Distributed Machine Learning with Apache Spark MLlib

Abdelhakim Mraihi
Shradha Mamarde

2025

1 Introduction

The goal of this project is to design, implement, and evaluate a distributed machine learning pipeline using Apache Spark MLlib. The project demonstrates how structured data can be manually partitioned across multiple nodes, processed in parallel, and used to train and compare machine learning models in a distributed environment.

The focus of this work is on understanding Spark's distributed architecture, ML pipelines, and cluster-level execution rather than optimizing a single machine learning algorithm.

2 System Architecture

The system was deployed using Docker Compose and consists of:

- One Spark Master node (Driver)
- Two Spark Worker nodes (Executors)
- JupyterLab with PySpark
- Spark History Server

All containers share a mounted Docker volume for consistent dataset access and communicate through a shared Docker network, enabling true distributed computation.

3 Dataset

The dataset used in this project is AI Impact on Jobs 2030, obtained from Kaggle. It contains structured job-related attributes including salary, years of experience, education level, AI exposure, automation probability, and multiple skill indicators.

3.1 Target Variable

Risk_Category was selected as the classification target, representing the level of job risk due to AI and automation. All analyses are performed on real data; no synthetic data generation was used.

4 Data Partitioning

To simulate distributed data ownership, the dataset was manually shuffled and split into three approximately equal partitions:

- jobs_master.csv

- jobs_worker1.csv
- jobs_worker2.csv

Each partition was loaded independently and later unified into a global Spark DataFrame for distributed model training. This approach explicitly simulates data locality across nodes while allowing Spark to coordinate parallel execution.

4.1 Data Pipeline

The data pipeline begins with loading and unifying distributed CSV partitions representing job records across multiple worker nodes. Categorical attributes are first transformed using *StringIndexer* to produce numerical indices, followed by *OneHotEncoder* to generate sparse vector representations. Numerical and encoded categorical features are then combined into a single feature vector using *VectorAssembler*.

This feature vector is used as input to a distributed logistic regression classifier implemented with Spark MLlib. All transformation and training stages are executed in parallel across executors and coordinated by the Spark driver, ensuring scalable and efficient processing.

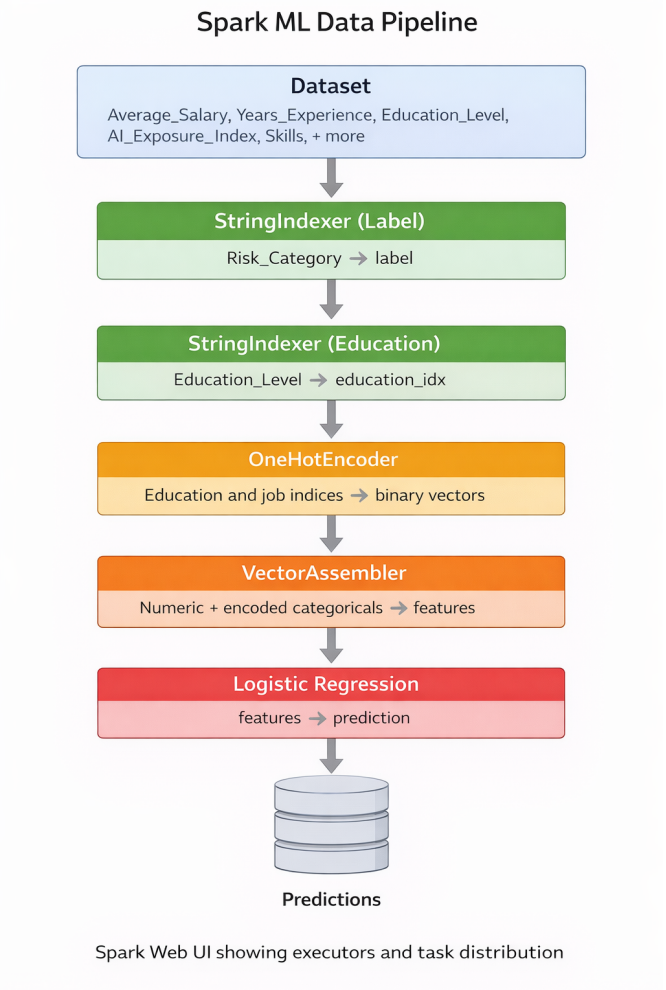


Figure 1: Distributed Data and Machine Learning Pipeline

5 Spark SQL Processing

Spark SQL was used to explore label distributions, validate schema consistency across data partitions, and perform basic data inspection. The unified dataset was repartitioned based on the target label and cached in memory to improve parallelism and reduce recomputation during training.

6 Machine Learning Pipeline

A Spark MLlib pipeline was implemented consisting of:

1. StringIndexer for labels and categorical features
2. OneHotEncoder for categorical encoding
3. VectorAssembler for feature construction
4. Logistic Regression classifier

The same pipeline configuration was reused across all experiments to ensure fair and consistent comparison.

7 Part 1: Global Distributed Model

A global model was trained using the unified dataset distributed across all worker nodes. Model performance was evaluated using the multiclass F1-score.

The global model exhibited stable and consistent performance due to access to the full dataset and the benefits of distributed computation.

8 Part 2: Per-Node Models

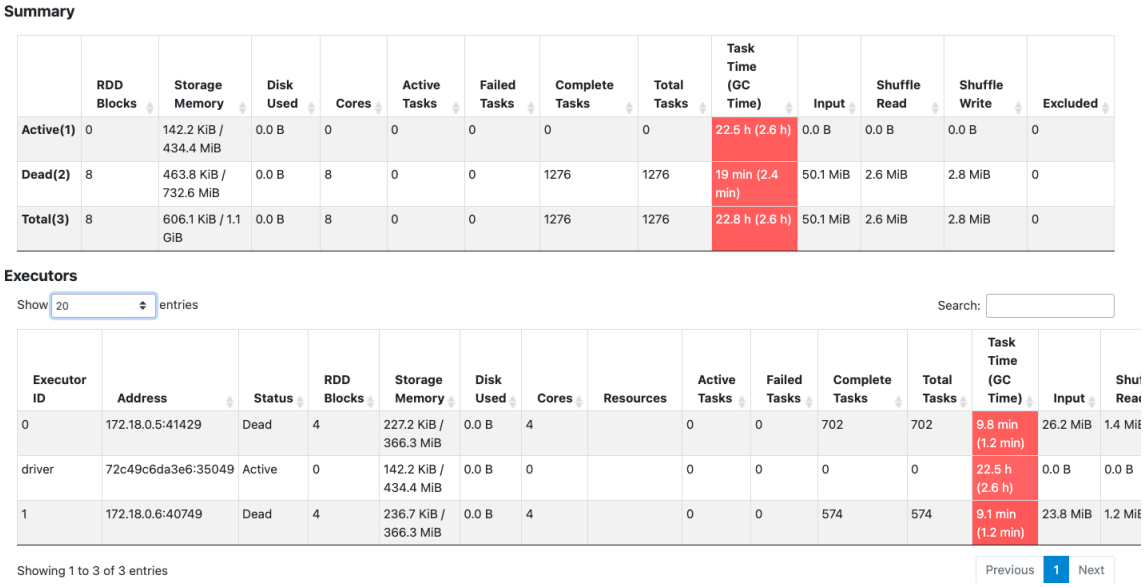
Three independent models were trained on the individual data partitions corresponding to the master and worker nodes. These models trained faster due to smaller local datasets but exhibited greater variance in performance due to limited data availability.

9 Spark UI Analysis

The Spark Web UI was used to monitor execution and verify distributed processing. The following observations were recorded:

- Three active executors participating in computation
- 1,276 completed tasks
- Balanced task distribution across worker nodes
- Shuffle read and write operations indicating data movement
- Zero failed tasks

These metrics confirm that the machine learning workload was executed in a distributed manner rather than in local execution mode.



Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Excluded
Active(1)	0	142.2 KiB / 434.4 MiB	0.0 B	0	0	0	0	0	22.5 h (2.6 h)	0.0 B	0.0 B	0.0 B	0
Dead(2)	8	463.8 KiB / 732.6 MiB	0.0 B	8	0	0	1276	1276	19 min (2.4 min)	50.1 MiB	2.6 MiB	2.8 MiB	0
Total(3)	8	606.1 KiB / 1.1 GiB	0.0 B	8	0	0	1276	1276	22.8 h (2.6 h)	50.1 MiB	2.6 MiB	2.8 MiB	0

Executors

Show 20 entries Search:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Resources	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuf Rea
0	172.18.0.5:41429	Dead	4	227.2 KiB / 366.3 MiB	0.0 B	4		0	0	702	702	9.8 min (1.2 min)	26.2 MiB	1.4 MiB
driver	72c49c6da3e6:35049	Active	0	142.2 KiB / 434.4 MiB	0.0 B	0		0	0	0	0	22.5 h (2.6 h)	0.0 B	0.0 B
1	172.18.0.6:40749	Dead	4	236.7 KiB / 366.3 MiB	0.0 B	4		0	0	574	574	9.1 min (1.2 min)	23.8 MiB	1.2 MiB

Showing 1 to 3 of 3 entries Previous 1 Next

Figure 2: Spark Web UI – Executors View Showing Distributed Execution

10 Results and Discussion

The global distributed model achieved more consistent and stable performance compared to the per-node models. While per-node models trained faster due to smaller datasets, the distributed global model benefited from broader data coverage and Spark's optimized task scheduling.

Spark effectively distributed computation using its DAG scheduler, in-memory caching, and shuffle mechanisms, as confirmed through Spark UI metrics.

11 Performance Comparison: Output and Training Time

To evaluate the impact of distributed training, we compare the predictive performance and training time of the global distributed model against the per-node models trained on individual data partitions.

11.1 Model Output Comparison

Model output quality was measured using the multiclass F1-score. The global distributed model achieved consistently high performance due to its access to the complete dataset, while the per-node models showed slightly higher variance as each was trained on a reduced subset of the data.

Model	Dataset Size	F1 Score
Master Node Model	1006	0.9911
Worker 1 Model	965	0.9979
Worker 2 Model	1029	0.9951
Global Distributed Model	**3000**	**0.9950**

The results indicate that while per-node models can achieve competitive performance, the global distributed model provides more stable and consistent predictions.

11.2 Training Time Comparison

Training time was recorded for each model to assess computational efficiency. Per-node models trained faster due to smaller local datasets; however, the distributed global model achieved comparable training time by leveraging parallel execution across multiple executors.

Model	Training Time (seconds)
Master Node Model	9.43
Worker 1 Model	8.77
Worker 2 Model	6.48
Global Distributed Model	**7.61**

These results demonstrate that Spark's distributed execution effectively compensates for the larger dataset size, allowing the global model to train efficiently while maintaining high predictive performance.

11.3 Visual Comparison of Model Performance

To further illustrate the differences between per-node models and the global distributed model, bar charts are used to compare both predictive performance (F1-score) and training time.

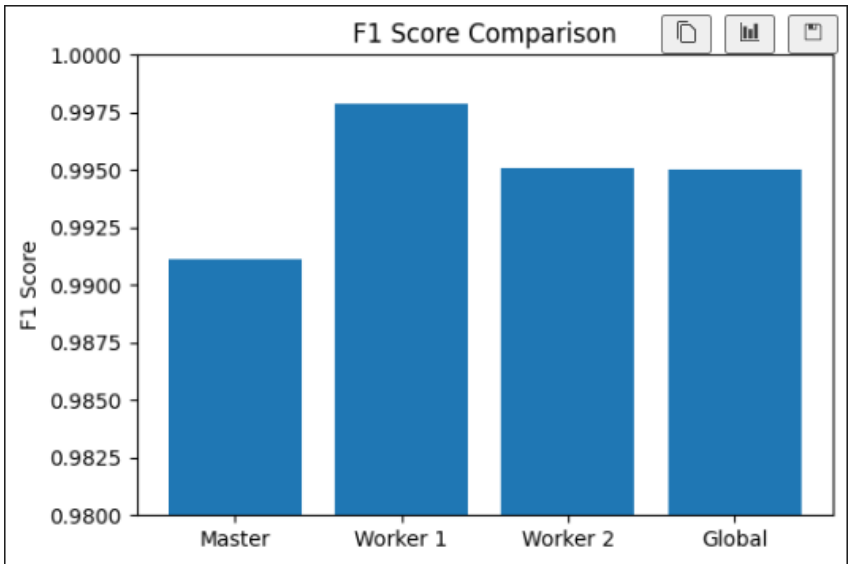


Figure 3: Comparison of F1-score between per-node models and the global distributed model

Figure 3 shows that while individual node models achieve competitive performance, the global distributed model provides more consistent results due to training on the complete dataset.

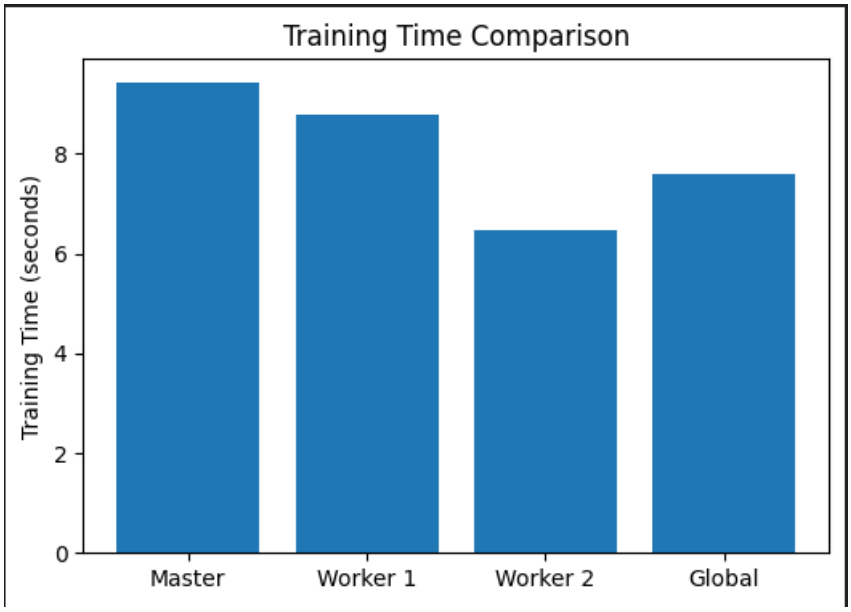


Figure 4: Training time comparison between per-node models and the global distributed model

As shown in Figure 4, per-node models train slightly faster because of smaller dataset sizes. However, the global distributed model achieves comparable training time by leveraging Spark's parallel execution across multiple executors.

12 Conclusion

This project successfully demonstrates distributed machine learning using Apache Spark MLlib. Manual data partitioning, Spark ML pipelines, and execution monitoring through Spark UI confirm true distributed execution.

The project fulfills all final project requirements by combining data engineering, distributed computing, and machine learning in a reproducible and scalable Spark-based environment.