Do the Best Cloud Configurations Grow on Trees? An Experimental Evaluation of Black Box Algorithms for Optimizing Cloud Workloads

Muhammad Bilal[1,2,*], Marco Serafini[3], Marco Canini[4], Rodrigo Rodrigues[2]
[1]UCLouvain, [2]IST(ULisboa)/INESC-ID, [3]University of Massachusetts Amherst, [4]KAUST
muhammad.bilal@uclouvain.be, marco@cs.umass.edu, marco@kaust.edu.sa, rodrigo.miragaia.rodrigues@tecnico.ulisboa.pt

ABSTRACT

Cloud configuration optimization is the procedure to determine the number and the type of instances to use when deploying an application in cloud environments, given a cost or performance objective. In the absence of a performance model for the distributed application, black-box optimization can be used to perform automatic cloud configuration. Numerous black-box optimization algorithms have been developed; however, their comparative evaluation has so far been limited to the hyper-parameter optimization setting, which differs significantly from the cloud configuration problem. In this paper, we evaluate 8 commonly used black-box optimization algorithms to determine their applicability for the cloud configuration problem. Our evaluation, using 23 different workloads, shows that in several cases Bayesian optimization with Gradient boosted regression trees performs better than methods chosen by prior work.

PVLDB Reference Format:
Muhammad Bilal, Marco Serafini, Marco Canini, and Rodrigo Rodrigues. Do the Best Cloud Configurations Grow on Trees? An Experimental Evaluation of Black Box Algorithms for Optimizing Cloud Workloads. *PVLDB*, 13(11): 2563-2575, 2020.
DOI: https://doi.org/10.14778/3407790.3407845

1. INTRODUCTION

Resource intensive applications such as analytical query processing, machine learning training, or other data-parallel applications are often run on cloud resources. The cloud provides great flexibility to scale computation up (by picking appropriately sized computing instances) and out (by choosing how many instances to use). However, this blessing can also be a curse since choosing the best configuration is often not straightforward.

Different applications and workloads have different behavior and resource requirements, and therefore their optimal cloud configuration is not the same. Furthermore, choosing the right configuration is crucial not only for cost efficiency but also to meet service-level objectives on the completion time for these jobs. To illustrate this, Figure 1 shows the heatmap of the normalized execution time and execution cost of two workloads on a cloud configuration search space. The configurations shown in orange are those with more than an order of magnitude higher runtime or execution cost compared to the best configuration in the search space. This highlights that choosing a bad configuration can lead to significantly higher execution time and execution cost than choosing the best configuration (up to 47 and 12.5 times higher, respectively).

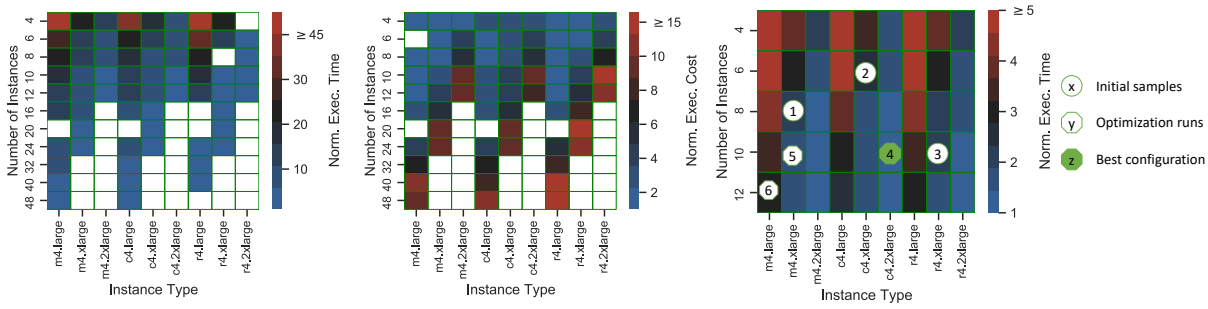
Exhaustively testing the workload on all possible cloud configurations is very expensive and wasteful. This has recently led to several proposals for automatically finding a close to optimal configuration, quickly and at a low cost. Cherrypick [9], PARIS [38], Scout [27], Micky [26], and Arrow [25] are just some examples of approaches in this space. Each of these proposals employs a specific black-box optimization algorithm, such as random forests, Bayesian optimization with Gaussian processes, or Bayesian optimization with extra trees. By leveraging generic black-box models, these systems avoid creating an analytical model, which would require capturing the complex relationship between the execution time of an application, the resources of the cloud instances, and the input workload.

Existing work on cloud configurations focuses on applying specific black-box optimization algorithms to solve the problem but falls short of providing a complete characterization of which algorithm is more suitable in which situation. As a consequence, it is not clear whether the optimization performance of these prior works is based on the selection of the optimization algorithms or on additional design decisions in their approaches. For example, Cherrypick uses Bayesian optimization with Gaussian processes while Arrow uses Bayesian optimization with extra trees. But it is not clear how much of the performance gain is because of the choice of the optimization algorithm itself, if any. A similar question arises when determining the advantages of techniques like augmenting Bayesian optimization with low-level performance metrics [25] or augmenting the search space with some prior knowledge about the performance of a similar workload [27].

This paper fills this gap by performing an exhaustive comparison of 8 black-box optimization techniques using a total of 23 different workloads and two different search spaces with 69 and 140 different cloud configurations, respectively. Our evaluation highlights how searching for a cloud configuration has some peculiar characteristics that are different from other applications of black-box modeling, for example, hyper-parameter search [17]. On the one hand, running a cloud configuration setting is very expensive. It requires starting up a cluster, deploying the application, and running the workload. Therefore, it is typically only feasible to explore a limited number of solutions. On the other hand, the search space is relatively small, since a configuration consists of selected combinations of instance type and cluster sizes, and both parameters –

*Work done in part while author was interning at KAUST.

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407845



(a) Execution time for reg_H **(b) Execution cost for lr_H** **(c) Example search space and optimization samples**

Figure 1: (a, b) Execution time and execution cost for two selected workloads in DS_1 (see Table 2) shown as a heatmap normalized by the best configuration for each respective metric. The blank boxes are configurations that were either not present in the dataset or do not execute the workload successfully. (c) An example of the optimization samples of a black-box algorithm on a search space.

instance type and the number of instances – have a relatively small range of possible values in practice. Furthermore, these parameters are integer and categorical, as opposed to continuous parameters.

Another contribution of this paper is to produce an analysis of different variants of Bayesian Optimization (BO), which is the most common technique used by existing work. Our analysis shows that, in several cases, BO with Gradient Boosted Regression Trees (GBRT) outperforms the methods used in prior research. Additionally, while prior work employs Expected Improvement (EI) as an acquisition function, we found the Probability of Improvement (PI) to be a better alternative for cloud configuration.

Overall, our results show that BO outperforms other algorithms that we analyzed in our empirical study. In terms of its variants, BO with Gaussian Processes (GP) and BO with GBRT provide the best performance when optimizing for execution cost and execution time, respectively. BO with GBRT provides up to 20% better execution time and BO with GP is able to provide up to 40% better execution cost. Unlike prior work, we also consider running optimization in an online mode, where different configurations are tested in the initial runs of a production setting with an upper bound on the execution time. We perform a break-even analysis to see when the cost of finding an optimal configuration can be amortized. Lastly, we provide a decision workflow for users to select an appropriate optimization algorithm based on their requirements.

Our experimental data and the implementation of black-box optimization algorithms are available [1] to facilitate further research on cloud configuration optimization.

2. BACKGROUND AND RELATED WORK

2.1 Problem statement

We formalize the cloud configuration problem as the problem of finding a configuration of resources (i.e., the type and number of machine instances) that minimizes an objective function. For example, the objective function could be job execution time or execution cost (or a combination of both). Formally, a cloud configuration can be denoted as a tuple $x = (N, I_F, I_S)$ where N is the number of instances, I_F is the instance family and I_S is the instance size. Instance family and instance size combine to form a particular instance type that can be requested from the cloud provider.

An instance type defines the CPU, memory, and storage capabilities of an instance. To define instance types, we follow the convention in use at many cloud providers that group their instance types based on their instance family (e.g., general-purpose, compute- or memory-optimized) and instance size (e.g., large, xlarge, 2xlarge).

Table 1: Prior work, corresponding optimization algorithms and cloud configurations that they optimize.

Prior work	Black-Box method	Optimization goal
Ernest [34]	N/A (White-Box Model)	Cluster size
Cherrypick [9]	BO with Gaussian Processes	Instance type and Cluster size
PARIS [38]	Random Forests	Instance type
Arrow [25]	BO with Extra Trees	Instance type
Micky [26]	Multi-armed bandit	Instance type
Scout [27]	Custom search	Instance type and Cluster size

The instance family expresses the class of hardware specifications that may best meet the requirements of different applications as well as a CPU-memory ratio. The instance size, in turn, allows for choosing the number of virtual CPUs, the available memory size, local storage, and network bandwidth. We assume that there is a finite set of instance types and a finite number of choices for the number of instances. The set of all possible combinations of instance types and number of instances yields the *configuration search space*.

Figure 1c shows an example of the sequence of configurations explored in a search space for cloud configuration. The x-axis has the instance types (available in AWS) and the y-axis has the number of nodes in the cluster. The heatmap colors represent normalized runtime w.r.t. the best execution time in the search space: configurations with darker colors are closer to the best configuration (which is shown with the green hexagon). Most optimization algorithms start with a set of initial samples (shown with white circles) and then proceed with performing some optimization runs (shown with white hexagons), eventually reaching, or at least approaching, the best configuration in the search space.

2.2 Prior approaches

The problem of automatic cloud configuration has attracted the attention of the research community as of late. Table 1 shows prior work in the area, the optimization algorithms that they use and the cloud configurations that they optimize for. In general, prior work compares only against random search or coordinate descent, excluding other alternative black-box optimization algorithms. Our work is aimed at filling this gap.

In Ernest [34], the authors develop an analytical model to predict performance for machine learning jobs in Apache Spark. The

model predicts the performance of the jobs on different cluster sizes. The coefficients of the analytical model are learned by actually executing the workload under consideration, on some of the cloud configurations. However, the model is limited to machine learning applications and has to be trained for each instance family.

White-box analytical performance models are complex to derive and limited in scope. Therefore, much of the subsequent work has focused on black-box performance models.

PARIS [38] aims to find the best instance types for a given workload and user goals. It builds a regression model to estimate the execution time and running cost of different instance types. Offline benchmarking is used to develop a collection of performance information about benchmarked applications. This data is then used to train a black-box performance model using random forests. New workloads are first fingerprinted using low-level performance metrics. Next, PARIS recommends an instance type by matching the fingerprint of the new workload with previously benchmarked applications using the performance model.

Cherrypick [9] uses BO with Gaussian processes, a well-known technique from prior research in hyper-parameter optimization [17], to perform automatic cloud configuration. Cherrypick targets constrained optimization: the goal is to find out the number of instances and types of instances that minimize the execution cost of the job while satisfying a maximum time constraint.

Arrow [25] uses low-level performance metrics to augment the BO process. The authors use BO with extra trees to find the best instance type to minimize execution cost or execution time.

Micky [26] targets the optimal choice of instance type for a group of workloads while considering execution time or execution cost objective functions. The goal of Micky is to find a single instance type that works best for a group of workloads. While Micky allows users to reduce the optimization cost, the instance type suggested by Micky will inevitably be sub-optimal for some workloads in the group, and methods like Cherrypick can be used to optimize further if needed.

This paper complements these prior works by evaluating the effect of a larger number of alternative black-box methods and by devising criteria to select the best method in different circumstances.

Scout [27] approaches the problem of cloud configuration differently: it builds a relaxed model that uses pairwise classification to determine whether there are better configurations in the search space, instead of trying to predict the performance of those configurations accurately. Scout assumes a large amount of historical data. Approaches like Scout and PARIS optimize the configuration of a workload by searching for similar workloads that have been previously modeled. The performance models previously built can then be used to establish prior information about the performance of the new workload on the cloud configurations. The techniques discussed in these papers can be used to find better initial configurations for the optimization algorithms discussed in our work.

Outside of the cloud configuration problem space, there is a large literature on black-box optimization algorithms. Prior works on system and software parameter tuning also make use of black-box optimization algorithms [13,15,37,39]. Several surveys [28,31,33] provide an overview of several black-box optimization algorithms, including the ones discussed in this work. Vizier [19] presents a black-box optimization-as-a-service framework used internally at Google. It includes algorithms like Bayesian optimization and CMA-ES [21]. Similarly, there are optimization-as-a-service companies like SigOpt [6] that allow external users to optimize any model using a proprietary black box optimization engine which is composed of an ensemble of global and Bayesian optimization algorithms.

3. OPTIMIZATION ALGORITHMS

Since performance models of distributed data processing applications on different types of cloud instances are not generally available, we treat this setting as a black box. Given a configuration x from the configuration search space X and the underlying unknown model as f, then $f(x)$ is the objective function value of a workload under consideration when executed on cloud configuration x. The function f can be used to model objectives like execution time or total execution cost. The cloud configuration problem seeks to find x^* such that

$$x^* = \arg\min_{x \in X} f(x) \tag{1}$$

In our setting, we will be minimizing objective functions. (Maximizing an objective function is an identical problem.) We assume that a user has a given optimization budget, i.e., the number of iterations that the optimization algorithm is allowed to make is limited. This is a common, practical assumption for existing work in cloud configuration optimization.

There is a wide variety of black-box optimization (BBO) algorithms that have been discussed and defined in the literature [17]. Most of these were created for hyper-parameter optimization. In this work, we pick a subset of these optimization algorithms based on both their popularity and, particularly, their optimization budget requirements. For example, we avoid population-based methods since they require a higher number of optimization steps.

Below we briefly review a series of black-box optimization methods that we later evaluate empirically. These can be classified into three main classes: Sampling-based (Random Search), Sequential Model-Based (Bayesian Optimization variants, TPE), and Search-based (Stochastic Hill Climbing and Simulated Annealing).

3.1 Random Search

The random search algorithm simply generates unique configurations by randomly sampling (without replacement) the configuration search space. In random search, new samples are generated without taking into account the samples that were previously generated. Thus, the samples generated could be quite close to each other and in the worst-case scenario concentrated in the same region of the search space. However, random sampling provides the benefit of allowing users to generate new samples progressively, i.e., the number of to-be-tested configurations does not need to be defined ahead of time.

3.2 Bayesian Optimization (BO)

BO proceeds by maintaining a probabilistic belief about f and using a function to determine where to evaluate f next (called *acquisition function*). BO is particularly well-suited to global optimization problems where f is an expensive black-box function. In our setting, evaluating f entails deploying and running a workload on a cloud configuration, which is very expensive.

BO reasons about f by starting with a prior $p(f)$. The prior is the initial belief about the behavior of the objective function. This prior is based on the *surrogate function* which is used to build the model (called a surrogate model $\mathcal{S}$) of the underlying unknown black-box function. Given the observations $\mathcal{D} = \langle x, f(x) \rangle$, it uses the Bayesian method to estimate a posterior distribution as:

$$p(f|\mathcal{D}) = \mathcal{S}(f; \mu_{f|\mathcal{D}}, \sigma_{f|\mathcal{D}})$$

where μ is the mean and σ is the variance or co-variance.

3.2.1 *Surrogate models*

There are many options to build a surrogate model; we focus on the most widely used methods: Gaussian Processes (GP) [36],

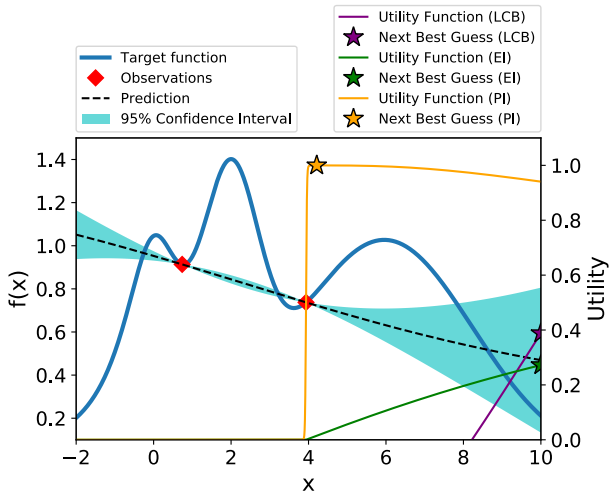


Figure 2: Example of an objective function being modeled by Bayesian optimization. The objective function is being minimized. The utility function for each acquisition function is shown along with the recommendation for the next point to evaluate, based on the highest value of the utility function.

Gradient Boosted Regression Trees (GBRT) [22], Random Forests (RF) [14] and Extra Trees (ET) [18].

Gaussian Processes (GP): Gaussian processes are generic supervised learning methods that can be used to solve regression as well as probabilistic classification problems. Gaussian processes describe probability distributions over a set of functions. The Bayes rule is used to update this distribution based on training data. After each observation, the updated Gaussian process is constrained to the possible functions that fit the training data.

Gradient Boosted Regression Trees (GBRT): GBRT is a flexible non-parametric statistical learning technique. Just like GP, it can be used for both regression and classification. Boosting works on the principle of ensemble by combining a set of weak learners to deliver improved prediction accuracy. Gradient boosting involves three elements: 1) a loss function, 2) a weak learner to make predictions (regression trees in our case), and 3) an additive model (gradient descent) to add weak learners to minimize the loss function. The regression trees are built one at a time, sequentially. Each new tree helps to correct errors made by a previously trained tree.

Random Forests (RF): In contrast to GBRT, RF use a bagging technique for ensemble learning. In bagging (or bootstrap aggregation), each model in the ensemble is created using a random subset of the original training dataset, which has the advantage of reducing variance. Unlike boosting, where trees are built sequentially, in bagging, each model is built independently, and then the outputs of each model are aggregated at the end. In this method, each model is a tree.

Extra Trees (ET): ET (or Extremely Randomized Trees) introduce more variation into the ensemble. Unlike RF, all data is available for training of each tree and the split of each feature (to form a node in the tree) is chosen at random. Both features and splits are chosen at random in contrast to RF where only features are selected at random. This added randomness allows ET to have lower variance than RF.

3.2.2 Acquisition functions

In BO, the role of an acquisition function is to select the next configuration to test in order to update the surrogate model. The acquisition function can be interpreted as a function that evaluates the expected utility associated with updating $f(x)$. BO then selects the point with the highest expected utility, hence picking the most desirable point to evaluate F. Therefore, the acquisition function $A(x)$ can be represented as the expected utility $u(x)$, given the configuration x and the previous observations $\mathcal{D}$:

$$A(x) = \mathbb{E}[(u(x)|x, \mathcal{D})]$$

We now review the most commonly used acquisition functions and their utility functions. We first introduce them using a toy example. Figure 2 shows the surrogate model of some artificial objective function (shown in blue) built using two initial samples (observations). The utility function for each acquisition function discussed below is also shown in the figure. We can see that different acquisition functions attribute completely different utility and the next best x to evaluate (illustrated as a star) is based on the highest value of the respective acquisition function.

Probability of Improvement (PI): Let f' be the minimum value of f observed so far; the PI function suggests a value of x that is most likely to improve upon this value. The utility function for PI is represented as:

$$u(x) = \begin{cases} 0 & f(x) > f' \\ 1 & f(x) \leq f' \end{cases}$$

Expected Improvement (EI): The PI function provides a reward for improving upon the current minimum that is irrespective of the margin of improvement. Alternatively, EI also takes into account the amount of improvement. Hence, the utility function becomes:

$$u(x) = \max(0, f' - f(x))$$

Thus, the reward is equal to the amount of improvement. Therefore, the EI acquisition function picks a point with the highest expected improvement.

Lower Confidence Bound (LCB): The LCB acquisition function takes the form:

$$A_{LCB}(x; \kappa) = \mu(x) - \kappa\sigma(x)$$

where $\kappa > 0$ is a trade-off hyper-parameter that can be used to control exploration and exploitation. $\sigma(x)$ is the standard deviation and $\mu(x)$ is the mean. This acquisition function is also called Upper Confidence Bound (UCB), when used for maximizing an objective function.

GP hedge: GP hedge [23] chooses one of the three acquisition functions described above in a probabilistic fashion, based on gains g. Initially, gains are set to zero. At every iteration, each acquisition function is optimized independently to get a candidate point x_t. Out of these candidates, the next point x_{best} is chosen based on a softmax function $softmax(\eta g_t)$. After fitting the surrogate model with a new observation (x_{best}, y_{best}), the gain for each acquisition function is updated as $g_t = g_{t-1} + \mu(x_t)$. The hyper-parameter η can be used to control the weights assigned to the gain for each acquisition function. This method is only available when using Gaussian processes for the surrogate model.

All the above acquisition functions provide a hyper-parameter to adjust the trade-off between exploitation (evaluating points with low mean) and exploration (evaluating points with high uncertainty).

3.3 Tree-structured Parzen Estimator (TPE)

Tree-structured Parzen Estimator and Bayesian optimization take a slightly different approach towards solving the same problem. While Gaussian process-based approaches model $p(y|x)$ directly (i.e., $y = f(x)$), TPE [12] models $p(x|y)$ and $p(y)$ separately. TPE

essentially divides the sorted observations $\mathcal{D} = \langle x, f(x) \rangle$ according to the objective function value into two sets using a threshold y^*. Thus, TPE defines $p(x|y)$ using two such densities:

$$p(x|y) = \begin{cases} l(x) & y < y^* \\ g(x) & y \geq y^* \end{cases}$$

where $l(x)$ is the density formed by observations that performed well and $g(x)$ is the density formed by observations that performed poorly. TPE algorithms choose y^* to be some quantile γ of the observed values y, such that $p(y < y^*) = \gamma$. The intuition is that good configurations will have a low probability in $g(x)$ and a high probability in $l(x)$. Thus, we can evaluate $\arg\min(g(x)/l(x))$ as the utility function for acquisition functions.

3.4 Stochastic Hill Climbing (SHC)

Stochastic hill climbing is a search based method that does not rely on modeling the underlying black-box function. In this paper, stochastic hill climbing works as follows. First the initial configurations X_0 are evaluated to get Y_0. The best configuration is picked from X_0 as the current configuration x_c. At each subsequent step a random neighbor of the current configuration is generated. The current configuration (x_c) is then replaced by a random neighbor (x_n) given:

$$x_c = \begin{cases} x_n & p(x_n) \geq 1 \vee p(x_n) \geq \text{random}() \\ x_c & \text{otherwise} \end{cases}$$

where $p = e^{-(f(x_n)-f(x_c))/T}$. Here, T is a temperature value that is provided by the user. The formulation for p basically means that if $f(x_n) < f(x_c)$ then the neighbor is selected as the current configuration. Otherwise, the temperature hyper-parameter controls the probability of the neighbor being selected as the current configuration based on the difference between the objective function value of the two configurations. Having a higher temperature means that configurations with a significantly worse objective function value than the current configuration can be selected. Therefore, a higher temperature allows for more exploration while a lower temperature allows for more exploitation. If $f(x_c)$ is less than the objective function value of the best configuration x_b (i.e., $f(x_b)$), then x_c is selected as the best configuration x_b. The algorithm terminates when one of the following two conditions are satisfied: a user-specified minimum is reached, or a specified budget is reached.

3.5 Simulated Annealing (SA)

Simulated Annealing is fairly similar to SHC, with the main difference being the introduction of a cool-down factor for the temperature T. In particular, for each step $i \rightarrow i+1$ a schedule constant α is used to lower the temperature as $T_{i+1} = \alpha T_i$, where $\alpha < 1$. Hence, simulated annealing lowers the temperature as the optimization progresses, therefore shifting the focus from exploration to exploitation later during optimization.

3.6 Excluded BBO methods

The literature on black-box algorithms is vast and there are numerous black-box algorithms used for hyper-parameter optimization that we have not included in this work. In particular, CMA-ES [21] and other evolutionary algorithms such as GGA [10] and particular swarm optimization [29] are not suitable for the cloud configuration problem because of the higher budget required for evolutionary algorithms. Hyperband [30], which is essentially strategic random sampling, is designed for cases where only partial data can be used for evaluating a configuration. Thus, it is better suited in cloud configuration scenarios where a single best configuration for multiple input datasets is the optimization goal.

4. EXPERIMENTAL METHODOLOGY

To experimentally compare the black-box optimization algorithms reviewed in §3, we optimize the cloud configuration of a large set workloads. We describe our methodology in this section and present the results in the next section.

Our methodology is carefully designed to establish the statistical significance of the results and achieve a fair comparison across algorithms. We proceed as follows:

Decouple optimization from profiling: We compare optimization algorithms on the basis of their performance on pre-collected profiling datasets comprising the execution time of every workload for every configuration in the search space. Since our focus is exclusively on the comparison of optimization algorithms, for a given configuration, we hold its execution time constant and thus, we avoid biasing our results from the effects of the many sources of performance variability in cloud environments. Further, this enables us to determine the best configuration in the search space.

Repetitions: For each workload, we run every optimization algorithm 50 times, each time picking a different random set of initial samples. Repeating the optimization process multiple times is crucial to assess the overall optimization performance because the performance of many optimization algorithms depends on the initial set of random samples.

Initialization consistency: The same set of initial samples is used across all optimization algorithms in every repetition of an optimization run. This prevents differences in the initial sample sets from adding variance to the results across different optimization algorithms and is the basis for a fair comparison thereof.

Statistical significance: We use statistical significance tests to determine whether the mean values of the performance metrics (see below) vary in a meaningful way from one algorithm to another. In particular, we use the independent t-test [16] (with $p \leq 0.05$) from the family of parametric significance tests.

The remainder of this section provides further details regarding the algorithm implementations, datasets, and performance metrics.

4.1 Implementation

We implement the black-box optimization algorithms atop several widely used libraries. BO methods are built using the SkOpt library [7], since it provides a range of surrogate models and acquisition functions; TPE is implemented using the Hyperopt library [4, 11]; SHC and SA are based on the Solid library [8]. To provide a comparison on common ground, we modified the Solid library such that SHC and SA start from a set of n initial samples, instead of a single sample that Solid originally allows. We modified the optimization libraries to provide different methods with the same initial samples for a given repetition of the optimization process.

4.2 Datasets

Table 2 shows an overview of the workloads and configuration search spaces of the two datasets used in this paper, for a total of 23 workloads.

DS_1 is provided by the authors of Scout [5]. This dataset includes execution time information on a set of multi-node workloads, for a search space comprising of 69 configurations on which each workload has been executed. There are a total of 18 workloads comprised of 9 different applications and 2 input data sizes per application. The applications consist of jobs executing within Spark or Hadoop as listed in Table 2. In DS_1, the best execution

Table 2: Workloads and configuration search space for each dataset.

Dataset	Workloads [3]			Configuration Search Space		
	Framework	Applications	Input Size	I_F	I_S	N
DS_1	Hadoop	Pagerank (hpr), Terasort (ts), Wordcount (wc)	Huge (H), Bigdata (B)	m4, c4, r4	large	4, 6, 8, 10, 12, 16, 24, 32, 40, 48
	Spark 1.5	Kmeans (km), Naive-Bayes (nb), Regression (reg)			xlarge	4, 6, 8, 10, 12, 16, 20, 24
	Spark 2.1	Join (jn), Pagerank (spr), Logistic Regression (lr)			2xlarge	4, 6, 8, 10, 12
DS_2	Spark 2.4.4	Linear Regression (lnr), Latent Dirichlet Allocation (lda), Random Forest (rf)	Huge (H), Gigantic (G)	m5, m5a, c5n, c5, r5	large	16, 24, 32, 40, 48, 56, 64
					xlarge	8, 12, 16, 20, 24, 28, 32
					2xlarge	4, 6, 8, 10, 12, 14, 16
					4xlarge	2, 3, 4, 5, 6, 7, 8

time and execution cost are within 150-2171s and 0.11-2.45 dollars, respectively.

To evaluate the performance of the optimization algorithms over a larger cloud configuration search space, we generated a second dataset (DS_2) by profiling runs of 5 additional workloads (based on 3 additional applications) in a configuration search space of 140 configurations. The size of this search space is more than twice the size of the search space used in Scout [5] (which had 69 configurations) and Cherrypick [9] (66 configurations). To reach this larger configuration search space, DS_2 includes execution time information for 20 instance types and 7 cluster sizes for each instance type, as shown in Table 2. For DS_2, the best execution time and execution cost are within 114-324s and 0.09-0.64 dollars, respectively.

The workloads for both datasets were obtained from Intel's HiBench big data benchmark suite [3]. The details about the characteristics of each input size (Huge, Gigantic, and Bigdata) as well as the applications are included in the public repository of HiBench [3]. Note that Random Forest failed to run successfully on a lot of cloud configurations in our search space with Gigantic input size and it is thus excluded.

Our experiments end up covering 63-69 configurations (out of 69) for DS_1 and 122-140 configurations (out of 140) for DS_2. The majority of optimization algorithms explore all configurations across 50 repetitions.

4.3 Evaluation metrics

Since we use 23 different workloads (18 in DS_1 and 5 in DS_2) for our evaluation, we must resort to aggregate metrics that summarize the results. In particular, we use the following two metrics in our evaluation to determine the performance of the optimization algorithms. Hereby, *objective function value* refers to either execution time or execution cost. Comparisons are made using the minimum of the objective function value (as per Eq. 1) within the given optimization budget.

Performance Score: The performance score is calculated by performing pairwise comparisons between all optimization algorithms. If the difference in the mean of the best objective function values achieved by two optimization algorithms is statistically significant, then the performance score of the algorithm with the lower objective function value is incremented. The pairwise comparison is performed for each optimization budget level and across all workloads. The performance score is aggregated across optimization budgets and workloads in each dataset.

Normalized Performance: The performance score provides an overall relative ranking of the optimization algorithms. Thus it helps us determine the best optimization algorithm in a given scenario. However, it does not give information about the actual difference in the best objective function value achieved by an algorithm compared to others. For that, we use normalized performance, which is the best objective function value of each optimization algorithm normalized w.r.t. the best algorithm according to the performance score. If there is no statistically-significant difference between the performance w.r.t. the best algorithm, then both are assumed to have the same performance.

5. EXPERIMENTAL RESULTS

Since we have a large number of optimization algorithms and choices for their hyper-parameters, we first narrow down the field of optimization algorithms to compare and then perform the comparison between those optimization algorithms using different scenarios. Thus, we split our analysis into three main parts: (i) the selection of hyper-parameter values for the optimization algorithms, (ii) the evaluation of BO variants, and (iii) the evaluation of selected optimization algorithms.

Henceforth, we use the following notation to represent a particular workload based on Table 2: $\mathsf{application}_{\mathsf{input\_size}}$. For instance, lr_{H} represents the linear regression application running with Huge input data size (preset size in HiBench).

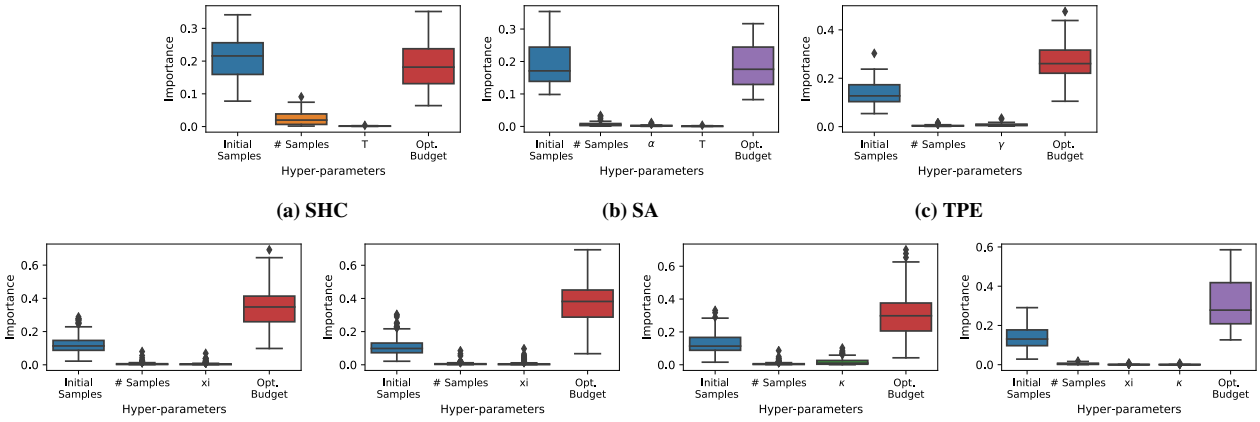
The comparisons are performed across 5 different optimization budgets (6, 12, 18, 24, 30). The results are based on 50 repeated runs of the optimization algorithms for each workload using different randomized initial sample sets for each run.

To present results succinctly, we omit plots of performance score values. Instead, in any given scenario, we use the performance scores to determine the best algorithm. Then, we illustrate performance comparisons using normalized performance graphs (see Figure 4 as a reference). This kind of graph shows the normalized mean performance of algorithms w.r.t. the best algorithm as a bar plot associated with the left y-axis while the normalized performance of the best algorithm w.r.t. the best configuration in the search space is shown as a line plot associated with the right y-axis of the graph. The error bars represent 95% confidence interval (after statistical significance is accounted for during normalization).

5.1 Configuring optimization algorithms

Every run of an optimization algorithm in a given scenario is associated with a certain configuration that comprises of the following hyper-parameters:[1] the number of initial samples, the seed to the random generator of initial samples, the optimization budget, and the algorithm-specific hyper-parameters (see Table 3). We now analyze the importance and impact of different hyper-parameter values across algorithms. We use performance scores to pick the best-performing hyper-parameter values for each algorithm.

[1]We use the term hyper-parameter to distinguish from the cloud configuration *parameters* (N, I_F, I_S), which are the output of the optimization process.



(a) SHC (b) SA (c) TPE

(d) BO with EI (e) BO with PI (f) BO with LCB (g) BO with GP Hedge

Figure 3: Importance (through Functional ANOVA) of each hyper-parameter of the optimization algorithms.

Table 3: Algorithm-specific hyper-parameters and their values. For BO with LCB acquisition function κ is used; for EI and PI, ξ is used.

Algorithm	Hyper-parameters	Values
SHC	temperature (T)	50, 100, 250, 500, 750, 1000
SA	temperature (T)	50, 100, 250, 500, 750, 1000
	schedule constant (α)	0.3, 0.5, 0.7, 0.9
BO	kappa (κ) (LCB)	1.0, 1.5 1.96, 3.0, 5.0
	ξ (EI, PI)	0.01, 0.05, 0.1, 0.2
TPE	Threshold (γ)	0.1, 0.25, 0.4, 0.55, 0.7

5.1.1 *Importance of hyper-parameters*

We use Functional ANOVA [24] to quantify the importance of each hyper-parameter towards the performance variance of the optimization algorithm. Note that, in the context of Functional ANOVA, performance variance refers to the variability in the optimization's output – the (N, I_F, I_S) tuple – in terms of the effects of variation due to different hyper-parameter. For each algorithm, in addition to the algorithm-specific hyper-parameters, we consider the importance of the initial samples selected (via random seed), the number of initial samples, and the optimization budget.

Figure 3 shows the fraction of variance (i.e., *importance*) in the performance introduced by each hyper-parameter for different optimization algorithms across all 23 workloads and for both objective functions (execution cost and execution time). For BO, the results also include the importance across all surrogate models.

The individual contribution of algorithm-specific hyper-parameters towards variance in performance is small compared to those of initial samples (via random seed) and optimization budget. On average, algorithm-specific hyper-parameters contribute to less than 5% to the overall variance. The exception is κ for BO with LCB acquisition function, in which case κ can contribute up to 10% to the variance in performance. On the other hand, the selection of initial samples and the optimization budget contributed up to 65% to performance variance, individually. The rest of the contribution is made by the pairwise marginals, which, for readability, are not shown in these plots.

5.1.2 *Best algorithm-specific hyper-parameters*

Table 3 shows the algorithm-specific hyper-parameters of different optimization algorithms and their respective values used to determine the best hyper-parameter settings. To find the best hyper-parameter setting, we evaluate all possible combinations of hyper-parameter values and the number of initial samples, for each algorithm on each workload and objective function. Using performance scores, we determine the best hyper-parameter values for each optimization algorithm across all workloads in a particular search space (DS_1 and DS_2) and for a particular objective function (execution time or execution cost). We see that the best hyper-parameter values change based on the workloads and type of objective function: there is no one winner in every scenario. For brevity, we list all of these values separately in the project repository [2].

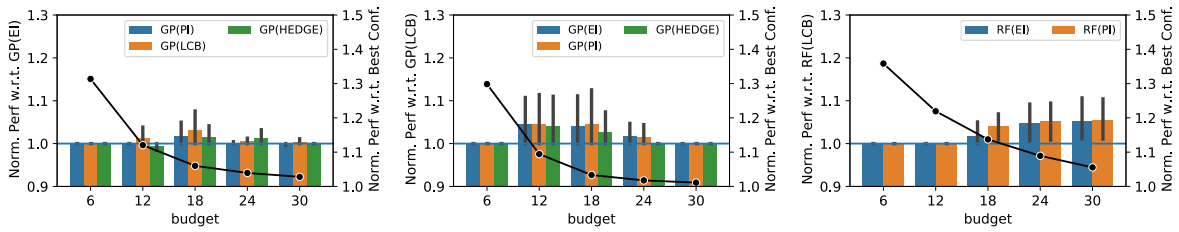
In the remainder of this section, we present results based on the best hyper-parameter values as determined by the performance scores. Since in a real setting, tuning algorithm-specific hyper-parameters would require additional optimization runs and that might not always be practical or feasible, we also present results using selected default configuration values for the optimization algorithms. In our implementation, SA and SHC are the only algorithms that do not have a default value for their temperature and schedule constant hyper-parameters. Therefore, we use the default value of 0.5 for α and 50 for T. These values are based on overall performance across multiple scenarios in our results. These are not the best hyper-parameter values in every case but they work well for most cases in our experiments, among the limited set of hyper-parameter values we tested for α and T. In contrast, for BO and TPE, there exist default values either provided in the libraries or recommended in literature: HyperOpt uses a default value of 0.25 for γ, and the SkOpt library uses a default value of 0.01 for ξ and 1.96 for κ. By default, we use these values.

For the default number of initial samples, we use 3 initial samples for BO methods and 9 for the rest of the algorithms based on the hyper-parameter evaluation. This means that for an optimization budget ≤ 9, SHC, SA, and TPE are essentially equivalent to random sampling.

Takeaways: Across the algorithm-specific hyper-parameter values that we have tested, the hyper-parameters individually only contribute between 1-10% towards the performance variance of the optimization algorithms, based on a Functional ANOVA analysis. Additionally, there is no one-size-fits-all setting for the hyper-parameters of these optimization algorithms. This means that while algorithm-specific hyper-parameter tuning can provide some benefits, the effort for tuning these hyper-parameters might not be worth the marginal expected gains.

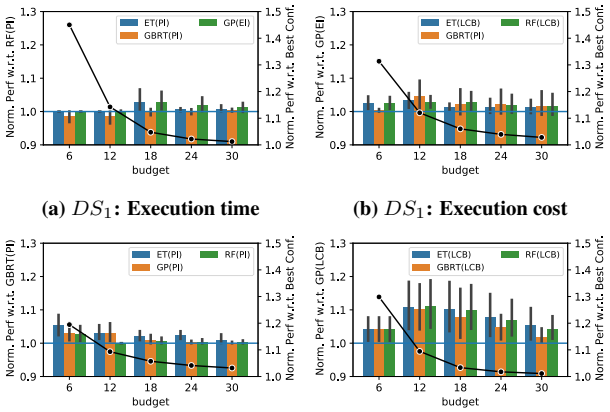
5.2 Narrowing down the choices for BO

We have several choices for acquisition functions as well as methods to build the surrogate model for Bayesian optimization. In



(a) BO(GP) on workloads in DS_1 **(b) BO(GP) on workloads in DS_2** **(c) BO(RF) on workloads in DS_2**

Figure 4: Comparison of acquisition function for different Bayesian optimization algorithms. All three cases shown here are scenarios in which the objective function is execution cost. In all other scenarios, there is less than 5% difference in the performance of different acquisition functions.



(a) DS_1: Execution time **(b) DS_1: Execution cost**

(c) DS_2: Execution time **(d) DS_2: Execution cost**

Figure 5: Comparison of different Bayesian optimization algorithms.

this section, we narrow down our choices for the best candidate(s) among the BO variants. In this set of experiments, we fix the values for the hyper-parameters for each BO variant to the best values derived from the evaluation in the previous section.

5.2.1 *Best acquisition function for BO*

First, we evaluate the best acquisition function for each available method to build surrogate models. As mentioned before, acquisition functions are used to select the next configuration to test at each step in the optimization process. When using ET, GBRT, and RF as surrogate models, we have three choices for acquisition functions: EI, PI, and LCB. When using GP as a surrogate model, we have the additional choice of using GP hedge.

We compare all the choices of acquisition function for each BO method on the workloads in both datasets and for both objective functions. Then, we use the performance score to determine the best candidate for acquisition function and then normalize the performance of other acquisition functions based on that best candidate. Figure 4 presents only three cases in which there is a significant difference in the performance of different acquisition functions. The figure shows the performance of different acquisition functions normalized to the best acquisition function in the respective scenarios (left y-axis). Additionally, the performance of the best acquisition function w.r.t. the best configuration in the search space is shown as the line plot (right y-axis).

In all three scenarios shown in Figure 4, only one workload leads to a significant performance difference. When optimizing for execution cost on DS_1 with GP surrogate model (shown in Figure 4a), using the EI acquisition function can lead up to 25% lower execution cost when optimizing for km_B. In Figures 4b and 4c, we see the gap in performance because of a different workload, namely lda_H, where using LCB results in up to 17% and 14% better performance than other acquisition functions. In all other scenarios, the difference between the performance of acquisition functions is less than 5%.

These results suggest that the use of EI as the acquisition function is not always the best option, thus contradicting some of the premises assumed by the prior work of CherryPick [9]. Henceforth, for each scenario, we use the acquisition function with the best performance score for that scenario.

Takeaways: There are only a handful of instances where there is a statistically significant difference in the performance of different acquisition functions. Generally, the performance difference is less than 10%, except for three cases out of 46 optimization scenarios where the difference in the performance between the best acquisition function and the rest was 14–25%.

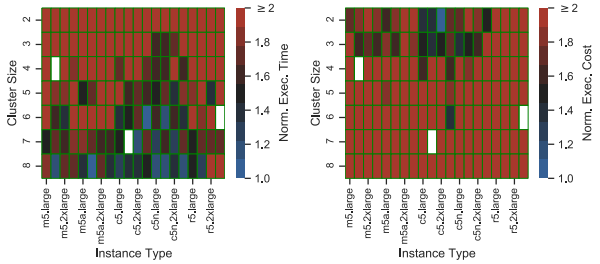
5.2.2 *Best BO variant*

Now that we have determined candidates for the acquisition function, we will compare the performance of different methods to build surrogate models using the chosen acquisition function. The normalized performance for different BO variants in different optimization scenarios is presented in Figures 5. Using the performance score, we determine that RF(PI) and GBRT(PI) are the best algorithms to optimize for execution time for DS_1 and DS_2, respectively. GBRT outperforms RF on lower budget for DS_1 but, on average, has a slightly higher minimum objective function value than RF(PI) for high optimization budget ($\geq$ 24). On the other hand, When optimizing for execution cost, BO with GP performs better than other BO variants, with LCB and EI as the best acquisition functions for DS_2 and DS_1, respectively.

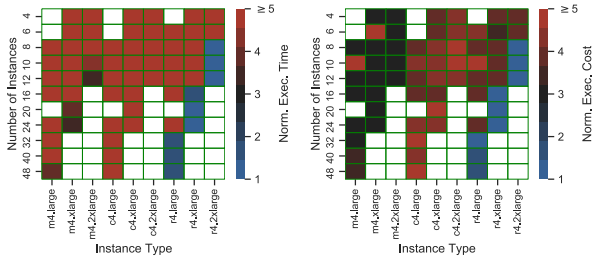
When optimizing execution time for DS_1 (Figures 5c RF(PI) provides up to 25% better execution time compared to GP and ET but overall it is comparable to GBRT(PI). For DS_2 (Figure 5a) using GBRT(PI) provides upto 9% better execution time, essentially when optimization budget is small ($\leq$ 12).

However, choice of model is more important when optimizing for execution cost (Figures 5d and 5b). GP(EI) and GP(LCB) provides up to 22% and 26% lower execution for DS_2 and DS_1, respectively. The workloads where the choice of BO variant is most important are lda_H (in DS_2) and km_B (in DS_1).

Figure 6 shows the normalized execution time and execution cost of different configurations in the search space for the lda_H and km_B workloads. By observing the values of the objective function on the search space for these two workloads, we recognize that for these workloads, very few configurations in the search space are close to optimal in performance. Additionally, these handful of configurations that are close to optimal are in the same instance family. For lda_H, there are 10 configurations within 20% of the lowest ex-



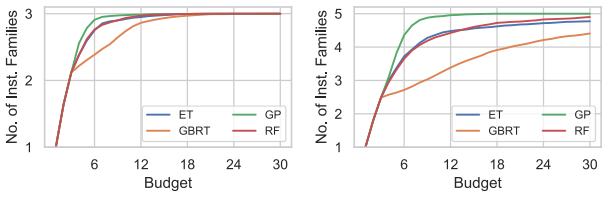
(a) lda_H: Execution time (b) lda_H: Execution cost



(c) km_B: Execution time (d) km_B: Execution cost

Figure 6: Execution time and execution cost of different configurations of lda_H and km_B. In (a) and (b), the actual number of nodes can be obtained from the cluster size using multiplying factors of 1, 2, 4, 8 for 4xlarge, 2xlarge, xlarge and large instance sizes, respectively.

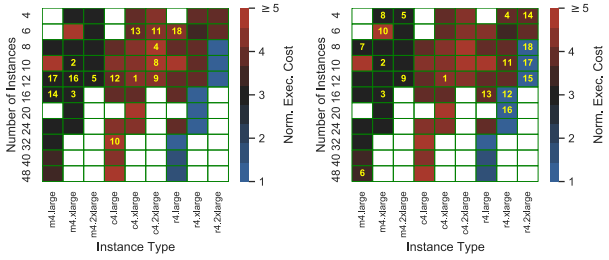
ecution time in the search space, but, when considering execution cost, the second-best configuration has 20% higher execution cost. In this case, we can conclude that optimizing for execution cost is much harder than optimizing for execution time and those are the scenarios in which BO(GP) performs better. By observing how BO(GP) explores the search space in this case, we deduce that compared to other BO variants, GP initially performs more exploration than exploitation and often tests cloud configurations with different instance families early on during the optimization process. Figure 7 shows the average number of instance families explored by each algorithm for workloads in both datasets when optimizing for execution cost. We can see that BO(GP) samples at least one configuration from each instance family more quickly than other BO variants. This leads BO(GP) to find a potentially better configuration when initial samples do not have good configurations in them. It also makes BO(GP) less susceptible to taking several steps near local minima.



(a) DS_1 (b) DS_2

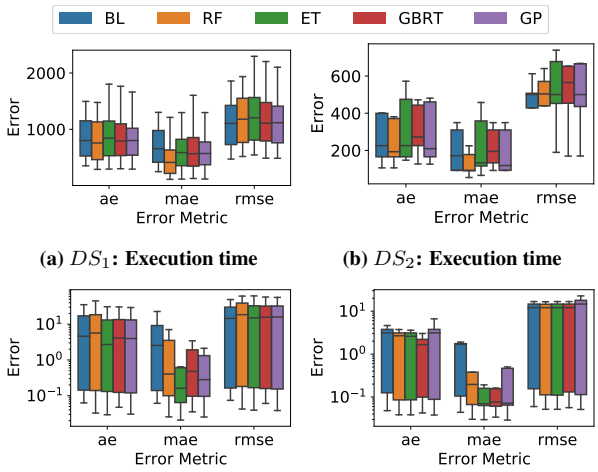
Figure 7: Number of instance families explored by GBRT and GP when optimizing for execution cost.

The latter is exactly why GP performs better than GBRT on km_B when optimizing for execution cost. Figure 6c and 6d show that good configurations for execution cost and execution time are pretty similar in the search space. However, the key difference is the fact that when optimizing for execution cost there is a large local minima when using the m4 family. Figure 8 shows a full run of GBRT and GP (using LCB acquisition function) when optimizing for execution cost for km_B. The annotated configurations represent the configurations that have been explored by the optimization algorithm (for optimization budget of 18). The configurations tested by GBRT are clustered together as GBRT is doing more exploitation based on the 3 initial samples (labeled 1-3). On the other hand, GP tests a configuration in the r4 family on the fourth run and then proceeds to explore more configurations in that family later on. This allows GP to explore the good configurations, which are all in the r4 family in this case.



(a) BO(GBRT) (b) BO(GP)

Figure 8: Configurations tested by GBRT and GP when optimizing for execution cost km_B.



(a) DS_1: Execution time (b) DS_2: Execution time

(c) DS_1: Execution cost (d) DS_2: Execution cost

Figure 9: Overall prediction error across all workloads using different accuracy metrics.

Henceforth, we show the results with the best performing variants based on each surrogate model for BO, for each optimization scenario.

Takeaways: Based on our evaluation, GBRT(PI) performs well when optimizing for execution time on both of the search spaces. RF(PI) outperforms GBRT(PI) in some cases with high optimization budget for DS_1. On the other hand, GP is the better BO variant when optimizing for execution cost, with EI and LCB as the best performing acquisition functions for DS_1 and DS_2, respectively.

5.2.3 *Model accuracy*

One of the benefits of using a BO algorithm is that it builds the surrogate model of the underlying black-box function. If this surrogate model is accurate, then it can be used to predict the value of the black-box function on unexplored points. We now evaluate the prediction accuracy of the models built by BO algorithms. As a

baseline (BL), we use a method that randomly samples the search space and always predicts the mean objective function value.

The models are built using each method after collecting 30 samples and then the objective function value for each configuration in the search space is predicted. Figure 9 shows the prediction error for different BO algorithms for three different error metrics. The error metrics shown are Absolute Error (AE), Mean Absolute Error (MAE), and Root Mean Squared Error (RMSE). The box plots comprise of data points that represent the mean value of error metrics for each workload in the dataset. Statistical significance is calculated between the baseline and BO methods. If the difference is not statistically significant, then the mean value is assumed to be the same between the two algorithms. Otherwise, the original mean value is used. Interestingly, we observe in Figure 9 that the models built by BO optimization are not always more accurate than our naive baseline predictor.

A counter-intuitive result here is that GP and GBRT have lower *prediction error* for execution time (Figure 9a and 9b) and execution cost (Figure 9c and 9d), respectively. This is despite the fact that, as shown in (§5.2.2), the ranking of the two algorithms is switched when it comes to the more general task of finding optimal solutions: GBRT is better for execution time and GP performs better for execution cost. We leave a thorough analysis of the relationship between model accuracy and optimization performance for future work.

Takeaways: Higher model accuracy is not necessarily a predictor for optimization performance. We also observe that the models created using BO methods are often not much more accurate than the simple baseline predictor that always predicts the mean of a set of randomly sampled cloud configurations.

5.3 Which optimization algorithm is better?

Now that we selected the best candidates from BO variants, we can compare them against the other optimization algorithms we discussed in Section 3. Just like the previous section, we compare both the execution time and the execution cost objective functions.

5.3.1 *Optimizing for execution time*

Figure 10 shows the comparison of different optimization algorithms when optimizing for execution time. We present the results for the scenario where we use the best algorithm-specific hyper-parameter values mentioned in Section 5.1.2 and a scenario where default hyper-parameter values are used. For DS_1 (shown in Figures 10a and 10b), BO with GBRT(PI) is the best candidate with tuned hyper-parameter values and BO with RF(PI) is only slightly better with the default. Using the best algorithms for execution time optimization of DS_1 provides up to 20% lower execution time. But in most cases, the benefit is less than 10%.

GBRT(PI) remains the best optimization algorithm for DS_2 (as shown in Figures 10c and 10d), with or without hyper-parameter tuning. With tuned hyper-parameters for all optimization algorithms, GBRT(PI) provides up to 10% better execution time compared to Random and LHS, but only up to 5% compared to the rest of the algorithms. Similar performance differences exist when using the default hyper-parameter values.

Takeaways: In most cases, BO with GBRT and PI acquisition function is able to find configurations with lower execution time than other methods. GBRT(PI) outperforms other algorithms with default as well as tuned values for hyper-parameters. However, RF(PI) performs slightly better than GBRT(PI) for an optimization budget of ≥ 24 runs.

5.3.2 *Optimizing for execution cost*

Using execution cost as the objective function to optimize also changes the characteristics of the search space, as previously shown in Figure 6. Therefore, in addition to using execution time as the objective function, we also want to compare different optimization algorithms using execution cost as the objective function. We have previously determined that the best optimization algorithm among the BO variants is GP instead of GBRT when optimizing for execution cost in our optimization scenarios.

As shown in Figure 11, when optimizing for execution cost, the performance differences between the best optimization algorithm and the rest are much more pronounced. When optimizing for execution cost for workloads in DS_1, using GP(EI) provides up to 29% and 40% lower execution cost than other algorithms with tuned and default hyper-parameters, respectively.

As shown in Figures 11c and 11d, the benefits of using GP(LCB) are more apparent when optimizing for workloads in the larger search space (DS_2). The performance benefit of using GP(LCB) is up to 40%.

Takeaways: The best optimization algorithm based on our results does not change with using either the default or tuned values for algorithm-specific hyper-parameters. The difference in the performance of optimization algorithms is much more pronounced when optimizing for execution cost and particularly when optimizing for execution cost on the large search space of DS_2.

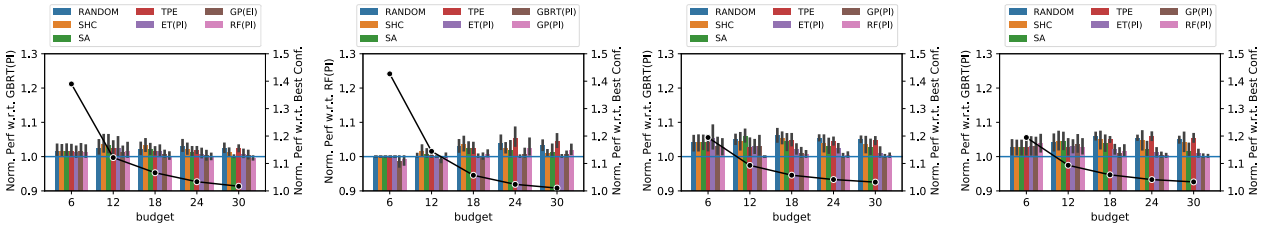
5.4 What is the best method for online optimization?

Prior work in choosing cloud configurations concentrates on *offline* optimization: the user waits until the completion of the optimization runs before deploying the workload in a production setting. However, that might require a lengthy offline benchmarking phase, which may not be feasible in some deployments. In this section, we evaluate an alternative *online* scenario, in which a user tests different configurations during production runs. In this case, the problem is that testing configurations online may lead to excessively long execution times for the production runs.

To evaluate this scenario, we assume the existence of a threshold beyond which the execution time obtained with a configuration is considered a *violation* to a service level objective (SLO). For configuring the various optimization algorithms, we use the respective default acquisition functions, as mentioned in the prior sections. For the values of algorithm-specific hyper-parameters, we use $\xi = 0.01$, $\kappa = 1.0$, and $\gamma = 0.1$, since, among the hyper-parameter values we have tested, these values provide the highest emphasis on exploitation rather than exploration. Emphasis on exploitation would mean that optimization algorithms would perform more conservatively, searching for better configurations in the vicinity of already known good configurations, and thus leading to fewer objective function value violations.

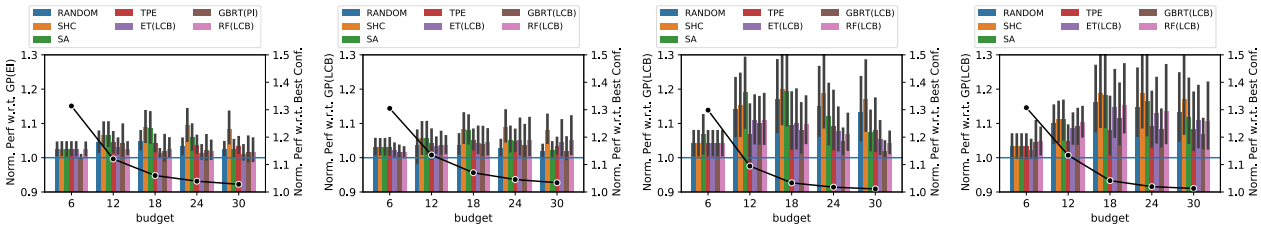
Figure 12 shows the average number of violations during optimization across all workloads in DS_1 and DS_2. The threshold is marked in the figure as a normalized runtime threshold, i.e., a threshold of 1.5 means that any configuration that leads to an objective function value that is more than 1.5$\times$ the minimum in the search space is considered a violation. Optimization algorithms that test configurations with widely different execution times result in a larger number of violations and are therefore less suitable for online optimization.

We run the optimization algorithm for 30 iterations. We will assume that the three initial samples are executed offline even with online optimization, so they are not counted as violations. We can see in Figure 12 that BO methods have fewer violations than



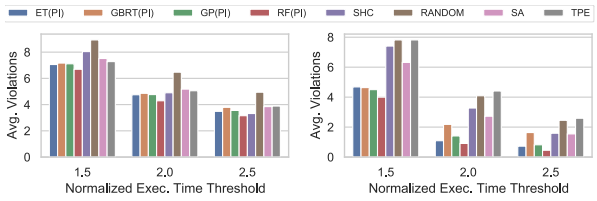
(a) DS_1 Tuned hyper-parameters **(b) DS_1 Default hyper-parameters** **(c) DS_2 Tuned hyper-parameters** **(d) DS_2 Default hyper-parameters**

Figure 10: Comparison of optimization algorithms when optimizing for execution time using tuned or default algorithm-specific hyper-parameters.

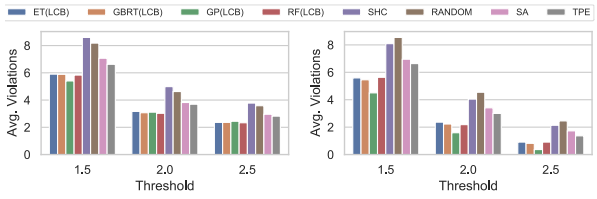


(a) DS_1 Tuned hyper-parameters **(b) DS_1 Default hyper-parameters** **(c) DS_2 Tuned hyper-parameters** **(d) DS_2 Default hyper-parameters**

Figure 11: Comparison of optimization algorithms when optimizing for execution cost using tuned or default algorithm-specific hyper-parameters.



(a) DS_1: Execution time **(b) DS_2: Execution time**



(c) DS_1: Execution cost **(d) DS_2: Execution cost**

Figure 12: Aggregate number of SLO violations for varying thresholds (as a factor of the objective function value of the best configuration in the search space). Each optimization algorithm is using the default algorithm-specific hyper-parameter values.

other optimization algorithms. BO variants have up to 2× fewer violations compared to Random search, SHC, SA, and TPE. We also analyzed the performance of the optimization algorithms with these configurations values and the findings from previous sections still hold true. In particular, BO(GP) and BO(GBRT) provide the best optimization performance when optimizing for execution cost and execution time, respectively. Similarly, in the latter case, BO(GBRT) is closely matched by BO(RF) when the optimization budget is high.

Takeaways: When doing online optimization using the hyper-parameter values that emphasize exploitation, BO variants cause up to 2× fewer performance constraint violations, on average, compared to other algorithms, especially Random Search and SHC. The reason is that Random Search and SHC inherently lack an objective function model.

Table 4: Number of extra periodic repetitions of the workload required in production to break-even with the time and cost of optimization when optimizing for execution time and execution cost, respectively.

Algorithm	Execution time (min–mean–max)		Execution cost (min–mean–max)	
	DS_1	DS_2	DS_1	DS_2
Random	71–361–1261	88–329–578	69–275–858	80–178–364
SHC	72–257–703	55–312–717	66–396–1521	98–195–365
SA	66–276–815	54–214–432	66–326–1016	89–167–262
TPE	50–263–1065	56–326–1008	66–265–946	45–110–234
BO(ET)	51–234–647	59–151–282	35–240–1997	55–152–249
BO(RF)	34–207–706	42–136–270	11–305–4436	38–148–242
BO(GBRT)	27–291–812	54–235–616	23–243–2003	37–99–154
BO(GP)	56–267–877	61–157–302	12–275–2112	44–109–217

5.5 When are more optimization runs worth the extra cost?

Cloud configuration optimization is generally targeted towards periodic workloads. Therefore, it is important to understand how many periodic runs are required to break-even with the extra cost and time required for performing the optimization runs. In this section, we will present these break-even points for our workloads. We generate the break-even points using a ratio between the cost or time it takes to do more optimization runs and the improvement in the objective function value as a result of those extra runs:

$$\frac{Cost(x_{bl+s}) - Cost(x_{bl})}{f(x_{bl}) - f(x_{bl+s})}$$

where $f(x_{bl})$ and $f(x_{bl+s})$ are the best objective function value after running the optimization for baseline optimization budget bl ($bl = 6$ runs) and for budget bl plus extra s steps, respectively. In turn, the cost function $Cost(x)$ is the monetary and time cost of doing optimization runs when optimizing for the execution cost and execution time objective functions, respectively. This ratio gives us the number of additional production runs that are necessary to fully amortize the increment in the optimization costs.

Table 4 shows the min, mean and max values for the number of extra production runs of workloads required to break even. The table presents summary statistics that provide an overall picture of the extra runs that are required across all workloads in a dataset for

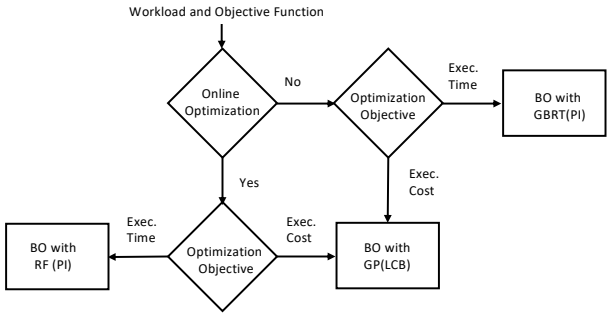


Figure 13: Decision tree for selecting an optimization algorithm.

a given objective function. The more detailed statistics for individual workloads can be found in the accompanying code and data repository [1]. The results show that a job may have to be repeatedly executed anywhere between a dozen times to a few thousand times to justify further optimization. Therefore, depending on the frequency with which a workload is executed, it might or might not be worth spending more time optimizing the cloud configuration. On average, BO variants require fewer extra runs of the workload to break-even, thus providing more improvement for a given increase in the optimization budget.

Takeaways: BO variants provide lower mean break-even points than other algorithms for both the execution time and execution cost objective functions. Hence, they provide more improvement given a fixed extra optimization budget compared to other optimization algorithms such as SA, SHC, TPE, and Random Search.

5.6 Summary

Overall, one of the main conclusions is that BO outperforms other optimization algorithms that we have evaluated. Algorithm-specific hyper-parameter tuning is less important than the choice of the initial samples and optimization budget. We found no clear best setting of hyper-parameter values (especially the number of initial samples) for any of the algorithms: the best setting varies across workloads and objective functions. For Bayesian optimization variants, different acquisition functions perform similarly in most cases, but overall PI works well when optimizing for execution time and LCB is better when optimizing for execution cost. BO with GBRT and PI acquisition function is able to outperform other algorithms by up to 20% and 10% when optimizing for execution time for DS_1 and DS_2, respectively. BO with GP and LCB provides up to 40% better performance when optimizing for execution cost in both DS_1 and DS_2. According to our evaluation, BO variants also provide fewer constraint violations for online optimization and more improvement in the objective function value as a result of a marginal increase in the optimization budget.

6. CHOOSING A BLACK-BOX OPTIMIZATION ALGORITHM

Figure 13 shows the decision tree based on the evaluation in this work. For offline optimization, if the objective function is execution time, then BO with GBRT(PI) is the best choice. Whereas for execution cost, the preferred choice is BO with GP(LCB). For online optimization, we found that BO variants in our experiments had fewer constraint violations, which can be a critical aspect of an online optimization algorithm. Specifically, RF(PI) and GP(LCB) have a lower average number of violations when optimizing for execution time and execution cost. Therefore, BO with RF(PI) is a better choice for online optimization, not only because of having a lower average number of constraint violations but also because it can perform comparably to the best algorithm for execution time optimization, i.e., BO with GBRT.

7. DISCUSSION

In this section, we will discuss orthogonal techniques that can be used in addition to the methods we have discussed in this work.

Early termination techniques to timeout likely bad cloud configurations can be used to decrease the time and monetary cost of optimization. These enhancements have not been included in our evaluation because they can be incorporated into many of the optimization algorithms and thus are not a factor that can be used to differentiate one optimization algorithm from another.

Workload fingerprinting, along the lines of [27, 38], can be used by all BO algorithms as a prior, but it has an extra cost associated with it. We did not consider fingerprinting in our evaluation to provide a fairground of comparison across BO and non-BO algorithms. The evaluation already shows that BO is often superior, even without fingerprinting.

In a cloud environment, we can execute multiple optimization runs at the same time, thus parallelizing the optimization process. Random sampling is easier to parallelize than *sequential* optimization algorithms such as the BO variants described in this paper. With random sampling, virtually all configurations can be tested independently of each other. It allows users to deploy and run a set of clusters and stop the rest of the optimization runs when the best configuration has been found. Bayesian Optimization also has parallel variants [20, 32, 35]. The evaluation of the effectiveness and efficiency of these parallel optimization algorithms is an interesting avenue for future work.

8. CONCLUSION

In this work, we evaluate the performance of many popular black-box optimization algorithms in the context of choosing cloud configurations. We evaluated 8 algorithms and 23 workloads using 2 objective functions. Our evaluation showed that algorithm-specific hyper-parameter tuning is less important than the choice of the initial samples and the optimization budget. We found no clear set of optimal hyper-parameter values for any of the algorithms. While some existing works have used Bayesian optimization with Gaussian processes and Bayesian optimization with extra trees, our evaluation suggests that there is no silver bullet. Bayesian optimization with GBRT and GP perform better when optimizing for execution time and execution cost, respectively. We also examine the problem of finding optimal cloud configurations in an online setting and perform a break-even analysis to evaluate when performing more optimization runs is worth it.

Acknowledgement

Muhammad Bilal was supported by a fellowship from the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) program funded by the European Commission (EACEA) (FPA 2012-0030). This research was supported by Fundação para a Ciência e a Tecnologia (FCT), under projects UIDB/50021/2020 and CMUP-ERI/TIC/0046/2014.

9. REFERENCES

[1] BBO Arena. `https://github.com/MBtech/bbo-arena`. [Online; accessed 10/07/2020].

[2] BBO Arena: Best hyper-parameters. `https://github.com/MBtech/bbo-arena/`

blob/master/docs/best-hyperparams.md. [Online; accessed 10/07/2020].

[3] HiBench Bigdata benchmark. https://github.com/Intel-bigdata/HiBench. [Online; accessed 10/07/2020].

[4] HyperOpt. https://pythonhosted.org/pyDOE/. [Online; accessed 10/07/2020].

[5] Scout Dataset. https://github.com/oxhead/scout. [Online; accessed 10/07/2020].

[6] SigOpt. https://sigopt.com/. [Online; accessed 10/07/2020].

[7] SkOpt. https://scikit-optimize.github.io. [Online; accessed 10/07/2020].

[8] Solid. https://github.com/100/Solid. [Online; accessed 10/07/2020].

[9] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[10] C. Ansótegui, M. Sellmann, and K. Tierney. A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms. In *International Conference on Principles and Practice of Constraint Programming (CP)*, 2009.

[11] J. Bergstra, D. Yamins, and D. Cox. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *International Conference on Machine Learning (ICML)*, 2013.

[12] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for Hyper-Parameter Optimization. In *Advances in Neural Information Processing Systems (NIPS)*, 2011.

[13] M. Bilal and M. Canini. Towards Automatic Parameter Tuning of Stream Processing Systems. In *Symposium on Cloud Computing (SoCC)*, 2017.

[14] L. Breiman. Random Forests. *Machine learning*, 45(1), 2001.

[15] S. Duan, V. Thummala, and S. Babu. Tuning Database Configuration Parameters with iTuned. *PVLDB*, 2(1):1246–1257, 2009.

[16] A. C. Elliott and W. A. Woodward. *Statistical Analysis Quick Reference Guidebook: With SPSS Examples*. Sage, 2007.

[17] M. Feurer and F. Hutter. Hyperparameter Optimization. In *Automated Machine Learning*. Springer, 2019.

[18] P. Geurts, D. Ernst, and L. Wehenkel. Extremely randomized trees. *Machine learning*, 63(1), 2006.

[19] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley. Google Vizier: A Service for Black-Box Optimization. In *International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2017.

[20] J. González, Z. Dai, P. Hennig, and N. Lawrence. Batch Bayesian Optimization via Local Penalization. In *Artificial Intelligence and Statistics (AISTATS)*, 2016.

[21] N. Hansen. The CMA Evolution Strategy: A Comparing Review. In *Towards a New Evolutionary Computation*. Springer, 2006.

[22] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Science & Business Media, 2009.

[23] M. Hoffman, E. Brochu, and N. de Freitas. Portfolio Allocation for Bayesian Optimization. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2011.

[24] H. Hoos, K. Leyton-Brown, and F. Hutter. An Efficient Approach for Assessing Hyperparameter Importance. In *International Conference on Machine Learning (ICML)*, 2014.

[25] C.-J. Hsu, V. Nair, V. W. Freeh, and T. Menzies. Arrow: Low-Level Augmented Bayesian Optimization for Finding the Best Cloud VM. In *International Conference on Distributed Computing Systems (ICDCS)*, 2018.

[26] C.-J. Hsu, V. Nair, T. Menzies, and V. Freeh. Micky: A Cheaper Alternative for Selecting Cloud Instances. In *International Conference on Cloud Computing (CLOUD)*, 2018.

[27] C.-J. Hsu, V. Nair, T. Menzies, and V. W. Freeh. Scout: An Experienced Guide to Find the Best Cloud Configuration. *arXiv preprint arXiv:1803.01296*, 2018.

[28] F. Hutter, L. Kotthoff, and J. Vanschoren. *Automated Machine Learning*. Springer, 2019.

[29] J. Kennedy and R. Eberhart. Particle swarm optimization. In *International Conference on Neural Networks (ICNN)*, 1995.

[30] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *The Journal of Machine Learning Research*, 18(1), 2017.

[31] M. A. Muñoz, Y. Sun, M. Kirley, and S. K. Halgamuge. Algorithm selection for black-box continuous optimization problems: A survey on methods and challenges. *Information Sciences*, 317, 2015.

[32] J. Očenášek and J. Schwarz. The Parallel Bayesian Optimization Algorithm. In *The State of the Art in Computational Intelligence*. Springer, 2000.

[33] S. Shan and G. G. Wang. Survey of modeling and optimization strategies to solve high-dimensional design problems with computationally-expensive black-box functions. *Structural and Multidisciplinary Optimization*, 41(2), 2010.

[34] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[35] J. Wang, S. C. Clark, E. Liu, and P. I. Frazier. Parallel Bayesian Global Optimization of Expensive Functions. *Operations Research*, 2020.

[36] C. K. Williams and C. E. Rasmussen. Gaussian Processes for Regression. In *Advances in Neural Information Processing Systems (NIPS)*, 1996.

[37] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang. A Smart Hill-Climbing Algorithm for Application Server Configuration. In *International Conference on World Wide Web (WWW)*, 2004.

[38] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz. Selecting the Best VM across Multiple Public Clouds: A Data-Driven Performance Modeling Approach. In *Symposium on Cloud Computing (SoCC)*, 2017.

[39] 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 *Symposium on Cloud Computing (SoCC)*, 2017.