Knowl Inf Syst manuscript No.
(will be inserted by the editor)

Distributed ReliefF based Feature Selection in Spark

Raul-Jose Palma-Mendoza · Daniel Rodriguez · Luis de-Marcos

Received: date / Accepted: date

Abstract Feature selection has become a key research area in the machine learning and data mining fields, by eliminating irrelevant and redundant features it has shown to be useful in reducing the effort required to process a dataset while maintaining or even improving the processing algorithm's accuracy. However, with the advent of big data, traditional algorithms designed for executing on a single machine lack scalability to deal with the increasing amount of data that is becoming available. ReliefF is one these traditional algorithms that has been successfully implemented in many feature selection applications. In this paper, we present a completely redesigned distributed version of the popular ReliefF algorithm. Our design is based in the novel Spark cluster computing model that has recently gained much attention due to its much faster processing times compared with Hadoop MapReduce model implementation. The effectiveness or our proposal is tested on four open datasets all of them with high number of instances, and two of them with also a high number of features. A traditional implementation is also tested on this datasets or portions of them. Results show that a traditional implementation is practically unable to handle such high amounts of data, and that our design can process them in a scalable way with much better time and memory consumption.

Keywords First keyword · Second keyword · More

1 Introduction

In the last years we have witnessed a vast increase in the amount of data that is being stored and processed by organizations of all types. As stated in [27], the so-called big data revolution has come to us with many challenges but also with

R. Palma
Information Systems Engineering Department
National Autonomous University of Honduras, Tegucigalpa, Honduras
E-mail: raul.palma@unah.edu.hn

D. Rodriguez · L. de-Marcos
Computer Science Department, University of Alcalá
Alcalá de Henares, 28871 Madrid

plenty of opportunities that these organizations are willing to take advantage of. According to [22], the main challenge is to extract useful information or knowledge from these huge volumes of data, this data would enable us to predict or to better understand the phenomena involved in their generation. This tasks area tackled by the widely known fields: data mining and machine learning.

In addition, feature selection has grown over the last decade to establish as a crucial step in the reduction of dimensionality of these amounts of data [4], by proving its usefulness to enhance the performance of many machine learning and data mining schemes. Feature selection algorithms are usually classified in any of three types: wrappers, filters and embedded methods. Wrappers refer to methods that require the learning of a classifier based on a single or on a subset of the original features, they are usually more computationally expensive. Filter methods rely only on the characteristics of data and are independent of any learning scheme, thereby requiring less computational effort. Finally, embedded methods refer to those techniques where the selection of features is done during the classification process.

By the way, as mentioned by García et al. [9], another important feature selection classification is the one that comes from viewing the feature selection problem as a search problem, and thereby, classifies methods according to the types of search that is performed: exhaustive, heuristic and stochastic. However, not all feature selection techniques can be viewed as search problems, so a fourth category can be included: feature weighting.

In this context, ReliefF [15] has become a widely applied feature weighting technique that estimates the quality of the features from a given dataset by assigning weights to each of them. It can be used as filter feature selection method by defining a significance threshold and selecting features with quality above it. Because of its advantages like: being able to work with nominal or continuous features, handling multi-class problems, detecting features interactions, handling missing data and noisy tolerance, it has been considered one of the most convenient filter-based feature selection methods available [2].

On the other hand, after its 2004 seminal paper Google's MapReduce [6,7] emerged as a programing model that simplified the development of scalable applications that process and generate large scale datasets. This applications are defined in terms of map an reduce tasks that are automatically parallelized by the framework and executed in clusters of thousands of machines, handling failures and scheduling resources. One important fact is that MapReduce was designed to run on commodity hardware, because it leads to lower costs per processor and per unit of memory of the cluster. The standard MapReduce implementation to date is Hadoop MapReduce [1], an open source implementation mainly developed at Yahoo Labs. However, the framework uses disk writing between every MapReduce job with the object of failure recovering, but this becomes a bottleneck for iterative nature algorithms like the ones used in machine learning and data mining, including the original ReliefF algorithm. For that reason, Spark [30] has gained much attention in the last couple of years, since it presents an improved model that is capable of handling most of its operations in-memory while maintaining the fault tolerance and scalability of MapReduce.

In this paper we present a distributed and scalable redesign of the original ReliefF algorithm to the Spark computing model, enabling it to deal with much more bigger datasets in terms of instances and features than the traditional version

would. In addition, after presenting and discussing the design of the algorithm we compare several runs of it with the traditional version implemented on the WEKA [11] platform, using 4 open datasets with numbers of instances in the order of 10^6 and a number of features that ranges in the order of 10^1 until 10^3. The main conclusion obtained is that is practically unfeasible to deal with such high amounts of data using the traditional version on standard hardware, while the distributed version is able to handle them smoothly.

2 Related Work

ReliefF was published by Kononenko in 1994 [15] and since then, many applications, extensions and improvements have been published. Moreover, ReliefF itself is an extension of the original Relief algorithm developed by [14], the latter initially was limited to binary class problems while the former can handle multi-class problems. As a recent examples of those extensions, Reyes et al. [23] present three of them for multi-label problems and compares them with previous extensions for the same purpose. Zafra et al. [28] extend ReliefF to the problem of multi-instance learning. Greene et al. [10] propose an adaptation to enhance ReliefF's ability to detect feature interactions called SURF (Spatially Uniform ReliefF) and use it in the genetic analysis of human diseases.

One of ReliefF's mayor flaws is its incapacity to detect redundant features, and hereby some attempts have been made to overcome this. For example, Li and He [17] use a forward selection algorithm to select non redundant critical quality characteristics of complex industrial products. Zhang et al. [31] combine ReliefF with the mRmR (minimal-redundancy-maximal-relevancy) algorithm [20] to select non redundant gene subsets that best discriminate biological samples of different types.

As it might be expected, most feature selection algorithms have asymptotic complexities that depend on the number of features and possibly in the number of instances, and in our case, ReliefF depends linearly on both [24]. Thereby, its performance gets compromised when faced with datasets with high dimensionality and/or high number of instances. For this reason, many attempts have been made to make feature selection, including ReliefF more scalable.

Recently, Canedo et al. in [3], propose a framework to deal with high dimensional data, distributing the dataset by features, processing in parallel the pieces, and then performing a merging procedure to obtain a single selection of features, however this method is oriented for high dimensional datasets and no tests were made with datasets with high amounts of instances. A some way similar approach is followed in Peralta et al. [21], where the MapReduce model is used to implement a wrapper-based evolutionary search feature selection method. In this case, the data is split by instances, and on each of this pieces a evolutionary feature selection is performed. Furthermore, the reduce step basically uses a simple majority voting of the selected features with a user-defined threshold to define the definitive subset of features. Finally, tests were made with the Epsilon dataset, that we also use here (see section 6).

Zhao et al. [32] presented a distributed parallel feature selection method based on variance preservation using the SAS High-Performance Analytics [1] proprietary software. Tests were made with both high dimensional and high number of instances datasets.

Kubica et al. [16], develop parallel versions of three forward search based feature selection algorithms, a wrapper with a logistic regression classifier is used to guide the search that is parallelized using the MapReduce model.

It is also worth to mention the recent work of Wang et al. [26], that uses the Spark computing model to implement feature selection strategy for classification of network traffic. Their approach basically consists in two phases, first it generates a feature ranking based on the Fisher score, and then a forward search is done using a wrapper approach. Nevertheless, this has the clear disadvantage that can only be applied to continuous data.

As it can be seen, none of the above contributions implement a purely filter approach, even when the framework proposed in [3] allows to apply filters on parts of the dataset, these results are then merged using a wrapper.

Finally, for the specific case of ReliefF, Huang et al. [13] propose an optimization in order to improve the computation efficiency of ReliefF on large datasets, but this improvement is only useful when no instance random sampling step is performed for the weights approximation. In other words, it only works when the m parameter (see section 3) is set equal to the number of instances. Furthermore, testing in their work is made with datasets with a number of instances in the order of 10^4, that is still manageable by a single machine.

Summing up and to the best knowledge of the authors, no published contribution has ever attempted to redesign the ReliefF filter method to a distributed environment as is proposed here.

3 ReliefF

In this section, we provide a short discussion about the ReliefF algorithm published by Kononenko [15] that will serve as a conceptual basis for the redesign presented in section 5. As indicated earlier, ReliefF is based in the original Relief algorithm proposed by Kira and Rendell [14]. They both share the same central idea, that consists in evaluating the quality of the features by their ability to distinguish the instances from one class to another in a local neighborhood, i.e. the best features are those contribute more to add distance between different class instances and contribute less to add distance between same class instances. ReliefF then, consists of an extension of the original Relief method thats is capable of working under multi-class, noisy and incomplete datasets.

Algorithm 1 displays ReliefF's pseudo-code. As we can observe, it consists of a main loop that iterates m times, where m corresponds to the number of samples from data to perform the quality estimation. Each selected sample R_i equally contributes to the a-size weights vector W, where a is the number of features in the dataset. The contribution for the A-esim feature is calculated by first finding k nearest neighbors of the actual instance for each class in the dataset. The k neighbors that belong to the same class as the actual instance are called hits(H), and

[1] `http://www.sas.com/en_us/software/high-performance-analytics.html`

Algorithm 1 ReliefF [24, 15]

1: calculate prior probabilities $P(C)$ for all classes
2: set all weights $W[A] := 0.0$
3: **for** $i = 1$ **to** m **do**
4: randomly select an instance R_i
5: find k nearest hits H_j
6: **for all** classes $C \neq cl(R_i)$ **do**
7: from class C find k nearest misses $M_j(C)$
8: **end for**
9: **for** $A := 1$ **to** a **do**
10: $SH := -\sum_{j=1}^{k} diff(A, R_i, H_j)$
11: $SM := \sum_{C \neq cl(R_i)} \left[\left(\frac{P(C)}{1-P(cl(R_i))} \right) \sum_{j=1}^{k} diff(A, R_i, M_j(C)) \right]$
12: $W[A] := W[A] + (SH + SM)/(m \cdot k)$
13: **end for**
14: **end for**

the other $k \cdot (c-1)$ neighbors are called misses (M), where c is the total number of classes, an $cl(R_i)$, represents the class of the $i-$esim sample. Once neighbors are found, their respective contributions to A-esim feature, is calculated. The contribution of the hits collection is equal to the negative of the sum of the differences between the actual instance and each hit, observe that this is a negative contribution because only non desirable features should contribute to create differences between neighbor instances of the same class. Analogously, the contribution of the misses collection is equal to the weighted sum of the differences between the actual instance and each miss, this is of course a positive contribution because good features should help differentiate between instances of a different class. The weights for this summation is defined according to the prior probability of each class, calculated from the dataset. Finally, the sum of the contributions is divided by k to calculate an average of them, and to maintain the definitive weights in the range $[-1, 1]$ a division by m must be done. ReliefF's weights are interpreted in the positive direction: the higher the weight, the higher the corresponding feature's relevance.

Originally, the $diff$ function used to calculate the difference between two instances I_1 and I_2 for an specific feature A was defined as shown on equation 1 for nominal features, and as shown on equation 2 for numeric features. However, this latter one has been proved to cause an underestimation of numeric features with respect to nominal ones in datasets with both types of features. Thereby, a so-called ramp function, depicted in equation 3, has been proposed in [12] to deal with these problem. The basic idea of it, is to relax to equality comparison on equation 2 by using two thresholds: t_{eq} is the maximum distance between two features to still consider them equal, and analogously, t_{diff} is the minimum distance between two features to still consider them different. Their default values are set to 5% and 10% of the feature's value interval respectively. In our implementation, the use of the original $diff$ function or the ramp one for numeric features is selected through a initialization parameter. In addition, there are also versions of the $diff$ function that deal with incomplete data, however, since the datasets chosen for the experiments don't have missing values, we will not consider these versions in this work. Finally, the $diff$ function is used in two cases in the ReliefF algorithm, one in lines 10 and 11 for the calculation of the weight but also in the finding

of the distances between the instances, defined as the sum of the differences over every feature (Manhattan distance).

$$diff(A, I_1, I_2) = \begin{cases} 0 & \text{if } value(A, I_1) = value(A, I_2), \\ 1 & \text{otherwise} \end{cases} \tag{1}$$

$$diff(A, I_1, I_2) = \frac{|value(A, I_1) - value(A, I_2)|}{max(A) - min(A)} \tag{2}$$

$$diff(A, I_1, I_2) = \begin{cases} 0 & \text{if } d \leq t_{eq}, \\ 1 & \text{if } d > t_{diff}, \\ \frac{d - t_{eq}}{t_{diff} - t_{eq}} & \text{if } t_eq < d \leq t_{diff} \end{cases} \tag{3}$$

4 Spark Cluster Computing Model

In this section, we discuss the main concepts behind the Spark computing model, focusing on those that will complete the conceptual basis for the description of our proposal in section 5. We end up with a brief discussion about other existent computing models such as MapReduce, with the aim of justifying our selection of Spark.

The main concept behind the Spark model is that of resilient distributed dataset or in short RDD. Zaharia et al. [30, 29] defined a RDD as a read-only collection of objects, that is partitioned and distributed across the nodes of a cluster, and has the ability to automatically recover lost partitions through the record of a lineage that knows where do the data came from and optionally, the calculations that went through it. What is more relevant, is that the operations run for an RDD are automatically parallelized by the Spark engine, this abstraction frees the programmer of having to deal with threads, locks and all the complexities involved in traditional parallel programming.

There are two types of operations that can be executed on a RDD: actions and transformations. Actions are the mechanism that permit to obtain results from a Spark cluster, five commonly used actions are: *reduce*, *sum*, *aggregate*, *sample* and *collect*. The action *reduce* is used to aggregate the elements of a RDD, by applying a commutative and associative function that receives as arguments two elements of the RDD an returns one element of the same type. Action *sum* is simply a shorthand for a reduce action that sums all the elements on the RDD. Next, action *aggregate* has a similar behavior to *reduce*, but the return type can be different from type of the elements of the RDD, it works in two steps: the first one aggregates the elements of each partition and returns an aggregated value for each of them, the second one, merges these values between all partitions to a single one, that becomes the definitive result of the action. Lastly, actions *sample* and *collect* are also similar, the former takes an amount of elements and returns a random sample of this size from the RDD, and the latter, simply returns an array with all the elements in the RDD, this of course has to be done with care to prevent exceeding the maximum memory available at the driver.

On other hand, transformations are the mechanism for creating RDD from another. Since RDDs are read-only a transformation does not affect the original RDD but creates a new one. Maybe the three most important transformations

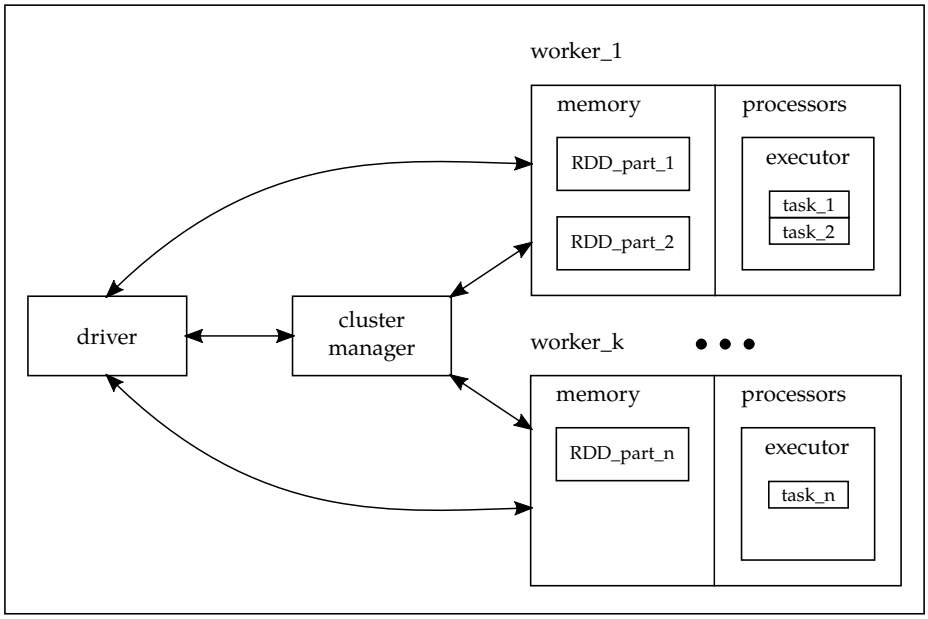

Fig. 1 Spark Cluster Architecture

are: *map*, $flatMap$ and $filter$. The first two: *map* and $flatMap$ are similar, they return a new RDD that is the result of applying a function to each element of the original one. In the case of *map*, the function applied takes a single argument and returns a single element, thus the new RDD has the same number of elements that the original one. And in the case of $flatMap$, the applied function takes a single element but it can return zero or more elements, so the resulting RDD is not required to have the same number of elements than the original one. Finally, $filter$ is straightforward, it receives a boolean function to discriminate the dataset elements possibly returning a subset of it.

In what respects to the cluster's architecture, Spark follows the master-slave model, there is a cluster manager (master) through which, the so-called driver program can access the cluster, the driver coordinates the execution of a user application by assigning tasks to the executors, which are programs that run in the worker nodes (slaves), by default only one executor is run per worker. With regard to the data, the RDD partitions are distributed across the worker nodes, and the number of tasks launched by the driver on for each executor will be according to the number of partitions of the RDD residing in the worker. A detailed view of the discussed architecture with respect to the physical nodes can be seen on Figure 1.

4.1 Other Cluster Computing Models

Spark has quickly emerged between other cluster computing models that have already took much attention, being MapReduce the most relevant of them. This has happened due to the many advantages Spark has over them. First of all, Spark

was designed from the beginning to efficiently handle iterative jobs in-memory, such as the ones used by many machine learning and data mining schemes. This has lead to the quick development of a machine learning library [19] that contains Spark model redesigned algorithms such as the one in this work. Moreover, besides the Spark author's own comparison [30,29], others [25] have shown that Spark is faster than MapReduce in most of the data analysis algorithms tested. Second, any MapReduce program can be directly translated to Spark, i.e. the MapReduce model can be completely expressed using the $flatMap$ and $groupByKey$ operations in Spark. And finally, from previous discussion is clear that Spark provides a wider range of operations other than $flatMap$ and $groupByKey$.

That said, its valuable to mention that other tools have already tried to fulfill the lack of efficient iterative job handling in MapReduce. Two of them are HaLoop [5] and Twister [8]. However, even when they have support for executing iterative MapReduce jobs, automatic data partitioning, and Twister has the ability to keep it in-memory, they lack facility of loading a dataset in memory and then deciding what to with it, preventing an interactive data mining session over the data. Moreover, a recently published work [18], that included comparisons of parallelized versions of a neural network algorithm over Hadoop, HaLoop and Spark, concludes that Spark is the most efficient in all cases.

5 Distributed ReliefF

In this section, we proceed to describe the proposed algorithm. The first design decision is where to concentrate the parallelization effort, ReliefF can be described as an embarrassingly parallel algorithm, since its outermost loop goes through completely independent iterations, each of these can be directly executed in different threads as stated by Robnik and Kononenko in [24]. However, parallelizing the algorithm in such way ties the parallelization to the number of samples m, and, as also stated by Robnik and Kononenko, this number of samples doesn't need to be very high when there is a high amount of data. In contrast, to achieve a good performance a high number of parallel tasks is needed. Thereby, a different approach must be followed. Continuing with Robnik and Kononenko discussion, ReliefF algorithm's complexity is $O(m \cdot n \cdot a)$, being n the number of instances in the dataset and m and a the two already defined variables. Moreover, the most complex operation is the selection of the k nearest neighbors, because first, the distance from the current sample to each of the instances must be calculated with $O(n \cdot a)$ steps, and then the selection must be done in $O(k \cdot log(n))$ steps. For this reasons, the parallelization is focused on these stages rather than on the independent iterations.

The first thing to note, is that ReliefF can be considered as a function applied to a dataset DS, to the number of samples s and to the number of neighbors k, that returns a $a-$size vector of weights W, as shown on equation 4. Thus, ReliefF's algorithm can be interpreted as the calculation of each individual weight W_A, using equation 5, where $sdiffs$ (equation 6) represents a function that returns the total sum of the differences in the $A-$esim feature between a given instance, and a set $NN_{C,i}$ of k neighbors of the $i-$esim sample where all belong to the same given class C.

$$relieff(DS, s, w) = W \tag{4}$$

$$W[A] = \sum_{i=1}^{m} \left[-sdiffs(A, R_i, cl(R_i)) + \sum_{C \neq cl(R_i)} \left[\left(\frac{P(C)}{1 - P(cl(R_i))} \right) sdiffs(A, R_i, C) \right] \right] \tag{5}$$

$$sdiffs(A, R_i, C) = \frac{1}{m \cdot k} \cdot \sum_{j=1}^{k} diff(A, R_i, NN_{C,i,j}) \tag{6}$$

The dataset DS can be defined (see equation 7) as a set of n instances each represented as a pair $I_i = (F_i, C_i)$ of a features vector F_i and a class C_i.

$$DS = \{(F_1, C_1), (F_2, C_2), \cdots, (F_n, C_n)\} \tag{7}$$

Given the initial definitions, and assuming that the features types are stored as metadata, we start by calculating the maximum and minimum values for all continuous features in the dataset, these values will be needed by the $diff$ function (see equations 3 and 2). This task, can be efficiently achieved by means of a *reduce* action with a function $fmax$ ($fmin$) that given two instances returns a third one that contains the maximum (minimum) values for each continuous feature of both instances or anything if the feature is not continuous. This is shown for maximum values on equations 8.

$$\begin{aligned} MAX &= DS.reduce(fmax) \\ MAX &= (max(F_1[1], \cdots, F_n[1]), \cdots, max(F_1[a], \cdots, F_n[a])) \end{aligned} \tag{8}$$

The next step consists in calculating the prior probabilities of all classes in DS, this is also a requirement of the $diff$ function. This values can be essentially obtained by means of a *map* and a $reduceByKey$ transformations, the former simply returns a dataset with all instance classes paired with a value of one, and the latter sums this ones using the class as a key, thereby obtaining a set of pairs $(C_i, count(C_i))$, containing the classes and the number of instances in DS belonging to that class, which can simply be divided by n to obtain the priors. Equations 9 depict the previous discussion, note that with the use of collect, P is turned into a local collection rather than a RDD.

$$\begin{aligned} f(I) &= (cl(I), 1) \\ g(a, b) &= a + b \\ h((a, b)) &= (a, b/n) \\ P &= DS.map(f).reduceByKey(g).map(h).collect() \\ P &= \{(C_1, prior(C_1)), \cdots, (C_c, prior(C_c))\} \end{aligned} \tag{9}$$

A rather short step, is the selection of the m samples, this is accomplished with the use of the *takeSample* action, as shown on equation 10.

$$R = \{R_1, \cdots, R_m\} = DS.takeSample(m) \tag{10}$$

Now we can proceed to the computationally intensive step of finding the k nearest neighbors of each sample for each class. First, we find the distances from every sample m to each of the n instances. This can be directly accomplished by means of a *map* transformation applied to DS where for every instance I, a vector of distances from it to the m samples is returned. Group of equations 11, depicts the previous discussion.

$$\begin{aligned} distances(I) &= (distance(I, R_1), \cdots, distance(I, R_m)) \\ f(I) &= (I, distance(I)) \\ DD &= DS.map(f) \\ DD &= \{(I_1, distances(I_1)), \cdots (I_n, distances(I_n)))\} \end{aligned} \tag{11}$$

Second and last, by using an *aggregate* action we are able to obtain the nearest neighbors NN matrix, where each element NN_C, i is a vector with the k nearest neighbors of a sample R_i belonging to a class C. As said before, *aggregate* action has two steps, the first step is defined in the function $localNN$, as its name suggests, this function returns a local neighbors matrix LNN for each partition of the RDD, it has a similar structure as the NN matrix but each vector element is treated as binary heap, where the max value is the neighbor with the highest distance found, so while traversing the whole partition, whenever an instance with shorter distance than the max is found, it is added to the heap, and if the heap is full the max element is removed to make space. Thus, at the end the heap will contain the nearest neighbors in the local partition. One important thing to note here, is that each sample is effectively the nearest neighbor of the same class to itself, so a condition must be set to prevent including it in the LNN. The next step, is the merging of the local matrices, the defined function $mergeNN$ combines two local matrices by merging its individual binary heaps, keeping only the elements with shorter distances. Once both functions have been defined, a call to the *aggregate* action can be performed, providing also an empty matrix LNN structure (with empty heaps) so the $localNN$ can start aggregating neighbors to it. Lastly, it is worth to note that since the NN is obtained via an action, it is a local object and not a RDD anymore and obtaining it concludes the complex step in the algorithm, step that has been fully parallelized. See equations 12 for the above discussion.

$$\begin{aligned} NN &= \begin{bmatrix} NN_{1,1} & \cdots & NN_{1,m} \\ \vdots & \ddots & \vdots \\ NN_{c,1} & \cdots & NN_{c,m} \end{bmatrix} \\ NN_{C,i} &= (N_1, \cdots, N_k \mid \forall N \ cl(N) = C) \\ localNN(I, LNN_{in}) &= LNN_{out} \\ mergeNN(LNN_a, LNN_b) &= LNN_{merged} \\ NN &= DS.aggregate(emptyNN, localNN, mergeNN) \end{aligned} \tag{12}$$

Once the NN matrix is stored on the driver program, operations are not distributed in the cluster anymore, however, this is not a problem but a requirement, because NN matrix is small: $c \times m$. The last remarkable step, consists in obtaining the matrix $SDIF$. Each element $SDIF_{C,i}$ represents an a-size vector, and each element of this vector stores the sum of the differences for the A-esim feature between the $NN_{c,i}$ group of neighbors and the R_i sample. Moreover, each element the vector $SDIF_{C,i}$ can be calculated by mapping the $diff$ function over the A-esim feature of all the instances in the $NN_{C,i}$ vector and then summing this differences. Observe that this map and sum functions are not RDD-related anymore, but local equivalents that can be parallelized only on the driver's local threads. Finally, note that each element of each vector of the matrix $SDIF$ effectively represents the value shown on equation 6, thus, the final weights vector W, can be easily figured by applying the formula given on equation 5 using the already obtained P set with the prior probabilities. Equations 13 depict the above discussion and a resume of the algorithm's main pipeline discussed in this section is shown on Figure 2.

$$\begin{aligned} SDIF_{C,i} &= (sdiffs(1, R_i, C), \cdots, sdiffs(a, R_i, C)) \\ f(N) &= diff(A, N, R_i) \\ SDIF_{C,i,A} &= sdiffs(A, R_i, C) = NN_{C,i}.map(f).sum/(m \cdot k) \\ SDIF &= \begin{bmatrix} SDIF_{1,1} & \cdots & SDIF_{1,m} \\ \vdots & \ddots & \vdots \\ SDIF_{c,1} & \cdots & SDIF_{c,m} \end{bmatrix} \end{aligned} \tag{13}$$

Before ending this section, there is one implementation issue that is worth to be mentioned. As previously said, a RDD is designed to be stored in memory, but this won't happen automatically, so if the RDD is going to be used in future operations it must be explicitly cached. In our case, after DD is obtained, it should be cached, because it will be again be used to calculate the NN matrix. Don't doing this will cause a serious performance downgrade because all the distances will need to be recalculated.

6 Experimental Evaluation

In this section, empirical results obtained from different executions of the proposed algorithm are presented. The experiments where performed with the aim of testing the algorithm scalability, and its time and memory consumption with respect with the traditional version. For the realization of the tests, an 8-node cluster of virtual machines was used, one node is used as a master and the rest are slaves, the cluster runs over the following hardware-software configuration:

- Host Processors: 16 X Dual-Core AMD Opteron 8216.
- Host Memory: 128 GB DDR2.
- Host Storage: 7 TB SATA 2 Hard Disk.
- Host Operating System: CentoOS 7.

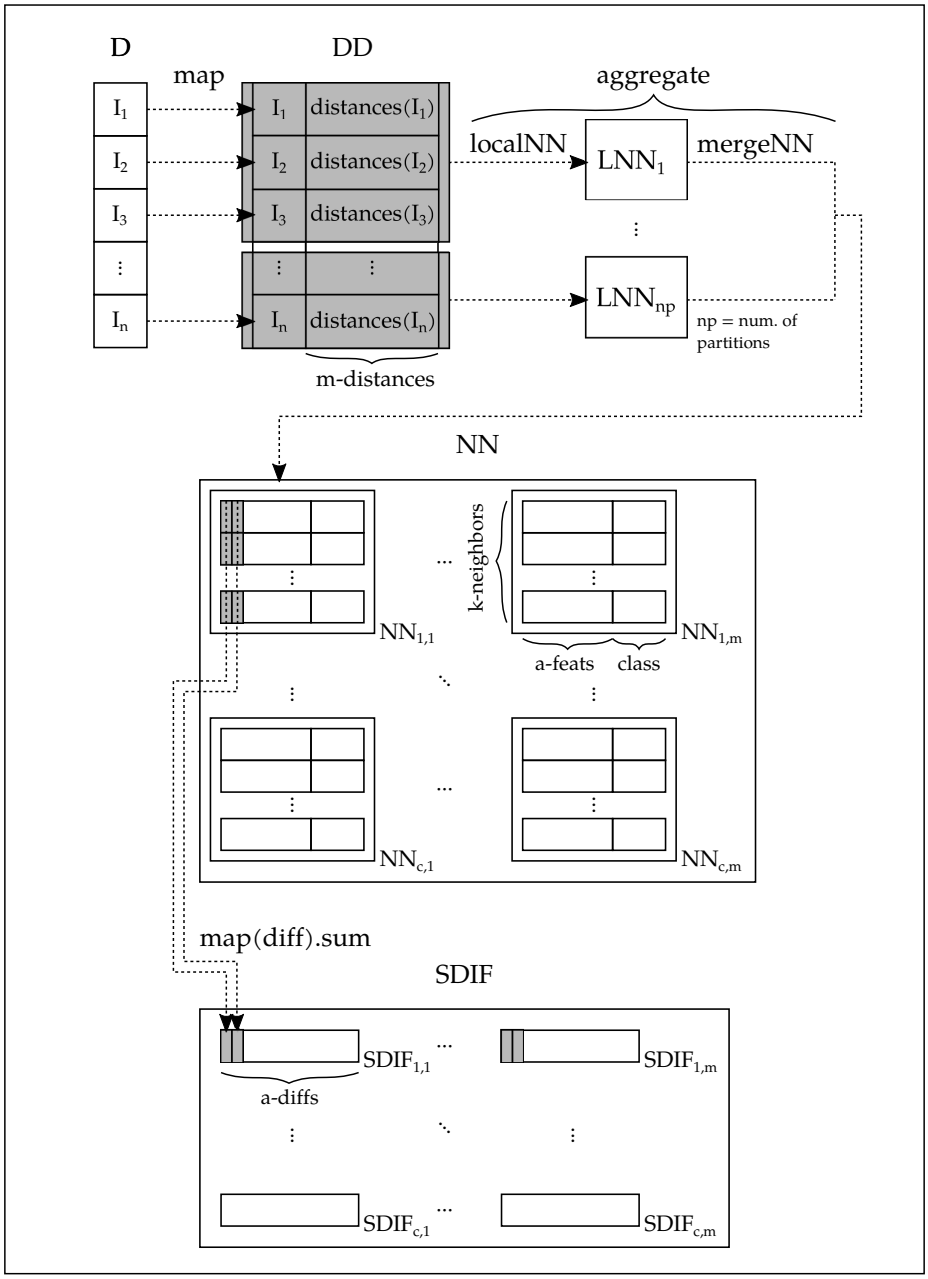

Fig. 2 Distributed ReliefF Main Pipeline

- Hypervisor: KVM (qemu-kvm 1.5.3).
- Guests Processors: 2.
- Guests Memory: 16 GB.
- Guests Storage: 500 GB.
- Java version: OpenJDK 1.8.
- Spark version: 1.6.1.
- HDFS version: 2.6.4.
- WEKA version: 2.8.

For the first part of the tests, the ECBDL14 dataset was used, this dataset comes from the Protein Structure Prediction field, and it was recently used during the ECBLD14 Big Data Competition of the GECCO2014 international conference. The dataset has more approximately 33.6 million instances, 631 attributes, 2 classes, 98% of negative examples and occupies about 56GB of disk space. All tests are run with a number of neighbors $k = 10$, which is a typical choice [15] and a relatively low number of samples $m = 10$ because as shown by Robnik and Kononenko [24], when a large amount of instances is available, lower values of m are needed. In addition, HDFS is used to store all the datasets or samples of datasets in the experiments related to the distributed version. Conversely, the local file system is used for tests of the traditional version of ReliefF.

An important configuration issue refers to driver memory consumption. In Spark computation model, there is no communication between tasks, so all the tasks results will be sent to the driver, this is especially important for the $aggregate$ action, because every task performing the $localNN$ operation will return a LNN matrix to the driver that then is going to be merged. The Spark configuration parameter $spark.driver.maxResultSize$ has a default value of 1 gigabyte but it was set to 4 for all the experiments performed. This is specifically important for tests involving higher matrix sizes: those with higher values of m or c.

Figures TODO, show respectively the time and memory consumption behavior of the distributed version of ReliefF version versus the traditional one implemented in the WEKA[11] platform for incrementally sized samples of the ECBDL14 dataset. For the comparison to be possible, the WEKA version was executed under the host environment, with no virtualization involved. The first thing to notice is that for the WEKA version a 30% sample was the largest that could be tested, this is because using a 40% sample would need more memory than the available in the system. This shows the lack of scalability of the traditional version. With respect to the distributed version, we observe that besides being able to handle the whole dataset, it preserves the linear behavior to number of instances the traditional version has, it is also able to process data in much less time, by leveraging the cluster nodes.

Analogous results are obtained by varying the number of features and the number of samples as can be seen in figures TODO., thereby confirming an empirical complexity equivalent to the original one: $O(m \cdot n \cdot a)$.

Another previously mentioned advantage of Spark that is designed to run in commodity hardware, by a simple look to the memory consumption of the traditional version, we infer that the required amounts of memory quickly grow beyond the limits of an ordinary computer. While in the distributed version, we observe that using nodes with 16GB of memory is enough to handle the task. Moreover, figures TODO show that even when the whole dataset does not fit in memory

(after a 50% sample and beyond), Spark is able to use a mixture of memory and disk storage to handle the task efficiently.

Now, we delve into the topic of scalability. To keep times reasonable, the test was performed with 5%, 10% and 20% samples of ECBDL14 dataset. Figures TODO, show the behavior of the distributed algorithm with respect to the number of cores used. As it can be seen, the first added cores greatly contribute the reduce the amount of time, but after a quantity nc of introduced cores, the contribution is more subtle even reaching to be completely null. By observing the three figures, it is clear that the nc quantity depends on the size of the dataset, so, larger datasets can take more benefit of larger amounts of cores. On the other hand, smaller datasets will face the case were they do not have enough partitions to be distributed over all the cluster nodes (figure 5%), in this case, adding more nodes, will not cause any performance improvements. Figures 5,10,y 20 also display with a dotted line the WEKA version behavior, since it can only take advantage of one core, the execution time is constant. However, the time is better than the distributed version in the case where one core is involved (TODO check this), clearly because, it doesn't have to deal with the driver scheduling, the selection of an executor and communication between both of them over the network, as the distributed version does.

We should note that this is quite typical behavior and usually we get stable estimates after 20-50 iterations, -¿ page 28 Theoretic

Check this:

7 Conclusions

References

1. Apache Software Foundation: Hadoop. URL `https://hadoop.apache.org`
2. Bolón-Canedo, V., Sánchez-Maroño, N., Alonso-Betanzos, A.: A review of feature selection methods on synthetic data. Knowledge and Information Systems **34**(3), 483–519 (2012). DOI 10.1007/s10115-012-0487-8. URL `http://dx.doi.org/10.1007/s10115-012-0487-8`
3. Bolón-Canedo, V., Sánchez-Maroño, N., Alonso-Betanzos, A.: Distributed feature selection: An application to microarray data classification. Applied Soft Computing **30**, 136–150 (2015). DOI 10.1016/j.asoc.2015.01.035. URL `http://www.sciencedirect.com/science/article/pii/S156849461500054X`
4. Bolón-Canedo, V., Sánchez-Maroño, N., Alonso-Betanzos, A.: Recent advances and emerging challenges of feature selection in the context of big data. Knowledge-Based Systems **86**, 33–45 (2015). DOI 10.1016/j.knosys.2015.05.014
5. Bu, Y., Howe, B., Ernst, M.D.: HaLoop: Efficient Iterative Data Processing on Large Clusters. Proceedings of the VLDB Endowment **3**(1-2), 285–296 (2010). DOI 10.14778/1920841.1920881
6. Dean, J., Ghemawat, S.: MapReduce: Simplied Data Processing on Large Clusters. Proceedings of 6th Symposium on Operating Systems Design and Implementation pp. 137–149 (2004). DOI 10.1145/1327452.1327492
7. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM **51**(1), 107 (2008). URL `http://dl.acm.org/citation.cfm?id=1327452.1327492`
8. Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.H., Qiu, J., Fox, G.: Twister: A Runtime for Iterative MapReduce. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC '10, pp. 810–818. ACM, New York, NY, USA (2010). DOI 10.1145/1851476.1851593. URL `http://doi.acm.org/10.1145/1851476.1851593`

9. García, S., Luengo, J., Herrera, F.: Feature Selection. In: Data Preprocessing in Data Mining, pp. 163–193. Springer International Publishing, Cham (2015). DOI 10.1007/978-3-319-10247-4_7. URL http://dx.doi.org/10.1007/978-3-319-10247-4{_}7
10. Greene, C.S., Penrod, N.M., Kiralis, J., Moore, J.H.: Spatially Uniform ReliefF (SURF) for computationally-efficient filtering of gene-gene interactions. BioData Mining **2**(1), 5 (2009). DOI 10.1186/1756-0381-2-5. URL http://biodatamining.biomedcentral.com/articles/10.1186/1756-0381-2-5
11. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA data mining software. ACM SIGKDD Explorations Newsletter **11**(1), 10 (2009). DOI 10.1145/1656274.1656278. URL http://portal.acm.org/citation.cfm?doid=1656274.1656278
12. Hong, S.J.: Use of Contextual Information for Feature Ranking and Discretization. IEEE Trans. on Knowl. and Data Eng. **9**(5), 718–730 (1997). DOI 10.1109/69.634751. URL http://dx.doi.org/10.1109/69.634751
13. Huang, Y., McCullagh, P.J., Black, N.D.: An optimization of ReliefF for classification in large datasets. Data & Knowledge Engineering **68**(11), 1348–1356 (2009). DOI 10.1016/j.datak.2009.07.011
14. Kira, K., Rendell, L.A.: A practical approach to feature selection. Proceedings of the ninth international workshop on Machine learning pp. 249–256 (1992)
15. Kononenko, I.: Estimating attributes: Analysis and extensions of RELIEF. Machine Learning: ECML-94 **784**, 171–182 (1994). DOI 10.1007/3-540-57868-4. URL http://www.springerlink.com/index/10.1007/3-540-57868-4
16. Kubica, J., Singh, S., Sorokina, D.: Parallel Large-Scale Feature Selection. In: Scaling Up Machine Learning, February, pp. 352–370 (2011). DOI 10.1017/CBO9781139042918.018. URL http://ebooks.cambridge.org/ref/id/CBO9781139042918A143
17. Li, J., Cheng, K., Wang, S., Morstatter, F., Trevino, R.P., Tang, J., Liu, H.: Feature Selection: A Data Perspective (2016). URL http://arxiv.org/abs/1601.07996
18. Liu, Y., Xu, L., Li, M.: The Parallelization of Back Propagation Neural Network in MapReduce and Spark. International Journal of Parallel Programming pp. 1–20 (2016). DOI 10.1007/s10766-016-0401-1. URL http://link.springer.com/10.1007/s10766-016-0401-1
19. Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., Xin, D., Xin, R., Franklin, M.J., Zadeh, R., Zaharia, M., Talwalkar, A.: MLlib: Machine Learning in Apache Spark. Journal Of Machine Learning **17**, 1–7 (2015). URL http://www.jmlr.org/papers/volume17/15-237/15-237.pdf
20. Peng, H., Long, F., Ding, C.: Feature selection based on mutual information: criteria of max-dependency, max-relevance, and min-redundancy. IEEE transactions on pattern analysis and machine intelligence **27**(8), 1226–38 (2005). DOI 10.1109/TPAMI.2005.159. URL http://www.ncbi.nlm.nih.gov/pubmed/16119262
21. Peralta, D., del Río, S., Ramírez-Gallego, S., Riguero, I., Benitez, J.M., Herrera, F.: Evolutionary Feature Selection for Big Data Classification : A MapReduce Approach Evolutionary Feature Selection for Big Data Classification : A MapReduce Approach. Mathematical Problems in Engineering **2015**(JANUARY) (2015). DOI 10.1155/2015/246139. URL http://sci2s.ugr.es/sites/default/files/2015-hindawi-peralta.pdf
22. Rajaraman, A., Ullman, J.D.: Mining of Massive Datasets. Lecture Notes for Stanford CS345A Web Mining **67**, 328 (2011). DOI 10.1017/CBO9781139058452. URL http://ebooks.cambridge.org/ref/id/CBO9781139058452
23. Reyes, O., Morell, C., Ventura, S.: Scalable extensions of the ReliefF algorithm for weighting and selecting features on the multi-label learning context. Neurocomputing **161**, 168–182 (2015). DOI 10.1016/j.neucom.2015.02.045. URL http://www.sciencedirect.com/science/article/pii/S0925231215001940
24. Robnik-Šikonja, M., Kononenko, I.: Theoretical and empirical analysis of ReliefF and RReliefF. Machine learning **53**(1-2), 23–69 (2003)
25. Shi, J., Qiu, Y., Minhas, U.F., Jiao, L., Wang, C., Reinwald, B., Özcan, F.: Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics. Proc. VLDB Endow. **8**(13), 2110–2121 (2015). DOI 10.14778/2831360.2831365. URL http://dx.doi.org/10.14778/2831360.2831365
26. Wang, Y., Ke, W., Tao, X.: A Feature Selection Method for Large-Scale Network Traffic Classification Based on Spark. Information **7**(1), 6 (2016). DOI 10.3390/info7010006. URL http://www.mdpi.com/2078-2489/7/1/6

27. Xindong Wu, X., Xingquan Zhu, X., Gong-Qing Wu, G.Q., Wei Ding, W.: Data mining with big data. IEEE Transactions on Knowledge and Data Engineering **26**(1), 97–107 (2014). DOI 10.1109/TKDE.2013.109. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6547630
28. Zafra, A., Pechenizkiy, M., Ventura, S.: ReliefF-MI: An extension of ReliefF to multiple instance learning. Neurocomputing **75**(1), 210–218 (2012). DOI 10.1016/j.neucom.2011.03.052
29. Zaharia, M., Chowdhury, M., Das, T., Dave, A.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. NSDI'12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation pp. 2–2 (2012). DOI 10.1111/j.1095-8649.2005.00662.x. URL https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf
30. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark : Cluster Computing with Working Sets. HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing p. 10 (2010). DOI 10.1007/s00256-009-0861-0
31. Zhang, Y., Ding, C., Li, T.: Gene selection algorithm by combining reliefF and mRMR. BMC Genomics **9**(Suppl 2), S27 (2008). DOI 10.1186/1471-2164-9-S2-S27. URL http://bmcgenomics.biomedcentral.com/articles/10.1186/1471-2164-9-S2-S27
32. Zhao, Z., Cox, J., Duling, D., Sarle, W.: Massively parallel feature selection: An approach based on variance preservation. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **7523 LNAI**(PART 1), 237–252 (2012). DOI 10.1007/978-3-642-33460-3_21. URL http://link.springer.com/10.1007/978-3-642-33460-3{_}21