Open access • Proceedings Article • DOI:10.1109/FUZZ-IEEE.2015.7337868

A MapReduce-based fuzzy associative classifier for big data — Source link

Pietro Ducange, Francesco Marcelloni, Armando Segatori

Institutions: Università degli Studi eCampus, University of Pisa

Published on: 01 Aug 2015 - IEEE International Conference on Fuzzy Systems

Topics: Fuzzy set operations, Neuro-fuzzy, Fuzzy logic, Fuzzy set and Set (abstract data type)

A MapReduce-based Fuzzy Associative Classifier for Big Data

Pietro Ducange
Facoltà di Ingegneria
eCampus University
Novedrate, Italy 22060
Email: pietro.ducange@uniecampus.it

Francesco Marcelloni
Dip. di Ingegneria dell'Informazione
University of Pisa
Pisa, Italy 56122
Email: francesco.marcelloni@unipi.it

Armando Segatori
Dip. di Ingegneria dell'Informazione
University of Pisa
Pisa, Italy 56122
Email: armando.segatori@for.unipi.it

***Abstract*—In this paper, we propose an efficient distributed fuzzy associative classification model based on the MapReduce paradigm. The learning algorithm first mines a set of fuzzy association classification rules by employing a distributed version of a fuzzy extension of the well-known FP-Growth algorithm. Then, it prunes this set by using three purposely adapted types of pruning.**

We implemented the distributed fuzzy associative classifier using the Hadoop framework. We show the scalability of our approach by carrying out a number of experiments on a real-world big dataset. In particular, we evaluate the achievable speedup on a small computer cluster, highlighting that the proposed approach allows handling big datasets even with modest hardware support.

I. Introduction

Association rule mining has become a very popular method for generating highly accurate classification models, called *associative classifiers* (ACs). Such models have been extensively studied in the literature [1], [2], [3] and have been recently exploited in a large number of real world applications, such as detection phishing activities in websites [4], and text analysis [5].

The generation of an associative classifier is typically performed in two steps. First, a set of classification association rules (CARs) is mined from the training set. Then a subset of high quality CARs is selected by pruning redundant or noisy information. The selected CARs are used to predict the class labels when the model is used for classifying unlabeled patterns.

Even though associative classifiers are often "interpretable" by the user and ensure high accuracy in pattern classification, these models suffer from two main weaknesses [6]. First, association rule mining algorithms deal with binary or categorical itemsets. On the other hand, real data objects are often described by numerical continuous features. Thus, appropriate discretization algorithms have to be applied to transform continuous feature domains into a set of items. The discretization process determines the data ranges (*bin boundaries*) and the number of bins. Then, each value is assigned to the bin which contains it. Since transitions between bins are generally gradual, recently a number of associative classification approaches have proposed to adopt fuzzy boundaries, thus leading to the so-called *fuzzy associative classifiers* (FACs) [7], [8], [9].

Second, when the number of training data objects is huge, the complexity of the learning process grows exponentially in terms of both time and memory. This issue is much more evident when dealing with classification datasets belonging to the so-called *big data* [10]. We recall that big data are mainly defined by three characteristics: volume, variety and velocity. Whenever these three characteristics increase, the classical software tools and technologies lose their capability of capturing, storing, managing and analyzing data. Indeed, in the framework of big data classification, classical algorithms for building classification models are practically inapplicable: the design and implementation of novel learning algorithms for dealing with big data is currently a challenging task [11].

When dealing with FACs another drawback occurs: the use of fuzzy partitions makes the fuzzy CAR mining process more complex. Indeed, while in the case of crisp partitions an input value supports a unique item, in the case of fuzzy partitions, an input value can support more than one fuzzy item. Thus, the number of possible fuzzy association rules is higher than the number of possible crisp rules. The approaches proposed so far in the literature for generating fuzzy association rules have limited the complexity by considering only the most frequent fuzzy item for each attribute [12], [13], thus achieving a good trade-off between the number of association rules and the amount of information described by these rules.

In this paper, we propose an efficient fuzzy associative classification scheme for dealing with big data. The proposed approach, based on the MapReduce paradigm [14], is a distributed version of our Associative Classifier based on a Fuzzy Frequent Pattern (AC-FFP) mining algorithm, recently introduced in [9]. We first mine fuzzy associative classification rules by employing a distributed implementation of the fuzzy frequent pattern mining algorithm proposed in AC-FFP [9]: this algorithm is a fuzzy extension of the well-known FP-Growth algorithm [15]. Then, we select a set of fuzzy CARs by performing a distributed pruning step. In the specialized literature, a number of association rule mining approaches, mainly based on the MapReduce paradigm, have been proposed [16], [17]. In particular, recent distributed implementations of Apriori [18], [19] and FP-Growth [20], [21] algorithms have been discussed. To the best of our knowledge, no method has however investigated distributed implementations of fuzzy association rule mining algorithms and their integration into classification models for handling big data.

We implemented the proposed approach on the Hadoop framework [22]. We carried out experiments on a real-world big dataset with 5 millions instances for evaluating the scalability and the achievable speedup of our distributed approach on a small computer cluster. We highlighted how our approach turns to be suitable to practically address big datasets even with modest hardware support.

The paper is organized as follows. Section II provides some preliminaries on fuzzy associative classifiers and MapReduce. Section III briefly describes the AC-FFP algorithm and then explains the distributed approach, including the details of each single phase that runs on the computer cluster. Section IV presents the experimental setup and discusses the results in terms of speedup and scalability. Finally, in Section V, we draw final conclusions.

II. Preliminaries

A. Fuzzy Associative Classifiers

Association rules are rules in the form $Z \rightarrow Y$, where Z and Y are set of items. These rules describe relations among items in a dataset [23] and have been widely employed in the market basket analysis [24].

In the fuzzy associative classification context, given a set of attributes $\mathbf{X} = \{X_1, \ldots, X_f, \ldots, X_F\}$ and a fuzzy partition $P_f = \{A_{f,1}, \ldots, A_{f,T_f}\}$ of T_f fuzzy sets defined for each attribute X_f, the single item is defined as the couple $I_{f,j} = (X_f, A_{f,j})$, where $A_{f,j}$ is one of the fuzzy sets defined in the partition P_f of variable X_f, $f = 1, ..., F$. A generic fuzzy classification association rule (FCAR) is expressed as:

$$FCAR_m : FAnt_m \rightarrow C_{j_m} \tag{1}$$

where C_{j_m} is the class label selected for the rule within the set $C = \{C_1, \ldots, C_L\}$ of possible classes and $FAnt_m$ is a conjunction of items. More familiarly, rule $FCAR_m$ can be represented as:

$$FCAR_m : \textbf{IF } X_1 \textbf{ is } A_{1,j_{m,1}} \textbf{ AND} \ldots \textbf{AND } X_F \textbf{ is } A_{F,j_{m,F}} \textbf{ THEN } Y \textbf{ is } C_{j_m}$$

where Y is the classifier output.

The rule base of an FAC contains a set of M FCARs. Each rule $FCAR_m$ in the rule base has associated a rule weight RW_m, which expresses a certainty degree of the classification in the class C_{j_m} for a pattern belonging to the fuzzy subspace delimited by the antecedent of rule $FCAR_m$.

Let $\mathbf{o}_n = (\mathbf{x}_n, y_n)$ be the n^{th} object in the training set T, with $\mathbf{x}_n = [x_{n,1} \ldots, x_{n,F}] \in \Re^F$ and $y_n \in C$. The strength of activation (*matching degree* of the rule with the input) of rule $FCAR_m$ is calculated as:

$$w_m(\mathbf{x_n}) = \prod_{f=1}^{F} A_{f,j_{m,f}}(x_{n,f}), \tag{2}$$

where $A_{f,j_{m,f}}(x)$ is the membership function associated with the fuzzy set $A_{f,j_{m,f}}$.

The *association degree* $h_m(\mathbf{x}_n)$ with the class C_{j_m} is calculated as

$$h_m(\mathbf{x}_n) = w_m(\mathbf{x}_n) \cdot RW_m \tag{3}$$

Different definitions have been proposed for the rule weight RW_m [25]. As discussed in [26], the rule weight of each fuzzy rule $FCAR_m$ can improve the performance of FACs. In this paper, we adopt the fuzzy confidence value, or certainty factor CF, defined as follows:

$$RW_m = CF_m = \frac{\sum_{\mathbf{x}_n \in C_{j_m}} w_m(\mathbf{x}_n)}{\sum_{n=1}^{N} w_m(\mathbf{x}_n)} \tag{4}$$

where N is the number of objects contained in the training set T.

In the association rule analysis, support and confidence are the most common measures to determine the strength of an association rule. Support and confidence of a rule $FCAR_m$ can be expressed as follows:

$$fuzzySupp(FAnt_m \rightarrow C_{j_m}) = \frac{\sum_{\mathbf{x}_n \in C_{j_m}} w_m(\mathbf{x}_n)}{N} \tag{5}$$

$$fuzzyConf(FAnt_m \rightarrow C_{j_m}) = \frac{\sum_{x_n \in C_{j_m}} w_m(\mathbf{x}_n)}{\sum_{\mathbf{x}_n \in T} w_{Ant_m}(\mathbf{x}_n)} \tag{6}$$

where N is the number of objects in the training set T, $w_m(\mathbf{x}_n)$ is the matching degree of rule $FCAR_m$ and $w_{FAnt_m}(\mathbf{x}_n)$ is the matching degree of all the rules which have the antecedent equal to $FAnt_m$.

B. MapReduce

In 2004, Google proposed the MapReduce programming framework [14] to divide the computation flow into a set of independent tasks and distribute them across large-scale clusters of machines. MapReduce is a programming model based on functional programming that divides the computational flow into two main phases: *Map* and *Reduce*. The overall computation is organized around $\langle key, value \rangle$ pairs: it takes a set of *input* $\langle key, value \rangle$ pairs and produces a set of *output* $\langle key, value \rangle$ pairs.

To execute a user program, the MapReduce execution environment distributes the independent Z *map tasks* and R *reduce tasks* across the computer cluster. While the number of map tasks is automatically determined by the number of the input splits for partitioning the dataset, the number of reduce tasks is defined by the user. Each map task defined by the user reads the contents of the corresponding input split formatted as $\langle key_1, value_1 \rangle$ pairs and produces a list of intermediate $\langle key_2, value_2 \rangle$ pairs as output: $map(key_1, value_1) \rightarrow list(key_2, value_2)$. The environment groups and sorts all the map task output data according to the intermediate keys key_2, and then, for each unique key, passes the computational flow to the reduce tasks. Each reduce task processes the key and the associated value list as input and generates a new list of values as output: $reduce(key_2, list(value_2)) \rightarrow list(value_3)$. Finally, the output of each reduce task is appended to the output final file.

In the last years, several open source projects have been developed to handle massive data, but the most popular open source execution environment for the MapReduce paradigm is Apache Hadoop [22]. It allows the execution of custom applications that rapidly process big datasets stored in its distributed file system, called Hadoop Distributed Filesystem (HDFS).

III. The Proposed Algorithm

In this section, first we introduce the AC-FFP mining algorithm we proposed in [9]. Then, we describe the distributed approach in detail.

A. AC-FFP

AC-FFP is a classifier which exploits a fuzzy version of the well-known FP-Growth [15] algorithm for generating the fuzzy association rules. In AC-FFP, the rule learning is performed in three phases: (i) discretization, (ii) fuzzy FCAR mining and (iii) FCAR pruning. In the following, for the sake of brevity, we introduce just a short description of AC-FFP: more information can be found in [9].

In the first phase, the discretization of continuous attributes is performed by using the multi-interval discretization algorithm based on entropy proposed by Fayyad and Irani [27]. For each feature X_f, the algorithm outputs a set of bin boundaries $\{b_{f,1}, \ldots, b_{f,Q_f}\}$, where $\forall r \in [1, \ldots, Q_f - 1], b_{f,r} < b_{f,r+1}$, and $b_{f,1}$ and b_{f,Q_f} are, respectively, the minimum and the maximum values in the universe of X_f. For each pair $(b_{f,r}, b_{f,r+1}$ of bin boundaries, we compute the middle point $m_{f,r}$ as $m_{f,r} = \frac{b_{f,r}+b_{f,r+1}}{2}$. Then, as shown in Figure 1, a strong fuzzy partition P_f is generated by defining triangular fuzzy sets with the cores positioned in correspondence to each bin boundary and middle point. At the end of the discretization phase, $T_f = 2 \cdot Q_f - 1$ fuzzy sets are defined for each feature. If no bin boundary has been found by the algorithm for feature X_f, then no fuzzy set is generated and the feature is discarded.

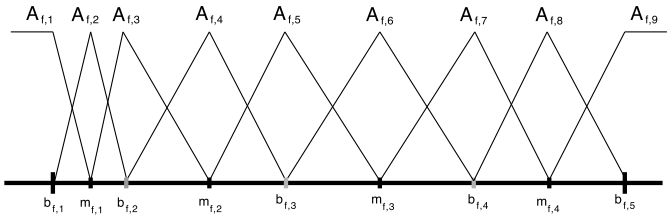

Fig. 1. An example of strong fuzzy partition obtained by the fuzzification of the output of the Fayyad and Irani's discretizer.

In the second step, the algorithm mines FCARs from the training set by employing a fuzzy version of the well-known FP-Growth algorithm for mining frequent patterns. The generation of the rules involves two scans of the overall training set and is very similar to the classic FP-Growth [15] with the only difference that the items correspond to fuzzy sets. In the first scan, the AC-FFP computes the fuzzy support of each fuzzy sets $A_{f,j}$, defined as

$$fuzzySupp(A_{f,j}) = \frac{\sum_{n=1}^{N} A_{f,j}(x_{f,n})}{N} \tag{7}$$

where $x_{f,n}$ is the value of the f^{th} feature of the object $\mathbf{o}_n$.

Only the fuzzy sets (*frequent fuzzy sets*), which have a support higher than the support threshold *minSup*, are retained. The frequent fuzzy sets are sorted in descending order according to their support and organized in a list, named f_{list}. In the second scan, the dataset consisting only of frequent items is compressed into a frequent pattern tree, called FP-Tree. Then, FP-Growth recursively mines patterns by dividing the compressed dataset into a set of projected datasets, each associated with a frequent item or a pattern fragment. For each pattern fragment, only its associated conditional dataset needs to be examined. Thus, the problem of mining frequent itemsets is converted into building and searching trees recursively.

To limit the complexity, the generation of the FP-Tree is performed by transforming the object $\mathbf{o}_n$ into a fuzzy object $\widetilde{\mathbf{o}}_n = (\widetilde{\mathbf{x}}_n, y_n)$ where $\widetilde{\mathbf{x}}_n = \{A_{1_n,j_{1_n}}, \ldots, A_{Z_n,j_{Z_n}}\}$ and $A_{i_n,j_{i_n}}$, $i_n \in [1, \ldots, F]$, $j_{i_n} \in [1, \ldots, T_{i_n}]$, indicates the frequent fuzzy value selected for the feature i_n. The values in $\widetilde{\mathbf{x}}_n$ are sorted according to the f_{list}. Note that, for strong fuzzy partitions, each $\mathbf{x}_n$ would generate 2^F possible patterns since each $x_{f,n}$ value on the universe of continuos feature belongs to two fuzzy sets. Thus, to limit the number of possible patterns, the $\widetilde{\mathbf{x}}_n$ contains only the fuzzy sets with the highest matching degree from the two fuzzy sets associated with value $\mathbf{x}_{f,n}$. If the selected fuzzy set is not present in the f_{list}, then it is discarded. Obviously, the number of features, which describe $\widetilde{\mathbf{x}}_n$, can be lower than F. As in the classic FP-Growth, the fuzzy object is added to the FP-Tree, considering the frequent fuzzy sets in $\widetilde{\mathbf{x}}_n$ as labels of the FP-Tree nodes. Note that whether a node already exists in the tree, the corresponding counter is simply incremented by 1. This approach achieves a good trade-off between complexity and quality of the rules. Indeed, each rule $FCAR_m$ mined from the FP-Tree represents the rule with the highest matching degree for the specific object $\mathbf{o}_n$. Other rules, which could be mined from $\mathbf{o}_n$, would have a lower matching degree and probably would be pruned. Only the rules with support, confidence and χ^2 value higher than, respectively, *minSup*, *minConf* and $min\chi^2$ thresholds are maintained.

The last step, which requires two additional scans of the overall training set, aims to prune redundant rules or noise information generated in the previous phase by performing three different types of pruning. The first one, which involves the third scan of the dataset, prunes rules with fuzzy support and confidence lower than *minFuzzySupp* and *minFuzzyConf*, respectively. These thresholds correspond to *minSupp* and *minConf* adapted to the number of conditions and number of instances of each class, respectively, so as to take into account both the effect of the product t-norm as conjunction operator and the imbalance of datasets. They are calculated respectively as follows:

$$minFuzzySupp_g = minSupp \cdot 0.5^{g-1} \tag{8}$$

$$minFuzzyConf_{C_j} = minConf \cdot \frac{N_{C_j}}{N_{MajorityClass}} \tag{9}$$

where $minSupp$ is the minimum support determined by the expert and $g \in [1..F]$ is the rule length, $N_{MajorityClass}$ is the number of occurrences of the majority class label in the data set, N_{C_j} is the number of occurrences of the consequent class C_j in the training set and $minConf$ is the minimum confidence fixed by the expert. In the second type of pruning, redundant rules are removed. First, the AC-FFP generates a rule ranking by sorting the rules according to their confidence, fuzzy support, and the number of antecedents, respectively. Second, if exists an $FCAR_m$ with lower rank and more specific than another fuzzy rule $FCAR_l$, then $FCAR_m$ is

pruned. In the third type of pruning, which involves the last scan of the training set, the AC-FFP maintains only the fuzzy rules that correctly classify at least one data object by performing the *training set coverage* step. Indeed, only those rules $FCAR_m$ with matching degree higher than the *fuzzy matching degree threshold* $\overline{w}_m = 0.5^{g_m - 1}$ where g_m is the rule length of $FCAR_m$, are considered. At the end of the four scans of the training set, the selected rules are inserted into the rule base and they are used to classify unlabeled patterns.

The AC-FFP adopts the weighted vote [28] as the *reasoning method*. Given an input pattern $\widehat{\mathbf{x}} = [\hat{x}_1 \ldots, \hat{x}_F]$, each fuzzy rule in the RB gives a vote for its consequent class as follows:

$$V_{C_k}(\widehat{\mathbf{x}}) = \sum_{FCAR_m \in RB; C_{j_m} = C_k} w_m(\widehat{\mathbf{x}}) \cdot 2^{g_m} \cdot CF_m \quad (10)$$

where $w_m(\widehat{\mathbf{x}})$ is the matching degree of $FCAR_m$ for the input $\widehat{\mathbf{x}}$, g_m is the number of antecedent conditions of $FCAR_m$ and CF_m is the certainty factor. As stated in Section II, the AC-FFP uses the confidence of the rule as certainty factor. The input pattern is classified into the class corresponding to the maximum total strength of vote or as *unknown*, if $\widehat{\mathbf{x}}$ activates no rule.

B. The Distributed Approach

In order to deal with big data, we implemented a distributed version of the AC-FFP algorithm (denoted as *DAC-FFP* in the following), which is based on the MapReduce paradigm. In particular, we parallelized the last two steps of the AC-FFP. The discretization process and the generation of the fuzzy partitions are performed, as in the first step of the AC-FFP algorithm, by using a reduced training set, obtained by randomly extracting a percentage S of objects from the overall training set. Once the fuzzy partitions have been defined, the DAC-FFP distributes on the cluster nodes the fuzzy CARs mining and pruning steps.

The distributed fuzzy CAR mining approach extracts, for each class label, K non-redundant fuzzy association rules, characterized by the highest confidence, by performing two scans of the overall training set. The implementation is based on the Parallel FP-Growth (PFP-Growth) proposed by Li et al. [20] for efficiently parallelizing the frequent patterns mining without generating candidate item sets. First, the algorithm selects the frequent items, builds the f_{list} and then distributes *item-projected datasets* on each node for building local and independent FP-Trees. An *item-projected dataset* $T(I_{f,j})$ contains only objects, also called *item-projected objets*, where the items are sorted according to the f_{list} and the items with lower support than $I_{f,j}$ are removed. In the last phase, the algorithm aggregates the results and, for each item, selects the highest supported patterns.

We adapted the PFP-Growth algorithm to generate frequent FCARs characterized by a high fuzzy confidence. As shown in Figure 2, the overall fuzzy FCAR mining process is carried out by performing three MapReduce phases: (i) *parallel fuzzy counting*, (ii) *parallel fuzzy FP-Growth*, and (iii) *parallel rules selection*.

The *parallel fuzzy counting* phase scans the dataset and counts both the fuzzy support for selecting the *frequent fuzzy sets*, and the number of occurrences of each class label. A

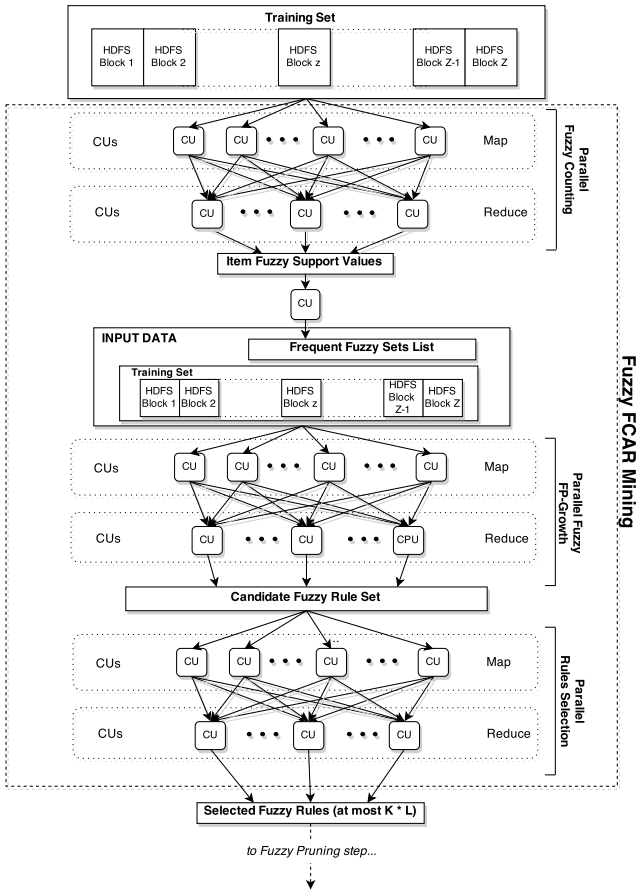

Fig. 2. The overall FCAR Mining process of the DAC-FFP algorithm.

fuzzy set is frequent if its fuzzy support is higher than a minimum threshold *minSup* fixed by the expert. The MapReduce framework divides the entire training set into blocks and assigns each of them to a map task. Each map task is fed by input key-value pairs represented as $\langle key = r, value = \mathbf{o}_r \rangle$, where $\mathbf{o}_r = (\mathbf{x}_r, y_r)$ is the r^{th} object of the training set block. For each fuzzy set $A_{f,j} \in P_f$, the mapper outputs a key-value pair $\langle key = A_{f,j}, value = A_{f,j}(x_{r,f}) \rangle$, where $A_{f,j}(x_{r,f})$ is the membership degree of the f^{fh} component of $\mathbf{x}_r$ to the j^{th} fuzzy set of partition P_f. Obviously, only the fuzzy sets with matching degree higher than zero are considered. Since we use strong partitions, for each $\mathbf{x}_r$, we output at most $2F$ values. Finally, the mapper outputs also the key-value pair $\langle key = y_r, value = 1 \rangle$. The reducer is fed by a list of corresponding values for each key: a set of membership degrees for the fuzzy sets, and a set of 1's for the class labels. The reducer input is formatted as $\langle key = A_{f,j}, value = list(A_{f,j}) \rangle$ and $\langle key = C_j, value = list(C_j) \rangle$, respectively, and outputs $\langle key = A_{f,j}, value = fuzzySupp(A_{f,j}) \rangle$, where $fuzzySupp(A_{f,j})$ is calculated according to the Eq. 7, and $\langle key = C_j, value = size(list(C_j)) \rangle$.

At the end of the first MapReduce phase, the algorithm selects only the fuzzy sets whose support is larger than the support threshold $minSup$, stores them in the f_{list}, sorting in descending order according the fuzzy support, and prunes the other ones. Only the frequent fuzzy sets will be considered in the subsequent phases. Since f_{list} is generally small, this step

can efficiently be performed on a single machine.

The second MapReduce phase, *parallel fuzzy FP-Growth*, mines fuzzy CARs whose support, confidence and χ^2 values are higher than *minSup*, *minConf* and $min\chi^2$ thresholds, respectively. The approach is very similar to the classical parallel FP-Growth described by Li et al. [20] with the only difference that we extend the projected dataset and projected object concepts to the fuzzy context. Indeed, each mapper computes the *item projected objects* so that the reducers are able to build the *item-projected datasets*, and then to generate local conditional FP-Trees. We recall that in our case an item is defined as $I_{f,j} = (X_f, A_{f,j})$. In the following we denote the $A_{f,j}$*-projected dataset* as $T(A_{f,j})$. Since, the local conditional FP-Trees are independent of each other, the fuzzy CARs can be mined by each node independently of the other nodes.

As in the first phase, each mapper reads a key-value pair $\langle key = r, value = \mathbf{o}_r \rangle$ in the training set block, and then builds the fuzzy object $\widetilde{\mathbf{o}}_r$ from $\mathbf{o}_r$. As stated in Section III-A, for each $x_{r,f}$ value, the mapper extracts the fuzzy sets $A_{f,j}$ with the highest matching degree from the two fuzzy sets activated by $x_{r,f}$. Then, the mapper sorts the fuzzy sets according to the f_{list}. If a fuzzy set is not present in the f_{list} then it is discarded. Finally, the mapper retrieves the class label C_{j_n} and, for each extracted fuzzy set, outputs the key-value pair $\langle key = index_{A_{f,j}}, value = A_{f,j} - projected\ object \rangle$, where $index_{A_{f,j}}$ is the index of the fuzzy set $A_{f,j}$ in the f_{list}.

The MapReduce framework groups all the item-projected objects with the same index, and passes them to the reducer. Each reducer is able to process the $A_{f,j}$-projected training sets independently, and generates the associated FCARs. Indeed, the reducer receives in input key-value pairs $\langle key = index_{A_{f,j}}, value = T(A_{f,j}) \rangle$, builds the local conditional FP-Tree and recursively mines the FCARs, as described in [15]. As in the AC-FFP, if a node already exists in the tree, the corresponding counter is simply incremented by 1 and no other information about the matching degrees are maintained.

Finally, when rules are extracted from the FP-Tree, only those FCARs whose support, confidence and χ^2 values are greater than the relative thresholds are considered. In particular, reducers output $\langle key = null, value = FCAR_m \rangle$ pairs, where $FCAR_m$ is the m^{th} generated rule. Note that the MapReduce framework automatically determines the number of conditional FP-Trees assigned to each reducer through the default partition function $hash(key)\ mod\ R$, where R is the number of reducers. Even though our implementation ensures more or less the same number of conditional FP-Trees processed by each reducer, such distribution does not necessarily guarantee a perfect load balancing among all the nodes, because the time spent in processing each specific FP-Tree depends on the number and length of its path [21], [29].

Since the number of the rules generated in the previous step can be very high, the next MapReduce phase, *parallel rules selection*, selects only the top K non-redundant rules for each class label C_j. Each mapper is fed by an input key-value pair formatted as $\langle key = m, value = FCAR_m \rangle$ and outputs a pair $\langle key = C_{j_m}, value = FCAR_m \rangle$, where C_{j_m} is the class label associated with $FCAR_m$. Each reducer processes all rules with the same class label, and selects the best K significant non-redundant rules. As for the AC-FFP algorithm, a rule $FCAR_m$ is more significant than another $FCAR_l$ if and only if:

1) $conf(FCAR_m) > conf(FCAR_l)$
2) $conf(FCAR_m) = conf(FCAR_l)$ **AND** $supp(FCAR_m) > supp(FCAR_l)$;
3) $conf(FCAR_m) = conf(FCAR_l)$ **AND** $supp(FCAR_m) = supp(FCAR_l)$ **AND** $RL(FCAR_m) < RL(FCAR_l)$.

where $conf(.)$, $supp(.)$, and $RL(.)$ are the confidence, the support and the rule length respectively. A fuzzy rule $FCAR_l$ is pruned if and only if exists a rule $FCAR_m$ with higher rank and more general than $FCAR_l$. A rule $FCAR_m : FAnt_m \rightarrow C_{j_m}$ is more general than a rule $FCAR_l : FAnt_l \rightarrow C_{j_l}$, if and only if, $FAnt_m \subseteq FAnt_l$. For each of these K rules the reducer outputs a key-value pair $\langle key = null, value = FCAR_m \rangle$.

At the end of the fuzzy FCAR mining, the algorithm performs two additional scans of the overall dataset to prune noisy information. Figure 3 depicts the two MapReduce phases involved in the pruning.

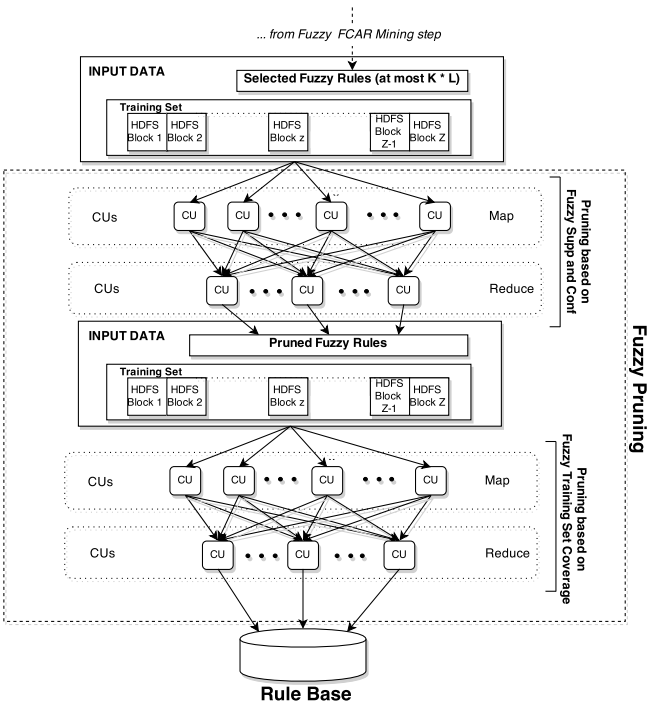

Fig. 3. The overall FCAR Pruning process of the DAC-FFP algorithm.

In the first one, a rule $FCAR_m$ is discarded if the fuzzy support and confidence are lower than a $minFuzzySupp$ and $minFuzzyConf$ thresholds calculated according to formulas 8 and 9, respectively. The mapper is fed by the training set block where each input takes the form $\langle key = r, value = \mathbf{o}_r \rangle$. Moreover, each mapper loads and ranks the rules mined in the previous steps into a ranked list and calculates the membership degree of each rule for $\mathbf{o}_r$ according to Eq. 2, by using the product as t-norm for implementing the conjunction operator. For each $FCAR_m$ with membership degree higher than zero, the mapper outputs the key-value pair

$\langle key = index_{FCAR_m}, value = w_m(\mathbf{x}_r)\rangle$, where the key is the index of rule $FCAR_m$ in the ranked list and $w_m(\mathbf{x}_r)$ is the membership degree of rule $FCAR_m$. Note that, when the class label C_{j_m} of rule $FCAR_m$ is different from class label y_r of object $\mathbf{o_r}$, the mapper outputs a negative value, $-w_m(\mathbf{x}_r)$, thus the reducer is able to properly compute the fuzzy support and the fuzzy confidence. Indeed, each reducer inputs a key-value pair as $\langle key = index_{FCAR_m}, value = list(w_m)\rangle$, where $list(w_m)$ is the list of all non-zero matching degrees for the rule $FCAR_m$. The fuzzy support and the fuzzy confidence are calculated according to Equations 5 and 6, where the numerators of both equations is computed by considering only the positive w_m, and the denominator of the second one is computed by considering the absolute value of each element in the $list(w_m)$. Finally, each reducer outputs only FCARs with confidence and support higher than $minFuzzySupp$ and $minFuzzyConf$, respectively.

The last type of pruning is carried out by performing the training set coverage phase. Since the mapper is fed with a training set block, the key-value input pair is $\langle key = r, value = \mathbf{o}_r\rangle$. Further, each mapper loads and ranks the filtered rule set extracted in the previous phase. As in the AC-FFP, a counter initialized to 0 is associated with each object $\mathbf{o_r}$. The mapper scans the rule list and, for each $FCAR_m$, if $\mathbf{x}_r$ matches the rule, then the counter is incremented by 1. Only those rules with matching degree higher than the *fuzzy matching degree threshold* $\overline{w}_m = 0.5^{g_m-1}$, where g_m is the rule length of $FCAR_m$, are considered. If the $FCAR_m$ correctly classifies $\mathbf{x}_r$, the mapper outputs a key-value pair as $\langle key = index_{FCAR_m}, value = null\rangle$, where the key is the index of the $FCAR_m$ in the ranked list. When the counter is higher than a *coverage threshold* δ, the corresponding object is not considered anymore. The reducer receives as input the key-value pair $\langle key = index_{FCAR_m}, value = null\rangle$, gets the rule from the ranked list and outputs $\langle key = null, value = FCAR_m\rangle$.

Note that the overall process is carried out by performing five MapReduce phases that involve four training set scans.

To classify unlabeled objects, we perform the same reasoning method described in Section III-A.

IV. Experimental Study

We tested our method on a real-word big dataset, namely Susy, extracted from the UCI repository[1]. Susy is characterized by 18 input features, 5 millions instances and 2 classes.

We implemented our algorithm by using Apache Hadoop 1.0.4 as the reference MapReduce implementation and performed all the experiments using a small computer cluster with one master and three slave nodes. All the nodes are connected by a Gigabit Ethernet (1 Gbps) and run Ubuntu 12.04. The master node has a 4-core CPU (Intel Core i5 CPU 750 x 2.67 GHz), 4 GB of RAM and a 500GB Hard Drive. Each slave node has a 4-core CPU with Hyperthreading (Intel Core i7-2600K CPU x 3.40 GHz), 16GB of RAM and 1 TB Hard Drive. The training sets stored in the HDFS are split into blocks with the default size (64MB).

[1]Available at https://archive.ics.uci.edu/ml/datasets.html

We adopted the following values of the parameters in the experiments: $MinSupp = 0.01$, $MinConf = 0.5$, $\delta = 4$, $min\chi^2 = 20\%$. These parameters are equal to the ones used in [9] for AC-FFP. Further, the number K of FCARs per class label mined by the FCAR mining process is set to 10,000.

We recall that the aim of this paper is not to highlight the effectiveness of DAC-FFP as classifier. On the other hand, DAC-FFP is the distributed version of AC-FFP and we have already analyzed the effectiveness of AC-FFP in [9] . By using non-parametric statistical tests, we have shown that AC-FFP outperforms in terms of accuracy other well-known associative classifiers, such as CMAR [1], and is statistically equivalent to two recent fuzzy associative classifiers, namely FARC-HD [7] and D-MOFARC [8]. On the other hand, to the best of our knowledge, there are no fuzzy associative classifier implementations that can handle big datasets. However, for the sake of completeness, we performed a five-fold cross-validation on the Susy dataset: the average values of classification rate achieved by DAC-FFP on the training and test sets were 74.627% and 74.619%, respectively. Just to give a glimpse of the goodness of these results, in another paper, where the Susy dataset was used with the K-NN classifier in the framework of prototype reduction for big data classification [30], the best average values of classification rates using a five-fold cross-validation were 69.41% and 72.82% on training and test sets, respectively.

In the following, we investigate the performance of the DAC-FFP in term of scalability.

A. Scalabilty

In order to evaluate the scalability of the proposed approach, we use as metric the *speedup* σ, which is commonly used in parallel computing. As stated by the speedup definition, the efficiency of a program using multiple computing units can be calculated comparing the execution time of the parallel implementation against the corresponding sequential version. Unfortunately, due to the large size of the dataset, the sequential implementation of the algorithm is impracticable because it would take an unreasonable amount of time. To overcome this drawback, we take as reference execution for our experiments a run over Q^* identical cores with $Q^* > 1$. We redefine the speedup formula on n computing units as follow:

$$\sigma_{Q^*}(n) = \frac{Q^* \cdot \tau(Q^*)}{\tau(n)} \tag{11}$$

where $\tau(n)$ is the program runtime using n computing units and Q^* is the number of computing units used to run the reference execution. In our tests, we have assumed $Q^* = 6$. Further, the computing units are uniformly distributed across the cluster so that each slave exploits 2 cores. Note that, in our case, σ_6 takes care of the basic overhead due to both the Hadoop platform and the thread interference.

According to the scalability experiments, we perform several executions by varying the number of switched-on cores per node, keeping the same number of running cores per node for avoiding unbalanced loads. Obviously, $\sigma_{Q^*}(n)$ makes sense only for $n \geq Q^*$, where the speedup is expected to be sub-linear due to the increasing overhead from the Hadoop procedures, the contention of shared resources between cores, and the necessary sequential parts of our algorithm. In practice,

we considered 6, 9, and 12 cores distributed on the three slave nodes. Note that our CPUs are equipped with the HyperThreading technology, which allows running two distinct processes per core. However, the performance gain due to HyperThreading highly depends on the target application, and in our case, specific tests showed that HyperThreading yields really limited performance improvements. For this reason we disabled the HyperThreading Technology and used only the available physical CPUs in all our experiments.

Considering the structure of our algorithm, the number of reducers is set equal to the number of cores available on the slaves.

Table I summarizes the results of the experiments and Figures 4 and 5 show, respectively, the speedup and the execution time according to the whole dataset.

TABLE I. SPEEDUP OF DAC-FFP ALGORITHM.

# Cores	Time (s)	Speedup	$\sigma_6(Q)/Q$ (Utilization)
6	10645	6	1.00
9	7847	8.14	0.90
12	6457	9.89	0.82

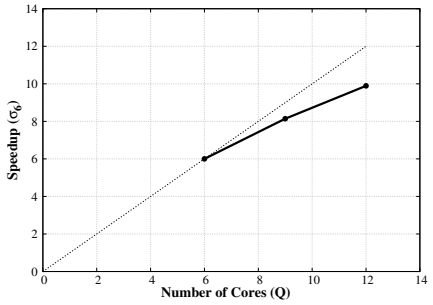

Fig. 4. The speedup of DAC-FFP.

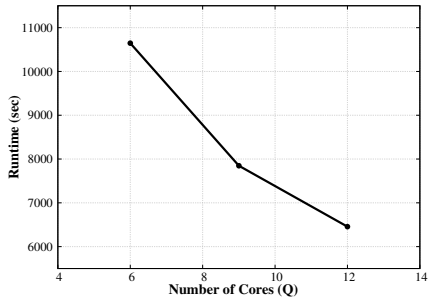

Fig. 5. The execution time of DAC-FFP varying the number of available cores.

Note that Hadoop automatically determines the number of Z mappers that, with the default settings, is driven by the HDFS block size. For the Susy dataset, Hadoop instantiates 36 mappers. Furthermore, if $Z \leq Q$, where Q is the number of available cores, then all the mappers are executed simultaneously on the cluster and the global runtime practically corresponds to the longest of the mappers' runtimes. On the other hand, if $Z > Q$, Hadoop runs Q mappers in parallel, and then queues the remaining $(Z - Q)$ mappers. As soon as one of the running mappers completes, Hadoop schedules a new mapper from the queue.

In the ideal case of the same execution time for all the mappers, the map phase for each MapReduce stage would require $\lceil \frac{Z}{Q} \rceil$ iterations. With the Susy dataset, this corresponds to 6, 4 and 3 iterations on 6, 9 and 12 cores, respectively. This observation can be used to get a very rough estimation of the runtime expected with a certain number of cores, once the runtime with another given number of cores has been recorded. For instance (see Table I), we expect that runtime decreases from 10645 seconds with 6 cores to about $10645 \times 4 \div 6 = 7097$ and $10645 \times 3 \div 6 = 5323$ seconds with 9 and 12 cores, respectively. As it can be noticed, such values do not excessively differ from the recorded ones. Obviously, such an estimation cannot be accurate and the actual runtimes are necessarily higher due to: (i) the incurred overheads, (ii) the different execution times of the mappers (they may even be assigned different input sizes, as for the case of the mapper assigned to the last HDFS block), and (iii) the influence of the different reducing phases.

As regards the third item, we recall that the number of reducers, denoted as R, can be set by the user and, in our tests, is equal to the number of available cores Q. Indeed, each reducer processes the values associated with the intermediate key generated by the Z mappers. With the default settings, Hadoop determines the queue of intermediate keys assigned to each reducer through the default partition function $hash(key) \, mod \, R$. If the partition function distributes uniformly all the intermediate keys, then the queues processed by each reducer have more or less the same size.

In our algorithm, the time spent by each MapReduce phase is driven essentially by the map phases, except for the parallel fuzzy FP-Growth. In this case, the most of the execution time is dominated by the reducer activites for mining fuzzy CARs from the local conditional FP-Trees. Indeed, the mappers generate as many intermediate keys (149 in our tests) as the number of frequent fuzzy sets in the f_{list}, thus each reducer processes about 25, 17 and 13 conditional FP-Trees in the 6, 9 and 12 cores cases, respectively. As stated in Section III-B, even though the number of conditional FP-Trees processed by each reducer is the same, such distribution does not necessarily guarantee a perfect load balancing among all the nodes, because the time spent in processing each specific FP-Tree depends on the number and length of its paths [21], [29].

However, the actual speedup σ_6 in our experiments is quite close to the ideal value, i.e. the number of CUs[2]: $\sigma_6(9)/9 = 0.90$ and $\sigma_6(12)/12 = 0.82$. Within the limitations due to the different experimental settings, this result is in line with [20] where the utilization is 0.768.

[2]The value σ_1/Q is the standard *utilization* index; in our case, as $\sigma_i(n) \leq \sigma_j(n) \ \forall n \geq j$, the utilization index σ_6/Q may be slightly greater than standard utilization.

V. Conclusion

We have recently proposed AC-FFP, a fuzzy associative classifier based on a fuzzy version of the well-known FP-Growth. AC-FFP has proved to be very effective in terms of accuracy, but results to be quite heavy both in terms of computational complexity and memory occupation. For this reason, in this paper, we have shown a MapReduce distributed version of the AC-FFP learning algorithm, based on a distributed implementation of the fuzzy FP-Growth. This version is able to process millions of objects. Further, the MapReduce paradigm and the distributed file system integrated in the Hadoop framework allow us to efficiently parallelize the computation flow across computer clusters by providing a robust and transparent environment that takes care of communications between them and possible failures. We tested the speedup of the algorithm on a real-world big dataset with 5 millions instances by using personal computers connected by a Gigabit Ethernet. The experimental results show that the algorithm achieves speedup close to the ideal achievable targets. We would like to point out that these results are obtained by using commodity cluster computing and not specific dedicated hardware.

References

[1] W. Li, J. Han, and J. Pei, "CMAR: accurate and efficient classification based on multiple class-association rules," in *Proceedings of IEEE International Conference on Data Mining 2001*, 2001, pp. 369–376.

[2] F. Thabtah, "A review of associative classification mining," *The Knowledge Engineering Review*, vol. 22, pp. 37–65, 3 2007.

[3] E. Baralis and P. Garza, "I-prune: Item selection for associative classification," *International Journal of Intelligent Systems*, vol. 27, no. 3, pp. 279–299, 2012.

[4] M. I. A. Ajlouni, W. Hadi, and J. Alwedyan, "Detecting phishing websites using associative classification," *European Journal of Business and Management*, vol. 5, no. 15, pp. 36–40, 2013.

[5] Y. Yoon and G. G. Lee, "Two scalable algorithms for associative text classification," *Information Processing and Management*, vol. 49, no. 2, pp. 484–496, 2013.

[6] F. P. Pach, A. Gyenesei, and J. Abonyi, "Compact fuzzy association rule-based classifier," *Expert Systems with Applications*, vol. 34, no. 4, pp. 2406 – 2416, 2008.

[7] J. Alcala-Fdez, R. Alcala, and F. Herrera, "A fuzzy association rule-based classification model for high-dimensional problems with genetic rule selection and lateral tuning," *IEEE Transactions on Fuzzy Systems*, vol. 19, no. 5, pp. 857–872, 2011.

[8] M. Fazzolari, R. Alcalá, and F. Herrera, "A multi-objective evolutionary method for learning granularities based on fuzzy discretization to improve the accuracy-complexity trade-off of fuzzy rule-based classification systems: D-MOFARC algorithm," *Applied Soft Computing*, vol. 24, no. 0, pp. 470 – 481, 2014.

[9] M. Antonelli, P. Ducange, F. Marcelloni, and A. Segatori, "A novel associative classification model based on a fuzzy frequent pattern mining algorithm," *Expert Systems with Applications*, vol. 42, no. 4, pp. 2086 – 2097, 2015.

[10] P. Zikopoulos, C. Eaton *et al.*, *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.

[11] X. Wu, X. Zhu, G.-Q. Wu, and W. Ding, "Data mining with big data," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 26, no. 1, pp. 97–107, 2014.

[12] C.-W. Lin, T.-P. Hong, and W.-H. Lu, "Linguistic data mining with fuzzy FP-trees," *Expert Systems with Applications*, vol. 37, no. 6, pp. 4560–4567, 2010.

[13] C.-T. P. Chien-Hua Wang, Wei-Hsuan Lee, "Applying fuzzy FP-Growth to mine fuzzy association rules," in *Engineering and Technology*. World Academy of Science, 2010, pp. 788–794.

[14] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[15] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining frequent patterns without candidate generation: A frequent-pattern tree approach," *Data Min. Knowl. Discov.*, vol. 8, no. 1, pp. 53–87, 2004.

[16] S.-Q. Wang, Y.-B. Yang, G.-P. Chen, Y. Gao, and Y. Zhang, "Mapreduce-based closed frequent itemset mining with efficient redundancy filtering," in *Proceedings of IEEE 12th International Conference onData Mining Workshops (ICDMW) 2012*, Dec 2012, pp. 449–453.

[17] M. Riondato, J. A. De Brabant, R. Fonseca, and E. Upfal, "PARMA: A parallel randomized algorithm for approximate association rules mining in mapreduce," in *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*. New York, NY, USA: ACM, 2012, pp. 85–94.

[18] M.-Y. Lin, P.-Y. Lee, and S.-C. Hsueh, in *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*. New York, NY, USA: ACM, 2012, pp. 76:1–76:8. [Online]. Available: http://doi.acm.org/10.1145/2184751.2184842

[19] X. Y. Yang, Z. Liu, and Y. Fu, "MapReduce as a programming model for association rules algorithm on Hadoop," in *Proceedings of 3rd International Conference on Information Sciences and Interaction Sciences (ICIS) 2010*, June 2010, pp. 99–102.

[20] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang, "PFP: Parallel FP-Growth for query recommendation," in *Proceedings of the 2008 ACM Conference on Recommender Systems*, ser. RecSys '08. New York, NY, USA: ACM, 2008, pp. 107–114.

[21] L. Zhou, Z. Zhong, J. Chang, J. Li, J. Huang, and S. Feng, "Balanced parallel FP-Growth with MapReduce," in *Proceedings of IEEE Youth Conference on Information Computing and Telecommunications (YC-ICT) 2010*, Nov 2010, pp. 243–246.

[22] T. White, *Hadoop: The Definitive Guide*, 3rd ed. O'Reilly Media, Inc, 2012.

[23] J. P. J. Han, M. Kamber, *Data Mining: Concepts and Techniques*, 3rd ed., ser. Data Management Systems. Morgan Kaufmann, 2012.

[24] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," *SIGMOD Record*, vol. 22, no. 2, pp. 207–216, Jun. 1993.

[25] H. Ishibuchi and T. Yamamoto, "Rule weight specification in fuzzy rule-based classification systems," *IEEE Transactions on Fuzzy Systems*, vol. 13, no. 4, pp. 428–435, 2005.

[26] H. Ishibuchi and T. Nakashima, "Effect of rule weights in fuzzy rule-based classification systems," *IEEE Transactions on Fuzzy Systems*, vol. 9, no. 4, pp. 506–515, 2001.

[27] U. M. Fayyad and K. B. Irani, "Multi-interval discretization of continuous-valued attributes for classification learning," in *Proceedings of IJCAI*, 1993, pp. 1022–1029.

[28] O. Cordón, M. J. del Jesus, and F. Herrera, "A proposal on reasoning methods in fuzzy rule-based classification systems," *International Journal of Approximate Reasoning*, vol. 20, no. 1, pp. 21–45, 1999.

[29] I. Pramudiono and M. Kitsuregawa, "Parallel FP-Growth on PC cluster," in *Advances in Knowledge Discovery and Data Mining*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, vol. 2637, pp. 467–473.

[30] I. Triguero, D. Peralta, J. Bacardit, S. Garca, and F. Herrera, "MRPR: A MapReduce solution for prototype reduction in big data classification," *Neurocomputing*, vol. 150, Part A, pp. 331 – 345, 2015.