Evolutionary Induction of Classification Trees on Spark

Daniel Reska, Krzysztof Jurczuk(✉), and Marek Kretowski

Faculty of Computer Science, Bialystok University of Technology,
Wiejska 45a, 15-351 Bialystok, Poland
{d.reska,k.jurczuk,m.kretowski}@pb.edu.pl

Abstract. Evolutionary-based approaches have recently been increasingly proposed for data mining tasks, but their real applicability depends on efficiency and scalability for large-scale data. It is clear that parallel and distributed processing support is indispensable herein. Apache Spark is one of the most promising cluster-computing engines for Big Data. In this paper, we investigate the application of Spark to speed up an evolutionary induction of classification trees in the Global Decision Tree (GDT) system. The system simultaneously searches for the tree structure and tests in non-terminal nodes due to specialized genetic operators. As the original GDT system is implemented in C++, the Java-based module is developed for Spark-based acceleration of the most computationally demanding fitness evaluation. The training dataset is transformed to Resilient Distributed Dataset, which enables in-memory processing of dataset's parts on workers. Preliminary experimental validation on large-scale artificial and real-life datasets shows that the proposed solution is efficient and scales well.

Keywords: Decision tree · Evolutionary algorithms · Spark
Distributed computing · Data mining · Large-scale data

1 Introduction

In the last decade, the availability of large data volumes in business, industry and research increased tremendously. This gives huge opportunities as well as challenges for knowledge discovery from data [20]. Typical data mining approaches, originating from classical statistical pattern recognition or machine learning algorithms, are usually computationally complex. They become often useless when confronted with contemporary Big Data warehouses. One of the possible solutions is adaptation and extension of existing methods by using parallel and/or distributed processing [12]. Nowadays a lot of researchers commitment is directed at developing flexible frameworks for migrating calculation to computing clusters or graphical processing units (GPU)s, as former low-level approaches like e.g. MPI were relatively demanding to apply. Spark [21] is a novel alternative for cluster-computing, especially well suited for in-memory computing. It offers fault-tolerance, robust horizontal scalability as well as rich monitoring and diagnostic tools.

Evolutionary algorithms [16] are currently one of the most popular population-based meta-heuristics for solving optimization and search problems. These nature-inspired techniques try to mimic biological evolution, where more fitted individuals have a better chance to survive and reproduce. Evolutionary algorithms are known for their robustness and are successfully applied to a wide range of problems. On the other hand, these are not the fastest techniques when run on a single processor, so especially for the big data mining, efficient parallel and/or distributed implementations and accelerations are indispensable [2,11].

In this paper, we investigate speeding up an evolutionary induction of decision trees. Global Decision Tree (GDT) is a data mining system that enables induction of various variants of decision trees: univariate, oblique and mixed ones. In a single run of an evolutionary algorithm, the tree structure, tests and predictions in the leaves are searched. The resulting decision trees are generally simple and accurate, but for large-scale data computation times are long [3]. To overcome this limitation, we tried to develop more time efficient implementations on computing clusters using MPI/OpenMP [5,6] and on GPUs [13] using CUDA. In this paper, we explore the benefits of Spark as an acceleration engine for the GDT system.

The rest of the paper is organized as follow. Firstly, Spark is briefly introduced and its applications in an evolutionary search and especially evolutionary data mining are listed. In the next section, the most important features of global induction of decision trees are shortly recalled. Then Spark-based acceleration of the GDT system is presented and vital implementation issues are also shortly discussed. In Sect. 4 experimental validation of the proposed solution on large-scale artificial and real datasets is described. The paper is concluded in the last section and possible directions of future works are sketched.

1.1 Spark

Apache Spark [21] is an open-source distributed computing engine for large-scale data processing. Spark provides high-level APIs in Java, Scala, Python and R and offers tools for structured data processing, machine learning, graph processing and data streaming.

Apache Spark architecture is based on a concept of Resilient Distributed Dataset (RDD) - an immutable distributed data structure that provides fault tolerance and can be processed in parallel.

In contrast to Hadoop MapReduce, where intermediate results are stored on disk, Spark processes data in distributed shared memory model, preferably in the RAM of the cluster nodes, which makes it much more suited for iterative algorithms and interactive data exploration. Furthermore, Spark offers a much broader set of high-level functional-style data operators that simplify the implementation of distributed applications.

1.2 Related Works

One of the first attempts to apply Spark to evolutionary algorithms was proposed by Deng et al. [8]. The authors implemented a parallel version of differential evolution where the population is treated as an RDD and only the fitness evaluation is distributed to workers. Teijero et al. [19] also tried to parallelize differential evolution, focusing on individual's mutation. The proposed three master-slave implementations were not efficient, so the authors decided to switch to the island model with local-range mutations and rare migrations. In [18] a parallel genetic algorithm for pairwise test suite generation was proposed. A population was stored as an RDD and the fitness was evaluated on workers, whereas genetic operators were applied in subpopulation corresponding to partitions.

As for evolutionary data mining approaches using Spark, in [9] fuzzy rule-based classifiers were generated. The fitness function of multi-objective algorithm scanned for entire training datasets and this computationally expensive operation was divided among cluster nodes. Another multi-objective fuzzy approach for subgroup discovery was presented in [17]. Evolutionary search for a set of rules was executed in separate dataset partitions and repeated for each value of the target variable. Then reduction of the rules obtained in each partition based on the global measures was carried out. In [10] the authors tried to scale a genetic programming-based solution for symbolic regression and proposed a fitness evaluation service based on Spark.

2 Global Decision Tree Induction Framework

The GDT system [7,14] enables induction of several types of decision trees, depending among others on the type of a predictive task to be solved, the permitted test types in nodes and models in leaves, etc. All variants of the algorithm share the same typical evolutionary process [16] with an unstructured, fixed size population and a generational selection.

Every individual in a population corresponds to a single decision tree, whose size and structure is dynamically changed during the evolution. There is no special encoding and the trees are represented in the actual form. In the simplest case of binary, univariate, classification trees, only inequality tests on continuous-valued features with two outcomes are placed in the internal nodes, whereas class labels are associated with leaves. More complicated variants can, for example, rely on oblique tests in non-terminal nodes or multiple-linear models in leaves.

Individuals in an initial population are created with a simplified top-down induction, which is applied to randomly selected small sub-samples of the learning data.

As decision trees are "peculiar", hierarchical structures, the variation operators should be appropriately designed and applied so that the evolution is efficient and robust. In the GDT system, two groups of specialized genetic operators were developed: mutation-like operators which are applied to single individuals and cross-over ones which operate on pairs of trees. In each group, several variants of operators were proposed depending, among others, on the tree type and node type. Each time, the choices of the variant and affected nodes (or node) are random, but probability distributions are not uniform across both nodes and variants. For nodes the location in the tree (modifying nodes from upper levels result in more global changes) and quality of the subtree (less accurate nodes should be modified more often) are taken into account. For a drawn node only matching variants are considered, but the user preferences can be also included.

When a non-terminal node is to be mutated, it can be pruned into a leaf or its test can be modified, recreated or copied from another node. These changes can be purely random or can be based on a local search (so-called memetic extensions). As for a leaf node, it can be transformed into a stump (one internal node and leaves) and a new test needs to be created. Concerning simpler symmetric cross-over variants, only tests or whole sub-tress can be exchanged between two individuals. In more complicated asymmetric scenarios, a subtree from a donor position in one tree is duplicated and implanted to a receiver position in the second tree. In certain variants of both mutation and cross-over operators, randomly chosen dipoles[1] can be used to guide the decision tree modifications.

The fitness function, which drives the evolutionary search, should as much as possible close reflect the goal of the algorithm. In many of data mining tasks, one tries to find the best predictor, but simultaneously simplicity of such a predictor is often desired. It is well known that a classifier which perfectly classifies the training data is usually much worse, when tested on unseen data, due to the over-fitting problem. In case of decision trees, the predictor complexity can be reduced just to the number of nodes or can rely on the complexity of tests in internal nodes and/or models in the leaves. In the GDT system many forms of the fitness function, both single-objective or multi-objective, can be applied. As, in this paper, only univariate decision trees applied to classification problem are considered, the simple weighted form of the fitness function can be used:

$$Fitness(T) = Accuracy(T) - \alpha * Size(T), \tag{1}$$

where $Accuracy(T)$ represents the reclassification quality of the tree T, $Size(T)$ stands for the number of nodes in T and α is the user-supplied parameter, which can be possibly tuned up for the given dataset (default value $\alpha = 0.001$).

3 Spark-Accelerated Evolutionary Induction

For large-scale data, the most time-consuming operation in evolutionary induction is clearly the fitness evaluation, as it requires re-classifying of the whole training dataset for every tree in each iteration. Thus, in the proposed Spark-based acceleration, we decided to concentrate only on distributing the training dataset, as it enables the most productive parallelization. The rest of the evolution is unaffected in principle and is realized sequentially.

Firstly, the training dataset is loaded line-by-line and transformed into an RDD of elements representing observation groups of the same size. The number

[1] A dipole is a pair of observations; if observations come from different classes, a dipole is called mixed.

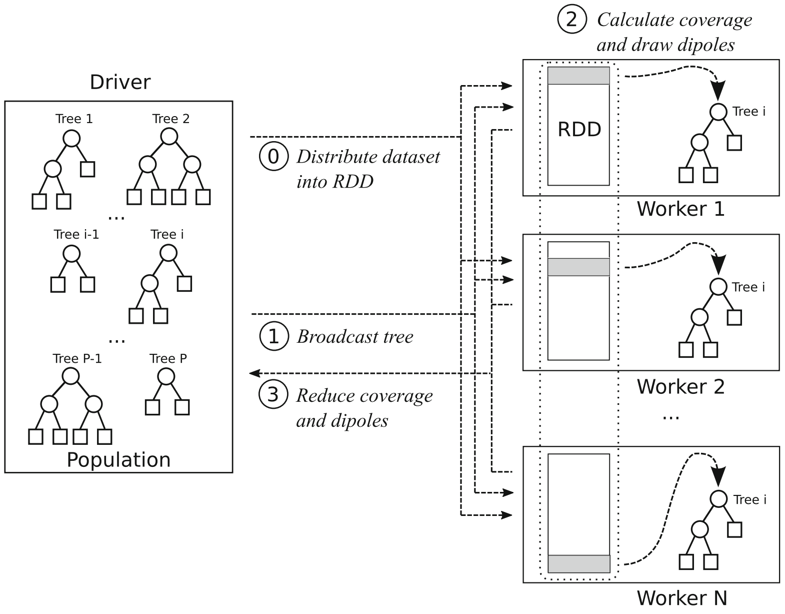

Fig. 1. Distributed fitness evaluation and dipoles searching.

of groups should correspond to the number of available computing nodes so the dataset chunks can be effectively cached in their memory. During the induction, all observations are passed through every transferred decision tree and arrangement of observations in its leaves is obtained (Fig. 1). It is realized by typical pair of `map-reduce` operations evoked on the grouped RDD: each group emits a locally processed copy of the tree (`map(group)` $\rightarrow$ `tree`) and the local trees are then reduced into a final result. Moreover, for each processed group a set of dipoles, which can be potentially used in genetic operators, is randomly chosen and implicitly reduced. Finally, the class distributions in each leaf can be calculated and the overall accuracy is estimated.

The proposed method is highly iterative, therefore an effective distribution of the dataset in the cluster memory is one of the crucial implementation aspects. The dataset can be loaded from a local file and distributed on the cluster (using `SparkContext::parallelize()` method from Spark API) or loaded from HDFS with `SparkContext::textFile()` in case of larger files. Each file line is parsed into an observation object which is randomly assigned to a group with a numeric ID. This creates a key-value RDD of <`groupID, observation`> elements that are grouped using `RDD::groupByKey()`. As the number of groups equals the number of computing nodes, this operation triggers a global dataset repartition that results in a uniform distribution of the data over the cluster. As this is a one-time operation, the cost of repartition is negligible and the lack of data skew is highly beneficial.

The Spark-based solution uses a multi-process architecture (Fig. 2). The original GDT system, implemented in C++, was modified to communicate with the

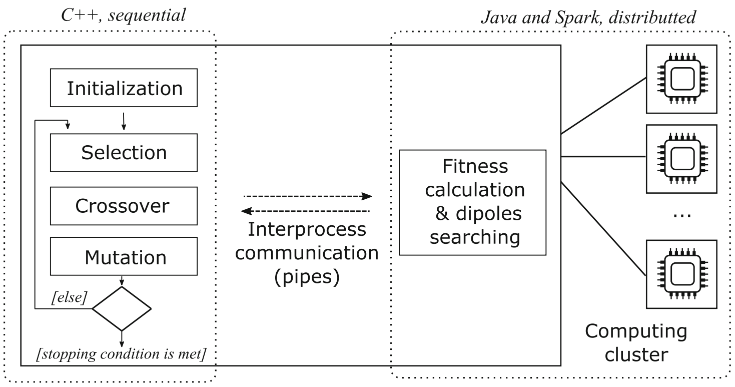
C++, sequential
Java and Spark, distributted
Initialization
Selection
Crossover
Mutation
[else]
[stopping condition is met]
Interprocess communication (pipes)
Fitness calculation & dipoles searching
...
Computing cluster

Fig. 2. Implementation concerns.

main Spark process (Driver) that reimplements the operations, which require access to the dataset, in Java. The Driver dispatches the work to multiple Spark Worker processes that are distributed over the cluster nodes. Both the Spark Driver and GDT applications are running on the same machine and use named pipes mechanism for inter-process request/response communication. As a result, the core evolution is performed in one process (C++ part), whereas the distributed fitness evaluation and dipoles searching are realized in Spark (Java part). Such a heterogeneous implementation provided a convenient way to independently modify and test both components.

4 Experimental Validation

The proposed solution is validated on both artificial and real-life large-scale datasets. The first group comprises 4 variants of *Chess* dataset (inspired by 3×3 chessboard) with increasing number of instances/observations. Moreover, two real-life datasets (*Suzy* and *Higgs*), which are the biggest classification problems available in UCI Repository [4], are analysed. The characteristics of the considered datasets are given in Table 1.

Table 1. Characteristics of the datasets: name, number of instances, number of attributes, and number of classes.

Dataset	Instances	Attributes	Classes
Chess1M	1 000 000	2	2
Chess5M	5 000 000	2	2
Chess10M	10 000 000	2	2
Chess20M	20 000 000	2	2
Suzy	5 000 000	18	2
Higgs	11 000 000	28	2

All presented results were obtained with a default set of parameter's values from the sequential version of the GDT system. In this paper, we are focused only on time performance thus results for the classification accuracy are not included. Moreover, the proposed solution only accelerates evolution so it does not really affect the resulting classifiers. For detailed information about accuracy performance, we refer a reader to our previous papers [7,14].

The experiments were performed on a cluster of 18 SMP workstations connected by a Gigabit Ethernet network. Each cluster node was equipped with a quad-core Intel Xeon E3-1270 3.4 GHz CPU, 16 GB RAM and was running Ubuntu 16.04 (Linux 4.4). 16 worker nodes were used by Spark executors, one node was dedicated to Spark Master and HDFS NameNode and the last node was running Spark Driver and GDT C++ processes. The cluster used Apache Hadoop 2.7.3 and Apache Spark 2.2.0 [1] deployed in a standalone mode with a single executor on each worker node.

The experiments were run on a different number of total CPU cores, distributed uniformly over the executor worker nodes. The speedups calculated for all *ChessXM* variants and the different number of cores (from 2 up to 64) are presented in Fig. 3. The speedup is defined as the ratio of the baseline execution time to the time of a given Spark run. The baseline time was obtained by running the solution on a single-threaded version of the Java module with the Spark integration completely disabled.

It should be noticed that for the smallest dataset (1 million of instances), the proposed Spark-based acceleration is completely useless. For larger variants, the observed speedups are much better, but not very impressive (e.g. around 10 for 32 cores). It could be also noted that moving from 32 to 64 cores, results in the visibly slower increase of speedup. Such behavior can be easily

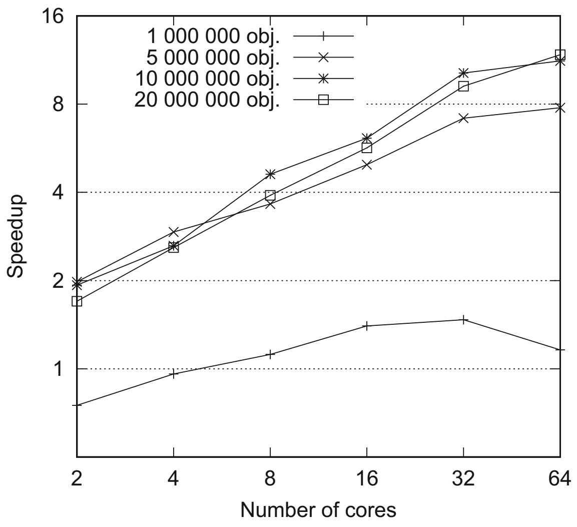

Fig. 3. Performance evaluation: speedup obtained on a different number of cores for the *Chess* dataset variants with increasing number of observations.

explained by an inevitable executor overhead (e.g. scheduler delay, task serialization/deserialization, result read/write and shuffle time) introduced by Spark. Figure 4 illustrates this situation for the *Higgs* dataset. Each computational task contains a short (about 8 ms) overhead that becomes more significant in the overall performance as the actual computing time decreases with the increase of a number of cores.

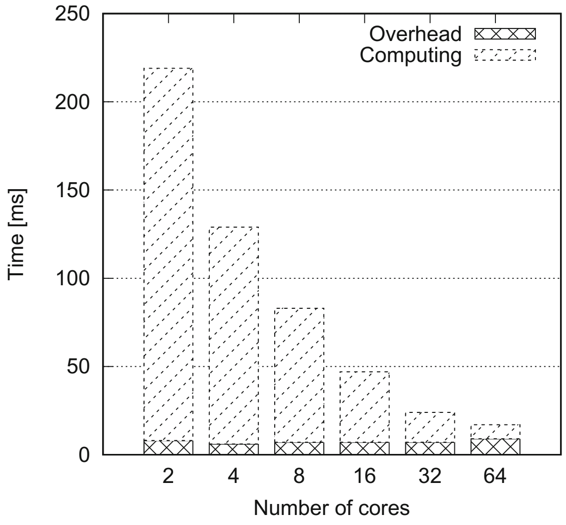

Fig. 4. Performance evaluation: detailed time-sharing information of induction task with a different number of cores. Evaluation performed on the *Higgs* dataset.

The results obtained on the real-life data are presented in Table 2. Although datasets contain a larger number of attributes, the observed patterns are rather expected and analogous to previously seen on 3 larger artificial datasets. It can be also noticed that for the smaller dataset (*Suzy*), increasing the number of cores from 32 to 64, does not give any significant acceleration as a slightly increased overhead cancels out the speedup gain.

In this paper, Spark is applied to speed up our global decision tree induction, but in the well-know Spark's *MLlib* library [15], distributed versions of classical top-down decision tree induction algorithms are available. In order to have at least an approximate reference, the basic algorithm (with default settings) was launched on *Suzy* and *Higgs* datasets. The estimated accuracies are practically the same as obtained by the GDT system, but the decision trees generated by *MLlib* are clearly overgrown (63 nodes for both datasets as opposed to 5 to 9 and 5 to 11 nodes, correspondingly). However, the induction time on 64 cores is only 1–2 min, which is substantially shorter than the evolutionary approach (around 30 min). Such a difference in tree complexity can be clearly explained by the fact that the *MLlib* algorithm is based on the top-down approach whereas the GDT system uses global induction. Global evolutionary inducers are known to provide smaller (less complicated) models [3].

Table 2. Obtained speedup for real-life datasets for a different number of CPU cores.

Dataset	Speedup on different number of cores					
	2	4	8	16	32	64
Suzy	1.74	2.64	3.79	5.80	8.16	8.03
Higgs	1.72	2.49	3.65	6.15	10.22	14.23

5 Conclusion

In this paper, Apache Spark is applied to speed up an evolutionary induction of classification trees in the Global Decision Tree (GDT) system. The most computationally demanding operations are fitness evaluations as they require reclassifying of the whole training dataset and for large-scale data it results in too long induction time. Hence, the dataset is transformed into RDD and it enables distributed and in-memory calculations on workers. Even preliminary experimental results show that the time of evolutionary induction can be significantly reduced for the largest datasets. It should be also noticed that the Spark-based solution can be easily scaled up just by connecting new computing stations.

In the future works, we plan to perform more extensive testing of the proposed approach. We are especially interested in revealing what are the limits of Spark-based induction: how big datasets could be processed on a given hardware in a fixed time. Our future investigations will also deal with a hybrid, e.g. Spark+CUDA, approaches as well as parallelization of other more elaborated decision trees, like model or oblique trees.

Acknowledgments. This work was supported by the grant S/WI/2/18 from Bialystok University of Technology founded by Polish Ministry of Science and Higher Education.

References

1. The Apache Software Foundation. Apache Spark - Lightning-Fast Cluster Computing (2018). https://spark.apache.org/
2. Alba, E., Tomassini, M.: Parallelism and evolutionary algorithms. IEEE Trans. Evol. Comput. **6**(5), 443–462 (2002)
3. Barros, R.C., Basgalupp, M.P., Carvalho, A.C., Freitas, A.A.: A survey of evolutionary algorithms for decision-tree induction. IEEE Trans. SMC, Part C **42**(3), 291–312 (2012)
4. Blake, C., Keogh, E., Merz, C.: UCI repository of machine learning databases (1998). http://www.ics.uci.edu/~mlearn/MLRepository.html
5. Czajkowski, M., Jurczuk, K., Kretowski, M.: A parallel approach for evolutionary induced decision trees. MPI+OpenMP implementation. In: Rutkowski, L., Korytkowski, M., Scherer, R., Tadeusiewicz, R., Zadeh, L.A., Zurada, J.M. (eds.) ICAISC 2015. LNCS (LNAI), vol. 9119, pp. 340–349. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19324-3_31

6. Czajkowski, M., Jurczuk, K., Kretowski, M.: Hybrid parallelization of evolutionary model tree induction. In: Rutkowski, L., Korytkowski, M., Scherer, R., Tadeusiewicz, R., Zadeh, L.A., Zurada, J.M. (eds.) ICAISC 2016. LNCS (LNAI), vol. 9692, pp. 370–379. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39378-0_32
7. Czajkowski, M., Kretowski, M.: Evolutionary induction of global model trees with specialized operators and memetic extensions. Inf. Sci. **288**, 153–173 (2014)
8. Deng, C., Tan, X., Dong, X., Tan, Y.: A parallel version of differential evolution based on resilient distributed datasets model. In: Gong, M., Pan, L., Song, T., Tang, K., Zhang, X. (eds.) BIC-TA 2015. CCIS, vol. 562, pp. 84–93. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-49014-3_8
9. Ferranti, A., Marcelloni, F., Segatori, A., Antonelli, M., Ducange, P.: A distributed approach to multi-objective evolutionary generation of fuzzy rule-based classifiers from big data. Inf. Sci. **415–416**, 319–340 (2017)
10. Funika, W., Koperek, P.: Towards a scalable distributed fitness evaluation service. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K., Kitowski, J., Wiatr, K. (eds.) PPAM 2015. LNCS, vol. 9573, pp. 493–502. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32149-3_46
11. Gong, Y.J., Chen, W.N., Zhan, Z.H., Zhang, J., Li, Y., Zhang, Q., Li, J.J.: Distributed evolutionary algorithms and their models: a survey of the state-of-the-art. Appl. Soft Comput. **34**, 286–300 (2015)
12. Grama, A., Karypis, G., Kumar, V., Gupta, A.: Introduction to Parallel Computing. Addison-Wesley, Boston (2003)
13. Jurczuk, K., Czajkowski, M., Kretowski, M.: Evolutionary induction of a decision tree for large-scale data: a GPU-based approach. Soft Comput. **21**(24), 7363–7379 (2017)
14. Kretowski, M., Grzes, M.: Evolutionary induction of mixed decision trees. Int. J. Data Warehous. Min. (IJDWM) **3**(4), 68–82 (2007)
15. 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**(1), 1235–1241 (2016)
16. Michalewicz, Z.: Genetic Algorithms + Data Structures = Evolution Programs. Springer Science & Business Media, Heidelberg (2013). https://doi.org/10.1007/978-3-662-03315-9
17. Pulgar-Rubior, F., Rivera-Rivas, A., Perez-Godoy, M., Gonzalez, P., Carmona, C., del Jesus, M.: MEFASD-BD: multi-objective evolutionary fuzzy algorithm for subgroup discovery in big data environments - a MapReduce solution. Knowl.-Based Syst. **117**, 70–78 (2017)
18. Qi, R., Wang, Z., Li, S.: A parallel genetic algorithm based on Spark for pairwise test suite generation. J. Comput. Sci. Technol. **31**(2), 417–427 (2016)
19. Teijeiro, D., Pardo, X.C., González, P., Banga, J.R., Doallo, R.: Implementing parallel differential evolution on spark. In: Squillero, G., Burelli, P. (eds.) EvoApplications 2016. LNCS, vol. 9598, pp. 75–90. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-31153-1_6
20. Wu, X., Zhu, X., Wu, G.Q., Ding, W.: Data mining with big data. IEEE Trans. Knowl. Data Eng. **26**(1), 97–107 (2014)
21. Zaharia, M., et al.: Apache Spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016)