1. 행렬

행렬은 벡터를 원소로 가지는 2 차원 배열이다. 행과 열이라는 인덱스를 가진다.

수식

$$\mathbf{X} = \begin{bmatrix} 1 & -2 & 3 \\ 7 & 5 & 0 \\ -2 & -1 & 2 \end{bmatrix}$$

코드

```
X = np.array([[1 ,-2, 3],
              [7, 5 ,0],
              [-2, -1 ,2]])
```


numpy 에선 행(row)이 기본 단위입니다

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_n \end{bmatrix} = \begin{array}{c} \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1m} \\ x_{21} & x_{22} & \cdots & y_{2m} \\ \vdots & \vdots & & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix} \end{array} \begin{array}{c} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \\ \mathbf{x}_n \end{array}$$

(n 개의 행, m 개의 열)

$\mathbf{X} = (x_{ij})$ 행렬을 xij 와 같은 표기로 표현할 수도 있다.

전치행렬을 행과 열의 인덱스가 바뀐 행렬을 의미한다. 즉, n*m 행렬이 m*n 행렬로 바뀐다. 벡터에 적용하면 행 벡터는 열 벡터가 되고 열 벡터는 행 벡터가 된다.

$$\mathbf{X}^{\top} = \begin{bmatrix} x_{11} & x_{21} & \cdots & x_{n1} \\ x_{12} & x_{22} & \cdots & x_{n2} \\ \vdots & \vdots & & \vdots \\ x_{1m} & x_{2m} & \cdots & x_{nm} \end{bmatrix} \text{ 행벡터}$$

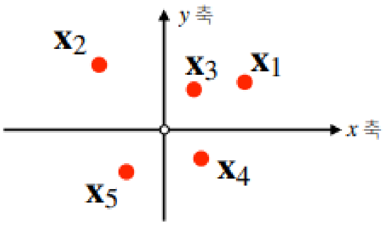
열벡터 　 $m \times n$ 행렬

$$\mathbf{X}^{\top} = (x_{ji})$$


전치행렬(transpose matrix)은 행과 열의 인덱스가 바뀐 행렬을 말합니다

벡터가 공간에서 한 점을 의미했다면 행렬은 여러 점들을 의미한다. 행렬의 행 벡터 xi 는 i 번째 데이터를 의미한다. xij 는 i 번째 데이터의 j 번째 변수 값을 의미한다.

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1m} \\ x_{21} & x_{22} & \cdots & x_{2m} \\ \vdots & \vdots & & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix} \begin{array}{c} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \\ \mathbf{x}_n \end{array}$$



2. 행렬의 연산

행렬은 벡터를 원소로 가지는 2 차원 배열이다. 따라서 벡터와 동일하게 행렬끼리 같은 모양을 가지면 **덧셈, 뺄셈을 계산**할 수 있다.

$$\mathbf{X} \pm \mathbf{Y} = (x_{ij} \pm y_{ij})$$

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1m} \\ x_{21} & x_{22} & \cdots & x_{2m} \\ \vdots & \vdots & & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix} \quad \mathbf{Y} = \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1m} \\ y_{21} & y_{22} & \cdots & y_{2m} \\ \vdots & \vdots & & \vdots \\ y_{n1} & y_{n2} & \cdots & y_{nm} \end{bmatrix}$$

$$\mathbf{X} \pm \mathbf{Y} = \begin{bmatrix} x_{11} \pm y_{11} & x_{12} \pm y_{12} & \cdots & x_{1m} \pm y_{1m} \\ x_{21} \pm y_{21} & x_{22} \pm y_{22} & \cdots & x_{2m} \pm y_{2m} \\ \vdots & \vdots & & \vdots \\ x_{n1} \pm y_{n1} & x_{n2} \pm y_{n2} & \cdots & x_{nm} \pm y_{nm} \end{bmatrix}$$

성분곱도 벡터와 동일하다. 스칼라곱도 마찬가지다.

$$\mathbf{X} \odot \mathbf{Y} = (x_{ij} y_{ij})$$

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1m} \\ x_{21} & x_{22} & \cdots & x_{2m} \\ \vdots & \vdots & & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix} \quad \mathbf{Y} = \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1m} \\ y_{21} & y_{22} & \cdots & y_{2m} \\ \vdots & \vdots & & \vdots \\ y_{n1} & y_{n2} & \cdots & y_{nm} \end{bmatrix}$$

$$\mathbf{X} \odot \mathbf{Y} = \begin{bmatrix} x_{11} y_{11} & x_{12} y_{12} & \cdots & x_{1m} y_{1m} \\ x_{21} y_{21} & x_{22} y_{22} & \cdots & x_{2m} y_{2m} \\ \vdots & \vdots & & \vdots \\ x_{n1} y_{n1} & x_{n2} y_{n2} & \cdots & x_{nm} y_{nm} \end{bmatrix}$$

3. 행렬의 곱셈

행렬 곱셈(matrix multiplication)은 i 번째 행벡터와 j 번째 열벡터 사이의 내적 성분으로 가지는 행렬을 계산한다. **x 의 행 길이는 y 의 열 길이와 같아야한다.**

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1m} \\ x_{21} & x_{22} & \cdots & x_{2m} \\ \vdots & \vdots & & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix} \qquad \mathbf{Y} = \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1\ell} \\ y_{21} & y_{22} & \cdots & y_{2\ell} \\ \vdots & \vdots & & \vdots \\ y_{m1} & y_{m2} & \cdots & y_{m\ell} \end{bmatrix}$$

$$\mathbf{XY} = \left(\sum_k x_{ik} y_{kj} \right)$$

아래에서 -8 이라는 값은 1*0 + (-2)*1 + 3*(-2)로 계산된 것이다.

```python
X = np.array([[1 ,-2, 3],
              [7, 5 ,0],
              [-2, -1 ,2]])

Y = np.array([[0 ,1],
              [1, -1],
              [-2, 1]])
```

```python
X @ Y
```

```
array([[-8,  6],
       [ 5,  2],
       [-5,  1]])
```

넘파이의 내적 함수인 **np.inner()를 사용하면 i 번째 행벡터와 j 번째 행벡터 사이의 내적을**

성분으로 가지는 행렬을 계산한다. 즉, x 의 행 길이와 y 의 행 길이가 같아야한다.

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1m} \\ x_{21} & x_{22} & \cdots & x_{2m} \\ \vdots & \vdots & & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix} \qquad \mathbf{Y} = \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1m} \\ y_{21} & y_{22} & \cdots & y_{2m} \\ \vdots & \vdots & & \vdots \\ y_{n1} & y_{n2} & \cdots & y_{nm} \end{bmatrix}$$

$$\mathbf{X}\mathbf{Y}^{\top} = \left(\sum_k x_{ik} y_{jk} \right)$$

-5 는 1*0 + (-2)*1 + 3*(-1)의 결과이다.

```
X = np.array([[1 ,-2, 3],
              [7, 5 ,0],
              [-2, -1 ,2]])

Y = np.array([[0 ,1, -1],
              [1, -1 ,0]])
```

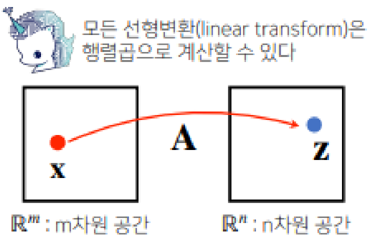
```
np.inner(X, Y)

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

4. 행렬_2

행렬은 벡터공간에서 사용되는 연산자(operator)로 이해할 수 있다. 행렬곱을 사용하면 **벡터를 다른 차원의 공간으로 보낼 수 있다**. 이런 특징을 이용해 **패턴을 추출**할 수 있고 **데이터를 압축**할 수도 있다.

$$\underset{\mathbf{z}}{\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix}} = \underset{\mathbf{A}}{\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix}} \underset{\mathbf{x}}{\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}}$$



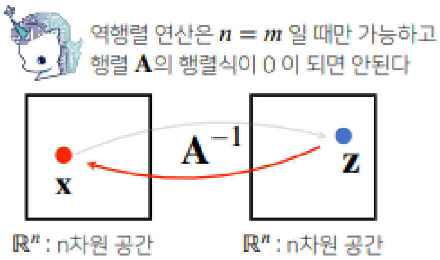
5. 역행렬

어떤 행렬 A 의 연산을 거꾸로 되돌리는 행렬을 **역행렬(inverse matrix)**이라고 부르고 A^(-1)이라 표기한다. **역행렬은 행과 열 숫자가 같고 행렬식이 0 이 아닌 경우에만 계산할 수 있다**.

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$$

항등행렬

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$



numpy.linalg.inv()로 구할 수 있다.

```
X = np.array([[1 ,-2, 3],
              [7, 5 ,0],
              [-2, -1 ,2]])
```

```
np.linalg.inv(X)
```

```
array([[ 0.21276596,  0.0212766 , -0.31914894],
       [-0.29787234,  0.17021277,  0.44680851],
       [ 0.06382979,  0.10638298,  0.40425532]])
```

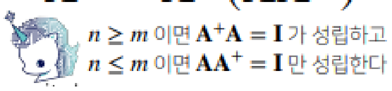
```
X @ np.linalg.inv(X)
```

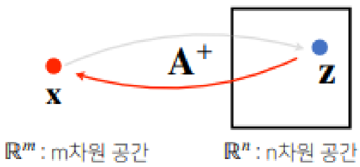
```
array([[ 1.00000000e+00, -1.38777878e-17,  0.00000000e+00],
       [-2.22044605e-16,  1.00000000e+00, -5.55111512e-17],
       [-2.77555756e-17,  0.00000000e+00,  1.00000000e+00]])
```

행과 열의 길이가 달라 역행렬을 계산할 수 없다면 **유사역행렬(pseudo-inverse) 또는 무어-펜로즈(Moore-Penrose) 역행렬** A^+을 이용할 수 있다.

$n \geq m$ 인 경우 $\mathbf{A}^{+} = (\mathbf{A}^{\top}\mathbf{A})^{-1}\mathbf{A}^{\top}$

$n \leq m$ 인 경우 $\mathbf{A}^{+} = \mathbf{A}^{\top}(\mathbf{A}\mathbf{A}^{\top})^{-1}$





numpy.linalg.pinv()로 구할 수 있다.

```
Y = np.array([[0 ,1],
              [1, -1],
              [-2, 1]])
```

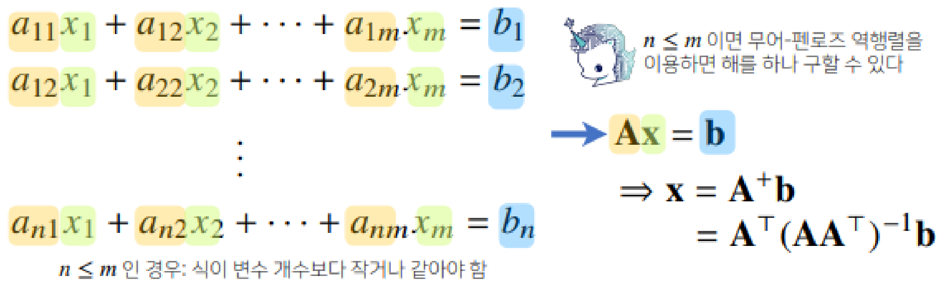
```
np.linalg.pinv(Y)
```

```
array([[ 5.00000000e-01,  4.09730229e-17, -5.00000000e-01],
       [ 8.33333333e-01, -3.33333333e-01, -1.66666667e-01]])
```

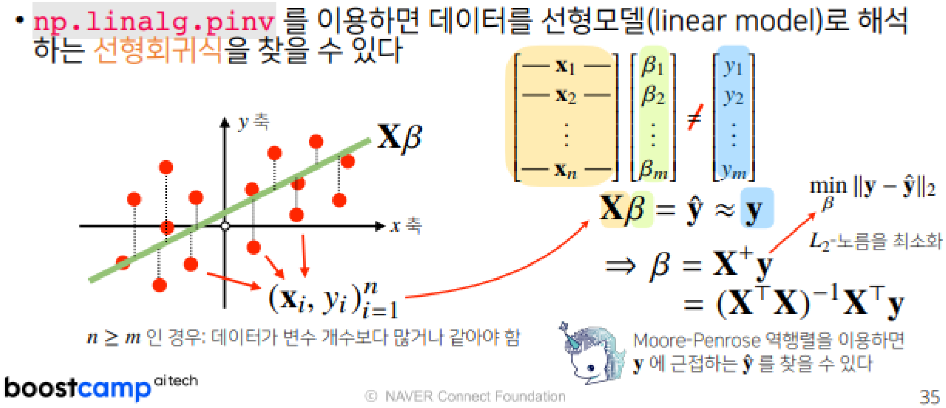
```
np.linalg.pinv(Y) @ Y
```

```
array([[ 1.00000000e+00, -2.22044605e-16],
       [ 0.00000000e+00,  1.00000000e+00]])
```

유사역행렬을 사용해 연립 방정식을 풀 수 있다.



유사역행렬을 이용해 선형 회귀 분석도 수행할 수 있다.



코드는 아래와 같이 쓸 수 있다.

```
# Scikit Learn 을 활용한 회귀분석
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(X, y)
y_test = model.predict(x_test)

# Moore-Penrose 역행렬

beta = np.linalg.pinv(X) @ y
y_test = np.append(x_test) @ beta
```

같은 방법이지만 결과가 다르게 된다. 왜일까?

두 방식의 결과값 차이는 y 절편 때문에 발생한다. sklearn 은 y 절편을 자동으로 추가해주지만 우리는 직접 추가해야한다. 완성 코드는 아래와 같다.

```
# Scikit Learn 을 활용한 회귀분석
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(X, y)
y_test = model.predict(x_test)

# Moore-Penrose 역행렬
X_ = np.array([np.append(x,[1]) for x in X]) # intercept 항 추가
beta = np.linalg.pinv(X_) @ y
y_test = np.append(x, [1]) @ beta
```

y절편(intercept) 항을 직접 추가해야 한다