# 4장
# 모델훈련(Training Models)

- 모델 훈련과정 이해 ➔ 적절한 모델 선택, 훈련 알고리즘 선택, 하이퍼파라메터 선정에 도움
- 신경망에서도 같은 기술 사용 ➔ 신경망 공부에 도움
- 선형회귀에서의 다양한 이슈, 다항회귀, 모델이 과대적합 되는지 감지하는 방법 배우게 됨
- 모델에 규제를 넣는 방법
- Logistic Regression, Softmax

# 선형회귀 (Linear Regression)

**삶의 만족도** $= \theta_0 + \theta_1 \times$ **1인당 GDP**

일반식

예측값

$$\hat{y} = \theta_0 + \theta_1 \times x_1 + \theta_2 \times x_2 + \cdots + \theta_n \times x_n$$

$n$ : 특성의 개수

절편 또는 편향

2번째 모델 파라미터

2번째 특성

벡터식

$$\hat{y} = h_\theta(\mathbf{x}) = \theta^T \cdot \mathbf{x}$$

가설

세타 벡터의 전치

점곱 표현을 위한 트릭 $x_0 = 1$

$$[\theta_0 \quad \theta_1 \quad \theta_2 \quad \cdots \quad \theta_n] \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

- predicted value
- bias
- j-th model parameter
- i-th feature value
- parameter vector
- feature vector
- hypothesis function, using the model parameter Θ

선형 회귀 : # 학습(훈련)

- 학습 : 성능측정지표(performance measure)를 최소화(혹은 최대화)하는 모델 파라메터를 계산
- 회귀에 가장 많이 사용되는 측정지표는 평균제곱근오차(RMSE) ➔ 제곱근오차(MSE)도 같은 결과

**평균 제곱 오차(RMSE에서 제곱근을 뺀 것)**

$$\mathbf{MSE}(\mathbf{X}, h_\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(\mathbf{x}^{(i)}) - y^{(i)}\right)^2 = \frac{1}{m}\sum_{i=1}^{m}\left(\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)}\right)^2$$

샘플 개수

타깃 or 레이블

- $\mathbf{MSE}(\mathbf{X}, h_\theta)$ ➔ $\mathrm{MSE}(\theta)$로 줄여서 써도 됨
- 학습 : $\mathrm{MSE}(\theta)$ 를 최소화하는 θ 계산하기. $\hat{\theta} = \arg\min_{\theta} MSE(\theta)$
  - 직접법 : 정규방정식(Normal Equation)
  - 반복법 : 경사하강법(Gradient Descent)

# 선형 회귀 : 정규 방정식(Normal Equation)

- Normal Equation

**증명: https://goo.gl/WkNEXH**

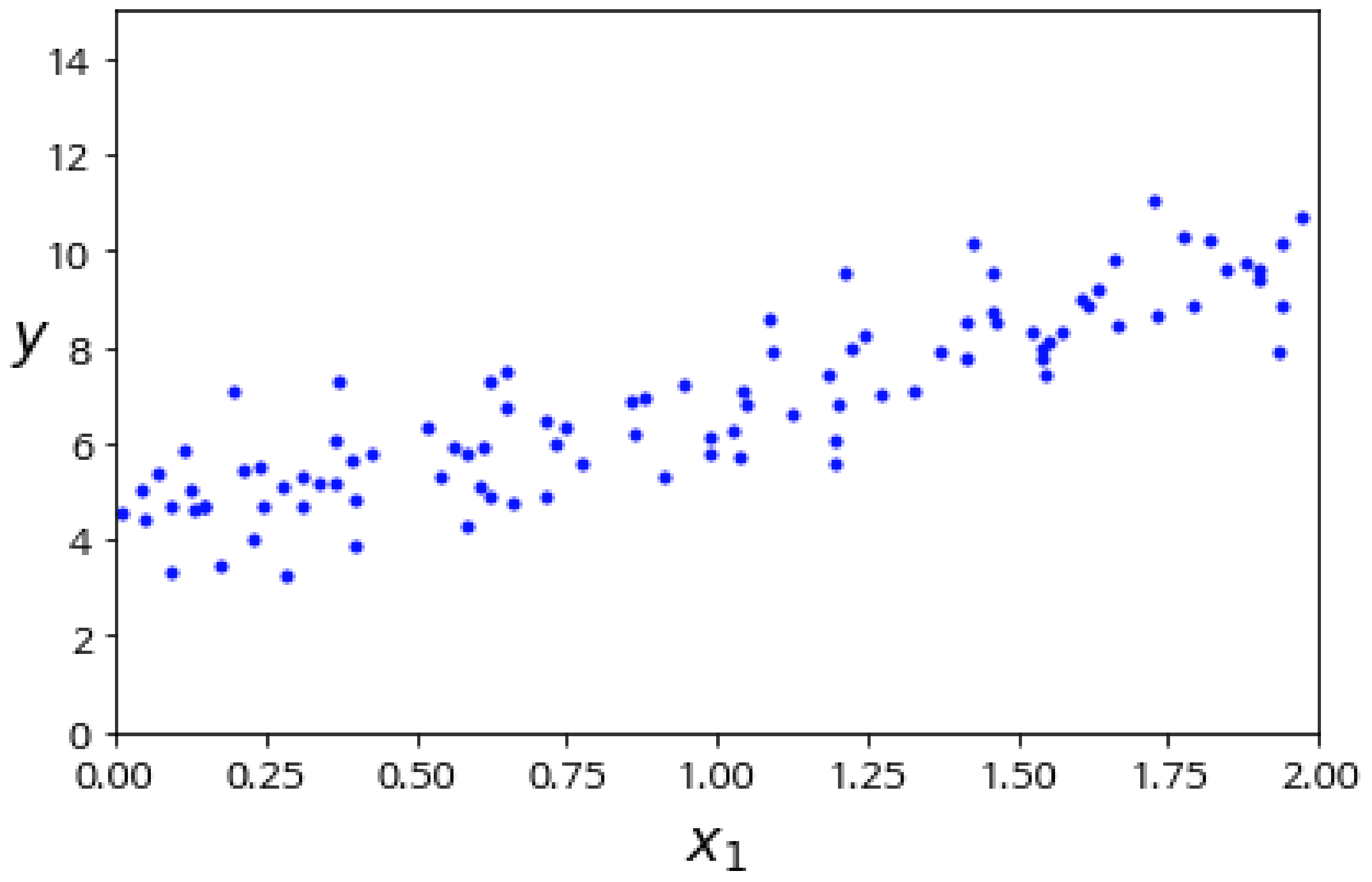
$$\hat{\theta} = (\mathbf{X}^T \cdot \mathbf{X})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

$$\mathbf{X} = \begin{bmatrix} (\mathbf{x}^{(1)})^T \\ (\mathbf{x}^{(2)})^T \\ \vdots \\ (\mathbf{x}^{(1999)})^T \\ (\mathbf{x}^{(20000)})^T \end{bmatrix} = \begin{bmatrix} -118.29 & 33.91 & 1{,}416 & 38{,}372 \\ & \cdots & & \\ & & & \\ & & & \end{bmatrix}$$

- 샘플 데이터 : $y = 4 + 3x +$ **가우시안 노이즈**

```
import numpy as np

X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

**(100, 1)**



$X\_train = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$ y= $\begin{bmatrix} 1 \\ 3 \end{bmatrix}$ 학습데이터

(2,1), (3,3) 점을 지나는 직선

$\theta_0 \; x_0 + \theta_1 \, x_1 =$ y
$(x_0 == 1)$

$\theta_0 + 2\theta_1 = 1$
$\theta_0 + 3\theta_1 = 3$

$X = \begin{bmatrix} 1 & 2 \\ 1 & 3 \end{bmatrix}$ y= $\begin{bmatrix} 1 \\ 3 \end{bmatrix}$

$X^T X = \begin{bmatrix} 1 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 1 & 3 \end{bmatrix} = \begin{bmatrix} 2 & 5 \\ 5 & 13 \end{bmatrix}$

$[X^T X]^{-1} = \begin{bmatrix} 2 & 5 \\ 5 & 13 \end{bmatrix}^{-1} = \frac{1}{|26-25|} \begin{bmatrix} 13 & -5 \\ -5 & 2 \end{bmatrix}$

$[X^T X]^{-1} X^T = \begin{bmatrix} 13 & -5 \\ -5 & 2 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 2 & 3 \end{bmatrix} = \begin{bmatrix} 3 & -2 \\ -1 & 1 \end{bmatrix}$

$[X^T X]^{-1} X^T$y = $\begin{bmatrix} 3 & -2 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \end{bmatrix} = \begin{bmatrix} -3 \\ 2 \end{bmatrix}$

➔ $\theta_0$ = -3, $\theta_1$=2

**학습데이터 개수가 더 많아지면 ?**
**유일한 해? 근사해?**

# 정규방정식: Numpy의 linalg.inv() 이용

(100, 2)

(100, 1)

np.hstack((np.ones((100, 1)), X))

```
X_b = np.c_[np.ones((100, 1)), X]  # 모든 샘플에 x0 = 1을 추가합니다.
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

$$\hat{\theta} = (\mathbf{X}^T \cdot \mathbf{X})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

이렇게 해도 됨

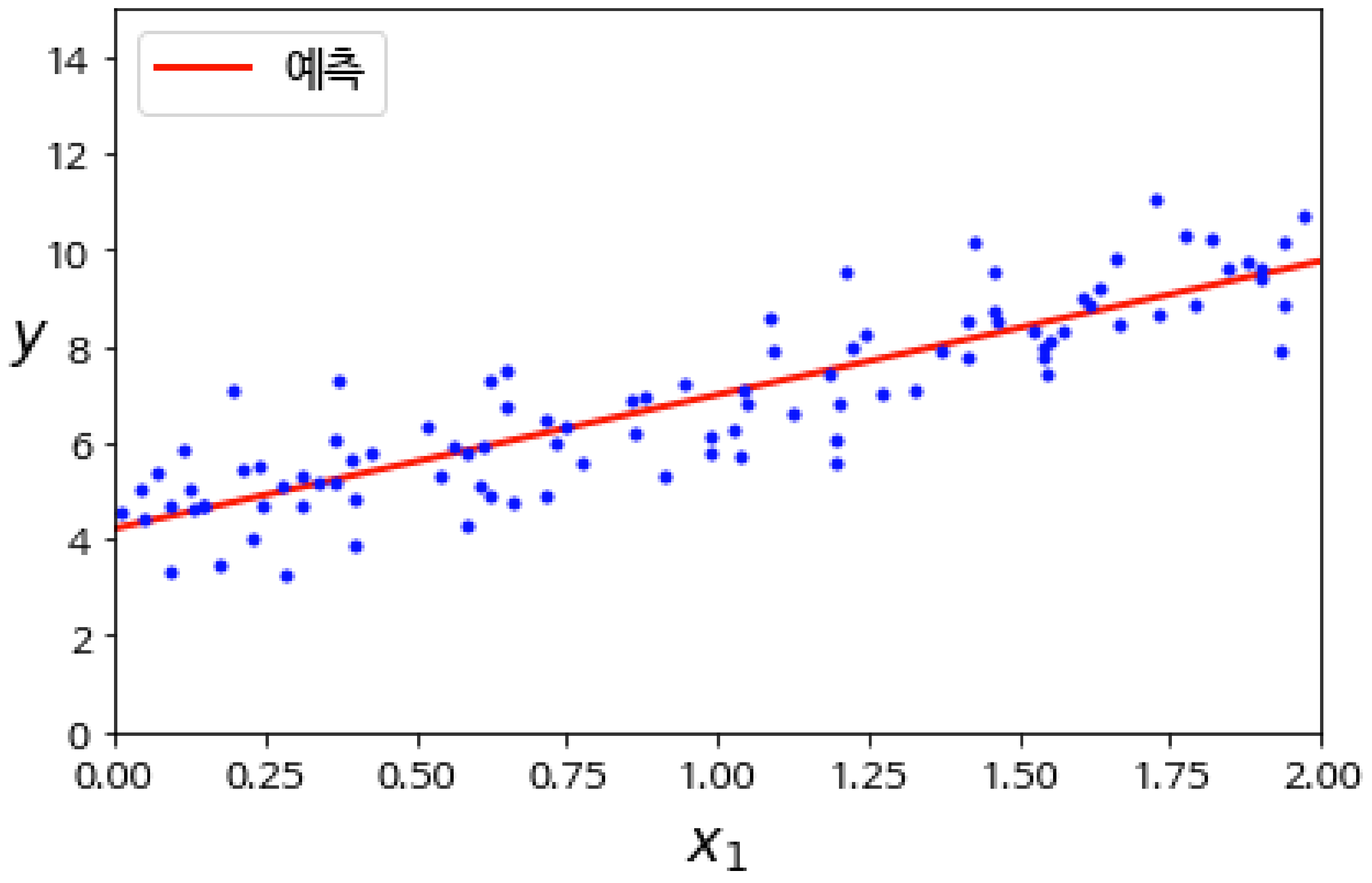
```
np.linalg.pinv(X_b).dot(y)
```

```
array([[4.21509616],
       [2.77011339]])
```

편향

```
array([[4.21509616],
       [2.77011339]])
```

기울기



# 정규방정식: sklearn의 LinearRegression이용

```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X, y)
lin_reg.intercept_, lin_reg.coef_
```

```
(array([4.21509616]), array([[2.77011339]]))
```

편향

기울기

```
np.sum(np.square(lin_reg.predict(X) - y))
```

오차의 제곱 합

```
80.6584563967053
```

```
lin_reg.score(X, y)
```

$R^2$

```
0.7692735413614223
```

# 정규 방정식 : 계산 복잡도

- 역행렬(inverse matrix) 계산이 정규 방정식의 복잡도를 좌우함

**(n+1, m) x (m, n+1) = (n+1, n+1)**

$$(\mathbf{X}^T \cdot \mathbf{X})^{-1}$$

$$O(n^{2.4}) \sim O(n^3)$$

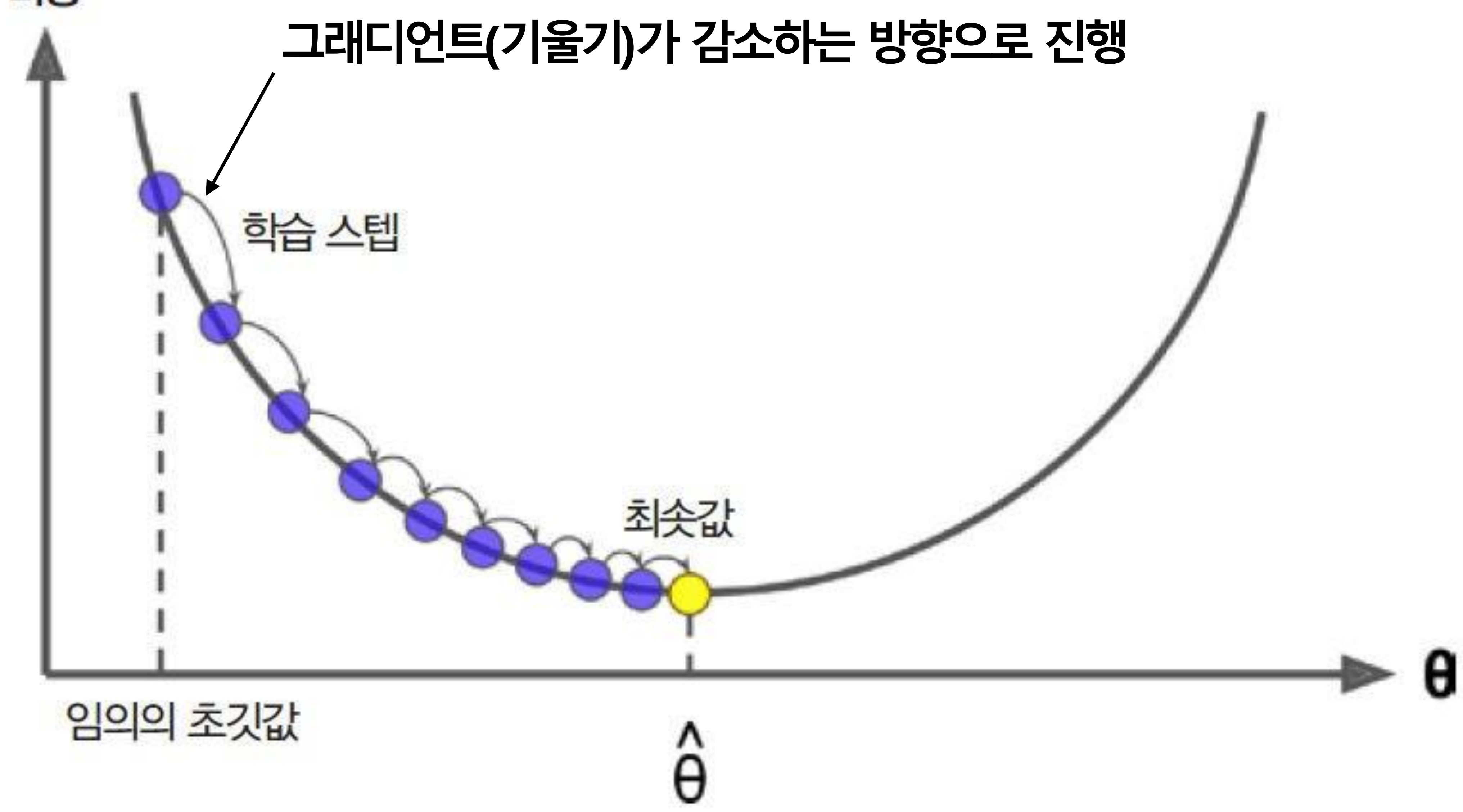
**샘플의 수에 대해서는 선형적으로 늘어남**

**특성의 수가 2배로 늘어 나면 계산 복잡도는 5~8배로 늘어남**

*** scipy의 lstsq 함수는 SVD 방법을 사용하며 $O(n^2)$의 복잡도를 가짐**

# 경사 하강법 (Gradient Descent)

- 모델 파라미터를 조금씩 수정하면서 비용 함수의 최소값을 찾는 방법



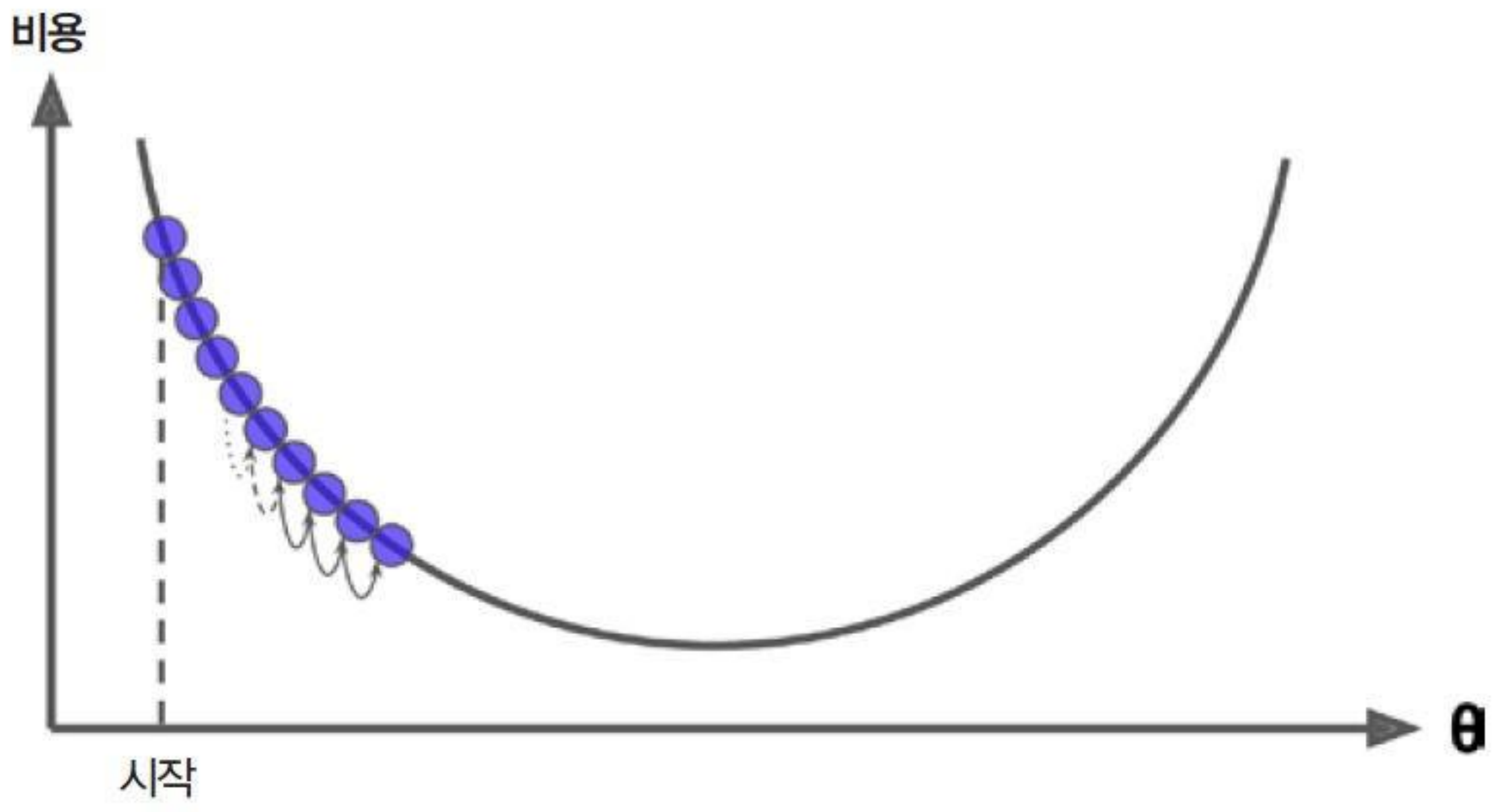

학습률 (Learning Rate)

$$\theta^{\{next\ step\}} = \theta - \eta \nabla_{\theta}$$

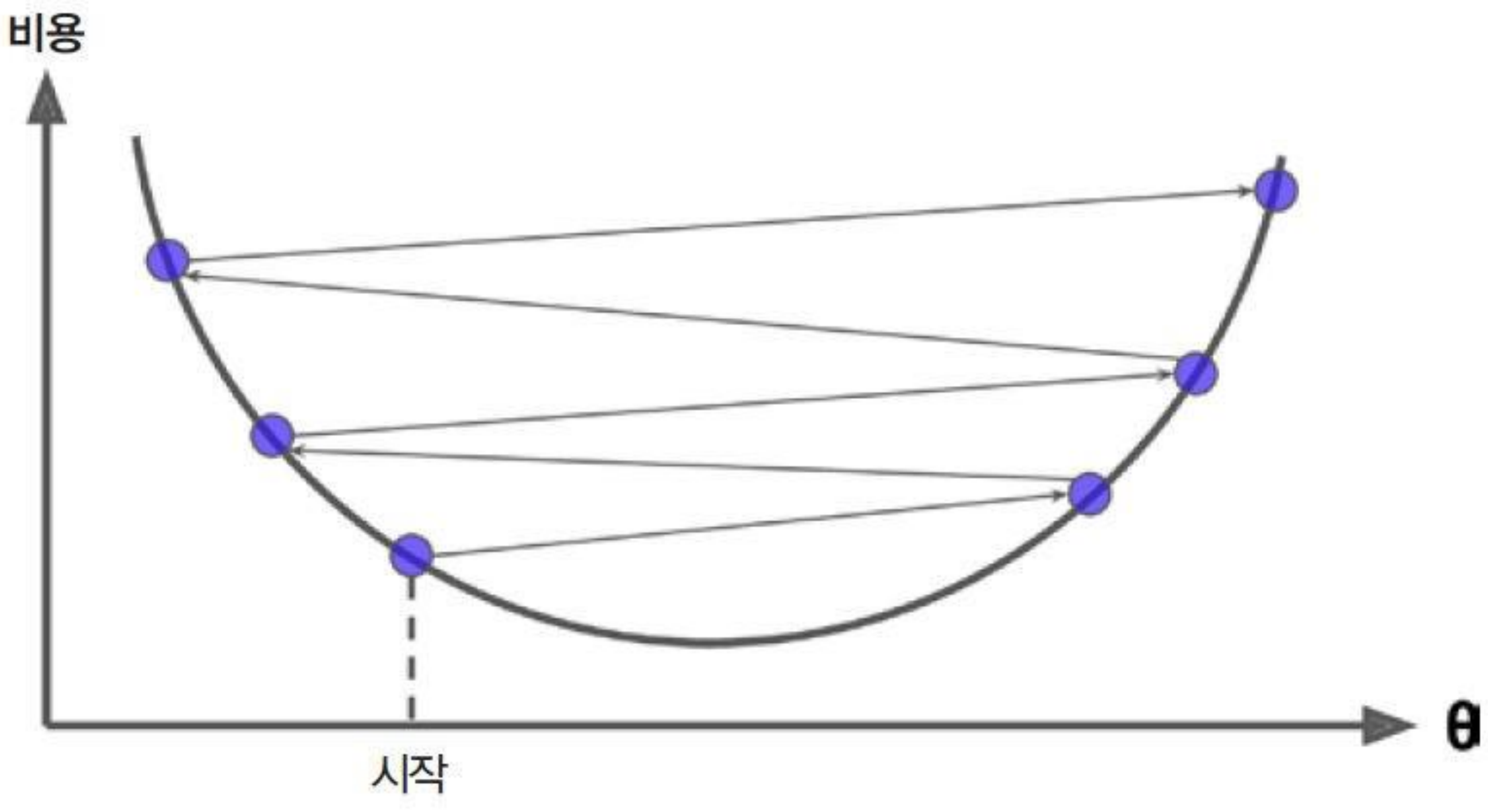
그래디언트: θ에 대한 비용함수의 미분값

# 경사하강법 : 학습율(Learning Rate)

- 학습율 : 그래디언트 적용량을 조정하는 하이퍼파라미터임

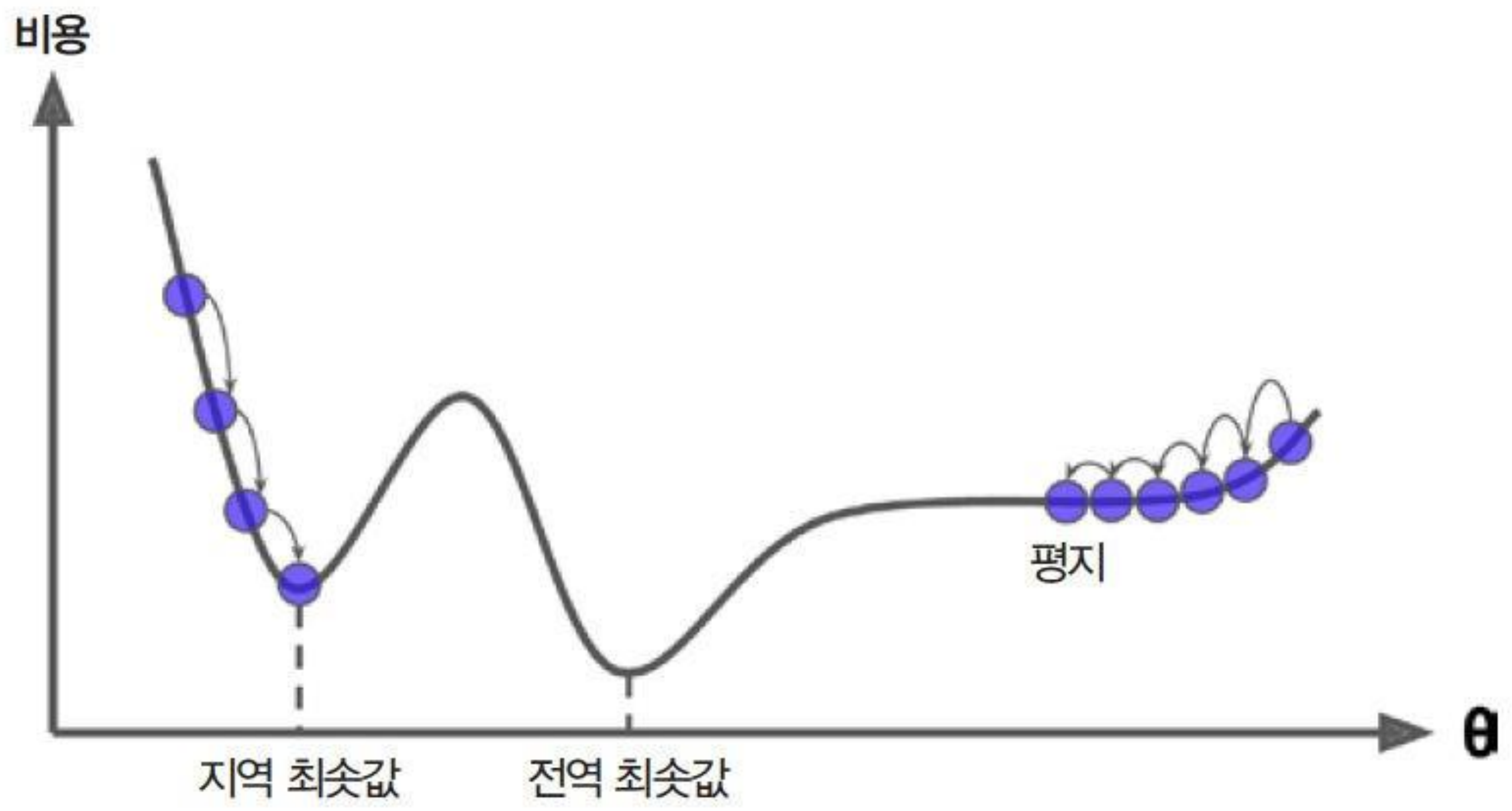



**학습률이 너무 작을 때**




**학습률이 너무 클 때**

# 경사하강법 : 문제점 – 전역 최소값 찾지 못함

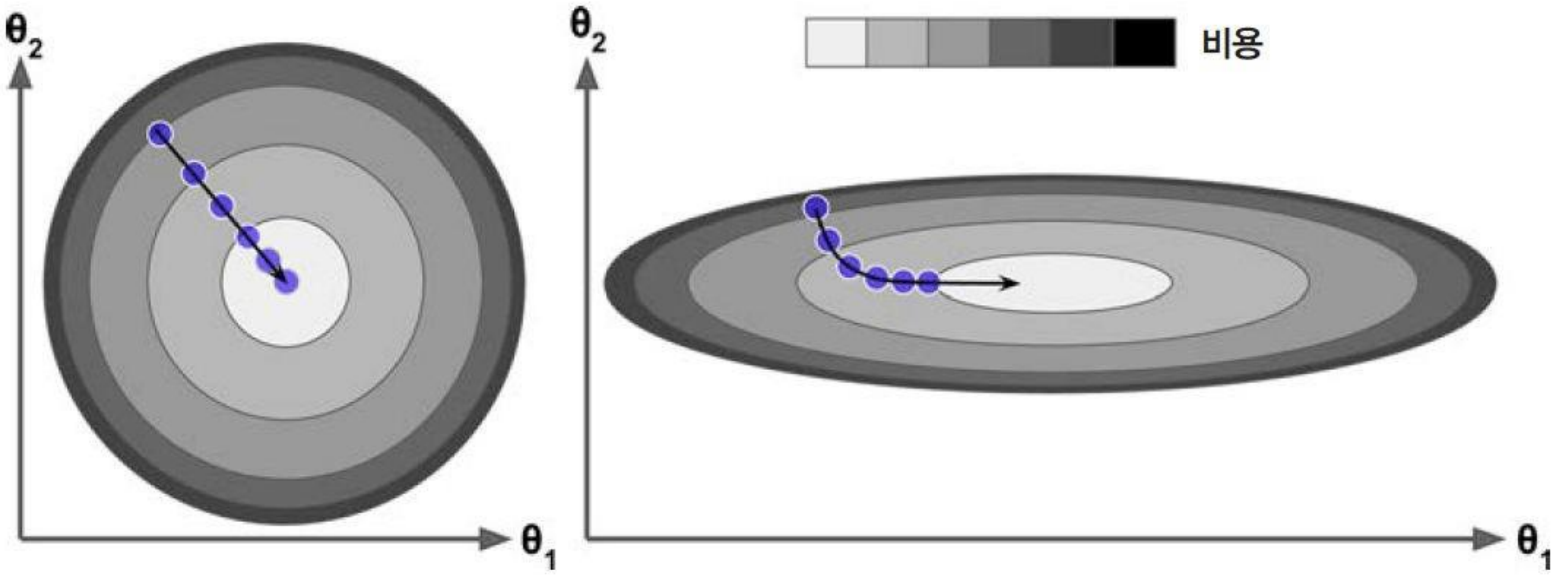



- MSE는 볼록 함수이므로 지역 최솟값이 없고 기울기가 일정하게 변함

# 경사하강법 : 문제점 – 특성 스케일에 민감

- 특성의 스케일이 다르면 모델 파라미터에 따른 비용 함수의 변화율이 달라짐

큰 특성의 모델 파라미터




작은 특성의 모델 파라미터

**경사하강법 사용할 때는 특성 의 크기를 균일하게 하기 바람: 특성 스케일링 (scikit-learn의 StandardScaler 사용)**

# 경사하강법 : 배치(batch) 경사 하강법

- 전체 훈련 세트를 사용하여 그래디언트를 계산
- 비용 함수의 최솟값에 안정적으로 수렴하지만 계산 비용이 큼

$$\nabla_\theta \mathbf{MSE}(\theta) = \begin{bmatrix} \frac{\partial}{\partial\theta_0}\mathbf{MSE}(\theta) \\ \frac{\partial}{\partial\theta_1}\mathbf{MSE}(\theta) \\ \vdots \\ \frac{\partial}{\partial\theta_n}\mathbf{MSE}(\theta) \end{bmatrix} = \frac{2}{m}\mathbf{X}^T \cdot (\mathbf{X} \cdot \theta - \mathbf{y})$$

X의 shape이 (m, n+1)일 때
$\nabla_\theta \mathbf{MSE}(\theta)$의 shape는?

$$\theta^{\{next\,step\}} = \theta - \eta \nabla_\theta \mathbf{MSE}(\theta)$$

$$\mathbf{MSE}(\theta) = \frac{1}{m}\sum_{i=1}^{m} (\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)})^2$$

$$\frac{\partial}{\partial\theta_j}\mathbf{MSE}(\theta) = \frac{2}{m}\sum_{i=1}^{m} (\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)})x_j^{(i)}$$

# 경사 하강법 : numpy 구현

```
eta = 0.1
n_iterations = 1000
m = 100
theta = np.random.randn(2,1)

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

$$\frac{2}{m}\mathbf{X}^T \cdot (\mathbf{X} \cdot \theta - \mathbf{y})$$

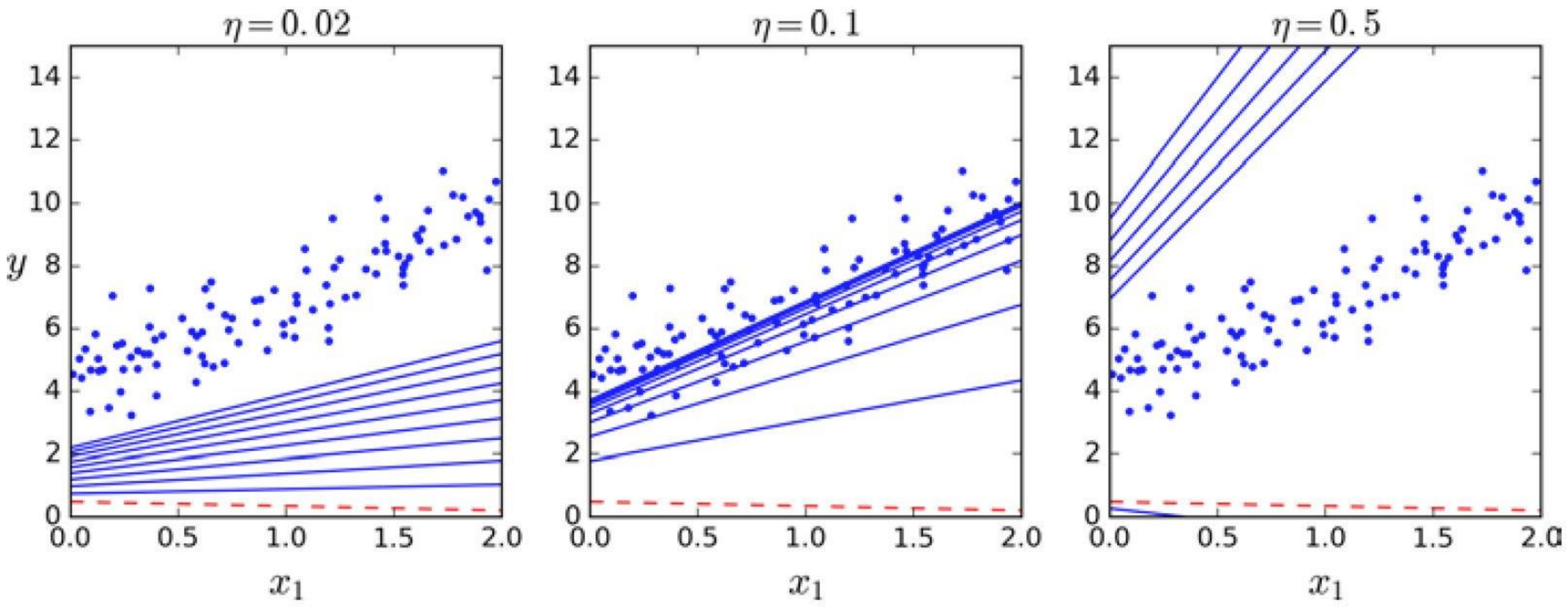
```
theta
```

```
array([[4.21509616],
       [2.77011339]])
```

**정규 방정식의 해**

```
array([[4.21509616],
       [2.77011339]])
```

# 경사하강법 : 학습률과 반복횟수

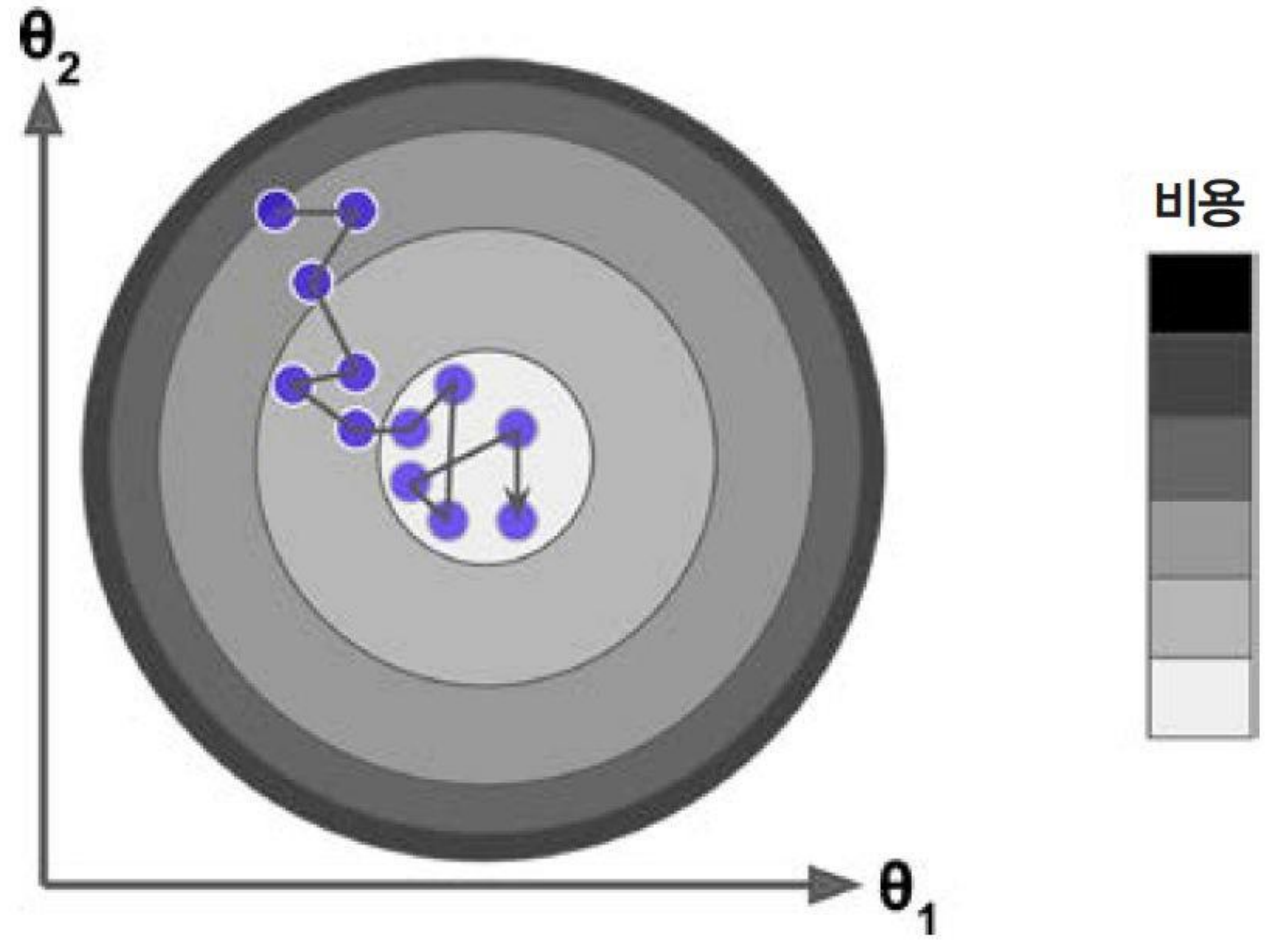



처음 10회 반복한 결과 (초기는 점선)
왼쪽 : 학습률 낮아 수렴이 늦음
중간 : 적당한 학습률
오른쪽 : 해를 지나친 후 발산

- 알맞은 학습률을 찾으려면 반복 횟수를 제한하여 그리드 탐색을 수행 : 그래디언트 벡터의 크기가 정해진 허용 오차보다 작으면(수렴이라 가정할 수 있으면) 알고리즘 반복을 중지함
- 학습률과 같은 하이퍼파라미터를 이론적으로 구할 수 있는 방법은 없음. 특히 딥러닝은 많은 하이퍼파라미터가 있기 때문에 과학보다는 기술에 가까움

# 경사하강법 : 확률적 경사 하강법(SGD)

- 확률적 : 무작위 라는 의미 (Stochastic) ←→ 배치(batch)
- 확률적 경사 하강법 : 훈련 데이터에서 하나의 샘플을 무작위로 선택해 경사 하강법을 수행 (Stochastic Gradient Descent)
- 배치 경사 하강법보다 빠르고 큰 데이터셋을 처리할 수 있음(외부 메모리 알고리즘)
- 불안정하게 요동하면서 수렴 (최소값에 빠지지 않을 가능성이 높음) ➔ 학습률을 점차 감소하는 것이 좋음(학습 스케줄링)



# 경사하강법 : SGD 넘파이 구현

$$\mathrm{MSE}(\theta) = \boxed{\frac{1}{m}\sum_{i=1}^{m}} (\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)})^2$$

$$\frac{\partial}{\partial \theta_j}\mathrm{MSE}(\theta) = \boxed{\frac{2}{m}\sum_{i=1}^{m}} (\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)}$$

SGD에서는 이것이 없어짐

```
n_epochs = 50
t0, t1 = 5, 50  # 학습 스케줄 하이퍼파라미터

def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1)  # 무작위 초기화

for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
```

**0.1에서부터 t가 커질수록 줄어듬**

**훈련 샘플 개수 만큼 반복**

**샘플 추출**
**(하나의 샘플이 한 번 에포크 내에서 여러번 추출될 수 있음)**

*** 사이킷런의 SGDClassifier, SGDRegressor는 에포크마다 전체 샘플을 뒤섞은 후 순서대로 처리함**

# 경사하강법 : SGD 사이킷런 구현 : SGDRegressor

- 사이킷런의 SGD 구현 : SGDRegressor(회귀), SGDClassifier(분류)

에포크

규제 없음

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(max_iter=50, penalty=None, eta0=0.1, random_state=42)
sgd_reg.fit(X, y.ravel())
```

1차원 배열로... y.reshape(-1)과 동일

$$\eta^{(t)} = \frac{\text{eta0}}{t^{\text{powet\_t}}}$$ ← power_t 기본값은 0.25

```
sgd_reg.intercept_, sgd_reg.coef_
```

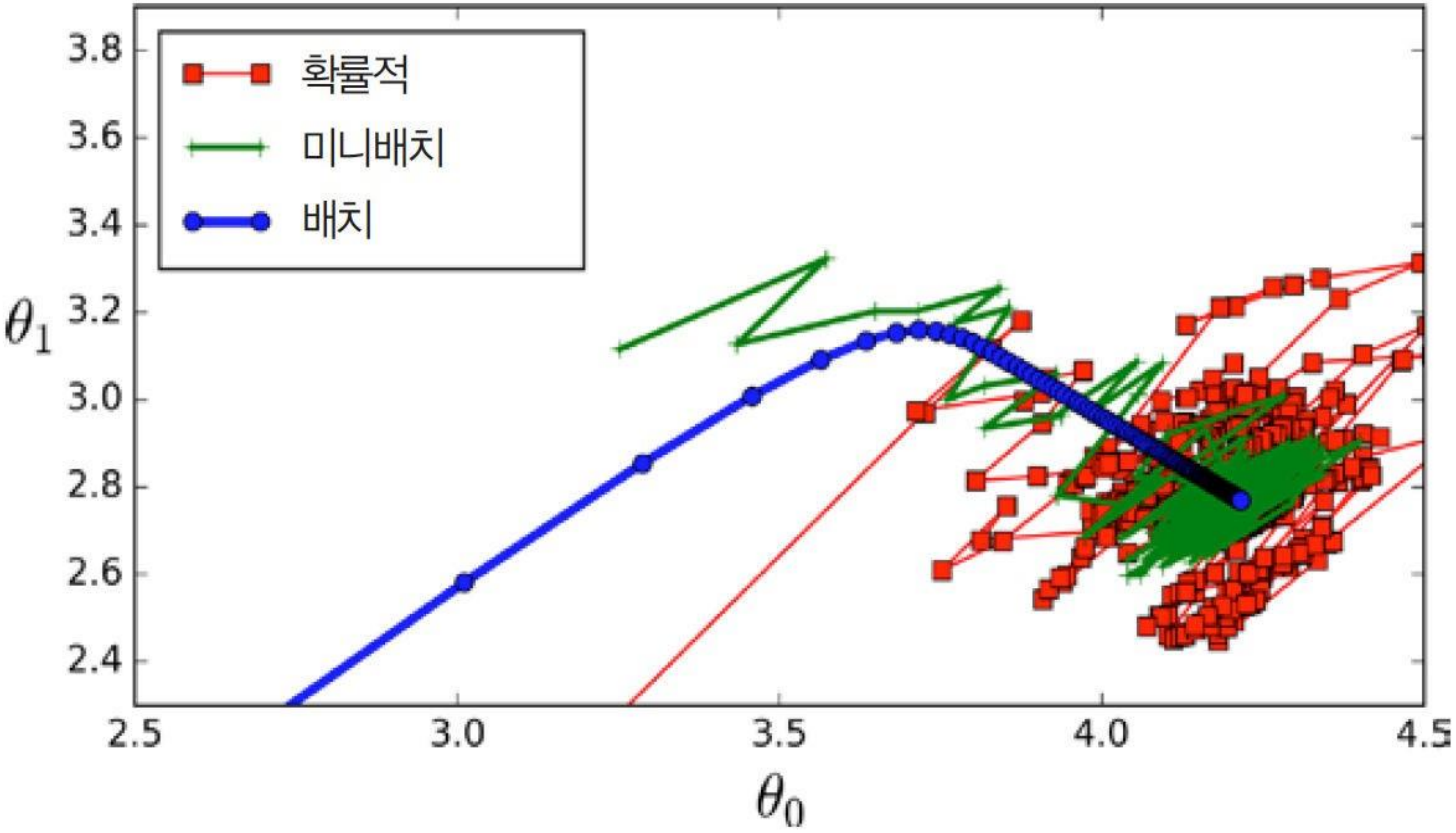
```
(array([4.16782089]), array([2.72603052]))
```

정규 방정식의 해

```
array([[4.21509616],
       [2.77011339]])
```

# 경사하강법 : 미니배치(Minibatch) 경사하강법

- 미니배치(mini-batch): 확률적과 배치의 장단점을 절충



# 경사하강법 : 경사 하강법 비교

- 선형 회귀를 사용한 비교

**학습률, 에포크수**

| 알고리즘 | *m*이 클 때 | 외부 메모리 학습 지원 | *n*이 클 때 | 하이퍼 파라미터 수 | 스케일 조정 필요 | 사이킷런 |
|---|---|---|---|---|---|---|
| 정규방정식 | 빠름 | No | 느림 | 0 | No | LinearRegression |
| 배치 경사 하강법 | 느림 | No | 빠름 | 2 | Yes | n/a |
| 확률적 경사 하강법 | 빠름 | Yes | 빠름 | ≥2 | Yes | SGDRegressor |
| 미니배치 경사 하강법 | 빠름 | Yes | 빠름 | ≥2 | Yes | n/a |

**SGDRegressor에서 훈련 샘플을 조금씩 나누어 학습할 수 있는 partial_fit() 메서드도 SGD 방식으로 경사 하강법을 적용합니다.**

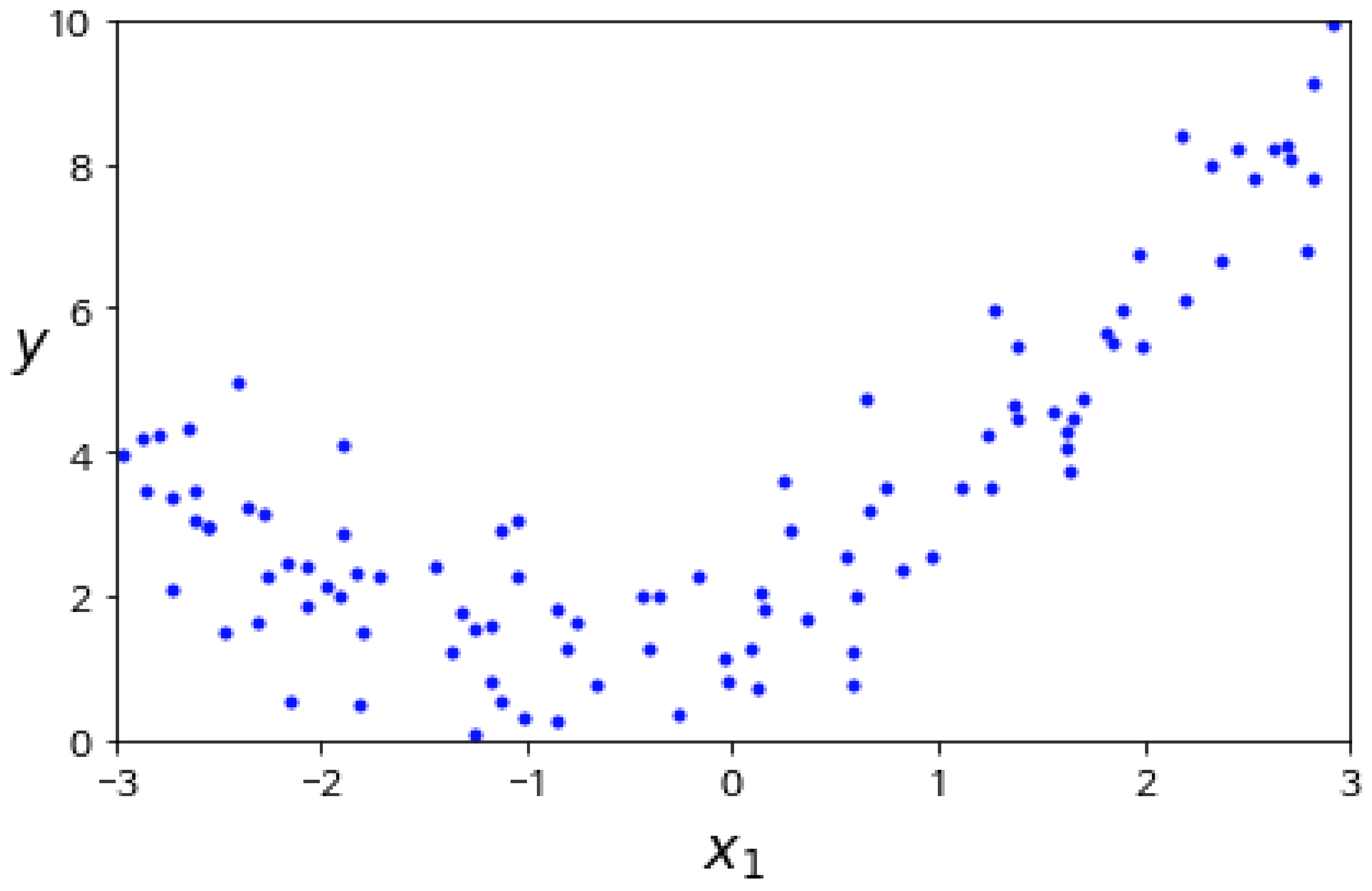
# 다항 회귀(Polymomial Regression)

- 특징의 고차항(예를들어 $x^3$. 비선형특징)을 만들어 새로운 특징으로 추가함
- 비선형 데이터를 선형 모델로 학습할 수 있음

- 샘플 데이터

$y = 0.5x^2 + x + 2 +$ **가우시안 노이즈**

```
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```



# 다항특징 : 사이킷런 PolynomialFeatures

- 특징이 a, b 두 개인 경우 degree=2 이면 ab, $a^2$, $b^2$ 항이 추가됨
- Interaction_only=True로 하면 ab항만 추가됨

0차부터 d차까지
전체 특징 수 :

$$\frac{(n+d)!}{d!n!}$$

```python
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X)
X[0]
```

2차항

편향을 위한 1을 추가하지 않음

```
array([-0.75275929])
```

```python
X_poly[0]
```

```
array([-0.75275929,  0.56664654])
```

```python
poly_features.get_feature_names()
```

특성 이름으로 확인

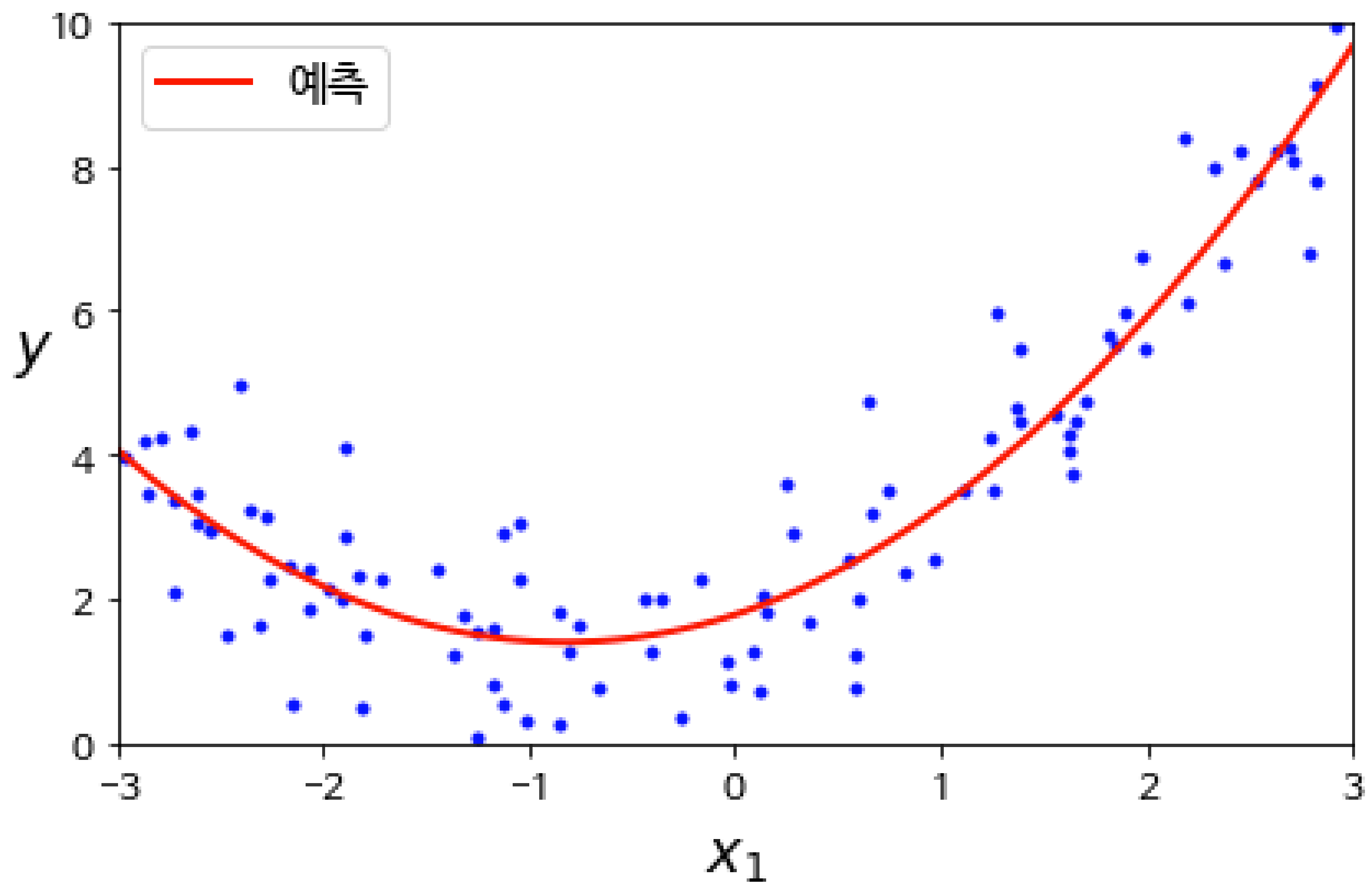
```
['x0', 'x0^2']
```

# 다항회귀 : LinearRegression으로 학습결과

```
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
lin_reg.intercept_, lin_reg.coef_
```

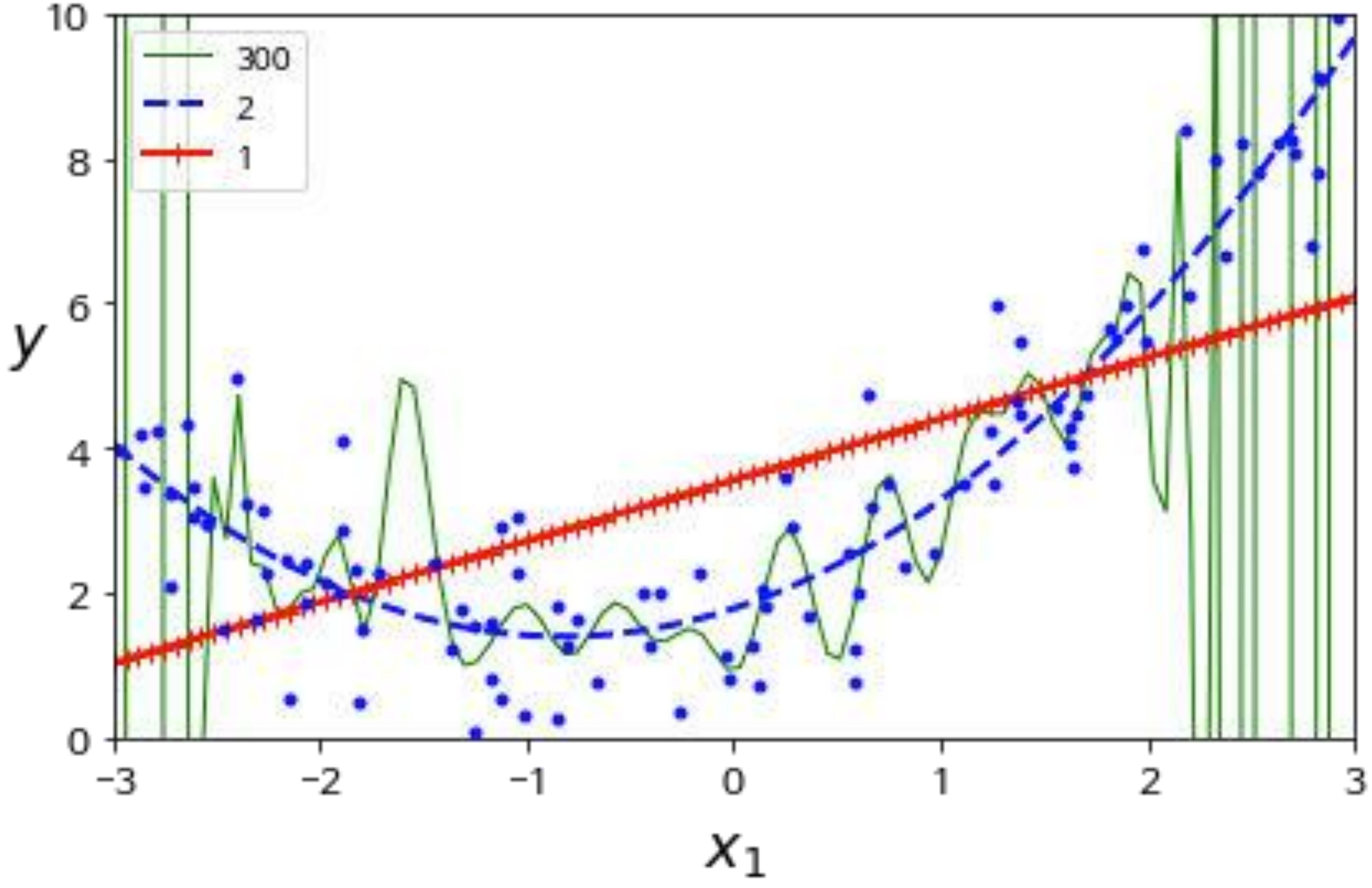
```
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

$y = 0.5x^2 + x + 2 +$ **가우시안 노이즈**



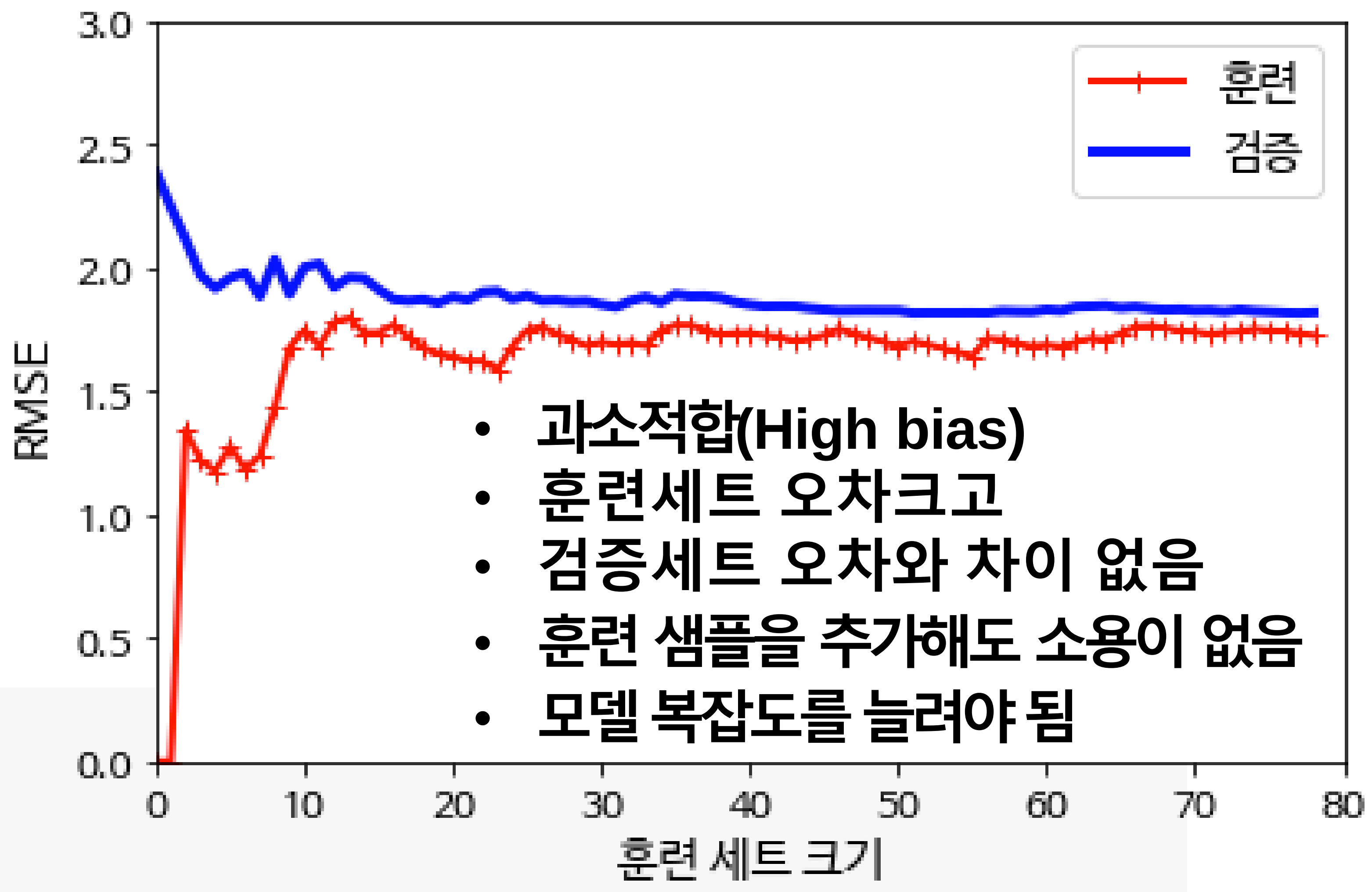
다항회귀 : 복잡도와 학습 결과

- 적절한 복잡도는 얼마일까?



# 학습 곡선

- 훈련세트크기 바꿔가면서 훈련세트, 검증세트의 오차 계산. (검증세트는 바꾸지 않음)
- 과대/과소/적정 적합을 판단할 수 있음




```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=10)
    train_errors, val_errors = [], []
    for m in range(1, len(X_train)):
        model.fit(X_train[:m], y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_val_predict = model.predict(X_val)
        train_errors.append(mean_squared_error(y_train[:m], y_train_predict))
        val_errors.append(mean_squared_error(y_val, y_val_predict))
```

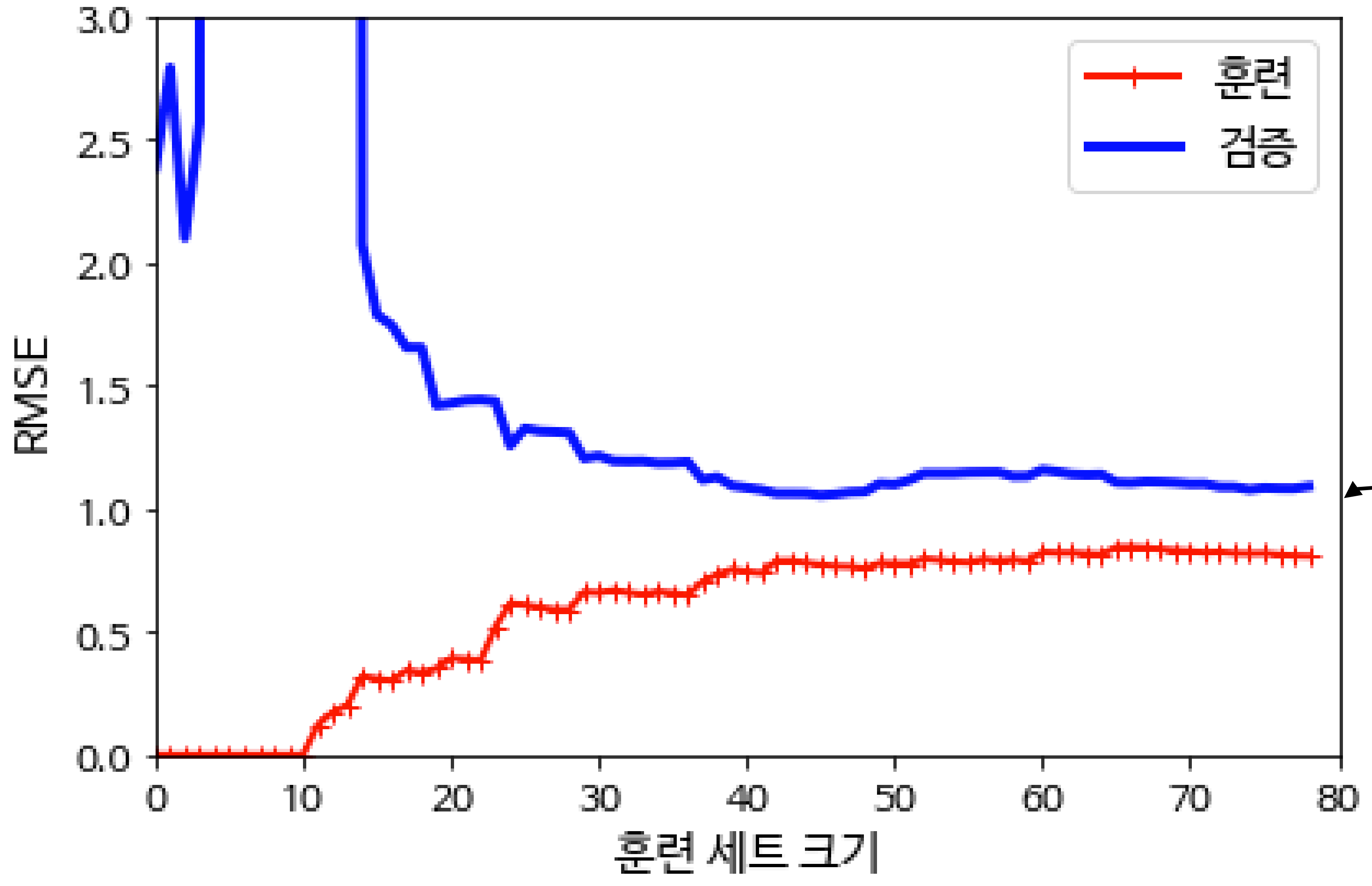
**훈련 세트 크기 1~m**

```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
```

# 학습곡선 : 과대적합

복잡도 크게 했음 (10차 다항식)

```
from sklearn.pipeline import Pipeline

polynomial_regression = Pipeline([
        ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
        ("lin_reg", LinearRegression()),
    ])

plot_learning_curves(polynomial_regression, X, y)
```

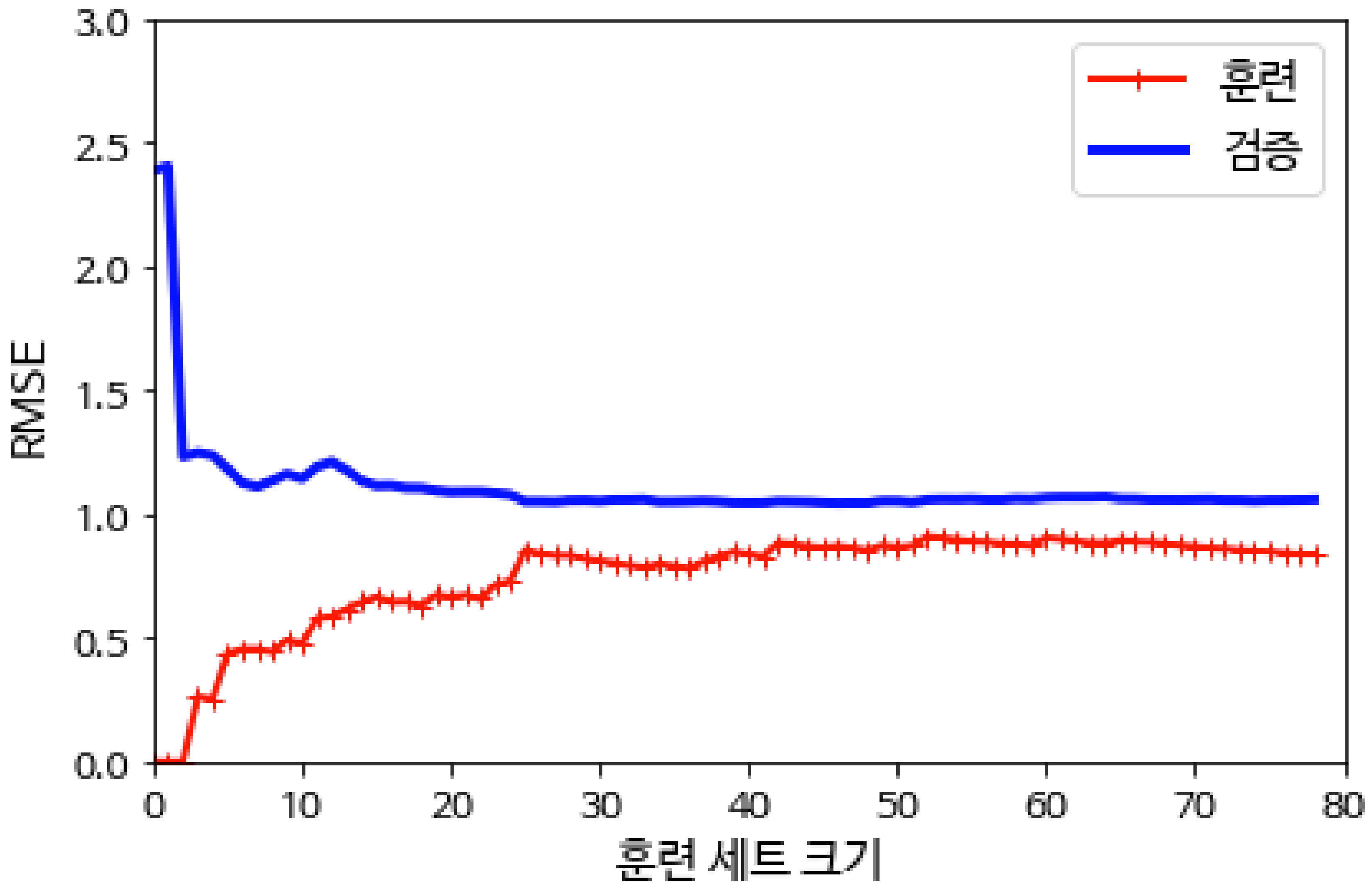



- 과대적합 (High Variance)
- 훈련세트 오차 작음
- 훈련세트와 검증세트간 차이 큼
- 모델 복잡도를 줄이거나
- 훈련세트를 매우 늘려야...

# 학습곡선 : 적정학습

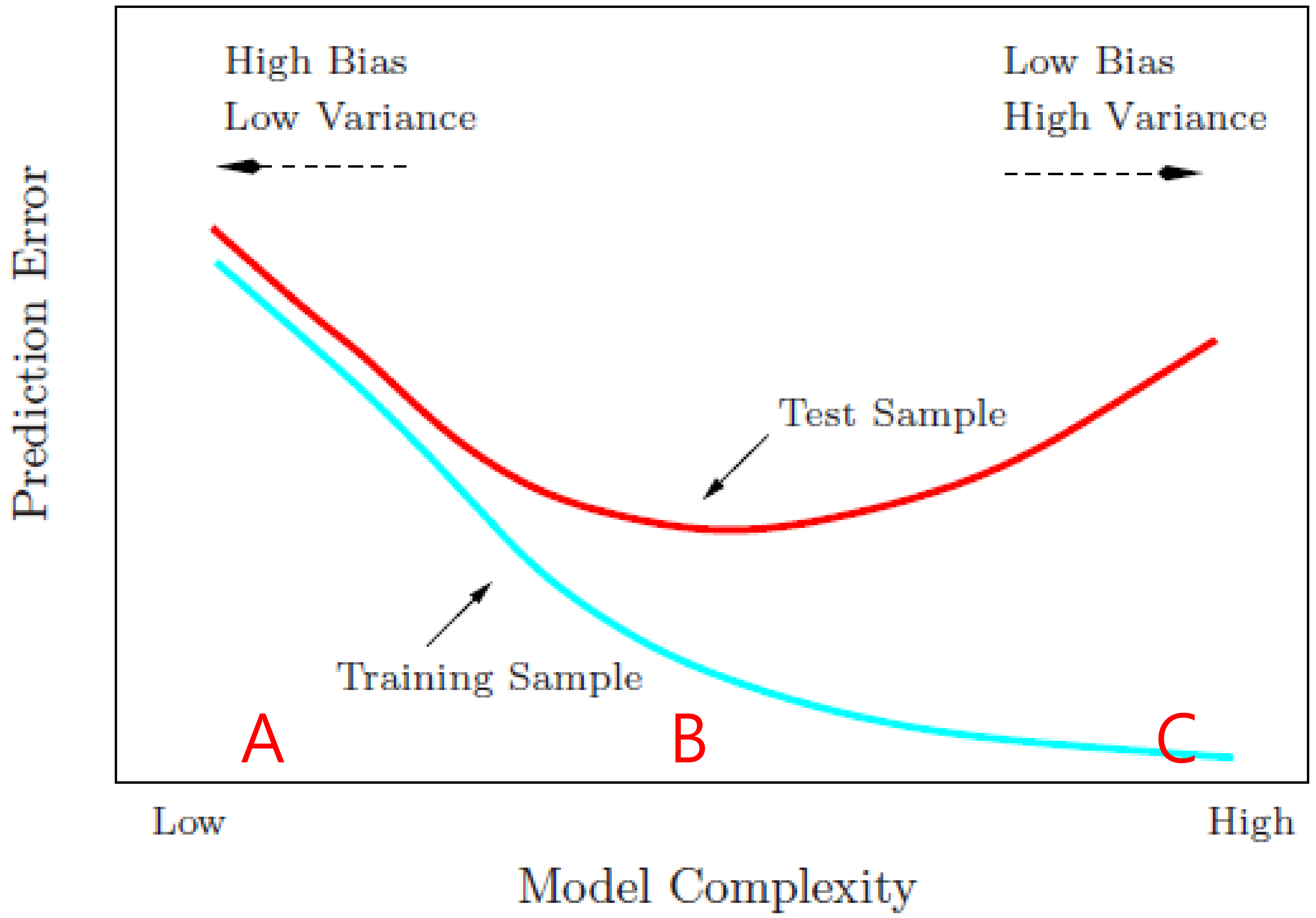
복잡도를 줄였음 (2차 다항식)

```
polynomial_regression = Pipeline([
        ("poly_features", PolynomialFeatures(degree=2, include_bias=False)),
        ("lin_reg", LinearRegression()),
    ])

plot_learning_curves(polynomial_regression, X, y)
```




- **적정적합**
- **훈련세트 오차 작음**
- **훈련세트와 검증세트간 작음**

# 모델 복잡도 곡선

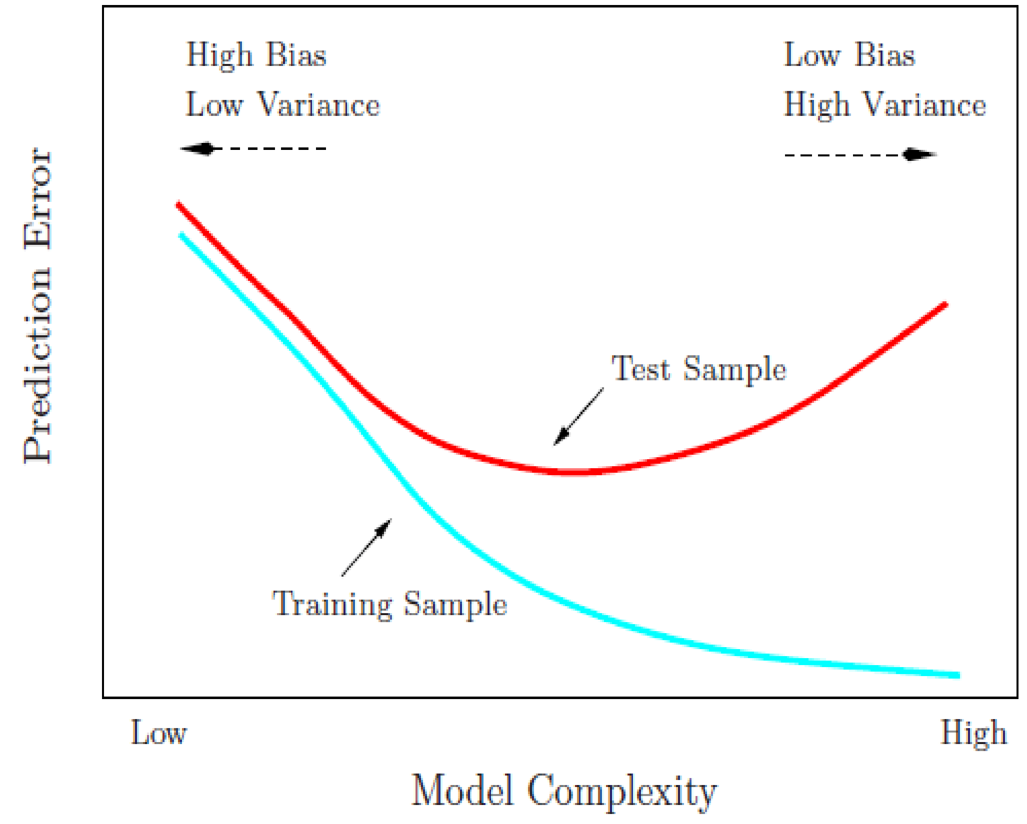



과소적합?

과대적합?

적정적합?

# 모델복잡도 : 편향/분산 트레이드오프




- 오차에는 3가지 종류가 있음 : Bias, Variance, Noise

- 편향:(bias) 잘못된 가정으로 발생. 과소 적합 모델
- 분산(variance): 훈련 데이터에 민감한 모델 때문에 발생. 과대 적합 모델
- 회피할 수 없는 오차(irreducible error. Noise) : 데이터 자체의 노이즈 (예. 잘못 입력된 데이터) 이상치를 감지해서 버리는 것이 좋음

✓ 복잡도가 커지면 분산이 늘고 편향은 줄어 듬.
복잡도가 줄어들면 분산이 줄고 편향이 커짐 ➔ **Bias / Variance Tradeoff**

# 규제 모델 (Regularized Linear Models)

- 과대적합을 줄이기 위해 복잡도를 줄이는 방법
  - ✓ 다항식 경우는 차수를 줄였음.  보다 일반적인 방법은?

- 가중치가 커지지 않도록 자유도에 제한을 줌
- 규제가 있는 선형 회귀 모델: 릿지(Ridge), 라쏘(Lasso), 엘라스틱넷(ElasticNet)

규제모델 : 

# 릿지 회귀(Ridge Regression)

- 티호노프(Tikhonov) 규제라고도 부름
- 비용 함수에 L2 노름의 제곱을 추가

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{2}{m} \sum_{i=1} (\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)} + \alpha\theta$$

미분 결과를 간단하게 만들기 위해 추가

규제량을 조절: 하이퍼파라미터

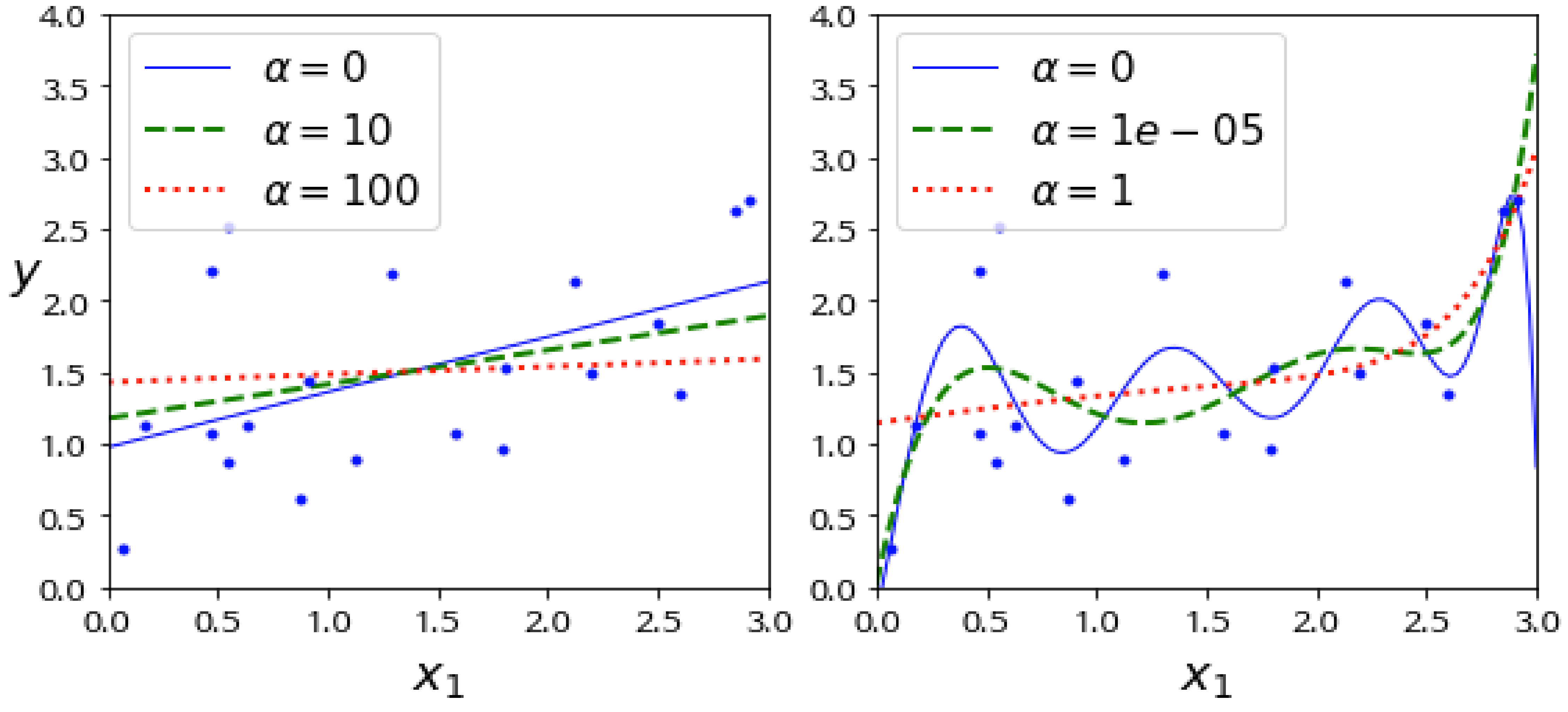
$$J(\theta) = \mathrm{MSE}(\theta) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$$

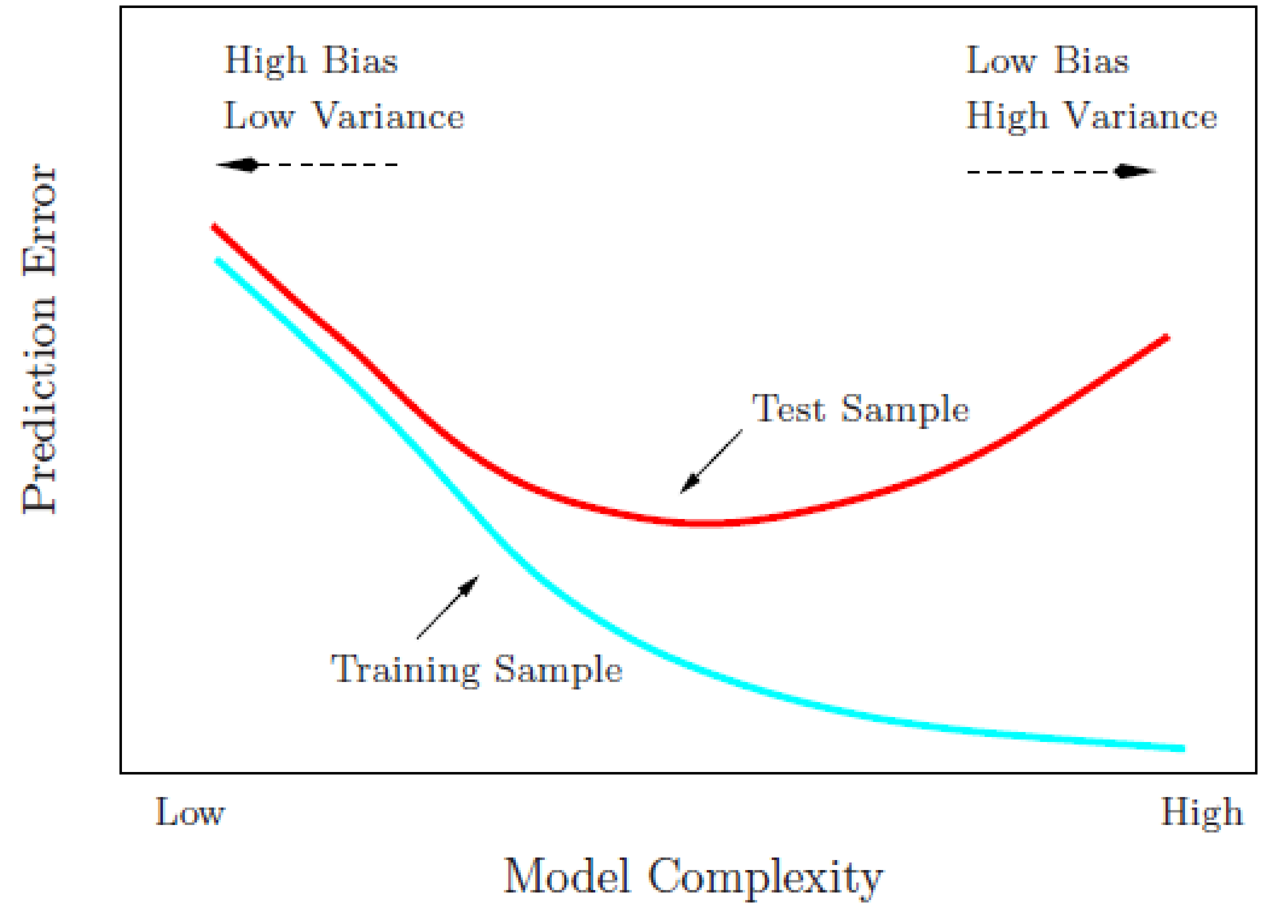
$\theta_0$ 는 규제하지 않음

- alpha가 커지면 가중치가 0에 가까워지고, alpha가 0에 가까우면 MSE만 남음
- 훈련이 끝나고 성능을 평가할 때는 규제를 포함하지 않음(성능 평가와 비용 함수 가 다른 경우가 많음).
- 각 가중치를 같은 비율로 규제하기 때문에 입력 데이터의 스케일에 민감함.

# 규제모델 : 릿지 모델의 예

- 선형 데이터에 대해 실험
- 왼쪽 : 리지 규제를 가지는 선형모델
- 오른쪽 : 리지 규제를 가지는 10차 다항식 모델




alpha가 커지면
모델 복잡도(complexity)는 줄어드는가?
오차의 분산(variance)이 작아지는가?
오차의 편향(bias)이 커지는가?

# 규제모델 : Ridge 정규 방정식

- 숄레스키 분해(Cholesky decomposition)을 사용하여 해를 구할 수 있음

$$\hat{\theta} = (\mathbf{X}^T \cdot \mathbf{X} + \alpha\mathbf{A})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

**A: (n+1, n+1)의 단위 행렬, 왼쪽 맨 위 원소는 0입니다.**

- 구현 1 : 리지정규방정식

```
from sklearn.linear_model import Ridge
ridge_reg = Ridge(alpha=1, solver="cholesky", random_state=42)
ridge_reg.fit(X, y)
```

solver='auto'가 기본값이며 희소, 특이 행렬이 아닐 경우 'cholesky' 방식을 사용함

- 구현 2 : 리지 SGD

```
sgd_reg = SGDRegressor(penalty="l2")
sgd_reg.fit(X, y.ravel())
```

penalty="$\ell_2$" 리지정규화 의미.   alpha=0.1처럼 alpha값 지정 가능

# 규제모델 : 라쏘(Lasso) 회귀 Linear Absolute Shrinkage and Selection Operator Regression

- 비용 함수에 L1 노름을 추가

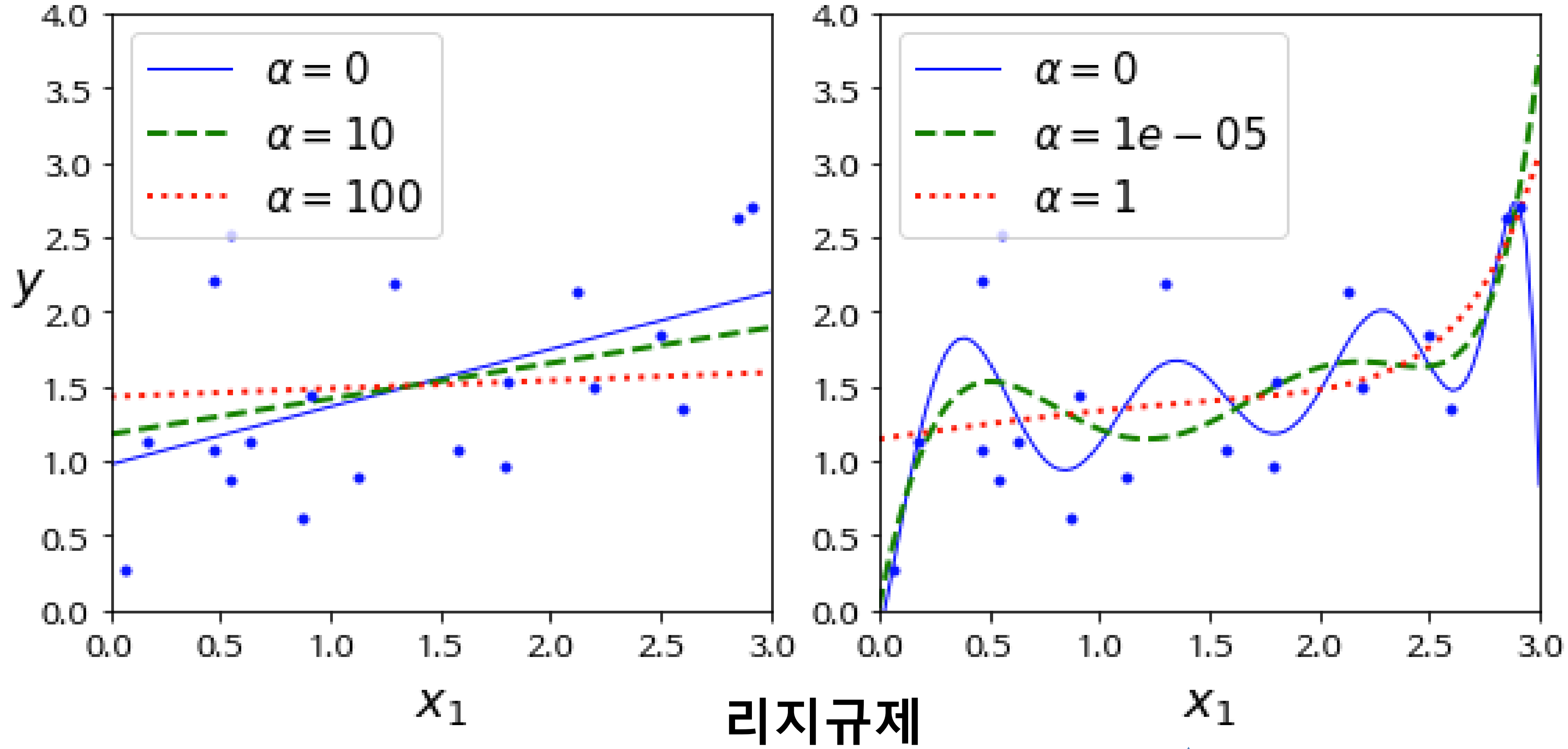
$$J(\theta) = \mathrm{MSE}(\theta) + \alpha \sum_{i=1}^{n} |\theta_i|$$

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{2}{m} \sum_{i=1} \left(\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)}\right) x_j^{(i)} \pm \alpha$$

$$g(\theta, J) = \nabla_\theta \mathrm{MSE}(\theta) + \alpha \begin{pmatrix} \mathrm{sign}(\theta_1) \\ \mathrm{sign}(\theta_2) \\ \vdots \\ \mathrm{sign}(\theta_n) \end{pmatrix} \quad \text{where } \mathrm{sign}(\theta_i) = \begin{cases} -1 & \text{if } \theta_i < 0 \\ 0 & \text{if } \theta_i = 0 \\ +1 & \text{if } \theta_i > 0 \end{cases}$$

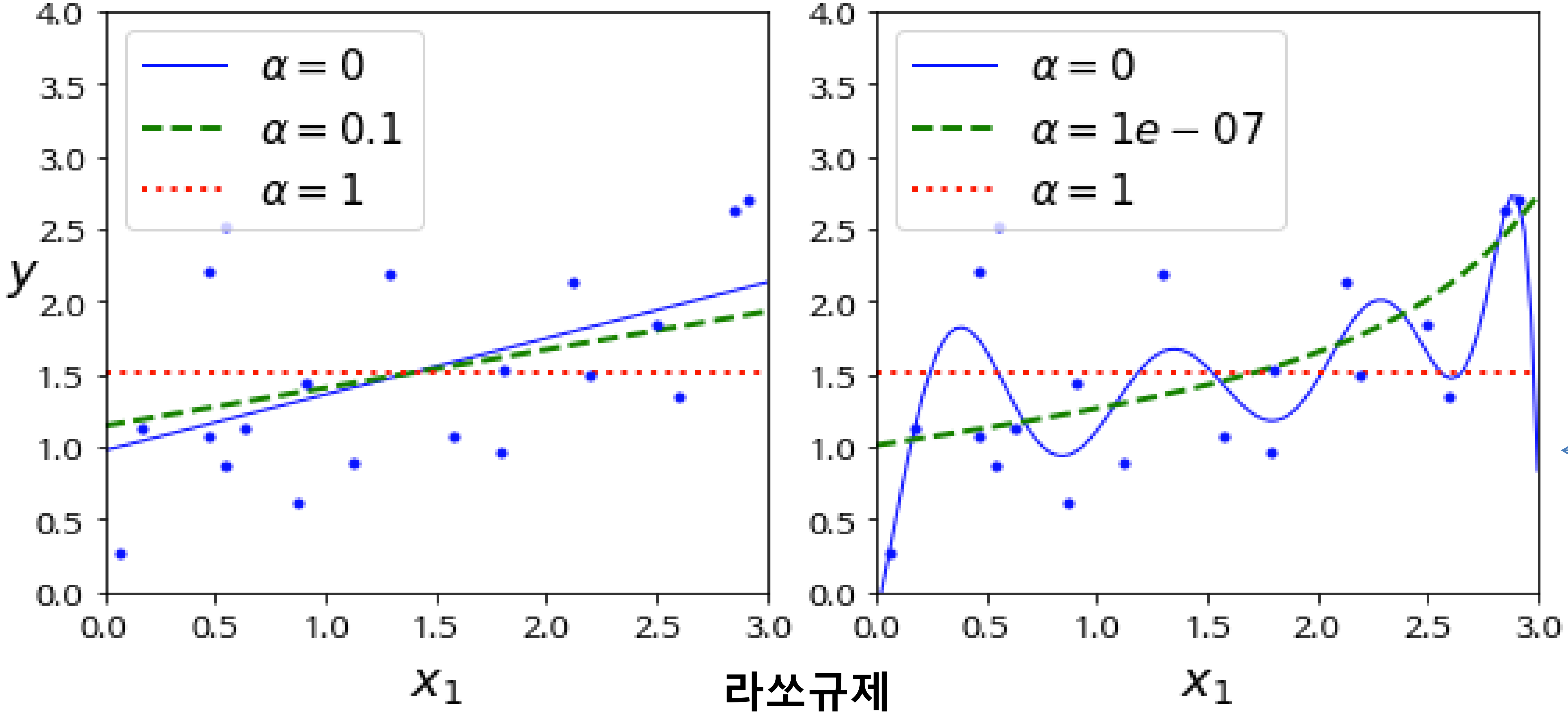
# 규제모델 : 라쏘 모델의 예

- 선형 데이터에 대해 실험
- 왼쪽 : 규제를 가지는 선형모델
- 오른쪽 : 규제를 가지는 10차 다항식 모델




리지규제

규제가 강해지면 (alpha가 커지면)
함수가 smooth해지지만
차수(10차)는 유지



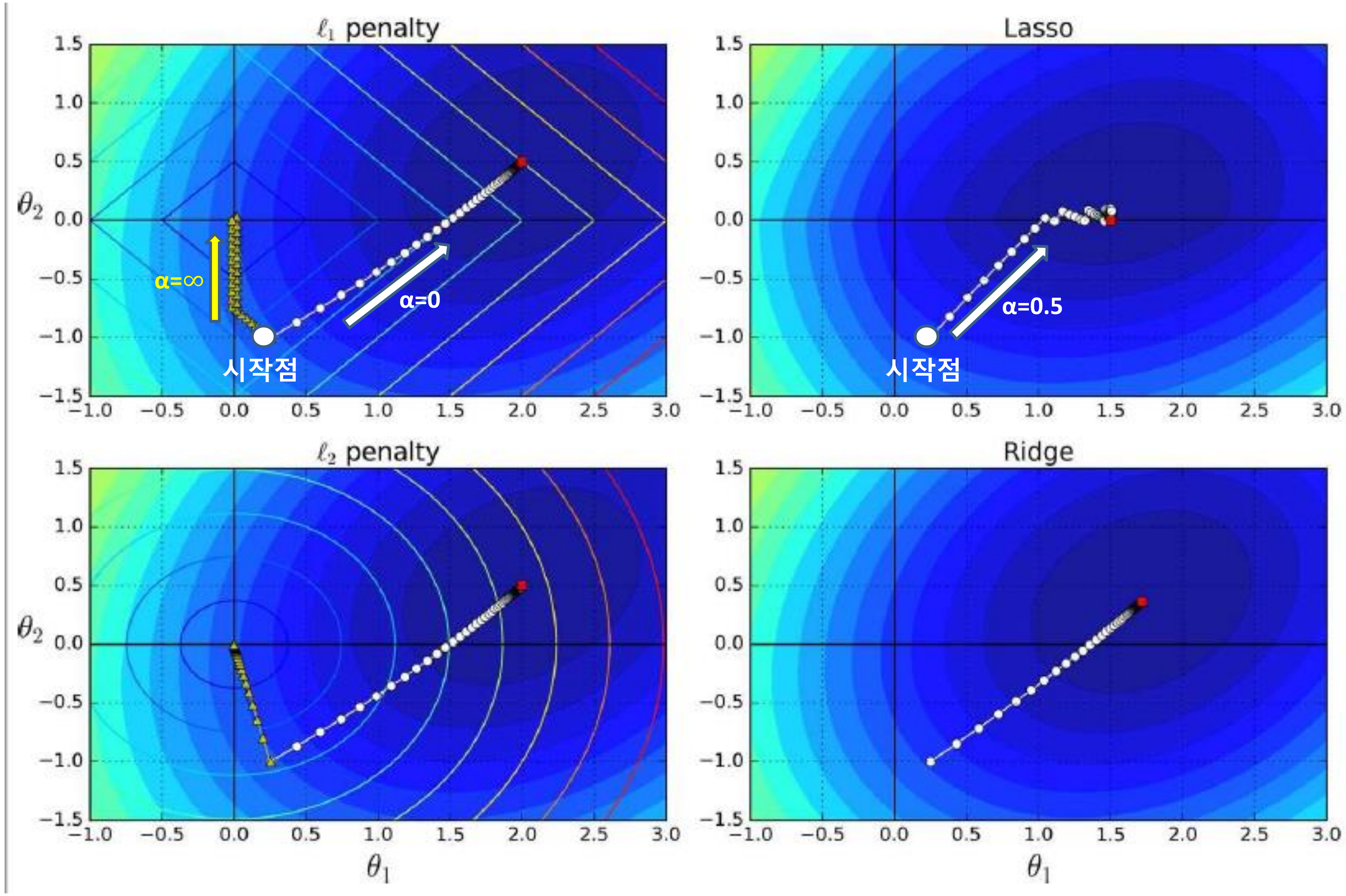

라쏘규제

규제가 강해지면 (alpha가 커지면)
차수가 낮아짐 (10차 → 2차 → 1차)
다항식에서 일부 항이 없어진다는 의미
특징 선택으로 사용할 수 있음

# 규제모델 : 라쏘의 특징

```
from sklearn.linear_model import Lasso
lasso_reg = Lasso(alpha=0.5)
lasso_reg.fit(X, y)
```

- 일부 가중치를 0으로 만듦(특징 선택 효과).




$y = 2 \times x_1 + 0.5 \times x_2$

Θ$_1$= 0.6, Θ$_2$= 0
다소 부정확, 그러나
특징 선택 효과

Θ$_1$= 1.7, Θ$_2$= 0.5
정확, 그러나
특징 선택은 안함

# 규제모델 : 엘라스틱넷(ElasticNet)

- 릿지와 회귀의 규제를 절충한 모델

$$J(\theta) = \mathrm{MSE}(\theta) + r\alpha\sum_{i=1}^{n}|\theta_i| + \frac{1-r}{2}\alpha\sum_{i=1}^{n}\theta_i^2$$

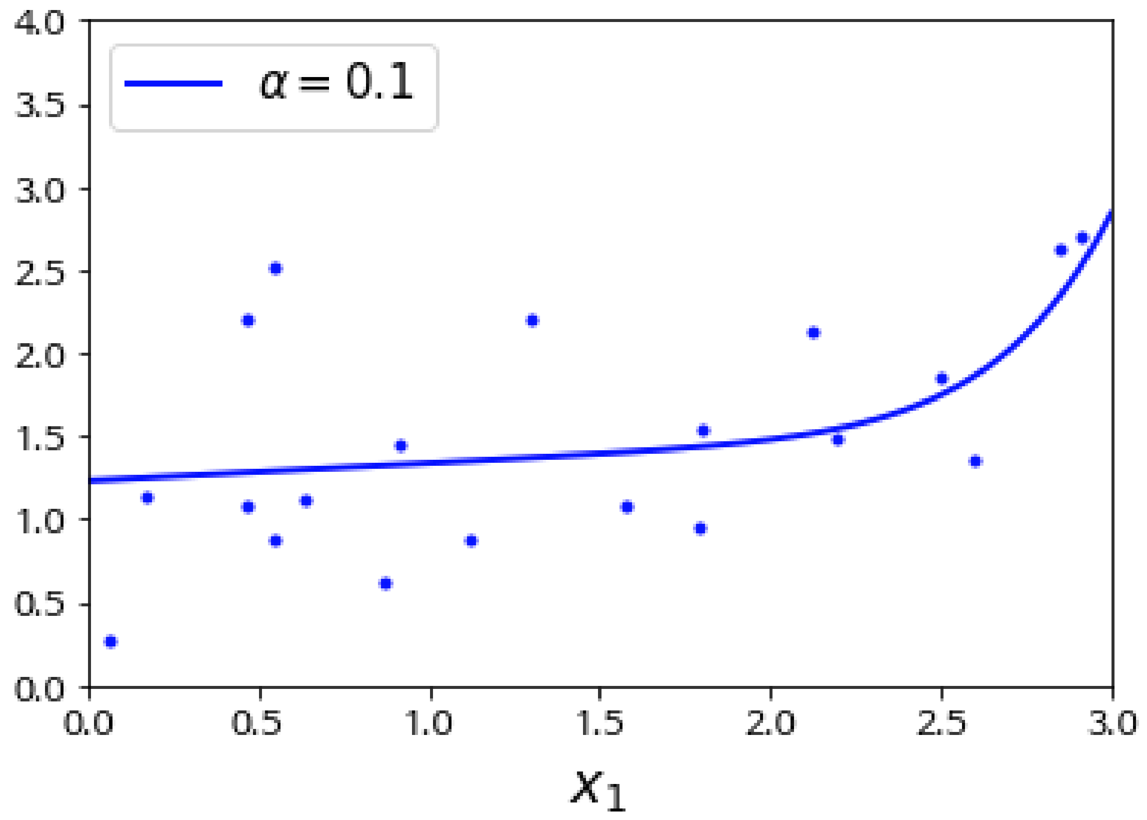
- r=0이면 릿지, r=1이면 라쏘.

- 보편적으로 라쏘 보다는 릿지나 엘라스틱넷을 선호함

```
from sklearn.linear_model import ElasticNet
elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5, random_state=42)
elastic_net.fit(X, y)
```

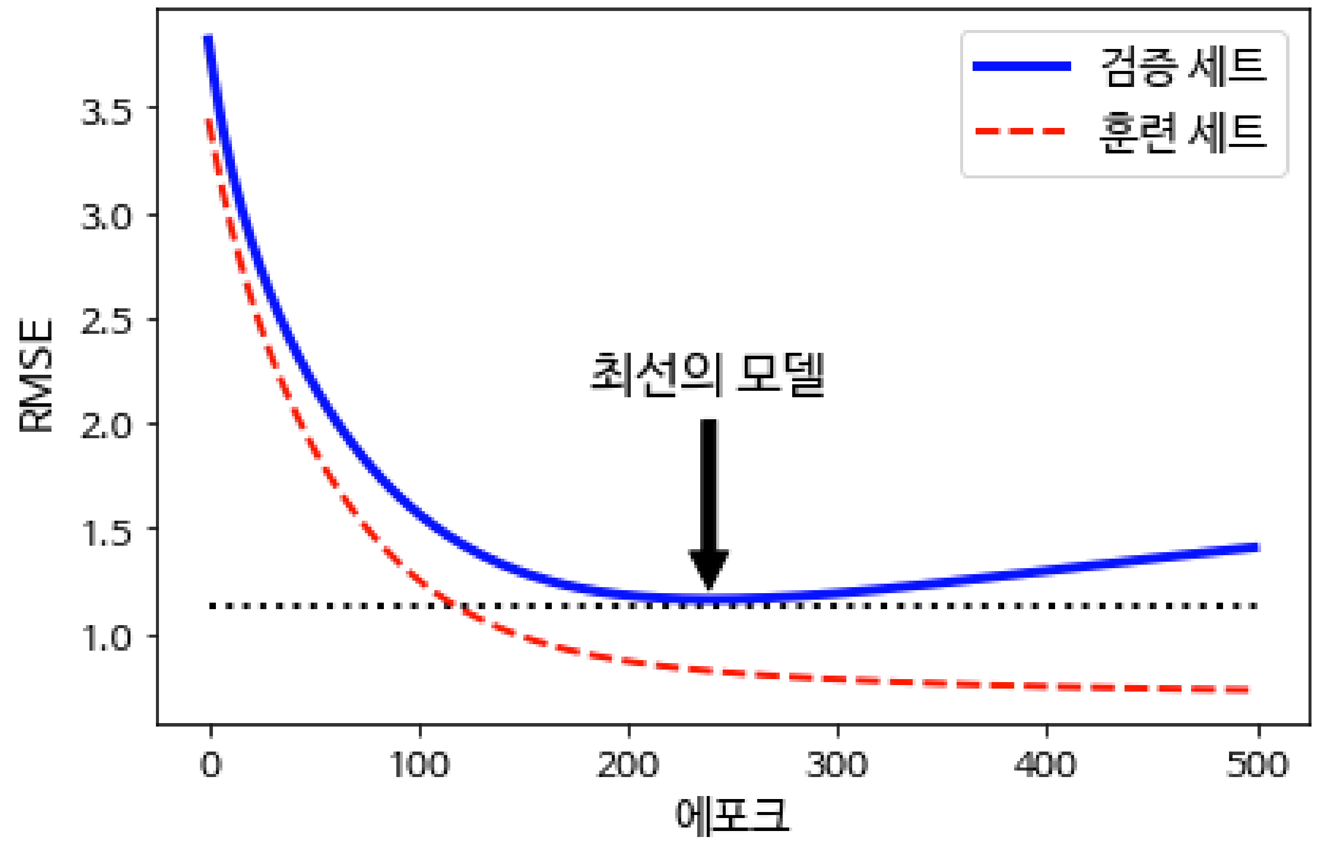
r

*** Lasso 클래스는 ElasticNet(l1_ratio=1)**



# 규제모델 : 조기 종료(Early stopping)

- 검증 에러가 최솟값에 도달했을 때 훈련을 중지
- 딥러닝 모델을 훈련할 때 충분히 과대적합된 모델을 만들고 난 후 다른 규제방법을 사용하라 : Andrew Ng



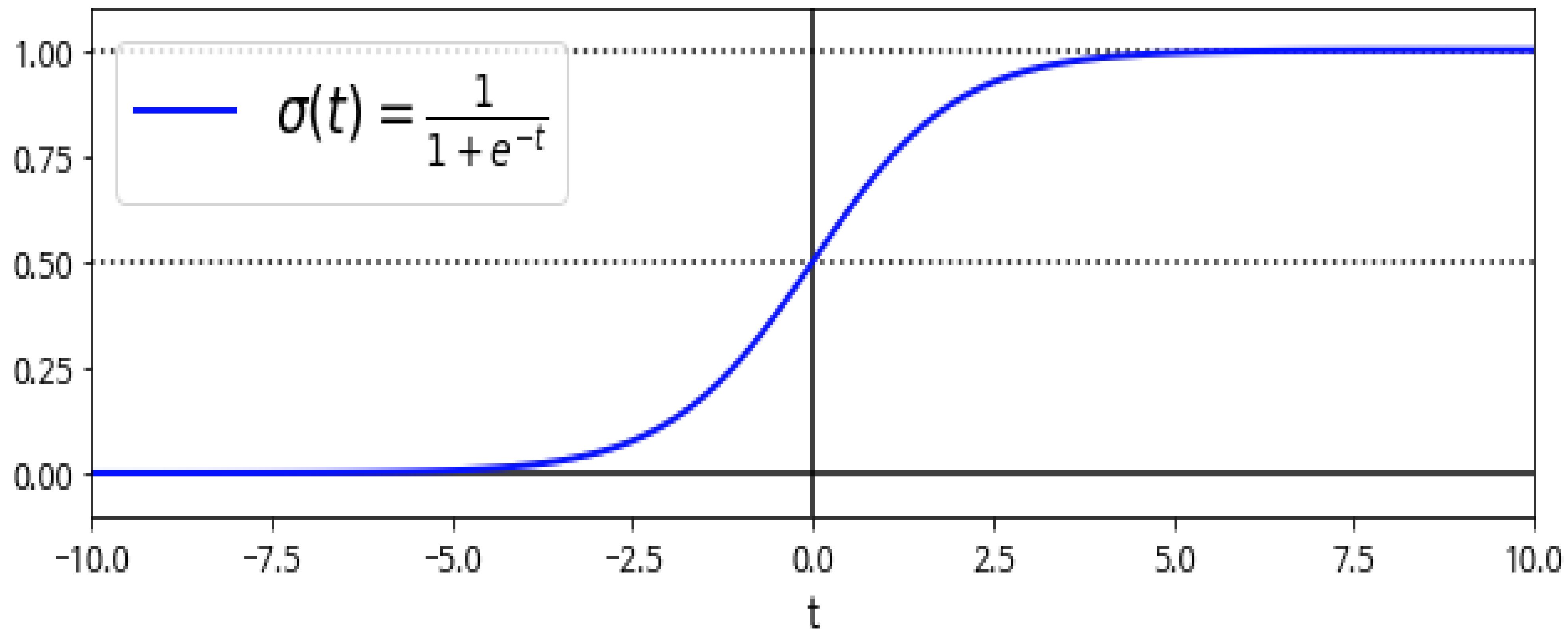

```
sgd_reg = SGDRegressor(n_iter=1, warm_start=True, penalty=None,
                       learning_rate="constant", eta0=0.0005)

minimum_val_error = float("inf")
best_epoch = None
best_model = None
for epoch in range(1000):
    sgd_reg.fit(X_train_poly_scaled, y_train)  # 훈련을 이어서 진행합니다.
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    val_error = mean_squared_error(y_val, y_val_predict)
    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = clone(sgd_reg)
```

실제로는 검증오차가 진동하면서 내려가므로
이동평균을 쓰는 것이 더 좋을 듯

# 로지스틱(logistic) 회귀

- 로짓(logit) 회귀라고도 함
- 샘플이 특정 클래스에 속할 확률을 추정하는데 많이 쓰임 ➔ 분류문제
- 선형 방정식을 시그모이드(sigmoid) 함수에 통과시켜 0~1 사이의 확률을 계산함
- 로지스틱 회기 모델 (Logistic Regression Model) $\hat{p} = h_\theta(\mathbf{x}) = \sigma(\theta^T \cdot \mathbf{x})$
- 로지스틱 함수(Logistic Function) $\sigma(t) = \dfrac{1}{1 + \exp(-t)}$
- 예측(Prediction)

$$\hat{y} = \begin{cases} 0 & \text{if } \hat{p} < 0.5, \\ 1 & \text{if } \hat{p} \geq 0.5. \end{cases}$$



# 로지스틱회귀 : 비용함수(Cost Function)

- 샘플 하나에 대해.

진짜값(레이블) : $y$

$$c(\theta) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1, \\ -\log(1-\hat{p}) & \text{if } y = 0. \end{cases}$$

$$c(\theta) = -y\log(\hat{p}) - (1-y)\log(1-\hat{p})$$

| C($\theta$) | | $\hat{p}$ (예측확률) | |
|---|---|---|---|
| | | → 1 | → 0 |
| $y$ | 1 | -log(1) → 0 | -log(0) → ∞ |
| | 0 | -log(0) → ∞ | -log(1) → 0 |

- 샘플 전체에 대해 : 비용함수

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}log(\hat{p}^{(i)}) + (1-y^{(i)})log(1-\hat{p}^{(i)})]$$

Cross Entropy

- 비용함수 미분

$$\frac{\partial}{\partial\theta_j}\mathrm{J}(\theta) = \frac{1}{m}\sum_{i=1}^{m}(\sigma(\theta^T \cdot \mathbf{x}^{(i)}) - y^{(i)})\, x_j^{(i)}$$

# 로지스틱회기 : 붓꽃(Iris) 예 1

- 3개 클래스 : Setosa, Versicolor, Virginica
- 4개 특징 : petal width(꽃잎 폭), petal length(꽃잎 길이), sepal width(꽃받침 폭), sepal length(꽃받침 길이)
- 각 클래스당 150개 샘플

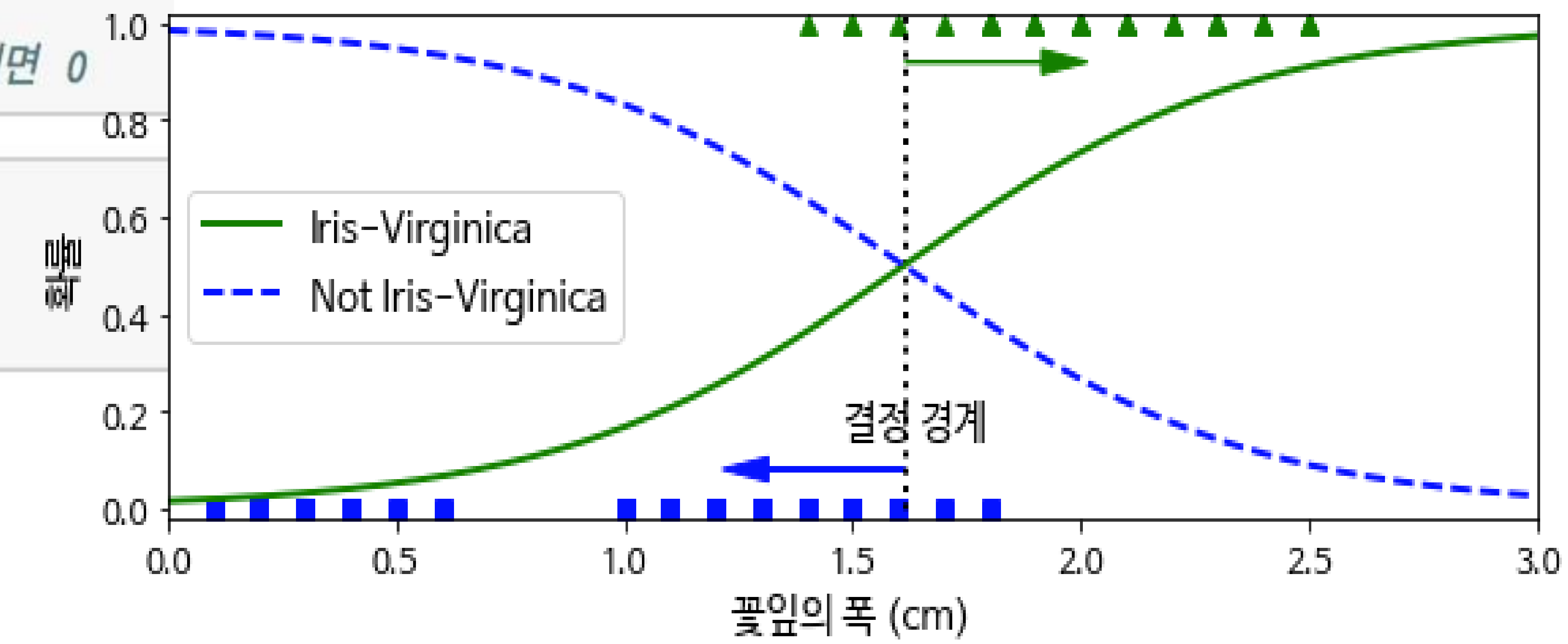

- 특징 1개(petal_width)로 분류 (Virginica / Not-Virginica)

```
from sklearn import datasets
iris = datasets.load_iris()
```

```
X = iris["data"][:, 3:]  # 꽃잎 넓이
y = (iris["target"] == 2).astype(np.int)  # Iris-Virginica이면 1 아니면 0
```

```
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression(random_state=42)
log_reg.fit(X, y)
```

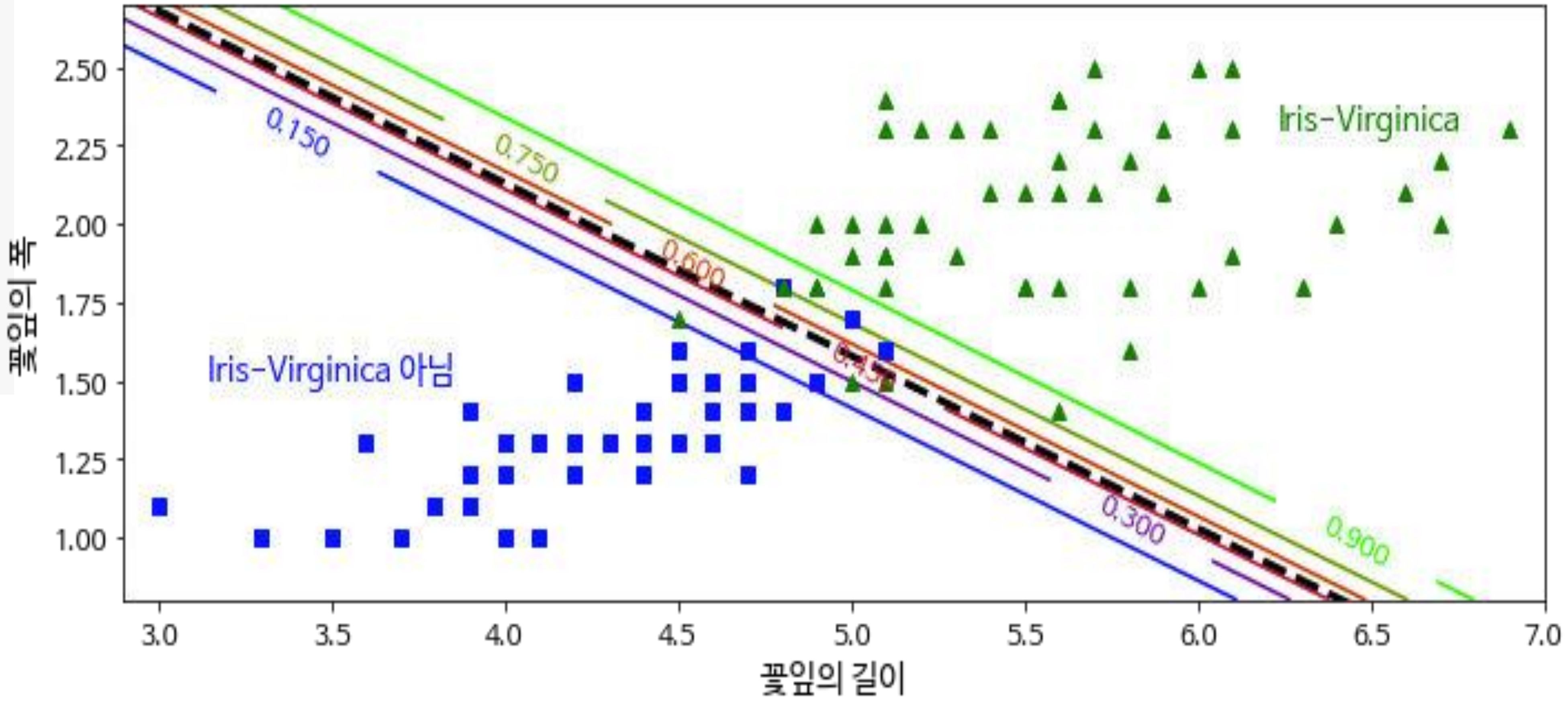
```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_proba = log_reg.predict_proba(X_new)
plt.plot(X_new, y_proba[:, 1], "g-", label="Iris-Virginica")
plt.plot(X_new, y_proba[:, 0], "b--", label="Not Iris-Virginica")
```



# 로지스틱회기 : 붓꽃(Iris) 예 2

```python
from sklearn.linear_model import LogisticRegression

X = iris["data"][:, (2, 3)]  # petal length, petal width
y = (iris["target"] == 2).astype(np.int)

log_reg = LogisticRegression(C=10**10, random_state=42)
log_reg.fit(X, y)
```

**penalty='L2'가 기본**
C를 매우 큰 값으로 했으므로
규제 사용 안 한 것




$$J(\theta) = C\frac{1}{m}\sum_{i=1}^{m}\log\left(e^{-y^{(i)}(\theta^T\cdot \mathrm{x}^{(i)})}+1\right)+\sum_{i=1}^{m}\theta^2$$

**사이킷런 LogisticRegression에서 사용하는 비용함수**

특징 "꽃잎의 폭" 하나만 사용했을 때 결정경계는?

특징을 2개 사용한 것이 더 잘 분류하는가?

# 소프트맥스 회귀 (Softmax Regression)

- Softmax Regression == Multinomial Logistic Regression
- 로지스틱 회귀를 사용한 다중 분류
- 각 클래스의 점수 $s_k(\mathrm{x})$에 softmax함수를 인가해서 그 클래스에 속할 확률을 계산
- 선형회귀 경우 $s_k(\mathrm{x}) = \theta_k^T \cdot \mathrm{x}$
- softmax 함수 $\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \dfrac{\exp\left(s_k(\mathbf{x})\right)}{\sum_{j=1}^{K} \exp\left(s_j(\mathbf{x})\right)}$

K=2 때는 Logistic regression 함수와 동일

- 예측(추론) : softmax함수 값이 가장 큰 클래스로 분류

$$\hat{y} = \underset{k}{\operatorname{argmax}}\ \sigma(\mathbf{s}(\mathbf{x}))_k = \underset{k}{\operatorname{argmax}}\ s_k(\mathbf{x}) = \underset{k}{\operatorname{argmax}}\left(\theta_k{}^T \cdot \mathbf{x}\right)$$

- 학습 : 비용함수 : Cross entropy function(크로스엔트로피 함수)

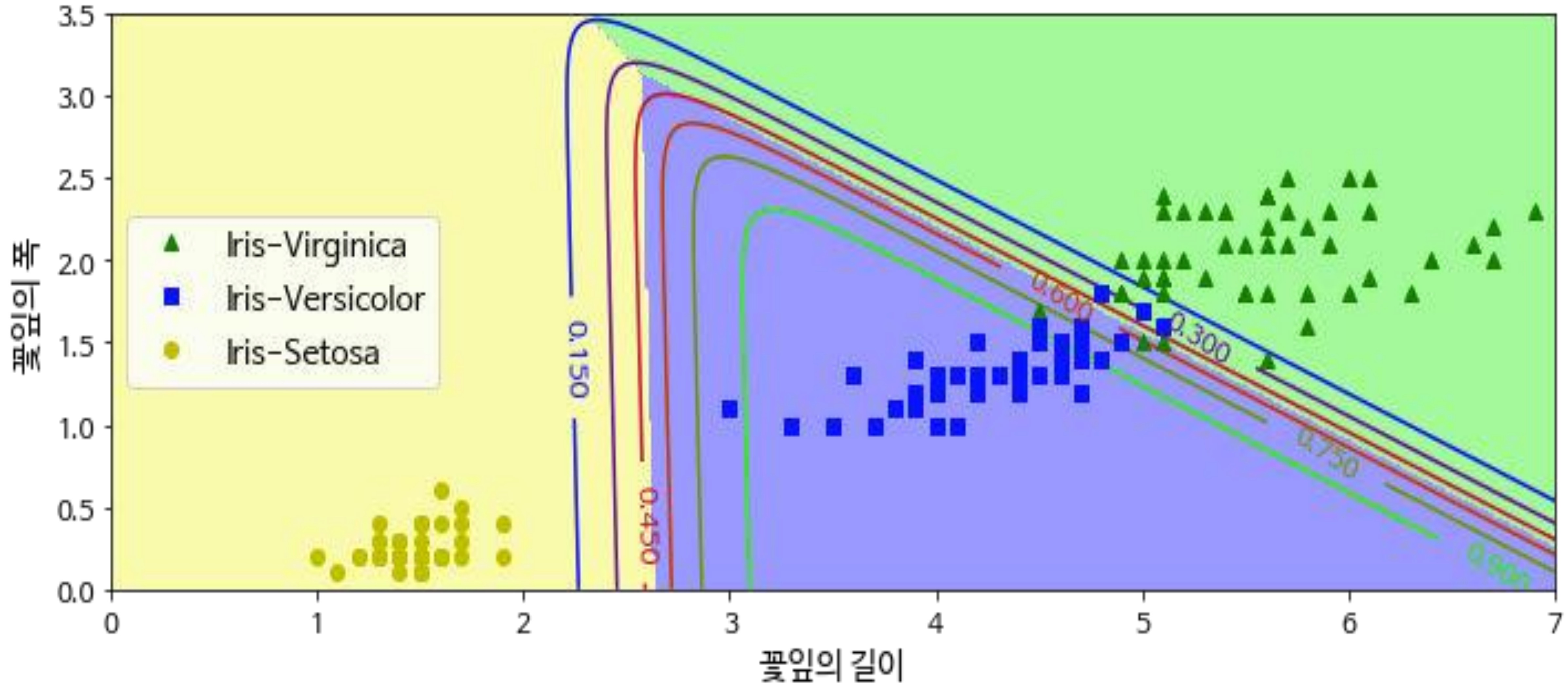
$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(\hat{p}_k^{(i)}\right)$$

# 소프트맥스 회귀의 예

Softmax.
기본값은
‘OvR’

quasi-Newton method중 하나
머신러닝에서 자주 이용. 기본값

```
X = iris["data"][:, (2, 3)]  # 꽃잎 길이, 꽃잎 넓이
y = iris["target"]

softmax_reg = LogisticRegression(multi_class="multinomial",solver="lbfgs", C=10, random_state=42)
softmax_reg.fit(X, y)
```



감사합니다