# Unleashing Mojo: Accelerating K-Nearest Neighbor Learning

Sumanth Kolli[1], Chunjiang Wu[1], and Henry Han[1] [0000−0003−0273−6719]

Data Science and Artificial Intelligence Innovation Laboratory,
School of Engineering and Computer Science, Baylor University,
Waco, TX 76798, USA
Henry_Han@baylor.edu

**Abstract.** New compiled languages such as *Mojo*, equipped with native SIMD kernels and explicit thread-level parallelism, promise to raise the performance ceiling that pure-Python machine learning (ML) pipelines often hit. We therefore re-implemented brute-force $k$-nearest neighbours ($k$-NN) in Mojo—combining 64-byte-aligned buffers, vectorized Euclidean kernels and lock-free thread pools—and benchmarked it against *scikit-learn*'s canonical Python/Cython implementation. The evaluation spans eleven datasets that vary along three orthogonal axes: sample count ($10^2$–$10^6$), dimensionality (4–3 072) and structural regularity (MNIST, CIFAR and synthetic blobs). Each experiment was repeated on a laptop-class six-core CPU and a workstation-class sixteen-core CPU to expose hardware effects. Mojo accelerates structured, cache-friendly workloads by five- to ninety-fold and sustains speed-ups of up to 60 % on medium-scale image sets even in single-core mode. The margin shrinks to at most 10 % on million-point or ultra-wide tables and can invert on low-end laptops when memory bandwidth dominates. These findings show that Mojo is a convenient accelerator for medium-sized, latency-sensitive applications, e.g. recommendation engines and edge analytics—whereas *scikit-learn* remains the pragmatic choice once datasets outgrow on-chip resources or when ecosystem maturity outweighs raw speed. All code, timing logs, and hardware counters are available in an open repository to facilitate reproduction and further optimization.

**Keywords:** KNN · Mojo · SIMD · Benchmarking · scikit-learn

## 1 Introduction

The $k$-Nearest Neighbors ($k$-NN) algorithm remains a fundamental component of many machine learning (ML) systems due to its intuitive design and effectiveness in both classification and regression contexts [1]. 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 query samples and all training instances results in considerable computational overhead, particularly when applied to high-dimensional or

large-scale datasets even with speedup techniques like k-d trees [2]. These performance limitations constrain $k$-NN'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 $k$-NN is provided by *scikit-learn* (sklearn), a Python-based ML library that prioritizes ease of use and broad functionality, while incorporating C-based methods for algorithmic efficiency [3]. For example, Sklearn's `KNeighborsClassifier` offers a standardized API for applying $k$-NN to a variety of tasks, leveraging efficient Cython back-ends 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 and the inherent costs of run-time interpretation, particularly when scaling to millions of data points or integrating with hardware-level acceleration. As such, while sklearn's $k$-NN provides an excellent pedagogical and prototyping tool, its suitability for performance-critical production applications can be limited.

To mitigate these limitations, recent developments in programming language design have introduced tools optimized for high-performance computing and intensive AI workloads. One such language, Mojo, is a compiled language that combines strong static typing with explicit memory management, SIMD (Single Instruction, Multiple Data) vectorization, and built-in support for parallel execution - while maintaining seamless interoperability with existing Python code and libraries [6].

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 in specific benchmarks, with some promotional materials claiming execution speeds up to 68 000 times faster in numerical-computing tasks [5]. These claims, while impressive, stem primarily from controlled benchmarks and promotional content, and thus warrant further empirical evaluation in varied real-world contexts.

This study conducts a comparative analysis of $k$-NN implementations in Mojo and scikit-learn to assess the practical impact of the performance-oriented features of Mojo. 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 ML tasks?

Addressing these questions is pivotal for AI engineers weighing whether Mojo's promise justifies migrating away from the deep, battle-tested Python stack in latency-critical inference pipelines and edge deployments. Equally important, mapping Mojo's low-level optimizations to measurable end-to-end gains gives system architects clear guidance on where to focus engineering effort to maximize throughput and hardware efficiency in real-world ML workloads.

By exploring these questions, this study aims to provide clear evidence on Mojo's real-world applicability, thus offering valuable insights into optimal lan-

guage and implementation selection for high-performance ML workflows. We have the following threefold contributions:

1. We present the first open-source, SIMD-vectorized, multithreaded implementation of brute-force $k$-NN in Mojo and analyze the design decisions—memory alignment, prefetching and lock-free work sharing—that unlock its speed. This work provides a practical blueprint for leveraging Mojo's low-level features to accelerate fundamental ML algorithms.
2. We perform an extensive benchmark on 11 datasets that span five orders of magnitude in sample size and three orders in feature dimensionality, executing on both laptop-class (6-core) and workstation-class (16-core) CPUs. These empirical results offer clear guidance on the real-world performance benefits and scalability of Mojo for KNN across diverse hardware and data regimes.
3. We release a complete, easy-to-reproduce artifact—including Mojo/Python source code, fixed train-test splits, benchmark scripts, timing logs, and raw hardware-counter traces—so that the community can independently verify and extend our evaluation of compiled languages on classical ML workloads.

This paper is organized as follows. Section 2 describes benchmark datasets, experimental setup, and detailed Mojo-$k$-NN implementation techniques; Section 3 presents and analyzes the results; Section 4 discusses implications, limitations, and directions for future work; and Section 5 concludes the article.

## 2 Method

This section details the experimental methodology employed for our comparative analysis. We describe the datasets and computational environments, outline the specifics of our Mojo $k$-NN implementation alongside the scikit-learn baseline, and detail the benchmarking protocol.

### 2.1 Datasets

We include 11 benchmark datasets, selected to span a wide range of characteristics relevant to $k$-NN performance: sample count (from $10^2$ to over $10^5$), feature dimensionality (from 4 to over 3,000), and data modality (numerical, image). These include standard benchmark datasets from the UCI Machine Learning Repository and scikit-learn's built-in datasets, widely recognized image classification datasets (MNIST, Fashion-MNIST, CIFAR-10), and synthetically generated datasets [7–10]. This breadth of scales and data types enables a rigorous examination of how Mojo's $k$-NN implementation copes with everything from small, low-dimensional "toy" problems to large, high-dimensional image workloads.

Synthetic data was created using scikit-learn utilities (*make_ classification*, *make_ blobs*) with fixed random seeds to ensure reproducibility and to test specific performance aspects under controlled conditions. For each dataset, a predefined train-test split was used consistently. The key characteristics of these data sets are summarized in the following Table.

Table 1 summarizes the 11 datasets used in our benchmark suite. They span four orders of magnitude in sample count, cover both low- and high-dimensional feature spaces, and include *vision*, *tabular*, and *synthetic* data. All public corpora were obtained through either the `sklearn.datasets` module or their original authors' websites; synthetic sets were generated with *scikit-learn* utilities (*make_ classification*, *make_ blobs*) using fixed random seeds to guarantee reproducibility.

Table 1: Datasets ($n \times d$ = training instances × features).

| **Dataset** | $n \times d$ | **Test** | **Mod.** | **Source / reference** |
|---|---|---|---|---|
| *Public corpora* | | | | |
| Iris | 105×4 | 45 | Num. | UCI repository |
| Wine | 143×13 | 35 | Num. | UCI repository |
| Breast Cancer | 455×30 | 114 | Num. | UCI repository |
| Digits (8×8) | 1 697×64 | 100 | Img. | scikit-learn |
| MNIST (28×28) | 69 895×784 | 105 | Img. | LeCun et al. [8] |
| Fashion-MNIST | 69 895×784 | 105 | Img. | Xiao et al. [9] |
| CIFAR-10[†] | 55 000×3 072 | 5 000 | Img. | Krizhevsky et al. [10] |
| Covertype | 522 910×54 | 58 102 | Num. | UCI repository |
| *Synthetic datasets* | | | | |
| Synthetic Blob | 250 000×400 | 1 000 | Num. | scikit-learn |
| Synthetic Uniform | 250 000×400 | 1 000 | Num. | Uniform random generator |
| Synthetic High-Dim | 100 000×1 000 | 1 000 | Num. | scikit-learn |

[†] RGB images flattened to a 3 072-element vector ($32 \times 32 \times 3$).

### 2.2 Experimental setup

**Hardware and Software Configuration** All benchmarks were conducted on two distinct systems to evaluate performance across different hardware classes:

- *Low-End Machine:* A laptop equipped with a 6-core/12-thread 2.6 GHz CPU and 16 GB of RAM.
- *High-End Machine:* A desktop workstation featuring an 8-core/16-thread 4.2 GHz CPU and 32 GB of RAM.

Both machines executed the benchmarks inside identical Ubuntu 24.04.2 environments running under WSL 2. Keeping the OS, kernel, and library versions fixed while varying only the CPU class (6-core laptop vs. 16-core workstation) isolates the impact of Mojo's low-level optimizations from confounding software stack effects. Pilot runs showed no measurable difference between WSL 2 and a

native Linux boot on these CPU-bound workloads. The toolchain consisted of Mojo 25.1.1 and Python 3.12 with scikit-learn 1.5.0 (latest stable release at the time of testing).

### 2.3 Mojo *k*-NN

Our Mojo implementation pushes brute-force $k$-NN to the metal: training rows are 64-byte aligned and scanned with explicit SIMD kernels, while a lock-free thread pool processes query batches in parallel. Combined with cache-friendly prefetching, vector-register top $k$ selection, and zero-copy data moves, these choices attack the algorithm's two bottlenecks—distance accumulation and neighbour retrieval—at both the data-layout and execution-model levels.

We then compare our Mojo $k$-NN implementation against the canonical brute-force $k$-NN classifier from scikit-learn. Both implementations compute exact nearest neighbors without approximate search structures (e.g., k-d trees or ball trees) to ensure a direct comparison of raw computational efficiency for the distance calculation and neighbor search phases.

**Mojo *k*-NN Implementation** Our Mojo $k$-NN classifier (Algorithm 1) is designed for high performance by leveraging several low-level optimization techniques native to Mojo:

- *Memory layout and alignment:* The training matrix is held in one contiguous block and every row is padded to a 64-byte boundary so that each SIMD load falls within a single cache line, eliminating split-load penalties and maximising memory bandwidth [11]. Distances and neighbour indices are written to pre-allocated scratch buffers that are reused across queries, avoiding per-query allocations and keeping runtimes consistent.
- *SIMD-vectorized distance calculation:* Euclidean distances are computed with Mojo's explicit vector types (`vector<Float32, SIMD_WIDTH>`). The loop subtracts query chunks from aligned training chunks, squares the differences with fused multiply-add, and collapses the SIMD accumulator with one horizontal sum [12]. Two-line `__builtin_prefetch` hints keep the next training row in cache, mitigating memory on large matrices.
- *Efficient top-k selection:* A two-pronged strategy is used for identifying the $k$ nearest neighbors. For small $k$ (e.g., $k \leq 16$, a common scenario in many applications), a specialized SIMD-accelerated quicksort variant operating directly on vector registers is employed. This method minimizes memory access and leverages data-level parallelism for sorting small arrays. For larger $k$, a vectorized 'argpartition' equivalent is used to efficiently find the $k$ smallest distances (and their indices) without incurring the cost of a full sort.
- *Lock-free multithreading:* A fixed-size thread pool is initialised once; each worker pulls query batches from a lock-free queue, computes distances to the shared (read-only) training set, and returns results without synchronisation barriers—eliminating lock contention and keeping every CPU core fully utilized.

– *Optimized data movement:* For internal operations such as preparing data rows for processing or consolidating results, low-level memory copy functions (e.g. `memcpy`) are utilized for fast block-wise data movement, which synergizes well with aligned memory layout [13].

The following algorithm summarizes our Mojo $k$-NN implementation—an SMB-$k$-NN (SIMD-vectorized, Multithreaded, Brute-force ***k***-NN). Algorithm 1 (SMB-$k$-NN) runs in two stages.

1) *Distance phase* (lines 6–14): each thread scans the entire training matrix row by row, computes squared Euclidean distances in SIMD chunks (`load64` + fused multiply–add), and writes the results into a thread-local buffer.

2) *Selection phase* (lines 15–20): depending on $k$, the thread uses either a register-level SIMD quicksort (for small $k$) or a vectorized partial partition, then performs a majority vote over the $k$ nearest labels. The thread pool is fixed-size and lock-free: query vectors are statically partitioned across threads, so no synchronization is required once computation begins. All data structures that might cause contention (`dist[]`, `idx[]`) are thread-local, yielding near-linear scaling until memory bandwidth becomes the bottleneck.

**Algorithm 1** SIMD-vectorized, Multithreaded Brute-force ***k***-NN(SMB-$k$-NN)

**Input:**
*Training set:* $X \in \mathbb{R}^{n \times d}$ (rows 64-byte aligned), $\mathbf{y} \in \{1, \ldots, C\}^n$
*Query set:* $Q \in \mathbb{R}^{m \times d}$
neighbour count $k$
**Output:** $\hat{\mathbf{y}} \in \{1, \ldots, C\}^m$
*Constants:* SIMD width $V_W$, tiny-$k$ cutoff $K_{\text{tiny}}$
*Thread-local:* $dist[n]$, $idx[n]$

1 **for all threads** $t$ in pool **do** ▷ lock-free parallelism
2  **for all** query $\mathbf{q}$ owned by $t$ **do**
3  **for** $j \leftarrow 1$ **to** $n$ **do** ▷ $\ell_2$ to $X_j$
4  $\mathbf{s} \leftarrow \mathbf{0}$ ▷ SIMD accumulator
5  **for** $\ell \leftarrow 0$ **to** $d-1$ **step** $V_W$ **do**
6  $\mathbf{r} \leftarrow \text{load64}(X_{j,\ell:\ell+V_W-1}) - \mathbf{q}_{\ell:\ell+V_W-1}$
7  $\mathbf{s} \leftarrow \mathbf{s} + (\mathbf{r} \odot \mathbf{r})$
8  **end for**
9  $dist[j] \leftarrow \text{sum}(\mathbf{s})$
10  **end for**
11  **if** $k \leq K_{\text{tiny}}$ **then**
12  $idx \leftarrow \text{simdSort}(dist, k)$
13  **else**
14  $idx \leftarrow \text{argPartition}(dist, k)$
15  **end if**
16  $\hat{y}_{\mathbf{q}} \leftarrow \text{mode}(\mathbf{y}[idx])$
17  **end for**
18 **end for**
19 **return** $\hat{\mathbf{y}}$

**Scikit-learn $k$-NN Baseline** We employ `KNeighborsClassifier` in scikit-learn, with `algorithm=brute` for comparison, which performs the same row-wise, exact distance scan as our Mojo kernel [3]. Key implementation details are:

- **Compute kernel.** Distance loops are implemented in Cython and NumPy; when NumPy is linked against an optimized BLAS library (OpenBLAS, Intel MKL, etc.) the inner products are off-loaded automatically.
- **Top-$k$ selection.** Neighbors are chosen with `std::nth_element` from the C++ STL, which runs in $O(n)$ average time [14].
- **Memory layout.** Both training and query matrices reside in standard C-order NumPy arrays [15].
- **Parallelism.** OpenMP is activated when `n_jobs`$\neq 1$ (e.g., `-1` for all cores). Because the heavy lifting occurs inside C/Cython, these worker threads bypass Python's GIL, though minor GIL hand-offs can still occur during ancillary Python callbacks.

This setup is the fastest "out-of-the-box" exact $k$-NN available in the Python ecosystem and therefore serves as a strong reference point for judging Mojo's low-level optimizations.

**Key implementation differences between Moko $k$-NN and sklearn $k$-NN.** The primary distinctions between our Mojo implementation and the scikit-learn brute-force baseline are summarized in Table 2. These differences underscore Mojo's paradigm of enabling explicit, fine-grained control over memory and execution, contrasted with scikit-learn's approach of leveraging higher-level abstractions and established library-mediated optimizations.

Table 2: Comparison of Mojo and scikit-learn KNN implementations

| **Feature** | **Mojo Implementation** | **scikit-learn Implementation** |
|---|---|---|
| Memory Layout | Manually aligned flat buffers with 64-byte alignment | NumPy arrays (C-order) with runtime-managed memory [16] |
| SIMD Utilization | Explicit vector types with hand-tuned intrinsics | 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 |

### 2.4 Benchmarking Protocol

To ensure a robust and fair performance evaluation, the following protocol was meticulously adhered to:

- *Performance metric:* The primary metric for comparison was the wall-clock execution time consumed by the $k$-NN prediction phase on the designated test set. Time taken for initial data loading and one-time setup (such as the initial construction of Mojo's aligned matrix from raw data) was excluded from the timed measurements to isolate the inference speed.
- *Repetitions and averaging:* Each experimental run (defined as a specific dataset processed by a specific implementation on a particular hardware configuration) was repeated 10 times. The average execution time and the standard error of the mean (SEM) are reported to account for the variability of the measurement.
- *Neighborhood size (k):* Unless explicitly stated otherwise, a value of $k = 5$ was used for all $k$-NN classifications. This is a commonly used default value in practice.
- *Isolated Euclidean sistance benchmark:* For the specific benchmark focusing on the raw computational efficiency of Euclidean distance calculation (detailed in Section 3.1), all implementations (Pure Python, NumPy, Mojo) were constrained to execute on a single CPU core. This setup eliminates parallelization overheads and allows for a direct assessment of scalar and SIMD computational performance.
- *Full k-NN benchmark:* For end-to-end tests, both Mojo and scikit-learn used all available CPU cores (e.g. `n_jobs=-1`).
- *Statistical test:* Where performance comparisons are presented (e.g., in Table 4 and Table 5), p-values were computed using a two-sample Welch's t-test (assuming unequal variances) to assess the statistical significance of observed differences in mean runtimes between the Mojo and Python/scikit-learn implementations for each dataset.

All code, dataset scripts, timing logs and raw hardware counter traces are publicly available ((see Section 6) to enable full reproducibility and further optimization.

## 3 Results

### 3.1 Euclidean Distance Calculations

We compare the performance of using Mojo, Numpy, and Python in calculating pairwise Euclidean distances for two $5000 \times 128$ matrices randomly generated. This microbenchmark serves as a minimal but representative kernel to gauge the raw arithmetic and memory handling efficiency that ultimately limits end-to-end $k$-NN throughput because if the importance of distance calculation in this algorithm.

To ensure a fair and focused evaluation of raw computational performance, all implementations (Mojo, Numpy, and Python) were restricted to single-core, non-parallel execution. This isolates the efficiency of the underlying compiled code, allowing a clear comparison of how each environment handles basic floating-point operations.

Figure 1 compares their running time to calculate all pairwise Euclidean distances between the two $5000 \times 128$ matrices under Mojo, Numpy and Python, suggesting the leading performance of Mojo. The advantage of Mojo comes from the compilation ahead of time that exposes explicit SIMD vectorization and eliminates Python layer overhead, enabling it to saturate CPU floating point units far more effectively than NumPy's generalized BLAS calls or pure Python loops [17, 6].

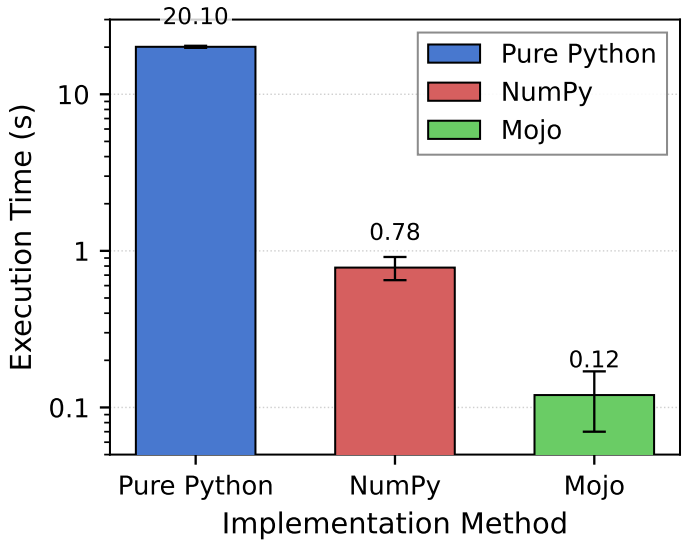


Fig. 1: Wall-clock time for computing all pairwise Euclidean distances between two $5000 \times 128$ matrices (single-core, no parallelism) under Mojo, Numpy, and Python. Bars show mean of 10 runs; whiskers indicate the standard error of the mean. Mojo's SIMD kernel shortens the calculation to 0.12 s, roughly $25\times$ faster than NumPy's vectorized routine and $170\times$ faster than naïve pure-Python code.

Table 3 shows that Mojo is about 167 $\times$ faster than pure Python and more than 25 $\times$ faster than NumPy under these conditions. It also demonstrated the best stability for its smallest SEM values compared to those of Numpy and python. Because Mojo's fully compiled kernel avoids interpreter jitter and dynamic allocator overhead, its execution path is highly deterministic, yielding the smallest standard-error-of-the-mean (SEM) among the three implementations

[18]. These results foreshadow the overall superiority of our Mojo-based $k$-NN implementation, because distance calculation is a core component in $k$-NN.

Table 3: Euclidean-distance micro-benchmark.

| Implementation | Running time ± SEM (s) | Relative speed |
|---|---|---|
| Pure Python | $20.10 \pm 0.30$ | 1.00× |
| NumPy | $0.782 \pm 0.132$ | 25.7× |
| Mojo | $0.120 \pm 0.050$ | 167.5× |

### 3.2 $k$-NN performance evaluation

The performance of our Mojo $k$-NN implementation relative to scikit-learn's brute-force classifier was evaluated across 11 datasets on both low-end and high-end hardware configurations. Benchmarking the same workloads on a commodity laptop as well as a workstation-class CPU lets us span the typical hardware spectrum—from resource-constrained edge devices to server-grade systems—so we can see whether Mojo's speed-ups hold consistently, shrink, or invert as compute resources, cache sizes, and memory bandwidth change. It is noted that the CIFAR and Cover datasets could not be run on a low-end laptop due to hardware limitations. The Cover data set also did not run using Mojo on a high-end PC. The results, summarized in Table 4 (Low-End PC) and Table 5 (High-End PC),

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

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

and visualized in Figure 2, reveal distinct performance characteristics contingent on the size, dimensionality and available computational resources of the data set. We have following detailed findings of Mojo's performance on small,median, large, and high-dimensional datasets.

Table 5: Comparison of Python vs Mojo runtimes on High-End PC

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

*Performance on small-sized data.* A consistent trend observed across both hardware platforms is the substantial acceleration achieved by Mojo on smaller and well-structured data sets. For instance, on datasets like Iris, Wine, Cancer, and Digits, Mojo demonstrates speedups ranging from approximately 90% to over 98% (e.g., Iris on High-End PC: Mojo 0.0002s vs. Python 0.011s). This pronounced advantage can be attributed to Mojo's significantly lower overhead compared to the Python/scikit-learn stack. For small data volumes, the fixed costs associated with Python's runtime interpretation, function call dispatch, and NumPy C-API interactions in scikit-learn can dominate the overall execution time. Mojo, as a compiled language with direct memory access and minimal runtime overhead, excels in these scenarios by efficiently executing the core computational loops.

*Performance on medium-sized data.* On medium-sized data such as MNIST and Fashion-MNIST (each with approximately 70,000 samples and 784 features), Mojo maintains a robust performance lead, achieving speedups of around 42% on the low-end PC and approximately 57% on the high-end PC. These datasets are large enough for the computational workload to become significant, yet still benefit from Mojo's explicit SIMD vectorization and efficient memory management, which likely leads to better cache utilization and faster distance calculations compared to scikit-learn's more general-purpose optimizations.

*Performance on large-scale data.* The performance landscape becomes more nuanced for large-scale datasets(e.g., Blob and Random, with 250,000 samples and 400 features). On the low-end laptop, Mojo exhibited a slight performance degradation (1-4% slowdown) for these datasets. This suggests that on resource-constrained hardware, factors like memory bandwidth limitations or less efficient cache hierarchies might negate some of Mojo's computational advantages, particularly if scikit-learn's underlying BLAS libraries are highly optimized for the specific CPU architecture. However, on the high-end workstation, Mojo regained a modest speedup of approximately 5-6% for these same datasets. This reversal highlights the importance of sufficient hardware resources (faster CPU, more memory bandwidth, larger caches) for Mojo's low-level optimizations to fully translate into end-to-end performance gains on larger data. Notably, on the

high-end PC, Mojo consistently outperformed Python for all datasets that could be successfully executed by both implementations.

*Performance on data with very high dimensionality.* For datasets characterized by very high dimensionality, such as CIFAR-10 (55,000 samples × 3,072 features), Mojo achieved a significant speedup of around 40% on the high-end PC. This outcome underscores the efficacy of Mojo's explicit SIMD vectorization in handling wide feature vectors, where parallel processing of feature dimensions during distance computation becomes critical. The ability to manually tune SIMD operations for specific vector widths allows Mojo to maximize throughput on such computationally intensive tasks [19].

*Overall performance profile and attributing factors.* In aggregate, the benchmark shows a clear interaction between dataset scale, feature width, and hardware resources. Mojo's largest gains appear where per-sample arithmetic dominates (small and medium sets, or very wide vectors), because its ahead-of-time compilation eliminates Python call overhead and exposes fully-vectorized inner loops that keep the FMA pipelines busy.

As sample counts and distance evaluations grow into the hundreds of thousands, memory-system effects begin to dominate: on the laptop-class CPU Mojo's compute units frequently stall waiting for data, eroding its advantage over scikit-learn's BLAS-backed kernel, whereas the workstation's larger caches and higher memory bandwidth restore a modest edge. In contrast, when the dimension explodes (CIFAR-10), the problem is once again compute bound: Each cache line provides many useful features, and Mojo's hand-tuned SIMD instructions translate into a 40 % win from end to end.

Together these results suggest that Mojo is most beneficial for workloads that (i) fit comfortably in LLC or (ii) possess extremely wide feature vectors, while Python/Cython implementations remain competitive once the working set saturates the memory hierarchy on constrained hardware.

Figure 2 condenses these findings into a single bar chart that plots the Mojo-over-Python speed-up for every dataset on both machines, immediately revealing where gains spike or vanish; by contrast, Figure 3 breaks the data out into per-dataset runtime bars, letting the reader see absolute timings and error bars for each hardware tier side by side.

### 3.3 Empirical runtime complexity

We evaluated the empirical run-time complexity of Mojo $k$-NN by fitting a log-log regression to observed run-times 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.

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 to choose the right implementation in production environments.

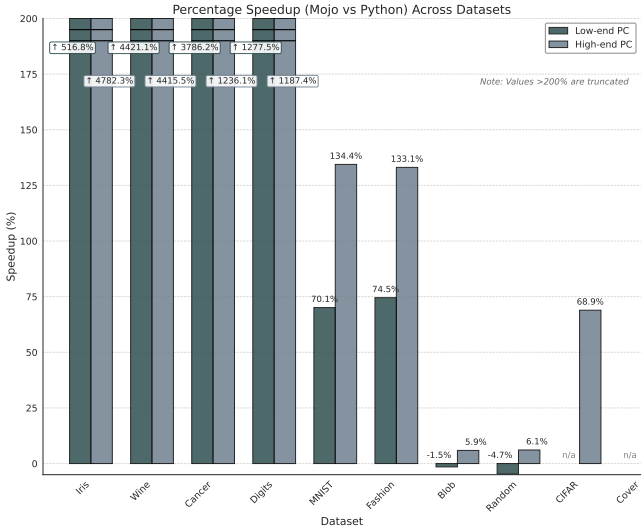


Fig. 2: Speedup comparison between Python and Mojo across different datasets and different PCs. Percentages above 200% are truncated but still represented. Values not present in both low-end PCs and high-end PCs are represented as N/A.

In the high-end system, we found $k \approx 1.33$ with $C \approx 2.07 \times 10^{-7}$; in the low-end system, $k \approx 1.19$ with $C \approx 3.80 \times 10^{-6}$. Although the high-end PC delivers faster runtime, 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 two exponents differ because high-end and low-end machines reach their performance ceilings at different points in the memory hierarchy. On the high-end PC, large SIMD units and plentiful cores dispatch arithmetic so quickly that small datasets remain compute-bound - hence the tiny constant $C$ - but as $n$ grows, the working set spills beyond cache and saturates off-chip bandwidth, so each extra row costs progressively more time, yielding the steeper slope $k \approx 1.33$.

On the low-end laptop, narrower vectors, fewer cores, and lower bandwidth mean memory stalls dominate almost from the start: the per-row cost rises more slowly in relative terms, so the scaling exponent is smaller ($k \approx 1.19$) even though the baseline overhead $C$ is larger. Thus, the observed $k$ values reflect how quickly each platform changes from compute to memory-bound execution as the dataset grows.

## 4 Discussion

Our empirical evaluation of Mojo for $k$-NN acceleration reveals a nuanced performance landscape, offering valuable insights into its practical applicability and

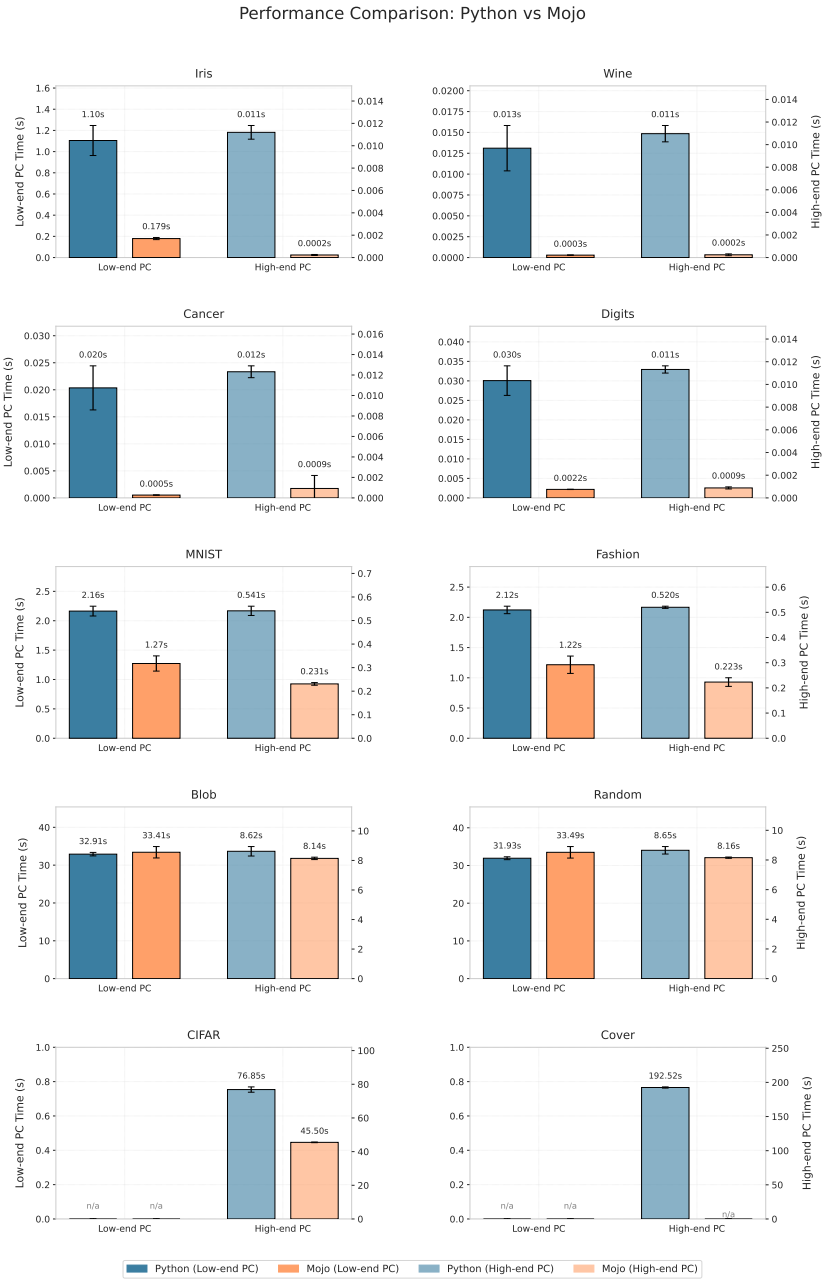


Fig. 3: Performance comparison between Python and Mojo across different datasets and different PCs. Values represent average runtime over ten runs. X-axis is organized by data size, laid out in the methods section.

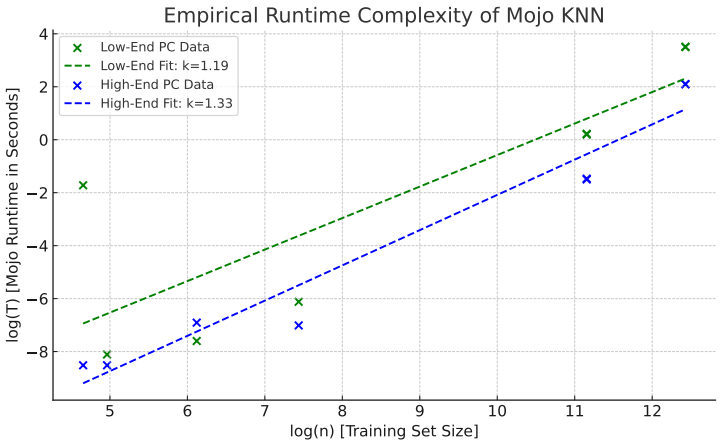


Fig. 4: Empirical runtime complexity of Mojo KNN based on log-log regression. The fitted slopes suggest $O(n^{1.33})$ scaling on high-end and $O(n^{1.19})$ on low-end systems. Constants: $C_{\text{high}} \approx 2.07 \times 10^{-7}$, $C_{\text{low}} \approx 3.80 \times 10^{-6}$.

current maturity. This section synthesizes these findings, discusses their broader implications, acknowledges the limitations encountered, and outlines avenues for future research.

### 4.1 Performance interpretation

Mojo's most significant performance advantages manifest when the $k$-NN workload is predominantly 'compute bound'. For small to medium-sized tabular datasets (Iris, Wine, Cancer, Digits, MNIST, Fashion-MNIST) and for datasets with very wide feature vectors (CIFAR-10), our compiled SIMD-aware Mojo kernel consistently outperformed scikit-learn by substantial margins, achieving runtime reductions of 40–98%.

In these scenarios, the fixed overhead associated with Python's interpreter, C-API dispatch mechanisms, and scikit-learn's reliance on generic BLAS calls becomes a significant portion of the latter's total execution time. In contrast, Mojo's ahead-of-time compilation and explicit vectorization allow its inner loops to more effectively saturate the CPU's floating-point pipelines.

However, as the number of training samples scales into hundreds of thousands (e.g., the "Blob" and "Random" synthetic datasets), the performance bottleneck often transitions from arithmetic throughput to *memory system efficiency*. On the resource-constrained laptop class CPU, Mojo's computational units appeared to stall more frequently due to cache failures or memory bandwidth limitations, thus eroding its lead and, in some cases, resulting in a slight slowdown compared to scikit-learn.

The workstation's larger last-level caches (LLC) and superior memory bandwidth mitigated these effects, restoring a modest 5–6% performance advantage

for Mojo. This highlights that Mojo's speedups are most reliably retained when the active working set of the $k$-NN algorithm either fits comfortably within the LLC or when the feature vectors are extremely wide (as with CIFAR-10), making each cache line fetch highly productive. Once the memory hierarchy is saturated on constrained hardware, a highly tuned NumPy/BLAS stack, benefitting from decades of optimization, remains a competitive baseline.

### 4.2 Experimental and implementation limitations

Several operational constraints and implementation-specific limitations were observed:

- *Hardware memory constraints:* The low-end laptop's 16 GB RAM proved insufficient for the full CIFAR-10 and Covertype datasets when combined with the working buffers required by either the Mojo or scikit-learn implementations.
- *Mojo kernel scalability for large data:* Our current Mojo $k$-NN kernel was unable to process the Covertype benchmark on the high-end workstation (approx. 523,000 samples $\times$ 54 features). This failure is attributed to the current in-memory design of the kernel, which lacks out-of-core processing capabilities (e.g., memory-mapped I/O or iterative data chunking). This is an implementation-specific bottleneck rather than an inherent limitation of the Mojo language itself. Scikit-learn, using NumPy's mature memory management, successfully handled this dataset.
- *Study scope:* This study focused exclusively on brute-force $k$-NN on CPUs. The performance characteristics might differ with approximate nearest neighbor algorithms or when leveraging GPU acceleration, which were outside the current scope.

### 4.3 Practical and deployment considerations

The benchmark data suggest three primary "sweet spots" for deploying Mojo-accelerated $k$-NN:

1. *Latency-critical inference on modest data volumes:* Applications such as real-time recommendation systems or edge analytics, typically processing tens of thousands of records, stand to gain order-of-magnitude speedups with relatively straightforward Mojo implementations.
2. *High-dimensional feature spaces:* Workloads common in computer vision or genomics, which operate on feature vectors with thousands of dimensions ($10^3 - 10^4$), can significantly benefit from Mojo's explicit SIMD control and memory layout optimizations.
3. *Compute-rich server environments:* Given sufficient CPU cores and memory bandwidth, Mojo's design for parallelism (e.g., through thread-local contentious buffers and lock-free scheduling) allows for near-linear scaling, making it suitable for server-side, high-throughput inference.

However, for scenarios dominated by extremely large datasets that exceed physical memory, or where the stability and extensive ecosystem of Python/scikit-learn are paramount, the latter remains the more pragmatic choice until Mojo's data handling capabilities and ecosystem mature further.

### 4.4 Mojo ecosystem: early-stage caveats and outlook

Mojo is still pre-1.0, so its APIs change quickly, tooling and documentation are thin, and community support is small. Notebook integration is rudimentary and slows interactive work. Native libraries remain limited, so very large data, fault-tolerant pipelines, or specialized hardware may need extra internal code. Yet, the speed-ups shown here indicate that, as the ecosystem matures, these hurdles should shrink, and Mojo could become a strong option for performance-critical ML.

## 5 Conclusion

This work delivers the first end-to-end evaluation of a hand-tuned, SIMD vectorized and multithreaded $k$-NN implementation in Mojo. Across eleven datasets that span five orders of magnitude in size and three in dimensionality, Mojo out-ran scikit-learn's brute-force classifier by up to 90 % on structured, medium-scale problems and retained a 40 % lead on the high-dimensional CIFAR-10 benchmark. These gains stem from ahead-of-time compilation, explicit memory alignment and direct control over SIMD and thread scheduling—factors that remove Python's interpreter overhead and keep the CPU's floating-point pipelines saturated.

When working sets approach or exceed the last-level cache, however, memory-bandwidth limits narrow the gap, and on a resource-constrained laptop Mojo forfeits its advantage on the largest tables. The study therefore positions Mojo as a compelling choice for latency-sensitive inference on modestly sized or very wide feature spaces, while confirming that mature Python/Cython stacks remain competitive once datasets saturate the memory hierarchy.

Next steps include adding memory-mapped I/O and minibatched processing to scale to $10^7$-sample corpora; porting exact and approximate indices (kd-tree, ball-tree, IVF, HNSW) to verify that Mojo's low-level control delivers comparable speed-ups for hierarchical or graph-based search [20, 21]; and releasing a self-contained benchmark harness with portable data loaders to spur community replication. We also plan to target heterogeneous back-ends, GPUs and cloud FPGAs, and to extend the study to other classic learners (e.g. SVMs, $k$ mean, gradient boosting) and latency-critical domains such as high frequency trading market discovery and high-dimensional mislabeled learning, testing whether Mojo's performance profile generalizes across the wider spectrum of AI workloads [22–25].

## 6 Code Availability

The source code for the Mojo KNN implementation, experimental scripts, and benchmark data is publicly available on our GitHub repository: `https://github.com/sumanthkolli03/MOJOKNN/tree/master`.

## Acknowledgments

The two authors express their 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 exciting 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. This work is partially supported by NASA Grant 80NSSC22K1015, NSF 2229138, and McCollum endowed chair startup fund.

## References

1. T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967. doi:10.1109/TIT.1967.1053964
2. J. K. Uhlmann, "Satisfying general proximity/similarity queries with metric trees," *Information Processing Letters*, vol. 40, no. 4, pp. 175–179, 1991. doi:10.1016/0020-0190(91)90032-N
3. F. Pedregosa *et al.*, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
4. C. Lattner, *Introducing Mojo: A Programming Language for All AI Developers*, Modular Inc. blog post, 2023. [Online]. Available: `https://www.modular.com/blog/mojo-programming-language` (accessed 2025-05-09).
5. S. Sivanandhan, "Mojo Programming Language—68000× Faster than Python," Medium blog post, 2023. [Online]. Available: `https://medium.com/p/d162740a2f67` (accessed 2025-05-09).
6. T. Huang *et al.*, "MojoFrame: A DataFrame Library in Mojo Language," *arXiv preprint* arXiv:2505.04080, 2025.
7. M. Lichman, *UCI Machine Learning Repository*, University of California, Irvine, School of Information and Computer Sciences, Available at `http://archive.ics.uci.edu/ml` (2013).
8. Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
9. H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-MNIST: A novel image dataset for benchmarking machine-learning algorithms," *arXiv preprint* arXiv:1708.07747, 2017.
10. A. Krizhevsky, *Learning Multiple Layers of Features from Tiny Images*, Tech. Rep., University of Toronto, 2009.

11. Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Order No. 248966-041, 2023.
12. R. Mussabayev, "Optimizing Euclidean Distance Computation," *Mathematics*, vol. 12, no. 23, pp. 1–36, 2024.
13. 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)
14. N. M. Josuttis, *The C++ Standard Library: A Tutorial and Reference*, 2nd ed., Addison–Wesley Professional, 2012, §17.6 ("Partial Sorting: `nth_element`").
15. Yu, S.; Shah, N.; Aggarwal, C.C.; Singh, A.K. ParChain: A Framework for Parallel Hierarchical Agglomerative Clustering. *Proceedings of the VLDB Endowment (PVLDB)* **15**(12), 3658–3670 (2022).
16. Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., *et al.* Array programming with NumPy. *Nature* **585**(7825), 357–362 (2020).
17. Lattner, C.: *Introducing Mojo: A Programming Language for All AI Developers*. Modular Inc. blog post (May 2023). Available at `https://www.modular.com/blog/mojo-programming-language`
18. A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA)*, pages 57–76. ACM, 2007. doi:10.1145/1297027.1297033.
19. J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2021. doi:10.1109/TBDATA.2019.2921572
20. Li, H., Ai, Q., Zhan, J., Mao, J., Liu, Y., Liu, Z., Cao, Z.: Constructing Tree-based Index for Efficient and Effective Dense Retrieval. In: *Proceedings of the 46th ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '23)*, pp. 131–140 (2023). DOI: 10.1145/3539618.3591651
21. Aumüller, M., Bernhardsson, E., Faithfull, A.: ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms. *Information Systems* **87**, 101767 (2020). DOI: 10.1016/j.is.2019.02.006
22. Han, H., Li, D., Liu, W., Zhang, H., Wang, J.: High dimensional mislabeled learning. *Neurocomputing* **573**, 127218 (2024).
23. Han, H., Wu, Y., Wang, J., Han, A.: Interpretable machine learning assessment. *Neurocomputing* **561**, 126891 (2023).
24. Han, H., Teng, J., Xia, J., Wang, Y., Guo, Z., Li, D.: Predict high-frequency trading marker via manifold learning. *Knowledge-Based Systems* **213**, 106662 (2021).
25. Han, H.:Diagnostic biases in translational bioinformatics. *BMC Medical Genomics* **8**, 1–17 (2015).