**K-Nearest Neighbors (KNN) Classification Tutorial with the Iris Dataset**

**Introduction**

Welcome, learners!

In this tutorial, we’ll take a deep dive into the **K-Nearest Neighbors (KNN)** algorithm — a classic machine learning approach that’s surprisingly powerful despite its simplicity. KNN is widely taught in introductory data science courses not only because it’s easy to understand but also because it lays a strong foundation for grasping how machine learning models make decisions based on **proximity**, **pattern recognition**, and **voting mechanisms**.

Our objective here is to go beyond just using the algorithm — we aim to understand:

* How KNN works under the hood
* Why it behaves the way it does
* How to evaluate it effectively
* Where it shines and where it struggles

To do this, we’ll pair theory with hands-on code using the **Iris dataset**, which is ideal for visual learning and small-scale experimentation.

****

**Section 1: Understanding KNN – Concept and Theory**

**Real-Life Analogy**

To make KNN relatable, let’s start with an analogy:

Imagine you move into a new neighborhood and want to find out which gym to join. You don’t know the area well, so you ask the 5 nearest neighbors (k=5) where they go. If 3 of them say “FitClub” and 2 say “BodyZone”, you’ll likely go with the majority — FitClub.

That’s how **KNN classifies**: It looks at the k closest labeled examples and chooses the most common label among them.

This proximity-based decision-making mimics human reasoning and is why KNN is referred to as a **memory-based** or **instance-based** learner.

**What is KNN?**

KNN is a **supervised learning** algorithm used for **classification** and **regression** problems. What sets it apart from other machine learning algorithms is that it is a **lazy learner**. That means it **does not build a model during training** — it simply stores the training data and waits until a prediction is requested.

When a new data point needs to be classified, KNN:

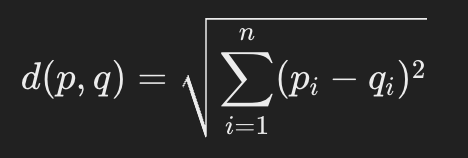
1. Measures the distance from the new point to every point in the training set.
2. Selects the k nearest neighbors.
3. Assigns the most common class label among those neighbors.

This approach makes KNN very simple and transparent, but it can be computationally heavy during prediction.

**Distance Metrics**

KNN relies heavily on how we define “closeness”. The most used metrics are:

* **Euclidean Distance**:



This is like the straight-line distance in geometric space.

* **Manhattan Distance**:

A black background with white letters and numbers

AI-generated content may be incorrect.

Also known as “taxicab” distance, useful when movement is restricted to grids.

Choosing the right metric is essential, and it often depends on the domain and data characteristics.

* **Tip:** **Always standardize your features** before applying KNN. If one feature has a much larger scale than others, it will dominate the distance calculation.

**Choosing the Right k**

k determines how many neighbors KNN consults to decide. This is a critical hyperparameter, and choosing it wisely is key to achieving good performance.

* **Small k (e.g., 1 or 3)**: May result in a model that is too sensitive to noise — high variance, low bias.
* **Large k (e.g., 15 or 25)**: Averages across many points, which may smooth over important local patterns — low variance, high bias.

Often, we use **cross-validation** to find the optimal k. Choosing an **odd number** is helpful when dealing with binary classification to avoid ties in voting.

**Pros and Cons**

Let’s summarize KNN’s strengths and weaknesses.

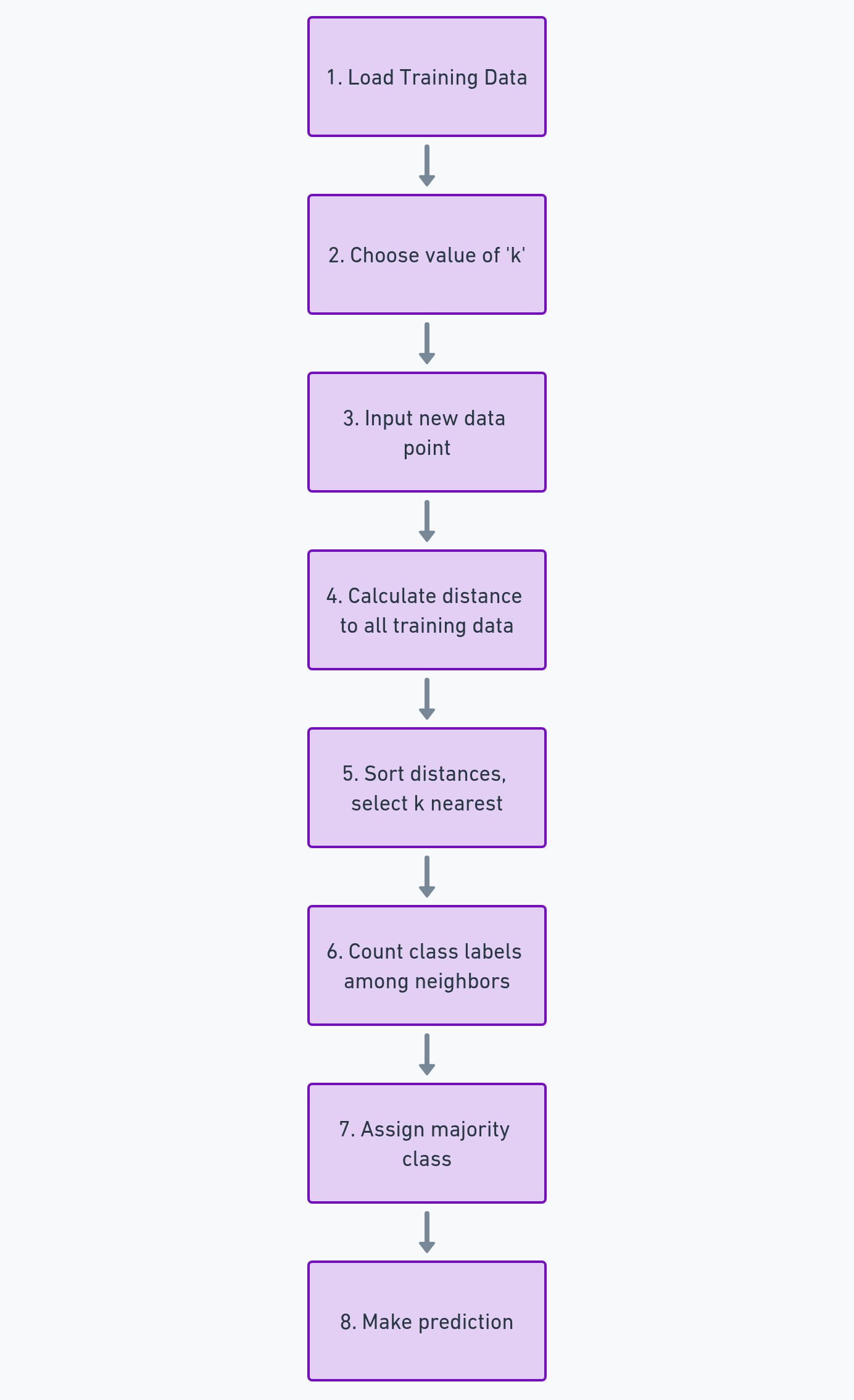
|  |  |
| --- | --- |
| **✅ Pros** | **❌ Cons** |
| Easy to implement and explain | Prediction is slow for large datasets |
| Naturally handles multi-class classification | Sensitive to noisy or irrelevant features |
| No training time | Suffers from the curse of dimensionality (many features) |
| Makes no assumptions about data distribution | Requires careful feature scaling |

**Lazy vs Eager Learning**

KNN belongs to the category of **lazy learners**. It doesn’t build an internal model or learn coefficients — instead, it **memorizes** the training data. In contrast, algorithms like Decision Trees or Logistic Regression are **eager learners** that build general models during training.

This makes KNN fast to “train” but potentially slow and memory-heavy during prediction.

**KNN Classification Process Diagram (Step-by-Step Workflow)**



**Explanation of Each Step:**

1. **Load Training Data**: This is your labeled dataset. Each data point has features and a class label.
2. **Choose Value of k**: You select how many neighbors to consider (e.g., k = 3 or 5).
3. **Input New Data Point**: You want to predict the label for a new, unseen data point.
4. **Calculate Distances**: Compute the distance between the new point and every point in the training data.
5. **Sort and Select Neighbors**: Find the k points that are closest (based on the chosen distance metric).
6. **Count Labels**: Among the k neighbors, tally how many belong to each class.
7. **Assign Majority Class**: Choose the class that appears most frequently.
8. **Prediction**: That class label becomes the model’s prediction for the new data point.

**Section 2: The Iris Dataset – Overview and Suitability**

**Why Use Iris?**

The Iris dataset is:

* Small (150 samples)
* Balanced (50 examples per class)
* Well-known and benchmarked
* Easy to visualize in 2D
* Built into scikit-learn

It contains four features: **sepal length**, **sepal width**, **petal length**, and **petal width** — all in centimeters — used to classify a sample as one of three species: **Setosa**, **Versicolor**, or **Virginica**.

The Iris dataset is ideal for learning KNN because it allows us to clearly see how changes in k and feature scaling affect model performance.





**Section 3: Code Walkthrough and Evaluation**

We now move into the hands-on portion of our tutorial. Each code block is preceded by markdown to explain its purpose.

**1. Import Libraries**

A computer screen with white text

AI-generated content may be incorrect.

This code imports essential libraries and modules for building, training, and evaluating a K-Nearest Neighbors (KNN) classifier using the Iris dataset. It includes tools for data handling, preprocessing, visualization, model evaluation, and ROC curve analysis.

**2. Load Dataset and Create DataFrame**

A screen shot of a computer code

AI-generated content may be incorrect.

A screenshot of a graph

AI-generated content may be incorrect.

We start by loading the famous Iris dataset using load\_iris(). Then, we separate the **features** (X) and the **target labels** (y). To make data exploration easier, we create a **Pandas DataFrame** with the feature names as column headers and add a new column called 'species' that holds the label for each flower. Finally, df.head() lets us preview the first few rows of our dataset.

A screenshot of a graph

AI-generated content may be incorrect.

df['species'].value\_counts().sort\_index().rename(index={0:'setosa', 1:'versicolor', 2:'virginica'})

A screenshot of a cell phone

AI-generated content may be incorrect.

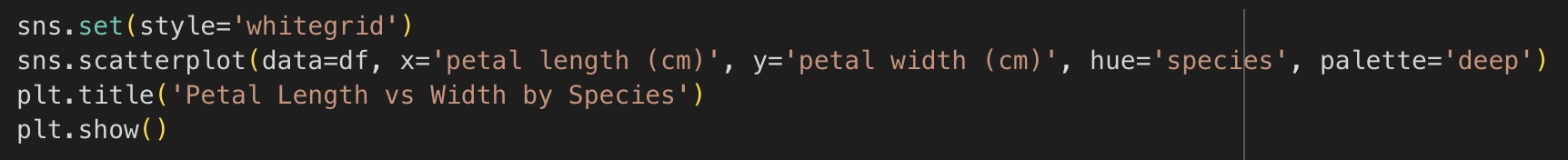
In this line, we're counting how many samples we have for each species in the dataset using value\_counts(). Then, we use sort\_index() to ensure the classes are listed in the correct order based on their original label (0, 1, 2). Finally, rename() replaces those numerical labels with the actual species names — **setosa**, **versicolor**, and **virginica** — so it’s easier to understand.

And from the output, we can see that each species has exactly **50 samples**, which tells us the dataset is perfectly balanced.

**Visualize Feature Distributions**

First, we set the Seaborn style to "whitegrid" to make our plot background cleaner and easier to read. Then, we create a **scatter plot** using sns.scatterplot() where the x-axis represents **petal length** and the y-axis represents **petal width**. We add the hue='species' argument so that each flower species is color-coded differently, making the clusters visually distinct.

Finally, plt.title() gives our plot a descriptive heading, and plt.show() displays the chart.  
From the plot, you can clearly see how the species are separated in terms of petal size — and this visual separation is exactly what helps classifiers like KNN work effectively!



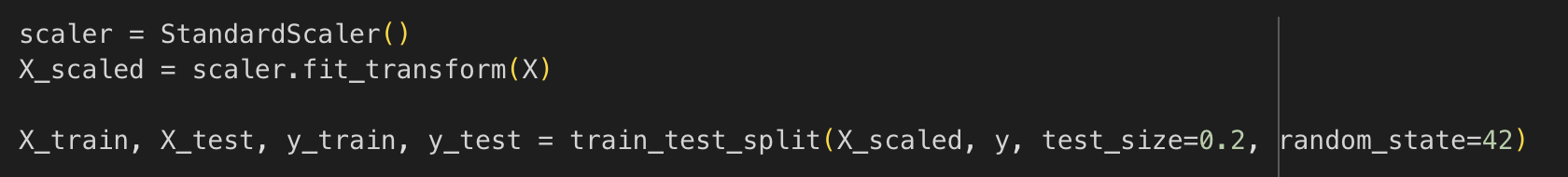
A graph of different species

AI-generated content may be incorrect.

**Split and Scale Data**

In this part of the code, we're getting our data **ready for modeling**. First, we apply StandardScaler() to normalize the feature values. This means we’re transforming the data so that each feature has a **mean of 0 and a standard deviation of 1**, which is super important for distance-based algorithms like KNN.

Next, we split the scaled data into training and testing sets using train\_test\_split(). We keep **20% of the data for testing** (test\_size=0.2) and use random\_state=42 to ensure reproducibility — so every time we run the code, we get the same split.



**Train KNN Classifier**

We start by creating a K-Nearest Neighbors classifier with n\_neighbors=3, which means the model will look at the 3 closest data points to decide the class of a new sample.  
Then, we **train the model** using knn.fit() with our training data. After that, we use the trained model to **predict** the class labels for our test data with knn.predict() — and those predictions are stored in y\_pred.

Simple and powerful, right? This is the full modeling step in just three lines!

A black background with white text

AI-generated content may be incorrect.

**6. Model Evaluation: Accuracy and Confusion Matrix**

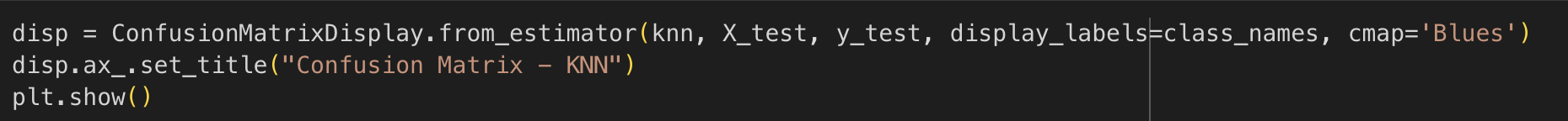
A screen shot of a computer code

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

**7. Confusion Matrix Visualization**



A blue squares with white text

AI-generated content may be incorrect.

Here, we’re using ConfusionMatrixDisplay.from\_estimator() to **visualize the performance** of our trained KNN model. It shows how well the model predicted each class by comparing the actual labels (y\_test) with the predicted ones. We pass class\_names to label the axes and set the color map (cmap) to 'Blues' for a clean look.

From the confusion matrix output, you can see that the model predicted **all test samples correctly** — 10 setosa, 9 versicolor, and 11 virginica — which means our classifier performed perfectly on this test set!

**8. Multiclass ROC Curve with AUC**

A screen shot of a computer program

AI-generated content may be incorrect.

A graph with a line

AI-generated content may be incorrect.

Each colored line (blue, green, red) represents one class’s ROC curve. Since **all AUC scores are 1.00**, this tells us that our KNN classifier perfectly separates each class on the test set — an ideal scenario.

The dashed diagonal line represents a random guess. The fact that all colored curves hug the top-left corner means the model's performance is **outstanding**!

**Accessibility Features**

To ensure this tutorial is accessible:

* Colorblind-safe palettes (e.g., Blues in confusion matrix)
* Clear labels and titles on plots
* All visuals explained in markdown for screen readers
* Logical notebook structure with readable font sizes
* PEP-8 coding style (easy to follow and modify)
* Alternative text-style descriptions for diagrams (via captions)

These design choices allow learners with visual, motor, or auditory impairments to follow along using tools like screen readers and voice control.

**GitHub Repository and Submission Structure**

|  |  |
| --- | --- |
| **File** | **Description** |
| knn\_iris\_tutorial\_enhanced.ipynb | Full tutorial notebook |
| README.md | Instructions, dependencies, objectives |
| LICENSE | MIT open-source license |
| requirements.txt | List of packages to install |
| tutorial.docx / tutorial.pdf | Final report for submission |

**Repository Links**

|  |  |
| --- | --- |
| **Component** | **GitHub Link** |
| Notebook | <https://github.com/your-username/knn-iris/blob/main/knn_iris_tutorial_enhanced.ipynb> |
| README | <https://github.com/your-username/knn-iris/blob/main/README.md> |
| LICENSE | <https://github.com/your-username/knn-iris/blob/main/LICENSE> |
| Tutorial Report | <https://github.com/your-username/knn-iris/blob/main/tutorial.pdf> |

**Academic References**

* T. Cover and P. Hart, "Nearest neighbor pattern classification," in IEEE Transactions on Information Theory, vol. 13, no. 1, pp. 21-27, January 1967,<https://doi.org/10.1109/TIT.1967.1053964>.
* Altman, N. S. (1992). An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression. The American Statistician, 46(3), 175–185. <https://doi.org/10.1080/00031305.1992.10475879>
* Géron, A. (2019). Hands-on machine learning with scikit-learn, keras, and tensorflow: concepts. Aurélien Géron-Google Kitaplar, yy <https://books.google.com.tr/books>.
* Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning*. Springer.