

Machine learning

Logistic Regression

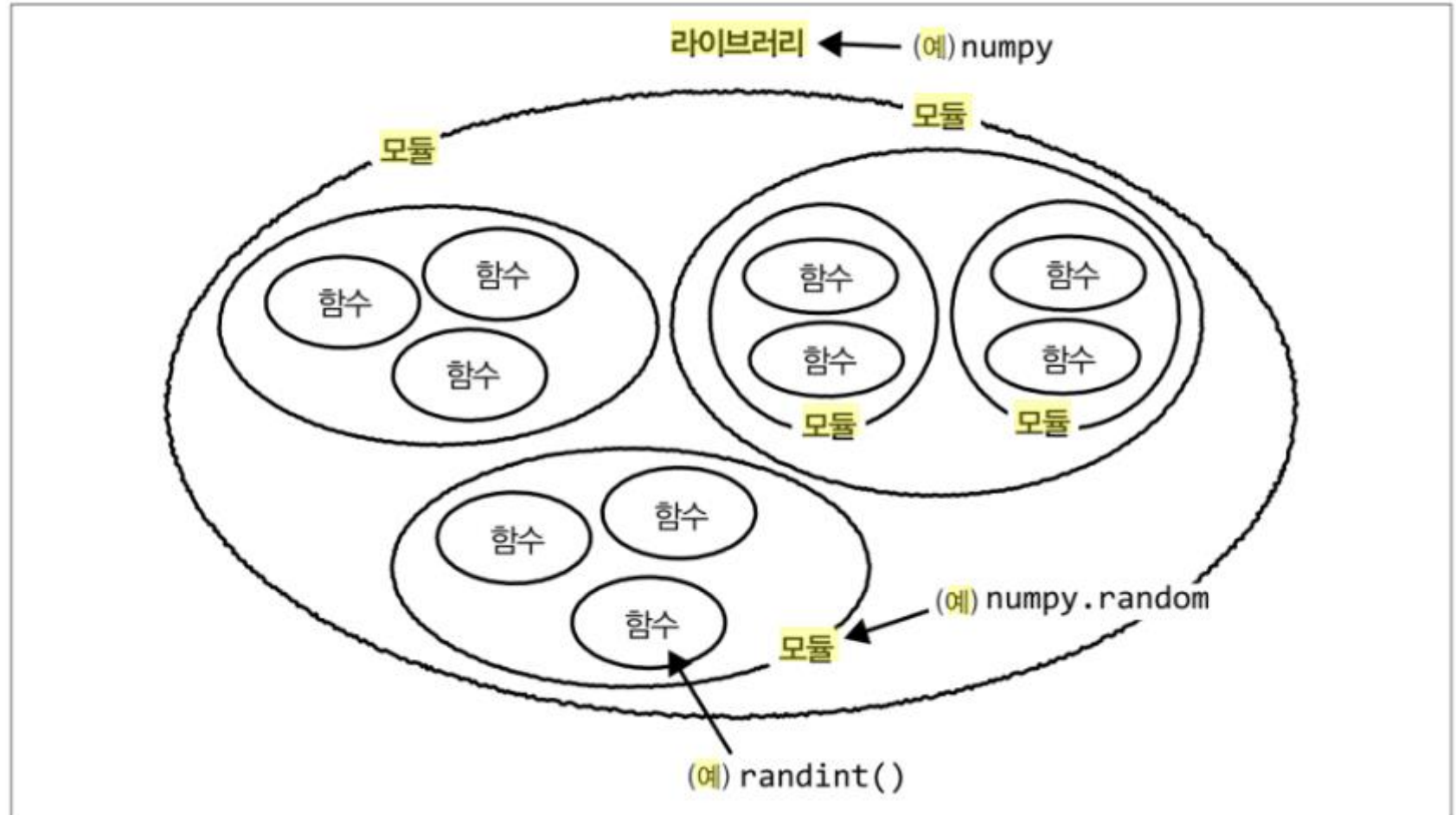
Logo

Python Code

Python 용어 정리 !

Module
Library
Package
Framework
Class

그림 7-1 라이브러리



출처: 파이썬으로 배우는 딥러닝 교과서, 한빛미디어, 이시카와 아키히코

Logistic Regression 로지스틱 회귀 코드로 돌아오면

```
def __init__(self, eta=0.05, n_iter=100, random_state=1):  
    self.eta = eta  
    self.n_iter = n_iter  
    self.random_state = random_state
```

```
def fit(self, X, y):  
    rgen = np.random.RandomState(self.random_state)
```



Odds Ratio = 오즈비

- 오즈비 = Odds / Odds

- Odds = 불균형 빈도비 = 불균형 확률비

= 고양이3 / 개7 (한 마리씩 나타나는데, 두가지 펫만 있다면)

$$= a / b = 3 / 7$$

$$= \{ a / (a+b) \} / \{ b / (a+b) \} = 0.3 / 0.7$$

고양이	$P(X)$	$1 - P(X)$	고양이 제외
	일어날 확률	일어나지 않을 확률	
	$0 \leq P(X) \leq 1$		
			$\frac{P(X)}{1 - P(X)}$
			= 개

Odds Ratio = 오즈비 = 승산?비 (승리의 발견율 비)

- 특정 **조건**의 승산(승리발견율) / 다른 조건에서 승산(승리발견율)의 비율

예) 클럽o, 확진자 확률 Odds 0.01/0.99

$$\frac{P(X)}{1 - P(X)} \quad \text{Odds (클럽o 조건)}$$

클럽x, 확진자 확률 Odds 0.04/0.96

$$\frac{P(X)}{1 - P(X)} \quad \text{Odds (클럽x 조건)}$$

$$\text{Odds Ratio} = \text{OR} = \frac{P(\text{disease} | \text{exposed}) / [1 - P(\text{disease} | \text{exposed})]}{P(\text{disease} | \text{unexposed}) / [1 - P(\text{disease} | \text{unexposed})]}$$

Odds Ratio = 오즈비 = 승산?비 (승리의 발견율 비)

- 특정 **조건**의 승산(승리발견율) / 다른 조건에서 승산(승리발견율)의 비율

[1] 환자-대조군 연구

이미 **질환이 발생한** 환자군과 질환이 발생하지 않은 대조군을 모집 후,
위험인자 노출 여부(특정 시점에서의 결과)를 **후향적**으로 조사하여 위험인자와 질환 발생 간의 연관성 추정.

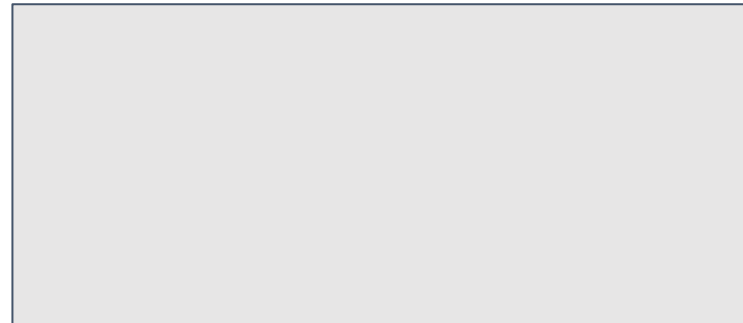
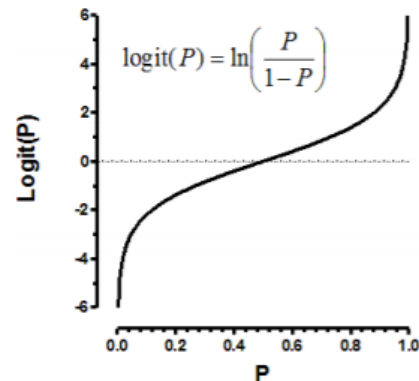
이러한 경우에는 위험인자에 노출된 전체 모집단과 노출되지 않은 전체 모집단을 파악할 수가 없으므로
(특정 시점에서의 집단 수만 파악할 수 있기 때문에) 승산비를 사용할 수밖에 없다.

$$\text{Odds Ratio} = \text{OR} = \frac{P(\text{disease} | \text{exposed}) / [1 - P(\text{disease} | \text{exposed})]}{P(\text{disease} | \text{unexposed}) / [1 - P(\text{disease} | \text{unexposed})]}$$

Logit link F. = **Logistic unit** link F.
 = **Log-odds** link F. = Logit is a transformation

- Q: 뭐가 **기호적**이라는걸까요? A: 바로 그래프 모양 **-무한, +무한**

$$\text{logit}(p(y = 1|x)) = \log_e \left(\frac{p}{1-p} \right)$$

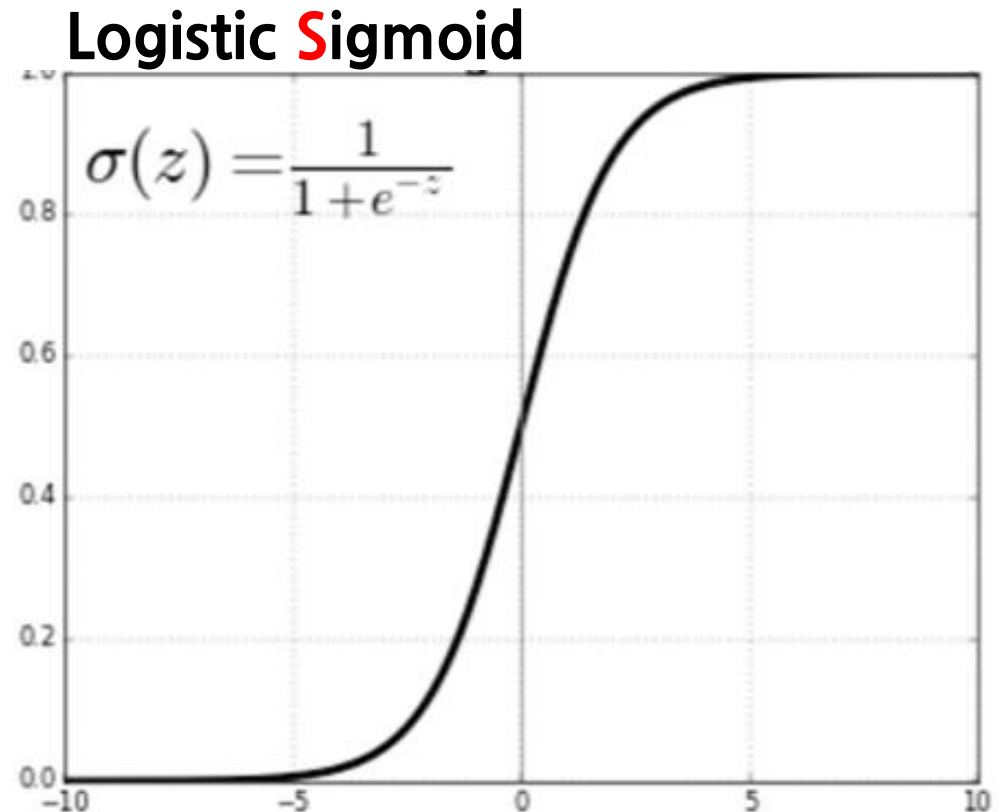
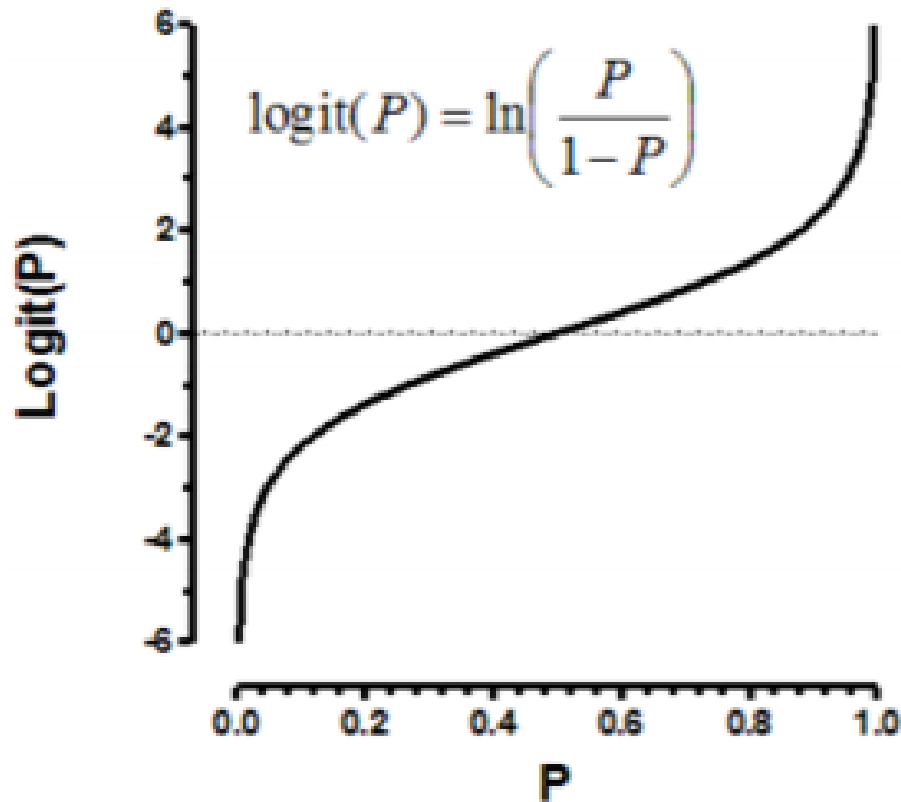


$$\text{logit}(P(Y = 1 | x_1, \dots, x_k)) = \beta_0 + \beta_1 x_1 + \dots + \beta_n x_n$$

Inverse Logit = Logistic Sigmoid **F**.

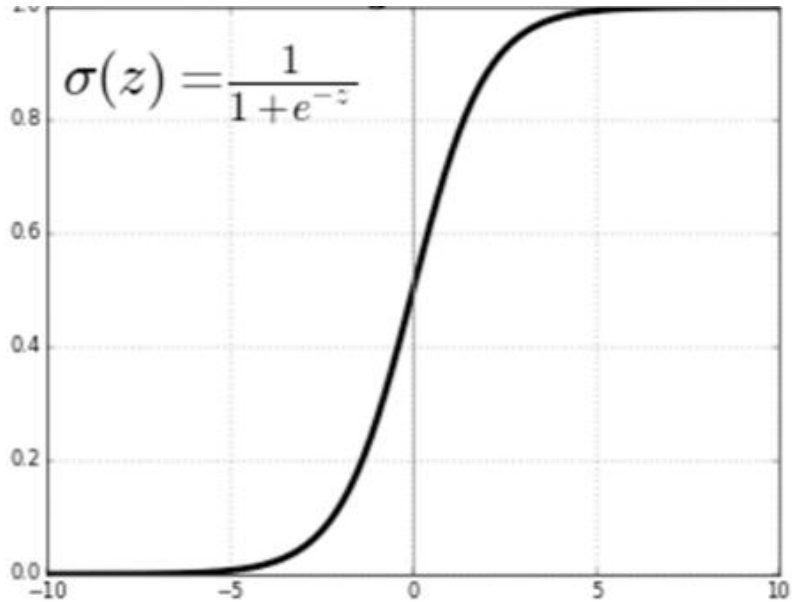
2

- Logit의 역함수 = Logistic Sigmoid vs 다른 S 모양은?

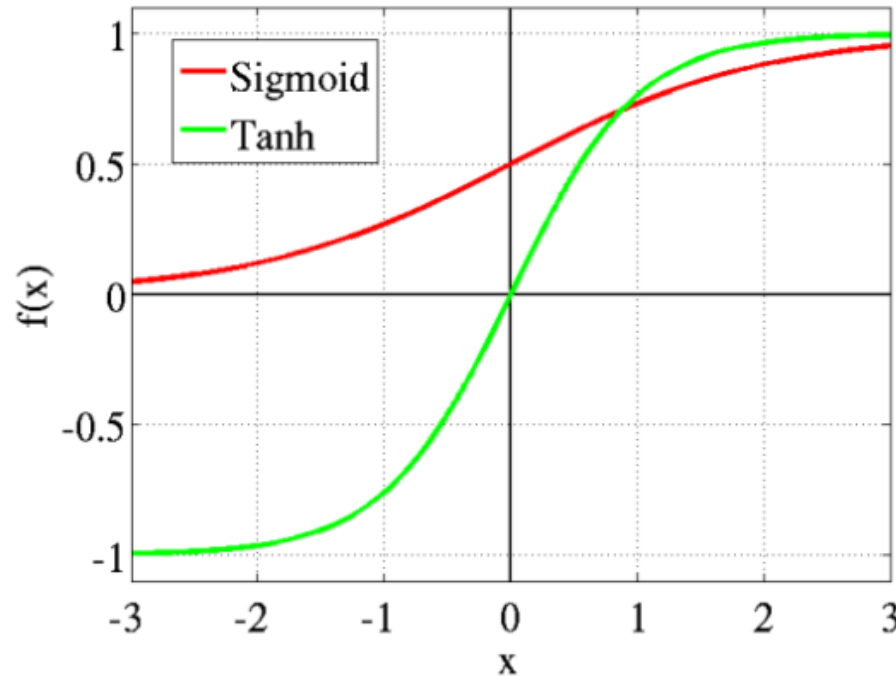


Inverse Logit = Logistic Sigmoid **F**.

- Logit의 역함수 = Logistic Sigmoid = **S** 모양의 로지스틱 함수



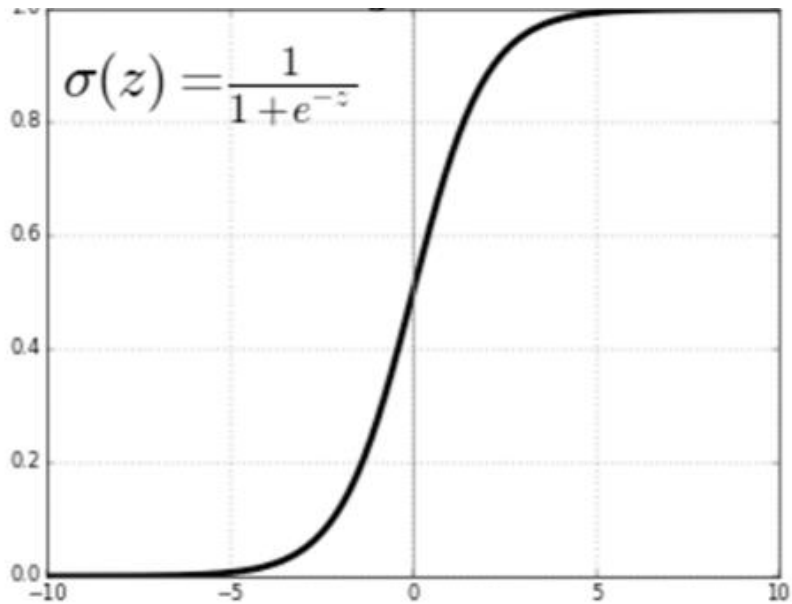
<https://goo.gl/38SsHw>



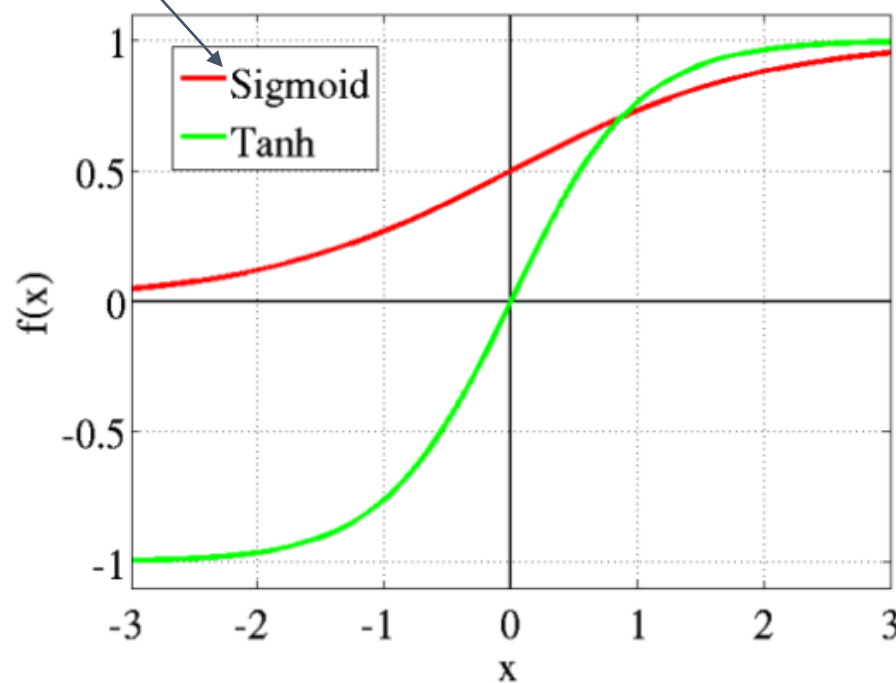
vs *tanh*

Inverse Logit = Logistic Sigmoid F.

- Logit의 역함수 = Logistic **Sigmoid** = **S** 모양의 로지스틱 함수



<https://goo.gl/38SsHw>



vs *tanh*
Sigmoid

S 모양의 하이퍼볼릭
탄젠트 함수

www.datamar

2019. 3. 12. - **tanh** : sigmoid function의 가중치 학습시 역전파된 gradient의 방향에 제약이 가해져 학습속도가 늦거나 수렴이 어렵게 되는 문제를 해결한 함수.

Weight update

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

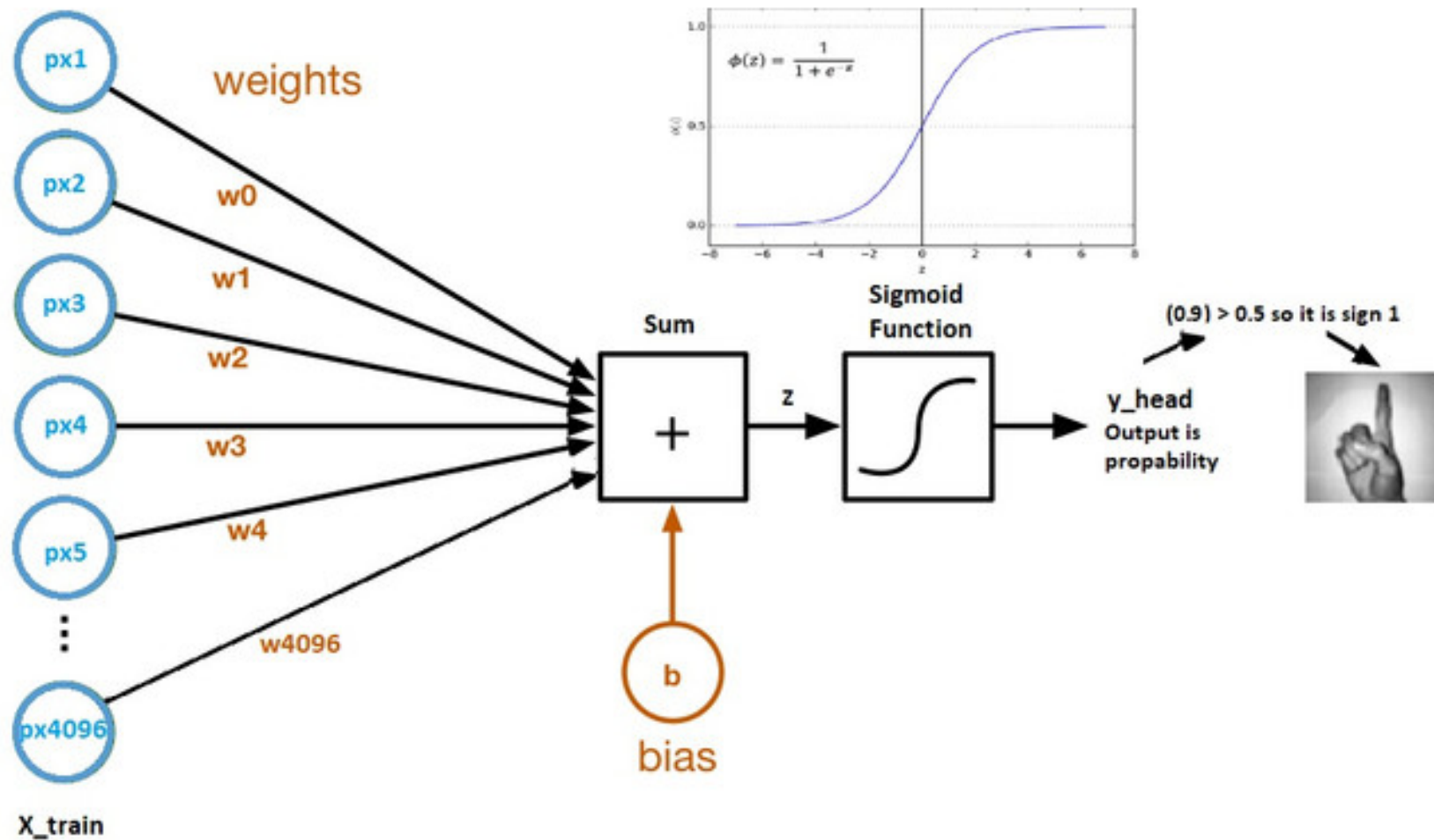
$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

모든 θ_j 동시에 업데이트

$$:= \theta_j - \alpha \sum_{i=1}^m (h_{\theta}(x^i) - y^i) x_j^i$$

Logistic Regression에서 Weight 학습하기

가설함수



Iris 아이리스, 붓꽃



Iris setosa



Iris versicolor



Iris virginica

꽃 세가지 종류(Versicolor, Setosa, Virginica)의 꽃을
4가지 숫자 cm: Sepal(꽃받침) 길이, 폭, **Petal(꽃잎) 길이, 폭**

<https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data>

Sir Ronald Aylmer Fisher (1936)

LogisticRegressionGD

[1] 하고싶은 메쏘드가 무엇인가?

[2] 그런 메쏘드들을

틀로 묶어서 마련해둔 것은 무엇인가

```
X_train_01_subset = X_train[(y_train == 0) | (y_train == 1)]  
y_train_01_subset = y_train[(y_train == 0) | (y_train == 1)]  
  
lrgd = LogisticRegressionGD(eta=0.05, n_iter=1000, random_state=1)  
lrgd.fit(X_train_01_subset,  
         y_train_01_subset)
```

[1]

method

정의 완료

```
def fit(self, X, y):
    """ 훈련 데이터 학습 """

    rgen = np.random.RandomState(self.random_state)
    # 표준편차(scale)가 0.01인 정규 분포에서 뽑은 랜덤한 작은수
    self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
    self.cost_ = []

    for i in range(self.n_iter):
        net_input = self.net_input(X)
        output = self.activation(net_input)
        errors = (y - output)
        self.w_[1:] += self.eta * X.T.dot(errors)
        self.w_[0] += self.eta * errors.sum()

        # 오차 제곱합 대신 로지스틱 비용을 계산합니다.
        cost = -y.dot(np.log(output)) - ((1 - y).dot(np.log(1 - output)))
        self.cost_.append(cost)

    return self
```

```
X_train_01_subset = X_train[(y_train == 0) | (y_train == 1)]
y_train_01_subset = y_train[(y_train == 0) | (y_train == 1)]
```

```
lrgd = LogisticRegressionGD(eta=0.05, n_iter=1000, random_state=1)
lrgd.fit(X_train_01_subset,
        y_train_01_subset)
```


[2]

메소드를

묶은 이름

class

```
class LogisticRegressionGD(object):  
    """경사 하강법을 사용한 로지스틱 회귀 분류기  
    """
```

```
    def fit(self, X, y):  
        """훈련 데이터 학습  
        """  
  
        rgen = np.random.RandomState(self.random_state)  
        # 표준편차(scale)가 0.01인 정규 분포에서 뽑은 랜덤한 작은수  
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])  
        self.cost_ = []  
  
        for i in range(self.n_iter):  
            net_input = self.net_input(X)  
            output = self.activation(net_input)  
            errors = (y - output)  
            self.w_[1:] += self.eta * X.T.dot(errors)  
            self.w_[0] += self.eta * errors.sum()  
  
            # 오차 제곱합 대신 로지스틱 비용을 계산합니다.  
            cost = -y.dot(np.log(output)) - ((1 - y).dot(np.log(1 - output)))  
            self.cost_.append(cost)  
        return self
```

```
X_train_01_subset = X_train[(y_train == 0) | (y_train == 1)]  
y_train_01_subset = y_train[(y_train == 0) | (y_train == 1)]
```

```
lrgd = LogisticRegressionGD(eta=0.05, n_iter=1000, random_state=1)  
lrgd.fit(X_train_01_subset,  
         y_train_01_subset)
```

```

def fit(self, X, y):
    """훈련 데이터 학습
    """

    rgen = np.random.RandomState(self.random_state)
    # 표준편차(scale)가 0.01인 정규 분포에서 뽑은 랜덤한 작은수
    self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
    self.cost_ = []

    for i in range(self.n_iter):
        net_input = self.net_input(X)
        output = self.activation(net_input)
        errors = (y - output)
        self.w_[1:] += self.eta * X.T.dot(errors)
        self.w_[0] += self.eta * errors.sum()

        # 오차 제곱합 대신 로지스틱 비용을 계산합니다.
        cost = -y.dot(np.log(output)) - ((1 - y).dot(np.log(1 - output)))
        self.cost_.append(cost)

    return self

```

외부 메소드를 활용한 결과일 때, 변수 이름 끝에 _언더바

```

def fit(self, X, y):
    """훈련 데이터 학습
    """

    rgen = np.random.RandomState(self.random_state)
    # 표준편차(scale)가 0.01인 정규 분포에서 뽑은 랜덤한 작은수
    self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
    self.cost_ = []

    for i in range(self.n_iter):
        net_input = self.net_input(X)
        output = self.activation(net_input)
        errors = (y - output)
        self.w_[1:] += self.eta * X.T.dot(errors)
        self.w_[0] += self.eta * errors.sum()

        # 오차 제곱합 대신 로지스틱 비용을 계산합니다.
        cost = -y.dot(np.log(output)) - ((1 - y).dot(np.log(1 - output)))
        self.cost_.append(cost)

    return self

```

인스턴스.net_input이라는 메소드 결과를
net_input이라는 변수 이름에 저장합니다.
 이유는?
 인스턴스.activation이라는 메소드의 입력으로.

net_input, activation

```
def net_input(self, X):  
    """최종 입력 계산"""  
    return np.dot(X, self.w_[1:]) + self.w_[0]
```

```
def activation(self, z):  
    """로지스틱 시그모이드 활성화 계산"""  
    return 1. / (1. + np.exp(-np.clip(z, -250, 250)))
```

net_input, activation

```
def net_input(self, X):  
    """최종 입력 계산"""  
    print('np.shape(X):', np.shape(X))  
    print('self.w_[1:]', self.w_[1:])  
  
    print('np.dot(X, self.w_[1:]):', np.dot(X, self.w_[1:]))  
    print('self.w_[0]:', self.w_[0])  
    print('np.dot(X, self.w_[1:]) + self.w_[0]:', np.dot(X, self.w_[1:]) + self.w_[0])  
    return np.dot(X, self.w_[1:]) + self.w_[0]
```

`np.dot(X, w_[1:])`

```
print(X)
np.shape(X)
```

클래스 레이블: [0 1 2]

```
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]
 [1.7 0.4]
 [1.4 0.3]
 [1.5 0.2]
 [1.4 0.2]
 [1.5 0.1]
 [1.5 0.2]
 [1.6 0.2]
 [1.4 0.1]]
```

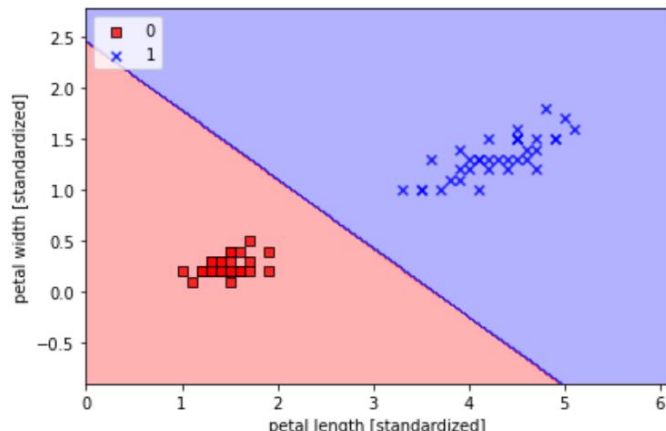
`np.shape(X): (70, 2)`

`self.w_[1:] [3.56295047 5.26970842]`

X_train

`[feat 1 feat 2] X_train[(y_train == 0) | (y_train == 1)]`

`(150, 2)`



np.dot(X, w_[1:])

```
print(X)
np.shape(X)
```

클래스 레이블: [0 1 2]

[[1.4 0.2]

[1.4 0.2]

[1.3 0.2]

[1.5 0.2]

[1.4 0.2]

[1.7 0.4]

[1.4 0.3]

[1.5 0.2]

[1.4 0.2]

[1.5 0.1]

[1.5 0.2]

[1.6 0.2]

[1.4 0.1]

self.w_[1:] [3.31854395 4.93118397]

self.w_[1:] [3.31945117 4.93244619]

self.w_[1:] [3.32035716 4.93370664]

self.w_[1:] [3.32126191 4.93496534]

self.w_[1:] [3.32216543 4.93622228]

self.w_[1:] [3.32306772 4.93747747]

self.w_[1:] [3.3239688 4.93873092]

self.w_[1:] [3.32486865 4.93998263]

self.w_[1:] [3.32576728 4.9412326]

self.w_[1:] [3.3266647 4.94248084]

self.w_[1:] [3.32756091 4.94372735]

self.w_[1:] [3.32845592 4.94497215]

self.w_[1:] [3.32934972 4.94621523]

self.w_[1:] [3.55788674 5.26272471]

self.w_[1:] [3.55852178 5.26360061]

self.w_[1:] [3.55915623 5.26447566]

self.w_[1:] [3.55979008 5.26534987]

self.w_[1:] [3.56042334 5.26622325]

self.w_[1:] [3.56105601 5.26709579]

self.w_[1:] [3.56168808 5.2679675]

self.w_[1:] [3.56231957 5.26883838]

self.w_[1:] [3.56295047 5.26970842]

self.w_[1:] [3.56358078 5.27057764]

[3.56 5.27]

[feat 1 feat 2]

w_[0]

```
print(X)  
np.shape(X)
```

클래스 레이블: [0 1 2]

[[1.4 0.2]

[1.4 0.2]

[1.3 0.2]

[1.5 0.2]

[1.4 0.2]

[1.7 0.4]

[1.4 0.3]

[1.5 0.2]

[1.4 0.2]

[1.5 0.1]

[1.5 0.2]

[1.6 0.2]

[1.4 0.1]

[feat 1 feat 2]

self.w_[1:] [3.56168808 5.2679675]

self.w_[0]: -12.947984078175484

self.w_[1:] [3.56231957 5.26883838]

self.w_[0]: -12.95021916175178

self.w_[1:] [3.56295047 5.26970842]

self.w_[0]: -12.952452142176151

self.w_[1:] [3.56358078 5.27057764]

self.w_[0]: -12.954683023495592

-12.95

np.dot(X, w_[1:])

```
print(X)
np.shape(X)
```

클래스 레이블: [0 1 2]

```
[[1.4 0.2]
```

```
[1.4 0.2]
```

```
[1.3 0.2]
```

```
[1.5 0.2]
```

```
[1.4 0.2]
```

```
[1.7 0.4]
```

```
[1.4 0.3]
```

```
[1.5 0.2]
```

```
[1.4 0.2]
```

```
[1.5 0.1]
```

```
[1.5 0.2]
```

```
[1.6 0.2]
```

```
[1.4 0.1]
```

[feat 1 feat 2]

(150, 2)

X_train

$W1x1 + W2x2$

Case 70개

```
np.shape(X): (70, 2)
```

```
self.w_[1:] [3.56295047 5.26970842]
```

```
np.dot(X, self.w_[1:]): [ 6.04207234  7.11095748  5.32948225  5.32948225  2
```

```
21.27309862  5.68577729 23.06951731 21.10242283  7.82354758 17.74003506
```

```
20.21915694  6.75466244 23.93783975 22.88389806 19.33589105 24.123459
```

```
21.45871787  7.45230907 22.8689546  6.75466244  6.04207234  8.87748926
```

```
4.61689215 25.36301993  6.39836739  4.44621636 24.65042984  7.63792832
```

```
17.74003506 19.87780535  6.39836739  6.39836739  6.39836739 18.45262516
```

```
6.21274814 25.36301993  5.68577729 24.46481059  6.75466244  6.04207234
```

```
22.00063217 19.67724264 21.45871787 26.58763741 23.93783975 23.93783975
```

```
5.8087  6.39836739 21.81501292
```

```
6.32333623  6.04207234  6.38304318 19.63218609 21.28804208 22.17130797
```

```
5.87139655 22.52760301  5.68577729  6.56904318 26.77325666  6.21274814
```

```
20.57545198  7.80860412  6.39836739  5.68577729]
```

```
self.w_[0]: -12.952452142176151
```

```
np.dot(X, self.w_[1:]) + self.w_[0]: [-6.9103798  -5.84149466 -7.62296989
```

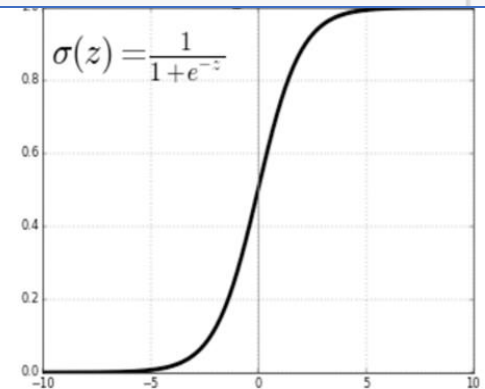
```
8.32064648 -7.26667485 10.11706517  8.14997068 -5.12890457  4.78758292
```

```
7.26670479 -6.19778971 10.9853876  9.93144592  6.38343891 11.17100685
```

net_input, activation

```
def net_input(self, X):  
    """최종 입력 계산"""  
    print('np.shape(X):', np.shape(X))  
    print('self.w_[1:]', self.w_[1:])  
  
    print('np.dot(X, self.w_[1:]):', np.dot(X, self.w_[1:]))  
    print('self.w_[0]:', self.w_[0])  
    print('np.dot(X, self.w_[1:]) + self.w_[0]:', np.dot(X, self.w_[1:]) + self.w_[0])  
    return np.dot(X, self.w_[1:]) + self.w_[0]  
  
def activation(self, z):  
    """로지스틱 시그모이드 활성화 계산"""  
    return 1. / (1. + np.exp(-np.clip(z, -250, 250)))
```

- **np.clip(z를 최소 -250, 최대 250 로 범위 클립)**



Plot_Decision_regions

```
lrgd = LogisticRegressionGD(eta=0.05, n_iter=1000, random_state=1)
lrgd.fit(X_train_01_subset,
        y_train_01_subset)
```

```
plot_decision_regions(X=X_train_01_subset,
                     y=y_train_01_subset,
                     classifier=lrgd)
```

```
plt.xlabel('petal length [standardized]')
plt.ylabel('petal width [standardized]')
plt.legend(loc='upper left')
```

```
plt.tight_layout()
plt.show()
```

```
lrgd = LogisticRegressionGD(eta=0.05, n_iter=1000, random_state=1)
lrgd.fit(X_train_01_subset,
        y_train_01_subset)
```

```
plot_decision_regions(X=X_train_01_subset,
                     y=y_train_01_subset,
                     classifier=lrgd)
```

```
plt.xlabel('petal length [standardized]')
plt.ylabel('petal width [standardized]')
plt.legend(loc='upper left')
```

```
plt.tight_layout()
plt.show()
```

```
def plot_decision_regions(X, y, classifier, test_idx=None, resolution=0.02):
```

```
# 마커와 컬러맵을 설정합니다.
```

```
markers = ('s', 'x', 'o', '^', 'v')
```

```
colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
```

```
cmap = ListedColormap(colors[:len(np.unique(y))])
```

```
# 결정 경계를 그립니다.
```

```
x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
```

```
x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
```

```
xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                      np.arange(x2_min, x2_max, resolution))
```

```
Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
```

```
Z = Z.reshape(xx1.shape)
```

```
np.ravel(x, order='C') # by default
```

```
plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
```

```
plt.xlim(xx1.min(), xx1.max())
```

```
plt.ylim(xx2.min(), xx2.max())
```

```

xx1:  [[0.    0.02 0.04 ... 7.84 7.86 7.88]
 [0.    0.02 0.04 ... 7.84 7.86 7.88]
 [0.    0.02 0.04 ... 7.84 7.86 7.88]
 ...
 [0.    0.02 0.04 ... 7.84 7.86 7.88]
 [0.    0.02 0.04 ... 7.84 7.86 7.88]
 [0.    0.02 0.04 ... 7.84 7.86 7.88]]
xx2:  [[-0.9 -0.9 -0.9 ... -0.9 -0.9 -0.9 ]
 [-0.88 -0.88 -0.88 ... -0.88 -0.88 -0.88]
 [-0.86 -0.86 -0.86 ... -0.86 -0.86 -0.86]
 ...
 [ 3.44  3.44  3.44 ...  3.44  3.44  3.44]
 [ 3.46  3.46  3.46 ...  3.46  3.46  3.46]
 [ 3.48  3.48  3.48 ...  3.48  3.48  3.48]]
np.shape(xx1):  (220, 395)
xx1.ravel():  [0.    0.02 0.04 ... 7.84 7.86 7.88]

```

```

def plot_decision_regions(X, y, classifier, test_idx=None, resolution=0.02):

    # 마커와 컬러맵을 설정합니다.
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # 결정 경계를 그립니다.
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

```

`a2 = a1.ravel()` # 또는 `a2 = a1.reshape(-1)` 또는 `a2 = a1.flatten()`

결정 경계를 그립니다.

```
x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
```

```
x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
```

```
xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),  
                        np.arange(x2_min, x2_max, resolution))
```

```
Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
```

```
Z = Z.reshape(xx1.shape)
```

```
plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
```

범위와
표현할 간격 결정

X and Y must both be 2-D with the same shape as Z

contour는 등고선만 표시, **contourf**는 색깔로 표시.

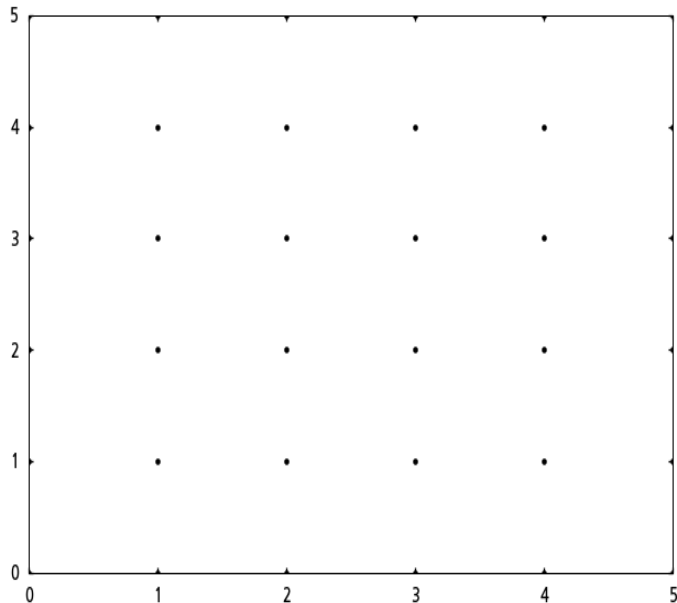
x, y 는 **meshgrid** 명령으로 **그리드 포인트** 행렬을 만들어야 합니다.

```
plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
```

X and *Y* must both be 2-D with the same shape as *Z*

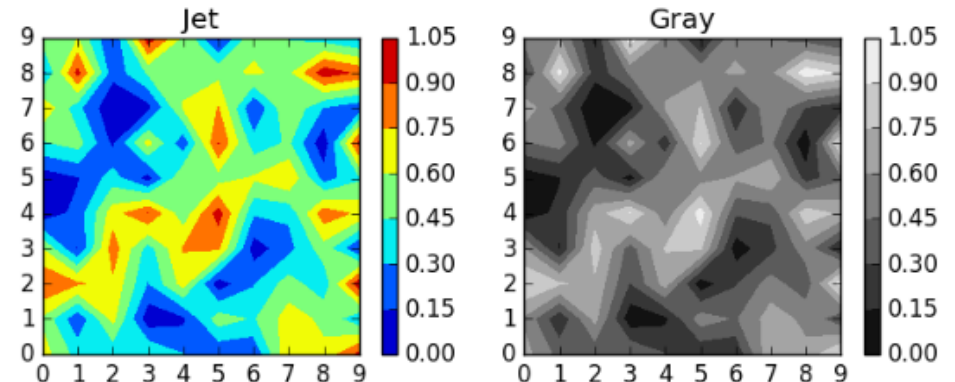
contour = 등고선, **contourf**는 색지정 기능 포함.
x, y는 **meshgrid** 명령으로 그리드 포인트 행렬을 만들어야 합니다.

alpha: alpha attribute = transparency(투명도) <-> opacity (불투명도)

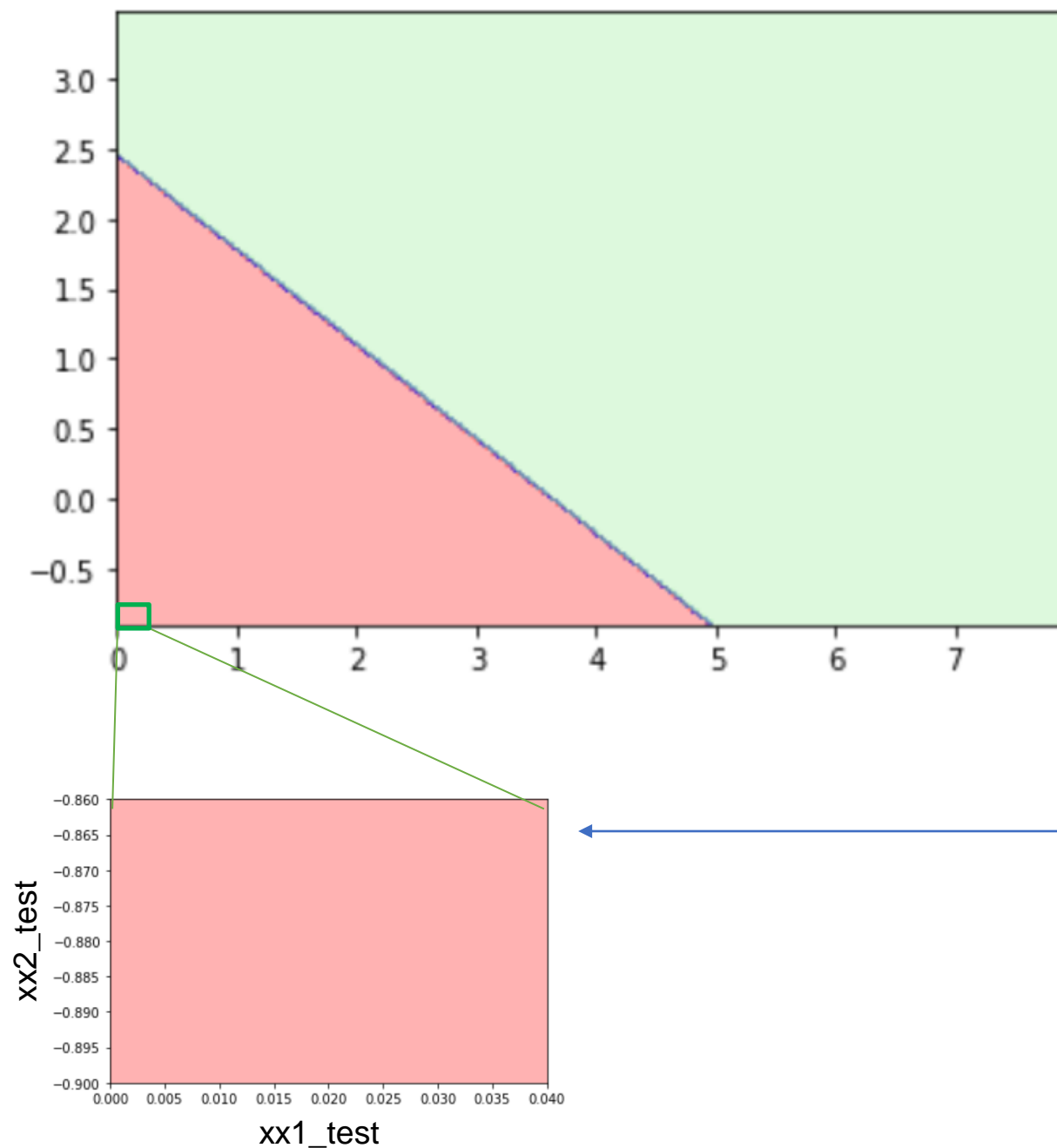


```
plt.subplot(1,2,1)
con = plt.contourf(data, cmap=cm.jet)
plt.title('Jet')
plt.colorbar()
```

```
hax = plt.subplot(1,2,2)
con = plt.contourf(data, cmap=cm.gray)
plt.title('Gray')
plt.colorbar()
```




```
plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
```



```
xx1:  [[0.   0.02 0.04 ... 7.84 7.86 7.88]
 [0.   0.02 0.04 ... 7.84 7.86 7.88]
 [0.   0.02 0.04 ... 7.84 7.86 7.88]
 ...
 [0.   0.02 0.04 ... 7.84 7.86 7.88]
 [0.   0.02 0.04 ... 7.84 7.86 7.88]
 [0.   0.02 0.04 ... 7.84 7.86 7.88]]
xx2:  [[-0.9  -0.9  -0.9  ... -0.9  -0.9  -0.9 ]
 [-0.88 -0.88 -0.88 ... -0.88 -0.88 -0.88]
 [-0.86 -0.86 -0.86 ... -0.86 -0.86 -0.86]
 ...
 [ 3.44  3.44  3.44 ...  3.44  3.44  3.44]
 [ 3.46  3.46  3.46 ...  3.46  3.46  3.46]
 [ 3.48  3.48  3.48 ...  3.48  3.48  3.48]]
np.shape(xx1):  (220, 395)
xx1_test:  [[0.   0.02 0.04]
 [0.   0.02 0.04]
 [0.   0.02 0.04]]
xx2_test:  [[-0.9  -0.9  -0.9 ]
 [-0.88 -0.88 -0.88]
 [-0.86 -0.86 -0.86]]
```

결정 경계를 그립니다.

```
x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
```

```
x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
```

```
xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),  
                        np.arange(x2_min, x2_max, resolution))
```

```
Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
```

```
Z = Z.reshape(xx1.shape)
```

```
plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
```

범위와
표현할 간격 결정

```
def predict(self, X):
```

```
    """단위 계단 함수를 사용하여 클래스 레이블을 반환합니다"""
```

```
    return np.where(self.net_input(X) >= 0.0, 1, 0)
```

```
    # 다음과 동일합니다.
```

```
    # return np.where(self.activation(self.net_input(X)) >= 0.5, 1, 0)
```

결정 경계를 그립니다.

```
x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
```

```
x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
```

```
xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),  
                        np.arange(x2_min, x2_max, resolution))
```

```
Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
```

```
Z = Z.reshape(xx1.shape)
```

```
plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
```

범위와
표현할 간격 결정

```
def predict(self, X):
```

```
    """단위 계단 함수를 사용하여 클래스 레이블을 반환합니다"""
```

```
    return np.where(self.net_input(X) >= 0.0, 1, 0)
```

```
    # 다음과 동일합니다.
```

```
    # return np.where(self.activation(self.net_input(X)) >= 0.5, 1, 0)
```

```
def net_input(self, X):
```

```
    """최종 입력 계산"""
```

```
    return np.dot(X, self.w_[1:]) + self.w_[0]
```

```

def fit(self, X, y):
    """훈련 데이터 학습
    """

    rgen = np.random.RandomState(self.random_state)
    # 표준편차(scale)가 0.01인 정규 분포에서 뽑은 랜덤한 작은수
    self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
    self.cost_ = []

    for i in range(self.n_iter):
        net_input = self.net_input(X)
        output = self.activation(net_input)
        errors = (y - output)
        self.w_[1:] += self.eta * X.T.dot(errors)
        self.w_[0] += self.eta * errors.sum()

        # 오차 제곱합 대신 로지스틱 비용을 계산합니다.
        cost = -y.dot(np.log(output)) - ((1 - y).dot(np.log(1 - output)))
        self.cost_.append(cost)

    return self

```

```

for i in range(self.n_iter):
    net_input = self.net_input(X)
    output = self.activation(net_input)
    errors = (y - output)
    self.w_[1:] += self.eta * X.T.dot(errors)
    self.w_[0] += self.eta * errors.sum()

```

Error와 입력X의 내적에 학습률을 곱해서 Weight에 누적합 반영

Bias편향은 에러만 학습률을 곱해서 누적합 반영

퍼셉트론 학습 알고리즘(perceptron Learning algorithm)

의 일부만 코드로 구현한 것

- 1) 가중치, 바이어스를 임의의 값으로 초기화
- 2) 하나의 학습 **features** 에 대한 뉴런의 **net_input**값을 계산
- 3) **activation** 활성화함수 (**net_input**입력)로 뉴런의 가설 출력값 도출
- 4) 에러에 따라 가중치 업데이트 및 2~4 반복

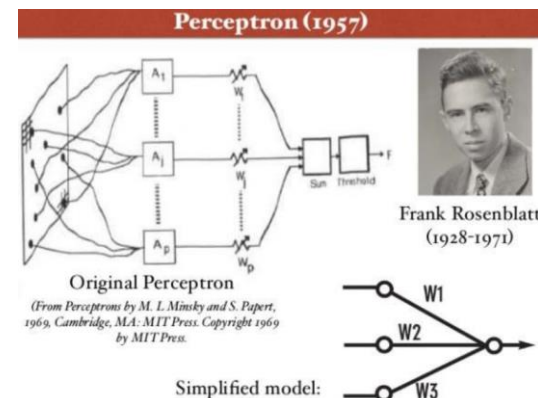
Perception 감지

Dendron 수상돌기 (나무)

Electron 전자



Dendron

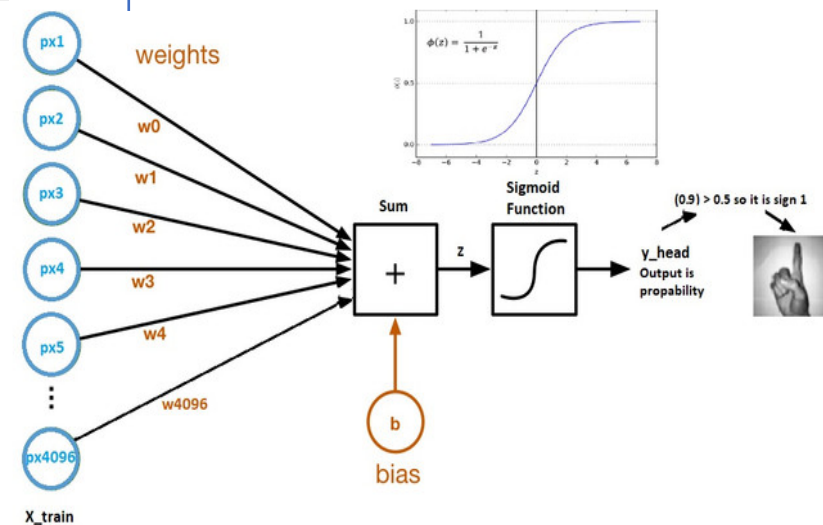


```
def net_input(self, X):
    """최종 입력 계산"""
    return np.dot(X, self.w_[1:]) + self.w_[0]

def activation(self, z):
    """로지스틱 시그모이드 활성화 계산"""
    return 1. / (1. + np.exp(-np.clip(z, -250, 250)))
```

```
def predict(self, X):
    """단위 계단 함수를 사용하여 클래스 레이블을 반환합니다"""
    return np.where(self.net_input(X) >= 0.0, 1, 0)
# 다음과 동일합니다.
# return np.where(self.activation(self.net_input(X)) >= 0.5, 1, 0)
```

$w_1x_0 + w_2x_1 + w_0 \geq 0$ or < 0



np.dot(X, w_[1:])

```
print(X)
np.shape(X)
```

클래스 레이블: [0 1 2]

[[1.4 0.2]

[1.4 0.2]

[1.3 0.2]

[1.5 0.2]

[1.4 0.2]

[1.7 0.4]

[1.4 0.3]

[1.5 0.2]

[1.4 0.2]

[1.5 0.1]

[1.5 0.2]

[1.6 0.2]

[1.4 0.1]

self.w_[1:] [3.31854395 4.93118397]

self.w_[1:] [3.31945117 4.93244619]

self.w_[1:] [3.32035716 4.93370664]

self.w_[1:] [3.32126191 4.93496534]

self.w_[1:] [3.32216543 4.93622228]

self.w_[1:] [3.32306772 4.93747747]

self.w_[1:] [3.3239688 4.93873092]

self.w_[1:] [3.32486865 4.93998263]

self.w_[1:] [3.32576728 4.9412326]

self.w_[1:] [3.3266647 4.94248084]

self.w_[1:] [3.32756091 4.94372735]

self.w_[1:] [3.32845592 4.94497215]

self.w_[1:] [3.32934972 4.94621523]

self.w_[1:] [3.55788674 5.26272471]

self.w_[1:] [3.55852178 5.26360061]

self.w_[1:] [3.55915623 5.26447566]

self.w_[1:] [3.55979008 5.26534987]

self.w_[1:] [3.56042334 5.26622325]

self.w_[1:] [3.56105601 5.26709579]

self.w_[1:] [3.56168808 5.2679675]

self.w_[1:] [3.56231957 5.26883838]

self.w_[1:] [3.56295047 5.26970842]

self.w_[1:] [3.56358078 5.27057764]

[3.56 5.27]

[feat 1 feat 2]

w_[0]

```
print(X)  
np.shape(X)
```

클래스 레이블: [0 1 2]

[[1.4 0.2]

[1.4 0.2]

[1.3 0.2]

[1.5 0.2]

[1.4 0.2]

[1.7 0.4]

[1.4 0.3]

[1.5 0.2]

[1.4 0.2]

[1.5 0.1]

[1.5 0.2]

[1.6 0.2]

[1.4 0.1]

[feat 1 feat 2]

self.w_[1:] [3.56168808 5.2679675]

self.w_[0]: -12.947984078175484

self.w_[1:] [3.56231957 5.26883838]

self.w_[0]: -12.95021916175178

self.w_[1:] [3.56295047 5.26970842]

self.w_[0]: -12.952452142176151

self.w_[1:] [3.56358078 5.27057764]

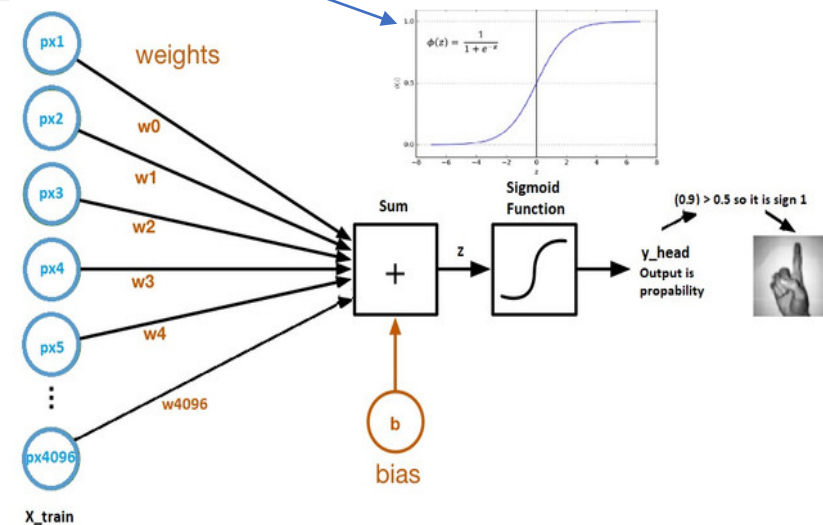
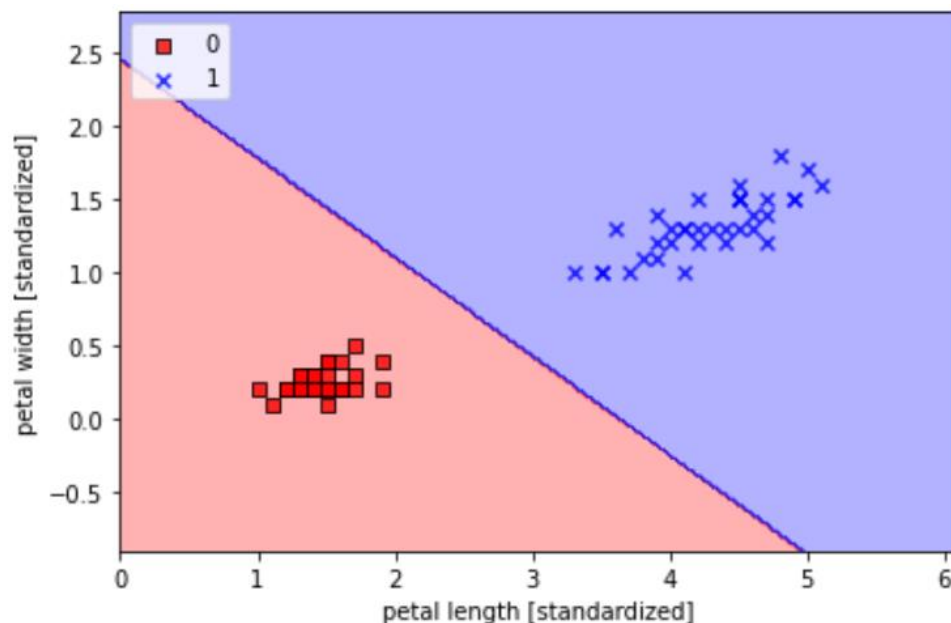
self.w_[0]: -12.954683023495592

-12.95

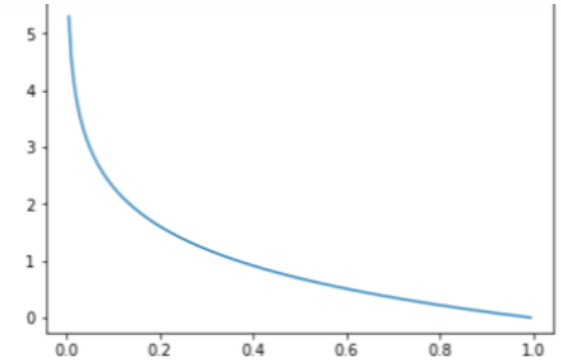

```
def net_input(self, X):
    """최종 입력 계산"""
    return np.dot(X, self.w_[1:]) + self.w_[0]

def activation(self, z):
    """로지스틱 시그모이드 활성화 계산"""
    return 1. / (1. + np.exp(-np.clip(z, -250, 250)))
```

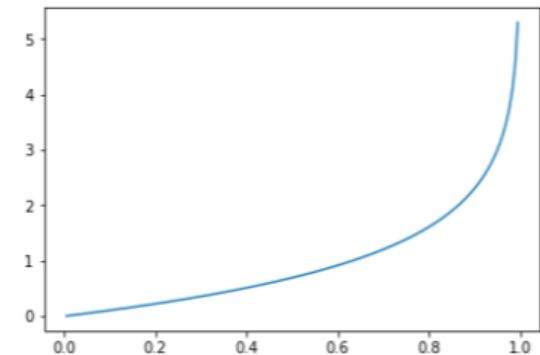
```
def predict(self, X):
    """단위 계단 함수를 사용하여 클래스 레이블을 반환합니다"""
    return np.where(self.net_input(X) >= 0.0, 1, 0)
# 다음과 동일합니다.
# return np.where(self.activation(self.net_input(X)) >= 0.5, 1, 0)
```



Cost Function



$$\text{Cost}(h_{\theta}(x), y) = \begin{cases} -\log(h_{\theta}(x)) & \text{if } y = 1 \\ -\log(1 - h_{\theta}(x)) & \text{if } y = 0 \end{cases}$$

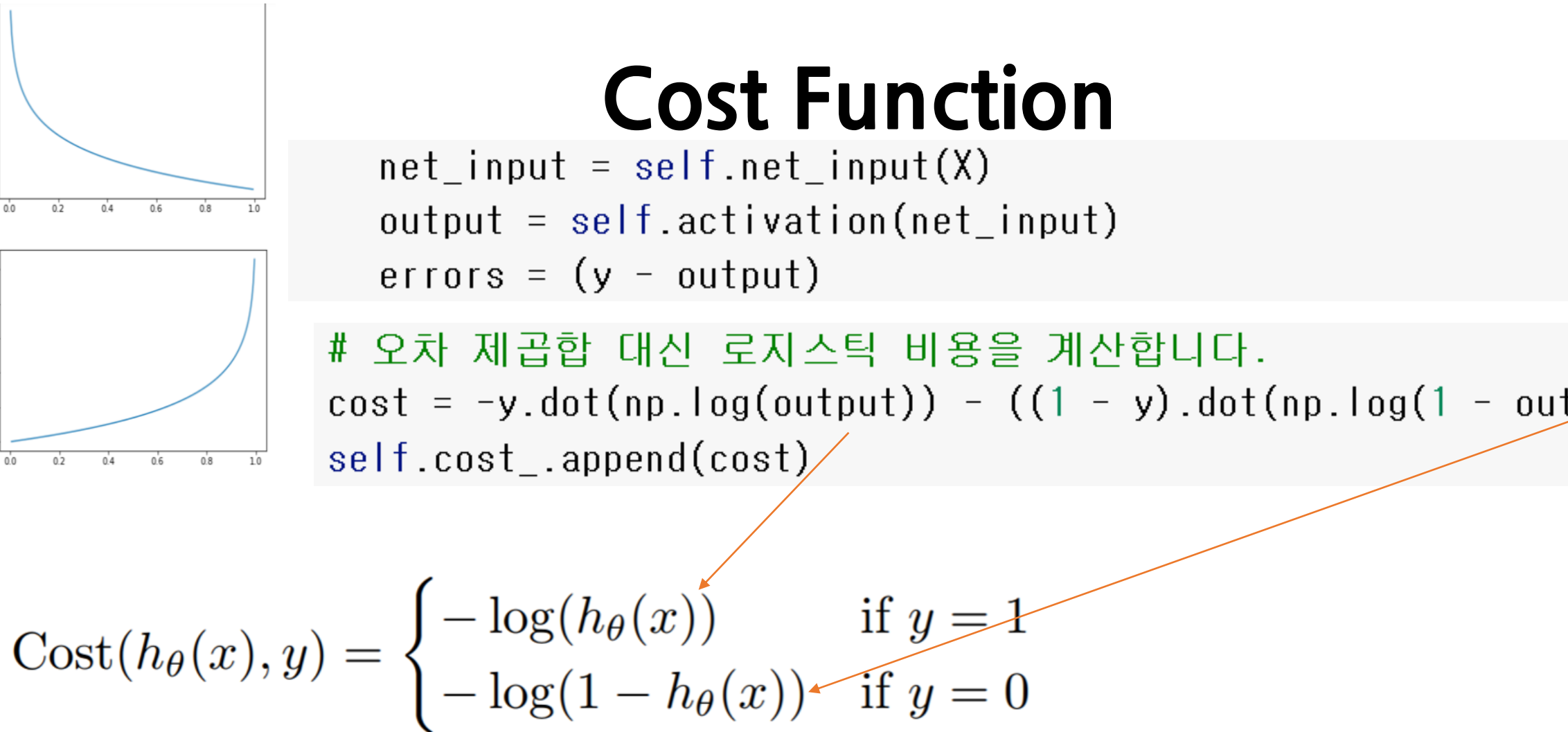


Cost Function

```
net_input = self.net_input(X)
output = self.activation(net_input)
errors = (y - output)
```

오차 제곱합 대신 로지스틱 비용을 계산합니다.

```
cost = -y.dot(np.log(output)) - ((1 - y).dot(np.log(1 - output)))
self.cost_.append(cost)
```

$$\text{Cost}(h_{\theta}(x), y) = \begin{cases} -\log(h_{\theta}(x)) & \text{if } y = 1 \\ -\log(1 - h_{\theta}(x)) & \text{if } y = 0 \end{cases}$$


The image contains two graphs of the cost function on the left. The top graph shows the cost for y=1, which is -log(h_theta(x)), decreasing from 5 to 0 as h_theta(x) goes from 0 to 1. The bottom graph shows the cost for y=0, which is -log(1-h_theta(x)), increasing from 0 to 5 as h_theta(x) goes from 0 to 1. Two orange arrows point from the code in the block above to the formula below: one from 'self.cost_.append(cost)' to the first case of the piecewise function, and another from '((1 - y).dot(np.log(1 - output)))' to the second case.

Cost Function

$$\begin{aligned} J(\theta) &= \frac{1}{m} \sum_{i=1}^m \text{Cost} \left(h_{\theta}(x^{(i)}), y^{(i)} \right) \\ &= -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log h_{\theta}(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))] \end{aligned}$$

미션!

$$\text{find } \theta, \text{ where } \min_{\theta} J(\theta) \quad h_{\theta}(x) = \frac{1}{1 + e^{-\theta^T \mathbf{x}}}$$

Partial derivation of cost function

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

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

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m \left[y_i \theta x^i - \theta x^i - \log(1 + e^{-\theta x^i}) \right] \quad h_{\theta}(x) = \frac{1}{1 + e^{-\theta^T \mathbf{x}}}$$

$$= -\frac{1}{m} \sum_{i=1}^m \left[y_i \theta x^i - \log(1 + e^{\theta x^i}) \right]$$

$$-\theta x^i - \log(1 + e^{-\theta x^i}) = - \left[\log e^{\theta x^i} + \log(1 + e^{-\theta x^i}) \right]$$

$$= -\log(1 + e^{\theta x^i}).$$

Partial derivation of cost function

$$-\frac{1}{m} \sum_{i=1}^m \left[y_i \theta x^i - \log(1 + e^{\theta x^i}) \right]$$

θ 에 관하여 미분하면

$$\frac{\partial}{\partial \theta_j} y_i \theta x^i = y_i x_j^i$$

$$\frac{\partial}{\partial \theta_j} \log(1 + e^{\theta x^i}) = \frac{x_j^i e^{\theta x^i}}{1 + e^{\theta x^i}} = x_j^i h_{\theta}(x^i),$$

Partial derivation of cost function

$$-\frac{1}{m} \sum_{i=1}^m \left[y_i \theta x^i - \log(1 + e^{\theta x^i}) \right]$$

$\frac{\partial}{\partial \theta_j} y_i \theta x^i = y_i x_j^i$ $\frac{\partial}{\partial \theta_j} \log(1 + e^{\theta x^i}) = \frac{x_j^i e^{\theta x^i}}{1 + e^{\theta x^i}} = x_j^i h_\theta(x^i),$

$h_\theta(x) = \frac{1}{1 + e^{-\theta^T \mathbf{x}}}$

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

양성 음성 각각의 이상적인 y값에 가까우면 최적인 것과 통함.