Multithreaded and Spark parallelization of feature selection filters

Carlos Eiras-Franco[a,*], Verónica Bolón-Canedo[a], Sabela Ramos[b], Jorge González-Domínguez[b], Amparo Alonso-Betanzos[a], Juan Touriño[b]

[a]*Computer Science Dept., University of A Coruña, 15071 A Coruña, Spain*

[b]*Dept. of Electronics and Systems, University of A Coruña, 15071 A Coruña, Spain*

Abstract

Vast amounts of data are generated every day, constituting a volume that is challenging to analyze. Techniques such as feature selection are advisable when tackling large datasets. Among the tools that provide this functionality, Weka is one of the most popular ones, although the implementations it provides struggle when processing large datasets, requiring excessive times to be practical. Parallel processing can help alleviate this problem, effectively allowing users to work with Big Data. The computational power of multicore machines can be harnessed by using multithreading and distributed programming, effectively helping to tackle larger problems. Both these techniques can dramatically speed up the feature selection process allowing users to work with larger datasets. The reimplementation of four popular feature selection algorithms included in Weka is the focus of this work. Multithreaded implementations previously not included in Weka as well as parallel Spark implementations were developed for each algorithm. Experimental results obtained from tests on real-world datasets show that the new versions offer significant reductions in processing times.

Keywords: Multithreading, Spark, Feature selection, Machine Learning

*Corresponding authors
Email address: `carlos.eiras.franco@udc.es` (Carlos Eiras-Franco)

1. Introduction

Massive amounts of data are generated on a daily basis nowadays. As an example, IBM estimated that in the time span between 2010 and 2012, 90% of the data worldwide were produced at a rate of 2.5 exabytes per day [1]. This data-generating trend has sparked interest in data analytics, which in turn has created the need for new tools, algorithms and methodologies that can cope efficiently with such massive amounts of data. Consequently, the dimensionality (samples $\times$ features) of datasets being analyzed in machine learning problems has been steadily growing in the last years. Taking the datasets posted in the popular libSVM Database [2] as a reference, their size has increased five hundredfold. This results in a challenge for traditional machine learning algorithms, as overfitting can negatively impact their performance, more complex models are harder to interpret and both speed and efficiency of these algorithms decrease as the dimensionality increases.

This situation has spawned a number of techniques designed to deal with big dimensionality datasets. This dimensionality can refer to samples, features or both. In the case in which we confront with datasets containing numerous features, feature selection techniques are mandatory. Feature selection consists in the process of determining the relevant features and trying to remove as much irrelevant and redundant information as possible, without leading to a degradation in the classification performance.

The Weka (Waikato Environment for Knowledge Analysis) suite [3] is a very popular machine learning platform that has been downloaded over six million times. It can be used as a stand-alone application or imported as a library from the user's code. Feature selection is among its functionalities with several algorithms available to the user. This ample range of algorithms included in Weka makes its use widespread among data scientists for data analysis and for the development and testing of new algorithms. In addition, the fact that Weka runs on Java and is designed with single-machine setups in mind, makes it very suitable for the average user. Nevertheless, some of the implementations in Weka still struggle when processing large datasets, requiring very long execution times, effectively limiting the size of the datasets that can be analyzed with it. An improvement in the time efficiency of these algorithms will enable its many users to process large datasets that up to now were out of reach for these implementations.

On the other hand, and to specifically address the Big Data issue, new parallel programming solutions have been created in the last decade, such as MapReduce [4], that was implemented in the open-source solution Apache Hadoop [5] or, more recently, Apache Spark [6], which aimed at being a solution to Big Data analysis. The advent of these new technologies led to the creation of parallel machine learning libraries: Mahout [7] that runs on top of Apache Hadoop, and MLlib [8] that uses Apache Spark, are some examples. Although these libraries contain a wide variety of machine learning algorithms, they do not provide many options when it comes to feature selection.

Spark is designed for distributed computing and can achieve great performance processing large amounts of data, but few implementations of feature selection algorithms are available. Moreover, to be able to use the Mahout or MLlib libraries, the user needs to have a Hadoop or Spark installation and, although they can run on single-machine environments, a cluster of computers would be needed to fully exploit these libraries, which is not always available for regular users.

The aim of this work is to obtain new implementations of these popular feature selection algorithms that are able to tackle sizable problems in different environments and find out the suitability of these implementations to different amounts of computing resources. To this end, multithreaded implementations for Weka and distributed versions in Spark will be proposed. This will allow users to analyze larger datasets in shorter times and choose the most adequate implementation to the resources available to them.

This paper is organized as follows: Sections 2 and 3 are an overview of feature selection and parallelization approaches respectively. Section 4 describes the algorithms that are the object of this paper. The results of our tests are presented in Section 5 and in Section 6 we discuss our conclusions.

2. Feature selection

Feature selection is the name given to the process that detects relevant features and discards those that are redundant or irrelevant. The goal of this technique is to obtain a subset of features that has minimum degradation of performance when used by a classifier while describing the given problem properly. It simplifies the dataset both in size and in complexity of understanding [9], which leads to simpler and faster classification algorithms, better problem comprehension and reduced storage requirements.

2.1. Feature selection methods

Feature selection methods can be classified into two categories: individual evaluators or subset evaluators. Individual evaluators are also called rankers and they assign a weight to each attribute that represents its relevance. Subset evaluators, on the contrary, employ a search strategy to determine a candidate subset of features and have the advantage of removing redundant attributes at the cost of being more complex. According to the relationship with the learning method used, feature selection methods can also be divided as follows [10]:

- **Filters** are methods that are applied independently of the induction process. They are, in general, computationally inexpensive.
- **Wrappers** use the induction algorithm as a black box to evaluate the fitness of each candidate subset. This results in algorithms that are computationally demanding but more accurate.
- **Embedded methods** perform feature selection in the process of training and are typically specific to given learning algorithms.

In this paper, three of the most commonly used filter methods (InfoGain, RELIEF-F and CFS) and an embedded method (SVM-RFE) were selected for reimplementation using a parallel approach. The first two filters are rankers that return an order for the features to be discarded below a threshold of the user's choice and they are included in the Weka suite:

- **Information Gain (InfoGain)** [11] is a filter that computes the mutual information of the different features with respect to the class and provides an ordered ranking of all the features according to this value.
- **RELIEF-F** [12] is a heuristic estimator built upon the RELIEF algorithm [13] that deals efficiently with noisy and incomplete datasets and with multiclass problems. It works by locating its nearest neighbors for each instance from the same and opposite class and updating the weights of each feature accordingly.

The remaining two algorithms are subset evaluators. To perform feature selection they search through the space of all possible attribute combinations for the set that offers a better score according to a heuristic method that depends on the algorithm.

- **CFS** [14] is a subset evaluator independent from the induction process that tries to identify correlations between attributes and the class.
- **SVM-RFE** [15], which stands for Support Vector Machine Recursive Feature Elimination is an embedded method that filters the attributes iteratively using a SVM at each stage to rank them.

The choice of these algorithms was made to obtain a set of tools that are well suited to a wide range of datasets.

CFS and InfoGain perform well when the data has a large number of attributes when compared to the number of instances and are very fast, but they do not perform as well when there is noise in the inputs. RELIEF-F is very good at eliminating redundant and correlated features, even when there is noise in the inputs and attributes are non-linear, but it is much slower and does not perform well when few examples are available. Lastly, SVM-RFE detects correlation and redundancy even with few examples, but it performs poorly when there is noise in the inputs and is very time consuming [9].

Table 1 shows the theoretical complexity of the four methods described above.

Table 1: Theoretical complexity of the four feature selection methods focus of this work (where m is the number of examples and n is the number of attributes)

Method	Complexity
InfoGain	nm
RELIEF-F	nm^2
CFS	n^2m
SVM-RFE	$max(n, m)m^2$

3. Parallel approaches

The main purpose of this work is to parallelize the standard implementations of RELIEF-F, InfoGain, CFS and SVM-RFE. In order to empower Weka users, multithreaded implementations are proposed. Furthermore, to enable users that can access computational clusters, we developed and tested Spark versions of the algorithms.

3.1. Multithreaded processing

Multithreading allows users to take advantage of multicore systems without imposing the overhead of creating multiple processes and providing direct access to a common address space. However the creation and management of threads introduces a computational overhead that makes the use of threads suboptimal when the tasks parallelized have low complexity.

Java provides parallel programming support in the core of the language. This feature enables programmers to write code that exploits multithreading without the need to use any external library. Since Weka is written in Java, we use this support to implement our multithreaded parallel codes. We divide the feature selection algorithms in tasks that can be performed in parallel, which allows us to exploit the computational power of multicore machines.

3.2. Parallelization with Apache Spark

To alleviate the difficulties of developing distributed programs, a team of Google engineers developed the MapReduce framework [4] that handles the common aspects of distributed programs, providing the programmer with a tool to run parallel programs without having to worry about anything but the implementation of the algorithm.

The programming paradigm introduced by MapReduce requires the tasks to be divided in two separate steps: the Map phase, that applies a function given by the user to every element; and the Reduce phase, that combines the resulting values. Oftentimes elements consist of key-value pairs and the Reduce phase merges results that have the same key, although this is not mandatory. The abstraction resulting of decomposing a job in simple Map and Reduce functions allows the framework to divide both data and code across the computing nodes, a task performed by a master node. Typically, the framework splits the data in as many chunks as nodes are available, distributes it among them so that each node can apply the Map function to the assigned elements. The results are then rearranged by the master node, using a key partitioning scheme, and distributed again back to the nodes so that they perform the Reduce phase.

MapReduce was implemented in the open-source framework Hadoop [5] and rapidly achieved great popularity for its reliability and scalability. Still, this direct implementation left room for an important improvement that was later implemented by the Spark [6] framework: the transition between the Map and Reduce phase requires data to be shuffled by the master node and redistributed to the nodes, in a time-consuming process that is unnecessary when several Map transformations need to be applied before the Reduce phase or in iterative algorithms. By avoiding unneeded data movement and introducing other optimizations Spark performs several times faster than Hadoop for certain applications [16].

Spark allows the programmer to manage work distribution by means of using Resilient Distributed Datasets (RDDs), an abstraction that represents a read-only set of objects that is distributed across multiple machines. RDDs can be transformed, performing an operation on each element, which can be done in parallel in each node, and they can be reduced, combining elements, which requires that the whole dataset is shuffled and redistributed to the nodes in a time-consuming process. Additionally, data can be sent to the nodes to work with by using broadcast variables, and the worker nodes can write increments to special variables named accumulators.

We decomposed the feature selection algorithms in independent tasks for a Spark implementation that will allow the user to take advantage of a computer cluster to process large datasets in reduced time.

4. Implemented algorithms

Four algorithms (listed in Table 2) were the object of this work. Of the 8 possible implementations (a Weka multithreaded and a Spark version for each algorithm), 2 were already available and 6 were developed as part of this work.

4.1. RELIEF-F algorithm

The original RELIEF-F algorithm [12] loops through a set of instances R finding for each instance its k nearest neighbors from the same class, called nearest hits H, and the k nearest neighbors from each different class, which are denoted as nearest misses $M(C)$. When all neighbors are found, the weight for each attribute $W[A]$ is updated by subtracting the weighted average distance (computed with the $diff$ function, that returns the Manhattan distance between two instances) of each hit H and adding, for each class C other than R's, the weighted average of the distance to each miss $M(C)$. When computing averages, distances are weighted by the probability P of the class and divided by the total number of instances m.

Table 2: Summary of algorithms in this paper

Algorithm	Multithreaded Weka	Spark implementation
RELIEF-F	New implementation	New implementation
InfoGain	New implementation	Available in Spark packages
CFS	Included in Weka	New implementation
SVM-RFE	New implementation	New implementation

Regarding the multithreaded implementation, the job was divided into as many tasks as threads we wanted to use, then a thread was created for each task. This approach avoided the need for a thread pooler to manage the execution of threads. This process is detailed in Algorithm 1.

The process of finding the nearest neighbors for each instance (by means of the loop described between Lines 2 and 7 of Algorithm 1) is very time consuming since it requires comparing it with all other instances. This search can be executed independently for each instance and therefore it can be performed in parallel with no synchronization issues.

In our Spark implementation the work is split in the same way: each node computes the nearest neighbors to a subset of the examples. Every possible pairing of example indices is generated and stored in a Spark RDD, which is then distributed to the nodes. The whole dataset is sent

Algorithm 1: Pseudo-code for multithreaded RELIEF-F

```
Input: R ←Set of instances having a set of
       attributes A and classified in classes from a
       set C
Output: W ← vector storing the weight of each
        attribute
1  set all weights $\mathbf{W[A]} \leftarrow 0.0$
2  for $i \leftarrow$ 1 to THREADS_AVAILABLE do in parallel
3      $S_i \leftarrow$ disjoint subset of instances
4      foreach $I$ in $S_i$ do
5          find $k$ nearest hits $H$
6          for each class $c \epsilon C \ / \ c \neq class(I)$ do
7              from class $c$ find $k$ nearest misses $M(c)$
           end
       end
   end
8  foreach $a$ in $A$ do
9      foreach $R_i$ in $R$ do
10         $\mathbf{W[a]} \leftarrow \mathbf{W[a]} - \sum_{j=1}^{k} \frac{diff(a,R_i,H)}{m*k} + \sum_{c \neq class(R)} \left[ \frac{P(c)}{1-P(class(R_i))} \sum_{j=1}^{k} \frac{diff(a,R_i,M(c))}{m*k} \right]$
       end
   end
```

to the nodes as a broadcast variable, so that they use it as a lookup table. This approach obtains a considerable speed gain, but effectively limits the size of the dataset to the maximum size a Spark broadcast variable can handle.

4.2. InfoGain algorithm

The InfoGain algorithm assigns the weight (W) of each attribute (A) by contrasting its information gain with respect to the class. To calculate this value, the entropy (H) of each class given the attribute in question is subtracted from the entropy of that class:

$$\begin{aligned} InfoGain(Class, Attribute) = \\ H(Class) - H(Class|Attribute) \end{aligned} \tag{1}$$

Entropy of a variable is defined as $-\sum_i p(i) * log(p(i))$, where i loops through every possible value of the variable. The observed probability of a variable taking a value is represented by $p(i)$, and it is calculated as the ratio of cases where the variable takes that value divided by the total number of appearances of the given variable.

Weka implements this calculation by looping through all the dataset counting the number of appearances of every possible value for each attribute, storing the counts in an array. Then this array is used to compute the information gain of each attribute. This process has linear complexity.

In our proposed multithreaded solution, detailed in Algorithm 2, the counting of every possible value is performed in parallel for a subset of the samples (Line 4). This requires an additional step, described in Line 7, that combines the counts of each thread into a global count. Since this division is performed on the number of instances, it will be more effective when the dataset has numerous instances. For small datasets, the additional accumulative step can take more time than the one that is gained from counting in parallel, but for large datasets the time required to add up the partial counts should be negligible when compared to the counting process.

Lastly, the process of obtaining the information gain values from the counts can also be performed independently for each attribute, therefore it can be computed in parallel (Line 11). The functions *Entropy* and *ConditionalEntropy* shown in Line 14 represent the calculation of $H(Class)$ and $H(Class|Attribute)$ respectively.

Again, the use of a thread pooler was avoided by creating as many tasks as threads are available.

The InfoGain algorithm is already included in the Spark Infotheoretic Feature Selection package [17] that implements several algorithms that share a common structure by the use of a framework [18]. This was the version tested in this paper.

Algorithm 2: Pseudo-code for multithreaded InfoGain

```
Input: R ←Set of instances having a set of attributes A and classified in classes from a set C
Output: W ← vector storing the weight of each attribute
1  set all counts ← 0.0
2  for $t$ ← 1 to THREADS_AVAILABLE do in parallel
3      $S_t$ ← disjoint subset of instances
4      foreach $I$ in $S_t$ do
5          foreach $a$ in $A$ do
6              $counts_{a,value_I(a),class(I)} \leftarrow partial\_counts_{t,a,value_I(a),class(I)} + weight(I)$
           end
       end
   end
7  for $t$ ← 1 to THREADS_AVAILABLE do
8      foreach $a$ in $A$ do
9          foreach $v$ in $values_a$ do
10             foreach $c$ in $C$ do
                   $counts_{a,v,c} \mathrel{+}= partial\_counts_{t,a,v,c}$
               end
           end
       end
   end
11 for $t$ ← 1 to THREADS_AVAILABLE do in parallel
12     $A_t$ ← disjoint subset of attributes
13     foreach $a$ in $A_t$ do
14         $W[a] \leftarrow Entropy(counts_a) - ConditionalEntropy(counts_a)$
       end
   end
```

4.3. CFS algorithm

CFS is a subset evaluator that uses the correlation between attributes to obtain a score for a group of attributes. The computational cost for this algorithm is greatly influenced by the need to obtain the matrix that contains the Pearson product-moment correlation coefficients between every possible pair of attributes. The time complexity of this process grows quadratically with the number of attributes and linearly with the amount of samples, which means that most of time of the CFS algorithm is spent in this process. Once the correlation matrix and the standard deviations of each attribute have been computed, CFS searches the space containing every possible attribute subset looking for one that obtains the highest score in its evaluation method.

The search algorithms used can vary in their complexity, from simple greedy algorithms as the one described in Algorithm 3 that simply adds to the set the best candidate at each step, to more complex backtracking ones like Best-First, listed in Algorithm 4. This search method keeps a list with every candidate set that it encounters ordered by their score in the evaluating function. For each candidate, it explores every possible addition to the set, adding the resulting new set to the candidate list if its score is high enough. This process goes on until the examination of candidate sets renders no new candidates for a given number of iterations (named MAX_STALE in Line 6 of Algorithm 4).

Algorithm 3: Greedy stepwise search used in CFS

```
Input: A ←Set of all possible attributes
Input: S ←Previously selected attributes
Input: $previous\_merit$ ←Merit of S
Output: $S\_out$ ← Selected attributes
1 $best\_merit \leftarrow previous\_merit$
2 $best\_set \leftarrow S$
3 for $a \leftarrow A$ do in parallel
      $set\_merit \leftarrow computeMerit((S, a))$
4     if $set\_merit > best\_merit$ then
5         $best\_merit \leftarrow set\_merit$
6         $best\_set \leftarrow (S, a)$
      end
  end
7 if $best\_set != S$ then
8     return $greedy\_stepwise(A, best\_set, best\_merit)$
  end
  else
9     return $best\_set$
  end
```

The evaluation function used by CFS is described in Algorithm 5. It increases when the attributes are highly correlated with the class and it decreases when any attribute is highly correlated with other attributes that are already in the set.

In the existing Weka implementation, which is included by default in the Weka suite, the computation of the correlation matrix is performed in parallel by several threads, although this only occurs when the user chooses to precompute the correlation matrix. Otherwise the matrix is computed in an on-demand basis, which offers better performance.

Our proposed Spark implementation first performs the

Algorithm 4: Best-first search used in CFS

```
Input: A ←Set of all possible attributes
Output: S ← Selected attributes
1 $stale \leftarrow 0$
2 $candidates \leftarrow OrderedList((empty, 0))$
3 $merit\_cache \leftarrow empty$
4 $best\_set \leftarrow empty$
5 $best\_merit \leftarrow 0$
6 while candidates.hasElements() and stale<MAX_STALE do
7     $S, S\_Merit \leftarrow candidates.popFirst()$
8     $added \leftarrow false$
9     for $a \leftarrow A$ do in parallel
10        if $(S, a)$ in merit_cache then
              $set\_merit \leftarrow merit\_cache.getMerit((S, a))$
          end
          else
              $set\_merit \leftarrow computeMerit((S, a))$
              $merit\_cache.storeMerit((S, a), set\_merit)$
          end
11        if $set\_merit > S\_merit$ then
12            $candidates.push((S, a), set\_merit)$
13            if $set\_merit > best\_merit$ then
14                $added \leftarrow true$
15                $stale \leftarrow 0$
16                $best\_merit \leftarrow set\_merit$
17                $best\_set \leftarrow (S, a)$
              end
          end
      end
18    if not added then
19        $stale \leftarrow stale + 1$
      end
  end
```

Algorithm 5: Subset evaluation in CFS

```
Input: A ← Subset of attributes
Input: C ← Matrix containing the correlation between the ith and jth attributes in $C[i][j]$
Input: SDev ←Array containing the standard deviation for each attribute
Output: M ← Merit of subset
1 $numerator \leftarrow 0$
2 $denominator \leftarrow 0$
3 for $a \leftarrow A$ do
4     $numerator \leftarrow numerator + C[a][class] * SDev[a]$
5     $denominator \leftarrow denominator + SDev[a]^2$
6     for $b \leftarrow A$ where $b < a$ do
7         $denominator \leftarrow denominator + 2 * SDev[a] * 2 * SDev[b] * C[a][b]$
      end
  end
8 $M \leftarrow \frac{numerator}{\sqrt{denominator}}$
```

correlation matrix computation in parallel and then the search process (either BestFirst or GreedyStepwise) is performed, evaluating the different candidate subsets also in parallel.

4.4. SVM-RFE algorithm

To perform feature selection, the SVM Recursive Feature Elimination (SVM-RFE) algorithm makes use of support vector machine classifiers to assign a weight to each attribute. Starting with the whole set of attributes, an SVM is trained to classify binary datasets. The weights assigned to the features by the SVM are then examined and those with the lowest absolute value are removed from the set and added to the ranking in the lowest positions, as shown in Line 16 of Algorithm 6 (the number of elements added at each iteration can be configured with the $STEP$ variable). Then the process is repeated for the remaining attributes until the ranking is complete (Line 12).

In order to work with multiclass datasets, a different ranking is obtained for each class (Line 3) using a one-vs-all approach, that is, assuming that those elements pertaining to a class other than the one being analyzed are negative examples. Then those rankings are combined by looping through them and adding to the final ranking the best of each list, then the second best and so on, in a loop described in Line 5. The process for obtaining the ranking for each class can be done in parallel, and this is the approach taken in our multithreaded Weka implementation. This allows the new version to take much less time when processing multiclass datasets, while not hindering the performance when used with binary datasets.

In the Spark implementation, by contrast, it is the process of training the SVMs that is done in parallel, allowing to save time both on multiclass and binary datasets. This can be done by using the existing SVM with stochastic gradient descent (SGD) implementation in Spark's MLlib library. SGD is an incremental algorithm that is well suited for parallelization. Weka employs Sequential Minimal Optimization (SMO [19]), an analytical method that is generally faster, but much harder to parallelize. This change in the nature of the SVM training algorithm results in a selected set of features that can be different from that obtained with Weka.

4.5. Further considerations

It is worth mentioning that the studied algorithms are very varied regarding their time complexity, which translates in differences in the portion of the total processing time that is devoted to the algorithm. Since one of the goals of this work is to provide a reference guide for users choosing what implementation to use, we have opted for listing the total execution time instead of just the time invested in the algorithm because we think that this will give users a more accurate idea of what to expect from a certain implementation. That being said, there may be some use cases where the algorithm is used in a different context (for instance loading a dataset once and then per-

Algorithm 6: Pseudo-code for multithreaded SVM-RFE

```
Input: S ←Set of instances having a set of
       attributes A and classified in classes from a
       set C
Output: ordered ← Ordered attributes
1  attributeScoresByClass ← empty
2  for $c \leftarrow C$ do in parallel
3      $attributeScoresByClass[c] \leftarrow RankBySVM(c, S)$
   end
4  ordered ← empty
5  for $a \leftarrow A$ do
6      for $c \leftarrow C$ do
7          if not $ordered.contains(attributeScoresByClass[c][a])$ then
8              $ordered.add(attributeScoresByClass[c][a])$
           end
       end
   end
9  return ordered
   Function RankBySVM(c, S)
10     numAttrs ← Number of attributes
11     ranking ← empty stack
12     while $numAttrs > 0$ do
13         weights ← new SVMClassifier(S,c).weights
14         for w in weights do
15             $weights[w] = weights[w]^2$
           end
16         for i in 0 to STEP do
17             worstAttr ← findWorst(weights)
18             S.removeAttr(worstAttr)
19             ranking.add(worstAttr)
           end
       end
20     return ranking
```

forming several iterations of a feature selection algorithm) that takes more advantage from the time gain associated with the parallel implementation. Nonetheless, we choose to compare the execution time of the whole process of performing feature selection on a dataset contained in a file since it will be the most common use case.

5. Experimental results

The goal of this work is to take advantage of multithreaded and distributed processing to speed up feature selection. Hence, the features selected and the weights assigned by the new versions of the algorithms are the same as those obtained with the original versions, excluding any differences that may arise due to rounding or numeric processing (except in the case of SVM-RFE that obtains different results in Spark due to the change of the nature of the underlying SVM). Consequently, these new versions do not modify the classification accuracy, but aim at being able to perform feature selection in a reasonable, shorter time. Therefore, the results listed below focus on the execution time of the feature selection process.

In order to provide a variety of scenarios to test the proposed Weka and Spark implementations, seven high dimensional datasets were chosen (see their characteristics in Table 3). We used the *Higgs* dataset, which consists of 11,000,000 instances with 28 numerical attributes that represent kinematic properties of particles detected in an accelerator [20]. The second dataset used, from here on called *Epsilon*, was artificially created in 2008 for the Pascal Large Scale Learning Challenge [21]. A preprocessed version available on the LibSVM dataset repository [22] was used. This dataset consists of 500,000 instances that have 2,000 numerical features each. Since both datasets mentioned above are binary datasets, one additional dataset with several classes was selected, KDD99 [23]. It contains close to 5 million samples of 41 connection parameters each that are categorized in 23 different classes. Also, SVMs require that datasets have numeric attributes only, so any non-numeric attribute needs to be transformed. Therefore, three multiclass datasets with numeric features were chosen: *Isolet* [24] consists of almost 8000 instances with 617 attributes each, divided in 27 classes. *USPS* [25] is a dataset containing over 7000 examples of elements with 256 attributes, representing handwritten characters, with 10 different labels. Lastly, the *Poker* dataset contains over a million elements with 10 features each, classified in 10 different classes, representing possible hands in the poker card game. An additional larger dataset named *KDDB* consisting of 19 million samples with 30 million attributes was included as an example of very high dimensionality [26].

Table 3: Dataset description

Dataset	Features	Instances	Classes
Higgs	28	11,000,000	2
Epsilon	2,000	500,000	2
KDD99	41	4,898,430	23
Isolet	617	7900	27
USPS	256	7291	10
Poker	10	1,025,010	10
KDDB	29,890,095	19,264,097	2

The experiments were run on up to 8 nodes of a computer cluster. Each node has the specifications described in Table 4. The Weka version used was 3.7.12 running on OpenJDK 1.7.0_55. The OS installed in this machine was Rocks 6.1, based on CentOS 6.x. Spark applications were run using the MapReduce Evaluator (MREv) tool, that unifies the configuration of various distributed computing environments [27].

To measure the performance of the new versions of the algorithms comparatively to the original implementations we used the speed-up measure, defined as the ratio between the original sequential time and the parallel one.

Table 4: Computer cluster description

16 nodes consisting of:	
Processor:	2 × Intel Xeon E5-2660 Sandy Bridge-EP at 2.20Ghz
Cores:	8 per processor (16 per node)
Threads:	2 per core (total of 32 threads per node)
Hard drive:	1 × SSD 480GB SATA3
RAM:	64 GB DDR3 1600 MHz
Network:	InfiniBand FDR & Gigabit Ethernet

5.1. On the preprocessing of the datasets: Parallelization of a discretization algorithm

Some feature selection algorithms, such as InfoGain, require the attributes of the dataset to be discrete. This specification often forces the user to preprocess the dataset in order to obtain a modified version with discrete features. Weka provides an implementation of the Fayyad-Irani Minimum Descriptive Length (MDL) algorithm [28] that fulfills that purpose, although this process can be very time consuming. The goal of this algorithm is to transform real-valued attributes to discrete ones while maintaining as much information as possible. To achieve this, real values need to be assigned to different bins that cover the whole range of values of the attribute. The size, number, and distribution of the bins is decided by the algorithm in a long process that is performed independently for each attribute. This allows us to obtain better performance by using separate threads to compute different attributes, as described in Algorithm 7. A similar parallelization with Spark has not been addressed in this section as it was already available in Spark packages [29]. Table 5 shows the execution times for the sequential implementation compared to the multithreaded one when run on a 16 core machine using the three more general datasets (with and without numerical features, as explained at the beginning of this section).

Although the computing process is independent for each thread, a separate copy of the dataset needs to be allocated for each task, since its first step is to order it by the attribute being examined. The overhead created by copying the dataset can be quite large if the dataset is sizable, but in most cases it is not as large as the gain obtained by computing in parallel. In our experiments all datasets but one obtained a favorable speed-up, independently of their size. The new version performed worse than the sequential one for the Higgs dataset, due to its large size and few attributes, which amounts to costly copies of the dataset and less parallelism.

Algorithm 7: Fayyad-Irani discretization

```
Input: A ← List of attributes
Input: S ← dataset
1 for $a \leftarrow A$ do in parallel
2     $orderedS \leftarrow S.orderBy(a)$
3     $bins[a] \leftarrow computeCutPoints(a)$
      // computeCutPoints uses mutual
         information to obtain the bins in
         which to discretize the values for
         attribute $a$.
  end
4 for $i \leftarrow 1$ to THREADS_AVAILABLE do in
  parallel
5     $S_i \leftarrow$ disjoint subset of instances
6     foreach $I$ in $S_i$ do
7         for $a \leftarrow A$ do
8             $S_i[a] \leftarrow bins[a].transform(S_i[a])$
          end
      end
  end
```

Table 5: Execution times of the discretization algorithm implementations

Dataset	Runtime (s)		Speed-up
	1 core	16 cores	
Higgs	1585	1709	0.93
KDD99	316	196	1.61
Epsilon	1976	881	2.24

5.2. Analysis of the RELIEF-F implementations

The good adaptability of RELIEF-F to a parallel environment (which is often referred to as being "embarrassingly parallel") translates into significant decreases in terms of execution time. Despite this improvement, RELIEF-F's complexity grows quadratically with the number of samples and linearly with the number of features and this still makes it yield long times when the number of instances of the dataset is very high. However, our multithreaded implementation can take advantage of machines with a large number of cores, decreasing computational times.

In order to be able to make a comparison with the sequential version, we have used reduced versions of the largest general datasets (with numerical and non-numerical features) when analyzing the RELIEF-F implementation. For the *Epsilon* and the *KDD*99 datasets the top 10% of the instances were used, amounting to a total of 50,000 and almost 500,000 instances, respectively. The *Higgs* dataset had to be further trimmed, using the top 4%, consisting of 440,000 instances.

We performed tests with different number of threads processing the same datasets in order to illustrate the relation between the execution time and the number of threads employed. The results of these experiments are shown in Figure 1. The node used to run the benchmarks offered 16 cores, each one capable of running two threads using HyperThreading. When 16 threads are used, they are mapped to different cores with exclusive use of resources, obtaining maximum performance. On the contrary, when we request the use of 32 threads, they are placed two on each core, competing for the core resources [30]. This results in a degradation of performance that, in our best case, barely improves on the use of 16 threads. Therefore, all subsequent experiments were made using just the 16 cores.

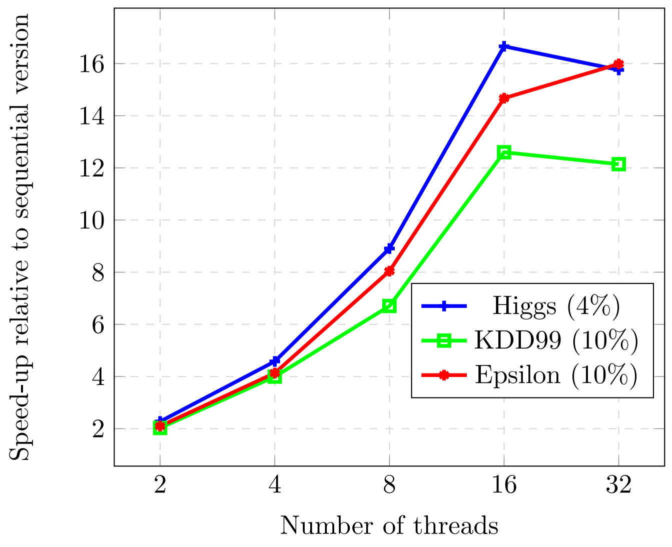

Figure 1: Speed-up vs number of threads for RELIEF-F

Table 6 lists the execution times of sequential and multithreaded Weka implementations. The multithreaded version was executed using the 16 cores available. A significant performance increase exists for all datasets. When the dataset being analyzed is large, the time taken to manage threads becomes irrelevant in comparison to the time gained by making computations in parallel. The multithreaded version of the algorithm was able to process the large datasets between 12.6 and 16.7 times faster than the sequential one. The good adaptability of this algorithm to a parallel paradigm reflects in the superlinearity of the speed-up obtained for the Higgs dataset.

Table 6: Execution times of RELIEF-F Weka implementations

Dataset	Runtime (s)		Speed-up
	1 core	16 cores	
Higgs (4%)	105443	6328	16.7
KDD99 (10%)	154305	10517	14.7
Epsilon (10%)	84149	6678	12.6

Table 7 lists the execution times of the RELIEF-F implementations in Weka and Spark for different amount of cores. The *Epsilon* dataset was chosen for this comparison since its execution time was high on Weka and its size was suitable for the Spark implementation. As discussed in Section 4.1, the Spark implementation of RELIEF-F requires that the entire dataset is broadcast to all nodes. Good scalability is observed when more nodes are added and, even with one node (16 cores), the Spark implementation is more efficient than the Weka one.

Table 7: Execution times of RELIEF-F implementations (speed-up listed for Weka 16 cores vs Spark 128 cores)[2]

Dataset	Runtime (s)						Speed-up
	Weka		Spark				
	# cores						
	1	16	16	32	64	128	
Epsilon (10%)	84149	6678	5382	2840	1076	608	10.98

[2]To assess the advantage of using Spark and a computer cluster

5.3. Analysis of the InfoGain implementations

The Weka implementation of the InfoGain feature selection algorithm requires the attributes to be discrete, so it performs a discretization process when needed before the feature selection is started. This discretization is independent from the InfoGain algorithm so, to eliminate its impact in the execution time and obtain a more accurate comparison of the two versions of the algorithm, all datasets used to test the InfoGain feature selector were discretized beforehand using the same algorithm employed by Weka [28]. This resulted in datasets that, in some cases, had several attributes with constant value. Additionally, to speed up this process for users, a multithreaded implementation of this algorithm is provided, as described in Section 5.1.

Table 8 shows the comparison of the execution times between both versions of the algorithm when run on the different datasets that have been previously discretized. The multithreaded version was run using the 16 cores of the node.

Table 8: Execution times of InfoGain Weka implementations

Dataset	Runtime (s)		Speed-up
	1 core	16 cores	
Higgs	204.3	192.4	1.06
KDD99	145.6	140.1	1.04
Epsilon	458.5	424.7	1.08
KDDB	200.0	192.0	1.08

When put in relation with the whole execution time, the speed improvement is negligible. Nevertheless, a deeper analysis of the implementation reveals that most of the time needed to perform InfoGain feature selection in Weka is spent getting the dataset ready, first reading it from disk and then checking that the attributes are fit for the algorithm. The feature selection process itself takes a short time when compared to the total execution time, so even a dramatic improvement in the time efficiency of the algorithm would lead to modest speed-ups for datasets that take a long time to process. Nevertheless, as discussed in Section 4.5, some use cases may take advantage of the speed-up obtained when just comparing the time devoted to the algorithm which, in the Weka implementation we are presenting, is close to the number of cores employed.

The Spark implementation tested was the one included in the InfoTheoretic Feature Selection Spark package [17]. Results can be seen in Table 9. Instead of the $KDD99$ dataset, $KDDB$ was used to illustrate how this method is capable of handling very high dimensional datasets.

Table 9: Execution times of InfoGain implementations (speed-up listed for Weka 16 cores vs Spark 128 cores)

Dataset	Runtime (s)						Speed-up[1]
	Weka		Spark				
	# cores						
	1	16	16	32	64	128	
Higgs	204	192	578	375	353	173	1.11
Epsilon	458	424	1067	642	448	335	1.27
KDDB	200	192	631	500	384	407	0.47

Although performance increases when adding more cores, for the same number of cores the existing Spark implementation performs much worse than Weka. This results in the need of more nodes to achieve the same times than in Weka, being highly inefficient in terms of resources. For this particular algorithm and datasets it would be more advisable to use Weka on a single machine rather than the existing Spark implementation.

5.4. Analysis of the CFS implementations

The existing multithreaded implementation of the CFS algorithm included in Weka does not offer a significant improvement over the sequential one, being even slower in

vs Weka on a single machine, the speed-up listed is the best Weka result vs the best Spark result using this particular setup.

some cases. This is a result of the parallelization approach used, that requires that the entire correlation matrix is precomputed beforehand, in contrast with the sequential version, that only calculates each value when needed. Since the search method does not try every possible combination of attributes, oftentimes only a small fraction of the correlation matrix needs to be computed. Avoiding to compute these unnecessary values saves significant time that, in some cases, results in smaller computation times than the ones obtained by precomputing the entire correlation matrix with several cores. Our Spark implementation computes the entire correlation matrix every time, but it is still much more time-efficient than the Weka one, as shown in Table 10. The computation time decreases as more nodes are added which and, when combined with the much better performance than the Weka algorithm obtained for the same number of cores, results in high speed-ups.

Table 10: Execution times of CFS implementations (speed-up listed for Weka 16 cores vs Spark 128 cores)

Dataset	Runtime (s)						Speed-up
	Weka		Spark				
	# cores						
	1	16	16	32	64	128	
Higgs	1350	1173	110	98	95	91	12.89
Epsilon	7183	8642	579	438	356	324	26.67

5.5. Analysis of the SVM-RFE implementations

Since the parallelization approach taken for the multithreaded Weka implementation divides the work along classes, multiclass datasets were needed for this experiment. Execution times are shown in Table 11 (please note that times marked with − are executions that take more than three days). The different SVM training algorithm used in Weka and Spark makes a real difference regarding the kind of dataset that can be tackled with each implementation. The Weka version (and thus our multithreaded version), which uses SMO (see Section 4.4), performs really well when there is a large number of attributes and, therefore, the SVM training process has to be repeated a large number of times. In this case the approach used by the Spark version takes much longer, because for every new training process the data needs to be shuffled. This, in some cases, makes its use unfeasible (for instance $Isolet$ and $USPS$). On the contrary, when datasets have fewer attributes (such as $Poker$), the SVM training process is repeated fewer times and SGD can be leveraged to train the model with a large number of examples in a much smaller time than SMO. This clearly differentiates both implementations in terms of the datasets that they handle efficiently. Table 11 shows how SGD is suitable for datasets with a large number of attributes and few instances ($Isolet$ and $USPS$) whereas SMO performs better when there is a large number of instances and fewer attributes (such as $Poker$).

Table 11: Execution times of SVM-RFE implementations

Dataset	Runtime (s)					
	Weka		Spark			
	# cores					
	1	16	16	32	64	128
Isolet	86730	15415	-	-	-	-
USPS	10098	2508	-	-	-	-
Poker	-	-	1229	1536	1220	1447
Poker (20 %)	28621	10280	530	520	472	465

6. Conclusions

This work has explored new implementations of four popular feature selection algorithms. We have proposed new versions that take advantage of multithreaded processing to speed up the computation for their use in Weka and also distributed versions that use Apache Spark, enabling users to tackle bigger datasets in a reasonable time.

For those implementations that already existed (see Table 2), tests were performed to assess their suitability for different kinds of datasets.

The experimental results obtained show a significant improvement in execution time for the RELIEF-F algorithm, achieving even superlinear speed-ups for large real-world datasets on a 16 core node, and scaling well in number of nodes for Spark. A considerable improvement was also obtained for a new distributed CFS implementation in Apache Spark that largely outperforms the existing multithreaded version included in Weka, and scales well when more cores are added. A new multithreaded InfoGain implementation was developed and compared to the existing Spark one, finding that its short execution times make the time gain obtained using a computer cluster less relevant, therefore advising the use of our proposed implementation on a single computer. Lastly, a new SVM-RFE multithreaded implementation enables users to process multiclass datasets up to four times faster than the sequential counterpart included in Weka, and a new Spark version allows the analysis of datasets that because of their dimensions could not be processed by Weka.

As future work, it would be interesting to explore different sampling techniques and their effects on the features selected for a variety of datasets, since this approach may offer a way to use algorithms that are computationally demanding on reduced versions of large datasets. Also pursuing a RELIEF-F implementation with Spark that could handle larger datasets than the ones at reach for the implementation presented in this paper would be advisable.

Acknowledgements

This work has been financed in part by Xunta de Galicia under Research Network R2014/041 and project GRC-2014/035, and by Spanish Ministerio de Economía y Competitividad under projects TIN2012-37954 and TIN-2015-65069-C2-1-R, partially funded by FEDER funds of the European Union. V. Bolón-Canedo acknowledges support of the Xunta de Galicia under postdoctoral Grant code ED481B 2014/164-0. Additionally, the collaboration of Jorge Veiga on setting up and using the MREv tool for Spark execution was essential for this work.

References

[1] IBM Big Data, IBM - Bringing Big Data to the Enterprise, 2015. `http://www-01.ibm.com/software/data/bigdata/`. Accessed: 2015-10-20.

[2] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011.

[3] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[4] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[5] Apache Hadoop Project. `http://hadoop.apache.org/`. Accessed: 2015-10-20.

[6] Apache Spark: Lightning-fast cluster computing. `https://spark.apache.org/`. Accessed: 2015-10-20.

[7] Apache Mahout Project. `http://mahout.apache.org/`. Accessed: 2015-10-20.

[8] Machine Learning Library (MLlib) Guide. `http://spark.apache.org/docs/latest/mllib-guide.html`. Accessed: 2015-10-20.

[9] Verónica Bolón-Canedo, Noelia Sánchez-Maroño, and Amparo Alonso-Betanzos. *Feature Selection for High-Dimensional Data*. Springer, 2015.

[10] Isabelle Guyon. *Feature extraction: foundations and applications*, volume 207. Springer Science & Business Media, 2006.

[11] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[12] Igor Kononenko. Estimating attributes: analysis and extensions of RELIEF. In *Machine Learning: ECML-94*, pages 171–182. Springer, 1994.

[13] Kenji Kira and Larry A Rendell. A practical approach to feature selection. In *Proceedings of the ninth international workshop on Machine learning*, pages 249–256, 1992.

[14] Mark A Hall. *Correlation-based feature selection for machine learning*. PhD thesis, The University of Waikato, 1999.

[15] Isabelle Guyon, Jason Weston, Stephen Barnhill, and Vladimir Vapnik. Gene selection for cancer classification using support vector machines. *Machine learning*, 46(1-3):389–422, 2002.

[16] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

[17] An infotheoretic feature selection framework for apache spark. `http://spark-packages.org/package/sramirez/spark-infotheoretic-feature-selection`. Accessed: 2015-10-20.

[18] Gavin Brown, Adam Pocock, Ming-Jie Zhao, and Mikel Luján. Conditional likelihood maximisation: a unifying framework for information theoretic feature selection. *The Journal of Machine Learning Research*, 13(1):27–66, 2012.

[19] John Platt et al. Fast training of support vector machines using sequential minimal optimization. *Advances in kernel methodssupport vector learning*, 3, 1999.

[20] Higgs dataset at the UCI Machine Learning Repository. `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`. Accessed: 2015-10-20.

[21] Soeren Sonnenburg, Vojtech Franc, Elad Yom-Tov, and Michele Sebag. Pascal large scale learning challenge. In *25th International Conference on Machine Learning (ICML2008) Workshop,*, volume 10, pages 1937–1953, 2008.

[22] LibSVM dataset repository. `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`. Accessed: 2015-10-20.

[23] S Hettich and SD Bay. KDD cup 1999 data. *The UCI KD Archive, Irvine, CA: University of California, Department of Information and Computer Science*, 1999.

[24] M. Lichman. Uci machine learning repository, 2013.

[25] Jonathan J Hull. A database for handwritten text recognition research. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 16(5):550–554, 1994.

[26] J. Stamper, A. Niculescu-Mizil, S. Ritter, G.J. Gordon, and K.R. Koedinger. Bridge to algebra data set from kdd cup 2010 educational data mining challenge, 2010.

[27] Jorge Veiga, Roberto R Expósito, Guillermo L Taboada, and Juan Touriño. Mrev: An automatic mapreduce evaluation tool for big data workloads. *Procedia Computer Science*, 51:80–89, 2015.

[28] Usama M. Fayyad and Keki B. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *IJCAI*, pages 1022–1029, 1993.

[29] Spark implementation of Fayyad's discretizer based on Minimum Description Length Principle (MDLP). `http://spark-packages.org/package/sramirez/spark-MDLP-discretization`. Accessed: 2015-10-20.

[30] Subhash Saini, Johnny Chang, and Haoqiang Jin. Performance Evaluation of the Intel Sandy Bridge Based NASA Pleiades Using Scientific and Engineering Applications. In *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation*, pages 25–51. Springer, 2014.