STATISTICS, OPTIMIZATION AND INFORMATION COMPUTING
Stat., Optim. Inf. Comput., Vol. **16**, December 2026, pp 5847–5864.
Published online in International Academic Press (www.IAPress.org)

Performance evaluation of distributed LSTM-based sentiment analysis on Hadoop MapReduce and Apache Spark

Shereen A. Kaml*, Mohamed Abdelfattah, Noha E. El-Attar

Department of Information Systems, Faculty of Computers and Artificial Intelligence, Benha University, Benha, Egypt

Abstract The release of massive text data volumes has led to an increased demand for distributed processing frameworks that are able to support deep learning applications. However, despite the widespread use of LSTM for sentiment analysis due to its ability to capture long-term contextual dependencies, limited studies have systematically evaluated Hadoop MapReduce and Apache Spark under identical experimental conditions for large-scale LSTM-based sentiment analysis. Data from 200 MB to 1 TB in size were used in experimentation in the same cluster configuration. The findings show that Spark provides systematically lower execution time and better scalability on all workloads due to its in-memory processing framework, whereas MapReduce is slower because disk writes and file merges are time-consuming. Spark consistently achieved up to a 63% decrease in execution time when processing the 1 TB dataset on a three-node cluster, demonstrating its efficiency for large-scale deep learning workloads. Furthermore, this study develops empirical execution-time models with $R^2 > 0.95$ to estimate the scalability of both frameworks across different dataset sizes, providing practical guidance for selecting appropriate distributed platforms for large-scale sentiment analysis. Our results provide partial insight into trade-offs between the two frameworks and assist the practitioners in selecting appropriate architectures of distributed NLP applications. Our future work will build on the analysis of transformer-based architecture along with streaming real-time data.

Keywords Hadoop MapReduce; Apache Spark; LSTM; Sentiment Analysis.

AMS 2010 subject classifications 68T07, 68T50, 68W10, 68P20.

DOI: 10.19139/soic-2310-5070-3775

1. Introduction

Digital ecosystems have been rapidly increasing resulting in the exponential increase in the rate of data transfer, amount of data, speed of data transfer and complexity of data is also increasing. This is because there are social media, IoT devices and sensor networks that generate data of varying magnitude on which large scale online services are relied on. The 5 Vs (Volume, Velocity, Variety, Veracity and Value) underscore the reason why it is so challenging to regulate and extract any tangible value out of this volume of data. To overcome these constraints we need a solution that is easily scalable, happens to be real time and also is accurate with time [1],[2].

In the current state of affairs, the sheer volume of content created by users has become a treasure trove to both states and corporations. These online footprints, be it a quick tweet, comment on Facebook, or an in-depth review on Amazon, serve as a direct peep into societal opinion and what is in reality capturing people's attention or what aggravates them is a reflection of the way people are feeling. Here sentiment analysis also known as opinion mining becomes involved. Its primary task is to cut noise, and in the process to isolate the emotional coloring of a writing, and to cluster it into simple categories, such as positive, negative, or neutral. It is in essence the tool that will allow brands and politicians to actually hear the people's pulse [3],[4]. However, it is not only about marketing, but also

*Correspondence to: Shereen A. Kaml (Email: shereen.kaml@fci.bu.edu.eg). Department of Information Systems, Faculty of Computers and Artificial Intelligence, Benha University, Benha, Egypt (13518).

ISSN 2310-5070 (online) ISSN 2311-004X (print)
Copyright © 2026 International Academic Press

regarding sentiment analysis, that is important to identify harmful content or detect fake news. We have observed that when a message is constructed in such a way that it gives us a raw emotion, fake news has a tendency to go viral. In this kind of a fast-paced world, monitoring the mood of the people now is not only a nice-to-have but an absolute must [5].

The move to deep learning did not occur through coincidence. At the beginning, everybody relied on SentiWordNet due to its easiness but the ease faltered with regard to context. These models using lexicon were fundamentally blind to sarcasm and irony, and simply could not cope with domain-specific slang. That is what made the field necessitate the move towards more sophisticated machine learning systems [6].

Such models as SVM or Naive Bayes were more accurate, but the actual nightmare was that they required humans to select the data features manually. Deep learning transformed this as it allows the model to discover the tricky nature of language discovering it all on its own. LSTMs were a great relief in this case since they addressed the problems that people had with the old RNNs. As they employ gated cells, they behave as a memory that remembers the entire sentence and therefore, the model does not lose the relationship between the words even in the event they are separated by a long distance [7].

LSTMs are implemented to a great effect when required to follow a story, be it a short tweet or a lengthy product review. Simple models, by contrast, are confused by word order, or even by a single negation and lstm, on the contrary, can follow the flow of the text. However, it also comes at a price, the enormous calculation burden they impose. These do not just work on any standard machine and give results, and the scale of processing data at these levels actually compels you to shift to the high performance distributed systems just to keep pace [7].

LSTMs are computationally demanding, and this is their primary downside. The training process is slower since it is step-by-step and consumes much more resources compared to simple models. A common setup just breaks down once you begin to feed large collections of data to it. A single machine cannot quite cope with a memory overflowing and a slow disk access slowing everything. This is the reason why migrating to distributed computing systems has become inevitable as opposed to being optional to manage such demanding deep learning-based tasks [8].

Hadoop served a long time as the most suitable choice of big data since it was excellent at processing large volumes of data without collapsing. However, the issue is that it is very slow as it constantly saves and reads the data on the disk. This is a poor application to deep learning that requires the execution of the same operations repeatedly at a high rate. Unless you are in need of tracking the sentiments of people on the social media in real time, Hadoop is simply too slow. The results of sentiment analysis are usually needed at that moment, and any delay renders the entire process useless [9].

Spark is different since it stores data in memory rather than retrieving it from the disk. This removes the repetitive training tasks required to a large extent as it is much faster. It equally integrates SQL and machine learning into a single setup together with graph processing and that is the reason why everyone is using it today as the main framework when implementing AI projects [10], [11].

Although the use of distributed systems is a large matter in sentiment analysis, we observe that few studies have compared Hadoop and Spark directly to each other, particularly with intensive models such as LSTMs. The majority of researches merely examine the accuracy and end there. They do not actually discuss the speed of execution or the efficiency of memory. It is difficult to determine which of them functions better in the real life. That is the reason why we chose to test both frameworks adequately with a robust model to observe the actual variance.

To seal these gaps, we chose the LSTM model for both frameworks to run the same conditions. We were not only trying to determine whether the performance was good, but to truly investigate the performance metrics that are important when data scales. We compared the change in performance of the execution time as well as how the resources were actually being utilized as the datasets increased, and this gave a better idea on which framework better manages the heavy activities.

1.1. Sentiment Analysis for Large-Scale Distributed Environments

Sentiment analysis or opinion mining is simply the process of deriving meaning in raw text by identifying positive, negative, or even neutral texts. This has become a survival mechanism to brands and governments that must be

ahead of the mood of the people since everybody is putting their ideas online these days. It is all about creating meaning out of the mess in order to make actual decisions. Nowadays, when we look at the way things have changed, we do not actually use those old dictionary methods. Every successive generation of machine learning and deep learning, and most recently of Large Language Models LLMs, certainly made it more accurate, yet they are extremely costly to operate, and require an enormous amount of data [12].

The initial systems simply used lexicon. Such methods refer to sentiment dictionaries such as SentiWordNet which assigned words a basic score as either positive or negative [13]. Then VADER came to manage social media such as emojis and capitalization [14]. These techniques are cool in the sense that they are quick and easy to use. But they suffer from not understanding the entire sentence as they just read words individually. This is the reason why they crumble with sarcasm or slang. When the text is everywhere you are just going to have junk answers [15].

The next thing to correct those dictionary issues was supervised machine learning. Rather than merely looking at word lists, scholars began to model sentiment as a classification task, and they now trained models such as SVM and Naive Bayes with data that had been manually labeled. They added feature extraction methods such as TF-IDF or n-grams to make the models to make sense of the text, which actually helped the models to achieve accuracy temporarily ,[16]. The main defect was that these models were manually configured. They simply are not constructed to cognize the deeper meaning or the connection of words in longer and more complex sentences. These models tend to lose track of the actual meaning of what is being said when the text is not short and straightforward [17].

Deep learning has transformed the game as it allows the models to extract features directly out of the raw text rather than human beings extracting them. Although CNNs performed well in short text bits, LSTMs were used in text since they actually address the historical issue of the lack of recall of information in a sequence. Through tracking word sequences and other phenomena such as negation over long sequences, LSTMs began outperforming classical machine learning in virtually every test, particularly on concerned social media data. However, there is a cost on computation. It takes a lot of memory and computing power to train these models due to the large recurrent layers and large datasets [12]. The deep learning sentence analysis pipeline can be visualized in Figure 1.

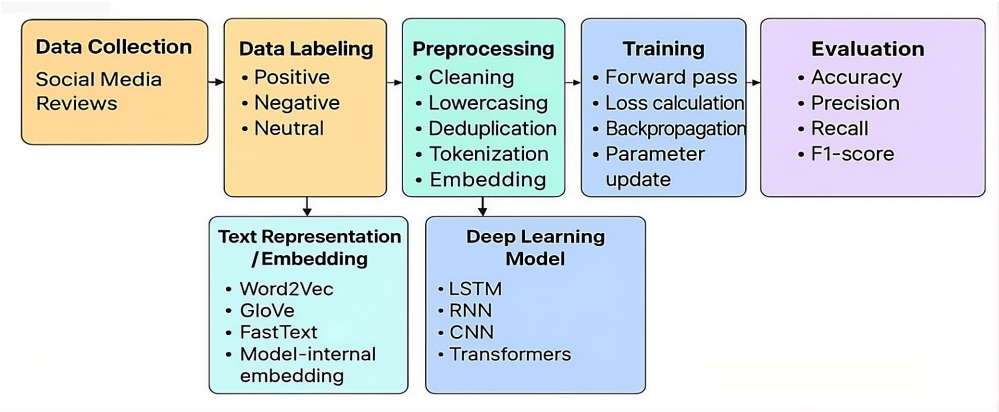

Figure 1. The workflow for sentiment analysis through deep learning.

As the data expanded to millions of files and models became complicated such as LSTMs, it was no longer possible to use only a single machine. To manage the load, researchers were forced to change to distributed systems. Hadoop is suitable in terms of storing large data, however it is slow in terms of training since it has to read and write data to the disk each time it performs a task. Spark is far superior, as data is stored in the RAM, which saves a lot of time in cases when one is repeating the same tasks. This is the primary factor that has led to the importance of assessing LSTMs in the distributed contexts [18]. In simple terms, Hadoop is designed to save data with its storage system HDFS, but Spark is fast, hence, Spark is the one that is likely to be used in regard to the neural networks load [19].

Although previous studies have demonstrated the performance benefits of Apache Spark over Hadoop MapReduce, most have focused primarily on framework-level comparisons or traditional machine learning approaches. Meanwhile, recent sentiment analysis research has consistently shown that LSTM-based models achieve superior classification performance by effectively capturing long-term contextual dependencies in textual data. However, limited attention has been devoted to evaluating LSTM-based sentiment analysis within distributed computing environments while simultaneously investigating the impact of different cluster configurations and increasing data volumes under identical experimental conditions. This research gap motivated the present study to conduct a systematic performance evaluation of Hadoop MapReduce and Apache Spark using an LSTM-based sentiment analysis workflow across multiple dataset sizes and different numbers of computing nodes.

To summarize, the main Contributions we achieved in this study are:

- We perform a controlled experimental comparison between Hadoop MapReduce and Apache Spark for LSTM-based sentiment analysis under identical and reproducible cluster configurations.
- We quantitatively evaluate scalability and execution efficiency across dataset sizes ranging from 200 MB to 1 TB by analyzing execution time, resource utilization, and deriving empirical execution-time models for both Hadoop MapReduce and Apache Spark.
- We analyze the scalability and performance of Hadoop MapReduce and Apache Spark by varying the number of computing nodes, providing practical insights into their suitability for large-scale LSTM-based sentiment analysis.

Here is how this paper is structured. Section 2 reviews the related work. Section 3 describes the methodology we did it and the configurations. Section 4 is everything about the experiments we had done and discovered. Section 5 is a conclusion and a few thoughts about the future, and the final Section 6 contains all the references that we have used.

2. Related Work

Big data analytics now center on distributed systems as datasets are becoming larger, more rapid and increasingly messy. The two tools are Hadoop MapReduce and Apache Spark that virtually are widely deployed in this area. The majority of the preceding studies are aimed at three primary directions: the assessment of distributed execution performance, scaling data pipelines, and finding ways to add machine and deep learning, in particular sentiment analysis on big data. The recent researches have discussed how Spark is transforming batch processing to be much better.

As an example, [20] demonstrated that Spark will help Hive workloads to run much faster. Using the memory-based system of Spark and the storage of Parquet they could process large volumes of data with virtually no delay. They demonstrate in their work that the approach that Spark takes to managing memory is an excellent solution to the slowness that they would have to encounter with Hadoop.

They compared Hadoop and Spark with the HGrid247-DE in another study [21] in order to determine the scaling capabilities of the two systems. The tests indicated that Spark was simply faster each time. This shows clearly that Spark is the solution to the large tasks since it simply has the capacity to manage the load.

Moreover, many people are looking at sentiment analysis in big systems. [22] created a Spark pipeline for large-scale sentiment analysis that integrated preprocessing, TF-IDF feature extraction, and Naïve Bayes classification. It was extremely quick it was approximately 12 times faster than MapReduce and 21 times faster than Mahout. And still with that rapidity the precision was very high.

The same is indicated by benchmark tests. They tested Hadoop and Spark with HiBench in [23] to understand the work of these platforms with such tasks as WordCount and Terasort. Spark was the winner. It was 2 times faster than the WordCount and 14 times faster than Terasort. This merely demonstrates that Spark will be much more suitable for the repetitive tasks that we are doing currently.

These large systems were employed by others to perform certain tasks. As an example, [24] analyzed Twitter data to identify which programming languages are the most talked about. Their tests revealed that Spark is faster

and it does not consume the power of the computer hence it is best when doing real-time work. They also searched transportation service reviews using Hadoop in [25]. With the help of logistic regression and PySpark they processed large amounts of text data and achieved 80 percent accuracy.

More recently, Patri [26] reviewed modern distributed data processing frameworks, including Hadoop MapReduce and Apache Spark, and analyzed their architectural characteristics, scalability, fault tolerance, and suitability for large-scale data analytics. The review highlighted Spark's advantages for iterative and memory-intensive workloads while discussing the evolving role of distributed computing frameworks in large-scale data processing. These observations further support the motivation for evaluating Hadoop MapReduce and Apache Spark in LSTM-based sentiment analysis.

Another attempt that was made by researchers was to perform sentiment analysis in real time. They developed a system in [27] to predict the emotions based on Arabic tweets with the help of Apache Spark to process the stream of data. To determine the best working model, they initially tested CNN, GRU and BiGRU. Eventually, they discovered that the BiGRU model was the winner with an accuracy of 78.71%. This demonstrates that it is possible to use Spark to capture emotions in Arabic literature in real time.

Also began experimenting with deep learning on these distributed systems. In the case study [28], they reviewed the Persian movies with the assistance of CNN and LSTM. They discovered that LSTM performed significantly better than the old techniques such as SVM and autoencoders. This only goes to show that LSTM is good at sentiment analysis since it is excellent at adhering to word sequence within a sentence.

Maharani et al. [29] compared conventional machine learning methods with an LSTM model for sentiment analysis using 10,000 labeled user reviews. The results showed that the LSTM model outperformed traditional machine learning approaches, achieving 92% classification accuracy and a weighted F1-score of 91%, demonstrating its effectiveness in capturing contextual information for sentiment classification.

They demonstrated that it is highly effective to use Hadoop to store the data and Spark to process it in [19]. The accuracy of Logistic Regression was 98.88% when they tested the Twitter sentiment, whereas the Random Forest was rather poor with 48.07%. That is why Spark is the most appropriate option when there is a multitude of sentiment data to work with.

In general, Hadoop is sufficiently fast when dealing with small tasks, but when the task becomes challenging, it becomes slow. Spark has proven to be much faster and more reliable, making it the best option for modern machine and deep learning. Research confirms sentiment analysis in distributed environments, especially Spark is excellent for big data, and LSTM is the best for accuracy. However, most studies haven't tested these powerful LSTM sentiment analysis models on Spark yet. Table 1 summarizes these research directions by outlining the datasets used, the employed frameworks, and the key findings of each study.

Existing studies have advanced two separate areas: distributed frameworks such as Hadoop MapReduce and Apache Spark, and deep learning methods particularly LSTM models for sentiment analysis. Research consistently shows that Spark outperforms MapReduce in iterative and large-scale workloads, while LSTMs deliver higher accuracy than traditional machine learning approaches [11]. However, these findings are hardly ever tested together.

The Previous studies don't simultaneously run the same deep learning model on both systems, and they tend to ignore the changes with varying data sizes and cluster configurations. Although distributed deep learning has received increasing attention in recent years, most studies have focused on Spark-based approaches because its in-memory computing model is well suited to iterative deep learning workloads. Consequently, direct experimental comparisons between Hadoop MapReduce and Apache Spark using the same deep learning workload under identical experimental conditions remain limited, particularly for large-scale sentiment analysis. We fill this gap in this paper by executing the same LSTM sentiment analysis model on Hadoop and Spark. In this manner, we get an actual glimpse of their comparisons in terms of speed, scaling and distributed efficiency.

3. Methodology

We have a distributed architecture experiment on large-scale sentiment analysis with deep learning LSTM algorithm. As it was costly to process millions of texts and also to run LSTM models and as computational overhead

Table 1. Comparative Analysis of Related Work in Distributed Processing and Sentiment Analysis.

Ref	Focus Area	Framework(s)	Dataset	Key Findings
[20]	Hive performance optimization via caching and columnar formats	Spark	Hive benchmark datasets	Spark significantly reduced query execution latency and improved throughput.
[21]	Comparative evaluation of data processing tasks	MapReduce, Spark	3–260 MB Kaggle datasets	Spark outperformed MapReduce in execution speed and scalability across all tasks.
[22]	Parallel sentiment analysis (TF-IDF + Naïve Bayes)	MapReduce, Spark	1 GB Twitter dataset	Spark achieved $\sim 12\times$ speedup over MapReduce and $\sim 21\times$ over Mahout.
[23]	HiBench workload benchmarking (WordCount, Terasort, K-Means)	MapReduce, Spark	Up to 600 GB	Spark achieved up to 2× faster WordCount and up to 14× faster Terasort.
[24]	Counting programming-language tweets	MapReduce, Spark	5.8–43 GB Twitter data	Spark was 10× faster and more efficient for large-scale and near-real-time analytics.
[25]	Large-scale review sentiment analysis using Logistic Regression	MapReduce, Spark	1.88M user reviews	Achieved ∼80% accuracy with efficient distributed processing.
[26]	Review of distributed data processing frameworks	Hadoop, Spark	Multiple benchmark studies	Highlighted Spark's advantages for iterative and scalable big-data processing.
[27]	Real-time Arabic tweet emotion detection (CNN/GRU/BiGRU)	Spark	COVID-time Arabic tweets	BiGRU achieved the highest accuracy among all tested models.
[28]	DL sentiment analysis (CNN, LSTM vs classical ML)	—	∼2,000 Persian movie reviews	LSTM outperformed all classical ML models and CNN.
[29]	DL sentiment analysis (LSTM vs conventional ML)	—	10,000 mobile app reviews	LSTM outperformed conventional ML methods, achieving 92% accuracy.
[19]	Hadoop + Spark sentiment visualization pipeline	HDFS, Spark	Twitter data	Logistic Regression achieved 98.88% vs Random Forest at 48%.

was heavy, we had to have a system that partitioned both storage and processing effectively. We tested that using Hadoop and Apache Spark. Both frameworks use HDFS, so MapReduce and Spark can access the identical data blocks and do not require any modifications.

3.1. Distributed Computing Environment

Three identical nodes were used to remove the hardware variability effect on the performance results. All machines are set up with Ubuntu 20.04 operating system, each equipped with an Intel i7 CPU with 4 cores at 2.90 GHz, 8 GB of RAM, and connected via a local network. The same setting was employed to compare the performance of Spark and Hadoop in distributing workloads, communicating and processing data under the same conditions.

3.1.1. Software Stack In our cluster, we have deployed Hadoop 3.3.6 and Spark 3.4.4 on all machines and use HDFS to address our requirements in terms of storage. Frameworks settings were set to default with only slight adjustments to make the system stable. We also installed Python, TensorFlow and Keras on each node and maintained this software stack the same in all the experiments to ensure that the comparison remained fair and the operation of each framework was without varying external factors. Table 2 contains full setup details.

3.1.2. Hadoop Framework The base layer is Hadoop which takes care of data organization within the cluster. It relies on HDFS to divide datasets into replicated blocks and distributes them in DataNodes. We have a single

Table 2. Cluster Hardware and Software Configuration.

Hardware Configuration	
Number of Nodes	3 (homogeneous worker nodes)
Processor	Intel i7-7820HQ CPUs (2.90 GHz)
CPU Cores	4 cores per node
Memory	8 GB RAM per node
Storage	HDFS distributed storage
Network	Local Ethernet network
Software Configuration	
Operating System	Ubuntu 20.04
Hadoop Version	Hadoop 3.3.6
Spark Version	Apache Spark 3.4.4
Python Version	Python 3
Deep Learning Libraries	TensorFlow + Keras
Execution Mode	Cluster mode
HDFS Replication Factor	2
Scheduler Settings	Default configurations

NameNode which has a record of metadata and access control. This configuration is essentially present to make storage fault-tolerant and distribute data among the nodes.

Hadoop's strength is in its ability to scale up effortlessly and it can process large volumes of data without collapsing. However, it is not really optimal with deep learning where we are required to access the same data repeatedly as it is based on disk storage. Although it is often effective with simple batch jobs, it is actually very slow with deep learning models that require high-speed access to data in multiple training epochs.

3.1.3. MapReduce Processing Model With MapReduce we work with two simplistic phases to process data. The map stage gets the blocks of data simultaneously in parallel on the nodes to produce key-value pairs. These pairs are then being shuffled and sorted and then aggregated in the reduce phase to get the final result. It is a stable setup and can scale to processes such as indexing or counting text in large datasets.

The use of disk I/O is simply too heavy for tasks that require more than one iteration. LSTM, Our model is deep learning that requires continuous scanning and updating parameters. MapReduce is not efficient in this case where the disk is repeatedly being written and read with intermediate results of the algorithm. It is, therefore, a good starting point in batch sentiment analysis but not optimized with neural networks that require such speedy and repeated tasks [30].

3.1.4. Apache Spark Framework Apache Spark helps us to eliminate those disk-bound Hadoop limits. We do this using the datafrmes that allow us to store our data in the memory between steps to avoid continuously hitting the disk. In order to schedule the workflow, Spark generates a dependency using a DAG scheduler to execute our transformations using an in-memory pipeline [31].

We discovered that Spark architecture is actually much faster with our iterative tasks, such as tokenization and embeddings, as well as gradient updates and processing. As Spark is directly linked with HDFS, we are able to continue using the same storage layer in both frameworks. Removing the disk I/O and storing our most frequently accessed data in memory, Spark is our best choice to do LSTM-based sentiment analysis where we need to process the dataset a number of times.

3.2. Dataset Preparation

We chose the IMDB movie review corpus as our main benchmark due to the fact that it provides us with a fair balance of both sentiment labels and a lot of linguistic variation. To check the scaling effect, we had to work on three sizes 200 MB, 1.6 GB, and 1 TB in order to observe how the systems can work with even more data. During our tests, each dataset preserved the original proportion of positive to negative reviews [32].

We divided all our versions of dataset 80/20 to be trained and tested. Our smallest set is a typical research scale workload, and larger scale ones are where we really feel the big data problems such as high memory pressure, disk throughput problems and lots of communication overhead. We might in fact recognize where our frameworks' execution bottlenecks are and evaluate the performance.

3.3. Preprocessing Pipeline

All our versions of the dataset were put through the same preprocessing pipeline. We lowercased the reviews first and removed noise, such as punctuations, tags, and non-alphabetic symbols. Following it, we removed stop words and tokenized the clean text. To complete the last step we constructed our vocabulary at a 3,000 token limit and an out-of-vocabulary marker, as shown in Figure 2.

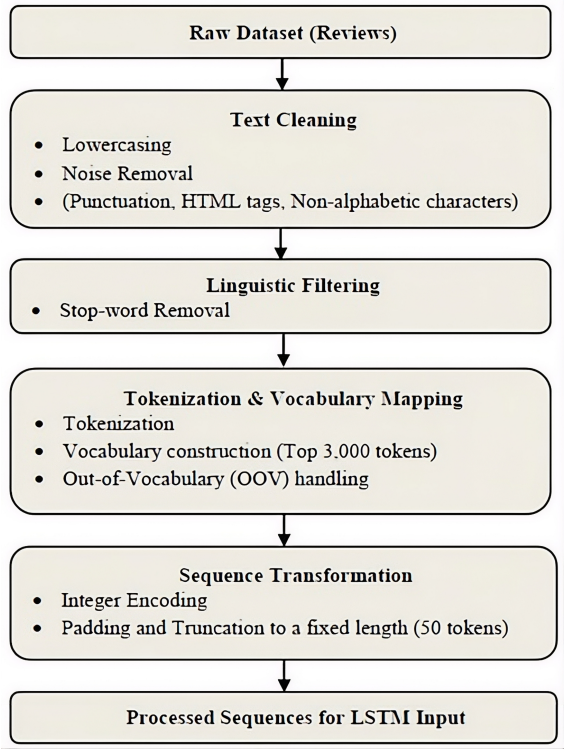

Figure 2. Process of converting text into fixed-length sequences.

After tokenizing, we indexed the reviews to integer codes and used a 50-token padding to ensure that we had equally long input. This we did to ensure that our LSTM input was of the same size. The entire process produces us with numbers that are simple in the memory and yet provide us with context on the sequence we need in our modelling of sequences.

3.4. LSTM Model Implementation

The model was constructed with the help of TensorFlow and Keras libraries and was essentially an attempt to make the model fast without consuming computational power. The padded sequences include a 0.2 dropout rate, an input layer, an LSTM layer with eight units, a dense layer for transformation, and an output layer that uses a sigmoid function for binary classification. As far as training was concerned, we trained locally Adam at 0.001 and binary cross-entropy and we ensured that we met early stopping to avoid overfitting. This architecture is ideal since it gives us consistency in performance, despite the framework we are using, either MapReduce or Spark. We used the same configurations and preprocessing for every run. This enables the comparison to focus on the impact of the distributed computing frameworks rather than differences in the neural network architecture. The full specifications of the model are presented in Table 3.

The LSTM model was trained using TensorFlow and Keras on the original IMDB dataset with the experimental settings described in this study. The resulting model was then used as the pretrained model for all subsequent distributed experiments. Since Hadoop MapReduce follows a disk-based execution model, iterative deep learning training introduces additional disk input and output overhead compared with inference. Therefore, the distributed experiments in this study focus on executing the pretrained LSTM model to evaluate the execution performance of Hadoop MapReduce and Apache Spark under identical experimental conditions. The pretrained LSTM model was made available to all worker nodes before execution. Each distributed task loads the same pretrained LSTM model locally and performs inference on its assigned data partition, after which the prediction results are aggregated to produce the final output.

Table 3. LSTM Model Architecture and Hyperparameters

Component	**Description / Value**
Model	Long Short-Term Memory (LSTM)
Embedding Method	Integer-encoded
Input Sequence Length	50 tokens
Vocabulary Size	3,000 words + OOV token
LSTM Units	8
Dropout Rate	0.2
Dense Layer	Fully connected (ReLU)
Output Layer	Sigmoid (binary)
Loss Function	Binary cross-entropy
Optimizer	Adam (learning rate = 0.001)
Batch Size	16
Maximum Epochs	50
Early Stopping	Patience = 15, min_delta = 0.0001

3.4.1. LSTM Execution on Hadoop MapReduce In our MapReduce system, we had everything partitioned through a job-based workflow which fits its batch design. All data partitions in HDFS were read and passed to a mapper. Which feeds the preprocessed sequences and flows the LSTM parameters across to load sentiment predictions which can be running on all the slaves concurrently. The predictions are then produced as key-value pairs and passed to the shuffle phase, where records are partitioned, sorted, and grouped by key before being forwarded to reducers. The reducers then aggregate the mapper outputs and write the final results to HDFS.

The default Hadoop MapReduce shuffle and sort mechanism was used without modifying the default partitioning or sorting configurations. This approach lets us run the model in parallel on multiple nodes, but it still follows MapReduce's disk-based execution. In our workflow, every stage of mapping, shuffling, and reducing reads and writes intermediate data to disk, which causes delays with large datasets. Since we can't keep the model state in memory between tasks. Therefore, we can run the model correctly, but our results show the limits of using a

deep learning model in the MapReduce framework. Figure 3 shows the workflow of the LSTM model on Hadoop MapReduce.

Hadoop MapReduce isn't suitable for iterative deep learning training. LSTM model require repeated passes over the data and continuous parameter updates, which don't align with MapReduce's disk-based execution model. Thus, in this work, MapReduce is used for distributed inference. Each mapper applies a pretrained LSTM model to its local data partition to generate sentiment predictions, without performing any model updates.

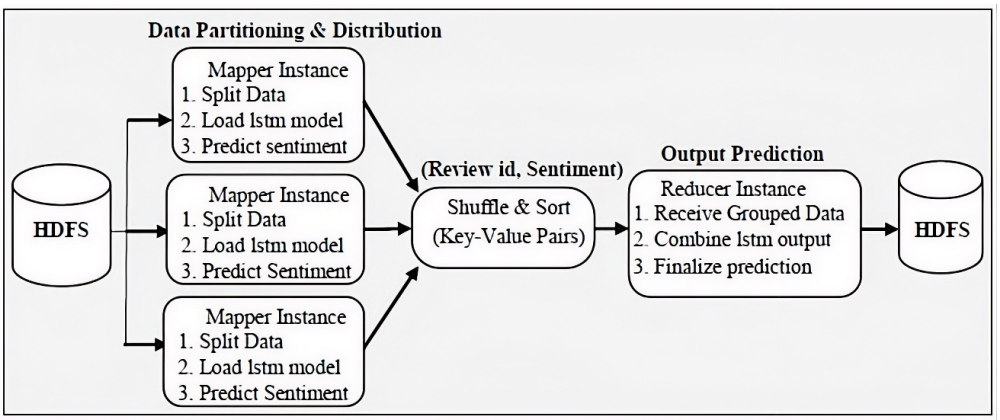

Figure 3. LSTM-Based Sentiment Workflow on Hadoop MapReduce.

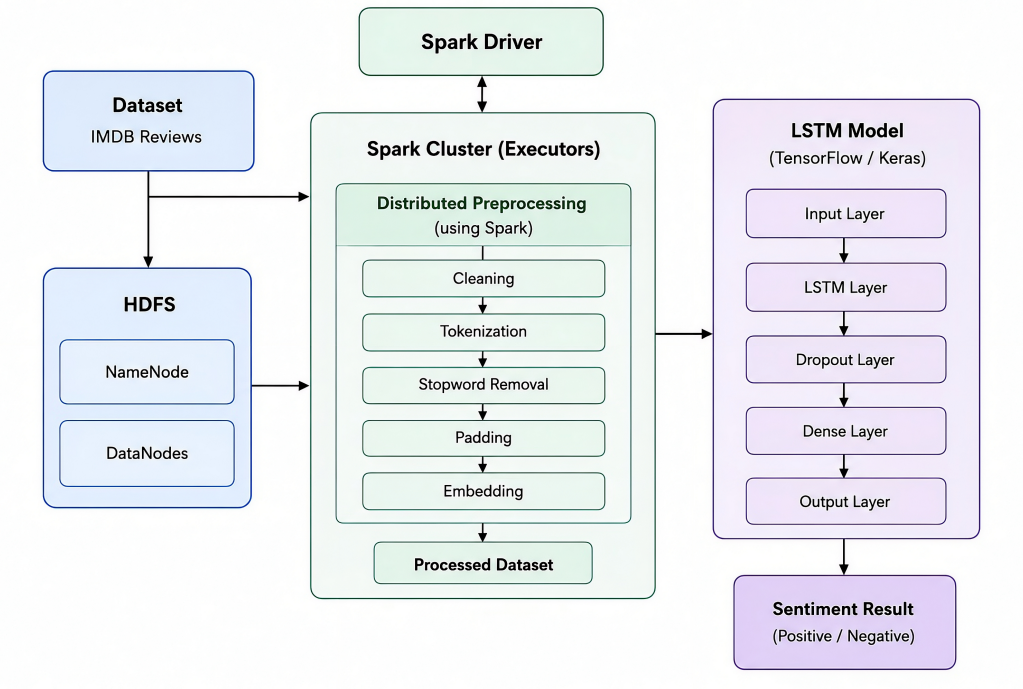

Figure 4. Distributed Pipeline for LSTM-Based Sentiment Analysis Using HDFS and Apache Spark.

3.4.2. LSTM Execution on Apache Spark Spark loads the prepared sequences into distributed DataFrame partitions. The driver constructs a DAG representing batching and output collection. Outputs are collected by

the driver or saved back to HDFS. Figure 4 shows the overall workflow of our proposed method. We first store the dataset in HDFS, where it is distributed across the cluster nodes. The Spark driver then coordinates our preprocessing steps including cleaning, tokenization, stop-word removal, padding, and sequence embedding across the executors. We feed the processed sequences into our LSTM-based sentiment analysis model, which contains an input layer, an LSTM layer, dropout, a dense layer, and an output layer. Our workflow runs in parallel using Spark's DAG engine, allowing us to process data efficiently and generate the final sentiment prediction.

The way Spark handles data in-memory just makes it more effective for how our LSTMs need to repeat tasks, and it's honestly way faster than what we got with MapReduce. In our experiments, we ensured that we used the same tokenizer outputs, data splits, and batches in both frameworks. We did it in this manner to ensure that any holes found in either the runtime or scaling would purely exist due to the engines. In our case, this is what we had to do to ensure that the comparison was controlled and fair.

3.5. Experimental Protocol

We performed all our experiments in the same dataset partitions, preprocessing outcomes, and LSTM configurations. Each execution follows a sequence of steps starting with loading the data from HDFS, preprocessing, and producing the padded sequences, and we run the LSTM sentiment analysis at scale. We didn't modify our preprocessing logic, vocabulary size or tokenization rules. Our hyperparameters such as our batch size, number of epochs and learning rate were also fixed in all tests so we could eliminate any changes due to model settings.

All experiments were carried out following the same execution protocol, with only the dataset size and the number of worker nodes being varied. The pretrained LSTM model and its training configuration were kept unchanged throughout all distributed inference experiments. This ensured that the observed differences were attributable to the distributed computing frameworks rather than changes in the experimental procedure. We used a uniform measurement approach to our research. The time was recorded from when we created a job that was distributed until the final output was written to HDFS. In this manner, we recorded the loading of data, scheduling, overhead of communication and our model inference. Each of the experiments was repeated three times and we took the median to remove any unusual values. To maintain an unbiased evaluation, accuracy was calculated for all dataset sizes. The complete summary of what we measured is in Table 4.

Table 4. Measurement Strategy Used in All Experiments [33]

Measurement	**Description**
Execution Time	Time measured from job submission to final output written to HDFS.
Included Phases	Data loading, scheduling, preprocessing, and LSTM inference.
Number of Runs	Three repetitions for each framework and dataset size.
Reported Time	Median runtime to Reduce Temporary System Variations.
Accuracy Evaluation	Computed on identical test split across all experiments.

3.5.1. Performance Metrics

1. Execution Time
   We relied on this metric to keep track of how long our distributed sentiment analysis workflow really lasted from start to finish. For us, timing includes every step from the initial data loading and task scheduling to the model execution, shuffle operations, and the final output. For each experiment, execution time was computed as in Eq. (1) [8]:

$$T = t_{end} - t_{start} \tag{1}$$

It's the most direct way we found to see the real computational cost for each framework. To figure out the performance gains, we calculated how much Spark reduced this time compared to MapReduce using the logic in Eq. (2) [8]:

$$\text{TimeReduction}(\%) = \frac{T_{MR} - T_{Spark}}{T_{MR}} \times 100 \tag{2}$$

In our formula, T_{MR} and T_{Spark} are the execution times we recorded for MapReduce and Spark, respectively.

2. Classification Accuracy
Even though we mainly cared about how fast the system ran, we still tracked accuracy just to make sure both frameworks were actually doing their job right. Accuracy basically tells us the percentage of sentiment labels we got right in the test set, and we used Eq. (3) [34] to calculate it:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{3}$$

We double-checked all our predictions across the frameworks to be certain that the faster execution times didn't affect how the model actually performed.

3. Scalability
We used scalability tests to see if our frameworks could actually hold up against larger datasets or more nodes without the execution time increasing proportionally. To test this, we compared how long it took to process from 200 MB dataset all the way up to the 1 TB. For the node-level scaling, we looked at the speedup ratio to see how much we actually gained by adding more nodes. We calculated it using Eq. (4) [35]:

$$\text{Speedup} = \frac{T_1}{T_n} \tag{4}$$

Here, T_1 is the execution time on a single node, and T_n is the time on n nodes. This ratio basically tells us if the frameworks are actually making good use of all that distributed parallelism or just wasting resources.

We dealt with the reproducibility of our tests by correcting the random seeds in each step including tokenization and the way we divide the data. Also, it was critical that each of the nodes in our cluster had the same versions of the OS, Hadoop, Spark, Python, TensorFlow and Keras to eliminate version incompatibilities. We wiped the memory of the cluster and removed temp files or cache partitions before we did any experiment. This was done to ensure that no framework would benefit by having leftover data. Lastly, we maintained the same resource allocation policies such that the access to CPU and memory was identical between Spark and MapReduce.

4. Experiments Results

Here we present the empirical analysis of our distributed sentiment analysis workflow. Hadoop MapReduce was used to process large volumes of data, while Spark handled in-memory preprocessing at scale. All experiments were conducted on the same hardware, software, and datasets to ensure that any variations in performance were derived from the differences in their frameworks.

4.1. Execution Results on Hadoop MapReduce

Hadoop MapReduce was experimented with all the dataset sizes in one, two and three node clusters. The resulting execution traces provide a clear indication of the disk-bound characteristic of the framework and how it doesn't exhibit well under heavy workloads.

Table 5 gathers the key performance indicators we have in the process of MapReduce processing including the dataset sizes, number of nodes in the cluster and the total time. As we may observe, we can get that record (200

MB) processed in 350s on a single node. The dataset was scaled to 1.6 GB to achieve a total execution time of 2,800 seconds, which is almost a linear increase in the time of the map and reduce phases.

With the 1TB workload dataset, a large MapReduce run time on the single node is observed (up to 1,750,000 seconds). Although the time taken to execute is 1,567,927 seconds, which is yet a little better than when there are three worker nodes. This is due to disk I/O dominating runtime, limiting the benefits of parallel computation.

We verified from the counters that our runs emitted 602,341 intermediate tokens and 42,311 unique keys for the 200 MB data set. It was our experience that regardless the change of our execution traces, the framework still remained intensively I/O bound with reads and writes during mapping, shuffling, merging and reducing as the primary runtime performance bottlenecks.

Table 5. Observed MapReduce Execution Times

Dataset	Nodes	Total Time (s)
200 MB	1	350
200 MB	2	326
200 MB	3	313
1.6 GB	1	2,800
1.6 GB	2	2,612
1.6 GB	3	2,508
1 TB	1	1,750,000
1 TB	2	1,632,807
1 TB	3	1,567,927

4.2. Execution Results on Apache Spark

To facilitate distributed preprocessing and handle the LSTM model inference, our research utilized Apache Spark as the primary framework. We captured a variety of execution traces ranging from job structure and stage behavior to memory utilization and shuffle cost. Together, these traces provide a system-level view of Spark's in-memory execution model and its impact on overall performance.

4.2.1. Preprocessing Performance We are able to finish the preprocessing within Spark for all data sizes. Our preprocessing included text normalization and tokenization. All the jobs worked perfectly and data were distributed in the executors. Task durations were similar, indicating a balanced load of the tasks. There were no spikes in memory and tasks were not stopped. In general, the preprocessing remained fast regardless of the size of the datasets due to its in-memory caching and pipelined execution based on Spark.

4.2.2. LSTM Execution We used Spark to handle the distributed input pipeline by feeding mini-batches of the preprocessed data into the pretrained LSTM model during distributed inference. The LSTM model was trained locally before the distributed experiments. Our process began with preprocessing and caching the tokenized sequences. Executor metrics collected during the runs indicated stable memory consumption, balanced task distribution and a continuous workload, proving that our data was properly distributed across nodes.

The resulting performance measures including total inference time and final accuracy are summarized in Table 6. In our experiment, we executed the full LSTM with both 200MB and 1.6GB, producing accuracies in the range of 88.7% to 85.8%. A slight decrease in prediction accuracy was observed as the dataset size increased. Since the same pretrained LSTM model was used throughout all distributed experiments without additional training or parameter updates, this variation may be attributed to the increased diversity and complexity of the larger distributed data partitions rather than changes in the model itself.

The obtained accuracy values demonstrate the effectiveness of the adopted LSTM model for sentiment analysis, which is consistent with previous studies highlighting the strong performance of LSTM-based approaches for

sentiment classification [28]. Therefore, the observed execution performance differences can be attributed primarily to the distributed computing frameworks rather than variations in the predictive capability of the deep learning model.

Table 6. Spark Execution Times and Performance

Dataset	Nodes	Total Time (s)	Accuracy (%)
200 MB	1	285	88.7%
200 MB	2	143	88.5%
200 MB	3	95	88.6%
1.6 GB	1	2700	86.3%
1.6 GB	2	1350	85.9%
1.6 GB	3	900	85.8%
1 TB	1	1,724,940	85.0%
1 TB	2	862,470	85.0%
1 TB	3	574,980	84.6%

4.3. Comparative Scalability Analysis

We made an experiment to analyze the scalability of MapReduce and Spark in terms of execution time with the growth of worker nodes from one to three. In both frameworks, adding more nodes made them faster, but the improvement is not the same. MapReduce remained bottlenecked by disk-based shuffle and merge operations, and Spark accelerated computing thanks to in-memory caching that allowed better parallel processing. Moreover, Spark improved execution time while maintaining similar prediction accuracy under the same model settings.

4.3.1. Execution Time Models From the actual measurements across all dataset sizes, we obtained execution-time models based on the measured results for all dataset sizes. The model parameters were estimated using regression fitting, and the resulting models were validated by comparing the predicted execution times with the corresponding experimental measurements. The coefficients remained stable, with a coefficient of determination above 0.95 for both frameworks. The execution-time models are given in Eq. (5) and Eq. (6):

$$T_M = 2800 \times \frac{S}{1.6} \times \frac{1}{N^{0.1}} \tag{5}$$

$$T_S = \frac{1725 \times S - 60}{N} \tag{6}$$

Where:

- T_M and T_S: are the execution times for MapReduce and Spark, respectively (in seconds).
- S: is the input size (GB).
- N: is the number of worker nodes.

Our MapReduce model uses 2800 not as a random number, but because it is what reflected the processing overhead that we continued to observe when we were shuffling data, merging and we kept encountering those repetitive read-write operations. To calculate this value we simply checked the execution times of the various node configurations and observed the responsiveness of the runtime as we increased the data volume. This coefficient is actually an expression of disk latency and shuffle costs.

With Spark, the 1725 constant does put the significance of in-memory processing being significantly more efficient on the same hardware into perspective. We also included a -60 offset to allow the startup overhead, such as scheduling jobs, getting executors assigned and preparing the distributed input pipeline prior to the actual work. Based on these values, it is rather easy to understand the functionality of Spark and why we are seeing those performance improvements as we increase the number of nodes in the cluster.

In order to be certain about these models we use linear regression on our experimental data. In both of the setups, the R^2 was much greater than 0.95 which basically informs us that the fit is very strong. We also wanted to ensure that our equations were no coincidence with this particular arrangement hence we added 95 percent confidence intervals with each coefficient. That assisted us in ensuring the robustness of the model.

4.3.2. Comparison Interpretation Table 7, which we provide is a clear demonstration that Spark is always faster than MapReduce, regardless of the size of datasets and the manner in which we configured the cluster. An in-memory processing allows us to store the intermediate results of a process across steps, hence the entire pipelining process became significantly smoother. Besides, the runtime decreased in a relatively linear way, as we scaled out by using more worker nodes, which indicates how well the system scales.

On the other hand, MapReduce was almost as unscaled as we had wished. The execution time, as we could see, even after growing the cluster up to three worker nodes, improved by about 10 percent, which demonstrates the disk-intensive operations decrease it. This bottleneck is reflected relatively in the limited speedup of the MapReduce model in the form of the small exponent $N^{0.1}$, which is simply a measure of the bottleneck.

It should be noted that the distributed implementation focuses on executing the LSTM inference pipeline within Hadoop MapReduce and Apache Spark. This design enables a fair evaluation of the execution performance of the two distributed frameworks using the same pretrained model, without introducing variations caused by the model training process.

The performance difference between the two frameworks becomes more apparent as the dataset size increases. As an example, consider the 1 TB dataset, which ran in distributed inference on Spark in our three-node cluster, and then on MapReduce, we took approximately 574,980 seconds and 1,567,927 seconds respectively. This corresponds to a 63 percent reduction in execution time under identical workload and hardware specifications.

Table 7. Execution Time Comparison of MapReduce and Spark

Nodes	Dataset Size	MapReduce (s)	Spark (s)
1	0.2 GB	350	285
1	1.6 GB	2,800	2,700
1	1,000 GB	1,750,000	1,724,940
2	0.2 GB	326	143
2	1.6 GB	2,612	1,350
2	1,000 GB	1,632,807	862,470
3	0.2 GB	313	95
3	1.6 GB	2,508	900
3	1,000 GB	1,567,927	574,980

The fact that we have such close matches in the model execution times, which we got when compared to our actual practical results in Table 7 is a true confirmation that these equations are indeed accurate in describing the performance of both frameworks. These results reinforce the previous experience that Spark is obviously better in terms of scalability and is far more suitable in terms of large-scale iterative processes, whereas the disk-bound nature of MapReduce is a massive limiting factor. This can be further observed in Figure 5, where we have plotted the side-by-side comparison of all dataset sizes and node setups. It is shown in the plots that Spark scales much better and continues to beat MapReduce at each stage.

Figure 5. MapReduce and Spark execution times visualization for all dataset sizes

5. Conclusion Future Work

In this paper, we used distributed deep-learning workflows in Hadoop MapReduce and Apache Spark to develop an end to end package of sentiment analysis at scale. As our experiments prove, the two frameworks were differently characterized by the performance. Thanks to Spark's in-memory processing and task scheduling, we were able to achieve significantly faster execution times on all dataset sizes and cluster configurations. We found that Spark demonstrated a significant advantage in terms of scalability under heavier computation burdens, where it achieved 63% speedup compared to MapReduce for the 1 TB dataset with three nodes. In contrast, the disk-based shuffle and merge stages of MapReduce were a bottleneck on its runtime, so adding more nodes sped up only slightly. On the limited datasets, the performance of LSTM reached 88.7% accuracy and showed considerably good scalability in large-scale inference with stable results for each workload.

Our observations emphasize the significance of selecting a right distributed framework that suits the workload. We found that Spark is better for iterative deep-learning pipelines and large-scale analytics, while MapReduce remains a reliable choice for fault-tolerant batch processing when disk throughput is acceptable. The observed execution times also show how the frameworks scale in different ways and give useful estimates of how long it will take to execute different data sizes with different node numbers.

One limitation of the present study is that the comparison was conducted using a relatively simple LSTM architecture. This design choice was intentional to ensure a fair evaluation of the distributed computing frameworks while minimizing the influence of model complexity on the observed execution performance. In addition, the experiments were conducted on a small academic cluster consisting of three worker nodes. Therefore, the reported results should be interpreted as a comparative evaluation of Hadoop MapReduce and Apache Spark under identical hardware constraints rather than as absolute performance measurements for large-scale production environments.

In our future work, we will further generalize our approach for more complicated NLP tasks like aspect based sentiment analysis and multi-label classification to evaluate its performance. We could investigate real-time flow with streaming data and compare the performance in healthcare, finance, and social media analysis. Investigating multilingual sentiment datasets would extend the applications of distributed NLP systems, which is important for

researchers to more practically understand their real-world performance. Future research will also investigate more advanced deep learning architectures, including deeper LSTM variants, hybrid models, and Transformer-based approaches, distributed neural network training strategies, together with optimization strategies for distributed execution. In addition, extending the evaluation to GPU-enabled and emerging cloud-native distributed computing frameworks, different data partitioning and resource allocation strategies, and additional performance metrics such as memory management efficiency, shuffle performance, garbage collection overhead, cost efficiency, fault tolerance, and energy consumption would provide a more comprehensive assessment of large-scale deep learning systems. We further aim to investigate how distributed deep learning could be used in an energy-efficient and environmentally friendly fashion.

REFERENCES

1. S. Dash, S. K. Shakyawar, M. Sharma, and S. Kaushik, *Big data in healthcare: management, analysis and future prospects*, Journal of Big Data, vol. 6, no. 1, p. 54, 2019. doi: 10.1186/s40537-019-0217-0
2. S. S. Moawad, M. Osman, and A. S. Moussa, *Fuzzified Clustering and Sample Reduction for Intelligent High Performance Distributed Classification of Heterogeneous Uncertain Big Data*, Stat. Optim. Inf. Comput., vol. 13, no. 3, pp. 1162–1191, 2025. doi: 10.19139/soic-2310-5070-2275.
3. S. Saad, and B. Saberi, *Sentiment analysis or opinion mining: A review*, International Journal on Advanced Science, Engineering and Information Technology, vol. 7, no. 5, pp. 1660–1666, 2017. doi: 10.18517/ijaseit.7.5.2137
4. M.-L. Thormann, J. Farchmin, C. Weisser, R.-M. Kruse, B. Säfken, and A. Silbersdorff, *Stock Price Predictions with LSTM Neural Networks and Twitter Sentiment*, Stat. Optim. Inf. Comput., vol. 9, pp. 268–287, 2021. doi: 10.19139/soic-2310-5070-1202.
5. A. A. Hagar, and B. W. Gawali, *Apache Spark and deep learning models for high-performance network intrusion detection using CSE-CIC-IDS2018*, Computational Intelligence and Neuroscience, 2022, Article ID 3131153. doi: 10.1155/2022/3131153
6. G. Divyashree, and K. Naganna, *A Comprehensive Survey on Sentiment Analysis and Opinion Mining on Social Media Using Machine Learning Techniques*, Frontiers in Health Informatics, vol. 13, no. 3, pp. 888–906, 2024. Available: https://healthinformaticsjournal.com/index.php/IJMI/article/view/130/130
7. L. Zhang, S. Wang, and B. Liu, *Deep learning for sentiment analysis: A survey*, Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, vol. 8, no. 4, e1253, 2018. doi: 10.1002/widm.1253
8. K. Shvachko, H. Kuang, S. Radia, and R. Chansler, *The Hadoop Distributed File System*, In Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–10, IEEE, 2010. doi: 10.1109/MSST.2010.5496972
9. M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, *Apache Spark: A unified engine for big data processing*, Communications of the ACM, vol. 59, no. 11, pp. 56–65, 2016. doi: 10.1145/2934664
10. M. Maatallah, M. Fariss, H. Asaidi, and M. Bellouki, *A Lambda Lakehouse Architecture Bridging Streaming and Batch Intelligence in Volatile and Scalable Financial Data Processing*, Stat. Optim. Inf. Comput., vol. 15, no. 3, pp. 2086–2104, 2026. doi: 10.19139/soic-2310-5070-3222.
11. A. Mostafaeipour, A. Jahangard Rafsanjani, M. Ahmadi, and J. Arockia Dhanraj, *Investigating the performance of Hadoop and Spark platforms on machine learning algorithms*, The Journal of Supercomputing, vol. 77, no. 2, pp. 1273–1300, 2021. doi: 10.1007/s11227-020-03328-5
12. J. Cui, Z. Wang, S. B. Ho, and E. Cambria, *Survey on sentiment analysis: evolution of research methods and topics*, Artificial Intelligence Review, vol. 56, no. 8, pp. 8469–8510, 2023. doi: 10.1007/s10462-022-10386-z
13. A. Esuli, and F. Sebastiani, *SENTIWORDNET: A publicly available lexical resource for opinion mining*, In Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC'06), pp. 417–422, European Language Resources Association (ELRA), 2006. Available: https://aclanthology.org/L06-1225/
14. C. J. Hutto, and E. Gilbert, *VADER: A parsimonious rule-based model for sentiment analysis of social media text*, In Proceedings of the International AAAI Conference on Web and Social Media, vol. 8, no. 1, pp. 216–225, 2014. doi: 10.1609/icwsm.v8i1.14550
15. M. Husnain, M. M. S. Missen, N. Akhtar, M. Coustaty, S. Mumtaz, and V. B. S. Prasath, *A systematic study on the role of SentiWordNet in opinion mining*, Frontiers of Computer Science, vol. 15, no. 4, 154614, 2021. doi: 10.1007/s11704-019-9094-0
16. S. M. Ferdous, S. N. E. Newaz, S. B. S. Mugdha, and M. Uddin, *Sentiment Analysis in the Transformative Era of Machine Learning: A Comprehensive Review*, Stat. Optim. Inf. Comput., vol. 13, no. 1, pp. 331–346, 2025. doi: 10.19139/soic-2310-5070-2113.
17. N. Singh, and U. C. Jaiswal, *A detailed sentiment analysis survey based on machine learning techniques*, ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal, vol. 12, no. 1, e29105, 2023. doi: 10.14201/adcaij.29105
18. W. N. C. Thein, and T. Thein, *Sentiment analysis system in big data environment*, Computer Systems Science and Engineering, vol. 33, no. 3, pp. 187–202, 2018. doi: 10.32604/csse.2018.33.187
19. S. D. Shetty, *Sentiment analysis, tweet analysis and visualization on big data using Apache Spark and Hadoop*, IOP Conference Series: Materials Science and Engineering, vol. 1099, no. 1, 012002, 2021. doi: 10.1088/1757-899X/1099/1/012002
20. S. Marru, *Optimizing batch processing techniques of Hive datasets using Apache Spark*, Journal of Computer Science and Technology Studies, vol. 7, no. 7, pp. 399–404, 2025. doi: 10.32996/jcsts.2025.7.7.44
21. F. D. Utami, and F. D. Astuti, *Comparison of Hadoop MapReduce and Apache Spark in Big Data Processing with Hgrid247-DE*, Journal of Applied Informatics and Computing, vol. 8, no. 2, pp. 390–399, 2024. doi: 10.30871/jaic.v8i2.8557
22. M. M. Iqbal, and K. Latha, *A parallel approach for sentiment analysis on social networks using Spark*, Intelligent Automation & Soft Computing, vol. 35, no. 2, pp. 1831–1842, 2023. doi: 10.32604/iasc.2023.029036

23. N. Ahmed, A. L. C. Barczak, T. Susnjak, and M. A. Rashid, *A comprehensive performance analysis of Apache Hadoop and Apache Spark for large scale data sets using HiBench*, Journal of Big Data, vol. 7, no. 1, p. 110, 2020. doi: 10.1186/s40537-02000388-5
24. M. N. Farhan, M. A. Habib, and M. A. Ali, *A study and performance comparison of MapReduce and Apache Spark on Twitter data on Hadoop cluster*, International Journal of Information Technology and Computer Science, vol. 10, no. 7, pp. 61–70, 2018. doi: 10.5815/ijitcs.2018.07.07
25. P. A. Aziz, S. B. N. Ilahi, S. Moka, and A. M. Sajiah, *Penerapan Hadoop untuk Analisis Sentimen Berbasis Big Data pada Ulasan Aplikasi Transportasi Online*, SATESI: Jurnal Sains Teknologi dan Sistem Informasi, vol. 5, no. 1, pp. 51–60, 2025. doi: 10.54259/satesi.v5i1.4051
26. T. Patri, *Distributed Data Processing at Scale: A Review of Key Frameworks and Their Characteristics*, KSII Transactions on Internet and Information Systems, vol. 20, no. 1, pp. 80–105, 2026. doi: 10.3837/tiis.2026.01.005
27. N. Abdelhady, I. E. Elsemman, and T. H. A. Soliman, *A real-time predicting online tool for detection of people's emotions from Arabic tweets based on big data platforms*, Journal of Big Data, vol. 11, no. 1, p. 171, 2024. doi: 10.1186/s40537-024-01035-z
28. K. Dashtipour, M. Gogate, A. Adeel, H. Larijani, and A. Hussain, *Sentiment analysis of Persian movie reviews using deep learning*, Entropy, vol. 23, no. 5, p. 596, 2021. doi: 10.3390/e23050596
29. V. P. Maharani, K. Harvanny, D. Samudra, L. Muthoharoh, A. Satria, and M. C. T. Manullang, *Sentiment Analysis of Mobile Legends App Reviews Using Machine Learning and LSTM-Based Deep Learning Models*, arXiv preprint arXiv:2605.01317, 2026. doi: 10.48550/arXiv.2605.01317
30. M. Kang, and J.-G. Lee, *An experimental analysis of limitations of MapReduce for iterative algorithms on Spark*, Cluster Computing, vol. 20, no. 4, pp. 3593–3604, 2017. doi: 10.1007/s10586-017-1167-y
31. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, *Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing*, In Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12), pp. 15–28, USENIX Association, 2012. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia
32. A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, *Learning word vectors for sentiment analysis*, In Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, pp. 142–150, Association for Computational Linguistics, 2011. Available: https://aclanthology.org/P11-1015/
33. G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, *Densely connected convolutional networks*, In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 2261–2269, 2017. doi: 10.1109/CVPR.2017.243
34. R. Martínez Prentice, M. Villoslada Peciña, R. D. Ward, T. F. Bergamo, K. Joyce, and K. Sepp, *Machine learning classification and accuracy assessment from high-resolution images of coastal wetlands*, Remote Sensing, vol. 13, no. 18, p. 3669, 2021. doi: 10.3390/rs13183669
35. V. Kumar, and A. Gupta, *Analysis of scalability of parallel algorithms and architectures: A survey*, In Proceedings of the 5th International Conference on Supercomputing (ICS '91), pp. 396–405, 1991. doi: 10.1145/109025.109118