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 (FS) is a key research area in the machine learning and data mining fields, removing irrelevant and redundant features usually helps to reduce the effort required to process a dataset while maintaining or even improving the processing algorithm's accuracy. However, traditional algorithms designed for executing on a single machine lack scalability to deal with the increasing amount of data that has become available in the current Big Data era. ReliefF is one of the most important algorithms successfully implemented in many FS applications. In this paper, we present a completely redesigned distributed version of the popular ReliefF algorithm based on the novel Spark cluster computing model that we have called DiReliefF. Spark is increasing its popularity due to its much faster processing times compared with Hadoop's MapReduce model implementation. The effectiveness of our proposal is tested on four publicly available datasets, all of them with a large number of instances and two of them with also a large number of features. Subsets of these datasets were also used to compare the results to a non-distributed implementation of the algorithm. The results show that the non-distributed implementation is unable to handle such large volumes of data without specialized hardware, while our design can process them in a scalable way with much better processing times and memory usage.

Keywords feature selection · relieff · distributed algorithm · big data · apache spark

1 Introduction

Recently we have witnessed a vast increase in the amount of data that is being stored and processed by organizations of all types. As stated by Xindong et al. [33],

R. Palma
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, Universidad de Alcalá
Edificio Politécnico. Ctra. Barcelona km. 33.7. Alcalá de Henares, 28871, Madrid, Spain

the so-called big data revolution has come to us not only with many challenges but also with plenty of opportunities that these organizations are willing to embrace. According to Leskovec et al. [27], the main challenge is to extract useful information or knowledge from these huge volumes of data that enables us to predict or to better understand the phenomena involved in its generation. These tasks are commonly tackled using data mining.

As part of the data mining process, Feature Selection (FS) or Feature Subset Selection (FSS) is a crucial preprocessing step for identifying the most relevant attributes from a dataset. Removing irrelevant and redundant attributes not only can generate less complex and more accurate models but also a reduced dataset allows us to enhance the performance of many data mining schemes. FS algorithms are usually classified into three types: *wrappers*, *filters* and *embedded methods*. Wrappers refer to methods that require a learning scheme (classifier) as part of the search process, 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 carried out as part of the classification process such as the case of decision trees.

Another important FS classification, as stated by García et al. [10], is the one that comes from viewing the FS as a search problem with the aim of finding a high quality feature subset, and thereby, they classified the methods according to the type of search performed: *exhaustive*, *heuristic* and *stochastic*. However, as not all FS techniques can be viewed as these types of search, a fourth category needed to be included, *feature weighting*.

In this context, ReliefF [17] is 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 a filter FS method by defining a significance threshold and selecting features with quality above it. As a result of its advantages such as being able to work with nominal or continuous features, handling multiclass problems, detecting features interactions, handling missing data and noisy tolerance, it is considered one of the most convenient filter-based FS methods available [4].

On the other hand, after its 2004 seminal paper, Google's MapReduce [7,8] emerged as a programing model that simplified the development of scalable applications that process and generate large scale datasets. These applications are defined in terms of map and reduce tasks that are automatically parallelized by a programming framework and executed in clusters of tens or even 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 in 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 objective of recovering from failures, and this becomes a bottleneck for algorithms with an iterative nature like many of the ones used in machine learning and data mining, including the original ReliefF algorithm. For this reason, Spark [36] has gained much attention in the last few 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 DiReliefF[1], a distributed and scalable redesign of the original ReliefF algorithm based on the Spark computing model. DiReliefF allows us to deal with much larger datasets in terms of both instances and features than the traditional version would be able to handle. In addition, after presenting and discussing the design of the algorithm, we compare the results obtained with our distributed version and the version implemented in the WEKA [12] platform. This comparison was carried out using four publicly available datasets with numbers of instances in the order of 10^7 and a number of features that ranges in the order of 10^1 to 10^3. The main conclusion obtained is that is practically unfeasible to deal with such large volume of data using the traditional version on standard hardware, while the distributed version is able to scale smoothly.

2 Related Work

The ReliefF algorithm was first described by Kononenko [17] in 1994 and since then, many applications, extensions and improvements have been proposed. For instance, ReliefF is itself an extension of the original Relief algorithm developed by Kira and Rendell [16], the latter was initially limited to binary class problems while the former can handle multi-class problems. As further recent examples of those extensions, Reyes et al. [29] presented three of them for multi-label problems and compared them with previous extensions for the same purpose. Zafra et al. [34] extended ReliefF to the problem of multi-instance learning. Greene et al. [11] proposed an adaptation to enhance ReliefF's ability to detect feature interactions called SURF (Spatially Uniform ReliefF) and applied it in the genetic analysis of human diseases.

Although one of ReliefF's mayor flaws is its incapacity to detect redundant features, some attempts have been made to overcome this flaw. For example, Li and He [20] used a forward selection algorithm to select non redundant critical quality characteristics of complex industrial products. Zhang et al. [37] combined ReliefF with the mRMR (minimal-redundancy-maximal-relevancy) algorithm [25] to select non redundant gene subsets that best discriminate biological samples of different types.

As it might be expected, most FS algorithms have asymptotic complexities that depend on the number of features or instances, and in our case, ReliefF depends linearly on both of them [30]. 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 methods, including ReliefF, more scalable.

Recently, Canedo et al. [5] proposed 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 large amounts of instances. A similar approach was followed by Peralta et al. [26], who used the MapReduce model to implement a wrapper-based evolutionary search FS method. In this case, the data is split by chunks

[1] `https://github.com/rauljosepalma/DiReliefF`

of instances, and an evolutionary FS is performed over each of these pieces. Furthermore, the reduce step basically uses a simple majority voting of the selected features with a user-defined threshold to select the definitive subset of features. All the tests were carried out with the EPSILON dataset that we also use in this work (see Section 6).

Zhao et al. [38] presented a distributed parallel feature selection method based on variance preservation using the SAS High-Performance Analytics[2] proprietary software. Experimental work was carried out with both high dimensional and large number of instances datasets.

Kubica et al. [18] developed parallel versions of three forward search based FS algorithms. A wrapper with a logistic regression classifier is used to guide the search, which is parallelized using the MapReduce model.

It is also worth mentioning the recent work by Wang et al. [32] that uses the Spark computing model to implement feature selection strategy for classification of network traffic. Their approach basically consists of two phases. First, it generates a feature ranking based on the Fisher score. Then, a forward search is carried out using a wrapper approach. A drawback is that this method can only be applied to continuous data.

As it can be observed, none of the above contributions implements a purely filter approach. Even in the case of the framework proposed by Bolón-Canedo et al. [5] that allows us to apply filters to parts of the dataset, the results are then merged using a wrapper. However, in a recent publication, Ramírez-Gallego et al. [28] presented three implementations of an extended version of the popular mRMR feature selection filter, including a distributed version under Spark.

For the specific case of ReliefF, Huang et al. [14] proposed an optimization to improve the computation efficiency on large datasets, but this improvement is only useful when no random sampling of the instances 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, the experimental work they performed was carried out with datasets with a number of instances in the order of 10^4, which is still manageable by a single machine.

Finally, to the best of our knowledge, no published contribution has ever attempted to redesign the ReliefF filter method to a distributed environment as it is proposed here.

3 ReliefF

We briefly describe the original ReliefF algorithm by Kononenko [17] that will serve as a conceptual basis for the redesign presented in Section 5. ReliefF's central idea consists in evaluating the quality of the features by their ability to distinguish instances from one class to another in a local neighborhood, i.e., the best features are those that contribute more to increase distance between different class instances while contribute less to increase distance between same class instances. ReliefF, as mentioned above, is an extension of the original Relief method that is capable of working with multi-class, noisy and incomplete datasets.

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

Algorithm 1 ReliefF [30, 17]

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: $\overline{H} := -\sum_{j=1}^{k} diff(A, R_i, H_j)/k$
11: $\overline{M} := \sum_{C \neq cl(R_i)} \left[\left(\frac{P(C)}{1-P(cl(R_i))} \right) \sum_{j=1}^{k} diff(A, R_i, M_j(C)) \right] /k$
12: $W[A] := W[A] + (\overline{H} + \overline{M})/m$
13: **end for**
14: **end for**
15: **return** W

Algorithm 1 displays ReliefF's pseudo-code, mostly preserving the original notation used in [30]. 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-th 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 the other $k \cdot (c-1)$ neighbors are called *misses* (M), where c is the total number of classes, and $cl(R_i)$, represents the class of the i-th sample. Once the neighbors are found, their respective contributions to A-th feature are calculated. The contribution of the hits collection $\overline{H}$ is equal to the negative of the average of the differences between the actual instance and each hit. It should be noted 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 $\overline{M}$ is equal to the weighted average of the differences between the actual instance and each miss. This is a positive contribution because good features should help to differentiate between instances of a different class. The weights for this summation are defined according to the prior probability of each class, calculated from the dataset. Finally, it is worth mentioning that adding $\overline{H}$ and $\overline{M}$ and then dividing both by m simply indicates another average between the contributions of all m samples. Since the *diff* function returns values between 0 and 1, the ReliefF's weights will be in the range $[-1, 1]$, and must be interpreted in the positive direction: the higher the weight, the higher the corresponding feature's relevance.

The *diff* function is used in two cases in the ReliefF algorithm. The obvious one is between lines 10 and 11 to calculate the weight. It is also used to find distances between instances, defined as the sum of the differences over every feature (Manhattan distance). The original *diff* function used to calculate the difference between two instances I_1 and I_2 for a specific feature A is defined in (1) for nominal features, and as in (2) for numeric features. However, the latter 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 (3), was proposed to deal with this problem [13]. The idea behind it is to relax the equality comparison on (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 addition, there are other versions of the *diff* function to deal with missing data. However, since the datasets chosen for the experiments in this work do not have missing values, we do not consider them here.

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

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

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

4 Spark Cluster Computing Model

We now briefly describe 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 also provide a short comparison with other existent computing models such as MapReduce, with the aim of justifying our selection of Spark.

The main concept behind the Spark model is the Resilient Distributed Dataset or in short RDD. Zaharia et al. [36,35] defined an RDD as a read-only collection of objects, that is partitioned and distributed across the nodes of a cluster. It has the ability to automatically recover lost partitions through the record of lineage that knows the origin of the data and optionally, the calculations that went through it. Even more relevant is the fact 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 the traditional parallel programming.

There are two types of operations that can be executed on an RDD: (i) actions and (ii) transformations. On the one hand, *actions* are the mechanism that permit to obtain results from a Spark cluster; five commonly used actions are: *reduce*, *sum*, *aggregate*, *takeSample* and *collect*. The action *reduce* is used to aggregate the elements of an 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 its return type can be different from the 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 *takeSample* and *collect* are also similar. The former takes

an amount of elements and returns a random sample of this size from the RDD. The latter simply returns an array with all the elements in the RDD. This has to be done carefully to prevent exceeding the maximum memory available at the driver (explained below).

On the other hand, *transformations* are the mechanism for creating an RDD from another one. Since RDDs are read-only, a transformation does not affect the original RDD but creates a new one. Four of the most important transformations are: map, $flatMap$, $reduceByKey$ and $filter$. The first two: map and $flatMap$ are similar. Both 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. In the case of $flatMap$, the applied function takes a single element but it can return zero or more elements, therefore the resulting RDD is not required to have the same number of elements as the original one. The next transformation is $reduceByKey$, it can only be applied on an RDD where there is a key associated to each element, the transformation uses this keys to group the elements and applies a $reduce$ function to these groups returning an RDD with unique keys each one associated to the result of each $reduce$. Finally, $filter$ is straightforward, it receives a boolean function to discriminate RDD elements to return a subset of it.

With respect 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. Once the driver has this access, it coordinates the execution of a user application by assigning tasks to the executors, which are programs that run in worker nodes (slaves). By default only one executor is run per worker. With regard to the data, RDD partitions are distributed across the worker nodes, and the number of tasks launched by the driver for each executor will be in correspondence with 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 observed in Figure 1.

4.1 Other Cluster Computing Models

Spark has quickly stood out from other cluster computing models (e.g. MapReduce as the most relevant) as the preferred platform due to its own advantages. First of all, Spark was designed from the beginning to efficiently handle iterative jobs in memory, such as the ones used by many data mining schemes. This has lead to the quick development of a machine learning library [24] that contains redesigned distributed algorithms such as the one studied in this work. Moreover, besides the Spark author's own comparison [36,35], others such as Shi et al. [31] 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. Finally, as previously stated, Spark provides a wider range of operations other than $flatMap$ and $groupByKey$.

It is worth noting that other models have already tried to fulfill the lack of efficient iterative job handling of MapReduce. Two of them include HaLoop [6] and Twister [9]. However, even though they support executing iterative MapReduce

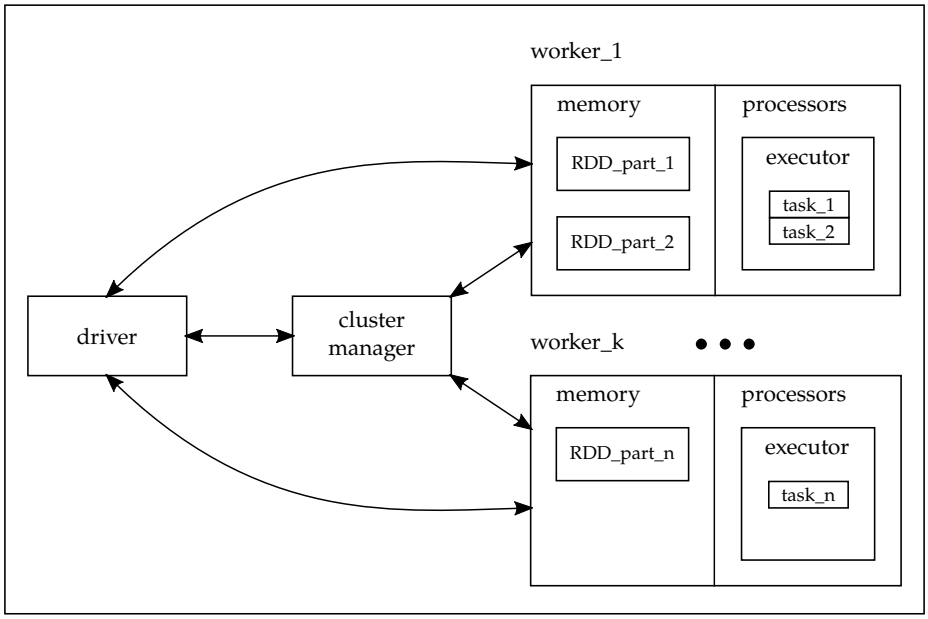

Fig. 1 Spark Cluster Architecture

jobs, automatic data partitioning, and Twister has also the ability to keep it in-memory, they both prevented an interactive data mining and can indeed be considered subsets of Spark functionality. In any case, both projects have become outdated.

A recently work by Liu et al [22] compared parallelized versions of a neural network algorithm over Hadoop, HaLoop and Spark, concluding that Spark was the most efficient in all cases.

5 DiReliefF

In this section, we describe our proposed algorithm. The first design decision is where to concentrate the parallelization effort. As the ReliefF algorithm could 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 [30]. However, parallelizing the algorithm in such way ties the parallelization to the number of samples m, and prevents Spark from doing optimizations based on the resources available, the size of the dataset, the number of partitions and the data locality. This would also require that every thread would read through the whole dataset, while as we show below, there is only one pass needed to process the distances and calculate the feature weights. Furthermore, continuing with Robnik and Kononenko discussion, ReliefF algorithm's complexity is $\mathcal{O}(m \cdot n \cdot a)$, where n is the number of instances in the dataset, m is the number of samples taken from the n instances and a is the number of features. Moreover, the most complex operation is the selection of the k nearest neighbors for two reasons: first, the distance from the current sample

to each of the instances must be calculated with $\mathcal{O}(n \cdot a)$ steps; and second, the selection must be carried out in $\mathcal{O}(k \cdot log(n))$ steps. As a result, the parallelization is focused on these stages rather than on the m independent iterations.

The ReliefF algorithm can be considered as a function applied to a dataset DS, having as input parameters the number of samples m and the number of neighbors k, and returning as output an $a-$size vector of weights W, as shown in (4). Thus, the ReliefF algorithm can be interpreted as the calculation of each individual weight $W[A]$, using (5), where *sdiffs* (6) represents a function that returns the total sum of the differences in the $A-$th feature between a given instance R_i, and a set $NN_{C,i}$ of k neighbors of this instance where all belong to a particular class C. Using this, we can state a series of steps in order to obtain the desired weights vector W. These steps are summarized in Algorithm 2, shown graphically in Figure 2 and are fully described in the following paragraphs.

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

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

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

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

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

Algorithm 2 DiReliefF

```
1: DS = input dataset
2:
3: {Begin steps distributed in cluster}
4:
5: (MAX, MIN) := max and min values for all continuous feats via reduce over DS
6: P := all class priors via reduceByKey over DS
7: R := m samples obtained via takeSample from DS
8: DD := distances RDD from DS to R via map over DS
9: NN = global nearest neighbors matrix via aggregate over DD
10:
11: {End of distributed steps}
12:
13: SDIF = sum of differences matrix using diff, MAX and MIN over NN
14: W = weights vector using SDIF, P and equation (5)
15: return W
```

Given the initial definitions and assuming that the features types (nominal or continuous) are stored as metadata, we first calculate the maximum and minimum values for all continuous features in the dataset. These values are needed by the *diff*

function (see (3) and (2)) in line 13. Our implementation, as the original version, uses (1) for nominal features and selects between (3) and (2) for continuous features via an initialization parameter. The task of finding maximum and minimum values is efficiently achieved applying a *reduce* action with a function $fmax$ ($fmin$) that given two instances returns a third one containing the maximum (minimum) values for each continuous feature. This is shown for maximum values on (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 calculates the prior probabilities of all classes in DS. These values can be essentially obtained by the means of a *map* and a $reduceByKey$ transformation. The former returns a dataset with all instance classes paired with a value of one. The latter sums these 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 array rather than an 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 in (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. We first find the distances from every sample m to each of the n instances. This can be directly accomplished by the 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 (as shown in (11)).

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

Next, as shown in (12), we obtain the nearest neighbors NN matrix by using an *aggregate* action, where each element $NN_{C,i}$ of this matrix is a set of the k nearest neighbors of a sample R_i belonging to a class C. The use of an *aggregate* action is an important design decision since it allows to obtain the NN matrix

with a single pass through DD. Previous experiments showed that trying to find the neighbors for each sample in independent loops was much less efficient.

As stated before, the *aggregate* action has two steps. The first step is defined in the function $localNN$ that returns a local neighbors matrix LNN for each partition of the RDD. This matrix has a similar structure as the NN matrix but each element is treated instead as a k-sized binary heap that is used to incrementally store the nearest neighbors found during the traverse of the local partition.

The second step of the *aggregate* action 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 that the $localNN$ can start aggregating neighbors to it. Lastly, note that since the NN matrix is obtained via an action, it is a local object and not an RDD. This step, which has been fully implemented using parallel operations, is the most complex step in the algorithm.

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

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 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-th feature between the $NN_{C,i}$ set of neighbors and the R_i sample. Each element of the vector $SDIF_{C,i}$ can be calculated by mapping the *diff* function over the A-th feature of all the instances in the $NN_{C,i}$ set and then summing these differences. Observe that this *map* and *sum* functions are not RDD-related anymore, but local equivalents that execute on the driver's local threads. Finally, note that each element of each vector of the matrix $SDIF$ effectively represents the value shown in (6), thus, the final vector of weights, W, can be easily calculated by applying (5) using the already obtained P set with the prior probabilities. The above discussion is depicted in 13.

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

Finally, there is one implementation issue worth mentioning. As previously stated, an RDD is designed to be kept in memory but this does not happen automatically, therefore, if the RDD is going to be used in future operations it must be explicitly cached. In our case, the most complex part of the algorithm is the calculation of the DD and NN matrices. This calculation is effectively performed in a single pass through the dataset initiated by the *aggregate* action, and therefore, caching of any intermediate result would indeed cause a waste of resources. However, the initial part of the algorithm that requires the calculations of the maximum, minimum and priors, each require a pass through the dataset DS and therefore can take advantage of caching but only when it fits in the distributed memory of the cluster. When it does not, caching does not help because its benefits are outweighed by the time needed for writing the dataset to disk. As a result, in our implementation caching is disabled by default but can be enabled with a parameter (it should be enabled only when we can assure that the dataset fits in the distributed memory).

6 Experimental Evaluation

In this section, experimental results obtained from different executions of the proposed algorithm are presented. The experiments were performed with the aim of testing the algorithm scalability and its time and memory consumption with respect to a traditional version in WEKA [12]. Further tests were also performed in order to observe sample sizes where the algorithm's weights become stable. Since the distributed version was designed to return exactly the same results as the non-distributed one, there was no need to perform tests comparing them.

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
- Hypervisor: KVM (qemu-kvm 1.5.3)
- Guests Processors: 2
- Guests Memory: 16 GB
- Guests Storage: 500 GB

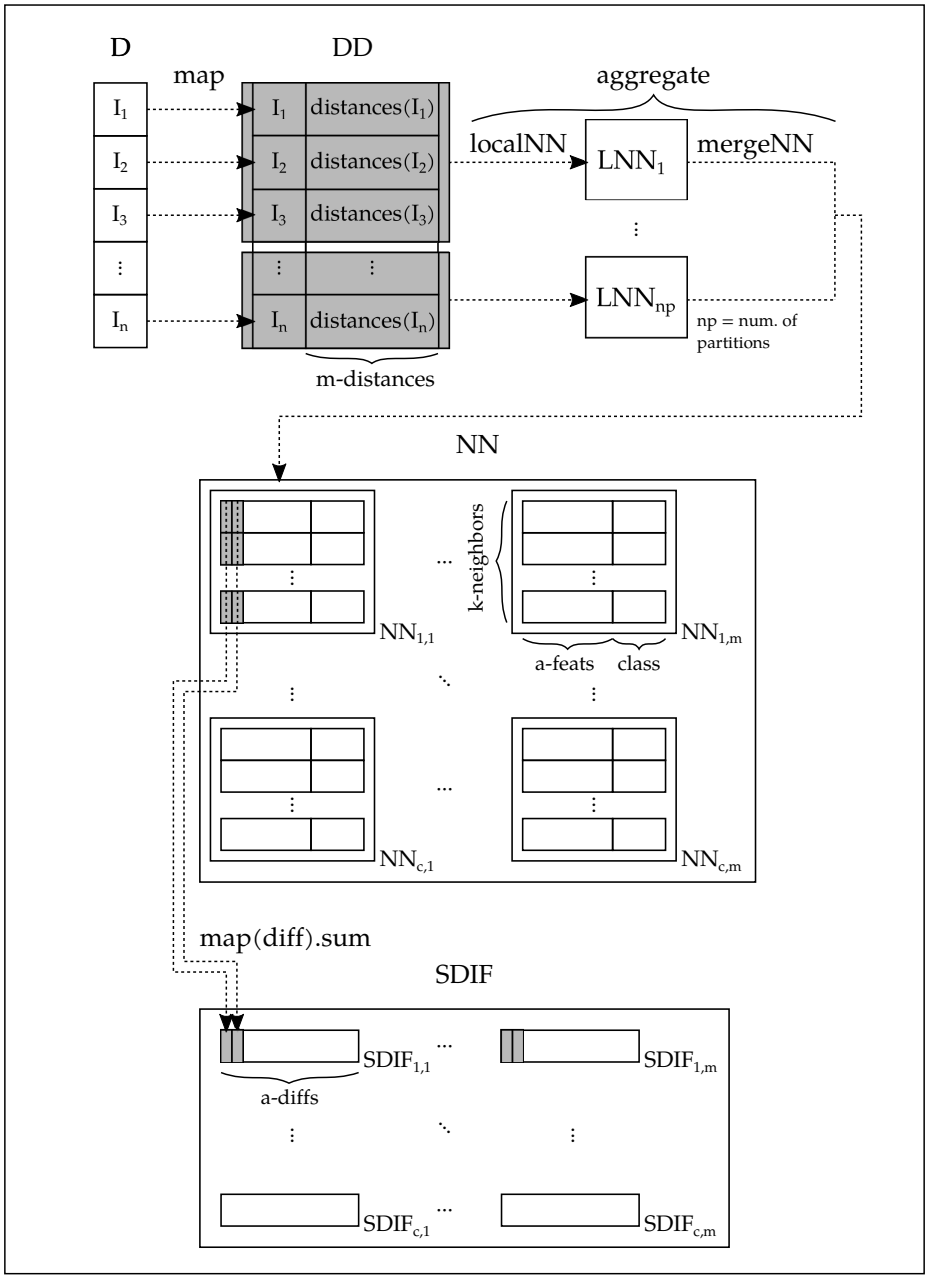

Fig. 2 DiReliefF's Main Pipeline

Table 1 Datasets used in the experiments

Dataset	No. of Inst.	No. of Feats.	Features Types	Problem Type
ECBDL14	∼33.6 million	632	Numerical and Categorical	Binary
HIGGS	11 million	28	Numerical	Binary
KDDCUP99	∼5 million	42	Numerical and Categorical	Multi-class
EPSILON	1/2 million	2,000	Numerical	Binary

- Java version: OpenJDK 1.8
- Spark version: 1.6.1
- HDFS version: 2.6.4
- WEKA version: 3.8

During the first part of the experimental work, the ECBDL14 [2] dataset was used. This dataset comes from the Protein Structure Prediction field, and it was used during the ECBLD14 Big Data Competition of the GECCO'2014 international conference. The dataset has approximately 33.6 million instances, 631 attributes, 2 classes, 98% of negative examples and occupies about 56GB of disk space. In a second part of the experimental work, we use the other three datasets described in Table 1. HIGGS [3], from the UCI Machine Learning Repository [21], is a recent dataset that represents a classification problem that distinguishes between a signal process which produces the Higgs bosons and a background process which does not. KDDCUP99 [23] represents data from network connections and classifies them between normal connections and different types of attacks (a multi class problem). Finally, EPSILON is an artificial data set built for the Pascal Large Scale Learning Challenge in 2008[3].

Initially, all tests are run with a number of neighbors $k = 10$ which is a typical choice [17], and a relatively low number of samples $m = 10$ to keep execution times reasonable. However, during the stability tests larger number of samples are used. 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 the tests with the traditional version of ReliefF.

Regarding the rest of the implementation parameters, two of them were left fixed for all the experiments. The first one selects the original *diff* function in (2) for the distance evaluation of numeric features, as this is the only one implemented in the WEKA version. However, it is worth mentioning that since the ramp function in (3) requires a constant number of extra operations, selecting it will simply multiply by a constant factor the total number of operations that the algorithm has to execute, while also bringing the benefits mentioned before and explained by Hong et al. [13]. The second parameter refers to whether or not apply caching in Spark. As stated above, caching provides benefits only when the entire dataset fully fits in memory. Therefore, it was decided to disable caching for all the experiments in order to facilitate the interpretation of the results.

An important configuration issue refers to driver memory consumption. In Spark computation model, there is no communication between tasks, so all the

[3] `http://largescale.ml.tu-berlin.de/about/`

task's 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 1GB but it was set to 6GB for all the experiments performed. This is specifically important for tests involving larger matrix sizes, i.e., those with higher values of m or c.

6.1 Empirical Complexity

Figure 3 shows time and memory consumption behavior of the distributed version of ReliefF version versus the one implemented in the WEKA platform for incrementally sized samples of the ECBDL14 dataset. This dataset was selected because it has the largest amount of instances and allows to show the limits of the WEKA implementation. Since the number of operations performed by the ReliefF algorithm depends only on the parameters m, n and a, and not on the internal characteristics of the data, the other three datasets were not considered for this part. To make the comparison possible, the WEKA version was executed under the host environment with no virtualization. It is worth noting that for the WEKA version a 30% sample was the largest that could be tested because larger samples would need more memory than is available in the system (showing the lack of scalability). The distributed version, in addition to being able to handle the whole dataset, preserves a linear behavior in relation to the number of instances, and it is also capable of processing data in less time by leveraging the cluster nodes. Another fact to observe is the change in the slope of the linear behavior observed by the Spark version between the 40% and 50% samples of the dataset. This is due to the fact that during this interval the dataset overflows the available memory in the cluster and the Spark engine starts using disk storage. In other words, the algorithm is capable of maintaining a linear complexity even when the dataset does not fit into memory.

The mentioned overflow issue can be observed on the right graph in Figure 3. This graph shows a previously mentioned advantage of Spark, i.e, it is designed to run in commodity hardware. A simple look to the memory consumption of the traditional version shows that the required amount of memory quickly overgrows beyond the limits of an ordinary computer. However, in our distributed version, we can observe that using nodes, in our case with 16GB of memory, is enough to handle the task.

Analogous run time results are obtained by varying the number of features and the number of samples (see Figure 4) confirming an empirical complexity equivalent to the original one, i.e., $\mathcal{O}(m \cdot n \cdot a)$.

6.2 Scalability

In order to test scalability and to keep execution times within manageable limits, the largest test was performed with a 30% sample size of the ECBDL14 dataset. Following this, 10% and 1% sample sizes were used. Also, in order to examine a smaller dataset, tests with 50% and 10% samples of the HIGGS dataset were run.

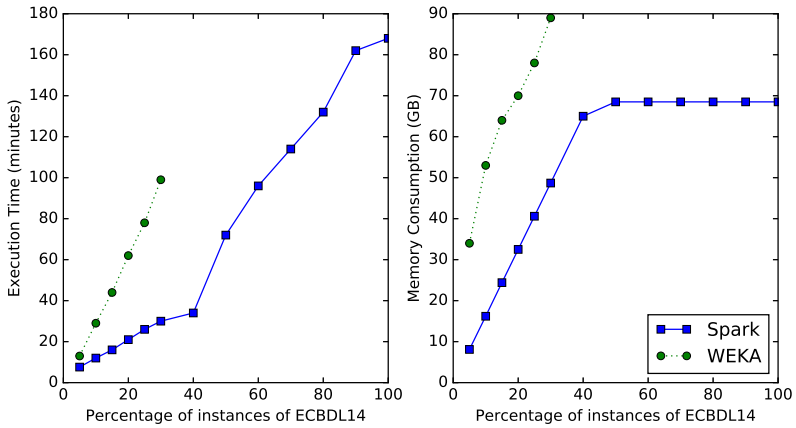

Fig. 3 Execution time and memory consumption of Spark DiRelieF and WEKA ReliefF versions

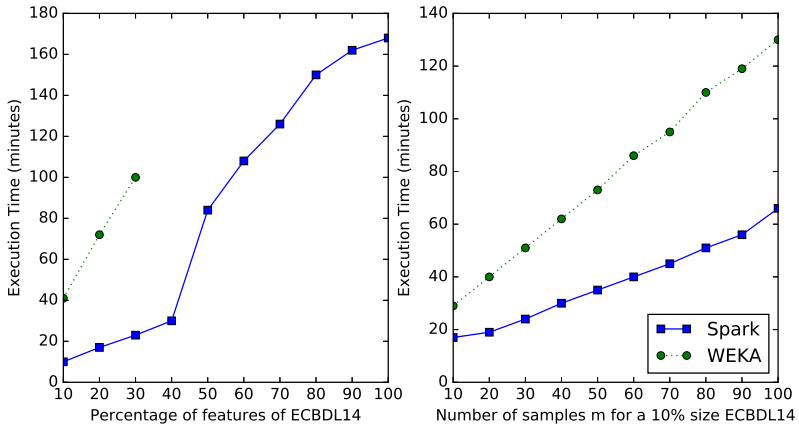

Fig. 4 Execution time of Spark DiReliefF and WEKA ReliefF with respect to parameters a and m

Figure 5 shows the behavior of the distributed algorithm with respect to the number of cores used. As it can be observed, adding cores at the beginning greatly contributes to reducing the execution time, but once a threshold is reached, the contribution is rather subtle or even null. Such threshold depends on the size of the dataset, in such a way that larger datasets can take further advantage of larger number 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. In this latter case, adding more nodes will not provide any performance improvement. As an example to quantify this fact, it can be observed in Figure 5 that the 30% sized sample of the ECBDL14 dataset can take advantage of 7 cores, while the 1% sample size only can take advantage of 4 cores. Similarly, there is no practical advantage of using more than 1 core for the 10% sample size of the HIGGS dataset.

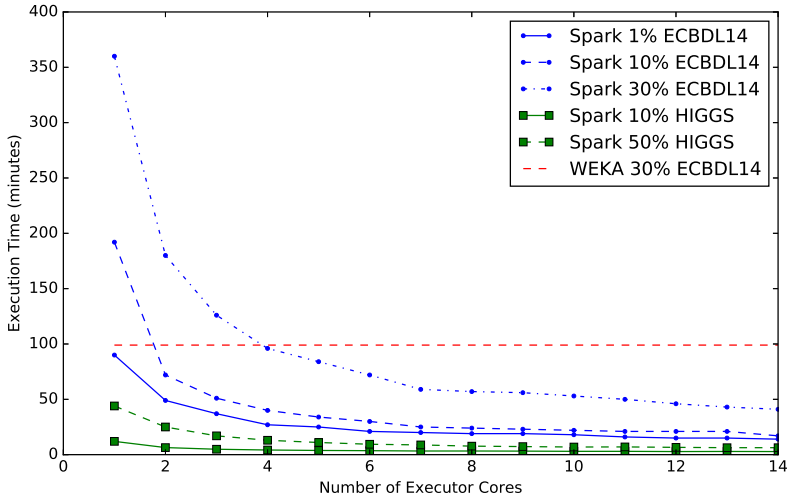

Fig. 5 Execution time of Spark DiReliefF and WEKA ReliefF with respect to the number of cores involved

Figure 5 also shows with an horizontal line the execution time of the WEKA version for 30% sample of the ECBDL14 dataset. 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 4 or less cores are involved. This is because it does not need 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. For this reason, the distributed version of ReliefF is only useful for large datasets.

6.3 Stability

The following set of tests was made in order to check the stability of the algorithm. In this case, with stability we refer to similarity of the assigned weights for different executions in the same dataset. The stability is important in the case of ReliefF since it selects a random sample in every execution.

Many stability measures have been defined. A commonly used measure is the Consistency Index presented by Kuncheva [19]. However, it cannot be directly applied because it requires to define a threshold for the selection of a subset. Kalousis et al. [15] proposed the use of the Pearson correlation coefficient to measure the similarity between features rankings, but previous tests showed that it returns unstable results for the EPSILON and HIGGS (pure numerical) datasets. For these reasons, we opted here for a simpler stability indicator, an average of the absolute differences between every feature weights of two rankings. More formally, if we have two feature ranking vectors $W1$ and $W2$, the average difference between them is calculated as shown in (14).

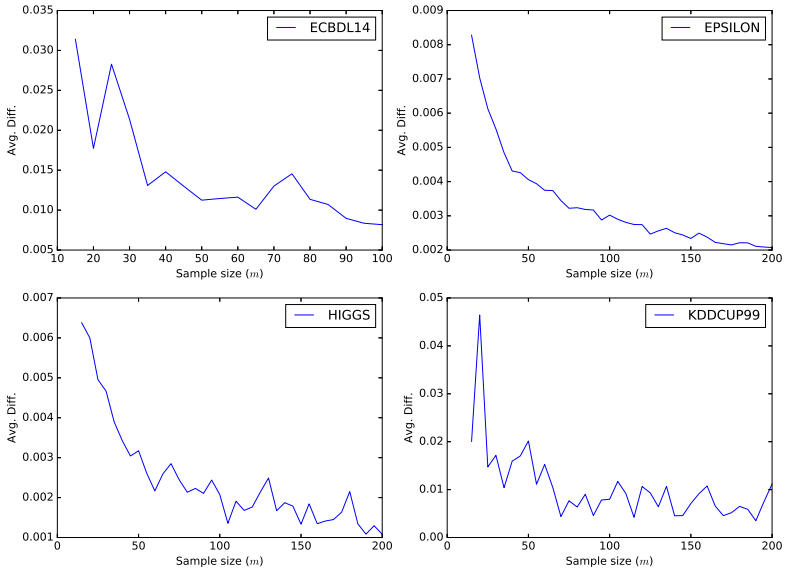

Fig. 6 DiReliefF's average difference in weight ranks for increasing values of m in different datasets

$$
\begin{aligned}
W1 &= (w1_1, w1_2, \cdots, w1_a) \\
W2 &= (w2_1, w2_2, \cdots, w2_a) \\
AvgDiff &= \frac{1}{a} \sum_{A=1}^{a} |w1_A - w2_A|
\end{aligned} \tag{14}
$$

As Robnik and Kononenko [30] stated, the correct sample size m is problem dependent, and as a consequence, we used all of the datasets described in Table 1. Moreover, their experiments showed that as the number of examples increases, the required sample size diminishes. Figure 6 shows the results obtained and evidences that the biggest gain in stability is obtained with a sample size between 50 and 100 instances, thereby confirming that relative small sample sizes are enough to obtain stable results in the large scale datasets tested.

7 Conclusions

In this work, we have presented DiRelieF, a distributed version of the well-known ReliefF feature selection algorithm. This version was implemented using the emerging Apache Spark programming model to deal with current Big Data requirements such as failure recovery and scalability over a cluster of commodity computers. Even when the ReliefF algorithm is easily parallelizable by associating jobs to each

independent iteration and then merging its results, this method ties the number of jobs to the sample size (number of iterations) and requires an equal number of passes through the dataset. For this reason we designed an alternative version based on the Spark *map* and *aggregate* operations and on the use of binary heaps. This method does not suffer the problems mentioned and requires a single pass through the whole dataset to perform the main operations of the algorithm compared to the m passes required by the original version.

As part of the experimental work, we have also compared our proposal with a non distributed version of the algorithm implemented on the WEKA platform. Our results showed that the non distributed version is poorly scalable, i.e., it is unable to handle large datasets due to memory requirements. Conversely, our version is fully scalable and provides better execution times and memory usage when dealing with very large datasets. Our experiments also showed that the algorithm is capable of returning stable results with sample sizes that are much smaller than the size of the complete dataset.

Acknowledgments

The authors thank the anonymous reviewers for helping us to improve the manuscript, the National Autonomous University of Honduras and the University of Alcalá. R. Palma-Mendoza holds a scholarship from the Spanish Fundación Carolina. This research was also supported by projects BadgePeople, TIN2016-76956-C3-3-R and TIN2015-71841-REDT.

References

1. Apache Software Foundation: Hadoop. URL `https://hadoop.apache.org`
2. Bacardit, J., Widera, P., Márquez-chamorro, A., Divina, F., Aguilar-Ruiz, J.S., Krasnogor, N.: Contact map prediction using a large-scale ensemble of rule sets and the fusion of multiple predicted structural features. Bioinformatics **28**(19), 2441–2448 (2012). DOI 10.1093/bioinformatics/bts472
3. Baldi, P., Sadowski, P., Whiteson, D., Neyman, J., Pearson, E., Hornik, K., Stinchcombe, M., White, H., Hochreiter, S., Bengio, Y., Simard, P., Frasconi, P., Baldi, P., Sadowski, P., Hinton, G.E., Osindero, S., Teh, Y.W., Aad, G., Aaltonen, T., Alwall, J., Sjostrand, T., Cheng, H.C., Han, Z., Barr, A., Lester, C., Stephens, P., Hocker, A., Aaltonen, T.: Searching for exotic particles in high-energy physics with deep learning. Nature Communications **5**, 694–706 (2014). DOI 10.1038/ncomms5308. URL `http://www.nature.com/doifinder/10.1038/ncomms5308`
4. 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`
5. 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`
6. 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
7. 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

8. 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
9. 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
10. 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
11. 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
12. 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
13. 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
14. 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
15. Kalousis, A., Prados, J., Hilario, M.: Stability of feature selection algorithms: a study on high-dimensional spaces. Knowledge and Information Systems **12**(1), 95–116 (2006). DOI 10.1007/s10115-006-0040-8. URL http://dx.doi.org/10.1007/s10115-006-0040-8
16. Kira, K., Rendell, L.A.: A practical approach to feature selection. Proceedings of the ninth international workshop on Machine learning pp. 249–256 (1992)
17. 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
18. 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
19. Kuncheva, L.I.: A stability index for feature selection. International Multi-conference: artificial intelligence and applications pp. 390–395 (2007). URL papers2://publication/uuid/FC5277DF-B494-4316-8D02-E8CE794BAE37
20. 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
21. Lichman, M.: UCI Machine Learning Repository (2013). URL http://archive.ics.uci.edu/ml
22. 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
23. Ma, J., Saul, L.K., Savage, S., Voelker, G.M.: Identifying Suspicious URLs : An Application of Large-Scale Online Learning. In: Proceedings of the International Conference on Machine Learning (ICML). Montreal, Quebec (2009)
24. 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
25. 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
26. 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

27. Leskovec, J., Rajaraman, A., Ullman, J.D.: Mining Massive Datasets, 2nd Edt., Cambridge University Press (2014). URL http://infolab.stanford.edu/~ullman/mmds/book.pdf
28. Ramírez-Gallego, S., Lastra, I., Martínez-Rego, D., Bolón-Canedo, V., Benítez, J.M., Herrera, F., Alonso-Betanzos, A.: Fast-mRMR: Fast Minimum Redundancy Maximum Relevance Algorithm for High-Dimensional Big Data. International Journal of Intelligent Systems (2016). DOI 10.1002/int.21833. URL http://doi.wiley.com/10.1002/int.21833
29. 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
30. Robnik-Šikonja, M., Kononenko, I.: Theoretical and empirical analysis of ReliefF and RReliefF. Machine learning **53**(1-2), 23–69 (2003)
31. 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
32. 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
33. 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
34. 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
35. 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
36. 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
37. 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
38. 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