

04

분류하는 뉴런을 만듭니다

이 장에서는 03장에서 배운 선형 회귀를 확장하여 분류(classification) 모델을 만들어보겠습니다. 분류 모델은 정해진 기준에 맞게 데이터를 분류합니다. 예를 들면 암 종양을 분류하는 모델은 ‘암 종양인지 아닌지’를 분류합니다. 이런 모델은 특별히 ‘이진 분류’라고 부르기도 하지요. 이 장에서는 이진 분류를 위한 로지스틱 회귀(logistic regression)를 알아볼 것입니다. 로지스틱 회귀를 이해하고 나면 딥러닝에 한층 더 가까이 다가간 여러분을 발견할 수 있을 것입니다. 그러면 이제 커피 한 잔을 옆에 두고 천천히 시작해 보겠습니다.

04-1 초기 인공지능 알고리즘을 알아봅니다

04-2 시그모이드 함수를 알아봅니다

04-3 로지스틱 함수를 알아봅니다

04-4 분류용 데이터 세트를 준비합니다

04-5 로지스틱 회귀로 모델을 만들어봅니다

04-6 단일층 신경망을 만들어봅니다

04-7 사이킷런의 경사 하강법을 사용해봅니다

04-1 초기 인공지능 알고리즘을 알아봅니다

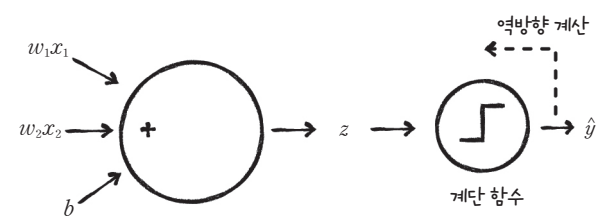
로지스틱 회귀를 제대로 이해하려면 로지스틱 회귀로 발전된 초창기 인공지능 알고리즘들을 순서대로 살펴보면 됩니다. 첫 번째로 알아볼 알고리즘은 ‘퍼셉트론’입니다.

퍼셉트론에 대해 알아봅니다

1957년 코넬 항공 연구소(Cornell Aeronautical Lab)의 프랑크 로젠블라트(Frank Rosenblatt)는 이진 분류 문제에서 최적의 가중치를 학습하는 퍼셉트론(Perceptron) 알고리즘을 발표하였습니다. 여기서 이진 분류란 임의의 샘플 데이터를 True나 False로 구분하는 문제를 말합니다. 예를 들어 과일이라는 샘플 데이터가 있을 때 사과인지(True) 아닌지(False) 판단하는 것이 이진 분류에 해당합니다.

퍼셉트론의 전체 구조를 훑어봅니다

퍼셉트론은 03장에서 공부한 선형 회귀와 유사한 구조를 가지고 있습니다. 직선의 방정식을 사용하기 때문이죠. 하지만 퍼셉트론은 마지막 단계에서 샘플을 이진 분류하기 위하여 계단 함수(step function)라는 것을 사용합니다. 그리고 계단 함수를 통과한 값을 통해 다시 가중치와 절편을 업데이트(학습)하는 데 사용합니다. 이를 그림으로 나타내면 다음과 같습니다.



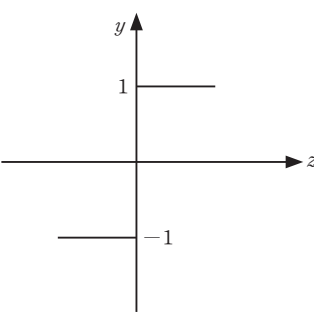
왼쪽의 큰 동그라미는 03장에서 살펴본 뉴런의 구조와 동일합니다(입력 신호가 1개 더 많아졌을 뿐입니다). 뉴런은 입력 신호들을 받아 z 를 만듭니다. 즉, 다음 수식에 의해 z 를 만드는 것이죠. 그리고 지금부터 다음 수식을 ‘선형 함수’라고 부르겠습니다.

$$w_1x_1 + w_2x_2 + b = z$$

그러면 계단 함수는 z 가 0보다 크거나 같으면 1로, 0보다 작으면 -1 로 분류합니다. 즉, 다음 연산을 진행합니다.

$$y = \begin{cases} 1 & (z > 0) \\ -1 & \end{cases}$$

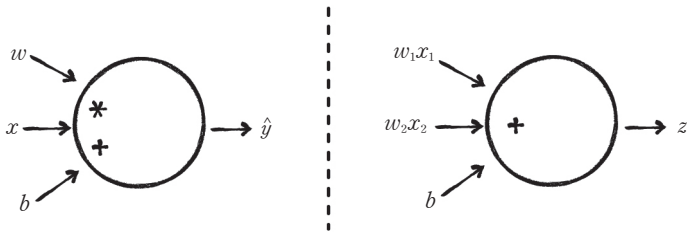
이때 1을 양성 클래스(positive class), -1 을 음성 클래스(negative class)라고 부르며 위의 함수를 그래프로 그리면 계단 모양이 됩니다. 그래서 계단 함수라고 부르게 된 것입니다.



이제 퍼셉트론이 어떤 구조로 되어 있는지 알 것 같나요? 쉽게 말해 퍼셉트론은 선형 함수를 통과한 값 z 를 계단 함수로 보내 0보다 큰지 작은지 검사하여 1과 -1 로 분류하는 아주 간단한 알고리즘입니다. 그리고 퍼셉트론은 계단 함수의 결과를 사용하여 가중치와 절편을 업데이트합니다.

여러 개의 특성을 사용합니다

그런데 여기서 잠깐! 그림을 보면 입력 신호에 특성이 2개 있습니다. 03장에서 본 뉴런에는 특성이 1개였죠.



03장에서는 그래프로 간단히 표현하기 위해 특성을 하나만 사용했지만 앞으로는 여러 특성을 사용하여 문제를 해결하는 경우가 많이 나오므로 오른쪽 그림에 익숙해져야 합니다. 앞에서 보았지만 특성이 2개인 경우 선형 함수를 다음과 같이 표기합니다. 아래 첨자로 사용한

숫자는 첫 번째 특성의 가중치(w_1)와 입력 데이터(x_1)를 의미합니다.

$$z = w_1x_1 + w_2x_2 + b$$

이를 응용하면 특성이 n 개인 선형 함수를 다음과 같이 표기할 수 있습니다.

$$z = w_1x_1 + w_2x_2 + \dots + w_nx_n + b$$

그러나 연구자들은 이렇게 장황한 식을 좋아하지 않습니다. 매번 쓰기도 번거롭죠. 그래서 덧셈을 축약해서 표현하는 시그마(Σ) 기호를 사용하여 위 식을 다음과 같이 표기합니다.

$$z = b + \sum_{i=1}^n w_i x_i$$

이때 상수 항(b)은 시그마 뒤가 아니라 앞에 두는 것이 좋습니다. 상수 항을 시그마 뒤에 두면 시그마 기호에 상수 항이 포함되었다고 착각할 수 있기 때문입니다.

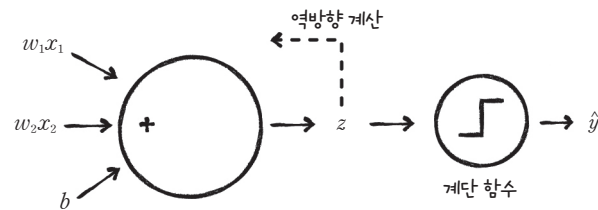
지금까지 퍼셉트론에 대해 알아보았습니다. 퍼셉트론은 사이킷런 패키지에서 **Perceptron**이라는 이름으로 클래스를 제공합니다. 퍼셉트론 다음에 등장한 알고리즘은 아달린입니다. 이번에는 아달린에 대해 알아보겠습니다.

Perceptron 클래스의 사용법은 이후 04-7에서 **SGDClassifier** 클래스를 설명할 때 자세히 설명하겠습니다.

아달린에 대해 알아봅니다

퍼셉트론이 등장한 이후 1960년에 스탠포드 대학의 버나드 위드로우(Bernard Widrow)와 테드 호프(Tedd Hoff)가 퍼셉트론을 개선한 적응형 선형 뉴런(Adaptive Linear Neuron)을 발표하였습니다. 적응형 선형 뉴런은 아달린(Adaline)이라고도 부르는데, 이 책에서는 적응형 선형 뉴런을 아달린이라고 부르겠습니다. 아달린은 선형 함수의 결과를 학습에 사용합니다. 계단 함수의 결과는 예측에만 활용하죠. 다음 그림을 보면서 조금 더 자세히 설명하겠습니다.

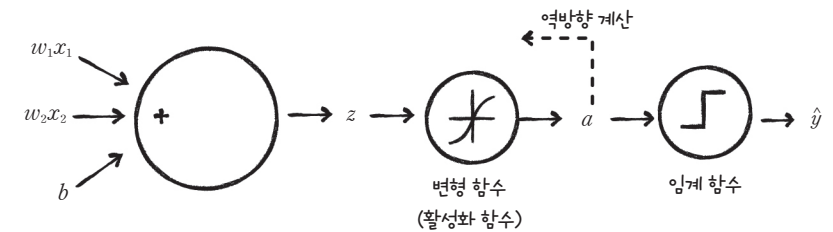
퍼셉트론은 계단 함수의 결과를 학습에 사용합니다.



역방향 계산이 계단 함수 출력 이후에 일어나지 않고 선형 함수 출력 이후에 진행되는 점에 주목하세요. 또 이 장에서 본격적으로 공부하게 될 로지스틱 회귀는 아달린의 개선 버전이므로 이 구조를 유심히 살펴보기 바랍니다. 아달린의 나머지 요소는 퍼셉트론과 동일하므로 여기까지만 설명하겠습니다.

로지스틱 회귀에 대해 알아봅니다

이제 로지스틱 회귀를 알아볼 차례입니다. 로지스틱 회귀는 아달린에서 조금 더 발전한 형태를 취하고 있습니다. 다음 그림을 보며 설명하겠습니다.



로지스틱 회귀는 선형 함수를 통과시켜 얻은 z 를 임계 함수에 보내기 전에 변형 함수로 변형시킵니다. 그리고 바로 이 변형 함수를 활성화 함수(activation function)라고 부릅니다. 활성화 함수를 통과한 값이 a 로 표현되어 있는데 앞으로 a 라고 하면 활성화 함수를 통과한 값이라고 이해하면 됩니다. 로지스틱 회귀는 마지막 단계에서 임계 함수(threshold function)를 사용하여 예측을 수행합니다. 임계 함수는 아달린이나 퍼셉트론의 계단 함수와 역할은 비슷하지만 활성화 함수의 출력값을 사용한다는 점이 다릅니다. 지금은 로지스틱 회귀의 전체 구조만 살펴봅니다. 자세한 내용은 다음 실습을 진행하며 다시 살펴보겠습니다.

활성화 함수는 비선형 함수를 사용합니다

그런데 활성화 함수로는 보통 비선형 함수를 사용합니다. 비선형 함수란 다음과 같이 생긴 함수를 말합니다.

〈수식 추가 예정〉

그런데 왜 활성화 함수는 비선형 함수를 사용할까요? 만약 활성화 함수가 선형 함수라면 어떻게 될까요? 예를 들어 선형 함수 $y = w_1x_1 + w_2x_2 + \dots + w_nx_n$ 과 활성화 함수 $\sim\sim\sim$ 가 있다고 생각해 봅시다. 그러면 이 둘을 쌓은 수식은 다음과 같다고 할 수 있습니다.

〈수식 추가 예정〉

위 식을 덧셈과 곱셈의 결합법칙과 분배법칙에 의하여 정리하면 다시 하나의 큰 선형 함수가 됩니다. 이렇게 되면 계단 함수 앞에 퍼셉트론(또는 아달린)을 여러 개 쌓아도 별 의미는 없습니다. 그래서 활성화 함수는 의무적으로 비선형 함수를 사용하게 된 것입니다. 그럼 로지스틱 회귀에는 어떤 활성화 함수가 사용되었을까요? 로지스틱 회귀의 활성화 함수는 바로 ‘시그모이드 함수’입니다.

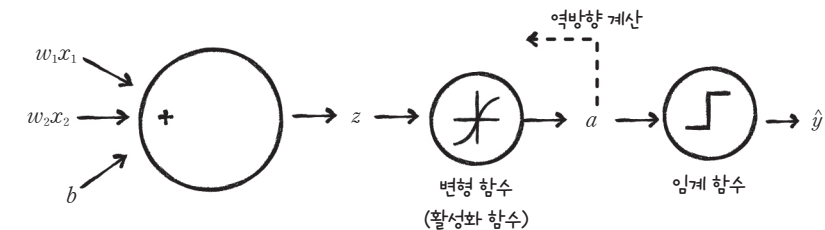
잠깐! 다음으로 넘어가려면

- ☒ 퍼셉트론은 선형 함수에 계단 함수만 추가한 것입니다.
- ☐ 퍼셉트론은 계단 함수의 결맞값으로 가중치를 학습합니다.
- ☐ 아달린은 퍼셉트론을 개선한 것으로, 선형 함수의 결맞값으로 가중치를 학습합니다.
- ☐ 로지스틱 회귀는 아달린에 활성화 함수(계단 함수)를 추가한 것입니다.

04-2 시그모이드 함수를 알아봅니다

시그모이드 함수의 역할을 알아봅니다

이쯤에서 로지스틱 회귀의 전체 구조를 한 번 더 살펴보며 시그모이드 함수의 역할을 알아보겠습니다.



가장 왼쪽에 있는 뉴런이 선형 함수이고 선형 함수의 출력값 z 는 다음과 같았습니다.

$$z = b + \sum_{i=1}^n w_i x_i$$

그림에서 볼 수 있듯이 출력값 z 는 활성화 함수를 통과하여 a 가 됩니다. 이때 로지스틱 회귀에 사용할 활성화 함수인 시그모이드 함수는 z 를 0~1 사이의 확률값으로 변환시켜주는 역할을 합니다. 즉, 시그모이드 함수를 통과한 값 a 를 암 종양 판정에 사용하면 ‘양성 샘플일 확률 (악성 종양일 확률)’로 해석할 수 있습니다. 확률은 해석하기 나름이지만 여기서는 a 가 0.5(50%)보다 크면 양성 클래스, 그 이하면 음성 클래스라고 구분하겠습니다.

시그모이드 함수가 만들어지는 과정을 살펴봅니다

그러면 시그모이드 함수는 어떻게 만들까요? 시그모이드 함수는 약간의 수학 기교를 사용하여 만들 수 있습니다. 시그모이드 함수가 만들어지는 과정은 다음과 같습니다.

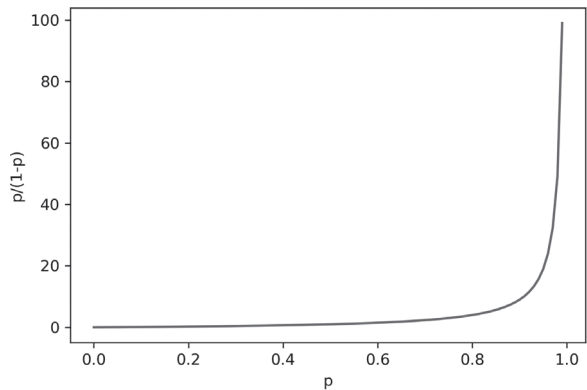
오즈 비 > 로짓 함수 > 시그모이드 함수

오즈 비에 대해 알아보까요?

시그모이드 함수는 오즈 비(odds ratio)라는 통계를 기반으로 만들어졌습니다(???). 오즈 비는 성공 확률과 실패 확률의 비율을 나타내는 통계이며 다음과 같이 정의합니다.

$$OR(odds\ ratio)=\frac{p}{1-p}(p=성공\ 확률)$$

오즈 비를 그래프로 그리면 다음과 같습니다. p 가 0부터 1까지 증가할 때 오즈 비의 값은 처음에는 천천히 증가하지만 p 가 1에 가까워지면 급격히 증가합니다. 오즈 비는 이정도로 간단히 살펴보고 넘어갑니다.



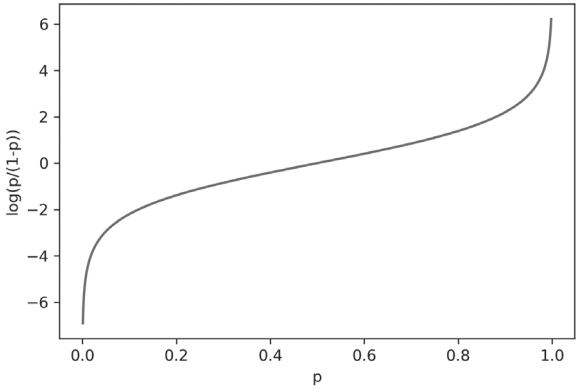
로짓 함수에 대해 알아보까요?

오즈 비에 로그 함수를 취하여 만든 함수를 로짓(logit) 함수라고 합니다. 로짓 함수의 식은 다음과 같습니다.

이 책에서는 자연 로그를 그냥 로그라고 부르겠습니다.

$$logit(p)=log(\frac{p}{1-p})$$

로짓 함수는 p 가 0.5일 때 0이 되고 p 가 0과 1일 때 각각 무한히 큰 음수와 양수가 되는 특징을 가집니다. 다음은 로짓 함수의 그래프입니다.




로짓 함수의 세로 축을 z 로, 가로 축을 p 로 놓으면 확률 p 가 0에서 1까지 변할 때 z 가 매우 큰 음수에서 매우 큰 양수까지 변하는 것으로 생각할 수 있습니다. 그러면 수식은 다음과 같이 전개할 수 있습니다.

$$log(\frac{p}{1-p})=z$$

로지스틱 함수에 대해 알아보까요?

위 식을 다시 z 에 대하여 정리하면 다음의 식이 됩니다. z 에 대해 정리하는 이유는 가로 축을 z 로 놓기 위해서입니다. 그리고 이 식을 로지스틱 함수라고 부릅니다.

$$p=\frac{1}{1+e^{-z}}$$



리키의 팁 메모 | 로지스틱 함수의 유도 과정이 궁금해요!

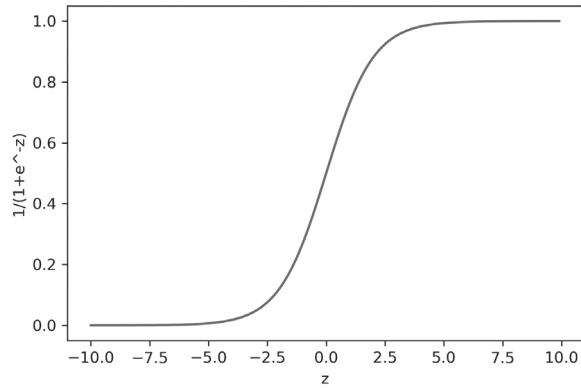
$$log(\frac{p}{1-p})=z$$

$$\frac{p}{1-p}=e^z$$

$$p(1+e^z)=e^z$$

$$p=\frac{e^z}{1+e^z}=\frac{1}{e^{-z}+1}$$

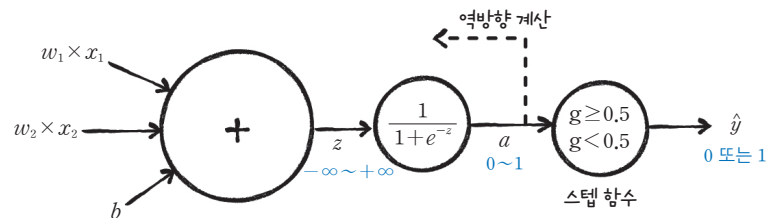
로지스틱 함수를 그래프로 그려보면 로짓 함수의 가로와 세로 축을 반대로 뒤집어 놓은 모양이 됩니다. 그리고 그래프는 S자 형태를 띄게 됩니다.



이 모양에서 착안하여 로지스틱 함수를 시그모이드(sigmoid) 함수라고도 부릅니다. 이 책에서는 시그모이드 함수라고 부르겠습니다. 이제 시그모이드 함수의 정체를 제대로 알았겠죠?

로지스틱 회귀 중간 정리하기

로지스틱 회귀에 필요한 설명을 모두 마쳤으니 이제는 다음 그림을 보며 로지스틱 회귀를 정리해 보겠습니다.



로지스틱 회귀는 이진 분류가 목표이므로 z 의 값을 조절할 방법이 필요했습니다. 그래서 시그모이드 함수를 활성화 함수로 사용한 것이죠. 이는 시그모이드 함수를 통과하면 z 를 확률처럼 해석할 수 있기 때문입니다. 그리고 시그모이드 함수의 확률인 a 를 0과 1로 구분하기 위하여 마지막에 임계 함수를 사용했습니다. 그 결과 입력 데이터(x), 가중치(w), 절편(b)은 0이나 1의 값으로 나뉘게 되었습니다. 즉, 이진 분류가 되었습니다. 드디어 로지스틱 회귀가 ‘이진 분류를 하기 위한 알고리즘’인 진짜 이유를 알았습니다. 그런데 아직 우리는 가중치와 절편을 적절하게 업데이트할 수 있는 방법을 배우지 않았습니다. 그렇다면 로지스틱 회귀에는 어떤 손실 함수를 사용해야 할까요? 선형 회귀에서 손실 함수로 제곱 오차를 사용했듯이 분류 문제에서도 제곱 오차를 사용할 수 있을까요? 이제 로지스틱 회귀를 위한 손실 함수인 로지스틱 손실 함수에 대해 알아보겠습니다.

04-3 로지스틱 손실 함수를 알아봅시다

선형 회귀는 정답과 예상치의 오차 제곱이 최소가 되는 가중치와 절편을 찾는 것이 주목적이었습니다. 그렇다면 분류의 목적은 무엇일까요? 분류는 올바르게 분류된 샘플 데이터의 비율 자체를 높이는 것이 목적입니다. 예를 들면 사과, 배, 감을 분류하는 문제에서 사과, 배, 감으로 분류한 과일 중 진짜 사과, 배, 감으로 분류한 비율을 높이는 것이 분류의 주목적입니다. 하지만 안타깝게도 올바르게 분류된 샘플의 비율은 미분 가능한 함수가 아니기 때문에 경사 하강법의 손실 함수로 사용할 수 없습니다. 대신 비슷한 목적을 달성할 수 있는 다른 함수를 사용해야 합니다. 바로 그 함수가 로지스틱 함수입니다.

로지스틱 손실 함수를 제대로 알아봅시다

로지스틱 손실 함수는 다중 분류를 위한 손실 함수인 크로스 엔트로피(cross entropy) 손실 함수를 이진 분류 버전으로 만든 것입니다. 크로스 엔트로피 손실 함수는 07장에서 다중 분류를 다룰 때 자세히 소개하겠습니다. 지금은 그냥 크로스 엔트로피 손실 함수를 이용하여 로지스틱 손실 함수를 만들었다는 사실만 기억하면 됩니다. 그리고 실무에서는 종종 이진 분류와 다중 분류를 구분하지 않고(어쨌든 분류이므로) 모두 크로스 엔트로피 손실 함수라고 부르는 경우도 많습니다. 하지만 이 책에서는 다중 분류와 이진 분류를 엄연히 구분하여 로지스틱 손실 함수라는 용어를 사용하겠습니다. 로지스틱 손실 함수는 다음과 같습니다. a 는 활성화 함수가 출력한 값이고 y 는 타깃입니다.

로지스틱 손실 함수에서 활성화 함수의 출력값을 표기할 때 a 대신 \hat{y} 를 쓰는 경우도 많습니다.

$$L = -(y \log(a) + (1-y) \log(1-a))$$

위 식은 어떻게 이해하면 좋을까요? 이진 분류는 그렇다(1), 아니다(0)라는 식으로 2개의 정답만 있습니다. 즉, 타깃의 값은 1또는 0입니다. 그러니 결국 위 식은 y 가 1이거나 0인 경우로 정리될 것입니다(???)

	L
y 가 1인 경우(양성 클래스)	$-\log(a)$
y 가 0인 경우(음성 클래스)	$-\log(1-a)$

그런데 앞 두 식의 값을 최소로 만들다 보면 a 의 값이 우리가 원하는 목표치가 된다는 것을 알 수 있습니다. 예를 들어 양성 클래스인 경우 로지스틱 손실 함수의 값을 최소로 만들려면 a 는 1에 자연스럽게 가까워집니다. 반대로 음성 클래스인 경우 로지스틱 손실 함수의 값을 최소로 만들려면 a 가 0에 자연스럽게 가까워집니다. 와! 로지스틱 손실 함수를 최소화하려니 a 의 값이 우리가 가장 이상적으로 생각할 수 있는 값이 되네요. 이제 로지스틱 손실 함수의 최솟값을 만드는 가중치와 절편을 찾기 위해 미분만 하면 되겠네요.

로지스틱 손실 함수 미분하기

로지스틱 손실 함수를 미분해 보겠습니다. 가중치와 절편에 대한 로지스틱 손실 함수의 미분 결과는 다음과 같습니다.

$$\frac{\partial}{\partial w_i}L=-(y-a)x_i$$

$$\frac{\partial}{\partial b}L=-(y-a)1$$

그런데 미분한 결과를 자세히 보면 \hat{y} 이 a 로 바뀌었을 뿐 제곱 오차를 미분한 결과와 동일합니다! 왼쪽이 03장에서 본 제곱 오차의 미분이고 오른쪽이 로지스틱 손실 함수의 미분입니다.

	제곱 오차 미분	로지스틱 손실 함수 미분
가중치에 대한 미분	$\frac{\partial SE}{\partial w} = -2(y-\hat{y})x$	$\frac{\partial}{\partial w_i}L=-(y-a)x_i$
절편에 대한 미분	$\frac{\partial SE}{\partial b} = -(y-\hat{y})1$	$\frac{\partial}{\partial b}L=-(y-a)1$

이쯤이면 로지스틱 회귀의 구현이 03장에서 만든 Neuron 클래스와 크게 다르지 않을 것이라는 기분이 듭니다. 어떻게 이런 식이 유도되는지 알아보겠습니다. 미분 유도 과정은 생각보다 쉽니다. 만약 내용을 읽어봐도 잘 이해가 되지 않는다면 바로 04-5로 넘어가도 괜찮습니다. 로지스틱 손실 함수의 미분을 통해 로지스틱 손실 함수의 값을 최소로 하는 가중치와 절편을 찾아야 한다는 점만 잊지 않으면 됩니다.

로지스틱 손실 함수와 연쇄 법칙

미분에서는 합성 함수의 도함수(미분한 함수)를 구하기 위한 방법인 연쇄 법칙(Chain Rule)이

있습니다. 예를 들어 다음과 같은 함수는

$$y=f(u), u=g(x)$$

아래와 같이 정리할 수 있는데

$$y=f(g(x))$$

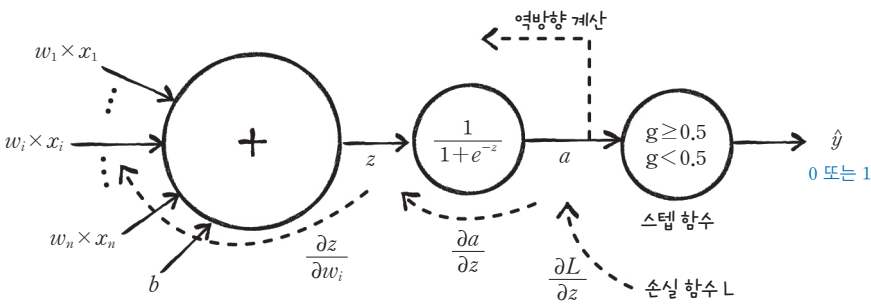
이때 y 를 x 에 대해 미분하는 방법은 y 를 u 에 대해 미분한 값과 u 를 x 에 대해 미분한 값을 곱하면 됩니다. 이것이 연쇄 법칙입니다.

$$\frac{\partial y}{\partial x}=\frac{\partial y}{\partial u}\frac{\partial u}{\partial x}$$

여기서 연쇄 법칙을 설명한 이유는 로지스틱 손실 함수(L)를 가중치(w)나 절편(b)에 대하여 바로 미분할 수 없기 때문입니다.

$$\frac{\partial L}{\partial w_i}=?, \frac{\partial L}{\partial b}=?$$

그런데 다음 그림을 살펴보면 연쇄 법칙으로 위의 곤란한 문제를 해결할 수 있다는 힌트를 얻을 수 있습니다.



그림을 보니 로지스틱 손실 함수(L)를 활성화 함수의 출력값(a)에 대하여 미분하고, 활성화 출력값(a)은 선형 함수의 출력값(z)에 대하여 미분하고, 활성화 출력값(z)은 가중치(w) 또는 절편(b)에 대하여 미분한 다음 서로 곱하면 결국 우리가 원하는 로지스틱 손실 함수를 가중치에 대하여 미분한 결과를 얻을 수 있다는 것을 알 수 있습니다. 그리고 이 과정은 그림의 오른쪽부터 왼쪽까지 역방향으로 진행된다는 것도 알 수 있습니다.

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial a} \frac{\partial a}{\partial z} \frac{\partial z}{\partial w_i}$$

로지스틱 손실 함수를 a 에 대하여 미분하기

그러면 이제 각각의 도함수를 구하기만 하면 됩니다. 먼저 로지스틱 손실 함수를 a 에 대하여 미분하겠습니다. 이때 y 는 a 의 함수가 아니므로 미분 기호 밖으로 뺄 수 있습니다.

$$\begin{aligned} \frac{\partial L}{\partial a} &= \frac{\partial}{\partial a} (-(y \log(a) + (1-y) \log(1-a))) \\ &= -(y \frac{\partial}{\partial a} \log(a) + (1-y) \frac{\partial}{\partial a} \log(1-a)) \end{aligned}$$

$\log(a)$ 을 a 에 대하여 미분하면 $\frac{1}{a}$ 이므로 위 식은 다음과 같이 간단하게 정리됩니다.

$$\frac{\partial L}{\partial a} = -(y \frac{1}{a} - (1-y) \frac{1}{1-a})$$

a 를 z 에 대하여 미분하기

이제 $\frac{\partial a}{\partial z}$ 를 계산해 보겠습니다. 여기에서 a 는 시그모이드 함수이므로 a 를 z 에 대한 식으로 표현할 수 있습니다. e^{-z} 를 z 에 대하여 미분하면 $-e^{-z}$ 가 되므로 다음과 같이 미분할 수 있습니다.

$$\begin{aligned} \frac{\partial a}{\partial z} &= \frac{\partial}{\partial z} \left(\frac{1}{1+e^{-z}} \right) = \frac{\partial}{\partial z} (1+e^{-z})^{-1} \\ &= -(1+e^{-z})^{-2} \frac{\partial}{\partial z} (e^{-z}) = -(1+e^{-z})^{-2} (-e^{-z}) = \frac{e^{-z}}{(1+e^{-z})^2} \end{aligned}$$

마지막으로 얻은 식을 두 덩어리의 분수 식으로 나눈 다음 공통 식을 묶어 정리하면 다음과 같은 식이 나옵니다.

$$\frac{\partial a}{\partial z} = \frac{1}{1+e^{-z}} \frac{e^{-z}}{1+e^{-z}} = \frac{1}{1+e^{-z}} \left(1 - \frac{1}{1+e^{-z}} \right) = a(1-a)$$

최종적으로 a 를 z 에 대하여 미분하면 다음과 같은 식이 나옵니다.

$$\frac{\partial a}{\partial z} = a(1-a)$$

z 를 w 에 대하여 미분하기

마지막으로 $\frac{\partial z}{\partial w_i}$ 를 구해 보겠습니다. z 는 선형 함수이므로 w_i 에 대해 미분하면 다른 항은 모두 사라지고 x_i 만 남아 $\frac{\partial z}{\partial w_i} = x_i$ 가 됩니다.

로지스틱 손실 함수를 w 에 대하여 미분하기

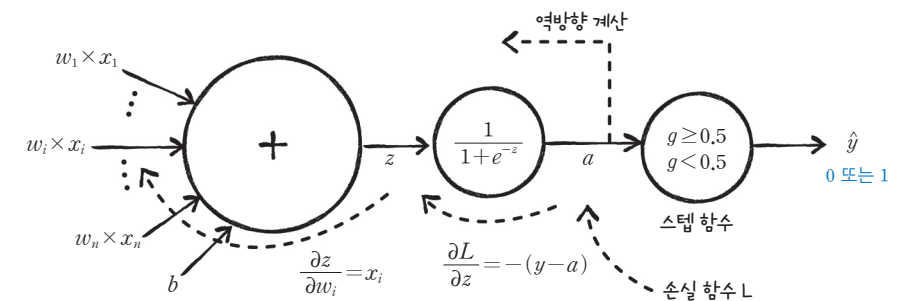
이제 연쇄 법칙을 위해 필요한 계산을 모두 마쳤습니다. 이제 각 단계에서 구한 도함수를 곱하기만 하면 됩니다.

$$\begin{aligned} \frac{\partial L}{\partial w_i} &= \frac{\partial L}{\partial a} \frac{\partial a}{\partial z} \frac{\partial z}{\partial w_i} \\ &= -(y \frac{1}{a} - (1-y) \frac{1}{1-a}) a(1-a) x_i = -(y(1-a) - (1-y)a) x_i \\ &= -(y - ya - a + ya) x_i = -(y - a) x_i \end{aligned}$$

결과를 보니 로지스틱 손실 함수를 w_i 에 대해 미분한 결과는 제곱 오차를 미분한 결과와 일치합니다.

로지스틱 손실 함수의 미분 과정 정리와 역전파 이해하기

다음은 지금까지 살펴본 미분 과정을 그림으로 나타낸 것입니다.



오른쪽부터 살펴보면 로지스틱 손실 함수 L 은 a 에 대하여 미분하고, a 는 z 에 대하여 미분하고, z 는 w 에 대하여 미분합니다. 그리고 각 도함수의 곱이 가중치 업데이트에 사용됩니다. 이렇게 로지스틱 손실 함수에 대한 미분이 연쇄 법칙에 의해 진행되는 구조를 보고 ‘그레이디언트 역전파된다’라고 말합니다. 이후 조금 더 복잡한 뉴런을 구현하게 되면 그레이디언트가 역전파되는 모습을 더 뚜렷하게 볼 수 있을 것입니다.

가중치 업데이트

로지스틱 회귀에서 가중치 업데이트는 가중치에서 로지스틱 손실 함수를 가중치에 대해 미분한 식을 빼면 됩니다.

$$w_i=w_i-\frac{\partial L}{\partial w_i}=w_i+(y-a)x_i$$

절편 업데이트

로지스틱 손실 함수를 절편에 대하여 미분하는 방법도 연쇄 법칙을 적용하면 쉽게 구할 수 있습니다. 이미 로지스틱 손실 함수를 z 에 대하여 미분한 도함수가 $-(y-a)$ 임을 알고 있으므로 다음과 같이 식을 전개할 수 있습니다.

$$\frac{\partial L}{\partial b}=\frac{\partial L}{\partial z}\frac{\partial z}{\partial b}=-(y-a)\frac{\partial}{\partial b}(b+\sum_{i=1}^nw_ix_i)=-(y-a)1$$

절편 업데이트 역시 절편에서 로지스틱 손실 함수를 절편에 대해 미분한 식을 빼면 됩니다.

$$b=b-\frac{\partial L}{\partial b}=b+(y-a)1$$

이제 가중치와 절편을 업데이트할 수 있는 방법을 모두 알았습니다. 이진 분류를 위한 클래스를 구현하는 것도 어렵지 않을 것입니다. 이제 본격적인 이진 분류를 위하여 분류용 데이터 세트를 준비하겠습니다. 이번에 사용할 데이터 세트는 위스콘신 유방암 데이터 세트입니다.

잠깐! 다음으로 넘어가려면

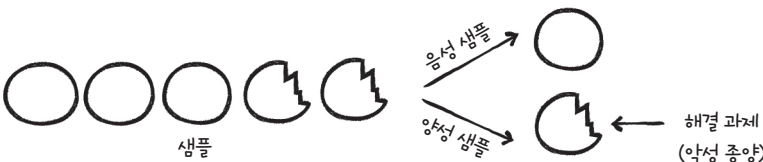
- ☒ 로지스틱 회귀는 가중치의 업데이트를 위해 로지스틱 손실 함수를 필요로 합니다.
- ☐ 로지스틱 손실 함수를 가중치에 대하여 미분하려면 연쇄 법칙을 사용해야 합니다.
- ☐ 연쇄 법칙을 사용한 모양새를 보고 ‘그레이디언트(기울기)가 역전파되었다’ 라고 표현합니다.

04-4 분류용 데이터 세트를 준비합니다

분류 문제를 위하여 데이터 세트를 준비해 보겠습니다. 데이터 세트는 사이킷런에 포함된 ‘위스콘신 유방암 데이터 세트(Wisconsin breast cancer dataset)’를 사용합니다.

유방암 데이터 세트를 소개합니다

유방암 데이터 세트에는 10개의 유방암 세포 특징에 대하여 평균, 표준 오차, 최대 이상치가 기록되어 있습니다. 여기서 해결할 문제는 유방암 데이터 샘플이 정상 종양(True)인지 혹은 악성 종양(False)인지를 구분하는 이진 분류 문제입니다.



그런데 여기서 주의할 점이 있습니다. 의학 분야에서는 건강한 종양을 양성 종양이라고 부르고 건강하지 않은 종양을 악성 종양(음성 종양이 아닙니다)이라고 부릅니다. 그런데 이진 분류 문제에서는 해결해야 할 목표를 양성 샘플이라고 부릅니다. 지금은 해결 과제가 악성 종양이므로 양성 샘플이 악성 종양인 샘플입니다. 양성이라는 긍정적인 단어 때문에 양성 샘플이 양성 종양이라고 착각할 수 있습니다. 그래서 이 책에서는 양성 종양 대신 정상 종양이라는 말을 사용하겠습니다. 표로 정리하면 다음과 같습니다.

	의학	이진 분류
좋은	양성 종양(정상 종양)	음성 샘플
나쁜	악성 종양	양성 샘플

유방암 데이터 세트 준비하기

1. load_breast_cancer() 함수 호출하기

이제 사이킷런에서 위스콘신 유방암 데이터 세트를 불러보겠습니다. 유방암 데이터 세트를 불러오려면 사이킷런의 datasets 모듈 아래에 있는 load_breast_cancer() 함수를 사용하면

됩니다. 이 함수를 호출해서 **Bunch** 클래스의 객체를 얻어오겠습니다.

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer( )
```

2. 입력 데이터 확인하기

Bunch 클래스로 객체를 만들어 **cancer**에 저장했으므로 **cancer**의 **data**와 **target**을 살펴보겠습니다. 먼저 입력 데이터인 **data**의 크기를 알아봅니다.

```
print(cancer.data.shape, cancer.target.shape)
(569, 30) (569,)
```


cancer에는 569개의 샘플과 30개의 특성이 있다는 것을 알 수 있습니다. 그러면 3개의 특성을 출력해 보겠습니다.

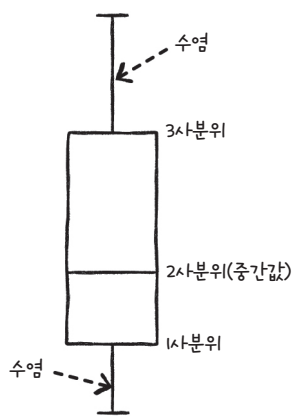
```
cancer.data[:3]
array([[1.799e+01, 1.038e+01, 1.228e+02, 1.001e+03, 1.184e-01, 2.776e-01,
        3.001e-01, 1.471e-01, 2.419e-01, 7.871e-02, 1.095e+00, 9.053e-01,
        8.589e+00, 1.534e+02, 6.399e-03, 4.904e-02, 5.373e-02, 1.587e-02,
        3.003e-02, 6.193e-03, 2.538e+01, 1.733e+01, 1.846e+02, 2.019e+03,
        1.622e-01, 6.656e-01, 7.119e-01, 2.654e-01, 4.601e-01, 1.189e-01],
       [2.057e+01, 1.777e+01, 1.329e+02, 1.326e+03, 8.474e-02, 7.864e-02,
        8.690e-02, 7.017e-02, 1.812e-01, 5.667e-02, 5.435e-01, 7.339e-01,
        3.398e+00, 7.408e+01, 5.225e-03, 1.308e-02, 1.860e-02, 1.340e-02,
        1.389e-02, 3.532e-03, 2.499e+01, 2.341e+01, 1.588e+02, 1.956e+03,
        1.238e-01, 1.866e-01, 2.416e-01, 1.860e-01, 2.750e-01, 8.902e-02],
       [1.969e+01, 2.125e+01, 1.300e+02, 1.203e+03, 1.096e-01, 1.599e-01,
        1.974e-01, 1.279e-01, 2.069e-01, 5.999e-02, 7.456e-01, 7.869e-01,
        4.585e+00, 9.403e+01, 6.150e-03, 4.006e-02, 3.832e-02, 2.058e-02,
        2.250e-02, 4.571e-03, 2.357e+01, 2.553e+01, 1.525e+02, 1.709e+03,
        1.444e-01, 4.245e-01, 4.504e-01, 2.430e-01, 3.613e-01, 8.758e-02]])
```

특성 데이터를 살펴보면 실수 범위의 값이고 양수와 음수가 섞여 있음을 알 수 있습니다. 괄호 1쌍으로 묶은 것이 특성 데이터 1개입니다. 특성을 세어보니 30개나 되네요. 산점도로 그려서 표현하기가 어려울 것 같습니다. 이번에는 산점도가 아니라 박스 플롯(box plot)을 이용하여 각 특성의 사분위(quartile) 값을 나타내 보겠습니다. 그리고 앞으로 맷플롯립 패키지(import matplotlib.pyplot)와 넘파이 패키지(import numpy)의 임포트는 생략하겠습니다. 그리

고 각각의 패키지는 줄임 표현(plt, np)을 사용하겠습니다.

3. 박스 플롯으로 특성의 사분위 관찰하기

박스 플롯은 1사분위와 3사분위 값으로 상자를 그린 다음 그 안에 2사분위(중간값) 값을 표시합니다. 그런 다음 1사분위와 3사분위 사이 거리(interquartile range)의 1.5배만큼 위아래 거리에서 각각 가장 큰 값과 가장 작은 값까지 수염을  박스 플롯은 상자 그래프 또는 상자 수염 그래프(box-and-whisker plot)라고도 부릅니다. 그립니다.



그러면 과정 1을 통해 얻어낸 데이터 세트를 이용하여 박스 플롯을 그려보겠습니다.

```
plt.boxplot(cancer.data)
plt.xlabel(' feature')
plt.ylabel('value')
plt.show( )
```

4. 눈에 띄는 특성 살펴보기

박스 플롯을 보면 3, 13, 23번째 특성이 다른 특성보다 값의 분포가 훨씬 크다는 것을 알 수 있습니다. 그러면 특성을 확인해 보겠습니다. 3, 13, 23번째 특성 인덱스를 리스트로 묶어 전달하면 각 인덱스의 특성을 확인할 수 있습니다. 결과를 보니 모두 넓이와 관련된 특성이네요. 세포의 넓이(???)가 유난히 크면(???) 유방암을 의심할 수 있겠죠. 로지스틱 회귀를 통해 이런 값을 골라내면 되는 것입니다(???)

```
cancer.feature_names[[3,13,23]]
array(['mean area', 'area error', 'worst area'], dtype='<U23')
```

5. 타깃 데이터 확인하기

여러분이 해결할 문제는 ‘음성 샘플(정상 종양)’과 ‘양성 샘플(악성 종양)’을 구분하는 이진 분류 문제입니다. 그래서 `cancer.target` 배열 안에는 0과 1만 들어 있습니다. 여기서 0은 음성 샘플, 1은 양성 샘플을 의미합니다(???)

다음은 타깃 데이터를 확인한 것입니다. 넘파이의 `unique()` 함수를 사용하면 고유한 값을 찾아 반환합니다. 이때 `return_counts` 매개변수를 `True`로 지정하면 고유한 값이 등장하는 횟수까지 세어 반환합니다.

```
np.unique(cancer.target, return_counts=True)
(array([0, 1]), array([212, 357]))
```

`unique()` 함수가 반환한 값을 확인해 보니 두 덩어리의 값을 반환하고 있습니다. 왼쪽의 값(`array([0, 1])`)은 `cancer.target`에 들어 있는 고유한 값(0, 1)을 의미합니다. 즉, `cancer.target`에는 0이나 1이라는 값만 들어 있습니다. 음성(0), 양성(1) 클래스의 값이므로 당연합니다. 오른쪽의 값(`array([212, 357])`)은 타깃 데이터의 고유한 값의 개수를 센 다음 반환한 것입니다. 즉, 위 타깃 데이터에는 212개의 음성 클래스(정상 종양)와 357개의 양성 클래스(악성 종양)가 들어 있습니다.

6. 훈련 데이터 세트 저장하기

이제 예제 데이터 세트를 `X, y` 변수에 저장하겠습니다. 여기서 `x`만 소문자가 아니라 대문자를 사용한 이유는 `cancer.data`가 2차원 배열(569, 30 크기)이기 때문입니다. `cancer.data`는 569개의 행(샘플)이 있고 30개의 열(특성)이 있습니다.

```
X = cancer.data
y = cancer.target
```

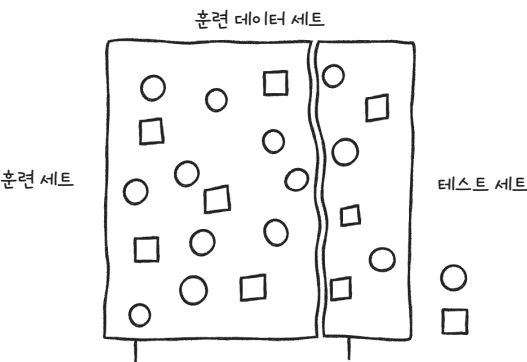
이제 훈련 데이터 세트 준비를 마쳤으니 로지스틱 회귀를 이용하여 모델을 만들어보겠습니다.

04-5 로지스틱 회귀로 모델을 만들어봅시다

03장에서는 훈련 데이터 세트 전체를 사용하여 모델을 훈련했습니다. 그런데 여기서 잠시 생각해 볼 주제가 있습니다. 훈련된 모델은 실전에서 얼마나 좋은 성능을 내는지 어떻게 알 수 있을까요? 여러분이 만든 모델의 성능을 평가하지 않고 실전에 투입하면 잘못된 결과를 초래할 수도 있으니 위험합니다. 또 훈련 데이터 세트로 학습된 모델을 다시 훈련 데이터 세트로 평가하면 어떨까요? 만약 모델이 훈련 데이터 세트를 몽땅 외워버렸다면(실제 이런 종류의 알고리즘도 있습니다) 평가를 해도 의미가 없을 것입니다. 그러면 어떻게 해야 모델의 성능을 제대로 평가할 수 있을까요? 모델을 만들기 전에 성능 평가에 대해 잠시 알아보겠습니다.

모델의 성능 평가를 위한 훈련 세트와 테스트 세트

훈련된 모델의 실전 성능을 일반화 성능(*generalization performance*)이라고 부릅니다. 그런데 앞에서 말한 것처럼 모델을 학습시킨 훈련 데이터 세트로 다시 모델의 성능을 평가하면 그 모델은 당연히 좋은 성능이 나올 것입니다. 이런 성능 평가를 ‘과도하게 낙관적으로 일반화 성능을 추정한다’고 말합니다. 조금 딱딱한 표현이지만 종종 다른 자료에서도 이런 표현을 볼 수 있으므로 알아두기 바랍니다. 그러면 올바르게 모델의 성능을 측정하려면 어떻게 해야 할까요? 훈련 데이터 세트를 두 덩어리로 나누어 하나는 훈련에, 다른 하나는 테스트에 사용하면 됩니다. 이때 각각의 덩어리를 훈련 세트(*training set*)와 테스트 세트(*test set*)라고 부릅니다.



훈련 데이터 세트를 훈련 세트와 테스트 세트로 나눌 때는 다음 2가지 규칙을 지켜야 합니다.

훈련 데이터 세트를 훈련 세트와 테스트 세트로 나누는 규칙

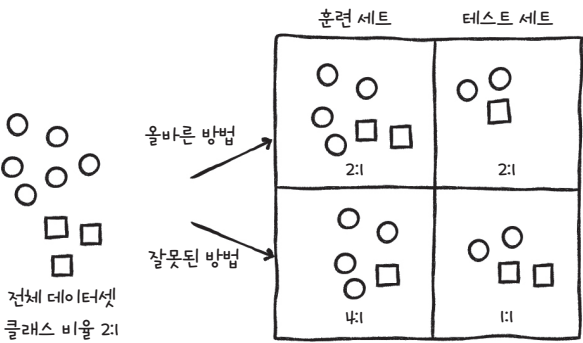
- 훈련 데이터 세트를 나눌 때는 테스트 세트보다 훈련 세트가 더 많아야 합니다.
- 훈련 데이터 세트를 나누기 전에 양성, 음성 클래스가 훈련 세트나 테스트 세트에 골고루 섞여야 합니다.

이 과정은 사이킷런에 준비되어 있는 도구를 사용하면 편리하게 진행할 수 있습니다. 그러면 지금부터 훈련 데이터 세트를 둘로 분리하겠습니다.

물론 여러분이 직접 훈련 데이터 세트를 섞고 나뉘도록 합니다. 하지만 데이터의 개수가 너무 많으면 어렵겠지요. 그래서 사이킷런의 힘을 빌리는 것입니다.

훈련 세트와 테스트 세트로 나누기

훈련 데이터 세트를 훈련 세트와 테스트 세트로 나눌 때는 양성, 음성 클래스가 훈련 세트와 테스트 세트에 고르게 분포하도록 만들어야 합니다. 예를 들어 *cancer* 데이터 세트를 보면 양성 클래스(악성 종양)와 음성 클래스(정상 종양)의 샘플 개수가 각각 212, 357개입니다. 이 클래스 비율이 훈련 세트와 테스트 세트에도 그대로 유지되어야 합니다. 만약 훈련 세트에 양성 클래스가 너무 많이 몰리거나 테스트 세트에 음성 클래스가 너무 많이 몰리면 모델이 데이터에 있는 패턴을 올바르게 학습하지 못하거나 성능을 잘못 측정할 수도 있습니다.



위 그림은 전체 데이터 세트의 클래스 비율이 2:1인 경우 제대로 나뉜 경우와 잘못 나뉜 경우를 보여줍니다. 위의 올바른 방법에 해당되는 경우를 보면 각 세트의 클래스 비율은 여전히 2:1입니다. 하지만 아래의 잘못된 방법에 해당되는 경우를 보면 클래스 비율이 망가져 있습니다. 따라서 이렇게 나뉘지면 안 됩니다. 지금부터 양성 클래스와 음성 클래스의 비율은 일정하게 유지하면서 훈련 데이터 세트를 훈련 세트와 테스트 세트로 나누어보겠습니다.

1. train_test_split() 함수로 훈련 데이터 세트 나누기

먼저 sklearn.model_selection 모듈에서 train_test_split() 함수를 임포트합니다. 사이킷런의 train_test_split() 함수는 입력된 훈련 데이터 세트를 훈련 세트 75%, 테스트 세트 25%의 비율로 나눠주는 함수입니다.

```
from sklearn.model_selection import train_test_split
```

그런 다음 train_test_split() 함수에 입력 데이터 X, 타깃 데이터 y와 그 밖의 설정을 매개변수로 지정하면 됩니다.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, test_size=0.2, random_state=42)
```

매개변수 설정에 대한 내용은 다음과 같습니다.

stratify=y
stratify는 일부 클래스의 양이 매우 적을 때 꼭 필요합니다. train_test_split() 함수는 기본적으로 양성 클래스와 음성 클래스의 비율을 보장해 주지만 클래스의 양이 너무 적으면 ~~~

test_size=0.2
앞에서 이야기했듯이 train_test_split() 함수는 훈련 데이터 세트를 75:25 비율로 나눕니다. 하지만 필요한 경우 이 비율을 조절하고 싶을 때도 있습니다. 그럴 때는 test_size 매개변수에 테스트 세트의 비율을 전달하면 비율을 조절할 수 있습니다. 여기에서는 입력된 데이터 세트의 20%를 테스트 세트로 나누기 위해 test_size에 0.2를 전달했습니다.

random_state=42
train_test_split() 함수는 무작위로 데이터 세트를 섞은 다음 나눕니다. 이 책에서는 섞은 다음 나눈 결과가 항상 일정하도록 random_state 매개변수에 난수 초깃값 42를 지정했습니다.

실험 결과를 똑같이 재현하기 위해 random_state 매개변수를 사용했습니다. 실전에서는 사용할 필요가 없습니다.

30은 유방암 세포의 특성 개수를 의미합니다.

2. 결과 확인하기

그러면 훈련 데이터 세트가 잘 나누어졌는지 훈련 세트와 테스트 세트의 비율을 확인해 보겠습니다. shape 속성을 통해 확인해 보니 각각의 훈련 세트와 테스트 세트는 4:1의 비율(455, 114)로 잘 나뉘었습니다.

```
print(X_train.shape, X_test.shape)
(455, 30) (114, 30)
```

3. unique() 함수로 훈련 세트의 타깃 확인하기

또 넘파이의 unique() 함수로 훈련 세트의 타깃 안에 있는 클래스의 개수를 확인해 보니 전체 훈련 데이터 세트의 클래스 비율과 거의 비슷한 구성입니다(양성 클래스가 음성 클래스보다 1.7배 정도 많습니다). 클래스의 비율이 그대로 유지되고 있네요.

```
np.unique(y_train, return_counts=True)
(array([0, 1]), array([170, 285]))
```

로지스틱 회귀 구현하기

이제 훈련 세트가 준비되었으니 본격적으로 로지스틱 회귀를 구현해 보겠습니다. 로지스틱 회귀는 정방향으로 데이터가 흘러가는 과정(정방향 계산)과 가중치를 업데이트하기 위해 역방향으로 데이터가 흘러가는 과정(역방향 계산)을 구현해야 합니다. 정방향 계산부터 역방향 계산까지 순서대로 구현해 보겠습니다. 여기서 만들 LogisticNeuron 클래스의 메서드는 03장에서 구현했던 Neuron 클래스의 __init__(), forpass(), backprop() 메서드와 거의 동일합니다. 다음을 따라 입력하세요.

```
class LogisticNeuron:

    def __init__(self):
        self.w = None
        self.b = None

    def forpass(self, x):
        z = np.sum(x * self.w) + self.b # 직선 방정식을 계산합니다.
        return z

    def backprop(self, x, err):
        w_grad = x * err # 가중치에 대한 그레이디언트를 계산합니다.
        b_grad = 1 * err # 절편에 대한 그레이디언트를 계산합니다.
        return w_grad, b_grad
```

이 코드는 03장과 크게 다르지 않습니다. 단, 몇 가지 세부 설정이 조금 바뀌었습니다. 다음은 각 메서드가 03장과 비교하여 어떻게 달라졌는지 설명한 것입니다.

`__init()` 메서드는 가중치와 절편을 미리 초기화하지 않습니다

여기서 `__init()` 메서드를 보면 입력 데이터의 특성이 많아 가중치를 미리 초기화하지 않았음을 알 수 있습니다. 가중치는 나중에 입력 데이터를 보고 특성 개수에 맞게 결정합니다.

`forpass()` 메서드에 넘파이 함수를 사용합니다

`forpass()` 메서드를 보면 가중치와 입력 특성의 곱을 모두 더하기 위해 `np.sum()` 함수를 사용한 것을 알 수 있습니다. `x * self.w`에서 `x`와 `w`는 1차원 넘파이 배열인데, 넘파이 배열에 사칙연산을 적용하면 자동으로 배열의 요소끼리 계산합니다. 예를 들어 다음과 같이 넘파이 배열 `[1, 2, 3]`과 `[3, 4, 5]`를 더하면 `[4, 5, 8]`이라는 넘파이 배열이 나오고 곱하면 `[3, 8, 15]`가 나옵니다.

```
a = np.array([1,2,3])
b = np.array([3,4,5])
a + b
array([4, 6, 8])
>>> a * b
array([ 3,  8, 15])
```

또 넘파이 배열을 `np.sum()` 함수의 인자로 전달하면 각 요소를 모두 더한 값을 반환합니다. 이 원리를 `forpass()` 메서드에 그대로 적용합니다.

```
np.sum(a * b)
26
```

로지스틱 뉴런이 구현되었습니다. 이제 훈련하고 예측하는 일만 남았습니다.

훈련하는 메서드 구현하기

훈련을 수행하는 `fit()` 메서드를 구현해 보겠습니다.

1. `fit()` 메서드 구현하기

`fit()` 메서드의 기본 구조는 03장의 `Neuron` 클래스와 같습니다. 다만 활성화 함수(`activation()`)가 추가된 점이 다릅니다. `activation()` 함수는 바로 다음에 완성합니다. 역방향 계산에는 로지스틱 손실 함수의 도

로지스틱 회귀에는 이진 분류를 위한 활성화 함수인 시그모이드 함수가 필요했습니다.

함수를 적용합니다. 앞에서 초기화하지 않은 가중치는 `np.ones()` 메서드를 이용하여 간단히 1로 초기화하고 절편은 간단히 0으로 초기화합니다.

🕒 `np.ones()` 메서드는 입력 특성과 동일한 크기의 배열을 만들고 값을 모두 1로 채웁니다.

```
def fit(self, x, y, epochs=100):
    self.w = np.ones(x.shape[1]) # 가중치를 초기화합니다.
    self.b = 0 # 절편을 초기화합니다.
    for i in range(epochs): # epochs만큼 반복합니다.
        for x_i, y_i in zip(x, y): # 모든 샘플에 대해 반복합니다.
            z = self.forpass(x_i) # 정방향 계산
            a = self.activation(z) # 활성화 함수 적용
            err = -(y_i - a) # 오차 계산
            w_grad, b_grad = self.backprop(x_i, err) # 역방향 계산
            self.w -= w_grad # 가중치 업데이트
            self.b -= b_grad # 절편 업데이트
```

2. `activation()` 메서드 구현하기

`activation()` 메서드에는 시그모이드 함수가 사용되어야 합니다. 시그모이드 함수는 넘파이의 `np.exp()` 메서드를 사용하면 간단히 만들 수 있습니다.

```
def activation(self, z):
    a = 1 / (1 + np.exp(-z)) # 시그모이드 계산
    return a
```



리키의 팁 메모 | 넘파이의 배열을 채울 수 있는 유용한 메서드를 알아볼까요?

넘파이에서는 배열을 만들 때 특정 값으로 채울 수 있는 몇 가지 메서드를 제공합니다. `np.ones()` 메서드도 그중 하나입니다. 다른 메서드도 알아볼까요? `np.zeros()` 메서드는 배열의 요소를 모두 0으로 채웁니다. 만약 `np.zeros()` 메서드로 다차원 배열을 만들려면 크기를 튜플 형식으로 전달하면 됩니다. 다음은 `np.zeros()` 메서드로 다차원 배열을 만든 예입니다.

```
np.zeros((2, 3))
array([[0., 0., 0.],
       [0., 0., 0.]])
```

0과 1이 아닌 임의의 값으로 배열을 생성하고 싶을 때는 `np.full()` 메서드를 사용해야 합니다. 다음은 요소의 값이 모두 7인 배열을 생성합니다.


```
np.full((2,3), 7)
array([[7, 7, 7],
       [7, 7, 7]])
```

예측하는 메서드 구현하기

03장의 선형 회귀는 입력값의 개수가 1개였으므로 새로운 샘플의 입력값에 대한 예측값을 계산하기가 쉬웠습니다(???). 그냥 훈련시킨 모델에 새로운 샘플의 입력값을 대입하기만 하면 됐죠. 하지만 유방암 데이터는 1개의 샘플 입력값이 30개나 됩니다. 30개나 되는 샘플의 입력값을 일일이 대입하면 번거롭겠죠(???). 그러니 새로운 샘플에 대한 예측값을 계산해 주는 메서드인 `predict()` 메서드가 필요합니다.

1. `predict()` 메서드 구현하기

`predict()` 메서드의 인자로 입력값 x 가 2차원 배열로 전달된다고 가정하고 구현하겠습니다. 예측값은 입력값을 선형 함수 > 활성화 함수 > 임계 함수 순서로 통과시키면 구할 수 있습니다. 앞에서 `forpass()`와 `activation()` 메서드를 이미 구현했으니 `predict()` 메서드는 다음과 같이 간단하게 만들 수 있습니다.

```
def predict(self, x):
    z = [self.forpass(x_i) for x_i in x] # 선형 함수 적용
    a = self.activation(np.array(z))     # 활성화 함수 적용
    return a > 0.5                      # 계단 함수 적용
```

여기서는 z 의 계산으로 파이썬의 리스트 내포(list comprehension) 문법을 사용했습니다. 리스트 내포란 대괄호(`[]`) 안에 `for`문을 삽입하여 새 리스트를 만드는 간결한 문법입니다. x 의 행을 하나씩 꺼내어 `forpass()` 메서드에 적용하고 그 결과를 이용하여 새 리스트(z)로 만든 것이죠. z 는 곧바로 넘파이 배열로 바꾸어 `activation()` 메서드에 전달합니다.

자, 이제 로지스틱 회귀가 구현되었습니다.

구현 내용 한눈으로 보기

다음은 지금까지 구현한 `LogisticNeuron` 클래스를 한눈에 볼 수 있도록 다시 정리한 것입니다.

```
class LogisticNeuron:

    def __init__(self):
        self.w = None
        self.b = None

    def forpass(self, x):
        z = np.sum(x * self.w) + self.b # 직선 방정식을 계산합니다.
        return z

    def backprop(self, x, err):
        w_grad = x * err                # 가중치에 대한 그레이디언트를 계산합니다.
        b_grad = 1 * err                # 절편에 대한 그레이디언트를 계산합니다.
        return w_grad, b_grad

    def activation(self, z):
        a = 1 / (1 + np.exp(-z))        # 시그모이드 계산
        return a

    def fit(self, x, y, epochs=100):
        self.w = np.ones(x.shape[1])   # 가중치를 초기화합니다.
        self.b = 0                      # 절편을 초기화합니다.
        for i in range(epochs):         # epochs만큼 반복합니다.
            for x_i, y_i in zip(x, y):  # 모든 샘플에 대해 반복합니다.
                z = self.forpass(x_i)   # 정방향 계산
                a = self.activation(z)   # 활성화 함수 적용
                err = -(y_i - a)         # 오차 계산
                w_grad, b_grad = self.backprop(x_i, err) # 역방향 계산
                self.w -= w_grad         # 가중치 업데이트
                self.b -= b_grad         # 절편 업데이트

    def predict(self, x):
        z = [self.forpass(x_i) for x_i in x] # 정방향 계산
        a = self.activation(np.array(z))     # 활성화 함수 적용
        return a > 0.5
```

로지스틱 회귀 모델 훈련시키기

이제 준비한 데이터 세트를 사용하여 로지스틱 회귀 모델을 훈련시켜 보고 정확도도 측정해 보겠습니다.

1. 모델 훈련하기

모델을 훈련하는 방법은 03장과 동일합니다. `LogisticNeuron` 클래스의 객체를 만든 다음 훈련 세트와 함께 `fit()` 메서드를 호출하면 됩니다.

```
neuron = LogisticNeuron()
neuron.fit(X_train, y_train)
```

2. 테스트 세트 사용해 모델의 정확도 평가하기

위 코드를 통해 훈련이 끝난 모델에 테스트 세트를 넣어 예측값을 넣고 예측한 값이 맞는지 비교합니다.

```
np.mean(neuron.predict(X_test) == y_test)
0.8245614035087719
```

`predict()` 메서드의 반환값은 `True`나 `False`로 채워진 `(m,)` 크기의 배열이고 `y_test`은 0이나 1로 채워진 `(m,)` 크기의 배열이므로 바로 비교할 수 있습니다. `np.mean()` 메서드는 인자로 전달한 비교문 결과 (넘파이 배열)의 평균을 계산합니다. 즉, 계산 결과 0.82...는 제대로 예측한 샘플의 비율이 됩니다. 이를 정확도 (accuracy)라고 합니다.

Ⓜ `X_test`와 `y_test`는 넘파이 배열이므로 `==` 연산자에 의해 각각의 요소를 비교합니다.

Ⓜ 파이썬은 `True`와 정수 1(또는 실수 1.0)은 같다고 판단합니다. 반대로 `False`는 1이 아닌 수와 같다고 판단합니다.

로지스틱 회귀를 제대로 구현한 것 같네요. 아주 훌륭하진 않지만 82%의 정확도가 나왔네요. 사실 이 모델은 성능이 좋은 편은 아닙니다. 실전에서 사용하려면 보통 ~~~%의 성능이 보장되어야 한다고 합니다. 실전에 사용하려면 사이킷런과 같은 안정적인 패키지를 사용하는 것이 바람직합니다. 지금은 학습을 위해 구현한 것이므로 성능은 신경 쓰지 않아도 됩니다.

이제 마지막으로 하나의 층(layer)을 가진 신경망을 구현해 보겠습니다.

잠깐! 다음으로 넘어가려면

- ☒ 전체 훈련 데이터 세트를 훈련 세트와 테스트 세트로 나눠야 모델을 훈련하고 평가할 수 있습니다.
- ☐ 훈련 세트와 테스트 세트로 나눌 때 훈련 데이터 세트의 클래스 비율이 그대로 유지되어야 합니다.
- ☐ 훈련 세트가 테스트 세트보다 더 많아야 합니다.

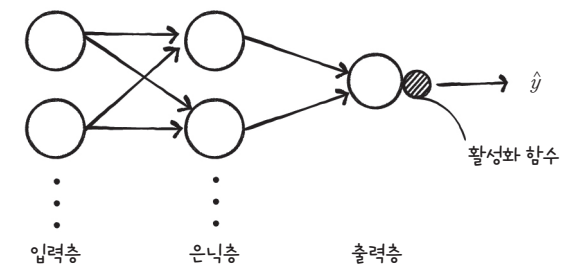
04-6 단일층 신경망을 만들어봅시다

사실 우리는 단일층 신경망을 구현했습니다! 로지스틱 회귀는 단일층 신경망(single layer neural network)과 동일하기 때문입니다. 하지만 지금까지는 층(layer)이라는 개념을 전혀 사용하지 않았습니다. 이제 신경망과 관련된 개념을 정리할 때가 된 것 같네요. 이 절을 읽고 나면 진짜로 신경망 알고리즘이란 무엇인지 알게 될 것입니다.

층 개념 이해하기

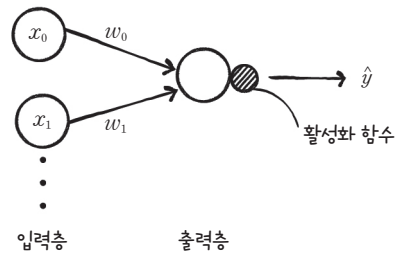
일반적인 신경망의 모습 알아보기

일반적으로 신경망은 다음과 같이 표현합니다. 여기서 가장 왼쪽이 입력층(input layer), 가장 오른쪽이 출력층(output layer) 그리고 가운데 층들을 은닉층(hidden layer)이라고 부릅니다. 또 출력층 오른쪽에 작은 원으로 표시된 활성화 함수는 출력층과 한 부분으로 생각합니다.



단일층 신경망의 모습 알아보기

그런데 지금까지 공부한 선형 회귀나 로지스틱 회귀는 은닉층이 없는 신경망 구조였습니다. 이런 입력층과 출력층만 가지는 신경망을 단일층 신경망이라고 부릅니다. 여기서 2개의 층 (입력층과 출력층)이 있는데도 단일층이라는 말을 사용하는 이유는 입력층은 입력 데이터 그 자체나 마찬가지로 실제 프로그램으로 구현할 때는 드러나지 않아 없는 것으로 간주하기 때문입니다. 이런 표현 방식이 처음에는 이해되지 않을 수 있겠지만 오랫동안 많은 사람들이 사용하면서 굳어진 용어이므로 실전을 위해서는 용어를 이해하고 넘어가는 것이 좋습니다. 다음은 단일층 신경망을 그린 것입니다. 이미 로지스틱 회귀를 공부하며 전체 구성 요소에 대해 공부했으므로 형태만 눈에 익히고 넘어갑시다.



단일층 신경망 구현하기

앞에서 구현한 `LogisticNeuron` 클래스는 단일층 신경망이므로 학습을 위해 단일층 신경망을 또 구현할 필요는 없습니다. 하지만 여기서 단일층 신경망을 다시 구현하는 이유는 몇 가지 유용한 기능을 추가하기 위해서입니다. 예를 들어 선형 회귀나 로지스틱 회귀는 모두 경사 하강법을 사용했습니다. 경사 하강법은 손실 함수(제곱 오차, 로지스틱 손실 함수)의 결괏값을 최소화하는 방향으로 가중치를 업데이트했죠. 만약 손실 함수의 결괏값이 줄어들지 않는다면 뭔가 잘못된 것이니 그 값을 관찰해 보기 바랍니다. 여기서는 이런 여러 기능들을 추가해 보겠습니다. `LogisticNeruon` 클래스를 기반으로 다음 코드를 작성하기 바랍니다.

손실 함수의 결괏값 조정해 저장하는 기능 추가하기

`__init__()` 메서드에 손실 함수의 결괏값을 저장할 리스트 `self.losses`를 만듭니다. 그런 다음 샘플마다 손실 함수를 계산하고 그 결괏값을 모두 더한 다음 샘플 개수로 나눈 평균값을 `self.losses` 변수에 저장합니다. 그리고 `self.activation()` 함수로 계산한 a 는 `np.log()`의 계산을 위해 한 번 더 조정합니다. 왜냐하면 a 가 0에 가까워지면 `np.log()` 함수의 값은 음의 무한대가 되고 a 가 1에 가까워지면 `np.log()` 함수의 값은 0이 되기 때문입니다. 손실값이 무한해지면 정확한 계산을 할 수 없으므로 a 의 값이 $-1 \times 10^{-10} \sim -1 \times 10^{10}$ 사이가 되도록 `np.clip()` 메서드로 조정해야 합니다.

```
def __init__(self):
    self.w = None
    self.b = None
    self.losses = []
    ...

def fit(self, x, y, epochs=100):
    ...
    for i in index:
        z = self.forpass(x[i])
        # 이 부분은 잠시 후에 설명합니다.
        # 모든 샘플에 대해 반복합니다.
        # 정방향 계산
```

```
a = self.activation(z) # 활성화 함수 적용
err = -(y[i] - a) # 오차 계산
w_grad, b_grad = self.backprop(x[i], err) # 역방향 계산
self.w -= w_grad # 가중치 업데이트
self.b -= b_grad # 절편 업데이트
# 안전한 로그 계산을 위해 클리핑한 후 손실을 누적합니다.
a = np.clip(a, 1e-10, 1-1e-10)
loss += -(y[i]*np.log(a)+(1-y[i])*np.log(1-a))
# 에포크마다 평균 손실을 저장합니다.

self.losses.append(loss/len(y))
```

미니 배치 경사 하강법 적용해 `fit()` 메서드 개선하기

지금까지 사용한 경사 하강법은 샘플 데이터 1개에 대한 그레이디언트를 계산했습니다. 이를 확률적 경사 하강법(stochastic gradient descent)이라고 부릅니다.

그러면 확률적 경사 하강법만 있을까요? 아닙니다. 전체 훈련 세트를 사용하여 한 번에 그레이디언트를 계산하는 방식은 배치 경사 하강법(batch gradient descent)이라 하고 배치(batch) 크기를 작게 하여(훈련 세트를 여러 번 나누어) 처리하는 방식은 미니 배치 경사 하강법(mini-batch gradient descent)이라고 합니다.

확률적 경사 하강법은 샘플 데이터 1개마다 그레이디언트를 계산하여 가중치를 업데이트하므로 계산 비용은 적은 대신 가중치가 최적값에 수렴하는 과정이 불안정합니다. 반면에 배치

경사 하강법은 전체 훈련 데이터 세트를 사용하여 한 번에 그레이디언트를 계산하므로 가중치가 최적값에 수렴하는 과정은 안정적이지만 그만큼 계산 비용이 많이 듭니다. 바로 이 둘의 장점을 절충한 것이 미니 배치 경사 하강법입니다. 그런데 확률적 경사 하강법과 미니 배치 경사 하강법은 에포크마다 훈련 세트를 사용하여 그레이디언트를 계산합니다. 이때 훈련 세트를 동일한 순서대로 계산하면 그레이디언트의 업데이트 방향에 무작위성이 줄어듭니다. 그러면 최적값의 탐색 과정이 다양하지 않아 최적값에 수렴하기 어렵습니다.

최적값 탐색 과정에 다양성을 부여하려면 에포크마다 훈련 세트를 다시 섞는 것이 좋습니다. 훈련 세트를 섞는 전형적인 방법은 넘파이 배열의 인덱스를 섞은 후 인덱스 순서대로 샘플을 뽑는 것입니다. 쉽게 말해 번호표를 따로 섞은 다음 번호표 순서대로 훈련 세트를 나열하는 것입니다. 이 방법이 훈련 세트 자체를 섞는 것보다 효율적이고 빠릅니다.

`np.random.permutation()` 메서드를 사용하면 이 방법을 구현할 수 있습니다. 다음 코드에서 두 번째 `for`문을 보면 `zip(Xb, y)` 메서드를 반복할 때 `indexes` 배열을 이용합니다. `indexes` 배열에 `[6, 2, 9, ...]`와 같은 무작위 번호표가 들어 있다고 생각하면 됩니다.

☺ 정방향 계산과 오차를 계산할 때 이 인덱스를 사용하여(`Xb[i]`, `y[i]`) 샘플을 참조합니다.

```
def fit(self, x, y, epochs=100):
    self.w = np.ones(x.shape[1]) # 가중치를 초기화합니다.
    self.b = 0 # 절편을 초기화합니다.
    for i in range(epochs): # epochs만큼 반복합니다.
        loss = 0
        indexes = np.random.permutation(np.arange(len(x))) # 인덱스를 섞습니다.
        for i in indexes: # 모든 샘플에 대해 반복합니다.
            z = self.forpass(x[i]) # 정방향 계산
            a = self.activation(z) # 활성화 함수 적용
            err = -(y[i] - a) # 오차 계산
            w_grad, b_grad = self.backprop(x[i], err) # 역방향 계산
            self.w -= w_grad # 가중치 업데이트
            self.b -= b_grad # 절편 업데이트
            a = np.clip(a, 1e-10, 1-1e-10) # 안전한 로그 계산을 위해 클리핑한 후 손실을 누적합니다.
        loss += -(y[i]*np.log(a)+(1-y[i])*np.log(1-a)) # 에포크마다 평균 손실을 저장합니다.
    self.losses.append(loss/len(y))
```

score() 메서드 추가하기

마지막으로 정확도를 계산해 주는 `score()` 메서드를 추가하고 `predict()` 메서드도 조금 수정하겠습니다. `score()` 메서드는 정확도를 직접 계산할 때 사용했던 `np.mean()` 메서드를 사용합니다.

```
def predict(self, x):
    z = [self.forpass(x_i) for x_i in x] # 정방향 계산
    return np.array(z) > 0 # 계단 함수 적용

def score(self, x, y):
    return np.mean(self.predict(x) == y)
```

시그모이드 함수의 출력값은 0~1 사이의 확률값이고 양성 클래스를 판단하는 기준은 0.5 이상입니다. 그런데 `z`가 0보다 크면 시그모이드 함수의 출력값은 0.5보다 크고 `z`가 0보다 작으면 시그모이드 함수의 출력값은 0.5보다 작습니다. 그래서 `predict()` 메서드에는 굳이 시그모이드 함수를 사용하지 않아도 됩니다. `z`가 0보다 큰지 작은지만 따지면 되기 때문이죠. 그래서 `predict()` 메서드에는 로지스틱 함수를 적용하지 않고 `z` 값의 크기만 비교하여 결과를 반환했습니다.

이제 단일층 신경망 클래스가 완성되었습니다. 전체 코드는 다음과 같습니다. 이 클래스를 cancer 데이터 세트에 적용해 보겠습니다.

```
class SingleLayer:

    def __init__(self):
        self.w = None
        self.b = None
        self.losses = []

    def forpass(self, x):
        z = np.sum(x * self.w) + self.b          # 직선 방정식을 계산합니다.
        return z

    def backprop(self, x, err):
        w_grad = x * err                        # 가중치에 대한 그레이디언트를 계산합니다.
        b_grad = 1 * err                        # 절편에 대한 그레이디언트를 계산합니다.
        return w_grad, b_grad

    def add_bias(self, X):
        return np.c_[np.ones((X.shape[0], 1)), X] # 행렬의 맨 앞에 1로 채워진 열 벡터를 추가
                                                    # 합니다.

    def activation(self, z):
        a = 1 / (1 + np.exp(-z))                # 시그모이드 계산
        return a

    def fit(self, x, y, epochs=100):
        self.w = np.ones(x.shape[1])            # 가중치를 초기화합니다.
        self.b = 0                              # 절편을 초기화합니다.
        for i in range(epochs):                  # epochs만큼 반복합니다.
            loss = 0

            # 인덱스를 섞습니다.
            indexes = np.random.permutation(np.arange(len(x)))
            for i in indexes:                    # 모든 샘플에 대해 반복합니다.
                z = self.forpass(x[i])          # 정방향 계산
                a = self.activation(z)           # 활성화 함수 적용
                err = -(y[i] - a)               # 오차 계산
                w_grad, b_grad = self.backprop(x[i], err) # 역방향 계산
                self.w -= w_grad                # 가중치 업데이트
                self.b -= b_grad                # 절편 업데이트
                a = np.clip(a, 1e-10, 1-1e-10)  # 안전한 로그 계산을 위해 클리핑한 후 손실을
                                                    # 누적합니다.
```

```
        loss += -(y[i]*np.log(a)+(1-y[i])*np.log(1-a))
        self.losses.append(loss/len(y))         # 에포크마다 평균 손실을 저장합니다.

    def predict(self, x):
        z = [self.forpass(x_i) for x_i in x]    # 정방향 계산
        return np.array(z) > 0                 # 스텝 함수 적용

    def score(self, x, y):
        return np.mean(self.predict(x) == y)
```

단일층 신경망 훈련하기

1. 단일층 신경망 훈련하고 정확도 출력하기

이전과 마찬가지로 SingleLayer 객체를 만들고 훈련 세트(X_train, y_train)로 이 신경망을 훈련시킨 다음 score() 메서드로 정확도를 출력해 보겠습니다.

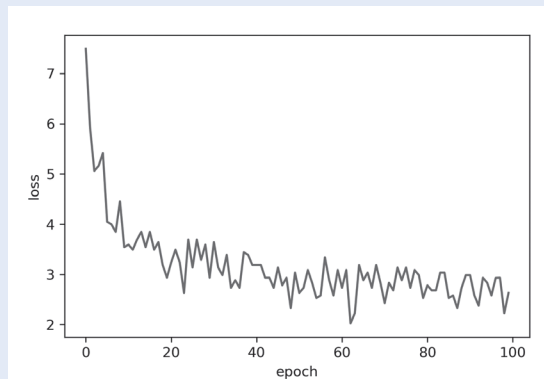
```
layer = SingleLayer( )
layer.fit(X_train, y_train)
layer.score(X_test, y_test)
0.9298245614035088
```

정확도가 훨씬 좋아졌네요! LogisticNeuron과 마찬가지로 fit() 메서드의 에포크(epochs) 매개변수의 기본값 100을 그대로 사용했는데도 이렇게 성능이 좋아진 이유는 무엇일까요? 에포크마다 훈련 세트를 무작위로 섞어 손실 함수의 값을 줄였기 때문입니다.

2. 손실 함수 누적값 확인하기

정말 그런지 손실 함수의 값을 확인해 볼까요? layer 객체의 losses 속성에 손실 함수의 결과값을 저장했으므로 이 값을 그래프로 그려 확인해 보겠습니다.


```
plt.plot(layer.losses)
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show( )
```



그래프를 그려보니 로지스틱 손실 함수의 값이 에포크가 진행됨에 따라 감소하고 있음을 확인할 수 있습니다.

축하합니다! 성공적으로 가장 기초적인 신경망 알고리즘을 구현했습니다. 이 장에서 배운 것처럼 신경망 알고리즘은 로지스틱 회귀 알고리즘을 확장한 네트워크로 생각해도 좋습니다. 아직까지는 은닉층을 사용하지 않았기 때문에 이 단일층 신경망은 로지스틱 회귀나 퍼셉트론 알고리즘과 매우 비슷합니다. 지금까지는 선형 회귀, 로지스틱 회귀 등 신경망 알고리즘들을 직접 구현했습니다. 하지만 사이킷런에는 이런 알고리즘들이 미리 구현되어 있습니다. 사이킷런에 있는 SGDClassifier 클래스를 사용해보며 이 장을 마무리하겠습니다.

04-7 사이킷런의 경사 하강법을 사용해 봅니다

사이킷런의 경사 하강법이 구현된 클래스는 SGDClassifier입니다. 회귀 문제에는 SGDRegressor 클래스를 사용할 수 있는데, 이 클래스는 로지스틱 회귀 문제 외에도 여러가지 문제에 경사 하강법을 적용할 수 있어 앞으로 자주 사용하게 될 것입니다(???). 여기서는 SGDClassifier 클래스를 통해 로지스틱 회귀 문제를 간단히 해결해 보겠습니다.

사이킷런으로 경사 하강법 적용하기

1. 로지스틱 손실 함수 지정하기

SGDClassifier 클래스에 로지스틱 회귀를 적용하려면 loss 매개변수를 log로 지정하면 됩니다.

```
sgd = SGDClassifier(loss='log', max_iter=100, tol=1e-3, random_state=42)
```

나머지 매개변수도 간단히 알아볼까요? max_iter를 통해 반복 횟수를 100으로 지정하고 반복 실행을 했을 때 결과를 동일하게 재현하기 위해 random_state를 통해 난수 초깃값을 42로 설정합니다. 반복마다 로지스틱 손실 함수의 값이 tol에 지정한 값만큼 감소되지 않으면 반복을 중단하도록 설정합니다. 만약 tol의 값을 설정하지 않으면(???) max_iter의 값을 늘리라는 경고가 발생합니다. 이렇게 설정하면 모델의 로지스틱 손실 함수의 값이 최적값으로 수렴할 정도로 충분한 반복 횟수를 입력했는지 연구자에게 알려주므로 유용합니다.

2. 사이킷런으로 훈련하고 평가하기

사이킷런의 SGDClassifier 클래스에는 지금까지 우리가 직접 구현한 메서드가 이미 구현되어 있습니다. 이름도 비슷하죠. 사이킷런의 fit() 메서드로 훈련하고 score() 메서드로 정확도를 계산하면 됩니다.

```
sgd.fit(X_train, y_train)
sgd.score(X_test, y_test)
0.8333333333333334
```


3. 사이킷런으로 예측하기

마찬가지로 SGDClassifier 클래스에는 예측을 위한 predict() 메서드도 구현되어 있습니다. 예시를 위해 테스트 세트에 대한 예측을 만들어보겠습니다. 이때 주의할 점은 사이킷런은 입력 데이터로 2차원 배열만 받아들입니다. 즉, 샘플 하나를 주입하더라도 2차원 배열로 만들어야 합니다. 여기서는 테스트 세트에서 10개 샘플만 뽑아 예측을 만들어보겠습니다.

```
sgd.predict(X_test[0:10])
array([0, 1, 0, 0, 0, 0, 1, 0, 0, 0])
```

결과와 실제 타깃을 비교하면 성능도 측정할 수 있습니다. 실제 타깃과의 비교는 여러분이 직접 수행해 보기 바랍니다.



04장에서 꼭 기억해야 할 내용

이 장에서는 선형 회귀 알고리즘을 확장한 로지스틱 회귀를 배웠습니다. 두 알고리즘 모두 선형 함수를 사용합니다. 최적의 해를 찾는 방법은 여러 가지이지만 이전 장과 마찬가지로 경사 하강법을 사용했습니다. 로지스틱 회귀는 선형 함수의 결과를 로지스틱 함수를 통과시켜 0~1 사이의 값으로 압축합니다. 이를 확률로 해석하고 원하는 타깃에 맞도록 훈련시켰습니다.

기억 카드 01

?

로지스틱 회귀를 경사 하강법으로 풀기 위해 로지스틱 손실 함수 또는 이진 크로스 엔트로피 손실 함수를 사용합니다. 놀랍게도 오차를 역전파할 때 이진 크로스 엔트로피 손실 함수와 로지스틱 손실 함수를 통과한 미분 결과가 선형 회귀와 마찬가지로 매우 간단해 집니다. 타깃에서 활성화 함수를 통과한 값을 뺀 값이 오차 그래디언트가 됩니다.

기억 카드 02

?

초기 신경망 알고리즘은 퍼셉트론에서 출발하여 아달린으로 개선되었고 활성화 함수가 적용되어 현재의 모습을 갖추었습니다. 입력층과 출력층만 있는 신경망이 로지스틱 회귀와 매우 비슷합니다. 입력층은 사실 입력 데이터 그 자체입니다. 층이라 부르긴 하지만 특별한 계산이 수행되지 않고 개념적으로만 존재합니다.

기억 카드 03

?

다음 장에서는 머신러닝 작업을 수행할 때 꼭 익혀야 할 필수적인 개념과 모범 사례를 살펴보겠습니다. 그리고 나서 그 다음 장에서 은닉층이 추가된 신경망을 직접 구현해 보겠습니다. 또한 사이킷런과 딥러닝 패키지 케라스를 사용하여 신경망을 구현해 보겠습니다.