[3주차]
모델 훈련

1기 이다현
1기 최하경

EURON

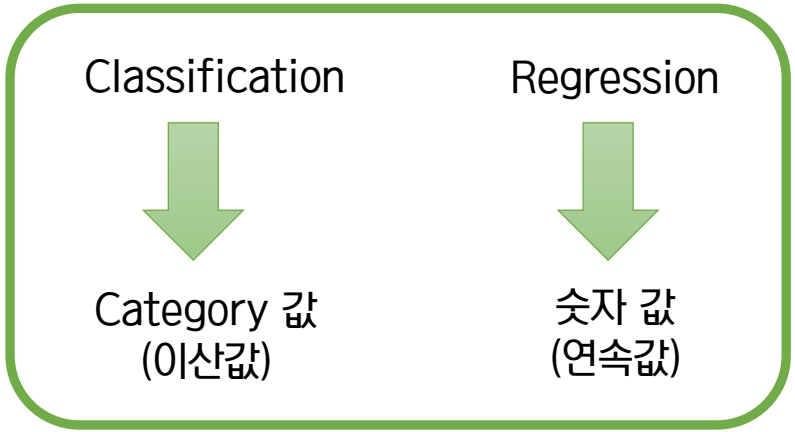
목차

4.1. 선형 회귀

회귀 여러 개의 독립변수와 한 개의 종속변수 간의 상관관계를 모델링하는 기법을 통칭한다.

- 지도학습 방법 중 하나로, 예측값이 연속형 숫자값이다.
- 독립변수=피처(feature), 종속변수=결정 값

- 머신러닝 회귀 예측의 핵심:
  주어진 피처와 결정 값 데이터 기반에서 학습을 통해
  최적의 회귀계수를 찾아내는 것



종류

독립변수 개수	회귀 계수의 결합
1개: 단일 회귀	선형: 선형 회귀
여러 개: 다중 회귀	비선형: 비선형 회귀

4.1. 선형 회귀

개념

잔차 제곱합(MSE)를 최소화하는 직선형 회귀선을 최적화하는 방식

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

(벡터 표현식) $\hat{y} = h_\theta(x) = \theta * x$

$\hat{y}$: 예측값
n : 특성의 수
$x^{(i)}$: (i)번째 특성
θ_j : j 번째 모델 파라미터

종류

규제 방법에 따라

- **일반 선형 회귀** : 규제를 적용하지 않은 모델
- **릿지 (Ridge)** : 선형 회귀에 L2 규제를 추가한 모델
- **엘라스틱넷 (ElasticNet)** : L2, L1 규제를 함께 결합한 모델
  주로 피처가 많은 데이터 세트에 적용
  L1 규제로 피처의 개수를 줄이는 동시에
  L2 규제로 계수 값의 크기 조정
- **라쏘 (Lasso)** : 선형 회귀에 L1 규제를 적용한 모델
- **로지스틱 회귀 (Logistic Regression)** : 회귀에 속하지만 분류에 사용되는 선형 모델
  매우 강력한 분류 알고리즘
  이진 분류뿐만 아니라 희소 영역의 분류에서도
  뛰어난 예측 성능을 보임

4.1. 선형 회귀

훈련 모델이 훈련 세트에 가장 잘 맞도록 모델 파라미터를 설정하는 것

선형 회귀 모델을 훈련시키려면 성능 측정 지표인 평균 제곱근 오차 (RMSE)를 최소화하는 파라미터 찾아야 한다.

가설함수 우리가 시도하는 함수 하나하나 $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \cdots$

손실함수 가설함수인 h_θ 의 성능을 평가하기 위한 함수

- 회귀 계수로 구성되는 MSE (비용/cost)
- 도출값인 MSE가 작을수록 데이터에 더 잘 맞는, 좋은 가설함수

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

가장 좋은 회귀식을 도출하는 방법

- 정규 방정식
- 특이값 분해 SVD
- 경사하강법

4.1.1 정규방정식

정규방정식 비용함수를 최소화하는 θ 값을 찾기 위한 해석적인 방법 (수학공식)

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

역행렬을 구할 수 없다면 정규방정식도 구할 수 없다 -> SVD는 유사역행렬을 구할 수 있다 (모든 직각행렬에 대해 연산 가능!)

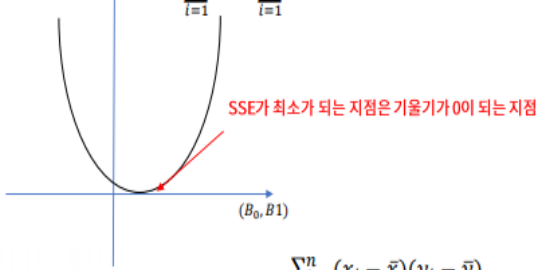
미분을 통해 수학적으로 모수(계수) 를 구하는 공식

사이킷런의 선형회귀는 (특잇값 분해를 사용해) 유사역행렬을 계산한다.

* 행렬과 벡터 : 머신러닝을 할 때는 데이터를 일차식에 사용하는 경우가 많다 → 행렬을 이용하면 정돈된 형태로 효율적이게 계산을 할 수 있다

SSE $\hat{\beta}_0$과 $\hat{\beta}_1$로 편미분하여 연립방정식을 푸는 방법(Least Square Method)

$$SSE = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i)^2$$



$$\frac{\partial L}{\partial \beta_0} = -2\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_i) = 0$$

$$\frac{\partial L}{\partial \beta_1} = -2\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_i)x_i = 0$$

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$$

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

예측값(1x1행렬)

RMSE가 최소가 되는 지점은 MSE의 미분식이 0일 때 이다. 시그마를 없애고 행렬곱으로 표현하여 미분하고 정리하면 정규방정식을 끌어낼 수 있다.

$$MSE = \frac{1}{m}(X\hat{\theta} - y)^2$$

$$0 = \frac{2}{m} X^T (X\hat{\theta} - y)$$

$$0 = X^T X\hat{\theta} - X^T y$$

$$X^T y = X^T X\hat{\theta}$$

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

4.1.1 정규방정식

정규방정식에서 벡터를 이용하는 이유

(1) 변수가 많지 않은 아래 2차 방정식 수식의 경우

$$J(\theta) = a\theta^2 + b\theta + c$$

➡ 미분값을 구해 0을 만드는 회귀 계수를 구하면 된다

(2) 회귀 계수가 많은 경우

$$J(\theta_0, \theta_1, \ldots, \theta_m) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

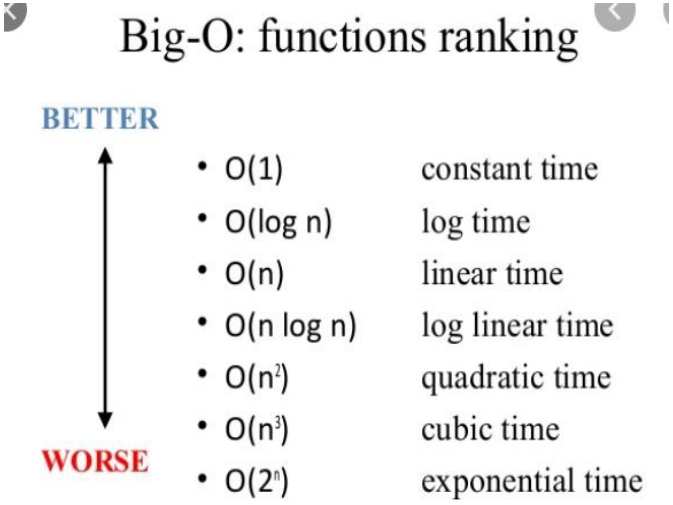
➡ 각 회귀계수에 대해 편미분을 하면 되지만, 회귀 계수가 많아 계산과정이 복잡하다.

4.1.2 계산복잡도

빅오 표기법 알고리즘의 **효율성**을 표기해주는 표기법

알고리즘의 효율성은 데이터 개수(n)가 주어졌을 때 덧셈, 뺄셈, 곱셈 같은 기본 **연산의 횟수**를 의미한다.

빅오 표기법은 보통 알고리즘의 시간 복잡도와 공간 복잡도를 나타내는데 주로 사용 된다.
(시간 복잡도는 알고리즘의 시간 효율성을 의미하고, 공간 복잡도는 알고리즘의 공간(메모리) 효율성을 의미한다.)



계산 복잡도 역행렬을 계산할 때 발생하는 복잡도

n = 특성 수

- 정규 방정식의 시간 복잡도 $O(n^{2.4})$
- SVD의 시간복잡도 $O(n^2)$

→ 두 방법 모두 특성수가 많아지면 매우 느려짐

m = 샘플 수 $\boldsymbol{O(m)}$

→ 훈련 세트의 샘플수에 대해선 복잡도가 선형적으로 증가하므로 메모리 공간이 허락된다면 훈련 세트도 효율적으로 처리할 수 있다.

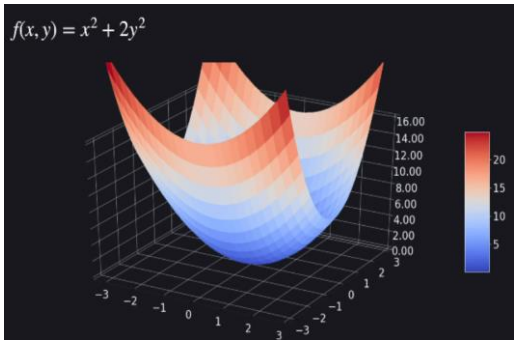
4.2 경사하강법

+ 수학 개념 복습 : 편미분 | 기울기의 2가지 의미

편미분

고차원 [다변수 함수] 에서의 미분
partial derivative

- input 변수가 2개 이상 = **다변수 함수** → 3차원 이상



input 변수가 2개 이상인 경우에는, 그냥 미분이 아니라 '편미분' 을 해야 한다. '편미분'이란 $f(x,y) = x^2 + 2y^2$을 예로 들면 함수를 두 인풋 변수 x,y에 대해서 모두 미분을 하는 게 아니라 변수 하나에 대해서만 미분을 하는 것이다.

- **o** X에 대한 편미분 : x를 제외한 나머지 변수들은 마치 상수인 것처럼 (즉, 값을 고정 시킨다는 뜻) 취급하고 미분

$$f(x,y) = x^2 + 2y^2$$

$$\frac{\partial}{\partial x} f(x,y) = 2x$$

- **o** y에 대한 편미분 : y를 제외한 나머지 변수들은 마치 상수인 것처럼 (즉, 값을 고정 시킨다는 뜻) 취급하고 미분

$$f(x,y) = x^2 + 2y^2$$

$$\frac{\partial}{\partial y} f(x,y) = 4y$$

$$\nabla f(x,y) = \begin{bmatrix} 2x \\ 4y \end{bmatrix}$$

- 다변수 함수 f(x,y)의 기울기 : x에 대한 편미분과 y에 대한 편미분의 결과를 합쳐서 **벡터**를 만든다.

편미분 → 기울기 **벡터** → 함수에서 가장 가파르게 올라가는 방향

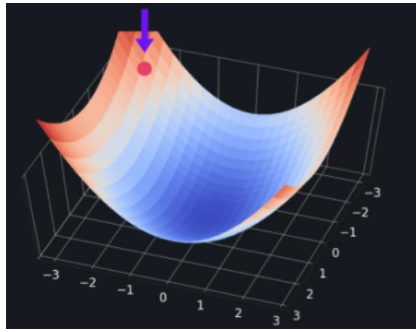
기울기

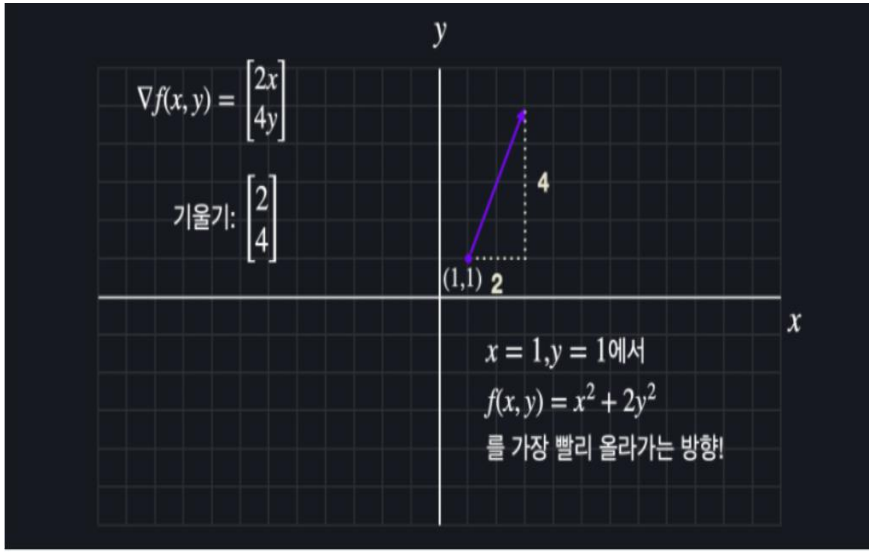
기울기는 그래프가 해당 지점에서 '얼마나 기울어져 있는지 (=x가 변화함에 따라 y는 어떻게 변화하는지)'도 알려주지만 '어떤 방향으로 가야 **가장 가파르게 올라갈 수 있는지**' 에 대해서도 알려준다. 특정 지점에서 가장 가파르게 올라가려면 왼쪽으로 가야 하는지, 오른쪽으로 가야 하는지 기울기가 그 방향을 알려주는 기준이 된다.

가파르게 올라간다?

$$f(x,y) = x^2 + 2y^2$$

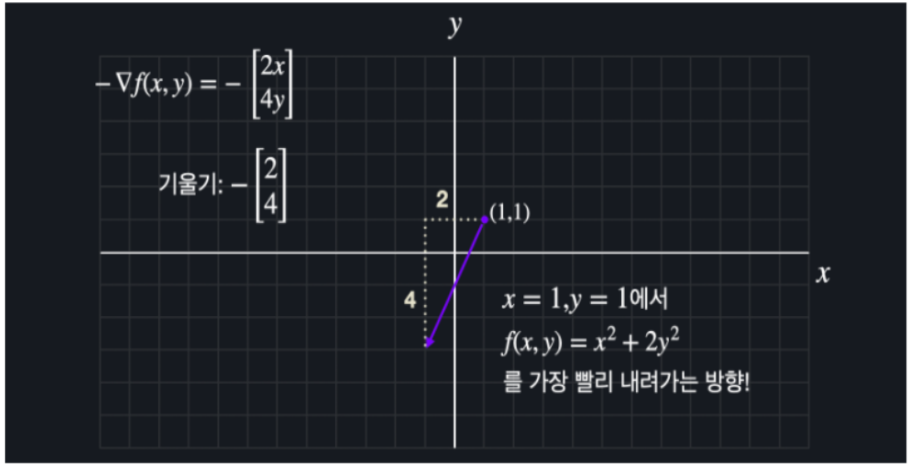
$$\nabla f(x,y) = \begin{bmatrix} 2x \\ 4y \end{bmatrix}$$





그러면 기울기 벡터는 2, 4죠? 이 벡터를 그래프에 표시하자면, x 방향으로 두 칸이고 y 방향으로 네 칸이니까, 이런 대각선 방향인데요. 이 대각선 방향으로 움직이면, 가장 가파르게 걸어 올라갈 수 있는 겁니다.

그럼 반대로 우리가 가장 가파르게 걸어 내려가고 싶다면 어느 방향으로 가야 할까요? 그냥 이 기울기 벡터에 마이너스를 붙여 주면 되겠죠?

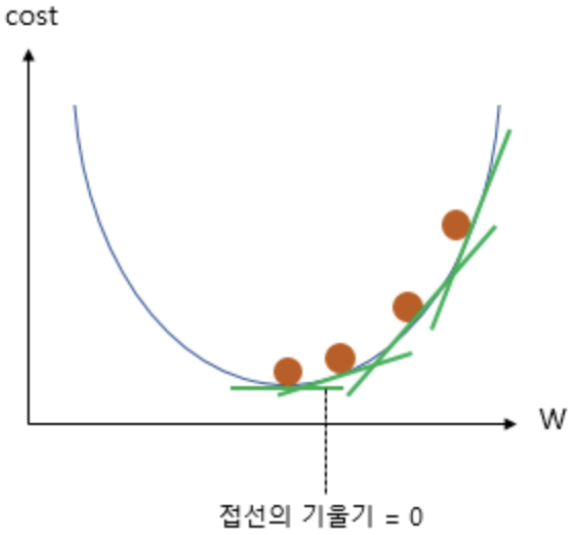


다시 x가 1이고 y가 1인 지점이라고 가정하면, 벡터는 -2, -4인데요. 이 벡터를 그래프에 표시하면 이런 대각선 방향입니다. 이 방향으로 움직이면, 가장 가파르게 걸어 내려갈 수 있습니다.

4.2 경사 하강법

개념 비용함수를 최소화하는 회귀 계수를 찾기 위해 사용되는 알고리즘

- 반복적으로 비용함수의 반환값인 MSE가 작아지는 방향성을 갖고 비용함수에서의 최솟값을 찾기 위해 회귀 계수를 지속적으로 보정해 나간다.



핵심 어떻게 하면 오류가 작아지는 방향으로 회귀 계수의 값을 보정할 수 있을까?

그래프 관점에서 회귀계수는 직선의 기울기

원함수를 미분한 도함수(직선의 기울기)가 양수일 때 원함수는 증가하다가 0이 되면 그때의 원함수 값이 최대이다.
직선의 기울기가 음수일 때 원함수는 감소하다가 기울기가 0이 되면 이때의 원함수 값이 최소이다.

MSE가 최소가 되는 값을 구하기 위해서는 손실함수를 미분해 0이 되는 회귀 계수를 구하면 된다.

4.2 경사 하강법

단순선형회귀모델에서의 경사 하강법

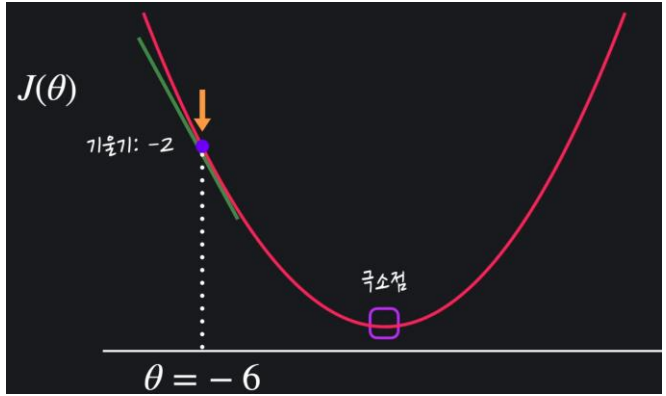
$$R(x) = \frac{1}{N}\sum_{i=1}^{N}(y_i - (\theta_0 + \theta_i * x_i))^2$$

$$\frac{\partial R(x)}{\partial \theta_1} = \frac{2}{N}\sum_{i=1}^{N} -x_i * (y_i - (\theta_0 + \theta_i * x_i)) = \frac{2}{N}\sum_{i=1}^{N} -x_i * (\text{실제값}_i - \text{예측값}_i))$$

$$\frac{\partial R(x)}{\partial \theta_0} = \frac{2}{N}\sum_{i=1}^{N} -(y_i - (\theta_0 + \theta_i * x_i)) = \frac{2}{N}\sum_{i=1}^{N} -(\text{실제값}_i - \text{예측값}_i))$$

$$w_1 = \text{이전 } w_1 + \eta\frac{2}{N}\sum_{i=1}^{N} x_i * (\text{실제값}_i - \text{예측값}_i))$$

$$w_0 = \text{이전 } w_0 + \eta\frac{2}{N}\sum_{i=1}^{N} (\text{실제값}_i - \text{예측값}_i))$$



현 위치보다 0에 가까운 기울기를 갖는
회귀계수를 찾아 이동한다.

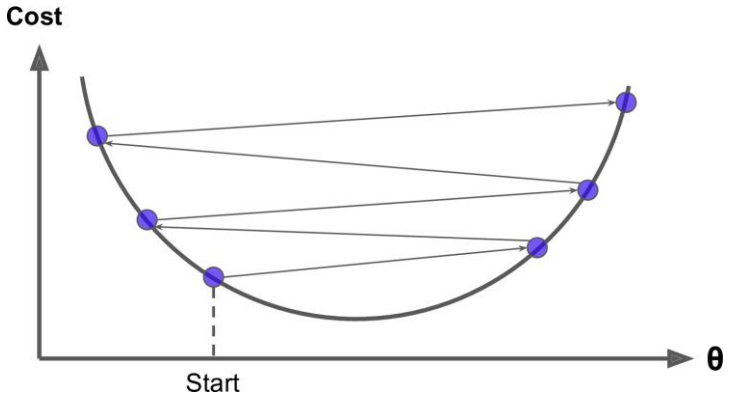
학습률 η 보정계수

최적의 회귀계수를 찾기 위해
움직일 때 얼마나 많이 움직일지
그 정도를 나타내는 수치

4.2 경사 하강법

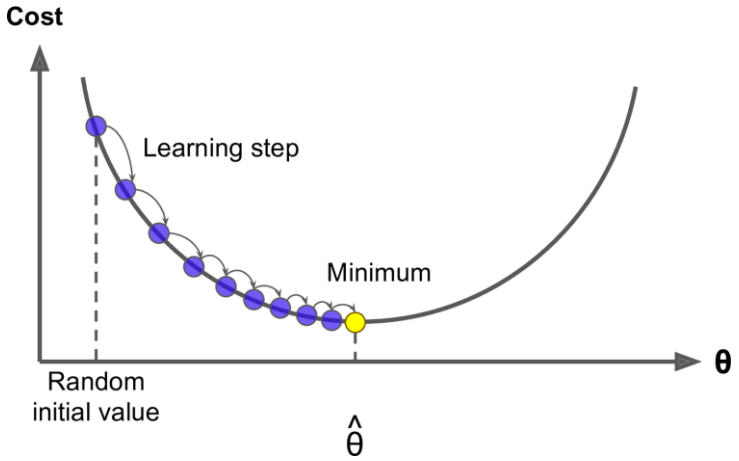
학습률 η 최적의 회귀계수를 찾기 위해 움직일 때 얼마나 많이 움직일지 그 정도를 나타내는 수치

학습률 η이 너무 클 때



최솟값을 가로질러 반대 경사로 뛰어넘는 경우가 발생할 수 있다.

학습률 η이 너무 작을 때

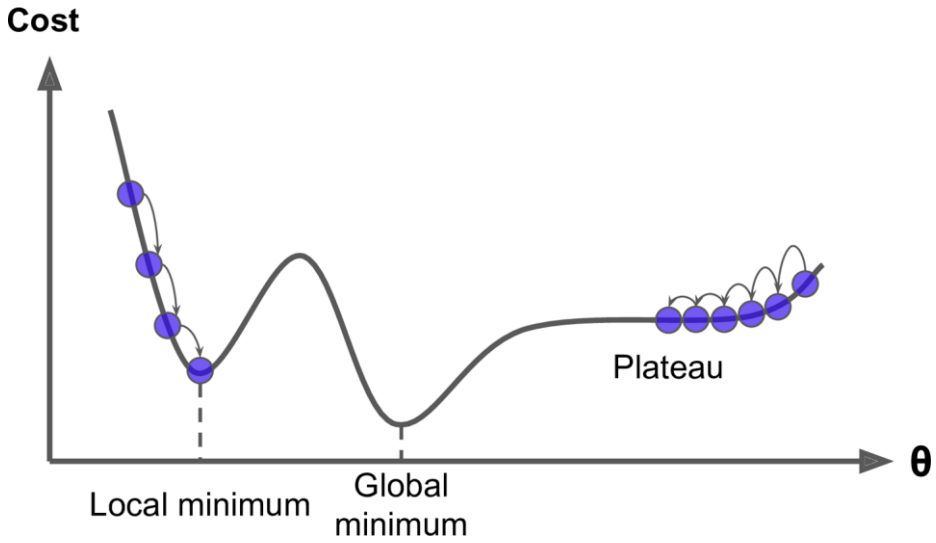


시간이 오래 걸리고 굴곡이 있는 비용함수(변곡점이 많은)의 경우 min값을 찾지 못할 수도 있다.

4.2 경사 하강법

볼록 함수

어떤 두 점을 선택해 선을 그어도 곡선을 가로지르지 않는 함수



하나의 전역 최솟값만을 가진다는 점, 연속된 함수이며 기울기가 갑자기 변하지 않는다는 점에서 학습률이 너무 높지 않고 충분한 시간이 주어진다면 경사 하강법이 전역 최솟값에 가깝게 접근할 수 있다.

4.2 경사 하강법

Step 1. 회귀계수들을 각각 임의의 값으로 설정하고 첫 손실 함수의 값을 계산한다.

Step 2. w_1 을 이전 $w_1 + \eta \frac{2}{N} \sum_{i=1}^{N} x_i * (\text{실제값}_i - \text{예측값}_i)$), w_0을 이전 $w_0 + \eta \frac{2}{N} \sum_{i=1}^{N} (\text{실제값}_i - \text{예측값}_i))$으로 업데이트한 후 다시 손실 함수의 값을 계산한다 .

Step 3. 손실 함수의 값이 감소했으면 다시 Step 2를 반복한다.

Step 4. 더 이상 손실 함수의 값이 감소하지 않으면 그때의 w_1, w_0를 구하고 반복을 중지한다.

4.2 경사 하강법

경사 하강법에 사용되는 용어

Epoch

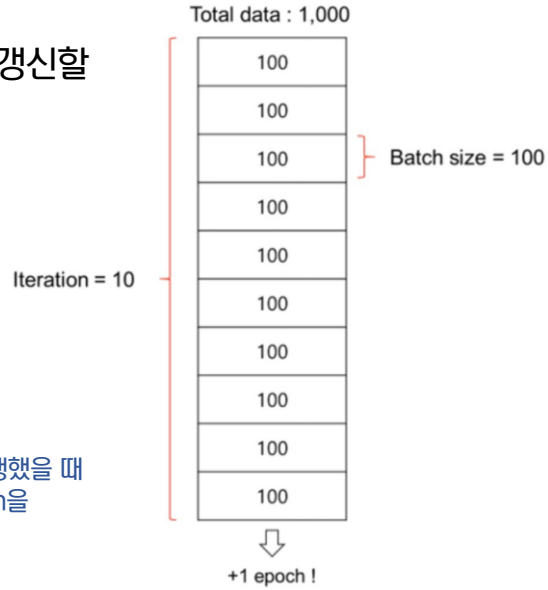
모든 훈련 데이터셋을 학습하는 횟수
너무 많은 epoch는 overfitting의 위험이, 너무 적은 epoch는 underfitting의 위험이 있다.

Batch size

훈련 데이터셋 중 몇 개의 데이터를 묶어 회귀 계수의 값을 갱신할 것인지에 대한 정보

Iteration

한 epoch를 진행하기 위해 몇 번의 업데이트가 이루어졌는지에 대한 정보



만약 훈련 데이터셋의 개수가 1000개, 1 epoch를 진행했을 때 batch size를 100으로 설정하면 총 10번의 iteration을 거친다.

4.2.1. 배치 경사 하강법 Batch Gradient Descent (BGD)

개념 배치 사이즈가 훈련세트 사이즈와 동일한 경사 하강법으로, 전체 훈련세트를 한번에 처리해 기울기를 업데이트한다.

경사 하강법을 구현하기 위해 각 모델 파라미터에 대해 손실 함수의 gradient를 계산해야 한다.

Gradient Vector

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

각각의 회귀 계수가 조금 변경될 때 손실함수가 얼마나 바뀌는지 계산하는 편도함수를 담고 있다

배치 경사 하강법에서는 각 모델 파라미터마다의 편도함수를 모두 담고 있다.

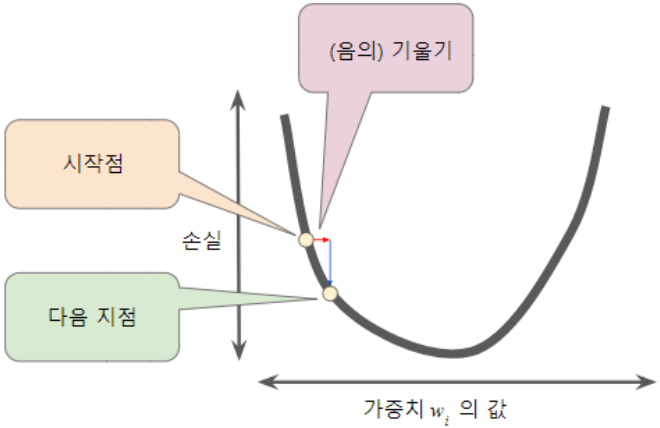
전체 데이터 셋에 대한 손실을 계산하고
그래디언트(Gradient)의 반대 방향으로 매개 변수를 개선하여 얼마나 큰 업데이트를 수행할지를 학습률 (Learning-Rate)을 이용하여 진행한다.

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

4.2.1. 배치 경사 하강법 Batch Gradient Descent (BGD)

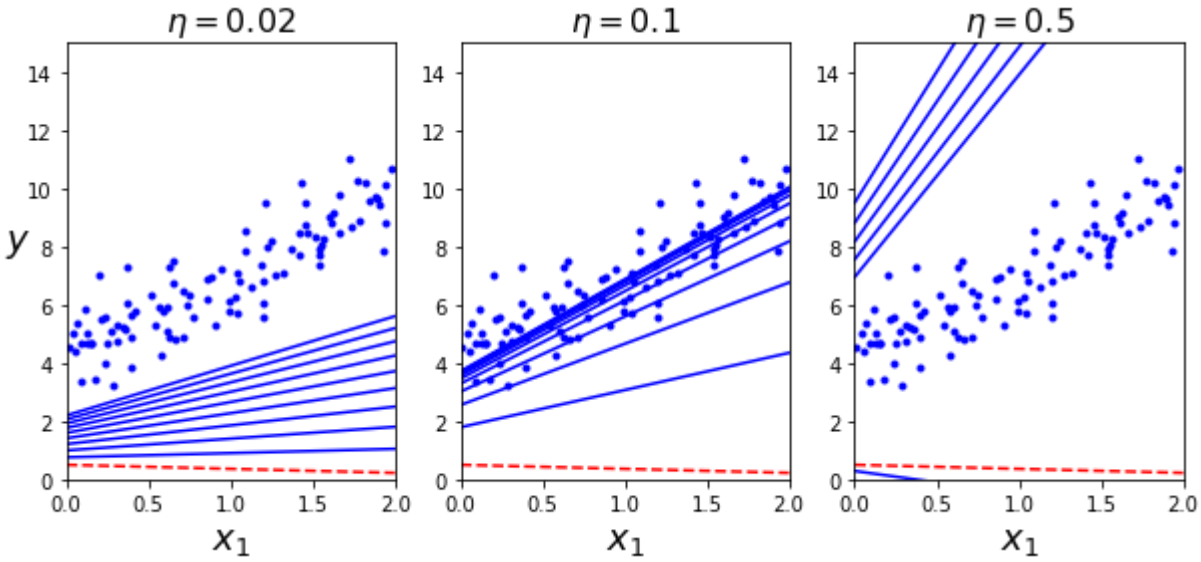
Gradient Vector

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



다음 파라미터에서 학습률을 곱한 gradient vector을 빼는 이유:
스텝이 전역 최솟값의 좌측에서 시작한 경우 음의 값을 갖기 때문에 +로 파라미터 값을 키워 우측으로 가야하기 때문이다.

4.2.1. 배치 경사 하강법 Batch Gradient Descent (BGD)

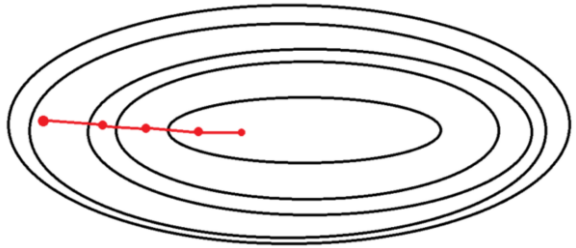


학습률이 너무 낮으면 시간이 오래 걸리고, 너무 높으면 최적점을 넘어가 버린다.

적절한 학습률을 찾기 위해 그리드 탐색을 사용할 수 있다.
적절한 반복횟수를 찾기 위해서는 먼저 반복횟수를 크게 지정하고, gradient vector이 매우 작아지면 알고리즘을 중지하게 하는 방법 채택한다.

4.2.1. 배치 경사 하강법 Batch Gradient Descent (BGD)

개념 배치 사이즈가 훈련세트 사이즈와 동일한 경사 하강법으로, 전체 훈련세트를 한번에 처리해 기울기를 업데이트한다.



장점

- 항상 같은 데이터를 이용해 경사를 구하기 때문에 수렴이 안정적이다.
- 수십 만개의 특성에서 선형회귀를 훈련시킬 때 정규방정식이나 SVD 분해보다 더 빠르다.

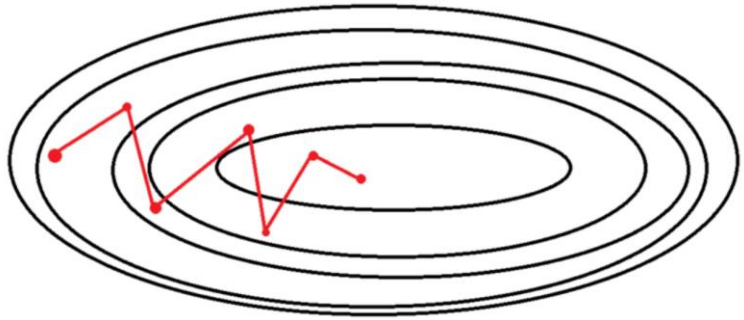
단점

- 경사 하강법들 중 가장 많은 메모리를 요구하며 긴 시간이 소요

4.2.2. 확률적 경사 하강법 Stochastic Gradient Descent (SGD)

개념 Batch size가 1인 경사 하강법으로, 전체 데이터 중 단 하나의 데이터를 랜덤하게 선택해 이를 이용하여 경사 하강법을 1회 진행한다.

확률적이기 때문에 기울기의 방향이 매번 크게 바뀐다.
(Shooting 발생)



장점

- 배치 경사 하강법에 비해 적은 데이터로 학습할 수 있다
- 계산량이 적어 속도가 빠르다.
- Shooting으로 지역 최솟값을 뛰어넘을 수 있다.

단점

- Shooting으로 인해 전역 최솟값에 수렴하기 어렵다.

학습률을 크게 설정하고, 점차 줄이면서 전역 최솟값에 도달하게 한다.

4.2.3. 미니 배치 경사 하강법

Mini Batch Gradient Descent

개념 배치 경사 하강법이나 확률적 경사 하강법과 달리 각 스텝에서 미니배치라고 부르는 임의의 작은 샘플 집합에 대해 계산한다.

장점

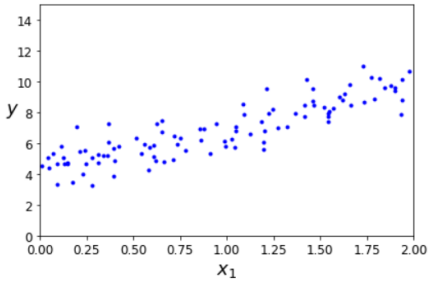
- GPU를 이용해 성능을 향상시킬 수 있다.
- 미니배치를 어느정도 크게 하면 파라미터 공간에서 SGD보다 덜 불규칙적으로 움직인다.

단점

- SGD보다 최솟값에 더 가까이 도달할 수 있지만 지역 최솟값에서 빠져나오기 힘들 수 있다.

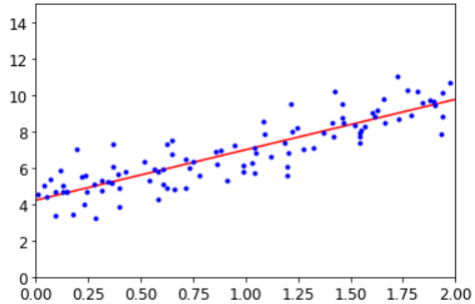
4.2 경사 하강법 코드 실습 #1 정규 방정식을 이용한 선형회귀

```
[2]  import numpy as np

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




```
[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) # 세타햇의 값을 구하는 공식입니다.
```

```
[5]  theta_best

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



```
[24] X_new = np.array([[0], [2]])
     X_new_b = np.c_[np.ones((2, 1)), X_new]X_new = np.array([[0], [2]])  # 위와 동일하게 모든 샘플에 x0 = 1을 추가합니다.
     y_predict = X_new_b.dot(theta_best) # 위에서 구한 세타햇의 값을 회귀계수로 두고 계산합니다.
     y_predict

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

4.2 경사 하강법 코드 실습 #2 선형 회귀

[9]
```
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]]))
```

[10]
```
X_new = np.array([[0], [2]])
lin_reg.predict(X_new)
```

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

4.2 경사 하강법 코드 실습 #3 배치 경사 하강법을 사용한 선형 회귀

[13]
```
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
```

[14]
```
theta
```

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

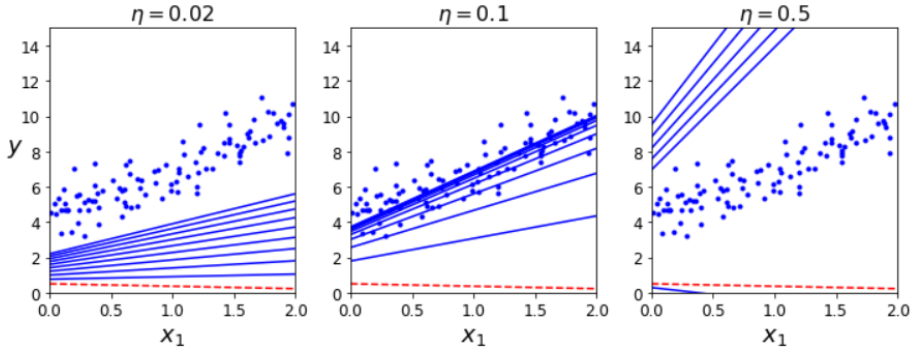
[15]
```
X_new_b.dot(theta)
```

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

4.2 경사 하강법

코드 실습 #3 배치 경사 하강법을 사용한 선형 회귀

```
[16] theta_path_bgd = []

     def plot_gradient_descent(theta, eta, theta_path=None):
         m = len(X_b)
         plt.plot(X, y, "b.")
         n_iterations = 1000
         for iteration in range(n_iterations):
             if iteration < 10:
                 y_predict = X_new_b.dot(theta)
                 style = "b-" if iteration > 0 else "r--"
                 plt.plot(X_new, y_predict, style)
             gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
             theta = theta - eta * gradients
             if theta_path is not None:
                 theta_path.append(theta)
         plt.xlabel("$x_1$", fontsize=18)
         plt.axis([0, 2, 0, 15])
         plt.title(r"$\eta = {}$".format(eta), fontsize=16)
```



plot_gradient_descent 함수를 만들어 학습률에 따라 회귀선이 어떻게 변하는지 알아보는 코드입니다.

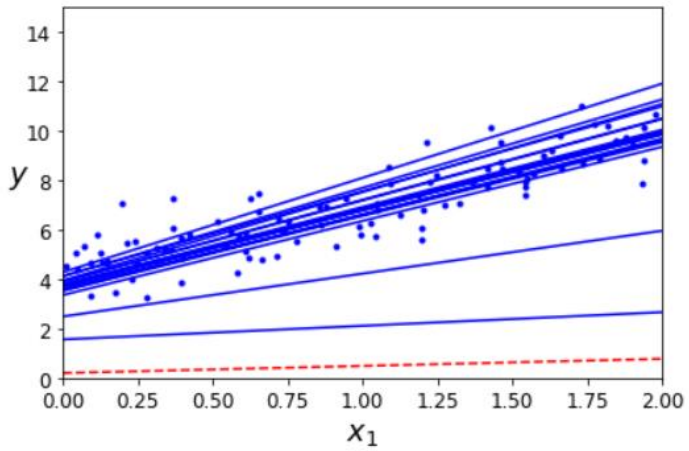
```
[17] np.random.seed(42)
     theta = np.random.randn(2,1)  # random initialization

     plt.figure(figsize=(10,4))
     plt.subplot(131); plot_gradient_descent(theta, eta=0.02)
     plt.ylabel("$y$", rotation=0, fontsize=18)
     plt.subplot(132); plot_gradient_descent(theta, eta=0.1, theta_path=theta_path_bgd)
     plt.subplot(133); plot_gradient_descent(theta, eta=0.5)

     save_fig("gradient_descent_plot")
     plt.show()
```

4.2 경사 하강법

코드 실습 #4 확률적 경사 하강법을 사용한 선형 회귀

```
theta_path_sgd = []
m = len(X_b)
np.random.seed(42)
```

```
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:
            y_predict = X_new_b.dot(theta)
            style = "b-" if i > 0 else "r--"
            plt.plot(X_new, y_predict, style)
        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)
```




```
[20] theta

array([[4.21076011],
       [2.74856079]])
```

4.2 경사 하강법

코드 실습 #4 확률적 경사 하강법을 사용한 선형 회귀

```
[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())
```

```
SGDRegressor(alpha=0.0001, average=False, early_stopping=False, epsilon=0.1,
             eta0=0.1, fit_intercept=True, l1_ratio=0.15,
             learning_rate='invscaling', loss='squared_loss', max_iter=1000,
             n_iter_no_change=5, penalty=None, power_t=0.25, random_state=42,
             shuffle=True, tol=0.001, validation_fraction=0.1, verbose=0,
             warm_start=False)
```

```
[22] sgd_reg.intercept_, sgd_reg.coef_
```

```
(array([4.24365286]), array([2.8250878]))
```

Scikit-learn에서 제공하는 확률적 경사 하강법인 SGDRegressor을 이용해
앞서 나온 과정을 간단하게 실행할 수 있다.

4.2 경사 하강법 코드 실습 #5 미니 배치 경사 하강법을 사용한 선형 회귀

```
[23] theta_path_mgd = []

     n_iterations = 50
     minibatch_size = 20

     np.random.seed(42)
     theta = np.random.randn(2,1)  # 랜덤 초기화

     t0, t1 = 200, 1000
     def learning_schedule(t):
         return t0 / (t + t1)

     t = 0
     for epoch in range(n_iterations):
         shuffled_indices = np.random.permutation(m)
         X_b_shuffled = X_b[shuffled_indices]
         y_shuffled = y[shuffled_indices]
         for i in range(0, m, minibatch_size):
             t += 1
             xi = X_b_shuffled[i:i+minibatch_size]
             yi = y_shuffled[i:i+minibatch_size]
             gradients = 2/minibatch_size * xi.T.dot(xi.dot(theta) - yi)
             eta = learning_schedule(t)
             theta = theta - eta * gradients
             theta_path_mgd.append(theta)
```

```
[ ] theta

     array([[4.25214635],
            [2.7896408 ]])
```

이 방법 외에도 scikit-learn에서 SGDRegressor
SGRClassifier에서 partial_fit 메서드를 이용해 파라미터를
초기화하지 않고, 미니배치 학습을 위해 반복적으로 호출할 수 있다.

4.2 정리

미분계수가 0인 지점을 찾는 것이 아니라 gradient descent를 이용해 최솟값을 찾는 이유

1. 닫힌 형태가 아니거나 함수의 형태가 복잡해 미분계수와 그 근을 계산하기 어려운 경우가 있다.

2. 실제 미분계수를 계산하는 과정을 컴퓨터로 구현하는 것에 비해 gradient descent는 컴퓨터로 비교적 쉽게 구현할 수 있다.

3. 데이터의 양이 매우 큰 경우 gradient descent 와 같은 iterative한 방법을 통해 해를 구하면 계산량 측면에서 더 효율적으로 구할 수 있다.

4.2 정리

데이터의 크기를 어느 정도로 할 것인가에 따라 구별될 뿐 전체적인 흐름은 같다.

Step 1. 회귀계수들을 각각 임의의 값으로 설정하고 첫 손실 함수의 값을 계산한다.

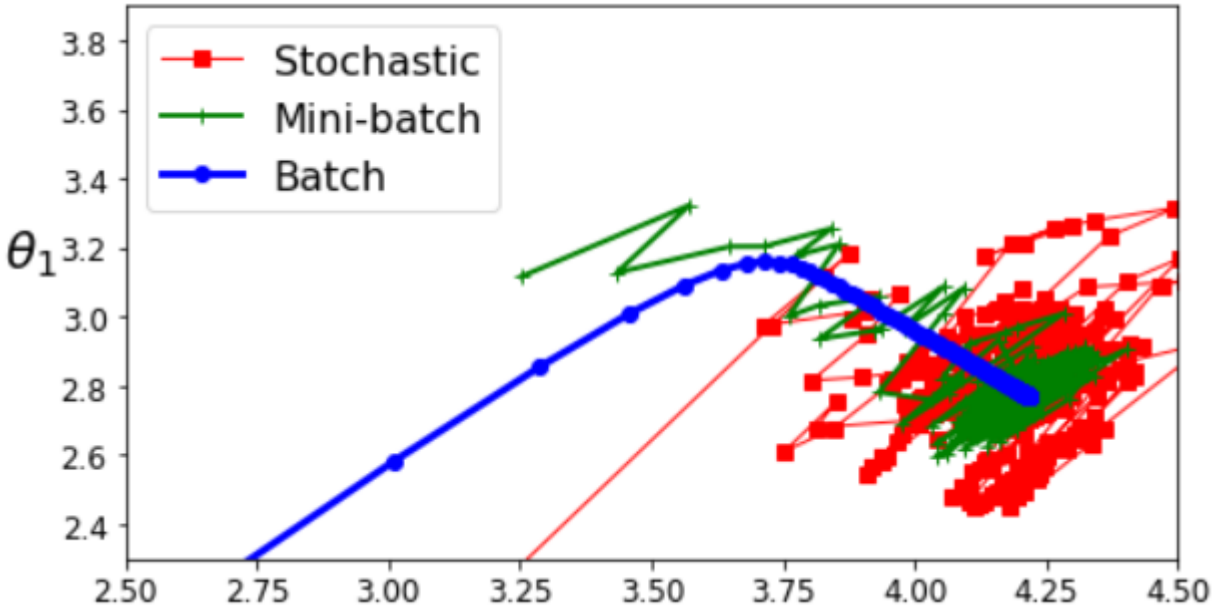
Step 2. w_1 을 이전 $w_1 + \eta \frac{2}{N} \sum_{i=1}^{N} x_i * (\text{실제값}_i - \text{예측값}_i)$), w_0을 이전 $w_0 + \eta \frac{2}{N} \sum_{i=1}^{N} (\text{실제값}_i - \text{예측값}_i)$)으로 업데이트한 후 다시 손실 함수의 값을 계산한다 .

Step 3. 손실 함수의 값이 감소했으면 다시 Step 2를 반복한다.

Step 4. 더 이상 손실 함수의 값이 감소하지 않으면 그때의 w_1, w_0를 구하고 반복을 중지한다.

4.2 정리

경사 하강법 종류별 움직인 경로 비교



배치 경사 하강법은 최솟값에서 멈춘 반면, SGD와 미니배치 경사하강법은 맴돌고 있다.
-> 적절한 학습 스케줄 (learning schedule)을 사용하면 최솟값 도달이 가능하다

4.2 정리

알고리즘별 비교

항목	normal	Batch	SGD	Mini- batch
m이 클때	빠름	느림	빠름	빠름
n이 클때	느림	빠름	빠름	빠름
외부 학습메모리 지원	X	X	O	O
하이퍼파라미터 수	0	2	>=2	>=2
스케일 조정 필요	X	O	O	O
사이킷런	LinearRegression	SGDRegressor	SGDRegressor	SGDRegressor

m : 샘플 수
n : 변수 개수

+ 수학 복습

다항식 polynomial

단항식들의 덧셈과 뺄셈으로 이루어진 식!

Ex $x^2 - 2x + 3$, $4x^3$, $5xy + 6$은 모두 다항식이다.

변수의 개수에 따라

일변수

$$a + bx + cx^2 + \cdots + dx^{n-1} + ex^n$$

다변수

$$x^3y + xy^5 + y$$

4.3 다항회귀

개념 데이터에 잘 맞는 모델의 모양이 '곡선'(=비선형) 일 것 같은 경우 다항식 혹은 곡선을 구해서 학습한다. 함수의 차수가 더 높을수록 더 굴곡이 많은 곡선 형태를 띈다.

종류

속성(x)이 1개
=단일속성다항회귀

2차항 회귀 가설 함수: $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2$

최적값을 찾는다!

3차항 회귀 가설 함수: $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$

최적값을 찾는다!

다중 '선형' 회귀와 모양이 비슷하다 !

3차항 회귀 가설 함수: $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$

똑같이 취급!

다중 선형 회귀 가설 함수: $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3$

단일 속성 다항회귀는 속성 하나의 값을 거듭 제곱하면서 '원하는 차항만큼 속성을 늘리고 새로운 열을 마치 다른 입력변수처럼 취급해 똑같이 다중 선형 회귀' 를 하면 된다.

4.3 다항회귀

속성(xi)이 여러개
=다중다항회귀

가설함수가 2차 함수 (제일 큰 항이 이차항) 이라고 가정해보자

가설함수 = 상수항 + 일차항 + 이차항

변수가 x_1, x_2, x_3일 때 가능한 이차항:

$$x_1x_2 \quad x_2x_3 \quad x_3x_1 \quad x_1^2 \quad x_2^2 \quad x_3^2$$

이차항
굳이 하나의 항이 제곱되어 있는 형태가 아니어도, 2개가 곱해져 있는 꼴은 모두 이차항!

변수가 x1,x2,x3 일 때, 가능한 가설함수 형태

$$h_\theta(x) = \theta_0 + \theta_1x_1 + \theta_2x_2 + \theta_3x_3 + \theta_4x_1x_2 + \theta_5x_1x_3 + \theta_6x_2x_3 + \theta_7x_1^2 + \theta_8x_2^2 + \theta_9x_3^2$$

x_1	x_2	x_3	x_1x_2	x_2x_3	x_3x_1	x_1^2	x_2^2	x_3^2	y
10	1	1	10	1	10	100	1	1	1

다중 다항 회귀도 결국 기존에 있었던 특성 x1, x2, x3를 가지고 새로운 열을 추가해 그냥 입력 변수가 9개인 다중 '선형' 회귀라고 생각할 수 있다.

4.3 다항회귀

다항 회귀의 힘 왜 x1x2 , x^2 의 항처럼, 다항회귀를 사용할까

다항회귀를 사용하면 단순히 복잡한 고차식에 데이터를 맞추는 것을 넘어서 모델의 성능을 극대화 할 수 있다.

집의 크기로 집 가격을 예측해보자고 할 때, 집이 사각형 형태이고 집의 높이와 너비 데이터만 있다고 가정해보자

높이(세로) X1	너비(가로) X2	집값 (y)

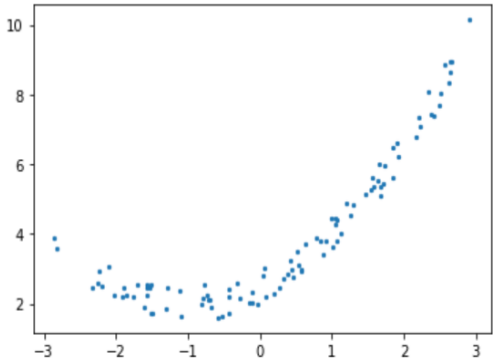
너비가 커도 높이가 작거나, 높이만 크고 너비만 작으면 크기가 작고 구조가 비효율 적이라 집값이 높지 않을 것! 이 두 변수를 각각 보는 것보다 이 둘을 곱한 값, 즉 집의 넓이로 봐야 더 좋다. 단순 선형 회귀를 사용하면 '높이와 너비가 같이 커야지만 집 값도 커진다' 라는 관계를 쉽게 학습할 수 없을 것이다.

속성들을 서로 곱해서 차항을 높이면, 즉 선형 회귀 문제를 다항회귀 문제로 만들어 주면 **속성들 사이에 있을 수 있는 복잡한 관계**들을 프로그램에 학습 시킬 수 있다!

4.3 다항회귀 코드 실습

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

n = 100

x = 6 * np.random.rand(n, 1) - 3
y = 0.5 * x**2 + x + 2 + np.random.rand(n, 1)

plt.scatter(x, y, s=5)

# 위와 같은 데이터는 데이터의 분포가 곡선으로 나타나기 때문에 일반적인 선형회귀로 해결할 수 없다. (비선형)
# 따라서 다항 회귀를 사용한다.
```

Out[1]:
```
<matplotlib.collections.PathCollection at 0x23afe24c
```



In [2]:
```
# 다항 속성을 만들어주는 툴
## 주어진 차수(degree옵션) 까지 특성 간의 모든 교차항을 추가한다.
from sklearn.preprocessing import PolynomialFeatures

# 기본 다항식형태를 만들기. 우리의 데이터를 다항회귀를 위해 가공해준다.
poly_features = PolynomialFeatures(degree=2, include_bias=False)
# 새롭게 정의된 numpy 배열은 행별로 각 데이터를 다항 형태로 변형해준다.
x_poly = poly_features.fit_transform(x) # x_poly는 이제 원래 특성 x와 이 특성의 제곱을 포함한다.
```

- PolynomialFeatures함수를 통해 현재 데이터를 다항식 형태로 변경한다. (각 특성의 제곱 혹은 그 이상을 추가)
- degree 옵션으로 차수를 조절한다.
- include_bias 옵션은 True로 할 경우 0차항(1)도 함께 만든다.

In [3]:
```
x[0], x_poly[0]
```

Out[3]:
```
(array([-1.08553123]), array([-1.08553123,  1.17837804]))
```

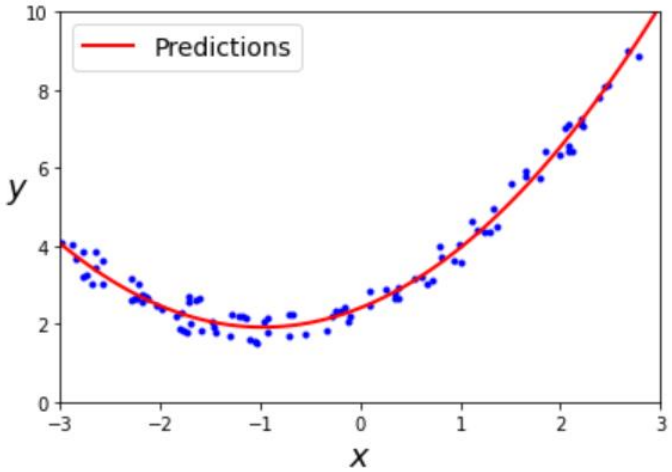
-1.08553123 × -1.08553123 =

1.178378051305313

4.3 다항회귀

In [4]:
```
# 이제 선형회귀의 절차를 동일하게 따라가면 된다.
from sklearn.linear_model import LinearRegression
# model.coef_, model.intercept_
model = LinearRegression()
model.fit(x_poly, y) # 모델에 다항 속성을 추가한 x_poly와 기존 y값을 넣고 학습시킨다.
model.coef_, model.intercept_
```

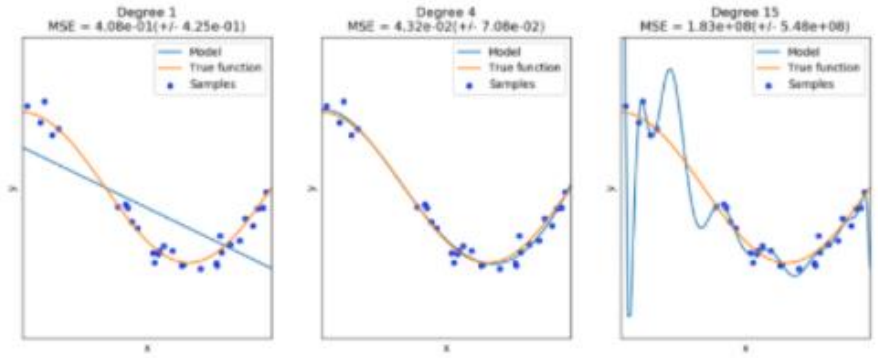
Out[4]:
```
(array([[0.99061269, 0.49513097]]), array([2.54225605]))
```



<- 완성된 회귀식과 기존 데이터(점) 그림

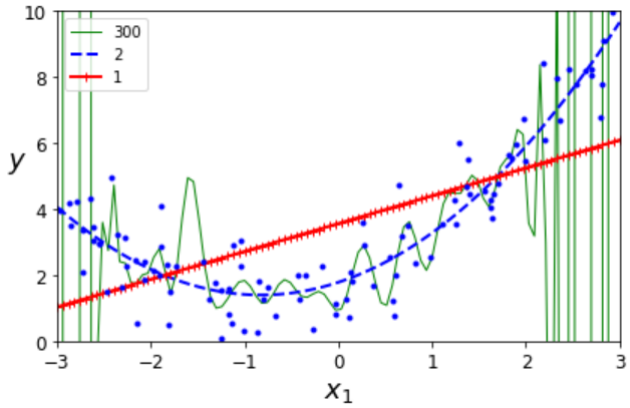
4.4 학습곡선

과소적합 / 과대적합



출처 : Scikit-learn

한번에 그린 그래프



위 사진은 선형모델로써 표현한 underfitting / fit / overfitting 사진 입니다. 언더피팅 모델은 train set의 sample 조차 제대로 모델링 하지 못했고, 오버피팅 모델은 train set의 sample에 지나치게 적합되어서 그 외의 데이터를 표현하지 못합니다. 이런 문제를 일반화가 안되었다고도 합니다.

4.4 학습곡선

얼마나 복잡한 모델을 사용 해야할지 어떻게 결정할 수 있을까? 과대적합, 과소적합을 결정하는 기준은?

학습곡선 데이터에서 훈련, 검증 세트를 추출하고 훈련세트를 하나부터 점차 학습 시키며 (훈련 세트에서 크기가 다른 서브 세트를 만들어 모델을 여러 번 훈련 시킨다) RMSE를 구하여 이를 그래프로 나타낸 것

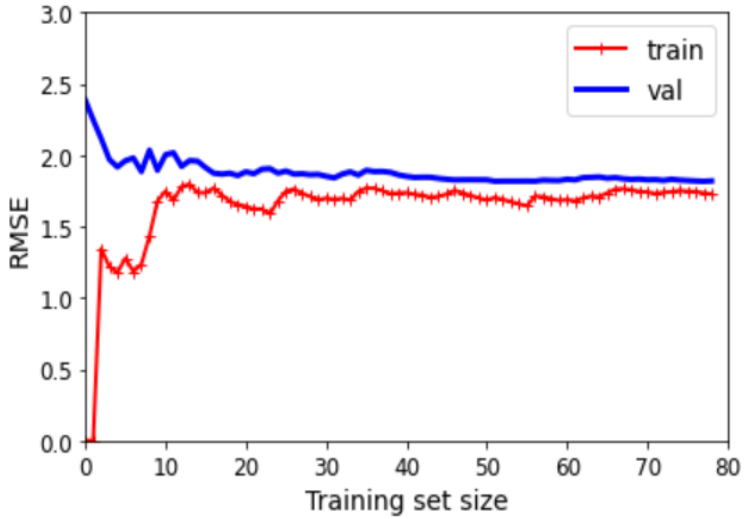
```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

# 학습곡선을 그리기 위한 함수 만들기
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 = [], [] #훈련 데이터의 MSE와 검증 데이터의 MSE 결과를 담을 그릇 생성
    for m in range(1, len(X_train)):
        model.fit(X_train[:m], y_train[:m]) #훈련하기
        y_train_predict = model.predict(X_train[:m]) #예측
        y_val_predict = model.predict(X_val) #예측
        train_errors.append(mean_squared_error(y_train[:m], y_train_predict)) #훈련 데이터의 MSE
        val_errors.append(mean_squared_error(y_val, y_val_predict)) #검증 데이터의 MSE

    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
```

4.4 학습곡선

단순 선형회귀 모델(직선)의 학습곡선

```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg, x, y)
plt.axis([0, 80, 0, 3])
plt.show()
```

그림 저장: underfitting_learning_curves_plot



[훈련 데이터의 성능]

샘플 사이즈가 적을 땐 RMSE가 작으므로 모델이 완벽하게 작동함을 알 수 있다. 하지만 샘플이 추가됨에 따라 잡음도 있고, 원래 데이터 형태가 비선형이기 때문에 모델이 완벽하게 학습하는 것이 불가능해짐에 따라, 오차가 계속 상승한다. 어느정도 샘플 사이즈가 커지면 샘플이 추가되어도 RMSE가 크게 나아지거나 나빠지지 않는다.

[검증 데이터의 성능]

모델이 적은 수의 훈련 샘플로 훈련될 때는, 회귀식이 데이터에 잘 들어맞지 않아 RMSE가 초반에 매우 크다. 점차 샘플 수가 증가하면, 데이터를 잘 반영하게 되면서 학습이 되고, 오차가 천천히 감소하게 된다. 그러나 궁극적으로 데이터 자체는 비선형이기 때문에, 선형 회귀 모델은 데이터를 잘 모델링 하지 못하므로 오차의 감소가 완만해져 훈련 세트의 그래프와 가까워진다. (=과소적합) 모델이 데이터에 과소적합 되어 있다면 훈련 샘플을 더 추가해도 효과가 없다. 더 복잡한 모델을 고려하거나 더 나은 특성을 선택해야 한다.

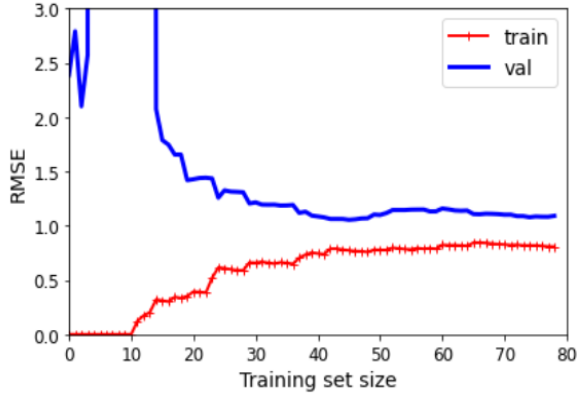
과소적합 된 모델 : 두 곡선이 수평한 구간을 만들고 꽤 높은 오차에서 매우 가까이 근접해 있다.

4.4 학습곡선

10차 다항 회귀모델 (곡선)의 학습곡선

```
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)
plt.axis([0, 80, 0, 3])
plt.show()
```

그림 저장: learning_curves_plot



과대적합 된 모델

[훈련 데이터의 성능]

훈련 데이터의 오차(RMSE)가 선형 회귀 모델보다 훨씬 낮음을 통해, 단순 선형보다 고차항 회귀 모델이 더 적합함을 알 수 있다.

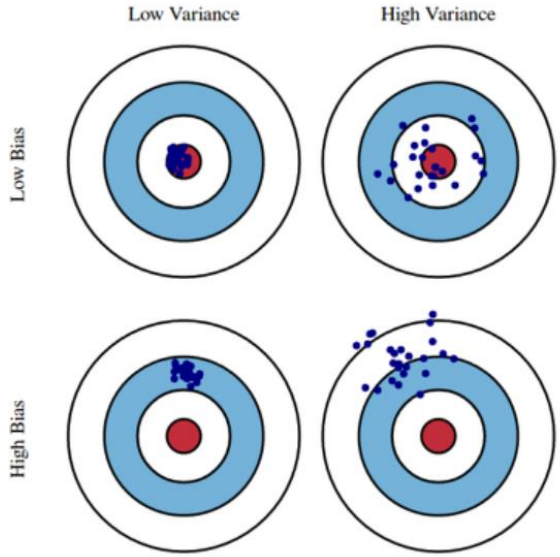
[검증 데이터의 성능]

훈련 데이터와 검증 데이터 곡선 사이에 공간이 있다. 훈련 데이터에서의 모델 성능이 검증 데이터에서보다 훨씬 좋다는 뜻이고, 따라서 이로 미루어 본다면 회귀식이 과대적합 된 것을 알 수 있다. 하지만 더 큰 훈련세트를 사용하게 되면 두 곡선이 점차 가까워지게 되면서 과대적합 문제가 어느정도 해소될 수 있다. 즉, 과대적합 모델을 개선하려면 검증오차가 훈련 오차에 근접할 때 까지 (두 곡선 사이의 공간이 좁혀질 때 까지) 더 많은 훈련데이터를 추가하면 된다.

4.4 학습곡선

[학습 후 예측할 때 생기는 오차의 종류]
① Bias (편향) ② 분산(Variance) => 우리가 줄일 수 있는 영역의 오차
③ 데이터 자체의 잡음 때문에 줄일 수 없는 오차

편향/분산 트레이드 오프

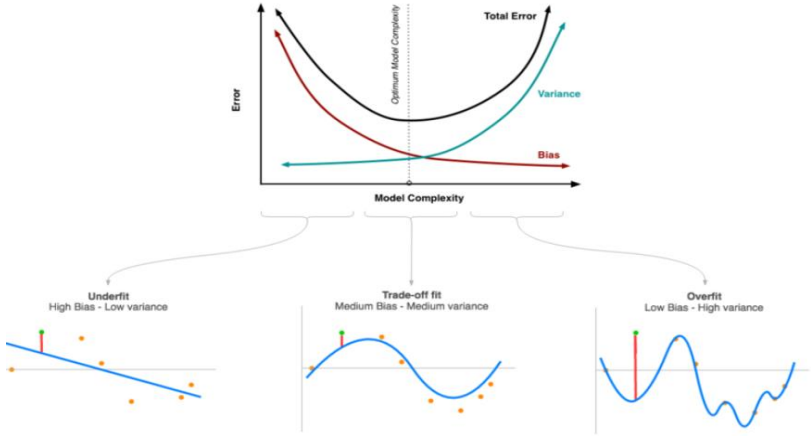


- 빨간색 과녁 = 실제 값
- 파란색 점들 = 회귀 식으로 예측한 값

- Bias : 점들이 빨간색 과녁과 떨어져 있는 정도
- variance : 점들이 퍼져있는 정도

- Bias 에러가 높다 : 많은 데이터를 고려하지 않고 모델이 너무 단순해 정확한 예측을 하지 못하는 경우 (꾸준히 틀리는 상황)
- variance 에러가 높다 : 노이즈까지 전부 학습해, 즉 모델이 너무 복잡해서 약간의 input에도 예측 Y 값이 크게 흔들리는 경우 (사소한 부분까지 학습하여 예측 분포가 큰 상황)

→ 이 두가지 에러는 트레이드 오프 관계가 있어 이 둘을 동시에 낮추는 것은 불가능 따라서 계속 학습시킨다고 해서 전체 에러가 줄어드는 것은 아님. 편향과 분산의 적정한 지점을 찾아 최적의 모델을 만드는 수밖에!



5. 규제 선형 모델

Balance

최적 모델을 위한 Cost 함수 구성요소 = 학습데이터 잔차 오류 최소화 ✚ 회귀계수 크기 제어

비용 함수 목표 = $Min(RSS(W) + alpha * ||W||_2^2)$를 만족하는 벡터 W 찾기

$alpha$: 학습 데이터 적합 정도와 회귀 계수 값 크기 제어

$alpha$ ⇧ 회귀 계수 W 의 값을 작게 해 과적합 개선

L2 규제 = $alpha * ||W||_2^2$와 같이 W의 제곱에 페널티를 부여하는 방식

⇨ 릿지(Ridge) 회귀

L1 규제 = $alpha * ||W||_1$와 같이 W의 절댓값에 대해 페널티를 부여하는 방식

⇨ 라쏘(Lasso) 회귀

5.1 릿지 회귀

릿지 회귀의 비용 함수

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

규제항으로 회귀계수의 제곱 합 대입

α**가 0이면** 선형 회귀와 같아지며
α**가 매우 크면** 모든 가중치는 거의 0에 가까워지고 결국 데이터의 평균을 지나는 수평선이 됨

(β_1, β_2)	$\beta_1^2+\beta_2^2$	MSE
(4,5)	41	20
(3,5)	34	23
(4,4)	32	25
(2,5)	27	27
(2,4)	18	25
(2,3)	13	29

MSE 기준인 일반 선형회귀 모델이라면
β_1과 β_2가 각각 4, 5가 되어야 함

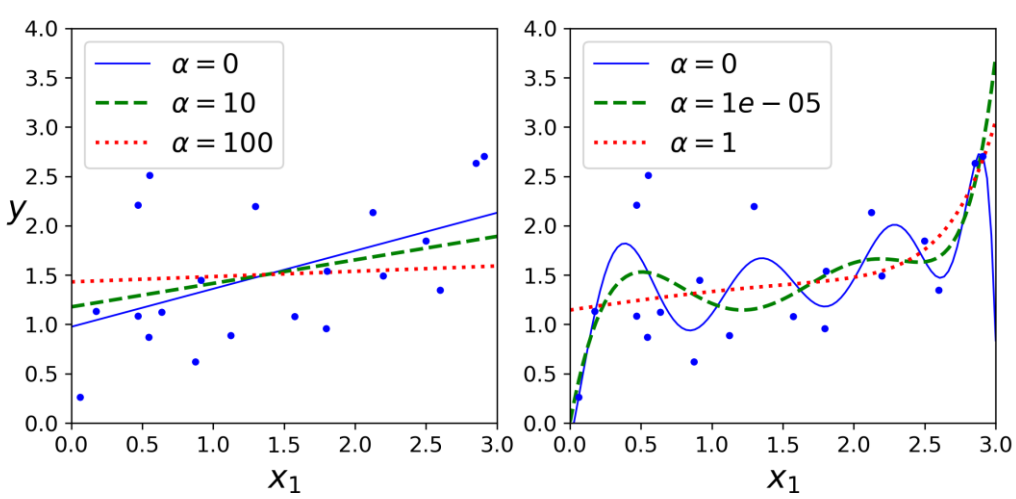
하지만 여기에 ${\beta_1}^2 + {\beta_2}^2$가 30이하여야 한다는 제약이 주어진다면 표 상단의 3가지 경우의 수는 제외되고 나머지 3개 중 MSE가 가장 최소인 (2, 4)가 회귀계수로 결정될 것

출처 : https://ratsgo.github.io/machine%20learning/2017/05/22/RLR/

5.1 릿지 회귀

```
from sklearn.linear_model import Ridge

def plot_model(model_class, polynomial, alphas, **model_kargs):
    for alpha, style in zip(alphas, ("b-", "g--", "r:")):
        model = model_class(alpha, **model_kargs) if alpha > 0 else LinearRegression()
        if polynomial:
            model = Pipeline([
                    ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
                    ("std_scaler", StandardScaler()),
                    ("regul_reg", model),
                ])
        model.fit(X, y)
        y_new_regul = model.predict(X_new)
        lw = 2 if alpha > 0 else 1
        plt.plot(X_new, y_new_regul, style, linewidth=lw, label=r"$\alpha = {}$".format(alpha))
    plt.plot(X, y, "b.", linewidth=3)
    plt.legend(loc="upper left", fontsize=15)
    plt.xlabel("$x_1$", fontsize=18)
    plt.axis([0, 3, 0, 4])

plt.figure(figsize=(8,4))
plt.subplot(121)
plot_model(Ridge, polynomial=False, alphas=(0, 10, 100), random_state=42)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.subplot(122)
plot_model(Ridge, polynomial=True, alphas=(0, 10**-5, 1), random_state=42)

save_fig("ridge_regression_plot")
plt.show()
```

PolynomialFeatures을 사용해 데이터를 확장하고
StandardScaler를 사용해 스케일 조정



α 를 증가시킬수록 직선에 가까워짐

5.1 릿지 회귀

릿지 회귀의 정규방정식

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

```
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]])
```

확률적 경사 하강법

```
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])
```

5.2 라쏘 회귀

라쏘 회귀의 비용 함수

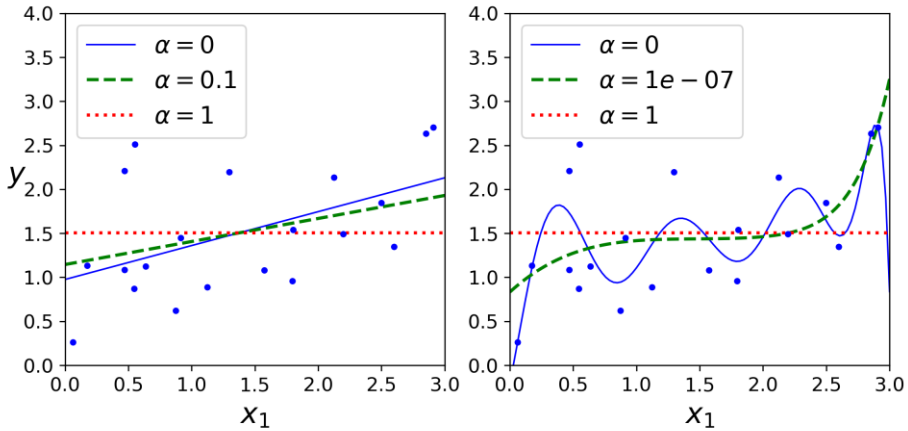
$$J(\theta) = \mathrm{MSE}(\theta) + \alpha \boxed{\sum_{i=1}^{n} |\theta_i|}$$ 규제항으로 회귀계수의 절댓값 합 대입

L2 규제가 회귀 계수의 크기를 감소시키는데 반해

L1 규제는 불필요한 회귀 계수를 급격하게 감소시켜 0으로 만들고 **제거**함

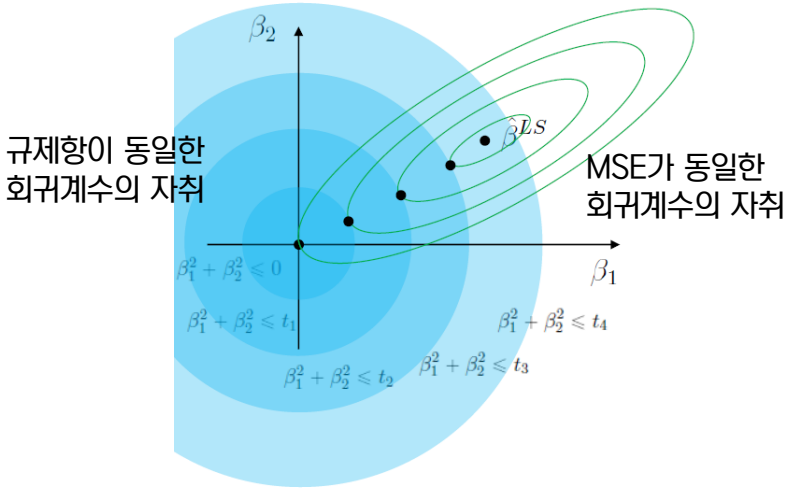
⇨ 적절한 피처만 회귀에 포함시키는 특성 선택을 하고 희소 모델 만듦

```
from sklearn.linear_model import Lasso

plt.figure(figsize=(8,4))
plt.subplot(121)
plot_model(Lasso, polynomial=False, alphas=(0, 0.1, 1), random_state=42)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.subplot(122)
plot_model(Lasso, polynomial=True, alphas=(0, 10**-7, 1), random_state=42)

save_fig("lasso_regression_plot")
plt.show()
```



5.3 기하학적 이해

릿지 회귀



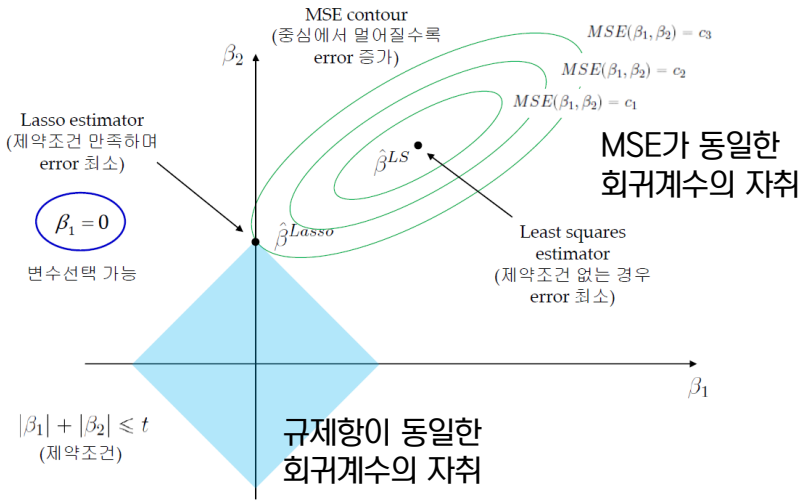
규제항이 동일한
회귀계수의 자취

MSE가 동일한
회귀계수의 자취

β^{LS} : 일반 선형회귀 모델 결과, MSE가 최소가 되는 지점
멀어질수록 MSE가 점점 커짐

원의 반지름이 작아질수록 규제항이 감소하고 제약이 커짐
= α가 클수록 규제항이 작아짐

라쏘 회귀



MSE가 동일한
회귀계수의 자취

규제항이 동일한
회귀계수의 자취

파란색 마름모 꼴의 제약 범위 내에 MSE가 최소인 점은
β_2 축 위의 검정색 점
즉, $\beta_1 = 0$인 지점
= 대응하는 독립변수 x_1이 예측에 중요하지 않음

0에서 미분이 불가능하므로 서브그레디언트 벡터를 사용해
경사 하강법 적용가능

5.4 엘라스틱넷

엘라스틱넷의 비용 함수

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

릿지와 라쏘의 규제항 절충

γ : 혼합비율
$\gamma = 0$ 이면 릿지 회귀, $\gamma = 10$이면 라쏘 회귀

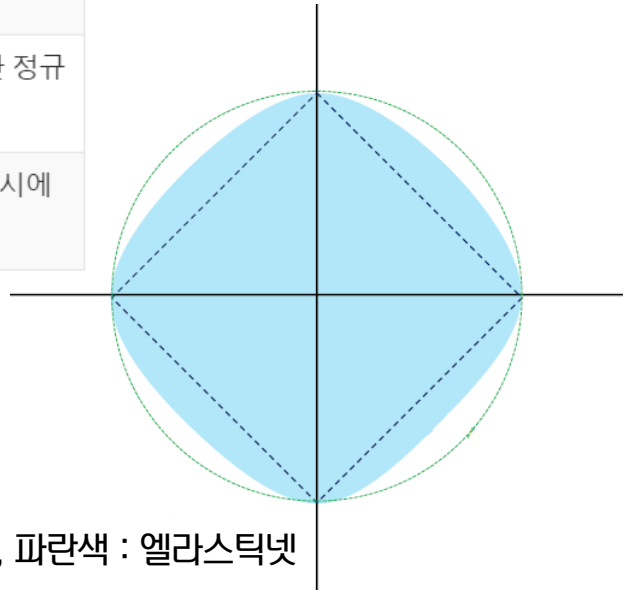
```
from sklearn.linear_model import ElasticNet
elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5, random_state=42)
elastic_net.fit(X, y)
elastic_net.predict([[1.5]])
```

```
array([1.54333232])
```

l1_ratio = γ(혼합비율)

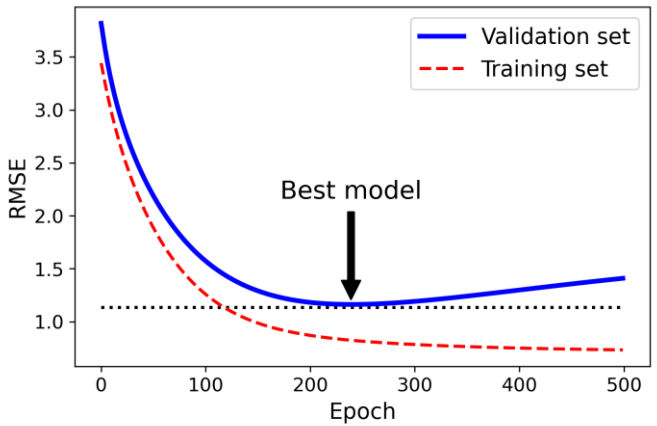
5.5 규제 모델 비교

구분	릿지회귀	라쏘회귀	엘라스틱넷
제약식	L_2 norm	L_1 norm	L_1+L_2 norm
변수선택	불가능	가능	가능
solution	closed form	명시해 없음	명시해 없음
장점	변수간 상관관계가 높아도 좋은 성능	변수간 상관관계가 높으면 성능↓	변수간 상관관계를 반영한 정규화
특징	크기가 큰 변수를 우선적으로 줄임	비중요 변수를 우선적으로 줄임	상관관계가 큰 변수를 동시에 선택/배제



녹색 : 릿지회귀, 검정색 : 라쏘회귀, 파란색 : 엘라스틱넷

5.6 조기 종료



Epoch가 진행됨에 따라 훈련 세트와 검증 세트에 대한 예측 에러(RMSE)가 줄어듦
하지만 특정 시점 이후 검증 에러가 다시 상승
= 과대적합되기 시작하는 것 의미

조기종료 : 검증 에러가 최소에 도달하는 즉시 훈련을 멈추는 것

```
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)
```

```
best_epoch, best_model
```

```
(239,
 SGDRegressor(eta0=0.0005, learning_rate='constant', max_iter=1, penalty=None,
              random_state=42, tol=-inf, warm_start=True))
```

6. 로지스틱 회귀

목적 : 샘플이 특정 클래스에 속할 확률 추정

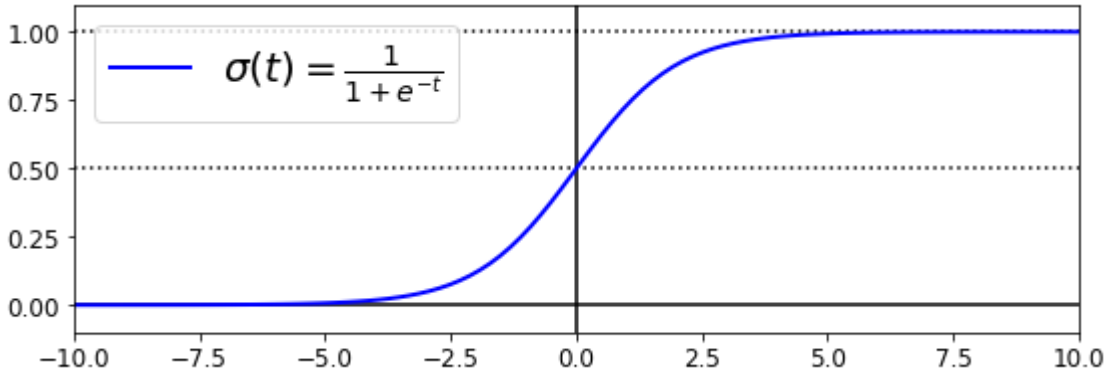
$\hat{p} = \sigma(\theta^T X) = \frac{1}{1+e^{-\theta X}}$: $(-\infty, \infty) \rightarrow (0, 1)$

$\hat{p} > 50\%$: 양성 (label = 1) → $\hat{y} = 1$

$\hat{p} < 50\%$: 음성 (label = 0) → $\hat{y} = 0$

$y^{(i)} \sim Ber(p^i)\ (0 < p^i < 1)$

$P(Y = y) = p^y(1-p)^{1-y}\ (y = 0, 1)$



6. 로지스틱 회귀

비용 함수

$$P(Y=y) = p^y(1-p)^{1-y}\ (y=0,1)$$
$$\rightarrow \log\big(P(Y=y)\big) = ylog(\hat{p}) + (1-y)log(1-\hat{p})\ (y=0,1)$$

<하나의 훈련 샘플에 대한 비용 함수>

평균

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

<로지스틱 회귀의 비용 함수>

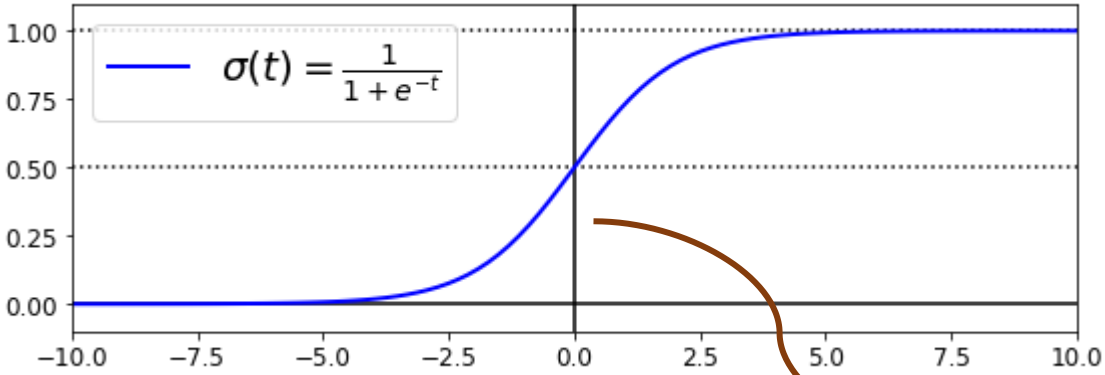
$$J(\theta) = -\frac{1}{m}\sum[y^i \log\big(\hat{p}^i\big) + (1-y^i)\log(1-\hat{p}^i)]$$

편미분

$$\frac{\partial}{\partial\theta_j}J(\theta) = \frac{1}{m}\sum\big(\sigma\big(\theta^T X^i\big) - y^i\big)X_j^i$$

6. 로지스틱 회귀

결정 경계



t = 0일 때를 기준으로 클래스 구분

∴ $\sigma(\theta^T X)$에서 $\theta^T X$=0이 되게 하는 x의 집합 → **결정 경계**

6. 로지스틱 회귀

결정 경계 – 특성이 1개일 때,

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

X=iris['data'][:, 3:]
y=(iris['target']==2).astype(np.int)
```

→ 꽃잎의 너비 (특성)

→ Iris virginica (0이진 분류 기준)

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

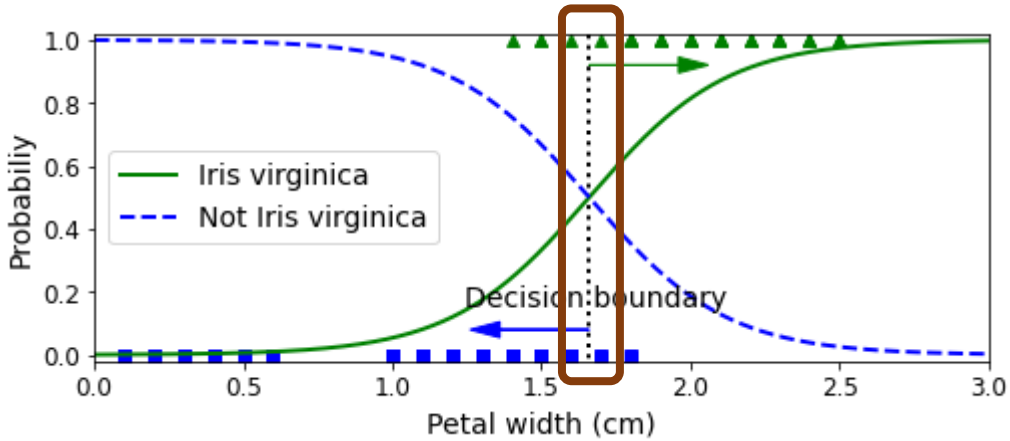
→ 로지스틱 회귀 모델 훈련

6. 로지스틱 회귀

```
X_new=np.linspace(0, 3, 1000).reshape(-1,1)
y_proba=log_reg.predict_proba(X_new)
decision_boundary=X_new[y_proba[:,1]>=0.5][0]
```

→ 결정 경계 ($\hat{p} \geq 0.5$인 가장 작은 X 값)

양성 클래스에 대한 확률



특성이 하나 → 결정 경계를 만드는 X 값 : 1개
∴ 결정 경계가 **축에 평행**하게 나타남

6. 로지스틱 회귀

결정 경계 – 특성이 2개일 때,

```
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(solver='lbfgs', C=10**10, random_state=42)
log_reg.fit(X, y)

x0, x1 = np.meshgrid(
        np.linspace(2.9, 7, 500).reshape(-1, 1),
        np.linspace(0.8, 2.7, 200).reshape(-1, 1),
    )
X_new = np.c_[x0.ravel(), x1.ravel()]

y_proba = log_reg.predict_proba(X_new)
```

특성 2개

Iris virginica (이진 분류 기준)

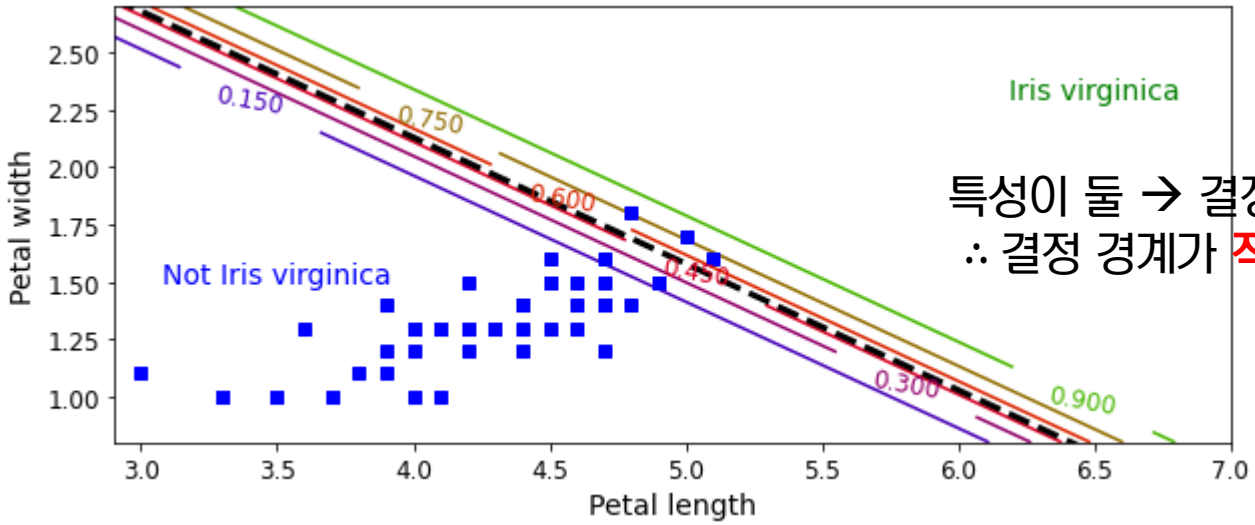
로지스틱 회귀 모델 훈련

축에 대한 point를 입력 받고
point들이 교차하는 좌표를 모두 계산

6. 로지스틱 회귀

```
zz = y_proba[:, 1].reshape(x0.shape)
contour = plt.contour(x0, x1, zz, cmap=plt.cm.brg)


left_right = np.array([2.9, 7])
boundary = -(log_reg.coef_[0][0] * left_right + log_reg.intercept_[0]) / log_reg.coef_[0][1]
```

→ 결정 경계 ($\hat{p} \geq 0.5$인 가장 작은 X 값)



특성이 둘 → 결정 경계를 만드는 X 값 : 2개
∴ 결정 경계가 **직선 방정식**으로 나타남

6. 로지스틱 회귀

소프트맥스 함수

$$\hat{p}_k = \sigma\big(s(\mathrm{X})\big)_k = \frac{\exp(s_{k(\mathrm{X})})}{\sum \exp(s_{j(\mathrm{X})})}, \quad where\ s_{k(x)} = \left(\theta^k\right)^T \boldsymbol{X}$$

$\hat{p}_k$: 샘플이 클래스 k에 속할 추정 확률

K : 클래스 수

$s_{k(x)}$: 샘플 x에 대한 각 클래스의 점수를 담은 벡터

$$\therefore \sum \hat{p}_k = 1 \quad \& \quad \hat{y} = argmax_k \boldsymbol{\sigma}\big(\boldsymbol{s}(\boldsymbol{X})\big)_{\boldsymbol{k}} = argmax_k \boldsymbol{s_k}(\boldsymbol{X}) = argmax_k \left(\boldsymbol{\theta^k}\right)^{\boldsymbol{T}} \boldsymbol{X}$$

주의! 소프트맥스 회귀 분류기는 **한 번에 하나의 클래스만 예측**

∴ **상호 배타적인 클래스**에서만 사용 (여러 사람의 얼굴을 인식하는 용도 X)

6. 로지스틱 회귀

소프트맥스 함수

i번째 샘플이 클래스 k에 속할 확률

<크로스 엔트로피 비용 함수>

$$J(\Theta) = -\frac{1}{m}\sum\sum y_k^i \log(\hat{p}_k^i)$$

클래스 2개일 때, → 로지스틱 회귀의 비용함수와 동일

<클래스 k에 대한 크로스 엔트로피의 그레이디언트 벡터> *(비용 함수 최소화 목적)*

$$\nabla_{\Theta_k} J(\Theta) = \frac{1}{m}\sum \left(\hat{p}_k^i - y_k^i\right) \mathrm{X}^i$$

6. 로지스틱 회귀

소프트맥스 함수

```
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)
```

소프트맥스 회귀

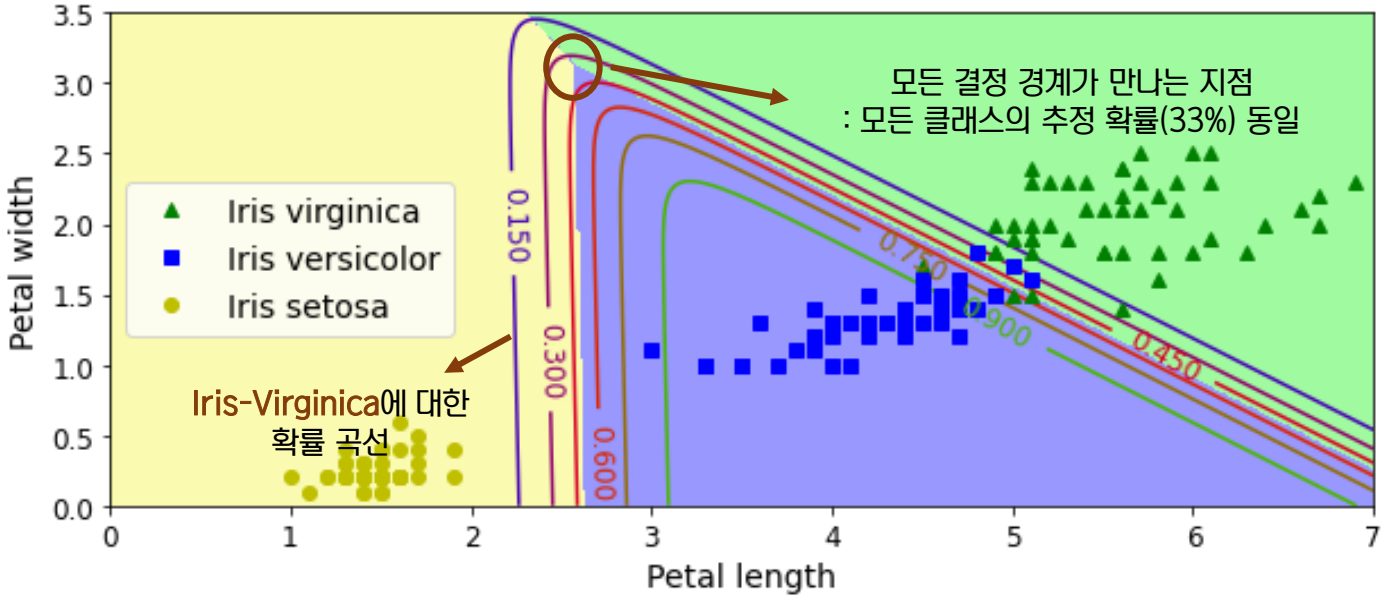
l_2 규제

```
print(softmax_reg.predict([[5, 2]]))
# [2]
print(softmax_reg.predict_proba([[5, 2]]))
# [[6.38014896e-07, 5.74929995e-02, 9.42506362e-01]]
```

최댓값 (index = 2)

6. 로지스틱 회귀

소프트맥스 함수의 결정 경계



감사합니다

Q&A