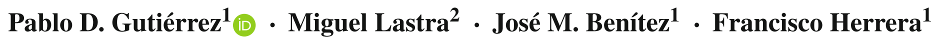
Prog Artif Intell
DOI 10.1007/s13748-017-0128-2

REGULAR PAPER

SMOTE-GPU: Big Data preprocessing on commodity hardware for imbalanced classification

Pablo D. Gutiérrez[1] · Miguel Lastra[2] · José M. Benítez[1] · Francisco Herrera[1]



Received: 20 March 2017 / Accepted: 27 April 2017
© Springer-Verlag Berlin Heidelberg 2017

Abstract Nowadays, it is usual to work with large amounts of data since our capacity of collecting and storing information has increased significantly. The extraction of knowledge from these scenarios is commonly known as "Big Data," and it is performed on large clusters with MapReduce platforms. Imbalanced classification poses a problem both in traditional and Big Data learning scenarios. Data sampling is one of the ways that allows to improve the performance on imbalanced problems. A commodity hardware-based method for Big Data problems can offload these computations from the expensive and highly demanded hardware that MapReduce platforms require. The characteristics of some sampling methods make them suitable to be adapted to commodity hardware, taking advantage of the parallel computation capabilities of graphics processing units. SMOTE is one of the most popular oversampling methods which is based on the nearest neighbor rule. The proposed SMOTE-GPU efficiently handles large datasets (several millions of instances) on a wide variety of commodity hardware, including a laptop computer.

✉ Pablo D. Gutiérrez
pdgp@decsai.ugr.es

Miguel Lastra
mlastral@ugr.es

José M. Benítez
J.M.Benitez@decsai.ugr.es

Francisco Herrera
herrera@decsai.ugr.es

[1] Department of Computer Science and Artificial Intelligence, CITIC-UGR, University of Granada, 18071 Granada, Spain

[2] Department of Software Engineering, CITIC-UGR, University of Granada, 18071 Granada, Spain

Keywords Imbalanced classification · SMOTE · CUDA · Big Data

1 Introduction

Nowadays, it is usual to work with large amounts of data since our capacity of collecting and storing information has increased significantly. The extraction of knowledge from these scenarios is commonly known under the term "Big Data" [17,27]. This term applies to situations that traditional Knowledge Discovery methods are unable to deal with. MapReduce [7] platforms, such as Apache Hadoop [25] or Apache Spark [26], were created to cope with the computational challenges that these new scenarios create. There are machine learning libraries [10,18,19,23] that have been created to work with these platforms on Big Data problems, but there are still challenges that have not been solved, like imbalanced classification [9].

In traditional knowledge discovery, it is not unusual to find situations where the number of instances of each class of a problem is significantly different, this problem is usually known as imbalanced classification [13,15,20] and poses challenges to traditional learning algorithms. Considering a binary problem with a majority class and a minority class, it is likely that a learning algorithm ignores the later and still achieves a high accuracy. There are three main ways of dealing with these situations [16]:

- *Algorithmic modification* Modifying learning algorithms in order to tackle the problem by design.
- *Cost-sensitive learning* Introducing costs for misclassification of the minority class at data or algorithmic level.
- *Data sampling* Preprocessing the data in order to reduce the breach between the number of instances of each class.

The algorithms that were used to sample the data on traditional scenarios could be used for Big Data, since the ideas behind them are not related to the number of instances of the problem, but traditional implementations cannot handle the large amounts of data required. These algorithms can be redesigned to use MapReduce platforms but these platforms require large clusters [7] that are usually expensive and shared among different users. Usually several configurations are tried in order to obtain the best performance [24].

An interesting alternative would be if the data sampling could be performed separated from the learning process and in commodity hardware using the MapReduce platforms only to get real classification results.

Graphics processing units (GPUs) are available in almost every medium or high-end commodity computer. These devices were created to compute 3D-related computations in an efficient way, but their parallel architecture makes them interesting for many other applications as varied as fingerprint identification [12], molecular simulation [22] or data mining [21]. Libraries like NVIDIA CUDA [6], have made these devices easier to use for general-purpose applications, presenting the GPU device as a parallel co-processor. Modern GPU devices should be powerful enough to perform the computations required for data sampling algorithms in a reasonable time.

In this paper, we explore the use of GPU devices combined with a proper data handling design in order to perform data sampling algorithms based on the well-known SMOTE algorithm [5] in a reasonable time and on commodity hardware, called SMOTE-GPU. Our test covers four different datasets up to 10 millions of instances with imbalanced ratios (the relation between the numbers of instances of both classes) up to 49 and hardware configurations ranging from a server class GPU device to a medium range laptop. All test where completed in every hardware platform in less than 2 h, for the most time-demanding experiment, while a traditional CPU implementation could not produce results for any experiment in more than 8 h.

The rest of the paper is organized as follows: Sect. 2 shows the characteristics of GPU devices and comments the main sampling solutions to deal with imbalanced data and its suitability to be adapted to commodity hardware; Sect. 3 describes our design; Sect. 4 presents the experiments and obtained results; Sect. 5 shows the conclusions and future work.

2 Background

In this section, we describe the main characteristics of GPU devices (Sect. 2.1), and we discuss the suitability for commodity hardware of different data sampling algorithms (Sect. 2.2).

2.1 Graphics processing units

GPU devices were created to offload the computations related to 3D related computations from the CPU device to a specific and efficient device. These devices have a parallel architecture, usually single instruction, multiple data (SIMD), which allows them to perform the same operation on different data at once. This architecture is different to the architecture that we find in CPU devices, making GPU device programming quite different from traditional parallel and distributed programming. NVIDIA CUDA is one widely used library that allows general-purpose programming of NVIDIA GPU devices by presenting them as parallel co-processors.

The functions that are run on GPU devices are called kernels. When programming a kernel, the kernel code includes the operations that a single thread is going to perform. When the kernel is called, the code that calls it specifies a set of threads that will run the kernel, called grid. The grid is divided into blocks of threads, each block has a three-dimensional block identifier, and each thread within a block has another three-dimensional thread identifier. These identifiers are accessible in the kernel code and are used to access data from each thread and to make threads cooperate in computations. The threads that belong to the same block can cooperate and communicate using a programmable cache that works as shared memory. The size of this cache is limited; if a kernel requires a large amount of shared memory, the number of blocks running at once will be reduced, reducing also the performance obtained.

The grid is distributed on the GPU cores in a transparent way, but there are some lower level characteristics that have to be taken into account in order to obtain good performance. The GPU cores are organized in streaming processors (SMX) that run sets of 32 threads, called warps in a synchronous way. This means that a warp of threads is always running the same instruction and that if there is code divergence, for instance an if clause with different results within the warp, that section of code will be serialized. A block has to be run on the same SMX, and this is because the shared memory is implemented at the SMX level. There are also limits to the maximum number of warps and blocks that an SMX can handle at once, if a block has a small number of threads the resources will not be fully used, but a large number of threads can also create this situation since the SMX only can handle complete blocks. In both situations, the performance obtained is reduced.

A proper use of the GPU resources is usually a critical factor on the performance because of the way the GPU optimizes its memory accesses. GPU devices have a large number of registers per SMX; this is because each thread is assigned the registers that is going to need to run the code beforehand. Because of this, a GPU device can switch from one warp to another in an extremely fast way since all the required

information is always in registers. When the main memory is accessed, the GPU device switches to another warp and runs it, and in this way, the latencies of memory accesses are hidden with useful computation.

Since GPU devices are separated from CPU devices, they have their own memory. This means that the data used in GPU computations needs to be copied from the computer's main memory to the GPU device memory. These computations can be performed asynchronously at the same time other computations are performed on the GPU device. Also, the GPU memory cache system is optimized for coalescent accesses, subsequent threads are expected to access subsequent memory positions. If the memory is not accessed in this way, it can also lead to lower performance.

A good GPU-based design has to tackle all these restrictions and dependencies, identify which parts of the algorithm are suitable for the GPU device, and try to take advantage of the asynchronous memory transferences in order to obtain the best performance.

2.2 Imbalanced classification

A problem is imbalanced when the amount of instances of one class is significantly larger than the amount of instances from the other. The imbalanced ratio (IR) measures how imbalanced a dataset is. This ratio is computed dividing the number of instances of the larger class by the number of instances of the smaller class. This type of problems poses a challenge for classification algorithms, especially as the IR increases, since the information regarding to the minority class is limited.

There are three main ways to deal with imbalanced classification: modifying the algorithm, introducing misclassification cost and data sampling. Data sampling is the only one that can be performed in a separate way from the classification algorithm, since the other two require direct or indirect modifications of the algorithm. When sampling data, there are two obvious strategies to solve the imbalanced problem: undersampling the majority class or oversampling the minority class. However, the memory requirements of each type of algorithm are quite different making only one type suitable to be run on commodity hardware.

Undersampling methods, such as random undersampling or instance selection methods, balance the number of instances of the classes by reducing the number of instances of the majority class. The instances selected can be chosen in a random way or using some type of expert knowledge over the majority class. This requires to manipulate most of the data in order to perform the selection. In a Big Data scenario, the resources required to handle the majority class are virtually the same required to handle the whole problem, and for this reason, this type of approach is not advised if we want to perform this data transformation on commodity hardware.

Oversampling methods, on the other hand, balance the number of instances of the classes by increasing the number of instances of the minority class. The new instances can be obtained by replicating existing instances or using some type of interpolation between existing instances. Since these algorithms only need to handle the information referred to the minority class it is likely that this can be performed on a single computer. The Synthetic Minority Oversampling Technique (SMOTE) [5] and the random oversampling technique (ROS) are some of the most common techniques of this type. The first one is an interpolation technique, while the second one is based on duplicating instances.

The random oversampling technique creates a new instance by selecting one real instance randomly and duplicating it. This procedure is repeated until the number of instances of both classes has been balanced or a user specified parameter value is reached.

The SMOTE technique is based on the idea of neighborhood of the k-nearest neighbor (kNN) rule. When used in classification, the kNN rule sets the class of an instance as the majority class of the k closest instances of the training set. SMOTE considers that a instance can be interpolated between an instance and one of its neighbors within the class. The algorithm computes the neighborhood of each instance of the minority class, choosing one of its neighbors and randomly interpolates a new instance using the values of each attribute as limits. If the number of instances to create is smaller than the size of the original dataset, the algorithm randomly chooses the original instances used to create the artificial ones. If that number is larger than the original dataset size, the algorithm iterates over the dataset creating an artificial instance per original instance until it reaches the previous scenario.

The main computational bottleneck of the SMOTE algorithm is the neighborhood computation. The distance between each pair of instances needs to be measured and all those distances needs to be compared in order to find the neighborhood. GPU devices have proven to be efficient in the computation of the kNN rule, reducing the time required for its computation in a significant way. Section 3 shows how we can run these algorithms on commodity hardware combining a proper data handling design and a GPU device.

3 Design for preprocessing on commodity hardware: SMOTE-GPU

In this section, we present how the memory requirements for SMOTE can be adjusted to fit in commodity hardware and how the computations can be performed on GPU devices. First, the memory requirements are tackled (Sect. 3.1)

and, after that, the design of SMOTE-GPU is addressed (Sect. 3.2).

3.1 Memory requirements

The usual applications that perform oversampling algorithms work in a pretty naive way, loading the complete dataset in memory, separating it in different classes, computing all the new instances and writing the whole result on the hard drive. This approach works perfectly for traditional problems, but working on Big Data scenarios, it is unlikely that the whole dataset fits in device memory.

As commented in Sect. 2.2, oversampling methods only require to work with the data of the minority class. Our approach only keeps in memory that data. When reading the dataset, each instance is written to disk straight away after it is read, and then, depending on its class, it is only kept in memory, if the class corresponds with the minority class.

In the same way, it is not necessary to create all the artificial instances before writing them to disk. A single extra instance can be reused to store the results of the interpolation if the results are stored on the hard drive just after that. Algorithm 1 lines 1–7 and 17–20 correspond to this data management scheme.

In this way, the memory requirements for the application, in terms of dataset storage, are reduced to the size of the instances of the minority class plus one extra instance. An oversampling method that does not perform much computation with the data, like ROS, can work with only these changes, but other methods, like SMOTE, would still require too much time to perform their computations.

Algorithm 1: Proposed algorithm pseudocode

```
   input : dataset
   output: oversampled dataset
1  while There are instances to read do
2  |  instance ← readInstance
3  |  writeToDisk(instance)
4  |  if instance.class = minorityClass then
5  |  |  minorityInstances ← instance
6  |  end
7  end
8  for i ← 1 to N do
9  |  copyMinorityInstances_i
10 |  for j ← 1 to N do
11 |  |  copyMinorityInstances_j
12 |  |  computeDistanceMatrix_{i,j}
13 |  |  computeSelection_j
14 |  end
15 |  copyNeighborhood_i
16 end
17 while There are artificial instances to create do
18 |  instance ← createArtificalInstance()
   |  writeToDisk(instance)
19 end
```

3.2 SMOTE-GPU design

Applying the previously mentioned memory scheme, the SMOTE algorithm needs to compute the neighborhood of each instance to interpolate. As commented in Sect. 2.2, there are several GPU implementations for the kNN classification problem that can be adapted to this situation.

The computation of the kNN rule on GPU devices is split into two kernels, one kernel that builds a distance matrix between test and training sets and another kernel that searches for the k minimum distances obtained. Most GPU-based designs struggle when the distance matrix does not fit on GPU device memory. In [11], the different versions that can handle large amounts of data are studied and compared. GPU–SME–kNN obtains the best performance among them, being able to compute the kNN rule in datasets of more than 4.5 millions of instances, so it can be considered a good candidate for the SMOTE method.

GPU–SME–kNN [11] computes the distance matrix using a block based scheme. The size of each block can be defined by user-defined parameters, so it can be adapted to a large variety of GPU devices. Each block of the matrix is computed in a kernel call, each row corresponds to a thread block, but the number of threads per block is fixed to a value d, smaller than the length of the matrix block. Each thread computes several distances in a coalescent way.

The selection of the neighborhood is computed sequentially after each block of the distance matrix is computed. For the first matrix block of a strip, the neighborhood is computed, and when the second block of the strip is ready, the new neighborhood is computed combining the previous one and the matrix block, and the process continues until the last block of the matrix strip has been computed. Lines 8–16 on Algorithm 1 correspond to a version of GPU–SME–kNN that show the general scheme of the method.

GPU–SME–kNN[1] uses a quicksort [14] based selection. This type of selection allows to discard all the elements of a block that do not improve the previous neighborhood in lineal time, using as pivot the furthest neighbor of the previous block.

Another particularity of GPU–SME–kNN is that it uses a separate kernel to compute the square root operation required for the distance computation. The neighborhood comparison can be performed obtaining the same results without performing that operation that it is only applied to the finally selected neighbors in order to obtain the real distance results.

Finally, one of the key aspects of GPU–SME–kNN is the use of asynchronous memory transfers. The data required for the distance computation corresponds with the instances attributes values, this data is not used during the selection process so the data required for the computation of the next

[1] http://sci2s.ugr.es/GPU-SME-kNN.

matrix block can be loaded while the selection is computed. In the same way, the final neighborhood can be copied to CPU main memory while the computations of the next matrix block are performed. This way all the memory transfers between CPU and GPU devices are overlapped with computation, except the initial transfer for the first matrix block and the transfer of the last neighborhood.

The last step of the SMOTE technique is the interpolation of the instances. This part is performed on CPU for two reasons, the first one is to minimize the memory requirements since each instance needs to be write as soon as is computed; the second one is because it would require to store large portions of the dataset, if not all of it, on device memory. Furthermore, the data are not accessed on a coalescent way which would lead to bad performance, if this step were computed on GPU devices.

SMOTE-GPU can be adapted to a broad range of GPU devices and can work with large datasets with the only prerequisite that the fit on CPU main memory. The memory scheme proposed in Sect. 3.1 produces a scenario, where all the data required for the oversampling process is stored in main memory, that combined with this GPU-based *k*NN computation makes possible the use of the SMOTE technique over large datasets in a single machine.

4 Experimental study

Different experiments have been carried out to check the results obtained by our design for SMOTE-GPU. The section is organized as follows: The experiments are described in Sect. 4.1; the different hardware configurations are detailed in Sect. 4.2; the obtained results are shown and discussed in Sect. 4.3.

4.1 Experimental framework

Two different types of experiments have been performed. The first type focuses on the time needed to apply the sampling technique to different datasets, while the second types studies the classification results obtained after applying the sampling method.

Four different datasets have been used: ECBDL14, HEPMASS, Higgs, and Susy. The first one from the ECBDL 14 competition [8], after a feature selection process to reduce it to 90 features [24] and the other three from the UCI repository [2,3]. The UCI datasets have been modified to create datasets with different imbalanced ratio, obtained undersampling the minority class. Table 1 shows the number of instances of each class, the number of attributes of the problem and the imbalance ratio for each dataset used. These datasets have been split following a fivefold cross-validation scheme, so every result shown in this paper corresponds to the average of the results obtained on each fold.

Table 1 Datasets information

Dataset	Majority	Minority	Att.	IR
ECBDL14-05mill-90	470,400	9600	90	49
ECBDL14-10mill-90	9,408,000	192,000	90	49
SUSY_IR_16	2,169,740	135,610	18	16
SUSY_IR_4	2,169,740	542,435	18	4
SUSY_IR_8	2,169,740	271,219	18	8
HIGGS_IR_16	4,663,300	291,457	28	16
HIGGS_IR_4	4,663,300	1,165,825	28	4
HIGGS_IR_8	4,663,300	582,913	28	8
HEPMASS_IR_16	4,200,100	262,507	28	16
HEPMASS_IR_4	4,200,100	1,050,029	28	4
HEPMASS_IR_8	4,200,100	525,013	28	8

SMOTE-GPU and ROS have been used as oversampling method, with two different configurations. The first configuration balances the number of instances of the minority class while the second one introduces 50% of overhead for the minority class. The value of k for the SMOTE-GPU algorithm is set to 5, and the rest of the parameters for the *k*NN algorithm have been the same specified in [11].

We also wanted to compare the time performance with the ones obtained by software available that performs data preprocessing. We tried to run the SMOTE implementation available on Keel [1] but we were not able to obtain results for a single experiment on these datasets after more than 8 h of runtime on a server node.

To check the accuracy of the oversampled datasets, the decision tree from MLlib has been used. Since the datasets are large, the depth of the trees has been set to the maximum value that MLlib allows for this algorithm, which is 30. The area under the ROC curve (AUC) [4] has been used as measure to show the quality of the results since the classification accuracy of the algorithm is not useful when the data is imbalanced.

4.2 Hardware configurations

Different hardware configurations have been use for the sampling methods in order to prove their suitability on hardware of different characteristics.

- *Server* a cluster node that uses an NVIDIA Tesla k20 GPU with 5 GB of memory and 2496 CUDA cores, an Intel Xeon E5-2630 processor at 2.30 GHz and 128 GB of main memory.
- *Desktop* a desktop computer that uses an NVIDIA GeForce GTX 680 with 2 GB of memory and 1532

Table 2 Time results, in seconds, for the SMOTE-GPU algorithm

	Server	Desktop	Laptop
ECBDL14-05mill-90	99.64	73.19	110.55
ECBDL14-10mill-90	2066.48	1553.82	2546.65
SUSY_IR_16	164.22	128.91	233.99
SUSY_IR_4	308.49	323.20	1149.29
SUSY_IR_8	196.51	169.55	420.34
HIGGS_IR_16	584.09	445.31	930.70
HIGGS_IR_4	1387.20	1508.78	6701.97
HIGGS_IR_8	730.63	661.74	2093.40
HEPMASS_IR_16	504.58	391.31	799.39
HEPMASS_IR_4	1174.35	1238.15	5477.82
HEPMASS_IR_8	637.18	553.78	1742.41

Table 3 Time results, in seconds, for the ROS algorithm

	Server	Desktop	Laptop
ECBDL14-05mill-90	55.90	40.25	58.43
ECBDL14-10mill-90	1101.16	777.76	1197.57
SUSY_IR_16	90.71	63.21	94.88
SUSY_IR_4	107.06	78.95	112.88
SUSY_IR_8	97.60	68.47	102.82
HIGGS_IR_16	281.42	204.17	305.73
HIGGS_IR_4	340.56	245.06	362.14
HIGGS_IR_8	301.10	215.81	324.14
HEPMASS_IR_16	254.42	180.47	269.97
HEPMASS_IR_4	299.76	214.69	317.97
HEPMASS_IR_8	268.01	190.02	283.71

Table 4 Time results, in seconds, for the SMOTE-GPU algorithm with extra 50% of minority class

	Server	Desktop	Laptop
ECBDL14-05mill-90	122.35	91.58	134.15
ECBDL14-10mill-90	2535.94	1858.10	3087.82
SUSY_IR_16	196.21	148.13	267.96
SUSY_IR_4	333.55	332.90	1183.28
SUSY_IR_8	226.92	185.30	447.18
HIGGS_IR_16	648.01	512.93	1034.11
HIGGS_IR_4	1440.68	1549.95	6782.25
HIGGS_IR_8	803.61	710.07	2186.71
HEPMASS_IR_16	582.31	448.76	910.48
HEPMASS_IR_4	1233.46	1292.08	5561.03
HEPMASS_IR_8	717.06	612.99	1840.47

Table 5 Time results, in seconds, for the ROS algorithm with extra 50% of minority class

	Server	Desktop	Laptop
ECBDL14-05mill-90	56.34	39.30	59.28
ECBDL14-10mill-90	1105.64	787.02	1181.07
SUSY_IR_16	90.67	64.75	97.82
SUSY_IR_4	106.89	76.13	112.46
SUSY_IR_8	96.73	69.72	101.96
HIGGS_IR_16	286.13	208.64	300.95
HIGGS_IR_4	332.43	241.33	356.74
HIGGS_IR_8	300.47	221.16	322.44
HEPMASS_IR_16	262.08	180.08	270.22
HEPMASS_IR_4	303.07	212.66	317.32
HEPMASS_IR_8	269.94	192.54	285.68

CUDA cores, an Intel core i7 3820 processor at 3.60 GHz and 24 GB of main memory.

- *Laptop* a laptop computer that uses an NVIDIA GeForce GTX 740 m with 384 CUDA cores, an Intel Core i5 3337U processor at 1.8 GHz and 8 GB of main memory.

The accuracy measurements of the MLlib experiments have been performed on a small spark cluster with four worker nodes, and each worker has 8 threads and 28 GB of main memory.

4.3 Analysis of the results

Tables 2, 3, 4 and 5 show the average time results, in seconds, obtained on each cluster for each algorithm.

The time to perform the complete process is included in these results, considering also the time spent in reading the dataset and writing the results on the hard disk. As expected, the largest time is obtained by the laptop computer. However, even for more time-demanding experiment, it is shorter than 2 h, and it is only around four times slower than the fastest time obtained for the same experiment.

The desktop configuration is faster than the server one in many cases. The reason for this is that the interpolation process is performed on CPU device, and single thread performance of the CPU device on the desktop is higher than the CPU device on the server, as it was shown in the hardware description. This behavior could be expected for the ROS algorithm but it also happens in SMOTE-GPU.

The NVIDIA Tesla k20 GPU from the server delivers a higher performance than the NVIDIA GTX 680 from the desktop but the faster CPU of the desktop compensates these differences in most cases. The number of instances to create has more importance now, comparing the results for the dataset SUSY_IR_4 in Tables 2 and 4, corresponding to both SMOTE-GPU configurations. The server is faster that the desktop in the first case but slower in the second. The only difference between both settings is the number of artificial instances created. This means that, if the GPU device is pow-

Table 6 Time results, in seconds, for the neighborhood computation in the SMOTE-GPU algorithm

	Server	Desktop	Laptop
ECBDL14-05mill-90	0.34	0.33	1.65
ECBDL14-10mill-90	57.75	64.85	390.17
SUSY_IR_16	10.63	13.50	64.81
SUSY_IR_4	149.77	201.35	979.60
SUSY_IR_8	38.81	51.16	247.95
HIGGS_IR_16	56.24	71.20	390.14
HIGGS_IR_4	862.24	1119.96	6149.25
HIGGS_IR_8	218.75	281.32	1543.06
HEPMASS_IR_16	47.39	59.44	327.08
HEPMASS_IR_4	701.65	911.01	4997.47
HEPMASS_IR_8	180.41	230.96	1269.28

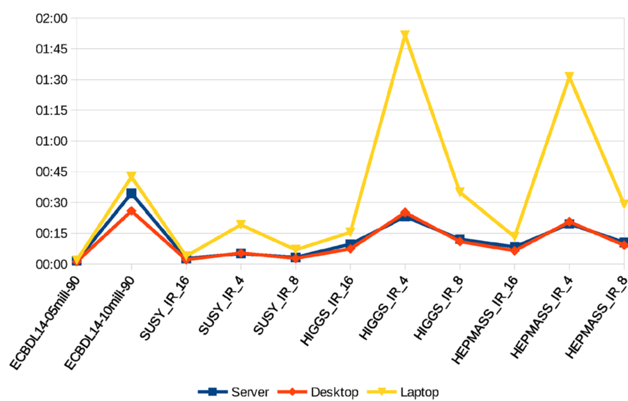


Fig. 1 SMOTE-GPU time performance in hours and minutes

erful enough, the bottleneck moves from the neighborhood computation to the interpolation and data reading and storage.

Table 6 shows the time required for the *k*NN algorithm on each dataset and confirms what could be expected considering the different capabilities of the GPU devices used. It can be seen how the server configuration is faster than any other in this step. It also shows that the importance of this step, in terms of time, is much higher than in the other configurations, reaching up to 90% of the total time in some cases. However, the fact that a medium range 3-year-old laptop can handle these datasets in less than 2 h, as can be seen in Fig. 1, shows how powerful this design can be.

Considering that the time required to read and store the data has now a significant importance on the global performance of the algorithm it is important to state that in all the experiments the data was stored in a traditional magnetic hard drive. It is likely that using a solid-state disk these times would be reduced, especially for the server and desktop configurations.

Table 7 shows the values for the area under the ROC curve obtained using the MLlib decision tree with the original dataset and the sampled datasets.

These results show how the oversampling methods outperform the results obtained by the original dataset. SMOTE-GPU with an extra 50% of the minority class achieves the best results in all datasets. We can also observe how the results of the first configuration of SMOTE-GPU are better than ROS and than the original problem for all datasets except for HIGGS. The ROS algorithm seems to be leading to overfitting when an extra 50% of the minority class is created. In that case, only four experiments improve the results obtained by the first configuration of ROS.

5 Conclusions

In this work, we have shown that it is possible to perform data oversampling for BigData datasets on commodity hardware by combining an efficient data handling scheme and the computational capacities of GPU devices. Different settings for the methods have been tested on different datasets up to 10 millions of instances and imbalanced ratios up to 51.

Table 7 Area under the ROC classifying with decision tree

	Original	SMOTE-GPU	ROS	SMOTE-GPU + 50%	ROS + 50%
ECBDL14-05mill-90	.5449	.5673	.5451	.5730	.5479
ECBDL14-10mill-90	.5636	.5942	.5641	.6059	.5642
SUSY_IR_16	.6657	.7176	.6798	.7233	.6660
SUSY_IR_4	.7116	.7347	.7122	.7355	.7116
SUSY_IR_8	.6911	.7270	.6976	.7297	.6918
HIGGS_IR_16	.5709	.5239	.5975	.6196	.5713
HIGGS_IR_4	.6288	.5279	.6359	.6376	.6294
HIGGS_IR_8	.5981	.5257	.6137	.6260	.5986
HEPMASS_IR_16	.7696	.8088	.7698	.8128	.7700
HEPMASS_IR_4	.8080	.8181	.8079	.8189	.8080
HEPMASS_IR_8	.7915	.8136	.7911	.8163	.7910

The area under the ROC curve results show that the use of oversampling methods improves the detection of the minority class in Big Data datasets. We have also shown how our design can successfully work on a wide range of devices, including a laptop, while requiring reasonable times, around 25 min on high-end devices, and less than 2 h on the laptop, for the most time-demanding experiment.

Acknowledgements This work was supported by the Spanish National Research Projects TIN2013-47210-P, TIN2014-57251-P and TIN2016-81113-R and by the Andalusian Regional Government Excellence Research Project P12-TIC-2958. P.D. Gutiérrez holds an FPI scholarship from the Spanish Ministry of Economy and Competitiveness (BES-2012-060450).

References

1. Alcalá-Fdez, J., Sánchez, L., García, S., del Jesus, M., Ventura, S., Garrell, J., Otero, J., Romero, C., Bacardit, J., Rivas, V., Fernández, J., Herrera, F.: KEEL: a software tool to assess evolutionary algorithms for data mining problems. Soft Comput. **13**(3), 307–318 (2009)
2. Bache, K., Lichman, M.: UCI machine learning repository (2013). http://archive.ics.uci.edu/ml
3. Baldi, P., Sadowski, P., Whiteson, D.: Searching for exotic particles in high-energy physics with deep learning. Nat. Commun. **5** (2014)
4. Bradley, A.P.: The use of the area under the ROC curve in the evaluation of machine learning algorithms. Pattern Recognit. **30**(7), 1145–1159 (1997)
5. Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P.: Smote: synthetic minority over-sampling technique. J. Artif. Int. Res. **16**(1), 321–357 (2002)
6. CUDA. http://www.nvidia.com/object/cuda_home_new.html. Accessed March 2017
7. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
8. ECBDL14 dataset: Protein structure prediction and contact map for the ECBDL2014 big data competition (2014). http://cruncher.ncl.ac.uk/bdcomp/
9. Fernández, A., del Río, S., Chawla, N.V., Herrera, F.: An insight into imbalanced big data classification: outcomes and challenges. Complex Intell. Syst. (in press). doi:10.1007/s40747-017-0037-9
10. Foundation, A.S.: Apache Mahout (2017). http://mahout.apache.org/. Accessed March 2017
11. Gutiérrez, P.D., Lastra, M., Bacardit, J., Benítez, J.M., Herrera, F.: GPU–SME–kNN: scalable and memory efficient *k*NN and lazy learning using GPUs. Inf. Sci. **373**, 165–182 (2016)
12. Gutiérrez, P.D., Lastra, M., Herrera, F., Benitez, J.M.: A high performance fingerprint matching system for large databases based on GPU. IEEE Trans. Inf. Forensics Secur. **9**(1), 62–71 (2014)
13. He, H., Garcia, E.A.: Learning from imbalanced data. IEEE Trans. Knowl. Data Eng. **21**(9), 1263–1284 (2009)
14. Hoare, C.A.R.: Algorithm 64: quicksort. Commun. ACM **4**(7), 321 (1961)
15. Krawczyk, B.: Learning from imbalanced data: open challenges and future directions. Progr. Artif. Intell. **5**(4), 221–232 (2016)
16. López, V., Fernández, A., García, S., Palade, V., Herrera, F.: An insight into classification with imbalanced data: Empirical results and current trends on using data intrinsic characteristics. Inf. Sci. **250**, 113–141 (2013)
17. Madden, S.: From databases to big data. IEEE Internet Comput. **16**(3), 4–6 (2012)
18. Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al.: MLLIB: machine learning in apache spark. J. Mach. Learn. Res. **17**(34), 1–7 (2016)
19. Owen, S., Anil, R., Dunning, T., Friedman, E.: Mahout in Action, Manning Publications Co., Greenwich, CT, USA, ISBN:1935182684, 9781935182689 (2011)
20. Prati, R.C., Batista, G.E.A.P.A., Silva, D.F.: Class imbalance revisited: a new experimental setup to assess the performance of treatment methods. Knowl. Inf. Syst. **45**(1), 247–270 (2015)
21. Rajaraman, A., Ullman, J.: Mining of Massive Datasets. Cambridge University Press, Cambridge (2011)
22. Salomon-Ferrer, R., Götz, A., Poole, D., Le Grand, S., Walker, R.: Routine microsecond molecular dynamics simulations with amber on GPUS. 2. Explicit solvent particle mesh ewald. J. Chem. Theory Comput. **9**(9), 3878–3888 (2013)
23. Spark, A.: Machine Learning Library (MLlib) for Spark (2017). http://spark.apache.org/docs/latest/mllib-guide.html. Accessed March 2017
24. Triguero, I., del Río, S., López, V., Bacardit, J., Benítez, J.M., Herrera, F.: ROSEFW-RF: the winner algorithm for the ECBDL'14 big data competition—an extremely imbalanced big data bioinformatics problem. Knowl. Based Syst. **87**, 69–79 (2015)
25. White, T.: Hadoop: The Definitive Guide, 4th edn. O'Reilly Media Inc, Sebastopol (2015)
26. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: 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, pp. 1–14. USENIX Association (2012)
27. Zikopoulos, P.C., Eaton, C., deRoos, D., Deutsch, T., Lapis, G.: Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data, 1st edn. McGraw-Hill, New York (2011)