K-Nearest Neighbors (KNN) Classifier — From Scratch

Problem Statement

You are provided with a small dataset of fruits classified based on their **weight**, **size**, and **color**.

The goal is to implement the **K-Nearest Neighbors (KNN)** algorithm **from scratch using only Python and NumPy** (no external libraries like sklearn, pandas, etc.) to classify new fruit samples into one of the three types: **Apple**, **Banana**, or **Orange**.

Toy Dataset

Weight (g)	**Size (cm)**	**Color (0: Yellow, 1: Red, 2: Orange)**	**Fruit Type**
150	7.0	1	Apple
120	6.5	0	Banana
180	7.5	2	Orange
155	7.2	1	Apple
110	6.0	0	Banana
190	7.8	2	Orange
145	7.1	1	Apple
115	6.3	0	Banana

Paste this into your code to start with the data:

```
data = [
   [150, 7.0, 1, 'Apple'],
   [120, 6.5, 0, 'Banana'],
   [180, 7.5, 2, 'Orange'],
   [155, 7.2, 1, 'Apple'],
   [110, 6.0, 0, 'Banana'],
   [190, 7.8, 2, 'Orange'],
   [145, 7.1, 1, 'Apple'],
   [115, 6.3, 0, 'Banana']
]
```

Step 1: Encoding of strings

- Encode the string labels ('Apple', 'Banana', 'Orange').
  Examples of encoding:
  - Label Encoding:
    - 'Apple' → 0
    - 'Banana' → 1
    - 'Orange' → 2
  - One-hot encoding(check it out will be fun)
- Separate the data into:
  - X: Feature matrix (NumPy array)
  - y: Label vector (NumPy array)

Step 2: Euclidean Distance Function

Implement a function that calculates the **Euclidean distance** between two points in N-dimensional space.
This function will be used as a metric to check how "close" two fruits are based on their features.

Step 3: Implement the KNN Classifier

Build a class KNN with the following structure:

- __init__(self, k=3): Stores the value of k.
- fit(self, X, y): Stores the training data
- predict(self, X_test): Returns predictions for an array of test points
- predict_one(self, x): Predicts the label for a single test sample by:
  - Calculating distances to all training samples.
  - Finding the k nearest neighbors.
  - Voting on the most common class among those neighbors.

Step 4: Test Your Classifier

Use the following test samples to evaluate your classifier:

```
test_data = np.array([

   [118, 6.2, 0],  # Expected: Banana
   [160, 7.3, 1],  # Expected: Apple
   [185, 7.7, 2]   # Expected: Orange
])
```

- Create an instance of your classifier with k=3
- Fit it on the training data (X, y)
- Use .predict(test_data) to get the predicted labels
- Print the predictions.

Step 5: Evaluation

- Evaluate the output of your model by comparing it with expected results.
- Try different values of k (e.g., 1, 3, 5) and observe how the predictions change.
- You can even experiment with small tweaks to the dataset and see how sensitive your model is to changes.

Bonus Challenges

Here are some optional extensions you can try - we'll be impressed if you can do some of this, so do give it a shot!

- Implement a simple accuracy checker if true labels are known
- Normalize your features using min-max or z-score normalization
- Add a basic train-test split (e.g., 70-30 or 80-20)
- Try different distance metrics like Manhattan or Minkowski
- Visualize the decision boundaries in 2D using matplotlib
- Implement **weighted KNN**, where closer neighbors have more voting power

Expected Deliverables

Submit a file named knn_classifier.py that includes:

- Data preprocessing logic
- A fully working KNN class
- A test section where predictions on test_data are printed
- Well-commented, modular code

Closing Statements

- Focus on writing clear, correct, and clean code.
- Remember, KNN is conceptually simple:
  Distance → Sorting → Voting
- Visualizing results or testing with multiple values of k can deepen your understanding.