

DA 231o: Data Engineering at Scale

Real-Time Fake News Prediction

Course Project Report

Version 1.0

Last Update: 20/11/2025

Ritik Agrawal (ritikagarwal@iisc.ac.in)

Sanjana R (sanjana2@iisc.ac.in)

Himanshu Sharma (himanshu5@iisc.ac.in)

Mayank Teckchandani (tmayank@iisc.ac.in)

GitHub Repo

<https://github.com/dlquad/iisc-deas-project-1>

Table of Contents

Introduction	3
Problem Statement	3
Data Description.....	3
Exploratory Data Analysis.....	3
Model Building	4
Benchmarking System Architecture	6
Server Configuration	6
Spark Configuration.....	6
Automation Workflow.....	6
Amdahl's Law Strong Scaling.....	7
Why did gains start diminishing?	7
Gustafson's Law of Weak Scaling	8
Effect of Shuffling on Performance	8
Worker Sweep Analysis	8
Worker Sweep Deep Dive: Proving the “Sweet Spot”	9
Inferencing Real-Time data with Kafka and Spark Streaming	9
Future Actions	10

Table of Figures

Figure 1 - Missing Values Chart	3
Figure 2 - Text Analysis Chart	4
Figure 3 - Sentiment Analysis Chart	4
Figure 4 - NGram Analysis	4
Figure 5 - Word Cloud	4
Figure 6 - Logistic Regression Scores Chart	5
Figure 7 - XG Boost Scores Chart.....	5
Figure 8 - Bi-Directional LSTM Scores Chart.....	5
Figure 9 - System Architecture	6
Figure 10 - Numa Benchmark.....	6
Figure 11 -- Gustafson's law for weak scaling against E2E Time (left), Throughput speedup (right).....	8
Figure 12 - Average Stage throughput vs Resources.....	8
Figure 13 - Streaming Architecture with Kafka + Spark	9

Introduction

Fake news has become a significant challenge in the digital age, impacting public opinion and spreading misinformation. This project aims to develop a machine learning model to detect fake news using data-driven techniques.

Problem Statement

The objective is to classify news articles as either fake or real based on their content. Accurate detection can help mitigate the spread of misinformation and support fact-checking efforts.

Data Description

The dataset used for the project contains news articles labelled as either fake or real. Key features include:

- Title: Headline of the news article
- Text: Full content of the article
- Label: Target variable (fake or real)

Initial data inspection revealed missing values in critical columns. Rows with missing titles or text were removed to ensure data quality, resulting in a clean dataset with complete records for modelling. Data retention after cleaning was above 95%, and memory usage was optimized for analysis.

Exploratory Data Analysis

EDA was performed to understand the distribution of fake and real news, identify key features, and visualize patterns. Key steps included:

1. **Missing Value Analysis:** Visualized and quantified missing data, ensuring only complete records were used for modelling.

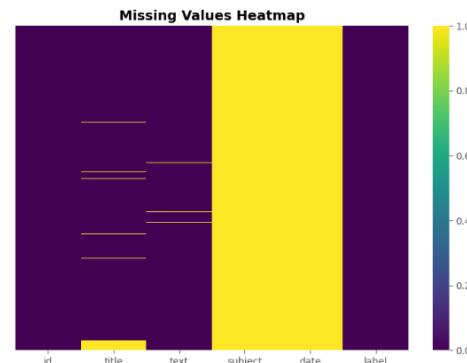


Figure 1 - Missing Values Chart

2. Text Analysis

Calculated and visualized the distribution of title and text lengths, as well as word counts, to identify differences between fake and real news articles.

a. Text Pattern

Analysis

b. Generated Word

Clouds

- c. N-Gram Analysis
- d. Sentiment Analysis

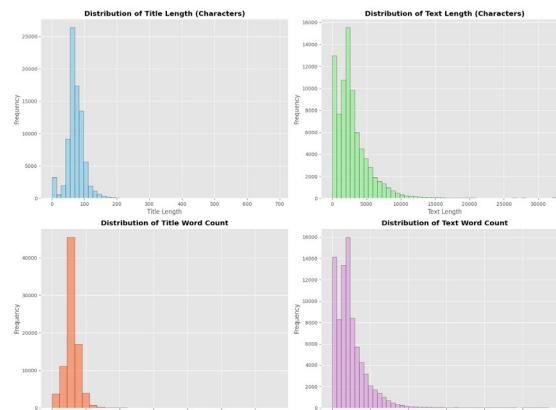


Figure 2 - Text Analysis Chart

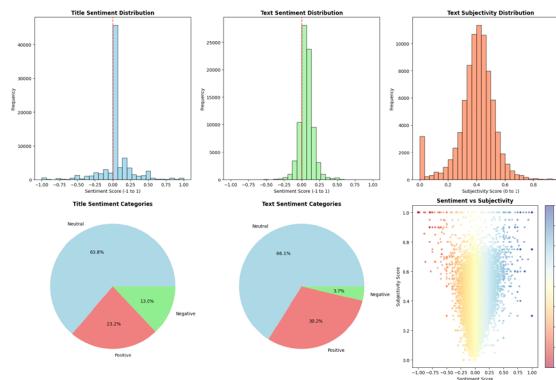


Figure 3 - Sentiment Analysis Chart

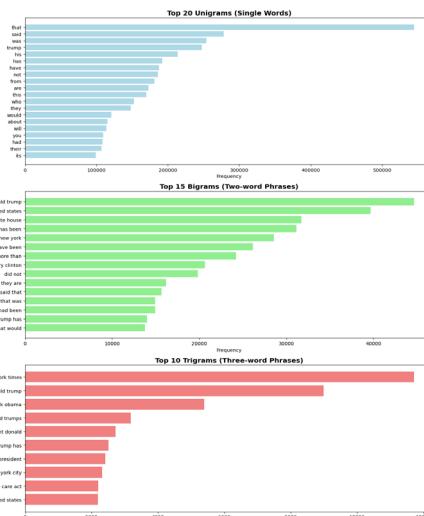
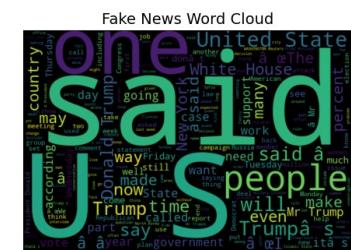


Figure 4 - NGram Analysis



Real News Word Cloud

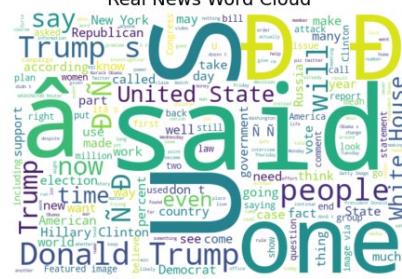


Figure 5 - Word Cloud

3. Feature Engineering: Created new features to enhance model performance.

- a. **Text Statistics** features, such as title length, text length, title word count, and text word count
 - b. **Emotional Indicators and Language Markers**
 - c. **Advanced NLP, TF-IDF vectors (word importance), N-gram features**

Model Building

Several machine learning models were explored, including Logistic Regression, XG Boost and Deep Learning Model like LSTM. The modelling pipeline included:

1. **Text Preprocessing:** Tokenization, stop word removal, and regular expression cleaning to standardize text data.
 2. **Vectorization:** TF-IDF vectorization to convert text into numerical features suitable for machine learning.
 3. **Model Training:** Built and compared models such as Logistic Regression, XG Boost and LSTM.
 4. **Feature Selection:** Evaluated the impact of engineered features on model accuracy.

Models Used

1. **Logistic Regression:** Baseline linear model using TF-IDF features. Achieved strong accuracy and interpretability.
 1. Accuracy: 80.44%
 2. ROC AUC: 0.88

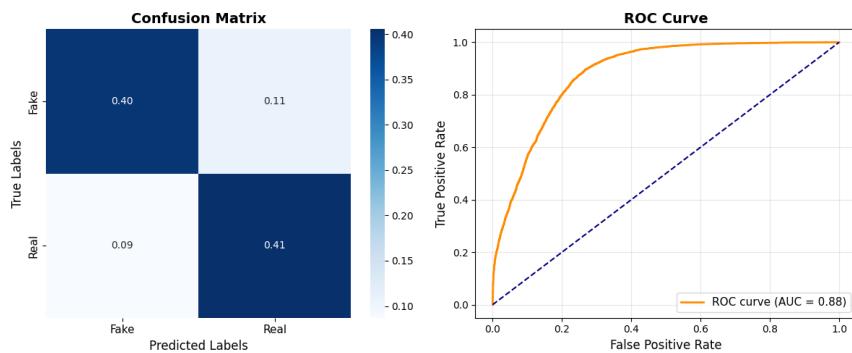


Figure 6 - Logistic Regression Scores Chart

2. **XGBoost:** Advanced gradient boosting model with hyperparameter tuning. Provided the highest accuracy and robust performance.
 1. Accuracy: 85.14%
 2. ROC AUC: 0.93

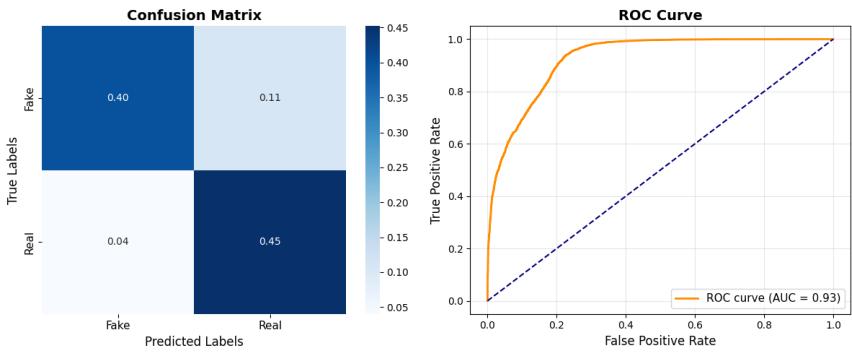


Figure 7 - XG Boost Scores Chart

3. **Bi-directional LSTM (Deep Learning):** Neural network model using word embeddings and sequence modelling. Captured complex text patterns.
 1. Accuracy: 87%
 2. ROC AUC: 0.87

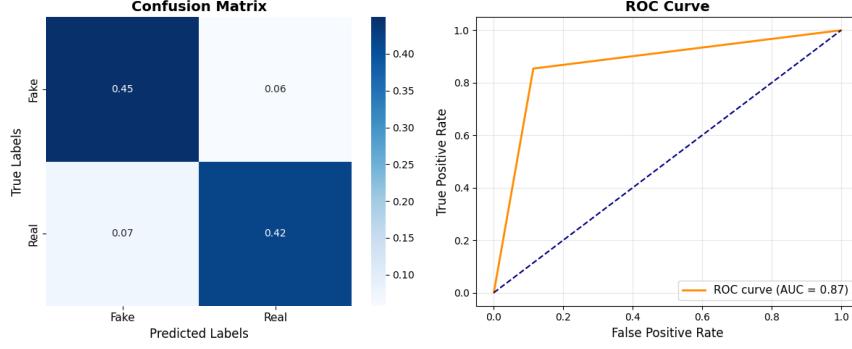


Figure 8 - Bi-Directional LSTM Scores Chart

Benchmarking System Architecture

We created a performance benchmarking environment that reliably tests different benchmark configurations. This environment comprises physical hardware, a containerized Spark cluster, and a set of automation scripts to orchestrate the experiments.

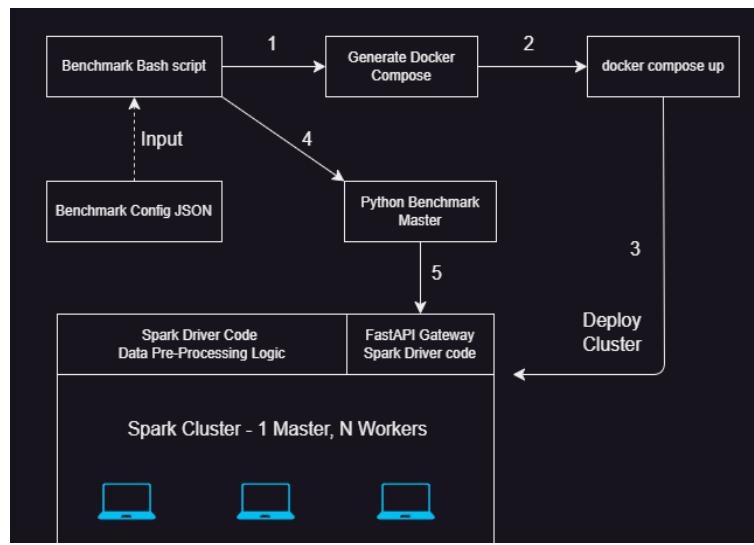


Figure 9 - System Architecture

Server Configuration

All benchmarks were executed on a single, bare-metal server to eliminate performance variability due to virtualization or shared cloud resources. The server specifications are as follows

1. CPU: Dual Socket, Xeon 6767P
2. Cores: 64 cores per socket
3. Sub-NUMA Nodes: 2 per socket
4. RAM: DDR5, 6400 MT/s
5. No network or virtualization overhead

NUMA Benchmark

Node	0	1	2	3
0	190.7	189.6	89.0	90.0
1	189.8	189.9	90.1	88.9
2	93.4	94.1	190.3	189.8
3	90.2	89.0	190.1	190.2

Figure 10 - Numa Benchmark

Spark Configuration

To ensure low overhead and consistent benchmarking, we applied the following tunings:

- **Memory Management:** Enabled *G1GC*, activated *off-heap memory*, and reserved 80% of the heap for execution (`spark.memory.fraction`).
- **Optimization:** Enabled *Adaptive Query Execution (AQE)* and set `spark.sql.autoBroadcastJoinThreshold` to *50MB* to minimize expensive shuffles.
- **Resources:** `spark.executor.cores` and `spark.executor.memory` were set to allocate specific CPU/RAM limits.

Automation Workflow

The end-to-end benchmarking process is automated by a collection of scripts that manage configuration, setup, execution, and teardown:

- **Configuration JSON:** A master JSON file stores an array of test scenarios. Each JSON object defines a complete set of parameters for a single benchmark run (e.g., worker count, memory, dataset scale).

- **Generation:** A Python script parses a single configuration object from the JSON file and dynamically generates a corresponding `docker-compose.yml` file.
- **Execution:** A single bash script is used to kickstart the benchmark. The JSON file is ingested, and each configuration is used to spin a new set of Docker containers for the Spark cluster. A Python script is called to initiate the data pre-processing pipeline, and metrics are recorded in a CSV file for later analysis.
- **NUMA Pinning:** To ensure stable performance and avoid inter-socket communication, the Python script that generates the Docker Compose file ensures that each Spark worker is limited to a single socket and sub-NUMA node.

Amdahl's Law Strong Scaling

Strong scaling tests measured efficiency by solving a fixed-size problem (100% dataset) while increasing the number of 4-core workers. Throughout the test, total RAM was held constant at 384 GB and distributed evenly across the cluster to isolate the impact of added processing power.

The results align with Amdahl's Law, demonstrating **diminishing returns** as more cores are added. The speedup increases steadily up to 4 workers (16 cores), achieving a peak of **1.66x**. Critically, at 8 workers (32 cores), the speedup flattens. This indicates that for this fixed problem size, the overhead of task scheduling, network shuffling, and data aggregation across 8 workers has surpassed the computational gains. The sequential portion of the pipeline and communication overhead have become the dominant bottleneck.

Why did gains start diminishing?

We take three metrics to find the reason:

1. **Total Executor Run Time:** Cumulative time across all stages, including overheads (scheduling, I/O, shuffling).
2. **Total Executor CPU Time:** Time spent strictly on usable data processing work.
3. **CPU Efficiency:** (**Executor CPU Time / Executor Run Time**) * 100

num_workers	cores_per_worker	E2E_time	E2E_throughput	Throughput Speedup
1	4	270.34	2933.36	1.00
2	4	183.86	4313.01	1.47
4	4	163.00	4865.14	1.66
8	4	146.84	5400.47	1.84
16	4	134.40	5900.44	2.01

The strong scaling test shows the workload getting slower instead of faster as you add more cores. The **Executor Run Time** (total time) increased by 36% from 4 cores to 64 cores. Ideally, this time should drop. This happens because the **effort to coordinate the extra workers (overhead)** is **greater than the gain from extra computing power**.

The CPU Efficiency stayed very low (around 32-39%), meaning the **cores were idle** or waiting for data most of the time. This wait time is caused by bottlenecks like moving data between workers (shuffling), limited network speed (I/O contention), or a single, slow part of the code that cannot be sped up (Amdahl's Law). The conclusion is that the workload is constrained by communication, not computation.

Num workers	Cores per worker	Total Executor Run_Time	Total Executor CPU_Time	CPU Efficiency
1	4	544593	186269	34.20
2	4	544582	210922	38.73
4	4	565368	209483	37.05
8	4	657928	212865	32.35
16	4	743418	241550	32.49

Gustafson's Law of Weak Scaling

We evaluated weak scaling by proportionally increasing both dataset size and resources, starting at 4 cores. While the goal was constant execution time, the results show a divergence between processing power and overhead.

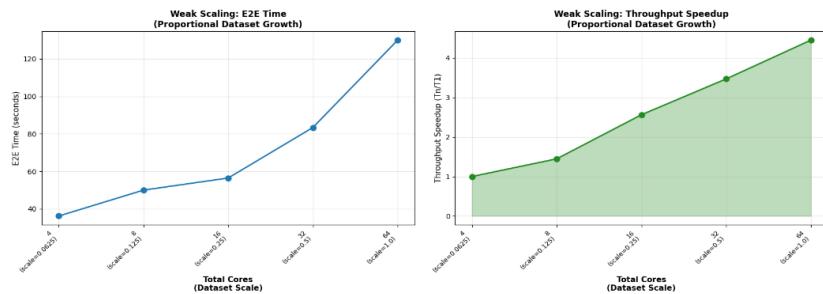


Figure 11 -- Gustafson's law for weak scaling against E2E Time (left), Throughput speedup (right)

The primary goal is to process a larger dataset while keeping the execution time constant. The **throughput** metric confirms the system is scaling, as it successfully grows from **1370.5 to 6098.4 records/sec**. This 4.45x increase in processed records/sec demonstrates the pipeline's ability to handle a larger workload as more resources are added, which is the expected positive outcome of a weak scaling test.

num_workers	Cores per worker	Dataset scale	E2E time	E2E throughput	Throughput Speedup
1	4	0.0625	36.16	1370.52	1.00
2	4	0.125	49.99	1983.10	1.45
4	4	0.25	56.45	3512.11	2.56
8	4	0.5	83.40	4754.02	3.47
16	4	1	130.04	6098.38	4.45

However, the system exhibits sub-linear scaling. In a perfect weak-scaling system, the E2E time would be a flat line, but here the E2E time increases by 3x between 4 and 64 cores. This slowdown is a classic symptom of **system overhead** becoming the dominant bottleneck. In a distributed system like Spark, adding more cores and workers increases communication and coordination costs, particularly from network-intensive data shuffling. This non-scalable overhead explains both the rising E2E time and the sub-linear throughput gains at higher core counts.

Effect of Shuffling on Performance

To understand the cost of shuffling, we average the throughputs at each stage and plot them against the hardware resources, thus discarding the inter-stage penalties that Spark incurred.

It is evident that at the stage level, both strong and weak scaling perform similarly. Strong scaling provides a more consistent increase due to the same dataset volume. This plot helps us put the cost of shuffling into perspective, underscoring the importance of parallelization.

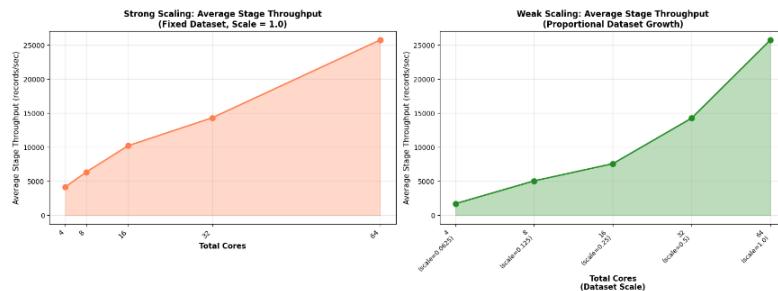


Figure 12 - Average Stage throughput vs Resources

Worker Sweep Analysis

To determine optimal allocation for the 64-core system, we conducted a worker sweep on the full dataset. The **8W-8C configuration** emerged as the "sweet spot," balancing parallelization and communication.

- **8W-8C (Optimal):** Fastest E2E (~117s) and highest throughput (>6700 records/sec).
- **1W-64C:** Slower (~192s) due to poor parallelization and GC pressure.
- **16W-4C:** Slower due to excessive scheduling and shuffle overhead.

Num Workers	Cores per worker	E2E Time	E2E Throughput
1	128	268.53	2953.16
2	64	161.92	4897.47
4	32	126.04	6291.50
16	8	132.56	5982.44
32	4	135.90	5835.26

Worker Sweep Deep Dive: Proving the “Sweet Spot”

Since we vary the number of workers, a more suitable metric for confirming the above results is **Total JVM GC Time**, which is the amount of time the executors spent in garbage collection.

Minimizing E2E time requires balancing Total GC Time (cumulative CPU usage, not pause time) against system overheads.

- **16W-4C (Overhead Bottleneck):** Achieved the lowest GC time (9306s) but suffered poor E2E time (144.96s). The 16-worker setup created excessive network I/O and serialization overhead.
- **8W-8C (Optimal):** The “sweet spot.” It accepts higher GC time (11113s) but drastically reduces coordination overhead by using fewer workers.

num_workers	cores_per_worker	Total_JVM_GC_Time
1	128	119477
2	64	44796
4	32	15778
16	8	8542
32	4	8399

Inferencing Real-Time data with Kafka and Spark Streaming

We implement a scalable real-time text classification pipeline using **Apache Kafka** (N Controllers, M Brokers) as the central message bus and **Apache Spark Streaming** for high-throughput transformation components that make this possible.

- **Data Ingestion:** A Web Scraping Producer spiders target websites to extract raw content. It structures the data (*source, title, text*) and publishes it immediately to the *scraped-data* topic.
- **Stream Processing:** A Spark Streaming application subscribes to the scraped data. It performs real-time cleaning, normalization, and feature extraction, publishing the transformed records to the *processed-data* topic.
- **Inference Engine:** The Model Inference Consumer reads the processed streams and applies the previously trained machine learning model to predict

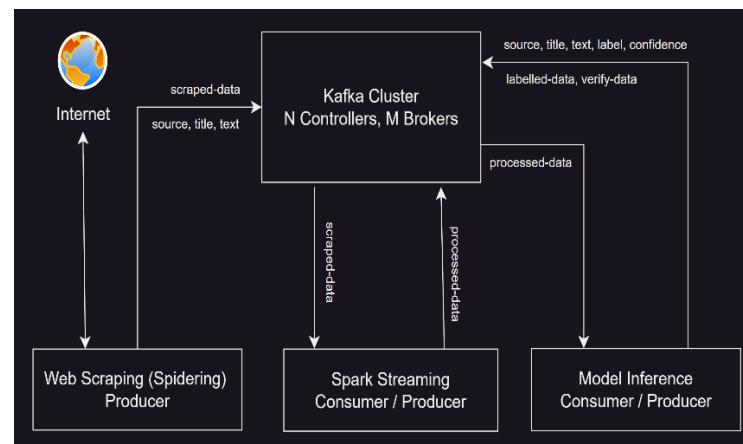


Figure 13 - Streaming Architecture with Kafka + Spark

classification labels and assign confidence scores. Content with a confidence score greater than 80% are deemed reliable and pushed to the *labelled-data* topic, while the others are flagged for human-in-the-loop review and routed to the *verify-data* topic.

Future Actions

- **Distributed Scaling & Persistence:** Optimize the Spark pipeline configuration for deployment on a fully distributed cluster to improve scalability. Additionally, integrate a permanent storage layer (e.g., S3, HDFS, or a database) to persist processing results for historical analysis and auditing.
- **Human-in-the-Loop Retraining Pipeline:** Implement an active learning feedback loop by integrating a training component with the Kafka verify-data topic. A user interface can be implemented for human verification for these sources. These verified results will be aggregated over specific intervals to trigger the ML training pipeline automatically, allowing the model to continuously learn and update based on fresh, human-verified data.