우리가 사는 공간의 차원을 이야기할 때, 점은 0차원, 직선은 1차원, 면은 2차원, 입체는 3차원 등으로 묘사하게 된다.

우리 주변의 많은 데이터를 숫자로 나타내다 보면, 숫자들을 차원으로 표현해야 할 때가 있다. 고등학교 때 배운 벡터는 차원을 나타내는 숫자들의 나열을 의미한다. 여러 개의 벡터가 모여서 된 숫자 블록은 행렬(matrix)라고 부른다. 그럼, 행렬들이 모여 있는 것을 무엇이라고 해야 할까?

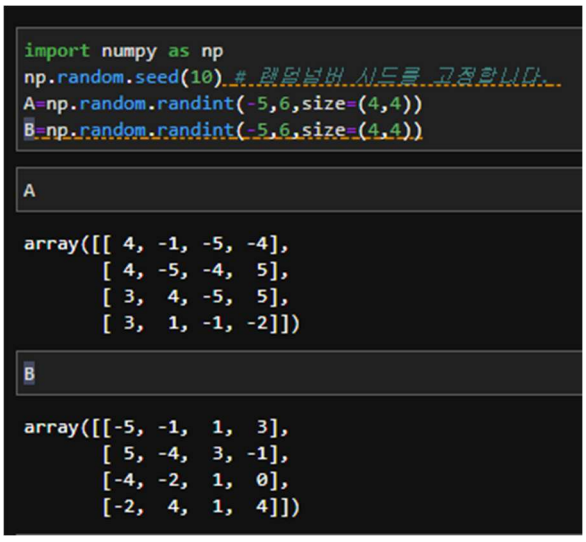
많은 숫자들의 모임을 다루기 위해 만들어진 개념이 바로 텐서(tensor)이다. 스칼라는 0차원 텐서, 벡터는 1차원 텐서, 행렬은 2차원 텐서로 정의하면, 행렬들의 모임은 3차원 텐서라고 부를 수 있다.

여기서는 파이썬을 사용해 큰 숫자들의 표현방법을 간단히 살펴보고 숫자들의 무리를 분석하기 위한 '거리'의 개념들과 데이터의 차원을 축소하는 방법론에 대해서 간략히 살펴보자.

행렬과 텐서

문제1. -5부터 5까지의 정수의 난수로 작성된 4 by 4 행렬 A,B를 작성하고 두 행렬의 곱 AB를 구하여라.

먼저 'numpy'를 이용해서 작성하도록 해 보자. 아래 식을 보면 'random.seed()'라는 메서드를 통해 난수 발생 시드값을 설정했다. 컴퓨터에서 발생하는 난수(random number)는 엄밀한 의미에서의 난수는 아니다. 특정한 발생장치에 의해 난수처럼 보이게 하는 장치라고 보면 되며, 이 때문에 특정한 난수초기값(seed)을 설정하면 수행할 때 마다 같은 난수로부터 시작하게 된다.



```
import numpy as np
np.random.seed(10) # 랜덤넘버 시드를 고정합니다.
A=np.random.randint(-5,6,size=(4,4))
B=np.random.randint(-5,6,size=(4,4))
```

```
A
```

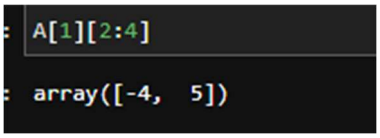
```
array([[ 4, -1, -5, -4],
       [ 4, -5, -4,  5],
       [ 3,  4, -5,  5],
       [ 3,  1, -1, -2]])
```

```
B
```

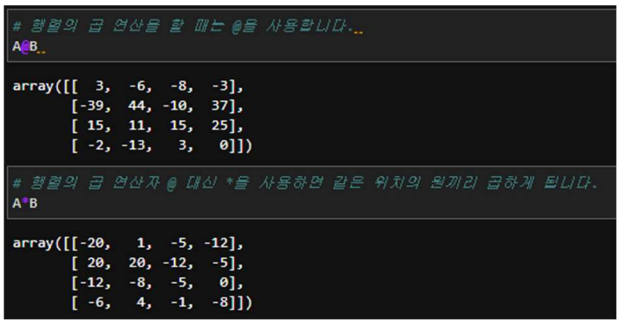
```
array([[-5, -1,  1,  3],
       [ 5, -4,  3, -1],
       [-4, -2,  1,  0],
       [-2,  4,  1,  4]])
```

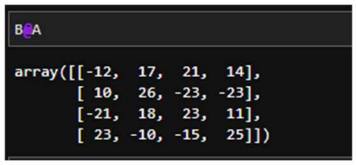
'numpy' 라이브러리는 벡터, 행렬, 텐서라는 용어를 사용하지 않고 모든 숫자들의 모임을 array()

라는 형식으로 모두 처리하고 있다. 넘파이 array 안에서 브레킷([])을 중첩하여 사용하면서 원하는 차원의 데이터를 자유롭게 표현할 수 있다. 참고로 A 행렬의 2행의 3,4열의 원소를 출력하려면 A[1][2:4]와 같이 슬라이싱 할 수 있다.

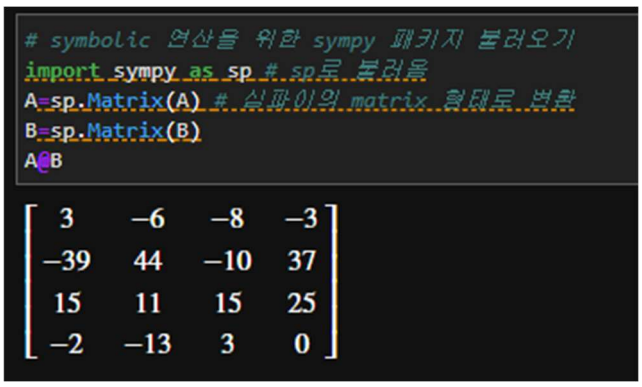


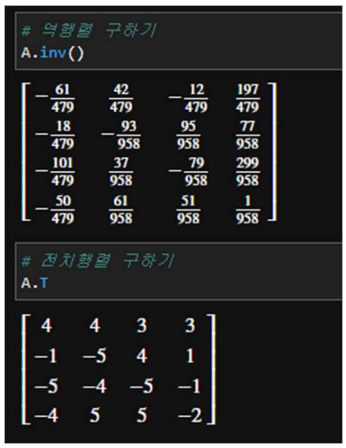
행렬에서 두 행렬 A,B의 곱셈 연산은 왼쪽 행렬(A)의 행과 오른쪽 행렬(B)의 열 벡터를 내적(inner product)한 값을 연속적으로 수행하는 것으로 정의된다. 따라서 AB와 BA의 결과는 다를 수 밖에 없다. 이런 연산을 위해 'numpy'에서는 '@'라는 기호를 사용하는 것을 알아 두자. '*'를 사용하여 곱을 하면 아래와 같이 같은 위치의 원끼리 곱한 결과만을 보여준다.





심볼릭 연산(symbolic calculations)을 하는 'sympy' 라이브러리를 사용하면 벡터, 행렬 연산시 좀 더 눈으로 보기에 명확하게 볼 수 있다는 장점도 있으니 참고하자.





딥러닝 분석에서 많이 사용되는 'tensorflow'라는 라이브러리[1]를 사용할 수도 있다. 아래 예에서 볼 수 있는 것처럼 'numpy'에서 만들어지는 array는 'tensorflow'에서도 쉽게 호환되기 때문에 파

[1] 자신의 PC가 NVIDIA GPU 사용이 가능하다면 conda install -c conda-forge tensorflow-gpu로 설치하면 되고, 없다면 conda install -c conda-forge tensorflow로 설치하면 된다.

이쎤에서 다차원의 데이터를 다룰 때는 기본적으로 넘파이를 가장 많이 활용한다고 알아두고 넘어가자.

```
# 텐서플로루 라이브러리 사용
# 텐서플로우 내부에도 난수 발생기가 있으나 numpy를 활용함
import tensorflow as tf
import numpy as np
np.random.seed(10) # 랜덤넘버 시드를 고정합니다.
A=np.random.randint(-5,6,size=(4,4))
B=np.random.randint(-5,6,size=(4,4))
A=tf.constant(A) # 텐서플로우의 숫자배열 형태로 정의
B=tf.constant(B)
A@B
```

```
<tf.Tensor: shape=(4, 4), dtype=int32, numpy=
array([[  3,  -6,  -8,  -3],
       [-39,  44, -10,  37],
       [ 15,  11,  15,  25],
       [ -2, -13,   3,   0]])>
```

문제2. 0부터 9까지의 정수 난수의 (3,4,5) shape의 tensor를 만들고 모든 원의 합을 구하여 보아라.

'numpy.randint'를 이용하면 아래와 같이 쉽게 그 모양을 확인할 수 있다.

```
import numpy as np
D=np.random.randint(10,size=(3,4,5))
D
```

```
array([[[1, 3, 7, 1, 6],
        [4, 0, 1, 8, 7],
        [9, 5, 9, 7, 5],
        [0, 5, 9, 2, 5]],

       [[6, 9, 0, 1, 5],
        [9, 1, 1, 0, 2],
        [8, 4, 2, 1, 1],
        [7, 1, 5, 9, 9]],

       [[0, 2, 9, 4, 5],
        [6, 0, 1, 6, 3],
        [6, 9, 7, 2, 4],
        [1, 2, 0, 5, 8]]])
```

결과를 보면 데이터 '[]'의 3중으로 결합되어 있음을 알 수 있다. (4,5) 행렬 3개가 중첩되어 있는 것이다. 모든 원의 합은 sum()을 이용해 구하면 된다.

```
D.sum()
```

```
255
```

그렇다면, 첫번째 축을 중심으로 그 합을 구하는 어떻게 할까? 즉, (4,5) 행렬 3개를 같은 모양의 원별로 합을 구하라는 의미이다. 이것은 'axis'라는 매개변수를 통해 할 수 있다. axis=0으로 하면 첫번째 축을 중심으로 (4,5)의 결과값을 출력해주고, axis=2로 하면 (3,4)의 결과를 보여주게 된다.

```
D.sum(axis=0)

array([[ 7, 14, 16,  6, 16],
       [19,  1,  3, 14, 12],
       [23, 18, 18, 10, 10],
       [ 8,  8, 14, 16, 22]])
```

```
D.sum(axis=2)

array([[18, 20, 35, 21],
       [21, 13, 16, 31],
       [20, 16, 28, 16]])
```

2차원 텐서인 (m,n) shape 행렬의 가로방향 합, 세로방향 합은 쉽게 그려지는데, 3차원 부터는 어느 방향 합인지 쉽게 그려지지 않는다. (m,n) shape 행렬의 가로방향의 합을 하게 되면 (m,) shape의 결과가 나오고, 세로방향 합을 하게 되면 (n,) shape의 결과가 나오게 됨을 알 수 있는데, 3차원 이상에서는 우리가 원하는 차원의 결과를 얻기 위해 어떤 축을 제거하는지에 착안하면 조금 쉽게 알 수 있을 듯 하다. 즉, (3,4,5) 텐서에서 (3,5) shape의 결과를 얻기 위해서는 axis=1을 제거하면 되기 때문에 D.sum(axis=1)과 같이 하는 것이다.

```
D.sum(axis=1)

array([[14, 13, 26, 18, 23],
       [30, 15,  8, 11, 17],
       [13, 13, 17, 17, 20]])
```

그렇다면, (5,)의 결과를 얻기 위해서는 어떻게 하면 좋을까? 위와 같은 논리에 의해 axis=0,1인 경우를 모두 선택하면 된다. 이것은 axis=0을 먼저 하면 결과가 (4,5)가 되기 때문에, 이를 다시 axis=0으로 해서 (5,) 모양만 남기는 경우와 같을 것이다.

```
D.sum(axis=(0,1))

array([57, 41, 51, 46, 60])
```

```
D.sum(axis=0).sum(axis=0)

array([57, 41, 51, 46, 60])
```

문제3. 앞 문제2에서 만들어진 D 텐서에는 (4,5) 행렬 3개가 차례로 들어 있다. 편의상 이 행렬들을 M0, M1, M2라고 하고, M0, M0+M1, M0+M1+M2 로 만들어지는 텐서로 수정해 보아라.

이 문제는 슬라이싱을 이용해서 아래와 같이 하면 된다.

```
M0,M1,M2=D[0],D[1],D[2]
[M0,M0+M1,M0+M1+M2]
```

```
[array([[1, 3, 7, 1, 6],
        [4, 0, 1, 8, 7],
        [9, 5, 9, 7, 5],
        [0, 5, 9, 2, 5]]),
 array([[ 7, 12,  7,  2, 11],
        [13,  1,  2,  8,  9],
        [17,  9, 11,  8,  6],
        [ 7,  6, 14, 11, 14]]),
 array([[ 7, 14, 16,  6, 16],
        [19,  1,  3, 14, 12],
        [23, 18, 18, 10, 10],
        [ 8,  8, 14, 16, 22]])]
```

다른 방법으로는 'numpy'의 'cumsum()'을 이용해도 된다. 누적 합을 합해 가는 함수인데 예를 들면 1부터 10까지의 자연수를 생성하고 이에 대한 array의 cumsum()은 아래와 같은 연산을 도와

준다.

```
E0=np.array([i for i in range(1,11)])
E0.cumsum()
```

```
array([ 1,  3,  6, 10, 15, 21, 28, 36, 45, 55])
```

이를 이용해서 원래 문제에 적용해 보면 (4,5)의 행렬을 더해가는 문제이기 때문에 axis=0 방향으로 더하면 될 것이다.

```
D.cumsum(axis=0)
```

```
array([[[ 1,  3,  7,  1,  6],
        [ 4,  0,  1,  8,  7],
        [ 9,  5,  9,  7,  5],
        [ 0,  5,  9,  2,  5]],

       [[ 7, 12,  7,  2, 11],
        [13,  1,  2,  8,  9],
        [17,  9, 11,  8,  6],
        [ 7,  6, 14, 11, 14]],

       [[ 7, 14, 16,  6, 16],
        [19,  1,  3, 14, 12],
        [23, 18, 18, 10, 10],
        [ 8,  8, 14, 16, 22]]])
```

문제4. (m,n) 행렬의 전치행렬은 (n,m)이 된다. 문제의 (3,4,5) 텐서를 (3,5,4) 텐서로 바꾸어 보아라.

먼저 (3,4) 행렬을 만들고 이를 바탕으로 먼저 전치행렬 연산자를 이해해 보자.

```
import numpy as np
np.random.seed(0) # 시드값 고정
D=np.random.randint(10,size=(3,4))
D
```

```
array([[5, 0, 3, 3],
       [7, 9, 3, 5],
       [2, 4, 7, 6]])
```

```
D.transpose()
```

```
array([[5, 7, 2],
       [0, 9, 4],
       [3, 3, 7],
       [3, 5, 6]])
```

```
np.transpose(D,(1,0))
```

```
array([[5, 7, 2],
       [0, 9, 4],
       [3, 3, 7],
       [3, 5, 6]])
```

위에서 알 수 있는 것 처럼, 'np.transpose(D,(1,0))'와 같이 바꾸고자 하는 axis를 표현하면 명확하다. 3차원 이상의 텐서에서 특정배열을 바꾸려면 아래와 같이 하면 된다.

```
D=np.random.randint(10,size=(3,4,5))
D

array([[[8, 8, 1, 6, 7],
        [7, 8, 1, 5, 9],
        [8, 9, 4, 3, 0],
        [3, 5, 0, 2, 3]],

       [[8, 1, 3, 3, 3],
        [7, 0, 1, 9, 9],
        [0, 4, 7, 3, 2],
        [7, 2, 0, 0, 4]],

       [[5, 5, 6, 8, 4],
        [1, 4, 9, 8, 1],
        [1, 7, 9, 9, 3],
        [6, 7, 2, 0, 3]]])
```

```
np.transpose(D,(0,2,1))

array([[[8, 7, 8, 3],
        [8, 8, 9, 5],
        [1, 1, 4, 0],
        [6, 5, 3, 2],
        [7, 9, 0, 3]],

       [[8, 7, 0, 7],
        [1, 0, 4, 2],
        [3, 1, 7, 0],
        [3, 9, 3, 0],
        [3, 9, 2, 4]],

       [[5, 1, 1, 6],
        [5, 4, 7, 7],
        [6, 9, 9, 2],
        [8, 8, 9, 0],
        [4, 1, 3, 3]]])
```

문제5. 딥러닝 학습시에는 데이터의 차원을 새로 만들거나 뭉치는 등의 작업이 필요할 때가 많다. (3,4,5) 텐서를 (3,1,4,5) 텐서로 만들어 보아라.

'numpy'에는 'expand_dims()'라는 함수가 있는데 이는 아래와 같이 작동한다.

```
D=np.random.randint(10,size=(3,4))
D

array([[5, 9, 4, 4],
       [6, 4, 4, 3],
       [4, 4, 8, 4]])
```

```
np.expand_dims(D,axis=0)

array([[[5, 9, 4, 4],
        [6, 4, 4, 3],
        [4, 4, 8, 4]]])
```

위 결과를 보면 동일한 것 같아 보이지만 오른쪽에 [] 하나가 더 겹쳐져 있다. 사실 (3,4) 텐서가 'expand_dims()'에 의해 (1,3,4)로 바뀐 것이다. 아래에서는 axis=1에 차원을 하나 더 추가한 것인데, (1,4) 벡터 3개의 연결로 변환된 것을 알 수 있다. 이와 같은 원리에 의해 D1이 (3,4,5) 텐서라고 하면 np.expand_dims(D1, axis=1)로 변환시키면 된다.

```
np.expand_dims(D,axis=1)

array([[[5, 9, 4, 4]],

       [[6, 4, 4, 3]],

       [[4, 4, 8, 4]]])
```

```
# 앞 셀 결과의 shape 보기
_.shape

(3, 1, 4)
```

```
D1=np.random.randint(10,size=(3,4,5))
D2=np.expand_dims(D1,axis=1)
D2.shape

(3, 1, 4, 5)
```

반대로 (3,1,4,5)의 텐서를 (3,4,5) 텐서로 변환하는 것은 'squeeze()'를 이용하면 된다. 'squeeze()'는 차원 중 1인 값만 제거할 수 있으니 참고하기 바란다.

```
D3=np.squeeze(D2)
D3.shape

(3, 4, 5)
```

참고로 'reshape'을 이용하면 다양한 차원으로 변형할 수 있다. 1차원 텐서인 벡터로 만드는 방법으로는 'flatten'을 사용해도 좋다.

```
D

array([[5, 9, 4, 4],
       [6, 4, 4, 3],
       [4, 4, 8, 4]])

D.reshape(12)

array([5, 9, 4, 4, 6, 4, 4, 3, 4, 4, 8, 4])

D.flatten()

array([5, 9, 4, 4, 6, 4, 4, 3, 4, 4, 8, 4])
```

```
D.reshape(2,6)

array([[5, 9, 4, 4, 6, 4],
       [4, 3, 4, 4, 8, 4]])

D.reshape(2,2,3)

array([[[5, 9, 4],
        [4, 6, 4]],

       [[4, 3, 4],
        [4, 8, 4]]])
```

여기서 하나 꼭 주의할 점이 하나 있다. 아래의 E0,E1,E2를 보면 모두 (1,2,3)이란 동일한 데이터가 들어가 있는데, 모두 배열의 모양이 다르다는 것이다.

```
E0=np.array([1,2,3])
E0

array([1, 2, 3])
```

```
E1=E0.reshape(-1,1)
E1

array([[1],
       [2],
       [3]])
```

```
E2=E0.reshape(1,-1)
E2

array([[1, 2, 3]])
```

위 데이터의 모양을 출력해 보면 더욱 확연하게 드러난다. 즉, 아래에서 보는 것과 같이 E0는 1차원 텐서이고, E1, E2는 2차원 텐서이다. 나중에 기계학습편에서 직면하게 되겠지만, 1차원 텐서를 사용하면 에러가 발생하는 클래스들이 많기 때문에 reshape(1,-1) 혹은 reshape(-1,1)을 이용해서 2차원 텐서로 만들 수 있다는 것을 기억해 두길 바란다.

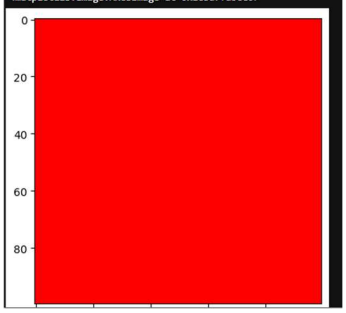
```
E0.shape, E1.shape, E2.shape

((3,), (3, 1), (1, 3))
```

문제6. 빛의 3요소는 빨강(R),초록(G),파랑(B)으로 알려져 있으며, 컴퓨터는 이미지를 표현할 때, 각 색을 0부터 255까지의 정수로 그 강함을 표시한다고 한다. 따라서 한 픽셀에 하나의 색을 표현할 때는 (R,G,B)의 3차원 데이터가 필요하다고 볼 수 있다. 한 평면에 만일 가로 2개, 세로 3개의 총 6개 픽셀이 있다고 하면 각 픽셀에 3차원 데이터를 넣어 주어야 하는 것이다. 사각형 6개를 만들고 첫 줄에는 빨강, 녹색, 파랑을, 두번째 줄에는 노랑색, 핑크색, 흰색을 표현해 보아라.

숫자를 이미지로 표현하기 위해서는 먼저 'matplotlib.pyplot' 을 주로 사용한다[2]. 먼저 빨강색을 이미지로 표현하는 'imshow()' 메서드를 사용해 보자.

```
import matplotlib.pyplot as plt # plt 별명으로 불러옴
import numpy as np

C=np.array([255,0,0]) # 빨강색의 값 지정

# np.uint8: 0부터 255까지의 정수값 표현지정
# np.zeros()를 사용하여 모든 원소가 0인 array 작성
image = np.zeros([100, 100, 3],dtype=np.uint8)
image[...] = C # image[...]=image[:,:,...] 와 같음
plt.imshow(image) # 숫자를 이미지로 표현
```

```
<matplotlib.image.AxesImage at 0x1c3d7fd5cc0>
```




```
image
```

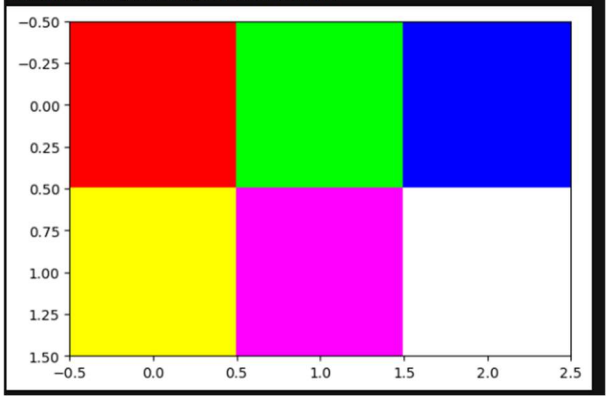
```
array([[[255,   0,   0],
        [255,   0,   0],
        [255,   0,   0],
        ...,
        [255,   0,   0],
        [255,   0,   0],
        [255,   0,   0]],

       [[255,   0,   0],
        [255,   0,   0],
        [255,   0,   0],
        ...,
        [255,   0,   0],
        [255,   0,   0],
        [255,   0,   0]],

       [[255,   0,   0],
        [255,   0,   0],
        [255,   0,   0],
        ...,
        [255,   0,   0],
        [255,   0,   0],
        [255,   0,   0]],

       ...,
```

위 코드를 보면 결국 [100,100]이라는 행렬에서 각 원소에 3차원 데이터를 지정하는 하나의 큰 3차원 tensor의 형태가 바로 np.zeros([100,100,3])이라고 볼 수 있다.

노랑은 '빨강+초록', 핑크는 '빨강+파랑', 흰색은 '빨강+초록+파랑'임을 알고, 위 예제를 이해하면 이 문제의 답은 아래와 같이 쉽게 할 수 있다.

```
import matplotlib.pyplot as plt # plt 별명으로 불러옴
import numpy as np

CM=np.zeros((2,3,3),dtype=np.uint8)
CM[0,0,:]=(255,0,0) # Red
CM[0,1,:]=(0,255,0) # Green
CM[0,2,:]=(0,0,255) # Blue
CM[1,0,:]=(255,255,0) # Yellow
CM[1,1,:]=(255,0,255)  # pink
CM[1,2,:]=(255,255,255) #white
plt.imshow(CM)
```

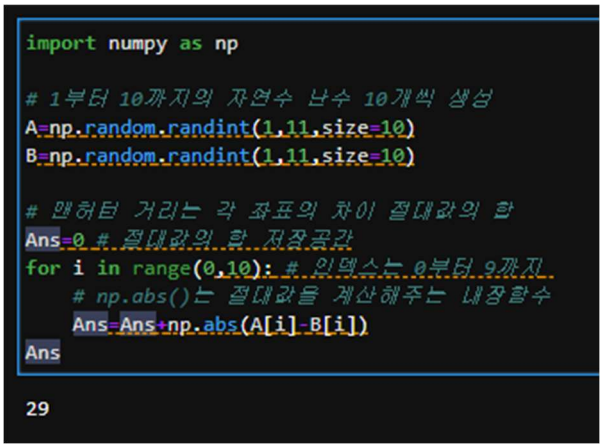
```
<matplotlib.image.AxesImage at 0x1c3d95c95d0>
```



[2] 다양한 그래프와 차트를 그리는데 사용되는 matplotlib 라이브러리는 anaconda를 설치할 때 이미 설치되어 있는 경우가 많다.

수학적 거리

문제7. 1부터 10까지의 자연수를 원으로 하는 10차원의 랜덤 벡터 2개를 작성하여 두 벡터의 맨하턴 거리(Manhattan Distance)를 구하여라.

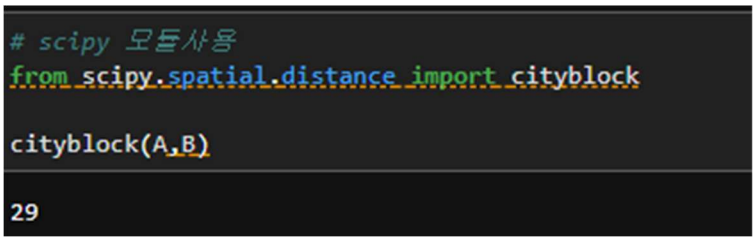


```
import numpy as np

# 1부터 10까지의 자연수 난수 10개씩 생성
A=np.random.randint(1,11,size=10)
B=np.random.randint(1,11,size=10)

# 맨허턴 거리는 각 좌표의 차이 절대값의 합
Ans=0 # 절대값의 합 저장공간
for i in range(0,10): # 인덱스는 0부터 9까지
    # np.abs()는 절대값을 계산해주는 내장함수
    Ans=Ans+np.abs(A[i]-B[i])
Ans
```

```
29
```

우리는 두 벡터의 거리를 구할 때, 일반적으로 유클리디안 거리(Euclidean distance)라고 불리는 L2 거리를 주로 사용한다. 하지만, 도시의 건물에서 사용되는 거리는 건물을 통과해서 가는 최단거리보다는 건물을 돌아서 가는 최단거리의 개념이 필요하며 이 거리의 개념이 바로 맨하탄 거리 혹은 도시블록거리(City Block Distance), L1 거리라고 한다.

맨하턴 거리는 'scipy' 패키지[3] 아래에 있는 'cityblock' 메서드를 사용하면 좀더 쉽게 답을 구할 수 있다.

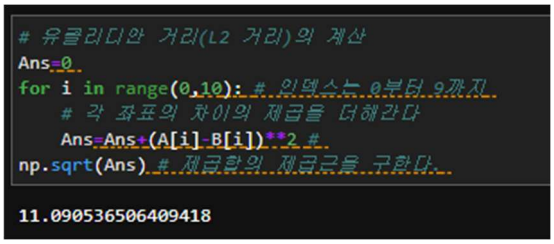


```
# scipy 모듈사용
from scipy.spatial.distance import cityblock

cityblock(A,B)
```

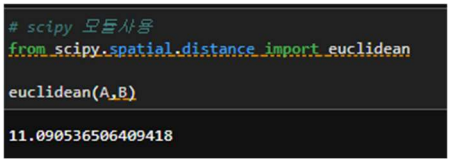
```
29
```

L2거리도 구해보면 아래와 같다.

[33] conda install -c conda-forge scipy

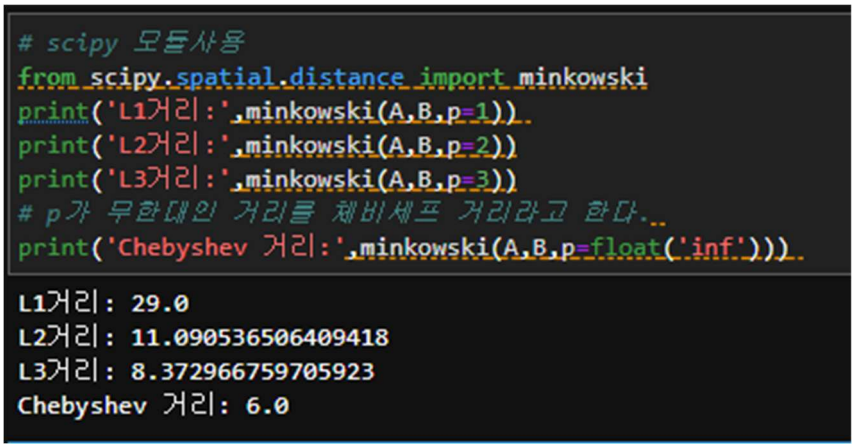


```
# 유클리디안 거리(L2 거리)의 계산
Ans=0
for i in range(0,10): # 인덱스는 0부터 9까지
    # 각 좌표의 차이의 제곱을 더해간다
    Ans=Ans+(A[i]-B[i])**2 #
np.sqrt(Ans) # 제곱합의 제곱근을 구한다.

11.090536506409418
```



```
# scipy 모듈사용
from scipy.spatial.distance import euclidean

euclidean(A,B)

11.090536506409418
```

L1거리, L2거리 등을 일반화하여 만든 민코우스키거리(Minkowski Distance)는 아래와 같이 사용할 수 있다.



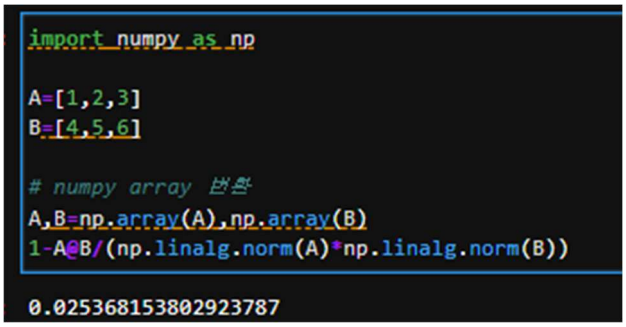
```
# scipy 모듈사용
from scipy.spatial.distance import minkowski
print('L1거리:',minkowski(A,B,p=1))
print('L2거리:',minkowski(A,B,p=2))
print('L3거리:',minkowski(A,B,p=3))
# p가 무한대인 거리를 체비셰프 거리라고 한다.
print('Chebyshev 거리:',minkowski(A,B,p=float('inf')))

L1거리: 29.0
L2거리: 11.090536506409418
L3거리: 8.372966759705923
Chebyshev 거리: 6.0
```

문제8. A=[1,2,3],B=[4,5,6]이라고 표현되는 두 벡터의 방향적 유사성을 나타내는 거리방법도 있는데, 대표적으로 코사인 거리(cosine distance)라는 것이 있다. 0에 가까울수록 두 벡터가 비슷하다고 정의한다. 코사인 거리를 구하여 보아라.

코사인 거리는 수학적으로 아래와 같이 정의된다.

$$d = 1 - \frac{A \cdot B}{\|A\|\|B\|}$$

위 정의에 의해 거리를 구하면 아래와 같다.



```
import numpy as np

A=[1,2,3]
B=[4,5,6]

# numpy array 변환
A,B=np.array(A),np.array(B)
1-A@B/(np.linalg.norm(A)*np.linalg.norm(B))

0.025368153802923787
```

'scipy'내 cosine 함수를 이용하면 리스트 형태의 자료에 대해 바로 거리를 구할 수 있다.

```
from scipy.spatial.distance import cosine

A=[1,2,3]
B=[4,5,6]

# 코사인 거리 계산
cosine(A,B)
```

```
0.0253681538029239
```

벡터 방향 유사성을 계산하는 방법과 유사하게 집합 간의 유사성을 측정하는 지표도 있는데, 대표적인 것이 바로 자카드(Jaccard) 거리이다. 자카드 거리는 수학적으로 아래와 같이 정의된다.

$$d = 1 - \frac{|A \cap B|}{|A \cup B|}$$

위 기호에서 $|A|$는 집합의 크기를 나타내고 있는데, 0에 가까울수록 두 집합이 유사하다는 것을 나타낸다.

```
A={'사과','배','감','멜론','수박'}
B={'참외','사과','바나나','감','오렌지','귤','망고'}

# Jaccard 거리 계산
1-len(A.intersection(B))/len(A.union(B))
```

```
0.8
```

참고로 'scipy'의 자카드 거리 함수 'jaccard'는 위와 같은 문제에서는 적용할 수가 없다. 아래 예제를 통해 차이점을 살펴보자.

```
# scipy 아래 있는 jaccard 함수는 같은 크기의 bool 형 자료형일 때만 작동한다.
# 또한 A,B의 크기가  동일해야 한다.
from scipy.spatial.distance import jaccard

A=[1,0,0]
B=[1,1,0]

jaccard(A,B)
```

```
0.5
```

위 예제를 'numpy'로 따로 구현해 보면 이해가 쉽다.

```python
def jaccard_distance(arr1, arr2):
    intersection = np.sum(np.logical_and(arr1, arr2))
    union = np.sum(np.logical_or(arr1, arr2))
    jaccard_similarity = intersection / union
    jaccard_distance = 1 - jaccard_similarity
    return jaccard_distance

A=[1,0,0]
B=[1,1,0]

A,B=np.array(A),np.array(B)
# Jaccard 거리 계산
distance = jaccard_distance(A, B)

print(f"Jaccard 거리: {distance}")
```

```
Jaccard 거리: 0.5
```

통계적 거리

평균이 5이고 표준편차가 1인 자료에서 나온 10이라는 점 A와 평균이 5이고 표준편차가 10인 자료에서 나온 12라는 점 B를 통계적 관점에서 바라보자.

앞에서 살펴본 수학적 거리에 의하면 5라는 점에서 더 가까운 점은 A라고 할 수 있다. 하지만, 점 A는 평균이 5이고 분산이 1인 자료에서는 거의 나오지 않는 특이점(outlier)에 가까운 점이고 점 B는 매우 나올 가능성이 높은 자료라는 점에서 바라보면 5에서 더 가까운 점은 B라고 이야기 할 수도 있다.

위에서 말한 것을 통계적 관점으로 계량적으로 계산해 보자면 표준정규분포로 표준화하여 볼 수 있다는 것이고, A라는 점은 (10-5)/1=5, B라는 점은 (12-5)/10=0.7로서 5에서 더 가까운 거리에 있는 점은 B가 된다.

문제9. 점 A (3,4,2), 점 B(1,2,5)는 x, y, z의 3차원 변수가 영향을 주고 있다. 각 변수 x,y,z는 서로 통계적으로 연관되어 있으며, 각 변수는 평균과 표준편차가 서로 다른 상황이라고 가정하자. 분산과 상관계수를 감안한 공분산행렬은 아래와 같다고 할 때, A,B점 간의 거리를 계량화하여 보아라.

$$\begin{bmatrix} 10 & 2 & 4 \\ 2 & 8 & 1 \\ 4 & 1 & 6 \end{bmatrix}$$

여러 변수들이 서로 얽혀 있는 상황을 감안하여 두 벡터의 차이를 정규화 하려면 조금 복잡한 과정을 거쳐야 한다. 이와 관련된 방법 중 하나가 바로 마할라노비스(Mahalanobis Distance)이다.

단일 변수 일 때의 정규화 과정을 살펴보면 아래와 같이 두 점의 거리를 표준편차로 나누는 과정으로 이해할 수 있다.

$$z = \frac{x - \mu}{\sigma}$$

앞선 단원에서 본 공분산 행렬의 정의 과정을 다시한번 살펴보면 공분산 행렬은 각 변수들의 관련도를 감안하여 분산과 공분산을 한꺼번에 계산한 것이다.

$(X - \mu)$: 각 칼럼에서 칼럼 평균값을 빼서 만든 행렬

$(X - \mu)^T$: $(X - \mu)$의 전치행렬

$$\text{공분산행렬(S)} = [(X - \mu)(X - \mu)^T] = E\left[\begin{pmatrix} X_1 - \mu_1 \\ \vdots \\ X_n - \mu_n \end{pmatrix}(X_1 - \mu_1 \quad \cdots \quad X_n - \mu_n)\right]$$

따라서, 다변량 두 벡터의 차이 d=X1-X2로 정의한다면 마할라노비스 거리(D)는 아래와 같이 계산된다.

$$D^2 = dS^{-1}d^T$$

이를 바탕으로 sympy를 사용해서 먼저 계산해 보면 아래와 같다.

```
import sympy as sp
S=sp.Matrix(covariance_matrix)

# 데이터 포인트
A = sp.Matrix(1,3,[3, 4, 2])
B = sp.Matrix(1,3,[1, 2, 5])

# 마할라노비스 거리계산
d=A-B # 두 벡터의 차이
(d@S.inv()@d.T)**0.5
```

```
[2.02381036246817]
```

여러가지 거리 개념이 들어 있는 scipy.spatial.distance를 사용하여 계산할 수도 있으니 참고하자. 아래 코드를 살펴보면 numpy를 이용해 데이터들을 재정의 했는데, sympy용 데이터는 심볼릭 연산 중심이다 보니 호환성이 떨어지는 단점이 있다.

```python
from scipy.spatial.distance import mahalanobis
import numpy as np

# 데이터 포인트 (예: 다변량 데이터)
A = np.array([3, 4, 2])
B = np.array([1, 2, 5])

# 공분산 행렬
S = np.array([[10, 2, 4],[2, 8, 1],[4, 1, 6]])

# numpy를 이용한 역행렬 계산
S_inv = np.linalg.inv(S)

# Mahalanobis 거리 계산
distance = mahalanobis(A, B, S_inv)

print("두 데이터 포인트 간의 Mahalanobis 거리:", distance)
```

```
두 데이터 포인트 간의 Mahalanobis 거리: 2.0238103624681667
```

수학적, 통계적 거리 이외에도 데이터의 특성에 따라 데이터의 유사도 측정하는 다양한 방법이 존재하고 있다[4]. 이 모든 것을 아는 것이 이 책의 목적은 아니니, 데이터의 거리를 측정하는 방법의 다양성에 대해서만 인지하고 있자.

데이터 차원의 축소(심화)

여기서는 2차원 텐서인 행렬을 중심으로 데이터의 차원을 축소하여 많은 데이터를 빠르게 분석하고 전송하는 방법론의 수학적인 배경을 간단히 소개하고자 한다. 사실, 이 내용을 몰라도 기계학습을 수행하는 데에는 아무런 지장이 없으니, 이해가 가지 않는다고 좌절하지 않아도 된다.

문제10. 아래 행렬을 A라고 할 때, Ax=ex를 만족하는 0벡터가 아닌 열벡터 x(3 by 1)와 스칼라 e를 구하여 보아라.

$$\begin{bmatrix} 2 & 0 & -1 \\ 0 & 4 & 0 \\ -1 & 0 & 1 \end{bmatrix}$$

---

[4] 방향성을 중시하는 cosine거리, 합집합과 교집합을 중심으로 계산하는 Jaccard 거리, 두 문장간의 편집 거리를 측정하는 Hamming 거리 등이 있다.

열벡터의 x의 원소를 x1,x2,x3로 하고, e를 이용하여 위 식을 만족하는 식을 쓰면 미지수 4가지에 방정식이 3개인 부정방정식[5]이 만들어진다. 'sympy'의 'solve()'를 이용해서 해를 구하면 e는 3가지가 나옴을 확인할 수 있다. Ax=ex를 만족하는 e의 값을 고유값(eigenvalue)라고 부른다.

```
x1,x2,x3=smp.symbols('x1 x2 x3') # 기호 정의
e=smp.symbols('e',nonzero=True) # 기호 정의
x=smp.Matrix(3,1,[x1,x2,x3])
# 행렬에서의 곱은 '@'를 쓰는게 좋다.
smp.solve(A@x-e*x,[x1,x2,x3,e])
```

```
[(0, x2, 0, 4),
 (x3*(-1/2 + sqrt(5)/2), 0, x3, 3/2 - sqrt(5)/2),
 (x3*(-sqrt(5)/2 - 1/2), 0, x3, sqrt(5)/2 + 3/2),
 (0, 0, 0, e)]
```

위에서 e값은 4, 3/2-sqrt(5)/2, sqrt(5)/2+3/2 와 같은 3가지가 결정되었는데, e=4라고 했을 때 x=[0,x2,0] 이므로 가장 간단한 벡터로 x=[0,1,0]이라고 해보자[6]. 아래와 같이 Ax=4x가 됨을 확인할 수 있다.

```
x=smp.Matrix(3,1,[0,1,0])
A@x-4*x
```

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

고유벡터들을 열벡터로 하나씩 모은 행렬을 P라고 하고, 고유값들을 대각원소에 넣은 벡터를 D라고 하면 AP=PD라는 관계식이 성립하며, $A=PDP^{-1}$인 관계가 성립함을 알 수 있다.

$$\begin{matrix} A\left(x_1\right)=e_1\left(x_1\right) \\ \vdots \\ A\left(x_n\right)=e_n\left(x_n\right) \end{matrix} \rightarrow \left[A\left(x_1\right) \quad A\left(x_2\right) \quad \cdots \quad A\left(x_n\right)\right]=\left[e_1\left(x_1\right) \quad e_2\left(x_2\right) \quad \cdots \quad e_n\left(x_n\right)\right]$$

[5] 미지수의 개수가 방정식의 개수보다 많은 방정식을 부정방정식이라고 하며, 일반적으로 부정방정식의 해는 무수히 많을 수 있다.

[6] 고유값에 해당하는 벡터 x를 고유벡터(eigenvector)라고 부른다. 보통 고유값에 해당하는 고유벡터들은 수없이 많아서 크기(L2 norm)가 1인 표준벡터로 변환하는게 일반적이다.

→ $A\left[\left(x_1\right)\ \left(x_2\right)\ \cdots\ \left(x_n\right)\right]=\left[\left(x_1\right)\ \left(x_2\right)\ \cdots\ \left(x_n\right)\right]\begin{bmatrix} e_1 & 0 & \cdots & 0 \\ 0 & e_2 & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & 0 & \cdots & e_n \end{bmatrix}$

$P=\left[\left(x_1\right)\ \left(x_2\right)\ \cdots\ \left(x_n\right)\right]$, $D=\begin{bmatrix} e_1 & 0 & \cdots & 0 \\ 0 & e_2 & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & 0 & \cdots & e_n \end{bmatrix}$

→ $AP=PD$ → $A=PDP^{-1}$

'sympy' 함수 중에 'diagonalize()'를 이용하면 P,D를 쉽게 구할 수가 있다.

```
P,D=A.diagonalize()
D
```

$$\begin{bmatrix} 4 & 0 & 0 \\ 0 & \frac{3}{2}-\frac{\sqrt{5}}{2} & 0 \\ 0 & 0 & \frac{\sqrt{5}}{2}+\frac{3}{2} \end{bmatrix}$$

```
P
```

$$\begin{bmatrix} 0 & -\frac{1}{2}+\frac{\sqrt{5}}{2} & -\frac{\sqrt{5}}{2}-\frac{1}{2} \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

```
A-P@D@P.inv() # P.inv()는 P의 역함수
```

$$\begin{bmatrix} \frac{\sqrt{5}\left(-\frac{\sqrt{5}}{2}-\frac{1}{2}\right)\left(\frac{\sqrt{5}}{2}+\frac{3}{2}\right)}{5}-\frac{\left(-\frac{1}{2}+\frac{\sqrt{5}}{2}\right)(1-\sqrt{5})\left(\frac{3}{2}-\frac{\sqrt{5}}{2}\right)}{-5+\sqrt{5}}+2 & 0 & -1-\frac{\left(-\frac{1}{2}+\frac{\sqrt{5}}{2}\right)\left(\frac{3}{2}-\frac{\sqrt{5}}{2}\right)}{\frac{5}{2}-\frac{\sqrt{5}}{2}}-\left(\frac{1}{2}-\frac{\sqrt{5}}{10}\right)\left(-\frac{\sqrt{5}}{2}-\frac{1}{2}\right)\left(\frac{\sqrt{5}}{2}+\frac{3}{2}\right) \\ 0 & 0 & 0 \\ -1-\frac{(1-\sqrt{5})\left(\frac{3}{2}-\frac{\sqrt{5}}{2}\right)}{-5+\sqrt{5}}+\frac{\sqrt{5}\left(\frac{\sqrt{5}}{2}+\frac{3}{2}\right)}{5} & 0 & -\left(\frac{1}{2}-\frac{\sqrt{5}}{10}\right)\left(\frac{\sqrt{5}}{2}+\frac{3}{2}\right)-\frac{\frac{3}{2}-\frac{\sqrt{5}}{2}}{\frac{5}{2}-\frac{\sqrt{5}}{2}}+1 \end{bmatrix}$$

```
_.evalf() # 도는 값들을 숫자로 계산
```

$$\begin{bmatrix} -9.0\cdot10^{-128} & 0 & 1.0\cdot10^{-126} \\ 0 & 0 & 0 \\ 1.0\cdot10^{-126} & 0 & 2.0\cdot10^{-125} \end{bmatrix}$$

'sympy' 결과가 맞는지 확인하는 코드를 살펴보면 아래와 같다. 다만, 고유벡터의 크기는 1로 표준화되어 있지 않다.

```
# 고유벡터인지 검증
for i in range(3):
    out=A@P[:,i]-D[i,i]*P[:,i]
    #pprint()를 사용하면 행렬 모양으로 출력된다.
    smp.pprint(out.evalf())
```

```
⎡0⎤
⎢ ⎥
⎢0⎥
⎢ ⎥
⎣0⎦
⎡-0.e-124⎤
⎢        ⎥
⎢   0    ⎥
⎢        ⎥
⎣   0    ⎦
⎡0.e-123⎤
⎢       ⎥
⎢   0   ⎥
⎢       ⎥
⎣   0   ⎦
```

```
# 고유벡터의 크기 출력
for i in range(3):
    out=P[:,i].T@P[:,i]
    smp.pprint(out.evalf())
```

```
[1.0]
[1.38196601125011]
[3.61803398874989]
```

'sympy' 뿐 아니라 'numpy' 라이브러리를 사용해서도 고유값과 고유벡터를 구할 수 있다. 'sympy'에서는 고유값을 대각행렬로 만들어 리턴해 주지만 'numpy'에서는 벡터값으로 리턴하기 때문에 대각행렬로 만들어주는 작업이 추가로 필요하다.

```
A=np.array(A).astype(int) # 심파이형태를 넘파이로 변환
D,P=np.linalg.eig(A) # eig()함수를 이용해 고유값과 고유벡터 계산
D
```

```
array([2.61803399, 0.38196601, 4.        ])
```

```
P
```

```
array([[ 0.85065081,  0.52573111,  0.        ],
       [ 0.        ,  0.        ,  1.        ],
       [-0.52573111,  0.85065081,  0.        ]])
```

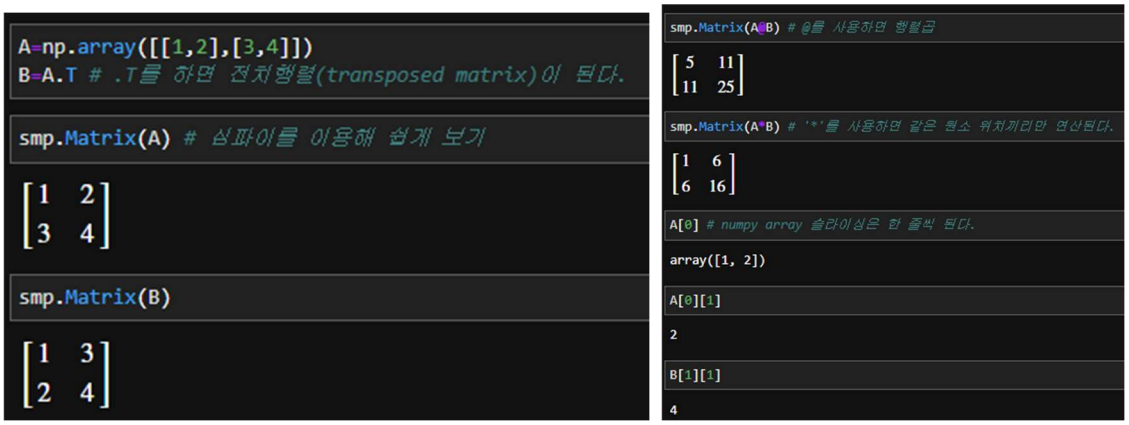
```
D=np.diag(D) #대각행렬로 변환
D
```

```
array([[2.61803399, 0.        , 0.        ],
       [0.        , 0.38196601, 0.        ],
       [0.        , 0.        , 4.        ]])
```

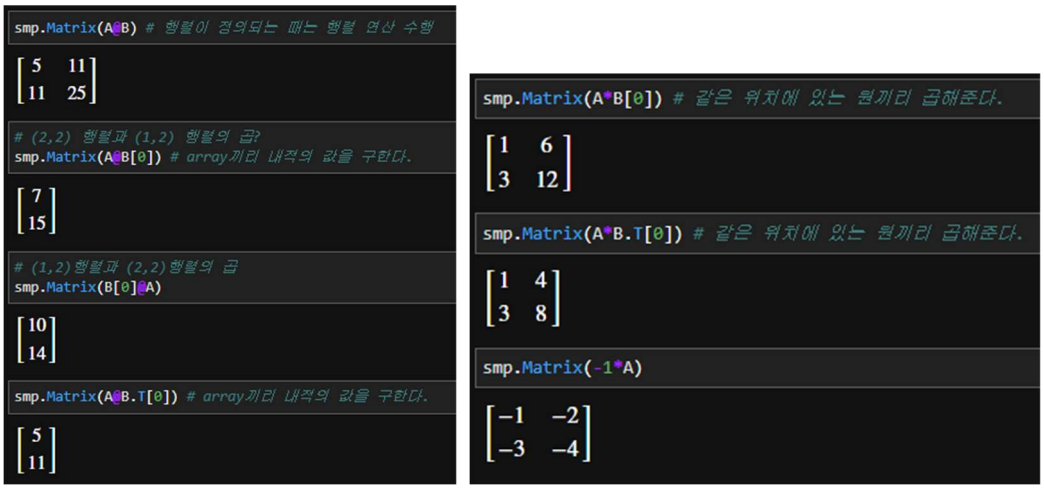
```
# AP-PD의 겨익 모든 원소가 0에 가까운지 검증
np.allclose(A@P-P@D,0)
```

```
True
```

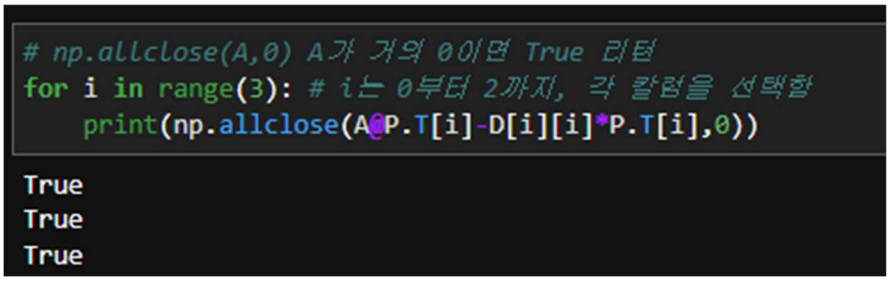
참고로 'numpy'에서 행렬을 따로 표현하는 것이 아니라 array 형식으로 표현하게 되는데, 간단히 예제를 보면서 'numpy' array의 연산 규칙을 이해할 필요가 있다.



특히, numpy '@'과 '*' 연산자의 차이를 잘 이해할 필요가 있는데, 예제를 통해서 보면서 그 차이를 명확히 알아두자[7].



'numpy'를 'eig()'를 통해서 얻어진 P 어레이에서 P.T[0]은 첫번째 고유벡터, P.T[1]은 두번째 고유벡터, P.T[2]는 세번째 고유벡터라고 이해할 수 있다.



그렇다면 $A=PDP^{-1}$로 변환시키면 어떤 장점이 생기는 것일까? 제일 큰 장점 중에 하나는 바로 행렬의 거듭제곱이 매우 쉽게 된다는 데 있다.

[7] sympy에서는 '@'와 '*' 이 동일하게 적용된다.

$$A = PDP^{-1}$$
$$A^2 = PDP^{-1} \cdot PDP^{-1} = PD^2P^{-1}$$
$$A^n = PD^nP^{-1}$$

$$where,\ D^n = \begin{bmatrix} {e_1}^n & 0 & \cdots & 0 \\ 0 & {e_2}^n & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & 0 & \cdots & {e_\mathrm{n}}^n \end{bmatrix}$$

행렬의 고유값 분해의 두번째 장점은 아래와 같은 변환에 의해 복잡하게 얽힌 변수들간의 상관관계를 제거하여 표현할 수 있다는 것이다. 아래 식에서 보면 y_i는 x_1부터 x_n까지 여러 변수들에 영향을 동시에 받게 되지만, x,y를 $w=P^{-1}x$, $z=P^{-1}y$로 변수변환을 시킨다면 $z_i=d_iw_i$의 단순한 관계식으로 바뀐다는 것을 확인할 수 있다.

$$y = Ax$$
$$\rightarrow\ y = PDP^{-1}x$$
$$\rightarrow\ P^{-1}y = DP^{-1}x,\ z \equiv P^{-1}y, w \equiv P^{-1}x$$
$$\rightarrow\ z = Dw$$
$$\rightarrow \begin{pmatrix} z_1 \\ \vdots \\ z_\mathrm{n} \end{pmatrix} = \begin{bmatrix} e_1 & \cdots & 0 \\ \vdots & \ddots & 0 \\ 0 & \cdots & e_\mathrm{n} \end{bmatrix} \begin{pmatrix} w_1 \\ \vdots \\ w_\mathrm{n} \end{pmatrix}$$

문제11. 큰 행렬을 보면 작은 행렬들로 쪼개서 하나의 숫자처럼 보고 처리할 수 있다. 아래와 같이 큰 행렬을 곱셈이 서로 정의되는 방식으로 쪼개서 곱하면, 하나의 행렬을 곱하는 것처럼 연산할 수 있다. 이를 파이썬으로 실행해 보자.

$$A_1 = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, A_2 = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}, B_1 = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, B_2 = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$
$$\begin{pmatrix} A_1 \\ A_1 \end{pmatrix} (B_1 \quad B_2) = \begin{pmatrix} A_1B_1 & A_1B_2 \\ A_2B_1 & A_2B_2 \end{pmatrix},$$

'sympy'의 Matrix.vstack(A1,A2)는 A1과 A2의 행렬을 수직으로 쌓아서 새로운 행렬을 만드는 것이고, Matrix.hstack(A1,A1)는 옆으로 쌓아서 만드는 메서드이다. 행렬을 작은 행렬로 쪼개서 연산하는 것을 블록행렬(block matrix) 연산이라고 한다.

```
from sympy import Matrix, BlockMatrix
# Block matrix A 정의
A1 = Matrix([[1, 2], [3, 4]])
A2 = Matrix([[5, 6], [7, 8]])
A = Matrix.vstack(A1,A2)

# Block matrix B 정의
B1 = Matrix([[1, 2], [3, 4]])
B2 = Matrix([[5, 6], [7, 8]])
B = Matrix.hstack(B1,B2)

A*B
```

$$\begin{bmatrix} 7 & 10 & 19 & 22 \\ 15 & 22 & 43 & 50 \\ 23 & 34 & 67 & 78 \\ 31 & 46 & 91 & 106 \end{bmatrix}$$

A1*B1

$$\begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix}$$

A1*B2

$$\begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

A2*B1

$$\begin{bmatrix} 23 & 34 \\ 31 & 46 \end{bmatrix}$$

A2*B2

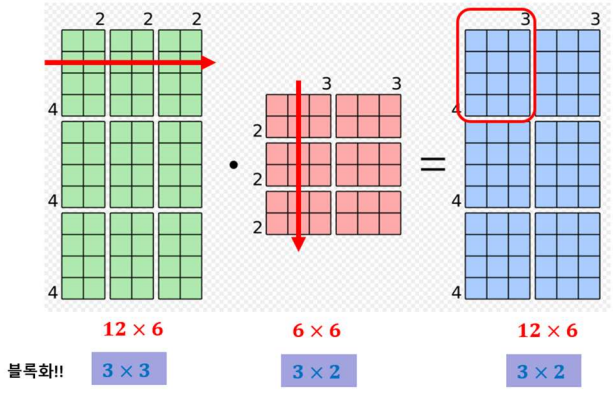
$$\begin{bmatrix} 67 & 78 \\ 91 & 106 \end{bmatrix}$$

'numpy'에서도 동일한 이름으로 vstack(), hstck() 함수가 있으니 참고하기 바란다.

```
# numpy를 이용해서 검증
A1=np.array(A1).astype(int)
A2=np.array(A2).astype(int)
A=np.vstack((A1,A2))
B1=np.array(B1).astype(int)
B2=np.array(B2).astype(int)
B=np.hstack((B1,B2))
Matrix(A@B)
```

$$\begin{bmatrix} 7 & 10 & 19 & 22 \\ 15 & 22 & 43 & 50 \\ 23 & 34 & 67 & 78 \\ 31 & 46 & 91 & 106 \end{bmatrix}$$

블록행렬의 곱은 컴퓨터가 수행할 때는 큰 의미가 없을 수도 있지만, 우리가 직관적으로 여러가지 분석을 할 때 용이하니 개념 정도만 알아두면 된다.

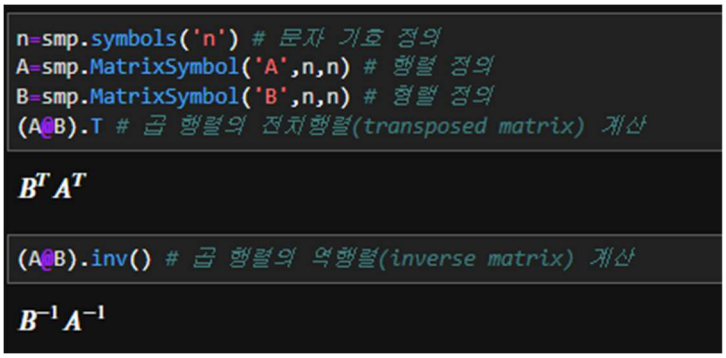



```
from sympy import Matrix, BlockMatrix, symbols

# 심볼 정의
a, b, c, d, e, f, g, h = symbols('a b c d e f g h')

# Block matrix A 정의
A1 = Matrix([[a, b], [c, d]])
A2 = Matrix([[e, f], [g, h]])
A = BlockMatrix([[A1,A2]])

# Block matrix B 정의
B1 = Matrix([[a, b], [c, d]])
B2 = Matrix([[e, f], [g, h]])
B = BlockMatrix([[B1],[B2]])

# Block matrix 곱셈
A * B
```

$$\begin{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} & \begin{bmatrix} e & f \\ g & h \end{bmatrix} \end{bmatrix} \begin{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} \\ \begin{bmatrix} e & f \\ g & h \end{bmatrix} \end{bmatrix}$$

문제12. 정방행렬[8] A가 만일 대칭행렬[9] 이라면 A=PDP^{-1}와 같이 고유값 분해(eigenvalue decomposition)를 수행했을 때, P 행렬의 열을 구성하고 있는 고유벡터들은 모두 서로 직교함[10]을 증명해 보아라.

A,B가 곱셈이 정의되는 행렬일 때, $(AB)^T=B^T A^T$, $(AB)^{-1}=B^{-1}A^{-1}$임이 증명되어 있다. 이 관계는 사실은 'sympy'를 이용해서도 확인해 볼 수 있다.



이를 이용하여 고유벡터들이 직교함을 아래와 같이 보일 수 있다.

서로 다른 고유값에 대한 고유벡터들을 상정해 보자. 즉, $e_i \neq e_j$ 일 때, 아래와 같이 전개된다.

$$A\left(x_i\right)=e_i\left(x_i\right),\ A\left(x_j\right)=e_j\left(x_j\right)$$

$$A\left(x_i\right)=e_i\left(x_i\right) \xrightarrow{\text{양변에 다른 고유벡터 곱하기}} \left(x_j\right)^T A\left(x_i\right)=\left(x_j\right)^T e_i\left(x_i\right)$$

$$A\left(x_j\right)=e_j\left(x_j\right) \xrightarrow{\text{양변을 전치행렬화}} \left(x_j\right)^T A^T=e_j\left(x_j\right)^T \xrightarrow{A=A^T} \left(x_j\right)^T A=e_j\left(x_j\right)^T$$

$$e_j\left(x_j\right)^T\left(x_i\right)=e_i\left(x_j\right)^T\left(x_i\right)$$

$$\therefore \left(x_j\right)^T\left(x_i\right)=0$$

[8] 행의 개수와 열이 개수의 같은 행렬을 정방행렬(square matrix)라고 한다.

[9] A=A^{T}가 성립하는 정방행렬을 의미한다.

[10] 두 벡터의 내적(inner product)이 0인 서로 다른 두 벡터를 직교한다고 말한다.

따라서, $\left(x_j\right)$와 $\left(x_i\right)$는 서로 직교한다.

아래 대칭행렬에 대해서 고유벡터들이 서로 직교함을 수치적으로도 살펴보자.

$$\begin{bmatrix} 2 & 0 & -1 \\ 0 & 4 & 0 \\ -1 & 0 & 1 \end{bmatrix}$$

```
import sympy as smp
import numpy as np

A=smp.Matrix(3,3,[2,0,-1,0,4,0,-1,0,1])
A-A.T #대칭행렬 검증
```

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

```
A=np.array(A).astype(int)
D,P=np.linalg.eig(A)
D=np.diag(D)
```

```
# 서로 같은 고유벡터의 내적값 확인
for i in range(3):
    print(P.T[i]@P.T[i])# 내적의 값구하기
```

```
0.9999999999999998
0.9999999999999998
1.0
```

```
# 서로 다른 고유벡터의 내적값 확인
for i in range(3):  # i=0,1,2
    for j in range(i):  #j=0,...,i-1
        print(P.T[j]@P.T[i])# 내적의 값구하기
```

```
1.212722197559916e-17
0.0
0.0
```

위 결과를 보면 같은 고유벡터들끼리의 내적값은 1이며, 서로 다른 고유벡터들끼리의 내적값들은 모두 0임을 확인할 수 있다. 이 사실로 미루어 $P^{-1}=P^T$임을 추정할 수 있다.

```
P@P.T
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

```
np.allclose(np.linalg.inv(P)-P.T,0)
```

```
True
```

따라서, A가 대칭행렬이라면 $A= PDP^{-1}=PDP^T$ 로 표현가능하다.

문제13. 아래 행렬은 정방행렬이 아닌 직사각형 행렬이다. 직사각형 행렬은 AA^T를 하면 정방행렬로 변행된다. AA^T의 고유값 분해를 수행해 보아라.

$$A = \begin{pmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix}$$

'numpy'를 이용해서 계산한 후, 'sympy' 를 이용해 행렬형식으로 보면 편하다.

```
from sympy import Matrix
import numpy as np
A=np.array([[-1,1,0],[0,-1,1]])
Matrix(A)
```

$$\begin{bmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix}$$

```
D,P=np.linalg.eig(A@A.T)
D=np.diag(D)
Matrix(D)
```

$$\begin{bmatrix} 3.0 & 0 \\ 0 & 1.0 \end{bmatrix}$$

```
Matrix(P)
```

$$\begin{bmatrix} 0.707106781186547 & 0.707106781186547 \\ -0.707106781186547 & 0.707106781186547 \end{bmatrix}$$

아래와 같은 약간의 수식적인 트릭을 통해서 A=USV^{T}의 형태로 바꿀 수 있음을 알 수 있다[11].

$$\underset{m\times n}{A}\ \underset{n\times m}{A^T} = \underset{m\times m}{P}\ \underset{m\times m}{D}\ \underset{m\times m}{P^T}$$

$$= \underset{m\times m}{P}\ \underset{m\times n}{S}\ \underset{n\times m}{S^T}\ \underset{m\times m}{P^T} = \underset{m\times m}{P}\ \underset{m\times n}{S}\ \underset{n\times n}{I}\ \underset{n\times m}{S^T}\ \underset{m\times m}{P^T}$$

$$= \underset{m\times m}{P}\ \underset{m\times n}{S}\ \underset{n\times n}{V^T}\ \underset{n\times n}{V}\ \underset{n\times m}{S^T}\ \underset{m\times m}{P^T}$$

$$\therefore \underset{m\times n}{A} = \underset{m\times m}{P}\ \underset{m\times n}{S}\ \underset{n\times n}{V^T} = \underset{m\times m}{U}\ \underset{m\times n}{S}\ \underset{n\times n}{V^T}$$

위 식에서 $D = SS^T$ 이기 때문에, S의 대각행렬의 원의 제곱은 D의 대각원소인 고유값(eigenvalue)가 됨을 알 수 있으며, 이때 S의 대각행렬의 값들을 특이값(singular value)라고 부른다. 여기서 $UU^T = VV^T$ 는 모두 단위행렬[12]임을 확인하고 넘어가자.

위 문제에서 주어진 직사각형 A를 'numpy'를 이용해 특이값분해(SVD)하게 되면 아래와 같다.

---

[11] 직사각형 행렬을 이와 같이 분해하는 것을 특이값 분해(SVD:Singular Value Decomposition)이라고 한다.

[12] 정방행렬로 모든 행이나 열벡터가 서로 직교하고 각 벡터의 크기가 1인 행렬을 직교행렬(orthogonal matrix)라고 한다.

```
U,S,Vt=np.linalg.svd(A)
```

Matrix(U)

$$\begin{bmatrix} 0.707106781186548 & 0.707106781186547 \\ -0.707106781186547 & 0.707106781186548 \end{bmatrix}$$

Matrix(S)

$$\begin{bmatrix} 1.73205080756888 \\ 1.0 \end{bmatrix}$$

Matrix(Vt)

$$\begin{bmatrix} -0.408248290463863 & 0.816496580927726 & -0.408248290463863 \\ -0.707106781186547 & -2.45660405160136 \cdot 10^{-16} & 0.707106781186548 \\ 0.577350269189626 & 0.577350269189626 & 0.577350269189626 \end{bmatrix}$$

```
# 특이값의 제곱은 고유값이 된다.
S**2
```

```
array([3., 1.])
```

```
# U,V 행렬은 모두 직교행렬(orthogonal matrix)
np.allclose(U@U.T,np.eye(2))
```

```
True
```

```
np.allclose(Vt@Vt.T,np.eye(3))
```

```
True
```

SVD는 'numpy' 뿐만 아니라 'scipy' 라이브러리를 이용해서도 계산할 수 있다.

```
# scipy를 이용한 SVD
import scipy.linalg as LA
u,s,vt=LA.svd(A)
Matrix(u)
```

$$\begin{bmatrix} -0.707106781186548 & 0.707106781186547 \\ 0.707106781186547 & 0.707106781186548 \end{bmatrix}$$

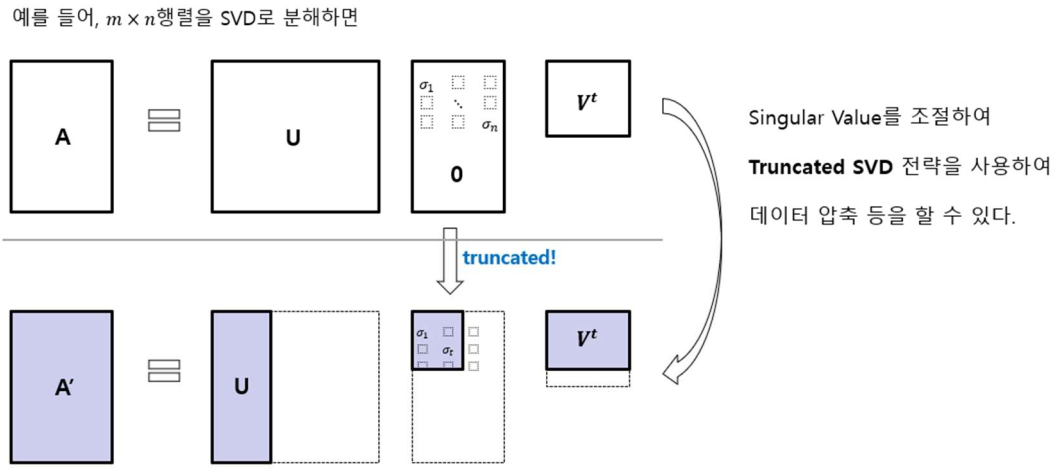
Matrix(s)

$$\begin{bmatrix} 1.73205080756888 \\ 1.0 \end{bmatrix}$$

Matrix(vt)

$$\begin{bmatrix} 0.408248290463863 & -0.816496580927726 & 0.408248290463863 \\ -0.707106781186547 & -2.77555756156289 \cdot 10^{-16} & 0.707106781186548 \\ 0.577350269189626 & 0.577350269189626 & 0.577350269189626 \end{bmatrix}$$

행렬의 특이값분해를 좀더 자세히 살펴보면, 아래와 같이 특이값이 큰 값들을 중심으로 데이터를 적당한 양으로 압축할 수 있는 수학적 근거를 만들어 준다는 사실에 있다.



SVD 를 통한 데이터 압축의 수학적 원리

$$A = U \cdot \Sigma \cdot V^t$$

A의 singular value들을 $\sigma_1, \sigma_2, \cdots, \sigma_r$이라 하자. 이를 정렬시켜 $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$

$$A = \sigma_1 U_1 V_1^T + \sigma_2 U_2 V_2^T + \cdots + \sigma_r U_r V_r^T$$

$$A \approx \sigma_1 U_1 V_1^T$$

$$A \approx \sigma_1 U_1 V_1^T + \sigma_2 U_2 V_2^T$$

$$A \approx \sigma_1 U_1 V_1^T + \sigma_2 U_2 V_2^T + \cdots + \sigma_r U_r V_r^T$$

압축률 큼

해상도 큼

위에서 기술한 방법으로 데이터를 압축하면 특이값의 일부를 가지고 U의 열과 V^t의 행을 발췌하여 자료의 정보를 줄인다는 의미로 해석된다. 앞에서 살펴보았던 아래 식에서 $\underset{m\times m}{P}$ 행렬은 고유벡터(eigenvector)들로 매우 중요한 정보를 가지고 있지만, $\underset{n\times n}{V^T}$는 상대적으로 단위행렬을 만들어주기 위한 테크닉으로 덜 중요하다고 볼 수 있다. 따라서, S의 특이값 2개를 가지고 $\underset{m\times n}{A}$ 를 압축한다고 할 때, $\underset{m\times 2}{S(1,1) \times U(:,1) + S(2,2) \times U(:,2)}$ 와 같이도 행렬의 차원을 줄일 수 있음을 알 수 있다.

$$\underset{m\times n}{A} \quad \underset{n\times m}{A^T} = \underset{m\times m}{P} \quad \underset{m\times n}{S} \quad \underset{n\times n}{V^T} \quad \underset{n\times n}{V} \quad \underset{n\times m}{S^T} \quad \underset{m\times m}{P^T}$$

$$\therefore \underset{m\times n}{A} = \underset{m\times m}{P} \quad \underset{m\times n}{S} \quad \underset{n\times n}{V^T} = \underset{m\times m}{U} \quad \underset{m\times n}{S} \quad \underset{n\times n}{V^T}$$

$$\approx \underset{m\times 2}{S(1,1) \times U(:,1) + S(2,2) \times U(:,2)}$$

실제로 임의의 행렬 (6,6)를 생성하고 이를 바탕으로 SVD를 수행하고, 특이값 2개를 바탕으로 차원을 축소해 보도록 하면 아래와 같다.

```python
import numpy as np

# 임의의 6x6 행렬 생성
X = np.random.rand(6, 6)

# numpy의 SVD 수행
U, S, Vt = np.linalg.svd(X, full_matrices=False)

# Truncated SVD: 상위 2개의 특이값만 사용하여 차원 축소
n_components = 2
X_reduced = np.dot(U[:, :n_components], np.diag(S[:n_components]))

# 결과 확인
print("원본 데이터의 모양:", X.shape)
print("축소된 데이터의 모양:", X_reduced.shape)
```

```
원본 데이터의 모양: (6, 6)
축소된 데이터의 모양: (6, 2)
```

```
S[:2] # 첫 2개의 특이값
```

```
array([2.74742139, 1.05215766])
```

```
X_reduced
```

```
array([[-1.18310281,  0.01886742],
       [-1.02953694,  0.62829119],
       [-1.00143855, -0.32398136],
       [-1.47669143,  0.26650611],
       [-1.12036784, -0.72610685],
       [-0.80617919,  0.09332393]])
```

'sklearn'에 있는 TruncatedSVD를 이용하면 훨씬 간단히 위 작업을 수행할 수 있으니 참고하도록

하자. 위의 X_reduced와 아래의 X_r의 일부 열이 부호가 뒤바뀐 것은 고유벡터 분해시 고유벡터의 부호는 달라도 성립하는 것과 관련되어 있으니 참고하자.

```
from sklearn.decomposition import TruncatedSVD

n= 2
X_r = TruncatedSVD(n_components=n)
X_r.fit_transform(X)
```

```
array([[ 1.18310281, -0.01886742],
       [ 1.02953694, -0.62829119],
       [ 1.00143855,  0.32398136],
       [ 1.47669143, -0.26650611],
       [ 1.12036784,  0.72610685],
       [ 0.80617919, -0.09332393]])
```

```
X_r.singular_values_
```

```
array([2.74742139, 1.05215766])
```

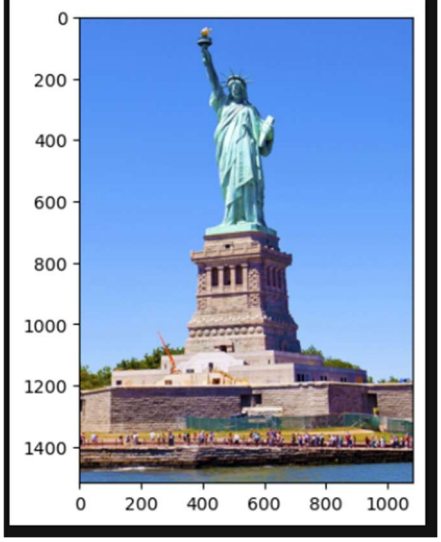
문제14. 구글에서 자유의 여신상(Statue of liberty) 사진을 다운 받아 이를 SVD 수행하여 보아라.

먼저 다운받은 파일을 디렉토리에 저장한 경로를 통해 사진을 읽어오면 아래와 같다. PIL 라이브러리[13]는 Python Imaging Library의 약자로 이미지 처리와 조작을 위한 것이다.

```
file='C:\Temp\liberty.jpg'
```

```
from PIL import Image
import matplotlib.pyplot as plt
image=Image.open(file) # 파일열기
plt.imshow(image) # 이미지 출력하기
```

```
<matplotlib.image.AxesImage at 0x2c29d9e01c0>
```



먼저 위 이미지의 데이터를 numpy로 변형하여 그 크기를 살펴보자. 가로는 1517, 세로는 1084,

---

[13] conda install -c conda-forge Pillow

그리고 3은 각 픽셀에서의 (R,G,B) 정보라고 보면 된다.

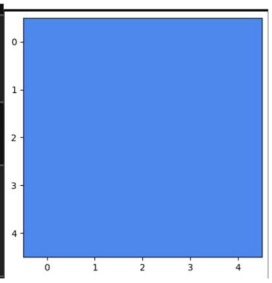
```
# 읽어온 이미지의 크기 살펴보기
image_N=np.array(image) # 이미지 파일을 넘파이 어레이로 변환
image_N.shape # 변환된 어레이의 크기, 모양 보기
```

```
(1517, 1084, 3)
```

```
# 맨 왼쪽 상단의 픽셀정보보기
C=image_N[0,0,:]
C
```

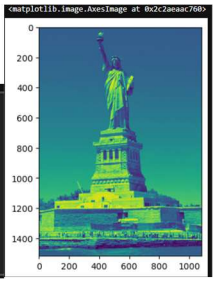
```
array([ 78, 136, 236], dtype=uint8)
```

```
# 위 색깔 정보를 색으로 표현하기
D_img=np.zeros([5,5,3],dtype=np.uint8) # 가로 5, 세로 5 빈공간
D_img[:,:,:]=C # 모든 픽셀을 C 값으로
plt.imshow(D_img) # 이미지출력
```



원래 이미지가 (1517,1084, 3)의 3차원 텐서 데이터이기 때문에, SVD를 수행하기 위해서는 2차원 데이터로 축약해야 한다. 이를 위해 픽셀의 R 기준(band=0)으로 일부 데이터를 받아서 축약하는 과정을 거쳤다.

```
# getdata()는 픽셀정보를 가져온다.
# band=0는 RGB에서 R에 대한 정보를 가져온다.
imageM=np.array(image.getdata(band=0),int)
# 보통 이미지를 다루는 라이브러리는 픽셀을 열 기준으로 먼저 읽고
# 넘파이는 행 기준으로 먼저 읽기 때문에, 넘파이의 가로, 세로를 바꿔줘야 함
imageM.shape=(image.size[1],image.size[0])
plt.imshow(imageM)
```



이 과정을 거친 이미지의 크기를 살펴보면 아래와 같이 2차원으로 변환되었음을 확인할 수 있다.

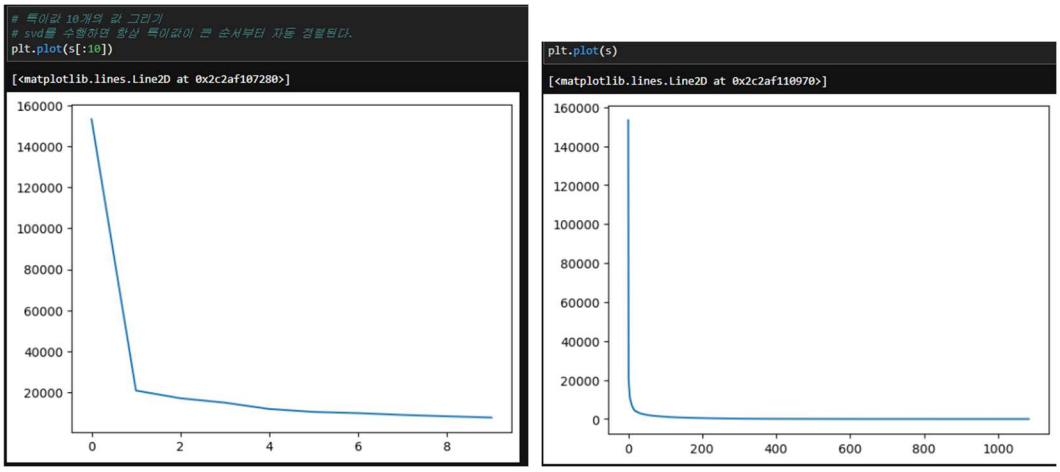
```
imageM.shape
```

```
(1517, 1084)
```

이제 이런 데이터를 바탕으로 SVD를 수행하여 보자 .

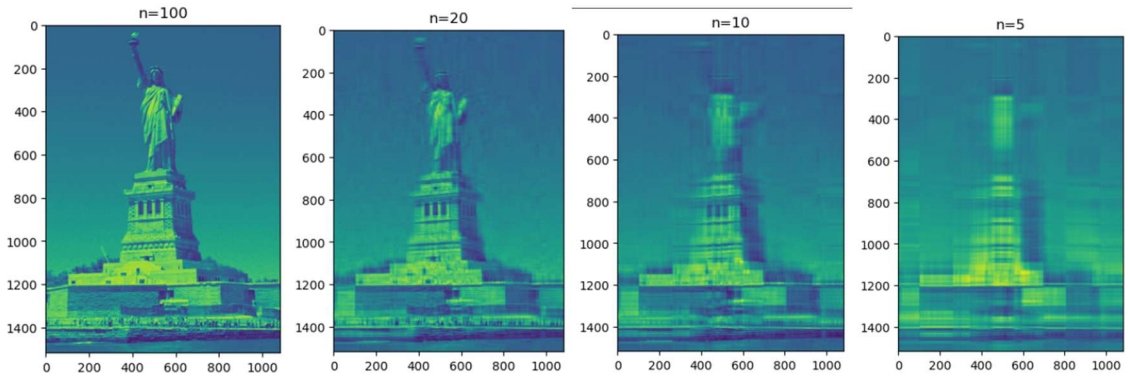
```
from scipy.linalg import svd
u,s,vt=svd(imageM) # svd 수행
print(u.shape,s.shape,vt.shape)
```

```
(1517, 1517) (1084,) (1084, 1084)
```



위 데이터를 보면 특이값은 모두 1084이며, 상위 몇 개의 데이터를 넘어가면 거의 0에 가까운 값임을 알 수 있다. 특이값에 해당하는 u행렬의 열과, vt 행렬의 행을 추출하여, 특이값이 큰 순서대로 상위 n개를 추출하여 만든 행렬을 바탕으로 이미지를 그려보면 아래와 같다.

```
# 데이터의 일부를 발췌하여 이미지 다시 그려보기
n=20 # 불러올 특이값 개수 설정
IM=np.zeros([1517,1084])

# 원래 행렬에서 특이값이 큰 순서대로 n개에 해당하는 행렬로 축약
for i in range(n): # 첫번째 부터 n-1번째 특이값까지
    # u 행렬의 i번째 열, vt행렬의 i번째 행을 불러서 행렬 곱 수행 (1517 by 1084 행렬이 됨)
    # 이렇게 만든 행렬에 s[i]의 고유값을 곱해서 행렬 업데이트
    # 업데이트된 행렬을 기존 행렬(IM)에 계속 더해감
    IM+=s[i]*(u[:,i].reshape(-1,1)@vt[i,:].reshape(1,-1))
plt.imshow(IM)
plt.title(f'n={n}')
```



출력된 사진을 보면, 추출되는 n이 줄어들수록 사진이 희미해져 가는 것을 확인할 수 있다. 원본 데이터는 1,084개의 특이값으로 만들어진 사진이지만, 특이값 100개로도 상당히 선명한 사진으로서의 역할은 수행하고 있다고 볼 수 있다. 특이값 분해는 이렇게 데이터를 압축하는데 매우 효율적으로 사용될 수 있다.

문제15. D=['coke','coke hamburger pizza','hamburger','라면','짜장면 라면','라면 떡볶이']라는 문장 리스트가 있다. 위와 같은 문자열은 아래와 같은 방법에 의해 숫자로 된 행렬(X)로 변환이 될 수 있다. X를 특이값이 2개인 행렬로 압축하여 (6,2)의 형태로 출력해 보아라.

```
from sklearn.feature_extraction.text import TfidfVectorizer

# TF-IDF 벡터화 객체 생성
vectorizer = TfidfVectorizer()

# TF-IDF 벡터화
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(D)
X = X.toarray() # 밀집행렬로 변환
```

위에서 사용된 'TfidfVectorizer'는 텍스트마이닝 편에서 더 자세히 다루겠지만, 문서에서 단어를 분리하여 숫자 행렬로 만들어주는 역할을 한다. D의 각 원소는 문서라고 볼 수 있고, 각 문서는 1~3개의 단어로 구성되어 있다고 볼 수 있다.

위에서 만들어진 X를 특이값이 2개인 SVD를 수행하면 아래와 같은 결과로 나타난다. 앞에서 살펴본 것처럼, 'sklearn'의 TruncatedSVD를 사용하면 행의 차원은 그대로 유지하면서 열의 차원을 줄여준다.

```
from sklearn.decomposition import TruncatedSVD

# LSA 수행
num_topics = 2
lsa_model = TruncatedSVD(n_components=num_topics)
lsa_topic_matrix = lsa_model.fit_transform(X)

# 단어 목록
terms = vectorizer.get_feature_names_out()
```

```
terms
```

```
array(['coke', 'hamburger', 'pizza', '떡볶이', '라면', '짜장면'], dtype=object)
```

분류된 방법을 바탕으로 어떤 키워드들이 나누어졌는지 살펴보면 아래와 같이 확인해 볼 수 있다.

```
# 주제와 관련된 주요 단어 출력
for i, topic in enumerate(lsa_model.components_):
    top_keywords_idx = topic.argsort()[-3:][::-1]  # 주요 단어 3개 추출
    top_keywords = [terms[idx] for idx in top_keywords_idx]
    print(f"Topic {i + 1}: {', '.join(top_keywords)}")

# 문서별 주제 할당
document_topics = lsa_model.transform(X)
print("\nDocument-Topic Matrix:")
print(document_topics)
```

```
Topic 1: 라면, 떡볶이, 짜장면
Topic 2: coke, hamburger, pizza
```

```
Topic 1: 라면, 떡볶이, 짜장면
Topic 2: coke, hamburger, pizza

Document-Topic Matrix:
[[ 0.00000000e+00  6.62819653e-01]
 [ 7.51212437e-17  9.37368542e-01]
 [ 6.45296364e-16  6.62819653e-01]
 [ 8.92153162e-01 -2.58329933e-16]
 [ 7.70451748e-01 -2.83442653e-16]
 [ 7.70451748e-01 -1.47044655e-16]]
```

위의 결과를 살펴보면, 뭔가 관련된 단어를 중심으로 Topic이 구분되어져 있다는 것을 알 수 있다. 문장 리스트를 '문서'라고 생각하면, 각 문서에는 수없이 많은 단어들이 들어갈 것이고, 각 문서에 들어있는 단어들의 빈도수 등을 바탕으로 SVD를 수행하여 특이값 중심으로 차원을 축소하여 문서의 의미를 파악하려는 방법론을 잠재 의미 분석(LSA)[14]이라고 부른다.

[14] 잠재 의미 분석(Latent Semantic Analysis, LSA)은 텍스트 문서나 단어들 사이의 의미적 유사성

참고문헌.

1. 주피터랩파일: Distance analysis.ipynb
2. 주피터랩파일: 데이터의 차원.ipynb
3. 네이버 지식백과: 마할라노비스 거리
4. 네이버 블로그: 군집분석
5. 네이버 블로그: 데이터의 거리 계산
6. 네이버 블로그: 데이터의 유사성 측정방법
7. 구글 자유의 여신상 사진: 하이퍼 링크
8. 네이버 블로그: 잠재의미분석
9. 유투브: 잠재의미분석

을 추출하기 위한 통계적 방법에 바탕을 둔 자연어 처리 기법 중 하나이다.