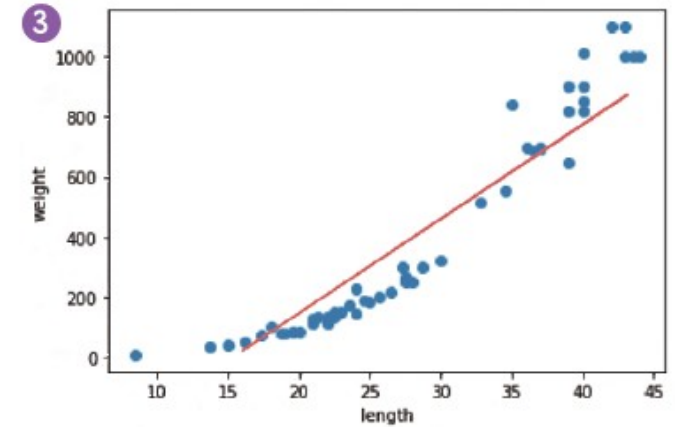
- 선형 회귀는 널리 사용되는 대표적인 회귀 알고리즘
- 비교적 간단하고 성능이 뛰어나기 때문에 맨 처음 배우는 머신러닝 알고리즘 중 하나
- 선형이란 말에서 짐작할 수 있듯이 특성이 하나인 경우 어떤 직선을 학습하는 알고리즘



- ▲ 그래프 ①은 모든 농어의 무게를 하나로 예측.  
이 직선의 위치가 만약 훈련 세트의 평균에 가깝다면  $R^2$ 는 0에 가까운 값이 됨



- ▲ 그래프 ②는 완전히 반대로 예측  
길이가 작은 농어의 무게가 높고  
길이가 큰 농어의 무게가 낮음.  
이렇게 예측을 반대로 하면  $R^2$ 가 음수가 될 수 있음



- ▲ 그래프 ③이 가장 적합한 직선.  
이런 직선을 머신러닝 알고리즘이  
자동으로 찾을 수 있음

## SECTION 3-2 선형 회귀(7)

### ○ 선형 회귀(linear regression)

- 사이킷런은 sklearn.linear\_model 패키지 아래에 LinearRegression 클래스로 선형 회귀 알고리즘을 구현
- 이 클래스의 객체를 만들어 훈련
- 사이킷런의 모델 클래스들은 훈련, 평가, 예측하는 메서드 이름이 모두 동일
  - 즉, LinearRegression 클래스에도 fit( ), score( ), predict( ) 메서드가 있음

```
from sklearn.linear_model import LinearRegression  
lr = LinearRegression()
```

```
# 선형 회귀 모델을 훈련합니다  
lr.fit(train_input, train_target)
```

```
# 50cm 농어에 대해 예측합니다  
print(lr.predict([[50]]))
```

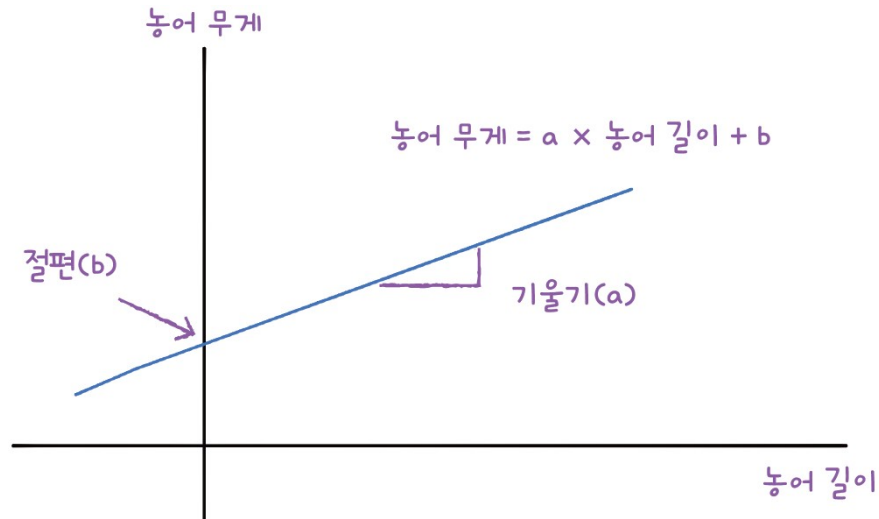


[1241.83860323]

## SECTION 3-2 선형 회귀(8)

- 선형 회귀(linear regression)

- k-최근접 이웃 회귀를 사용했을 때와 달리 선형 회귀는 50cm 농어의 무게를 아주 높게 예측
- 선형 회귀가 학습한 직선을 그려 보고 어떻게 이런 값이 나왔는지 알아보기



- LinearRegression 클래스가 찾은  $a$ 와  $b$ 는 lr 객체의 `coef_`와 `intercept_` 속성에 저장돼 있음

```
print(lr.coef_, lr.intercept_)
```



```
[39.01714496] -709.0186449535477
```

## SECTION 3-2 선형 회귀(8)

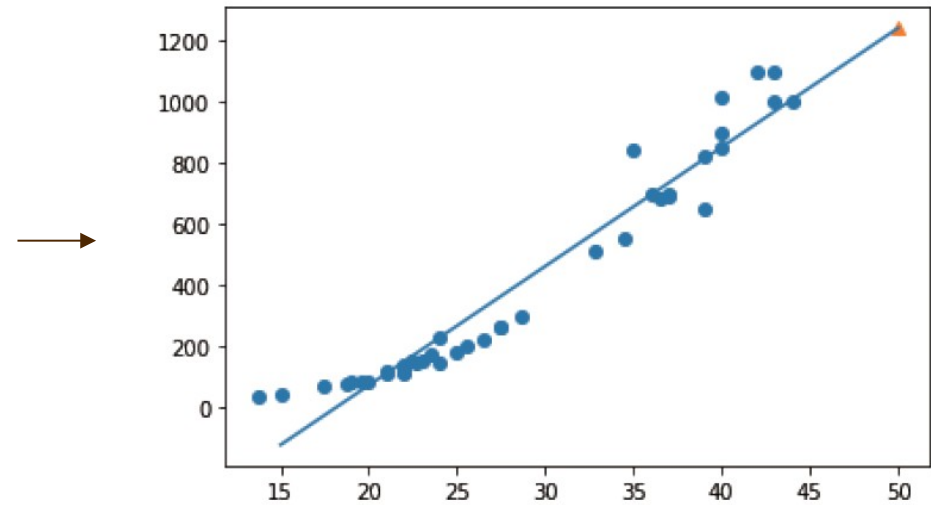
### ○ 선형 회귀(linear regression)

- 농어의 길이 15에서 50까지 직선으로 나타내고, 훈련 세트의 산점도 확인
  - 이 직선을 그리려면 앞에서 구한 기울기와 절편을 사용하여 (15, 15×39-709)와 (50, 50×39-709) 두 점을 이으면 됨

```
# 훈련 세트의 산점도를 그립니다
plt.scatter(train_input, train_target)

# 15에서 50까지 1차 방정식 그래프를 그립니다
plt.plot([15, 50], [15*lr.coef_+lr.intercept_,
                    50*lr.coef_+lr.intercept_])

# 50cm 농어 데이터
plt.scatter(50, 1241.8, marker='^')
plt.show()
```



- 훈련 세트와 테스트 세트에 대한  $R^2$  점수 확인

```
print(lr.score(train_input, train_target)) # 훈련 세트
print(lr.score(test_input, test_target)) # 테스트 세트
```

→ 0.9398463339976039  
→ 0.8247503123313558

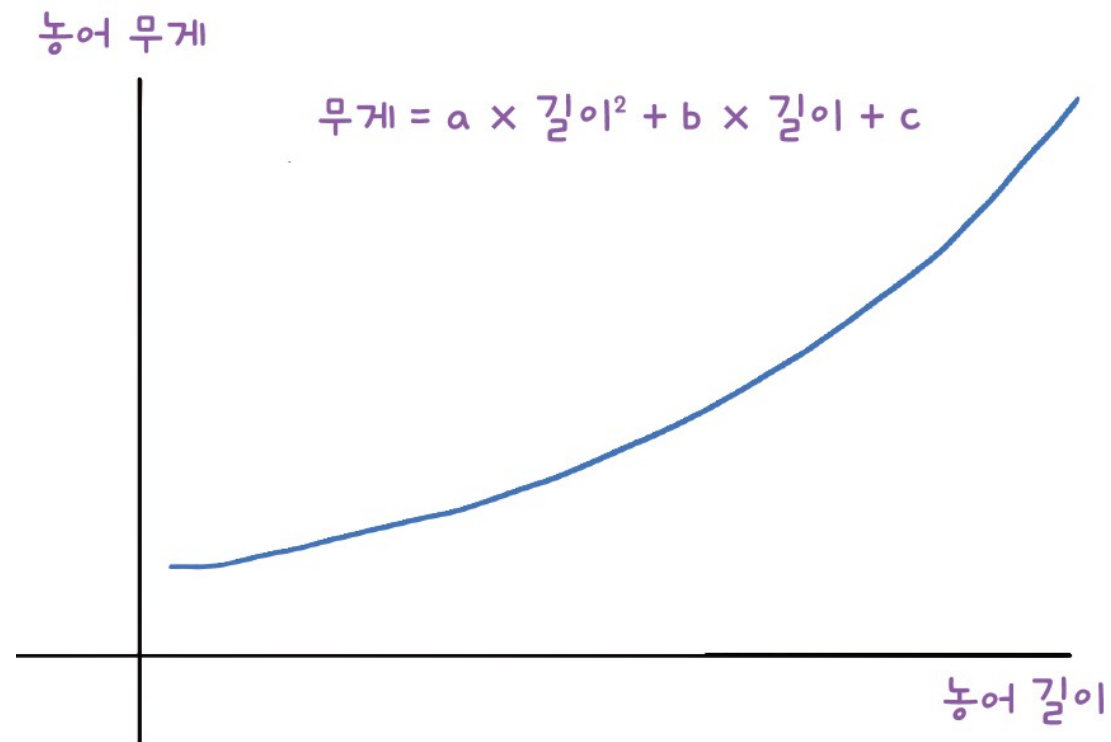
- 훈련 세트와 테스트 세트의 점수가 조금 차이남
- 훈련 세트의 점수도 높지 않아 전체적으로 과소적합되었다고 볼 수 있음



## SECTION 3-2 선형 회귀(9)

- 다항 회귀(polynomial regression)

- 최적의 곡선을 찾기



## SECTION 3-2 선형 회귀(10)

### ○ 다항 회귀(polynomial regression)

- 종어의 길이를 제공해서 원래 데이터 앞에 추가
- `column_stack( )` 함수 사용: `train_input`을 제공한 것과 `train_input` 두 배열을 나란히 붙이면 됨

```
train_poly = np.column_stack((train_input ** 2, train_input))  
test_poly = np.column_stack((test_input ** 2, test_input))
```

- 데이터셋 크기 확인

```
print(train_poly.shape,  
      test_poly.shape)
```

→ (42, 2) (14, 2)

- 원래 특성인 길이를 제공하여 왼쪽 열에 추가했기 때문에  
훈련 세트와 테스트 세트 모두 열이 2개로 늘어남

A diagram illustrating the structure of the data matrix. It shows a 42x2 matrix with two columns. The first column contains squared values of the input, and the second column contains the original input values. The matrix is divided into a top section with 5 rows of numerical data and a bottom section with 37 rows of dots representing the rest of the data. A purple bracket on the left indicates the total number of rows is 42. A purple bracket at the bottom indicates the total number of columns is 2. A purple arrow at the top points to the second column, labeled '제공' (provided).

384.16	19.6
484	22
349.69	18.7
.	.
.	.
.	.
1190.25	34.5

## SECTION 3-2 선형 회귀(11)

- 다항 회귀(polynomial regression)

- train\_poly를 사용해 선형 회귀 모델을 다시 훈련

```
lr = LinearRegression()  
lr.fit(train_poly, train_target)  
print(lr.predict([[50**2, 50]]))
```

→ [1573.98423528]

- 모델이 훈련한 계수와 절편을 출력

```
print(lr.coef_, lr.intercept_)
```

→ [ 1.01433211 -21.55792498] 116.0502107827827

- 이 모델이 학습한 그래프

$$\text{무게} = 1.01 \times \text{길이}^2 - 21.6 \times \text{길이} + 116.05$$

- 이런 방정식을 다항식(polynomial)이라 부르며 다항식을 사용한 선형 회귀를 다항 회귀라 함

## SECTION 3-2 선형 회귀(12)

### ◦ 다항 회귀(polynomial regression)

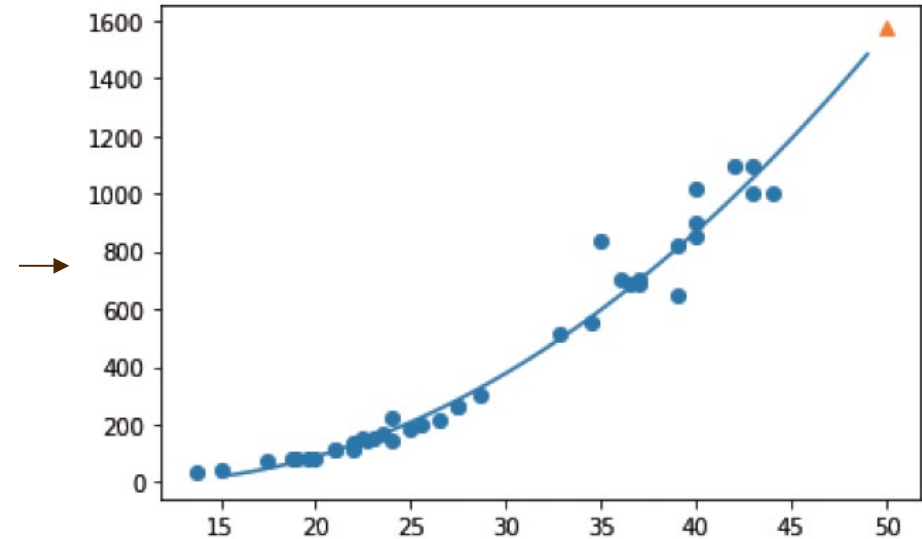
- 훈련 세트의 산점도에 그래프로 그리고
  - 짙은 직선을 이어서 그리면 마치 곡선처럼 표현 가능(여기에서는 1씩 짙게 끊어서 그리기)

```
# 구간별 직선을 그리기 위해 15에서 49까지 정수 배열을 만듭니다
point = np.arange(15, 50)

# 훈련 세트의 산점도를 그립니다
plt.scatter(train_input, train_target)

# 15에서 49까지 2차 방정식 그래프를 그립니다
plt.plot(point, 1.01*point**2 - 21.6*point + 116.05)

# 50cm 놓어 데이터
plt.scatter(50, 1574, marker='^')
```



- 훈련 세트와 테스트 세트의  $R^2$  점수를 평가

```
print(lr.score(train_poly, train_target))
print(lr.score(test_poly, test_target))
```



0.9706807451768623  
0.9775935108325122

## SECTION 3-2 선형 회귀(13)

- 선형 회귀로 훈련 세트 범위 밖의 샘플 예측(문제해결 과정)

- 문제

- k-최근접 이웃 회귀를 사용해서 농어의 무게를 예측했을 때 발생하는 큰 문제는 훈련 세트 범위 밖의 샘플을 예측할 수 없다는 점
- k-최근접 이웃 회귀는 아무리 멀리 떨어져 있더라도 무조건 가장 가까운 샘플의 타깃을 평균하여 예측

- 해결

- 선형 회귀 사용하여 최적의 직선의 방정식을 찾는 것
- 사이킷런의 LinearRegression 클래스를 사용하면 k-최근접 이웃 알고리즘을 사용했을 때와 동일한 방식으로 모델을 훈련하고 예측에 사용할 수 있음
- 직선의 최적의 기울기와 절편값들은 선형 회귀 모델의 coef\_와 intercept\_ 속성에 저장
- 직선 모델은 단순하여 농어의 무게가 음수일 수도 있 0 기 때문에, 다항 회귀를 사용
- 농어의 길이를 제공하여 훈련 세트에 추가한 다음 선형 회귀 모델을 다시 훈련 - 이 모델은 2차 방정식의 그래프 형태를 학습하였고 훈련세트가 분포된 형태를 잘 표현
- 훈련 세트와 테스트 세트의 성능이 단순한 선형 회귀보다 훨씬 높아짐
- 하지만 훈련 세트 성능보다 테스트 세트 성능이 조금 높은 것으로 보아 과소적합된 경향이 아직 남음

## SECTION 3-2 마무리

### ○ 키워드로 끝나는 핵심 포인트

- 선형 회귀는 특성과 타겟 사이의 관계를 가장 잘 나타내는 선형 방정식을 찾음
  - 특성이 하나면 직선 방정식
- 선형 회귀가 찾은 특성과 타겟 사이의 관계는 선형 방정식의 계수 또는 가중치에 저장됨
  - 머신러닝에서 종종 가중치는 방정식의 기울기와 절편을 모두 의미하는 경우가 많음
- 모델 파라미터는 선형 회귀가 찾은 가중치처럼 머신러닝 모델이 특성에서 학습한 파라미터를 말함
- 다항 회귀는 다항식을 사용하여 특성과 타겟 사이의 관계를 나타냄
  - 이 함수는 비선형일 수 있지만 여전히 선형 회귀로 표현할 수 있음

### ○ 핵심 패키지와 함수

- scikit-learn
  - LinearRegression: 사이킷런의 선형 회귀 클래스

## SECTION 3-2 확인 문제

1. 선형 회귀 모델이 찾은 방정식의 계수를 무엇이라고 부르는가?

- ① 회귀 파라미터                      ② 선형 파라미터
- ③ 학습 파라미터                      ④ 모델 파라미터

2. 사이킷런에서 다항 회귀 모델을 훈련할 수 있는 클래스는 무엇인가?

- ① LinearRegression                      ② PolynomialRegression
- ③ KNeighborsClassifier                      ④ PolynomialClassifier

## SECTION 3-2 확인 문제

3. 다음 중 사이킷런의 모델 클래스에서 제공하는 메서드가 아닌것은 무엇인가?

- ① fit()                      ② score()
- ③ evaluate()              ④ predict()

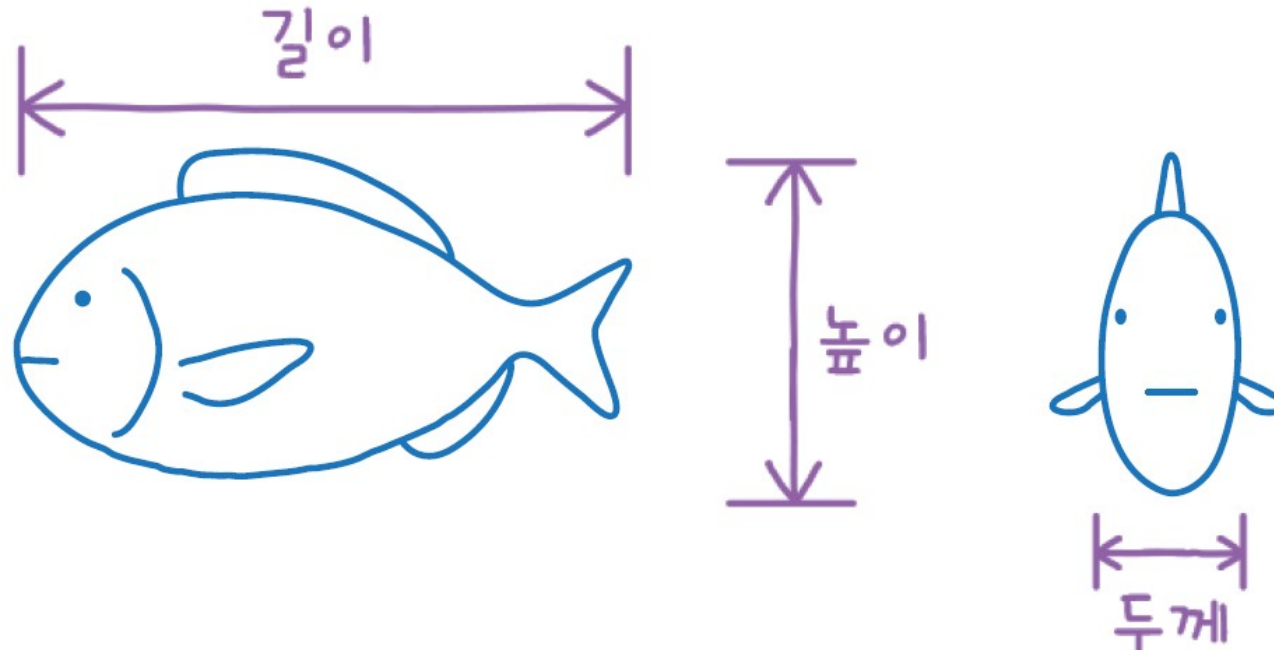
4. 사이킷런에서 다항 회귀 모델을 훈련할 수 있는 클래스는 무엇인가?

- ① LinearRegression                      ② PolynomialRegression
- ③ KNeighborsClassifier    ④ PolynomialClassifier



## SECTION 3-3 특성 공학과 규제(1)

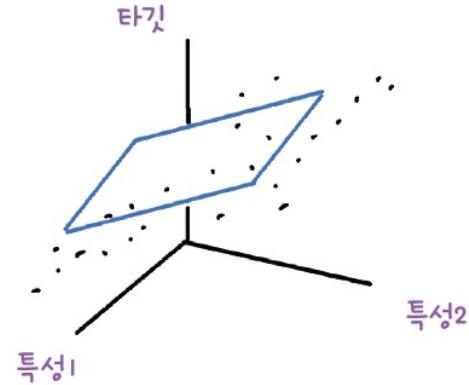
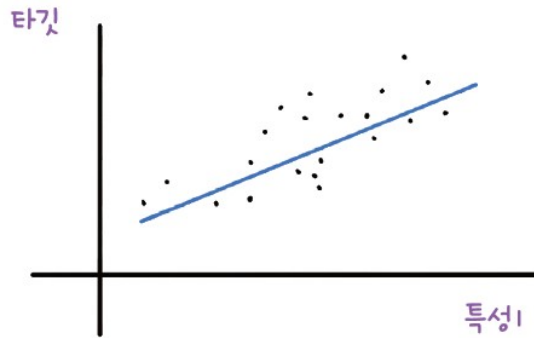
- 선형 회귀는 특성이 많을수록 효과가 커짐
- PolynomialFeatures 클래스에 농어의 높이와 두께 특성도 추가하여 과소적합 해소



## SECTION 3-3 특성 공학과 규제(2)

### ◦ 다중 회귀(multiple regression)

- 여러 개의 특성을 사용한 선형 회귀
- 오른쪽 그림처럼 특성이 2개면 타깃값과 함께 3차원 공간을 형성하고 선형 회귀 방정식 ' $\text{타깃} = a \times \text{특성1} + b \times \text{특성2} + \text{절편}$ '은 평면이 됨



- 특성이 많은 고차원에서는 선형 회귀가 매우 복잡한 모델을 표현
- 특성 공학(feature engineering): 기존의 특성을 사용해 새로운 특성을 뽑아내는 작업
- 예제에서는 농어의 길이뿐만 아니라 농어의 높이와 두께도 함께 사용하고, 이전 절에서처럼 3개의 특성을 각각 제공하여 추가
- 각 특성을 서로 곱해서 또 다른 특성을 생성. 즉, '농어 길이 × 농어 높이'를 새로운 특성으로 생성

## SECTION 3-3 특성 공학과 규제(3)

### ○ 데이터 준비

#### - 판다스(pandas)의 데이터프레임(dataframe)

- 놓여 데이터를 인터넷에서 내려받아 넘파이 배열로 변환하여 선형 회귀 모델을 훈련
- 전체 파일 내용: 웹 브라우저로 [https://bit.ly/perch\\_csv\\_data](https://bit.ly/perch_csv_data)에 접속
- 판다스의 read\_csv() 함수에 주소 삽입
- read\_csv() 함수로 데이터프레임을 만든 다음 head() 메서드를 사용해 처음 다섯 개의 행 출력

CSV 파일

```
length, height, width  
8.4, 2.11, 1.41  
13.7, 3.53, 2.0  
⋮
```

판다스 데이터프레임

pd.read\_csv()

다섯 개 행 출력

head()

```
import pandas as pd # pd는 관례적으로 사용하는 판다스의 별칭입니다  
perch_full = pd.read_csv('https://bit.ly/perch_csv_data')  
perch_full.head()
```

	length	height	width
0	8.4	2.11	1.41
1	13.7	3.53	2.00
2	15.0	3.82	2.43
3	16.2	4.59	2.63
4	17.4	4.59	2.94

## SECTION 3-3 특성 공학과 규제(4)

- 데이터 준비

- 타깃 데이터 준비: 이전과 같이 소스 [http://bit.ly/perch\\_data](http://bit.ly/perch_data)에서 복사

```
import numpy as np
perch_weight = np.array(
    [ 5.9, 32.0, 40.0, 51.5, 70.0, 100.0, 78.0, 80.0, 85.0, 85.0,
      110.0, 115.0, 125.0, 130.0, 120.0, 120.0, 130.0, 135.0, 110.0,
      130.0, 150.0, 145.0, 150.0, 170.0, 225.0, 145.0, 188.0, 180.0,
      197.0, 218.0, 300.0, 260.0, 265.0, 250.0, 250.0, 300.0, 320.0,
      514.0, 556.0, 840.0, 685.0, 700.0, 700.0, 690.0, 900.0, 650.0,
      820.0, 850.0, 900.0, 1015.0, 820.0, 1100.0, 1000.0, 1100.0,
      1000.0, 1000.0]
)
```

- perch\_full과 perch\_weight를 훈련 세트와 테스트 세트로 나누기

```
from sklearn.model_selection import train_test_split
train_input, test_input, train_target, test_target = train_test_split(
    perch_full, perch_weight, random_state=42)
```

## SECTION 3-3 특성 공학과 규제(5)

### 사이킷런의 변환기(transformer)

- 특성을 만들거나 전처리하기 위한 사이킷런의 다양한 클래스
- 사이킷런의 모델 클래스에 일관된 `fit()`, `score()`, `predict()` 메서드가 있는 것처럼 변환기 클래스는 모두 `fit()`, `transform()` 메서드를 제공
- 앞서 배운 `LinearRegression` 같은 사이킷런의 모델 클래스는 추정기(estimator)라고도 부름
- `sklearn.preprocessing` 패키지의 `PolynomialFeatures` 클래스

```
from sklearn.preprocessing import PolynomialFeatures
```

- 2개의 특성 2와 3으로 이루어진 샘플 하나를 적용

이 클래스의 객체를 만든 다음 `fit()`, `transform()` 메서드를 차례대로 호출

```
poly = PolynomialFeatures()  
poly.fit([[2, 3]])  
print(poly.transform([[2, 3]]))
```



`[[1. 2. 3. 4. 6. 9.]]`

- `fit()` 메서드는 새롭게 만들 특성 조합을 찾고 `transform()` 메서드는 실제로 데이터를 변환  
변환기는 입력데이터를 변환하는 데 타깃 데이터가 필요하지 않으므로 모델 클래스와는 다르게 `fit()` 메서드에 입력 데이터만 전달  
즉, 여기에서는 2개의 특성(원소)을 가진 샘플 `[2, 3]`이 6개의 특성을 가진 샘플 `[1. 2. 3. 4. 6. 9.]`로 바뀜

## SECTION 3-3 특성 공학과 규제(6)

### ○ 사이킷런의 변환기(transformer)

- PolynomialFeatures 클래스는 기본적으로 각 특성을 제공한 항을 추가하고 특성끼리 서로 곱한 항을 추가.  
2와 3을 각기 제공한 4와 9가 추가되었고, 2와 3을 곱한 6이 추가됨

- 1은 왜 추가되었을까?

$$\text{무게} = a \times \text{길이} + b \times \text{높이} + c \times \text{두께} + d \times 1$$

- 선형 방정식의 절편은 항상 값이 1인 특성과 곱해지는 계수라고 볼 수 있으며, 특성은 (길이, 높이, 두께, 1)이 됨
- 사이킷런의 선형 모델은 자동으로 절편을 추가하므로 굳이 이렇게 특성을 만들 필요가 없음

- include\_bias=False로 지정하여 다시 특성을 변환

절편을 위한 항이 제거되고 특성의 제곱과 특성끼리 곱한 항만 추가됨

```
poly =  
PolynomialFeatures(include_bias=False)  
poly.fit([[2, 3]])  
print(poly.transform([[2, 3]]))
```

→ `[[2. 3. 4. 6. 9.]]`

## SECTION 3-3 특성 공학과 규제(7)

- 사이킷런의 변환기(transformer)

- train\_input을 변환한 데이터를 train\_poly에 저장하고 이 배열의 크기를 확인

```
poly =  
PolynomialFeatures(include_bias=False)  
poly.fit(train_input)  
train_poly = poly.transform(train_input)  
print(train_poly.shape)
```

→ (42, 9)

- get\_feature\_names\_out( ) 메서드를 호출하여 9개의 특성이 각각 어떤 입력의 조합으로 만들어졌는지 알려줌

```
poly.get_feature_names_out()  
( )
```

→ array(['length', ' height', ' width', 'length^2', 'length height',  
'length width', ' height^2', ' height width', ' width^2'],  
dtype=object)

- 테스트 세트를 변환

```
test_poly = poly.transform(test_input)
```

## SECTION 3-3 특성 공학과 규제(8)

### 다중 회귀 모델 훈련하기

- 사이킷런의 LinearRegression 클래스를 импорт
- 앞에서 만든 train\_poly를 사용해 모델을 훈련시켜 회귀 모델을 훈련

```
from sklearn.linear_model import  
LinearRegression  
lr = LinearRegression()  
lr.fit(train_poly, train_target)  
print(lr.score(train_poly, train_target))
```

→ 0.9903183436982124

- 테스트 세트 점수 확인

```
print(lr.score(test_poly, test_target))
```

→ 0.9714559911594134

- PolynomialFeatures 클래스의 degree 매개변수를 사용하여 필요한 고차항의 최대 차수를 지정
- 5제곱까지 특성을 만들어 출력

```
poly = PolynomialFeatures(degree=5,  
include_bias=False)  
poly.fit(train_input)  
train_poly = poly.transform(train_input)  
test_poly = poly.transform(test_input)  
print(train_poly.shape)
```

→ (42, 55)



특성의 개수

train\_poly 배열의 열의 개수



## SECTION 3-3 특성 공학과 규제(9)

### ○ 다중 회귀 모델 훈련하기

- 이 데이터를 사용해 선형 회귀 모델을 다시 훈련

```
lr.fit(train_poly, train_target)
print(lr.score(train_poly, train_target))
```

→ 0.999999999999991098

- 테스트 세트 점수 확인

```
print(lr.score(test_poly, test_target))
```

→ -144.40579242684848 매우 큰 음수가 나옴

- 특성의 개수를 크게 늘리면 선형 모델은 아주 강력해져, 훈련 세트에 대해 거의 완벽하게 학습
- 하지만 이런 모델은 훈련 세트에 너무 과대적합되므로 테스트 세트에서는 형편없는 점수를 만들게 됨
- 이 문제를 해결하기 위해 다시 특성을 줄여야 함
  - 이런 상황은 과대적합을 줄이는 또 다른 방법을 배워 볼 좋은 기회

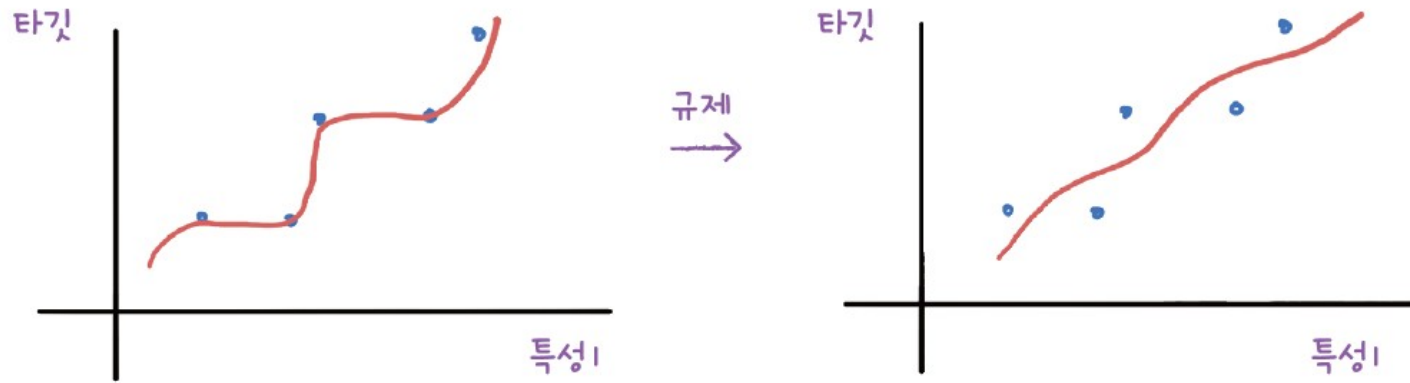
### ✚ 여기서 잠깐 샘플 개수보다 특성이 많다면 어떨까?

- 여기에서 사용한 훈련 세트의 샘플 개수는 42개 밖에 되지 않음
- 42개의 샘플을 55개의 특성으로 훈련하면 완벽하게 학습할 수 있는 것이 당연
- 예를 들어 42개의 참새를 맞추기 위해 딱 한 번 새총을 쏘아 한다면 참새 떼 중앙을 겨냥하여 가능한 한 맞출 가능성을 높여야 함
- 하지만 55번이나 쏠 수 있다면 한 번에 하나씩 모든 참새를 맞출 수 있음

## SECTION 3-3 특성 공학과 규제(10)

### ○ 규제(regularization)

- 머신러닝 모델이 훈련 세트를 너무 과도하게 학습하지 못하도록 회향하는 것
- 즉, 모델이 훈련 세트에 과대적합되지 않도록 만드는 것
- 선형 회귀 모델의 경우 특성에 곱해지는 계수(또는 기울기)의 크기를 작게 만드는 일
- 아래 그림에서 왼쪽은 훈련 세트를 과도하게 학습했고, 오른쪽은 기울기를 줄여 보다 보편적인 패턴을 학습



- 앞서 55개의 특성으로 훈련한 선형 회귀 모델의 계수를 규제하여 훈련 세트의 점수를 낮추고 대신 테스트 세트의 점수를 높이기

- 훈련 세트에서 학습한 평균과 표준편차는 StandardScaler 클래스 객체의 mean\_, scale\_ 속성에 저장됨
- 특성마다 계산하므로 55개의 평균과 표준 편차가 들어 있음

## SECTION 3-3 특성 공학과 규제(11)

### ○ 규제(regularization)

- 특성의 스케일이 정규화되지 않으면 여기에 곱해지는 계수 값도 차이남
- 일반적으로 선형 회귀 모델에 규제를 적용할 때 계수 값의 크기가 서로 많이 다르면 공정하게 제어되지 않음
- 규제를 적용하기 전에 먼저 정규화 필요
  - 2장에서는 평균과 표준편차를 직접 구해 특성을 표준점수로 바꾸었음
- 이번에는 사이킷런에서 제공하는 StandardScaler 클래스를 사용
  - StandardScaler 클래스의 객체 ss를 초기화한 후 PolynomialFeatures 클래스로 만든 train\_poly를 사용해 이 객체를 훈련 (꼭 훈련 세트와 학습한 변환기를 사용해 테스트 세트까지 변환해야 함)

```
from sklearn.preprocessing import StandardScaler
ss = StandardScaler()
ss.fit(train_poly)
train_scaled = ss.transform(train_poly)
test_scaled = ss.transform(test_poly)
```

- 표준점수로 변환한 train\_scaled와 test\_scaled가 준비됨
  - 선형 회귀 모델에 규제를 추가한 모델
    - 릿지(ridge): 계수를 제공한 값을 기준으로 규제를 적용
    - 라쏘(lasso): 계수의 절댓값을 기준으로 규제를 적용 (계수값을 0으로도 만들 수 있음)

## SECTION 3-3 특성 공학과 규제(12)

### ○ 릿지 회귀

- 릿지와 라쏘 모두 `sklearn.linear_model` 패키지
- 모델 객체를 만들고 `fit()` 메서드에서 훈련한 다음 `score()` 메서드로 평가
- 앞서 준비한 `train_scaled` 데이터로 릿지 모델을 훈련

```
from sklearn.linear_model import Ridge
ridge = Ridge()
ridge.fit(train_scaled, train_target)
print(ridge.score(train_scaled, train_target))
```

→ 0.9896101671037343

- 테스트 세트에 대한 점수 확인

```
print(ridge.score(test_scaled, test_target))
```

→ 0.9790693977615398

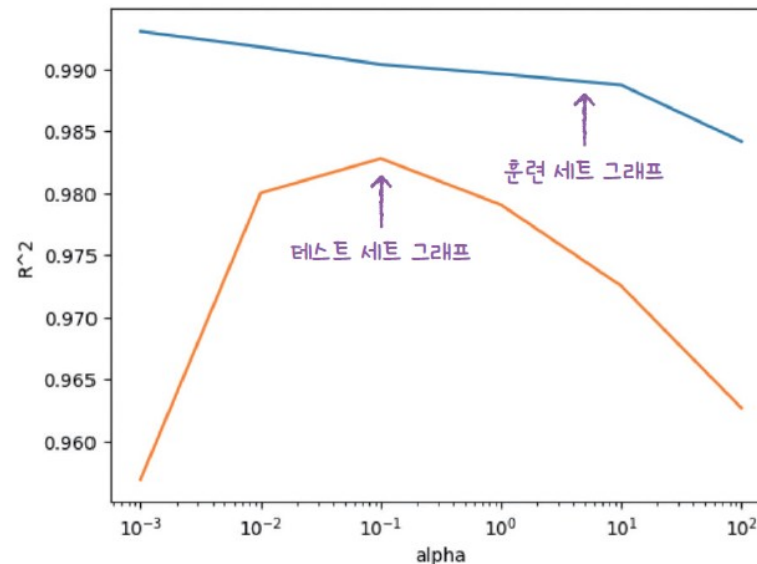
- 릿지와 라쏘 모델을 사용할 때 규제의 양을 임의로 조절 가능
- 모델 객체를 만들 때 `alpha` 매개변수로 규제의 강도를 조절
  - `alpha` 값이 크면 규제 강도가 세지므로 계수 값을 더 줄이고 조금 더 과소적합되도록 유도
  - `alpha` 값이 작으면 계수를 줄이는 역할이 줄어들고 선형 회귀 모델과 유사해지므로 과대적합될 가능성이 큼

## SECTION 3-3 특성 공학과 규제(13)

### ○ 릿지 회귀

- 그래프 그리기
- $\alpha$  값을 0.001부터 10배씩 늘렸기 때문에 이대로 그래프를 그리면 그래프 왼쪽이 너무 촘촘해짐
- $\alpha\_list$ 에 있는 6개의 값을 동일한 간격으로 나타내기 위해서는 X 축을 로그 스케일로 나타내야 함

```
plt.plot(alpha_list, train_score)
plt.plot(alpha_list, test_score)
plt.xscale('log')
plt.xlabel('alpha')
plt.ylabel('R^2')
plt.show()
```



- 그래프의 왼쪽  
훈련 세트와 테스트 세트의 점수 차이가 아주 큼 훈련 세트에는 잘 맞고 테스트 세트에는 형편없는 과대적합의 전형적인 모습
- 그래프 오른쪽  
훈련 세트와 테스트 세트의 점수가 모두 낮아지는 과소적합으로 가는 모습을 보임

### 넘파이 로그 함수

- $\text{np.log}()$ : 자연 상수  $e$ 를 밑으로 하는 자연로그
- $\text{np.log10}()$ : 10 을 밑으로 하는 상용로그

## SECTION 3-3 특성 공학과 규제(14)

- 릿지 회귀

- 적절한 alpha 값은 두 그래프가 가장 가깝고 테스트 세트의 점수가 가장 높은 -1, 즉  $10^{-1}=0.1$
- alpha 값을 0.1로 하여 최종 모델을 훈련

```
ridge = Ridge(alpha=0.1)
ridge.fit(train_scaled, train_target)
print(ridge.score(train_scaled,
train_target))
print(ridge.score(test_scaled,
test_target))
```



0.9903815817570366  
0.9827976465386926

- 이 모델은 훈련 세트와 테스트 세트의 점수가 비슷하게 모두 높고 과대적합과 과소적합 사이에서 균형을 맞춘다

## SECTION 3-3 특성 공학과 규제(15)

### ○ 라쏘 회귀

- 라쏘 모델을 훈련하는 것은 릿지와 유사
- Ridge 클래스를 Lasso 클래스로 바꾸면 됨

```
from sklearn.linear_model import  
Lasso  
lasso = Lasso()  
lasso.fit(train_scaled, train_target)  
print(lasso.score(train_scaled,  
train_target))
```



0.9897898972080961

- 테스트 세트 점수 확인

```
print(lasso.score(test_scaled,  
test_target))
```



0.9800593698421883

## SECTION 3-3 특성 공학과 규제(16)

### ○ 라쏘 회귀

- 라쏘 모델도 alpha 매개변수로 규제의 강도를 조절 가능
- alpha 값을 바꾸어 가며 훈련 세트와 테스트 세트에 대한 점수 계산

```
train_score = []
test_score = []
alpha_list = [0.001, 0.01, 0.1, 1, 10, 100]
for alpha in alpha_list:
    # 라쏘 모델을 만듭니다
    lasso = Lasso(alpha=alpha, max_iter=10000)
    # 라쏘 모델을 훈련합니다
    lasso.fit(train_scaled, train_target)
    # 훈련 점수와 테스트 점수를 저장합니다
    train_score.append(lasso.score(train_scaled,
                                   train_target))
    test_score.append(lasso.score(test_scaled, test_target))
```

#### ✚ 여기서 잠깐 경고(Warning)가 뜨는데 정상인가?

- 라쏘 모델을 훈련할 때 **ConvergenceWarning**이란 경고가 발생할 수 있음
- 사이킷런의 라쏘 모델은 최적의 계수를 찾기 위해 반복적인 계산을 수행하는데, 지정한 반복 횟수가 부족할 때 이런 경고가 발생
- 이 반복 횟수를 충분히 늘리기 위해 max\_iter 매개변수의 값을 10000으로 지정했음
- 필요하면 더 늘릴 수 있지만 이 문제에서는 큰 영향을 끼치지 않음



## SECTION 3-3 특성 공학과 규제(17)

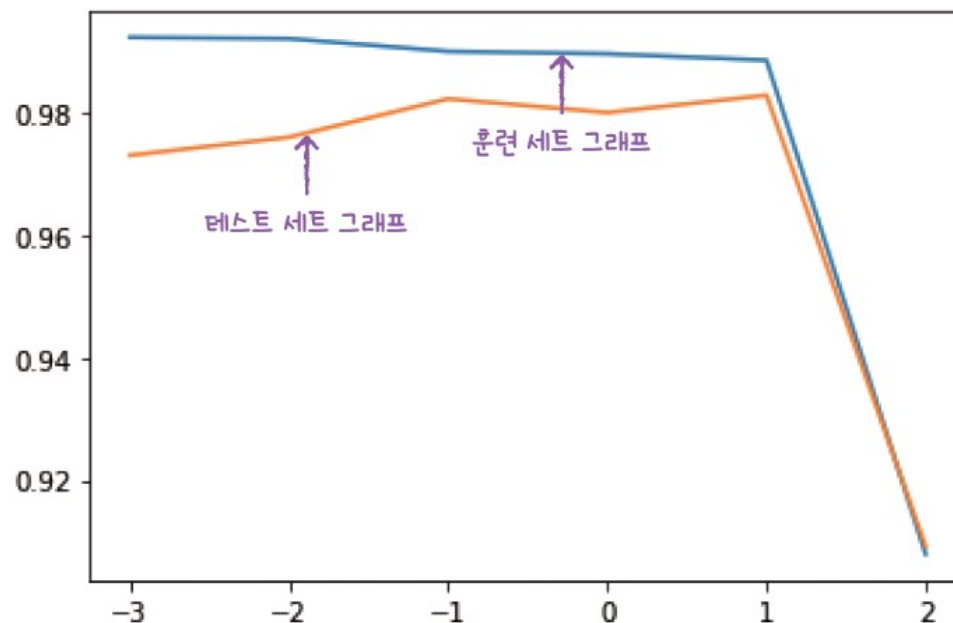
### ○ 라쏘 회귀

- train\_score와 test\_score 리스트를 사용해 그래프 그리기

- X축은 로그 스케일로 바꿈

```
plt.plot(np.log10(alpha_list), train_score)
plt.plot(np.log10(alpha_list), test_score)
plt.show()
```

- 그래프의 왼쪽은 과대적합을 보여주고 있고,  
오른쪽으로 갈수록 훈련 세트와 테스트 세트의 점수가  
좁혀짐가장 오른쪽은 아주 크게 점수가 떨어짐  
이 지점은 분명 과소적합되는 모델일 것임



## SECTION 3-3 특성 공학과 규제(18)

### ○ 라쏘 회귀

- 라쏘 모델에서 최적의  $\alpha$  값은 1, 즉  $10^1=10$
- 이 값으로 다시 모델을 훈련

```
lasso = Lasso(alpha=10)
lasso.fit(train_scaled, train_target)
print(lasso.score(train_scaled,
train_target))
print(lasso.score(test_scaled,
test_target))
```



0.9888067471131867  
0.9824470598706695

- `coef_` 속성에 저장되어 있는 라쏘 모델의 계수 중에 0인 것을 헤아려보기

```
print(np.sum(lasso.coef_ == 0))
```



40

- 55개의 특성을 모델에 주입했지만 라쏘 모델이 사용한 특성은 15개에 불과함  
이런 특징 때문에 라쏘 모델을 유용한 특성을 골라내는 용도로도 사용할 수 있음

## SECTION 3-3 특성 공학과 규제(19)

- 모델의 과대적합을 제어하기(문제해결 과정)

- 문제

- 선형 회귀 알고리즘을 사용해 농어의 무게를 예측하는 모델을 훈련시켰지만 훈련 세트에 과소적합

- 해결

- 농어의 길이뿐만 아니라 높이와 두께도 사용하여 다중 회귀 모델을 훈련
- 또한 다항 특성을 많이 추가하여 훈련 세트에서 거의 완벽에 가까운 점수를 얻는 모델을 훈련
- 특성을 많이 추가하면 선형 회귀는 매우 강력한 성능을 발휘하지만, 특성이 너무 많으면 선형 회귀 모델을 제약하기 위한 도구가 필요
- 이를 위해 릿지 회귀와 라쏘 회귀에 대해 학습
- 사이킷런을 사용해 다중 회귀 모델과 릿지, 라쏘 모델을 훈련
- 릿지와 라쏘 모델의 규제 양을 조절하기 위한 최적의 alpha 값 찾기

## SECTION 3-3 마무리(1)

- 키워드로 끝나는 핵심 포인트

- **다중 회귀**는 여러 개의 특성을 사용하는 회귀 모델
  - 특성이 많으면 선형 모델은 강력한 성능을 발휘
- **특성 공학**은 주어진 특성을 조합하여 새로운 특성을 만드는 일련의 작업 과정
- **릿지**는 규제가 있는 선형 회귀 모델 중 하나이며 선형 모델의 계수를 작게 만들어 과대적합을 완화
  - 릿지는 비교적 효과가 좋아 널리 사용하는 규제 방법
- **라쏘**는 또 다른 규제가 있는 선형 회귀 모델
  - 릿지와 달리 계수 값을 아예 0으로 만들 수도 있음
- **하이퍼파라미터**는 머신러닝 알고리즘이 학습하지 않는 파라미터
  - 이런 파라미터는 사람이 사전에 지정해야 함
  - 대표적으로 릿지와 라쏘의 규제 강도  $\alpha$  파라미터

## SECTION 3-3 마무리(2)

- 핵심 패키지와 함수

- pandas

- read\_csv(): CSV 파일을 로컬 컴퓨터나 인터넷에서 읽어 판다스 데이터프레임으로 변환하는 함수
    - 자주 사용하는 매개변수
      - sep: CSV 파일의 구분자를 지정. 기본값은 'coma(,)'
      - header에 데이터프레임의 열 이름으로 사용할 CSV 파일의 행 번호를 지정. 기본적으로 첫 번째 행을 열 이름으로 사용
      - skiprows는 파일에서 읽기 전에 건너뛸 행의 개수를 지정
      - nrows는 파일에서 읽을 행의 개수를 지정

## SECTION 3-3 마무리(3)

- 핵심 패키지와 함수

- scikit-learn

- PolynomialFeatures는 주어진 특성을 조합하여 새로운 특성을 만듦
  - degree는 최고 차수를 지정. 기본값은 2
  - interaction\_only가 True이면 거듭제곱 항은 제외되고 특성 간의 곱셈 항만 추가됨. 기본값은 False
  - include\_bias가 False이면 절편을 위한 특성을 추가하지 않음. 기본값은 True
- Ridge는 규제가 있는 회귀 알고리즘인 린지 회귀 모델을 훈련
  - alpha 매개변수로 규제의 강도를 조절. alpha 값이 클수록 규제가 강해지며, 기본값은 1
  - solver 매개변수에 최적의 모델을 찾기 위한 방법을 지정할 수 있음. 기본값은 'auto'이며 데이터에 따라 자동으로 선택
  - 사이킷런 0.17 버전에서 추가된 'sag'는 확률적 평균 경사 하강법 알고리즘으로 특성과 샘플 수가 많을 때 성능이 빠르고 좋음
  - 사이킷런 0.19 버전에는 'sag'의 개선 버전인 'saga'가 추가
  - random\_state는 solver가 'sag'나 'saga'일 때 넘파이 난수 시드값을 지정할 수 있음
- Lasso는 규제가 있는 회귀 알고리즘인 라쏘 회귀 모델을 훈련
  - 이 클래스는 최적의 모델을 찾기 위해 좌표축을 따라 최적화를 수행해가는 좌표 하강법 coordinate descent을 사용
  - alpha와 random\_state 매개변수는 Ridge 클래스와 동일
  - max\_iter는 알고리즘의 수행 반복 횟수를 지정. 기본값은 1000

## SECTION 3-3 확인 문제

1. a, b, c 특성으로 이루어진 훈련 세트를 PolynomialFeatures(degree=3)으로 변환했다. 다음 중 이 변환된 데이터에 포함되지 않는 특성은 무엇인가?

① 1

② a

③  $a * b$

④  $a * b^3$

2. 다음 중 특성을 표준화하는 사이킷런 변환기 클래스는 무엇인가?

① Ridge

② Lasso

③ StandardScaler

④ LinearRegression

## SECTION 3-3 확인 문제

3. 다음 중 과대적합과 과소적합을 올바르게 표현하지 못한 것은 무엇인가?

- ① 과대적합인 모델은 훈련 세트의 점수가 높음
- ② 과대적합인 모델은 테스트 세트의 점수도 높음
- ③ 과소적합인 모델은 훈련 세트의 점수가 낮음
- ④ 과소적합인 모델은 테스트 세트의 점수도 낮음

4. 다음 중 훈련하기 전에 특성을 표준화해야 하는 모델 클래스를 모두 고르세요.

- ① KNeighborsClassifier
- ② KNeighborsRegressor
- ③ Ridge
- ④ Lasso