Algorithm Selection: A Predictive Model for Optimal Sorting

Artem Kiselev

April 6, 2025

1 Introduction

Sorting is a fundamental task that appears across various applications in computer science, from database management to data analytics and real-time processing systems. Due to its critical importance, hundreds of sorting algorithms have been developed, each with unique performance characteristics optimized for particular scenarios. Some algorithms excel at sorting nearly-sorted data, others at handling large datasets, and others at optimizing memory usage. As a consequence, no single algorithm universally outperforms all others for all problem instances. This naturally leads to the following research question:

> *Can we design a model that dynamically and intelligently selects the optimal sorting algorithm based on the characteristics of a given dataset?*

Successfully addressing this question and constructing an effective predictive model would represent a meaningful advancement. Such a model, capable of analyzing a dataset's characteristics and predicting the optimal sorting strategy based on that analysis, could deliver substantial performance improvements over current static approaches. In practical terms, this could significantly enhance efficiency in real-world scenarios, including large-scale database operations, data-intensive computations, and latency-sensitive applications, providing tangible benefits over existing sorting implementations.

2 Background and Problem Formalization

2.1 Algorithm Selection Problem

This question is an instance of the algorithm selection problem, formulated by John Rice in 1976 [5]. It is formally stated as follows:

Given:

- A problem space $\mathcal{P}$, containing all possible problem instances.
- A feature space $\mathcal{F}$, where each $f(x) \in \mathcal{F}$ represents measurable characteristics of problem $x \in \mathcal{P}$.
- An algorithm space $\mathcal{A}$, containing all applicable algorithms A_i.
- A performance space $\mathbb{R}^n$, where $p(A_i, x)$ represents the performance of algorithm $A_i \in \mathcal{A}$ on problem $x \in \mathcal{P}$.
- The final algorithm performance metric $||p||$, obtained by normalizing the raw performance measures.

Find: A selection mapping

$$S : \mathcal{F} \to \mathcal{A}$$

that maximizes performance according to some criteria.

2.2 Measures of Presortedness

Before we discuss how the defined model applies to our research question, we must talk about measures of presortedness, which are a measurement of the pre-existing order of a list. A good measure of presortedness must satisfy the following criteria:

Let m be a measure of presortedness, and X, Y be lists:

1. $m(X) = 0$ if X is *fully sorted in ascending order.*
2. If $X = [x_1, \cdots, x_n]$, $Y = [y_1, \cdots, y_n]$, and
$$x_i < x_j \iff y_i < y_j \quad \text{for all } i \text{ and } j,$$
then $m(X) = m(Y)$.
3. If $X \subseteq Y$, then $m(X) \leqslant m(Y)$.
4. If $X < Y$, then $m(XY) \leqslant m(X) + m(Y)$.
5. *For any element a,* $m(a + X) \leqslant |X| + m(X)$.

For this project, I will explicitly mention the following three measures of presortedness:

Runs [3]: The number of maximal contiguous ascending subsequences in a list X.

$$\text{Runs}(X) = |\{i : 1 \leqslant i < n \text{ and } a_{i+1} < a_i\}|\,.$$

This takes $O(n)$ time, and $O(1)$ memory.

Dis [2]: The largest distance for which an inversion exists

$$\text{Dis}(X) = \max\{j - i : 1 \leqslant i < j \leqslant n, x_i > x_j\}.$$

This takes $O(n)$ time, and $O(n)$ memory.

Mono [6]: The minimum number k such that X can be broken into k subarrays where each subarray is *monotonic*. A sequence is monotonic if it is increasing *or* decreasing.

$$\text{Mono}(X) = \min\Big\{k \in \mathbb{N} : \exists\, 1 = i_0 < i_1 < \cdots < i_k = n+1 :$$
$$\forall\, j \in \{1, \dots, k\}, (X_{i_{j-1}}, X_{i_{j-1}+1}, \dots, X_{i_j - 1}) \text{ is monotonic}\Big\}$$

This takes $O(n)$ time, and $O(1)$ memory.

Note: These are the only measures of presortedness that have a linear running time. This quality enables them to be used as features of lists for the problem of sorting.

2.3 Other List Features

Aside from the mentioned measures of presortedness, we define five other features:

1. **Size:** The total number of elements in the list,
$$\text{Size}(L) = |L|.$$
2. **Average duplicates per unique element:** For a list L with unique elements U, if $|L| = n$, then
$$\text{Avg. duplicates} = \frac{n - |U|}{|U|}.$$
3. **Shannon entropy:** With frequency function $f(u)$ for $u \in U$ and $p(u) = \frac{f(u)}{n}$,
$$H(L) = -\sum_{u \in U} p(u) \log_2 p(u).$$
4. **Categorical skewness:** For categorical data—where each unique category $u \in U$ occurs with frequency $f(u)$ and the mean μ and standard deviation σ are computed over these frequencies—the skewness is defined as
$$\gamma_1 = \frac{1}{|U|} \sum_{u \in U} \left(\frac{f(u) - \mu}{\sigma} \right)^3.$$
This measure reflects the asymmetry of the frequency distribution for categorical variables.
5. **Categorical kurtosis:** For categorical data with frequencies $f(u)$ for each $u \in U$, the kurtosis is defined by
$$\gamma_2 = \frac{1}{|U|} \sum_{u \in U} \left(\frac{f(u) - \mu}{\sigma} \right)^4.$$
This measure quantifies the tailedness of the frequency distribution for categorical variables.

2.4 Sorting Algorithm Selection Problem

In our context, the problem space $\mathcal{P}$ consists of datasets that require sorting, and an instance $x \in \mathcal{P}$ is a list to be sorted. The feature space $\mathcal{F}$ is comprised of the size, categorical skewness, categorical kurtosis, Shannon entropy and, most importantly, the three measures of presortedness Runs, Dis, and Mono.

Our algorithm space $\mathcal{A}$ includes various sorting algorithms like QuickSort, MergeSort, InsertionSort, and others. For simplicity, our performance space is defined as $\{0, 1\}$, where a value of 1 indicates that the selection algorithm has correctly identified the fastest sorter for a given problem instance. We can now more formally state our research question:

Given:

- Datasets $\mathcal{P} \ni x$, where x is a list that requires sorting.
- A feature space $\mathcal{F}$, containing of the above stated features.
- An algorithm space $\mathcal{A}$, containing all applicable sorting algorithms A_i.
- A performance space $\{0, 1\}$, where the performance metric is defined as
$$p(A_i, x) = \begin{cases} 1, & \text{if } A_i \text{ is the fastest sorter for } x, \\ 0, & \text{otherwise.} \end{cases}$$

Find: A selection mapping
$$S : \mathcal{F} \to \mathcal{A}$$
that maximizes the average number of fastest sorter selections, i.e.,
$$\max_S \frac{1}{|\mathcal{P}|} \sum_{x \in \mathcal{P}} p(S(f(x)), x).$$

3 Analysis and Implementation

In this section, I reference specific files used to conduct the analysis, all of which can be found in the project's GitHub repository at

`github.com/kiselevart/sorting-selector/`.

3.1 cpp-sort Library Integration

This project relies extensively on the `cpp-sort`[4] library, a C++ library featuring optimized implementations of various state-of-the-art sorting algorithms. Additionally, `cpp-sort` provides implementations of most measures of presortedness, referred to as probes.

To effectively leverage `cpp-sort` within Python, I created two separate interface files using `pybind11`: one file for interfacing with sorting algorithms and another dedicated to probes. These can be found in the `src/` directory. Utilizing `pybind11` allowed seamless integration of high-performance compiled C++ code into Python, thus maintaining Python's advantages in flexibility, extensive support for data science tasks, and ease of use within Jupyter Notebooks.

Before proceeding with the main analysis, I conducted preliminary benchmarks to ensure the reliability and efficiency of `cpp-sort`'s algorithms. The results from these benchmarks, detailed in `verify_running_time.ipynb`, confirmed that the chosen sorting methods performed consistently and significantly outperformed standard alternatives. Specifically, integrating the compiled sorters from `cpp-sort` provided, on average, a tenfold speedup compared to NumPy's sorting methods and over a fortyfold improvement compared to Python's built-in sorter.

3.2 Sorters Redundancy Analysis

After verifying the usability of the sorters within the `cpp-sort` library, I performed a "redundancy analysis" to identify and remove sorters that do not provide unique advantages over others. This process involved benchmarking various sorters across multiple datasets and visually analyzing performance graphs to detect any that were consistently outperformed. To confirm these observations, I conducted direct comparisons between selected algorithm pairs. The detailed analysis and graphs can be found in `compare_sorts.ipynb`.

Based on this analysis, the following eight sorters were retained:

- **heap_sort**
- **insertion_sort**
- **merge_sort**
- **quick_sort**
- **quick_merge_sort**
- **spin_sort**
- **std_sort**
- **tim_sort**

These selected sorters offer a diverse set of strategies and performance characteristics, ensuring robust and efficient sorting capabilities across various types of datasets.

3.3 Probe Analysis

The next step of the project involved an analysis of the available probes within the `cpp-sort` library. To achieve this, we computed the relative performance of each probe by evaluating the ratio between the execution time of each probe and the actual sorting time for every given list.

Formally, let m_i represent a probe, A_j a sorting algorithm, and $X_k \in X$ a specific list. Then, the ratio for a single list is defined as:

$$r_k = \frac{m_i(X_k)}{A_j(X_k)}$$

The average ratio across all lists is computed as:

$$\bar{r} = \frac{1}{|X|} \sum_{k=1}^{|X|} r_k.$$

The resulting heatmap visualization of these ratios can be accessed in `probes_vs_sorts.ipynb`. Analysis of this heatmap, combined with considerations regarding the time complexity of all the presortedness measures clearly indicates that `Dis`, `Mono`, and `Runs` exhibit significantly superior efficiency. Thus, these three probes were selected for the feature space of the predictive model.

4 Machine Learning and Results

After experimentally verifying the feature space, we synthesized a training dataset by computing all features in $\mathcal{F}$ and recording the fastest sorter for each list instance. Using these labeled data, we trained a classification model where the target variable was the fastest sorter. After exploring various algorithms and optimizing parameters, a Random Forest Classifier [1] yielded the highest overall accuracy of 80.7%. Details on the training process can be found in `train_model.ipynb`.

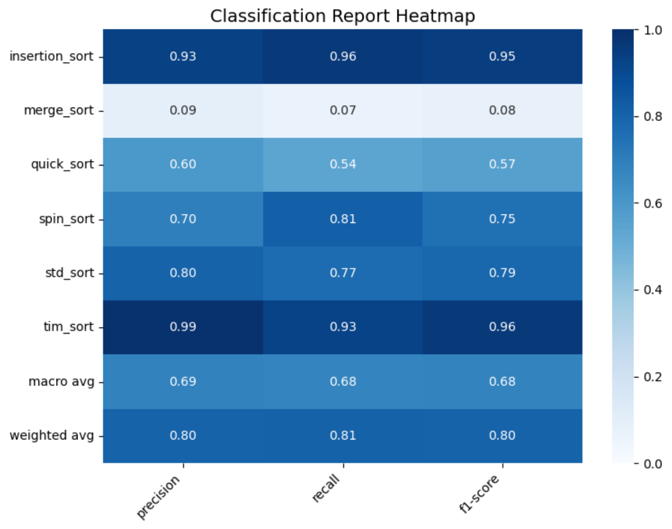

Figure 1: Classification Report Heatmap of the Random Forest Model

Figure 1 shows a classification report heatmap for the trained model. The model classifies `tim_sort` and `insertion_sort` with notably high accuracy, likely because their performance characteristics differ distinctly from the other algorithms. By contrast, `merge_sort` shows a high misclassification rate.

One probable cause is a consistent performance spike around lists of size 750,000 to 800,000 elements, as shown in Figure 2. This spike may arise from caching or memory-access patterns specific to `merge_sort`, which significantly degrades its performance in that narrow range of input sizes. Consequently, the model undervalues `merge_sort` and fails to predict it even when it would otherwise be optimal.

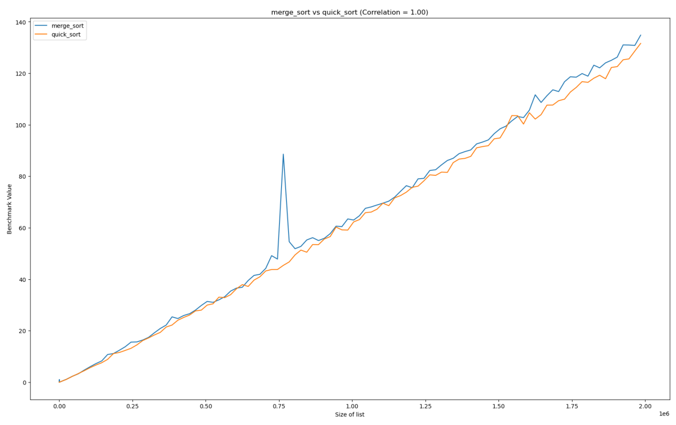

Figure 2: Performance Spike of `merge_sort` (blue) Around 750–800k Elements

Turning to feature importance (Figure 3), the three presortedness metrics clearly dominate as the most significant features. This aligns well with the intuition that knowing how close a list is to being sorted is critical when predicting the optimal sorting algorithm.

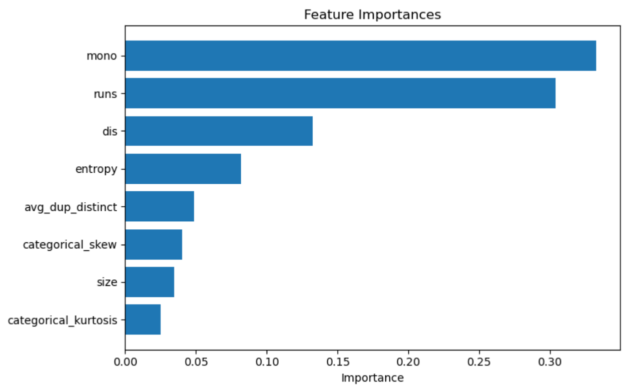

Figure 3: Feature Importances from the Random Forest Classifier

To further assess the model's performance in practice, Figure 4 compares the running time of the model's predicted sorter (blue) against each static sorter (orange) across 35,000 test cases. While there are certain anomalies, the predictor generally runs comparably to the slower among the static sorters, which is a promising start for this initial model. Through implementing and researching the refinements that will be discussed below, there is a good chance that the predictive model could approach the performance of the optimal sorter on each instance.

5 Further Work

Due to the time constraints of this project, several improvements can be explored, along with numerous promising avenues for deeper research.

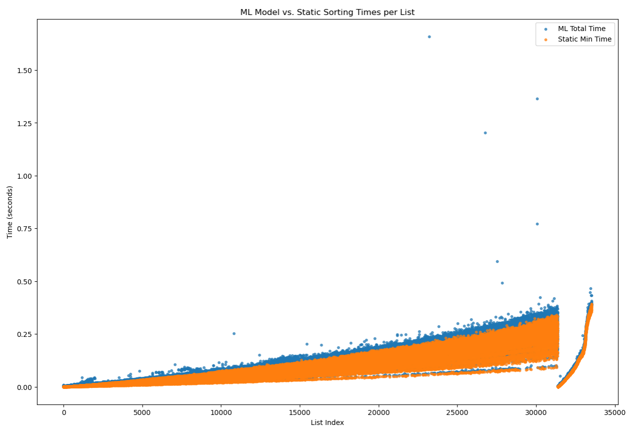

Figure 4: Comparison of Model Predictions (blue) vs. Static Sorters (orange) Over 35,000 Cases

5.1 Potential Improvements

Several aspects of the current work could significantly benefit from enhancements, primarily in data quality and analytical methods. The datasets utilized throughout this project were synthetic, limiting the robustness of the results. Employing real-world data could substantially enhance the rigor and validity of the analysis. Furthermore, the synthetic datasets suffered from imbalances and were insufficiently large for training robust machine learning models. Increasing the dataset size and ensuring balanced class distributions could lead to more accurate and generalizable models.

Furthermore, the redundancy analysis of sorting algorithms relied predominantly on visual and graph-based inspections. While useful for exploratory analysis, such approaches may introduce biases and inaccuracies. Automating this step through statistical or algorithmic measures could reduce subjectivity and improve the reliability of the analysis. Lastly, the algorithm portfolio was significantly reduced for simplicity, which may have led to excluding potentially high-performing or contextually relevant algorithms. Expanding and rigorously evaluating a broader portfolio of sorting algorithms could yield more comprehensive and accurate insights.

5.2 Areas for Deeper Research

Several promising research directions could further extend the scope and applicability of this project:

- **Enhanced Cost Functions:** Experimenting with more sophisticated or cost functions could lead to better model performance. For example, cost functions could aim to minimize the squared difference or absolute deviation between predicted sorter performance and the optimal sorter, rather than relying solely on categorical accuracy.

- **Hybrid Model Development:** Investigating hybrid approaches that combine learned predictive models with manually engineered heuristics would almost certainly yield benefits. For instance, manually overriding the model's predictions with counting sort when inputs are known integers within a small, bounded range could enhance overall efficiency and accuracy.

- **Approximation of Presortedness Metrics:** Developing methods to approximate presortedness could significantly enrich the feature space and thus improve model performance.

Each of these areas represents significant potential for deeper exploration and could substantially advance this research topic.

6 Conclusion

This project has successfully demonstrated the viability of using machine learning techniques to predict sorting algorithm performance based on input data features. Despite limitations due to synthetic data and algorithm simplifications, the results provided valuable insights and established a foundation for further enhancements. By addressing these limitations through improved data quality, expanded algorithm portfolios, and more rigorous analytical approaches, future research can significantly enhance predictive accuracy and robustness.

Exploring advanced cost functions, hybrid models, presortedness metrics, and enhanced feature engineering promises to expand the utility and effectiveness of the predictive models developed. Successfully implementing these improvements could significantly enhance efficiency in real-world scenarios, including large-scale database operations, data-intensive computations, and latency-sensitive applications, providing tangible benefits over existing sorting implementations.

References

[1] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[2] Vladmir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Comput. Surv.*, 24(4):441–476, December 1992.

[3] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, 1973.

[4] Morwenn. cpp-sort. `https://github.com/Morwenn/cpp-sort`. Accessed: April 6, 2025.

[5] John R. Rice. The algorithm selection problem. volume 15 of *Advances in Computers*, pages 65–118. Elsevier, 1976.

[6] Hantao Zhang, Baoluo Meng, and Yiwen Liang. Sort race, 2016.