

핸즈온 머신러닝

4장. 모델 훈련

박해선(옮긴이)

haesun.park@tensorflow.blog
<https://tensorflow.blog>

신경망의 빌딩 블록

선형 회귀 → 로지스틱 회귀 → 층 → 신경망

$$\hat{y} = \theta_1 \times x + \theta_0$$

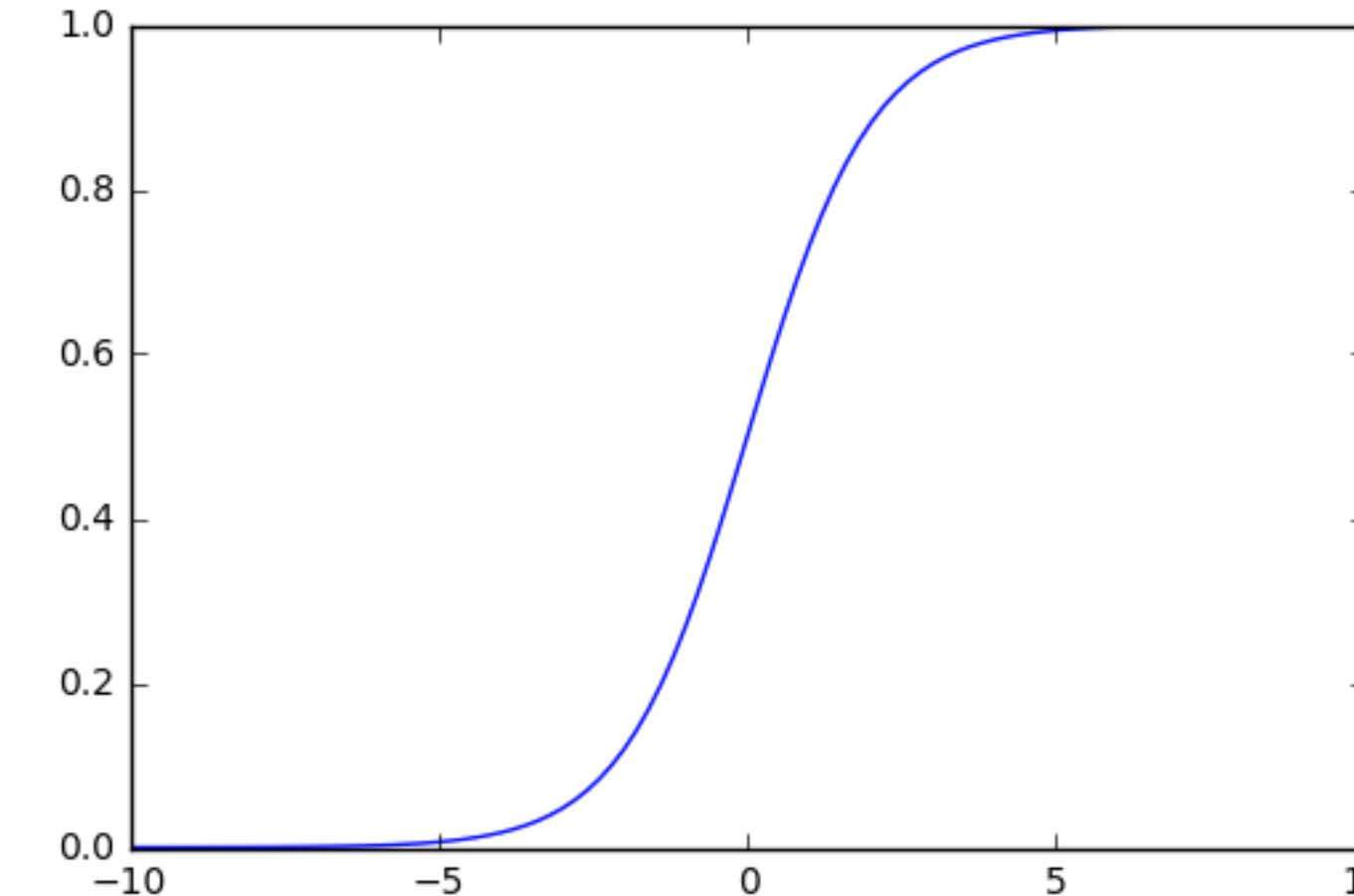
-무한 ~ 0 ~+무한

y > 0 : 양성(True)
y < 0 : 음성(False)

$$\hat{y} = \sigma(\theta_1 \times x + \theta_0)$$

0 ~ 0.5 ~ 1

y > 0.5 : 양성(True)
y < 0.5 : 음성(False)

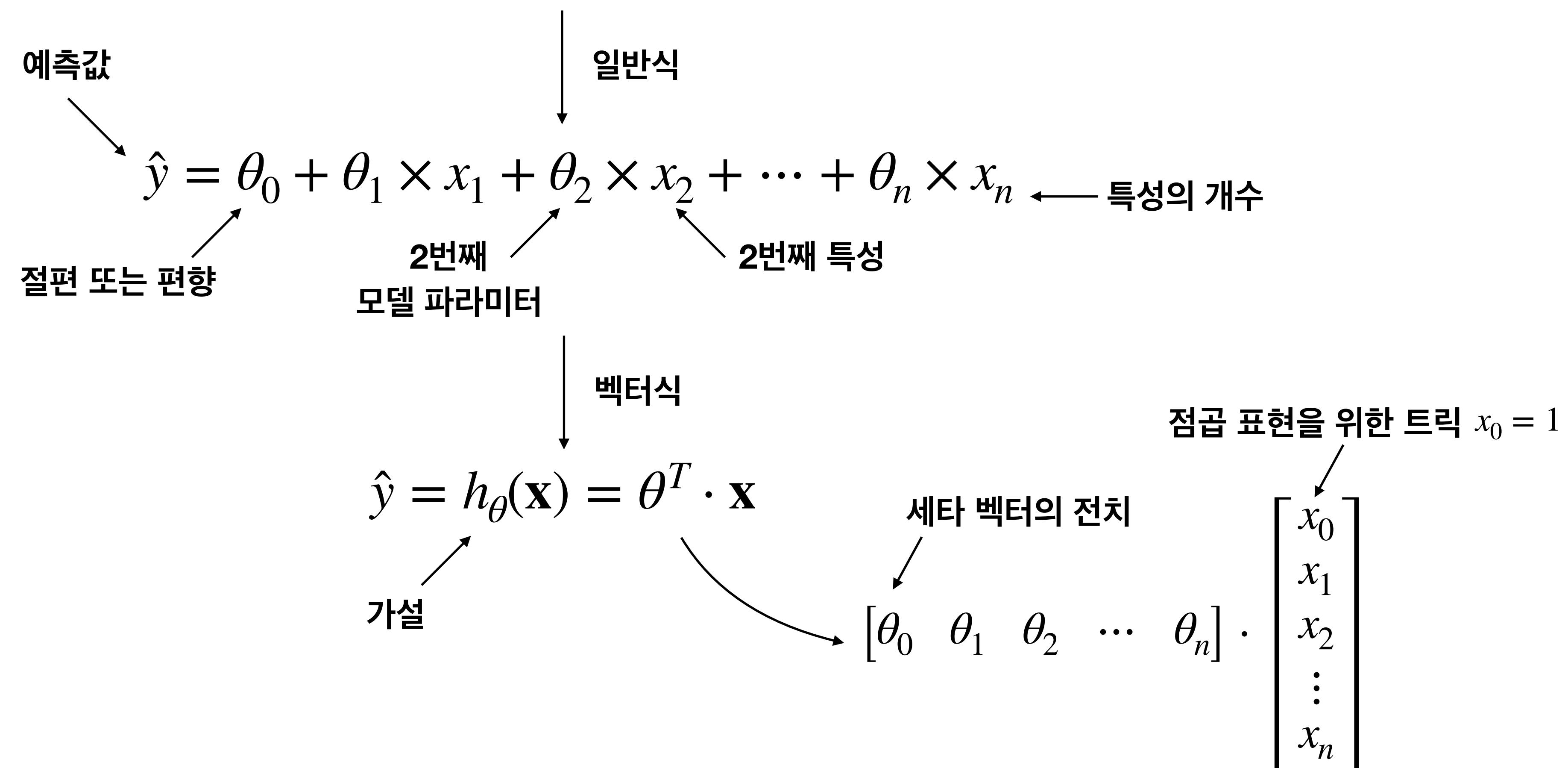


선형 회귀 방법

- 비용 함수의 해를 해석적으로 구합니다(정규 방정식).
- 경사 하강법(Gradient Descent)으로 비용 함수가 최소가 되는 해를 조금씩 찾아갑니다(배치, 미니배치, 확률적).

선형 회귀 방정식

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



선형 회귀 비용 함수

평균 제곱 오차(RMSE에서 제곱근을 뺀 것)

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

샘플 개수

타깃 or 레이블

정규 방정식

- Normal Equation

증명: <https://goo.gl/WkNEXH>

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

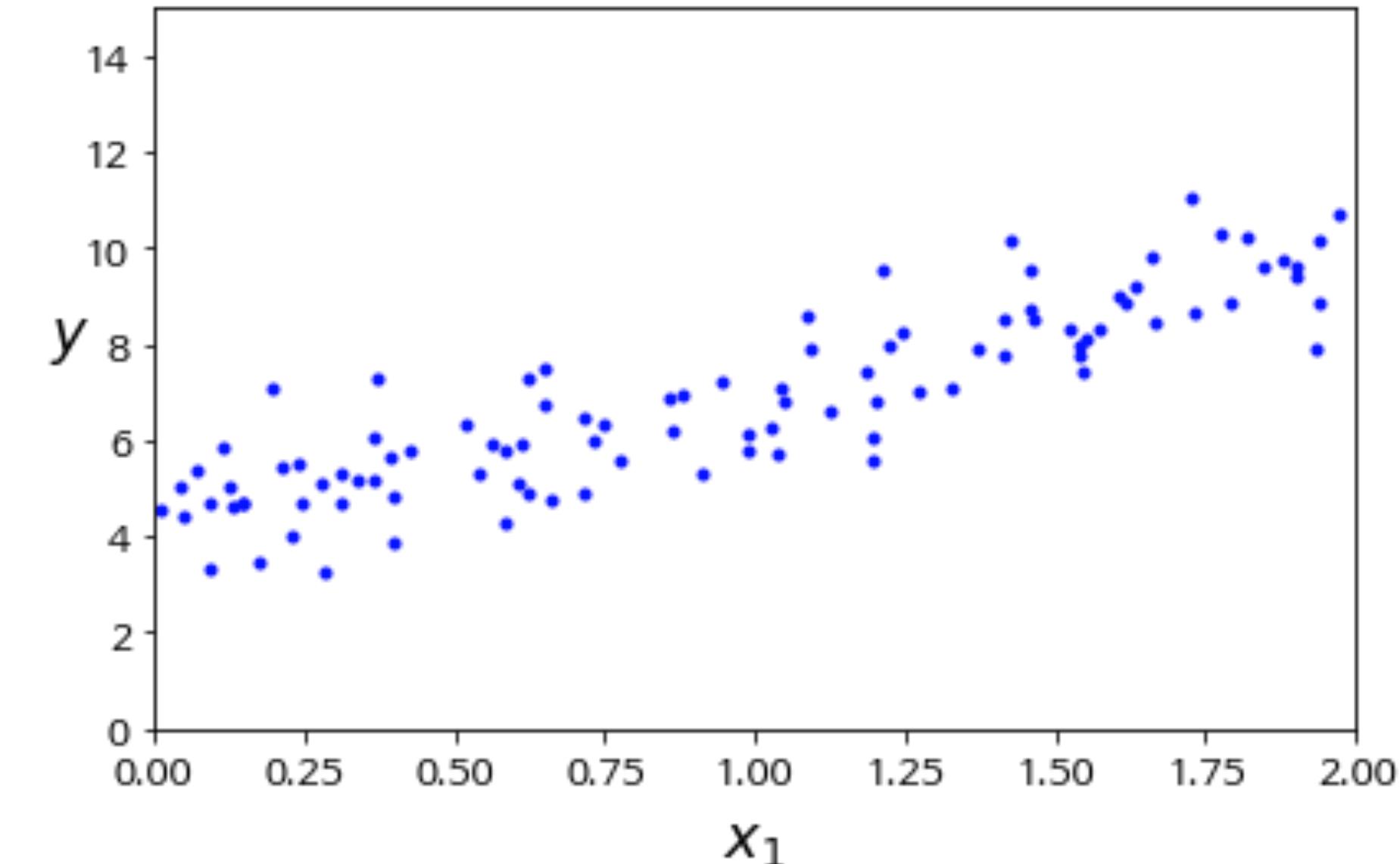
$$\mathbf{X} = \begin{bmatrix} (\mathbf{x}^{(1)})^T \\ (\mathbf{x}^{(2)})^T \\ \vdots \\ (\mathbf{x}^{(1999)})^T \\ (\mathbf{x}^{(20000)})^T \end{bmatrix} = \begin{bmatrix} -118.29 & 33.91 & 1,416 & 38,372 \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

- 샘플 데이터

$$y = 4 + 3x + \text{가우시안 노이즈}$$

```
import numpy as np  
  
x = 2 * np.random.rand(100, 1)  
y = 4 + 3 * x + np.random.randn(100, 1)
```

(100, 1) → y도 2차원 배열이 됩니다.



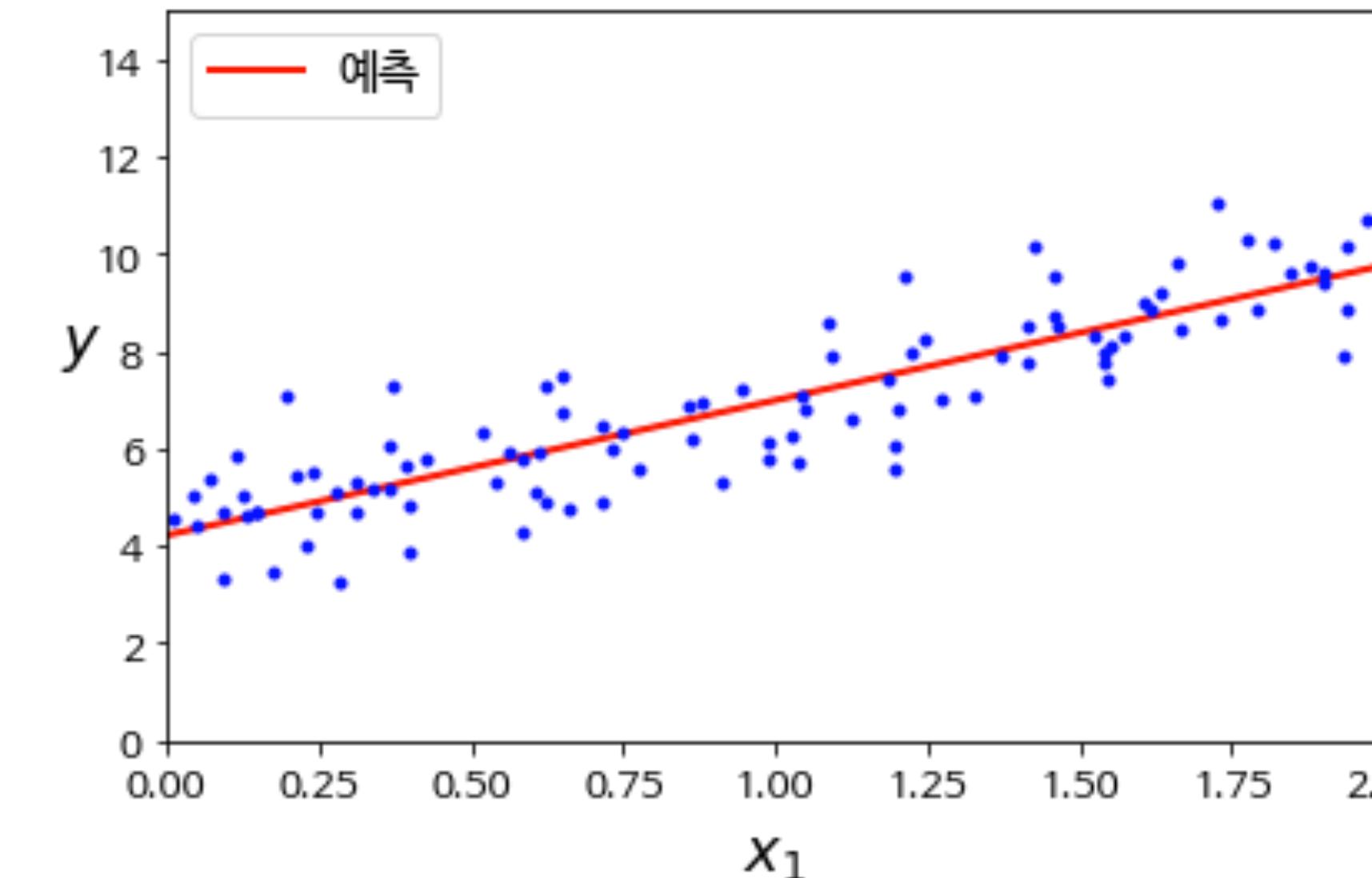
Using np.linalg.inv()

```
(100, 2)  
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)  
(100, 1) np.hstack((np.ones((100, 1)), X))
```

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

편향
기울기

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



Using np.linalg.pinv()

- SVD 기법으로 유사역행렬을 구하는 pinv() 함수를 사용할 수 있습니다.

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

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

$$\hat{\theta} = X^{-1} \cdot y$$

$$(n+1, 1) = (n+1, m) \times (m, 1)$$

Using LinearRegression

- LinearRegression은 lapack 라이브러리를 래핑한 `scipy.linalg.lstsq()` 함수를 사용합니다.

```
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
```

80.65845639670533

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

R^2

0.7692735413614223

Using lstsq()

```
import scipy
theta_best_svd, residuals, rank, s = scipy.linalg.lstsq(X_b, y)
theta_best_svd, residuals
```

(array([[4.21509616],
 [2.77011339]]), array([80.6584564])) 오차의 제곱 합

편향, 기울기

scipy의 lstsq 함수를 직접 호출

```
theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_b, y, rcond=1e-6)
theta_best_svd, residuals
```

(array([[4.21509616],
 [2.77011339]]), array([80.6584564]))

특이값의 최소를 지정, None으로 지정

np.linalg는 scipy.linalg의 간소화 버전

정규 방정식의 계산 복잡도

- 역행렬(inverse matrix) 계산이 정규 방정식의 복잡도를 좌우합니다.

$$(n+1, m) \times (m, n+1) = (n+1, n+1)$$

샘플의 수에 대해서는 선형적으로 늘어납니다.

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

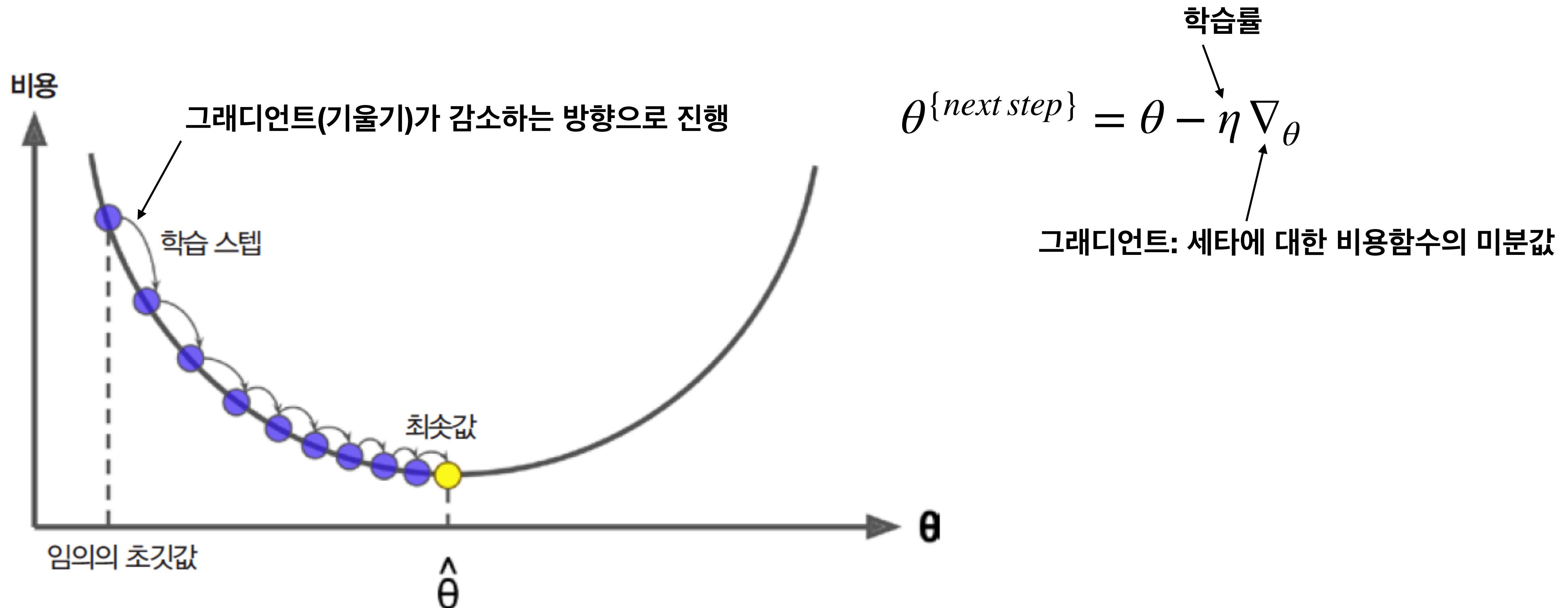
특성의 수가 2배로 늘어 나면 계산 복잡도는 5~8배로 늘어납니다.

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

* **scipy**의 **lstsq** 함수는 **SVD** 방법을 사용하며 $O(n^2)$ 의 복잡도를 가집니다.

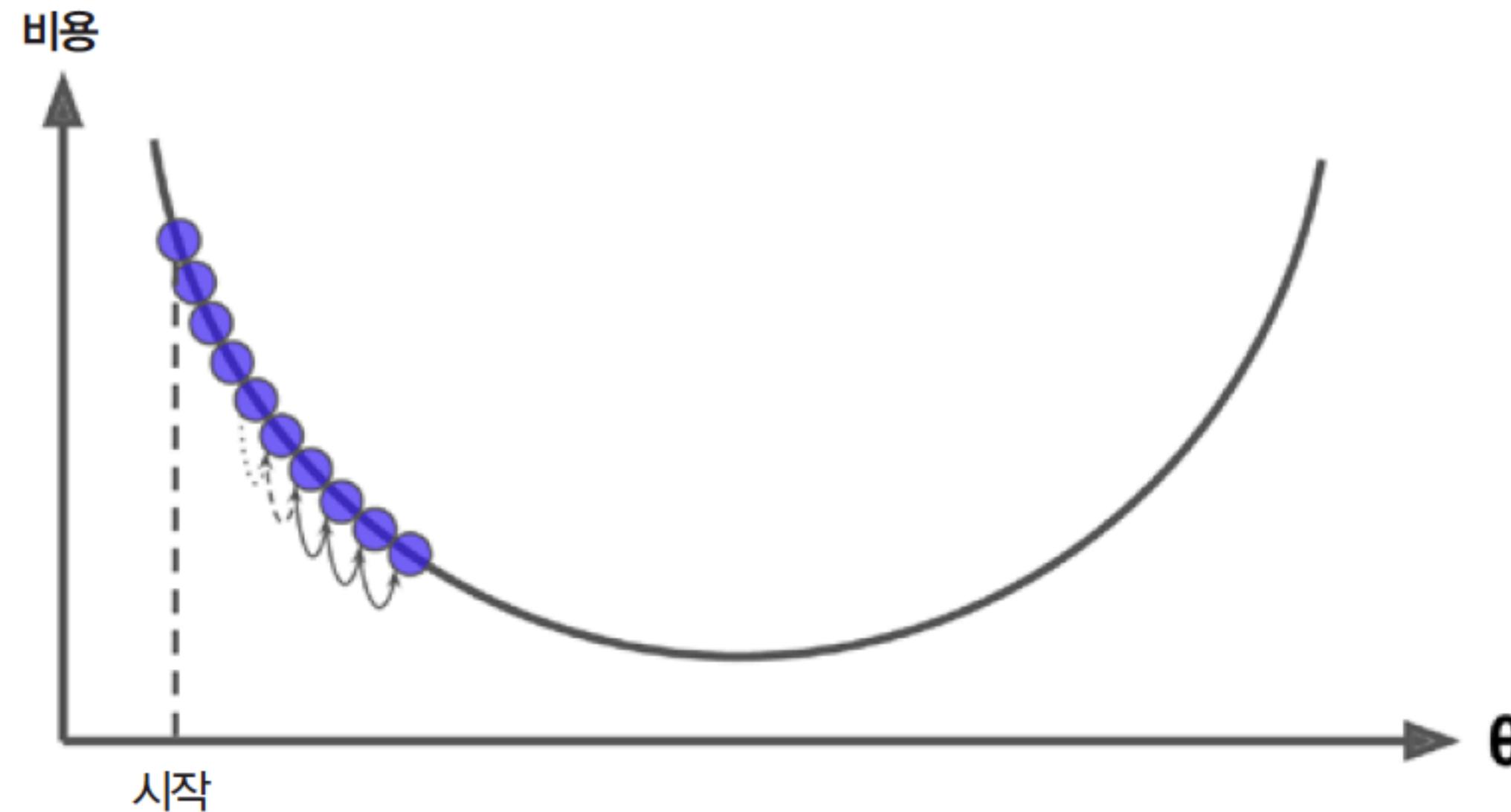
경사 하강법

- Gradient Descent(GD): 모델 파라미터를 조금씩 수정하면서 비용 함수의 최소값을 찾는 방법입니다.

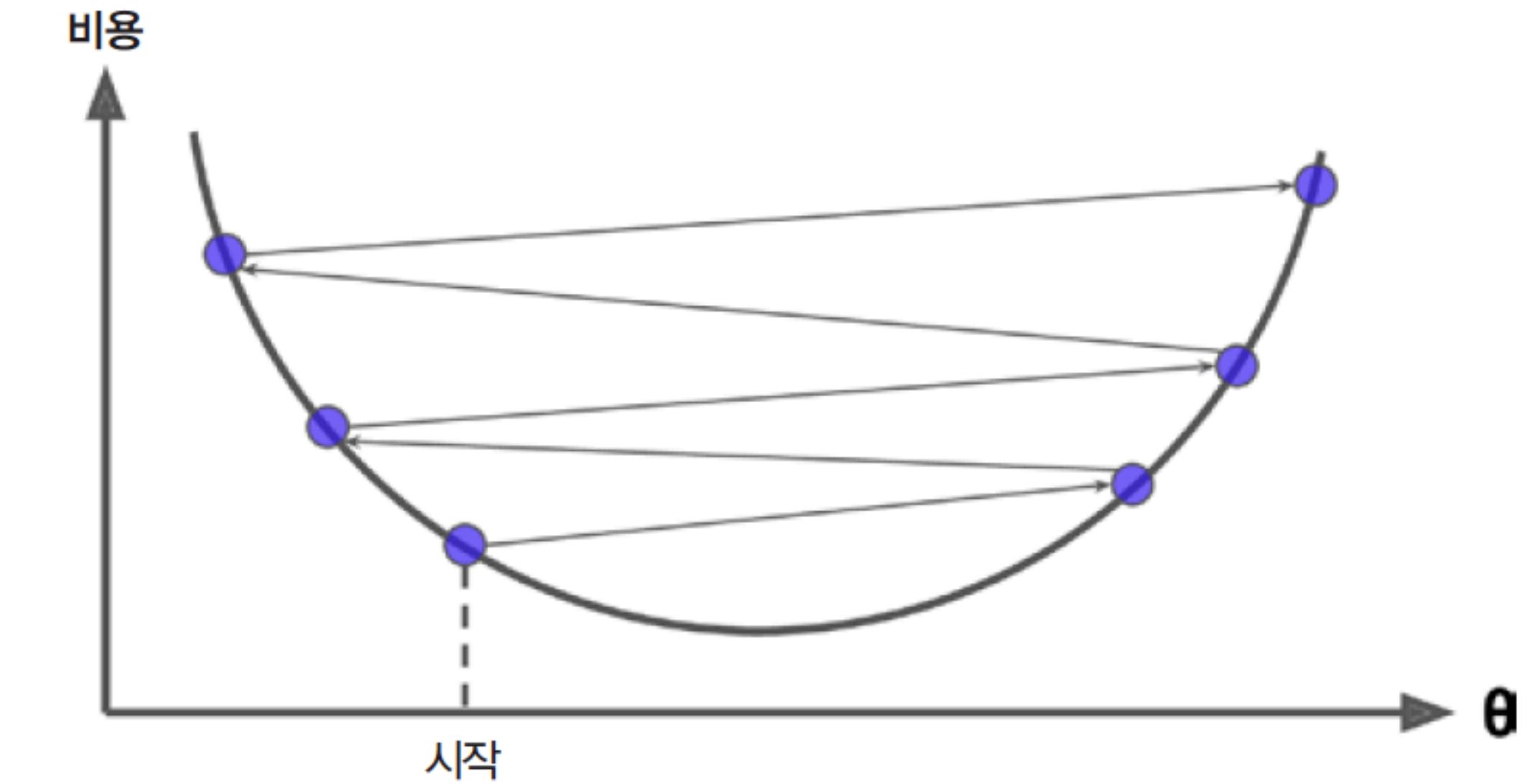


학습률

- 학습률은 그래디언트 적용량을 조정하는 하이퍼파라미터입니다.

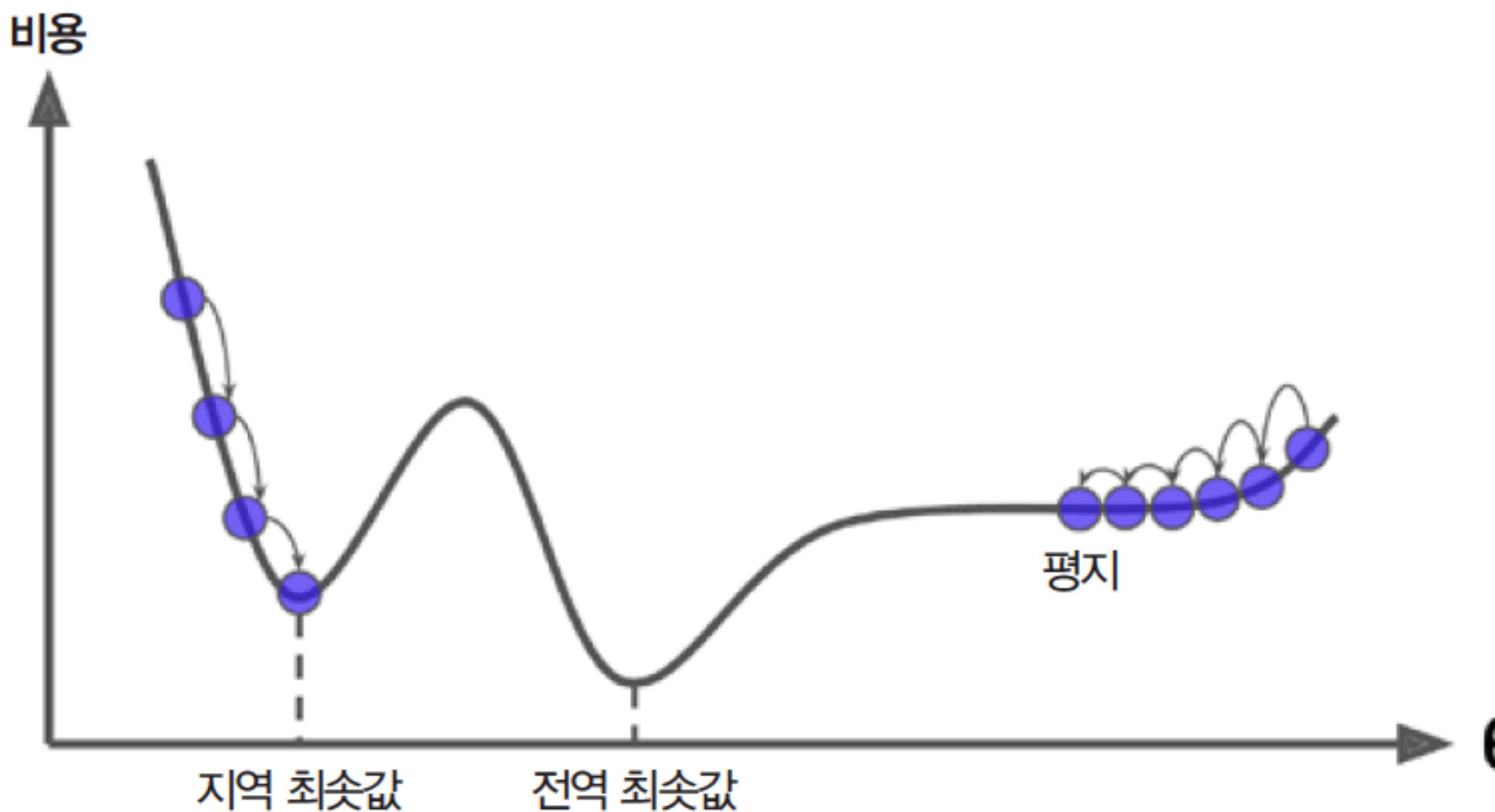


학습률이 너무 작을 때



학습률이 너무 클 때

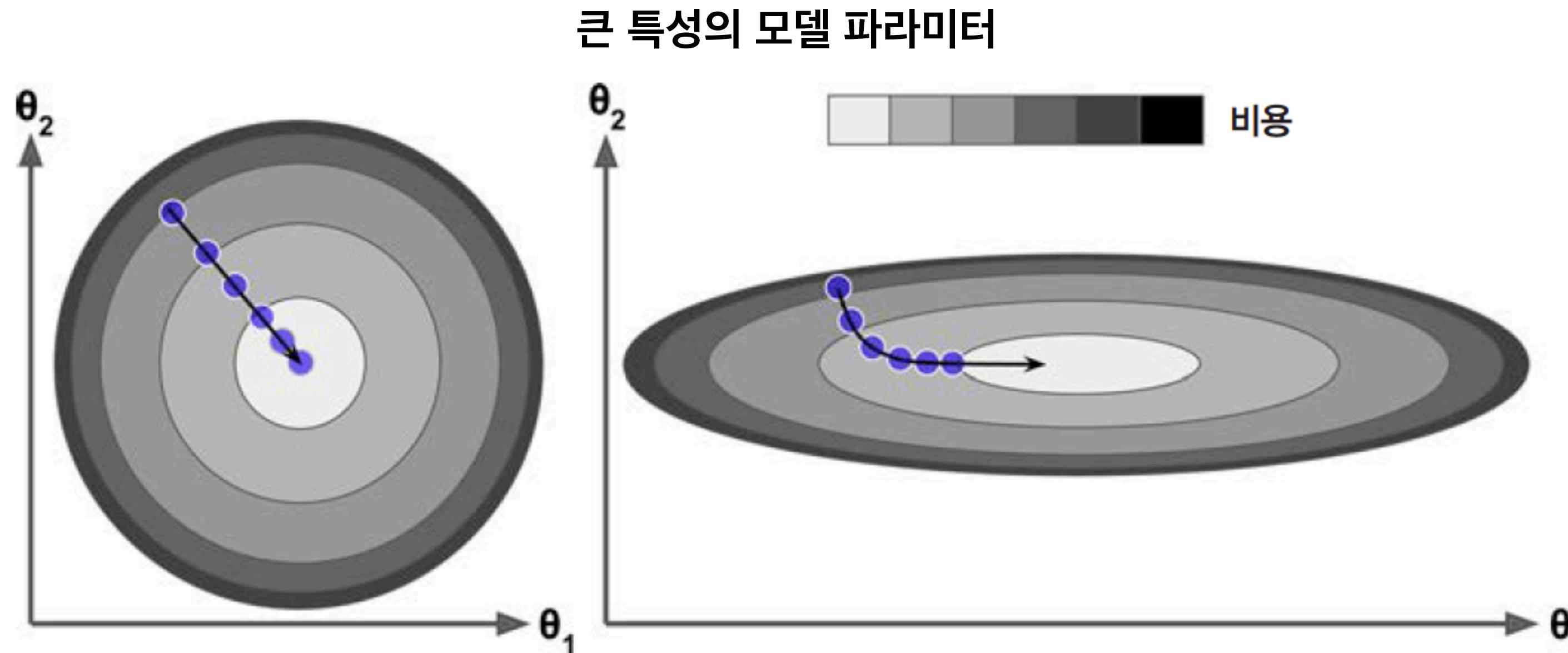
지역 최솟값, 전역 최솟값



- MSE는 볼록 함수이므로 지역 최솟값이 없고 기울기가 일정하게 변합니다.

특성 스케일

- 특성의 스케일이 다르면 모델 파라미터에 따른 비용 함수의 변화율이 달라집니다.

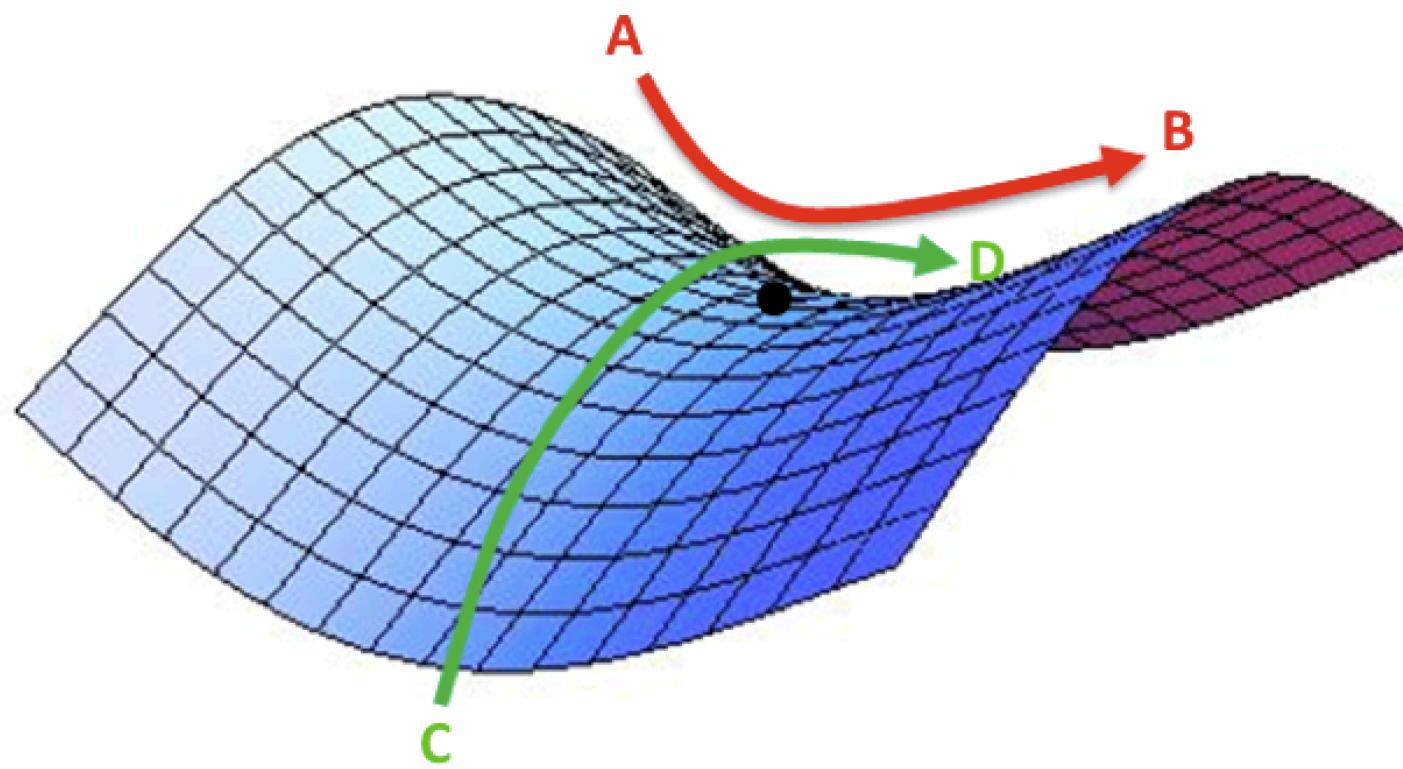


스케일링(StandardScaler)된 데이터셋의 비용 함수

작은 특성의 모델 파라미터

지역 최솟값 보다는 말안장

- 비용 함수의 기울기가 모두 0인 어떤 지점이 지역 최솟값일 확률은 차원이 높을수록 크게 줄어듭니다($100\text{차원} \rightarrow 2^{-100}$).
- 기울기가 모두 0인 지점은 대부분 말안장(saddle point)입니다. 만약 최솟값을 찾았다면 전역 최솟값과 같은 매우 훌륭한 모델 파라미터일 것입니다.



그래디언트=미분

- 각 세타 값에 대해 비용 함수를 미분합니다(편미분, 편도함수).
- 경사 하강법의 비용 함수는 미분 가능해야 합니다.

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

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

* 미분결과를 간소하게 표현하기 위해 $1/2m$ 을 곱하기도 합니다.

배치 경사 하강법

- 전체 훈련 세트를 사용하여 그래디언트를 계산합니다.
- 비용 함수의 최솟값에 안정적으로 수렴하지만 계산 비용이 큽니다.

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

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

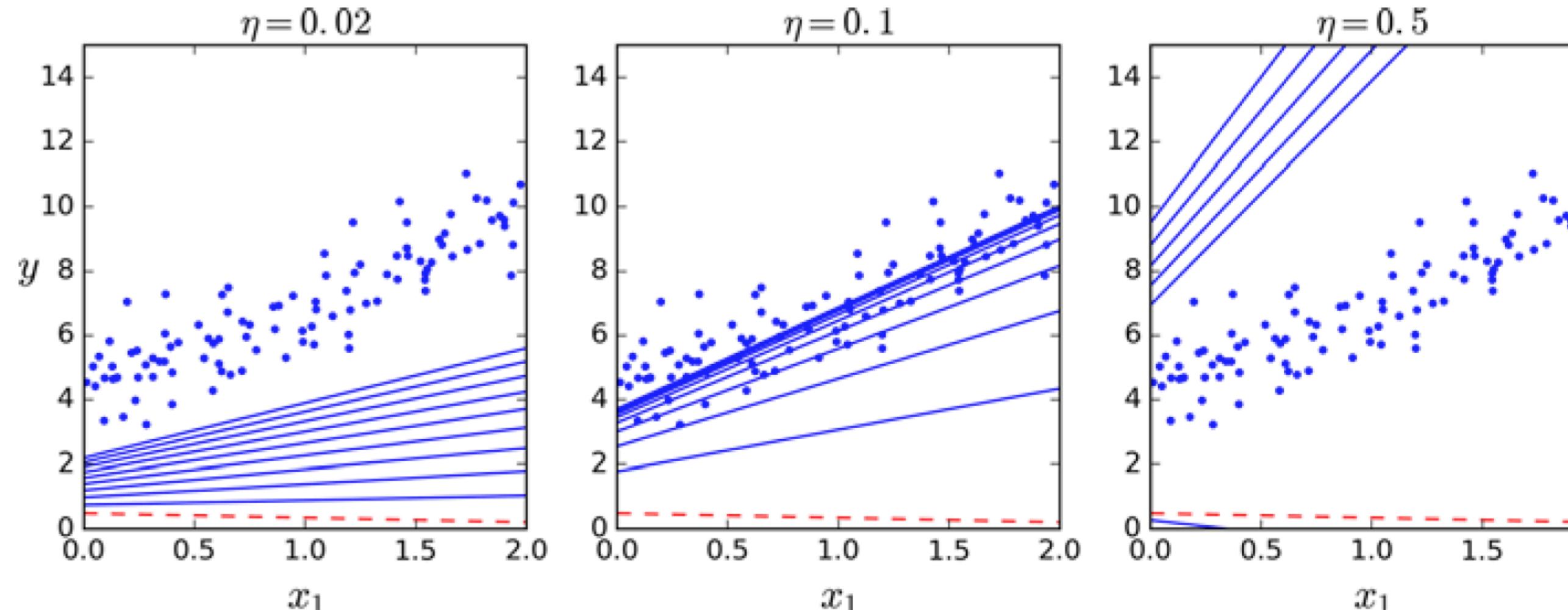
경사 하강법 범파이 구현

```
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
```

```
theta
```

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



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

정규 방정식의 해

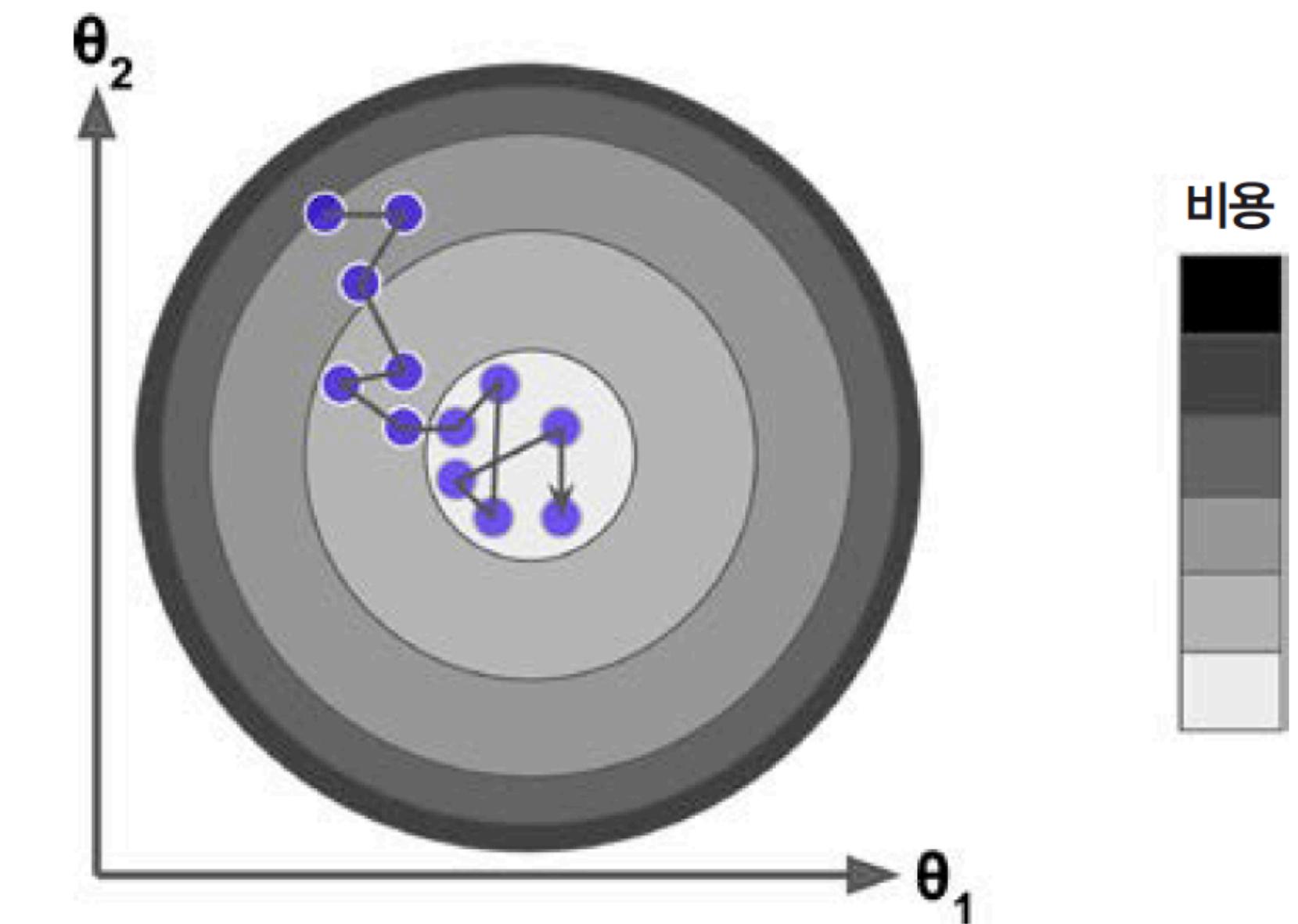
```
array([[4.21509616],
       [2.77011339]])
```

학습률과 반복횟수

- 알맞은 학습률을 찾으려면 반복 횟수를 제한하여 그리드 탐색을 수행하는 것이 좋습니다.
- 그래디언트 벡터가 어떤 허용 오차보다 작으면(수렴이라 가정할 수 있으면) 알고리즘 반복을 중지합니다.
- 학습률과 같은 하이퍼파라미터를 이론적으로 구할 수 있는 방법은 없습니다.
- 특히 딥러닝은 많은 하이퍼파라미터가 있기 때문에 과학보다는 예술에 가깝다고 말합니다.

확률적 경사 하강법(SGD)

- 확률적=무작위. 훈련 데이터에서 하나의 샘플을 무작위로 선택해 경사 하강법 단계를 진행합니다.
- 배치 경사 하강법보다 빠르고 아주 큰 데이터셋을 처리할 수 있습니다(외부 메모리 알고리즘으로 활용).
- 불안정하게 요동하면서 수렴하기 때문에 학습률을 점차 감소하는 것이 좋습니다(학습 스케줄링).
- 비용 함수의 최솟값에 빠지지 않을 가능성이 높습니다.



SGD 넘파이 구현

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

def learning_schedule(t):
    return t0 / (t + t1) ← 0.1에서부터 t가 커질수록 줄어듭니다.

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

for epoch in range(n_epochs):
    for i in range(m): ← 1에포크: 훈련 샘플 개수 만큼 반복
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1] ← 슬라이싱으로 2D 배열을 추출
        yi = y[random_index:random_index+1]          (하나의 샘플이 한 번 에포크 내에서 여러번 추출될 수 있습니다)
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
```

* 사이킷런의 **SGDClassifier**, **SGDRegressor**는 에포크마다 전체 샘플을 뒤섞은 후 순서대로 처리합니다.

Using SGDRegressor

- 사이킷런의 SGD 구현은 SGDRegressor(회귀), SGDClassifier(분류)입니다.

```
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())
```

에포크
규제 없음
1차원 배열을 기대합니다.

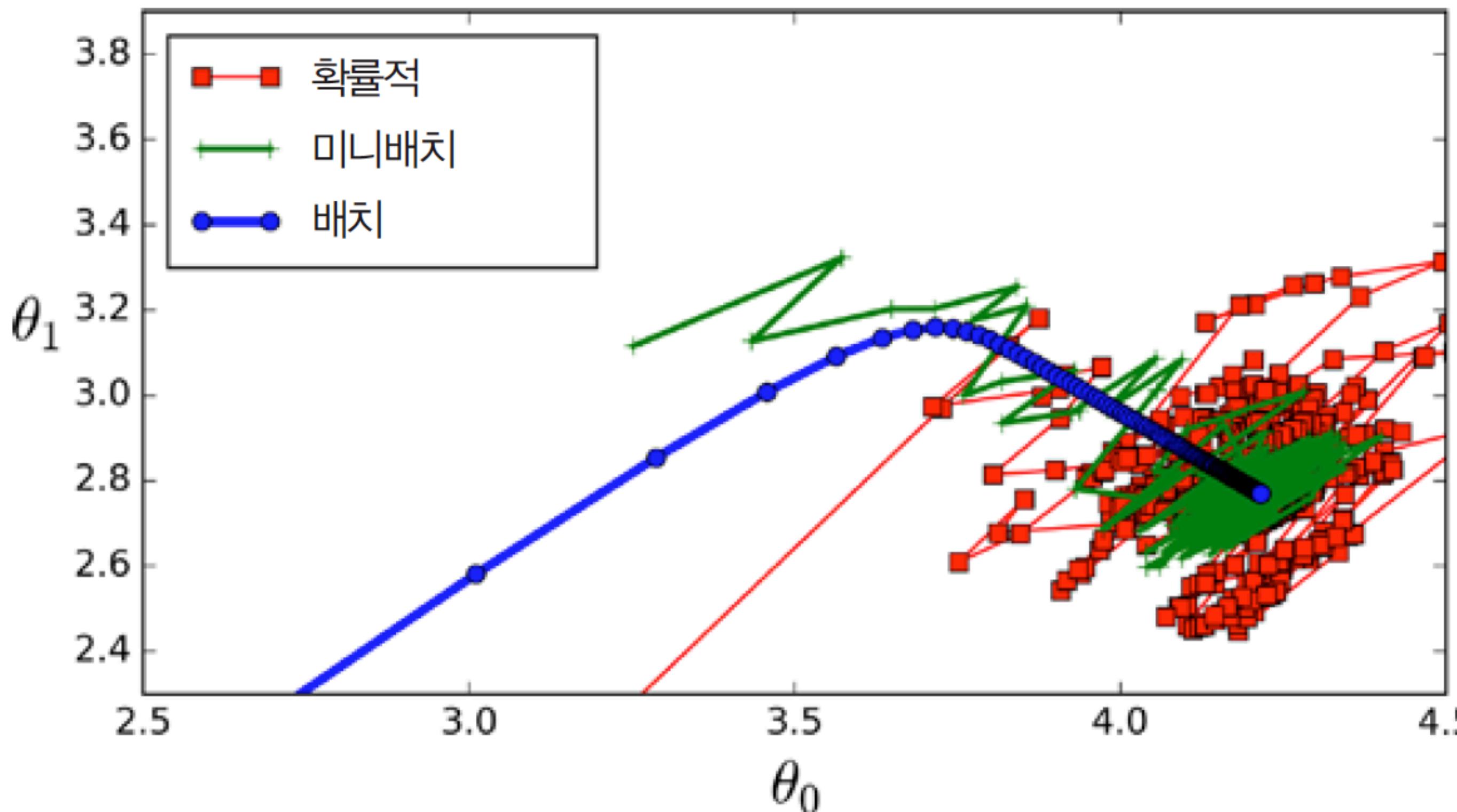
```
sgd_reg.intercept_, sgd_reg.coef_
(array([4.16782089]), array([2.72603052]))
```

정규 방정식의 해
array([[4.21509616],
 [2.77011339]])

기본값 learning_rate='invscaling'일 때
 $\eta^{(t)} = \frac{\text{eta0}}{t^{\text{power_t}}}$

미니배치 경사 하강법

- 미니배치(mini-batch): 확률적과 배치의 장단점을 절충합니다.



경사 하강법 비교

- 선형 회귀를 사용한 비교

알고리즘	m 이 클 때 학습 지원	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

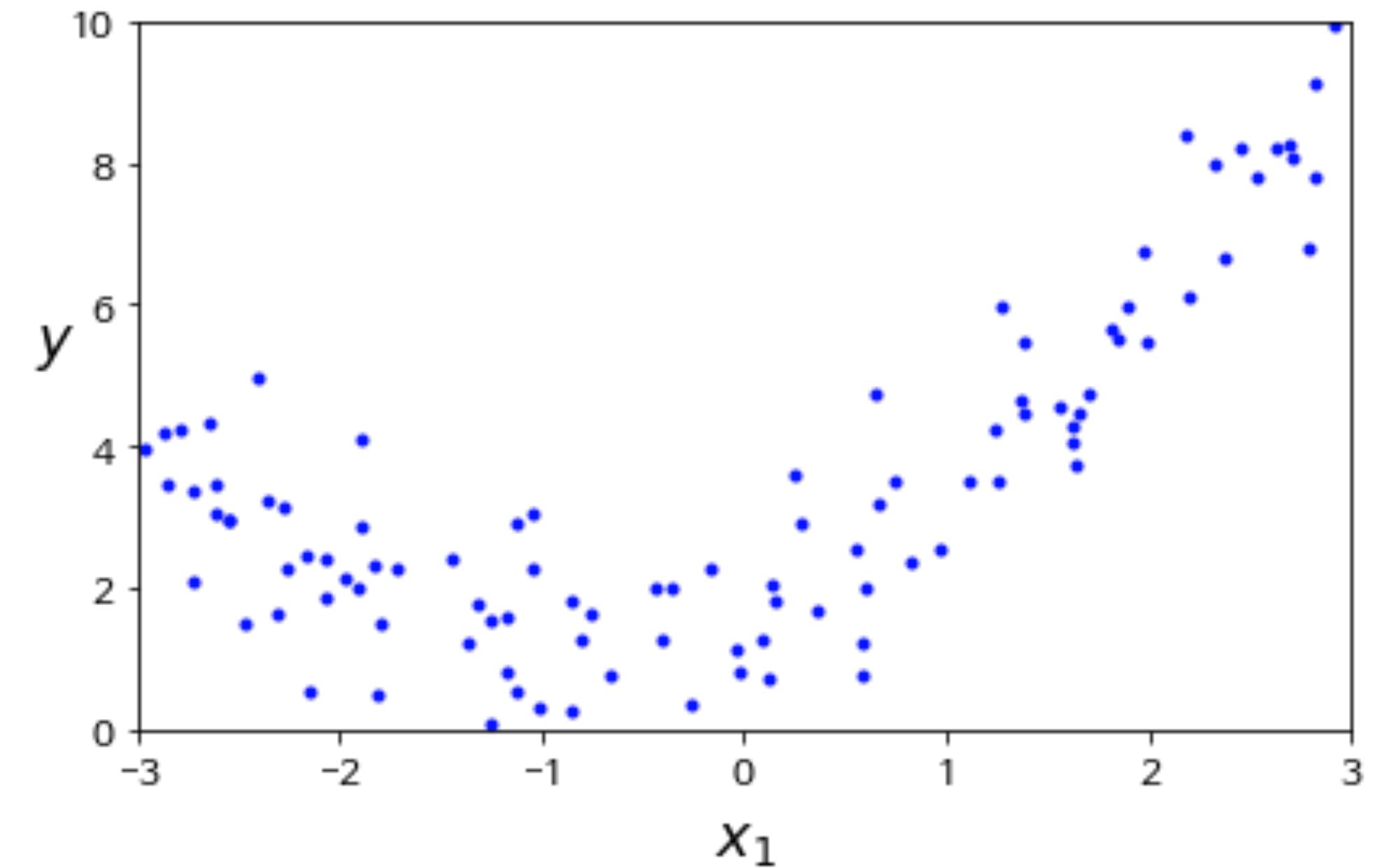
SGDRegressor에서 훈련 샘플을 조금씩 나누어 학습할 수 있는 `partial_fit()` 메서드도 SGD 방식으로 경사 하강법을 적용합니다.

다항 회귀

- 특성의 고차항을 만들어 추가합니다.
- 샘플 데이터

$$y = 0.5x^2 + x + 2 + \text{가우시안 노이즈}$$

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



PolynomialFeatures

- a, b 특성일 때 degree=2 이면 ab, a², b² 항이 추가됩니다. $\frac{(n + d)!}{d!n!}$ 개.
- Interaction_only=True로 하면 ab항만 추가됩니다.

```
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X)
X[0]
array([-0.75275929])
```

2차항 편향을 위한 1을 추가하지 않음

```
X_poly[0]
array([-0.75275929,  0.56664654])
```

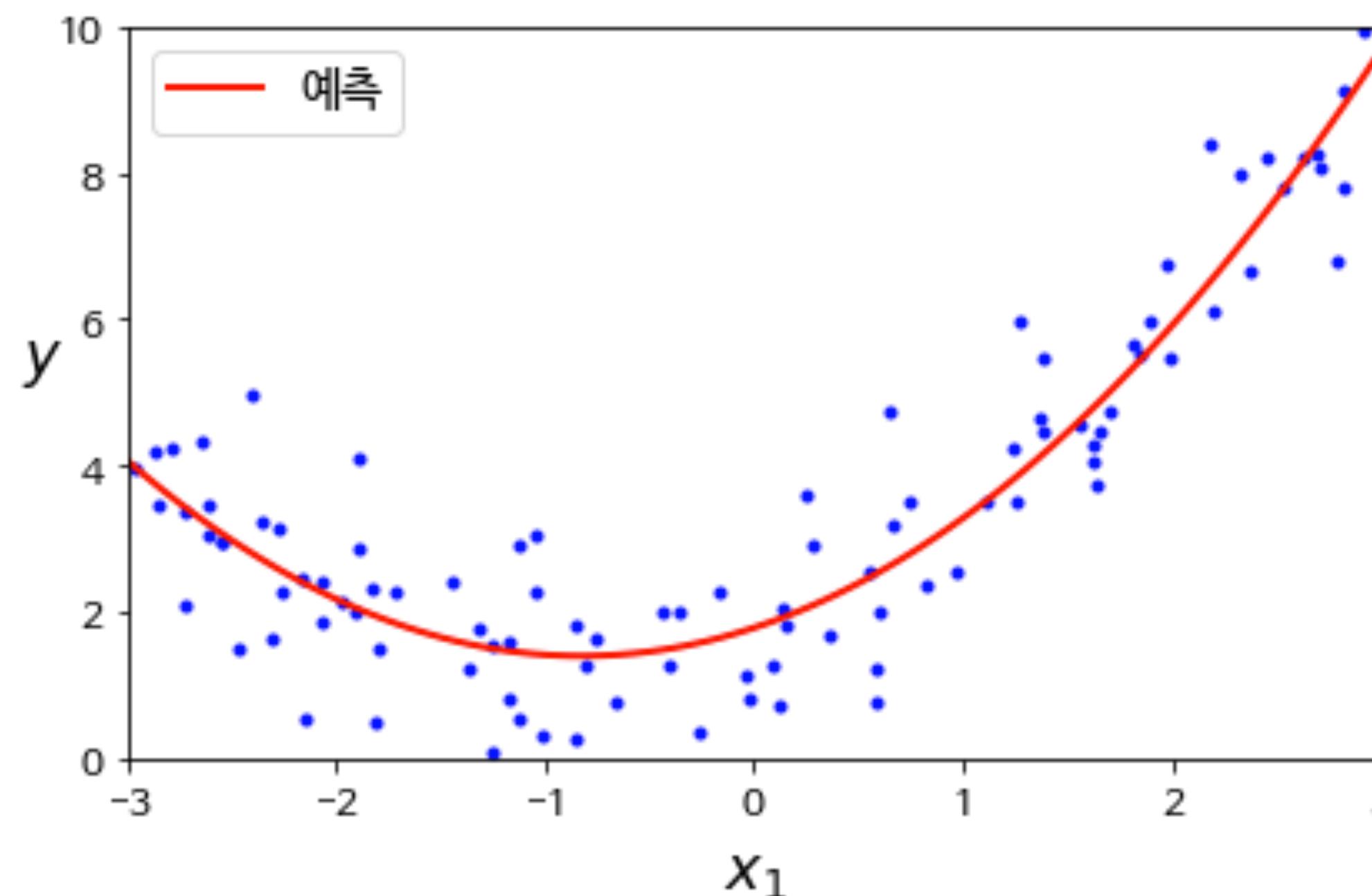
```
poly_features.get_feature_names()
['x0', 'x0^2']
```

특성 이름으로 확인

LinearRegression + X_poly

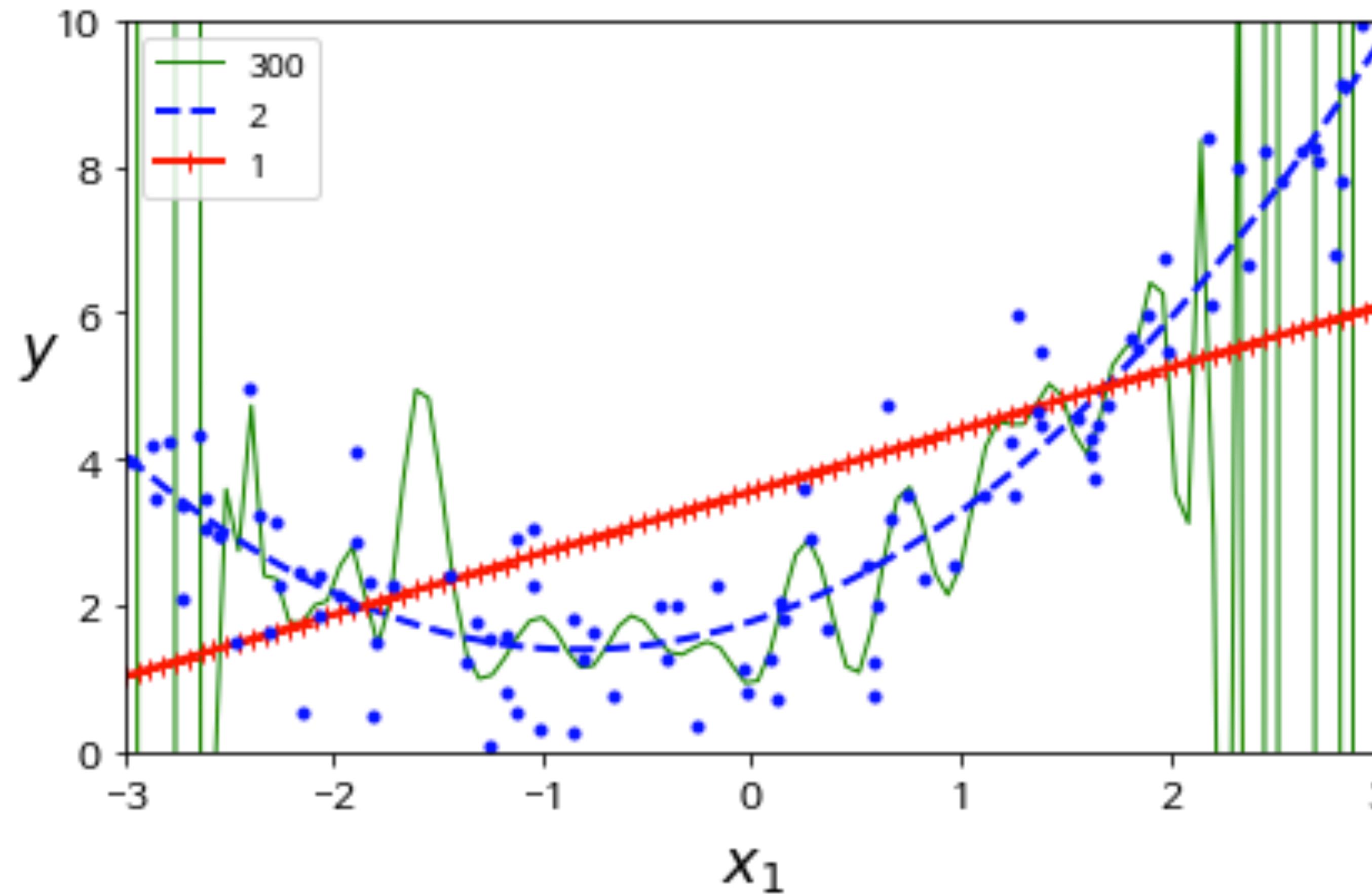
```
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
lin_reg.intercept_, lin_reg.coef_
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

$$y = 0.5x^2 + x + 2 + \text{가우시안 노이즈}$$



고차 다행 회귀

- 적절한 복잡도는 얼마일까요?



학습 곡선 그리기

- 훈련 세트와 검증 세트의 크기에 따라 오차를 측정합니다.

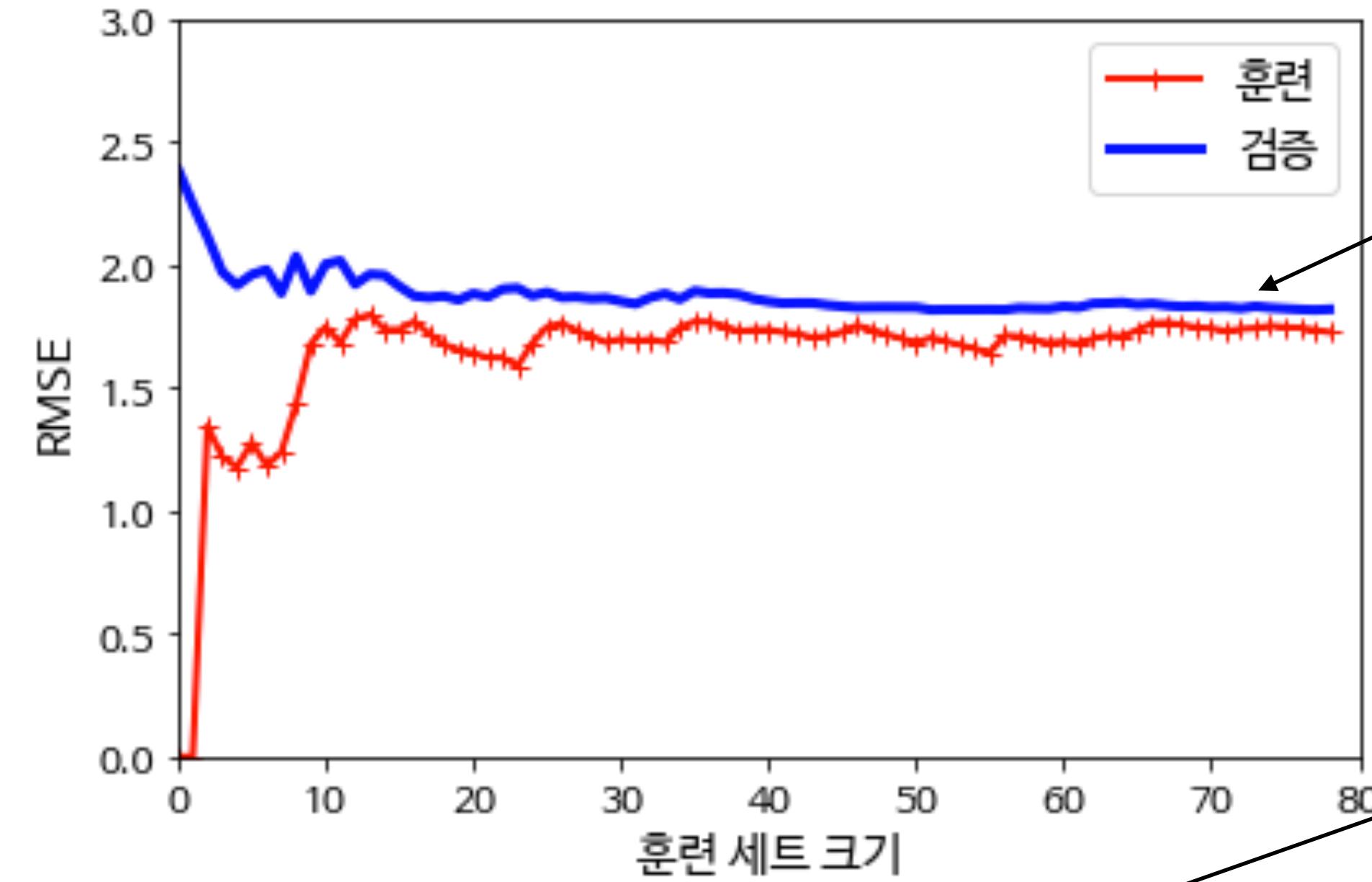
```
from sklearn.metrics import mean_squared_error ← 평균 제곱 오차
from sklearn.model_selection import train_test_split

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)): ← 훈련 세트 크기 1~m
        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))

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

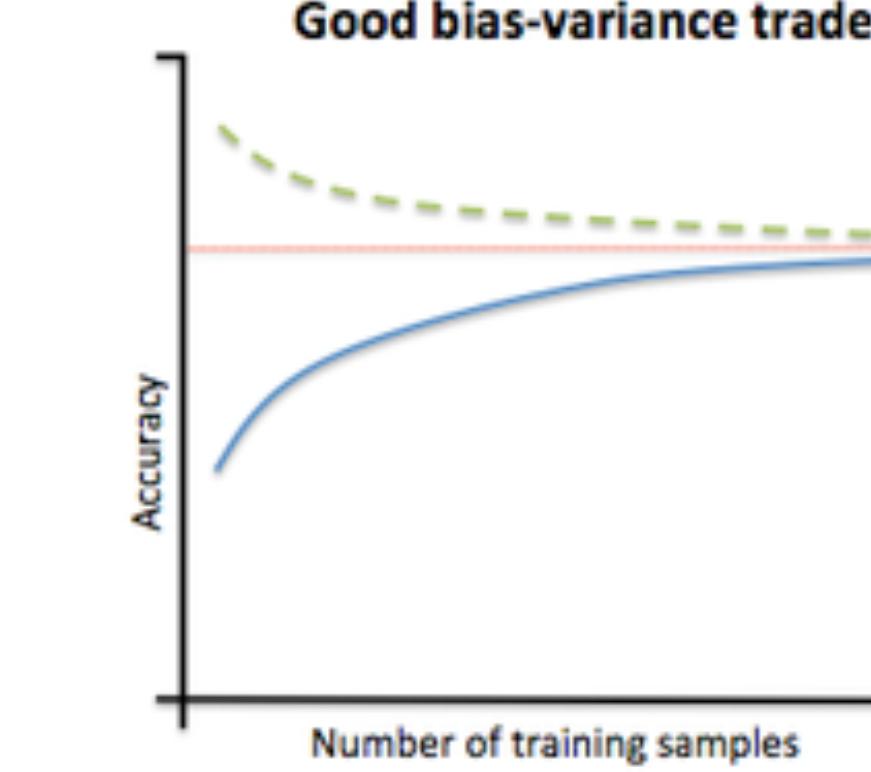
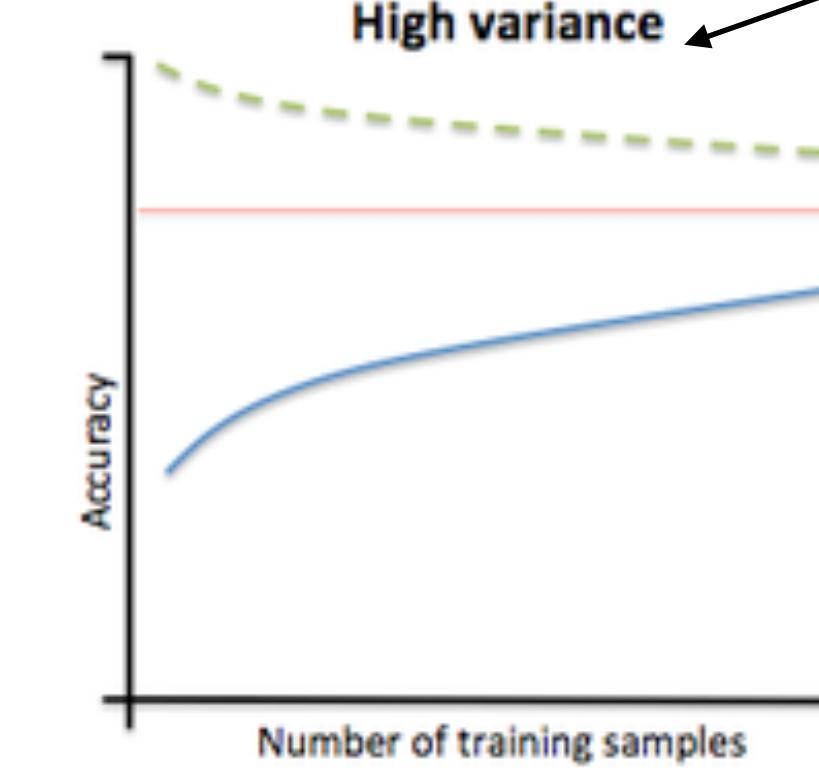
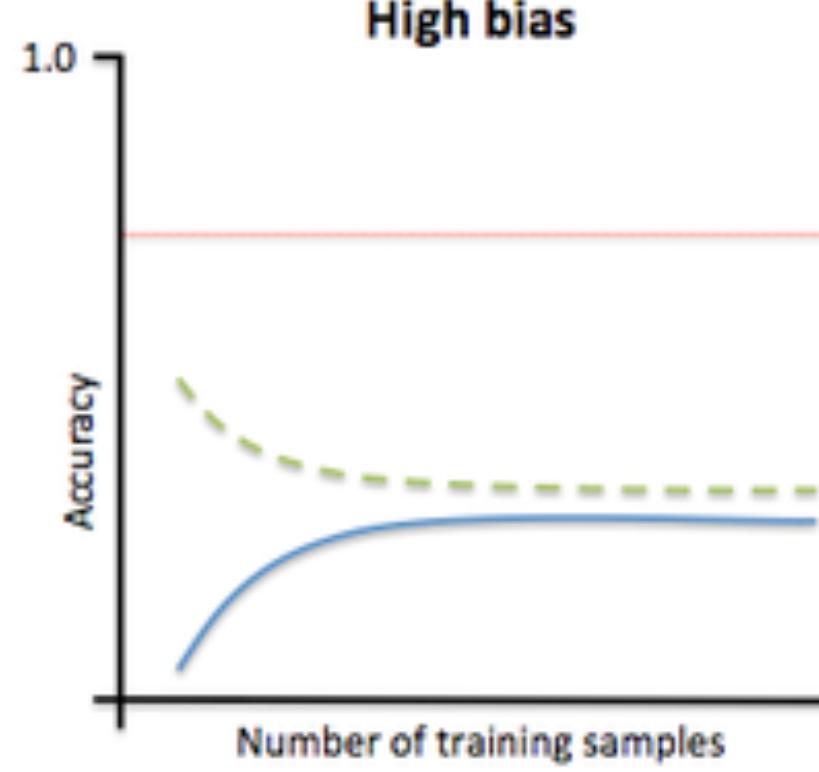
학습 곡선

모델 파라미터 bias와는 다릅니다.



과소적합(High bias)
훈련 샘플을 추가해도 소용이 없습니다.

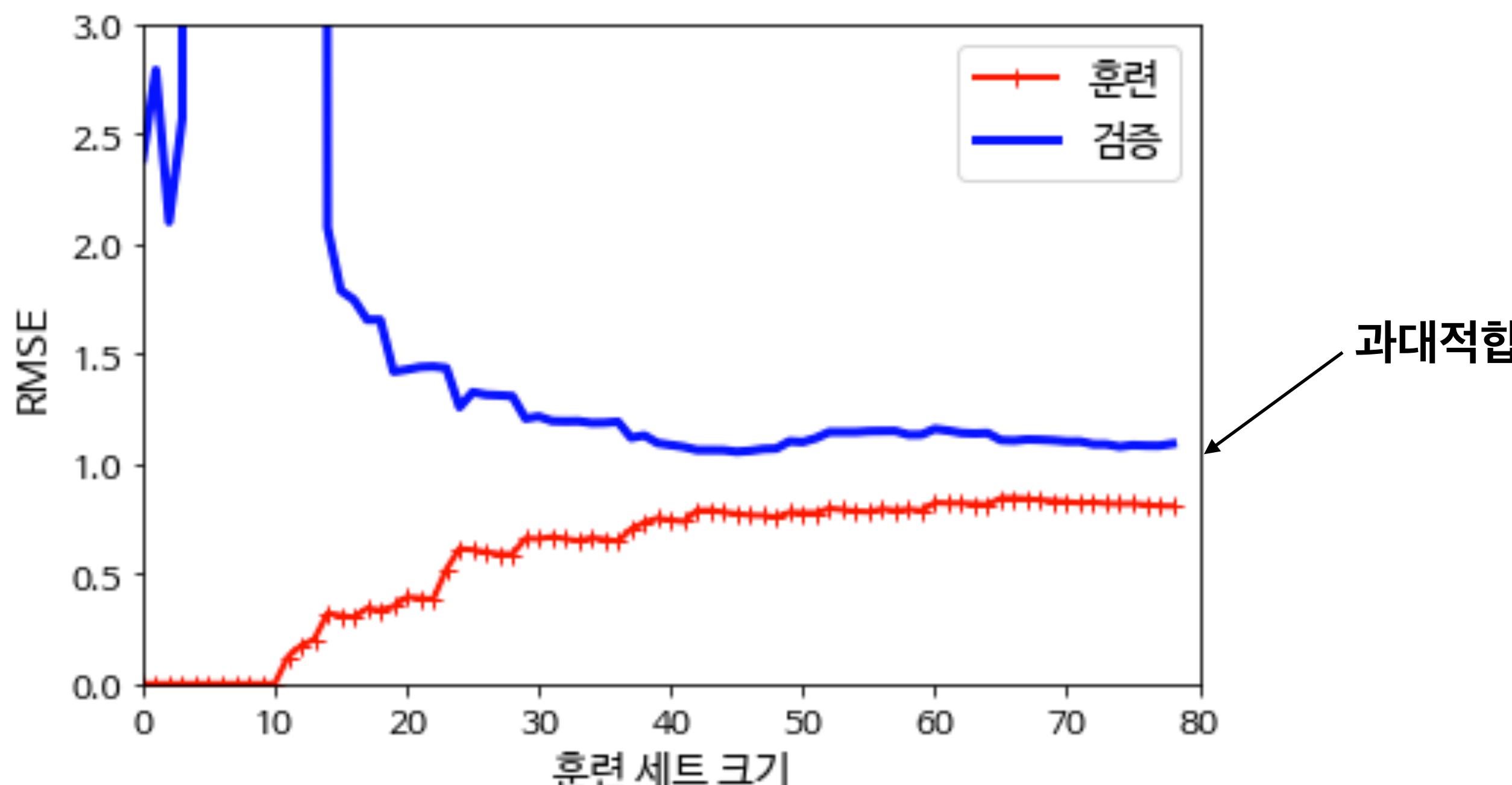
과대적합일 때는 훈련 샘플을
추가하면 도움이 됩니다



10차 다항 회귀의 학습 곡선

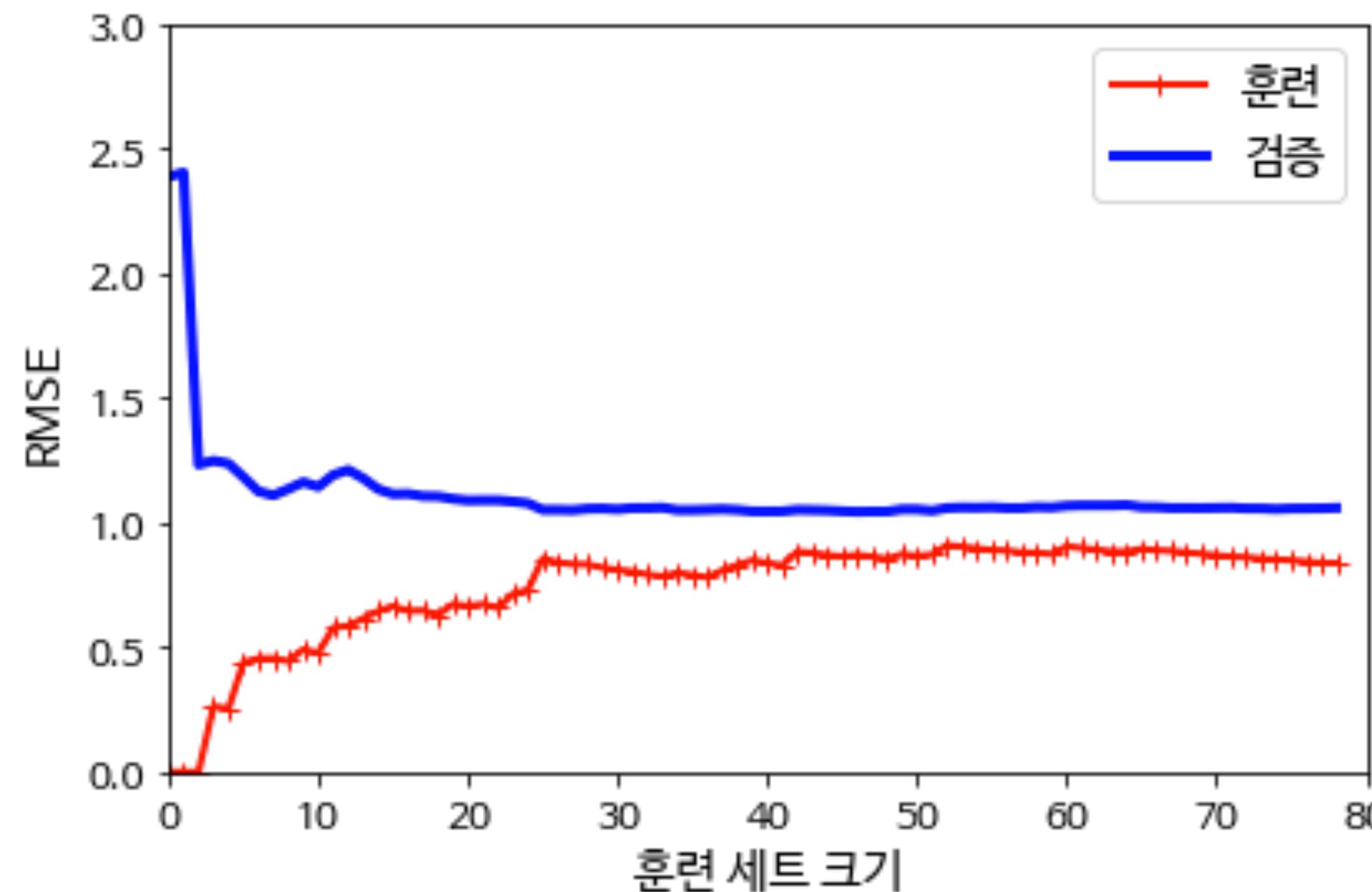
```
from sklearn.pipeline import Pipeline

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

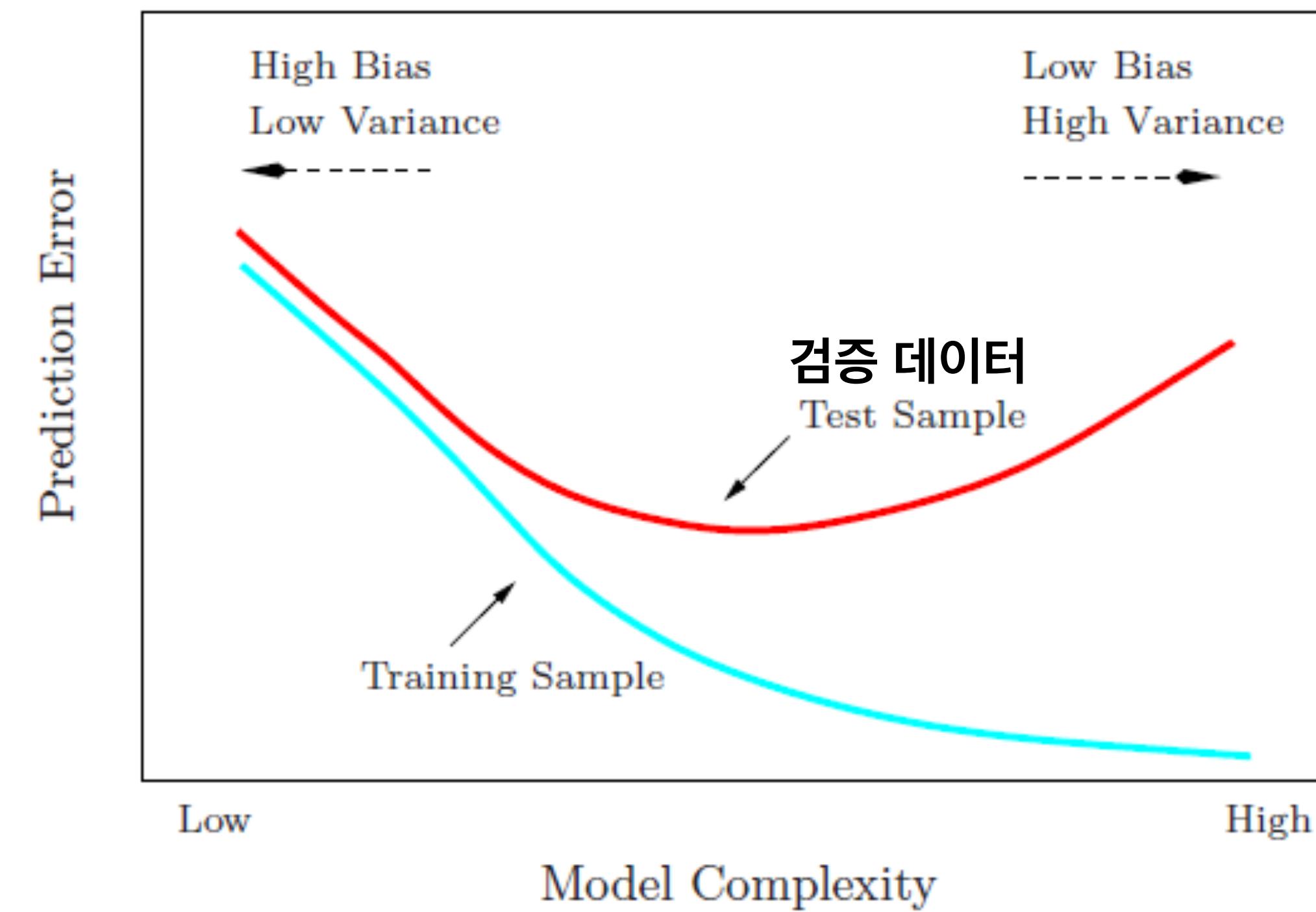
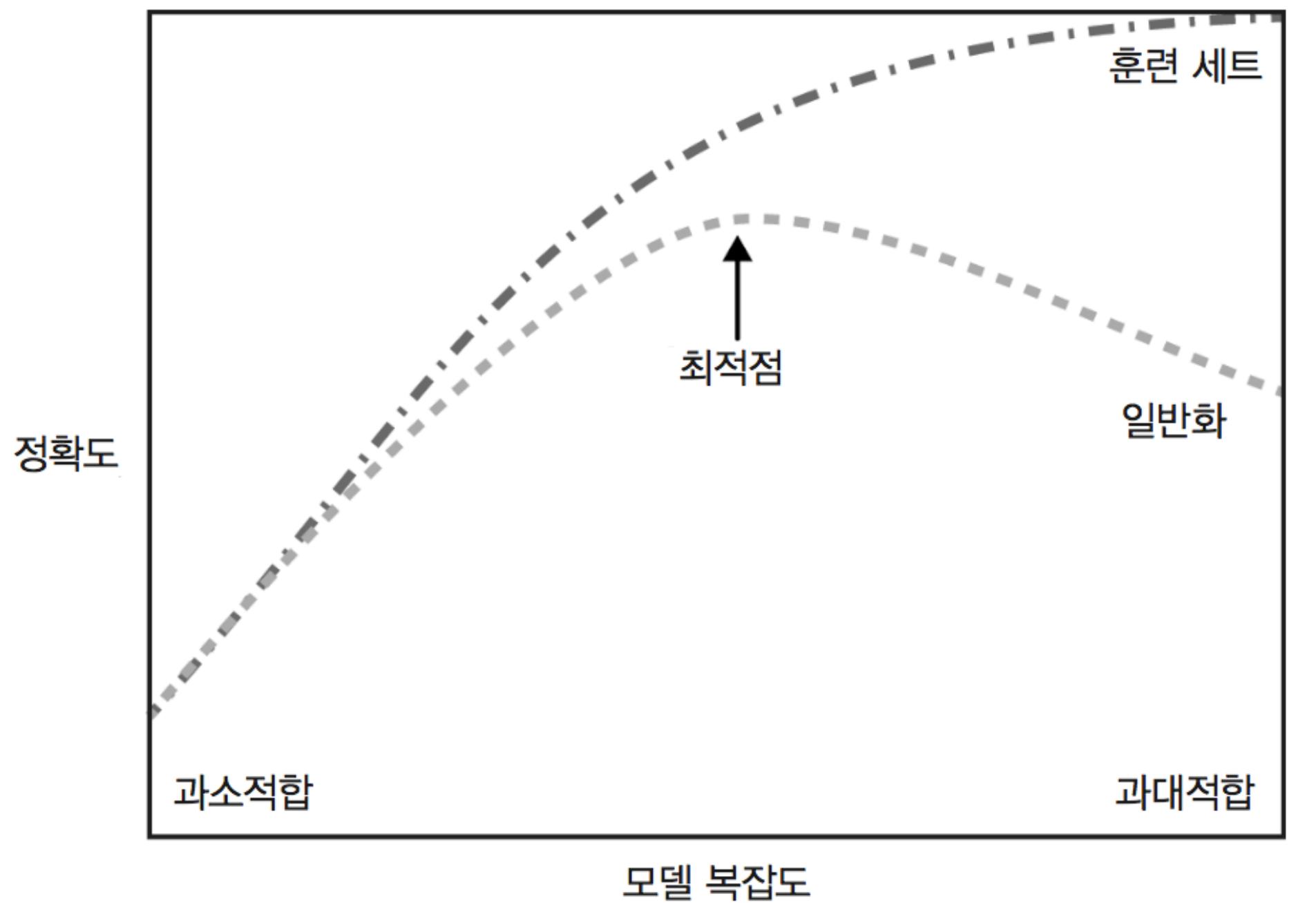


2차 다항 회귀의 학습 곡선

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



모델 복잡도 곡선



편향/분산 트레이드오프

- 편향: 잘못된 가정으로 발생. 과소 적합 모델.
- 분산: 훈련 데이터에 민감한 모델때문에 발생. 과대 적합 모델.
- 복잡도가 커지면 분산이 늘고 편향은 줄어 듭니다. 반대로 복잡도가 줄어들면 분산이 줄고 편향이 커집니다.

} 회피할 수 없는 편향

베이즈 에러(bayes error/irreducible error), 사람 수준의 에러(human-level error)

} 회피 가능 편향(avoidable bias)

훈련 에러

} 분산

검증 에러

규제 모델

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

릿지(Ridge) 회귀

- 티호노프(Tikhonov) 규제라고도 부릅니다.

- 비용 함수에 L2 노름의 제곱을 추가합니다.

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

- alpha가 커지면 가중치가 0에 가까워지고, alpha가 0에 가까우면 MSE만 남습니다.
- 훈련이 끝나고 성능을 평가할 때는 규제를 포함하지 않습니다(성능 평가와 비용 함수가 다른 경우가 많습니다).
- 각 가중치를 같은 비율로 규제하기 때문에 입력 데이터의 스케일에 민감합니다.

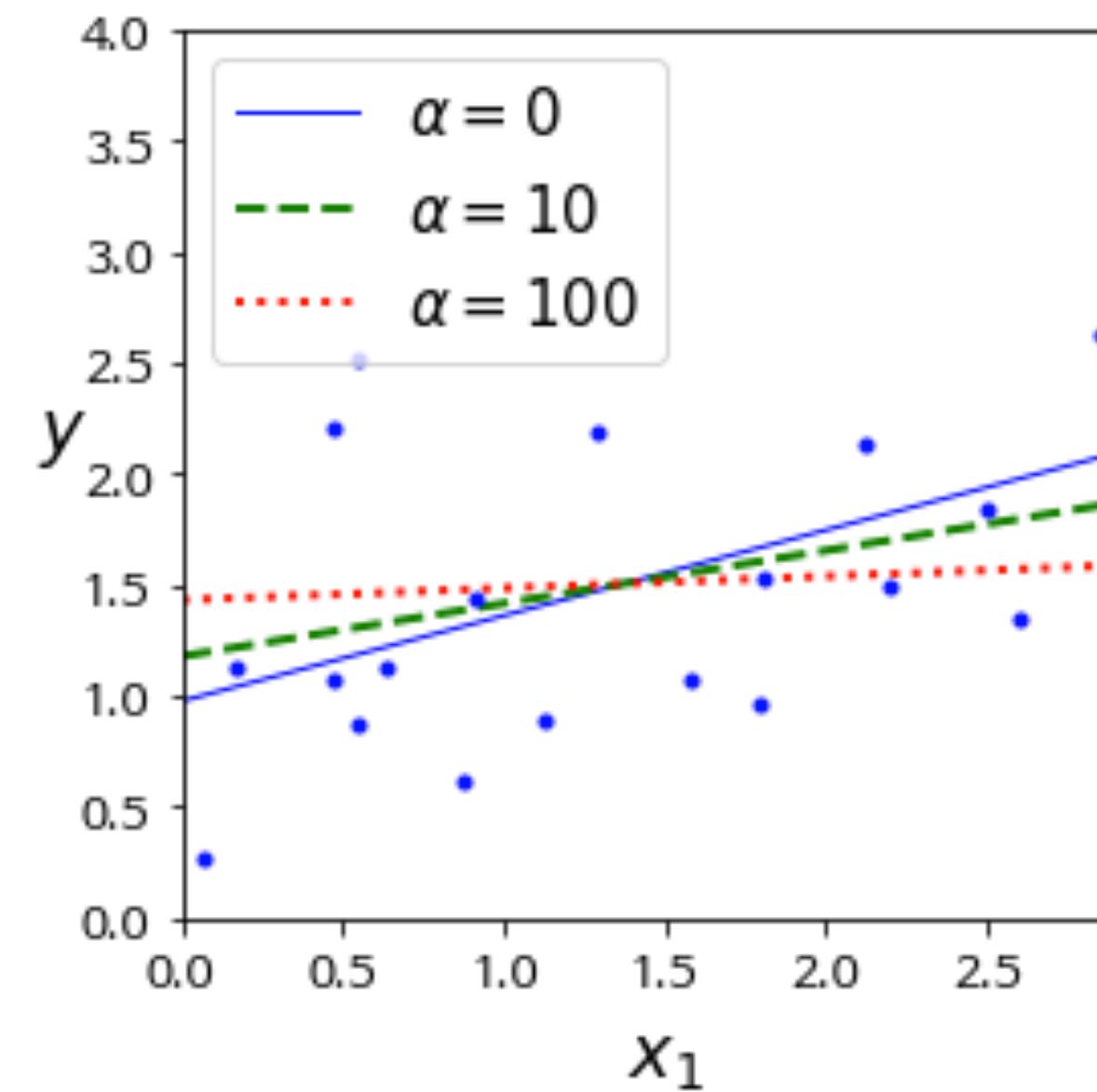
미분 결과를 간단하게 만들기 위해 추가
규제량을 조절: 하이퍼파라미터

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

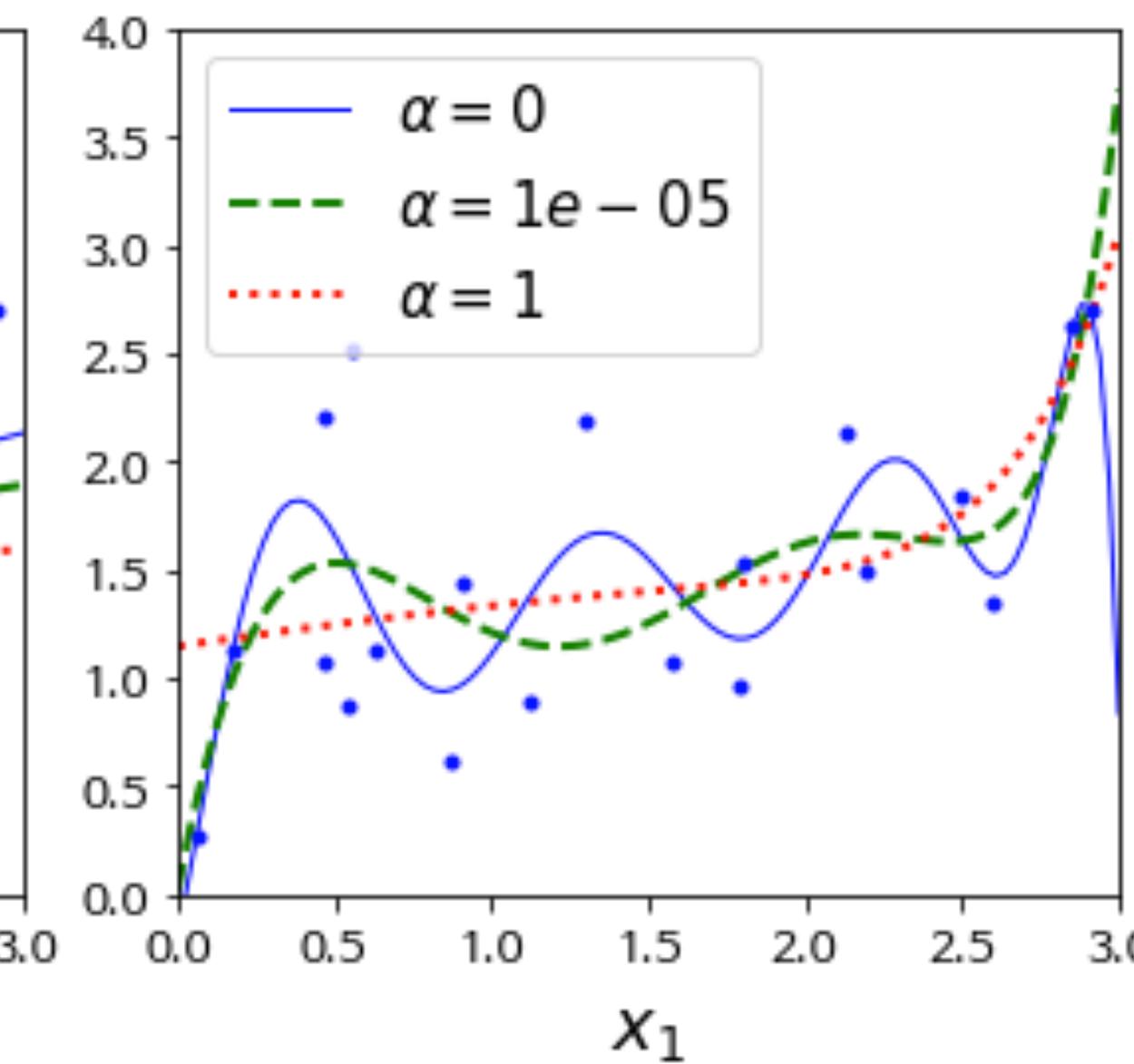
θ_0 는 규제하지 않습니다

릿지 모델의 예

- `sklearn.linear_model.Ridge`
- `alpha`가 커지면 분산이 줄고 편향이 커집니다.



Ridge()



PolynomialFeatures(degree=10) + Ridge()

Ridge 정규 방정식

- 솔레스키 분해(Cholesky decomposition)을 사용하여 해를 구할 수 있습니다.

$$\hat{\theta} = (\mathbf{X}^T \cdot \mathbf{X} + \alpha \mathbf{A})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y} \quad \mathbf{A}: (n+1, n+1) \text{의 단위 행렬, 왼쪽 맨 위 원소는 } 0 \text{입니다.}$$

- 어떤 행렬 M이 대칭이고 양의 정부호 행렬일 때(모든 고윳값이 양수) 하삼각 행렬 L·LT로 분해 가능합니다.

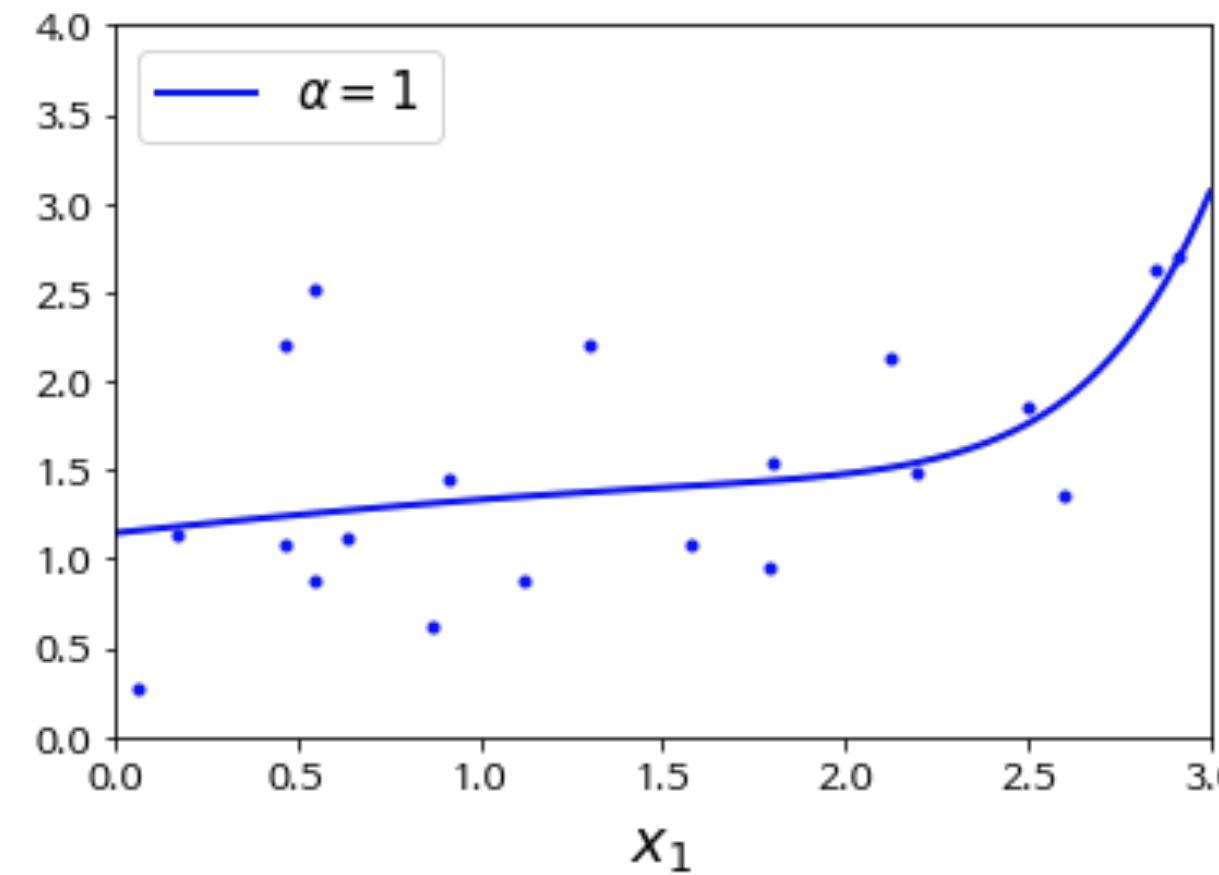
$$(\mathbf{X}^T \cdot \mathbf{X} + \alpha \mathbf{A}) \cdot \hat{\theta} = (L \cdot L^T) \cdot \hat{\theta} = \mathbf{X}^T \cdot \mathbf{y} \quad \begin{aligned} L \cdot a &= X^T \cdot y \\ L^T \cdot \hat{\theta} &= a \end{aligned}$$

- Ridge 클래스는 solver='auto'가 기본값이며 희소, 특이 행렬이 아닐 경우 'cholesky' 방식을 사용합니다.

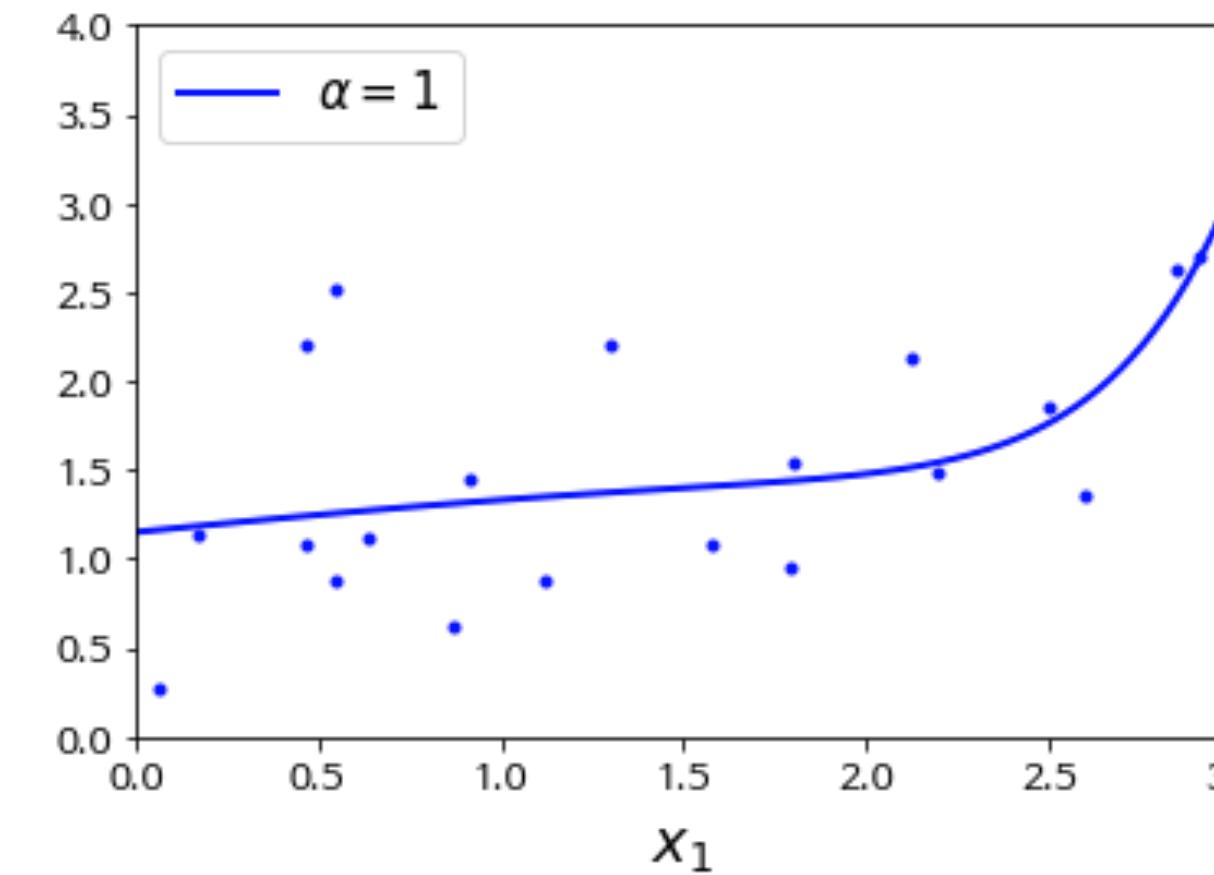
```
from sklearn.linear_model import Ridge
ridge_reg = Ridge(alpha=1, solver="cholesky", random_state=42)
ridge_reg.fit(X, y)
```

cholesky, sag, saga, SGD

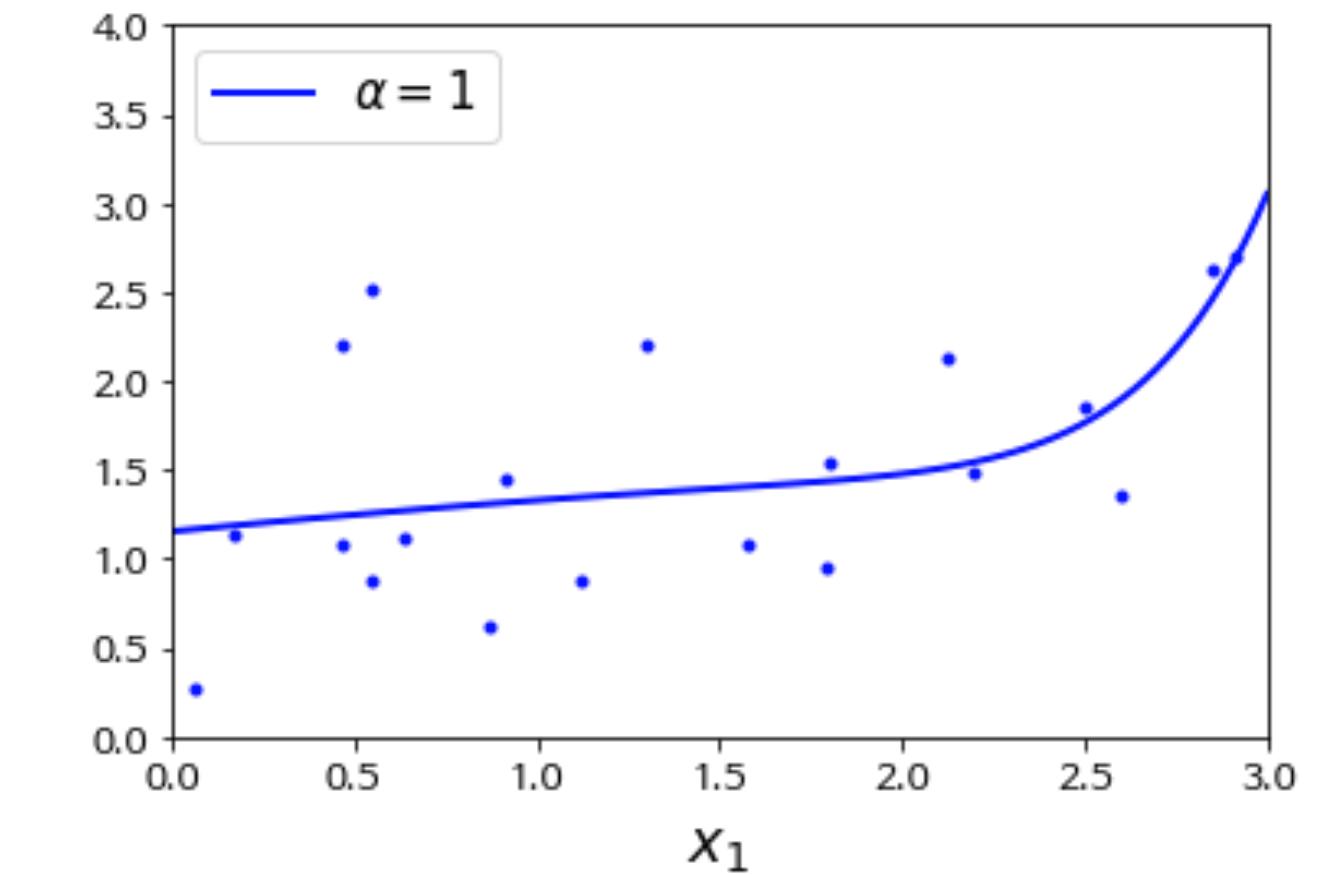
PolynomialFeatures(degree=10)



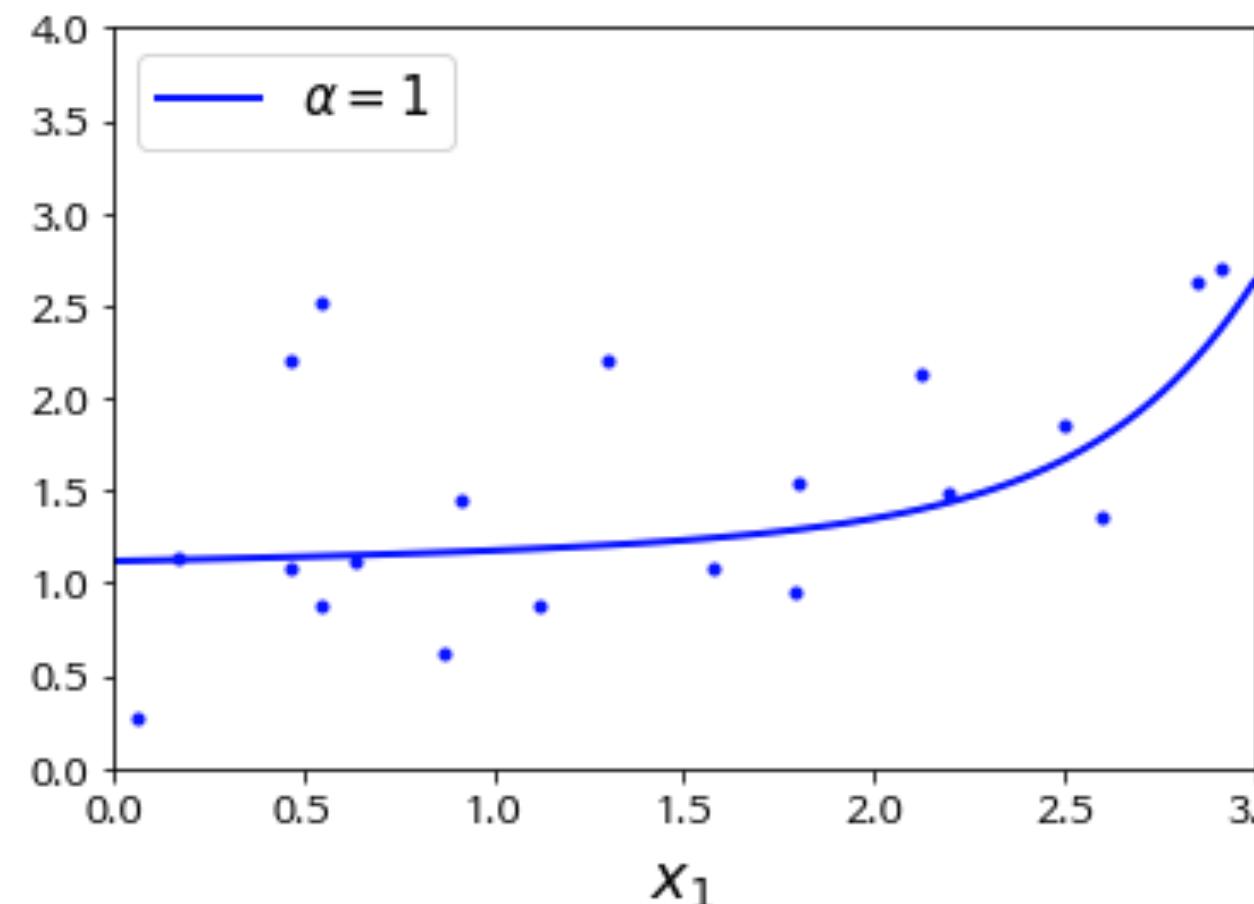
`solver='cholesky'`



`solver='sag'` (확률적 평균 경사 하강법)



`solver='saga'` (0.19버전에 추가)



`SGDRegressor(alpha=1, penalty='l2', max_iter=50)`

라쏘(Lasso) 회귀

- 비용 함수에 L1 노름을 추가합니다.

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

규제량을 조절: 하이퍼파라미터

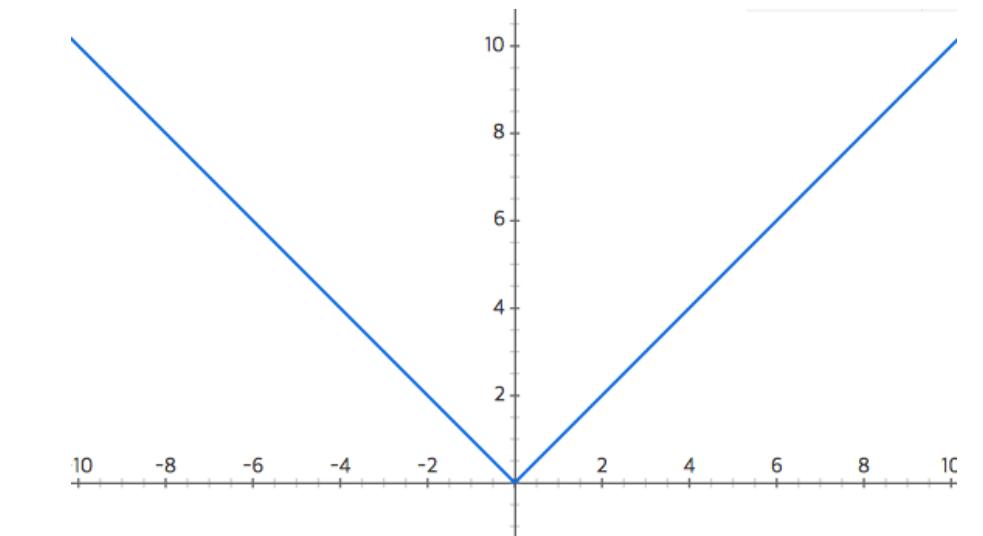
$$J(\theta) = \mathbf{MSE}(\theta) + \alpha \sum_{i=1}^n |\theta_i|$$

θ_0 는 규제하지 않습니다

- 원점에서 미분 불가능하므로 서브그래디언트를 사용합니다.

$$g(\theta, J) = \nabla_{\theta} \mathbf{MSE}(\theta) + \alpha \begin{bmatrix} \mathbf{sign}(\theta_1) \\ \mathbf{sign}(\theta_2) \\ \vdots \\ \mathbf{sign}(\theta_n) \end{bmatrix}$$

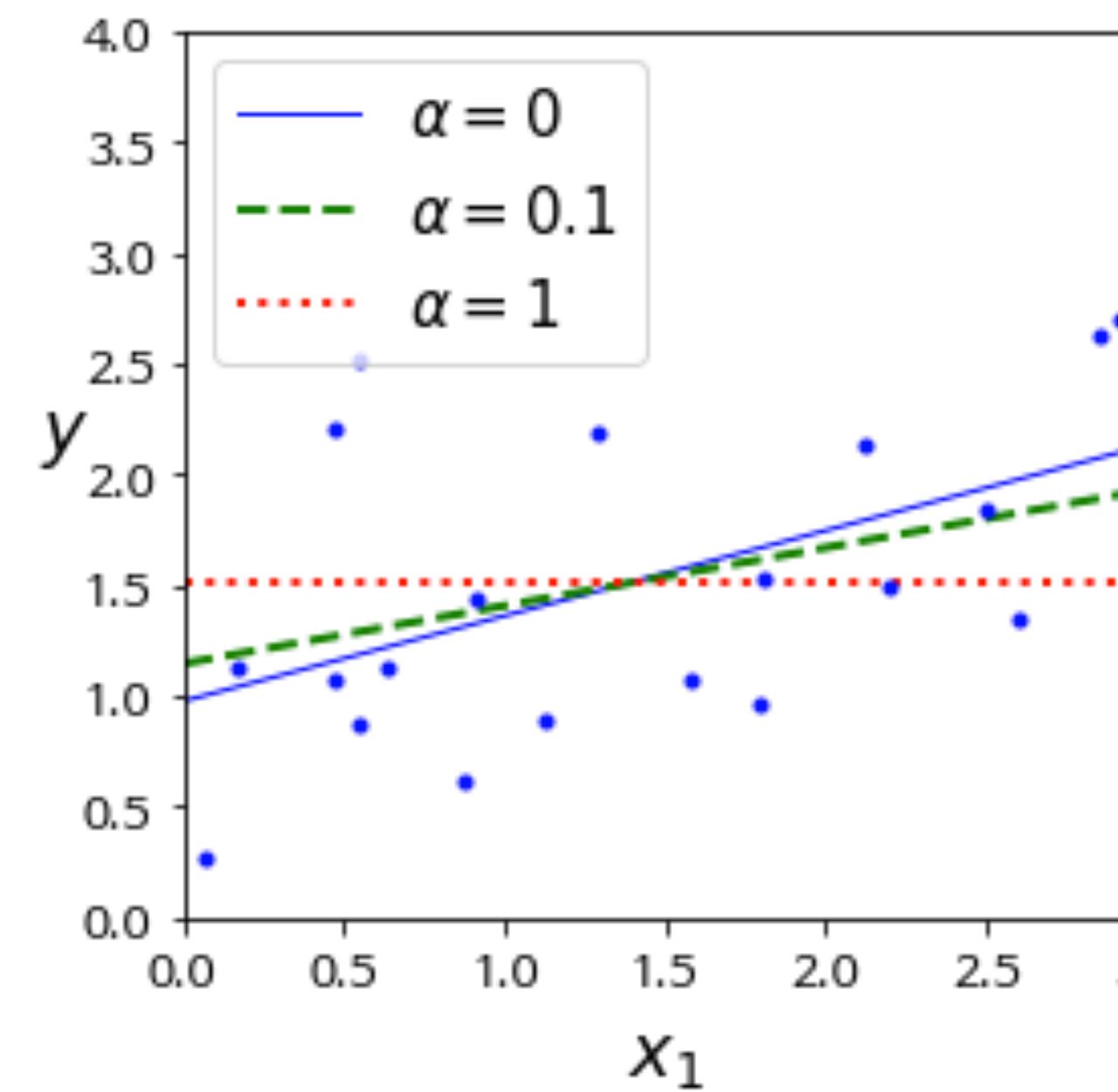
여기서 $\mathbf{sign}(\theta_i) = \begin{cases} -1 & \theta_i < 0 \text{일 때} \\ 0 & \theta_i = 0 \text{일 때} \\ +1 & \theta_i > 0 \text{일 때} \end{cases}$ ← 1을 사용해도 무방합니다.



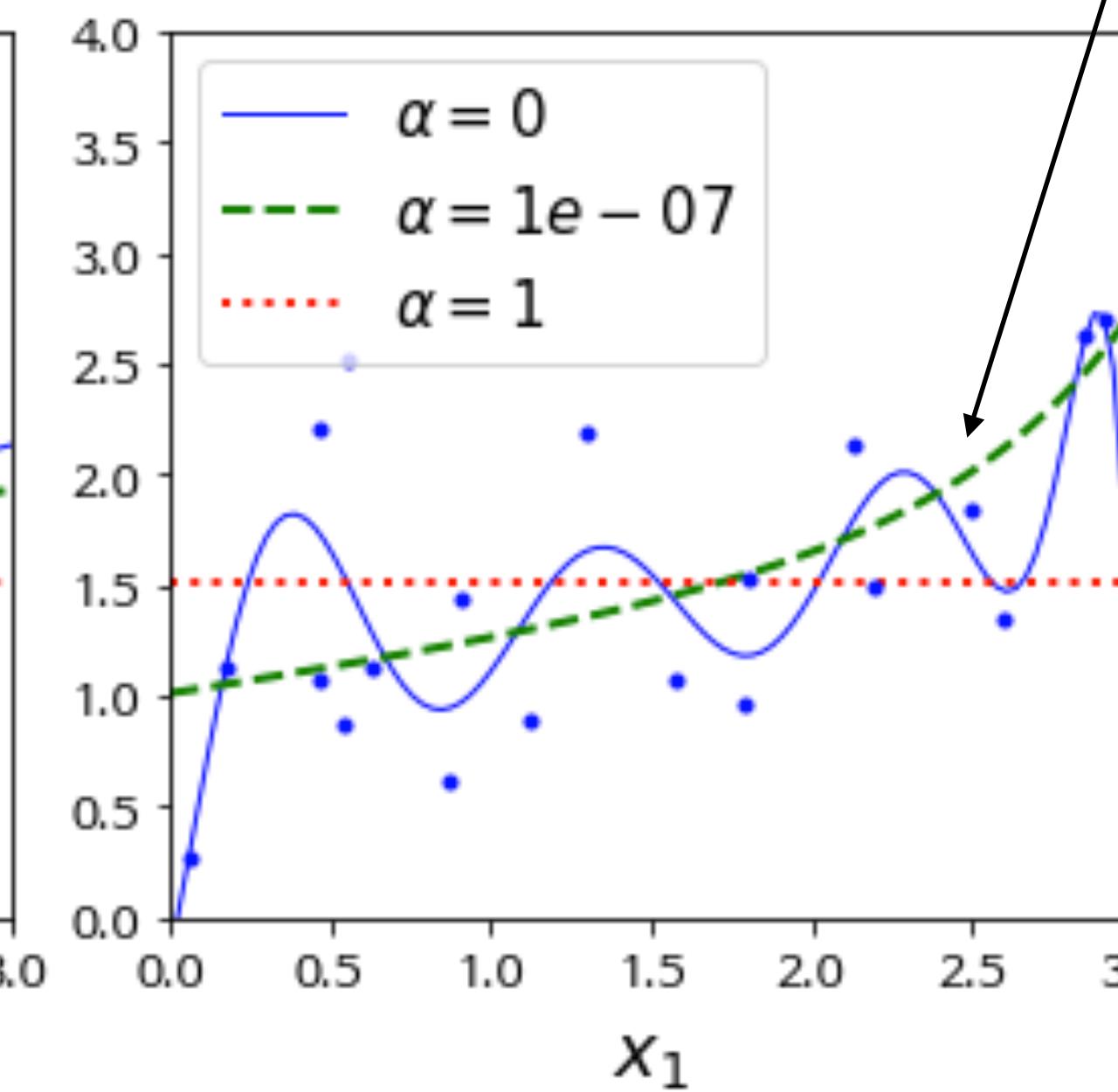
라쏘 모델의 예

- `sklearn.linear_model.Lasso`
- `alpha`가 커지면 분산이 줄고 편향이 커집니다.

규제가 커지면 선형에 가까운 모델을 만듭니다.



Lasso()



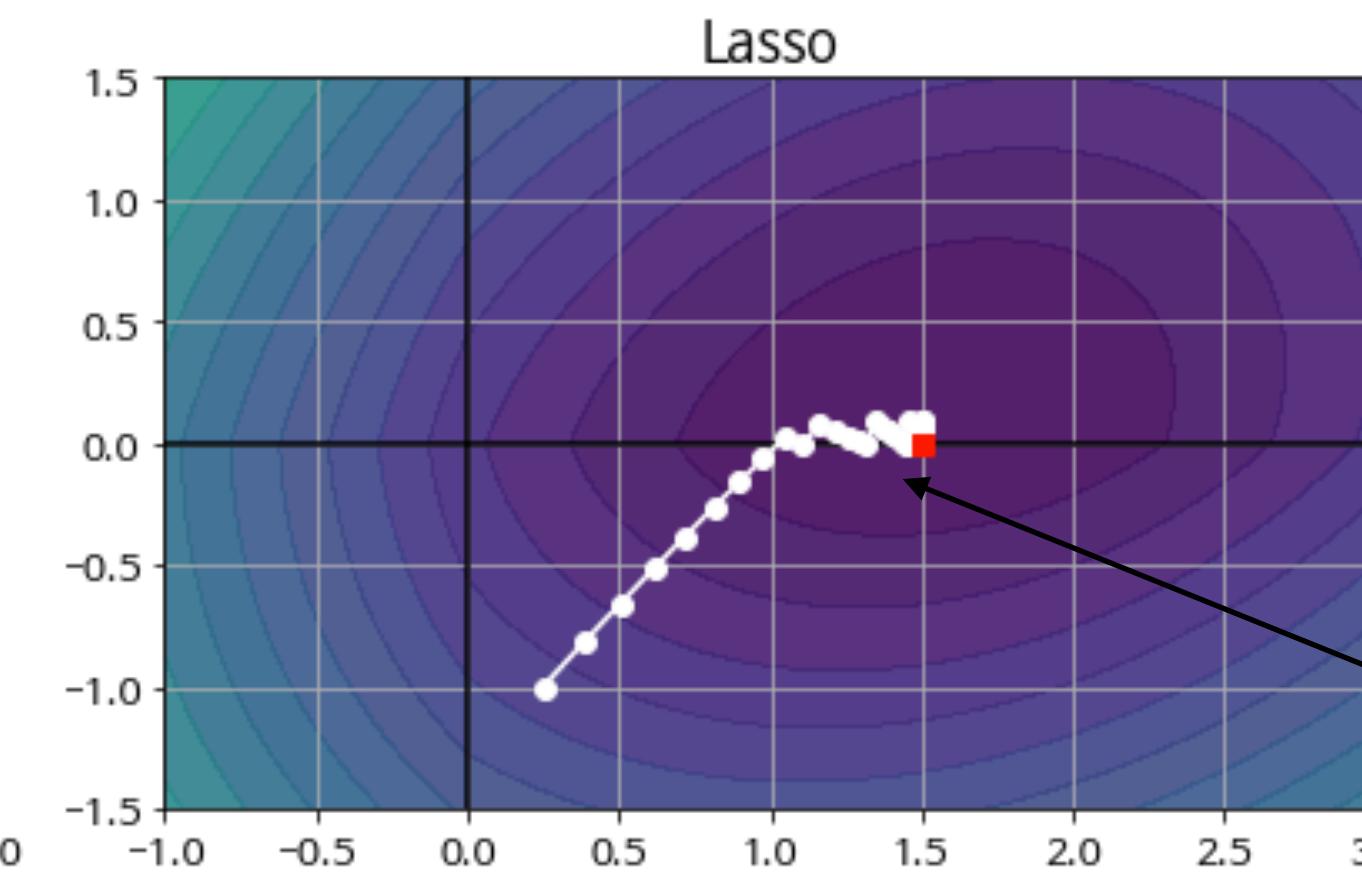
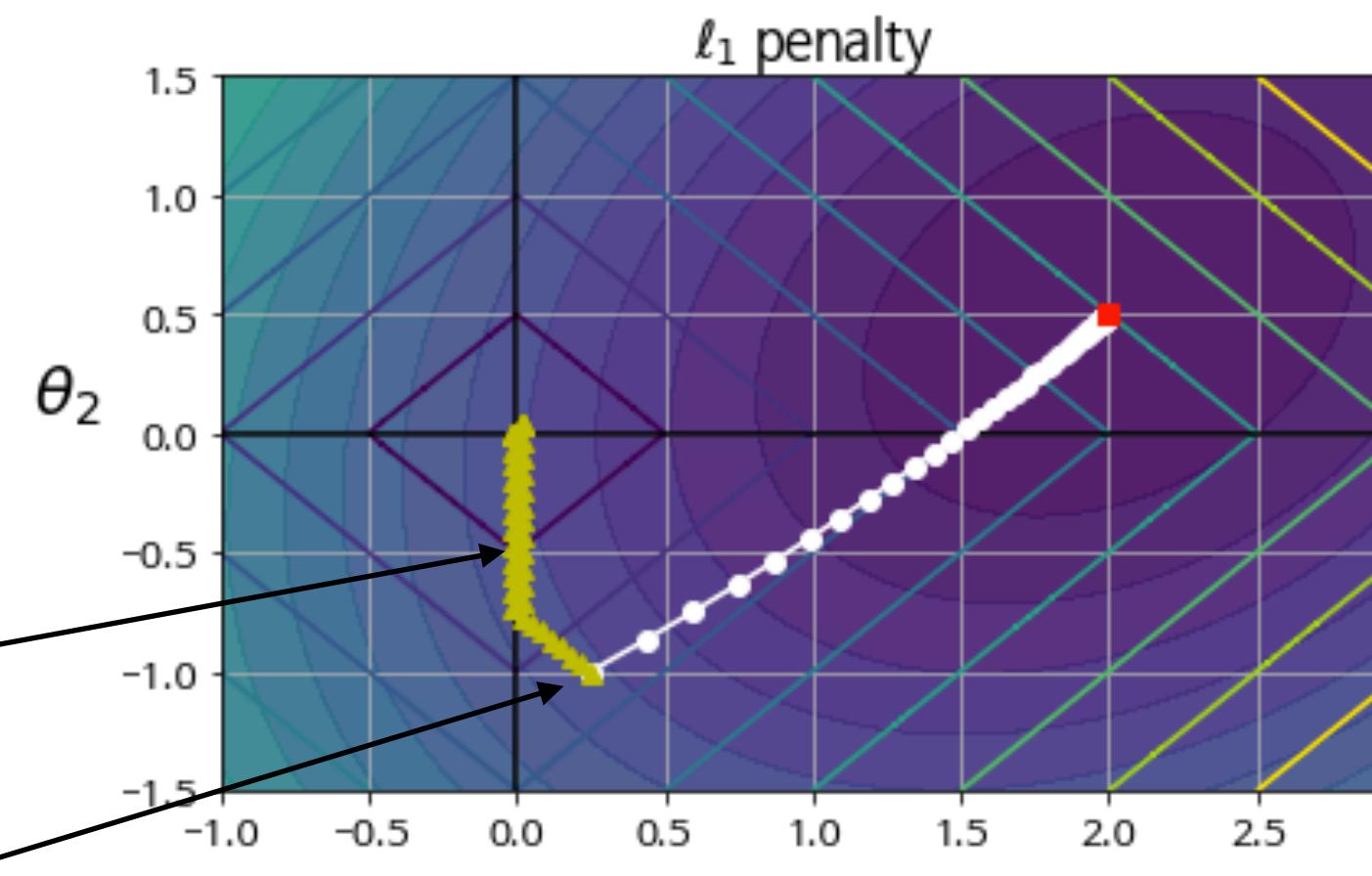
PolynomialFeatures(degree=10) + Lasso()

라쏘의 특징

- 일부 가중치를 0으로 만들어 희소 모델을 만듭니다(특성 선택 효과).

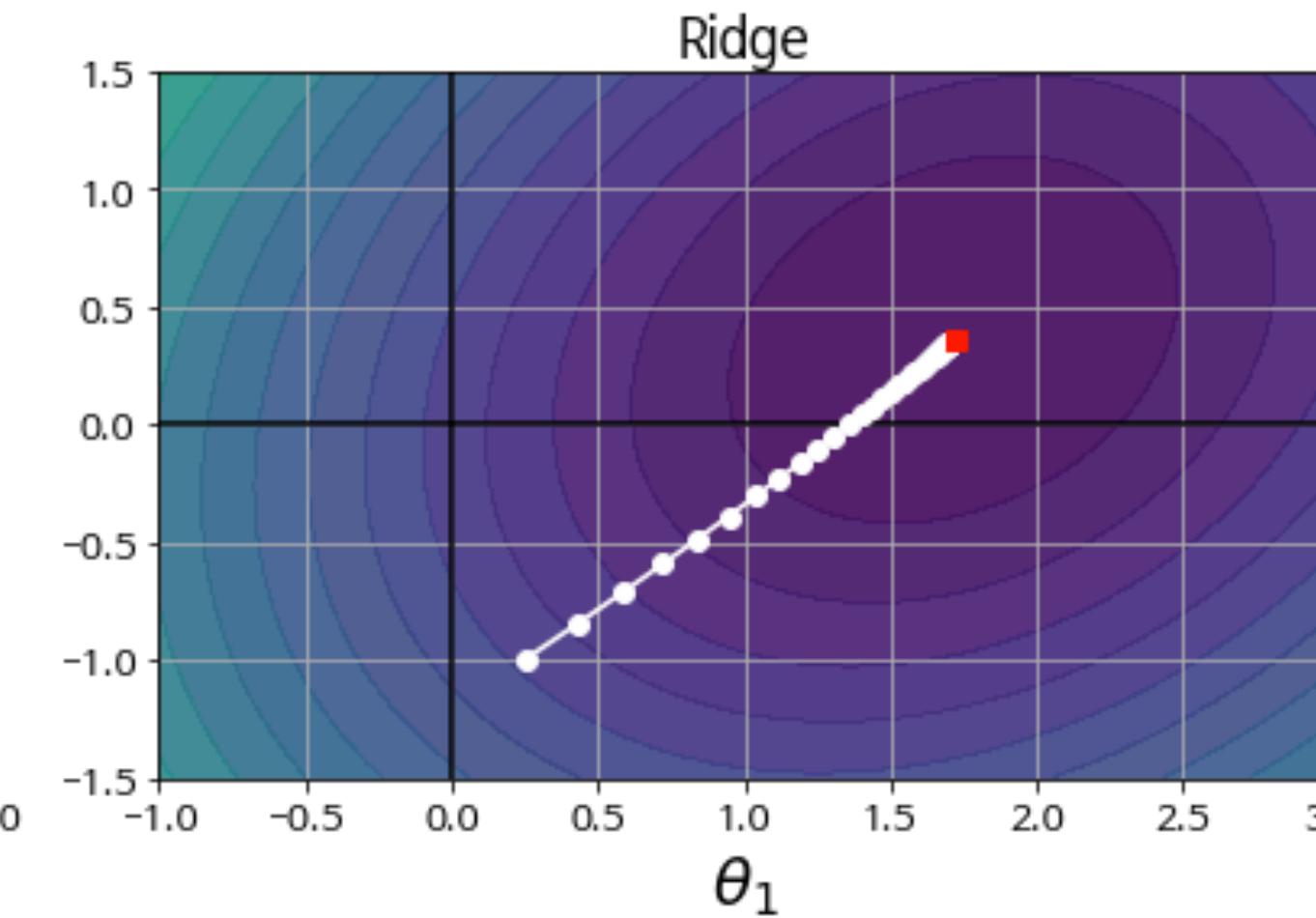
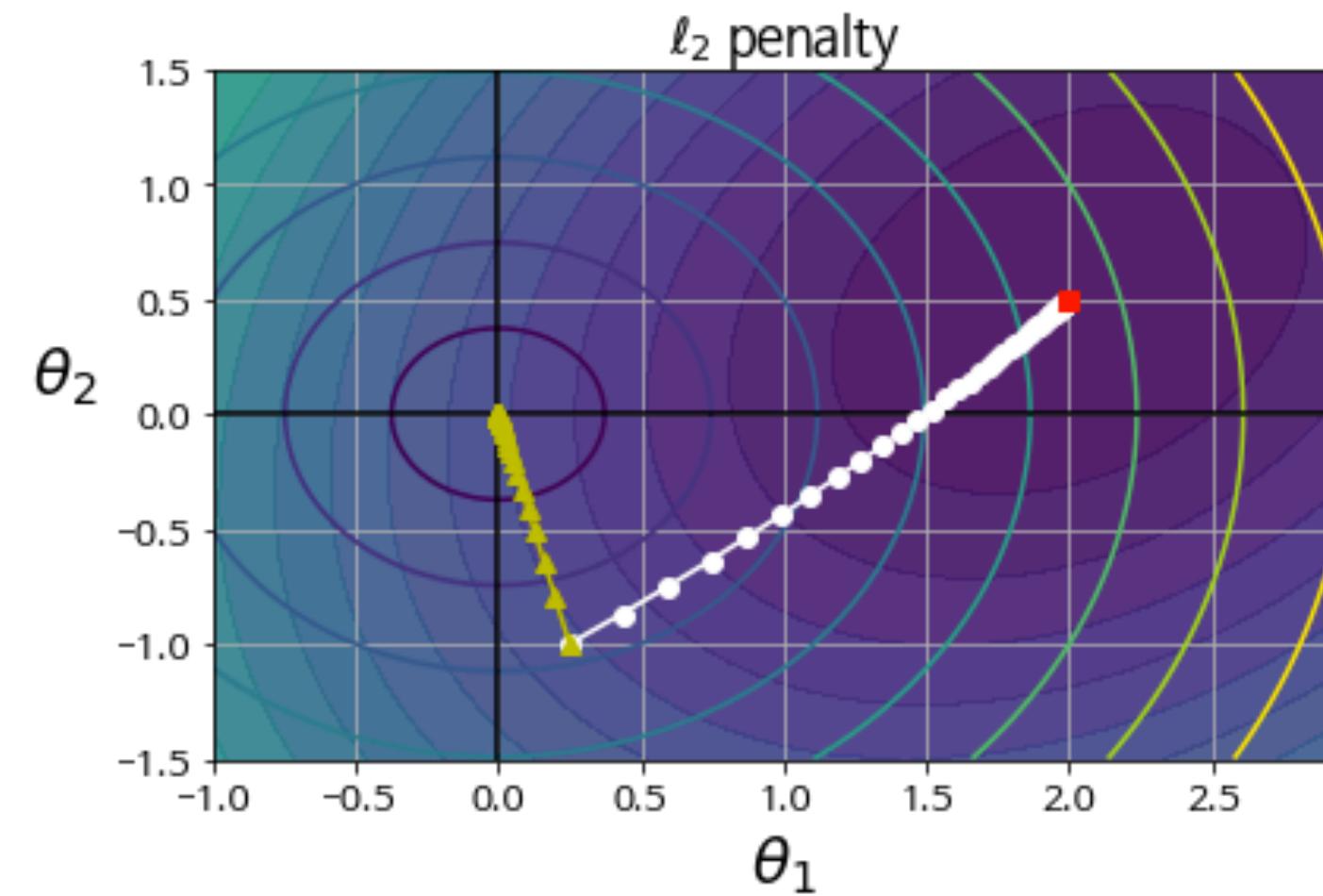
페널티에 대한
그래디언트 경로

(0.25, -1)에서 시작



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

서브그래디언트의 효과



엘라스틱넷(ElasticNet)

- 릿지와 회귀의 규제를 절충한 모델입니다.

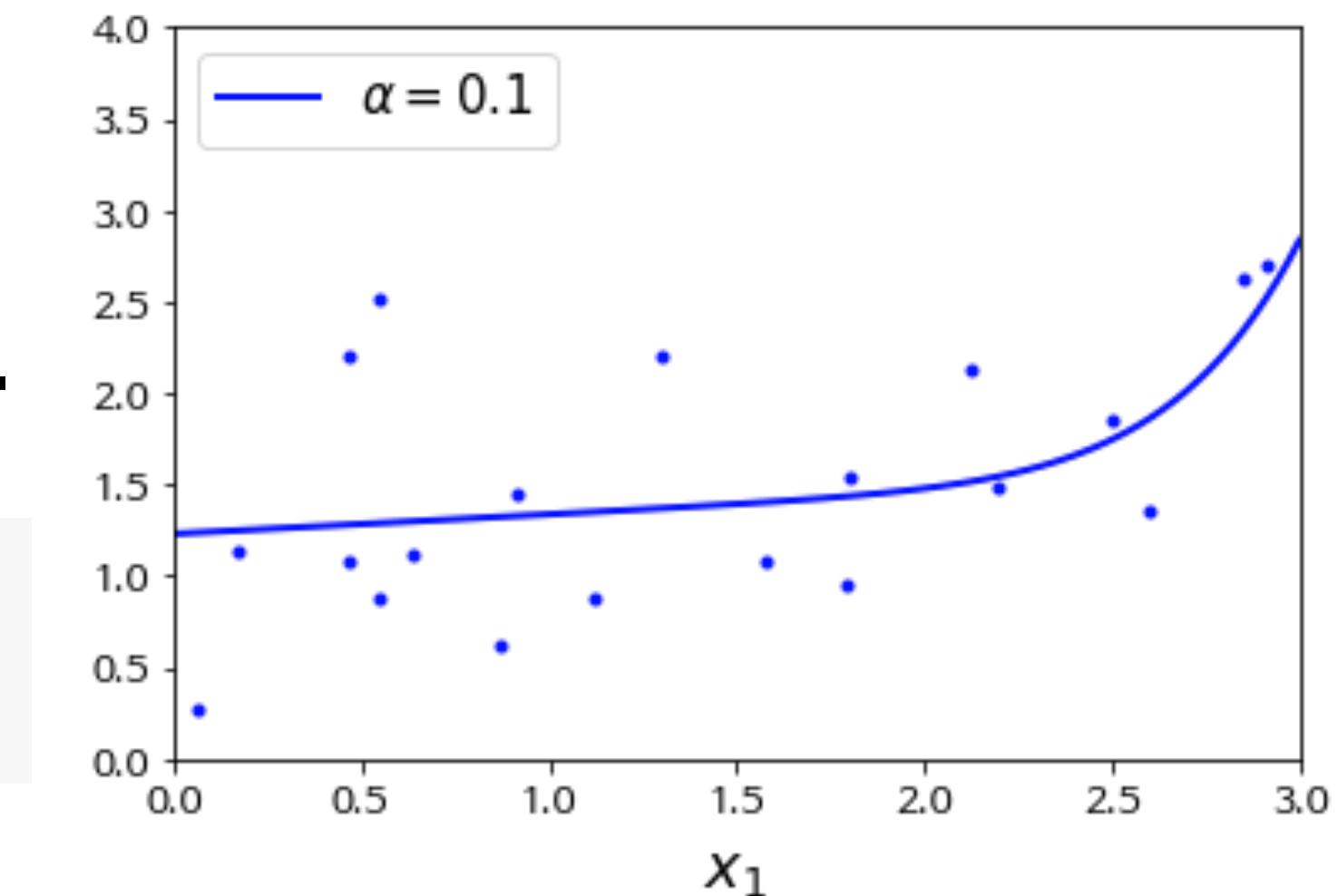
$$J(\theta) = \text{MSE}(\theta) + r\alpha \sum_{i=1}^n |\theta_i| + \frac{1-r}{2}\alpha \sum_{i=1}^n \theta_i^2$$

- $r=0$ 이면 릿지와 같고, $r=1$ 이면 라쏘와 같습니다.
- 보편적으로 라쏘 보다는 릿지나 엘라스틱넷을 선호합니다.

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

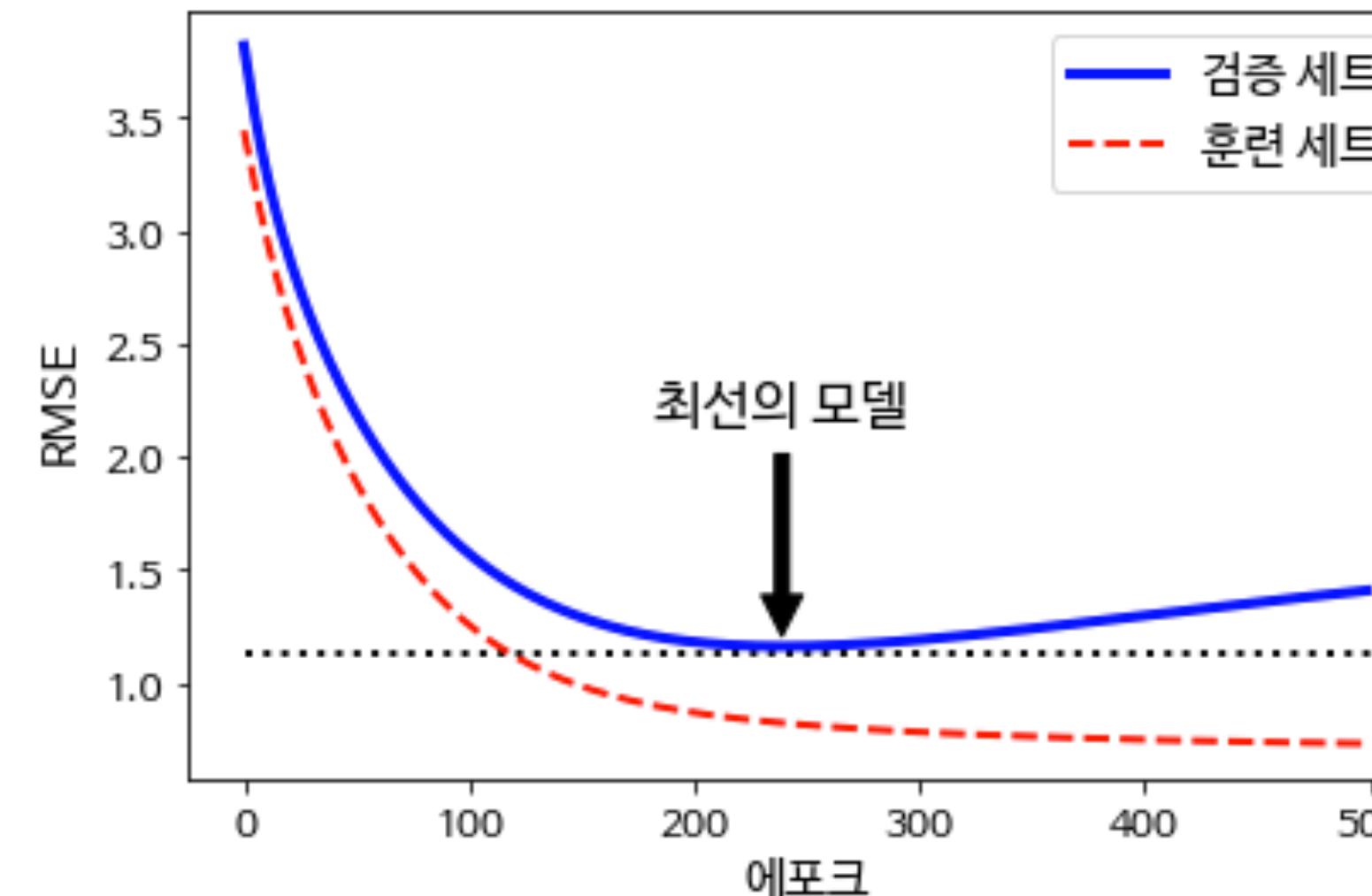
* Lasso 클래스는 ElasticNet(l1_ratio=1)입니다.

r



조기 종료(Early stopping)

- 반복적인 학습 알고리즘의 경우 검증 에러가 최솟값에 도달했을 때 훈련을 중지합니다
- 제프리 힌튼(Geoffrey Hinton)은 ‘공짜 점심’으로 불렀다고 합니다.
- 앤드류 응(Andrew Ng)은 딥러닝 모델을 훈련할 때 충분히 과대적합된 모델을 만들고 난 후 다른 규제 방법을 사용하라고 권장합니다.



조기 종료 구현

1에포크

```
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) ← 최선의 모델을 저장합니다.
```

로지스틱(logistic) 회귀

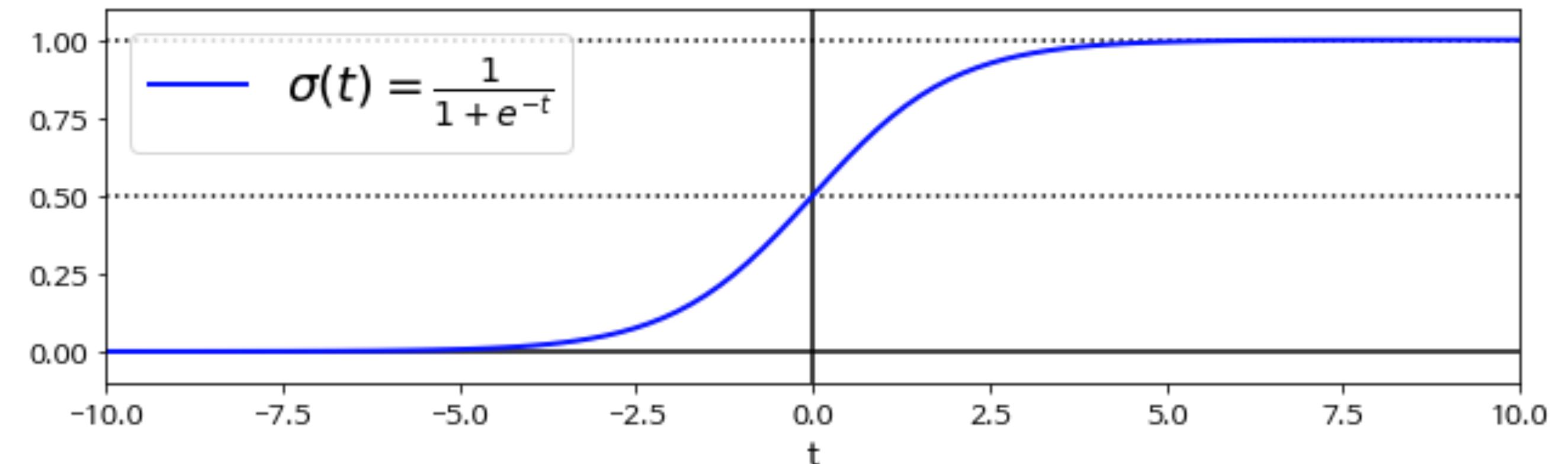
- 로짓(logit) 회귀이라고도 부르며 이진 분류와 다항 분류를 지원합니다.
- 이진 분류는 양성(1) 클래스와 음성(0) 클래스를 분류합니다.
- 선형 방정식을 시그모이드(sigmoid) 함수에 통과시켜 0~1 사이의 확률을 얻습니다.

$$\hat{p} = h_{\theta}(\mathbf{x}) = \sigma(\theta^T \cdot \mathbf{x}) , \quad \sigma(t) = \frac{1}{1 + e^{-t}}$$

- 예측

$$\hat{y} = \begin{cases} 0 & \hat{p} < 0.5 \text{ or } \theta^T \cdot \mathbf{x} < 0 \text{일 때} \\ 1 & \hat{p} \geq 0.5 \text{ or } \theta^T \cdot \mathbf{x} \geq 0 \text{일 때} \end{cases}$$

↑ ↑
predict_proba() predict()



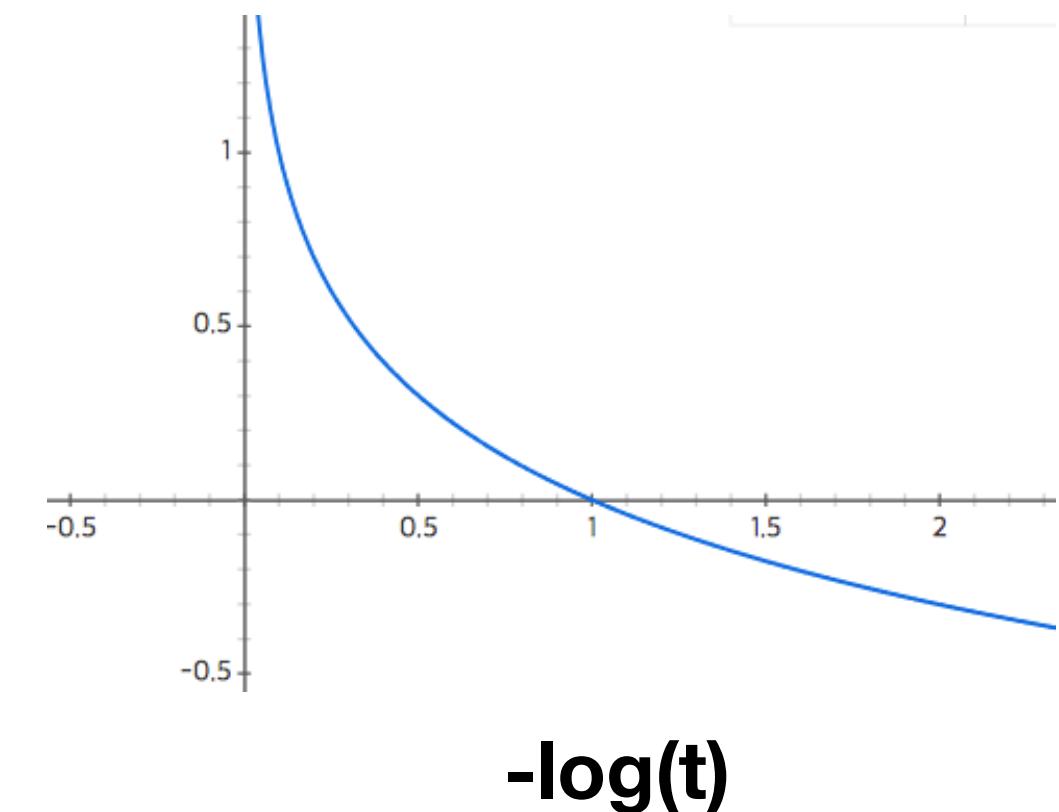
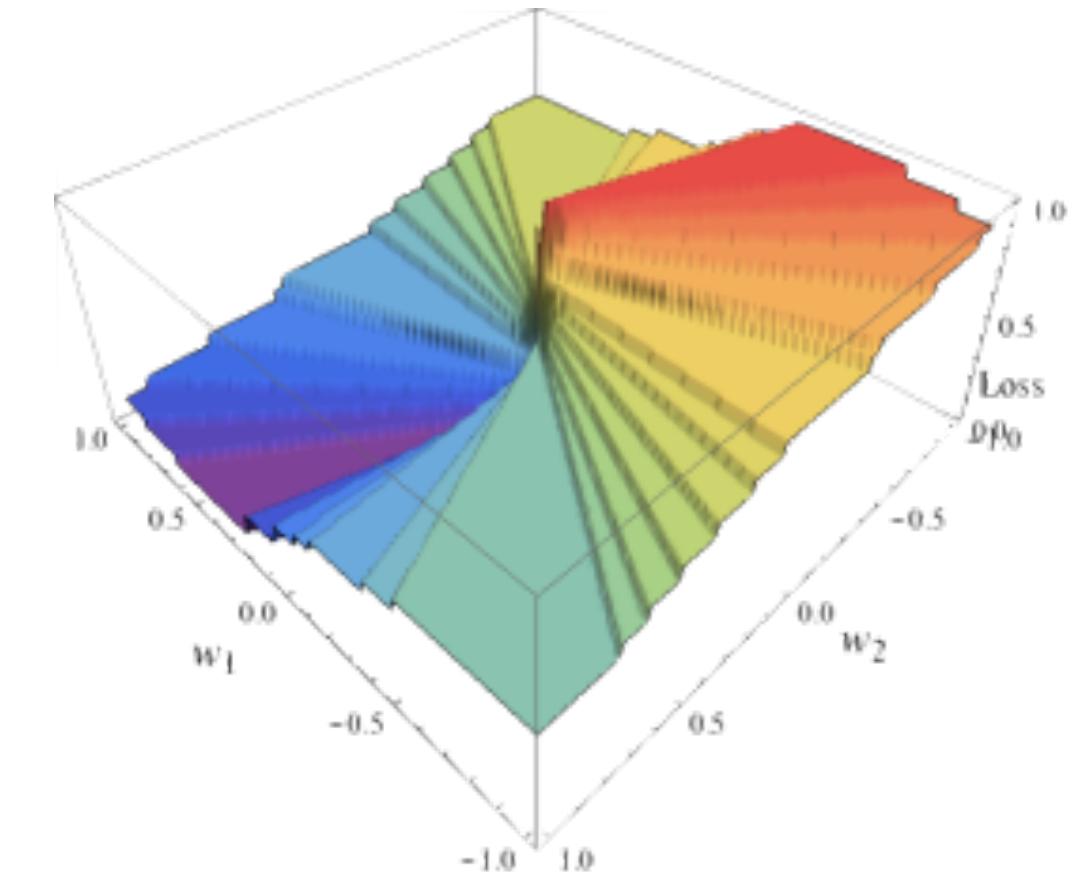
로그 손실(log loss)

- 분류의 정확도 등은 계단 함수이며 미분 불가능합니다.
- 대신 로그 손실 함수를 사용합니다.

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

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

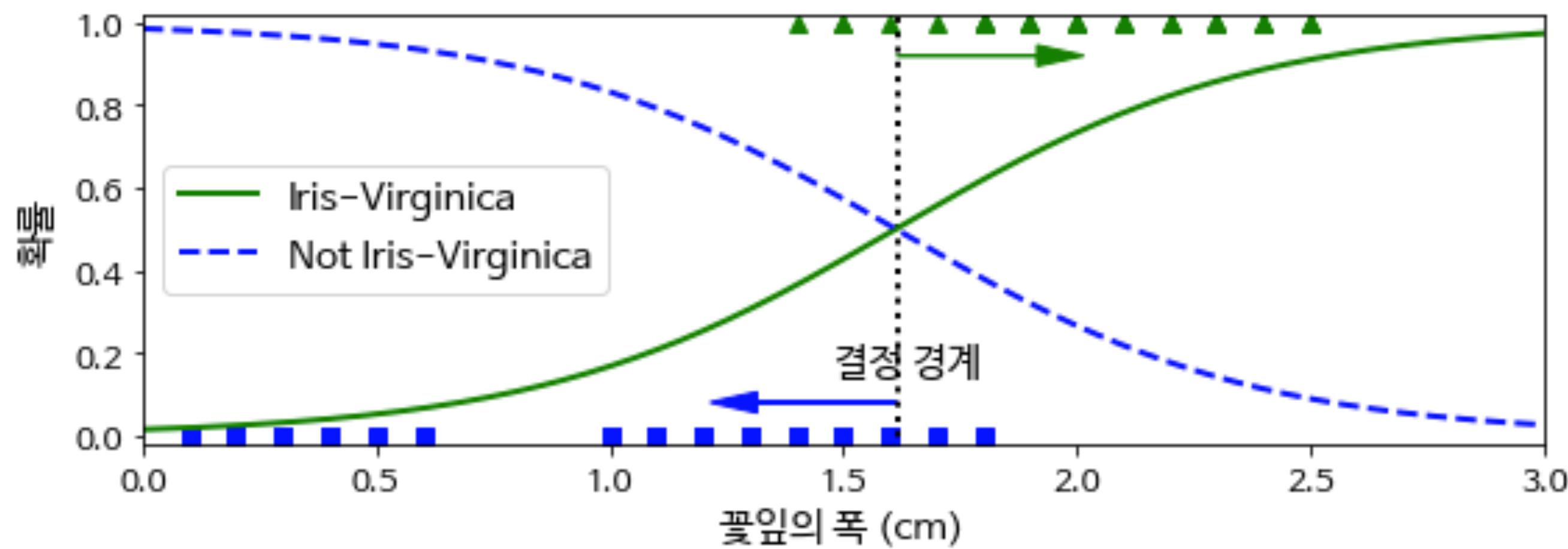
- scikit-learn에서는 $J(\theta) = \frac{1}{m} \sum_{i=1}^m \log(e^{-y^{(i)}(\theta^T \cdot \mathbf{x}^{(i)})} + 1)$, $y = \{-1, 1\}$



붓꽃 예제(특성 1개)

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

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



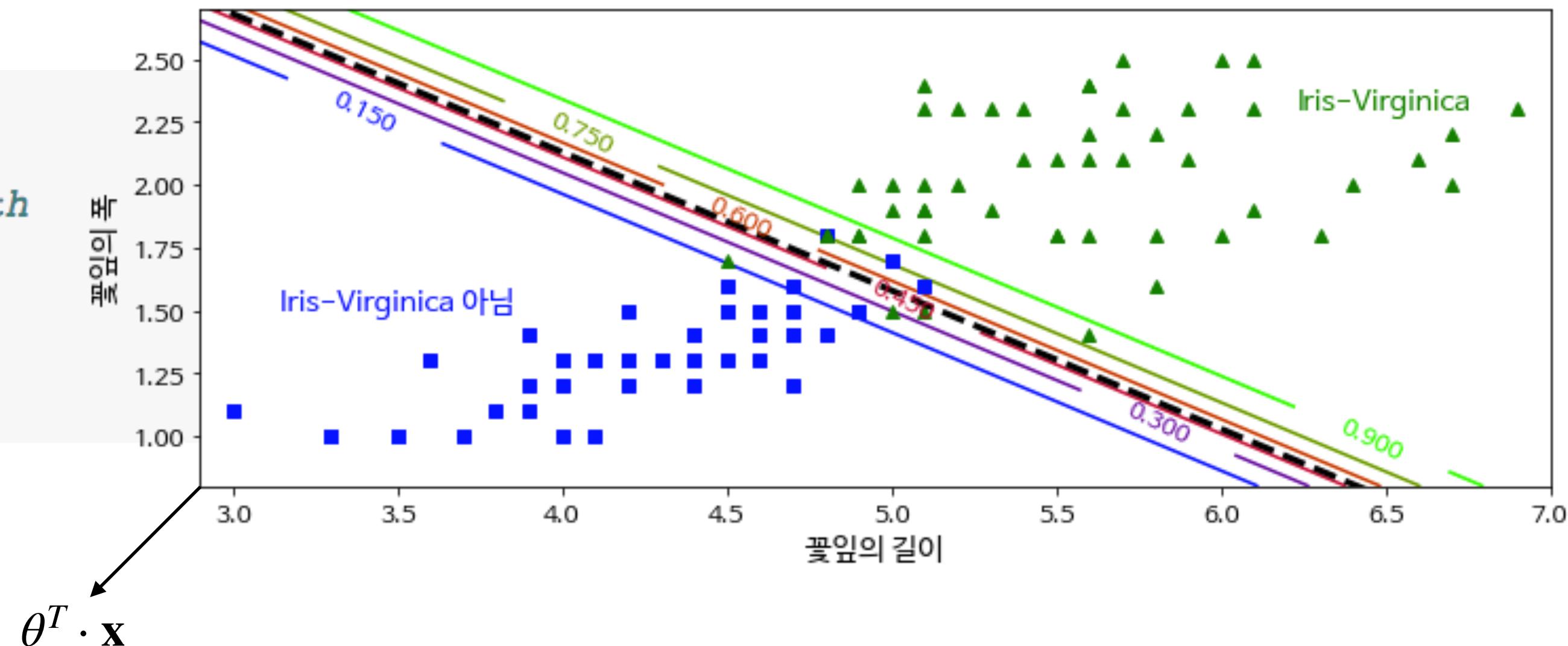
붓꽃 예제(특성 2개)

- LogisticRegression의 규제는 C입니다. C가 감소할수록 규제가 커집니다.

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

```
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



소프트맥스 회귀

- 로지스틱 회귀를 사용한 다중 분류
- 각 클래스의 결정 함수 값을 지수 함수로 정규화합니다.

$$s_k(\mathbf{x}) = (\boldsymbol{\theta}^{(i)})^T \cdot \mathbf{x}$$

$$\hat{p}_k = \sigma(s(\mathbf{x}))_k = \frac{e^{s_k(\mathbf{x})}}{\sum_{j=1}^K e^{s_j(\mathbf{x})}} = \frac{e^{s_k(\mathbf{x})}}{e^{s_1(\mathbf{x})} + e^{s_2(\mathbf{x})} + \dots + e^{s_k(\mathbf{x})} + \dots + e^{s_K(\mathbf{x})}}$$

$$\hat{y} = \operatorname{argmax}_k \sigma(s(\mathbf{x}))_k = \operatorname{argmax}_k s_k(\mathbf{x})$$

- 손실 함수(크로스 엔트로피 함수)

$$D_{KL} = H(p, q) - H(p) = - \sum p(x) \log q(x) + \sum p(x) \log p(x)$$

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

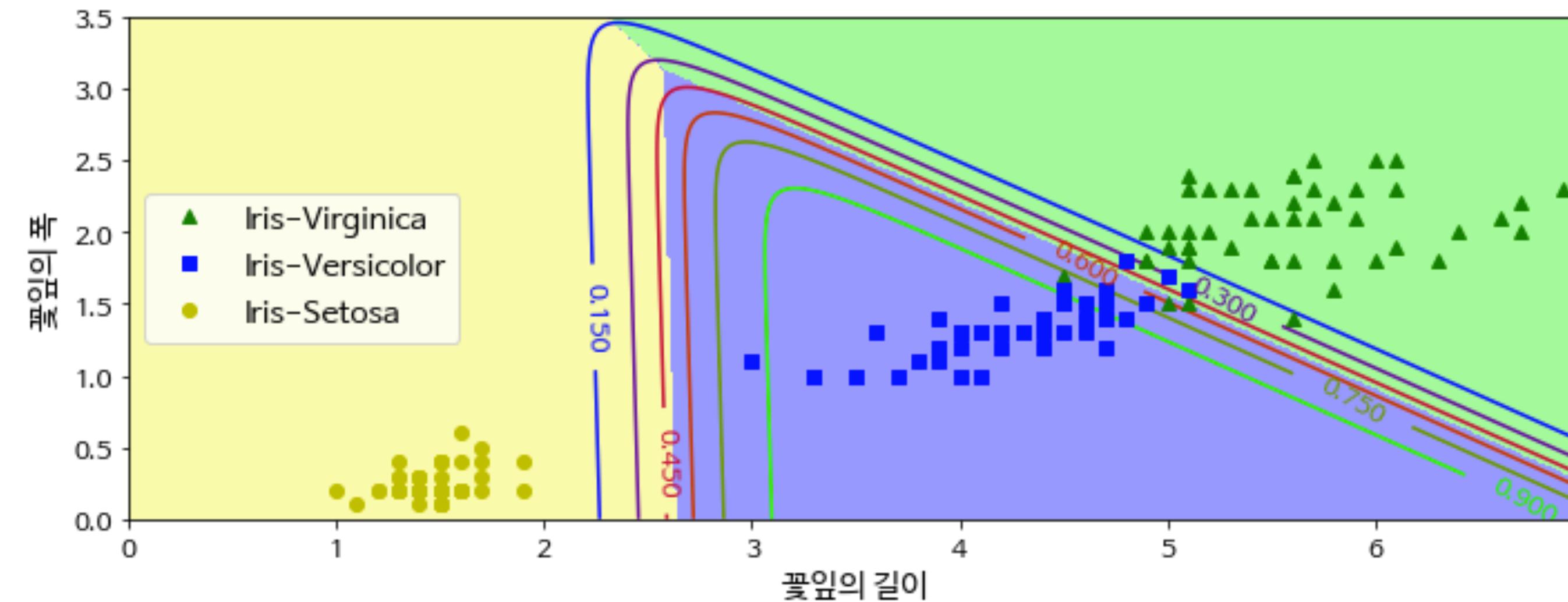
$$\frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m (\hat{p}^{(i)} - y^{(i)}) x_j^{(i)}$$

* K=2 이면 로그 손실과 같습니다.

소프트맥스 회귀의 예

```
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)
```

기본값은 'ovr' newton-cg, sag가 'multinomial' 지원



* 0.19버전에서 추가된 `solver='saga'` 가 다중 분류를 지원하고 l1, l2 규제를 모두 지원합니다.

감사합니다