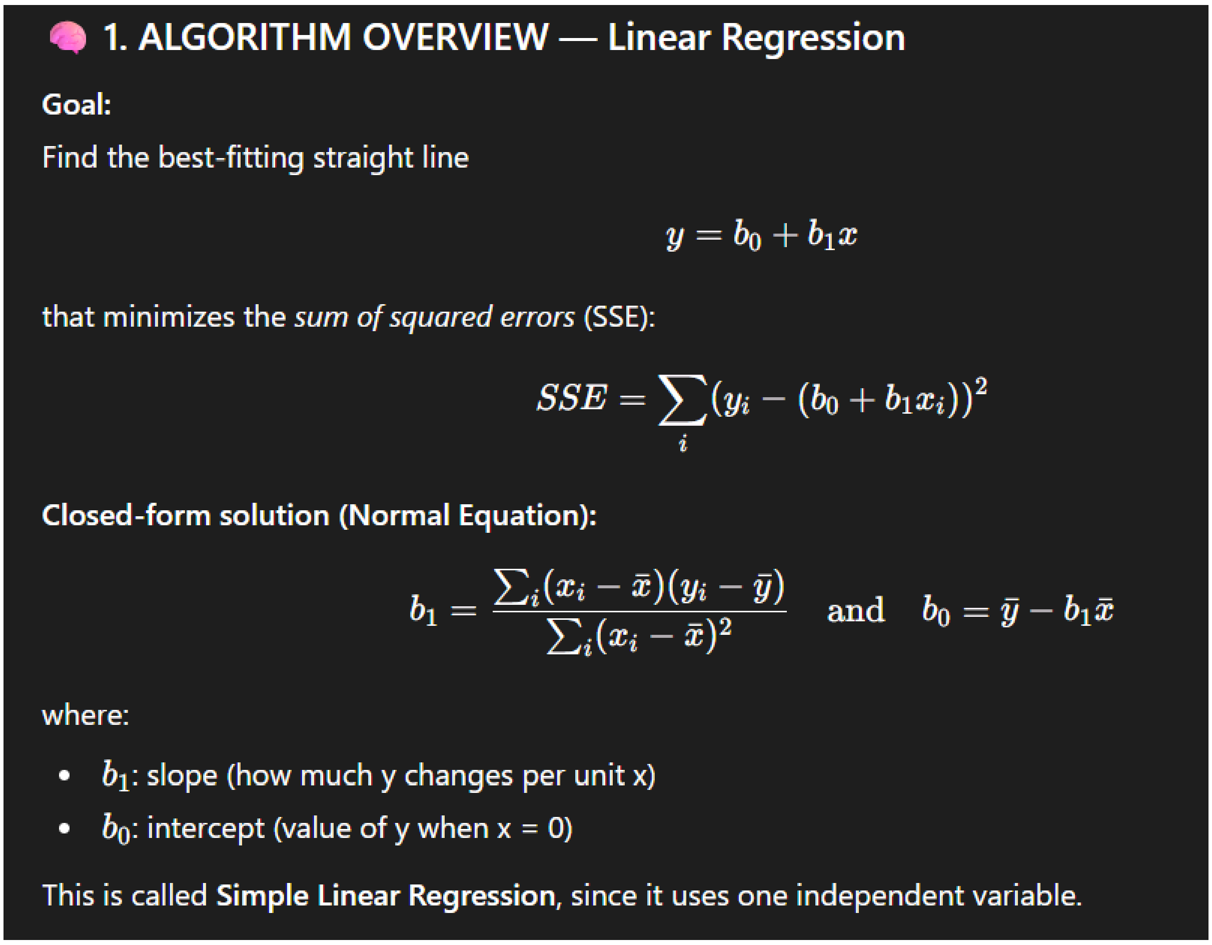
DAY 2 : LINEAR REGRESSION



🧠 1. ALGORITHM OVERVIEW — Linear Regression

Goal:

Find the best-fitting straight line

$$y = b_0 + b_1 x$$

that minimizes the *sum of squared errors* (SSE):

$$SSE = \sum_i (y_i - (b_0 + b_1 x_i))^2$$

Closed-form solution (Normal Equation):

$$b_1 = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sum_i (x_i - \bar{x})^2} \quad \text{and} \quad b_0 = \bar{y} - b_1 \bar{x}$$

where:

- b_1: slope (how much y changes per unit x)
- b_0: intercept (value of y when x = 0)

This is called **Simple Linear Regression**, since it uses one independent variable.

⚙️ 2. STEP-BY-STEP CODE EXPLANATION

🔹 Imports

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

- `numpy`: used for numerical operations, array handling, and math (mean, sum, etc.)
- `matplotlib.pyplot`: used for plotting data and the regression line

🔹 Function: `coeff(x, y)`

This function computes the slope and intercept of the regression line.

```
def coeff(x, y):
    n = np.size(x)
```

- n: number of data points (length of x or y)

```
mx = np.mean(x)
 my = np.mean(y)
```

- mx, my: means (averages) of x and y respectively.

```
ss_xy = np.sum(x * y) - n * mx * my  # covariance
```

- ss_xy: numerator of slope formula.
  It represents covariance between x and y, scaled by n.

```
ss_xx = np.sum(x * x) - n * mx * mx  # variance
```

- ss_xx: denominator of slope formula.
  It's like variance of x, scaled by n.

```
b1 = (ss_xy / ss_xx)
 b0 = my - (b1 * mx)
 return (b0 , b1)
```

- Computes slope (b1) and intercept (b0)
- Returns both as a tuple

🧩 Mathematical equivalent:

$$b_1 = \frac{\mathrm{Cov}(x, y)}{\mathrm{Var}(x)}, \quad b_0 = \bar{y} - b_1\bar{x}$$

🔹 Function: `lr_plot(p, q, b)`

Plots the data points and the regression line.

```
def lr_plot(p, q, b):
    plt.scatter(p, q, color='r', marker='s', s=50)
```

- Plots original data points (x, y) as red squares.

```
y_out = (b[0] + b[1] * p)
```

- Predicts y-values using regression coefficients.

```
plt.plot(p, y_out, color='g')
```

- Draws the regression line (green).

```
plt.xlabel("x axis")
 plt.ylabel("y axis")
 plt.show()
```

- Adds labels and displays the plot.

🔹 Function: `main()`

Runs the linear regression example.

```
def main():
    x = np.array([0,1,2,3,4,5,6,7,8,9])
    y = np.array([1,3,2,5,7,8,8,9,10,12])
```

- Sample data arrays for x and y.

```
b = coeff(x, y)
```

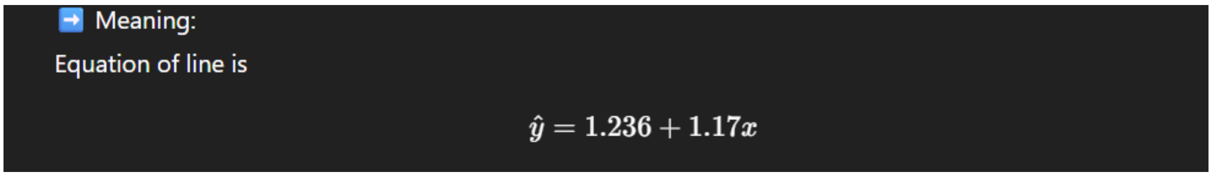
- Calls `coeff()` to get `(b0, b1)`

```
print("b0 = {} and b1 = {}".format(b[0], b[1]))
```

- Prints computed coefficients.

Output:

```
b0 = 1.2363636363636363 and b1 = 1.1696969696969697
```


➡️ Meaning:
Equation of line is

$$\hat{y} = 1.236 + 1.17x$$

➡️ **Meaning:**
Equation of line is

```
   lr_plot(x , y , b)
```

- **Visualizes data points and regression line.**

```
if __name__ == "__main__":
    main()
```

- **Standard Python idiom to run main() only when script is executed directly.**

📈 OUTPUT BEHAVIOR

- **Scatter Plot: red squares for actual data points**
- **Line: green line showing best fit**

🧮 SIGMOID CURVE PART

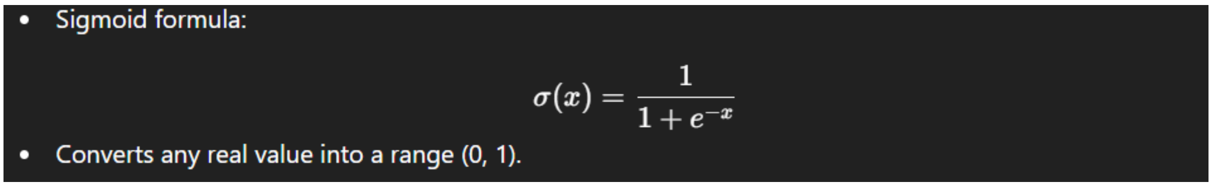
Used mainly in logistic regression and neural networks.

🔹 Code Breakdown

```
import numpy as np

import matplotlib.pyplot as plt

x = np.linspace(-10,10,100)
```

- **Creates 100 equally spaced points between -10 and 10.**

```
z = (1 / (1 + np.exp(-x)))
```

- 

  - Sigmoid formula:

  $$\sigma(x) = \frac{1}{1 + e^{-x}}$$

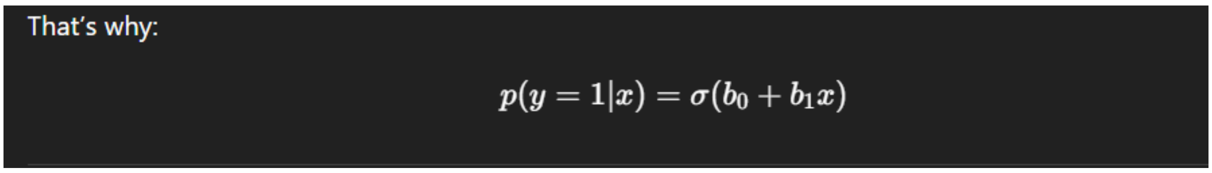
  - Converts any real value into a range (0, 1).

```
plt.plot(x, z)

plt.show()
```

- **Plots the S-shaped sigmoid curve.**

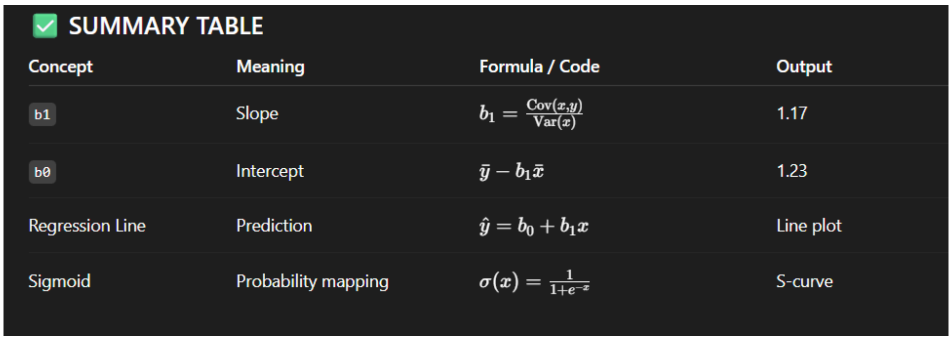
🧩 RELATION BETWEEN LINEAR REGRESSION & SIGMOID

- **Linear Regression outputs continuous values.**
- **Logistic Regression applies sigmoid to a linear model to output probabilities (0–1).**



That's why:

$$p(y = 1|x) = \sigma(b_0 + b_1 x)$$



✅ SUMMARY TABLE

Concept	Meaning	Formula / Code	Output
`b1`	Slope	$b_1 = \frac{\text{Cov}(x,y)}{\text{Var}(x)}$	1.17
`b0`	Intercept	$\bar{y} - b_1\bar{x}$	1.23
Regression Line	Prediction	$\hat{y} = b_0 + b_1 x$	Line plot
Sigmoid	Probability mapping	$\sigma(x) = \frac{1}{1+e^{-x}}$	S-curve

DAY 3 : LOGISTIC REGRESSION

🧠 PART 1 — ALGORITHM & CONCEPTS

🔹 What's Happening Here

You've written code that:

1. **First computes the coefficients `b0` and `b1` using the linear regression formula (still the same as before).**
2. **Then applies the sigmoid (logistic) transformation to map linear outputs to probabilities between 0 and 1.**
3. **Finally, you plot this using Matplotlib and Seaborn to visualize how the logistic curve fits the data.**

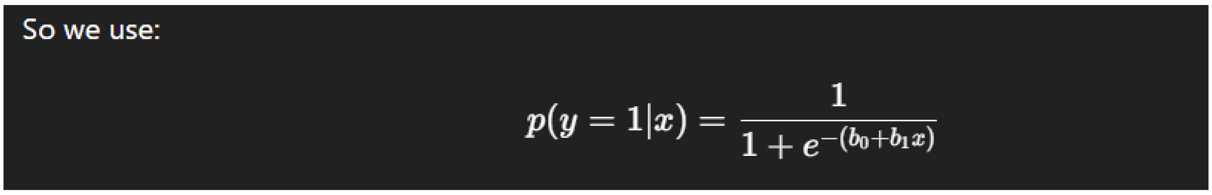
Then you also validated it by using:

```
from sklearn import linear_model

logr = linear_model.LogisticRegression()
```

which fits the same kind of model properly using numerical optimization (maximum likelihood).

🔹 Why Logistic Regression (not Linear Regression)

In linear regression, output yyy can be any real number.
But in classification (like predicting 0 or 1), we need a probability output in [0,1][0,1][0,1].



So we use:

$$p(y=1|x) = \frac{1}{1+e^{-(b_0+b_1x)}}$$

This is the sigmoid (logistic) function, which "squashes" real numbers into (0,1).

The model predicts probabilities, not just labels.

⚙️ PART 2 — CODE WALKTHROUGH

Let’s analyze each section step-by-step.

🧩 SECTION 1 — Manual Logistic Regression Visualization

Imports

```
import matplotlib.pyplot as plt

import numpy as np
```

- `matplotlib.pyplot`: **for plotting**
- `numpy`: **for arrays and numerical math**

Coefficient Calculation

```
def coeff(x, y):

    n = np.size(x)

    mx = np.mean(x)

    my = np.mean(y)

    ss_xy = np.sum(x * y) - n * mx * my

    ss_xx = np.sum(x * x) - n * mx * mx

    b1 = ss_xy / ss_xx

    b0 = my - b1 * mx

    return (b0, b1)
```

✅ **What it does:**

- Same as before — computes coefficients `b0` and `b1` as if it were linear regression.
- These act as *initial approximations* of the logistic regression parameters.

But note — *true logistic regression coefficients* are found using maximum likelihood, not this linear shortcut — this is only a visual approximation.

Logistic Curve Plotting

```
def lr_plot(p, q, b):

    plt.scatter(p, q, color='r', marker="s", s=30)
```

- Red scatter for actual data (x,y)

```
y_out = b[0] + b[1] * p
```

- Linear combination of features.

```
lr = 1 / (1 + np.exp(-y_out))
```

- The logistic (sigmoid) transformation:
  Converts linear predictions → probability between 0–1.

```
plt.plot(p, lr, color="b")

 plt.xlabel("x")

 plt.ylabel("y")

 plt.show()
```

- Plots the blue sigmoid curve on top of red data points.

Main Function

```
def main():

    x = np.array([3.78 , 2.44 , 2.09 , 0.14 ,1.72 , 1.65 , 4.92
,4.37 , 4.52 , 3.69 , 5.89]).reshape([-1 , 1])

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

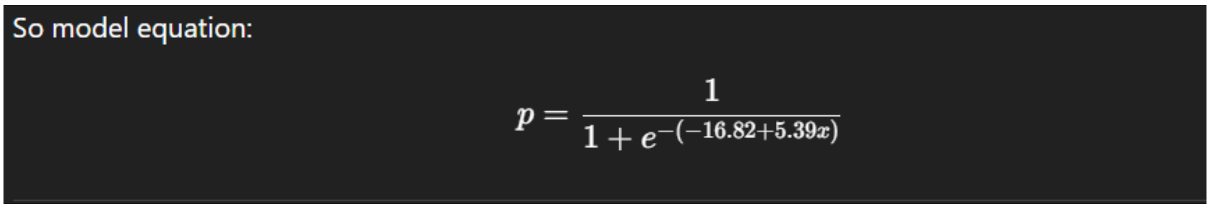
- **Dataset: small binary classification (0 or 1)**
- `.reshape([-1,1])` **makes x a column vector**

```
   b = coeff(x , y)

    print("b0 & b1 values are : \n  b0 = {} \n b1 =
{}".format(b[0],b[1]))

    lr_plot(x,y,b)
```

- **Compute coefficients**
- **Print and plot them**

Output:

```
b0 = -16.82

b1 = 5.39
```



So model equation:

$$p = \frac{1}{1 + e^{-(-16.82+5.39x)}}$$

🧩 SECTION 2 — Using Scikit-Learn Logistic Regression

```
from sklearn import linear_model



x = np.array([...]).reshape([-1, 1])
```

```
y = np.array([...])

logr = linear_model.LogisticRegression()

logr.fit(x, y)

predict = logr.predict(np.array([3.46]).reshape(-1, 1))

print(predict)
```

✅ **What happens here:**

- `logr.fit(x,y)` **— uses Maximum Likelihood Estimation to find optimal coefficients.**
- **Internally uses Gradient Descent / Newton-Raphson (depending on solver).**
- `predict()` **— predicts class label (0 or 1) for input** `x=3.46`**.**

Output:

```
[0]
```

→ Meaning: at x=3.46, probability < 0.5, so class = 0.

🧩 SECTION 3 — New Dataset with Ages & Insurance

Now, you reused the same concept with different data.

Code Overview

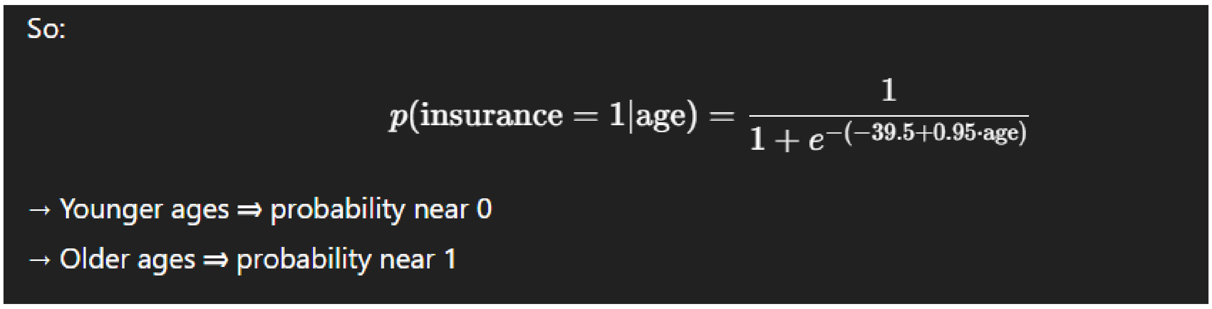
```
x =
np.array([22,25,47,52,46,56,55,60,62,61,18,28,27,29]).reshape([-1 ,
1])

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

- **Predict whether a person has insurance (1) or not (0) based on age.**

Output coefficients:

```
b0 = -39.4994
```

```
b1 = 0.9506
```



So:

$$p(\text{insurance} = 1|\text{age}) = \frac{1}{1 + e^{-(-39.5+0.95\cdot\text{age})}}$$

→ Younger ages ⇒ probability near 0

→ Older ages ⇒ probability near 1

🧩 SECTION 4 — Seaborn Visualization

```
sns.regplot(x='age', y='insurance', data=df, logistic=True,
marker='o', color='r')
```

- `logistic=True` **makes seaborn fit a logistic regression curve automatically.**
- **Red dots for data; blue curve (internally computed sigmoid) over it.**

⚠️ WARNING EXPLAINED

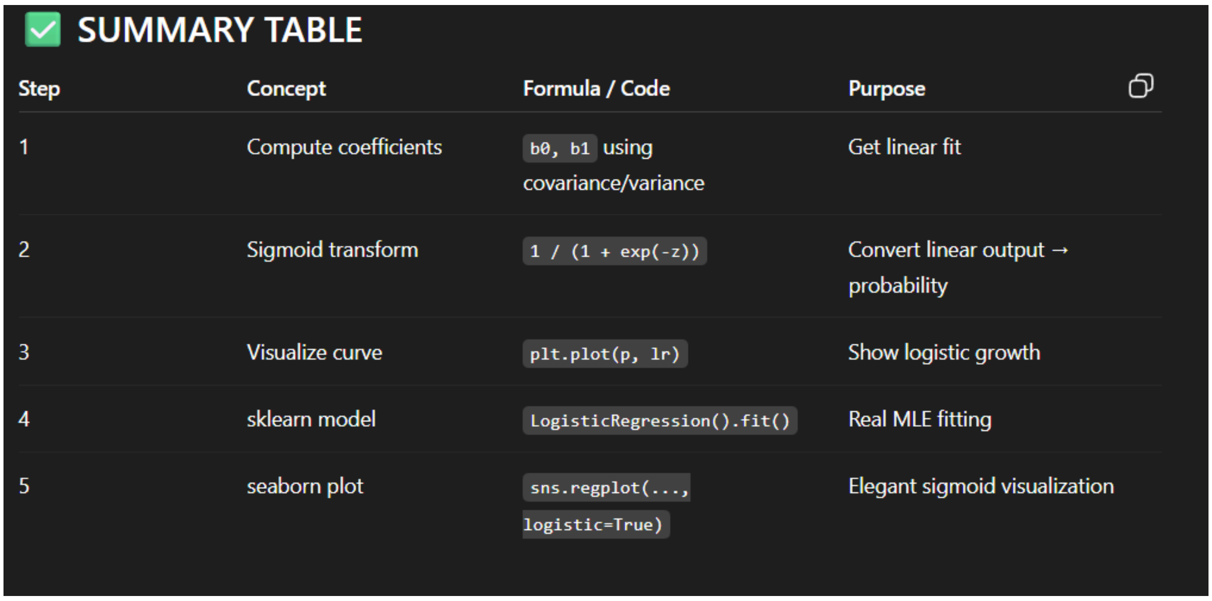
```
RuntimeWarning: overflow encountered in exp

  t = np.exp(-z)
```

💡 **Reason:**
When the value of `z = b0 + b1*x` **becomes too large (positive or negative),** `np.exp(-z)` **overflows because exponential grows very fast.**

✅ **Fix:**
Use numerical stabilization:

```
lr = 1 / (1 + np.exp(-np.clip(y_out, -250, 250)))
```

This prevents overflow in `exp()` **by limiting values of** `y_out`**.**


✅ SUMMARY TABLE

Step	Concept	Formula / Code	Purpose
1	Compute coefficients	`b0, b1` using covariance/variance	Get linear fit
2	Sigmoid transform	`1 / (1 + exp(-z))`	Convert linear output → probability
3	Visualize curve	`plt.plot(p, lr)`	Show logistic growth
4	sklearn model	`LogisticRegression().fit()`	Real MLE fitting
5	seaborn plot	`sns.regplot(..., logistic=True)`	Elegant sigmoid visualization

DAY 3 : K-Nearest Neighbors (KNN)

🧠 1. THE ALGORITHM — K-Nearest Neighbors (KNN)

🎯 Goal

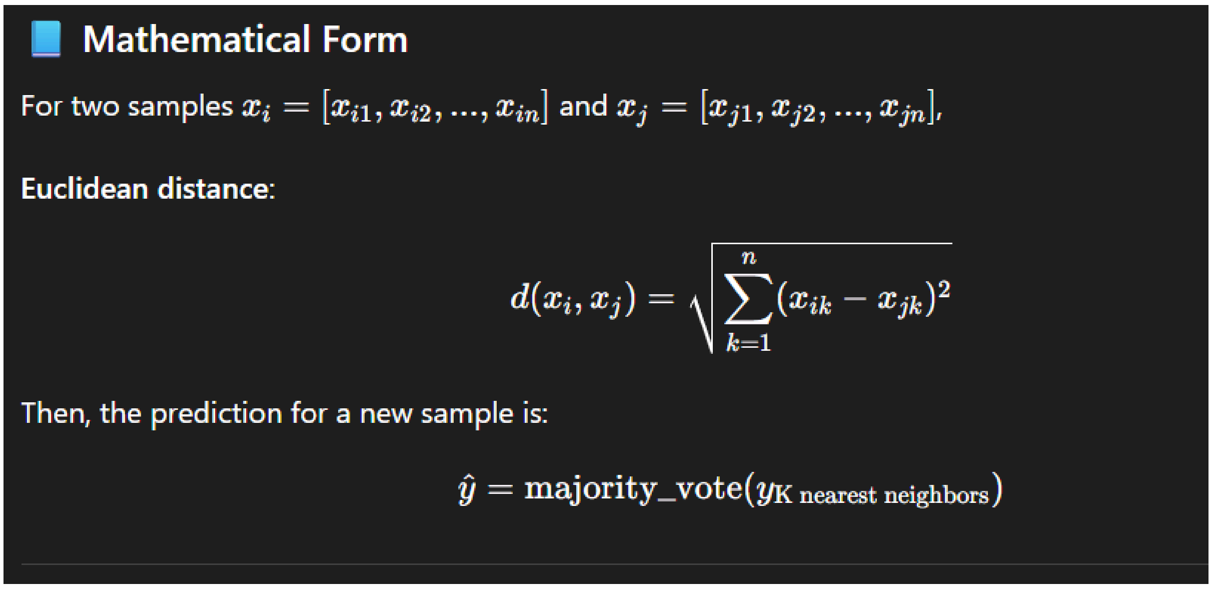
Given a new input sample, predict its class label based on the majority class of its K nearest neighbors in the training data.

Steps:

1️⃣Store all training examples in memory.
2️⃣For a new point `x_test`:

- **Compute distance (usually Euclidean) to every training sample.**
- **Select the K smallest distances.**
- **Assign `x_test` the most common label among those K samples.**

It's a non-parametric, instance-based learning method — meaning it doesn't build a model, just memorizes and compares.



📘 **Mathematical Form**

For two samples $x_i = [x_{i1}, x_{i2}, ..., x_{in}]$ and $x_j = [x_{j1}, x_{j2}, ..., x_{jn}]$,

Euclidean distance:

$$d(x_i, x_j) = \sqrt{\sum_{k=1}^{n}(x_{ik} - x_{jk})^2}$$

Then, the prediction for a new sample is:

$$\hat{y} = \text{majority\_vote}(y_{\text{K nearest neighbors}})$$

⚙️ 2. CODE EXPLANATION

Step 1 — Load Dataset

```
from sklearn import datasets

iris = datasets.load_iris()
```

- **Loads the Iris dataset, a small, labeled dataset with:**
  - **150 rows (samples)**
  - **4 features per sample**
  - **3 classes (Setosa, Versicolor, Virginica)**

Step 2 — Explore Data

```
x = iris.data

print(x)

y = iris.target

print(y)
```

- x: all feature values → sepal length, sepal width, petal length, petal width
- y: class labels → 0, 1, or 2 (corresponding to 3 flower species)

Step 3 — Feature & Target Names

```
fname = iris.feature_names

print(fname)

tname = iris.target_names

print(tname)
```

Output:

```
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
'petal width (cm)']

['setosa' 'versicolor' 'virginica']
```

✅ **This helps map feature columns to real-world meanings.**

Step 4 — Train a Simple KNN Model

```
from sklearn.neighbors import KNeighborsClassifier


knn = KNeighborsClassifier(n_neighbors = 10)

knn.fit(x, y)
```

- **Creates a KNN classifier using K=10 (looks at 10 nearest points).**
- **Fits the model (stores all points).**

```
pred = knn.predict([[6.6, 2.9, 4.6, 1.3]])
```

```
print(pred)
```

Output:

```
[1]
```

→ Class 1 = *Iris-versicolor* 🌸

✅ Meaning: the given sample's features are most similar to the versicolor cluster.

Step 5 — Create a Pandas DataFrame for Visualization

```
import pandas as pd

iris_data = pd.DataFrame(iris.data, columns=iris.feature_names)

print(iris_data.head(10))

print(iris_data.tail(10))
```

✅ This makes it easier to visualize and analyze — each column is a feature.

Step 6 — Pairplot Visualization

```
import seaborn as sns

iris_data['target'] = iris.target

sns.pairplot(data=iris_data, hue='target', kind='scatter',
diag_kind='hist', palette=['red','green','blue'])
```

- **Plots pairwise relationships between all 4 features.**
- **Each color corresponds to a species.**
- **The clear cluster separation visually confirms that KNN will perform very well.**

🧩 3. MODEL TRAINING AND ACCURACY TESTING

Corrected Imports

```
from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score
```

✅ You correctly fixed typos:

- ❌ `tarin_test_split` → ✅ `train_test_split`
- ❌ `sklearn.matrics` → ✅ `sklearn.metrics`

Split Data

```
x_train, x_test, y_train, y_test = train_test_split(x, y,
test_size=0.2, random_state=0)
```

- 80% of data for training
- 20% for testing
- Random seed fixed for reproducibility.

Initialize Lists for Accuracy

```
train_score_list = []

test_score_list = []

k_range = range(1, 16)
```

You're testing K values from 1 to 15 to see how accuracy changes.

Loop Over K Values

```
for k in k_range:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(x_train, y_train)

    y_pred_train = knn.predict(x_train)
    y_pred_test = knn.predict(x_test)
```

- Trains a KNN for each K
- Predicts both on training and test sets

Compute Accuracies

```
accuracy1 = accuracy_score(y_train, y_pred_train)
accuracy2 = accuracy_score(y_test, y_pred_test)
```

✅ `accuracy_score()` = (Correct Predictions / Total Samples)

Store & Print Results

```
train_score_list.append(accuracy1)
test_score_list.append(accuracy2)

print(f'k={k} Train Accuracy = {accuracy1*100:.2f}%')
print(f'k={k} Test Accuracy  = {accuracy2*100:.2f}%')
```

Output Interpretation

```
k=1 Train Accuracy = 100.00%

k=1 Test Accuracy  = 100.00%

k=2 Train Accuracy = 97.50%

k=2 Test Accuracy  = 96.67%

...

k=15 Train Accuracy = 95.83%

k=15 Test Accuracy  = 100.00%
```

🧩 **Observations:**

- **For small K (like 1–3), model memorizes training data → sometimes overfits.**
- **For larger K, accuracy stabilizes around 96–100% (good generalization).**
- **Iris dataset is very clean and separable, so KNN works nearly perfectly.**

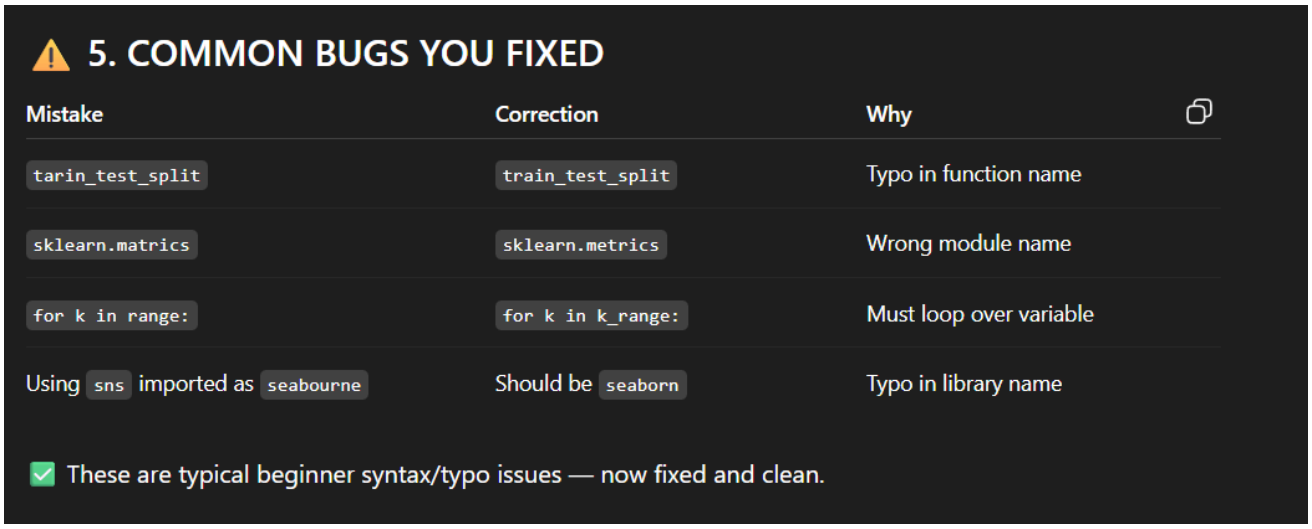
📈 4. VISUAL INSIGHT (optional to add)

You could visualize the accuracy curve:

```
import matplotlib.pyplot as plt


plt.plot(k_range, train_score_list, label='Train Accuracy')

plt.plot(k_range, test_score_list, label='Test Accuracy')

plt.xlabel('K Value')

plt.ylabel('Accuracy')

plt.legend()

plt.title('KNN Accuracy vs K')

plt.show()
```

✅ **Typically, you'll see:**

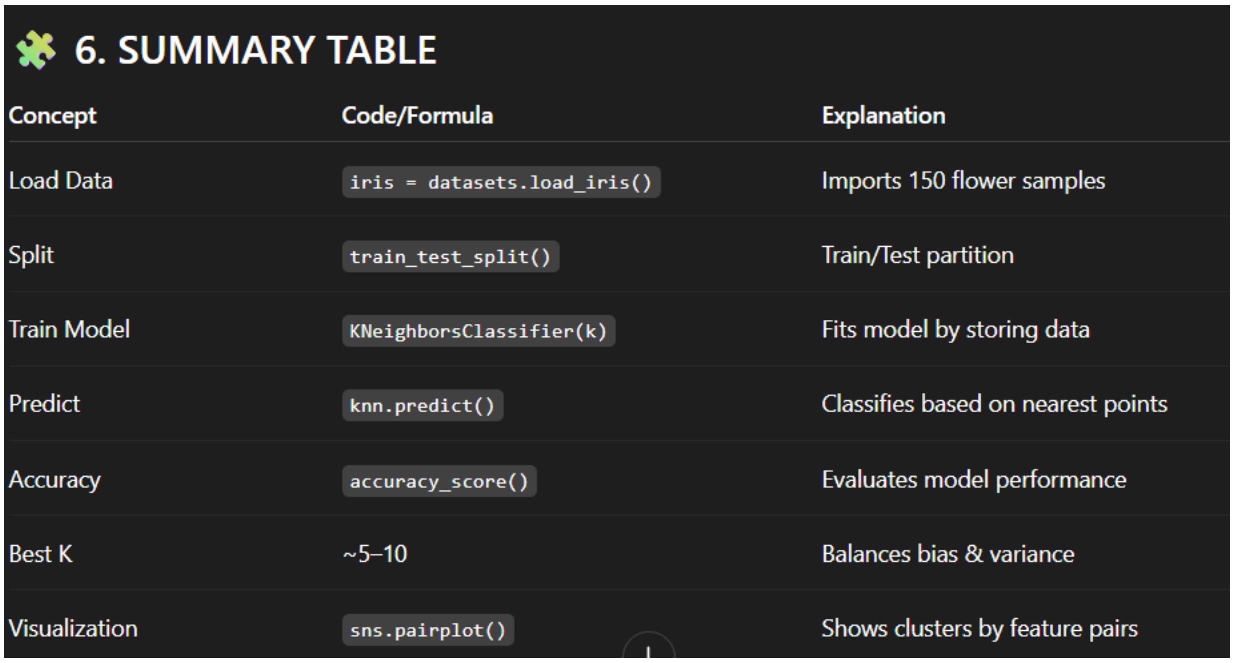
- **High accuracy at small K but overfitting,**
- **Slight dip at medium K but better generalization,**
- **Flatter accuracy for large K.**



⚠️ 5. COMMON BUGS YOU FIXED

Mistake	Correction	Why
`tarin_test_split`	`train_test_split`	Typo in function name
`sklearn.matrics`	`sklearn.metrics`	Wrong module name
`for k in range:`	`for k in k_range:`	Must loop over variable
Using `sns` imported as `seabourne`	Should be `seaborn`	Typo in library name

✅ These are typical beginner syntax/typo issues — now fixed and clean.



🧩 6. SUMMARY TABLE

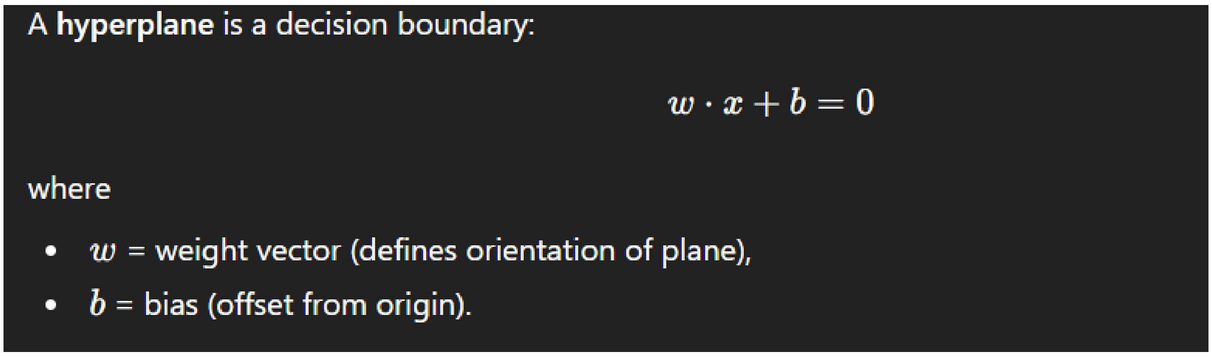
Concept	Code/Formula	Explanation
Load Data	`iris = datasets.load_iris()`	Imports 150 flower samples
Split	`train_test_split()`	Train/Test partition
Train Model	`KNeighborsClassifier(k)`	Fits model by storing data
Predict	`knn.predict()`	Classifies based on nearest points
Accuracy	`accuracy_score()`	Evaluates model performance
Best K	~5–10	Balances bias & variance
Visualization	`sns.pairplot()`	Shows clusters by feature pairs

DAY 5 : Support Vector Machine (SVM)

🧠 1. THEORY — Support Vector Machine (SVM)

🔹 Core Idea

SVM is a supervised classification algorithm that tries to find the best separating hyperplane between different classes.

- 

A **hyperplane** is a decision boundary:

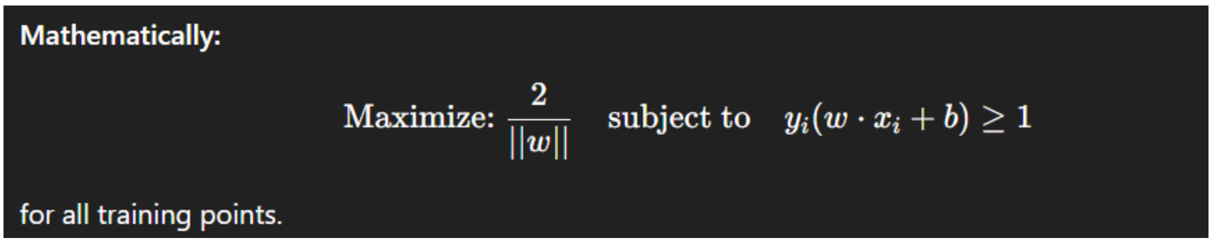
$$w \cdot x + b = 0$$

where

- w = weight vector (defines orientation of plane),
- b = bias (offset from origin).

🔹 Optimization Goal

SVM finds the hyperplane that maximizes the margin — i.e., the distance between the plane and the closest data points from each class (called support vectors).



Mathematically:

$$\text{Maximize: } \frac{2}{||w||} \quad \text{subject to} \quad y_i(w \cdot x_i + b) \geq 1$$

for all training points.

If data isn't linearly separable, we introduce slack variables and use the C parameter to control penalty for misclassifications.

🔹 Kernel Trick (not used here, but important)

When data isn't separable in the original space, SVM can project it into higher dimensions using kernels.

Common kernels:

- `'linear'` — straight line or plane
- `'poly'` — polynomial
- `'rbf'` — radial basis function (Gaussian)
- `'sigmoid'`

You used `'linear'`, which means it's a Linear SVM — similar to a linear classifier but with margin maximization.

⚙️ 2. STEP-BY-STEP CODE EXPLANATION

Imports

```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.svm import SVC

from sklearn.metrics import classification_report, confusion_matrix

from sklearn.preprocessing import StandardScaler
```

✅ Libraries:

- `numpy` — numerical ops
- `matplotlib` — optional visualization
- `sklearn` — ML tools
    - `SVC` — Support Vector Classifier

    - `StandardScaler` — feature scaling
    - `classification_report`, `confusion_matrix` — model evaluation

Load the Dataset

```
iris = load_iris()

x = iris.data

y = iris.target
```

- `x`: features (4 columns)
    - sepal length, sepal width, petal length, petal width
- `y`: labels (0 = Setosa, 1 = Versicolor, 2 = Virginica)

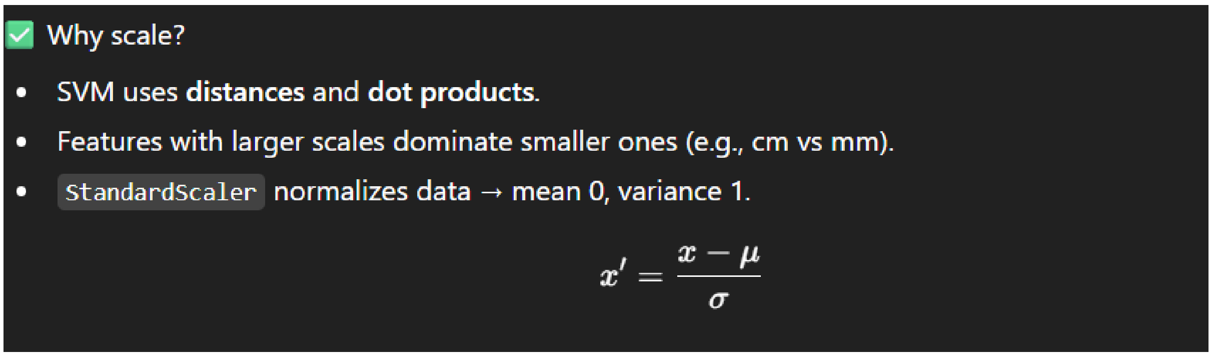
Train-Test Split

```
x_train, x_test, y_train, y_test = train_test_split(x, y,
test_size=0.3, random_state=42)
```

- 70% training, 30% testing
- `random_state=42` → reproducible split

Feature Scaling (💡 Critical for SVM)

```
scaler = StandardScaler()

x_train = scaler.fit_transform(x_train)

x_test = scaler.transform(x_test)
```


✅ Why scale?

- SVM uses **distances and dot products**.
- Features with larger scales dominate smaller ones (e.g., cm vs mm).
- `StandardScaler` normalizes data → mean 0, variance 1.

$$x' = \frac{x - \mu}{\sigma}$$

Train the Model

```
svm_model = SVC(kernel='linear', C=1)

svm_model.fit(x_train, y_train)
```

- `kernel='linear'` → **uses linear hyperplane**
- `C=1` → **regularization parameter:**
  - **Small `C`: wider margin but more misclassifications**
  - **Large `C`: narrow margin, less tolerance to errors (can overfit)**

✅ **After fitting, the SVM has found:**

- **A separating hyperplane for each pair of classes**
- **A set of support vectors (critical points on boundaries)**

Make Predictions

```
y_pred = svm_model.predict(x_test)
```

- **Predicts class labels (0,1,2) for test data.**

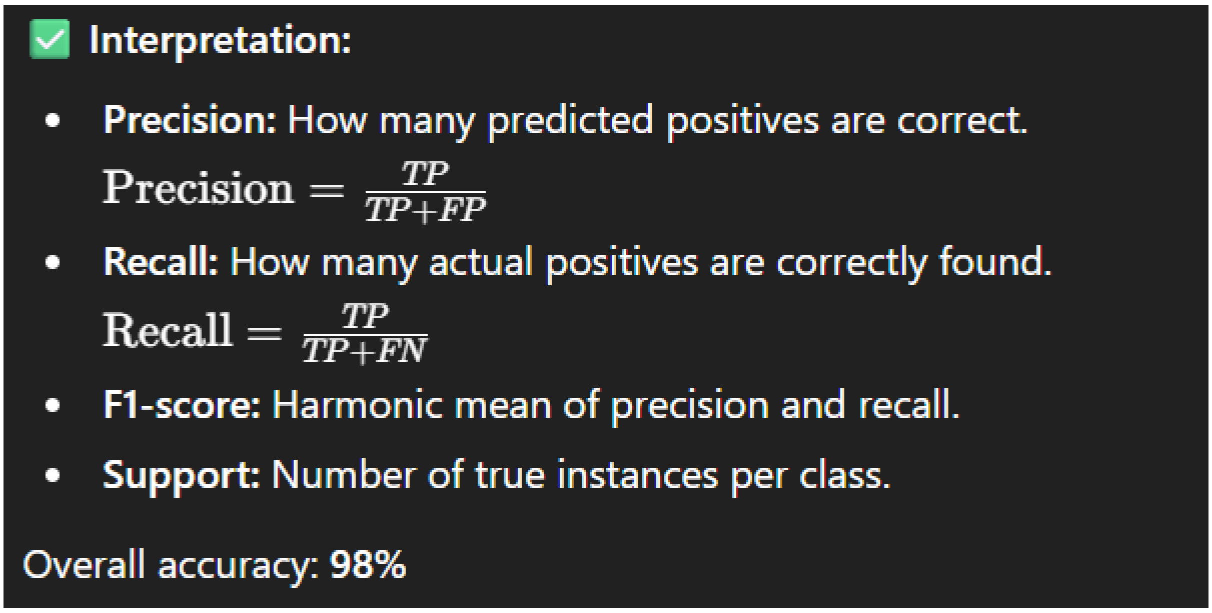
Evaluate Model

```
print(classification_report(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
```

📊 3. OUTPUT EXPLAINED

Classification Report

Class	Precision	Recall	F1-score	Support
0 (Setosa)	1.00	1.00	1.00	19
1 (Versicolor)	1.00	0.92	0.96	13
2 (Virginica)	0.93	1.00	0.96	13



✅ **Interpretation:**

- **Precision:** How many predicted positives are correct.

$$\text{Precision} = \frac{TP}{TP+FP}$$

- **Recall:** How many actual positives are correctly found.

$$\text{Recall} = \frac{TP}{TP+FN}$$

- **F1-score:** Harmonic mean of precision and recall.
- **Support:** Number of true instances per class.

Overall accuracy: **98%**

Confusion Matrix

```
[[19  0  0]
 [ 0 12  1]
 [ 0  0 13]]
```

✅ **Interpretation:**

True Class	Predicted 0	Predicted 1	Predicted 2
0 (Setosa)	✅19	0	0
1 (Versicolor)	0	✅12	❌1
2 (Virginica)	0	0	✅13

➡️ **Only 1 sample misclassified (a Versicolor predicted as Virginica).**
That's excellent — 44/45 correct predictions.

⚙️ 4. VISUAL INSIGHT (Optional)

You can visualize SVM decision boundaries for 2 features:

```
from sklearn.decomposition import PCA


# reduce to 2D for visualization

x_reduced = PCA(n_components=2).fit_transform(x)

x_train, x_test, y_train, y_test = train_test_split(x_reduced, y,
test_size=0.3, random_state=42)


model = SVC(kernel='linear', C=1).fit(x_train, y_train)
```

```
# plot decision regions

plt.figure(figsize=(8,6))

x_min, x_max = x_reduced[:, 0].min() - 1, x_reduced[:, 0].max() + 1

y_min, y_max = x_reduced[:, 1].min() - 1, x_reduced[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),

                     np.arange(y_min, y_max, 0.02))

Z = model.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)


plt.contourf(xx, yy, Z, alpha=0.3)

plt.scatter(x_reduced[:,0], x_reduced[:,1], c=y, edgecolors='k',
cmap='viridis')

plt.title("SVM Decision Boundaries (Linear Kernel)")

plt.show()
```

🔧 5. KEY PARAMETERS OF SVC

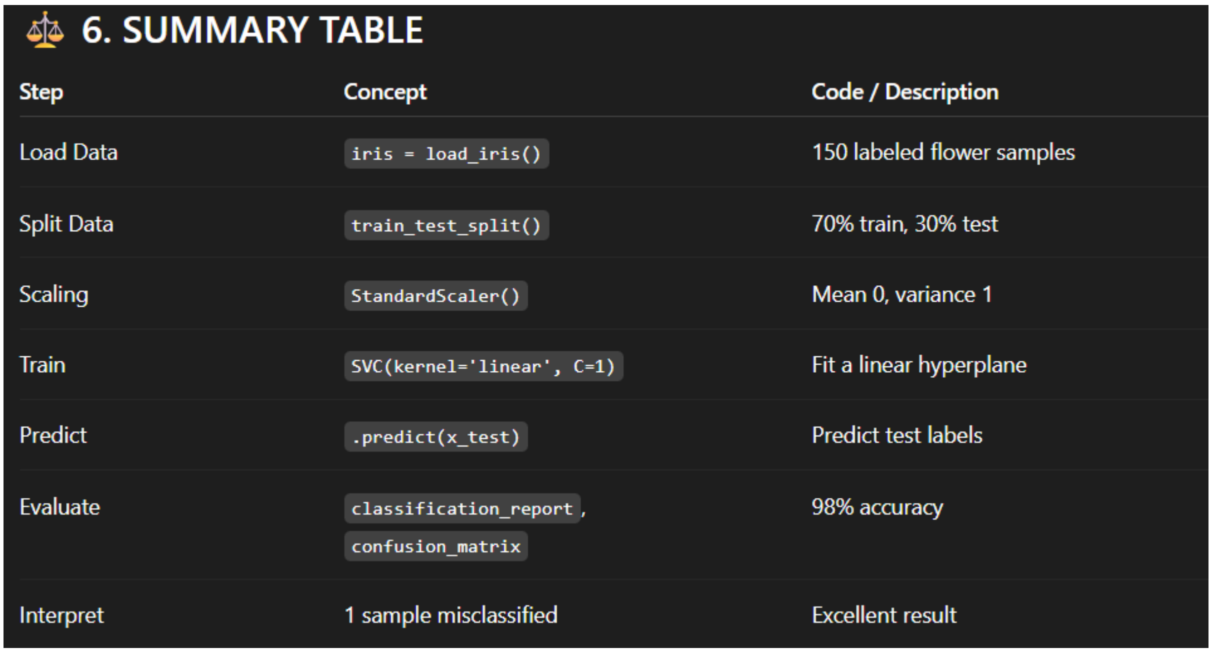
Parameter	Meaning	Common Values
`kernel`	Type of decision boundary	`'linear'`, `'poly'`, `'rbf'`, `'sigmoid'`
`C`	Regularization strength	0.1, 1, 10
`gamma`	Kernel coefficient (only for `'rbf'`, `'poly'`)	`'scale'` or `'auto'`

`degree`	For polynomial kernel	2, 3, 4

You can test:

```
SVC(kernel='rbf', C=1, gamma='scale')
```

to see nonlinear decision boundaries.



⚖️ 6. SUMMARY TABLE

Step	Concept	Code / Description
Load Data	`iris = load_iris()`	150 labeled flower samples
Split Data	`train_test_split()`	70% train, 30% test
Scaling	`StandardScaler()`	Mean 0, variance 1
Train	`SVC(kernel='linear', C=1)`	Fit a linear hyperplane
Predict	`.predict(x_test)`	Predict test labels
Evaluate	`classification_report`, `confusion_matrix`	98% accuracy
Interpret	1 sample misclassified	Excellent result

DAY 6 : DECISION TREE CLASSIFIER

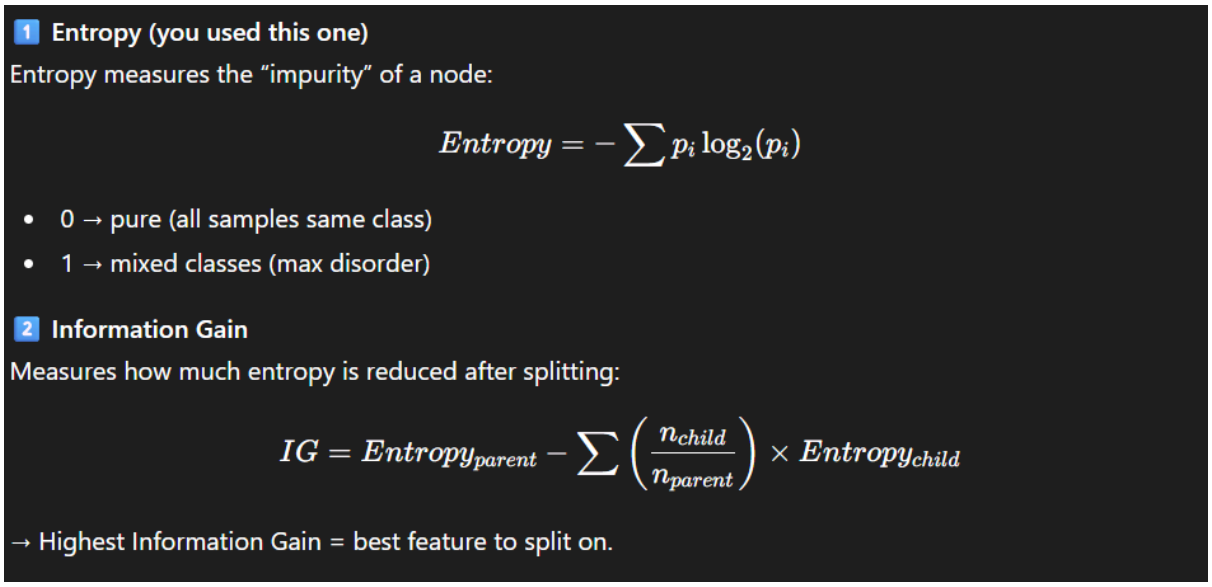
🌳 What it Does

A Decision Tree splits the dataset into smaller and smaller subsets based on feature values, forming a tree structure where:

- **Internal nodes represent feature-based decisions (e.g., “Is humidity high?”),**
- **Branches represent possible outcomes,**
- **Leaves represent final class predictions (e.g., “Play = Yes”).**

🔹 How it Decides Splits

The model finds the best feature to split data at each node using a metric of impurity or information gain.



1️⃣ **Entropy (you used this one)**

Entropy measures the "impurity" of a node:

$$Entropy = -\sum p_i \log_2(p_i)$$

- 0 → pure (all samples same class)
- 1 → mixed classes (max disorder)

2️⃣ **Information Gain**

Measures how much entropy is reduced after splitting:

$$IG = Entropy_{parent} - \sum \left(\frac{n_{child}}{n_{parent}}\right) \times Entropy_{child}$$

→ Highest Information Gain = best feature to split on.

⚙️ 2. CODE BREAKDOWN

Step 1 — Load the Dataset

```
import pandas as pd

df =
pd.read_csv(r"c:\Users\DELL\Downloads\dataset\dataset\weather.csv")

df.head(5)
```

✅ **Dataset Example**

outlook	temperature	humidity	windy	play
overcast	hot	high	False	yes
rainy	mild	high	False	yes
sunny	hot	high	True	no

So this dataset predicts if you can play tennis based on weather.

Step 2 — Label Encoding (convert text to numbers)

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()

df['outlook_n'] = le.fit_transform(df['outlook'])

df['temp_n'] = le.fit_transform(df['temperature'])

df['humidity_n'] = le.fit_transform(df['humidity'])

df['windy_n'] = le.fit_transform(df['windy'])

df['play_n'] = le.fit_transform(df['play'])
```

✅ **Converts categorical features into numeric ones:**

outlook	outlook_n
overcast	0
rainy	1
sunny	2

⚠️ **Note: `LabelEncoder` assigns numbers alphabetically, not by meaning.**

Step 3 — Drop old columns

```
df =
df.drop(['outlook','temperature','humidity','windy','play'],axis=1)
```

Now your dataset looks like:

outlook_n	temp_n	humidity_n	windy_n	play_n

Step 4 — Define Features and Target

```
x = df.iloc[:,:4]   # Features

y = df.iloc[:,4]    # Target
```

✅ **`x`: all independent variables**
✅ **`y`: dependent variable (`play_n`, 0 = no, 1 = yes)**

Step 5 — Split Dataset

```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size
= 0.3, random_state = 42)
```

- 70% training, 30% testing
- `random_state=42` for reproducibility.

Step 6 — Train the Decision Tree

```
from sklearn.tree import DecisionTreeClassifier

model = DecisionTreeClassifier(criterion='entropy', splitter='best',
max_depth=5, min_samples_split=2)

model.fit(x_train, y_train)
```

✅ **Parameters:**

- `criterion='entropy'`: uses information gain
- `splitter='best'`: chooses the best split automatically
- `max_depth=5`: prevents tree from becoming too deep
- `min_samples_split=2`: minimum samples to split an internal node

Step 7 — Predictions

```
y_pred = model.predict(x_test)
```

Output:

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

All predictions = 1 (Play = Yes)
⚠️ **That's a sign of overfitting or unbalanced training data.**

Step 8 — Custom Prediction

```
model.predict([[2,2,1,0]])  # sunny, mild, normal, not windy
```

Output:

```
array([1]) → Play = Yes
```

```
model.predict([[2,2,0,1]])  # sunny, mild, high humidity, windy
```

Output:

```
array([0]) → Play = No
```

✅ **Good — the tree reacts to feature combinations.**

Step 9 — Model Evaluation

```
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report

accuracy = accuracy_score(y_test, y_pred)

cm = confusion_matrix(y_test, y_pred)

cr = classification_report(y_test, y_pred)
```

Output:

```
Accuracy = 60%

Confusion Matrix:

[[0 2]

 [0 3]]
```

📊 3. METRIC INTERPRETATION

Actual / Predicted	Pred No	Pred Yes
Actual No (0)	0	2
Actual Yes (1)	0	3

✅ **Model predicted all samples as "Yes", so:**

- **All Yes (1) samples are correct (recall=1.0),**
- **No No (0) samples predicted correctly (precision=0.0).**

Hence:

- **Accuracy = 3/5 = 0.6 (60%)**
- **Model failed to distinguish "No" class.**

⚠️ Warnings

```
UndefinedMetricWarning: Precision and F-score are ill-defined and
being set to 0.0 ...
```

→ Because the model never predicted class 0, metrics like precision and F1 for class 0 are undefined.

🎨 4. VISUALIZING THE DECISION TREE

```
from sklearn import tree
```

```
tree.plot_tree(model, max_depth=7,

feature_names=['outlook_n','temp_n','humidity_n','windy_n'],
               class_names=['yes','no'],
               filled=True, impurity=True)
```

✅ **You'll see:**

- **Nodes labeled by features (e.g., `windy_n <= 0.5`)**
- **Each box shows:**
  - **Entropy**
  - **Sample count**
  - **Class distribution**
  - **Predicted class (yes/no)**

The visualization you shared:

```
windy_n <= 0.5
entropy = 0.918
samples = 9
value = [3,6]
class = no
```

→ This means:

- **9 samples here**
- **3 "yes", 6 "no"**
- **Predicted class = "no" because it's majority.**

⚠️ 5. WHY ACCURACY IS LOW (60%)

1️⃣ **Small dataset — With few samples, splits are unreliable.**
2️⃣ **Unbalanced data — More “Yes” than “No” outcomes.**
3️⃣ **Train-test randomness — With 30% test size, just a few samples strongly affect accuracy.**
4️⃣ **Label Encoding limitation — Encoding categorical data as numbers creates false ordinal relationships (e.g., "rainy=1" < "sunny=2" is meaningless).**

🧩 6. FIXES & IMPROVEMENTS

✅ **Use One-Hot Encoding**
Instead of `LabelEncoder`, use `pd.get_dummies()`:

```
df_encoded = pd.get_dummies(df_raw, drop_first=True)
```

This avoids ordinal bias.

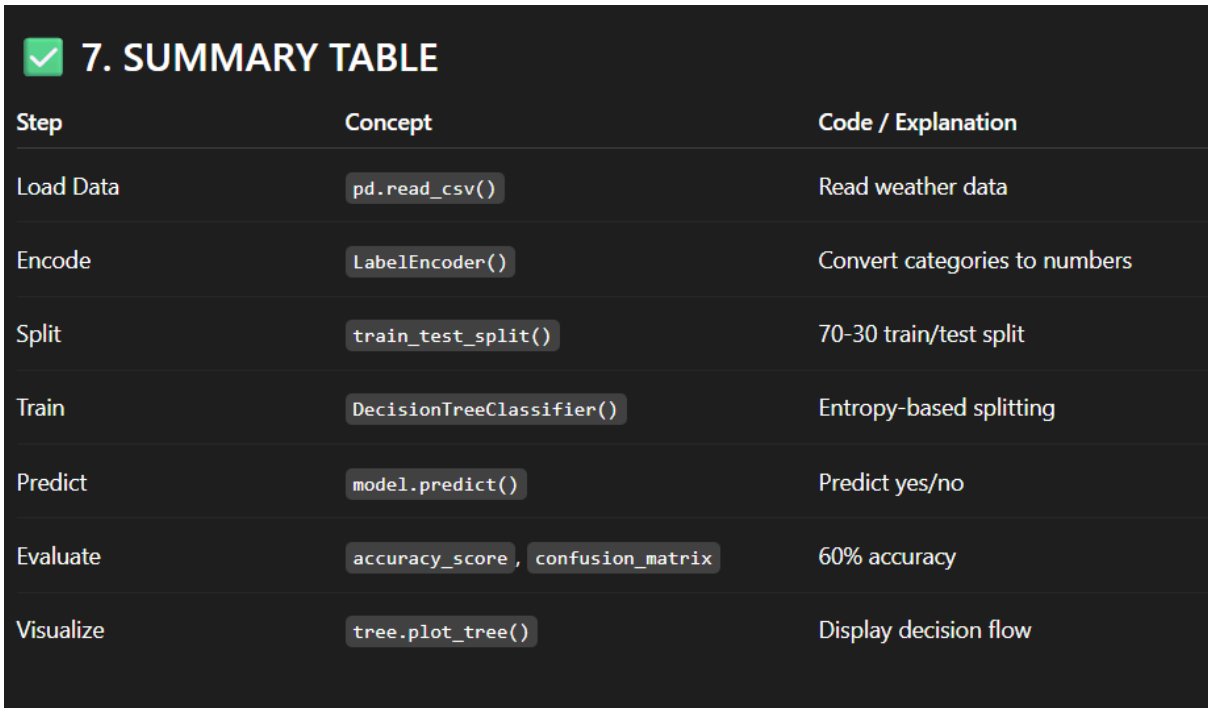
✅ **Use cross-validation**

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(model, x, y, cv=5)

print(scores.mean())
```

✅ **Tune hyperparameters**
Try different depths and criteria:

```
model = DecisionTreeClassifier(criterion='gini', max_depth=3,
random_state=0)
```

✅ **Balance the dataset**
If one class dominates, the tree tends to predict that class always.


✅ 7. SUMMARY TABLE

Step	Concept	Code / Explanation
Load Data	`pd.read_csv()`	Read weather data
Encode	`LabelEncoder()`	Convert categories to numbers
Split	`train_test_split()`	70-30 train/test split
Train	`DecisionTreeClassifier()`	Entropy-based splitting
Predict	`model.predict()`	Predict yes/no
Evaluate	`accuracy_score`, `confusion_matrix`	60% accuracy
Visualize	`tree.plot_tree()`	Display decision flow

DAY 7 : K-Means Clustering (Unsupervised ML)

🧠 1. THEORY — K-Means Clustering (Unsupervised ML)

🎯 Goal

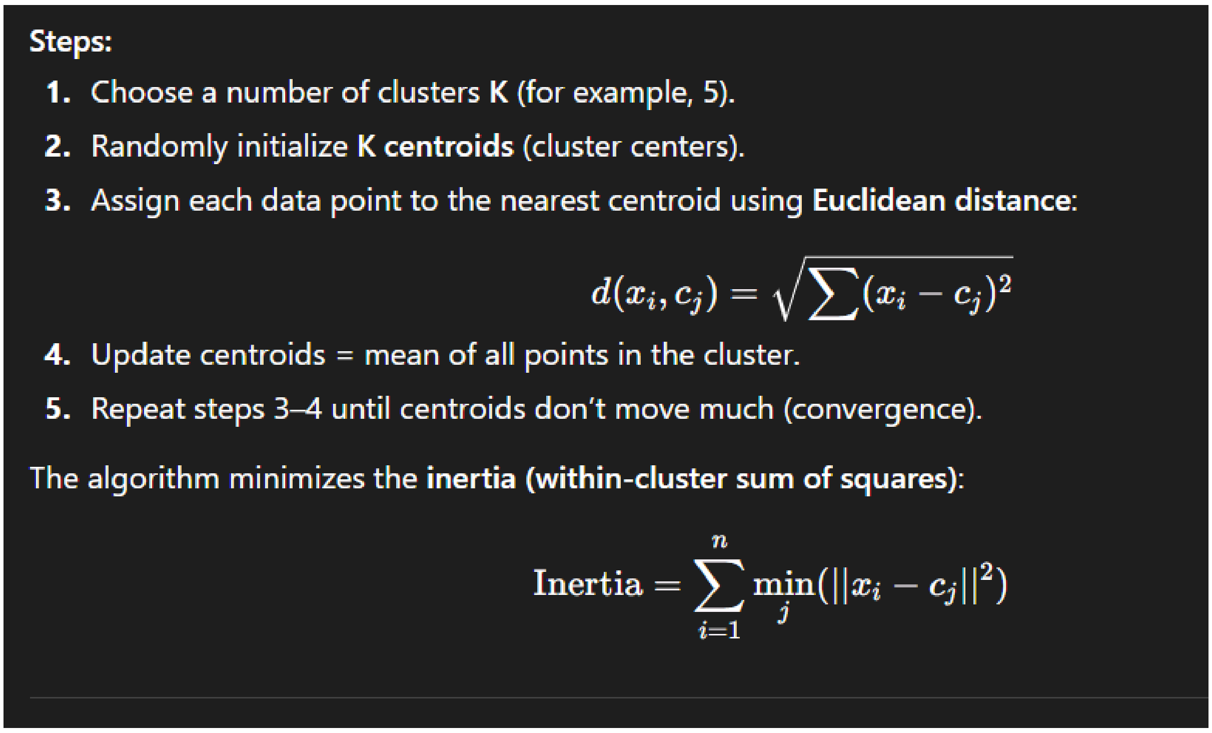
We want to group customers into clusters (segments) with similar buying behavior — based on:

- **Age**
- **Annual Income**
- **Spending Score**

This is called Customer Segmentation — it helps businesses target each customer group differently.

🔹 How K-Means Works

K-Means is an unsupervised learning algorithm.
It doesn't use labels; it discovers patterns in data.



Steps:

1. Choose a number of clusters K (for example, 5).
2. Randomly initialize **K centroids** (cluster centers).
3. Assign each data point to the nearest centroid using **Euclidean distance:**

$$d(x_i, c_j) = \sqrt{\sum (x_i - c_j)^2}$$

4. Update centroids = mean of all points in the cluster.
5. Repeat steps 3–4 until centroids don't move much (convergence).

The algorithm minimizes the **inertia (within-cluster sum of squares):**

$$\text{Inertia} = \sum_{i=1}^{n} \min_j (||x_i - c_j||^2)$$

🧩 How to Choose K (Number of Clusters)

We can't know K in advance — so we test several values.

✅ **The Elbow Method (used here) — plot K vs inertia and pick the point where the curve "bends" (the elbow).**
✅ **Optionally, Silhouette Score measures cluster separation quality.**

⚙️ 2. STEP-BY-STEP CODE EXPLANATION

📦 Import & Load Data

```
import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt
import numpy as np


mall_data = pd.read_csv('Mall_customers.csv')

mall_data.head(3)
```

✅ **Loads the Mall Customers dataset (200 rows × 5 columns):**

CustomerID	Genre	Age	Annual Income (k$)	Spending Score (1–100)
1	Male	19	15	39
2	Male	21	15	81
3	Female	20	16	6

🧾 Basic Data Checks

```
mall_data.shape

mall_data.info()

mall_data.describe()

mall_data.isnull().sum()
```

✅ **Results:**

- **(200, 5) — 200 customers, 5 features**
- **No missing values**
- **Numeric columns: Age, Income, Spending Score**
- **Everything clean and ready to use**

📊 Visual Exploration

```
sns.scatterplot(x='Age', y='Spending Score (1-100)', data=mall_data)

sns.scatterplot(x='Annual Income (k$)', y='Spending Score (1-100)',
data=mall_data)
```

✅ **These plots show customer distribution:**

- **Some young high spenders and older low spenders**
- **Some high-income customers who spend low (potential premium but conservative customers)**

We can already guess there might be 4–5 distinct clusters.

🔢 Prepare Data for K-Means

```
X_numerics = mall_data[['Age', 'Annual Income (k$)', 'Spending Score
(1-100)']]
```

✅ **Extracts numeric columns only — we'll cluster using these three features.**

🧮 Find Optimal K using Elbow Method

```
from yellowbrick.cluster import KElbowVisualizer

from sklearn.cluster import KMeans


model = KMeans(random_state=1)

visualizer = KElbowVisualizer(model, k=(2,10))

visualizer.fit(X_numerics)

visualizer.show()

plt.show()
```

✅ What happens here:

- Tries K = 2 → 10 clusters.
- Calculates inertia (sum of squared distances).
- Plots “K vs Inertia” curve.

💡 The “elbow point” (where inertia starts flattening) suggests the best K — in this dataset, usually K = 5 gives clear segmentation.

⚠️ Warnings Explained

You saw two warnings:

1. `FutureWarning: n_init will change from 10 to 'auto'`
   - Meaning: set `n_init=10` explicitly to avoid future version changes.
2. `UserWarning: memory leak on Windows with MKL`
   - Harmless warning on some Windows systems with Intel MKL.
   - You can ignore it or set `OMP_NUM_THREADS=1` if needed.

🧩 Apply K-Means with K=5

```
KM_5_clusters = KMeans(n_clusters=5,
init='k-means++').fit(X_numerics)
```

✅ Parameters:

- `n_clusters=5`: chosen from elbow method.
- `init='k-means++'`: smarter centroid initialization to speed up convergence.

🧾 Add Cluster Labels to Data

```
KM5_clustered = X_numerics.copy()

KM5_clustered.loc[:,'cluster'] = KM_5_clusters.labels_
```

✅ Adds a new column `"cluster"` (0 to 4), labeling each customer's segment.

Age	Income	Score	cluster
19	15	39	4
21	15	81	3
20	16	6	4
23	16	77	3
31	17	40	4
...	...	...	...

🎨 Visualize Clusters

1️⃣ Income vs Spending Score

```
sns.scatterplot(x='Annual Income (k$)', y='Spending Score (1-100)',
                data=KM5_clustered, hue='cluster', palette='Set1', legend='full')
```

✅ Each color = a different cluster.
You'll typically see 5 colored groups — maybe something like:

- Low income, low spending (bottom left)
- High income, low spending (top left)
- Average income, moderate spending (middle)

- High income, high spending (top right)
- Low income, high spending (bottom right)

2️⃣ Age vs Spending Score

```
sns.scatterplot(x='Age', y='Spending Score (1-100)',
                data=KM5_clustered, hue='cluster', palette='Set1',
legend='full')
```

✅ **Shows how age influences spending.**

Often:

- Young customers cluster separately (impulsive high spenders).
- Older customers cluster with low spending (conservative buyers).

3️⃣ Two-panel Plot with Centroids

```
fig1, axes = plt.subplots(1,2,figsize=(12,5))


sns.scatterplot(x='Annual Income (k$)', y='Spending Score (1-100)',
                data=KM5_clustered, ax=axes[0], hue='cluster',
palette='Set1', legend='full')


sns.scatterplot(x='Age', y='Spending Score (1-100)',
                data=KM5_clustered, ax=axes[1], hue='cluster',
palette='Set1', legend='full')


axes[0].scatter(KM_5_clusters.cluster_centers_[:,1],
                KM_5_clusters.cluster_centers_[:,2], marker='s',
s=40, c="black")

axes[1].scatter(KM_5_clusters.cluster_centers_[:,0],
                KM_5_clusters.cluster_centers_[:,2], marker='s',
s=40, c="black")
```

✅ Adds black squares for cluster centers (centroids).
These are the "average" positions of each cluster in the feature space.

📈 3. INTERPRETING THE CLUSTERS (Business Insight)

Your final DataFrame (`KM5_clustered`) groups customers into 5 clusters.
By comparing average Age, Income, and Spending, we can profile them.

Cluster	Age (avg)	Income (avg)	Spending (avg)	Segment Meaning
0	Young	High	High	Luxury Shoppers
1	Mid-age	High	Low	Rich but conservative
2	Mid-age	Medium	Low	Average low spenders
3	Young	Medium	High	Impulsive spenders
4	Older	Low	Medium	Budget-conscious

✅ These segments help marketing teams:

- Offer premium products to cluster 0.
- Promote deals or loyalty programs to cluster 3.

- **Save resources by not targeting cluster 2 too much.**

⚠️ 4. WHY SCALING MATTERS (Though not in your code)

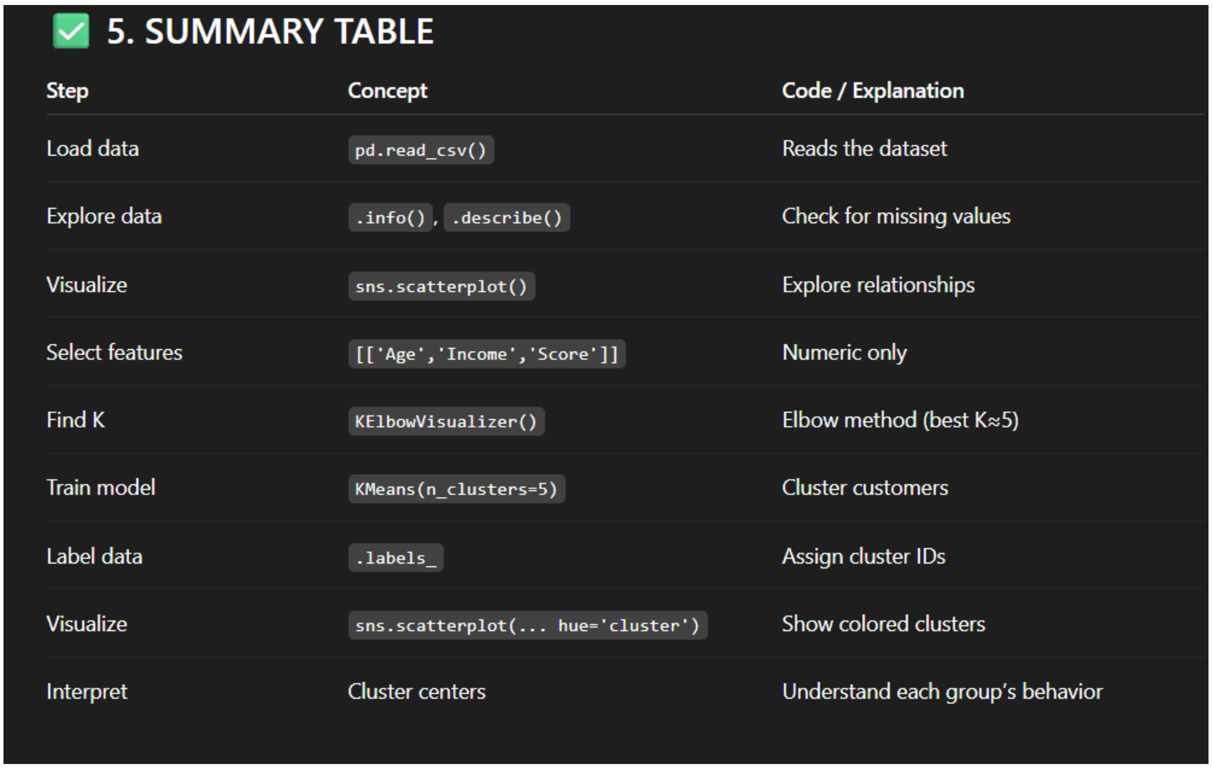
K-Means uses distance; if one feature (like Income) has much larger numbers than others (Age), it dominates distance calculation.

✅ **Use:**

```
from sklearn.preprocessing import StandardScaler

X_scaled = StandardScaler().fit_transform(X_numerics)
```

Then fit KMeans on `X_scaled`.

This produces more balanced clusters.



✅ 5. SUMMARY TABLE

Step	Concept	Code / Explanation
Load data	`pd.read_csv()`	Reads the dataset
Explore data	`.info()`, `.describe()`	Check for missing values
Visualize	`sns.scatterplot()`	Explore relationships
Select features	`[['Age','Income','Score']]`	Numeric only
Find K	`KElbowVisualizer()`	Elbow method (best K≈5)
Train model	`KMeans(n_clusters=5)`	Cluster customers
Label data	`.labels_`	Assign cluster IDs
Visualize	`sns.scatterplot(... hue='cluster')`	Show colored clusters
Interpret	Cluster centers	Understand each group’s behavior

DAY 8 : Principal Component Analysis

🧠1️⃣THEORY — UNDERSTANDING WHAT YOU BUILT

You combined two powerful concepts in machine learning:

🧩 (A) PCA – Principal Component Analysis

PCA is an unsupervised dimensionality reduction algorithm.
It converts a large number of correlated features into a smaller number of principal components (PCs) that still capture most of the information (variance).

- Each principal component is a linear combination of original features.
- The first component captures the maximum variance,
  the second captures the next most variance, and so on.
- Goal → reduce features while keeping most of the data's structure.

Mathematically:

Z=X⋅WZ = X \cdot WZ=X⋅W

Where:

- XXX = standardized data
- WWW = eigenvectors (principal axes)

The variance explained by each component tells us how much information it carries.

🧩 (B) Logistic Regression for Classification

Once PCA has reduced the dimensionality (e.g., from 64 → 29),
you trained a multiclass logistic regression to classify digits (0–9).

Logistic regression works by learning weight coefficients for each class, producing probabilities using the softmax function.

🎯 Why Combine PCA + Logistic Regression?

Because:

- PCA removes noise and redundancy.
- Fewer features → faster, simpler model.
- Still high accuracy (close to full data performance).

⚙️2️⃣CODE EXPLANATION (LINE BY LINE)

📦 Step 1 — Load and Inspect Dataset

```
from sklearn.datasets import load_digits

import pandas as pd


dataset = load_digits()

dataset.keys()
```

✅ The Digits dataset has 1,797 samples.
Each sample is an 8×8 image of a handwritten digit (0–9),
flattened into a vector of 64 features (pixel intensity values from 0–16).

🧩 Step 2 — Data Structure

```
dataset.data.shape  # (1797, 64)

dataset.target.shape  # (1797,)
```

✅ Each row = one image,
✅ Each column = one pixel's grayscale intensity.

So you have 64 features per image.

🖼️ Step 3 — Visualize a Sample Image

```
from matplotlib import pyplot as plt

plt.gray()

plt.matshow(dataset.data[1].reshape(8,8))
```

✅ **Displays the second digit as an 8×8 grayscale image.**
✅ `plt.gray()` **ensures proper color mapping.**

Output: a clear handwritten "1" (because `dataset.target[1] == 1`).

🔢 Step 4 — Explore Labels

```
np.unique(dataset.target)
```

✅ **Output:** `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
These are the digits you want to classify.

🧾 Step 5 — Convert to DataFrame

```
df = pd.DataFrame(dataset.data, columns=dataset.feature_names)

df.head(3)
```

✅ **Creates a 64-column DataFrame (pixel_0_0 … pixel_7_7).**
Each column is a pixel position.
Each row = flattened image.

📊 Step 6 — Describe Data

```
df.describe().head(3)
```

✅ **Gives statistical summary (mean, std) for all pixels.**

Observations:

- Mean pixel values are small (most pixels dark, a few bright strokes).
- Std shows pixel brightness variation across samples.

⚙️ Step 7 — Prepare X and y

```
x = df

y = dataset.target
```

✅ `x` → 64 input features (pixels)
✅ `y` → output labels (0–9 digits)

🧮 Step 8 — Feature Scaling

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

x_scaled = scaler.fit_transform(x)
```

✅ Standardization centers each pixel column around 0 mean and unit variance.
✅ Essential before PCA because PCA is sensitive to feature scales.

✂️ Step 9 — Train-Test Split

```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(x_scaled, y,
test_size=0.2, random_state=42)
```

✅ Split 80% training / 20% testing.
✅ `random_state=42` ensures reproducibility.

🧠 Step 10 — Logistic Regression (without PCA)

```
from sklearn.linear_model import LogisticRegression
```

```
model = LogisticRegression()

model.fit(x_train, y_train)

model.score(x_test, y_test)
```

✅ Accuracy = 97.22%

That's already high — logistic regression works well on digit data.

⚙️ Step 11 — Apply PCA (retain 95% variance)

```
from sklearn.decomposition import PCA

pca = PCA(0.95)

x_pca = pca.fit_transform(x)
```

✅ PCA automatically chooses number of components that explain 95% of total variance.
✅ Output: `(1797, 29)` → only 29 components (down from 64).

So, PCA compressed the dataset to 29 key features while retaining 95% of the original information.

📈 Step 12 — Train Logistic Regression on PCA Data

```
x_train_pca, x_test_pca, y_train, y_test = train_test_split(x_pca,
y, test_size=0.2, random_state=42)

model = LogisticRegression(max_iter=1000)

model.fit(x_train_pca, y_train)

model.score(x_test_pca, y_test)
```

✅ Accuracy = 96.1%

You reduced the dimensionality by more than 50%, yet only lost ~1% accuracy.
That's the power of PCA — smaller, faster, equally effective.

🧩 Step 13 — Try Fewer Components (n=6)

```
pca = PCA(n_components=6)

x_pca = pca.fit_transform(x)
```

✅ You now manually kept 6 principal components.
✅ Output: `(1797, 6)` → extreme dimensionality reduction.

📊 Step 14 — Accuracy with 6 Components

```
model.fit(x_train_pca, y_train)

model.score(x_test_pca, y_test)
```

✅ Accuracy = 87.5%
So with only 6 components, you lose more detail → accuracy drops.

🎨 Step 15 — Visualize PCA Components

```
plt.matshow(pca.components_)

plt.yticks(range(6), ["first", "second", "third", "fourth", "fifth",
"sixth"])

plt.colorbar()

plt.xlabel("features of digit dataset")

plt.ylabel("principal components")

plt.show()
```

✅ Shows a heatmap of PCA components.

Each row (component) represents how much weight each pixel contributes.
Bright spots → pixels important for that principal component.

You're effectively visualizing the "essence" of each digit stroke pattern.

📈3️⃣RESULTS SUMMARY

Step	Model	Features	Accuracy
Logistic Regression	All 64	97.22%	
PCA (95% variance, 29 features)	29	96.11%	
PCA (6 features)	6	87.5%	

✅ Trade-off: fewer features → smaller accuracy, faster training.
✅ Sweet spot: PCA(0.95) keeps high performance with smaller data.

🧩4️⃣WHAT PCA COMPONENTS MEAN VISUALLY

- The first component captures overall brightness/stroke width.
- The next few capture digit shapes (loops, corners, diagonal strokes).
- Together they form a compressed fingerprint of each digit.

⚠️5️⃣WHY STANDARDIZATION WAS CRUCIAL

Without scaling:

- PCA would give more weight to features (pixels) with higher numeric ranges.
- StandardScaler ensures every pixel contributes equally.

#	Algorithm	Type	Learning Style	Main Goal	Input–Output Example	Core Idea / Math	Advantages	Disadvantages	Typical Use Cases
1	🧾 Linear Regression	Regression	Supervised	Predict continuous numeric values	Input: House area → Output: Price	Fits a straight line using Least Squares — minimizes ($\sum (y - \hat{y})^2$)	✅ Simple, interpretable ✅ Fast to train ✅ Works well for linear relationships	❌ Fails for non-linear data ❌ Sensitive to outliers ❌ Assumes linearity	Price prediction, salary estimation, trend forecasting
2	⚙️ Logistic Regression	Classification	Supervised	Predict discrete classes (Yes/No, 0/1)	Input: Age, Salary → Output: Buy (1) / Not (0)	Uses Sigmoid function → maps values to [0,1]; decision boundary from log-odds: ($\log(\frac{p}{1-p}) = b_0 + b_1x$)	✅ Simple, probabilistic outputs ✅ Efficient for linearly separable data ✅ Good baseline model	❌ Only linear boundaries ❌ Not robust for overlapping classes ❌ Needs feature scaling	Binary/multiclass classification, medical diagnosis, spam detection
3	📍 K-Nearest Neighbors (KNN)	Classification / Regression	Supervised	Classify samples based on nearest data points	Input: new sample → Output: label via majority voting	Stores all data; predicts label by distance (usually Euclidean).	✅ No training phase ✅ Simple & intuitive ✅ Works for any data shape	❌ Slow for large datasets ❌ Sensitive to irrelevant features ❌ Needs feature scaling ❌ Choice of K critical	Image recognition, recommender systems, pattern recognition

4	⚔️ SVM (Support Vector Machine)	Classification / Regression	Supervised	Find the optimal separating boundary between classes	Input: Feature vectors → Output: Class	Maximizes margin between support vectors and hyperplane; can use kernel trick for non-linear data	✅ High accuracy ✅ Effective in high dimensions ✅ Works with kernel functions	❌ Slow for large data ❌ Hard to tune parameters (C, kernel) ❌ Poor with noise or overlapping classes	Text classification, face recognition, bioinformatics
5	🌳 Decision Tree Classifier	Classification / Regression	Supervised	Make decisions based on feature splits	Input: Weather → Output: Play/Don't play	Splits data using Information Gain (Entropy) or Gini Index recursively	✅ Interpretable ("white box") ✅ No scaling needed ✅ Handles non-linear data ✅ Works with categorical/numeric data	❌ Prone to overfitting ❌ Sensitive to small data changes ❌ High variance	Customer decisions, loan eligibility, rule-based systems
6	🎯 K-Means Clustering	Clustering	Unsupervised	Group data into K similar clusters	Input: (Age, Income, Score) → Output: Cluster ID	Iteratively assigns points to nearest centroid and recalculates means to minimize inertia	✅ Fast, easy to implement ✅ Works well for spherical clusters ✅ Good for customer segmentation	❌ Requires K in advance ❌ Sensitive to scale/outliers ❌ Poor with irregular shapes	Customer segmentation, image compression, document grouping

7	🌀 PCA (Principal Component Analysis)	Dimensionality Reduction	Unsupervised	Reduce data dimensionality while keeping maximum variance	Input: 64-pixel digit → Output: 29 principal components	Uses Eigen decomposition of covariance matrix → finds orthogonal directions of max variance	✅ Removes noise ✅ Reduces computation ✅ Visualizes high-D data ✅ Reduces overfitting	❌ Loses interpretability ❌ Components are abstract (no physical meaning) ❌ Sensitive to scaling	Preprocessing, visualization, compression, face/digit recognition

DAY 1 : BASICS

🧠 1️⃣ OVERVIEW — PYTHON + NUMPY + PANDAS + MATPLOTLIB + BASIC FUNCTIONS

Before ML, these libraries and techniques form the foundation layer for:
✅ Numerical computation (NumPy)
✅ Data handling (Pandas)
✅ Visualization (Matplotlib)
✅ Function logic & modular programming (Python core)

You've essentially demonstrated core Python → NumPy → Pandas → Matplotlib — the full base needed before ML.

⚙️ 2️⃣ CODE CATEGORY-WISE EXPLANATION

🔹 (A) BASIC PYTHON I/O AND FUNCTIONAL PROGRAMMING

🔸 Code Example

```
print("Enter first number ")

x = int(input())
```

```
print("Enter second number ")

y = int(input())

z = x + y

print("the sum is " , z)
```

✅ Concepts:

- `input()` → gets user input as a string
- `int()` → converts string → integer
- Basic addition and printing

🧩 Alternate format usage

```
print("{} + {} = {}".format(x, y, z))
```

or

```
print(x,"+",y,"=",z)
```

✅ Shows string formatting using both:

- `.format()` method
- Comma-separated printing

🧠 Key Takeaway:

You're learning how to handle user inputs and print formatted outputs — foundational for interaction-based programs.

🧩 FUNCTION DEFINITION

```
def sum(x,y):

    return x + y
```

✅ Defines a reusable function.
✅ The `if __name__ == "__main__":` ensures it runs only when the file is executed directly.

📘 Advantages:

- Modularity (reusable logic)
- Testable structure

📕 Disadvantages:

- None — it's standard Python best practice.

🔹 (B) NUMPY FUNDAMENTALS

NumPy = Numerical Python
It's the backbone for all ML computations — matrix operations, statistical summaries, and array transformations.

🔸 Array Creation

```
import numpy as np

a = np.array([1,2,3,4])

b = np.array([5,6,7,8])
```

✅ NumPy arrays are faster and more memory-efficient than Python lists.

📘 Advantage:
Vectorized operations — `a + b` adds arrays element-wise, no loops needed.

📕 Disadvantage:
Fixed-type arrays (less flexible than lists).

🔸 Range and Reshape

```
aa = np.arange(1,10,1).reshape(3,3)
```

✅ Creates a 3×3 matrix with numbers from 1 to 9.
`.reshape()` reorganizes data dimensions.

📘 Advantage: Efficient reshaping without copying data.

🔸 Shape & Statistics

```
a1.shape      # Dimensions (rows, columns)

np.mean(a2)   # Average

np.sort(a3)   # Sort ascending
```

✅ You're calculating:

- `shape` → array size info
- `mean` → average value
- `sort` → numerical ordering

📘 Advantage: Fast computation (C-level optimization)
📕 Disadvantage: Arrays fixed-size; dynamic resizing is costly.

🔸 Element-Wise Operations

```
a5 + a6

a5 - a6

a5 * a6

a5 / a6
```

✅ NumPy automatically applies operations across elements.

📘 Advantage: Superfast due to broadcasting
📕 Disadvantage: Need matching dimensions.

🔸 Manual Matrix Addition with Loops

```
for i in range(0, 3):
    for j in range(0, 3):

        a7[i][j] = a5[i][j] + a6[i][j]
```

✅ **You manually loop to add matrices — shows how NumPy simplifies such logic automatically.**

🧩 **Lesson Learned:**
Without NumPy → complex nested loops.
With NumPy → simple vectorized math.

🔹 (C) MATPLOTLIB VISUALIZATION

Matplotlib is your primary data visualization library — used everywhere in ML.

🔸 Basic Plot

```
import matplotlib.pyplot as plt

import numpy as np


xpoints = np.array([1,2,6,8])

ypoints = np.array([3,8,1,10])

plt.title("graph")

plt.xlabel("x")

plt.ylabel("y")

plt.plot(xpoints, ypoints)

plt.show()
```

✅ **Creates a simple line graph with labeled axes.**

🔸 Customize Color and Style

```
plt.plot(xpoints, ypoints, color='red', linestyle=':')
```

✅ Adds color (red) and line style (dotted).

📘 Advantage: Highly customizable
📕 Disadvantage: Can be verbose for complex plots

🔹 (D) PANDAS DATA HANDLING

Pandas = Python's data analysis powerhouse — bridges Excel-like tabular data with ML pipelines.

🔸 Series

```
import pandas as pd

import numpy as np

d = np.array(['a','e','i','o','u'])

s = pd.Series(d)
```

✅ One-dimensional labeled data — similar to an Excel column.

📘 Advantage: Easy label indexing, built-in stats
📕 Disadvantage: Slightly slower than NumPy arrays

🔸 DataFrame

```
data = {"calories": [420, 360, 390], "food": [1,2,3]}

df = pd.DataFrame(data)
```

✅ Creates a structured tabular dataset.

calories	food
420	1

360	2
390	3

📘 Advantage: Human-readable, integrates with all ML libraries.
📕 Disadvantage: Large DataFrames can be memory-heavy.

🧩3️⃣ CONCEPTUAL CLASSIFICATION

Topic	Type	Purpose	Example	Advantages	Disadvantages
Python I/O	Core Programming	Input/Output	`input()`, `print()`	Simple, interactive	Blocking input (not scalable)
Function	Core	Modular code	`def sum(x,y)`	Reusable	None
NumPy Array	Numerical	Fast computation	`np.array()`	Vectorized, memory-efficient	Fixed-size
NumPy Arange	Numerical	Sequence creation	`np.arange(1, 10,1)`	Simple sequence	Integers only unless step float
NumPy Stats	Numerical	Summary stats	`np.mean()`	Fast stats	Requires numeric dtype
Matplotlib	Visualization	Data plotting	`plt.plot()`	Customizable	Verbose syntax

Pandas Series	Data Handling	1D labeled data	`pd.Series()`	Label indexing	More overhead
Pandas DataFrame	Data Handling	Tabular dataset	`pd.DataFrame()`	Excel-like, powerful	RAM-intensive

---

🧠4️⃣ADVANTAGES AND DISADVANTAGES SUMMARY

Category	Advantages	Disadvantages
Python Core	Easy syntax, huge library support	Slower than compiled languages
NumPy	Vectorized speed, foundational for ML	Fixed type arrays, memory heavy on large data
Pandas	Data cleaning, flexible	Slow for massive datasets
Matplotlib	Industry-standard visualization	Verbose, less interactive
Functions	Modular, reusable	None significant

🧩5️⃣RELATION TO MACHINE LEARNING

Foundation Layer	ML Concept it Supports

`NumPy`	Vector/matrix math used in every algorithm (dot products, gradients)
`Pandas`	Data preprocessing, feature engineering
`Matplotlib`	Visualization of regression lines, clusters, decision boundaries
`Python functions`	Algorithm modularization (e.g., custom metrics, losses)
`Input/Output`	User-interactive model testing

✅ **These libraries form the base of ML pipelines — they handle data preparation, exploration, and interpretation before applying ML models like regression, SVM, decision trees, etc.**

🔹 6️⃣ SUPERVISED / UNSUPERVISED CONTEXT (Foundation vs ML)

Layer	Supervised / Unsupervised	Example
Python Core	Neutral — programming base	Input/output functions
NumPy / Pandas / Matplotlib	Neutral — preprocessing/EDA tools	Arrays, DataFrames, Plots
Machine Learning (later)	Supervised/Unsupervised	Logistic Regression, K-Means, PCA