Chapter 4 모델 훈련

탐색적 자료분석과 기계학습

202140191 엄태훈

index

4.1 선형 회귀

선형 회귀 모델의 예측

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

- $\hat{y}$ = 예측값
- n = 특성의 수
- x_i = i번째 특성값
- θ_j= j번째 모델 파라미터

선형 회귀 모델의 예측(벡터 형태)

$$\hat{y} = h_\theta(X) = \theta \cdot X$$

- θ = 절편θ_0와 θ_1에서θ_n까지의 특성 가중치를 담은 모델 파라미터 벡터
- X = x_0에서 x_n까지 담은 샘플의 특성벡터. x_0는 항상 1이다.
- $\theta \cdot X$ = 벡터 θ와 x의 점곱. $\theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$와 같다.
- h_θ= 모델 파라미터 θ를 사용한 가설함수.

4.1 선형 회귀

- 모델의 훈련 = 모델이 훈련 세트에 가장 잘 맞도록 **모델 파라미터**를 설정한다.

- 회귀에서 가장 널리 사용되는 **성능 측정 지표**
  - ▶ RMSE(평균 제곱근 오차)
  - ▶ MSE(평균 제곱 오차)

- 선형 회귀 모델을 훈련시키려면 RMSE 혹은 MSE를 최소화하는 $\boldsymbol{\theta}$를 찾아야 한다.

- 선형 회귀 모델의 MSE 비용함수는 다음과 같다.

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

4.1 선형 회귀

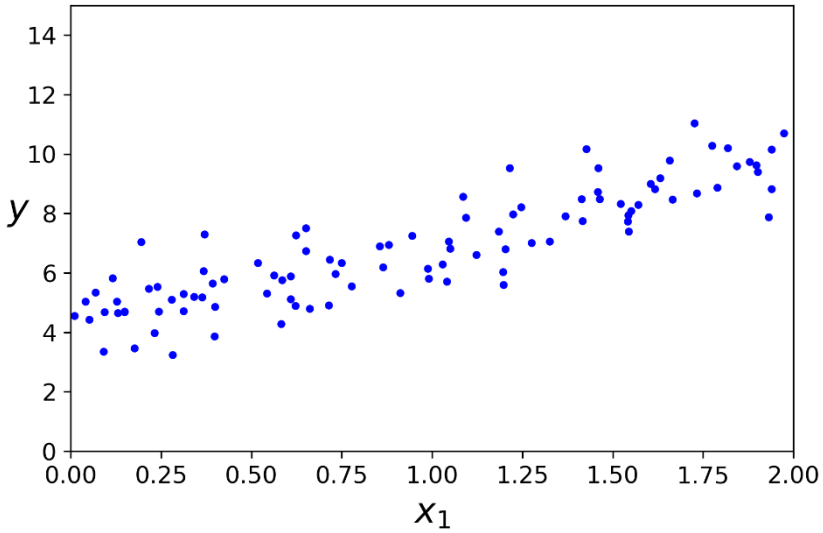
정규방정식: 비용 함수를 최소화 하는 $\boldsymbol{\theta}$값을 찾기 위한 해석적인 방법.

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

· $\hat{\theta}$ = 비용 함수를 최소화 하는 θ값.

· $y = y^{(1)}\ y^{(m)}$까지 포함하는 타깃 벡터

```
In [2]: import numpy as np

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



4.1 선형 회귀

```
In [4]: 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)
```

```
In [5]: theta_best

Out[5]: array([[4.21509616],
               [2.77011339]])
```

・데이터를 생성하기 위해 사용된 함수는 $y = 4 + 3x_1 +$ 가우시안 잡음

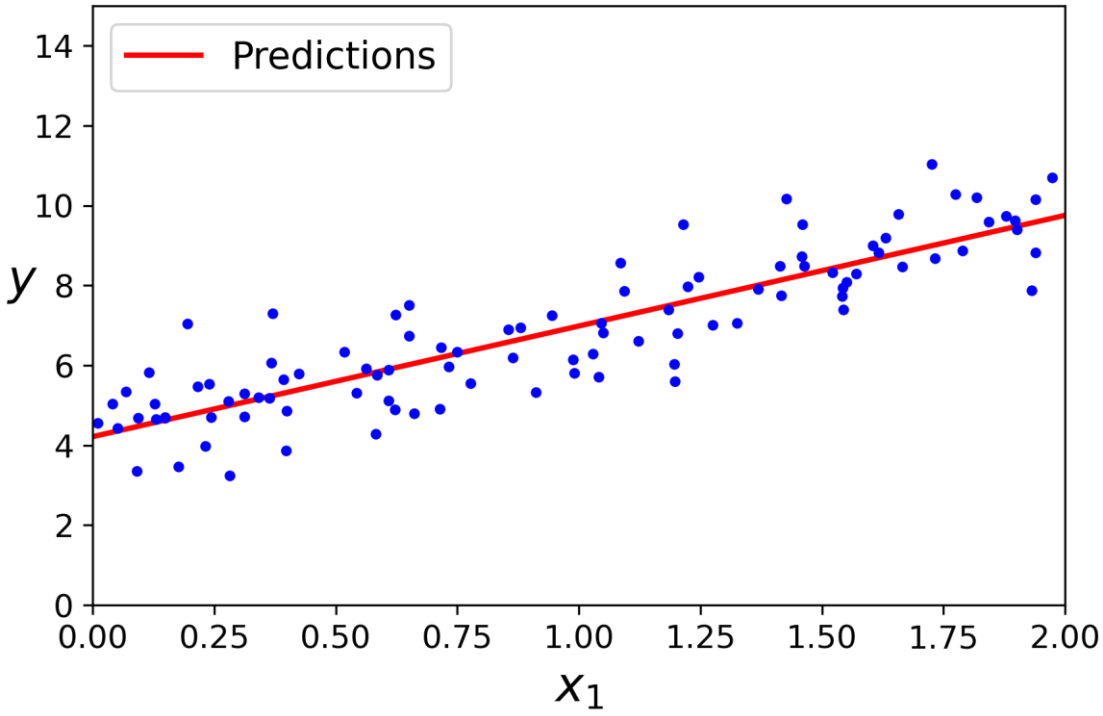
・$\theta_0 = 4$, $\theta_1 = 3$의 값을 기대

・정규방정식으로 계산된 값 -> $\hat{\boldsymbol{\theta}}_{\mathbf{0}} = \mathbf{4.215}, \hat{\boldsymbol{\theta}}_{\mathbf{1}} = \mathbf{2.770}$

4.1 선형 회귀

• 정규방정식으로 얻어진 $\hat{\theta}$을 통해 예측을 진행.

```
In [13]: X_new = np.array([[0], [2]])
         X_new_b = np.c_[np.ones((2, 1)), X_new]  # 모든 샘플에 x0 = 1을 추가합니다.
         y_predict = X_new_b.dot(theta_best)
         y_predict

Out[13]: array([[4.21509616],
                [9.75532293]])
```



4.1 선형 회귀

・사이킷런에서 선형 회귀는 더 간단히 수행될 수 있다.

```
In [16]: 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]]))
```

・모델을 저장한 객체Name.**intercept_:** 선형회귀의 절편값 출력.

・모델을 저장한 객체Name.**coef_**: 선형회귀의 계수값 출력.

```
In [10]: lin_reg.predict(X_new)

         array([[4.21509616],
                [9.75532293]])
```

・모델을 저장한 객체Name.**predict**

-> 만들어진 모델을 통해 새로운 값 x에 대한 예측.

4.1 선형 회귀

```
In [11]: # 싸이파이 lstsq() 함수를 사용하려면 scipy.linalg.lstsq(X_b, y)와 같이 씁니다.
         theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_b, y, rcond=1e-6)
         theta_best_svd

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

・LinearRegression클래스는 scipy.linalg.lstsq() 함수를 기반으로 한다.

・$\hat{\theta} = X^{+}y$를 계산한다. (X^{+} = X의 유사역행렬)

```
In [12]: np.linalg.pinv(X_b).dot(y)

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

・np.linalg.pinv()함수를 이용해 유사역행렬을 직접 구할 수 있다.

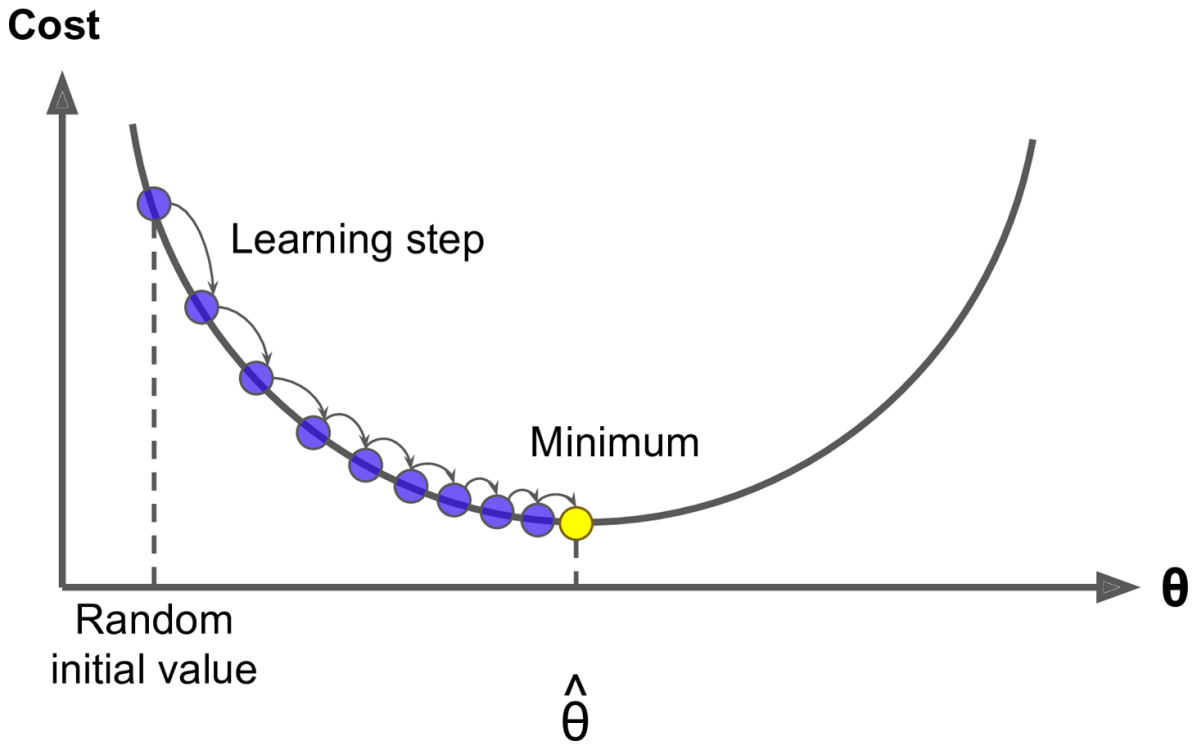
・정규방정식의 경우 특성 수가 두 배로 늘어나면 계산 시간도 대략 두 배 늘어난다.

4.2 경사하강법

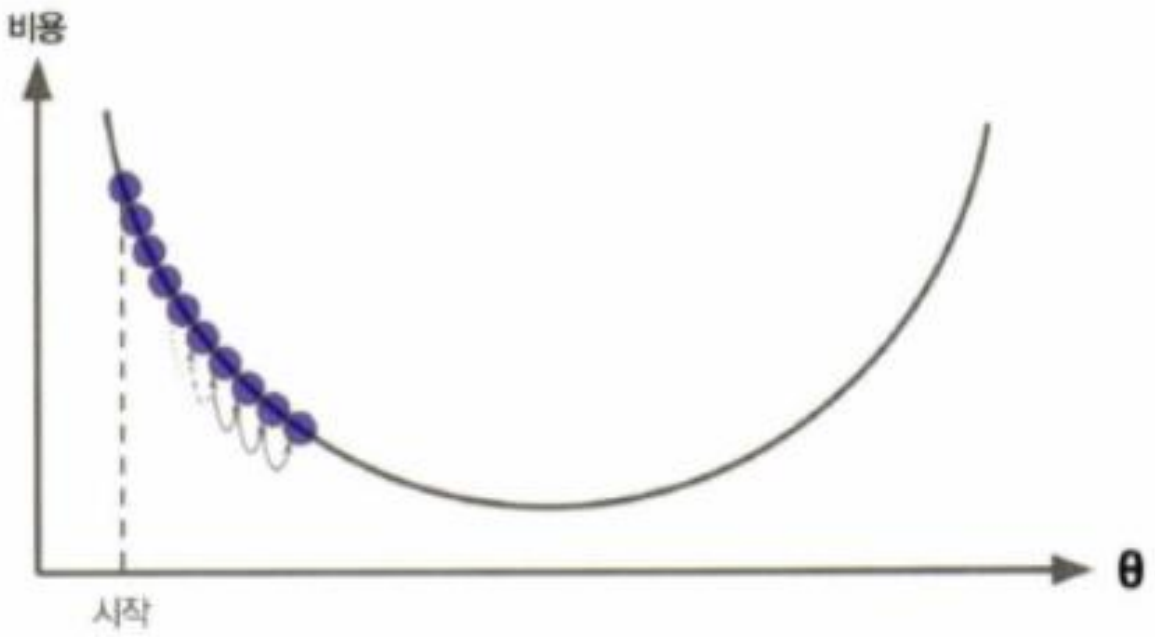
- 특성이 매우 많고 훈련 샘플이 너무 많아 메모리에 담을 수 없을 때 적합하다.

- 여러 종류의 문제에서 **최적의 해법**을 찾을 수 있는 일반적인 최적화 알고리즘이다.

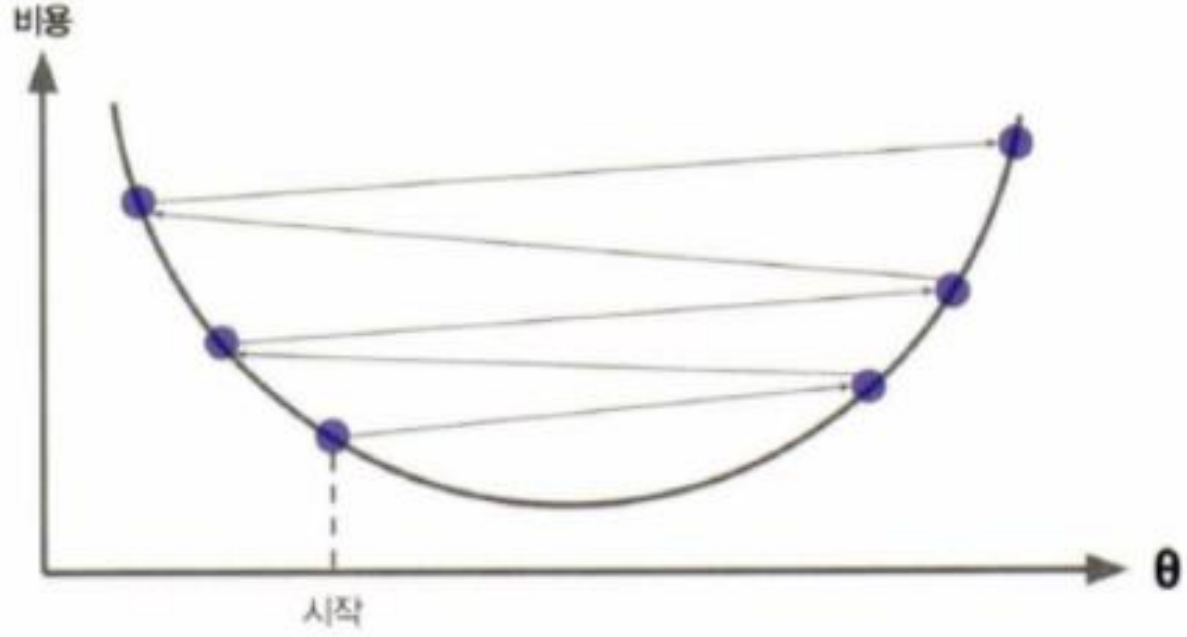
-> 기본 아이디어는 비용 함수를 최소화하기 위해 반복해서 파라미터를 조정해가는 방법이다.



4.2 경사하강법

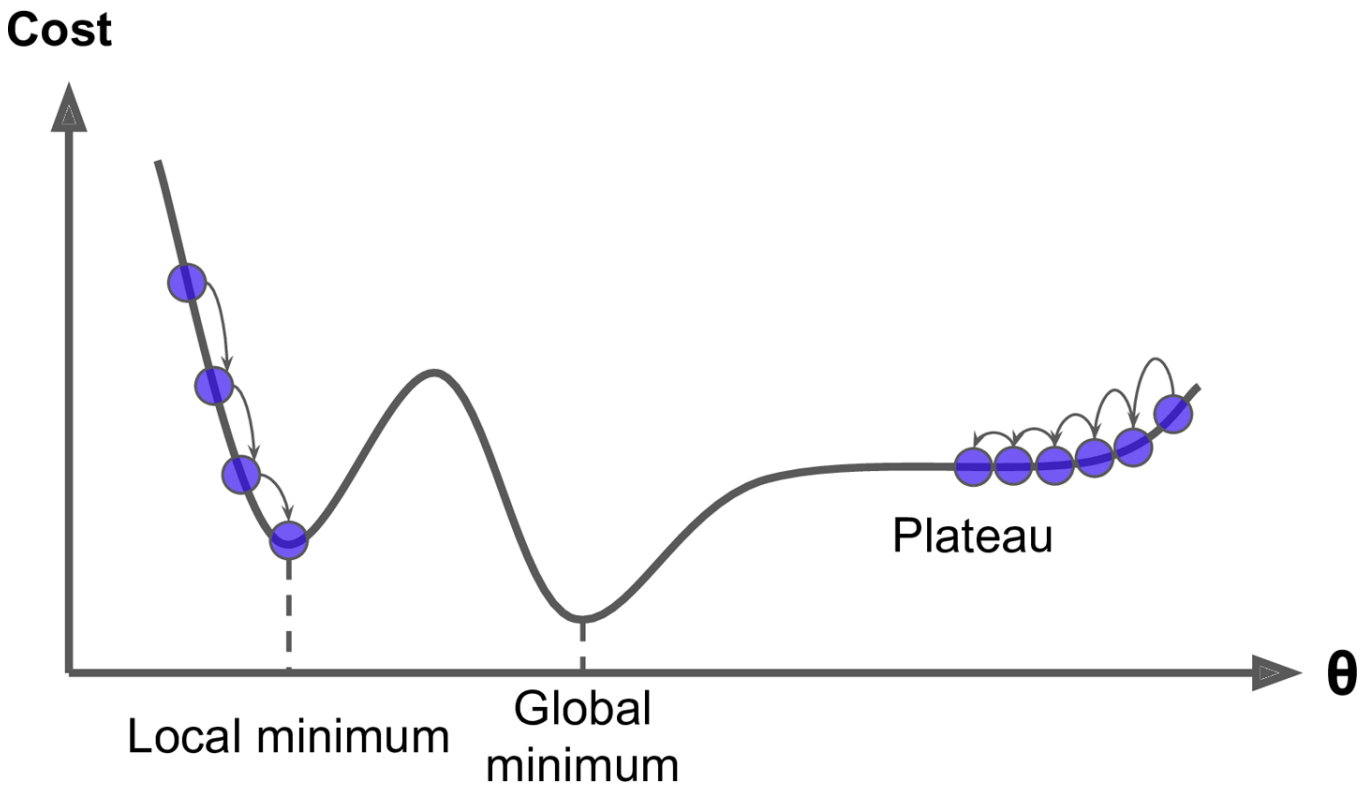


· 학습률이 너무 작은 경우



· 학습률이 너무 큰 경우

4.2 경사하강법



· 왼쪽에서 시작하면 지역 최솟값에 수렴.

· 오른쪽에서 시작하면 시간이 오래 걸리고 일찍 멈추게 되어 전역 최솟값에 도달하지 못함.

But,

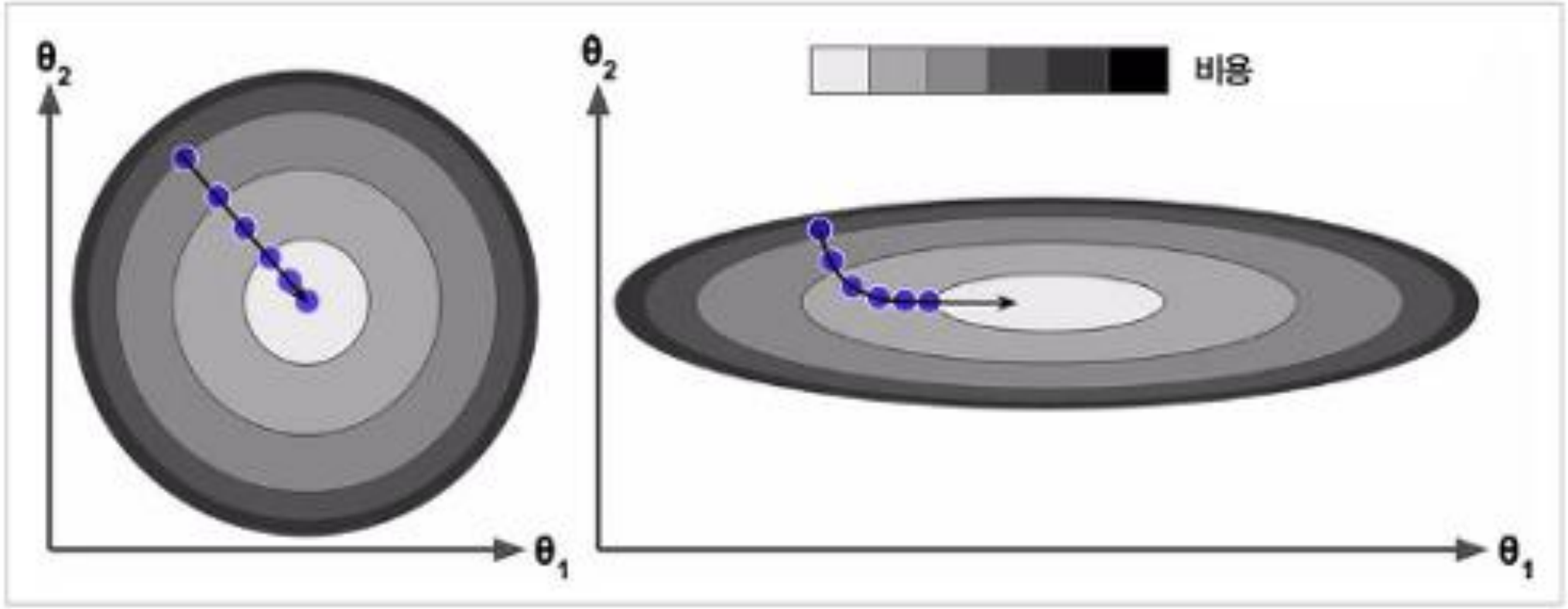
· 선형 회귀의 MSE비용 함수는 볼록 함수 형태.

· 하나의 전역 최솟값만 있으며, 기울기가 갑자기 변하지 않음.

· 경사 하강법이 전역 최솟값에 가깝게 접근할 수 있다는 것을 보장.

4.2 경사하강법

그림 4-7 특성 스케일에 따른 경사 하강법



- 왼쪽 그래프는 특정 스케일을 적용한 방법으로 경사 하강법 알고리즘이 최솟값으로 빠르게 도달한다.

- 오른쪽 그래프는 처음엔 최솟값의 방향으로 직각으로 향하다가 평면한 골짜기를 길게 돌아서 나간다.

4.2 경사하강법

・경사하강법을 구현하려면 각 모델 파라미터 θ_j에 대해 비용함수의 그라디언트를 계산해야한다.

・비용 함수의 편도함수는 다음과 같다.

$$\frac{\partial}{\partial \theta_j} \mathrm{M}SE(\theta) = \frac{2}{m} \sum_{i=1}^{m} (\theta^T X^{(i)} - y^{(i)}) x_j{}^{(i)}$$

・모델 파라미터θ_j가 많아질 경우,
편도함수를 위의 식처럼 각각 계산하지 않고 한꺼번에 계산할 수 있다.

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

4.2 경사하강법

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

・이 공식은 매 경사 하강법 스텝에서 전체 훈련 세트X에 대해 계산한다.

・이 알고리즘을 우리는 **배치 경사 하강법** 이라고 한다.

・위로 향하는 그라디언트 벡터가 구해지면 반대 방향인 아래로 가야한다.

・이때, 학습률 η가 사용된다.

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

4.2 경사하강법

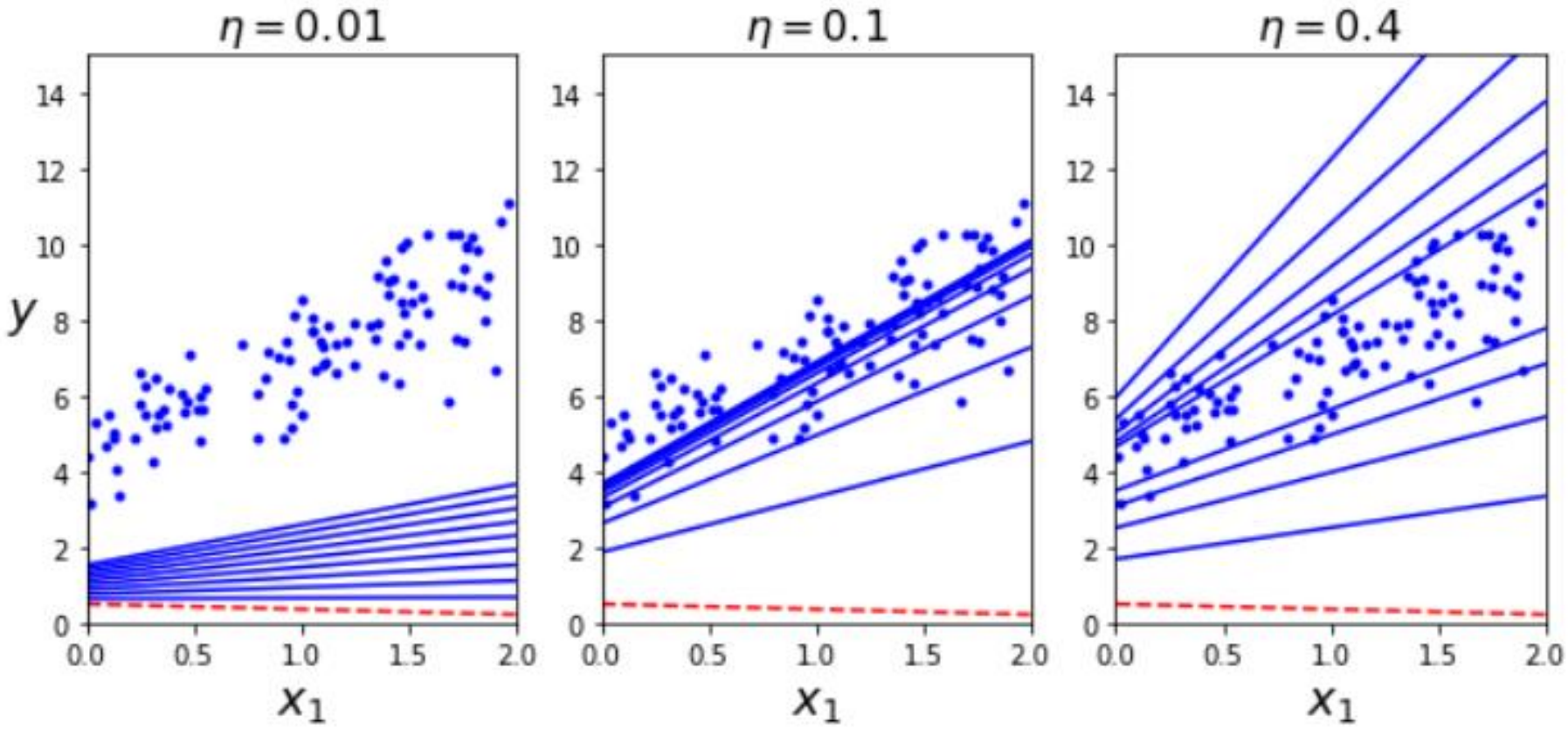
```
In [18]: eta = 0.1  # 학습률
         n_iterations = 1000
         m = 100

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

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

```
In [14]: theta

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

- eta: 학습률
- n_iterations: 반복횟수
- m: 샘플 수

$\theta_0 = 4.215, \theta_1 = 2.770$

4.2 경사하강법



4.2 경사하강법

・경사하강법의 반복 횟수를 정하는 것 또한 중요하다.

・너무 **작은** 경우 최적점에 도달하기 전에 알고리즘이 멈춘다.

・너무 **큰** 경우 파라미터가 변하지 않는 동안 시간을 낭비한다.

・반복횟수를 크게 지정하고 그라디언트 벡터 (벡터의 norm)이 특정 ε(**허용오차)**보다 작아지면 알고리즘을 중지한다.

4.2 경사하강법

· 배치 경사 하강법의 가장 큰 문제는 매 스텝에서 전체 훈련세트를 사용해 그라디언트를 계산한다.

-> 훈련 세트가 커지면 매우 느려지게 된다.

· **확률적 경사 하강법**: 매 스텝에서 한 개의 샘플을 무작위로 선택, 그 하나의 샘플에 대한 그라디언트 계산.

▶ 배치 경사 하강법에 의해서 알고리즘이 훨씬 빠르다.

▶ 매우 큰 훈련 세트도 훈련 시킬 수 있다.

But,

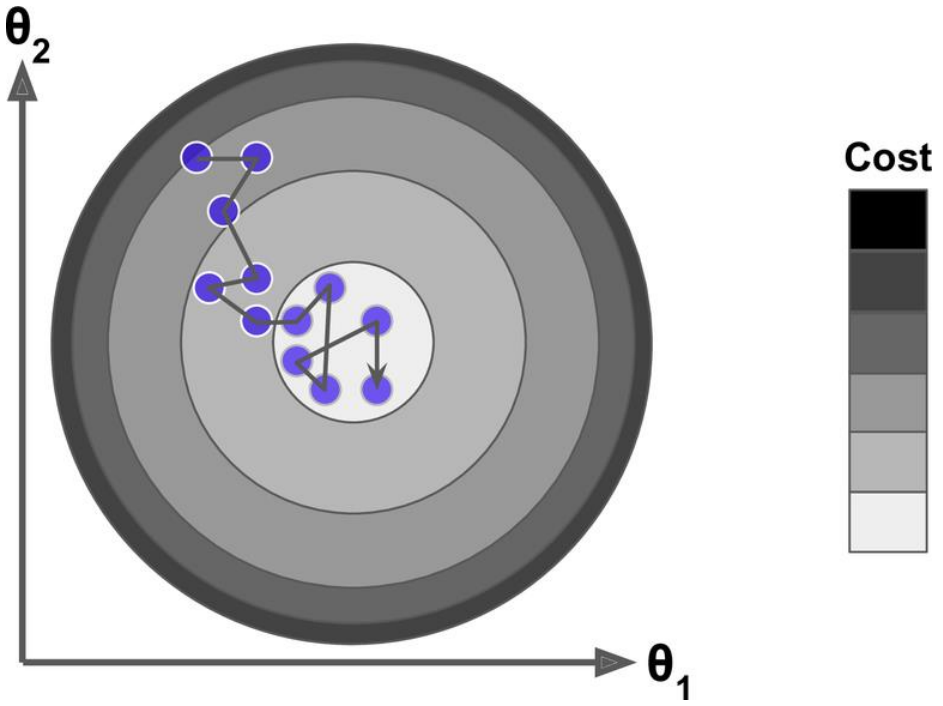
▶ 배치 경사 하강법보다 훨씬 불안정하다.

▶ 무작위성은 지역 최솟값에서 탈출시켜 주지만,
전역 최솟값에 다다르지 못하게 한다는 점에서 좋지않다.

Thus,

▶ 학습률을 점진적으로 감소시킨다. (담금질 기법)
- 시작할 때는 학습률을 크게 하고 점차 감소시킨다.

· **학습 스케줄:** 매 반복에서 학습률을 결정하는 함수



4.2 경사하강법

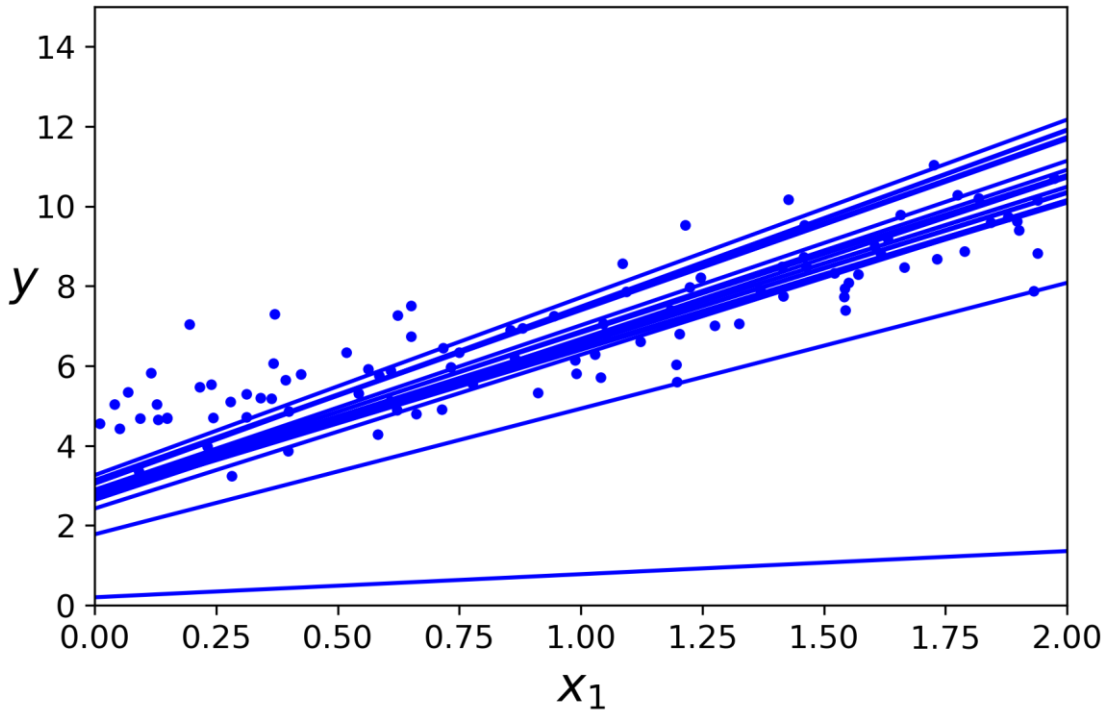
```
In [22]: n_epochs = 50
         t0, t1 = 5, 50  # 학습 스케줄 하이퍼파라미터

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

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

         for epoch in range(n_epochs):
             for i in range(m):
                 if epoch == 0 and i < 20:                    # 책에는 없음
                     y_predict = X_new_b.dot(theta)           # 책에는 없음
                     style = "b-" if i > 0 else "r--"         # 책에는 없음
                     plt.plot(X_new, y_predict, style)        # 책에는 없음
                 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
                 theta_path_sgd.append(theta)                 # 책에는 없음
```

```
In [23]: theta

Out[23]: array([[4.21076011],
                [2.74856079]])
```

- 일반적으로 한 반복에서 m번 되풀이 되고, 이때 각 반복하는 횟수를 **epoch**이라고 한다.

- 경사 하강법 코드가 전체 훈련 세트에 대해 1,000번 반복하는 동안
  훈련 세트에서 50번만 반복하고도 매우 좋은 값에 도달했다.

4.2 경사하강법



• 훈련 스텝의 첫 20개를 보여준다. (스텝이 불규칙하게 진행된다.)

4.2 경사하강법

・사이킷런에서 확률적 경사 하강법(SGD)방식으로 선형회귀를 사용할 수 있다.

・**max_iter**: (최대 에포크)

・**tol**: 허용 오차 (지정한 값 보다 손실이 줄어들 때 까지 실행.)

・**eta0**: 기본 학습 스케줄

・**Penalty**: 규제

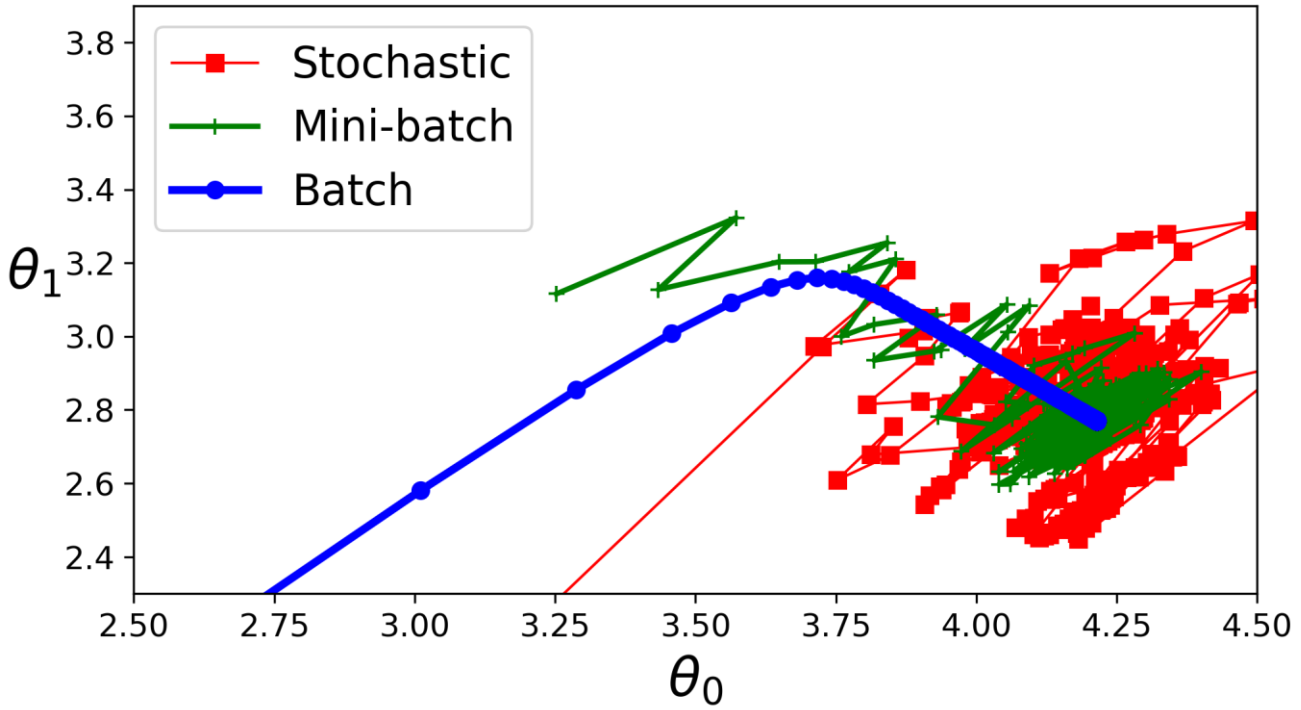
```
In [21]: from sklearn.linear_model import SGDRegressor

         sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3, penalty=None, eta0=0.1, random_state=42)
         sgd_reg.fit(X, y.ravel())

Out[21]: SGDRegressor(eta0=0.1, penalty=None, random_state=42)

In [22]: sgd_reg.intercept_, sgd_reg.coef_

Out[22]: (array([4.24365286]), array([2.8250878]))
```

4.2 경사하강법

· **미니배치 경사 하강법**: 미니배치라 부르는 임의의 작은 샘플 세트에 대해 그라디언트를 계산한다.

· 확률적 경사 하강법에 비해 행렬 연산에 최적화된 하드웨어, 특히 GPU를 사용해서 성능 향상을 얻을 수 있다.

· 미니배치를 어느 정도 크게 하면 SGD보다 덜 불규칙하게 움직인다.

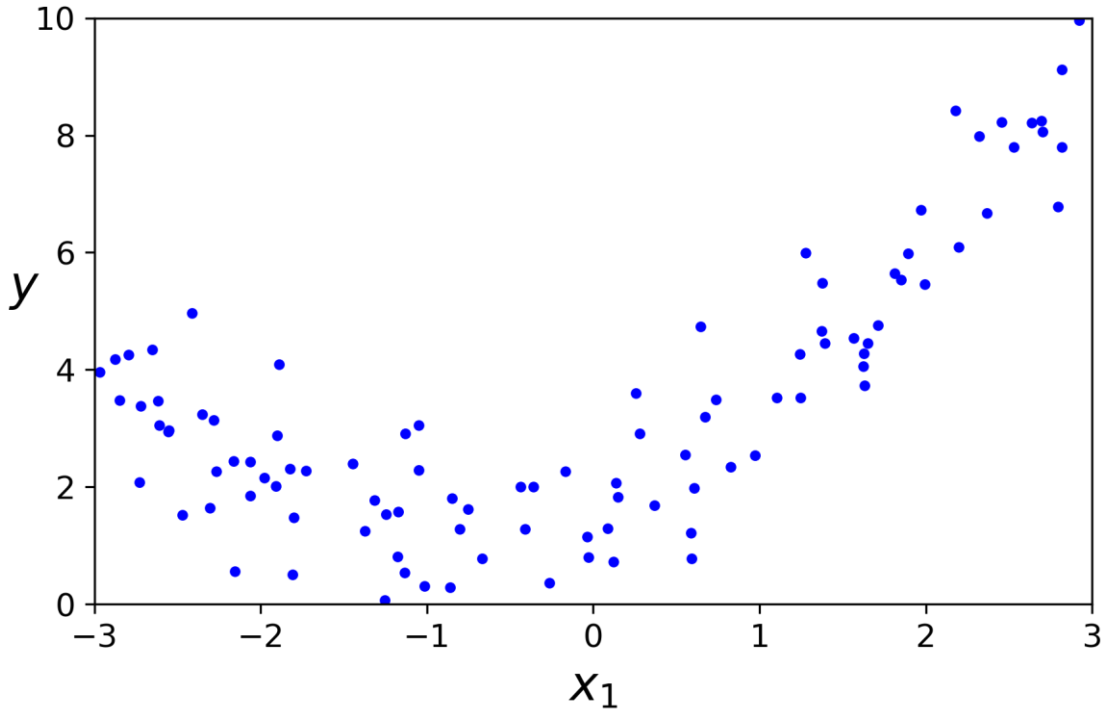


4.3 다항 회귀

· **다항 회귀:** 비선형 데이터를 학습하는 데 사용되는 선형 모델.

· 간단한 방법은 각 특성의 거듭제곱을 새로운 특성으로 추가하고 확장된 특성을 포함한 데이터셋에 선형 모델을 훈련.

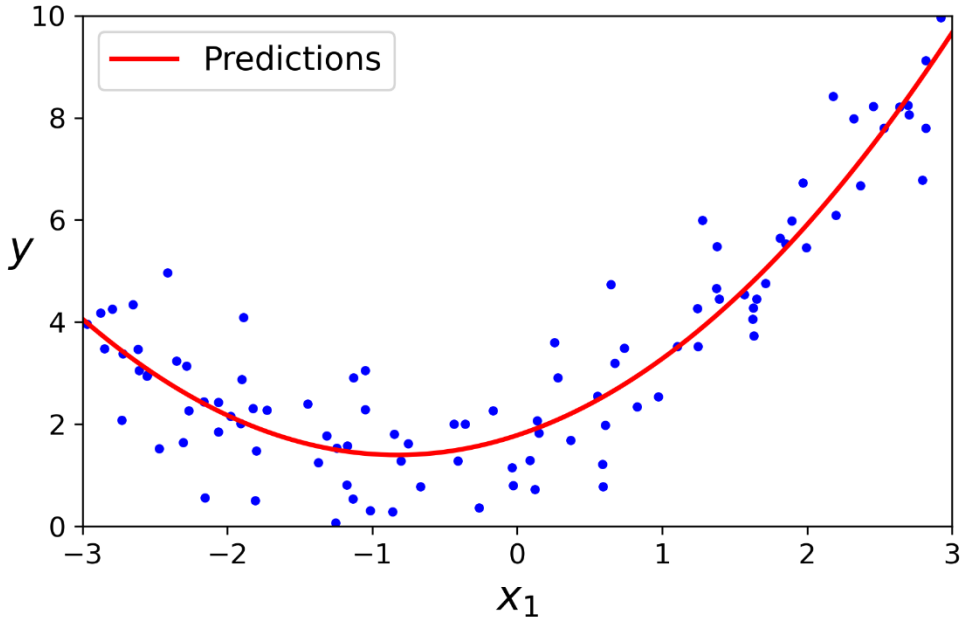
```
In [51]: m = 100
         X = 6 * np.random.rand(m, 1) - 3
         y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```



4.3 다항 회귀

```
In [53]: 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])

In [31]: X_poly[0]

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

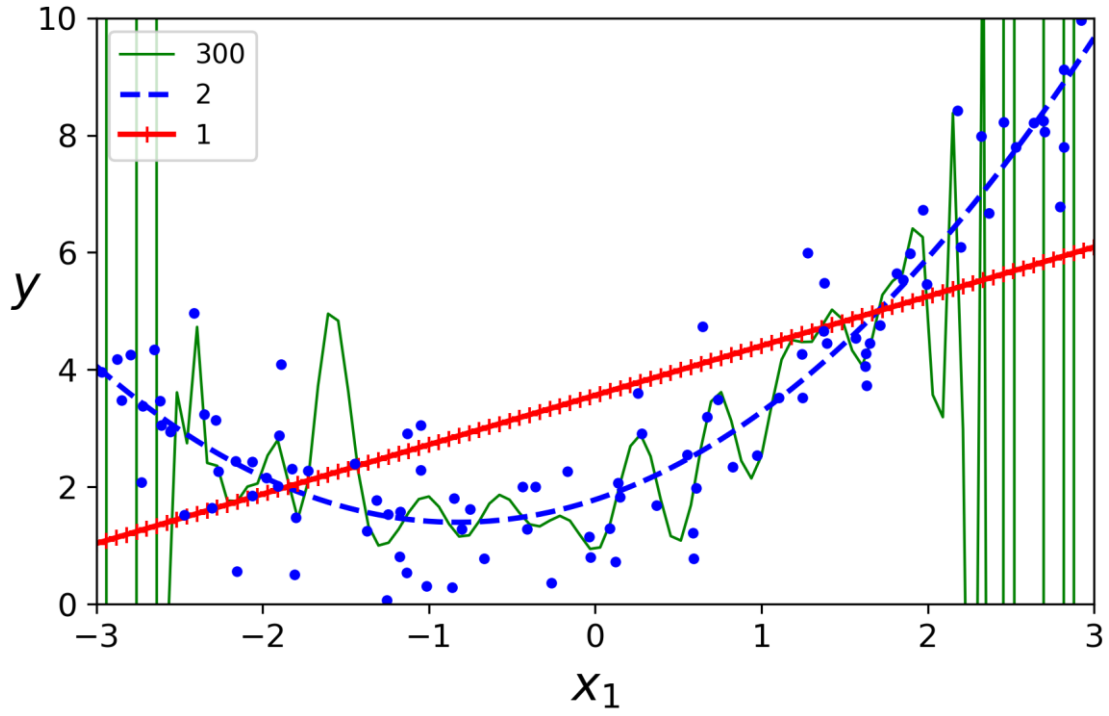
```
In [57]: lin_reg = LinearRegression()
         lin_reg.fit(X_poly, y)
         lin_reg.intercept_, lin_reg.coef_

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



- 실제 함수는 $y = 0.5{x_1}^2 + 1.0x_1 + 2.0 +$ 가우시안 잡음
- 예측 함수는 $\hat{y} = 0.56{x_1}^2 + 0.93x_1 + 1.78$

4.4 학습 곡선

· 고차 다항 회귀는 보통의 선형 회귀보다 훨씬 더 훈련 데이터에 잘 맞추려는 특성이있다.



· 고차 다항 회귀 모델은 심각하게 훈련 데이터에 과대적합. 선형 모델은 과소 적합.

· 모델의 일반화 성능을 추정하기 위해 **교차 검증** 사용.

▶ 훈련 데이터에서 성능 Good, 교차 검증 점수 Bad = **과대적합**

▶ 훈련 데이터에서 성능 Bad, 교차 검증 점수 Bad = **과소적합**

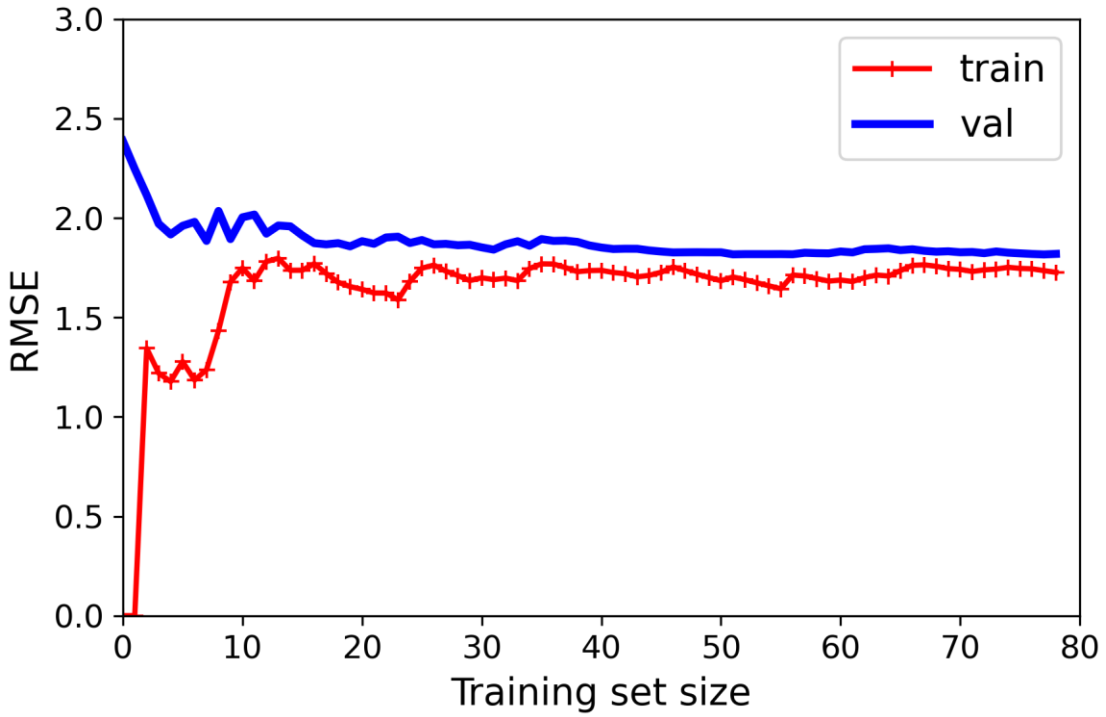
4.4 학습 곡선

- **학습 곡선:** 훈련 세트와 검증 세트의 모델 성능을 훈련 세트 크기의 함수로 표현.

- 단순히 훈련 세트에서 크기가 다른 서브 세트를 만들어 모델을 여러 번 훈련시키면 된다.

In [60]:
```
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))

    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
    plt.legend(loc="upper right", fontsize=14)   # 책에는 없음
    plt.xlabel("Training set size", fontsize=14) # 책에는 없음
    plt.ylabel("RMSE", fontsize=14)              # 책에는 없음
```

4.4 학습 곡선



· **훈련** 데이터의 성능

▶ 훈련 세트에 하나 혹은 두 개의 샘플이 있을 땐 완벽하게 작동.

▶ 샘플이 추가 됨에 따라 잡음 & 비선형이기 때문에 완벽한 학습 불가.

▶ 곡선이 어느 정도 평평해질 때 까지 오차가 계속 상승

· **검증** 데이터의 성능

▶ 적은 수의 샘플의 경우, 제대로 일반화가 불가능. 검증 오차↑

▶ 샘플이 추가 됨에 따라서 학습이 되어 검증 오차 ↓

▶ 선형 회귀의 직선은 데이터를 잘 모델링 할 수 없어서 오차의 감소가 완만.

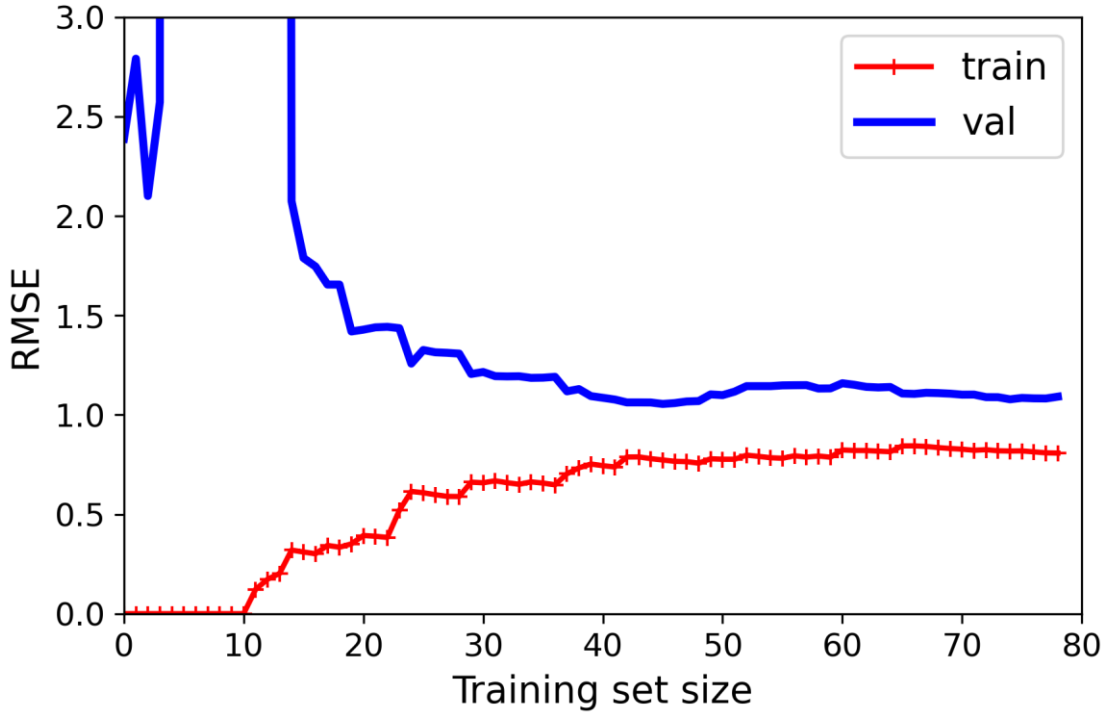
· **단순 선형 회귀 모델(직선)의 학습 곡선.**

· 학습 곡선이 과소적합된 모델의 전형적인 모습이다.

· 두 곡선이 수평한 구간을 만들고 꽤 높은 오차에서 매우 가까이 근접.

4.4 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)
```



· 10차 다항 회귀의 학습 곡선.

· 훈련 데이터의 오차가 선형 회귀 모델보다 훨씬 낮다.

· 두 곡선 사이에 공간이 있다. → 훈련 데이터에서의 성능이 검증 데이터셋보다 훨씬 낮다 → 과대적합 모델

· 더 큰 훈련 세트를 사용하면 두 곡선이 가까워진다.

4.5 규제가 있는 선형 모델

· 과대적합을 감소시키는 좋은 방법을 모델을 규제하는 것이다.

· 다항 회귀 모델을 규제하는 간단한 방법은 다항식의 차수를 감소시키는 것이다.

· 선형 회귀 모델에서는 보통 모델의 가중치를 제한함으로써 규제를 가한다.

· **릿지 회귀**: 규제가 추가된 선형 회귀 버전

▶ 규제항 $\alpha \sum_{i=1}^{n} {\theta_i}^2$ 이 비용 함수에 추가된다.

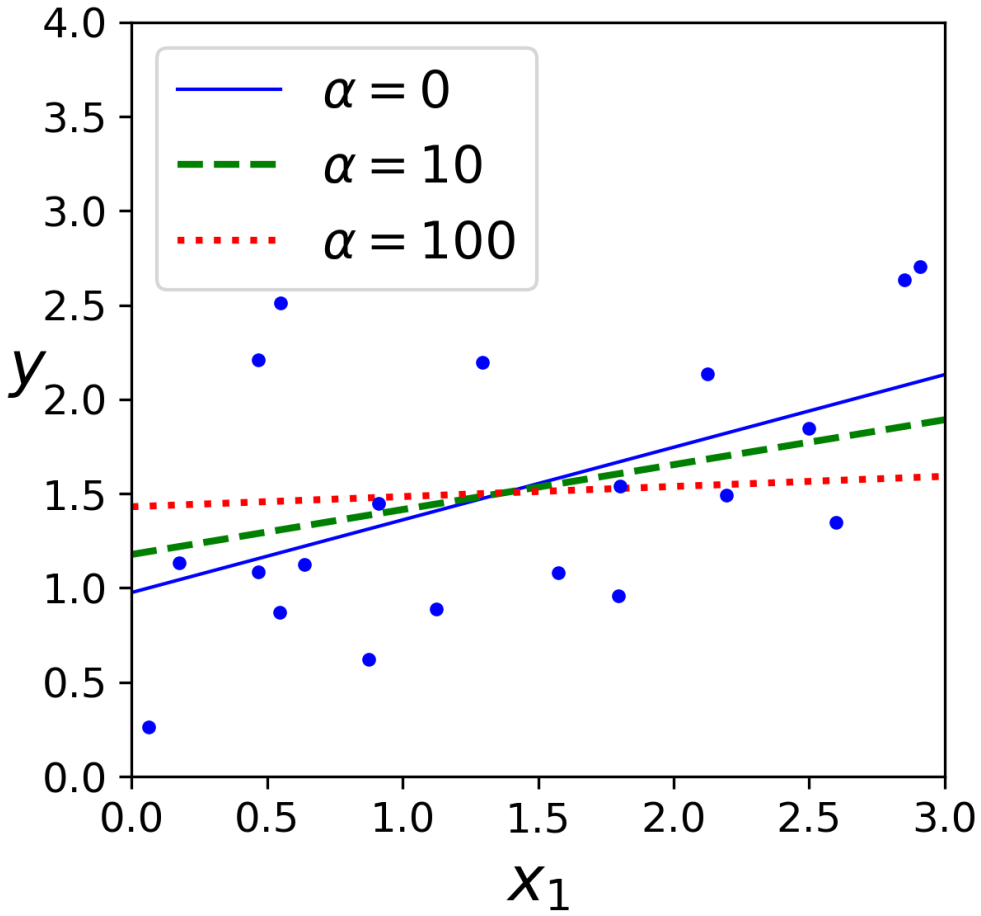
▶ 규제항은 학습 알고리즘을 데이터에 맞추는 것과 더불어 모델의 가중치가 가능한 작게 유지 되도록 노력한다.

▶ 규제항은 훈련하는 동안에만 비용 함수에 추가된다. → 모델의 성능은 규제가 없는 성능 지표로 평가.

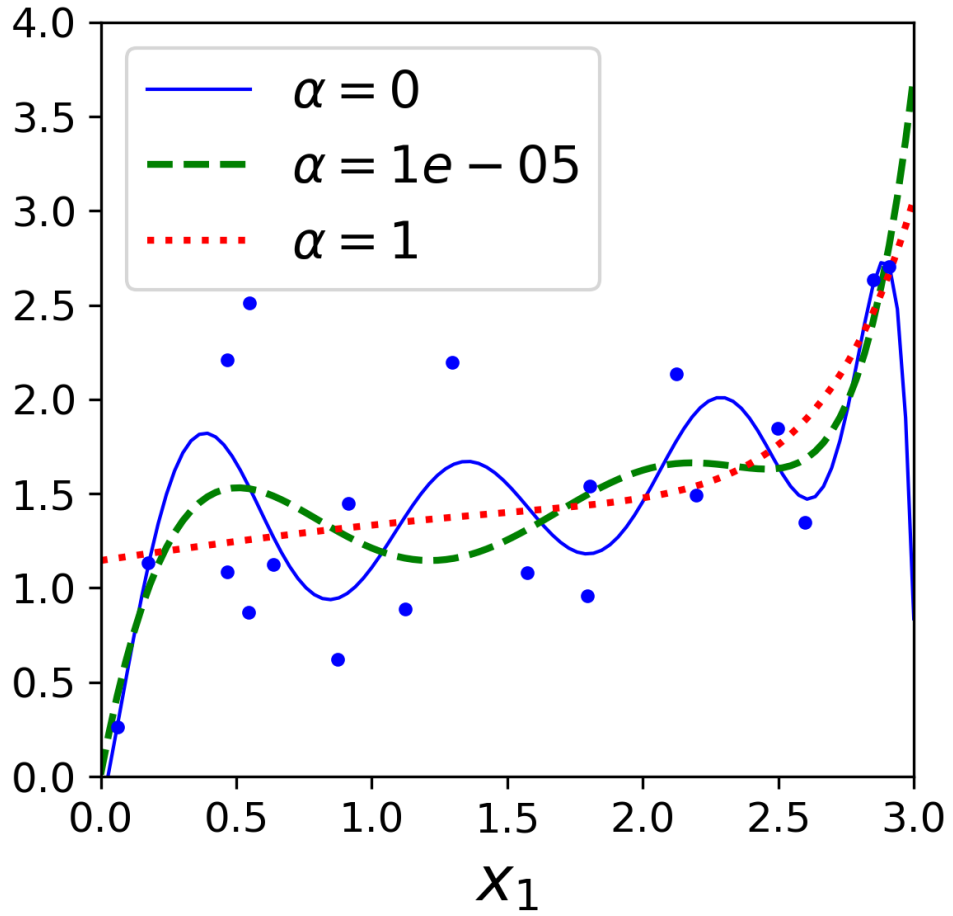
· 릿지 회귀의 비용 함수는 다음과 같다. α는 모델을 얼마나 규제할지 조절한다. (α=0, 선형 회귀와 같다.)

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

4.5 규제가 있는 선형 모델



· 릿지 규제를 사용한 선형 회귀



· 릿지 규제를 사용한 다항 회귀

· α를 증가시킬 수록 모델의 분산은 줄고 편향은 커지게 된다.

4.5 규제가 있는 선형 모델

```
In [67]: from sklearn.linear_model import Ridge
         ridge_reg = Ridge(alpha=1, solver="cholesky", random_state=42)
         ridge_reg.fit(X, y)
         ridge_reg.predict([[1.5]])

         array([[1.55071465]])
```

· 정규 방정식을 사용한 릿지회귀

```
In [42]: sgd_reg = SGDRegressor(penalty="l2", max_iter=1000, tol=1e-3, random_state=42)
         sgd_reg.fit(X, y.ravel())
         sgd_reg.predict([[1.5]])

         array([1.47012588])
```

· 확률적 경사 하강법을 사용한 릿지 회귀

· Penalty 매개변수는 사용할 규제를 지정한다.

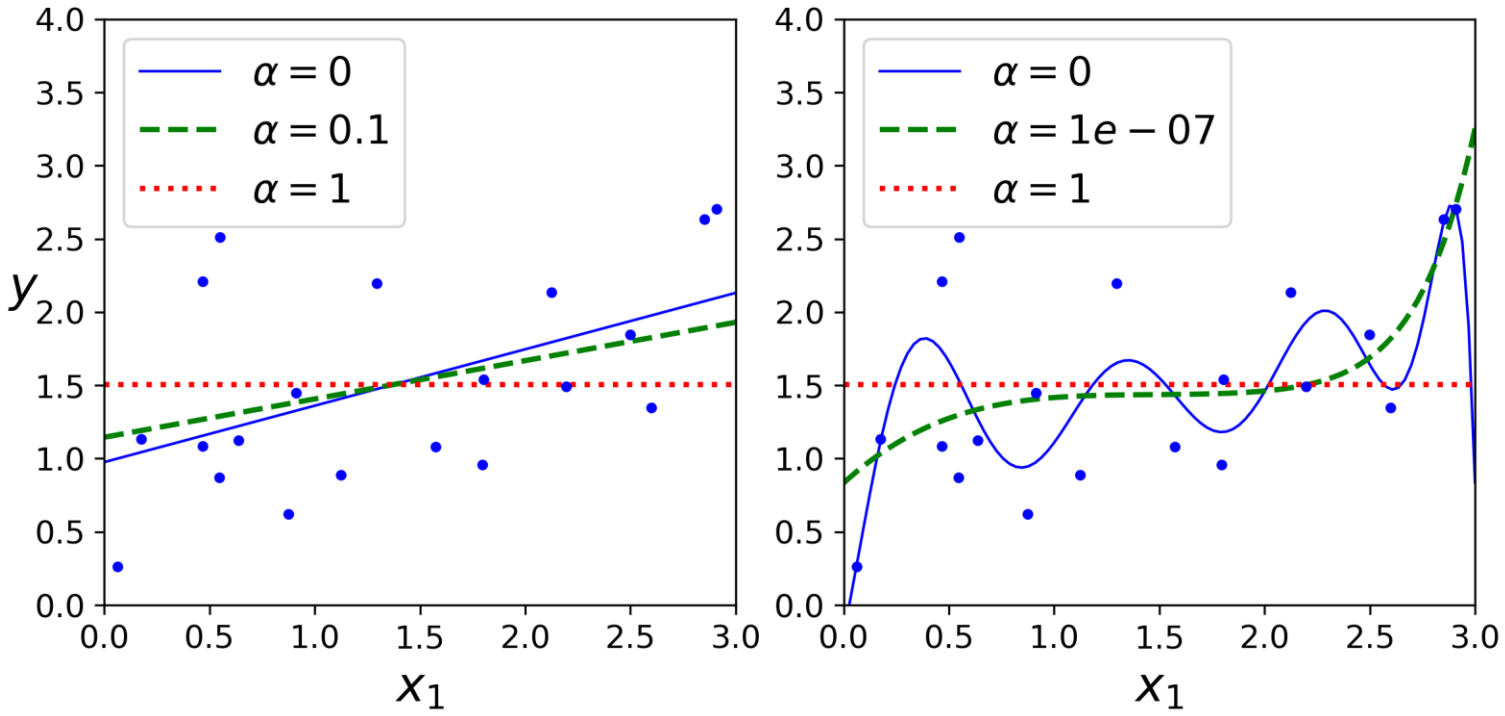
· "l2"는 SGD가 비용 함수에 가중치 벡터의 l_2 노름의 제곱을 2로 나눈 규제항을 추가하게 만든다.

4.5 규제가 있는 선형 모델

· **라쏘 회귀:** 선형 회귀의 또 다른 규제된 버전이다.

▶ 릿지 회귀처럼 비용 함수에 규제항을 더하지만 l_2노름의 제곱을 2로 나눈 것 대신 가중치 벡터 l_1노름을 사용한다.

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



· 라쏘 휘귀의 중요한 특징은 덜 중요한 특성의 가중치를 제거하려고 한다는 점이다. (즉, 가중치가 0이 된다.)

· 라쏘 회귀는 자동으로 특성 선택을 하고 희소 모델을 만든다.

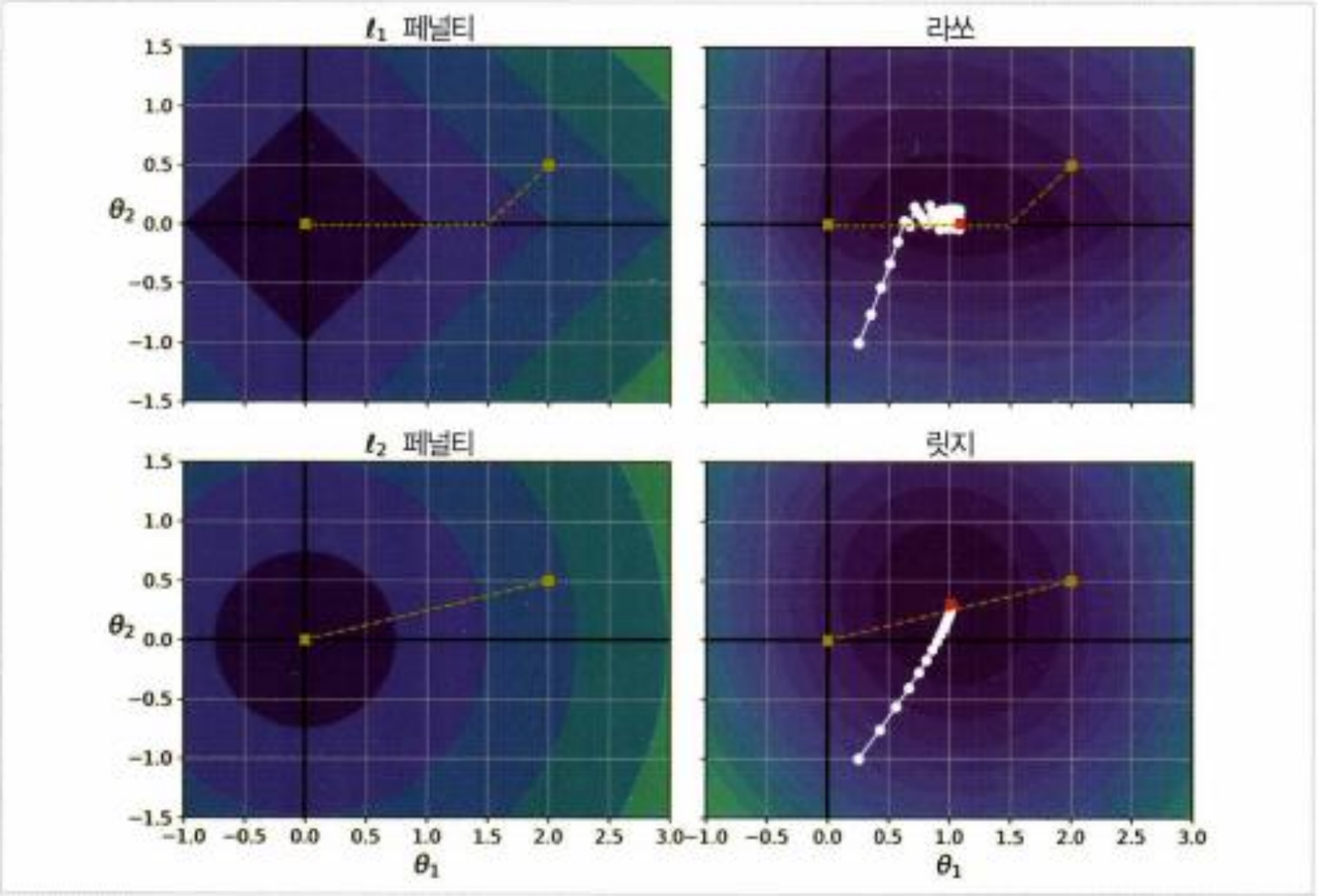


그림 4-19 라쏘 대 릿지 규제

4.5 규제가 있는 선형 모델

- 라쏘의 비용함수는 $\theta_i = 0\ (i = 1,2,\dots,n$일 때)에서 미분 가능하지 않다.
- 하지만, $\theta_i = 0$일 때, **서브그레이디언트 벡터 g**를 사용하면 경사 하강법을 적용하는데 문제가 없다.
  - ▶ 미분이 불가능한 지점 근방 그레디언트들의 중간값으로 생각 가능하다.

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

```
In [71]: from sklearn.linear_model import Lasso
         lasso_reg = Lasso(alpha=0.1)
         lasso_reg.fit(X, y)
         lasso_reg.predict([[1.5]])

         array([1.53788174])
```

- Laaso 대신 SGDRegressor (penalty = "l1")을 사용할 수도 있다.

4.5 규제가 있는 선형 모델

· **엘라스틱넷:** 릿지 회귀와 라쏘 회귀를 절충한 모델이다.

▶ 규제항은 릿지와 라쏘의 규제항을 단순히 더해서 사용, 혼합 정도는 혼합 비율 r을 사용해 조절한다.

▶ r = 0이면, 릿지 회귀와 같고, r = 1이면, 라쏘 회귀와 같다.

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

· 규제가 약간 있는 것이 대부분의 경우가 좋으므로 일반적으로 평범한 선형 회귀는 피해야 한다.

· 일반적으로 릿지가 기본이 되고 쓰이는 특성이 몇 개 뿐이라면, 라쏘나 엘라스틱넷이 좋다.

▶ 불필요한 특성의 가중치를 0으로 만들기 때문.

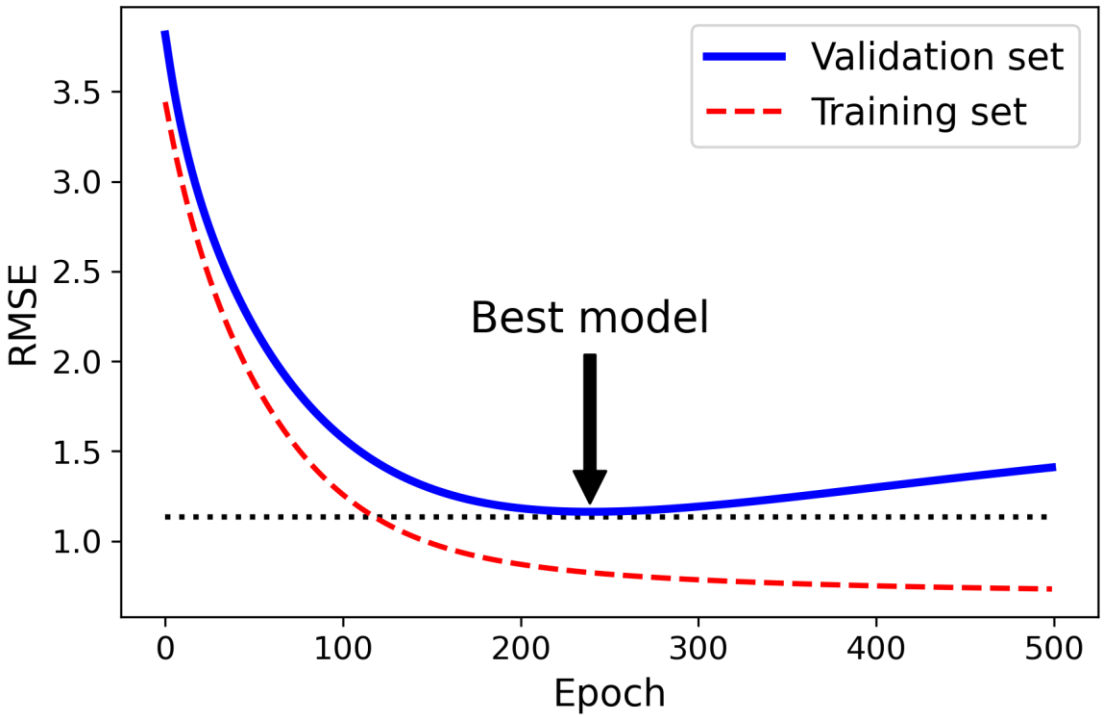
· 특성 수가 훈련 샘플 수보다 많거나, 특성 몇 개가 강하게 연관돼 있을 때는 엘라스틱넷을 더 선호한다.

```
In [73]: from sklearn.linear_model import ElasticNet
         elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5, random_state=42)
         elastic_net.fit(X, y)
         elastic_net.predict([[1.5]])

         array([1.54333232])
```

4.5 규제가 있는 선형 모델

· **조기 종료:** 검증 에러가 최솟값에 바로 도달하면 훈련을 중지시키는 것.

▶경사 하강법과 같은 반복적인 학습 알고리즘을 규제한다.



·에포크가 진행됨에 따라서 RMSE가 감소. But, 다시 상승하는 구간 존재.

▶ **과대적합** 반응

· 검증 에러가 최소에 도달하는 즉시 훈련을 멈춘다.

4.5 규제가 있는 선형 모델

In [75]:
```
from copy import deepcopy

poly_scaler = Pipeline([
        ("poly_features", PolynomialFeatures(degree=90, include_bias=False)),
        ("std_scaler", StandardScaler())
    ])

X_train_poly_scaled = poly_scaler.fit_transform(X_train)
X_val_poly_scaled = poly_scaler.transform(X_val)

sgd_reg = SGDRegressor(max_iter=1, tol=-np.infty, warm_start=True,
                       penalty=None, learning_rate="constant", eta0=0.0005, random_state=42)

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)  # 중지된 곳에서 다시 시작합니다
    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 = deepcopy(sgd_reg)
```

4.6 로지스틱 회귀

· **로지스틱 회귀:** 샘플이 특정 클래스에 속할 확률을 추정하는 데 널리 사용된다.

▶ 예를 들어, 추정 확률이 50%가 넘으면 모델을 그 샘플이 해당 클래스에 속한다고 예측한다.

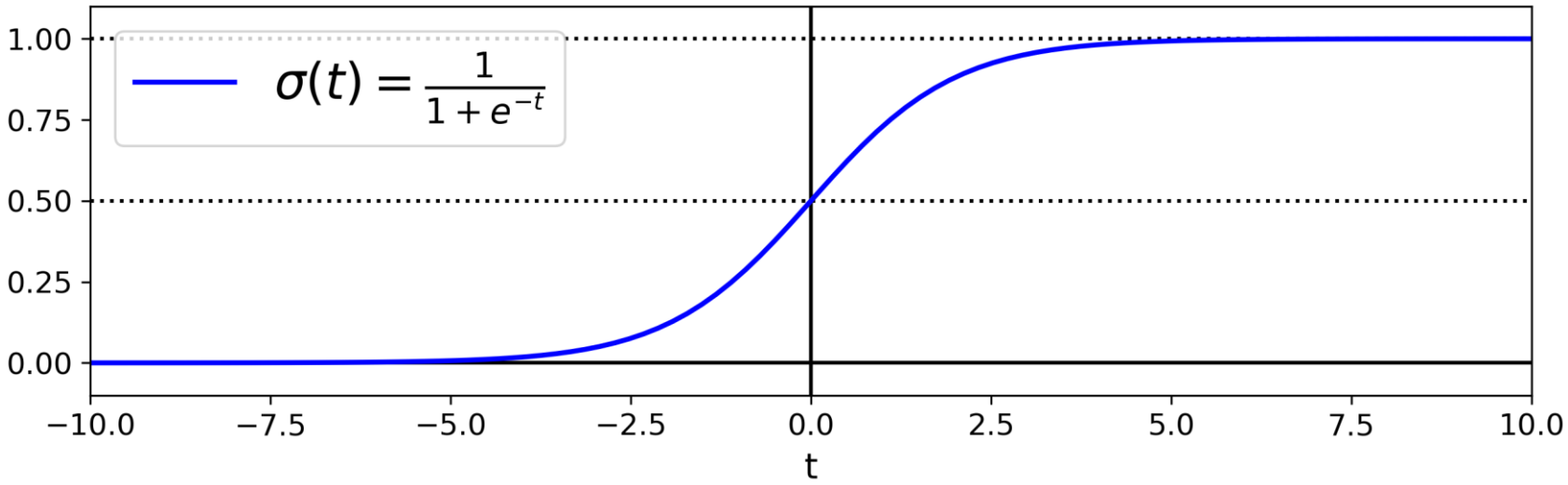
▶ 즉, 레이블이 '1'인 **양성 클래스.** 아니면 클래스에 속하지 않는다고 예측한다. (레이블이 '0'인 **음성 클래스**)

· 로지스틱 회귀 모델의 확률 추정 벡터 표현식은 다음과 같다.

$$\hat{p} = h_\theta(x) = \sigma(\theta^T x)$$

$$\hat{y} = \begin{cases} 0\ (\hat{p} < 0.5\text{일때}) \\ 1\ (\hat{p} \geq 0.5\text{일때}) \end{cases}$$

· 로지스틱 σ는 0과 1사이의 값을 출력하는 **시그모이드 함수(S자 형태)**이다.



· t<0이면, σ(t)<0.5 이므로 0으로 예측.

· t≥ 0이면, σ(t)≥ 0.5 이므로 1로 예측.

4.6 로지스틱 회귀

· 로지스틱 회귀 훈련의 목적은 양성 샘플에 대해서는 높은 확률, 음성 샘플에 대해서는 낮은 확률을 추정하는 모델의 파라미터 벡터 θ를 찾는것이다.

· 하나의 훈련 샘플에 대한 로지스틱 비용 함수는 다음과 같다.

$$c(\theta) = \begin{cases} -\log(\hat{p}) \ y = 1\text{일 때} \\ -\log(1-\hat{p}) \ y = 0 \text{ 일 때} \end{cases}$$

· 양성 샘플을 0에 가까운 확률로 추정 or 음성 샘플을 1에 가까운 확률로 추정할 경우 비용이 커진다.

· 전체 훈련 세트에 대한 비용 함수는 모든 훈련 샘플의 비용을 평균한 것이며, **로그 손실**이라고 부른다.

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

· 이 비용함수의 최솟값을 계산하는 알려진 해는 없다.
· 하지만, 볼록 함수 이므로 경사 하강법을 통해 전역 최솟값을 찾을 수 있다.

$$\frac{\partial}{\partial\theta_j}J(\theta) = \frac{1}{m}\sum_{i=1}^{m}(\sigma\left(\theta^T X^{(i)}\right) - y^{(i)})x_j{}^{(i)}$$

4.6 로지스틱 회귀

```
In [98]: from sklearn import datasets
         iris = datasets.load_iris()
         list(iris.keys())
```

```
['data',
 'target',
 'frame',
 'target_names',
 'DESCR',
 'feature_names',
 'filename']
```

```
In [56]: X = iris["data"][:, 3:]  # 꽃잎 너비
         y = (iris["target"] == 2).astype(np.int)  # Iris virginica이면 1 아니면 0
```

```
In [57]: from sklearn.linear_model import LogisticRegression
         log_reg = LogisticRegression(solver="lbfgs", random_state=42)
         log_reg.fit(X, y)
```

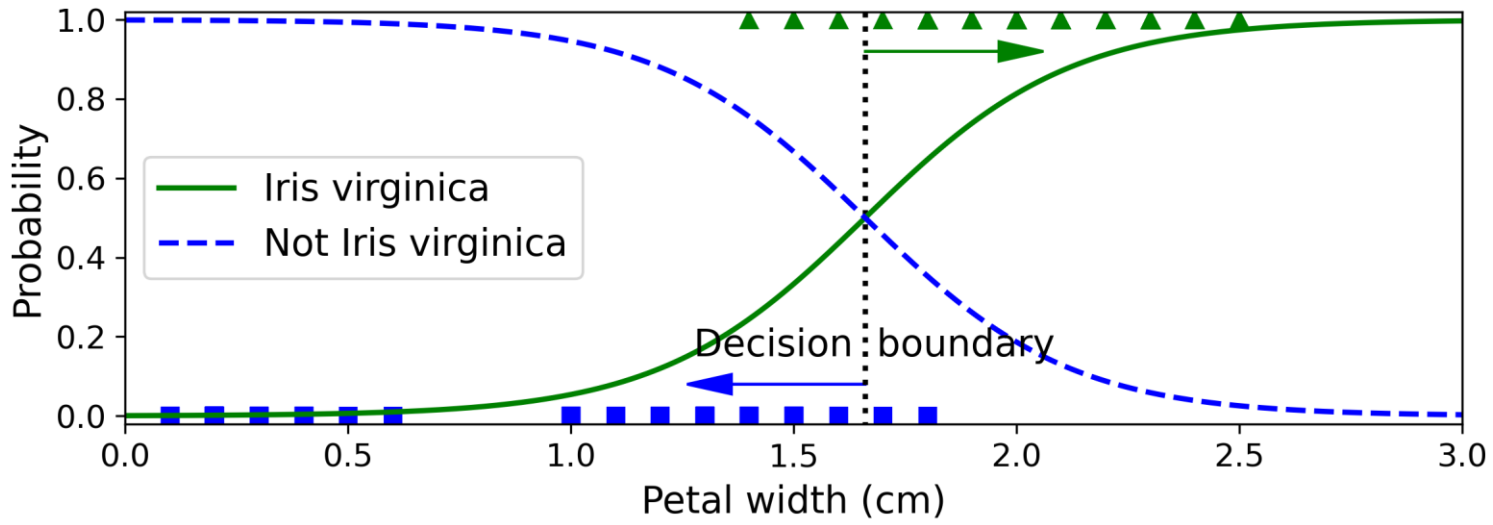
```
LogisticRegression(random_state=42)
```

• 로지스틱 회귀를 설명하기 위해 세 개의 품종을 포함하고 있는 iris 데이터를 사용한다.

4.6 로지스틱 회귀

```
In [108]: X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
          y_proba = log_reg.predict_proba(X_new)
```

- 꽃잎의 너비가 0~3cm인 꽃에 대해 모델의 추정확률 계산.



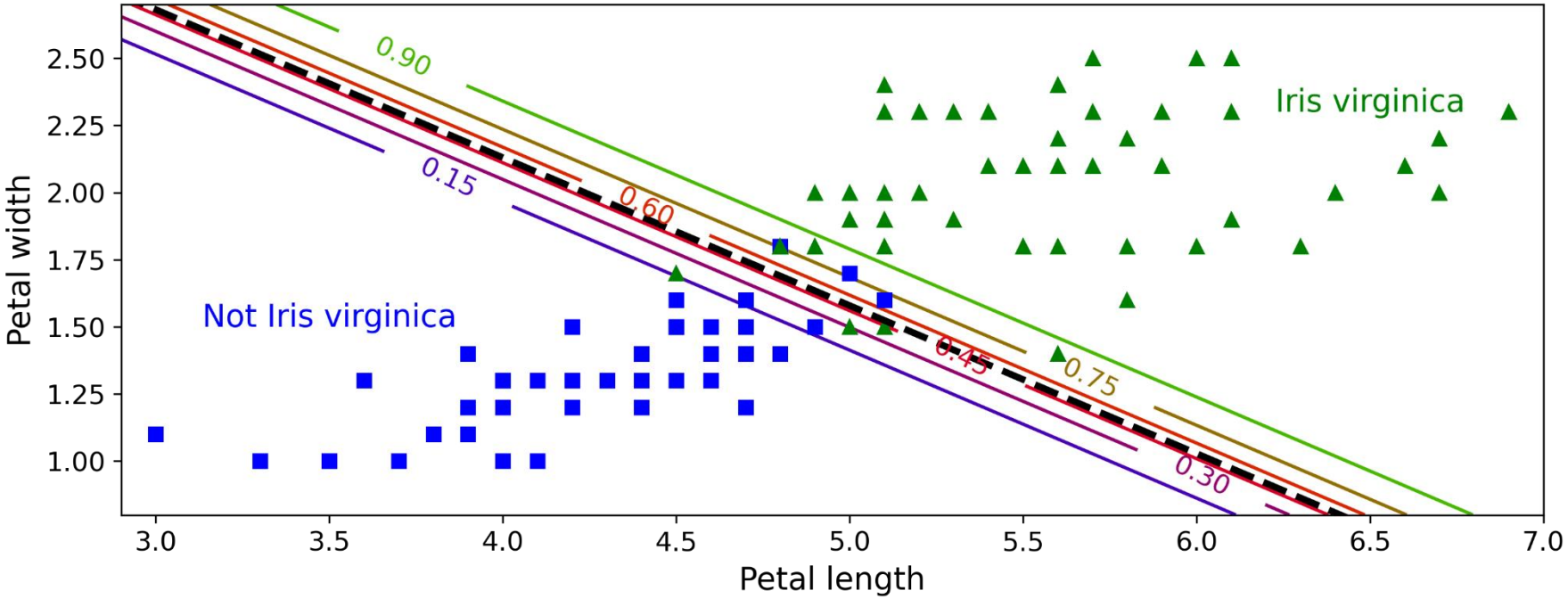
- 꽃잎 너비가 2cm 이상이면 분류기가 Iris-Verginica라고 강하게 확신하고 1cm 아래면 Iris-Verginica가 아니라고 강하게 확신한다.

- 결정 경계는 1.6cm 근방에서 만들어진다. 1.6보다 크면 Iris-Verginica, 1.6보다 작으면 Not-Iris-Verginica.

4.6 로지스틱 회귀

```
In [61]: log_reg.predict([[1.7], [1.5]])

array([1, 0])
```

- 꽃잎의 너비가 1.7인 경우 Iris-Verginica
- 꽃잎의 너비가 1.5인 경우 Not-Iris-Verginica



- 꽃잎의 너비와 꽃잎의 길이 두 개의 특성을 이용하여 분류.
- 모델은 맨 오른쪽 위의 직선을 넘어서 있는 꽃들을 90% 이상의 확률로 iris-Viginica라고 판단.

4.6 로지스틱 회귀

```
In [189]: X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,random_state=42)
```

```
In [196]: from sklearn.metrics import accuracy_score, confusion_matrix,roc_auc_score,roc_curve
          clf_reg = LogisticRegression(solver="lbfgs", random_state=42)
          clf_reg.fit(X_train, y_train)
          y_pred=clf_reg.predict(X_val)
          print(clf_reg.score(X_train,y_train))
          print(clf_reg.score(X_val,y_val))

          print("₩n")
          print(confusion_matrix(y_train,clf_reg.predict(X_train)))
          print("₩n")
          print(confusion_matrix(y_val,y_pred))
```

```
0,95
1,0


[[79  2]
 [ 4 35]]


[[19  0]
 [ 0 11]]
```

- 훈련 데이터의 Accuracy: 95%
- 검증 데이터의 Accuracy: 100%

4.6 로지스틱 회귀

- 소프트맥스 회귀는 타깃 클래스에 대해서는 높은 확률을, 다른 클래스에 대해서는 낮은 확률을 추정하는 것이 목적이다.
- **크로스 엔트로피** 비용 함수를 최소화 하는 것은 이 목적에 부합.
  - ▶ 타깃 클래스에 대해 낮은 확률을 예측하는 모델을 억제.
  - ▶ 추정된 클래스의 확률이 타깃 클래스에 얼마나 잘 맞는지 측정하는 용도로 종종 사용.

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

- $\boldsymbol{y_k}^{(i)} = i$번째 샘플이 클래스 k에 속할 타깃 확률.
(일반적으로 샘플이 클래스에 속하는지 아닌지에 따라 1또는 0이다.)

$$\nabla_{\boldsymbol{\theta}^{(k)}} J(\mathbf{\Theta}) = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{p}_k^{(i)} - y_k^{(i)}\right)\mathbf{x}^{(i)}$$

- 각 클래스에 대한 그레이디언트 벡터를 계산할 수 있으므로 경사 하강법을 사용할 수 있다.

4.6 로지스틱 회귀

· **소프트맥스 회귀(다항 로지스틱 회귀):** 여러 개의 이진 분류기를 훈련시켜 연결하지 안혹 직접 다중 클래스를 지원하도록 일반화시킨 것.

· 샘플 x가 주어지면 소프트맥스 회귀 모델이 각 클래스 k에 대한 점수 $s_k(x)$를 계산.

$s_k(x) = (\theta^{(k)})^T x$ 여기서, $\theta^{(k)}$ = 각 클래스의 파라미터 벡터

· 점수에 소프트맥스 함수를 적용하여 각 클래스의 확률 $\hat{p}_k$를 추정.

$$\hat{p}_k = \sigma(s(x))_k = \frac{\exp(s_k(x))}{\sum_{j=1}^{K} \exp(s_j(x))}$$

· **k**: 클래스 수

· $\boldsymbol{s(x)}$: 샘플 x에 대한 각 클래스의 점수를 담은 벡터

· $\boldsymbol{\sigma(s(x))_k}$: 샘플 x에 대한 각 클래스 점수가 주어졌을 때, 이 샘플이 클래스 k에 속할 추정 확률.

$$\hat{y} = \underset{k}{\operatorname{argmax}}\, \sigma(\mathbf{s}(\mathbf{x}))_k = \underset{k}{\operatorname{argmax}}\, s_k(\mathbf{x}) = \underset{k}{\operatorname{argmax}}\, ((\boldsymbol{\theta}^{(k)})^T \mathbf{x})$$

· 소프트맥스 회귀 분류기는 추정확률이 가장 높은 클래스를 선택.

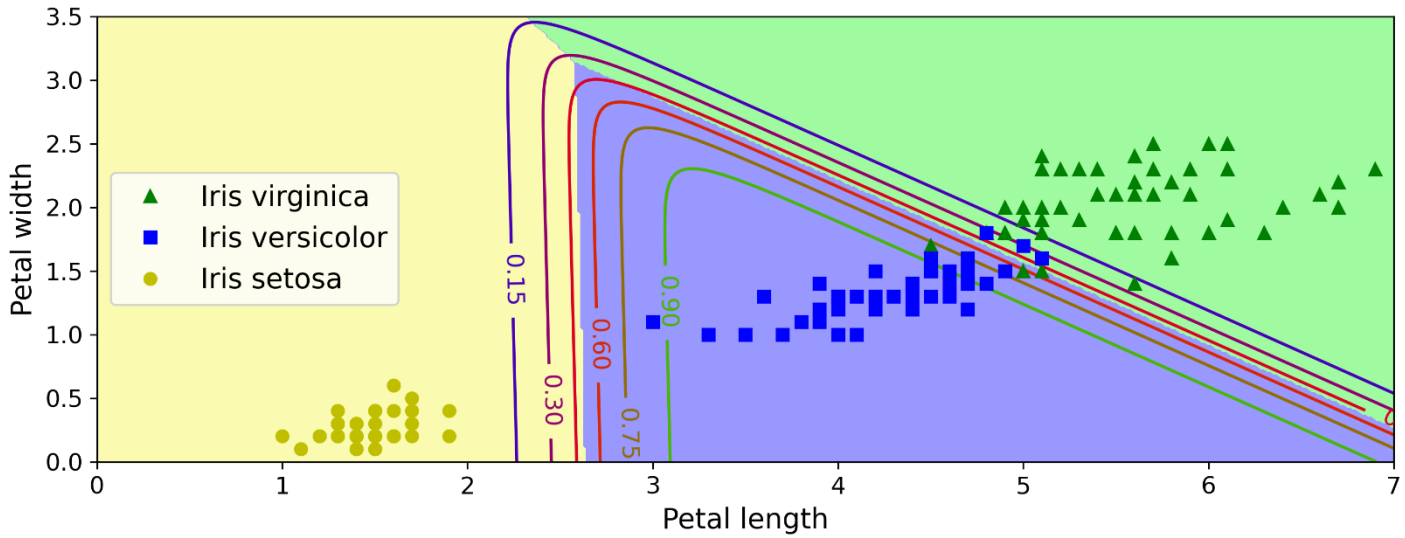
4.6 로지스틱 회귀

```
In [115]:
X = iris["data"][:, (2, 3)]  # 꽃잎 길이, 꽃잎 너비
y = iris["target"]

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

LogisticRegression(C=10, multi_class='multinomial', random_state=42)
```

- multi_class 매개변수를 "multinomial"로 바꾸면 소프트맥스 회귀 사용 가능.
- 기본적으로 하이퍼파라미터 C를 사용하여 조절할 수 있는 l2 규제가 적용된다.

```
In [120]:
print(softmax_reg.predict([[5,2]]))
print(softmax_reg.predict_proba([[5,2]]))

[2]
[[6.38014896e-07 5.74929995e-02 9.42506362e-01]]
```

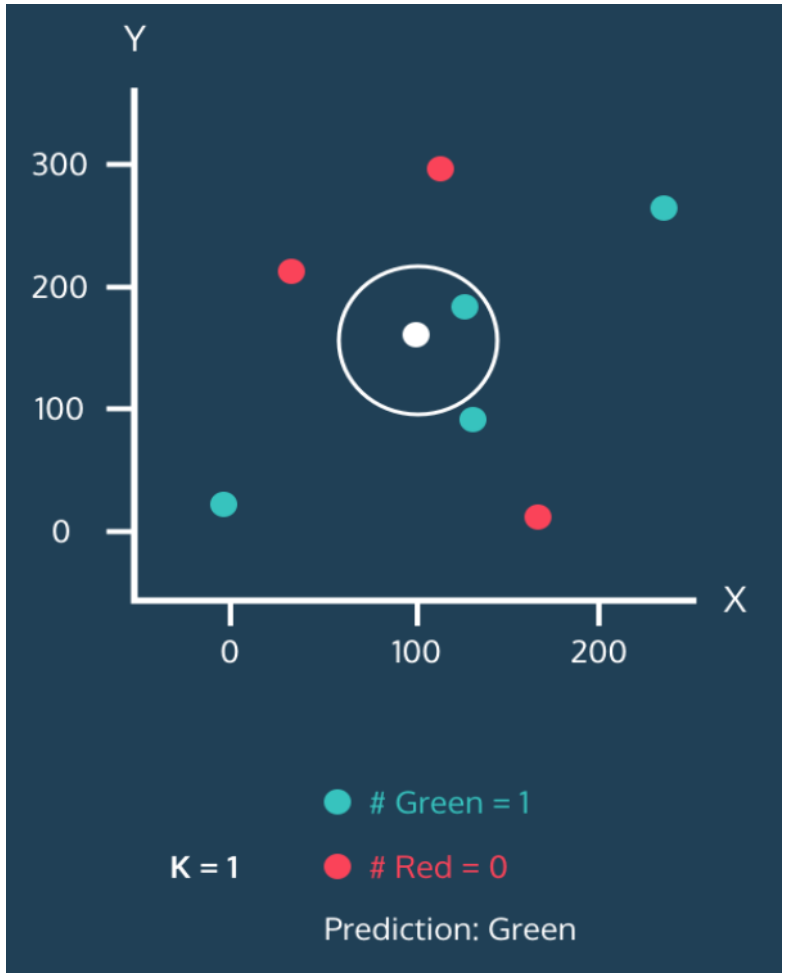
- 꽃잎의 길이가 5cm, 너비가 2cm 꽃을 발견했다고 가정.
- Class2(Iris-Virginica) 예측 확률 94.2%



- 결정 경계를 배경색으로 구분하여 표현.

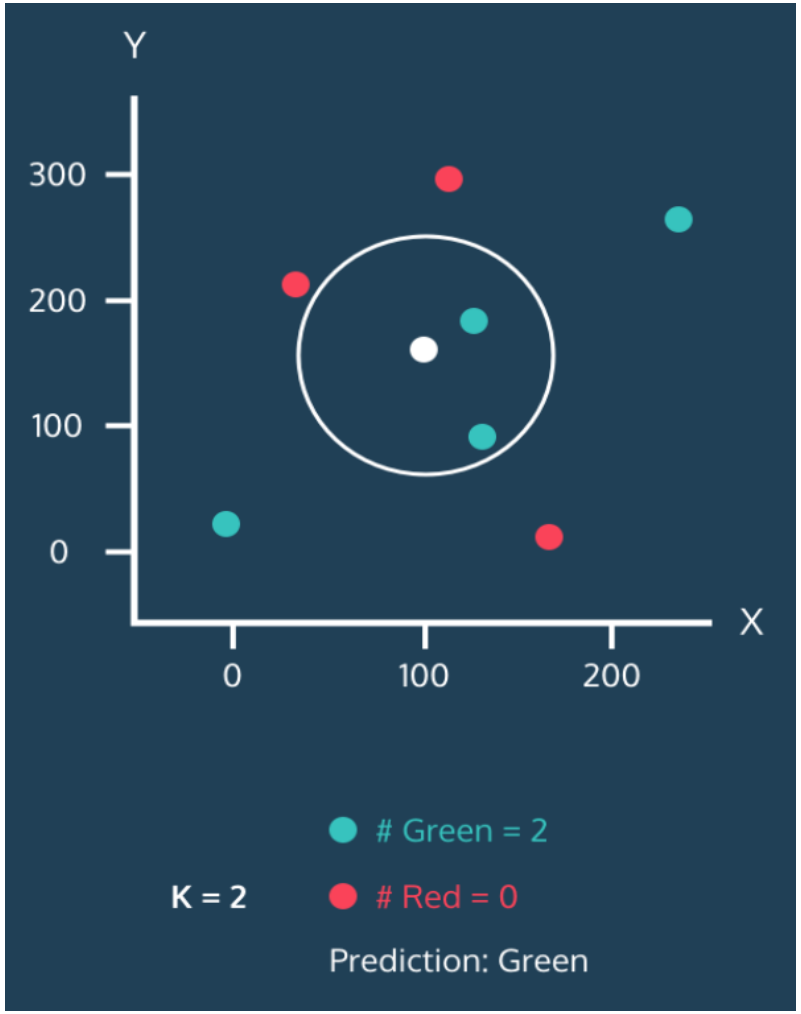
4.7 K-최근접 이웃

- 머신러닝에서 사용되는 분류 알고리즘 중 하나이다.
- 유사한 특성을 가진 데이터는 유사한 범주에 속하는 경향이 있다는 가정.

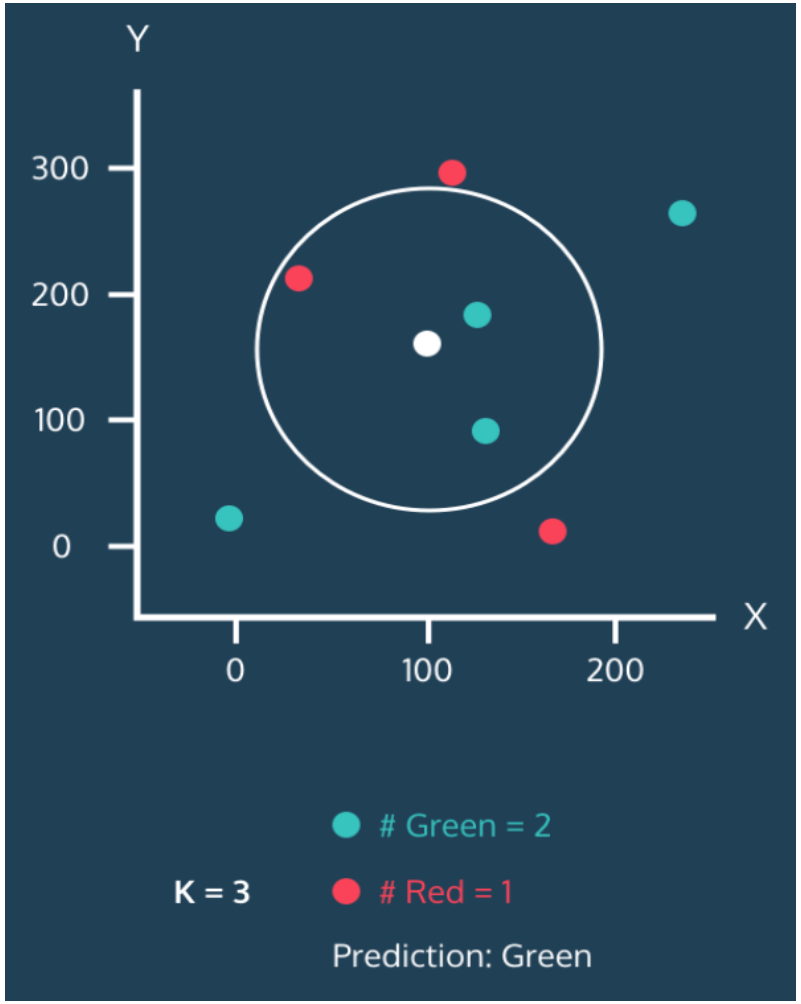


- 하얀 점은 아직 분류가 되지 않은 새로운 데이터이다.
- K-최근접 이웃의 목적은 이렇게 새로운 데이터를 분류하는 것이다.
- K는 원 안에 포함될 가장 가까운 이웃의 개수 라고 한다.
- K=1인 경우, 하얀 점은 초록으로 분류된다.

4.7 K-최근접 이웃

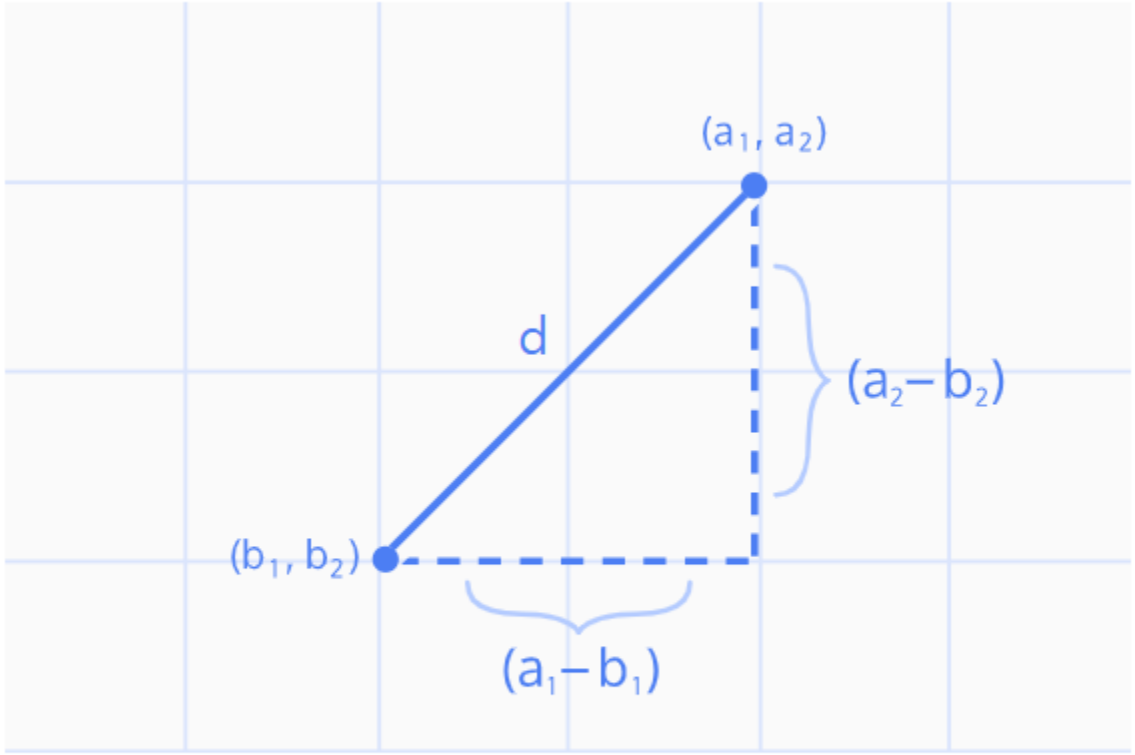


・K = 2인 경우



・K = 3인 경우

4.7 K-최근접 이웃



$$d = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2}$$

· 유클리드 거리

· **거리**는 일종의 유사도 개념을 나타낸다.

▶ 거리가 가까울수록 특성들이 비슷하다는 의미이기 때문이다.

4.7 K-최근접 이웃

1. 정규화

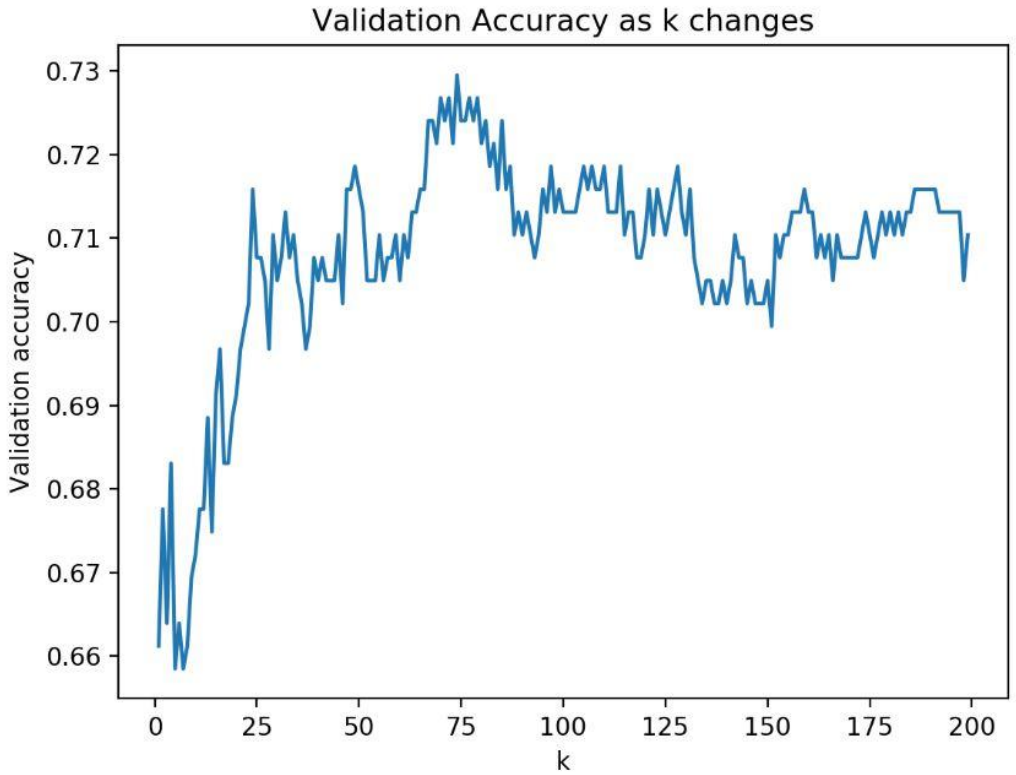
▶ 특성들의 단위가 다른 경우, 터무니 없는 결론이 도출될 수도 있다.

▶ 모든 특성을 고르게 반영하기 위해 정규화를 진행하곤 한다.

2. K개수 선택

▶ K가 너무 **작은 경우** 점 하나에 민감하게 영향을 받는다. (과대 적합↑)

▶ K가 너무 **큰 경우** 점 사이의 거리가 의미가 없어진다. (과소 적합↑)



・학습 곡선을 통해 적절한 K값을 찾는 것도 방법 중 하나이다.

4.7 K-최근접 이웃

```
In [378]: from sklearn.neighbors import KNeighborsClassifier
          classifier = KNeighborsClassifier(n_neighbors = 3)
```

- n_neighbors를 통해 k의 수를 지정한다.

```
In [470]: X = iris["data"]  # 꽃잎 너비
          y = (iris["target"] == 2).astype(np.int)  # Iris virginica이면 1 아니면 0
```

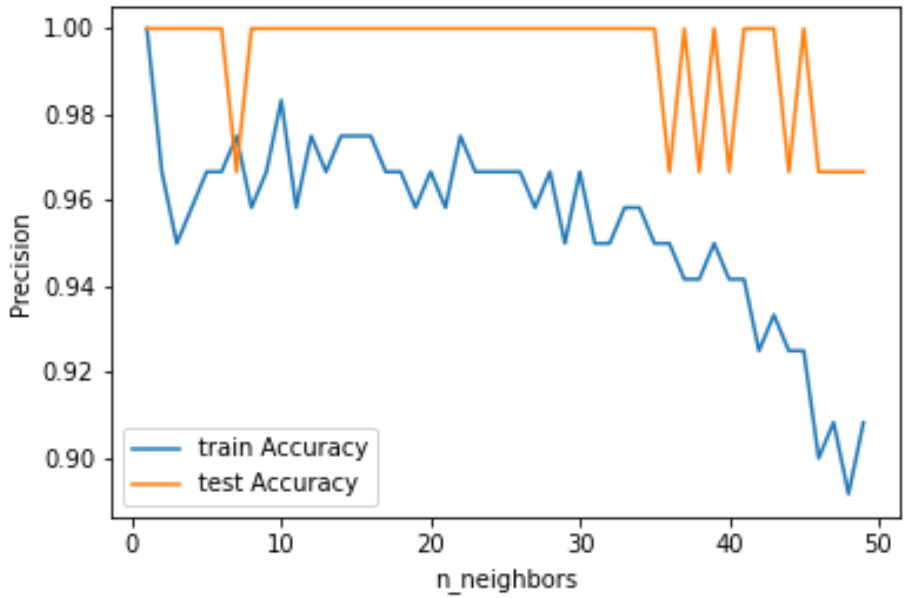
- K-최근접 이웃을 위해 iris 데이터를 사용하였다.

4.7 K-최근접 이웃

```
In [483]: classifier.fit(X_train,y_train)

          KNeighborsClassifier(n_neighbors=3)

In [484]: classifier.predict(X_val)

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

```
In [453]: print(classifier.score(X_train,y_train))
          print(classifier.score(X_val,y_val))

          0.95
          1.0
```

・훈련 데이터의 Accuracy: 95%
・검증 데이터의 Accuracy: 100%




```
In [475]: print(confusion_matrix(y_train,classifier.predict(X_train)))
          print("₩n")
          print(confusion_matrix(y_val,classifier.predict(X_val)))

          [[78  3]
           [ 3 36]]


          [[19  0]
           [ 0 11]]
```

4.8 Naïve Bayes 분류

· **베이즈 정리**는 두 확률 변수의 사전 확률과 사후 확률 사이의 관계를 나타내는 정리이다.

· 즉, 새로운 사건의 확률을 계산하기 전에 이미 일어난 사건을 고려하는 것을 전제로 한다.

· 베이지안 관점에서는 확률을 조건부 확률로 계산을 하게 된다.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

· **"Gachon"**이라는 단어가 메일에 포함되어 있을 때, 그 메일이 Spam일 확률을 구한다고 해보자.

▶"Gachon"이라는 단어는 ham에서 1% 정도 등장한다.

▶"Gachon"이라는 단어는 Spam에서 7% 정도 등장한다.

▶ 전체 메일의 30%는 spam이고, 70%는 ham이다.

$$P(Spam|Gachon\text{이라는 단어가 메일에 포함}) = \frac{P(Gachon\text{이라는 단어가 등장}|Spam)P(Spam)}{P(Gachon\text{이라는 단어가 메일에 포함})} = \frac{0.07 * 0.3}{0.07 * 0.3 + 0.01 * 0.7} = 0.75 = 75\%$$

4.8 Naïve Bayes 분류

· **나이브 베이지 분류기**는 베이지 정리를 활용하여 분류를 수행하는 머신러닝 지도학습 알고리즘이다.

▶ 텍스트(문서)의 분류에 많이 사용된다.

▶ B를 데이터라고 생각하고, A를 레이블이라고 생각하면 일종의 분류기가 된다.

▶ B라는 데이터가 주어졌을 때 A라는 레이블로 분류될 확률을 계산하는 것이기 때문이다.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

· 모든 차원의 개별 독립변수가 서로 조건부 독립이라는 가정을 사용. (**나이브 가정)**

$$P(x_d \mid y = k) = \frac{1}{\sqrt{2\pi\sigma_{d,k}^2}} \exp\left(-\frac{(x_d - \mu_{d,k})^2}{2\sigma_{d,k}^2}\right)$$

· 정규분포 가능도 모형

· GaussianNB()

$$P(x_d \mid y = k) = \mu_{d,k}^{x_d}(1 - \mu_{d,k})^{(1-x_d)}$$

$$P(x_1, \ldots, x_D \mid y = k) = \prod_{d=1}^{D} \mu_{d,k}^{x_d}(1 - \mu_{d,k})^{(1-x_d)}$$

· 베르누이분포 가능도 모형

· BernoulliNB()

$$P(x_1, \ldots, x_D \mid y = k) \propto \prod_{d=1}^{D} \mu_{d,k}^{x_{d,k}}$$

$$\sum_{d=1}^{D} \mu_{d,k} = 1$$

· 다항분포 가능도 모형

· MultinomialNB()

4.8 Naïve Bayes 분류

```
In [436]: X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,random_state=42)

In [437]: Gnb = GaussianNB()
```

・iris 데이터를 이용하여 Naïve Bayes 분류 진행. (**정규분포 가능도 모형 사용**)

```
In [479]: Gnb.fit(X_train, y_train)
          Gnb.predict(X_val)
          print(Gnb.score(X_train,y_train))
          print(Gnb.score(X_val,y_val))

          0.9166666666666666
          0.9333333333333333
```

・훈련 데이터의 Accuracy: 91.6%
・검증 데이터의 Accuracy: 93%

```
In [480]: confusion_matrix(y_train,Gnb.predict(X_train))

          array([[72,  9],
                 [ 1, 38]], dtype=int64)
```

```
In [481]: confusion_matrix(y_val,Gnb.predict(X_val))

          array([[17,  2],
                 [ 0, 11]], dtype=int64)
```