부록 B 선형대수 기초

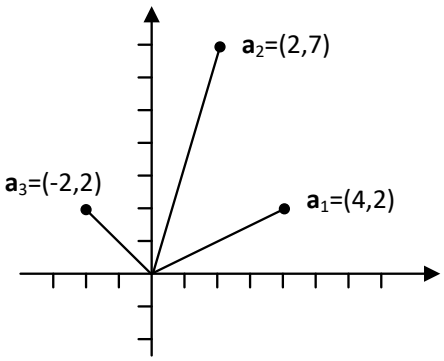
선형대수는 벡터vector와 행렬matrix을 중요하게 다룬다. 먼저 벡터와 행렬을 어떻게 표현하는지 소개하고 벡터와 행렬에 적용하는 다양한 연산을 설명한다.

벡터와 행렬의 표현

벡터 $\mathbf{a}$는 식 (B.1)로 표현한다. 벡터는 굵은 체로 표기하고, 벡터를 구성하는 요소는 이탤릭 체로 표기하며 아래 첨자를 붙여 요소끼리 구분한다. d는 벡터를 구성하는 요소의 개수인데, 벡터의 차원dimension이라 부른다. 수학에서는 벡터가 d차원 실수 공간의 한 점이라는 뜻에서 $\mathbf{a} \in \mathbb{R}^d$로 표현한다. $\mathbb{R}^d$는 축이 d개인 실수 공간이다.

$$\mathbf{a} = (a_1\ a_2\ \cdots\ a_d) \qquad \text{(B.1)}$$

[그림 B-1]은 d=2인 경우의 예를 보여준다. 벡터가 여러 개면 벡터에 아래 첨자를 붙여 $\mathbf{a}_1$, $\mathbf{a}_2$, $\mathbf{a}_3$처럼 구분하거나 $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$처럼 구분한다. 상황에 따라 편리한 방식을 사용하면 된다.



[그림 B-1] 2차원 벡터

행렬은 여러 개의 벡터를 한꺼번에 담는다. [그림 B-1]에 있는 세 개의 2차원 벡터를 행렬에 담아 표현하면 $\mathbf{A} = \begin{pmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \mathbf{a}_3 \end{pmatrix} = \begin{pmatrix} 4 & 2 \\ 2 & 7 \\ -2 & 2 \end{pmatrix}$이다.

식 (B.2)는 행렬을 표현한다. 행렬은 대문자로 표기하여 벡터와 구분한다. 행렬은 행row과 열column로 구성되는데, r은 행의 개수이고 c는 열의 개수이다. $\mathbf{A}$는 r*c 행렬이라고 말한다. 행렬 $\mathbf{A}$의 요소 a_{ij}는 i번째 행의 j번째 열에 있는 값이다. i번째 행 $\mathbf{a}_i = (a_{i1}a_{i2}\cdots a_{ic})$는 행렬을 구성하는 행 벡터row vector이다.

[TIP] r*c는 r by c라고 읽는다.

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1c} \\ a_{21} & a_{22} & \cdots & a_{2c} \\ \vdots & \vdots & \ddots & \vdots \\ a_{r1} & a_{r2} & \cdots & a_{rc} \end{pmatrix} = \begin{pmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \vdots \\ \mathbf{a}_r \end{pmatrix} \quad \text{(B.2)}$$

몇 가지 특수한 행렬이 있다. 행의 개수 r과 열의 개수 c가 같은 행렬을 정방 행렬$_{\text{square matrix}}$, 대각선을 제외한 요소가 모두 0인 행렬을 대각 행렬$_{\text{diagonal matrix}}$이라고 한다. 대각 행렬은 반드시 정방일 필요는 없다. 모든 대각선 요소가 1이면서 정방이고 대각인 행렬을 단위 행렬$_{\text{identity matrix}}$이라고 부른다. 보통 단위 행렬을 **I**로 표기한다. a_{ij}와 a_{ji}가 같은 행렬을 대칭 행렬$_{\text{symmetric matrix}}$이라고 한다.

$$\text{정방 행렬} \begin{pmatrix} 2 & 0 & 1 \\ 1 & 21 & 5 \\ 4 & 5 & 12 \end{pmatrix}, \quad \text{대각 행렬} \begin{pmatrix} 50 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 8 \end{pmatrix},$$

$$\text{단위 행렬} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \text{대칭 행렬} \begin{pmatrix} 1 & 2 & 11 \\ 2 & 21 & 5 \\ 11 & 5 & 1 \end{pmatrix}$$

데이터셋을 벡터와 행렬로 표현

컴퓨터 비전은 데이터를 다루는 학문이다. 가장 단순한 축에 속하는 iris 데이어셋을 가지고 행렬 표현을 설명한다. 세 줄로 구성된 다음 코드를 실행해보자.

```
In [1]: from sklearn import datasets
In [2]: d=datasets.load_iris()
In [3]: print(d.data)
[[5.5 3.5 1.4 0.2]
 [4.9 3.   1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.   3.6 1.4 0.2]
 [5.4 3.9 1.7 0.4]
 [4.6 3.4 1.4 0.3]
 [5.   3.4 1.5 0.2]
…
…
 [5.9 3.0 5.1 1.8]]
```

iris 데이터셋은 붓꽃에 대한 정보를 담고 있는데, 150개 샘플로 구성되며 각각의 샘플은 4개 특징으로 표현된다. 4개 특징은 꽃받침의 길이, 꽃받침의 너비, 꽃잎의 길이, 꽃잎의 너비에 해당한다. 다시 말해 iris는 4차원 특징 벡터 150개로 구성된다. 앞의 코드가 출력한 샘플을 식 (B.1)에 맞추어 표현하면 다음과 같다.

iris의 첫번째 샘플의 특징 벡터: $\mathbf{a}_1 = (5.5 \quad 3.5 \quad 1.4 \quad 0.2)$
iris의 두번째 샘플의 특징 벡터: $\mathbf{a}_2 = (4.9 \quad 3.0 \quad 1.4 \quad 0.2)$
iris의 세번째 샘플의 특징 벡터: $\mathbf{a}_3 = (4.7 \quad 3.2 \quad 1.3 \quad 0.2)$
…
iris의 150번째 샘플의 특징 벡터: $\mathbf{a}_{150} = (5.9 \quad 3.0 \quad 5.1 \quad 1.8)$

전체 샘플을 식 (B.2)와 같이 행렬에 담으면 행의 개수는 150, 열의 개수는 4인 150*4 행렬이 된다.

$$\mathbf{A} = \begin{pmatrix} 5.5 & 3.5 & 1.4 & 0.2 \\ 4.9 & 3.0 & 1.4 & 0.2 \\ 4.7 & 3.2 & 1.3 & 0.2 \\ \vdots & \vdots & \vdots & \vdots \\ 5.9 & 3.0 & 5.1 & 1.8 \end{pmatrix} = \begin{pmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \mathbf{a}_3 \\ \vdots \\ \mathbf{a}_{150} \end{pmatrix}$$

벡터 연산

스칼라에 덧셈과 곱셈, 역수 등의 연산을 수행하는 것처럼 벡터와 행렬에도 비슷한 연산을 적용할 수 있다. [그림 B-1]의 벡터 $\mathbf{a}_1 = (4 \;\; 2)$, $\mathbf{a}_2 = (2 \;\; 7)$, $\mathbf{a}_3 = (-2 \;\; 2)$로 벡터 연산을 설명한다. 벡터의 내적$_{\text{dot product}}$은 요소끼리 곱한 결과를 더하는 연산이다. 벡터 연산은 차원이 같은 벡터끼리만 가능하다. 마지막 연산은 차원이 2와 3인 벡터를 더할 수 없음을 보여준다.

- 벡터에 스칼라 곱: $3 * \mathbf{a}_1 = 3 * (4 \quad 2) = (12 \quad 6)$
- 벡터 덧셈: $\mathbf{a}_1 + \mathbf{a}_2 = (4 \quad 2) + (2 \quad 7) = (6 \quad 9)$
- 벡터의 요소별 곱: $\mathbf{a}_1\mathbf{a}_2 = (4 \quad 2)(2 \quad 7) = (8 \quad 14)$
- 벡터의 복합 연산: $\mathbf{a}_1\mathbf{a}_2 + 2 * \mathbf{a}_3 = (4 \quad 2)(2 \quad 7) + 2 * (-2 \quad 2) = (4 \quad 18)$
- 벡터의 내적: $\mathbf{a}_1 \cdot \mathbf{a}_2 = (4 \quad 2) \cdot (2 \quad 7) = 4 * 2 + 2 * 7 = 22$
- 벡터 덧셈 오류: $(4 \quad 2) + (2 \quad 7 \quad 1)$ 요소의 개수가 달라 오류 발생

다음 코드는 numpy 모듈을 이용하여 앞의 벡터 연산을 한다. [4]행은 numpy 모듈을 불러온다. [5]~[7]행은 벡터 $\mathbf{a}_1$~$\mathbf{a}_3$을 만든다. [8]~[12]행은 앞에 예시한 연산을 수행한다. [12]행에서 np.dot 함수는 벡터 내적을 수행해준다. [13]행은 벡터의 차원이 다를 때 오류가 발생함을 보여준다.

```
In [4]: import numpy as np
In [5]: a1=np.array([4,2])
In [6]: a2=np.array([2,7])
In [7]: a3=np.array([-2,2])
In [8]: print(3*a1)
[12  6]
In [9]: print(a1+a2)
[6 9]
```

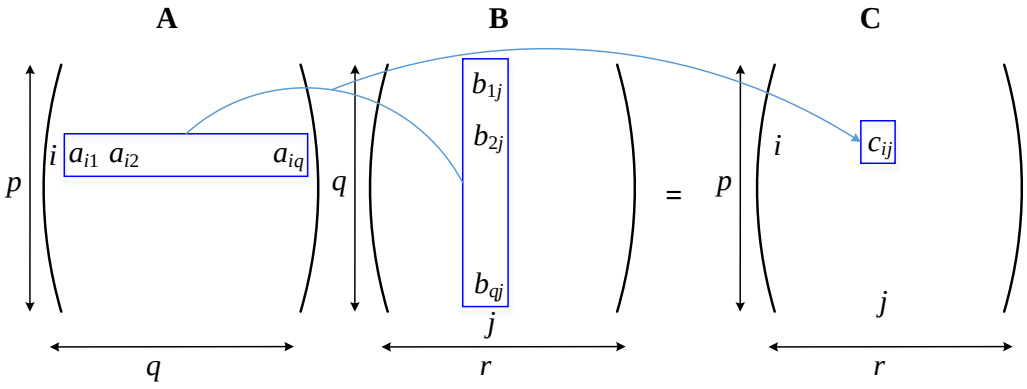
```
In [10]: print(a1*a2)
[ 8 14]
In [11]: print(a1*a2+2*a3)
[ 4 18]
In [12]: print(np.dot(a1,a2))
22
In [13]: print(np.array([4,2])+np.array([2,7,1]))
ValueError: operands could not be broadcast together with shapes (2,) (3,)
```

행렬 연산

행렬도 여러 가지 유용한 연산을 제공한다. 행렬에 스칼라를 곱하는 연산은 행렬 요소 각각에 스칼라를 곱하면 된다. 행렬의 덧셈은 두 행렬의 크기가 같을 때만 가능하며 같은 위치에 있는 요소끼리 더하면 된다. 행렬의 전치$_{\text{transpose}}$는 열과 행의 위치를 바꾸는 연산이다. **A**의 i번째 행의 j번째 열에 있는 요소가 a_{ij}라면 **A**의 전치 행렬에서는 a_{ij}가 j번째 행의 i번째 열에 위치한다. **A**가 p*q라면 **A**의 전치 행렬은 q*p이고 $\mathbf{A}^{\mathrm{T}}$로 표기한다.

행렬의 곱셈은 조금 복잡하다. 행렬 **A**와 행렬 **B**를 곱하는 연산 **AB**는 **A**의 열의 개수와 **B**의 행의 개수가 같은 경우에만 가능하다. 다시 말해 **A**가 p*q이고 **B**가 q*r이면 두 행렬을 곱할 수 있다. 예를 들어 3*2 행렬과 2*3 행렬은 곱할 수 있지만 3*2 행렬과 3*2 행렬은 곱할 수 없다. [그림 B-2]는 행렬 **A**와 **B**를 곱하여 결과를 **C**에 저장하는 과정을 보여준다. [그림 B-2]는 **C**의 요소 c_{ij}를 계산하는 과정을 설명한다. c_{ij}는 **A**의 i번째 행과 **B**의 j번째 열을 요소끼리 곱하고 결과를 더한 값이다. 이를 공식으로 쓰면 식 (B.3)이다.

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{iq}b_{qj} = \sum_{k=1.q} a_{ik}b_{kj} \qquad \text{(B.3)}$$



[그림 B-2] 행렬의 곱셈

아래에 있는 세 행렬 **A**, **B**, **C**를 가지고 행렬 연산을 설명한다. **A**는 3*2, **B**는 3*2, **C**는 2*3 행렬이다. 마지막 두 행을 보면, 행렬의 곱셈 **AB** 는 3*2와 3*2를 곱할 수 없어 오류이지만, $\mathbf{AB}^{\mathrm{T}}$에서는 3*2와 2*3를 곱해 3*3 행렬이 된다.

$$\mathbf{A}=\begin{pmatrix}4 & 2\\ 2 & 7\\ -2 & 1\end{pmatrix}, \mathbf{B}=\begin{pmatrix}0 & 1\\ 2 & 1\\ 3 & -2\end{pmatrix}, \mathbf{C}=\begin{pmatrix}1 & -2 & 3\\ 5 & 0 & 2\end{pmatrix}$$

- 행렬에 스칼라 곱: $3 * \mathbf{A} = 3 * \begin{pmatrix}4 & 2\\ 2 & 7\\ -2 & 1\end{pmatrix} = \begin{pmatrix}12 & 6\\ 6 & 21\\ -6 & 3\end{pmatrix}$
- 행렬의 덧셈: $\mathbf{A}+\mathbf{B} = \begin{pmatrix}4 & 2\\ 2 & 7\\ -2 & 1\end{pmatrix} + \begin{pmatrix}0 & 1\\ 2 & 1\\ 3 & -2\end{pmatrix} = \begin{pmatrix}4 & 3\\ 4 & 8\\ 1 & -1\end{pmatrix}$
- 행렬의 덧셈: $\mathbf{A}+\mathbf{C}$는 크기가 달라 덧셈 불가함
- 전치 행렬: $\mathbf{A}$의 전치 행렬 $\mathbf{A}^{\mathrm{T}}=\begin{pmatrix}4 & 2 & -2\\ 2 & 7 & 1\end{pmatrix}$
- 행렬의 곱셈: $\mathbf{AC} = \begin{pmatrix}4 & 2\\ 2 & 7\\ -2 & 1\end{pmatrix}\begin{pmatrix}1 & -2 & 3\\ 5 & 0 & 2\end{pmatrix} = \begin{pmatrix}14 & -8 & 16\\ 37 & -4 & 20\\ 3 & 4 & -4\end{pmatrix}$
- 행렬의 곱셈: $\mathbf{AB} = \begin{pmatrix}4 & 2\\ 2 & 7\\ -2 & 1\end{pmatrix}\begin{pmatrix}0 & 1\\ 2 & 1\\ 3 & -2\end{pmatrix}$는 크기가 맞지 않아 곱셈 불가함
- 행렬의 곱셈: $\mathbf{AB}^{\mathrm{T}} = \begin{pmatrix}4 & 2\\ 2 & 7\\ -2 & 1\end{pmatrix}\begin{pmatrix}0 & 2 & 3\\ 1 & 1 & -2\end{pmatrix} = \begin{pmatrix}2 & 10 & 8\\ 7 & 11 & -8\\ 1 & -3 & -8\end{pmatrix}$

다음 코드는 numpy 모듈을 이용하여 앞의 행렬 연산을 수행한다. [14]~[16]행은 행렬 **A**, **B**, **C**를 만들고, [17]~[23]행은 앞에서 예시한 연산을 수행한다. 행렬의 크기가 맞지 않아 연산 오류가 나는 경우 적절한 오류 메시지가 출력되는 것을 알 수 있다.

```
In [14]: A=np.array([[4,2],[2,7],[-2,1]])       # 3*2 행렬 A
In [15]: B=np.array([[0,1],[2,1],[3,-2]])       # 3*2 행렬 B
In [16]: C=np.array([[1,-2,3],[5,0,2]])         # 2*3 행렬 C
In [17]: print(3*A)
[[12  6]
 [ 6 21]
 [-6  3]]
In [18]: print(A+B)
[[ 4  3]
 [ 4  8]
 [ 1 -1]]
In [19]: print(A+C)
ValueError: operands could not be broadcast together with shapes (3,2) (2,3)
In [20]: print(A.transpose())
[[ 4  2 -2]
 [ 2  7  1]]
```

```
In [21]: print(np.matmul(A,C))
[[14 -8 16]
 [37 -4 20]
 [ 3  4 -4]]
In [22]: print(np.matmul(A,B))
ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, with gufunc signature (n?,k),(k,m?)->(n?,m?) (size 3 is different from 2)
In [23]: print(np.matmul(A,B.transpose()))
[[ 2 10  8]
 [ 7 11 -8]
 [ 1 -3 -8]]
```

역행렬

3에 3의 역수 $3^{-1}=\frac{1}{3}$을 곱하면 1이 된다. 행렬에도 비슷한 원리가 적용되는데, 행렬 $\mathbf{A}$에 역행렬 inverse matrix을 곱하면 단위 행렬 $\mathbf{I}$가 된다. $\mathbf{A}$의 역행렬을 $\mathbf{A^{-1}}$로 표기한다. 역행렬은 정방 행렬에만 적용된다. 아래 예시는 3*3 행렬 $\mathbf{A}$의 역행렬를 보여준다.

$$\mathbf{A}=\begin{pmatrix}2 & 2 & 0\\ -2 & 1 & 1\\ 3 & 0 & 1\end{pmatrix}, \mathbf{A^{-1}}=\begin{pmatrix}1/12 & -2/12 & 2/12\\ 5/12 & 2/12 & -2/12\\ -3/12 & 6/12 & 6/12\end{pmatrix}$$

$\mathbf{AA^{-1}}$를 계산하면 단위 행렬 $\mathbf{I}$가 된다.

$$\mathbf{AA^{-1}}=\begin{pmatrix}2 & 2 & 0\\ -2 & 1 & 1\\ 3 & 0 & 1\end{pmatrix}\begin{pmatrix}1/12 & -2/12 & 2/12\\ 5/12 & 2/12 & -2/12\\ -3/12 & 6/12 & 6/12\end{pmatrix}=\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{pmatrix}$$

아래 코드는 numpy를 이용하여 역행렬을 구한 다음, 원래 행렬과 역행렬을 곱하면 단위 행렬이 되는지 확인한다. 역행렬은 numpy의 linalg.inv 함수로 구한다. [26]~[27]행을 보면, 역행렬에 나타나는 1/12같은 수가 무리수이기 때문에 무시할 수 있을 정도의 작은 수치 오류가 발생했음을 알 수 있다.

```
In [24]: A=np.array([[2,2,0],[-2,1,1],[3,0,1]])
In [25]: Ainv=np.linalg.inv(A)
In [26]: print(Ainv)
[[ 0.08333333 -0.16666667  0.16666667]
 [ 0.41666667  0.16666667 -0.16666667]
 [-0.25        0.5         0.5       ]]
In [27]: print(np.matmul(A,Ainv))
[[ 1.00000000e+00 -5.55111512e-17  0.00000000e+00]
 [-1.38777878e-16  1.00000000e+00  2.22044605e-16]
 [ 0.00000000e+00  0.00000000e+00  1.00000000e+00]]
```

행렬의 고윳값과 고유 벡터

식 (B.4)를 만족하는 0이 아닌 실수 λ를 행렬 **A**의 고윳값$_{\text{eigen value}}$이라고 하며, 고윳값에 대응하는 벡터 **v**를 고유 벡터$_{\text{eigen vector}}$라고 한다.

$$\mathbf{Av} = \lambda\mathbf{v} \qquad \text{(B. 4)}$$

예를 들어 $\mathbf{A} = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$라 하면 $\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \end{pmatrix} = 3\begin{pmatrix} 1 \\ 1 \end{pmatrix}$이 성립하므로 3과 $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$은 **A**의 고윳값과 고유 벡터이다. 그런데 $\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}\begin{pmatrix} 1 \\ -1 \end{pmatrix} = 1\begin{pmatrix} 1 \\ -1 \end{pmatrix}$도 성립하므로 1과 $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$은 또 다른 고윳값과 고유 벡터이다. *n*n* 행렬은 최대 *n*개의 고윳값과 고유 벡터를 가질 수 있다. 따라서 $\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$의 고윳값은 3과 1뿐이다.

여러 개의 고윳값을 구분하기 위해 보통 크기 순으로 첨자를 부여하여 $\lambda_1 = 3, \lambda_2 = 1$과 같이 표기하고 해당 고유 벡터도 $\mathbf{v}_1$과 $\mathbf{v}_2$로 표기한다. 어떤 행렬의 고윳값이 모두 0이 아니면 역행렬을 가진다는 정리가 있다. 따라서 예시한 행렬 **A**는 역행렬을 갖는다.

아래 코드에서 [29]행은 numpy의 linalg.eig 함수를 이용하여 고윳값과 고유 벡터를 계산한다. eig 함수는 두개의 반환값 w와 v를 제공하는데, 첫번째 반환값 w는 고윳값이고 두번째 반환값 v는 고유 벡터다. w는 1차원 배열, v는 2차원 배열로 표현된다. i번째 고윳값 w[i]에 해당하는 고유 벡터는 v의 i번째 열, 즉 v[:,i]에 들어있다. 고유 벡터는 길이가 1인 단위 벡터로 표현된다.

```
In [28]: A=np.array([[2,1],[1,2]])
In [29]: w,v=np.linalg.eig(A)
In [30]: print(w)
[3. 1.]
In [31]: print(v)
[[ 0.70710678 -0.70710678]
 [ 0.70710678  0.70710678]]
```

놈과 유사도

32는 20보다 크고 둘의 차이는 12이다. 벡터도 길이를 가질까? 벡터의 길이를 정의할 때는 놈$_{\text{norm}}$을 사용한다. 식 (B.5)는 벡터 $\mathbf{x} = (x_1, x_1, \cdots, x_d)$의 p차 놈이다. 예를 들어, $\mathbf{x} = (3 \quad -4 \quad 1)$일 때, 1차 놈은 $\|\mathbf{x}\|_1 = (|3| + |-4| + |1|) = 8$, 2차 놈은 $\|\mathbf{x}\|_2 = (3^2 + (-4)^2 + 1^2)^{1/2} = 5.099$, 3차 놈은 $\|\mathbf{x}\|_3 = (3^3 + |-4|^3 + 1^3)^{1/3} = 4.514$이다. 2차 놈 $\|\mathbf{x}\|_2$를 유클리디언 놈$_{\text{Euclidean norm}}$이라 하고, 가장 널리 쓰인다는 이유로 종종 첨자를 생략하고 $\|\mathbf{x}\|$로 표기한다. $\|\mathbf{x}\|_p^q$는 p차 놈의 q 승이다. 예를 들어 $\|\mathbf{x}\|_2^2 = (3^2 + (-4)^2 + 1^2) = 26$이다. $p = \infty$일 때의 놈을 최대 놈이라 부르고, 식 (B.6)으로 정의한다.

$$p\text{차 놈: } \|\mathbf{x}\|_p = \left(\sum_{i=1,d} |x_i|^p\right)^{\frac{1}{p}} \qquad \text{(B.5)}$$

$$\text{최대 놈: } \|\mathbf{x}\|_\infty = \max(|x_1|, |x_2|, \cdots, |x_d|) \qquad \text{(B.6)}$$

numpy에 벡터의 놈을 구해주는 linalg.norm(x,ord,axis,keepdims) 함수가 있다. 첫째 매개변수 x는 벡터, 둘째 매개변수 ord는 식 (B.5)의 p에 해당한다. axis 매개변수는 x가 2차원 이상일 때 어느 축을 기준으로 적용할 지를 지정한다. [33]행은 벡터 $\mathbf{x} = (3 \quad -4 \quad 1)$에 1차,

2차, 3차, 최대 놈을 적용한다. ord 인수를 생략하면 2가 기본값이다. 따라서 np.linalg.norm(x)는 np.linalg.norm(x,2)와 같다.

```
In [32]: x=np.array([3,-4,1])
In [33]: print(np.linalg.norm(x,1),np.linalg.norm(x,2),np.linalg.norm(x,3),np.linalg.norm(x,np.inf))
8.0  5.0990195135927845  4.514357435474001  4.0
```

길이가 1인 벡터를 단위 벡터라unit vector 부르는데, 컴퓨터 비전에서 단위 벡터가 유용한 경우가 종종 있다. 벡터 **x**를 단위 벡터로 만들려면, **x**의 길이, 즉 2차 놈으로 나누면 되는데 식 (B.7)과 같다. 예를 들어, $\mathbf{x} = (3 \quad -4 \quad 1)$의 단위 벡터는 $\left(\frac{3}{5.099} \quad \frac{-4}{5.099} \quad \frac{1}{5.099}\right) = (0.588 \quad -0.784 \quad 0.196)$이다.

$$\text{단위벡터: } \frac{\mathbf{x}}{\|\mathbf{x}\|_2} \qquad \text{(B.7)}$$

아래 코드는 **x**를 단위 벡터로 만든다. numpy에는 단위 벡터로 만들어주는 함수가 없기 때문에 벡터의 놈으로 나누어 단위 벡터를 구한다.

```
In [34]: print(x/np.linalg.norm(x))
[ 0.58834841 -0.78446454  0.19611614]
```

두 벡터 사이의 거리를 측정하는 기준이 여럿인데, 여기서는 식 (B.8)이 정의하는 코사인 유사도cosine similarity를 소개한다. 코사인 유사도는 두 벡터를 단위 벡터로 변환한 다음 단위 벡터의 내적을 계산한다. 이 방식을 코사인 유사도라 한 이유는 두 벡터가 이루는 각 θ와 같기 때문이다.

$$cosine\_similarity(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a}}{\|\mathbf{a}\|} \cdot \frac{\mathbf{b}}{\|\mathbf{b}\|} = cos(\theta) \qquad \text{(B.8)}$$

다음 코드는 내적을 계산하는 dot 함수와 놈을 계산하는 linalg.norm 함수를 사용하여 두 벡터의 코사인 유사도를 계산한다. [35]행의 벡터 a는 동쪽을 가리키고, [36]행의 b는 북쪽을 가리킨다. [37]행의 c는 45도 방향의 동북쪽을 가리킨다. [38]행의 코사인 유사도는 0인데, a와 b가 수직이기 때문이다. [39]행의 코사인 유사도는 0.7071인데, a와 c가 45도를 이루기 때문이다. cos(45도)는 0.7071이다. [40]행의 코사인 유사도는 1.0인데, a와 a는 0도이기 때문이다. 코사인 유사도는 가장 멀 때가 0.0이고 가장 가까울 때가 1.0이다. 여기서는 2차원 벡터를 예시했는데, 식 (B.8)의 코사인 유사도는 일반적으로 d차원 벡터에 그대로 적용된다.

```
In [35]: a=np.array([5.0, 0.0])
In [36]: b=np.array([0.0, 3.2])
In [37]: c=np.array([2.0,2.0])
In [38]: print(np.dot(a,b)/(np.linalg.norm(a)*np.linalg.norm(c))) # 수직인 두 벡터
0.0
In [39]: print(np.dot(a,c)/(np.linalg.norm(a)*np.linalg.norm(c))) # 45도를 이루는 두 벡터
0.7071067811865475
In [40]: print(np.dot(a,a)/(np.linalg.norm(a)*np.linalg.norm(a))) # 같은 방향인 두 벡터
1.0
```