🚀 Machine Learning: A Subset of Artificial Intelligence

Machine Learning: A Subset of Artificial Intelligence

Machine Learning (ML) is a branch of Artificial Intelligence (AI) that focuses on building systems capable of learning from data to make predictions, draw inferences, and recognize patterns **without being explicitly programmed**.

Key Concepts in Machine Learning

Prediction

Using a model to estimate future outcomes.

Examples:

- Weather forecasting
- Predicting stock prices
- Estimating unknown values in data

Inference

Understanding relationships within the data and drawing insights.

Example:

- Identifying which factors most influence customer churn

Explicit vs. Implicit Programming

Explicit Programming

Involves giving the computer exact instructions on what to do. Every rule and step is manually coded by the programmer.

Example: To sort a list of numbers in ascending order, you would:

- Compare two numbers
- Swap them if necessary

- Repeat this until sorted

Implicit Programming (Used in ML)

Allows the computer to learn from data and figure out the rules on its own. You don't explicitly program every step—the system learns patterns and behaviors through examples.

Example: To teach a computer to recognize whether an image is a cat or dog:

- You don't provide step-by-step rules
- Instead, you give many labeled images
- The system learns to recognize patterns by itself

Main Difference:

Explicit Programming	Implicit Programming
Programmer writes detailed rules	System learns rules from data
"Do this, and do it this way"	"Learn from this data and figure it out"

Machine Learning Development Life Cycle (MLDLC)

1. Problem Definition

- Clearly define the business or technical problem
- Identify success metrics and constraints (e.g., accuracy, speed, budget)
- Understand data availability and stakeholder needs

Key Outputs:

- Problem statement
- Success criteria
- Initial hypotheses

---

2. Data Collection

- Gather relevant data from sources (databases, APIs, web scraping, sensors, etc.)
- Ensure data is representative of the problem domain

Key Considerations:

- Data privacy and security

- Licensing and usage rights

---

3. Data Preprocessing

- Clean and prepare the data for analysis
- Handle missing, noisy, or inconsistent data
- Perform Exploratory Data Analysis (EDA)

Key Steps:

- Data cleaning
- Scaling, encoding, feature engineering
- Splitting data into training, validation, and test sets

---

4. Model Development

- Select and experiment with different ML algorithms (e.g., Linear Regression, Decision Trees, Neural Networks)
- Train the model using the training dataset
- Fine-tune hyperparameters for performance

Key Activities:

- Algorithm selection
- Model architecture design
- Training and validation

---

5. Model Evaluation

- Assess model performance using metrics like Precision, Recall, F1-Score, or Mean Squared Error (MSE)
- Test on unseen data to verify generalizability

Key Outputs:

- Performance metrics
- Confusion matrix, ROC curves, etc.

---

6. Model Deployment

- Integrate the model into production (apps, web services, mobile apps)
- Set up real-time or batch prediction pipelines

Key Steps:

- Deployment architecture design
- Model containerization (e.g., Docker)
- Monitoring setup

---

7. Monitoring and Maintenance

- Continuously monitor for concept drift, data drift, and biases
- Update the model as new data or requirements emerge

Key Activities:

- Performance tracking
- Retraining with new data
- Incident handling

---

8. Model Explainability and Compliance

- Ensure model transparency for stakeholders
- Follow ethical AI guidelines and legal regulations

Key Tools:

- SHAP, LIME for interpretability
- Compliance checks (GDPR, CCPA, etc.)

---

9. Continuous Improvement

- Iterate on models and pipelines for better performance
- Incorporate feedback from stakeholders and users

Key Considerations:

- Scalability
- Automation of pipelines (CI/CD)

Basics & Backbone of Machine Learning

Tensor

A **Tensor** is essentially a **data structure** — a way of storing data.

- You can think of a tensor as a **container** that stores numerical values.

- Though it's possible to store characters or strings in tensors, that is rare in ML applications.
- Tensors generalize scalars, vectors, and matrices.

Dimensions in Tensor

Dimension	Example	Type
0-D	`Scalar`	Single value (e.g., `5`)
1-D	`Vector`	List of values (e.g., `[5, 10, 15]`)
2-D	`Matrix`	2D array of values
3D or more	`Tensor`	Multi-dimensional data

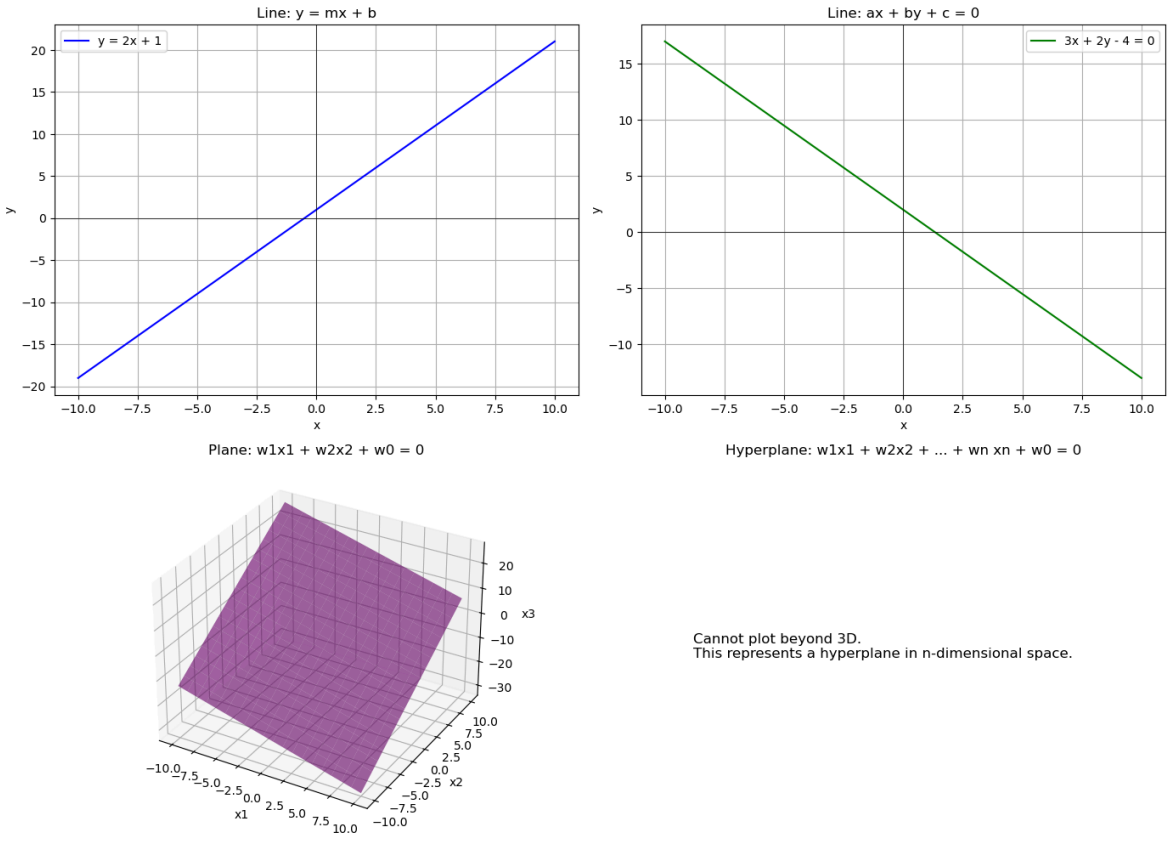
- **No. of axes = Rank of the tensor**

Examples:

- **$y = mx + b$** — represents a straight line in 2D
- **$ax + by + c = 0$** — also a line
- **$w_1x_1 + w_2x_2 + w_0 = 0$** — a plane in higher dimension (hyperplane)
- **$w_1x_1 + w_2x_2 + \ldots + w_nx_n + w_0 = 0$** — defines a **hyperplane** in n-dimensional space

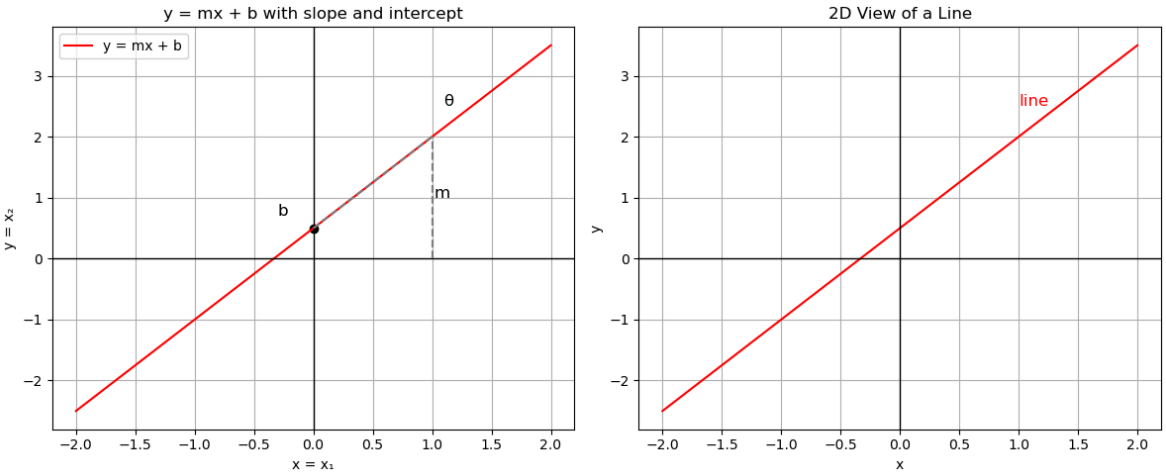
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# 1. y = mx + b
m, b = 2, 1
x = np.linspace(-10, 10, 100)
y = m * x + b

plt.figure(figsize=(14, 10))

plt.subplot(2, 2, 1)
plt.plot(x, y, label="y = 2x + 1", color="blue")
plt.title("Line: y = mx + b")
plt.xlabel("x")
plt.ylabel("y")
plt.grid(True)
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.legend()

# 2. ax + by + c = 0 → y = -(a/b)x - c/b
a, b_coef, c = 3, 2, -4
x2 = np.linspace(-10, 10, 100)
y2 = -(a/b_coef)*x2 - c/b_coef

plt.subplot(2, 2, 2)
plt.plot(x2, y2, label="3x + 2y - 4 = 0", color="green")
plt.title("Line: ax + by + c = 0")
plt.xlabel("x")
```

```
plt.ylabel("y")
plt.grid(True)
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.legend()

# 3. w1x1 + w2x2 + w0 = 0 → Plane in 3D
w1, w2, w0 = 1, -2, 3
fig = plt.subplot(2, 2, 3, projection='3d')
x_vals = np.linspace(-10, 10, 20)
y_vals = np.linspace(-10, 10, 20)
X, Y = np.meshgrid(x_vals, y_vals)
Z = -(w1 * X + w2 * Y + w0) / 1  # Assuming w3 = 1

fig.plot_surface(X, Y, Z, alpha=0.6, color='purple')
fig.set_title("Plane: w1x1 + w2x2 + w0 = 0")
fig.set_xlabel("x1")
fig.set_ylabel("x2")
fig.set_zlabel("x3")

# 4. Placeholder for n-Dimensional Hyperplane
plt.subplot(2, 2, 4)
plt.axis('off')
plt.title("Hyperplane: w1x1 + w2x2 + ... + wn xn + w0 = 0")
plt.text(0.1, 0.5, "Cannot plot beyond 3D.\nThis represents a hyperplane in n-di
         fontsize=12, va='center')

plt.tight_layout()
plt.show()
```



In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

# Set up figure
```

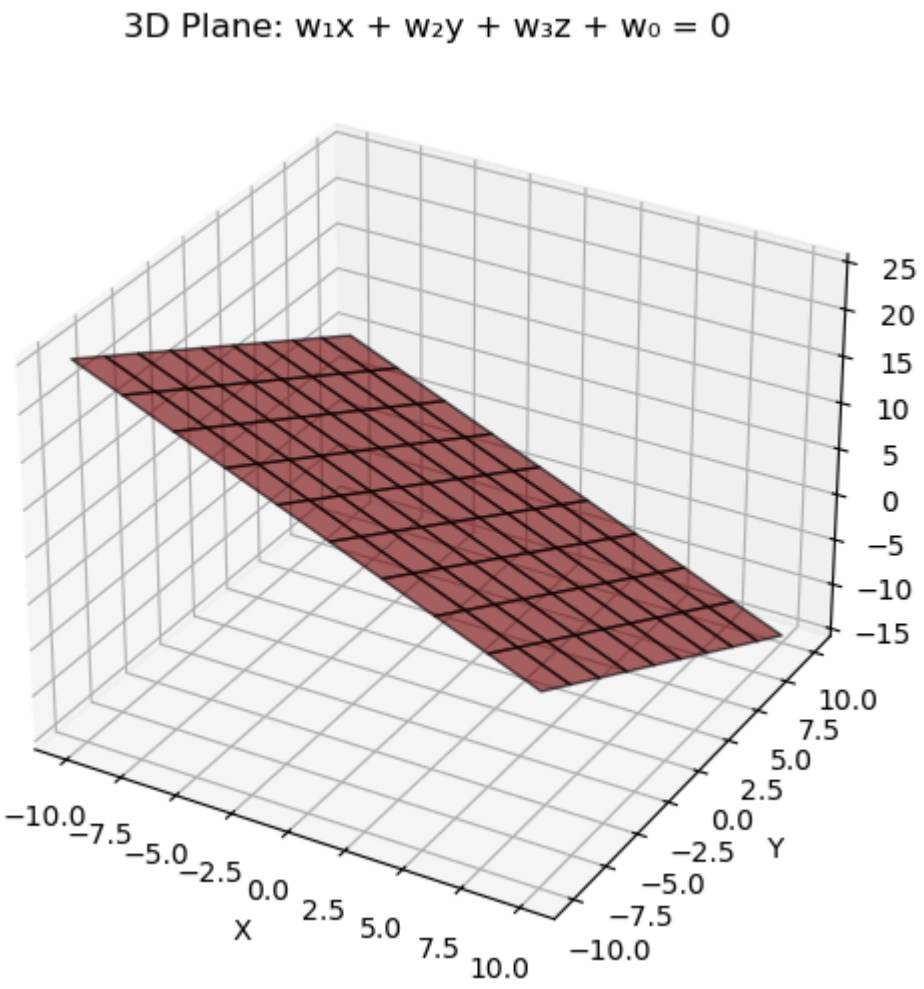
```
fig, axs = plt.subplots(1, 2, figsize=(12, 5))

# First subplot: y = mx + b with slope, intercept, and angle
x = np.linspace(-2, 2, 100)
m = 1.5   # slope
b = 0.5   # y-intercept
y = m * x + b

axs[0].plot(x, y, label='y = mx + b', color='red')
axs[0].axhline(0, color='black', linewidth=1)
axs[0].axvline(0, color='black', linewidth=1)
axs[0].scatter(0, b, color='black')
axs[0].annotate('b', xy=(0, b), xytext=(-0.3, b+0.2), fontsize=12)
axs[0].annotate('θ', xy=(1, m*1 + b), xytext=(1.1, m*1 + b + 0.5), fontsize=12)
axs[0].plot([0, 1], [b, m*1 + b], linestyle='--', color='gray')
axs[0].plot([1, 1], [0, m*1 + b], linestyle='--', color='gray')
axs[0].annotate('m', xy=(1.02, (m*1 + b)/2), fontsize=12)
axs[0].set_title("y = mx + b with slope and intercept")
axs[0].set_xlabel('x = x₁')
axs[0].set_ylabel('y = x₂')
axs[0].grid(True)
axs[0].legend()

# Second subplot: line in 2D plane
axs[1].plot(x, y, color='red')
axs[1].axhline(0, color='black', linewidth=1)
axs[1].axvline(0, color='black', linewidth=1)
axs[1].annotate('line', xy=(1, m*1 + b), xytext=(1, m*1 + b + 0.5), fontsize=12,
axs[1].set_title("2D View of a Line")
axs[1].set_xlabel('x')
axs[1].set_ylabel('y')
axs[1].grid(True)

plt.tight_layout()
plt.show()
```



In [2]:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Coefficients of the plane: w1*x + w2*y + w3*z + w0 = 0
w1, w2, w3, w0 = 1, 1, 1, -5  # Change as needed

# Create a meshgrid
x_vals = np.linspace(-10, 10, 10)
```

```
y_vals = np.linspace(-10, 10, 10)
X, Y = np.meshgrid(x_vals, y_vals)

# Solve for Z from the plane equation: w1*X + w2*Y + w3*Z + w0 = 0
Z = (-w1 * X - w2 * Y - w0) / w3

# Plotting
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, alpha=0.6, color='red', edgecolor='k')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_title("3D Plane: w₁x + w₂y + w₃z + w₀ = 0")
plt.show()
```

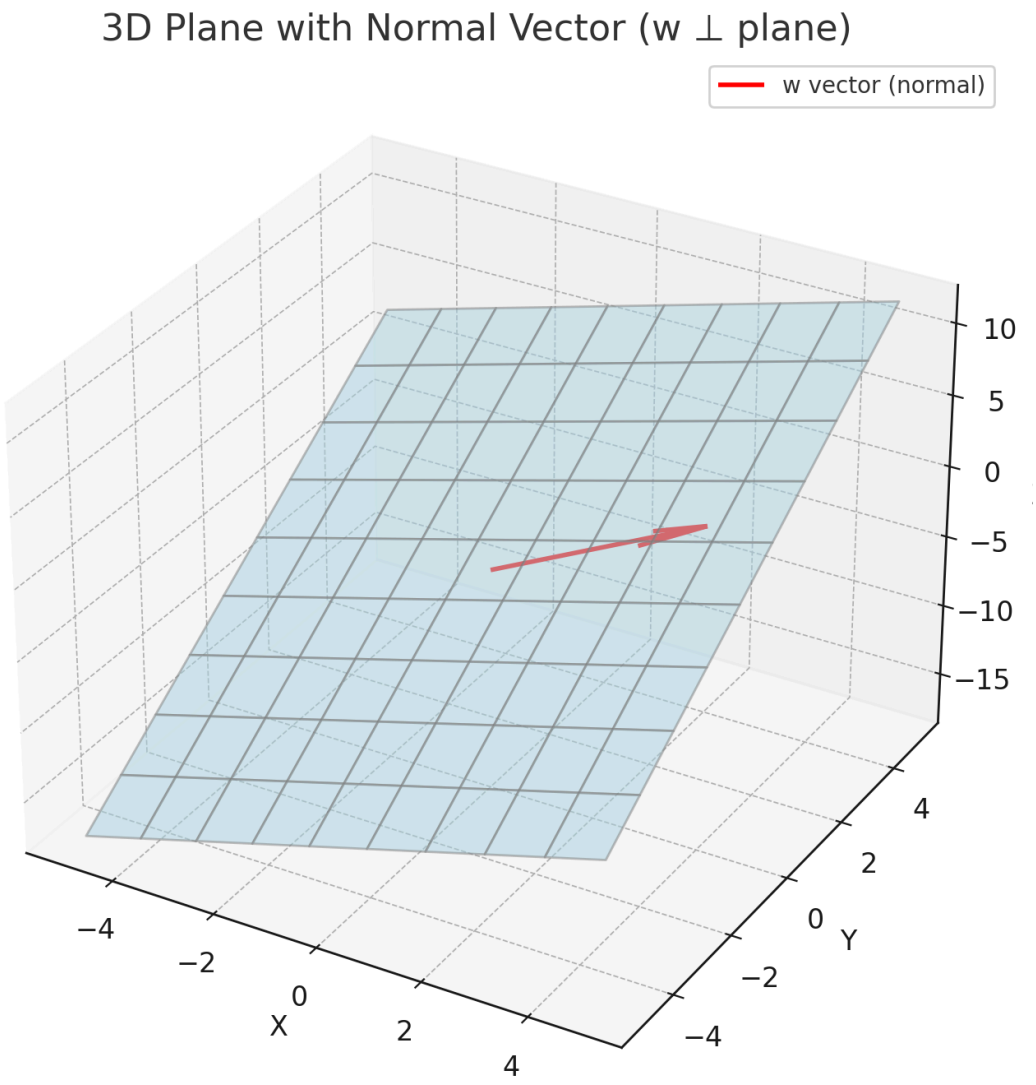


In [3]:

```
import numpy as np

# Random vector x and weight vector w
x = np.array([2, 3, -1, 5])
w = np.array([1, 0, 2, -1])
w0 = -3

# Dot product check
dot_product = np.dot(w, x) + w0
print("w^T x + w0 =", dot_product)

# Check if point x lies on the hyperplane
if np.isclose(dot_product, 0):
```

```
    print("Point x lies on the hyperplane.")
else:
    print("Point x does NOT lie on the hyperplane.")

# Showing perpendicularity concept
angle = np.arccos(np.dot(w, x) / (np.linalg.norm(w) * np.linalg.norm(x)))
print("Angle between w and x (radians):", angle)
print("Angle in degrees:", np.degrees(angle))
```

```
w^T x + w0 = -8
Point x does NOT lie on the hyperplane.
Angle between w and x (radians): 1.9037757275172553
Angle in degrees: 109.07831432618656
```



Geometric Interpretation of `w` Vector

> **The vector `w` is always perpendicular (orthogonal) to the hyperplane.**

What is a Hyperplane?

A **hyperplane** in an n-dimensional space is a decision boundary that can be mathematically expressed as:

$$w_1x_1 + w_2x_2 + \cdots + w_nx_n + w_0 = 0$$

Or, using vector notation:

$$\vec{w}^T \cdot \vec{x} + w_0 = 0$$

Where:

- (\vec{w} = [w_1, w_2, ..., w_n]) is the **normal vector** to the hyperplane.
- (\vec{x} = [x_1, x_2, ..., x_n]) is a **point** on the hyperplane.
- (w_0) is the **bias** (intercept term).

📐 Orthogonality

The **dot product** between the normal vector `w` and any point `x` on the hyperplane satisfies:

$$\vec{w} \cdot \vec{x} = -w_0$$

This means the angle between `w` and the surface of the hyperplane is **90°**, hence:

$$\text{w vector is always perpendicular to the hyperplane}$$

📊 Geometrical Intuition by Dimensions:

Dimension	Equation	Geometry	Normal Vector `w`
2D	(w_1x + w_2y + w_0 = 0)	Line	Perpendicular to line
3D	(w_1x + w_2y + w_3z + w_0 = 0)	Plane	Perpendicular to plane
nD	(\vec{w}^T \cdot \vec{x} + w_0 = 0)	Hyperplane	Perpendicular to hyperplane

---

🧠 Visual Summary

- `w` points in the direction of **maximum change**.
- The hyperplane lies in a direction **orthogonal** to `w`.
- In **Support Vector Machines (SVM)**, `w` helps to define the **decision boundary**.

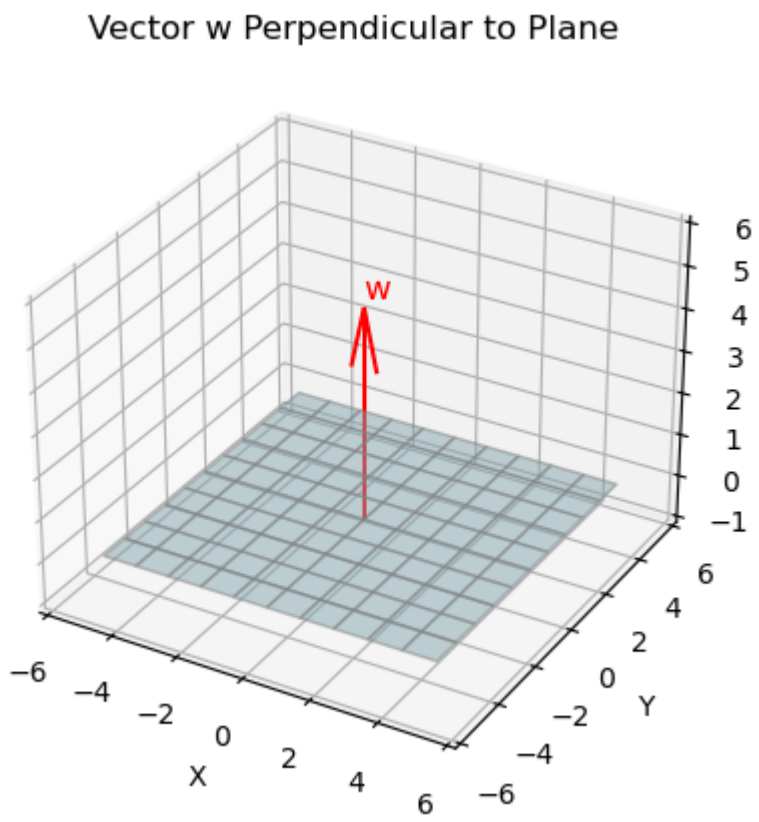
✍️ Code (Optional Plotting in 2D)

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

# Create x values
x = np.linspace(-10, 10, 100)
m = 1  # slope
b = 2  # intercept

# Line equation y = mx + b
```

```
y = m * x + b

# Plot the line
plt.plot(x, y, label='Line: y = mx + b', color='blue')

# w vector (normal to the line)
origin = np.array([[0, 0]])
w = np.array([[1], [1]])  # perpendicular to the line (for m=1)

# Plotting w vector
plt.quiver(*origin, *w, angles='xy', scale_units='xy', scale=1,
color='red', label='w (normal vector)')

# Decorations
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.grid(True)
plt.legend()
plt.title("w Vector Perpendicular to Line (2D)")
plt.xlabel("x-axis")
plt.ylabel("y-axis")
plt.axis('equal')
plt.show()
```

In [12]:

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Define plane (XY-plane for simplicity)
xx, yy = np.meshgrid(range(-5, 6), range(-5, 6))
zz = np.zeros_like(xx)  # Z=0 plane

# Plot the plane
ax.plot_surface(xx, yy, zz, alpha=0.5, color='lightblue', edgecolor='gray')

# Define the perpendicular vector w (pointing out of the plane)
origin = [0, 0, 0]
w = [0, 0, 1]  # Perpendicular to XY plane (Z-axis)

# Plot the vector
ax.quiver(*origin, *w, length=5, color='red')
ax.text(0, 0, 5.2, 'w', color='red', fontsize=12)

# Set labels and limits
ax.set_xlim([-6, 6])
ax.set_ylim([-6, 6])
ax.set_zlim([-1, 6])
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')

plt.title('Vector w Perpendicular to Plane')
plt.show()
```



🧠 Understanding Hyperplanes and Normal Vectors

📌 General Equation of a Hyperplane

A hyperplane in **n-dimensional space** is defined as:

$$\vec{w}^T \vec{x} + w_0 = 0$$

Where:

- (\vec{w}) is the **normal vector** (perpendicular to the hyperplane)
- (\vec{x}) is a point on the hyperplane
- (w_0) is the **bias term**

🎯 Important Properties

- The vector (\vec{w}) is **always perpendicular** to the hyperplane.
- If two points (\vec{x}_1) and (\vec{x}_2) lie on the hyperplane, then:

$$\vec{w}^T(\vec{x}_2 - \vec{x}_1) = 0$$

Which means the difference vector lies **within** the plane, and (\vec{w}) is orthogonal to it.

🔁 Alternate Forms of the Equation

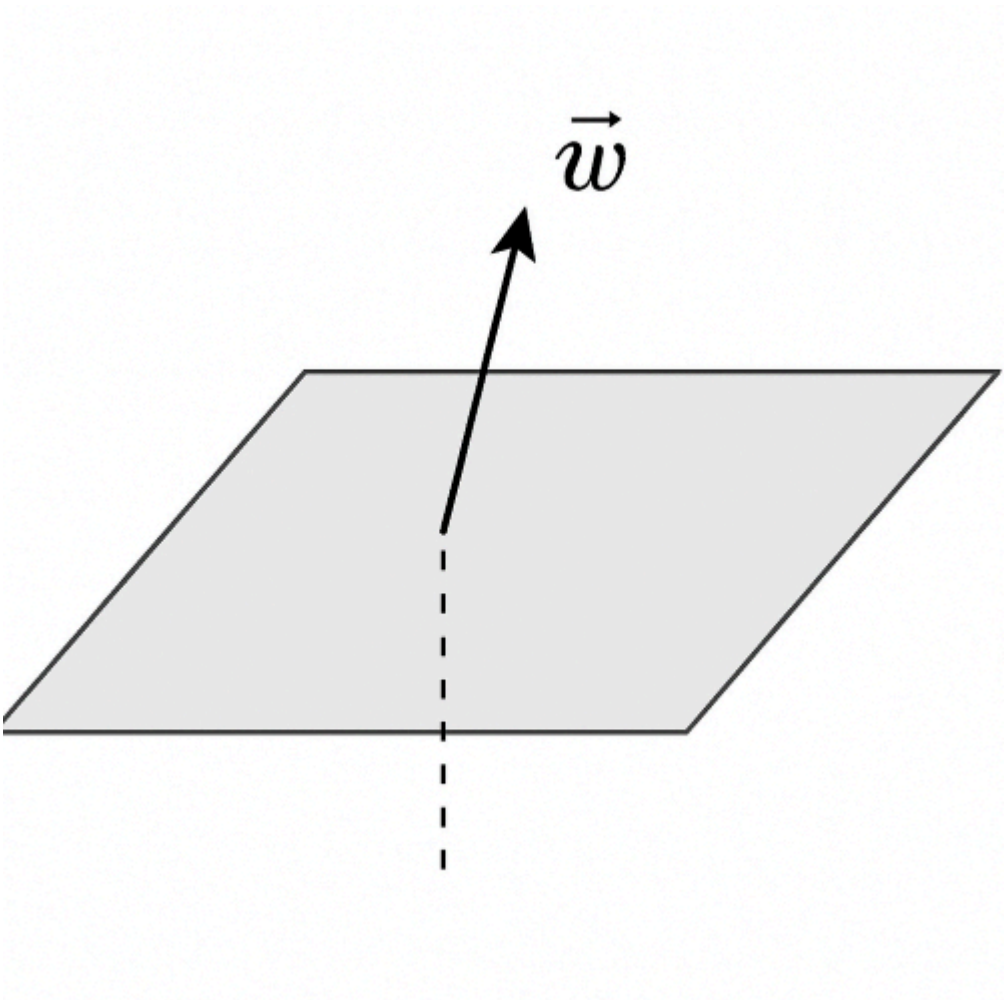
- (\vec{w}^T \vec{x} + w_0 = 0)
- (\vec{w}^T \vec{x} = -w_0)
- If the plane passes through the origin:

$$\vec{w}^T \vec{x} = 0$$

🧭 Geometric Intuition by Dimensions

Dimensions	What It Represents	Hyperplane Shape	Normal Vector
2D	Line	1D line	Perpendicular vector
3D	Plane	2D surface	Vector sticking out
4D+	Higher Dimensions	3D+ slices	Still orthogonal

🖼️ Visual



Here, the arrow represents (\vec{w}), which is perpendicular to the surface (hyperplane).

🧮 Matrices in Linear Algebra

📌 What is a Matrix?

A **matrix** is a rectangular array of numbers arranged in rows and columns.

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

- (m) = number of rows
- (n) = number of columns
- Matrix (A) is called an **(m \times n)** matrix

📏 Types of Matrices

- **Row Matrix**: 1 row, (n) columns
- **Column Matrix**: (m) rows, 1 column
- **Square Matrix**: (m = n)
- **Diagonal Matrix**: Non-zero only on the diagonal
- **Identity Matrix** (I): Diagonal elements = 1, others = 0
- **Zero Matrix**: All elements are 0
- **Symmetric Matrix**: (A = A^T)

🔁 Matrix Operations

➕ Addition / Subtraction

Only valid for same dimension matrices:

$$A + B = [a_{ij} + b_{ij}]$$

✖️ Scalar Multiplication

$$kA = [ka_{ij}]$$

🔢 Matrix Multiplication

Let (A) be of shape (m \times n) and (B) be (n \times p):

$$C = AB \quad \text{is of shape } m \times p$$

Each element (c_{ij}) is:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

🔄 Transpose of a Matrix

$$A^T = \text{flip rows and columns}$$

If:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad \Rightarrow \quad A^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

🧮 Determinant and Inverse (for square matrices)

✅ Determinant:

Defined only for square matrices. For a 2×2 matrix:

$$\det(A) = a_{11}a_{22} - a_{12}a_{21}$$

♻️ Inverse:

Matrix (A^{-1}) such that:

$$AA^{-1} = A^{-1}A = I$$

Only exists if (\text{det}(A) \neq 0)

🧠 Applications

- Solving systems of linear equations
- Computer graphics
- Machine learning (e.g., neural networks)
- Data transformation
- Eigenvalues/eigenvectors

Matrix Examples in Code

In [13]:
```
import numpy as np

# Identity Matrix (3x3)
I = np.identity(3)
```

```
# Zero Matrix (3x3)
Z = np.zeros((3, 3))

# Diagonal Matrix
D = np.diag([4, 5, 6])

# Random Matrix
A = np.array([[1, 2], [3, 4]])

print("Identity Matrix:\n", I)
print("Zero Matrix:\n", Z)
print("Diagonal Matrix:\n", D)
```

```
Identity Matrix:
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
Zero Matrix:
 [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
Diagonal Matrix:
 [[4 0 0]
 [0 5 0]
 [0 0 6]]
```

Matrix Operations

Matrix Addition

In [16]:
```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
C = A + B
C
```

Out[16]:
```
array([[ 6,  8],
       [10, 12]])
```

Matrix Multiplication

In [17]:
```
A = np.array([[1, 2], [3, 4]])
B = np.array([[2, 0], [1, 2]])
result = np.dot(A, B)
result
```

Out[17]:
```
array([[ 4,  4],
       [10,  8]])
```

Matrix Transpose

In [19]:
```
A = np.array([[1, 2], [3, 4]])
print(A)
A_T = A.T
A_T
```

```
[[1 2]
 [3 4]]
```

Out[19]:
```
array([[1, 3],
       [2, 4]])
```

Matrix Inverse (only for square, non-singular matrices)

In [20]:
```
from numpy.linalg import inv

A = np.array([[4, 7], [2, 6]])
A_inv = inv(A)
A_inv
```

Out[20]:
```
array([[ 0.6, -0.7],
       [-0.2,  0.4]])
```

➕ Additive Identity & Additive Inverse in Matrices

📌 Additive Identity

The **additive identity** of a matrix is a matrix that, when **added** to any matrix (A), gives back the **same matrix** (A).

✅ Formula:

$$A + O = A$$

Where:

- (A) is any (m \times n) matrix
- (O) is the **zero matrix** of the same size as (A)

📘 Example:

Let:

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

Then:

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

✅ Hence, (O) is the **additive identity** for matrix (A).

🔁 Additive Inverse

The **additive inverse** of a matrix (A) is another matrix (-A) such that their sum is the **zero matrix**.

✅ Formula:

$$A + (-A) = O$$

Where:

- (-A) is obtained by **negating** each element of (A)
- (O) is the zero matrix of the same size

📘 Example:

Let:

$$A = \begin{bmatrix} 5 & -3 \\ 2 & 0 \end{bmatrix} \Rightarrow -A = \begin{bmatrix} -5 & 3 \\ -2 & 0 \end{bmatrix}$$

Then:

$$A + (-A) = \begin{bmatrix} 5 & -3 \\ 2 & 0 \end{bmatrix} + \begin{bmatrix} -5 & 3 \\ -2 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

✅ Hence, (-A) is the **additive inverse** of matrix (A).

🧠 Summary

Concept	Formula	Result
Additive Identity	(A + O = A)	Original matrix
Additive Inverse	(A + (-A) = O)	Zero matrix

Code example

In [21]:
```
import numpy as np

# Define matrix A
A = np.array([[5, -3],
              [2,  0]])

# Define additive identity (zero matrix of same shape)
O = np.zeros_like(A)

# Compute additive inverse
neg_A = -A

# Check additive identity: A + O
add_identity = A + O
```

```
# Check additive inverse: A + (-A)
add_inverse = A + neg_A

print("Matrix A:\n", A)
print("\nAdditive Identity (Zero Matrix):\n", O)
print("\nA + O:\n", add_identity)
print("\nAdditive Inverse (-A):\n", neg_A)
print("\nA + (-A):\n", add_inverse)
```

```
Matrix A:
 [[ 5 -3]
 [ 2  0]]

Additive Identity (Zero Matrix):
 [[0 0]
 [0 0]]

A + O:
 [[ 5 -3]
 [ 2  0]]

Additive Inverse (-A):
 [[-5  3]
 [-2  0]]

A + (-A):
 [[0 0]
 [0 0]]
```

Matrix Operations

Matrix Inverse

The inverse of a square matrix (A) is given by:

$$A^{-1} = \frac{1}{|A|} \cdot \text{Adj}(A)$$

> Only defined when (|A| \neq 0) (non-singular matrix).

Matrix Multiplication

Let:

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}_{2\times2}, \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}_{2\times2}$$

Then:

$$A \times B = C \quad \text{where each } c_{ij} = \sum_k a_{ik} \cdot b_{kj}$$

✅ Rule:

- Matrix **A's columns** must equal Matrix **B's rows**
- Resulting matrix shape: **rows of A × columns of B**

📘 Example:

Let:

- (A \rightarrow 2 \times 3)
- (B \rightarrow 3 \times 2)

✔️ Multiplication possible.
🧻 Resulting matrix: (2 \times 2)

🔄 Transpose of a Matrix

✅ Rules:

$$(A^T)^T = A$$

$$(A + B)^T = A^T + B^T$$

$$(AB)^T = B^T \cdot A^T$$

🔍 Special Notes on Dimensions

- If **rows > columns** → **Tall Matrix** (e.g., (3 \times 2)):
  🔁 Goes from **lower to higher dimension**
- If **columns > rows** → **Wide Matrix** (e.g., (3 \times 4)):
  🔁 Goes from **higher to lower dimension**

🚫 Matrix Inverse Restrictions

- ❌ **Inverse not possible for non-square matrices**
- ❌ **Inverse not possible for singular matrices** (when determinant = 0)

🔄 Solving System of Equations

Given:

$$AX = B$$

If (A) is invertible:

$$X = A^{-1}B$$

Alternate (Moore-Penrose pseudo-inverse):

$$X = A^{+}B$$

1.1 Introduction to Machine Learning

What is Machine Learning?

Machine Learning is a branch of computer science where statistical techniques are used to find patterns in data.
It allows systems to learn from data and make predictions or decisions without being explicitly programmed.

Types of Machine Learning

1. **Supervised Learning**
   - The model learns from labeled data.
   - Types:
     - **Regression**: Predicting continuous values
       Example: House Price Prediction
     - **Classification**: Predicting categories or labels
       Example: Spam Detection
2. **Unsupervised Learning**
   - The model learns from unlabeled data.
   - Types:
     - **Clustering**: Grouping similar data points
       Example: Customer Segmentation
     - **Dimensionality Reduction**: Reducing features
       Example: PCA (Principal Component Analysis)
     - **Anomaly Detection**: Identifying outliers
       Example: Fraud Detection
     - **Association Rule Learning**: Discovering rules in data
       Example: Market Basket Analysis
3. **Semi-Supervised Learning**
   - The model learns from a mix of labeled and unlabeled data.
4. **Reinforcement Learning**
   - The model learns by interacting with the environment and receiving feedback in the form of rewards or penalties.

Types of Data

- **Numerical Data**
  - Often used in **Regression** tasks

  Target data is numerical (e.g., price, temperature)
- **Categorical Data**
  - Used in **Classification** tasks

  Target data is categorical (e.g., labels like spam/ham)

How Machine Learning Models Learn

There are two primary ways models learn:

1. **Memorizing**
   - Also known as **Instance-Based Learning**
   - The model memorizes the training data to make predictions

   Example: k-NN (k-Nearest Neighbors)
2. **Generalizing**
   - Known as **Model-Based Learning**
   - The model learns the underlying principle or pattern in the data

   Example: Linear Regression, Decision Trees

Instance-Based vs Model-Based Learning

Approach	Description
Instance-Based	Memorizes the training data and compares new data to it
Model-Based	Extracts the fundamental patterns and builds a predictive model

---

Sampling Concepts

Sampling Noise

- Random variability in the sample due to chance.
- Occurs because we're using a subset (sample) of the full population.

Sampling Bias

- Systematic error that occurs when the sample is **not representative** of the population.

- Can lead to misleading conclusions.

> **In summary**:

- **Sampling Noise** = Random error from selecting a sample
- **Sampling Bias** = Systematic error due to poor sampling strategy

---

Who is a Data Scientist?

> *"A Data Scientist is someone who is better at statistics than any software engineer, and better at software engineering than any statistician."*

A **Data Scientist** blends the skills of:

- **Statistics**: Understanding data distributions, trends, and patterns
- **Programming**: Writing efficient and scalable code
- **Domain Knowledge**: Understanding the business and data context

---

In [ ]: