

2024년

LG전자 에어솔루션사업부 - KNU

# AI / Big Data 하계 워크샵

## 지도학습 및 비지도학습

안상태  
경북대학교 전자공학부



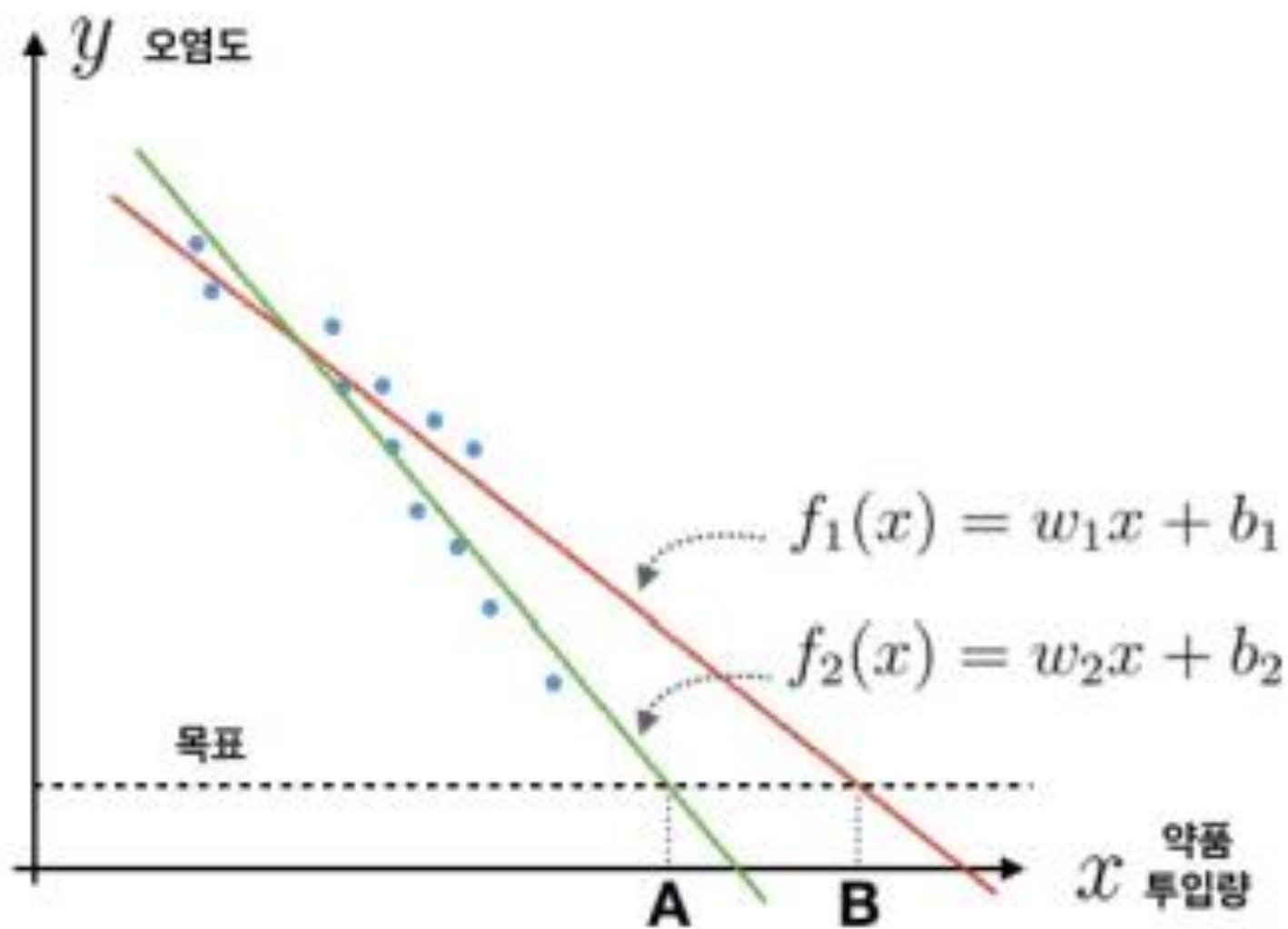
# 지도학습

## 1. 선형회귀와 다항회귀

# 1.1 회귀 모델

- 회귀 regression : 회귀 분석은 대표적인 지도학습 알고리즘
  - 관측된 데이터를 통해 독립변수와 종속변수 사이의 숨어있는 관계를 추정하는 것
- 선형 회귀 linear regression 는  $y = f(x)$ 에서 입력  $x$ 에 대응되는 실수  $y$ 들이 주어지고 추정치  $f(x)$ 가 가진 오차를 측정
  - 이 오차를 줄이는 방향으로 함수의 계수를 최적화하는 일

## 1.1 회귀 모델

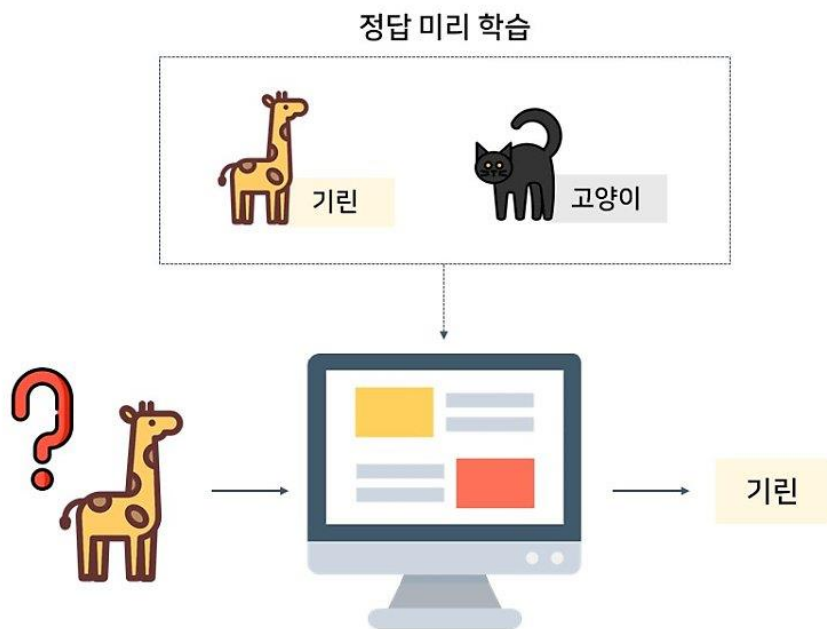


# 1.1 회귀 모델

- 데이터에 숨겨진 관계를 표현하고, 약품 투입량과 같은 독립변수에 대해 오염도라는 종속 변수가 어떤 값을 가질지 예측하는  $f_a(x)$ 와  $f_b(x)$ 를 가설hypothesis이라고 부름
- 좋은 가설은 오차error가 작은 가설
  - 회귀 분석은 데이터를 설명하는 좋은 가설을 찾는 것

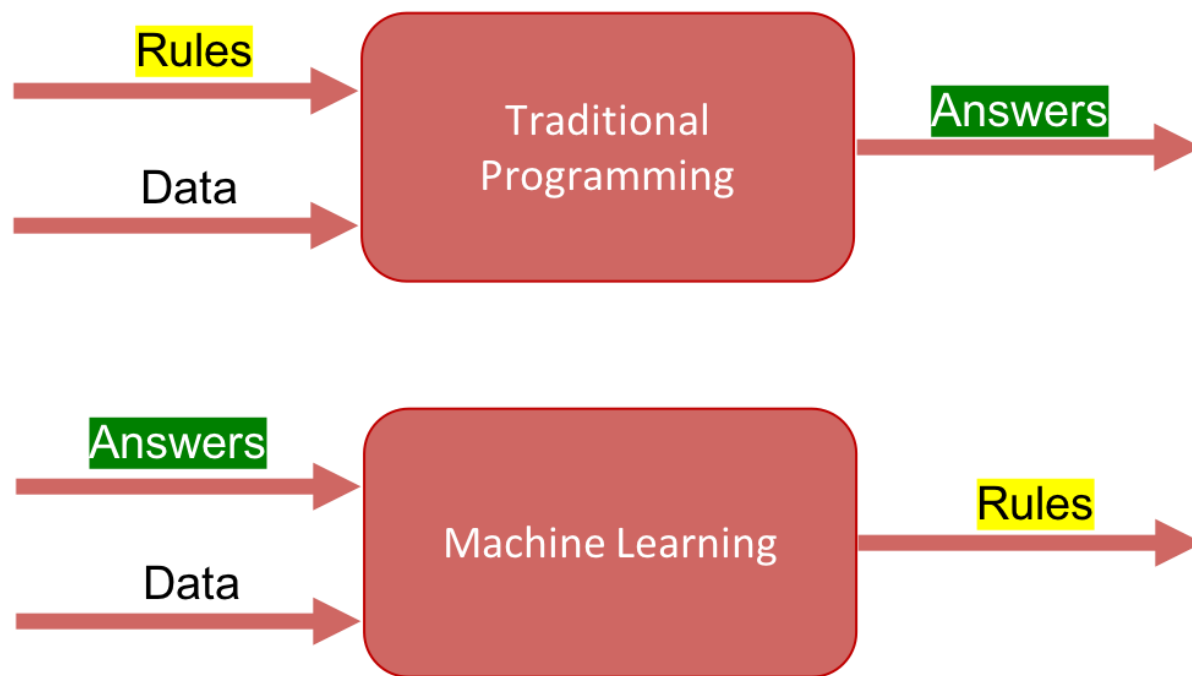
## 1.2 선형 회귀와 지도 학습

- 데이터에 제시된 목표값을 정답값 혹은 레이블<sup>label</sup>이라고 부름
  - 지도 학습은 주어진 입력-출력 쌍을 학습한 후에 새로운 입력값이 들어왔을 때, 합리적인 출력값을 예측하는 것.



## 1.2 선형 회귀와 지도학습

- 전통적 프로그래밍 : 사람이 이 함수  $f(x)$ 를 고안하여 구현한 뒤, 입력  $x$ 를 넣어 답  $y$ 를 얻는 것
- 머신러닝: 데이터  $(x, y)$ 를 주면 함수  $f(x)$ 를 만들어내는 일



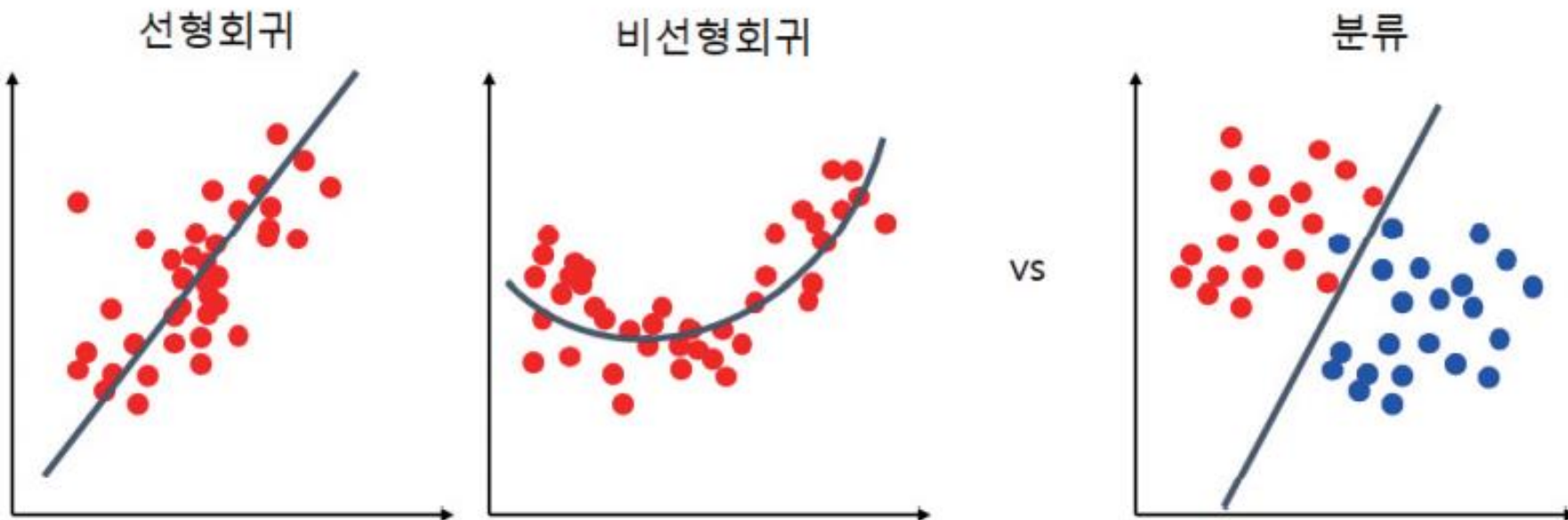
## 1.2 선형 회귀와 지도 학습

- 지도 학습 알고리즘의 대표적인 두 유형은 회귀 분석과 분류classification
  - 회귀는 입력 데이터 하나 하나에 대응하는 출력값을 예측
  - 분류는 입력 데이터를 몇 가지 이산적인 범주category 중의 하나로 대응



## 1.2 선형 회귀와 지도학습

- 회귀 분석 : 데이터를 설명하는 직선을 찾는 **선형**linear 회귀와 곡선을 찾는 **비선형**nonlinear 회귀
  - 이진 분류**binary classification는 데이터를 양분하는 경계 직선 혹은 곡선을 찾는 것.



## 1.2 선형 회귀와 지도학습

- 데이터를 학습시킬 때, 데이터에서 중요한 일부 정보만을 추출하여 이것으로 학습시키고 테스트할 수도 있음
  - **특징**<sup>feature</sup> : 특징이란 관찰되는 현상에서 측정할 수 있는 개별적인 **속성**<sup>attribute</sup>을 의미
  - $y = f(x)$ 의 함수를 찾는다고 할 때, 입력 데이터로 사용되는  $x$ 가 특징
- 기계학습에서 다룰 수 있는 Feature의 예
  - **사람의 키와 몸무게**
  - **개의 몸통 길이와 높이**
  - **주택 가격에 영향을 주는 주택의 특징들**

## 1.3 실제 데이터를 읽고 가설 설정

- Pandas를 이용하여 CSV 파일을 읽어 lin\_data라는 데이터프레임을 만듦
- 투입량에 따른 오염도 측정 결과 100건이 담겨 있음
  - 데이터프레임의 input 열은 오염도를 줄이기 위한 약품의 투입량
  - pollution 열은 실제 측정된 오염도 수치



```
import matplotlib.pyplot as plt
import pandas as pd
# 데이터 저장 위치
data_home = 'https://github.com/dknife/ML/raw/main/data/'
lin_data = pd.read_csv(data_home+'pollution.csv') # 데이터 파일 이름
print(lin_data)
```

	input	pollution
0	0.240557	4.858750
1	0.159731	4.471091
..	...	...
99	0.290294	3.169049

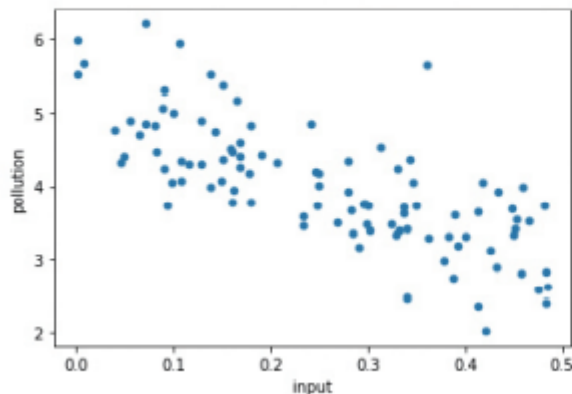
[100 rows x 2 columns]

## 1.3 실제 데이터를 읽고 가설 설정

- 데이터를 시각적으로 확인하기 위하여 plot() 메소드를 사용.
  - x축으로는 input 열을 사용
  - y축으로 pollution 열을 사용
  - 투입량을 늘이면 오염도가 줄어드는 경향이 있는 것을 확인할 수 있음



```
lin_data.plot(kind = 'scatter', x = 'input', y = 'pollution')
```

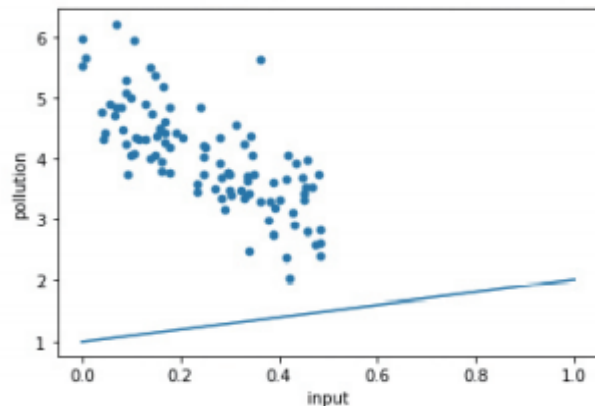


- input을 독립변수  $x$ 로, pollution을 종속변수  $y$ 로 하는  $y = wx + b$ 라는 직선으로 표현하면, 데이터가 이 함수를 따를 것이라는 **가설** hypothesis을 제시
  - pyplot의 plot() 함수는 독립변수의 리스트와 종속변수의 리스트를 주면, 이들을 연결한 선을 그려낸다.



```
w, b = 1, 1
x0, x1 = 0.0, 1.0
def h(x, w, b):          # 가설에 따라 값을 계산하는 함수
    return w*x + b

# 데이터(산포도)와 가설(직선)을 비교
lin_data.plot(kind = 'scatter', x = 'input', y = 'pollution')
plt.plot([x0, x1], [h(x0, w, b), h(x1, w, b)])
```



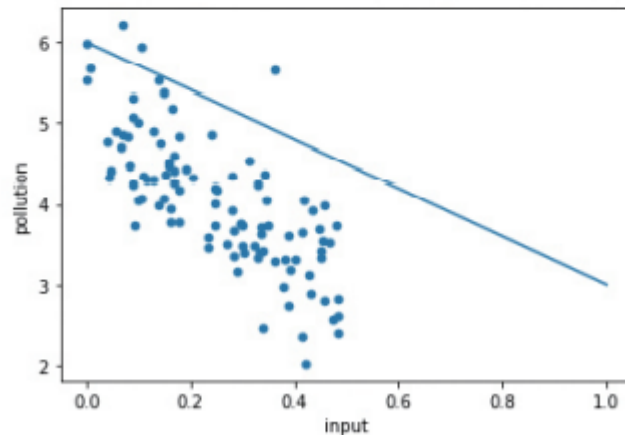
- 데이터와 일치시키기 위해서는 음수 기울기가 필요할 것
  - 최적의  $w$  와  $b$  를 찾을 수 있는 방법은?



```
w, b = -3, 6  
x0, x1 = 0.0, 1.0
```

*# 새로운 파라미터로 가설(직선)과 데이터(산포도) 비교*

```
lin_data.plot(kind = 'scatter', x = 'input', y = 'pollution')  
plt.plot([x0, x1], [h(x0, w, b), h(x1, w, b)])
```



## 1.4 좋은 가설과 모델의 오차

- 데이터를 추정하는 가설이 얼마나 정확한지를 평가하는 방법
  - 가설이 훌륭한 모델이라면 데이터는 가설이 나타내는 직선 위에 모두 놓이게 될 것
  - 좋은 가설이라면 데이터가 이 직선들 근처에 있을 것

## 1.4 좋은 가설과 모델의 오차

- 대표적인 오차 척도는 평균 제곱 오차
- 예측치  $\hat{y}$ 와 정답 레이블  $y$ 사이의 차이를 제곱하여 모두 더한 뒤에 전체 데이터의 개수  $m$ 으로 나누는 것
- 평균 제곱 오차 mean squared error:MSE라고 하며, 다음과 같은 식

$$E_{mse} = \frac{1}{m} \sum_{i=1}^m (\hat{y}_i - y_i)^2$$



## 1.4 좋은 가설과 모델의 오차

- 예측 결과가  $y_{\text{hat}}$ , 정답 레이블이  $y$ 에 저장되어 있다고 할 때, 평균 제곱 오차는 아래와 같이 구할 수 있음

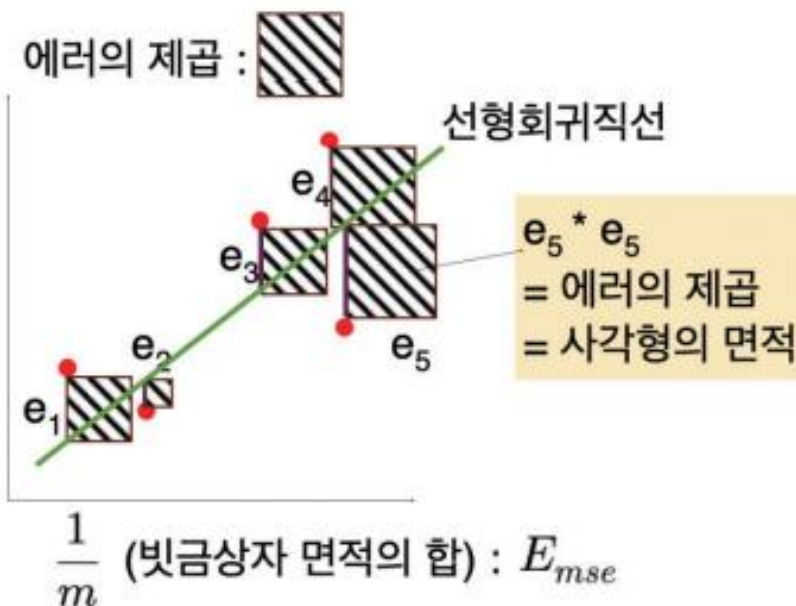
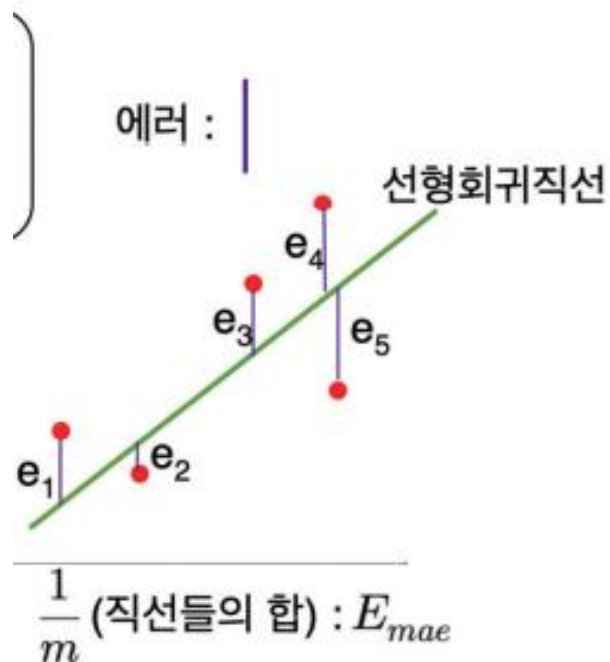


```
import numpy as np
y_hat = np.array([1.2, 2.1, 2.9, 4.1, 4.7, 6.3, 7.1, 7.7, 8.5, 10.1])
y = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
diff_square = (y_hat - y)**2
e_mse = diff_square.sum() / len(y)
e_mse
```

```
0.060999999999999996
```

## 1.4 좋은 가설과 모델의 오차

- 오차를 제공하는 이유
  - 빨간색 점으로 표시된 레이블과 가설 사이에 차이가 있음
  - $e_1$ 에서  $e_5$ 까지 전체 에러의 합이 최소가 되는 모델이 바람직한 모델
  - 오차합 곡면의 기울기를 따라 내려가 최소 오차에 접근하기 위하여



## 1.4 좋은 가설과 모델의 오차

- 평균 제곱 오차는 머신러닝에서 가장 흔히 사용되는 오차 척도
- 머신러닝의 대표적인 패키지 중 하나인 scikit-learn 역시 이러한 오차 계산을 지원한다 : `mean_squared_error()` 함수



```
from sklearn.metrics import mean_squared_error  
print('Mean squared error:', mean_squared_error(y_hat, y))
```

```
Mean squared error: 0.060999999999999996
```

## 1.4 좋은 가설과 모델의 오차

- 제곱을 하지 않고 오차를 더하고 싶다면 **평균 절대 오차** mean absolute error: MAE 라는 것을 사용할 수 있음
  - 오차를 제곱하지 않고 절대값을 취해 더하는 것

$$E_{mae} = \frac{1}{m} \sum_{i=1}^m |\hat{y}_i - y_i|$$

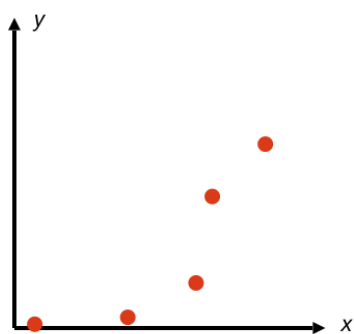


```
from sklearn.metrics import mean_absolute_error  
print('Mean absolute error:', mean_absolute_error(y_hat, y))
```

```
Mean absolute error: 0.209999999999999988
```

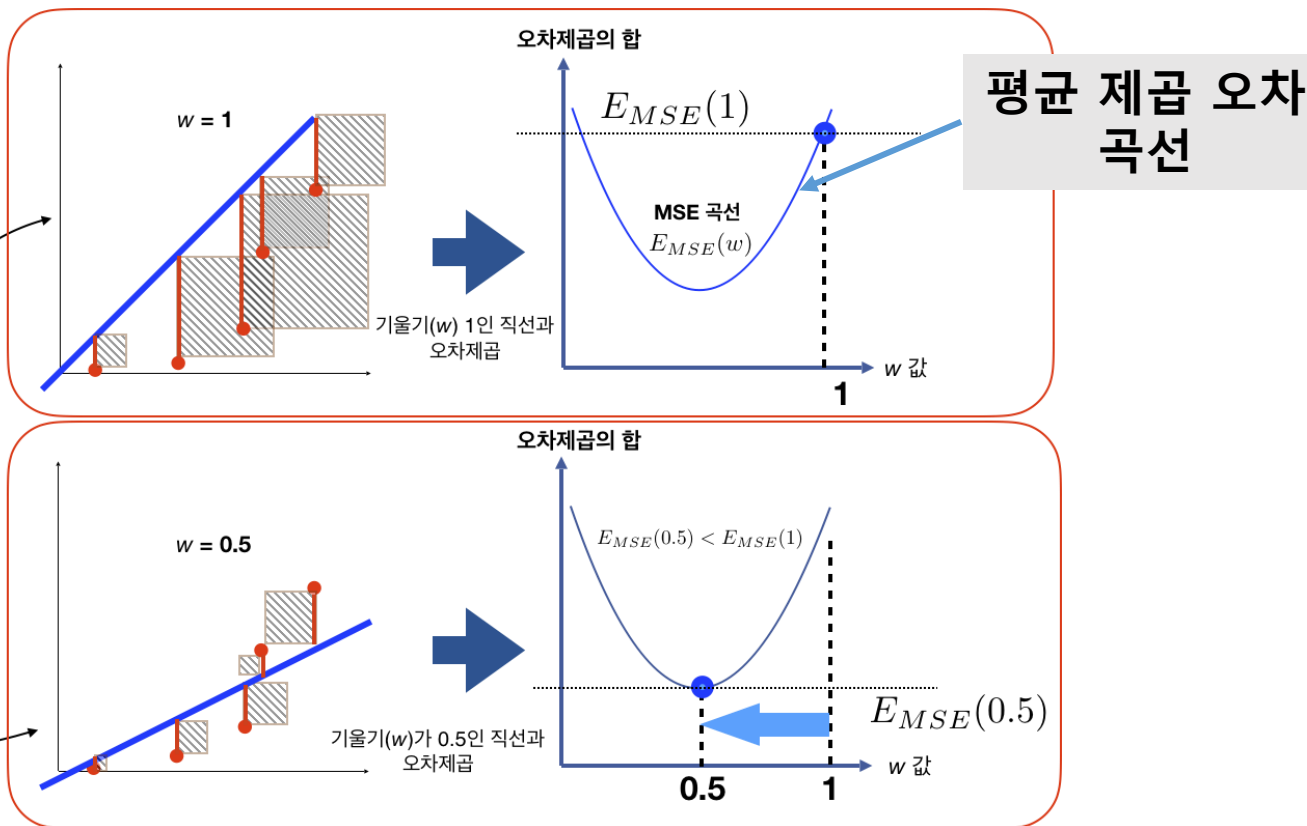
# 1.5 선형 회귀 함수의 시각적 이해

- 5 개의 점이 분포하고 있는데 이 분포를 잘 설명하는  $y = f(x)$ 의 함수를 찾는 것이 바로 선형 회귀의 목적
  - $y = wx$ 와 같이  $y$ 절편이 0인 경우를 가정하고 최적의  $w$ 값을 찾는 과정



위의 다섯개 점들의 분포를 가장 잘 설명하는 1차 함수를 만들자.

가능한 여러 가설 중에서  $w=1$ 인 경우와  $w=0.5$ 인 경우만 살펴 보자



## 1.5 선형 회귀 함수의 시각적 이해

- 오차 제곱의 합  $E_{mse}^{0.5}$ 는  $E_{mse}^1$ 와 비교하여 작은 값
- 기울기 0.5가 이 점들의 분포를 설명하는 함수의 최적 기울기 값이라고 한다면, 오른쪽 하단과 같은 오목한 곡선의 가장 낮은 지점이 될 것
- 이러한 정보를 바탕으로 다음과 같은 코드를 작성해 보도록 하자
- 우선 실제 값을 가지는 5개의  $x, y$  데이터를 생성하고  $w = 1.0$ 으로 하는  $\hat{y}$ 에서  $w = 0.3$ 인  $\hat{y}$ 까지 0.1씩 값을 감소시켜가며  $w$ 와 평균제곱오차를 출력해 보도록 하자



```
import numpy as np
from sklearn.metrics import mean_squared_error as mse

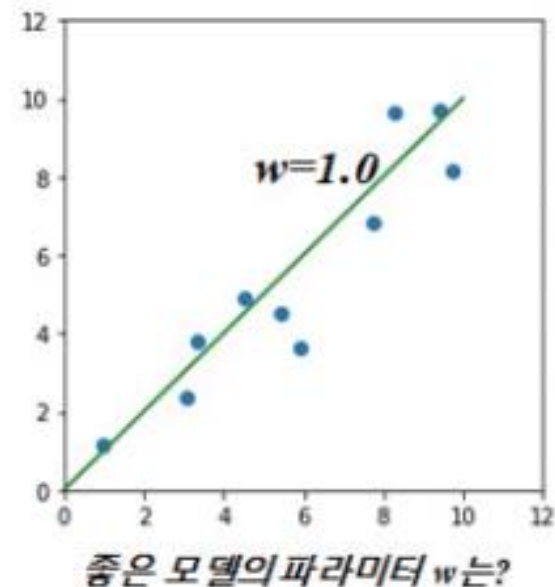
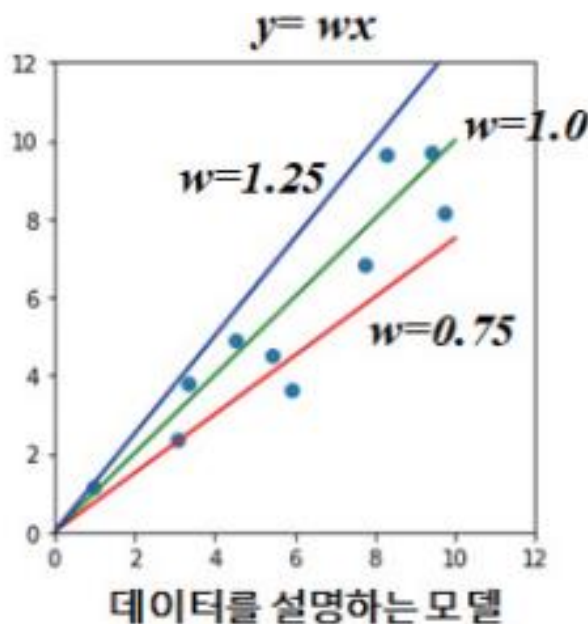
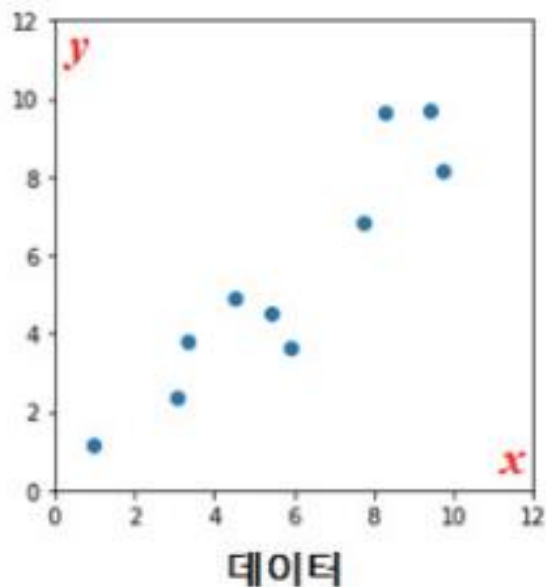
# 5개 점의 x, y 좌표값
x = np.array([1, 4.5, 9, 10, 13])
y = np.array([0, 0.2, 2.5, 5.4, 7.3])

w_list = np.arange(1.0, 0.2, -0.1)
for w in list(w_list):      # w를 바꾸어가며 예측치와 정답의 오차 비교
    y_hat = w * x
    print('w = {:.1f}, 평균제곱 오차: {:.2f}'.format(w, mse(y_hat, y)))
```

```
w = 1.0, 평균제곱 오차: 23.08
w = 0.9, 평균제곱 오차: 15.86
w = 0.8, 평균제곱 오차: 10.13
w = 0.7, 평균제곱 오차: 5.89
w = 0.6, 평균제곱 오차: 3.13
w = 0.5, 평균제곱 오차: 1.85
w = 0.4, 평균제곱 오차: 2.06
w = 0.3, 평균제곱 오차: 3.75
```

## 1.6 오차의 종류에 따른 오차 곡면의 모습

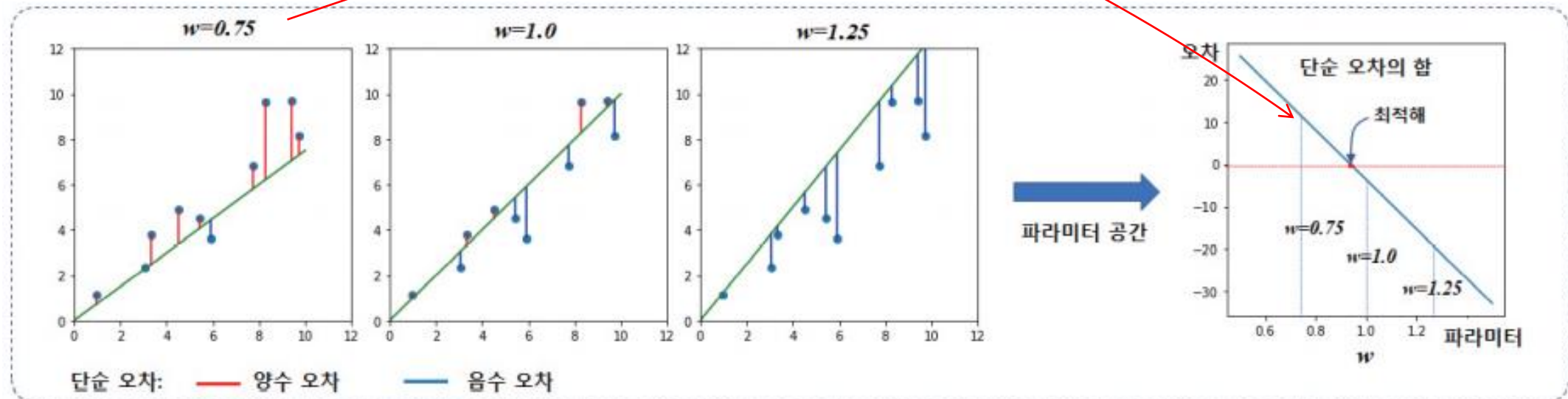
- 모델의 오차를 계산하는 방법에 따른 오차 곡면의 모습을 살펴보고, 이 오차 곡면을 이용하여 최적의 모델을 찾는 방법에 대해 살펴보도록 하자
- 목표: 데이터  $x$ 와  $y$ 의 관계를 가장 잘 설명하는 모델  $y = wx$ 를 찾는 것





## 1.6 오차의 종류에 따른 오차 곡면의 모습

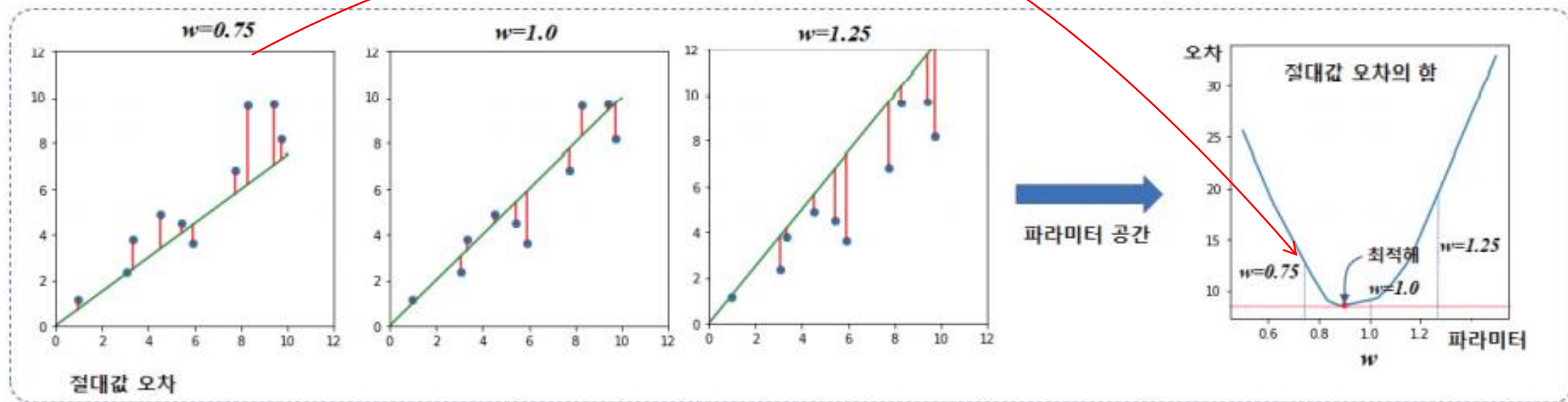
- 예측과 정답의 차이로 계산하는 단순한 오차를 살펴 보자.
- 그림의 왼쪽에는 파라미터가 0.75, 1.0, 1.25일 때 모델과 데이터의 오차를 보이고 있다.



# 1.6 오차의 종류에 따른 오차 곡면의 모습

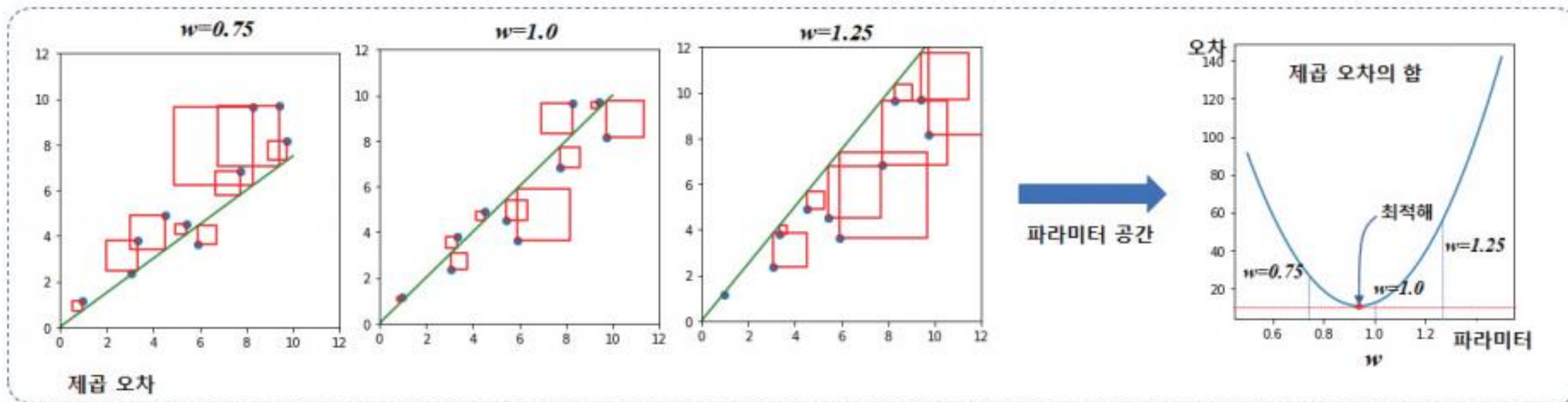
- 평균 절대 오차<sup>MAE</sup>
- 교차지점을 구할 필요없이 오차 곡선의 극소만 구하면 된다.

w가 0.75일 때 오차의 합은 14근처의 값



# 1.6 오차의 종류에 따른 오차 곡면의 모습

- 정답과 예측의 차이를 제공하는 평균 제곱 오차<sup>MSE</sup>



## 1.7 오차로 가설을 평가하고 좋은 가설 찾기

- 여러가지 가설이 존재할 경우 가설이 추정한 종속변수와 실제 데이터의 종속변수의 차이가 적을수록 좋은 것
- 가설에 의한 추정치 값과 실제 값의 차이를 오차<sup>error</sup>
  - 추정치는 일반적으로 종속변수  $y$ 의 위에  $\wedge$ 기호를 씌운  $\hat{y}$ ('^'기호는 hat으로 읽는다)로 표기
  - 오차  $E$ 는 다음과 같은 식으로 계산할 수 있다.

$$E = \hat{y} - y = wx + b - y$$

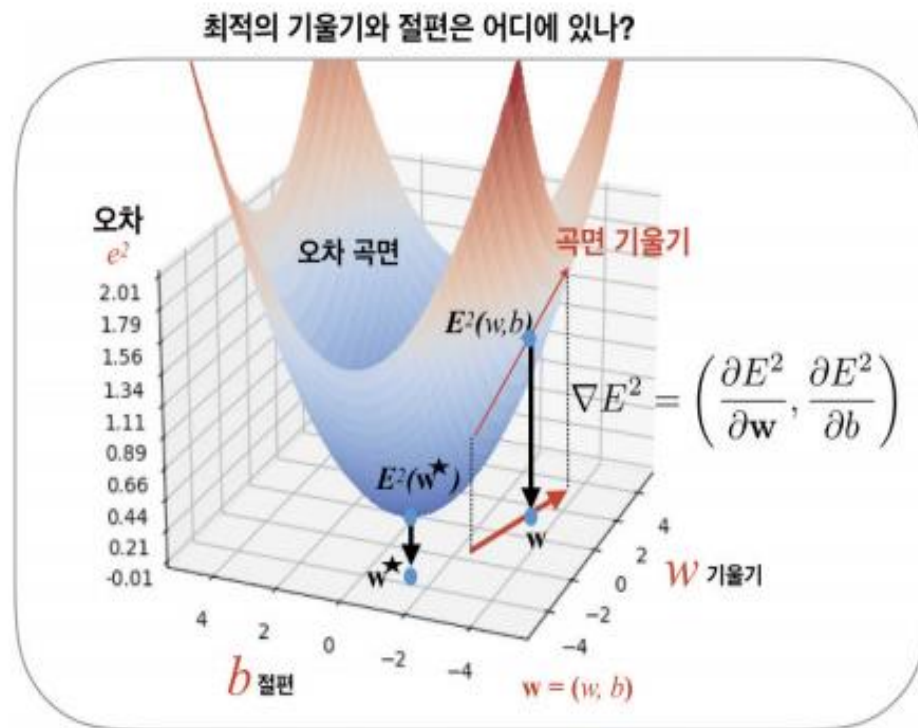
## 1.7 오차로 가설을 평가하고 좋은 가설 찾기

- **최소 제곱법** least squares approximation은 오차를 제공하여 오차 곡면의 기울기를 따라 내려가 기울기가 0인 극소 지점을 찾는 것
- 오차의 제곱  $E^2(w, b)$ 이 그림과 같은 곡면이라면, 최적의  $w$ 와  $b$ 를 찾기 위한 오차 곡면의 기울기 방향은 다음과 같이 구할 수 있음

$$\nabla E^2 = \left( \frac{\partial E^2}{\partial w}, \frac{\partial E^2}{\partial b} \right)$$

$$\frac{\partial E^2}{\partial w} = \frac{\partial (wx + b - y)^2}{\partial w} = 2(wx + b - y)x = 2Ex$$

$$\frac{\partial E^2}{\partial b} = \frac{\partial (wx + b - y)^2}{\partial b} = 2(wx + b - y) \cdot 1 = 2E$$



## 1.7 오차로 가설을 평가하고 좋은 가설 찾기

- $(w, b)$  벡터를  $-(E_x, E)$  방향으로 옮겨주면 최적의  $w$ 와  $b$ 에 가까워질 것
- 아래 코드는 "조금"의 정도를 learning rate(학습률)라는 하이퍼파라미터로 제어하고 있다.  $n$ 개의 데이터  $x_i$ 에 대한 예측 오차가  $E_i$ 라고 할 때 다음과 같이 기울기  $w$ 와 절편  $b$ 를 수정하는 것

$$w \leftarrow w - \eta \sum_{i=1}^n E_i x_i, \quad b \leftarrow b - \eta \sum_{i=1}^n E_i$$

- 벡터화 연산을 사용하여 아래와 같이 직선의 기울기와 절편을 갱신



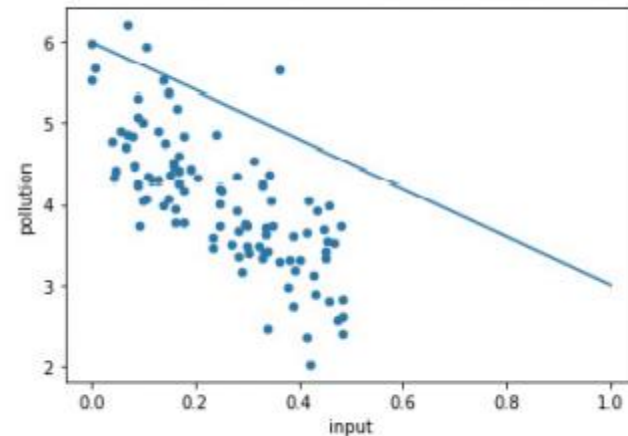
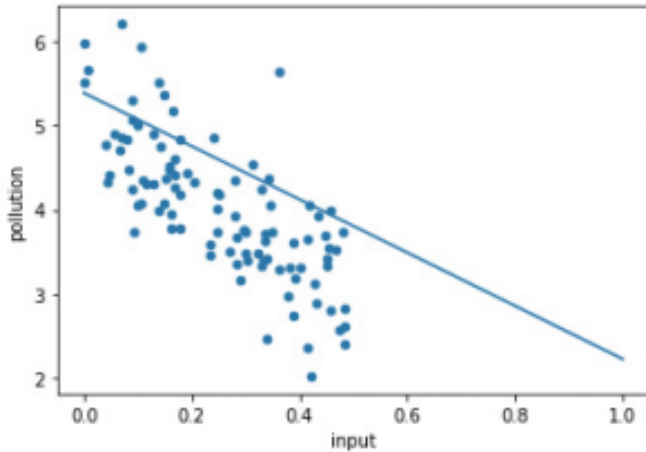
```
learning_rate = 0.005
w = w - learning_rate * (error * x).sum()
b = b - learning_rate * error.sum()
```

## 1.7 오차로 가설을 평가하고 좋은 가설 찾기

- 수정된 기울기  $w$ 와 절편  $b$ 를 이용하여 직선을 그려보자.
- 직선이 데이터를 잘 표현하도록 옮겨진 것을 확인할 수 있을 것

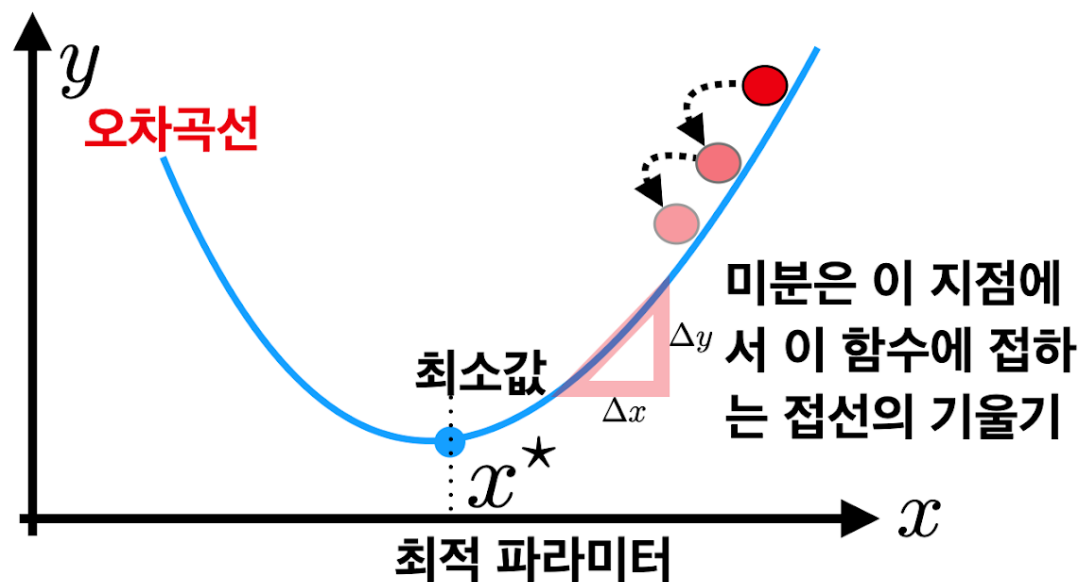


```
lin_data.plot(kind = 'scatter', x = 'input', y = 'pollution')  
plt.plot([x0, x1], [h(x0, w, b), h(x1, w, b)])
```



## 1.8 경사하강법

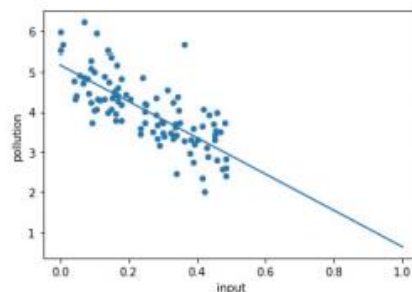
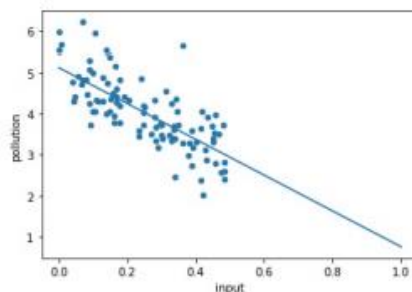
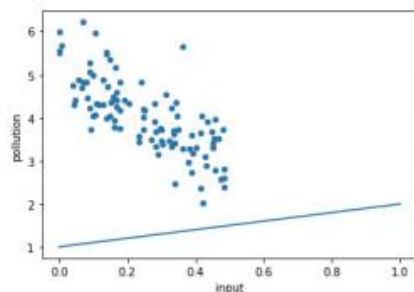
- 선형 회귀의 모델은 선형 방정식이고, 동작을 결정하는 파라미터는 직선의 기울기  $w$ 와 절편  $b$
- 벡터로 표현하면  $(w, b)$ 가 파라미터 벡터
- 학습 과정은 오차를 줄이도록 오차 곡면의 경사를 따라 내려가는 최적화 과정 : 경사 하강법(gradient descent)







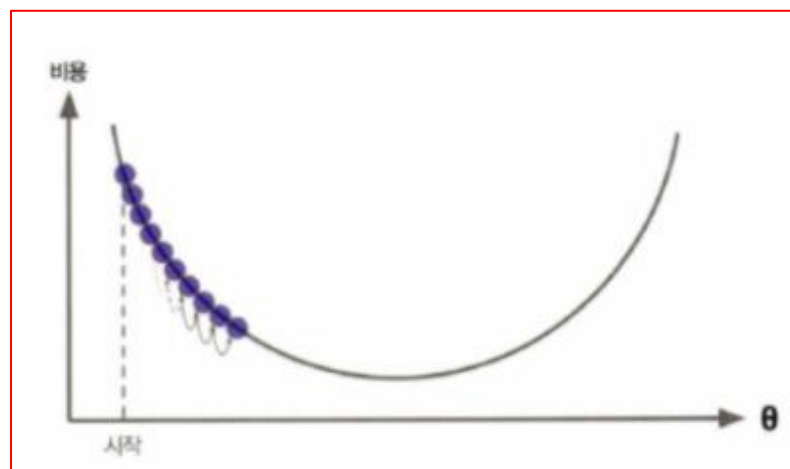
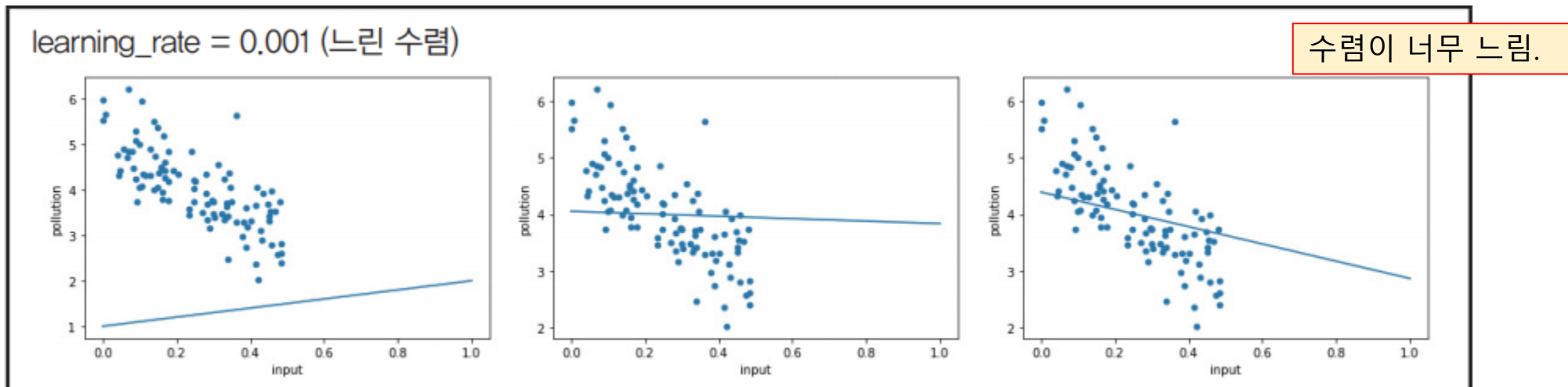
```
def h(x, param):  
    return param[0]*x + param[1]  
  
learning_iteration = 1000 # 하이퍼파라미터 : 학습반복 횟수  
learning_rate = 0.0025 # 하이퍼파라미터 : 학습율로 0.05, 0.001등이 가능  
  
param = [1, 1] # w, b를 하나의 변수로 함  
  
x = lin_data['input'].to_numpy()  
y = lin_data['pollution'].to_numpy()  
  
for i in range(learning_iteration):  
    if i % 200 == 0:  
        lin_data.plot(kind = 'scatter', x = 'input', y = 'pollution')  
  
        plt.plot([0, 1], [h(0, param), h(1, param)])  
        error = ( h(x, param) - y)  
        param[0] -= learning_rate * (error * x).sum()  
        param[1] -= learning_rate * error.sum()
```



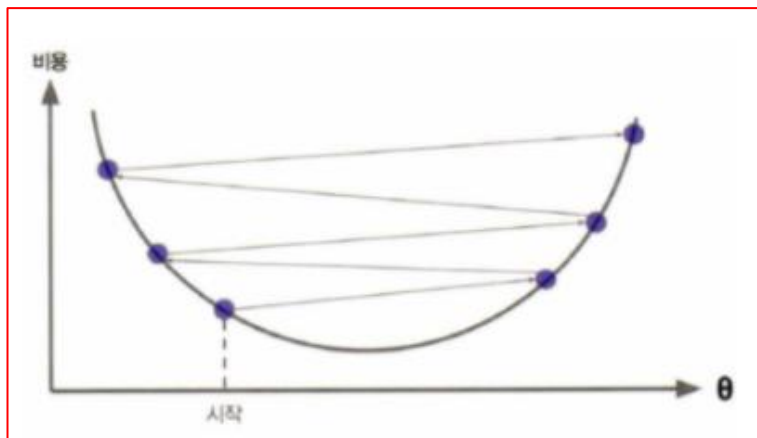
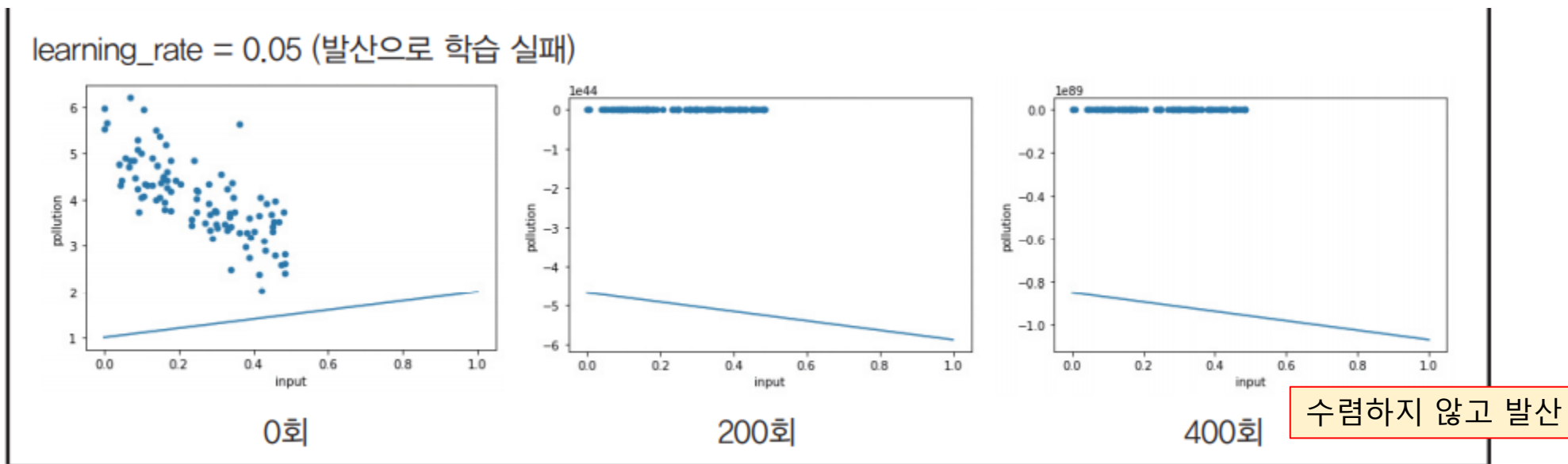
...

오차가 점점 줄어들면 회귀 직선이 데이터를 점점 더 정확하게 모델링

- learning\_iteration과 learning\_rate가 변경되면 학습으로 얻는 모델의 파라미터가 달라질 것
  - 학습을 제어하는 변수 : 하이퍼파라미터 hyperparameter 라고 함



- learning\_iteration과 learning\_rate가 변경되면 학습으로 얻는 모델의 파라미터가 달라질 것
  - 학습을 제어하는 변수 : 하이퍼파라미터 hyperparameter 라고 함



## 1.9 scikit-learn을 이용한 선형 회귀

- scikit-learn 라이브러리를 활용하여 선형 회귀를 구현
  - 사이킷런<sup>scikit-learn</sup>: 선형 회귀, k-NN 알고리즘, 서포트 벡터머신, k-means 등 다양한 머신러닝 알고리즘을 쉽게 구현할 수 있게 해줌




```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn import linear_model  # scikit-learn 모듈을 가져온다

data_home = 'https://github.com/dknife/ML/raw/main/data/'
lin_data = pd.read_csv(data_home+'pollution.csv')
```

# 1.9 scikit-learn을 이용한 선형 회귀

- <https://scikit-learn.org/stable/>

 Install User Guide API Examples Community More ▾

# scikit-learn

Machine Learning in Python

Getting Started Release Highlights for 1.2 GitHub

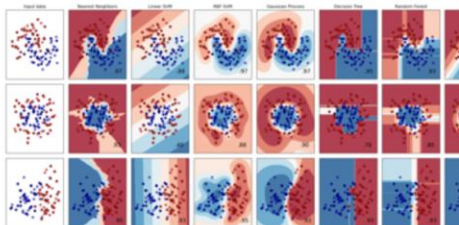
- Simple and efficient tools for predictive data analysis
- Accessible to everybody, and reusable in various contexts
- Built on NumPy, SciPy, and matplotlib
- Open source, commercially usable - BSD license

## Classification

Identifying which category an object belongs to.

**Applications:** Spam detection, image recognition.

**Algorithms:** SVM, nearest neighbors, random forest, and more...

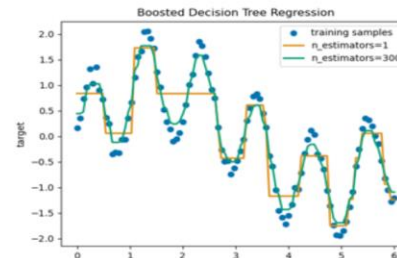


## Regression

Predicting a continuous-valued attribute associated with an object.

**Applications:** Drug response, Stock prices.

**Algorithms:** SVR, nearest neighbors, random forest, and more...

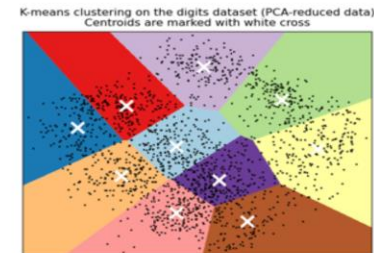


## Clustering

Automatic grouping of similar objects into sets.

**Applications:** Customer segmentation, Grouping experiment outcomes

**Algorithms:** k-Means, spectral clustering, mean-shift, and more...



## 1.9 scikit-learn을 이용한 선형 회귀

- scikit-learn의 선형 회귀 모델의 입력 데이터는 2차원 배열로, 각 행이 데이터 인스턴스이며, 각 데이터 인스턴스가 여러 개의 특징값을 가질 수 있음
- 현재 우리는 하나의 특징값만을 사용하지만 이 경우에도 하나의 원소를 가진 벡터로 제공해야 함.



```
x = lin_data['input'].to_numpy()
y = lin_data['pollution'].to_numpy()
x = x[:, np.newaxis]      # 선형 회귀 모델의 입력형식에 맞게 차원을 증가시킴
print(x)
```

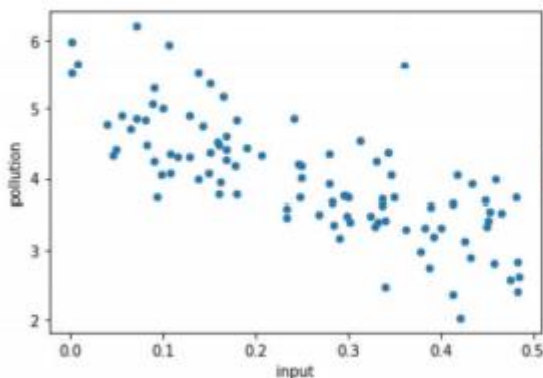
```
[[0.24055707]
 [0.1597306 ]
 ...
 [0.00720486]
 [0.29029368]]
```

# 1.9 scikit-learn을 이용한 선형 회귀

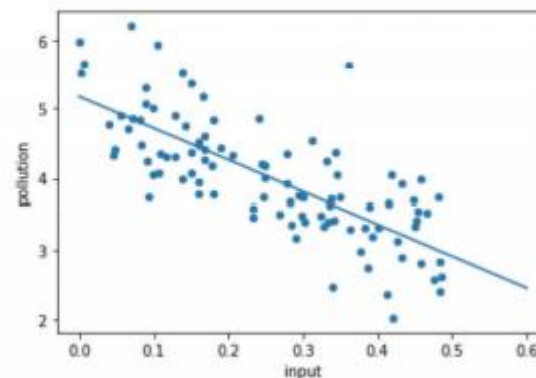
- 선형회귀 모델과 fit() 메소드의 역할?



```
regr = linear_model.LinearRegression()  
regr.fit(x, y) # 선형 회귀 모델에 데이터를 넣어 학습을 진행함
```



데이터를 기반으로 최적의  
선형회귀 모델 생성



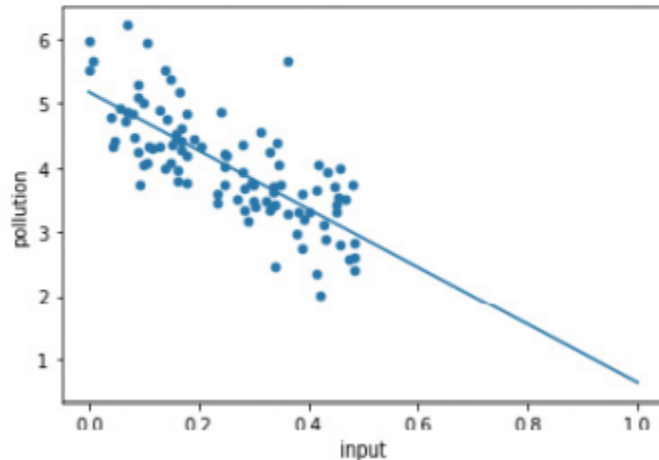
이전의 수렴 과정을 모  
두 자동화하는 메소드가  
바로 fit()

## 1.9 scikit-learn을 이용한 선형 회귀

- 입력으로 0과 1을 주고 이에 해당하는 출력값을 예측하도록 하면 됨



```
lin_data.plot(kind = 'scatter', x = 'input', y = 'pollution')  
y_pred = regr.predict([[0], [1]])  
plt.plot([0, 1], y_pred) # x 구간을 0에서 1 사이로 두자
```





## 1.10 다항 회귀

- 선형회귀는 입력과 출력이 선형관계를 갖는 것으로 가정
- 실제 데이터는 아래와 같이 비선형 방정식을 따를 수도 있으며 이것을 **다항 회귀** polynomial regression라고 함

$$f(x) = y = \theta_2 x^2 + \theta_1 x + \theta_0$$

## 1.10 다항 회귀

- 다변량<sup>multivariate</sup> 회귀를 할 때 다항<sup>polynomial</sup> 회귀를 이용하면 두 개의 특징  $x_1, x_2$ 를 가진 2차 다항식은 아래와 같은 함수가 될 것

$$f(x) = y = \theta_5 x_1^2 + \theta_4 x_2^2 + \theta_3 x_1 x_2 + \theta_2 x_1 + \theta_1 x_2 + \theta_0$$

- 위 함수는  $(x_1 + x_2 + 1)^2$ 로 얻어지는 다항식의 계수를 변경하는 것과 같음

## 1.10 다항 회귀

- $n$ 개의 특징을 가진  $X = (x_1, x_2, \dots, x_n)$ 를 입력으로 하는  $d$ 차 다항식 다변량 회귀라는 것은 입력을 그대로 사용하지 않고  $(x_1 + x_2 + \dots + x_n + 1)^d$ 에서 나타나는 항들을 사용하는 것
  - 데이터 정제로 간주할 수 있는데 scikit-learn의 preprocessing 서브 모듈에 있는 PolynomialFeatures 클래스를 사용함
  - 클래스를 생성할 때 차수를 지정하고, 입력을 `fit_transform()` 메소드에 넘기면 다항 회귀에 필요한 형태로 변환하며 이것을 **다항 특징 변환** *polynomial feature transformation*이라고 함



```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

data_loc = 'https://github.com/dknife/ML/raw/main/data/'
life = pd.read_csv(data_loc + 'life_expectancy.csv')
```

```
life.head()

life = life[['Life expectancy', 'Alcohol', 'Percentage expenditure',
             'Polio', 'BMI', 'GDP', 'Thinness 1-19 years']]
life.dropna(inplace = True)
X = life[['Alcohol', 'Percentage expenditure', 'Polio',
          'BMI', 'GDP', 'Thinness 1-19 years']]
y = life['Life expectancy']
```

- preprocessing 서브 모듈의 PolynomialFeatures 클래스를 활용해 입력 데이터들을 다항 회귀에 사용



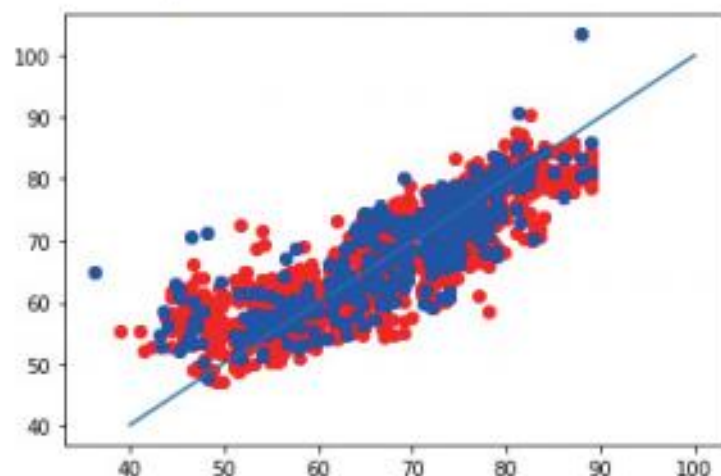
```
from sklearn.preprocessing import PolynomialFeatures  
poly_feature = PolynomialFeatures(degree = 3)  
X = poly_feature.fit_transform(X)
```



```
X_train,X_test,y_train,y_test = train_test_split(X, y, test_size = 0.2)
lin_model = LinearRegression()
lin_model.fit(X_train, y_train)

y_hat_train = lin_model.predict(X_train)
y_hat_test = lin_model.predict(X_test)
plt.scatter(y_train, y_hat_train, color='r')
plt.scatter(y_test, y_hat_test, color='b')
plt.plot([40, 100], [40, 100])
print('Mean squared error:', mean_squared_error(y_test, y_hat_test))
```

Mean squared error: 26.166291726402985

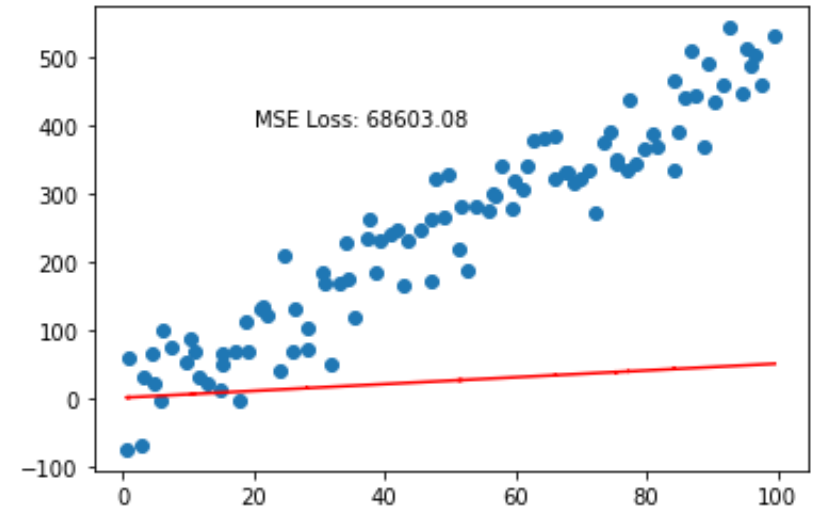


실습

# 실습 1-1

1) 주어진 데이터(linear.csv)를 이용하여 아래 함수들을 정의하고 MSE 를 구하세요.

- $y = 5x + 50 \cdot N(0,1)$  로 y 데이터 생성
- $E_{MSE} = \frac{1}{N} \sum_i^N (y_i - \hat{y}_i)^2$  로 MSE 함수 정의
- w와 b의 초기값은 0.5
- 위의 w,b를 사용하는  $y = wx + b$ 라는 함수를 정의  
wx+b와 x,y를 다음과 같이 plot해보고, MSE 출력

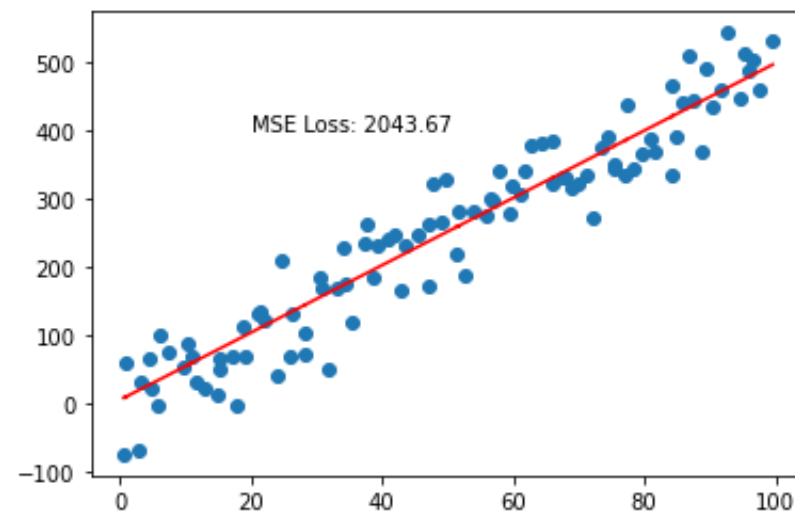
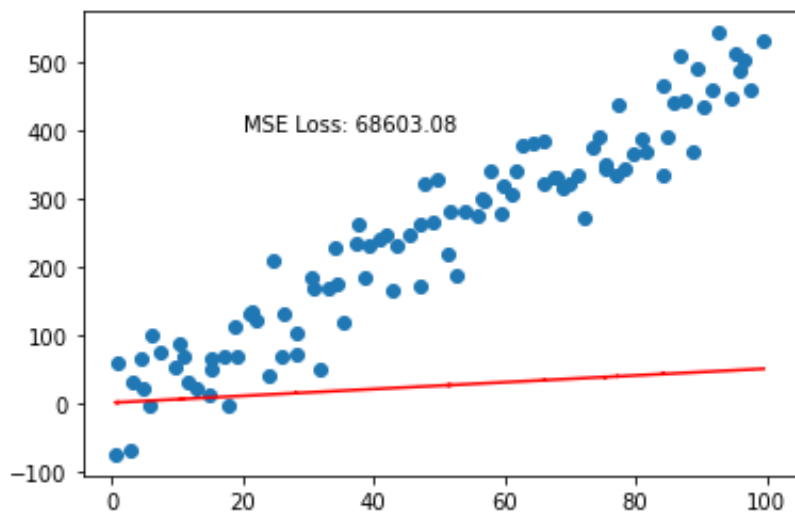




# 실습 1-1

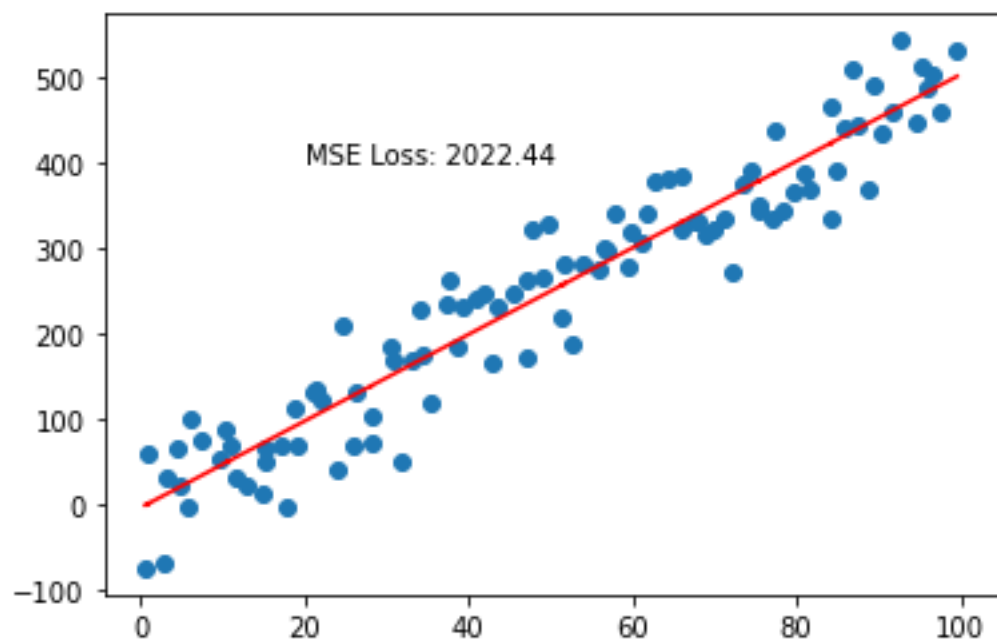
2) 앞의 식을 Gradient Descent 방법을 통해 주어진 데이터를 학습하고 plot

- 학습률 :  $1e-4$ , 반복 100번
- 학습이 되지 않은 경우 plot: 아래 왼쪽 그림
- 학습된 완료 된 후 plot: 아래 오른쪽 그림



## 실습 1-1

3) 앞 문제에서 사용한 동일한 데이터를 scikit-learn 라이브러리를 사용하여 출력.



## 실습 1-2

- 다음은 P 자동차 회사의 차종과 마력, 그리고 평균연비(단위 : km/l)를 나타낸 표이다.

	A	B	C	D	E	F	G
마력	130	250	190	300	210	220	170
연비	16.3	10.2	11.1	7.1	12.1	13.2	14.2

1) scikit-learn을 이용하여 선형회귀모델을 구현하고 이 모델의 절편과 계수를 구하여라. 그리고 구현한 선형회귀 모델이 입력 마력값에 대해 연비를 예측하는데 얼마나 적합한지 예측 점수를 출력해보라.

```
계수 : [-0.05027473]
절편 : 22.58626373626374
예측 점수 : 0.8706727649378525
```

## 실습 1-2

2) 앞의 선형 회귀 모델을 이용하여 270마력을 가지는 새로운 자동차를 개발했을 때 예상되는 연비를 출력하라.

270 마력 자동차의 예상 연비 : 9.01 km/l

## 실습 1-2

자동차의 연비에 영향을 미치는 요소는 마력뿐만 아니라 총중량도 중요한 요소가 될 것이다. 다음은 P 자동차 회사의 차종과 마력, 뿐만 아니라 자동차의 총중량(단위:kg)을 추가한 표이다.

	A	B	C	D	E	F	G
마력	130	250	190	300	210	220	170
총중량	1900	2600	2200	2900	2400	2300	2100
연비	16.3	10.2	11.1	7.1	12.1	13.2	14.2

3) 위의 자료를 바탕으로 적절한 선형 회귀 모델을 구현하라. 이 모델의 계수와 절편, 예측모델의 점수를 출력하라.

```
계수 : [-0.00689189 -0.00731081]  
절편 : 30.60405405405406  
예측 점수 : 0.8871254041192391
```

## 실습 1-2

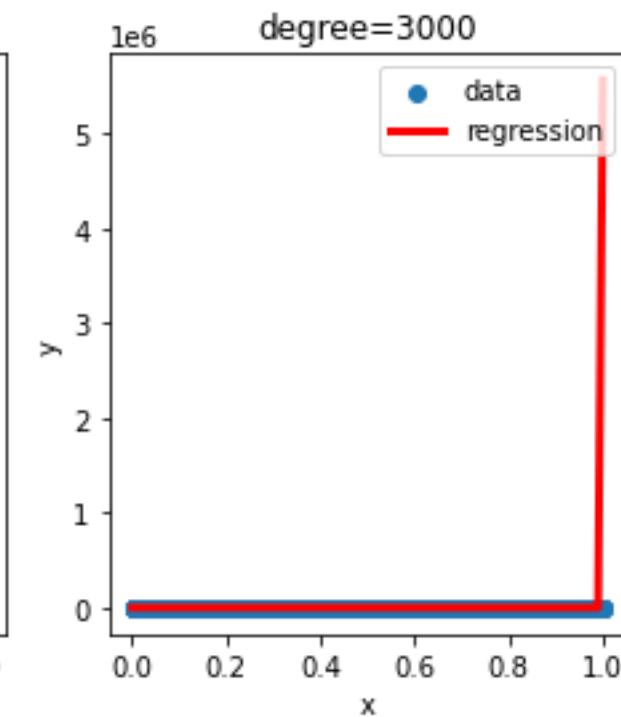
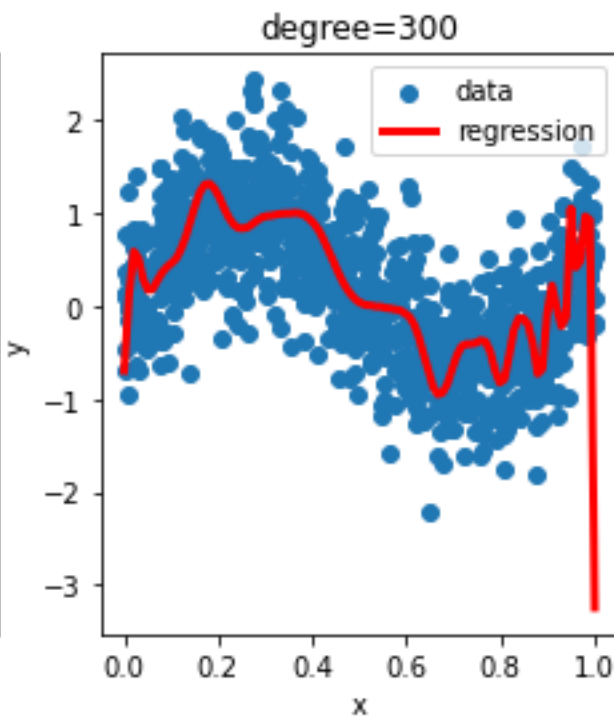
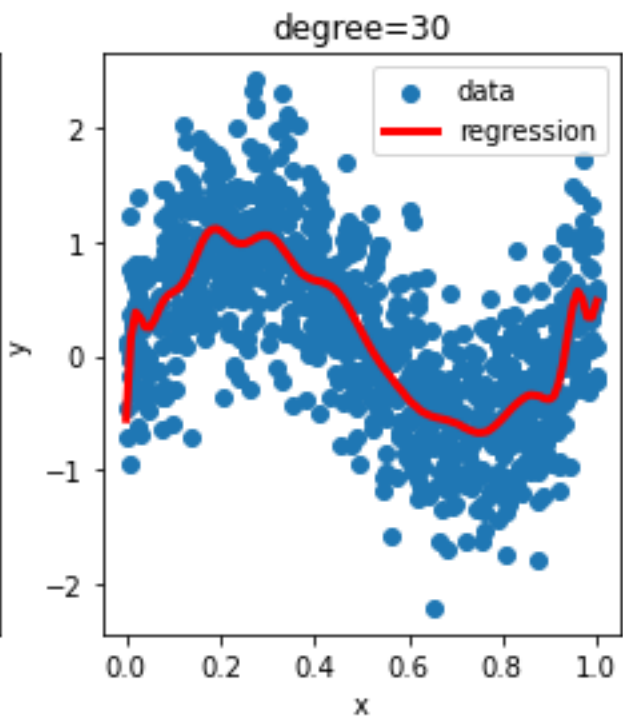
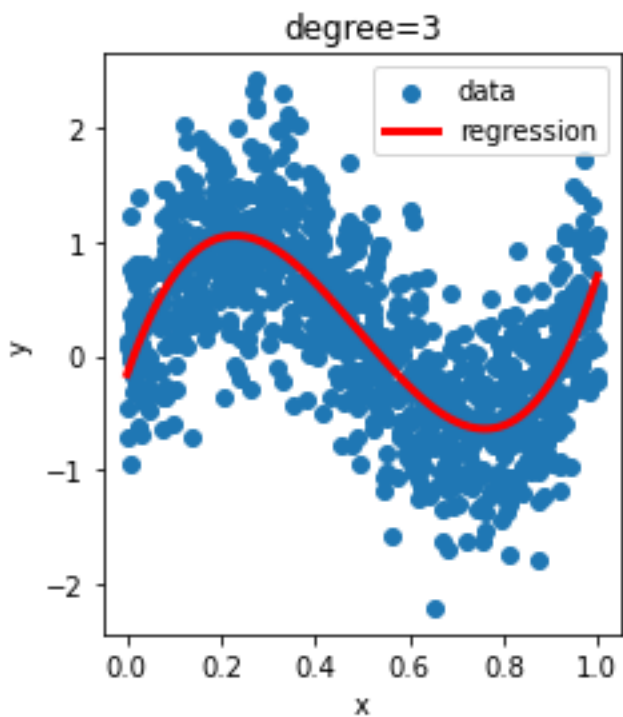
4) 앞의 선형 회귀 모델을 바탕으로 270마력의 신형엔진을 가진 총중량 2,500kg의 자동차를 개발하려 한다. 이 자동차의 연비를 선형 회귀 모델에 적용하여 다음과 같이 구해 보라.

270 마력 자동차의 예상 연비 : 10.47 km/l

# 실습 1-3

- 주어진 데이터(nonlinear.csv)를 불러와서 아래 문제를 해결하라

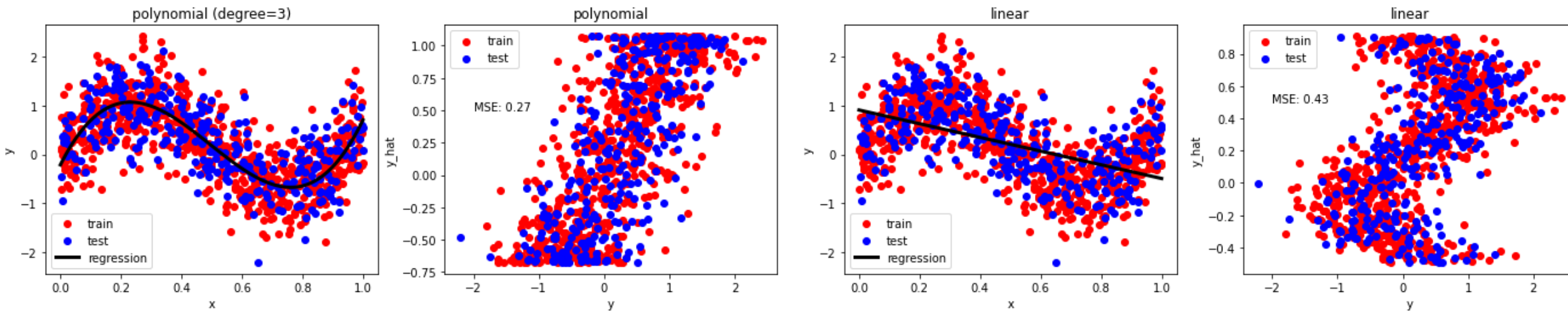
1) 3차, 30차, 300차, 3000차 다항회귀를 구현하여 각각 아래 회귀 곡선을 그려라.  
(회귀 곡선의 범위는 0부터 1사이 100개)



# 실습 1-3

- 주어진 데이터(nonlinear.csv)를 불러와서 아래 문제를 해결하라

2) 주어진 데이터를 train(70%), test(30%) 로 랜덤하게 구분하고, 3차 다항회귀와 선형회귀를 구현하여 아래 그래프를 plot하고 평균제곱오차(mean squared error)를 계산하라.





# 지도 학습

## 2. 분류

## 2.1 k-NN 알고리즘과 분류문제

- 선형회귀에 이어 **분류**classification와 **군집화**clustering
- 두 가지 방법 모두 데이터를 비슷한 집단으로 묶는다는 점에서는 유사
  - 분류는 소속 집단의 정보를 이미 알고 있는 상태에서 비슷한 집단으로 묶는 방법
  - 군집화란 소속 집단의 정보가 없는 상태에서 비슷한 집단으로 묶는 방법

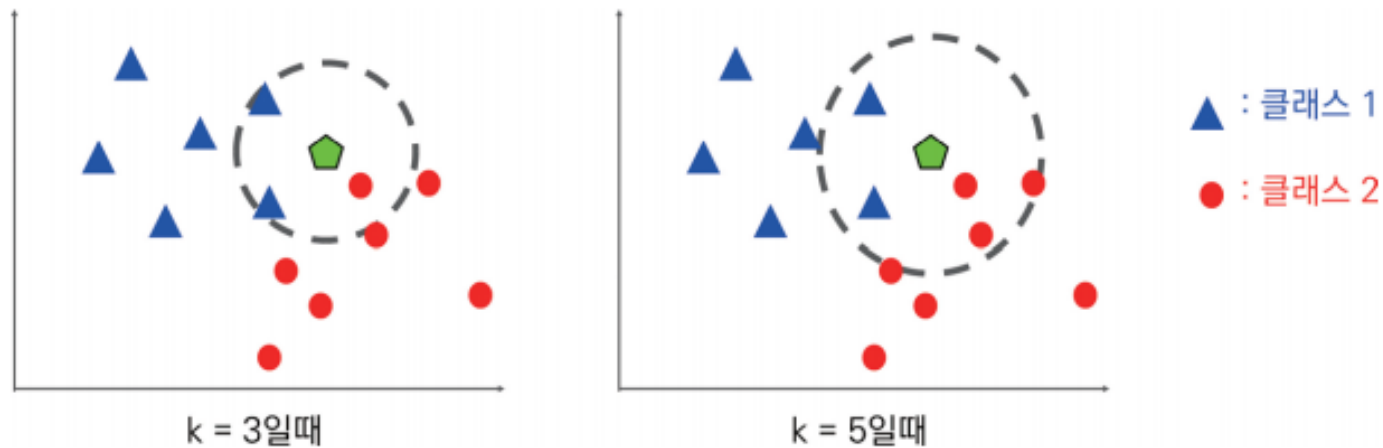
## 2.1 k-NN 알고리즘과 분류문제

- **닥스훈트** dachshund와 **사모예드** samoyed라는 종이 있는데, 두 종은 몸의 높이와 길이 비율이 서로 다름
- 몸통의 길이와 높이 **특징** feature을 각각  $x_1, x_2$ 라고 두고 이 개들의 표본 집합에 대하여  $x_1, x_2$ 를 측정하면 아래 그림의 오른쪽과 같은 산포도 그래프를 얻을 수 있음



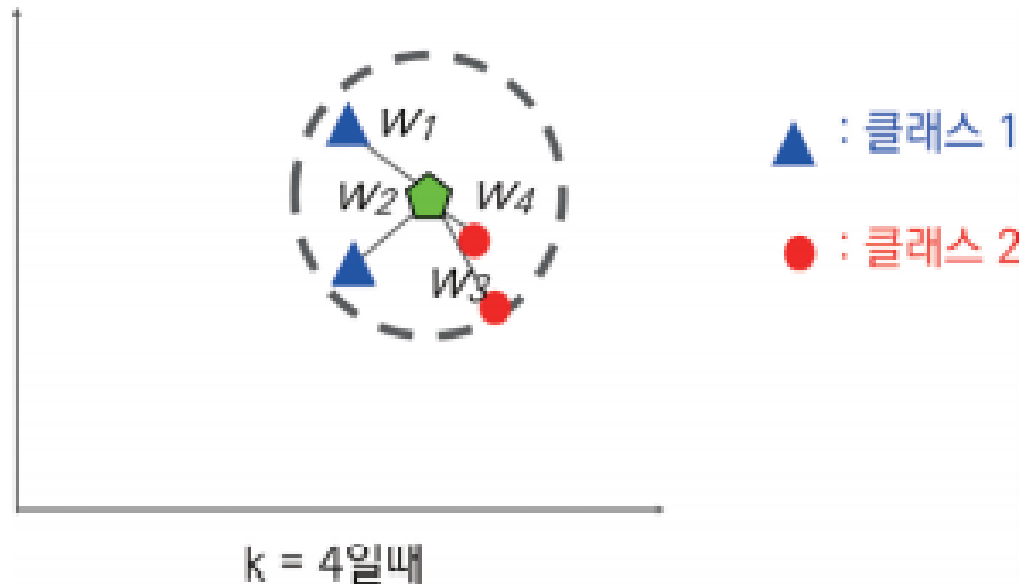
## 2.1 k-NN 알고리즘과 분류문제

- 두 가지 데이터를 효과적으로 분류하려면?
- k-NN 알고리즘에서 k-NN은 **k-최근접 이웃** (k-Nearest Neighbors)의 약자로 특징 공간에 분포하는 데이터에 대하여 k개의 가장 가까운 이웃을 살펴보고 다수결 방식으로 데이터의 레이블을 할당 하는 분류방식



## 2.1 k-NN 알고리즘과 분류문제

- k가 4와 같이 짝수이고 클래스 1과 클래스 2의 개수가 같은 경우에도 새 데이터의 클래스를 판정하는 방법
- 예를 들어 2개의 빨간색 원은 다른 2개의 파란색 삼각형보다 더 가까이 있다는 것을 고려해 클래스 2로 분류하는 것

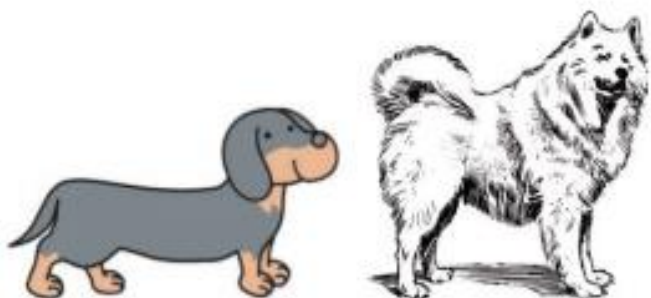


## 2.1 k-NN 알고리즘과 분류문제

- k-NN 방법은 특징 공간에 있는 모든 데이터에 대한 정보가 필요
  - 데이터 인스턴스, 클래스, 특징의 요소들의 개수가 많다면, 많은 메모리 공간과 계산 시간이 필요하다는 단점
  - 알고리즘이 매우 단순하고 직관적이며, 사전 학습이나 특별한 준비 시간이 필요 없다는 점은 장점

## 2.2 k-NN 알고리즘을 위한 데이터 준비

- 닥스훈트와 사모예드의 데이터를 바탕으로 k-NN 알고리즘 구현



닥스훈트 8마리의 길이와 높이								
길이	77	78	85	83	73	77	73	80
높이	25	28	29	30	21	22	17	35
사모예드 8마리의 길이와 높이								
길이	75	77	86	86	79	83	83	88
높이	56	57	50	53	60	53	49	61



```
import matplotlib.pyplot as plt
import numpy as np

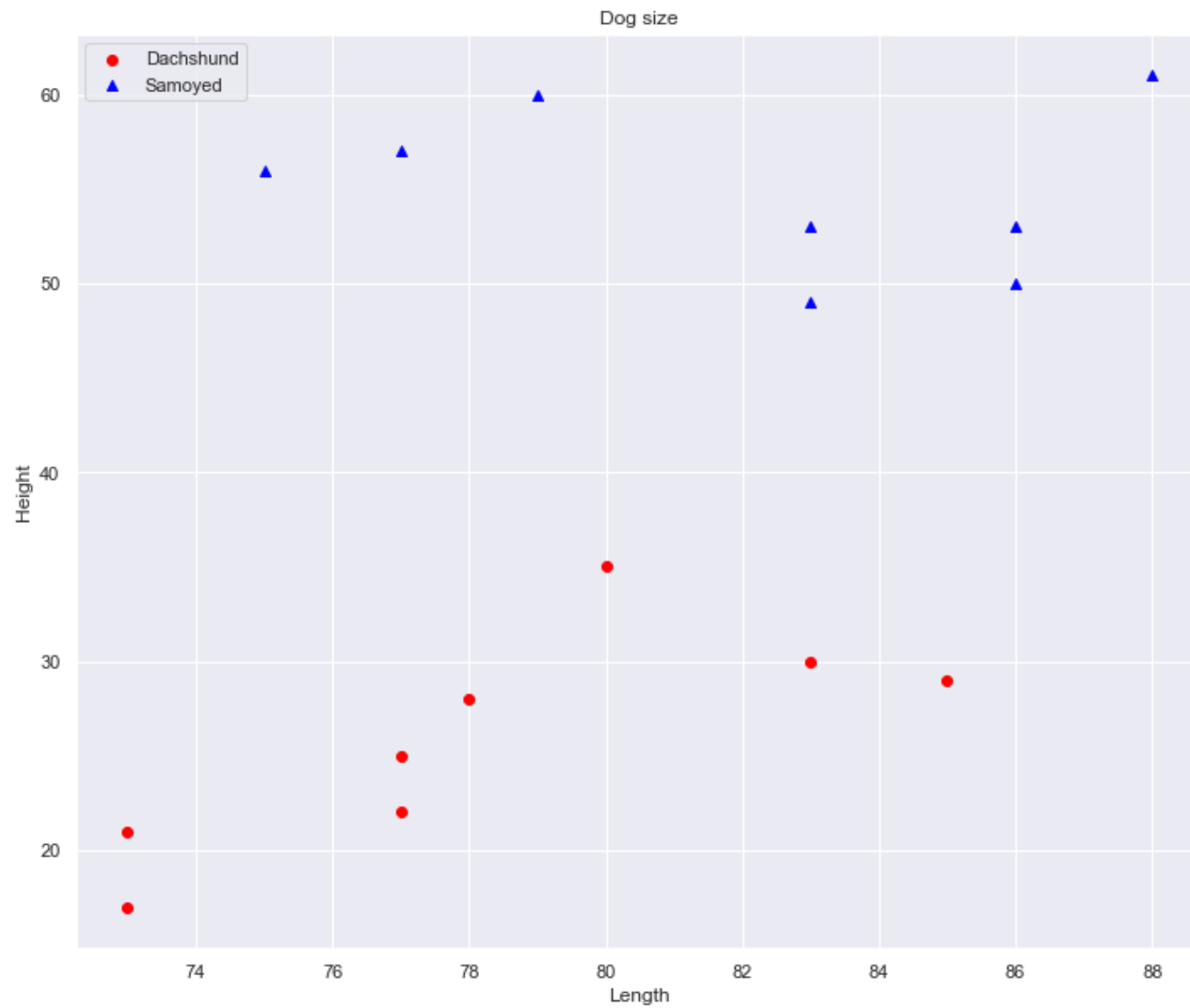
# 닥스훈트의 길이와 높이 데이터
dach_length = [77, 78, 85, 83, 73, 77, 73, 80]
dach_height = [25, 28, 29, 30, 21, 22, 17, 35]
# 사모에드의 길이와 높이 데이터
samo_length = [75, 77, 86, 86, 79, 83, 83, 88]
samo_height = [56, 57, 50, 53, 60, 53, 49, 61]

plt.scatter(dach_length, dach_height, c='red', label='Dachshund')
plt.scatter(samo_length, samo_height, c='blue', marker='^', label='Samoyed')

plt.xlabel('Length')
plt.ylabel('Height')
plt.title("Dog size")
plt.legend(loc='upper left')

plt.show()
```





- 이제 길이와 높이가 각각 79, 35인 새로운 데이터가 들어왔고 이 특징을 가진 개의 종류를 모를 경우, 이 데이터는 닥스훈트와 사모예드 중에서 어떤 개로 분류하는 것이 바람직할지 살펴 보자



# ...코드 생략

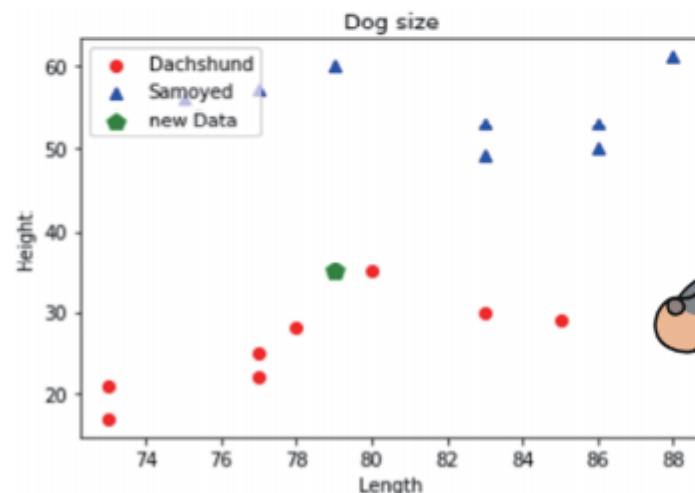
```
newdata_length = [79]    # 새로운 데이터의 길이
```

```
newdata_height = [35]    # 새로운 데이터의 높이
```

# ...코드 생략

# 새 데이터의 표식은 오각형(pentagon)으로 설정하고, 레이블은 new Data로

```
plt.scatter(newdata_length, newdata_height, s=100,  
            marker='p', c='green', label='new Data')
```



new Data는 두 클래스 사이에 위치하네요. 하지만 주위에 있는 가까운 점들이 대부분 닥스훈트 클래스예요



## 2.3 k-NN 분류기 구현

- 준비된 데이터에 k-NN 알고리즘을 적용
- 닥스훈트의 몸 길이 `dach_length`, 몸 높이 `dach_height` 배열 데이터를 묶어서 `[몸길이1, 몸높이1], [몸길이2, 몸높이2], ...`와 같은 형태의 항목을 가지는 `d_data` 배열을 만들어보자
- `d_data` 배열의 크기와 같은 0으로 이루어진 배열 `[0, 0, ..]`을 만들어 보자



```
d_data = np.column_stack((dach_length, dach_height))  
d_label = np.zeros(len(d_data))    # 닥스훈트는 0으로 레이블링
```

## 2.3 k-NN 분류기 구현

- 동일한 명령으로 사모예드 데이터 s\_data 배열을 만들고 s\_data 배열과 크기가 같은 1로 이루어진 s\_label 배열 [1, 1, ..]을 만들도록 하자



```
s_data = np.column_stack((samo_length, samo_height))  
s_label = np.ones(len(s_data)) # 사모예드는 1로 레이블링
```

- 이렇게 생성된 데이터와 레이블을 기반으로 k-NN 모델을 만들어 학습
  - 각 클래스의 특징을 바탕으로 분류를 실시할 데이터는 newdata라는 이름으로 생성



```
newdata = [[79, 35]]
```

## 2.3 k-NN 분류기 구현

- 아래의 코드에서 dog\_classes는 **키**key 값이 0일때 'Dachshund', 1일때 'Samoyed'라는 레이블을 가지도록 생성한 딕셔너리
- 딕셔너리는 fit() 함수를 통해 생성한 모델이 잘 적용되는가를 보기 쉽게 출력하기 위하여 사용



```
from sklearn.neighbors import KNeighborsClassifier  
  
dog_classes = {0:'Dachshund', 1:'Samoyed'}  
  
k = 3      # k를 3으로 두고 kNN 분류기를 만들어 보자  
knn = KNeighborsClassifier(n_neighbors = k)  
knn.fit(dogs, labels)  
y_pred = knn.predict(newdata)
```



```
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics

# 닥스훈트의 몸 길이와 몸 높이
dach_length = [77, 78, 85, 83, 73, 77, 73, 80]
dach_height = [25, 28, 19, 30, 21, 22, 17, 35]

# 사모예드의 몸 길이와 몸 높이
samo_length = [75, 77, 86, 86, 79, 83, 83, 88]
samo_height = [56, 57, 50, 53, 60, 53, 49, 61]

d_data = np.column_stack((dach_length, dach_height))
d_label = np.zeros(len(d_data)) # 닥스훈트는 0으로 레이블링
s_data = np.column_stack((samo_length, samo_height))
s_label = np.ones(len(s_data)) # 사모예드는 1로 레이블링

newdata = [[79, 35]]

dogs = np.concatenate((d_data, s_data))
labels = np.concatenate((d_label, s_label))

dog_classes = {0:'Dachshund', 1:'Samoyed'}

k = 3 # k를 3으로 두고 knn 분류기를 만들어 보자
knn = KNeighborsClassifier(n_neighbors = k)
knn.fit(dogs, labels)
y_pred = knn.predict(newdata)
print('데이터', newdata, ', 판정 결과:', dog_classes[y_pred[0]])
```

데이터 [[79, 35]] , 판정 결과: Dachshund

## 2.4 k-NN 활용 예제 - 붓꽃 데이터

- 붓꽃<sup>iris</sup>은 관상용으로도 재배되는 아름다운 꽃이며 크기와 색상이 다른 여러 종이 있음



**Iris Versicolor**



**Iris Setosa**



**Iris Virginica**

## 2.4 k-NN 활용 예제 - 붓꽃 데이터

- 세 붓꽃 종의 이름은 *Versicolor*, *Setosa*, *Virginica*
- 각 종에 따라 꽃받침의 길이와 너비, 꽃잎의 길이와 너비가 약간씩 차이
- 꽃받침과 꽃잎의 크기를 측정한 데이터를 기반으로 새로운 종을 분류하는 k-NN 모델을 구축

```
from sklearn.datasets import load_iris

iris = load_iris()
iris.data[:5]      # 최초 5개 데이터의 값을 출력

array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2]])
```

- 각각의 속성은 꽃받침 길이(sepal length)와 너비(sepal width) 및 꽃잎 길이(petal length)와 너비(petal width)를 cm 단위로 측정한 값



## 2.4 k-NN 활용 예제 - 붓꽃 데이터

- Numpy 다차원 배열인 iris.data를 조금 다루기 쉽게 Pandas의 DataFrame으로 만들
- 다섯 번째 속성이 0, 1, 2의 소속 클래스의 값을 갖는 target 속성임을 알 수 있음



```
import pandas as pd
```

```
iris_df = pd.DataFrame(iris.data, columns=iris.feature_names)
```

```
iris_df['target'] = pd.Series(iris.target)
```

```
iris_df.head()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

## 2.5 k-NN 활용 예제 - 붓꽃 데이터로 학습

- describe() 를 통해서 그 속성을 상세히 살펴보며 다음과 같은 결과를 얻을 수 있음

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
count	150.000000	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.057333	3.758000	1.199333	1.000000
std	0.828066	0.435866	1.765298	0.762238	0.819232
min	4.300000	2.000000	1.000000	0.100000	0.000000
25%	5.100000	2.800000	1.600000	0.300000	0.000000
50%	5.800000	3.000000	4.350000	1.300000	1.000000
75%	6.400000	3.300000	5.100000	1.800000	2.000000
max	7.900000	4.400000	6.900000	2.500000	2.000000



```
iris_df['target'].value_counts()
```

```
2    50  
1    50  
0    50
```

```
Name: target, dtype: int64
```

붓꽃 데이터에는 3개의  
종에 대하여 각각 50개의  
특징값 데이터들이 저장  
되어 있어요.



```
iris_df.values
```

```
array([[5.1, 3.5, 1.4, 0.2, 0. ],  
       [4.9, 3. , 1.4, 0.2, 0. ],  
       [4.7, 3.2, 1.3, 0.2, 0. ],  
       ...])
```

5개의 속성값으로 이루어진  
다차원 배열



```
X = iris_df.iloc[:, :4]
y = iris_df.iloc[:, -1]
```

제일 마지막 속성에는  
0, 1, 2 중 하나의 레이블  
값이 들어 있어요.

- X, y를 훈련 데이터와 검증 데이터로 구분하여 k-NN 알고리즘을 적용시키고 정확도를 측정



```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics

def iris_knn(X, y, k):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
    knn = KNeighborsClassifier(n_neighbors = k)
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_test)
    return metrics.accuracy_score(y_test, y_pred)

k = 3
scores = iris_knn(X, y, k)
print('n_neighbors가 {0:d}일때 정확도: {1:.3f}'.format(k, scores))
```

n\_neighbors가 3일때 정확도: 0.956

k가 3일때 95.6% 정확도로 예측

## 2.6 새로운 꽃에 대해서 모델을 적용하고 분류

- 이번에는 훈련 데이터와 검증 데이터를 나누어서 훈련하지 않고 사용가능한 모든 데이터를 사용해서 모델을 학습



```
from sklearn.datasets import load_iris
from sklearn.neighbors import KNeighborsClassifier

iris = load_iris()
k = 3
knn = KNeighborsClassifier(n_neighbors = k)
knn.fit(iris.data, iris.target)
```

## 2.6 새로운 꽃에 대해서 모델을 적용하고 분류

- knn.fit()을 통해서 k-NN 분류기 모델을 얻었으며 이 분류기 모델이 전혀 다른 적이 없는 새로운 데이터를 가지고 예측



```
classes = {0:'setosa', 1:'versicolor', 2:'virginica'}
```

```
# 새로운 데이터를 제시해 보자.
```

```
X = [[4, 2, 1.3, 0.4],  
      [4, 3, 3.2, 2.2]]
```

```
y = knn.predict(X)
```

```
print('{} 특성을 가지는 품종: {}'.format(X[0], classes[y[0]]))
```

```
print('{} 특성을 가지는 품종: {}'.format(X[1], classes[y[1]]))
```

```
[4, 2, 1.3, 0.4] 특성을 가지는 품종: setosa
```

```
[4, 3, 3.2, 2.2] 특성을 가지는 품종: versicolor
```

## 2.6 새로운 꽃에 대해서 모델을 적용하고 분류

- 테스트 데이터의 규모가 적은 편이므로 전체 데이터에 대해 분류를 실시
- iris.data를 분류기에 넣고, 결과 y\_pred\_all을 구한 뒤에 정답인 iris.target과 비교할 수 있도록 준비



```
y_pred_all = knn.predict(iris.data)
scores = metrics.accuracy_score(iris.target, y_pred_all)
print('n_neighbors가 {0:d}일때 정확도: {1:.3f}'.format(k, scores))
```

```
n_neighbors가 3일때 정확도: 0.960
```

## 2.6 새로운 꽃에 대해서 모델을 적용하고 분류

- 혼동 행렬 `confusion matrix`: 분류 결과를 행렬 형태로 표현



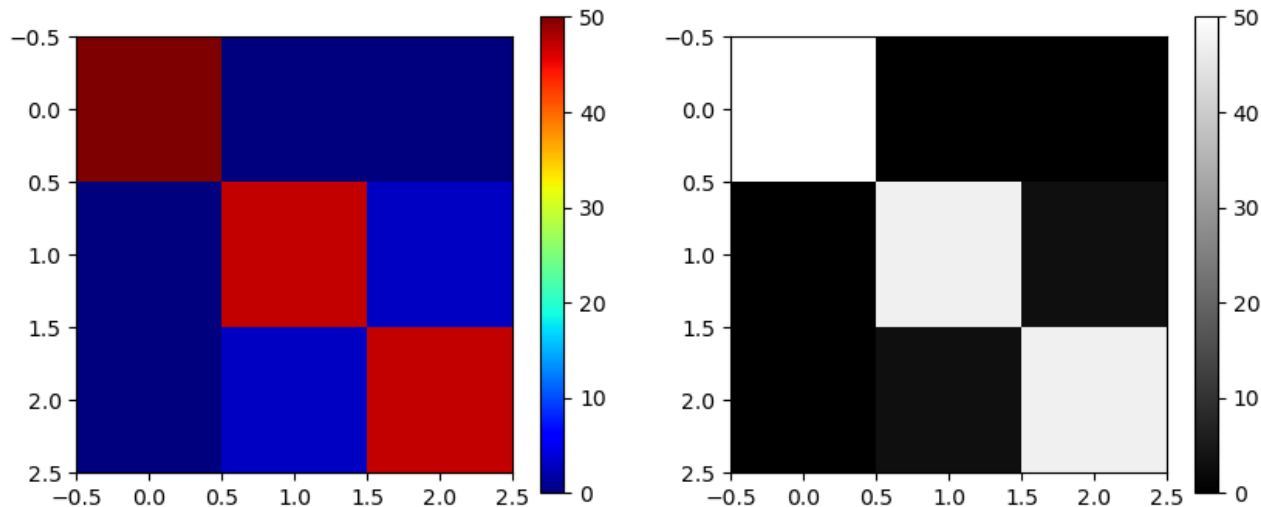
```
from sklearn.metrics import confusion_matrix  
conf_mat = confusion_matrix(iris.target, y_pred_all)  
conf_mat
```

```
array([[50,  0,  0],  
       [ 0, 47,  3],  
       [ 0,  3, 47]])
```

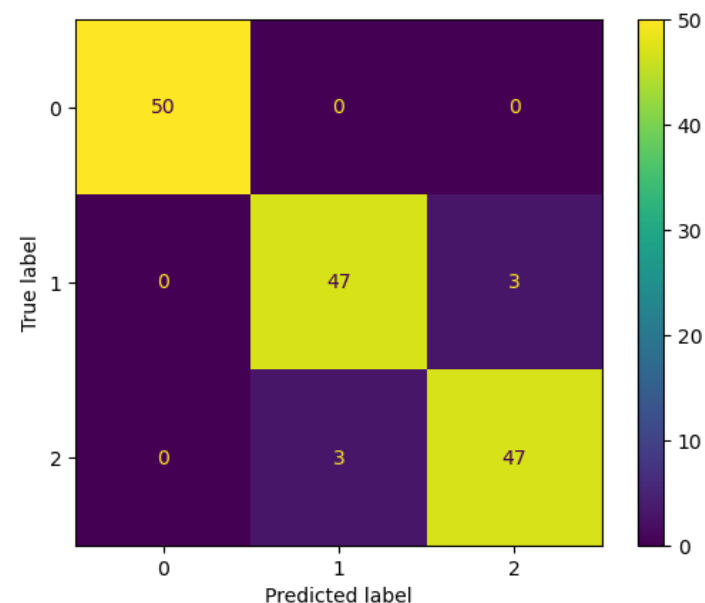
## 2.6 새로운 꽃에 대해서 모델을 적용하고 분류

- 결과에 대해 (정답, 예측) 데이터를 만들어 그래프 그리기

```
plt.figure(figsize=(10,4))  
plt.subplot(121)  
plt.imshow(conf_mat,cmap=plt.cm.jet)  
plt.colorbar()  
plt.subplot(122)  
plt.imshow(conf_mat,cmap=plt.cm.gray)  
plt.colorbar()
```



```
from sklearn.metrics import ConfusionMatrixDisplay  
ConfusionMatrixDisplay(confusion_matrix = conf_mat).plot()
```





## 2.7 표집 편향과 성능 측정을 위한 평가지표

- 실제 데이터를 다룰 경우 99%의 정확도를 가지는 분류기는 매우 구현하기 어려우며, 더구나 단 두 줄 만으로 만든 아래의 분류기는 매우 신뢰하기 힘들



```
def classifier_A(length, height): # 입력값에 관계없이 'Samoyed'를 반환
    return 'Samoyed'
```

## 2.7 표집 편향과 성능 측정을 위한 평가지표

- 분류기 A가 입력으로 9,900 마리의 사모예드와 100 마리의 닥스훈트 종의 길이와 높이를 받는다고 가정
- 분류기 A의 정확도는 다음과 같다.

$$\begin{aligned}\text{분류기 A의 정확도} &= \text{제대로 분류한 데이터} / \text{전체 데이터} \\ &= 9,900 / 10,000 \\ &= 0.99 (= 99 \%) \end{aligned}$$

- 표본sample 데이터가 편향된 경우 우수한 머신러닝 알고리즘을 사용한다고 할지라도 학습 성능의 개선을 기대하기는 힘들.
  - 표집편향sampling bias이라고 함

## 2.7 표집 편향과 성능 측정을 위한 평가지표

- 정밀도<sup>precision</sup>와 재현율<sup>recall</sup>을 살펴보고 이 성능을 행렬형태로 표시한 혼동 행렬<sup>confusion matrix</sup> 혹은 오차행렬에 대해서 알아보도록 하자.
- 100명의 환자와 100 명의 건강한 사람에 대하여 이 검사 키트의 성능을 테스트한다고 가정
  - 검사 키트가 COVID-19 환자(양성<sup>positive:P</sup>)에 대해서 5명을 음성
  - COVID-19에 감염되지 않은 건강한 사람(음성<sup>negative:N</sup>)에 대해서 89명을 음성으로, 11명을 양성으로 판정

		KoKIT22의 예측값 (검사결과)					
		음성			양성		
환자의 실제 상태값		N			P		
음성 (COVID 안걸림)	N	89 TN	T 일치	N 예측	11 FP	F 불일치	P 예측
양성 (COVID 걸림)	P	5 FN	F 불일치	N 예측	95 TP	T 일치	P 예측

## 2.7 표집 편향과 성능 측정을 위한 평가지표

- 정확도 Accuracy는 다음 식을 이용해서 구할 수 있음
- 전체 데이터(FP+FN+TP+TN)중에서 제대로 판정한 데이터(TP + TN)의 비율

$$Acc = \frac{TP+TN}{FP+FN+TP+TN} = \frac{95+89}{11+5+95+89} = 0.92$$

		KoKIT22의 예측값 (검사결과)					
		음성			양성		
환자의 실제 상태값		N			P		
음성 (COVID 안걸림)	N	89 TN	T 일치	N 예측	11 FP	F 불일치	P 예측
양성 (COVID 걸림)	P	5 FN	F 불일치	N 예측	95 TP	T 일치	P 예측

## 2.7 표집 편향과 성능 측정을 위한 평가지표

- 진짜 양성 비율 True Positive Rate: TPR은 재현율 recall이라고도 함
- 양성환자 중에서 이 키트가 올바르게 양성이라고 분류한 환자의 비율로 다음과 같은 식 (TPR 혹은 Rec로 표기함)

$$TPR = Rec = \frac{TP}{P} = \frac{TP}{FN + TP} = \frac{95}{100} = 0.95$$

		KoKIT22의 예측값 (검사결과)					
		음성			양성		
환자의 실제 상태값		N			P		
음성 (COVID 안걸림)	N	89 TN	T 일치	N 예측	11 FP	F 불일치	P 예측
양성 (COVID 걸림)	P	5 FN	F 불일치	N 예측	95 TP	T 일치	P 예측

## 2.7 표집 편향과 성능 측정을 위한 평가지표

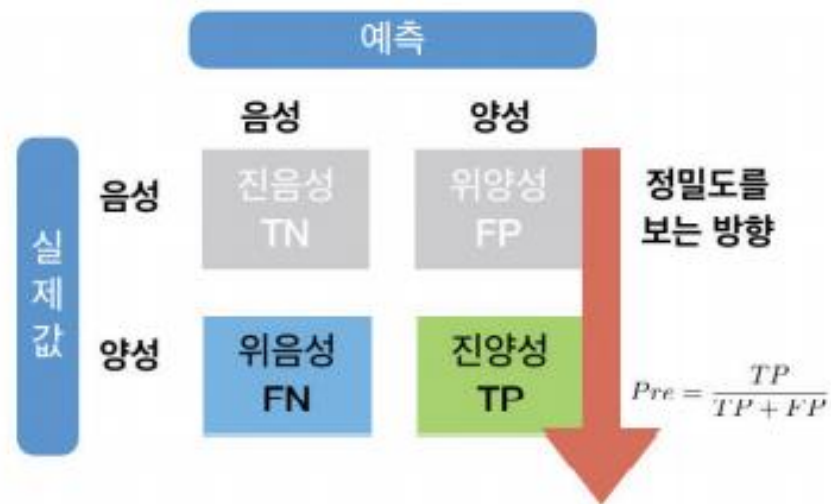
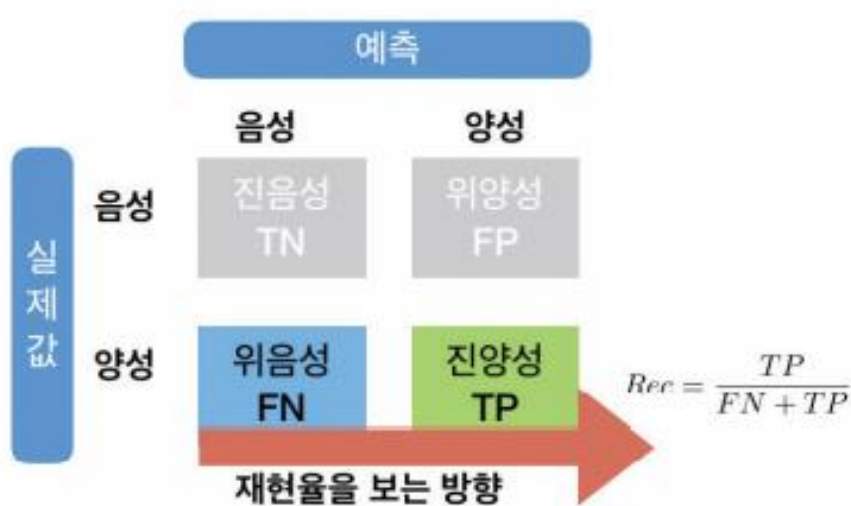
- 정밀도 *precision*은 검사 키트가 확진자로 분류한 사람들 중 실제 양성인 경우로 다음과 같은 식(*Pre*로 표기함)

$$Pre = \frac{TP}{TP+FP} = \frac{95}{106} = 0.896$$

		KoKIT22의 예측값 (검사결과)					
		음성			양성		
환자의 실제 상태값		N			P		
음성 (COVID 안걸림)	N	89 TN	T 일치	N 예측	11 FP	F 불일치	P 예측
양성 (COVID 걸림)	P	5 FN	F 불일치	N 예측	95 TP	T 일치	P 예측

## 2.8 성능지표 함수

- Recall : 실제로 COVID-19에 양성인 대상자를 양성으로 분류할 확률
- Precision : 양성으로 예측한 사람들 중에서 실제 COVID-19 양성자일 확률



## 2.8 성능지표 함수

- 재현율과 정밀도라는 각각의 지표는 관심있는 척도가 다르기 때문에 하나의 척도만을 측정 방법으로 사용할 경우 왜곡이 발생할 수 있음
- 두 지표를 조합한 F1 점수도 사용 하는데 이 점수는 다음과 같은 식
- F1 점수는 정밀도와 재현율의 **조화 평균**harmonic mean

$$F_1 = \frac{2}{\frac{1}{Pre} + \frac{1}{Rec}} = 2 \frac{Pre \times Rec}{Pre + Rec}$$



## 2.8 성능지표 함수

- 다음과 같이 10개의 0 값(음성), 10개의 1 값(양성)으로 이루어진 목표값 target
- 이 목표값에 대한 예측값이 pred 넘파이 배열에 들어있는 경우를 가정



```
target = np.array([0] * 10 + [1] * 10)  
target  # 10개의 0(Negative), 10개의 1(Positive)를 가짐
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```



```
pred = np.array([0,0,0,0,0,1,1,1,0,0,1,1,1,1,1,1,0,0,1,1])  
pred
```

```
array([0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1])
```

## 2.8 성능지표 함수

- 목표값과 예측값을 confusion matrix로 표시



```
from sklearn.metrics import confusion_matrix  
confusion_matrix(target, pred)  # 혼동행렬 만들기
```

```
array([[7, 3],  
       [2, 8]])
```

- F1 점수 계산



```
from sklearn.metrics import precision_score, recall_score  
print('정밀도 :', precision_score(target, pred))  
print('재현율 :', recall_score(target, pred))
```

```
from sklearn.metrics import accuracy_score, f1_score  
print('정확도 :', accuracy_score(target, pred))  
print('F1 점수 :', f1_score(target, pred))
```

```
정밀도 : 0.7272727272727273  
재현율 : 0.8  
정확도 : 0.75  
F1 점수 : 0.761904761904762
```

# 비지도 학습

## 3. 군집화

## 3.1 군집화의 개념과 비지도 학습

- **군집화**clustering란 소속집단의 정보가 없고 모르는 상태에서 비슷한 집단으로 묶는 **비지도 학습**
- 입력 데이터를 바탕으로 출력값을 예측하는 목적으로 사용되기 보다는 데이터에서 의미를 파악하고 기준을 만드는 목적으로 사용

	분류	군집화
소속집단에 대한 정보	있다	없다
레이블 유무	있다	없다
종류	지도 학습	비지도 학습
공통점	데이터를 비슷한 집단으로 묶는 방법	

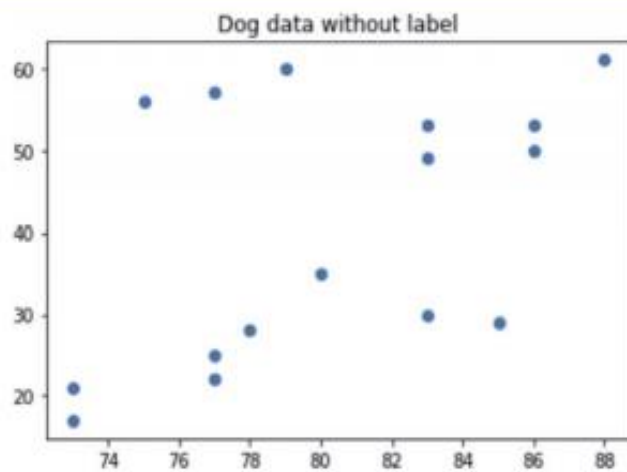


```
import numpy as np
import matplotlib.pyplot as plt

..
# 사모예드와 닥스훈트의 길이, 높이 데이터는 중복되어 생략한다
# 개의 길이와 높이를 각각 ndarray 형태로 만든다
dog_length = np.array(dach_length + samo_length) # 리스트를 이어 ndarray로
dog_height = np.array(dach_height + samo_height) # 리스트를 이어 ndarray로

dog_data = np.column_stack((dog_length, dog_height))

plt.title("Dog data without label")
plt.scatter(dog_length, dog_height)
```



이전 절의 데이터와는 달리  
어느 데이터가 닥스훈트인지  
사모예드인지 사전 정보가  
없어요.

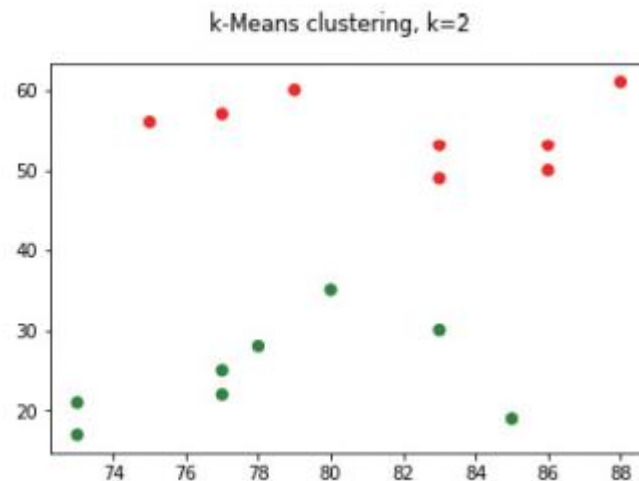
- 데이터를 두 개의 그룹으로 나누는 k-평균 알고리즘을 적용
  - k-평균 k-means 알고리즘은 scikit-learn에서 제공하는 cluster 모듈에 존재



```
from sklearn import cluster
```

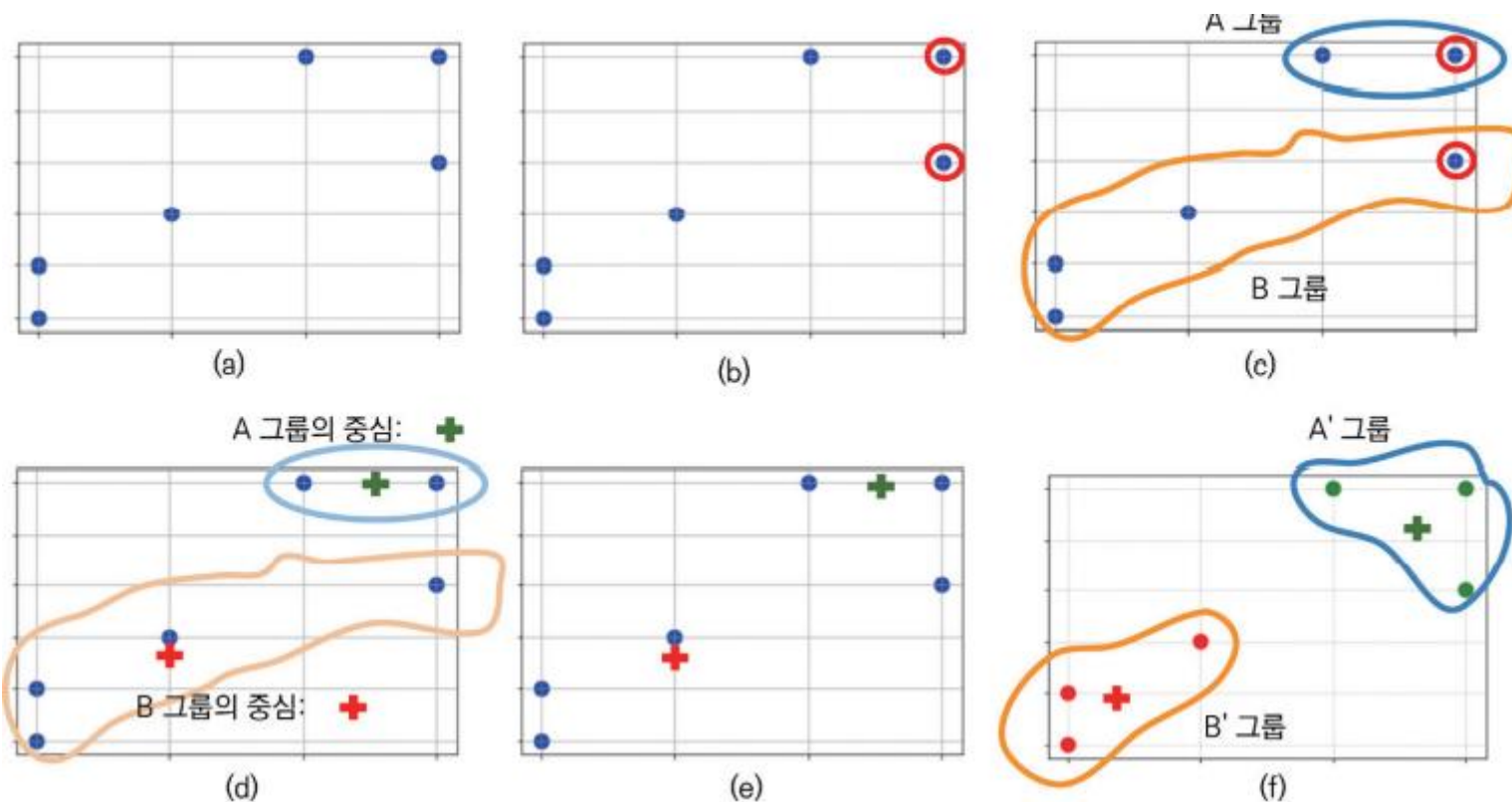
```
def kmeans_predict_plot(X, k):  
    model = cluster.KMeans(n_clusters=k)  
    model.fit(X)  
    labels = model.predict(X)  
    colors = np.array(['red', 'green', 'blue', 'magenta'])  
    plt.suptitle('k-Means clustering, k={}'.format(k))  
    plt.scatter(X[:, 0], X[:, 1], color=colors[labels])
```

```
kmeans_predict_plot(dog_data, k = 2)
```



## 3.2 k-평균 알고리즘

- k-평균 알고리즘은 원리가 단순하고 직관적이며 성능이 좋은 군집화 알고리즘  
단, 사전에 군집의 개수  $k$  값을 지정해야 하는 단점.

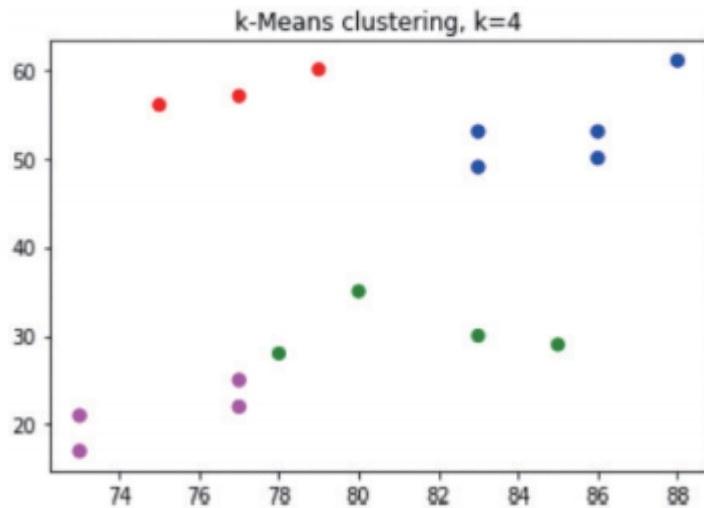
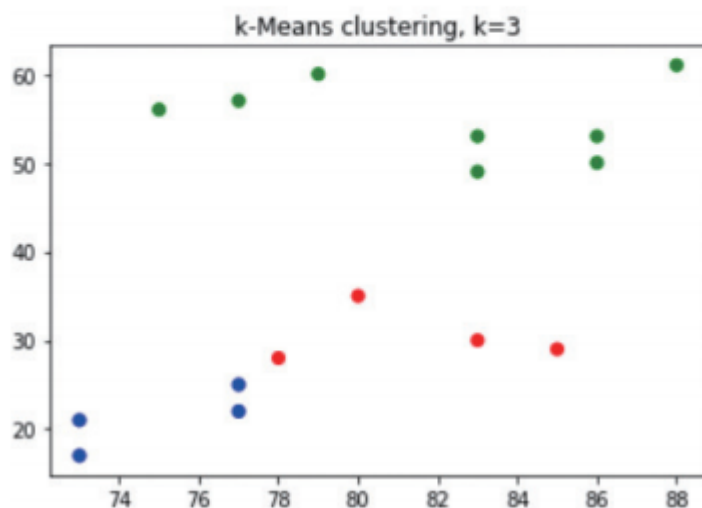


## 3.2 k-평균 알고리즘

- k-평균 알고리즘은 데이터 분포에 대한 사전 지식이 없을 경우에도 사용가능
- 군집에 대한 사전 정보가 있을 경우 이를 바탕으로 모범 샘플을 사용하여 그 샘플을 초기화 에 사용할 경우 더 나은 성능을 보일 수 있음



```
kmeans_predict_plot(dog_data, k = 3) # 3개의 군집 생성  
kmeans_predict_plot(dog_data, k = 4) # 4개의 군집 생성
```





실습

## 실습 2-1

- 다음은 철수네 동물병원에 치료를 받은 개의 종류와 그 크기 데이터이다. 이 데이터를 바탕으로 k-NN 알고리즘을 적용해 보자.

닥스훈트

길이	75	77	83	81	73	99	72	83
높이	24	29	19	32	21	22	19	34

사모예드

길이	76	78	82	88	76	83	81	89
높이	55	58	53	54	61	52	57	64

말티즈

길이	35	39	38	41	30	57	41	35
높이	23	26	19	30	21	24	28	20

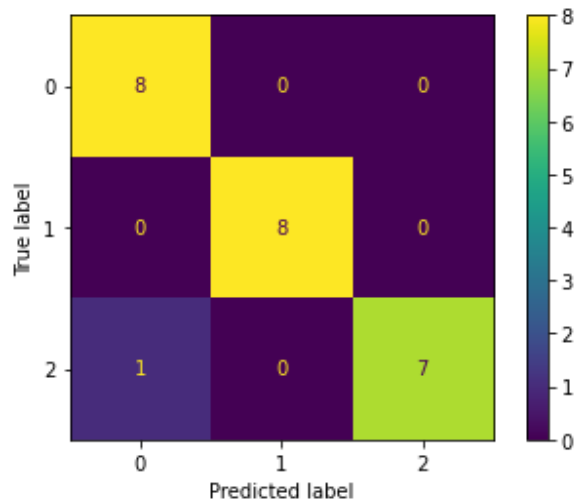
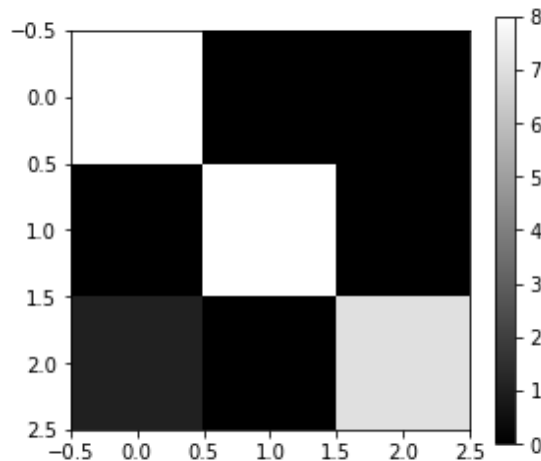
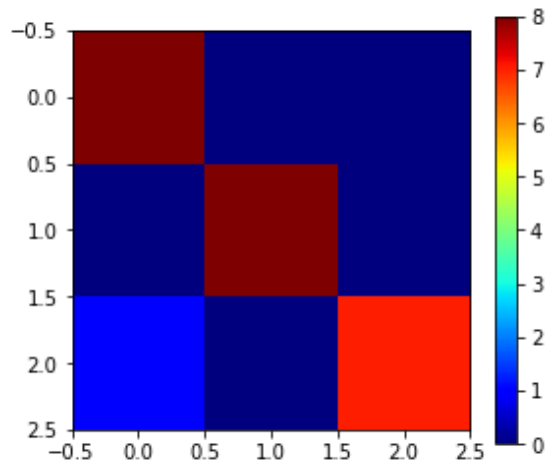
## 실습 2-1

1) 위의 정보를 바탕으로 닥스훈트를 0, 사모예드를 1, 말티즈를 2로 Labeling하여 데이터와 Label을 각각 생성하도록 하여라. 그리고 다음과 같이 각 견종별 데이터를 2차원 배열로 만들어서 출력하라.

```
닥스훈트(0) : [[75, 24], [77, 29], [83, 19], [81, 32], [73, 21], [99, 22], [72, 19], [83, 34]]  
사모예드(1) : [[76, 55], [78, 58], [82, 53], [88, 54], [76, 61], [83, 52], [81, 57], [89, 64]]  
말티즈(2) : [[35, 23], [39, 26], [38, 19], [41, 30], [30, 21], [57, 24], [41, 28], [35, 20]]
```

2) k 값이 3일때 k-NN 분류기의 분류결과를 다음과 같은 confusion matrix를 출력하고 이를 아래와 같이 2가지 색깔 형태와 ConfusionMatrixDisplay() 함수를 이용하여 plot 하라.

```
[[8 0 0]  
 [0 8 0]  
 [1 0 7]]
```



## 실습 2-1

3) 앞에서 계산한 분류 결과값 중 닥스훈트와 말티즈의 결과값만 가져와서 정밀도 (precision), 재현율(recall), 정확도(accuracy), F1 점수(F1-score)를 함수를 사용하지 않고 계산하고 이를 scikit-learn의 함수를 이용한 값과 비교하여 출력하라.

```
내가 만든 Precision: 1.0
내가 만든 Recall: 0.875
내가 만든 Accuracy: 0.9375
내가 만든 F1-score: 0.9333333333333333

scikit-learn의 Precision: 1.0
scikit-learn의 Recall: 0.875
scikit-learn의 Accuracy: 0.9375
scikit-learn의 F1-score: 0.9333333333333333
```

## 실습 2-1

4) 다음과 같은 개의 길이, 높이 데이터 A, B, C, D에 대하여 각각 k를 3, 5, 7로 하여 아래와 같이 분류하고 그 분류 결과를 출력하여라.

A : 길이 58, 높이 30

B : 길이 80, 높이 26

C : 길이 80, 높이 41

D : 길이 75, 높이 55

```
A 데이터 분류결과
A [[58, 30]] : k가 3 일때 : 말티즈
A [[58, 30]] : k가 5 일때 : 말티즈
A [[58, 30]] : k가 7 일때 : 닥스훈트

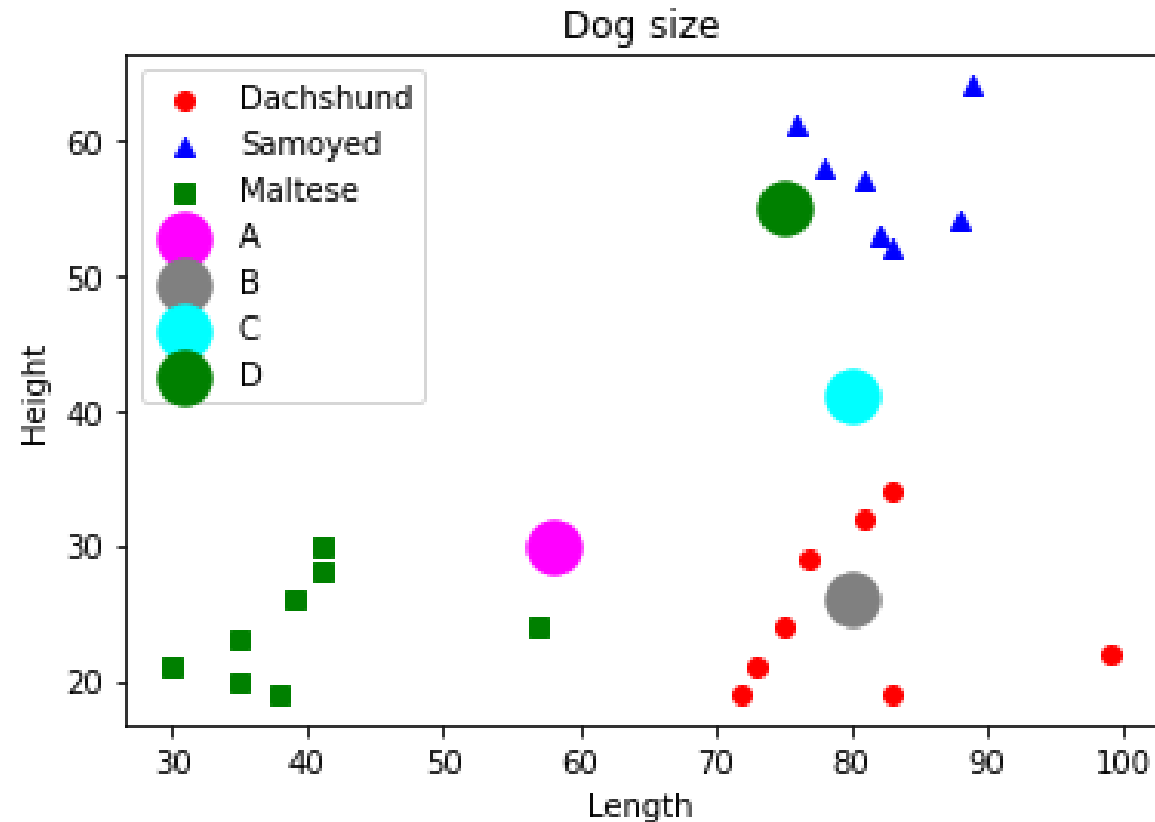
B 데이터 분류결과
B [[80, 26]] : k가 3 일때 : 닥스훈트
B [[80, 26]] : k가 5 일때 : 닥스훈트
B [[80, 26]] : k가 7 일때 : 닥스훈트

C 데이터 분류결과
C [[80, 41]] : k가 3 일때 : 닥스훈트
C [[80, 41]] : k가 5 일때 : 닥스훈트
C [[80, 41]] : k가 7 일때 : 사모예드

D 데이터 분류결과
D [[75, 55]] : k가 3 일때 : 사모예드
D [[75, 55]] : k가 5 일때 : 사모예드
D [[75, 55]] : k가 7 일때 : 사모예드
```

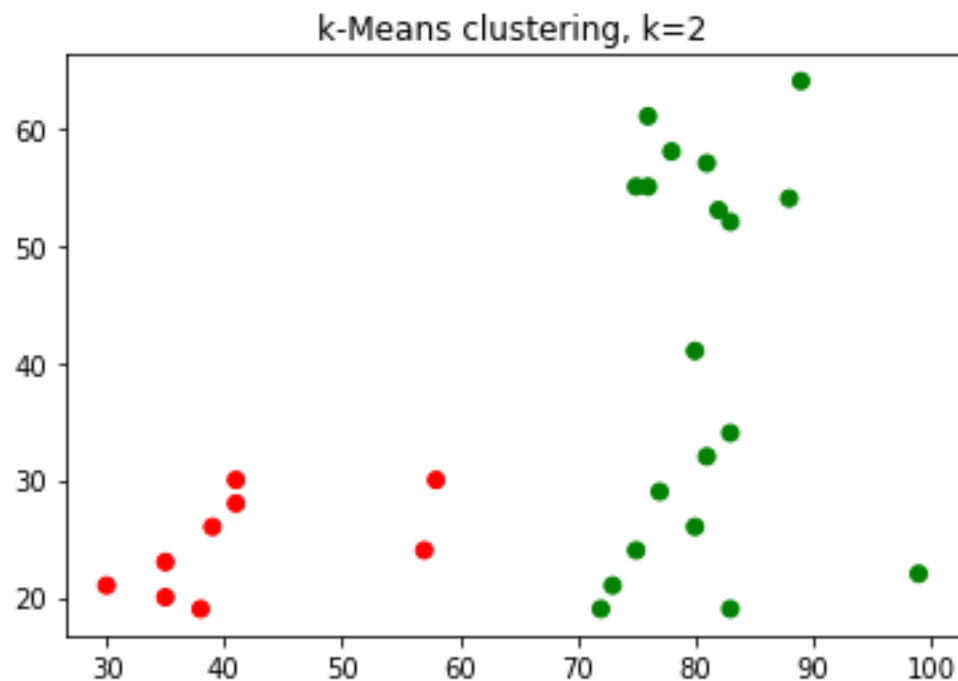
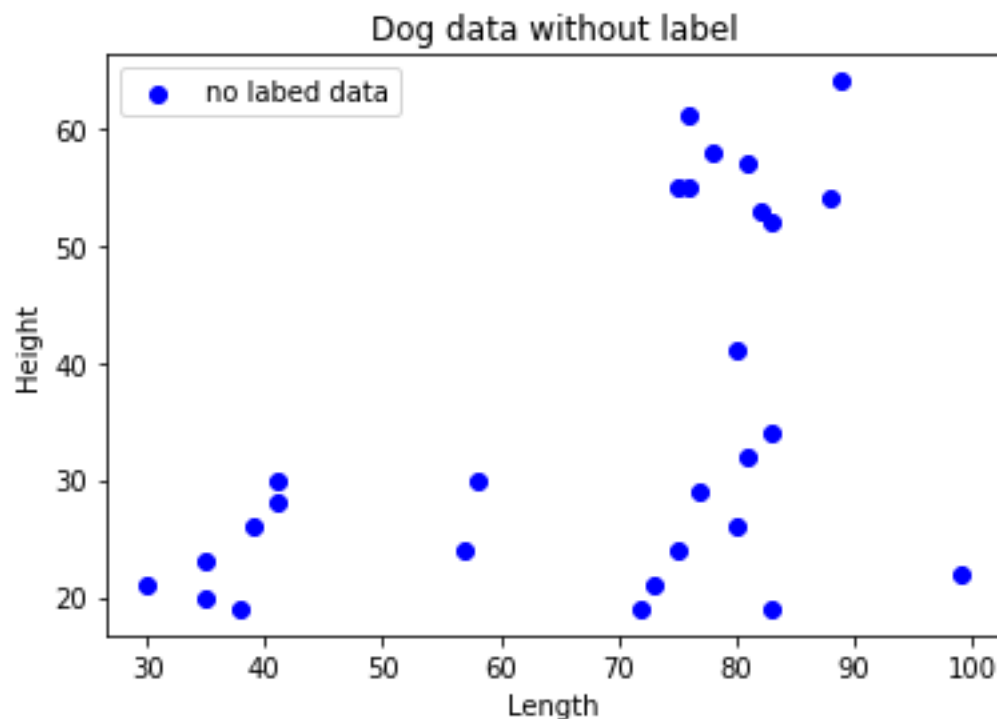
## 실습 2-1

5) 앞의 데이터를 기반으로 아래와 같은 산포도(scatter) 그래프를 그려서 다음과 같이 A,B,C,D 데이터를 나타내라. (가로축은 길이, 세로축은 높이)

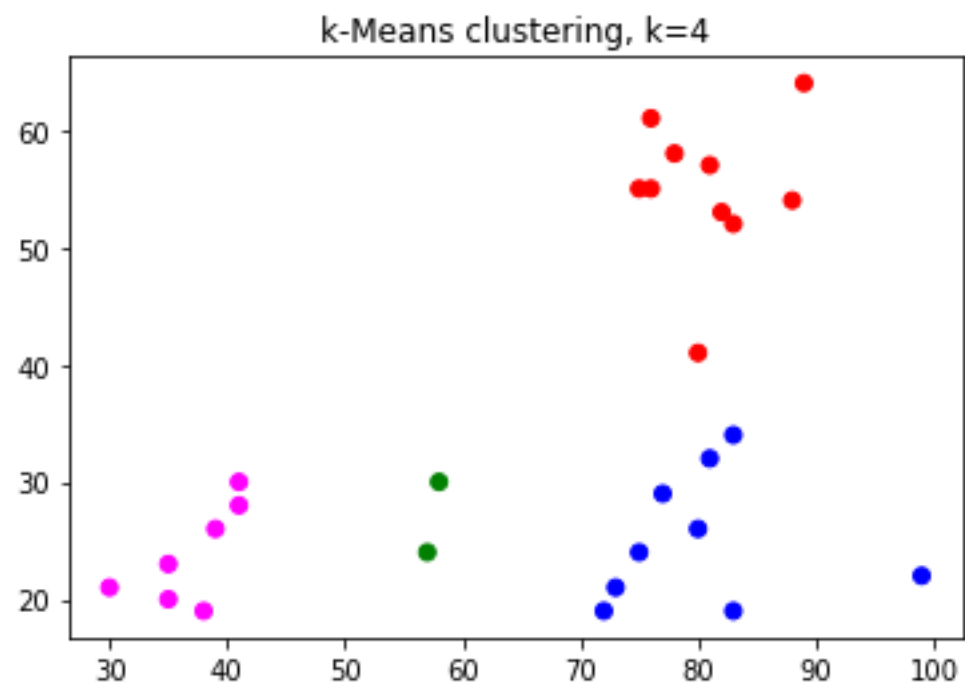
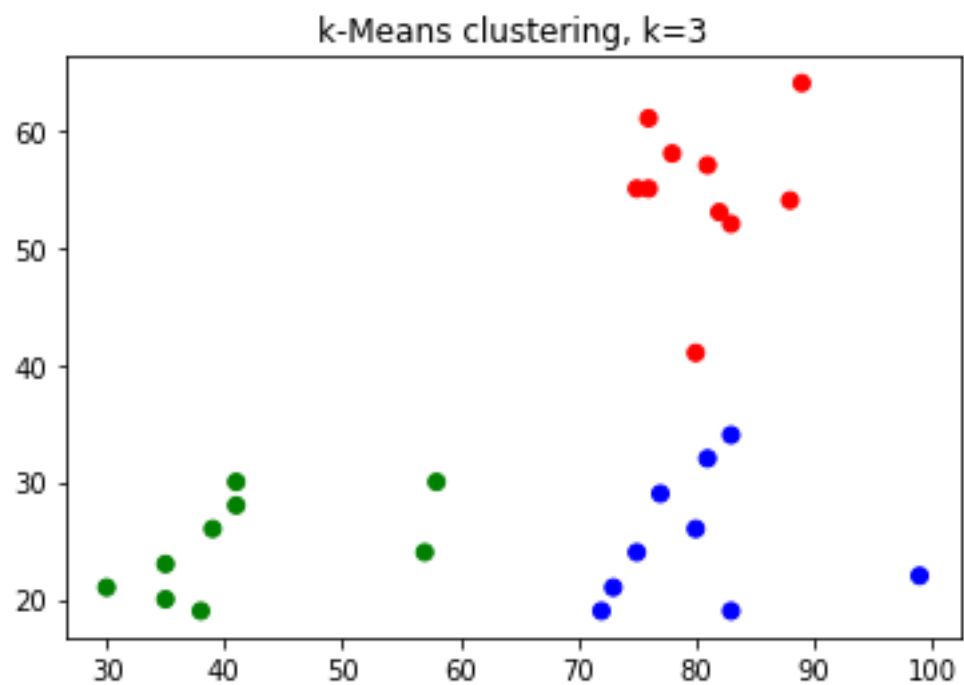


## 실습 2-1

6) 원래의 데이터와 A, B, C, D 데이터를 모두 포함한 dog\_data를 만들어라. 이 때 데이터가 가진 모든 레이블을 삭제하고 k-평균 알고리즘을 적용하여 군집화 (clustering)를 수행하고, 다음과 같이 k가 2, 3, 4일 때의 수행결과를 시각화하라.



# 실습 2-1





## 실습 2-2

- scikit-learn을 이용하여 iris 데이터를 불러와 k-means 군집화 알고리즘을 적용한 후 정확도(accuracy)를 계산하라.
- 데이터를 불러오면 레이블(target)도 포함되어 있지만 군집화를 할 때는 활용하지 않음

```
iris 데이터의 군집화 정확도: ?
```

감사합니다