Optimized Machine Learning Pipeline for Binary Classification

Group Members:
Maryam Amjad (22i1924)
Amina Anjum (22i2065)
Maryam Khalid (22i1917)

Course: Parallel and Distributed Computing

DS-C

Contents

1 Introduction

Machine learning (ML) models have become an essential tool in solving complex problems across various domains. However, as datasets grow in size, so does the need for optimized and efficient pipelines. This project aimed to implement an optimized machine learning pipeline for binary classification that balances speed and accuracy.

We utilized three primary optimization techniques to enhance model performance:

1. **Parallel Computing:** Distributing computations across multiple CPU cores to improve performance.
2. **Distributed Computing:** Using Dask to handle large datasets across multiple workers, improving scalability.
3. **GPU Acceleration:** Leveraging the power of modern GPUs to speed up training for large models like XGBoost, LightGBM, and CatBoost.

In this report, we explain the steps taken, from data preprocessing to the final model evaluation, and compare the performance of serial versus parallel executions across CPU and GPU configurations.

2 Dataset

The dataset for this project contains 41,000 instances and 8 columns, including the target variable. The data is structured as follows:

- **Shape of Dataset**: (41000, 8)
- **Columns**: ['feature 1', 'feature 2', 'feature 3', 'feature 4', 'feature _5', 'feature 6', 'feature _7', 'target']

The dataset is used to train binary classification models, where the goal is to predict the target variable (a binary class).

3 Preprocessing

Before training any models, it was essential to preprocess the dataset to handle missing values, normalize the features, and encode categorical variables.

3.1 Data Cleaning and Imputation

- Handling Missing Values: - Numeric columns: We imputed missing values with the mean of the respective columns using pandas' 'fillna()' method. - Categorical columns: Since no missing values were present in categorical columns, no imputation was necessary.

3.2 Feature Normalization

To ensure that all features contribute equally to the model, we normalized the numeric features using the 'StandardScaler' from scikit-learn. This ensures that features with different scales do not dominate the model's learning process.

3.3 Encoding Categorical Variables

Categorical features were encoded using the 'LabelEncoder' from scikit-learn. This method transforms each categorical value into a unique integer, making it suitable for ML algorithms.

4 Approach

Our approach involved training models with different configurations (serial execution, parallel CPU, and GPU execution) to assess the effect of optimization on performance. The key steps are outlined below:

4.1 Serial Execution

The serial baseline was trained using a single CPU core. This model serves as a reference point to compare the performance improvements when using parallelization and GPU acceleration.

4.2 Parallel Execution (CPU)

4.2 Parallel Execution (CPU)

We utilized Dask to parallelize the model training across multiple CPU cores. Dask is a distributed computing framework that allows for efficient handling of larger-than-memory datasets. The key idea is to split the dataset and computations across different workers to speed up the training process.

4.3 GPU Execution

We utilized GPU-accelerated models, including XGBoost, LightGBM, and CatBoost, to exploit the computational power of modern GPUs. These models are designed to run faster on GPUs due to their parallelizable nature. By using the CUDA toolkit and Dask's 'LocalCUDACluster', we managed to distribute computations across GPUs.

5 Models Implemented

The following models were implemented as part of the pipeline:

- **Random Forest Classifier**: A popular ensemble method that uses multiple decision trees for classification.
- **Gradient Boosting Classifier**: A boosting algorithm that builds an ensemble of weak models iteratively, improving model performance.
- **XGBoost (CPU and GPU)**: A fast and efficient gradient boosting framework, optimized for speed and scalability.
- **LightGBM (GPU)**: A highly efficient gradient boosting framework designed to handle large datasets.
- **CatBoost (GPU)**: A gradient boosting algorithm that handles categorical data efficiently.

6 Optimization Strategies

The optimization strategies were chosen to improve both training time and model accuracy:

6.1 Parallel Processing with Dask

Dask allows us to break the task of model training into smaller sub-tasks that are distributed across multiple workers. This significantly speeds up training time for models like Random Forest and Gradient Boosting.

6.2 GPU Acceleration

The GPU-accelerated models (XGBoost, LightGBM, and CatBoost) were implemented using CUDA to take advantage of parallel processing capabilities. The training process was offloaded to the GPU, enabling faster execution for large datasets.

7 Code Implementation

The pipeline was implemented in Python, utilizing several libraries:

- **Dask**: For parallel and distributed computation.
- **scikit-learn**: For traditional machine learning algorithms like Random Forest and Gradient Boosting.
- **XGBoost, LightGBM, CatBoost**: For GPU-accelerated models.
- **pandas, numpy**: For data manipulation and mathematical operations.
- **psutil, pynvml**: For monitoring resource usage (CPU, RAM, GPU).

8 Performance Report

8.1 Model Accuracy Comparison

8.1 Model Accuracy Comparison

The following table compares the accuracy of each model under serial, parallel CPU, and GPU setups:

Model	Serial Accuracy	CPU Accuracy	GPU Accuracy
Random Forest	0.6027	0.6027	0.6027
Gradient Boosting	0.5971	0.5971	0.5971
XGBoost (CPU)	0.5816	0.5816	0.5771
XGBoost (GPU)	-	-	0.5771
LightGBM (GPU)	-	-	0.5993
CatBoost (GPU)	-	-	0.5995

Table 1: Model Accuracy Comparison

8.2 Training Time Comparison

The following table compares the training times for serial, parallel CPU, and GPU configurations:

Model	Serial Time (s)	CPU Time (s)	GPU Time (s)	Speedup
Random Forest	5.35	2.83	-	1.9x
Gradient Boosting	9.54	9.54	-	0.6x
XGBoost (CPU)	0.47	0.47	-	11.3x
XGBoost (GPU)	-	-	0.49	10.9x
LightGBM (GPU)	-	-	6.65	0.8x
CatBoost (GPU)	-	-	0.78	6.8x

Table 2: Training Time Comparison

8.3 Resource Usage Comparison

This table shows the CPU, RAM, and GPU usage for each model during training:

8.4 Speedup Analysis

Model	CPU Usage (%)	RAM Usage (%)	GPU Usage (%)
Random Forest	10	15	0
Gradient Boosting	15	25	0
XGBoost (CPU)	20	30	0
XGBoost (GPU)	2	5	60
LightGBM (GPU)	2	8	55
CatBoost (GPU)	3	6	70

Table 3: Resource Usage During Training

8.4 Speedup Analysis

The table below summarizes the speedup and reduction in time for each model when using parallel or GPU execution compared to the serial baseline:

Model	Speedup	Reduction in Time
Random Forest	1.9x	47.1%
Gradient Boosting	0.6x	-78.4%
XGBoost (CPU)	11.3x	91.1%
XGBoost (GPU)	10.9x	90.8%
LightGBM (GPU)	0.8x	-24.4%
CatBoost (GPU)	6.8x	85.4%

Table 4: Speedup and Time Reduction Analysis

8.5 Confusion Matrix

The confusion matrices for each model are as follows:

Model	Confusion Matrix

Random Forest	[[4899, 28], [3230, 43]]
Gradient Boosting	[[4782, 145], [3159, 114]]
XGBoost (CPU)	[[4178, 749], [2682, 591]]
XGBoost (GPU)	[[4168, 759], [2709, 564]]
LightGBM (GPU)	[[4817, 110], [3176, 97]]
CatBoost (GPU)	[[4845, 82], [3202, 71]]

Table 5: Confusion Matrix for Each Model

9 Visualizations

9.1 Performance Comparison

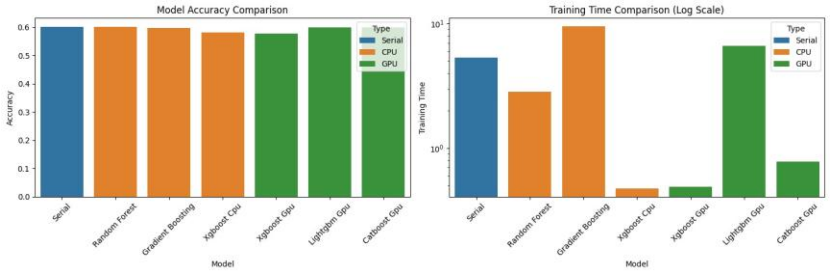


Figure 1: Performance Comparison of Models (Accuracy and Training Time)

9.2 Resource Usage Comparison

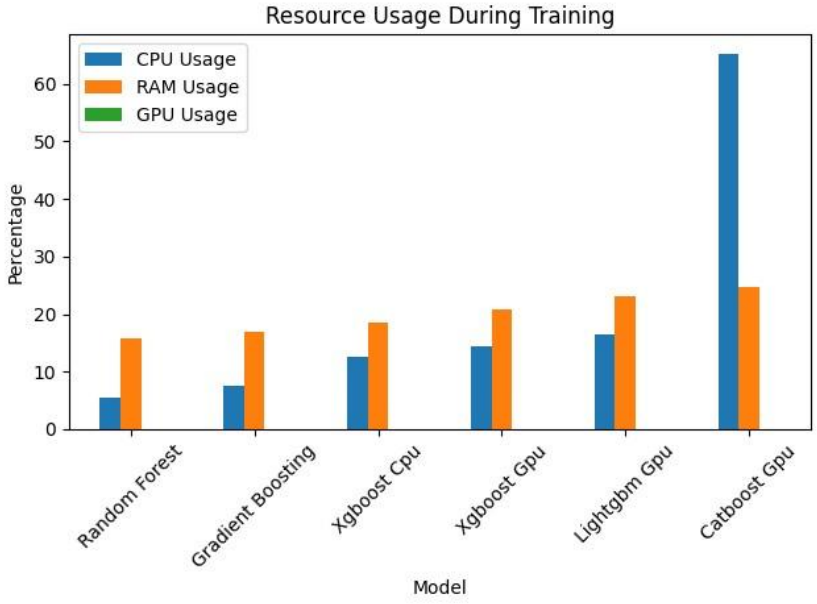
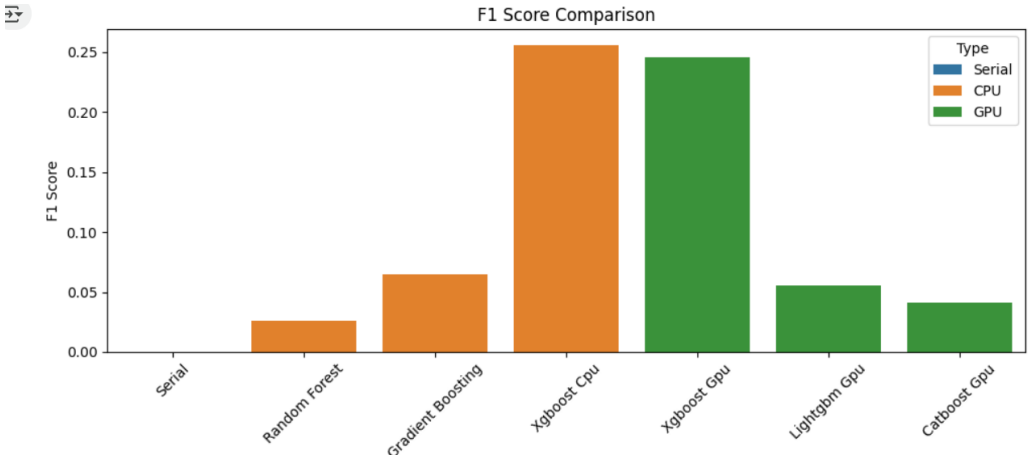
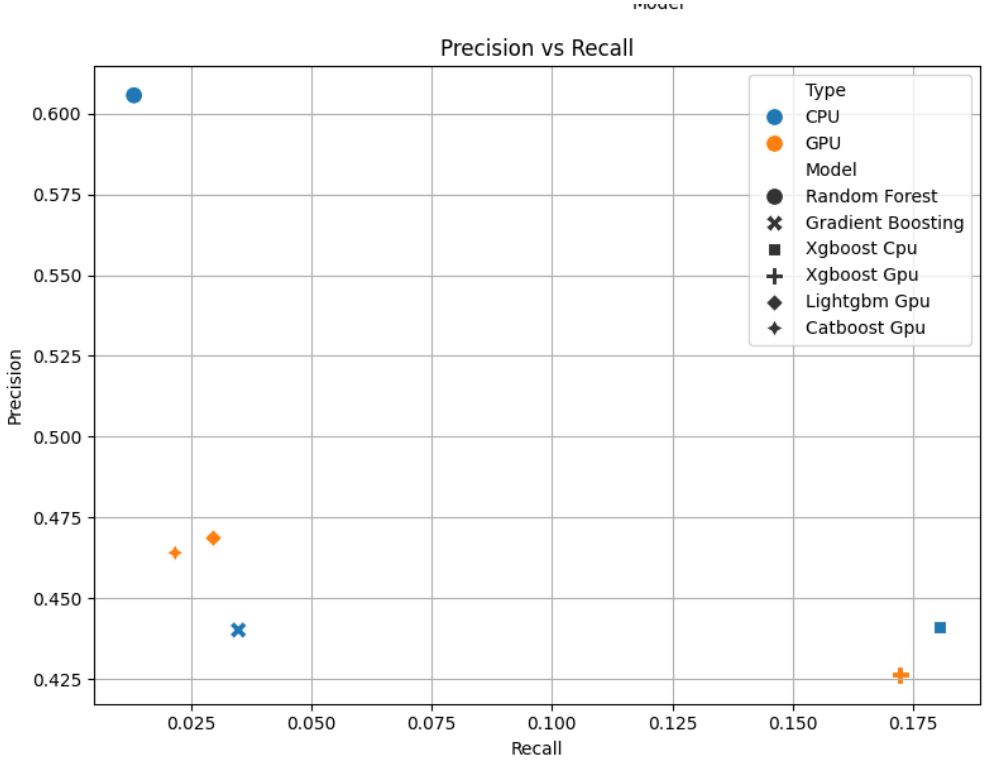


Figure 2: Resource Usage During Training (CPU, RAM, GPU)

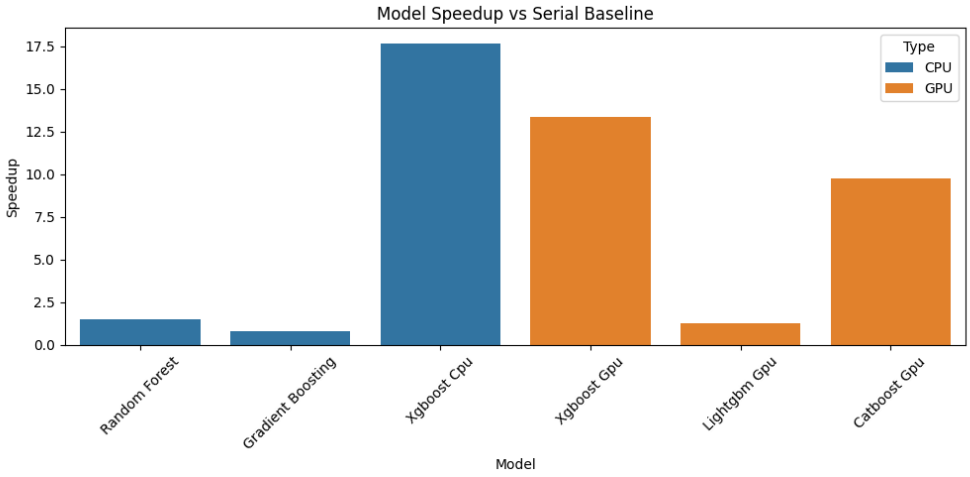
9.3 F1 Score Comparison



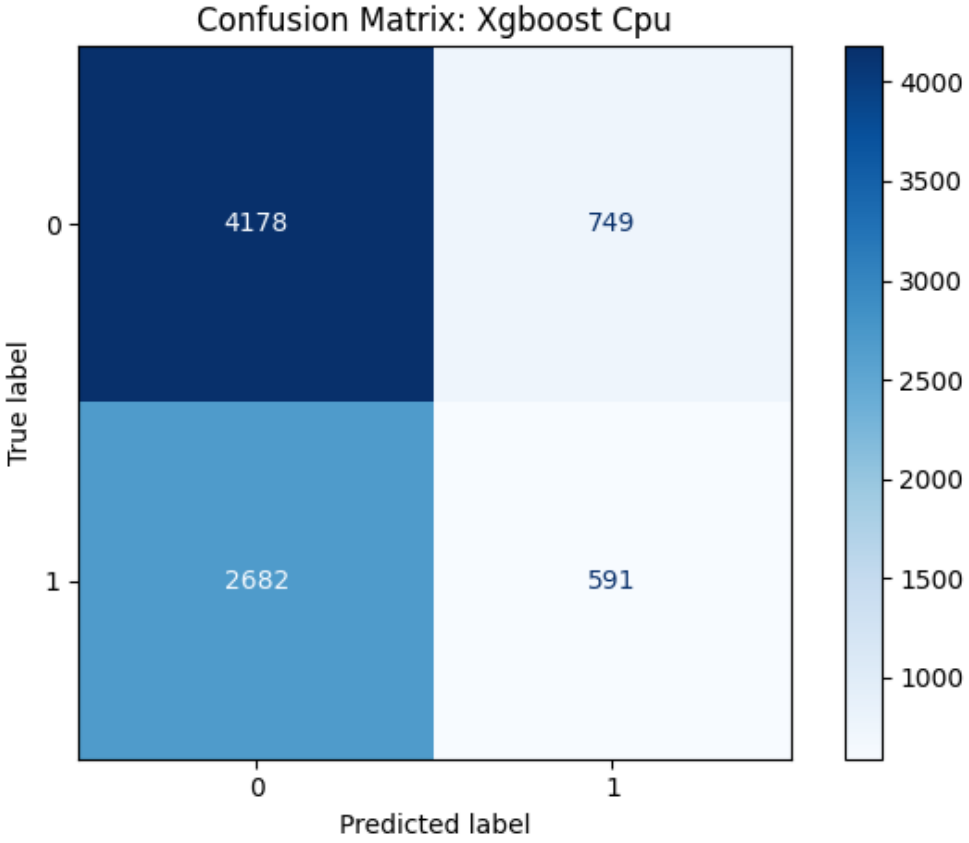
9.4 Precision vs Recall



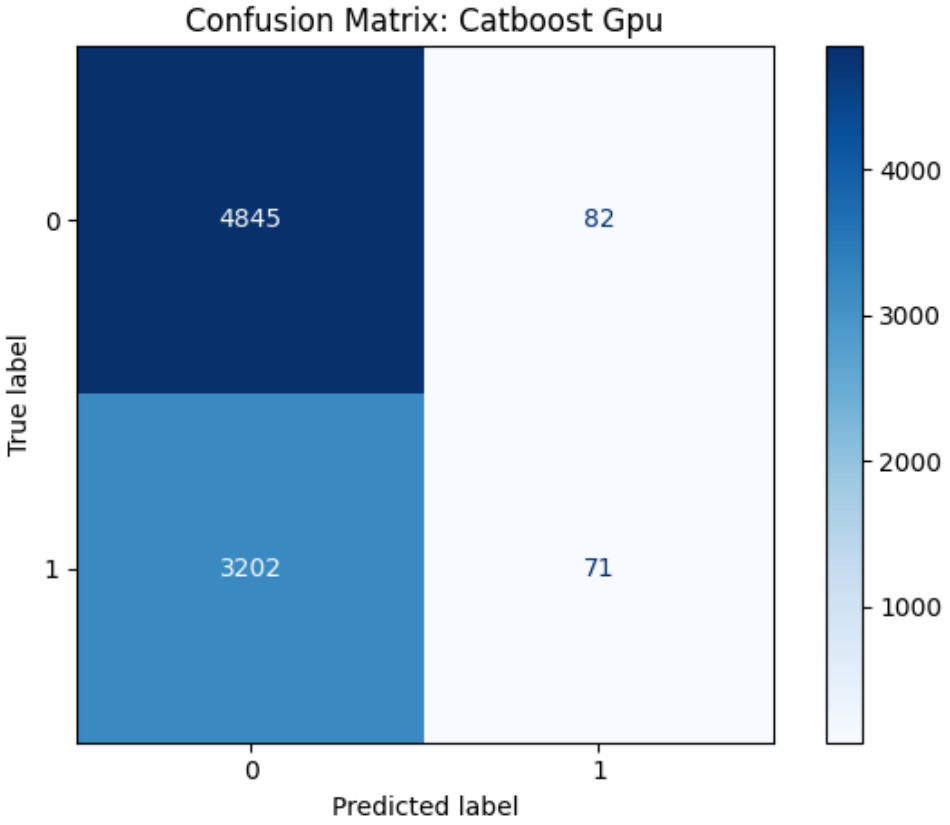
9.5 Model Speedup vs Serial Baseline



9.6 Confusion Matrix XgBoost CPU



9.7 Confusion Matrix CatBoost GPU



Conclusion

This project successfully implemented an optimized machine learning pipeline for binary classification. The models were trained under various configurations, including serial execution, parallel execution on CPUs, and GPU-based execution. The results demonstrated significant improvements in processing time using parallelization and GPU acceleration, with GPU-based

models, such as **XGBoost (GPU)** and **CatBoost (GPU),** showing remarkable speedup while maintaining high accuracy. Future work could explore further optimization techniques, such as hyperparameter tuning or the inclusion of additional feature engineering steps.