© 2020 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes,creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI 10.1109/TNSM.2020.3034824

You Only Run Once: Spark Auto-tuning from a Single Run

David Buchaca*, Felipe Portella‡, Carlos Costa§, and Josep LLuis Berral*

*Barcelona Supercomputing Center (BSC), C. Jordi Girona 1-3, 08034, Barcelona, Spain †Universitat Politècnica de Catalunya (UPC) - BarcelonaTECH, Barcelona, Spain ‡Petróleo Brasileiro S.A. (PETROBRAS), Rio de Janeiro, RJ, Brazil §IBM T. J. Watson Research Center, Yorktown Heights, NY, USA

***Abstract*—Tuning configurations of Spark jobs is not a trivial task. State-of-the-art auto-tuning systems are based on iteratively running workloads with different configurations. During the optimization process, the relevant features are explored to find good solutions. Many optimizers enhance the time-to-solution using black-box optimization algorithms that do not take into account any information from the Spark workloads. In this paper, we present a new method for tuning configurations that uses information from one run of a Spark workload. To achieve good performance, we mine the SparkEventLog that is generated by the Spark engine. This log file contains a large amount of information from the executed application. We use this information to enhance a performance model with low-level features from the workload to be optimized. These features include Spark Actions, Transformations, and Task metrics. This process allows us to obtain application-specific workload information. With this information our system can predict sensible Spark configurations for unseen jobs, given that it has been trained with reasonable coverage of Spark applications. Experiments show that the presented system correctly produces good configurations, while achieving up to 80% speedup with respect to the default Spark configuration, and up to 12x speedup of the time-to-solution with respect to a standard Bayesian Optimization procedure.**

***Index Terms*—Decision making for workload auto-tuning, Machine Learning, Spark auto-tuning, Workload modeling, Workload placement**

I. Introduction

BIG data processing frameworks, such as Apache Spark, are utilized in a wide range of industries. To support the diverse range of applications, Spark allows users to tune parameters that might significantly affect the performance of the workload. There are more than 100 parameters that can be configured in Spark [1], which makes the selection a difficult task.

There are different strategies for overcoming the problem of selecting good parameters for a specific job. Several methods have been proposed for auto-tuning hyper-parameters of configurable software, many of which are based on machine learning models. Models can be used to estimate how the application will perform under a particular configuration of the hyper-parameters. The best parameters found during the search are then provided to the user.

Note: David Buchaca and Felipe Portella contributed equally to this work. Corresponding authors: David Buchaca (email: david.buchaca@bsc.es), Felipe Portella (email: felipe@portella.com.br) and Carlos Costa (email: carlos.costa@ibm.com).

Current state-of-the-art work based on performance model search techniques have the drawback that retraining is required in order to generalize to unseen workloads [2]. This is a consequence of current models only receiving information from the configuration space that they need to optimize. Even works that provide an application identifier or the dataset size as an input to the learning algorithm need retraining for new applications. This limitation has led the community to explore how to improve current performance models and how to build solutions that do not need machine learning models. Some solutions [3] without performance models have been built based on an efficient search mechanism of the configuration space. This approach can achieve a relatively fast time-to-solution, while maintaining good quality in the results.

The claim that some performance model based systems are slower than non-model-based systems is reasonable. Auto-tuning systems that do not use a performance model do not need to spend time training an oracle when a new workload needs to be optimized. Therefore, it is possible that such systems can find a solution with fewer job executions than the number of runs required to train the oracle in a performance model based system. The objective of our work is to improve the current performance model based approaches in such a way that very few samples are needed to find a good configuration for a given job. In the most extreme case, we would like to perform model based optimization with a single run. Nevertheless, to achieve this goal we need a method to transfer previous knowledge about past examples to optimize new workloads.

Recent work from the area of configurable software [4] explores the idea of using transfer learning to facilitate the task of extracting measurements from a system in order to build a dataset. In our scenario, the data consist of Spark runs, and we can assume that cloud providers already possess a large number of Spark execution logs. Therefore, we focus our method on improving the time-to-solution of finding Spark configurations for applications instead of focusing on decreasing the number of Spark executions that have to be performed before good performance models are created. Nevertheless, we adapt our methodology with the goal of using transferable knowledge across Spark workloads.

Performance model based systems have to deal with three problems: sampling the search space to build a dataset, learning a model that can predict performance metrics accurately on new data, and searching in the parameter space to find a

suitable configuration for a new workload. Our work focuses on improving the last two components, previously mentioned, in the context of Spark auto-tuning. We want to build better features to achieve higher quality models that can more accurately predict performance metrics for new applications. Moreover, we want to speed up the time-to-solution of the whole process for a new application that the performance model has not seen in the training data.

To address the above challenges, we present a system that builds upon a performance model based search technique modifying a standard Bayesian Optimization (BO) process. Our system is enhanced with a mechanism that transfers knowledge from the execution logs to provide more information to the Machine Learning model. The key part of the proposed approach is that, in order to make the model generalize to unseen applications, we create a rich feature descriptor from the internal log file of events (SparkEventLog) generated by Spark. This descriptor contains a summary of the Spark Tasks and Stages performed by the workload, providing the oracle with low-level details about an application, without the need for extra instrumenting of the computational environment. Using this information, the model can learn the relationships between the parameters to be optimized and the application descriptor. This allows our model to generalize better to new unseen workloads without retraining. Moreover, our method does not need input from the user with respect to the type of application executed. In addition, since the feature descriptor is built from the SparkEventLog, our methodology respects code privacy because it does not need to access the source code of the application to be optimized.

Using the proposed feature descriptor, we can simulate a Bayesian Optimization process guided by an oracle that makes predictions conditioned on the application to be optimized. In this way, we can avoid costly runs of the application, which allows our methodology to achieve a faster time-to-solution with respect to a standard Bayesian Optimization process, while achieving almost the same quality.

We evaluated our approach on a dataset containing executions of different types of Spark applications, representing a wide range of workloads. The applications came from two popular Spark benchmarks suites: Hi-Bench [5] (from Intel) and Spark-Perf [6] (from DataBricks). Our experiments show that the proposed feature descriptor enhances the quality of the machine learning models with respect to other features used in the literature. Moreover, the cost of finding a good configuration is considerably reduced because our methodology can perform a reliable search in the configuration space.

CONTRIBUTIONS

In summary, our work makes the following contributions:

- We propose a novel feature descriptor created from the SparkEventLog that provides extra knowledge to the learning algorithms. We show that this extra information improves the quality of the predictions of machine learning models for unseen workloads by up to 34% with respect to the same models trained on standard feature representations.
- We develop a practical strategy that can auto-tune Spark configurations for new, never seen, workloads. Our solution can achieve up to 80% speedup on the execution with respect to the Spark default configuration. We show that our method can achieve up to 12x speedup compared to a standard Bayesian Optimization process, with almost equal results.

II. PROBLEM FORMULATION AND MOTIVATION

Let S be the Spark-Configuration search space. Let $s \in \mathcal{S}$ be a particular configuration of hyper-parameters in the search space. Let $e : \mathcal{S} \longrightarrow \mathbb{R}$ be the evaluation metric we aim to minimize, for example the execution time of a workload. The problem of finding a good SparkConfiguration has been framed by some studies [4] as finding s^*, defined as follows:

$$s^* = \arg\min_{s \in S} e(s) \tag{1}$$

The formulation of the problem in equation (1) states that the optimization to be performed only depends on $s \in \mathcal{S}$. Other works [7] define the problem with a different e function. Let $e : \mathcal{S} \times \mathcal{X} \longrightarrow \mathbb{R}$ be an evaluation metric, where $\mathcal{X}$ is a space of features that might capture relevant information about the problem. Given a vector of features $x \in \mathcal{X}$ the optimization problem becomes:

$$s^* = \arg\min_{s \in S} e(s; x) \tag{2}$$

Note that in equation (2) $s \in \mathcal{S}$ can change but $x \in \mathcal{X}$ is fixed. In some works [7], [8], $\mathcal{X}$ is a single variable containing the dataset size. This implies that the only information expected to affect the performance of an application is the dataset size. Since applications can be very different, this becomes an oversimplification of the problem, particularly in environments that use heterogeneous workloads. To improve this feature vector, other works add an identifier of the application [2], providing more contextual information to the optimization problem. Nevertheless, it is difficult to reuse and extrapolate from this contextual information, since it is an identifier that is usually encoded as a categorical variable.

A. Popular methodologies for the problem

A standard, general-purpose procedure for finding good configurations for workloads consists of using a sequential optimization procedure that learns over time which areas of the search space are likely to contain good solutions. A popular algorithm that follows this logic is Bayesian Optimization [9], which uses a Gaussian process to decide which of the regions of the search space are likely to contain a probable gain in performance. This methodology has been tested for searching workload configurations [4] with great success. First, a time budget T and a search space $\mathcal{S}$ are provided to the optimization procedure. Then, finding a configuration corresponds to updating the black box optimizer with the execution time $y = e(s)$, as shown in Fig. 1. This process generates vectors from the search space $\mathcal{S}$, while storing the best configuration found so far. Once the time T expires,

the optimization process finishes and the best configuration is returned. The main drawback of this approach is that a job execution is required for each update of the optimizer. Nevertheless, Bayesian Optimization techniques [10] have been used successfully to tune software. Since executions might take a long time, this technique might not be practical in some scenarios. To improve upon this solution, some works such as [3] focus on identifying a subspace of $\mathcal{S}$ relevant to the problem. They then focus on optimizing the subset

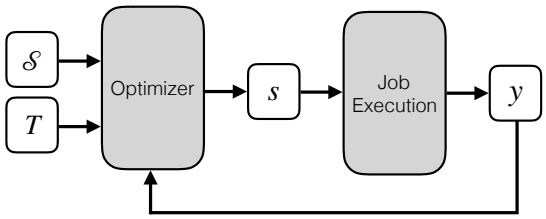


Fig. 1: Classical optimization pipeline followed by a Bayesian Optimization process.

In order to avoid running an application many times to find a good configuration, many works use a machine learning model trained to predict the evaluation metric from past executions [8], [11]–[15]. This model, sometimes called the performance model, aims to replace many of the job execution calls with the predictions provided by the model. The diagram in Figure 2 represents the overall pipeline for the optimization process. Note that this process has two different phases, which depend on whether an application is known or unknown. Let A be the set of applications that the system has observed. If the identifier of the app, app_{id}, is in A, then the optimization procedure is fast and the oracle can be used to make the predictions $\hat{y}$ that are provided to the search algorithm. Then the best prediction according to the model is retrieved and executed. If the application is not in A, the system enters into a sandboxing phase, where many samples have to be generated and executed, and the oracle needs to be retrained (or updated). This is necessary because the only information that the tuning system has is the application identifier.

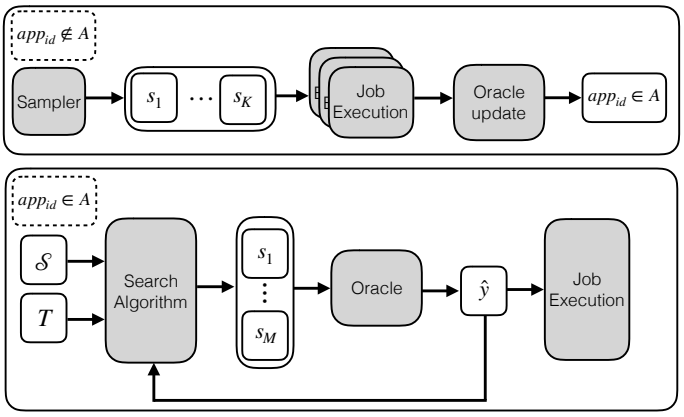


Fig. 2: Classical optimization pipeline.

The question as to which optimization pipeline is better is still open. Works like [3] suggest that by smartly selecting a subset of the configuration space, good results can be achieved without any need of a performance model.

B. Overview of our proposed solution

The drawback of the optimization pipelines in Figures 1 and 2 is that the application needs to be executed multiple times. In practice, the model-based approach needs advance knowledge of the application in order to make sensible predictions. If an application is unknown, several job executions are required to update the oracle. Recent work [2] has studied methods to reduce the number of samples required to update the oracle to around 100 executions, which is still a high number of runs. Since the selection of the parameters depends on the quality of the model, improving the oracle predictions might boost any performance model based search results. In our solution, the oracle is a machine learning regressor that predicts the application execution time based on a feature vector that characterizes its behavior.

In order to mitigate the main drawbacks of the previous methodologies, our work focuses on improving two paramount pieces of performance model based Spark auto-tuning systems. First, in order to improve the generalization of the machine learning models, we built a feature descriptor that characterizes applications without any specific input from the user of the application. This vector is generated by extracting information from the SparkEventLog, which is synthesized in a feature vector $\boldsymbol{x}$. This component of the optimization pipeline can be seen in the upper part of the diagram in Figure 3. It should be noted that, before the oracle is queried, the job is executed (with the default Spark configuration) and $\boldsymbol{x}$ is extracted. This vector is then used to condition the output predictions of the oracle. In Section III, we describe in detail how $\boldsymbol{x}$ is constructed. It is relevant to emphasize that this feature vector contains information from the application that can potentially be transferred to new applications because new workloads might be implemented using a similar type of execution graph, which will yield a similar $\boldsymbol{x}$. Note that the underlying assumption in our methodology is that similar applications should benefit from similar configurations.

The optimization process then follows a simple Bayesian Optimization procedure where, instead of running the application, we query the oracle, providing information about the application with the feature vector $\boldsymbol{x}$.

III. Feature descriptors from Spark workloads

A SparkEventLog is a text file created when a user executes a Spark application. The SparkEventLog is a list of key-value pairs in the form `"key": "value"`, which contain information about the execution. Each element in the list starts with the `Event` key that is used to decide whether the row needs to be parsed or skipped.

A. Feature descriptor from the SparkEventLog

In order to build our feature vector we parse three different types of `Event` keys. The keys we parse are strings of the

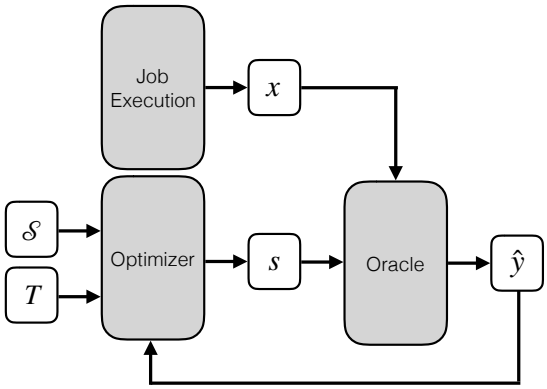


Fig. 3: Our proposed search pipeline. Note that the job is only executed once to extract the workload characteristics. Once the workload characterization $\boldsymbol{x}$ is known, the optimization is performed using the predictions from the oracle.

events that can be constructed concatenating `SparkListener + p + e` where $p \in \{\texttt{Application}, \texttt{Stage}, \texttt{Task}\}$ and $e \in \{Start, End\}$. As the names suggest, `Event` strings are initiated with the suffix `Start` and finished with `End`, marking the beginning and end of the `Event`, respectively. The most important features are retrieved from `Stage` and `Task` events. We only use `Application` events for verifying whether a job starts and finishes without errors.

The resulting descriptor contains a total of 75 features. From those features, 11 are retrieved from Stage events and 64 are retrieved from Task events. We denote by ef (eventlog features) this feature vector of 75 components containing Task and Stage features. A detailed description of the features is presented below.

B. Stage Features

`Stage` events provide information about the Spark actions and transformations used in the Stage. This type of event provides high-level information about the workload because it shows core Spark function calls made during the execution of the workload. Some examples of `Stage` events include the well-known "map", "reduce" and "reduce by key" operations typically used to leverage distributed computing systems. This information tells us the percentage of time an execution spends on each of the actions and transformations found in the workload. The descriptor contains the actions and transformations from the following list: ["collect", "count", "countByKey", "first", "flatMap", "map", "reduce", "reduceByKey", "runJob", "takeSample","treeAggregate"].

Note that a workload might only use a subset of the previous features.

C. Task Features

`Task` events provide information about the different Tasks found in a Stage. Tasks provide us with low-level metrics such as the number of bytes read and written to disk, the time spent during garbage collection and the CPU time. We aggregate the information across all tasks into a vector of aggregated statistics. This information tells us whether, overall, the execution was read or write-intensive, CPU intensive, etc. Table I lists all the features gathered from Task events.

	Task features
0	Input Metrics: Bytes Read
1	Executor Deserialize Time
2	Executor Deserialize CPU Time
3	Executor Run Time
4	Executor CPU Time
5	Result Size
6	JVM GC Time
7	Result Serialization Time
8	Memory Bytes Spilled
9	Disk Bytes Spilled
10	Shuffle Read Metrics: Remote Blocks Fetched
11	Shuffle Read Metrics: Local Blocks Fetched
12	Shuffle Read Metrics: Fetch Wait Time
13	Shuffle Read Metrics: Remote Bytes Read
14	Shuffle Read Metrics: Remote Bytes Read To Disk
15	Shuffle Read Metrics: Local Bytes Read
16	Shuffle Read Metrics: Total Records Read

TABLE I: Task features.

Since there can be an arbitrary number of Tasks in an SparkEventLog, we generate four feature vectors containing the mean, minimum, maximum and standard deviation across all the features in Table I. The final descriptor summarizes the behavior over multiple tasks and is created by concatenating the previous four feature vectors. This process creates a $68 = 17 \cdot 4$ dimensional feature vector. Finally, we add a feature "dataset size" that contains the total data size read from disk. This feature is extracted by summing the values in "Input Metrics: Bytes Read" across tasks.

SparkEventLog example: The example given in this subsection shows different parts of the SparkEventLog created when a job is executed. Our goal is to show the correspondence between the text from the SparkEventLog and the features described in Section III-B and III-C. To make the explanation clearer, some parts of the SparkEventLog have been omitted. The snippets of text shown below correspond to the SparkEventLog generated executing an Fp-growth algorithm. This is a popular data-mining application that generates association rules.

- `ApplicationStart` and `ApplicationEnd` mark the start and end of the application. In this example, we can see that the `AppID` is `local-1561554220820`. This string is precisely the name given to the SparkEventLog generated when running this application. If an application does not crash, it will have a `SparkListenerApplicationEnd` event at the end of the list.


```
{"Event":"SparkListenerApplicationStart",
"App Name":"05_fpgrowth_itemset_mining.py",
"App ID":"local-1561554220820","Timestamp":1561554312331,"User":"dbuchaca"}
```

- `StageSubmitted` and `StageCompleted` denote the start and end of a Spark Stage, respectively. In the example below, we can see the field `Stage Name`, which takes the value `count`, one of the features from III-B. The word `count` is a Spark keyword that describes the type of action or transformation being performed in the stage (in this case, Stage 0).


```
{"Event":"SparkListenerStageSubmitted
"Stage Info":{"Stage ID":0,"Stage Attempt ID":0,
"Stage Name":"count at FPGrowth.scala:217","Number of Tasks":2,
"RDD Info":[{"RDD ID":4,"Name":"MapPartitionsRDD",
Scope":"{\"id\":\"2\",\"name\":\"map\"}",.... }]
```


- `TaskStart` and `TaskEnd` denote the start and end of the task, respectively. In the snippet provided, we can see several metrics that tell us the read and write statistics, garbage collection metrics, etc. The information found here can be used to fill in the features in Table I.


```
["Event":"SparkListenerTaskEnd","Stage ID":0,"Stage Attempt ID":0,
"Task Type":"ResultTask", "Task End Reason":{"Reason":"Success"},...
"Task Metrics":{"Executor Deserialize Time":33,
"Executor Deserialize CPU Time":19157489,
"Executor Run Time":647,"Executor CPU Time":147466180,"Result Size":1569,
"JVM GC Time":0,"Result Serialization Time":1,"Memory Bytes Spilled":0,
"Disk Bytes Spilled":0,"Shuffle Read Metrics":{"Remote Blocks Fetched":0,
"Local Blocks Fetched":0,"Fetch Wait Time":0,"Remote Bytes Read":0,
"Remote Bytes Read To Disk":0,"Local Bytes Read":0,"Total Records Read":0},
"Shuffle Write Metrics":{"Shuffle Bytes Written":0,"Shuffle Write Time":0,
"Shuffle Records Written":0},"Input Metrics":{"Bytes Read":125875,
"Records Read":304}, "Output Metrics":{"Bytes Written":0,"Records Written":0},
"Updated Blocks":[]}}]
```


IV. Simulated Bayesian Optimization

Using an optimization procedure that requires executing Spark jobs to evaluate Spark configurations can be a slow and expensive process. In order to avoid executing a program many times with different configurations, our work proposes a "Simulated Bayesian Optimization" (SBO) search. To speedup the optimization process, SBO replaces costly executions of the applications with the expected results provided by an oracle. The oracle is provided with a characterization vector of the workload $\boldsymbol{x}$ in order to adapt the predictions of the configurations to the workload to be optimized. The inputs to the SBO procedure from Algorithm 1 are:

- h: Machine learning model already trained with access to a method $h.predict$ that can be used to predict the execution time from an input vector (s, x). Here, s is a vector containing the SparkConfiguration values and x is a vector of features that characterizes the workload.
- Opt: Class of optimization procedure used to instantiate an optimizer opt. We use a Bayesian Optimization process that has access to two methods. The method $opt.generate()$ creates a configuration s. The method $opt.update(s, \hat{y})$ updates the internal Bayesian Optimization process for the configuration s with value $\hat{y}$.
- job: Workload to be optimized by SBO. Note that we assume we have access to a function $run(job; s)$ that runs the workload and returns the execution time and the SparkEventLog of the workload.
- R: Maximum number of runs allowed to our method.
- $\mathcal{S}$: Search space definition. The space is used to provide boundaries for configurations $s \in \mathcal{S}$.
- s_0: Default Spark configuration.
- T: Maximum Time allowed for the optimization process. Note that even if the total number of runs does not reach the maximum R, the optimization process can be stopped if it reaches a maximum time T.
- ϕ: Feature descriptor described in Section III. This function generates a feature descriptor for the current application using the default configuration.

Algorithm 1 is a simplified version of the SBO procedure which keeps only the best configuration in memory.

Algorithm 1

procedure SBO(h, Opt, job, R, $\mathcal{S}$, s_0, T, ϕ)
  $t \leftarrow 0$
  $opt \leftarrow Opt(\mathcal{S})$ ▷ Optimizer is created
  y_0, SparkEventLog $\leftarrow run(job; s_0)$
  $y^* \leftarrow y_0$
  $s^* \leftarrow s_0$
  $x \leftarrow \phi$(SparkEventLog)
  $opt.update(s_0, y_o)$ ▷ Optimizer is updated with s_0
  for $r \in 1 \dots R$ **do**
    $s \leftarrow opt.generate()$
    $z \leftarrow (s, x)$
    $\hat{y} \leftarrow h.predict(z)$
    $opt.update(s, \hat{y})$ ▷ Optimizer is updated with s
    if $y < y^*$ **then**
      $y^* \leftarrow y$
      $s^* \leftarrow s$
    $t \leftarrow t + get\_time()$
    if $t >= T$ **then**
      break
  return s^*

Once the algorithm finishes, it returns the estimated best configuration s^*. To ensure that s^* is a good choice, we need to run the workload with the provided configuration. Therefore, a reasonable way to implement this algorithm involves storing the best K values found during the optimization process. Then the process would return $s_1^*, \dots, s_K^*$, which should be run to find which is the best configuration. We refer to this process as SBO-K. For example, if a single configuration is returned and executed, we would say that it followed an SBO-1 optimization procedure. In the event that SBO-K returns a solution that is worse than the default configuration, the application should be flagged and the model h should be updated with training samples of that application. In that case the default configuration should be returned.

The previous explanation describes SBO in general terms. The following Section V includes a detailed description of our selection process for the different inputs involved in the SBO algorithm, as well as the Spark applications used to evaluate our methodology. In particular, the election of regressor h (as well as the tuning of the model) is described in Section V-D. The search space S, the Default configuration s_0 and the maximum number of executions allowed R can be found in Section V-E. We did not specify a maximum time budget T.

V. Experiments

We evaluated the effectiveness of our proposed solution by comparing the results with popular state-of-the-art approaches. In particular, we aimed to answer the following research questions:

- **RQ1**: Do the proposed features help to improve predictions based on machine learning models? Does an oracle

trained with the SparkEventLog features predict sensible configurations for unseen Spark workloads?
- **RQ2**: Does the overall methodology provide a good configuration when compared with a solution found with a well-known optimization process, such as a Bayesian Optimizer? Does the process provide a good configuration in less time?

The answer to **RQ1** was obtained by comparing our results with other feature representations found in similar works [2], [8], [11], [16]. or **RQ2**, we needed to measure the quality of the results and the time-to-solution of our pproach with respect to a Bayesian Optimization process.

A. Dataset Description

To evaluate our system we decided to use a set of workloads that cover different popular uses of Spark. The set of workloads we used is based on two benchmarks. First, we included workloads found in HiBench [5], a popular benchmark for Spark, which includes micro benchmarks, as well as workloads from machine learning, data mining, natural language processing and web-search. Second, we also included benchmarks from *spark-perf* [6], a benchmark created by DataBricks, the company founded by the original creators of Spark. Table II shows the different workloads used.

Id	Workload name	Application type
01	n-gram	NLP
02	Logistic Regression	Machine Learning
03	Support Vector Machine	Machine Learning
04	Pi computation	Scientific Computing
05	Fp-growth	Data Mining
06	Word Count	NLP
07	K-Means	Data Mining
08	PCA	Data Mining
09	GaussianMixture	Machine Learning
10	Pagerank	Graph Processing
11	Random forest	Machine Learning
12	Databricks K-Means	Data Mining
13	Databricks Naive Bayes	Machine Learning
14	Databricks Pearson Correlation	Statistics
15	Euler computation	Scientific Computing

TABLE II: Applications used for the experiments.

We used the previous benchmarks to build a dataset containing executions of Spark jobs. To build our dataset, we generated 100 combinations of parameters for each workload, as other works [3] have done. We used a Bayesian Optimization process to generate the samples for our dataset. The main reason for using a BO process instead of a random search is to minimize bad quality examples and the number of failed runs. We did try a random process to build a dataset, but it generated many configurations that, once run, returned memory errors or were slower than the default configuration. Since the execution time of an application with a particular SparkConfiguration can have some variance, we ran the jobs five times and stored the mean over the repetitions. This value was used as input to the Bayesian Optimizer during the dataset creation.

B. Computational Environment

All of the experiments described in this section used a 4 node cluster, with Spark version 2.4.1 configured in cluster mode, with one node as master and three as slave nodes. Each node had two 10-core Intel®Xeon E5-2630 v4 CPU @ 2.20GHz, which sums up a total of 40 threads per node, as the hyper-threading was active. Each node also had 128 GB of RAM and they were interconnected by a 10Gbps Ethernet network. In order to make a fair and easy to understand comparison the executions for **RQ2** are evaluated in containers executed with exclusivity of resources. In other words, a single Spark application is being executed for each Spark configuration tested. It is out of the scope of our work, but an interesting area of research, to generalize the oracle for environments with co-executions as well as heterogeneous hardware.

This happens because, among other things, there is a trade-off between how many executor cores can be run and the available memory for each executor core.

We used scikit-learn [17] to build the different performance models presented in this section.

C. A note on the distances between feature descriptors

The SBO algorithm proposed uses a feature vector $\boldsymbol{x}$ that is built after an application is run using the Default Spark configuration. A natural question that may arise is: how well the information summarized in the feature vector describes similar feature vectors generated with other Spark configurations for the same application. Note that the descriptor is not invariant with respect to the Spark configuration provided when running the application. This happens because several of the metrics that are gathered (such as the aggregated executor CPU usage across tasks) are influenced by configuration decisions (such as the number of executor cores).

In Section V-D, we show that the descriptors boost the performance of the learning algorithms. Nevertheless, another reasonable approach to assess the quality of a descriptor is to investigate whether the vector for a particular job is closer to vectors from the same Spark workload than to vectors from very different workloads. Let us denote by $\boldsymbol{x}^{(0,w_j)}$ the descriptor for workload w_j generated using the default Spark configuration, which we will call the default feature vector for application w_j. Let us denote by $\boldsymbol{x}^{(k,w_i)}$ the feature vector (for application w_i) generated using the k-th Spark configuration sampled from the training data.

Let us denote by M a matrix containing at position $M_{i,j}$ the mean of the euclidean distances between $\boldsymbol{x}^{(0,w_j)}$ and the vectors $\boldsymbol{x}^{(1,w_i)}, \ldots, \boldsymbol{x}^{(100,w_i)}$. That is,

$$M_{i,j} = \frac{1}{100} \sum_{k=1}^{100} \left\| \boldsymbol{x}^{(0,w_j)} - \boldsymbol{x}^{(k,w_i)} \right\|_2 \tag{3}$$

This matrix contains at column j the average distance between the default feature vector of application w_j and all the feature descriptors of application w_i for $i \in \{1, \ldots, 15\}$.

Spark Configuration Parameters	Search Space	Space Encoding	Default Configuration
spark.task.cpus	[1, 2]	Integer	1
spark.executor.cores	[1, 40]	Integer	40
spark.executor.memory	[4, 40]	Integer	20g
spark.memory.fraction	[0.4, 0.5, 0.6, 0.7, 0.8]	Ordinal	0.6
spark.memory.storageFraction	[0.3, 0.4, 0.5, 0.6, 0.7, 0.8]	Ordinal	0.5
spark.default.parallelism	[20, 400]	Integer	40
spark.shuffle.compress	[true, false]	Categorical	true
spark.shuffle.spill.compress	[true, false]	Categorical	true
spark.broadcast.compress	[true, false]	Categorical	true
spark.rdd.compress	[true, false]	Categorical	false
spark.io.compression.codec	[lz4, lzf, snappy]	Categorical	lz4
spark.reducer.maxSizeInFlight	[24m, 48m, 96m]	Categorical	48m
spark.shuffle.file.buffer	[32k, 64k]	Categorical	32k
spark.serializer	[JavaSerializer, KryoSerializer]	Categorical	JavaSerializer

TABLE III: Search Space of the different parameters we tuned with the default values. The units in the Default Configuration correspond to the notation used by Spark. For example, 20g corresponds to 20 GB.

We observed that the matrix has a diagonal containing most of the lowest values in each column. This shows that the default descriptor for application w_j is generally closer to the vectors for the same application $\boldsymbol{x}^{(k,w_j)}$ than to other vectors from different applications $\boldsymbol{x}^{(k,w_i)}$.

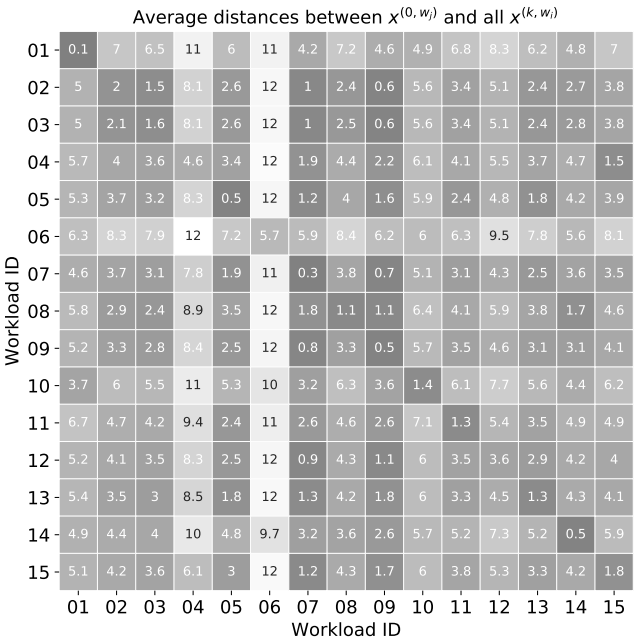


Fig. 4: Matrix showing the average distance between the default descriptor of application j and all the descriptors of application i. Position (i, j) contains the value computed by Equation 3.

A manual inspection of M reveals some interesting results:

- Some applications are similar in their types and feature descriptors. For example, the smallest value in column 2 is $M_{2,2} = 2$. This means that the default feature vector from application 2 is at an average distance of 2 from all the vectors of the same application. Moreover, the second lowest value in column 2 corresponds to $M_{3,2} = 2.1$. This is reasonable, since applications 2 and 3 correspond to a Logistic Regression and a Support Vector Machine respectively; both of these workloads share a very similar iterative training algorithm.
- Some applications are dissimilar in their workload types and feature descriptors. For example, the smallest value in column 6 is $M_{6,6} = 5.7$. This means that the default feature vector from application 6 is at an average distance of 5.7 from all the vectors of the same application. Moreover, the second lowest value in column 6 corresponds to $M_{14,6} = 9.7$. In this case, this suggests that applications 6 and 14 are not very similar. This is reasonable, because applications 6 and 14 are a Word Count and a Pearson Correlation test, respectively.

D. *Experiments for **RQ1***

RQ1 is critical for our system. If the oracle is not able to provide sensible predictions for new, unseen workloads, then the selection process based on the values given by the oracle will be flawed. Moreover, our work requires that the features extracted from the SparkEventLog generalize to new workloads, otherwise the models cannot be used for the SBO process.

The goal of this experiment was to assess the quality of the features described in Section III with respect to standard feature representations of workloads found in many state-of-the-art works [2], [8], [11], [16]. The most common feature representations of the workloads combine three sources of information: the Spark configuration (sc), the dataset size (ds) and an identifier for the application (app_{id}). Works like [8] use (sc, ds), whereas [11], [16] use (sc) as input and [2] uses (sc, ds, app_{id}). Our work uses (sc, ds, ef) as the feature vector, where (ds, ef) corresponds to the "SparkEventLog features" described in Section III.

Data Processing: In order to input the data to the machine learning models, we performed some preprocessing on the raw

data. The types of variables involved in the Spark Configuration are shown in Table III. The heterogeneity of the variable types required some preprocessing in order to build the sc vector. In particular, categorical variables were converted using a one-hot encoding transformation. The remaining variables were then treated with either Rescaling or Standardization. Rescaling was performed on the [0,1] range and Standardization transformed the variables so that the values have zero-mean and unit-variance.

The models may perform differently depending on whether Rescaling or Standardization is applied. We implemented a pipeline mechanism to automatically decide during cross-validation which preprocessing was best for each model, so as to adapt the decision to each of the learners. The data transformation was introduced as one of the steps during model selection, as if it were a hyper-parameter of the model.

Model Evaluation: We created 15 test sets by splitting the data 15 different ways. The final test set metrics provided in this section show the average of the results for a model with a fixed set of hyper-parameters over the 15 test sets. The 15 splits were created using the application identifier in order to evaluate models on samples of a workload that did not appear during training. This methodology of generating partitions is usually referred to as Leave One Group Out Cross-Validation (LOGO-CV). Using this methodology we ensure that models are evaluated on samples of a workload that had never been seen during training (here the "group" consists on all the examples that contain the same workload identifier). We chose LOGO-CV, instead of standard Cross-validation, because it more closely resembles the scenario we want our model to generalize. In a production environment, we would like a system that can provide good predictions for workloads that have never appeared in the past. In general, we do not want to assume that the training and validation splits contain examples of all possible workloads, because in a production environment this scenario is unlikely.

After deciding which algorithm and hyper-parameters are the best ones, the error values computed using LOGO-CV reveal how well (on average) a model should perform on a new workload never seen during training.

Model selection: We trained 8 machine learning models using 5 different sets of features. In order to select the models robustly, we performed LOGO-CV on each of the 15 splits of our data. The average errors across all of the validation partitions of the different splits were used to select the best oracle candidate.

We tested the following combinations of features: $(sc), (sc, app_{id}), (sc, ds), (sc, ds, app_{id})$ and (sc, ds, ef) as input to the learning algorithms. The last combination of features corresponds to our proposed solution. In order to make a fair comparison between the different features found in the literature, we performed all the experiments with the same space of hyper-parameters for the machine learning models. The search space is given in Table IV. An exhaustive exploration of all the combinations of hyper-parameters was performed.

Training each of the models from Table IV, including hyper-parameter optimization, takes less than one minute in a single node of our computation environment. Therefore, the overhead of training a model in our system is negligible with the dataset previously described.

Model	Hyper-parameter search space
Elastic Net	alpha = [1e-10, 1e-9, ..., 10.0]
GBM	learning_rate = [0.001, 0.01, 0.1, 0.5]
	max_depth =[3, 10, 20, None]
KNN	n_neighbors= [1,2,3,4,5,6], p=[1,2]
Lasso	alpha = [1e-10, ..., 1.0, 10.0]
MLP	hidden_layer_sizes = [[200], [300],[500],
	[200, 200], [300, 300], [500, 500]]
	activation = ["tanh","relu"]
	learning_rate = [0.0001, 0.001, 0.01]
Random Forest	max_depth = [10, 20, None]
	max_features: ["auto", "sqrt"]
	n_estimators: [10, 20, 50, 100, 200]
Ridge	alpha = [1e-10, 1e-9, ..., 10.0]
SVR	C = [1e-10, 1e-9, ..., 10.0]
	kernel = ["linear", "poly", "rbf"]

TABLE IV: Hyper-parameter search space.

Results: The results of the experiments with the different models are given in Table V, which contains the training and validation Mean Squared Errors (MSE) for the best parameters found for each model. The table is organized in five column groups, one for each set of features tested. Each row in the table shows the results for a particular model containing the best hyper-parameters found during training. The results in the table show that the best results in the validation were achieved with the descriptor (sc, ds, ef). It can also be seen that with input features (sc, ds, ef), the Random Forest Regressor achieved the best results in the validation data. Therefore, we chose the Random Forest (with the features (sc, ds, ef)) as our oracle for the experiments in Section V-E. The selected Random Forest was trained with the hyper-parameters that minimize the error during hyper-parameter exploration.

After the best hyper-parameters for each model had been found, we evaluated each model across the test sets. Table VI provides the test set results for the models selected from Table V for each of the different feature groups. Note that (sc, app_{id}) has no Reference because we could not find any work using that feature vector as input to the learning algorithms. Nevertheless, we have included it for completeness because it is a sensible approach that improves the results over sc.

Features	Test MSE	Improvement over sc	Reference
(sc)	167.9	0.0%	[11], [16]
(sc, app_{id})	162.0	3.5%	-
(sc, ds)	155.6	7.3%	[7], [8]
(sc, ds, app_{id})	158.4	5.6%	[2]
(sc, ds, ef)	110.7	34.1%	ours

TABLE VI: Improvement over the basic sc feature set.

Features	(sc)		(sc, app_{id})		(sc, ds)		(sc, ds, app_{id})		(sc, ds, ef)	
model	train	validation	train	validation	train	validation	train	validation	train	validation
Elastic Net	161.2	198.4	69.4	204.5	153.0	193.7	67.4	199.5	50.3	167.9
GBR	85.6	**178.6**	14.3	**164.2**	50.9	195.3	12.4	**165.1**	2.8	173.7
KNN	117.7	224.5	47.3	224.5	106.5	211.1	44.1	211.1	8.7	227.6
Lasso	161.0	196.2	63.4	205.2	152.4	192.7	61.1	214.9	29.2	157.2
MLP	194.4	211.5	190.3	211.2	194.6	210.5	194.4	210.7	17.2	371.6
Random Forest	17.2	183.5	5.7	178.8	12.2	**175.5**	4.6	179.0	0.8	**132.6**
Ridge	159.2	202.6	61.2	212.3	150.5	202.8	57.9	236.2	20.7	272.0
SVR	167.5	209.6	73.5	213.0	158.0	207.8	73.1	219.8	31.1	258.5

TABLE V: Training and validation MSE errors for the different models. The proposed features (sc, ds, ef) are contextualized with different sets of features found in the state-of-the-art work.

E. Experiment for **RQ2**

To validate **RQ2**, we needed to measure two metrics. First, the quality (runtime) of the solution provided by SBO in comparison with the best configuration found by BO. Second, the time required by SBO to provide a configuration with respect to the time needed by BO.

Default Settings: To assess the quality of a SparkConfiguration found by BO or SBO, we compared the execution time of the provided configurations with the execution time achieved by the default configuration. This quantity is the speedup with respect to the default SparkConfiguration. The default configuration we used is given in Table III. Note that this configuration is the same used by Spark by default, except for the number of executor cores and the amount of executor memory. We set the number of executor cores to 40 (the total number of cores per node in our cluster) and the executor memory to 20 Gigabytes ("20GB"). The memory was increased to avoid job failures for the workloads with the default Spark configuration. This is important because in the event of an everlasting runtime for a job, the speedup with respect to the default configuration could theoretically be infinite, providing unrealistic speedups with respect to a bad default baseline. Changing the memory settings is a common decision also found in other works such as [8]. In our case, eight of the fifteen jobs failed using the default Spark configuration, which uses $1GB$ of executor memory. To prevent this issue arising, we tested the default configuration with the executor memory set to $1GB, 2GB, \ldots, 50GB$ and stopped at the first value able to run all workloads without any memory crash, which was $20GB$.

Experiments: The objective of the experiments was to measure the quality of the solutions found by SBO with respect to the best solution found with BO. Here we will use BO-K to denote a Bayesian Optimization process that executes an application K times. In order to make a useful comparison, all the results were evaluated on workloads that neither BO nor SBO had accessed before starting the optimization procedure. In the case of BO, we set the budget to 20 iterations because it is the minimum number of iterations needed to reach a speedup over the default configuration across the workloads we used. As other works, such as [3], use 35 or 100 iterations as maximum budget, we decided to perform BO-100 as a reasonable upper bound on the quality of the results. In the case of SBO, we allowed a budget of 100 queries to the oracle (which takes less than two seconds). Then, only one or two evaluations of the workload were executed (which we refer to as SBO-1 and SBO-2, respectively), because the goal of our methodology is to quickly find solutions with a reasonable quality.

Speedup over default configuration results: The improvement of SBO with respect to the default configuration is shown in Figure 5a, which contains the runtime of the default configuration and the runtime of the configuration found with SBO-1 and SBO-2. The figure also shows a box plot of the distribution of runtimes found during the Bayesian Optimization process performed to create the dataset. We can see that SBO-2 improved on the default configuration in all cases, except for workload 11, where there is no speedup. It is interesting to notice that workloads 1, 10 and 14 need SBO-2, since SBO-1 does not improve the quality of the results. Nevertheless, such cases are precisely the examples where the box plot shows little variance in the runtime. In those cases, even BO-100 was not able to achieve a significant speedup with respect to the default Spark configuration. The box plot shows that workloads 10 and 14 have most runtimes clustered around a single point. This suggests that those cases are very hard to optimize since a BO with 100 executions of the application was not able to improve the results.

The results of the speedups found by BO-20, BO-100, SBO-1 and SBO-2 with respect to the default configuration runs are shown in Figure 5b. Note that BO achieved higher quality solutions than SBO-1 in almost all cases, except for 12. Nevertheless, the difference between BO and SBO-2 is minimal. It should also be noted that SBO-1 always achieves better results than random search, even if we perform 10 random searches. Nevertheless, the random search can achieve surprisingly good results, which is consistent with the

experimental results observed in other literature [3].

Time-to-solution: Table VII gives the time-to-solution speedups of our proposed method with respect to the BO process with 20 iterations (BO-20). The results shown in the table include the time spent running the Spark configurations, as well as the time needed to parse the SparkEventLog and the time required by the oracle to make predictions. Moreover, the table includes the total search time for the different algorithms tested. Our methodology allows us to find a good solution in 5 minutes (on average) using SBO-2. A naive solution, such as random search, takes almost half an hour to run, while achieving worse results than SBO-2, as can be seen in Figure 5b. BO-20 can achieve, in some cases, better configurations than SBO-2, but with a higher average cost of 53 minutes. Figure 5c shows the total search time for the different methods across applications. In Table VII we can see that the SBO-1 optimization process is 25 times faster than the BO-20 process. At first glance, it might be surprising that the SBO-1 is 25 times faster than BO-20, which performs only 20 iterations. The explanation for this behavior is that, some of the runs performed by a Bayesian Optimizer may include bad configurations that are executed during the optimization procedure, which have to be sampled in order to explore the search space and provide a good exploration-exploitation trade-off. In the case of SBO-1, we only executed the best configuration predicted by the oracle and none of them was bad. This allows the average speedup of SBO-2 to be up to 12x faster than BO-20.

	Speedup over BO-20	Search Time (min)
BO-20	$1.0x$	53.6
BO-100	$0.2x$	243.2
Random-10	$1.9x$	28.6
SBO-1	$25.1x$	2.6
SBO-2	$12.4x$	5.0

TABLE VII: Time-to-solution for the different methods. The first column shows the improvement with respect to 20 runs of BO. The second column the overall search time in minutes.

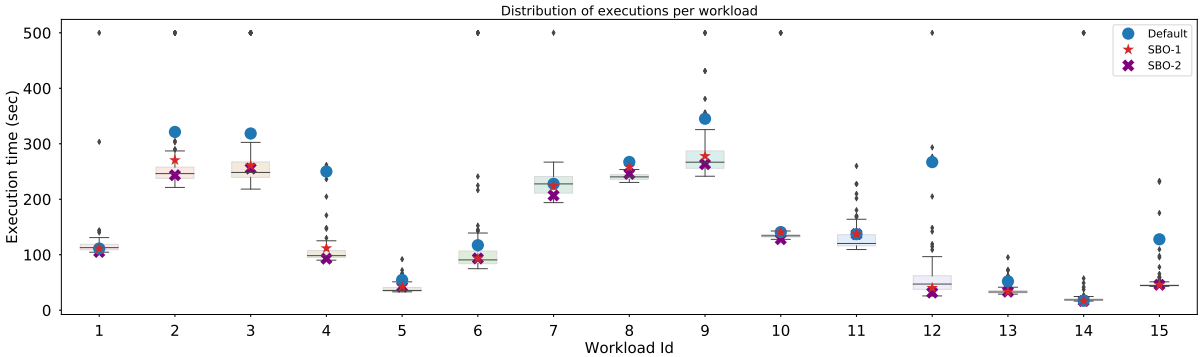
VI. Related Work

Auto-tuning is a popular research field with a wide variety of solutions that target Big Data technologies [3], [12], [14], [18]–[24]. In particular, tuning Spark workloads is an important concern in the Big Data community that has worked on different approaches to improve current solutions. Different surveys on automatic parameter tuning for Spark such as [21] and [22] classify the available techniques into six categories: rule-based, cost modeling, simulation-based, experiment-driven, machine learning (or model-based), and adaptive tuning. **Rule-based** is the most simple approach as only relies on guidelines/recommendations for each configuration, however , it brings complexity as it requires a deep understanding of the system internals and the workloads to match perfectly the recommendations. Such systems, typically rely on human experience and experts knowledge from guides such as [25]. **Cost models** use an analytical cost function to predict the workload performance [26]. Such techniques do not have the capability to generalize to new applications or hardware environments. **Simulation-based** techniques do not tune directly Spark configurations, but instead try to capture various factors that impact the system performance to provide estimates on given configurations [27]. In the **experiment-driven** approaches, such as in [19], many different runs of the application with different parameter settings are performed until convergence. The **machine learning** approach, also known as performance model based, is the one followed by our work and by many others such as [8], [11]–[15]. Such works investigate a wide range of input features used to train the model, which we have evaluated in **RQ1**. The **adaptive-approach**, used in [20], relies on past executions to dynamically tune an application while running.

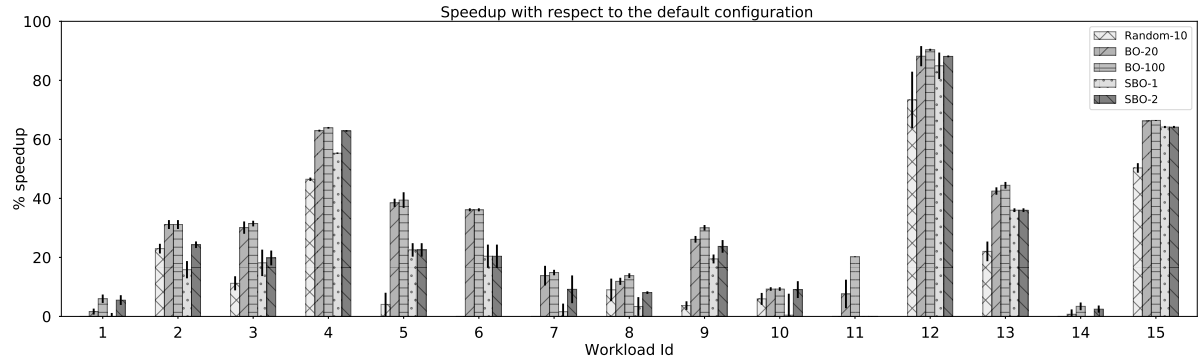
The rest of this section presents a summary of related work, focused mostly on performance model based auto-tuning methods. A brief description of the features used in the models is provided to facilitate the comparison of other methods found in the literature with our work.

Improving the auto-tuning pipeline: The authors of [4] focus on an orthogonal approach for improving the efficiency of model-based solutions for auto-tuning highly configurable software. This work focuses on building a simulator for the process from which they want to learn. Therefore, this work essentially builds a model that generates samples so as to avoid running workloads and, thereby, reduces the overall cost of the auto-tuning system. In our setting, this work could be applied to generate Spark configuration and execution time pairs. This could be achieved using a simulation mechanism, instead of actually running applications.

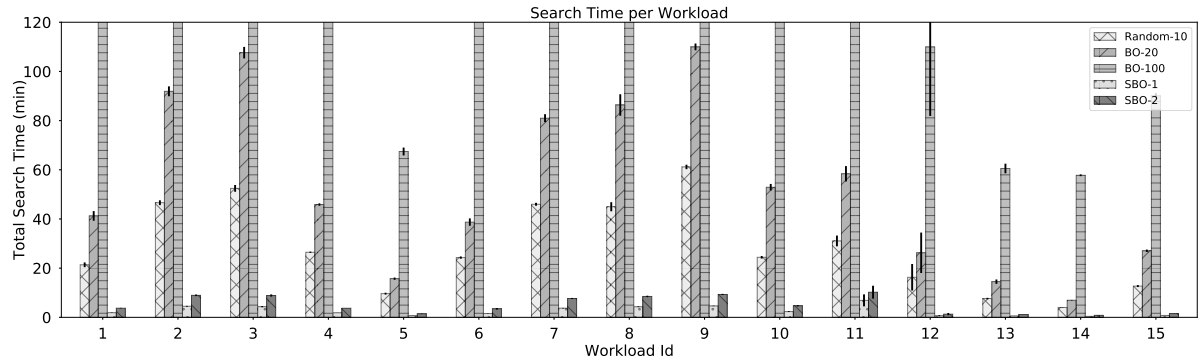
Auto-tuning without a performance model: Expert knowledge is used in [28] to select a relevant set of hyper-parameters to be studied for each workload. The performance of the different applications is evaluated under a discretized set of values for the selected parameters. This approach requires experts to determine which hyper-parameters have to be tuned, making the solution impractical for a general-purpose system that aims to tune different types of applications with different needs. A straightforward approach to improve the previous solution relies on optimizing via trial-and-error [29]. In this work, authors tune 12 parameters from the Spark configuration for three applications. For each application, a parameter is selected and several values of the parameter are tested. If there is a large gain with respect to the default configuration, the parameter is considered as significant to the overall performance. With this approach, a graph is generated with the most important parameters as nodes. Once an application is presented it is supposed to be run for all the nodes in the graph and the parameters that maximize performance are selected as the best parameters. In [3], a significance-aware tuning system is presented with the goal of achieving fast time-to-solution. This work uses a Bayesian Optimization process to find good configurations for the workloads. The main novelty of this work is that the system can identify a relevant subset of parameters to tune, which greatly reduces the search space. Nevertheless, this system does not gather log data to speed up the process of learning which are the relevant parameters for



(a) Best solutions found by SBO-1 and SBO-2. The box plots how the distribution of the execution times found by a BO process with a budget of 100 runs per application.



(b) Speedup with respect to the default configuration per application (the higher the better). An empty bar reflects no improvement over the default configuration.



(c) Total search time, in minutes, to find a configuration for each method (the lower the better). The plot was cropped at 120 min to improve readability.

Fig. 5: Experiments for RQ2.

an incoming job, requiring at least 20 runs to select a good configuration.

Auto-tuning with a performance model trained on a subset of $\{sc, ds, app_{id}\}$: Most works that optimize Spark configurations are based on performance models using a subset of features from $\{sc, ds, app_{id}\}$ (which we have defined in Section V-D). Works like [11] uses sc as input to train different learning algorithms for each application that the systems aim to tune. In this work, special focus is given to constructing the dataset using a Latin Hypercube Sampling algorithm. Some works, such as [7], [8], [14] or [13], use (sc, ds) as input to the learning algorithm. In [7] a Genetic Algorithm is used to perform the search, instead of a Bayesian Optimization process. Unlike our work, the proposed solution in [8] learns a model per workload and uses Recursive Random Search to find a good Spark configuration. The genetic search is based on the estimates of the execution time predicted by a performance model. A geometric interpolation method mixed with a sampling strategy to build a faster and accurate model by a small number of historical job executions was introduced by [13]. Finally, a big training dataset created by exhaustive search and random exploration was the focus of [14], resulting

in a 1500 training points for every four workloads explored with several common machine learning models.

ALOJA-ML [2] uses (sc, ds, app_{id}) as input to the learning algorithms. This work uses a single performance model that is shared across applications. The main downside is that all of the application-related information is captured in a single categorical variable that is provided to the model. Therefore, unseen applications do not provide any information about the nature of the workload and retraining is required in order to search good parameters for new workloads.

Auto-tuning with a performance model which leverages an application characterization to aid the search: [15] and [30] are two publications in the area of Spark auto-tuning that proposes more similar systems to our work, generalizing unseen workloads by some characterization of the application. [15] approach to extract the features is more close to our work as it capture Task and Stage information from an Spark application, while [30] follows a more general approach by profiling the application to extract statistics like the average CPU and disk usage that can work with other Big Data frameworks rather than Spark.

In [30], a vector of statistics is extracted from the runtime of the application to be optimized. This feature vector $\boldsymbol{q}$ is used to modify a BO procedure to guide the search process. This work presents a Guided Bayesian Optimization process (GBO) that uses an slight modification of the Expected Improvement function in the BO optimizer which receives as input the monitored metrics $\boldsymbol{q}$. Our work shares the core idea of providing specific knowledge about the running application to improve the search process. The main difference is that, in [30], the authors do not try to proxy application executions with metrics provided with an oracle.

To the best of our knowledge, [15] is the only publication in the area of Spark auto-tuning that proposes a system that generalizes unseen workloads by making use of features that capture Task and Stage information from an application. This work builds a single model that is trained by taking as input a descriptor from each of the stages of a Spark application. After learning, once an application is selected to be optimized, the model is queried for each of the stages in the application and for each of the configurations that the system is enabled to tune. Then the final predicted execution time is the sum for the predicted times of the stages. Once all predictions have been made for all the allowed configurations, the minimum sum is selected as the best one. Note that the predictions are made for all the allowed configurations, making this approach impractical for auto-tuning a big space of configurations, such as the one we propose. In [15], only two parameters are tuned and the overall search space contains only 12 possible combinations of the two parameters.

Other related Auto-tuning works: The idea of using information from logs is not completely new in some related areas. For example, in database management system (DBMS), tools like OtterTune [18] use Gaussian Process (GP) to recommend suitable parameters (or "knobs", as more known in DBMS) for different workloads by extracting the internal state of the database to reflect the workload characteristics. This work uses statics collected from the amount of data written or read as well as the time spent waiting for resources.

VII. Conclusions

This paper introduced a Spark feature descriptor that is built with metrics extracted from a single run of a Spark application. We built this descriptor by parsing the SparkEventLog file generated automatically by Spark after a single run of an application. This process produces a feature vector that encodes relevant information from the Spark application without requiring any human intervention. Our experiments show that this information significantly benefits the quality of the machine learning models built on top of it, given that it has been trained with reasonable coverage of Spark applications, improving the prediction by up to 34% with respect to the same models trained on state-of-the-art features.

Moreover, we introduced a simple, easy to implement, modification of a Bayesian Optimization (BO) process that we called "Simulated Bayesian Optimization" (SBO). Using this algorithm, in conjunction with the performance models built on top of the proposed Spark feature descriptor, we implemented an optimization pipeline that can be used for auto-tuning Spark workloads. Our experiments show that SBO can speed up the optimization process considerably with respect to the standard BO approach, up to a $12x$ speedup with results of almost equal quality.

Acknowledgment

The authors would like to thank Asser Tantawi, Tonghoon Suk and Alaa Youssef for the helpful discussions during the development of this work. Funding: This work is supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement n^{o} 639595); the Spanish Ministry of Economy, under contract TIN2015-65316-P, and the Generalitat de Catalunya under contract 2014SGR1051; the ICREA Academia program; the BSC-CNS Severo Ochoa program (SEV-2015-0493); and by Petroleo Brasileiro S.A. (PETROBRAS).

References

[1] "Spark Configuration," 2020. [Online]. Available: http://spark.apache.org/docs/latest/configuration.html

[2] J. L. Berral, N. Poggi, D. Carrera, A. Call, R. Reinauer, and D. Green, "ALOJA-ML: A framework for automating characterization and knowledge discovery in hadoop deployments," Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, vol. 2015-Augus, pp. 1701–1710, 2015.

[3] A. Fekry, L. Carata, T. Pasquier, A. Rice, and A. Hopper, "Tuneful: An Online Significance-Aware Configuration Tuner for Big Data Analytics," pp. 27–29, 2020. [Online]. Available: http://arxiv.org/abs/2001.08002

[4] P. Jamshidi, M. Velez, C. Kästner, N. Siegmund, and P. Kawthekar, "Transfer Learning for Improving Model Predictions in Highly Configurable Software," Proceedings - 2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2017, pp. 31–41, 2017.

[5] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis," in Proceedings - International Conference on Data Engineering, 2010, pp. 41–51.

[6] "Spark-perf: Performance tests for Apache Spark," 2020. [Online]. Available: https://github.com/databricks/spark-perf

[7] Z. Yu, Z. Bei, and X. Qian, "Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing," ACM SIGPLAN Notices, vol. 53, no. 2, pp. 564–577, 2018.

[8] N. Nguyen, M. Maifi Hasan Khan, and K. Wang, "Towards Automatic Tuning of Apache Spark Configuration," IEEE International Conference on Cloud Computing, CLOUD, vol. 2018-July, pp. 417–425, 2018.

[9] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas, "Taking the human out of the loop: A review of Bayesian optimization," Proceedings of the IEEE, vol. 104, no. 1, pp. 148–175, 2016.

[10] P. Jamshidi and G. Casale, "An uncertainty-aware approach to optimal configuration of stream processing systems," Proceedings - 2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2016, no. i, pp. 39–48, 2016.

[11] L. Bao, X. Liu, and W. Chen, "Learning-based Automatic Parameter Tuning for Big Data Analytics Frameworks," Proceedings - 2018 IEEE International Conference on Big Data, Big Data 2018, pp. 181–190, 2019.

[12] Z. Bei, Z. Yu, H. Zhang, and W. Xiong, "RFHOC : A Random-Forest Approach to Auto-Tuning Hadoop ' s Configuration," vol. 27, no. 5, pp. 1470–1483, 2016.

[13] Y. Chen, P. Goetsch, M. A. Hoque, J. Lu, and S. Tarkoma, "d-Simplexed: Adaptive Delaunay Triangulation for Performance Modeling and Prediction on Big Data Analytics," IEEE Transactions on Big Data, p. 1, 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8878273

[14] G. Wang, J. Xu, and B. He, "A Novel Method for Tuning Configuration Parameters of Spark Based on Machine Learning," in Proceedings - 18th IEEE International Conference on High Performance Computing and Communications, 14th IEEE International Conference on Smart City and 2nd IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2016. Institute of Electrical and Electronics Engineers Inc., jan 2017, pp. 586–593.

[15] Á. B. Hernández, M. S. Perez, S. Gupta, and V. Muntés-Mulero, "Using machine learning to optimize parallelism in big data applications," Future Generation Computer Systems, vol. 86, pp. 1076–1092, 2018.

[16] N. Luo, Z. Yu, Z. Bei, C. Xu, C. Jiang, and L. Lin, "Performance modeling for spark using SVM," Proceedings - 2016 7th International Conference on Cloud Computing and Big Data, CCBD 2016, pp. 127–131, 2017.

[17] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and É. Duchesnay, "Scikit-learn: Machine Learning in Python," Journal of Machine Learning Research, vol. 12, no. 85, pp. 2825–2830, 2011.

[18] D. V. Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," Proceedings of the ACM SIGMOD International Conference on Management of Data, vol. Part F127746, pp. 1009–1024, 2017.

[19] G. Liao, K. Datta, and T. L. Willke, "Gunther: Search-based Auto-tuning of MapReduce," in Proceedings of the 19th International Conference on Parallel Processing, ser. Euro-Par'13, F. Wolf, B. Mohr, and D. an Mey, Eds. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 406–419. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40047-6{_}42

[20] J. Polo, D. Carrera, Y. Becerra, J. Torres, E. Ayguadé, M. Steinder, and I. Whalley, "Performance-driven task co-scheduling for mapreduce environments," in 2010 IEEE Network Operations and Management Symposium - NOMS 2010, 2010, pp. 373–380.

[21] J. Lu, Y. Chen, H. Herodotou, and S. Babu, "Speedup your analytics: Automatic parameter tuning for databases and big data systems," Proceedings of the VLDB Endowment, vol. 12, no. 12, pp. 1970–1973, 2018.

[22] H. Herodotou, Y. Chen, and J. Lu, "A Survey on Automatic Parameter Tuning for Big Data Processing Systems," ACM Computing Surveys, vol. 53, no. 2, pp. 1–37, 2020.

[23] 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," Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT, pp. 303–315, 2014.

[24] M. M. Bersani, F. Marconi, D. A. Tamburri, A. Nodari, and P. Jamshidi, "Verifying big data topologies by-design: a semi-automated approach," Journal of Big Data, vol. 6, no. 1, 2019. [Online]. Available: https://doi.org/10.1186/s40537-019-0199-y

[25] "Tuning Spark 3.0.0," 2020. [Online]. Available: http://spark.apache.org/docs/latest/tuning.html

[26] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: A self-tuning system for big data analytics," CIDR 2011 - 5th Biennial Conference on Innovative Data Systems Research, Conference Proceedings, pp. 261–272, 2011.

[27] G. Wang, A. R. Butt, P. Pandey, and K. Gupta, "A simulation approach to evaluating design decisions in MapReduce setups," in 2009 IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems, 2009, pp. 1–11. [Online]. Available: https://ieeexplore-ieee-org.recursos.biblioteca.upc.edu/abstract/document/5366973

[28] R. Tous, A. Gounaris, C. Tripiana, J. Torres, S. Girona, E. Ayguade, J. Labarta, Y. Becerra, D. Carrera, and M. Valero, "Spark deployment and performance evaluation on the MareNostrum supercomputer," Proceedings - 2015 IEEE International Conference on Big Data, IEEE Big Data 2015, pp. 299–306, 2015.

[29] P. Petridis, A. Gounaris, and J. Torres, "Spark parameter tuning via trial-and-error," Advances in Intelligent Systems and Computing, vol. 529, pp. 226–237, 2017.

[30] M. Kunjir and S. Babu, "Black or White? How to Develop an AutoTuner for Memory-based Analytics." in Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, 2020, pp. 1667–1683.