scientific reports

OPEN

An automated parallel genetic algorithm with parametric adaptation for distributed data analysis

Laila Al-Terkawi[1]✉ & Matteo Migliavacca[2]

Unleashing the potential of large-scale data analysis requires advanced computational methods capable of managing the immense size and complexity of distributed data. Genetic algorithms (GAs), known for their adaptability, benefit significantly from parallelization, prompting ongoing enhancements to boost performance further. This study proposes the integration of automatic termination and population sizing mechanisms into parallel GAs to augment their flexibility and effectiveness. We extend PDMS-BioHEL and PDMD-BioHEL, two parallel GA-based classifiers implemented on the Spark platform, and through extensive experimentation, demonstrate the efficacy of our approach in enhancing computational efficiency and user-friendliness. However, while these automated strategies significantly reduce the need for manual parameter tuning, thereby increasing time efficiency, they may sometimes lead to a slight reduction in final solution accuracy, particularly under complex scenario conditions. This trade-off between efficiency and accuracy is critical, especially when high precision is paramount. Our techniques enable more efficient and effective large-scale data analysis using parallel GAs, providing a robust foundation for future advancements and inviting further investigation into balancing these aspects.

Keywords Genetic algorithms (GAs), GAs parameter control, Classification, Large-scale data processing, Spark

Large-scale data analytics has emerged as a pivotal force in both academic research and the business world, unlocking valuable insights into scientific phenomena and driving informed decision-making. The process of extracting meaningful knowledge from vast datasets, however, presents a formidable challenge, primarily due to the constraints of conventional database and software techniques. To address this challenge, large-scale parallel computing platforms have become indispensable, offering cost-effective, reliable, and scalable resources for storing and processing massive amounts of data.

In this context, data mining plays a crucial role. It involves analyzing large sets of data to discover underlying patterns, relationships, and insights. Among the various approaches, evolutionary algorithms, and particularly Genetic Algorithms (GAs), have proven to be essential. These algorithms are highly adaptable and have been successfully applied in a wide range of data mining tasks, such as classification, clustering, regression, and time-series analysis. The advantages of GAs are particularly noticeable in complex, large-scale data mining challenges. These methods employ a population-based heuristic that demonstrates strong performance in navigating extensive and complex search spaces, particularly in instances where traditional approaches may encounter challenges. Moreover, the inherent parallelism of GAs complements parallel computing architectures well, significantly boosting their performance in scenarios that involve complex or extensive search spaces.

However, while the efficiency gains from these advanced computational strategies are substantial, they often come with the trade-off of potentially reduced accuracy in the final solutions, particularly when automated parameter selection is employed to expedite the analysis process. This trade-off is critical and must be carefully managed, especially in applications where precision is paramount.

Academic surveys conducted in[1] and[2] have meticulously documented the progression of parallelization techniques in GAs, tracing their development back from the 1980s. These studies highlight the significant successes achieved in diminishing computation time through such techniques. Nevertheless, it is important to note that the majority of these proposals operate under the assumption that the complete dataset is immediately

[1]International University - Kuwait (IUK), Al-Ardiya, Kuwait. [2]School of Computing, University of Kent, Canterbury, UK. ✉email: laila.alterkawi@iuk.edu.kw

accessible at each worker node, either through replication or via shared/distributed memory systems. This assumption presents a practical constraint when dealing with datasets of an exceptionally large scale.

To tackle this challenge, data partitioning emerges as a viable solution, wherein the dataset is divided into distinct partitions, and random access to data across partitions is restricted during intensive computations, such as fitness evaluation. In[3], we introduced a system to tightly control data movement, enabling genetic algorithms (GAs) to scale up and enhance processing efficiency. Two partitioned data models are introduced, namely PDMD and PDMS, specifically tailored to address large-scale data classification problems within the context of parallel GAs. These models were applied to develop two parallel versions of BioHEL[4], a popular single-node Iterative Rule Learning (IRL) GA classifier, utilizing the Spark distributed data processing platform with a primary focus on data partitioning.

Our studies highlighted a substantial decrease in classifier learning time with increased parallelization. However, a challenge arose in scaling the PDMD model: a higher number of partitions resulted in smaller local populations per partition, leading to premature convergence and, consequently, suboptimal solutions. Efforts to address this by increasing migration between partitions did not yield substantial improvements. This underscores the intricate interplay between parallelization, population distribution, and the quality of solutions in the PDMD model, necessitating further investigation into optimal configurations.

These findings underscore the critical role of parameter control techniques in achieving superior performance, accuracy, and efficiency. Implementing parameter control alleviates the need for exhaustive exploration of parameter value combinations for new problem instances. In this study, our focus centers on two pivotal GA parameters: the number of iterations and the population size. We propose automated approaches for their selection, with the aim of enhancing the overall performance of parallel GAs, rendering them more adaptable and efficient for large-scale data analysis tasks.

The remainder of the paper is organized as follows: In Section 2, we provide an overview of related work in the field of parameter control, discussing the advancements and approaches employed by researchers. In section 3 we introduce and explain our AUTO+PDMS and AUTO+PDMD models implementations. The effectiveness and performance of both models are evaluated in Section 4. In Section 5, we analyze and discuss the results obtained from our evaluation. In section 6 we highlight the main scenarios where the proposed methods may not perform optimally. Finally, in Section 7, we conclude the paper by summarizing the key findings and contributions of our research. Additionally, we discuss potential avenues for future research in this domain.

Related work

Genetic Algorithms (GAs) are known for their versatility and robustness as optimization techniques, showcasing their effectiveness in a diverse array of domains. They have gained widespread recognition for their ability to tackle various challenges, including machine learning tasks such as clustering[5,6], regression[7], time-series analysis[8], and improving the accuracy of classification algorithms[9,10]. Notably, Genetic Algorithms have demonstrated robust efficacy in the execution of classification tasks[11,12].

However, despite their wide-ranging applicability, GAs can sometimes fall short of expectations, often due to the critical factor of parameter selection. This becomes especially vital in complex problems or when optimizing performance is essential. Suboptimal performance frequently arises from the misalignment of GA parameters, including population size, crossover probability, and mutation probability. These parameters, in both their absolute values and relative proportions, determine the GA's behavior during the search process. Striking a delicate balance between exploiting the current best solution and exploring the entire solution space by combining solutions is essential[13].

One common issue encountered in GAs is premature convergence[14], a situation where the algorithm prematurely converges to a suboptimal solution. It occurs when GAs settle on a suboptimal solution too early, failing to explore potentially superior alternatives. Researchers have recognized this challenge and in[15], they focus on identifying methods to prevent genetic algorithms from prematurely terminating.

The process of selecting suitable parameters for a genetic algorithm often involves trial and error approach, which can be time-consuming and computationally demanding. To address this issue, researchers have proposed three methods: empirical studies, parameter adaptation, and theoretical modeling.

Empirical studies involve conducting analyses on various test functions to observe the relationships between different aspects of a genetic algorithm. While this method provides valuable insights, it may not always yield generalizable results for all cases. Additionally, deriving precise recommendations from these studies can be challenging.

Parameter adaptation methods have also been suggested, focusing on dynamically controlling one or more operators, such as crossover probability or mutation rate. These methods adaptively adjust parameter values based on fitness values obtained from each generation. By continuously monitoring the performance of the algorithm, these adaptive techniques aim to maintain population diversity and improve convergence towards optimal solutions.

Theoretical modeling is another approach that seeks to provide a deeper understanding of the GA dynamics and the influence of parameter values on its behavior. Through mathematical models and analyses, researchers aim to uncover the underlying principles governing the GA's performance. This theoretical knowledge can guide the selection of parameter values and lead to more informed decision-making. Theoretical studies on GA parameter tuning are based on simplified models of the problem and GA behavior, which may not accurately reflect real-world situations. Additionally, these models often rely on binary coding, which limits their applicability to problems that benefit from more complex representations. As a result, theoretical studies require knowledge of the problem that is not readily available, and their results may not be generalizable to other problems or encoding schemes.

Over the past decade, automated methods for configuring state-of-the-art algorithms have gained widespread recognition as a highly effective alternative. Notable examples of such general-purpose automated algorithm configurators include irace[16], ParamILS[17], and SMAC[18]. These automated tools have streamlined the process of fine-tuning algorithms, providing a robust alternative to manual configuration.

The software package irace is dedicated to implementing multiple automated configuration methods[16]. Specifically, it introduces iterated racing procedures, which have proven successful in automating the configuration of various cutting-edge algorithms. Irace's primary objective is to streamline the demanding task of fine-tuning optimization algorithm parameters. Furthermore, it can also be applied to discover optimal settings in diverse computational systems, such as robotics and traffic light controllers. However, it's important to note that the current iterated racing algorithms within irace have some limitations. Most notably, they were primarily designed for scenarios where reducing computation time is not the primary focus.

In a related context,[17] presents ParamILS, an approach grounded in Reinforcement Learning (RL). ParamILS is designed to fine-tune RL agents in environments belonging to the algorithm configuration domain. This domain seeks to identify better parameters to enhance overall algorithm performance.

Another method tailored to addressing computationally expensive black box problems is SMAC, the sequential configuration of algorithms based on the search space[18]. This approach involves adapting a general algorithm to a specific problem, effectively replacing manual parameter adjustments with automatic determination.

One of the primary challenges in automated algorithm configuration is the time required to assess a configuration's performance and the size of the configuration space, particularly when dealing with complex training dataset. The practice of randomly selecting configurations can lead to timeouts and provide limited valuable insights. While irace employs racing methods based on statistical tests to discard underperforming configurations, it may still result in insufficient information about the configurations' effectiveness.

In the subsequent sections, we will emphasize the primary approaches for overseeing and automating two crucial GA parameters: the number of iterations and the population size, both of which constitute the central focus of our research.

Number of iterations

Our research, as outlined in[3], extensively investigates the impact of the number of iterations on both PDMS and PDMD parallel GA models. Our findings consistently reveal a notable trend: in most scenarios, GA achieves its final optimal solution using only half or even fewer iterations than the default predefined value, rendering the remaining iterations redundant. However, there are instances where the predetermined fixed number of iterations falls short of identifying the optimal solution. This underscores the critical need for an automated approach to regulate the number of iterations, a development that holds substantial potential for enhancing overall efficiency.

Nevertheless, it's essential to acknowledge that many GA applications lack an automatic termination mechanism. Typically, Evolutionary Algorithms (EAs) cannot autonomously determine when or where to terminate, often relying on users to predefine a termination criteria. The most commonly employed termination criteria are as follows: (1) reaching a maximum number of generations or function evaluations, which is a straightforward yet non-optimal approach and is often problem-dependent, requiring prior knowledge about the test problem; (2) achieving a fitness value that approaches the known global minimum, a rarity in real-world scenarios; and (3) hitting a maximum number of consecutive generations without improvement. For instance, in[19], termination occurs after 50 generations without improvement, while[20] allows users to decide when to halt the algorithm. It's worth noting that these termination criteria are generally inapplicable to many real-world problems since they are based on unconstrained test problems with known minima.

Recent studies in the field of EAs have explored various termination criteria, with some notable examples found in[21] and[22]. In[21], empirical studies aim to detect the number of generations required for population convergence based on problem characteristics. Meanwhile,[22] conducted a survey that categorizes prominent termination criteria in EAs. The study emphasizes the effectiveness of combining direct termination criteria with threshold-based criteria to ensure reliable EA convergence. The prevailing trend, as highlighted in[23], leans toward adopting adaptive termination conditions that rely on either genotypical or phenotypical terminations instead of a fixed number of iterations.

In a broader context, recent termination criteria for EAs can be classified into four categories[23]:

- TFit Criterion: This category relies on convergence measures of the best fitness function values across generations.
- TPop Criterion: It focuses on convergence measures of the entire population across generations.
- The TBud Criterion for computational budget is characterized by utilizing the maximum number of generations or function evaluations, which is dependent on the specific problem.
- TSFB Criterion: This category employs search feedback measures to assess the progress of exploration and exploitation processes.

Using only TFit as a termination criterion carries risks, as it might cause EAs to get stuck in local minima, particularly if the algorithm quickly settles into a profound local minimum. Depending solely on TPop for termination isn't typically suitable either, as driving the whole population or a large part of it to converge genotypically can be computationally expensive.

Using TBud depends on the specific problem, and the user should already know about the addressed problem before employing it. Often, having one individual reach a global minimum is sufficient. Although[24] demonstrated that an upper bound for the number of iterations required for convergence can be theoretically estimated based on a desired confidence level,[25] pointed out that these criteria are of limited practical interest, primarily because they tend to be excessively large. Finally, while employing TSFB appears to be efficient, it might encounter challenges related to the curse of dimensionality. Additionally, storing and reviewing extensive historical search data presents complexity issues.

As a result, an intelligent termination strategy becomes imperative. In our modified implementation for both PDMD and PDMS models, we've made a significant change by replacing the fixed number of iterations used in the original BioHEL. Instead, we've introduced a convergence condition that combines TFit and TPop. Under this new approach, the algorithm continually assesses improvements in the best-found solution (fitness) and the average fitness level. If a fixed count of consecutive generations (in our case, 3) passes with no improvement in either the best-found solution (fitness) or the average fitness level, the rule-learning process is terminated. This fixed threshold was chosen based on preliminary experiments conducted across multiple datasets, including KDD and HEPMASS, where we observed that longer periods without improvement rarely resulted in significantly better solutions. Limiting the count to three generations ensures computational efficiency while maintaining solution quality. This approach is further supported by prior studies that recommend short stagnation thresholds to prevent overfitting or unnecessary computational overhead in genetic algorithms [CITATION].

Population size

Since the early days of Genetic Algorithm (GA) research, the influence of population size on solution quality has been a subject of investigation.[26]. Yet, there's a compelling argument in favor of considering variable population sizes in GAs, supported by insights documented in[27]. The author noted that population size significantly impacts algorithm performance and highlighted instances where adaptive population size adjustments improved results. Notable approaches, such as the Genetic Algorithm with Variable Population Size (GAVaPS)[28], replace the fixed population size parameter with two properties: age and maximum lifetime. Similarly, the Adaptive Population Size-based GA (APGA)[29], a variant of GAVaPS, employs a steady-state GA and maintains the best individual's lifetime unchanged when individuals age. In[30], a growing population size is introduced when individuals exhibit high fitness or during prolonged stagnation periods, effectively decreasing the population size during short stagnation phases.

Collectively, these studies underscore the pivotal role of selecting an appropriate population size to enhance GA efficiency. Notably, one of the most comprehensive GA calibration strategies has been proposed in[31]. This approach runs multiple populations of varying sizes concurrently, organizing a competitive race among them. These populations progress at different rates, with smaller populations leading in terms of generations completed. For example, at a specific point in time, this GA may feature populations of sizes 256, 512, and 1024. The population of size 256 might be in its 30th generation, while the population of size 512 is in its 6th generation, and the population of size 1024 is still in its 1st generation. Over time, smaller populations are phased out, and larger ones are introduced. These transitions are guided by monitoring average fitness levels within each population. For instance, if the population of size 512 exhibits superior average fitness compared to the population of size 256, the smaller population is terminated, as it's unlikely to outperform the larger one. This approach operates indefinitely and doesn't decide when an optimal population size is reached, making it unsuitable for an IRL/sequential covering setting. In our context, we seek an approach that can eventually terminate with confidence, having achieved a high-quality solution derived from an appropriately sized population for the problem at hand. Thus, our approach involves sequential evolution, doubling the population size until further doubling no longer enhances solution quality. This aligns with the parallel case, ensuring that if a population begins to drift, it is promptly terminated due to lack of progress and restarted with a larger population.

The overall outcome of this approach closely resembles a continuous increase in population size over time. Furthermore, our intention is to commence our GA with a modest initial population size. In theory, as long as the initial size is adequate to kickstart the learning process, the specific value becomes less critical because the population can expand dynamically if required. However, when the population starts exceedingly small, such as with just one or two individuals, it may not facilitate an efficient initial phase for addressing large-scale data problems, potentially leading to premature learning termination. To establish an appropriate starting point, we draw inspiration from the micro-genetic algorithm (micro-GA) as outlined by[26] and first implemented by[32]. Krishnakumar employed a population size of 5 and observed that his micro-GA surpassed the standard GA in terms of processing speed.

The following sections will introduce a strategy for managing and automating the number of iterations and population size of BioHEL to enhance the performance of PDMD and PDMS.

Automated data partitioned models

In our previous work[3], we utilized the widely used Spark framework[33] to implement two parallelization models: the Partitioned-Data Master-Slave (PDMS) model and the Partitioned-Data Multiple-Deme (PDMD) model. These models adapt the Genetic Algorithm (GA) in the BioHEL classifier for large-data classification tasks which would sustantially benefit from parallelisation. In the following paragraph, we explains the general workflow of BioHEL.

```
1  BioHEL(TrainS):
2      ruleList = ∅; stop = false;
3      while stop is false do
4          for repetition=1 to numRepetitions do
5              candidates += findRules(TrainS);
6          (bestRule, TrainS) = bestRule(candidates,
             TrainS);
7          if bestRule not null then
8              ruleList += bestRule; candidates = ∅;
9          else stop = true;
10     return ruleList;
11 findRules(TrainS):
12     pop = TrainS.sample(N);
13     pop.fitness = doFitness(pop,TrainS);
14     for iter = 1 to numIter do
15         offsp = pop.selection();
16         offsp.crossover();
17         offsp.mutation();
18         offsp.fitness = doFitness(offsp,TrainS);
19         pop.replacement(offsp);
20     return pop.best();
21 bestRule(candidates,TrainS):
22     bestRule = candidates.bestFitness();
23     matched = TrainS.matchedBy(bestRule);
24     if bestRule.class = matched.majorityClass() then
25         TrainS = TrainS.removeMatched(bestRule);
26         return (bestRule, TrainS);
27     else return (null, TrainS);
```

Algorithm 1. BioHEL general workflow.

BioHEL algorithm constructs a classifier as a list of rules. Each rule is composed by a set of conditions leading to a predicted class. These rules are learned sequentially where the learning process terminates when no new rules can be derived. The overall workflow can be represented as Algorithm 1, highlighting two key structural components, represented by the two functions `findRules` and `bestRule`.

`findRules` (lines 11–20) is where a rule is crafted using a generational Genetic Algorithm (GA). It starts with a sample from the training set (line 12), where each individual in the initial population is a generalized version of a training instance, and the rule's predicted class aligns with the class of the training instance. At each iteration, individuals are selected for reproduction using tournament selection then the crossover and mutation operation are applied. After the evolutionary process concludes, the best individual (deemed the candidate rule) is selected. This procedure may be repeated more than once, trying to avoid settling on an underperforming local optima (lines 4–5).

Then, after `findRules` identifies the candidate solutions, the `bestRule` function (lines 21–27) selects the best rule. A rule is added to the rule list if its associated class is the majority class of the matched examples (lines 23– 24). If a valid rule is identified, the corresponding training examples are filtered from the training set (line 25), prompting the GA to explore different regions of the search space for subsequent rules. The final rule set is then augmented with a default rule representing the majority class, so that any example not matching any of the rules in the rule list is classified under this default rule.

Spark Our implementation utilizes Spark, a framework with a master node and several worker nodes. This framework operates on Resilient Distributed Datasets (RDDs), a distributed collection of data elements, partitioned across the worker nodes, which enable resilient and parallel operations. In Spark, tasks are composed of transformations, which generate new RDDs/partitions, and actions, which initiate computations. Notably, transformations are lazy, executing only upon an action's request, thus optimizing task flow for efficiency in distributed processing environments. The primary Spark functions and actions utilized are summarized in Tables 1 and 2.

AUTO+PDMS model primary objective is to accelerate the most time-intensive aspect: calculating the fitness of individuals in the population. Within the AUTO+PDMS framework, the Spark master orchestrates all evolutionary tasks. It dispatches the population's individuals to the workers to compute their fitness values. Operating in parallel, these workers evaluate partial fitness values using the data within their local partitions. Subsequently, the master gathers these aggregated results for further processing. The AUTO+PDMS model also

Transformation	Description
map(func)	Return a new distributed dataset formed by passing each element of the source through a function func.
mapPartitions(func)	Similar to map, but runs separately on each partition (block) of the RDD, so func must be of type Iterator<T> => Iterator<U> when running on an RDD.
zipPartitions(RDD2, RDD3, func)	Zip this RDD's partitions with one (or more) RDD(s) and return a new RDD by applying a function func to the zipped partitions.
repartition(numPartitions)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
sample(withReplacement, fraction, seed)	Sample a fraction fraction of the data, with or without replacement, using a given random number generator seed.
filter(func)	Return a new RDD containing only the elements that satisfy a predicate.

Table 1. Spark RDD Transformations.

Action	Description
collect()	Returns all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Returns the number of elements in the dataset.
take(n)	Returns an array with the first n elements of the dataset.
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
aggregate(zeroValue) (seqOp, combOp)	Aggregate the elements of each partition, and then the results for all the partitions, using given combine functions and a neutral "zero value". This function can return a different result type, than the type of this RDD. Thus, we need one operation seqOp for merging the original RDD into a new type RDD and one operation combOp for merging two of the new type RDD.

Table 2. Spark RDD Actions.

replaces fixed iteration numbers and population size with mechanisms for automatic termination and auto-population resizing.

These improvements are comprehensively outlined in Algorithm 2: The AUTO+PDMS BioHEL implementation follows the main structure of BioHEL, with specialized `findRulesPDMS`and `bestRulePDMS`functions. The training set is stored as partitioned collection (RDD) (TrainS, underlined, in Algorithm 2) with instances equally partitioned across workers. In `findRulesPDMS`(lines 12–30), the population is initialized as follows: a sample function is used to derive a sample from TrainS, then `collect`function is used to retrieve the sample from the workers and make it available to the master (line 15). The fitness computation is then parallelized using the `aggregate`function (line 19): a copy of the population is transferred to each worker and used in the partialFitness function to compute the fitness of the population on the local partition of TrainSand the resulting values are aggregated into a global fitness measure at the master (using the mergeFitness function) This change to the findRule function is the primary way in which the BioHEL model is parallelised in the PDMS strategy.

With respect to parameter control, findRulesPDMS is composed by two nested loops, in the outer loop (lines 13– 30) different population sizes are explored sequentially, while in the inner loop (lines 17 – 27) a GA with adaptive termination for the currently selected population size is used to find candidate rules. Initially the population size is set to 10, while `bestFoundCandidate` is initialized to `null` (line 12). The inner GA loop which in the original BioHEL was executed for a fixed number of iterations, is now substituted by loop with a countdown condition (line 17). Although this loop replicates GA evolution processes, it incorporates two additional steps: (1) the computation of the best-found fitness and (2) the calculation of the average fitness of the present generation (lines 21– 22). If any of these values surpass previously computed ones, the superior value(s) replace the previous ones, and the count value is reset to 3 (lines 24– 26). Otherwise, the count value is reduced by one. Eventually, the loop in line 17 terminates when the count value reaches zero, and no improvement is observed in both the best and average fitness values. The best candidate rule assessed in the outer loop which compares it with `bestFoundCandidate` (lines 28– 29). If the candidate rule demonstrates higher fitness than the `bestFoundCandidate` which originated from a smaller population, it will replace the `bestFoundCandidate`, and the population size will be doubled (line 29). If not, the learning process will be terminated setting the `terminate` flag to `true` (line 30), consequently stopping the outer loop (line 13).

```
BioHEL(TrainS):
    ruleList = ∅; stop = false;
    while stop is false do
        for repetition=1 to numRepetitions do
            candidates += findRules(TrainS);
        (bestRule, TrainS) = bestRule(candidates, TrainS);
        if rule not null then
            ruleList += bestRule; candidates = ∅;
        else stop = true;
    return ruleList;
findRulesPDMS(TrainS):
    terminate=false; popSize=10; bestFoundCandidate=$null$;
    while terminate = false do
        count=3;
        pop = TrainS.sample(...).collect();
        pop.fitness = TrainS.aggregate(zeroes,
          partialFitness(pop),mergeFitness);
        while count != 0 do
            offsp = pop.selection(); offsp.crossover(); offsp.mutation() ;
            offsp.fitness=TrainS.aggregate(zeroes,
              partialFitness(offsp), mergeFitness);
            pop.replacement(offsp);
            candidateIndividual=pop.Best();
            currentAverage=pop.computeAverageFitness();
            if candidateIndividual is better than bestIndividual OR
              currentAverage is better than fitnessAverage then
                bestIndividual= takeBest(candidateIndividual,
                  bestIndividual);
                fitnessAverage= takeBest(currentAverage,
                  fitnessAverage);
                count=3
            else count−−;
        if bestIndividual is better than bestFoundCandidate then
            bestFoundCandidate=bestIndividual; popSize=popSize*2
        else terminate=true ;
    return bestFoundCandidate;
bestRulePDMS(candidates,TrainS):
    bestRule = candidates.bestFitness();
    matchedCl = TrainS.filter(_.matches(bestRule))
      .map(_.class).countByValue();
    if bestRule.class = majorityClass(matchedCl) then
        TrainS = TrainS.filter(!_.matches(bestRule));
        return (bestRule, TrainS);
    else return (null, TrainS);
```

Algorithm 2. AUTO+PDMS BioHEL.

Dataset	HEPMASS	KDD-99	HIGGS
No. of instances	10.5 M	4.9 M	11 M
No. of attributes	28 real-valued	41 (26 real-valued & 15 nominal)	28 real-valued
No. of classes	2	23	2
Class distribution	50%	56.85% 21.70% 19.69% (other classes <1.00%)	53% - 47%

Table 3. KDD, HEPMASS, and HIGGS data composition.

```
1  BioHEL(TrainS):
2      ruleList = ∅; stop = false;
3      while stop is false do
4          for repetition=1 to numRepetitions do
5              candidates += findRules(TrainS);
6          (bestRule, TrainS) = bestRule(candidates, TrainS);
7          if rule not null then
8              ruleList += bestRule; candidates = ∅;
9          else stop = true;
10     return ruleList;
11 findRulesPDMD(popSize,TrainS):
12     terminate=false; popSize=10; bestFoundCandidate=null;
13     while terminate = false do
14         pop = ∅; count=3; bestIndividual=null;
15         pop = TrainS.zipPartitions(pop)(
16         (TrainSPart, popPart)⇒
17             if popPart = ∅ then  popPart = TrainSPart.sample(popSize);
18             popPart.fitness=doFitness(popPart, TrainSPart);
19             while count != 0 do
20                 offsp = popPart.selection();
21                 offsp.crossover();
22                 offsp.mutation();
23                 offsp.fitness=doFitness(offsp,popPart);
24                 popPart.replacement(offsp);
25                 candidateIndividual=Best(popPart);
26                 currentAverage=computeAverageFitness(popPart);
27                 if candidateIndividual is better than bestIndividual OR
                     currentAverage is better than fitnessAverage then
28                     bestIndividual= takeBest(candidateIndividual,
                         bestIndividual);
29                     fitnessAverage= takeBest(currentAverage,
                         fitnessAverage);
30                     count=3
31                 else count−−;
32                 return popPart
33         );
34         candidates = pop.mapPartitions(best(1)).collect();
35         candidates.fitness=TrainS.aggregate(zeroes,
             partialFitness(candidates), mergeFitness);
36         bestIndividual = candidates.bestFitness();
37         if bestIndividual is better than bestFoundCandidate then
38             bestFoundCandidate=bestIndividual; popSize=popSize*2
39         else terminate=true ;
40     return bestFoundCandidate;
41 bestRulePDMD(candidates,TrainS):
42     bestRule = candidates.bestFitness();
43     matchedCl = TrainS.filter(_.matches(bestRule))
         .map(_.class).countByValue();
44     if bestRule.class = majorityClass(matchedCl) then
45         TrainS = TrainS.filter(!_.matches(bestRule));
46         return (bestRule, TrainS);
47     else return (null, TrainS);
```

Algorithm 3. AUTO+PDMD BioHEL.

AUTO+PDMD In the context of AUTO+PDMD, both the population of solutions popand the training instances TrainSare partitioned across workers where each worker runs independently a genetic algorithm evolving the local population of popusing the local training instances from TrainSin parallel. The difference between the AUTO+PDMS and AUTO+PDMD is the implementation of the `findRulesPDMD`function as shown in in Algorithm 3. Initially, each worker initializes its sub-population (`popPart`) by sampling from its partition of the training data (`TrainSPart`) (line 17). This approach of localized intialisation and evolution enable each worker to explore different regions of the solution space, typical of island-based evolutionary approaches[2]. Similar to AUTO+PDMS, the fixed iteration numbers and the fixed population size are replaced with automatically selected values. Each worker locally employs the countdown condition (referenced in line 19) and iteratively undergoes the GA evolution processes. This step in AUTO+PDMD is accompanied by two additional steps: the computation of best fitness (line 25)and the calculation of the current generation's average fitness (line 26), similar to the procedure followed in AUTO+PDMS. Upon the countdowns reaching zero, signaling the termination of all local GAs, the `zipPartitions`Spark function then updates pop(global population) as the result of all the subpopulation in each partition (as indicated in line 15). Following this, each local best individual (or rule) undergoes a global reevaluation (line 34), after which it is compared to the `bestFoundCandidate` previously identified. This comparison informs the subsequent decision to either duplicate the size of the local population and repeat the entire learning process or terminate it by setting the `terminate` flag to `true`. Note that the `bestRulePDMD`function works exactly as in AUTO+PDMS.

Parameter	Value
crossover prob. Rule sets per ensemble	0.6
Selection algorithm	tournament
Tournament size	4
Individual-wise mutation prob.	0.6
Default class policy	MAJOR
Expected value of #expressed att. in init.	15
Repetitions of rule learning process	2
Prob. generalize	0.1
Prob. specialise	0.1

Table 4. General Parameters of BioHEL.

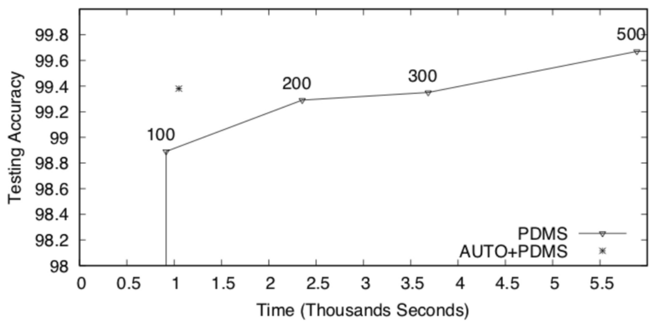

Fig. 1. Accuracy/Time Tradeoff for AUTO+PDMS and Original PDMS at Different Population Sizes on the KDD Dataset. This graph compares the efficiency of AUTO+PDMS, which achieves the accuracy level of PDMS with a population size of 300, within the time needed for PDMS at a population size of 100.

Experimental study

The experimental design aims to estimate the performance of the AUTO+PDMS and AUTO+PDMD models

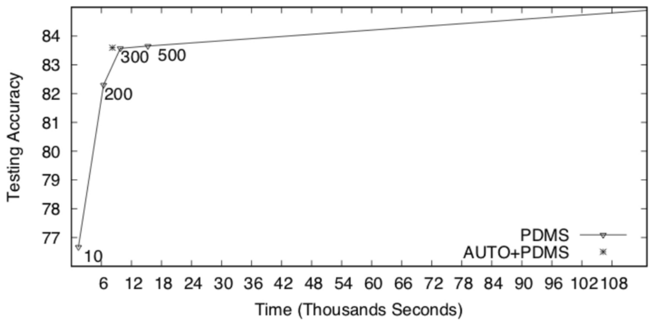

Fig. 2. Accuracy/Time Tradeoff for AUTO+PDMS and Original PDMS at Different Population Sizes on the HEPMASS Dataset. AUTO+PDMS achieves accuracy comparable to PDMS with population sizes of 300 and 500 while operating in a time frame close to PDMS with a population size of 300.

presented in this study. Specifically, we investigate the influence of automatic termination and population resizing on accuracy and execution time. To evaluate the efficiency of these automated implementations, we employ the HEPMASS dataset, (https://archive.ics.uci.edu/ml/datasets.html.) the KDD-cup99 (full version) dataset, (https://www.openml.org/d/1110.), and the Higgs dataset[1]. Table 3 shows the datasets composition.

The experiments were executed on a cluster composed of 17 servers, each equipped with 2x Intel Xeon E5520 CPUs running at 2.27GHz. Each server possesses 8 cores and 12GB RAM, with the cluster size fixed at 96 executors. We report the average results over 20 runs, utilizing 10-fold cross-validation and 95% confidence intervals.

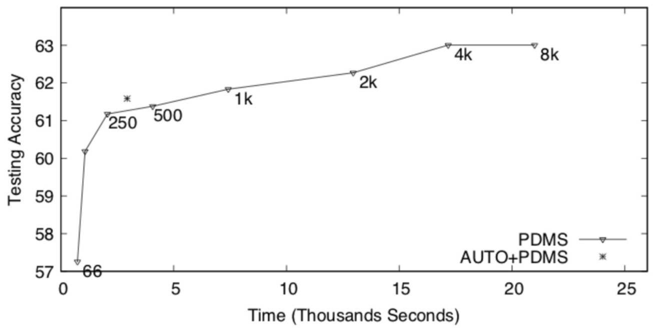

Fig. 3. Accuracy/Time Tradeoff for AUTO+PDMS and Original PDMS at Different Population Sizes on the HIGGS Dataset. This graph shows AUTO+PDMS achieving accuracy equivalent to PDMS at a population size of 1,000, but within the shorter time frame typical of PDMS population sizes between 250 and 500.

	popSize	Accuracy	Time
PDMS	10	21.57 ± 7.0	165 ± 20
	20	30.10 ± 7.0	282 ± 215
	100	98.89 ± 0.6	915 ± 100
	200	99.29 ± 1.0	2,350 ± 100
	300	99.35 ± 0.4	3,680 ± 100
	500	99.68 ± 0.2	5,800 ± 80
	2K	99.68 ± 0.0	40,900 ± 680
AUTO+PDMS		99.37 ± 0.0	1,040 ± 93
PDMD	20	21.56 ± 0.0	55 ± 2
	500	93.20 ± 1.0	260 ± 15
	960	98.68 ± 0.0	590 ± 60
	2K	99.01 ± 0.0	630 ± 20
	4K	99.03 ± 0.0	650 ± 15
	8K	99.20 ± 0.0	800 ± 50
	16K	99.32 ± 0.0	1,790 ± 160
	32K	99.32 ± 0.0	4,550 ± 300
AUTO+PDMD		99.23 ± 0.0	540 ± 12

Table 5. Performance metrics of AUTO+PDMS, AUTO+PDMD, and PDMS & PDMD at varying population sizes (KDD dataset).

In the subsequent section, we compare the performance of the AUTO+PDMS model to the original model results, where the same number of nodes were utilized. Subsequently, we compare the AUTO+PDMD model to the original version and the original PDMS for comparative purposes.

For the BioHEL configuration, we set the algorithm parameters using the same configuration provided in[4], summarised in Table 4.

AUTO+PDMS performance

This section evaluates the performance of AUTO+PDMS compared to the original PDMS across three datasets: KDD, HEPMASS, and HIGGS. The evaluation focuses on the trade-off between accuracy and execution time, with results summarized in Figures 1, 2, 3 and Tables 5, 6, 7.

For the KDD dataset, as shown in Figure 1 and Table 5, the PDMS model was executed with population sizes of 10, 20, 200, 300, 500, and 2,000. Results for population sizes of 10 and 20 are excluded due to poor accuracy (below 50%). AUTO+PDMS achieved an average accuracy of $99.37 \pm 0.0\%$ in $1,040 \pm 93$ seconds, compared to the original PDMS model, which attained its peak accuracy of $99.68 \pm 0.0\%$ at a population size of 500, requiring $5,800 \pm 80$ seconds. While the original PDMS achieved slightly higher accuracy at larger population sizes, AUTO+PDMS demonstrated statistically significant time savings ($p < 0.05$). This efficiency can be attributed to the Incremental Rule Learning (IRL) approach, where most rules in AUTO+PDMS were learned with a population size of 80, with larger sizes used occasionally for more complex rules. This adaptive

	popSize	Accuracy	Time
PDMS	10	76.67 ± 1.0	1409 ± 200
	200	82.30 ± 0.3	6,434 ± 50
	300	83.57 ± 0.2	9,782 ± 150
	500	83.65 ± 0.2	16,500 ± 250
	2K	84.90 ± 0.1	116,047 ± 600
AUTO+PDMS		83.59 ± 0.1	8,200 ± 500
PDMD	500	78.67 ± 0.6	900 ± 200
	960	79.89 ± 0.4	1,100 ± 400
	2K	80.70 ± 0.3	1,432 ± 300
	4K	81.59 ± 0.3	1,600 ± 250
	6K	82.27 ± 0.2	1,910 ± 320
	8K	82.71 ± 0.2	2,612 ± 300
	16K	83.94 ± 0.0	5,009 ± 250
	32K	84.50 ± 0.0	10,239 ± 500
AUTO+PDMD		83.43 ± 0.5	6,210 ± 620

Table 6. Performance metrics of AUTO+PDMS, AUTO+PDMD, and PDMS & PDMD at varying population sizes (HEPMASS dataset).

	popSize	Accuracy	Time
PDMS	66	57.25 ± 1.0	733 ± 170
	126	60.18 ± 0.7	1,071 ± 300
	250	61.18 ± 0.3	2,061 ± 158
	500	61.38 ± 0.3	4,075 ± 450
	1K	61.83 ± 0.2	7,425 ± 620
	2K	62.27 ± 0.3	12,961 ± 1,220
	4K	63.00 ± 0.2	17,179 ± 1,600
	8K	63.00 ± 0.2	21,006 ± 2,000
AUTO+PDMS		61.59 ± 0.2	2,900 ± 690
PDMD	500	53.12 ± 0.0	116 ± 20
	1K	54.09 ± 0.4	148 ± 16
	2K	57.48 ± 0.7	225 ± 30
	4K	57.42 ± 0.8	299 ± 120
	8K	60.64 ± 0.7	503 ± 100
	16K	61.24 ± 1.0	951 ± 300
	32K	61.85 ± 0.3	2014 ± 100
	64K	61.94 ± 0.2	3,626 ± 260
	128K	62.01 ± 0.1	11,523 ± 640
AUTO+PDMD		61.57 ± 0.4	3,440 ± 400

Table 7. Performance metrics of AUTO+PDMS, AUTO+PDMD, and PDMS & PDMD at varying population sizes (Higgs dataset).

mechanism enabled AUTO+PDMS to balance exploration and exploitation effectively, reducing computational overhead without significantly compromising accuracy.

For the HEPMASS dataset, Figure 2 and Table 6 show the performance of AUTO+PDMS compared to the original PDMS model, evaluated with population sizes of 10, 200, 300, 500, and 2,000. AUTO+PDMS achieved an average accuracy of $83.59 \pm 0.1\%$ in $8,200 \pm 500$ seconds, comparable to the original PDMS model's accuracy of $83.65 \pm 0.2\%$ at a population size of 500, which required $16,500 \pm 250$ seconds. Although AUTO+PDMS fell short of the highest accuracy achieved by PDMS at a population size of 2,000 ($84.90 \pm 0.1\%$), it delivered competitive accuracy in nearly half the execution time. Statistical analysis confirmed significant time savings for AUTO+PDMS compared to PDMS-500 ($p < 0.01$), highlighting its efficiency. This performance reflects the adaptability of the dynamic resizing mechanism, which minimizes redundant computations while maintaining strong classifier performance on high-dimensional datasets like HEPMASS.

For the HIGGS dataset, as illustrated in Figure 3 and Table 7, PDMS was executed with population sizes ranging from 66 to 8,000. AUTO+PDMS achieved an average accuracy of $61.59 \pm 0.2\%$ in $2,900 \pm 690$ seconds, compared to the original PDMS model, which reached its peak accuracy of $63.00 \pm 0.2\%$ at a population size of 8,000, requiring $21,006 \pm 2,000$ seconds. While AUTO+PDMS slightly underperformed compared to PDMS at larger population sizes, it delivered competitive accuracy with significantly reduced execution time. For instance, its execution time was closer to PDMS with a population size of 500 ($4,075 \pm 450$ seconds), while its accuracy exceeded that of PDMS-500 ($61.38 \pm 0.3\%$). Statistical analysis confirmed that the time savings provided by AUTO+PDMS were significant ($p < 0.05$), demonstrating its suitability for large-scale datasets. By dynamically tailoring population sizes to rule complexity, AUTO+PDMS minimized computational overhead while maintaining a competitive level of accuracy.

Overall, AUTO+PDMS consistently demonstrated a favorable trade-off between accuracy and execution time across all datasets. By leveraging dynamic population resizing, it achieved competitive accuracy levels with significantly reduced computational effort. These findings underscore the robustness and versatility of AUTO+PDMS in addressing diverse dataset challenges, including class imbalance, high dimensionality, and large scale.

AUTO+PDMD performance

In this section, we present the evaluation of the automated PDMD BioHEL model and compare it with both the original PDMS and PDMD models. Figure 4 illustrates the performance in terms of accuracy versus time for the KDD dataset. The PDMD model was executed with population sizes of 100, 200, and 300, while the graph also showcases the performance of the original PDMS with population sizes of 20, 500, 960, and 2K for comparison purposes. The results demonstrate that the AUTO+PDMD offers a superior trade-off compared to both original models. Specifically, the AUTO+PDMD model achieves a similar level of accuracy to the original PDMD model in only 50% of the execution time as shown in Table 5. Additionally, during our investigation of the AUTO+PDMD runs on the KDD dataset, we observed that the population size of the best-selected rule exceeded 10 per island (i.e., 960 in total) for only a minority of the learned rules, averaging approximately 25% of them.

Moving on to the HEPMASS dataset, Figure 5 presents the accuracy versus time performance. We first report the previous results of PDMS with different population sizes (10, 200, 300, and 500). Subsequently, we executed the PDMD model with population sizes of 500, 960, 2K, 4K, 8K, 16K, and 32K. The graph indicates that the AUTO+PDMD model achieves an average accuracy of 83.43%, which falls between the accuracy levels of PDMD-8K and PDMD-16K, albeit with a longer execution time compared to PDMD-16K. Although PDMD-16K provides a better trade-off than AUTO+PDMD, the latter model delivers comparable results without requiring prior knowledge of the ideal population size.

In Figure 6, we present the performance evaluation (accuracy vs. time) for the Higgs dataset. The PDMD model was run with different population sizes (500, 1K, 2K, 4K, 8K, 16K, 32K, 64K & 128K), and the AUTO+PDMD model was also evaluated. The accuracy and learning time of the AUTO+PDMD model were found to be comparable to those of the original PDMD model. However, looking at the average accuracy across all runs, it was bounded by the confidence interval, with the lowest recorded at 61.17%. This accuracy level is situated between that of PDMD-8K and PDMD-16K, while its learning time is longer, falling between the learning times of PDMD-32K and PDMD-64K. Despite AUTO+PDMD achieving a respectable accuracy, it experienced a considerably high execution time. However, this was still less than the cumulative execution time for all tested population sizes of PDMD, ranging from 500 to 32K. For further details, please refer to Table 7.

Discussion

Our results lead to several conclusions regarding the performance of the proposed AUTO+PDMS and AUTO+PDMD models compared to their original counterparts. Firstly, the performance of AUTO+PDMS was consistently higher across a range of population sizes, demonstrating its ability to provide a better trade-off between computational efficiency and accuracy. This improvement is attributed to the variable population size utilized by the automated BioHEL, which adapts to the complexity of the rules being learned. Specifically, when learning a complex rule, the larger population sizes in AUTO+PDMS facilitate more effective exploration of the search space. Conversely, smaller populations are sufficient for simpler rules, reducing computational time. In

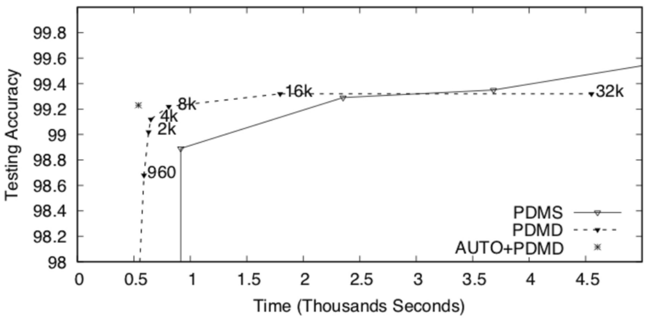

Fig. 4. Accuracy/Time tradeoff for AUTO+PDMD and Original PDMS & PDMD at different population sizes (KDD Dataset). The graph demonstrates that the AUTO+PDMD model achieves a similar level of accuracy to the original PDMD model in only 50% of the execution time.

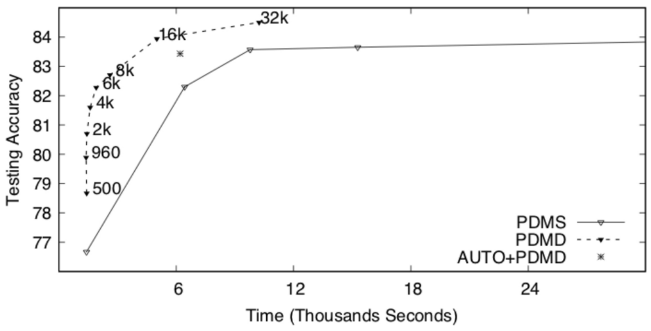

Fig. 5. Accuracy/Time tradeoff for AUTO+PDMD and Original PDMS & PDMD at different population sizes (HEPMASS Dataset). This graph indicates that the AUTO+PDMD model achieves an average accuracy of 83.43%, positioning it between the accuracy levels of PDMD-8K and PDMD-16K, but with a longer execution time compared to PDMD-16K.

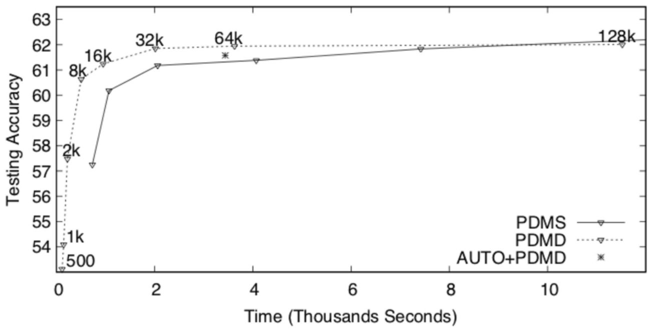

Fig. 6. Accuracy/Time tradeoff for AUTO+PDMD and Original PDMS & PDMD at different population sizes (Higgs Dataset). The graph shows that while the accuracy of the AUTO+PDMD model is comparable to that of the original PDMD model, its execution time falls between the learning times of PDMD-32K and PDMD-64K.

contrast, the original PDMS algorithm uses a fixed population size for all rules, leading to inefficiencies in both scenarios.

However, while the advantages of our approach in terms of computational efficiency and user-friendliness are clear, it is crucial to keep in mind the accompanying reality that sometimes, the final solution accuracy may suffer slightly. This is particularly observable in our study as the performance improvements were not uniformly consistent across all datasets. While AUTO+PDMS achieved a better trade-off between accuracy and execution time on datasets such as KDD and HEPMASS, its performance on the HIGGS dataset was less pronounced. This variation suggests that the effectiveness of the automated strategy is influenced by dataset-specific characteristics, such as dimensionality, noise, and rule complexity. These findings underline the importance of tailoring automated strategies to the properties of individual datasets to achieve optimal results, and they highlight the inherent trade-off that must be managed when choosing to automate parameter selection.

As for the AUTO+PDMD model, scaling up the original PDMD to a large cluster size required significantly larger populations to achieve acceptable accuracy. By contrast, AUTO+PDMD achieved comparable performance to the optimal configuration of the original model without requiring manual tuning of the population size. Moreover, AUTO+PDMD demonstrated the ability to leverage smaller population sizes when learning simpler rules, ultimately resulting in reduced computational costs. However, similar to AUTO+PDMS, its reliance on dynamic adjustments may not always yield the highest accuracy, particularly in datasets with high noise levels or extreme class imbalances.

Overall, our study demonstrates that the proposed extensions to PDMS and PDMD models significantly enhance their ability to achieve high accuracy within reasonable time frames. The automation of parameter selection, including population size and number of iterations, reduces the dependency on trial-and-error tuning. Nonetheless, it is essential to acknowledge the limitations of the proposed approach: the automated models may not always achieve the highest possible accuracy due to the reliance on convergence conditions that prioritize efficiency over exact genotype convergence. Additionally, their performance is influenced by dataset characteristics, which may limit their generalizability to datasets with high noise levels, extreme class imbalances, or fundamentally different distributions.

These findings suggest promising avenues for future research. Specifically, there is a need to refine convergence criteria to better handle diverse data scenarios and investigate the models' applicability to a broader range of datasets. Furthermore, exploring the impact of dataset-specific properties, such as noise and rule complexity, on the models' performance could provide deeper insights into their strengths and limitations, helping to balance efficiency and accuracy in future applications.

Limitations

While this study demonstrates the effectiveness of automated termination and population resizing strategies in improving the performance of PDMS and PDMD models, several limitations should be acknowledged:

Performance vs. accuracy trade-off

The automated models prioritize a balance between computational efficiency and accuracy. In cases where achieving the highest possible accuracy is the primary goal, these models may not always provide the best solution, especially for datasets with challenging characteristics.

Dataset scope and generalizability

The models were evaluated on three datasets: KDD Cup 1999, HEPMASS, and HIGGS, each with distinct characteristics that influenced the outcomes of this study. The KDD Cup 1999 dataset is highly imbalanced, with the majority class dominating the distribution, posing challenges for the models in detecting minority classes effectively. The automated approaches demonstrated adaptability in handling the imbalance, though further testing on datasets with even greater disparities is needed to generalize these findings. The HEPMASS dataset, characterized by high dimensionality and balanced classes, provided a suitable benchmark for evaluating the scalability and feature-handling capabilities of the models. The automated strategies performed well on this dataset by dynamically adjusting population sizes to explore the search space efficiently; however, datasets with higher levels of irrelevant or redundant features may present additional challenges. The HIGGS dataset, both large and complex, includes features that require extensive interactions for effective classification. While the automated strategies achieved competitive results on this dataset, the performance improvements were less pronounced compared to the other datasets, suggesting that the automated termination and population resizing mechanisms may be less effective for datasets with subtle distinctions between classes. Although these datasets represent a range of characteristics, including class imbalance, high dimensionality, and large scale, they do not encompass all possible scenarios. For example, datasets with significant noise, extreme sparsity, or fundamentally different structures, such as text or time-series data, may present additional challenges. Future work should evaluate the proposed models across more diverse datasets to better assess their generalizability and robustness.

Population distribution and rule complexity

The performance of the automated models may vary depending on the initial population distribution and the complexity of the rules being learned. For instance, datasets requiring highly complex rules may challenge the automated termination and resizing mechanisms, potentially limiting their effectiveness. Further studies are needed to explore these scenarios in greater detail.

Impact of noise and imbalance

The current study does not explicitly evaluate the models on datasets with high levels of noise or severe class imbalance. These factors could affect the models' ability to converge on meaningful solutions. Future work should focus on testing and refining the models to handle such challenging datasets effectively.

Conclusion

In conclusion, this study introduced enhancements to parallel genetic algorithms by implementing automated termination and population resizing mechanisms within the AUTO+PDMS and AUTO+PDMD frameworks. These advancements aim to balance computational efficiency and classifier accuracy, particularly for large-scale data analysis. Building on the dynamic population increase policy proposed in [31], we adapted it to address challenges specific to iterative rule learning (IRL), as the infinite solution advocated in [31] was not suitable for sequential rule learning. While these automated methods significantly reduce the dependency on manual trial-and-error parameter tuning, they do not entirely eliminate the need for human oversight. The results are encouraging but context-dependent, varying based on the datasets and specific scenarios involved.

Our experiments demonstrated that the automated models consistently outperformed their original counterparts in terms of time efficiency and adaptability. For instance, AUTO+PDMS achieved competitive accuracy and significantly reduced computational time on the KDD dataset, while AUTO+PDMD excelled on the HEPMASS dataset by delivering faster execution times with comparable accuracy. However, these improvements were not universal, as the performance on the HIGGS dataset was less pronounced. This highlights the importance of considering dataset-specific characteristics, such as dimensionality and noise levels, when applying these strategies.

Despite these promising results, certain limitations remain. The automated termination and population resizing methods, while effective, rely on convergence criteria that may not always yield optimal results in highly complex or noisy datasets. Future research may concentrate on developing more sophisticated and adaptive convergence criteria that can dynamically adjust to the complexity and diversity of different datasets, thereby improving both the robustness and generalizability of these models. This involves creating algorithms that can more accurately detect when optimization has reached a point of diminishing returns in terms of accuracy, thus preventing overfitting while maximizing computational efficiency. Additionally, expanding the application of these techniques beyond iterative rule learning to include other genetic algorithm paradigms, such as the Pittsburgh and Michigan approaches[34], offers a promising direction. Further exploration into other problem domains beyond classification[35], including optimization and feature selection, could significantly broaden the impact and applicability of our findings. Such advancements would not only refine the trade-offs inherent in current models but also enhance their practical utility in more complex, real-world scenarios where flexibility and adaptability are crucial.

By addressing these challenges and introducing a practical framework for automated parameter control, this study contributes to advancing the usability and efficiency of evolutionary computation techniques in big data analytics.

Data availability

The datasets generated and/or analyzed during the current study are available from the following sources: - KDD-cup99 (full version) and HIGGS datasets: https://archive.ics.uci.edu/ml/datasets.html - HEPMASS datase t: https://www.openml.org/d/1110 These datasets are publicly accessible. Alternatively, the data that support the findings of this study are available from the corresponding author, Laila Al-Terkawi, upon reasonable request.

Received: 22 July 2024; Accepted: 10 March 2025
Published online: 29 March 2025

References

1. Alba, E. & Troya, J. M. A survey of parallel distributed genetic algorithms. *Complex.* **4**(4), 31–52 (1999).
2. Cantú-Paz, E. "A survey of parallel genetic algorithms," *CALCULATEURS PARALLELES*, vol. 10, (1998).
3. Alterkawi, L. & Migliavacca, M."Parallelism and partitioning in large-scale gas using spark," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '19. New York, NY, USA: Association for Computing Machinery, pp. 736–744 (2019). [Online]. Available: https://doi.org/10.1145/3321707.3321775
4. Bacardit, J. & Krasnogor, N. *BioHEL: Bioinformatics-oriented Hierarchical Evolutionary Learning*, (2006). [Online]. Available: http://eprints.nottingham.ac.uk/id/eprint/482
5. Maulik, U. & Bandyopadhyay, S. Genetic algorithm-based clustering technique. *Pattern Recognition* **33**(9), 1455–1465 (2000). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0031320399001375
6. Murty, M. N., Rashmin, B. & Bhattacharyya, C. *Clustering Based on Genetic Algorithms*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 137–159 (2008). [Online]. Available: https://doi.org/10.1007/978-3-540-77467-9_7
7. Paterlini, S. & Minerva, T. "Regression model selection using genetic algorithms," in *Proceedings of the 11th WSEAS International Conference on Nural Networks and 11th WSEAS International Conference on Evolutionary Computing and 11th WSEAS International Conference on Fuzzy Systems*, ser. NN'10/EC'10/FS'10. Stevens Point, Wisconsin, USA: World Scientific and Engineering Academy and Society (WSEAS), pp. 19–27 (2010).
8. Baragona, R., Battaglia, F. & Calzini, C. Genetic algorithms for the identification of additive and innovation outliers in time series. *Computational Statistics & Data Analysis* **37**(1), 1–12 (2001).
9. Minaei-Bidgoli, B. & Punch, W. F. "Using genetic algorithms for data mining optimization in an educational web-based system," in *Genetic and Evolutionary Computation — GECCO 2003*, E. Cantú-Paz, J. A. Foster, K. Deb, L. D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. A. Dowsland, N. Jonoska, and J. Miller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 2252–2263 (2003).
10. Cervantes, J., Li, X. & Yu, W. "Using genetic algorithm to improve classification accuracy on imbalanced data," in *2013 IEEE International Conference on Systems, Man, and Cybernetics*, pp. 2659–2664 (2013).
11. Dam, H. H., Abbass, H. A. & Lokan, C. J. "Dxcs: an xcs system for distributed data mining," in *GECCO*, (2005).
12. O'Reilly, U.-M., Wagy, M. & Hodjat, B. *EC-Star: A Massive-Scale, Hub and Spoke, Distributed Genetic Programming System*. New York, NY: Springer New York, pp. 73–85 (2013). [Online]. Available: https://doi.org/10.1007/978-1-4614-6846-2_6
13. Herrera, F., Lozano, M. & Verdegay, J. L. Tackling real-coded genetic algorithms: Operators and tools for behavioural analysis. *Artificial Intelligence Review* **12**(4), 265–319. https://doi.org/10.1023/A:1006504901164 (1998).
14. Oppacher, F. & Wineberg, M. "Improving the behavior of the genetic algorithm in a dynamic environment," (2000).
15. Pandey, H. M., Chaudhary, A. & Mehrotra, D. A comparative review of approaches to prevent premature convergence in ga. *Applied Soft Computing* **24**, 1047–1077 (2014). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1568494614003901
16. López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Birattari, M. & Stützle, T. "The irace package: Iterated racing for automatic algorithm configuration," *Operations Research Perspectives*, vol. 3, pp. 43–58, (2016). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214716015300270
17. Hutter, F., Hoos, H. H., Leyton-Brown, K. & Stützle, T. Paramils: An automatic algorithm configuration framework. *J. Artif. Int. Res.* **36**(1), 267–306 (2009).
18. Hutter, F., Hoos, H. H. & Leyton-Brown, K. "Sequential model-based optimization for general algorithm configuration," in *International conference on learning and intelligent optimization*. Springer, pp. 507–523 (2011).
19. Leung, Y.-W. & Wang, Y. An orthogonal genetic algorithm with quantization for global numerical optimization. *IEEE Transactions on Evolutionary Computation* **5**(1), 41–53 (2001).
20. Venturini, G. "Sia: A supervised inductive algorithm with genetic search for learning attributes based concepts," in *Machine Learning: ECML-93*, P. B. Brazdil, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 280–296 (1993).
21. Gibbs, M., Maier, H., Dandy, G. & Nixon, J. "Minimum number of generations required for convergence of genetic algorithms," in. *IEEE International Conference on Evolutionary Computation* **2006**, 565–572 (2006).
22. Ghoreishi, S. N., Clausen, A. & Jørgensen, B. N. "Termination criteria in evolutionary algorithms: A survey," in *IJCCI*, (2017).
23. Safe, M., Carballido, J., Ponzoni, I. & Brignole, N. "On stopping criteria for genetic algorithms," in *Advances in Artificial Intelligence – SBIA 2004*, A. L. C. Bazzan and S. Labidi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 405–413 (2004).
24. Greenhalgh, D. & Marshall, S. Convergence criteria for genetic algorithms. *SIAM J. Comput.* **30**(1), 269–282. https://doi.org/10.1137/S009753979732565X (2000).
25. Safe, M., Carballido, J., Ponzoni, I. & Brignole, N. "On stopping criteria for genetic algorithms," in *Advances in Artificial Intelligence – SBIA 2004*, A. L. C. Bazzan and S. Labidi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 405–413 (2004).
26. Goldberg, D. E. "Sizing populations for serial and parallel genetic algorithms," in *Proceedings of the 3rd International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 70–79 (1989). [Online]. Available: http://dl.acm.org/citation.cfm?id=645512.657266
27. Aleti, A. & Moser, I. A systematic literature review of adaptive parameter control methods for evolutionary algorithms. *ACM Comput. Surv.* **49**(3), 56:1-56:35. https://doi.org/10.1145/2996355 (2016).
28. Arabas, J., Michalewicz, Z. & Mulawka, J. Gavaps-a genetic algorithm with varying population size. *in Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence* **1**, 73–78 (1994).

29. Bäck, T., Eiben, A. E. & van der Vaart, N. A. L. "An emperical study on gas without parameters," in *Parallel Problem Solving from Nature PPSN VI*, M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. J. Merelo, and H.-P. Schwefel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 315–324 (2000).
30. Eiben, A. E., Marchiori, E. & Valkó, V. A. "Evolutionary algorithms with on-the-fly population size adjustment," in *Parallel Problem Solving from Nature - PPSN VIII*, X. Yao, E. K. Burke, J. A. Lozano, J. Smith, J. J. Merelo-Guervós, J. A. Bullinaria, J. E. Rowe, P. Tiňo, A. Kabán, and H.-P. Schwefel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 41–50 (2004).
31. Harik, G. R. & Lobo, F. G. "A parameter-less genetic algorithm," in *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 1*, ser. GECCO'99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 258–265 (1999).
32. Krishnakumar, K. "Micro-Genetic Algorithms For Stationary And Non-Stationary Function Optimization," in *Intelligent Control and Adaptive Systems*, G. Rodriguez, Ed., vol. 1196, International Society for Optics and Photonics. SPIE, pp. 289 – 296 (1990). [Online]. Available: https://doi.org/10.1117/12.969927
33. Zaharia, M. et al. Apache spark: A unified engine for big data processing. *Commun. ACM* **59**(11), 56–65. https://doi.org/10.1145/2934664 (2016).
34. De Jong, K. A., Spears, W. M. & Gordon, D. F. Using genetic algorithms for concept learning. *Mach. Learn.* **13**(23), 161–188. https://doi.org/10.1007/BF00993042 (1993).
35. Whitley, D. A genetic algorithm tutorial. *Statistics and Computing* **4**(2), 65–85. https://doi.org/10.1007/BF00175354 (1994).

Author contributions

Laila Al-Terkawi: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data Curation, Writing - Original Draft, Writing - Review & Editing, Visualization, Project Administration. Matteo Migliavacca: Methodology, Software, Validation, Formal analysis, Writing - Review & Editing, Supervision.

Declarations

Competing interests

The authors declare no competing interests.

Additional information

Correspondence and requests for materials should be addressed to L.A.-T.

Reprints and permissions information is available at www.nature.com/reprints.