writing a classifier from scratch. The main concept is to understand and implement a basic machine learning algorithm rather than just importing it from a library.

Here's a breakdown of the key concepts explained in the tutorial:

• **Building on Previous Work**: The tutorial starts with existing code from Episode 4, which sets up a machine learning pipeline. This pipeline involves importing a dataset (the Iris dataset, which has three types of flowers), splitting it into **training** and **testing** sets, training a classifier on the training data, and then evaluating its **accuracy** on the test data.

• **Replacing a Library Classifier**: The core task is to **comment out the lines that import a pre-built classifier** and instead **write a custom classifier class**, named ScrappyKNN. This class is "bare bones" but sufficient to get the pipeline working.

• **Essential Classifier Methods**: Any classifier needs two main methods:

◦ **fit**: This method performs the **training**. In the case of ScrappyKNN, it primarily involves **storing or "memorising" the training data** (features and labels) within the class.

◦ **predict**: This method takes the features of the **testing data** as input and **returns predictions for their labels**.

• **Initial "Random Classifier"**: To quickly get the pipeline working and understand the methods, a **simple "random classifier" is first implemented**. In this version, the predict method simply **guesses a label randomly** from the training data for each test example. For the Iris dataset, this results in an accuracy of about **33%**.

• **Implementing k-Nearest Neighbors (KNN)**: The primary goal is to improve accuracy to over 90% by implementing a classifier based on the **k-Nearest Neighbors algorithm**. The intuition behind KNN is as follows:

◦ For a given **testing point**, the algorithm finds the **closest training point** (its "nearest neighbor").

◦ It then **predicts that the testing point has the same label as its closest neighbor**.

◦ The "k" in k-Nearest Neighbors refers to the **number of neighbors considered** when making a prediction. If k=1, only the single closest point is used. If k is greater than 1 (e.g., k=3), the algorithm looks at the k closest points and **predicts the majority class among them** through a vote.

• **Euclidean Distance**: To determine the "closest" point, the tutorial explains the concept of **Euclidean Distance**. This formula measures the **straight-line distance between two points** and is similar to the Pythagorean Theorem. A key insight is that the Euclidean Distance formula

works the same way regardless of the number of features or dimensions a dataset has. The scipy library is used for its implementation.

• **KNN Algorithm Steps**: To implement the KNN classifier (initially with k hard-coded to 1, meaning it's a nearest neighbor classifier), the predict method performs these steps for each test point:

◦ It **calculates the Euclidean distance from the test point to *all* training points**.

◦ It **keeps track of the shortest distance found so far** and the **index of the corresponding training point**.

◦ Finally, it **returns the label of the training example that had the shortest distance** (the closest one).

• **Outcome and Significance**: After implementing the k-Nearest Neighbors classifier, the accuracy returns to **over 90%**, demonstrating that a functional classifier can be written from scratch. The tutorial emphasizes that being able to code and understand this simple classifier is a **big accomplishment**.

• **Pros and Cons of KNN**: The tutorial briefly touches on the characteristics of the k-Nearest Neighbors algorithm:

◦ **Pros**: It's **relatively easy to understand** and **works reasonably well** for some problems.

◦ **Cons**: It can be **slow** because it needs to iterate over every training point to make a prediction. Additionally, it doesn't easily account for some features being more informative than others.

• **Future Directions**: The tutorial concludes by mentioning that more complex classifiers, such as **decision trees** and **neural networks**, are better suited for learning more intricate relationships between features and labels.