# Performance Analysis and Auto-tuning for SPARK in-memory analytics

Dimitra Nikitopoulou*, Dimosthenis Masouros*, Sotirios Xydis*†, Dimitrios Soudris*
*Microprocessors and Digital Systems Laboratory, ECE , National Technical University of Athens, Greece
†Department of Informatics and Telematics (DIT), Harokopio University of Athens (HUA), Greece
*{diminiki, dmasouros, sxydis, dsoudris}@microlab.ntua.gr, †sxydis@hua.gr

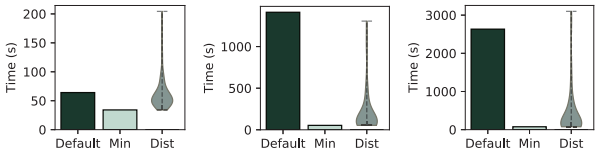
***Abstract*—Recently the Apache Spark in-memory computing framework has gained a lot of attention, due to its increased performance on large-scale data processing. Although Spark is highly configurable, its manually tuning is time consuming, due to the high-dimensional configuration space. Prior research has emerged frameworks able to analyze and model the performance of Spark applications, however they either rely on empirical selection of important parameters or/and follow a pure application-specific modeling approach. In this paper, we propose an end-to-end performance auto-tuning framework for Spark in-memory analytics. By adopting statistical hypothesis testing techniques, we manage to extract the higher order effects among differing parameters and their significance in performance optimization. In addition, we propose a new systematic meta-model driven approach utilizing cluster-, rather than application-wise performance modeling for traversing the configuration search space. We evaluate our approach using real scale analytic benchmarks from HiBench suite and show that the proposed framework achieves an average performance gain of $\times$3.07 for known and $\times$2.01 for unknown applications, compared to the default configuration.**

***Index Terms*—In-Memory Computing, Apache Spark, Auto-Tuning, Performance Analysis, Machine Learning, Optimization**

## I. Introduction

The growing demand for faster processing and analytics on big data, has led to the adoption of In-Memory Computing (IMC) frameworks, which can provide much higher performance gains [1] over traditional on-disk computing frameworks, e.g. Hadoop MapReduce. Apache Spark [2] forms the most popular IMC solution, supporting distributed in-memory computation of large-scale analytics. Spark offers a wide variety of configuration parameters which can be adjusted to alter several aspects of its runtime engine for increasing performance. In fact, Spark exposes more than 150 configuration parameters. Analyzing and exploring the impact of various configurations on the performance of Spark applications and also examining the inter-correlation between different parameters is a painful procedure for developers, due to *i)* the high-dimensional configuration space, *ii)* the huge, cumulative, running time required and *iii)* the time required to comprehend in depth the purpose of each parameter. Developers tend to tune empirically only the most obvious performance-related parameters, such as the number of executors or the amount of RAM per executor, while, also, not taking into account the performance variations of applications for various dataset sizes.

This work is partially funded by the EU Horizon 2020 research and innovation programme, under project EVOLVE, grant agreement No 825061.




(a) Bayes (b) Kmeans (c) SVM

Fig. 1: Performance variation of Spark applications for different configurations.

However, Spark applications can be bottlenecked by any resource in the cluster with CPU, memory and network bandwidth being the most common congestion points [3]. Even though Spark itself specifies a tuning guide to direct the regulation of its aspects [4], it still does not provide a procedure to automate the process of parameters exploration, while, on top of that, newer Spark versions tend to provide additional available parameters, which further increase the complexity in an exponential manner. From the above discussion, it is evident that there is a need of automated tuning frameworks, to ease the exploration of this high-dimensional search space.

Figure 1 showcases, through a representative example, the importance of Spark parameter tuning regarding to their effect on performance variability and optimization. We examine three Spark applications, i.e. Bayessian classification, Kmeans clustering and SVM classification, executed with the default configuration and 100 random generated alternatives. Figure 1 depicts the respective results, where *Default* refers to the execution time for the default Spark configuration, *Min* to the best performance achieved from the 100 different executions and *Dist* to the distribution of execution times over all the experiments. The considered exemplary scenario reveals the variation regarding the performance of the default configuration compared to the rest of the design space, since for Bayes the default execution relies in the lower quartile of the distribution, for Kmeans it forms an outlier of the respective distribution, while in SVM it is located in the upper quartile. Most importantly, proper tuning always achieves higher performance compared to the default one, ranging from $\times$1.9 (for Bayes) up to $\times$34 (for SVM).

As expected, performance optimization of Spark applica-

tions has been in the center of attention of many research groups. A large amount of prior works concern the improvement of the internal runtime engine and resource allocation of Spark itself [5], [6], [7], [8], [9], either by optimizing internal Spark characteristics, such as the way tasks are scheduled on the executors [5], [8] or Spark's serialization technique [6], [7] or by providing external advanced resource allocation mechanisms [9] for better utilization of the Spark cluster. Another considerable amount of research activities have examined the development of frameworks for automatically tuning Spark applications [10], [11], [12], [13], [14], [15]. Couple of these works [10], [13] rely on simple ML approaches to model performance, e.g., Decision Trees, while the others either employ hierarchical modeling techniques [11], deep neural networks [15] or do not model performance at all [14].

Most of these works [10], [12], [13], [15] rely on designating the significance of Spark parameters intuitively, thus, possibly missing potential significant parameters. In our work, we recognize this inefficiency, bringing into statistical models to assess the importance of each parameter. In addition, all the aforementioned Spark tuning solutions examine and model application-specific scenarios, where each one is executed repeatedly on the cluster, hence, failing to predict the performance of new applications arriving on the cluster.

Though prior scientific works have proposed tuning frameworks, which automatically configure Spark parameters to optimize performance, they fail to provide robust methodologies able to comply with changes in application characteristics, data volumes and the Spark itself. As recently discussed in [16], major principles as *versioning adaptivity*, *data resiliency*, *workload heterogeneity* and *fast convergence* should be considered as first class design concerns for Spark auto-tuning. These standards tackle the provision of accurate performance predictions for different workloads and volume sizes as well as enable near-optimal configuration in an instant manner, to amortize optimization costs through the resulting savings.

To address the aforementioned challenges, in this work, we present a comprehensive analysis of parameter tuning and propose an end-to-end performance auto-tuning framework for Spark in-memory analytics. Our framework employs robust statistical hypothesis testing approaches to evaluate the significance of Spark parameters, thus confronting the *version adaptivity* challenge. We address *workload heterogeneity* and *data resiliency* through a clustering-wise performance modeling on low level time-series monitoring data for a wide range of scenarios, thus handling the unpredictability of both unseen applications and varied input dataset sizes. By utilizing genetic optimization algorithms, our framework is able to traverse the search space efficiently, providing *fast convergence* to near-optimal solutions, while its gained performance scales across increased dataset sizes. Our experimental evaluation presents that our employed ML model achieves high accuracy, i.e., $R^2$ scores equal to $0.97$ for known and $0.96$ for the majority of unknown applications, providing an average speedup of $\times 3.07$ and $\times 2.01$ respectively compared to the default configuration.

The rest of the paper is organized as follows. Section II gives an overview of the Spark framework and describes our experimental setup. Section III presents our proposed Spark auto-tuning framework and also provides insights observed during the development. Finally, sections IV and V show our experimental results and conclude the paper respectively.

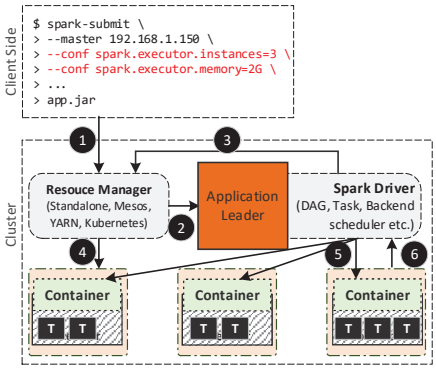



Fig. 2: Apache Spark overview.

## II. Spark Overview & Experimental Setup

**Spark Overview:** Figure 2 shows an overview of the Spark framework. From the client side, developers submit their applications (❶) to a cluster Resource Manager (RM), e.g. YARN, Kubernetes, etc. When submitting their application, users can specify Spark configuration parameters, such as the number of executor instances, the executor's memory and others. Once the RM receives the request, an Application Leader process is spawned (❷), which contains the Spark Driver. The main objective of the Spark driver is to perform Directed Acyclic Graph (DAG) scheduling of tasks on the executor nodes. Once the Driver creates the applications tasks, it further interacts with the RM to initialize the spark executors on the underlying cluster (❸,❹) and assign tasks to them (❺). Finally, throughout the lifetime of the application, executors interact directly with the Driver (❻) and once all tasks are finished the output is collected by the latter.

**Experimental Setup:** All of our experiments have been conducted on a high-end dual-socket Intel® Xeon® E5-2658A v3 server with 24 logical cores per socket and 128GB of memory, with Apache Spark v2.3.0 and a Hadoop Distributed File System (HDFS) v2.7.2. Although Spark traditionally assumes cluster environments, the need for data analytics at the edge over single "fat" server infrastructures with tens of cores and hundreds of gigabytes of memory [17], as well as the statistics from production class dataset sizes [18], challenge this assumption. To quantify the impact of parameter tuning, we study 18 open-source Spark benchmarks derived from the HiBench suite [19]. During the development of our framework we utilized 12 out of the 29 benchmarks provided by HiBench, i.e., Bayes Classification, Kmeans clustering, SVM, Aggregate, Join, Scan, Pagerank, Linear Regression, Gradient Boosted Trees, Sort, Latent Dirichlet Allocation and Terasort which form a representative set of benchmarks from a wide

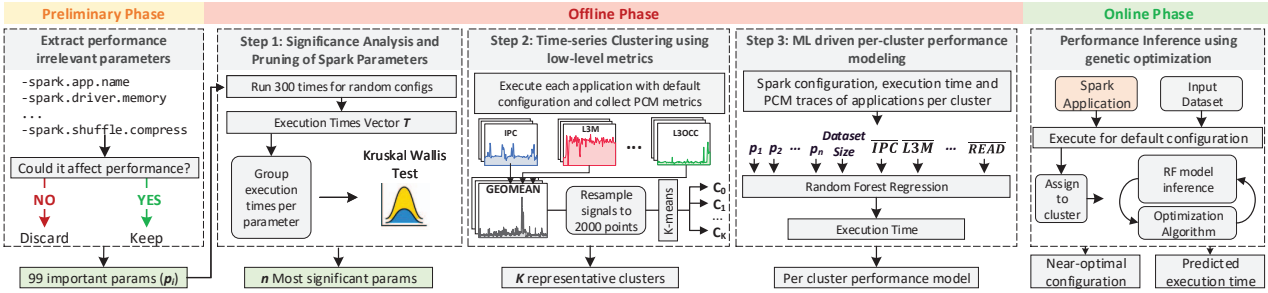



Fig. 3: Overview of our ML-driven Spark configuration auto-tuning framework.

range of categories - micro operations, Machine Learning, Websearch and SQL related. Last, HiBench provides different input data sizes ranging from Tiny up to Big Data. To immitate data diversity on Spark "fat" nodes, we examine Tiny, Small and Large datasets which represent volume sizes ranging from 10.8KB up to 48GB of data.

## III. ML-DRIVEN AUTO-TUNING FRAMEWORK

Figure 3 depicts an overview of the proposed approach for end-to-end and automatic performance optimization of Spark applications. Opposed to previous works that rely on hypothetical assessment of Spark parameters [11], [12], [14], our approach depends on statistical models to evaluate the impact of each parameter on the performance of Spark applications (Step 1). By employing cluster-wise performance models, our framework is able to find near-optimal configuration over unseen applications (Steps 2 and 3). Finally, the performance models are utilized online in conjunction with meta-heuristic optimization to quickly traverse the configuration search space, thus, enabling optimized Spark deployments.

### A. Preliminary parameter pruning and design space assembly

Spark offers more than 150 configuration parameters [20]. However, a large portion of these parameters refer to Spark configurations not relative to performance, e.g. the name of the spark application (spark.app.name). Thus, as a preliminary step, these parameters have been identified and removed from the rest of the exploration process. This procedure resulted in 99 possibly significant parameters, which concern a wide range of configurations that can be applied on the spark system. These 99 parameters concern either numeric configurations (e.g. spark.executor.memory etc.) or boolean variables (e.g. spark.shuffle.compress). We employed a partial factorial design of experiments for each parameter (except for booleans) sampling 3 representative values from their accepted range.

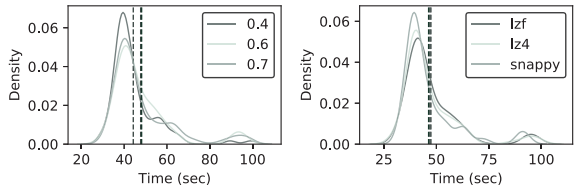
### B. Offline: Sensitivity analysis and performance modeling

The offline phase aims to further prune the parameter configuration design space and also build performance prediction models for spark applications. It consists of 3 discrete stages:

**Step 1:** This step concerns the further pruning of the parameter configuration design space. The impact of each one of the 99 parameters extracted in the preliminary phase on the overall performance of the applications differs. To determine the sensitivity of each parameter, our framework performs a significance analysis over these 99 important parameters. To do so, first, the framework executes each spark application 300 times for diverse input datasets, each time with a different, random selected configuration vector. The above procedure results in a 300-dimensional vector, where each element corresponds to the execution time of the application using the respective random generated configuration.

To determine the importance of each parameter our framework applies the Kruskal-Wallis test [21] on the results produced through these executions. Kruskal-Wallis tests the null hypothesis H0, which states that samples in all groups are drawn from populations with the same mean values, whereas the alternative hypothesis states that at least two of the groups have mean values that differ. The algorithm returns a $p$-value, based on which the hypothesis is either accepted or rejected. We set that value to 0.05, which denotes that execution time distributions that differ more than 5% in their mean value designate an important spark parameter. Figure 4 depicts two representative density plots, showing the process followed to determine the importance of each spark parameter. Figure 4a, depicts a significant parameter, since for the value 0.4 we have a shift of the distribution to the left. On the other hand, Figure 4b has no such significant shift so it corresponds to an insignificant parameter. Regarding HiBench, this procedure resulted in 23 significant parameters, which are depicted in Figure 5, where values closer to 0 denote a more important instance. We can see that certain parameters (e.g. yarn.executor.cores) are critical for all the applications, while others are application or even datasize specific.

**Step 2:** Identifying the optimal configuration of applications requires examining multiple alternatives and evaluating them based on the execution time, which can be a very time consuming task. Thus, it is crucial to build an accurate performance model able to predict the execution time of spark applications for a given configuration. However, building a universal model able to predict the performance of any Spark application is not trivial, due to the diverse characteristics and performance behaviors between them. To address this difficulty, we adopted an application clustering strategy, thus, allowing the development of clustering-wise performance models (Step 3).




(a) spark.memory.fraction (b) spark.io.compression.codec

Fig. 4: Execution time density plot of a more significant (left) and a less significant (right) spark parameter.

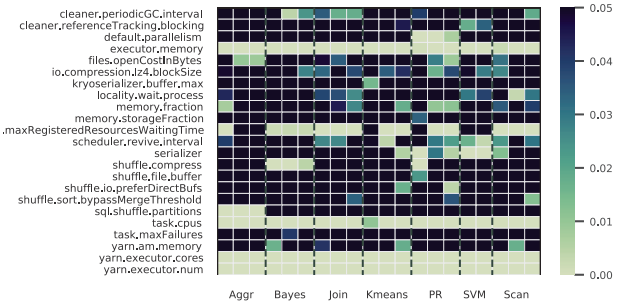



Fig. 5: Per-application importance of 23 most significant for the HiBench suite. Each square per benchmark refers to the dataset size, i.e., tiny, small and large respectively

In contrary to other monolithic approaches [10], [13], we evaluate the similarity between different applications by testing their low-level micro-architectural characteristics, i.e. IPC, core frequency, cache behavior and memory I/O for the default Spark configuration. To capture such attributes, we employ the Performance Counter Monitoring (PCM) API [22], which provides a plethora of hardware performance counters. for each logical core, each socket, as well as the whole server system in a time-series manner. This procedure results in 9 time-series, where each one corresponds to a different metric (e.g. IPC, L3 cache misses etc.). Then, these signals are fused into a single, complex time-series signal which is used to characterize the respective application, where, each point is calculated as the element-wise geometric mean of the initial 9 low-level metrics. Due to the different execution duration of the applications, the geometric mean data-series are resampled and reformed to have a length of 2000 points in time. Finally, the framework assigns applications into groups using time-series Kmeans clustering. For HiBench, we explore several clustering configurations to balance the trade-off between maximizing prediction accuracy vs. minimizing application specificity. Figure 6 shows the resulted clustering with 6 clusters, where the blue line denotes the centroid signal of each cluster. As shown, certain applications (Pagerank and Terasort) have been assigned their individual cluster, due to the intense fluctuations of their fused, "complex" signal.

**Step 3:** This step concerns the development of cluster-wise performance models. After the formation of the $K$ representative clusters on Step 2, our framework employs a machine learning approach for predicting the performance of Spark applications. During the development stage we have examined various ML modeling algorithms, provided by the scikit-learn library [23]. Each model receives as input 33 parameters, i.e., the 23 values of Spark's parameter, the size of the input dataset and the mean value of the respective low-level metrics ($\overline{IPC}$,$\overline{L3M}$,etc.) and performs the regression over the execution time of applications on the cluster.

TABLE I: Per-cluster $R^2$ score of different ML algorithms

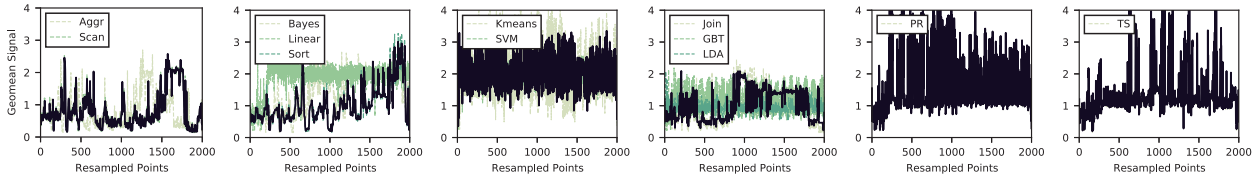
| Algorithm | C0 | C1 | C2 | C3 | C4 | C5 |
|---|---|---|---|---|---|---|
| Linear | 0.51 | 0.36 | 0.40 | 0.34 | 0.39 | 0.41 |
| Ridge | 0.52 | 0.36 | 0.40 | 0.34 | 0.39 | 0.41 |
| Lasso | 0.16 | 0.31 | 0.31 | 0.33 | 0.35 | 0.39 |
| ElasticNet | 0.43 | 0.30 | 0.29 | 0.30 | 0.33 | 0.38 |
| BayesRidge | 0.52 | 0.36 | 0.40 | 0.34 | 0.39 | 0.42 |
| SGD | 0.51 | 0.32 | 0.19 | 0.22 | 0.38 | 0.43 |
| SVR | 0.75 | 0.36 | 0.40 | $< 0$ | 0.28 | 0.16 |
| kNeighbors | 0.78 | 0.35 | 0.50 | 0.23 | 0.28 | 0.16 |
| GaussProc | 0.77 | 0.51 | 0.56 | 0.49 | 0.36 | 0.31 |
| DecisTree | 0.83 | 0.83 | 0.90 | 0.88 | 0.80 | 0.56 |
| **RandForest** | **0.90** | **0.87** | **0.93** | **0.92** | **0.85** | **0.73** |
| MLP | 0.87 | 0.71 | 0.88 | 0.76 | 0.50 | 0.40 |

Table I shows the accuracy achieved by each algorithm evaluated using the coefficient of determination, known as $R^2$ score, for all the clusters. As shown, the linear algorithms, as anticipated, as well as SVM, K-neighbors and Gaussian Process regressors cannot train highly accurate models for our 33 dimensional problem. Multi-Layer Perceptron (MLP) and Random Forest regression models were the most prominent solutions, with the latter achieving $R^2$ scores greater than $0.85$ in most cases, and $0.73$ in the worst.

### C. Online: Inference of performance optimized deployment

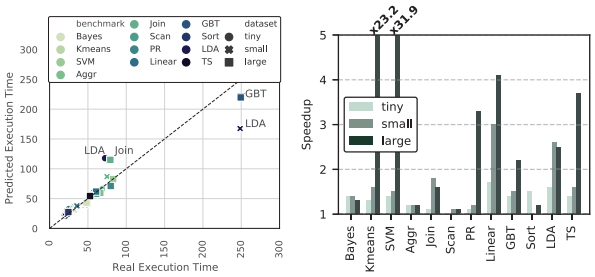
This phase identifies an optimized Spark configuration at the time of submission of the Spark job. In cases of unseen applications, they are first executed once, with the default configuration, in order to collect PCM metrics and be assigned to a cluster. Then, the framework iteratively queries the respective performance model (from Step 3) for different input parameters to determine a performance optimized configuration.

We utilize meta-heuristic optimization, to efficiently explore the underlying search space and ensure convergence towards optimal solutions. We utilize OpenTuner [24] framework encoding as objective function the minimization of the execution time. During the development of our framework we examined four different optimization techniques, i.e., Bandit, Particle Swarm Optimization, Differential Evolution and Genetic Algorithms [24]. Our exploration showed that the Bandit and the Genetic algorithms were the most prominent solutions (data omitted due to limited space), with the latter providing faster convergence to near-optimal solutions and, thus, was selected as the optimization technique of our framework.



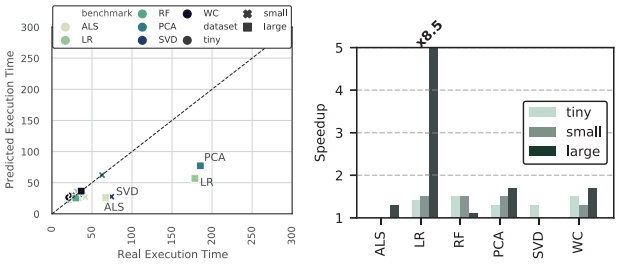

(a) Cluster 0 (b) Cluster 1 (c) Cluster 2 (d) Cluster 3 (e) Cluster 4 (f) Cluster 5

Fig. 6: Kmeans clustering of development applications into 6 partitions. Blue signal denotes the centroid signal per cluster.




(a) Real vs. Predicted performance (b) Speedup through auto-tuning

Fig. 7: Framework evaluation on development applications




(a) Predicted to Real performance ratio (b) Auto-tuning speedup achieved

Fig. 8: Framework evaluation on unseen applications

## IV. Experimental Evaluation

### A. Model accuracy and performance improvements

Figures 7a and 7b show the prediction accuracy as well as the application speedup achieved through auto-tuning respectively, for all the different application-dataset pairs. Regarding the efficiency of the RF regressor, we see that model achieves high levels of accuracy, with $R^2 = 0.97$, and the majority of the points lying on the $45^o$ residual line. Taking a closer look at the four outliers, we see that they all belong to the same cluster (Cluster 3), confirming the evident deviation between the applications and the centroid signal.

Regarding the speedup achieved through the auto-tuning process, we notice an average of $\times 3.07$ speedup achieved over all the examined applications. The maximum speedup achieved is $\times 31.9$ for SVM with 20 GB of input data, which can be explained by the fact that SVM is both compute and memory intensive and, therefore, there is a great number of parameters that affect its performance. Moreover, the minimum speedup is observed for the Scan application, due to the fact that the default configuration provides near-optimal performance and, thus, there is no margin for improvement. Finally, we also see increased performance gains for higher dataset sizes in most cases, which is beneficial considering the *"data resiliency"* requirement described in Section I.

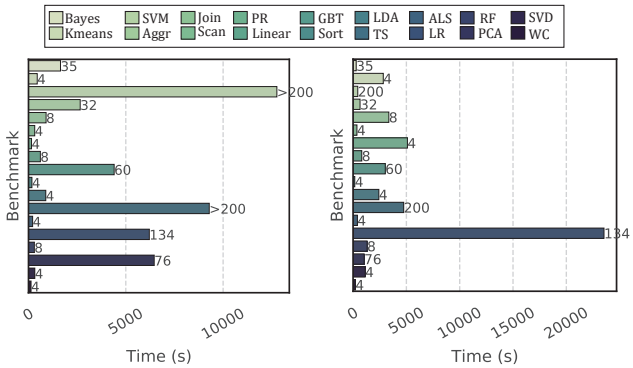
### B. Generalizing on unseen applications

In this section, we repeat the experiments of Section IV-A, but this time for a set of *unseen* applications. With the term *unseen* we designate applications that have not been used during the development stage of our framework (Offline Phase) and can be thought as new applications arriving on a Spark cluster. For this purpose, we exploit 6 additional benchmarks from the HiBench suite, i.e., Alternating Least Squares, Logistic Regression, Random Forest, Principal Components Analysis, Singular Value Decomposition and WordCount. As mentioned in Section IV-A, when new applications are submitted, they are executed with the default configuration, to be designated to a cluster. Table II shows the respective cluster assignments.

TABLE II: Cluster assignment of unseen applications

| ALS | LR | RF | PCA | SVD | WC |
|---|---|---|---|---|---|
| Cluster 1 | Cluster 2 | Cluster 1 | Cluster 3 | Cluster 1 | Cluster 1 |

Similarly to Section IV-A, Figures 8a and 8b show the performance prediction accuracy and speedup achieved for the unseen applications. Regarding the accuracy of the cluster models, we see that for the majority of application-dataset pairs $(14/18)$ the inference engines provide accurate enough predictions, with an average $R^2$ score of $0.96$. Regarding the performance, we can see that even though in certain cases we achieved no performance improvement (e.g. ALS and SVD), in general we observe an average of $\times 2.01$ speedup over all the examined applications. What is worth noting here is that the LR benchmark achieved the maximum speedup of $\times 8.5$ for large input dataset. If we take a closer look at Table II, we notice that the LR benchmark has been assigned to Cluster 2, same as Kmeans and SVM of experiment IV-A, which also achieved similar performance gains. This observation clearly validates the prospects of cluster-wise performance modeling.




(a) Small input dataset (b) Large input dataset

Fig. 9: Exploration time savings versus model-free optimization process. Bars show the time for depicting the optimal configuration, and values the number of configurations searched.

### C. Comparison with model-free optimization process

We compare the time savings of the employment of the ML regressor compared to a model-free optimization process, which has also been employed in previous works [14]. To do so, we apply the genetic optimizer directly on Spark, testing 200 different configurations and check the time overhead and the number of different configurations that need to be tested for each application with small and large datasets to find a solution with comparable performance, i.e., within 1% to the one provided by the RF regressor. Figure 9 shows the corresponding results. We notice that, sometimes, the optimizer that evaluates the candidate solutions directly on Spark can produce comparable results with a small time overhead of less than fifteen minutes. However, in specific cases, this process could take several hours with over 200 configurations examined.

## V. Conclusion

We propose an end-to-end auto-tuning framework for Spark in-memory analytics that introduces statistical hypothesis testing and cluster-wise performance modeling in an ML-driven performance optimization pipeline to enable high quality tuning solutions. Experimental results show that the proposed approach can improve the performance by an average of $\times 3.07$ for known and $\times 2.01$ for unknown applications and, also, reduce the time required to find optimized configurations.

## References


[1] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan, "Clash of the titans: Mapreduce vs. spark for large scale data analytics," *Proceedings of the VLDB Endowment*, vol. 8, no. 13, pp. 2110–2121, 2015.

[2] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[3] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pp. 293–307, 2015.

[4] *Tuning Spark*. https://spark.apache.org/docs/latest/tuning.html.

[5] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 69–84, 2013.

[6] Y. Zhao, F. Hu, and H. Chen, "An adaptive tuning strategy on spark based on in-memory computation characteristics," in *2016 18th International Conference on Advanced Communication Technology (ICACT)*, pp. 484–488, IEEE, 2016.

[7] T. Chiba and T. Onodera, "Workload characterization and optimization of tpc-h queries on apache spark," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 112–121, IEEE, 2016.

[8] K. Wang, M. M. H. Khan, N. Nguyen, and S. Gokhale, "A model driven approach towards improving the performance of apache spark applications," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 233–242, IEEE, 2019.

[9] Z. Hu, D. Li, D. Zhang, and Y. Chen, "Reloca: Optimize resource allocation for data-parallel jobs using deep learning," in *IEEE Conference on Computer Communications (INFOCOM) 2020*, pp. 1163–1171, 2020.

[10] G. Wang, J. Xu, and B. He, "A novel method for tuning configuration parameters of spark based on machine learning," in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 586–593, IEEE, 2016.

[11] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang, "Bestconfig: tapping the performance potential of systems via automatic configuration tuning," in *Proceedings of the 2017 Symposium on Cloud Computing*, pp. 338–350, 2017.

[12] Z. Yu, Z. Bei, and X. Qian, "Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 564–577, 2018.

[13] Z. Bei, Z. Yu, N. Luo, C. Jiang, C. Xu, and S. Feng, "Configuring in-memory cluster computing using random forest," *Future Generation Computer Systems*, vol. 79, pp. 1–15, 2018.

[14] A. Fekry, L. Carata, T. Pasquier, A. Rice, and A. Hopper, "Tuneful: An online significance-aware configuration tuner for big data analytics," *arXiv preprint arXiv:2001.08002*, 2020.

[15] M. Li, Z. Liu, X. Shi, and H. Jin, "Atcs: Auto-tuning configurations of big data frameworks based on generative adversarial nets," *IEEE Access*, vol. 8, pp. 50485–50496, 2020.

[16] A. Fekry, L. Carata, T. Pasquier, A. Rice, and A. Hopper, "To tune or not to tune? in search of optimal configurations for data analytics," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2494–2504, 2020.

[17] K. Iliakis, S. Xydis, and D. Soudris, "Resource-aware mapreduce runtime for multi/many-core architectures," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 897–902, 2020.

[18] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas, "Nobody ever got fired for using hadoop on a cluster," HotCDP '12, Association for Computing Machinery, 2012.

[19] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pp. 41–51, IEEE, 2010.

[20] *Spark Configuration*. https://spark.apache.org/docs/2.3.0/configuration.html. (Last accessed: 20/09/2020).

[21] W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis," *Journal of the American statistical Association*, vol. 47, no. 260, pp. 583–621, 1952.

[22] T. Willhalm, R. Dementiev, and P. Fay, "Intel performance counter monitor-a better way to measure cpu utilization," *Dosegljivo: https://software. intel. com/en-us/articles/intelperformance-counter-monitor-a-better-way-to-measure-cpu-utilization.[Dostopano: September 2014]*, 2012.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: Machine learning in python," *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[24] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pp. 303–316, 2014.