K-NN Algorithm

Authors:

Krystian Ruszczak

Paweł Michalcewicz

Contents

Introduction

The goal of this project is to implement the k-Nearest Neighbors (k-NN) classification algorithm from scratch using Python and to compare the results with the reference implementation available in scikit-learn library.

Despite its simplicity, k-NN is still widely used as a baseline classifier in many practical applications. Implementing it manually helps in understanding key concepts such as distance metrics, neighborhood selection, and data-driven decision making.

K-NN Algorithm Overview

The k-nearest neighbors algorithm (k-NN) is a non-parametric supervised learning method used for both classification and regression (also known as nearest neighbor smoothing). It belongs to lazy learning algorithms, meaning that it does not build an explicit model during training. Instead it stores the training data and makes predictions only when new input samples are provided.

To classify new data, the algorithm looks at the k closest samples in the training set and assigns the class that appears most frequently among them. The closeness of two data points is measured using a distance metric. A commonly used distance metric for continuous variables is Euclidean distance, which is used also in this project.

For two dimensions in the Euclidean plane, let point p have Cartesian coordinates (p1, p2) and let point q have coordinates (q1, q2). Then the distance between p and q is given by:

$$d(p,\ q)\ =\ \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2}$$

The algorithm calculates this distance between the input sample and every sample in the training set.

After computing the distance, the next step is to determine which of the samples in the training set are actually the closest ones to the input sample. All calculated distances are sorted in ascending order. Then from this ordered list, the first k elements are chosen - these represent the k nearest neighbors. Their class labels are then inspected and the most frequent label among them is taken as the final prediction. This decision-making mechanism is commonly called “majority voting”, since the algorithm simply picks the class occurring between most of the selected neighbors.

A characteristic feature - and sometimes even a downside - of the k-NN algorithm is that it is very sensitive to the local structure of the data. In k-NN classification, the model does not learn a global decision function during training. Instead, everything happens when a new sample needs to be classified. Because the algorithm depends heavily on distance calculations and k number, it can perform poorly if the features use different physical units or have very different scales. When k is small, such as 1 or 3, the model becomes sensitive to noise or outliers because even a single unusual data point can influence the prediction. On the other hand, using a larger k smooths the decision boundaries, producing a more stable classifier. However, very large values of k can also lead to incorrect predictions, especially when the dataset contains imbalanced classes.

Implementation

The implementation of this version of algorithm is contained in a single class *KNNClassifier*, whose structure resembles the interface of KNeighborsClassifier from sklearn.neighbors, limiting the functionality to the essential components. The class stores the number of neighbors (n_neighbors) and the training dataset (X_train and y_train). The class contains following methods:

- fit,
- _euclidean_distance,
- _predict_single,
- predict,
- score.

The fit() method does not perform any parameter optimization or internal model construction. Instead, it simply stores the training samples and their labels for later use during classification. The private method _euclidean_distance() handles distance computations with implementation of standard Euclidean metric. The prediction process for a single sample is encapsulated in _predict_single(). For an input vector, the method calculates distances to all stored training samples, pairs each distance with the corresponding class label, and sorts the pairs in ascending order. The first k entries in this sorted list represent the nearest neighbors. Their labels are extracted and subjected to a majority voting scheme, where the most frequent class becomes the final prediction. For batch prediction, the predict() method iterates over all input samples and applies _predict_single() to each one. Finally, the classifier provides a score() method for computing accuracy by comparing predicted labels with ground truth.

Comparison with implementation from scikit-learn library

To evaluate the correctness and efficiency of the custom implementation of the k-Nearest Neighbors classifier, it was compared against the reference implementation available in the scikit-learn library. Four well-known datasets were selected for the experiment:

- Iris,
- Wine,
- Breast Cancer,
- Digits.

For each dataset, models were tested using five values of k: 1, 3, 5, 7 and 9. The comparison included classification accuracy, prediction consistency between both implementations, and computational performance (time of training, prediction and scoring).

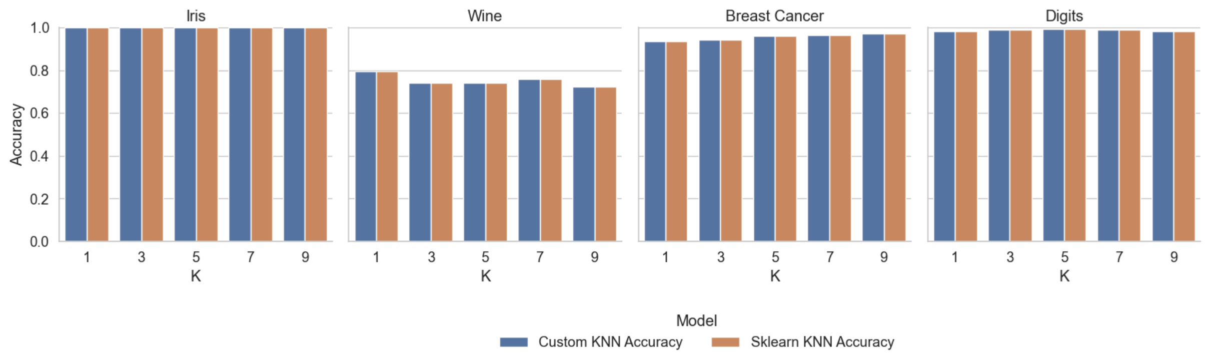
Accuracy and Prediction Consistency

Across all tested datasets and all values of k, the custom implementation produced identical predictions to those generated by sklearn’s KNeighborsClassifier. This means that:

- predicted class labels were exactly the same,
- the number of differing predictions was zero,
- the accuracy achieved by both implementations was identical.

This confirms that the custom classifier faithfully reproduces the logic of scikit-learn’s KNN algorithm, including distance computation, neighbor selection and majority voting. The accuracy values observed across datasets align with expectations:

- Iris: 100% for all k, due to its clear class separation.
- Wine: moderate accuracy (0.72–0.79), affected by the small dataset size and feature variability.
- Breast Cancer: high accuracy (0.93–0.97).
- Digits: very high performance (0.98–0.99).



Performance Comparison

A major difference between both implementations appears in computational performance.

Training Time

Training time for both implementations is minimal, because KNN is a lazy-learning algorithm. The custom implementation is slightly faster (microseconds vs. milliseconds), which is expected because it only stores references to arrays.

Prediction and Scoring Time

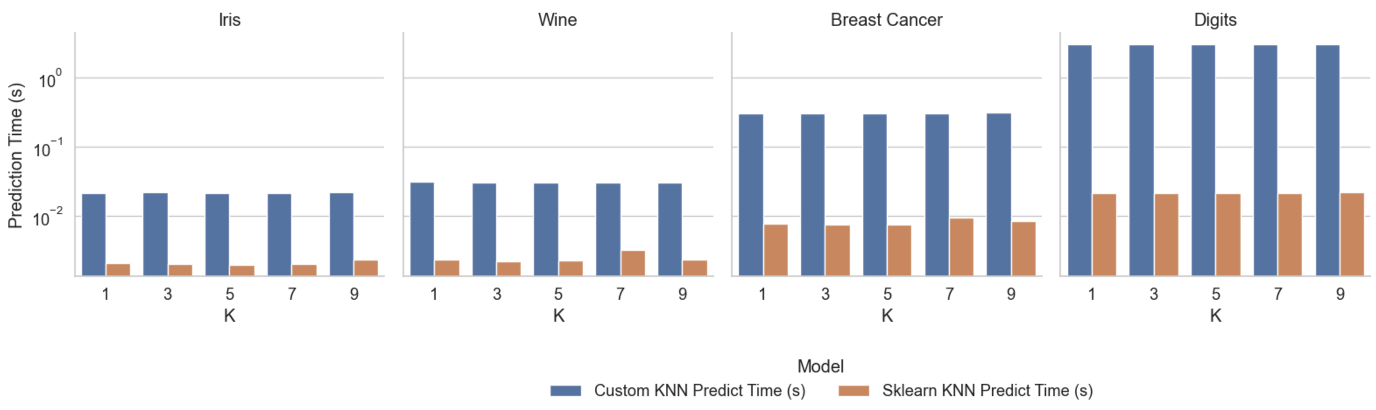
The most significant discrepancy appears in prediction speed:

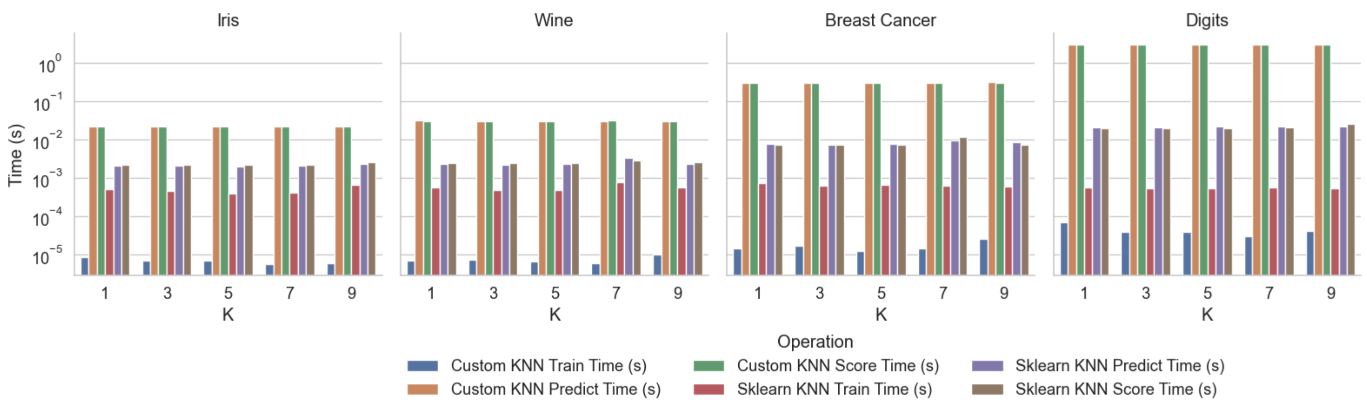
- For small datasets (Iris, Wine), the custom KNN is slower than scikit-learn by roughly one order of magnitude.
- For medium datasets (Breast Cancer), the difference grows to ~40 times slower.
- For the largest dataset (Digits), the difference becomes dramatic: Custom KNN prediction time ~3.0 seconds, scikit-learn prediction time ~0.02 seconds.

The slowdown results from:

- the Python-level loop computing Euclidean distance sample-by-sample,
- lack of vectorization,
- absence of efficient data structures,
- pure Python sorting for each input sample.

In contrast, scikit-learn uses optimized C implementations and specialized search structures, which drastically reduce computation time.





Conclusions

The custom implementation fully reproduces the behavior of scikit-learn's KNN classifier. All predictions match exactly across all datasets and all tested values of k. Both implementations obtain identical accuracy results, confirming that the core logic of KNN was correctly replicated. Although the custom classifier is functionally correct, it is significantly slower than the optimized scikit-learn implementation, especially on larger datasets. The performance gap increases with dataset size due to non-vectorized Python loops. The custom implementation is sufficient for educational purposes and small datasets, but is not suitable for practical use with larger datasets due to computational inefficiency.

Testing

To ensure correctness, robustness, and full functional compatibility of the custom KNN implementation, a comprehensive set of unit and integration tests was prepared using the pytest framework. Testing was divided into two levels:

- Unit Tests – verify isolated components such as initialization, data storage, distance calculation, and prediction logic on simple synthetic datasets.
- Integration Test – compares the entire model workflow against the scikit-learn reference implementation using the Iris dataset.

Initialization Test

This test verifies whether the classifier correctly stores the value of the parameter n_neighbors:

- checks default initialization (n_neighbors = 3),
- checks custom initialization using a manually passed parameter.

This ensures the constructor behaves as expected.

Fit Method Test

The test validates whether the .fit() method correctly stores the training data without modification - compares internal X_train and y_train with the input arrays using np.array_equal. Since KNN is a lazy learner, proper storage of the dataset is essential.

Parametrized Prediction Logic Test

A parametrized test checks prediction correctness for specific points using a simple, manually crafted dataset.

Tested scenarios include:

- a point near class 0,
- a point near class 1,
- a point placed in the middle.

For k = 1, predictions were verified against expected class labels. This confirms correctness of Euclidean distance computation, selection of nearest neighbor and prediction process.

Euclidean Distance Mathematical Test

A direct mathematical test validates correctness of the _euclidean_distance() method.

Distance between points (0,0) and (3,4) equals 5. This ensures the implementation of the core distance function is mathematically correct.

Integration Test Against Scikit-Learn

A full workflow test compares the custom KNN classifier against scikit-learn's KNeighborsClassifier:

- dataset: Iris
- split: train/test (80/20)

- K = 3
- scikit-learn model configured with algorithm='brute' to match the logic of the custom implementation.

The test computes the percentage of matching predictions between both models.

A threshold of 90% prediction match was required, but the actual match was 100%, confirming full consistency with scikit-learn. This validates end-to-end prediction behavior, handling of real multi-class data and compatibility with established implementations.

Results

All tests completed successfully, confirming the reliability and correctness of the custom KNN implementation. The model behaves as expected at every level: it initializes with the correct parameters, stores training data without modification, computes Euclidean distances accurately, and produces valid predictions across a range of synthetic and real inputs. Most importantly, the integration test demonstrated complete agreement with the scikit-learn reference classifier on the Iris dataset, which provides strong evidence that the overall algorithmic logic has been implemented correctly.