SCALING GENETIC ALGORITHMS TO LARGE DISTRIBUTED DATASETS

A THESIS SUBMITTED TO

THE UNIVERSITY OF KENT

IN THE SUBJECT OF COMPUTER SCIENCE

FOR THE DEGREE

OF PHD.

By

Laila Alterkawi

September 2021

Abstract

Analysing large-scale data brings promises of new levels of scientific discovery and economic value. However, the fact that such volume of data is by its nature distributed and the need for new computational methods to be effective in the face of significant changes in data complexity and size has led to the need to develop large-scale data analytics. Genetic algorithms (GAs) have proven their flexibility in many application areas, and substantial research has been dedicated to improving their performance through parallelisation. In contrast with most previous efforts, we reject approaches based on the centralisation of data in the main memory of a single node or requiring remote access to shared/distributed memory. We focus instead on scenarios where data is partitioned across machines.

In this partitioned scenario, we explore two parallelisation models: PDMS, inspired by the traditional master-slave model, and PDMD, based on island models. We adopt the two models to distribute BioHEL, a popular large-scale single-node GA classifier, using the Spark distributed data processing platform. We investigate the effect of GA control parameters (population size and migration frequency).We study the accuracy, time performance and scalability of the proposed models. Our results show that our distributed genetic algorithm design provides a good tradeoff between accuracy and time.

We then extend the two models using automatic termination and population sizing to enhance the distributed genetic algorithm ease-of-use. Moreover, after testing this strategy on both models, we show that the applied automation offers a promising enhancement on the performance of the initially designed GA models.

Acknowledgements

Writing this dissertation took me much longer than I initially estimated and required countless sacrifices. When I first started, writing lengthy acknowledgements seemed pretentious to me. However, as time passed, I became indebted to many people, and I am happy to be able to acknowledge them now. It is customary to thank your committee members first, then mention all people associated with the dissertation and finish with your beloved one. I am not following this order. While I understand that this dissertation is an academic achievement, studying was not only a scientific experience but a life experience in which personal aspects played a significant role. Therefore, thanks go first to my parents: First for bringing me into this world, I find life a very interesting experience! Second, and more importantly, for bringing me up and giving me all that they have. It is intimidating to see how little you leave for yourselves. I thank my spouse, Fadi Dib. I admire how patient, believing and forgiving you were throughout this time. Thank you for loving me. Together with my husband, I need to thank my mother in law "Fatimah" for being a great mom for both me and my little chicks. Thank my little Farah for being the fun part of my PhD life and my new Baby Yahya, who joined us during the Covid-19 pandemic. I also wish to thank Dr Mohammad Alfailakawi for being a supportive and caring Professor. He gave me the confidence to continue pursuing my dream of earning my

PhD.

Switching to the academic side, I would like to thank Dr Matteo Migliavacca, my advisor, for keeping me as a student and tolerating my "last minute attitude". I appreciate your unconditional logic and precise criticism, which were intended to make me a better scientist. I hope I have improved at least a bit. Special thanks go to Dr Alex Freitras and Dr Fernando Otero. While we might have not always agreed, I think I became a more mature person thanks to you. I also want to thank Dr Sally Fincher for serving on my committee prior to her retirement.

Thanks to all who have supported me with a prayer, a word, or even a smile.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Overview Genetic Algorithms (GAs) are used to solve optimisation problems based on their population-based search nature, which gives them great potential for solving large and complex problems where single-solution-based algorithms may fail. On the other hand, GAs are relatively expensive as they involve a more comprehensive search of the problem space using an iterative process over a population (multiple solutions). There have therefore been substantial efforts made to enhance the performance of GAs on parallel architectures, which started in the 80s and led to many successful implementations capable of reducing the time needed to obtain good results.

However, as GAs are applied to different optimisation disciplines, most of these proposals focus on time and quality improvements and do not necessarily consider the need for processing large volumes of data. The frequent assumption is that the dataset is readily available at each worker node through replication or shared/distributed memory, which is impractical or inefficient for vast datasets. Our work in this field addresses distribution of the Parallel GA (PGA) over a cluster of nodes

where large data is distributed and only local data access is allowed.

1.1 Motivation

Large-scale data analytics is becoming increasingly important in the academic and business sectors as it helps us to understand scientific phenomena and supports effective decision making. The effective use of large data has the potential to transform economies, delivering a new wave of productivity growth and consumer surplus. Extracting knowledge from such volumes of data is challenging, however. As the size of data produced increases rapidly, storage and processing requirements quickly grow beyond the capabilities of centralised solutions, which become prohibitively expensive as they require distributed infrastructure. This has fostered the development of large-scale parallel computing platforms such as those provided by Amazon and Google, which promise affordable, reliable and scalable storage capacity and processing power. In addition, scalable analytics requires algorithmic techniques that could benefit from these infrastructures.

Data mining is a branch of artificial intelligence (AI) that focuses on building applications that extract knowledge from a massive volume of structured and unstructured data that is so large that it is difficult to process using traditional database and software techniques.

Evolutionary algorithms, and genetic algorithms (GAs) in particular, have been widely used in a variety of data mining tasks such as clustering (Maulik and Bandyopadhyay 2000; Murty, Rashmin and Bhattacharyya 2008), regression (Paterlini and Minerva 2010), time-series (Baragona, Battaglia and Calzini 2001), enhancing the accuracy of classification algorithms (Minaei-Bidgoli and Punch 2003; Cervantes, Li and Yu 2013), and in particular classification (Dam, Abbass and Lokan 2005;

O'Reilly, Wagy and Hodjat 2013).

GAs offer an attractive solution for large data mining problems for multiple reasons: (i) They are population-based search techniques that give great potential for solving large and complex problems, as they offer a high chance of escaping from local optima. Moreover, they help to provide a broader and more efficient search of the problem space than single solution techniques. (ii) They offer flexible search techniques to address general optimisation tasks and have been applied successfully to problems in many different disciplines. (iii) Their ability to exploit the information accumulated about an initially unknown search space is their key feature, particularly in large, complex and poorly understood search spaces where classical search tools (enumerative, heuristic) are inappropriate. (iv) Most importantly, their method of exploration of the solution space is inherently parallel, making them an excellent candidate for execution on parallel architectures and undoubtedly speeding up the large data processing time. There has therefore been a substantial effort to enhance the performance of GAs on parallel architectures. Alba and Troya (1999) and Cantú-Paz (1998) have summarised in their surveys the fundamental approaches to GA parallelisation, which started in the 80s and led to many successful implementations capable of reducing the time needed to obtain good results. However, most of these proposals focus on time and quality improvements without considering the case of large data volume processing. The frequent assumption is that the dataset is readily available at each worker node through replication or shared/distributed memory, which is impractical or inefficient for very large datasets.

A way of speeding up the GA is to distribute data access over a cluster of nodes where processing capability and storage capacity are sufficient. Since our work is

focused on using GAs for large-scale data classification problems, we search for solutions to scale GAs for this purpose. We found some proposals that address distribution of the GA process taking into account data distribution. However, they are either based on ensemble approaches, such as DXCS (Dam, Abbass and Lokan 2005), or they assume frequent changes to the set of worker nodes, typical of volunteer computing such as EC-Star (O'Reilly, Wagy and Hodjat 2013).

MapReduce and Spark are programming models designed to support massively parallel computations on distributed data. MapReduce performs the parallel computations on disk-based datasets, which could be efficiently computed with a single pass over the input (e.g. web indexing). On the other hand, iterative jobs can be implemented as a pipeline of MapReduce jobs repeated until convergence, which would require transferring intermediate results from and to an external storage layer (typically a distributed file system such as GFS/HDFS) at every iteration. As GAs typically require many iterations to converge, this would result in substantial overhead due to disk access latency and throughput. Spark is a better fit for in-memory iterative computations such as those required for GA processing. Spark is aimed at compensating for the limitations of MapReduce while retaining its scalability, data locality and fault tolerance.

Even though efforts have been made to parallelise GAs and other evolutionary algorithms using these platforms, our goal is to apply the data partitioning concept. Data partitioning can be defined as data being physically split into distinct parts. All data access for the sake of massive computation, such as fitness evaluation, is restricted inside a partition. At the same time, the system tightly controls data movement. This approach will enable GAs to scale up and will improve processing time. The main objective of this work is to propose a GA model that can be used in large-scale data classification while maintaining efficient output results in reasonable

learning time.

1.2 Contribution

The contribution presented in this thesis is in answer to the question of how to ***enhance the behaviour of rule-based GA classifiers over large-scale data processing using data partitioning***. The goals of our contribution are:

1. Reducing the execution time of the system.
2. Scaling parallel GAs while maintaining a high level of accuracy.
3. Maintain a high GA scalability by automatically controlling its key parameters.

Two models are proposed targeting partitioned data processing:

Partitioned-Data Master-slave (PDMS) This model is a master-slave parallel implementation of GA, which considers the case where data is partitioned over multiple nodes. Master-slave parallel implementation focuses on the most expensive GA processing stage, *fitness computation*. This thesis reports the scalability of the PDMS model scales over different cluster sizes.

Partitioned-Data Multiple-deme (PDMD) This model differs from the original multiple-deme parallel GA as the local GAs would access only a local data partition for all GA processing stages rather than having access to the complete dataset. A potential advantage from this modeling is a reduction in processing time since the local GA accesses only part of the dataset to complete its process. This implementation is compared to the PDMS to report how it performed in terms of

accuracy and processing time. This work has been published in GECCO'19 (Alterkawi and Migliavacca 2019). The contribution included proposing PDMS and PDMD model designs, implementations and performing the experiments.

The proposed models have been tested within the BioHEL machine learning system. We presented the learning accuracy and training time (execution time) when we tested the scalability of the models on the largest classification datasets available on the UCI repository. For context, we compared our proposed models with DEAP-Spark, a GP implementation over Spark (Hmida et al. 2019). We used the learning accuracy and execution time to formulate our conclusion. The experiments performed show that the implementations have considerably reduced the execution time, and it also achieves a high accuracy level for both implementations compared to DEAP-Spark.

AUTO-PDMS & AUTO-PDMD As we have stated, scalability is the core strength of both models, and our experiments show that it could be enhanced by automatically controlling some of the GA fixed parameters. Therefore, we reimplement both models considering an automated strategy for two GA parameters (number of iterations and population size). We compare the accuracy and the execution time of the modified models with the ones proposed initially. These modified models give a good tradeoff between accuracy and time. A key advantage of this automated design is that it can save the user from the trial-and-error experiments to find the ideal number of iterations and population size for the different problems addressed.

1.3 Summary of Chapters

This thesis is divided into a number of chapters:

Chapter 2 This chapter outlines the essential background for the main idea behind this work.

Chapter 3 This chapter discusses the existing work on implementation of distributed parallel genetic algorithms in the literature. The literature is divided into three sections: implementing DPGA using custom distribution topology; MapReduce; and Spark.

Chapter 4 This chapter introduces our first contribution: two GA Parallel Models considering a partitioned data environment. In this chapter, we explain the design of the PDMS and the PDMD models as well as the two models' implementations using the BioHEL classifier over the Spark platform. It also includes the experimental results and a comparison with a Genetic Programming over Spark Classifier (DEAP-Spark).

Chapter 5 This chapter represents our second contribution: enhancing PDMS and PDMD to optimize their performance and enhance their scalability while reducing the number of GA parameters. Experimental results compare the original PDMS and PDMD implementations with the updated versions.

Chapter 6 This chapter presents some conclusions drawn from this research and proposes a number of promising areas for further research.

Chapter 2

Background

This chapter describes the two main aspects of this thesis: genetic-based classifiers and large-scale data processing. It briefly introduces the classification task: a data mining technique used for intelligent decision making. As classification is a big topic, this chapter is intended to provide enough background material to connect the contributions contained in this thesis with the classification task. Therefore, the chapter focuses on explaining the closest topics to our field of application: a rule-based machine learning classifier system, genetic algorithms, and in particular the scaling-up techniques used to handle large-scale data analysis.

The chapter is structured as follows. First, Section 2.1 will provide a brief definition of data mining and its main techiniques and will focus specifically on the classification task. Section 2.2 will describe the main knowledge representations (and the corresponding inference mechanisms) used to solve the classification problem, focusing primarily on prediction rules. Section 2.3 will briefly discuss rules extraction. Section 2.4 explains the basic concept of the genetic algorithms and their general

workflow. Section 2.5 explains BioHEL, a rule-based machine learning classifier, which adopts a genetic algorithm as the rule discovery approach.Section 2.6 and its subsections will present how researchers improved the execution time of GAs using parallel approaches. Section 2.7 introduces the problem of mining large-scale data, also presenting some of the existing approaches. Finally, Section 5.6 will provide a summary of the chapter.

2.1 Data mining

Data mining can be defined as the process of extracting useful patterns from large and raw datasets. These useful patterns are turned into outcomes which help to understand the data and/or make predictions on future data; this is called learning. As defined by Witten, Frank and Hall (2011), learning is the acquisition of knowledge and the ability to use it. The first stage in data mining is preparing the input data *instances*; that is, cleaning data by filtering the noisy instances or instances with missing values. Prior to that step, in some cases data is completely unstructured (such as pure text), and a pre-step is then needed to organize and structure the raw data into a dataset of instances to enable data modeling and pattern extraction. Each data instance that enters into the learning process is characterized by its values on a fixed, predefined set of features or attributes. In this work, we define an instance as follows:

Definition 1. *Given a set of attributes* $A = \{a_1,\ldots,a_m\}$*, and a set of data instances* $I = \{i_1,\ldots,i_n\}$*, where all instances share the same attributes, a structured instance* i *is represented as a discrete sequence of values for those attributes* $i = \{ia_1,\ldots,ia_m\}$*, where* ia_k *represents the value of attribute* a_k *in instance* i*.*

The attribute types can vary between binary, nominal, ordinal and continuous values. Many practical data mining systems accommodate just two of these four: **nominal** (binary and nominal) and **real-values** (ordinal and continuous). **Nominal** attributes (also called cathegorical) represent discrete values which belong to a specific finite set of categories. For example, the traffic light signal could be one of three values: red, green or yellow. The binary value is a special case of nominal with just two values; ordinal is also a special case where there is an order relationship defined between values. On the other hand, **real-valued** attributes are those that take the form of any numerical value.

The second stage in data mining is using machine learning to extract useful patterns and create knowledgeable models that can help to conclude and describe the dataset or even to predict future data instances. Some of the data mining techniques are shown in Figure 1. Descriptive analytics focuses on summarising and converting the data into meaningful information for reporting and monitoring. This category of data mining explains and analyzes the current state of the data. On the other hand, the primary objective of predictive mining techniques is to predict future results from current states. Based on a training set of some input (data instances) and output (target values), a supervised machine learning algorithm is used to build a learning model that is designed to predict the correct output value for future inputs. Regression is a method in statistics which expresses the output as a combination of the attributes, with predetermined weights. It is commonly used to make projections, such as sales revenue for a given business, and it is only applicable if the attributes are numerical. In time series data analysis, each instance represents a different time step and the attributes give values associated with that time. Time series are studied both to interpret a phenomenon (identifying the cyclicity and seasonality components of a trend) and to predict its future values. Classification is about determining a

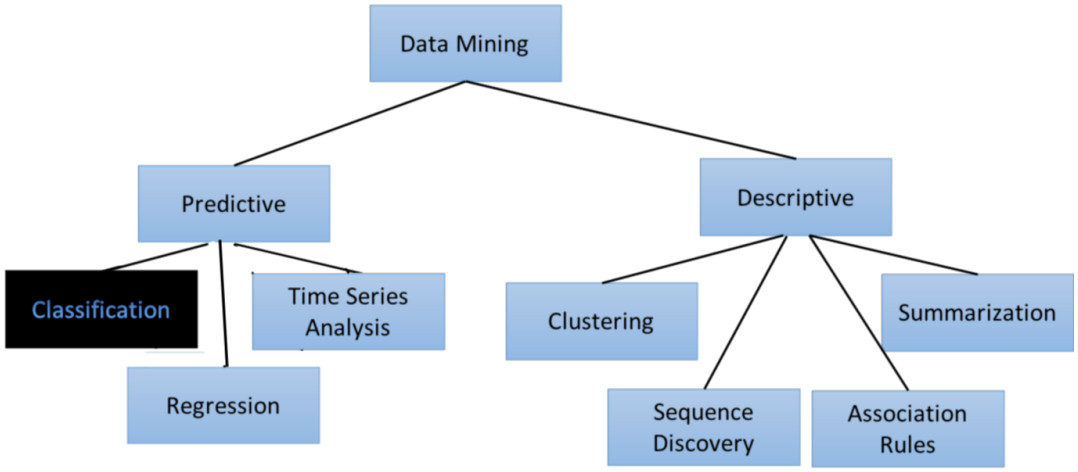


Figure 1: Data Mining Techniques

cathegorical class (or label) for an element in a dataset.

In this thesis, we focus on the classification task. Classification has numerous applications, including fraud detection, target marketing, performance prediction, manufacturing and medical diagnosis. In the following section, I will briefly explain the classification task.

2.1.1 Classification

As shown in Figure 2, the classification learning process involves two phases: the **Training Phase** and **Classification (Prediction) Phase**. A prior step is to divide the dataset into $TrainS$ and $TestS$ sets where each is used in the corresponding phase. (1)**Training Phase** builds a classifier model based on the input training sample, $TrainS$, where the training sample contains labelled data instances; (2)**Classification (Prediction) Phase**, where the classifier model is fed with data instances without their labels, also called testing samples, $TestS$, for predicting their correct classes or labels. The correct classes are then compared with the predicted

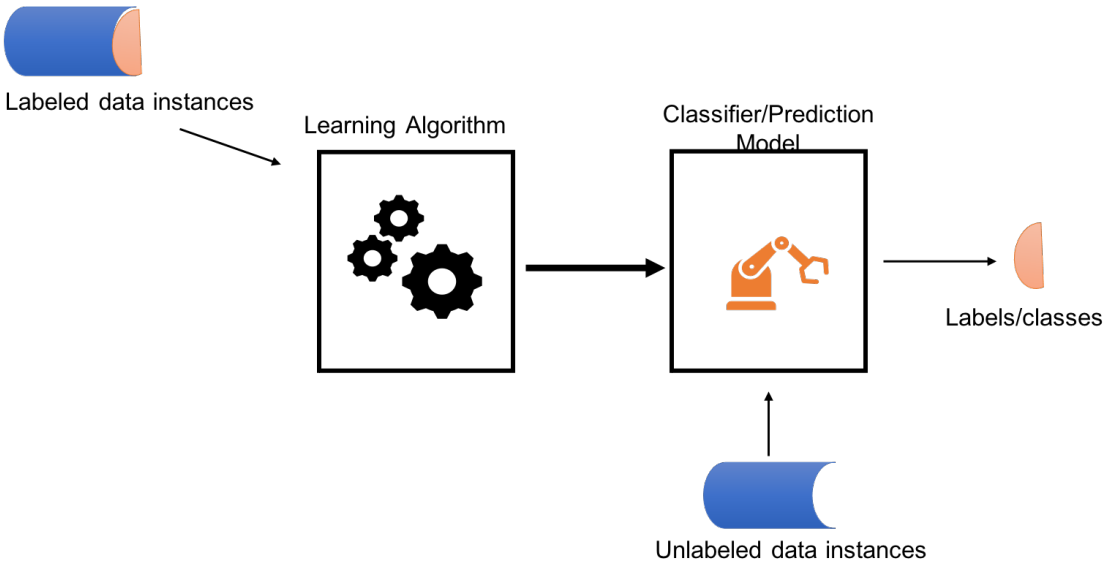


Figure 2: The Learning Process for Classification

ones to estimate the accuracy of the developed classifier model. Usually, a classifier is evaluated with its testing accuracy as well as the training time.

In this work, we follow Bacardit Bacardit (2004) definition of classification:

Definition 2. *Given a set of instances* $I = i_1, \ldots, i_n$*, each of them labeled with a finite set of classes* $C = c_1, \ldots, c_m$*, the task of classification is to create a certain theory* T *based on* I *and* C *that, given an unlabeled new instance, can give a prediction of the class of this instance.*

Researchers use different algorithms to create the classifier model in the training phase, such as Decision Trees, Logistic Regression, K-Nearest Neighbor, and Neural Networks. It is difficult to make precise statements about the effectiveness of the classifier modeling algorithms, as each of these learning algorithms has its advantages and limitations in data mining applications. As mentioned in (Freitas (2002)), it is enough to say that each classification learning algorithm performs better on some

datasets than others, and hence none is universally superior across all datasets. Another technique that is used to learn a classifier model is rule-based machine learning. The defining characteristic of a rule-based machine learner is the identification and utilization of a set of relational rules (prediction rules) that collectively represent the knowledge captured by the system (classifier model), in contrast to the other machine learners that commonly identify a single model that is used to make the prediction.

This thesis focuses on rule-based classifier modeling which concludes rules using a GA. The following section briefly defines the different ways to represent knowledge. It will mainly focus on knowledge representation of prediction rules.

2.2 Knowledge Representation

There are different ways to represent a learned model, or knowledge, in the literature, and they are strongly mapped to the classification task Bacardit (2004). We represent below six of the main representations:

Decision trees Predict an instance class, starting from the root of the tree. Each internal node represents a "test" on an attribute and each branch represents the outcome of the test. An instance attribute value is compared against the node's specified attribute and based on this comparison one descending branch, which evaluates to true, is followed to the next node until a leaf is reached. The leaf node provides the classification of the instance (Mitchell 1997). A decision tree representation is shown in Figure 3.

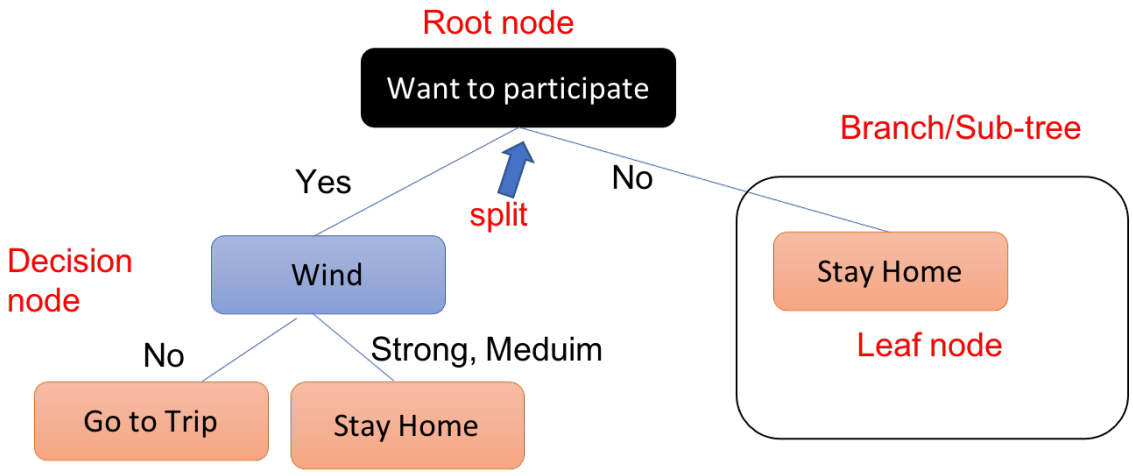


Figure 3: Example of a Simple Decision Tree

Canonical tree-structure In this model knowledge is represented as a tree in which nodes are function nodes, representing a subroutine in the algorithm, or terminal nodes, representing constants or variables defined by the algorithm. A simple example is shown in Figure 4. The algorithm is executed by a depth-first traverse of the tree, starting from the root node, searching for function nodes and their operands. An examination of nodes 2, 3, and 4 yields the logical value true or false. A true result causes the traverse to continue by examining node 5 and its operands. A true value for Z calls the subroutine XYZ, while a false value calls the sub-routine XY. If the expression X AND Y is false, the algorithm relinquishes control. Lemczyk and Heywood (2007) uses this representation for GP-based classifier model.

Sets of instances This knowledge representation consists of storing a set of instances, either taken from previous experience or modifying the instances using a new representation, and using them to classify input examples. Instance-based learning

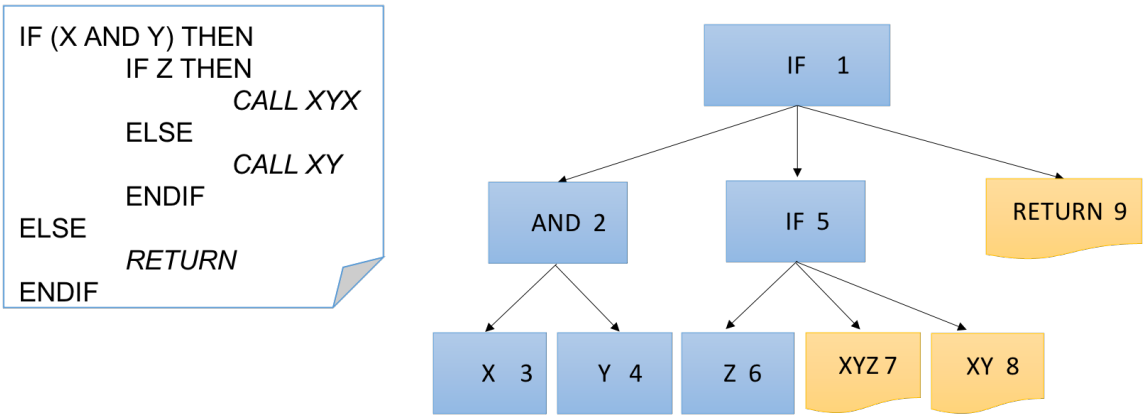


Figure 4: Canonical tree-structure (Keane 2008)

algorithms look for k instances from the selected stored set nearest to the input example based on distance metrics. When the k closest instances to this example are selected, the predicted class can be based on a simple voting mechanism or more complex techniques. Different instances-set algorithms and a comparison between them can be found in (Yu et al. 2014). The instances-set representation is shown in Figure 5. These classifiers do not construct any classifier model explicitly; instead they keep all training data in memory (or a data subset) for classification.

Bayesian networks Also known as belief networks, or decision networks, they are graphically represented as an acyclic graph of nodes and directed edges connecting them. Each node represents a random variable and the directed edge connecting nodes defines a dependency relation: the node origin of the arrow influences the pointed node. Nodes that are not connected represent variables that are conditionally independent of each other. Each node is associated with a probability function that takes as input a particular set of values for the node's parent variables and gives as output the variable's probability represented by the node. For example, if a node

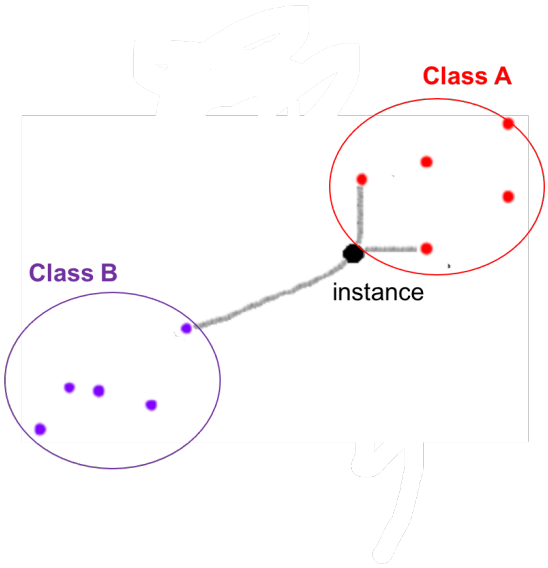


Figure 5: Example of Sets of Instances

has two parent nodes and each node is represented with a Boolean variable, then the probability function is represented by a table of 2^2 entries, representing all the possible combinations of the Boolean values of the two parents. Figure 6 shows a simple example of three nodes, where each is assigned a table representing the probability function. More details about Bayesian networks and their application can be found in (Niedermayer 2008).

Artificial Neural Networks are biologically inspired from the functions of neurons in the brain. As shown in Figure 7, each neuron acts as a computational unit, accepting a number of inputs (possibly outputs from other neurons) and outputting a single output. Neural networks learn (or are trained) by processing examples, where each example consists of an "input" and a known "result", forming probability-weighted associations between the two which are stored within the data structure of

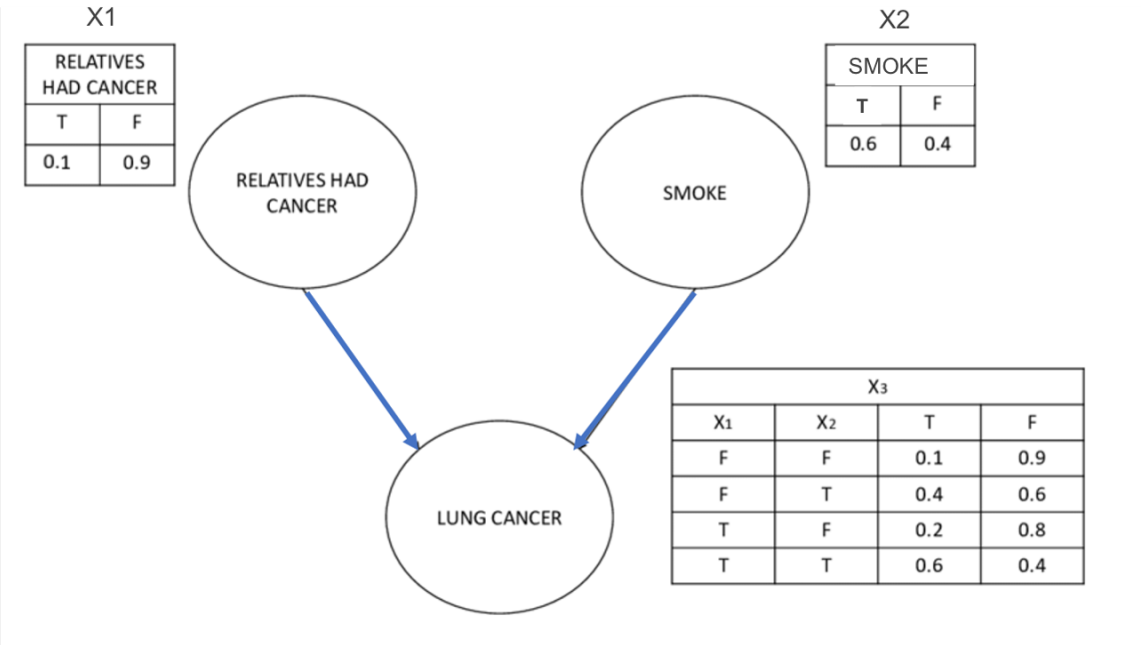


Figure 6: A Simple Example of Bayesian networks

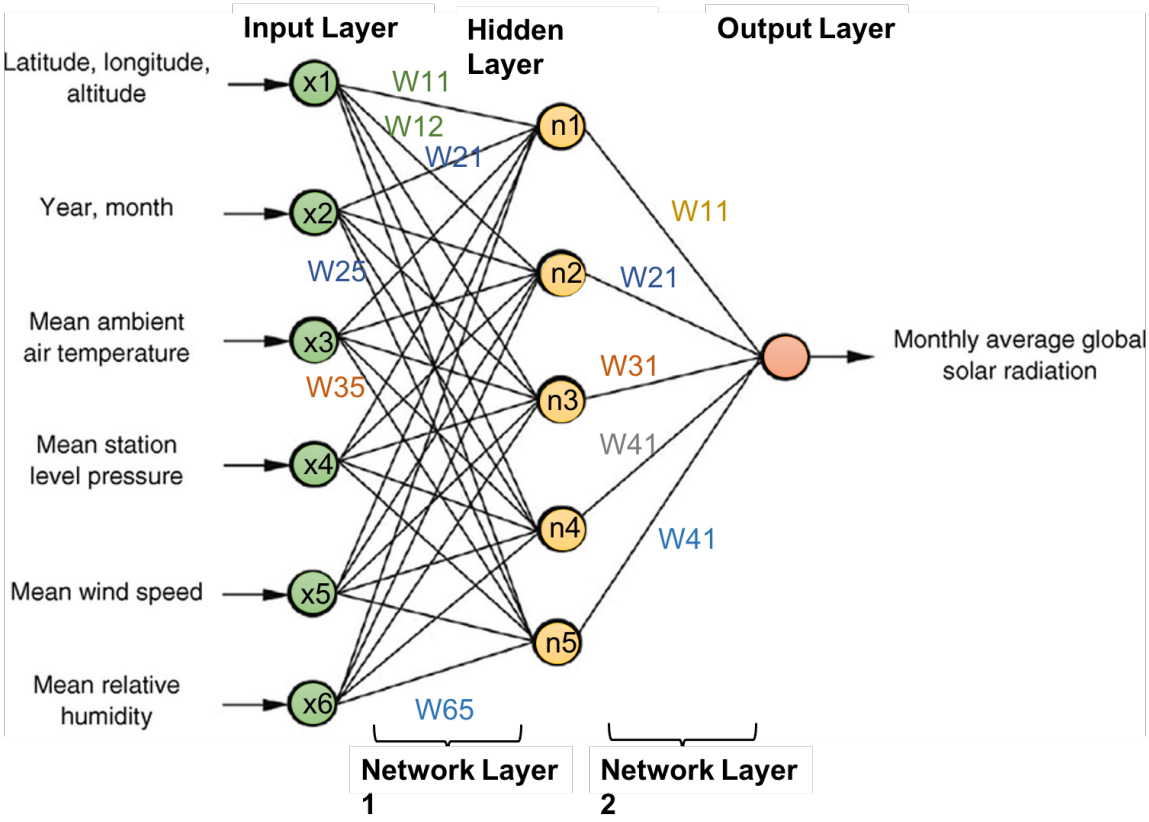


Figure 7: A Simple Example of an Artificial Neural network

the net itself. The training of a neural network from a given example is usually conducted by determining the difference between the processed output of the network (often a prediction) and a target output. This difference is the error. The network then adjusts its weighted associations according to a learning rule and using this error value. Successive adjustments will cause the neural network to produce output that is increasingly similar to the target output. After a sufficient number of these adjustments, the training can be terminated based upon specific criteria. The origin of NN can be found in (Clark 1950).

Prediction Rules Rule sets are the most ancient knowledge representations and probably the easiest to understand. Their origins can be traced back to the ancient Greek philosophers and their propositional logic. A classification model is represented by a rule set composed by individual rules. Rules can take very different forms and also there are many different ways to interpret the rule set in order to classify an input instance. Below is a brief description of three aspects: the syntax of rules; the classification process; and the rules discovery approach.

1. Rule Syntax Logical conjunctions are a kind of knowledge representation used to express **IF-THEN** prediction rules. The rule antecedent **(IF part)** consists of a conjunction of conditions (e.g. {IF 'temparature >37.8' AND 'cough' }). The rule consequent **(THEN part)** predicts the class or label value for a data instance satisfying the rule antecedent (e.g. THEN {'Covid-19'}). Rules can be represented in many different ways to handle different data types (e.g. binary, discrete-valued, ordinal, continuous-valued).

2. Classification There are two main strategies for using a combination of multiple rules for classification: Unordered rule set, and ordered rule set (rule list). In the former a voting scheme is used where all rules either have the same importance or votes or some rules might have higher voting power. In the set, the rules are either mutually exclusive or a strategy for resolving conflicts, such as majority voting, which may be weighted by each rule's accuracy or some other quality measurement. Interpretability potentially suffers when several rules apply.

In the Rule list, an ordered rules-set, is created, and the first rule that matches the instance is used to predict its class. Rules Set solves the problem of overlapping rules by only returning the prediction of the first rule in the list that matches the

instance.

Both strategies have to deal with the situation in which no rule applies to an instance. Usually this is resolved by introducing a default rule. The prediction of the default rule could simply be the majority class (the class with the largest amount of instances in TrainS) or minority class (the class with the minimum amount of instances in TrainS).

These simple policies perform pretty well compared to the original system. However, they perform poorly on specific datasets, showing a lack of robustness. Bacardit (Bacardit, Goldberg and Butz 2007) represented a better automatic selection for the default rule-class, which increases the accuracy of the classifier. It is noteworthy to highlight that even though using a default rule may conceal the actual performance of the classifier, especially when some classes suffer from normalisation where they represent a small number of dataset instances and end up misclassified entirely.

3. Rules Discovery Decision rules are discovered using rule induction (Michalski 1983; Holsheimer and Siebes 1994) or evolutionary algorithms, more specifically GAs (Alves de Araujo, Lopes and Freitas 1999; Giordana et al. 1997; Augier, Venturing and Kodratoff 1995; Janikow 1993; Greene and Smith 1993) search strategies, where the former uses a local, greedy search strategy, whereas the latter uses a kind of global, population-based search strategy inspired by natural selection. We shall focus our discussion on the GA approaches, giving a brief explanation of rule induction in the next section.

It is worth mentioning that some researchers conclude rules from decision trees using the rule-pruning method rather than learning the decision rules from the

dataset (QUINLAN 1993; BRESLOW and AHA 1997; Mingers 1989). The rule-pruning method first converts the tree into a set of rules, and then for each rule considers removal of conditions regardless of the order in which conditions were generated (Freitas 2002). Rule-pruning has some disadvantages: it is computationally expensive, because the computation of statistics required to decide which condition should be removed from a rule often requires new scans of the training set. In addition, the entire approach can be considered somewhat cumbersome: it consists of first building a decision tree, then converting the tree into a rule set and finally pruning those rules to achieve a more "flexible" rule set that does not follow a decision-tree structure.

2.3 Rule-based Classifier using GAs

There are three main designs for rule-based classifiers that use GAs for rule discovery:

Michigan Approach Each individual in the GA represents a single rule, i.e. a part of a classifier. The GA evolves new individual at each generation, then a subpopulation—a set of most efficient rules—is derived to represent the prediction model (WILSON 1987). The main problem with the Michigan approach is that in general GAs measure the performance of a rule (individual) out of the context of the other rules, ignoring the critical issue of rule interaction. Hence, in the Michigan approach, we often need some additional methods that foster the discovery of a good set of rules, a subpopulation rather than a single good individual. In particular, the method used to discover a group of rules should avoid converging all individuals to the same point in the search space.

Pittsburgh Approach Each individual encodes a complete set of rules. Greffenstette (1989) initially proposed this model, and a sophisticated implementation of this approach can be found in (Bacardit 2004). In the Pittsburgh approach, since an individual represents a set of rules, the entire solution for the classification problem, it is easy to evaluate the performance of the complete classifier as the quality of the individual (fitness) On the other hand, an individual tends to be significantly more complex in this approach —at least, longer—than in the Michigan approach. This often leads to more complex genetic operators. In addition, the fact that at any given generation there are many more rules to be matched against the data being mined, in comparison with the Michigan approach, usually leads to a longer GA execution time.

Iterative Rule Learning (IRL) As with the Michigan approach, each individual in the population represents a single rule, but rather than evolving all rules together, a sequential covering algorithm approach is adopted, where one rule is learned at a time. A GA is used for rule learning, and after each rule is obtained the training examples covered by this rule are removed from the training set. Thus, the GA is forced to explore other areas of the search space to learn other rules. The approach continues searching for rules until a terminating condition is met. The terminating condition can vary across different implementations, but is typically linked to the fact that no more interesting rules can be induced when the training set is empty. All rules are inserted into a rule set with an explicit default rule, covering the majority class of the domain (Tung 2009).

Algorithm 1: Iterative Rule Learning

```
1 Rule-Set=∅ ;
2 do
3 |   Rule ← findBestRule(TrainS);
4 |   Covered ← Cover(Rule,TrainS);
5 |   TrainS ← TrainS-Covered;
6 |   Rule-Set ← Rule-Set ∪ Rule;
7 while TrainS ≠ ∅ ∧ ¬ StopCriteria(Rule,Rule-Set,TrainS);
8 return Rule-Set
```

Compared to the Michigan and Pittsburgh approaches, the advantage of the IRL approach is that it drastically reduces the genetic search space, even if several searches must be performed if several rules are to be learned (De Jong and Spears 1991).

Rule Induction Similar to GA IRL, a rule induction algorithm induces a rule at a given time and adds it to the rule set. To create a rule, a search takes place to find the best condition to be added to the current induced rule, based on a given evaluation function and on the current contents of the training set, and then it adds the selected condition to the current induced rule. This process is repeated, while the induced rule can be improved by adding a new condition to it. The rule is then added to the rule set, and a match process matches the current training set with the recently induced rule in order to remove the data instances that are covered and have the same class of the induced rule. The algorithm keeps adding rules to the rule set and stops when all data instances are covered by at least one induced rule, or until the number of uncovered instances is very small and could be neglected. Algorithms of the rule induction paradigm usually perform a greedy local search for rules, selecting

one condition at a time to be inserted into or removed from a candidate rule. As a result, this may cause the algorithm to miss the best rule(s). Some conditions may seem irrelevant separately, but in conjunction with other conditions they may have a significant influence on the rules-set quality.

Work by Dhar, Chou and Provost (2004) compared a GA with two greedy rule induction algorithms and presented some evidence that the GA is more effective at finding hidden attribute interactions. Moreover, Greene has reported that genetic algorithms (GA) have shown great promise on complex search domains compared to rule induction algorithms (Greene and Smith 2004). The following section explains the basic elements of GAs and how they operate.

2.4 Genetic Algorithms

GAs are search algorithms which use principles inspired by Charles Darwin's theory of natural selection, whereby the fittest individuals are chosen to produce offspring, which are the individuals of the next generation. Holland's book *"Adaptation in Natural and Artificial Systems"* (Holland 1992a) presented the genetic algorithm as an abstraction of biological evolution and gave a theoretical framework for adaptation under the GA. In contrast with evolution strategies and evolutionary programming, Holland's original goal was not to design algorithms to solve specific problems but rather to formally study the phenomenon of adaptation as it occurs in nature and to develop ways in which the mechanisms of natural adaptation might be introduced into computer systems. Because of their adaptation, genetic algorithms (GAs) proved to be successful in search and optimization problems. GAs can exploit information

about an initially unknown search space and bias subsequent searches into useful subspaces. This is their key advantage in complex and poorly understood search spaces, where classical search tools (enumerative, heuristic, ...) are inappropriate, offering a valid approach to problems requiring efficient and effective search techniques. In GAs, each candidate solution is represented using the following:

Individual A candidate solution to the problem to be solved,

Chromosome The codification of an individual. Usually an individual is codified using a single chromosome.

Gene Each of the atomic values of a chromosome, either a single bit or a short block of adjacent bits, in case of binary or nominal attributes. For real-valued attributes, the attribute values could be codified as integers, real numbers or intervals, depending on the problem.

Allele Refers to a single variant of the gene. For example, considering a gene represented with a single bit, an allele is either 0 or 1.

Fitness value A value that indicates the degree of adaptation of an individual (solution) to the environment or, in other words, how "fit" or how "good" the solution is with respect to the problem under consideration.

One crucial task that affects the efficiency of GAs is choosing a suitable scheme for encoding the genes. Encoding mainly depends on the type of problem. Most GAs use fixed-length, fixed-order bit strings to encode candidate solutions. However, in recent years there have been many experiments with other kinds of encodings, several of which are described in (Mitchell 1998).

Algorithm 2 shows the basic steps of GAs that were first laid down by Holland (1992b). The general workflow for the simple GA can be explained as follows:

Algorithm 2: The basic steps of a simple GA

```
1 Generate initial population;
2 Evaluation();
3 while Not termination-condition do
4 |   Selection();
5 |   Crossover();
6 |   Mutation();
7 |   Evaluation();
8 |   Replacement()
9 end
```

Initial population. The GA starts with its first step (line 1) of creating an initial population which is the first generation of feasible solutions or individuals that are randomly or heuristically created (Kazimipour, Li and Qin 2014). The initial population contributes genetic materials. Heuristic-created individuals can lead to a quick solution, while random ones result in a higher diversity, leading to a better solution.

Fitness function. The created solutions are then evaluated using a fitness function (line 2). A fitness value is assigned to each newly created individual, representing how good the solution is. The fitness function may become the bottleneck of the algorithm, especially in big data problems where an individual is evaluated across the entire dataset at every generation, resulting in a reduction in the efficiency of the algorithm. Considering classification, the fitness function should evaluate the performance of a classification rule with respect to the training dataset. To understand

how a rule is evaluated, let us assume that a rule with antecendent A predicts the consequent (class) c. A and c are matched with the attributes and the actual class of each instance in the training datasets and, a result, four different values are gathered for the rule: TP (True Positives) = Number of instances satisfying A and having class c; FP (False Positives) = Number of instances satisfying A but not having class c; FN (False Negatives) = Number of instances not satisfying A but having class c; TN (True Negatives) = Number of instances not satisfying A nor having class c.

Given the values of TP, FP, FN and TN, as discussed above, several measures can be defined to represent the quality of the rule, for example:

$$Precision = TP/(TP + FP) \tag{1}$$

$$TruePositiveRate = TP/(TP + FN) \tag{2}$$

$$TrueNegativeRate = TN/(TN + FP) \tag{3}$$

$$AccuracyRate = (TP + TN)/(TP + FP + FN + TN) \tag{4}$$

$$CoverageRate = TP + FP/(TP + FP + FN + TN) \tag{5}$$

In principle, a fitness function could use any combination of the above measures to evaluate rule quality concerning predictive accuracy. Accuracy is one of the most popular metrics in both binary and multi-class classification problems. Grandini, Bagli and Visani (2020) present an overview of different fitness evaluation metrics such as "Balanced Accuracy" and "Confusion Matrix", where most of these metrics evaluate an entire classifier fitness and are not intended to evaluate individual rules.

Selection. The algorithm selects parents, *individuals*, who will crossbreed and create offspring (line 4). There are different methods used for the parent selection, including *Roulette-Wheel Selection*, *Tournament Selection*, and *Boltzmann Selection*. Roulette-Wheel is the most common selection method, where each individual is assigned a slice of a circular "roulette wheel", the size of the slice being proportional to the individual's fitness. The wheel is spun N times, where N is the number of individuals in the population. On each spin, the individual under the wheel's marker is selected to be in the pool of parents for the next generation. In tournament selection, a random number s of individuals is selected (with or without replacement) from the population, and then the best of the s individuals is chosen to be added to a mating pool. This process is repeated until the mating pool is filled (selection of enough parents to generate offspring). Boltzmann selection works similarly to simulated annealing, in which a varying "temperature" controls the rate of selection according to a preset schedule. The temperature starts high, which means that selection pressure is low and every individual has a reasonable probability of being a parent. The temperature is gradually lowered and the selection pressure increases, thereby allowing the GA to narrow in ever more closely to the best part of the search space while maintaining a reasonable degree of diversity. More selection methods can be found in (Saini 2017; Mitchell 1998).

Crossover. The first genetic operator which is applied to the parent to reproduce is crossover (line 4). Crossover consists of exchanging genetic material between two parents' chromosomes. The most crucial criterion for crossover operator design is that children inherit the superior characteristics of their parents. The crossover and selection operators create a link between the old and new generations as they are responsible for passing the good properties, *knowledge*, from the old to the new

populations' chromosomes. Crossover can be considered the most important operator in the genetic algorithm, determining the global convergence of the genetic algorithm. There are different ways that the crossover operator is applied to the two parents. The three main types can be defined as follows:

- **One point**: A point on both parents' chromosomes is picked randomly, called a 'crossover point': the genes to the right of that point are swapped between the two parent chromosomes. This results in two offspring, each carrying some genetic information from both parents.
- **Two-point:** Two crossover points are picked randomly from the parent chromosomes. The genes in-between the two points are swapped between the parent chromosomes.
- **Uniform**: Each gene is chosen from either parent with equal probability. Sometimes a different mixing ratio is set and results in offspring inheriting more genetic information from one parent than the other.

Mutation. The second genetic operator is mutation (line 6), which acts on a single individual at a time. It replaces the value of a gene with a randomly generated value. Mutation is a blind operator, whereby the selected gene and its new value are randomly chosen without any attempt to maximize the fitness of the new individual. Mutation is used to maintain genetic diversity from one generation of a population of genetic algorithm chromosomes to the next.

After reproduction, the offspring will be evaluated using the fitness function and are usually sorted according to their fitness values (line 7).

Replacement. Is the process of creating new generations (line 8). Following the

genetic operator activity, parents are replaced by offspring. Some GAs adopt a process that allows the best individuals from the parents' generation to carry over to the next generation, which is called elitism. If elitism is used, the individuals with the best fitness replace the offspring with the worst fitness. Elitism is used to guarantee that the solution quality obtained by the GA will not decrease from one generation to the next. Elites should only represent a small portion of the population in order to maintain diversity and avoid premature convergence.

The steps in lines (line 4 – line 8) are repeated until a termination condition is met. A maximum number of generations (iterations) is the most commonly used condition to end the GA process. GA designers use other various conditions in the implementation of their algorithms to finish the processing of their algorithms. One possible condition is when the specified time to run the algorithm has been met. Another condition is when no significant improvement in terms of solution quality occurs. In some problems (classification might be one of these), the user can specify a fitness level that corresponds to a good solution. Termination could be reached when the best-found individual reaches or exceeds that accepted fitness value. However, in general, the best achievable fitness is not predictable. A smarter termination for GAs would satisfy a convergence condition (Schmitt 2001). Genotypic convergence is the point at which evolution stops because every individual in the population is identical. This type of convergence can only take place in genetic algorithms when no mutation is used.

Next we present BioHEL (Bioinformatics-oriented Hierarchical Evolutionary Learning) Bacardit, Burke and Krasnogor (2009). BioHEL is an IRL-based classifier designed for single node large dataset, which we later extend to multi-node operation on distributed datasets.

2.5 BioHEL: GA IRL Classifier

BioHEL is a classifier designed to handle large and complex data classification tasks, such as protein structure prediction. It has been successfully applied to large-scale bioinformatics datasets (Bacardit et al. 2007). Algorithm 3 shows the pseudo-code of the general workflow of BioHEL.

Algorithm 3: BioHEL general workflow

```
   Input: TrainSet
1  RuleSet =∅ ;
2  stop=false;
3  while stop = false do
4      BestRule = null;
5      for repetition ← 1 to NumRepetitionsRuleLearning do
6          CandidateRule = RunGA(TrainS);
7          if CandidateRule is better than BestRule then
8              BestRule = CandidateRule ;
9      end
10     Matched = Examples from TrainS matched by BestRule;
11     if class of BestRule is the majority class in Matched then
12         Remove Matched from TrainS;
13         Add BestRule to RuleSet;
14     else
15         stop = true
16 end
   Output: RuleSet
```

BioHEL follows a typical IRL by learning a rule at a time (lines 4–9) and adds

it to a rule list (line 13). BioHEL repeats several learning attempts with different seeds (lines 5–9), and only the best candidate rule is selected (`bestRule`), (lines 7–8). The selected rule is then penalised to induce niche formation in the search space. To achieve this as, per the typical IRL strategy, BioHEL deletes the training examples that have been covered by the selected rules (matching the selected rule) (line 12). The iterative search for new rules stops when it is no longer possible to find any rule where the associated class is the majority class of the matched examples (lines 11–15). When this happens, all remaining examples are assigned to the default rule. BioHEL is strongly influenced by the GAssist Pittsburgh LCS (Bacardit 2004), inheriting from it some main mechanisms. For a full description of BioHEL, as well as the justification for its design issues, please see (Bacardit and Krasnogor 2006). As described above, BioHEL evolves and learns each rule using a GA.

In order to understand the different terminologies represented in section 2.4 under the heading of rules knowledge representation, we will use BioHEL representations to give an example of each. A graphical representation will be used to explain some of these terminologies also.

- **Individuals representation** since BioHEL is an IRL-based GA classifiers, which discover one rule at a time, the solution of the problem is a rule and the individual is the representation of a rule. BioHEL uses the attribute list knowledge representation (ALKR) (Bacardit and Krasnogor 2009) to encode the individuals. This meta-representation is able to handle real-valued and nominal attributes at the same time.

- **gene**: A gene is a condition that could include either a real-valued attribute or a nominal one. In the case of a real-valued attribute, the condition is represented with a lower and an upper bound of its associated interval. In a nominal

attribute, the condition is represented with a string of bits, where each bit represents a possible value for the nominal attribute. If the condition covers the value, then a '1' in that bit represents that coverage. Otherwise, the bit is set to zero. Figure 8 shows an example for both gene representations.

- **An individual chromosome** A chromosome is a rule (with multiple genes) represented by four elements, as shown in the example in Figure 9: (1) an integer containing the number of expressed attributes; (2) a vector specifying which attributes are expressed; (3) a vector of the lower and upper bound for each real-valued expressed attribute; (4) a vector of the covered values for nominal attributes; and (5) the class associated with the rule.

- **initialization** Smart initialization operators create rules that are generalised versions of randomly sampled training instances. In the case of a real attribute, for example, the size of the interval is randomly initialised, with a uniform distribution of between 25% and 75% of the domain size. This interval is centered with the attribute value of a randomly selected training instance. In case the interval overlaps with the lower or upper bound of the attribute domain, it is shifted so that it lies fully within the domain. It also considers the use of the explicit default rule, sampling only instances not belonging to the class of the default rule. The detailed initialisation process can be found in (Bacardit 2004).

- **one-point crossover** A crossover point can have one of three positions in the selected parent chromosome: (1) between attributes, and in this case, the list of attributes before the point are copied from the other parent; (2) inside a nominal attribute; or (3) inside a real-valued attribute. The crossover point could

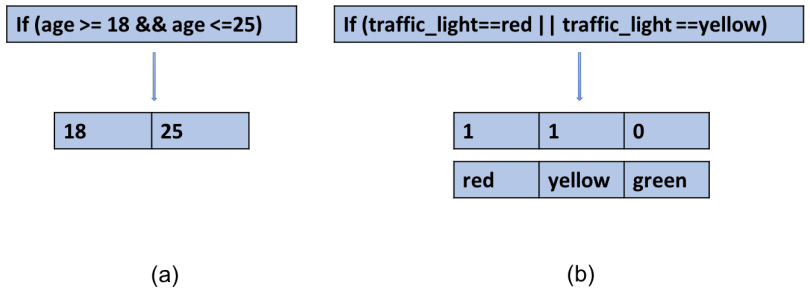


Figure 8: BioHEL: Gene representation. (a) shows how a real-valued attribute is represented with the lower bound and the upper bound of attribute in the condition. (b) shows an example of the traffic light colors and how they are represented for this attribute in the condition

be inside an attribute which does not exist in the other parent's chromosome. Figure 10 shows how the swap would take place in all situations.

- **mutation** Mutation may affect three different parts of the chromosome: a nominal attribute gene's bit; an upper or lower bound of a real-valued attribute; or the class. A bit-flip mutation is applied to the selected bit in the nominal attribute. In the case of the real-valued attribute, the mutation operator selects one bound with uniform probability and adds or subtracts a randomly generated offset to the bound, the size (picked with uniform distribution) being between 0 and 50% of the attribute domain. If the mutation affects the class value, a different class value is assigned to the rule, picked at random.

  If the crossover or the mutation operators create invalid intervals where the lower bound is higher than the upper bound, a repair would swap the bounds.

- **Fitness Function** In BioHEL, the fitness function is defined as follows:

$$Fitness = TL \cdot W + EL \tag{6}$$

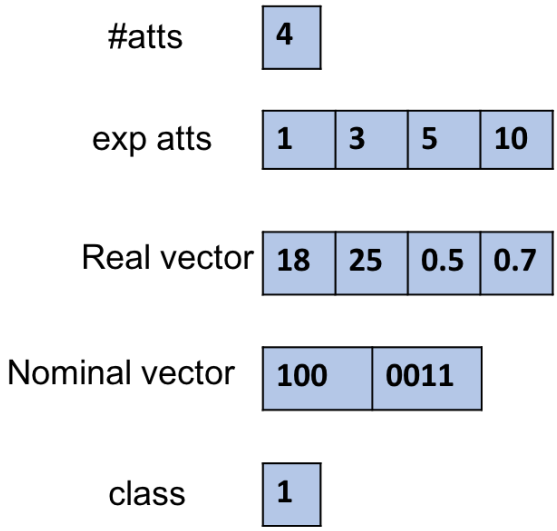


Figure 9: BioHEL: Chromosome representation

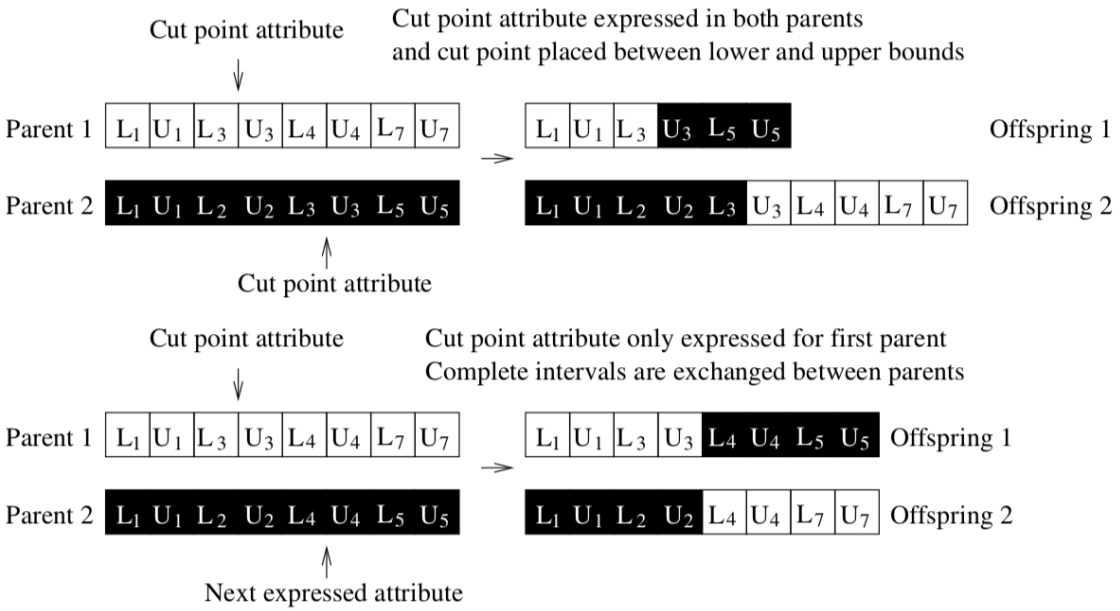


Figure 10: BioHEL crossover: Example of the crossover operator for two cases: when the cut point attribute is expressed in both parents or only in the first parent (Bacardit 2004)

where TL stands for theory length (the complexity of the solution) and EL stands for exceptions length (the accuracy of the solution). EL is designed taking into account a suitable tradeoff between accuracy rate and coverage, thus covering as many examples as possible without sacrificing accuracy. W is a weight that adjusts the relationship between TL and EL. BioHEL seeks to minimize its fitness. For more details about the EL, W and TL values, the reader may refer to (Bacardit 2004)).

As we are addressing large-scale data, the BioHEL, specifically the GA in BioHEL, may require substantial training time to conclude the classifier. One of the most promising solutions for enhancing the performance of GAs is the parallelisation approach. Although using a finite number of computing resources in parallel cannot lower the intrinsic time complexity, parallelism can reduce the time to reasonable levels. In the following section, I shall explain the most well-known GA parallelisation methods.

2.6 Parallel GAs

Parallelisation is key to the success of a genetic algorithm: as GA is applied to complex and large problems in the domains of business, science and engineering, there is a solid drive to reduce the execution time required to obtain reasonable quality solutions. Consequently, there are several well-established strategies, together with some more recent techniques, to implement parallel GA (PGA).

There are three main traditional approaches to PGA Alba and Troya (1999): master-slave, coarse-grained (*multiple demes*) and fine-grained parallelisation. Each model distributes data and computation tasks differently across different processors.

In the following two sections, we explain the first two approaches. In the fine-grained model, the population is distributed over a sizeable topological mesh where each processor hosts mostly one individual. While fine-grained approaches have been successfully implemented on massively parallel hardware using e.g. MPI or GPGPU techniques, it is challenging to adapt these techniques to a large data scenario, given the high degree of communication required between the different nodes/machines. Consequently, I do not consider such models in my implementation and leave the exploration of such techniques as a possible area for future research.

2.6.1 Master Slave Model

In the master-slave PGA model, all individuals belong to a single population, which is managed by a master processor. The master node is responsible for handling the selection and genetic operators, and there is therefore no need to communicate with the other processors during this evolution process. However, it delegates to the processors, or *slaves*, the fitness computation task, i.e. the most expensive operation of the GA. In particular, the master distributes each individual to a processor to evaluate its fitness, thus parallelising the fitness function computation. The slaves determine the fitness for the assigned individuals on the training instances, then the results are sent back to the master processor, which selects individuals to pass to the next generation. In this model, the data is replicated over the different processors, or all processors access the data in shared memory for fitness computation. This approach leads to solutions of the same quality as the sequential approach (as the fitness evaluation is equivalent to the sequential case), and the selection and crossover operate across all individuals in the *panmictic* population.

Master-slave PGA is easy to implement, and it can be a very efficient method

of parallelisation when the fitness evaluation needs considerable computation. One drawback of this model is the overhead resulting from the shared memory being accessed by all processors. Also, the master needs to wait for all the slaves to return the fitness results to start the following process, which impedes speed.

2.6.2 Multiple-deme or Island Model

Multiple-deme models are a popular and efficient way to implement GA on both serial and parallel processors. Multiple-deme GAs are known by different names, such as coarse-grained GAs, and the island model. Probably the first systematic study of PGAs with multiple populations was Grosso's dissertation (Grosso 1985). In some cases it has proven to be more efficient than panmictic populations, even in the case of sequential implementations. Whitley's island implementation managed to optimise a broad range of sample problems more accurately and more consistently than a GA with a single large population (WHITLEY and STARKWEATHER 1990).

In a parallel implementation of a multiple-deme model, each processor executes a GA independently and maintains its subpopulation for search. For example, a total population N_{total} for a serial algorithm could be spread across M machines by giving each processor a population size of $N_{\text{island}} = N_{\text{total/M}}$. The processors work together by periodically exchanging a portion of their population in a process called migration. Migration is affected by a number of different parameters: (1) migration frequency, the number of generations (or evaluations) between two migrations; and (2) migration size (rate), the number of individuals in the population to be replaced between the parts of the population (subpopulation). Grosso observed that when the demes were running without migration, the fitness rose rapidly, but stopped at a lower value than with a single large population, and therefore the quality of the solution

found after convergence was worse in that scenario. At intermediate migration rates however, the multiple-deme found solutions close to those found in the panmictic population, as this process allows exploration of areas not yet discovered. These observations indicate a critical migration rate below which the isolation of the demes obstructs the algorithm's performance, and above which the partitioned population may be able to find solutions of the same quality as the panmictic population.

The impact of migration was also studied by Luque and Alba (2010), and the impact of the frequency of migration has been studied by Mambrini and Sudholt (2014). Broadly speaking, dense migration topologies, such as a complete graph, lead to a fast spread of good solutions at the price of high communication overhead. Less frequent communication between islands may save the communication overhead, at the price of delays in spreading new good individuals. Setting the migration interval correctly is a challenge for designing efficient island algorithms (Mambrini and Sudholt 2015). In most multi-deme PGAs, migration is synchronous, which means that it occurs at predetermined constant intervals. Migration may also be asynchronous, that is, the demes communicate only after some events occur.

As data size increases, current trends support the implementation of GAs over multiple networked computing nodes (horizontal scalability) rather than using multi-cores with a shared memory (vertical scalability). Distributed PGA (DPGA) considers using multiple distributed nodes to scale up to solve large-scale data mining problems. Even though both the master-slave and the multiple-deme have been implemented in a distributed environment, multiple-demes offer an excellent opportunity for distribution, as different nodes run independently and occasionally communicate to perform the migration. Thus, the resulting communication overhead is minor in the context of the total execution time.

We consider the following assumption in our DPGA: a coordinator (master)

processor controls the overall progress in the distributed genetic algorithm and coordinates the distributed process between the other processors. Below, we motivate Distributed PGA (DPGA). Then, in the following section we explain the data partitioning concept. Two distributed parallel programming platforms, MapReduce and Spark, are described briefly at the end of the chapter.

2.7 Large-scale Data Processing

Processing large datasets brings challenges for both storage and processing capabilities. Usually, large data is stored in a data warehouse (local storage) or on cloud platform. A data warehouse is a centralised storage scheme for organising data from different sources. Data warehouse tools make it possible to manage data efficiently by finding, accessing, visualising and analysing data. With cloud storage, data and information are stored remotely and can be accessed from anywhere. It can rapidly scales up the storage capacity according to application demands, making it an attractive option for meeting increases in data volume. Of course, the importance of storing such a massive amount of data lies in the benefit of extracting the knowledge hidden inside. Analysing this massive data is not easy with traditional data mining and machine learning algorithms. Large-scale data mining is highly demanding for hardware resources (processors and memory) and time-consuming. Thus, we need to provide an efficient processing mechanism as well as reducing the time required to analyse data.

A possible approach to sidestepping the challenges of analysing large datasets is to reduce their size using sampling methods. The main idea is to choose a representative sample of the data for learning instead of using the entire dataset, allowing the user to trade result accuracy for response time. Sampling methods can effectively

reduce the amount of data and help speed up data processing. Hence, sampling techniques have been widely studied and used in big data contexts, e.g. methods for determining the sample size, combining sampling with big data processing frameworks (Mahmud et al. 2020). Albattah (2016) has studied the role of sampling in big data analysis. The main focus of the study is the benefit of using sampling in the Artificial Intelligence field. The experimental results show that sampling not only reduces the data processing time, but also yields better results in some cases. The author's results also show that not all examples of sampling are as effective as using the original dataset. In a recently published work by Wu et al. (2014), the author implemented a GP classifier (DEAP) over Spark cluster to parallelise the fitness evaluation. He integrated a simple random sampling technique to speed up the training time. The author highlights that the theoretical trade-off between accuracy and sample size needs further investigation. He also stated that big data could be analysed incrementally to obtain results close, but not equal, to those computed using the entire dataset.

He and Garcia (2009) present a detailed explanation of why sampling may fail to return the same accuracy as using the original dataset. First, the authors explain how learning algorithms would fail to generalise inductive rules over a small sample size when the presented dataset has a high degree of features with a limited number of instances. If the sample space is sufficiently large enough, a set of generalised inductive rules can be defined for the data space. However, when the sample has a limited number of representative instances, the formed rules can become too specific, leading to overfitting. Also, sampling may hide some details about the data when targeting imbalanced datasets; one or more classes are represented with a small number of instances. Hence sampling may hide completely or maybe barely represent those instances in the small sample resulting in maybe a 100% accuracy while ignoring

the non-represented classes. If the sample size becomes larger, even using sampling (considering the above limitations) would lead to a point where the sample size is big enough to warrant parallel or distributed execution to obtain results faster. Hence, sampling and parallelism are best regarded more as complementary techniques than alternative ones.

Distributed systems can increase computing power by adding hardware resources. **Distributed computing** can be defined as using a distributed network of computing nodes to solve a single large problem by breaking it down into several tasks, where each task is handled by the individual computers of the distributed system. Here, it is essential to differentiate between parallel and distributed systems. A parallel computing system consists of multiple processors that communicate with each other using shared memory, whereas a distributed computing system contains multiple processors connected by a communication network. As a result, parallel systems have high bandwidth access to a shared memory, while distributed systems do not. Nowadays, this definition becomes tricky when considering modern multicore and GPGPUS, which have local memory with faster access than main memory. So, in principle, the approach to organising work between these cores would be similar to our presented designs, even if they are on a single node machine.

Many researchers addressing large data processing using distributed approaches would use MapReduce and Spark to speed up their implementation. In this study, we focus on Distributed Parallel GA implementations over the Spark platform. The following section briefly explains the MapReduce and Spark distributed frameworks.

2.7.1 Distributing Data

MapReduce and Spark are two frameworks that simplify the programming of parallel and distributed systems and provide scalability and efficiency.

MapReduce—proposed and developed by Google, and then reimplemented by Yahoo as the open-source version that is available nowadays—and Spark—developed at the University of California at Berkeley—are two famous distributed programming models that help programmers to process distributed data across multiple computing nodes. In the following two sections, the programming model of MapReduce and Spark will be explained.

2.7.2 Hadoop MapReduce

MapReduce is a paradigm of parallel programming (Dean and Ghemawat 2008) designed to simplify the processing of large datasets, as it abstracts many details of the underlying software and hardware platform by providing a standardised API. Hadoop MapReduce allows programmers to parallelise applications over multiple nodes/machines in an easy way without the need to manage a number of details such as data access, serialisation of tasks and data across the network, synchronisation, and any failure that could occur during these steps. The model works in two phases inspired by functional programming: the `map` phase and the `reduce` phase. The map and reduce functions are defined by the user to specify the processing logic and effectively constitute the MapReduce "program". Each phase has key-value (`<k; v>`) pairs as input and output. The map user-defined function phase takes each `<k; v>` pair and generates a set of intermediate `<k; v>` pairs. Then, all pairs' values with the same key are merged and are associated with the same key as a list (known as the shuffle phase). The reduce user-defined function takes that list as input to

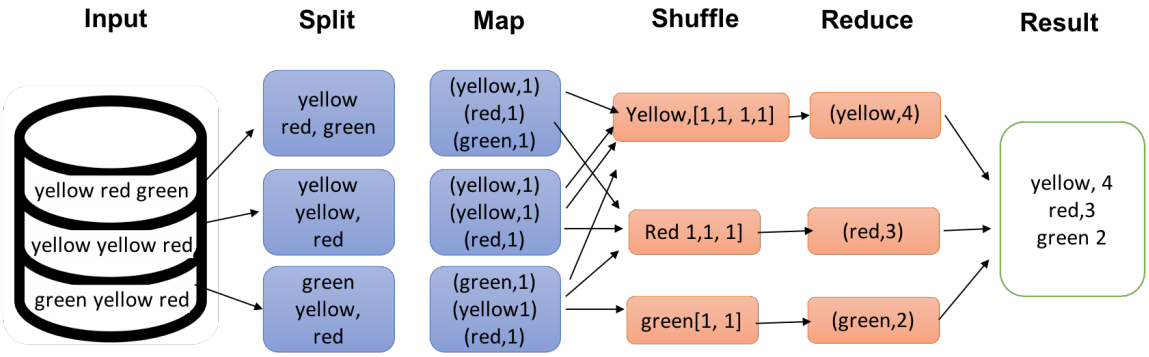


Figure 11: Map Reduce Word-count Example

produce the final values. In a MapReduce program, both map and reduce operations run in parallel. Inputs and outputs of a `map-reduce` job are typically stored in an associated distributed file system accessible from any computer of the used cluster. HDFS takes a dataset file, divides it into separate blocks and distributes them on the different data-nodes for storage. Figure 11 shows how a `map-reduce` job is executed to handle a simple program of word counting.

First, the file is divided into 3 splits (generally they are defined according to blocks in the underlying HDFS system). Each mapper will access one or more splits. The mappers will execute the map function defined by the word-count program independently, i.e. with no communication between them, and will create a `<key, value>` pair for every word where the key is the word, and the value is one. Then, the outputs of the mappers are shuffled, preparing for the reduce task. The shuffle creates a list of all values for each key. Each key is assigned to a reducer (also three in this example). The reducers take the `<key, list[values]>` and apply the reduce task to the list, in this word-count example it is summation, resulting in `<key,value2>` output where `value2` is the number of times each word (key) appeared in the input.

For this example, the user needs to specify the map and reduce task source code before the MapReduce job can be created and run on the Hadoop cluster.

In the MapReduce programming model, each map operation needs to be followed by a `reduce` operation. The MapReduce model is considered a good solution for applications that require a single pass over the dataset. On the other hand, a number of complex jobs cannot be efficiently implemented in a single `mapreduce` step; thus, this often leads to processing pipelines composed of a sequence of `map` and `reduce` jobs. The model is not designed to cache intermediate data in memory for faster performance; instead, it pushes intermediate data to disk after each step. Such overhead makes algorithms requiring many quick steps unacceptably slow using the MapReduce paradigm. Therefore, it is not the best paradigm for implementing parallel GA, which needs to run many iterations over the intermediate results. Compared with MapReduce, Spark can provide faster iteration on datasets that fit in the memory of distributed machines without pushing the intermediate data to the disk. Spark has a library of APIs that help the user to manipulate high-level typed objects masking the complex underlying (or the detailed structural) code.

2.7.3 Spark

Spark programs are executed in a distributed fashion by the driver and a set of executor processes. The driver manages the user application execution on the cluster, and executes the serial part of the code and distributes tasks to the executors in form of Resilient Distributed Datasets (RDD) operations.

RDDs are Spark's core abstraction: an immutable, distributed collection of objects that can be operated on in parallel. There are two ways to create RDDs: (1) parallelising an existing collection in the driver program; or (2) referencing a dataset

in an external storage system, such as a distributed file system (for example, HDFS). Internally, each RDD is partitioned, and those partitions are simply cached in the executors' memory if there is enough space. RDDs support two types of operations:

- Transformations: lazy operations that return another RDD. Some examples of transformations are `map()`, `flatmap()`, and `filter()`—see Table 1 for a description of key Spark operations. RDD transformations are designed for efficient parallel computation across partitions; e.g., `filter(pred)` can be executed in parallel in each partition to filter RDD elements, and `aggregate`(`start`, `seqOp`, `combOp`) aggregates elements in each partition first using a `seqOp` function (starting from `start`) and then aggregates partial results from each partition using function `combOp`. RDDs do not compute unless an **action** is performed on them. This design enables Spark to run more efficiently. For example, we can realise that a dataset created through map will be used in a reduce, and will return only the result of the reduce to the driver, rather than the larger mapped dataset. By default, each transformed RDD may be recomputed each time we run an action on it. However, we may also persist an RDD in memory using the persist (or cache) method, in which case Spark will keep the elements on the cluster for much faster access the next time you query it. There is also support for persisting RDDs on disk, or replicated across multiple nodes.

- Actions: operations that trigger computation and return values to the driver. Some examples of actions are count, top(), and savetofile(); more examples are found in Table 2. Each action in the user program will trigger jobs, and then Spark will split the work into multiple tasks that can be computed separately.

Transformation	**Description**

aggregateByKey(zeroValue) (seqOp, combOp, [numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different from the input value type while avoiding unnecessary allocations.
filter(func)	Return a new dataset formed by selecting those elements of the source on which func returns true.
flatMap(func)	Similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item).
groupByKey([numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
join(otherDataset, [numTasks])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.
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 of type T.
pipe(command, [envVars])	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.

reduceByKey(func, [numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
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.
sortByKey([ascending], [numTasks])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascendingargument.

Table 1: Spark RDD Transformations

Action	**Description**
reduce(func)	Aggregates the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
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.

Action	Description
first()	Returns the first element of the dataset (similar to take(1)).
take(n)	Returns an array with the first n elements of the dataset.
takeSample(withReplacement, num, [seed])	Returns an array with a random sample of num elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(n, [ordering])	Returns the first n elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile(path)	Writes the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
saveAsSequenceFile(path)	Writes the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that either implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable.
saveAsObjectFile(path)	Writes the elements of the dataset in a simple format using Java serialization, which can then be loaded usingSparkContext.objectFile().
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.

Action	Description
foreach(func)	Runs a function func on each element of the dataset. This is usually done for side effects such as updating an accumulator variable (see below) or interacting with external storage systems.

Table 2: Spark RDD Actions

Figure 12 shows how a word-count example is handled by Spark. This example is written in Scala language. Text file RDDs can be created using SparkContext's `textFile` method. This method takes a URI for the file and reads it as a collection of lines and generates an RDD of Strings. Once `textFile` is executed, then an RDD is created, and now transformations can be applied to this RDD. On the second line, a `.flatmap` operation would take every String with the line content and transform it to an RDD of words (the `split` function is used to divide the line into collections of words). The `.map` will transform the RDD of words into RDD of tuples (word, 1), also called key-value RDD, where the word represents the key and 1 is the value. Finally, `.reduceByKey` for each key (word), the values are reduced by summing them together. The reduce operation will apply the same summing operation to all previously processed elements. The third line in the code saves the count result as a text file back to HDFS.

Once the user submits their program, the Spark driver identifies the tasks that can be computed in parallel with partitioned data in the cluster. With these identified tasks, the Spark driver builds the program's dataflow, represented in a directed acyclic graph (DAG). Figure 13 shows the DAG for the word-count example. From this graph, Spark builds a parallel execution plan where tasks in each stage are bundled together and are sent to the executors to be run in parallel.

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

Figure 12: Spark: Word Count Code Example

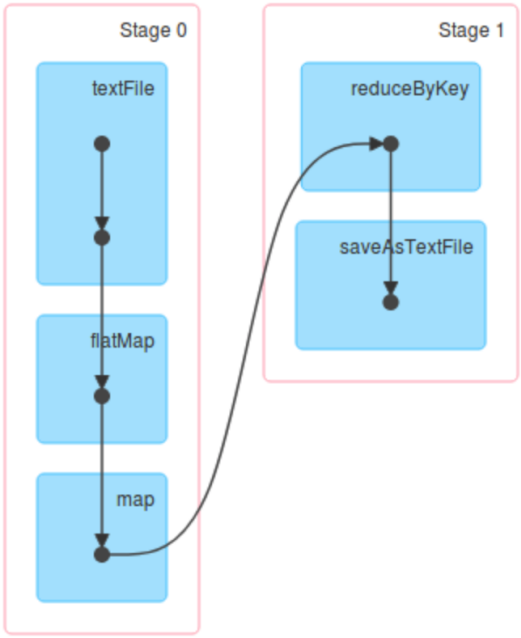


Figure 13: Spark DAG (Directed Acyclic Graph) for Word-count

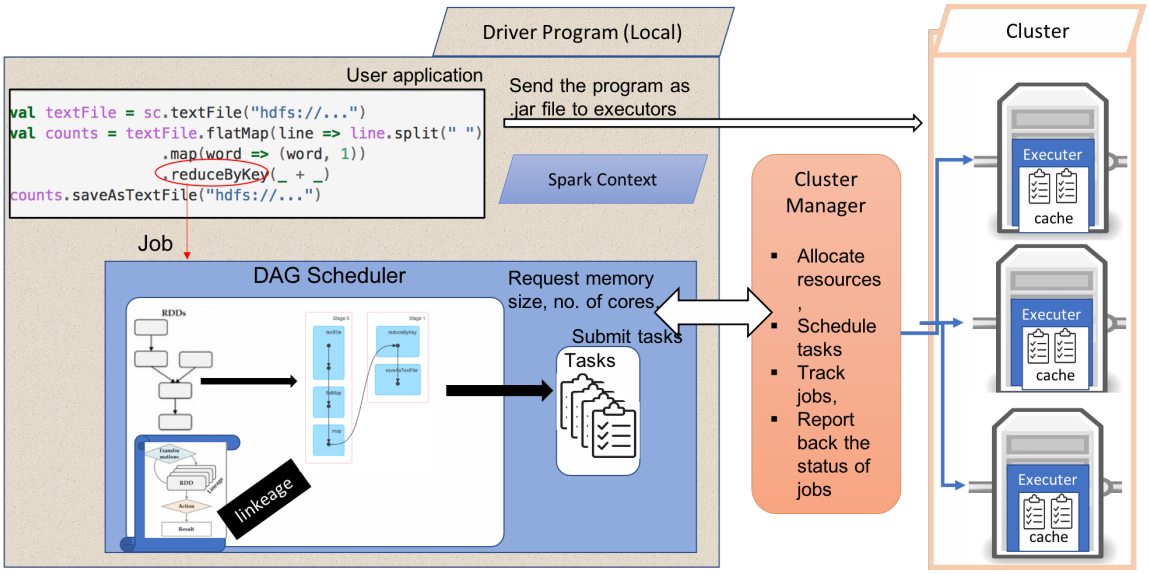


Figure 14: Spark: Job Execution

The driver program is the part of Spark which loads the user application. In the user program, once an action is encountered, a Spark job is created. A job is translated to a Directed Acyclic Graph of stages as mentioned before, where each stage may include multiple tasks. At the same time, the driver also performs optimisations, such as pipelining transformations.

The driver program requests resources from the cluster manager, such as the memory and cores to be used by the executors. The cluster manager launches executors, while the driver sends a .jar file to the executors to run. The cluster manager keeps track of jobs and reports back their status to the driver. Executors register themselves, through the cluster manager, with the driver program before starting the execution so that the driver program can monitor the executors' running of all the tasks by tracking the location of cached data based on data placement. Figure 14 gives further details of Spark's job execution.

The Spark driver converts the DAG into a physical execution plan which contains tasks that are bundled and sent to nodes through the cluster manager. The Spark driver keeps track of RDD lineages: a fault during the computation of an RDD operation would mean recomputing the content of RDD in case of an executor's failure. Spark allows checkpoint RDDs to disk to shorten the amount of recomputation necessary to recover from a fault.

2.8 Summary

This chapter has described the specific area of focus of this thesis: the classification of large-scale data. The chapter started with a general description of data mining: classification and knowledge representation. After defining and comparing the different approaches to creating a rule-based classifier and explaining the BioHEL classifier, GAs and their parallelisation methods were briefly described. Finally, the chapter concluded by describing two distributed frameworks that are used to implement distributed PGAs. The chapter aimed to give a description of just the background material needed to introduce the contribution of this thesis to the research field. We have omitted many details of the BioHEL background materials and a lot of hybrid PGAs modeling. The next chapter will present the work most related to the contribution of this thesis.

Chapter 3

Literature Review

PGA can be implemented through parallel programming using multiprocessor CPUs, however, when big data is addressed, vast processing power is needed to analyse it, and there is a limit to the number of cores in multi-core machines. Moreover, massively parallel hardware like GPGPUs have limited bandwidth to the local main memory. Thus, it is pretty inadequate to fit the large dataset in one machine's memory. Furthermore, fetching and storing data on a disk during processing would impact the execution time. Moreover, as GAs are iterative algorithms, this processing scenario requires an extended processing time. Distributed computing—, such as Cluster or Cloud Computing—is suitable for solving large-scale problems, making massive data processing feasible using the distributed processing power. These distributed systems are used to implement distributed PGA (DPGA), where the author defines the infrastructure and the communication topology. MapReduce and Spark frameworks have been used to generalise distributed parallel processing and enhance the system's reliability, uniformity, and scalability.

The chapter is structured as follows. First, Section 3.1 will discuss the work of

handling DPGA using custom distribution topology. Next, Section 3.2 will present existing research efforts on distributing GA using the MapReduce platform. Then, Section 3.3 will address the research work considering DPGA over Spark. Finally, Section 3.4 will summarize and conclude the related work.

3.1 Custom Distribution Topology

In large-scale data mining, where data is evenly distributed among multiple machines, a fundamental step is to distribute the data processing in order to take advantage of parallelism and in-memory computation. Due to the inherently parallel nature of GAs, many researchers have utilized this feature and have considered using GA in large-scale classification problems. Therefore, distributed-processing topologies have been developed for this purpose. In this section, we address three of the state-of-the-art applications: EC-Star, DXCS, and Flexible-GP. Even though Flexible-GP is concerned with genetic programming, the topology followed for distributing the data processing is worth mentioning here.

3.1.1 EC-Star

EC-Star is a parallel computing framework which uses a distributed Genetic Programming (GP) model upon a commercial volunteer resource (O'Reilly, Wagy and Hodjat 2013). The model consists of Evolution Coordinators, Evolutionary Engines, and Resources. It is elastic, since volunteer nodes *evolutionary engines* can independently enter the framework and the evolutionary algorithm can seamlessly integrate them as Evolutionary Engines. These volunteers offer to execute the "guest" program on behalf of EC-Star in their "spare time", using idle cycles that their primary

applications leave unused; they can withdraw from volunteering at any time. The volunteers do not need to communicate with each other, thus assuring their privacy. A volunteer can store a state file on its disk and shut down its program. Then, when it decides to volunteer again, it can resurrect itself, using the state file to resume where it left off. "Resources nodes" is the part of the framework which holds the data instances, and evolution coordinators coordinate the activities of the volunteer computer nodes. The main advantages that were reported for the EC-Star model are elasticity, robustness, and scalability. Nevertheless, if we assume that the system is so dynamic that there will be multiple departure and join during the computation the design forgoes the possibility of keeping track about which data has been processed where, which lead to a less efficient implementation.

3.1.2 DXCS

Many researchers have been extending traditional data mining approaches to distributed environments. DXCS is an example of the XCS distributed data mining system (Dam, Abbass and Lokan 2005). XCS is a genetics-based machine learning algorithm that applies reinforcement learning (RL) techniques for rule learning. The DXCS system consists of a number of clients and a single server. Each client is placed at a distributed site and runs a complete XCS that is trained independently on its own local database. Clients then forward their XCS models and their misclassified and untrained instances to the server. The server holds copies of all clients' models and applies the knowledge-probing approach (Guo et al. 1997) to combine the local models. The server combines misclassified and untrained instances and uses them as inputs for all copies of XCS local models available at the server; the server then trains an XCS to learn the mapping between the output of these local models and

the target class. The paper reports experiments with two clients and a server, which shows accuracy that is competitive with a centralised XCS. DXCS uses a fused model approach where several ensemble models are combined into a global model, however, such approach tends to generate models which are more complex to understand.

3.1.3 Flex-GP

Flex-GP is a large-scale genetic programming cloud computing system (Sherry et al. 2012). It uses the Amazon EC2 cloud service as its infrastructure and is based on ECJ—an evolutionary computation system written in Java—which adopts an island model with socket-based communication.

The work does not address the problem of large datasets —experiments consider a symbolic regression problem using 100 samples —and focuses instead on removing bottleneck of the implementation encountered as the system scale up the use of computation resources. The approach run a local GP on different islands each with its own local population of fixed size, thus resulting a linear increase of the global population as the number of islands increase. Results show a significant (40%) increase in accuracy with 256 islands but resulting in an 3× increase in computation time (750× increase in cost). Most of the increase in cost appears to be due to coordination cost which later focussed on a complete system re-design to improve scalability Derby, Veeramachaneni and O'Reilly (2013).

Arnaldo et al. (2015) introduced FCUBE, a machine learning framework that harnesses cloud computing to solve largescale supervised learning problems via massive ensemble learning. FCUBE helps machine learning researchers to integrate their learning algorithms into the framework. FCUBE treats those learning algorithms as

black boxes decoupled from the framework's code. However, learners must comply with a standardized specification, i.e. a predetermined list of input parameters and expected outputs. The framework has been demonstrated by integrating five different classifiers and deploying them in a massive data-parallel fashion on Amazon EC2. The five classifiers are; Rule List, GA-based, Rule Tree, GP-based, GP Function, Memetic Pittsburgh Learning Classifier, and Symbiotic Bid-Based Genetic programming. Even though the ensemble model might return high accuracy and can usually overcome the overfitting problem, the output of the ensembled model is hard to predict and explain. It is considered expensive in terms of both time and space.

As the above systems show adopting custom implementation for distributed computation can offer a great degree of flexibility, but also incurs in substantial development and maintenance costs. Next we describe instead DPGA approaches based on MapReduce and Spark, two general-purpose platforms that simplify the implementation of DPGA and are designed to scale to thousands of cores.

3.2 DPGA using MapReduce

Verma used the MapReduce model to scale the simple GA (Verma et al. 2009). In his work the authors use one map-reduce job for each generation of the GA, and a new map-reduce job is executed for each generation. individuals' fitness is computed using a map function, and the selection of individuals with the best fitness is then performed using tournament selection within the reduce function. At the end of each step, the population's individuals are saved on HDFS. The model had a big input-output footprint, as the entire population is saved to HDFS for each iteration of the GA, and this external storage access limit the iteration performance. Kečo and Subasi (2012) implemented a DPGA using MapReduce, with only one map-reduce

phase for all generations of GA. The author reconfigured Verma's implementation by moving most of the processing from the reduce phase to the map phase. This change reduces the amount of IO footprint as all processed data is kept in the memory instead of HDFS, but the drawback of this procedure is that different populations exist for each node, leading to a species problem in the GA, as the author noted.

Di Geronimo et al. (2012) proposed a DPGA based on Hadoop MapReduce for the automatic generation of JUnit Test suites, using the Master-Slave GA model developed on Hadoop MapReduce. The parallel genetic algorithm module lets the user manage the execution of GA and specify the SUT (Software Under Test). Upon termination of GA for the software under test, a test suite is returned to the user. The proposed algorithm is carried out in 3 phases: a split phase, in which the input acquires the current population from the HDFS and divides and distributes it among the mapper modules; a map phase, in which all mappers carry out the fitness evaluation task in parallel; and a reduce phase, in which the master uses a reduce job to collect the output of the mappers, that is, the entire population. Thus, selection, crossover, and mutation operators are applied to the current generation by the reducer to produce new offspring. The offspring is then saved into the HDFS, allowing the DPGA to validate the stopping criteria. In this phase, the work of the master module consists of notifying Parallel Genetic Algorithm to restart the computation invoking the map-reduce job for the new offspring created by Reducer. The paper preliminary analysis was to calculate the speed-up concerning the sequential execution. The results obtained highlighted that using PGA allowed for a time saving of over 50%.

MRPGA is an extension of the MapReduce model with a hierarchical reduction phase (Chao Jin 2008). The proposed work uses MapReduce as a distributed model to parallelize the GA algorithm. Each worker within the system sub-evaluates a set

of individuals and applies the GA operators. MRPGA follows the Pittsburgh GA approach, where an individual representing a complete solution. The system uses three MapReduce phases: 1) a local map within the worker, 2) local reduce, and 3) a global reduce process to complete one iteration. The local map process within a worker is used to partially evaluate individuals locally, then the local reduce process selects the local optimum individual for this worker. A global merging and sorting step is then used to select only the best individuals for all workers. These best individuals are fed into the last global reduce process and is for the global selection , called once at the end of each iteration of the loop. The complete process is repeated until the optimal individual meets the requirement set by the user. Two benchmark multi-objective problems were used in the experiments: DLTZ4 and DLTZ5. The results show that such an approach can upscale GA to 20 workers, after which adding more workers doesn't speed up the system any further. Hans, Mahajan and Omkar (2015) implemented a coarse-grained PGA model using Hadoop MapReduce to solve clustering problems. The authors formed chromosomes of centroids for each data split randomly and merged the centroids in the reduce phase.

Kondekar et al. (2012) proposed a MapReduce-based hybrid genetic algorithm using an island approach for solving the Time-Dependent Vehicle Routing Problem. MR-GEP is a scalable parallel evolutionary algorithm model based on MapReduce and presented in (Du et al. 2013). The authors propose a hybrid model which consists of two layers. The upper layer uses a coarse-grained computational model, while the lower layer uses a fine-grained master-slave model. The population is divided into a number of subpopulations equivalent to the number of processor "islands". Each node computes the fitness within a map function. Then, the processor applies the genetic operations to its subpopulation within a reduce function. The processors use a global shared memory to share intermediate results (population). The author used

speed-up as a measure for scalability. MR-GEP proved effective in improving the speed as more processors were added. 16 is the largest number of processors tested.

A practical application of DPGA on MapReduce is found in (Huang and Lin 2010). The DPGA model is used to solve a job-shop scheduling problem (JSSP). The author ran the experiments with various population sizes (up to 107) and clusters of multiple dimensions. The author followed a simple MapReduce approach where the mapper is used to evaluate the individuals' fitness (the makespans). Then, a partitioner assigns the emitted individuals from the mappers to the reducers according to their IDs. Finally, a reducer uses a tournament- based selection process to choose the good individuals and produces descendants by crossing over their chromosomes. As the author concluded in his work, using the MapReduce platform allows the exploration of population sizes significantly larger than those in typical experiments and reveals interesting trade-offs between population sizes and generations.

MapReduce does not utilize the full memory of the Hadoop cluster. In Spark, the concept of RDDs (Resilient Distributed Datasets) allows data to be saved in the memory and to be preserved to the disc (only if this is required). Spark can execute batch processing jobs substantially faster than the Hadoop MapReduce framework, just by reducing the number of reads and writes to the disk.

3.3 DPGA using Spark

As we discussed in (cfr §2.7.3), Spark allows a richer set of data-processing primitives compared to the MapReduce and supports a processing model that can benefit from in-memory data processing, thus eliminating a significant amount of disk I/O operations. Therefore, it is more suitable for data mining iterative computations. Hoewever, as we will show in this section, most parallel GA proposal focus on using

Spark to address hard optimisation problems which commend very large populations but where the environment required to evaluate the individual's fitness is *small*. The resulting overall approach is to represent the GA population as an RDD (cfr §2.7.3) and exploit Spark to parallelise the fitness evaluation which can be computed from relatively small (local) data. Spark is then also used to parallelise the genetic operators, which is key because of the large population by either using parallel transformations or adopting an island model where evolution can be performed in each population partition in an independent way.

Qi, Wang and Li (2016), and Paduraru, Melemciuc and Stefanescu (2017) applied genetic algorithms (GA) to software test suite generation. Both follow a Spark-based architecture that parallelises fitness evaluation and genetic operators. A single individual in the population represents an input test data for the evaluated program. The input data associated to an individual can even represent complex data such as a file uploaded by a user or online data that a software application has to process. The main objective of the test data is to execute instructions of the program being evaluated that are difficult to reach using common input. To guide the individuals toward inputs that lead to rare paths of a program, the fitness function associates higher scores to individuals that take uncommon paths when the program is tested against them. The works focus on evolving very large populations (up to 3 million individuals), using an RDD to partition the population. The approaches assume that a form of shared memory or data replication exists which is not applicable in the case of large-scale data.

The work in (Hu et al. 2017) addresses instances of water pollution, where a viable solution is to install water quality-monitoring sensors in water supply networks (WSNs) for real-time pollution detection. The work uses GA to optimize the sensor placement problem in large-scale WDSs, with the objective of minimising the

impact of contamination events. The author specifically highlighted the need for a large-scale water distribution system which would result in significant computation overhead. The author proposes a two-phase Spark-based genetic algorithm (SGA). After the driver initializes the population, a `map` function parallelises the population into different partitions of RDD so that different workers evaluate the fitness of different individuals on different workers and then a `collect` return the population to the driver to carry out the genetic operators. Experimental results show that SGA outperforms other traditional algorithms in both accuracy and efficiency, thus demonstrating the feasibility and effectiveness of the proposed approach.

A distributed version of the GA algorithm implemented on Spark was used to solve the traveling salesman problem more efficiently than earlier implementations (Lu, Hwang and Huang 2020). The work in (AlJame, Ahmad and Alfailakawi 2020)) proposes a Spark Whale Optimization Algorithm (Spark-WOA) to enhance WOA execution time and reduce its computational complexity. To evaluate Spark-WOA , two types of benchmark were used: unimodal and multimodal functions. The performance of Spark-WOA was compared to a MapReduce version of WOA. The results demonstrate that Spark-WOA outperforms MR-WOA, running at more than 300 times the speed of the latter. The work highlights in particular Spark's advantages over MapReduce in terms of its in-memory computing capabilities.

As we mentioned, all of the above works use the Spark RDDs to represent the individuals of the GA populations. Some try to optimize functions, while others aim to optimize a network design. None of the above works address large-scale data for which the dataset needs to be partitioned and distributed over the cluster nodes. We now mention some approaches that instead addressed the use of GA-based data-mining on large datasets.

In (Cheraghchi, Iranzad and Raahemi 2017), the author addresses the problem of

mining for outliers in subspaces with relevant attributes for high-dimensional large-scale datasets. The author implemented a grid-based GA in Apache Spark to search for subspaces that are candidates for outlier detection (with regard to the subset of features in the subspace within the high-dimensional data), and to find the sub-space with maximal sparsity. The algorithm runs targeting projections from the full-dimension data space into lower dimensions with the best fitness values (the most sparsity). One GA iteration contains several map & reduce functions. In every iteration, each dimension in a subspace is divided into an equal number of slices which would result in a fixed number of cells. Individuals are a set of sub-spaces which are initially chosen randomly. Each data point in the input dataset is mapped into a cell for all subspaces of a population in one generation. The result of this map is an (individual,cell),1), where 1 represents a data point that could be mapped to that subspace/cell. Of course this map would result in a large amount of data, significantly larger than the dataset itself. Then, a reduce-by-key (where the individual-cell combinations represent the keys) is used to sum all 1s and end with an individual- cell-sum intermediate RDD. Another map-reduce combination is used to determine an RDD of the population's individuals and their total fitness. A final collect function returns the evaluated individuals to the master node. The master node is responsible for the evolution processes after the offspring is created, and another iteration starts. After each iteration, the algorithm reserves a list of elitists which is carried to the next generation. The experiment uses 100 generations and a population size of 50. For Spark settings, there are 8 partitioned RDDs for the purpose of diversity, and to evaluate more sub-spaces, the algorithm used keeps members of each population unique (a sub-space already seen before is not evaluated again). A dictionary data structure is used for this purpose. If crossover and mutation lead to duplicate individuals being generated, the duplicates are replaced

with new random individuals. The data is original mp3 music files from archive.org, divided into 16 categories based on their meta tags and instruments used. The data was created by concatenating segments of 190 music tracks. Mixed sound tracks were converted into 1,370 dimensions, with 77,000 instances, and a size of 2.7 GB. The data is labelled "outlier" and "inlier". The result shows that there were some missing relevant subspaces which the GA did not find during its iterations. The author attributed the resulting problem to the heuristic nature of GA.

A parallel work to my own is found in (Hmida et al. 2019), where the author reshaped an existing GP implementation (DEAP) by parallelising evaluations on Spark. The author added a sampling step whereby the large data were fitted into small clusters. Evaluating the performance of DEAP-Spark using different combinations of population sizes and number of generations, the results are encouraging and demonstrate that Spark is an efficient environment for distributing GP evaluations. The work has been tested on a cluster of 4 nodes with 64 processors. Experiments are conducted according to Higgs boson classification with different settings. In our evaluation section, we compare our results with the performance of DEAP-Spark.

3.4 Summary

Throughout this chapter, we have covered the different available scenarios of DPGA implementation in the literature. We divided the literature into three sections: implementing DPGA using custom distribution topology; MapReduce; and Spark. The next chapter describes our first contribution: designing and implementing PDMS and PDMD our two DPGA models over Spark. The two proposed models are then used to reimplement BioHEL.

Chapter 4

Parallelised GA for Partitioned Data

This chapter describes the contributions of this thesis to the area of PGA scalability, focusing on a data partitioning approach, Two PGA models are presented: Partitioned-Data Master-Slave (PDMS) and Partitioned-Data Multiple-Deme (PDMD). These two models are then used to obtain two distributed versions of the BioHEL classifier. In order to support their implementation in large-scale scenarios, we build them on top of the Spark framework. We evaluate both with three large datasets. Moreover, we evaluate their scalability and how their performance changes in different configurations.

The chapter is structured as follows. First, Section 4.1 will define data partitioning and describe the models under this definition. Then, Section 4.2 will provide an extensive description of the BioHEL reimplementation considering these two models. Section 2 will report the results of the experiments and will also provide an investigation of the effects of both the migration frequency and the population size on the

performance of the PDMD model. Section 4.4 will provide a comparison between the two implemented models and a recent Genetic Programming (GP) implementation (DEAP) over Spark (Hmida et al. 2019). Finally, Section 4.5 will summarize the contribution presented in this chapter and the results obtained.

4.1 Data Partitioned Models

Traditionally, PGAs have been used to speed up the computation of GA in complex optimisation problems (including combinatorial problems such as the travelling salesman problem (Varadarajan and Whitley 2021; Wang et al. 2005)). In many such scenarios, the fitness evaluation on a single candidate solution (population individual) can involve relatively small data access and computational costs; thus, parallelisation can be efficiently implemented by distributing the individuals across the computing nodes. Each node can compute the fitness of the assigned individuals, and data can be replicated over the nodes or accessed in a shared memory subsystem.

In contrast, it is more challenging to apply PGAs to large datasets such as the training instances in big-data classification problems. Data access costs need to be considered in the parallelisation strategy: each computational unit should access data locally for the purposes of fitness computation. However, given the size of the data, this local access cannot be given to the whole dataset, but to only a part of it, which we call the *partition*.

Our proposed approach for scaling PGA is based on four basic principles related to partitions: (i) partitions must be kept in memory for fast access required by GA iterative processing; (ii) partitions must be distributed, for scaling memory storage and accessing bandwidth, in addition to harnessing distributed processing power; (iii) processors should be restricted to local data access during GA iterations; and

(iv) remote access should be tightly controlled to reduce network contention.

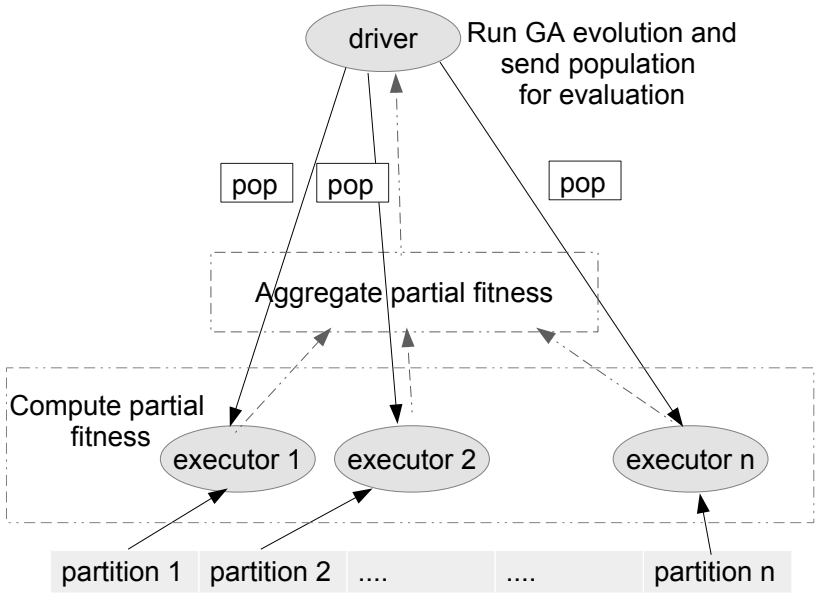
Based on these principles, we will now outline two possible parallelisation models: the Partitioned-Data Master-Slave (PDMS) model and the Partitioned-Data Multiple-Deme (PDMD) model.

Partitioned-Data Master-Slave Model (PDMS) The Partitioned-Data Master-Slave model is a typical master-slave model in which the master handles the core GA algorithm and the slaves are responsible for computing the fitness of the population's individuals. In contrast to the typical master-slave model, where the master sends different subsets of the population's individuals to different workers, in PDMS the master node sends a copy of the complete population to all the slaves, and each slave then computes a *partial fitness* value for the individuals on the local data partition. Next, the slaves send their results to the master which, in turn, aggregates all partial fitness values and uses them in the following GA algorithm steps. Our design for this model is represented in Figure 15a. As the master will be sending the population and collecting the fitness values in every iteration, communication overhead can be a disadvantage of this model. It is worth mentioning that in this scenario it must be possible to compute the global fitness level by efficiently combining the partial fitness values computed on the different partitions; in other words, the fitness function is required to be associative and commutative. In this way, by co-locating the computation of the partial fitness value with the data in a partition, and later combining the results to compute the global fitness value, PDMS requires less communication overhead than the approach of providing remote access to the full dataset for each processor to directly compute the global fitness values.

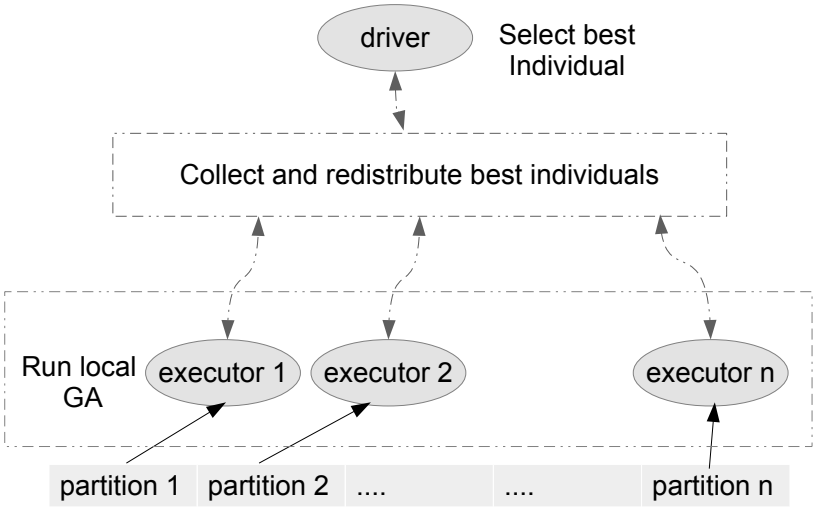
Partitioned-Data Multiple-Deme Model (PDMD) Figure 15b represents the basic design for the Partitioned-Data Multiple-Deme (PDMD) model. As shown, each node runs the complete GA on the local data partition, that is, each subpopulation is initialised, evaluated and evolved only with respect to the local data of that node. Occasionally, migration occurs; nodes exchange their best individuals with randomly selected nodes to allow interaction among individuals in different populations. Finally, a solution is selected after an individual is globally nominated as the best individual. It is worth mentioning that in such implementations, as individuals are not evaluated over the complete dataset, their fitness values may not reflect their global quality. We will return to this aspect when discussing the implementation in section 4.2 , and also in the evaluation in section 4.3.2 where we examine the quality of the PDMD solution.

4.2 Implementation

Like the original master-slave and multiple-deme parallelisation models, the principles of PDMS and PDMD can be applied to any GA algorithm. However, the resulting benefits would depend on their reification in a specific parallel GA. We have reimplemented BioHEL, turning it into a distributed GA according to the PDMS and PDMD models. To support the implementation of PDMS and PDMD models in large-scale scenarios, we have built on top of Spark (Zaharia et al. 2012). In this section, we will firstly explain a redesigned pseudocode for the original BioHEL, which simplifies the injection of changes into the workflow. Then, we will detail the changes to BioHEL according to these two proposed models and implemented using Spark's primitives.



(a) PDMS



(b) PDMD

Figure 15: Partitioned-Data GA Models.

The BioHEL function (lines 1–10) represents the main loop of BioHEL which builds a classifier as a list of rules, terminating when no further rules can be derived. This function calls two other functions: `findRules` and `bestRule`. `findRules` (lines 11–20) represents the execution of the GA, (runGA of algorithm (4) the typical GA steps (2). `findRules` returns candidate solutions, while `bestRule` (lines 21–27) selects the best rule, checks its validity as a rule to be added to the rules set (if the associated class is the majority class of the matched examples) and deletes the matching instances from TrainS.

4.2.1 BioHEL PDMS Implementation

The main goal of the PDMS model is to speed up the most costly step, that being the computation of the fitness of the population's individuals. All the evolutionary work is handled by the Spark driver, which sends the individuals across the executors to compute the fitness value. Each executor computes a partial fitness value based on the individuals in the local partition, while the master node performs the aggregation.

The PDMS BioHEL implementation follows the main structure of BioHEL, but with specialized `findRulesPDMS` and `bestRulePDMS` functions. The training set is stored as an RDD (TrainS, underlined, in Algorithm 5) with instances equally partitioned across executors. In `findRulesPDMS` (lines 1–10) the population is initialised as follows: a sample primitive is used to derive a sample RDD from TrainS, then `collect` is used to retrieve the sample from the executors and make it available to the driver (line 2). The fitness computation is then parallelised using the `aggregate` function (lines 3,8): a copy of the population is transferred to each executor and used in the *partialFitness* function to compute the fitness of the population on the local TrainS partition. In BioHEL, the data-dependent part of the fitness function consists

Algorithm 4: BioHEL general workflow.

```
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 5: PDMS BioHEL.

```
1  findRulesPDMS(TrainS):
2      pop = TrainS.sample(...).collect();
3      pop.fitness = TrainS.aggregate(zeroes, partialFitness(pop),
         mergeFitness);
4      for iter = 1 to numIter do
5          offsp = pop.selection();
6          offsp.crossover();
7          offsp.mutation();
8          offsp.fitness=TrainS.aggregate(zeroes, partialFitness(offsp),
             mergeFitness);
9          pop.replacement(offsp);
10     return pop.best();
11 bestRulePDMS(candidates,TrainS):
12     bestRule = candidates.bestFitness();
13     matchedCl = TrainS.filter(_.matches(bestRule))
         .map(_.class).countByValue();
14     if bestRule.class = majorityClass(matchedCl) then
15         TrainS = TrainS.filter(!_.matches(bestRule));
16         return (bestRule, TrainS);
17     else return (null, TrainS);
```

of computing the confusion matrix of the rule to be evaluated, thus partialFitness computes four integers for each rule: true-positives, false-positives, true-negatives and false-negatives. Then a *mergeFitness* function is used to aggregate the counters of each rule from each partition, thus obtaining the total fitness value.

`bestRulePDMS` (lines 32–38) is then used to select the best rule among repetitions (line 33); and the stopping condition is evaluated by comparing the class of the rule and the most frequent (*majorityClass* line 35) of the classes of the matched instances (*matchedCl*, computed using `countByValue`, `map` and `filter`, line 34). If the rule is accepted, a new TrainS RDD is created by removing the instances not matched by the selected rule (line 36).

4.2.2 BioHEL PDMD Implementation

In the multiple-deme model, each island is an executor. In addition to the training instances TrainS, also the population pop and the incoming migrants mig are represented by RDDs, partitioned across executors. The driver instructs each executor to evolve each sub-population in parallel by invoking the `zipPartitions` function (line 4 in Algorithm 6). `zipPartitions` allows the combining of partitions of several RDDs that are colocated on the same executor by providing a user-provided function. This (anonymous) function (lines 5–16) first initialises the population in the local partition (popPart) by taking a sample of size $n = N/numPartitions$ from the local training set (TrainSPart, line 7), then repeats the main evolutionary loop until migration takes place. Finally the population in the local partition popPart is returned (line 16), becoming a partition of pop referenced by the driver (line 4). The driver randomly redistributes the best half of the population in each partition as migrants (line 18) which will be merged with the local population in the next

Algorithm 6: PDMD BioHEL.

```
1  findRulesPDMD(TrainS):
2      pop = ∅; mig = ∅;
3      for migration=1 to numMigrations do
4          pop = TrainS.zipPartitions(pop, mig)(
5          (TrainSPart, popPart, migPart)⇒
6              if popPart = ∅ then
7                  popPart = TrainSPart.sample(n);
8              popPart.replacement(migPart);
9              popPart.fitness=doFitness(popPart, TrainSPart);
10             for iter = 1 to numIter/numMig do
11                 offsp = popPart.selection();
12                 offsp.crossover();
13                 offsp.mutation();
14                 offsp.fitness=doFitness(offsp,popPart);
15                 popPart.replacement(offsp);
16             return popPart;
17         );
18         mig =pop.mapPartitions(best(n/2)) .repartition();
19     return pop.mapPartitions(best(2)).collect();
20 bestRulePDMD(candidates,TrainS):
21     candidates.fitness=TrainS.aggregate(zeroes, partialFitness(candidates),
         mergeFitness);
22     bestRule = candidates.bestFitness();
23     matchedCl = TrainS.filter(_.matches(bestRule))
         .map(_.class).countByValue();
24     if bestRule.class = majorityClass(matchedCl) then
25         TrainS = TrainS.filter( !_.matches(bestRule));
26         return (bestRule, TrainS);
27     else return (null, TrainS);
```

migration (line 8). After *numMigrations*, the best two individuals from each partition are collected by the driver as candidate rules (line 19). Candidate rules for every repetition are then evaluated globally (`bestRulePDMD`, line 22), the best being selected and used for termination detection and filtering as in the original BioHEL.

4.3 Evaluation

Our primary goal in the evaluation is to assess the scalability of the PDMS and PDMD models and their relative performance in terms of accuracy and execution time. The result leads us to investigate how the performance of the PDMD BioHEL is affected by changes in key parameters such as population size and multiple-deme migration frequency. Therefore, we shall evaluate the performance of PDMD against different settings for both parameters separately. Then, in order to put the result into perspective, we shall compare our models with DEAP Spark in terms of accuracy and execution time.

4.3.1 Experimental Setting

In order to train and test the PDMS and PDMD models, we used the HEPMASS dataset, obtained from `https://archive.ics.uci.edu/ml/datasets.html`, and KDD-cup99 (full version) obtained from `https://www.openml.org/d/1110`. Table 3 shows the two datasets composition.

For the BioHEL configuration, we set the algorithm parameters using the same configuration provided in (Bacardit and Krasnogor 2006), summarised in Table 4.

We ran the experiments on a cluster with 17 servers, each with 2x Intel Xeon E5520 CPUs running at 2.27GHz. Each server has 8 cores and 12GB RAM. In order

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

Table 3: KDD and HEPMASS data composition

Parameter	**Value**
crossover prob. Rule sets per ensemble	0.6
Selection algorithm	tournament
Tournament size	4
Population size	500
Individual-wise mutation prob.	0.6
Default class policy	MAJOR
Iterations	50
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.

to test the scalability of the two models, we ran configurations with different numbers of executors (thus increasing the parallelism level) and data equally partitioned between them (1 partition per executor). We report the results for both models using a cluster of 8, 16, 32, 64 and 96 executors, which have been selected according to both memory limitations and the number of processing units available. All results are averaged over 20 runs using 10-fold cross-validation. We report average and 95% confidence intervals.

4.3.2 Scalability

In this section we evaluate the scalability and performance of both implemented models. Performance evaluation results are summarized in Table 5. the following is an outline of the scalability of both PDMS and PDMD.

PDMS. Table 5 shows the PDMS results as the parallelism level p is increased. As expected, PDMS maintains a constant degree of accuracy 83.67% for HEPMASS and 99.7% for KDD-cup which is in line with centralised BioHEL. This is due to the fact that, in PDMS, increasing the number of partitions does not impact the accuracy of the fitness computation, thus leading to the same solution quality.

Figure 16 shows the speed-up with respect to the execution time with 8 partitions, i.e. T_8/T_p. PDMS shows a good degree of scalability overall—as the cluster size increases, the time decreases. For HEPMASS, scalability is linear up to 64 cores, and sublinear at 96 cores, while for the smaller KDD-Cup it is sublinear starting from 32 cores upwards. The maximum speed-up is 11× for 12× cores for HEPMASS and 7× for 12× cores for KDD-Cup. This reduction in scalability is due to thread contention and synchronisation overhead between the driver and the executors in

Model	Data	P	R	Accuracy	Time
PDMS	Hepmass	8	65 ±0	83.69 ±0.1	169950 ±5300
		16	67 ±0	83.67 ±0.1	89400 ±1156
		32	69 ±2	83.67 ±0.1	44730 ± 987
		64	70 ±2	83.65 ±0.2	20040 ± 229
		96	70 ±2	83.65 ±0.2	16500 ± 250
	KDD	8	30 ±2	99.73 ±0.3	40030 ± 800
		16	29 ±2	99.70 ±0.0	16900 ± 450
		32	29 ±3	99.70 ±0.1	13080 ± 290
		64	28 ±2	99.68 ±0.1	6300 ± 210
		96	28 ±3	99.68 ±0.2	5800 ± 80
PDMD	Hepmass	8	45 ±1	81.93 ±0.2	16770 ± 494
		16	40 ±2	81.45 ±0.2	4700 ± 200
		32	40 ±2	79.89 ±0.2	2700 ± 145
		64	39 ±3	79.87 ±0.3	1190 ± 95
		96	38 ±3	78.67 ±0.6	900 ± 200
	KDD	8	25 ±4	99.69 ±0.0	4558 ± 500
		16	21 ±3	99.65 ±0.2	1300 ± 70
		32	20 ±6	99.58 ±0.1	730 ± 50
		64	19 ±3	99.44 ±0.1	420 ± 35
		96	13 ±5	93.20 ±1.0	260 ± 15

Table 5: PDMS and PDMD Scalability. P is number of partitions, R is number of rules.

collecting the fitness results.

PDMD PDMD maintains a relatively good degree of accuracy compared to PDMS. For HEPMASS, accuracy decreases by 0.5% up to 16 partitions and then settles at around -2.0% at 32 and 64, falling by 1.2% at 96 partitions. Two possible factors contribute to this decrease in accuracy. Firstly, the fitness is computed locally in each partition, thus fewer rules are discovered and the quality is potentially lower. Secondly, there is a drop in the local subpopulation size. As we scale the cluster up to 64 and 96 partitions, the local population size decreases to 8 and 5 individuals,

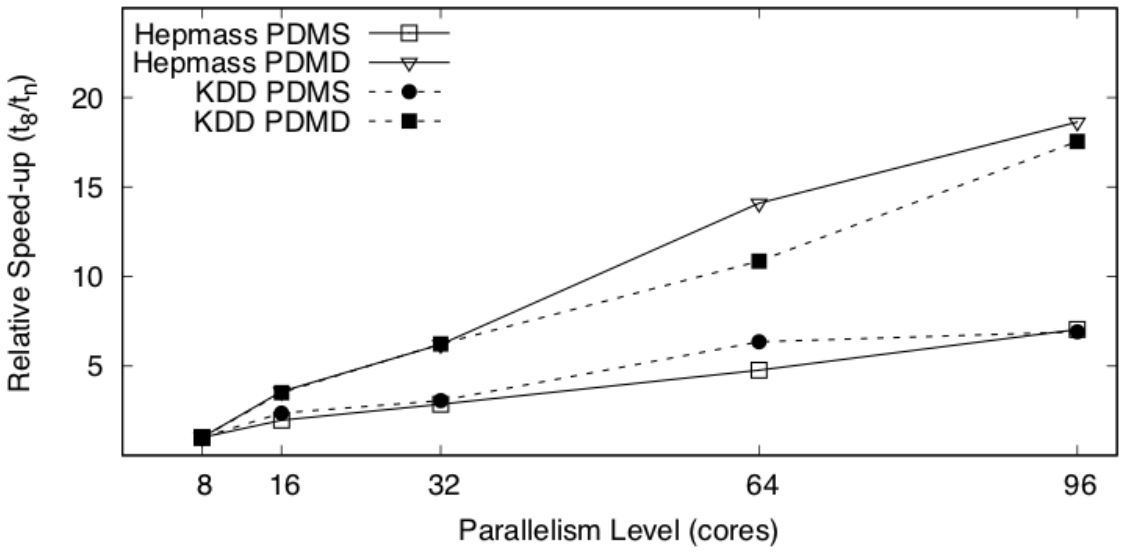


Figure 16: PDMS and PDMD Scalability

respectively. We will report the influence of the population size in further detail in section 4.3.4. For KDD-Cup, the degree of accuracy is close to the significance threshold up to 64 partitions but then falls by 6.5% at 96. This increased drop is likely due to the nature of the dataset, which has fewer instances, higher dimensionality and a higher number of classes.

As seen in Figure 16, PDMD shows good scalability up to 96 cores for both datasets, with a 20× speedup for 12× cores. This super-linear scalability is attributable to two factors: the reduction in the fitness computation time and the reduction in synchronisation overhead. The fitness computation time decreases quadratically: the size of both the local population and the local training dataset decrease linearly as the value of p increases, an effect that is dominant at low p values, resulting in a quadratic time reduction from 8 to 16 cores; as p increases however other linear factors start to dominate, such as the filtering of the training

Dataset	P	mig-8				mig-2				no-mig			
		Accuracy		Time		Accuracy		Time		Accuracy		Time	
Hepmass	16	82.03	0.2	8450	700	81.45	0.2	4700	200	79.61	0.2	1790	50
	32	79.88	0.2	4450	260	79.89	0.2	2700	145	79.10	0.2	1960	100
	64	78.88	0.2	2650	220	78.87	0.3	1194	95	78.01	0.5	1100	20
	96	78.42	0.2	2070	200	78.67	0.6	900	200	77.30	0.5	1300	400
KDD	16	99.71	0.1	3000	200	99.65	0.2	1300	70	99.60	0.3	850	100
	32	99.60	0.2	2000	250	99.58	0.1	730	50	96.7	0.4	800	80
	64	99.46	0.2	1020	70	99.44	0.1	420	35	93.50	1.0	390	40
	96	96.70	0.6	440	30	93.20	1.0	260	15	90.77	0.8	230	10

Table 6: Impact of migration on PDMD accuracy and execution time.

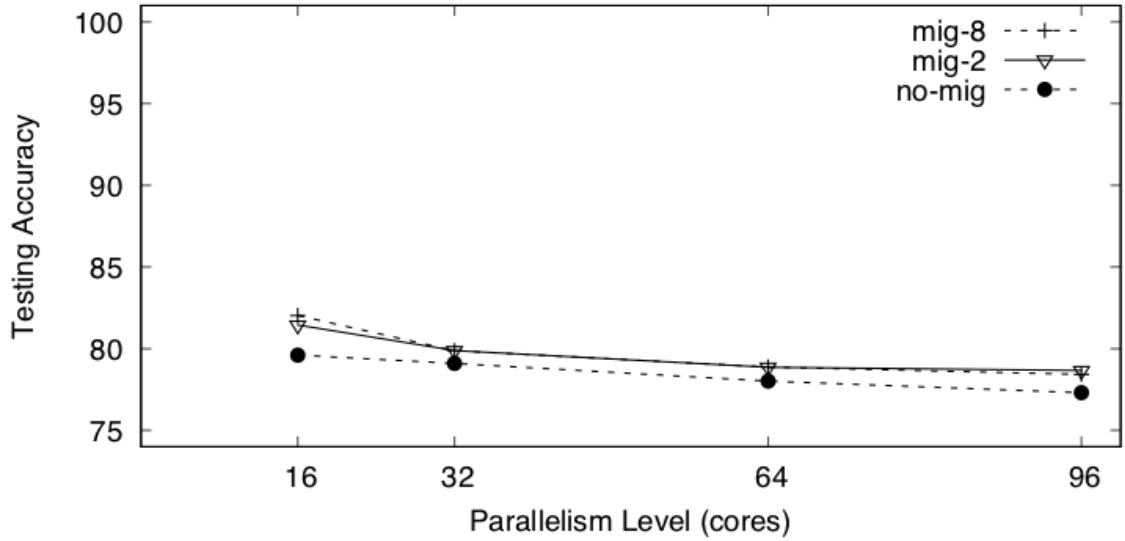
set characteristic of IRL which improves linearly as the number of cores increases. Finally, PDMD is also less affected by synchronisation overhead as islands do not need to synchronise at every fitness evaluation but only at migration intervals.

4.3.3 Migration

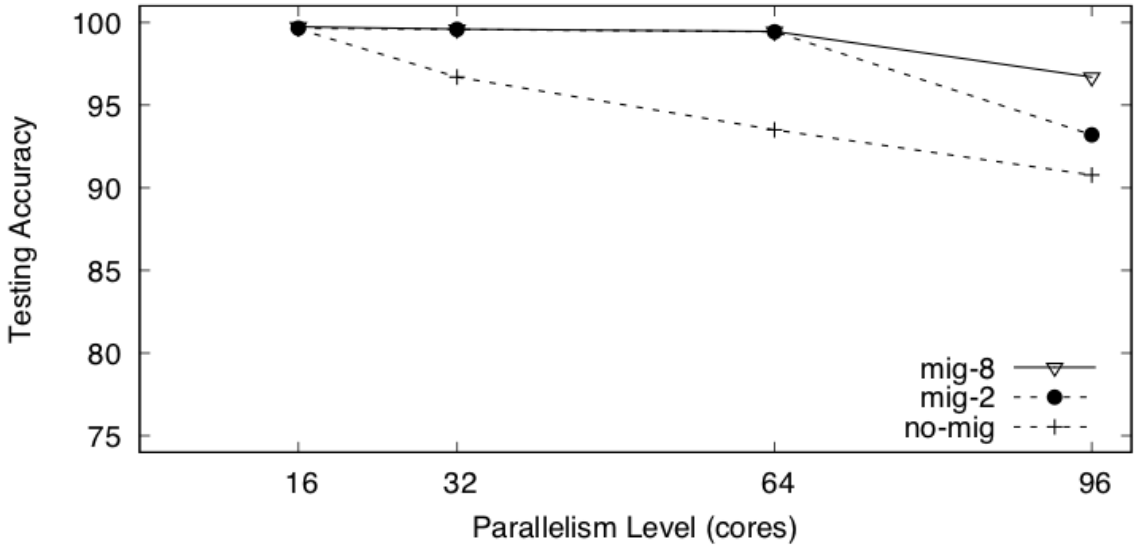
In order to study the influence of migration on PDMD accuracy and execution time we tested the impact of a change in the frequency of migrations ($numMigrations$) while performing GA iterations. In addition to the default case, 2 migrations per 50 iterations (`mig-2`), we tested with 8 migrations (`mig-8`) and with no migrations (`no-mig`).

We performed experiments for parallelism levels 16, 32, 64, and 96 for both HEPMASS and KDD-cup datasets (Table 6).

Accuracy results are shown in Figures 17a and 17b. For Hepmass `mig-2` improves accuracy of approximately 2% for 16 partitions with respect to `no-mig`, and of 1% for 32, 64 and 96 partitions, while generally, `mig-8` does not improve accuracy further. For KDD-Cup there are no differences at 16 partitions, but at 32 and 64 partitions `mig-2` is able to keep the accuracy close to the maximum, improving accuracy by



(a) Hepmass



(b) KDD-cup

Figure 17: Impact of migration on PDMD accuracy

$4-6\%$ effectively negating the accuracy loss due to increased partitioning; finally at 96 partitions `mig-2` is no longer enough and `mig-8` starts providing benefits improving accuracy of 6% over `no-mig`. Migration has a significant impact on runtime performance due to synchronisation effects: `mig-8` total execution time is $33-65\%$ higher than the default case, while `no-mig`, results in 60% lower execution time at 16 partitions and close to `mig-2` at 32, 64, and 96 partitions.

4.3.4 Impact of Population Size on PDMD

As our results report, scaling the PDMD model results in an accuracy drop. One factor which highly influences the accuracy is the subpopulation size. A small population would lead to a quick convergence to a poor solution. Of course, migration would help boost the fitness value, but as the results show, it could not support the model to reach the accuracy accomplished by the centralised model. In the literature, it has been shown that a too-small population can be the cause of poor solution (Koumousis and Katsaras 2006; Pelikan, Goldberg and Cantu-Paz 2000; Piszcz and Soule 2006)and that the population size has to be large enough to provide good coverage of the solutions space (Lobo and Lima 2005; Vrajitoru 2000). Thus, we rerun all the parallelism scenarios with 250, 500, 1000, and 1500 population sizes for the HEPMASS dataset to study the population size's influence on accuracy and execution time. For each population size, we used the average of 10 runs for 10-fold cross-validation. Table 7 reports the changes in accuracy and time. In general, as we increase the population size, the accuracy improves of 2% for 64 and 96 partitions. Furthermore, reducing the population size to 250 results with 30% drop in accuracy for both scenarios. In terms of execution time, Figure 18 reports the trends of increasing the population size at the different partitions. As the figure

no. nodes	250				500				1000				1500			
	time		**acc**		**time**		**acc**		**time**		**acc**		**time**		**acc**	
16	3000	±150	79.10	±0.2	4700	±320	81.45	±0.1	6300	±370	82.48	±0.1	8700	±390	83.14	±0.1
32	2340	±200	78.03	±0.4	2700	±210	79.89	±0.2	3730	±300	81.51	±0.1	5030	±340	81.87	±0.1
64	1100	±150	74.64	±0.7	1194	±100	78.87	±0.2	2080	±170	80.21	±0.1	3000	±150	80.94	±0.2
96	750	±150	58.5	±1.2	900	±70	78.67	±0.3	1100	±85	80.60	±0.2	1200	±90	80.67	±0.2

Table 7: Impact of population size on PDMD accuracy and execution time (HEPMASS dataset)

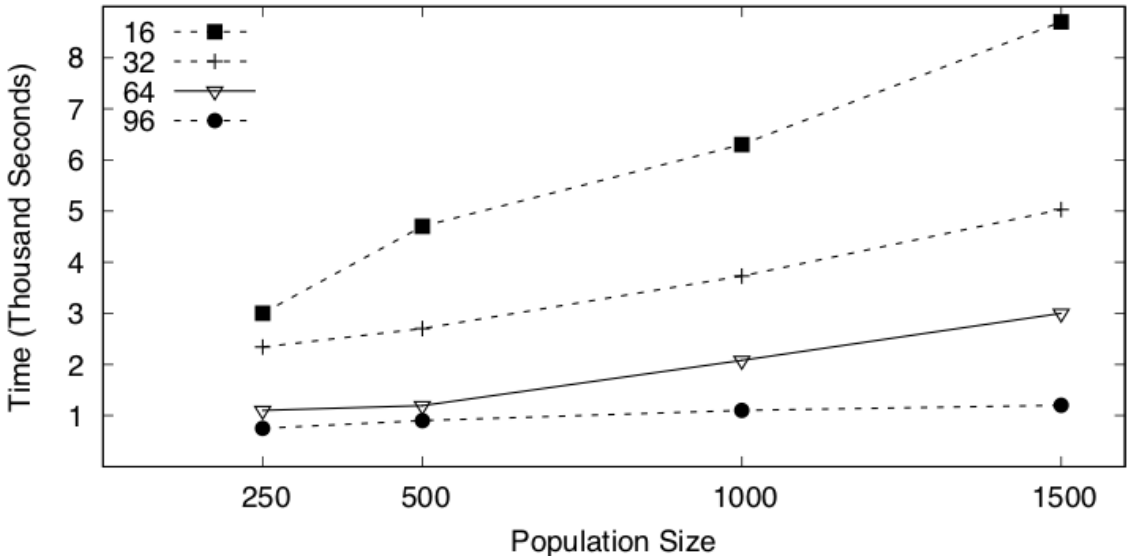


Figure 18: Population size effect on execution time (HEPMASS dataset)

shows, the total time increases linearly with the increase in population size for 16, 32, 64, and 96 partitions.

4.4 Comparison with DEAP-Spark

To put the above results into perspective, both in terms of model efficiency and absolute performance, we have compared PDMS, and PDMD BioHEL implementation against the DEAP GP algorithm which is also implemented on the Spark framework (Hmida et al. 2019). In this work, the author presents a GP implementation (DEAP) modified by distributing the fitness evaluation on a Spark cluster. Experiments are performed on Higgs Bosons dataset obtained from `https://archive.ics.uci.edu/ml/datasets.html`. Figure **??** shows the dataset features. Comparing the performance of DEAP-Spark to the sequential DEAP algorithm, Hmida shows that Spark is an efficient environment to distribute GP fitness evaluations. Also, Hmida has integrated a simple sampling technique that preserves learning performance while providing the possibility to probe GP with large populations or for a high number of generations. We compare our two models against DEAP-Spark with the complete Higgs dataset results since we target large-data processing. In Figure 19, we report the accuracy and the execution time performance for different population sizes reported by Hmida. We have used the same cluster size and we ran the PDMS with 65, 125, 250, 500, and 1000 population sizes, as well as, the PDMD with 500, 1,000, 2,000, 4,000, 8,000, 16,000, 32,000, 64,000, and 128,000 populations sizes. As the graph shows, compared to DEAP-Spark, both PDMS and PDMD appear to offer a better speed-accuracy tradeoff. It is clear to see that PDMD has a poor performance with population less than 8000. Nevertheless, PDMD provides good results starting from a population of 16,000.

In conclusion, the PDMS and PDMD models offers a good tradeoff between accuracy and execution time. The results indicate that the population size helped both models to enhance their accuracy.

no. of instances	11 M
no. of attributes	28 real-valued
no. of classes	2
Class distribution	53% - 47%

Table 8: Higgs dataset composition

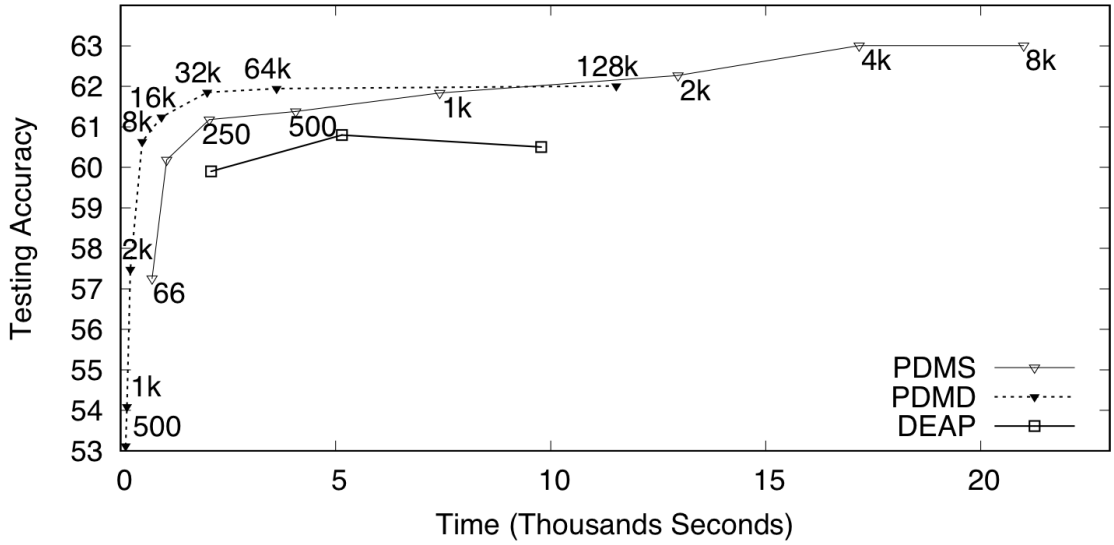


Figure 19: Comparison between DEAP-SPARK and the Implemented PDMS and PDMD Models at different population sizes

4.5 Summary

This chapter describes our first contribution. First, we presented two partitioned data models for parallel GA, PDMD and PDMS, to address big data classification problems. We used the two proposed models to reimplement BioHEL, a popular large-scale single-node GA classifier, using the Spark distributed data processing platform under the assumption of data partitioning. Then the two models scalability

were tested using different datasets with different cluster sizes. Our results have shown that the training time reduces as the parallelisation level increase. The PDMS scales up linearly to 64 nodes, while overhead negatively affects the training speed when the cluster size increases to 96 nodes. The PDMD model is substantially faster than PDMS while maintaining relatively good accuracy. Then the chapter showed the influence of adopting different migration frequencies and population sizes in the PDMD model. The results found that both changes increase the accuracy of the PDMD model with a linear increase in time. Furthermore, using a sufficient population size helped PDMD score higher accuracy than the one accomplished by PDMS while offering a better accuracy-time tradeoff. Finally, a comparison with a GP implementation over Spark shows that both proposed models offer a good accuracy-time tradeoff. While examining both models against the Higgs Bosons dataset, we noticed that increasing the population size not only enhanced the PDMD accuracy but it also boosted the PDMS one. The next chapter will describe further work focusing on how to boost the scalability of both models.

Chapter 5

Automatic Parameter Control

As we showed in the previous chapter, the accuracy-runtime trade-off for both PDMS and PDMD is influenced by the population size. For the PDMD model, this trade-off also depends on the number of partitions; thus, keeping the population fixed limits the scalability of PDMD. This is an example of a parameter that plays a crucial role in expressing the accuracy-time trade-off. Another such parameter is the number of iterations. Changing the number of iterations from 50 to 30 resulted in a change of training time—from 26k (sec) to 13k (sec)—while maintaining the accuracy level for PDMS HEPMASS using 96 cores. Furthermore, as IRL learns rules and consequently filters instances from the training dataset, the fitness landscape changes over time, requiring new parameter settings (rather than using fixed ones) for optimal algorithm performance. Therefore, using parameter control results in better performance, accuracy and time, as the search strategy evolves and adapts. Parameter control removes the need for conducting time-consuming and laborious explorations of the large space of possible parameter value combinations whenever the algorithm is applied to new problem instances. In this Chapter, we shall discuss these

parameters and possible approaches to automating their selection, before proposing AUTO+PDMS and AUTO+PDMD models that integrate some of these techniques, and finally, evaluating their performance in comparison to the baseline.

The chapter is structured as follows. Section 5.1 presents existing research efforts aimed at reducing the number of key GA parameters, the number of iterations and the population size more specifically. Section 5.2 and 5.3 explain the automated version of the PDMS and the PDMD models respectively. Section 2 will report the results of the experiments and will provide a comparison with the original models. Section 5.5 discusses the results presented, and finally, Section 5.6 summarizes the contribution presented in this chapter and the results obtained.

5.1 Related Work: GA Parameters Setting

GAs are flexible and rather robust optimisation strategies, with a high likelihood of obtaining good results for many combinations of parameters. On the other hand, for complex problems or when it is crucial to optimise performance, choosing a suitable parameter is essential. One potential cause of poor performance of a GA is the incorrect calibration of the GA parameters. Parameters such as population size, probability of crossover, and probability of mutation must be calibrated to each specific problem that is tackled. The absolute values used for these parameters, together with their relative values, will determine how a GA finds new solutions and, ultimately, the quality of the final solution found.

The values chosen for the GA parameters will produce a search behavior between two possible extremes: exploitation, where the current best solution is used as a basis for finding better solutions; and exploration, where solutions are combined to

explore the entire search space Herrera, Lozano and Verdegay (1998).

Suitable GA parameters are typically found using a trial and error approach, as in most cases, the characteristics of the problem are largely unknown. However, this approach requires a great deal of time and computational effort. Consequently, a number of methods have been suggested in an attempt to provide an insight into how to set these values. These methods can be divided into three classes: empirical studies; parameter adaptation; and theoretical modelling. The simplest method for determining beneficial relationships among all aspects of a GA is to perform empirical analyses on various test functions. However, there is the potential that the results will be specific to the cases that have been considered in the analyses, and it would be difficult to get precise recommendations from these kinds of studies. Based on theoretical and practical approaches, several authors have proposed methods of adaptively controlling one or more of the operators (i.e. crossover probability or mutation). Instead of using fixed values for these parameters, GAs utilize the knowledge gained about the individuals' fitness values in each generation and adaptively adjust the parameter value in order to maintain the population diversity.

A serious concern with the theoretical studies is that they are derived from rather simplified models of both the problem and the behaviour of the GA; this requires knowledge of aspects of the problem that are not generally known. Also, most GA modelling has been conducted using binary coding, as it is much simpler to perform the analysis when each bit in the solution string can only be in one state—1 or 0.
For the past decade, automated methods for configuring state-of-the-art algorithms have been broadly established as an effective alternative to this manual approach. Prominent examples of general-purpose automated algorithm configurators include irace (López-Ibáñez et al. 2016), ParamILS (Hutter et al. 2009), and SMAC (Hutter, Hoos and Leyton-Brown 2011).

The irace package is a software package that implements several automatic configuration procedures (López-Ibáñez et al. 2016). In particular, it proposes iterated racing procedures, which have been used successfully to configure various state-of-the-art algorithms automatically. The primary goal of irace is to automatize the arduous task of configuring the parameters of an optimization algorithm. However, it may also be used for determining good settings in other computational systems such as robotics and traffic light controllers. The iterated racing algorithms currently implemented in irace have some limitations. The most notable one is that they were mainly designed for scenarios where reducing computation time is not the primary objective. Moreover, the default parameters of irace assume that a minimum number of iterations can be performed and a minimum number of candidate configurations can be sampled. If the tuning budget is too small, the resulting configuration might not be better than random ones. Hutter et al. (2009) introduces ParamILS, a Reinforcement Learning (RL) approach that tunes RL agents across environments that fall into the algorithm configuration method class, which seeks to find better parameters to improve general algorithm performance. A method designed to solve "black box" problems with high computational cost is the sequential configuration of algorithms based on search space (Hutter, Hoos and Leyton-Brown 2011). The algorithm configuration procedure consists in adapting the existing general algorithm to the given problem, allowing the replacement of the expert's attempts by automatic determination of algorithm parameters.

Among the main challenges of the automated algorithm configuration is the time required to evaluate a configuration's performance and the configuration space's size, especially when dealing with a set of challenging training instances. The mechanism of choosing random configurations can easily time out and thus provide little helpful information. Although irace leverages racing methods based on a statistical test to

discard configurations that are unlikely to perform well, yet it leads to insufficient information about the configurations.

In the following two sections, we shall introduce a strategy for managing and automating the number of iterations and population size of BioHEL to enhance the performance of PDMD and PDMS.

5.1.1 Number of iterations

Our objective is to reduce the number of parameters that need to be defined in the GA algorithm, while still being able to sustain good performance. From our experiments conducted to date, we have noticed that most often, half—or less—of the default predefined number of iterations were required to reach the achieved degree of accuracy; however, in a few cases the predefined fixed number of iterations was not enough to find the optimal solution. This highlights that an automated approach to control the number of iterations would be useful and could lead to a significant increase in efficiency.

Still, most of the GA applications have no automatic termination criteria. Actually, in general, EAs cannot decide when or where they can terminate and a user should usually pre-specify a maximum number of generations or function evaluations as termination criteria, as previously mentioned in Section 2.4. The most common used termination criteria that have been used in experiments (as recorded in the recent EA literature) are: (1) a maximum number of generations or function evaluations, which is a simple approach that does not guarantee optimal performance. Moreover, this is usually considered problem-dependent, and a user should have prior information about the test problem; (2) a fitness value approaching the known

global minima, which is not usually known in real problems; 3) a maximum number of consecutive generations without improvement. To give an example in (Leung and Wang 2001), a value of 50 was used before stopping, while in (Venturini 1993), the user is given the choice when to stop the algorithm. It is worth noting that the test suites used to examine these methods are unconstrained test problems with known minima. Hence, these measures for termination are generally not applicable to many real-world problems. Some of the recent studies on termination criteria for EAs can be found in (Gibbs et al. 2006), and Ghoreishi, Clausen and Jørgensen (2017). In (Gibbs et al. 2006), an empirical study is conducted to detect the number of generations before the individuals in the population converge, using the problem characteristics. The survey conducted by Ghoreishi, Clausen and Jørgensen (2017) studies eight termination criteria and provides a concise categorisation of prominent termination criteria in EA. The author stresses the efficacy of using a combination of direct termination criteria and threshold-based termination criteria in order to guarantee the convergence of EA in a reliable manner. Safe et al. (2004a) highlighted the fact that the recent trends are in the direction of employment of adaptive termination conditions. Either genotypical or phenotypical terminations are used instead of a fixed number of iterations.

In general, recent termination criteria for EAs can be classified as follows:

1. TFit Criterion uses convergence measures of the best fitness function values over generations.

2. TPop Criterion uses convergence measures of the population over generations.

3. TSFB Criterion uses search feedback measures to check the progress of exploration and exploitation processes.

EAs can easily be trapped in local minima if only TFit is applied to termination, especially if the algorithm quickly reaches a deep local minimum. Using TPop only for termination is generally improper, since causing the whole population or a part of it to converge is not cheap. However, this is not generally needed, and reaching global minima with one individual is enough. Even though Greenhalgh and Marshall (2000) had already shown that, given a desired confidence level, an upper bound for the number of iterations required to ensure convergence can be estimated, Safe et al. (2004b) showed that, despite being theoretically correct, these criteria are of little practical interest, either because they are too large or for other reasons. Finally, although using TSFB seems efficient, it may face the curse of dimensionality. For example, De Jong (1975) & Manner, Mahfoud and Mahfoud (1992) used performance measuring techniques that maximize the diversity between individuals as a strategy for exploring the entire search space. De Jong (1975) introduced the idea of crowding. This technique was intended for a single population. Its main idea is to remove from the population individuals most similar genetically to the new offspring, thus making a place for the offpsring in that single population. This was later extended to deterministic crowding (Manner, Mahfoud and Mahfoud 1992), where each child created by recombination replaces the genetically closer parent in case it has a better degree of fitness. The crowding concept considers that recombination must create better individuals instantly. Moreover, it does not allow individuals to survive or mate too closely; therefore, they are pushed away from each other, which prevents the algorithm from exploiting a single region. Each crowding and deterministic crowding technique uses its own unique performance measuring criteria, where speed is not a factor in their measurements.

Relying on all of these conclusions, in a modified implementation for both PDMD

and PDMS, we replaced the fixed number of iterations defined in the original Bio-HEL with a convergence condition combining both TFit and TPop. The convergence test determines that if there is no improvement in the best-found solution (fitness) so far, and the average fitness level does not improve, then the rule-learning process is terminated. We added an upper count (consecutive generations without improvement) to estimate convergence, where a fixed counter to 3 is used to ensure that the convergence state is reached. This would be a smarter termination, as it gets feedback from the population about the current state of the search. The algorithm searches for a better solution until no further enhancement is observed in either the best solution or the entire population (average fitness), which indicates that continuing the search process is ineffective. On the other hand, the search continues further if it is progressing well, rather than being forced to stop as the maximum number of iterations is reached.

5.1.2 Population Size

As we have discussed in Chapter 4, larger population sizes are likely to give better solutions for the PDMD model as it scales up, while PDMS may also benefit from a larger population size, as shown by the experiments in Section 4.4. On the other hand, maintaining a large population takes a lot of computational effort. A trade-off has to be made between the quality of the final solution and the amount of computation one is willing to do.

Since the early days of the genetic algorithm (GA) community, many researchers have considered in their studies the effects of the population size on the quality of the obtainable solutions (Goldberg 1989). While there is potential to do things more optimally, designing and drawing valid conclusions naturally becomes more

complex, given that static population sizing is still an open problem. However, there are many factors in population genetics and nature that would support the use of variable population size during the evolution of GAs. Aleti has documented in his literature review (Aleti and Moser 2016) that in 35 papers, population size is being adapted and, as has been observed, population size has a crucial effect on the performance of the algorithm. The genetic algorithm with variable population size (GAVaPS) (Arabas, Michalewicz and Mulawka 1994) replaces the explicit fixed population size parameter with a variable one by introducing two properties: the age and the maximal lifetime. The adaptive population size-based GA (APGA) (Bäck, Eiben and van der Vaart 2000) is a variant of GAVaPS, where a steady-state GA is used, and the lifetime of the best individual remains unchanged when individuals grow older. In (Eiben, Marchiori and Valkó 2004), the author introduces a growing population size when the individuals' fitness is high, in order to improve capacity or in case of longer periods of stagnation. Short stagnation periods decrease the population size.

All previous research has highlighted that the choice of proper population size could substantially increase the efficiency of GA. One of the most comprehensive GA calibration strategies has been proposed in (Harik and Lobo 1999). The approach runs multiple populations with different sizes simultaneously, while establishing a race among those populations.

The different populations are at different stages of evolution, the smaller ones being ahead of the larger ones in terms of generations. For example, a snapshot of the parameter-less GA at a particular point in time could reveal the existence of three populations whose sizes may be 256, 512 and 1024. The population of size 256 could be running its 30th generation, while the population of size 512 could be on generation 6, and the population of size 1024 could still be on generation 1. As time

goes by, the smaller populations are eliminated, and larger populations are created. The creation and deletion of populations is controlled by inspecting the average fitness of the populations and taking decisions based on their relative readings. For example, suppose the population of size 512 has an average fitness greater than that of size 256. This suggests that the smaller population could be stopped because it is improbable that it will produce a fitter individual than the larger population. The larger population is at a much earlier stage of evolution but already contains better individuals than those contained in the smaller one, a clear indication that the smaller population is not large enough. This approach never terminates (that is, it never makes a decision about when it has received the optimal population size, and to run higher populations would therefore just be a waste of computational resources), rendering it inappropriate in an IRL/sequential covering setting. Instead, we are looking for an approach which at some point would be able to terminate, with some confidence that it has obtained a solution with good quality (that is, one obtained from a population with an appropriate size for the problem at hand). Thus, we run the evolution sequentially, doubling the population until doubling it further will not improve the solution quality. This is indeed equivalent to the parallel case. The main motivation for running populations in parallel is to avoid waiting for a long time in the eventuality that a population starts drifting. In our case however, given the termination criteria discussed in the previous section, in case the population starts drifting (and termination is most likely), the run should be terminated due to lack of progress and immediately restarted with a bigger population.

The overall net effect of this strategy is equivalent to a scheme that continuously increases the population size as time progresses.

In addition, we aim to start our GA with a small population. Theoretically, if the starting size is sufficient to kickstart the learning process, the exact value

does not matter, since the population can grow later if necessary. Otherwise, if the population is very small (one or two individuals), the number of individuals may not be sufficient to enable the GA to carry out an efficient starting step in large-scale data problems, and the learning would be terminated at a very early stage. As a point of reference for identifying a good starting value, we can look at the micro-genetic algorithm (micro-GA), which is defined by Goldberg (1989) and was first implemented by Krishnakumar (1990). Krishnakumar used a population size of 5 and reported that his micro-GA obtained faster and better results than the standard GA when tested on two stationary functions and a real-world engineering control problem (a wind-shear controller task). We have chosen the initial size to be ten, which is a reasonable number considering large-scale and complex datasets.

5.2 AUTO+PDMS

The AUTO+PDMS is the result of the application of the above strategy to the original PDMS model by replacing the fixed iteration numbers and the fixed population size with an automated termination and a simple auto-population resizing, as shown in Algorithm 7.

The main BioHEL function is reported for clarity. The first modification in the AUTO+PDMS is in `findRulesPDMS` where the population size is explicitly initialized to 10, and a bestFoundCandidate is set to *null* (line 12). Then, within the main learning loop, and after terminating an inner loop that returns a candidate rule, the best candidate rules are evaluated (lines 28– 29). If the candidate rule has a better fitness level than the previously found one (bestFoundCandidate) using a smaller population, then the candidate replaces the old bestFoundCandidate and the population size is doubled (line 29). Otherwise, the learning process is terminated

Algorithm 7: AUTO+PDMS 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 rule not null then
8              ruleList += bestRule; candidates = ∅;
9          else stop = true;
10     return ruleList;
11 findRules(TrainS):
12     terminate=false; popSize=10; bestFoundCandidate=null;
13     while terminate = false do
14         count=3;
15         pop = TrainS.sample(...).collect();
16         pop.fitness = TrainS.aggregate(zeroes,
             partialFitness(pop),mergeFitness);
17         while count != 0 do
18             offsp = pop.selection(); offsp.crossover(); offsp.mutation() ;
19             offsp.fitness=TrainS.aggregate(zeroes, partialFitness(offsp),
                 mergeFitness);
20             pop.replacement(offsp);
21             candidateIndividual=pop.Best();
22             currentAverage=pop.computeAverageFitness();
23             if candidateIndividual is better than bestIndividual OR
                 currentAverage is better than fitnessAverage then
24                 bestIndividual= takeBest(candidateIndividual,
                     bestIndividual);
25                 fitnessAverage= takeBest(currentAverage, fitnessAverage);
26                 count=3
27             else count−−;
28         if bestIndividual is better than bestFoundCandidate then
29             bestFoundCandidate=bestIndividual; popSize=popSize*2
30         else terminate=true ;
31     return bestFoundCandidate;
32 bestRulePDMS(candidates,TrainS):
33     bestRule = candidates.bestFitness();
34     matchedCl = TrainS.filter(_.matches(bestRule))
         .map(_.class).countByValue();
35     if bestRule.class = majorityClass(matchedCl) then
36         TrainS = TrainS.filter(!_.matches(bestRule));
37         return (bestRule, TrainS);
38     else return (null, TrainS);
```

by setting the "terminate" value to true (line 30) , and thus, the outer loop (line 13) stops.

The second modification in the code is in the GA evolution repeating condition. The GA loop with a fixed number of iterations is replaced with a countdown condition (line 17). This loop repeats the same GA evolution processes, but two additional steps are added to it: (1) computation of the best fitness found and (2) computation of the average fitness of the current generation (lines 21– 22). In case any of these values is better than before, then the better value/s replace the old one/s, and the count value is reset to 3 (lines 24– 26). Otherwise, the count value is decremented by one. Eventually, the loop (line 17) will stop when the count reaches 0 and no further enhancement is found in both best and average fitness values.

For both AUTO+PDMS & AUTO+PDMD, we disable the repetitions in the original BioHEL (cfr. numRepetitions in Algorithm 4, § 4.2), since the modified model is already repeating the learning of a rule with double the population size of the previous run. This gives a clearer indication that the solution has not been trapped in a poor quality local minima. As a consequence `bestRulePDMS`, which is the same in AUTO+PDMS matches the original one defined in the original PDMS-BioHEL (Algorithm 5 in § 4.2) but operates on a single rule (the best rule discovered in `findRulesPDMS`). Moreover, the same `bestRulePDMS` is used again in AUTO+PDMD model.

5.3 AUTO+PDMD

Similarly, we redefined the original PDMD model by replacing the fixed iteration numbers and the fixed population size, as shown in Algorithm 8.

AUTO+PDMD uses the same definition of the `BioHEL` function as AUTO+PDMS.

Algorithm 8: AUTO+PDMD BioHEL

```
1  findRulesPDMD(popSize,TrainS):
2      terminate=false; popSize=10; bestFoundCandidate=null;
3      while terminate = false do
4          pop = ∅; count=3; bestIndividual=null;
5          pop = TrainS.zipPartitions(pop)(
6          (TrainSPart, popPart)⇒
7              if popPart = ∅ then popPart = TrainSPart.sample(popSize);
8              popPart.fitness=doFitness(popPart, TrainSPart);
9              while count != 0 do
10                 offsp = popPart.selection();
11                 offsp.crossover();
12                 offsp.mutation();
13                 offsp.fitness=doFitness(offsp,popPart);
14                 popPart.replacement(offsp);
15                 candidateIndividual=Best(pop);
16                 currentAverage=computeAverageFitness(pop);
17                 if candidateIndividual is better than bestIndividual OR
                    currentAverage is better than fitnessAverage then
18                     bestIndividual= takeBest(candidateIndividual,
                        bestIndividual);
19                     fitnessAverage= takeBest(currentAverage, fitnessAverage);
20                     count=3
21                 else count−−;
22                 return popPart
23         );
24         candidates = pop.mapPartitions(best(1)).collect();
25         candidates.fitness=TrainS.aggregate(zeroes,
            partialFitness(candidates), mergeFitness);
26         bestIndividual = candidates.bestFitness();
27         if bestIndividual is better than bestFoundCandidate then
28             bestFoundCandidate=bestIndividual; popSize=popSize*2
29         else terminate=true ;
30     return bestFoundCandidate;
```

Moreover, in `findRulesPDMD` all nodes will locally use the countdown condition (line 9) and repeat the GA evolution processes, with the addition of two further steps: computation of the best fitness (line 13), and computation of the average fitness of the current generation (line 15). After all local GAs terminate as their local count reaches 0, the zippartition function returns all the partitioned subpopulations (line 5). Then, all local best individuals (rules) are reevaluated globally (line 24), and the overall best individual is compared to the previously found bestFoundCandidate where a decision is taken whether to duplicate the population size or the learning process is to be terminated.

Note that in AUTO+PDMD we apply a no-migration policy, to separate the effect of migration from the automated parameter management strategy.

5.4 Experimental Study

The experimentation design aims to estimate the performance of the presented AUTO+PDMS & AUTO+PDMD models. We investigate how the automatic termination and population resizing influence the accuracy and the execution time. In order to examine the efficiency of automated implementations, we used the HEPMASS, KDD, and the Higgs datasets, with the same cluster and BioHEL configuration used as in Chapter 4. The cluster size is set to 96 executors. We report the average over 20 runs, using 10-fold cross-validation and 95% confidence intervals.

In the following section, we compare the performance of the AUTO+PDMS to the original model results where the same number of nodes were used. Then, the AUTO+PDMD model is compared to the original version as well as to the original PDMS for comparison.

5.4.1 AUTO+PDMS Performance

In this section, we evaluate the performance of AUTO+PDMS and compare it to the original PDMS across the selected datasets. Figure 20 reports the performance (accuracy vs execution time) for the KDD dataset. We ran the PDMS with different population sizes (10, 20, 200, 300, 500 & 2000). The figure shows the resulting accuracy/time performance as we grow the population in the PDMS model. Moreover, the figure reports the average accuracy and execution time for the proposed AUTO+PDMS BioHEL. We can see that AUTO+PDMS BioHEL offers a better trade-off compared to the original PDMS BioHEL. Although AUTO+PDMS does not reach the degree of accuracy achieved by the original PDMS with population size 500 and larger, it obtains a good level of accuracy in a considerably shorter time. This can be returned to the fact that in IRL approach being able to have different population sizes for the different learned rules could lead to a better efficiency where some rules were concluded using smaller population sizes compared to the fixed default one. In observing the size of the population used to learn the different rules in AUTO+PDMS, we have noticed that a population of size 80 was used for learning most of the selected rules in the AUTO+PDMS model while occasionally the evolution terminated with a larger population sizes.

Figure 21 and Table 10 report the performance for the HEPMASS dataset. We ran the PDMS with different population sizes (10, 20, 200, 300, 500 & 2000).

As shown in Figure 21, AUTO+PDMS accuracy/time tradeoff is in line with the original PDMS. AUTO+PDMS is capable of achieving a high degree of accuracy within a reasonable time. Nevertheless, the model was unable to reach the degree of accuracy attained by the original PDMS model with a population size of 2000. The average accuracy of the model is 83.59, which is somehow in line with the accuracy

	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	2350 ±100
	300	99.35 ±0.4	3680 ±100
	500	99.68 ±0.2	5800 ± 80
	2000	99.68 ±0.0	40900 ±680
AUTO+PDMS		99.37 ±0.0	1040 ± 93
PDMD	20	21.56 ±0.0	55 ± 2
	500	93.20 ±1.0	260 ± 15
	960	98.68 ±0.0	590 ± 60
	2000	99.01 ±0.0	630 ± 20
	4000	99.03 ±0.0	650 ± 15
	8000	99.20 ±0.0	800 ± 50
	16000	99.32 ±0.0	1790 ±160
	32000	99.32 ±0.0	4550 ±300
AUTO+PDMD		99.23 ±0.0	540 ± 12

Table 9: Performance metrics of AUTO+PDMS, AUTO+PDMD, and PDMS & PDMD at varying population sizes (KDD dataset).

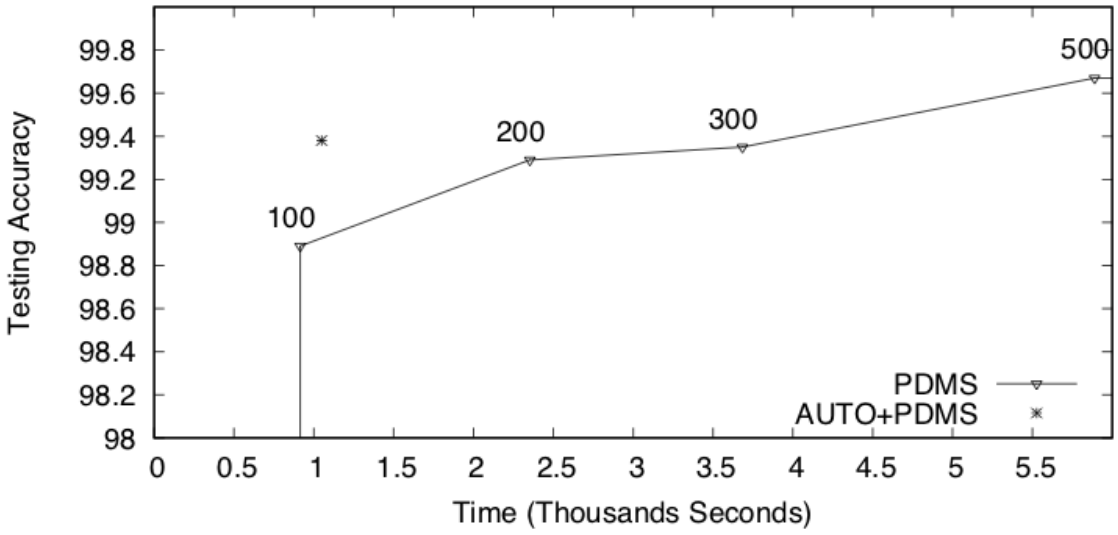


Figure 20: Accuracy/Time tradeoff for AUTO+PDMS and Original PDMS at different population sizes (KDD Dataset)

reached by the original model using the default population size "500" while it saves approximately half of its execution time. (Table 10).

Figure 22 and Table 11 reports the performance for the Higgs dataset. We ran the PDMS with different population sizes (66, 126, 250, 500, 1000, 2000, 4000 & 8000). The accuracy/time tradeoff of PDMS+AUTO is slightly more competitive than the original PDMS, with an execution time between PDMS-250 and PDMS-500 but an accuracy between PDMS-500 and PDMS-1000.

5.4.2 AUTO+PDMD Performance

In this section, we evaluate the performance of the automated PDMD BioHEL while comparing it with both the original PDMS and PDMD models.

Figure 23 reports the performance (accuracy vs time) for the KDD dataset. We

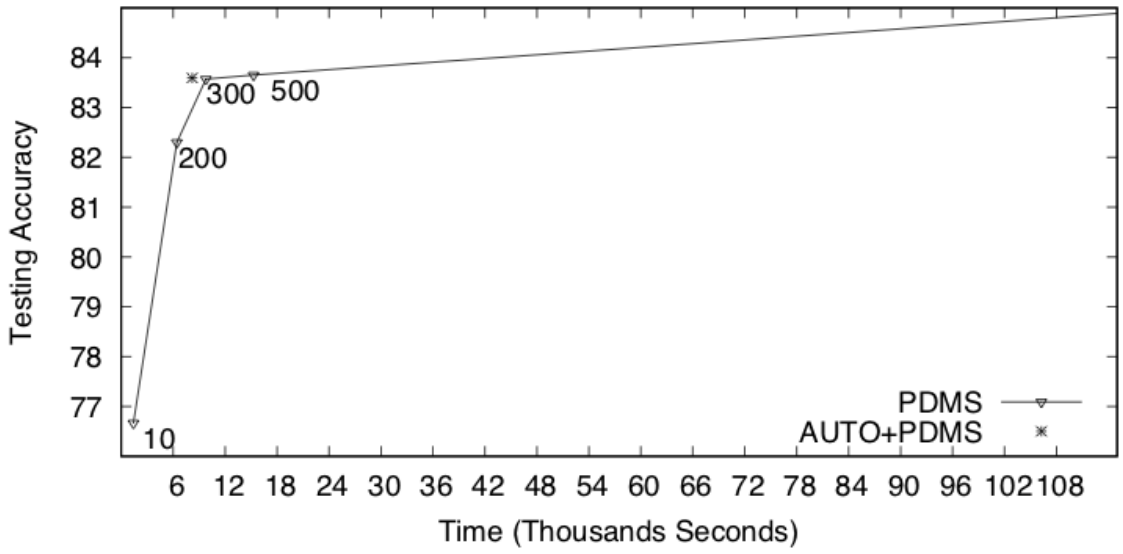


Figure 21: Accuracy/Time tradeoff for AUTO+PDMS and Original PDMS at different population sizes (HEPMASS Dataset)

	popSize	**Accuracy**	**Time**
PDMS	10	76.67 ±1.0	1409 ±200
	200	82.30 ±0.3	6434 ± 50
	300	83.57 ±0.2	9782 ±150
	500	83.65 ±0.2	16500 ±250
	2000	84.90 ±0.1	116047 ±600
AUTO+PDMS		83.59 ±0.1	8200 ±500
PDMD	500	78.67 ±0.6	900 ±200
	960	79.89 ±0.4	1100 ±400
	2000	80.70 ±0.3	1432 ±300
	4000	81.59 ±0.3	1600 ±250
	6000	82.27 ±0.2	1910 ±320
	8000	82.71 ±0.2	2612 ±300
	16000	83.94 ±0.0	5009 ±250
	32000	84.50 ±0.0	10239 ±500
AUTO+PDMD		83.43 ±0.5	6210 ±620

Table 10: 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	1071 ± 300
	250	61.18 ±0.3	2061 ± 158
	500	61.38 ±0.3	4075 ± 450
	1000	61.83 ±0.2	7425 ± 620
	2000	62.27 ±0.3	12961 ±1220
	4000	63.00 ±0.2	17179 ±1600
	8000	63.00 ±0.2	21006 ±2000
AUTO+PDMS		61.59 ±0.2	2900 ± 690
PDMD	500	53.12 ±0.0	116 ± 20
	1000	54.09 ±0.4	148 ± 16
	2000	57.48 ±0.7	225 ± 30
	4000	57.42 ±0.8	299 ± 120
	8000	60.64 ±0.7	503 ± 100
	16000	61.24 ±1.0	951 ± 300
	32000	61.85 ±0.3	2014 ± 100
	64000	61.94 ±0.2	3626 ± 260
	128000	62.01 ±0.1	11523 ± 640
AUTO+PDMD		61.57 ±0.4	3440 ± 400

Table 11: Performance metrics of AUTO+PDMS, AUTO+PDMD, and PDMS & PDMD at varying population sizes (Higgs dataset).

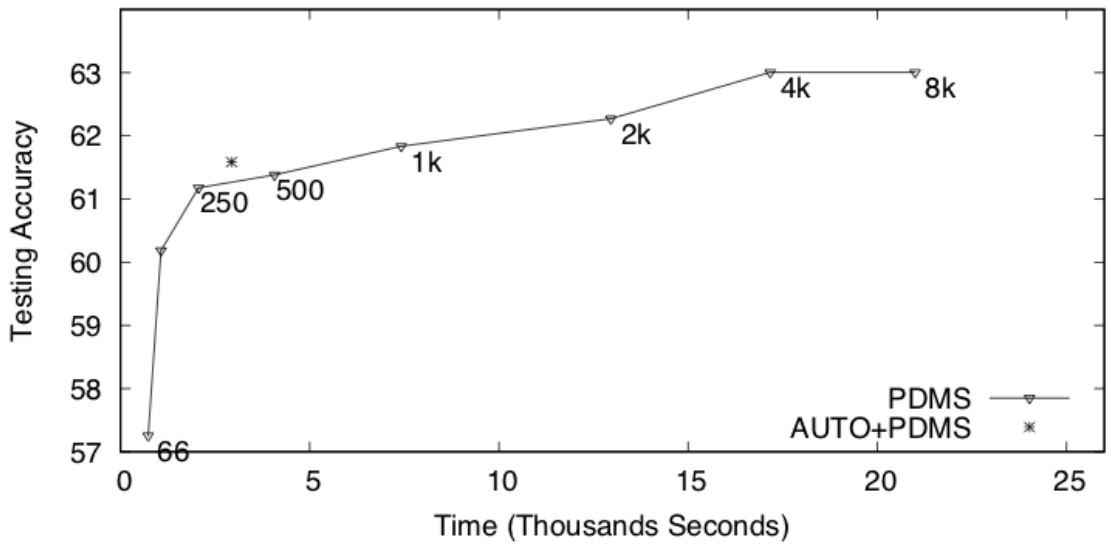


Figure 22: Accuracy/Time tradeoff for AUTO+PDMS and Original PDMS at different population sizes (Higgs Dataset)

ran the PDMD with population sizes 100, 200, & 300. The graph also reports the performance of the original PDMS with population sizes 20, 500, 960, 2000, for comparison. We can clearly see that the AUTO+PDMD offers a better trade-off compared to both original models. AUTO+PDMD managed to reach an accuracy level close to that reached by the original PDMD in 50% of the execution time. Moreover, while investigating our AUTO+PDMD runs on the KDD dataset, it was discovered that, only for a minority of rules rules (on average 25% of the learned rules), the population size of the best selected rule was greater than 10 per island (i.e. 960 in total).

Figure 24 reports the performance (accuracy vs time) for the HEPMASS dataset. For comparison, We report the earlier results of PDMS with different population sizes (10, 200, 300, & 500). Then, we ran the PDMD with population sizes 500,

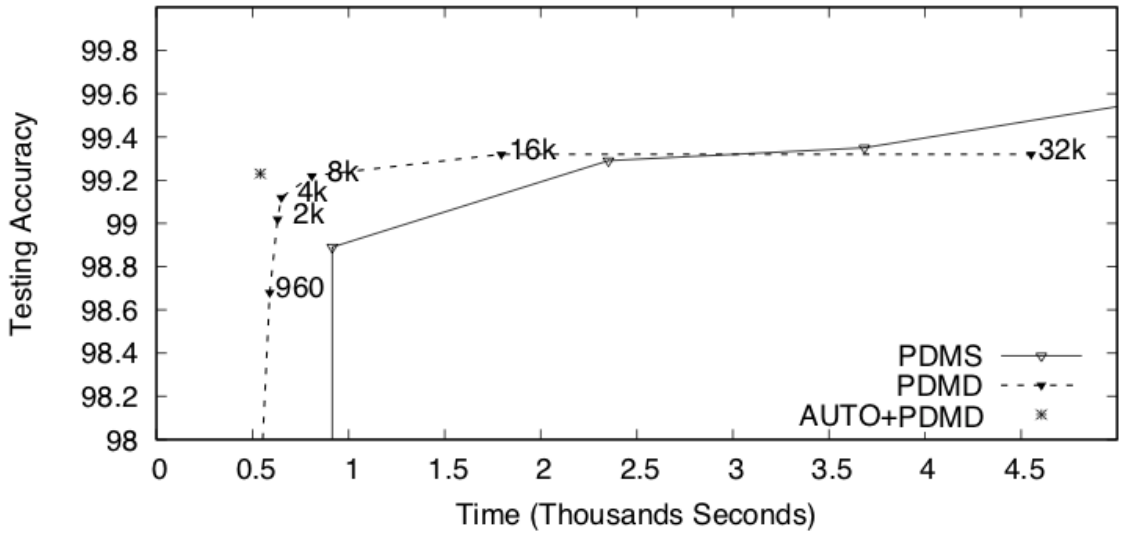


Figure 23: Accuracy/Time tradeoff for AUTO+PDMD and Original PDMS & PDMD at different population sizes (KDD Dataset)

960, 2000, 4000, 8000, 16000 & 32000. The graph shows that the AUTO+PDMD average accuracy is 83.43%; this level of accuracy is between PDMD-8000 and PDMD- 16000 but with a worse execution time between PDMD-16000 and PDMD-32000. Even though PDMD-16000 offers a better tradeoff than AUTO+PDMD, the AUTO+PDMD model returns a close result without knowing the ideal population size.

Figure 25 reports the performance (accuracy vs time) for the Higgs dataset. We ran the PDMD with population sizes 500, 1000, 2000, 4000, 8000, 16000, 32000, 64000 & 128000. The AUTO+PDMD model running time is stable, with its highest accuracy value and learning time being equivalent to those of the original PDMD. The accuracy average across all runs has a wider confidence interval (see Table 11), with a lowest accuracy of 61.17%; this level of accuracy is between PDMD-8000

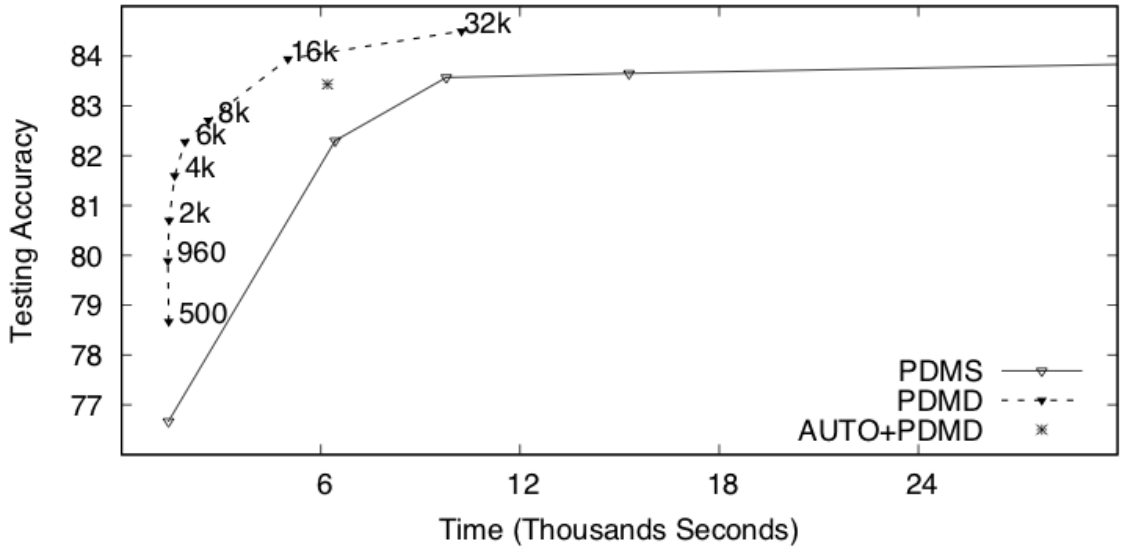


Figure 24: Accuracy/Time tradeoff for AUTO+PDMD and Original PDMS & PDMD at different population sizes (HEPMASS Dataset)

and PDMD-16000 with a higher execution time between PDMD-32000 and PDMD-64000. The AUTO+PDMD returns a good accuracy level with a considerable high execution time but is yet less than the total execution time of all trial population sizes starting from 500 to 32000.

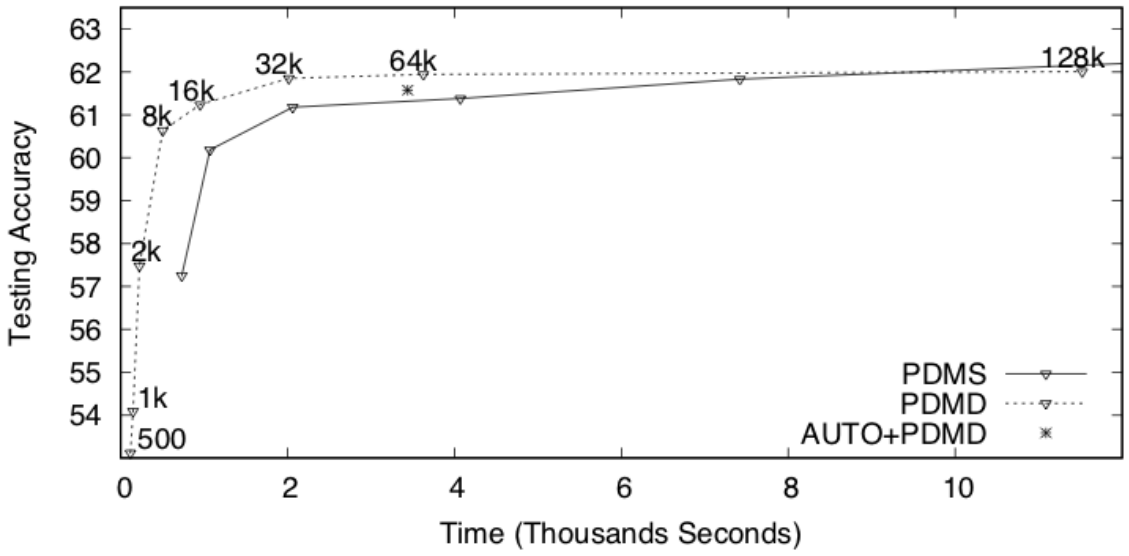


Figure 25: Accuracy/Time tradeoff for AUTO+PDMD and Original PDMS & PDMD at different population sizes (Higgs Dataset)

5.5 Discussion

Some general conclusions can be drawn from our results. Firstly, AUTO+PDMS performed very competently compared to the original PDMS. We can see that the AUTO+PDMS is above the original PDMS curve at different population sizes, which shows that it offers a better trade-off. We can attribute this to the variability of the population size used by the automated BioHEL. When the system is in the process of learning a complicated rule, a larger population will help to explore the search space and locate that solution; however, when the system is in the process of learning an easy rule, a small population could be sufficient for exploiting the search space within a shorter time rather than using a large population. On the contrary, the original PDMS BioHEL algorithm uses the same population for each discovered rule.

AUTO+PDMS could not attain the level of accuracy reached by the original

PDMS, since the convergence condition used is not based on exact convergence (genotype convergence) but rather expresses a tradeoff between cost (execution time) and efficiency (chance to improve the fitness). Overall, the PDMS improves when augmented by the auto-strategy, since a good degree of accuracy can be reached with a smaller population size and thus a shorter time. Regarding the PDMD model, in terms of scaling up the original PDMD to a large cluster size, the model needed a far larger population than the default one to reach a good accuracy level. The performance achieved by AUTO+PDMD proved to be comparable to the optimal performance level of the original; a high accuracy degree was reached without the need to guess the ideal fixed population size. Moreover, AUTO+PDMD benefited from learning good rules with small population sizes when possible, ultimately leading to a shorter learning time and thus a better trade-off than the original PDMD.

In general, we have found that the proposed extensions are significant in that they allow the proposed models to attain high accuracy in a reasonable time. In summary, the automated solution has proved to be a capable tool for solving the problem of setting the population size and the number of iterations; consequently, the automation of both PDMS & PDMD models is a good solution. Nevertheless, the proposed automated models are not ideal when the highest accuracy is the main objective, as the convergence test used may not always lead to the best solution space. Further work could be focused on finding a more sophisticated convergence test to be used for learning termination. Another question to consider is "to which extent the decision to stop at a given population size is reliable".

5.6 Summary

In this chapter, we complemented the proposed models with the automation strategy. The motivation behind automation is proposing a more flexible model which can handle large-scale data while offering high levels of accuracy in a reasonable timeframe, without the need to search manually for the best population size or the number of iterations to be used. The experiments in this chapter compared the AUTO+PDMS & AUTO+PDMD to the original versions, using three datasets. The experiments revealed that both the models, when complemented with the auto-strategy, can scale up reasonably when addressing large-scale data.

Chapter 6

Conclusion

This chapter concludes the contributions presented in this thesis. Moreover, it presents suggestions for future research and areas for application.

In this thesis, we have made great efforts to simplify the use of GA in large-scale problems for practitioners seeking to apply state-of-the-art GA technology to solving real-world problems. The work is motivated by the successful applications of GAs in many areas, and more specifically by the parallel GA power in handling large-scale design optimization problems. GA can be used in knowledge acquisition from mass data; it has been used to great effect by LCS. This research focused on two interesting research areas: firstly, applying data mining to a partitioned data environment, and more precisely, extending a GA-based LCS to learn in a partitioned data environment; secondly, exempting the GA algorithm from some of its user-defined parameters. The main objective is to propose a system to be used in large-scale data classification problems, while maintaining simplicity and efficiency. The proposed algorithms adopted BioHEL, a popular large-scale single-node IRL classifier using the Spark distributed data processing platform under the assumption

of data partitioning.

6.1 Contributions

Two partitioned data models for parallel GA, PDMD and PDMS were implemented to address big data classification problems. The two proposed models reimplemented BioHEL, a popular large-scale, single-node IRL classifier, using the Spark distributed data processing platform under the assumption of data partitioning. The fundamental strategy followed here is to partition the training dataset in the distributed nodes? main memory and to structure the computation and data access according to this partitioning. PDMS runs a unique master-slave parallel GA model, whereby the master runs the complete GA evolutionary processes until the fitness computation step is reached. The master sends a copy of the entire population to all the slaves, and each slave then computes a *partial fitness* for the individuals on the local data partition. Slaves send their results back to the master, which, in turn, aggregates all partial fitness values, thus completing the GA processes. On the other hand, the PDMD model runs isolated parallel GA, each with its own subpopulation. Occasionally, nodes exchange their best individuals with randomly selected nodes, allowing interaction with different subpopulations. Once the GA terminates (that is, completes the preset number of iterations), a solution is selected after an individual has been globally nominated as the best individual (the individual is evaluated over the complete training dataset). Data partitioning merits the parallel implementation of GA; both implementations proved their scalability while preserving high levels of accuracy. Overhead has a minor effect on the learning speed as the cluster size increases. As shown by the results, the PDMD model is substantially faster than

PDMS while maintaining relatively high accuracy. Comparison of the two implemented models with a Parallel GP, presented in (Hmida et al. 2019), found that both models offer a better trade-off of accuracy and execution time. The global conclusions that can be extracted from the work in this thesis are positive.

All of the four kinds of implementations presented are able to improve the basic BioHEL under the assumption of data partitioning. More importantly, the added automated aspect of both AUTO-PDMS & AUTO-PDMD can help the GA user to analyze large-scale data without the need to think about how to terminate the learning process and what the most efficient population size to use could be.

We have investigated the effects of migration on the PDMD model, and of the population size on both PDMS & PDMD models. The investigations revealed that both of these considerations could enhance accuracy degrees while adding a tangible runtime. In order to enhance both PDMS and PDMD implementations, we decided to track the impact of the population size and to find a better alternative for the general default fixed-value policy used by most GA implementations.

Usually, the search space in big data problems is vast, and choosing the right population size to arrive at a correct solution is complex. In this work, we used the dynamic population increase policy proposed in (Harik and Lobo 1999) to overcome the unlucky population size selection. Since our work addressed IRL, it was impossible to adopt the infinite proposed solution in (Harik and Lobo 1999) to sequentially learn multiple rules; therefore, we implemented a greedy termination for the population increase policy. Moreover, as we have dug deeper into the iterative learning process, we have found that the default fixed number of iterations is generally a waste of resources, not even being sufficient in some cases. Therefore, we have replaced the fixed number of iterations with a simple convergence test.

The experimental results have shown that the proposed change gives a good

trade-off between accuracy and time needed to get the classifier concerning the predefined PDMS and PDMD models. These automated models can save the user from conducting trial-and-error experiments to find the ideal number of iterations and population size for the different addressed problems. However, although this study improves the performance of GA in some respects, certain limitations still exist. As discussed earlier, the learning termination criteria (convergence test) impeded the effectiveness of both Dynamic PDMS and PDMD.

6.2 Future Areas

Future work should consist of further experiments with other datasets, as for this work, we were unable to find other large real datasets. Moreover, the work presented in this thesis offers many opportunities for further development and exploration. First, as our contribution is not limited to IRL, an exciting extension would be to examine the modeled 'automated GA' parameters and the 'population evaluation considering data partitioning' with other kinds of GAs used in classification problems (i.e. Pittsburgh and Michigan). Moreover, further work should focus on investigating the applicability of our proposed modeling beyond classification tasks.

While the benefits of using auto-population have been demonstrated, more research needs to be conducted to estimate to what end the auto-population increase scheme was utilized. For example, the role of selection in GAs is a curious one (Harik, Lobo and Sastry 2006), and therefore, the selection method and the selection pressure needs could be other parameters to vary or automate as the population size changes. We believe that this would allow further control of the ratio between exploration and exploitation. Furthermore, as highlighted by Črepinšek, Liu and Mernik (2013), it would be helpful to use a direct measure for exploration and exploitation in order to

boost the explorative and exploitative factors of EA.
Even though AUTO-PDMD yields promising results, inserting migration into its process may boost the level of accuracy, exactly as was the case with the original PDMD. Therefore, we would consider reimplementing AUTO-PDMD with different migration policies.

These extensions to the work of the thesis would be of value to pursue. However, the work presented here already suggests a promising future for using the PDMS & the PDMD models. We hope that this thesis will trigger more interest in considering data partitioning in large-scale data applications. More research will be devoted to enhancing the scalability of GA in such applications. We are convinced that our approach to enhance large-scale data processing is promising in many applications. The automated GA will not only relieve GA users from searching for the best parameters' values for their problems, but it will also return satisfactory results.

Bibliography

Alba, E. and Troya, J. M. (1999). A survey of parallel distributed genetic algorithms. *Complex*, 4(4), pp. 31–52.

Albattah, W. (2016). The role of sampling in big data analysis. In *Proceedings of the International Conference on Big Data and Advanced Wireless Technologies*, New York, NY, USA: Association for Computing Machinery, BDAW '16.

Aleti, A. and Moser, I. (2016). A systematic literature review of adaptive parameter control methods for evolutionary algorithms. *ACM Comput Surv*, 49(3), pp. 56:1–56:35.

AlJame, M., Ahmad, I. and Alfailakawi, M. (2020). Apache spark implementation of whale optimization algorithm. *Cluster Computing*, 23(3), pp. 2021–2034.

Alterkawi, L. and Migliavacca, M. (2019). Parallelism and partitioning in large-scale gas using spark. In *Proceedings of the Genetic and Evolutionary Computation Conference*, New York, NY, USA: Association for Computing Machinery, GECCO '19, pp. 736–744.

Alves de Araujo, D., Lopes, H. and Freitas, A. (1999). A parallel genetic algorithm for rule discovery in large databases. In *IEEE SMC'99 Conference Proceedings.*

1999 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No.99CH37028), vol. 3, pp. 940–945 vol.3.

Arabas, J., Michalewicz, Z. and Mulawka, J. (1994). Gavaps-a genetic algorithm with varying population size. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pp. 73–78 vol.1.

Arnaldo, I., Veeramachaneni, K., Song, A. and OŔeilly, U.-M. (2015). Bring your own learner: A cloud-based, data-parallel commons for machine learning. *IEEE Computational Intelligence Magazine*, 10, pp. 20–32.

Augier, S., Venturing, G. and Kodratoff, Y. (1995). Learning first order logic rules with a genetic algorithm. In *Proceedings of the First International Conference on Knowledge Discovery and Data Mining*, AAAI Press, KDD'95, pp. 21–26.

Bacardit, J. (2004). *Pittsburgh Genetics-Based Machine Learning in the Data Mining era: Representations, generalization, and run-time*. Ph.D. thesis, Ramon Llull University, Barcelona, Spain.

Bacardit, J., Burke, E. K. and Krasnogor, N. (2009). Improving the scalability of rule-based evolutionary learning. *Memetic Computing*, 1(1), pp. 55–67.

Bacardit, J., Goldberg, D. E. and Butz, M. V. (2007). Improving the performance of a pittsburgh learning classifier system using a default rule. In *Proceedings of the 2003-2005 International Conference on Learning Classifier Systems*, IWLCS'03-05, pp. 291–307.

Bacardit, J. and Krasnogor, N. (2006). *BioHEL: Bioinformatics-oriented Hierarchical Evolutionary Learning*.

Bacardit, J. and Krasnogor, N. (2009). A mixed discrete-continuous attribute list representation for large scale classification domains. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, New York, NY, USA: ACM, GECCO '09, pp. 1155–1162.

Bacardit, J. et al. (2007). Automated alphabet reduction method with evolutionary algorithms for protein structure prediction. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, New York, NY, USA: Association for Computing Machinery, GECCO '07, pp. 346–353.

Bäck, T., Eiben, A. E. and van der Vaart, N. A. L. (2000). An emperical study on gas without parameters. In M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. J. Merelo and H.-P. Schwefel, eds., *Parallel Problem Solving from Nature PPSN VI*, Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 315–324.

Baragona, R., Battaglia, F. and Calzini, C. (2001). Genetic algorithms for the identification of additive and innovation outliers in time series. *Computational Statistics & Data Analysis*, 37(1), pp. 1–12.

BRESLOW, L. A. and AHA, D. W. (1997). Simplifying decision trees: A survey. *The Knowledge Engineering Review*, 12(01), pp. 1–40.

Cantú-Paz, E. (1998). A survey of parallel genetic algorithms. *CALCULATEURS PARALLELES*, 10.

Cervantes, J., Li, X. and Yu, W. (2013). Using genetic algorithm to improve classification accuracy on imbalanced data. In *2013 IEEE International Conference on Systems, Man, and Cybernetics*, pp. 2659–2664.

Chao Jin, R. B., Christian Vecchiola (2008). Mrpga: an extension of mapreduce for parallelizing genetic algorithms. In *2008 IEEE Fourth International Conference on eScience*, Indianapolis, IN, USA, pp. 214–221.

Cheraghchi, F., Iranzad, A. and Raahemi, B. (2017). Subspace selection in high-dimensional big data using genetic algorithm in apache spark. In *Proceedings of the Second International Conference on Internet of Things, Data and Cloud Computing*, New York, NY, USA: Association for Computing Machinery, ICC '17.

Clark, G. (1950). The organization of behavior: A neuropsychological theory. d. o. hebb. john wiley and sons, inc., new york, 1949, 335 pages, 19 illustrations, 288 references. $4.00. *Journal of Comparative Neurology*, 93(3), pp. 459–460, `https://onlinelibrary.wiley.com/doi/pdf/10.1002/cne.900930310`.

Dam, H. H., Abbass, H. A. and Lokan, C. J. (2005). Dxcs: an xcs system for distributed data mining. In *GECCO*.

De Jong, K. A. (1975). *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* Ph.D. thesis, USA, aAI7609381.

De Jong, K. A. and Spears, W. M. (1991). Learning concept classification rules using genetic algorithms. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'91, pp. 651–656.

Dean, J. and Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Commun ACM*, 51(1), pp. 107–113.

Derby, O., Veeramachaneni, K. and O'Reilly, U.-M. (2013). Cloud driven design of a distributed genetic programming platform. In A. I. Esparcia-Alcázar, ed.,

Applications of Evolutionary Computation, Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 509–518.

Dhar, V., Chou, D. and Provost, F. (2004). Discovering interesting patterns for investment decision making with glower c -a genetic learner overlaid with entropy reduction. *Data Mining and Knowledge Discovery*, 4, pp. 251–280.

Di Geronimo, L., Ferrucci, F., Murolo, A. and Sarro, F. (2012). A parallel genetic algorithm based on hadoop mapreduce for the automatic generation of junit test suites. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 785–793.

Du, X., Ni, Y., Yao, Z., Xiao, R. and Xie, D. (2013). High performance parallel evolutionary algorithm model based on mapreduce framework. *IJCAT*, 46, pp. 290–295.

Eiben, A. E., Marchiori, E. and Valkó, V. A. (2004). Evolutionary algorithms with on-the-fly population size adjustment. In 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., *Parallel Problem Solving from Nature - PPSN VIII*, Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 41–50.

Freitas, A. A. (2002). *Data mining and knowledge discovery with evolutionary algorithms*. Springer.

Ghoreishi, S. N., Clausen, A. and Jørgensen, B. N. (2017). Termination criteria in evolutionary algorithms: A survey. In *IJCCI*.

Gibbs, M., Maier, H., Dandy, G. and Nixon, J. (2006). Minimum number of generations required for convergence of genetic algorithms. In *2006 IEEE International Conference on Evolutionary Computation*, pp. 565–572.

Giordana, A., Anglano, C., Giordana, A., Bello, G. L. and Saitta, L. (1997). A network genetic algorithm for concept learning. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, Morgan Kaufmann, pp. 436–443.

Goldberg, D. E. (1989). 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.

Grandini, M., Bagli, E. and Visani, G. (2020). Metrics for multi-class classification: an overview.

Greene, D. and Smith, S. F. (2004). Competition-based induction of decision models from examples. *Machine Learning*, 13, pp. 229–257.

Greene, D. P. and Smith, S. (1993). Competition-based induction of decision models from examples. *Machine Learning 13*, 13(2), pp. 229 – 257.

Greenhalgh, D. and Marshall, S. (2000). Convergence criteria for genetic algorithms. *SIAM J Comput*, 30(1), pp. 269–282.

Greffenstette, J. J. (1989). A system for learning control strategies with genetic algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 183–190.

Grosso, P. B. (1985). *Computer Simulations of Genetic Adaptation: Parallel Subcomponent Interaction in a Multilocus Model*. Ph.D. thesis, USA, aAI8520908.

Guo, Y., Rüger, S. M., Sutiwaraphun, J. and Forbes-millott, J. (1997). Meta-learning for parallel data mining. In *In Proceedings of the Seventh Parallel Computing Workshop*, pp. 1–2.

Hans, N., Mahajan, S. and Omkar, S. (2015). Big data clustering using genetic algorithm on hadoop mapreduce. *International Journal of Scientific & Technology Research*, 4, pp. 58–62.

Harik, G. R. and Lobo, F. G. (1999). A parameter-less genetic algorithm. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 1*, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., GECCO'99, pp. 258–265.

Harik, G. R., Lobo, F. G. and Sastry, K. (2006). *Linkage Learning via Probabilistic Modeling in the Extended Compact Genetic Algorithm (ECGA)*, Berlin, Heidelberg: Springer Berlin Heidelberg. pp. 39–61.

He, H. and Garcia, E. A. (2009). Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 21(9), pp. 1263–1284.

Herrera, F., Lozano, M. and Verdegay, J. L. (1998). Tackling real-coded genetic algorithms: Operators and tools for behavioural analysis. *Artificial Intelligence Review*, 12(4), pp. 265–319.

Hmida, H., Hamida, S. B., Borgi, A. and Rukoz, M. (2019). Genetic programming over spark for higgs boson classification. In W. Abramowicz and R. Corchuelo, eds., *Business Information Systems - 22nd International Conference, BIS 2019, Seville, Spain, June 26-28, 2019, Proceedings, Part I*, *Lecture Notes in Business Information Processing*, vol. 353, Springer, pp. 300–312.

Holland, J. H. (1992a). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA: MIT Press.

Holland, J. H. (1992b). Genetic algorithms. *Scientific American*, 267(1), pp. 66–73.

Holsheimer, M. and Siebes, A. (1994). Data mining: the search for knowledge in databases.

Hu, C., Ren, G., Liu, C., Li, M. and Jie, W. (2017). A spark-based genetic algorithm for sensor placement in large scale drinking water distribution systems. *Cluster Computing*, 20(2), pp. 1089–1099.

Huang, D.-W. and Lin, J. (2010). Scaling populations of a genetic algorithm for job shop scheduling problems using mapreduce. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pp. 780–785.

Hutter, F., Hoos, H. H. and Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, Springer, pp. 507–523.

Hutter, F., Hoos, H. H., Leyton-Brown, K. and Stützle, T. (2009). Paramils: An automatic algorithm configuration framework. *J Artif Int Res*, 36(1), p. 267?306.

Janikow, C. Z. (1993). A knowledge-intensive genetic algorithm for supervised learning. *Machine Learning*, 13(2), pp. 189–228.

Kazimipour, B., Li, X. and Qin, A. K. (2014). A review of population initialization techniques for evolutionary algorithms. In *2014 IEEE Congress on Evolutionary Computation (CEC)*, pp. 2585–2592.

Keane, A. (2008). Canonical representation in genetic programming. In K. Bearpark, ed., *Adaptive Computing in Design and Manufacture 2008 (28/04/08 - 30/04/08)*.

Kečo, D. and Subasi, A. (2012). Parallelization of genetic algorithms using hadoop map/reduce. *SOUTHEAST EUROPE JOURNAL OF SOFT COMPUTING*, 1, pp. 56–59.

Kondekar, R. et al. (2012). A mapreduce based hybrid genetic algorithm using island approach for solving time dependent vehicle routing problem. In *2012 International Conference on Computer Information Science (ICCIS)*, vol. 1, pp. 263–269.

Koumousis, V. and Katsaras, C. (2006). A saw-tooth genetic algorithm combining the effects of variable population size and reinitialization to enhance performance. *IEEE Transactions on Evolutionary Computation*, 10(1), pp. 19–28.

Krishnakumar, K. (1990). Micro-Genetic Algorithms For Stationary And Non-Stationary Function Optimization. In G. Rodriguez, ed., *Intelligent Control and Adaptive Systems*, vol. 1196, International Society for Optics and Photonics, SPIE, pp. 289 – 296.

Lemczyk, M. and Heywood, M. I. (2007). Training binary gp classifiers efficiently: A pareto-coevolutionary approach. In M. Ebner, M. O'Neill, A. Ekárt, L. Vanneschi and A. I. Esparcia-Alcázar, eds., *Genetic Programming*, Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 229–240.

Leung, Y.-W. and Wang, Y. (2001). An orthogonal genetic algorithm with quantization for global numerical optimization. *IEEE Transactions on Evolutionary Computation*, 5(1), pp. 41–53.

Lobo, F. G. and Lima, C. F. (2005). A review of adaptive population sizing schemes in genetic algorithms. New York, NY, USA: Association for Computing Machinery, GECCO '05, pp. 228–234.

Lu, H.-C., Hwang, F. and Huang, Y.-H. (2020). Parallel and distributed architecture of genetic algorithm on apache hadoop and spark. *Applied Soft Computing*, 95, p. 106497.

Luque, G. and Alba, E. (2010). Selection pressure and takeover time of distributed evolutionary algorithms. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, New York, NY, USA: Association for Computing Machinery, GECCO '10, pp. 1083–1088.

López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Birattari, M. and Stützle, T. (2016). The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3, pp. 43–58.

Mahmud, M. S., Huang, J. Z., Salloum, S., Emara, T. Z. and Sadatdiynov, K. (2020). A survey of data partitioning and sampling methods to support big data analysis. *Big Data Mining and Analytics*, 3(2), pp. 85–101.

Mambrini, A. and Sudholt, D. (2014). Design and analysis of adaptive migration intervals in parallel evolutionary algorithms. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, New York, NY, USA: Association for Computing Machinery, GECCO '14, pp. 1047–1054.

Mambrini, A. and Sudholt, D. (2015). Design and Analysis of Schemes for Adapting Migration Intervals in Parallel Evolutionary Algorithms. *Evolutionary Computation*, 23(4), pp. 559–582, `https://direct.mit.edu/evco/article-pdf/23/4/559/1523126/evco_a_00153.pdf`.

Manner, R., Mahfoud, S. and Mahfoud, S. W. (1992). Crowding and preselection revisited. In *Parallel Problem Solving From Nature*, North-Holland, pp. 27–36.

Maulik, U. and Bandyopadhyay, S. (2000). Genetic algorithm-based clustering technique. *Pattern Recognition*, 33(9), pp. 1455–1465.

Michalski, R. S. (1983). *A Theory and Methodology of Inductive Learning*, Berlin, Heidelberg: Springer Berlin Heidelberg. pp. 83–134.

Minaei-Bidgoli, B. and Punch, W. F. (2003). Using genetic algorithms for data mining optimization in an educational web-based system. In 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., *Genetic and Evolutionary Computation — GECCO 2003*, Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 2252–2263.

Mingers, J. (1989). An empirical comparison of pruning methods for decision tree induction. *Machine Learning*, 4(2), pp. 227–243.

Mitchell, M. (1998). *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT Press.

Mitchell, T. M. (1997). *Machine Learning*. USA: McGraw-Hill, Inc., 1st edn.

Murty, M. N., Rashmin, B. and Bhattacharyya, C. (2008). *Clustering Based on Genetic Algorithms*, Berlin, Heidelberg: Springer Berlin Heidelberg. pp. 137–159.

Niedermayer, I. S. P. D. (2008). An introduction to bayesian networks and their contemporary applications. In *Innovations in Bayesian Networks*.

O'Reilly, U.-M., Wagy, M. and Hodjat, B. (2013). *EC-Star: A Massive-Scale, Hub and Spoke, Distributed Genetic Programming System*, New York, NY: Springer New York. pp. 73–85.

Paduraru, C., Melemciuc, M.-C. and Stefanescu, A. (2017). A distributed implementation using apache spark of a genetic algorithm applied to test data generation. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, New York, NY, USA: Association for Computing Machinery, GECCO '17, pp. 1857–1863.

Paterlini, S. and Minerva, T. (2010). 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*, Stevens Point, Wisconsin, USA: World Scientific and Engineering Academy and Society (WSEAS), NN'10/EC'10/FS'10, pp. 19–27.

Pelikan, M., Goldberg, D. E. and Cantu-Paz, E. (2000). Bayesian optimization algorithm, population sizing, and time to convergence. In *Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation*, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., GECCO'00, pp. 275–282.

Piszcz, A. and Soule, T. (2006). Genetic programming: Optimal population sizes

for varying complexity problems. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, New York, NY, USA: Association for Computing Machinery, GECCO '06, pp. 953–954.

Qi, R., Wang, Z.-J. and Li, S.-Y. (2016). A parallel genetic algorithm based on spark for pairwise test suite generation. *Journal of Computer Science and Technology*, 31, pp. 417–427.

QUINLAN, J. R. (1993). Chapter 5 - from trees to rules. In J. R. QUINLAN, ed., *C4.5*, San Francisco (CA): Morgan Kaufmann, pp. 45–56.

Safe, M., Carballido, J., Ponzoni, I. and Brignole, N. (2004a). On stopping criteria for genetic algorithms. In A. L. C. Bazzan and S. Labidi, eds., *Advances in Artificial Intelligence – SBIA 2004*, Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 405–413.

Safe, M., Carballido, J., Ponzoni, I. and Brignole, N. (2004b). On stopping criteria for genetic algorithms. In A. L. C. Bazzan and S. Labidi, eds., *Advances in Artificial Intelligence – SBIA 2004*, Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 405–413.

Saini, N. (2017). Review of selection methods in genetic algorithms. *International Journal of Engineering and Computer Science*, 6(12), pp. 22261–22263.

Schmitt, L. M. (2001). Theory of genetic algorithms. *Theoretical Computer Science*, 259(1), pp. 1–61.

Sherry, D., Veeramachaneni, K., McDermott, J. and O'Reilly, U.-M. (2012). Flex-gp: Genetic programming on the cloud. In *Applications of Evolutionary Computation*, Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 477–486.

Tung, A. K. H. (2009). *Rule-based Classification*, Boston, MA: Springer US. pp. 2459–2462.

Varadarajan, S. and Whitley, D. (2021). *A Parallel Ensemble Genetic Algorithm for the Traveling Salesman Problem*, New York, NY, USA: Association for Computing Machinery. pp. 636–643.

Črepinšek, M., Liu, S.-H. and Mernik, M. (2013). Exploration and exploitation in evolutionary algorithms: A survey. *ACM Comput Surv*, 45(3).

Venturini, G. (1993). Sia: A supervised inductive algorithm with genetic search for learning attributes based concepts. In P. B. Brazdil, ed., *Machine Learning: ECML-93*, Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 280–296.

Verma, A., Llorà, X., Goldberg, D. E. and Campbell, R. H. (2009). Scaling genetic algorithms using mapreduce. In *Proceedings of the 2009 Ninth International Conference on Intelligent Systems Design and Applications*, Washington, DC, USA: IEEE Computer Society, ISDA '09, pp. 13–18.

Vrajitoru, D. (2000). *Large Population or Many Generations for Genetic Algorithms? Implications in Information Retrieval*, Heidelberg: Physica-Verlag HD. pp. 199–222.

Wang, L., Maciejewski, A. A., Siegel, H. J., Roychowdhury, V. P. and Eldridge, B. D. (2005). A study of five parallel approaches to a genetic algorithm for the traveling salesman problem. *Intelligent Automation & Soft Computing*, 11(4), pp. 217–234, https://doi.org/10.1080/10798587.2005.10642906.

WHITLEY, D. and STARKWEATHER, T. (1990). Genitor ii: a distributed genetic

algorithm. *Journal of Experimental & Theoretical Artificial Intelligence*, 2(3), pp. 189–214, https://doi.org/10.1080/09528139008953723.

WILSON, S. W. (1987). Quasi-darwinian learning in a classifier system. In P. Langley, ed., *Proceedings of the Fourth International Workshop on MACHINE LEARNING*, Morgan Kaufmann, pp. 59–65.

Witten, I. H., Frank, E. and Hall, M. A. (2011). Chapter 1 - what's it all about? In I. H. Witten, E. Frank and M. A. Hall, eds., *Data Mining: Practical Machine Learning Tools and Techniques, Third Edition*, Boston: Morgan Kaufmann, The Morgan Kaufmann Series in Data Management Systems, pp. 3–38.

Wu, X., Zhu, X., Wu, G.-Q. and Ding, W. (2014). Data mining with big data. *IEEE Transactions on Knowledge & Data Engineering*, 26(01), pp. 97–107.

Yu, Y., Liu, Y., Xu, B. and He, X. (2014). Experimental comparisons of instances set reduction algorithms.

Zaharia, M. et al. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, San Jose, CA: USENIX, pp. 15–28.