Linear Algebra

week N ~ week N+ 3

Linear Algebra

Source : https://pabloinsente.github.io/intro-linear-algebra

What is Linear Algebra?

- Linear algebra is to machine learning as flour to bakery: every machine learning model is based in linear algebra, as every cake is based in flour.
- Machine learning models need vector calculus, probability, and optimization, as cakes need sugar, eggs, and butter.

Learning Objectives

- Preliminary concepts
- Vectors
- Matrices
- Linear Mapping
- Norm (L1, L2)
- Numpy Applications

Sets

- The Sets are well-defined collections of objects.
  - Such objects are called **elements or members** of the set.
  - The crew of a ship, a caravan of camels, and the Jordan of Chicago Bulls, are all examples of sets.

- Belonging
  - *a* belongs to *A* with the Greek epsilon as :

$$a \in A$$

- Inclusion
  - When every element of A is an element of B, we say that A is a subset of B or that B includes A.

$$A \subset B$$

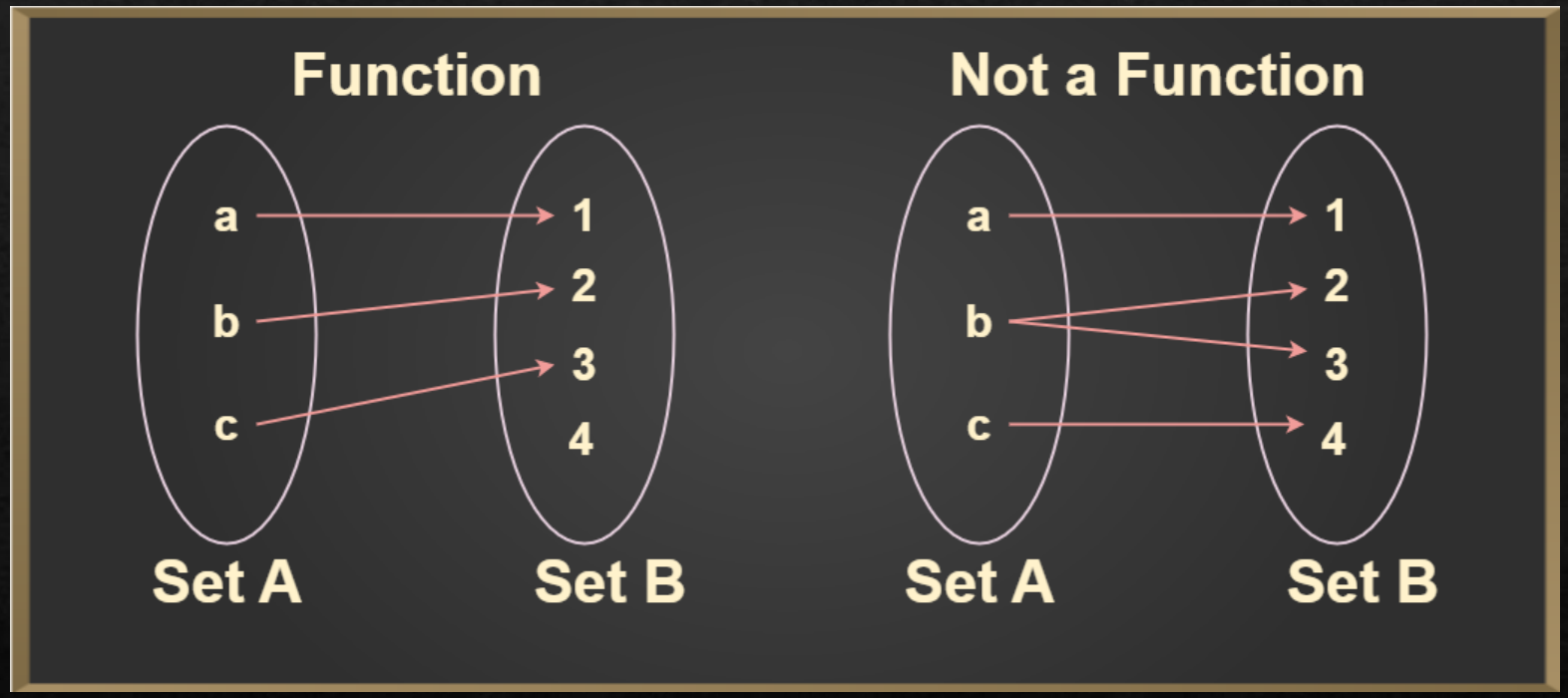
Functions

- Consider a pair of sets X and Y.
  - We say that a function from X to Y
  - More Informally. we say that a function as “transform” or “map” or “send” x to y

$$f: X \to Y$$

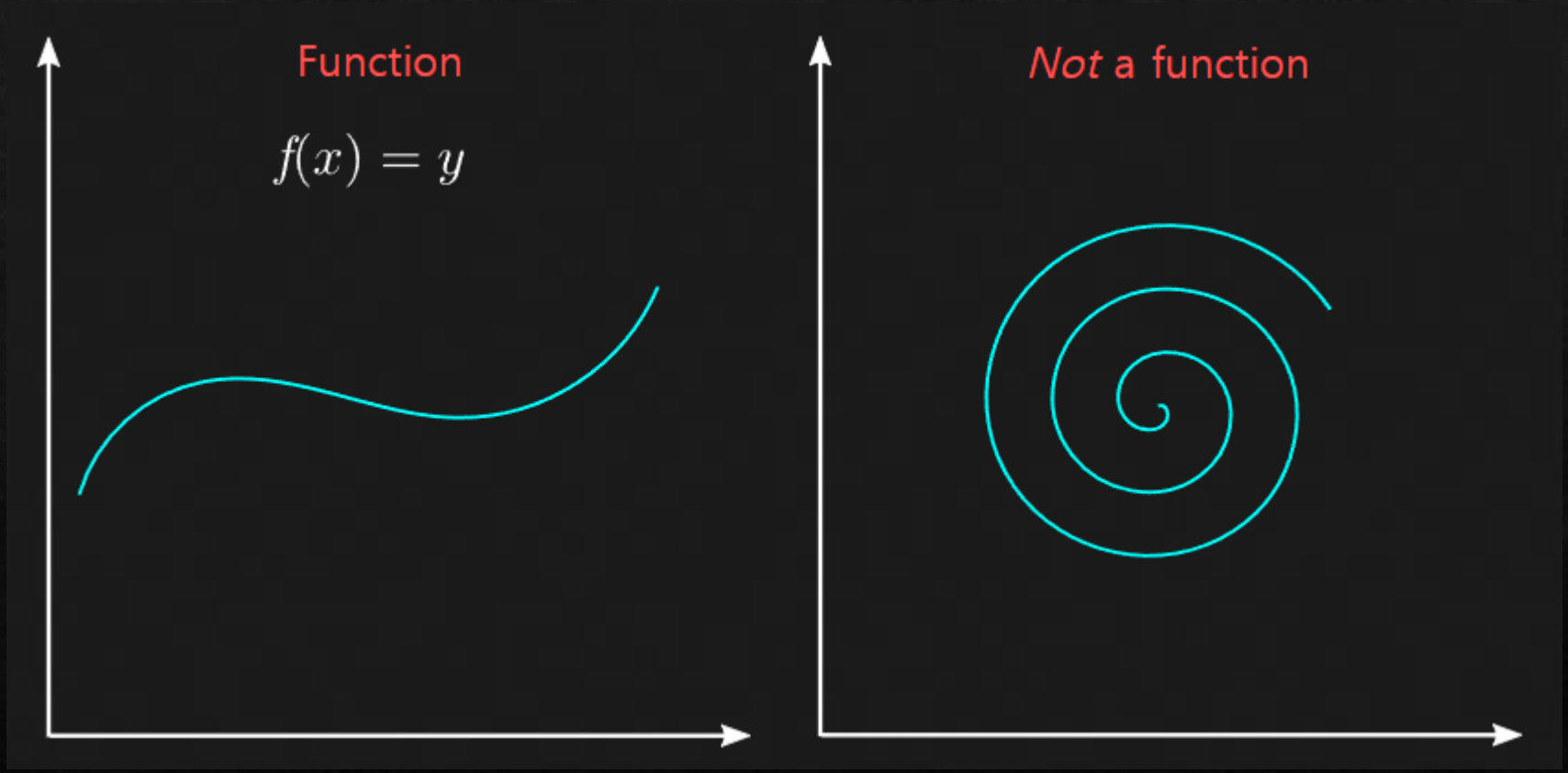
$$f(x) = y$$

Functions


Function
a
b
c
1
2
3
4
Set A
Set B
Not a Function
a
b
c
1
2
3
4
Set A
Set B

Functions

- The left-pane shows a valid function, i.e., each value f(x) maps uniquely onto one value of y
- The right-pane is not a function, since each value f(x) maps onto multiple values of y



Simple Linear Regression Function

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import linregress

# 예제 데이터 생성
X = 2 * np.random.rand(100, 1)  # 독립 변수
y = 4 +  3 * X +  np.random.randn(100, 1)  # 종속 변수 (선형 관계에 노이즈 추가)

# Scipy를 사용한 선형 회귀
slope, intercept, r_value, p_value, std_err = linregress(X.flatten(), y.flatten())

# 학습된 선형 회귀 모델의 파라미터 출력
print(f'기울기 (slope): {slope}, 절편 (intercept): {intercept}')
```

```
# 새로운 데이터에 대한 예측
X_new = np.array([0, 2]) # X의 최소값과 최대값에 대한 예측
y_predict = intercept +  slope * X_new
print("예측값:", y_predict)

# 학습된 선형 회귀 모델 시각화
plt.plot(X_new, y_predict, "r-")
plt.scatter(X, y)
plt.xlabel('X'); plt.ylabel('y')
plt.title('Simple Linear Regression with Scipy')
```

→ It's your turn : Sigmoid Function

Summary

- To understand: (1) the concept of a set, (2) basic set notation, (3) how sets are generated, (4) how sets allow the definition of functions, (5) the concept of a function.
- Set theory is a monumental field, but there is no need to learn everything about sets to understand linear algebra.

Learning Objectives

- Preliminary concepts
- Vectors
- Matrices
- Linear Mapping
- Norm (L1, L2)
- Numpy Applications

Vectors

- Linear algebra is the study of vectors which vectors are ordered finite lists of numbers.
  - Vectors are the most fundamental mathematical object in machine learning.
  - We use them to represent attributes of entities: age, sex, test scores, etc. We represent vectors by a bold lower-case letter like v or as a lower-case letter with an arrow on top like $\vec{v}$

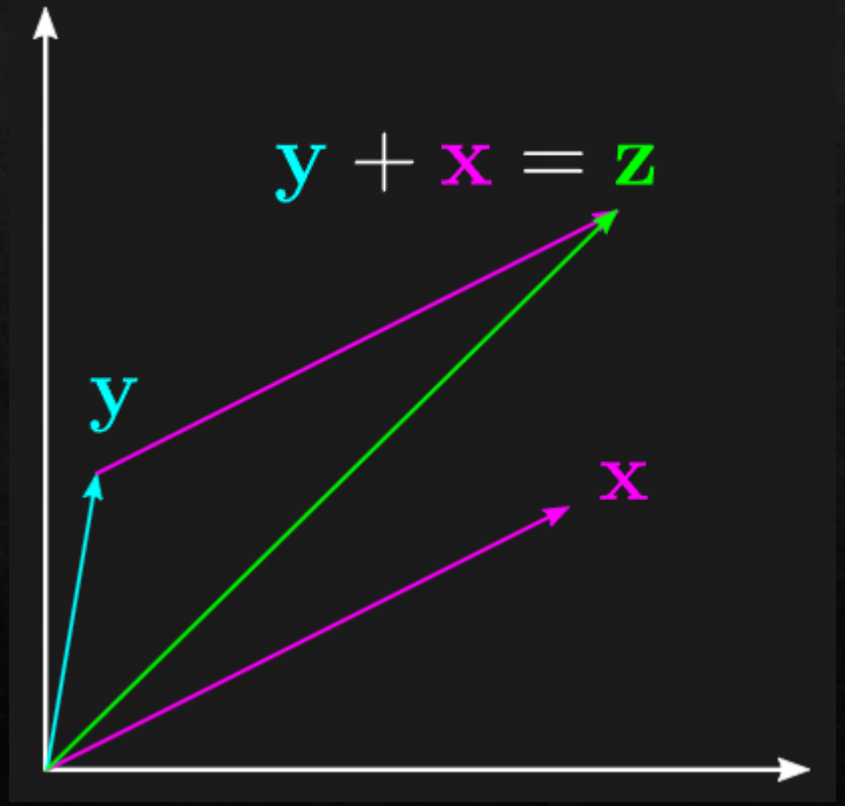
- Vectors are a type of mathematical object that can be added together and/or multiplied by a number to obtain another object of the same kind.
  - For instance, if we have a vector x=age and a second vector y=weight , we can add them together and obtain a third vector z=x+ y.
  - We can also multiply $2 \times x$ to obtain 2x , again, a vector.

- This is what we mean by the same kind: the returning object is still a vector.

The Types of Vectors

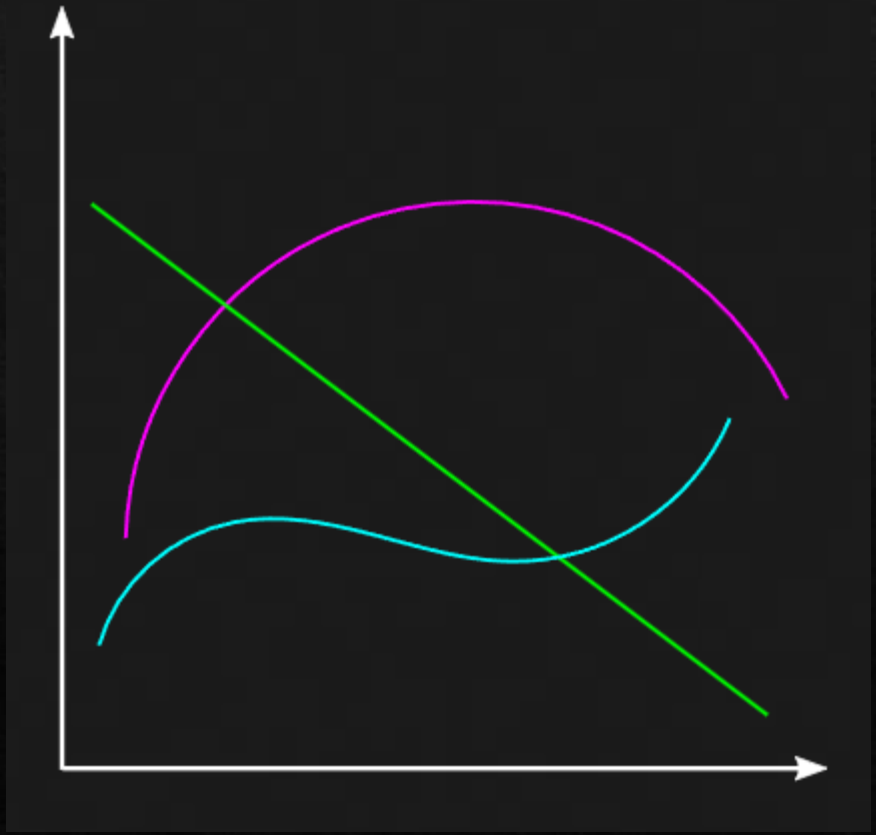
Geometric vectors

- **Geometric vectors** are oriented segments



Polynomial

- A **polynomial** is an expression like f(x)=x^2+ y+ 1
  - This is, a expression adding multiple “terms” (nomials).



The set of Real Numbers

- Elements of $\mathbb{R}^n$ are sets of real numbers.
  - This type of representation is arguably the **most important for machine learning**.
  - It is how data is commonly represented in computers to build machine learning models.

    For instance, a vector in $\mathbb{R}^3$ takes the shape of:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \in \mathbb{R}^3$$

Indicating that it contains three dimensions.

addition is valid *and* multiplying by a scalar is valid

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix} \qquad 5 \times \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 5 \\ 10 \\ 15 \end{bmatrix}$$

Numpy vectors : n-dimensional array

```
import numpy as np

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

x.shape # (3 dimensions, 1 element on each)
# (3,1)

print(f'A 3-dimensional vector:₩n{x}')
# A 3-dimensional vector:
  [[1]
   [2]
   [3]]
```

→ It's your turn

```
# Zero vector
np.zeros(3)

# Unit vector
np.eye(3)

arr = np.eye(3, dtype=int)
arr

for n in range(3):
    print(arr[:, n], '₩n')
```

Zero vector, unit vector, and sparse vector

- Zero vectors, are vectors composed of zeros, and zeros only.
- Unit vectors, are vectors composed of a single element equal to one, and the rest to zero.
- Sparse vectors, are vectors with most of its elements equal to zero.

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

$$\mathbf{x_1} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \mathbf{x_2} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \mathbf{x_3} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Sparse Matrix

```
import numpy as np
from scipy.sparse import coo_matrix

# Create a dense 10x10 matrix
dense_matrix = np.random.rand(5,5)

# Set 80% of the values to 0
threshold = np.percentile(dense_matrix, 80)
dense_matrix[dense_matrix <= threshold] = 0

# Create a sparse matrix using coo_matrix
sparse_matrix = coo_matrix(dense_matrix)

sparse_matrix

sparse_matrix.toarray()

dense_matrix
```

```
<5x5 sparse matrix of type '<class 'numpy.float64'>'
with 5 stored elements in COOrdinate format>


array([[0. , 0. , 0. , 0.65163976, 0.73859746],
       [0. , 0. , 0. , 0.71320688, 0. ],
       [0.890215 , 0. , 0. , 0. , 0. ],
       [0. , 0. , 0.83548023, 0. , 0. ],
       [0. , 0. , 0. , 0. , 0. ]])
```

Sparse matrix in Text mining

```
from sklearn.feature_extraction.text import CountVectorizer

corpus = [
    'This is the first document.',
    'This document is the second document.',
    'And this is the third one.',
    'Is this the first document?',
]

vectorizer = CountVectorizer()
X = vectorizer.fit_transform(corpus)
vectorizer.get_feature_names_out()
print(X.toarray())

pd.DataFrame(X.toarray(),
             index = [corpus[i] for i in range(len(corpus))],
             columns=vectorizer.get_feature_names_out())
```

```
[[0 1 1 1 0 0 1 0 1]
 [0 2 0 1 0 1 1 0 1]
 [1 0 0 1 1 0 1 1 1]
 [0 1 1 1 0 0 1 0 1]]
```

	and	document	first	is	one	second	the	third	this
This is the first document.	0	1	1	1	0	0	1	0	1
This document is the second document.	0	2	0	1	0	1	1	0	1
And this is the third one.	1	0	0	1	1	0	1	1	1
Is this the first document?	0	1	1	1	0	0	1	0	1

Vector dimensions and Coordinate systems

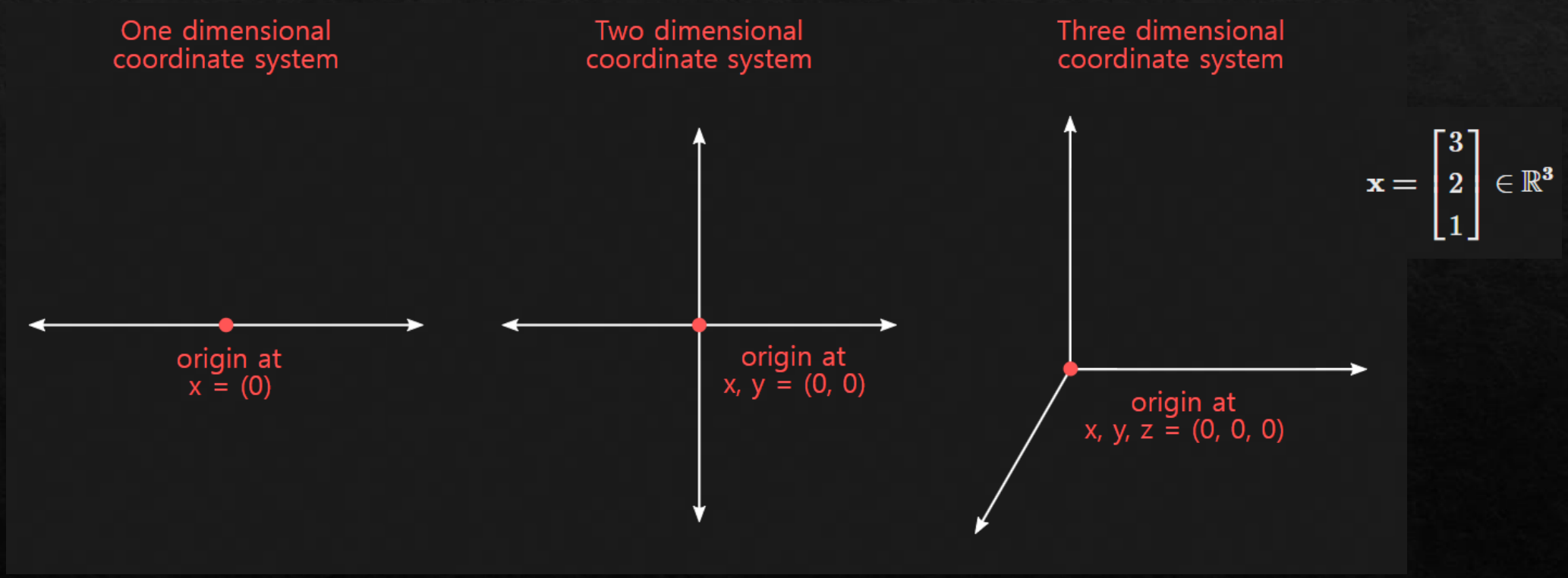
- Vector dimensions

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^n$$

$$\mathbf{x} = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix} \in \mathbb{R}^3$$

Vector dimensions and Coordinate systems

- Coordinate systems



Vector Operations

Vector – Vector Addition

- vector-vector addition

$$\mathbf{x}+\mathbf{y}=\begin{bmatrix}x_1\\ \vdots\\ x_n\end{bmatrix}+\begin{bmatrix}y_1\\ \vdots\\ y_n\end{bmatrix}=\begin{bmatrix}x_1+y_1\\ \vdots\\ x_n+y_n\end{bmatrix}$$

$$\mathbf{x}+\mathbf{y}=\begin{bmatrix}1\\ 2\\ 3\end{bmatrix}+\begin{bmatrix}1\\ 2\\ 3\end{bmatrix}=\begin{bmatrix}1+1\\ 2+2\\ 3+3\end{bmatrix}=\begin{bmatrix}2\\ 4\\ 6\end{bmatrix}$$

```
a = np.arange(10)
b = np.ones(10)*3
print(a)
print(b)

# 벡터 더하기
a + b
a - b
a*b
a/b
```

Vector – scalar multiplication

- vector-scalar multiplication is an element-wise operation
  - consider alpha = 2

$$\alpha \mathbf{x} = \begin{bmatrix} \alpha x_1 \\ \vdots \\ \alpha x_n \end{bmatrix}$$

$$\alpha \mathbf{x} = \begin{bmatrix} 2 \times 1 \\ 2 \times 2 \\ 2 \times 3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix}$$

```
x = np.arange(1, 4)

# 상수 곱하기
alpha = 2
alpha*x
```

Linear combinations of vectors(1/4)

- There are only two legal operations with vectors in linear algebra: **addition** and **multiplication by numbers**. When we combine those, we get a **linear combination**
  - consider alpha = 2, beta = 3

$$\alpha\mathbf{x} + \beta\mathbf{y} = \alpha \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \beta \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \alpha x_1 + \alpha x_2 \\ \beta y_1 + \beta y_2 \end{bmatrix}$$

$$\alpha\mathbf{x} + \beta\mathbf{y} = \mathbf{2} \begin{bmatrix} 2 \\ 3 \end{bmatrix} + \mathbf{3} \begin{bmatrix} 4 \\ 5 \end{bmatrix} = \begin{bmatrix} 2 \times 2 + 2 \times 4 \\ 2 \times 3 + 3 \times 5 \end{bmatrix} = \begin{bmatrix} 10 \\ 21 \end{bmatrix}$$

```
x = np.arange(2, 4)
y = np.arange(4, 6)

# vectors 선형결합(linear combination)
alpha = 2; beta = 3
alpha*x + beta*y
```

Linear combinations of vectors(2/4)

- Linear combinations are the **most fundamental** operation in linear algebra.
- Everything in linear algebra results from linear combinations.
  - For instance, **linear regression is a linear combination of vectors.**

Another way to express linear combinations you'll see often is with summation notation.

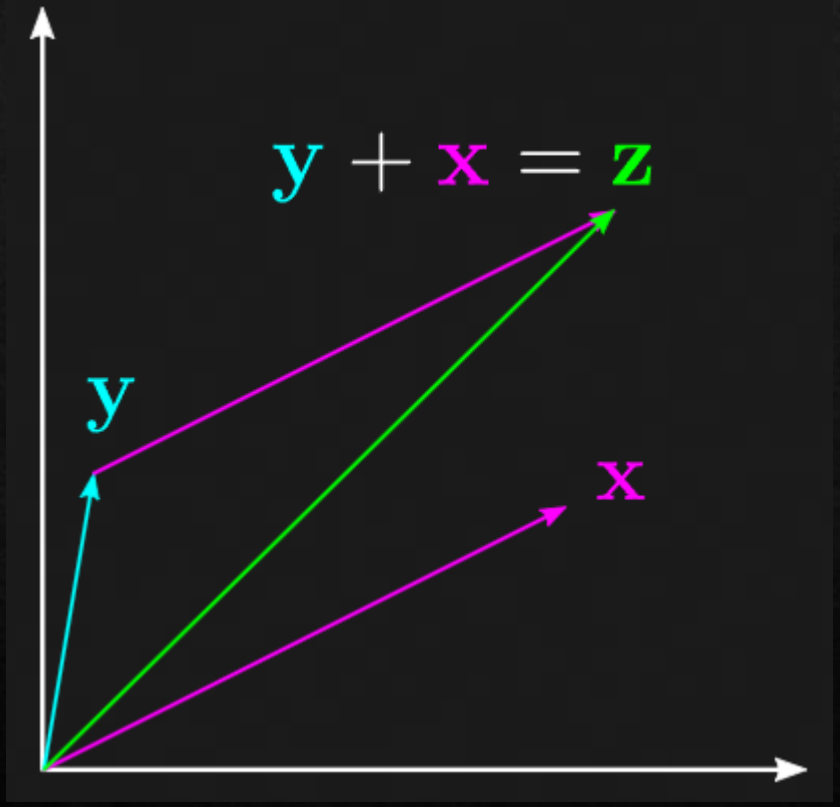
Consider a set of vectors $x_1,\ldots,x_k$ and scalars $\beta_1,\ldots,\beta_k \in \mathbb{R}$, then:

$$\sum_{i=1}^{k} \beta_i x_i := \beta_1 x_1 + \ldots + \beta_k x_k$$

Note that $:=$ means "*is defined as*".

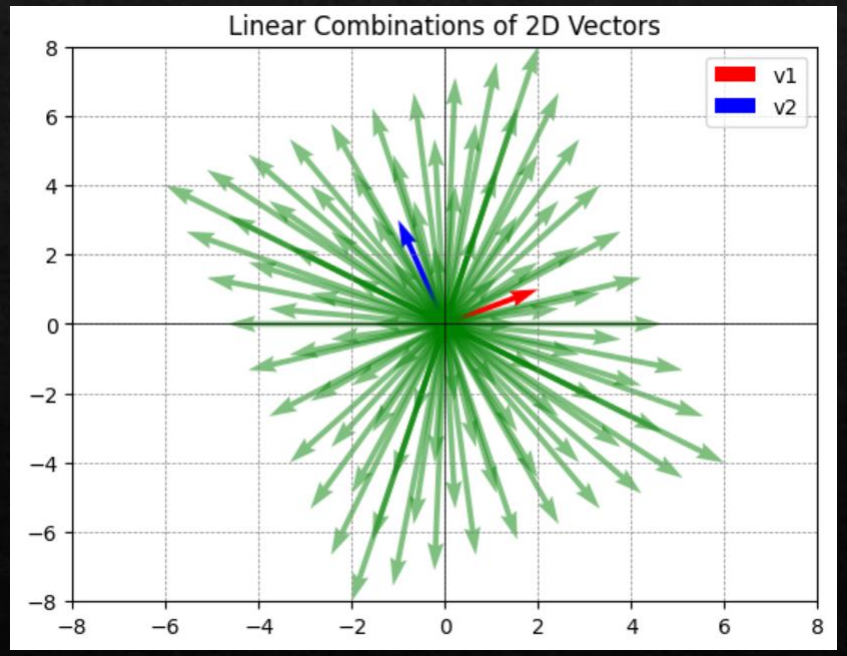
Linear combinations of vectors(3/4)

- Linear combinations in two dimensions



Linear combinations of vectors(4/4)

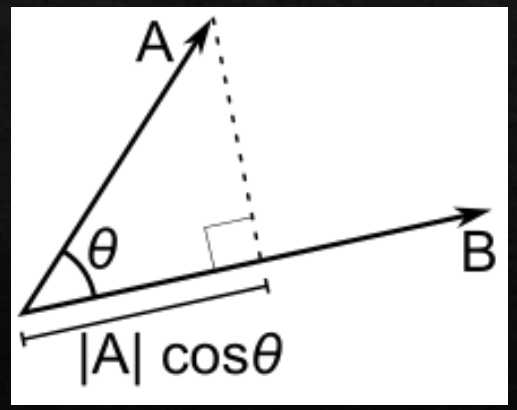
```
# Define two 2D vectors
x = np.array([2, 1]); y = np.array([-1, 3])

# Define scalar coefficients for linear combinations
coefficients = np.linspace(-2, 2, 10)

# Calculate linear combinations
combinations = np.array([c*x +  d*y for c in coefficients for d in coefficients])

# Plot the vectors and linear combinations
plt.quiver(0, 0, x[0], x[1], angles='xy', scale_units='xy', scale=1, color='r', label='v1')
plt.quiver(0, 0, y[0], y[1], angles='xy', scale_units='xy', scale=1, color='b', label='v2')

for i in range(len(combinations)):
    plt.quiver(0, 0, combinations[i, 0], combinations[i, 1],
               angles='xy', scale_units='xy', scale=1, color='g',
               alpha=0.5)

plt.xlim(-8, 8); plt.ylim(-8, 8)
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.grid(color='gray', linestyle='--', linewidth=0.5)
plt.legend()
plt.title('Linear Combinations of 2D Vectors')
```



Vector-vector multiplication: dot product

- dot product or inner product

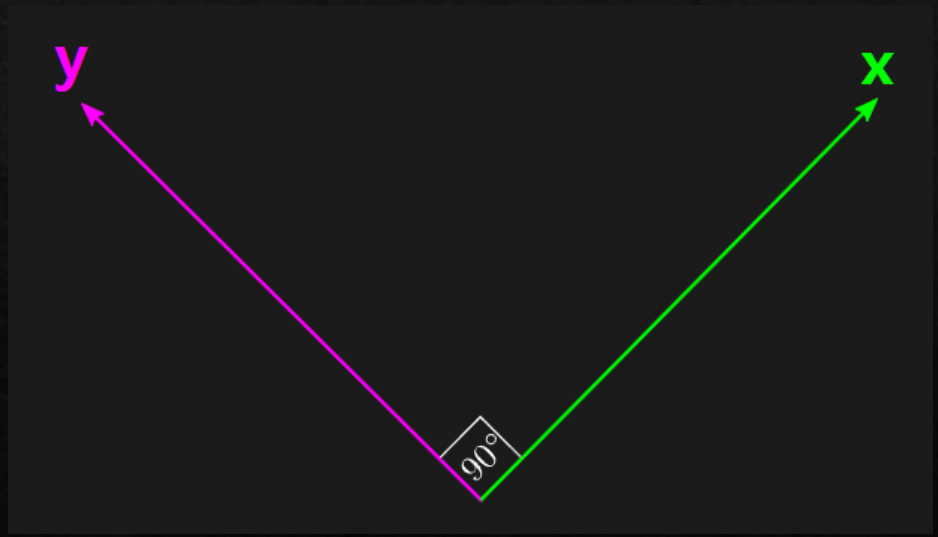
$$\mathbf{x} \cdot \mathbf{y} := \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^{\mathbf{T}} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \mathbf{x_1} \times \mathbf{y_1} + \mathbf{x_2} \times \mathbf{y_2}$$




```
# 두 벡터의 내적의 기하학적 의미
A = np.array([3,4])
B = np.array([4,1])

# Calculate the dot product
dot_product = np.dot(A, B) # A@B

# Calculate the magnitudes of the vectors
mag_A = np.linalg.norm(A)
mag_B = np.linalg.norm(B)

# Calculate the cosine of the angle
cosine_angle = dot_product / (mag_A * mag_B)

# Display the result
dot_product == mag_A*cosine_angle*mag_B
```

Orthogonal vectors

- Orthogonality is often used interchangeably with “**independence**”
  - We say that a pair of vectors x and y are orthogonal if their inner product is zero, **⟨x,y⟩=0**
  - The notation for a pair of orthogonal vectors is **x⊥y**
  - In the 2-dimensional plane, this equals to a pair of vectors forming a 90◦ **angle**.




```
x = np.array([[2], [0]])
y = np.array([[0], [2]])

x.T@y
```

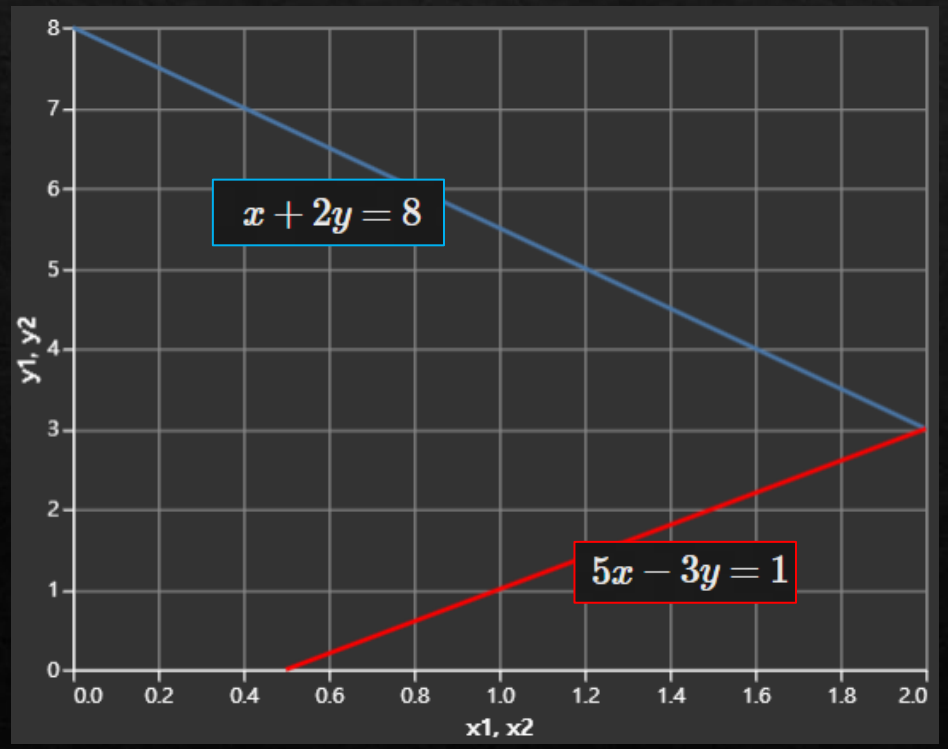
Systems of linear equations

- The purpose of linear algebra as a tool is to solve systems of linear equations.
  - Informally, this means to figure out the right combination of linear segments to obtain an outcome.
  - Even more informally, think about making pancakes: In what proportion $(w_i \in \mathbb{R})$ we have to mix ingredients to make pancakes? You can express this as a linear equation:

$$f_{\text{flour}} \times w_1 + b_{\text{baking powder}} \times w_2 + e_{\text{eggs}} \times w_3 + m_{\text{milk}} \times w_4 = P_{\text{pancakes}}$$

Systems of linear equations

$$x + 2y = 8$$
$$5x - 3y = 1$$




```
import numpy as np

A = np.array([[1, 2],
              [5, -3]])

B = np.array([8, 1])

# Solve the binary linear equations
solution = np.linalg.solve(A, B)

print("Solution to the binary linear equations:")
print(solution)
```

Learning Objectives

- Preliminary concepts
- Vectors
- Matrices
- Linear Mapping
- Norm (L1, L2)
- Numpy Applications

What is a Matrix?

- With matrices, we can represent sets of variables.
  - In this sense, a matrix is simply an **ordered collection of vectors**.
  - Conventionally, column vectors, but it's always wise to pay attention to the authors' notation when reading matrices.
  - we represent a matrix with a italicized upper-case letter like A. In two dimensions, we say the matrix A has m rows and n columns.

Each entry of A is defined as a_{ij}, $i = 1, \ldots, m$, and $j = 1, \ldots, n$. A matrix $A \in \mathbb{R}^{m \times n}$ is defines as:

$$A := \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, a_{ij} \in \mathbb{R}$$

Matrix Operations

Matrix – Matrix Addition

- matrix – matrix addition

We add matrices in a element-wise fashion. The sum of $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{m \times n}$

$$A + B := \begin{bmatrix} a_{11} + b_{11} & \cdots & a_{1n} + b_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} + b_{m1} & \cdots & a_{mn} + b_{mn} \end{bmatrix} \in \mathbb{R}^{m \times n}$$

For instance: $A = \begin{bmatrix} 0 & 2 \\ 1 & 4 \end{bmatrix} + B = \begin{bmatrix} 3 & 1 \\ -3 & 2 \end{bmatrix} = \begin{bmatrix} 0+3 & 2+1 \\ 3+(-3) & 2+2 \end{bmatrix} = \begin{bmatrix} 3 & 3 \\ -2 & 6 \end{bmatrix}$

python code

a = np.arange(10).reshape(2.5)

b = np.ones(10).reshape(2,5)

matrix 더하기

a + b

Matrix-scalar multiplication

- matrix - scalar multiplication

Matrix-scalar multiplication is an element-wise operation. Each element of the matrix A is multiplied by the scalar α. Is defined as:

$$a_{ij} \times \alpha, \text{such that } (\alpha A)_{ij} = \alpha(A)_{ij}$$

$$\alpha A = 2\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 \times 1 & 2 \times 2 \\ 2 \times 3 & 2 \times 4 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

```
# python code
# scalar multiplication
b = np.ones(10).reshape(2,5)
3*b
```

Matrix-vector multiplication

- matrix – vector multiplication

Matrix-vector multiplication equals to taking the dot product of each column n of a A with each element $\mathbf{x}$ resulting in a vector $\mathbf{y}$. Is defined as:

$$A \cdot \mathbf{x} := \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \mathbf{x_1} \begin{bmatrix} a_{11} \\ \vdots \\ a_{m1} \end{bmatrix} + \mathbf{x_2} \begin{bmatrix} a_{12} \\ \vdots \\ a_{m2} \end{bmatrix} + \mathbf{x_n} \begin{bmatrix} a_{1n} \\ \vdots \\ a_{mn} \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_{mn} \end{bmatrix}$$

For instance:

$$A \cdot \mathbf{x} = \begin{bmatrix} 0 & 2 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \mathbf{1} \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \mathbf{2} \begin{bmatrix} 2 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \times 0 + 2 \times 2 \\ 1 \times 1 + 2 \times 4 \end{bmatrix} = \begin{bmatrix} 4 \\ 9 \end{bmatrix}$$

Python code

```
A = np.array([[0,2],
              [1,4]])
x = np.array([[1],
              [2]])

A.shape, x.shape
# ((2, 2), (2, 1))

A@x

x@A

x.T@A

x.T,shape, A.shape
((1, 2), (2, 2))
```

Matrix – matrix multiplication

$$A \in \mathbb{R}^{m \times n} \cdot B \in \mathbb{R}^{n \times p} = C \in \mathbb{R}^{m \times p}:$$

$$A \cdot B := \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{np} \end{bmatrix} = \begin{bmatrix} c_{11} & \cdots & c_{1p} \\ \vdots & \ddots & \vdots \\ c_{m1} & \cdots & c_{mp} \end{bmatrix}$$

$$c_{ij} := \sum_{l=1}^{n} a_{il} b_{lj}, \text{ with } i = 1, \dots m, \text{ and}, j = 1, \dots, p$$

$$A \cdot B = \begin{bmatrix} 0 & 2 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} 1 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 \times 0 + 2 \times 2 & 3 \times 0 + 1 \times 2 \\ 1 \times 1 + 2 \times 4 & 3 \times 1 + 1 \times 4 \end{bmatrix} = \begin{bmatrix} 4 & 2 \\ 9 & 7 \end{bmatrix}$$

```
# python code

A = np.array([[0,2], [1,4]])
B = np.array([[1,3], [2,1]])
C = A@B
C
np.dot(A, B)
```

Identity Matrix

- An identity matrix is a square matrix with ones on the diagonal from the upper left to the bottom right, and zeros everywhere else

$$I_n := \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix} \in \mathbb{R}^{n \times n}$$

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
# python code
np.eye(10)


np.eye(3)
```

Inverse Matrix

- In the context of real numbers, the multiplicative inverse (or reciprocal) of a number x is the number that when multiplied by x yields 1
  - We denote this by x^{-1} or $\frac{1}{x}$
  - Take the number 5. Its multiplicative inverse equals to 5×1/5=1.

Consider a system of linear equations as:

$$A\mathbf{x} = \mathbf{y}$$

Assuming A has an inverse, we can multiply by the inverse on both sides:

$$A^{-1}A\mathbf{x} = A^{-1}\mathbf{y}$$

And get:

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

Since the I does not affect $\mathbf{x}$ at all, our final expression becomes:

$$\mathbf{x} = A^{-1}\mathbf{y}$$

Python code

```
A = np.array([[1, 2, 1],
              [4, 4, 5],
              [6, 7, 7]])

A_i = np.linalg.inv(A)
print(f'A inverse:₩n{A_i}')

# We can check the A inverse  is correct by multiplying. If so, we should obtain the identity I3

I = np.round(A_i @ A)
print(f'A_i times A resulsts in I_3:₩n{I}')
```

Transpose Matrix

Consider a matrix $A \in \mathbb{R}^{m \times n}$. The **transpose** of A is denoted as $A^T \in \mathbb{R}^{m \times n}$. We obtain A^T as:

$$(A^T)_{ij} = A_j i$$

In other words, we get the A^T by switching the columns by the rows of A. For instance:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

```
# python code
A = np.arange(1, 7).reshape(3,2)
A
```

```
A.T
```

Special Matrix

Rectangular matrix

Matrices are said to be *rectangular* when the number of rows is $\neq$ to the number of columns, i.e., $A^{m \times n}$ with $m \neq n$. For instance:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Square matrix

Matrices are said to be **square** when the number of rows $=$ the number of columns, i.e., $A^{n \times n}$. For instance:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Diagonal matrix

Square matrices are said to be **diagonal** when each of its non-diagonal elements is zero, i.e., For $D = (d_{i,j})$, we have $\forall i, j \in n, i \neq j \implies d_{i,j} = 0$. For instance:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \end{bmatrix}$$

Learning Objectives

- Preliminary concepts
- Vectors
- Matrices
- Linear Mapping
- Norm (L1, L2)
- Numpy Applications

What is Linear Mapping?

- Now we have covered the basics of vectors and matrices, we are ready to introduce the idea of a linear mapping.

- Linear mappings, also known as **linear transformations** and **linear functions**, indicate the correspondence between vectors in a vector space V and the same vectors in a different vector space W.

What is Linear Mapping?

- This is an abstract idea. I like to think about this in the following manner: imagine there is a multiverse as in Marvel comics, but instead of humans, aliens, gods, stars, galaxies, and superheroes, we have vectors.
  - In this context, a linear mapping would indicate the correspondence of entities (i.e., planets, humans, superheroes, etc) between universes.

- Just imagine us, placidly existing in our own universe, and suddenly a linear mapping happens: **our entire universe would be transformed into a different one, according to whatever rules the linear mapping has enforced**. Now, switch universes for vector spaces and us by vectors, and you'll get the full picture.

Linear transformations

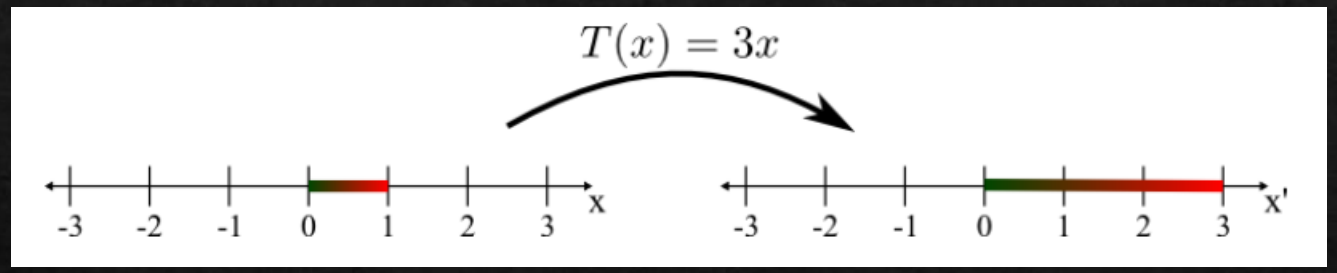
- In linear algebra, linear mappings are represented as matrices and performed by matrix multiplication. Take a vector x and a matrix A.
  - We say that when A multiplies x, the matrix transform the vector into another one:

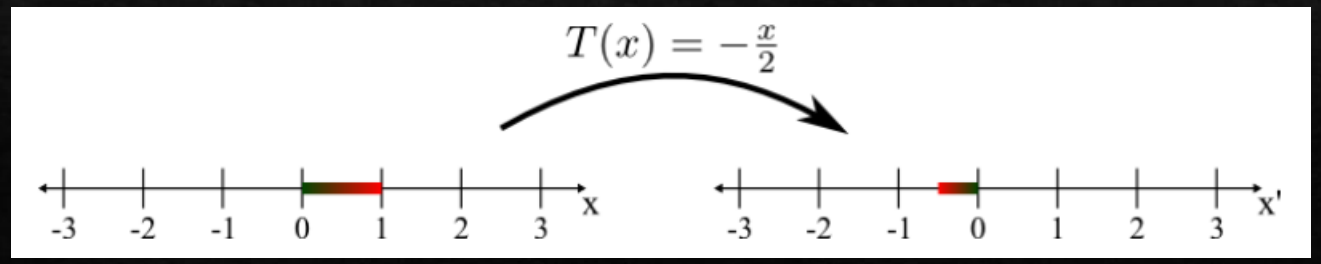
$$T(\mathbf{x}) = A\mathbf{x}$$

  - The typicall notation for a linear mapping is the same we used for functions. For the vector spaces V and W, we indicate the linear mapping as

$$T: V \to W$$

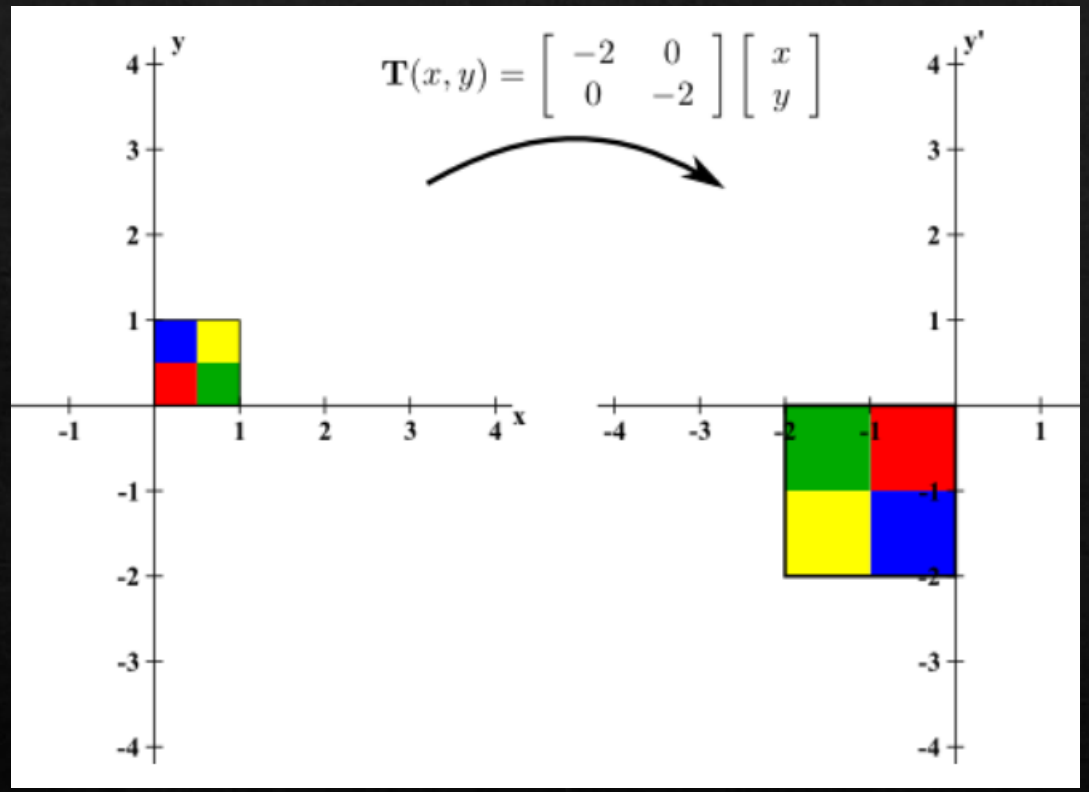
One-dimensional linear transformations





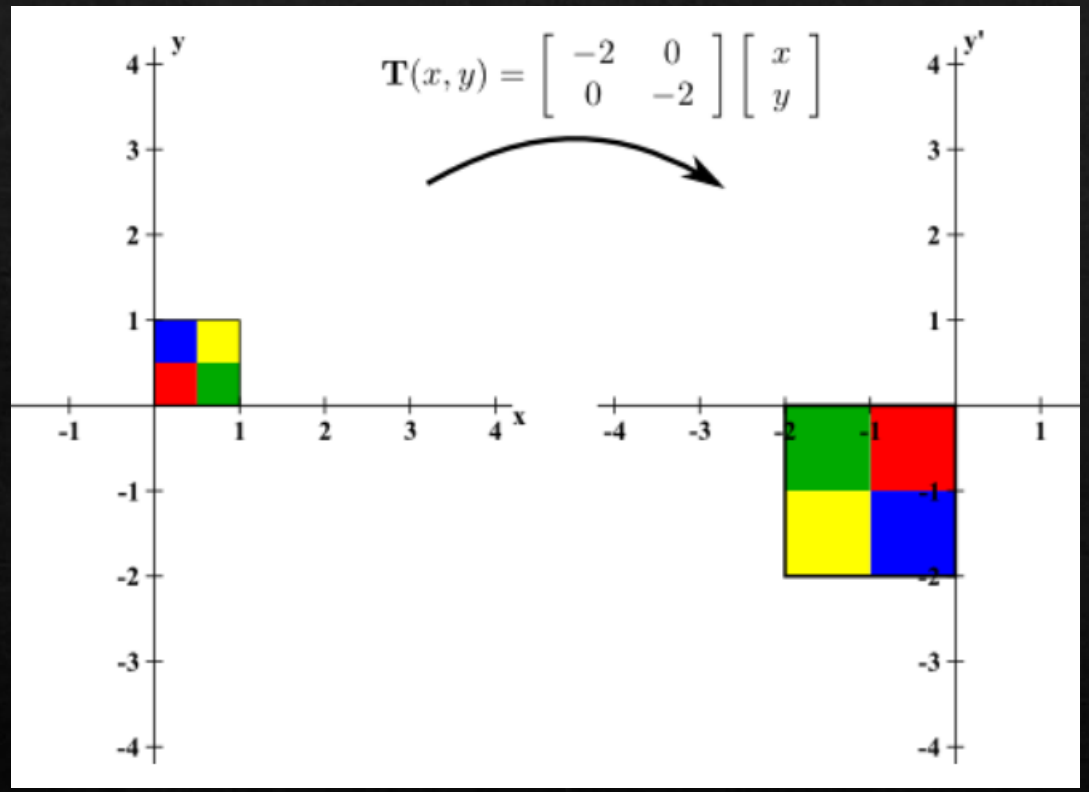

```
# python code
T = 3
x = np.array(3)
T*x

T = -0.5
T*x
```

Two-dimensional linear transformations




```
# python code
T = np.array([[-2,0], [0,-2])
T = np.array([[-2,0], [0,-2]])
x = np.array([1,1])
T.shape, x.shape
((2, 2), (2,))

# Linear Transformation
T@x
```

Two-dimensional linear transformations

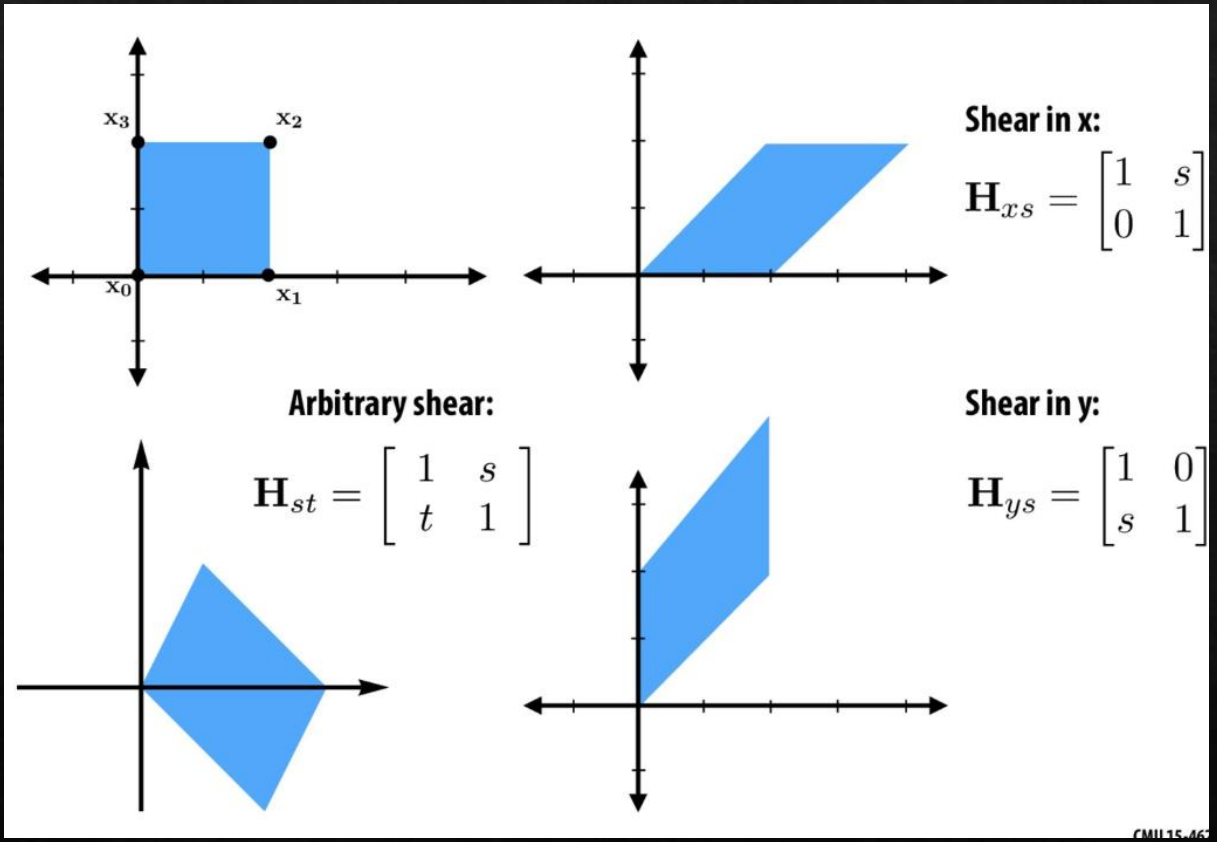



```
# python code
T = np.array([[-2,0], [0,-2])
T = np.array([[-2,0], [0,-2]])
x = np.array([1,1])
T.shape, x.shape
```

((2, 2), (2,))

```
# Linear Transformation
T@x
```

Shear linear transformations



http://15462.courses.cs.cmu.edu/fall2019/lecture/transformations/slide_028

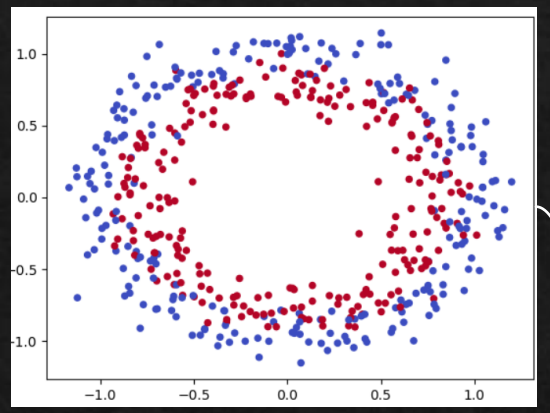
```
# python code
# Shear in x
sheer_matrix = np.array([[1,10], [0,1]])
x = np.array([1,1])
sheer_matrix@x

# Shear in y
sheer_matrix = np.array([[1,0], [10,1]])
sheer_matrix@x

# Arbitrary shear
sheer_matrix = np.array([[1,5], [10,1]])
sheer_matrix@x
```

Digress : Shear Bradpit Image

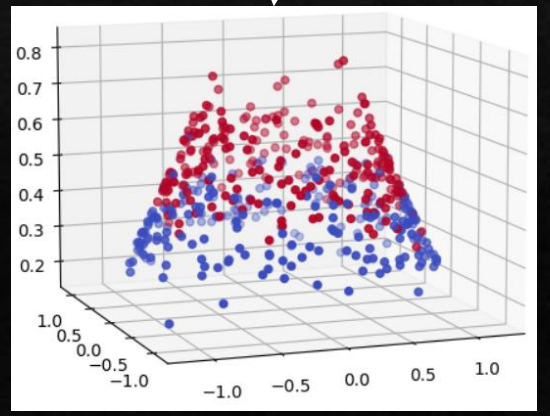
```
# python code
from PIL import Image, ImageOps, ImageFilter

# Load the image (Brad Pit)
image_url = "https://people.com/thmb/h67yba4NaGbpnv9DZVZm894kSjo=/1500x0/filters:no_upscale():max_bytes(150000):strip_icc():focal(979x292:981x294)/Brad-Pitt_1-2000-e8a294b80b034e659785ad813d3f02f2.jpg"
response = requests.get(image_url)
image = Image.open(BytesIO(response.content))

# Define the shear factor (adjust as needed)
shear_factor = 1.2

# Create a shear matrix for horizontal shear
shear_matrix = (1, shear_factor, 0, 0, 1, 0)

# Apply the shear transformation to the image
sheared_image = image.transform(image.size, Image.AFFINE, shear_matrix, Image.BICUBIC)
```

Digress : Kernel Transformation

```
# python code
from sklearn.datasets import make_circles
X, y = make_circles(n_samples=500, random_state=11, noise=0.1)
plt.scatter(X[:, 0], X[:, 1], c=y, s=20, cmap='coolwarm');

r =np.exp(-(X ** 2).sum(1))

from mpl_toolkits import mplot3d
plt.figure(figsize=(8,6))
ax = plt.axes(projection='3d')
ax.scatter3D(X[:, 0], X[:, 1], r, c=y, cmap='coolwarm')
#ax.view_init(10, 250)
```



$T : \mathbb{R}^2 \rightarrow \mathbb{R}^3$



Affine Transformation

- Now The simplest way to describe affine mappings (or transformations) is as a linear mapping + translation. Hence, an affine mapping M takes the form of:

$$M(\mathbf{x}) = A\mathbf{x} + \mathbf{b}$$

  - Where A is a linear mapping or transformation and b is the translation vector.

- If you are familiar with linear regression, you would notice that the above expression is its **matrix form. Linear regression is usually analyzed as a linear mapping plus noise, but it can also be seen as an affine mapping.**

  - Alternative, we can say that $A\mathbf{x} + \mathbf{b}$ is a linear mapping *if and only if* $\mathbf{b} = 0$.

Affine Transformation

$$\mathbf{y} = \mathbf{X}\beta + \epsilon$$

$$M(\mathbf{x}) = A\mathbf{x} + \mathbf{b}$$

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \qquad \mathbf{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1k} \\ 1 & x_{21} & x_{22} & \cdots & x_{2k} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{nk} \end{bmatrix}$$

$$\beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_k \end{bmatrix}, \qquad \epsilon = \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{bmatrix}$$

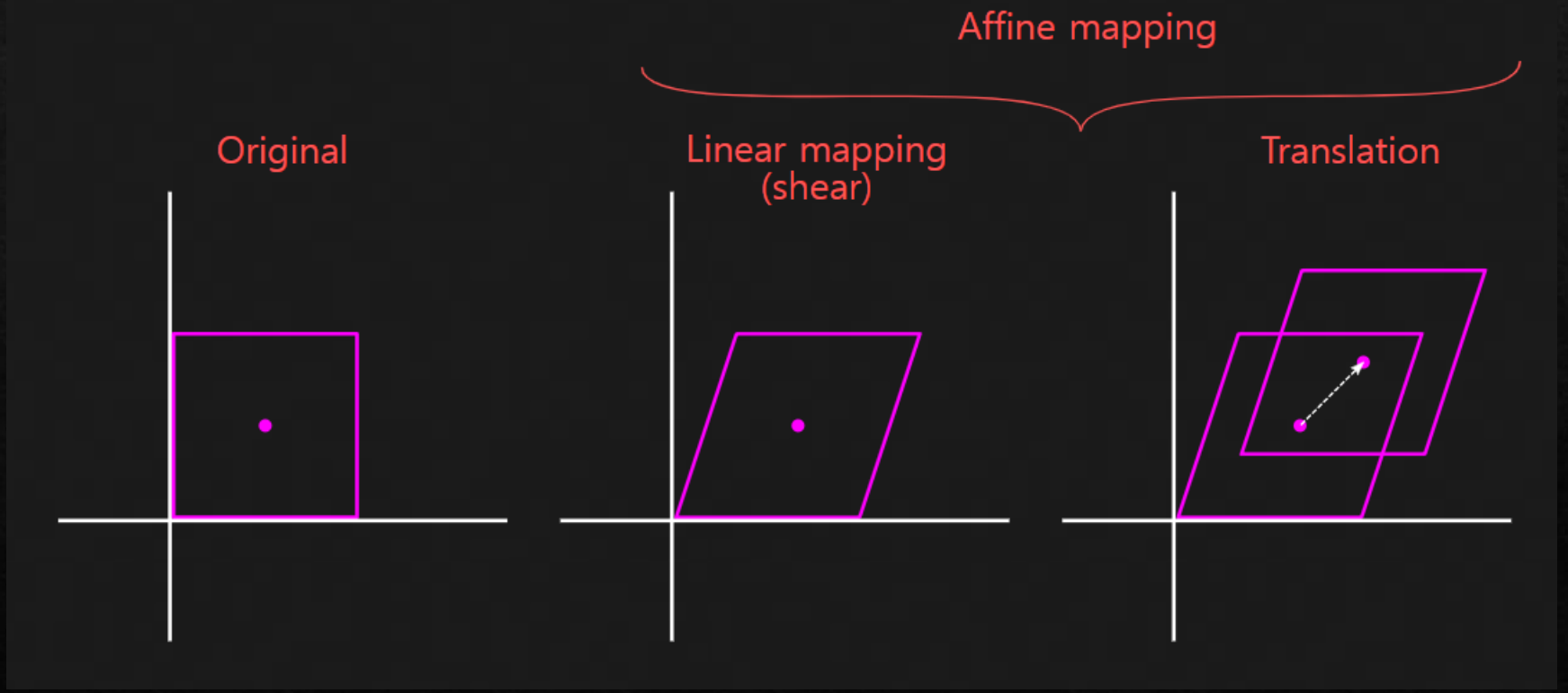
$$\sum_{i=1}^{n} \epsilon_i^2 = \epsilon'\epsilon = (\mathbf{y} - \mathbf{X}\beta)'(\mathbf{y} - \mathbf{X}\beta)$$

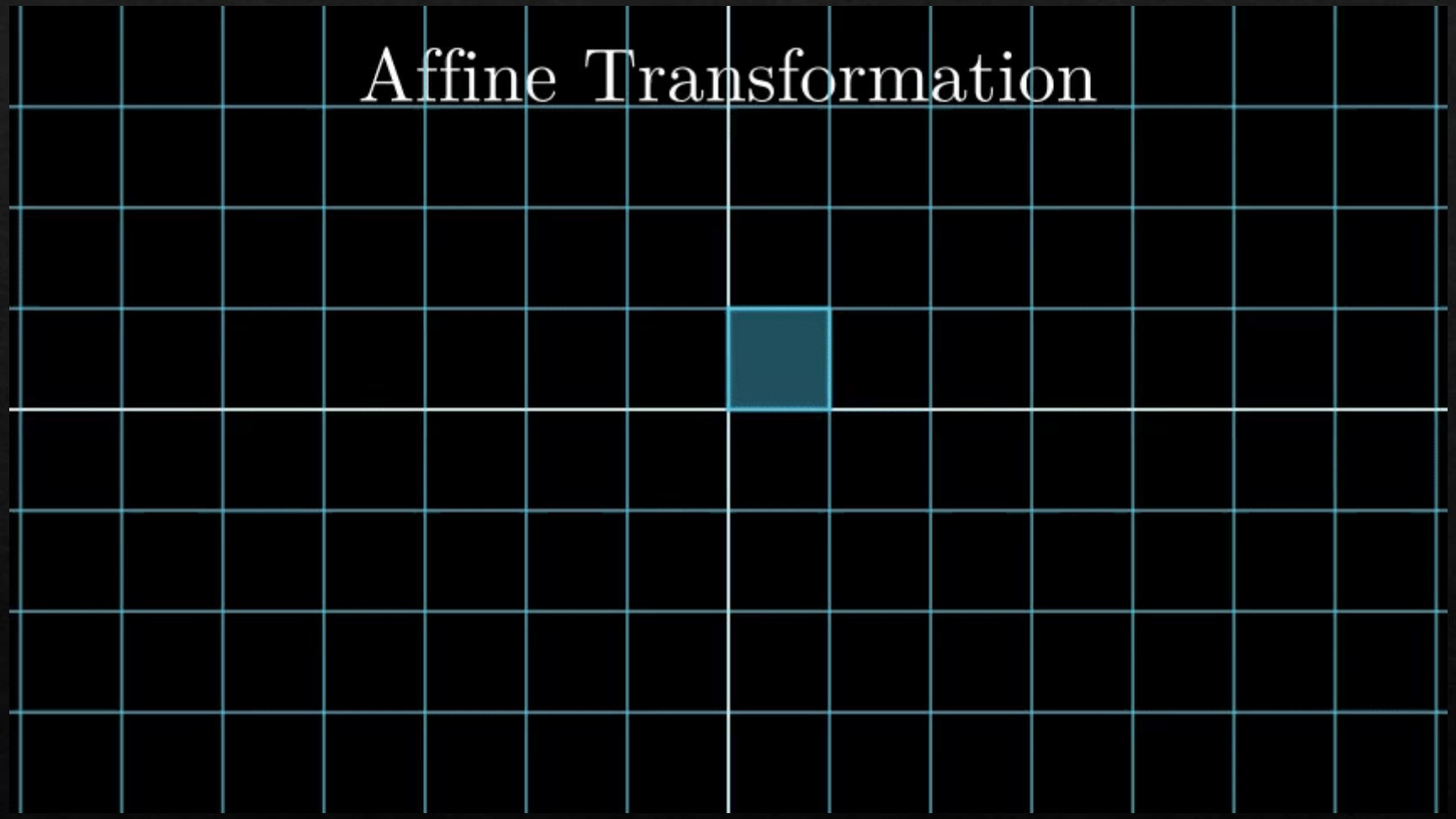
$$\hat{\mathbf{y}} = \mathbf{X}\hat{\beta}$$

$$A\mathbf{x} + \mathbf{b}$$

Affine Transformation

- From a geometrical perspective, affine mappings displace spaces (lines or hyperplanes) from the origin of the coordinate space.
  - Consequently, affine mappings do not operate over vector spaces as the zero vector condition $0 \in S$ does not hold anymore. Affine mappings act onto affine subspaces, that I'll define later in this section.




Affine Transformation

Affine Transformation

Learning Objectives

- Preliminary concepts
- Vectors
- Matrices
- Linear Mapping
- Norm (L1, L2)
- Numpy Applications

Norm

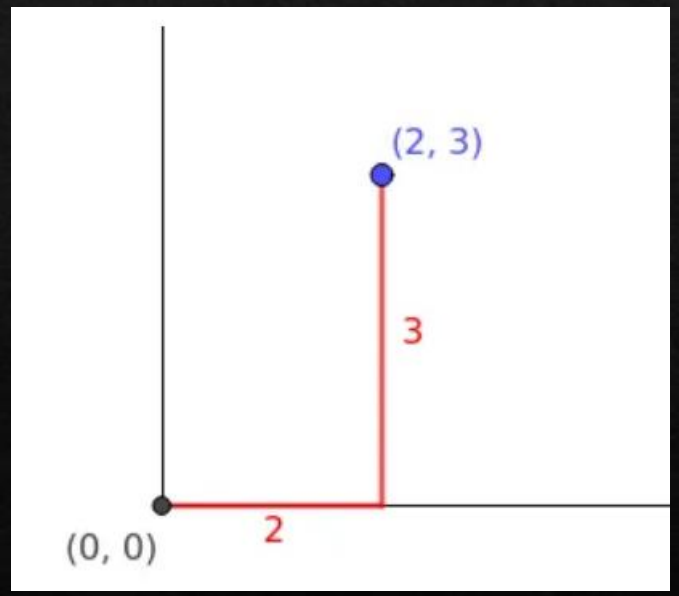
- The total length of all the vectors in a space (non-negative real numbers that behaves in certain ways like **the distance from the origin**)
  - The p-norm of vector x will be denoted as

$$||x||_p = (x_1^p + x_2^p + x_3^p + \ldots + x_n^p)^{1/p} = (\sum_{i=1}^{n} x_i^p)^{1/p}$$

  - $\|X\|_1$ (L1 norm)
    - $\vec{X} = [2,3]$, $\|X\|_1 = (2+3)^1 = 5$
  - $\|X\|_2$ (L2 norm)
    - $\vec{X} = [2,3]$, $\|X\|_2 = (2^2 + 3^2)^{\frac{1}{2}} = (4+9)^{\frac{1}{2}} = \sqrt[2]{13}$

1-Norm

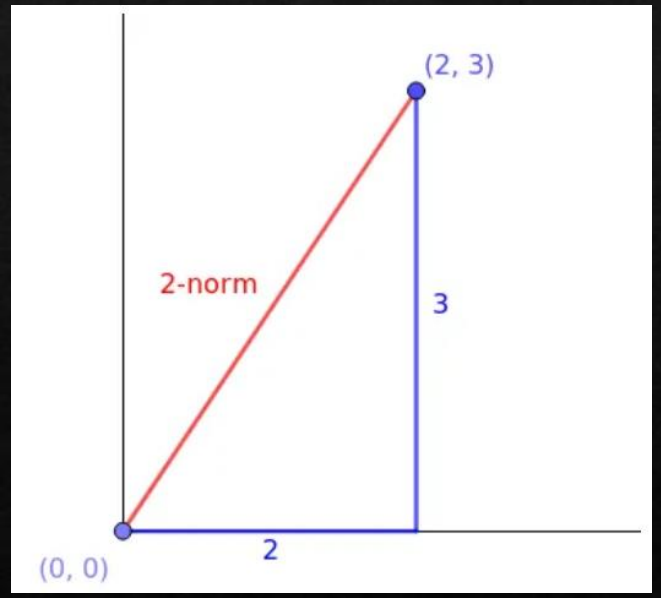
- L1 Norm (Mahattan Distance, Taxicab geometry)
  - $\|X\|_1$ (L1 norm)
    - $\vec{X} = [2,3]$, $\|X\|_1 = (2+3)^1 = 5$




```
# python code
import numpy as np

x = np.array([2, 3])
norm_l1 = np.linalg.norm(x, ord=1)  # L1 norm
norm_l1
# 5.0

x/norm_l1           # scaled by 1-norm
np.square(x/norm_l1)
np.sum(x/norm_l1)
# 1
```

2-Norm

- L2 Norm(Euclidean Distance)
  - $\|X\|_2$ (L2 norm)
    - $\vec{X} = [2,3]$, $\|X\|_2 = (2^2 + 3^2)^{\frac{1}{2}} = (4+9)^{\frac{1}{2}} = \sqrt[2]{13}$




```
# python code

norm_l2 = np.linalg.norm(x, ord=2)  # L2 norm
norm_l2
# 3.605551275463989

x/norm_l2                    # scaled by 2-norm
np.square(x/norm_l2)
np.square(x/norm_l2).sum()
# 1
```

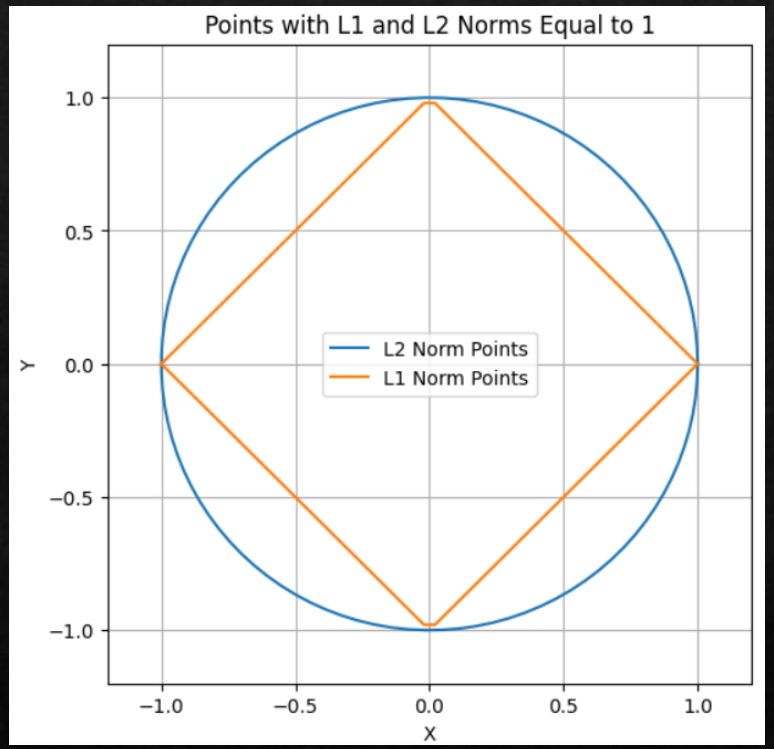
Lasso, Ridge Penalty in machine learning

Lasso

$$\|\boldsymbol{x}\|_1 := \sum_{i=1}^{n} |x_i|\,.$$

Ridge

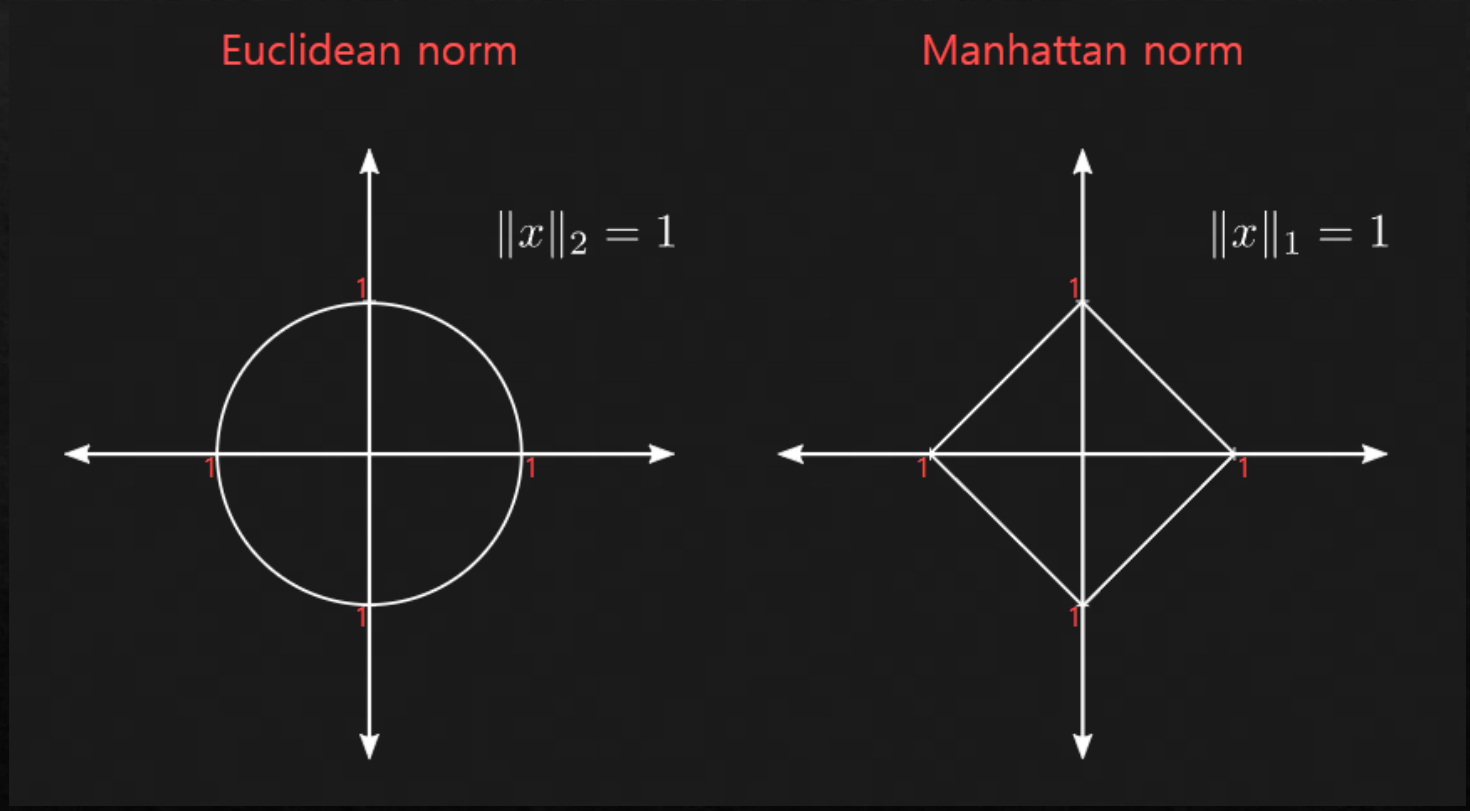
$$\|\boldsymbol{x}\|_2 := \sqrt{x_1^2 + \cdots + x_n^2}.$$




```
import numpy as np
import matplotlib.pyplot as plt

# Generate values for x
x = np.linspace(-1, 1, 400)

# Calculate corresponding values for y
y_pos = 1 - np.abs(x); y_neg = -1 + np.abs(x)

# Plot the curve where |x + y| = 1
plt.plot(x, y_pos, label='|x + y| = 1')
plt.plot(x, y_neg)

# Set axis labels
plt.xlabel('X'); plt.ylabel('Y')

# Add a legend
plt.legend()

plt.title('Curve where |x + y| = 1')
plt.grid(True)
plt.axhline(0, color='black',linewidth=0.5)
plt.axvline(0, color='black',linewidth=0.5)
plt.show()
```

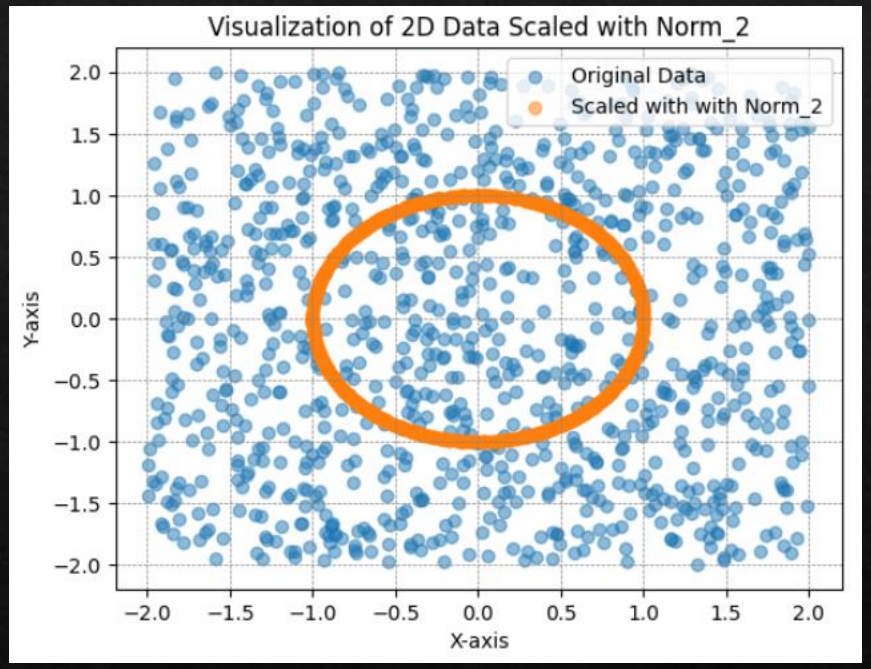
Vector norms

- norm or the length of a vector is the distance between its “origin” and its “end”.



Python code

```
import numpy as np
import matplotlib.pyplot as plt

def visualize_l1_norm(data, norm):
    # Calculate L1 norms for each data point
    l1_norms = np.linalg.norm(data, ord=norm, axis=1)

    # Scale data based on L1 norms
    scaled_data = data / l1_norms[:, np.newaxis]

    # Plotting
    plt.scatter(data[:, 0], data[:, 1], label='Original Data', alpha=0.5)
    plt.scatter(scaled_data[:, 0], scaled_data[:, 1],
                label=f'Scaled with with Norm_{norm}', alpha=0.5)

    plt.axhline(0); plt.axvline(0)
    plt.grid(color='gray', linestyle='--', linewidth=0.5)

    plt.title(f'Visualization of 2D Data Scaled with Norm_{norm}')
    plt.xlabel('X-axis'); plt.ylabel('Y-axis')
    plt.legend()
    plt.show()

# Generate random 2D data
random_data = np.random.rand(1000, 2) * 4 - 2  # Values between -2 and 2 for both dimensions

# Visualize the data scaled with L1 norm
visualize_l1_norm(random_data, norm = 2) # Eucledean Norm (L2 norm)
```



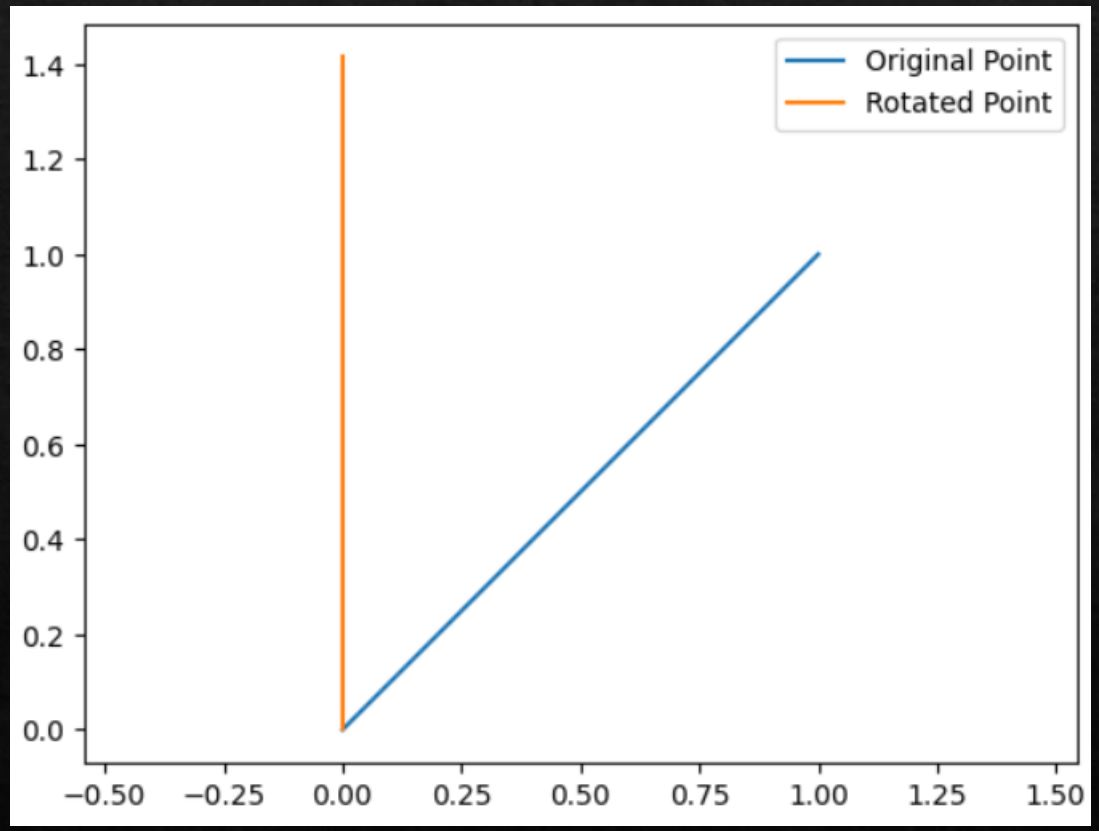
→ It's your turn to try L1 norm

Learning Objectives

- Preliminary concepts
- Vectors
- Matrices
- Linear Mapping
- Norm (L1, L2)
- Numpy Applications

Geometry(1/2)

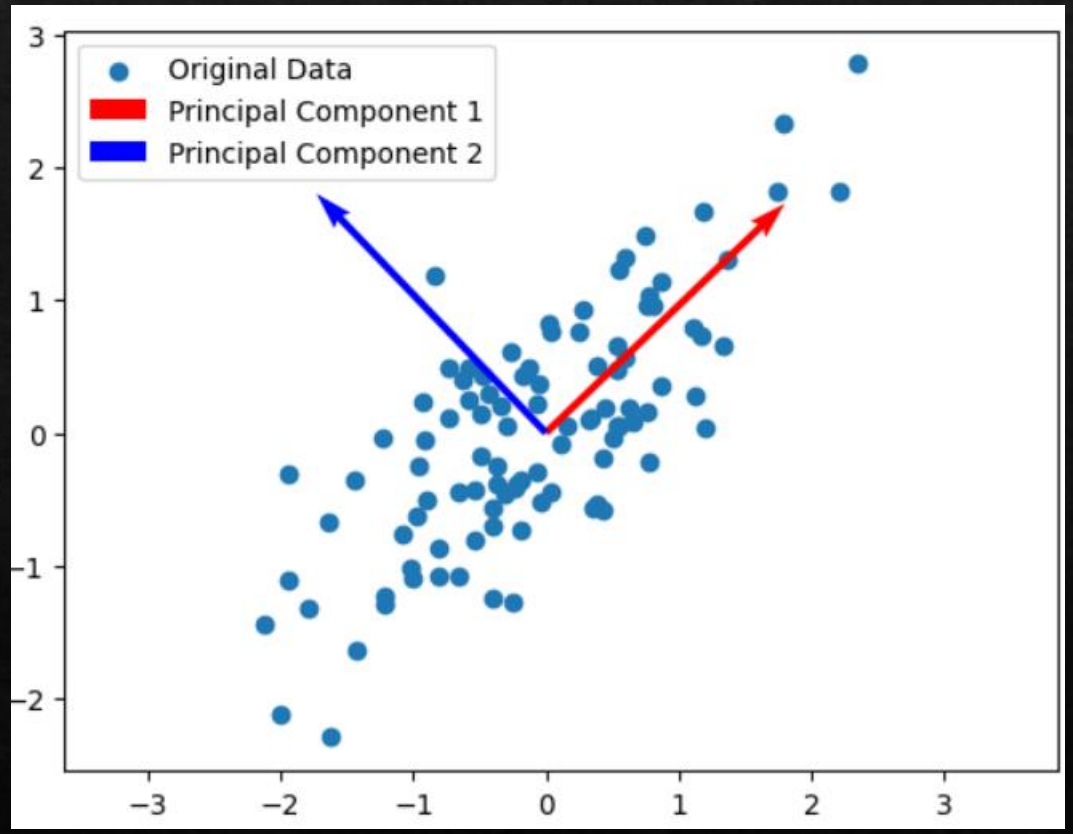
1. Transformation Matrices (2D Rotation)




```
import numpy as np
import matplotlib.pyplot as plt

# Define a 2D point
point = np.array([1, 1])

# Define a 2D rotation matrix
theta = np.pi / 4  # 45 degrees
rotation_matrix = np.array([[np.cos(theta), -
np.sin(theta)],
                            [np.sin(theta),
np.cos(theta)]])

# Apply rotation to the point
rotated_point = np.dot(rotation_matrix, point)

# Plot the original and rotated points
plt.plot([0, point[0]], [0, point[1]],
label='Original Point')
plt.plot([0, rotated_point[0]], [0,
rotated_point[1]], label='Rotated Point')
plt.legend()
plt.axis('equal')
plt.show()
```

Geometry(2/2)

2. Eigenvalues and Eigenvectors (Principal Component Analysis - PCA):

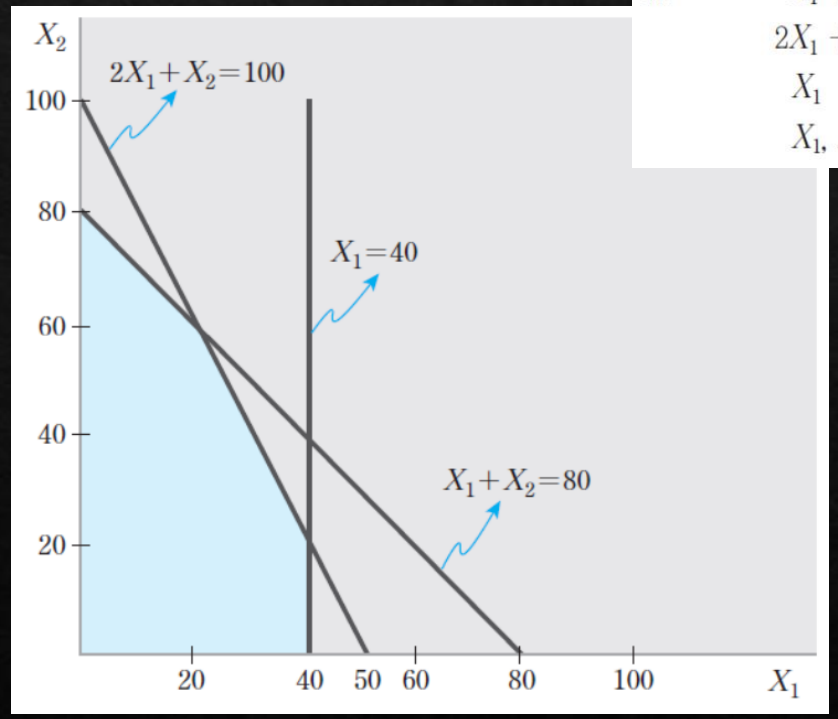



```
import numpy as np
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Generate 2D data
data = np.random.multivariate_normal(mean=[0,
0], cov=[[1, 0.8], [0.8, 1]], size=100)

# Apply PCA
pca = PCA()
pca.fit(data)

# Plot the original data and principal
components
plt.scatter(data[:, 0], data[:, 1], label='Original
Data')
plt.quiver(0, 0, pca.components_[0, 0],
pca.components_[0, 1], scale=3, color='r',
label='Principal Component 1')
plt.quiver(0, 0, pca.components_[1, 0],
pca.components_[1, 1], scale=3, color='b',
label='Principal Component 2')
plt.legend()
plt.axis('equal')
```

Optimization(1/2)

1. Linear Programing :

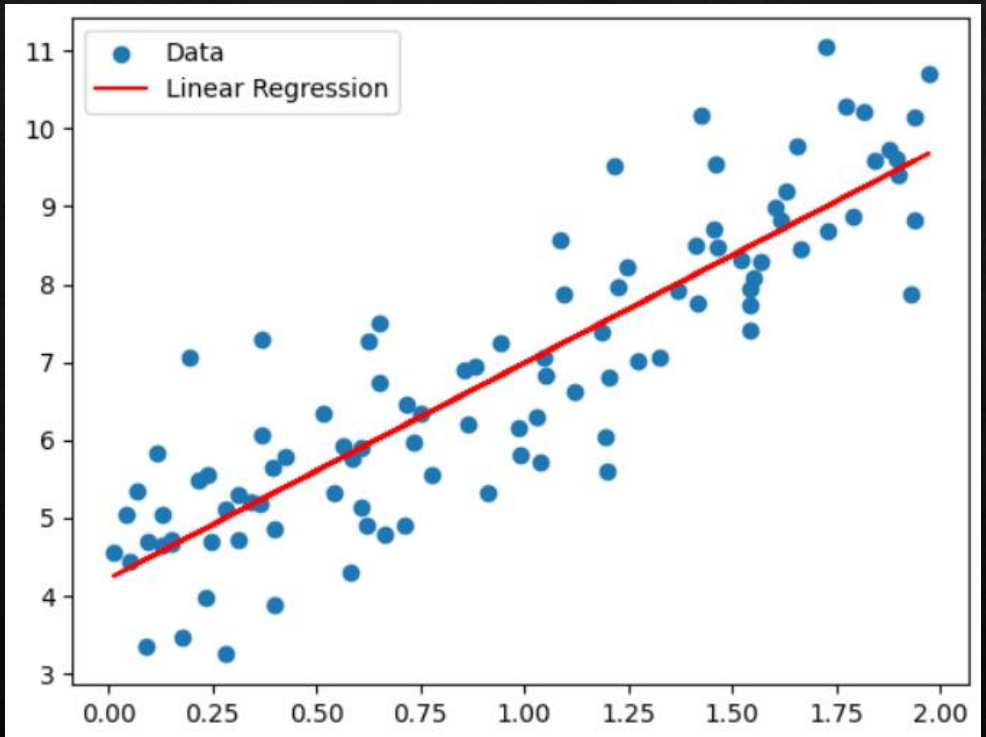
$$
\begin{aligned}
MAX\ Z &= 3X_1 + 2X_2 \\
s.t. \quad & X_1 + X_2 \le 80 \\
& 2X_1 + X_2 \le 100 \\
& X_1 \le 40 \\
& X_1, X_2 \ge 0
\end{aligned}
$$




```
from scipy.optimize import linprog

# Objective function coefficients
c = [-3, -2]  # Minimize -3x1 - 2x2 is equivalent to maximizing 3x1 + 2x2
# Inequality constraints matrix
A = [[1, 1], [2, 1], [-1, 0]]  # Note the change in the third constraint
# Inequality constraints vector
b = [100, 100, -40]

# Bounds for variables
x0_bounds = (0, None)
x1_bounds = (0, None)

# Solve linear programming problem
result = linprog(c, A_ub=A, b_ub=b, bounds=[x0_bounds, x1_bounds], method='highs')
# Display the results
# Negate the result because linprog minimizes by default
print('Optimal value (max z):', -result.fun)
print('Optimal point (x1, x2):', result.x)
```

Optimization(2/2)

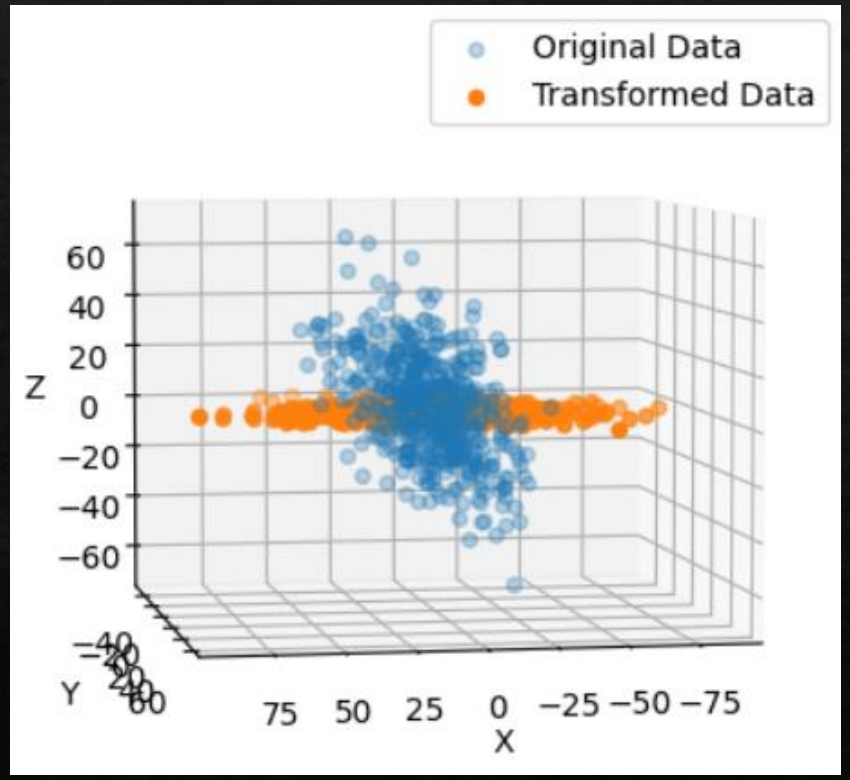
2. Least Squares Regression:




```
import numpy as np
import matplotlib.pyplot as plt

# Generate synthetic data
np.random.seed(42)
X = 2 * np.random.rand(100, 1)
y = 4 +  3 * X +  np.random.randn(100, 1)

# Perform linear regression
X_b = np.c_[np.ones((100, 1)), X]  # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)

# Plot the data and regression line
plt.scatter(X, y, label='Data')
plt.plot(X, X_b.dot(theta_best), color='red', label='Linear Regression')
plt.legend()
```

Machine Learning(1/2)

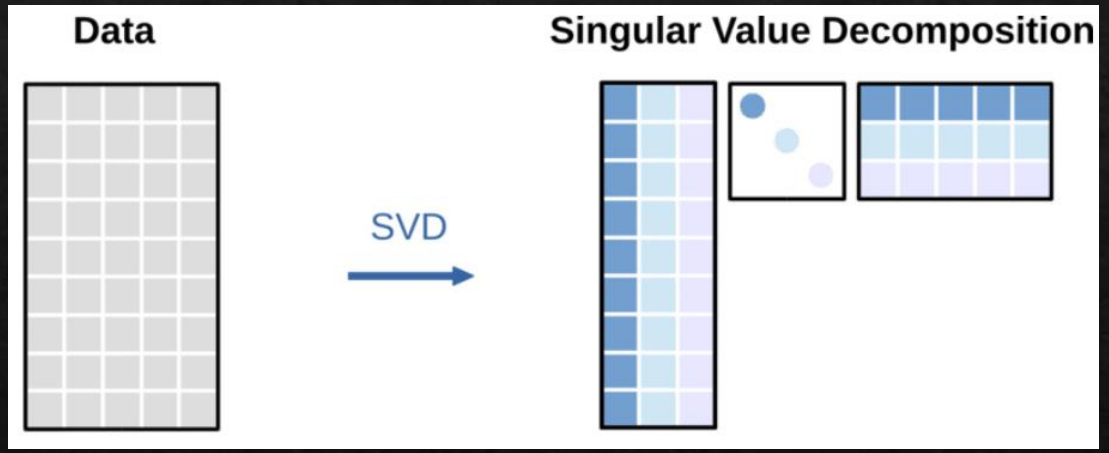
1. Principal Component Analysis(PCA):

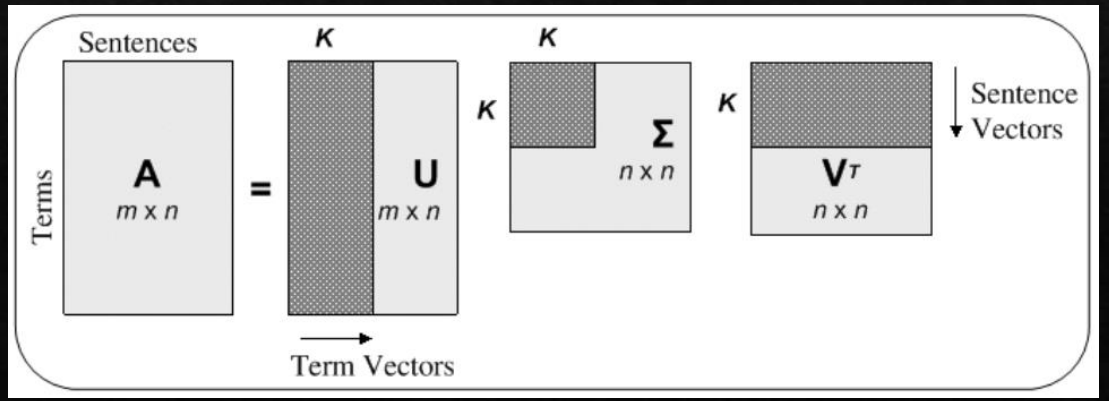



```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D  # Import necessary module for 3D projection
from sklearn.decomposition import PCA
# Generate 3D data
data = np.random.multivariate_normal(mean=[0, 0, 0],
                                     cov=[[1, 0.8, 0.5], [0.8, 1, 0.3], [0.5, 0.3, 1]],
                                     size=500)*20
# Apply PCA
pca = PCA(n_components=2)
transformed_data = pca.fit_transform(data)
# Plot the original and transformed data in 3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(data[:, 0], data[:, 1], data[:, 2], label='Original Data', alpha = 0.3)
ax.scatter(transformed_data[:, 0], transformed_data[:, 1],
np.zeros_like(transformed_data[:, 1]), label='Transformed Data',)
ax.set_xlabel('X'); ax.set_ylabel('Y'); ax.set_zlabel('Z')
ax.legend()

# Adjust the viewing angle
ax.view_init(elev=5, azim=80)  # Set the elevation and azimuth angles
plt.show()
```

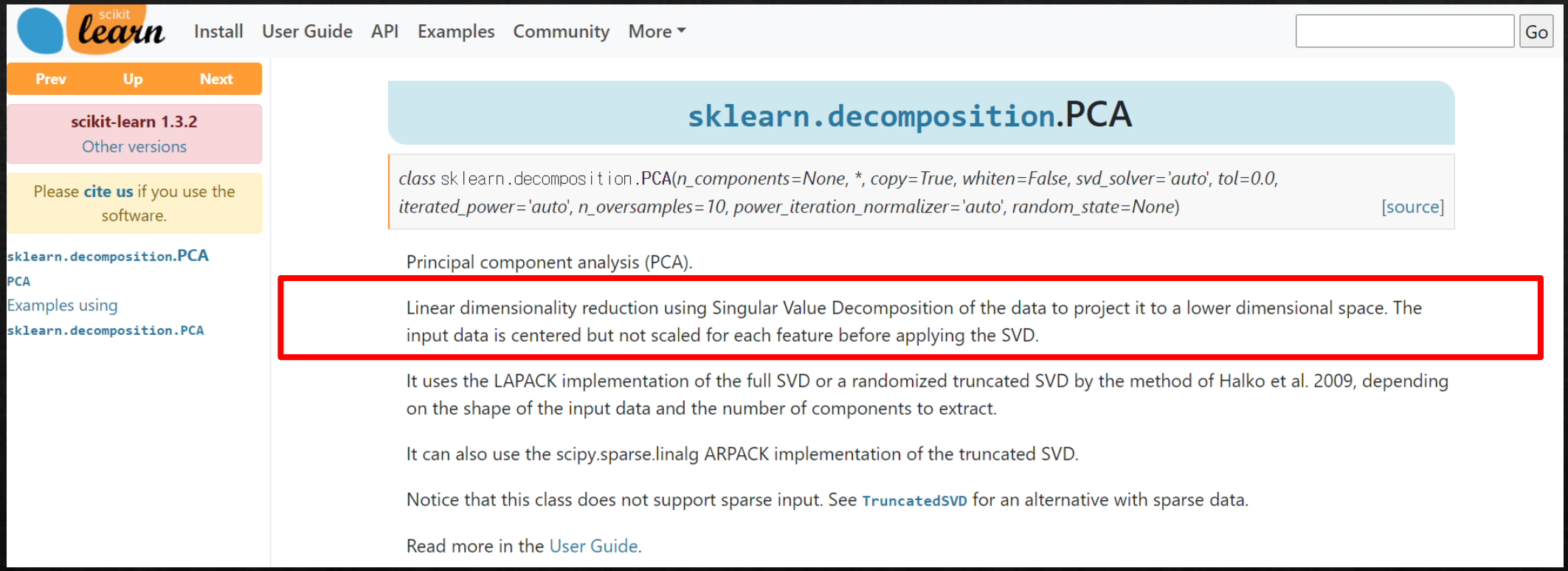
Machine Learning(2/2)

2. Singular Value Decomposition (SVD):



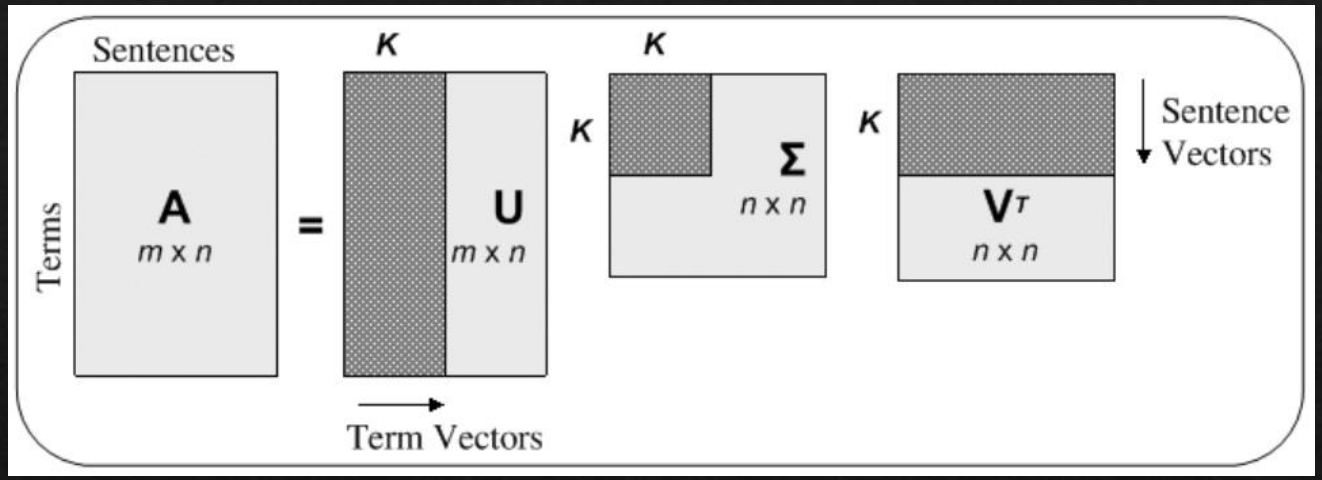


Machine Learning(2/2)



Machine Learning(2/2)

```
import numpy as np

# Generate a matrix
A = np.random.rand(3, 3)

# Perform SVD
U, S, Vt = np.linalg.svd(A, full_matrices=False)

# Reconstruct the original matrix
A_reconstructed = U @ np.diag(S) @ Vt

print("Original Matrix:")
print(A)
print("₩nReconstructed Matrix:")
print(A_reconstructed)
```




```
Original Matrix:
[[0.40895294 0.17329432 0.15643704]
 [0.2502429 0.54922666 0.71459592]
 [0.66019738 0.2799339 0.95486528]]

Reconstructed Matrix:
[[0.40895294 0.17329432 0.15643704]
 [0.2502429 0.54922666 0.71459592]
 [0.66019738 0.2799339 0.95486528]]
```