

# 4. Training models

## 4.1 Linear Regression (선형 회귀)

### 4.1.1 Normal equation (정규방정식)

### 4.1.2 computational complexity (계산복잡도)

## 4.2 Gradient Descent (경사하강법)

### 4.2.1 BGD (Batch Gradient Descent)

### 4.2.2 SGD (Stochastic Gradient Descent)

### 4.2.3 Mini-BGD (Mini-Batch Gradient Descent)

## 4.3 Polynomial Regression (다항 회귀)

## 4.4 Learning curves (학습 곡선)

## 4.5 편향/분산 Tradeoff

## 4.6 Regularized Linear Models

### 4.6.1 Ridge Regression (=L2 Regression)

### 4.6.2 Lasso Regression (=L1 Regression)

### 4.6.3 Elastic Net

+. 규제 summary

## 4.7 Early Stopping

## 4.8 Logistic Regression

### 4.8.1 확률 추정

### 4.8.2 훈련과 비용 함수

### 4.8.3 결정 경계

### 4.8.4 softmax regression (= Multinomial Logistic Regression)

## 4.1 Linear Regression (선형 회귀)

예측값

식 4-1 선형 회귀 모델의 예측

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

특성 벡터

•  $\hat{y}$ 는 예측값입니다.

파라미터 벡터  $\theta$ . 편향(절편).

파라미터 벡터

식 4-2 선형 회귀 모델의 예측(벡터 형태)

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

가설함수. 특성벡터 ( $x_0 \sim x_n$ )

$x_0 = 1$  (벡터 형태로 나타내기 위한 가정.)

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

이게 맞음. 전치일 때는 “행렬곱”

- 특성 벡터 = feature vector
- dot product 함.
- 훈련한다 = 모델이 훈련 세트에 잘 맞도록 ‘모델 파라미터( $\theta$ )를 설정’한다.
- 평균 제곱근 오차(RMSE = Root Mean Square Error)

$$RMSE = \sqrt{MSE}$$

- 선형 회귀 모델을 훈련 시키기 위해서는 이 값을 최소화하는  $\theta$  찾아야.
- MSE가 더 간단함 (root 안하니까 연산량 줄음)

#### 식 4-3 선형 회귀 모델의 MSE 비용 함수

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

### 4.1.1 Normal equation (정규방정식)

- MSE를 최소화하는  $\theta$ 를 찾는 수학 공식 = 정규방정식

#### 식 4-4 정규방정식

$$\hat{\theta} = (\mathbf{X}^T \cdot \mathbf{X})^{-1} \cdot \mathbf{X}^T \cdot \underline{\mathbf{y}} \quad \text{※증명 복잡하니.}$$

- $\hat{\theta}$ 는 비용 함수를 최소화하는  $\theta$  값입니다.
- $\mathbf{y}$ 는  $y^{(1)}$ 부터  $y^{(m)}$ 까지 포함하는 타깃 벡터입니다.

- 역행렬: np.linalg의 inv() 함수
- 행렬 곱셈: dot()

```
import numpy as np

X = 2 * np.random.rand(100, 1)
y = 4 + 3*X + np.random.randn(100, 1) # 가우시안 노이즈를 더함

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

theta_best
# array([[4.2], [2.7]])
# array([[편향(intercept)], [기울기]])
```

```
# 세타를 이용해 예측하기
```

```
X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2, 1)), X_new] # 모든 샘플에 x0=1을 추가
y_predict = X_new_b.dot(theta_best)
y_predict
# array([[4.2], [9.7]])
```

```
# 같은 작업을 하는 sklearn code
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X, y)
```

```
# sklearn은 특성의 가중치와 편향을 분리하여 저장함
lin_reg.intercept_, lin_reg.coef_
# array([4.2]), array([2.7])
lin_reg.predict(X_new)
# array([4.2], [9.7])
```

## 4.1.2 computational complexity (계산복잡도)

- 정규방정식은  $(n+1) \times (n+1)$  크기가 되는  $X^T * X$ 의 역행렬을 계산함. ( $n$ 은 특성의 수)
- 역행렬을 계산하는 계산복잡도: 일반적으로  $O(n^{2.4}) \sim O(n^3)$  사이.
- 특성 수가 2배로 늘어나면 계산 시간이 대략  $2^{2.4}$ 에서  $2^3$ 배로 증가함

---

- 훈련 세트의 샘플 수(인스턴스 수)에는 ‘선형적’으로 증가. ( $O(m)$ ,  $m$ 은 샘플 수)
  - 샘플 2배, 걸리는 시간 2배
- 선형 회귀 모델은 예측이 매우 빠름.

---

- 따라서 예측 계산 복잡도는 샘플 수, 특성 수에 선형적임.

## 4.2 Gradient Descent (경사하강법)

- 여러 종류의 문제에서 최적의 해법을 찾을 수 있는 매우 일반적인 최적화 알고리즘
- 특성 수가 클 때 linear regression 학습에 좋음

---

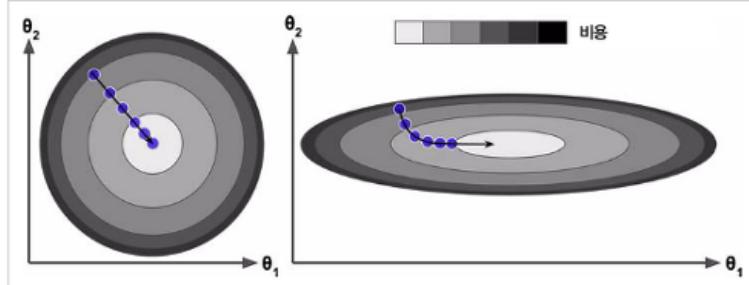
- error (function) 오차함수 = prediction - label
- loss (function) 손실함수 = 하나의 데이터 포인트 또는 샘플에 대한 Error
- cost (function) 비용함수 = 전체 데이터 세트 또는 미니배치(batch) 데이터에 대한 평균 Loss

---

- $\theta$ 를 무작위 초기화 후 MSE가 감소되는 방향으로 진행하여 loss가 global minimum에 도달하도록 함.
- 학습률 (learning rate): 스텝의 크기
  - 너무 작으면 ‘local minimum’에 수렴할 수도
- MSE cost function: Convex Function(볼록 함수)
  - GD가 global minimum에 가깝게 접근함을 보장함.

- 경사하강법을 사용할 때는 “모든 특성이 같은 스케일을 갖도록” 해야함.
  - 수렴하는데 오래걸리기 때문

그림 4-7 특성 스케일에 따른 경사 하강법



- sklearn → StandardScaler (normalization 과정)

### 4.2.1 BGD (Batch Gradient Descent)

식 4-5 비용 함수의 편도함수<sup>14</sup>

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

편도함수 :  $\theta_j$ 가 조금 변경될 때 비용 함수가 얼마나 바뀌는가

- 경사하강법 구현을 위해 각 모델 파라미터  $\theta_j$ 에 대해 비용 함수의 기울기를 계산해야하기 때문
- 위 식으로 (편도함수를 각각 계산하는 대신) ‘한꺼번에’ 계산하는거임.
  - 비용함수의 편도함수를 모두 담고 있는거지.

식 4-6 비용 함수의 그래디언트 벡터

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

- **Batch Gradient Descent** : gradient descent STEP에서 전체 훈련 세트 X에 대해 편도함수를 계산. (느림, 정규방정식보단 빠름)

#### 식 4-7 경사 하강법의 스텝

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

$\eta$  : 학습률

```

eta = 0.1
n_iterations = 1000
m = 100

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

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

theta
# array([[4.2], [2.7]])

```

- 학습률 너무 높으면 발산, 너무 낮으면 엄청 오래걸려서 최적점에 도달.
- 적절한 학습률 찾으려면 → ‘그리드 탐색 (grid search)’
  - 반복횟수 어떻게 정함?
  - 반복 횟수를 아주 크게 정하고, gradient가  $\epsilon$  (허용오차=tolerance)보다 작아지면 (경사하강법이 거의 최솟값에 도달한 것으로) 알고리즘을 중지

## 4.2.2 SGD (Stochastic Gradient Descent)

- 확률적 = 무작위
- 매 스텝에서 딱 하나의 샘플만 무작위로 선택해서 gradient 계산
  - 적은 데이터 처리로 빠름.
  - 매 반복마다 한 샘플만 메모리에 있으면 됨 - 매우 큰 훈련 세트도 훈련 가능
- 무작위성의 딜레마 → 지역 최솟값 탈출, but 전역 최솟값에 다다르지 못할 수 있음

- 학습률을 점진적으로 감소시키자. (시작할 때는 빠르게 학습, 점점 조금씩 수렴 → 결국 전역 최솟값 도달)
- 학습 스케줄(learning schedule) : 매 반복에서 학습률을 결정하는 함수
  - 너무 빨리 줄어들면 지역 최솟값에 갇히거나, 중간에 멈춤
  - 너무 천천히 줄어들면 최솟값 주변을 맴돌거나, 훈련을 너무 일찍 중지함 (지역에 머무름)

---

```

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

def learning_schedule(t):
    return t0 / (t + t1) // 학습률을 시간 지속으로 각아짐.

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

for epoch in range(n_epochs): // 몇 번째 epoch?
    for i in range(m): // m번 반복
        random_index = np.random.randint(m) // 해당 index.
        xi = X_b[random_index:random_index+1] // 하나만 뽑기.
        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
    ↗ 2 번째 대안
    ↗ 가능성을 확장하기.
  
```

- 에포크: 전체 데이터셋을 훈련하는 횟수
- 샘플을 무작위로 선택하기 때문에, 어떤 샘플은 한 에포크에서 여러번 선택 될 수 있고, 어떤 샘플은 전혀 선택되지 못할 수도 있음.
  - 에포크마다 모든 샘플을 쓰게 하고 싶으면 → 에포크마다 섞으면 됨.

---

```

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

```

규제X.

---

여기에서도 정규방정식으로 구한 것과 매우 비슷한 값을 얻었습니다.

---

```

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

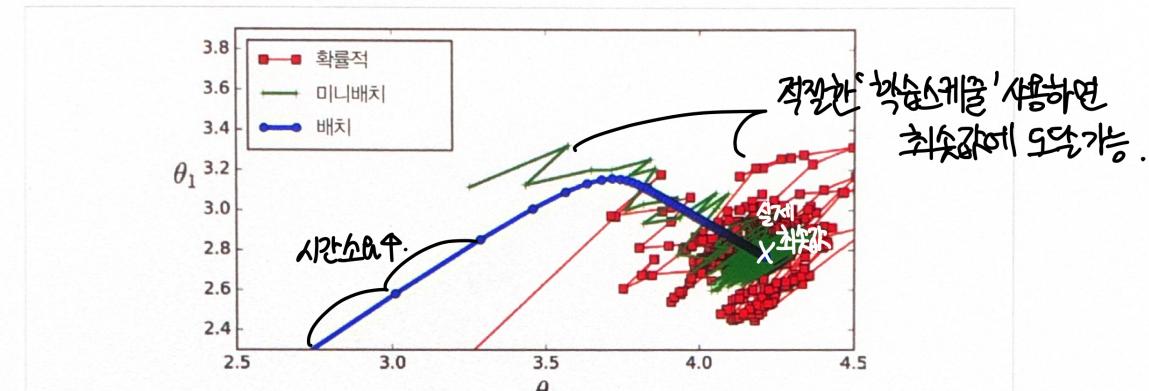
```

---

### 4.2.3 Mini-BGD (Mini-Batch Gradient Descent)

- 미니배치라고 부르는 임의의 작은 샘플 세트에 대해 gradient를 계산함.
- 행렬 연산에 최적화된 하드웨어(GPU)를 사용하여 성능 향상 가능
- BGD → SGD 일수록 지역 최솟값에서 탈출하기 쉬움.

그림 4-11 파라미터 공간에 표시된 경사 하강법의 경로



### 4.3 Polynomial Regression (다항 회귀)

- 데이터가 복잡한 형태 → 각 특성의 거듭제곱을 새로운 특성으로 추가하여 학습 : 다항 회귀
- 2차 방정식(quadratic equation)

---

```

>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolyomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929, 0.56664654])

```

---

```

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

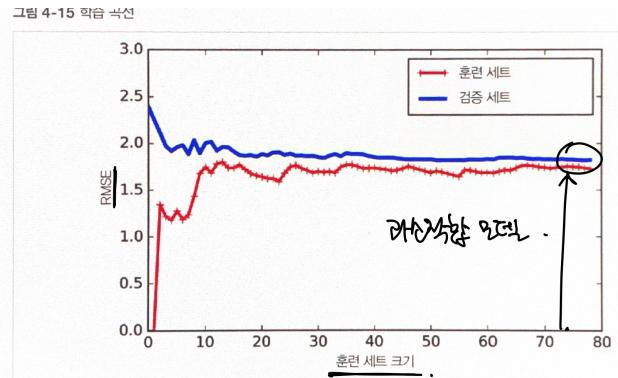
```

y 값

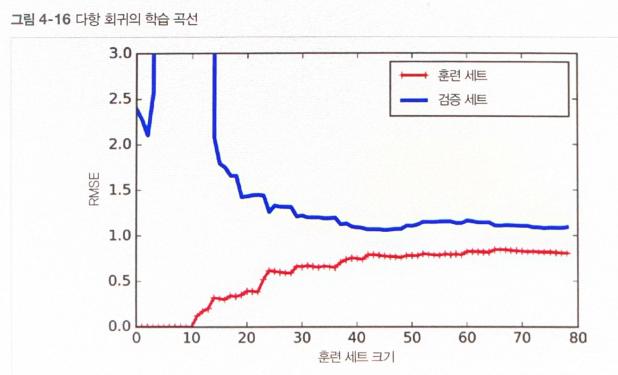
- 다항 회귀는 특성 사이의 관계를 찾을 수 있음 ← PolnimialFeatures가 주어진 차수까지 특성 간의 모든 교차항을 추가하기 때문.
  - ex. a, b인데 degree=3일 때, ab, a^2b, ab^2 같은것도 특성으로 추가하기 때문.

## 4.4 Learning curves (학습 곡선)

- 모델의 과대적합, 과소적합을 어떻게 알 수 있을까?
  - 교차 검증: 훈련 데이터에서의 성능이 좋지만, 교차 검증 점수가 나쁘면 과대적합 된 것 반면 양쪽 모두 좋지 않으면 과소적합.
  - 학습 곡선: 훈련과 검증의 그래프가 가까우면 과소적합, 멀면 과대적합
    - LinearRegression()
      - 과소적합 개선: 더 복잡한 모델 사용 (예를들어 1차가 안되면 2차로, 3차로, ...)
      - 훈련 샘플을 더 추가해봤자 개선되지 않음.



- b. `polynomial_regression = Pipeline(poly_features(degree=10) + lin_reg)`
- 과대적합 개선: 훈련 데이터 추가 (검증 오차가 훈련 오차에 근접할 때까지)



- 훈련 데이터의 오차가 훨씬 낮다.

## 4.5 편향/분산 Tradeoff

- model complexity를 늘리면, variance가 올라가고, bias는 낮아진다. (Overfit)
  - 높은 bias (실제 데이터와 얼마나 일치하는지에 대한 편견)
    - 과소적합
  - 높은 분산 (학습 데이터의 작은 변화에 민감)
    - 과대적합

## 4.6 Regularized Linear Models

- 규제: 과대적합 방지
  - 선형 모델의 '가중치 제한'

1. Ridge Regression
2. Lasso Regression
3. Elastic Net

### 4.6.1 Ridge Regression (=L2 Regression)

- 원래 cost function에 정규화항을 추가. (가중치 제곱 합에 비례)

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

- 알파: 정규화 정도 (0이면 정규화 X)
- closed form

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

- A:  $(n+1)(n+1)$  크기의 identity matrix(항등 행렬 = 단위 행렬)
  - identity matrix: 주 대각선(diagonal) 상의 모든 원소가 1이고, 나머지 원소는 모두 0인 정사각 행렬



이제 이마 으로 알고 싶다. identity matrix는 단위 행렬이다.

- SGD에서는 penalty를 l2로 지정해서 ridge regularization된 linear model로 사용.

```
>>> sgd_reg = SGDRegressor(penalty="l2")
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
array([1.47012588])
```

- l2 norm을 cost function에 더해준다는 뜻

$$\frac{1}{2} (\| \mathbf{w} \|_2)^2$$

l2 norm

## 4.6.2 Lasso Regression (=L1 Regression)

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

- 덜 중요한 특성의 가중치를 제거.
  - sparse model을 만드는거임 (0이 아닌 특성의 가중치가 적다)

불필요한 특성을 제거한다는게, 예를 들어서 polynomial model의 차수가 2차원이라면,  $y = ax^2 + bx + c$ 잖아? 그럼 만약 여기서  $x^2$  특성이 중요하지 않다면  $a$ 를 0으로 만든다는 얘기야?

맞아, 맞는 말이야! 예를 들어, 라쏘 회귀를 사용할 때 특성의 가중치를 0으로 만든다면 해당 특성은 모델에 더 이상 기여하지 않게 되어. 따라서, 특성의 중요성에 따라 라쏘 회귀는 일종의 특성 선택 기능을 수행하게 돼. 만약 특정 특성이 중요하지 않다면, 라쏘 회귀는 해당 특성의 가중치를 0으로 만들고, 이는 모델이 해당 특성을 무시하게 만들어 효과적인 특성 선택을 수행하는 것이다.

내가 말한 예시가 맞는 이유를 자세히 설명해봐

물론이야! 예를 들어, 다음과 같은 2차 다항식 회귀 모델을 생각해보자:

$$y = ax^2 + bx + c$$

여기서  $x^2$ 은 2차 다항식의 제곱 항을 나타냅니다. 만약 라쏘 회귀를 사용하여 이 모델을 훈련시키고, 특성 선택을 수행한다면, 라쏘는 가중치  $a$ 를 0으로 만들 수 있어. 이것은  $x^2$  특성이 모델에 미치는 영향을 없애는 것이고, 결과적으로 모델은 다음과 같이 단순화될 수 있어:

$$y = bx + c$$

즉,  $x^2$  특성이 중요하지 않다고 판단되면 라쏘 회귀는 해당 특성의 가중치를 0으로 만들어 제거하고, 모델을 더 간단하게 만드는 효과가 있어. 이는 모델의 해석이나 과적합 방지에 도움이 될 수 있어.

- 알파값을 적게 설정해도 규제 빽세게 됨

### 4.6.3 Elastic Net

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

- 중간정도.
- $r : 0$ 이면 ridge,  $1$ 이면 Lasso가 됨.
- ElasticNet(alpha=0.1, l1\_ratio=0)

#### + 규제 summary

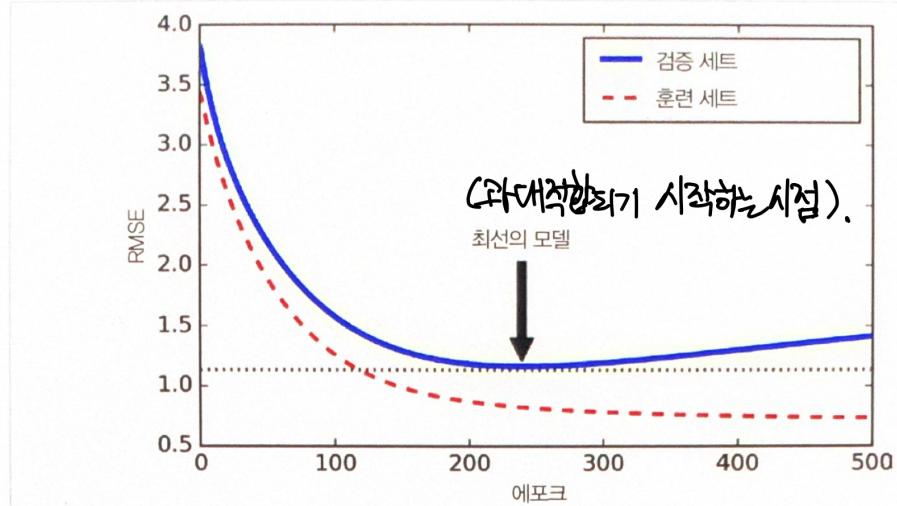
- 거의 대부분 상황에서 사용하는게 좋음 (preferable)
- ridge가 일반적으로 좋음
- 특정한 feature가 중요하면, lasso나 elastic net이 나음.
  - 일반적으로는 elastic net이 나음.

- lasso: 입력 특성 > 학습 특성 → 결과가 이상하게 나옴
- 몇 가지 특성이 strongly correlate 되어있는 경우

## 4.7 Early Stopping

- 과대적합을 막는 방법
  - 검증 에러가 다시 상승하기 시작하는 시점에서 훈련 멈추기.

그림 4-20 조기 종료 규제



## 4.8 Logistic Regression

### 4.8.1 확률 추정

- 샘플이 특정 클래스에 속할 확률을 추정함. (50프로 넘으면 속함, 아니면 안 속함 → 이진 분류기)

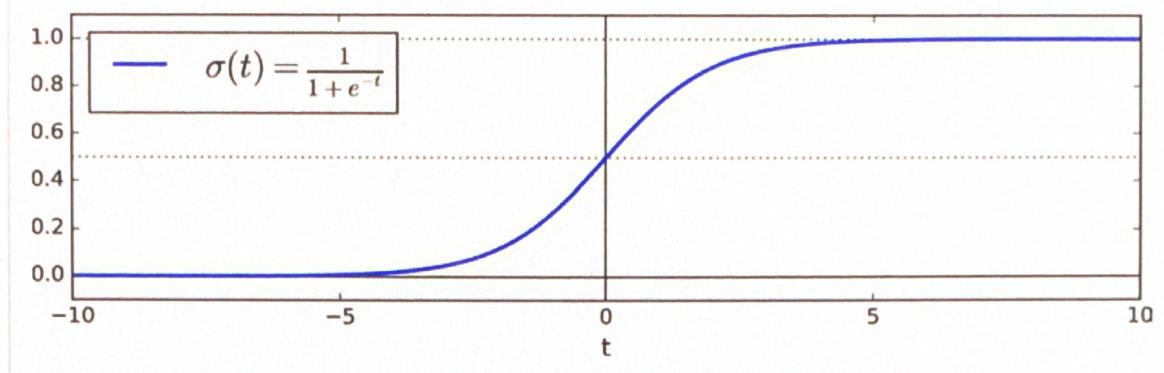
식 4-13 로지스틱 회귀 모델의 확률 추정(벡터 표현식)

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

식 4-14 로지스틱 함수

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

그림 4-21 로지스틱 함수



식 4-15 로지스틱 회귀 모델 예측

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

## 4.8.2 훈련과 비용 함수

- 훈련의 목적: 양성 샘플( $y=1$ )에 대해서는 높은 확률을 추정, 음성 샘플( $y=0$ )에 대해서는 낮은 확률을 추정하는 - 모델 파라미터 벡터 '세타'를 찾는 것.

식 4-16 하나의 훈련 샘플에 대한 비용 함수

$$c(\theta) = \begin{cases} -\log(\hat{p}) & y = 1 \text{일 때} \\ -\log(1 - \hat{p}) & y = 0 \text{일 때} \end{cases} \quad \begin{array}{l} \text{양성 샘플 (높은 확률)} \\ \text{음성 샘플 (낮은 확률)} \end{array}.$$

- 전체 훈련 세트에 대한 비용 함수 = 모든 훈련 샘플의 비용을 평균한 것. → log loss(로그 손실)

식 4-17 로지스틱 회귀의 비용 함수(로그 손실)

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



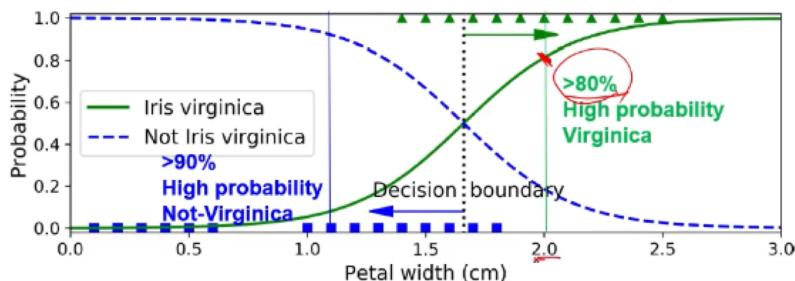
#### 식 4-18 로지스틱 비용 함수의 편도함수

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

?????

### 4.8.3 결정 경계

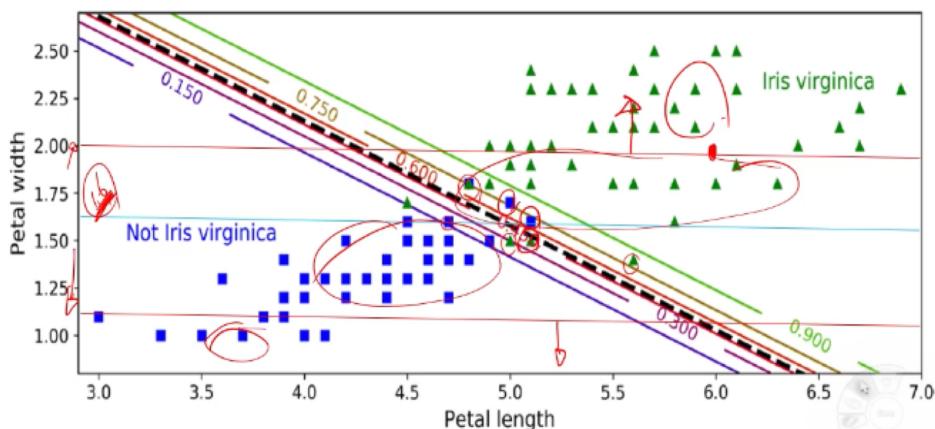
- decision boundary 이동 시, 입력 인스턴스의 판정 결과가 달라짐. (Tradeoff 상황 발생)



이동된 초록색 세로선: virginica로 판명될 확률 80프로 이상 (not-virginica가 절대 virginica로 판명되지 않음) ↗ 모든 virginica를 virginica로 판명. (파란색 세로선)

→ log\_reg.predict([1.7], [1.5]): array([1, 0]) // 1.7이 들어오면 1(positive), 0이 들어오면 0(negative)로 판정됨

- 아래는 input feature를 한 개 했을 때보다 두개 했을 때가 더 오판단이 적은 이유이다.



가로선 - input feature 하나인 경우. 예를 들어 위쪽 빨간색이 decision boundary라고 한다면, 초록색 아래부분이 virginica인데 not virginica라고 판단되는 세모점들이 너무 많음.

### 4.8.4 softmax regression (= Multinomial Logistic Regression)

- 다중 분류: 이진 분류기 연결하지 않아도 직접 다중 클래스 지원 가능

- 다중 클래스이지, 다중 출력이 아님. (한 번에 하나의 클래스만 예측)

식 4-19 클래스  $k$ 에 대한 소프트맥스 점수

$$s_k(\mathbf{x}) = (\boldsymbol{\theta}^{(k)})^T \cdot \mathbf{x} \xrightarrow{\text{Softmax func.}} \text{각 클래스의 확률.}$$

- $\boldsymbol{\theta}$ 는 각 클래스 자신만의 파라미터 벡터 =  $\boldsymbol{\Theta}$ (파라미터 행렬)에 저장됨.

식 4-20 소프트맥스 함수

$$\hat{p}_k = \sigma(s(\mathbf{x}))_k = \frac{\exp(s_k(\mathbf{x}))}{\sum_{j=1}^K \exp(s_j(\mathbf{x}))} \quad * \text{ 확률이 가장 높은 class를 선택.}$$

클래스 수.

p값 모두 더하면 1임 (각각 클래스에 속할 확률들의 합이니까)

식 4-21 소프트맥스 회귀 분류기의 예측

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

- 훈련 방법: 모델이 target class에 대해서는 높은 확률을 추정하도록 만드는 것이 목적.
  - 크로스 엔트로피(cross entropy) 함수를 최소화하기 (= 낮은 확률을 예측하는 모델을 억제)
    - 클래스가 2개이면 로그 손실이랑 똑같음.

식 4-22 크로스 엔트로피 비용 함수

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

클래스 개수.

- $i$  번째 샘플에 대한 타깃 클래스가  $k$  일 때  $y_k^{(i)}$  가 1이고, 그 외에는 0입니다.

#### 식 4-23 클래스 $k$ 에 대한 크로스 엔트로피의 그래디언트 벡터

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

- gradient vector를 계산할 수 있으므로 '경사 하강법'을 사용 가능

그림 4-25 소프트맥스 회귀 결정 경계

