An Empirical Evaluation of the Efficacy of an ADMM Classification Technique in Modern Spark

Phillip Gregory
pgrego10@students.kennesaw.edu
Kennesaw State University
Marietta, Georgia, USA

Colin Pittman
cpittm24@students.kennesaw.edu
Kennesaw State University
Marietta, Georgia, USA

Amonn Brewer
abrewe47@students.kennesaw.edu
Kennesaw State University
Marietta, Georgia, USA

Jennifer Felton
jfelto14@students.kennesaw.edu
Kennesaw State University
Marietta, Georgia, USA

ABSTRACT

The scope of this research project was to revisit the work of Xiaodong Su's 2020 paper, "Efficient Logistic Regression with L2 Regularization using ADMM on Spark," [5] which presents an implementation of the Alternating Direction Method of Multipliers (ADMM) on Apache Spark 2.4 shown to provide a significant improvement over the standard MLLib implementation (L-BFGS) without requiring hyperparameter tuning. We aimed to review its technical efficacy and robustness by comparing it with modern Spark 4.0.0 across the RCV1 and Higgs datasets. Our results demonstrate that ADMM's performance is substantially reduced runtimes in Spark 4.0 compared to Spark 2.4. However, under the constraints of our single-machine distributed environment, the standard L-BFGS solver proved to be faster and more accurate.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning algorithms**; **Classification and regression trees**; **Supervised learning by classification**; • **Theory of computation** → *Distributed algorithms.*

KEYWORDS

ADMM, Alternating Direction Method of Multipliers, Logistic Regression, Apache Spark, Distributed Computing, L2 Regularization, Machine Learning Optimization, MLLib, Convergence Analysis

1 RESEARCH STATEMENT AND CONJECTURE

This research revisits and builds upon Su's 2020 implementation of ADMM for logistic regression with L2 regularization in Apache Spark, which introduced a dynamic penalty parameter to enhance coverage without requiring manual tuning. Although the ADMM-based approach demonstrated scalability and efficiency in Spark 2.4, it has not been thoroughly evaluated against modern Spark MLLib versions or across diverse datasets.

This study focuses on extending and evaluating Su's improved ADMM algorithm in a modern Spark environment by:

- Benchmarking it against current MLLib logistic regression in Spark 4.0.0.
- Validating its adaptability and coverage across varied datasets (RCV1 and HIGGS).

2 RELATED WORK

The scope of our literature review consisted of the previous 10 years to maintain relevant results as they relate to ADMM. A 10-year period was chosen because there has not been a robust series of work to improve the algorithm with respect to big data in the past five years on the topic of ADMM; however, there has been some research on the application of ADMM in various industry contexts. Although there are many papers related to the application of ADMM, the research conducted here is restricted to improving the existing ADMM algorithm and not providing an additional application; therefore, this research attempted to review the most recent advancements in the ADMM algorithm.

In "ADMM-based Scalable Machine Learning on Spark," [3] Sauptik Dhar et al. present a direct implementation of ADMM on Spark that allows the ADMM algorithm to be parallelizable on Spark. This appears to be the first implementation of ADMM in Spark based on the research carried out. ADMM was implemented in Spark by decomposing the objective function,

$$\min_{w\in\mathbb{R}^d}\left\{\frac{1}{2N}\sum_{i=1}^{N}\left(y_i-\sum_{j=1}^{d}x_{ij}w_j\right)^2+\lambda\sum_{j=1}^{d}|w_j|\right\},$$

into two specialized functions. The first function, $f(w)$, was created for smooth data-fitting terms:

$$f(w)=\frac{1}{2N}\sum_{i=1}^{N}(y_i-x_i^Tw)^2.$$

The second function, $g(z)$, was created to handle non-smooth regularization terms:

$$g(z)=\lambda\sum_{j=1}^{d}|z_j|.$$

CS7265 Project, June 03–05, 2025, Marietta, GA

ACM ISBN 555-5-5555-5555-5/25/06
https://doi.org/This.Is.Only.A.Class.Project

The first function was decomposed to allow parallelization where $f(w)$ can be solved independently on each data partition while $g(z)$ can manage the global consensus.

Building directly on this limitation, Su's work in "Efficient Logistic Regression with L2 Regularization using ADMM on Spark" [5] introduced an algorithm that restructures the original ADMM algorithm on Spark for distributed computing. This implementation appears to be a direct improvement on the approach from Sauptik Dhar et al.[3]. The goal is to solve the consensus optimization problem where local model parameters w_j on each data partition must agree with a global parameter z:

$$\min_{w_j,z} \sum_{j=1}^{N} f_j(w_j) + \lambda||z||_2^2 \quad \text{subject to } w_j - z = 0$$

This is accomplished by iteratively minimizing the augmented Lagrangian function for the problem, a process whose individual components are detailed in equations (3), (4), and (5) of Su's paper:

$$L_p(\{w_j\}, z, \{u_j\}) = \sum_{j=1}^{N} f_j(w_j) + \lambda||z||_2^2 + \sum_{j=1}^{N} u_j^T(w_j - z) + \frac{p}{2}\sum_{j=1}^{N} ||w_j - z||_2^2$$

This was accomplished by creating three main classes, ADMMState, ADMMUpdater, and ADMMOptimizer. The ADMMState class stores all variables for each Resilient Distributed Dataset (RDD) partition, and the ADMMUpdater consists of the update logic for each of the variables and implements the stopping criteria, which is based on the primal and dual residuals. Additionally, a penalty parameter, p, was created as a result of this implementation. The penalty parameter controls the consensus between all of the models across partitions. A small penalty parameter will have slower convergence, which will have a weak consensus across all models. This will increase the number of iterations required for convergence, while a large penalty parameter will have the opposite effect. The penalty parameter is dynamically adjusted according to the primal and dual residuals. If the primal residual is greater than the dual residual, it indicates that w and z are far apart. Therefore, p is increased to strengthen consensus among the models. Conversely, if the dual residual is greater than the primal residual, p is decreased to allow greater flexibility between all models; otherwise, p remains unchanged.

Finally, the ADMMOptimizer class orchestrates the distributed execution of the algorithm. It manages the ADMMState and ADMMUpdater in parallel, a process in which each partition first updates its local model variables. These local variables are then aggregated to update the global variables and form the global consensus. While the results show that this implementation successfully implements ADMM on Spark, the main drawback is that the implementation does not address any other aspects of ADMM. The authors state this as a limitation, but one that provides a foundation for further enhancements.

In "Optimizing Logistic Regression with Enhanced Convergence: A Modified ADMM Approach," [2] Al-Zamili et al. introduced a modified ADMM algorithm that incorporates a novel "stability term" alongside standard regularization. The purpose of the regularization term is to prevent overfitting by penalizing large model weights. The stability term serves to prevent large changes in weights between iterations, resulting in improved stability and convergence. This term is defined as $\frac{1}{2\eta}\|x - x^k\|_2^2$ which computes the sum of the squared difference between the weight vectors of the current and previous k iterations of the algorithm. The experimental results showed that the proposed algorithm had an F1 score increase of 20% via a synthetic dataset. Although the proposed algorithm does increase F1, it does not improve the scalability of the algorithm as it relates to big data.

Other research literature, like the work by Qiu et al. in "PSRA-HGADMM: A Communication Efficient Distributed ADMM Algorithm," [4] highlights another avenue for optimizing ADMM; their approach is to indirectly improve the ADMM algorithm by focusing on system-level optimizations rather than algorithmic changes. They identify the communication bottleneck as the primary hindrance to scalability and introduce advanced architectural solutions, such as a PSR-Allreduce model and hierarchical grouping, to improve efficiency. Although their low-level approach demonstrates notable scalability improvements, our project is focused on evaluating the mathematical properties of the ADMM algorithm within the standard Apache Spark framework, not on implementing system-level communication protocols.

3 METHODOLOGY

Algorithms and Implementation

This study evaluated and contrasted two primary logistic regression solvers:

- **ADMM**: Based on the distributed algorithm by Su (2020) [5], which we implemented and tested across both Spark 2.4 and 4.0.0.
- **Spark MLlib (L-BFGS)**: The standard, built-in solver available in Spark 4.0.0's MLlib library.

Our code follows the implementation details described in Su's original paper. During the initial development and learning process, we also referenced the Spark-Optimization-ADMM repository by GitHub user GMarzinotto [1].

Classification Problems

The performance of each algorithm will be evaluated using a consistent benchmark task applied across two distinct datasets. This approach will test the algorithms' effectiveness on both the original problem and a new problem with different data characteristics. The two datasets are:

- **RCV1 (Reuters Corpus Volume 1):** This is the dataset used in the original Su (2020) study, for direct comparison and replication of the baseline results.
- **Higgs Dataset:** This features different data distributions and densities, which will be used to gain insights on how our algorithms perform under different conditions.

Execution Environment

All trials are conducted with a containerized environment using Docker. This was to enable team members to rapidly get into an equal environment and be able to export test results during the exploratory and analysis phases of the research. The two environments are configured for Spark 2.4 and Spark 4.0.0.

Due to significant resource constraints and persistent memory and disk issues encountered during initial trials on our containerized Spark environment, only a subset of the RCV1 dataset was used for the majority of our experiments. Specifically, our primary analysis for RCV1 was conducted on approximately 15,000 training samples, which represents about 5% of Su's original 677,399-row dataset. A similar approach was taken for the Higgs dataset, which was also limited to a reduced training set, with both datasets using a testing set of 10,000 samples. This measure was taken to ensure the successful execution of trials without encountering memory cascade failures or cluster instability. Further exploration of system configuration and dataset size may resolve this to a degree; however, our findings suggest that the extent to which we can meaningfully address this to comparably match the original results within the time frame of the project proved to be too inconsequential to pursue.

Evaluation Metrics

Both algorithms are evaluated according to a few key metrics, including:

- **Convergence Speed**: Number of iterations until loss stabilizes within a defined tolerance.
- **Convergence Behavior:** Model norms are used to monitor and analyze convergence behavior.
- **Model Accuracy**: Measured via classification accuracy across tests.
- **Model Sparsity:** Non-zero weights counts are used to compare sparsity.
- **Robustness**: Stability of performance metrics across multiple runs, quantified via standard deviation.

4 EXPERIMENTAL DESIGN

The experiments are designed to enable fair and reproducible comparison of algorithm performance across environments and datasets. Key elements of the experimental setup are outlined below.

Independent Variables

All possible combinations were made from the following:

- **Algorithm**: ADMM and Spark MLlib
- **Spark Version**: 2.4.0 and 4.0.0
- **Dataset**: RCV1 and Higgs

Each combination defines a unique experimental condition.

Control Variables and Reproducibility

To isolate the effect of the algorithm and environment, we control for:

- **Hyperparameters**: All models are trained with the same regularization strength and maximum iterations.
  - **Regularization:** 0.1
  - **Maximum Iterations:** 100
- **Random Seeds**: Fixed seeds (42) were used for reproducibility across data splits and model initialization.
- **Resource Limits**: Containers are configured with identical CPU and memory allocations.
  - **Partitions:** 5, 8, 10, and 15
  - **Cores:** 4
  - **RAM:** 16 GB limit, 8 GB reserved
- **Preprocessing**: Standardized pipelines are used for normalization, tokenization, and label encoding.

Trial Procedure

Each experimental run follows the same protocol:

(1) Launch Spark environment (2.4 or 4.0.0) in Docker.
(2) Load and preprocess the assigned dataset.
(3) Train the selected algorithm using the defined hyperparameters.
(4) Evaluate performance on a held-out test set.
(5) Log key metrics: convergence history, accuracy, F1, sparsity, and runtime.

Each experiment is repeated 20 times to compute variance and support statistical reliability.

Logging and Analysis

All trials log intermediate and final results to structured output files. Logged metrics include:

- Per-iteration loss and runtime
- Final model accuracy, F1-score, and sparsity

Results are aggregated to compute mean and standard deviation for each configuration. Visualization of convergence curves and summary tables will be used to support final conclusions.

5 RESULTS

This section presents a comparative analysis of the performance between ADMM and L-BFGS across Spark versions 2.4 and 4.0 on the RCV1 dataset. The metrics examined include runtime, convergence iterations, model sparsity (percentage of non-zero weights), and classification accuracy. Each algorithm was tested using partition sizes of 5, 8, 10, and 15 to examine scaling behavior.

Runtime and Accuracy. The first set of experiments evaluates the trade-off between runtime and accuracy across different partition counts, as shown in Figure 1.

- L-BFGS completed in 2–11 seconds across all Spark versions and partition settings, representing the fastest performance across all configurations.
- ADMM under Spark 2.4 ranged from 132–290 seconds across partition counts, with 5 partitions showing the longest runtimes (259–290 seconds) and 10 partitions showing the shortest (129–146 seconds).
- ADMM under Spark 4.0 demonstrated consistently faster performance than Spark 2.4, with runtimes ranging from 85–170 seconds across all partition configurations.
- L-BFGS maintained an accuracy of 94.69% across all Spark versions and partition counts.
- L-BFGS and ADMM achieved approximately 60–64% accuracy on the Higgs dataset, significantly lower than RCV1 performance.
- ADMM under Spark 2.4 achieved accuracy between 92.08% and 92.34% on RCV1, with minimal variation across partition counts.
- ADMM under Spark 4.0 showed similar accuracy performance to Spark 2.4, ranging from 92.08% to 92.34% on RCV1.

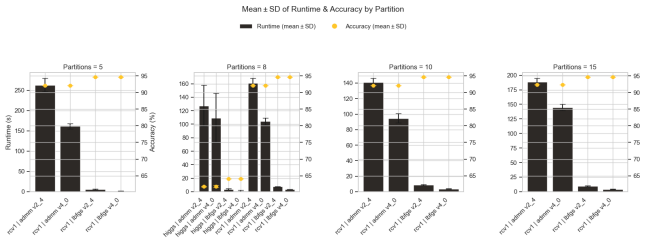

Figure 1: Runtime and Accuracy of Algorithms across Spark Versions and Partition Counts.

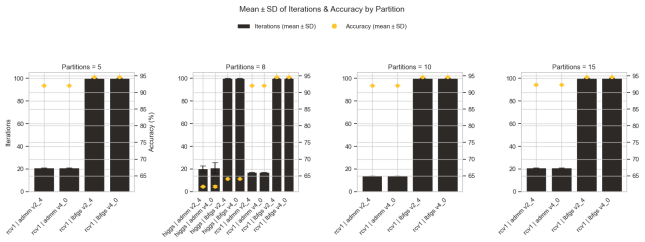

Figure 2: Iterations and Accuracy of Algorithms across Spark Versions and Partition Counts.

- Spark 4.0 provided substantial runtime improvements for ADMM while maintaining equivalent accuracy to Spark 2.4.
- L-BFGS maintained consistent performance characteristics across both Spark 2.4 and 4.0 environments.
- ADMM runtime performance varied significantly with partition count, showing optimal performance at 10 partitions for Spark 2.4 and 8-10 partitions for Spark 4.0.

Iterations and Accuracy. The second set of experiments examines the number of iterations required for each algorithm to complete, alongside accuracy, as shown in Figure 2.

- L-BFGS reached the maximum of 100 iterations in every run across all Spark versions and partition settings while maintaining stable accuracy with a minimal SD.
- ADMM under Spark 2.4 required between 13 and 22 iterations, with fewer iterations observed at higher partition counts while maintaining consistent accuracy on the RCV1 dataset.
- ADMM under Spark 2.4 and Spark 4.0 required between 14 and 21 iterations, with no clear trend in relation to partition size on the RCV1 dataset.
- L-BFGS accuracy remained fixed at 94.69% regardless of iterations.
- ADMM accuracy on Spark 2.4 ranged from 91.10% to 92.41% as iterations decreased with more partitions.
- ADMM accuracy on Spark 4.0 ranged from 92.08% to 92.34%, with accuracy values not clearly linked to iteration count.
- ADMM showed the highest SD for iterations on 8 partitions.
- ADMM and L-BFGS maintained low accuracy on the Higgs dataset while maintaining similar iteration convergence.

Model Sparsity and Accuracy. The third set of experiments explores the number of non-zero weights in each trained model as a proxy for model sparsity, alongside accuracy, as shown in Figure 3.

- L-BFGS and ADMM produced 100 percent (39,378) non-zero weights in every run across all partition counts and both Spark versions for the Higgs dataset.
- L-BFGS and ADMM under Spark 2.4 produced 83.3 percent non-zero weights in every run acorss all partition counts in both versions of Spark for the RCV1 dataset.
- No clear trend with respect to non-zero weights and accuracy.

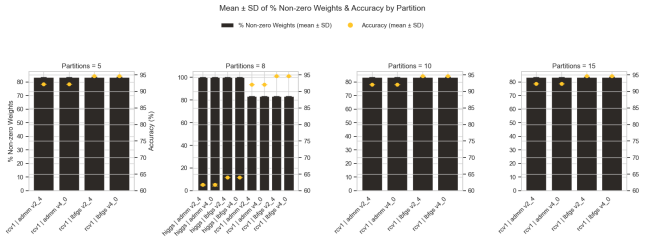

Figure 3: Model Sparsity and Accuracy of Algorithms across Spark Versions and Partition Counts.

6 COMPARISON

Across all evaluation dimensions—runtime, accuracy, iteration count, and model sparsity—L-BFGS and ADMM displayed distinct performance characteristics that appear to be influenced by the underlying Spark version.

Runtime

L-BFGS was consistently faster than ADMM by a wide margin. Regardless of partition size or Spark version, L-BFGS completed in under 11 seconds. In contrast, ADMM runtimes varied significantly:

- Under Spark 2.4, ADMM runtime ranged from 132 to 290 seconds.
- Under Spark 4.0, runtime improved substantially to between 85 and 170 seconds.

This demonstrates that Spark 4.0 provided meaningful performance improvements for the ADMM implementation, though L-BFGS maintained its significant speed advantage.

Accuracy

L-BFGS achieved the highest and most stable accuracy at 94.69% across all configurations. ADMM's accuracy was lower and more sensitive to partitioning:

- Spark 2.4 ADMM achieved up to 92.41%, improving slightly with more partitions.
- Spark 4.0 ADMM produced similar but slightly lower accuracy values, ranging from 92.08% to 92.34%.

L-BFGS offered a 2–3 percentage point advantage in classification accuracy while maintaining stability across versions.

Iterations

L-BFGS always ran for the maximum 100 iterations, showing no early stopping behavior. ADMM required far fewer iterations:

- Spark 2.4: 13–22 iterations.
- Spark 4.0: 14–21 iterations.

While ADMM converged in fewer steps, the higher overall runtime suggests that per-iteration cost or overhead was significantly greater, especially in Spark 4.0.

Model Sparsity

L-BFGS and ADMM matched sparsity performance with respect to the Higgs and RCV1 datasets across all partitions and Spark versions. The identical sparsity levels achieved by L-BFGS and ADMM on both the Higgs and RCV1 datasets—regardless of partition count or Spark version—demonstrate that the induced feature-selection pattern is governed by the data and regularization criteria rather than the choice of solver or distributed execution settings.

Scale, Infrastructure, and Performance Implications

The observed performance discrepancy between ADMM and L-BFGS from Su's original paper, considering our implementation, suggests both scale- and version-dependent behavior in the performance of ADMM. With only 15,000 training samples (~95 % smaller than Su's 677 399-row dataset), the communication overhead of reaching consensus across partitions outweighs the benefit of parallel updates at this reduced scale. However, the substantial runtime improvements observed in Spark 4.0 (85-170 seconds vs. 132-290 seconds in Spark 2.4) demonstrate that modern Spark optimizations can significantly enhance ADMM performance.

The extent of our exploration involved testing both single-worker and multi-worker configurations to evaluate ADMM's distributed processing advantages. Our primary analysis revolved around the single-node setup. The reason for this is because the multi-worker results showed only modest performance improvements, with some of the successful runs showing ADMM achieving speedups of 1.34-1.50× for most partition configurations; however, these marginal gains came at the cost of significant operational complexity, including memory cascade failures, shuffle data corruption, and frequent cluster instability on our single-machine Docker environment. This was compounded by the memory demand of increasing worker counts, which quickly outpaced our best available hardware for which we could rely on for testing.

The runtime improvements in Spark 4.0 combined with the substantial technical overhead suggest that both larger datasets and significant system resources, akin to a genuine multi-node setup, may be required to realize the full benefits of ADMM, though modern Spark versions show promise for better ADMM performance.

7 CONCLUSION AND FUTURE WORK

While some of our performance results deviate from Su's original research, analyzing the nuances of these differences provided valuable insights. L-BFGS consistently achieved superior runtime and accuracy, ADMM showed competitive convergence in fewer iterations. Notably, ADMM demonstrated substantial runtime improvements under Spark 4.0, with execution times decreasing from 132-290 seconds in Spark 2.4 to 85-170 seconds in Spark 4.0, while maintaining equivalent accuracy and model sparsity characteristics.

Implementing Docker added significant complexity to the project, especially given the team's limited prior experience with containerization and distributed system configuration. A considerable amount of time was spent troubleshooting Spark resource management, dependency compatibility, and environment reproducibility. Despite these challenges, the team successfully built a dual-version Spark environment and executed ADMM across both platforms, demonstrating functional parity and compatibility.

While resource constraints limited the study to a subset of the full RCV1 dataset, this limitation also highlighted the strength of L-BFGS as a classification tool. The fact that it outperformed ADMM in our experiment suggests the choice between them is about finding the right fit for the use case, and not necessarily that one is a strict improvement.

Although early results highlight meaningful differences in performance between the two algorithms, a full-scale analysis is currently limited by our system constraints. Future work should prioritize resolving technical barriers related to data scale and refining Docker performance to enable more comprehensive benchmarking at scale. This includes developing a clearer and more standardized methodology for launching and configuring Spark within Docker, with particular emphasis on scaling memory and disk resources effectively. Furthermore, more comprehensive documentation and best practices for resource allocation, performance tuning, and distributed workload management to support this project and others using Spark in containerized environments would also be required. Finally, rerunning tests on the complete dataset will be necessary for drawing definitive conclusions about the algorithm's performance at the scale envisioned by the original research.

REFERENCES

[1] 2020. Spark Optimization with ADMM. https://github.com/GMarzinotto/Spark-Optimization-ADMM. Accessed: 2025-07-20.

[2] Ayad Al-Zamili and Ali Aljilawi. 2025. Optimizing Logistic Regression with Enhanced Convergence: A Modified ADMM Approach. *International Journal of Mathematics and Computer Science* 20 (2025), 9–15.

[3] Sutanay Dhar, Chen Yi, Naren Ramakrishnan, and M. Arif Shah. 2015. ADMM based Scalable Machine Learning on Spark. In *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 1174–1182.

[4] Yongwen Qiu, Yongmei Lei, and Guozheng Wang. 2023. PSRA-HGADMM: A Communication Efficient Distributed ADMM Algorithm. In *Proceedings of the 52nd International Conference on Parallel Processing (ICPP '23) (ICPP '23)*. Association for Computing Machinery, New York, NY, USA, 82–91. https://doi.org/10.1145/3605573.3605610

[5] Xiaodong Su. 2020. Efficient Logistic Regression with L2 Regularization using ADMM on Spark. In *Proceedings of the 2020 International Conference on Machine Learning and Big Data Analytics (ICMLBDA)*. 23–28. https://doi.org/10.1145/3409073.3409077