CHAPTER 4
모델 훈련

CONTENTS

학습목표

머신러닝 모델 및 훈련 알고리즘에 대해 블랙박스처럼 취급

이번 장에서는 실제로 어떻게 작동하는지에 대해 학습할 예정

선형회귀

삶의 만족도 = $\theta_0 + \theta_1 * 1$인당_GDP

모델 파라미터

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

$\hat{y}$: 예측값

n : 특성의수

x_n : 특성값

θ_n : 모델파라미터

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

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

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

θ : θ_0부터 θ_n까지의 특성 가중치를 담은 모델의 파라미터 벡터

x : x_0부터 x_n까지 담은 샘플의 특성벡터

$\theta \cdot x$ 는 점곱(dot product) = 벡터의 내적

모델훈련

-모델이 훈련세트에 가장 잘 맞도록 모델 파라미터를 설정하는것

-이를 위해서 먼저 모델이 훈련 데이터에 얼마나 잘 들어맞는지 측정해야함.

-측정지표는 rmse를 사용함.

-그러므로 rmse를 최소화하는 θ를 찾아야 함.

-실제로는 mse를 최소화하는 것이 같은 결과를 내면서 더 간단함.

$$MSE(x, h_\theta) = \frac{1}{m}\sum_{\bar{I}=1}^{m}\left(\overbrace{\theta^T x^{(i)}}^{\hat{y}} - y^{(i)}\right)^2$$

정규방정식

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

$\hat{\theta}$은 비용 함수를 최소화 하는 θ 입니다.

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

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

In [4]:
```
X_b = np.c_[np.ones((100, 1)), X]
```


```
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

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

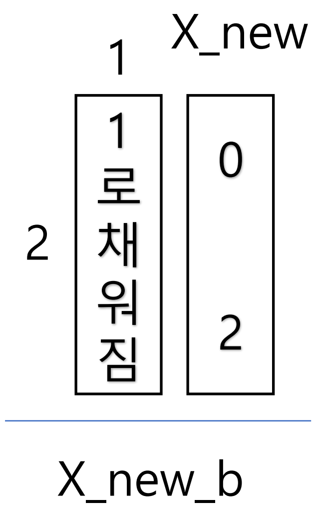
In [5]:
```
theta_best
```

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

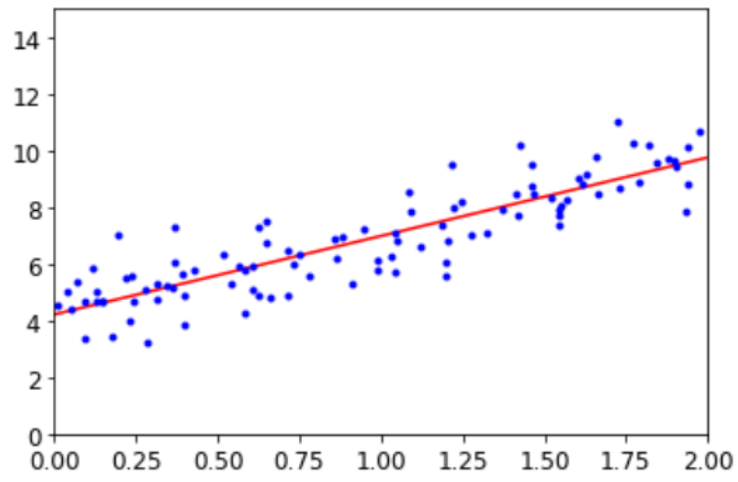
$$\hat{y} = \mathbf{X}\hat{\boldsymbol{\theta}}$$

In [6]:
```
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[6]:
```
array([[4.21509616],
       [9.75532293]])
```



In [7]:
```
plt.plot(X_new, y_predict, "r-")
plt.plot(X, y, "b.")
plt.axis([0, 2, 0, 15])
plt.show()
```



In [9]:
```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(X, y)
lin_reg.intercept_, lin_reg.coef_
```
가중치와 편향이 저장됨

Out[9]:
```
(array([4.21509616]), array([[2.77011339]]))
```

$$\hat{\theta} = x^{+}y$$

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

x^{+}는 x의 유사역행렬

계산 복잡도

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

(n+1) x (n+1)

$(x^T x)^{-1}$

O(n^2.4)~O(n^3)

특성의 수가 두배로 늘어나면 계산 시간이 대략

O(2^2.4)=5.3 ~ O(2^3)=8배 로 증가함

SVD

O(n^2)

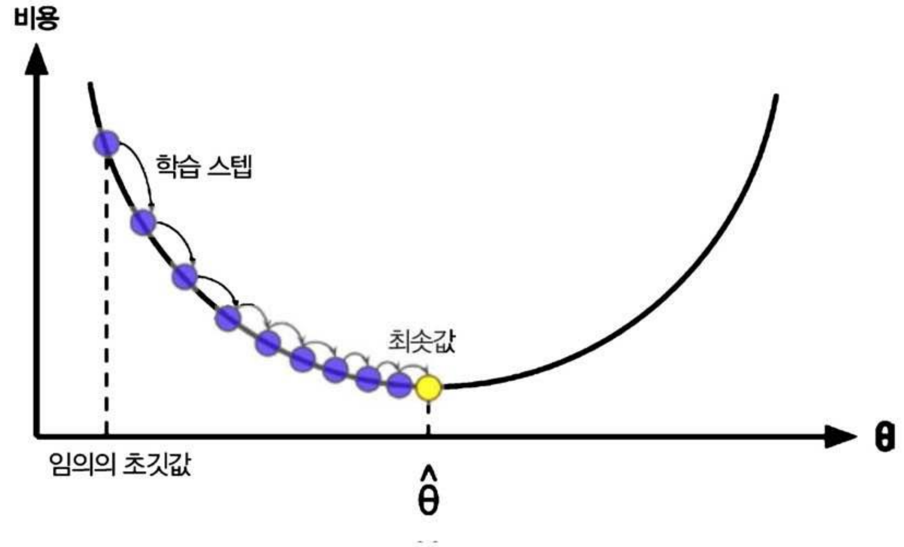
특성의 수가 두배로 늘어나면 2배 늘어남

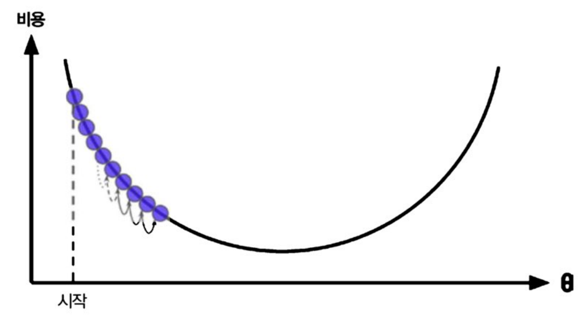
경사하강법

여러 종류의 문제에서 최적의 해법을 찾을 수 있는 일반적인 최적화 알고리즘

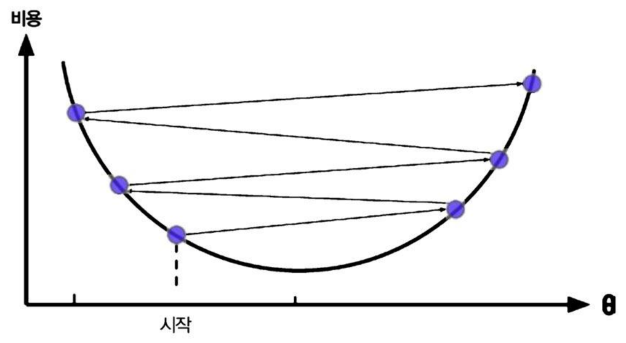
비용 함수를 최소화하기 위해 반복해서 파라미터를 조정한다.

θ(파라미터벡터)에 대해
비용함수의 기울기를 계산

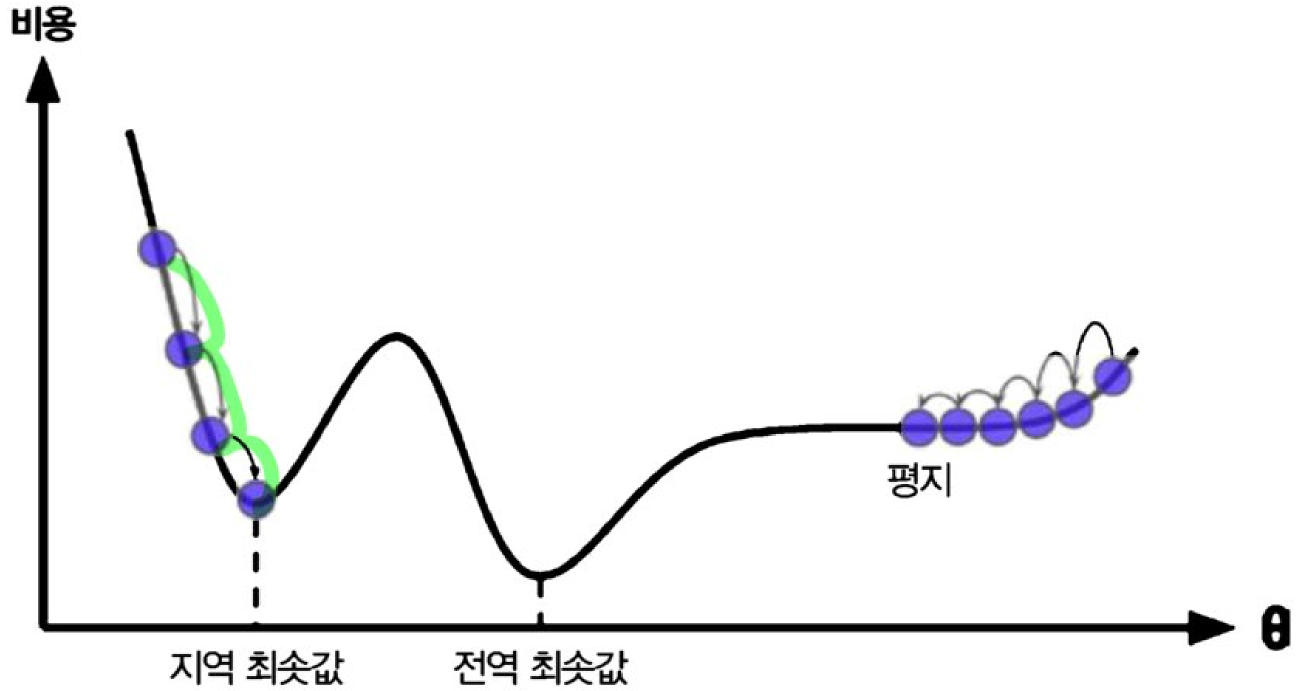




학습률이 너무 작을때

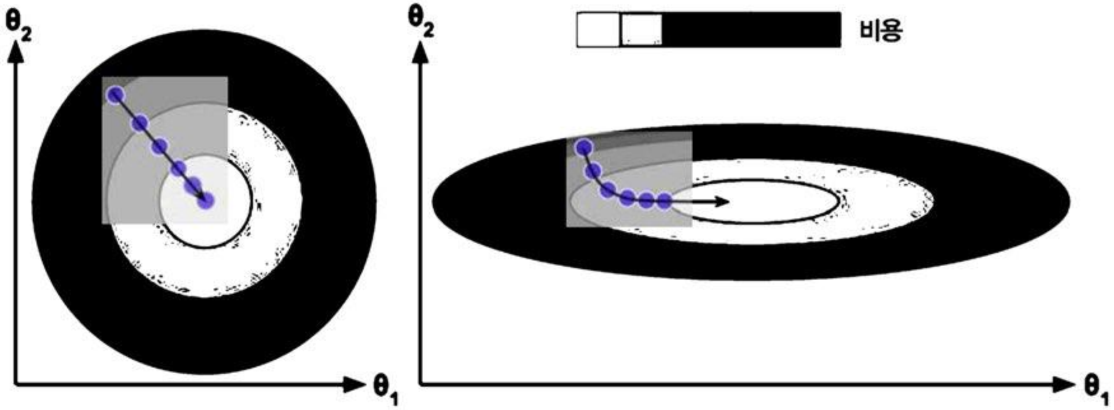


학습률이 너무 클때

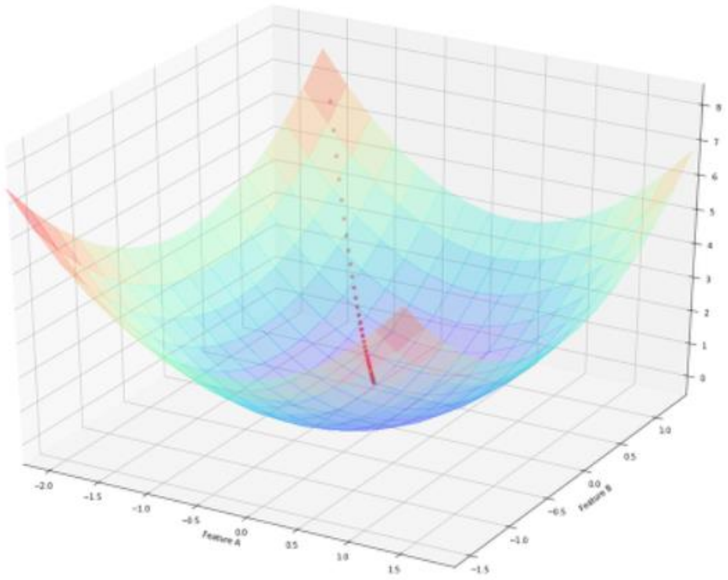


왼쪽
전역최솟값 보다 덜 좋은 지역 최솟값에 수렴 한다

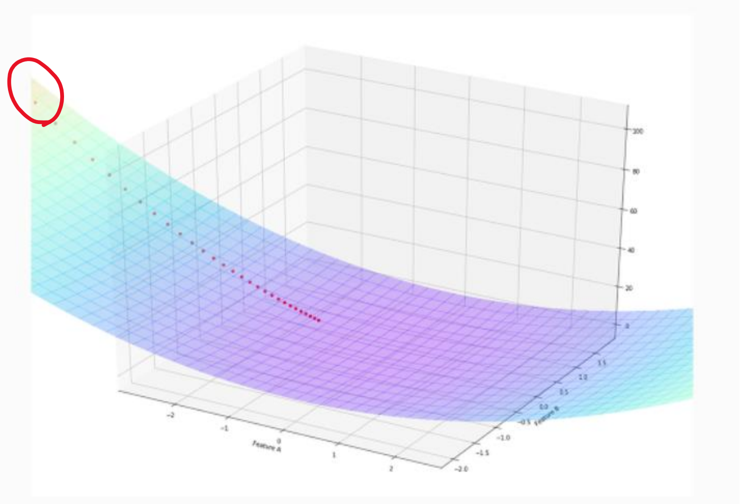
오른쪽
평탄한 지역을 지나기 위해 시간이 오래 걸리고
일찍 멈추게 되어 전역 최솟값에 도달하지 못한다.



θ_2
θ_1
θ_2
비용
θ_1



Feature A
Feature B



Feature A
Feature B

배치 경사 하강법

θ_j에 대해 비용함수의 경사도를 계산 해야함

= θ_j가 미세하게 변경될때 비용함수에 미치는 영향을 알아야한다.

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

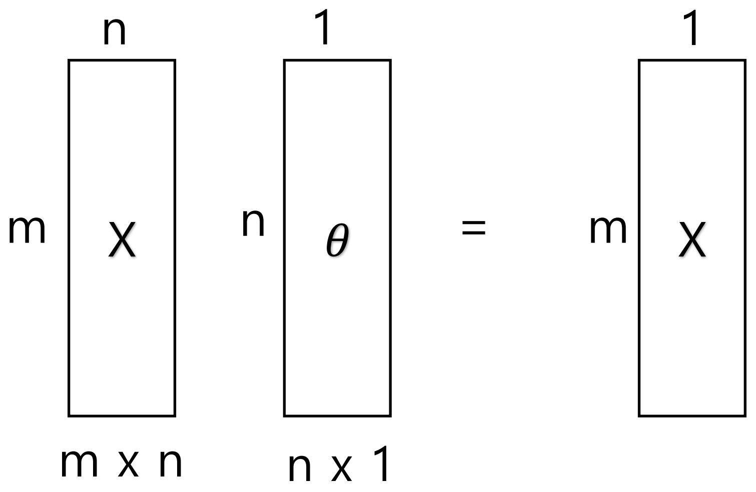
$$\frac{\partial}{\partial\theta_j} MSE(\theta) = \frac{2}{m}\sum_{i=1}^{m}\left(\theta^T x^{(i)} - y^{(i)}\right) x_j^{(i)}$$

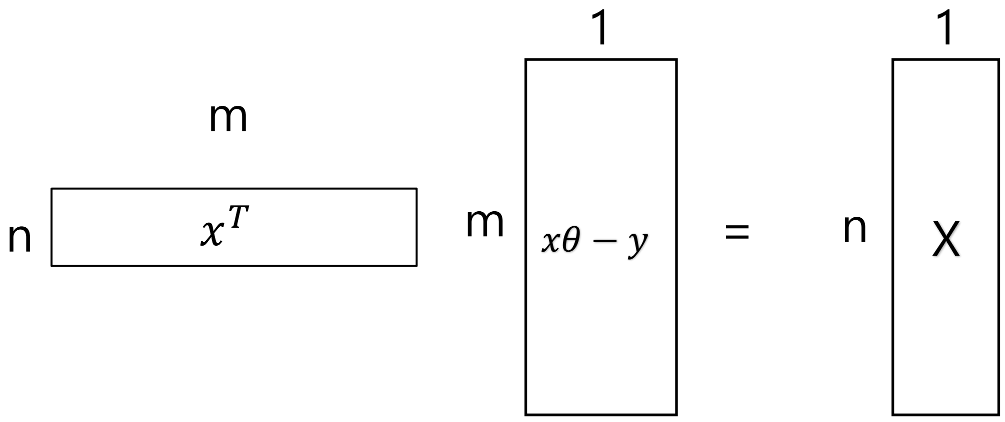
θ_j 에 대해편미분

편도함수를 한꺼번에 계산하기

$$\frac{\partial}{\partial \boldsymbol{\theta}} \mathrm{MSE}(\boldsymbol{\theta}) = \frac{2}{m} \mathbf{X}^T (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

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





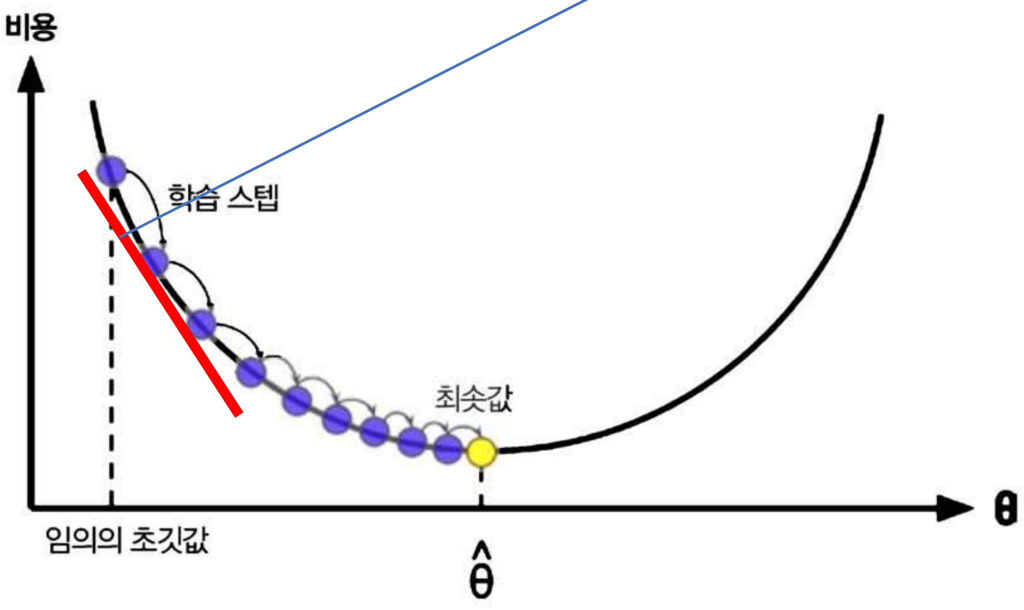
크기

$$\boldsymbol{\theta}^{(\text{next step})} = \boldsymbol{\theta} - \eta \frac{\partial}{\partial \boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta})$$

방향

학습률 = 내려가는 스텝의 크기가 됩니다.
그래디언트 백터 = 가야할 방향

gradient = 음의 값

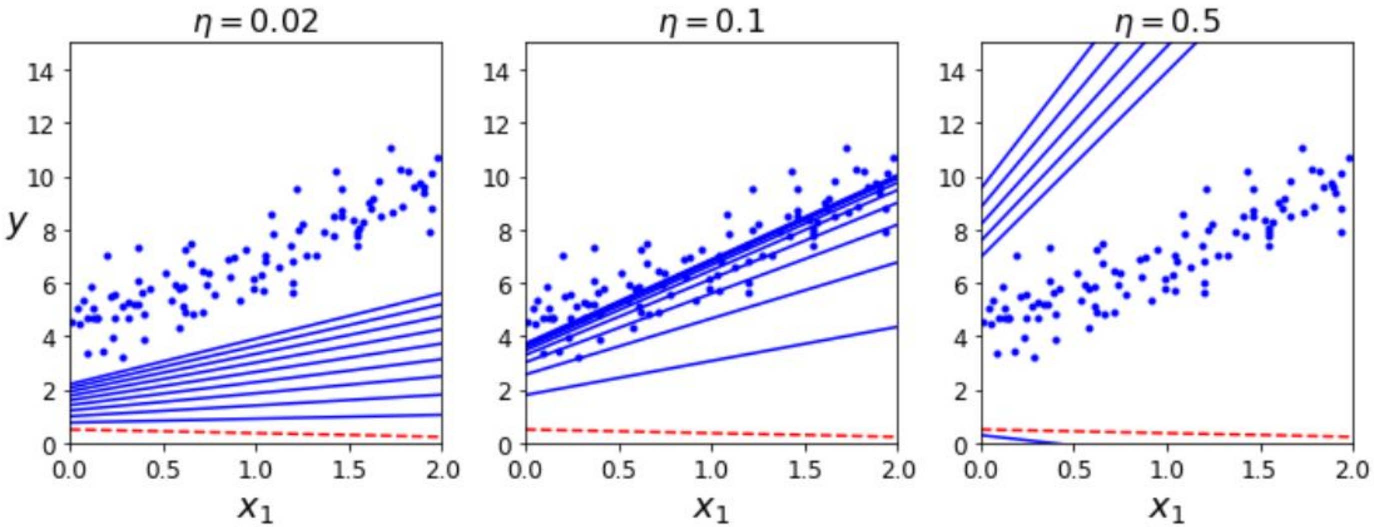


$$\boldsymbol{\theta}^{(\text{next step})} = \boldsymbol{\theta} - \eta \frac{\partial}{\partial \boldsymbol{\theta}} \mathrm{MSE}(\boldsymbol{\theta})$$

In [13]:
```python
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
```

$$\frac{\partial}{\partial \boldsymbol{\theta}} \mathrm{MSE}(\boldsymbol{\theta}) = \frac{2}{m}\mathbf{X}^T(\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

$$\boldsymbol{\theta}^{(\text{next step})} = \boldsymbol{\theta} - \eta \frac{\partial}{\partial \boldsymbol{\theta}} \mathrm{MSE}(\boldsymbol{\theta})$$



적절한 학습률

반복횟수를 ↑

Tolerance > gradient_vector => 알고리즘 중지

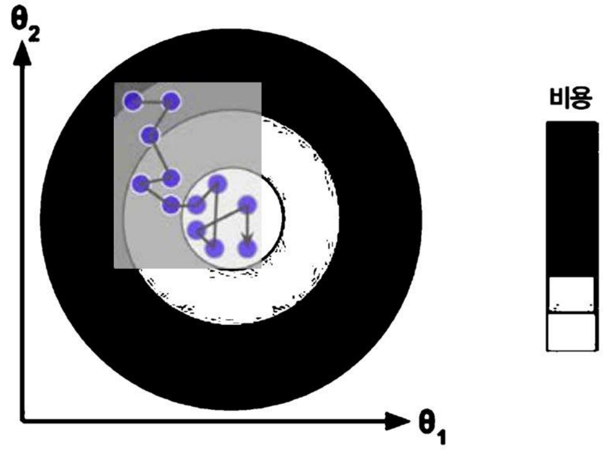
확률적 경사하강법

경사 하강법 스텝에서 전체 훈련세트 X에 대해 계산한다. 즉 매 스텝에서 훈련 데이터 전체를 사용한다.

⇒ 매우 큰 훈련 세트에서는 매우느림.

-매 스텝에서 한 개의 샘플을 무작위로 선택

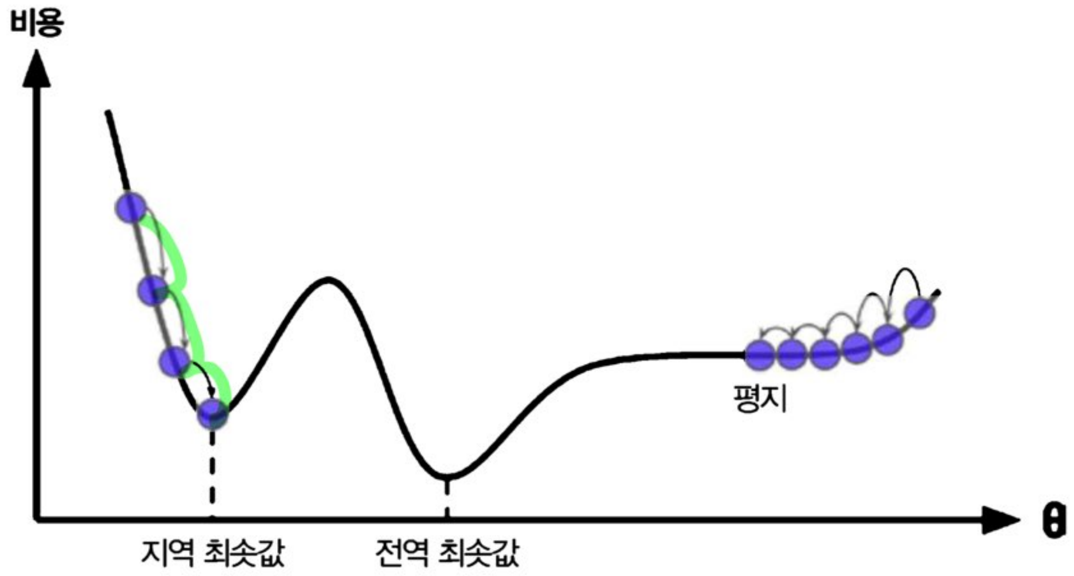
-샘플에 대한 그레디언트를 계산



장점
-훨씬 개선된 속도
-매 반복에서 하나의 셈플만 메모리에 있으면 매우 큰 훈련 세트도 훈련시킬 수 있음

단점

-비용함수가 최솟값에 수렴할 때까지 위아래로 요동치며 평균적으로 감소함

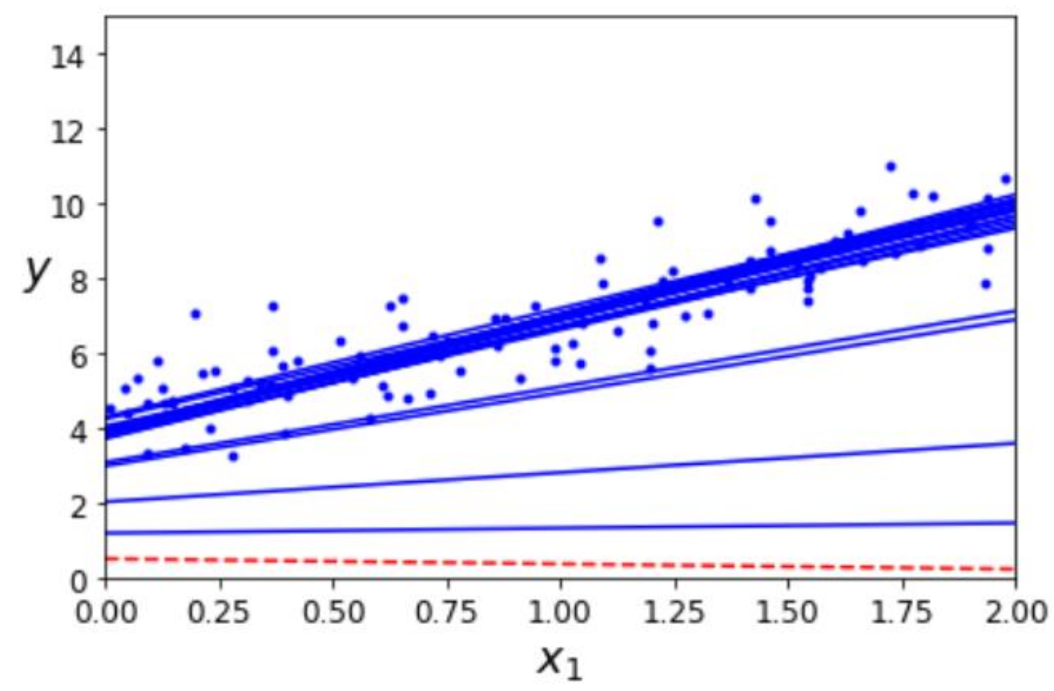


비용 함수가 매우 불규칙할때 지역 최솟값을
건너 뛸 수 있음
-> 단 전역 최솟값에 다다르지 못할 수 있음

how? 학습률을 점진적으로 감소시킴

학습스케쥴링

```python
import numpy as np
import matplotlib.pyplot as plt
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
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)
X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2, 1)), X_new]
y_predict = X_new_b.dot(theta_best)

theta_path_sgd = []
m =100   #100
np.random.seed(42)

n_epochs = 50
t0, t1 = 5, 50  # 학습 스케쥴 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:                    # 훈련스텝의 첫 20개를 보여준다
            y_predict = X_new_b.dot(theta)           # 예측값구하기
            style = "b-" if i > 0 else "r--"         # 각 에폭의 첫번째 값은 빨간점선으로 표시하구 나머지는 파란색 실선으로 표현
            plt.plot(X_new, y_predict, style)        # 그래프생성
        random_index = np.random.randint(m)          # 0부터 99까지 중 랜덤으로 숫자 생성(한개의 셈플에 대해 무작위로 선택하기 위함)
        xi = X_b[random_index:random_index+1]        # 랜덤으로 무작위 샘플을 선택함(위에서 생성한 랜덤변수인 random_index를 사용한다.)
        yi = y[random_index:random_index+1]          # 랜덤으로 무작위 샘플을 선택함(위에서 생성한 랜덤변수인 random_index를 사용한다.)
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi) # 배치경사하강법
        eta = learning_schedule(epoch * m + i)       # 학습스케쥴
        theta = theta - eta * gradients              # 다음스텝을 위한 수정
        theta_path_sgd.append(theta)

plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([0, 2, 0, 15])
plt.show()
```



실제로 학습률이 어떻게 감소하고 있는지 궁금해졌다.

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

for epoch in range(n_epochs):
    for i in range(m):
      eta = learning_schedule(epoch*m+i)
      print(eta)
    print('\n')
```

epoch=0

0.03333333333333333   0.020080321285140562

epoch=1

0.02   0.014326647564469915

사이킷런에서 SGD 방식으로 선형 회귀 사용법

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]))
```

미니배치 경사 하강법

미니배치?
임의의 작은 샘플 세트
=>이것에 대해 경사를 계산
=>행렬에 대한 병렬 연산을 진행하기 때문에 gpu연산에 특화

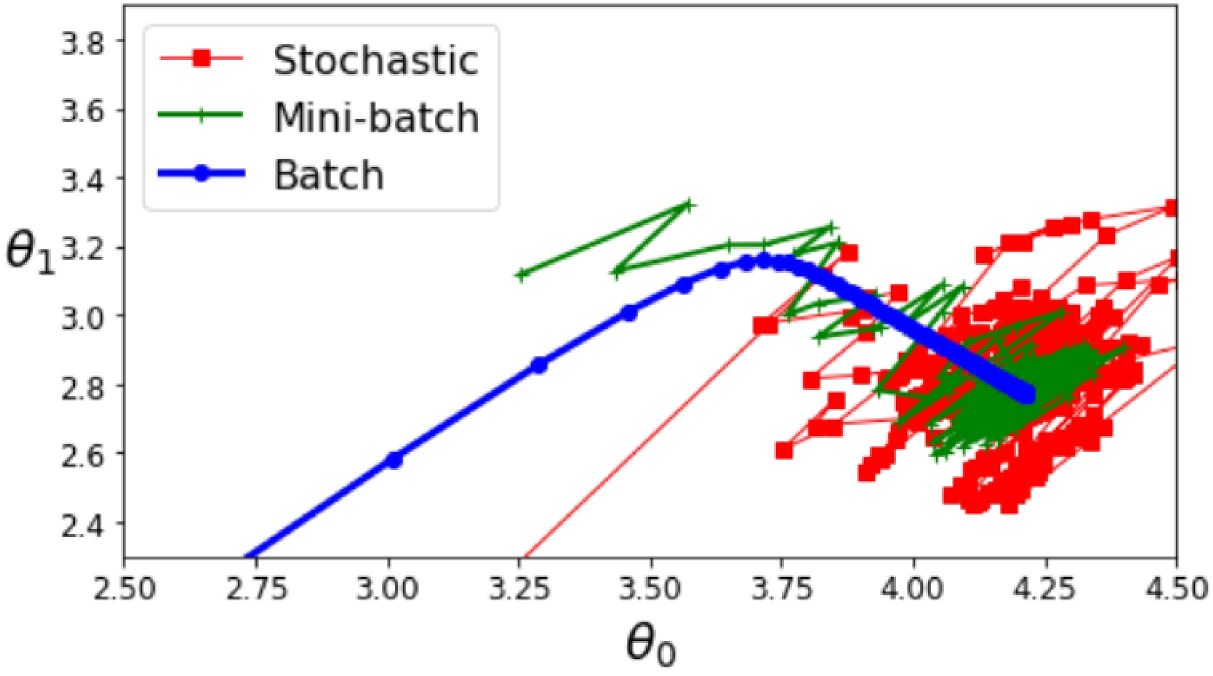


표 4-1 선형 회귀를 사용한 알고리즘 비교[20]

알고리즘	m이 클 때	외부 메모리 학습 지원	n이 클 때	하이퍼 파라미터 수	스케일 조정 필요	사이킷런
정규방정식	빠름	No	느림	0	No	`N/A`
SVD	빠름	No	느림	0	No	`LinearRegression`
배치 경사 하강법	느림	No	빠름	2	Yes	`SGDRegressor`
확률적 경사 하강법	빠름	Yes	빠름	≥2	Yes	`SGDRegressor`
미니배치 경사 하강법	빠름	Yes	빠름	≥2	Yes	`SGDRegressor`

다항회귀

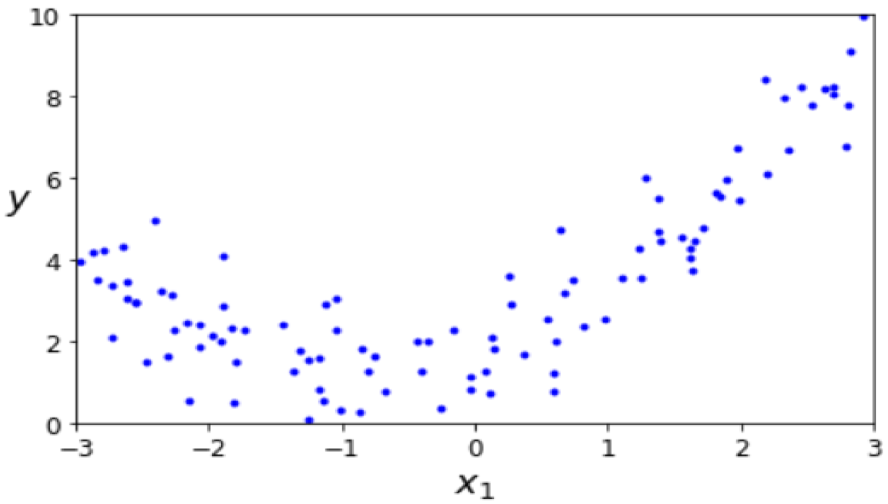
예측하기 원하는 데이터들의 관계가 선형적이지 않다면 선형회귀를 사용해서 예측값을 구할 수 있을까?

In [27]:
```
import numpy as np
import numpy.random as rnd

np.random.seed(42)
```

In [28]:
```
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```

In [29]:
```
plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([-3, 3, 0, 10])
save_fig("quadratic_data_plot")
plt.show()
```

그림 저장: quadratic_data_plot



```
In [30]: from sklearn.preprocessing import PolynomialFeatures
         poly_features = PolynomialFeatures(degree=2, include_bias=False)
         X_poly = poly_features.fit_transform(X)
         X[0]

Out[30]: array([-0.75275929])
```

degree : 방정식의 차수
include_bias : True ->상수항 포함, False-상수항 제거

```
In [31]: X_poly[0]

Out[31]: array([-0.75275929,  0.56664654])
```

원래 특성,　원래 특성의 제곱

```
In [32]: lin_reg = LinearRegression()
         lin_reg.fit(X_poly, y)
         lin_reg.intercept_, lin_reg.coef_

Out[32]: (array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

1차항　2차항

실제함수 : $y = 0.5x_1^2 + 1.0x_1 \ + 2.0 + noise$

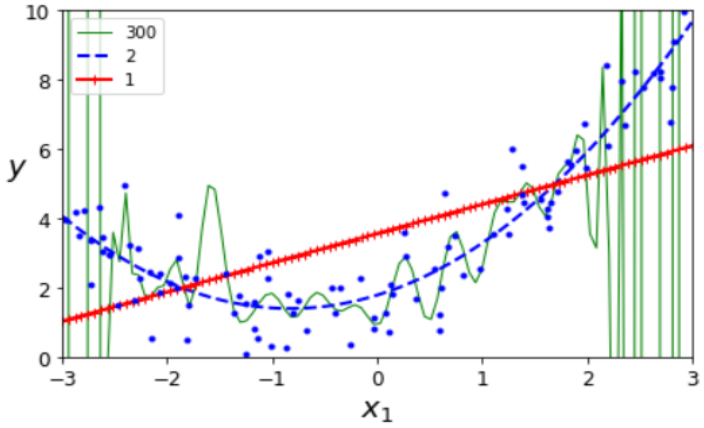
예측모델 : $y = 0.56x_1^2 + 0.93x_1 \ + 1.78$

학습곡선

300차항 회귀 모델

In [34]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

for style, width, degree in (("g-", 1, 300), ("b--", 2, 2), ("r-+", 2, 1)):
    polybig_features = PolynomialFeatures(degree=degree, include_bias=False)
    std_scaler = StandardScaler()
    lin_reg = LinearRegression()
    polynomial_regression = Pipeline([
            ("poly_features", polybig_features),
            ("std_scaler", std_scaler),
            ("lin_reg", lin_reg),
        ])
    polynomial_regression.fit(X, y)
    y_newbig = polynomial_regression.predict(X_new)
    plt.plot(X_new, y_newbig, style, label=str(degree), linewidth=width)

plt.plot(X, y, "b.", linewidth=3)
plt.legend(loc="upper left")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([-3, 3, 0, 10])
save_fig("high_degree_polynomials_plot")
plt.show()
```

그림 저장: high_degree_polynomials_plot



300차항 모델 = 과대적합
선형 모델 = 과소적합

모델의 과대적합과 과소적합의 판단

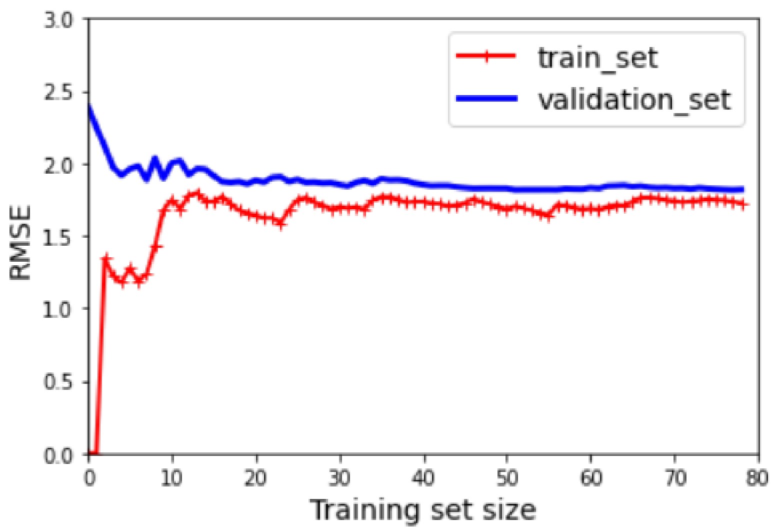
1. 교차검증 사용
   -훈련데이터에서 성능이 좋음, 교차검증점수가 나쁨
   =>과대적합

   -훈련데이터에서 성능 나쁨, 교차검증점수 나쁨
   =>과소적합

2. 학습곡선

-훈련세트와 검증세트의 모델성능을 훈련 세트 크기의
함수로 나타냄

```python
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])              #데이터 fitting
        y_train_predict = model.predict(X_train[:m])    #train set에 대한 예측
        y_val_predict = model.predict(X_val)            #validation set에 대한 예측
        train_errors.append(mean_squared_error(y_train[:m], y_train_predict)) #y_train과 y_train_predict에 대한 MSE
        val_errors.append(mean_squared_error(y_val, y_val_predict))          #y_val과 y_val_predict에 대한 MSE

    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)
    plt.xlabel("Training set size", fontsize=14)
    plt.ylabel("RMSE", fontsize=14)
```



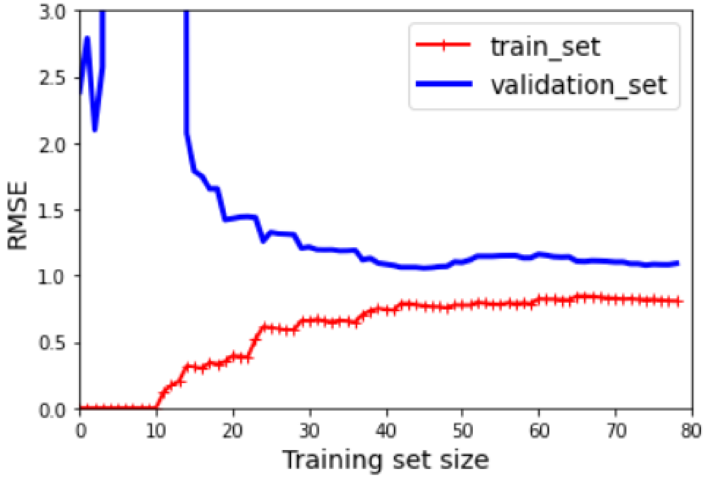
train set
-train set에 샘플이 하나 혹은 두개 있을 때 모델이 완벽하게 작동
-비선형->완벽한 데이터학습 불가
-샘플이 추가되어도 평균 오차 크게 변동 없음

validation set
-validation에 샘플이 적으면 일반화 될 수 없어서 초기에 매우 오차가 큼.
-샘플이 추가되면서 오차 감소
-train set의 그래프와 가까워짐.

같은 데이터로 10차항 다항 회귀 모델 곡선 생성

In [79]:

```python
from sklearn.pipeline import Pipeline

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

plot_learning_curves(polynomial_regression, X, y)
plt.axis([0, 80, 0, 3])
plt.show()
```



-선형 회귀 모델보다훨씬 낮음

-두 곡선 사이에 공간이 존재
=>train set의 모델 성능이 validation set에서 보다 훨씬 낫다. 이는 과대적합 모델의 특징

-더 큰 훈련 세트 사용시 두 곡선이 점점 가까워짐

편향 분산 Trade-off

편향에 따른 오차 : 우리가 주로 말하는 y절편
-편향으로 인한 오차는 기대예측과 실제 값 사이의 차이로써 얻어짐
=모델들의 예측이 올바른 값으로부터 얼마나 떨어져 있는지를 측정

-> 훈련데이터에 과소적합 되기 쉽다

분산에 따른 오차 : 데이터에 대한 모델예측의 다양성
-전체 모델링 과정의 반복 속에서 나타나는 예측값들 사이에서 예측이 얼마나 다양한지

-자유도가 높은모델

->훈련데이터에 과대적합 되기 쉽다.

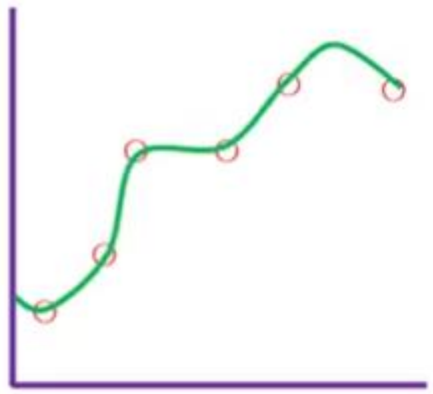
규제가 있는 선형 모델 – 릿지 회귀

-선형 회귀 모델에서는 보통 모델의 가중치를 제한함

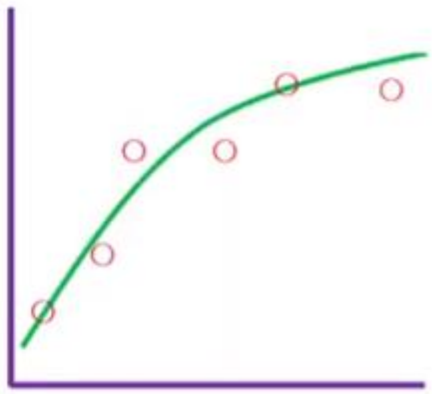
-일반적으로 영향을 거의 미치지 않는 특성에 대하여 0에 가까운 가중치를 줌

$$J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2 = \frac{1}{2} (\|w\|_2)^2$$

규제가 있는 선형 모델 – 릿지 회귀



$\beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \beta_4 x^4$



$\beta_0 + \beta_1 x + \beta_2 x^2$

$$\min_{\beta} \sum_{i=1} (y_i - \hat{y}_i)^2 + 5000{\beta_3}^2 + 5000{\beta_4}^2$$

전체식을 최소화 시키기 위해서는 베타3과 베타4는
무조건적으로 최소화 되어야한다.(0에 근사해짐)
즉 5000은 penalty이다.

$$\min_{\beta} \sum_{i=1} (y_i - \hat{y}_i)^2 + 5000{\beta_3}^2 + 5000{\beta_4}^2$$

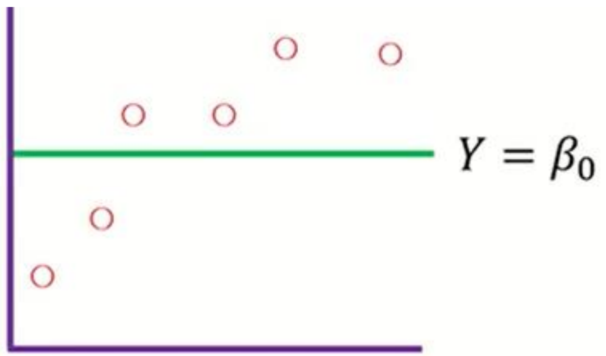
$$L(\beta) = \min_{\beta} \sum_{i=1} (y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

(1)내가 가지고 있는 데이터의 오차만을 최소화 시키겠다.

(2)현재데이터에 대한 정확도도 중요하지만 예측데이터를 위한 제약

$$L(\beta) = \min_{\beta} \sum_{i=1} (y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

λ = 규제의 정도
if λ= ↑, 모든 가중치가 거의 0에 가까워짐.
=>underfitting



$\beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \beta_4 x^4$

λ = 규제의 정도
if λ=0, bias는 ↓, variance ↑
=>overfitting

$\beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \beta_4 x^4$

In [93]:

```python
#릿지회귀
ridge_reg = Ridge(alpha=1, solver="cholesky", random_state=42)
ridge_reg.fit(X, y)
ridge_reg.predict([[1.5]])
```

Out[93]: array([[1.55071465]])

In [94]:

```python
#확률적 경사하강법
sgd_reg = SGDRegressor(penalty="l2", max_iter=1000, tol=1e-3, random_state=42) #penalty 매개변수는 사용할 규제를 지정, l2는 sgd
sgd_reg.fit(X, y.ravel())
sgd_reg.predict([[1.5]])
```

Out[94]: array([1.47012588])

규제가 있는 선형 모델 – 라쏘 회귀

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

–릿지회귀와 비슷하지만 L2노름의 제곱을 2로 나눈 것
–대신 가중치 벡터의 L1 노름을 사용함(절대값의 합에 대한 제한)

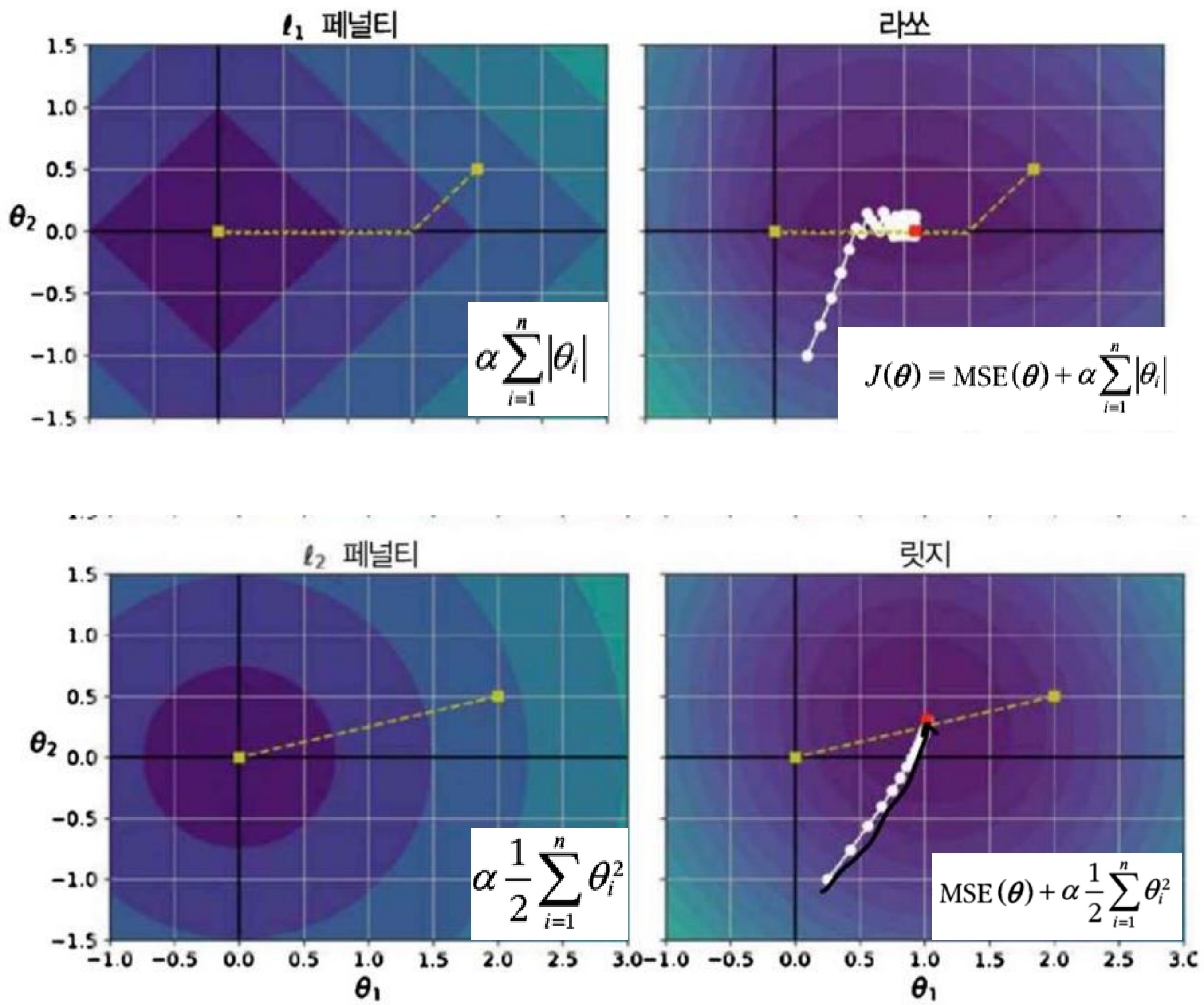
규제가 있는 선형 모델 – 라쏘 회귀

In [96]:

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

Out[96]:

```
array([1.53788174])
```



ℓ_1 페널티
라쏘
$\alpha \sum_{i=1}^{n} |\theta_i|$
$J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + \alpha \sum_{i=1}^{n} |\theta_i|$
ℓ_2 페널티
릿지
$\alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$
$\mathrm{MSE}(\boldsymbol{\theta}) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$
θ_1
θ_2

규제가 있는 선형 모델 – 엘라스틱넷

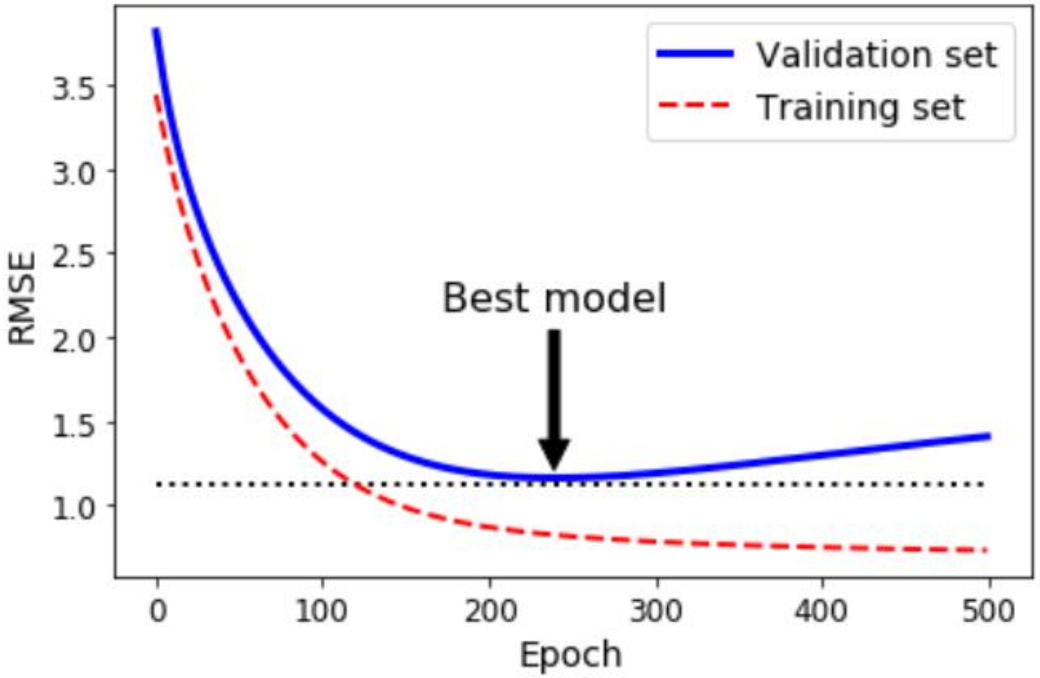
-릿지 회귀와 라쏘 회귀의 절충안

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

if r=0, 릿지회귀
if r=1, 라쏘회귀

규제가 있는 선형 모델 – 조기종료

-검증 에러가 최솟값에 도달하면 바로 훈련을 중지

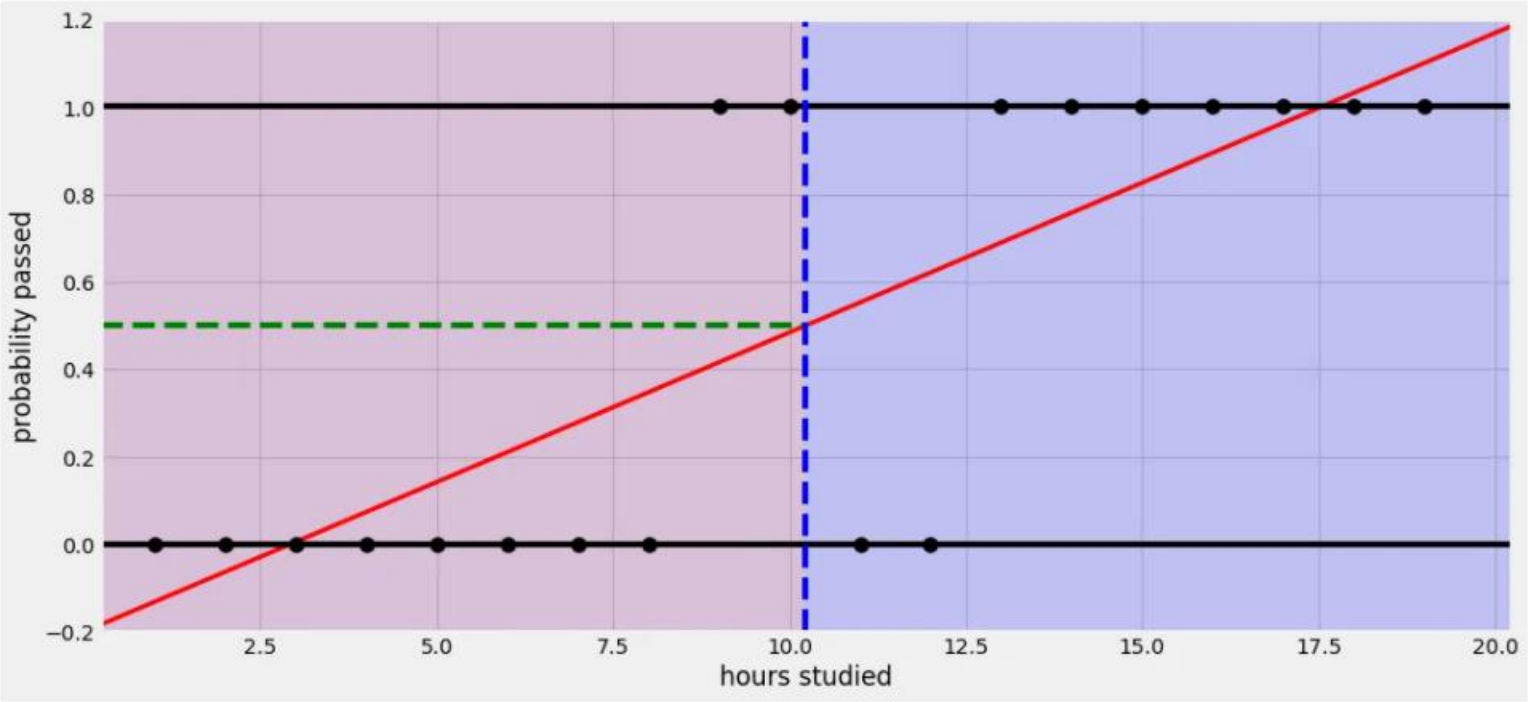


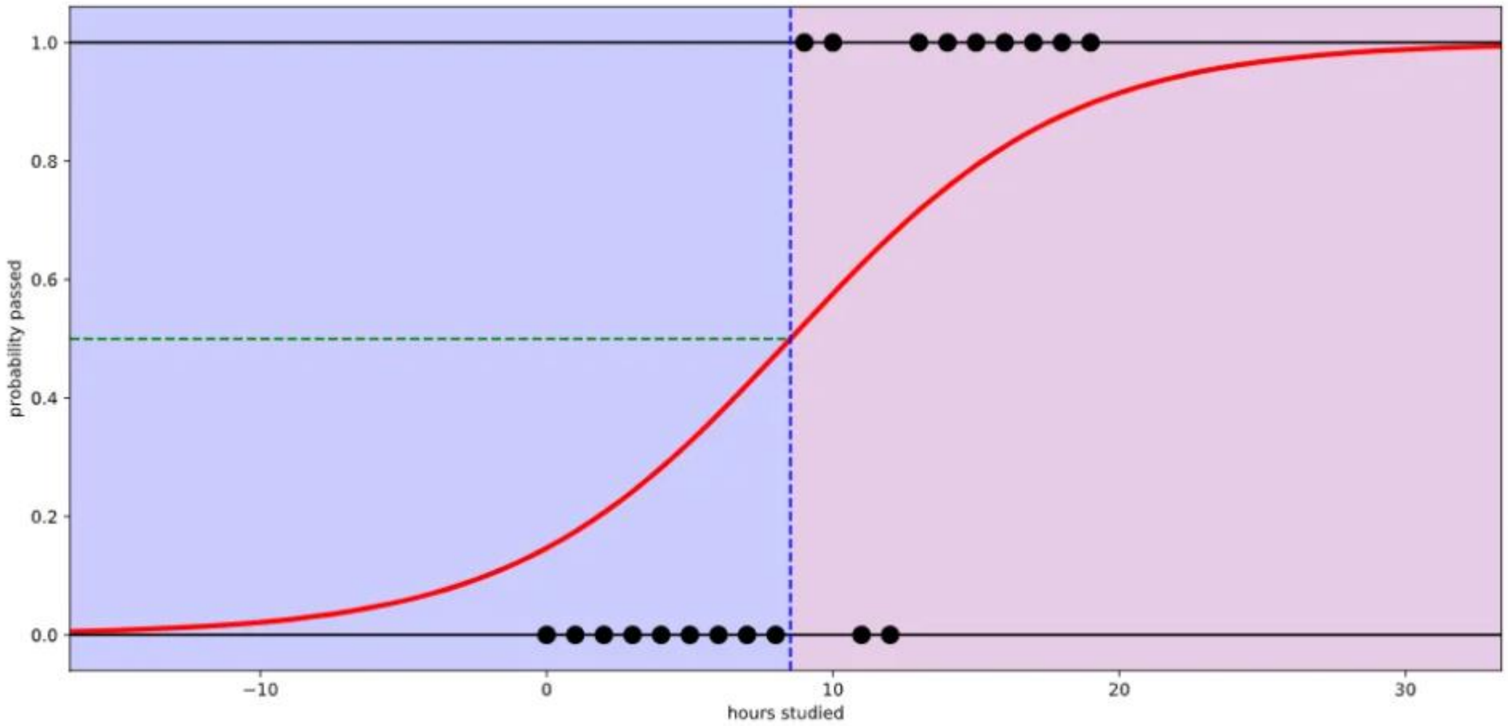
```python
from copy import deepcopy

#90차항 다항회귀모델 및 스케일 조정
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 모델생성
sgd_reg = SGDRegressor(max_iter=1, tol=-np.infty, warm_start=True,
                       penalty=None, learning_rate="constant", eta0=0.0005, random_state=42) #warm_start = True : 이전에 업데이
                                                                                              #eta0 : 학습률


#초기값설정
minimum_val_error = float("inf") #최소에러값을 무한대로 초기화
best_epoch = None
best_model = None
for epoch in range(1000):
    sgd_reg.fit(X_train_poly_scaled, y_train)  #모델학습
    y_val_predict = sgd_reg.predict(X_val_poly_scaled) #validation_set에 대한 예상
    val_error = mean_squared_error(y_val, y_val_predict)#validation set에 대한 MSE
    if val_error < minimum_val_error:                   #만약 val_error가 최소에러값 보다 작다면 val_error값이 최소에러값이 된다
        minimum_val_error = val_error
        best_epoch = epoch                              #그 순간의 에폭값 저장
        best_model = deepcopy(sgd_reg)                  #그 순간의 SGD모델을 저장
```

로지스틱 회귀

회귀를 사용하여 데이터가 어떤 범주에 속할 확률을 0에서 1사이 값으로 예측

확률에 따른 2진분류




1.0
0.8
0.6
0.4
0.2
0.0
probability passed
−10
0
10
20
30
hours studied

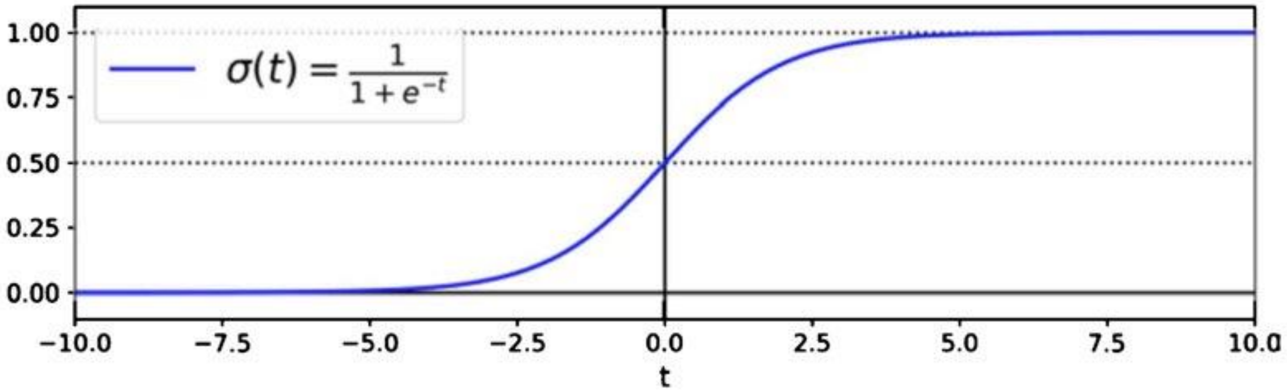
샘플 x가 양성클래스에 속할 확률 $\hat{p} = h_\theta(x) = \sigma(\theta^T x)$

$$\hat{y} = \begin{cases} 0 & \hat{p} < 0.5\text{일 때} \\ 1 & \hat{p} \geq 0.5\text{일 때} \end{cases}$$

추정치

식 4-14 로지스틱 함수

$$\sigma(t) = \frac{1}{1+\exp(-t)}$$



로지스틱 회귀의 비용함수

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

$c(\theta) = -\log(1 - \hat{p})$ y=0(음성샘플)일때

가능도함수와 최대가능도 추정

가정 : 제비뽑기 기계에서 당첨이 나올 확률은 일정이며 독립시행이다.
ex)다섯 번 제비를 뽑았을 때, 첫번째와 네번째에 당첨이 나오고 나머지 세번은 당첨되지 않는다.
제비를 한 번 뽑았을 때 당첨될 확률을 p라고 가정할때 가능성이 높은 p를 구하라

확률변수 $x_i = 1$ (당첨인경우)
= 0 (당첨이 아닌경우)

i	x_i	$P(x = x_i)$
1	1	P
2	0	1-p
3	0	1-p
4	1	P
5	0	1-p

$$P(x = x_1) * P(x = x_2) * P(x = x_3) * P(x = x_4) * P(x = x_5)$$

=p*(1-p)*(1-p)*p*(1-p)
=p^2 * (1-p)^3

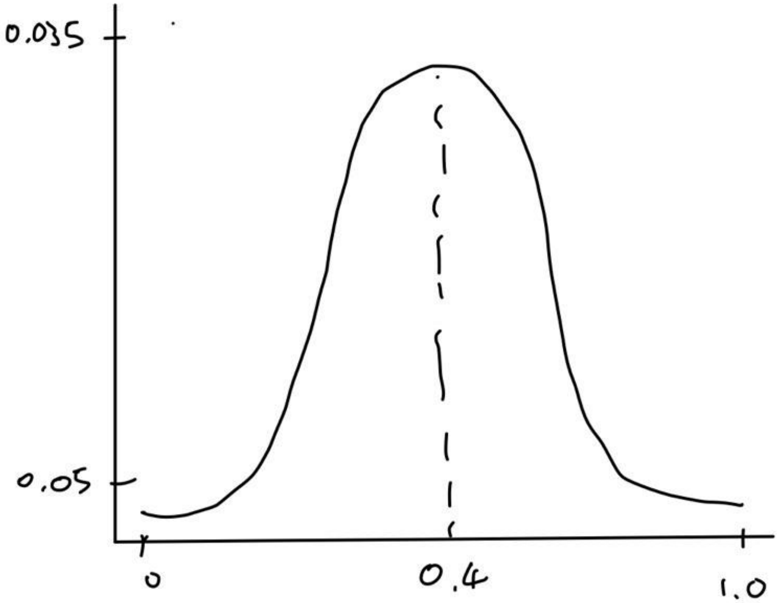
가능도 함수
-모델의 확률적 특징을 나타내는 변수를 포함한 식

최대가능도 추정
-가능도 함수를 매개변수로 미분 했을 때, 그 값이 0이 되게 하는 매개변수 값을 가장 확률이 높은 매개변수로 추정하는 알고리즘

$\log(P^2(1-P)^3)$ = 2logp +3log(1-p)

$$\frac{2}{p}+\frac{3\cdot(-1)}{1-P}=0$$

$$p=\frac{2}{5}$$



가정

입력값 x1 x2 x3 x4 x5

입력값 x는 (x0, x1, x2)=(1, x1, x2)

정답값 y1 y2 y3 y4 y5

정답값은 (1,0,0,1,0)

i	x_i	**yP**
1	1	P
2	0	1-p
3	0	1-p
4	1	P
5	0	1-p

$$P^{(1)} * P^{(2)} * P^{(3)} * P^{(4)} * P^{(5)}$$

$$=> \log(P^{(1)} * P^{(2)} * P^{(3)} * P^{(4)} * P^{(5)})$$

$$= \log(P^{(1)}) + \log(P^{(2)}) * \log(P^{(3)}) * \log(P^{(4)}) * \log(P^{(5)}))$$

$$= \log(P^{(\mathrm{m})}) = y^{(m)}\log\left(yp^{(m)}\right) + \left(1 - y^{(m)}\right)\log(1 - yp^{(m)})$$

$m = i$	x_i	yP
1	1	P
2	0	1-p
3	0	1-p
4	1	P
5	0	1-p

$\log(P^{(\mathrm{m})}) = y^{(m)}\log\left(yp^{(m)}\right) + \left(1 - y^{(m)}\right)\log(1 - yp^{(m)})$

if m=1 ->
$y^{(1)}\log\left(yp^{(1)}\right) + \left(1 - y^{(1)}\right)\log(1 - yp^{(1)})$
=1*log($yp^{(1)}$)+(1-1)log($1 - yp^{(1)}$)

if m=2 ->
$y^{(2)}\log\left(yp^{(2)}\right) + \left(1 - y^{(2)}\right)\log(1 - yp^{(2)})$
=0*log($yp^{(2)}$)+(1-0) log($1 - yp^{(2)}$)

$$\sum_{m=1}^{5} \log(P^{(\mathrm{m})}) = \sum_{m=1}^{5} y^{(m)}\log\left(yp^{(m)}\right) + \left(1 - y^{(m)}\right)\log(1 - yp^{(m)})$$

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

로지스틱 비용함수의 편도함수

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

배치 경사 하강법에서 사용한 비용함수의 편도함수

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

결정경계

In [52]:

```python
from sklearn import datasets
iris = datasets.load_iris()
list(iris.keys())
```

Out [52]:

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

In [56]:

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

In [58]:

```python
#로지스틱 회귀모델 훈련
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression(solver="lbfgs", random_state=42) #lbfgs = 멀티클래스의 분류모델에 사용/ 성능이 좋은 알고리즘으로 알려
log_reg.fit(X, y)
```

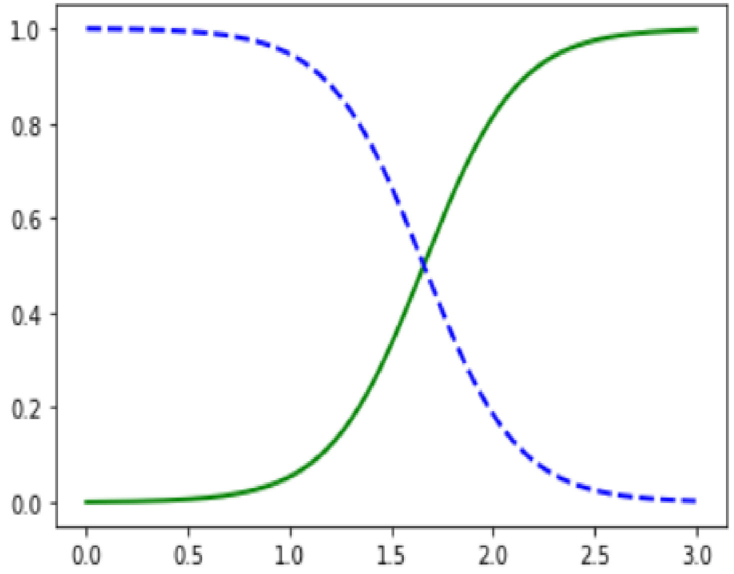
Out [58]:

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

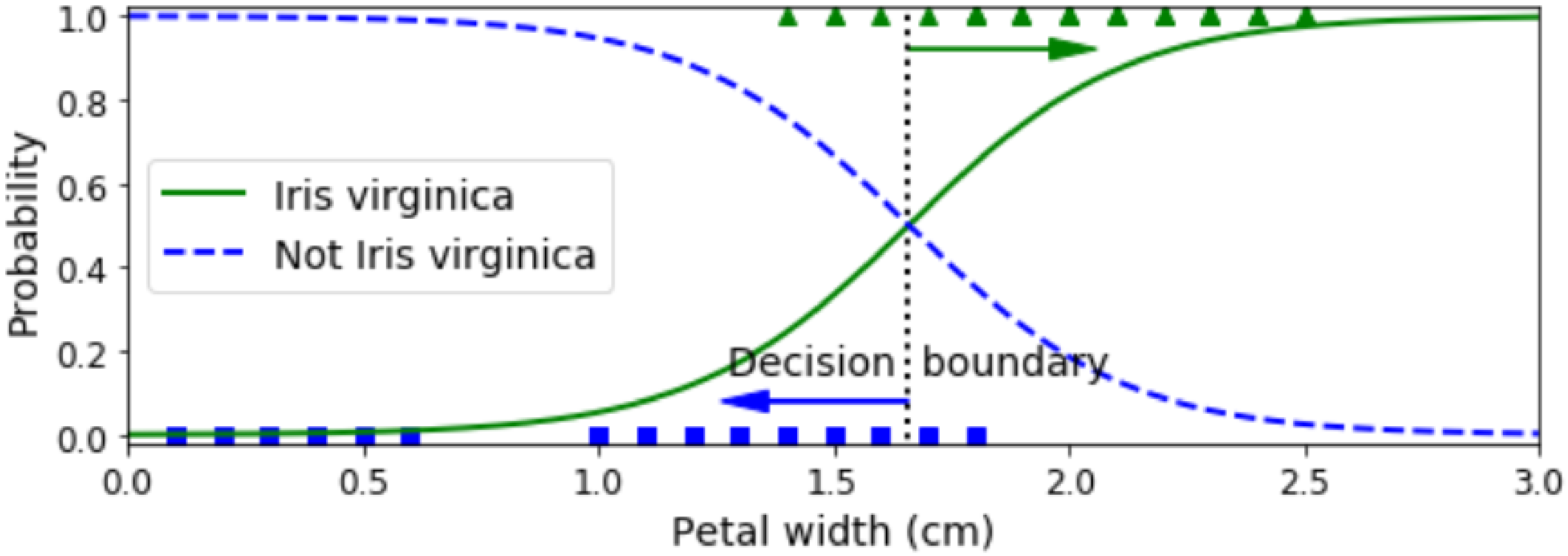
In [141]:

```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1) #0부터 3까지 1000개의 숫자를 만든다.
y_proba = log_reg.predict_proba(X_new)
decision_boundary = (X_new[y_proba[:, 1] >= 0.5])[0] #[y_proba의 두번째행의값>=0.5]의 첫번째 값(결정경계값)
plt.plot(X_new, y_proba[:, 1], "g-", linewidth=2, label="Iris virginica")
plt.plot(X_new, y_proba[:, 0], "b--", linewidth=2, label="Not Iris virginica")
```

Out[141]: [<matplotlib.lines.Line2D at 0x7f2adf49c6a0>]



확률이 똑같이 50%가 되는 부근



소프트맥스 회귀-다중클래스

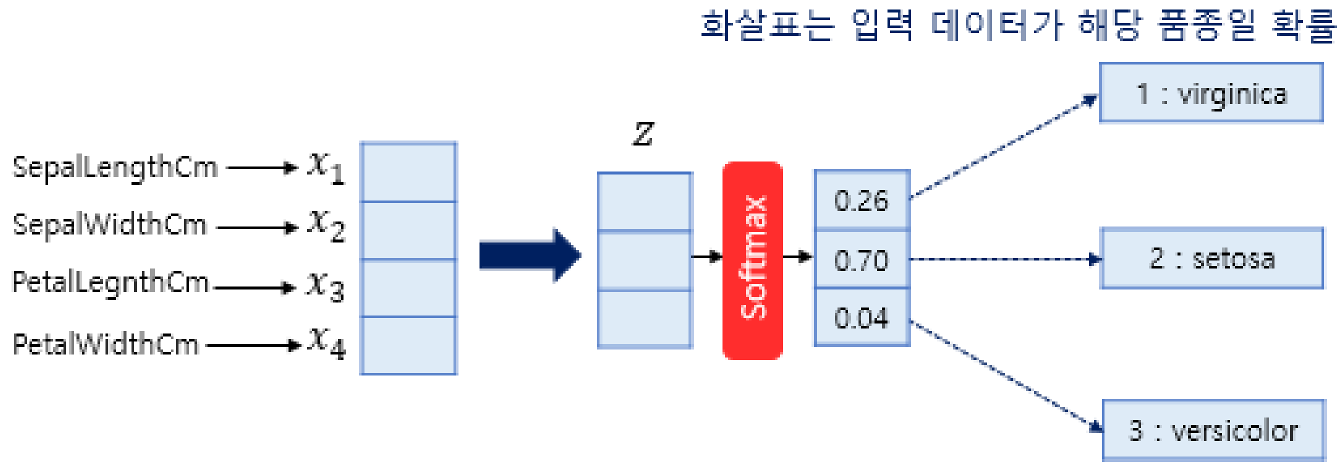
$$s_k(\mathbf{x}) = \left(\theta^{(k)}\right)^T \cdot \mathbf{x}$$

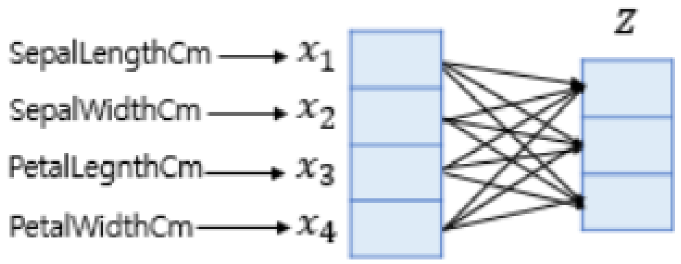
샘플 X가 주어졌을 때 각 클래스 k에 대한 점수 $s_k(x)$를 계산

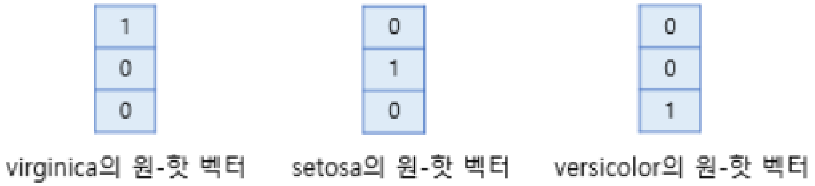
그 점수에 소프트맥스 함수를 적용하여 확률 추정

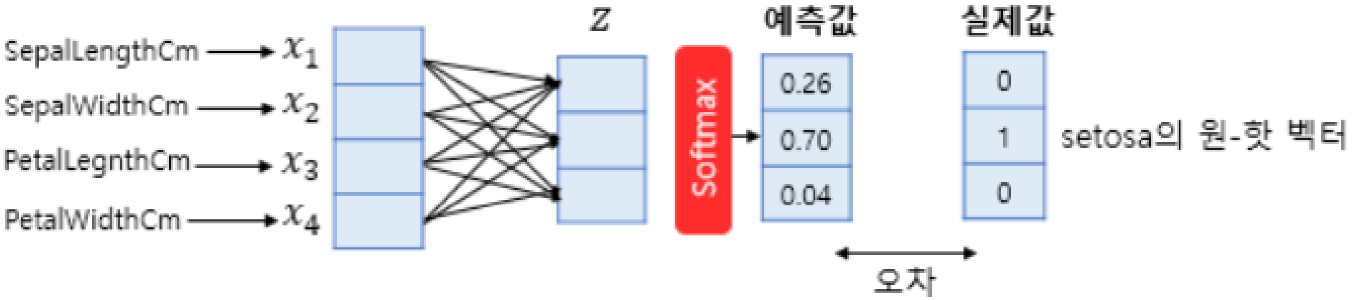
$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp(s_k(\mathbf{x}))}{\sum_{j=1}^{K}\exp(s_j(\mathbf{x}))}$$

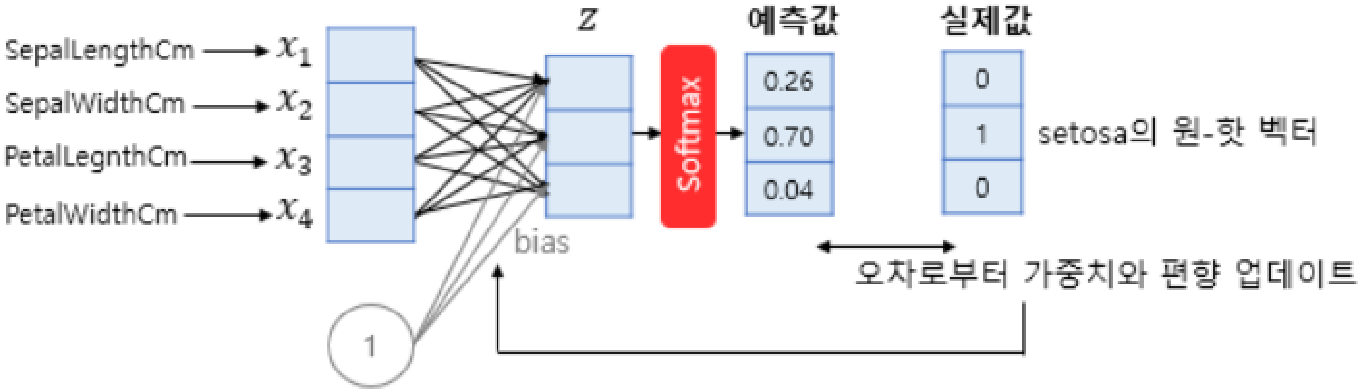
- $s_k(x)$는 샘플x에 대한 각 클래스의 점수를 담은 벡터
- $\sigma(s(x))_k$는 샘플x에 대한 각 클래스의 점수가 주어졌을 때 샘플이 클래스 k에 속할 추정확률


화살표는 입력 데이터가 해당 품종일 확률
SepalLengthCm → x_1
SepalWidthCm → x_2
PetalLegnthCm → x_3
PetalWidthCm → x_4
z
Softmax
0.26
0.70
0.04
1 : virginica
2 : setosa
3 : versicolor


SepalLengthCm → x_1
SepalWidthCm → x_2
PetalLegnthCm → x_3
PetalWidthCm → x_4
z


1
0
0
virginica의 원-핫 벡터
0
1
0
setosa의 원-핫 벡터
0
0
1
versicolor의 원-핫 벡터


SepalLengthCm
SepalWidthCm
PetalLegnthCm
PetalWidthCm
x_1
x_2
x_3
x_4
z
Softmax
예측값
0.26
0.70
0.04
실제값
0
1
0
setosa의 원-핫 벡터
오차


SepalLengthCm
SepalWidthCm
PetalLegnthCm
PetalWidthCm
x_1
x_2
x_3
x_4
z
Softmax
예측값
0.26
0.70
0.04
실제값
0
1
0
setosa의 원-핫 벡터
bias
1
오차로부터 가중치와 편향 업데이트

cross entropy를 비용함수로 사용

$$cost(W) = -\sum_{j=1}^{k} y_j \, log(p_j)$$

y_j는 실제값(원-핫 벡터의 j번째 인덱스)
K는 클래스의 개수
P_j는 샘플데이터가 j번째 클래스일 확률

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

cross entropy의 gradient vector

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

-각 클래스에 대한 그래디언트 백터를 계산할 수 있으므로
비용함수를 최소화하기 위한 파라미터 행렬을 찾기위해 SGD를 사용할 수 있음