Distributed Entropy Minimization Discretizer for Big Data Analysis under Apache Spark

Sergio Ramírez-Gallego*, Salvador García* Héctor Mouriño-Talín†, David Martínez-Rego†‡,
Verónica Bolón-Canedo†, Amparo Alonso-Betanzos†, José Manuel Benítez* and Francisco Herrera*
*Department of Computer Science and Artificial Intelligence, CITIC-UGR, University of Granada, 18071 Granada, Spain
Emails: sramirez@decsai.ugr.es, salvagl@decsai.ugr.es, j.m.benitez@decsai.ugr.es, herrera@decsai.ugr.es
†Department of Computer Science, University of A Coruña, 15071 A Coruña, Spain.
Emails: h.mtalin@udc.es, dmartinez@udc.es, veronica.bolon@udc.es, ciamparo@udc.es
‡Department of Computer Science, University College London, WC1E 6BT London, United Kingdom.

***Abstract*—The astonishing rate of data generation on the Internet nowadays has caused that many classical knowledge extraction techniques have become obsolete. Data reduction techniques are required in order to reduce the complexity order held by these techniques. Among reduction techniques, discretization is one of the most important tasks in data mining process, aimed at simplifying and reducing continuous-valued data in large datasets. In spite of the great interest in this reduction mechanism, only a few simple discretization techniques have been implemented in the literature for Big Data.**

Thereby we propose a distributed implementation of the entropy minimization discretizer proposed by Fayyad and Irani using Apache Spark platform. Our solution goes beyond a simple parallelization, transforming the iterativity yielded by the original proposal in a single-step computation. Experimental results on two large-scale datasets show that our solution is able to improve the classification accuracy as well as boosting the underlying learning process.

I. Introduction

Many real-world areas from science and industry are generating massive amounts of data nowadays, thanks to the fast development of data storage and networking. Only within the period between 2010 and 2012, 90 percent of the data in the world were produced at a rate of 2.5 quintillion of bytes per day according to the estimations of IBM [1]. This phenomenon has led to the pressing needs for new technologies and methods, capable of dealing with this emerging era of humongous data. Big Data can be defined as data beyond the storing and processing capacity of traditional systems [2]. Extracting valuable knowledge from such data is taking great interest in data analytics research as it supposed a gain of critical insights for companies and institutions.

Many knowledge extraction techniques have become obsolete since they were not conceived for dealing with such amounts of data. Data reduction techniques can be applied in order to decrease the original data and even to improve the learning performance of many algorithms [3]. Discretization, as an important part of data reduction, has received increasing attention in recent years [4], [5]. The objective in discretization is to simplify and reduce data by transforming continuous-valued attributes into discrete ones using a finite number of intervals or discrete values [6], [7]. Among its main benefits, discretization causes in learning methods remarkable improvements in learning speed and accuracy. Although most of real-world problems often imply numerical attributes, many algorithms can only handle categorical attributes, for example some feature selection methods (which are relevant in the Big Data picture). Besides, some decision trees-based algorithms produce shorter, more compact, and accurate results using discrete values [6], [8].

Common data reduction methods are not expected to scale well when managing huge data -both in terms of features and instances- so that its execution can be undermined or even become impracticable. Distributed techniques and frameworks have appeared along with the issue of Big Data. MapReduce [9] and its open-source version Apache Hadoop [10], [11] were the first distributed programming techniques to face this problem. Recently, several distributed tools have emerged as consequence of the advent of Big Data. Apache Spark [12], [13] is one of these new frameworks, designed as a fast and general engine for large-scale data processing based on in-memory computation. Through this Spark's ability, it is possible to speed up iterative processes present in many Machine Learning (ML) problems. Because of that, this tool has become especially popular among Machine Learning researchers and business experts.

Similarly, several ML libraries for Big Data have appeared as support for this task. The first one was Mahout [14] (as part of Hadoop), subsequently followed by MLlib [15] which is part of Spark project [13]. In spite of many state-of-the-art ML algorithms have been implemented in MLlib, is not the case of discretization algorithms yet.

In order to fill this gap, we propose a distributed version of the entropy minimization discretizer proposed by Fayyad and Irani in [16] using Apache Spark, which is based on the Minimum Description Length Principle (MDLP). Our main goal is to prove that well-knowm discretization algorithms as MDL-based discretizer (henceforth called MDLP) can be parallelized in these frameworks, providing good discretization solutions for Big Data analytics. Moreover, we have transformed the iterativity yielded by the original proposal in a single-step computation. Notice that this new version for distributed environments has involved a deep restructuring of the original proposal and a challenge for the authors. Finally, to

978-1-4673-7952-6/15 $31.00 © 2015 IEEE
DOI 10.1109/Trustcom-BigDataSe-ISPA.2015.559

IEEE computer society

demonstrate the effectiveness of our framework, we perform an experimental evaluation with two large datasets, namely, ECBDL14 and epsilon.

The remainder of this paper is organized as follows: Section II outlines the main concepts about MDLP discretizer and Big Data solutions. Section III explains our distributed approach based on entropy minimization for Big Data. Section IV describes the experiments carried out to demonstrate the effectiveness of this proposal. Finally, Section V establishes the main conclusions of the work carried out.

II. Background

In this section we briefly explain the MDLP algorithm used as reference in our distributed adaptation, as well as basic concepts about discretization. Finally, we present the topic of Big Data and distributed frameworks that cope with this problem.

A. Discretization: An Entropy Minimization Approach

In supervised learning, and specifically in classification, the problem of discretization can be defined as follows. Assuming a data set S consisting of N examples, M attributes and c class labels, a discretization scheme D_A would exist on the continuous attribute $A \in M$, which partitions this attribute into k discrete and disjoint intervals: $\{[d_0, d_1], (d_1, d_2], \ldots, (d_{k_A-1}, d_{k_A}]\}$, where d_0 and d_{k_A}, respectively, are the minimum and maximal value, and $P_A = \{d_1, d_2, \ldots, d_{k_A-1}\}$ represents the set of cut points of A in ascending order.

The discretization problem consists of finding a set boundary points that yield the best result according to a given evaluation measure (e.g.: inconsistency or information gain). Achieving the optimal discretization scheme is NP-complete [17], where the search space of this problem is determined by the number of *candidate cut points*, namely, all distinct values in the dataset for each attribute. As this problem can become truly expensive when the number of candidates increases, a possible optimization would be to use a reduced subset of points, only formed by the *boundary points* in the whole set. This assumption is also supported by the fact that boundary points typically form the optimal intervals for most of the evaluation measures used in the literature [18].

Let $A(e)$ denote the value for attribute A in the example e. A boundary point $b \in Dom(A)$ can be defined as the midpoint value between $A(u)$ and $A(v)$, assuming that in the sorted collection of points in A, there exist two examples $u, v \in S$ with different class labels, such that $A(u) < b < A(v)$; and there does not exist other example $w \in S$ such that $A(u) < A(w) < A(v)$. The set of boundary points for attribute A is defined as B_A.

MDLP discretizer [16] implements this optimization so as to speed up the underlying discretization process. However, this method also introduces other important improvements. One of them is related to the number of cut points to derive in each iteration. In contrast to discretizers like ID3 [19], the authors proposed a multi-interval extraction of points demonstrating that better classification models -both in error rate and simplicity- are yielded by using these schemes.

As quality measure, MDLP discretizer computes the class entropy of the partitioning schemes. The objective is to minimize this measure to obtain the best cut decision. Let b_α be a boundary point to evaluate, $S_1 \subset S$ be a subset where $\forall a' \in S_1,\ A(a') \leq b_\alpha$, and S_2 be equal to $S - S_1$. The class information entropy yielded by a given binary partitioning can be expressed as:

$$EP(A, b_\alpha, S) = \frac{|S_1|}{|S|} E(S_1) + \frac{|S_2|}{|S|} E(S_2), \tag{1}$$

where E represents the class entropy [1] of a given subset following the Shannon's definitions [20].

Finally, a decision criterion is defined in order to control when to stop the partitioning process. The use of MDLP is a inference method that provides a criterion for the selection of models avoiding overfitting. This is based on the idea of the more the model is able to compress the data, the more regularities it has found and thus, the more the model is capable of learning. For discretization, this principle is used as decision criterion that allows us to decide whether or not to partition. Thus, a cut point b_α will be applied iff:

$$G(A, b_\alpha, S) > \frac{\log_2(N-1)}{N} + \frac{\Delta(A, b_\alpha, S)}{N}, \tag{2}$$

where $\Delta(A, b_\alpha, S) = log_2(3^c) - [cE(S) - c_1 E(S_1) - c_2 E(S_2)]$, c_1 and c_2 the number of class labels in S_1 and S_2, respectively; and $G(A, b_\alpha, S) = E(S) - EP(A, b_\alpha, S)$

B. Big Data: Distributed Models and Frameworks

The ever-growing generation of data on the Internet is heading us to managing huge collections of data using classical data analytics solutions. Exceptional paradigms and algorithms are thus needed to efficiently process these collections of data so as to obtain valuable information, becoming this problem one of the most challenging tasks in Big Data analytics.

Gartner [21] introduced the popular denomination of Big Data and the 3Vs terms that define it as high volume, velocity and variety information that require a new large-scale processing. This list was then extended with 2 additional terms. All of them are described in the followings:

- **Volume**: the massive amount of data that is produced every day is still exponentially growing (from terabytes to exabytes).
- **Velocity**: data needs to be loaded, analyzed and stored as quickly as possible.
- **Variety**: data come in many formats and representations. The sources are also highly different between them: text, image, multimedia, etc.
- **Veracity**: the quality of data to process is also an important factor. The Internet is fulfilled of missing, incomplete, ambiguous, and sparse data.

[1] Logarithm in base 2 is used in this function

- **Value**: extracting value from data is also established as a relevant objective in big analytics. Not only is important the performance, but also to extract valuable information from data.

The unsuitability of many knowledge extraction algorithms in the Big Data field has made that new methods are developed to manage such amounts of data effectively and in tolerable speed.

1) MapReduce Model and Other Distributed Frameworks: The MapReduce framework [9], designed by Google in 2003, is currently one of the most relevant tools in Big Data analytics. It was aimed at processing and generating large-scale datasets, automatically processed in an extremely distributed fashion through several machines. This framework is in charge of partitioning and distributing the data, fault-tolerance, job scheduling and networking; only leaving the responsibility of launching their tasks to the final users[2].

The MapReduce model defines two primitives to work with distributed data: **Map** and **Reduce**. These two primitives imply two stages in the distributed process, which we describe below. In the beginning, the master node breaks up the dataset into several splits, distributing them across the cluster for a parallel processing. Each node then hosts several Map threads that transform the generated key-value pairs in a set of intermediate pairs. After all Map tasks have finished, these notify its ending to the master node, starting the Reduce phase. Here, the master node distributes the matching pairs across the nodes according to a key-based partitioning scheme, combining those coincident pairs so as to form the final output.

Briefly, the Map function takes <key, value> pairs as input and outputs a collection of intermediate <key, value> pairs. The user-defined Map function needs to be specified following the key-value scheme specified below:

$$Map(< key1, value1 >) \rightarrow list(< key2, value2 >) \quad (3)$$

The combined result (grouped by key) is then distributed across the cluster and processed by the Reduce phase. Here, a Reduction function is applied to each list value, producing a single output value as follows:

$$Reduce(< key2, list(val2) >) \rightarrow \quad < key2, val3 > \quad (4)$$

Apache Hadoop [10], [11] is presented as the most popular open-source implementation of MapReduce for large-scale processing. Despite its popularity, Hadoop presents some important weaknesses, such as a poor performance on iterative and online computing, a poor inter-communication capability or inadequacy for in-memory computation, among others [23].

In the Hadoop Ecosystem appears Apache Spark [12], [13]. This novel framework is presented as a revolutionary tool capable of performing even faster large-scale processing than Hadoop through in-memory primitives, demonstrating to perform 100 times faster than Hadoop for certain cases. Spark allows users to persist data into memory and to load them rapidly, making this framework a leading tool for iterative and online processing.

Spark is built on distributed data structures called Resilient Distributed Datasets (RDDs), which were designed as fault-tolerant collection of elements that can be operated in parallel by means of data partitioning. RDDs defines two types of operations: transformations, which transform existing data into a new dataset, and actions, which return a value to the driver program (hosted in the master node) after doing some computations. Furthermore, this tool provides a thorough set of programming primitives that allow the users to implement much more complex distributed programs than those implemented in Hadoop. Thanks to its generality capability, Spark is able to implement several distributed models like MapReduce or Pregel.

III. Distributed MDLP discretization

In [16], a discretization algorithm based on an information entropy minimization heuristic is presented. In this work, the authors prove that multi-interval extraction of points and the use of boundary points can improve the discretization process, both in efficiency and error rate. Here, we adapt this well-known algorithm for distributed environments, proving its discretization capability against real-world large problems.

One important point in this adaption is how to distribute the complexity of this algorithm across the cluster. This is mainly determined by two time-consuming operations: on the one hand, the sorting of candidate points, and, on the other hand, the evaluation of these points. The sorting operation conveys a $O(|A|log(|A|))$ complexity (assuming that all points in A are distinct), whereas the evaluation conveys a $O(|B_A|^2)$ complexity. In the worst case, it implies a complete evaluation of entropy for all points.

Note that the previous complexity is bounded for a single attribute. To avoid repeating the previous process on all attributes, we have designed our algorithm to sort and evaluate all points in a single step. Only when the number of boundary points in an attribute is higher than the maximum per partition, computation by feature is necessary (which is extremely rare according to our experiments).

In order to implement our method, we have used some extra primitives from Spark API. Spark primitives extend the idea of MapReduce to implement more complex operations on distributed data. Here, we outline those more relevant for our method [3]:

- $map\ partitions$: Similar to Map, this runs a function independently on each partition. For each partition, an iterator of tuples is fetched.
- $coalesce$: Reduce the number of partitions in a RDD to a given value.

[2]For a complete description of this model and other distributed models, please review [22].

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

- $sort_by_key$: A distributed version of sequential sorting. It sorts by key the values in each partition and then sorts the partitions.

Our proposal is divided in six algorithms and three sections as follows: Section III-A explains the main procedure, including most operations. Section III-B describes the selection of boundary points. Finally, Section III-C details the evaluation of final points using the MDLP criterion.

A. Main discretization procedure

Algorithm 1 explains the main procedure in our discretization algorithm. The algorithm calculates the minimum-entropy cut points by feature according to the MDLP criterion. It defines as input parameters the followings: the dataset, the indexes of features to discretize and the maximum number of points per partition.

Algorithm 1 Main discretization procedure

Input: S Data set
Input: M Feature indexes to discretize
Input: $maxcand$ Maximum number of candidates per partition
Output: Cut points by feature
1: $comb \leftarrow$
2: **map** $s \in S$
3: $\quad v \leftarrow zeros(k)$
4: $\quad v(c) \leftarrow 1$
5: $\quad$**for all** $A \in M$ **do**
6: $\qquad EMIT < (A, A(s)), v >$
7: $\quad$**end for**
8: **end map**
9: $distinct \leftarrow reduce(comb,\ sum_vectors)$
10: $sorted \leftarrow sort_by_key(distinct)$
11: $first \leftarrow first_by_part(sorted)$
12: $boundaries \leftarrow get_boundary_points(sorted, first)$
13: $boundaries \leftarrow$
14: **map** $b \in boundaries$
15: $\quad < (att, point,)q > \leftarrow b$
16: $\quad EMIT < (att, (point, q)) >$
17: **end map**
18: $(small, big) \leftarrow divide_attributes(boundaries, maxcand)$
19: $sthresholds \leftarrow select_thresholds(small, maxbins, maxcand)$
20: **for all** $att \in keys(big)$ **do**
21: $\quad bthresholds \leftarrow bthresholds + select_thresholds(big(att), maxbins, maxcand)$
22: **end for**
23: $return(union(bthresholds, sthresholds))$

The first step creates combinations from instances through a Map function in order to separate values by feature. It generates tuples with the value and the index for each feature as key and a class counter as value ($< (a, a(s)), v >$). Afterwards, the tuples are reduced using a function that aggregates all subsequent vectors with the same key, obtaining the class frequency for each distinct value in the dataset. The resulting tuples are sorted by key so that we obtain the complete list of distinct values ordered by feature index and feature value. This structure will be used later to evaluate all these points in a single step. The first point by partition is also calculated (line 11) for this process. Once such information is saved, the process of evaluating the boundary points can be started.

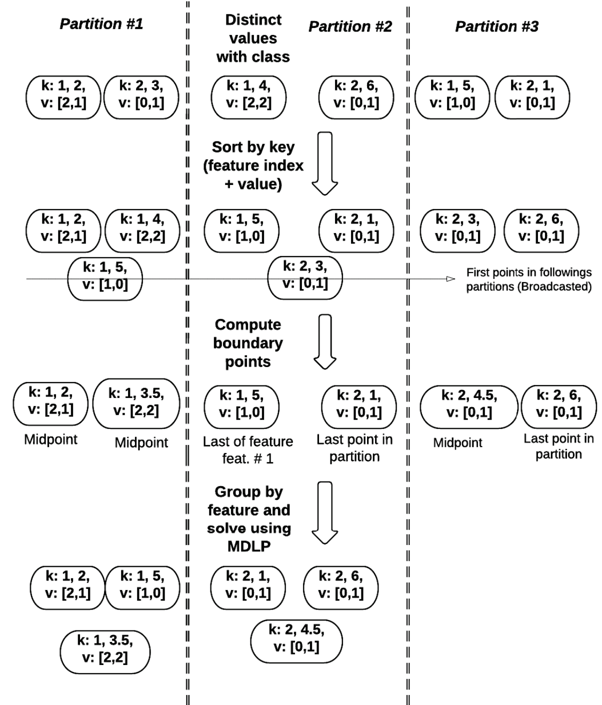

Fig. 1. Distributed MDLP flowchart

B. Boundary points selection

Algorithm 2 ($get_boundary_points$) describes the function in charge of selecting those points falling in the class borders. It executes an independent function on each partition in order to parallelize the selection process as much as possible so that a subset of tuples is fetched in each thread. The evaluation process is described as follows: for each instance, it evaluates if the feature index is distinct from the index of the previous point; if it is so, this emits a tuple with the last point as key and the accumulated class counter as value. This means that a new feature has appeared, saving the last point from the current feature as its last threshold. If the previous condition is not satisfied, the algorithm checks whether the current points is a boundary with respect to the previous point or not. If it is so, this emits a tuple with the midpoint between these points as key and the accumulated counter as value.

Finally, some evaluations are performed over the last point in the partition. This point is compared with the first point in the next partition to check whether there is a change in the feature index -emitting a tuple with the last point saved-, or not -emitting a tuple with the midpoint- (as described above). All tuples generated by the partition are then joined into a new mixed RDD of boundary points, which is returned to the main algorithm as $boundaries$.

In Algorithm 1 (line 14), the $boundaries$ variable is transformed by using a Map function, changing the previous key to a new key with a single value: the feature index ($< (att, (point, q)) >$). This is done to group the tuples by feature so that we can divide them in two groups according

Algorithm 2 Function to generate the boundary points *(get_boundary_points)*

Input: $points$ An RDD of tuples ($< (att, point), q >$), where att represents the feature index, $point$ the point to consider and q the class counter.
Input: $first$ A vector with all first elements by partition
Output: An RDD of points.

1: $boundaries \leftarrow$
2: **map partitions** $part \in points$
3: $\quad < (la, lp), lq > \leftarrow next(part)$
4: $\quad accq \leftarrow lq$
5: $\quad$ **for all** $< (a, p), q > \in part$ **do**
6: $\quad\quad$ **if** $a <> la$ **then**
7: $\quad\quad\quad EMIT < (la, lp), accq >$
8: $\quad\quad\quad accq \leftarrow ()$
9: $\quad\quad$ **else if** $is_boundary(q, lq)$ **then**
10: $\quad\quad\quad EMIT < (la, (p + lp)/2), accq >$
11: $\quad\quad\quad accq \leftarrow ()$
12: $\quad\quad$ **end if**
13: $\quad\quad < (la, lp), lq > \leftarrow < (a, p), q >$
14: $\quad\quad accq \leftarrow accq + q$
15: $\quad$ **end for**
16: $\quad index \leftarrow get_index(part)$
17: $\quad$ **if** $index < npartitions(points)$ **then**
18: $\quad\quad < (a, p), q > \leftarrow first(index + 1)$
19: $\quad\quad$ **if** $a <> la$ **then**
20: $\quad\quad\quad EMIT < (la, lp), accq >$
21: $\quad\quad$ **else**
22: $\quad\quad\quad EMIT < (la, (pt.point + lp)/2), accq >$
23: $\quad\quad$ **end if**
24: $\quad$ **else**
25: $\quad\quad EMIT < (la, lp), accq >$
26: $\quad$ **end if**
27: **end map**
28: $return(boundaries)$

to the total number of candidate points by feature. The $divide_attributes$ function is then aimed to divide the tuples in two groups (big and $small$) depending on the number of candidate points by feature. Features in each group will be treated differently. The previous function performs this separation according to whether the total number of points by feature exceeds a threshold ($maxcand$) or not. The tuples are now re-formatted as follows: ($< point, q >$).

C. MDLP evaluation

Features in each group are evaluated differently from we mentioned before. Small features are evaluated in a single step where each feature corresponds with a single partition, whereas big features are evaluated iteratively since each feature corresponds with a complete RDD with several partitions. The first option is obviously more efficient, however, the second case is less frequent due to the default value of $maxcand$ is rarely exceeded. Therefore, $maxcand$ value can affect the final performance of the algorithm. It is defined to a value of 10,000 points. In case of low performance (iterativity), it can be increased to 100,000 or even more.

In both cases, the $select_thresholds$ function is applied to evaluate and select the most relevant cut points by feature. For small features, a Map function is applied independently to each partition (each one represents a feature) ($arr_select_thresholds$). In case of big features, the process is more complex and each feature needs a complete iteration over a distributed set of points ($rdd_select_thresholds$). Algorithm 3 ($arr_select_thresholds$) evaluates and selects the most promising cut points grouped by feature according to the MDLP criterion (single-step version).

Algorithm 3 Function to select the best cut points for a given feature *(select_thresholds)*

Input: $candidates$ A RDD/array of tuples ($< point, q >$), where $point$ represents a candidate point to evaluate and q the class counter.
Input: $maxbins$ Maximum number of intervals or bins to select
Input: $maxcand$ Maximum number of candidates to eval in a partition
Output: An array of thresholds for a given feature

1: $stack \leftarrow enqueue(stack, (candidates, ()))$
2: $result \leftarrow ()$
3: **while** $|stack| > 0$ & $|result| < maxbins$ **do**
4: $\quad (subset, lth) \leftarrow dequeue(stack)$
5: $\quad$ **if** $|subset| > 0$ **then**
6: $\quad\quad$ **if** $type(subset) = 'array'$ **then**
7: $\quad\quad\quad bound \leftarrow arr_select_thresholds(subset, lth)$
8: $\quad\quad$ **else**
9: $\quad\quad\quad bound \leftarrow rdd_select_thresholds(subset, lth, maxcand)$
10: $\quad\quad$ **end if**
11: $\quad\quad$ **if** $bound <> ()$ **then**
12: $\quad\quad\quad result \leftarrow result + bound$
13: $\quad\quad\quad (left, right) \leftarrow divide_partitions(subset, bound)$
14: $\quad\quad\quad stack \leftarrow enqueue(stack, (left, bound))$
15: $\quad\quad\quad stack \leftarrow enqueue(stack, (right, bound))$
16: $\quad\quad$ **end if**
17: $\quad$ **end if**
18: **end while**
19: $return(sort(result))$

This algorithm starts by selecting the best cut point in the whole set. If the criterion accepts this selection, the point is added to the result list and the current subset is divided into two new partitions using this cut point. Both partitions are then evaluated, repeating the previous process. This process finishes when there is no partition to evaluate or the number of selected points is fulfilled.

Algorithm 4 ($arr_select_thresholds$) explains the process that accumulates frequencies and then selects the minimum-entropy candidate. This version is more straightforward than RDD version as it only needs to accumulate frequencies sequentially. Firstly, it obtains the total class counter vector by aggregating all candidate vectors. Afterwards, a new iteration is necessary to obtain the accumulated counters for the two partitions generated by each point. This is done by aggregating the vectors from the most-left point to the current one, and from the current point to the right-most point. Once the accumulated counters for each candidate point are calculated (in form of $< point, q, leftq, rightq >$), the algorithm evaluates the candidates using the $select_best_cut$ function.

Algorithm 5 ($rdd_select_thresholds$) explains the selection process but for "big" features (more than one partition). This process needs to be performed in a distributed manner

Algorithm 4 Function to select the best cut point according to MDLP criterion (single-step version) *(arr_select_thresholds)*

Input: $candidates$ An array of tuples ($< point, q >$), where $point$ represents a candidate point to evaluate and q the class counter.
Input: lth Last threshold selected.
Output: The minimum-entropy cut point

1: $total \leftarrow sum_freqs(candidates)$
2: $leftacc \leftarrow ()$
3: **for** $< point, q > \in candidates$ **do**
4: $\quad leftacc \leftarrow leftacc + q$
5: $\quad freqs \leftarrow freqs + (point, q, leftacc, total - leftacc)$
6: **end for**
7: $return(select_best_cut(candidates, freqs))$

since the number of candidate points exceeds the maximum size defined. For each feature, the subset of points is hence re-distributed in a better partition scheme to homogenize the quantity of points by partition and node ($coalesce$ function, line 1-2). After that, a new parallel function is started to compute the accumulated counter by partition. The results (by partition) are then aggregated to obtain the total accumulated frequency for the whole subset. In line 9, it is started a new distributed process with the aim of computing the accumulated frequencies by point from both sides (as explained in Algorithm 4). In this procedure, the process accumulates the counter from all previous partitions to the current one to obtain the first accumulated value (left one). Then the function computes the accumulated values for each inner point using the counter for points in the current partition, the left value and the total ones (line 7). Once these values are calculated ($< point, q, leftq, rightq >$), the algorithm evaluates all candidate points and their associated accumulators using the $select_best_cut$ function (as above).

Algorithm 6 evaluates the discretization schemes yielded by each point by computing the entropy for each partition generated, also taking into account the MDLP criterion. Thus, for all points[4], it is calculated the entropy for the two generated partitions (line 8) as well as the total entropy for the whole set (lines 1-2). Using these values, the entropy gain for each point is computed and its MDLP condition, according to Equation 2. If the point is accepted by MDLP, the algorithm emits a tuple with the weighted entropy average of partition and the point itself. From the set of accepted points, the algorithm selects the one with the minimum class information entropy.

The results produced by both groups (Algorithm 1, line 19-22) are joined into the final point set of cut points.

Figure 1 outlines the most important steps in our method. The first phase computes all distinct points from the original data, and then distributed in several partitions. These points are sorted by feature and value in the second phase. Before this phase, the first points in each partition[5] are sent to the next one, so that each node has the node to the right to decide if generating a midpoint or not. In the third phase, the boundary points are generated. For instance, we can see as a midpoint ([1, 3.5]) is generated between point (1,2) and point (1,4) due to their class vectors have elements of both class (there is a limit between classes). In contrast, points (2,1) and (2,3) have only elements for the same class, not generating any point. Finally, other points are added in spite of not forming

[4]If the set is an array, it is used a loop structure else it is used a distributed map function

[5]Those in the second row in the second phase

Algorithm 5 Function that selects the best cut points according to MDLP criterion (RDD version) *(rdd_select_thresholds)*

Input: $candidates$ An RDD of tuples ($< point, q >$), where $point$ represents a candidate point to evaluate and q the class counter.
Input: lth Last threshold selected
Input: $maxcand$ Maximum number of candidates to eval in a partition
Output: The minimum-entropy cut point

1: $npart \leftarrow round(|candidates|/maxcand)$
2: $candidates \leftarrow coalesce(candidates, npart)$
3: $totalpart \leftarrow$
4: **map partitions** $partition \in candidates$
5: $\quad return(sum(partition))$
6: **end map**
7: $total \leftarrow sum(totalpart)$
8: $freqs \leftarrow$
9: **map partitions** $partition \in candidates$
10: $\quad index \leftarrow get_index(partition)$
11: $\quad lefttotal \leftarrow ()$
12: $\quad freqs \leftarrow ()$
13: $\quad$ **for** $i = 0$ $until$ $index$ **do**
14: $\quad\quad lefttotal \leftarrow lefttotal + totalpart(i)$
15: $\quad$ **end for**
16: $\quad$ **for all** $< point, q > \in partition$ **do**
17: $\quad\quad freqs \leftarrow freqs + (point, q, lefttotal + q, total - lefttotal)$
18: $\quad$ **end for**
19: $\quad return(freqs)$
20: **end map**
21: $return(select_best_cut(candidates, freqs))$

Algorithm 6 Function that calculates class entropy values and selects the minimum-entropy cut point *(select_best_cut)*

Input: $freqs$ An array/RDD of tuples ($< point, q, leftq, rightq >$), where point represents a candidate point to evaluate, leftq the left accumulated frequency, rightq the right accumulated frequency and q the class frequency counter.
Input: $total$ Class frequency counter for all the elements
Output: The minimum-entropy cut point

1: $n \leftarrow sum(total)$
2: $totalent \leftarrow ent(total, n)$
3: $k \leftarrow |total|$
4: $accepted \leftarrow$
5: **map** $< point, q, leftq, rightq > \in freqs$
6: $\quad k1 \leftarrow |leftq|; k2 \leftarrow |rightq|$
7: $\quad s1 \leftarrow sum(leftq); s2 \leftarrow sum(rightq);$
8: $\quad ent1 \leftarrow ent(s1, k1); ent2 \leftarrow ent(s2, k2)$
9: $\quad partent \leftarrow (s1 * ent1 + s2 * ent2)/s$
10: $\quad gain \leftarrow totalent - partent$
11: $\quad delta \leftarrow log_2(3^k - 2) - (k * hs - k1 * ent1 - k2 * ent2)$
12: $\quad accepted \leftarrow gain > ((log_2(s - 1)) + delta)/n$
13: $\quad$ **if** $accepted = true$ **then**
14: $\quad\quad EMIT < partent, point >$
15: $\quad$ **end if**
16: **end map**
17: $return(min(accepted))$

limits. Last points in the partitions are added to pool in order to isolate the selection process between partitions. The last phase consists of grouping the candidate points by feature and evaluating each feature independently using the MDLP criterion (as explained above).

IV. Experimental Framework and Analysis

This section describes the experiments carried out to demonstrate the usefulness and performance of our discretization solution over two Big Data problems.

A. Experimental Framework

Two huge classification datasets are employed as benchmarks in our experiments. The first one (hereinafter called *ECBDL14*) was used as a reference at the ML competition of the Evolutionary Computation for Big Data and Big Learning held on July 14, 2014, under the international conference GECCO-2014. This consists of 631 characteristics (including both numerical and categorical attributes) and 32 million instances. It is a binary classification problem where the class distribution is highly imbalanced: 2% of positive instances. For this problem, the MapReduce version of the Random OverSampling (ROS) algorithm presented in [24] was applied in order to replicate the minority class instances from the original dataset until the number of instances for both classes was equalized.

As a second dataset, we have used *epsilon*, which consists of 500 000 instances with 2000 numerical features. This dataset was artificially created for the Pascal Large Scale Learning Challenge in 2008. It was further pre-processed and included in the LibSVM dataset repository [25].

Table I gives a brief 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.), the total number of training data (#Total), and the number classes (#Cl) are shown.

TABLE I
Summary description for classification datasets

Data Set	#Train Ex.	#Test Ex.	#Atts.	#Total	#Cl.
epsilon	400 000	100 000	2000	800 000 000	2
ECBDL14 (ROS)	65 003 913	2 897 917	631	41 017 469 103	2

For our algorithm, we have established a maximum number of 50 intervals and a maximum number of candidates per partition of 100,000. For evaluation purposes, Naive Bayes [26] has been chosen as reference in classification, using the implementation included in MLlib library [15]. The default values for the classifier are: $\lambda = 1.0$, $iterations = 100$.

As evaluation criteria, we use two well-known evaluation metrics to assess the quality of the underlying discretization schemes. First of all, **classification accuracy** is used to evaluate the accuracy yielded by the classifiers -number of examples correctly labeled as belonging to a given class divided by the total number of elements belonging to the positive class-. On the other hand, in order to prove the time benefits on using discretization, we have employed the overall classification **runtime** (in seconds) in training as well as the overall time in discretization as additional measures.

For all experiments we have used a cluster composed of twenty computing nodes and one master node. The computing nodes hold the following characteristics: 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. Regarding software, we have used the following configuration: Hadoop 2.5.0-cdh5.3.1 from Cloudera's open-source Apache Hadoop distribution[6], Apache Spark and MLlib 1.2.0, 480 cores (24 cores/node), 1040 RAM GB (52 GB/node).

Spark implementation of the algorithm can be downloaded from the Spark's community repository[7]. The design of the algorithm has been adapted to be integrated in MLlib Library as a third-party package.

B. Experimental Results and Analysis

Table II shows the accuracy values in classification for both datasets using our distributed discretization approach[8]. According to these results, we can assert that our discretization algorithm always outperforms the version without discretization (both for training and test). It is specially important in ECBDL14 where there is a improvement of 0.1.

TABLE II
Classification accuracy values (distributed MDLP)

	NB		NB-disc	
Dataset	*Train*	*Test*	*Train*	*Test*
ECBDL14	0.5260	0.6276	**0.6659**	**0.7260**
epsilon	0.6542	0.6550	**0.7094**	**0.7065**

Table III shows classification runtime values for both datasets distinguishing between whether discretization is applied or not. As we can see, there is a slight improvement in both cases.

TABLE III
Classification time values (with discretization vs. w/o discretization) in seconds

Dataset	NB	NB-disc
ECBDL14	31.06	**26.39**
epsilon	5.72	**4.99**

Table IV shows discretization time values for two different versions of MDLP discretizer, namely, sequential and distributed. For the sequential version, we have measured the time employed by the original proposal on smaller subsets for

[6]http://www.cloudera.com/content/cloudera/en/documentation/cdh5/v5-0-0/CDH5-homepage.html
[7]http://spark-packages.org/package/sramirez/spark-MDLP-discretization
[8]In all tables, the best result by column (best by method) is highlighted in bold.

both datasets. The overall time value shown on the table was estimated from the previous measurements. Comparing both implementations, we can notice the advantage of using the distributed version against the sequential one. For ECBDL14, our version is almost 300 times faster than the sequential one in the best case whereas for epsilon is 12 times faster. This demonstrates that the bigger the dataset, the higher the efficiency improvement.

TABLE IV
DISCRETIZATION TIME VALUES (SEQUENTIAL VS. DISTRIBUTED) IN SECONDS

Dataset	Sequential (estimation)	Distributed
ECBDL14	295 508	**1 087**
epsilon	5 764	**476**

V. CONCLUSIONS

In this paper, we have presented the problem of large-scale data processing, focusing on the process of discretization of large datasets. We have proposed a sound multi-interval discretization method based on entropy minimization and also described the difficulty of adapting this to the new distributed paradigms present in Big Data. In spite of the great interest in this reduction mechanism, only few simple discretization techniques have been implemented in the literature.

Thereby we propose a completely distributed version of MDLP discretizer capable of transforming the iterativity yielded by the original proposal in a single-step computation. To accomplish that, a complete redesign of the original proposal has been necessary. Furthermore, with this work we would like to contribute to MLlib library by adding a complex discretization algorithm where only very simple discretizers are available.

The experimental results have demonstrated the improvement in both classification accuracy and time when using our discretization solution for both datasets used. Additionally, our algorithm has shown to perform 300 times faster than the sequential version for the largest dataset.

ACKNOWLEDGMENT

This work is supported by the National Research Project TIN2011-28488, TIN2012-37954 and TIN2013-47210-P, and the Andalusian Research Plan P10-TIC-6858, P11-TIC-7765 and P12-TIC-2958, and by the Xunta de Galicia through the research project GRC 2014/035 (all projects partially funded by FEDER funds of the European Union). S. Ramírez-Gallego holds a FPU scholarship from the Spanish Ministry of Education and Science (FPU13/00047). D. Martínez-Rego acknowledges support of the Xunta de Galicia under postdoctoral Grant code POS-A/2013/196.

REFERENCES

[1] IBM Big Data, "IBM - Bringing Big Data to the Enterprise," 2015. [Online]. Available: http://www-01.ibm.com/software/data/bigdata/

[2] M. Minelli, M. Chambers, and A. Dhiraj, *Big Data, Big Analytics: Emerging Business Intelligence and Analytic Trends for Today's Businesses*. John Wiley and Sons, 2013.

[3] S. García, J. Luengo, and F. Herrera, *Data Preprocessing in Data Mining*. Springer, 2015.

[4] Y. Yang, G. I. Webb, and X. Wu, "Discretization methods," in *Data Mining and Knowledge Discovery Handbook*, 2010, pp. 101–116.

[5] S. García, J. Luengo, J. A. Sáez, V. López, and F. Herrera, "A survey of discretization techniques: Taxonomy and empirical analysis in supervised learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 4, pp. 734–750, 2013.

[6] H. Liu, F. Hussain, C. L. Tan, and M. Dash, "Discretization: An enabling technique," *Data Mining and Knowledge Discovery*, vol. 6, no. 4, pp. 393–423, 2002.

[7] H.-W. Hu, Y.-L. Chen, and K. Tang, "A novel decision-tree method for structured continuous-label classification." *IEEE T. Cybernetics*, vol. 43, no. 6, pp. 1734–1746, 2013.

[8] H. W. Hu, Y. L. Chen, and K. Tang, "A dynamic discretization approach for constructing decision trees with a continuous label," *IEEE Trans. on Knowl. and Data Eng.*, vol. 21, no. 11, pp. 1505–1514, 2009.

[9] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI 2004*, 2004, pp. 137–150.

[10] T. White, *Hadoop, The Definitive Guide*. O'Reilly Media, Inc., 2012.

[11] Apache Hadoop Project, "Apache Hadoop," 2015. [Online]. Available: http://hadoop.apache.org/

[12] M. Hamstra, H. Karau, M. Zaharia, A. Konwinski, and P. Wendell, *Learning Spark: Lightning-Fast Big Data Analytics*. O'Reilly Media, Incorporated, 2015.

[13] Apache Spark: Lightning-fast cluster computing, "Apache spark," 2015. [Online]. Available: https://spark.apache.org/

[14] Apache Mahout Project, "Apache Mahout," 2015. [Online]. Available: http://mahout.apache.org/

[15] Machine Learning Library (MLlib) for Spark, "Mllib," 2015. [Online]. Available: http://spark.apache.org/docs/latest/mllib-guide.html

[16] U. M. Fayyad and K. B. Irani, "Multi-interval discretization of continuous-valued attributes for classification learning," in *International Joint Conference On Artificial Intelligence*, vol. 13. Morgan Kaufmann, 1993, pp. 1022–1029.

[17] B. S. Chlebus, "On the klee's measure problem in small dimensions," in *SOFSEM '98: Theory and Practice of Informatics, 25th Conference on Current Trends in Theory and Practice of Informatics, Jasná, Slovakia, November 21-27, 1998, Proceedings*, 1998, pp. 304–311.

[18] T. Elomaa and J. Rousu, "General and efficient multisplitting of numerical attributes," *Machine Learning*, vol. 36, pp. 201–244, 1999.

[19] J. Quinlan, "Induction of decision trees," in *Readings in Machine Learning*, I. S. J.W. and D. T.G., Eds. Morgan Kaufmann Publishers, 1990, originally published in *Machine Learning* 1:81-106, 1986.

[20] C. E. Shannon, "A mathematical theory of communication," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 5, no. 1, pp. 3–55, Jan. 2001.

[21] M. Beyer and D. Laney, "3d data management: Controlling data volume, velocity and variety," 2001. [Online]. Available: http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf

[22] A. Fernández, S. del Río, V. López, A. Bawakid, M. J. del Jesús, J. M. Benítez, and F. Herrera, "Big data with cloud computing: an insight on the computing environment, mapreduce, and programming frameworks," *Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery*, vol. 4, no. 5, pp. 380–409, 2014.

[23] J. Lin, "Mapreduce is good enough? if all you have is a hammer, throw away everything that's not a nail!" *CoRR*, vol. abs/1209.2191, 2012.

[24] S. Rio, V. Lopez, J. Benitez, and F. Herrera, "On the use of mapreduce for imbalanced big data using random forest." *Information Sciences*, no. 285, pp. 112–137, 2014.

[25] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011, datasets available at http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/.

[26] R. O. Duda and P. E. Hart, *Pattern classification and scene analysis*. Wiley New York, 1973, vol. 3.