# Parallel Metaheuristic Algorithms for Hyperparameter Search in Logistic Regression

---

**CS239 Project**

Jeryl Salas
Master of Engineering in Artificial Intelligence
College of Engineering
University of the Philippines Diliman
Quezon city, Philippines
jasalas@upd.edu.ph

## Abstract

Hyperparameter tuning has been an integral part of machine learning. The researcher aims to use parallel metaheuristic algorithms using the power of the Nvidia GPU to search the hyperparameter space. The parallel metaheuristic algorithms used were Parallel Genetic Algorithm, Parallel Particle Swarm Optimization, Parallel Ant Colony Optimization, Parallel Variable Neighborhood Search, Parallel Tabu Search, and Parallel Simulated Annealing. Python was used as the main programming language, where it controls the global search strategy and dispatches threads to execute the Logistic Regression training that is programmed in the CUDA C++ language, which is embedded in the Python code using the PyCUDA library. The researcher found out that the Parallel Particle Swarm Optimization was able to have decent evaluation results despite having the fastest runtime and least memory usage among the six algorithms mentioned.

*Keywords: Machine learning, Logistic Regression, Hyperparameter Tuning, Parallel Metaheuristics, CUDA*

## Introduction

Machine learning models have transformed the landscape of artificial intelligence (AI) and have been applied in different fields of study. The proliferation of diverse machine learning models that cater to various data-driven tasks and decision-making needs has been witnessed as well. However, as the demand for sophisticated data-driven research and decision-making grows, so does the challenge of optimizing the performance of these models. Every machine learning pipeline needs hyperparameter tuning, which are parameter settings that influence how the model learns and cannot be tuned during the training phase. These are fixed values that are defined by the programmers and must be pre-tuned before proceeding with the training phase.

The real complexity comes from the fact that each machine learning algorithm has its own set of hyperparameters. The Support Vector Regression (SVR) model alone comprises three hyperparameters: kernel function, regularization constant, and epsilon (tolerance). What if researchers use more sophisticated machine learning algorithms that use more hyperparameters? Because of this, researchers

are forced to use search methods—optimization strategies designed to find the optimal solution to a problem within a given search space—in order to find the optimal hyperparameters for their machine learning models. These hyperparameter combinations are laid out on a multi-dimensional space called the hyperparameter space.

There are several methods for hyperparameter searching. Researchers can perform an exhaustive search where they try all possible combinations, which guarantees that they are able to find the optimal setting but is very time- and space-consuming, as described below in Table 1.

| Hyperparameter Methods | Strengths | Limitations |
|---|---|---|
| Grid search | • Simple. | • Only effective with categorical hyperparameters;<br>• Takes a long time to find hyperparameters. |
| Random search | • Performs parallelization;<br>• More effective than a grid search. | • Does not rely on previous results;<br>• Cannot perform well with conditional hyperparameters. |
| Genetic algorithm | • No need for good initialization;<br>• Performs well with all types of hyperparameters. | • Weak performance for parallelization. |
| Gradient-based techniques | • Quick convergence speed for continuous hyperparameters. | • Detects local optimum;<br>• Supports continuous hyperparameters. |
| Bayesian optimization-Gaussian process | • Quick convergence speed for continuous hyperparameters. | • Weak performance for parallelization. |
| Particle swarm optimization | • Good with parallelization;<br>• Performs well with all types of hyperparameters. | • Needs good initialization. |

*Table 1: Hyperparameter Search Methods (Khoei T. T., 2023)*

Other methods include using advanced optimization algorithms that are designed to tackle complex optimization problems, such as metaheuristics. Metaheuristic algorithms, which are inspired by natural systems, have proven to be highly effective in hyperparameter tuning. Metaheuristic algorithms include the Genetic Algorithm (GA) (Wicaksono, A., & Afif, A., 2018), Particle Swarm Optimization (PSO) (Aguerchi, K., et al., 2024), and Simulated Annealing (SA) (Bhandare, Y. S., et al., 2024). According to Alba, E. et al. (2012), most basic metaheuristics are sequential; therefore, researchers naturally seek ways to parallelize these algorithms in order to reduce search time and find better solutions. This method, known as parallel metaheuristics, combines the fields of metaheuristics and parallel computing.

With the advent of Graphics Processing Units (GPUs) designed for scientific computing and Compute Unified Device Architecture (CUDA), a parallel computing platform and an application programming interface (API) designed to harness the power of GPUs for faster simultaneous computing, this research aims to discover different parallel metaheuristic algorithms that can be used for hyperparameter search in Logistic Regression. The research will use Python as the main programming language, with support from Sklearn, Matplotlib, and Numpy, in building the Logistic Regression model with a defined hyperparameter space. Using the PyCUDA extension as the CUDA API, the researcher will be able to harness the power of both metaheuristic algorithms and parallel computing for hyperparameter search.

## Significance of the Study

The research will encompass the fields of parallel computing, machine learning, and optimization. By investigating the use of parallel metaheuristic strategies, the researcher will be able to efficiently tune machine learning models, such as Logistic Regression models with hyperparameter spaces. The research aims to leverage the computational power of parallel computing as well as the efficiency of the metaheuristic algorithms to identify optimal hyperparameter configurations more effectively. The study not only contributes to the ongoing advancements of parallel computing techniques, but it also sets the stage for the development of more robust machine learning models and efficient optimization strategies that can be applied to a wide range of domains, including healthcare, finance, data science, operations research, and so on. Furthermore, the research aims to serve as a cornerstone for future studies, offering valuable insight and methodologies that promote further exploration and innovation.

## Preliminaries

Before proceeding with the methodology, the methods used in hyperparameter searching and the machine learning model used in this study must be defined. The first part of this study involves implementing the parallel metaheuristic strategy.

There are many metaheuristic algorithms that can be used for searching the hyperparameter space. In this study, the researcher used six metaheuristic algorithms, which are divided into two categories: neighborhood-based and population-based.

Neighborhood-based algorithms typically involve making incremental changes to a solution to explore nearby solutions. The algorithms that fall under this category are Variable Neighborhood Search, Tabu Search, and Simulated Annealing.

- **Variable Neighborhood Search (VNS)**: According to Ripon K. S. N. et al. (2012), this algorithm is based on a systematic change in the neighborhood. It explores increasingly distant neighborhoods of the current solution, jumping from one to a neighboring solution if there is improvement. This allows it to avoid becoming stuck on the local optima as the search distances increase.

- **Tabu Search (TS)**: According to Glover F. et al. (2008), this algorithm emphasizes adaptive memory and responsive exploration. The adaptive memory allows it to efficiently navigate the search space by using the information gathered during the search process. This algorithm uses memory and prioritizes purposeful exploration rather than random choices, where even a bad decision can give valuable insights.

- **Simulated Annealing (SA)**: According to Busetti F. (2001), this algorithm is a random-search technique inspired by the cooling process of metals. Imagine the extensive exploration of SA as a bouncing ball traversing over mountains and valleys. As the temperature decreases (cooling), the explorable space can be more constrained, which is akin to the ball settling into valleys. It employs generating and acceptance distributions, influenced by temperature, to explore and determine whether it should transition into a new state.

Population-based algorithms, on the other hand, focus on maintaining populations and how they evolve over successive iterations. Below are some metaheuristic algorithms that fall under this category:

- **Genetic Algorithm (GA)**: According to Mueller J. P. et al. (2022), this algorithm is inspired by natural evolution by starting with a pool of solutions called a population and generating new generations of solutions through mutation, cross-over, and selection.

- **Particle Swarm Optimization (PSO):** According to Hassanien E. (2017), this algorithm is related to the study of swarms. It starts by initializing a set of potential solutions, and the optimal solution is found by following the best particles. It is associated with artificial life, theories of swarms, and social behaviors.

- **Ant Colony Algorithm (ACO):** According to Maniezzo A., et al. (2004), this algorithm is loosely inspired by the behavior of real ants. Think of ants as a set of computationally concurrent and asynchronous agents moving through states of the problem, which correspond to the partial solutions to the problem. They move by applying stochastic local decision policy, which is influenced by two parameters: trials and attractiveness. After an ant completes the whole solution, it evaluates and modifies the trail values of the components it used in the solution. This is how the ants release pheromones, which serve as information for the ants currently searching for a solution.

There are many ways to parallelize a metaheuristic algorithm, and thus a classification method was designed by Crainic, T. G., and Tolouse, M. (2010) that indicate how the global problem-solving process is controlled, how information is exchanged among processes, and the variety of methods used in the search for solutions.

In Figure 1 below, the researcher considers three dimensions. First, is the global search controlled by a single process or several processes (1C or pC)? Second, can the search threads communicate with each other during the search process, and how much information is exchanged (RS, KS, KC, and C)? Lastly, do the search threads start with the same or different solutions, and do they use the same or different search strategies (SPSS, SPDS, MPSS, and MPDS)?

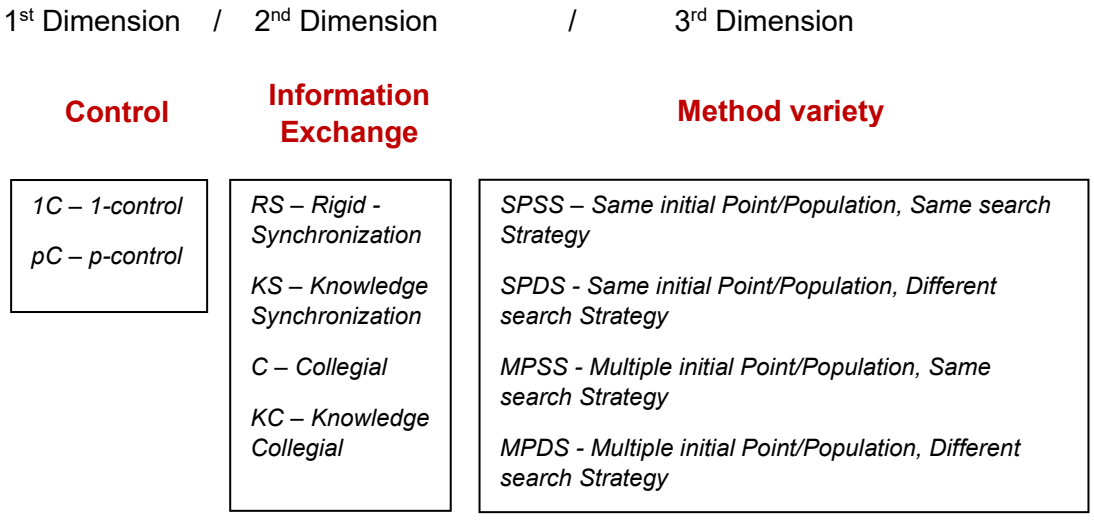


*Figure 1: Visual Illustration of Classification Dimensions for Different Parallel Metaheuristic Strategies*

Crainic, T. G., and Tolouse, M. (2010) states that there are various methods for parallelizing metaheuristics. For example, it can parallelize the execution of inner loops. The fitness function is evaluated using a GA, or the neighbors are evaluated via a neighborhood search. This is the only place in the algorithm where researchers can parallelize processes, since enforcing synchronization on the other steps can yield significant delays, making the parallelization of the algorithm redundant. However, the search space can be partitioned in two different ways.

First, the search space is explicitly divided into regions and assigned to different computational units using domain decomposition. Second, using multi-search, where multiple independent search algorithms are placed with diverse exploration strategies and are executed simultaneously. With the use of parallel computation techniques, the researcher will be able to explore different strategies and algorithms that will, presumably, perform well on hyperparameter searching. There are many parallel strategies that can be used to parallelize the metaheuristic algorithms, like 1C/RS/SPSS, 1C/KS/MPDS, 1C/KS/MPSS, pC/RS/MPSS, pC/KS/SPDS, or pC/C/MPSS. In this study, 1C/RS/SPSS will be used. This global search strategy is controlled by one process called the "master," which dispatches the search threads called "slaves" to execute their assigned task. The slaves are not allowed to communicate with other search threads, and they all start at the same initial point or population, all while performing the same search strategy.

The researcher used Logistic Regression as the machine learning model for the binary classification of a given dataset. Logistic Regression is a machine learning model that predicts whether a given input belongs to a certain class. This is known as a binary classifier, where outputs are either 0s or 1s. This classifier is also used to determine whether an email is spam or ham and whether a tumor is malignant or benign. The sigmoid function is at the core of a Logistic Regression, mapping any linear combination of input features, x, into a value between 0 and 1.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1}$$

After getting the probability output, it will make a classification decision on whether the sample belongs to category 0 or 1, based on a threshold value, in this case, 0.5. The accuracy is computed by the number of samples that the model was able to correctly predict, divided by the total number of samples. The slave threads will train this model, as the researcher wants them to experiment with different hyperparameter combinations and report back on the accuracy results.

The hyperparameters required for Logistic Regression will be used as the hyperparameter search space.

1. **Regularization parameter C**: This parameter controls the strength of the regularization. Smaller values would indicate stronger regulation.
2. **Penalty mode**: This parameter determines what regularization technique the model should use. L1 regularization (lasso) penalizes the absolute value of coefficients, while L2 regularization (ridge) penalizes the squared magnitude of the coefficients.
3. **Learning rate**: This controls the step size of the gradient descent optimization. A high learning rate may cause the model to converge too quickly, while a low learning rate may cause the model to converge slowly. This parameter affects the overfitting and underfitting of the model.
4. **Solver**: This hyperparameter in Logistic Regression determines the algorithm used to optimize the model's parameters. This includes two, namely, Liblinear and saga. **Liblinear** uses the coordinate descent algorithm to optimize the weights and gradients and is known to be effective on smaller datasets with fewer features. Another solver is a variant of stochastic average descent called **saga**. It is known to be more effective on larger and sparse datasets.

In addition, the dataset used for this study is the breast cancer dataset. This is a popular Scikit-learn dataset for binary classification tasks that was utilized for Logistic Regression training. The dataset contains features computed from digitized images of fine needle aspirates (FNA) of breast masses. The dataset used for the Logistic Regression training is the breast cancer dataset, which is a popular dataset in Scikit-learn that is primarily used for binary classification tasks. The dataset contains features computed from digitized images of the FNA of breast masses. The goal is to classify tumors as malignant or benign based on the features that were extracted from the images. The dataset consists of 569 samples and 30 features. The target classes are malignant (0) and benign (1).

| | mean radius | mean texture | mean perimeter | mean area | mean smoothness | \ |
|---|---|---|---|---|---|---|
| 0 | 17.99 | 10.38 | 122.80 | 1001.0 | 0.11840 | |
| 1 | 20.57 | 17.77 | 132.90 | 1326.0 | 0.08474 | |
| 2 | 19.69 | 21.25 | 130.00 | 1203.0 | 0.10960 | |
| 3 | 11.42 | 20.38 | 77.58 | 386.1 | 0.14250 | |
| 4 | 20.29 | 14.34 | 135.10 | 1297.0 | 0.10030 | |
| .. | ... | ... | ... | ... | ... | |
| 564 | 21.56 | 22.39 | 142.00 | 1479.0 | 0.11100 | |
| 565 | 20.13 | 28.25 | 131.20 | 1261.0 | 0.09780 | |
| 566 | 16.60 | 28.08 | 108.30 | 858.1 | 0.08455 | |
| 567 | 20.60 | 29.33 | 140.10 | 1265.0 | 0.11780 | |
| 568 | 7.76 | 24.54 | 47.92 | 181.0 | 0.05263 | |

*Figure 2: Breast Cancer Dataset*

## Limitations of the Study

Due to time constraints, the researcher opted to utilize Logistic Regression as the primary machine learning model. The focus is primarily on investigating the feasibility of employing parallel metaheuristic algorithms for hyperparameter searching in machine learning models. This investigation serves as a precursor to the upcoming capstone project, where the researcher intends to delve into more intricate machine learning models, including ensemble methods that offer a more complex hyperparameter space. The approach to this study involves utilizing the 1C/RS/SPSS parallel strategy, which operates on a master-slave paradigm. While this strategy offers a foundational understanding, there are numerous other parallel strategies available that warrant exploration, which the researcher plans to undertake in a subsequent capstone project to further enrich the theoretical and practical understanding of this hyperparameter tuning strategy.

## Review of Related Literature

Metaheuristics have already been shown to be widely used for hyperparameter optimization in many different machine learning models. For instance, in a study on predicting online news popularity, Wicaksono and Afif (2018) highlighted the importance of determining the best hyperparameters for machine learning methods. They noted that traditional methods like Grid Search can be time-consuming due to their exhaustive nature, as they explore all possible combinations of hyperparameters. To address this challenge, the researchers proposed the use of genetic algorithms as an alternative solution. Through their implementation, they demonstrated that GA could efficiently identify optimal hyperparameters with significantly faster computational time compared to grid search, reporting reductions for various machine learning models: Support Vector Machine by 425.06%, Random Forest by 17%, Adaptive Boosting by 651.06%, and K-Nearest Neighbors by 396.72%.

Aguerchi et al. (2024) demonstrated the effectiveness of employing metaheuristic techniques for optimizing hyperparameters in Convolutional Neural Networks (CNNs) for mammography and breast cancer classification. They utilized the PSO algorithm, achieving success rates of 98.23% and 97.98% on the DDSM and MIAS datasets, respectively, surpassing previous accuracy values. Bhandare and Hajiarbabi (2023) highlighted the significance of hyperparameter tuning in maximizing the accuracy of machine learning models. They proposed SA and GA to automate the process of finding optimal hyperparameters. Ripon et al. (2013) investigated an evolutionary approach employing VNS for solving the unequal area multi-objective facility layout problem (FLP). Their results showed that multi-objective VNS is more effective than traditional multi-objective GA. There are also studies focusing on optimizing hyperparameters for Logistic Regression models. For example, Arafa et al. (2022) evaluated grid search, random search, Bayesian Tree Parzen Estimator (TPE), and SA for Logistic Regression models in cancer classification. They found that Bayesian TPE outperformed other techniques, requiring fewer iterations and less runtime, with the optimized model achieving a test accuracy of 98.2%.

The need to parallelize metaheuristics arises from the significant computational time required for sequential hyperparameter optimization, especially as the complexity and size of datasets grow. By leveraging parallel computing, the efficiency of metaheuristic algorithms can be greatly enhanced. Mostaghim et al. (2008) demonstrated the effectiveness of the parallel metaheuristic approach using a master-slave paradigm. They studied the parallelization of multi-objective optimization algorithms on heterogeneous resources. Their proposed hybrid method, combining multi-objective PSO and binary search methods, efficiently utilized available computing resources, regardless of their speed. The results showed that the new algorithm not only performed well on parallel resources but also outperformed a normal serial run on a single computer.

This study builds on the concept of using parallelized metaheuristic algorithms to achieve efficient hyperparameter optimization. By utilizing a master-slave model, where tasks are distributed among multiple processors, the optimization process can be significantly accelerated, making it suitable for large-scale and computationally intensive machine learning applications.

## Methodology

The researcher will focus on 1-control/Rigid-Synchronization/Same Point, Same Search Strategy (1C/RS/SPSS) for this study and will use Logistic Regression as the chosen machine learning model. The breast cancer dataset from Scikit-learn will be used, where the data is split into an 80% training set and a 20% test set. The features are also standardized by removing the mean and scaling to unit variance for faster convergence of the model. Six metaheuristic algorithms will be used as stated in the preliminaries section, namely: Parallel Genetic Algorithm (PGA), Parallel Particle Swarm Optimization (PPSO), Parallel Ant Colony Optimization (PACO), Parallel Variable Neighborhood Search (PVNS), Parallel Tabu Search (PTS), and Parallel Simulated Annealing (PSA). All of these algorithms will employ the 1C/RS/SPSS strategy, also known as the master-slave strategy.

The global search strategy will be controlled by one process called the master, which will be executed on the Central Processing Unit (CPU) and programmed in Python code. The master will dispatch the threads, also known as slaves, each with their own assigned hyperparameter combination that they will try out on the Logistic Regression model. Each thread on the GPU will perform machine learning training on the training set and will get the training accuracy. The training accuracies are then sent back to the CPU in order to execute the rest of the metaheuristic algorithm. Essentially, the evaluation part of the metaheuristic algorithm is the one that is parallelized since the evaluation portion involves machine learning training, which is computationally exhaustive when done sequentially. Parallelizing this portion of the algorithm will theoretically improve the efficiency of the hyperparameter search. The flowchart, Figure 3, demonstrates how to integrate these parallel metaheuristic search strategies into the machine learning pipelines.

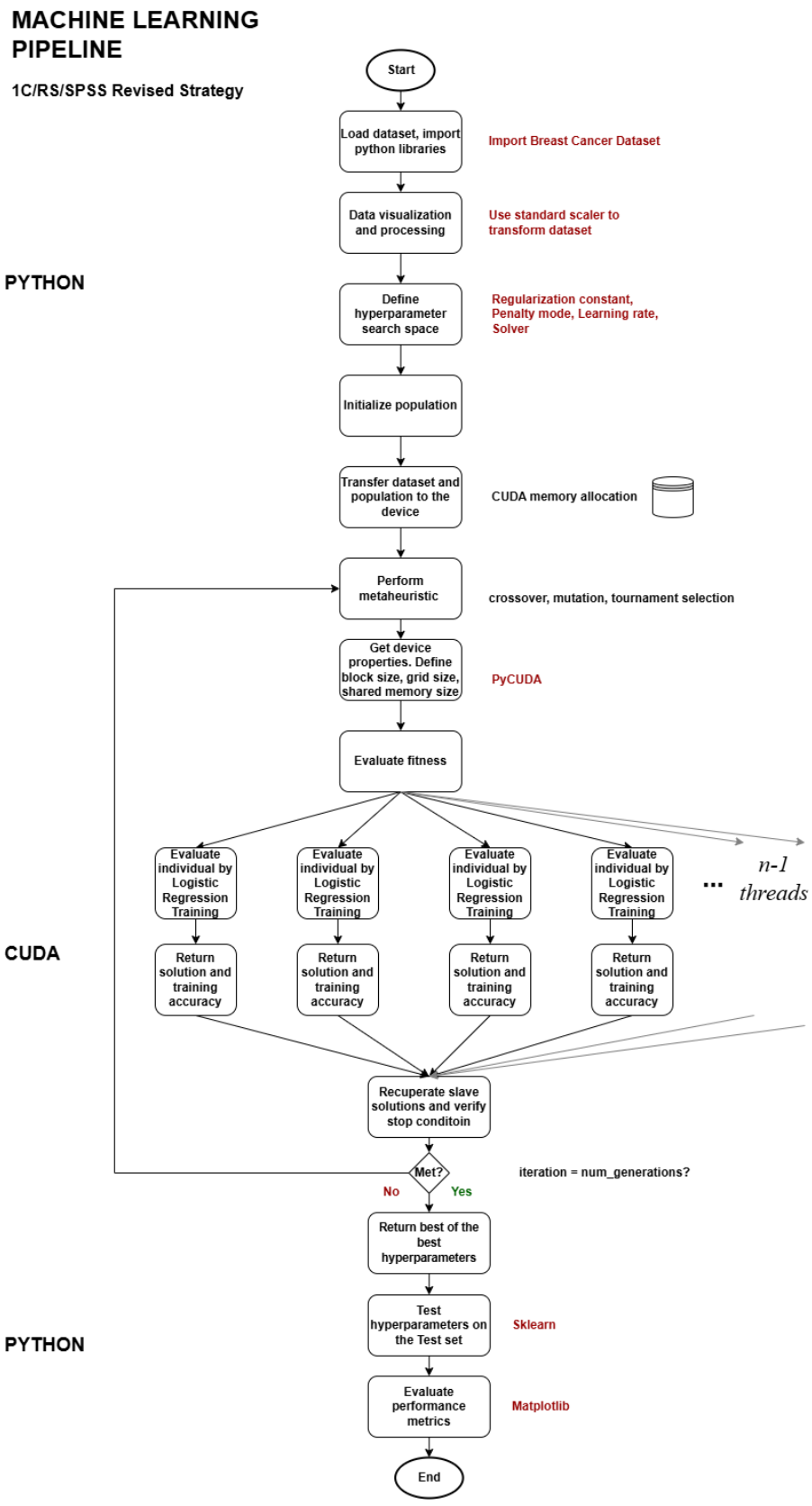


*Figure 3: Machine Learning Pipeline with Parallel Genetic Algorithm 1C/RS/SPSS Hyperparameter Search Strategy*

After finding the best hyperparameter combination, the program will then test the hyperparameters on the test set and then evaluate its performance. The six parallel algorithms will be measured on nine metrics. Six of the metrics will focus on the machine learning part, namely: training set accuracy, testing set accuracy, precision, f1 score, recall, and Matthews Correlation Coefficient (MCC). In determining the efficiency of computation, metrics such as runtime (sec.), memory usage, and number of function evaluations will be used. Talaei Khoei, T., et al. (2023) were able to enumerate the different evaluation metrics that are commonly used in supervised machine learning models.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2}$$

Accuracy represents the correct predictions made by a machine learning model. It is represented in equation (2), where TP is true positive, FP is false positive, TN is true negative, and FN is false negative. Accuracy is basically the number of times the model got it right divided by the total number of samples in the set.

$$Precision = \frac{TP}{TP + FP} x\ 100 \tag{3}$$

Precision, on the other hand, shows the number of true positives divided by the total number of true positives and false positives. It provides insight into the model's ability to correctly identify positive instances.

$$Recall = \frac{TP}{TP + FN} \tag{4}$$

Recall computes the number of positive samples that were correctly classified by the model, divided by the total number of positive samples. This will allow the researcher to know how good the model is at classifying positive samples. It provides insight into the model's ability to identify all relevant positive instances within the dataset.

$$F_1 = \frac{2TP}{2TP + FP + FN} \tag{5}$$

The F1 score is actually derived from precision and recall. A high F1-score would indicate that the model has a good balance between precision and recall, suggesting that it is effective at identifying positive cases while minimizing false positives and false negatives. A low F1-score, on the other hand, indicates that the model is biased on one metric, either precision or recall. This is actually more reliable than accuracy when there is an imbalance in the dataset.

$$MCC = \frac{(TP \times TN) - (FP \times FN)}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{6}$$

Matthews Correlation Coefficient is a performance metric used to evaluate the quality of binary classifications. It is considered one of the best measures for assessing the performance of a classifier since it is highly effective, especially in highly imbalanced datasets. An MCC of 1.0 would indicate a perfect prediction of the samples; 0 would indicate the classifier is not better at classifying than just flipping a coin; and -1 would indicate a total disagreement between the predicted and actual values.

The three other metrics are used to calculate the algorithm's efficiency in computation. The runtime, measured in seconds, indicates how quickly the algorithm converges to a solution. A shorter runtime suggests that the algorithm is efficient in finding a good solution within a reasonable time frame, given that the results are good. Memory usage, measured in megabytes (mb), indicates how efficiently the algorithm utilizes available memory resources. Lower memory usage is generally preferable, as it implies the algorithm can run on systems with limited memory. Lastly, the number of functions evaluated indicates how expensive the algorithms are in terms of computation, how effectively they exploit good solutions, and how well the parallel processes are utilized. The study was carried out on a modest computer with an Nvidia GPU, as the researcher wants to find methods that can be done at home rather than in a sophisticated laboratory.

| Device Name | NVIDIA GeForce RTX 2050 |
|---|---|
| Compute capability | 8.6 |
| Max threads per block | 1024 |
| Max threads per multi-processor | 1536 |
| Max blocks per multi-processor | 1.5 |
| Number of multi-processors | 16 |
| Maximum blocks | 24 |
| Memory Clock Rate (KHz) | 7001000 |

*Table 2: GPU Device Specifications*

The six parallel algorithms are compared to the grid search method (SEQ GS). This method basically tries all hyperparameter combinations sequentially. In this way, it can be said how much time and space the parallel algorithms can save in performing hyperparameter optimization. The hyperparameter search will consist of 10 trials, where in each trial, the number of possible values of C and learning rate hyperparameters is increased by 10, which makes the hyperparameter search space more finely grained.

$$\text{Learning rate} = \text{np.logspace}(-1,0,10\text{x}) \tag{7}$$

$$\text{C} = \text{np.logspace}(-1,1,10\text{x}) \tag{8}$$

Above are equations [7] and [8], where x is the iteration number from 1 to 10. Both C and the learning rate generate an array of numbers that are logarithmically spaced. The researcher intends to make the array sizes larger each iteration to check how the performance of each algorithm changes as the hyperparameter space gets larger. As can be inferred, the SEQ GS's runtime will probably grow exponentially as it is sequential, as well as a steep increase in its memory usage since the functions it has to evaluate will grow.

## Results and Discussion

Figures 4 and 5 illustrate what the terminal looks like as the program is running. In Figure 4, it can be seen what each thread's assigned hyperparameter combination is, as well as the training accuracy that they got. Figure 5, on the other hand, shows the summary of the results, where it shows the best hyperparameter combination that it found, the results of the evaluation of the best hyperparameters on the test set, and also the total runtime and memory size that are used. The results of each algorithm are presented on a table and a line chart for visual analysis.

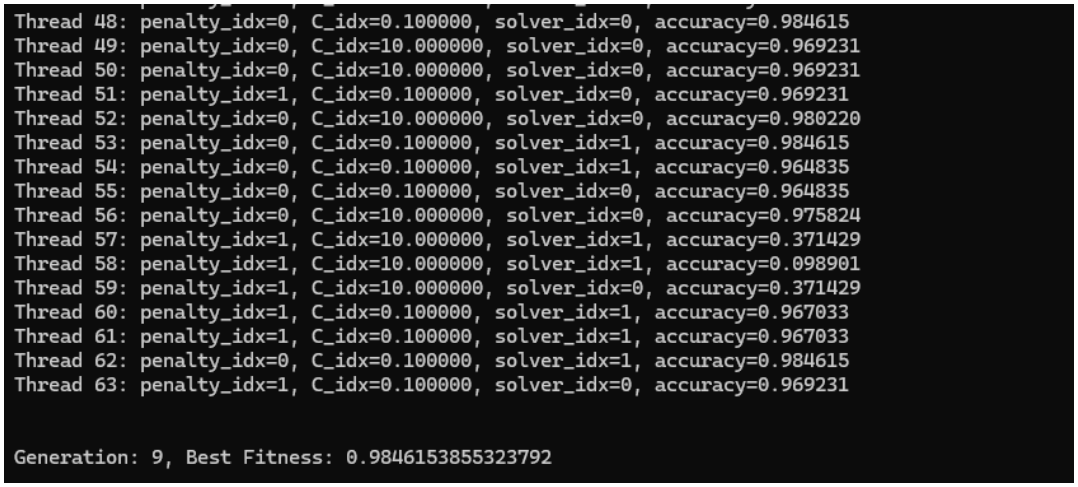


*Figure 4: Parallel Threads Executed in PPSO 1C/RS/SPSS*

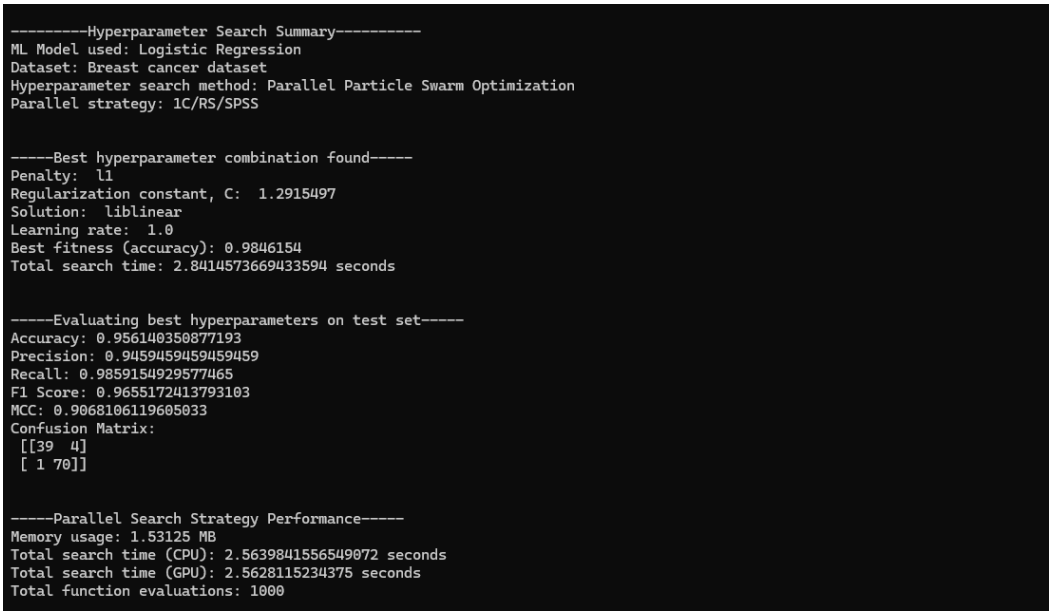


*Figure 5: Summary of Results in PPSO 1C/RS/SPSS*

As shown from Table 3 and Figure 5, SEQ GS increases rapidly as the hyperparameter search space gets larger than expected since the algorithm is done sequentially and involves trying out every single combination, unlike the six algorithms that are only defined by the number of iterations. In the last iteration, the runtime of SEQ GS reached 40 minutes. In Table 3, the PPSO was able to perform the fastest, with each iteration only taking 2.4 seconds.

| RUNTIME (seconds) | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10*x | SEQ GS | PGA | PPSO | PACO | PVNS | PTS | PSA |
| 1 | 22.4569 | 10.5253 | 2.4939 | 20.9120 | 3.5537159 | 15.8780 | 15.7060 |
| 2 | 90.5855 | 10.5611 | 2.4372 | 20.9629 | 3.3724899 | 15.6250 | 15.6110 |
| 3 | 203.4697 | 10.6102 | 2.4757 | 21.1430 | 3.6543965 | 15.6100 | 15.6030 |
| 4 | 363.7686 | 10.5090 | 2.4647 | 21.3311 | 5.2773964 | 15.6190 | 15.5910 |
| 5 | 675.9955 | 10.5111 | 2.4719 | 21.6154 | 3.8488803 | 15.6030 | 15.6150 |
| 6 | 827.6643 | 10.5133 | 2.4858 | 21.9519 | 4.9485662 | 15.6170 | 15.6230 |
| 7 | 1121.4829 | 10.4916 | 2.4693 | 22.3176 | 4.3895621 | 15.6230 | 15.6070 |
| 8 | 1473.3368 | 10.4788 | 2.4457 | 22.7963 | 3.3430972 | 15.6120 | 15.6140 |
| 9 | 1951.7606 | 10.5010 | 2.4550 | 23.4743 | 3.7216 | 15.6240 | 15.5900 |
| 10 | 2442.2025 | 10.5010 | 2.4597 | 23.8877 | 3.7204443 | 15.6160 | 15.6010 |

*Table 3: Runtime Table Measured in Seconds*

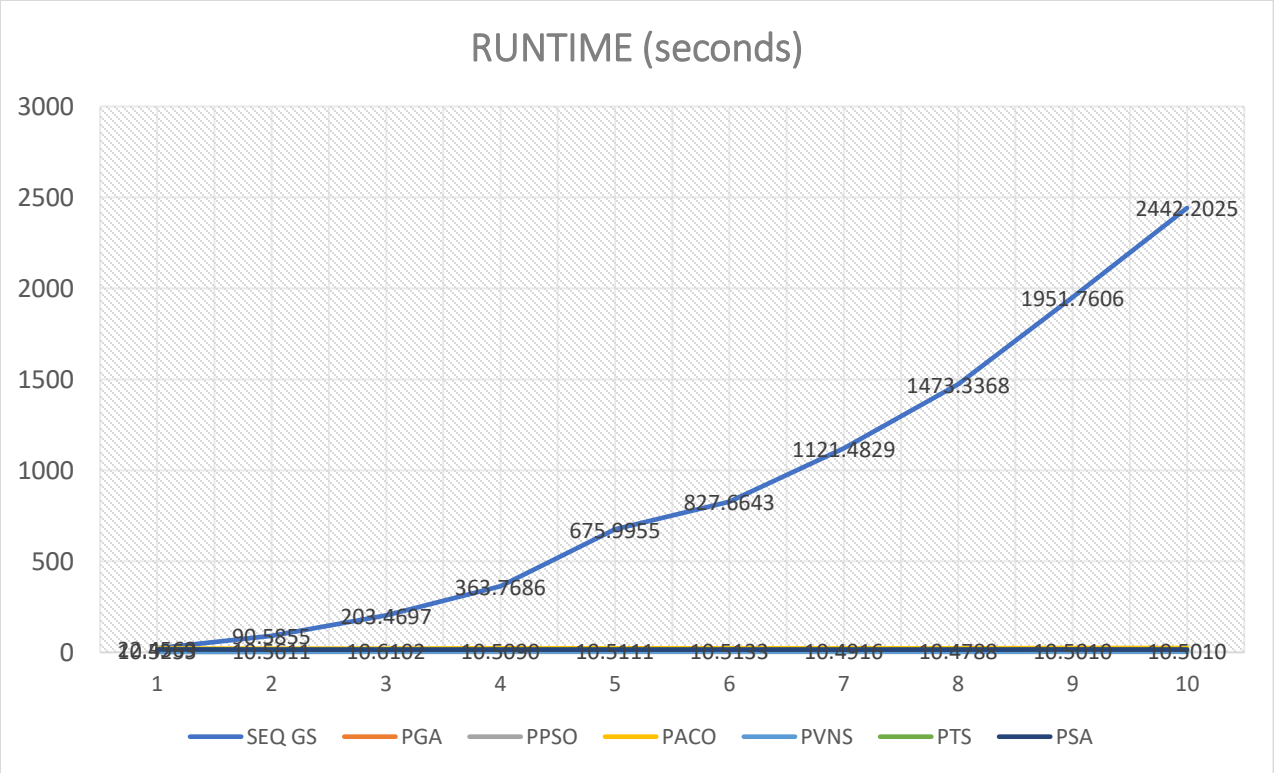


*Figure 5: Runtime Line Chart*

As shown in Table 4 and Figure 6, the training accuracy of SEQ GS is quite low compared to the rest of the parallel algorithms, and it is also the most inconsistent one in the bunch. As visualized in Figure 6, the accuracies of PACO and PGA seem to oscillate while the rest of the algorithms are more stabilized.

| TRAINING ACCURACY | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10*x | SEQ GS | PGA | PPSO | PACO | PVNS | PTS | PSA |
| 1 | 0.967 | 0.9846 | 0.9846 | 0.9868 | 0.9846 | 0.9846 | 0.9846 |
| 2 | 0.9758 | 0.9868 | 0.9846 | 0.9846 | 0.9846 | 0.9846 | 0.9846 |
| 3 | 0.967 | 0.9846 | 0.9846 | 0.9868 | 0.9846 | 0.9846 | 0.9846 |
| 4 | 0.967 | 0.9868 | 0.9846 | 0.9868 | 0.9846 | 0.9846 | 0.9846 |
| 5 | 0.9736 | 0.9846 | 0.9846 | 0.9846 | 0.9846 | 0.9846 | 0.9846 |
| 6 | 0.967 | 0.9868 | 0.9846 | 0.9846 | 0.9846 | 0.9846 | 0.9846 |
| 7 | 0.978 | 0.9846 | 0.9846 | 0.9846 | 0.9846 | 0.9846 | 0.9846 |
| 8 | 0.9736 | 0.9868 | 0.9846 | 0.9846 | 0.9846 | 0.9846 | 0.9846 |
| 9 | 0.9692 | 0.9868 | 0.9846 | 0.9868 | 0.9846 | 0.9846 | 0.9846 |
| 10 | 0.9714 | 0.9868 | 0.9846 | 0.9846 | 0.9846 | 0.9846 | 0.9846 |

*Table 4: Training Accuracy Table*

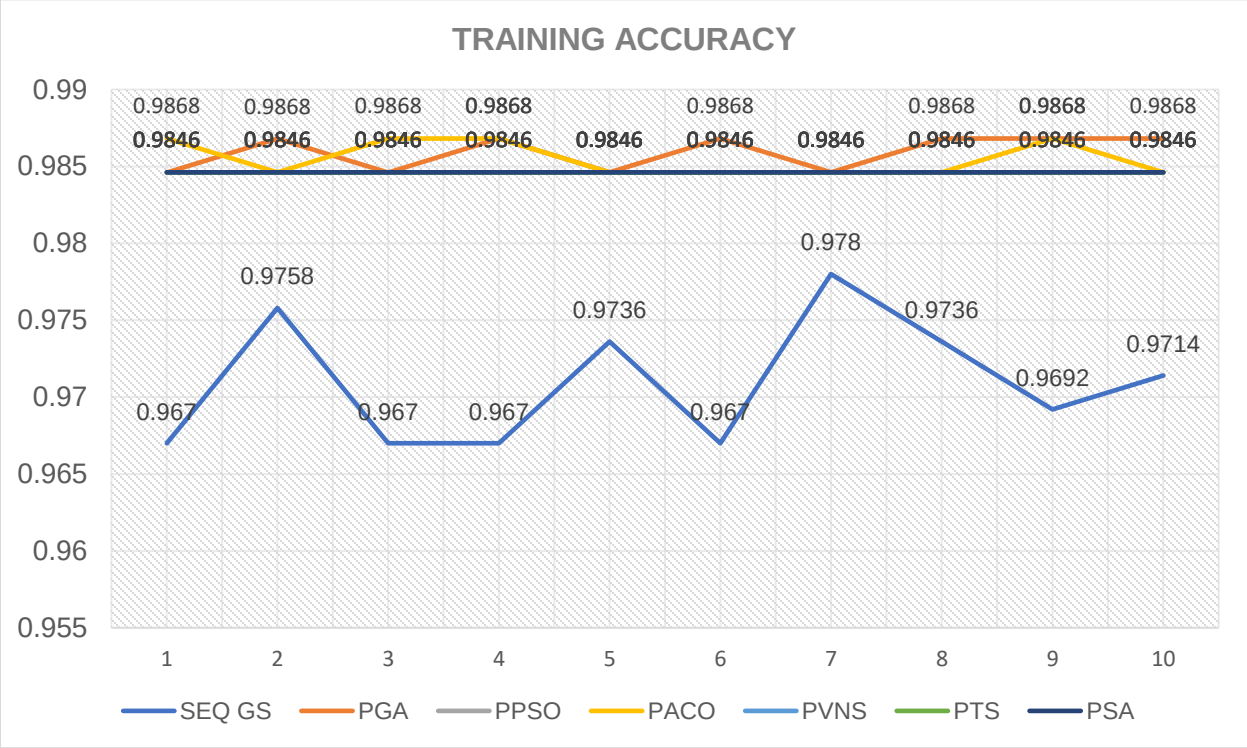


*Figure 6: Training Accuracy Line Chart*

For the testing accuracy shown in Table 5 and visualized in Figure 7, the SEQ GS was able to get the highest testing accuracy as expected since it was able to search the entire space. Although the parallel algorithms are not that far behind, PVNS has a peak accuracy of 0.9846. It seems that the parallel algorithms are right around the range of 0.95 and 0.98 test accuracy.

| TESTING ACCURACY | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10*x | SEQ GS | PGA | PPSO | PACO | PVNS | PTS | PSA |
| 1 | 0.9825 | 0.9649 | 0.9649 | 0.9649 | 0.9561 | 0.9474 | 0.9649 |
| 2 | 0.9912 | 0.9737 | 0.9649 | 0.9561 | 0.9649 | 0.9649 | 0.9649 |
| 3 | 0.9912 | 0.9649 | 0.9649 | 0.9561 | 0.9649 | 0.9649 | 0.9649 |
| 4 | 0.9912 | 0.9649 | 0.9561 | 0.9561 | 0.9561 | 0.9561 | 0.9561 |
| 5 | 0.9912 | 0.9649 | 0.9649 | 0.9561 | 0.9561 | 0.9649 | 0.9649 |
| 6 | 0.9912 | 0.9649 | 0.9649 | 0.9649 | 0.9649 | 0.9561 | 0.9561 |
| 7 | 0.9912 | 0.9561 | 0.9561 | 0.9649 | 0.9649 | 0.9649 | 0.9561 |
| 8 | 0.9912 | 0.9561 | 0.9649 | 0.9649 | 0.9649 | 0.9561 | 0.9561 |
| 9 | 1 | 0.9649 | 0.9649 | 0.9649 | 0.9846 | 0.9561 | 0.9561 |
| 10 | 0.9912 | 0.9649 | 0.9649 | 0.9561 | 0.9649 | 0.9561 | 0.9649 |

*Table 5: Testing Accuracy Table*

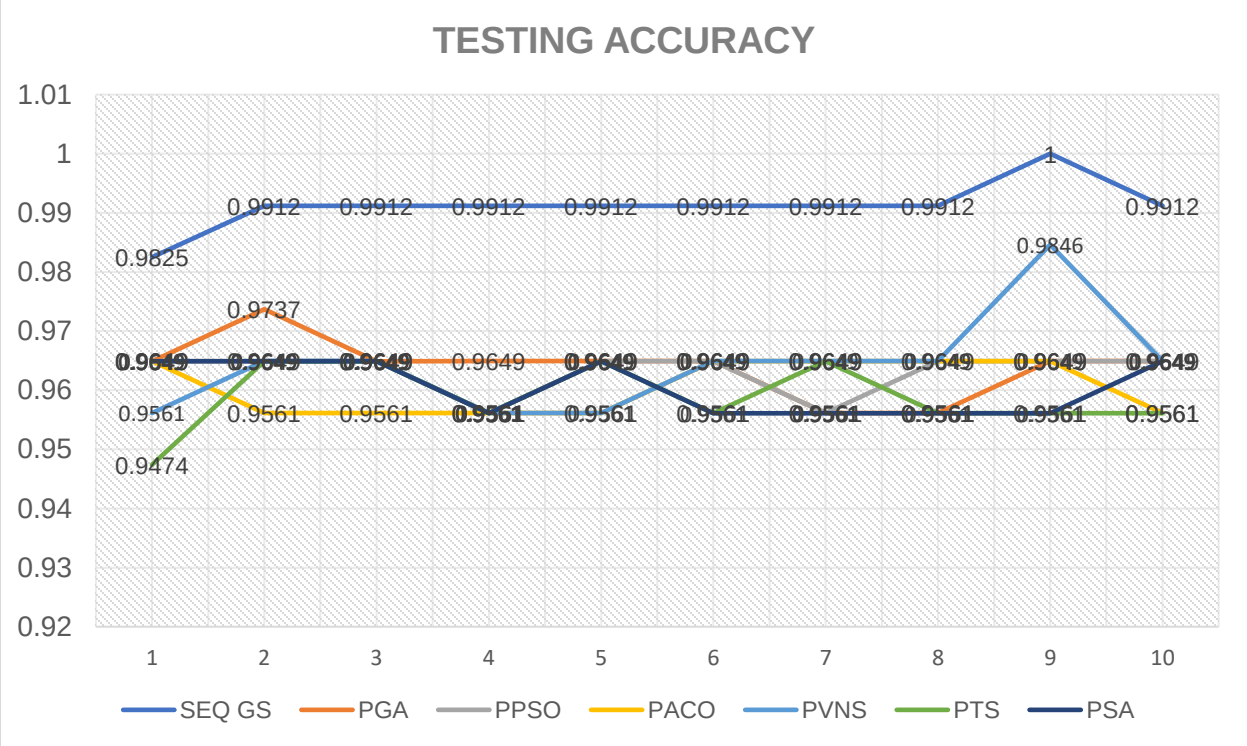


*Figure 7: Test Accuracy Line Chart*

As shown in Table 6 and visualized in Figure 8, the SEQ GC has the highest precision, as expected, while the parallel algorithms are in the range of 0.93 to 0.96. On the recall Table 7, however, it can be seen in that the parallel algorithms are on par with SEQ GC, with the lowest drop experienced by PACO with a value of 0.9467. Other than that, the parallel algorithms were able to get good recall.

| PRECISION | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10*x | SEQ GS | PGA | PPSO | PACO | PVNS | PTS | PSA |
| 1 | 1 | 0.9467 | 0.9467 | 0.9467 | 0.9459 | 0.9333 | 0.9467 |
| 2 | 0.9861 | 0.9595 | 0.9467 | 0.9467 | 0.9467 | 0.9467 | 0.9467 |
| 3 | 1 | 0.9467 | 0.9589 | 0.9589 | 0.9467 | 0.9467 | 0.959 |
| 4 | 1 | 0.9467 | 0.9459 | 0.9459 | 0.9459 | 0.946 | 0.9459 |
| 5 | 0.9861 | 0.9467 | 0.9467 | 0.9467 | 0.9459 | 0.9467 | 0.9467 |
| 6 | 0.9861 | 0.9467 | 0.9467 | 0.9467 | 0.9467 | 0.946 | 0.9459 |
| 7 | 0.9861 | 0.9459 | 0.9459 | 0.9459 | 0.9467 | 0.9467 | 0.9459 |
| 8 | 1 | 0.9459 | 0.9589 | 0.9589 | 0.9467 | 0.946 | 0.9459 |
| 9 | 1 | 0.9467 | 0.9589 | 0.9589 | 0.9649 | 0.946 | 0.9459 |
| 10 | 0.9861 | 0.9467 | 0.9467 | 0.9467 | 0.9467 | 0.946 | 0.9467 |

*Table 6: Precision Table*

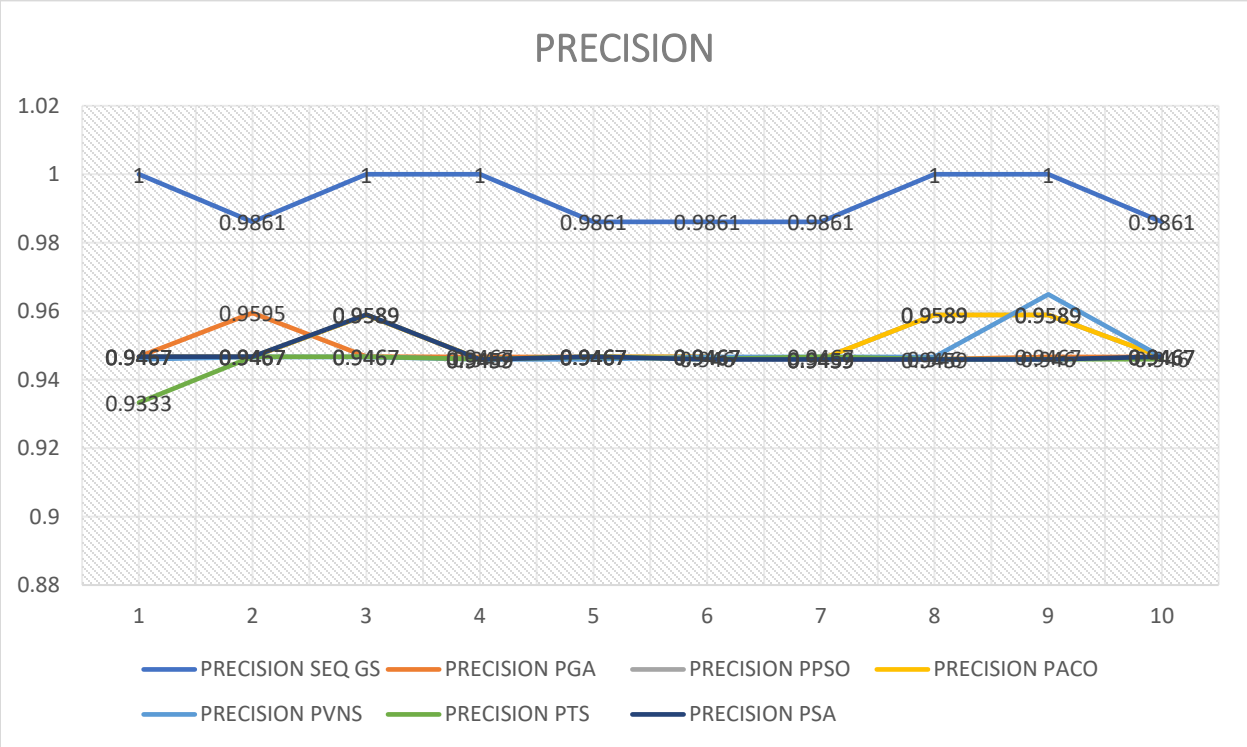


*Figure 8: Precision Line Chart*

| RECALL | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10*x | SEQ GS | PGA | PPSO | PACO | PVNS | PTS | PSA |
| 1 | 0.9718 | 1 | 1 | 0.9859 | 0.9859 | 0.9859 | 1 |
| 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 3 | 0.9859 | 1 | 0.9859 | 1 | 1 | 1 | 0.9859 |
| 4 | 1 | 1 | 0.9859 | 0.9859 | 0.9859 | 0.9859 | 0.9859 |
| 5 | 1 | 1 | 1 | 0.9859 | 0.9859 | 1 | 1 |
| 6 | 1 | 1 | 1 | 1 | 1 | 0.9859 | 0.9859 |
| 7 | 1 | 0.9859 | 0.9859 | 1 | 1 | 1 | 0.9859 |
| 8 | 0.9859 | 0.9859 | 0.9859 | 1 | 1 | 0.9859 | 0.9859 |
| 9 | 1 | 1 | 0.9859 | 0.9467 | 0.9467 | 0.9859 | 0.9859 |
| 10 | 1 | 1 | 1 | 1 | 1 | 0.9859 | 1 |

*Table 7: Recall table*

As shown in Table 8 and Figure 10, SEQ GC once again reached the highest F1-score, with the parallel algorithms not far behind. The parallel algorithms oscillate in the range of values between 0.96 and 0.98. PVNS was able to reach a peak of 1.0 in the 9th iteration, which was on par with SEQ GS.

| F1 SCORE | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10^x | SEQ GS | PGA | PPSO | PACO | PVNS | PTS | PSA |
| 1 | 0.995 | 0.9726 | 0.9726 | 0.9726 | 0.9655 | 0.9589 | 0.9726 |
| 2 | 0.993 | 0.9793 | 0.9726 | 0.9655 | 0.9726 | 0.9726 | 0.9726 |
| 3 | 0.99 | 0.9726 | 0.9722 | 0.9655 | 0.9726 | 0.9726 | 0.9722 |
| 4 | 0.995 | 0.9726 | 0.9655 | 0.9655 | 0.9655 | 0.9655 | 0.9655 |
| 5 | 0.993 | 0.9726 | 0.9726 | 0.9655 | 0.9655 | 0.9726 | 0.9726 |
| 6 | 0.993 | 0.9726 | 0.9726 | 0.9726 | 0.9726 | 0.9655 | 0.9655 |
| 7 | 0.993 | 0.9655 | 0.9655 | 0.9726 | 0.9726 | 0.9726 | 0.9655 |
| 8 | 0.9929 | 0.9655 | 0.9722 | 0.9726 | 0.9726 | 0.9655 | 0.9655 |
| 9 | 1 | 0.9726 | 0.9722 | 0.9726 | 1 | 0.9655 | 0.9655 |
| 10 | 0.993 | 0.9726 | 0.9726 | 0.9655 | 0.9726 | 0.9655 | 0.9726 |

*Table 8: F1 Score Table*

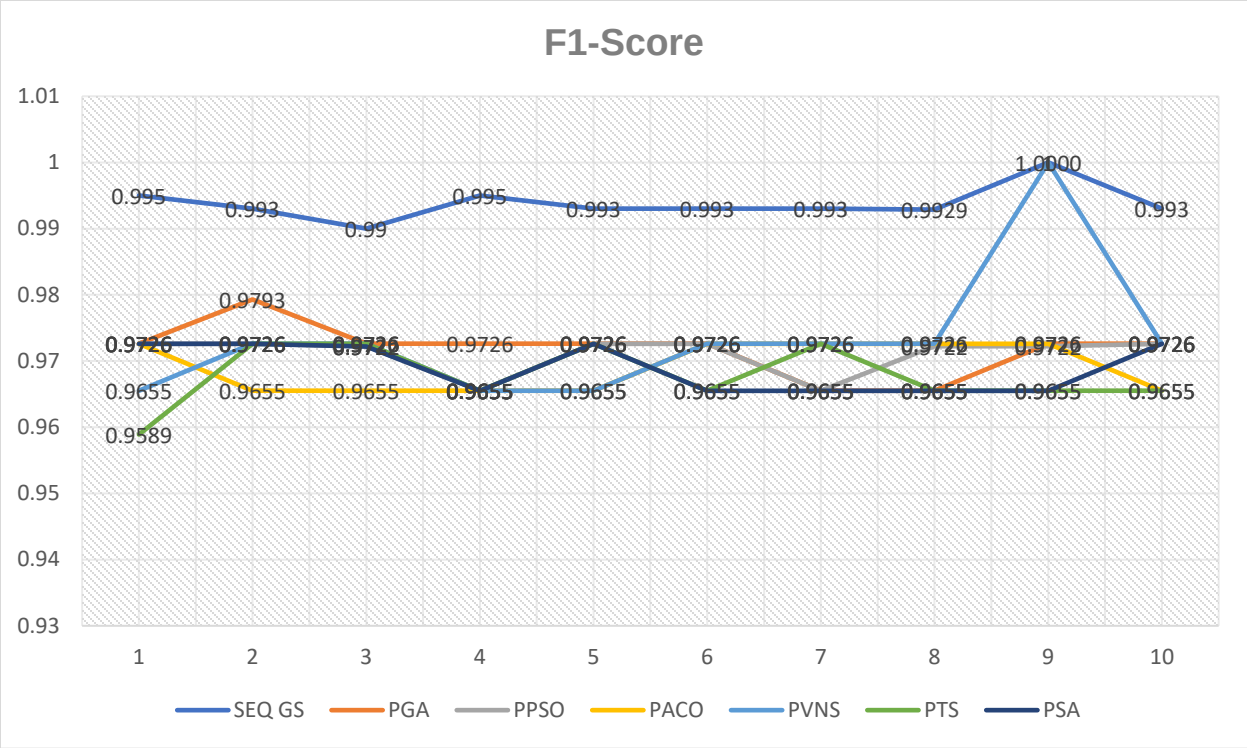


*Figure 10: F1-Score Line Chart*

For MCC, as shown in Table 9 and visualized in Figure 11, the SEQ GC once again reached the highest performance out of all the algorithms, but the parallel algorithms were not far behind. The algorithms seem to get a lower score compared to the other metrics, as the lowest value attained was from PTS, where it got 0.88 in the first iteration. The PGA was able to reach a peak of 0.9447.

| MCC | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10*x | SEQ GS | PGA | PPSO | PACO | PVNS | PTS | PSA |
| 1 | 0.9637 | 0.9266 | 0.9068 | 0.9266 | 0.9068 | 0.8885 | 0.9266 |
| 2 | 0.9814 | 0.9447 | 0.9266 | 0.9068 | 0.9266 | 0.9266 | 0.9266 |
| 3 | 0.9816 | 0.9266 | 0.9068 | 0.9068 | 0.9266 | 0.9266 | 0.9253 |
| 4 | 0.9816 | 0.9266 | 0.9266 | 0.9068 | 0.9068 | 0.9068 | 0.9068 |
| 5 | 0.9814 | 0.9266 | 0.9266 | 0.9068 | 0.9068 | 0.9266 | 0.9266 |
| 6 | 0.9814 | 0.9266 | 0.9266 | 0.9266 | 0.9266 | 0.9068 | 0.9068 |
| 7 | 0.9814 | 0.9068 | 0.9068 | 0.9266 | 0.9266 | 0.9266 | 0.9068 |
| 8 | 0.9816 | 0.9068 | 0.9068 | 0.9266 | 0.9266 | 0.9068 | 0.9068 |
| 9 | 1 | 0.9266 | 0.9266 | 0.9266 | 0.9266 | 0.9068 | 0.9068 |
| 10 | 0.9814 | 0.9266 | 0.9266 | 0.9068 | 0.9266 | 0.9068 | 0.9266 |

*Table 9: MCC Table*

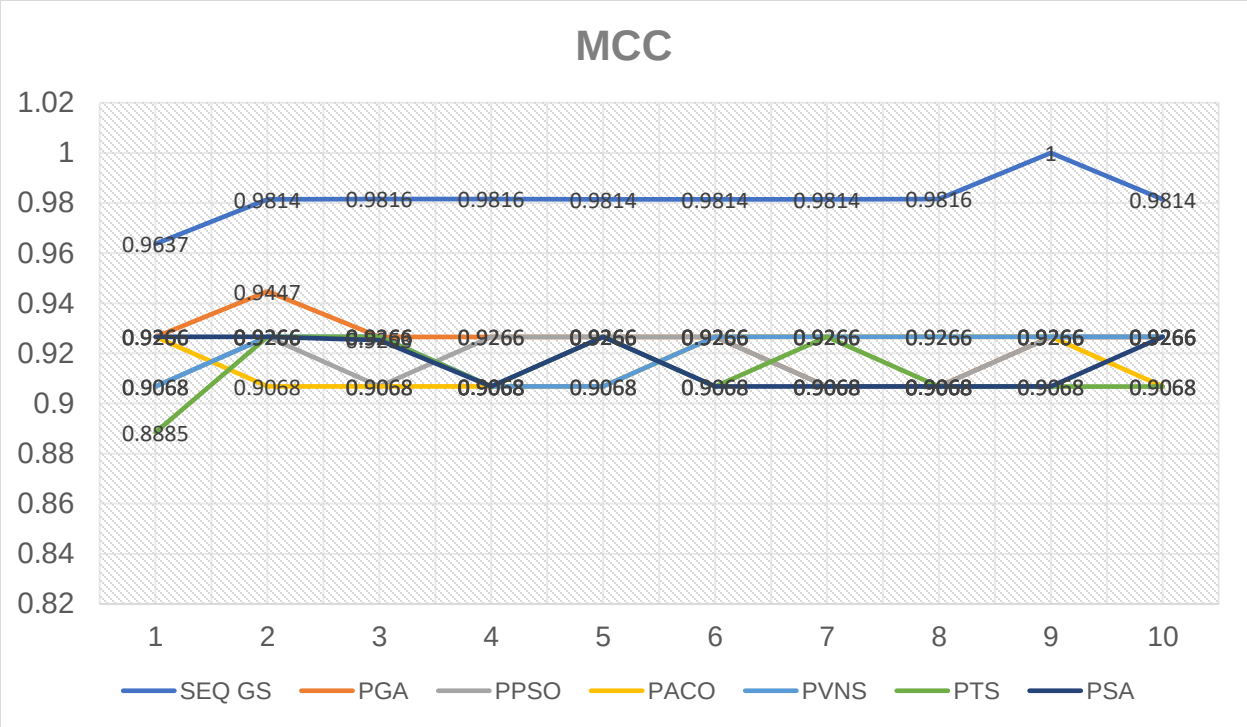


*Figure 11: MCC Line Chart*

Overall, in terms of evaluation metrics for the test set, it is no surprise that SEQ GS would get better results than the parallel algorithms, but as seen from the tables and figures, they were also able to get really good results that are not far off from the results of SEQ GC and with significantly lower runtime, as discussed earlier. Machine learning models do not aim for optimal accuracy, as the researcher wants the models to be generalizable so that they can adapt to different datasets. This means that, with the help of the parallel algorithms, the model was able to perform really well. What can also be noticed is that the PPSO performed really well despite only having a runtime of 2 seconds per iteration.

| **MEMORY USAGE** | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10*x | SEQ GS | PGA | PPSO | PACO | PVNS | PTS | PSA |
| 1 | 1.9453 | 6.9648 | 4.323 | 8.8125 | 1.5703 | 8.7969 | 44.9492 |
| 2 | 1.3438 | 7.2461 | 4/557 | 9.0313 | 1.7031 | 8.957 | 8.7148 |
| 3 | 10.957 | 6.793 | 5.2123 | 8.6563 | 1.746 | 8.7734 | 8.8555 |
| 4 | 52.1172 | 6.7934 | 4.8702 | 2.3633 | 1.707 | 8.9961 | 8.8867 |
| 5 | 59.5195 | 6.8535 | 4.4557 | 8.9492 | 1.7656 | 8.7695 | 8.7148 |
| 6 | 51.2656 | 7.457 | 5.757 | 9.1055 | 1.9296 | 8.5117 | 8.9805 |
| 7 | 24.5938 | 6.5575 | 4.7789 | 9.2773 | 1.8086 | 8.9258 | 7.2305 |
| 8 | 50.7539 | 6.793 | 5.1102 | 107.2617 | 1.9336 | 8.8164 | 8.7422 |
| 9 | 74.4297 | 7.457 | 5.3445 | 35.8086 | 1.7148 | 8.6953 | 23.8672 |
| 10 | 75.4102 | 6.793 | 4.8723 | 23.1367 | 1.5 | 8.7656 | 8.7578 |

*Table 10: Memory Usage Table (mb)*

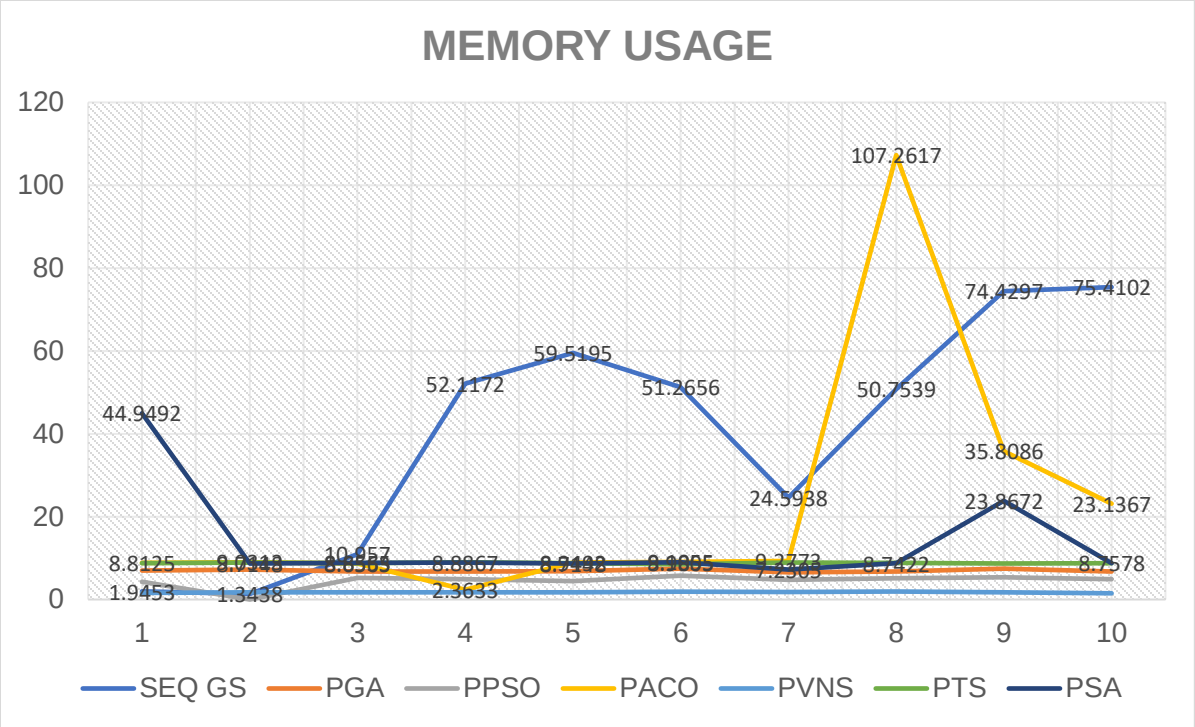


*Figure 12: Memory Usage Line Chart*

As shown in Table 10, the parallel algorithms seem to use less memory and are more stable, unlike the SEQ GC, whose memory usage has increased dramatically. PACO was also able to experience a sudden peak on the 8th iteration. From Figure 13, it can be seen that the parallel algorithms have stable numbers of functions that are evaluated, which makes sense since the functions that are evaluated are done by the threads, which are in fixed numbers. The SEQ GC, on the other hand, has its function evaluations increase since this algorithm tries out every hyperparameter combination. It can also be seen that PACO had the highest number of function evaluations when compared to the other five parallel algorithms.

| NUMBER OF FUNCTION EVALUATIONS | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10*x | SEQ GS | PGA | PPSO | PACO | PVNS | PTS | PSA |
| 1 | 200 | 5120 | 1000 | 10000 | 240 | 3000 | 3000 |
| 2 | 800 | 5120 | 1000 | 10000 | 280 | 3000 | 3000 |
| 3 | 1800 | 5120 | 1000 | 10000 | 260 | 3000 | 3000 |
| 4 | 3200 | 5120 | 1000 | 10000 | 380 | 3000 | 3000 |
| 5 | 5000 | 5120 | 1000 | 10000 | 300 | 3000 | 3000 |
| 6 | 7200 | 5120 | 1000 | 10000 | 360 | 3000 | 3000 |
| 7 | 9800 | 5120 | 1000 | 10000 | 340 | 3000 | 3000 |
| 8 | 12800 | 5120 | 1000 | 10000 | 280 | 3000 | 3000 |
| 9 | 16200 | 5120 | 1000 | 10000 | 280 | 3000 | 3000 |
| 10 | 20000 | 5120 | 1000 | 10000 | 280 | 3000 | 3000 |

*Table 11: Function Evaluations Table*

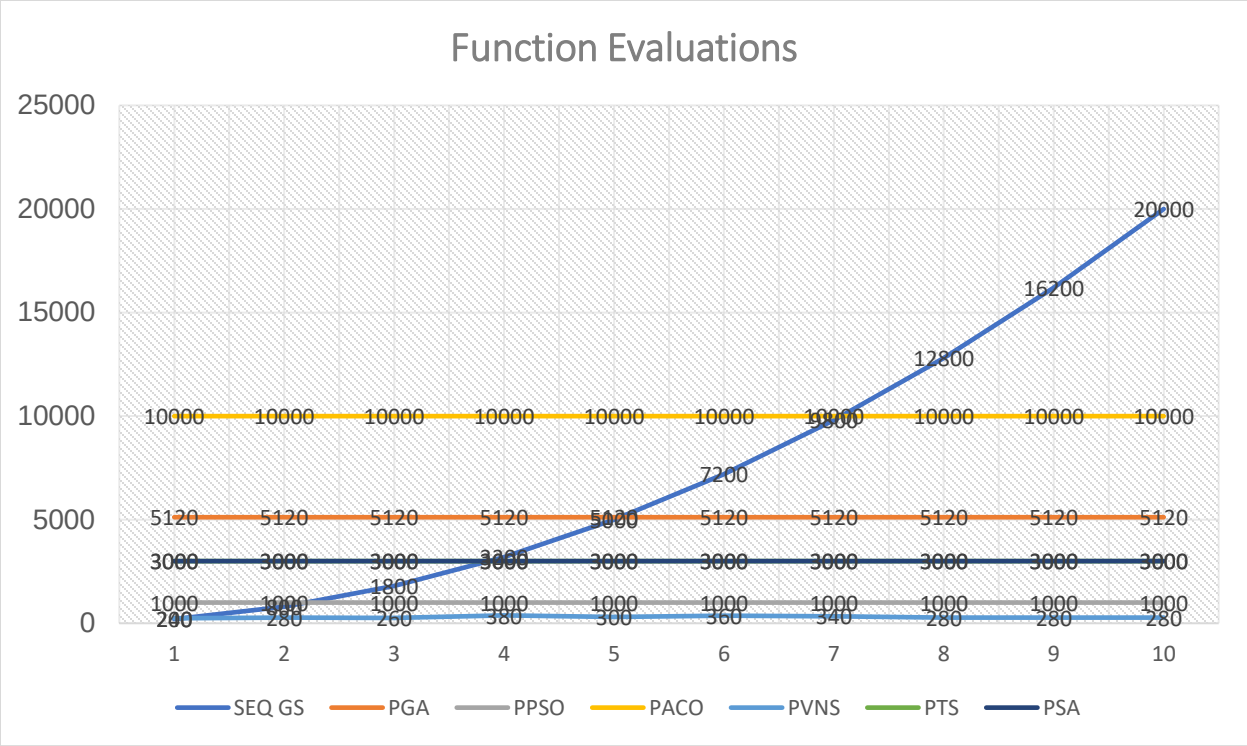


*Figure 13: Function Evaluations Line Chart*

Ultimately, this demonstrates that the parallel algorithms seem to perform really well as hyperparameter searches for Logistic Regression, and it seems that PPSO was able to perform the fastest with the least memory usage and a low number of function evaluations, but still was able to be on par with other algorithms in terms of evaluation metrics like accuracy, precision, etc.

## Conclusion

In conclusion, the use of parallel metaheuristic algorithms for hyperparameter search in Logistic Regression is a promising method for harnessing the power of GPUs for more efficient searching. Not only is it parallel, but it also utilizes a smarter way of searching that allows the researcher to get a near-optimal solution.

Other aspects that can be considered adding to the study are the use of more complex machine learning models that need more computational power and have a larger hyperparameter space. Ensemble models or AutoML models could benefit from this kind of parallel strategy since both of these complex models involve a lot of searching. The researcher can also consider more sophisticated parallel strategies that involve partitioning the search space or using shared memory to help guide the threads to a better solution for the upcoming capstone project.

**References**


- *Alba, E., Luque, G., & Nesmachnow, S. (2012). Parallel Metaheuristics: Recent Advances and New Trends. International Transactions in Operational Research, 20, 1-48.* https://doi.org/10.1111/j.1475-3995.2012.00862.x

- *Crainic, T. G., & Toulouse, M. (2010). Parallel Meta-heuristics. In Handbook of Metaheuristics (pp. 497-541). ISBN: 978-1-4419-1663-1.* https://doi.org/10.1007/978-1-4419-1665-5_17

- *Wicaksono, A., & Afif, A. (2018). Hyper Parameter Optimization using Genetic Algorithm on Machine Learning Methods for Online News Popularity Prediction. International Journal of Advanced Computer Science and Applications, 9.* https://doi.org/10.14569/IJACSA.2018.091238 [1]

- *Aguerchi, K., Jabrane, Y., Habba, M., & Hajjam El Hassani, A. (2024). A CNN Hyperparameters Optimization Based on Particle Swarm Optimization for Mammography Breast Cancer Classification. Journal of Imaging, 10(2), 30.* https://doi.org/10.3390/jimaging10020030 [2]

- *Bhandare, Y. S., & Hajiarbabi, M. (2023). Hyperparameter Tuning with Simulated Annealing and Genetic Algorithm. Available at SSRN: https://ssrn.com/abstract=4432719 or* http://dx.doi.org/10.2139/ssrn.4432719 [3]

- *Ripon, K. S. N., Glette, K., Khan, K., Høvin, M., & Torresen, J. (2013). Adaptive Variable Neighborhood Search for Solving Multi-Objective Facility Layout Problems with Unequal Area Facilities. *Swarm and Evolutionary Computation, 8*, 1-12. doi:10.1016/j.swevo.2012.07.003*

- *El-Shorbagy, M., & Hassanien, A. E. (2018). Particle Swarm Optimization from Theory to Applications. International Journal of Rough Sets and Data Analysis, 5. doi:10.4018/IJRSDA.2018040101*

- *Maniezzo, V., Gambardella, L. M., & Luigi, F. (2004). Ant Colony Optimization. ISBN: 978-3-642-05767-0. doi:10.1007/978-3-540-39930-8_5*

- *Glover, F., Laguna, M., & Marti, R. (2008). Tabu Search (Vol. 16). doi:10.1007/978-1-4615-6089-0*

- *Busetti, F. (2001). Simulated annealing overview.*

- *Mueller, J. P., & Massaron, L. (2022). Algorithms For Dummies, 2nd Edition. Wiley.*

- *Talaei Khoei, T., & Kaabouch, N. (2023, October 9). Machine Learning: Models, Challenges, and Research Directions. Future Internet, 15, 332.* https://doi.org/10.3390/fi15100332

- *Arafa, A., Radad, M., El-Fishway, N., & Badawy, M. (2022). Logistic Regression Hyperparameter Optimization for Cancer Classification. Menoufia Journal of Electronic Engineering Research, Volume(Issue), page numbers. doi:10.21608/MJEER.2021.70512.1034*

- *Mostaghim, S., Branke, J., Lewis, A., & Schmeck, H. (2008, June). Parallel Multi-objective Optimization using Master-Slave Model on Heterogeneous Resources. In Proceedings of the Conference on Evolutionary Computation (pp. page numbers). doi:10.1109/CEC.2008.4631060*


**Kernel Code For Evaluation (used in all parallel algorithms)**

```
_global_ void evaluate_fitness(float *hyperparameters, float *data, float
*labels, float *accuracies,
    int num_features, int num_samples, int pop_size, unsigned long long seed) {
int idx = threadIdx.x + blockIdx.x * blockDim.x;
if (idx < pop_size) {
      int penalty_idx = int(hyperparameters[4 * idx]);
      float C = hyperparameters[4 * idx + 1];
      int solver_idx = int(hyperparameters[4 * idx + 2]);
      float l_r = hyperparameters[4 * idx + 3];

      // Initializing hyperparameters
      float penalty = penalty_idx == 0 ? 0 : 1;

      // Allocating global memory for weights and gradients
      float *weight = new float[num_features];
      float *gradients = new float[num_features];
      float initial_weight = idx * 0.01;

      // Initializing weights
      for (int i = 0; i < num_features; i++) {
            weight[i] = initial_weight;
      }

      float learning_rate = l_r;
      int num_iterations = 100;

      for (int iter = 0; iter < num_iterations; iter++) {
            // zero gradients at the start
            for (int j = 0; j < num_features; j++) {
                  gradients[j] = 0.0;
```

```
            }
            // Compute gradients
            for (int sample = 0; sample < num_samples; sample++) {
                  float linear_combination = 0.0;
            for (int j = 0; j < num_features; j++) {
                  linear_combination += data[sample * num_features + j] *
            weight[j];
            }
            float prediction = 1.0 / (1.0 + exp(-linear_combination));
            float error = prediction - labels[sample];
            for (int j = 0; j < num_features; j++) {
                  gradients[j] += data[sample * num_features + j] * error;
            }
      }

      // We update weights with L1 or L2 penalty based on the value of penalty
      if (penalty == 0) { // L1 penalty
            for (int j = 0; j < num_features; j++) {
                  float regularization_term = (weight[j] > 0) ? C : -C;
                  weight[j] -= learning_rate * (gradients[j] / num_samples +
            regularization_term / num_samples);
            }
      } else if (penalty == 1) { // L2 penalty
            for (int j = 0; j < num_features; j++) {
                  weight[j] -= learning_rate * (gradients[j] / num_samples + 2 *
            C * weight[j]);
            }
      }
}

// Calculate accuracy
int correct_predictions = 0;
for (int sample = 0; sample < num_samples; sample++) {
      float linear_combination = 0.0;
      for (int j = 0; j < num_features; j++) {
            linear_combination += data[sample * num_features + j] * weight[j];
      }
      float prediction = 1.0 / (1.0 + exp(-linear_combination));
      int predicted_label = (prediction >= 0.5) ? 1 : 0;
      if (predicted_label == int(labels[sample])) {
            correct_predictions += 1;
      }
}
accuracies[idx] = (float)correct_predictions / num_samples;
```

```
// Debug message
printf("Thread %d: penalty_idx=%d, C_idx=%f, solver_idx=%d, accuracy=%f\\n",
idx, penalty_idx, C, solver_idx, accuracies[idx]);

// Free allocated memory
delete[] weight;
delete[] gradients;
}
}
```

**Kernel Code For Pheromone Update (used only in PACO)**

```
_global_ void update_pheromone(float *pheromone, int *best_hyperparameters, int
num_penalty_mode, int num_C_values, int num_Solver, int num_Learning_rate, float
evaporation_rate) {
    int penalty_mode_idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (penalty_mode_idx < num_penalty_mode) {
        for (int C_values_idx = 0; C_values_idx < num_C_values; ++C_values_idx) {
            for (int Solver_idx = 0; Solver_idx < num_Solver; ++Solver_idx) {
                for (int Learning_rate_idx = 0; Learning_rate_idx <
num_Learning_rate; ++Learning_rate_idx) {
                    int idx = (penalty_mode_idx * num_C_values * num_Solver *
num_Learning_rate) + (C_values_idx * num_Solver * num_Learning_rate) +
(Solver_idx * num_Learning_rate) + Learning_rate_idx;
                    if (penalty_mode_idx == best_hyperparameters[0] &&
C_values_idx == best_hyperparameters[1] && Solver_idx == best_hyperparameters[2]
&& Learning_rate_idx == best_hyperparameters[3]) {
                        pheromone[idx] *= (1.0 - evaporation_rate);
                        pheromone[idx] += evaporation_rate;
                    } else {
                        pheromone[idx] *= (1.0 - evaporation_rate);
                    }
                }
            }
        }
    }
}
```