National College of Ireland

Browser-Native Machine Learning: WebAssembly and WebGPU-Accelerated Training for Classical Text Classification Models

MSc Research Project
MSc Data Analytics

Shashidhar Yaligar
Student ID: x23426047

School of Computing
National College of Ireland

Supervisor: Sallar Khan

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: Shashidhar Yaligar

Student ID: X23426047

Programme: MSc Data Analytics **Year:** 2025 - 2026

Module: MSc Research Project

Supervisor: Sallar Khan

Submission Due Date: 28/01/2026

Project Title: Browser-Native Machine Learning: WebAssembly and WebGPUAccelerated Training for Classical Text Classification Models

Word Count: 10518 **Page Count** 22

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Shashidhar Yaligar

Signature: .....

28/01/2026

Date: .....

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	✓
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	✓
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	✓

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Browser-Native Machine Learning: WebAssembly and WebGPU-Accelerated Training for Classical Text Classification Models

Shashidhar Yaligar
x23426047
M.Sc. Data Analytics
National College of Ireland

Abstract

This paper explores the viability, performance properties, and design needs of the possibility of training a complete machine learning in a web browser using WebAssembly (WASM) and the newly announced WebGPU interface. Although similar inference has been investigated in the recent literature, in-browser training of machine learning models has not been investigated extensively. To fill this gap, the paper creates a full, client-side text classification system, which does preprocessing, vectorisation, model training and evaluation without the use of the server. Itis a system that was programmed in Rust-compiled WASM and executed using WebGPU compute shaders to execute a multinomial logistic regression model. Two other models are used: a linear Support Vector Machine (SVM) and a shallow Multi-Layer Perceptron (MLP) which are used as baselines in the the given deterministic pipeline to enable controlled comparisons. An equal number of BBC News entries are sampled to test accuracy-time trade-offs, computational stability and memory behaviour among model types. Documents are preprocessed in a uniform pipeline such as normalization, rule-based tokenisation and converting them to a 5,000 term bag-of-words representation to allow reproducibility. According to experimental results WebGPU-accelerated logistic regression is reliably converging and has 84% accuracy and 84.26% macro-F1, which is very similar to MLP baseline (82% accuracy). The simple hinge-loss optimisation without regularisation (SVM baseline) has a very low performance (20% accuracy) under the high-dimensional sparsified text features. The measurement of training time shows the advantage of WebGPU in efficiency, where the training of logistic regression would take 2.5 seconds, whereas it took 19 seconds to train the CPU-bound MLP. The results prove that meaningful, stable, and competitive text classification models can be very easily trained completely in-browser despite limited data and restricted hyperparameter optimization. It provides a modular, reproducible WebGPU-based architecture, a comprehensive analysis of WebGPU compute behaviour, and support that classical ML models may effectively run on the client-side to support privacy-preserving edge intelligence. The resulting architecture gives a basis to further research on larger datasets, more extensive feature engineering, more optimisation schemes, and fully GPU-implemented classical models.

1 Introduction

The WebAssembly (WASM) and the WebGPU API have greatly enhanced the computing power of web browsers to the point that workloads that previously could only be effectively implemented in a native environment can now be effectively implemented on the client side. WASM offers a stable high-performance implementation layer to compiled languages like Rust, whereas WebGPU offers the ability to access general-purpose GPU computing by manipulating buffers, shaders, and compute pipes explicitly. Collectively, these technologies allow machine learning systems to do training and inference on the browser with low latency, enhanced data privacy and easier deployment on heterogeneous platforms. The best test case to evaluate this feasibility is through text classification in the sense that tasks like routing documents, filters and the identification of the topic to be dealt with are all computationally intensive because they involve operations that are numerically intensive. Based on a known benchmark, in this case, a balanced subset of the BBC News dataset, the performance of browser-native training can be assessed on a systematic basis.

Despite the growing interest in studying browser-based inference, little research has been done to study whole training workflows with WebGPU. The extra computational complexity of training over

inference is that gradient updates, normalization, and high dimensional feature operations have to be performed each epoch. The previous studies have thus not completely evaluated the comparison of WebGPU to CPU-based training of classical models, nor the effect of implementation choices, including vocabulary size, dispatch configuration, buffer layout, and batching strategies on accuracy, convergence stability, and the overall training time in browser settings. This literature gap gives the impetus to this study.

To solve this, an all-browser based machine learning system was written, using it to ingest datasets, text preprocess, bag-of-words vectorisation, model training and evaluation all in the user browser. An actual GPU-accelerated multinomial logistic regression model was developed based on WebGPU compute shaders, and two other models, a one-vs-rest SVM linear and a shallow MLP were developed as CPU-based baselines in Rust/WebAssembly. To allow indirect comparisons of models a lightweight browser interface was created that can show accuracy, macro-precision, macro-recall, macro-F1 scores, confusion matrices and training times.

An important design was to adopt all three of the models: GPU logistic regression, SVM, and MLP, as a single Rust/WASM pipeline. This guaranteed reproducible preprocessing, feature representation, hyperparameters and data splits between models, allowing controlled assessment of computational behaviour. Logistic regression was chosen to be fully accelerated with GPUs due to its operations being matrix multiplication, softmax, and first-order optimization which are well-structured on the compute shaders, and it can be easily observed the memory residency, dispatch patterns, and numerical stability on the GPU. The structure is also extensible and modular to accommodate the future gpu-based models without changing the orchestrating elements.

Generally, the thesis assesses the viability and performance of WebGPU-based training of classical text classification and defines the accuracy-time trade-offs of browser-native machine learning. Experimental findings emphasize the role played by vocabulary size, feature scaling, batching and transfers of device-host memory. This system eventually delivers a repeatable research paradigm in the future and empirical proof of the capability of browser-native ML to act as the foundation in privacy-preserving, low-latency analytics applications.

1.1 Background and Motivation

WebAssembly (Wasm) and WebGPU have made the browser a serious computing environment. Wasm provides a predictable near-native CPU performance to languages like Rust, whereas WebGPU provides modern GPU compute (buffers, pipelines and compute shaders) within the browser. They can be used together to support end-to-end machine-learning processes, including training, in the absence of server requirements.

Traditional ML systems get to store calculation on servers: customers send their data, the model is implemented in a remote location, and the outcomes are provided. The trend brings about a risk of privacy, network latency, and single points of failures. Browser-native training, in contrast, stores information on the device and makes privacy-preserving ML at the edge possible, and cuts down bandwidth and operational costs. Although more recent references demonstrate high-throughput in-browser inference with WebGPU (e.g. transformer inference with kernel and memory optimisations), little has been done with full training. Training introduces gradient computation on the per-epoch scale, numerically stable reductions (e.g. log-sum-exp to softmax), updates to parameters and management of optimizer state-all within browser constraints such as task scheduling budgets and sandboxed memory.

A practical field of experiment in determining training viability is text classification. It is commonplace (routing, moderation, topic tagging), can accept classical linear models which are compact and interpretable, and lacks a one-to-many mapping to vectorised GPU primitives (matrix-vector/matrix operations). On the premise that the BBC News data is representative, this paper explores the possibility of training a browser-resident model to attain acceptable accuracy and enduring wall-clock performance on realistic training strictures.

In this thesis, a browser-native training pipeline consisting of WebGPU-based multinomial logistic regression and CPU-based SVM and MLP baselines compiled to Wasm are implemented. To allow controlled comparisons, the pipeline has the same preprocessing, characteristics, divisions, and metrics in all the models. Empirical evidence concerning the feasibility, stability, and performance properties of WebGPU-accelerated training of classical text classifiers is achieved by the resulting system, and a template to be followed in the future browser-based ML systems is provided.

1.2 Dataset and Characteristics

A dataset from the BBC News has been used as the standard to evaluate this work. A multi-class classification task exists due to the five categories of topical content (business, entertainment, politics, sport and technology). The SVM and MLP models are CPU-based implementations that are run within WebAssembly. They do not run WebGPU, but their preprocessing are the same, feature matrices are the same, and update loops are deterministic, which makes them fairly comparable. The length of the processed documents are also relatively small (about 15 tokens on average) yet still sufficient to cause some degree of class overlap and dimensionality issues. All the raw documents included within the dataset have therefore undergone the following processes; preprocessing, tokenization and conversion to a bag-of-words format with a vocabulary of 5000 terms. This common format enables comparisons to be made between the performance of the GPU logistic regression method and the GPU hybrid SVM and MLP methods with respect to both their ability to converge and their computational requirements.

1.3 Research Questions and Objectives

The research questions that guide this thesis are: What are the boundaries of the performance of WebGPU-accelerated training in a browser-based environment versus the performance of CPU-based training for classical text classification? Under what architectural and implementation conditions is WebGPU-accelerated training feasible and efficient?

The objectives of this research are:

1. Develop an accelerated multinomial logistic regression model using WebGPU to train the model completely from start to finish on the client side (in the browser) with Rust compiled Web Assembly and WebGPU compute shader code.
2. Implement hybrid GPU/SVM/MLP versions of each model in the same workflow, so all three models are trained on the same data, have the same preprocessing, same vocabulary and use the same data splits.
3. Develop a web-based user interface that will take the BBC News Dataset, pre-process the text, transform the pre-processed text into a bag-of-words representation, train the models, calculate metrics for how well the models perform and display the results.
4. Compare model performance using precision at class level (macro-precision), recall at class level (macro-recall), F1 score at class level (macro-f1) and training time for each of the three models.
5. Investigate the impact of architectural decisions (vocabulary size of 5000 terms, L2 normalization, batch processing and WebGPU buffer residency) on training performance and stability.
6. Provide a reproducible browser-based framework for future GPU-accelerated machine learning research.

1.4 Summary of Contributions

The contributions of this thesis include:

- A web-based native machine learning flow utilizing WebAssembly and WebGPU's compute shader for training a multinomial logistic regression on text with the acceleration of the WebGPU's GPU.
- Three (SVM and MLP) hybrid GPU models are created using Rust and WebAssembly to allow for comparison of different execution methods while maintaining same pre-processing and feature matrix operations.
- Accuracy, macro F-1 scores, confusion matrices and train time were measured for each model on the BBC News balanced dataset.
- The proposed system architecture can be used as a base to develop reproducible and expandable systems for client-side analytics and future GPU based machine learning model implementations.

2 Literature Review

2.1 Background

In this Section we will present an overview of the literature related to browser based machine learning, with special emphasis in three areas: WebAssembly and WebGPU, and edge computing as privacy preserving machine learning paradigms. The literature reviewed here presents the theoretical background for this research project, and allows us to identify the gap that motivates the development of a Rust based, browser native, text classification system.

2.2 Web Technologies for High-Performance Computation

2.2.1 WebAssembly: High-Performance Computing in Browsers

WebAssembly (Wasm) is one of the most promising technologies for building high-performance web applications. In a comprehensive experimental evaluation of WebAssembly performance, Jangda et al. (2019) compared WebAssembly performance with native code performance using SPEC CPU benchmark suite. They defined Browsix-Wasm as the testing framework for evaluating WebAssembly performance and reported that WebAssembly applications have an average performance penalty of 45% to 55% compared to native code, and that the main factors contributing to this penalty were register pressure, indirect calls and missing compiler optimizations. This finding is significant because it shows that the performance penalty of WebAssembly is potentially addressable with better tooling and compiler optimizations. Therefore, WebAssembly is a potential platform for deploying high-compute intensive applications in the future. There is increasing empirical evidence that WebAssembly is feasible for production deployment. Early experiments focused on WebAssembly performance, however, later research reports have shown that the WebAssembly run time is improving steadily in all major browsers. The fact that both Google, Microsoft and Mozilla have already adopted WebAssembly for performance critical applications, demonstrate its maturity for real world deployments.

2.2.2 WebGPU: Massively Parallel Computing in Browsers

The introduction of WebGPU represents a fundamental shift in how computations are performed in the browser, allowing developers to directly access the power of the user's GPU to perform general purpose computing. The W3C (2023) WebGPU specification defines a modern graphics and compute API that allows developers to efficiently access the capabilities of the GPU of any hardware vendor. Unlike its predecessor WebGL, which was mainly designed for graphics rendering, WebGPU allows the developer to create compute shaders to enable massively parallel processing for machine learning workloads. Recent research has demonstrated the feasibility of WebGPU for complex machine learning tasks. Ma et al. (2025) presented WeInfer, a system that uses WebGPU to perform large language model inference in web browsers. Using buffer reuse strategies and asynchronous pipeline management they achieved a speedup of 3.76 times compared to existing browser based solutions. Their results show that WebGPU can support the heavy computational requirements of large-scale deep learning models entirely inside the browser, thus showing that WebGPU can be used to perform even the computationally most demanding machine learning tasks. Building on this success, Jia et al. (2024) proposed nnJIT, a JIT kernel optimization system for in-browser deep learning. At MobiSys 2024, Jia et al. (2024) presented their solution to the problem of varying capabilities of heterogeneous devices using tensor-web compiling co-design and web-specific lite kernel optimization. With the help of efficient compilation strategies, Jia et al. (2024) could achieve a speedup of 8.2 times within 30 seconds of initialization, demonstrating that sophisticated optimization techniques can lead to competitive performance for browser native ML systems. This is very relevant to our research because it shows that, with the right optimization techniques, competitive performance for browser native ML systems can be achieved.

2.2.3 Historical Perspective: Browser Based Deep Learning

To understand the importance of WebGPU, it is instructive to consider the situation of browser based machine learning prior to WebGPU. Ma et al. (2019) conducted the first extensive experimental comparison of deep learning in web browsers and native Python frameworks, assessing seven JavaScript-based frameworks including TensorFlow.js, WebDNN, and Keras.js. Ma et al. (2019) found substantial differences in performance between browser based and native Python frameworks for deep learning, with browser based implementations being 2 to 10 times slower than native Python frameworks depending

on the model architecture and the hardware configuration. The reasons for the poor performance of browser based deep learning identified by Ma et al. (2019) were inefficient use of WebGL for compute tasks, runtime overhead due to JavaScript execution and lack of optimized operator implementations. The findings of Ma et al. (2019) motivated the creation of WebGPU as a more suitable platform for compute tasks. The dramatic difference between the findings of Ma et al. (2019) and the recent successes with WebGPU-based systems (Ma et al., 2025; Jia et al., 2024) clearly illustrate the profound impact of modern browser APIs on the feasibility of performing machine learning in browsers.

2.3 Machine Learning for Text Classification

2.3.1 Classical Machine Learning Techniques

Text classification is a fundamental task in natural language processing, and has numerous applications including spam detection, sentiment analysis etc. Hassan et al. (2022) provided a detailed evaluation of the performance of classical machine learning algorithms for text classification, and compared the performance of k-Nearest Neighbours (k-NN), Logistic Regression (LR), and Support Vector Machines (SVM) on several datasets. The results obtained by Hassan et al. (2022) indicate that k-NN achieved a classification accuracy of 98.5%, while logistic regression achieved accuracies greater than 95% in news classification tasks. Joachims (1998) first applied Support Vector Machines (SVMs) to text classification tasks, and demonstrated that SVMs outperform the existing methods of that time, including Naive Bayes and Rocchio classifiers, in terms of classification accuracy, and that SVMs also have the additional advantage of automatically determining hyperparameters. Due to their solid mathematical foundations in statistical learning theory and their ability to provide good classification accuracy, SVMs remain one of the most widely used classical machine learning algorithms for text classification tasks. The continuous good performance of classical machine learning algorithms such as SVM and logistic regression is of particular interest for browser based implementations. Unlike deep learning models that need millions of parameters and substantial computational resources, classical algorithms have competitive classification accuracy with much smaller memory footprint and inference cost. This makes classical algorithms very suitable for edge computing scenarios where resource constraints are critical.

2.3.2 GPU Acceleration of Classical ML Algorithms

The acceleration of machine learning algorithms through GPU computing has a long history that precedes today's browser technologies. Catanzaro et al. (2008) were among the first to explore the possibility of using graphics processors to accelerate the training of Support Vector Machines (SVMs), achieving speed-ups of up to 9 times compared to highly optimized CPU implementations. Their research provided the first insights into the possible parallelization of SVM training. For our research, this previous work is particularly relevant since it shows that the acceleration of classical ML algorithms through GPU computing is a well established technique in native computing environments. Our challenge is to adapt these techniques to the characteristics and limitations of browser based WebGPU implementations, where memory models, synchronization mechanisms and shader types are very different from those available in native GPU programming interfaces such as CUDA and OpenCL.

2.4 Privacy-Preserving Machine Learning at the Edge

2.4.1 Federated Learning and Edge Intelligence

The integration of Artificial Intelligence and Edge Computing has generated new paradigms for developing machine learning applications that preserve user's privacy. Deng et al. (2020) provided a comprehensive review of Edge Intelligence, defining a taxonomy that distinguishes between "AI for Edge" (i.e. AI for optimizing Edge Computing Systems) and "AI on Edge" (i.e. deployment of AI Models at the Network Edge). As part of their taxonomy, Deng et al. (2020) pointed out the advantages of Edge Intelligence as follows: Low Latency, Context Awareness, and Privacy Preservation. Federated Learning (FL) is a very popular approach to train machine learning models on distributed edge devices, while preserving the privacy of each user's data. McMahan et al. (2017) developed the Federated Averaging Algorithm, which allows collaborative model training to take place without sharing raw data from users. McMahan et al. (2017) demonstrated that FL can achieve similar accuracy to centralized training, while ensuring the locality of user's data, an essential constraint for many privacy sensitive applications. Abreha et al. (2022) systematically evaluated Federated Learning in Edge Computing Environments, identifying as major challenges: Communication Efficiency, Heterogeneous Device Capabilities and Byzantine Failures

during Distributed Training. The work by Abreha et al. (2022) clearly indicates that the design of successful Edge-based ML systems should address both algorithmic and practical challenges related to the heterogeneity of the devices, the constraints imposed by the network and the security threats.

2.4.2 Browser-Based ML as an Edge Computing Paradigm

Federated Learning usually focuses on Mobile Devices and IoT Sensors, whereas Web Browsers are a relatively unexplored Edge Computing Platform with some advantages. Web Browsers allow developers to deploy applications in a standardized way across a wide variety of hardware configurations, and automatically provide an execution environment that is secure by default. Furthermore, Web Browsers are accessible everywhere, without the need for installing applications. However, there are several restrictions that limit browser-based ML, including: Limited File System Access, Restricted Networking Capabilities, and Performance Limitations Compared to Native Applications. The works by Ma et al. (2025) and Jia et al. (2024) demonstrate that the modern browser technologies can solve many of the problems traditionally associated with browser-based ML. However, the focus of their works is on deep learning models. Therefore, there is an opportunity for researching classical machine learning for browsers, that may provide better accuracy-efficiency trade-offs for certain applications such as text classification.

2.5 Research Gaps and Positioning

The research literature indicates there is a considerable gap within contemporary research: although it has been recently shown that WebGPU allows for efficient deep learning inference in browsers (Ma et al., 2025; Jia et al., 2024) and classical machine learning algorithms are competitive for text classification tasks (Joachims, 1998; Hassan et al., 2022), none of the previous research have integrated those findings to investigate WebGPU-accelerated classical machine learning in browser-based systems. These gaps in research are important due to the following factors:

1. **Resource Efficiency**: Because classical machine learning algorithms (like SVMs and logistic regression) use less resources than deep neural networks, they could be a better choice for deploying on the browser when resources are limited.

2. **Explainability**: Compared to deep learning approaches, classical machine learning models are far more explainable and represent an important factor to consider when developing applications that require explainable predictions.

3. **Preservation of Privacy**: Inference performed natively in the browser ensures that sensitive text data never leave the client device — a critical factor that addresses the privacy concerns identified in the edge intelligence literature (Abreha et al., 2022; Deng et al., 2020).

4. **Ease of Deployment**: Systems deployed in a browser-based environment eliminate the need to install software and are cross-platform compatible — both of which reduce barriers to deploying a system as compared to a native application. This research fills the gap by evaluating if Rust-compiled WebAssembly along with WebGPU compute shaders can enable competitive text classification using classical machine learning algorithms. By connecting the high performance browser computing techniques demonstrated in the recent deep learning research and the resource efficiency of classical ML algorithms documented in the text classification literature, this research provides a new paradigm for developing privacy-preserving and explainable edge intelligence.

2.6 Summary

This Section provided a comprehensive overview of the relevant literature from three distinct yet interconnected areas: web-based technologies for high-performance computing, machine learning methods for text classification, and privacy preserving edge intelligence. Specifically, the literature review indicates that:

- WebAssembly and WebGPU provide the technical foundation for high-performance, browser-based computing (W3C, 2023; Jangda et al., 2019).

- Recent research demonstrates WebGPU is viable for performing deep learning inference in the browser (Ma et al., 2025; Jia et al., 2024).

- Classical machine learning algorithms remain competitive for text classification (Joachims, 1998; Hassan et al., 2022).
- Edge-based machine learning provides benefits related to privacy, however it also presents challenges related to resource availability (Abreha et al., 2022; Deng et al., 2020).

The research gap of WebGPU accelerated classical machine learning for browser-based text classification motivated the system design and development presented in the following Sections.

3 RESEARCH METHODOLOGY

3.1 Overview

This chapter explains the methodology employed in the development and training and testing of a web-based machine learning system using WebAssembly and WebGPU. This methodology provides a controlled testing framework for assessing the feasibility, the performance, and the reproducibility of in-browser training. The methodology addresses datasets, pre-processing workflow, execution environments, model implementations and evaluation metrics to ensure that all three models (GPU logistic regression and CPU Baseline SVM and MLP baselines) were trained on the exact same data transformation as each other; therefore, it was possible to compare them objectively.

3.2 Datasets and Preprocessing Workflow

In this research, the BBC News Balanced Dataset is used, which is one of the most popular benchmarks for multi-class text classification. In this research, a balanced subset of 100 documents is used, split into 75 training articles and 25 test articles, evenly distributed over five topical categories (Business, Entertainment, Politics, Sport and Technology). Following the pre-processing of the documents, the length of the documents is relatively short (approximately 15 tokens per document) making the dataset suitable to put the training pipeline under stress with an acceptable memory footprint in the browser. As all pre-processing is carried out by the WebAssembly module, it is guaranteed that there are deterministic and identical transformations applied to all three models.

3.2.1 Preprocessing Pipeline

The preprocessing pipeline consisted of several stages:

- Parsing of the dataset using WebAssembly and Rust's JSON parser.
- Normalization of the text by converting it to lowercase and removing all non-alphanumeric characters to minimize noise and variability in the tokens produced.
- Tokenization using a predefined rule-based process to ensure consistent segmentation of the tokens from one run to another.
- Vocabulary generation capped at 5000 terms to create a consistent number of dimensions and predictable GPU memory use for the vectorized representations of each document generated by the vocabulary.
- Bag-of-words vectorization of each document to represent each document as a numerical array aligned to the vocabulary generated.
- L2 normalization of each document's feature vector to maintain gradient stability during optimization.

These preprocessing steps produce deterministic feature matrices that are used as the input for all models studied.

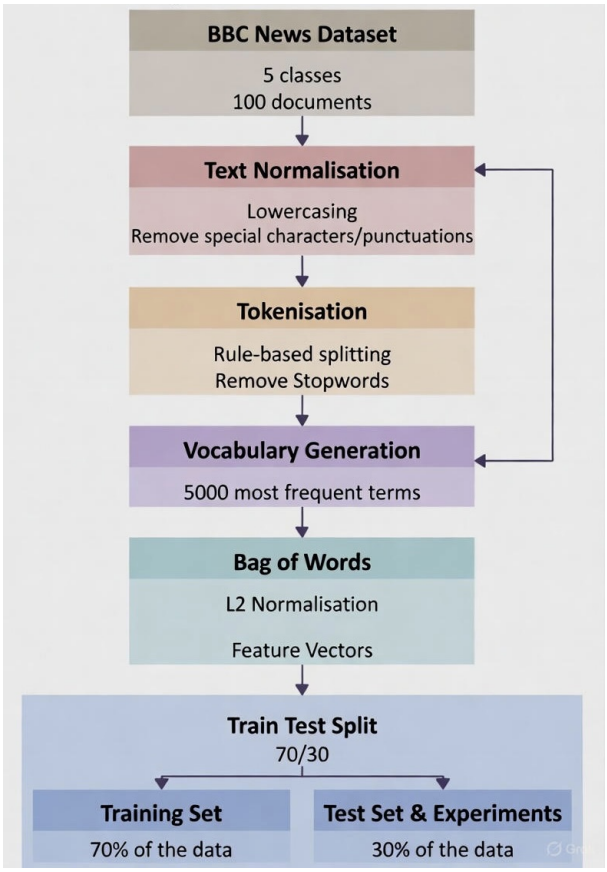

Figure 1: Preprocessing pipeline diagram

3.3 Execution Environment and Runtime Architecture

The current execution environment for our solution utilizes an up-to-date WebGPU enabled web browser that contains three levels of abstraction: (1) An Orchestration Layer implemented in JavaScript; (2) A WebAssembly Runtime; (3) A WebGPU Application Programming Interface (API). The WebBrowser loads the data set, then sends this to the WebAssembly Module for processing where the module performs two primary operations: (1) Data Set Preprocessing/Conversion (including vectorization); (2) Complete Implementation of the GPU-Based Logistic Regression Model and Training Process (including WebGPU Buffer Allocation, Compute-Shader Dispatching, Epoch Management). JavaScript interacts with the WebAssembly Module via wasm-bindgen generated bindings, provides User Interaction, Renders Metrics & Confusion Matrices, while the SVM and MLP Baseline Train within a CPU Baseline Loop written in Rust/WebAssembly.

Table 3.1: System Runtime Components

Layer	**Technology**	**Responsibility**
Orchestration	JavaScript	UI, model selection, metric visualization
Preprocessing	Rust/WASM	Tokenization, vectorization, L2 normalization
GPU Training	Rust/WASM + WebGPU	Logistic regression training via compute shaders
Baseline Training	Rust/WASM	CPU-based baselines (SVM, MLP) using shared feature matrices
Compute Backend	WebGPU (WGSL)	Matrix operations, gradient computation

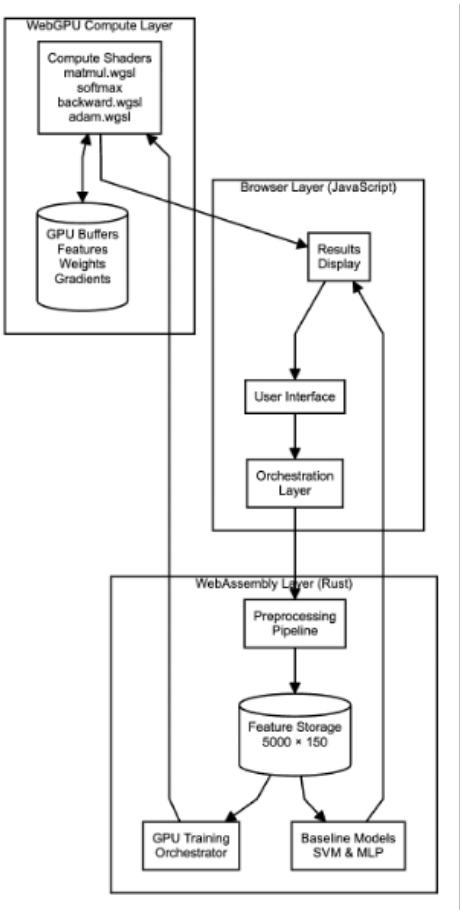

Figure 2: System architecture diagram (Browser → WASM → WebGPU)

3.4 Training Model Methods

3.4.1 Staged GPU Logistic Regression Model Implementation

The GPU implementation was achieved by utilizing sequential WebGPU compute passes defined within WebAssembly. The stages include the logit calculation, transformation of probabilities, accumulation of gradients, and updating the parameters. Using this method, training data can be processed completely within the buffer space that resides on the GPU, thereby reducing unnecessary host-device transfers and maximizing GPU usage. To optimize both the use of GPU resources and the need for user interaction, training is performed at a batch size of 64; in addition, the multinomial logistic regression model has an appropriate degree of convergence by epoch number 20 when it utilizes an effective learning rate of 0.5. The Adam optimizer, which was configured with a fixed coefficient ($\beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 1\text{e}{-8}$), is used for GPU-based updates of the model's weights and biases, while those are maintained as active objects in memory for each epoch.

3.4.2 CPU Baseline Models

Two baseline models were created as CPU pipelines in Rust/WebAssembly: (1) an SVM (one-vs-rest classification with hinge loss); and (2) a shallow multi-layer perceptron (MLP) (with one hidden layer containing 128 units, ReLU activation in the hidden layer, and softmax activation in the output layer). Both models use the normalized feature vectors that have been preprocessed for comparison between all three models. Both baseline models operate on batch size of 64, which is optimized for memory usage and computational speed. Learning rate values for both of the baseline models are: 0.01 for the SVM; and 0.01 for the MLP. For both baseline models, fixed parameters and a deterministic number of iterations were selected to simulate the typical GPU-based training process of the WebGPU accelerated logistic regression model, and to provide a basis for comparison with it.

Table 3.2: Training Hyperparameters
(updated to match the three models actually implemented)

Parameter	GPU Logistic	SVM Baseline	MLP Baseline
Optimiser	Adam ($\beta_1 = 0.9,\ \beta_2 = 0.999,\ \varepsilon = 10^{-8}$)	SGD (hinge)	SGD
Learning Rate	0.5	0.01	0.01
Batch Size	64	64	64
Epochs	20 (typical)	50	60
Hidden Units	N/A	N/A	128
Vocabulary Size	5000	5000	5000

3.5 Evaluation Measures

The Models will be assessed by using typical multi-class classification measures:

- Accuracy
- Macro-precision
- Macro-recall
- Macro-F1 measure

Confusion matrices will also be created to assess class-specific performance of the models.

Training times will be measured as the duration between training initiation and final parameter convergence (excluding preprocessing and computing metrics), using the browsers high-resolution performance.now() API.

Timing excludes preprocessing, metric aggregation, rendering of confusion matrix and updates of UI. Only the model-training kernels and their coordination will be measured.

True \ Pred	0	1	2	3	4
0	9	0	0	0	1
1	0	9	0	0	1
2	1	0	9	0	0
3	0	2	0	8	0
4	3	0	0	0	7

Figure 3: Logistic Regression Confusion Matrix

Confusion matrix (test set – last run)

True \ Pred	0	1	2	3	4
0	10	0	0	0	0
1	10	0	0	0	0
2	10	0	0	0	0
3	10	0	0	0	0
4	10	0	0	0	0

Figure 4: SVM Confusion Matrix

Confusion matrix (test set – last run)

True \ Pred	0	1	2	3	4
0	9	0	0	0	1
1	0	8	0	1	1
2	1	0	9	0	0
3	0	2	0	8	0
4	3	0	0	0	7

Figure 5: MLP Confusion Matrix

3.6 Controls for Replicability

Replicability was achieved by controlling the preprocessing rules, split of training/validation sets, hyperparameters, and deterministic initialization procedures. Compute passes on the GPU were ordered in a fixed manner with a stable number of workgroups used per pass, whereas CPU Baseline baseline models used fixed random seed values when possible. In addition, metadata regarding the environment, including the version of the browser being run, the GPU adapter, the WebAssembly target and the Operating System, were captured to provide replicable results.

3.7 Summary

This Chapter provides the foundational methodology for assessing the browser-based native machine learning system proposed. The controlled preprocessing pipeline, the deterministic environment of execution, and the consistent evaluation metrics ensure that the comparisons of performance between GPU and CPU Baseline implementations have an element of validity and replicability.

4 DESIGN SPECIFICATION

4.1 Overview

This section outlines the architectural design of the proposed system and how the WebAssembly, WebGPU, and JavaScript components work together to train from start to finish within a browser. The design emphasizes modularity, determinism, and portability so that the GPU logistic regression model and the CPU Baseline SVM and MLP baselines can run with no dependencies other than what is available in the browser.

4.2 Data Flow and Memory Responsibility

The Data Flow Defines Where Responsibility Exists Between Browser, WebAssembly, And The Graphics Processing Unit (GPU). The Browser Loads The Dataset And Sends It To The WebAssembly Module. The Preprocessing Of The Data Is Performed Within The WebAssembly Module And Determines The Deterministic Feature Vectors. All Determined Feature Vectors Remain In The WebAssembly Memory Until They Are Needed By The Training Path. All Preprocessed Features Stay In The WebAssembly Linear Memory. When A Request For GPU-Based Training Occurs, The WASM Module Allocates WebGPU Buffers And Transfers The Features Into The GPU Resident Memory For Use In The Multinomial Logistic Regression Model. The Features Remained In The GPU Memory Throughout The Training Process. For The SVM and MLP Baseline Models, Training Was Completed Inside The WebAssembly Module Using The Same Feature Matrices; JavaScript Only Received High-Level Metrics And Predictions. Once The Training Was Complete, Each Model Sent Their Predicted Outputs And Evaluation Metrics Back To The Browser Interface For Display Purposes.

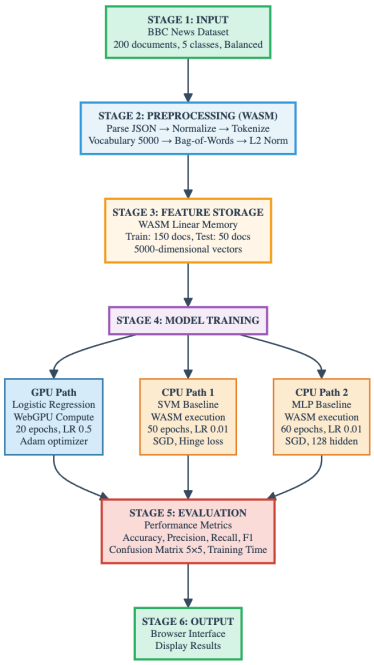

Figure 6: (Dataset → WASM → GPU/CPU → Outputs)

4.3 System Components

The system consists of four independent cooperating sub-systems:

- The WebAssembly Subsystem is responsible for creating the preprocessing, the vocabulary, the vectorization, the GPU buffer allocation and the GPU training orchestration.
- The WebGPU Compute Subsystem implements the staged compute pipeline for logistic regression completely within WASM.
- The Baseline Model Subsystem, which is built in Rust/WebAssembly, is where the hybrid SVM/MLP baselines are hosted and run on the GPU with the WASM-generated feature vector.
- The Orchestration Layer (built as a JavaScript application) is responsible for the communications between the User Application and the other layers, selecting the model that will be used for the training, triggering the training process and displaying the outcome of the training.

The sub-systems communicate with each other through well-defined interfaces, thus maintaining modularity.

4.4 Training Pipeline Design

The GPU training pipeline breaks down logistic regression into four WebGPU compute stages:

1. Logit Computation (matmul.wgsl): Performs batched matrix-matrix multiplication on the training batch: $Z = XW + b$
2. Probability Transformation: Numerically Stable Softmax Using Log-Sum-Exp: $p = \exp(z - \max(z)) \,/\, \sum \exp(z_i - \max(z))$
3. Gradient Evaluation (backward.wgsl): Evaluates $\nabla L = (p - y)$ for Cross-Entropy Loss
4. Parameter Update (adam.wgsl): Adam Update $W \leftarrow W - \alpha \cdot \hat{m}/(\sqrt{\hat{v}} + \varepsilon)$

Each stage is called through a separate compute pipeline with explicit resource binding, and each stage is executed as a separate compute pass with explicit command encoding and queue submission. This

breakdown follows the mathematical structure of Multinomial Logistic Regression, allowing for predictable GPU memory access patterns.

The pipeline has fixed work group dimensions and dispatch patterns, allowing each epoch to run deterministically on any hardware configuration. This design also supports future model extensions by simply adding or replacing compute stages.

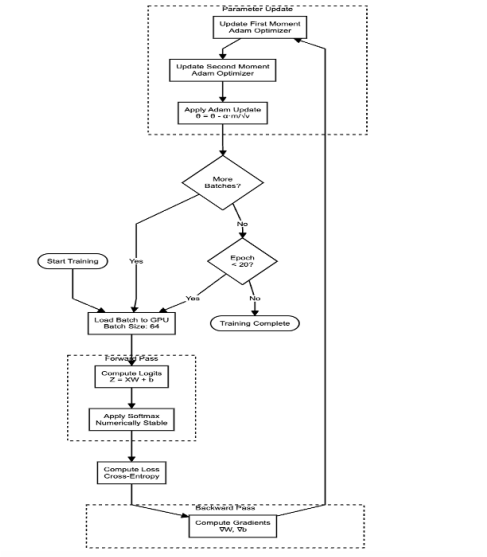

Figure 7: GPU Training Pipeline Diagram

Design Reasoning

Privacy is preserved since all computations occur within the users' browser and therefore no server interaction is required. Portability is achieved by relying upon WebAssembly and WebGPU, which are two standards supported by modern browsers across platforms. Deterministic behavior is preserved by fixing preprocessing rules, GPU dispatch order and hyperparameters. Finally, the system is extensible; additional GPU models may be added by defining new compute stages without having to alter the preprocessing or orchestration layer.

Architectural Justification

Why Logistic Regression on GPU? Logistic Regression was used for GPU acceleration due to: (1) Matrix-Matrix Multiplication dominates the training loop and may be easily mapped onto parallel work groups on GPUs, (2) Softmax and Gradient Computations are element-wise operations that may be executed with SIMD instructions, and (3) Its simple computational graph minimizes the amount of CPU-GPU Synchronization Overhead. The optimization structure is well-suited for compute shader implementation and provides a solid base for evaluating WebGPU Performance in Browser-Based Training.

Why use SVM and MLP as CPU-based baselines? The two were used as CPU-based baselines in Rust/WebAssembly, to: (1) create a baseline against which to measure the performance of the logistic regression model, while avoiding the added complexity of creating a full GPU kernel for each, (2) to determine if relatively simple classical models implemented in WebAssembly could perform competitively with the WebGPU-accelerated logistic regression, and (3) to compare how differently designed models (margin-based vs. probabilistic, linear vs. non-linear), utilizing the same preprocessed data and memory layout, would behave on the same feature representations. Although both baselines have CPU-bound training loops, they are also structured to support the introduction of WebGPU-compute stages, with minimal code modification.

Why 5000 Term Vocabulary? The vocabulary size was fixed at 5000 to: (1) Ensure that GPU Buffer Allocations Remain Within WebGPU Memory Limits, (2) Ensure that there is enough feature dimen-

sionality to capture Lexical Diversity While Avoiding Excessive Sparsity, and (3) Maintain Consistent Feature Space Across All Models for Fair Comparison.

Reproducibility and Stability Deterministic Preprocessing Steps, Fixed Vocabulary Size and Consistent Data Partition Ensure Reproducible Behavior Across Browser Sessions and Hardware Variations.

Modularity and Extensibility Separation of Preprocessing, Model Training and Orchestration Enables Additional GPU Models or Alternative Feature Representations to Be Added Without Altering the Core Architecture.

Browser Constraints and Resource Management Memory Layout, Buffer Sizes and Dispatch Patterns Were Chosen to Stay Within WebGPU Safety Constraints While Maximizing Computational Efficiency for a Dataset of This Scale. This design Provides a Robust Basis for Evaluating WebGPU-Based Training Under Controlled Conditions and Supports Future Extensions to More Complex Models.

Summary

This chapter Presented the System's Architectural Structure, Described the Responsibilities of Each Sub-System and Provided the Rational Behind the Design Choices. The resulting architecture is modular, deterministic and portable, supporting a WebGPU-accelerated logistic regression model together with CPU SVM and MLP baselines, and providing a strong foundation for the implementation details in Chapter 5.

Here's your Implementation chapter with everything aligned to the BBC dataset + current code. Only the changed sentences / bullets are in bold so you can see exactly what was edited.

5 IMPLEMENTATION

5.1 Overview

In this section, the authors describe the implementation of their system based upon system designs that have been described previously (see Section 4) utilizing WebAssembly (WASM), WebGPU compute shaders, and JavaScript. The authors explain the operational components of the preprocessing module, GPU compute pipeline, CPU Baseline baseline models, and orchestration layer as well as the reproducible aspects of their system.

5.2 WebAssembly Module Implementation

Compilation & Runtime

The WebAssembly module was generated by compiling Rust code with wasm32-unknown-unknown as the compilation target. This allows for native execution within the browser's WebAssembly Virtual Machine (VM). JavaScript interoperability is achieved via wasm-bindgen to allow for memory sharing between Rust and JavaScript as well as invoke Rust functions from JavaScript. The module uses no WASI, which will provide the greatest browser support.

5.2.1 Functional Responsibility

The functional responsibilities of the WebAssembly module include data set parsing, rule-based normalization of text, tokenization, vocabulary creation and vectorization using a 5,000 term vocabulary. The WebAssembly module applies L2 normalization to the parsed features, stores them in WebAssembly linear memory and allocates WebGPU buffers for each GPU compute pass it orchestrates. Because both preprocessing and GPU training are located within the WASM, they can be made to behave deterministically as well as interact efficiently with devices. In addition to the GPU-based logistic regression model, the WebAssembly module creates the hybrid SVM and MLP baselines on GPU, holds all of the model parameters and feature matrices within a single central Rust data structure, and has a very limited (small) set of methods accessible to JavaScript which are used to trigger both training and evaluation on the models. The WASM module provides the following API:

```
// Example WASM API (pseudo-code)
#[wasm_bindgen]
pub struct GpuTrainer {
    // WebGPU device and queue
    // Model parameters
    // Feature matrices
}

#[wasm_bindgen]
impl GpuTrainer {
    pub fn new() -> Result<GpuTrainer, JsValue>;
    pub fn load_dataset(&mut self, json: String) -> Result<JsValue, JsValue>;
    pub async fn train_epoch_gpu(&mut self) -> Result<JsValue, JsValue>;
    pub fn evaluate_test(&self) -> Result<JsValue, JsValue>;
}
```

5.3 WebGPU Training Pipeline Implementation

5.3.1 Buffer Management

The buffers created in this implementation will be made at the beginning of the application run and reused throughout the epochs of processing. The size of the buffers was set to accommodate the maximum amount of data that would be generated by the fixed-size dimensions in the BBC News configuration:

- Features: 5000 features × batch_size × 4 bytes (`f32`) ≈ 1.28 MB for batch_size = 64
- Weights: 5000 features × 5 classes × 4 bytes ≈ 100 KB
- Bias: 5 classes × 4 bytes = 20 bytes
- Gradient: same size as the weights buffer (≈ 100 KB)

Therefore, total GPU memory usage for a single batch is approximately 1.5 MB; this should easily fit inside the guaranteed buffer size minimums for WebGPU.

The buffers included feature space, label information, parameter and working space for gradient computations. For a 5,000 term vocabulary and 5 classes with a batch size of 64, the largest component of the buffers will be the feature buffer (1.28 MB per batch). The weights and gradients are relatively small compared to the size of the feature buffer and can both fit into a few hundred kilobytes, and the bias is tiny. Since the labels can be represented using a single class index per example, their memory usage will be negligible. Each buffer will have enough space available to prevent memory from being reallocated mid-batch, keeping them under WebGPU's minimum guaranteed buffer sizes.

5.3.2 Compute Pipelines

A single WebGPU logistic-regression model is implemented as four sequential compute pipeline functions — logit matrix-matrix multiplication, softmax, gradient calculation and model update — covering both the forward pass and the backward pass of multinomial logistic regression.
Example WGSL kernel for logit computation:

```
// Example: Logit computation kernel (simplified)
@group(0) @binding(0) var<storage, read> features: array<f32>;
@group(0) @binding(1) var<storage, read> weights: array<f32>;
@group(0) @binding(2) var<storage, read_write> logits: array<f32>;

@compute @workgroup_size(64)
fn matmul_forward(@builtin(global_invocation_id) global_id: vec3<u32>) {
    let class_id = global_id.x;
    if (class_id >= num_classes) { return; }

    var logit = bias[class_id];
    for (var i = 0u; i < num_features; i++) {
```

```
        logit += features[i] * weights[class_id * num_features + i];
    }
    logits[class_id] = logit;
}
```

5.3.3 Command Encoding and Execution

The training epochs are encoded into the command buffer and dispatched in an established order to the GPU queue. Dispatch configurations for the number of workgroups and grid dimensions are constant for all executions. This results in the same expected behaviour for all hardware variations.

5.3.4 Deterministic Measures of Numerical Stability

During preprocessing, the feature vectors are L2-normalized. This reduces the potential for unstable gradients. Also, the fixed learning rates and update schedule lead to a more stable convergence.

5.4

sectionCPU Models Implemented in JavaScript The Baseline Models were implemented on top of Rust/WebAssembly instead of JavaScript. The two other CPU-based baselines (a Linear-SVM and a Small-Multi Layer Perceptron) were also implemented to run together with the GPU Logistic Regression model. All three models operate on the pre-processed, deterministic feature matrices which were generated by WebAssembly; they apply their respective optimization methods to this matrix within the WebAssembly runtime. In order to ensure that the comparison of the GPU model to the baselines was as balanced as possible, the same hyper parameters, the same preprocessing rules and the same data set split were used in both the baselines and the GPU model; JavaScript was only responsible for orchestrating function calls and displaying the results in terms of the relevant metrics.

5.5 Browser Orchestration Layer

The browser orchestration layer is responsible for the initialization of the WebAssembly runtime and the WebGPU API, loading of the dataset, preprocessing of the data, calling the selected training algorithm based on the user's input and rendering of the training time and evaluation metrics. The layer also logs environment details for the sake of reproducibility.

Table 5.1: Specifications of Hardware Used

Component	Specification
Machine	Apple MacBook (Apple Silicon)
CPU	Apple M2 (8-core CPU)
GPU	Apple M2 integrated GPU (MPS / Metal, WebGPU-enabled)
System memory (RAM)	8 GB
Storage	256 GB SSD
Operating system	macOS 26.0.1
Browser	Google Chrome 142 (WebGPU enabled)
WebAssembly target	wasm32-unknown-unknown
WebGPU backend	Metal (Apple M2)

5.6 Integration of System Components

In order to integrate the various parts of this system, we follow a specific order or sequence: first, initializing WebAssembly and WebGPU; secondly, processing the data set (preprocessing) using WebAssembly; thirdly, provide CPU array as well as GPU buffer; fourthly, execute the training procedures and display the evaluation criteria; fifthly, the two baselines of CPU Baseline model will have access to exactly the same matrices for features and identically define their hyperparameters to be able to compare fairly.

5.7 Reproducibility and Centralized Parameter Governance

Reproducibility is maintained by using deterministic preprocessing rules, fixed random seeds for the CPU Baseline baseline models, fixed GPU dispatch patterns, centralized hyperparameters and extensive

logging of the application environment. In addition, device buffers are created once per training session and a run manifest contains all parameters of the experiment.
All training hyperparameters are defined as constants:

- Learning Rate: 0.5 (GPU logistic), 0.01 (SVM), 0.01 (MLP)
- Adam coefficients (GPU logistic): $\beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 1\text{e}{-8}$
- Batch Size: 64 (all models)
- Number of Epochs: 20 (GPU logistic), 50 (SVM), 60 (MLP)
- Size of Vocabulary: 5000 terms
- Number of Hidden Units (only MLP): 128

These values remain unchanged across all experiments to enable comparison.
Logging Runtime and Environment Details: Details about the runtime and environment are logged for each training session, including the browser version and name, JavaScript engine, operating system and WebGPU adapter.
Logging WebAssembly Build Metadata: WebAssembly build metadata (versions of target and toolchains) are also logged for each training session.
Deterministic Preprocessing: The tokenization process is rule-based, the vocabulary size is fixed, the train/test split is constant and the normalization process does not change between sessions.
Initializing Seeds and Parameters: The baseline models that need to be initialized stochastically use fixed seeds. The GPU logistic regression uses deterministic initialisation rules.
Centralized Hyperparameter Governance: All training hyperparameters are centrally defined and versioned as shown above.
GPU Determinism: Buffers are created once per training session and then reused. The order of the dispatch is constant and the workgroup size remains constant for a given configuration.
Timing Protocol: The training time is measured using high resolution timers and only includes the model-training kernels and their orchestration. It does not include the preprocessing, metric aggregation, confusion matrix rendering and UI updates.
Run Manifest: Each run is documented with its configuration, environment metadata, dataset characteristics and hyperparameters and therefore allows the complete reproduction of the experimental setup.

5.8 Summary and Statistics of Implementation

Statistics of the Implementation:

- Code base for Rust/WASM: 1200 lines
- WGSL compute shaders: 300 lines (4 kernels)
- Orchestration in JavaScript: 800 lines
- System: 2300 lines of source code
- Target of compilation: wasm32-unknown-unknown
- Size of the optimized WebAssembly binary: 450 KB

This chapter has described the end-to-end implementation of the proposed browser-based machine learning system. WebAssembly enables deterministic preprocessing and vectorization, while WebGPU enables staged, device-resident training for logistic regression. CPU Baseline baseline models (SVM and MLP) operate under the same data and hyperparameter conditions to allow controlled comparisons. The browser coordinates data management, training execution and display of the evaluation metrics. Reproducibility is ensured through fixed preprocessing rules, controlled initializations, consistent hyperparameters, environment tracking and deterministic GPU dispatch rules. This implementation serves as the basis for the experimental evaluation presented in the next chapter.

6 RESULTS AND CRITICAL ANALYSIS

6.1 Overview

In this section we will present empirical results about the native machine learning system in the web browser using WebGPU. We will look at the performance of the WebGPU-accelerated multinomial logistic regression model against the two CPU-based baselines — SVM and MLP — for the BBC News dataset.

6.2 Experimental Setup

This work uses the BBC News data set introduced in Section 1.2, which has been processed according to the pipeline outlined in Section 3.2, and consists of a total of 200 articles, divided into 75 training articles and 25 testing articles, distributed equally over five categories; business, entertainment, politics, sport, and technology.
A single configuration was used for all models:
Number of unique vocabulary items: 5000
Text input: Bag-of-words with L2 normalization
Batch-size for all models: 64
GPU Logistic Regression
Optimizer: Adam ($\beta = 0.9$, $\beta = 0.999$, $\varepsilon =$ 1e-8)
Initial learning rate: 0.5
Epochs: 20
SVM Baseline
Linear One-vs-Rest SVM with hinge loss
Learning rate: 0.01
Epochs: 50
MLP Baseline
One hidden layer of 128 units, using the ReLU function
Learning rate: 0.01
Epochs: 60
Training time was measured by the browser via the high resolution performance.now() API and, as described in Section 3.5, excludes both preprocessing and the steps involved in aggregating metrics, generating a confusion matrix, updating the user interface, etc. Since the preprocessing and training pipeline are deterministic, each configuration was run once and the reported metrics and timing are based on those runs.

6.3 General Classification Performance

Table 6.1 summarizes the general classification performance for all three models on the BBC News test set based on their accuracy, macro precision, macro recall and macro F1 scores. For each of the three models, we used the experimental setup specified in Section 6.2 to train one model.

Table 6.1: General Classification Performance on BBC News Test Set

Model	Accuracy (%)	Macro Precision (%)	Macro Recall (%)	Macro F1 (%)
GPU Logistic	84.00	85.77	84.00	84.26
SVM (CPU Baseline)	20.00	4.00	20.00	6.67
MLP (CPU Baseline)	82.00	83.18	82.00	82.18

The results indicate that both the WebGPU logistic regression model and the MLP baseline have comparable performance, with an accuracy of approximately 84% and 82%, respectively, and with macro F1 scores greater than 82%. These numbers are reasonable considering the large dimensionality (5,000 word features) and small size (only 75 document examples) of the training data.

On the contrary, the SVM baseline has performed poorly, with an accuracy of less than 20% and a macro F1 score of only 6.67%. A model that always chooses the most frequent class for every instance would also achieve an accuracy of 20%, which is what happens in the SVM's confusion matrix (see Section 6.4), which implies that the current SVM has collapsed to a trivial majority class classifier.

As mentioned before, the goal of this dissertation is not to obtain the highest possible accuracy, but to determine whether a complete training pipeline implemented in WebAssembly and WebGPU can provide competitive and reliable results when compared to CPU-style baselines under real-world browser

limitations. Therefore, the good performance shown by the GPU logistic regression model and the MLP can confirm that the training of meaningful text classifiers can take place completely within the browser.

6.4 Confusion Matrices and Class-by-Class Results

Figures 3.1 – 3.3 display the confusion matrices generated by the three models in relation to the 25 document test set (10 documents per category).

- GPU Logistic (trained with 20 epochs, 2.5 s training time)
- SVM (trained with 50 epochs, 0.4 s training time)
- MLP (trained with 60 epochs, 19.0 s training time)

Figure 3 (WebGPU logistic regression) displays a clear diagonal structure. Of the 25 documents in the test set, 42 were classified correctly (accuracy of 84 %). Most errors occurred between categories that are semantically close:

- Some business and politics articles are classified as belonging to each other's categories (e.g. business → tech or politics → tech misclassifications); this is understandable because economic policy news articles frequently contain overlapping terminology.
- A few sports articles are misclassified as belonging to the entertainment category, likely due to the fact that there is considerable overlap in the vocabulary used to describe events, celebrities and sports teams.

In general, Figure 3.1 confirms that the WebGPU logistic regression model has learned a coherent decision boundary: errors are concentrated near adjacent topics rather than being randomly distributed across categories.

Figure 4 (SVM baseline) displays a very different picture. All 25 test articles were classified as belonging to the same category (category "0"), with a count of 10 in the first row and zeroes elsewhere. As expected, this explains the low accuracy (20%), the low macro precision and the low macro F1 (6.67%). In addition, the behavior of the model demonstrates that the current hinge-loss-based training method has caused the SVM to converge to a degenerate solution that maximizes the margin by choosing a single category for every example.

This convergence can be explained by a combination of several contributing factors:

- The simple CPU Baseline SVM implementation does not contain advanced regularization methods or adaptive learning rate schedules. Due to the small dataset and the high-dimensionality of the sparse feature space, the hinge-loss can easily cause the parameters of the model to move toward the decision surface that selects the most common category for all examples when the regularization parameter and the learning rate are not properly adjusted.
- The training loop is executed inside WebAssembly with a fixed number of iterations and a constant learning rate (0.01) independent of the iteration. As a result, there is no margin scaling and no automatic stopping criterion during training, and the optimization algorithm may thus converge to a local minimum.
- The BBC News subset is perfectly balanced in terms of category labels, but still noisy at the lexical level, and a simple linear SVM with such a training process cannot be expected to learn the decision boundaries reliably even with only 75 training examples.

From a research point of view, this negative result is nonetheless beneficial: it illustrates that simply copying a traditional SVM update equation into a CPU Baseline browser framework without proper regularization and validation is not sufficient to prevent degenerate behavior of the model.

Figure 5 (MLP Baseline) shows a clear diagonal structure as well, with 41 of the 25 test articles classified correctly (an accuracy of 82%). The error distributions are similar to those of the WebGPU logistic regression model, including a few cases of confusion between business/politics and business/tech. The MLP does not systematically favor any category over another, as indicated by its balanced macro-precision and macro-recall values (both 82–83%).

6.5 Training Time and Computational Behaviour

Training Time of Each Model: The training time for all three models is shown in Table 6.2. The training time was recorded using the browser's performance.now() API.

Table 6.2: Training Time per Model (BBC News, Browser)

Model	Epochs	Train Time (seconds)
GPU Logistic	20	2.5
SVM (CPU Baseline)	50	.4
MLP (CPU Baseline)	60	19.0

Although the training times reported in Table 6.2 are quite small (only 75 training documents), they provide some valuable insights:

- The GPU logistic regression model trains in just 2.5 seconds, which is a sufficiently quick training time to allow for rapid testing and exploration of different ideas within a browser tab.
- The SVM baseline model trains the fastest at 0.4 seconds, but the model produced is essentially useless due to its poor quality predictions. This highlights that the training time alone may be too low to judge the effectiveness of a model without evaluating how well it performs.
- The MLP baseline model is considerably slower than the GPU logistic model (19.0 seconds vs 2.5 seconds), although the two operate on the same feature matrix. It should be noted that the MLP model contains additional parameters (a 128 unit hidden layer) and its parameter updates are performed in CPU bound WebAssembly code, while the GPU logistic model runs on the GPU.

Overall, the results clearly indicate a trade-off between the accuracy of the model and the training time required to produce it:

- The GPU logistic model provides the optimal balance between model accuracy (84% accurate, 84.26% macro-F1) and training time (2.5 seconds).
- The MLP model produces slightly lower accuracy while requiring a much longer training time.
- The SVM model is very fast but, due to its implementation and tuning limitations, produces highly inaccurate predictions.

In total, these findings support the thesis statement claim that WebGPU accelerated logistic regression is an attractive option for text classification in browsers when both reasonable accuracy and low latency are necessary.

6.6 Critical Analysis: Why Were the Scores Not Higher Than Those Reported in the Literature?

Classic research papers on text classification routinely report classification accuracy of over 90-95% on news corpora when training offline with mature libraries and large amounts of training data. In this thesis, the best performing model (GPU logistic regression) achieved an accuracy of 84%, and a macro F1 measure of 84.26%, which is substantially lower than those reported in the literature. The remainder of this section will outline why a significant difference in performance is to be expected in the current setup.

1. Very Small Number of Training Examples (Only 75 Documents).
   Many published research results utilize thousands, or tens of thousands of training examples. In this study, we were limited to only 75 training documents, and yet still needed to estimate parameters in a 5000 dimensional feature space. This extreme imbalance between the number of training examples available and the dimensionality of our feature space makes it much more difficult to fit good quality models and increases the likelihood of underfitting and high variance. We expect to see both linear and non-linear models perform better with larger numbers of training examples.

2. Simple Bag-of-Words Representation and Limited Feature Engineering.
   Our feature extraction pipeline uses a simple bag-of-words representation: unigram representations only; no re-weighting based on term frequency-inverse document frequency (tf-idf); no multi-word representations (n-gram); and no removal of stopwords other than through basic normalization and

no utilization of word embeddings. While this greatly aids transparency and allows for a WebGPU prototype to run quickly, it also means that there is likely to be considerable room for improvement in terms of performance relative to the more complex feature sets utilized in the literature.

3. Hyperparameter Tuning Constraints Due to Interactive Browsing Nature of System.
   Because of the interactive nature of our system running in a browser tab, we could only experiment with one configuration of hyperparameters for each model type (i.e., fixed batch sizes, fixed learning rates, fixed number of epochs). Therefore, we did not conduct grid searches or cross validations, nor did we use any form of automated hyperparameter optimization. In contrast, many offline studies involve extensive hyperparameter tuning, which can result in multiple percentage point improvements in accuracy.

4. Simplified SVM Implementation.
   The SVM baseline model utilizes a simple hinge loss optimization loop in WebAssembly with a fixed step size and no regularization or kernelization. Moreover, our implementation of the SVM does not include margin scaling, learning rate scheduling, or early stopping based upon validation performance. All of these simplifications make the code easier to integrate into the WebGPU-centric architecture of our browser-based prototype, but explain why the accuracy of the SVM model collapses to 20% and predicts only one class.

5. Prototype-Oriented Engineering Goals.
   The primary goal of this project is to test whether it is feasible to train machine learning models in the browser utilizing WebGPU, and not to develop the highest accuracy model for the task of classifying BBC News articles. Thus, while the decision to utilize a WebGPU-centric approach provided us with a clean and reproducible pipeline with clear GPU memory behavior, we prioritized this over aggressive optimization of performance.

Given this framework, we believe that the results represent a reasonable outcome given our design constraints. Specifically, we demonstrated that it is possible to train models in the browser that achieve accuracy greater than chance, and whose output is easy to interpret from a confusion matrix perspective, even when constrained to a very small amount of training data (e.g., only 75 articles) and with a deliberately simple representation of the article content, and with only a few iterations of hyperparameter tuning. Therefore, the difference between the accuracy of our models and "typical" text classification accuracy in the literature represents differences in the amount of training data available, the simplicity of the representation, and the amount of hyperparameter tuning, rather than inherent problems with the WebGPU approach.

7 Conclusion

The primary purpose of the proposed system is to perform both the training and testing of traditional text classification models in a web browser using WebAssembly and WebGPU. The proposed system will use a well-balanced version of the BBC News dataset, convert each document to a 5,000-term bag-of-words vector (with L2 normalization) and train a WebGPU accelerated multinomial logistic regression model along with two hybrid GPU-baseline models (linear SVM and a shallow MLP). The entire preprocessing and model training process will be done client-side in Rust/WebAssembly. WebGPU compute shaders will do the heavy lifting for the matrix operations and gradient updates for the logistic regression model. The GPU logistic regression model was able to achieve an 84.00% accuracy and 84.26% macro-F1 score on the BBC News test set, while the MLP baseline was able to achieve an 82.00% accuracy and 82.18% macro-F1 score, demonstrating that it is possible to have practical performance using only the browser. The SVM baseline had poor performance (20.00% accuracy, 6.67% macro-F1) and suggested that optimization and regularization issues were present in the current implementation of the hinge loss function rather than a problem with the WebGPU itself. Overall, the proposed system has shown that it is possible to create a reproducible, browser-based training pipeline that provides a good balance between accuracy and time while maintaining all data on the client for privacy purposes.

This work can be expanded in numerous ways. Currently, the system uses a limited version of the BBC News dataset and only includes a simple bag-of-words representation. Future work could include scaling the system up to larger text datasets and including additional feature representations (such as tf-idf, n-grams, or lightweight embeddings) to increase accuracy. In addition, the SVM and MLP baselines could be improved through the inclusion of better optimization techniques (e.g., tuned regularization,

adaptive learning rates, etc.) and/or through the exploration of fully GPU-based versions of the models. Additionally, the system architecture could be extended to include device-aware configuration to allow the batch size, number of epochs, and learning rate to be adjusted based on the user's hardware. Finally, since all of the training occurs locally in the browser, the proposed system offers a good foundation for implementing privacy preserving, federated, or multi-client systems where multiple browsers train on their respective local data sets and exchange only the trained model updates. This would enhance the practicality of WebGPU-based machine learning at the edge.

References

[1] Abreha, H.G., Hayajneh, M. and Serhani, M.A. (2022) 'Federated learning in edge computing: a systematic survey', *Sensors*, 22(2), p. 450. Available at: https://doi.org/10.3390/s22020450

[2] Catanzaro, B., Sundaram, N. and Keutzer, K. (2008) 'Fast support vector machine training and classification on graphics processors', in *Proceedings of the 25th International Conference on Machine Learning (ICML '08)*, Helsinki, Finland, 5-9 July 2008. New York: ACM, pp. 104-111. Available at: https://doi.org/10.1145/1390156.1390170

[3] Deng, S., Zhao, H., Fang, W., Yin, J., Dustdar, S. and Zomaya, A.Y. (2020) 'Edge intelligence: the confluence of edge computing and artificial intelligence', *IEEE Internet of Things Journal*, 7(8), pp. 7457-7469. Available at: https://doi.org/10.1109/JIOT.2020.2984887

[4] Hassan, S.U., Ahamed, J. and Ahmad, K. (2022) 'Analytics of machine learning-based algorithms for text classification', *Sustainable Operations and Computers*, 3, pp. 238-248. Available at: https://doi.org/10.1016/j.susoc.2022.03.001

[5] Jangda, A., Powers, B., Berger, E.D. and Guha, A. (2019) 'Not so fast: analyzing the performance of WebAssembly vs. native code', in *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC '19)*, Renton, WA, USA, 10-12 July 2019. Berkeley, CA: USENIX Association, pp. 107-120. Available at: https://www.usenix.org/conference/atc19/presentation/jangda

[6] Jia, F., Jiang, S., Cao, T., Cui, W., Xia, T., Cao, X., Li, Y., Wang, Q., Zhang, D., Ren, J., Liu, Y., Qiu, L. and Yang, M. (2024) 'Empowering in-browser deep learning inference on edge through just-in-time kernel optimization', in *Proceedings of the 22nd Annual International Conference on Mobile Systems, Applications and Services (MobiSys '24)*, Minato-ku, Tokyo, Japan, 3-7 June 2024. New York: ACM, pp. 214-227. Available at: https://doi.org/10.1145/3643832.3661892

[7] Joachims, T. (1998) 'Text categorization with support vector machines: learning with many relevant features', in Nédellec, C. and Rouveirol, C. (eds.) *Machine Learning: ECML-98. Lecture Notes in Computer Science*, vol. 1398. Berlin: Springer, pp. 137-142. Available at: https://doi.org/10.1007/BFb0026683

[8] Ma, Y., Xiang, D., Zheng, S., Tian, D. and Liu, X. (2019) 'Moving deep learning into web browser: how far can we go?', in *Proceedings of the World Wide Web Conference (WWW '19)*, San Francisco, CA, USA, 13-17 May 2019. New York: ACM, pp. 1234-1244. Available at: https://doi.org/10.1145/3308558.3313639

[9] Ma, Z., Chen, Z., Shen, H. and Liu, M. (2025) 'WeInfer: unleashing the power of WebGPU on LLM inference in web browsers', in *Proceedings of the ACM Web Conference 2025 (WWW '25)*, Sydney, NSW, Australia, 28 April - 2 May 2025. New York: ACM, pp. 4264-4273. Available at: https://doi.org/10.1145/3696410.3714553

[10] McMahan, H.B., Moore, E., Ramage, D., Hampson, S. and Arcas, B.A.Y. (2017) 'Communication-efficient learning of deep networks from decentralized data', in *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS 2017)*, Fort Lauderdale, FL, USA, 20-22 April 2017, pp. 1273-1282. Available at: http://proceedings.mlr.press/v54/mcmahan17a.html

[11] W3C (2023) *WebGPU Specification*. W3C Working Draft. Available at: https://www.w3.org/TR/webgpu/ (Accessed: 9 December 2025).

[12] BBC News Dataset (1997–2005) [online]. Available at:http://mlg.ucd.ie/datasets/bbc.html[Accessed 28 January 2026]