A Comparative Analysis of K-Nearest Neighbors Performance: Mojo vs. Scikit-learn

Sumanth Kolli[#], CJ Wu[#], and Dr. Henry Han[*]
Data Science and Artificial Intelligence Innovation Lab, Baylor University, TX, USA
[#] These authors contributed equally to this work.
[*] Corresponding author. Henry_Han@baylor.edu

Abstract. This study presents a rigorous comparative evaluation of Mojo, a newly developed compiled programming language featuring strong typing, SIMD vectorization, and parallel execution capabilities, specifically in its application to the K-Nearest Neighbors (KNN) algorithm. We conducted experiments using two distinct and representative datasets: the structured, moderately sized MNIST dataset and a large-scale, synthetically generated high-dimensional dataset. The evaluation was structured around two central research questions:

1. Under what specific dataset conditions—considering size, structural regularity, and dimensionality—does Mojo deliver measurable runtime improvements over the widely-used Python scikit-learn (sklearn) implementation of KNN? [18]
2. To what extent do Mojo's low-level computational optimizations translate into meaningful performance gains in practical, performance-critical machine learning workflows?

Our findings reveal that Mojo consistently outperforms the standard sklearn implementation on structured datasets of moderate scale, primarily due to its efficient memory management, explicit compilation strategies, and parallel computation.[3] However, these performance benefits become less predictable or diminish as dataset size and structural complexity increase on lower-end hardware. These nuanced results underscore the critical importance of carefully matching language features and tool selection to the particular computational characteristics and performance demands of machine learning tasks. Consequently, our study contributes valuable insights into the conditions under which Mojo represents a superior alternative to traditional Python implementations [15], informing strategic decisions for high-performance computing applications in machine learning.

Keywords: KNN · Mojo · Performance optimization · scikit-learn

[3] For all benchmark data and reproducibility scripts, see Appendix A.

1 Introduction

The K-Nearest Neighbors (KNN) algorithm remains a fundamental component [6] of many machine learning systems [22] due to its intuitive design and effectiveness in both classification and regression contexts. As a non-parametric method, it makes no prior assumptions about data distribution, which contributes to its broad applicability. However, the algorithm's reliance on exhaustive distance computations between input samples and all training instances results in considerable computational overhead, particularly when applied to high-dimensional or large-scale datasets. These performance limitations constrain KNN's deployment in latency-sensitive or resource-constrained environments, where algorithmic efficiency is critical.

In practice, one of the most accessible and widely adopted implementations of KNN is provided by *scikit-learn* (sklearn), a Python-based machine learning library that prioritizes ease of use and broad functionality, while incorporating C-based methods for algorithmic efficiency. Sklearn's `KNeighborsClassifier` [4] offers a standardized API for applying KNN to a variety of tasks, leveraging efficient Cython backends for distance calculations and support for parallel processing via multi-threading. Despite these optimizations, sklearn remains constrained by the performance ceilings of the Python ecosystem [16] and the inherent costs of runtime interpretation, particularly when scaling to millions of data points or integrating with hardware-level acceleration. As such, while sklearn KNN provides an excellent pedagogical and prototyping tool, its suitability for production-grade, performance-critical applications can be limited.

To mitigate such limitations, recent developments in programming language design have introduced tools optimized for high-performance computing. One such language, Mojo, is a compiled language that combines strong static typing with explicit memory management, SIMD (Single Instruction, Multiple Data) vectorization, and support for parallel execution. These capabilities position Mojo as a potentially advantageous alternative to traditional interpreted environments like Python, especially in workloads dominated by low-level numerical operations. Mojo has been widely advertised as significantly outperforming Python [14] in specific benchmarks, with some promotional materials claiming execution speeds up to 68,000 times faster in numerical computing tasks [19]. These claims, while impressive, stem primarily from controlled benchmarks and promotional content, and thus warrant further empirical evaluation in varied real-world contexts.

For more color, consider adding the graph from the article

This study conducts a comparative analysis of KNN implementations in Mojo and scikit-learn to assess the practical impact of Mojo's performance-oriented features. The evaluation is guided by the following key questions:

- Under what dataset conditions-such as varying size, structural regularity, and dimensionality—does Mojo demonstrate tangible runtime improvements?
- To what extent do Mojo's low-level optimizations translate into meaningful advantages for performance-critical machine learning tasks?

By exploring these questions, this paper aims to provide clear evidence about Mojo's real-world applicability, thereby offering valuable insights into optimal language and implementation selection for high-performance machine learning workflows.

2 Methods

2.1 Experimental Setup

Hardware and software configuration:

- WSL2 Environment running ubuntu 24.04.2 [1]
- Mojo 25.1.1 environment, required files (`Current KNN Test.mojo`, `split.mojo`, `csv.mojo`).
- Python 3.12 environment with Scikit-learn for baseline comparison.
- Low-End Machine: laptop running 6-core/12-processor 2.6GHZ CPU and 16GB RAM
- High-End Machine: dekstop running an 8-core/16-processor 4.2GHZ CPU and 32GB RAM

Datasets:

- Iris (105 × 4 training set, 45 test cases)
- Wine (143 × 13 training set, 35 test cases)
- Cancer (455 × 30 training set, 114 test cases)
- Digits (1,697 × 64 training set, 100 test cases)
- MNIST (69,895 × 784 training set, 105 test cases)
- Fashion (69,895 × 784 training set, 105 test cases)
- Randomly Generated Blob Data (250,000 × 400 training set, 1,000 test cases)
- Randomly Generated Data (250,000 × 400 training set, 1,000 test cases)
- CIFAR (55,000 × 3,072 training set, 5,000 test cases)
- Cover (522,910 × 54 training set, 58,102 test cases)
- Randomly Generated Euclidean Distance Data (100,000 × 1,000 training set, 1,000 test cases)

2.2 Implementation Details

To evaluate the performance of K-Nearest Neighbors (KNN) across different implementation paradigms, we developed a custom brute-force KNN classifier in Mojo that emphasizes memory alignment, SIMD vectorization, and thread-level parallelism. This section details our design and optimization strategies and contrasts them with scikit-learn's standard KNN implementation.

Mojo KNN Implementation Our Mojo implementation leverages low-level control to optimize each computational stage:

Matrix and Memory Layout. We designed a simple `Matrix` type that stores all values in a flat block of memory. Each row starts at a 64-byte boundary so that vectorized instructions (like AVX) can load data safely and efficiently. [7] This manual alignment avoids errors and improves speed. Instead of allocating

[1] Running Mojo in a dedicated linux boot did not show signifigant speed improvements over WSL

new memory every time we run a query, we allocate one large buffer in advance and reuse it. This reduces memory waste and improves consistency in timing.

Distance Calculation with SIMD. Euclidean distance is computed using Mojo's built-in vector types (`vector<f32,8>` or `vector<f32,16>`). We unroll the inner loop using these vectors and apply horizontal additions to sum squared differences. [20] This reduces the number of required operations by roughly a factor of 8, since multiple elements are processed at the same time using vector registers. This takes advantage of data-level parallelism. We also employ software prefetching to reduce memory latency during sequential scans of large matrices.

Efficient Top-k Selection. To find the k closest neighbors, we use different strategies depending on how many values we need:

- If k is small, we use a fast sorting method that works directly in vector registers. [8] This avoids memory overhead and is very fast.
- If k is larger, we use a technique called `argpartition`, which helps us find the top k values without sorting everything. This saves time by skipping unnecessary comparisons.

Parallel KNN Inference. We split the test dataset into equal parts and assign them to different CPU threads. Each thread works on its own portion of the data without needing to wait for others. Once all threads finish, we combine the results into a final answer. This method avoids slowdowns from locking and ensures all CPU cores are fully used.

Buffer Copying with `memcpy`. When we need to move data between memory blocks, we use `memcpy` to copy it quickly and efficiently. [21] This is helpful when preparing rows for processing or combining outputs. It avoids slow, loop-based copying and works well with our aligned memory setup.

scikit-learn KNN Implementation Scikit-learn's `KNeighborsClassifier` employs Cython and NumPy back-ends. [1] Its brute-force method relies on nested C-level loops and supports BLAS acceleration. Note that brute-force was the method we chose to compare in this paper, as runtimes are agnostic of data distribution, only data size. Top-k selection is handled by `std::nth_element`, and data is stored in standard C-order NumPy arrays. [23] While optimized at the system level, the library must interface with Python's dynamic runtime, incurring boundary-crossing overhead.

Parallelism is achieved through OpenMP, which may interact unpredictably with Python's Global Interpreter Lock (GIL). Additionally, although scikit-learn offers kd-tree and ball-tree acceleration modes, we limited our comparison to brute-force for consistency and to highlight Mojo's raw performance capabilities.

Summary of Key Differences Table **??** provides a comparison of the key differences between our Mojo implementation and scikit-learn's implementation.

Feature	Mojo Implementation	scikit-learn Implementation
Memory Layout	Manually aligned flat buffers with 64-byte alignment	NumPy arrays (C-order) with runtime-managed memory [10]
SIMD Utilization	Explicit vector types with hand-tuned intrinsics [9]	Implicit, via Cython and BLAS
Top-k Selection	SIMD quick-sort and vectorized argpartition	Partial sort via C++ STL (`nth_element`)
Multithreading	Manual thread pool, pinned to CPU cores	OpenMP, subject to Python GIL limitations
Memory Management	Static allocation, zero runtime fragmentation	Dynamic allocation via Python runtime
Data Movement	`memcpy` used for aligned block copying	Internal copying via NumPy and Python buffers

Table 1: Comparison of Mojo and scikit-learn KNN implementations

3 Results

3.1 Euclidean Distance Calculation

Implementation	Execution Time (s)	Standard Error	Relative Speed
Pure Python	20.100	± 0.300	1.00×
NumPy	0.782	± 0.132	25.70×
Mojo	0.120	± 0.050	167.50×

Table 2: Performance Comparison of Different Implementations

To ensure a fair and focused evaluation of raw computational performance, all implementations were constrained to single-core, non-parallelized execution. This setup isolates the efficiency of the underlying compiled code, allowing for a clearer assessment of how each environment handles fundamental floating-point operations (FLOPs). As shown in Table **??**, the Mojo implementation significantly outperformed both NumPy and pure Python. Specifically, Mojo achieved an execution time of 0.120 seconds, making it approximately 167 times faster than the baseline Python implementation [13] and over 20 times faster than NumPy. These results highlight Mojo's efficiency in executing low-level numerical computations, when advanced parallelism or hardware acceleration is not utilized.

3.2 KNN Results

FIX FORMATTING HERE

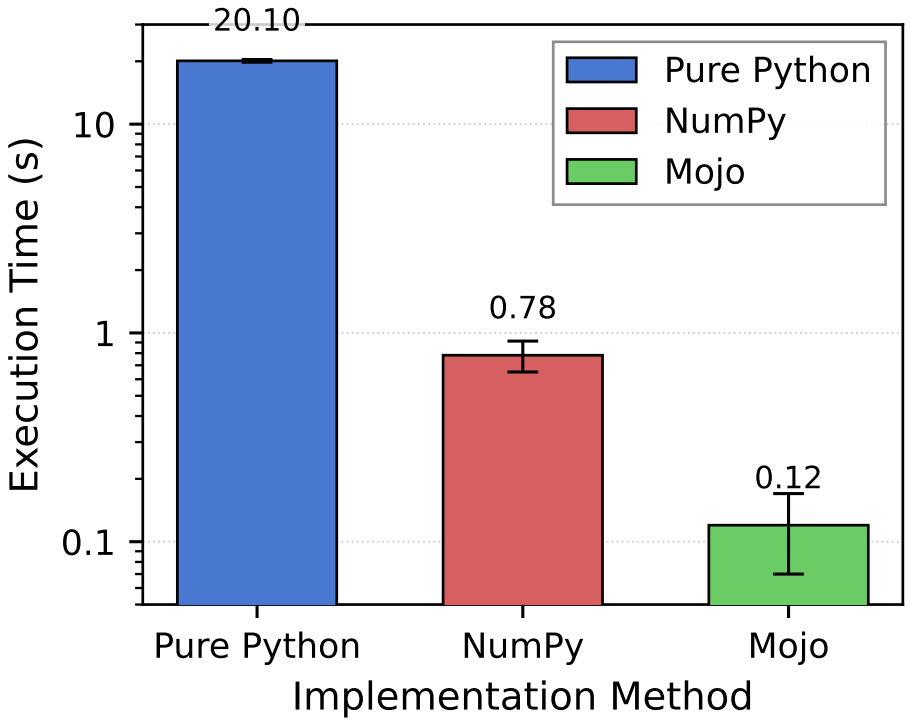

Fig. 1: Comparison of runtime of Euclidean Distance Vector Calculation.

Table 3: Comparison of Python vs Mojo runtimes on Low-End PC

Dataset	**Python (s)**	**Mojo (s)**	**Speedup (%)**	**p-value**
Iris	1.105 ± 0.141	0.179 ± 0.010	83.8%	< 0.0001
Wine	0.013 ± 0.004	0.0003 ± 0.00002	97.8%	< 0.0001
Cancer	0.020 ± 0.006	0.0005 ± 0.00003	97.4%	< 0.0001
Digits	0.030 ± 0.005	0.0022 ± 0.00003	92.9%	< 0.0001
MNIST	2.164 ± 0.096	1.252 ± 0.143	42.1%	< 0.0001
Fashion	2.122 ± 0.060	1.216 ± 0.160	42.7%	< 0.0001
Blob	32.916 ± 0.492	33.298 ± 1.573	-1.2%	0.47
Random	31.926 ± 0.443	33.296 ± 1.690	-4.3%	0.02
CIFAR	N/A	N/A	N/A	N/A
Cover	N/A	N/A	N/A	N/A

Table 4: Comparison of Python vs Mojo runtimes on High-End PC

Dataset	**Python (s)**	**Mojo (s)**	**Speedup (%)**	**p-value**
Iris	0.011 ± 0.001	0.0002 ± 0.00005	98.1%	< 0.0001
Wine	0.011 ± 0.001	0.0002 ± 0.00009	97.9%	< 0.0001
Cancer	0.012 ± 0.001	0.0010 ± 0.0014	91.8%	< 0.0001
Digits	0.011 ± 0.000	0.0009 ± 0.0001	91.8%	< 0.0001
MNIST	0.541 ± 0.020	0.230 ± 0.011	57.5%	< 0.0001
Fashion	0.520 ± 0.004	0.223 ± 0.019	57.1%	< 0.0001
Blob	8.620 ± 0.312	8.140 ± 0.084	5.6%	0.0002
Random	8.653 ± 0.294	8.156 ± 0.050	5.7%	0.0001
CIFAR	76.841 ± 1.675	45.500 ± 0.186	40.8%	< 0.0001
Cover	192.524 ± 0.917	N/A	N/A	N/A

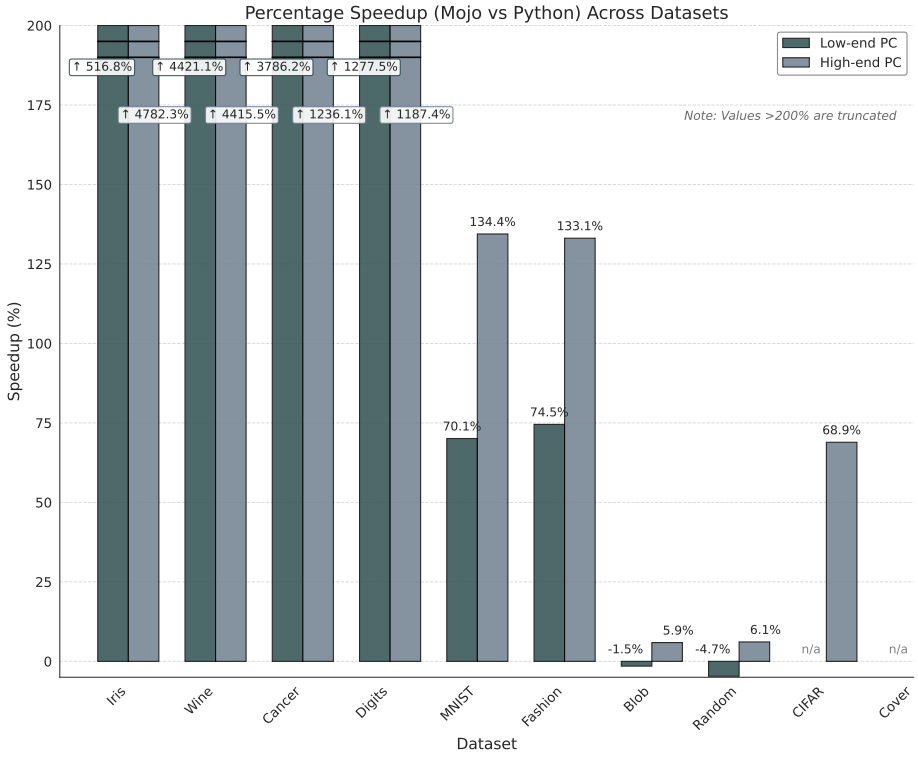

Fig. 2: Speedup comparison between Python and Mojo across different datasets and different PC's. Percentages above 200% are truncated, but still represented. Values not present in both low-end PC's and high-end PC's are represented as N/A.

Note that CIFAR and Cover data both were unable to run on the low-end laptop, due to hardware limitations. Cover data also did not run in mojo on the higher-end PC. See appendix for attached figures visualizing these tables.

Regardless of PC, smaller datasets were speedup significantly when using MOJO. The most likely cause for this is sklearn's significant overhead when compared to mojo. Medium-sized data was speedup pretty consistently (MNIST and Fashion). Large data (Blob and Fashion) was dependent on the PC, experiencing a 1 to 4% speed down on MOJO on the low-end PC, and a 5% speedup on the high-end PC. Note that for every dataset that was within the PC's scope, mojo ran faster than python on the high-end PC. With data with very large feature counts (CIFAR) [11], there is a significant speedup (40%) on higher-end PCs. The lack of support for chunking large datasets is not a mojo issue, but rather a programmatic issue that can be improved upon.

4 Discussion

4.1 Interpretation of Results

The strengths of Mojo as a compiled language, including its support for SIMD optimization and parallelization, were evident in scenarios involving structured, smaller datasets. These features contributed to significant performance improvements, particularly in tasks requiring intensive numerical operations such as distance calculations and neighbor searches within the KNN algorithm. Notably, Mojo-based implementations demonstrated superior performance over NumPy in high-throughput, low-latency tasks. A detailed analysis of the benchmark results reveals that the performance advantage was most prominent during compute-heavy operations, where the overhead associated with Python's interpreted nature in [5] scikit-learn became a limiting factor.

4.2 Practical Implications

The results suggest that Mojo is particularly well-suited for use cases where real-time predictions are required on relatively smaller datasets. Examples include applications in recommendation systems and edge computing, where low-latency and high-throughput are critical. Furthermore, Mojo's capabilities in parallelism and explicit control over threading make it highly suitable for deployment on high-performance systems, including those equipped with GPUs or multi-core processors. Systems that prioritize raw computational speed and efficient system-level performance will benefit from Mojo, provided the data size remains within manageable limits.

4.3 Limitations

While Mojo offers considerable advantages, there are limitations that must be considered. The language is still in its early stages of development, which contributes to variability in performance, stability, and the availability of features.

As such, developers may encounter bugs or issues that could remain unresolved for extended periods due to the relatively small development community. The evolving nature of Mojo further necessitates frequent adaptations to maintain compatibility with new releases, which may pose challenges for long-term project stability.

4.4 Early Development

The results presented in this study are influenced by the early-stage development of Mojo, which has not yet reached a mature and stable version. Decisions regarding algorithmic choices, such as the selection of distance metrics, memory handling strategies, and parallel task scheduling, were made based on the capabilities available at the time. Mojo's claims of enhanced performance through modern compilation techniques and efficient memory management were generally supported by the experimental results, particularly in the context of raw computational speed. Further details of these claims, as well as early-stage benchmarking results, are presented in Appendix A and Figure **??**, which provide an overview of the initial test outcomes.

4.5 User and Developer Experience

From a user and developer perspective, Mojo presents several challenges that are inherent to working with a language still in early development. One of the most notable difficulties is the limited and often outdated documentation. [17] While some resources are available, they tend to be sparse and lack the depth necessary for efficient use of the language. This can lead to frustration, as developers must often resort to trial and error or reverse-engineering code to understand the functionality and proper usage of various features. The absence of comprehensive tutorials or examples further complicates the learning process, making it more time-consuming for developers to gain proficiency.

Additionally, Mojo is still evolving, which presents its own set of challenges. Frequent updates introduce changes in syntax, semantics, and functionality, necessitating continuous adaptation of codebases. These changes can result in compatibility issues, requiring developers to allocate time and effort to ensure that their code remains functional after each new release. This ongoing evolution, while indicative of the language's potential, introduces an added layer of instability that may hinder development efforts, particularly for long-term projects.

Another challenge developers face is the absence of a dedicated interactive development environment, such as Jupyter or Colab, which is commonly used in data science and machine learning workflows. The lack of such an environment complicates the prototyping and iterative development process. Without an intuitive interface that allows for rapid experimentation and real-time feedback, developers are forced to rely on less efficient and more cumbersome workflows. This significantly increases the time required for testing and refining models, which is a critical aspect of many data-driven applications.

These challenges, while significant, are not insurmountable and should be understood within the context of Mojo's early-stage development. As the language matures and the developer community grows, it is anticipated that many of these issues will be addressed. Nevertheless, developers currently working with Mojo must be prepared to navigate these obstacles and adapt to a language that is still in the process of evolving. Such considerations should be taken into account when evaluating the potential of Mojo for future projects.

4.6 Empirical Runtime Complexity

We evaluated the empirical runtime complexity of Mojo KNN by fitting a log-log regression to observed runtimes across datasets of varying training sizes. Using the model $\log T(n) = k \log n + \log C$, we estimated the effective complexity as $T(n) \approx C \cdot n^k$, where k reflects scaling behavior and C represents the constant baseline runtime cost.

On the high-end system, we found $k \approx 1.33$ with $C \approx 2.07 \times 10^{-7}$; on the low-end system, $k \approx 1.19$ with $C \approx 3.80 \times 10^{-6}$. Although the High-End PC delivers faster runtimes, its higher complexity exponent suggests steeper growth as data size increases. In contrast, the Low-End PC scales more gradually but incurs greater initial overhead.

The constant C captures hardware-independent efficiency—such as memory layout, data access speed, and language-level overhead. A smaller C indicates better low-level optimization. This is why C matters: while k tells us *how fast* runtime grows, C tells us *how fast it starts*. Understanding both is essential for choosing the right implementation in production environments.

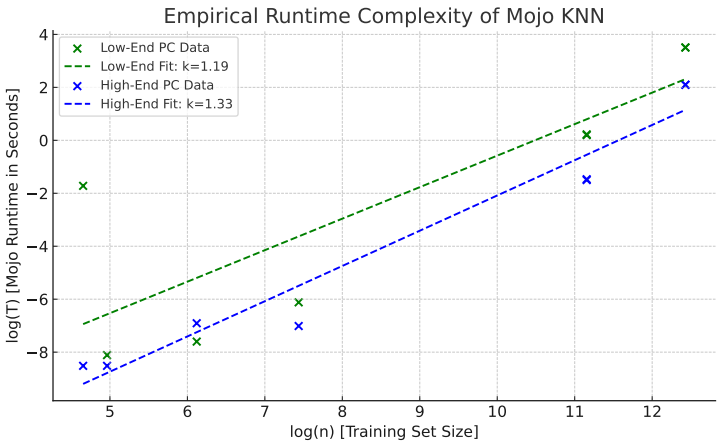

Fig. 3: Empirical runtime complexity of Mojo KNN based on log-log regression. The fitted slopes suggest $O(n^{1.33})$ scaling on high-end and $O(n^{1.19})$ on low-end systems. Constants: $C_{\text{high}} \approx 2.07 \times 10^{-7}$, $C_{\text{low}} \approx 3.80 \times 10^{-6}$.

4.7 Future Work

1. Investigate further optimization techniques in Mojo, including memory management strategies, caching, and manual memory pooling.
2. Explore additional algorithms for KNN, such as Ball Trees and KD-Trees, as implemented in scikit-learn [3], to assess the potential for performance improvements.
3. Contribute to the development of a comprehensive data loading and exporting system that enhances the Mojo ecosystem and its usability for data-driven applications.
4. Conduct a comparative analysis of Mojo's performance with specialized approximate nearest neighbor libraries, such as Faiss or Annoy [2], to better understand its relative strengths and limitations.
5. Move away from KNN, and attempt to implement more complex machine learning algorithms in mojo. [12]

5 Conclusion

We evaluated KNN implementations in Mojo and scikit-learn, focusing on low-level optimizations such as SIMD vectorization and memory alignment. Mojo showed substantial speed-ups, up to 90%, on structured, medium-sized data and remained competitive on larger datasets with sufficient hardware.

Its performance is based on tight control over compilation and memory, enabling efficient execution of compute-bound operations. However, the gains varied with the structure and hardware quality, and the early-stage Mojo tooling remains a limitation.

This study offers practical insights into when and why Mojo delivers performance benefits, contributing to a broader understanding of emerging compiled languages in ML.

6 ACKNOWLEDGMENTS

We express our gratitude to the Data Science and Artificial Intelligence Innovation Lab at Baylor University, under the leadership of Dr. Henry Han, for providing the resources and support necessary to carry out this research. We also acknowledge the contributions of the Mojo developer community, whose voluntary efforts in addressing questions on GitHub and publishing insightful articles were invaluable during the development process.

7 References

References

1. 4, S.N.P.: Knn with scikit-learn (2023), `https://github.com/sch-notes-phase-4/knn-with-scikit-learn`

2. Aumüller, M., Bernhardsson, E., Faithfull, A.: Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. Information Systems **87**, 101374 (2020)
3. Cao, K., et al.: Optimization study of knn classification algorithm on large-scale datasets: Real-time optimization strategy based on balanced kd tree and multi-threaded parallel computing. In: 2023 4th International Conference on Intelligent Computing and Human-Computer Interaction (ICHCI). IEEE (August 2023). https://doi.org/10.1109/ICHCI57844.2023.10255066
4. Creek, P.: Python sklearn.neighbors.kneighborsclassifier() examples (2023), https://www.programcreek.com/python/example/104209/sklearn.neighbors.KNeighborsClassifier
5. scikit-learn developers: How to optimize for speed — scikit-learn 1.6.1 documentation (2023), https://scikit-learn.org/stable/developers/performance.html
6. English, A.P.: Everything you need to know about k-nearest neighbors (knn) (2023), https://ai.plainenglish.io/everything-you-need-to-know-about-k-nearest-neighbors-knn-69191186452d
7. Franchetti, F., Kral, S., Lorenz, J., Ueberhuber, C.W.: Efficient utilization of simd extensions. Proceedings of the IEEE **93**(2), 409–425 (2005)
8. Furtak, S., Gregg, D., Almeida, A.M.T.F.: Using simd registers and instructions to enable instruction-level parallelism in sorting algorithms. In: Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). pp. 348–349. ACM (2007)
9. Intel: Explicit vector programming – best known methods (2023), https://www.intel.com/content/www/us/en/developer/articles/training/explicit-vector-programming-best-known-methods.html
10. Intel Corporation: Explicit vector programming – best known methods. https://www.intel.com/content/www/us/en/developer/articles/training/explicit-vector-programming-best-known-methods.html (2024), accessed: 2025-05-09
11. Johnson, J., Douze, M., Jégou, H.: Billion-scale similarity search with gpus. IEEE Transactions on Big Data **7**(3), 535–547 (2021)
12. Lu, A., et al.: Chip-knn: A configurable and high-performance k-nearest neighbors accelerator on cloud fpgas. In: International Conference on Field-Programmable Technology (ICFPT). IEEE (December 2020). https://doi.org/10.1109/ICFPT51103.2020.00025
13. Luqman, D., Isa, S.M.: Machine learning model to identify the optimum database query execution platform on gpu assisted database. Advances in Science, Technology and Engineering Systems Journal **5**(3), 214–225 (2020)
14. Modular: Fast k-means clustering in mojo: a guide to porting (2023), https://www.modular.com/blog/fast-k-means-clustering-in-mojo-guide-to-porting-python-to-mojo-for-accelerated-k-means-clusteri
15. Modular Inc.: Fast k-means clustering in mojo: a guide to porting. https://www.modular.com/blog/fast-k-means-clustering-in-mojo-guide-to-porting-python-to-mojo-for-accelerated-k-means-clusteri (2024), accessed: 2025-05-09
16. Pandey, A.: Comparing k-nearest neighbors (knn) using sci-kit learn vs custom implementation in python (2023), https://medium.com/@adiipandey3/comparing-k-nearest-neighbors-knn-using-sci-kit-learn-vs-custom-implementation-in-python-84af6f9b
17. Raid, S.: Efficient systems programming with mojo: Python to low-level (2023), https://app.studyraid.com/en/read/12633/409990/simd-instruction-optimization

18. scikit-learn: How to optimize for speed — scikit-learn 1.6.1 documentation. https://scikit-learn.org/stable/developers/performance.html (2024), accessed: 2025-05-09
19. Sivanandhan, S.: Mojo programming language — 68000x faster than python: Programming in mojo — part ii. https://shriramsivanandhan.medium.com/mojo-programming-language-68000x-faster-than-python-programming-in-mojo-part-ii-d162740a2f67 (September 2023), medium blog post
20. Wassenberg, H., Peter, T.: Vectorized and performance-portable quicksort. Software: Practice and Experience **52**(3), 547–561 (2022)
21. Wong, S., Duarte, F., Vassiliadis, S.: A hardware cache memcpy accelerator. In: Proceedings of the 2005 International Conference on Computer Design (ICCD). pp. 234–239. IEEE (2005)
22. YourTechDiet: K-nearest neighbors algorithm in ml: Working (2023), https://yourtechdiet.com/blogs/k-nearest-neighbors-knn-algorithm-in-machine-learning-a-complete-guide/
23. Yu, S., Shah, N., Aggarwal, C.C., Singh, A.K.: Parchain: A framework for parallel hierarchical agglomerative clustering using nearest-neighbor chain. Proceedings of the VLDB Endowment **15**

A Appendix

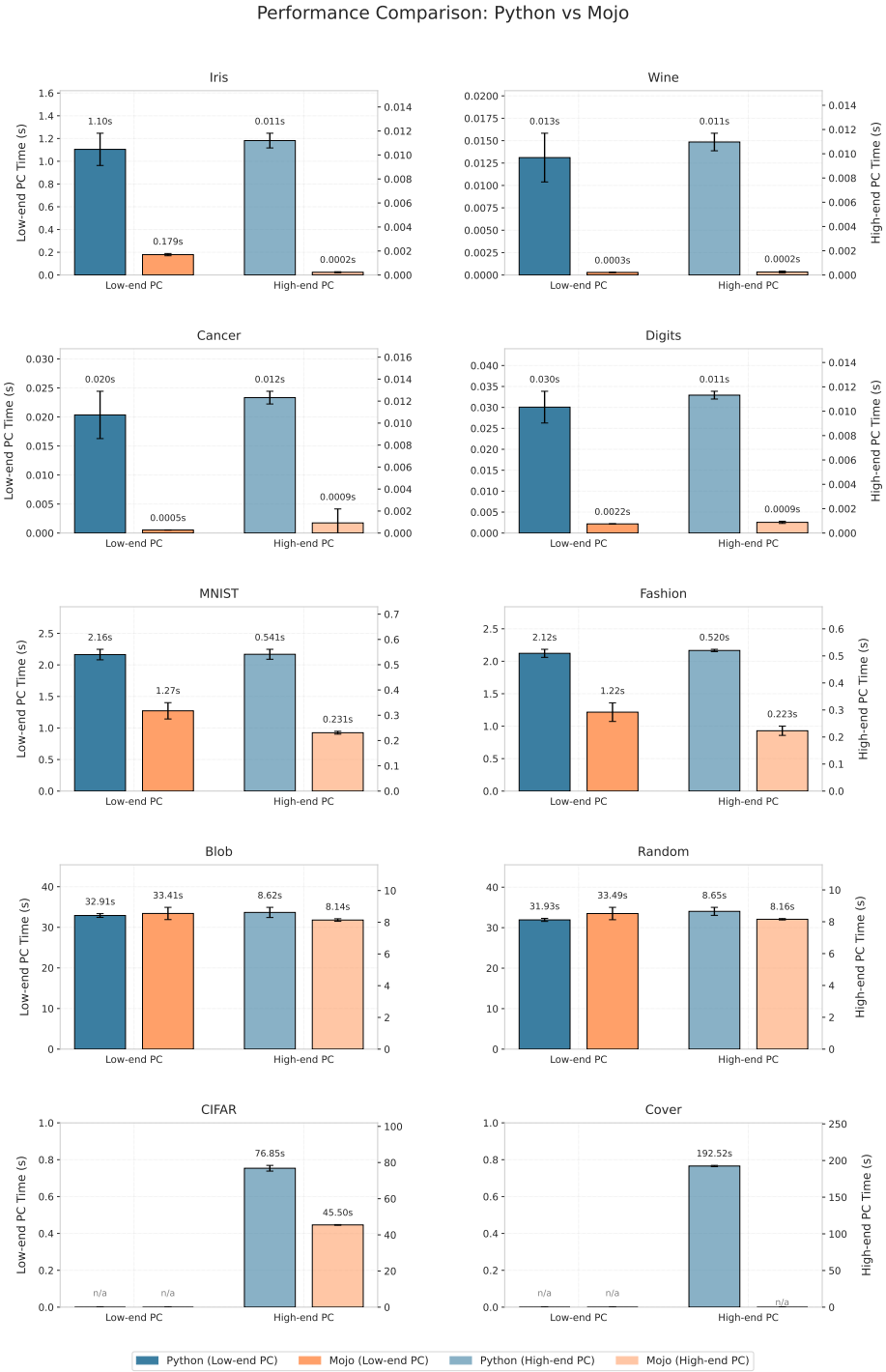

Fig. 4: Performance comparison between Python and Mojo across different datasets and different PC's. Values represent average runtime over ten runs. X-axis is organized by data size, laid out in the methods section.