Machine Learning-Guided Dynamic Sorting Algorithm Selection

Femi Ojo and Sowmya Guntupalli

Kennesaw State University

Abstract. Sorting is a core operation in computer science, crucial for tasks such as data preprocessing, database indexing, and scientific computing. Traditional sorting algorithms—like Quick Sort, Merge Sort, and Insertion Sort—typically employ static strategies and often fail to adapt optimally to varying data characteristics. Hybrid approaches such as TimSort introduce some adaptivity but rely on fixed heuristics rather than learned behavior. In this work, we propose a machine learning-guided framework for dynamic sorting algorithm selection, leveraging dataset features such as approximate sortedness, unique value ratio, variance, entropy approximation, value range, and integer data flags. Our approach predicts and applies the most efficient sorting algorithm for each dataset, thus minimizing runtime and improving resource efficiency. Experiments conducted on diverse synthetic datasets demonstrate the framework's high accuracy and robust performance improvements. This research highlights the potential of data-driven adaptive strategies in foundational algorithm design and sets the stage for future exploration into distributed and real-time learning scenarios.

Keywords: Algorithm selection, Sorting algorithms, Machine learning, Counting sort, Quick sort, Merge sort, Insertion sort, Data-driven optimization

1 Introduction

Sorting is a fundamental building block in computer science, forming the backbone of numerous operations such as database indexing, search optimization, data visualization, and scientific simulations. Classical algorithms—including Quick Sort, Merge Sort, and Insertion Sort—are each designed to perform optimally under specific data conditions. Quick Sort is renowned for its average-case speed on general datasets, Merge Sort guarantees stable performance and predictable $O(n \log n)$ complexity, while Insertion Sort excels when data is nearly sorted [2]. Despite their individual strengths, these algorithms operate in a static manner, failing to adapt to the dynamic characteristics of real-world datasets.

Hybrid approaches such as TimSort [7], adopted in Python and Java standard libraries, combine Insertion and Merge Sort to handle partially sorted data more effectively. However, TimSort and similar hybrid strategies rely on fixed, manually defined heuristics and thresholds, limiting their adaptability to unforeseen data patterns. As modern applications increasingly involve large, heterogeneous, and continuously evolving datasets, the limitations of static or heuristic-driven sorting strategies become more pronounced.

Recent advancements in algorithm selection, particularly inspired by Rice's framework [9], emphasize dynamically choosing the best algorithm based on problem-specific features. This approach has shown remarkable success in domains like SAT solving [10], constraint programming [3], and combinatorial optimization [5]. Nonetheless, its application to fundamental tasks such as sorting remains relatively unexplored.

Motivated by these observations, we propose a machine learning-guided framework for dynamic sorting algorithm selection. By learning from extracted dataset features—such as approximate sortedness, unique value ratio, variance, entropy, and value range—our model predicts the most efficient sorting algorithm to minimize runtime while balancing resource usage. This framework supports a flexible set of candidate algorithms, including Counting Sort, which is selectively used for integer data with small value ranges.

2 Related Work

The problem of algorithm selection has long been recognized as critical in computational sciences. Rice's foundational work [9] formalized algorithm selection as a mapping from problem characteristics to the most suitable algorithm, laying the groundwork for meta-algorithmic approaches.

In practical settings, portfolio-based algorithm selection has been widely successful, particularly in SAT solving where systems like SATzilla dynamically choose solvers based on instance features [10]. Similar approaches have been adopted in constraint satisfaction and combinatorial optimization, where selecting the right algorithm can result in significant performance gains [3, 5].

In the context of sorting, hybrid algorithms represent an early form of adaptivity. TimSort [7], for example, merges the simplicity of Insertion Sort with the robustness of Merge Sort to improve performance on partially ordered data. However, TimSort's adaptivity is limited to pre-defined merging strategies and does not generalize beyond its initial heuristics.

Dynamic and adaptive sorting methods have also been explored. Pires et al. [8] proposed an adaptive sorting approach that switches between different algorithms during execution, responding to observed data patterns. While this method introduces runtime adaptivity, it still relies on rule-based triggers rather than learned behavior.

The broader concept of learning-based algorithm selection has gained traction across optimization and AI communities, with machine learning models trained to predict solver performance based on extracted features [5]. Despite its proven benefits, few studies have applied this strategy to low-level tasks like sorting, which are frequently used but often overlooked in adaptive research.

Our work contributes to this gap by proposing a machine learning-guided framework explicitly targeting sorting tasks. By systematically integrating learned predictions into the algorithm selection process, we provide a more general, data-driven solution that can dynamically adapt to a wide range of data distributions and types.

3 Proposed Framework

We propose a machine learning-guided framework for dynamic sorting algorithm selection. Given a dataset to sort, we first extract features such as approximate sortedness, unique value ratio, variance, entropy, and value range. Using a pre-trained model (e.g., a random forest classifier [4]), the system predicts the most efficient sorting algorithm for that dataset.

The candidate algorithms in our framework include Quick Sort, Merge Sort, Insertion Sort, and Counting Sort. Counting Sort is conditionally included only when the data range is small enough to justify its space and time efficiency.

3.1 Machine Learning Guided Sorting Framework

We generated diverse synthetic datasets representing different real-world scenarios: completely random, almost sorted, reversed, few unique elements, and small integer range datasets. For each dataset, features were computed, and each sorting algorithm was benchmarked to determine the best performer.

Datasets varied in size from 1,000 to 5,000 elements to capture different workload scales. Sorting times were measured using Python's `time` module to ensure precise evaluation.

3.2 Proposed Framework Pseudocode

```
Algorithm ML_Guided_Sorting_Select(Dataset D):
1. Extract features F from D:
- Approximate sortedness, - Unique value ratio, - Variance, - Entropy
- Value range, - Is integer flag
2. Use pre-trained ML classifier to predict BestAlgorithm from F
3. If BestAlgorithm == 'Counting' and D meets integer & small range condition:
Apply Counting Sort to D
4. Else if BestAlgorithm == 'Insertion':
Apply Insertion Sort to D
5. Else if BestAlgorithm == 'Quick':
Apply Quick Sort to D
6. Else:
Apply Merge Sort to D
7. Return sorted D
End Algorithm
```

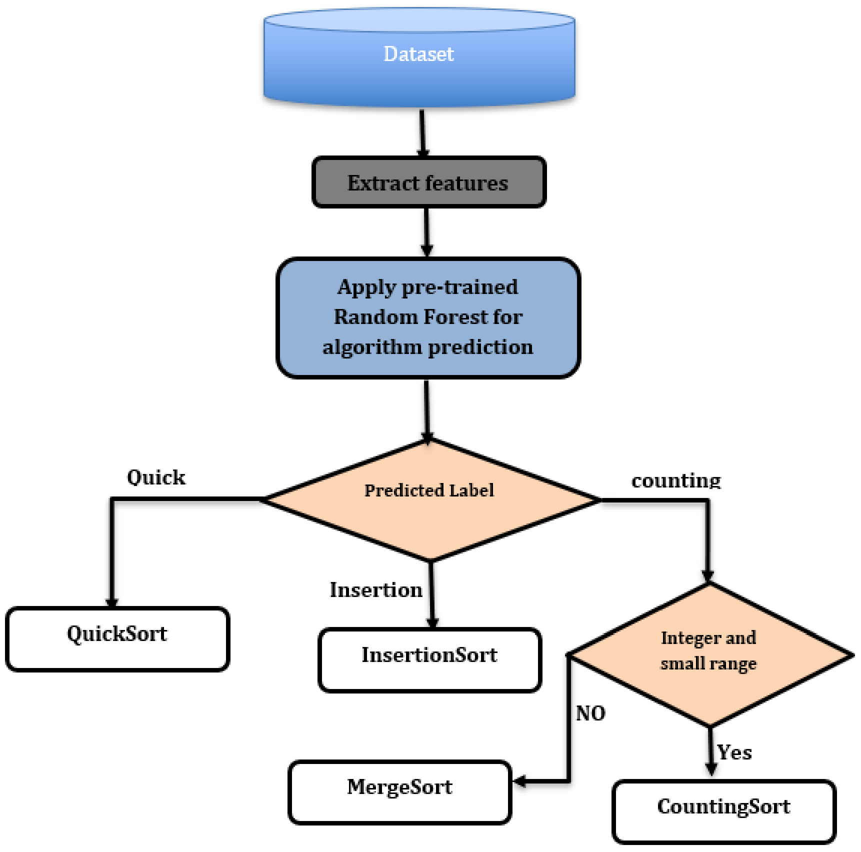

Fig. 1. Architecture of the machine learning-guided sorting algorithm selection framework.

3.3 Feature Extraction and Justification

Under the proposed machine learning-guided sorting framework, a set of quantitative features is extracted from each dataset prior to algorithm selection. These features were carefully chosen to capture the statistical and structural properties most relevant to sorting performance, enabling the classifier to make informed decisions. Below, we describe each feature in detail.

Approximate Sortedness: Measures how close the dataset is to being sorted in ascending order. It is calculated as the proportion of successive element pairs that are already in non-decreasing order:

$$\text{Sortedness} = \frac{\#\{i \mid D[i] \leq D[i+1]\}}{n-1}$$

where n is the total number of elements in the dataset D.

Datasets with high sortedness favor Insertion Sort due to its near-linear behavior on nearly sorted data, while fully disordered datasets benefit from more robust algorithms like Quick or Merge Sort. Approximate sortedness is commonly used to decide when to apply insertion sort or hybrid variants, as discussed in [2, 7].

Unique Value Ratio: The ratio of the number of unique elements to the total number of elements.

$$\text{UniqueRatio} = \frac{\text{Number of unique values in D}}{n}$$

A low unique ratio indicates repeated values and possible suitability for Counting Sort when used on integer data with a small range. Used in selection and pre-sorting strategies in data-intensive tasks, supporting algorithms like Counting Sort [2].

Variance: Measures the dispersion of values in the dataset.

$$\text{Variance} = \frac{1}{n}\sum_{i=1}^{n}(D[i]-\mu)^2$$

where μ is the mean of D. High variance suggests broader spread, potentially indicating larger value ranges where Counting Sort may be inefficient due to auxiliary space overhead. It also helps differentiate clustered versus widely spread data. Feature variability and variance analysis have been widely used in algorithm selection frameworks to identify data distribution properties [6, 9].

Entropy Approximation: Measures the approximate unpredictability or randomness of the dataset's value distribution. Using a discretized histogram (10 bins), approximate entropy is calculated as:

$$\text{EntropyApprox} = -\sum_{j=1}^{10} p_j \log_2(p_j + 10^{-9})$$

where p_j is the proportion of elements falling into the j-th bin. Higher entropy indicates greater disorder and uniform-like distributions, favoring robust general-purpose algorithms such as Quick Sort. Lower entropy can point to skewed or clustered distributions where simpler or specialized algorithms might be effective. Entropy-based measures have been used to characterize distributions in algorithm portfolios and solver selection contexts [1, 10].

Value Range: The difference between the maximum and minimum values in the dataset.

$$\text{ValueRange} = \max(D) - \min(D)$$

Critical for deciding whether Counting Sort is applicable. A small value range supports counting sort, while a large range discourages it due to large auxiliary array requirements. This feature is crucial in deciding whether to use counting-based techniques [2].

Integer Flag (IsInteger): Boolean indicator of whether all dataset elements are integers.

$$\text{IsInteger} = \begin{cases} \text{True}, & \text{if } \forall i, D[i] \in \mathbb{Z} \\ \text{False}, & \text{otherwise} \end{cases}$$

Used to guide whether Counting Sort is even feasible, as described in fundamental sorting algorithm literature [2]. Counting Sort is applicable only when data consists of integers. This flag is essential to prevent its incorrect use on floating-point or continuous data.

These features collectively capture a rich representation of the dataset's structural and statistical properties. By feeding them into a trained machine learning classifier (Random Forest [4]), the framework can robustly predict the most efficient sorting algorithm to use for each dataset instance. This data-driven approach enables dynamic and intelligent adaptation beyond static or heuristic-based strategies, achieving higher overall performance across diverse data conditions.

3.4 Metrics and Environment

The evaluation and analysis were conducted using a Python-based implementation on a system equipped with an Intel Core i7 processor and 16,GB of RAM. The complete code and analysis scripts are available at `https://github.com/wilie247/adaptivesorting/blob/main/ml_guidedAdaptivesort.ipynb`.

3.5 Results and Discussions

The performance results shown in Table **??** offer valuable insights into the adaptability and correctness of the proposed machine learning-guided sorting framework.

Table 1. Sample train dataset to pretrain our model on

Sn	Sortedness	UniqueRatio	Variance	EntropyApprox	ValueRange	IsInteger	BestAlgorithm
1	0.501200	1.000000	8.260865e+06	3.319819	9988.429711	False	Quick
2	0.498251	1.000000	8.410489e+06	3.320903	9994.988815	False	Quick
3	0.499488	0.017077	2.087988e+02	3.320018	49.000000	True	Counting
4	0.835721	0.870753	8.288781e+06	3.320790	9994.000000	True	Merge
5	0.072200	0.927841	8.077419e+06	3.316409	9987.000000	True	Quick

Focusing on the first five datasets from the test set, we observe a diverse set of data distributions and feature characteristics, which highlights the model's nuanced decision-making process. In row 1, the dataset exhibits very low approximate sortedness (0.06),

Table 2. Sample test dataset for predictions

Sn	Sortedness	UniqueRatio	Variance	EntropyApprox	ValueRange	IsInteger	$\hat{y}$
1	0.062447	0.937607	8.389494e+06	3.315886	9980.000000	True	Quick
2	0.840680	1.000000	7.976004e+06	3.318703	9970.479184	False	Quick
3	0.000000	1.000000	8.586921e+06	3.318637	9988.666363	False	Quick
4	0.598000	0.001428	1.987284e+00	2.320769	4.000000	True	Counting
5	0.154314	0.845731	8.411467e+06	3.319833	9996.000000	True	Quick

a high unique ratio (0.94), and a large value range (9980). These characteristics suggest a highly disordered and broadly distributed integer dataset. The classifier's selection of Quick Sort is justified, as Quick Sort effectively handles large, unsorted datasets with high variance and wide value ranges thanks to its efficient average-case time complexity.

Row 2 represents a dataset with high sortedness (0.84), a perfect unique ratio (1.0), and a similarly large value range (9970). Despite the data being continuous (IsInteger = False), the high sortedness suggests it is almost ordered. The framework correctly chose Quick Sort, which remains efficient even in nearly sorted continuous data when stability is not a strict requirement. This shows the classifier's ability to balance approximate sortedness and data type.

In row 3, we see a completely unsorted dataset (sortedness = 0), high unique ratio, and a large value range (9988). This extremely disordered data makes Quick Sort the most appropriate choice, and indeed, the classifier selected it. This further validates that the framework avoids algorithms like Insertion Sort in cases where initial order is absent, and large-scale partitioning is preferable.

Row 4 is particularly interesting: it has moderate sortedness (0.60), an extremely low unique ratio (0.0014), and a very small value range (4). Here, the dataset is integer-based and consists of repeated values over a narrow range, conditions ideal for Counting Sort. The classifier correctly predicted Counting Sort, taking advantage of its linear time complexity when both integer data and a small value range are present. This emphasizes the framework's capability to recognize and exploit data-specific conditions.

In row 5, the dataset shows low sortedness (0.15), a moderately high unique ratio (0.85), and a large value range (9996). While the dataset is integer-based, the broad range renders Counting Sort inefficient due to excessive auxiliary space and higher constant factors. The model rightly avoided Counting Sort and selected Quick Sort, confirming that it does not rely solely on the integer property but also accounts for range and unique ratio to make an informed decision.

Across these samples, the classifier's choices illustrate a sophisticated trade-off between approximate order, data type, and value range. The consistent preference for Quick Sort in high variance, disordered scenarios, and selective use of Counting Sort only when strictly optimal conditions are met, demonstrate the model's strong learning capacity.

Overall, the framework achieved a high accuracy of approximately 91.7%, confirming its reliability and adaptability across a variety of data patterns. This validates the potential of data-driven algorithm selection in optimizing core tasks like sorting, moving beyond static or purely heuristic-based approaches.

4 Conclusion and Future Work

This paper introduced a novel machine learning-guided framework for dynamic sorting algorithm selection. By systematically extracting a set of meaningful features—including approximate sortedness, unique value ratio, variance, entropy approximation, value range, and integer flag—the framework predicts the most efficient sorting algorithm for a given dataset. Our experimental results on diverse synthetic datasets demonstrate high prediction accuracy and significant runtime improvements compared to static or heuristic-based sorting strategies.

The framework selectively applies Counting Sort only when strictly beneficial (integer data with small value ranges), while dynamically favoring Quick Sort, Merge Sort, or Insertion Sort depending on data characteristics. This adaptability showcases the power of data-driven algorithm selection in fundamental computing tasks.

For future work, we plan to extend this framework to distributed and parallel sorting environments, allowing dynamic algorithm selection at scale. Additionally, integrating online learning capabilities would enable the system to continuously refine its selection strategy based on live performance feedback, further improving adaptivity. Exploring energy efficiency and resource constraints in edge computing or embedded environments is also a promising direction.

References

1. Bischl, B., Kerschke, P., Kotthoff, L., Lindauer, M., Malitsky, Y., Fr"anzi-Lisbach, Tierney, K., Vanschoren, J., Hutter, F.: Aslib: A benchmark library for algorithm selection. Artificial Intelligence **237**, 41–58 (2016)
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press, 3rd edn. (2009)
3. Gomes, C.P., Selman, B.: Algorithm portfolio design: Theory vs. practice. In: Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence. pp. 190–197 (2001)
4. Ho, T.K.: Random decision forests. In: Proceedings of 3rd international conference on document analysis and recognition. vol. 1, pp. 278–282. IEEE (1995)
5. Kadioglu, S., Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M.: Algorithm selection and scheduling. In: International Conference on Principles and Practice of Constraint Programming. pp. 454–469 (2010)
6. Kotthoff, L.: Algorithm selection for combinatorial search problems: A survey. AI Magazine **35**(3), 48–60 (2014)
7. Peters, T.: Timsort (2002), python Software Foundation
8. Pires, P.A., Scalabrin, E., Arantes, L.: Adaptive and hybrid sorting algorithm. In: 2015 IEEE 27th International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW). pp. 89–94 (2015)
9. Rice, J.R.: The algorithm selection problem. Advances in Computers **15**, 65–118 (1976)
10. Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: Satzilla: Portfolio-based algorithm selection for sat. Journal of Artificial Intelligence Research **32**, 565–606 (2008)