Advanced Review

Data discretization: taxonomy and big data challenge

Sergio Ramírez-Gallego,[1] Salvador García,[1*] Héctor Mouriño-Talín,[2] David Martínez-Rego,[2,3] Verónica Bolón-Canedo,[2] Amparo Alonso-Betanzos,[2] José Manuel Benítez[1] and Francisco Herrera[1]

Discretization of numerical data is one of the most influential data preprocessing tasks in knowledge discovery and data mining. The purpose of attribute discretization is to find concise data representations as categories which are adequate for the learning task retaining as much information in the original continuous attribute as possible. In this article, we present an updated overview of discretization techniques in conjunction with a complete taxonomy of the leading discretizers. Despite the great impact of discretization as data preprocessing technique, few elementary approaches have been developed in the literature for Big Data. The purpose of this article is twofold: a comprehensive taxonomy of discretization techniques to help the practitioners in the use of the algorithms is presented; the article aims is to demonstrate that standard discretization methods can be parallelized in Big Data platforms such as Apache Spark, boosting both performance and accuracy. We thus propose a distributed implementation of one of the most well-known discretizers based on Information Theory, obtaining better results than the one produced by: the entropy minimization discretizer proposed by Fayyad and Irani. Our scheme goes beyond a simple parallelization and it is intended to be the first to face the Big Data challenge.

How to cite this article:
WIREs Data Mining Knowl Discov 2015. doi: 10.1002/widm.1173

INTRODUCTION

Data are present in diverse formats, for example in categorical, numerical, or continuous values. Categorical or nominal values are unsorted, whereas numerical or continuous values are assumed to be sorted or represent ordinal data. It is well-known that data mining (DM) algorithms depend very much on the domain and type of data. In this way, the techniques belonging to the field of statistical learning work with numerical data (i.e., support vector machines and instance-based learning) whereas symbolic learning methods require inherent finite values and also prefer to perform a branch of values that are not ordered (such as in the case of decision trees or rule induction learning). These techniques are either expected to work on discretized data or to be integrated with internal mechanisms to perform discretization.

The process of discretization has aroused general interest in recent years[1,2] and has become one of the most effective data preprocessing techniques in DM.[3] Roughly speaking, discretization translates quantitative data into qualitative data, procuring a nonoverlapping division of a continuous domain. It also ensures an association between each numerical value and a certain interval. Actually, discretization is considered a data reduction mechanism because it diminishes data from a large domain of numeric values to a subset of categorical values.

There is a necessity to use discretized data by many DM algorithms which can only deal with

*Correspondence to: salvagl@decsai.ugr.es

[1]Department of Computer Science and Artificial Intelligence, University of Granada, Granada, Spain

[2]Department of Computer Science, University of A Coruña, A Coruña, Spain

[3]Department of Computer Science, University College London, London, UK

Conflict of interest: The authors have declared no conflicts of interest for this article.

discrete attributes. For example, three of the ten methods pointed out as the top ten in DM[4] demand a data discretization in one form or another: C4.5,[5] Apriori,[6] and Naïve Bayes.[7] Among its main benefits, discretization causes that the learning methods show remarkable improvements in learning speed and accuracy. Besides, some decision tree-based algorithms produce shorter, more compact, and accurate results when using discrete values.[1,8]

The specialized literature reports on a huge number of proposals for discretization. In fact, some surveys have been developed attempting to organize the available pool of techniques.[1,2,9] It is crucial to determine, when dealing with a new real problem or dataset, the best choice in the selection of a discretizer. This will imply the success and the suitability of the subsequent learning phase in terms of accuracy and simplicity of the solution obtained. In spite of the effort made in Ref 2 to categorize the whole family of discretizers, probably the most well-known and surely most effective are included in a new taxonomy presented in this article, which has now been updated at the time of writing.

Classical data reduction methods are not expected to scale well when managing huge data—both in number of features and instances—so that its application can be undermined or even become impracticable.[10] Scalable distributed techniques and frameworks have appeared along with the problem of Big Data. MapReduce[11] and its open-source version Apache Hadoop[12,13] were the first distributed programming techniques to face this problem. Apache Spark[14,15] 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 DM problems. Similarly, several DM libraries for Big Data have appeared as support for this task. The first one was Mahout[16] (as part of Hadoop), subsequently followed by MLlib[17] which is part of the Spark project.[14] Although many state-of-the-art DM algorithms have been implemented in MLlib, it is not the case for discretization algorithms yet.

In order to fill this gap, we face the Big Data challenge by presenting a distributed version of the entropy minimization discretizer proposed by Fayyad and Irani in Ref 18, using Apache Spark, which is based on Minimum Description Length Principle. Our main objective is to prove that well-known discretization algorithms as MDL-based discretizer (henceforth called MDLP) can be parallelized in these frameworks, providing good discretization solutions for Big Data analytics. Furthermore, we have transformed the iterativity yielded by the original proposal in a single-step computation. This new version for distributed environments has supposed a deep restructuring of the original proposal and a challenge for the authors. Finally, to demonstrate the effectiveness of our framework, we perform an experimental evaluation using two large datasets, namely, ECBDL14 and epsilon.

In order to achieve the goals mentioned above, this article is structured as follows. First, we provide in the next Section (Background and Properties) an explanation of discretization, its properties and the description of the standard MDLP technique. The next Section (Taxonomy) presents the updated taxonomy of the most relevant discretization methods. Afterwards, in the Section *Big Data Background*, we focus on the background of the Big Data challenge including the MapReduce programming framework as the most prominent solution for Big Data. The following section (Distributed MDLP Discretization) describes the distributed algorithm based on entropy minimization proposed for Big Data. The experimental framework, results, and analysis are given in last but one section (Experimental Framework and Analysis). Finally, the main concluding remarks are summarized.

BACKGROUND AND PROPERTIES

Discretization is a wide field and there have been many advances and ideas over the years. This section is devoted to providing a proper background on the topic, including an explanation of the basic discretization process and enumerating the main properties that allow us to categorize them and to build a useful taxonomy.

Discretization Process

In supervised learning, and specifically in classification, the problem of discretization can be defined as follows. Assuming a dataset 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} are, respectively, 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.

We can associate a typical discretization as a univariate discretization. Although this property will be reviewed in the next section, it is necessary to introduce it here for the basic understanding of the

basic discretization process. Univariate discretization operates with one continuous feature at a time while multivariate discretization considers multiple features simultaneously.

A typical discretization process generally consists of four steps (seen in Figure 1): (1) *sorting* the continuous values of the feature to be discretized, either (2) *evaluating* a cut point for splitting or adjacent intervals for merging, (3) *splitting or merging* intervals of continuous values according to some defined criterion, and (4) *stopping* at some point. Next, we explain these four steps in detail.

- Sorting: The continuous values for a feature are sorted in either descending or ascending order. It is crucial to use an efficient sorting algorithm with a time complexity of $O(NlogN)$. Sorting must be done only once and for the entire initial process of discretization. It is a mandatory treatment and can be applied when the complete instance space is used for discretization.
- Selection of a Cut Point: After sorting, the best cut point or the best pair of adjacent intervals should be found in the attribute range in order to split or merge in a following required step. An evaluation measure or function is used to determine the correlation, gain, improvement in performance, or any other benefit according to the class label.
- Splitting/Merging: Depending on the operation method of the discretizers, intervals either can be split or merged. For splitting, the possible cut points are the different real values present in an attribute. For merging, the discretizer aims to find the best adjacent intervals to merge in each iteration.
- Stopping Criteria: It specifies when to stop the discretization process. It should assume a trade-off between a final lower number of intervals, good comprehension, and consistency.

Discretization Properties

In Ref 1,2,9 various pivots have been used in order to make a classification of discretization techniques. This section reviews and describes them, underlining the major aspects and alliances found among them. The taxonomy presented afterwards will be founded on these characteristics (acronyms of the methods correspond with those presented in Table 1):

- *Static versus Dynamic:* This property refers to the level of independence between the discretizer and the learning method. A static discretizer is run prior to the learning task and is autonomous from the learning algorithm,[1] as a data preprocessing algorithm.[3] Almost all isolated known discretizers are static. By contrast, a

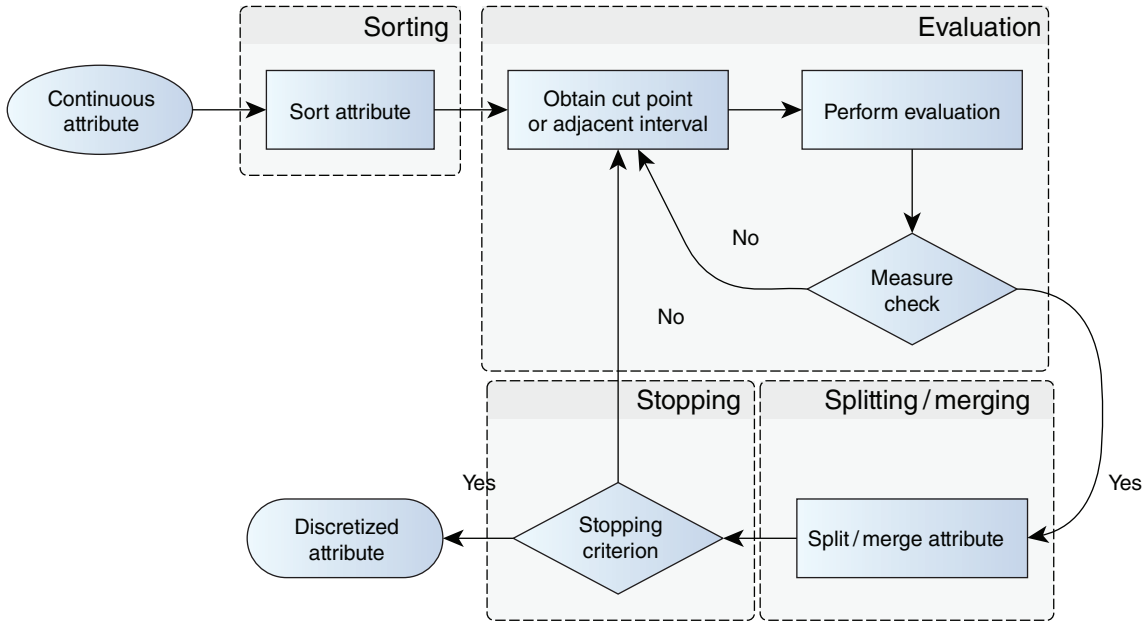

FIGURE 1 | Discretization process.

TABLE 1 | Most Important Discretizers

Acronym	Ref.	Acronym	Ref.	Acronym	Ref.
EqualWidth	19	EqualFrequency	19	Chou91	20
D2	21	ChiMerge	22	1R	23
ID3	5	MDLP	18	CADD	24
MDL-Disc	25	Bayesian	26	Friedman96	27
ClusterAnalysis	28	Zeta	29	Distance	30
Chi2	31	CM-NFD	32	FUSINTER	33
MVD	34	Modified Chi2	35	USD	36
Khiops	37	CAIM	38	Extended Chi2	39
Heter-Disc	40	UCPD	41	MODL	42
ITPF	43	HellingerBD	44	DIBD	45
IDD	46	CACC	47	Ameva	48
Unification	49	PKID	7	FFD	7
CACM	50	DRDS	51	EDISC	52
U-LBG	53	MAD	54	IDF	55
IDW	55	NCAIC	56	Sang14	57
IPD	58	SMDNS	59	TD4C	60
EMD	61				

MDLP, Minimum Description Length Principle.

dynamic discretizer responds when the learner requires so, during the building of the model. Hence, they must belong to the local discretizer's family (see later) embedded in the learner itself, producing an accurate and compact outcome together with the associated learning algorithm. Good examples of classical dynamic techniques are ID3 discretizer[5] and ITFP.[43]

- *Univariate versus Multivariate:* Univariate discretizers only operate with a single attribute simultaneously. This means that they sort the attributes independently, and then, the derived discretization disposal for each attribute remains unchanged in the following phases. Conversely, multivariate techniques, concurrently consider all or various attributes to determine the initial set of cut points or to make a decision about the best cut point chosen as a whole. They may accomplish discretization handling the complex interactions among several attributes to decide also the attribute in which the next cut point will be split or merged. Currently, interest has recently appeared in developing multivariate discretizers because they are decisive in complex predictive problems where univariate operations may ignore important interactions between attributes[60,61] and in deductive learning.[58]
- *Supervised versus Unsupervised:* Supervised discretizers consider the class label whereas unsupervised ones do not. The interaction between the input attributes and the class output and the measures used to make decisions on the best cut points (entropy, correlations, etc.) will define the supervised manner to discretize. Although most of the discretizers proposed are supervised, there is a growing interest in unsupervised discretization for descriptive tasks.[53,58] Unsupervised discretization can be applied to both supervised and unsupervised learning, because its operation does not require the specification of an output attribute. Nevertheless, this does not occur in supervised discretizers, which can only be applied over supervised learning. Unsupervised learning also opens the door to transferring the learning between tasks because the discretization is not tailored to a specific problem.
- *Splitting versus Merging:* These two options refer to the approach used to define or generate new intervals. The former methods search for a cut point to divide the domain into two intervals among all the possible boundary points. On the contrary, merging techniques begin with a predefined partition and search for a candidate cut point to mix both adjacent intervals after removing it. In the literature, the terms

top-down and *bottom-up* are highly related to these two operations, respectively. In fact, top-down and bottom-up discretizers are thought for hierarchical discretization developments, so they consider that the process is incremental, property which will be described later. Splitting/merging is more general than top-down/bottom-up because it is possible to have discretizers whose procedure manages more than one interval at a time.[44,46] Furthermore, we consider the *hybrid* category as the way of alternating splits with merges during running time.[24,61]

- *Global versus Local:* In the time a discretizer must select a candidate cut point to be either split or merged, it could consider either all available information in the attribute or only partial information. A local discretizer makes the partition decision based only on partial information. MDLP[18] and ID3[5] are classical examples of local methods. By definition, all the dynamic discretizers and some top-down-based methods are local, which explains the fact that few discretizers apply this form. The dynamic discretizers search for the best cut point during internal operations of a certain DM algorithm, thus it is impossible to examine the complete dataset. Besides, top-down procedures are associated with the divide-and-conquer scheme, in such manner that when a split is considered, the data is recursively divided, restricting access to partial data.
- *Direct versus Incremental:* For direct discretizers, the range associated with an interval must be divided into k intervals simultaneously, requiring an additional criterion to determine the value of k. One-step discretization methods and discretizers which select more than a single cut point at every step are included in this category. However, incremental methods begin with a simple discretization and pass through an improvement process, requiring an additional criterion to determine when it is the best moment to stop. At each step, they find the best candidate boundary to be used as a cut point and, afterwards, the rest of the decisions are made accordingly.
- *Evaluation Measure:* This is the metric used by the discretizer to compare two candidate discretization schemes and decide which is more suitable to be used. We consider five main families of evaluation measures:

  – *Information:* This family includes *entropy* as the most used evaluation measure in discretization (MDLP,[18] ID3,[5] FUSINTER[33]) and others derived from information theory (*Gini index*, *Mutual information*).[49]

  – *Statistical:* Statistical evaluation involves the measurement of dependency/correlation among attributes (Zeta,[29] ChiMerge,[22] Chi2[31]), interdependency,[38] probability and Bayesian properties[26] (MODL[42]), contingency coefficient,[47] etc.

  – *Rough Sets:* This class is composed of methods that evaluate the discretization schemes by using rough set properties and measures,[59] such as class separability, lower and upper approximations, etc.

  – *Wrapper:* This collection comprises methods that rely on the error provided by a classifier or a set of classifiers that are used in each evaluation. Representative examples are MAD,[54] IDW,[55] and EMD.[61]

  Binning: In this category of techniques, there is no evaluation measure. This refers to discretizing an attribute with a predefined number of bins in a simple way. A bin assigns a certain number of values per attribute by using a non-sophisticated procedure. EqualWidth and EqualFrequency discretizers are the most well-known unsupervised binning methods.

Minimum Description Length-Based Discretizer

Minimum Description Length-based discretizer,[18] proposed by Fayyad and Irani in 1993, is one of the most important splitting methods in discretization. This univariate discretizer uses the MDLP to control the partitioning process. This also introduces an optimization based on a reduction of whole set of candidate points, only formed by the *boundary points* in this set.

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, two examples exist $u, v \in S$ with different class labels, such that $A(u) < b < A(v)$; and the other example $w \in S$ does not exist, such that $A(u) < A(w) < A(v)$. The set of boundary points for attribute A is defined as B_A.

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 such as ID3,[5] 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.

It recursively evaluates all boundary points, computing the class entropy of the partitions derived as quality measure. 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:

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

where E represents the class entropy[a] of a given subset following Shannon's definitions.[62]

Finally, a decision criterion is defined in order to control when to stop the partitioning process. The use of MDLP as a decision criterion allows us to decide whether or not to partition. Thus a cut point b_α will be applied iff:

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

where $\Delta(A, b_\alpha, S) = \log_2(3^c) - [cE(S) - c_1E(S_1) - c_2E(S_2)]$, c_1 and c_2 the number of class labels in S_1 and S_2, respectively; and $G(A, b_\alpha, S) = E(S) - EP(A, b_\alpha, S)$.

TAXONOMY

Currently, more than 100 discretization methods have been presented in the specialized literature. In this section, we consider a subgroup of methods which can be considered the most important from the whole set of discretizers. The criteria adopted to characterize this subgroup are based on the repercussion, availability, and novelty they have. Thus, the precursory discretizers which have served as inspiration to others, those which have been integrated in software suites and the most recent ones are included in this taxonomy.

Table 1 enumerates the discretizers considered in this article, providing the name abbreviation and reference for each one. We do not include the descriptions of these discretizers in this article. Their definitions are contained in the original references, thus we recommend consulting them in order to understand how the discretizers of interest work. In Table 1, 30 discretizers included in KEEL software are considered. Additionally, implementations of these algorithms in Java can be found.[63]

In the previous section, we studied the properties which could be used to classify the discretizers proposed in the literature. Given a predefined order among the seven characteristics studied before, we can build taxonomy of discretization methods. All techniques enumerated in Table 1 are collected in the taxonomy depicted in Figure 2. It represents a hierarchical categorization following the next arrangement of properties: static/dynamic, univariate/multivariate, supervised/unsupervised, splitting/merging/hybrid, global/local, direct/incremental, and evaluation measure.

The purpose of this taxonomy is twofold. First, it identifies a subset of most representative state-of-the-art discretizers for both researchers and practitioners who want to compare them with novel techniques or require discretization in their applications. Second, it characterizes the relationships among techniques, the extension of the families and possible gaps to be filled in future developments.

When managing huge data, most of them become impracticable in real-world settings, due to the complexity they cause (for example, in the case of MDLP, among others). The adaptation of these classical methods implies a thorough redesign that becomes mandatory if we want to exploit the advantages derived from the use of discrete data on large datasets.[64,65] As reflected in our taxonomy, no relevant methods in the field of Big Data have been proposed to solve this problem. Some works have tried to deal with large-scale discretization. For example, in Ref 66, the authors proposed a scalable implementation of Class-Attribute Interdependence Maximization algorithm by using GPU technology. In Ref 67, a discretizer based on windowing and hierarchical clustering is proposed to improve the performance of classical tree-based classifiers. However, none of these methods have been proved to cope with the data magnitude presented here.

BIG DATA BACKGROUND

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

Gartner[68] introduced the concept of Big Data and the 3V terms that define it as high volume,

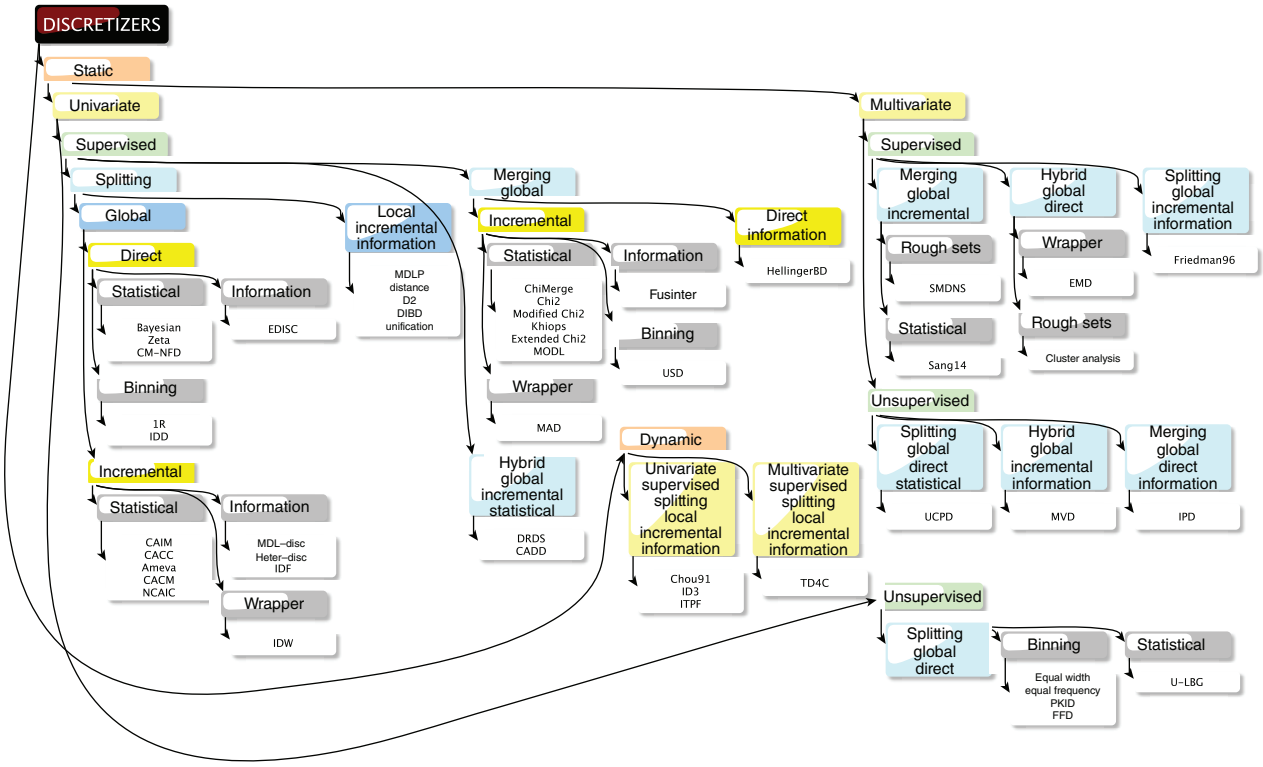

FIGURE 2 | Discretization taxonomy.

velocity, and variety of information that require a new large-scale processing. This list was then extended with two additional terms. All of them are described in the following: *Volume*, the massive amount of data that is produced every day is still exponentially growing (from terabytes to exabytes); *Velocity*, data need to be loaded, analyzed, and stored as quickly as possible; *Variety*, data come in many formats and representations; *Veracity*, the quality of data to process is also an important factor. The Internet is full of missing, incomplete, ambiguous, and sparse data; *Value*, extracting value from data is also established as a relevant objective in big analytics.

The unsuitability of many knowledge extraction algorithms in the Big Data field has meant that new methods have been developed to manage such amounts of data effectively and at a pace that allows value to be extracted from them.

MapReduce Model and Other Distributed Frameworks

The MapReduce framework,[11] 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.[b] 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 first step, the master node breaks up the dataset into several splits, distributing them across the cluster for parallel processing. Each node then hosts several Map threads that transform the generated key-value pairs into a set of intermediate pairs. After all Map tasks have finished, the master node distributes the matching pairs across the nodes according to a key-based partitioning scheme. Then the Reduce phase starts, combining those coincident pairs so as to form the final output.

Apache Hadoop[12,13] 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 poor performance on iterative and online computing, and a poor intercommunication capability or inadequacy for in-memory computation, among others.[70] Recently, Apache Spark[14,15] has appeared and integrated with the Hadoop ecosystem. This novel framework is presented as a revolutionary tool capable of performing even faster large-scale processing than Hadoop through in-memory primitives,

making this framework a leading tool for iterative and online processing and, thus, suitable for DM algorithms. Spark is built on distributed data structures called resilient distributed datasets (RDDs), which were designed as a fault-tolerant collection of elements that can be operated in parallel by means of data partitioning.

DISTRIBUTED MDLP DISCRETIZATION

In the Background section, a discretization algorithm based on an information entropy minimization heuristic was presented.[18] In this work, the authors proved 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 the two time-consuming operations: on one hand, the sorting of candidate points, and, on the other hand, the evaluation of these points. The sorting operation exhibits 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 to 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).

Spark primitives extend the idea of MapReduce to implement more complex operations on distributed data. In order to implement our method, we have used some extra primitives from Spark's API, such as: mapPartitions, sortByKey, flatMap, and reduceByKey.[c]

Main Discretization Procedure

Algorithm 1 explains the main procedures in our discretization algorithm. The algorithm calculates the minimum-entropy cut points by feature according to the MDLP criterion. It uses a parameter to limit the maximum number of points to yield.

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

Algorithm 1: Main discretization procedure

Input: S Data set
Input: M Feature indexes to discretize
Input: mb Maximum number of cut points to select
Input: mc Maximum number of candidates per partition
Output: Cut points by feature

```
comb ←
  map s ∈ S
    v ← zeros(|c|)
    ci ← class_index(v)
    v(ci) ← 1
    for all A ∈ M do
      EMIT < (A, A(s)), v >
    end for
  end map
distinct ← reduce(comb, sum_vectors)
sorted ← sort_by_key(distinct)
first ← first_by_part(sorted)
bds ← get_boundary(sorted, first)
bds ←
  map b ∈ bds
    < (att, point), q > ← b
    EMIT < (att, (point, q)) >
  end map
(SM, BI) ← divide_atts(bds, mc)
sth ←
  map sa ∈ SM
    th ← select_ths(SM(sa), mb, mc)
    EMIT < (sa, th) >
  end map
bth ← ()
for all ba ∈ BI do
  bth ← bth + select_ths(ba, mb, mc)
end for
return(union(bth, sth))
```

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.

Boundary Points Selection

Algorithm 2 (*get_boundary*) 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 whether 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 point 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 *bds*.

In Algorithm 1 (line 14), the *bds* 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 into two groups according to the total number of candidate points by feature. The *divide_atts* function is then aimed to divide the tuples in two groups (*big* and *small*) depending on the number of candidate points by feature (count operation). Features in each group will be filtered and treated differently according to whether the total number of points for a given feature exceeds the threshold *mc* or not. Small features will be grouped by key so that these can be processed in a parallel way. The subsequent tuples are now reformatted as follows: (< *point*, *q* >).

MDLP Evaluation

Features in each group are evaluated differently from that mentioned before. Small features are evaluated in a single step where each feature corresponds with a single partition, whereas big features are evaluated

Algorithm 2: Function to generate the boundary points *(get_boundary)*

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

iteratively because 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 fact the number of candidate points for a single feature fits perfectly in one partition. In both cases, the *select_ths* 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_ths*). 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_ths*).

Algorithm 3 (*select_ths*) evaluates and selects the most promising cut points grouped by feature according to the MDLP criterion (single-step version). 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_ths*) explains the process that accumulates frequencies and then selects the minimum-entropy candidate. This version is more straightforward than the RDD version as it only needs to accumulate frequencies sequentially. First, 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*, *lq*, *rq* >), the algorithm evaluates the candidates using the *select_best* function.

Algorithm 5 (*rdd_select_ths*) explains the selection process; in this case for 'big' features (more than one partition). This process needs to be performed in a distributed manner because the number of candidate points exceeds the maximum size defined. For each feature, the subset of points is hence redistributed in a better partition scheme to homogenize the quantity of points by partition and node (*coalesce* function, line 12). 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, a new distributed process is started with the aim of computing the accumulated frequencies at points on 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 (the 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 values (line 7). Once these values are calculated (< *point*, *q*, *lq*, *rq* >), the algorithm evaluates all candidate points and their associated accumulators using the *select_best* function (as above).

Algorithm 6 evaluates the discretization schemes yielded by each point by computing the

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

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

1: $st \leftarrow enqueue(st, (candidates, ()))$
2: $result \leftarrow ()$
3: **while** $|st| > 0$ & $|result| < mb$ **do**
4: $(set, lth) \leftarrow dequeue(st)$
5: **if** $|set| > 0$ **then**
6: **if** $type(set) = 'array'$ **then**
7: $bd \leftarrow arr\_select\_ths(set, lth)$
8: **else**
9: $bd \leftarrow rdd\_select\_ths(set, lth, mc)$
10: **end if**
11: **if** $bd <> ()$ **then**
12: $result \leftarrow result + bd$
13: $(left, right) \leftarrow divide(set, bd)$
14: $st \leftarrow enqueue(st, (left, bd))$
15: $st \leftarrow enqueue(st, (right, bd))$
16: **end if**
17: **end if**
18: **end while**
19: $return(sort(result))$

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

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

```
1: total ← sum_freqs(cands)
2: lacc ← ()
3: for < point, q >∈ cands do
4:     lacc ← lacc + q
5:     freqs ← freqs+(point, q, lacc, total−lacc)
6: end for
7: return(select_best(cands, freqs))
```

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

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

```
1: npart ← round(|cands|/mc)
2: cands ← coalesce(cands, npart)
3: totalpart ←
4:   map partitions partition ∈ cands
5:     return(sum(partition))
6:   end map
7: total ← sum(totalpart)
8: freqs ←
9:   map partitions partition ∈ cands
10:    index ← get_index(partition)
11:    ltotal ← ()
12:    freqs ← ()
13:    for i = 0 until index do
14:      ltotal ← ltotal + totalpart(i)
15:    end for
16:    for all < point, q >∈ partition do
17:      freqs ← freqs + (point, q, ltotal + q, total − ltotal)
18:    end for
19:    return(freqs)
20:  end map
21: return(select_best(cands, freqs))
```

entropy for each partition generated, also taking into account the MDLP criterion. Thus, for each point,[d] the entropy is calculated for the two generated partitions (line 8) as well as the total entropy for the whole set (lines 12). Using these values, the entropy gain and the MDLP condition are computed for each point, according to Eq. (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 (small and big) are joined into the final point set of cut points.

Analysis of efficiency

In this section, we analyze the performance of the main operations that determined the overall performance of our proposal. Note that the first two operations are quite costly from the point of view of network usage, because they imply shuffling data across the cluster (wide dependencies). Nevertheless, once data are partitioned and saved, these remain unchanged. This is exploited by the subsequent steps, which take advantage of the data locality property. Having data partitioned also benefits operations such as groupByKey, where the grouping is performed locally. The list of such operations (showed in Algorithm 1) is presented below:

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

Input: $freqs$ An array/RDD of tuples ($< point, q, lq, rq >$), 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: $accp \leftarrow ()$
5: **for all** $< point, q, lq, rq > \in freqs$ **do**
6: $k1 \leftarrow |lq|; k2 \leftarrow |rq|$
7: $s1 \leftarrow sum(lq); s2 \leftarrow sum(rq);$
8: $ent1 \leftarrow ent(s1, k1); ent2 \leftarrow ent(s2, k2)$
9: $partent \leftarrow (s1 * ent1 + s2 * ent2)/s$
10: $gain \leftarrow totalent - partent$
11: $delta \leftarrow log_2(3^k - 2) - (k * hs - k1 * ent1 - k2 * ent2)$
12: $accepted \leftarrow gain > ((log_2(s-1)) + delta)/n$
13: **if** $accepted = true$ **then**
14: $accp \leftarrow accp + (partent, point)$
15: **end if**
16: **end for**
17: $return(min(accp))$

1. Distinct points (lines 1–10): this is a standard MapReduce operation that fetches all the points in the dataset. The map phase generates and distributes tuples using a hash partitioning scheme (linear distributed complexity). The reduce phase fetches the set of coincident points and sums up the class vectors (linear distributed complexity).
2. Sorting operation (line 11): this operation uses a more complex primitive of Spark: sortByKey. This samples the set and produces a set of bounds to partition this set. Then, a shuffling operation is started to redistribute the points according to the previous bounds. Once data are redistributed, a local sorting operation is launched in each partition (loglinear distributed order).
3. Boundary points (lines 12–13): this operation is in charge of computing the subset candidate of points to be evaluated. Thanks to the data partitioning scheme generated in the previous phases, the algorithm can yield the boundary points for all attributes in a distributed manner using a linear map operation.
4. Division of attributes (lines 14–19): once the reduced set of boundary points is generated, it is necessary to separate the attributes into two sets. To do that, several operations are started to complete this part. All these suboperations are performed linearly using distributed operations.
5. Evaluation of small attributes (lines 20–24): this is mainly formed by two suboperations: one for grouping the tuples by key (done locally thanks to the data locality), and one map operation to evaluate the candidate points. In the map operation, each feature starts an independent process that, like the sequential version, is quadratic. The main advantage here is the parallelization of these processes.
6. Evaluation of big features (lines 26–28): The complexity order for each feature is the same as in the previous case. However, in this case, the evaluation of features is done iteratively.

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.

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 involving 2% of positive instances. For this problem, the MapReduce version of the Random Over Sampling (ROS) algorithm presented in Ref 71 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 preprocessed and included in the LibSVM dataset repository.[72]

Table 2 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.), and the number of classes (#Cl) are shown. For evaluation purposes, Naïve Bayes[73] and two variants of Decision Tree[74]—with different impurity measures—have been chosen as reference in classification, using the distributed implementations included in MLlib library.[17] The recommended parameters of the classifiers, according to their authors' specification,[e] are shown in Table 3.

As evaluation criteria, we use two well-known evaluation metrics to assess the quality of the underlying discretization schemes. On one hand, *Classification accuracy* is used to evaluate the accuracy yielded by the classifiers—number of examples correctly labeled divided by the total number of examples. On the other hand, in order to prove the time benefits of 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 20 computing nodes and 1 master node. The computing nodes hold the following characteristics: 2 processors × 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,[f] 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 first author' GitHub repository.[g] The design of the algorithm has been adapted to be integrated in MLlib Library.

TABLE 2 | Summary Description for Classification Datasets

Dataset	#Train Ex.	#Test Ex.	#Atts.	#Cl.
Epsilon	400,000	100,000	2000	2
ECBDL14 (ROS)	65,003,913	2,897,917	631	2

TABLE 3 | Parameters of the Algorithms Used

Method	Parameters
Naive Bayes	Lambda = 1.0
Decision Tree—gini (DTg)	Impurity = gini, max depth = 5, max bins = 32
Decision Tree—entropy (DTe)	Impurity = entropy, max depth = 5, max bins = 32
Distributed MDLP	Max intervals = 50, max by partition = 100,000

MDLP, Minimum Description Length Principle.

Experimental Results and Analysis

Table 4 shows the classification accuracy results for both datasets.[b] According to these results, we can assert that using our discretization algorithm as a preprocessing step leads to an improvement in classification accuracy with Naïve Bayes, for the two datasets tested. It is especially relevant in ECBDL14 where there is an improvement of 5%. This shows the importance of discretization in the application of some classifiers such as Naïve Bayes. For the other classifiers, our algorithm is capable of producing the same competitive results as those performed implicitly by the decision trees.

Table 5 shows classification runtime values for both datasets distinguishing whether discretization is applied or not. As we can see, there is a slight improvement in both cases on using MDLP, but not enough significant. According to the previous results, we can state that the application of MDLP is relevant at least for epsilon, where the best accuracy result has been achieved by using Naïve Bayes and our discretizer. For ECBDL14, it is better to use the implicit discretization performed by the decision trees, because our algorithm is more time-consuming and obtains similar results.

Table 6 shows discretization time values for the two versions of MDLP, namely, sequential and distributed. For the sequential version on ECBDL14, the time value was estimated from small samples of this dataset, because its direct application is unfeasible. A graphical comparison of these two versions is shown in Figure 3. Comparing both implementations, we can notice the great advantage of using the distributed version against the sequential one. For ECBDL14, our version obtains a speedup

TABLE 4 | Classification Accuracy Values

Dataset	NB	NB-disc	DTg	DTg-disc	DTe	DTe-disc
ECBDL14	0.6276	**0.7260**	**0.7347**	0.7339	0.7459	**0.7508**
Epsilon	0.6550	**0.7065**	0.6616	**0.6623**	0.6611	**0.6624**

TABLE 5 | Classification Time Values: with Versus w/o Discretization (In Seconds)

Dataset	NB	NB-Disc	DTg	DTg-Disc	DTe	DTe-Disc
ECBDL14	31.06	**26.39**	347.76	**262.09**	281.05	**264.25**
Epsilon	5.72	**4.99**	68.83	**63.23**	74.44	**39.28**

TABLE 6 | Sequential Versus Distributed Discretization Time Values (In Seconds)

Dataset	Sequential	Distributed	Speedup Rate
ECBDL14	295,508	**1087**	271.86
Epsilon	5764	**476**	12.11

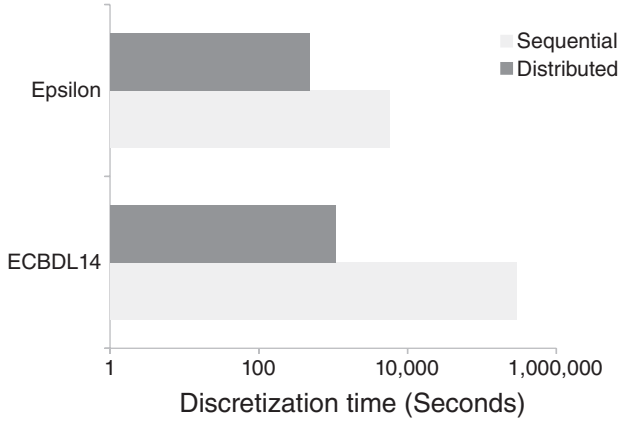

FIGURE 3 | Discretization time: sequential versus distributed (logaritmic scale).

ratio (*speedup* = *sequential/distributed*) of 271.86, whereas for epsilon, the ratio is equal to 12.11. This shows that the bigger the dataset, the higher the efficiency improvement; and, when the data size is large enough, the cluster can distribute fairly the computational burden across its machines. This is notably the case study of ECBDL14, where the resolution of this problem was found to be impractical using the original approach.

Discretization, as an important part in DM preprocessing, has raised general interest in recent years. In this work, we have presented an updated taxonomy and description of the most relevant algorithms in this field. The aim of this taxonomy is to help the researchers to better classify the algorithms that they use, on one hand, while also helping to identify possible new future research lines. At this respect, and although Big Data is currently a trending topic in science and business, no distributed approach has been developed in the literature, as we have shown in our taxonomy.

Here, we propose a completely distributed version of the MDLP discretizer with the aim of demonstrating that standard discretization methods can be parallelized in Big Data platforms, boosting both performance and accuracy. This version is capable of transforming the iterativity yielded by the original proposal in a single-step computation through a complete redesign of the original version. According to our experiments, our algorithm is capable of performing 270 times faster than the sequential version, improving the accuracy results in all used datasets. For future works, we plan to tackle the problem of discretization in large-scale online problems.

NOTES

[a] Logarithm in base 2 is used in this function.

[b] For a complete description of this model and other distributed models, please review Ref 69.

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

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

[e] https://spark.apache.org/docs/latest/api/scala/index.html.

[f] http://www.cloudera.com/content/cloudera/en/documentation/cdh5/v5-0-0/CDH5-homepage.html.

[g] https://github.com/sramirez/SparkFeatureSelection.

[h] In all tables, the best result by column (best by method) is highlighted in bold.

ACKNOWLEDGMENTS

This work is supported by the National Research Project TIN2014-57251-P, 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 and V. Bolón-Canedo acknowledge support of the Xunta de Galicia under postdoctoral Grant codes POS-A/2013/196 and ED481B 2014/164-0.

REFERENCES

1. Liu H, Hussain F, Lim Tan C, Dash M. Discretization: an enabling technique. *Data Min Knowl Discov* 2002, 6:393–423.
2. García S, Luengo J, Antonio Sáez J, López V, Herrera F. A survey of discretization techniques: taxonomy and empirical analysis in supervised learning. *IEEE Trans Knowl Data Eng* 2013, 25:734–750.
3. García S, Luengo J, Herrera F. *Data Preprocessing in Data Mining*. Germany: Springer; 2015.
4. Wu X, Kumar V, eds. *The Top Ten Algorithms in Data Mining*. USA: Chapman & Hall/CRC Data Mining and Knowledge Discovery; 2009.
5. Ross Quinlan J. *C4.5: Programs for Machine Learning*. USA: Morgan Kaufmann Publishers Inc.; 1993.
6. Agrawal R, Srikant R. Fast algorithms for mining association rules. In: *Proceedings of the 20th Very Large Data Bases conference (VLDB)*, Santiago de Chile, Chile, 1994, pages 487–499.
7. Yang Y, Webb GI. Discretization for Naïve-Bayes learning: managing discretization bias and variance. *Mach Learn* 2009, 74:39–74.
8. Hu H-W, Chen Y-L, Tang K. A dynamic discretization approach for constructing decision trees with a continuous label. *IEEE Trans Knowl Data Eng* 2009, 21:1505–1514.
9. Yang Y, Webb GI, Wu X. Discretization methods. In: *Data Mining and Knowledge Discovery Handbook*. Germany: Springer; 2010, 101–116.
10. Wu X, Zhu X, Wu G-Q, Ding W. Data mining with big data. *IEEE Trans Knowl Data Eng* 2014, 26:97–107.
11. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. In: San Francisco, CA, *OSDI*, 2004, pages 137–150.
12. Apache Hadoop Project. Apache Hadoop, 2015. [Online https://hadoop.apache.org/; Accessed March, 2015].
13. White T. *Hadoop, The Definitive Guide*. USA: O'Reilly Media, Inc.; 2012.
14. Apache Spark: lightning-fast cluster computing. Apache spark, 2015. [Online http://spark.apache.org/; Accessed March, 2015].
15. Hamstra M, Karau H, Zaharia M, Konwinski A, Wendell P. *Learning Spark: Lightning-Fast Big Data Analytics*. USA: O'Reilly Media, Incorporated; 2015.
16. Apache Mahout Project. Apache Mahout, 2015. [Online http://mahout.apache.org/; Accessed March, 2015].
17. Machine Learning Library (MLlib) for Spark. Mllib, 2015. [Online https://spark.apache.org/docs/1.2.0/mllib-guide.html; Accessed March, 2015].
18. Fayyad UM, Irani KB. Multi-interval discretization of continuous-valued attributes for classification learning. In: *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI)*, San Francisco, CA, 1993, pages 1022–1029.
19. Wong AKC, Chiu DKY. Synthesizing statistical knowledge from incomplete mixed-mode data. *IEEE Trans Pattern Anal Mach Intell* 1987, 9:796–805.
20. Chou PA. Optimal partitioning for classification and regression trees. *IEEE Trans Pattern Anal Mach Intell* 1991, 13:340–354.
21. Catlett J. On changing continuous attributes into ordered discrete attributes. In: *European Working Session on Learning (EWSL)*. Lecture Notes on Computer Science, vol. 482. Germany: Springer-Verlag; 1991, 164–178.
22. Kerber R. Chimerge: discretization of numeric attributes. In: *National Conference on Artifical Intelligence American Association for Artificial Intelligence (AAAI)*, San Jose, California, 1992, pages 123–128.
23. Holte RC. Very simple classification rules perform well on most commonly used datasets. *Mach Learn* 1993, 11:63–90.
24. Ching JY, Wong AKC, Chan KCC. Class-dependent discretization for inductive learning from continuous and mixed-mode data. *IEEE Trans Pattern Anal Mach Intell* 1995, 17:641–651.

25. Pfahringer B. Compression-based discretization of continuous attributes. In: *Proceedings of the 12th International Conference on Machine Learning (ICML)*, Tahoe City, California, 1995, pages 456–463.
26. Xindong W. A Bayesian discretizer for real-valued attributes. *Comput J* 1996, 39:688–691.
27. Friedman N, Goldszmidt M. Discretizing continuous attributes while learning Bayesian networks. In: *Proceedings of the 13th International Conference on Machine Learning (ICML)*, Bari, Italy, 1996, pages 157–165.
28. Chmielewski MR, Grzymala-Busse JW. Global discretization of continuous attributes as preprocessing for machine learning. *Int J Approx Reason* 1996, 15:319–331.
29. Ho KM, Scott PD. Zeta: a global method for discretization of continuous variables. In: *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining (KDD)*, Newport Beach, California, 1997, pages 191–194.
30. Cerquides J, De Mantaras RL. Proposal and empirical comparison of a parallelizable distance-based discretization method. In: *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining (KDD)*, Newport Beach, California, 1997, pages 139–142.
31. Liu H, Setiono R. Feature selection via discretization. *IEEE Trans Knowl Data Eng* 1997, 9:642–645.
32. Hong SJ. Use of contextual information for feature ranking and discretization. *IEEE Trans Knowl Data Eng* 1997, 9:718–730.
33. Zighed DA, Rabaséda S, Rakotomalala R. FUSINTER: a method for discretization of continuous attributes. *Int J Unc Fuzz Knowl Based Syst* 1998, 6:307–326.
34. Bay SD. Multivariate discretization for set mining. *Knowl Inform Syst* 2001, 3:491–512.
35. Tay FEH, Shen L. A modified chi2 algorithm for discretization. *IEEE Trans Knowl Data Eng* 2002, 14:666–670.
36. Giráldez R, Aguilar-Ruiz JS, Riquelme JC, Ferrer-Troyano FJ, Rodríguez-Baena DS. Discretization oriented to decision rules generation. In: *Frontiers in Artificial Intelligence and Applications*, vol. 82. Netherlands: IOS press; 2002, 275–279.
37. Boulle M. Khiops: a statistical discretization method of continuous attributes. *Mach Learn* 2004, 55:53–69.
38. Kurgan LA, Cios KJ. CAIM discretization algorithm. *IEEE Trans Knowl Data Eng* 2004, 16:145–153.
39. Chao-Ton S, Hsu J-H. An extended chi2 algorithm for discretization of real value attributes. *IEEE Trans Knowl Data Eng* 2005, 17:437–441.
40. Liu X, Wang H. A discretization algorithm based on a heterogeneity criterion. *IEEE Trans Knowl Data Eng* 2005, 17:1166–1173.
41. Mehta S, Parthasarathy S, Yang H. Toward unsupervised correlation preserving discretization. *IEEE Trans Knowl Data Eng* 2005, 17:1174–1185.
42. Boullé M. MODL: a Bayes optimal discretization method for continuous attributes. *Mach Learn* 2006, 65:131–165.
43. Au W-H, Chan KCC, Wong AKC. A fuzzy approach to partitioning continuous attributes for classification. *IEEE Trans Knowl Data Eng* 2006, 18:715–719.
44. Lee C-H. A Hellinger-based discretization method for numeric attributes in classification learning. *Knowl Based Syst* 2007, 20:419–425.
45. Wu QX, Bell DA, Prasad G, McGinnity TM. A distribution-index-based discretizer for decision-making with symbolic AI approaches. *IEEE Trans Knowl Data Eng* 2007, 19:17–28.
46. Ruiz FJ, Angulo C, Agell N. IDD: a supervised interval Distance-Based method for discretization. *IEEE Trans Knowl Data Eng* 2008, 20:1230–1238.
47. Tsai C-J, Lee C-I, Yang W-P. A discretization algorithm based on class-attribute contingency coefficient. *Inform Sci* 2008, 178:714–731.
48. González-Abril L, Cuberos FJ, Velasco F, Ortega JA. Ameva: an autonomous discretization algorithm. *Expert Syst Appl* 2009, 36:5327–5332.
49. Jin R, Breitbart Y, Muoh C. Data discretization unification. *Knowl Inform Syst* 2009, 19:1–29.
50. Li M, Deng S, Feng S, Fan J. An effective discretization based on class-attribute coherence maximization. *Pattern Recognit Lett* 2011, 32:1962–1973.
51. Gethsiyal Augasta M, Kathirvalavakumar T. A new discretization algorithm based on range coefficient of dispersion and skewness for neural networks classifier. *Appl Soft Comput* 2012, 12:619–625.
52. Shehzad K. EDISC: a class-tailored discretization technique for rule-based classification. *IEEE Trans Knowl Data Eng* 2012, 24:1435–1447.
53. Ferreira AJ, Figueiredo MAT. An unsupervised approach to feature discretization and selection. *Pattern Recognit* 2012, 45:3048–3060.
54. Kurtcephe M, Altay Güvenir H. A discretization method based on maximizing the area under receiver operating characteristic curve. *Intern J Pattern Recognit Artif Intell* 2013, 27.
55. Ferreira AJ, Figueiredo MAT. Incremental filter and wrapper approaches for feature discretization. *Neurocomputing* 2014, 123:60–74.
56. Yan D, Liu D, Sang Y. A new approach for discretizing continuous attributes in learning systems. *Neurocomputing* 2014, 133:507–511.
57. Sang Y, Qi H, Li K, Jin Y, Yan D, Gao S. An effective discretization method for disposing high-dimensional data. *Inform Sci* 2014, 270:73–91.

58. Nguyen H-V, Müller E, Vreeken J, Böhm K. Unsupervised interaction-preserving discretization of multivariate data. *Data Min Knowl Discov* 2014, 28:1366–1397.

59. Jiang F, Sui Y. A novel approach for discretization of continuous attributes in rough set theory. *Knowl Based Syst* 2015, 73:324–334.

60. Moskovitch R, Shahar Y. Classification-driven temporal discretization of multivariate time series. *Data Min Knowl Discov* 2015, 29:871–913.

61. Ramírez-Gallego S, García S, Benítez JM, Herrera F. Multivariate discretization based on evolutionary cut points selection for classification. *IEEE Trans Cybern*. In press. doi:10.1109/TCYB.2015.2410143.

62. Shannon CE. A mathematical theory of communication. ACM *SIGMOBILE Mob Comput Commun Rev* 2001, 5:3–55.

63. Alcalá-Fdez J, Sánchez L, García S, del Jesus MJ, Ventura S, Garrell JM, Otero J, Romero C, Bacardit J, Rivas VM, et al. KEEL: a software tool to assess evolutionary algorithms for data mining problems. *Soft Comput* 2009, 13:307–318.

64. Verónica Bolón-Canedo, Noelia Sánchez-Maroño, and Amparo Alonso-Betanzos. On the effectiveness of discretization on gene selection of microarray data. In: *International Joint Conference on Neural Networks, IJCNN 2010*, Barcelona, Spain, 18–23 July, 2010, pages 1–8.

65. Bolón-Canedo V, Sánchez-Maroño N, Alonso-Betanzos A. Feature selection and classification in multiple class datasets: an application to KDD Cup 99 dataset. *Expert Syst Appl* May 2011, 38:5947–5957.

66. Cano A, Ventura S, Cios KJ. Scalable CAIM discretization on multiple GPUs using concurrent kernels. *J Supercomput* 2014, 69:273–292.

67. Zhang Y, Cheung Y-M. Discretizing numerical attributes in decision tree for big data analysis. In: *ICDM Workshops*, Shenzhen, China, 2014, pages 1150–1157.

68. Beyer M.A., Laney D. 3D data management: controlling data volume, velocity and variety, 2001. [Online http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf; Accessed March, 2015].

69. Fernández A, del Río S, López V, Bawakid A, del Jesús MJ, Benítez JM, Herrera F. Big data with cloud computing: an insight on the computing environment, mapreduce, and programming frameworks. *WIREs Data Min Knowl Discov* 2014, 4:380–409.

70. Lin J. Mapreduce is good enough? If all you have is a hammer, throw away everything that's not a nail!. *Clin Orthop Relat Res* 2012, abs/1209.2191.

71. Rio S, Lopez V, Benitez JM, Herrera F. On the use of mapreduce for imbalanced big data using random forest. *Inform Sci* 2014, 285:112–137.

72. Chang C-C, Lin C-J. LIBSVM: a library for support vector machines. *ACM Trans Intell Syst Technol* 2011, 2:1–27. Datasets available at http://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets/.

73. Duda RO, Hart PE. *Pattern Classification and Scene Analysis*, vol. 3. New York: John Wiley & Sons; 1973.

74. Quinlan JR. Induction of decision trees. In: Shavlik JW, Dietterich TG, eds. *Readings in Machine Learning*. Burlington, MA: Morgan Kaufmann Publishers; 1990. Originally published in *Machine Learning* 1:81–106, 1986.