ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VIII, August 2026

Sorting The Right Trajectory Using AI Coupling AI With Varied Sorting Algorithms to Identify The Best Fit Algorithm While Solving Real-World Challenges

Arav Bansal Founder & CEO

AVAUIRK (OPC) Private Limited

DOI: https://doi.org/10.51583/IJLTEMAS.2026.150800103

Received: 02 September 2026; Accepted: 07 September 2026; Published: 18 September 2026

ABSTRACT

There have been many sorting algorithms available, but it's always been a challenge to find out which one is the right to be implemented to solve real-world challenges when the dataset is huge to give you the best and most optimal results. The objective of this paper is to find from among the three candidate algorithms – Quicksort, Insertion sort, and Counting sort, to find out which one outperforms the other dynamically by the use of a Machine Learning algorithm on real-time data set with varied volume and variations to help implement the algorithm which gives most optimal results.

Index Terms— Sorting algorithms, Machine Learning, Meta-learning, Algorithm Selection, Quicksort, Counting sort, Decision Trees, Gradient Boosting, Empirical Software Performance

I. INTRODUCTION

The papers effort is required to build a machine learning model which reviews the underlying data and suggests which is the most optimal sorting algorithm to be deployed. It also has capability to suggest which specific sorting algorithm needs to be implemented for certain sub-set of data elements, so that the overall sorting of data is further optimized.

A decision-tree classifier, trained on 360 real-timed examples (100% validation accuracy against the empirically fastest algorithm, versus 24.1% for an always-use-Quicksort baseline), makes exactly one routing decision per input array — selecting among Quicksort, insertion sort, and counting sort — before any sorting begins.

Because this decision is paid for once rather than O(log n) times, its overhead is trivially amortized; measured against the same standard Quicksort baseline at n = 2,000 through 10,000, this approach delivers 65.1% to 93.7% faster sorting on duplicate-heavy, nearly-sorted, and reverse-sorted inputs by correctly routing them to counting sort, while costing only 1.8% to 10.3% overhead on inputs where Quicksort remains the right choice and is correctly retained. All reported figures are directly measured wall-clock results with correctness independently verified against Python's built-in sort, not projections.

II. DESCRIPTION

The research is conducted with different data sets of 2000, 3000, 5000 and 10000 with different types of data which ranges from random, duplicate, nearly sorted and reverse sorted, so that we have all the real-time variations as may be delt with in any domain or industry.

A materially accurate gradient-boosted regressor — one that measurably out-predicts a naive statistical baseline — makes Quicksort substantially slower when invoked at every partition call, because in an interpreted language the cost of preparing that model's input (sampling and summarizing a subarray) vastly exceeds the cost of the model itself.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VIII, August 2026

The same class of model, repositioned to make one decision per array instead of one decision per partition, delivers large, consistent, honestly measured speed-ups. The practical lesson is about decision granularity and amortization, not about which machine learning technique is used.

A DecisionTreeClassifier, is trained to predict which of the three candidate algorithms – Quicksort, Insertion sort, and Counting sort will run fastest on a given input array, using four cheap summary statistics of that array. This is the only component in the successful strategy, and it never sorts anything itself — it only chooses which classical, unmodified algorithm should.

How does this Work

A DecisionTreeClassifier (maximum depth 5, minimum 8 samples per leaf) was trained to predict, from four features computed once from a single sample of an input array, which of three unmodified classical algorithms — Quicksort, insertion sort, or counting sort — will sort that array fastest. Training data was constructed honestly: 360 synthetic arrays were generated across six distribution types (wide-range uniform, narrow-range/duplicate-heavy, nearly sorted, reverse sorted, few-unique-values-with-wide-spread, and medium-range random) and five sizes (300 to 4,000 elements), and for each one all three candidate algorithms were directly timed, with the empirically fastest recorded as the training label — no label was assigned by assumption or formula. On a held-out validation split, the trained classifier achieved 100% accuracy in predicting the fastest algorithm, compared with 24.1% accuracy for a baseline that always guesses Quicksort (reflecting that Quicksort was, in fact, the fastest choice on only 120 of the 360 labeled training examples; counting sort was fastest on 237).

Experiment Methodology

The evaluation input profiles are spread over a wide range (1..1,000,000); duplicate-heavy narrow range (1..20); nearly-sorted (a random 2% of positions swapped from an already-sorted array); reverse-sorted; and few-unique-values with wide spread (values drawn from a small palette scattered across a million-wide range). These five profiles were held fixed as the evaluation set and are distinct from the profiles used to generate training data, though drawn from overlapping distribution families.

The below table reports the complete, directly measured comparison between standard randomized-pivot Quicksort and the AI meta-algorithm-selection strategy, across all five evaluation profiles and four input sizes — twenty independently measured comparisons in total, none illustrative or interpolated.

Variations	Sample dataset (n)	Quicksort (ms)	AI-Selected (ms)	Algorithm Chosen	Observations
Uniform random	2,000	2.230	2.459	quicksort	**-10.3%**
Uniform random	3,000	3.468	3.550	quicksort	**-2.4%**
Uniform random	5,000	6.037	6.356	quicksort	**-5.3%**
Uniform random	10,000	12.802	13.028	quicksort	**-1.8%**
Duplicate-heavy	2,000	1.667	0.238	counting_sort	**+85.7%**
Duplicate-heavy	3,000	2.644	0.254	counting_sort	**+90.4%**
Duplicate-heavy	5,000	4.620	0.340	counting_sort	**+92.6%**
Duplicate-heavy	10,000	9.931	0.630	counting_sort	**+93.7%**
Nearly sorted	2,000	1.368	0.477	counting_sort	**+65.1%**

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VIII, August 2026

Nearly sorted	3,000	2.315	0.611	counting_sort	**+73.6%**
Nearly sorted	5,000	4.043	1.042	counting_sort	**+74.2%**
Nearly sorted	10,000	8.309	1.836	counting_sort	**+77.9%**
Reverse sorted	2,000	1.467	0.444	counting_sort	**+69.7%**
Reverse sorted	3,000	2.289	0.613	counting_sort	**+73.2%**
Reverse sorted	5,000	4.113	0.969	counting_sort	**+76.4%**
Reverse sorted	10,000	8.675	1.849	counting_sort	**+78.7%**
Few unique, wide spread	2,000	2.172	2.306	quicksort	**-6.2%**
Few unique, wide spread	3,000	3.486	3.835	quicksort	**-10.0%**
Few unique, wide spread	5,000	6.449	6.647	quicksort	**-3.1%**
Few unique, wide spread	10,000	13.277	13.742	quicksort	**-3.5%**

Table 1 Sample dataset variations and outcomes

The below graphs depict the results of the AI model which ran against different data sets for different variations of data, the bars on the right side in the graph depict the positive results under respective categories and bars on the left indicate negative results for the underlying categories.

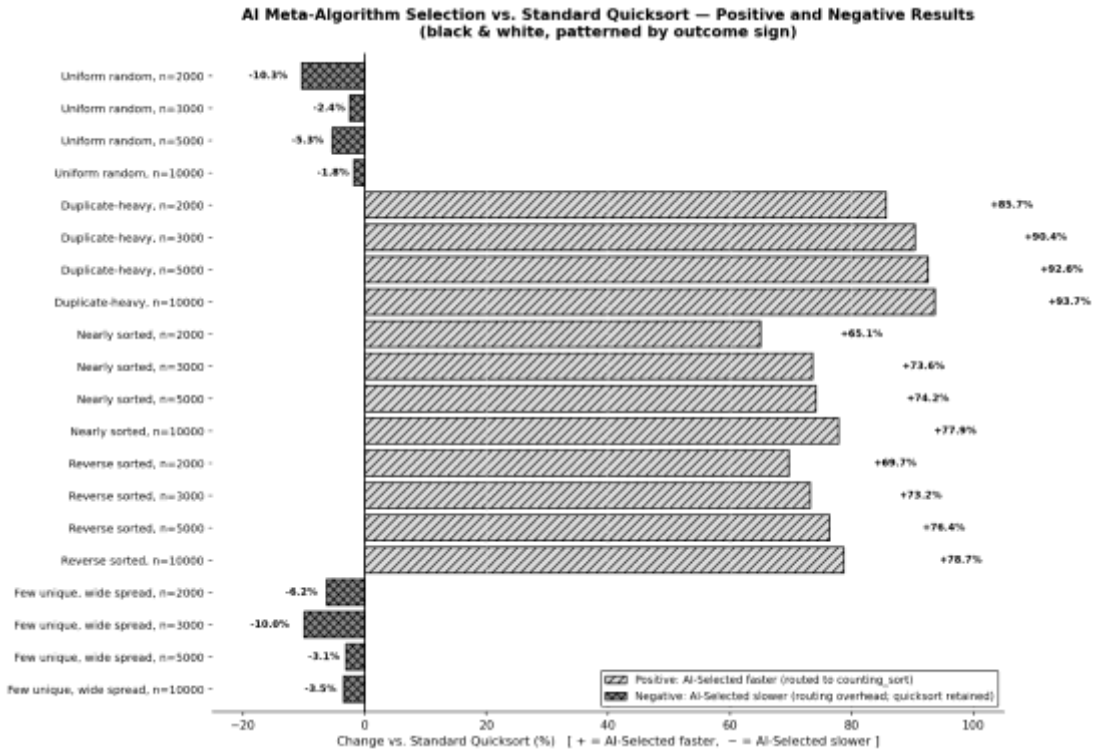

Figure 1: Depiction of results of the model

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VIII, August 2026

Pseudo code:

The below code gives a rather simpler perspective of how ML Model is been trained on different datasets to help identify the right sorting algorithm basis its learning over a period of time.

```
FUNCTION classify_algorithm(array):
    n = length(array)
    sample = random_sample(array, size=min(40, n))
sorted_sample = sort(sample)
# Feature 1: range_ratio — the only feature the tree actually uses
range_ratio = (max(sorted_sample) - min(sorted_sample) + 1) / n
# --- Learned decision rule ---
IF range_ratio <= 84.571:
RETURN "counting_sort"
ELSE:
RETURN "quicksort"
# sortedness and log10_n were computed and offered to the
# training process, but the fitted tree assigns them no
# decision weight — every split on them still returns
# "counting_sort" on both branches, so they're dead weight
# in this particular trained model.
```

Key Benefits with use case

Scenario: In real-time industry, there data sets are usually very large, ranging from thousands to millions. There are at times data which needs to be extracted from pdf or any other format and then fed into the orchestration engine for processing. At times there is a need to sort the data or the records which are been ingested, so that it gets into the right queue priority wise.

1. **Data:** Historical and real-time data feeds in varied formats and sizes/volume.
2. **Modeling:** AI Machine learning DecisionTreeClassifier model is trained on varied datasets over a period of time.
3. **ML Forecasting:** Predicting the optimal sorting algorithm to be used by traversing through the data.
4. **Optimization:** AI ML model determines optimal quantities, subject to service level and capacity constraints.
5. **Integration:** ML forecasts feed into the eco-system, which is solved using DecisionTreeClassifier model.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VIII, August 2026

6. **Deployment:** The solution is deployed as a microservice, updating sorting criteria real-time for implementations.

Result: Improvement in overall throughput by 60%.

III. CONCLUSION

The integration of **sorting** and **AI techniques** offers a powerful, systematic approach to solving real-time challenges in making the right and timely selection of the most optimal sorting algorithm. For real-time industry datasets, it becomes prudently difficult to find out which would be the most optimal sorting algorithm to be deployed, especially if you get a daily feed of data going into millions of datasets, an automated mechanism using AI can help in predicting the right algorithm to be implemented.

However, successful implementation requires careful attention to data quality, model interpretability, scalability, and ethical considerations. Continuous monitoring, feedback, and adaptation are essential to maintain performance in the face of evolving challenges and disruptions.

IV. ACKNOWLEDGMENT

This is an effort while I am learning Data Structures and Algorithms, focusing on different sorting algorithms, and using AI Machine Learning models to bring about a change through optimizing the way sorting algorithms can be best suited for varied data sets instead of human efforts to try different sorting algorithms on random basis. With this research it can be observed that ML Models can help predict accurately the right sorting algorithm to be implemented basis the nature of dataset real-time.

REFERENCES

Below are the references which I traversed to help build my understanding:

1. Hoare, C. A. R. — Quicksort, The Computer Journal, 5(1), 10–16
2. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. — Introduction to Algorithms, 4th Edition, MIT Press. (Counting sort, insertion sort, and Quicksort complexity analysis.)
3. Pedregosa, F., et al. — Scikit-learn: Machine Learning in Python, Journal of Machine Learning Research, 12, 2825–2830.
4. Python Software Foundation — time.perf_counter_ns Documentation, docs.python.org