Vectors

Linear Algebra

- Linear algebra is a branch of mathematics that deals with the study of linear systems, which are sets of equations involving linear functions of variables.
- It deals with **vectors**, **vector spaces** (linear spaces), **linear transformations**, and **matrices**.

Scalar→ Numbers like 2, 5, 3

Vectors → Collection of 1D numbers → **[1, 2, 3]**

Matrices → Collection of 2D numbers

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

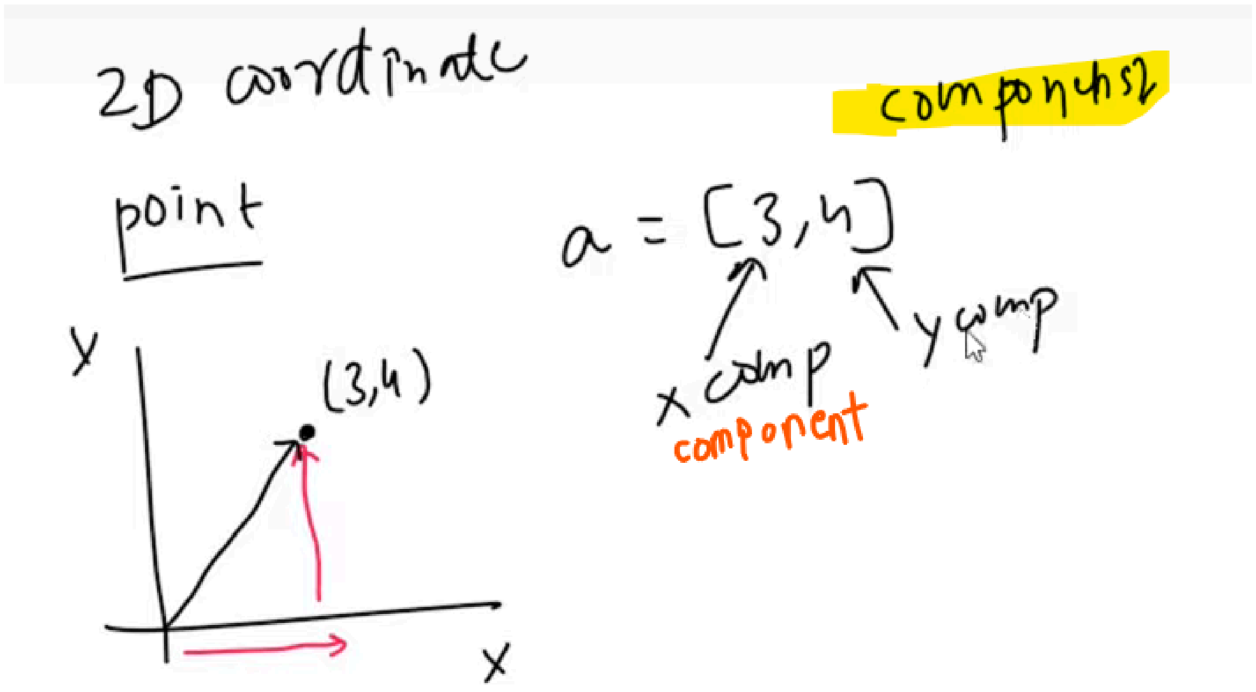
Tensors→ Objects in higher dimension

Uses in ML

- Useful when you have to work with higher dimension data
- Data representation → Table, Image, Text, Video, etc.

Vectors

When vector is not specified→You assume that it's a column vector.



- Vector is a point in a particular dimension system like 2D,3D,...nD
- Represented in square brackets → $[x1, x2, x3 \ldots xn]$

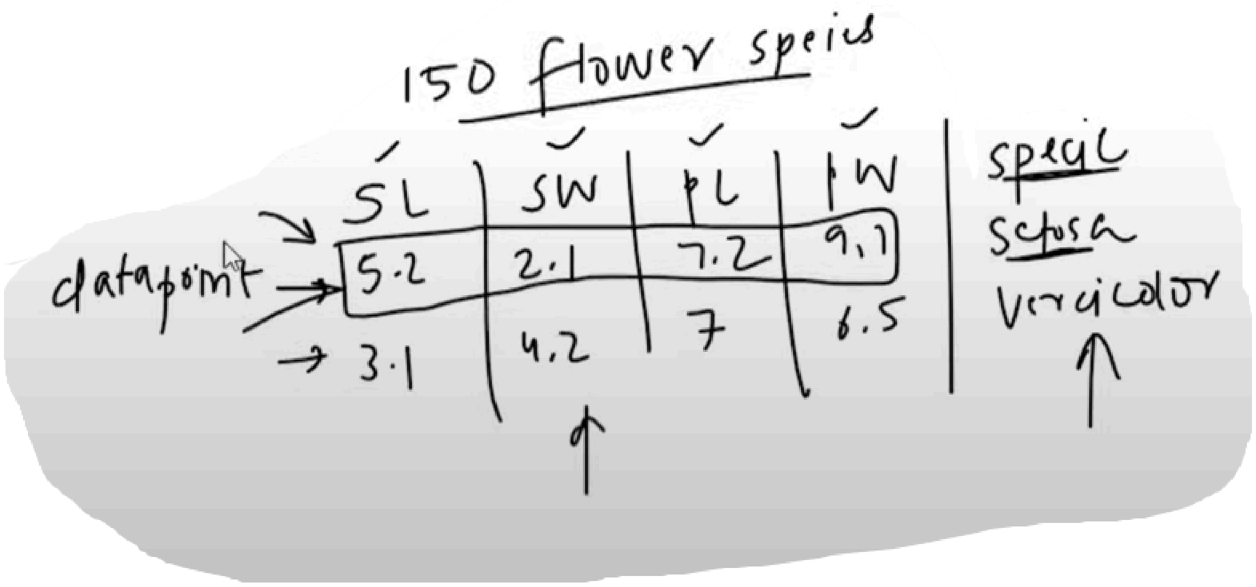
Dimension

- Vector coordinate system
- 2D, 3D, ...nD (2,3,...n)

How do vectors work in ML?

There's concept called **Feature Vector**

- You consider the Input features = Data point
- When you have 4 columns, each row can be used as vector..and it'll be 4D vector
    - It will be a point in nD
- We will give this vector to ML model and it will give prediction that it's Setosa.



- In iris dataset, species is Target column (You want to predict it)

💡 **You cannot use Characters in Vectors.**

- So, you have to convert them into numbers
- Ex. Male = 1, Female=2

Convert Characters into numbers

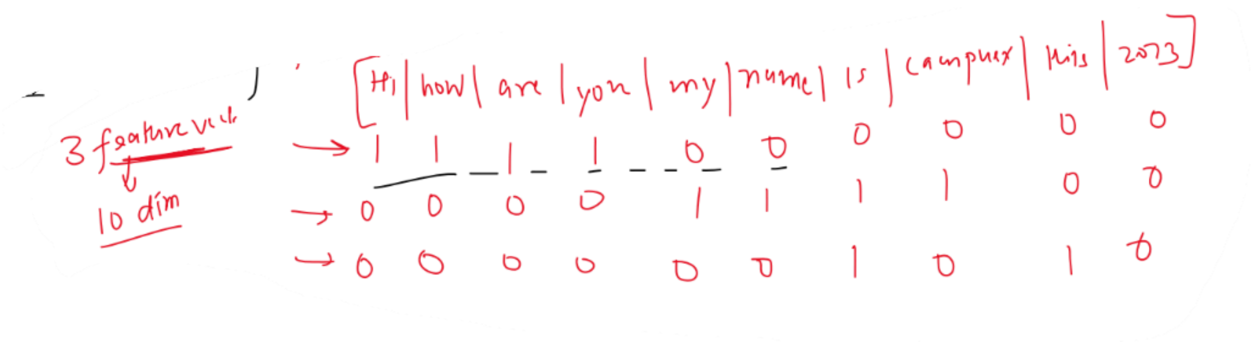
Bag of words (BoW) technique

- It is used to represent text data in a way that machine learning algorithms can process
- The idea behind BoW is to treat each document as a collection (or "bag") of words, without considering the order of the words or grammar.
- The key is to focus only on the **frequency** of words in a document.

1. **Tokenization**:
    - First, you split the text into individual words (tokens). For example, for the sentence "The cat sat on the mat," the tokens would be:
        - `["The", "cat", "sat", "on", "the", "mat"]`.
2. **Create Vocabulary**:
    - The next step is to create a vocabulary (or dictionary) from all unique words in the entire dataset (corpus). For example, for two sentences:The vocabulary will be: `["The", "cat", "sat", "on", "the", "mat", "dog"]`.
        - Sentence 1: "The cat sat on the mat."
        - Sentence 2: "The dog sat on the mat."
3. **Create Feature Vectors**:
    - For each sentence or document, we then create a **vector** representing the frequency of each word from the vocabulary. Each word in the vocabulary gets its own dimension in the vector, and the value in that dimension corresponds to the frequency of that word in the sentence.

    For example:

    - Sentence 1: "The cat sat on the mat."
        - BoW vector: `[1, 1, 1, 1, 1, 1, 0]`
          (The words are: "The", "cat", "sat", "on", "the", "mat", "dog". The frequency of each word is listed in the vector.)
    - Sentence 2: "The dog sat on the mat."
        - BoW vector: `[1, 0, 1, 1, 1, 1, 1]`



Hi	how	are	you	my	name	is	campusx	hiis	2023
1	1	1	1	0	0	0	0	0	0
0	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	1	0	1	0

- The dimension will be **10D** as we have 10 words
- If the corpus is large, the vocabulary can become very large, leading to high-dimensional vectors that can be sparse (containing many zeros). This can increase the complexity and computational cost.

Row and Column Vector

- **Row Vector:**
  - A row vector is an ordered list of **numbers written horizontally.** It is typically represented as a 1×n matrix.
  - A **1×n** matrix (1 row, *n* columns).

  Notation Example:

  $$\mathbf{v} = \begin{bmatrix} v_1 & v_2 & \cdots & v_n \end{bmatrix}$$

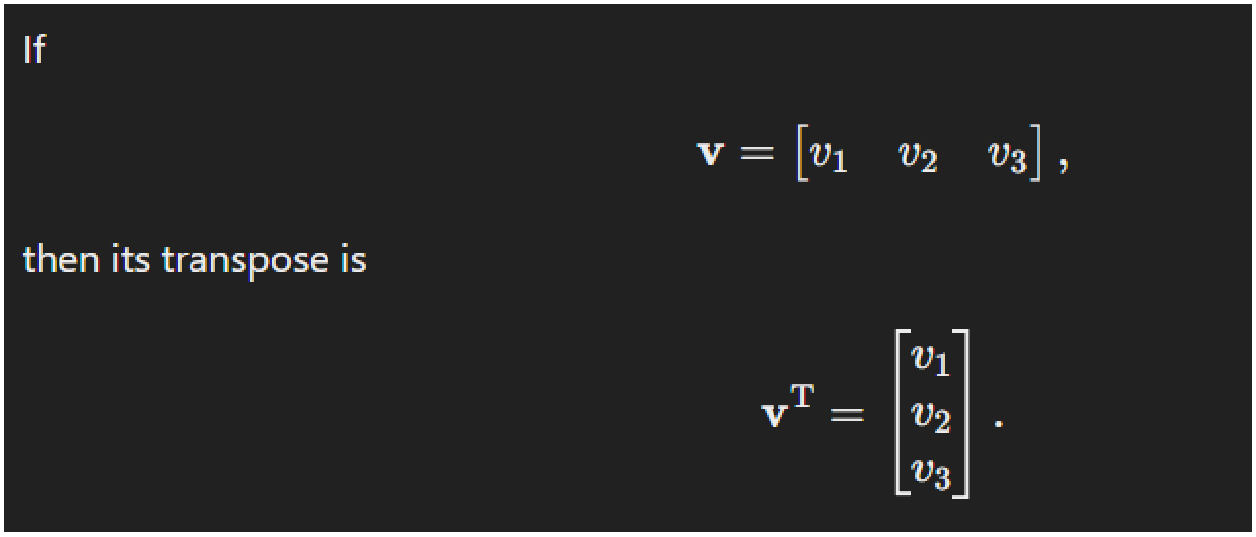
- **Column Vector :**
  - A column vector is an ordered list of numbers written vertically. It is represented as an n×1 matrix.
  - An **n×1** matrix (n*n* rows, 1 column).

  Notation Example:

  $$v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

Transpose Operation:

The transpose operation converts a row vector to a column vector and vice versa.



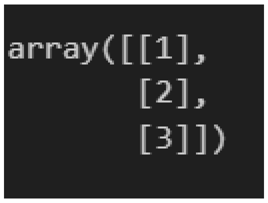
If

$$\mathbf{v} = \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix},$$

then its transpose is

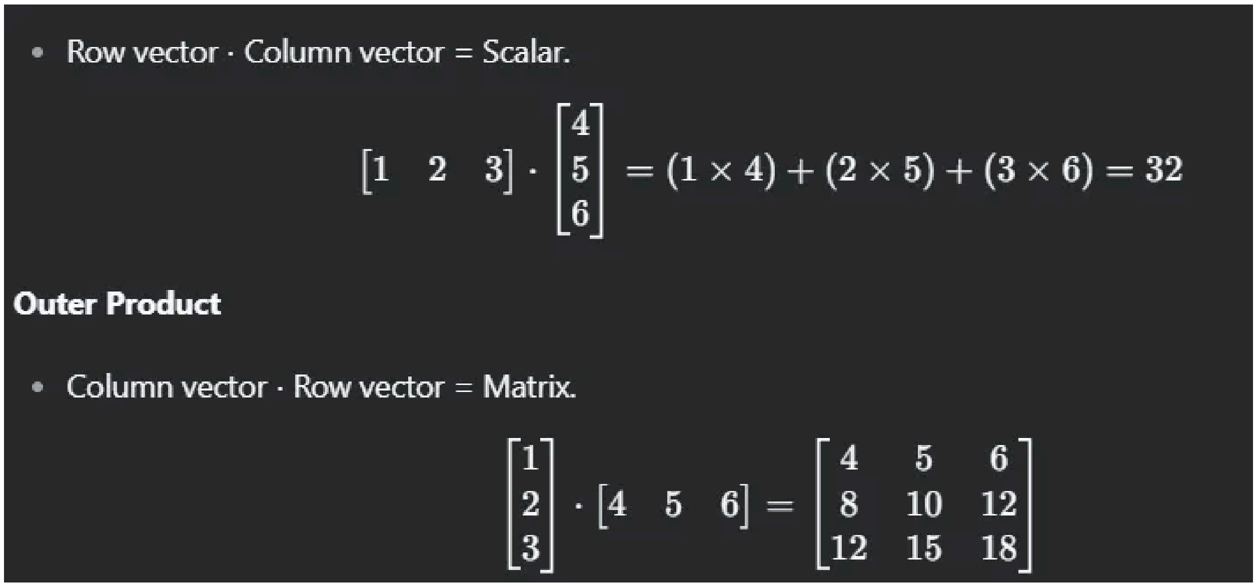
$$\mathbf{v}^{\mathrm{T}} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}.$$

```
v = np.array([[1, 2, 3]])  # row vector
v_transposed = v.T  # column vector
v_transposed
```



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

Dot Product



- Row vector · Column vector = Scalar.

$$\begin{bmatrix}1 & 2 & 3\end{bmatrix} \cdot \begin{bmatrix}4\\5\\6\end{bmatrix} = (1 \times 4) + (2 \times 5) + (3 \times 6) = 32$$

Outer Product

- Column vector · Row vector = Matrix.

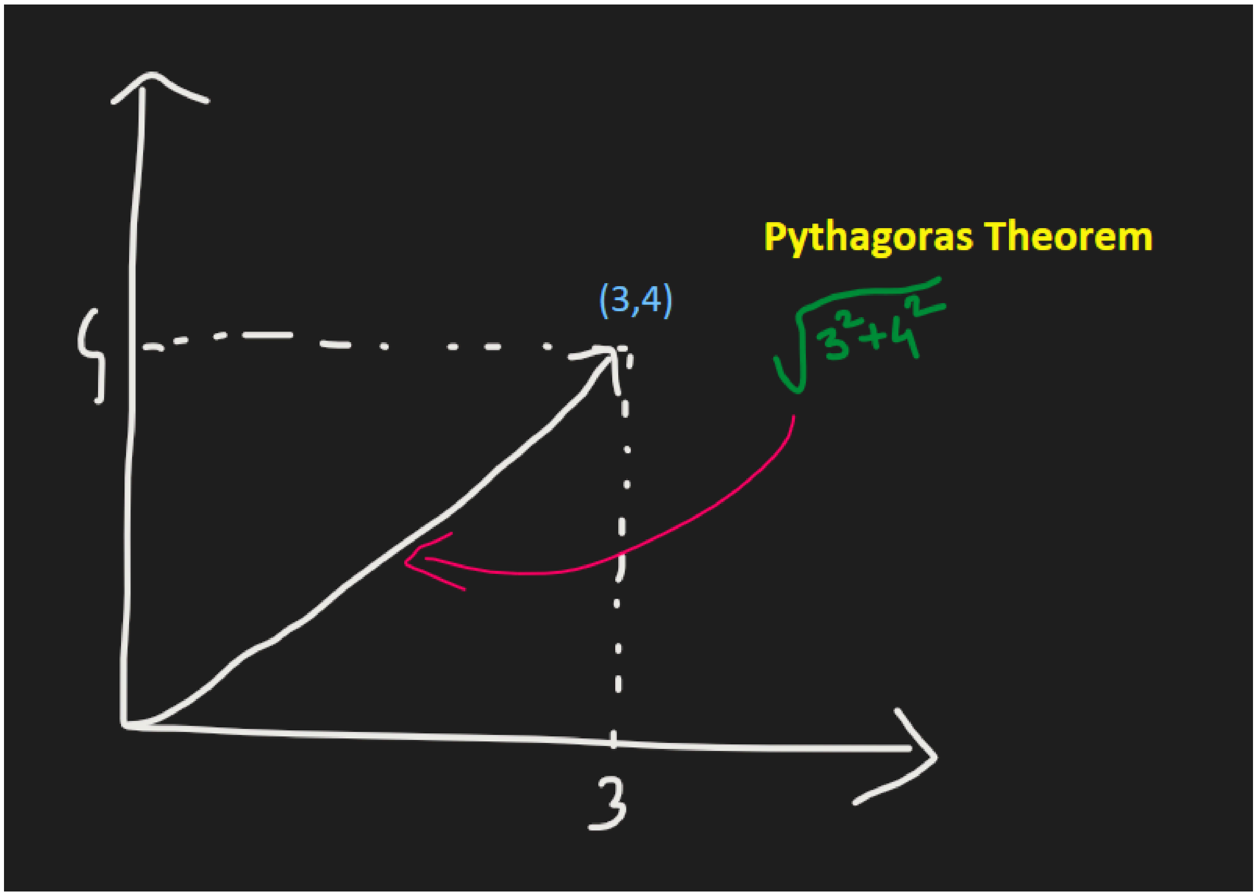
$$\begin{bmatrix}1\\2\\3\end{bmatrix} \cdot \begin{bmatrix}4 & 5 & 6\end{bmatrix} = \begin{bmatrix}4 & 5 & 6\\8 & 10 & 12\\12 & 15 & 18\end{bmatrix}$$

Key Differences

Aspect	**Row Vector**	**Column Vector**
Shape	1×n1×*n*	n×1*n*×1
Orientation	Horizontal	Vertical
Dot Product	Multiplies with a column vector.	Multiplies with a row vector.
Transpose	Transpose of a row vector is a column vector (and vice versa).	
Programming (NumPy)	Created as `shape=(1, n)` or 1D array.	Created as `shape=(n, 1)` or 1D array.

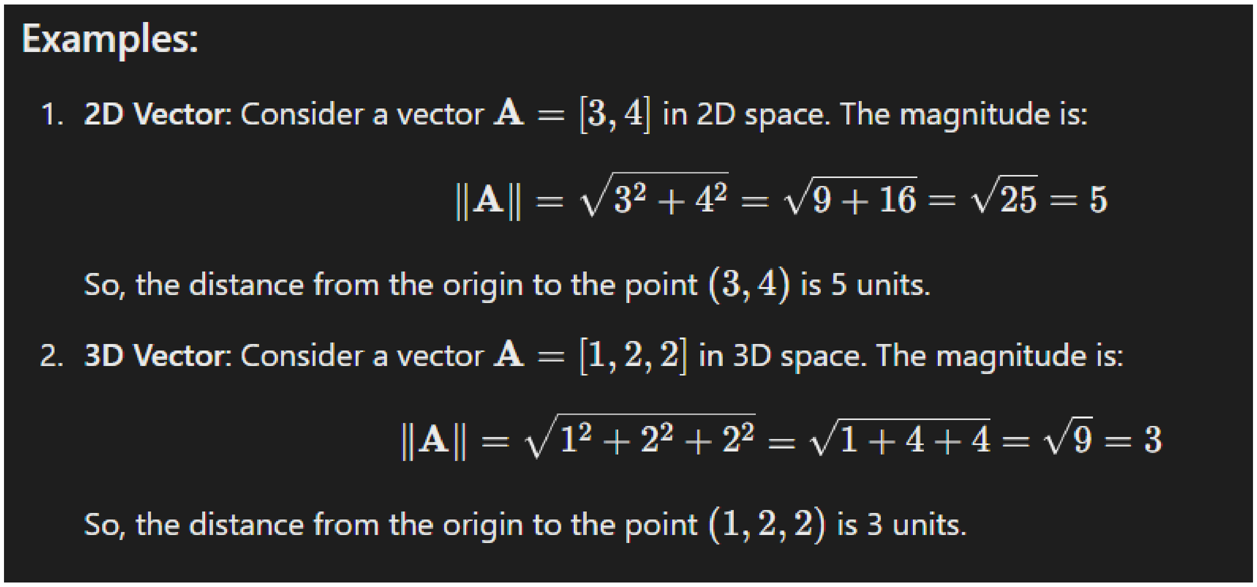
Distance From Origin ||A|| (Magnitude)

- For 2D vectors:



- For a vector $\mathbf{A} = [a_1, a_2, \ldots, a_n]$, n-dimensional space, the magnitude (or Euclidean norm) is calculated using the **Pythagorean theorem** and is given by the formula:

$$\|\mathbf{A}\| = \sqrt{a_1^2 + a_2^2 + \cdots + a_n^2}$$



Examples:

1. **2D Vector**: Consider a vector $\mathbf{A} = [3, 4]$ in 2D space. The magnitude is:

$$\|\mathbf{A}\| = \sqrt{3^2 + 4^2} = \sqrt{9 + 16} = \sqrt{25} = 5$$

So, the distance from the origin to the point $(3, 4)$ is 5 units.

2. **3D Vector**: Consider a vector $\mathbf{A} = [1, 2, 2]$ in 3D space. The magnitude is:

$$\|\mathbf{A}\| = \sqrt{1^2 + 2^2 + 2^2} = \sqrt{1 + 4 + 4} = \sqrt{9} = 3$$

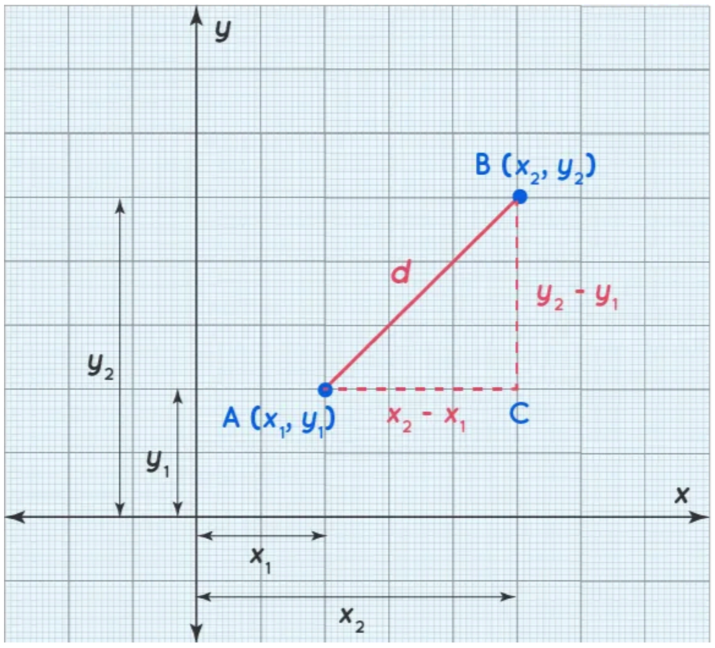
So, the distance from the origin to the point $(1, 2, 2)$ is 3 units.

In Python

```
# distance from origin → euclidean norm

import numpy as np

# Define an n-dimensional vector A
A = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 112, 12])

# Calculate the Euclidean distance from the origin (L2 norm)
distance = np.linalg.norm(A)

print("Euclidean distance from the origin:", distance)



Output:
Euclidean distance from the origin: 114.86513831445988
```

Euclidean Distance

- **Euclidean distance** is a measure of the straight-line distance between two points in a Euclidean space (i.e., a geometric space with the usual notions of distance). It's the most common way of measuring the distance between two points and is based on the **Pythagorean theorem**.



Formula for Euclidean Distance:

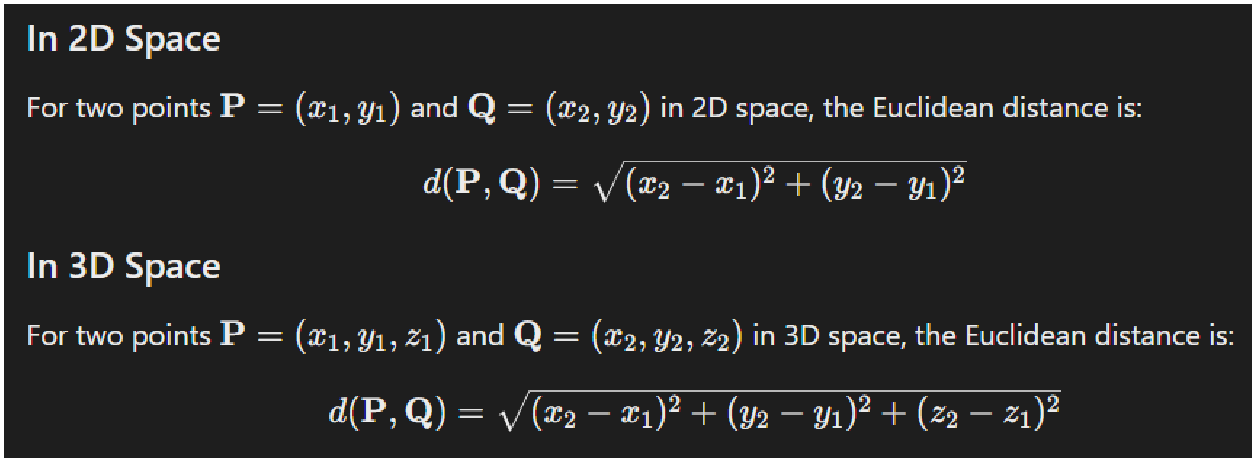
For two points

$\mathbf{P} = (x_1, y_1, \ldots, n_1)$ and

$\mathbf{Q} = (x_2, y_2, \ldots, n_2)$

in an n-dimensional space, the **Euclidean distance** between them is given by:

$$d(\mathbf{P}, \mathbf{Q}) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + \cdots + (n_2 - n_1)^2}$$



In 2D Space

For two points $\mathbf{P} = (x_1, y_1)$ and $\mathbf{Q} = (x_2, y_2)$ in 2D space, the Euclidean distance is:

$$d(\mathbf{P}, \mathbf{Q}) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

In 3D Space

For two points $\mathbf{P} = (x_1, y_1, z_1)$ and $\mathbf{Q} = (x_2, y_2, z_2)$ in 3D space, the Euclidean distance is:

$$d(\mathbf{P}, \mathbf{Q}) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

Python code

```
# to find euclidean distance in n-D
import numpy as np

# Define two n-dimensional vectors A and B
A = np.array([1, 2, 3, 4, 5])
B = np.array([6, 7, 8, 9, 10])

# Calculate the difference vector
difference = A - B

difference

Output: array([-5, -5, -5, -5, -5])
```

- First, we have calculated the difference between the 2 points

```
# Calculate the Euclidean distance between A and B (L2 norm of the difference)
```

```
distance = np.linalg.norm(difference)

print("Euclidean distance between A and B:", distance)

Output:
Euclidean distance between A and B: 11.180339887498949
```

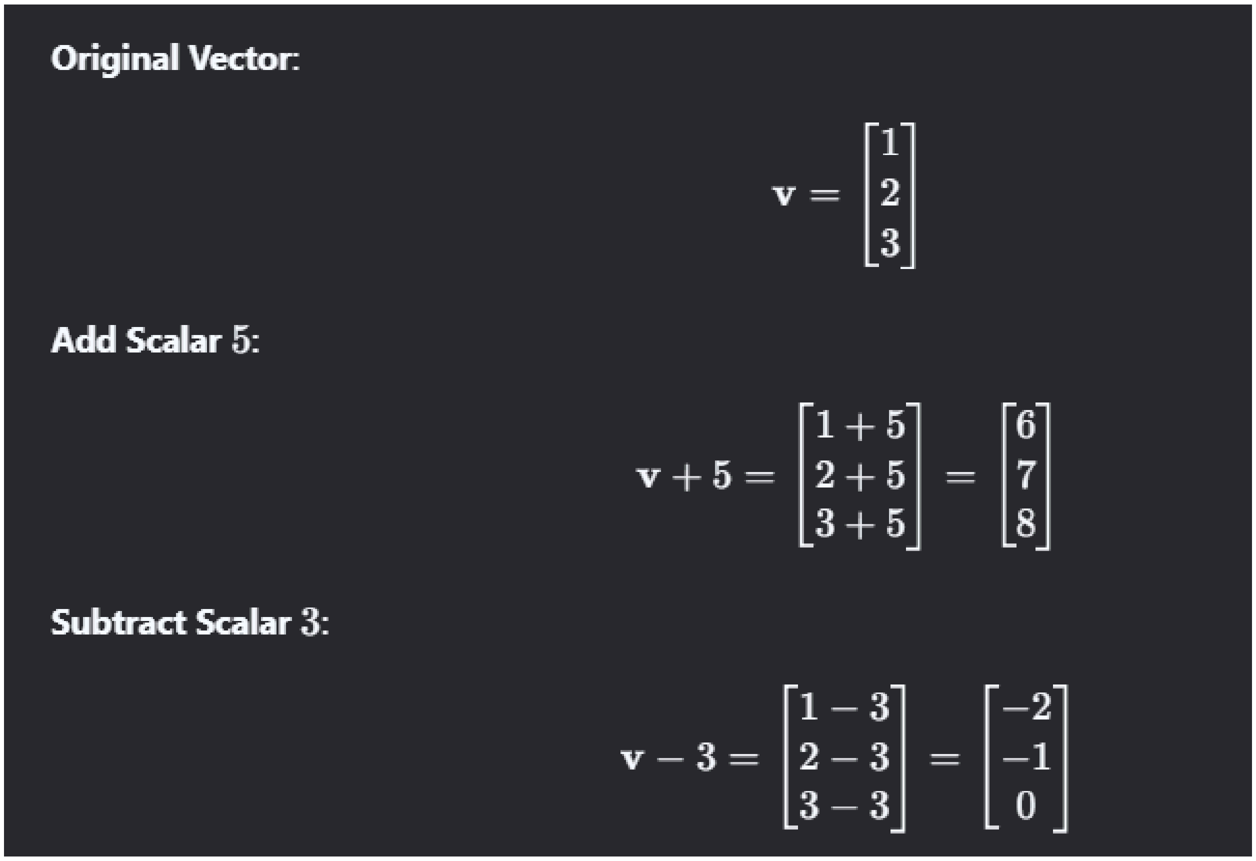
- With `linlag.norm()` calculate the diagonal (distance from one another)

Applications of Euclidean Distance:

- **Clustering**: In clustering algorithms like **k-means**, Euclidean distance is often used to measure how close a point is to the center (or centroid) of a cluster.
- **Classification**: In **k-nearest neighbors (k-NN)**, Euclidean distance is used to find the closest neighbors of a data point in feature space.
- **Computer Vision**: In image recognition or similarity analysis, Euclidean distance can be used to measure the difference between feature vectors representing images.
- **Physics**: It represents the straight-line distance between two points in space.

Scalar Addition/Subtraction (Shifting)

- Scalar addition/subtraction is the operation of adding (or subtracting) a constant (a scalar) to every element of a matrix or vector.
- This “shifts” the entire array by the given constant



Original Vector:

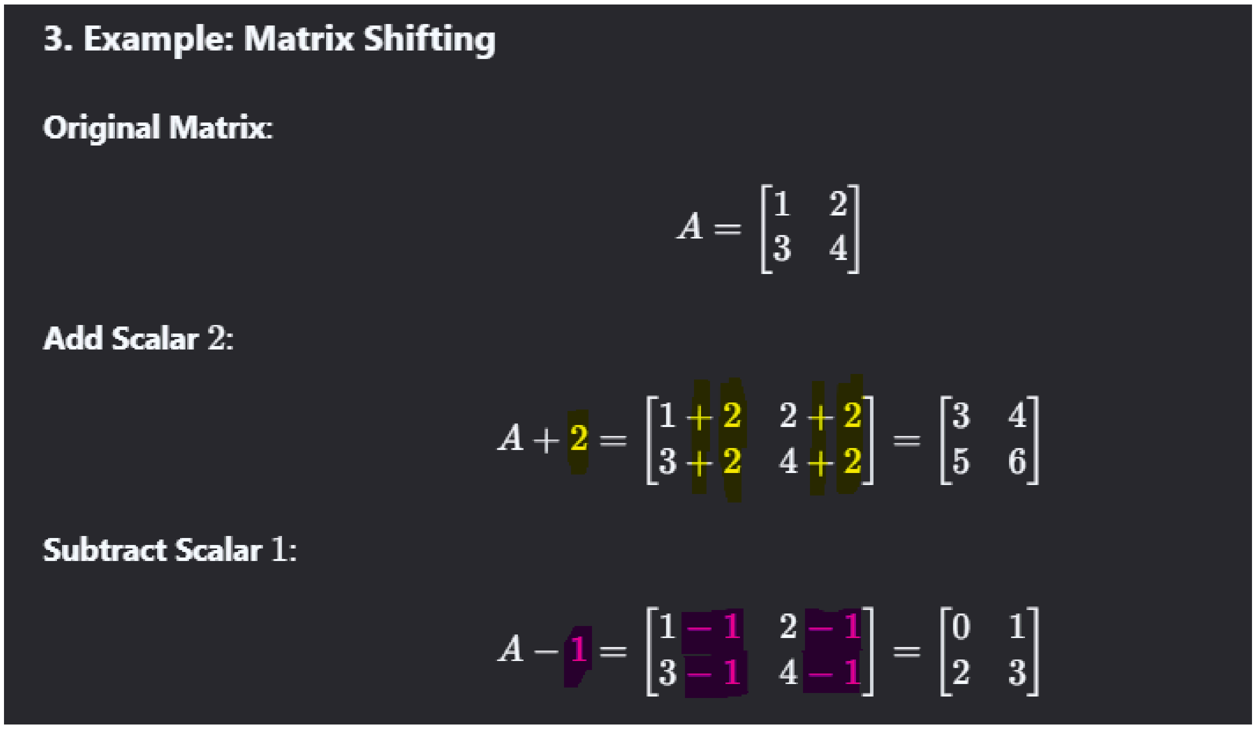
$$\mathbf{v} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

Add Scalar 5**:**

$$\mathbf{v} + 5 = \begin{bmatrix} 1+5 \\ 2+5 \\ 3+5 \end{bmatrix} = \begin{bmatrix} 6 \\ 7 \\ 8 \end{bmatrix}$$

Subtract Scalar 3**:**

$$\mathbf{v} - 3 = \begin{bmatrix} 1-3 \\ 2-3 \\ 3-3 \end{bmatrix} = \begin{bmatrix} -2 \\ -1 \\ 0 \end{bmatrix}$$



3. Example: Matrix Shifting

Original Matrix:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Add Scalar 2**:**

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

Subtract Scalar 1**:**

$$A - 1 = \begin{bmatrix} 1-1 & 2-1 \\ 3-1 & 4-1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$$

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

# Create a sample 2D array (matrix)
A = np.array([[1, 2, 3],
            [4, 5, 6],
            [7, 8, 9]])

# Define a scalar to shift by
c = 5

# Scalar addition (shifting all elements up by c)
A_plus_c = A + c

# Scalar subtraction (shifting all elements down by c)
A_minus_c = A - c

# Print the original and shifted matrices
print("Original Matrix A:")
print(A)
print("\nMatrix A after adding c (shift up):")
print(A_plus_c)
print("\nMatrix A after subtracting c (shift down):")
print(A_minus_c)
```

Output:

```
Original Matrix A:
[[1 2 3]
[4 5 6]
[7 8 9]]

Matrix A after adding c (shift up):
[[ 6  7  8]
[ 9 10 11]
[12 13 14]]
```

```
Matrix A after subtracting c (shift down):
[[-4 -3 -2]
[-1  0  1]
[ 2  3  4]]
```

Applications

1. **Data Normalization**:
   - Shift data to have a mean of 0 (e.g., Xnormalized=X−μ$X\text{normalized}=X-\mu$).
2. **Image Processing**:
   - Adjust pixel brightness by adding/subtracting a scalar.
3. **Gradient Descent**:
   - Shift weights/biases during optimization.

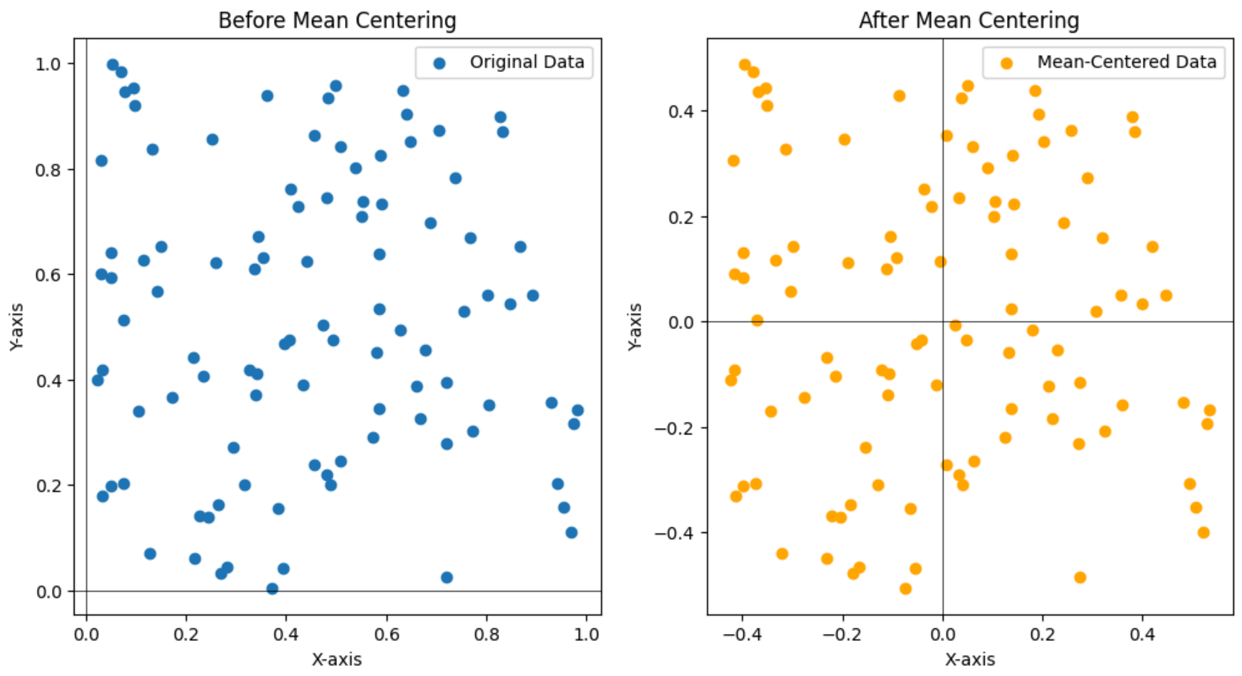
Mean Centering in Machine Learning: Applications and Benefits

Mean centering is a valuable pre-processing technique used in various machine learning applications. It offers improvements in model performance, convergence speed, and interpretability. Here are some practical examples where mean centering is commonly applied:

1. **Principal Component Analysis (PCA):**
   PCA, a dimensionality reduction technique, transforms data into a new coordinate system based on directions of highest variance (principal components). Mean centering the data
   before applying PCA is crucial. This ensures that the first principal component accurately represents the direction of maximum variance within the *dataset itself*, rather than being skewed by the data's position in the original coordinate system.
2. **Linear Regression:**
   In linear regression, mean centering enhances the interpretability of model coefficients. By mean centering features, the intercept term becomes

meaningful: it represents the expected value of the dependent variable when *all* independent variables are set to their mean values. Furthermore, mean centering can mitigate multicollinearity issues, particularly when interaction terms are present in the model.

3. **Gradient-Based Optimization Algorithms:**
   Algorithms like gradient descent, used in many machine learning models, can converge faster when input features are mean-centered. Mean centering improves the conditioning of the optimization problem, allowing gradient descent to take larger, more consistent steps towards finding the optimal solution.
4. **Clustering Algorithms (e.g., k-means):**
   Mean centering improves the performance of clustering algorithms like k-means. It ensures that the initial positions of cluster centroids are not unduly influenced by the data's location within the coordinate system. This leads to quicker convergence and improved clustering results.
5. **Regularization Techniques (e.g., Ridge, LASSO):**
   In models utilizing regularization (like Ridge or LASSO regression), mean centering ensures the regularization term applies consistently across all features. By centering the features, the model is less prone to inappropriately penalizing the intercept term, which ultimately promotes better generalization.



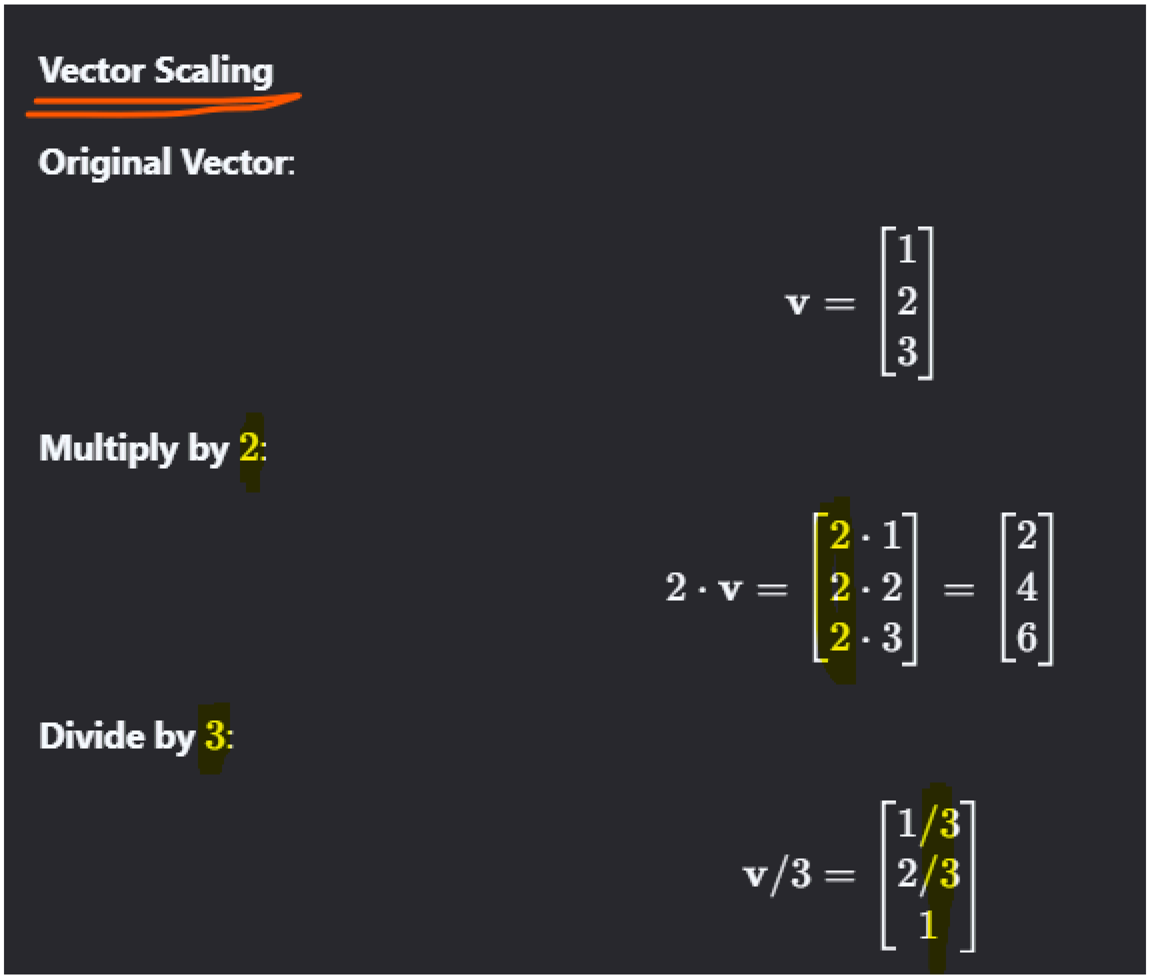
Common Mistakes

- **Confusing with Vector Addition**:
  - Scalar shifting affects all elements uniformly.
  - Vector addition requires vectors of the same size (e.g., v+w$\mathbf{v}+\mathbf{w}$).
- **Ignoring Data Types**:
  - Shifting integers by a float converts the array to float (e.g., `[1, 2] + 0.5 → [1.5, 2.5]`).

Scalar Multiplication/Division [Scaling]

- Scalar multiplication and division involve multiplying or dividing each element of a **vector** or **matrix** by a scalar value. These operations **scale** the magnitude of the data uniformly.
- Scalar multiplication involves multiplying a vector by a scalar (a single number).

- The operation scales the vector, either stretching or shrinking its length, depending on whether the scalar is greater than or less than 1.
- If the scalar is negative, it also **reverses the direction** of the vector.



Vector Scaling

Original Vector:

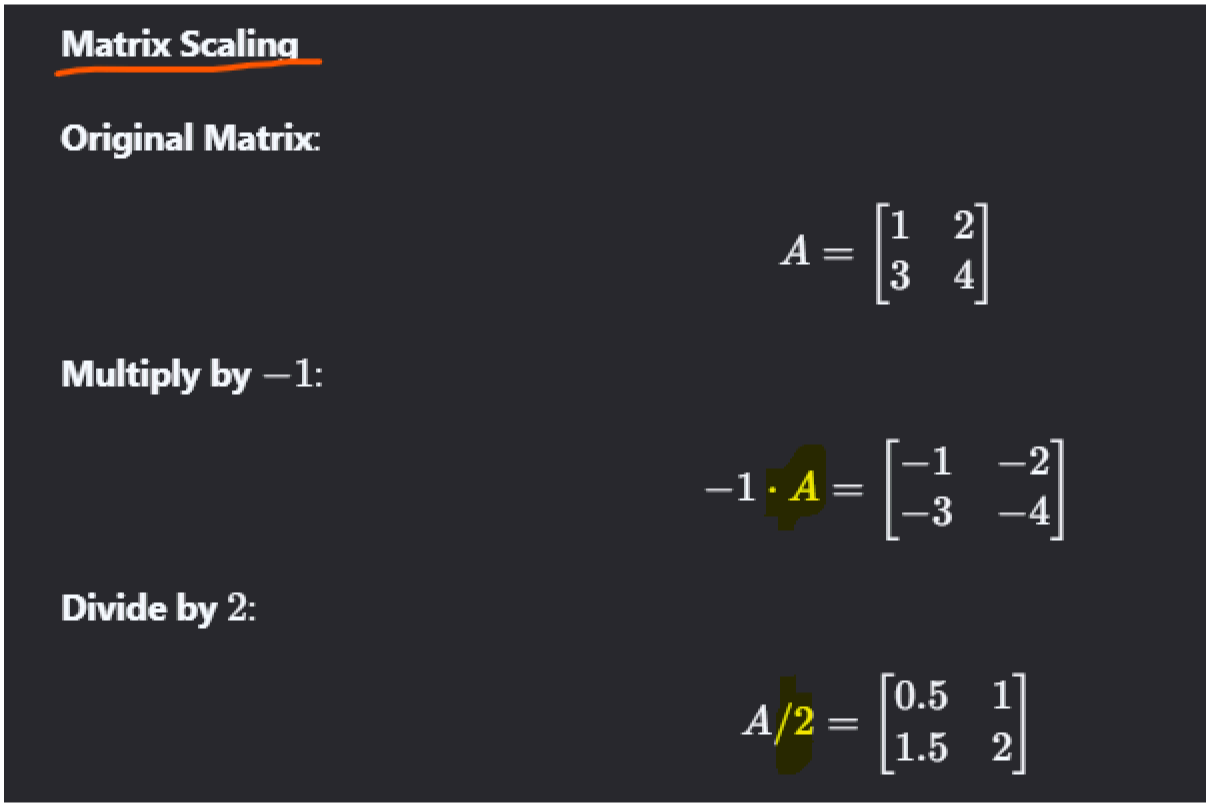
$$\mathbf{v} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

Multiply by 2:

$$2 \cdot \mathbf{v} = \begin{bmatrix} 2 \cdot 1 \\ 2 \cdot 2 \\ 2 \cdot 3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix}$$

Divide by 3:

$$\mathbf{v}/3 = \begin{bmatrix} 1/3 \\ 2/3 \\ 1 \end{bmatrix}$$



Matrix Scaling

Original Matrix:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Multiply by -1:

$$-1 \cdot A = \begin{bmatrix} -1 & -2 \\ -3 & -4 \end{bmatrix}$$

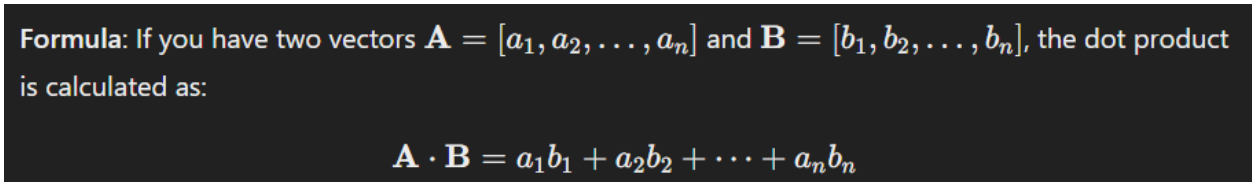
Divide by 2:

$$A/2 = \begin{bmatrix} 0.5 & 1 \\ 1.5 & 2 \end{bmatrix}$$

. Dot Product (also known as Scalar Product):

💡 **More important than cross product in ML.**

- The **dot product** is an operation between **two vectors**, which results in a scalar (not a vector).

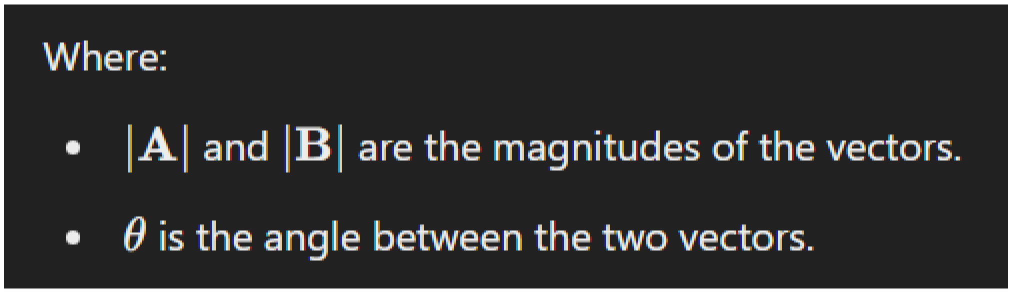


Formula: If you have two vectors $\mathbf{A} = [a_1, a_2, \ldots, a_n]$ and $\mathbf{B} = [b_1, b_2, \ldots, b_n]$, the dot product is calculated as:

$$\mathbf{A} \cdot \mathbf{B} = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

- **Geometric Meaning**: The dot product gives you the **magnitude of the projection** of one vector onto the other. It is related to the **cosine of the angle** between the two vectors, so:

$$\mathbf{A} \cdot \mathbf{B} = |\mathbf{A}||\mathbf{B}| \cos(\theta)$$


Where:

- $|\mathbf{A}|$ and $|\mathbf{B}|$ are the magnitudes of the vectors.
- θ is the angle between the two vectors.

Key Points:

- Commutative:
  - $A.B = B.A$
- Distributive
  - $A.(B + C) = A.B + A.C$

Dot Product using Python:

```
import numpy as np

# Define three vectors
A = np.array([1, 2, 3])
B = np.array([4, 5, 6])

print(np.dot(A, B))
print(A@B)

Output:
32
32
```

- `A@B` is also a syntax to calculate dot product.

Uses of Dot Product

- To compute similarity between 2 vectors
- Calculate projection
- Perform matrix multiplication

$$\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\| \, \|\mathbf{v}\| \cos\theta,$$

- $||u||$and $||v||$ are distances from origin (diagonal of the triangle)

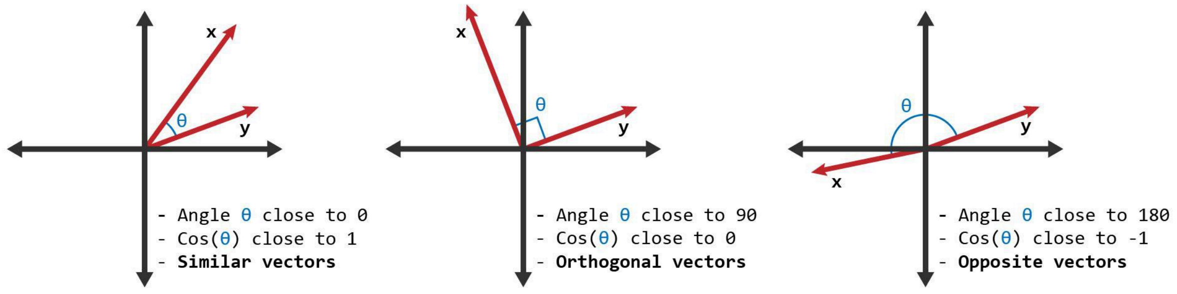
Calculate Angle between 2 vectors with dot product

$$\cos\theta = \frac{A \cdot B}{\|A\| \, \|B\|} \Rightarrow \theta = \cos^{-1}\left\{\frac{A \cdot B}{\|A\| \, \|B\|}\right\}$$

Cosine Similarity

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}$$

- The cosine similarity value ranges from **-1 to 1.**
  - **1** indicates the vectors point in exactly the same direction.
  - **0** indicates the vectors are orthogonal (at 90° to each other), meaning no similarity in direction.
  - **1** indicates the vectors are diametrically opposed.



Use:

- **Recommender Systems:**
  - **Helps compute similarity between users or items.**

Cosine Similarity using Python:

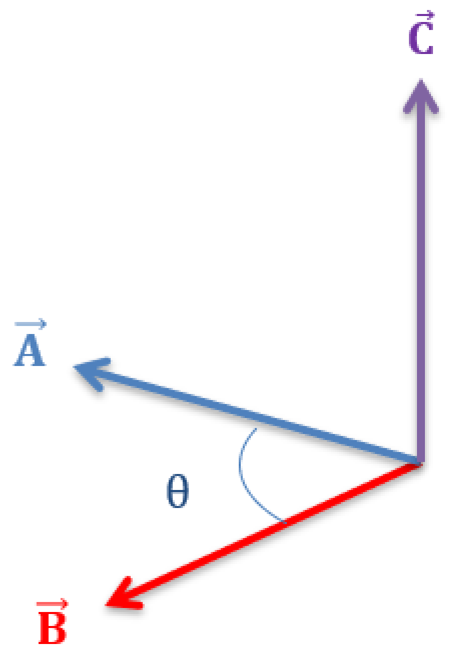
- Here, we want to find out the similarity between A & B and A & C

```
import numpy as np

# Define two vectors
```

```
A = np.array([1, 2, 3])
B = np.array([-4, -5, -6])
C = np.array([5,5,5])


# Calculate the cosine similarity
cosine_similarity = np.dot(A, B) / (np.linalg.norm(A) * np.linalg.norm(B))

print("Cosine similarity between A and B:", cosine_similarity)

# Calculate the cosine similarity
cosine_similarity = np.dot(A, C) / (np.linalg.norm(A) * np.linalg.norm(C))

print("Cosine similarity between A and C:", cosine_similarity)

Output:
Cosine similarity between A and B: -0.9746318461970762
Cosine similarity between A and C: 0.9258200997725513
```

Cross Product

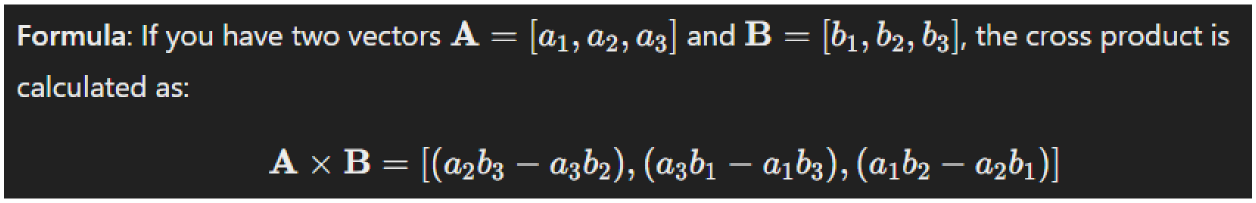
- The cross product is defined for 3-dimensional vectors and **produces another vector**.
- The resulting vector is **perpendicular** (orthogonal) to both original vectors, with a **magnitude equal to the area of the parallelogram formed by them.**



$$\vec{C} = \vec{A} \text{ x } \vec{B}$$

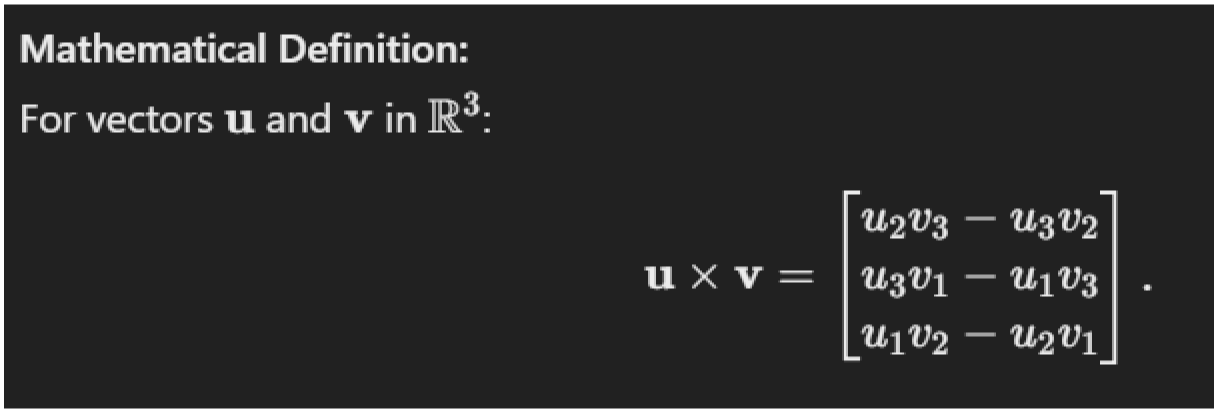
$$|C| = |A||B|\sin(\theta)$$

Formula:



Formula: If you have two vectors $\mathbf{A} = [a_1, a_2, a_3]$ and $\mathbf{B} = [b_1, b_2, b_3]$, the cross product is calculated as:

$$\mathbf{A} \times \mathbf{B} = [(a_2b_3 - a_3b_2), (a_3b_1 - a_1b_3), (a_1b_2 - a_2b_1)]$$

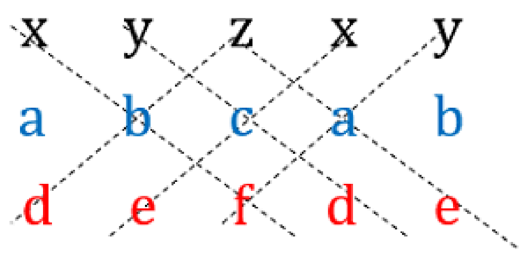


Mathematical Definition:

For vectors $\mathbf{u}$ and $\mathbf{v}$ in $\mathbb{R}^3$:

$$\mathbf{u} \times \mathbf{v} = \begin{bmatrix} u_2v_3 - u_3v_2 \\ u_3v_1 - u_1v_3 \\ u_1v_2 - u_2v_1 \end{bmatrix}.$$

$$\vec{\mathbf{A}} = [a, b, c] \qquad \vec{\mathbf{B}} = [d, e, f]$$



$$\vec{\mathbf{A}} \times \vec{\mathbf{B}} = [(bf - ce), (cd - af), (ae - bd)]$$

Equation of Hyperplane

Hyperplane

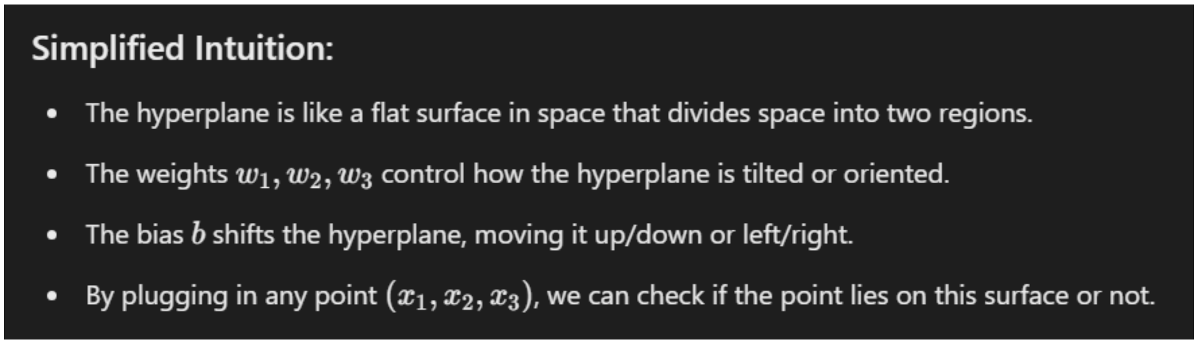
A **hyperplane** in an n-dimensional real space $\mathbb{R}^n$ is an (n–1)-dimensional subspace that divides the space into two half-spaces. In simpler terms:

- In 2D, a hyperplane is a line.
- In 3D, a hyperplane is a plane.
- In higher dimensions, it is the generalization of these concepts.

Equation of a Hyperplane

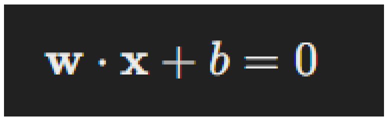
$$w_1 \cdot x_1 + w_2 \cdot x_2 + w_3 \cdot x_3 + b = 0$$

- x_1, x_2, x_3 are the **coordinates** of any point on the hyperplane. For example, in 3D, the coordinates of any point are represented as (x_1, x_2, x_3)
- w_1, w_2, w_3 are the **weights** (or "normal vector" components). These determine the **direction** of the hyperplane. Think of them as numbers that control how the hyperplane is tilted or oriented.
- b is the **bias**. This is a number that **shifts** the hyperplane. If b is 0, the hyperplane will pass through the origin (the center of the coordinate system), and if b is non-zero, it shifts the hyperplane away from the origin.

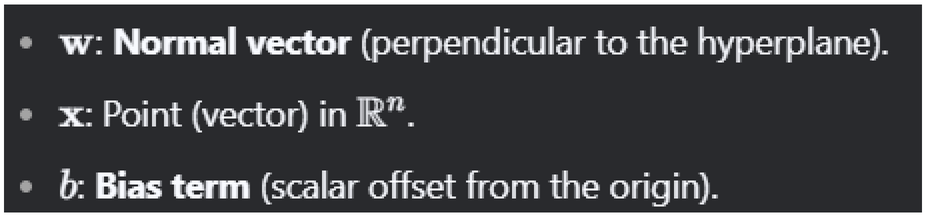


Simplified Intuition:

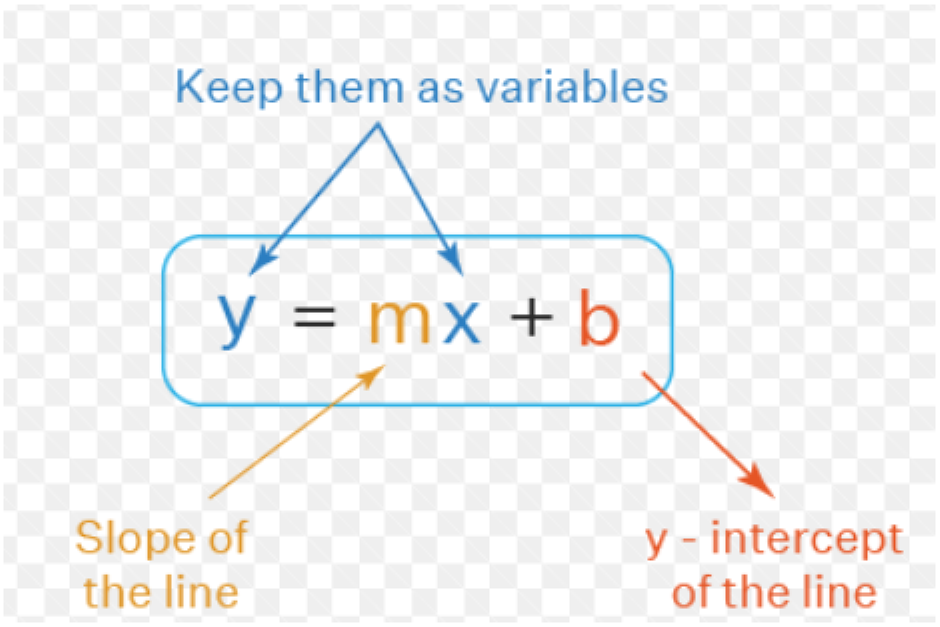
- The hyperplane is like a flat surface in space that divides space into two regions.
- The weights w_1, w_2, w_3 control how the hyperplane is tilted or oriented.
- The bias b shifts the hyperplane, moving it up/down or left/right.
- By plugging in any point (x_1, x_2, x_3), we can check if the point lies on this surface or not.



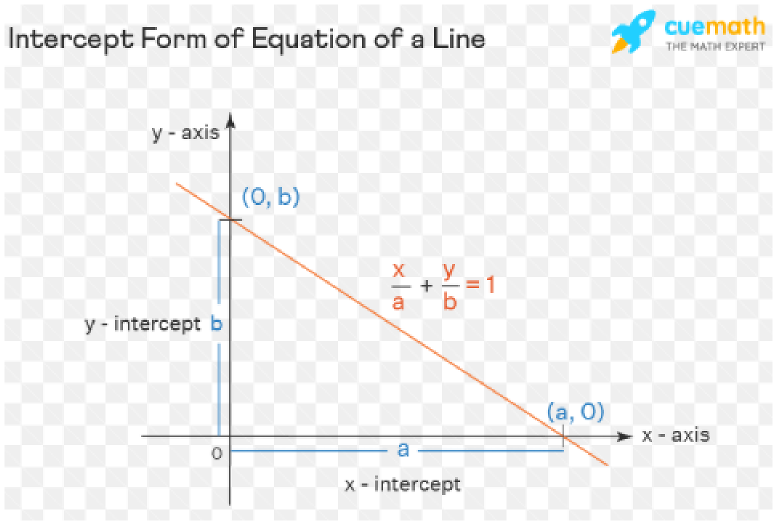
$$\mathbf{w} \cdot \mathbf{x} + b = 0$$



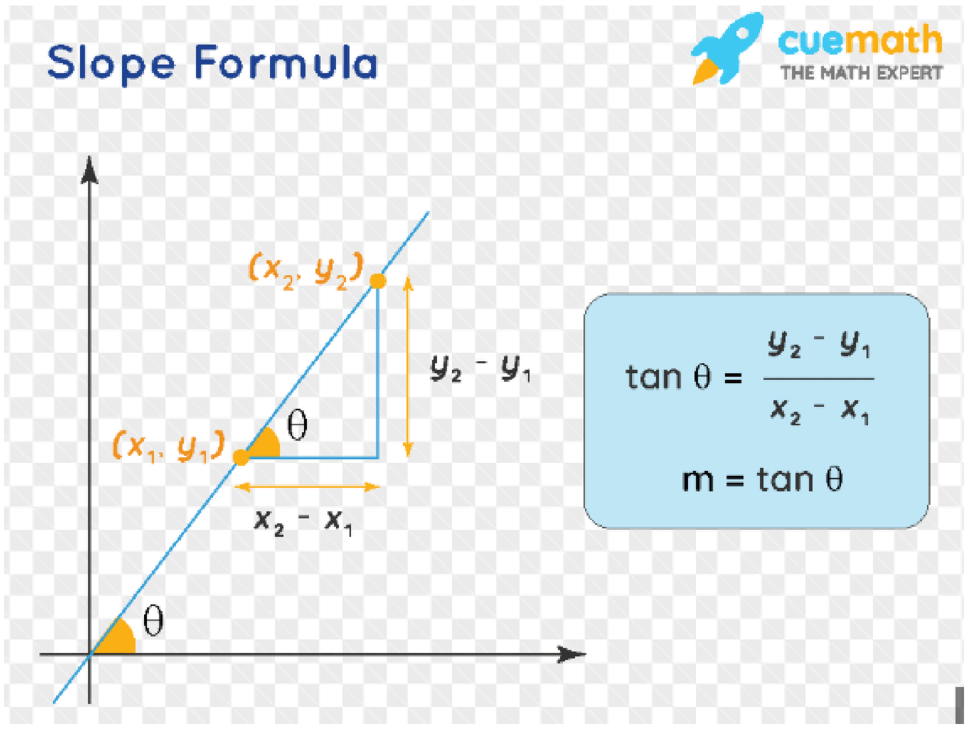
- $\mathbf{w}$: **Normal vector** (perpendicular to the hyperplane).
- $\mathbf{x}$: Point (vector) in $\mathbb{R}^n$.
- b: **Bias term** (scalar offset from the origin).

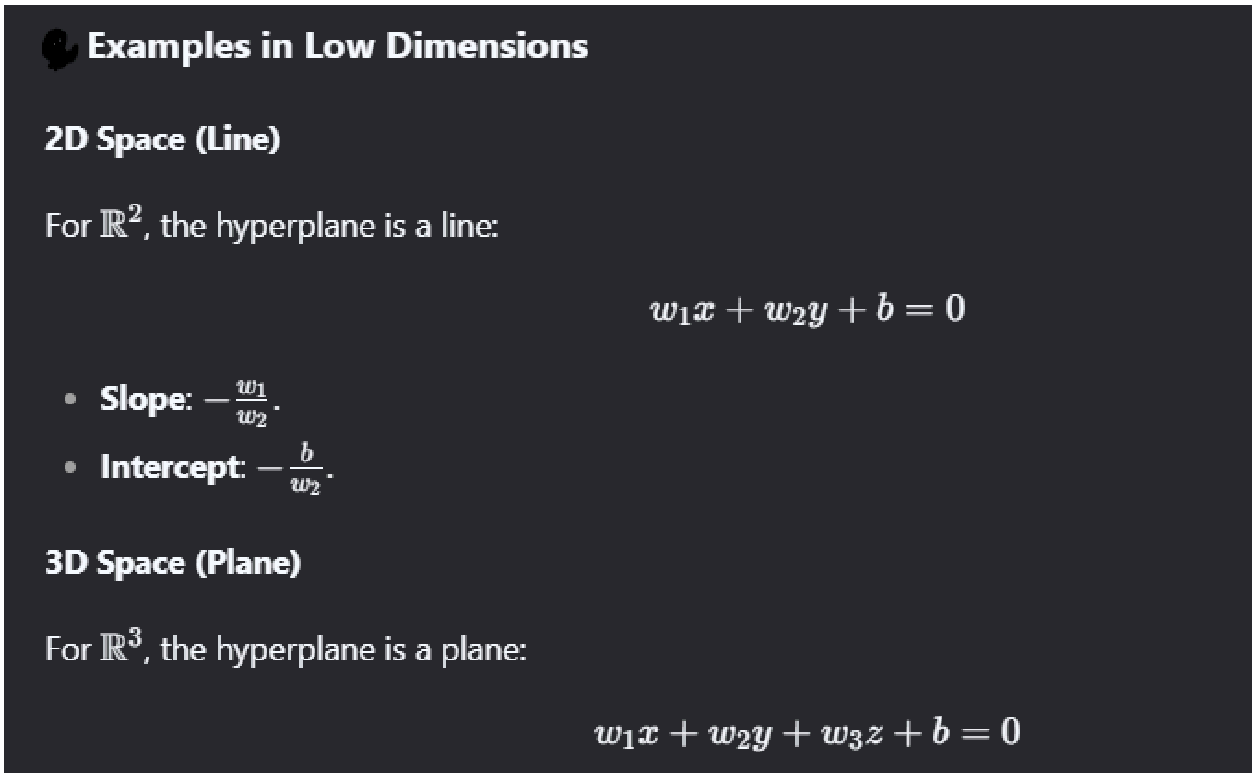


Intercept:



Slope:





Examples in Low Dimensions

2D Space (Line)

For $\mathbb{R}^2$, the hyperplane is a line:

$$w_1x + w_2y + b = 0$$

- **Slope**: $-\frac{w_1}{w_2}$.
- **Intercept**: $-\frac{b}{w_2}$.

3D Space (Plane)

For $\mathbb{R}^3$, the hyperplane is a plane:

$$w_1x + w_2y + w_3z + b = 0$$