Discrete Bayesian Optimization via Machine Learning

Roberto Sala[a,], Bruno Guindani[a], Danilo Ardagna[a], Alessandra Guglielmi[a],

[a]*Politecnico di Milano, Piazza Leonardo da Vinci, 32, Milano, 20133, Italy*
name.surname@polimi.it

Abstract

Bayesian Optimization (BO) is a family of powerful algorithms designed to solve complex optimization problems involving expensive black-box functions. These sequential algorithms iteratively update a surrogate model of the objective function (OF), effectively balancing exploration and exploitation to identify near-optimal solutions within a limited number of iterations. Originally designed for continuous, unconstrained domains, its efficiency has inspired adaptations for discrete, constrained optimization problems. On the other hand, Machine Learning (ML) models allow accurate predictions for black-box functions, although they typically require large amounts of data for training. Leveraging the strengths of BO and ML, research tackles the challenge of identifying optimal configurations in the context of cloud computing. This paradigm has become pervasive due to its ability to provide flexible and scalable resources. Identifying the optimal hardware-software configuration is essential for minimizing costs while meeting Quality of Service constraints. This task involves solving complex optimization problems over multidimensional discrete domains and black-box objective functions and constraints, within a limited number of iterations. To address this challenge, this work introduces *d-MALIBOO*, a BO-based algorithm that integrates ML techniques to enhance the efficiency of finding near-optimal solutions in discrete and bounded domains. While BO builds the surrogate model of the OF, ML models determine the feasible region of the black-box constraints and guide the BO algorithm toward promising regions of the discrete domain. Furthermore, we introduce an ε-greedy approach to favor exploration in domains with multiple local optima. Experimental results show that our algorithm outperforms OpenTuner, a popular framework for constrained optimization, by reducing the average regret by 29%, and SVM-CBO, a BO-based algorithm that integrates SVM models to determine the feasible region, by 82%.

Keywords: Discrete Variables, Bayesian Optimization, Machine Learning, Black-box Optimization, Cloud Computing

1. Introduction

The Cloud computing paradigm has become pervasive across various domains, including industry, e-commerce, blockchain technology, and drug discovery [1, 2, 3, 4]. Among these, it plays a fundamental role in the development of Artificial Intelligence (AI) and Large Language Models [5, 6]. Cloud computing offers flexible and scalable resources, leveraging a vast pool of virtually unlimited Virtual Machines (VMs) housed in data centers. The diverse range of use cases and the heterogeneous nature of machines within these data centers make it essential to identify optimal configurations for both applications and infrastructure. Inefficient configurations can lead to substantial costs for providers and users alike [7]. Moreover, these applications often require Quality of Service (QoS) criteria to be met, such as achieving a maximum model accuracy or maintaining a minimum pipeline response time. This requirement translates into constrained optimization problems over discrete, multidimensional domains. In the aforementioned scenarios, frequently each application run incurs a high cost, either in terms of time or money. Consequently, the optimization process must be carried out in a limited number of iterations. Consider, for example, determining the optimal hardware-software configuration for an AI application. The goal is to maximize the model accuracy (at training time) guaranteeing low response time and costs (at inference time when the AI model is in production). In such cases, each configuration test necessitates re-training the AI model, resulting in substantial costs for the application provider.

Bayesian Optimization (BO) is a powerful category of methods for tackling global optimization problems that involve expensive-to-evaluate black-box functions [8]. BO operates iteratively: at each step, it updates the surrogate model of the objective function (OF) and selects the next point to evaluate by balancing the exploration-exploitation trade-off. This trade-off involves exploring regions with high uncertainty and exploiting areas with promising OF values. This iterative process ensures convergence to near-optimal solutions within a minimal number of evaluations [8]. BO is well-suited for continuous variables, unbounded domains, and continuous OF. However, in the scenarios mentioned earlier, the optimization domains are

typically discrete and bounded. This problem setup introduces additional challenges, such as selecting the best point to evaluate in the discrete domain and avoiding repeated evaluations of the same point.

This work presents *d-MALIBOO* (discrete MAchine Learning In Bayesian OptimizatiOn), a BO-based algorithm designed to solve constrained optimization problems in discrete domains. Specifically, our focus is on optimizing Edge-Cloud computing and high-performance computing applications. In our previous work [9], we introduced MALIBOO, a framework that integrates ML techniques into the BO algorithm to determine the feasibility region from black-box constraints. This algorithm is designed to reduce the costs of recurring computing tasks by incorporating a technique to prevent the repetition of previously evaluated points. Building on the original MALIBOO, we introduce several novel ML-based approaches for estimating black-box constraints and improving the search for optimal solutions within discrete domains. Additionally, we propose an ε-greedy approach to balance the exploration-exploitation trade-off, particularly in complex domains with multiple local optima, reducing the risk of the optimization process stalling. Furthermore, we test the novel approach across a range of applications, from AI to edge and HPC domains. Finally, we compare our technique with two alternative methods from the literature, SVM-CBO and OpenTuner [10, 11]. Experimental results show that d-MALIBOO outperforms the original MALIBOO framework, reducing the average regret by 9% across all applications. Additionally, d-MALIBOO reduces the average regret of OpenTuner by over 29%, while SVM-CBO struggles with handling black-box constraints, leading to an 82% increase in average regret. This work is an extension of our previous study [12] and provides a more comprehensive description of the framework, along with a deeper experimental campaign that validates the proposed approach further.

This paper is structured as follows. Section 2 presents the mathematical formulation of our problem of interest. Section 3 provides an overview of BO techniques and MALIBOO. Section 4 introduces the d-MALIBOO algorithm, including the proposed ML-based approaches and the integration of the ε-greedy approach. Section 5 presents the results of experiments to validate our framework. Section 6 discusses related works, and finally, Section 7 offers conclusions and suggests future developments.

2. Problem Overview

In this work, we present a BO-based algorithm designed to find the optimal configuration for discrete optimization problems involving black-box target functions and constraints. Specifically, we focus on applications in Edge-Cloud and HPC systems. In this context, the optimization involves identifying the optimal values for various software and hardware parameters, such as buffer size, the number of parallel nodes or GPUs, and application-specific settings like penalization parameters or the number of algorithm iterations. The OF typically represents the cost of running an application, or the value of a specific performance metric, under a given configuration. Constraints may include execution time, resource utilization, or quality thresholds for a model. For both the objective and constraints, the analytical dependence on input configurations is unknown, which is why these functions are treated as black-box. Let $f(\cdot)$ denote the OF to minimize, and $g_j(\cdot)$ represent scalar black-box constraints for each j. The mathematical formulation of the problem under study is as follows:

$$\begin{aligned} &\min_{x \in D} f(x) + \eta \\ &\text{s.t. } g_j(x) \in [G^j_{min}, G^j_{max}],\ j = 1, \ldots, C \end{aligned} \tag{1}$$

Here, x denotes the input vector of configurations, also referred to as parameters or features, representing the set of variables being optimized. η is a noise term that accounts for variability in the measurement of the OF. This term is particularly relevant in cloud environments, where the OF, often related to execution time, may vary due to resource contention or network congestion. $D \in \mathbb{R}^d$ represents the d-dimensional discrete optimization domain, while C denotes the total number of black-box constraints. The constraint functions $g_j(\cdot)$ may either depend on or be independent of $f(\cdot)$. Typically, $d \leq 20$, and the OF $f(\cdot)$ is continuous, black-box, expensive to evaluate, and has unknown derivatives [8]. The only way to gather information on such a function is by evaluating the function itself. This formulation is quite versatile for modeling a range of cloud optimization scenarios, including hyperparameter tuning and energy efficiency [13, 14].

The original BO algorithm usually deals with optimization problems in unconstrained domains. However, in this study, we aim to challenge the assumption of continuity of the optimization domain and instead explore discrete, bounded domains. These characteristics significantly increase the

complexity of the optimization problem, though they model many realistic scenarios. Consider, for example, the hyperparameter tuning problem for an HPC system, for which the features are the number of cores, the memory size, the number of nodes used, etc.; we address such scenarios in our experimental analysis (Section 5). In these cases, we are not considering all possible integer values within an interval for each feature, nor are these values equispaced.

3. Bayesian Optimization Background

In this section, we discuss the classical BO algorithm (Section 3.1) and the MALIBOO algorithm (Section 3.2), which is the starting point of our novel framework.

3.1. The Bayesian Optimization algorithm

BO is a robust method for optimizing black-box OFs, as it can achieve near-optimal solutions within relatively few iterations [8]. This is especially valuable in scenarios where each evaluation of the OF is costly or time-intensive, as highlighted in Section 2. For example, consider tuning AI applications that require a large number of virtual machines (VMs) or drug discovery processes where each evaluation is computing intensive. In such cases, conducting an exhaustive search over the domain is economically unsustainable. Instead, an efficient strategy is needed to achieve strong performance with minimal iterations.

The classical BO algorithm can be summarized in the following steps:

1. Select a set of initial points with some strategy (e.g. random).
2. Update the surrogate model for the OF.
3. Update the acquisition function (AF) based on the surrogate model.
4. Identify the point that maximizes the AF.
5. Evaluate the OF at that point.
6. Repeat steps (2–5) until the maximum number of iterations is reached.

The key components of the BO algorithm are the surrogate model, typically a Gaussian Process (GP), and the AF.

A *Gaussian Process* is a statistical model used to describe knowledge about the OF [15]. It assumes that each unknown value of $f(x)$ is drawn from a Gaussian distribution whose mean, $\mu_n(x)$, and variance, $\sigma_n^2(x)$, depend on x and the history of n previously observed points. Specifically, given the history

$H_n = \{(x_i, f(x_i)), i = 1, \ldots, n\}$ and a configuration x, the distribution of $f(x)$ is:

$$f(x)|H_n \sim \pi_x(\cdot|H_n) = N(\mu_n(x), \sigma_n^2(x, x)). \tag{2}$$

The mean function $\mu_0(\cdot)$ and the kernel function $\sigma_0^2(\cdot, \cdot)$, which define the GP model, serve as its hyperparameters. These functions are iteratively updated after each evaluation of the OF to obtain the *posterior distribution*, characterized by $\mu_n(\cdot)$ and $\sigma_n^2(\cdot)$, as shown in Equation (2). While a constant mean function, $\mu_0(\cdot) = \mu_0$, is commonly used [8], the choice of the kernel function varies widely in the literature due to its influence on the smoothness of the GP model. Popular kernel options include the *Radial Basis Function* (RBF) and the *Matérn* covariance function [8]. Specifically, the Matérn kernel used in our frameworks is defined as follows:

$$\sigma_0^2(x_i, x_j) = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\sqrt{2\nu}\ d(x_i, x_j)\right)^{\nu} K_\nu \left(\sqrt{2\nu}\ d(x_i, x_j)\right), \tag{3}$$

where $\nu > 0$ is the smoothness parameter, $\Gamma(\cdot)$ and $K_\nu(\cdot)$ denote the gamma and the modified Bessel functions, respectively [15].

The other core element of the BO algorithm is the *Acquisition Function* (AF), denoted as $\alpha(x)$. The AF evaluates the utility of sampling $f(x)$ at a given point x, balancing exploration and exploitation. The AF is derived from the *posterior distribution* of the OF and is computationally much cheaper to evaluate than the OF itself. At each iteration, the point that maximizes the AF is identified, and the OF is evaluated at that point [8]. In this paper, we focus on the *Expected Improvement* (EI), defined as:

$$\mathrm{EI}_n(x) \coloneqq E_n\left[\max([f_n^* - f(x)], 0)\right], \tag{4}$$

where $f_n^* = \min_{m \leq n} f(x_m)$ represents the minimum value observed so far. Other commonly used AFs are the Upper Confidence Bound (UCB) and the Probability of Improvement (PoI) [8].

The classical BO algorithm is well-suited for continuous variables and unconstrained domains. The second limitation has been addressed in MALIBOO [9]. The key idea of this approach is to leverage the predictive capabilities of an ML model to estimate the feasible region directly within the AF, thereby reducing the number of configurations visited that are unfeasible with respect to the constraints. The next section presents the main contributions of MALIBOO, highlighting both its strengths and weaknesses, which we address in this work.

3.2. The MALIBOO algorithm

MALIBOO is a BO-based algorithm that incorporates an ML technique to assess whether the black-box constraints are satisfied at a given point [9]. Specifically, this evaluation is integrated within the AF to reach its maximum value in regions deemed feasible by the model, based on the history of previously observed points. One ML-integrated AF proposed in [9] is the following:

$$\bar{\alpha}(x) = \alpha(x)\ \mathbb{1}\{\tilde{g}(x) \in [G_{min}, G_{max}]\}, \tag{5}$$

where $\tilde{g}$ is the surrogate ML model to estimate the value of the black-box constraints function $g(x)$. This straightforward approach effectively avoids evaluating unfeasible points, thereby preventing a waste of resources. The ML model is updated at each iteration, becoming increasingly precise as the number of evaluations grows.

In this paper, we propose a framework specifically designed to handle purely discrete variables, building on the strengths of MALIBOO while addressing its limitations. Firstly, the use of an indicator function in the AF (see Equation (5)) can hinder the efficiency of the algorithm during early iterations due to the limited accuracy of ML models at this stage. This issue may exclude optimal points located at the boundaries of the domain, a situation that frequently arises in practice. Moreover, in particularly complex optimization scenarios, we have observed that the search process often stalls after a few iterations, reducing the overall effectiveness of the optimization.

4. d-MALIBOO

Moving from continuous to discrete domains poses a challenge for BO, particularly in maximizing the AF. A naive approach assumes that the OF can only be evaluated at integer values, by optimizing the AF under the assumption that all variables are continuous, and replacing the values of integer-valued variables with their nearest integers. This can lead to a mismatch between the points where the AF reaches high values and where the actual evaluations occur [16]. Furthermore, once the argmax of the AF is approximated, there is a risk of proposing evaluations at points that have already been observed, particularly during the exploitation phase when the search area becomes more limited. MALIBOO [9] addresses this by relaxing the discrete domain into a continuous one, maximizing the AF, and then rounding to the nearest discrete point. To further avoid repeated evaluations

of previously observed points, MALIBOO employs a queue that stores the last q observed points, ensuring that the argmax of the AF is not rounded to any point in the queue. Without introducing additional specialized techniques to address optimization in discrete domains beyond the standard requirements of BO, even the original MALIBOO experiences a performance stall within the iterations in such settings, as shown in Section 5.

In this section, we present several novel approaches for managing optimization scenarios involving discrete variables. First, we propose two distinct methods for estimating black-box constraints (Section 4.1), along with four different strategies to speed up the search for the optimum (Section 4.2). Then, we incorporate the ε-greedy approach to encourage exploration when improvements appear to stall (Section 4.3). Note that all these functionalities can be used together and henceforth we will refer to a set of these approaches as a *combination* for *d-MALIBOO*. Finally, we summarize the complete *d-MALIBOO* algorithm in Section 4.4.

4.1. Evaluation of black-box constraints

Accurately estimating the feasible region in a constrained BO framework is essential, as a precise definition of the feasible region reduces the likelihood of testing infeasible configurations, thereby lowering both costs and execution time. In this section, we describe how we integrate ML models into the AF to determine the feasibility region. We refer to these approaches as "ML constraint", as also shown in Algorithm 1.

Let $\alpha(x)$ be the original AF for the BO algorithm, such as EI, UCB or PoI (see Section 3.1). The optimization process must estimate the C black-box constraints $g_j(x) \in [G^j_{min}, G^j_{max}],\ j = 1, \ldots, C$. MALIBOO [9] suggests multiplying the current AF by the evaluation of the feasible region, which is modeled using an ML model trained on the current history H_n (see Equation (5)). This evaluation is integrated through an indicator function, which introduces a jump discontinuity in the values of the AF at the boundary of the estimated feasibility domain. We refer to this approach as *indicator*. However, when only a few points have been evaluated, the ML model may lack accuracy and exclude optimal points that lie on the boundary of the feasible domain, especially in high-dimensional domains. Therefore, we propose a smoother approach to integrating the evaluation of the constraint function within the AF, which we refer to as the *probability* approach:

$$\bar{\alpha}(x) = \alpha(x)\ \mathbb{P}\left(\tilde{g}(x) \in [G_{min}, G_{max}]\right), \tag{6}$$

where $\tilde{g}$ is an ML model that predicts the value of the constraint function at a given point x. This probability is computed with a classifier, such as Ridge, which transforms the decision function output $\tilde{g}$ with the sigmoid function to estimate it.

4.2. Evaluation of the objective function

Dealing with vast discrete domains requires multiple evaluations of the OF to build an accurate surrogate model. A GP alone struggles to capture the peculiarities of the OF within a discrete domain, where the concept of proximity between points becomes less meaningful as the dimensionality of the domain increases. This concept of proximity is fundamental to defining the GP kernel. Building on the approach for handling black-box constraints, we propose four distinct approaches that leverage the capabilities of ML to learn from collected data and predict the values of black-box functions, thereby assisting the AF in identifying the optimum. We refer to these approaches as "ML target" (see Algorithm 1).

In the first approach, *indicator*, we train an ML model $\tilde{f}$ on the history H_n to predict whether the OF evaluation at a new point is greater than or equal to the evaluation at the best point thus far, f^*. This prediction is then integrated into the AF through an indicator function, as follows:

$$\bar{\alpha}(x) = \alpha(x)\ \mathbb{1}\{\tilde{f}(x) \leq f^*\}. \tag{7}$$

Following the same concept used in the treatment of constraints, a smoother evaluation of the location of points with lower OF values can be achieved by computing the probability that $\tilde{f}(x) \leq f^*$, referred to as *probability* approach:

$$\bar{\alpha}(x) = \alpha(x)\ \mathbb{P}\left(\tilde{f}(x) \leq f^*\right). \tag{8}$$

We now introduce the *sum* approach. Let $\tilde{f}$ be the ML that estimates the value of the OF at a given point x. Let $\alpha(D)$ and $\tilde{f}(D)$ be the vectors of evaluations of the AF and the regression model, respectively, over the entire discrete domain D, and x being the jth element in D. The AF is as follows:

$$\bar{\alpha}_j = (1 - \gamma_t)\ m_j(\alpha(D)) + \gamma_t\ m_j(-\tilde{f}(D)). \tag{9}$$

Here, $\gamma_t \in [0, 0.5]$ is a weight parameter that increases exponentially with BO iterations t, as the accuracy of the ML model improves. The operator

$m_j(\cdot)$ is the min-max normalization applied to the jth element in order to scale two quantities that may have different magnitudes. Specifically, given a vector v, the min-max normalization operator scales its jth element as follows: $m_j(v) = \frac{v_j - \min_p v_p}{\max_p v_p - \min_p v_p} \in [0, 1]$. The minus sign of the second term is required to find the minimum over the discrete domain. Finally, we introduce the *product* approach. In this case, we scale the evaluation of the AF using the estimate from the ML model as follows:

$$\bar{\alpha}_j = \alpha(x)\ m_j(\tilde{f}(D)). \tag{10}$$

Note that while the first two approaches penalize regions where better configurations are unlikely to be found compared to the best solution discovered so far, the latter two approaches weight the value of the AF by the estimate provided by the ML model.

The *sum* and *product* techniques are highly effective but can only be applied in scenarios involving purely discrete variables, as they require exhaustive evaluation of the AF over the entire discrete domain. However, the evaluation time for these techniques is remarkably small compared to the OF evaluation time, even for large optimization domains.

4.3. ε-greedy approach

In our preliminary experiments, we observed that in high-dimensional domains with strict constraints, improvement in the OF evaluation may stagnate over iterations. Drawing inspiration from the Reinforcement Learning literature [17], we integrated the ε-greedy approach into the BO algorithm to address this issue. This approach involves selecting a random point with a probability of ε, rather than always maximizing the AF. Most importantly, we only select among points that the ML model $\tilde{g}$ predicts as feasible.

4.4. The algorithm

Algorithm 1 outlines our *d-MALIBOO* technique, specifically designed for handling optimization problems with discrete variables in bounded domains. First, the history H_n is initialized with evaluations of n_0 initial points, randomly sampled within the domain (lines 2–4). This initialization step is necessary to train the GP and ML models in the starting phase. At each iteration, a Bernoulli distribution with parameter ε is sampled to determine whether the next point will be selected randomly within the feasible domain estimated by the *ML constraint* model (lines 6–8). If not, the ML models

for constraints and the OF are updated, along with the GP and the AF. The AF is then maximized to determine the next point for evaluation (lines 10–12). In either case, the chosen point is evaluated, and the history H_n is updated (line 14). Finally, the algorithm returns its estimated optimum, i.e., the configuration with the lowest OF value among those observed (line 16).

Algorithm 1 *d-MALIBOO algorithm*

1: **Input**: n_0, N, GP, *ML constraint*, *ML target*, ε
2: **Initialization**: History $H_n \leftarrow []$
3: evaluate $f(\cdot)$ in n_0 randomly chosen initial points
4: $H_n \leftarrow \{(x_i, f(x_i)), i = 1, \ldots, n_0\}$
5: **for** iterations $n = 1 : N$ **do**
6: $\quad\varepsilon$-greedy_eval $\leftarrow$ sample from a $Be(\varepsilon)$
7: $\quad$**if** ε-greedy_eval **then**
8: $\quad\quad$pick a random point x_{n+1} considering the *ML constraint* approach
9: $\quad$**else**
10: $\quad\quad$train *ML constraint*, *ML target* models with data in H_n
11: $\quad\quad$update the posterior distribution of the GP model and $\tilde{\alpha}(\cdot)$ with data in H_n
12: $\quad\quad$find point x_{n+1} which maximizes the AF $\tilde{\alpha}(\cdot)$
13: $\quad$**end if**
14: $\quad$evaluate $f(x_{n+1})$, add the evaluation to H_n
15: **end for**
16: **return** $\widehat{x} = \arg\min_{x \in H_n} f(x)$

5. Experimental Results

In this section, we present an extensive experimental campaign to validate the *d-MALIBOO* technique. Specifically, the testing applications include tuning configurations of public and private edge-cloud servers and optimizing hyperparameters of HPC applications, as detailed in Section 5.1. Furthermore, we compare the performance of our algorithm with MALIBOO and two state-of-the-art (SOTA) algorithms described in Section 5.2. Section 5.3 outlines the experimental setup, including hyperparameter configurations and hardware specifications, while Section 5.4 provides the empirical results

of our experimental campaign. The experimental data are available on Zenodo[1].

5.1. Reference Applications

This section introduces the applications used to evaluate the *d-MALIBOO* algorithm. These real-world scenarios include common tasks in HPC and edge-cloud computing, ranging from executing big data queries to training ML models on large datasets, optimizing hardware parameters for AI pipelines, and tuning software parameters for computationally intensive applications. Each application operates in either a public or private cloud data center. Furthermore, these scenarios differ in terms of domain dimensionality, the number of possible configurations, and the number of configurations evaluated during the optimization process. Table 1 provides a detailed summary of these applications, including the number of initial points and iterations fixed for all algorithms used to optimize them.

Table 1: Experimental applications for validating *d-MALIBOO*.

Application	**Scenario**	**Deployment**	**Dim.**	**Config.**	**Init. points**	**Iterations**
Query26 monodim.	Big data query	Private Cloud	1	115	3	5
Query40 monodim.	Big data query	Private Cloud	1	115	3	5
Query55 monodim.	Big data query	Private Cloud	1	115	3	5
K-means	Training on big data	Private Cloud	1	120	3	5
Recipe Transcriber	AI pipeline inference	Public Cloud, FaaS	5	276	3	10
Query52	Big data query	Public Cloud	2	672	3	30
Query26	Big data query	Public Cloud	2	696	3	30
Stereomatch	Param. tuning	Private Cloud	4	18527	3	60
LiGen	Param. tuning	Private Cloud	8	491520	11	60

For all the following applications, the objective is to identify the configuration x that minimizes the cost of the application while ensuring that the execution time does not exceed a specified maximum threshold, denoted by T_{max}.

- *Monodimensional Queries*: The first scenario involves querying big data on Spark Cloud applications, specifically interactive queries from the TPC-DS industry benchmark[2], which represent SQL-like tasks. These applications were deployed on a dedicated IBM Power8 cluster to ensure consistent benchmarking without resource contention [18]. The

[1] `https://doi.org/10.5281/zenodo.14674182`

[2] `https://www.tpc.org/tpcds/`

Power8 cluster comprises 4 VMs, each equipped with 12 CPU cores and 58 GB of RAM, providing a total of 48 CPU cores for Spark workers. The memory allocated to each Spark executor was set to 4 GB. In this scenario, the objective is to determine the optimal number of cores, $x_1 \in \{6, 8, \ldots, 44\}$. For each application—specifically, *Query26*, *Query40*, and *Query55*—we consider five distinct thresholds for T_{max}, selected to represent significant values in the empirical distribution of processing times.

- *K-means*: The second application involves a widely used clustering algorithm that is the foundation of many ML applications. This unsupervised learning technique is applied across various domains, including anomaly detection, market research, and healthcare [19, 20, 21]. K-means is an iterative algorithm often characterized by significant variability in execution time. For this scenario, the algorithm was applied to Spark dataframes containing 100 features with values uniformly distributed in the range $[0, 1]$ [18]. The dataframes had 20 million rows. This application was deployed on the same IBM Power8 cluster as the previous scenario, with the optimization variable—the number of VMs—taking the same set of values. As in the previous case, we selected five significant values for T_{max}.

- *Recipe Transcriber*: The third application is a computationally intensive AI pipeline that processes input videos to generate audio transcriptions and identify the ingredients of a recipe by detecting them in video frames. This application was developed within the OSCAR [22] framework and is deployed across the Edge-Cloud continuum. It consists of five compute-intensive components deployed on AWS EC2 VMs[3] and two components deployed on AWS Lambda[4], a serverless computing service offering FaaS. The application runs within OSCAR-P [23], an automated tool for testing, deploying, and profiling containerized applications. For each component on VMs, we vary the *Number of cores* from the minimum required for a single node to the maximum value that avoids resource saturation. The set of these five values forms the input vector x for our optimization. Each execution processes a batch

[3] `https://aws.amazon.com/it/ec2`

[4] `https://aws.amazon.com/it/lambda/`

of 100 input videos, each 10 seconds long. In this scenario, we consider four significant values of T_{max} based on the empirical distribution of processing times.

- *Bidimensional Queries*: The fourth application scenario involves two interactive queries from the TPC-DS industry benchmark: *Query26* and *Query52*. These applications are deployed on Microsoft Azure using the HDInsight service[5] [18], a public cloud service where resource contention could introduce variability in the evaluations of both cost and execution time. The input datasets for these experiments range from 250 GB to 1000 GB. The application can use up to 26 VMs from five different types: A3, A4, D12v2, D13v2, and D14v2. In this setup, the optimization variables are the VM type and the number of instances, resulting in a two-dimensional configuration domain $x = (x_1, x_2)$, where $x_1 \in \{3, \ldots, 26\}$ represents the number of VMs, and $x_2 \in \{2, \ldots, 90\}$ represents the memory size (in GB) allocated per VM, with each value of x_2 corresponding to a specific VM configuration. For both applications, we consider five significant values for T_{max}.

- *Stereomatch*: The fifth application is an image-processing edge computing application [24] that computes the disparity value between a pair of stereo images captured from the same scene by two different cameras. This disparity value is used to determine the depth of objects within the scene. The application runs on a private cloud. In this case, we optimize four parameters $x = (x_1, x_2, x_3, x_4)$: the number of parallel cores, color similarity confidence, granularity of disparity hypotheses to be tested, and the length of the support window arm. The search space for this application is significantly larger than that of the previous applications. For this scenario, we select six different values for T_{max}, based on the distribution of execution times.

The final and most complex application differs from the others in both its OF and the constraints imposed.

- *LiGen*: The final scenario involves a molecular docking application integrated into the EXSCALATE drug discovery platform [25]. The

[5] `https://azure.microsoft.com/it-it/products/hdinsight`

LiGen code simulates diverse ligand-pocket interactions by identifying promising docking poses of the ligand within the pocket through multiple restarts of a gradient descent algorithm, followed by a clustering analysis. Subsequently, several representative poses are selected and evaluated using a scoring function [26]. The quality of the docking solution is assessed by calculating the average Root Mean Square Distance (RMSD) across 100 ligand-protein pairs with known optimal crystal positions. In this scenario, a configuration x is defined by eight discrete parameters. Six of these parameters—controlling gradient descent restarts, clustering distance, and the number of poses to evaluate—affect the accuracy, i.e., the RMSD. The other two parameters, the number of CUDA threads per block and the reading buffer size, primarily influence the performance of the application. Fine-tuning these parameters is essential to achieve an optimal balance between performance and accuracy. However, the parameter space include nearly half a million possible configurations, making precise optimization critical. The application outputs are the RMSD $R(x)$ and the execution time $T(x)$. The OF to minimize is $R^3(x)T(x)$, subject to a quality constraint $R(x) \leq R_{max}$. Six different thresholds for R_{max}, representing significant points in the empirical distribution of evaluations, are considered.

5.2. SOTA comparison approaches

In this section, we introduce the optimization algorithms from the literature we compare with besides the original MALIBOO: *SVM-CBO* and *OpenTuner*.

The first reference approach is Support Vector Machine Constrained BO (SVM-CBO) [10]. This algorithm combines ML and BO to identify optimal configurations in a constrained setting. The SVM-CBO algorithm operates in two phases. In Phase 1, the algorithm estimates the feasible region defined by the constraints using M function evaluations. Feasibility is determined by a nonlinear separating hyperplane constructed by an SVM classifier trained on these points. The next evaluation point is selected to refine the feasible region estimate and detect any disconnected feasible regions. In Phase 2, SVM-CBO performs a modified BO process within the feasible region identified in Phase 1, fitting a GP to the OF. The search space is restricted to the identified feasible region, and the GP constructs a probabilistic surrogate model of the OF using only the feasible points observed so far. This approach is applied during the last $N - M$ iterations. For a fair comparison, we adopt the same

proportions between M and N as recommended by the authors in [10]. The SVM-CBO optimization approach is comparable to our *d-MALIBOO*, as it also leverages the strengths of an ML model, specifically an SVM, to estimate the feasible region. Unlike SVM-CBO, *d-MALIBOO* concurrently evaluates both the constraints and the OF, while also considering unfeasible points for training the GP model.

Our second reference solution is OpenTuner [11], a widely used autotuner that employs an ensemble of search techniques to collaboratively explore large and complex search spaces. This approach excels through its adaptive allocation of resources: techniques that identify better configurations receive a larger testing budget, while less effective ones are assigned fewer tests or are disabled. OpenTuner facilitates collaboration among techniques by sharing results through a common database, enhancing the search for optimal configurations. The allocation process is guided by solving the multi-armed bandit problem using the area under the curve credit assignment metric. This widely used approach differs significantly from ours, but we use it as a competitor because it is a popular SOTA method for constrained optimization scenarios, particularly in the HPC community.

5.3. Experimental Setup

We now outline the setup for our experimental campaign and the hyperparameters for *d-MALIBOO*. The optimization algorithms were deployed on a Linux server equipped with a 40-core Intel(R) Xeon(R) CPU operating at 2.20 GHz and 32 GB of memory. The total overhead for *d-MALIBOO* to select configurations, excluding OF evaluations, does not exceed 9 minutes in the worst-case scenario, i.e., *LiGen*. For this specific application, the average evaluation time of the OF is approximately equal to the time required for 60 iterations of *d-MALIBOO*. To ensure the robustness of our experimental analysis, each experiment was repeated $K = 30$ times with different random seeds, leading to different sets of initial points. We then compute the following performance metrics to evaluate the performance of the algorithms:

- *Mean Absolute Percentage Regret* (MAPR): This metric quantifies the relative difference between the best solution found by the algorithm and the true optimum, averaged over K repetitions. It is computed as:

$$MAPR(\hat{\mathbf{f}}, f^*) = \frac{1}{K} \sum_{k=1}^{K} \left| \frac{\hat{f}^k - f^*}{f^*} \right| * 100\%, \tag{11}$$

where $\hat{\mathbf{f}}$ is the vector of the best OF values found in each repetition, and f^* is the ground-truth minimum of the OF. MAPR provides a normalized measure of regret, expressed as a percentage relative to the true optimum.

- *Percentage standard deviation*: This metric represents the variability of the best solutions over the K repetitions, normalized by the true optimum. It is calculated as:

$$stddev = \frac{\sigma}{f^*} * 100\%, \tag{12}$$

where σ is the standard deviation of the optimal values across repetitions. Normalizing by f^* allows the variability to be expressed relative to the scale of the problem, enabling comparisons between problems with different ranges of OF values. Note that, in cases where certain algorithms fail to find any feasible configurations during some repetitions, we include only those repetitions that return at least one feasible solution when computing both MAPR and the standard deviation.

- *Feasibility rate*: This metric measures the robustness of an algorithm. We define it as the percentage of repetitions in which a feasible solution is found, i.e., when at least one feasible point is identified during the optimization process.

As recommended in [9], we use a constant mean function, $\mu(\cdot) = \mu_0$, and a Matérn kernel for both *d-MALIBOO* and MALIBOO. When implementing the ε-greedy approach, we set $\varepsilon = 0.1$. For both the constraint and target function prediction ML models, we employ Ridge regression with a second-degree polynomial feature expansion. This choice ensures fast and accurate predictions. Based on our preliminary analysis, the test-set Mean Absolute Percentage Error for these models is consistently below 10% [9].

5.4. Empirical Results

In this section, we present the empirical results validating the proposed *d-MALIBOO* algorithm. First, we identify the optimal configuration for *d-MALIBOO*, based on the approaches outlined in Section 4 (Section 5.4.1). Next, we compare its performance with that of SVM-CBO and OpenTuner (Section 5.4.2). Finally, we provide a detailed discussion of the obtained results (Section 5.4.3).

5.4.1. Best d-MALIBOO configuration

In the first part of our experimental campaign, we evaluate the 20 possible configurations of the *d-MALIBOO* algorithm to identify the most effective setup in terms of regret and configuration feasibility. Specifically, we aim to determine whether a single factor has the greatest impact on the performance of the algorithm or if a combination of different approaches leads to significant improvements over the original version. For instance, Table 2 presents the results obtained for the *Stereomatch* application with $T_{\max} = 12$s.

Table 2: Results of *d-MALIBOO* on Stereomatch, $T_{max} = 12s$. MALIBOO in bold blue.

ML constr.	**ML target**	**ε-greedy**	**MAPR**	**stddev [%]**	**feas. rate [%]**
indicator	None	True	9.63	11.60	100.00
indicator	**None**	**False**	**13.84**	**21.69**	**100.00**
indicator	indicator	True	6.66	9.47	100.00
indicator	indicator	False	7.17	9.88	100.00
indicator	probability	True	16.37	19.27	100.00
indicator	probability	False	7.27	10.32	100.00
indicator	sum	True	12.12	14.66	100.00
indicator	sum	False	12.44	17.74	100.00
indicator	product	True	10.96	15.37	100.00
indicator	product	False	13.18	13.03	100.00
probability	None	True	9.22	12.31	100.00
probability	None	False	5.69	9.67	100.00
probability	indicator	True	1.89	3.21	100.00
probability	indicator	False	1.86	3.04	100.00
probability	probability	True	5.06	8.32	100.00
probability	probability	False	1.00	1.53	100.00
probability	sum	True	4.62	7.42	100.00
probability	sum	False	1.81	1.59	100.00
probability	product	True	3.31	5.77	100.00
probability	product	False	4.20	4.39	100.00

To identify the best-performing configurations, we compute the following metrics:

- *upper confidence regret* $= \frac{MAPR \, + \, 2 \cdot stddev}{\text{feasibility rate}}$. This metric can be interpreted as an upper confidence bound on the regret, normalized by the percentage of feasible results. Including the standard deviation along with the mean regret helps capture the variability across multiple runs, ensuring that the evaluation reflects both the average performance and the consistency of the results.

- *normalized distance from optimum* $= \frac{\hat{f}-f^*}{F^*-f^*}$. Here, $\hat{f}$ represents the average best value of the OF for a specific setting, f^* denotes the minimum feasible value, and F^* is the maximum feasible value. This metric captures the relative proximity of the results to the optimal solution, normalized by the feasible range.

We gather data from experiments conducted across all applications and constraint bounds, compute the metrics, and average them over the bounds for each application. Next, we calculate the product of the two metrics to rank the configurations. Table 3 shows the five best configurations of *d-MALIBOO* according to the experimental data.

ML constraint	**ML target**	**ε-greedy**	**Label**
indicator	probability	True	Config. 1
indicator	product	True	Config. 2
indicator	product	False	Config. 3
indicator	indicator	True	Config. 4
indicator	sum	True	Config. 5

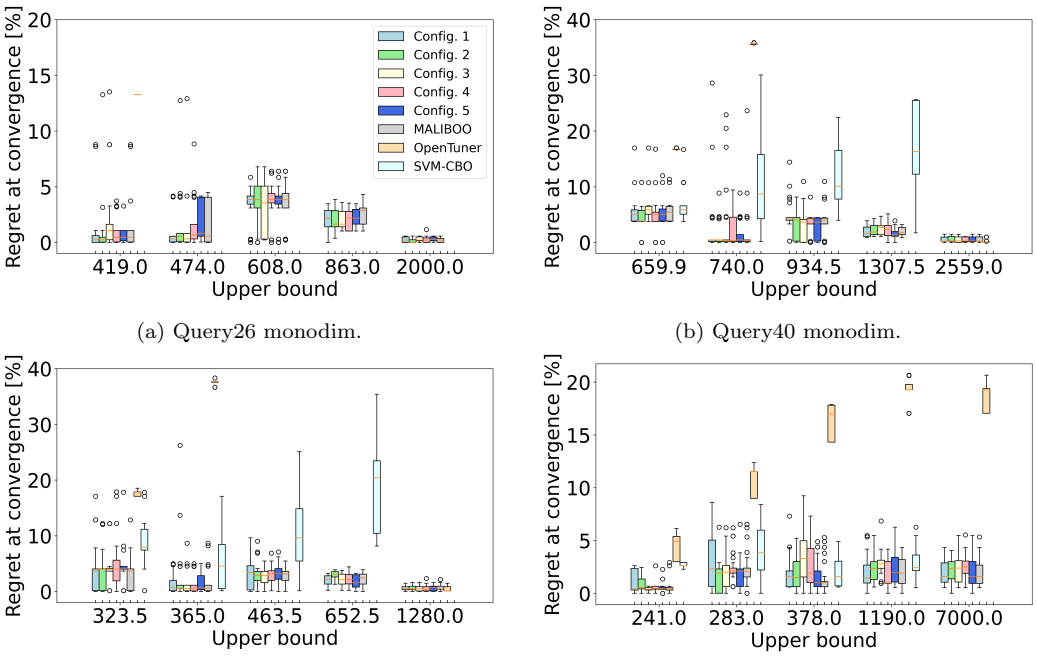
Table 3: Best five *d-MALIBOO* configurations.

The results demonstrate that the features introduced in *d-MALIBOO* perform particularly well when used together. The most effective configuration uses the "indicator" approach as the estimator for the black-box constraint, the "probability" to estimate the OF, and the ε-greedy strategy to encourage exploration. Furthermore, note that almost all the top-performing configurations use "indicator" for constraints and the ε-greedy approach. This represents a balanced approach combining the exploitation strength of the former with the exploration benefits of the latter.

5.4.2. Comparison with the state of the art

As a final analysis, we compare the best configurations of our *d-MALIBOO* algorithm with MALIBOO, SVM-CBO and OpenTuner.

First, we analyze the results of the mono-dimensional applications presented in Figure 1, namely *Query26*, *Query40*, *Query55*, and *K-means*. The box plots show the distribution of regret across the repetitions, with the orange line indicating the median. If the box corresponding to a specific configuration is not visible, it means either that the MAPR is significantly higher than the others, as in Figure 1a, or that the feasibility rate is 0%. Detailed data for these results are provided in Tables 4, 5, 6, and 7.



(a) Query26 monodim.

(b) Query40 monodim.

(c) Query55 monodim.

(d) K-means

Figure 1

Table 4: Results of the five best configurations of *d-MALIBOO*, MALIBOO, OpenTuner and SVM-CBO on the *Query26 monodimensional* application. We highlight the best result for each constraint in **bold**.

	$T(x) \leq 419s$		$T(x) \leq 474s$		$T(x) \leq 608s$		$T(x) \leq 863s$		$T(x) \leq 2000s$	
Algorithm	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**
Config. 1	1.80	89.29	**0.58**	**100.00**	3.31	100.00	2.11	100.00	0.27	100.00
Config. 2	1.50	85.71	2.19	96.00	3.47	100.00	2.01	100.00	**0.15**	**100.00**
Config. 3	2.88	89.29	0.95	100.00	**3.28**	**100.00**	2.04	100.00	0.29	100.00
Config. 4	**1.25**	**85.71**	2.30	100.00	3.73	100.00	**1.95**	**100.00**	0.29	100.00
Config. 5	1.37	71.43	1.72	100.00	3.43	100.00	2.29	100.00	0.30	100.00
MALIBOO	2.22	85.71	1.43	100.00	3.47	100.00	2.64	100.00	0.20	100.00
OpenTuner	13.26	100.00	36.61	100.00	63.47	100.00	72.89	100.00	75.50	100.00
SVM-CBO	inf	0.00	inf	0.00	inf	0.00	inf	0.00	inf	0.00

Since the optimization domain of these applications is limited, we set a small number of iterations for the different algorithms. Consequently, the ML models are less accurate than in other experiments. Despite this limitation, *d-MALIBOO* demonstrates slightly better performance compared to MALIBOO in almost all scenarios. In contrast, the two competitor approaches perform consistently worse. Specifically, OpenTuner yields significantly higher regrets, ranging from 10% to 70% more than *d-MALIBOO*.

Table 5: Results of the five best configurations of *d-MALIBOO*, MALIBOO, OpenTuner and SVM-CBO on the *Query40 monodimensional* application. We highlight the best result for each constraint in **bold**.

	$T(x) \leq 659.9s$		$T(x) \leq 740.0s$		$T(x) \leq 934.5s$		$T(x) \leq 1307.5s$		$T(x) \leq 2559.0s$	
Algorithm	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**
Config. 1	6.25	85.71	2.48	100.00	4.58	100.00	1.90	100.00	0.46	100.00
Config. 2	**4.70**	**82.14**	**1.34**	**100.00**	3.66	100.00	2.32	100.00	0.61	100.00
Config. 3	6.97	64.29	3.02	100.00	2.97	100.00	2.45	100.00	0.52	100.00
Config. 4	5.40	89.29	2.07	100.00	3.40	100.00	2.31	100.00	0.54	100.00
Config. 5	5.70	75.00	1.81	89.29	**2.76**	**100.00**	**1.72**	**100.00**	0.53	100.00
MALIBOO	5.97	78.57	2.85	100.00	4.08	100.00	1.97	100.00	0.61	100.00
OpenTuner	16.80	100.00	35.63	100.00	65.53	100.00	78.58	100.00	**0.17**	**100.00**
SVM-CBO	6.98	75.00	10.04	100.00	12.13	96.43	18.31	50.00	inf	0.00

Table 6: Results of the five best configurations of *d-MALIBOO*, MALIBOO, OpenTuner and SVM-CBO on the *Query55 monodimensional* application. We highlight the best result for each constraint in **bold**.

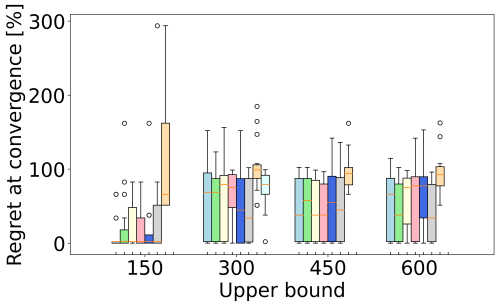
	$T(x) \leq 323.5s$		$T(x) \leq 365.0s$		$T(x) \leq 463.5s$		$T(x) \leq 652.5s$		$T(x) \leq 1280.0s$	
Algorithm	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**
Config. 1	4.18	100.00	1.53	100.00	3.54	100.00	1.97	100.00	0.54	100.00
Config. 2	3.94	96.43	2.40	100.00	2.85	100.00	2.98	100.00	0.61	100.00
Config. 3	5.17	89.29	**1.19**	**100.00**	**2.37**	**100.00**	2.21	100.00	0.51	100.00
Config. 4	5.29	100.00	1.28	100.00	2.67	100.00	2.27	100.00	0.47	100.00
Config. 5	5.02	89.29	1.82	100.00	2.90	100.00	**1.70**	**100.00**	0.64	100.00
MALIBOO	**3.77**	**92.86**	1.51	100.00	2.55	100.00	2.17	100.00	0.68	100.00
OpenTuner	17.64	100.00	37.49	100.00	64.60	100.00	77.21	100.00	**0.40**	**100.00**
SVM-CBO	9.82	39.29	5.50	100.00	11.73	92.86	20.30	46.43	inf	0.00

Table 7: Results of the five best configurations of *d-MALIBOO*, MALIBOO, OpenTuner and SVM-CBO on the *K-means* application. We highlight the best result for each constraint in **bold**.

	$T(x) \leq 241s$		$T(x) \leq 283s$		$T(x) \leq 378s$		$T(x) \leq 1190s$		$T(x) \leq 7000s$	
Algorithm	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**
Config. 1	1.29	71.43	3.02	96.43	1.92	100.00	2.10	100.00	**1.91**	**100.00**
Config. 2	1.03	78.57	**1.80**	**92.86**	1.86	100.00	2.34	100.00	2.00	100.00
Config. 3	**0.66**	**67.86**	2.47	89.29	3.56	100.00	2.83	100.00	2.37	100.00
Config. 4	0.99	53.57	2.58	92.86	2.67	100.00	**1.73**	**100.00**	2.47	100.00
Config. 5	0.70	60.71	2.12	78.57	1.59	100.00	2.38	100.00	1.96	100.00
MALIBOO	0.71	75.00	2.59	92.86	**1.43**	**100.00**	2.12	100.00	**1.91**	**100.00**
OpenTuner	4.65	100.00	10.84	100.00	16.49	100.00	19.18	100.00	18.46	100.00
SVM-CBO	3.04	21.43	4.48	85.71	2.22	100.00	3.14	71.43	inf	0.00

SVM-CBO struggles with the strict thresholds imposed by the black-box constraints, particularly with the *Query26 monodimensional* application. In fact, SVM-CBO did not find any feasible solution in any run.

Recipe Transcriber is a multi-dimensional application. Since the number of configurations in the domain is limited, we perform relatively few algorithm iterations in this case as well. We present the box plot for this application in Figure 2, with the detailed data provided in Table 8.



(a) Recipe Transcriber

Figure 2

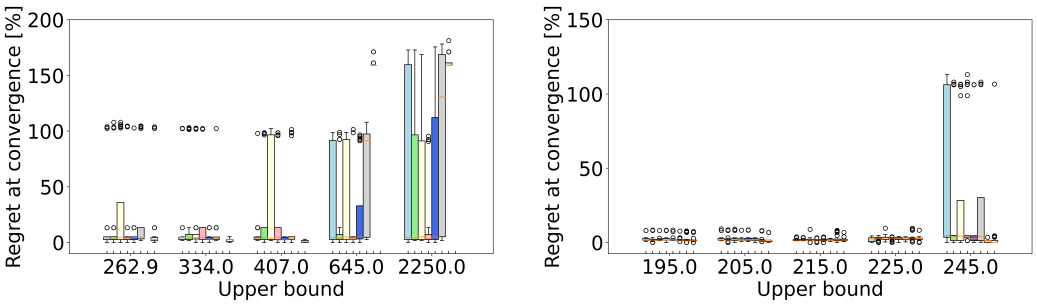
Table 8: Results of the five best configurations of *d-MALIBOO*, MALIBOO, OpenTuner and SVM-CBO on the *Recipe Transcriber* application. We highlight the best result for each constraint in **bold**.

	$T(x) \leq 150s$		$T(x) \leq 300s$		$T(x) \leq 450s$		$T(x) \leq 600s$	
Algorithm	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**
Config. 1	**14.07**	**67.86**	56.27	100.00	46.80	100.00	48.95	100.00
Config. 2	32.46	57.14	47.87	100.00	49.86	100.00	45.52	100.00
Config. 3	30.88	67.86	60.56	100.00	45.09	100.00	55.64	100.00
Config. 4	27.41	64.29	66.27	100.00	**43.42**	**100.00**	60.21	100.00
Config. 5	31.42	46.43	49.01	100.00	53.31	100.00	60.58	100.00
MALIBOO	46.95	75.00	**43.66**	**100.00**	52.76	100.00	**39.30**	**100.00**
OpenTuner	136.04	35.71	99.60	100.00	93.06	100.00	93.17	100.00
SVM-CBO	inf	0.00	74.17	82.14	95.15	3.57	inf	0.00

The results indicate that *d-MALIBOO* and MALIBOO achieve comparable performance, consistent with previous scenarios. As before, OpenTuner exhibits very high regrets, while SVM-CBO again struggles to find feasible solutions.

Next, we discuss the bidimensional queries, namely *Query52* and *Query26*. In this scenario, the configuration space is larger than the previous applications. Figure 3 and Tables 9 and 10 summarize the results.

The results show that *d-MALIBOO* outperforms all competitor approaches. Notably, OpenTuner struggles significantly with *Query52*, exhibiting a MAPR that exceeded 160%, while *d-MALIBOO* consistently achieves values below 23%. OpenTuner performs more competitively on *Query26*, particularly under the least restrictive threshold. SVM-CBO struggles to define the feasibility region, particularly with *Query52*. Finally, while MALIBOO performs comparably to *d-MALIBOO* on *Query26* for most constraints, it struggles



(a) Query52 (b) Query26

Figure 3

Table 9: Results of the five best configurations of *d-MALIBOO*, MALIBOO, OpenTuner and SVM-CBO on the *Query52* application. We highlight the best result for each constraint in **bold**.

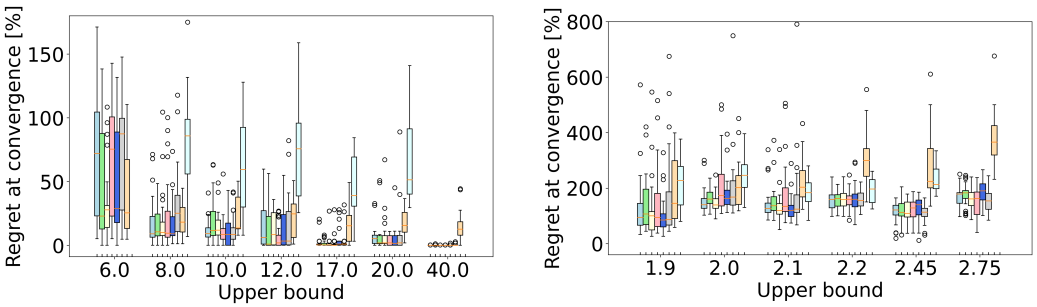
	$T(x) \leq 262.9s$		$T(x) \leq 334.0s$		$T(x) \leq 407.0s$		$T(x) \leq 645.0s$		$T(x) \leq 2250.0s$	
Algorithm	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**
Config. 1	14.74	100.00	18.02	100.00	8.06	100.00	29.76	100.00	56.16	100.00
Config. 2	14.72	100.00	21.73	100.00	21.36	100.00	17.26	100.00	46.38	100.00
Config. 3	29.00	100.00	22.10	100.00	31.01	100.00	36.74	100.00	38.64	100.00
Config. 4	14.30	100.00	21.89	100.00	20.87	100.00	**10.26**	**100.00**	**22.35**	**100.00**
Config. 5	**8.33**	**100.00**	**3.73**	**100.00**	**3.11**	**100.00**	26.61	100.00	58.41	100.00
MALIBOO	22.99	100.00	8.31	100.00	23.32	100.00	60.16	100.00	106.06	100.00
OpenTuner	569.00	100.00	564.71	100.00	567.28	100.00	160.15	100.00	162.05	100.00
SVM-CBO	17.83	96.43	8.48	57.14	1.36	25.00	inf	0.00	inf	0.00

Table 10: Results of the five best configurations of *d-MALIBOO*, MALIBOO, OpenTuner and SVM-CBO on the *Query26* application. We highlight the best result for each constraint in **bold**.

	$T(x) \leq 195s$		$T(x) \leq 205s$		$T(x) \leq 215s$		$T(x) \leq 225s$		$T(x) \leq 245s$	
Algorithm	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**
Config. 1	2.31	100.00	2.34	100.00	1.80	100.00	**2.32**	**100.00**	33.18	100.00
Config. 2	2.08	100.00	2.11	100.00	1.85	100.00	2.83	100.00	21.40	100.00
Config. 3	1.96	100.00	2.53	100.00	1.91	100.00	2.90	100.00	28.51	100.00
Config. 4	2.65	100.00	2.04	100.00	1.79	100.00	2.67	100.00	25.77	100.00
Config. 5	2.32	100.00	2.39	100.00	**1.65**	**100.00**	2.52	100.00	13.42	100.00
MALIBOO	**1.89**	**100.00**	**1.96**	**100.00**	1.75	100.00	2.44	100.00	42.47	100.00
OpenTuner	2.31	100.00	2.08	96.43	2.51	100.00	3.64	100.00	**0.97**	**100.00**
SVM-CBO	2.46	67.86	2.35	64.29	2.05	82.14	2.90	89.29	9.18	96.43

with the least restrictive one. Moreover, on *Query52*, the MAPR of MALIBOO is two to five times higher than that of *d-MALIBOO*.

The final two applications, *Stereomatch* and *LiGen*, have the largest configuration spaces. Consequently, we allocate a greater number of iterations for the optimization algorithms. We show their box plots in Figure 4, with detailed data provided in Tables 11 and 12.



(a) Stereomatch

(b) LiGen

Figure 4

Table 11: Results of the five best configurations of *d-MALIBOO*, MALIBOO, OpenTuner and SVM-CBO on the *Stereomatch* application. We highlight the best result for each constraint in **bold**.

	$T(x) \leq 6s$		$T(x) \leq 8s$		$T(x) \leq 10s$		$T(x) \leq 12s$		$T(x) \leq 17s$		$T(x) \leq 20s$		$T(x) \leq 40$	
Algorithm	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**
Config. 1	78.86	100.00	17.48	100.00	**10.57**	**100.00**	16.37	100.00	1.96	100.00	6.55	100.00	0.03	100.00
Config. 2	50.99	100.00	**16.22**	**100.00**	18.37	100.00	10.96	100.00	**1.51**	**100.00**	6.48	100.00	0.07	100.00
Config. 3	37.46	92.86	17.43	100.00	14.52	100.00	13.18	100.00	2.50	100.00	7.52	100.00	**0.01**	**100.00**
Config. 4	68.83	96.43	23.63	100.00	10.67	100.00	**6.66**	**100.00**	1.84	100.00	**3.52**	**100.00**	0.04	100.00
Config. 5	53.49	96.43	17.82	100.00	11.95	100.00	12.12	100.00	4.06	100.00	4.41	100.00	0.11	100.00
MALIBOO	72.95	100.00	30.85	100.00	11.71	100.00	13.84	100.00	2.94	100.00	7.88	100.00	0.39	100.00
OpenTuner	**39.15**	**100.00**	20.89	100.00	22.98	100.00	23.40	100.00	17.22	100.00	17.50	100.00	14.42	100.00
SVM-CBO	inf	0.00	91.28	71.43	63.03	100.00	80.79	89.29	49.56	46.43	77.86	35.71	inf	0.00

Table 12: Results of the five best configurations of *d-MALIBOO*, MALIBOO, OpenTuner and SVM-CBO on the *LiGen* application. We highlight the best result for each constraint in **bold**.

	$R(x) \leq 1.9$		$R(x) \leq 2.0$		$R(x) \leq 2.1$		$R(x) \leq 2.2$		$R(x) \leq 2.45$		$R(x) \leq 2.75$	
Algorithm	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**	**MAPR**	**feas. [%]**
Config. 1	165.30	100.00	153.03	100.00	143.07	100.00	**154.40**	**100.00**	114.11	100.00	166.78	100.00
Config. 2	163.12	89.29	166.56	100.00	151.42	100.00	155.90	100.00	114.06	100.00	178.21	100.00
Config. 3	141.02	89.29	**146.86**	**100.00**	128.78	100.00	159.53	100.00	113.16	100.00	163.77	100.00
Config. 4	141.13	100.00	234.95	100.00	174.66	100.00	157.80	100.00	122.46	100.00	**154.39**	**100.00**
Config. 5	**119.68**	**96.43**	178.37	100.00	**127.89**	**100.00**	161.70	100.00	123.55	100.00	188.73	100.00
MALIBOO	161.07	100.00	200.92	100.00	170.67	100.00	158.68	100.00	**107.60**	**100.00**	159.08	100.00
OpenTuner	201.30	96.43	210.86	100.00	236.45	100.00	305.00	100.00	275.97	100.00	376.19	100.00
SVM-CBO	315.84	32.14	261.43	82.14	184.19	100.00	192.59	100.00	252.13	46.43	inf	0.00

In both applications, *d-MALIBOO* consistently outperforms all other approaches. SVM-CBO demonstrates the poorest performance in both scenarios, primarily due to its low feasibility rate, especially at threshold values at the extremes of the distributions. OpenTuner performs best in a single specific scenario in *Stereomatch*, namely with $T(x) \leq 6s$. However, in all other scenarios, it is outperformed by *d-MALIBOO*, with a MAPR that is 75%–170% higher than the best configuration of *d-MALIBOO*. Similarly, MALIBOO shows approximately twice the MAPR of *d-MALIBOO* on *Stere-*

omatch. Furthermore, *d-MALIBOO* achieves an average MAPR that is 25% lower than MALIBOO on *LiGen*.

5.4.3. Discussion

The results of this experimental campaign indicate that employing ML models to guide the AF is highly effective in discrete domains, particularly in scenarios requiring a greater number of iterations, such as those with very large optimization domains. Furthermore, we observe that using the "indicator" approach to define the feasible region, as proposed in [9], becomes even more effective when combined with techniques that balance the exploration-exploitation trade-off, such as the ε-greedy approach. Notably, this approach proves most beneficial in domains with multiple local optima, as seen in *LiGen*, as it helps prevent stagnation in the optimization process. These enhancements to the original MALIBOO algorithms yield an average MAPR reduction of approximately 9% for the proposed approach, with notable improvements of 11.2% on *Recipe Transcriber*, 17.5% on *LiGen*, and 40.4% on *Query52*.

Additionally, we observe that *d-MALIBOO* outperforms the two state-of-the-art methods in nearly every application and across all values of the threshold on the black-box constraint. Notably, OpenTuner requires more iterations to achieve MAPRs comparable to our algorithm but is almost always successful in finding feasible configurations. In contrast, SVM-CBO often struggles to define the feasibility region, particularly with thresholds that split the domain in an unbalanced way. If all the points explored in phase 1 are either entirely feasible or entirely unfeasible, the algorithm halts due to the absence of an estimated boundary. As a result, OpenTuner achieves an average MAPR that is 29.3% higher than *d-MALIBOO*, while SVM-CBO exhibits an even greater difference, with an MAPR 82.4% higher than our proposed algorithm.

6. Related Work

BO algorithms have gained significant popularity in recent years as efficient methods for hyperparameter tuning and optimization. Although originally designed for continuous and unconstrained optimization, recent research has addressed the challenges of discrete variable optimization, extending the applicability of BO to a wide range of fields. This section focuses on the techniques and applications proposed in the literature.

A key challenge in BO with discrete variables is modeling the OF effectively. Naive approaches, such as rounding for integer variables and one-hot encoding for categorical variables, often fail to capture the true structure of the OF. These techniques assume that the OF varies continuously, even though it remains constant within regions corresponding to the same integer values. As a result, they produce a GP model that fails to capture the piecewise-constant nature of the relaxed OF, leading to an inaccurate representation. To address this issue, [16] proposes a variable transformation that adjusts the GP covariance function, enabling a more accurate representation of constant regions and ultimately improving optimization performance. However, this approach is not applicable in our scenarios, where the feature values are not equispaced. Another challenge in discrete optimization is avoiding the repetition of previously observed points. To tackle this issue, [27] propose Discrete-BO, a BO algorithm that reparametrizes the parameters of the UCB AF and the RBF kernel of the GP, based on whether the suggested point has already been visited. Optimization of combinatorial domains is closely related to discrete optimization. [28] introduces Bayesian Optimization for Combinatorial Structures, employing convex optimization techniques to efficiently approximate the maximum of the AF. This approach also integrates an interpretable model that captures interactions among structural features, enhancing its effectiveness in combinatorial optimization.

In the context of constrained optimization, [29] introduces a probabilistic algorithm designed to handle noisy constraints, enabling independent evaluations of the objective and constraint functions. Expanding on this concept, [30] presents Predictive Entropy Search with Constraints, which breaks down the information gain into contributions from each individual function, thus improving optimization efficiency.

The natural extension of frameworks designed for both continuous and discrete domains is mixed-variable optimization. [31] presents MIVABO, a BO algorithm tailored for optimizing mixed-variable functions under known linear and quadratic integer constraints. MIVABO employs a linear surrogate model to separate the continuous, discrete, and mixed components of the OF. This approach allows the algorithm to efficiently sample from the posterior distribution using Thompson sampling, thereby optimizing the constrained AF. [32] introduces CoCaBO, which integrates multi-armed bandits and BO to manage both continuous and categorical inputs. It uses a kernel specifically designed for mixed-type spaces, enabling the capture of interactions between continuous and categorical inputs without requiring one-hot

encoding. Lastly, [33] presents the Probabilistic Reparameterization (PR) approach, which maximizes the expectation of the AF over a probability distribution defined on continuous parameters, rather than directly optimizing the AF over discrete parameters. This is achieved by appropriately reparameterizing discrete variables into continuous ones.

Finally, we present several BO-based techniques and applications. Hyperopt [34] is an open-source framework designed for hyperparameter optimization with applications in deep learning. A key contribution of their work is the Tree-structured Parzen Estimator (TPE), which uses probabilistic modeling to differentiate between good and bad configurations, thus guiding optimization toward promising regions of the search space. [35] introduces a BO-based framework for intrusion detection, leveraging optimized ML classifiers such as SVMs with RBF kernels, Random Forests, and k-Nearest Neighbors. Their work highlights the effectiveness of BO in fine-tuning ML models for complex tasks, such as those encountered in cybersecurity. [14] applies BO to automate the tuning of benchmark parameters for supercomputers. Their method optimized energy efficiency, helping their system, "Kukai", secure second place in the Green500 rankings. [36] presents HiPerBOt, an active learning framework that leverages BO to identify high-performing configurations, significantly reducing the cost of evaluating expensive application runs. [37] introduces a BO-based approach for task offloading and resource allocation in dynamic, multi-server, multi-user mobile edge-computing systems. Their method incorporates an innovative kernel design that integrates temporal and contextual information, enhancing decision-making under bandit feedback.

7. Conclusions and Future Work

This work presents *d-MALIBOO*, a collection of algorithms that integrates BO and ML to address the optimization of constrained problems in discrete and multidimensional domains. Specifically, we have incorporated novel ML approaches for identifying the feasible region within black-box constraints and for enhancing the search for the optimum in discrete domains. Additionally, we have enhanced the exploration phase by introducing an ε-greedy approach within the BO algorithm to prevent stagnation in the improvement of the MAPR. The framework supports any combination of ML models for the bounds, target, and the ε-greedy approach. We then tested *d-MALIBOO* to first determine the optimal configuration of the algorithm

and later compared its performance with SVM-CBO, OpenTuner, and the original MALIBOO. The results demonstrate that d-MALIBOO reduces the MAPR by 9% compared to MALIBOO, 29% compared to OpenTuner, and 82% compared to SVM-CBO.

Future work will focus on developing novel strategies to avoid repeating already observed points, different from simply vanishing the AF at those points. Additionally, we will extend our framework to optimize general mixed-variable domains.

References

[1] Z. R. Alashhab, M. Anbar, M. M. Singh, Y.-B. Leau, Z. A. Al-Sai, S. A. Alhayja'a, Impact of coronavirus pandemic crisis on technologies and cloud computing applications, Journal of Electronic Science and Technology 19 (1) (2021) 100059.

[2] G. Habib, S. Sharma, S. Ibrahim, I. Ahmad, S. Qureshi, M. Ishfaq, Blockchain technology: benefits, challenges, applications, and integration of blockchain technology with cloud computing, Future Internet 14 (11) (2022) 341.

[3] S. A. Bello, L. O. Oyedele, O. O. Akinade, M. Bilal, J. M. D. Delgado, L. A. Akanbi, A. O. Ajayi, H. A. Owolabi, Cloud computing in construction industry: Use cases, benefits and challenges, Automation in Construction 122 (2021) 103441.

[4] S. Vinoth, H. L. Vemula, B. Haralayya, P. Mamgain, M. F. Hasan, M. Naved, Application of cloud computing in banking and e-commerce and related security threats, Materials Today: Proceedings 51 (2022) 2172–2175.

[5] H. K. Mistry, C. Mavani, A. Goswami, R. Patel, The impact of cloud computing and ai on industry dynamics and competition, Educational Administration: Theory and Practice 30 (7) (2024) 797–804.

[6] H. Li, S. X. Wang, F. Shang, K. Niu, R. Song, Applications of large language models in cloud computing: An empirical study using real-world data, International Journal of Innovative Research in Computer Science & Technology 12 (4) (2024) 59–69.

[7] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, M. Zhang, {CherryPick}: Adaptively unearthing the best cloud configurations for big data analytics, in: 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), 2017, pp. 469–482.

[8] P. I. Frazier, A tutorial on Bayesian optimization, arXiv preprint arXiv:1807.02811 (2018).

[9] B. Guindani, D. Ardagna, A. Guglielmi, R. Rocco, G. Palermo, Integrating Bayesian optimization and machine learning for the optimal configuration of cloud systems, IEEE Transactions on Cloud Computing 12 (1) (2024) 277–294.

[10] A. Candelieri, Sequential model based optimization of partially defined functions under unknown constraints, Journal of Global Optimization 79 (2) (2021) 281–303.

[11] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, S. Amarasinghe, Opentuner: An extensible framework for program autotuning, in: IEEE PACT, 2014, pp. 303–316.

[12] R. Sala, B. Guindani, D. Ardagna, A. Guglielmi, d-MALIBOO: a Bayesian Optimization framework for dealing with Discrete Variables, in: 2024 32nd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), IEEE, 2024, pp. 1–8.

[13] J. Wu, X.-Y. Chen, H. Zhang, L.-D. Xiong, H. Lei, S.-H. Deng, Hyperparameter optimization for machine learning models based on bayesian optimization, Journal of Electronic Science and Technology 17 (1) (2019) 26–40.

[14] T. Miyazaki, I. Sato, N. Shimizu, Bayesian optimization of hpc systems for energy efficiency, in: High Performance Computing: 33rd International Conference, ISC High Performance 2018, Frankfurt, Germany, June 24-28, 2018, Proceedings 33, Springer, 2018, pp. 44–62.

[15] C. E. Rasmussen, C. K. Williams, Gaussian processes for machine learning, Vol. 1, Springer, 2006.

[16] E. C. Garrido-Merchán, D. Hernández-Lobato, Dealing with categorical and integer-valued variables in Bayesian optimization with Gaussian processes, Neurocomputing 380 (2020) 20–35.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al., Human-level control through deep reinforcement learning, nature 518 (7540) (2015) 529–533.

[18] A. Maros, F. Murai, A. P. C. da Silva, J. M. Almeida, M. Lattuada, E. Gianniti, M. Hosseini, D. Ardagna, Machine learning for performance prediction of spark cloud applications, in: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), IEEE, 2019, pp. 99–106.

[19] G. Münz, S. Li, G. Carle, Traffic anomaly detection using k-means clustering, in: Gi/itg workshop mmbnet, Vol. 7, 2007.

[20] R. Kuo, L. Ho, C. M. Hu, Integration of self-organizing feature map and k-means algorithm for market segmentation, Computers & Operations Research 29 (11) (2002) 1475–1493.

[21] R. A. Haraty, M. Dimishkieh, M. Masud, An enhanced k-means clustering algorithm for pattern discovery in healthcare data, International Journal of distributed sensor networks 11 (6) (2015) 615740.

[22] S. Risco, G. Moltó, D. M. Naranjo, I. Blanquer, Serverless workflows for containerised applications in the cloud continuum, Journal of Grid Computing 19 (2021) 1–18.

[23] R. Sala, B. Guindani, E. Galimberti, F. Filippini, H. Sedghani, D. Ardagna, S. Risco, G. Moltó, M. Caballer, OSCAR-P and aML-Library: Profiling and Predicting the Performance of FaaS-based Applications in Computing Continua, Journal of Systems and Software 221 (2025) 112282. doi:10.1016/j.jss.2024.112282.

[24] E. Paone, G. Palermo, V. Zaccaria, C. Silvano, D. Melpignano, G. Haugou, T. Lepley, An exploration methodology for a customizable opencl stereo-matching application targeted to an industrial multi-cluster architecture, in: CODES+ISSS, 2012.

[25] D. Gadioli, E. Vitali, F. Ficarelli, C. Latini, C. Manelfi, C. Talarico, C. Silvano, C. Cavazzoni, G. Palermo, A. R. Beccari, Exscalate: an extreme-scale virtual screening platform for drug discovery targeting polypharmacology to fight sars-cov-2, IEEE Transactions on Emerging Topics in Computing 11 (1) (2022) 170–181.

[26] E. Vitali, D. Gadioli, G. Palermo, A. Beccari, C. Cavazzoni, C. Silvano, Exploiting openmp and openacc to accelerate a geometric approach to molecular docking in heterogeneous hpc nodes, The Journal of Supercomputing 75 (2019) 3374–3396.

[27] P. Luong, S. Gupta, D. Nguyen, S. Rana, S. Venkatesh, Bayesian optimization with discrete variables, in: AJCAI, Springer, 2019, pp. 473–484.

[28] R. Baptista, M. Poloczek, Bayesian optimization of combinatorial structures, in: International conference on machine learning, PMLR, 2018, pp. 462–471.

[29] M. A. Gelbart, J. Snoek, R. P. Adams, Bayesian optimization with unknown constraints, arXiv preprint arXiv:1403.5607 (2014).

[30] J. M. Hern, M. A. Gelbart, R. P. Adams, M. W. Hoffman, Z. Ghahramani, et al., A general framework for constrained bayesian optimization using information-based search, Journal of Machine Learning Research 17 (160) (2016) 1–53.

[31] E. Daxberger, A. Makarova, M. Turchetta, A. Krause, Mixed-variable Bayesian optimization, arXiv preprint arXiv:1907.01329 (2019).

[32] B. Ru, A. Alvi, V. Nguyen, M. A. Osborne, S. Roberts, Bayesian optimisation over multiple continuous and categorical inputs, in: ICML, 2020.

[33] S. Daulton, X. Wan, D. Eriksson, M. Balandat, M. Osborne, E. Bakshy, Bayesian optimization over discrete and mixed spaces via probabilistic reparameterization, NeurIPS (2022).

[34] J. Bergstra, D. Yamins, D. D. Cox, et al., Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms., SciPy 13 (2013) 20.

[35] M. Injadat, F. Salo, A. B. Nassif, A. Essex, A. Shami, Bayesian optimization with machine learning algorithms towards anomaly detection, in: 2018 IEEE global communications conference (GLOBECOM), IEEE, 2018, pp. 1–6.

[36] H. Menon, A. Bhatele, T. Gamblin, Auto-tuning parameter choices in hpc applications using bayesian optimization, in: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2020, pp. 831–840.

[37] J. Yan, Q. Lu, G. B. Giannakis, Bayesian optimization for online management in dynamic mobile edge computing, IEEE Transactions on Wireless Communications (2023).