2014 IEEE Congress on Evolutionary Computation (CEC)
July 6-11, 2014, Beijing, China

# A Combined MapReduce-Windowing Two-Level Parallel Scheme for Evolutionary Prototype Generation

Isaac Triguero, Daniel Peralta, Jaume Bacardit, Salvador García and Francisco Herrera

***Abstract*— Evolutionary prototype generation techniques have demonstrated their usefulness to improve the capabilities of the nearest neighbor classifier. They act as data reduction algorithms by generating representative points of a given problem. Their main purposes are to speed up the classification process and to reduce the storage requirements and sensitivity to noise of the nearest neighbor rule. Nowadays, with the increment of available data, the use of this kind of reduction techniques becomes more important. However, their applicability can be limited to problems with no more than tens of thousands of instances. In order to address this limitation, in this work we develop a two-level parallelization scheme for evolutionary prototype generation methods. Firstly, it distributes the functioning of these algorithms in several tasks based on a MapReduce framework. Then, for each one of these tasks (mappers), we accelerate the prototype generation process by using a windowing approach. This model enables evolutionary prototype generation algorithms to be applied over large-scale classification problems without accuracy loss. Our preliminary experiments using a dataset of 1 million instances show that this proposal is an appropriate tool to improve the performance of the nearest neighbor classifier with big data.**

## I. Introduction

EXTRACTING knowledge from large amounts of data is currently a very challenging task. This problem is termed as big data when the quantity of information overwhelms the processing capacities of a system [1]. In a wide variety of fields such as marketing, medicine or industry, their researchers have collected a lot of raw data that could be very valuable if they are properly processed. Analyzing these data is a difficult assignment for most of the standard data mining tools [2]. Nevertheless, with the availability of cloud platforms and emerging technologies [3], these methods can be adapted in order to be applied on big data.

The parallelization of data mining techniques should be performed according to their specific requirements. In this sense, the MapReduce framework [4] provides a simple but robust environment to tackle the processing of large-scale problems over a cluster of computers. The use of this scheme for data mining is highly recommended, instead of other parallelization schemes such as MPI, because it includes a fault-tolerant mechanism (recommendable for time-consuming tasks) and because of its ease of use. Several data mining techniques have been implemented within this paradigm with promising results [5], [6].

Data reduction techniques aim to represent original data in such a way that data mining algorithms can be run faster and more accurately. Most well-known data reduction methods are feature selection and generation, instances selection and generation, and the hybridization of them [7]. As such, they should ease data mining algorithms to tackle big data problems. However, these methods also suffer several drawbacks when the size of the data sets is increased, being unable to provide a resulting set in a reasonable time.

This work is focused on Prototype Generation (PG) techniques [8], which are instance generation methods designed to enhance the performance of the Nearest Neighbor rule (NN) [9]. These techniques generate a reduced set, by selecting or building new prototypes, which better adjust the decision boundaries between classes in NN. PG models have proved to reduce the computational costs and high storage requirements of NN. Among the existing proposals, Evolutionary Prototype Generation (EPG) algorithms have been highlighted as the best performing approaches [8], [10].

Dealing with large-scale data becomes impractical for most of the PG techniques. This issue is more remarkable in EPG approaches where an excessive increment of the individual size can limit their practicality. Two different solutions, based on partitioning the training set, can be used to tackle this problem: stratification [11] and windowing [12]. The former consists of a distributed partitioning that preserves the class distribution. This splits the original data into several parts that are individually addressed. Then, it joins all solutions into a global one. The latter selects a training subset during the learning process in order to evaluate or refine solutions. For evolutionary algorithms, windowing can be applied as an incremental learning with alternating strata [12] in which each iteration the evolutionary process uses a different subset for fitness computation.

In this work, we propose a two-level parallel scheme for EPG that is based on both stratification and windowing techniques. Our main objective is to offer a whole framework to speed up EPG methods without accuracy losses. As first parallelization level, we develop a MapReduce approach that is based on our previous stratification model for PG methods [13], aiming to overcome its main drawbacks (they will be introduced in Section III). As second parallelization component, the windowing procedure will be used

This work was supported by the Research Projects TIN2011-28488, P10-TIC-6858 and P11-TIC-7765.

I. Triguero, D. Peralta and F. Herrera are with the Department of Computer Science and Artificial Intelligence of the University of Granada, CITIC-UGR, Granada, Spain, 18071. E-mails: {triguero, dperalta, herrera}@decsai.ugr.es

S. García is with the Department of Computer Science of the University of Jaén, Jaén, Spain, 23071. E-mail: sglopez@ujaen.es

J. Bacardit is with the School of Computing Science, Newcastle University, NE1 7RU, Newcastle, UK. E-mail: jaume.bacardit@newcastle.ac.uk

978-1-4799-1488-3/14/$31.00 ©2014 IEEE

to accelerate the processing without reducing the available information for an EPG algorithm. Moreover, it does not require more computing elements. We will name this method as "MapReduce and Windowing for Evolutionary Prototype Generation (MRW-EPG)".

In our experiments we will use a recent EPG method, called IPADECS [14], which is based on differential evolution [15]. We will analyze its capabilities in terms of accuracy, runtime, speed up and reduction rate. Several variations of the proposed model will be investigated, varying the number of used mappers and windows.

The paper is organized as follows. Section II provides background information about PG and MapReduce. Section III describes the proposal. Section IV analyzes the empirical results. Finally, Section V summarizes the conclusions.

## II. Background

In this section we briefly describe the topics used in this paper. Section II-A presents the PG problem and its main drawbacks to tackle big data classification. Section II-B introduces the MapReduce paradigm.

### A. Prototype generation and big data

A formal notation for PG is the following: Let $TR$ be a training data set and $TS$ a test set, they are formed by a number **n** and **t** of samples, respectively. Each sample $\mathbf{x}_p$ is a tuple $(\mathbf{x}_{p1}, \mathbf{x}_{p2}, ..., \mathbf{x}_{pD}, \omega)$, where, $\mathbf{x}_{pf}$ is the value of the $f$-th feature of the $p$-th sample. This sample belongs to a class $\omega$, given by $\mathbf{x}_{p\omega}$, and a $D$-dimensional space. For the $TR$ set the class $\omega$ is known, while it is unknown for $TS$.

The purpose of PG is to provide a reduced set $RS$ which consists of **rs** (**rs** $<$ **n**) prototypes, which are either selected or generated from the examples of $TR$. The prototypes of $RS$ should be calculated to efficiently represent the distributions of the classes and to discern well when they are used to classify the training objects. The size of $RS$ should be sufficiently reduced to deal with the storage and evaluation time problems of the NN classifier.

In the literature, best performing models follow an evolutionary positioning adjustment of the prototypes [8]. These techniques aim to correct the position of a subset of prototypes $RS$ by using an optimization procedure according to $TR$. Among them, the IPADECS algorithm [10], [14] highlights as a very competitive PG algorithm[1].

Despite their bright performance with medium size problems, they lack of scalability with big data sets (from tens of thousands of instances [16]). Their main problems are:

- **Runtime**: The complexity of PG models is $O((n \cdot D)^2)$ or higher, where $n$ is the number of instances and $D$ the number of features. Although these techniques are only applied once on a $TR$, if this process takes too long, its application could become inoperable for real applications.
- **Memory consumption**: PG methods store in the main memory many partial calculations, intermediate solutions, and/or also the entire $TR$. When $TR$ is too big, it could easily exceed the available RAM memory.

These problems are usually more pronounced in EPG algorithms. As we will see in further sections, these weaknesses motivate the use of distributed partitioning procedures in conjunction with a windowing scheme.

### B. MapReduce

MapReduce is a paradigm of parallel programming [17], [4] designed to process large data sets over a computer cluster regardless the underlying hardware.

This model works in two different steps: the map phase and the reduce phase. Each one has key-value ($< k, v >$) pairs as input and output. The map phase takes each $< k, v >$ pair and generates a set of intermediate $< k, v >$ pairs. Then, MapReduce merges all the values associated with the same intermediate key as a list (shuffle phase). The reduce phase takes that list as input for producing the final values. Figure 1 depicts a flowchart of the MapReduce framework. In a MapReduce program, all map and reduce operations run in parallel. First of all, all map functions are independently run. Meanwhile, reduce operations wait until their respective maps are finished. Then, they process different keys concurrently and independently. Note that inputs and outputs of a MapReduce job are stored in an associated distributed file system that is accessible from any computer of the used cluster.

In this work we will focus on the Hadoop implementation [18] of the MapReduce framework because of its performance, its open source nature and its distributed file system (Hadoop Distributed File System, HDFS). A Hadoop cluster is formed by a master-slave architecture, where one master node manages an arbitrary number of slave nodes. The HDFS replicates file data in multiple storage nodes that can concurrently access to the data. As such cluster, a certain percentage of these slave nodes may be out of order temporarily. For this reason, Hadoop provides a fault-tolerant mechanism, so that, when one node fails, Hadoop restarts automatically the task on another node. Thus, Hadoop offers a propitious environment to successfully speed up data mining techniques.

## III. MRW-EPG: MapReduce and windowing for evolutionary prototype generation

In this section we present the proposed MapReduce and windowing approach for EPG. Firstly, we argue the motivation that justifies our proposal (Section III-A). Then, we detail the proposed model in depth (Section III-B).

### A. Motivation

PG methods reduce their performance when tackling large-scale data sets. The distribution and parallelization of workload in different sub-processes may ease the issues previously mentioned: runtime and memory consumption.

In a previous work [13], we proposed a distributed partitioning technique to alleviate these problems for any PG

[1]More information and results about PG can be found at the SCI2S thematic public website on *Prototype Reduction in Nearest Neighbor Classification: Prototype Selection and Prototype Generation* `http://sci2s.ugr.es/pr/`

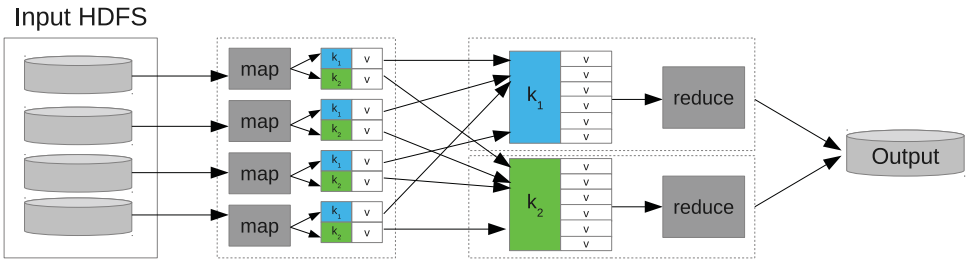


Fig. 1: The MapReduce framework

method. This model splits the training set, called $TR$, into disjoint $d$ subsets $(TR_1, TR_2, ..., TR_d)$ with equal class distribution and size. Then, a PG model is applied to each $TR_j$, obtaining a resulting reduced set $RS_j$. Finally, all $RS_j$ $(1 \leq j \leq d)$ are merged into a final reduced set $RS$, which is used to classify the instances of $TS$ with the NN rule.

This partitioning process shows to perform well in medium size domains. However, it has some limitations:

- Maintaining the proportion of examples per class of $TR$ within each $TR_j$ cannot be accomplished when the size of the data set does not fit in the main memory. Hence, this strategy cannot scale to data sets of arbitrary size.
- Joining all the partial reduced sets $RS_j$ into a final $RS$ may lead to the introduction of noisy and/or redundant examples. Each resulting $RS_j$ tries to represent, with the minimum number of instances, a proportion of the entire $TR$. Thus, when the size of $TR$ tends to be very high, the instances contained in some $TR_j$ subsets may be located very near in the $D$-dimensional space. Therefore, the final $RS$ may enclose unnecessary instances to represent the training data. The likelihood of this issue increases with the number of partitions.

Apart from these general weaknesses for any kind of PG algorithm, it is important to note that existing EPG methods perform their fitness evaluation in terms of training set classification [8]. It means that this promising PG family of methods may spend more time to tackle a problem of the same size in comparison with other PG models.

These drawbacks motivate the design of a combined MapReduce and windowing system for EPG. The MapReduce strategy will allow us to avoid the general limitations of the partitioning process for any PG method. In addition, we will apply a windowing scheme to accelerate EPG models.

### B. A two-level parallelization model for EPG

The proposed MRW-EPG model is composed by two parallelism levels. In this section we detail each parallelization level of MRW-EPG by following a bottom-up description. First, Section III-B.1 explains the use of windowing for EPG and the implementation followed in this work. Then, Section III-B.2 describes the distributed partitioning process, based on MapReduce, that uses the windowing scheme for EPG as part of its processing. At the end of this section, Figure 5 depicts a flowchart of the proposed model.

**Require:** $TR_j$, $nw$
1: $W_1, W_2, ..., W_{nw}$ = Divide $TR_j$ into $nw$ stratified sub sets.
2: Initialize Population
3: iter=0
4: **while** iter $<$ iterations **do**
5: Modification of the prototypes according to operators
6: fitness= NN-leave-one-out($RS_j$, $W_{iter\%nw+1}$)
7: Selection operator
8: iter=iter+1
9: **end while**
10: **return** $RS_j$

Fig. 2: Structure of a EPG algorithm based on windowing

*1) Parallelizing EPG with windowing:* The windowing approach was originally proposed in [12] to speed up genetic-based machine learning algorithms [3]. The underlying idea of this technique is the use of a subset of the whole training data to perform the fitness evaluations. Thus, as the distributed partitioning methods explained before, the windowing strategy divides the input training data into $nw$ disjoint windows $(W_1, W_2, ..., W_{nw})$ with equal class distribution and size. At each iteration of a genetic algorithm, it utilizes a different subset to compute the fitness function, using a round-robin policy. It is noteworthy that in contradistinction to the distributed partitioning method, within this scheme, the algorithm disposes of the whole information $TR$ although it is accessed in successive iterations.

This technique could be easily extensible to many other evolutionary algorithms developed to solve data mining tasks. In this sense, we consider its application for EPG. To the best of our knowledge, all existing EPG methods calculate their fitness function as follows: The instances in the training set $TR$ are classified with the prototypes of the current $RS_j$ by the NN rule with a leave-one-out validation scheme. The resulting fitness value is measured as the number of successful hits in comparison with the total number of classifications. Therefore, the windowing scheme for EPG merely consists of using different windows $W_j$ at each iteration as the training set that will be classified by the NN rule. Figure 2 shows the pseudo-code of windowing for a general EPG method.

In our experiments we include this idea into the IPADECS algorithm [14]. This inclusion requires minor changes to the structure of the algorithm. We introduce the number of windows as a parameter, create the subsets and modify every fitness computation into the differential evolution algorithm that is used in IPADECS according to Figure 2.

This model itself aims to improve the runtime requirements of EPG algorithms without accuracy losses. However, it does not deal with the memory consumption problem because it is run in a single computing node. Hence, this strategy will be used as a second level parallelization scheme after a previous distribution of the processing in a cluster of computing elements.

*2) Parallelizing with MapReduce:* Splitting the data into several subsets, and processing them separately, fits better with the MapReduce philosophy than with other parallelization schemes because of two reasons: Firstly, each subset is individually processed, so that it does not need any data exchange between nodes to proceed [19]. Secondly, the computational cost of each chunk could be so high that a fault-tolerant mechanism is mandatory. As a MapReduce model, it organizes the processing into two key operations: the map phase corresponds to the splitting procedure and the application of the EPG technique. Concretely, the EPG algorithm is applied with the described windowing scheme, as a second level of parallelization. The reduce stage performs a fusion of prototypes to avoid the introduction of harmful prototypes to the resulting data set.

Suppose a training set $TR$ of a given size, stored in the HDFS as a single file. The first step of MRW-EPG is the splitting of $TR$ into a number of disjoint subsets. Within a Hadoop perspective, the $TR$ file is composed by $h$ HDFS blocks that are accessible from any computer of the cluster independently of its size. Let $m$ the number of map tasks (a user-defined parameter). Each map task $(Map_1, Map_2, ..., Map_m)$ will form an associated $TR_j$, where $1 \leq j \leq m$, with the instances of each chunk in which the training set file is divided. This partitioning process is performed sequentially, so that the $Map_j$ corresponds to the $j$ data chunk of $h/m$ HDFS blocks. So, each map will process approximately the same number of instances.

Under this scheme, if the partitioning procedure is directly applied over $TR$, the class distribution of each subset $TR_j$ could be biased to the original distribution of instances in its corresponding file. As we stated before, no proper stratified partitioning can be carried out if the size of $TR$ does not fit in the main memory. To develop a scheme easily scalable to any number of instances, we previously randomize the entire file. This operation is not time-consuming in comparison with the application of the PG technique and should be applied only once. It does not ensure that every class is represented proportionally to its number of instances in $TR$. However, probabilistically, each chunk should include approximately a number of instances of class $\omega$ according to the probability of belonging to this class in the original $TR$.

When each map has formed its corresponding $TR_j$, an EPG step should be performed using $TR_j$ as the input training data to generate a reduced set $RS_j$.

At this point, we concatenate the windowing scheme for EPG in order to accelerate the generation process. Although the proposed MapReduce approach can distribute the instances of a given data set over a whole cluster, avoiding memory consumption problems, it does not guarantee the execution of EPG algorithms to be reasonably fast. Note that we could accelerate EPG techniques by increasing the number of used mappers (it implies more computing units). However, as we will see in the experimental results, this could deteriorate their performance because the available information may not be enough. As we commented previously, the windowing scheme gradually uses the whole information of the problem during its evolutionary process. Figure 3 summarizes the pseudo-code of the map function.

**Require:** Number of split $j$, Number of windows $nw$
1: Constitute $TR_j$ with the instances of split $j$.
2: $RS_j$=EPG_windowing($TR_j$, $nw$)
3: **return** $RS_j$

Fig. 3: Map function

**Require:** $RS_j$, $typeOfReducer$ {Initially $RS = \emptyset$}
1: $RS = RS \cup RS_j$
2: **if** $typeOfReducer$==Fusion **then**
3: $RS = RS \cup RS_j$
4: $RS$=Fusion($RS$)
5: **end if**
6: **return** $RS$

Fig. 4: Reduce function

As each map finishes its processing the results are forwarded to a single reduce task. The reduce phase consists of a iterative aggregation of all the $RS_j$ as a single set $RS$. Figure 4 shows the pseudo-code of the reduce function. Initially $RS = \emptyset$. We propose two different alternatives:

- *Join*: This option concatenates all the $RS_j$ sets into a final reduce set $RS$. Instruction 2 of Figure 4 indicates how the reduce function progressively joins all the $RS_j$ as the mappers finish their processing. This type of reducer implements with MapReduce the strategy proposed in [13]. This joining process does not guarantee that the resulting $RS$ does not contain irrelevant or even harmful instances, but it is included as a baseline.
- *Fusion*: In this variant we aim to eliminate redundant prototypes. To accomplish this objective we rely on the success of centroid-based methods for prototype generation [8]. These techniques reduce a prototype set by merging similar examples [20]. Since in this step we have to fuse all the $RS_j$ into a single one, these methods can be very useful to generate a final set without redundant or very similar prototypes. The fusion phase will be progressively applied during the creation of $RS$. It means that as the mappers end their execution, the reduce function is run and the next $RS$ is computed as the fused set obtained with its current content and the new $RS_j$. Instructions 4-7 of Figure 4 explain how to apply the fusion phase.
  We will use the ICLP2 method presented in [21]. This model integrates several prototypes by identifying borders and merging those instances that are not located in these borders. It highlights as the best performing model of the centroid-based family in [8].

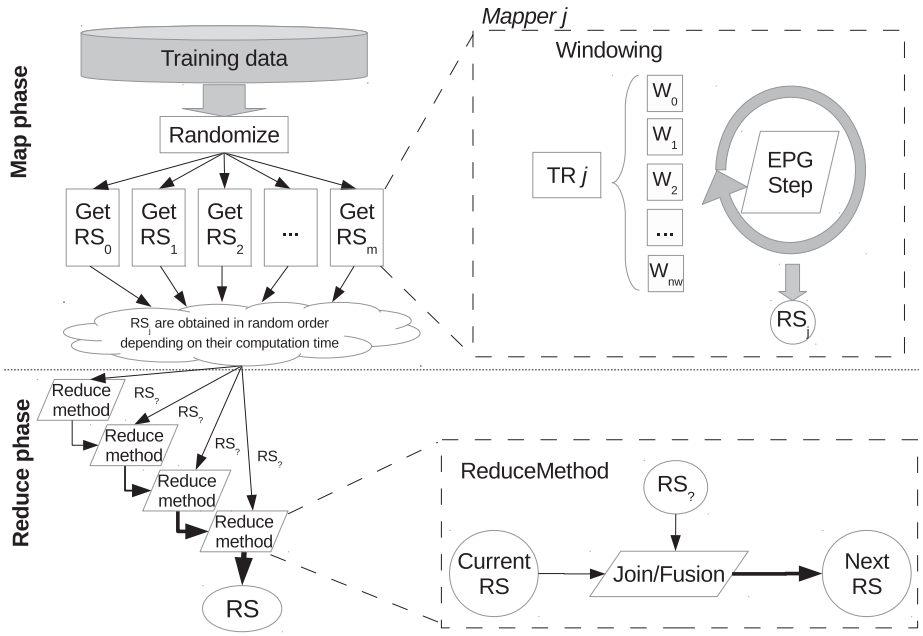


Fig. 5: MRW-EPG scheme

As we have explained, MRW-EPG only uses one single reducer that is run every time that a mapper is completed. With the adopted strategy, the use of a single reducer is computationally less expensive than use more than one, decreasing the network overhead.

As summary, Figure 5 outlines the way of working of the MRW-EPG framework, differentiating between the map and reduce phases. The resulting $RS$ will be used as training set for the NN rule to classify the unseen data of the $TS$ set.

## IV. Experimental study

In this section we present all the questions raised with the experimental study. Section IV-A establishes the experimental framework and Section IV-B presents and discusses the results achieved.

### A. Experimental Framework

We will use the following measures to characterize the abilities of MRW-EPG:

- *Accuracy:* It counts the number of correct classifications regarding the total number of instances [2].
- *Reduction rate:* It measures the reduction of storage requirements achieved by a PG algorithm.

$$ReductionRate = 1 - size(RS)/size(TR) \quad (1)$$

  Reducing the stored instances in the $TR$ set will yield a time reduction to classify a new input sample.
- *Runtime:* We will quantify the total time spent by MRW-EPG to generate the $RS$, including all the computations performed by the MapReduce framework.
- *Test classification time:* It refers to the time needed to classify all the instances of $TS$ regarding a given $TR$.
- *Speed up:* It checks the efficiency of a parallel algorithm in comparison with a slower version. In our experiments we will compute the speed up achieved depending on the number of mappers and windows.

$$Speedup = \frac{reference\_time}{parallel\_time} \quad (2)$$

  where $reference\_time$ is the runtime spent by the algorithm we want to accelerate and $parallel\_time$ is the runtime achieved with its improved version.

TABLE I: Parameter specification for all the methods

| *Algorithm* | *Parameters* |
|---|---|
| MRW-EPW | Mappers = 16/32/64/128, Reducers= 1<br>Windows = [1-7], ReduceType = Join/Fusion. |
| IPADECS<br><br>ICLP2 (Fusion) | PopulationSize = 10, iterations of Basic DE = 500<br>iterSFGSS =8, iterSFHC=20, Fl=0.1, Fu=0.9<br>Filtering method = RT2 |
| NN | Number of neighbors = 1, Euclidean distance. |

The experiments have been carried out on twelve nodes in a cluster: a master node and eleven compute nodes. Each one of these compute nodes has 2 Intel Xeon CPU E5-2620 processors, 6 cores per processor (12 threads), 2.0 GHz and 64GB of RAM. The network is Gigabit ethernet (1Gbps). In terms of software, we have used the Cloudera's open-source Apache Hadoop distribution (Hadoop 2.0.0-cdh4.4.0). A maximum of 128 map tasks are available and one reducer.

This experimental study is focused on analyzing the different components of MRW-EPG. To do so, we will use

TABLE II: Results obtained without using the windowing scheme ($nw = 1$).

| Reduce type | #Mappers | Training | | Test | | Runtime | | Reduction rate | | Classification |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Avg. | Std. | Avg. | Std. | Avg. | Std. | Avg. | Std. | time ($TS$) |
| Join | 16 | **0.5037** | 0.0013 | **0.5035** | 0.0017 | 14684.2590 | 1640.2804 | 99.9449 | 0.0025 | 43.8560 |
| Fusion | 16 | ***0.5121*** | 0.0028 | ***0.5120*** | 0.0031 | 15058.4740 | 1824.6586 | 99.9863 | 0.0007 | 26.2472 |
| Join | 32 | 0.4979 | 0.0018 | 0.4977 | 0.0018 | 7018.4668 | 334.6388 | 99.8990 | 0.0023 | 63.7146 |
| Fusion | 32 | **0.5089** | 0.0031 | **0.5086** | 0.0029 | 6963.5734 | 294.3580 | 99.9772 | 0.0018 | 28.1252 |
| Join | 64 | 0.4947 | 0.0029 | 0.4947 | 0.0032 | 3309.3414 | 176.6006 | 99.8152 | 0.0031 | 98.2246 |
| Fusion | 64 | **0.5074** | 0.0023 | **0.5076** | 0.0021 | 3212.5440 | 200.6037 | 99.9548 | 0.0025 | 32.2760 |
| Join | 128 | 0.4895 | 0.0041 | 0.4887 | 0.0049 | 1966.6030 | 193.9369 | 99.6530 | 0.0096 | 163.4240 |
| Fusion | 128 | 0.4969 | 0.0028 | 0.4973 | 0.0034 | 1976.9906 | 277.8757 | 99.8769 | 0.0029 | 55.2700 |
| NN | 0 | 0.5003 | 0.0007 | 0.5001 | 0.0011 | – | – | – | – | 48760.8242 |

IPADECS as a representative EPG method and we will test its performance over the PokerHand data set. It has been taken from the UCI repository [22] and contains 1025010 instances, 10 features and 10 different classes.

This data set has been partitioned using a 5 fold cross-validation (5-fcv) scheme. Because of the randomness of some operations that IPADECS includes, the MRW-EPG has been run three times per partition. Table I presents all the parameters involved in our experimental study. These parameters have been fixed according to the author's recommendations. Our research is not devoted to optimize the accuracy obtained with a PG method over a specific problem. We focus our experiments on the analysis of the behavior of the proposed parallel system. We will study the influence of the number of mappers, windows and type of reduce regarding to the accuracy achieved and the runtime needed.

### *B. Results and discussion*

This section presents and analyzes the results collected in the experimental study. Firstly, we only focus on the first level of parallelization, studying the results of MRW-EPG without using windowing (it corresponds to the case in which $nw = 1$). Table II summarizes the results obtained on PokerHand. It shows the training/test accuracy, runtime and reduction rate obtained by the IPADECS algorithm, in our MRW-EPG framework, depending on the number of mappers (#Mappers) and the reduce type. For each one of these measures, average (Avg.) and standard deviation (Std.) results are presented (from the 3x5-fcv experiment). Moreover, the average classification time in the $TS$ is computed as the time needed to classify all the instances of $TS$ with the corresponding $RS$ generated by MRW-EPG. Furthermore, we compare these results with the accuracy and the test classification time achieved by the NN classifier. It uses the whole $TR$ set to classify all the instances of $TS$. In these tables, average accuracies higher or equal than the obtained with the NN algorithm have been highlighted in bold. The best ones in overall are stressed in italic.

From this table we can stress several factors:

- Since within the proposed framework an EPG algorithm does not dispose of the full information about the addressed problem, it is expected that the accuracy obtained decreases as the number of available instances in the used training set is reduced. Nevertheless, in this table we can see that the performance keeps close to the one obtained with the NN rule or is even better. This situation occurs because PG techniques remove noisy instances from the $TR$ set that damage the classification performance of the NN rule. Moreover, PG models typically smooth the decision boundaries between classes, and this usually rebounds in an improvement of the generalization capabilities (test accuracy).
- Comparing join and fusion reducers, we can check that in general the fusion approach highlights as the best performing one. It is also notorious that this scheme always reports the highest reduction rate. As we explained before, the fusion scheme requires extra computations regarding to the join approach. However, we take advantage from the way of working of MapReduce, so that the reduce stage is being executed while the mappers are still finishing. In this way, most of the extra calculations needed by the fusion approach are performed before all the mappers have finished.
- Analyzing the runtime achieved, an approximate linear reduction is shown when the number of mappers is increased.
- When tackling large-scale problems, the reduction rate of a PG technique becomes much more important, maintaining the premise that the accuracy is not very deteriorated. A high reduction rate implies a significant decrease in the computational time spent to classify new instances. For example, we can see in this table that MRW-EPG can perform the classification up to 1857 times faster than the NN classifier when the fusion method and 16 mappers are used.

Taking into consideration the windowing scheme, we study the behavior of MRW-EPG varying the number of windows. Due to limitations of space, Table III only presents the numerical results obtained with different number of windows when 16/32 mappers and the fusion reducer type are used. Once again, average accuracies higher or equal than the one obtained with the NN rule have been stressed in bold. The best ones in overall are emphasized in italic.

Nevertheless, we provide visually all the resulting information. Figure 6 shows a scatterplot that compares the average accuracy test versus the average runtime needed, depending on the number of windows and mappers and the type of reducer studied. The average accuracy result of the NN rule is presented as a line $x = AverageAccuracy$, to show

TABLE III: Results obtained incorporating the windowing scheme with MRW-EPG and fusion reducer.

| #Windows $nw$ | #Mappers | Training | | Test | | Runtime | | Reduction rate | | Classification time ($TS$) |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Avg. | Std. | Avg. | Std. | Avg. | Std. | Avg. | Std. | |
| 1 | 16 | 0.5121 | 0.0028 | 0.5120 | 0.0031 | 15058.4740 | 1824.6586 | 99.9863 | 0.0007 | 26.2472 |
| 2 | 16 | 0.5115 | 0.0035 | 0.5113 | 0.0036 | 8813.7134 | 678.1335 | 99.9875 | 0.0007 | 23.8804 |
| 3 | 16 | 0.5038 | 0.0032 | 0.5039 | 0.0033 | 4666.5424 | 412.5351 | 99.9883 | 0.0010 | 26.5612 |
| 4 | 16 | 0.5052 | 0.0060 | 0.5055 | 0.0057 | 4095.8610 | 941.5737 | 99.9890 | 0.0011 | 25.8442 |
| 5 | 16 | 0.5041 | 0.0024 | 0.5034 | 0.0022 | 3244.0716 | 534.8720 | 99.9899 | 0.0015 | 25.0526 |
| 6 | 16 | 0.5031 | 0.0042 | 0.5028 | 0.0041 | 2639.4266 | 360.3121 | 99.9905 | 0.0011 | 26.6988 |
| 7 | 16 | 0.5000 | 0.0067 | 0.4998 | 0.0069 | 2099.5182 | 339.7356 | 99.9895 | 0.0010 | 25.8770 |
| 1 | 32 | 0.5089 | 0.0031 | 0.5086 | 0.0029 | 6963.5734 | 294.3580 | 99.9772 | 0.0018 | 28.1252 |
| 2 | 32 | 0.5084 | 0.0045 | 0.5080 | 0.0041 | 4092.5484 | 855.7351 | 99.9789 | 0.0016 | 30.6644 |
| 3 | 32 | 0.5067 | 0.0025 | 0.5065 | 0.0024 | 2343.1542 | 104.7222 | 99.9794 | 0.0012 | 33.6744 |
| 4 | 32 | 0.5012 | 0.0045 | 0.5012 | 0.0039 | 1639.0032 | 335.6036 | 99.9785 | 0.0015 | 26.8272 |
| 5 | 32 | 0.5012 | 0.0045 | 0.5012 | 0.0039 | 1639.0032 | 335.6036 | 99.9785 | 0.0015 | 26.8272 |
| 6 | 32 | 0.4824 | 0.0104 | 0.4820 | 0.0101 | 1083.1116 | 143.9288 | 99.9768 | 0.0019 | 35.1896 |
| 7 | 32 | 0.4838 | 0.0072 | 0.4835 | 0.0065 | 1129.8838 | 173.9482 | 99.9757 | 0.0024 | 35.4692 |

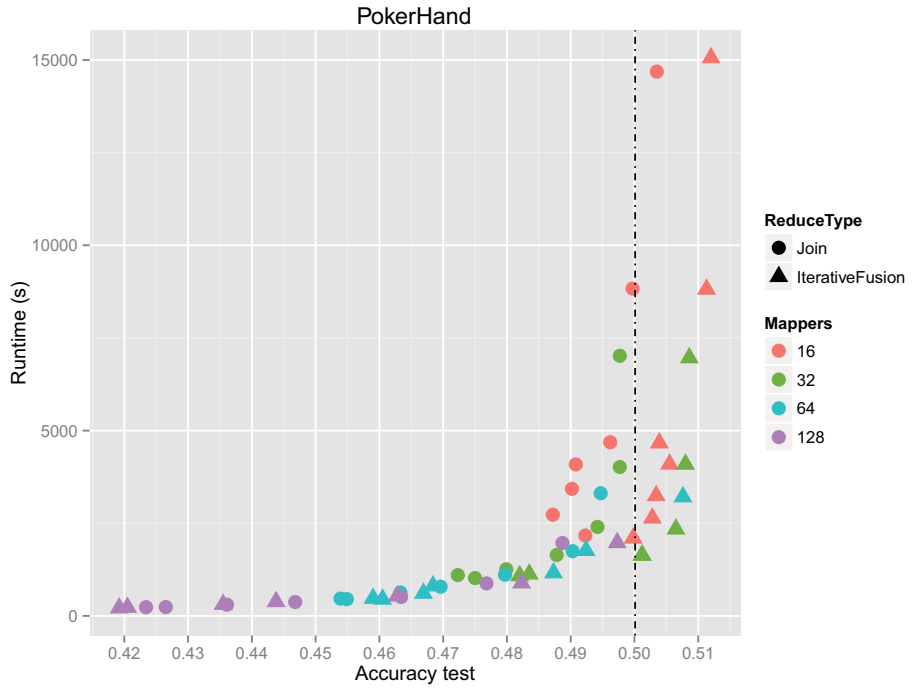


Fig. 6: PokerHand: Accuracy Test vs. Runtime results obtained by MRW-EPG

the accuracy differences between using the whole $TR$ or a generated $RS$ as training data set.

To apply MRW-EPG, we have to consider how representative the used training subset ($W_j$) is, regarding to the original $TR$. Table IV shows the approximate number of instances per chunk, that is, the size of each $W_j$ for MRW-EPG, attending to the number of mappers and windows established.

TABLE IV: Approximate number of instances in each $W_j$ subset according to the number of mappers and windows used. PokerHand data set

| Number of mappers | Number of windows | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 16 | 51250.50 | 25625.25 | 17083.50 | 12812.63 | 10250.10 | 8541.75 | 7321.50 |
| 32 | 25625.25 | 12812.63 | 8541.75 | 6406.31 | 5125.05 | 4270.88 | 3660.75 |
| 64 | 12812.63 | 6406.31 | 4270.88 | 3203.16 | 2562.53 | 2135.44 | 1830.38 |
| 128 | 6406.31 | 3203.16 | 2135.44 | 1601.58 | 1281.26 | 1067.72 | 915.19 |

Finally, Figure 7 depicts the speed up achieved with the windowing scheme in MRW-EPG, fixing the number of mappers to 16. The speed up has been computed using the runtime spent with one window as the reference time.

According to these results, we can make some comments:

- As it could be expected, an increment in the number of windows implies a slight reduction of the accuracy capabilities of the EPG technique. With the partitioning scheme, the performance should decrease when the number of available instances is lesser than a determined threshold. Exceeding this limit, the used data do not represent the addressed problem. According to Table IV and Figure 6, we can see that for the PokerHand data set, it occurs when the number of instances is lesser than 10000. Thus, a higher number of mappers and windows may deteriorate the performance of the IPADECS.

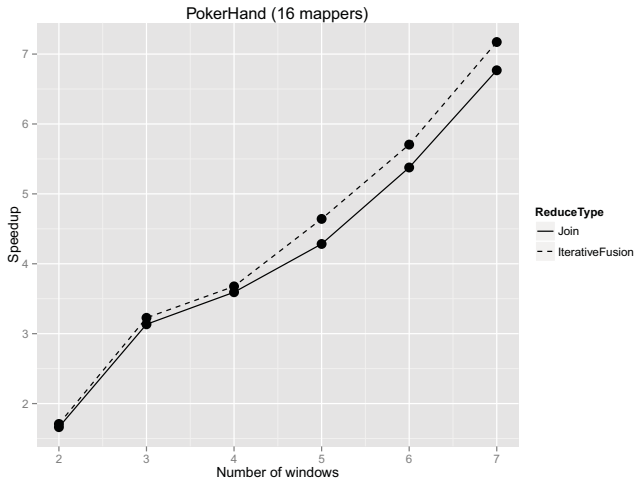


Fig. 7: PokerHand: Speed up

- Comparing the results obtained with one window with the use of more windows, the windowing scheme provides higher accuracy test results with the same number of available instances (see Table IV). For example, by using 16 mappers and 4 windows, in comparison to 32 mappers and 2 windows, IPADECS uses approximately 12800 instances to evaluate the fitness function, but it performs better with the first setting. As explained in Section III-B.1, it is due to the fact that with the windowing scheme the EPG algorithm utilizes all the information although it is accessed partially during the evolutionary cycle. Thus, it is important to find a trade-off between number of mappers and windows.
- In terms of runtime and speed up, we observe that the model reduces the runtime computation in a linear way (approximately) as the number of mappers or windows is incremented. The runtime crucially depends on the number of instances that the algorithm has to evaluate during its evolutionary process. In Figure 7, a superlinear speed up is shown for some number of windows (3 and 7). It is due to the fact that the windowing scheme modifies the way of working of IPADECS. Since this algorithm decides automatically its number of iterations according to the fitness function, it may occur that it finishes earlier.

## V. Concluding remarks

In this contribution we have developed a two-level parallelization scheme for evolutionary prototype generation. At first level, it is based on a MapReduce framework that is later enhanced with a windowing approach. The experimental study carried out has shown the good synergy between the windowing and MapReduce approaches and how they complement themselves in our two-level proposal. The application of this model has resulted in a very big reduction of storage requirements and classification time for the NN rule, when dealing with large data sets. Without this model, evolutionary prototype generation could not be applied to data sets larger than approximately ten thousands instances.

## References

[1] M. Minelli, M. Chambers, and A. Dhiraj, *Big Data, Big Analytics: Emerging Business Intelligence and Analytic Trends for Today's Businesses (Wiley CIO)*, 1st ed. Wiley Publishing, 2013.

[2] E. Alpaydin, *Introduction to Machine Learning*, 2nd ed. MIT Press, Cambridge, MA, 2010.

[3] J. Bacardit and X. Llorà, "Large-scale data mining using genetics-based machine learning," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 3, no. 1, pp. 37–61, 2013.

[4] J. Dean and S. Ghemawat, "Map reduce: A flexible data processing tool," *Communications of the ACM*, vol. 53, no. 1, pp. 72–77, 2010.

[5] I. Palit and C. Reddy, "Scalable and parallel boosting with mapreduce," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 10, pp. 1904–1916, 2012.

[6] A. Srinivasan, T. Faruquie, and S. Joshi, "Data and task parallelism in ILP using mapreduce," *Machine Learning*, vol. 86, no. 1, pp. 141–168, 2012.

[7] J. Derrac, I. Triguero, S. Garcia, and F. Herrera, "Integrating instance selection, instance weighting, and feature weighting for nearest neighbor classifiers by coevolutionary algorithms," *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, vol. 42, no. 5, pp. 1383–1397, 2012.

[8] I. Triguero, J. Derrac, S. García, and F. Herrera, "A taxonomy and experimental study on prototype generation for nearest neighbor classification," *IEEE Transactions on Systems, Man, and Cybernetics–Part C: Applications and Reviews*, vol. 42, no. 1, pp. 86–100, 2012.

[9] T. M. Cover and P. E. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967.

[10] I. Triguero, S. García, and F. Herrera, "IPADE: Iterative prototype adjustment for nearest neighbor classification," *IEEE Transactions on Neural Networks*, vol. 21, no. 12, pp. 1984–1990, 2010.

[11] J. R. Cano, F. Herrera, and M. Lozano, "Stratification for scaling up evolutionary prototype selection," *Pattern Recognition Letters*, vol. 26, no. 7, pp. 953–963, 2005.

[12] J. Bacardit, D. E. Goldberg, M. V. Butz, X. Llorà, and J. M. Garrell, "Speeding-up pittsburgh learning classifier systems: Modeling time and accuracy," in *Parallel Problem Solving from Nature - PPSN VIII*, ser. LNCS, vol. 3242, 2004, pp. 1021–1031.

[13] I. Triguero, J. Derrac, S. García, and F. Herrera, "A study of the scaling up capabilities of stratified prototype generation," in *Proceedings of the third World Congress on Nature and Biologically Inspired Computing (NABIC'11)*, 2011, pp. 304–309.

[14] I. Triguero, S. García, and F. Herrera, "Enhancing IPADE algorithm with a different individual codification," in *Proceedings of the 6th International Conference on Hybrid Artificial Intelligence Systems (HAIS). LNAI 6679*, 2011, pp. 262–270.

[15] S. Das and P. Suganthan, "Differential evolution: A survey of the state-of-the-art," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 1, pp. 4–31, 2011.

[16] J. Derrac, S. García, and F. Herrera, "Stratified prototype selection based on a steady-state memetic algorithm: a study of scalability," *Memetic Computing*, vol. 2, no. 3, pp. 183–199, 2010.

[17] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[18] A. H. Project, "Apache hadoop," 2013. [Online]. Available: http://hadoop.apache.org/

[19] W.-Y. Chen, Y. Song, H. Bai, C.-J. Lin, and E. Chang, "Parallel spectral clustering in distributed systems," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 33, no. 3, pp. 568–586, 2011.

[20] C.-L. Chang, "Finding prototypes for nearest neighbor classifiers," *IEEE Transactions on Computers*, vol. 23, no. 11, pp. 1179–1184, 1974.

[21] W. Lam, C. K. Keung, and D. Liu, "Discovering useful concept prototypes for classification based on filtering and abstraction." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 14, no. 8, pp. 1075–1090, 2002.

[22] A. Frank and A. Asuncion, "UCI machine learning repository," 2010. [Online]. Available: http://archive.ics.uci.edu/ml