📘 K-Nearest Neighbors (KNN) From Scratch

1. Project Overview

This project presents a full **from-scratch implementation of the K-Nearest Neighbors (KNN)** classification algorithm using Python. It demonstrates the core steps of KNN, including distance calculation, neighbor selection, and majority voting—without using machine learning libraries like scikit-learn for the algorithm itself.

---

2. Dataset Description

The dataset used in this project is the **Cancer Data** from Kaggle.

- **Target column**: diagnosis (Benign B or Malignant M)
- **Input features**: Radius, texture, perimeter, area, smoothness, and other tumor characteristics.

---

3. Data Preprocessing

a. Label Encoding

The diagnosis column is encoded to binary values:

- M → 1 (Malignant)
- B → 0 (Benign)

b. Feature Standardization

To ensure fair distance calculations in KNN, the features are standardized using the **Z-score normalization**:

```
standard_data = (data - mean) / std_dev
```

c. Splitting the Data

The dataset is split into training and testing subsets using train_test_split (70% training, 30% testing).

---

4. Euclidean Distance Function

Implemented as:

```
def euclidean_distance(x1, x2):
  return np.sqrt(np.sum((x1 - x2)**2))
```

This measures the straight-line distance between two points in multi-dimensional space.

---

5. KNN Class - Custom Implementation

This project implements the **K-Nearest Neighbors (KNN)** algorithm from scratch using Python and NumPy, without relying on machine learning libraries like scikit-learn.

The logic is encapsulated in a custom KNN class that performs distance calculation, neighbor selection, and majority voting to make predictions.

Class Definition: KNN

```
class KNN:
  def __init__(self, k):
    self.k = k
```

- The constructor initializes the value of k, which represents the number of nearest neighbors used to make the prediction.

Training Method: fit()

```
def fit(self, X, y):
  self.X_train = np.array(X, dtype=np.float64)
  self.y_train = np.array(y)
```

- This method stores the training data and corresponding labels.
- KNN is a **lazy learner**, so no actual training occurs. It just memorizes the training data to use later during prediction.

Prediction Method: predict()

```
def predict(self, X_test):
  predictions = []
  X_test = np.array(X_test, dtype=np.float64)
  for x in X_test:
    # Step 1: Calculate distances
    distances = np.sqrt(np.sum((self.X_train - x)**2, axis=1))
    # Step 2: Find indices of the k closest neighbors
    nearest_neighbors_indices = np.argsort(distances)[:self.k]
    # Step 3: Retrieve the corresponding labels
    nearest_neighbors_labels = self.y_train[nearest_neighbors_indices]
```

```
        # Step 4: Perform majority voting
        prediction = np.argmax(np.bincount(nearest_neighbors_labels))
        predictions.append(prediction)
    return np.array(predictions)
```

Step-by-Step Explanation:

1. **Distance Calculation**:
   Computes the Euclidean distance from a test point to all training points using vectorized operations:

```
distances = np.sqrt(np.sum((self.X_train - x)**2, axis=1))
```

2. **Neighbor Selection**:
   Sorts all distances and picks the indices of the k closest points:

```
nearest_neighbors_indices = np.argsort(distances)[:self.k]
```

3. **Label Retrieval**:
   Fetches the labels (0 for benign, 1 for malignant) of the nearest neighbors:

```
nearest_neighbors_labels = self.y_train[nearest_neighbors_indices]
```

4. **Majority Voting**:
   Counts the labels and picks the most frequent one using:

```
prediction = np.argmax(np.bincount(nearest_neighbors_labels))
```

5. **Final Prediction**:
   The predicted label is stored in a list and returned as a NumPy array.

---

6. Model Evaluation

After prediction, accuracy is calculated as:

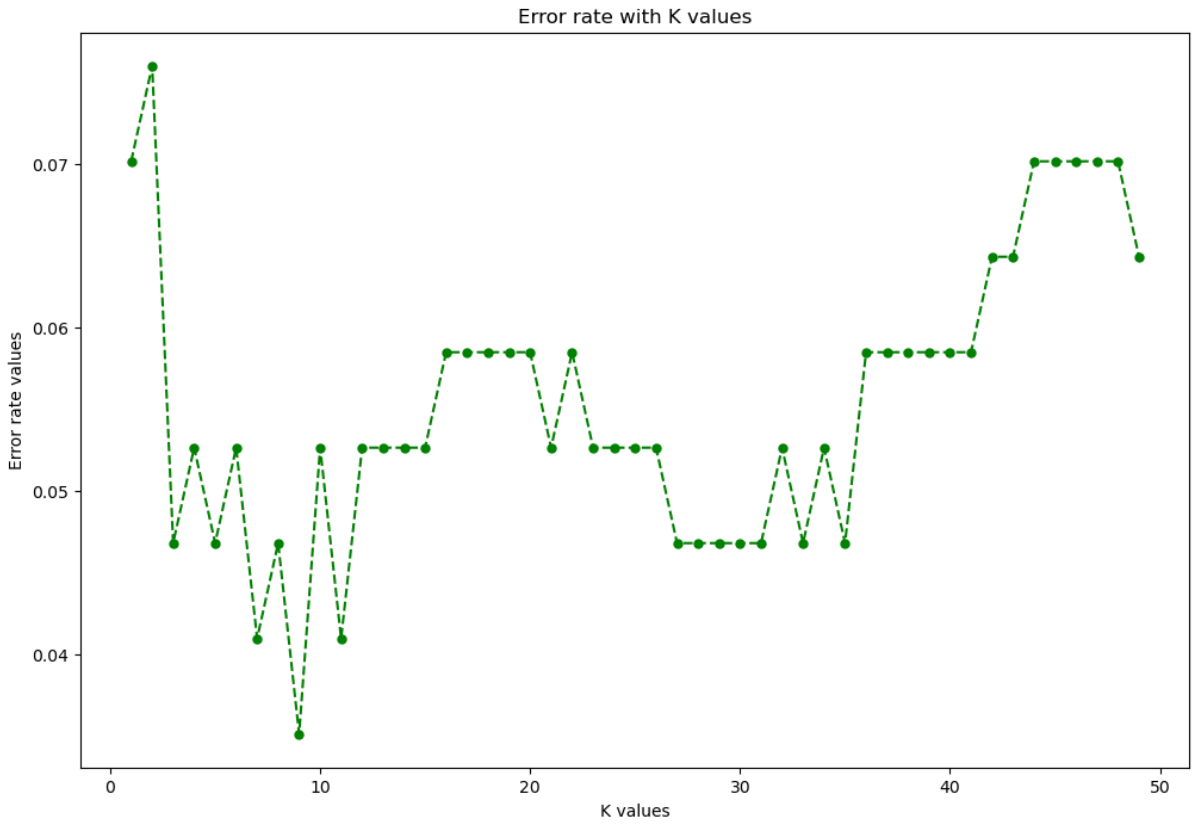
```
acc = np.sum(y_test == predictions) / len(y_test)
```

You also printed actual vs. predicted values for inspection.

---

6. Visualization – Error Rate vs. K Values

The plot illustrates how the error rate of the KNN classifier varies with different values of k, ranging from 1 to 49. The y-axis represents the error rate (i.e., the proportion of incorrect predictions), and the x-axis shows the corresponding k value. Initially, the error rate decreases sharply, reaching its minimum around k = 9, where the model performs best. After this point, the error rate starts to rise gradually, indicating that higher values of k may lead to underfitting. This visualization helps identify the optimal k value that minimizes error and balances model performance.



---

7. Results

The KNN classifier achieved strong performance on the test set. From the confusion matrix, we observe that the model correctly predicted **100 out of 100** instances for class 0 and **65 out of 71** instances for class 1. This results in an overall **accuracy of 96%**. In terms of individual class performance, class 0 achieved a **precision of 0.94**, **recall of 1.00**, and **F1-score of 0.97**, while class 1 achieved **perfect precision of 1.00**, **recall of 0.92**, and **F1-score of 0.96**. The **macro and weighted averages** for F1-score are both **0.96**, indicating balanced performance across classes. These metrics confirm that the model not only maintains high accuracy but also performs well in handling both majority and minority classes.

---

8. Conclusion

This project clearly illustrates:

- The mechanics of implementing the KNN algorithm from scratch
- The role of distance metrics and the impact of choosing different values of *k*
- How error rate analysis helps in tuning hyperparameters effectively

It's a strong foundational example for understanding non-parametric models in machine learning.