Prog Artif Intell (2017) 6:211–219
DOI 10.1007/s13748-017-0117-5

REGULAR PAPER

MR-DIS: democratic instance selection for big data by MapReduce

Álvar Arnaiz-González[1] · **Alejandro González-Rogel**[1] · **José-Francisco Díez-Pastor**[1] · **Carlos López-Nozal**[1]

Received: 18 November 2016 / Accepted: 28 January 2017 / Published online: 10 February 2017

Abstract Instance selection is a popular preprocessing task in knowledge discovery and data mining. Its purpose is to reduce the size of data sets maintaining their predictive capabilities. The usual emerging problem at this point is that these methods quite often suffer of high computational complexity, which becomes highly inconvenient for processing huge data sets. In this paper, a parallel implementation for the instance selection algorithm *Democratic Instance Selection* (DIS) is presented. The main advantages of the DIS algorithm turn out to be its computational complexity, linear in the number of instances, as well as its internal structure, intuitively parallelizable. The purpose of this paper is threefold: firstly, the design of the DIS algorithm by following the *MapReduce* model; secondly, its implementation in the popular big data framework *Spark*; and finally, its empirical comparison over large-scale data sets. The results show that the processing time is reduced in a linear manner as the number of Spark executors increases, what makes it suitable for big data applications. In addition, the algorithm is publicly accessible to the scientific community.

Keywords Big data · MapReduce · Apache Spark · Instance selection · Democratic Instance Selection · Classification

1 Introduction

Nowadays, the large quantity of data amassed all over the world has made unsuitable most of the techniques and tools that have traditionally been used in the field of data mining and automatic learning. A new term has been coined to describe this problem: *big data*. There is no clear definition for the meaning of *big data*, although Laney [16], at the start of this century, described it as the opportunities and difficulties that appear as the volume, variety and speed of the data increases. Here, volume refers to massive information that is been generated and collected. Variety refers to the diversity of formats in which we can find the data: structured conventionally, semi-structured or without any structure, such as video or audio. Finally, speed refers to the fact that data should be processed in an swift way, so that its utilization and exploitation are of commercial value [7]. Another commonly accepted definition describes the problems of *big data* as the difficulties that appear when the quantity of data to process exceeds the capacities (memory and/or time) of a given system [20].

According to Wu et al. [25], the most acceptable approach for the treatment of massive data sets is to distribute them into parallel environments and *cloud* computing platforms. This parallelization of tasks is not new at all, and the "divide and conquer" strategy has been used since the dawn of machine learning. Nevertheless, new *frameworks* have recently become popular for facilitating this task and for helping programmers in their work. Among the frameworks used today, *MapReduce* is one of the most popular models of parallel computation, due to its robustness and transparency with regard to resource management [9,21].

Looking at the problem from a different perspective, *data reduction* presents a direct solution to the problem of processing large volumes of information that can also be applied in conjunction with any type of learning algorithm. Of all the different paradigms, instance selection is a commonly used method that consists of selecting a subset of examples. This method is capable of maintaining, or even improving, the predictive capability of the original set [10]. The main problem,

✉ Álvar Arnaiz-González
alvarag@ubu.es

[1] University of Burgos, Avda. Cantabria s/n, 09006 Burgos, Burgos, Spain

however, is that the majority of instance selection methods have a high computational complexity (quadratic order in the number of instances or higher), which means that it is infeasible to use them in data sets of large dimensions [22]. One of the instance selection algorithms that are not subject to this limitation is *Democratic Instance Selection* (DIS) [12].

In this paper, the adaptation of the DIS algorithm to the *MapReduce* model is presented through Apache Spark *framework* [26]. In summary, the objectives of the present paper are as follows:

– To design a parallelized version of the DIS algorithm following the model *MapReduce*.
– To implement the previously parallelized DIS algorithm in Spark.
– To confirm its correct performing in terms of accuracy and compression.
– To evaluate its scalability through its execution on large data sets.

The organization of the present paper is as follows: Sect. 2 presents an introduction to instance selection techniques, explaining in detail the algorithm *Democratic Instance Selection* (DIS), Sect. 3 introduces the *MapReduce* model and its application to the DIS algorithm, Sect. 4 details the experiments conducted for evaluating the performance of the algorithm and, finally, Sect. 5 expounds the conclusions and the principal lines of future work.

2 Instance selection

A fundamental task in knowledge extraction from data sets is their preprocessing. *Data preparation* groups together a large number of techniques that are necessary as a preliminary step in learning processes such as classification, regression or clustering. Among these, we can find many algorithms for data set size reduction. These algorithms are grouped under the term *data reduction* which, in turn, is divided into: discretization, feature extraction, instance generation, feature selection and instance selection [11]. The aim of these techniques is to minimize the size of the initial data set, by reducing either the number of instances or the number of attributes (avoiding the so-called *curse of dimensionality* [15]).

Both instance selection and instance generation work at the level of instances, reducing their number. While instance generation creates new instances and discards existing ones, instance selection (which concerns us in this paper) selects the most representative instances from the original data set. The selected subset[1] should be able to maintain, or even improve, the predictive capability of the original set. One of the benefits of size reduction is the corresponding reduction in classification time (for lazy learning algorithms) and training time (for eager learning algorithms).

Since their appearance [14], multiple instance selection algorithms have been described in the literature.[2] Nevertheless, most of these algorithms suffer from a problem that complicates their application to massive data sets: their computational complexity. The computational complexity of the majority of the above-mentioned methods is, at least, quadratic in the number of instances. Recent algorithms have been developed to overcome this drawback [2,3,6,8,12], among which we will highlight *Democratic Instance Selection* [12], which is analysed in detail in the following section.

2.1 Democratic Instance Selection

The algorithm *Democratic Instance Selection* or DIS [12] is based on the idea of divide and conquer. Putting it briefly, it consists of the execution of a number of rounds r, in each of which the original set is divided into a number of disjoint equally sized partitions (called bins). Then, a classic instance selection algorithm is applied independently over all those partitions. The instances selected to be eliminated by this last instance selection algorithm receive a vote. The process is repeated during the predefined number of rounds r (chosen by the user), after which the instances with a number of votes equal or higher than the calculated threshold are eliminated.

The biggest advantage of DIS is the reduction of the execution time due to its computational complexity: linear in the number of instances. The details of how this calculation is obtained is given in Sect. 2.1.3.

Another benefit is the ease with which the algorithm can be adapted to work in a parallel environment. This is possible due to the execution of each instance selection process on each partition is independent from the others. It is worth noting that the user can select the number and size of the subsets in such a way that it is adjusted to the capabilities of the system where the algorithm will be executed.

Pseudocode 1 presents the original algorithm, which is divided into two equally important parts that are analysed below:

– Partitioning and voting: the input set is divided into disjoint subsets and the desired instance selection algorithm is applied to each one of them. This process is repeated as many times as rounds we have previously defined. Each time an instance is selected to be eliminated, it receives a vote. These votes will be stored for subsequent processing.

[1] The subset selected by the algorithm is indistinctly referred to as filtered or selected set in the present paper.

[2] We recommend the work of S. García et al. [10] for readers interested in this field.

– Selection of instances: a threshold for the number of votes is calculated, in accordance with a *fitness function*, and all instances with a number of votes equal to or over that threshold are removed.

Algorithm 1: Democratic Instance Selection (DIS) algorithm

```
Input: A training set $T = \{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)\}$, subset size $s$,
       number of rounds $r$
Output: Selected set $S \subseteq T$
1 for $k = 1$ to $r$ do
2     Divide instances of $T$ into $n_s$ disjoint subsets $t_i$: $\bigcup_i t_i = T$ of
      size $s$
3     for $j = 1$ to $n_s$ do
4         Apply instance selection to $t_j$
5         Store votes of removed instances from $t_j$
6 Obtain threshold of votes, $v$, to remove an instance
7 $S = T$
8 Remove from $S$ all instances with a number of votes $\geqslant v$
9 return $S$
```

An example of how the DIS algorithm operates is shown in Fig. 1. The problem depicted here contains 30 instances, 10 rounds were performed and, at the end, all those instances with a number of votes below 5 were selected. In other words, all those with a number of votes equal to or greater than 5 were eliminated.

2.1.1 Data set partition

An important stage in the method is the partitioning of the initial set T into smaller subsets t_i (called bins), which comprise the complete data set $\bigcup_i t_i = T$. The size of the bins is selected by the user. It is worth highlighting that the execution time of the complete algorithm depends on the size of the biggest bin (the one with the greatest number of instances). Therefore, it is recommended that each bin has, approximately, the same number of instances.

The simplest partitioning method is by random selection, in which each instance is assigned a bin in a random manner. The problem with this method is that the neighbourhoods are not maintained from one round to another, i.e. instances that are neighbours in one round will likely not be so in the following round. However, this method has been selected because of its easy implementation and speed.

2.1.2 Determining the threshold of votes

A keystone of the selection process is the estimation of the threshold, which will determine the instances that are selected for the filtered set. Experimentation reported by García-Osorio et al. [12] demonstrated that this value depends on the data set under analysis. It is therefore impossible to come up with a fixed value for all possible data sets, and a function that selects the value directly from the input data during execution time is necessary. The most intuitive approach would be to perform a cross-validation on the initial set, although this would be tremendously costly in terms of time.

The chosen method is much less computationally expensive. It estimates the value of the most appropriate number of votes on the basis of the initial set. When computing the threshold, two criteria are taken into account: training error ϵ_t (on a subset of the initial set) and the resulting size m of the possible solution set. As is logical, both values should be minimized. A criterion called fitness or $f(v)$, is defined as the weighted combination of both:

$$f(v) = \alpha\epsilon_t(v) + (1 - \alpha)m(v)$$

where m is the percentage of instances conserved by the algorithm, ϵ_t is the training error and α is a parameter within the interval [0, 1]. This parameter represents the relative importance of each of the two previously mentioned criteria. The user-selected α value means the user has the choice to maximize either accuracy or reduction.

Calculation of error can be a costly process. To minimize this cost, the error is estimated by using a small percentage of the complete training set, which can be, according to the authors [12], 10% for large sets and 0.1% for massive sets.

The process to obtain the threshold is as follows: first, we sum the number of votes received by each instance after r number of rounds have been completed. Then we use those votes to calculate the threshold $v \in [1, r]$ which minimizes the function $f(v)$. Once calculated, all the instances that have a number of votes greater or equal to the threshold v are eliminated.

2.1.3 Complexity analysis

As previously stated, DIS algorithm divides the data set into disjoint sets of a fixed size s, such that the instance selection algorithm used during the voting, whatever its complexity, is applied to a set of fixed size. Let K be the number of operations necessary to complete the selection of instances in a set of size s. For an initial data set of size n, if we perform r rounds, the total time is proportional to $r(n/s)K$, which is lineal as K is a constant value.

In addition to the selection of instances applied in each partition, there are another two processes: the partitioning of the original set and the threshold computation. The partitioning is random, so its complexity is $\mathcal{O}(1)$. With regard to the decision on the number of votes, if all the instances were considered, the cost of this step would be $\mathcal{O}(n^2)$; however, the number of instances that are considered decreases as the

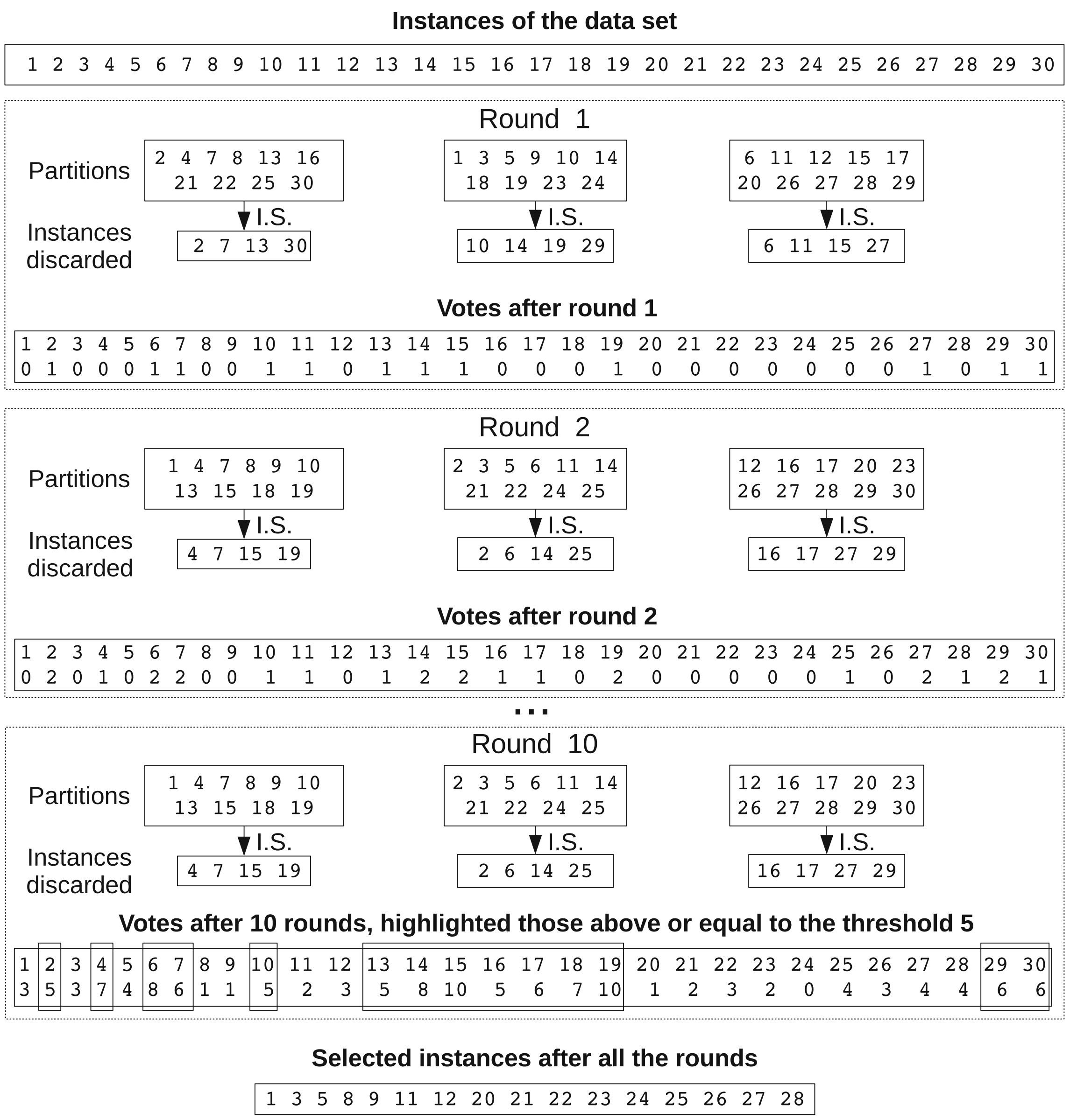


Fig. 1 Example of a DIS algorithm execution on a set of 30 instances. The algorithm executes 10 rounds and the calculated threshold of votes is 5, so that all those instances with a lower number of votes are selected for the filtered set

size increases (e.g. 10% of the training set in large sets and 0.1% in huge sets). Therefore, the complexity of this step and the whole algorithm is maintained linear in the number of instances.

3 MapReduce

MapReduce emerged as a programming model that faces the problem of processing large data sets from the perspective of parallel computing. It is based on the use of pairs ⟨key, value⟩ and on the division of problems into two phases: *Map* and *Reduce*. The *Map* stage operates on the initial pairs ⟨key, value⟩ to generate intermediate pairs ⟨key, value⟩. Subsequently, these pairs are combined according to their key in the *Reduce* phase, yielding the final result [9].

The tendency to work with ever larger amounts of data, this system's high tolerance to failure and its transparency with regard to resource management have made of this model one of the most widely used over the recent years [19].

Apache Hadoop and Apache Spark are two of the most popular *frameworks* at present for massive processing in *big data* environments. They both implement MapReduce and both use a master–slave architecture where a single node (master) is in charge of managing a variable number of worker nodes (slaves). Nevertheless, Spark has a clear advantage over Hadoop given that it is designed to make iterative operations faster through the intensive use of mem-

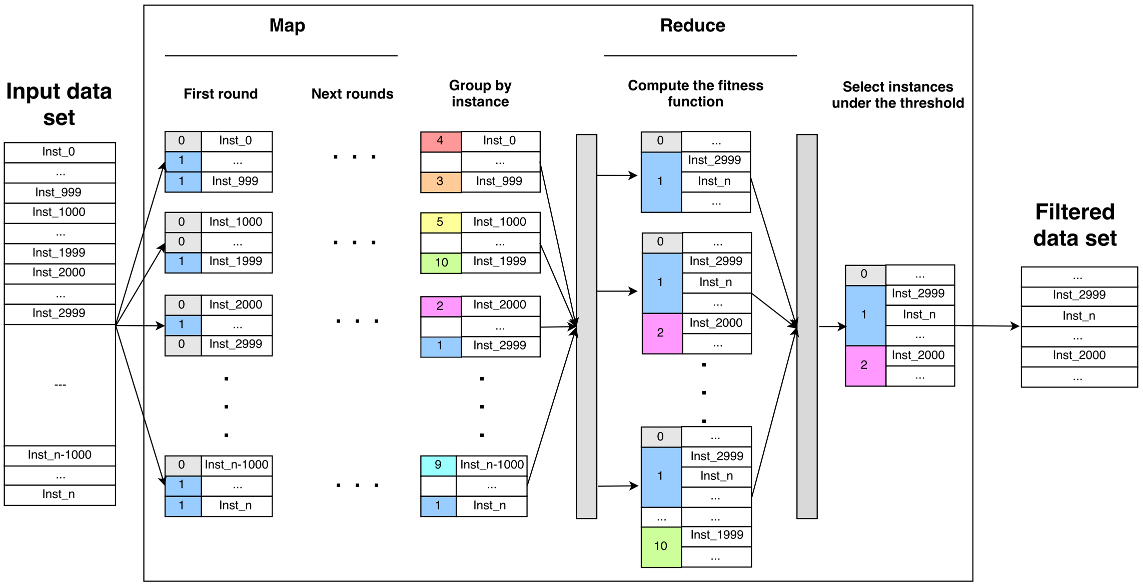


Fig. 2 Example of a DIS algorithm execution following the MapReduce model

ory (instead of using only disc as Hadoop does). This makes Spark more appropriate for data mining and machine learning tasks, where it is able to outperform Hadoop by up to 100 times [26].

Currently, there is a lack of instance selection methods in Spark [23]. Thus, the present paper also seeks to provide the community with a new parallel algorithm and its implementation, for the preprocessing of large volume of information.

3.1 MR-DIS: the MapReduce design for Democratic Instance Selection

The MapReduce parallelization technique explained earlier was used for adapting the DIS algorithm to the *big data* environment.

With regard to the structure of the data, pairs ⟨votes, inst⟩ were used to operate throughout the *Map* and the *Reduce* phases. The term "inst" refers to a concrete instance, while the value "votes" refers to the number of times that the instance that accompanies it has been selected during the voting phase.

The algorithm is described with a dynamic view in Fig. 2, and with two static views in pseudocodes 2 and 3. Figure 2 presents a possible scenario where 10 votes are performed over n instances and a threshold of 3 votes is calculated for the final selection. In both cases, we can differentiate the two characteristic phases of the MapReduce model, which are analysed as follows:

– **Map phase** Completes the rounds of voting, updating the value of the key "votes" of the pairs in each iteration. Each *mapper* receives a bin (a partition of the original set), to which it applies an instance selection algorithm, updating the votes of those instances that the algorithm discards. The input of each algorithm is, therefore, one of the disjoint sets described in line 4 of pseudocode 1. In the current implementation, the distribution of the data is random, but the size of the bins is selected by the user. The user also decides the instance selection algorithm that it is applied to each bin. The number of maps (num_maps) that are executed is obtained with the following formula:

$$\text{num\_maps} = r \times \left\lceil \frac{|T|}{|t_i|} \right\rceil$$

where r is the number of rounds, $|T|$ is the number of instances of the original set and $|t_i|$ is the size of each one of the bins.

Algorithm 2: MR-DIS: Map function

```
  Input: A subset of the original training set $T_j$
  Output: Array of pairs $V_j = \{\langle \text{votes}_1, \mathbf{x}_1\rangle, ..., \langle \text{votes}_n, \mathbf{x}_n\rangle\}$
1 Apply instance selection algorithm to $T_j$
2 foreach instance $\mathbf{x} \in T_j$ do
3     if $\mathbf{x}$ has been selected then
4         Add to $V_j$ the pair $\langle 1, \mathbf{x}\rangle$
5     else
6         Add to $V_j$ the pair $\langle 0, \mathbf{x}\rangle$
7 return $V_j$
```

An intermediate grouping step occurs between the map and the reduce functions.[3] In this process, the ⟨votes, inst⟩

[3] In Spark, the process located between the map and the reduce phases is usually referred to as *shuffle*.

pairs are grouped by instance, summing the votes that each instance has received during the voting phase. The instances are then distributed to the *reducers* in accordance with the votes received, creating as many groups as rounds r that have been completed.

- **Reduce phase** Includes the calculation of the threshold of votes. Each *reducer* completes the calculation of the fitness function for a certain number of votes $k \in [1, r]$ (where r is the number of rounds). The input data set of each *reducer* is determined by the number of votes of each instance. In other words, the times that the instance has been discarded by the individual instance selection algorithms in the map phase. An estimation of the accuracy of the subset and the compression rate is necessary to calculate the fitness function. Compression is directly calculated by taking into account the number of instances of the original set and of the selected subset. It is necessary to apply a classifier to calculate the estimation of accuracy, which could imply a problem with large data sets. A parallel implementation of the algorithm k-NN [19] over a percentage of the initial set of instances was used to avoid the possible bottleneck in this calculation. The number of *reducers* will be determined by the number of rounds r.

Algorithm 3: MR-DIS: Reduce function

```
Input: Array of pairs $V_k = \{\langle \text{votes}_1, \mathbf{x}_1 \rangle, ..., \langle \text{votes}_n, \mathbf{x}_n \rangle\}$, test set
       $X \subset T$, round key $k$, $\alpha$ value
Output: Fitness value $f_k$
1 Train $k$NNIS with $V_k$
2 $\varepsilon \leftarrow$ Error($k$NNIS, $X$)
3 $m \leftarrow \frac{|V_k|}{|T|}$
4 $f_k = \alpha\varepsilon + (1 - \alpha)m$
5 return $f_k$
```

Once the values of the fitness function have been calculated for each of the values $k \in [1, r]$, the lowest function value is selected. The last step of the algorithm is the generation of the resulting data set, which is constructed by selecting all those instances with a number of votes below the threshold (value k to which the fitness is minimum).

4 Experimental set-up

The objective of this section is the study of the scalability of the DIS algorithm. As a side note, the results for both accuracy (% accuracy) and compression (% of instances of the original data set selected in the filtered set) were similar to those obtained through sequential implementation, in order to validate that the parallel version performs in a similar way to its sequential counterpart. The parallel k-nearest neighbour was used for the calculation of accuracy (in its available version for Spark [19]).

The implementation of DIS was done with Scala on Spark 1.6.1 and is publicly accessible at BitBucket.[4]

Our initial hypothesis was as follows:

Hypothesis 1 The increase in the number of processors in the execution of the MR-DIS algorithm reduces execution time in large data sets.

We have developed two case studies, aimed at assessing this hypothesis:

- Small data set: the algorithm was executed on various configurations of *executors*,[5] from 4 up to 256, on a medium-sized data set. Accuracy, compression and filtering time were all measured.
- Big data sets: the filtering times of the MR-DIS algorithm with configurations of *executors* from 64 to 256 were measured on two data sets of up to one million instances.

The structure of the present section is as follows: Sect. 4.1 defines the framework of experimentation; Sect. 4.2 details the instrumentation; and, finally, the case studies are presented in Sects. 4.3 and 4.4.

4.1 Experimental framework

The experimental study is centred on verifying the scalability of the implemented algorithm. To do so, the algorithm was executed with a varying number of executors. It is expected that parallelization will not affect the general operation of the algorithm The following measures were extracted from a ten*fold* cross-validation without repetition:

- Mean accuracy: percentage accuracy calculated on a 1-NN classifier trained with the subset selected by the MR-DIS algorithm.
- Average compression: percentage of instances of the original set present in the selected subset.
- Filtering time: Time spent by DIS algorithm on the filtering task.
- Speedup: measure of the efficiency of the parallel version with respect to the sequential version. It is calculated by dividing the execution time of the sequential version by

[4] Author: Alejandro González-Rogel, https://bitbucket.org/agr00095/tfg-alg.-seleccion-instancias-spark.

[5] In the Spark framework, each *worker* node has one or more executors, each one of which completes a task. A processor was assigned to each *executor* in the experimental work.

the time of the parallel version. The speedup of a single processor is one, such that it is easy to guess that the maximum theoretical speedup that may be obtained is the same as the number of processors in use. Nevertheless, the speedup is normally lower than the number of processors as the system tends to become over-saturated [13]. In other words, efficiency falls as the numbers of processors increase, which is known as Amdahl's law [1].

$$\text{Speed\_up} = \frac{\text{base\_line}}{\text{parallel\_time}}$$

In all the case studies, the parameters used in the execution of the MR-DIS algorithm were as follows:

– Algorithm used: condensed nearest neighbour (CNN) [14].
– Number of rounds: 10.
– Number of instances by partition: 1000.
– Percentage of instances for the error calculation: 1%.[6]
– Alpha value: 0.75.

4.2 Instrumentation

The experimentation was performed using the Apache Spark 1.6.1 distribution on the Google DataProc service.[7] Several clusters were deployed, each one of them with twice the number of processing cores than the earlier one, from 4 to 256. The cluster computation nodes were formed of Intel Xeon E5 with 16 cores and 60 GB of RAM. Additionally, each cluster has a master node in charge of the administration and management of the *executors*. The master node was an Intel Xeon E5 with 2 cores and 7.5 GB of RAM.

Each *executor* was assigned to a core. The collections of data used by Spark (RDDs) were stored under the persistence level `MEMORY_ONLY` of the *framework*. This storage option implies that the data is stored without serialization in memory and, in case of there being no space, some partitions of these structures can be removed and recalculated whenever necessary.

4.3 Small experiment

One of the objectives of the present paper is to test the scalability and the correct distribution of the work between the nodes, in such a way that the processing time is reduced in a linear manner as the number of executors increases. This was tested under various configurations ranging from 4 to 256 executors. A subset of 10% of the Susy data set [18] was selected to be able to run the job on such a small number of executors within a reasonable time. Susy comprises 5,000,000 instances produced by a physics particle simulation. Each instance is formed of 18 attributes and it is a binary classification problem.

Table 1 Results of the experimentation with 10% of the Susy data set

# Executors	Comp. (%)	Acc. (%)	Filt. time (s)	Speedup
4	21.79	66.58	6 877.92	1.81
8	21.80	66.61	3 774.22	3.30
16	21.81	66.56	1 641.45	7.58
32	21.81	66.52	995.02	12.50
64	21.80	66.52	640.44	19.42
128	21.80	66.53	342.74	36.29
256	21.78	66.61	276.70	44.95

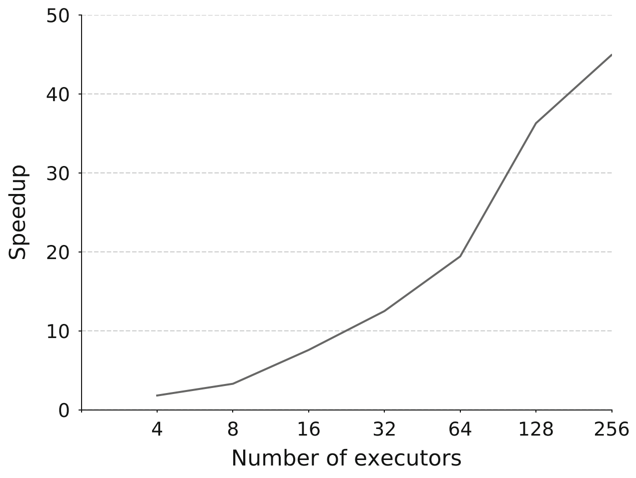


Fig. 3 Evolution of speedup with 10% of Susy as the numbers of executors increase. The *scale of the horizontal axis* is not linear and its number is duplicated at each step

Table 1 shows the results of those executions. The speedup was calculated by using the execution time of MR-DIS with a single executor as the lineal base. This arrangement means that the linear base is much faster than a possible sequential external implementation without Spark, as access to the data sets is done in a more efficient manner (Spark access is not sequential line by line but it is able to read and to load blocks of instances in memory) [19].

Additionally, speedup is shown in graph form in Fig. 3. It may be seen that, as the number of executors increases, the speedup slope increases up until 128. After this point, however, speedup is lower than might be expected. This reduction in the speed could be caused by the fact that a 10% of the Susy data set is not large enough to be able to give work to all of the available nodes, and part of the parallelization power is not exploited.

[6] In [12] the percentage of instances used for error estimation in massive data sets was 0.1%, but the use of a parallel implementation of 1-NN permits an increase in this percentage, improving the precision of the estimation.

[7] Google Cloud Platform: https://cloud.google.com/dataproc/.

Table 2 Average filtered and classification time for the data set types Poker and Cover Type

# Executors	Filtering time (s)		Classification time (s)	
	Cov.Type	Poker	Cov.Type	Poker
64	1 689.39	1 243.38	125.98	330.21
128	717.85	684.33	42.99	160.48
256	361.93	415.75	27.61	78.61

It is worth mentioning that the slight variations in reduction and accuracy that may be observed in Table 1, are due to the indeterminism generated by parallel execution. In other words, there is no exact order in which the operations are executed and this indeterminate process usually means that the partitioning or selection of instances from the initial set is different in each iteration

4.4 Big data sets

In the previous subsection, the experiment carried out on the subset of 10% of the Susy data set has demonstrated that the implementation maintains accuracy and compression regardless of the numbers of executors that are used. In this subsection, the execution was done with clusters of 64, 128 and 256 executors on two data sets of up to one million instances. Two popular data sets from the UCI [18] repository were used: Cover type and Poker Hand. Cover type is formed of 581,012 instances, which describe 30×30 square metre regions in terms of cartographic variables. The value to predict is forestry coverage. There are 7 types of forestry cover; in other words, it is a problem with 7 classes. The attributes collected for each sample are summarized in 54 nominal and numeric attributes.

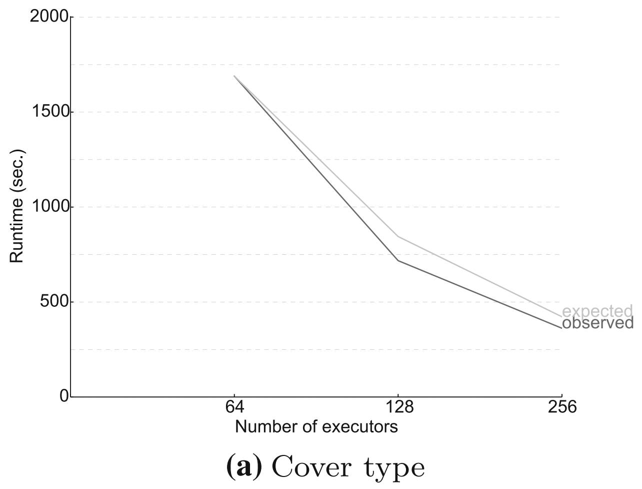
Poker Hand is composed of 1,025,010 instances, and each of them represents an example of a hand consisting of 5 of the 52 possible cards from the deck. Each card is represented by two attributes (suit and rank), so the set has ten attributes: five numeric (from 1 to 13) and five categorical with four possible values (Hearts, Spades, Diamonds, Clubs). It has ten classes that indicate the possible game: one pair, two pairs, flush and so on.

Table 2 shows the evolution over filtering (time needed to apply MR-DIS) and classification (using MR-kNN over the output of MR-DIS) time, in seconds, for Cover type and Poker data sets. In both cases, it may be appreciated that the time is shortened in a linear way as the number of executors is doubled. In addition, the algorithm execution time is shown in a graph in Fig. 4. A line (expected) has been drawn to facilitate comparison, which uses the time with 64 executors as a reference and is halved as the number of executors increases.

5 Conclusions and further work

In the present paper, a Spark implementation of the instance selection algorithm, known as Democratic Instance Selection (DIS), has been presented. The implementation has been done according to the MapReduce model. This algorithm was selected for two reasons: on the one hand its complexity, lineal in the number of instances, and on the other, its intuitively parallelizable design.

During the implementation process, Spark's ability to distribute work between different machines has become clear.



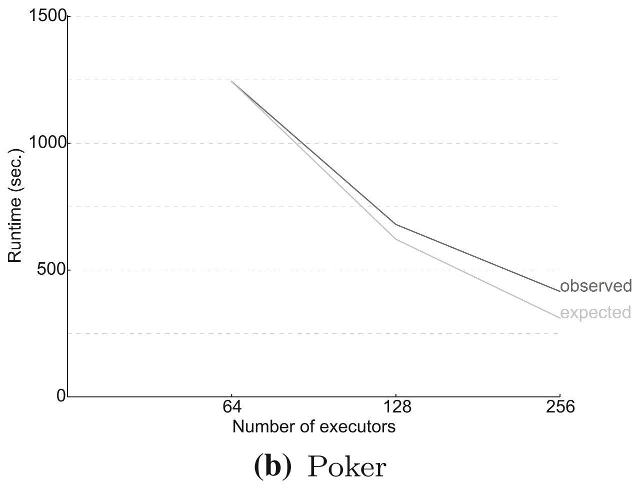


Fig. 4 Evolution over time of filtering in the data sets Cover type and Poker as the number of executors increases. The filtering time (observed) and the expected linear progression are shown. In order to facilitate the comparison, a *grey line* (expected) is drawn that shows the time with 64 executors as its origin and is halved as the number of executors is doubled. The *scale of the horizontal axis* is not linear, and at each step, its number is doubled. **a** Cover type, **b** Poker

The work of the developer is to implement the algorithm following the MapReduce model, but Spark framework is responsible for assigning the work load of each node in a transparent way.

The experimentation has proved the scalability of the MR-DIS implementation and its ability to deal with large dimensional data sets. Since in its current version it already uses a parallel implementation of *k*NN, the MR-DIS is scalable on massive data sets without having any bottleneck that could weaken its performance. In addition, the code is publicly available at the Bitbucket repository.

Thus far, this study has only implemented and tested one instance selection algorithm (CNN) in the internal process of DIS. One immediate improvement, in which we are working, is the addition of new algorithms such as ICF [5], DROP [24] or LSBo [17] that could provide a faster, more efficient and novel approach.

Moreover, the random partitioning method is not the most intelligent. In the original paper where DIS was presented, the authors proposed another partitioning alternative based on *Grand Tour* theory [4]. Its execution under the *MapReduce* model is a line of investigation that is proposed for the future.

Acknowledgements This work was funded by the Ministry of Economy and Competitiveness, Project TIN 2015-67534-P.

References

1. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18–20, 1967, Spring Joint Computer Conference, AFIPS '67 (Spring), pp. 483–485. ACM, New York (1967). doi:10.1145/1465482.1465560
2. Angiulli, F., Folino, G.: Distributed nearest neighbor-based condensation of very large data sets. IEEE Trans. Knowl. Data Eng. **19**(12), 1593–1606 (2007). doi:10.1109/TKDE.2007.190665
3. Arnaiz-González, Á., Díez-Pastor, J.F., Rodríguez, J.J., García-Osorio, C.I.: Instance selection of linear complexity for big data. Knowl. Based Syst. **107**, 83–95 (2016). doi:10.1016/j.knosys.2016.05.056
4. Asimov, D.: The grand tour: a tool for viewing multidimensional data. SIAM J. Sci. Stat. Comput. **6**(1), 128–143 (1985)
5. Brighton, H., Mellish, C.: Advances in instance selection for instance-based learning algorithms. Data Min. Knowl. Discov. **6**(2), 153–172 (2002). doi:10.1023/A:1014043630878
6. Cano, J.R., Herrera, F., Lozano, M.: Stratification for scaling up evolutionary prototype selection. Pattern Recognit. Lett. **26**(7), 953–963 (2005). doi:10.1016/j.patrec.2004.09.043
7. Chen, M., Mao, S., Liu, Y.: Big data: a survey. Mob. Netw. Appl. **19**(2), 171–209 (2014). doi:10.1007/s11036-013-0489-0
8. de Haro-García, A., García-Pedrajas, N.: A divide-and-conquer recursive approach for scaling up instance selection algorithms. Data Min. Knowl. Discov. **18**(3), 392–418 (2009). doi:10.1007/s10618-008-0121-2
9. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008). doi:10.1145/1327452.1327492
10. Garcia, S., Derrac, J., Cano, J., Herrera, F.: Prototype selection for nearest neighbor classification: taxonomy and empirical study. IEEE Trans. Pattern Anal. Mach. Intell. **34**(3), 417–435 (2012). doi:10.1109/TPAMI.2011.142
11. García, S., Luengo, J., Herrera, F.: Data Preprocessing in Data Mining. Springer, Berlin (2014)
12. García-Osorio, C., de Haro-García, A., García-Pedrajas, N.: Democratic instance selection: a linear complexity instance selection algorithm based on classifier ensemble concepts. Artif. Intell. **174**(56), 410–441 (2010). doi:10.1016/j.artint.2010.01.001
13. Grama, A.Y., Gupta, A., Kumar, V.: Isoefficiency: measuring the scalability of parallel algorithms and architectures. IEEE Parallel Distrib. Technol. **1**(3), 12–21 (1993). doi:10.1109/88.242438
14. Hart, P.: The condensed nearest neighbor rule (corresp.). IEEE Trans. Inf. Theory **14**(3), 515–516 (1968)
15. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, STOC '98, pp. 604–613. ACM, New York (1998). doi:10.1145/276698.276876
16. Laney, D.: 3-d data management: controlling data volume, velocity and variety, Technical Report META Group Research Note (2001)
17. Leyva, E., González, A., Pérez, R.: Three new instance selection methods based on local sets: a comparative study with several approaches from a bi-objective perspective. Pattern Recognit. **48**(4), 1523–1537 (2015). doi:10.1016/j.patcog.2014.10.001
18. Lichman, M.: UCI machine learning repository (2013). http://archive.ics.uci.edu/ml
19. Maillo, J., Ramírez, S., Triguero, I., Herrera, F.: kNN-IS: An iterative spark-based design of the k-nearest neighbors classifier for big data. Knowledge-Based Systems (2016). doi:10.1016/j.knosys.2016.06.012
20. Minelli, M., Chambers, M., Dhiraj, A.: Big Data, Big Analytics: Emerging Business Intelligence and Analytic Trends for Today's Businesses. Wiley, London (2012). doi:10.1002/9781118562260.fmatter
21. Ramírez-Gallego, S., García, S., Mouriño Talín, H., Martínez-Rego, D., Bolón-Canedo, V., Alonso-Betanzos, A., Benítez, J.M., Herrera, F.: Data discretization: taxonomy and big data challenge. Wiley Interdiscip. Rev. Data Min. Knowl. Discov. **6**(1), 5–21 (2016). doi:10.1002/widm.1173
22. Triguero, I., Peralta, D., Bacardit, J., García, S., Herrera, F.: Mrpr: a mapreduce solution for prototype reduction in big data classification. Neurocomputing **150 Part A**, 331–345 (2015). doi:10.1016/j.neucom.2014.04.078
23. Tsai, C.F., Lin, W.C., Ke, S.W.: Big data mining with parallel computing: a comparison of distributed and mapreduce methodologies. J. Syst. Softw. **122**, 83–92 (2016). doi:10.1016/j.jss.2016.09.007
24. Wilson, D.R., Martinez, T.R.: Instance pruning techniques. In: Machine Learning: Proceedings of the Fourteenth International Conference (ICML97), pp. 404–411. Morgan Kaufmann (1997)
25. Wu, X., Zhu, X., Wu, G.Q., Ding, W.: Data mining with big data. IEEE Trans. Knowl. Data Eng. **26**(1), 97–107 (2014). doi:10.1109/TKDE.2013.109
26. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. HotCloud **10**, 10–10 (2010)