Exploring parallelization of adaboost algorithm on binary classification

By  
Hritika Rai - 24513

AND

Ayesha Nayyer - 25141

**Introduction**

The growth of data in the economic and scientific realms in recent years has brought benefits as well as concerns. Processing and extracting relevant insights from datasets with millions of samples and hundreds of characteristics is becoming an increasingly common challenge for classical machine learning methods. Because of this increase in data volume, effective processing methods must be created in order to manage the size and complexity of contemporary datasets. As a result, there is an increasing need for algorithms that can efficiently process and optimize data, especially in parallel architectures, to quickly extract insights from large datasets.

With the rise of complex datasets, boosting methods like Adaboost become crucial tools for classification tasks. It uses multiple base learners to improve accuracy. Each one of these learners are iterated upon, where the main focus is on correcting errors of the previous classifier by penalizing misclassified instances.

This sequential nature makes it incredibly difficult to parallelize Adaboost. This research project aims to address this challenge by evaluating the efficiency and effectiveness of both serial and parallel implementation of Adaboost on binary classification datasets. Through a comparative analysis of running time and metrics like accuracy, this study will identify the most suitable approach for handling binary classification tasks efficiently.

**Significance of the Project**



The project is important in the sense that it can make AdaBoost classifiers more scalable and efficient for handling big and complex data in various fields. Therefore, as the volume of data increases, the possibility of rapidly analysing huge amounts of information in domains such as finance, healthcare or social media becomes a must. To gain an advantage from AdaBoost during training processes, we could reduce turnaround time for predictive model learning and decision-making in practice.



Also, this project makes an important contribution to current conversations about parallel computing and distributed systems by examining whether parallelism can be used to speed up machine-learning algorithms’ training phase. It’s imperative to recognize their effect on algorithmic performance given widespread use of parallel architecture. Additionally, some insights into the trade-off between computational efficiency and predictive quality might be gained by comparing AdaBoost implemented serially with it being implemented with parallelism.

**Objectives**

The goal of this research are as follows:

1. The aim of this implementation is to develop both serial and parallel versions of the AdaBoost algorithm for binary classification problems.
2. Evaluate these implementations on a binary classification dataset by considering training time and accuracy among other performance metrics.
3. Also, this paper examines computational overhead due to parallelization by identifying possible bottle necks.

**Overview of the Paper**

This paper is organized as follows: In Section 2, we give an extended review of previous work focusing on AdaBoost and parallel computers in machine learning. In Section 3, we discuss the methodology used in our experiments which includes the dataset employed, the implementation details and the evaluation metrics utilized. In Section 4, we present results from our empirical analysis comparing serial versus parallel implementations of AdaBoost across different datasets and experimental conditions. Finally, in Section 5, we draw implications from these findings for future research directions and summarize our findings.

This research intends to provide insights into computational challenges associated with binary classification problems and gives suggestions to improve the efficiency and effectiveness of machine learning algorithms when handling actual problems by systematically comparing serial and parallel implementations of AdaBoost on datasets.

**Literature Review**

The literature review examines the various methods and implementations developed for parallelizing the Adaboost algorithm with a focus on improving its performance through parallel computation. One of such notable methods aforementioned would be P-AdaBoost; a model proposed in [(Merler et al., 2007)](https://www.zotero.org/google-docs/?UFTHOw) aimed at distributing computations effectively over multiple computing nodes while reducing the serial nature of classic Adaboost while also achieving great accuracy despite a shortened initial sequential. The most essential aspect about P-AdaBoost is its capacity to parallelize loop that appears in step 4 of its algorithm by letting the existing computing units work concurrently thereby quickening the combination of instance models into a finished single model.

As we delve into the field of distributed computing frameworks such as MapReduce, several studies apply the MapReduce framework for implementing various boosting algorithms, For instance, [(Palit & Reddy, 2012)](https://www.zotero.org/google-docs/?FS1sZY) use the Map and Reduce functions in the ADABOOST.PL and LOGITBOOST.PL algorithms to process data subsets separately, which create weak classifiers or regression functions before being combined and weighted during the Reduce stage. By using this technique, there is no need for iterative processes which leads to reduced execution times when compared to serial implementations. Studying further into it, the proposal of a parallel Adaboost-BP neural network algorithm in the [(Cao et al., 2016)](https://www.zotero.org/google-docs/?bymkDY) comes up. By harnessing the capabilities of the MapReduce framework, this algorithm uses combine() to process intermediate results locally before beginning the Reduce stage. Experimental results show that the parallelization level achieved by the algorithm presented in this paper is good as all the resources of a distributed system are exploited to enhance the classification performance. In addition, the performance of MapReduce based distributed parallel system greatly improved as compared to traditional single node algorithm architectures. This shows the powerful computing ability of parallel processing.

As discussed in [(Galtier et al., 2007)](https://www.zotero.org/google-docs/?ZmCIRT), JavaSpace framework introduces a virtual shared memory system which is suitable for clusters and grids. By utilizing the master-workers pattern, each process discovers the JavaSpace and the master writes the dataset path and distributes jobs to the workers. The workers initialize their classifiers and enter the computation loop of the AdaBoost algorithm. During each round, the workers train their classifiers, identify the best base learner, and write it in the JavaSpace. The master retrieves the best candidates, compares their errors, and selects the overall best classifier. The process continues with each worker updating their example weights and entering the next round. The master adds the new base learner to the strong classifier aggregate and evaluates it on the training set. The program stops if the targeted error is reached, otherwise it continues with the next round. This distributed approach optimizes resource utilization and enhances scalability in parallel implementations.

Exploring hardware-specific optimizations, [(Chen et al., 2008)](https://www.zotero.org/google-docs/?UEecmS) focuses on utilizing multi-core processors to accelerate AdaBoost-based algorithms. The study analyses the algorithm's parallelism levels and extracts hybrid parallelism strategies to achieve optimal performance on 4-core and 8-core systems, this being 7 times faster than the serial version. By leveraging thread-level parallelism effectively, the proposed approach demonstrates significant speedups in person detection tasks. The paper also provides a methodology and parallelization schemes that can be applied to other applications targeting multi-core processors. The parallelism is classified into coarse-grained TLP, fine-grained TLP, and fine-grained DLP. Coarse-grained parallelization is easy to implement but has a higher load imbalance, while fine-grained parallelization has a better load balance but incurs overheads.

However, in another implementation [(Datta et al., 2020)](https://www.zotero.org/google-docs/?ovlLCK) for the AdaBoost algorithm, the speedup decreases when four cores of CPU are used instead of three cores, which shows the parallel overheads that must have been created due to inter-core communications. The average speedup, on the other hand, is decently increasing, in almost a straight line. This means that the choice of parallelization depends on the specific computation requirements and the size of the code regions.

Another innovative approach is presented in [(Abualkibash et al., 2013)](https://www.zotero.org/google-docs/?ca45BO) where AdaBoost rounds are initiated to find the feature with the minimum error in each group. Five features are selected from each group, and the one with the least minimum error is chosen to update the weight for the next round of the AdaBoost algorithm. By feature grouping and parallel processing,the training time reduces by a factor of five. Furthermore, when the workload is distributed across multiple processing units and web services are employed, both scalability and efficiency improved in parallel implementations.

This literature also delves into other concurrent approaches, such as the one proposed in [(Allende-Cid et al., 2017)](https://www.zotero.org/google-docs/?cJQWu8). In this approach, multiple processors are utilized to train weak learners concurrently, resulting in improved execution speed without compromising on accuracy, particularly notable with large datasets. So, instead of using a single weak learner in each AdaBoost round, the main idea is to use all p processors available to subsample, in a parallel fashion, the original data, while also training several weak learners in parallel. Based on their training accuracy, the ensemble of weak learners are weighted and then updated using the ensemble's output which eliminates the need to explicitly select the best learner. In the classic AdaBoost approach, a single weak learner is trained with a resample of the original data, while in the concurrent approach, multiple resamples are used in parallel. Another study to back this approach is in [(Lazarevic & Obradovic, 2002)](https://www.zotero.org/google-docs/?7Zacu8). Each data point in the training set is assigned to a different classifier. The distribution is updated based on the performance of the mixed hypothesis on the training set and is used to draw samples in subsequent rounds. It is important to note that only the classifiers, not the data examples themselves, are moved within the system. The performance of parallel boosting was then examined in relation to the size of the training data set. Just like [(Allende-Cid et al., 2017)](https://www.zotero.org/google-docs/?fI2mzI), the parallel implementation not only had shorter computation time but also achieved slightly better accuracy in fewer boosting rounds compared to standard boosting, particularly when learning on larger data sets. Similarly, in [(Barczak et al., 2008)](https://www.zotero.org/google-docs/?RltwbO) where in the proposed PSL (Parallel Strong classiﬁers within the same Layer,  each layer contains independently trained nodes, allowing for parallel processing of strong classifiers. This approach, unlike sequential AdaBoost, where training occurs sequentially with a single strong classifier, PSL improves training efficiency by using subsets of the positive set in each node, enhancing algorithm efficiency.

In [(Huang & Shi, 2010)](https://www.zotero.org/google-docs/?U4zIl7), the authors recognize that training classifiers exhibit data parallelism, but in their method, they also partition all obtained features into a series of feature blocks and then feed it to the classifiers.

For multiclass implementations, as discussed in [(Haffner, 2006)](https://www.zotero.org/google-docs/?LNNDEJ), sorting classes based on frequency distribution and dividing them into groups using a round-robin partition can significantly accelerate training times. This parallelization technique distributes the learning process over S processors, resulting in up to S times faster training, especially when dealing with independent 1-vs-other classifiers. Furthermore, studies like [(Bagci & Bai, 2009)](https://www.zotero.org/google-docs/?87H1TZ) explore innovative ways to decrease the computational cost of AdaBoost. By approximating asymptotic weight distributions using Gamma distributions and early weight estimates, Parallel AdaBoost selects weights efficiently without having to wait for sequential outputs at each step. As a result, computational overhead reduces while algorithm efficiency also improves. This is very useful in computationally intensive tasks like face recognition using Gabor wavelets.

Additionally, hybrid programming paradigms like MPI, OpenMP, and Transactional Memory are used in [(Zeng et al., 2011)](https://www.zotero.org/google-docs/?Kx0tMw) to boost parallel processing capabilities. By combining message passing, loop level parallelism, and fine-grained parallelism, the hybrid parallelized AdaBoost algorithm outperformed purely MPI-based approaches by a factor of 6-14%. In [(Luo et al., 2016)](https://www.zotero.org/google-docs/?V2DC49) the lowest weighted error rate was used to select the best weak classifier. To parallelize the process, the subsets of dataset were assigned to different threads of MIC. Each subset's winner then competed in the master thread to determine the overall best weak classifier. In this implementation, Open MP was used to parallelize the program in both CPU and the MIC.

In summary, the literature review showcases a diverse range of parallelization techniques for Adaboost algorithms, including MapReduce frameworks, hardware-specific optimizations for multi-core processors, and innovative approaches in Python frameworks, especially for large dataset scenarios. All of these approaches aim to enhance functioning, expandability, as well as effectiveness of algorithm run under parallelism premises.

**Methodology**

Logic and Architecture of Serial and Parallel AdaBoost Implementations:

1. Inputting Data and Preparation:
   * In both implementations, data is read from a CSV file, resulting in X\_data (features) and y\_data (labels).
2. Calculating error for Weak Classifier:
   * *Serial Implementation*: For each feature ,error and split value are calculated  iteratively within the weak\_classifier function. The best split is chosen based on minimum error.
   * *Parallel Implementation*: The error calculation is parallelized using the joblib library and by distributing the workload across multiple CPU cores for faster computation.
3. Weak Classifier Training:
   * Serial Implementation: The weak\_classifier function sequentially searches for the best split. It then calculates the beta value, which is used to update sample weights later.
   * Parallel Implementation: The weak\_classifier function uses parallel processing to compute errors concurrently which improves efficiency.
4. Sample Weight Update:
   * Serial Implementation: Sample weights are updated sequentially within the update\_weights function.
   * Parallel Implementation: Sample weight are updated in a parallelized manner using joblib and by dividing the workload into smaller chunks to be processed concurrently.
5. Weak Classifier Prediction:
   * Both implementations use the weak\_predict function to make predictions based on the trained weak classifiers.
6. AdaBoost Training Loop:
   * The training loop in both implementations remains the same. It iterated through rounds and updates sample weights based on weak classifiers' performance. These weak classifiers are stored in a list (hlist).
7. Model Evaluation:
   * Serial and Parallel Implementations: Model evaluation function computes predictions using weak classifiers and calculates accuracy, however the parallel version uses parallel processing for prediction aggregation.

**Key Differences:**

* Parallelism: The basic difference is that instead of using a single CPU core, the parallel implementation parallelizes critical sections like error calculation and weight updating so as to optimize performance.
* Efficiency: Parallel implementation is appreciably faster when used on large data sets due to its capability to simultaneously operate on different groups of data items.
* Workload Distribution: In the parallel implementation, however, error calculation and weight updating tasks are broken down into smaller units and executed at once while the serial version performs these tasks subsequently
* Library Usage: The parallel implementation utilizes joblib library for parallel processing for efficient work distribution andcontrol among CPU cores.

**Pseudocode**

*Serial Implementation*

1. Input:
   1. Training dataset X\_train, y\_train
   2. Number of iterations num\_iter
2. Initialize:
   1. Set for all amples
   2. Empty list hlist is initialized to store weak classifiers
3. For each iteration to num\_iter:
   1. Determine which weak classifier that minimizes the weighted error:
      1. Calculate error for every feature and split value using sample weights
      2. Choose the optimal split based on least weighted error
   2. Calculate epsilon the error rate of the weak classifier:

* 1. Calculate alpha the weight of the weak classifier:
  2. Update sample weights:
     1. For each sample
        1. Update as follows:

where is predicted target value

* + 1. Divide by the normalization constant Z to normalize it

* 1. Keep the weak classifier and its parameters (, split feature, split value) in hlist

1. Combine all weak classifiers:
   1. For every sample in the test dataset X\_test:
      1. Calculate the weighted sum of predictions from all weak classifiers:
      2. Based on the on the sign of determine class label
2. Evaluation:
   1. Evaluate the model's accuracy using the combined weak classifiers on the test dataset

*Parallel Implementation*

1. Input:
   1. Training dataset X\_train, y\_train
   2. Number of iterations num\_iter
2. Initialize:
   1. Set for all samples
   2. Initialize empty list hlist to store weak classifiers
3. For each iteration to num\_iter:
   1. Find weak classifier in parallel that minimizes the weighted error:
      1. Distribute feature calculation across multiple cores using parallel processing
      2. Choose the best split based on the minimum weighted error
   2. Calculate epsilon the error rate of the weak classifier:

where N is the number of sample

* 1. Calculate alpha (αt), the weight of the weak classifier:
  2. Update sample weights in parallel:
     1. Distribute weight updates across multiple cores using parallel processing
     2. Normalize updated sample weights concurrently
  3. Store the weak classifier and its parameters (, split feature, split value) in hlist

1. Combine all weak classifiers:
   1. For each sample in the test dataset X\_test:
      1. Calculate the weighted sum of predictions from all weak classifiers:
      2. Predict the class label based on the sign of
2. Evaluation:
   1. Evaluate the model's accuracy using the combined weak classifiers on the test dataset.

A screenshot of a phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

**Execution Time Evaluation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Implementation** | **Number of Classifiers** | **Time** | **Accuracy** |
| Serial | 400 | 4 min 25 seconds | 97% |
| Parallel | 400 | 3 min 7 seconds | 76% |
| Parallel | 55 | 34 seconds | 81% |

Here's an analysis of the results:

1. Time:
   * In the serial execution with 400 classifiers, it took 4 minutes and 20 seconds to complete the task.
   * In the parallel execution with 400 classifiers, the time reduced significantly to 3 minutes and 7 seconds.
   * With a reduced number of classifiers (55), the parallel execution time further decreased to 34 seconds.
2. The reduction in time from serial to parallel execution is due to the concurrent processing of weak classifiers, which speeds up the overall computation.
3. Accuracy:
   * The serial execution with 400 classifiers achieved an accuracy of 97%.
   * Surprisingly, the parallel execution with 400 classifiers resulted in a lower accuracy of 76%. This could be due to various factors such as the specifics of the parallel implementation, data distribution, or how parallelization affected the learning process.
   * However, reducing the number of classifiers to 55 in parallel execution slightly improved the accuracy to 81%.
4. The accuracy can vary based on the number of classifiers, their quality, and how they are combined in the Adaboost ensemble. Having too few classifiers can lead to underfitting, while having too many can lead to overfitting.

Overall, these results highlight the trade-offs between computation time, accuracy, and the number of weak classifiers in an Adaboost implementation. In our implementation, with the same number of classifiers, the parallel code runs 30% faster.

# **A graph of a computer error Description automatically generated with medium confidenceA graph of a graph Description automatically generated with medium confidenceSerial for 400 classifiers**

# **Parallel for 400 classifiers**

A graph of a cpu usage

Description automatically generatedA graph showing a number of data

Description automatically generated with medium confidence

# **Conclusion**

The comparison between synchronous and parallel execution in the AdaBoost implementation yielded several key findings:

1. Efficiency Gains through Parallelization:

We achieved major efficiency gains by parallelizing weak classifier training, weight update and model evaluation procedures. By spreading the load across multiple CPU cores and taking advantage of concurrent execution, we were able to achieve faster convergence and shorter training time.

1. Impact on Execution Time:

Parallel execution reduced training and evaluation time compared to serial execution. The times obtained from the experiments clearly showed that parallelization increased the speed of the entire algorithm.

1. Optimal Number of Classifiers:

It is interesting to note that the best number of classifiers for parallel execution was different from that in the serial implementation. Even though increasing their numbers in the case of serial approach enhanced accuracy, lesser classifiers were found to be more effective in the parallel execution; thereby demonstrating how parallelization can impact on model performance.

The primary goal of this project was the parallel implementation of AdaBoost algorithm, which could reduce the training time when compared to serial implementation. Our current parallel implementation reduces the training time by 30% all the while keeping CPU usage the same as the one in the original serial implementation. However, it came at the cost of reduced accuracy and higher RAM consumption. Although optimization of accuracy and minimizing resource consumption are not at all our priority in this implementation yet there are useful opportunities for further research and improvement in these areas.

**Future** **Development**:

1. Scalibility:

The understanding of the effective large data handling in AdaBoost implementation, running scalability tests using bigger datasets and reveal some indications on the adaptability of the algorithm.

1. Ram Usage:

Research further into possibilities of minimizing parallel implementation of RAM usage.

1. Algorithm Enhancements:

If put togather to the test with different weak learners like decision stumps, decision trees or any other machine learning model types could amplify AdaBoost's robustness and adaptability. Also another approach is integration of regularization techniques for handling overfitting and enhancing generalization capability onto unseen data.

1. Parameter Tuning:

Carrying out an exhaustive analysis by modifying hyper-parameters such as the number of iterations (num\_iter) and learning rate for AdaBoost algorithm has potential to greatly enhance performance.

1. Real-World Application:

Implementing this approach on different real world classification problems like fraud detection, medical diagnosis or sentiment analysis would show its practical usefulness and adaptability.

**References**

Abualkibash, M., ElSayed, A., & Mahmood, A. (2013). Highly Scalable, Parallel and Distributed Adaboost Algorithm Using Light Weight Threads and Web Services on A Network of Multi-Core Machines. *International Journal of Distributed and Parallel Systems*, *4*(3), 29–40. https://doi.org/10.5121/ijdps.2013.4303

Allende-Cid, H., Acuña, D., & Allende, H. (2017). Subsampling the Concurrent AdaBoost Algorithm: An Efficient Approach for Large Datasets. In C. Beltrán-Castañón, I. Nyström, & F. Famili (Eds.), *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications* (Vol. 10125, pp. 318–325). Springer International Publishing. https://doi.org/10.1007/978-3-319-52277-7\_39

Bagci, U., & Bai, L. (2009). *Parallel AdaBoost Algorithm for Gabor Wavelet Selection in Face Recognition* (arXiv:0907.3218). arXiv. http://arxiv.org/abs/0907.3218

Barczak, A. L. C., Johnson, M. J., & Messom, C. H. (2008). Empirical evaluation of a new structure for AdaBoost. *Proceedings of the 2008 ACM Symposium on Applied Computing*, 1764–1765. https://doi.org/10.1145/1363686.1364109

Cao, J., Chen, L., Wang, M., Shi, H., & Tian, Y. (2016). A Parallel Adaboost-Backpropagation Neural Network for Massive Image Dataset Classification. *Scientific Reports*, *6*(1), 38201. https://doi.org/10.1038/srep38201

Chen, Y.-K., Wenlong Li, & Xiaofeng Tong. (2008). Parallelization of AdaBoost algorithm on multi-core processors. *2008 IEEE Workshop on Signal Processing Systems*, 275–280. https://doi.org/10.1109/SIPS.2008.4671775

Datta, D., Agarwal, R., & Evangeline, P. (2020). Performance Enhancement of Customer Segmentation Using a Distributed Python Framework, Ray. *International Journal of Scientific & Technology Research*, *9*, 130–139.

Galtier, V., Pietquin, O., & Vialle, S. (2007). AdaBoost Parallelization on PC Clusters with Virtual Shared Memory for Fast Feature Selection. *2007 IEEE International Conference on Signal Processing and Communications*, 165–168. https://doi.org/10.1109/ICSPC.2007.4728281

Haffner, P. (2006). Scaling large margin classifiers for spoken language understanding. *Speech Communication*, *48*(3–4), 239–261. https://doi.org/10.1016/j.specom.2005.06.008

Huang, Z., & Shi, X. (2010). A distributed parallel AdaBoost algorithm for face detection. *2010 IEEE International Conference on Intelligent Computing and Intelligent Systems*, *1*, 147–150. https://doi.org/10.1109/ICICISYS.2010.5658736

Lazarevic, A., & Obradovic, Z. (2002). Boosting Algorithms for Parallel and Distributed Learning. *Distributed and Parallel Databases*, *11*(2), 203–229. https://doi.org/10.1023/A:1013992203485

Luo, H., Yuan, H., Cheng, S., Li, Y., Yuan, F., & Wei, M. (2016). Parallelization of Adaboost Algorithm on Intel MIC Architecture. *Proceedings of the 2016 International Conference on Artificial Intelligence: Technologies and Applications*. 2016 International Conference on Artificial Intelligence: Technologies and Applications, Bangkok, Thailand. https://doi.org/10.2991/icaita-16.2016.61

Merler, S., Caprile, B., & Furlanello, C. (2007). Parallelizing AdaBoost by weights dynamics. *Computational Statistics & Data Analysis*, *51*(5), 2487–2498. https://doi.org/10.1016/j.csda.2006.09.001

Palit, I., & Reddy, C. K. (2012). Scalable and Parallel Boosting with MapReduce. *IEEE Transactions on Knowledge and Data Engineering*, *24*(10), 1904–1916. https://doi.org/10.1109/TKDE.2011.208

Zeng, K., Tang, Y., & Liu, F. (2011). Parallization of Adaboost Algorithm through Hybrid MPI/OpenMP and Transactional Memory. *2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 94–100. https://doi.org/10.1109/PDP.2011.97