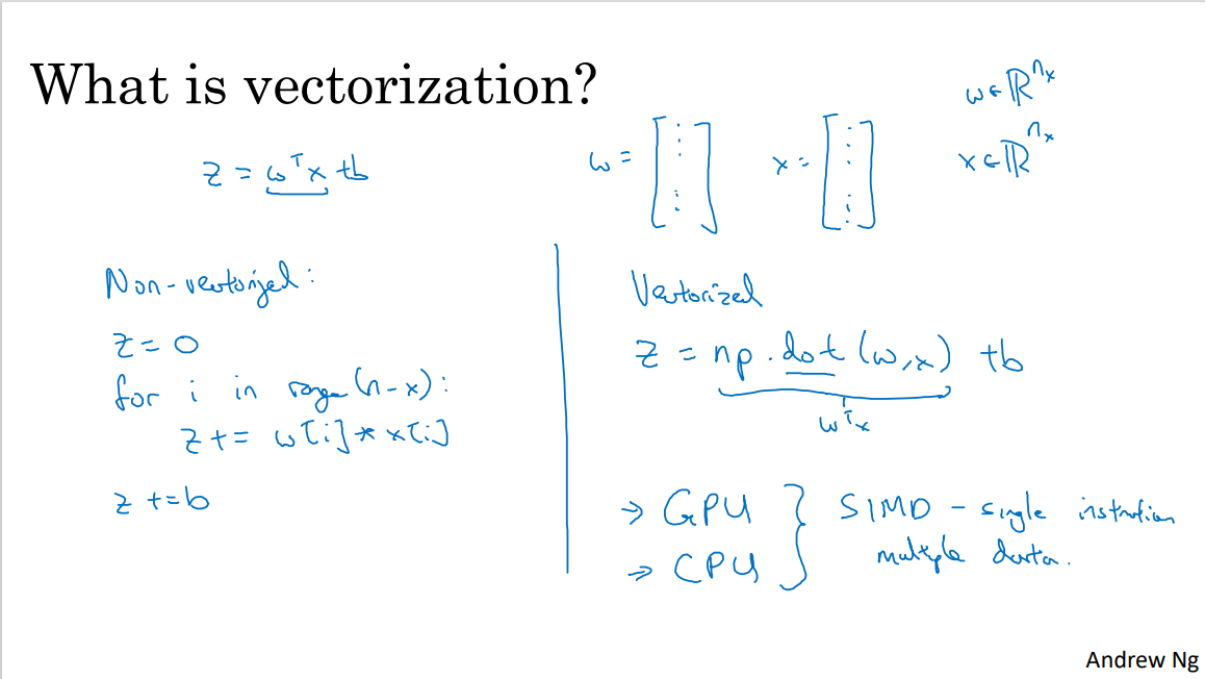
3. 파이썬과 벡터화

벡터화(Vectorization)

- 코드에서 for문 없앨 수 있음
- 결과를 빠르게 얻기 위해 큰 데이터를 학습시킬 때 코드가 빠르게 실행되는 게 중요 (아이디어 → 피드백 속도 빠름)



- 벡터화되지 않으면?
  - for문을 돌며 z += w[i] * x[i] 연산을 해주어야 함.
- 벡터화되면?
  - wTx 직접 계산!
  - np.dot(w, x) = wTx
  - 더 빠름

예시: Jupiter Notebook 코드

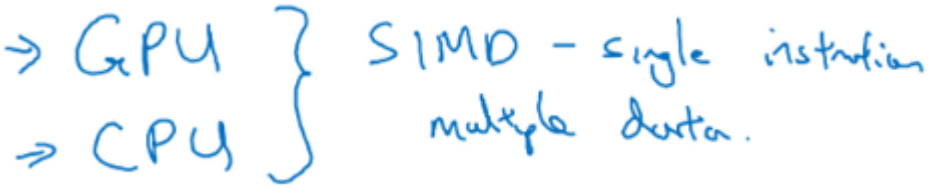
- Shift + Enter 코드 실행

```
import time
a = np.random.rand(1000000) # 랜덤 수로 이루어진 백만 차원의 배열
b = np.random.rand(1000000)

tic = time.time() # 계산 전 시간
c = np.dot(a, b)  # 벡터화
toc = time.time() # 계산 완료 시간

print("Vectorized: " + str(1000*(toc-tic)) + "ms")

c = 0
tic = time.time()
for i in range(1000000):
  c += a[i] * b[i]
toc = time.time()

print("For loop: " + str(1000*(toc-tic)) + "ms")
```

```
Vectorized: 1.47...ms
For loop: 474.29...ms
```

→ **벡터화하면 코드 실행 속도가 300배 가량 상승됨!**

그렇다면 속도에서 차이가 나는 이유는?



- GPU와 CPU에게는 **병렬 명령어: SIMD**(Single Instruction Multiple Data)가 있음
- GPU가 CPU 보다 SIMD 계산에 뛰어남

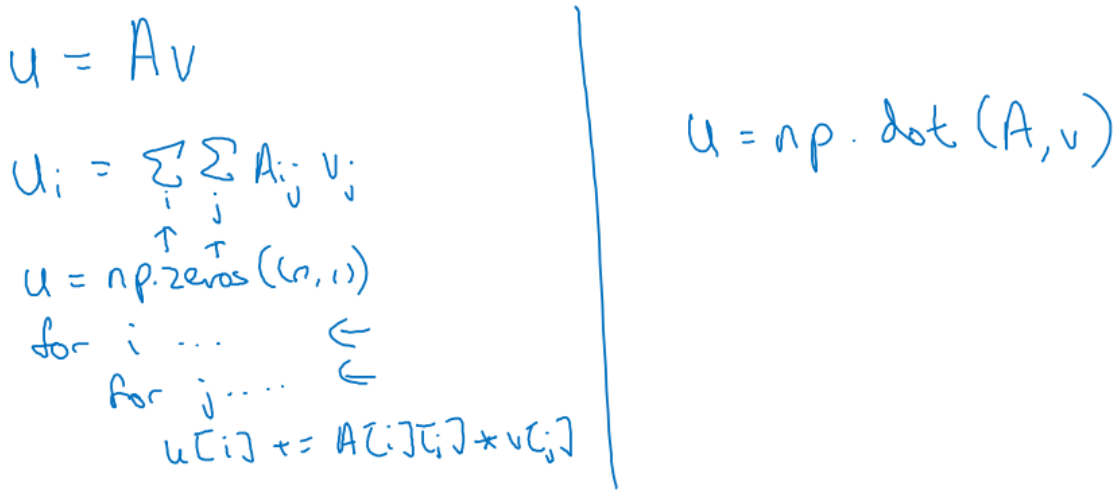
→ np.dot 또는 for문 대신 다른 함수를 사용할 때, **파이썬의 NumPy가 병렬화를 통해 계산을 훨씬 빠르게 수행**하게 해줌

→ **가능한, for문은 쓰지 말자!**

더 많은 벡터화 예제

1. 행렬 A와 벡터 v의 곱인 벡터 u 계산

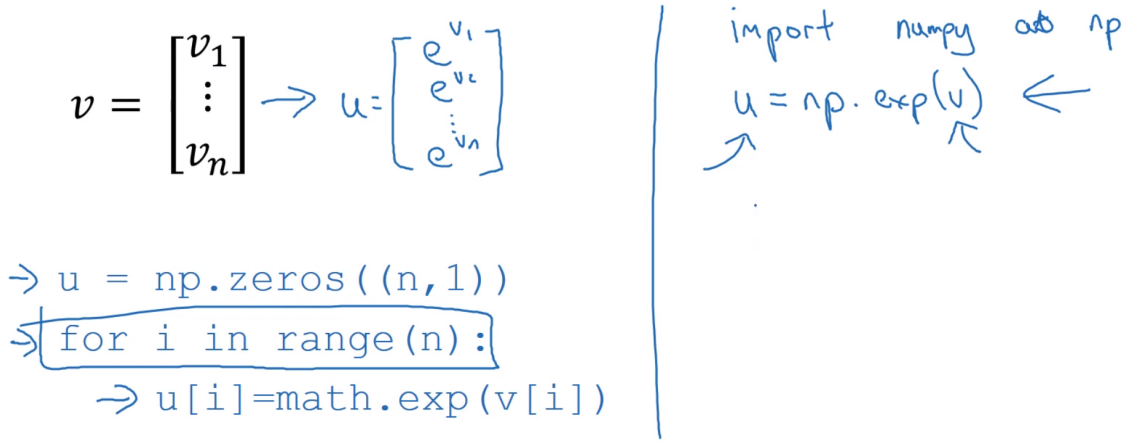
벡터화 X vs 벡터화 O



→ for문 2개를 없애므로 훨씬 빠르다!

2. 벡터 v의 모든 원소에 지수 연산

= 원소가 e^(v_1) 부터 e^(v_n)인 벡터 u 계산



- np.exp(v) → e^v
- np.exp(a, b) → a^b
- np.log(v) → 각 원소의 로그값 계산

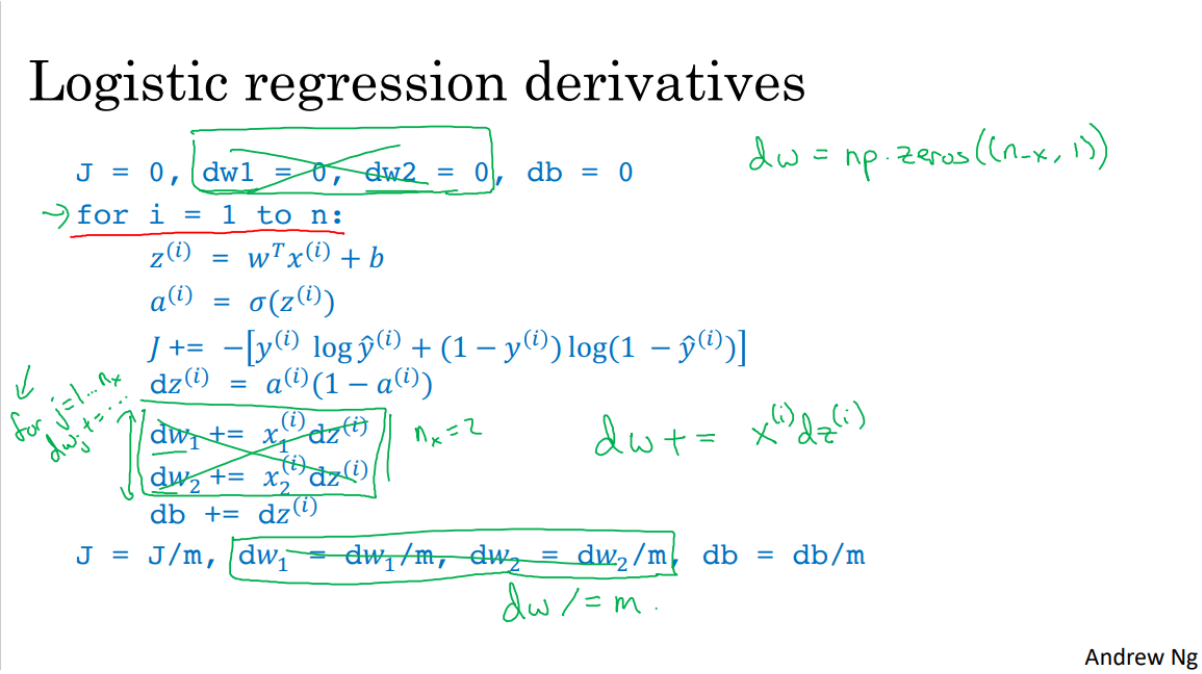
- np.abs(v) → 절댓값
- np.max(v, 0) → 원소와 0 중에서 더 큰 값 반환
- v**2 → 모든 원소 제곱
- 1/v → 모든 원소 역수

→ **for문을 쓰고 싶을 때, 해당하는 NumPy 내장 함수가 있는지 확인!**

Logistic Regression에 적용

- 두 개의 for문이 있었음.

1. 훈련 세트를 도는 for문
2. 입력 벡터의 차원 n만큼 도는 반복문 → 이걸 없앨 것!



코드에서 달라진 점

1. 초기화: dw들을 0으로 초기화하는 대신, n차원의 벡터로 만든다.

```
dw = np.zeros((n_x,1)) # n_x차원의 벡터
```

2. 반복되는 부분을 벡터 연산으로 대체한다.

```
dw += x^(i)*dz(i)
```

3. 평균을 구하는 부분도 간략하게 표현한다.

```
dw /= m
```

for문을 하나만 없애도 속도가 매우 올라가지만, **for문을 하나도 쓰지 않고 훈련 세트를 동시에 처리할 수 있는 법이 있음!**

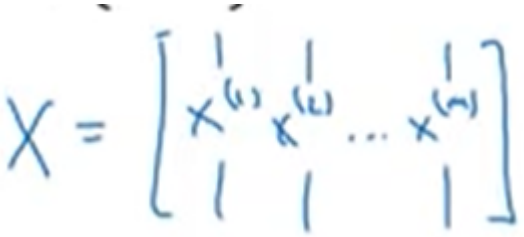
로지스틱 회귀의 벡터화

1. 정방향 전파 단계

$$z^{(1)} = w^T x^{(1)} + b \qquad z^{(2)} = w^T x^{(2)} + b \qquad z^{(3)} = w^T x^{(3)} + b$$
$$a^{(1)} = \sigma(z^{(1)}) \qquad a^{(2)} = \sigma(z^{(2)}) \qquad a^{(3)} = \sigma(z^{(3)})$$

m개의 훈련 샘플이 있다면, m번 동안 z와 활성값을 계산해야 함.

- X는 훈련 입력을 열로 쌓은 행렬 (n, m)차원



z 계산하기

- (1, m) 크기의 행 벡터

Z(z를 쌓은 행 벡터) = np.dot(w.T, X) + b # np.transpose(W)

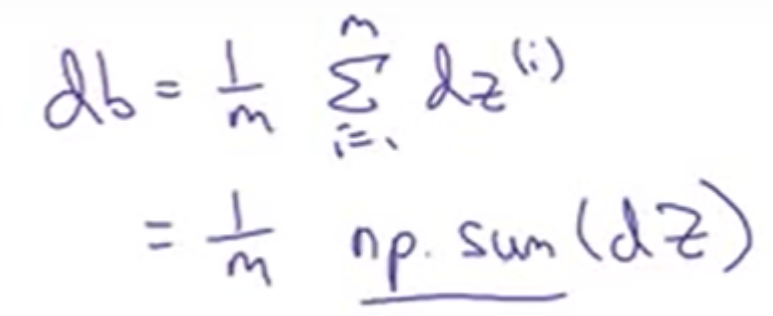
- 브로드캐스팅: b는 실수지만, 이 벡터와 더하면 자동으로 (1, m) 벡터로 변환

a 계산하기

- 시그모이드 함수 구현 필요
- Z 전체를 입력으로 받아 A 전체 출력

로지스틱 회귀의 경사 계산 벡터화

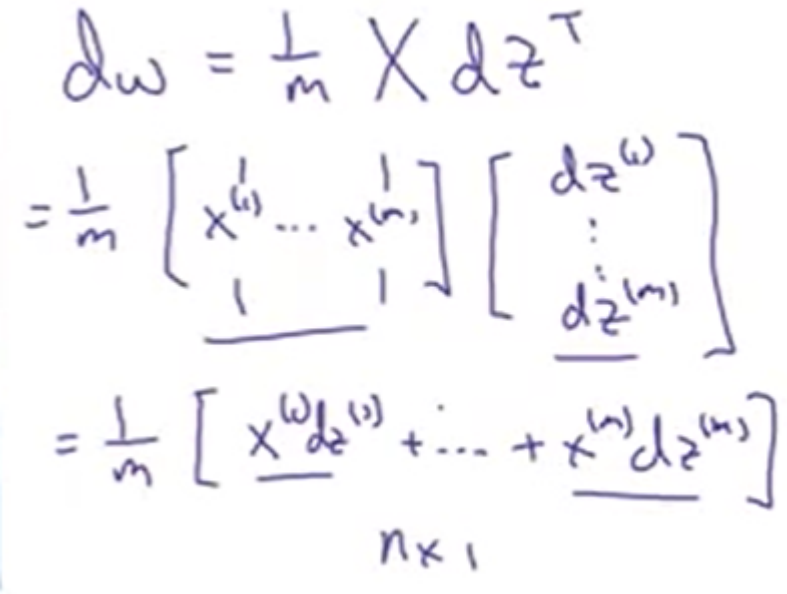
db 계산하기



최종 db = i가 1부터 m까지일 때, dz(i)의 합을 m으로 나눈 값

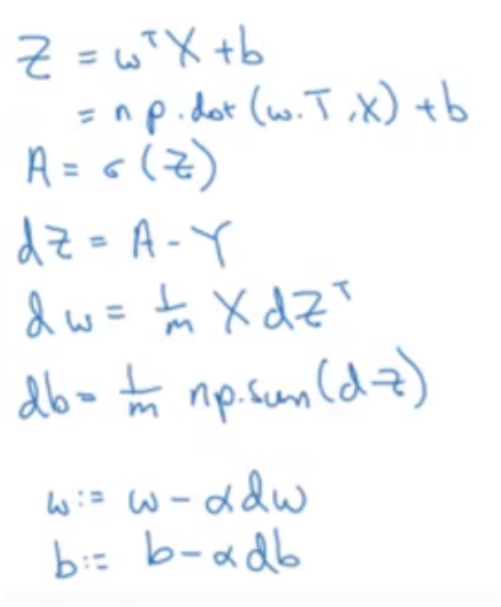
- db = 1/m * np.sum(dZ)

dw 계산하기



최종 dw = 1/m * X * dZ^T

최종 코드



1. Z, A를 벡터화해 한 번에 연산
2. dz = A - Y
3. db = 1/m * np.sum(dZ), dw = (1/m) X dZ^T

→ 경사하강법의 한 반복을 구현한 것, 경사하강법을 여러 번 적용하려면 for문 필요

브로드캐스팅(Broadcasting)

- 파이썬 코드 실행 시간을 줄일 수 있는 기법

예시 1: 사과, 고기, 계란, 감자의 탄수화물, 단백질, 지방 칼로리 함유량

Calories from Carbs, Proteins, Fats in 100g of different foods:

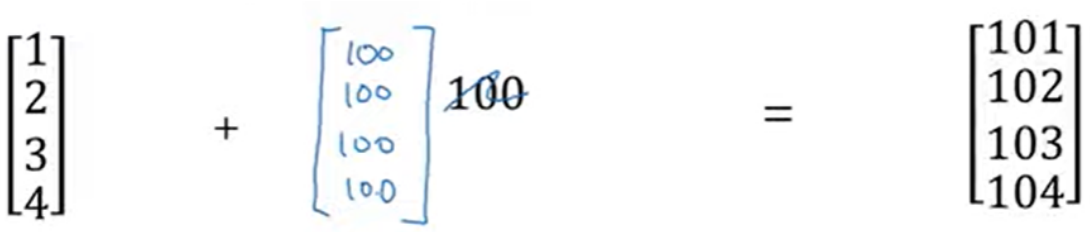
	Apples	Beef	Eggs	Potatoes
Carb	56.0	0.0	4.4	68.0
Protein	1.2	104.0	52.0	8.0
Fat	1.8	135.0	99.0	0.9

- 목표: 네 가지 음식의 탄수화물, 단백질, 지방이 주는 칼로리의 백분율을 구하는 것
  - for문 없이 수행할 수 있는가? → 가능!

```
cal = A.sum(axis=0) # 세로로 더함
percentage = 100 * A / cal.reshape(1, 4) # 브로드캐스팅
```

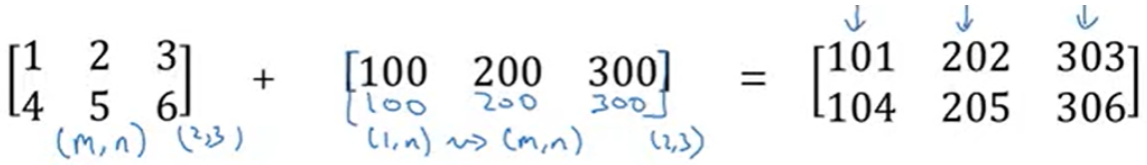
- A.sum(axis=0)
  - 세로로 더함(가로는 axis = 0)
- cal.reshape(1, 4)
  - (3, 4) 행렬인 A를 (1, 4) 행렬로 나눈다.
  - 행렬의 크기가 확실하지 않다면 reshape 함수로 확실히 해준다.
  - 상수 시간이 소요되어 호출이 가볍다.

예시 2: 상수 더하기



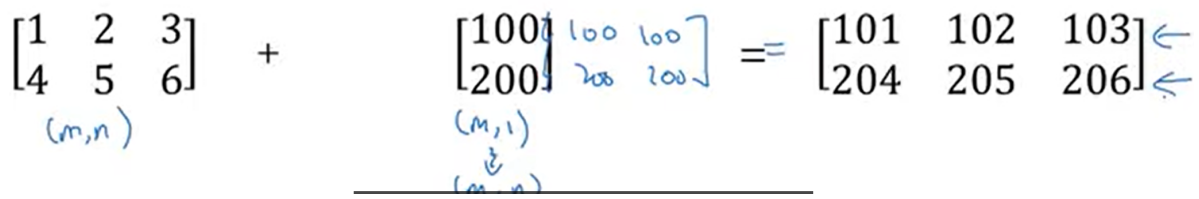
(4, 1) 벡터에 상수 100을 더한다면, 파이썬에서는 자동으로 100을 브로드캐스팅해 모든 요소가 100인 (4, 1) 벡터로 만들어준다.

예시 3: 열이 같은 행렬 연산



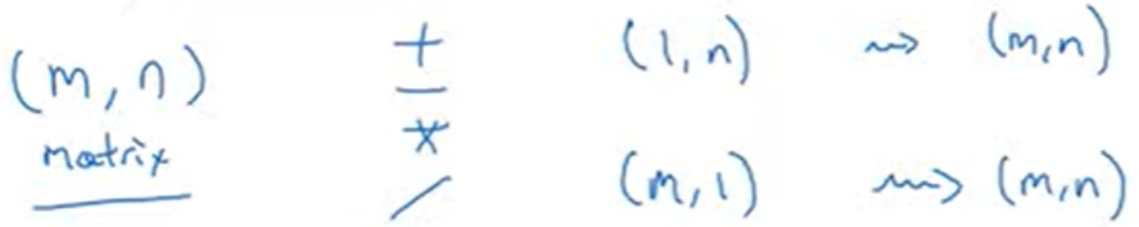
(m, n) 행렬에 (1, n) 행렬을 더한다면, 파이썬은 두 번째 행렬을 m번 세로로 복사해서 (m, n) 행렬로 만들어준다.

예시 4: 행이 같은 행렬 연산



(m, n) 행렬과 (m, 1) 행렬을 더한다면, 파이썬은 두 번째 행렬을 n번 가로로 복사해서 (m, n) 행렬로 만들어준다.

Generalization



(m, n) 행렬과 (1, n) 행렬/(m, 1) 행렬을 사칙연산하면 더 큰 행렬의 크기에 맞게 복사하여 요소별 연산

(m, 1) 행렬 혹은 열 벡터에 실수와 사칙연산을 하면 실수를 행렬에 크기에 맞게 복사하여 요소별 연산

파이썬과 넘파이 벡터

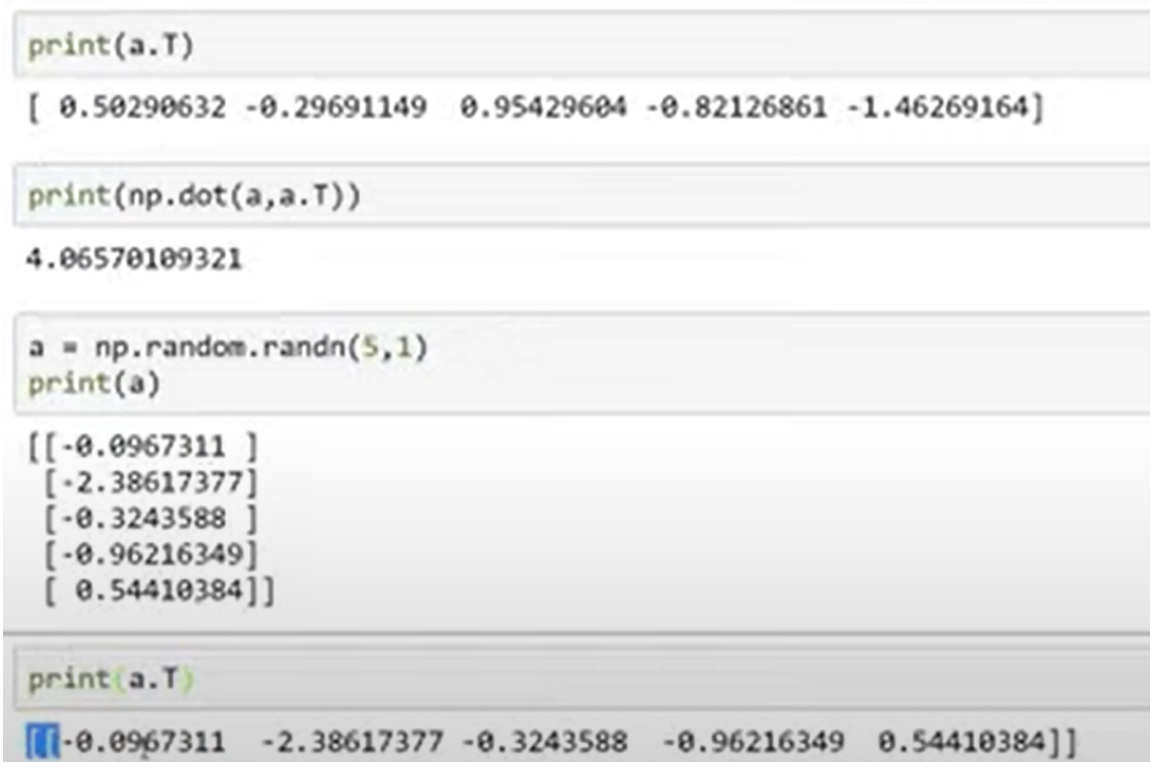
- 브로드캐스팅은 장점이자 약점
  - 장점: 넓은 표현성과 유연성
  - 약점: 내용 이해를 못 하면 찾기 어려운 오류가 생김 (ex) 행 벡터 + 열 벡터)

교수님의 코드 조언

🚨 **크기가 (n,) 또는 랭크가 1인 배열을 사용하지 않는다. 대신, 행 벡터나 열 벡터를 사용한다.**

```
a = np.random.randn(5)    # 크기: (5,)
a = np.random.randn(5, 1) # 크기: (5, 1)
```

- np.random.randn(5)
  - 가우시안 분포를 따르는 변숫값 5개를 배열 a에 저장
  - 전치, a와 a의 전치의 내적 연산이 정상적으로 실행되지 않음.
- np.random.randn(5, 1)
  - a는 5×1 열 벡터
  - 전치



  - a와 a의 전치의 내적: 벡터의 외적, 즉 행렬

랭크 1 벡터를 사용하지 않는 세 가지 방법

```
a = np.random.randn(5, 1)

assert(a.shape == (5, 1))

a = a.reshape((5, 1))
```

- assert(a.shape == (5, 1))
  - a가 (5, 1) 열 벡터라는 걸 확실히 하는 함수
  - 필요할 때 언제나 써도 좋음
- a = a.reshape((5, 1))
  - a가 랭크 1 벡터였어도 reshape 함수를 통해 행 또는 열 벡터로 바꿀 수 있음

Jupyter/iPython Notebooks 가이드

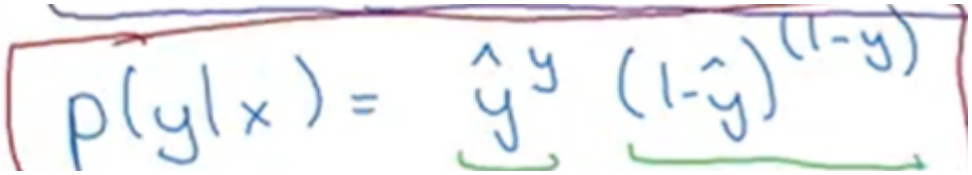
1. Start code here, End code here 사이에 코드 적기
2. 코드 실행: Shift + Enter
3. 커널이 작동하지 않는다는 오류: Kernel 탭 → Restart
4. 라이브러리 코드들도 실행
5. Submit assignment 눌러서 채점

로지스틱 회귀의 비용함수 설명

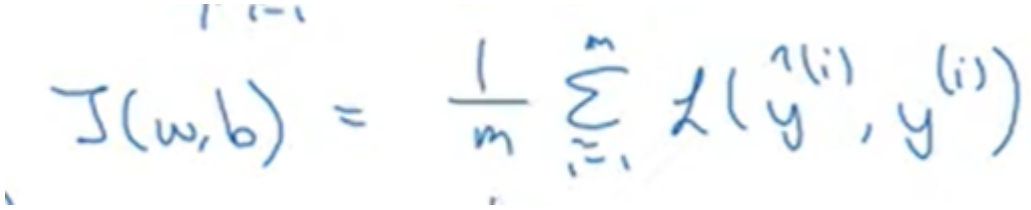
- 로지스틱 회귀에 왜 그 비용 함수를 쓰는지

$$\text{If} \quad y = 1: \qquad p(y|x) = \hat{y}$$
$$\text{If} \quad y = 0: \qquad p(y|x) = 1 - \hat{y}$$

- y는 1이거나 0임. 이 두 식을 합치면 아래의 식이 됨.



- 전체 훈련 세트에 대한 비용 함수는?



- 비용함수는 손실함수들의 평균을 최소화

퀴즈

10

```
import numpy as np

a = np.array([[3,0,1],[2,1,2]])
b = np.zeros([2,3])
c = np.ones([1,3])
d = a + b + c
print(d[1,1:3])
```

a = np.array([[3, 0, 1],[2,1,2]])

- 2x3 크기의 배열 a 선언
- a = [[3, 0, 1],[2,1,2]]

b = np.zeros([2,3])

- 2x3 크기의 배열 b 선언, 모든 원소 0으로 초기화
- b = [[0, 0, 0],[0, 0, 0]]

c = np.ones([1,3])

- 1x3 크기의 배열 c 선언, 모든 원소 1로 초기화

- c = [[1, 1, 1]]

d = a+b+c

- c를 브로드캐스팅하여 더하기
- d = [[4, 1, 2], [3, 2, 3]]

print(d[1,1:3])

- d의 1행, 1-2열 부분 배열 출력
- [2, 3]

3

1. np.random.randint(100)
    a. 0, 99 사이 랜덤 정수
2. np.random.rand(100)
    a. 0, 1 사이 랜덤 수 100개로 이루어진 배열 생성
3. np.random.randn(100)
    a. 평균 0, 표준편차 1인 정규분포에서 랜덤한 값 생성
4. np.random.rand()
    a. 0에서 1 사이 랜덤값 생성