A First Attempt on Global Evolutionary Undersampling for Imbalanced Big Data

I. Triguero, M. Galar, H. Bustince, F. Herrera

Abstract— **The design of efficient big data learning models has become a common need in a great number of applications. The massive amounts of available data may hinder the use of traditional data mining techniques, especially when evolutionary algorithms are involved as a key step. Existing solutions typically follow a divide-and-conquer approach in which the data is split into several chunks that are addressed individually. Next, the partial knowledge acquired from every slice of data is aggregated in multiple ways to solve the entire problem. However, these approaches are missing a global view of the data as a whole, which may result in less accurate models.**

In this work we carry out a first attempt on the design of a global evolutionary undersampling model for imbalanced classification problems. These are characterised by having a highly skewed distribution of classes in which evolutionary models are being used to balance the dataset by selecting only the most relevant data. Using Apache Spark as big data technology, we have introduced a number of variations to the well-known CHC algorithm to work with very large chromosomes and reduce the costs associated to the fitness evaluation. We discuss some preliminary results, showing the great potential of this new kind of evolutionary big data model.

I. Introduction

Learning from big datasets is a great challenge for most machine learning techniques. Although they are supposed to work better when there is an abundance of data to leverage their outcome, in practice, they cannot be really applied due to memory and time limitations [1]. New parallelisation technologies, however, provide us powerful tools to handle large amounts in the form of distributed datasets [2]. Thus, the problem now consists of figuring out the most suitable way of using such technology to come up with effective learning algorithms.

Hadoop [3] and the MapReduce paradigm [4] served as the first alternatives to deal with data-intensive kind of applications. The key point lied in the use of a distributed file system that allowed us to parallelise multiple tasks across a cluster of computing nodes in a transparent and fault-tolerant manner [5]. Soon enough, the machine learning community found multiple limitations [6] to efficiently deploy algorithms that share data across multiple stages (e.g. iterative algorithms). New platforms such as Spark [2] or Flink [7] have been built upon the MapReduce paradigm to provide us with a new kind of high throughput in-memory distributed datasets that easily allow us to repeatedly carry out operations on the data.

Multiple MapReduce-like strategies have been developed to adapt traditional machine learning and data mining techniques to the new big data scenario. Most of these methods are approximations of the original algorithms, and just a few of them are exact replicas of the sequential version. Approximate models typically divide the data into smaller subsets in which the original algorithm is applied. Then, the different outcomes from each part are somehow combined [8]. Global or sometimes exact approaches aim to replicate the behaviour of the sequential version by letting it see the data as a whole (and not as a combination of smaller parts). As an example, we can find the Decision Trees implemented in Apache Spark [2] or the big data version of the k-nearest neighbours proposed in [9]. The great advantage of this last approach is that they may become more robust and precise, but they tend to be slower.

Even when there are lots amounts of data, we may also run into the situation where there is scarcity of a particular class of samples. Focusing on two-class problems, this issue is known as the class imbalance problem [10], in which positive data samples (usually the class of interest) are highly outnumbered by negative ones [11]. This issue brings along a series of difficulties such as overlapping, small sample size, or small disjuncts [12]. Several approaches have been designed to tackle this problem, which can be divided into three main groups: data sampling, algorithmic modifications and cost-sensitive solutions. These models have also been successfully combined with ensemble learning algorithms [13].

Evolutionary undersampling (EUS) [14] belongs to the data sampling family, where the main objective is to balance the distribution of classes of the original dataset by removing examples of the negative class. This removal is carefully guided by a genetic-based algorithm that aims to increase the performance on the two classes of the problem. However, dealing with a large number of negative examples would lead to a large chromosome size, resulting in a huge search space that limits the straightforward application of EUS on big data. In previous works [15], [16], we devised approximate approaches, based on Hadoop and Spark technologies, that split the original problem into small pieces in which EUS could be concurrently applied. Despite their performance, these models lack of a global view of the entire dataset.

The main goal of this work is to investigate whether a global EUS is feasible with the current technology, in terms

This work was supported by the Research Projects TIN2014-57251-P, P11-TIC-7765 and TIN2016-77356-P

I. Triguero is with the School of Computer Science, University of Nottingham, United Kingdom. E-mail: isaac.triguero@nottingham.ac.uk

F. Herrera is with the Department of Computer Science and Artificial Intelligence of the University of Granada, CITIC-UGR, Granada, Spain, 18071. E-mail: herrera@decsai.ugr.es

M. Galar and H. Bustince are with the Department of Automatics and Computation, Universidad Pública de Navarra, Campus Arrosadía s/n, 31006 Pamplona, Spain. E-mails: {mikel.galar, bustince}@unavarra.es

978-1-5090-4601-0/17/$31.00

of runtime and in comparison to approximate models. As new technologies, such as Spark, allow us to take multiple iterations over the same data without a heavy penalty, we can now devise a parallel EUS that basically distributes time consuming and high memory demanding operations across a number of worker processes, while the main procedure would be running in the driver process. As an evolutionary algorithm, we focus on the widely-used CHC evolutionary algorithm [17], which is modified to create a more compact representation of the chromosomes, and make use of distributed datasets when evaluating the current population.

The paper is structured as follows. Section II provides background information about evolutionary undersampling for imbalanced big data classification. Section III discusses the decisions made to take the CHC model to the big data context with Apache Spark. Section IV analyses the empirical results. Finally, Section V summarises the conclusions.

II. Background

This section describes the big data technologies used in this paper (Section II-A) as well as the current state-of-the-art on imbalanced big data classification (Section II-B).

A. Big Data Technologies

The MapReduce paradigm [4] was designed by Google in 2003 as a scalable data processing tool. Although it was primarily designed as a part of the most powerful search engine on the Internet, its usefulness produced a rapid development becoming one of the most commonly used approach for data intensive applications.

The most popular open-source implementation of MapReduce is Apache Hadoop [18]. Its usage is widely extended due to its performance, easiness of installation, open source nature and the included distributed file system (Hadoop Distributed File System, HDFS). Nevertheless, there are several tasks for which Hadoop MapReduce is not the most appropriate solution due to the additional costs required for reusing data. This is the case of interactive queries and online or iterative computing.

Apache Spark was developed aiming at solving the drawbacks of Hadoop when dealing with these types of tasks. Spark is only a data processing engine that is usually used in top of Hadoop ecosystem. Hence, it commonly relies in HDFS for the storage part. In order to make the data processing faster on distributed environments, this framework considers a set of in-memory primitives, which make the reuse of data faster. The main abstraction of Spark are named as Resilient Distributed Datasets (RDDs). With this data structure parallel computations can be easily implemented in a transparent way. RDDs can persist in memory, making it easy to efficiently reuse results. Moreover, the lazy evaluation of Spark allows the engine to optimise consecutive data transformations without requiring any action from the user. Among other properties, the partitioning of RDDs can also be managed to optimise data placement. Very recently, Spark is moving towards even more efficient APIs such as DataFrame and Datasets offering even greater optimization capabilities due to the newly introduced Catalyst optimiser and Tungsten memory management.

B. Imbalanced classification in the Big Data context

In a binary classification scenario a dataset is said to be imbalanced whenever the number of instances of one class outnumbers that of the other. In this situation, performance measures like the accuracy rate (percentage of correctly classified examples) are no longer valid to measure the quality of the models obtained, since the performance over both classes is not equally weighted. Two commonly used alternatives are the Area Under the ROC Curve (AUC) and the g-mean.

The AUC (Area Under the ROC-Curve) [19] provides a scalar value measuring how well a classifier trades off its true positive (TP_{rate}) and false positive rates (FP_{rate}). A popular approximation [10] of this measure is given by

$$AUC = \frac{1 + \mathrm{TP}_{rate} - \mathrm{FP}_{rate}}{2}. \quad (1)$$

Similarly, the g-mean is the acronym for the geometric mean. In this case, the balance between the true positive rates and true negative rates (TN_{rate}) of the classifier is measured, that is, how well the classifier is able to recognize both classes at the same time:

$$\text{g-mean} = \sqrt{\mathrm{TP}_{rate} \cdot \mathrm{TN}_{rate}} \quad (2)$$

These two measures have been extensively and interchangeably used in various experimental studies of imbalanced classification [10], [14].

Any classification problem can be affected by the presence of class imbalance, and big data problems are not an exception. Even though the quantity of data is much bigger, the imbalance ratio (the number of majority class examples divided by the number of negative class examples) can still be to high so as to extract meaningful models. One main drawback of distributing large imbalanced datasets across different nodes is that the sample size of the minority class in each node will become lower. As a consequence, when a local model is learned using only a subset of the training set, the presence of too little minority class examples can end hindering the classifier learning phase as it is one of the main sources of problems in imbalanced domains [10].

EUS is an interesting alternative to deal with big data imbalanced problems as it reduces the dataset size, on the contrary to oversampling methods that generate even more data [20]. Hence, the corresponding model can be built faster. Another way of reducing the dataset size is by means of random undersampling (RUS). However, its main disadvantage is that it could discard important data from the majority class due to the random nature behind its functioning procedure, whereas EUS guides the balancing of the dataset to preserve or even improve the final performance.

Several data level algorithms were tested in [20] to deal with imbalanced big data classification problems (random over/undersampling and SMOTE). Afterwards, a Random Forest classifier [21] was trained. A different approach was

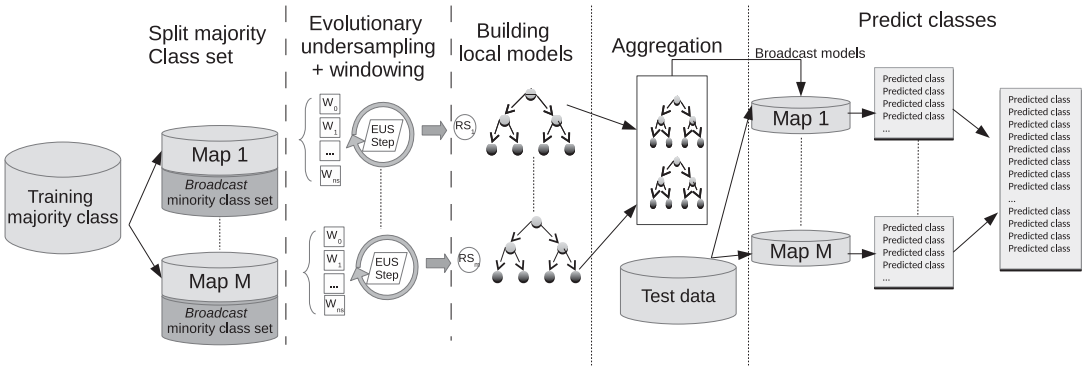

Fig. 1: EUS local for extremely imbalanced datasets [16]

taken in [22] where a fuzzy rule-based classification system was developed to address the class imbalance problem in the big data context. In order to do so, the authors proposed a cost-sensitive approach developed over the MapReduce adaptation of the fuzzy classifier.

With respect to EUS in big data applications, a preliminary work was presented in [15]. The authors proposed a two-level parallelisation model where MapReduce was used to divide the problem into smaller subproblems over which EUS was applied and a windowing scheme was used to reduce the evaluation of each chromosome in each node. However, the small-sample size problem was not addressed in this first approach due to the limitations of Hadoop framework. Nevertheless, this was the main focus of their subsequent work [16], where the authors took advantage of the primitives provided by Spark to properly deal with the small-sample size problem. Spark allows one to broadcast a set of data to all the nodes. This useful property was used to broadcast all the minority class examples to all the nodes so that EUS and the corresponding decision tree could make use of the whole minority class information. Figure 1 depicts this local EUS model. In this work, our aim is to go one step further using all the potential offered by Spark to develop a first attempt on a global EUS model. This way, we will be able not only to get rid of the small-sample size problem, but also to obtain a reduced set which is selected considering the dataset as a whole, which has not been developed before.

III. A GLOBAL EVOLUTIONARY UNDERSAMPLING FOR IMBALANCED BIG DATA WITH APACHE SPARK

In this section we describe the proposed global EUS for imbalanced big datasets based on Apache Spark. We discuss the necessary changes made to the original EUS proposal to extend it to the big data context.

EUS [14] was devised as a new kind of evolutionary instance selection algorithm [23] that accounts for the class imbalance problem. The focus of EUS is to balance the dataset in such a way that the performance is maximised in both classes of the problem.

Following the general procedure of an evolutionary algorithm, it starts off with a population of NP candidate solutions. In the original EUS, a binary chromosome is used to encode every possible solution. In this chromosome, each bit represents the presence (1) or absence (0) of an instance in the training set. To reduce the search space, only majority class instances are considered for removal, including always all the minority class instances in the final dataset.

Having a set of M majority class instances, the first issue we encounter when dealing with big datasets (i.e., M is very big) is that this chromosome will be extremely big as it is representing every single majority class instance. To alleviate this situation, we change the codification used in EUS for a sparse chromosome that only contains the indexes of those majority instances that are being selected. This is a very tailored modification that works well for EUS because in the end its main goal is to balance both classes. Therefore, we assume here that chromosomes are going to select a very few number of majority instances (similar to the number of minority class examples). Otherwise, this codification would probably take even more space than the binary representation. Figure 2 illustrates a comparison between the standard binary representation and the proposed sparse representation.

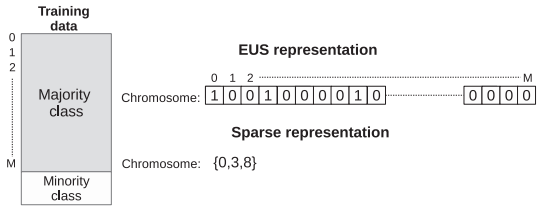

Fig. 2: Differences in representation for EUS. In this example, the chromosome is representing the selection of three majority class instances with indexes 0, 3 and 8, respectively.

The main implication of this decision is that we will want to keep chromosomes representing a reduced number of majority class instances from the beginning. This will cause a few variations on the following steps of the evolutionary process.

The initialisation procedure will be the first mechanism affected by this. Originally, EUS randomly initialises all the chromosomes of the population, so that, the number of 1s and 0s tend to be similar in the initial population. For imbalanced classification, it means that the resulting

preprocessed dataset would probably have an imbalanced distribution of classes. The original EUS corrects this issue throughout the evolution, by having a fitness function that favors chromosomes that produce a balanced dataset (so, typically a few number of selected majority instances are selected). To keep the chromosome size to a minimum, in our implementation, we randomly take a set of indexes in the range [0,M-1] of size equal to the number of minority class examples. Hence, the initial chromosomes and their corresponding datasets are initially balanced.

In order to assess and rank the quality of the chromosomes, the original EUS uses a fitness function that is based on how well the current chromosome balances the class distributions and an expected performance of the selected instances. Specifically, the performance is computed using the nearest neighbour algorithm to classify the examples of the training set with the selected instances represented in the chromosome. As performance measure, the g-mean is applied (defined in Eq. (2)).

The complete fitness function looks like this:

$$\text{fitness}_{\text{EUS}} = \begin{cases} \text{g-mean} - \left|1 - \frac{n^+}{N^-}\right| \cdot P & \text{if } N^- > 0 \\ \text{g-mean} - P & \text{if } N^- = 0, \end{cases} \quad (3)$$

where n^+ is the number of positive instances, N^- is the number of selected negative instances and P is a penalization factor that focuses on the balance between both classes. P is typically set to 0.2 as recommended by the authors, since it provides a good trade-off between both objectives.

As we stated before, the new codification obliges us to keep the chromosome size to a minimum from the beginning of the evolution. This means that we can get rid of the balancing component of the fitness function. Therefore, the fitness function will basically end up being the g-mean obtained in the training set.

Definitely, the fitness function will be the most costly operation throughout the whole evolutionary process. Thus, this step is going to be parallelised using Apache Spark. One should notice that the evaluation cost of nearest neighbour algorithm for each chromosome is intractable with big datasets. Subsection III-A discusses the details.

So far, the discussion above is valid for any genetic algorithm. As a particular search algorithm, we use the CHC evolutionary algorithm [17] that offers an excellent balance between exploration and exploitation. CHC is an elitist genetic algorithm making use of the heterogeneous uniform cross-over (HUX) for the combination of two chromosomes. It also uses an incest prevention mechanism and when the evolution does not progress, it reinitialises the population. The changes made in the representation of the chromosome slightly affect some of the operators of CHC.

- The HUX operator is combined with an incest prevention mechanism aiming at producing offspring that are maximally different from the parents. Incest prevention is achieved by impeding that two parents that are too similar in terms of their Hamming distance (over the original binary chromosome) are crossed. A crossover is then only permitted between randomly paired chromosomes with a Hamming distance divided by two greater than a given threshold d. When allowed, the heterogeneous uniform crossover (HUX) mechanism will exchange at random exactly half of the differing bits of the parents' chromosomes to make sure that offspring are significantly different from both parents.
  Within the sparse representation of the chromosome, we can simply extend the application of the Hamming distance to our representation. Indexes not present in both chromosomes will have a Hamming distance of 0. Therefore, the Hamming distance computation comes down to compare how many elements in both chromosomes are different from each other. For example, with a chromosome $X = \{4, 6, 8, 9\}$ and other $Y = \{4, 5, 7, 9\}$, there is a Hamming distance of 4. The crossover operator will keep common elements in both parents and it will take 50% of the elements from X that are not in Y (e.g. index 6), and vice-versa (e.g. index 7), to create two new chromosomes $Z = \{4, 6, 7, 9\}$ and $T = \{4, 5, 8, 9\}$.
- When the Hamming distance between any selected parents divided by two does not exceed the distance threshold d (i.e., not offspring are generated), the population is partially reinitialised. The new population is created, using the best chromosome obtained so far as a template. A percentage of the elements in the best chromosome (e.g. 35%) are randomly reinitialized (0 or 1 at random) according to a given parameter.
  With the sparse chromosome, we cannot get exactly the same implementation. However, we can achieve a very close implementation in which we randomly take elements from the best-so-far chromosome. For each element in the best chromosome if the random number in the range [0,1] is greater than a certain probability (e.g. 0.35), we take the corresponding element from the best chromosome, otherwise, we generate a random index in the range [0,M] and add it to the chromosome. In this way, we will end up having the same number of elements as the best chromosome. This point is important due to the fact that we are no longer considering the balancing of the dataset in the fitness function and hence, this phase will maintain the number of instances selected.

The parameter d is usually initialised to $d = L/4$, where L is the length of the chromosome. However, in our implementation the length of the chromosome is variable due to the sparse codification and L should be equal to the number of negative instances. The problem is that the probability of one index to be entered in the chromosome is too small (due to the imbalance ratio) given that few indexes are included in the initialisation (as many as the size of that of the minority class). As a consequence, the Hamming distance divided by two (incest prevention) will never be as big as $L/4$. In order to model the original behaviour of this parameter, we need to set it such that d is equal to half the number of indexes in the

chromosomes, that is, the number of minority class instances divided by two. This models the same behaviour because in the original case the number of ones in a chromosome is approximately $L/2$, which is divided by two to obtain $L/4$.

A. Spark-based CHC for Imbalanced big data

Here we now discuss the parallelisation details of our proposal, focusing on the required Spark operations. Algorithm 1 shows the pseudo-code of the EUS method with precise details of the functions utilised from Spark. In the following, we describe the most significant instructions, enumerated from 1 to 28.

Let *trainFile* be the training set stored in the HDFS as a single file. This file is composed of h HDFS blocks that can be examined from any computing node. The global EUS algorithm starts off reading the entire *trainFile* set from HDFS as an RDD, splitting the dataset into an user-defined number of *#Map* disjoint partitions (Instruction 1). This operation spreads the data across the computing nodes, caching the different subsets (Map_1,Map_2,...,Map_m) into main memory. Using a function *toLabeledPoint()*, the original text data is transformed into the LabeledPoint data structure of Spark.

Next, we split this dataset into two subsets: positive set *posTrainRDD* and negative set *negTrainRDD*, which contain only positive and negative instances, respectively. The filter transformation provided by Spark is used for this purpose. For sake of simplicity on the implementation of the chromosomes, the negative training set is zipped with indexes (using *zipWithIndex()* operation, see Instruction 2). In this work, we assume that the number of existing positive instances is so reduced that it will perfectly fit in the main memory of the driver node (as we did in [16]). Thus, Instruction 3 also collects the data from worker nodes and bring it to the driver. We will use this copy of the positive training set *posTrainDriver* later on.

When the data is well distributed across the cluster of computing nodes, we can now create the initial population and assess its quality (Instructions 5-8). To do so, we first follow the scheme explained above, creating sparse chromosomes at random. Later, we have to evaluate the quality of such chromosomes.

Algorithm 2 deepens into the necessary instructions to carry out such relevant operation. For each chromosome we have a collection of indexes representing the instances selected from the negative training set. On the one hand, we have to obtain the actual subset of the training set that is represented by the indexes of every chromosome of the population (from now on *reducedSet*). On the other hand, we have to evaluate such a subset against the training set.

To obtain the actual subset of the training set, we first have to filter the negative training set according to the indexes contained in the current chromosome. To do this, once again, we rely on the filter function provided with Spark (Instruction 2 of Algorithm 2). Since this is going to be a fairly small dataset, we also collect the data from the worker nodes to the driver. Next, both the selected negative instances and all the local copy of positive instances (posTrainDriver) are joined together (Instruction 3 of Algorithm 2) to form the resulting *reducedSet*.

Typically, the nearest neighbour algorithm is used to classify the training set with *reducedSet*. Due to the way of working of nearest neighbour, this would oblige us to have the *reducedSet* available in every single node (using for example the broadcast function). However, after some preliminary experiments with this approach, we concluded that the overhead created sending such information from the driver to the nodes slows down quite a lot the fitness function evaluation, compromising the feasibility of the whole approach.

For this reason, we base the fitness function on an eager model (and more specifically a Decision Tree), so that, we can learn a single model in the driver node, and broadcast it over all the worker nodes to classify the training set (See Instructions 4-5 in Algorithm 2). The main benefit of doing this is that the model will be a very small data structure compared to the *reducedSet*, and the classification phase will also be faster than in the case of nearest neighbour. One should question whether it would be better to avoid the collect phases by learning a global Decision Tree using the distributed and filtered data. However, we should emphasise the fact that the distributed learning is much slower due to the communication requirements, and repeating this process for each chromosome becomes too slow.

To accelerate the fitness evaluation, we have considered the usage of the windowing scheme defined in [15]. Under such scheme, the chromosomes will only be assessed against a subset of the training set each evaluation. It always includes the entire positive set and a random subset of the negative training set. Both datasets (posTrainRDD and negTrainRDD) were formerly in the form distributed datasets. Therefore, applying transformations on them is very straighforward and not very time consuming operations. Specifically, we randomly take negative instances according to the a given number of windows (established in [15] as the imbalanced ratio). Later, this the entire positive set is joined using the *union* operation, obtaining the *window* of the training set used for fitness evaluation. The detailed operation can be found in Instruction 6 of Algorithm 2. We should notice that this windowing scheme was totally necessary when nearest neighbour classifier was used to compute the fitness function. In this case, since we are simply obtaining the output of the model for each instance and not computing all the distances, the computational cost highly decreases. Hence, we will study whether the usage of the windowing scheme in this context continues providing a good trade-off between speed and performance or whether it is a better choice not to consider it in order to achieve the best performance. When the windowing scheme is deactivated (i.e. $NumWindows = 0$), Instruction 6 will simply join the entire *negTrainRDD* and the *posTrainRDD* sets.

After that, the classification takes place. We make use of a *mapPartitions(func)* transformation to concurrently access to the instances contained in the *window* set. The function

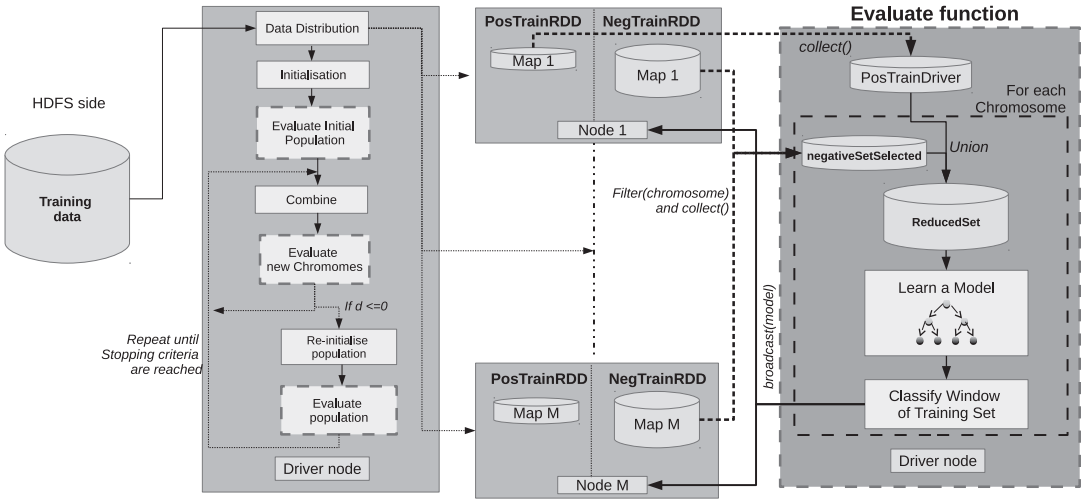

Fig. 3: EUS-global: Data flow

to be applied in every single portion of the data consists of classifying every single instance using the model broadcast before. As a result, this function will provide pairs $< trueclass, predictedclass >$ for each instance of the window. This information is brought to the driver node to compute the g-mean as fitness measure.

When the initial population is evaluated, this is sorted according to their fitness values (Instruction 10). Then, the CHC algorithm enters into a loop (Instructions 11 to 29) where search relies upon recombination and reproduction to create new potential solutions until a number of evaluation ($MAX\_EVALUATIONS$) is reached or we have re-initialised the population a number of times ($MAX\_REINITIALISATIONS$). This loop includes all the operations described before about the evolutionary process i.e. crossover with incest prevention (Instructions 12-15), elitist selection (Instructions 20-21), and reinitialisation of the population (Instructions 23-27).

As a result of the evolutionary process, we will obtain a balanced reduced set of instances that globally represent the entire training set. This dataset will be later used by a classifier to learn a model and classify the test set. In particular, we will use the Decision Tree provided in Apache Spark to classify the test set.

Figure 3 summarises the proposed model with a focus on which operations are carried out in the driver node and which are done in parallel.

IV. Preliminary results and discussion

In order to assess the correctness of the proposed method for imbalanced big data classification problems, we have conducted some preliminary experiments in one big dataset. It comes from the Evolutionary Big Data Competition ECBDL'14 [24], [25]. For this study, we consider a subset of 10% of the instances, in which the number of features was reduced from 631 to 90 by means of the feature selection algorithm applied in [25]. This dataset contains a total

Algorithm 1 EUS Global preprocessing

Require: trainFile; #Maps; #Windows
1: trainRDD ← textFile(trainFile, #Maps).toLabeledPoint().cache()
2: negTrainRDD = trainRDD.filter(line → line.contains("negative")).zipWithIndex()
3: posTrainRDD = trainRDD.filter(line → line.contains("positive"))
4: posTrainDriver = posTrainRDD.collect()
5: d = L/2
{Initialisation}
6: **for** $i = 1$ to NP **do**
7: $population_i$ = Randomly take numPositive indexes in the range [0,M-1].
8: $fitness_i$ = evaluate($population_i$, negTrainRDD, posTrainDriver,#Windows)
9: **end for**
10: population.sorted
11: **while** eval < MAX_EVALUATIONS **and** reinitialisations < MAX_REINITIALISATIONS **do**
12: offspring = crossover(population)
13: **if** $offspring.size > 0$ **then**
14: evaluate(offspring,negTrainRDD, posTrainDriver,#Windows)
15: offspring.sorted
16: **end if**
17: **if** $offspring.size == 0$ **or** $offspring(0).fitness < population(0).fitness$ **then**
18: d = d - 1
19: **else**
20: population = (offspring ++ population).sorted.take(NP)
21: evaluate(population.tail,negTrainRDD, posTrainDriver,#Windows)
22: **end if**
23: **if** d <= 0 **then**
24: re-initialise(population)
25: d = L/2
26: reinitialisations += 1
27: **end if**
28: **end while**

Algorithm 2 EUS-global: Parallel fitness function using Spark

Require: population; negTrainRDD; posTrainDriver; NumWindows
1: **for** $i = 1$ to NP **do**
2: negativeSetSelected = negTrainRDD.filter{case (key, value) → population(i).contains(key).collect()}
3: reducedSet = negativeSetSelected.union(posTrainRDD)
4: model = LearnDecisionTree(reducedSet)
5: model_broadcast sc.broadcast(model)
6: window = negTrainRDD.mapPartitions(dataset → RandomSelection(NumWindows)).union(posTrainRDD)
7: Outputs = window.mapPartitions(dataset → Classify(dataset, model_broadcast)).collect()
8: **end for**
9: **return** ComputeGmean(Outputs(True Class, Predicted Class))

of 3,489,083 instances, from which 69,133 belong to the positive class (i.e., an imbalanced ratio of 49, approximately).

In our experiments we consider a 5-fold stratified cross-validation model. To evaluate our model, we consider the AUC and g-mean measures. The experiments are carried out in a cluster with 12 nodes: a master node and eleven computing nodes. Each one of these nodes has 2 Intel Xeon CPU E5-2620 processors, 6 cores per processor (12 threads), 2.0 GHz and 64GB of RAM. The network is Gigabit ethernet (1Gbps). In terms of software, we have used the Cloudera's open-source Apache Hadoop distribution (Hadoop 2.6.0-cdh5.4.2) and Spark 1.6.2. A maximum of 216 concurrent tasks are available.

Table I collects the parameters used for the algorithms involved in this experiment. We consider the standard parameters for EUS [16]. In order to obtain a measure of the quality of the selected set after the evolutionary undersampling process, we have considered the Decision Tree included in Spark to learn a model and classify the test examples.

TABLE I: Parameters used for the involved algorithms.

Algorithm	*Parameters*
Decision Tree	Max Depth = 5, Minimum number of item-sets per leaf = 2
EUS-ImbBD	Population Size = 50, Number of Evaluations = 10000, Probability of inclusion HUX = 0.25, Evaluation Measure = g-mean, Selection Type = Majority, Balancing = True, P = 0.2 Maps = 512, Windowing activated
EUS-global	Population Size = 50, Number of Evaluations = 10000, Probability of inclusion HUX = 0.25, Evaluation Measure = g-mean,

Since the main bottleneck of a global approach is the fitness function evaluation, the first point we wanted to check is the average runtime required to carry out that operation depending on the number of maps used. To do this, we make use of the performance measures provided by the web UI of Apache Spark. Note that this information is always limited to actions, rather than transformations.

There are two main processes involved in the fitness evaluation, where information must come from the worker nodes to the driver: collecting the selected negative set of instances, and classifying and evaluating the corresponding window (if any) of the training data. We also account for the necessary time to learn a model within the driver node. Table II presents a comparison of the average times (average over 100 fitness evaluations) of each of these phases, including a comparison amongst the classification time needed keeping windowing activated and deactivated.

TABLE II: Fitness evaluation time according to the number of maps

#Maps	Collect time(s)	Learning time(s)	Classif. time(s) No windows	Classif. time(s) Windowing
12	0.5830	0.8766	0.2427	0.5278
64	0.6250	0.8621	0.2273	0.5148
128	0.6412	0.8505	0.1924	0.4854

As we can observe, the larger the number of maps is, the higher the time spent to collect the negative selected set from the workers to the driver is. On the other hand, the classification time seems to follow the opposite behaviour. Nevertheless, this phase is so quick (due to the use of Decision Tree model) that increasing the number of maps does not really speed up the classification time, which is fairly small. Hence, it is clear that using a very high number of maps creates a small overhead in the distributed processing, making it not really worthy to accelerate the fitness evaluation.

We can also observe that the classification time can be reduced by half using the windowing scheme. Although it does seem to be a very high improvement with respect to the time necessary to evaluate the fitness function, the reduction in the total time invested by the whole evolutionary process (with up to 10,000 evaluations) may be worthwhile. It is interesting to note that the reduction in time due to the usage of the windowing scheme is not linear, since the set of data evaluated is approximately 25 times smaller (half of the IR) and the time is only reduced to a half. Hence, the filtering phase for selecting the data to be classified also produces some overhead reducing the effectiveness of the windowing, given that the classification is really fast due to the usage of a model instead of nearest neighbour classifier.

The second goal of this experiment is to check whether the algorithm is able to converge or not in such a huge search space. Figure 4 depicts a convergence plot in which every time we evaluate a new bunch of chromosomes, we provide the best g-mean obtained so far. Both, the algorithm using windowing and not using it are compared in this plot.

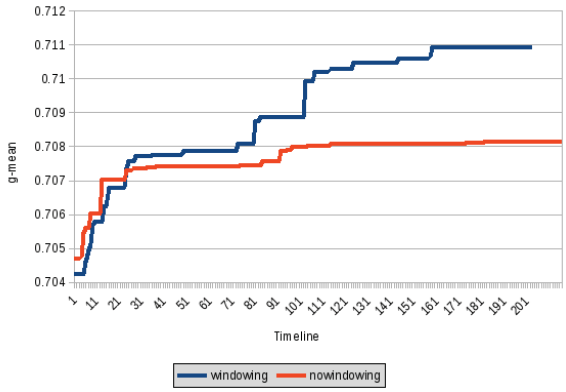

Fig. 4: EUS-global: Convergence of the algorithm.

From this figure, we can observe that the algorithm clearly evolves towards a better solution in both cases and therefore, the global selection seems to work as expected. Nevertheless, there are differences in the behaviour of the model with or without windowing. When the windowing mechanism is applied, the bunch of training data assessed every single fitness evaluation is way smaller. Thus, it is somehow easier that the selection of more relevant examples in the chromosomes are more rapidly reflected in the fitness value.

Finally, in Table III we compare the performance of the global model against our previous local approach EUS-Imb. In this case and to make the comparison as fair as possible,

we have used the local EUS model from [15] but instead of learning a local decision tree in each partition, all the selected negative instances and the positive ones have been used to build a global Decision Tree with Spark. This way, the only difference between the two models is the way the EUS is carried out. We have not considered the EUSextImb model from [16] as it really replicates positive class instances, being less comparable in terms of global vs. local approaches.

As we stated before, in terms of runtime, the advantage of the local model against this new proposal is clear independently of the use of windowing. However, with respect to the classification performance, both in terms of AUC and G-mean the global model obtains better results. Comparing our proposal with and without windowing, the runtime is naturally reduced when applying windowing. Very interestingly, however, the performance for this particular dataset is also improved with the windowing approach as it seemed to converge to a better solution.

It is clear that we could not draw meaningful conclusions from one dataset with such a slight improvement. However, the results obtained are promising and encourage us to further concentrate on developing global models, or instead, consider hybrid models that could take advantage of the global view of the whole data and the speed of the local approaches.

TABLE III: Preliminary results obtained with ECBDL14 datasets (10%) with 90 features

	Preprocessing Time	Reduction Rate	AUC	G-mean
EUS-local	99.1720	95.7613	0.7090	0.7088
EUS-global(windowing)	18338.6602	95.7256	0.7111	0.7108
EUS-global	21622.9775	95.8645	0.7100	0.7099

V. Concluding remarks

In this contribution we have carried out a first attempt on global evolutionary undersampling for imbalanced big data classification. To do so, we have focused on Spark as big data technology. The main advantage of this model in comparison to existing local approaches is that it will analyse all the data as a whole. Our preliminary results show the potential of this scheme. However, we still need to extend our experiments to get more insights. As future work, we consider that the design of hybrid approaches that accelerate even more the fitness function evolution may result in a very suitable approach to deal with imbalance big data classification from a global perspective.

References

[1] M. Minelli, M. Chambers, and A. Dhiraj, *Big Data, Big Analytics: Emerging Business Intelligence and Analytic Trends for Today's Businesses (Wiley CIO)*, 1st ed. Wiley Publishing, 2013.

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 1–14.

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP '03, 2003, pp. 29–43.

[4] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[5] A. Fernández, S. Río, V. López, A. Bawakid, M. del Jesus, J. Benítez, and F. Herrera, "Big data with cloud computing: An insight on the computing environment, mapreduce and programming frameworks," *WIREs Data Mining and Knowledge Discovery*, vol. 4, no. 5, pp. 380–409, 2014.

[6] K. Grolinger, M. Hayes, W. Higashino, A. L'Heureux, D. Allison, and M. Capretz, "Challenges for mapreduce in big data," in *Services (SERVICES), 2014 IEEE World Congress on*, June 2014, pp. 182–189.

[7] A. F. Project, "Apache flink," 2017. [Online]. Available: https://flink.apache.org/

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

[9] J. Maillo, S. Ramírez, I. Triguero, and F. Herrera, "knn-is: An iterative spark-based design of the k-nearest neighbors classifier for big data," *Knowledge-Based Systems*, vol. 117, pp. 3 – 15, 2017, volume, Variety and Velocity in Data Science.

[10] V. López, A. Fernández, S. García, V. Palade, and F. Herrera, "An insight into classification with imbalanced data: Empirical results and current trends on using data intrinsic characteristics," *Information Sciences*, vol. 250, no. 0, pp. 113 – 141, 2013.

[11] G. Weiss, "Mining with rare cases," in *Data Mining and Knowledge Discovery Handbook*. Springer, 2005, pp. 765–776.

[12] M. Galar, A. Fernández, E. Barrenechea, H. Bustince, and F. Herrera, "A review on ensembles for the class imbalance problem: Bagging-, boosting-, and hybrid-based approaches," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 42, no. 4, pp. 463–484, 2012.

[13] M. Galar, A. Fernández, E. Barrenechea, and F. Herrera, "Eusboost: Enhancing ensembles for highly imbalanced data-sets by evolutionary undersampling," *Pattern Recognition*, vol. 46, no. 12, pp. 3460–3471, 2013.

[14] S. García and F. Herrera, "Evolutionary under-sampling for classification with imbalanced data sets: Proposals and taxonomy," *Evolutionary Computation*, vol. 17, no. 3, pp. 275–306, 2009.

[15] I. Triguero, M. Galar, S. Vluymans, C. Cornelis, H. Bustince, F. Herrera, and Y. Saeys, "Evolutionary undersampling for imbalanced big data classification," in *Evolutionary Computation (CEC), 2015 IEEE Congress on*, May 2015, pp. 715–722.

[16] I. Triguero, M. Galar, D. Merino, J. Maillo, H. Bustince, and F. Herrera, "Evolutionary undersampling for extremely imbalanced big data classification under apache spark," in *2016 IEEE Congress on Evolutionary Computation (CEC)*, July 2016, pp. 640–647.

[17] L. J. Eshelman, "The CHC adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination," in *Foundations of Genetic Algorithms*, G. J. E. Rawlins, Ed. San Francisco, CA: Morgan Kaufmann, 1991, pp. 265–283.

[18] A. H. Project, "Apache hadoop," 2013. [Online]. Available: http://hadoop.apache.org/

[19] T. Fawcett, "An introduction to roc analysis," *Pattern recognition letters*, vol. 27, no. 8, pp. 861–874, 2006.

[20] S. del Río, V. López, J. Benítez, and F. Herrera, "On the use of mapreduce for imbalanced big data using random forest," *Information Sciences*, vol. 285, pp. 112–137, 2014.

[21] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[22] V. López, S. del Río, J. Benítez, and F. Herrera, "Cost-sensitive linguistic fuzzy rule based classification systems under the mapreduce framework for imbalanced big data," *Fuzzy Sets and Systems*, vol. 258, pp. 5–38, 2014.

[23] S. García, J. Derrac, J. Cano, and F. Herrera, "Prototype selection for nearest neighbor classification: Taxonomy and empirical study," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, no. 3, pp. 417–435, 2012.

[24] "ECBDL14 dataset: Protein structure prediction and contact map for the ECBDL2014 big data competition," 2014. [Online]. Available: http://cruncher.ncl.ac.uk/bdcomp/

[25] I. Triguero, S. del Río, V. López, J. Bacardit, J. M. Benítez, and F. Herrera, "ROSEFW-RF: The winner algorithm for the ecbdl'14 big data competition: An extremely imbalanced big data bioinformatics problem," *Know.-Based Syst.*, vol. 87, no. C, pp. 69–79, Oct. 2015.