SoftwareX 15 (2021) 100767

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

SOUL: Scala Oversampling and Undersampling Library for imbalance classification

Néstor Rodríguez, David López, Alberto Fernández, Salvador García [*], Francisco Herrera

DaSCI Andalusian Institute of Data Science and Computational Intelligence, University of Granada, Spain

A R T I C L E   I N F O

Article history:
Received 3 August 2019
Received in revised form 12 April 2021
Accepted 9 July 2021

Keywords:
Oversampling
Undersampling
Scala
Imbalanced classification

A B S T R A C T

The improvements in technology and computation have promoted a global adoption of Data Science. It is devoted to extracting significant knowledge from high amounts of information by means of the application of Artificial Intelligence and Machine Learning tools. Among the different tasks within Data Science, classification is probably the most widespread overall.

Focusing on the classification scenario, we often face some datasets in which the number of instances for one of the classes is much lower than that of the remaining ones. This issue is known as the imbalanced classification problem, and it is mainly related to the need for boosting the recognition of the minority class examples.

In spite of a large number of solutions that were proposed in the specialized literature to address imbalanced classification, there is a lack of open-source software that compiles the most relevant ones in an easy-to-use and scalable way. In this paper, we present a novel software approach named as SOUL, which stands for Scala Oversampling and Undersampling Library for imbalanced classification. The main capabilities of this new library include a large number of different data preprocessing techniques, efficient execution of these approaches, and a graphical environment to contrast the output for the different preprocessing solutions.

Code metadata

Current code version	1.0.0
Permanent link to code/repository used of this code version	https://github.com/ElsevierSoftwareX/SOFTX_2019_253
Legal Code License	GPL3.0
Code versioning system used	git
Software code languages, tools, and services used	scala ($\geq$ 2.12.0)
Compilation requirements, operating environments & dependencies	sbt ($\geq$ 1.2.3)
If available Link to developer documentation/manual	https://soul-doc.github.io/soul/index.html
Support email for questions	nestorrodriguezvico@gmail.com derwey@correo.ugr.es

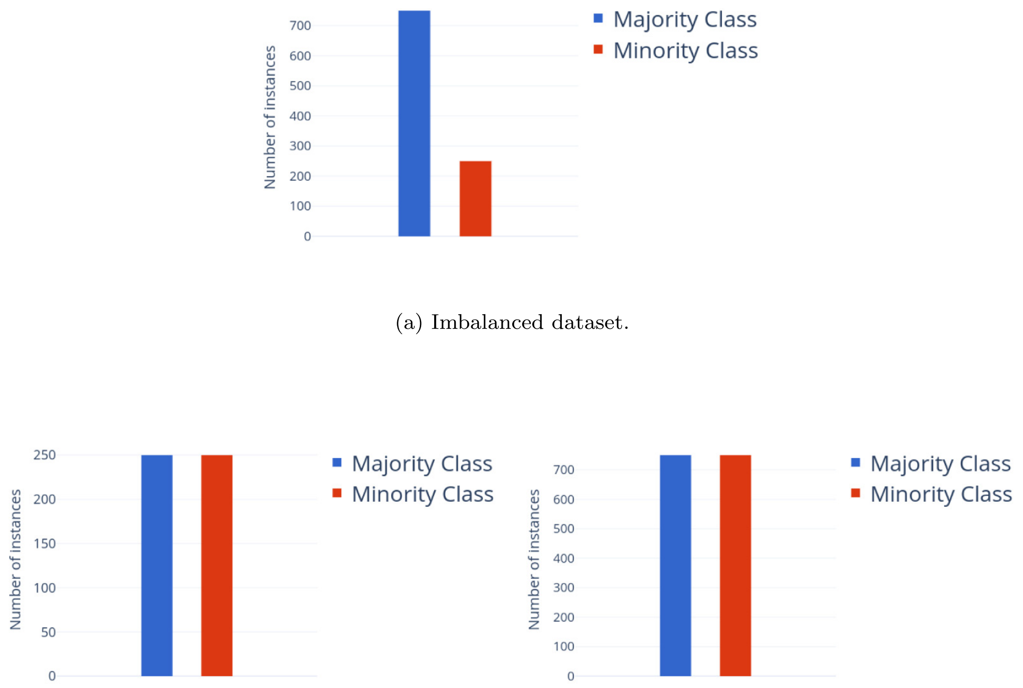
1. Introduction

In current applications of Data Science and Machine Learning (ML), the problem of classification can be viewed as one of the major areas of investigation [1,2]. Different non-standard scenarios need to be faced in any classification problem [3]. Among them, when the dataset comprises a number of instances from a class that is radically different to the number of instances that belong to another class, the problem is known as the imbalanced classification [4–6]. The current relevance and implications of this area of research in classification are beyond all doubt [7].

The main characteristic of the imbalanced problem is the bias of standard ML techniques towards the majority class instances. To solve this issue we may proceed it in two different ways, namely algorithm level techniques and data level techniques [8]. Algorithm level techniques aim for modifying the ML algorithms to take into account the skewed data distribution and/or to boost the recognition of the minority class instances. Data level techniques aim for directly modifying the dataset prior to the learning stage to create a balanced one. Regarding this type of approach, we further distinguish between two categories. On the one hand, oversampling algorithms that replicate and/or generate minority

[*] Corresponding author.
E-mail address: salvagl@decsai.ugr.es (Salvador García).

https://doi.org/10.1016/j.softx.2021.100767



(a) Imbalanced dataset.

(b) Result of applying undersampling techniques.

(c) Result of applying oversampling techniques.

Fig. 1. Example of running undersampling and oversampling techniques.

class instances. On the other hand, undersampling algorithms that remove instances mainly from the majority class [9]. Removing instances of the majority class may be a good idea to avoid possible noise, which may worsen the performance of the classifier [10]. In addition, feature selection is an approach that could help the techniques described before identifying discriminative features and reducing computational time [11].

Fig. 1 shows an illustrative representation of the result of applying oversampling and undersampling techniques. Both techniques balance the data, despite working differently. Fig. 1(a) shows the distribution of an example of the imbalanced distribution of classes in a given dataset. Fig. 1(b) shows the outcome of an undersampling method, where the majority class is reduced until 250 instances to balance the data. In contrast, oversampling generates instances labeled as minority class, expanding the former until 750 instances, balancing the data as well, as we can see in Fig. 1(c).

It is well-known that applying oversampling or undersampling before running a classification algorithm enhance its performance [12]. In addition, these techniques can be part of the learning process within an ensemble [13], extending its role beyond just pre-processing the data.

In the specialized literature, we may find a large number of proposals on oversampling and undersampling algorithms. Unfortunately, there is not so much software available that implements any of these techniques. The most known open source libraries that include algorithms for imbalanced classification are `KEEL` [14] library for Java, `imbalanced-learn` [15] for Python and `unbalanced` [16], `smotefamily` [17], `rose` [18] and `imbalance` [19] for R. These libraries have a large set of included algorithms for undersampling and oversampling techniques to address the imbalance problem from a data level perspective. However, there are three main issues associated with these software solutions:

1. Among the different software solutions, only `imbalanced-learn` allows for a direct representation of preprocessed datasets.
2. Existing software focuses only on oversampling approaches or may include few approaches of undersampling techniques, but they do not offer a good repository of both kind of techniques.
3. Regarding scalability and efficiency, Scala is known to be usually faster than both Python and R. One of the reasons Scala is faster is that it uses static data types, while Python or R use dynamic typing. That means that during execution, Python or R need to do additional processing to identify the types of the values in order to evaluate the expressions [20,21]. Also, the framework Spark works better in Scala [22,23]. This fact is quite relevant when we need to face large datasets as those generated in the Big Data era.

In this paper, we present the first Scala library for imbalance classification, which includes all the classic and well-established algorithms. Those algorithms work on the data level, pre-processing the dataset in order to mitigate the negative effects caused by the imbalance distribution of classes present in the data. The new dataset is ready to be used by any other algorithm from other libraries that the user considers appropriate. We named the library `SOUL`, which stands for `Scala Oversampling and Undersampling Library`. This library provides a set of almost 30 different algorithms for data preprocessing. Specifically, it includes 13 techniques for oversampling and another 15 algorithms for undersampling. It is also the first open-source library for imbalanced library purely written in Scala. Also, `SOUL` is compatible with `SMILE` [24], the biggest library for ML in Scala.

To provide a clear picture of `SOUL`, the rest of this paper is divided into the following sections. First, Section 2 presents the software framework. Then, Section 3 describes the implemented algorithms. In Section 4 we present a use case of this software. After that, in Section 5 we perform two different experiments to compare this library with the currently available software. Finally, Section 6 presents the conclusions.

2. SOUL: A Scala library for imbalanced preprocessing

The SOUL library is implemented in Scala [20], a powerful language built on top of the Java Virtual Machine (JVM), so it is compatible with any system that can run Java. Scala provides a mix design pattern, combining functional with object-oriented programming. Scala is as simple as powerful, allowing users to create from a simple script to a huge project, with a few lines of code. As Scala is compiled by the JVM, it is faster than R and can get similar execution times than Python, which is compiled to C. Scala is statically typed, meanwhile, Python and R are dynamically typed, which allows Scala less prone to errors. Also, Scala provides some parallels collection, using the advantage of the functional programming, allowing the programmer to implement parallel operations without any extra effort. Parallelism is important due to the amount of data used in classification problems. Scala is easily integrated with Spark [25], an Apache framework used for Big Data.

A clear positive feature of this library in comparison to similar solutions is the large number of algorithms that are implemented. As commented previously, up to 30 different approaches have been developed within SOUL, divided between oversampling and undersampling. In particular, there are a number of 13 oversampling algorithms and another 15 undersampling algorithms. As commented before, it almost duplicates the number of approaches in the `imbalanced-learn` library (16 algorithms), and 3 times more than `unbalanced`. In addition to the former, another significant issue for this library is to support the use of different distance metrics when we run the algorithms, such as the Heterogeneous Value Difference Metric [26]. This is of high importance as the behavior of the preprocessing algorithms is very dependent for this characteristic.

It is noteworthy to mention the possibility of a complete parameterization by the algorithms included in this library, allowing the end-user to adapt the execution of the algorithms to their needs. In order not to overload the user's work, all the parameters have a default value, thus allowing a quick execution of the algorithms without having to understand what each parameter does and which would be a correct value for it, as will be shown in Section 4.

It is important to point out that we have our own input/output system, which makes the library easier to use. The supported formats are CSV, any delimited text data file and ARFF, which are the most common formats in data science applications.

In Code metadata table, we can see the most recent version metadata of the project.

3. Preprocessing algorithms included in SOUL

As pointed out throughout this paper, SOUL imbalanced library supports a large number of different preprocessing approaches. Next, in Table 1 all the included algorithms are enumerated.

4. User manual for SOUL library

In order to use SOUL library from another `sbt` project, we just need to clone the repository of the project, in the root folder of the cloned repository execute `sbt publishLocal` and add the following dependency to the `build.sbt` file of our project:

```
libraryDependencies += "com.github.soul"
```

Listing 1: sbt dependency.

Table 1
Enumeration of algorithms included. In brackets, the name of the algorithm in the code.

Oversampling	Undersampling
`Random Oversampling (RO)` [27]	`Random Undersampling (RU)` [27]
`SMOTE (SMOTE)` [28,29]	`Condensed Nearest Neighbor (CNN)` [30]
`SMOTE + Edited Nearest Neighbor rule (SMOTEENN)` [27]	`Edited Nearest Neighbor rule (ENN)` [31]
`SMOTE + Tomek Link (SMOTETL)` [27]	`Tomek Link (TL)` [32]
`Borderline-SMOTE (BorderlineSMOTE)` [33]	`One-Side Selection (OSS)` [34]
`ADASYN (ADASYN)` [35]	`Neighborhood Cleaning Rule (NCL)` [36]
`ADOMS (ADOMS)` [37]	`NearMiss (NM)` [38]
`SafeLevel-SMOTE (SafeLevelSMOTE)` [39]	`Class Purity Maximization (CPM)` [40]
`Spider2 (Spider2)` [41]	`Undersampling Based on Clustering (SBC)` [42]
`DBSMOTE (DBSMOTE)` [43]	`Balance Cascade (BC)` [44]
`SMOTE-RSB* (SMOTERSB)` [45]	`Easy Ensemble (EE)` [44]
`MWMOTE (MWMOTE)` [46]	`Evolutionary Undersampling (EUS)` [47]
`MDO (MDO)` [48]	`Instance Hardness Threshold (IHTS)` [49]
	`ClusterOSS (clusterOSS)` [50]
	`Iterative Instance Adjustment for Imbalanced Domains (IPADE)` [51]

To read a data file we only need to use the code from the Listing 2. We just need to import the `Reader` class (line 1), import the `Data` class (line 2) and read the required file in the format we want (lines 5 and 7).

```
1 import soul.io.Reader
2 import soul.data.Data
3
4 /* Read a csv file or any delimited text file */
5 val csvData: Data = Reader.readDelimitedText(file = <
      pathToFile>)
6 /* Read a WEKA arff file */
7 val arffData: Data = Reader.readArff(file = <pathToFile>)
```

Listing 2: Read data from a file.

Listing 3 shows the steps to run the chosen algorithm. We have to import the algorithm class we want to use (line 1) and instantiate the algorithm using the data we have read in Listing 2 (lines 4 and 7). After that, we only need to call the `compute` method (lines 5 and 8). If we provide a file name representing where to store the log, the execution of the algorithm will save the information into this file (argument passed to the constructor of algorithm's class in lines 4 and 7). In the example presented in Listing 3 we have used an undersampling algorithm but it is the same for an oversampling one. As commented in Section 2, all the algorithm's parameters have a default value so we do not need to specify any of them to run an algorithm, but we can modify them to fit our needs.

```
1 import soul.algorithm.undersampling.NCL
2 import soul.data.Data
3
4 val nclCSV = new NCL(csvData)
5 val resultCSV: Data = nclCSV.compute()
6
7 val nclARFF = new NCL(arffData)
8 val resultARFF: Data = nclARFF.compute()
```

Listing 3: Apply the algorithm.

Finally, we only need to save the result to a file using the code shown in Listing 4. Import the `Writer` class (line 1) and, based on the format we want to use as output, call the corresponding function (lines 3 and 4), specifying the output file and the data we want to save.

```
1 import soul.io.Writer
2
3 Writer.writeDelimitedText(file = <pathToFile>, data =
      resultCSV)
4 Writer.writeArff(file = <pathToFile>, data = resultARFF)
```

Listing 4: Save result to a file.

For further information about the parameters and available algorithms, please refer to the documentation website at: https://soul-doc.github.io/soul/index.html.

5. SOUL use case: Experimental results

With the objective of showing the capabilities of `SOUL`, we have generated a two-dimension synthetic imbalanced dataset with 1,871 instances. Among them, 1,600 instances belong to the majority class and the remaining 271 belongs to the minority class, leading to about a 17% of minority instances in the whole dataset (Imbalance Ratio, IR=5.9). The representation of this dataset can be found in Fig. 2, where we may observe a clear overlapping between the classes, as well as a cluster of minority instances in the middle of the majority instances.

Next, we have applied the following preprocessing algorithms, together with the configuration specified:

- `MWMOTE`: *seed*: 0, *N*: 1400, *k1*: 5, *k2*: 5, *k3*: 5, *dist*: euclidean, *normalize*: false, *verbose*: false.
- `SMOTE`: *seed*: 0, *percent*: 500, *k*: 5, *dist*: euclidean, *normalize*: false, *verbose*: false.
- `ADASYN`: *seed*: 0, *d*: 1, *B*: 1, *k*: 5, *dist*: euclidean, *normalize*: false, *verbose*: false.
- `SafeLevelSMOTE`: *seed*: 0, *k*: 5, *dist*: euclidean, *normalize*: false, *verbose*: false.
- `IHTS`: *seed* = 0, *nFolds* = 5, *normalize* = false, *randomData* = false, *verbose* = false
- `IPADE`: *seed* = 0, *iterations* = 100, *strategy* = 1, *randomChoice* = true, *normalize* = false, *randomData* = false, *verbose* = false
- `NCL`: *seed* = 0, *dist* = euclidean, *k* = 3, *threshold* = 0.5, *normalize* = false, *randomData* = false, *verbose* = false
- `SBC`: *seed* = 0, *method* = "NearMiss1", *m* = 1.0, *k* = 3, *numClusters* = 50, *restarts* = 1, *minDispersion* = 0.0001, *maxIterations* = 200, val *dist* = euclidean, *normalize* = false, *randomData* = false, *verbose* = false

As commented above, we intend to visually contrast the differences regarding the output of the different alternatives.

Fig. 3 shows the output of four oversampling algorithms. Fig. 3(d) shows that `SMOTE` can generate a large number of synthetic samples in the minority class space. `SafeLevelSMOTE`, in Fig. 3(b), works on safe zones and create one synthetic sample per minority class sample at most. Hence, we can observe less synthetic samples created compared to `SMOTE`. The original `SafeLevelSMOTE` does not have a parameter to choose the number of synthetic samples to be generated. `ADASYN` and `MWMOTE`, whose behavior is depicted in Figs. 3(a) and 3(c) work on borderline zones. `ADASYN` creates synthetic samples based on the number of majority class samples in the neighborhood of each minority class sample. The larger the number of majority class examples the dataset has, the more synthetic samples are created. Hence, we can see a large number of synthetic samples in borderline. `MWMOTE` generates a lesser amount of synthetic samples because it only works with the minority class samples that are adjacent to majority class samples. Furthermore, it only creates one synthetic sample for every set of samples, so that the amount of synthetic samples created is smaller.

Fig. 4 shows the output of four undersampling algorithms. `IHTS`, Fig. 4(a), shows that it is not always necessary to use *k*-NN rules in undersampling algorithms to get good results. `IPADE`, Fig. 4(b), achieves impressive results maintaining the essential distribution of the original dataset. `IPADE` uses a representation of evolutionary techniques applied to the problem described in this paper. `IPADE` follows an iterative scheme, where it determines the most appropriate number of instances per class and their positioning for a determined classifier, focusing on the positive class. `NCL` was initially selected as a classic algorithm, however, does not provide remarkable results, as we can see in Fig. 4(c). Finally, `SBC`, represents the clustering techniques, as we can see in Fig. 4(d), it does not reduce drastically the number of majority instances but it is capable of separating the borders between the two classes, which is an outstanding property of these techniques.

To compare the execution time of the library, we created another synthetic dataset with five dimensions and one million instances. Then, we iteratively reduced the dataset by a ten per cent until we get a set with less than a thousand instances. In this way, we have generated 67 synthetic datasets with sizes ranging from 950 instances to one million instances. We have executed our `NCL` and `SMOTE` implementations versus the ones in `imbalanced (Python)` and `unbalanced (R)` libraries. In Figs. 5 and 6 the results of `NCL` and `SMOTE` algorithms are depicted respectively.

We have to be careful when we look at the graphics as they are in logarithmic scale. That is why in Fig. 6 it seems that `SOUL (Scala)` takes half the time of `unbalanced (R)`, but in the tables of times, we can see that the difference is quite significant.

As can be seen in Fig. 6, `SOUL (Scala)` implementation is slower than `imbalanced (Python)`. `SOUL (Scala)` implementation computes a new random gap for every attribute when a synthetic sample is generated, as described in the original proposal [28,29]. However, `imbalanced (Python)` implementation computes the random gap just once, therefore it run faster[1]. In the case of `NCL`, as we can see in Fig. 5, `SOUL (Scala)` takes practically the same as `imbalanced (Python)`, surpassing it in certain datasets, and also being much faster than `unbalanced (R)`.

6. Conclusion

The imbalanced classification scenario is present in many current applications. Therefore, researchers and practitioners need to address the problem with the solutions available at hand. However, the number of open-source libraries that comprise preprocessing algorithms is not so high as expected.

In this paper, we have addressed this problem of the lack of solutions in the field of classification with imbalanced datasets by proposing a novel, complete, scalable, and easy-to-use software written in Scala. We have stressed its characteristics and capabilities, possibly being the most significant one large number of preprocessing techniques that have been included (about 30 different solutions).

Along with the good features included in this first version of the library, in the near future, our efforts will be oriented towards the addition of novel preprocessing algorithms, and its extension to Spark to cope with Big Data problems.

[1] The implementation (which can be found in https://github.com/scikit-learn-contrib/imbalanced-learn/blob/master/imblearn/over_sampling/_smote/base.py) shows that the mentioned gap is always 1. Therefore, the Python implementation avoid to compute this gap in every iteration, so that, it run faster.

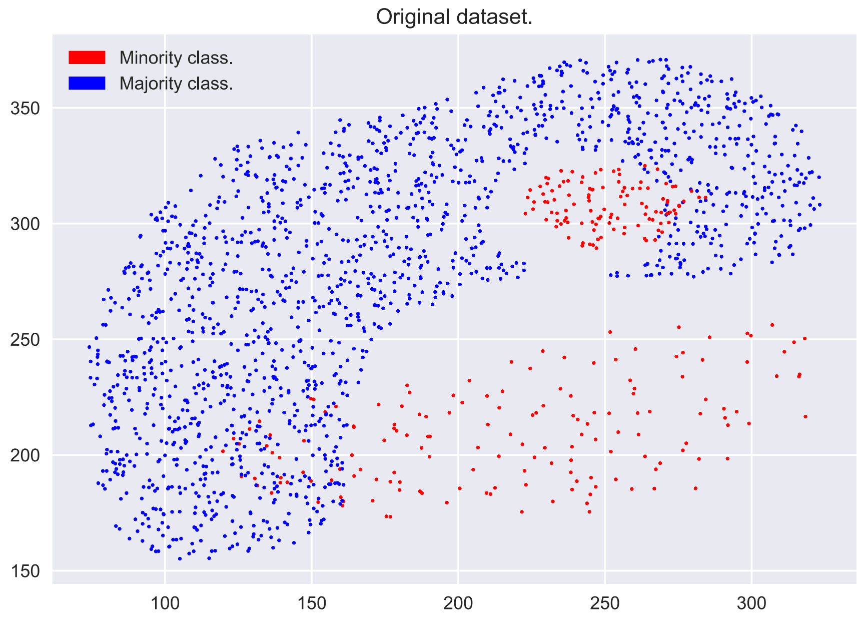
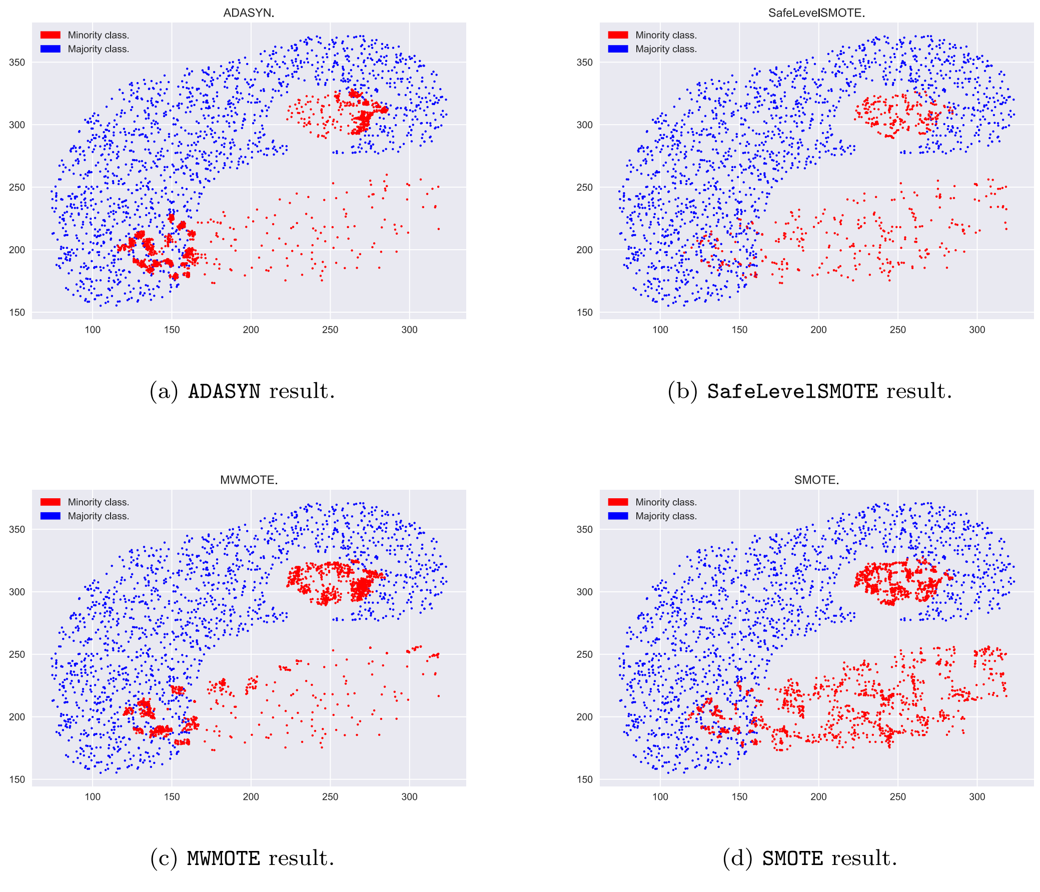


Fig. 2. Original toy binary imbalanced dataset.



(a) `ADASYN` result.

(b) `SafeLevelSMOTE` result.

(c) `MWMOTE` result.

(d) `SMOTE` result.

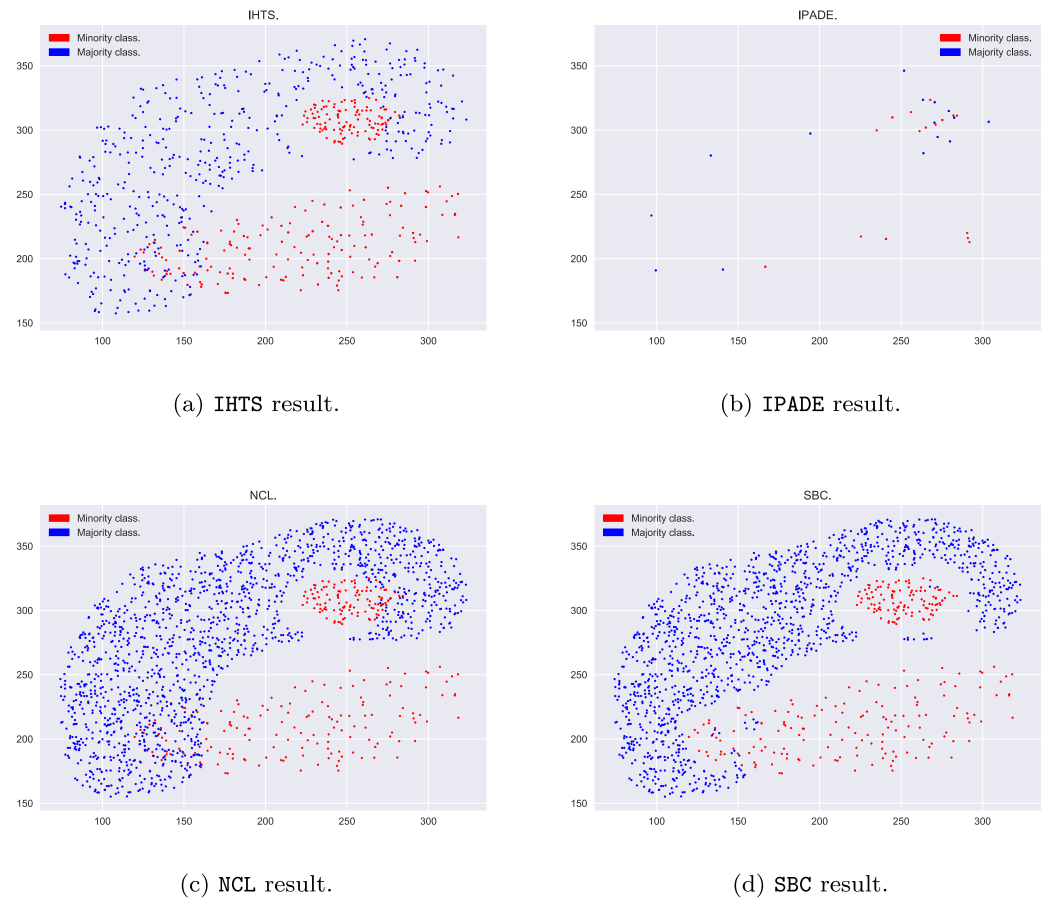
Fig. 3. Oversampling algorithms.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work has been supported by the research project TIN2017-89517-P, by the UGR research contract OTRI 3940 and by a research scholarship, given to the authors Néstor Rodríguez and David López by the University of Granada, Spain.



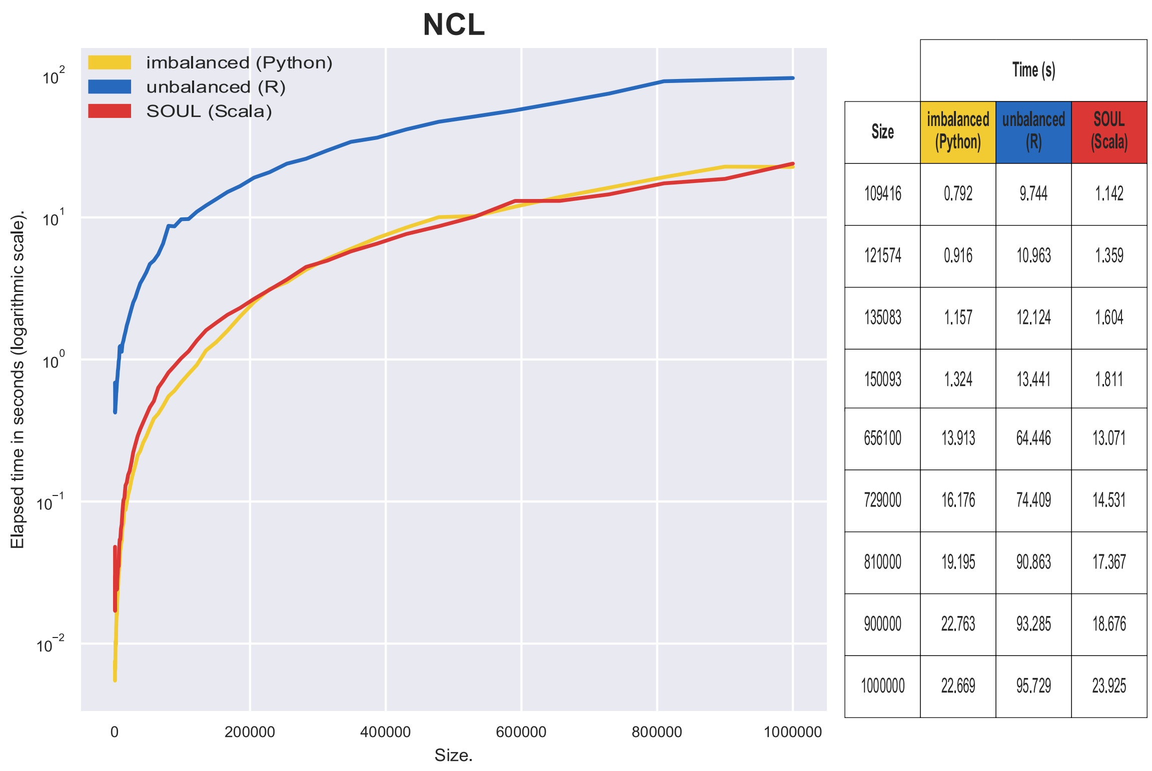
(a) IHTS result.

(b) IPADE result.

(c) NCL result.

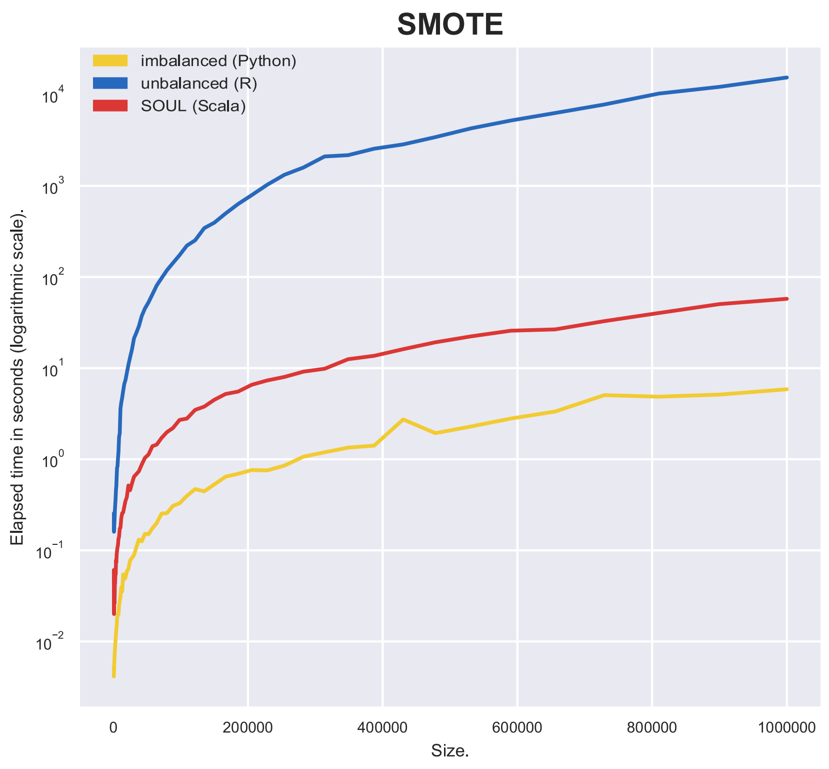
(d) SBC result.

Fig. 4. Undersampling algorithms.



Size	imbalanced (Python)	unbalanced (R)	SOUL (Scala)
109416	0.792	9.744	1.142
121574	0.916	10.963	1.359
135083	1.157	12.124	1.604
150093	1.324	13.441	1.811
656100	13.913	64.446	13.071
729000	16.176	74.409	14.531
810000	19.195	90.863	17.367
900000	22.763	93.285	18.676
1000000	22.669	95.729	23.925

Fig. 5. Comparison of execution times of NCL for imbalanced (Python), unbalanced (R) and SOUL (Scala).



	Time (s)		
Size	imbalanced (Python)	unbalanced (R)	SOUL (Scala)
109416	0.396	221.322	2.795
121574	0.471	252.682	3.481
135083	0.444	343.969	3.780
150093	0.529	394.962	4.492
656100	3.340	6312.080	26.613
729000	5.053	7823.772	32.784
810000	4.858	10314.092	40.276
900000	5.124	12218.961	50.436
1000000	5.861	15479.011	57.671

Fig. 6. Comparison of execution times of `SMOTE` for `imbalanced (Python)`, `unbalanced (R)` and `SOUL (Scala)`.

References

[1] Aggarwal CC. Data mining: The textbook. Springer; 2015.
[2] Emre Yetgin O, Gerek O, extraction Feature. Feature extraction selection and classification code for power line scene recognition. SoftwareX 2018;8:43–7.
[3] Charte D, Charte F, García S, Herrera F. A snapshot on nonstandard supervised learning problems: taxonomy, relationships, problem transformations and algorithm adaptations. Progr Artif Intell 2019;8(1):1–14.
[4] He H, Garcia EA. Learning from imbalanced data. IEEE Trans Knowl Data Eng 2009;(9):1263–84.
[5] Krawczyk B. Learning from imbalanced data: open challenges and future directions. Progr Artif Intell 2016;5(4):221–32.
[6] Roy A, Cruz RM, Sabourin R, Cavalcanti GD. A study on combining dynamic selection and data preprocessing for imbalance learning. Neurocomputing 2018;286(C):179–92.
[7] Fernández A, García S, Galar M, Prati R, Krawczyk B, Herrera F. Learning from imbalanced data sets. Springer; 2018.
[8] 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. Inform Sci 2013;250:113–41.
[9] Prati RC, Batista GE, Silva DF. Class imbalance revisited: a new experimental setup to assess the performance of treatment methods. Knowl Inf Syst 2015;45(1):247–70.
[10] Kang Q, Chen X, Li S, Zhou M. A noise-filtered under-sampling scheme for imbalanced classification. IEEE Trans Cybern 2017;47(12):4263–74. https://doi.org/10.1109/TCYB.2016.2606104.
[11] Liu H, Zhou M, Liu Q. An embedded feature selection method for imbalanced data classification. IEEE/CAA J Autom Sin 2019;6(3):703–15. https://doi.org/10.1109/JAS.2019.1911447.
[12] Kang Q, Shi L, Zhou M, Wang X, Wu Q, Wei Z. A distance-based weighted undersampling scheme for support vector machines and its application to imbalanced classification. IEEE Trans Neural Netw Learn Syst 2018;29(9):4152–65. https://doi.org/10.1109/TNNLS.2017.2755595.
[13] Tang W, Ding Z, Zhou M. A spammer identification method for class imbalanced weibo datasets. IEEE Access 2019;7:29193–201. https://doi.org/10.1109/ACCESS.2019.2901756.
[14] Triguero I, González S, Moyano JM, García S, Alcalá-Fdez J, Luengo J, et al. KEEL 3.0: an open source software for multi-stage analysis in data mining. Int J Comput Intell Syst 2017;10(1):1238–49.
[15] Lemaître G, Nogueira F, Aridas CK. Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. J Mach Learn Res 2017;18(1):559–63.
[16] Dal Pozzolo A, Caelen O, Waterschoot S, Bontempi G. Racing for unbalanced methods selection. In: International conference on intelligent data engineering and automated learning. Springer; 2013, p. 24–31.
[17] Siriseriwan W. smotefamily: A Collection of Oversampling Techniques for Class Imbalance Problem Based on SMOTE. 2016, https://cran.r-project.org/web/packages/smotefamily/index.html.
[18] Lunardon N, Menardi G, Torelli N. ROSE: A package for binary imbalanced learning. R J 2014;6(1). https://cran.r-project.org/web/packages/ROSE/index.html.
[19] Cordón I, García S, Fernández A, Herrera F. Imbalance: Oversampling algorithms for imbalanced classification in R. Knowl-Based Syst 2018;161:329–41.
[20] Alexander A. Scala cookbook: Recipes for object-oriented and functional programming. 1st ed. O'Reilly Media, Inc.; 2013.
[21] Beazley D, Jones BK. Python cookbook: Recipes for mastering Python 3. O'Reilly Media, Inc.; 2013.
[22] Armbrust M, Das T, Davidson A, Ghodsi A, Or A, Rosen J, et al. Scaling spark in the real world: performance and usability. Proc VLDB Endowment 2015;8(12):1840–3.
[23] Divya Sistla, Scala vs. Python for Apache Spark https://www.dezyre.com/article/scala-vs-python-for-apache-spark/213.
[24] Li H. Smile - Statistical Machine Intelligence and Learning Engine. URL http://haifengl.github.io/smile/.
[25] Zaharia M, Xin RS, Wendell P, Das T, Armbrust M, Dave A, et al. Apache spark: A unified engine for big data processing. Commun ACM 2016;59(11):56–65.
[26] Wilson DR, Martinez TR. Improved heterogeneous distance functions. J Artificial Intelligence Res 1997;6(1):1–34.
[27] Batista GE, Prati RC, Monard MC. A study of the behavior of several methods for balancing machine learning training data. ACM SIGKDD Explor Newsl 2004;6(1):20–9.
[28] Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP. SMOTE: synthetic minority over-sampling technique. J Artificial Intelligence Res 2002;16:321–57.
[29] Fernández A, García S, Herrera F, Chawla NV. SMOTE for learning from imbalanced data: Progress and challenges, marking the 15-year anniversary. J Artificial Intelligence Res 2018;61:863–905.
[30] Hart P. The condensed nearest neighbor rule (Corresp.). IEEE Trans Inform Theory 1968;14(3):515–6.
[31] Wilson DL. Asymptotic properties of nearest neighbor rules using edited data. IEEE Trans Syst Man Cybern 1972;(3):408–21.
[32] Tomek I. Two modifications of CNN. IEEE Trans Syst Man Cybern 1976;6:769–72.

[33] Han H, Wang W-Y, Mao B-H. Borderline-SMOTE: a new over-sampling method in imbalanced data sets learning. In: International conference on intelligent computing. Springer; 2005, p. 878–87.
[34] Kubat M, Matwin S, et al. Addressing the curse of imbalanced training sets: one-sided selection. In: Icml, vol. 97. Nashville, USA; 1997. p. 179–86.
[35] He H, Bai Y, Garcia EA, Li S. ADASYN: Adaptive synthetic sampling approach for imbalanced learning. In: Neural networks, 2008. IJCNN 2008.(IEEE world congress on computational intelligence). IEEE international joint conference on. IEEE; 2008, p. 1322–8.
[36] Laurikkala J. Improving identification of difficult small classes by balancing class distribution. In: Conference on artificial intelligence in medicine in Europe. Springer; 2001, p. 63–6.
[37] Tang S, Chen S-P. The generation mechanism of synthetic minority class examples. Inf Technol Appl Biomed 2008;444–7.
[38] Mani I, Zhang I. *k*NN approach to unbalanced data distributions: a case study involving information extraction. In: Proceedings of workshop on learning from imbalanced datasets, vol. 126; 2003.
[39] Bunkhumpornpat C, Sinapiromsaran K, Lursinsap C. Safe-level-smote: Safe-level-synthetic minority over-sampling technique for handling the class imbalanced problem. In: Pacific-Asia conference on knowledge discovery and data mining. Springer; 2009, p. 475–82.
[40] Yoon K, Kwek S. An unsupervised learning approach to resolving the data imbalanced issue in supervised learning problems in functional genomics. In: Hybrid intelligent systems, 2005. HIS'05. Fifth international conference on. IEEE; 2005, p. 6.
[41] Napierała K, Stefanowski J, Wilk S. Learning from imbalanced data in presence of noisy and borderline examples. In: International conference on rough sets and current trends in computing. Springer; 2010, p. 158–67.
[42] Yen S-J, Lee Y-S. Under-sampling approaches for improving prediction of the minority class in an imbalanced dataset. In: Intelligent control and automation. Springer; 2006, p. 731–40.
[43] Bunkhumpornpat C, Sinapiromsaran K, Lursinsap C. DBSMOTE: density-based synthetic minority over-sampling technique. Appl Intell 2012;36(3):664–84.
[44] Liu X-Y, Wu J, Zhou Z-H. Exploratory undersampling for class-imbalance learning. IEEE Trans Syst Man Cybern B 2009;39(2):539–50.
[45] Ramentol E, Caballero Y, Bello R, Herrera F. SMOTE-RSB*: a hybrid preprocessing approach based on oversampling and undersampling for high imbalanced data-sets using SMOTE and rough sets theory. Knowl Inf Syst 2012;33(2):245–65.
[46] Barua S, Islam MM, Yao X, Murase K. MWMOTE–majority weighted minority oversampling technique for imbalanced data set learning. IEEE Trans Knowl Data Eng 2014;26(2):405–25.
[47] García S, Herrera F. Evolutionary undersampling for classification with imbalanced datasets: Proposals and taxonomy. Evol Comput 2009;17(3):275–306.
[48] Abdi L, Hashemi S. To combat multi-class imbalanced problems by means of over-sampling techniques. IEEE Trans Knowl Data Eng 2016;28(1):238–51.
[49] Smith MR. An empirical study of instance hardness. Tech. rep., Brigham Young University-Provo; 2009.
[50] Barella VH, Costa EdP, Carvalho ACPdL, et al. ClusterOSS: a new undersampling method for imbalanced learning. In: Brazilian conference on intelligent systems, 3th; encontro nacional de inteligência artificial E computacional, 11th. Universidade de São Paulo-USP; 2014.
[51] López V, Triguero I, Carmona CJ, García S, Herrera F. Addressing imbalanced classification with instance generation techniques: IPADE-ID. Neurocomputing 2014;126:15–28.