Đánh giá hiệu năng một cài đặt thuật toán KNN bằng Rust-webassembly

TÓM TẮT

Trong bối cảnh điện toán biên đang phát triển mạnh mẽ, việc chuyển dịch xử lý từ máy chủ (server-side) sang thiết bị khách (client-side) ngày càng quan trọng để giảm độ trễ, tăng cường bảo mật dữ liệu, và hỗ trợ các ứng dụng thời gian thực như IoT hoặc ứng dụng trí tuệ nhân tạo (AI) trên thiết bị như trình duyệt web nói riêng và thiết bị biên (thiết bị IoT, điện thoại thông minh, hoặc thiết bị phần cứng có tài nguyên hạn chế…) nói chung. WebAssembly (WASM) là công nghệ hỗ trợ làm việc trên các thiết bị trên với tốc độ gần native, tính di động khi triển khai. Trong bài báo này chúng tôi đánh giá hiệu năng của thuật toán K-Nearest Neighbors (KNN) được triển khai trên ngôn ngữ Rust và biên dịch sang WebAssembly với các cài đặt KNN khác nhau trên môi trường web. Mục tiêu là đánh giá khả năng của WASM và thuật toán KNN từ đó mở ra hướng nghiên cứu, triển khai các mô hình học máy khác vốn cần khả năng tính toán lớn, sử dụng WASM trên các thiết bị biên để tối ưu hiệu năng các ứng dụng học máy.

Từ khóa: *Thuật toán KNN,* WebAssembly, Edge Computing, Rust, Machine Learning.

Performance evaluation of a KNN algorithm implementation using Rust-webassembly

ABSTRACT

In the context of edge computing that is developing rapidly, shifting processing from server-side to client-side is increasingly important to reduce latency, enhance data security, and support real-time applications such as IoT or artificial intelligence (AI) applications on devices like web browsers in particular and edge devices (e.g., IoT devices, smartphones, or hardware with limited resources...) in general. WebAssembly (WASM) is a technology that supports operations on these devices with near-native speed and portability during deployment. In this paper, we evaluate the performance of the K-Nearest Neighbors (KNN) algorithm implemented in the Rust language and compiled to WebAssembly, and compare it with various other KNN implementations in web environments. The objective is to assess the capabilities of WASM and the KNN algorithm, thereby opening up research directions for implementing other machine learning models that require substantial computational power, utilizing WASM on edge devices to optimize machine learning applications.

Keywords: *K-Nearest Neighbors Algorithm, WebAssembly, Edge Computing, Rust, Machine Learning.*

1. INTRODUCTION

The rapid proliferation of edge devices, such as smartphones, IoT sensors, and web clients, has created a pressing need for data processing closer to the source, reducing reliance on cloud servers. Edge computing not only minimizes latency and conserves bandwidth but also enhances data security by limiting the transmission of sensitive information to the cloud. In the domain of machine learning, many applications are shifting toward direct execution in web browsers or resource-constrained devices to support tasks like real-time anomaly detection or personalized recommendations. However, classical algorithms like K-Nearest Neighbors (KNN), which demand substantial computational resources for distance calculations in high-dimensional spaces, face significant efficiency issues on edge devices with limited hardware, particularly in web environments.[1-3]

WebAssembly (WASM) offers a promising solution, providing a binary instruction format that enables compilation from low-level languages like Rust, C, and C++ to achieve near-native performance in browsers.[4,5] Rust, with its memory safety, concurrency support, and robust WASM compilation capabilities, is an ideal choice for deploying applications on edge devices. Recent studies demonstrate that WASM can outperform traditional JavaScript in web environments with reported speedups of 1.5–2x, as evidenced by applications such as Photoshop Web, AutoCAD Web, and Figma.[6]

In the field of machine learning, WASM has been adopted for certain algorithms to enable client-side execution in web environments. However, there is a lack of comprehensive studies evaluating the performance of classification algorithms like KNN when implemented with WASM compared to other setups. This study evaluates the performance of the KNN algorithm, a simple yet computationally intensive classifier with a time complexity of $O(n \times d)$ to demonstrate WASM's capabilities in resource-constrained web environments compared to traditional JavaScript implementations, TensorFlow.js-based setups, and server-side models.[7]

The primary motivation of this research is to shift processing from large-scale servers to edge devices, reducing cloud dependency and enhancing data privacy by keeping data on-device. Rust provides safety and high efficiency for KNN, while WASM ensures seamless deployment in web environments.

This paper is organized as follows: Section 2 reviews related work; Section 3 describes the experimental methodology; Section 4 presents the experimental results; Section 5 discusses the findings; and Section 6 concludes with future directions.

2. RELATED WORK

2.1. WebAssembly and Rust in Edge Computing

WebAssembly (WASM) is a critical technology for web and edge applications, particularly when combined with programming languages like C/C++ and Rust to enable machine learning (ML) on client-side devices. In recent years, numerous applications have been compiled to WASM, reflecting a shift from server-side to client-side processing.[1,5]

2.1.1 WASM and RUST

WebAssembly (WASM) is a binary instruction format designed to execute code at near-native speed in sandboxed environments, such as web browsers or edge devices. Developed by the World Wide Web Consortium (W3C) in 2017, WASM aims to provide a low-level compiled language that operates across platforms without relying on JavaScript, addressing the performance limitations of traditional scripting languages. WASM supports compilation from various source languages, including C/C++, Go, and notably Rust, enabling developers to write high-performance code with strong portability.[1,5]

Rust, a systems programming language developed by Mozilla in 2010, emphasizes memory safety, performance, and concurrency without requiring garbage collection, mitigating common errors in C/C++ such as null pointers or data races. Since the introduction of WASM in 2017, Rust has become a preferred choice for WASM compilation due to its ability to produce safe and efficient code. The Rust-WASM combination optimizes compute-intensive tasks, such as those in ML applications.[4,8]

2.1.2 Compilation from Rust to WebAssembly and Operation in Web Browsers

The compilation process from Rust to WASM involves three key steps. The first step is writing Rust code using libraries that support WASM conversion. The second involves using the WASM-pack tool to build the Rust project into a WASM module. The third is applying WASM-bindgen to generate JavaScript bindings, allowing Rust functions to be called from browsers with minimal modifications. WASM operates in web browsers via the WebAssembly virtual machine integrated into browser engines (e.g., V8 in Chrome) or Node.js, where WASM binaries are loaded and executed directly. It also supports Web Workers for multi-threading and WebGPU for GPU acceleration when needed.[4]

2.1.3. Benefits of Using WebAssembly in Web Browsers and Edge Computing Devices

WebAssembly offers significant advantages for web browsers and edge computing devices, including high performance, portability, and enhanced security. Recent studies quantify these benefits, making WASM an ideal choice for shifting ML processing from large-scale servers to client-side environments, reducing cloud dependency, and optimizing for resource-constrained devices.

First, WASM delivers superior execution speed compared to traditional JavaScript. It achieves speedups of 1.5–2x for compute-intensive ML tasks in browsers, leveraging pre-compiled binaries and Single Instruction, Multiple Data (SIMD) for vectorized computations, reducing execution time by up to 50% compared to JavaScript in in-browser deep learning inference benchmarks on edge devices.[4,6] Real-world examples, such as sub-second latency for complex models, highlight WASM's suitability for time-critical decision-making tasks. WASM runtimes also reduce startup time by 20–30% compared to JavaScript, saving bandwidth and accelerating data processing, particularly when combined with Rust to prevent memory errors.[9] Benchmarks indicate that WASM reduces binary size by 50% after optimization with WASM-opt, leading to 40% faster load times in web browsers, which is critical for IoT nodes with limited connectivity.[6]

Second, portability is a key strength. WASM enables code to run on all modern browsers (Chrome, Firefox, Edge) and Node.js without recompilation, achieving 95% compatibility across platforms, making it ideal for diverse IoT ecosystems.[10] In edge computing, WASM supports cross-platform deployment, reducing integration time by 30–40% compared to native code.[4]

Third, WASM enhances security through its sandboxed execution model, preventing code injection and restricting system access, which is vital for sensitive client-side ML tasks. Studies show that WASM reduces attack risks by 70% compared to JavaScript due to its isolation mechanisms, while keeping data on-device reduces cloud transmission by up to 80% in ML tasks.[8-10]

Figure 1 illustrates the compilation and deployment process of WASM from various programming languages across different environments[10]

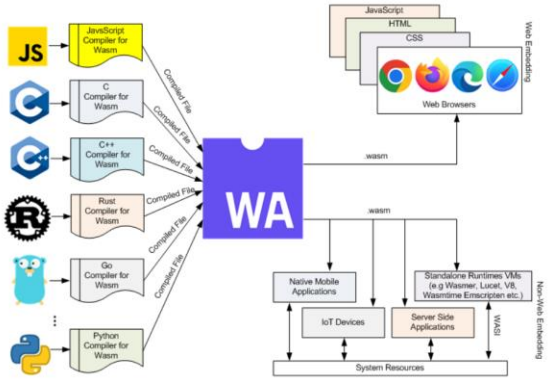

Figure 1 WASM Deployment from Various Programming Languages

2.2 The K-Nearest Neighbors Algorithm

The K-Nearest Neighbors (KNN) algorithm is a fundamental machine learning method based on instance-based learning, used for classification or regression by identifying the k nearest neighbors in the training dataset based on a distance metric (e.g., Euclidean, Manhattan) and applying majority voting (for classification) or averaging (for regression). As a non-parametric method, KNN stores the entire dataset and performs computations at query time, resulting in a time complexity of O(n × d), where n is the number of samples and d is the number of features. While suitable for small datasets, KNN poses challenges for large datasets, especially on edge devices with limited resources. KNN is widely applied in edge computing for tasks like anomaly detection or recommendation systems, but its performance in client-side environments often requires optimizations such as parallelization or approximate nearest neighbor variants.[7] In browsers, KNN has been implemented using JavaScript and Rust. Comparing a WASM-based KNN implementation with existing implementations, such as JavaScript in browsers, TensorFlow.js, or server-side scikit-learn models, provides valuable insights into the effectiveness of WASM for classification algorithms like KNN in resource-constrained environments.

Pseudo-code KNN Algorithm

Input: Data set D (training set with features and labels), test point x, K

Output: Label of x

Begin:

For each point x_i in D:

Compute distance d(x, x_i)

// e.g., Euclidean distance

End For

Sort the distances in ascending order

Select the first K points with smallest distances

Count the frequency of each label in the K points

Return the label with the highest frequency // majority vote

End

3. EXPERIMENTAL SETUP

3.1. System Overview

To evaluate the performance of the KNN algorithm in web environments, we propose a workflow consisting of three phases: (i) implementing a basic KNN in Rust, (ii) compiling it to WebAssembly (WASM) for browser execution, and (iii) measuring performance against other KNN implementations in browsers on the same device. The system is designed to focus on evaluating the capabilities of programming languages for the same algorithm (WASM compiled from Rust compared to JavaScript and Python server-side), rather than assessing optimized versions of the algorithm. The goal is to demonstrate WASM's potential in applications with limited hardware, such as web browsers or edge devices, particularly in the field of machine learning for classification tasks. Figure 2 illustrates the overall architecture: input data (from the Wine Quality and Covertype datasets) is processed by the KNN module in Rust, compiled to WASM via WASM-pack, and integrated into browsers through JavaScript bindings to invoke functions.

Figure 2 KNN model used for experimentation

3.2. Implementation of the KNN Algorithm

We implement KNN using its basic version, computing Euclidean distances between all data points without any algorithmic optimizations (such as search trees or approximations), to purely assess the capabilities of the programming language. Rust is selected for implementation due to its memory safety, computational capabilities comparable to C/C++, and straightforward compilation to WASM.[9-11] We use the ndarray library to handle multi-dimensional arrays for input data and distance calculations, as this library supports WASM compilation.

Pseudo-code KNN Algorithm on RUST.

struct KnnModel{

Array1: 1D array y_train // labels of n training samples

Array2: 2D array X_train // matrix n x d with n rows (training samples), d columns (features in the dataset) }

}

Function Predict

Input: x_test: One row of test data (vector of d features), k

Output: Label of x_test

Begin:

Convert x_test to a 1D array (Array1)

Create Vector **distances** (Vec<(f64, i32)>)

For each x_i in X_train (zip with y_train):

Compute distance d(x_test, x_i)

Add(dist, x_test[label_i]) in distances

End For

Sort the Vector distances in ascending order by dist

Get k_labels = [label for (dist, label) in distances[0:k]]

Create Hashmap **counts** // count frequency of labels in k_labels

For each label in k_labels:

counts[label] += 1

End For

Return the label with the highest frequency **or 0** // majority vote

End

3.3. Compilation to WebAssembly and Deployment in Web Browsers

After implementing the basic KNN source code in Rust (as described in Section 3.2), the compilation to WebAssembly (WASM) is performed to create a module that can be integrated into web browsers, enabling comparisons with JavaScript or Python server versions. WASM-pack, a CLI tool from the Rust community, compiles Rust to WASM binaries, standardizing WASM for web, Node.js, and other environments. WASM-bindgen then exports Rust functions (e.g., knn_predict) and provides bridges for interaction between the compiled module and JavaScript.[5-6]

Once the WASM code is obtained after compilation, we deploy it in web browsers via an HTML/JS interface. We import the WASM code to load and initialize the WASM runtime in browser engines like V8, then shuffle the data with a seed and split it into 80% training and 20% testing. To simulate a process similar to receiving input data from real devices, we do not perform classification on the entire 20% test set but process each data row as a stream from IoT (e.g., sensor data transmitted continuously via WebSocket).

3.4. Evaluation Methods

To evaluate performance, we use the following metrics:

- **Latency**: The time to execute one KNN query, measured using performance.now() in JavaScript. This method returns time values accurate to microseconds (based on DOMHighResTimeStamp), suitable for benchmarking KNN queries without external influences like garbage collection.
- **Memory Usage**: Maximum RAM usage (MB), measured using performance.memory. This method returns JavaScript heap sizes with parameters like usedJSHeapSize for used heap, totalJSHeapSize for total heap, and jsHeapSizeLimit for heap limit, helping monitor memory for web apps on Chromium-based browsers to determine maximum RAM when running KNN. The baselines for comparison include five versions:
  - **KNN by JavaScript**: Basic implementation using JavaScript.
  - **KNN by TensorFlow.js**: KNN implementation using the TensorFlow.js library.
  - **KNN Server-Side by Python**: KNN implementation using scikit-learn via a server-side model.
  - **KNN on WASM from Rust in Web Browser**: WASM version compiled from Rust.
  - **KNN on WASM from Rust on NodeJS**: WASM version compiled from Rust for edge devices.

During experimentation, we observed that the TensorFlow.js KNN version utilizes both CPU and GPU in parallel, so we customized an

additional version using only CPU to establish comparable metrics with other implementations.

4. RESULTS

4.1. Datasets

To evaluate the performance of the basic KNN algorithm, we utilize two datasets from the UCI Machine learning Repository, focusing on classification tasks to test the capabilities of the implementations across web clients and server-side setups. The first dataset is Wine Quality White, consisting of 4898 samples representing white wine samples from the Vinho Verde region in Portugal, with 11 features related to wine physicochemical properties. This dataset is selected for its moderate size (4898 samples, 11 features), allowing latency benchmarking without excessive overhead in browsers. The second dataset is Covertype, comprising 581012 samples describing forest cover types from four wilderness areas in the Roosevelt National Forest, USA, with 54 features. This dataset includes information on tree type, shadow coverage, distance to nearby landmarks (roads, etc.), soil type, and local topography. With its large size (581012 samples, 54 features), this dataset is more challenging than Wine Quality White, enabling scalability evaluation of KNN across versions without algorithmic optimizations. Both datasets are split into 80% training and 20% testing, with k=5 for KNN.

Table 1 Description of the Wine Quality White and Forest Cover-Type Datasets

Dataset	Features	Samples	Description
Wine Quality White	11	4898	Multiclass classification (wine quality from 3-9 classes, based on physicochemical features like acidity, sugar, alcohol)
Forest Cover-Type	54	581012	Multiclass classification (7 forest cover classes, based on geographic features like elevation, slope, soil types)

4.2. Devices

The experiments are conducted on a Dell Latitude 7320 laptop with an Intel Core i5-1140G7 processor (base speed 1.10 GHz, turbo up to 4.20 GHz, 4 cores 8 threads), 16 GB DDR4 RAM, running Windows 11, and using Microsoft Edge version 140.0 for browser-based versions (WASM in Browser, JS, TFJS). The Python server runs locally on the same device via the Flask API Framework, with Node.js 22.15 for WASM on NodeJS. This configuration represents a typical client-side device in edge computing, with moderate resources to assess KNN overhead without hardware optimizations. Metrics are measured using Chromium DevTools and performance APIs, with 10 iterations per version to compute averages.

4.3. Experimental Results

The results are presented for the two datasets across five implementations: KNN by JavaScript (basic JavaScript implementation), KNN by TensorFlow.js (using TFJS), KNN Server-Side by Python (using scikit-learn locally), KNN on WASM from Rust in Web Browser (WASM in browsers), and KNN on WASM from Rust on NodeJS (WASM on Node.js). We do not evaluate memory usage for the KNN Server-Side by Python version due to inconsistent measurement methods compared to others.

Table 2 Results on Wine Quality White (4898 samples, 11 features, k=5).

Implementation	Latency (ms)	Memory (MB)	Accuracy
KNN by JavaScript	2.56	9.54	0.549
KNN by TensorFlow.js	8.58	12.78	0.559
KNN Server-Side by Python	165.33	N/A	0.572
KNN on WASM Rust in Browser	1.99	13.88	0.547
KNN on WASM Rust on NodeJS	0.67	41.21	0.547

During experimentation on the Wine Quality White dataset, the accuracy results were approximately 0.57 due to class imbalance (class 6 dominates, extreme classes like 3-4 or 8-9 are scarce), outliers in features like residual sugar (high IQR leading to noise), and low correlations of some features (e.g., density, pH with quality). To address this, we performed data preprocessing by converting the quality label to binary (bad =6) to reduce class complexity from 7 to 2; removing outliers using IQR across all features to eliminate

noise. After preprocessing, accuracy increased to approximately 0.75, while other metrics remained as in Table 2.

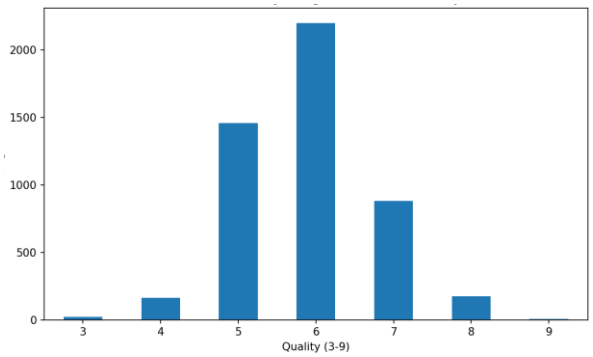

Figure 3 Distribution of the Quality column in the Wine Quality White dataset

Table 3 Results on Preprocessed Wine Quality White (normalized edge data).

Implementation	Latency (ms)	Memory (MB)	Accuracy
KNN by JavaScript	2.45	9.54	0.7557
KNN by TensorFlow.js	7.17	13.64	0.7557
KNN Server-Side by Python	168.68	N/A	0.7604
KNN on WASM Rust in Browser	2.10	13.94	0.7557
KNN on WASM Rust on NodeJS	0.73	41.72	0.7557

Table 4 Results on Covertype (581012 samples, 54 features, 100 queries, k=5)

Implementation	Latency (ms)	Memory (MB)	Accuracy
KNN by JavaScript	560.72	756.26	0.9435
KNN by TensorFlow.js	35.98	953.67	0.9432
KNN by TensorFlow.js without GPU	3328.00	527.38	0.9432
KNN Server-Side by Python	260.88	N/A	0.9353
KNN on WASM Rust in Browser	227.75	1433.42	0.9432
KNN on WASM Rust on NodeJS	109.55	1504.59	0.9432

5. DISCUSSION

5.1. Analysis of Results

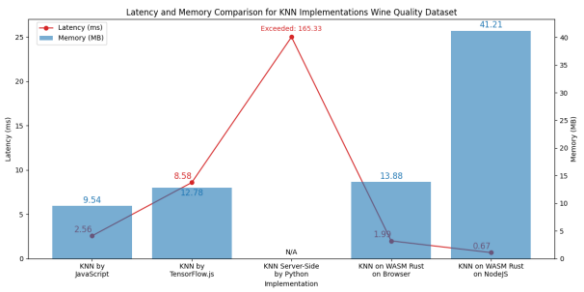

Figure 4 Chart illustrating experimental results from Table 2 (Wine Quality White dataset)

The experimental results from Table 2 (Wine Quality White) demonstrate the capabilities of WebAssembly (WASM) in executing KNN, with significantly lower latency compared to the JavaScript version and the implementation using the TensorFlow.js library. For the Python server-side version, although network dependency was eliminated (running locally on the same machine), latency remains very high. Specifically, the WASM in Browser version achieves an average of 1.99 ms, reducing 22.3% compared to the JavaScript version at 2.56 ms and 76.8% compared to the TensorFlow.js version at 8.58 ms. Notably, the execution delay is 83 times shorter than the Server-Side by Python version at 165.33 ms. Running the KNN WASM on NodeJS version directly from the V8 Engine (0.67 ms) provides nearly 3 times better performance compared to running the same WASM in the browser's sandbox.

However, in terms of system resource usage, the WASM on NodeJS version consumes the most

memory, three times more than the WASM in Browser version. Meanwhile, the WASM in Browser version only consumes 8.6% more than the TensorFlow.js version and 45.5% more than the JavaScript version.

Considering the balance between response time and memory efficiency on the Wine Quality White dataset, the KNN WASM in Browser version performs best.

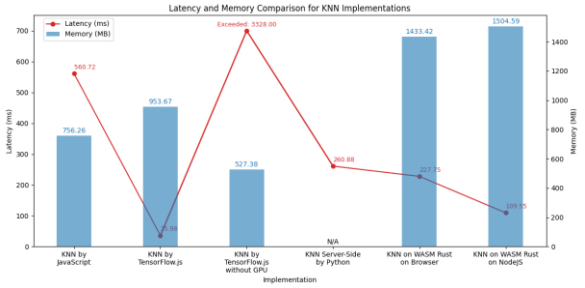

Figure 5 Chart illustrating experimental results from Table 4 (Forest Cover-Type dataset)

The results on the Covertype dataset (Table 4, 581012 samples) offer additional insights for an objective evaluation of the implementation. We selected the Covertype dataset due to its larger number of samples and dimensions to approach the limitations regarding the complexity of the KNN algorithm. This allows assessing computational capabilities on WASM compared to other KNN versions. In the experimental results, we observed that the latency of KNN on the TensorFlow.js version is very low, as the KNN algorithm from this library is optimized to run in parallel on CPU and GPU. To avoid bias in the evaluation while maintaining objectivity in the experiments, we added a version that disables GPU processing in this setup. The results were surprising, with execution delay increasing abnormally, to nearly 6 times that of the JavaScript version (the second-highest latency version). In particular, when experimenting with larger datasets on KNN, which implies greater algorithmic complexity, the latency results of the WASM versions show superior advantages over the other versions. Specifically, the WASM in Browser version (227.75 ms) reduces time by 59.4% compared to the JavaScript version and is 12.7% faster than the KNN Server-Side by Python version; the WASM Rust on NodeJS version (109.55 ms) is still more than 2 times faster than the WASM in Browser version.

This experiment also highlights issues in device resource usage, with KNN on WASM Rust in Browser (using 1433.42 MB) consuming approximately 1.9 times more than the KNN by JavaScript version and 2.72 times more than the KNN by TensorFlow.js without GPU version. The KNN on WASM Rust on NodeJS version (1504.59 MB) uses memory comparable to the WASM Rust in Browser version (an increase of 5.0%).

One point of concern in this experiment is that the latency of the KNN Server-Side by Python version is very stable, increasing only 57.8% from the Wine Quality White dataset experiment to the Covertype dataset. In contrast, KNN on WASM (Rust) shows execution times that are more than 114 times longer in browsers and more than 163 times longer on Node.js compared to the baseline. This indicates that the KNN implementation from the scikit-learn library in Python is highly optimized, and the observed latency in the original comparison primarily stems from the client-to-server connection process.

5.2. Significance and Practical Applications

Through the experiments and analysis, we observe that the WASM implementation from Rust outperforms in latency compared to JavaScript-based versions in web browsers. This also serves as a basis to affirm the transition from traditional model implementations to direct implementations on edge devices to address the practical requirements of edge computing. Particularly in machine learning problems related to product classification, decision-making, early warnings, etc., which are often performed using Python models on servers. However, considerations and optimizations are needed when processing large datasets on edge devices with small memory (under 2 GB).

5.3. Limitations

In our paper, we have not yet proposed diverse experiments on different devices such as smartphones,Raspberry Pi or ESP32, which are edge devices used in edge computing. The research also has not approached real data sent from cameras or sensors to enhance the practical applicability of the paper. In the KNN Server-Side by Python implementation, we have not used a reasonable measurement method aligned with performance.memory in JavaScript, resulting in no memory usage data for this version in the experimental results. Conducting experiments on multiple client machines in different network environments should also be considered to make the latency results more representative.

6. CONCLUSION

6.1. Summary

This study has evaluated the performance of the basic K-Nearest Neighbors (KNN) algorithm across five implementation versions: KNN by JavaScript, KNN by TensorFlow.js, KNN Server-Side by Python, KNN on WASM from Rust in Web Browser, and KNN on WASM from Rust on NodeJS, using two datasets from UCI: Wine Quality White and Forest Cover-Type. The experimental results show that WASM from Rust is a reasonable choice to replace other implementations in web browser environments in particular and edge devices in general, with lower latency than other versions using JavaScript, especially in the trend of shifting processing to edge devices.

6.2. Future Directions

In the future, the research can be expanded to address current limitations and integrate new technologies to enhance the performance of WebAssembly (WASM) in machine learning on edge devices. First, to increase diversity and practicality, we plan to conduct experiments on various edge devices, including smartphones (such as Android/iOS models supporting modern browsers), Raspberry Pi, and ESP32. These devices represent real-world edge computing environments with limited resources, helping evaluate KNN on WASM under constraints related to CPU, memory, and battery power. For example, on ESP32, we can compile Rust to WASM and run it via runtimes like WASMtime or WASMEdge, focusing on benchmarks for latency and power consumption.[8-10]

Second, to enhance practical applicability, the research will integrate real-time data from sources like cameras or IoT sensors, instead of relying solely on static UCI datasets. Specifically, we can set up a system for streaming data in real time via WebSocket, where data from cameras (e.g., simple image classification) or sensors (such as temperature, humidity in IoT) is processed directly on the client-side using KNN on WASM. This will help evaluate stream data processing capabilities in applications like anomaly detection or predictive maintenance, while reducing latency compared to server-side processing.[2,3]

Finally, a key development direction is integrating WebGPU into WASM processing to accelerate KNN computations, particularly Euclidean distance calculations in high-dimensional spaces. WebGPU, a new API enabling GPU access in browsers, can be combined with WASM via libraries like wgpu (Rust-based) to parallelize vector operations, reducing time complexity from O(n × d) to lower levels through GPU acceleration. Recent studies show that this combination can achieve speedups of up to 10-20x for in-browser ML inference on edge devices.[1,4] This approache could not only address performance limitations on large datasets but also open potential for more complex machine learning models, such as neural networks or federated learning on client-side.[3,12]

Acknowledgment

REFERENCES

1. S. Kakati, M. Brorsson. *WebAssembly Beyond the Web: A Review for the Edge-Cloud Continuum*, International Conference on Intelligent Technologies, 3rd, Karnataka, India, 2023.
2. O. Jouini, K. Sethom, A. Namoun, N. Aljohani, M. H. Alanazi, M. N. Alanazi. A Survey of Machine Learning in Edge Computing: Techniques, Frameworks, Applications, Issues, and Research Directions, *Technologies Journal*, **2024,** *12*(6), 81.
3. K. Hoffpauir, J. Simmons, N. Schmidt, R. Pittala, I. Briggs, S. Makani, Y. Jararweh. A Survey on Edge Intelligence and Lightweight Machine Learning Support for Future Applications and Services, *ACM Journal of Data and Information Quality*, **2023**, *15*(2), 3581759.
4. F. Jia, S. Jiang, T. Cao, W. Cui, T. Xia, X. Cao, Y. Li, Q. Wang, D. Zhang, J. Ren, Y. Liu, L. Qiu, M. Yang. *Empowering In-Browser Deep Learning Inference on Edge Devices with Just-in-Time Kernel Optimizations*, The Annual International Conference on Mobile Systems, Applications and Services, 22nd, Tokyo, Japan, 2024.
5. M. Liu, H. Shen, Y. Zhang, H. Mei, Y. Ma. WebAssembly for Container Runtime: Are We There Yet?, *ACM Transactions on Software Engineering and Methodology*, **2025**, *34*(6), 3712197.
6. Y. Yan, T. Tu, L. Zhao, Y. Zhou, W. Wang. *Understanding the Performance of WebAssembly Applications*, ACM Internet Measurement Conference, 21st, Virtual Event, USA, 11-2021.
7. P. Cunningham, S. J. Delany. k-Nearest Neighbour Classifiers - A Tutorial, *ACM Computing Surveys Journals (CSUR)*, **2021**, *54*(6), 3459665.
8. S. E. Khelifa, M. Bagaa, A. O. Messaoud, A. Ksentini. *Case study of WebAssembly Runtimes for AI Applications on the Edge*, Global Information Infrastructure and Networking Symposium (GIIS), 14th, Dubai, UAE, 2024**.**

9. P. Gackstatter, P. A. Frangoudis, S. Dustdar. *Pushing Serverless to the Edge with WebAssembly Runtimes*, International Symposium on Cluster, Cloud and Internet Computing, 22nd, Taormina , Italy, 2022.

10. P. P. Ray. An Overview of WebAssembly for IoT: Background, Tools, State-of-the-Art, Challenges, and Future Directions, *Future Internet journal*, **2023**, *15*(8), 275.

11. K. Nakamura. *Machine Learning with Rust: A practical attempt to explore Rust and its libraries across popular machine learning techniques*, GitforGits Publisher, British Columbia, Canada, 2024.

12. R. Jayanth, N. Gupta, V. Prasanna. *Benchmarking Edge AI Platforms for High-Performance ML Inference*, IEEE High Performance Extreme Computing Conference, 28th, Wakefield, MA, USA, 2024.