ROBOTune: High-Dimensional Configuration Tuning for Cluster-Based Data Analytics

Md Muhib Khan
Florida State University
Tallahassee, Florida, USA
khan@cs.fsu.edu

Weikuan Yu
Florida State University
Tallahassee, Florida, USA
yuw@cs.fsu.edu

ABSTRACT

Spark is popular for its ability to enable high-performance data analytics applications on diverse systems. Its great versatility is achieved through numerous user- and system-level options, resulting in an exponential configuration space that, ironically, hinders data analytics's optimal performance. The colossal complexity is caused by two main issues: the high dimensionality of configuration space and the expensive black-box configuration-performance relationship. In this paper, we design and develop a robust tuning framework called ROBOTune that can tackle both issues and tune Spark applications quickly for efficient data analytics. Specifically, it performs parameter selection through a Random Forests based model to reduce the dimensionality of analytics configuration space. In addition, ROBOTune employs Bayesian Optimization to overcome the complex nature of the configuration-performance relationship and balance exploration and exploitation to efficiently locate a globally optimal or near-optimal configuration. Furthermore, ROBOTune strengthens Latin Hypercube Sampling with caching and memoization to enhance the coverage and effectiveness in the generation of sample configurations. Our evaluation results demonstrate that ROBOTune finds similar or better performing configurations than contemporary tuning tools like BestConfig and Gunther while improving on search cost by 1.59× and 1.53× on average and up to 2.27× and 1.71×, respectively.

KEYWORDS

Performance Tuning; Bayesian Optimization; Spark Configurations

ACM Reference Format:
Md Muhib Khan and Weikuan Yu. 2021. ROBOTune: High-Dimensional Configuration Tuning for Cluster-Based Data Analytics. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3472456.3472518

1 INTRODUCTION

Spark [48] has been a popular framework leveraged by many organizations for interactive and scalable data analytics on system and web logs, social media, graph processing [17, 33], etc. For enabling the execution of various types of applications, the number of parameters in Spark has grown from 20 initially to around 200 in version 2.4 [42, 43]. These parameters form a high-dimensional configuration space that controls different execution behaviours, including shuffling, compression, memory, networking and scheduling, etc. Unfortunately, this leads to a problem space of exponential complexity for users to pinpoint the optimal configuration.

In addition, the performance of analytics applications typically exhibits a complex black-box relationship [2, 9, 16, 18] with respect to the choices of configuration parameters. There is not a closed form function that can directly capture the black-box configuration-performance relationship, which is also expensive to evaluate [2, 18]. Choosing some settings based on rules-of-thumb recommendations or trial-and-error can lead to perplexing performance scenarios for non-expert users. Furthermore, scalable analytics systems are usually a shared resource in which application performance can vary due to contentions or noise on the network and storage systems. Even experts would rather avoid time-consuming explorations to determine the best configurations for an application.

Together, the high dimensionality of configurations and the expensive black-box configuration-performance relationship present an immense challenge for analytics applications to take advantage of available processors, memory, storage and network resources from the underlying cluster systems.

There have been two main strategies, *learning* and *searching*, to tune the configuration of cluster-based analytics applications. Learning-based approaches collect a large number of experimental runs (e.g., at least 2000) for modeling the behavior of each application for a specific cluster [5, 46]. It is not feasible to collect so many samples for tuning every application in most real-life situations [49]. In addition, the models created by learning-based approaches are typically specific to the applications and/or the underlying clusters, and have to be retrained for new clusters. Instead, searching-based approaches [4, 25, 49] try to overcome these limitations by using intelligent sampling and searching algorithms like recursive bound and search [49] without training any performance models. These can be much more robust to change in the applications or clusters as they do not rely on creating a model from the data. However, when the dimensionality of the configuration space is too high, these approaches may fail to improve over random search [4]. There have been many other studies on tuning the configuration of different systems, including pattern search, which can suffer from slow local (asymptotic) convergence rates [44], and genetic algorithm [22, 26], which can be robust against local optima [22]. Gunther [25] uses genetic algorithm to find near-optimal configurations of Hadoop workloads, but requires a large number of initial executions before making aggressive selections and mutations.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPP '21, August 9–12, 2021, Lemont, IL, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9068-2/21/08…$15.00
https://doi.org/10.1145/3472456.3472518

Bayesian Optimization (BO) has been exploited successfully to tune the performance of storage [9], database [45] and cloud resources [2]. It is exceptionally potent when the objective function (e.g., execution time) does not have a known closed-form expression and is costly to evaluate. These features render BO quite suitable to address the blackbox non-parametric challenge of tuning cluster-based analytics applications, provided that we can equip it with a solution for the high dimensionality of the configuration space.

To this end, we design and deveop a robust tuning framework called ROBOTune (**R**andom F**O**rests and **B**ayesian **O**ptimization based **Tune**) that can tackle both issues and tune cluster-based data analytics applications quickly for efficient data analytics. ROBOTune is equipped with three main techniques. First, it trains a Random Forests based model for parameter selection and reduces the dimensionality of the configuration space. Next, it features a Bayesian Optimization engine that overcomes the complex configuration-performance relationship and incrementally searches for an optimal configuration based on the observations of prior configuration samples. Particularly, the Hedge method [18] with multiple acquisition functions is employed to balance exploration and exploitation of the configuration space, with which ROBOTune can quickly converge to either an optimal or near-optimal configuration within a given budget. Furthermore, ROBOTune strengthens Latin Hypercube Sampling with caching and memoization to enhance the coverage and effectiveness in the generation of sample configurations.

We have performed an extensive set of tests with applications on machine learning, web search, and graph computation. Our evaluation demonstrate that ROBOTune outperforms contemporary tuning tools like BestConfig and Gunther by 1.14× and 1.15× on average, and up to 1.3× and 1.28×, respectively. In terms of search cost, the average improvement over BestConfig and Gunther is 1.59× and 1.53× and up to 2.27× and 1.71×, respectively.

2 BACKGROUND AND MOTIVATION

2.1 Spark Based Data Analytics

Spark is a cluster computing framework that has replaced Hadoop as the de-facto choice for running data analytics workloads. It provides easy to use APIs in various languages (e.g., Java, Scala, Python, R), which makes it very attractive to application developers for speedy development. Spark utilizes in-memory data processing, which makes it significantly faster than the previous generation of disk-based MapReduce frameworks for a range of increasingly important classes of workloads (e.g., machine learning, graph processing) [39].

Spark supports memory-resident data analytics and fault-tolerance through a memory abstraction termed Resilient Distributed Datasets (RDDs) [47]. An RDD is a collection of objects which is partitioned across the nodes of the cluster. RDDs that are used repeatedly can be cached in memory to avoid unnecessary recomputation, thus considerably speeding up iterative workloads. In Spark, a *master* node orchestrates the execution of a workload using several *worker* nodes. Within these *worker* nodes, independent JVMs named *executors* are launched that execute the required tasks.

2.2 Challenges for Parameter Configuration

High-dimensional Configuration Space. The number of parameters for data analytics on cluster systems continues to rise. For example, a large number of configuration parameters have been introduced to manage different aspects of execution, e.g., *runtime environment, shuffle, data serialization, memory management, networking*, for these diverse analytics workloads. The total number of parameters in Spark-2.4 has reached around 200 from its initial release with about 20 parameters. This translates to roughly an increase of 10×. These parameters exponentially increase the complexity of the configuration space.

Costly Sample Collection. Configuration tuning for cluster-based analytics applications requires sufficient sample points that can capture the performance impact of different parameters. However, sample collection is a critical challenge for both dimensionality reduction and BO initialization. A straightforward approach is random sampling of the configuration space. Many studies have noted the impact of increasing parameters for configuration tuning. For learning-based approaches like RFHOC [5], the number of required samples increases as more parameters are added to the consideration (3,300 samples needed as opposed to 2,000 when parameter count is increased to 34 from 10). McKay et al. [28] stated that random sampling requires a large number of points to ensure coverage of the space. Golovin et al. [16] reported that when the dimensionality of the configuration space is sufficiently high (e.g., over 16), even BO-based approaches can face difficulty in tuning. Collecting samples is also a costly proposition and thus demands an efficient method of sample generation.

Suboptimal Manual Configurations. As most data analytics workloads recur in a cluster [46], reduction of their execution time can be very beneficial in terms of both time and cost. However, the default configurations of analytics frameworks are often underperforming compared to near-optimal ones [5, 46, 49]. If an inexperienced user tries to tune the configuration through trial-and-error, it will waste valuable cluster resources without possibly yielding a good configuration. The increasing complexity of the configuration space and the infeasibility of manual tuning mandate an automation on the configuration tuning process. However, to be viable, auto-tuners must find near-optimal configurations in a fast and efficient manner and within a budget constraint.

2.3 Bayesian Optimization

Bayesian optimization (BO) is a global searching strategy to find the extrema of objective functions that are non-convex, non-parametric, or expensive to evaluate [8, 38]. It builds on top of Bayes' theorem [35] to construct a probabilistic model based on prior observations of an objective function, and incrementally select the next sample point to make another observation of the objective function.

Bayesian optimization can effectively avoid local minima and is very data efficient in its use of sample observations [38].

BO has been used to tune the hyperparameters of deep neural networks, where training a neural network requires a considerable amount of time and resources [20, 40]. BO has also shown its potential in the selection of near-optimal cloud VM instances [2, 19] using limited search cost, which is a desirable trait in our use-case. Storage configuration tuning has also seen the application of Bayesian Optimization [9]. In the field of DBMS tuning, OtterTune [45] showed that utilizing knowledge from previous tuning sessions can speed up the tuning process. All these studies have employed BO for its

intelligent reuse of prior configurations to achieve significant reduction on the cost of tuning. Thus BO is also appealing for searching near-optimal configurations of data analytics applications within a given budget.

3 DESIGN

In this section, we provide a design overview for the proposed ROBOTune framework and then elaborate on the design details.

3.1 Problem Formulation and Design Overview

Problem Formulation. Given a data analytics workload and its input, we aim to find an optimal or near-optimal configuration that minimizes the cost, i.e., the execution time. Let n denote the number of tunable parameters. Then a configuration $x = \{x_1, x_2, x_3, ..., x_n\}$ is a vector of n parameters. Further, let $f : \mathbb{R}^n \rightarrow \mathbb{R}$ denote the objective function. Our goal of finding the optimal configuration x^* can be expressed as the following:

$$x^* = \arg\min_x f(x) \tag{1}$$

But we do not know a closed-form expression or a definitive configuration to performance relationship for f. Thus f is a black-box function with a high-dimensional configuration space. As mentioned in §2.3, BO provides a good strategy to find x^* if we can employ it to search for the global extremum of f. However, because of the challenges caused by computation and statistical variance from high-dimensional settings, the efficiency and high accuracy of BO is limited to low-dimensional objective functions, typically 5 or less dimensions [30]. Therefore, alongside the adoption of BO for configuration tuning, we need to provide a dimension reduction strategy to prune the high-dimensional cluster configuration space.

Design Overview. Given our problem formulation, we have designed a robust tuning framework called ROBOTune that can leverage **R**andom F**O**rests and **B**ayesian **O**ptimization to **Tune** the configuration of cluster-based data analytics applications. Figure 1 provides a design overview of ROBOTune. It features three main components as described below.

- **Memoized Sampling** leverages Latin Hypercube Sampling (LHS) to ensure a fair selection of samples from a high-dimensional space. It also provides a parameter selection cache and a configuration memoization buffer to speed up tuning for repeated workloads with different inputs.
- **Parameter Selection** performs dimension reduction through Random Forests, based on samples from a high-dimensional configuration space.
- **Bayesian Optimization (BO) Engine** equips ROBOTune with a gaussian process (GP) model to estimate the objective function $f(x)$, and iteratively searches for the optimal configuration through a combination of three acquisition functions with balanced exploration and exploitation.

As shown in Figure 1, upon the arrival of an workload and a user-specified tuning budget, Memoized Sampling starts by checking its parameter selection cache. Upon a hit, it identifies a small selected set of parameters and generates a set of *LHS Tuning Samples* through Latin Hypercube Sampling. These tuning samples will be used as part of the *Training Set* for BO to search for the optimal configuration. Besides an output of the optimal configuration from BO, a few well-tuned configurations are stored at the configuration memoization buffer. These configurations will be provided as *Best Recent Configs* to be included in the training set when we need to tune the same workload with a different input dataset. Upon a miss, all the generic parameters will be included as a *Generic Set* to designate a high-dimensional configuration space. Through Latin Hypercube Sampling in this space, a large set of *LHS Generic Samples* will be generated. Parameter Selection then performs dimension reduction through Random Forests and identifies a small set of high-impact parameters for the workload. The selected parameters will be stored in the parameter selection cache to be reused for the same workload.

3.2 Memoized Sampling

In ROBOTune, it is critical to generate initial sample points for the BO engine to create a GP model and for the parameter selection component to select high-impact parameters. In addition, the optimal configuration for a workload can differ for different datasets. For example, because Spark is heavily reliant on memory, there is a good chance that the optimal value of parameters related to memory usage (e.g., *spark.memory.fraction*) changes when the dataset size is changed. However, many other parameter values may remain the same, and the new dataset's optimal configuration may lie near the same region of previous high-performance configurations. Thus it is important to reuse the results from prior sessions to expedite the tuning of repeated workloads.

Latin Hypercube Sampling. While the needs of Parameter Selection and BO engine are different, we need a sampling strategy to cover both configuration spaces evenly without generating too many sample points. Latin Hypercube Sampling (LHS) is a form of stratified sampling that needs fewer samples than random sampling to reach a similar conclusion without compromising the quality of the analysis [28]. LHS is also more stable with fewer samples than another established stratified sampling method, Monte Carlo [10]. Furthermore, LHS is dimension agnostic, meaning that the number of samples required is not tied to the dimensionality of the configuration space. Considering these advantages, we employ LHS to generate the samples for BO initialization and parameter selection. For generating M samples, the LHS divides every parameter range into M equally probable intervals and generates samples such that only one sample point is taken from each interval [4].

Parameter Selection Cache. We have observed that high-impact parameters remain the same for a specific workload within a range of different datasets. Thus, we store the previous selections in a table as the *Parameter Selection Cache*. Each row in the table contains a previously tuned workload and its high-impact parameters. As shown in Figure 1, a repeated workload will incur a hit in the cache and directly pull a *Selected Set* of parameters, which form a low-dimensional configuration space, for which LHS will generate a total of 20 *LHS Tuning Samples* for the BO engine. We have observed through experiments that a choice of 20 points works well for initializing the GP model. An unseen workload will incur a miss in this cache. In that case, 100 LHS samples on the *Generic Set* containing the initial parameters (in our case 44) will be generated for use by the parameter selection component. We further discuss the number of samples required for this case in §5.5.

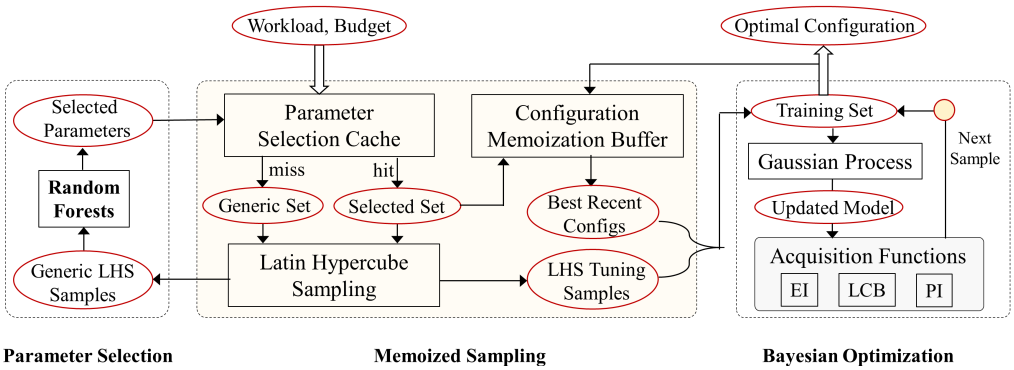


Figure 1: Design Overview of ROBOTune

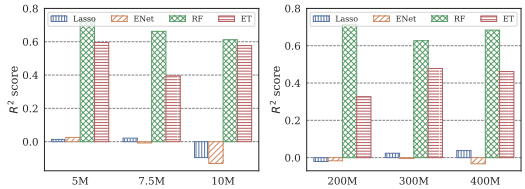
Configuration Memoization Buffer. We also find that previously tuned configurations are beneficial to the BO engine when tuning the same workload with a different input. Thus we create a *Configuration Memoization Buffer* to save a few high-performance configurations at the completion of each BO tuning session. We pull 4 *Best Recent Configs* from the Configuration Memoization Buffer for a workload that has been tuned before and combine with 16 LHS tuning samples to create the initial training set of 20. For an unseen workload, the initial training set only contains 20 LHS tuning samples. By incorporating high-performance configurations from previous tuning sessions, we can help the GP model identify a high-performing region in the configuration space for exploitation.

3.3 Parameter Selection

As discussed in §3.1, BO prefers low-dimensionality for efficiency and accuracy reasons. We propose to leverage machine learning models to select only the high-impact parameters for tuning, thereby reducing the complexity of the configuration space.

Choice of Random Forests. Parameter selection can be achieved by training machine learning models on a set of sample executions and measuring the strength of the relationship between the response variable and the configuration parameters. Dimension reduction techniques like Principal Component Analysis (PCA), which create linear combinations of the original features to reduce dimensions, are not applicable in our case, because we need to keep the original parameters for configuration tuning. Linear Regression models are one of the prevalent methods for computing feature importances [45]. However, Linear models (e.g., Lasso) require a sufficiently large sample size; otherwise, they can perform randomly. Linear models are also quite susceptible to collinearity between features and perform poorly when the modeled relationship is non-linear. Another way of computing feature importances is through tree-based estimators. Random Forests (RF) [7] is an ensemble learning method that utilizes a multitude of decision trees. It has shown to perform better than linear models given a similar number of samples for cluster computing frameworks [5] and is robust against over-fitting. Extremely Randomized Trees (ET) [15] is another promising tree-based estimator that we consider.

We compare the coefficient of determination (R^2) for two linear and two tree-based models in Figure 2. The coefficient of determination is a measure of how well observed outcomes are estimated by the model, based on the proportion of total variation of outcomes explained by the model [12], where the value ranges from 1.0 to negative for arbitrarily worse models. We generate 200 LHS configurations and collect the execution times for three input datasets of PageRank and KMeans workload and use them for training the considered models for comparison. The five-fold cross-validation scores for both linear-models (Lasso and ElasticNet) are significantly lower than tree-based models (RF and ET). RF performs the best and explains most of the variance. Note that we only need a model that identifies the features that are important and not be a perfect predictor. Considering all these, we decide to use Random Forests as the model for calculating the importance of parameters.



(a) PageRank. (b) KMeans.

Figure 2: R^2 scores of examined models for different datasets of the PageRank and KMeans workload. Higher is better.

Ranking the Parameters. Correctly calculating the feature (parameter) importance is crucial to our process. Parameter importance for random forests is commonly calculated using the Mean Decrease in Impurity (MDI) mechanism. However, Strobl et al. [41] states that the conventional method of utilizing MDI can be unreliable where potential predictor variables vary in their scale of measurement or their number of categories, which is true for our case. We thus consider the Mean Decrease in Accuracy (MDA) method, which is a bit slower than MDI but more robust [31]. For calculating MDA importances, we record a baseline using the out-of-bag (OOB) R^2 score first. Then each of the feature columns is permuted to see how much R^2 score may decrease. If a feature is not important, permuting the values of that column should not affect the R^2 score. Using this method of feature importance calculation, we identify the parameters that impact the workload performance.

Handling Collinearity. Presence of collinearity can hamper feature importance calculation. There are several dependent parameters in Spark, whose values are only valid when the independent parameter is active. To avoid issues introduced by collinearity during the feature importance calculation, we group the collinear parameters and permute them together.

3.4 Bayesian Optimation Engine

As shown in Figure 1, the BO engine is the pivotal component in ROBOTune to search for the optimal configuration. We describe its components in detail below.

Choice of Model. BO requires a multivariate regression model as a surrogate to estimate the objective function as mentioned in §3.1. Various models such as Gaussian Process (GP) Regression, Decision trees, Random Forests, have been previously employed [8]. We choose GP for our purpose because it provides a theoretically justified way to trade-off exploration and exploitation [21], and has been applied successfully to real-world systems [2, 24, 45].

GP is a distribution over multivariate functions. It can be completely specified by its mean function m and covariance function k: $f(x) \sim \mathcal{GP}(\mu, \sigma^2)$, where $\mu = m(x)$, $\sigma^2 = k(x, x')$, and x denotes an arbitrary point, x' another point in a stationary pair with x. Instead of a smooth parametric function, GP provides a statistical model that returns the mean μ and variance σ^2 of a normal distribution over the possible values of f for x [8]. Statisically, μ represents the model's estimation for the objective, σ^2 the uncertainty.

Choice of Acquisition Function. The acquisition function guides the search for the optimal configuration by progressively evaluating the possible rewards from candidate points and selecting the best candidate. The search efficiency in BO, in large part, is dependent upon the acquisition functions, which must balance between points that produce lower mean μ (better exploitation) or higher uncertainty σ^2 (better exploration), or both. There are three main choices for a minimization acquisition function [40]: (1) *Probability of Improvement (PI)* that optimizes the probability of improvement over the current best point x^+ [23]; (2) *Expected Improvement (EI)* that optimizes the expected improvement with respect to the current best value [29]; and (3) *Lower confidence bound (LCB)* that selects points with the best confidence interval [11].

Among the three, EI and LCB take both exploitation and exploration into account, while PI can be too biased towards exploitation [8]. Some recent research work [2, 13, 45] employ EI because of its demonstrated performance and ease-of-implementation. But EI is known to have an issue of under exploration [34]. Furthermore, it has also been empirically observed that the preferred strategy can change at various stages of the optimization process [38]. Thus no acquisition function is guaranteed to perform the best on an unknown objective function.

We adopt a strategy called Hedge [18] which constructs an adaptive portfolio of multiple acquisition functions and, at each iteration, chooses one probabilistically. The probability of choosing an acquisition function is updated based on the cumulative rewards (gain) at each step [3]. An adaptive portfolio of multiple functions often performs substantially better than the best individual function. In the portfolio, we adapt these acquisition functions to the problem of minimizing the execution time as follows:

$$\mathrm{PI}(x) = P(f(x) \leq f(x^+) - \xi) = \Phi\left(d/\sigma(x)\right) \tag{2}$$

$$\mathrm{EI}(x) = \begin{cases} d\Phi(d/\sigma(x)) + \sigma(x)\phi(d/\sigma(x)) & \text{if } \sigma(x) > 0 \\ 0 & \text{if } \sigma(x) = 0 \end{cases} \tag{3}$$

$$\mathrm{LCB}(x) = \mu(x) - \kappa\sigma(x) \tag{4}$$

where $d = (f(x^+) - \mu(x) - \xi)$, Φ and ϕ are the CDF and PDF of the standard Normal distribution, respectively, and ξ and κ are knobs that control the exploration-exploitation tradeoff [23, 27].

Algorithm 1 Bayesian Optimation with GP and Hedge

```
1: X: m initial configuration samples {x_1, x_2, ..., x_m}.
2: Y: An empty set of expensive evaluations.
3: t: Budget of total allowable evaluations.
4: for i = 1 → m do                                  ▷ Evaluate training samples
5:     y_i ← f(x_i)
6:     Y ← Y ∪ {y_i}
7: end for
8: for i = m → t do
9:     GP(μ, σ^2) ← GaussianProcess(X, Y)            ▷ Train a GP model
10:    x_i ← H(GP(μ, σ^2))                           ▷ Get optimal x_i via Hedge
11:    y_i ← f(x_i)                                  ▷ Evaluate x_i
12:    (X, Y) ← (X, Y) ∪ {(x_i, y_i}                 ▷ Augument priors (X, Y)
13:    Update accumulative gains for PI, EI, LCB in H.
14: end for
```

Integrating GP and Hedge. Algorithm 1 shows the tuning process of our BO Engine with the GP model and the adaptive Hedge portfolio. With a set of m initial configurations, it collects a set of prior observations of our objective function (i.e., execution time) through expensive evaluations. In a loop (Lines 9-13), the BO engine then (1) trains a GP model (*GP*) with these priors, (2) selects a new point using the Hedge-based portfolio function ($\mathcal{H}$) based on statistics from the GP model, (3) makes another observation of the objective function, and (4) updates the set of prior observations and accumulative gains of individual acquisition functions in Hedge. This process attempts to locate the global minimum of the objective, or nearly so within a limited budget of t allowed evaluations.

4 IMPLEMENTATION

ROBOTune is implemented in Python utilizing Random Forests from the popular Scikit-learn library [32] for parameter selection and Scikit-Optimize [37] for Bayesian Optimization. Scikit-Optimize provides a customizable baseline for sequential model-based optimization. Our customizations include the development of memoized sampling, custom GP covariance kernel, and automated early stopping. For sample generation, we have used the DOEPY [36] library as a space-filling LHS implementation. In this section, we provide some notable implementation details of ROBOTune.

Configuration Encoder. We must encode the sample points (numeric vectors) generated by the LHS sampler and the BO engine into a workload configuration. The numeric values from the sample point are converted into different types of suitable parameter types (e.g., boolean, categorical, size, time) and passed through a configuration file to the submitted workload.

Guard against bad configurations. During the execution of initial samples, we use a static threshold to prevent extremely imbalanced configurations from running an inordinate amount of time. In addition, during the search using the BO engine, a configurable multiple of the median execution time is used as a threshold for stopping imbalanced configurations.

Parameter Selection. Some configuration parameters exhibit high collinearity with each other. We group such collinear parameters as a *joint parameter* during the parameter importance calculation. Besides grouping collinear parameters, we also create

joint parameters from our domain knowledge (e.g., *executor size* by grouping *spark.executor.cores* and *spark.executor.memory*). Creating these joint parameters enables robust calculation of parameter importance. To differentiate important parameters from the non-important ones, we empirically set a drop of at least 0.05 in R^2 score as our threshold, which is configurable. In our experience, the current value reliably purges parameters that sometimes gain a bit of importance due to execution noise. We permute each parameter 10 times and take the average of the drop in accuracy to get a reliable and stable ranking.

Bayesian Optimization. To account for noises in the observations, we assume them to be i.i.d. following gaussian distribution. The acquisition functions are optimized using the L-BFGS-B optimizer. For ξ and κ, we choose a value of 0.01 and 1.96, respectively, as they perform well in most cases [18]. Furthermore, we choose the summation of Matérn $\frac{5}{2}$ and white noise as the covariance kernel, which is preferred to model practical functions [2, 40]. The white noise kernel is used to account for the noise in the objective evaluation.

Our implementation of the components of ROBOTune is highly modular. We choose Spark as our tuning target, as it represents systems that exhibit issues of high-dimensionality and complex multi-modal configuration-performance relationship. We note that some modifications are needed in the parameter selection and configuration encoder to apply ROBOTune to other systems, while other components can be mostly reused.

5 EVALUATION

5.1 Experimental Setup

Cluster Configuration. Our experimental platform consists of six nodes; one serves as the master, and the other five are workers. Each of our nodes is equipped with two 16-core 2.1 GHz Intel(R) Xeon(R) Gold 6130 CPUs, 192 GB of memory, and a 7200-RPM 2 TB Seagate hard disk. A 10-Gigabit Ethernet network connects the nodes. The experimental platform has a total of 192 cores and 1152 GB memory, which is sufficiently high, considering the testbeds of similar research work [4, 5, 25, 46, 49]. The tuning is done on top of Spark 2.4.1, and HDFS 2.7.3 is used for storing data.

Configuration Parameters. Spark has a long list of parameters that influence performance and some that don't (e.g., app properties, UI). We chose to tune a total of 44 performance-related ones, which is a superset of considered parameters of previous research work that tuned Spark configurations [4, 46, 49], minus a few deprecated and unsuitable ones (e.g., streaming parameters). As discussed earlier (§2.2), an exhaustive search is infeasible due to the enormity and high-dimensionality of the configuration space. For example, just considering two parameters *spark.executor.cores* and *spark.executor.memory* with value ranges of (1-32 cores) and (8-180 GB) leads to a configuration plane of 5,504 possible values.

Comparative Solutions. We have considered several prior work to compare with ROBOTune. However, as we discussed before (§1), model-based approaches require at least 2,000 executions of each workload to train models and are infeasible in most real-life scenarios. We select search-based solutions that operate within a user-provided budget, and these are described below.

BestConfig is a search-based tuning approach that uses divide-and-diverge sampling and recursive bound-and-search to find near-optimal configurations [49]. BestConfig is open-sourced[1], and we implement the necessary scripts to integrate it into our cluster.

Gunther is a configuration tuner for Hadoop, which utilizes genetic optimization with aggressive selection and mutation steps for searching near-optimal configurations within a resource-budget [25]. We implement Gunther for Spark, utilizing the DEAP [14] library.

Random Search (RS) [6] is a simple search-based tuning approach where parameter value ranges are explored uniformly at random. It has been shown to be well-performing when the search space is high-dimensional in tuning hyper-parameters and configuring data analytics frameworks [4, 6].

ROBOTune and BestConfig both have a stopping mechanism to guard against long-running bad configurations. To make the search cost of tuning comparable across tuners, we augment Gunther and RS with a static threshold-based mechanism to stop imbalanced configurations from running too long.

Workloads. Five representative workloads (Table 1) are selected from SparkBench [1]. These include popular machine-learning (KMeans and Logistic Regression), graph-computation (PageRank and ConnectedComponents), and micro-benchmark (TeraSort) workloads. The corresponding datasets are generated using Sparkbench. For each workload, we use three different datasets (D1, D2, D3 as listed in Table 1) and run each tuner 15 times, five for each dataset. We use a budget constraint of 100 executions for all tuners and set the time limit of evaluating each configuration to 480s.

Objective. We minimize the workload's execution time as the objective, as similar work target this metric in their evaluation. It should be noted that by modifying or replacing the objective function, ROBOTune can be easily adapted for optimizing other metrics.

Table 1: Workloads and their datasets

Workload	Input Datasets (D1, D2, D3)
PageRank (PR)	5, 7.5, 10 (Million Pages)
KMeans (KM)	200, 300, 400 (Million Points)
ConnectedComponents (CC)	5, 7.5, 10 (Million Pages)
LogisticRegression (LR)	100, 200, 300 (Million Examples)
TeraSort (TS)	20, 30, 40 (GB)

5.2 Performance of Optimal Configurations

Comparison with other tuners. We scale the execution time of the best performing configurations of ROBOTune, BestConfig and Gunther by the ones found by RS. Figure 3 shows the results for different workloads and datasets. ROBOTune outperforms BestConfig by 1.14× on average and up to 1.3×. Similarly, it beats Gunther by 1.15× on average and up to 1.28×. For RS, the speedup is 1.15× on average and up to 1.27×. These results demonstrate that ROBOTune finds better configurations under the same budget constraints. Across all the cases, ROBOTune is more beneficial for workloads like PR, CC, and LR, which benefit from fine-tuning through exploitation and likely have small high-performing configuration regions. For TS, the improvement is mediocre (~1.1×). For KM, all tuners find configurations with similar performance (difference of less than 10%). We believe that, for KM and TS workloads, the high-performing configuration regions are sufficiently large and can be

[1] https://github.com/zhuyuqing/bestconf

found just through exploration, and further fine tuning by exploitation is not required. Thus all four tuners are capable of finding these regions and identifying the (near) optimal configurations.

We find that the search-based tuners (i.e., BestConfig and Gunther) perform very similar to RS, as they lean heavily towards exploration and do very little exploitation. BestConfig suggests a sampling set size of 100 with a single round, which leads to only divide & diverge sampling (exploration) with no recursive bound and search (exploitation). Gunther's initial configurations are generated randomly and comprise a significant portion of the allocated budget, leading to a lack of balance between exploration and exploitation.

Comparison with the default. We also compare the performance of tuned configurations to the default one. It was reported that the default configuration performs poorly for large inputs [46]. For our cluster and evaluated dataset sizes, the problem is more pronounced as the default value of *spark.executor.memory* being only 1024 MB is too low and leads to Out-Of-Memory (OOM) errors in resource-intensive workloads like PR and CC. For KM and LR, the speedup over the default is 27.1× and 2.17× on average. For TS, the speedup over the default configuration for 20GB input is 4.16×, while for the other two larger datasets, TS encounters runtime errors.

5.3 Search Cost

We evaluate all four tuners on the cost of searching the optimal configurations. Here we define the *search cost* as the total time to generate and evaluate configurations during each tuner's search for optimal configurations. For ROBOTune, we do not include the initial cost of evaluating sample configurations for parameter selection, because it is a one-time cost for a new workload. This initial cost is later discussed in §5.5. Figure 4 shows the search cost of the tuners scaled to RS. ROBOTune outperforms other approaches significantly in terms of search cost. ROBOTune outperforms BestConfig by 1.59× on average and up to 2.27×. The improvement over Gunther is 1.53× on average and up to 1.71×. Similarly, RS is outperformed by 1.6× on average and up to 1.93×. Because of the significant reduction in cost, ROBOTune is very attractive for workloads where dataset sizes change and require retuning.

We further examine the reason of different search costs for these tuners under the same tuning budget of 100 executions. We plot the distribution of execution time of the sampled configurations for two representative workloads (Figure 5), PR and KM, where ROBOTune finds better and similar configurations compared to other tuners, respectively. We observe that BestConfig, Gunther, and RS execute many low-performing configurations, resulting in much higher search costs. For ROBOTune, the distribution of execution time centers around a low median, with only a few poor configurations. For PR and KM, the median of execution time for BestConfig is 1.53× and 1.42× that of ROBOTune, respectively. For Gunther, the median is 1.52× and 1.35× of ROBOTune, for PR and KM, respectively. We observe similar results for RS, where the median is 1.53× and 1.35× of ROBOTune. KM especially shows a long tail on the distribution, where at the 90th percentile, the execution time for BestConfig, Gunther, and RS is 4.21× ,3.42×, and 3.5× that of ROBOTune. As KM caches all RDDs in memory, configurations that cause RDD evictions take significantly more time. ROBOTune can infer this using BO and avoid the execution of such configurations.

Table 2: Avg. number of iteration needed to reach within a certain percentage of the avg. best achieved time.

Workload	Within 1%	Within 5%	Within 10%
PageRank	83	33	26
KMeans	57	17	12
ConnectedComponents	70	32	21
LogisticRegression	42	20	20
TeraSort	86	37	19

We notice a significant similarity between the execution time distribution of Gunther and RS, which can be attributed to the randomly generated initial samples of Gunther. Both benefit from the introduction of a threshold-based stopping mechanism, which reduces their search cost. The stoppage of imbalanced or invalid configurations manifests as an increase of samples at the tail end of the execution time distributions. We also observe that even though we provide the same configuration execution time limit for all tuners, BestConfig's distribution shows a bigger range due to its policy of modifying the threshold during runtime.

5.4 Search Speed

A primary advantage of BO is that it requires a limited number of samples to reach optimal configurations. We evaluate the number of iterations needed by ROBOTune to get within a few percentages of the best execution time. We refer to this as the *search speed* of ROBOTune. Table 2 provides the search speed, i.e., the number of iterations for ROBOTune to reach within 1%, 5% and 10% of the best performing configuration. It is evident that ROBOTune quickly finds configurations within 5% of the best. In our experience, we observe that memoized sampling helps find well-performing configurations very early in the search. We show an example of search speedup due to memoized sampling in Figure 6, which shows the minimum execution time of tuners at each iteration for two datasets of the PR workload. When no memoized configurations are available (PR-D1), ROBOTune needs 58 iterations to reach within 5% of the observed minimum. However, with well-performing memoized configurations in the initial set, only 21 iterations on average are required for PR-D3. In general, using well-performing configurations from previous tuning sessions gets ROBOTune within ~10% of the best observed time, and from there, the BO-engine further improves the performance. We can also observe that ROBOTune performs better than other tuners at all steps after the initial iterations.

5.5 Minimizing Selection Overhead

For a new workload, we train a Random Forests model using a set of Generic LHS samples to detect the performance-critical parameters. A large number of samples is desirable to increase model accuracy, but it can drive up parameter selection cost in ROBOTune. Even though it is a one-time cost per workload, it is essential to minimize the number of samples while still identifying all critical parameters.

As discussed in §3.3, we rank and select the parameters based on their impact on the R^2 score. We take the performance-critical parameters identified by a model trained with 200 Generic LHS samples as the ground truth. To determine the minimal number of samples required to identify the same parameters, we calculate the recall score while decreasing the number of Generic LHS samples used to train the model. The recall or sensitivity score is measured

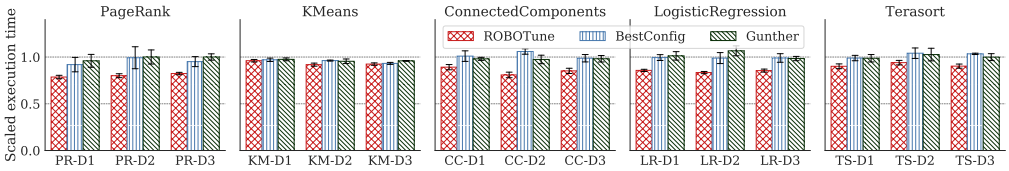


Figure 3: Execution time (lower is better) of suggested configurations scaled to Random Search. Datasets (D1, D2, D3) for each workload correspond to the order listed from left to right in Table 1.

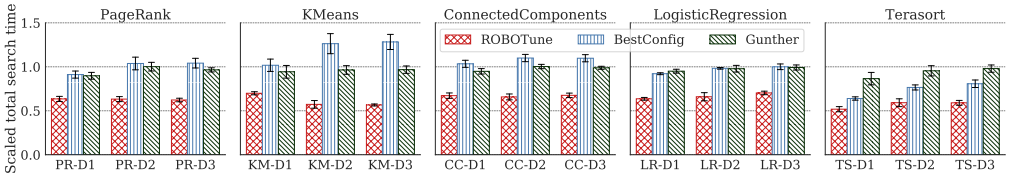
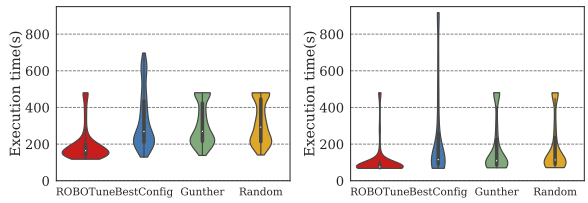
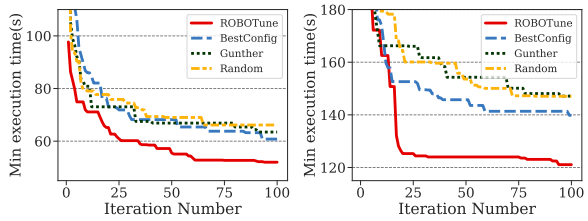


Figure 4: Comparison of search cost (lower is better) scaled to Random Search. The D1, D2, D3 of each workload correspond to the input datasets listed from the left to the right in Table 1.



(a) Distribution for PR-D3 (b) Distribution for KM-D3

Figure 5: Distribution of execution time for PR and KM.



(a) PR-D1 (b) PR-D3

Figure 6: Minimum Execution time at each iteration for two datasets of the PageRank workload.

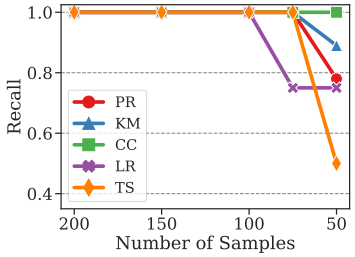


Figure 7: Recall scores with a decreasing number of samples for parameter selection of different workloads.

as the true positive rate, i.e., the fraction of parameters correctly identified compared to the ground truth set. A recall score of 1 means that none is missed. Figure 7 shows that the average recall score stays 1 until the sample count goes below 100. Thus in all our tests, we use a total of 100 Generic LHS samples in the parameter selection phase. Parameter selection is only required for unseen workloads, once for a range of datasets. ROBOTune is preferable in terms of cost when multiple datasets (e.g. two or more) of a workload are tuned, as the parameter selection cost is amortized across tuning sessions.

5.6 Exploration-vs-Exploitation Trade-off

We examine the exploration and exploitation behavior of ROBOTune compared to the other tuners. We show our results using PR as a representative workload and visualize the sampled configurations of each tuner for a tuning session of the PR-D3 dataset in Figure 8. We take the configuration plane formed by two parameters: *spark.executor.(cores, memory)*, because, these two are common in the selected set of high-impact parameters of all the tested workloads. A configuration with an imbalance between cores and memory performs poorly, especially for complex workloads like PR. One very noticeable distinction is that ROBOTune samples more points in a specific area while also sampling points from different regions of the configuration plane. On the other hand, the alternative tuners explore different sample points without any discernible pattern, showing little to no sign of exploitation.

We further examine how different regions are sampled by ROBOTune in Figure 9. We generate the perceived response surface of the cores-vs-memory configuration plane of the GP model at different iterations in the tuning process. As shown in Figure 9, ROBOTune has identified promising high-performing regions (in lighter color) even at iteration 25. This is due to the combined effects from the GP model, the acquisition function and a good set of training samples covering different regions of the configuration space. ROBOTune balances the need of exploration across the entire plane and the need of exploitation in high-performing regions (light-color areas with densely-populated points).

6 RELATED WORK

There is a large body of literature on auto-tuning the configurations of computer systems. DAC [46] is a learning-based auto-tuner that aims to find the optimal workload configuration for Spark using a combination of supervised machine learning (i.e., hierarchical model) and optimization techniques (i.e., genetic algorithm). Bei et al. [5] propose a similar line of research for tuning Hadoop named RFHOC, where Random Forest is used as the predictive model.

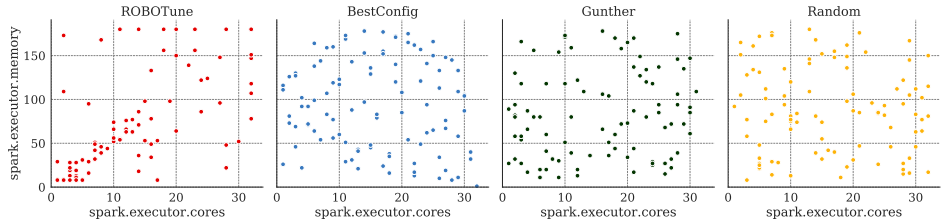
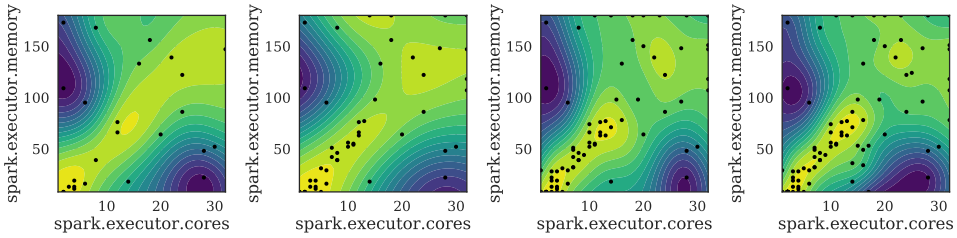


Figure 8: Sampling behavior of different tuners in the cores-vs-memory configuration plane.



(a) Iteration 25. (b) Iteration 50. (c) Iteration 75. (d) Iteration 100

Figure 9: Response surface of PR at different tuning iterations. Lighter color denotes better execution time. ROBOTune exploits more in the promising regions while exploring unknown regions.

Unlike DAC and RFHOC, which require many samples, ROBOTune aims to minimize sample collection to reduce tuning costs.

Several studies propose tuning system configurations using different searching algorithms. BestConfig [49] is a research work that aims to provide a framework-agnostic auto-tuning mechanism. The strategy is based on finding the best configuration within a resource limit by using divide and diverge sampling, and recursive bound and search algorithm. Unlike ROBOTune, BestConfig does not tackle the challenging issue of high-dimensional configuration space and, consequently, fails to improve much over Random Search. AutoTune [4] proposes an algorithm for creating a smaller-scale testbed to collect more samples within a limited time and utilizes a combination of learning and search-based approach. AutoTune only considers 13 configuration parameters, where ROBOTune automatically selects and tunes from a list of 44 parameters. Gunther [25] uses a genetic algorithm to search for optimal configurations of Hadoop workloads. Gunther only tunes six hand-picked parameters and the number of random configurations for initialization increase by two for each new parameter.

OtterTune [45] and iTuned [13] are research work that employ BO with GP to search for an optimal DBMS configuration. OtterTune uses supervised and unsupervised machine learning to reduce the configuration space and map unseen workloads to known ones. Metis [24] is an auto-tuning service that uses customized BO to reduce tail latencies of cloud systems. Google Vizier [16] is an internal service for black-box optimization within Google. CherryPick [2] and Arrow [19] uses BO with GP and Extra Trees respectively for recommending cloud VMs. While the concept of using BO to reduce search cost is similar, the cloud VM sub-space selected by the authors is orders of magnitude smaller than cluster computing configuration spaces. ROBOTune also differs from them in its reuse of knowledge and the use of a portfolio of acquisition functions. Besides OtterTune and ROBOTune, these approaches do not deal with the configuration space's high-dimensionality. However, OtterTune requires a large corpus of previous tuning sessions (~30k samples) to initialize the system, which is not necessary for ROBOTune.

7 CONCLUSION

To cope with the challenging issues of high-dimensionality and complex multi-modal configuration-performance relationship faced by the configuration tuning of cluster-based data analytics applications, we have designed and developed a tuning framework called ROBOTune. Our framework features Random Forests for parameter selection and Bayesian Optimization for balanced exploration and exploitation of the configuration space. ROBOTune is also equipped with memoization and caching to leverage previous tuning results for fast parameter selection and configuration tuning. Our evaluation with an extensive set of analytics workloads on machine learning, web search, and graph computation demonstrate that ROBOTune finds configurations that perform better on average while achieving a significant improvement in terms of both search cost and search speed.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work is supported in part by the National Science Foundation awards 1561041, 1564647, 1744336, 1763547, and 1952302, and has used the NoleLand facility funded by the National Science Foundation award CNS-1822737. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] [n.d.]. Spark-Bench. https://github.com/SparkTC/spark-bench
[2] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 469–482.

[3] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire. 1995. Gambling in a rigged casino: The adversarial multi-armed bandit problem. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*. 322–331.

[4] L. Bao, X. Liu, and W. Chen. 2018. Learning-based Automatic Parameter Tuning for Big Data Analytics Frameworks. In *2018 IEEE International Conference on Big Data (Big Data)*. 181–190. https://doi.org/10.1109/BigData.2018.8622018

[5] Z. Bei, Z. Yu, H. Zhang, W. Xiong, C. Xu, L. Eeckhout, and S. Feng. 2016. RFHOC: A Random-Forest Approach to Auto-Tuning Hadoop's Configuration. *IEEE Transactions on Parallel and Distributed Systems* 27, 5 (May 2016), 1470–1483.

[6] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-Parameter Optimization. *J. Mach. Learn. Res.* 13, null (Feb. 2012), 281–305.

[7] Leo Breiman. 2001. Random Forests. *Mach. Learn.* 45, 1 (Oct. 2001), 5–32. https://doi.org/10.1023/A:1010933404324

[8] Eric Brochu, Vlad M Cora, and Nando de Freitas. 2010. *A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning*. eprint arXiv:1012.2599. arXiv.org.

[9] Zhen Cao, Vasily Tarasov, Sachin Tiwari, and Erez Zadok. 2018. Towards Better Understanding of Black-box Auto-Tuning: A Comparative Analysis for Storage Systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 893–907.

[10] Liu Chu, Eduardo Souza de Cursi, Abdelkhalak El Hami, and Mohamed Eid. 2015. Reliability Based Optimization with Metaheuristic Algorithms and Latin Hypercube Sampling Based Surrogate Models. *Applied and Computational Mathematics* 4, 6 (2015), 462–468.

[11] Dennis D. Cox and Susan John. 1997. SDO: A Statistical Method for Global Optimization. In *in Multidisciplinary Design Optimization: State-of-the-Art*. 315–329.

[12] N. R. Draper and H. Smith. 1998. *Applied Regression Analysis* (3rd ed.). John Wiley & Sons, Inc., New York, NY, USA.

[13] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with iTuned. *PVLDB* 2, 1 (2009), 1246–1257.

[14] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research* 13 (jul 2012), 2171–2175.

[15] Pierre Geurts, Damien Ernst, and Louis Wehenkel. 2006. Extremely Randomized Trees. *Mach. Learn.* 63, 1 (April 2006), 3–42.

[16] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. 2017. Google Vizier: A Service for Black-Box Optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Halifax, NS, Canada) *(KDD '17)*. ACM, New York, NY, USA, 1487–1495. https://doi.org/10.1145/3097983.3098043

[17] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 599–613.

[18] Matthew Hoffman, Eric Brochu, and Nando de Freitas. 2011. Portfolio Allocation for Bayesian Optimization. In *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence* (Barcelona, Spain) *(UAI'11)*. AUAI Press, Arlington, Virginia, USA, 327–336.

[19] C. Hsu, V. Nair, V. W. Freeh, and T. Menzies. 2018. Arrow: Low-Level Augmented Bayesian Optimization for Finding the Best Cloud VM. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. 660–670.

[20] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. 2017. Fast Bayesian hyperparameter optimization on large datasets. *Electron. J. Statist.* 11, 2 (2017), 4945–4968. https://doi.org/10.1214/17-EJS1335SI

[21] Andreas Krause and Cheng S. Ong. 2011. Contextual Gaussian Process Bandit Optimization. In *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 2447–2455.

[22] Manoj Kumar, Mohd Husain, Naveen Upreti, and Deepti Gupta. 2010. Genetic Algorithm: Review and Application. *Journal of Information & Knowledge Management* 2 (12 2010), 451–454.

[23] Harold J. Kushner. 1964. A New Method of Locating the Maximum Point of an Arbitrary Multipeak Curve in the Presence of Noise. *Journal of Basic Engineering* 86 (1964), 97–106.

[24] Zhao Lucis Li, Chieh-Jan Mike Liang, Wenjia He, Lianjie Zhu, Wenjun Dai, Jin Jiang, and Guangzhong Sun. 2018. Metis: Robustly Tuning Tail Latencies of Cloud Systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 981–992.

[25] Guangdeng Liao, Kushal Datta, and Theodore L. Willke. 2013. Gunther: Search-Based Auto-Tuning of MapReduce. In *Euro-Par 2013 Parallel Processing*, Felix Wolf, Bernd Mohr, and Dieter an Mey (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 406–419.

[26] Luo Lie. 2010. Heuristic Artificial Intelligent Algorithm for Genetic Algorithm. In *Advanced Measurement and Test X (Key Engineering Materials, Vol. 439)*. Trans Tech Publications Ltd, 516–521.

[27] Daniel James Lizotte. 2008. *Practical Bayesian Optimization*. Ph.D. Dissertation. CAN. AAINR46365.

[28] M. D. McKay, R. J. Beckman, and W. J. Conover. 1979. Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code. *Technometrics* 21, 2 (1979), 239–245.

[29] Jonas Mockus, Vytautas Tiesis, and Antanas Zilinskas. 1978. The application of Bayesian methods for seeking the extremum. *Towards global optimization* 2, 117-129 (1978), 2.

[30] Mojmír Mutný and Andreas Krause. 2018. Efficient High Dimensional Bayesian Optimization with Additivity and Quadrature Fourier Features. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 9019–9030.

[31] Kristin K. Nicodemus. 2011. Letter to the Editor: On the stability and ranking of predictors from random forest variable importance measures. *Briefings in Bioinformatics* 12, 4 (04 2011), 369–373. https://doi.org/10.1093/bib/bbr016

[32] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[33] Michal Podhoranyi and Lukas Vojacek. 2019. Social Media Data Processing Infrastructure by Using Apache Spark Big Data Platform: Twitter Data Analysis. In *Proceedings of the 2019 4th International Conference on Cloud Computing and Internet of Things* (Tokyo, Japan) *(CCIOT 2019)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3361821.3361825

[34] Carl Edward Rasmussen and Christopher K. I. Williams. 2005. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press.

[35] J.A. Rice. 2006. *Mathematical Statistics and Data Analysis*. Number p. 3 in Advanced series. Cengage Learning.

[36] Tirthajyoti Sarkar. [n.d.]. Design of Experiment Generator in Python. https://github.com/tirthajyoti/doepy

[37] Scikit-Optimize. [n.d.]. Scikit-Optimize. https://scikit-optimize.github.io/

[38] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas. 2016. Taking the Human Out of the Loop: A Review of Bayesian Optimization. *Proc. IEEE* 104, 1 (2016), 148–175.

[39] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. 2015. Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics. *Proc. VLDB Endow.* 8, 13 (Sept. 2015), 2110–2121.

[40] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2* (Lake Tahoe, Nevada) *(NIPS'12)*. Curran Associates Inc., Red Hook, NY, USA, 2951–2959.

[41] Carolin Strobl, Anne-Laure Boulesteix, Achim Zeileis, and Torsten Hothorn. 2007. Bias in random forest variable importance measures: Illustrations, sources and a solution. *BMC Bioinformatics* 8, 1 (jan 2007). https://doi.org/10.1186/1471-2105-8-25

[42] Aparch Spark Team. 2012. Spark 0.6.0 Confguration. Retrieved June 2, 2020 from https://spark.apache.org/docs/0.6.0/configuration.html

[43] Aparch Spark Team. 2019. Spark 2.4.1 Confguration. Retrieved June 2, 2020 from https://spark.apache.org/docs/2.4.1/configuration.html

[44] Virginia Torczon and Michael W. Trosset. 1998. From Evolutionary Operation to Parallel Direct Search: Pattern Search Algorithms for Numerical Optimization. *Computing Science and Statistics* 29 (1998), 396–401.

[45] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 1009–1024.

[46] Zhibin Yu, Zhendong Bei, and Xuehai Qian. 2018. Datasize-Aware High Dimensional Configurations Auto-Tuning of In-Memory Cluster Computing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. ACM, New York, NY, USA, 564–577. https://doi.org/10.1145/3173162.3173187

[47] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (San Jose, CA) *(NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301

[48] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664

[49] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. ACM, New York, NY, USA, 338–350. https://doi.org/10.1145/3127479.3128605