Swarm and Evolutionary Computation 38 (2018) 240–250

Contents lists available at ScienceDirect

Swarm and Evolutionary Computation

journal homepage: www.elsevier.com/locate/swevo

A distributed evolutionary multivariate discretizer for Big Data processing on Apache Spark

S. Ramírez-Gallego[b,*], S. García[a,b], J.M. Benítez[b], F. Herrera[b]

[a] Department of Computer Science and Artificial Intelligence, CITIC-UGR, University of Granada, 18071 Granada, Spain
[b] Faculty of Computing and Information Technology, King Abdulaziz University, North Jeddah, Saudi Arabia

ARTICLE INFO

Keywords:
Discretizacion
Evolutionary computation
Big Data
Data Mining
Apache Spark

ABSTRACT

Nowadays the phenomenon of Big Data is overwhelming our capacity to extract relevant knowledge through classical machine learning techniques. Discretization (as part of data reduction) is presented as a real solution to reduce this complexity. However, standard discretizers are not designed to perform well with such amounts of data. This paper proposes a distributed discretization algorithm for Big Data analytics based on evolutionary optimization. After comparing with a distributed discretizer based on the Minimum Description Length Principle, we have found that our solution yields more accurate and simpler solutions in reasonable time.

1. Introduction

Among all Data Mining tasks, Data Preprocessing [1,2] stands as one of the most important steps in the knowledge discovery process. As input data must be provided in a suitable structure and format for a subsequent high-quality mining process, Data Preprocessing becomes essential in most of data analytic problems. Preparation techniques aim at cleaning negative factors present in current databases –missing, noise, inconsistent and superfluous data–. Conversely, data reduction family is applied to simplify data and their inherent complexity, while maintaining their original structure. Discretization [3,4], as part of data reduction, has received increasing attention in last years. It transforms quantitative data into qualitative data by performing a non-overlapping partitioning of continuous attributes, and then associating a set of discrete values to the resulting partitions.

Although the Data Mining discipline has been successfully applied for several years [5], in this new era of Big Datum [6,7], the capabilities of traditional mining systems have been surpassed by the exponential growth of databases. Learning from large-scale datasets has become a labored or even impracticable task when classical algorithms are used. As in standard mining, Big Data preprocessing [8] plays an essential role in improving the quality of large-scale data. This importance can be deemed even greater in Big Data scenario since large amounts of data usually implies more noise. Novel scalable, and efficient discretizers [4], developed on recent distributed paradigms and tools [9], are thus required to face the Big Data discretization problem. Up to date, only one distributed solution for Big Data discretization [4] has been presented in the literature.

Data discretization can be deemed as an optimization problem, where partial solutions can be coded via binary representation. Given the previous problem, an evolutionary-based metaheuristic [10] can be useful at dealing with binary-based optimization. Although quite effective, evolutionary algorithms are known for being time-consuming and hardly scalable, specially when large-scale problems are faced [11]. A distributed solution based on evolutionary heuristics would bring us an scalable and effective solution for Big Data discretization [12].

Evolutionary algorithms (EA) have shown their usefulness on several optimization-based learning problems, see the following overviews for several mining contexts, such as: rule learning [13], evolutionary fuzzy systems [14], clustering [15], or multi-objective learning [16]. Recently, we can find novel evolutionary and bio-inspirited approaches for learning, such as: data discretization [17], rule induction [18], feature selection [19], clustering [20], or diverse applications, like face recognition [21].

In this paper we propose a novel design for a distributed multivariate discretizer for Apache Spark [22] based on an evolutionary points selection scheme. Our approach, called Distributed Evolutionary Multivariate Discretizer (DEMD), has been inspired by the EMD discretizer [17]. EMD is an evolutionary-based discretizer with binary representation and a wrapper fitness function. Although both algorithms share some common aspects (like representation and fitness function), DEMD goes beyond a simple parallelization, and offers an approximative, scalable and resilient solution to deal with the Big Data discretization problem. Alike EMD, in DEMD, partial solutions are generated locally, and eventually fused to produce the final discretization scheme. Up to our knowledge, our proposal is the first evolutionary

* Corresponding author at: Department of Computer Science and Artificial Intelligence, CITIC-UGR, University of Granada, 18071 Granada, Spain.
E-mail addresses: sramirez@decsai.ugr.es (S. Ramírez-Gallego), salvagl@decsai.ugr.es (S. García), J.M.Benitez@decsai.ugr.es (J.M. Benítez), herrera@decsai.ugr.es (F. Herrera).

http://dx.doi.org/10.1016/j.swevo.2017.08.005
Received 28 December 2016; Received in revised form 19 April 2017; Accepted 19 August 2017
Available online 23 August 2017

approach in dealing with the large-scale discretization problem.

In order to show the usefulness of our solution, a thorough experimental evaluation has been performed using several huge datasets (up to $O(10^7)$ instances and $O(10^4)$ features). A distributed discretizer based on the Minimum Description Length Principle (called DMDLP) [4] has also been included for comparison purposes. Experiments on these real-world datasets have shown that DEMD obtains more accurate and simpler discretization schemes than its competitor.

The remainder of this paper is organized as follows. Section 2 briefly explains some concepts about Big Data, and the environment of tools and paradigms around this phenomenon. Section 3 introduces the discretization task, some related concepts, as well as a brief description of EMD. Section 4 discusses the complexity problems faced, as well as the solution adopted by our proposal. Section 5 describes the experimental framework carried out, and the results derived from these experiments. Lastly Section 6 gives the conclusions derived from this work.

2. Big Data: concepts, paradigms and tools

In this section, the Big Data phenomenon is introduced through the scheme of 5Vs. Here we also present the distributed frameworks, paradigms and tools which have served to address Big Data problems in a distributed manner.

Humongous amounts of information are stored in data centers now, ready to be processed. The efficient extraction of valuable knowledge from these datasets raises a considerable challenge for data scientists. Gartner [23] introduced the popular concept of Big Data in 2001. In its report, Gartner defines this concept as the conjunction of the 3Vs: high volume, velocity and variety information that require a new large-scale processing. This list was extended with 2 extra terms: veracity and value.

One of the most relevant frameworks in Big Data analytics is the MapReduce framework [24]. This framework, devised by Google in 2003, allows us to automatically process huge data by distributing the complexity burden among a cluster of machines. Final users only have to design their tasks specifying the Map and Reduce functions. Partitioning and distributing of data, job scheduling or fault-tolerance are responsibility of the platform.[1]

One of the most popular open-source implementation of MapReduce is Apache Hadoop [25,26]. Despite of being well-used and very popular, Hadoop seems not to work well with iterative and online processes [27]. In general, it is not intended for those programs that continuously reads data and need to keep them in memory.

The MapReduce model offers two primitives –Map and Reduce–, which correspond with two execution stages in the whole process. Firstly, the master node retrieves the dataset (split into several chunks) from the distributed file system so that each node reads those data chunks allocated in its local disk. Each node then starts one or more Map threads to process the raw chunks. The result is a set of key-value pairs (intermediate pairs), which are also stored in disk. After all Map tasks have ended, the master node starts the Reduce phase by distributing those pairs with coincident keys to the same node. Each Reduce task combines those matching pairs to yield the final output. Fig. 1 depicts a simplified scheme of MapReduce and its two main functions.

Related to the Hadoop Ecosystem, Apache Spark [28,22] has emerged as a new revolutionary tool capable of outperforming Hadoop for certain cases (100x faster). This is possible due to the in-memory primitives available in Spark. This platform allows users to persist data into memory and to read them rapidly, making it suitable for iterative and online jobs.

[1] For a complete review of this model and other distributed models, please check [9].

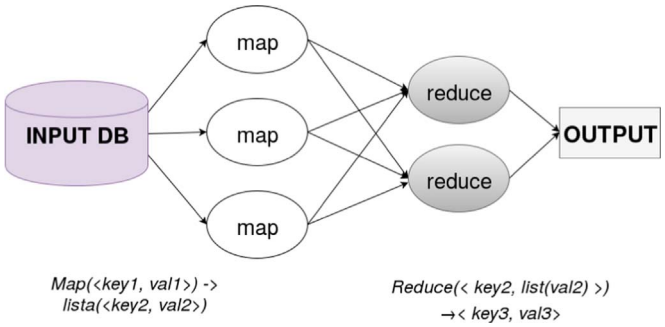

Fig. 1. Scheme of the MapReduce paradigm.

3. Discretization: theoretical background

In this section, the problem of discretization, as well as some optimizations are presented. Additionally, a brief description of EMD is also given.

3.1. Definitions

Discretization is a data preprocessing technique that generates disjoint intervals from continuous features. The resulting intervals are associated to a set of discrete values to yield nominal data. A complete description and taxonomy of discretization algorithms can be found in [3,4].

Given a dataset D with n examples, a set of features F, the subset of continuous features $FC \subset F$, and o target classes, a discretization process would divide a continuous feature c into k_c disjoint and discrete intervals, yielding the following discretization scheme:

$$D_c = \{[d_0, d_1], (d_1, d_2], \ldots, (d_{k_c-1}, d_{k_c}]\} \quad (1)$$

where d_0 and d_{k_c} are the minimum and maximum value, respectively. Note that all values in D_c are sorted in ascending order. Likewise,

$$CP_c = \{d_1, d_2, \ldots, d_{k_c-1}\} \quad (2)$$

is defined as the set of cut points of feature c, and CP denotes the whole set of cut points for all the continuous features in D.

Optimal discretization can be considered as a NP-hard problem [29], where the search space is basically composed by all the different values (for all features) in the training set. To alleviate the subsequent complexity, it can be considered a reduced subset of points, formed by the *boundary points* among classes.

Let c has its values sorted in ascending order, and val_c be a function that returns the value for c, given an example in D. Given two examples $a, b \in D$ with different classes, such that $val_c(a) < val_c(b)$. If there is no sample $c \in D$ such that $val_c(a) < val_c(c) < val_c(b)$, a boundary point can be defined as the half-point value between $val_c(a)$ and $val_c(b)$. The set of boundary points for c is denoted as BP_c, whereas the complete set as BP.

Boundary points has shown to form the optimal intervals for most of the evaluation measures [30], thanks to that the previous definition always search the maximum separability between classes. Additionally, a reduced subset of points offers significant savings in complexity, as shown in [17]. This fact is specially relevant in Big Data, where the performance is a determining factor.

3.2. Evolutionary multivariate discretizer

An important contribution to the discretization field is EMD [17], an evolutionary-based discretizer that uses a wrapper fitness function to evaluate binary solutions. This approach goes beyond deterministic algorithms, and offers the possibility of improving the discretization schemes generated by tuning several parameters. EMD follows a

multivariate approach to leverage the existing dependencies among different features.

EMD utilizes the operators and mechanisms defined in CHC [31] to tackle the cut points selection problem. CHC is a well-known evolutionary algorithm specially designed for binary optimization, which looks for a proper balance between exploration and exploitation. Our algorithm includes several features from CHC in order to achieve this balance, such as: incest prevention (via mating threshold) or chromosome reboot.

EMD consists of two steps that are constantly repeated in each iteration. Firstly, pairs of chromosomes are fused using a crossover operation. Intermediate population is then randomly paired to generate new offspring. Afterwards a survival competition decides what is the best solution from the set of parent and offspring. The result is a new population which will be mixed and selected in further steps.

Like CHC, EMD implements HUX, a heterogeneous crossover operation to recombine chromosomes. HUX is specially designed to generate maximally distant offspring from two parents by exchanging half of the different points according to the Hamming distance. Regarding the mutation operator, EMD replaces it by a rebooting process which randomly changes 35% of the bits in the best chromosome in order to create new templates. This reseeding process will be applied whenever the population gets stuck for a number of evaluations.

This discretizer uses a binary chromosome representation to annotate the selection (1) or not-selection (0) of all the boundary points in BP. There is therefore an unique correspondence between each gene and each boundary point. Finally, all points marked as selected in the best chromosome will be included in S.

In order to evaluate the different binary solutions (chromosomes), EMD defines a wrapper fitness function that aggregates two factors: the classification error on training and the number of cut points selected. This function has showed more promising accuracy results and simpler solutions than those based on inconsistency measures [32]. The objective of this EA is the minimization of this function, which is defined as follows:

$$Fitness(P') = \alpha \cdot \frac{|P'|}{|BP|} + (1 - \alpha) \cdot \Delta \quad (3)$$

where P' is the subset of selected points, Δ the error obtained after classifying the discretized data, and α a weight factor for these two factors.

In EMD, the classification error is computed as the average mean error yielded by two classifiers: Naïve Bayes [33] and an unpruned version of C4.5 [34].

EMD also introduces a reduction mechanism to speed up the convergence of our method, which is based on the reduction of the chromosome size. In our solution, this is done by maintaining those points selected more than a determined number of evaluations (the most relevant ones). In each reduction phase, several points are removed according to a counter that indicates the number of selections by point.

This reduction mechanism fixes the long delays associated to the application of EAs when facing huge problems. As any optimization problem, the cut points selection problem offers multiple valid solutions (local optima). It is thus convenient to reach some local optima in order to avoid long executions.

4. Distributed evolutionary multivariate discretizer

This section explains the design of our distributed discretization solution for Big Data. Our algorithm splits both the set of cut points and instances into partitions, and evaluates them through a cross-evaluation system. With this distributed scheme we maximize the resource usage throughout the entire process. If a point is selected by one of the evaluation processes, it counts as a single vote. All these votes are aggregated to obtain the final score per point. Finally, the final discretization scheme is obtained through a voting scheme.

Section 4.1 starts discussing the main concerns that affect our distributed approach: a large number of instances and cut points. In Section 4.2, the main procedure in charge of partitioning the instances are feature, and aggregating the partial solutions is presented. Section 4.3 illustrates the process of computing boundary points. Section 4.4 exposes how the chromosomes are evaluated in a distributed manner by using EMD.

4.1. Discussion about the DEMD's distributed design

This section presents the main problems that our proposal needs to overcome to produce discretization schemes efficiently. The two problems to consider are: a high number of cut points to evaluate, and therefore, long chromosomes to evaluate; and a huge amount of instances to use in this evaluation phase.

The first problem is related to the high complexity derived from EA problems. In the cut points selection problem, discretizers are mainly affected by the number of boundary points to evaluate (long chromosomes). In particular, this problem is influenced by two factors: the number of instances and features present in the problem. Another hidden factor that influences the complexity is the number of distinct points present in each feature. If this value is high, the algorithm will have to process a high number of boundary points.

In order to keep the multivariate philosophy and to alleviate the complexity derived from these two problems, our distributed proposal has introduced some major changes with respect to the sequential version. The first change we propose is to divide the complete set of features into partitions so that the evaluation of points is performed in a parallel way. This modification has also demonstrated to maintain the effectiveness of the original method.

For the second problem (high number of instances), we propose to partition the set of instances into a set of equal-sized partitions. Each data partition will serve to evaluate different parts of the chromosome. Once the partitions have been evaluated following the EMD scheme (Section 3.2), the subsequent partial solutions are aggregated through a voting scheme (complete description in Section 4.4). This modification has showed to work well with large datasets, even when the generated schemes are approximative.

Regarding the evaluation, Naïve Bayes has been elected to evaluate the candidate solutions in the fitness function because of its simplicity and efficiency in its close-form expression [35] (linear order). Nevertheless, users can introduce another classifier/s to customize the distributed approach.

To implement our method, some extra primitives from Spark's API have been used. Spark primitives implement more complex operations than those proposed by MapReduce. Some of them are: mapPartitions, broadcast, sortByKey, Map and reduceByKey.[2]

DEMD includes several user-defined input parameters, which are described in Table 1.

4.2. Main discretization procedure

Procedure 1 explains the main procedure of our discretization algorithm. Hereafter we will use the term partition to describe the data partitions, and the term chunk to describe the feature partitions. This procedure is in charge of distributing the initial cut points (computed in Section 4.3) among the set of chunks. The partitions already created are associated with these chunks so that each chunk is evaluated on the instances contained in one or more partitions. After the parallel selection process is performed (in Section 4.4), this procedure creates

[2] For a complete description of Spark's operations, please refer to Spark's API: https://spark.apache.org/docs/latest/api/scala/index.html.

the final matrix of selected cut points.

The first step computes the boundary points (BF) in a distributed way using the function *getBoundary* (line 1, Section 4.3). Each tuple in BF consists of a feature ID (fid) and a list of points. Based on this variable, DEMD creates FI (feature information), and BP (boundary points per feature). All this information will serve us to create the chromosome chunks.

Procedure 1. Main discretization procedure.

```
Input:  D dataset
Input:  M Feature indexes to discretize
Input:  uf Multivariate user factor
Input:  alp Alpha parameter
Input:  ne Number of evaluations
Input:  sr Sampling rate
Input:  vp Percentage of selected points
Output: Cut points by feature
1:  BF ← getBoundary(D, M)
2:  BP ← (); FI ← ();
3:  for all <fid, l> ∈ BF do
4:      BP(fid) = l
5:      FI.add(Feature(fid, l.size))
6:  end for
7:  FI ← broadcast(sortBySize(FI))
8:  BP ← broadcast(BP)
9:  nbp ← totalSize(BP)
10: ds ← nbp/D.npartitions
11: ms ← max(FI(0).size, ds)
12: df ← max(uf, ms/ds)
13: ncp ← nbpoints/(df*ds)
14: windows ← makeGroups(FI, ncp)
15: CH ← ()
16: for all w ∈ windows do
17:     p ← shuffle(w)
18:     for i = 0 → i < p.size do
19:         CH(i).add(p(i))
20:     end for
21: end for
22: CH ← broadcast(CH)
23: SD ← stratifiedSampling(D, sr)
24: SP ← select(SD, CH, uf, alp, sr, vp)
25: TH ← ()
26: for <chid, lf> ∈ SP do
27:     ind ← 0; chunk ← CH(chid)
28:     for feat ∈ chunk do
29:         for i = 0 → i < feat.size do
30:             if lf(i + ind)==true then
31:                 point ← BP(feat.id)(i + ind)
32:                 TH(feat.id).add(point)
33:             end if
34:         end for
35:         ind ← ind + feat.size
36:     end for
37: end for
38: return(TH)
```

The procedure divides the evaluation of cut points using subsets of features (called chunks) (lines 2–13). To do that DEMD first sorts all features by the number of boundary points contained in each one (ascending order). Then, DEMD computes the number of chunks (ncp) in which the entire list of boundary points will be divided. ncp is computed using several variables which are related according to Eq. (4).

$$ncp = np/(max(uf, ms/ds)\cdot ds) \tag{4}$$

where np is the total number of boundary points, ds the current proportion of points by data partition, uf the split factor specified by the user, and ms the maximum between the largest feature size and ds.

Usually each feature is contained in a single chunk, but it may change in case the user specifies a greater value, or the largest feature surpasses the default size since points belonging to the same feature can not be separated. In the latter case, a finer-grained division will be performed, which means more chunks. This scenario normally entails a quicker evaluation, but a loss in effectiveness.

The evaluation procedure starts to distribute points between the chunks (CH) (lines 14-21). In each iteration, a group of nc features is collected and randomly distributed among the chunks. The loop ends when there is no feature to collect. This mechanism will enable a fairly distribution of boundary points, without points from the same feature in different chunks, and with a similar number of features per chunk.

Once the distribution of points is completed, a stratified sampling process (by class) is performed on D (line 23). The resulting sample SD is used to evaluate the boundary points in a distributed manner. According to the multivariate factor ($max(uf, ms/ds)$), each partition randomly selects as many chunks as indicated by these factor (usually only one). Then, each partition is responsible of evaluating the points contained in their associated chunks (line 24). The selection phase is described in detail in Section 4.4.

Each selection process returns its aggregated partial solution (the best chromosome per chunk), and saves the tuples (chunk ID, best solution) in SP. All these partial results are then summarized using a voting scheme, considering the threshold (vp). Finally, the main procedure processes the binary vectors to obtain the final matrix of cut points (TH) (line 26-38). This procedure fetches the features in each chunk, and its correspondent points. If a given point has been selected, it is added to the final matrix. If not, this is omitted.

An illustrative scheme of the entire process is detailed in Fig. 2. In this example, there are four features with different amounts of boundary points (8, 5, 4,10). Boundary points are then uniformly distributed into three chunks where features may be mixed, like in chunk $C1$. Afterwards chromosome chunks are grouped with seven data partitions following a correspondence table that relates chunks and partitions according to the multivariate factor. Once local evaluation threads have ended, partial discretization results (binary vectors) for the same chromosome part are aggregated by summing votes. Most-voted points in each chunk according to vp (proportion of points to select) are selected, and adapted to create the global selection matrix.

Table 1
DEMD's parameters. For each parameter, name, description and range are shown.

Parameter	Description	Range
D	Input dataset (RDD)	–
M	Feature indexes to discretize	$[0, f]$
uf	Ratio between the number of feature chunks and the number of data partitions	$[1, \infty)$
alp	Weight factor for the fitness function	[0, 1]
ne	Number of chromosome evaluations to be performed in each process	$[100, \infty)$
sr	Percentage of instances used in evaluation	[0, 1]
vp	Percentage of points selected in each aggregation process	[0, 1]

4.3. Computing the boundary points

Procedure 2. Function to generate the boundary points *(getBoundary)*.

```
Input:  D dataset
Input:  M Feature indexes to discretize
Output: The set of boundary points (feature index, point value).
1:  CB←
2:    map s ∈ D
3:      v ← zeros(|cl|)
4:      ci ← classIndex(v)
5:      v(ci) ← 1
6:      for all A ∈ M
7:          EMIT < (A, A(s)), v>
8:      end for
9:    end map
10: D ← reduce(CB, sumVectors)
11: S ← sortByKey(D)
12: FP ← firstByPart(S)
13: BP←
14:   map partitions PT ∈ S
15:       <(la, lp), lq > ← next(PT)
16:       for all <(a, p), q > ∈ PT do
17:         if a < > la then
18:             EMIT < la, lp>
19:         else if isBoundary(q, lq) then
20:           EMIT < la, (p + lp)/2>
21:         end if
22:         <la, lp > ← < a, p>
23:     end for
24:     index ← getIndex(PT)
25:     if index < npartitions(S) then
26:         <(a, p), q > ← FP(index + 1)
27:         if a < > la then
28:           EMIT < la, lp>
29:         else
30:           EMIT < la, (p + lp)/2>
31:         end if
32:     else
33:         EMIT < la, lp>
34:     end if
35:   end map
36: return(BP. groupByKey())
```

Procedure 2 (*getBoundary*) describes the function that computes border points in data. This procedure consists of three steps. Firstly, the distinct points (*D*) in the dataset are calculated by removing duplicated elements. Secondly, the resulting points are sorted (*S*) and distributed by feature index so that all the points from the same feature will not be separated. Finally, the boundary points (*BP*) in each feature are evaluated sequentially.

The procedure starts by launching a parallel process on each partition (taking advantage of data locality) with the aim of computing the distinct points (*CB*) (lines 1-9). Once the points are sorted (*D*) and the first point by partition is distributed (*FP*), DEMD evaluates whether each points belong to any border as follows (lines 13–36): for each point, it checks whether the feature index is distinct from the index of the previous point; if it is so, DEMD generates a tuple with the feature index of the last point as key, and its correspondent value as value. By doing so, the last point from the current feature is always kept as the last threshold. If there are more points in this feature, the procedure evaluates whether the current point accomplishes the boundary condition with respect to the previous point. If it is so, this generates a tuple with the feature index as key, and the midpoint between these two points as value.

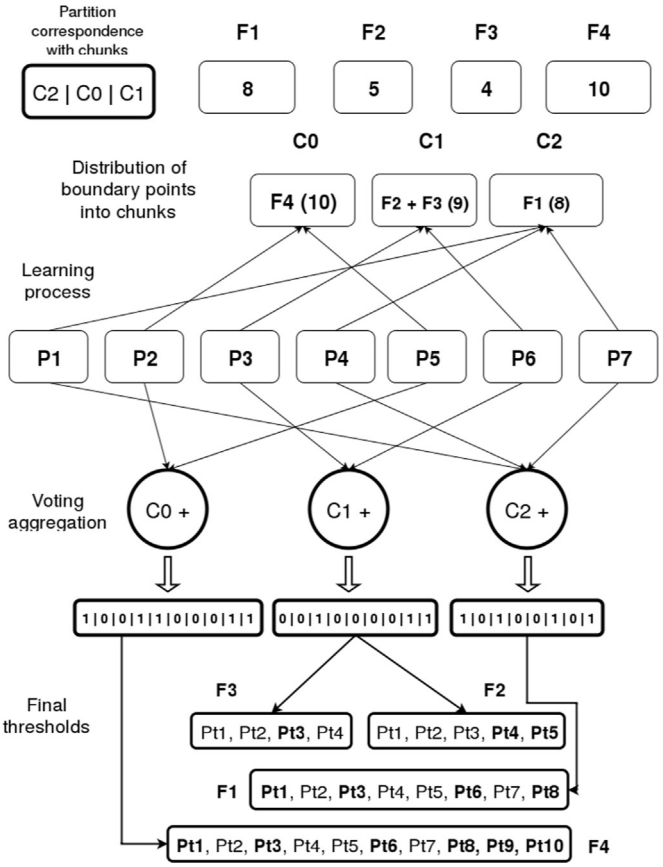

Fig. 2. A simplified representation of the DEMD process. F represent the features, C the chromosome chunks, P the dataset partitions to evaluate, and Pt the boundary points. The selected points have been highlighted in bold.

The last point in each partition is considered as an special case (lines 25–34). These points are compared with the first point in the following partition (broadcasted). If the feature indexes are different, the procedure emits a tuple with the last point. If not and the point is boundary, DEMD emits a tuple with the midpoint between these two points. Finally, all the tuples generated in each partition are joined into a RDD of boundary points, which is returned to the main procedure.

The previous process is depicted in Fig. 3. In this figure we can see three partitions with six different points. The points are sorted by key (feature index and point value) to perform the evaluation. The first point for each partition is sent to the following partition to perform the evaluation of the last points. As result, three boundary points are generated, some are midpoints and some are the last points in features.

4.4. Distributed cut points selection

Procedure 3 explains the distributed operations used to aggregate the solutions generated by each local evaluation process, and to decide the final discretization scheme. Note that local evaluation of points is performed by launching a single instance of EMD on each data partition (Section 3.2). This process consists of two steps: the first one starts a selection process (map) on each pair chunk-partition and aggregates the subsequent solutions to produce the final number of votes. The second step is aimed at selecting the most voted points by chunk according to the threshold (*vp*) defined by the user.

Procedure 3. Function to perform the evolutionary selection process *(select)*.

```
Input:  SM Sampled boundary points
Input:  CH Feature chunks
Input:  uf Multivariate user factor
Input:  alpha Alpha parameter (evolutionary process)
Input:  ne Number of evaluations (evolutionary process)
Input:  sr Sampling rate
Input:  vp Percentage of selected points
Output: Cut points by feature
 1:  PO ← shuffle(seq(0, CH. size))
 2:  R←
 3:    map partitions <index, DT > ∈ SM
 4:          chid ← PO(index%CH. size)
 5:          C ← CH(chid)
 6:          npoints ← totalSize(chunk)
 7:          CP ← (); FD ← ()
 8:          for i = 0 → i < chunk. size do
 9:            FD(i) ← DT(i)
10:            CP(i) ← C(i)
11:          end for
12:          BI ← EMD(FD, CP, alpha, ne)
13:          CO ← ()
14:          for i = 0 → i < BI. size do
15:            if BI(i)==1 then
16:                  CO(i) = 1
17:            else
18:                  CO(i) = 0
19:            end if
20:          end for
21:          EMIT < chid, (CO, 1)>
22:    end map
23:  CO ← R. reduceByKey(sum())
24:  SL←
25:    map <chid, (AC, c) > ∈ CO
26:          S ← sort(AC)
27:          ps ← AC. size*vp
28:          BA ← take(S, ps)
29:          EMIT < chid, BA>
30:    end map
31:  return(SL)
```

Firstly, each chunk is associated with one or more data partitions using *PO*, which is a table formed by tuples (chunk ID, data partition). For each tuple, a map operation is started (lines 3–22). This map operation starts by creating a data matrix with the instances contained on each partition and those features present in the chunk. Afterwards, the procedure executes an evaluation thread on each submatrix (*FD*) in order to evaluate the corresponding boundary points (*CP*). As result, the best chromosome (a binary vector) in the population is returned (*BI*).

The binary vector will be transformed into a numeric vector to annotate number of selections (CO) (lines 13–22). The final result emitted by the partition is a tuple with the identifier of the chunk as key, and the vector count –number of times each point has been selected– and a chunk count –maximum number of partitions in which has been evaluated– as value. This procedure will indicate the selection ratio for each point. The partial values generated above are aggregated by reducing the tuples by key.

Secondly, the procedure starts a map operation (lines 25–30) to select the most voted points by chunk (*SL*). The procedure orders all the points by number of votes, and selects in order as much points as specified by *vp*. The result is a tuple with chunk ID as key, and the selection vector as value. Finally, previous results will eventually be transformed to a matrix of points in the main procedure.

4.5. Computational and communication complexity analysis

In this section we analyze the computational and communication complexity for all procedures presented. Big O notation is used to specify the upper limit for the run-time and communication cost of each procedure.

- Boundary points (line 1 – Algorithm 2): complexity here is determined by the computation of distinct points: $O(\frac{|D|\cdot|M|}{nc})$ for run-time, and $O(|D|\cdot|M|)$ for communication. *nc* represents the total number of cores used to distribute the complexity burden.
- Selection process (line 24 – Algorithm 3): a single evolutionary selection process is executed on each partition. The overall computational complexity is linear: $O(ne\cdot\frac{|D|\cdot|BP|\cdot df}{nc})$, and it is mainly bounded by Naive Bayes's complexity ($O(|D|\cdot|BP|)$). The procedure communicates $O(|BP|)$ integer data.
- Main algorithm (Algorithm 1): all sequential operations here are linear ($O(|BP|)$), as well as the communication processes between the nodes and the master node ($O(|BP|)$).

Notice that, in most of cases, the number of boundary points to be processed and communicated is much lower than the number of original points according to the Table 6 (#Pt.). This fact allows us to say that our algorithm can perform efficiently in many large-scale problems.

5. Experimental framework and results

This section describes the experimental framework carried out and analyzes the results derived from these experiments. The aim of these experiments is to prove the benefit derived from using our discretization solution. DMDLP, an distributed discretizer based on entropy minimization, is included in the experiments for comparison purposes.

5.1. Datasets and methods

In these experiments, we have used four large-scale classification datasets as benchmarks. The largest dataset in our framework is ECBDL14. This dataset was used as benchmark at the international conference GECCO-2014, in an classification competition for Big Data. This consists of 32 million instances with a high imbalance ratio: 98% of negative instances. To equalize both classes, the MapReduce version of the Random OverSampling (ROS) technique [36] was used to replicate the minority class (henceforth called ECBDL14R). This version has been used in the experiments instead of the original one.

From the LibSVM dataset repository [37], the dataset *epsilon* has been used as example of artificially created (and noisy) dataset with many features and boundary points. The rest of datasets (*higgs* and *susy*) have been taken from the UCI Machine Learning Repository [38]. *susy* is also an imbalanced problem with a ratio of 34%. All datasets presented in this section are two-class problems.

Table 2 gives a short description of these datasets. For each one, the number of examples for training and test (#Train Ex., #Test Ex.), the total number of attributes (#Atts.), and the number of classes (#Cl) are shown. In order to reduce the number of candidate points, all data has been rounded up to four decimal places. It affects to problems like higgs or epsilon where the number of decimal places is large enough.

Naïve Bayes has been elected as the reference classifier to assess the quality of solutions. Namely, the distributed version of Naïve Bayes in MLlib [39] have been chosen for the experiments. In Table 3, the recommended parameters (according to their authors' specification)

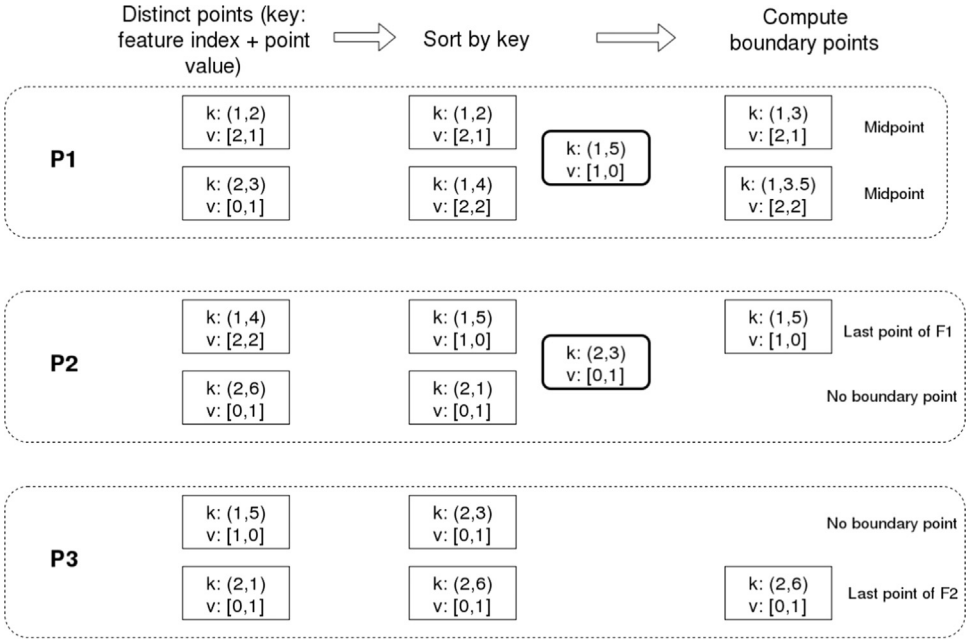

Fig. 3. Distributed computation of boundary points. P represents the partitions. The points broadcasted have been highlighted in bold.

Table 2
Summary description of datasets. For each one, the number of examples in training and test (#Train Ex., #Test Ex.), the total number of features (#Atts.), and the number of continuous features (#Cont.) are shown.

Dataset	#Train Ex.	#Test Ex.	#Atts.	#Cont.
ECBDL14R	65,003,913	2,897,917	630	539
higgs	8,800,000	2,200,000	28	28
susy	4,000,000	1,000,000	18	18
epsilon	400,000	100,000	2000	2000

Table 3
Parameters of the algorithms used.

Method	Parameters
DEMD	$\alpha = 0.5$, $sr = 1.0$, $vp = \{0.25, 0.5, 1.0\}$, $ne = 10,000$
Distributed MDLP	Max cut points = 15, max by partition = 100,000
Naïve Bayes	Lambda = 1.0

for this algorithm[3] is shown.

The distributed discretizer DMDLP [4] has been included in the experiment for comparison purposes. The parameter values for both discretizers are also defined in Table 3. For some special cases, some modifications to these parameters have been introduced, as explained in Section 5.4. For all experiments, five executions for each pair method-dataset have been launched to assess the quality of the solutions generated by our non-deterministic algorithm. The details of all runs are reported in Appendix A.

As evaluation measures, three standard metrics have been used to assess the performance of the discretizers and the quality of the subsequent solutions. The classification accuracy and the Area Under Curve Receiver Operating Characteristic (AUC-ROC) have been used for quality evaluation of test set. The overall discretization time has also been used to measure the quickness of discretizers.

A cluster of machines of twenty computing nodes and a master node was used to accomplish the experiments. All nodes hold the following features: 2 processors x Intel Xeon CPU E5-2620, 6 cores per processor, 2.00 GHz, 15 MB cache, QDR InfiniBand Network (40 Gbps), 2 TB HDD, 64 GB RAM. The software installed on these machines was the following: Hadoop 2.5.0-cdh5.3.1 from Cloudera's open-source Apache Hadoop distribution,[4] Apache Spark and MLlib 1.5.0, 460 cores (23 cores/node), 960 RAM GB (48 GB/node). The source code of DEMD, designed to be integrated in MLlib, can be downloaded from the correspondent author' GitHub account.[5]

5.2. Analysis of classification performance

In this section, the classification performance of our discretizer is evaluated against two classifiers and several huge datasets. The discretization schemes generated by our solution and another alternative are used as a preprocessing step before the classification phase.

In Table 4, the average classification results on test after applying Naïve Bayes are shown. Before classifying, the datasets have been discretized in a preprocessing stage using both discretizers. As can be seen in this table, the accuracy results yielded by our method outperforms those yielded by DMDLP in 4 out of 5 cases. This is specially remarkable for *ECBDL14R* (the biggest dataset), with a difference of several tenths. Notice that a slight improvement in accuracy in these large-scale problems could imply a high number of instances is correctly classified (1300 instances for susy).

Likewise, we have measured the impact of discretization in classifying two imbalanced datasets: *ECBDL14R* and *susy*. Table 5 shows the AUC results on the test set. No remarkable difference between both methods can be seen in this table, but only a slight advantage for DMDLP.

Beyond the improvement in accuracy, our solution has shown to yield simpler discretization schemes, with far lower number of points. These simpler solutions, apart from being much more understandable for experts, also have a positive impact on the learning process (from

[3] https://spark.apache.org/docs/latest/api/scala/index.html.
[4] http://www.cloudera.com/content/cloudera/en/documentation/cdh5/v5-0-0/CDH5-homepage.html.
[5] https://github.com/sramirez/.

Table 4
Classification test accuracy by discretizer and dataset.

Method		ECBDL14R	higgs	epsilon	susy
DEMD - 0.25	*Avg*	0.6912	**0.5960**	0.6734	0.7133
	Std-Dev	0.0127	0.0131	0.0106	0.0077
DEMD - 0.5	*Avg*	0.7215	0.5680	0.6752	0.7175
	Std-Dev	0.0141	0.0256	0.0098	0.0102
DEMD - 1.0	*Avg*	**0.7500**	0.5630	0.6718	**0.7406**
	Std-Dev	0.0107	0.0261	0.0185	0.0033
DMDLP	*Avg*	0.7272	0.5933	**0.7047**	0.7393
	Std-Dev	0.00	0.00	0.00	0.00

Table 5
Test AUC by discretizer and dataset.

Method		ECBDL14R	susy
DEMD - 0.25	*Avg*	0.5113	0.7006
	Std-Dev	0.0007	0.0076
DEMD - 0.5	*Avg*	0.5128	0.7048
	Std-Dev	0.0007	0.0060
DEMD - 1.0	*Avg*	**0.5143**	**0.7157**
	Std-Dev	0.0006	0.0032
DMDLP	*Avg*	0.5131	0.7126
	Std-Dev	0.00	0.00

Table 6
Number of cut points generated by discretizer and dataset. The best solution for each dataset according to Naïve Bayes is highlighted in bold, whereas the best one for DEMD is highlighted in italic.

Method	ECBDL14R	higgs	epsilon	susy
DEMD - 0.25	210	**248**	240	247
DEMD - 0.5	462	496	**495**	494
DEMD - 1.0	**959**	1000	990	988
DMDLP	8624	410	1718	**267**

Table 7
Discretization time by discretizer and dataset (in seconds).

Method		ECBDL14R	higgs	epsilon	susy
DEMD - 0.25	*Avg*	1178.70	733.26	1850.80	268.84
	Std-Dev	25.66	60.54	15.95	3.42
DEMD - 0.5	*Avg*	1181.42	771.75	1858.94	271.20
	Std-Dev	18.23	74.46	13.21	7.23
DEMD - 1.0	*Avg*	1169.80	764.68	1861.83	271.22
	Std-Dev	20.97	51.01	22.92	6.26
DMDLP	*Avg*	**975.80**	**36.01**	**117.44**	**23.45**
	Std-Dev	0.00	0.00	0.00	0.00

Table 8
Information derived from data and chromosome partitioning tasks. For each dataset, the original chromosome size (#Pt.), the computed multivariate factor (Mvf.), the maximum feature size (Mfs.), the final chunk size (Cs.), and the number chunks generated (#Ch.) are shown.

Dataset	#Pt.	Mvf.	Mfs.	Cs.	#Ch.
ECBDL14R	41,937	10.82	985	985	42
higgs	514,524	52,96	59,214	59,214	8
susy	573,792	33.71	42,046	42,046	13
epsilon	3,013,813	1.00	2555	2555	460
epsilon[*]	3,013,813	30.00	76,650	76,650	15

[*] Epsilon without uf multivariate factor set to 30.

both time and accuracy points). This is essential in Big Data environments where efficiency and simplicity are a plus. Table 6 illustrates the simplicity of these solutions. The results prove that the most accurate solutions for DEMD are also those with a lower number of points, except for *susy*.

5.3. Analysis of efficiency

Another aspect when evaluating discretizers is the efficiency in generating the discretization schemes. This is specially important in Big Data environments, where the quickness is an important factor. This section presents a comparison between DEMD and DMDLP, in terms of time used to obtain the discretization model.

Table 7 illustrates this comparison by presenting the average time results for both algorithms. For all cases, DMDLP performs much faster than our method due to its greedy iterative nature. Nevertheless, our solution offers competitive results. All of them far below a limit of one hour, which are quite reasonable in Big Data analytics.

5.4. Case study: explosive growth of chromosomes and use of sampling

An overwhelming number of candidate points and instances to evaluate are the two most important problems when dealing with large-scale datasets and EAs. This section aims at showing how our approach can be tuned to deal with these problems, quite common in some big datasets.

The number of candidates points to be evaluated straightly determines the chromosome size that has to be managed by the EA. In classical learning problems, this size can range to 15,000, as we verified in a our study [17] where a long list of UCI datasets was analyzed. Big datasets though presents a complexity (number of points) much higher than presented in small/medium datasets, as shown in Table 8. Despite some preprocessing stage has been applied to these datasets (as presented in 5.1), the chromosome size can go to several millions of genes. This is the case of *epsilon*, in which the EA starts with 3,013,813 points to evaluate.

Table 8 shows the value of the factors and variables which are implied in the data partitioning and the creation of point chunks. It is specially remarkable the *epsilon* case where the number of chunks corresponds with the number of data partitions (460). This case represents the simplest case of voting so that all chunks will be evaluated by a single data partition, which implies a clear degradation on the overall performance of the discretizer. In the experiments, the multivariate factor variable *uf* was changed for *epsilon* in order to cope with this problem (marked with an asterisk in the table). *uf* was then established to 30, a similar value to that present in *susy* (the closest problem in number of points to *epsilon*).

EAs are also affected by the sample size. In our case, the wrapper classifier used in our EA needs to evaluate each solution using the complete set of instances. Even after partitioning the points into chunks, the size of chromosomes remains quite complex for the fitness evaluation. In order to make feasible this evaluation for big datasets (like ECBDL14R), some simplification techniques could be applied to alleviate this complexity. One of them is the stratified sampling of instances. In our algorithm, this technique is applied just after computing the candidate points so as to only use this sample to evaluate solutions.

According to the previous idea, a stratified sampling was applied on ECBDL14R, the biggest dataset in terms of number of instances. The

sampling rate sr was then established to 0.1 in order to equalize the performance of our solutions for all datasets used (see Table 7). Furthermore, the accuracy results confirms that even using a reduced sample of instances, there is a considerable improvement in classification results.

6. Conclusions

In this paper, we have presented DEMD, a distributed multivariate discretization algorithm for Big Data based on evolutionary optimization under Apache Spark. Our solution is aimed at optimizing the cut points selection problem by selecting accurate and simple solutions. In this version, a new system of cross-evaluation between partitions of instances and points has been introduced. Despite its non-deterministic nature, this kind of evaluation offers promising discretization schemes.

The experimental results obtained on big datasets (up to $O(10^7)$ instances and $O(10^4)$ features) have shown the improvement on accuracy and simplicity when using DEMD. Our approach also allows to tune the simplicity/accuracy rate of the generated solutions using several parameters.

Our future work will concentrate on showing that evolutionary computation can also be useful in dealing with other Big Data preprocessing tasks [12], such as feature or instance selection. We will also envision that our approach can be adapted to the streaming environment where discretization schemes evolve over time, and concept drifts might affect them [40].

Acknowledgements

This work is supported by the Spanish National Research Project TIN2014-57251-P, the Foundation BBVA project 75/2016 BigDaPTOOLS, and the Andalusian Research Plan P11-TIC-7765. S. Ramírez-Gallego holds a FPU scholarship from the Spanish Ministry of Education and Science (FPU13/00047).

Appendix A. Detailed classification results on test

In this section, we present the detailed results (by execution) derived from test classification. Tables A.9, A.10, A.11, and A.12 show the acurracy results for all datasets, whereas Tables A.13 and A.14 show the results on AUC for the two imbalanced problems used in the experiments (*ECBDL14R* and *susy*).

Table A.9
Test accuracy obtained for ECBDL14R.

Method - *vp*	*Ex. #1*	*Ex. #2*	*Ex. #3*	*Ex. #4*	*Ex. #5*	**Avg**	**Std-Dev**
DEMD - 0.25	0.6923	0.696	0.6832	0.6757	0.7089	0.6912	0.0127
DEMD - 0.5	0.7172	0.7262	0.7434	0.7068	0.7137	0.7215	0.0141
DEMD - 1.0	0.7456	0.7619	0.7483	0.7587	0.7353	**0.7500**	**0.0107**
DMDLP	0.7272	–	–	–	–	0.7272	0.0000

Table A.10
Test accuracy obtained for higgs.

Method - *vp*	*Ex. #1*	*Ex. #2*	*Ex. #3*	*Ex. #4*	*Ex. #5*	**Avg**	**Std-Dev**
DEMD - 0.25	0.6172	0.587	0.5926	0.5844	0.5987	**0.5960**	**0.0131**
DEMD - 0.5	0.5741	0.5891	0.5614	0.5884	0.5271	0.5680	0.0256
DEMD - 1.0	0.5249	0.5507	0.5705	0.5927	0.5762	0.5630	0.0261
DMDLP	0.5933	–	–	–	–	0.5933	0.0000

Table A.11
Test accuracy obtained for epsilon.

Method - *vp*	*Ex. #1*	*Ex. #2*	*Ex. #3*	*Ex. #4*	*Ex. #5*	**Avg**	**Std-Dev**
DEMD - 0.25	0.6599	0.6871	0.6668	0.6737	0.6797	0.6734	0.0106
DEMD - 0.5	0.6679	0.6646	0.6837	0.6728	0.6870	0.6752	0.0098
DEMD - 1.0	0.6403	0.6839	0.681	0.6838	0.6698	0.6718	0.0185
DMDLP	0.7047	–	–	–	–	**0.7047**	**0.0000**

Table A.12
Test accuracy obtained for susy.

Method - *vp*	*Ex. #1*	*Ex. #2*	*Ex. #3*	*Ex. #4*	*Ex. #5*	**Avg**	**Std-Dev**
DEMD - 0.25	0.7188	0.7160	0.7004	0.7121	0.7192	0.7133	0.0077
DEMD - 0.5	0.7322	0.7234	0.7086	0.7086	0.7147	0.7175	0.0102
DEMD - 1.0	0.7413	0.7445	0.7406	0.7354	0.7411	**0.7406**	**0.0033**
DMDLP	0.7393	–	–	–	–	0.7393	0.0000

Table A.13
Test AUC obtained for ECBDL14R.

Method - *vp*	*Ex. #1*	*Ex. #2*	*Ex. #3*	*Ex. #4*	*Ex. #5*	**Avg**	**Std-Dev**
DEMD - 0.25	0,5112	0,5118	0,5109	0,5103	0,5121	0,5113	0,0007
DEMD - 0.5	0,5125	0,513	0,5139	0,5122	0,5122	0,5128	0,0007
DEMD - 1.0	0,5142	0,515	0,5146	0,5143	0,5133	**0,5143**	0,0006
DMDLP	0.5131	–	–	–	–	0.5131	0.0000

Table A.14
Test AUC obtained for susy.

Method - *vp*	*Ex. #1*	*Ex. #2*	*Ex. #3*	*Ex. #4*	*Ex. #5*	**Avg**	**Std-Dev**
DEMD - 0.25	0,7072	0.7068	0.6892	0.7028	0.6968	0.7006	0.0076
DEMD - 0.5	0.7110	0.7065	0.6954	0.7032	0.7081	0.7048	0.0060
DEMD - 1.0	0.7152	0.7212	0.7128	0.7148	0.7144	**0.7157**	0.0032
DMDLP	0.7126	–	–	–	–	0.7126	0.0000

References

[1] S. García, J. Luengo, F. Herrera, Data Preprocessing in Data Mining, Springer, 2015.
[2] S. García, J. Luengo, F. Herrera, Tutorial on practical tips of the most influential data preprocessing algorithms in data mining, Knowl.-Based Syst. 98 (2016) 1–29.
[3] S. García, J. Luengo, J.A. Sáez, V. López, F. Herrera, A survey of discretization techniques: taxonomy and empirical analysis in supervised learning, IEEE Trans. Knowl. Data Eng. 25 (4) (2013) 734–750.
[4] S. Ramírez-Gallego, S. García, H. Mouriño Talín, D. Martínez-Rego, V. Bólon-Canedo, A. Alonso-Betanzos, J.M. Benítez, F. Herrera, Data discretization: taxonomy and big data challenge, Wiley Interdiscip. Rev.: Data Min. Knowl. Discov. 6 (1) (2016) 5–21.
[5] C.C. Aggarwal, Data Mining: The Textbook, Springer, 2015.
[6] M. Minelli, M. Chambers, A. Dhiraj, Big Data, Big Analytics: Emerging Business Intelligence and Analytic Trends For Today's Businesses, John Wiley and Sons, 2013.
[7] V. Mayer-Schnberger, K. Cukier, Big Data: A Revolution That Will Transform How We Live, Work and Think, John Murray Publishers, 2013.
[8] S. García, S. Ramírez-Gallego, J. Luengo, J.M. Benítez, F. Herrera, Big data preprocessing: methods and prospects, Big Data Anal. 1 (1) (2016) 9.
[9] A. Fernández, S. del Río, V. López, A. Bawakid, M.J. del Jesús, J.M. Benítez, F. Herrera, Big data with cloud computing: an insight on the computing environment, mapreduce, and programming frameworks, Wiley Interdiscip. Rewiews: Data Min. Knowl. Discov. 4 (5) (2014) 380–409.
[10] N. Xiong, D. Molina, M.L. Ortiz, F. Herrera, A walk into metaheuristics for engineering optimization: principles, methods and recent trends, International, J. Comput. Intell. Syst. 8 (2015) 606–636.
[11] A. LaTorre, S. Muelas, J.-M. Peña, A comprehensive comparison of large scale global optimizers, Inf. Sci. 316 (2015) 517–549.
[12] S. Cheng, B. Liu, Y. Shi, Y. Jin, B. Li, Evolutionary computation and big data: Key challenges and future directions, in: Proceedings of the Data Mining and Big Data, First International Conference, DMBD 2016, Bali, Indonesia, June 25–30, 2016, pp. 3–14.
[13] A. Fernández, S. García, J. Luengo, E. Bernado-Mansilla, F. Herrera, Genetics-based machine learning for rule induction: state of the art, taxonomy, and comparative study, IEEE Trans. Evolut. Comput. 14 (6) (2010) 913–941.
[14] A. Fernández, V. López, M.J. del Jesús, F. Herrera, Revisiting evolutionary fuzzy systems: taxonomy, applications, new trends and challenges, Knowl. Based Syst. 80 (2015) 109–121.
[15] S.J. Nanda, G. Panda, A survey on nature inspired metaheuristic algorithms for partitional clustering, Swarm Evolut. Comput. 16 (2014) 1–18.
[16] A. Mukhopadhyay, U. Maulik, S. Bandyopadhyay, C.A.C. Coello, A survey of multiobjective evolutionary algorithms for data mining: Part I, IEEE Trans. Evolut. Comput. 18 (1) (2014) 4–19.
[17] S. Ramírez-Gallego, S. García, J.M. Benítez, F. Herrera, Multivariate discretization based on evolutionary cut points selection for classification, IEEE Trans. Cybern. 46 (3) (2016) 595–608.
[18] N. Sreeja, A. Sankar, A hierarchical heterogeneous ant colony optimization based approach for efficient action rule mining, Swarm Evolut. Comput. 29 (2016) 1–12.
[19] P. Mohapatra, S. Chakravarty, P. Dash, Microarray medical data classification using kernel ridge regression and modified cat swarm optimization based gene selection system, Swarm Evolut. Comput. 28 (2016) 144–160.
[20] W. Sheng, S. Chen, M. Sheng, G. Xiao, J. Mao, Y. Zheng, Adaptive multi-subpopulation competition and multiniche crowding-based memetic algorithm for automatic data clustering, IEEE Trans. Evolut. Comput. 20 (6) (2016) 838–858.
[21] S. Nebti, A. Boukerram, Swarm intelligence inspired classifiers for facial recognition, Swarm and Evolutionary Computation 23 (2017) 150–166. http://dx.doi.org/10.1016/j.swevo.2016.07.001 (In press).
[22] Apache Spark: Lightning-fast Cluster Computing, Apache spark, 2016. (Online; Accessed December 2016). ⟨https://spark.apache.org/⟩.
[23] M. Beyer, D. Laney, 3D Data Management: Controlling Data Volume, Velocity and Variety, 2001. ⟨http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf⟩.
[24] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, in: Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation, vol. 6 of OSDI'04, 2004, pp. 10–10.
[25] T. White, Hadoop The Definitive Guide, O'Reilly Media, Inc., 2012.
[26] Apache Hadoop Project, Apache Hadoop, 2016. (Online; Accessed December 2016). ⟨http://hadoop.apache.org/⟩.
[27] J. Lin, Mapreduce is good enough? If all you have is a hammer, throw away everything that's not a nail!, Big Data 1 (1) (2013) 28–37.
[28] M. Hamstra, H. Karau, M. Zaharia, A. Konwinski, P. Wendell, Learning Spark: Lightning-Fast Big Data Analytics, O'Reilly Media, Incorporated, 2015.
[29] B.S. Chlebus, S.H. Nguyen, On finding optimal discretizations for two attributes, in: Proceedings of the First International Conference on Rough Sets and Current Trends in Computing, RSCTC '98, 1998, pp. 537–544.

[30] T. Elomaa, J. Rousu, General and efficient multisplitting of numerical attributes, Mach. Learn. 36 (1999) 201–244.
[31] L.J. Eshelman, The CHC adaptive search algorithm: how to have safe search when engaging in nontraditional genetic recombination, in: FOGA, 1990, pp. 265–283.
[32] M. Dash, H. Liu, Consistency-based search in feature selection, Artif. Intell. 151 (1–2) (2003) 155–176.
[33] K.J. Cios, W. Pedrycz, R.W. Swiniarski, L.A. Kurgan, Data Mining: A Knowledge Discovery Approach, Springer, 2007.
[34] J.R. Quinlan, C4.5: Programs for Machine Learning, Morgan Kaufmann Publishers Inc, 1993.
[35] S.J. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, 2nd ed., Pearson Education, 2003.
[36] S. Río, V. López, J. Benítez, F. Herrera, On the use of mapreduce for imbalanced big data using random forest, Inf. Sci. 285 (2014) 112–137.
[37] C.-C. Chang, C.-J. Lin, LIBSVM: a library for support vector machines, ACM Trans. Intell. Syst. Technol. 2 (2011) 27:1–27:27 (datasets available at ⟨http://www.csie.ntu.edu.tw/cjlin/libsvmtools/datasets/⟩).
[38] K. Bache, M. Lichman, UCI machine learning repository, 2013. ⟨http://archive.ics.uci.edu/ml⟩.
[39] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M.J. Franklin, R. Zadeh, M. Zaharia, A. Talwalkar, Mllib: machine learning in apache spark, J. Mach. Learn. Res. 17 (34) (2016) 1–7.
[40] S. Ramírez-Gallego, B. Krawczyk, S. García, M. Woźniak, F. Herrera, A survey on data preprocessing for data stream mining: current status and future directions, Neurocomputing 239 (2017) 39–57.