5. 선형회귀

목차

01

선형회귀의 이해 및 기초 수식

01 선형회귀의 이해 및 기초 수식

1. 선형회귀의 개념

- 선형회귀(Linear Regression) : 종속변수 y와 한 개 이상의 독립변수 x와의 선형 상관관계를 모델링하는 회귀분석 기법
- 기존 데이터를 활용해 연속형 변수값을 예측
- 다음과 같은 수식을 만들고 a와 b의 값을 찾아냄

$$y = ax + b$$

- 단순 선형 회귀: 독립변수 x가 하나인 선형 회귀

$$y = ax + b$$

- 다중 선형 회귀: 독립 변수 x가 여러 개인 선형 회귀

$$y = w_1x_1 + w_2x_2 + \cdots + w_{13}x_{13} + w_0x_0 = \sum_{j=0}^{13} = w_jx_j = w^TX$$

01 선형회귀의 이해 및 기초 수식

1. 선형회귀의 개념

- 앞으로 개봉할 영화 예상 관객 수 y를 예측하는 문제

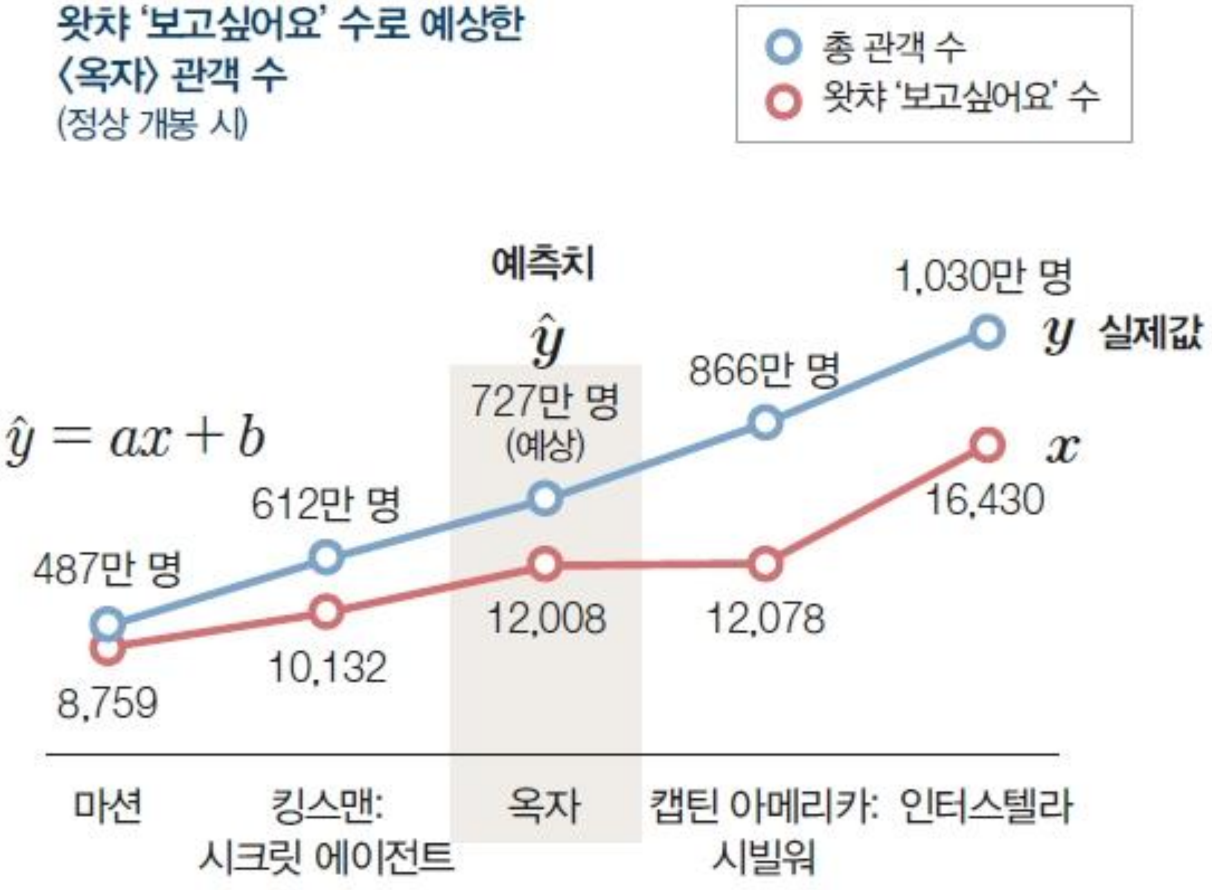


그림 7-1 왓챠 '보고싶어요' 수로 예상한 '옥자' 관객 수

01 선형회귀의 이해 및 기초 수식

1. 선형회귀의 개념

- 실제 관객 수를 y로 표현하여 좌표평면 상에 나타냄

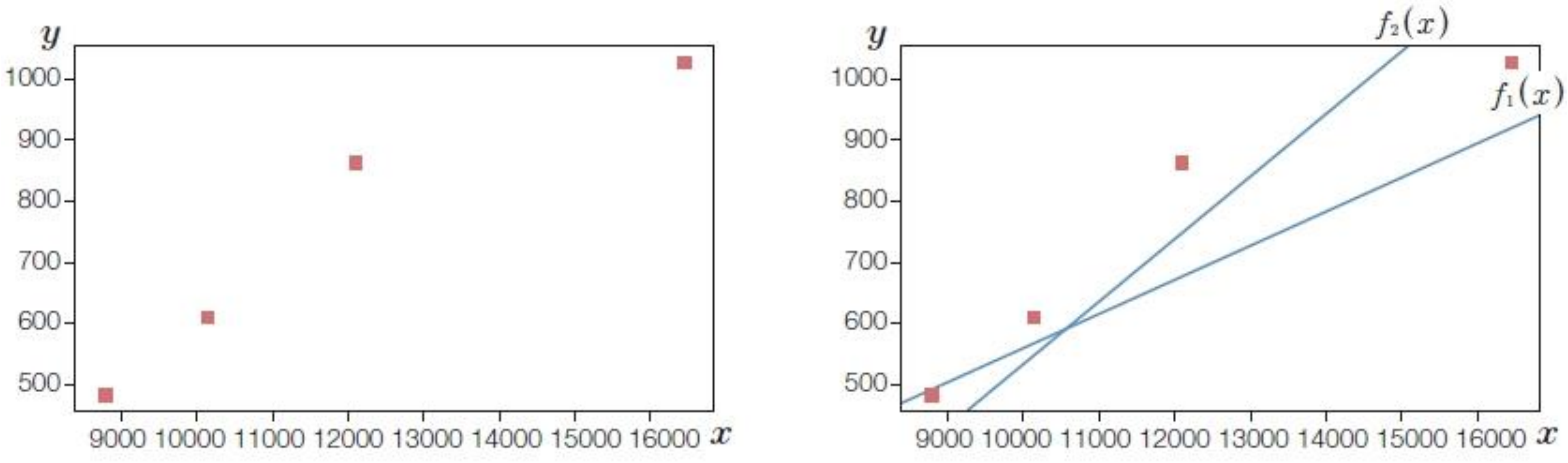


그림 7-2 좌표평면에 작성한 2개의 직선, $f_1(x)$와 $f_2(x)$

01 선형회귀의 이해 및 기초 수식

1. 선형회귀의 개념

- 두 그래프 중 어떤 것이 기존 데이터를 '잘 표현하는가'
- 예측값이 실제값 대비 차이가 많이 나지 않는 그래프

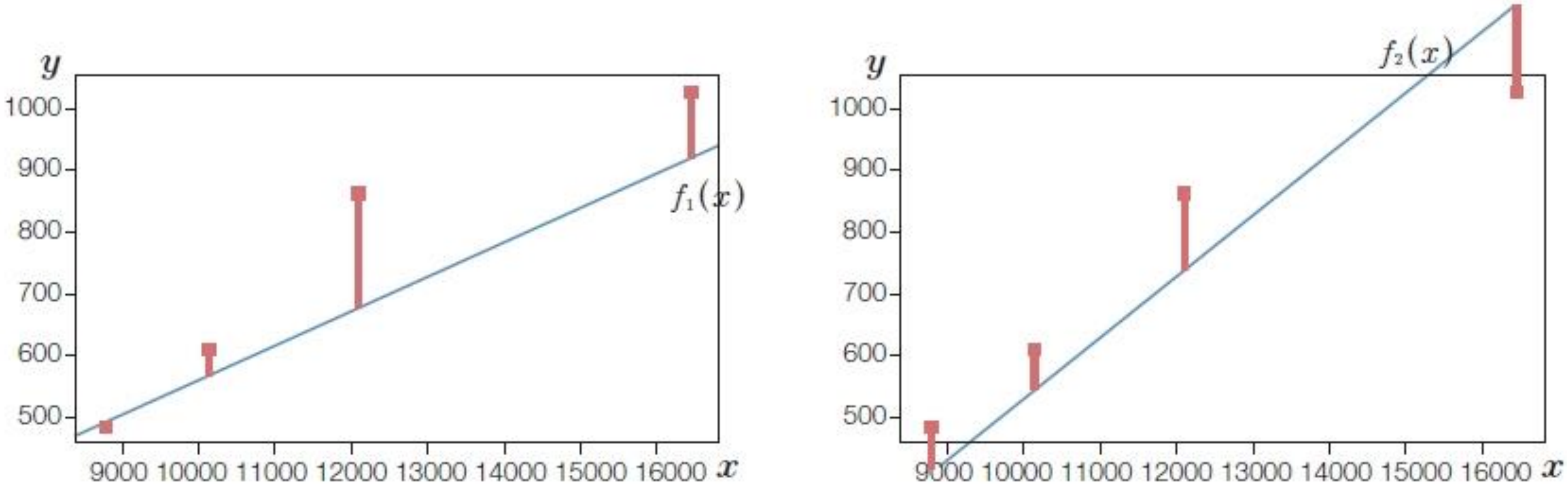


그림 7-3 $f_2(x)$가 $f_1(x)$보다 좀 더 데이터를 잘 표현하고 있다.

2. 예측 함수와 실제값 사이의 차이

- 예측 함수는 예측값과 실제값 사이의 차이를 최소화하는 방향으로
- 데이터 n개 중 i번째 데이터의 y 값에 대한 실제값과 예측값의 차이 $\hat{y}^i - y^i$
- 데이터가 5개 있을 때 5개 데이터의 오차의 합

$$(\hat{y}^{(1)} - y^{(1)}) + (\hat{y}^{(2)} - y^{(2)}) + (\hat{y}^{(3)} - y^{(3)}) + (\hat{y}^{(4)} - y^{(4)}) + (\hat{y}^{(5)} - y^{(5)})$$

- 오차 값들이 음수와 양수로 나왔을 때 값들 간의 차이가 상쇄되어 0으로 계산될 수 있음

01 선형회귀의 이해 및 기초 수식

2. 예측 함수와 실제값 사이의 차이

- 값의 제곱을 사용하여 오차의 합을 표현

$$(\hat{y}^{(1)}-y^{(1)})^2+(\hat{y}^{(2)}-y^{(2)})^2+(\hat{y}^{(3)}-y^{(3)})^2+(\hat{y}^{(4)}-y^{(4)})^2+(\hat{y}^{(5)}-y^{(5)})^2$$

$$\sum_{i=1}^{n}(\hat{y}^{(i)}-y^{(i)})^2$$

- 같은 식을 행렬로 표현

$$\hat{y}=\begin{bmatrix} w_1\times 8759+w_0 \\ w_1\times 10132+w_0 \\ w_1\times 12078+w_0 \\ w_1\times 16430+w_0 \end{bmatrix} \qquad y=\begin{bmatrix} 487 \\ 612 \\ 866 \\ 1030 \end{bmatrix}$$

$$(\hat{y}-y)^2=\begin{bmatrix} (w_1\times 8759+w_0-487)^2 \\ (w_1\times 10132+w_0-612)^2 \\ (w_1\times 12078+w_0-866)^2 \\ (w_1\times 16430+w_0-1030)^2 \end{bmatrix}$$

2. 예측 함수와 실제값 사이의 차이

- 제곱 오차(square error) : $(\hat{y}-y)^2$로 예측값과 실제값의 제곱을 표시하여 오차를 나타냄

- 제곱 오차를 최소화하는 w_0와w_1을 찾아야 함

$$\sum_{i=1}^{n}\left(w_i x^{(i)}+w_0\times 1-y^{(i)}\right)^2$$

3. 비용함수의 개념

- 비용함수(cost function) : 머신러닝에서 최소화해야 할 예측값과 실제값의 차이
- 가설함수(hypothesis function) : 예측값을 예측하는 함수

$$f(x) = h_\theta(x)$$

• 함수 입력값은 x이고 함수에서 결정할 것은 θ, θ가 가중치(weight) 값인 w_n을 의미함.

01 선형회귀의 이해 및 기초 수식

3. 비용함수의 개념

- 비용함수가 두 개의 가중치 값으로 결정됨

$$J(w_0, w_1) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - (y^{(i)})^2)$$

- 잔차의 제곱합(Error sum of squares) : 예측값인 가설함수와 실제값인 y값 간의 차이를 제곱해서 모두 합함
  - 총 데이터는 m개가 존재하고 각 데이터의 예측값과 실제값을 뺀 후 제곱한 값들을 모두 합한 값
- 손실함수(loss function) : 비용함수에서 잔차의 제곱합 부분
- 평균 제곱 오차(mean squared error, MSE) : 잔차의 제곱합을 2m으로 나눈 값, m이 아니라 2m으로 나눈 이유는 미분을 좀 더 명확하게 보여주기 위함임.

02

경사하강법으로 선형회귀 풀기

02 경사하강법으로 선형회귀 풀기

1. 경사하강법의 개념

- **경사하강법(gradient descent)** : 경사를 하강하면서 수식을 최소화하는 매개변수의 값을 찾아내는 방법, 현재 머신러닝의 핵심이라고 할 수 있는 **딥러닝의 기본이 되는 알고리즘**

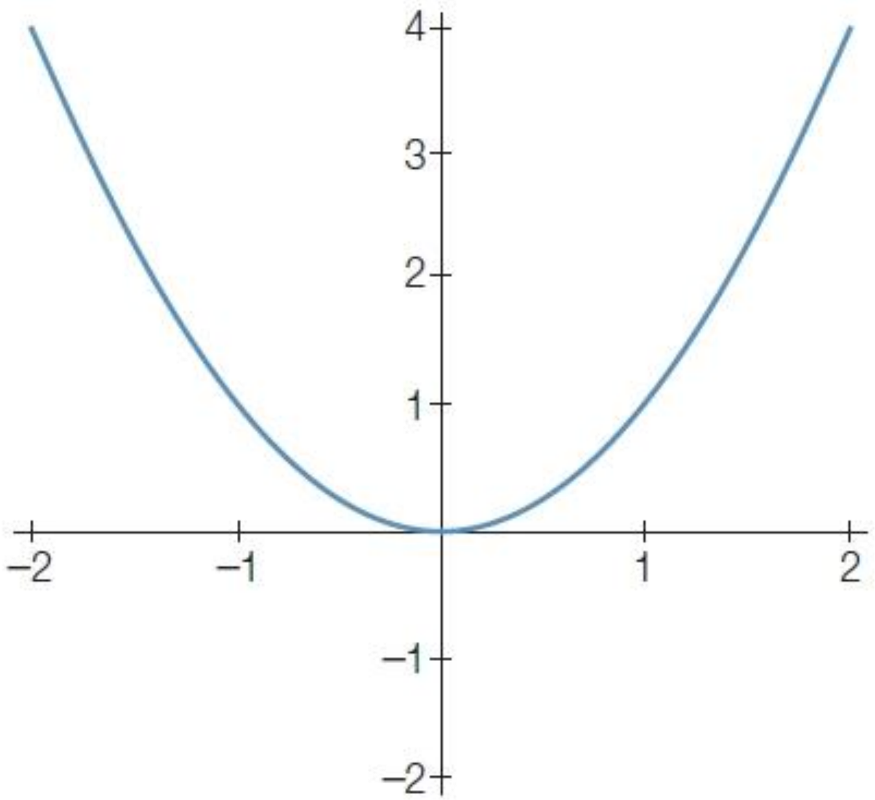


그림 7-5 $y = x^2$ 그래프

02 경사하강법으로 선형회귀 풀기

1. 경사하강법의 개념

- 점이 최솟값을 달성하는 방향으로 점점 내려감
  - 몇 번 적용할 것인가? : 많이 실행할수록 최솟값에 가까워짐
  - 한 번에 얼마나 많이 내려갈 것인가? : 한 번에 얼마나 많은 공간을 움직일지를 기울기, 즉 경사라고 부름
    - 경사(gradient) : 경사하강법의 하이퍼 매개변수

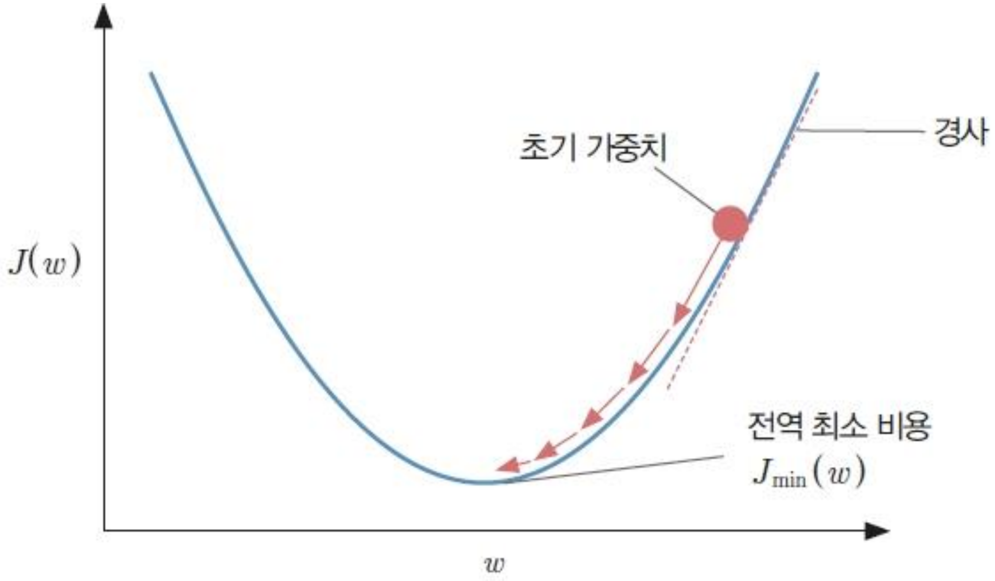


그림 7-6 $y = x^2$ 그래프에 경사하강법 적용

02 경사하강법으로 선형회귀 풀기

2. 경사하강법 알고리즘

- $f(x)$는 최소화시켜야 하는 값이고, $2x$는 이를 미분한 값인 경사

$$f(x) = x^2 \quad \rightarrow \quad \frac{dy}{dx} = 2x$$

- 경사하강법의 기본 수식

$$x_{new} = x_{old} - \alpha \times (2x_{old})$$

  - x_{old} 는 현재의 x 값, x_{new}는 경사 값이 적용된 후 생성된 값
- 경사만큼의 변화가 계속 x에 적용되어 x의 최솟값 찾음
  - 반복적으로 미분 값을 적용시키면서 더 이상 값이 변하지 않거나 변화가 미미해지는 지점까지 값이 줄어든다

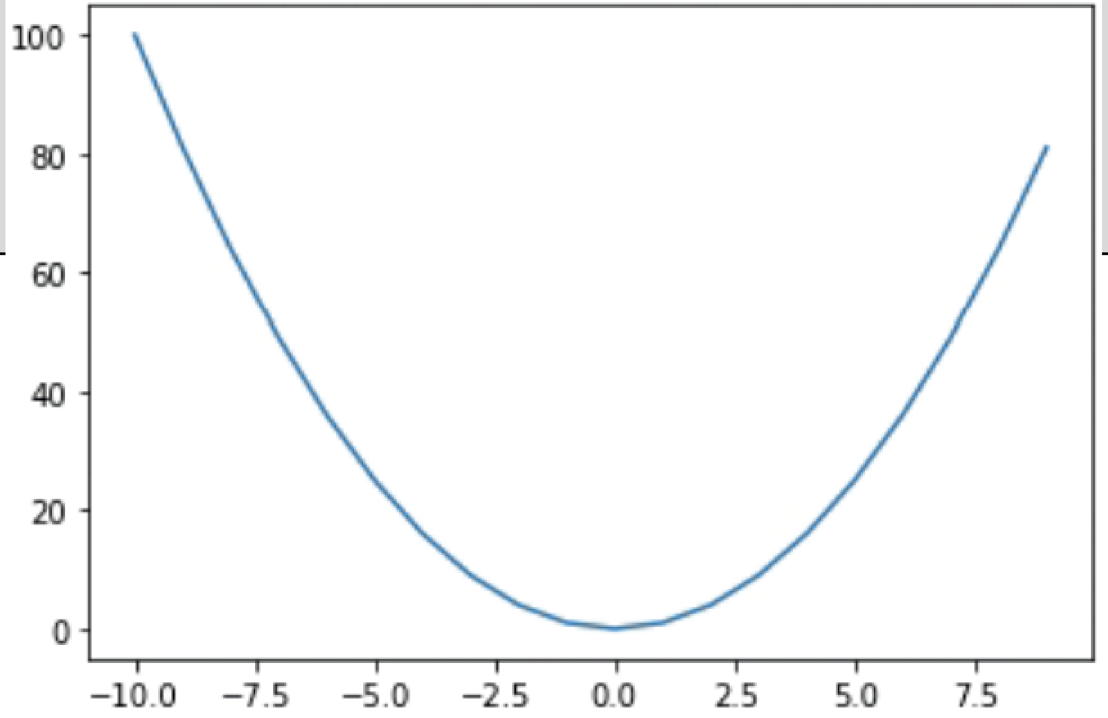
02 경사하강법으로 선형회귀 풀기

2. 경사하강법 알고리즘

In [1]:

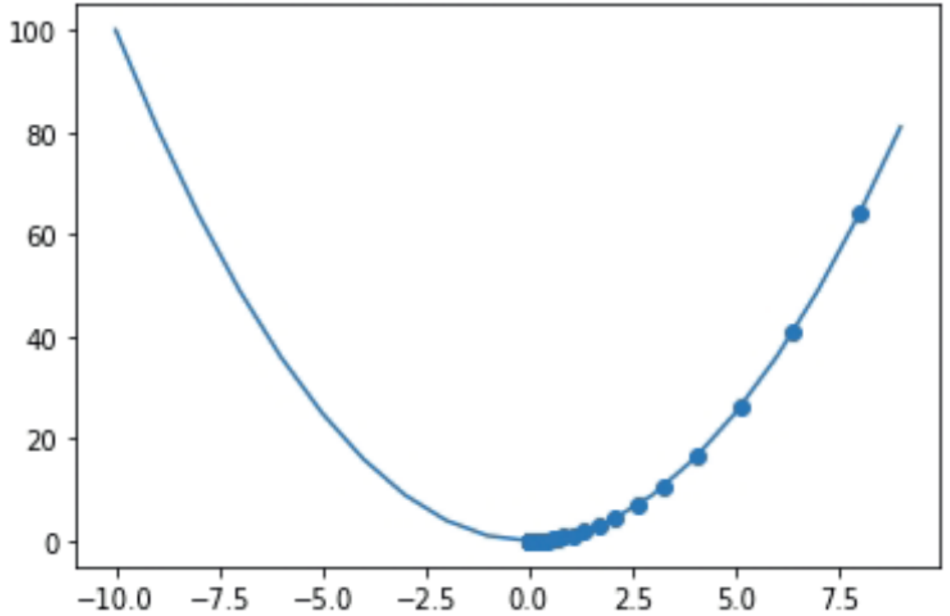
```
import numpy as np
import matplotlib.pyplot as plt

x= np.arange(-10,10,1)  # np.arrange(시작, 끝, 간격)
f_x = x ** 2

plt.plot(x, f_x)
plt.show()
```

Out [1]:



02 경사하강법으로 선형회귀 풀기

2. 경사하강법 알고리즘



In [2]:

```
x_new = 10
derivative = [ ]
y = [ ]
learng_rate= 0.1
for i in range(100):
    old_value = x_new
    derivative.append(old_value -
                learng_rate * 2 * old_value)
    x_new = old_value - learng_rate *2* old_value # Xnew=Xold-a*(2Xold)
    y.append(x_new ** 2)
plt.plot(x, f_x)
plt.scatter(derivative, y)
plt.show()
```

Out [2]:

02 경사하강법으로 선형회귀 풀기

2. 경사하강법 알고리즘

- 경사하강법에서 개발자가 결정해야 할 것
  - 학습률(learning rate)을 얼마로 할 것인가? α 값을 결정

$$x_{new} = x_{old} - \alpha \times (2x_{old})$$

    - 반복이 수행될 때마다 최솟값 변화
    - 값이 너무 작으면 충분히 많은 반복을 적용해도 원하는 최적값을 찾지 못하는 경우 발생
    - 값이 너무 크면 발산하여 최솟값 수렴 않거나 시간이 너무 오래 걸림
  - 얼마나 많은 반복(iteration)으로 돌릴 것인가?
    - 반복 횟수가 충분하지 않다면 최솟값을 찾지 못하는 경우 발생
    - 반복 횟수가 너무 많다면 필요 없는 시간을 허비할 수도 있음

02 경사하강법으로 선형회귀 풀기

2. 경사하강법 알고리즘

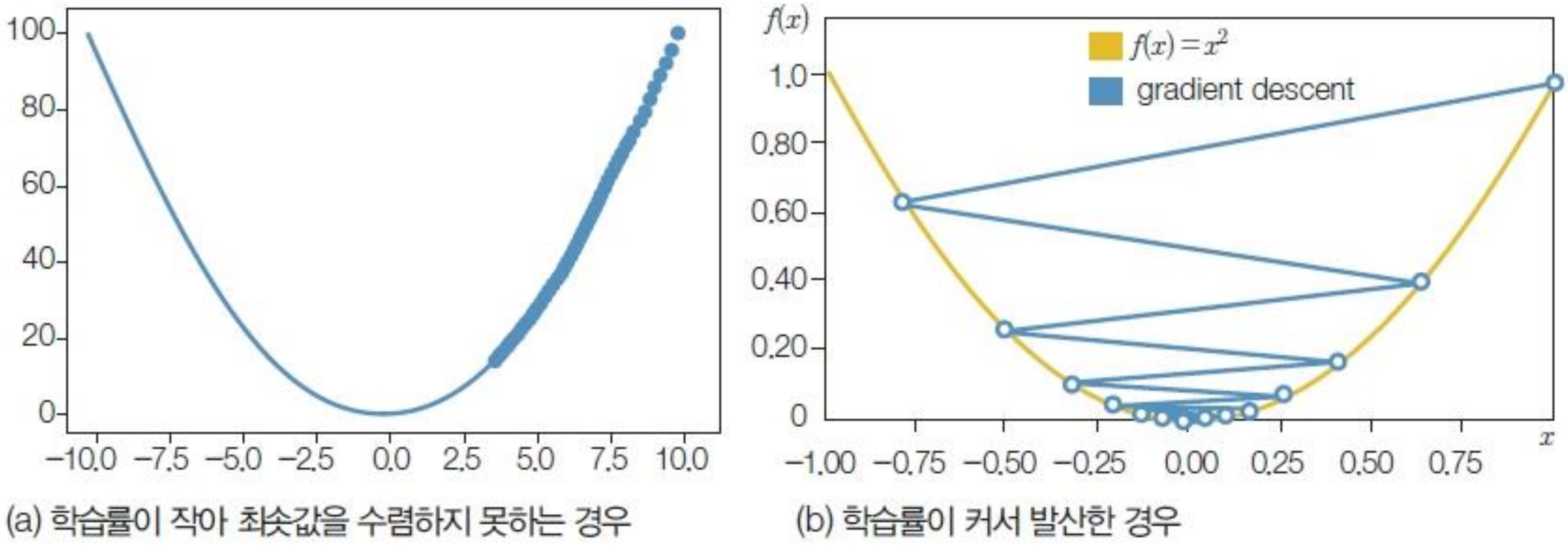


그림 7-7 학습률이 너무 크거나 작을 때 발생하는 문제

03

선형회귀 성능 측정하기

03 선형회귀 성능 측정하기

1. 훈련/테스트 분할

- 훈련/테스트 분할(train/test split) : 머신러닝에서 데이터를 학습을 하기 위한 학습 데이터셋(train dataset)과 학습의 결과로 생성된 모델의 성능을 평가하기 위한 테스트 데이터셋(test dataset)으로 나눔
- 모델이 새로운 데이터셋에도 일반화(generalize)하여 처리할 수 있는지를 확인하기 위해서

03 선형회귀 성능 측정하기

1. 훈련/테스트 분할

- 모델이 데이터에 과다적합(over-fit)된 경우 :
  생성된 모델이 특정 데이터에만 잘 맞아서 해당 데이터셋에 대해서는 성능을 발휘할 수 있지만 새로운 데이터셋에서는 전혀 성능을 낼 수 없다
- 모델이 데이터에 과소적합(under-fit)된 경우 :
  기존 학습 데이터를 제대로 예측하지 못함

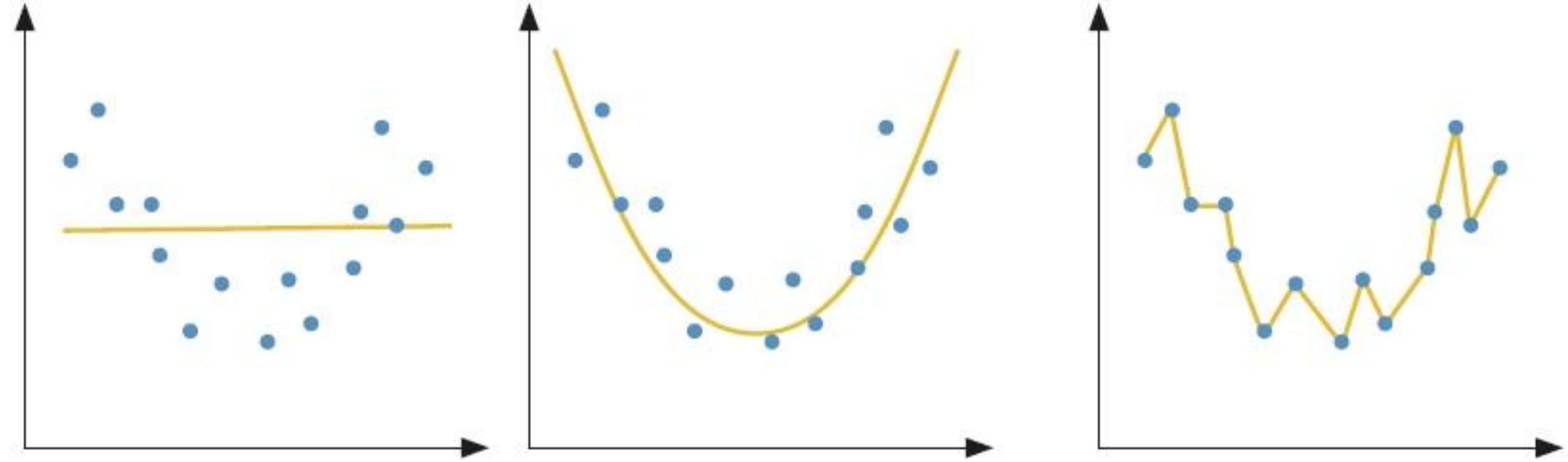


그림 7-10 다양한 데이터셋과 성능 파악

03 선형회귀 성능 측정하기

1. 훈련/테스트 분할

<과대적합(overfitting) 극복하기>

- 편향(bias) : 학습된 모델이 학습 데이터에 대해 만들어 낸 예측값과 실제값과의 차이
  - 모델의 결과가 얼마나 한쪽으로 쏠려 있는지 나타냄
  - 편향이 크면 학습이 잘 진행되기는 했지만 해당 데이터에만 잘 맞음
- 분산(variance) : 학습된 모델이 테스팅 데이터에 대해 만들어 낸 예측값과 실제값과의 차이
  - 모델의 결과가 얼마나 퍼져 있는지 나타냄

03 선형회귀 성능 측정하기

1. 훈련/테스트 분할

<과대적합(overfitting) 극복하기>

- 편향-분산 트레이드오프

(bias-variance trade-off) :

편향과 분산의 상충관계

[TIP] 과대적합(overfitting) : 높은 분산 낮은 편향 상태로 함수가 훈련 데이터셋에만 맞음. 피쳐의 개수를 줄이거나 정규화하여 해결

[TIP] 과소적합(underfitting) : 낮은 분산 높은 편향 상태로 함수가 훈련 데이터셋과 테스트 데이터셋에 모두 맞지 않음. 피쳐를 추가하여 해결

그림 8-4 편향–분산 트레이드오프

03 선형회귀 성능 측정하기

1. 훈련/테스트 분할

<과대적합(overfitting) 극복하기>

- 과대적합이 발생할 때 경사하강법 루프가 진행될수록 학습 데이터셋에 대한 비용함수의 값은 줄어들지만 테스트 데이터셋의 비용함수 값은 증가
  - 선형회귀 외에도 결정트리(decision tree)나 딥러닝처럼 연산에 루프가 필요한 모든 알고리즘에서 똑같이 발생

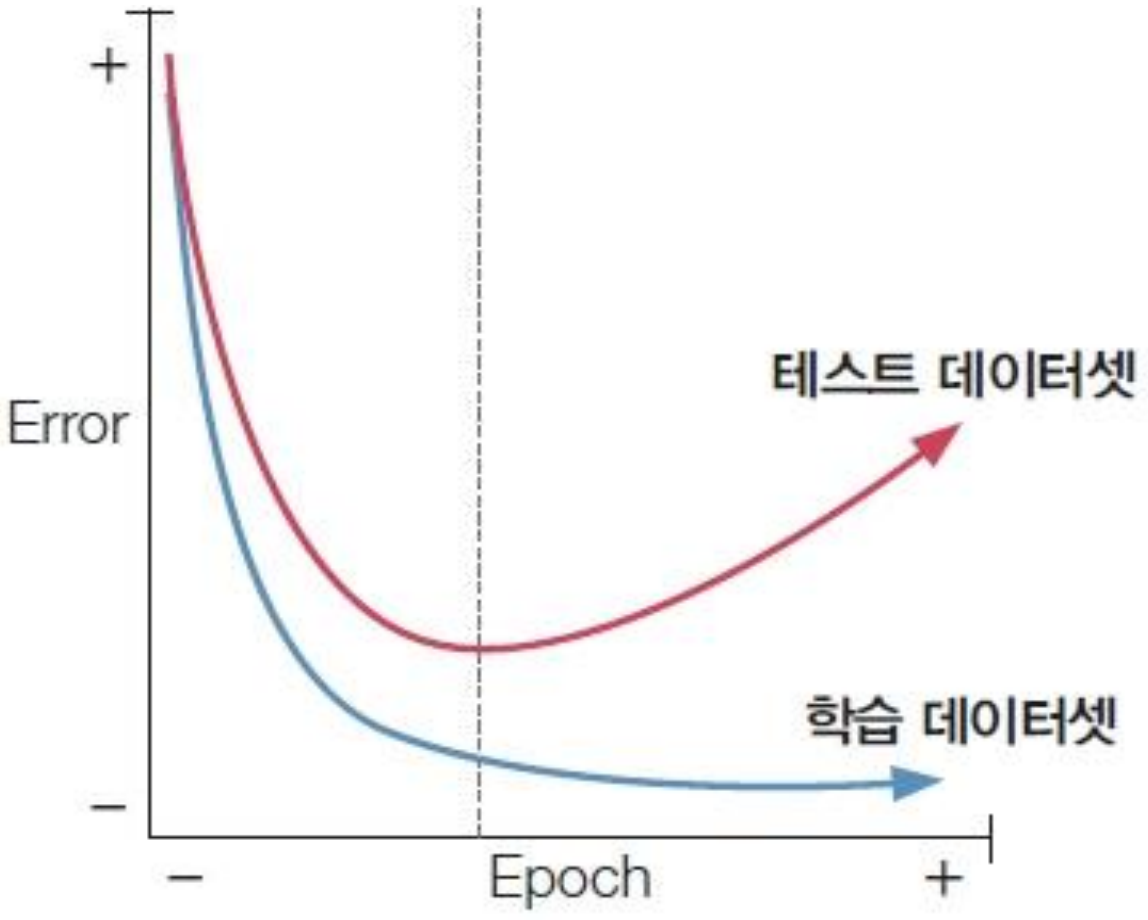


그림 8-5 과대적합이 발생할 때의 경사하강법

03 선형회귀 성능 측정하기

1. 훈련/테스트 분할

<과대적합(overfitting) 극복하기>

- 선형회귀에서 과대적합 해결책
  - **더 많은 데이터 활용하기** : 오류가 없고, 분포가 다양한 데이터를 많이 확보
  - **피쳐의 개수 줄이기** : 필요한 피쳐만 잘 찾아 사용
  - **적절한 매개변수 선정하기** : 확률적 경사하강법(Stochastic Gradient Decent, SGD, 학습용 데이터에서 샘플들을 랜덤하게 뽑아서 사용하는 방법) 의 학습률이나 루프의 횟수처럼 적절한 하이퍼 매개변수를 선정
  - **정규화 적용하기** : 데이터 편향성에 따라 필요 이상으로 증가한 피쳐의 가중치 값을 적절히 줄이는 규제 수식을 추가

03 선형회귀 성능 측정하기

1. 훈련/테스트 분할

- 홀드아웃 메서드(hold-out method) : 전체 데이터셋에서 일부를 학습 데이터와 테스트 데이터로 나누는 일반적인 데이터 분할 기법
    - 전체 데이터에서 랜덤하게 학습 데이터셋과 테스트 데이터셋을 나눔
    - 일반적으로 7:3 또는 8:2 정도의 비율

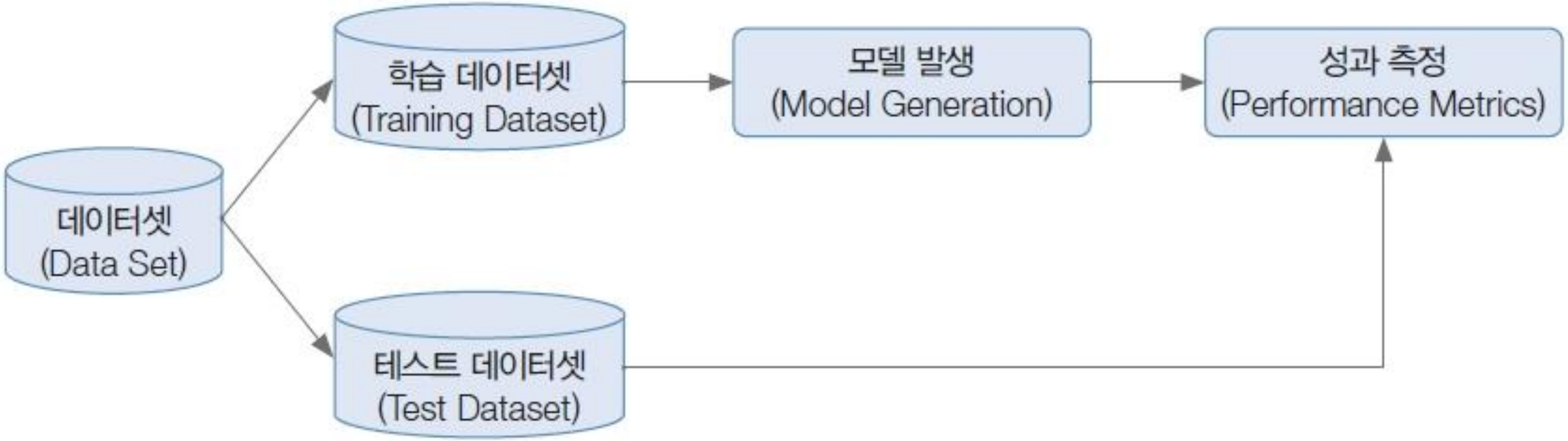


그림 7-11 홀드아웃 메서드(hold–out method) 기법

03 선형회귀 성능 측정하기

1. 훈련/테스트 분할

- sklearn 모듈이 제공하는 train_test_split 함수 사용
  - X와 y 벡터 값을 각각 대입
  - 매개변수 test_size에 테스트 데이터로 사용할 데이터의 비율을 지정
  - random_state는 랜덤한 값을 기준으로 임의로 지정하는 값

In [1]:
```
import numpy as np
from sklearn.model_selection import train_test_split

X, y = np.arange(10).reshape((5, 2)), range(5)

X_train, X_test, y_train, y_test = train_test_split(
 X, y, test_size=0.33, random_state=42)
```

03 선형회귀 성능 측정하기

2. 선형회귀의 성능 측정 지표

MAE, RMSE, 결정계수(R-squared)가 있음.

2.1 MAE

- MAE(Mean Absolute Error) : 평균 절대 잔차
- 모든 테스트 데이터에 대해 예측값과 실제값의 차이에 대해 절댓값을 구하고, 이 값을 모두 더한 후에 데이터의 개수만큼 나눈 결과

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| = \frac{1}{n}\sum_{i=1}^{n}|e_i|$$

- 직관적으로 예측값과 실측값의 차이를 알 수 있음

03 선형회귀 성능 측정하기

2. 선형회귀의 성능 측정 지표

2.1 MAE

- sklearn 모듈에서는 median_absolute_error 함수로 MAE를 구함

In [2]:	from sklearn.metrics import median_absolute_error y_true = [3, -0.5, 2, 7] y_pred = [2.5, 0.0, 2, 8] median_absolute_error(y_true, y_pred)
Out [2]:	0.5

03 선형회귀 성능 측정하기

2. 선형회귀의 성능 측정 지표

2.2 RMSE

- RMSE(Root Mean Squared Error) : 평균제곱근 오차
- 오차에 대해 제곱을 한 다음 모든 값을 더하여 평균을 낸 후 제곱근을 구함
- MAE에 비해 상대적으로 값의 차이가 더 큼
- 차이가 크게 나는 값에 대해서 페널티를 주고 싶다면 RMSE 값을 사용

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

2. 선형회귀의 성능 측정 지표

2.2 RMSE

- sklearn 모듈에서 RMSE를 직접적으로 지원하지는 않고 mean_squared_error만 지원

In [3]:	from sklearn.metrics import mean_squared_error y_true = [3, -0.5, 2, 7] y_pred = [2.5, 0.0, 2, 8] mean_squared_error(y_true, y_pred)
Out [3]:	0.375

2. 선형회귀의 성능 측정 지표

2.3 결정계수

- 결정계수(R-squared) : 두 개의 값의 증감이 얼마나 일관성을 가지는지 나타내는 지표
- **예측값이 크면 클수록 실제값도 커지고, 예측값이 작으면 실제값도 작아짐**
- 두 개의 모델 중 어떤 모델이 조금 더 상관성이 있는지를 나타낼 수 있지만, 값의 차이 정도가 얼마인지는 나타낼 수 없다는 한계가 있음

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \mu)^2}$$

2. 선형회귀의 성능 측정 지표

2.3 결정계수

- sklearn 모듈에서 r2_score 사용

In [3]:	from sklearn.metrics import r2_score y_true = [3, -0.5, 2, 7] y_pred = [2.5, 0.0, 2, 8] r2_score(y_true, y_pred)
Out [3]:	0.9486081370449679

04

코드로 선형회귀 구현하기

04 코드로 선형회귀 구현하기

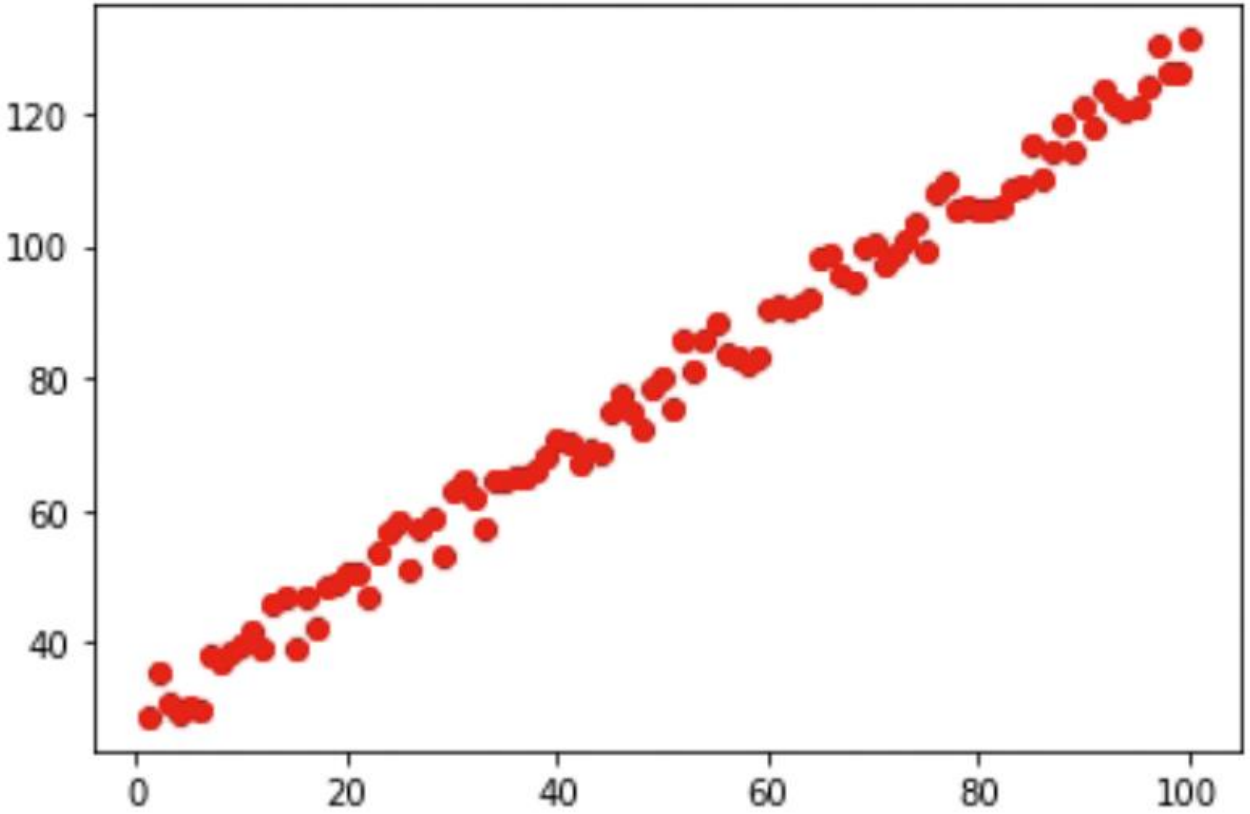
- 경사하강법을 선형회귀로 구현
  - 데이터 생성

In [1]:

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

def gen_data(numPoints, bias, variance):
    x = np.zeros(shape=(numPoints, 2))
    y = np.zeros(shape=numPoints)

    for i in range(0, numPoints):
        x[i][0] = 1    # (2) 100개의 데이터 x의 상수항에는 1
        x[i][1] = i    # (3) 100개의 데이터 x 값은 1씩 증가시킴
        y[i] = (i+bias) + random.uniform(0, 1) * variance # (4) 데이터 y에 bias 생성, bias 
를 더해서 균등분포로 0에서 1사이의 값으로 들어가게 함.
    return x, y
x, y = gen_data(100, 25, 10) # (1) 100개의 데이터 생성
```

04 코드로 선형회귀 구현하기

- 경사하강법을 선형회귀로 구현
  - 데이터 생성

In [1]:	`plt.plot(x[:,1]+1,y,"ro") # (5) 데이터 x와 y의 상관관계 그래프 작성` `plt.show()`
Out [1]:	

04 코드로 선형회귀 구현하기

- 경사하강법을 선형회귀로 구현
  - 생성된 데이터에 경사하강법 적용

In [2]:

```
def gradient_descent(x, y, theta, alpha, m, numIterations):
    xTrans = x.transpose() # (6) x값의 transpose 함수를 생성
    theta_list = [] # (7) theta 값의 저장 리스트를 생성
    cost_list = [] # (8) cost 값의 저장 리스트를 생성
    for i in range(0, numIterations): # (9) 반복 횟수만큼 반복(loop) 시작
        hypothesis = np.dot(x, theta) # (10) y_hat의 값 계산, 100개의 예측값 생성
        loss = hypothesis - y # (11) 예측값과 실제값 사이의 차를 loss에 저장함
        cost = np.sum(loss ** 2) / (2 * m) # (12) 비용함수의 값을 산출
        gradient = np.dot(xTrans, loss) / m # (13) gradient 계산
        theta = theta - alpha * gradient # (14) 가중치 값 theta 값 업데이트
        if i % 250 == 0: # (15) 매회 250번째마다 theta 값과 cost 값 업데이트 저장
            theta_list.append(theta)
            cost_list.append(cost)
    return theta,np.array(theta_list), cost_list # (16) 결과 리턴
```

04 코드로 선형회귀 구현하기

- 경사하강법을 선형회귀로 구현

In [2]:
```
m, n = np.shape(x) # (1)x의 데이터 개수에서 데이터개수 m, 피쳐 개수 n 추출
numIterations= 5000 # (2)반복횟수 지정
alpha = 0.0005 # (3)학습률(learning late) 지정
theta = np.ones(n) # (4)가중치(weight)값의 초깃값을 지정

theta,theta_list, cost_list = gradient_descent(x, y, theta, alpha, m, numIterations) # (5)경사하강 함수 호출
```

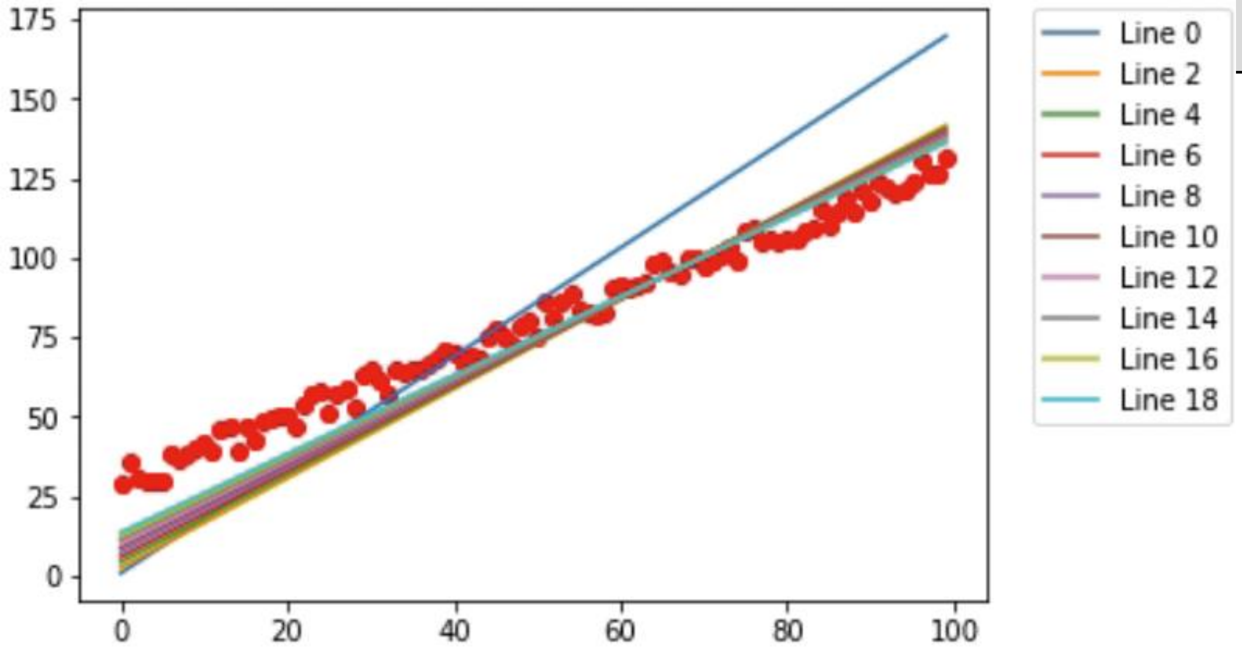
04 코드로 선형회귀 구현하기

- 경사하강법을 선형회귀로 구현

In [3]:

```
y_predict_step= np.dot(x, theta_list.transpose())
plt.plot(x[:,1],y,"ro")
for i in range (0,20,2):
    plt.plot(x[:,1],y_predict_step[:,i], label='Line %d'%i)

plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.show()
```

Out [3]:



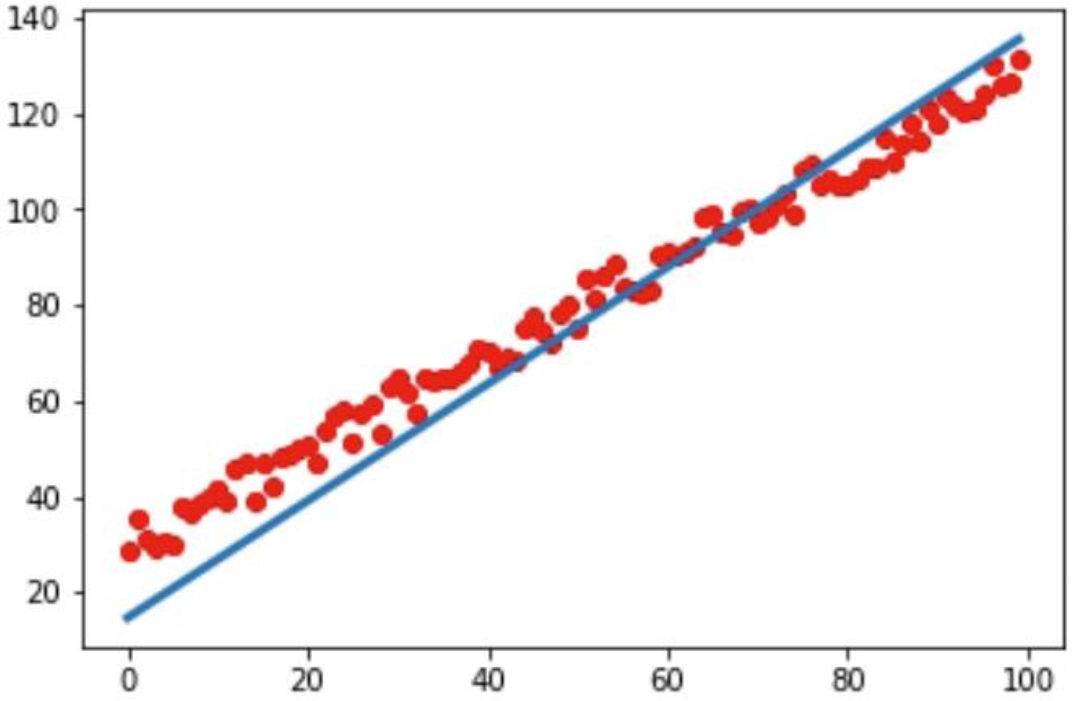
04 코드로 선형회귀 구현하기

- 경사하강법을 선형회귀로 구현: y_hat 값 그래프로 표현

In [4]:

```
y_predict= np.dot(x, theta)
plt.plot(x[:,1],y,"ro")
plt.plot(x[:,1],y_predict, lw=3) # lw: line width
plt.show()
```

Out [4]:



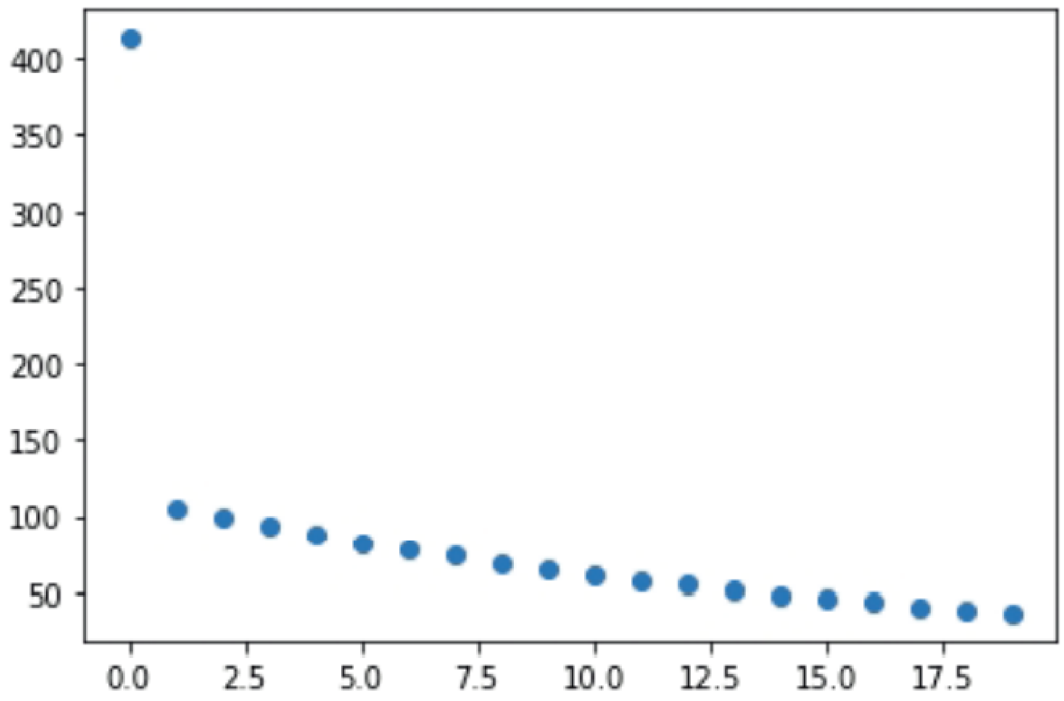
04 코드로 선형회귀 구현하기

- 경사하강법을 선형회귀로 구현:cost 값의 변화를 그래프로 표현

In [5]:

```
iterations = range(len(cost_list))

plt.scatter(iterations,cost_list)
plt.show()
```

Out [5]:



05

사이킷런을 이용한 선형회귀

05 사이킷런을 이용한 선형회귀

1. 사이킷런과 선형회귀 관련 함수

- 사이킷런(scikit-learn) : 대표적인 머신러닝 라이브러리

표 8-1 사이킷런의 선형회귀 관련 함수

함수명	설명	알고리즘
LinearRegression	가장 기본적인 선형회귀 알고리즘을 사용하며, SGD가 아닌 최소자승법으로 계산한다.	최소자승법
Lasso	L1 손실을 활용한 라쏘 알고리즘을 사용한다.	최소자승법
Ridge	L2 손실을 활용한 리지 알고리즘을 사용한다.	최소자승법
SGDRegressor	확률적 경사 하강법을 사용한 회귀 모델을 만든다. SGD에서 비용함수만을 변경하여 모든 함수를 지원하고 있어 필요한 하이퍼 매개변수를 설정해야 한다.	SGD

05 사이킷런을 이용한 선형회귀

2. 사이킷런을 활용하여 선형회귀 구현하기

- 'boston housing prices(보스턴 집값)' 데이터셋

x 변수 13개

[01] CRIM	자치시(town)별 1인당 범죄율
[02] ZN	25,000 평방피트를 초과하는 거주지역의 비율
[03] INDUS	비소매상업지역이 점유하고 있는 토지의 비율
[04] CHAS	찰스강에 대한 더미변수(강의 경계에 위치한 경우는 1, 아니면 0)
[05] NOX	10ppm 당 농축 일산화질소
[06] RM	주택 1가구당 평균 방의 개수
[07] AGE	1940년 이전에 건축된 소유 주택의 비율
[08] DIS	5개의 보스턴 직업센터까지의 접근성 지수
[09] RAD	방사형 도로까지의 접근성 지수
[10] TAX	10,000달러 당 재산세율
[11] PTRATIO	자치시(town)별 학생/교사 비율
[12] B	1000(Bk-0.63)^2, 여기서 Bk는 자치시별 흑인의 비율을 말함
[13] LSTAT	모집단의 하위 계층의 비율(%)
[14] MEDV	본인 소유의 주택 가격(중앙값) (단위 : $1,000)

y 변수

그림 8-8 boston housing prices(보스턴 집값) 데이터셋

05 사이킷런을 이용한 선형회귀

2. 사이킷런을 활용하여 선형회귀 구현하기

2.1 데이터 확보하기

- sklearn.datasets 라이브러리 load_boston 모듈을 사용하여 데이터를 추출
  - 딕셔너리 타입의 객체를 반환

In [1]:	from sklearn.datasets import load_boston import matplotlib.pyplot as plt import numpy as np boston = load_boston() boston.keys()
Out [1]:	dict_keys(['data', 'target', 'feature_names', 'DESCR', 'filename'])

```
dict_keys(['data', 'target', 'feature_names', 'DESCR', 'filename', 'data_module'])
```

data: x 데이터셋, target: y 데이터셋, feature_names: 모든 피쳐의 정보를 담고 있음, DESCR: 해당 피쳐에 대한 설명, Filename: 해당 데이터가 현재 컴퓨터에 저장된 위치

2. 사이킷런을 활용하여 선형회귀 구현하기

2.1 데이터 확보하기

- data 키 값 추출: 넘파이 객체 형태로 데이터가 출력됨

In [2]:	boston["data"]
Out [2]:	array([[6.3200e-03, 1.8000e+01, 2.3100e+00, ..., 1.5300e+01, 3.9690e+02, 4.9800e+00], [2.7310e-02, 0.0000e+00, 7.0700e+00, ..., 1.7800e+01, 3.9690e+02, 9.1400e+00], [2.7290e-02, 0.0000e+00, 7.0700e+00, ..., 1.7800e+01, 3.9283e+02, 4.0300e+00], ..., [6.0760e-02, 0.0000e+00, 1.1930e+01, ..., 2.1000e+01, 3.9690e+02, 5.6400e+00], [1.0959e-01, 0.0000e+00, 1.1930e+01, ..., 2.1000e+01, 3.9345e+02, 6.4800e+00], [4.7410e-02, 0.0000e+00, 1.1930e+01, ..., 2.1000e+01, 3.9690e+02, 7.8800e+00]])

2. 사이킷런을 활용하여 선형회귀 구현하기

2.1 데이터 확보하기

- x와 y 각 데이터셋을 추출
  - y_data는 n×1의 형태로 변환하기 위해 reshape를 적용

In [3]:	x_data = boston.data y_data = boston.target.reshape(boston.target.size,1) y_data.shape
Out [3]:	(506, 1)

2. 사이킷런을 활용하여 선형회귀 구현하기

2.2 데이터 전처리하기

- 피쳐 스케일링 적용: 피쳐 스케일링이 가능한 MinMaxScaler 객체 생성

In [4]:
```
from sklearn import preprocessing

minmax_scale = preprocessing.MinMaxScaler(feature_range=(0,5)).fit(x_data)
# (1) feature_range는 최대 최솟값을 지정하는 매개변수
x_scaled_data = minmax_scale.transform(x_data) # (2) 이미 만들어진 MinMaxScaler 클래스를 실제 데이터에 적용하여 스케일(scaled)된 데이터를 생성함, 0~5사이의 스케일된 데이터 출력값을 위하여

x_scaled_data[:3]
```

2. 사이킷런을 활용하여 선형회귀 구현하기

2.2 데이터 전처리하기

Out [4]:

```
array([[0.00000000e+00, 9.00000000e-01, 3.39076246e-01,
        0.00000000e+00, 1.57407407e+00, 2.88752635e+00,
        3.20803296e+00, 1.34601570e+00, 0.00000000e+00,
        1.04007634e+00, 1.43617021e+00, 5.00000000e+00,
        4.48399558e-01],
       [1.17961270e-03, 0.00000000e+00, 1.21151026e+00,
        0.00000000e+00, 8.64197531e-01, 2.73998850e+00,
        3.91349125e+00, 1.74480990e+00, 2.17391304e-01,
        5.24809160e-01, 2.76595745e+00, 5.00000000e+00,
        1.02235099e+00],
       [1.17848872e-03, 0.00000000e+00, 1.21151026e+00,
        0.00000000e+00, 8.64197531e-01, 3.47192949e+00,
        2.99691040e+00, 1.74480990e+00, 2.17391304e-01,
        5.24809160e-01, 2.76595745e+00, 4.94868627e+00,
        3.17328918e-01]])
```

2. 사이킷런을 활용하여 선형회귀 구현하기

2.3 데이터 분류하기

- 데이터를 훈련과 테스트 형태로 분류

In [5]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(x_scaled_data,
y_data, test_size=0.33)
# X 데이터의 학습 데이터셋, X 데이터의 테스트 데이터셋
# Y 데이터의 학습 데이터셋, Y 데이터의 테스트 데이터셋

X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

Out [5]:
```
((339, 13), (167, 13), (339, 1), (167, 1))
```

2. 사이킷런을 활용하여 선형회귀 구현하기

2.4 데이터 학습하기

- 학습에 사용할 알고리즘 해당하는 모델의 클래스 호출
  - 각 클래스의 매개변수를 이해해야 함
- 공통적으로 사용하는 매개변수
  - fit_intercept : 절편을 사용할지 말지를 선택
  - normalize : 학습할 때 값들을 정규화할지 말지
  - copy_X : 학습 시 데이터를 복사한 후 학습을 할지 결정
  - n_jobs : 연산을 위해 몇 개의 CPU를 사용할지 결정

In [6]:
```
from sklearn import linear_model
regr = linear_model.LinearRegression(
        fit_intercept=True, normalize=False, copy_X=True, n_jobs=8)
```

2. 사이킷런을 활용하여 선형회귀 구현하기

2.4 데이터 학습하기

- 사이킷런은 ‘적합-예측(fit-predict)’ 또는 ‘적합-변형(fit-transform)’의 구조
  - 모델을 생성한 후 예측을 하거나 전처리 모델의 규칙을 세운 후 데이터 전처리를 적용하는 구조

In [7]:	regr.fit(X_train, y_train)
Out [7]:	LinearRegression(n_jobs=8)

```
LinearRegression(n_jobs=8, normalize=False)
```

05 사이킷런을 이용한 선형회귀

2. 사이킷런을 활용하여 선형회귀 구현하기

2.4 데이터 학습하기

- 사이킷런은 ‘적합-예측(fit-predict)’ 또는 ‘적합-변형(fit-transform)’의 구조

In [8]:	`print('Coefficients: ', regr.coef_)` `print('intercept: ', regr.intercept_)`
Out [8]:	Coefficients: [[-2.86129759 0.27632862 0.10322333 0.33532791 -1.89104482 3.55479622 -0.04952964 -3.01015804 1.22330686 -1.00916771 -1.98466467 0.62386235 -3.9996908]] intercept: [29.42877381]

```
Coefficients:  [[-1.96690438  1.10629704 -0.20930004  0.32014968 -1.2959773   3.84127845
  -0.62377726 -3.73111241  1.44866999 -1.48041408 -1.72079525  0.7601843
  -2.88837323]]
intercept:  [27.97117752]
```

05 사이킷런을 이용한 선형회귀

2. 사이킷런을 활용하여 선형회귀 구현하기

2.5 예측하기와 결과 분석하기

- 만들어진 함수로 실제 예측을 한다

In [9]:	**regr.predict(x_data[:5])**	
Out [9, 10]:	**array([[-58.72562452], [-32.88598964], [-11.38914021], [10.01207448], [1.87764423]])**	array([[-159.73176275], [-127.92969501], [-102.22604655], [-62.43387916], [-72.3509974]])

- regr 대신 수식을 그대로 재현해도 같은 결과가 출력됨

In [10]:	**x_data[:5].dot(regr.coef_.T) + regr.intercept_**

2. 사이킷런을 활용하여 선형회귀 구현하기

2.5 예측하기와 결과 분석하기

- 사이킷런에서 지표들(metrics)을 호출하여 성능을 비교

In [11]:
```
from sklearn.metrics import r2_score
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error

y_true = y_test.copy()
y_hat = regr.predict(X_test)

r2_score(y_true, y_hat), mean_absolute_error(y_true, y_hat),
mean_squared_error(y_true, y_hat)
```

Out [11]:
```
(0.7012192205071575, 3.6874625281998266, 28.869826251555843)
```

(0.7008643556860734, 3.4336470300364486, 29.214428580679424)

2. 사이킷런을 활용하여 선형회귀 구현하기

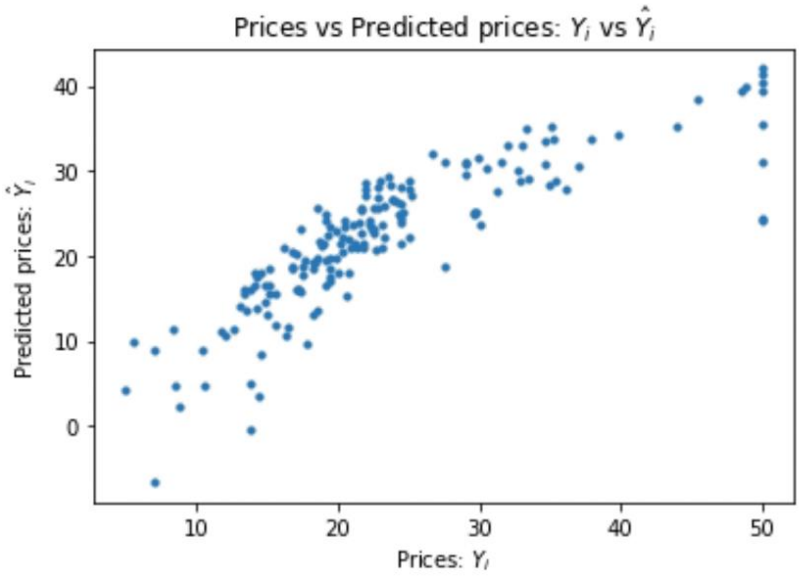
2.5 예측하기와 결과 분석하기

- 필요에 따라 시각화 도구로 예측값과 실제값 비교

In [12]:

```
plt.scatter(y_true, y_hat, s=10)
plt.xlabel("Prices: $Y_i$")
plt.ylabel("Predicted prices: $\hat{Y}_i$")
plt.title("Prices vs Predicted prices: $Y_i$ vs $\hat{Y}_i$")
```

Out [12]:



06
선형회귀의 응용

인간-로봇의 상호작용을 위한 JAFFE와 CK+ 데이터셋에 기반한 연속 정서 추정 시스템 개발

제안한 연속 정서 추정 시스템

- 3종류의 데이터셋

213	577	790
JAFFE 데이터셋	CK+ 데이터셋	JAFFE & CK+ 데이터셋

- 4종류의 이미지들

- 이미지 전처리 방법
  - 얼굴 추적
  - 고유얼굴
  - 주성분 분석
- 추정 방법
  - 선형 회귀 분석

응용분야

- 감성 로봇(서비스 로봇)

- 감정 노동 분야(콜센터)

- 자동차 분야(졸음 운전)

인간-로봇의 상호작용을 위한 JAFFE와 CK+ 데이터셋에 기반한 연속 정서 추정 시스템 개발

인간에 의한 정답 정서 값 수집

정답 정서 값

- JAFFE 데이터셋에 대한 정서 구성(213장)
  - 중립: 30장, 행복: 31장, 슬픔: 31장, 놀람: 30장, 분노: 30장, 역겨움: 29장, 공포: 32장
- 한 점(point): 55명의 참가자들이 측정한 정서 값들의 평균값
  - 중립이 (0, 0)인 경우에 나머지 6종류의 기본 정서들도 중립으로부터 떨어져 있음

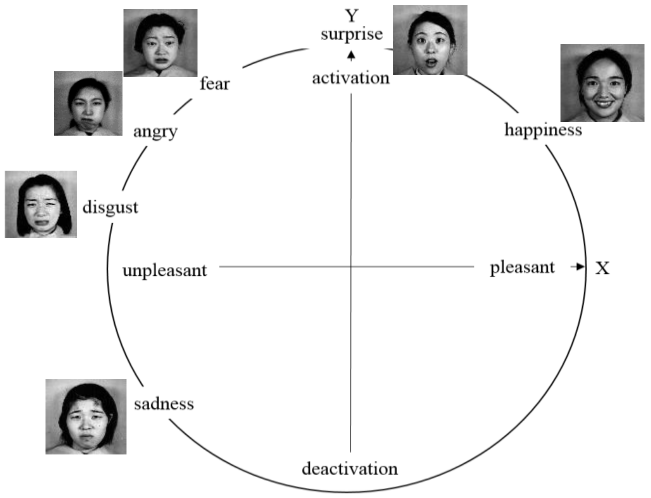


그림 1. Russell의 2차원 정서 평면 (X축: 쾌/불쾌, Y축: 각성/비각성)

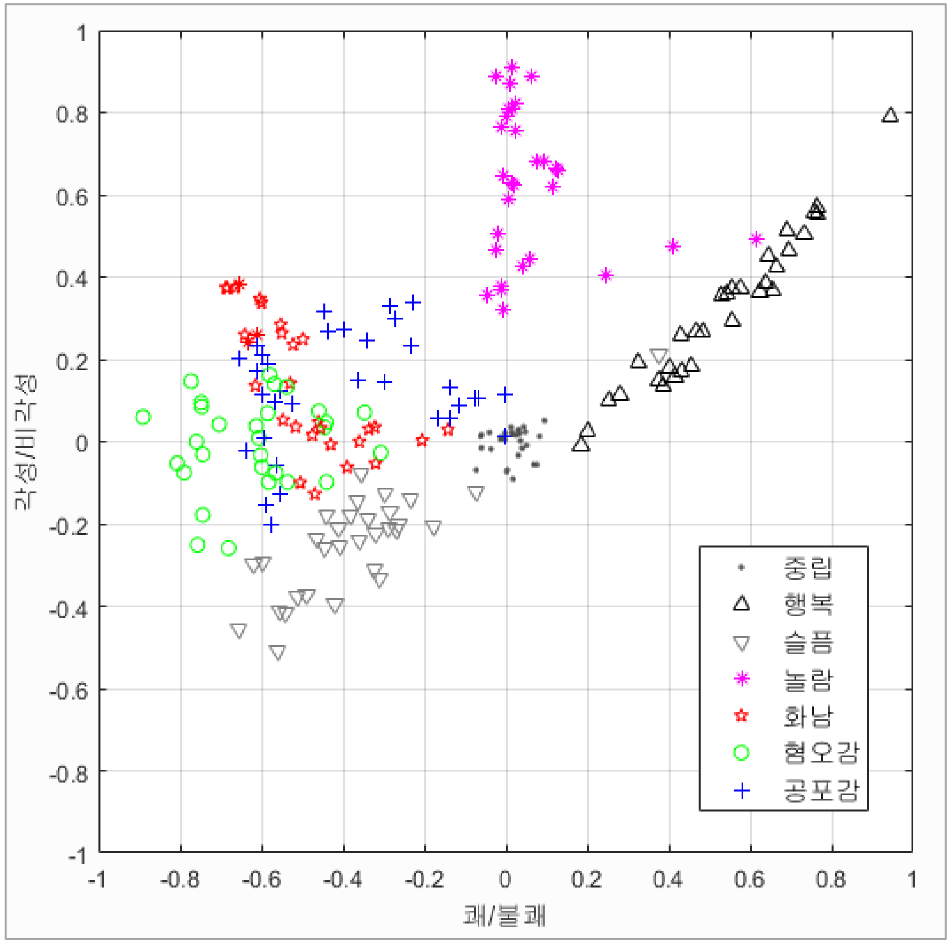


그림 2. JAFFE 데이터셋의 얼굴 표정 이미지들에 대하여 참가자들에 의해 측정된 7종류의 정답 정서 값

연속 정서 추정 시스템 인간-로봇 적용

인간-로봇 상호작용 적용 실험

실험 환경

- 정서 추정 시스템(훈련 및 테스트)
  - 우분투(ubuntu) 16.04 LTS 환경의 매틀랩 R2017a 프로그램
- 인간의 얼굴 표정 촬영
  - 나오 1.14.5의 이마쪽 카메라(OV7670 카메라)
- 촬영이미지 및 음성 제어
  - 우분투 16.04 LST 환경의 파이썬(python) 2.7 프로그램
- 나오의 동작 제어
  - 나오-큐아이(Nao-qi)

- 정서 추정을 위해 사용한 방법: 회귀 분석 방법
  - 훈련: JAFFE 데이터셋 213장, 얼굴 전체 이미지
  - 테스트: 나오 로봇이 촬영한 1장, 얼굴 전체 이미지

- 나오가 추정한 정서
  - 그림 2의 JAFFE 데이터셋의 정답 정서 값 기반

인간(음성): 안녕.
로봇(음성): 안녕.
로봇(행동): 오른쪽 팔을 흔들며 인사한 후 다시 차렷 자세를 한다.
로봇(음성): 잘 지내니?
로봇(행동): 로봇 이마의 카메라를 사용하여 인간의 얼굴 표정을 촬영한다.
인간(음성): 잘 지내니? *제안한 정서 추정 시스템으로 정서 추정*
나오, 내 정서가 어떤 것 같니?
로봇(음성): 쾌/불쾌는 0.2451이야.
각성/비각성은 0.2855야.
그래서, 너는 행복해 보여.

그림 3. 인간-로봇 상호작용에 대한 시나리오

연속 정서 추정 시스템 인간-로봇 적용

인간-로봇 상호작용 적용 실험

실험 환경

- 추정 결과(평균 제곱근 오차)
  - 중립 → 쾌/불쾌: –0.0644, 각성/비각성: 0.1111, 1차원
  - 놀람 → 쾌/불쾌: –0.1805, 각성/비각성: 0.2111, 22차원
  - 화남 → 쾌/불쾌: –0.2564, 각성/비각성: 0.2155, 33차원
  - 행복 → 쾌/불쾌: 0.2451, 각성/비각성: 0.2855, 200차원
- 동영상 촬영
  - 정면: 삼성 갤럭시 S10+ 내장 카메라(후면 카메라: 1200만 화소, F/1.5~2.4),
  - 옆면: 삼성 갤럭시 와이드 4 내장 카메라(전면 카메라: 800만 화소, F/2.0)

(ㄱ) 중립 (ㄴ) 놀람 (ㄷ) 화남 (ㄹ) 행복

그림 4. 촬영한 이미지들에 대한 정서 추정 결과들

그림 5. 인간-로봇 상호작용에 대한 동영상 (https://www.youtube.com/watch?v=Too8I7tVyTE)

Assignment

Assignment

- 강의 PPT 47~58쪽 사이의 코드 In [1] ~ In [12]의 코드를 실행시킨 후 각 결과를 화면 캡쳐하여 제출하시오. (한글, 워드, PPT 등 이용 가능)

Thank You !