# Phronesis: Efficient Performance Modeling for High-dimensional Configuration Tuning



YUHAO LI, Duke University, USA
BENJAMIN C. LEE, University of Pennsylvania, USA

We present Phronesis, a learning framework for efficiently modeling the performance of data analytic workloads as a function of their high-dimensional software configuration parameters. Accurate performance models are useful for efficiently optimizing data analytic performance. Phronesis explicitly considers the error decomposition in statistical learning and implications for efficient data acquisition and model growth strategies in performance modeling. We demonstrate Phronesis with three popular machine learning models commonly used in performance tuning: neural network, random forest, and regression spline. We implement and evaluate it for Spark configuration parameters. We show that Phronesis significantly reduces data collection time for training predictive models by up to 57% and 37%, on average, compared to state-of-the-art techniques in building Spark performance models. Furthermore, we construct a configuration autotuning pipeline based on Phronesis. Our results indicate up to 30% gains in performance for Spark workloads over previous, state-of-the-art tuning strategies that use high-dimensional models.

CCS Concepts: • **Computer systems organization** → **Real-time system specification**; *Cloud computing*; • **Computing methodologies** → **Modeling methodologies**; • **Information systems** → Computing platforms;

Additional Key Words and Phrases: Spark, performance modeling, autotuning, machine learning

**ACM Reference format:**
Yuhao Li and Benjamin C. Lee. 2022. Phronesis: Efficient Performance Modeling for High-dimensional Configuration Tuning. *ACM Trans. Archit. Code Optim.* 19, 4, Article 56 (September 2022), 26 pages.
https://doi.org/10.1145/3546868

## 1 INTRODUCTION

The emergence of distributed computing frameworks, such as Hadoop [27], Spark [2], Flink [20], and Storm [38], has enabled scalable data storage and processing in datacenters. These frameworks provide a variety of configuration parameters for optimizing hardware allocations, software properties, scheduling parameters, and so forth [1, 10, 11, 48]. Spark, an in-memory computing framework for data analytic jobs and SQL databases, provides more than 100 parameters for performance tuning [11]. Identifying good configurations in the space defined by these parameters is challenging, given the massive number of parameters and the diverse applications supported

Authors' addresses: Y. Li, Duke University, 308 Research Dr., Durham, NC, 27705; email: yuhao.li@duke.edu; B. C. Lee, University of Pennsylvania, 200 S. 33rd St., Philadelphia, PA, 19104; email: leebcc@seas.upenn.edu.

1544-3566/2022/09-ART56 $15.00
https://doi.org/10.1145/3546868

by these frameworks. Manually tuning configurations requires substantial knowledge of specific frameworks and is intractable. Instead, automatic configuration and optimization are needed.

Previous studies on tuning distributed computing frameworks have proposed various approaches to optimize configuration parameters. These approaches can be classified into three strategies that use rules, search, or models. Rule-based approaches tune parameters based on expert knowledge from past tuning experience [3, 5, 12, 66]. These strategies normally focus on tuning a few specific parameters and cannot scale to a high-dimensional configuration space.

Search-based methods repeatedly execute the applications with different parameters identified by advanced search techniques [34, 55] such as genetic algorithms [49] or divide and diverge sampling [76]. Then they choose the best-performing configuration encountered during the search. However, the number of sampled configurations required increases exponentially with the number of parameters [60], and search tends to find sub-optimal configurations in high-dimensional spaces. Furthermore, search lacks the flexibility to pursue multiple optimization targets and instead optimizes one specific target at a time. As a result, search may abandon previously sampled configurations and start a new trajectory through the space when the optimization target changes.

Finally, model-based tuning uses performance models [14, 18, 22, 29, 39, 46, 47, 59, 63, 64, 71, 72]. These techniques first collect a substantial amount of training data and then use that data to build a machine learning model to precisely predict each configuration's performance. Embedding the predictive model within an optimization heuristic allows a performance analyst to explore the parameter space efficiently.

Performance models can accurately predict and combine a variety of system metrics to define flexible optimization targets. However, training these models requires a massive amount of training data because the configuration space's size increases exponentially with the number of parameters. The required amount of training data also increases with workload diversity, which is typically high in datacenters. Data collection requires sampling configurations for measurement, which consumes computational resources and time, making performance models less attractive in datacenters. Although previous works have proposed techniques to accurately model configuration performance, few of these works address the need to build predictive models efficiently.

**Bayesian Optimization (BO)** [14, 21, 36, 50, 56] partially addresses these efficiency challenges by building a performance model that directly guides the search for an optimal configuration. An acquisition function predicts the likelihood of improving application performance for unknown configurations. BO uses this function to identify and evaluate new configurations, improving data efficiency. However, the BO model only accurately predicts configurations that lie in the search path of the optimization procedure and cannot provide accurate predictions over the global space. Such an incomplete model lacks the flexibility needed to solve new optimization problems for the same application, which is common for Spark applications in commercial clouds. For example, the resources allocated to Spark computation depends on a user's budget. If the budget changes, the most effective Spark configuration may change and the user must re-optimize. Unfortunately, BO must be re-run from scratch for each new budget because the model developed for the old optimization problem may not provide reasonable predictions for configurations explored in the new problem.

Finally, building performance models often requires users to specify hyperparameters to describe model structure, which impacts accuracy [76]. Previous works rely on empirical results or prior knowledge to set hyperparameters statically, a strategy that is not easily adapted to varied system settings.

In this article, we focus on efficient performance modeling to predict system outcomes as a function of configuration parameters. We propose Phronesis, a machine learning framework that reduces the amount of training data required for accurate performance modeling. Phronesis

strategically samples new configurations for training data and dynamically adjusts the model architecture during data collection.

It achieves these goals by explicitly reasoning about error decomposition. First, Phronesis considers **prediction variance** with ensemble learning, which trains multiple models and compares their outputs. Measures of prediction variance guide configuration sampling, identifying parts of the configuration space that most improve model accuracy [44]. Second, Phronesis mitigates **prediction bias** by dynamically and cautiously growing the model during data collection and training. Dynamic model growth matches model capacity with the amount of training data, avoiding both under- and over-fitting problems.

We demonstrate Phronesis for three categories of machine learning models—neural network, random forest, and regression spline. We implement and evaluate Phronesis for the Spark configuration space, deploying benchmarks for five data analytics workloads. We show that Phronesis reduces model training costs by up to 57% and 37%, on average, when compared against previously proposed methods for building Spark performance models. Furthermore, we deploy Phronesis within a model-based autotuning pipeline, which improves Spark performance by up to 30% when compared against state-of-the-art model-based strategies for tuning high-dimensional Spark configurations.

## 2 PERFORMANCE MODELING FOR SPARK

This section provides a case study of building performance models for Spark, a distributed framework for data analytic applications. For a given workload (defined as an application-input pair), we are interested in predicting its execution time, denoted by $y$, as the function of a Spark configuration, denoted by $x$. The vector $x$ is a set of values for parameters described in Table 1.[1]

We summarize Spark's important characteristics and illustrate the need for data-driven modeling for performance tuning. Next, we provide an overview of past model-based techniques for tuning software configurations and indicate their limitations in performance modeling for Spark. Finally, we describe how prediction error can be decomposed, according to statistical learning theory, and how the decomposition has implications for modeling strategies.

- **Efficient Data Acquisition**—Acquire configurations that are most useful for reducing variance in models' expected predictions.
- **Dynamic Model Growth**—Refine model structure to reduce bias in models' predictions.

### 2.1 Spark Parameter Study

Spark is a distributed, in-memory computing framework that supports varied applications in data analysis such as machine learning and graph processing [2]. Spark defines a **Resilient Distributed Dataset (RDD)**, an abstraction of distributed memory [73], and provides programmers with composable operators to compute on it. An application defines a sequence of RDD operators and specifies dependencies between them in a directed acyclic graph. Operators are organized into stages, each with a set of parallel tasks that operate on RDD partitions. The Spark run-time system schedules tasks on executors distributed across worker machines. Executors are Java Virtual Machines that execute tasks, which are Java threads, and store the partitions of the RDD.

Applications are sensitive to the Spark configuration. Spark provides more than 100 parameters. Configuration parameters specify computational resources (i.e., CPU time and memory) for software processes distributed over worker machines. Second, workload parameters

[1]We consider Spark parameters without loss of generality; $x$ could also include hardware configurations and application-level features.

configure both thread-level (i.e., `tasks.cpu`) and data-level parallelism (e.g., `default.parallelism`, `files.maxPartitionBytes`). Third, parameters configure data compression (e.g., `io.*`) and serialization (e.g., `kyroserializer.*`) for transfers through the disk and network. Other parameters (e.g., `shuffle.*`) determine how data is shuffled between executors to synchronize intermediate results. Finally, memory management parameters (e.g., `memory.*`) configure the layout of Java virtual machines, which is important when storing large in-memory objects.

Table 1 shows representative parameters of these important categories for performance tuning. The default values of these parameters are often far from optimal for varied applications, and tuning them can improve performance by up to 5$x$ [72]. Although these parameters are Spark-specific, many of them represent common parameters in other cluster computing frameworks [1, 5, 10, 11].

We make several observations about these parameters. First, these parameters constitute a parameter space of approximately $10^{25}$ configuration points, presenting challenges in generating heuristic search based on expert knowledge and intuitions. Second, parameters interact in complex ways. The effect of `shuffle.file.buffer` depends on `memory.fraction`, which controls the fraction of heap space allocated for computing and temporal storage; `default.parallelism` depends on `executor.core`, which controls per-machine parallelism for the application. Identifying and modeling interactions manually is difficult, motivating modeling strategies that discover and account for these interactions automatically.

Third, some important parameters represent trade-offs in resource and performance. For example, tuning `driver.cores` and `executors.cores` trades off core resources and computation performance, whereas tuning `executor.memory` trades off in-memory resources and data-loading performance. Spark users may have different optimization targets that weigh resource and performance differently. For example, users running Spark on private clusters may prefer maximizing performance, whereas users running on public clouds may prefer minimizing cost with a performance constraint. Building accurate performance models enables flexible optimization targets.

Finally, an application's computational characteristics determine the parameters that affect performance most. Applications that frequently communicate data between machines are affected more by network and shuffle parameters. Applications with massive parallel computations are more sensitive to workload parallelism and CPU resource parameters. Applications that need to store large in-memory objects are more sensitive to the memory parameters. As a result, it is difficult to specify a single set of important parameters for all Spark applications.

### 2.2 Model-based Tuning

Previous research has proposed various machine learning techniques to predict software configuration performance and tune configurations with those predictions. These techniques include tree-based regression models [17, 19, 22, 29, 72], support vector machines [47, 71], multivariate linear regressions [59, 63], and neural networks [39, 46, 64]. Although these works demonstrate the potential of using machine learning for accurate models, they neglect several practical issues.

First, training performance models require a substantial amount of training data. We generate training data by sampling configurations from the space and measuring their performance as the application runs. This procedure is time-consuming as application runs require from minutes to hours. In this setting, reducing the number of configurations for training a performance model is essential. Most works neglect these costs and use simple random sampling to generate configurations for training.

A few works propose more advanced techniques to improve sample efficiency. Latin-Hypercube Sampling can better cover the configuration space, compared with random sampling, with the

Table 1. Spark Configuration Parameters

| Parameter | Default | Tuning Range |
|---|---|---|
| driver.cores | 1 | [2, 12] |
| executor.cores | all workers' cores | [4, 32] |
| executor.instances | number of workers | [3, 12] |
| executor.memory | 1g | [2g, 48g] |
| memory.fraction | 0.6 | [0.5, 95] |
| memory.offHeap.size | 0 | [50m, 1000m] |
| broadcast.blockSize | 4m | [2m, 32m] |
| files.maxPartitionBytes | 128m | [32m, 512m] |
| storage.memoryMapThreshold | 2m | [2m, 128m] |
| default.parallelism | operator-specific | [4, 96] |
| task.cpus | 1 | [1, 6] |
| reducer.maxSizeInFlight | 48m | [16m, 4096m] |
| shuffle.file.buffer | 32k | [2k, 128k] |
| shuffle.compress | true | [true, false] |
| shuffle.accurateBlockThreshold | 100m | [50m, 500m] |
| io.compression.codec | lz4 | [lz4, lzf, snappy] |
| lz4.blockSize | 32k | [16k, 256k] |
| snappy.blockSize | 32k | [16k, 256k] |
| zstd.level | 1 | [1, 5] |
| zstd.buffersize | 32k | [16k, 256k] |
| shuffle.spill.compress | true | [true, false] |
| broadcast.compress | true | [true, false] |
| rdd.compress | false | [true, false] |
| kryoserializer.buffer.max | 64m | [32m, 256m] |
| kryoserializer.buffer | 64k | [32k, 256k] |
| driver.memory | 1g | [1g, 256g] |
| executor.memoryOverhead | 384m | [192m, 796m] |
| io.encryption.enabled | false | [true, false] |

same number of sampled configurations [17, 19]. However, maximizing coverage of the configuration space does not necessarily identify the most useful configurations for reducing a model's prediction error. Adaptively selecting configurations that can most improve model accuracy may help, but prior system studies are limited to specific classes of models—linear regression [63] and Delaunay Triangulation [23]—and neither class of models can effectively model performance from high-dimensional configurations.

Second, training requires specifying model hyperparameters, which determine the model's complexity and impact its prediction accuracy. Prior works are unable to specify hyperparameter values automatically. They rely on either manual tuning [17, 64, 72] or prior knowledge [63] to set hyperparameter values.

### 2.3 Error Decomposition in Performance Modeling

In machine learning, error decomposition provides insight in analyzing a performance model's expected generalization error. Given a workload's performance model, we can predict execution time $\hat{y}$ and measure actual execution time $y$ for Spark configuration $x$. The expected squared error

$\mathrm{E}[(y-\hat{y})^2|x]$ can be decomposed as

$$\mathrm{E}_{\mathbb{Y}}[(y-\mathrm{E}_{\mathbb{Y}}[y|x])^2]+\mathrm{E}_{\theta}[(\hat{y}-\mathrm{E}_{\theta}[\hat{y}])^2]+(\mathrm{E}_{\theta}[\hat{y}]-\mathrm{E}_{\mathbb{Y}}[y|x])^2.$$

The first term is **noise**, the difference between measured and expected performance for a Spark configuration. The second term is **variance**, the difference between the model's actual and expected prediction. The third term is **bias**, the difference between the expected prediction and the expected measurement. Because noise refers to the randomness inherent in the data generation procedure and is non-reducible from the perspective of performance modeling strategy, we focus on how to efficiently reduce bias and variance.

**Variance** accounts for prediction errors related to randomness in estimated model parameters, which in turn arises from randomness in the training procedure and data collection. First, model training often includes non-deterministic elements. Neural network training randomly initializes a model [28] and samples batch data [24]. Training a tree-based model randomly selects thresholds for splitting a node [35].

Second, randomness arises in system measurements and data collection. To generate a dataset for training, system architects sample randomly from the configuration space and measure each sample's performance. Models may over-fit to the performance of specific samples and therefore produce high variant predictions when fitted with different training samples.

A formal definition of model variance places a distribution on model parameter $\theta$ that is conditional on the training data $\mathbf{X}_{tr}, \mathrm{Y}_{tr}$:

$$\theta \sim P(\Theta|\mathbf{X}_{tr}, \mathbf{Y}_{tr}).$$

For a given training set, we can construct such a distribution by training multiple instances of models with bootstrapped samples from the training data. For a given input $x$, the prediction variance $\mathrm{Var}_{\theta}(y)$ is the variance of $y$'s estimate with respect to the variance in $\theta$. For example, variance in a simple linear model $y = Wx$ is given by the variance of linear combinations:

$$\mathrm{Var}_{w}[y] = \mathrm{Var}[W] \cdot x^2.$$

In practice, performance models with higher prediction variance will make less accurate predictions. System architects reduce prediction variance by sampling more configurations for training.

On the other hand, **Bias** describes errors produced by models despite having been trained with enough data. This type of error represents a model's inability to capture complex relationships in the data, usually due to simplified model structures. For example, a linear model cannot fit a cubic function regardless of training time and data. Model complexity is often determined by hyperparameters, such as regularization terms and model structure. Neural networks, regression splines, and tree models offer a large space of models defined by neural architectures, polynomial degrees, and tree depth, respectively, as shown in Table 2.

**Bias versus Variance.** For a given class of models, increasing model structure and complexity leads to a trade-off between higher bias and higher variance. A simpler model (e.g., lower-degree polynomials, fewer interaction terms, shallower neural networks) uses fewer model parameters and produces less variant predictions. But a simpler model's predictions may not become more accurate as the size of the training dataset increases; its simple structure cannot characterize the complicated interactions among configuration parameters and results in higher prediction bias.

In contrast, a complex model (e.g., higher-degree polynomials, more interaction terms, deeper neural networks) produces better predictions with more model parameters when more training data becomes available. But its predictions might be unstable and sensitive to specific configurations due to over-fitting [4], and this results in higher prediction variance.

Table 2. Model Hyperparameters

| Model Class | Model Parameters | Parameter Ranges |
|---|---|---|
| Neural Network | $1^{st}$ layer size | [4,∞] |
| | $2^{nd}$ layer size | [0,∞] |
| | $3^{rd}$ layer size | [0,∞] |
| Random Forest | tree numbers | [10,∞] |
| | tree depth | [3,∞] |
| Regression Splines | max degree | [1,5] |
| | # terms | [1,100] |
| | # interactions | [1, # terms] |

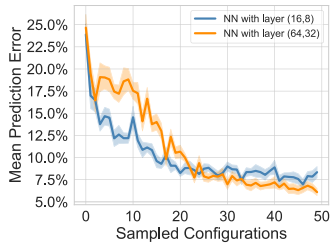



Fig. 1. Comparison between neural network models for predicting Spark logistic regression performance from five configuration parameters. Figure summarizes results from five repeated measurements with different random seeds. Lines show the averages and shaded areas show the standard deviation.

Figure 1 compares errors from neural networks that predict performance for Spark's logistic regression workload. The networks differ in model structure and training dataset size. The smaller network with layer size (16, 8) performs better when the number of sampled configurations is small (<1,000). But it cannot efficiently utilize more training data due to its simpler model structure and larger bias. The larger network suffers from high variance and cannot predict performance accurately when trained with small datasets. However, it gradually outperforms the smaller neural network as dataset size increases beyond 1,000 sampled configurations.

**Cross-validation and Generality.** $K$-fold cross-validation [43] is a popular, successful method for evaluating a model structure's generalized prediction ability considering bias and variance. Cross-validation divides data into $K$ groups. For the $k$th group, it takes group $k$ for test data and uses the remaining groups for training data. Cross-validation uses the average test score across the $k$ groups to estimate the model structure's generalized prediction ability.

Our experiments show that cross-validation produces different evaluation results when the amount of training data changes. Table 3 shows 10-fold cross-validation scores as model structure, for neural networks and random forests, and the amount of training data vary. When training data is relatively scarce (e.g., 50 to 150 configurations), cross-validation scores are higher with smaller models. When data is abundant, scores are higher with larger models. This insight, combined with observations from Figure 1, shows that simple models are preferred when there is insufficient data to balance variance and bias.

### 2.4 Implications for Performance Modeling

Training performance models is expensive and requires measuring outcomes for varied configurations. For a given workload-input pair, the time to train performance models depends on the time required to evaluate a configuration and the number of configurations that must be evaluated.

Table 3. 10-Fold Cross-validation for Neural Networks and Random Forests with Different Amounts of Training Data

(a) Neural Network for LogReg

| Data Size \ Layer Size | (8, 4) | (16, 8) | (32, 16) | (64, 32) | (128, 64) |
|---|---|---|---|---|---|
| 50 | −1.94 ± 0.19 | −0.67 ± 0.29 | −4.09 ± 0.24 | −0.67 ± 0.21 | −1.94 ± 0.199 |
| 100 | −0.06 ± 0.29 | 0.03 ± 0.14 | −1.52 ± 0.13 | −1.00 ± 0.21 | −0.49 ± 0.24 |
| 150 | 0.27 ± 0.14 | −0.66 ± 0.215 | −0.55 ± 0.2 | −0.45 ± 0.26 | −0.49 ± 0.18 |
| 500 | 0.45 ± 0.30 | 0.59 ± 0.21 | 0.69 ± 0.15 | 0.72 ± 0.14 | 0.69 ± 0.23 |
| 10,000 | 0.59 ± 0.32 | 0.72 ± 0.32 | 0.67 ± 0.23 | 0.81 ± 0.16 | 0.84 ± 0.09 |

(b) Random Forest for LogReg

| Data Size \ (tree num,tree depth) | (10, 2) | (30, 4) | (50, 6) | (70, 8) | (90, 10) |
|---|---|---|---|---|---|
| 50 | −2.50 ± 0.26 | −4.16 ± 0.26 | −2.41 ± 0.27 | −2.54 ± 0.31 | −2.28 ± 0.30 |
| 100 | 0.13 ± 0.1 | −0.01 ± 0.17 | −0.12 ± 0.26 | −0.27 ± 0.26 | −0.18 ± 0.25 |
| 150 | 0.43 ± 0.34 | 0.45 ± 0.28 | 0.49 ± 0.25 | 0.45 ± 0.29 | 0.39 ± 0.27 |
| 500 | 0.64 ± 0.08 | 0.71 ± 0.12 | 0.75 ± 0.11 | 0.78 ± 0.11 | 0.79 ± 0.12 |
| 10,000 | 0.71 ± 0.08 | 0.76 ± 0.07 | 0.82 ± 0.07 | 0.82 ± 0.09 | 0.88 ± 0.06 |

Table reports averages and standard deviations of $R^2$ scores; higher $R^2$ indicates better generalized prediction ability.

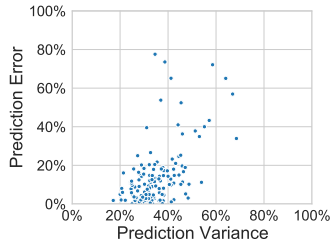



Fig. 2. Prediction variance versus error for 100 configurations. Results for neural network's performance predictions and Spark logistic regression workload. Configurations with higher variance are likely to report higher error.

Reducing evaluation time is difficult and constrained by the time required to read and process the dataset. As a result, we seek to reduce the number of configurations needed to train an effective model.

For efficient data collection, the system architect should identify configurations for which model predictions are likely to be inaccurate. Although the architect has not measured the performance of an unknown configuration, he or she can estimate the model's prediction variance for that configuration. Intuitively, the performance model makes inaccurate performance for the configuration for which it has high prediction variance and less confidence in the performance prediction.

Figure 2 quantifies this intuition, comparing prediction variance and prediction error as neural networks predict the performance of the Spark logistic regression workload. Configurations with higher prediction variance are associated with higher prediction errors. And updating the model with performance data from these configurations is more likely to improve predictive accuracy. Thus, the system architect should identify configurations with high prediction variance to improve a model.

**Active Learning.** Active learning [58] applies this idea in collecting the smallest possible dataset to achieve target accuracy. This technique initializes performance models with a small amount of uniformly sampled data points for training. It then strategically acquires new data samples by locating points with high prediction variances.

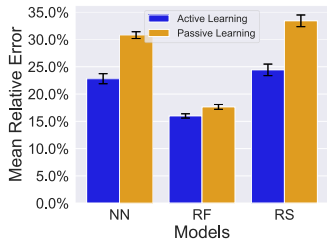



Fig. 3. Comparing active and passive learning. Results reported for performance predictions and Spark logistic regression workload. Random forest (RF), neural network (NN), and regression splines (RS) models are trained with 1,500 configurations to predict performance for previously unobserved configurations. Error bars show standard deviations from five repeated experiments with different random seeds.

Figure 3 illustrates the benefits of active learning for three classes of machine learning models: neural network, random forest, and regression splines. For each model class, the figure reports errors given the same number of measured configurations (i.e., training data). Active learning trains models with lower error by measuring configurations with higher prediction variance. It outperforms passive learning, which samples configurations uniformly at random. Active learning is particularly appealing when modeling distributed analytic applications, where data can be expensive to acquire, because it accurately predicts performance with fewer system profiles.

**Dynamic Model Growth.** Complementing data acquisition strategies that reduce variance, model growth strategies reduce bias. In statistical learning theory, the capacity and complexity of a model should grow linearly with the amount of training data in order to bound generalization error [32].[2] If updates to model structure increase capacity too quickly, a model may over-fit the training data because of its high expressiveness and may be sensitive to model variance. If updates increase capacity too slowly, a model may exhibit larger approximation uncertainty and bias between model predictions and true observations.

Dynamic modeling requires techniques that continuously collect data and periodically update not only model parameters but also model structure. The technique should start with a simple, minimally defined model architecture and dynamically grow the model as training data is collected.

Given our understanding of variance and bias, we propose Phronesis, a machine learning framework that accurately predicts performance with less data. It strategically collects training data with active learning and cautiously grows models to increase their capacity for accurate predictions.

## 3 PHRONESIS LEARNING FRAMEWORK

Figure 4 presents the Phronesis (Greek word for "practical virtue") framework. The Data Acquisition and Model Engines operate alternately. The Data Acquisition Engine generates configurations and acquires the data most beneficial for an existing model. The Model Engine grows the model architecture and trains new models with new and updated training data. Phronesis proceeds iteratively to generate a training dataset and to train an ensemble of performance models. In this article, models predict system outcomes (e.g., latency) as a function of system configuration (e.g., Spark parameters).

Phronesis supports three classes of machine learning models—neural networks, random forests, and regression splines—accounting for their relative merits. First, deep neural networks can model any complex, non-linear interaction of configuration parameters as stated by the universal approximation theorem [26]. The downside of neural networks is that they normally require a significant amount of training data and lack interpretability.

[2]One measure of model capacity and expressiveness is the **Vapnik-Chervonenkis (VC)** dimension.

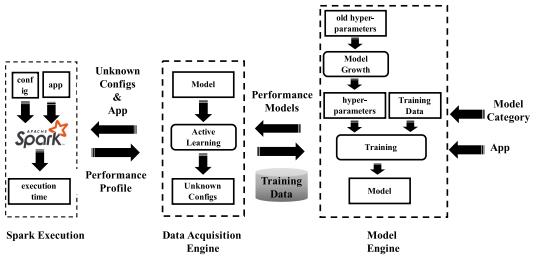



Fig. 4. Phronesis framework.

Second, random forests are a type of ensemble learning. They typically perform better than neural networks with small training datasets. Furthermore, they provide a feature importance metric that scores each input parameter's impact on the output. However, random forests' predictions are not smooth and may not fit a complex, smooth function.

Finally, regression splines provide interpretability. System architects can embed domain expertise into their specification of the regression model. And they can examine the model to determine the features and interactions between them that are used to capture the relationships between inputs and the output. System architects could infer the interaction between parameters from coefficients fit to the products of basis terms.

In this section, we detail the Phronesis framework in two parts. First, we describe how Phronesis estimates model variance and acquires data to reduce variance. Second, we describe how Phronesis grows models dynamically to reduce bias.

### 3.1 Data Acquisition and Variance Reduction

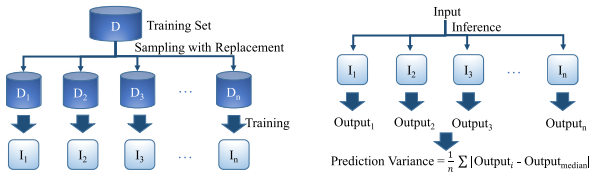
Prediction variance determines whether performance models need additional data from particular regions of the parameter space. Measures of prediction variance guide data acquisition and reveal unobserved configurations that, if measured, could most improve model accuracy.

**Estimating Prediction Variance.** Phronesis estimates variance with ensemble learning. First, Phronesis initializes an ensemble of $m$ models with distinct, random parameter values. It then independently samples training data and produces a different dataset to train each model in the ensemble. This bootstrapping technique trains $m$ distinct models and $m$ distinct performance predictions for each system configuration. Phronesis approximates the prediction variance for an unknown configuration by calculating the average deviation of each model's prediction $\hat{g}_i$ from the median prediction.

$$Var(\hat{g}) = \frac{1}{m}\sum_{i=1}^{m} |\hat{g}_i - \hat{g}_{median}|^2$$

For neural networks and regression splines, Phronesis maintains an ensemble of models and calculates variance in the ensemble's outputs as illustrated in Figure 5. We choose m = 10 to balance estimation accuracy and computation efficiency. For random forests, Phronesis maintains one instance of random forest, which is already an ensemble of decision trees, and calculates variance in the outputs from individual trees.

**Acquiring Configuration Data.** Phronesis scores each configuration in the parameter space with measures of prediction variance. A batch of configurations with high scores are evaluated to generate new training data. First, finding the configuration in a high-dimensional space that maximizes the acquisition score is difficult. Second, strictly searching for configurations with the highest scores may yield similar configurations for training. Two configurations with high scores can lie close to each other in the parameter space, and sampling both is redundant for training.




(a) Training Ensemble (b) Estimating Variance

Fig. 5. Estimating prediction variance with ensembles.

Solving these issues would permit us to evaluate multiple configurations at a time yet avoid evaluating similar, redundant configurations when one is sufficient [57].

Search heuristics, such as the **Genetic Algorithm (GA)**, can be used to search the configuration with the highest prediction variance. However, GA has two disadvantages. First, GA is quite computationally expensive and time-consuming. Our experiments show that GA typically takes tens of minutes to converge, which could match up with the time to evaluate unknown configurations. This is not desired in an iterative procedure in which we need to search for new configurations when the model is updated repeatedly. Second, GA normally converges to a single solution, but we want to acquire a batch of configurations at one time.

Phronesis addresses these challenges by seeking to maximize the mutual distances between configurations and maximizing the prediction variance in the training set. The distance approximates how similarly two configurations perform. By maximizing distance, Phronesis obtains configurations that are dissimilar to each other and also contain useful information for improving the model.

To calculate distances among configurations, Phronesis vectorizes configuration files and calculates vector distances. For continuous/numerical parameters, Phronesis records its value. For categorical parameters, it maps categories into multiple dummy parameters using one-hot encoding [68]. Furthermore, Phronesis re-scales the continuous parameters such that each parameter is in $[0, 1]$. Re-scaling ensures each parameter is considered equally in the similarity measure between two configurations. Then, for two vectorized configurations $c^1$ and $c^2$ of length $n$, Phronesis calculates the average Manhattan distance [7]:

$$dist(c^1, c^2) = \frac{1}{n} \sum_{i=1}^{n} \left| c_i^1 - c_i^2 \right|. \tag{1}$$

Phronesis seeks to maximize the mutual distance of sampled configurations to get a diverse set of configurations for training. It uses a two-step procedure. First, it samples $N = 5{,}000$ configurations uniformly at random from the space and ranks samples by their estimated prediction variance. Second, it initializes an empty set $S$ and iterates over ranked configurations. In each iteration, it adds a configuration if its average distance to those already in set $S$ and its average distance to the existing training set $S_{train}$ are both greater than a pre-defined threshold. The procedure halts when the set $S$ reaches some budget (e.g., 15). If it cannot find enough configurations to meet the budget, it re-samples another $N = 5{,}000$ configurations and repeats the procedure. We define the cut-off threshold to be the average distance between configurations in training set $S_{train}$. Phronesis evaluates the set of configurations, deploying the workload and measuring each configuration's performance.

### 3.2 Model and Ensemble Growth

Phronesis initializes its ensemble of performance models with a starting dataset, determining an appropriate model architecture (from a pool of 200 candidates randomly sampled from the

Table 4. Phronesis's Model Growth Strategies

| Model Class | Growth Strategy |
|---|---|
| Neural Network | Incremental neural architecture search, modified from [32] |
| Random Forest | Coordinate-ascent search |
| Regression Splines | Multivariate adaptive regression spline [30] |

**ALGORITHM 1:** Growing Neural Networks

1: **procedure** GROW($A = (i * b, j * b, k * b)$,$T$) ($A$ is current architecture, $T$ is training data)
2: $A^{new} \leftarrow A$
3: **for** `t = 1: 10` **do**
4: $(i, j, k, b) \leftarrow A^{new}$
5: $\mathscr{A} \leftarrow \{\}$ ▷ $\mathscr{A}$ represents new architecture candidates
6: $b^{new} \leftarrow \lceil \frac{(i+j+k)*b}{i+j+k+1} \rceil + 1$
7: $\mathscr{A} \leftarrow \mathscr{A} \bigcap ((i+1) * b^{new}, j * b^{new}, k * b^{new})$
8: $\mathscr{A} \leftarrow \mathscr{A} \bigcap (i * b^{new}, (j+1) * b^{new}, k * b^{new})$
9: $\mathscr{A} \leftarrow \mathscr{A} \bigcap (i * b^{new}, j * b^{new}, (k+1) * b^{new})$
10: $\mathscr{A} \leftarrow \mathscr{A} \bigcap (0, (i+j) * b^{new}, (k+1) * b^{new})$
11: $A^{'} = \arg\min_{A \in \mathscr{A}}$ {cross validation score of $A$ on $T$}
12: **if** cross validation score does not improve **then Break**
$A^{new} \leftarrow A^{'}$
13: **return** $A^{new}$

structure space) using 10-fold cross-validation [33]. The ensemble grows dynamically each time a large batch of new configurations (e.g., 150) is evaluated and added to the training set. Such dynamic growth gradually adjusts predictive structure, improving models' capacity to learn from new data and balancing between prediction variance and bias. Phronesis uses different model growth strategies for each machine learning model category, as summarized in Table 4, tailoring the strategy to each model's specific structure and hyperparameters.

**Neural Networks.** Algorithm 1 outlines the procedure for neural network growth. Phronesis modifies the incremental Neural Architecture Search Algorithm [32], which was originally designed for convolutional neural networks, for fully connected neural networks. Phronesis grows models within a space of neural architectures, restricting the model to a maximum of three layers.

The architecture is described by a three-value vector $(i * b, j * b, k * b)$, where the value in each position specifies the layer size as a multiple of the smallest granularity value $b$. The algorithm increases model size until model growth no longer reduces the loss function. In each iteration, Phronesis first tries to increase layer and network size (line 6). It then increases the ratio among the network's three layers (lines 7–10). Phronesis considers multiple candidates and chooses the one with the lowest cross-validation loss. If the loss does not fall from one iteration to the next, the algorithm terminates.

This growth strategy ensures that a neural network's model complexity, measured by VC-dimension, does not increase faster than the training datasets. The VC-dimension of a neural network is bounded by $O(\hat{L}W \log(U))$, where $\hat{L}$ is the average number of parameters per layer, $W$ is the total number of parameters, and $U$ is the total number of neurons [25, 67]. By adjusting each layer individually, Phronesis grows $\hat{L}W \log(U)$ sub-quadratically to ensure the rate of model growth is similar to that of training datasets.

Table 5. Spark Applications for Experiments

| Applications | Dataset | Data Size |
|---|---|---|
| PageRank | Amazon Review Pages [31] | 1.2M pages |
| TeraSort | Wikipedia Corpous [31] | 7.5 GB |
| Logistic Regression | kdda10 [61] | 20M features |
| K-Means | kdda10 | 20M features |
| Wordcount | Wikipedia Corpous | 10G |

**Random Forests.** Phronesis uses coordinate ascent to grow random forests incrementally. Phronesis increases the number of trees or each tree's depth until the generalized error does not improve. Specifically, consider a random forest defined by ($nt$,$td$), in which $nt$ denotes the number of trees, and $td$ denotes the tree depth. Phronesis creates two new models, ($nt+10$,$td$) and ($nt$,$td+1$). It then determines the best model from three candidates: ($nt$,$td$), ($nt+10$,$td$), ($nt$,$td+1$). Phronesis evaluates each candidate using the 10-fold cross-validation score on the training set.

**Regression Splines.** Phronesis uses **Multivariate Adaptive Regression Splines (MARS)** to automatically determine non-linearities and interactions between parameters in the spline model [30]. MARS uses a greedy two-phase approach to determine the appropriate splines for regression. In the forward pass, MARS greedily adds a pair of splines that result in the maximum reduction in the training loss. In the backward pass, MARS prunes the model, removing the least effective splines recursively until it identifies the sub-model with the highest generalized predictability. MARS uses **generalized cross-validation (GCV)** to score the generalized prediction ability of a regression spline where $RSS$ is the prediction error on the training set and $N$ is the size of the training set. GCV balances training errors and spline complexity:

$$GCV = \frac{RSS}{\left(N \cdot \left(1 - \frac{\#\ of\ parameters}{N}\right)^2\right)}.$$

## 4 EXPERIMENTAL METHODS

We implement Phronesis with over 1,500 lines of Python code and over 200 lines of bash scripts. We use Python to implement the data acquisition and model engines. We use bash scripts to deploy applications and collect performance data. We use Pytorch [54] to build neural networks. We use Scikit-Learn [9] to build random forests and regression splines. Finally, we use Pyearth [8] to build a model growth strategy for regression splines.

**Benchmarks.** We evaluate Phronesis by building predictive models for Spark, tuning its configuration parameters for five applications. Table 1 lists 28 parameters. We evaluate five data analytics workloads in Table 5 and use Phronesis to predict expected execution time on a given Spark configuration.

**Cluster.** We run Spark on a cluster, initiating four workers on four machines and colocating the master with a worker on one of these machines. Each machine has 48 cores and 256GB of memory. Phronesis runs on a separate machine with 48 cores and 256GB.

Phronesis trains models efficiently, requiring much less data than prior approaches. We compare Phronesis with four baseline methods for training machine learning models.

- **Single Model + Passive Learning (SM+PL)** trains a single predictive model with randomly sampled Spark configurations [46, 64]. We only implement this strategy for neural network and regression splines as random forest is already an ensemble learning algorithm.

- **Ensemble Learning + Passive Learning (EL+PL)** trains an ensemble of models with randomly sampled Spark configurations. Ensemble learning is popular for tree-based models for distributed computing [18, 36, 72].
- **Ensemble Learning + Active Learning (EL+AL)** trains an ensemble of models with actively sampled Spark configurations, which are selected for their high epistemic uncertainty [37].
- **Ensemble Learning + Passive Learning + Model Growing (EL+PL+MG)** trains an ensemble of models with randomly sampled Spark configurations. It adjusts the network architecture with Phronesis model growth strategy once 100 configurations are added to the training set.
- **Phronesis** combines ensemble learning, active learning, and dynamic model growing.

We initialize the neural network and random forest architectures by performing 10-fold cross-validation on 50 labeled configurations. We uses MARS to initialize regression spline terms also with 50 labeled configurations. We train neural networks with the Adam algorithm [42], setting both the learning rate and regularization term to 0.001. We train random forests and regression splines using default hyperparameters in the Scikit-Learn library.

We report **mean relative error (MRE)** on 200 random test configurations not used in training as the evaluation metric for predictive modeling. To account for randomness in machine learning training, we repeat each training procedure five times with different random seeds:

$$\text{MRE} = \frac{1}{200}\sum_{i=1}^{200}\frac{\left|y_i^{pred} - y_i^{obs}\right|}{y_i^{obs}}. \tag{2}$$

## 5 EVALUATION

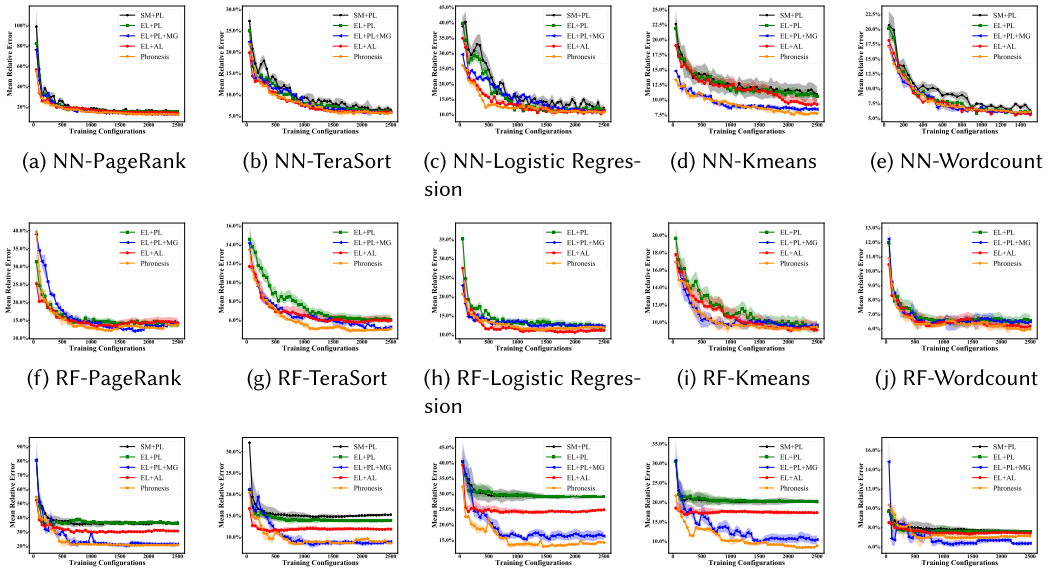
### 5.1 Training Models Efficiently

**Effects of Ensembles Alone.** Figure 6 compares model errors as the amount of training data increases, leading to a number of conclusions for three model categories. Ensemble learning improves prediction accuracy (SM+PL versus EL+PL) for training neural networks (Figures 6(a)–6(e)). Ensembles reduce the portion of prediction error due to model variance, the variance in parameter estimation.

However, ensembles do not consistently improve accuracy for regression splines when modeling performance for the five Spark applications (Figures 6(k)–6(c)). This observation suggests that splines initialized with small training data have large biases such that more effective variance analysis and data acquisition alone are also insufficient. We must also re-adjust splines during data collection.

**Effects of Active Learning Alone.** Active learning reduces error (EL+AL versus EL+PL), but benefits are observed inconsistently for different models and tend to be greater when data is scarce. For neural networks (Figures 6(a)–6(e)), active learning benefits PageRank, TeraSort, Logistic Regression, and WordCount more when the system has a modest amount of training data (e.g., fewer than 750 configurations). For random forest (Figures 6(f)–6(j)), active learning consistently outperforms passive learning for Logistic Regression. It also reduces errors and improves data efficiency for Pagerank, TeraSort, and Kmeans when training data is modest. It reduces errors for WordCount after more than 1,500 configurations are sampled. Active learning has a more significant impact on regression splines (Figures 6(k)–6(o)). Both active and passive learning converge quickly to stable prediction error in all experiments, but active learning results in substantially lower errors.

**Effects of Dynamic Model Growth Alone.** Dynamic model growth reduces error primarily after the system has acquired a significant amount of training data (e.g., more than 1,500




(a) NN-PageRank (b) NN-TeraSort (c) NN-Logistic Regression (d) NN-Kmeans (e) NN-Wordcount

(f) RF-PageRank (g) RF-TeraSort (h) RF-Logistic Regression (i) RF-Kmeans (j) RF-Wordcount

(k) RS-PageRank (l) RS-TeraSort (m) RS-Logistic Regression (n) RS-Kmeans (o) RS-Wordcount

Fig. 6. Comparison of training methods for three models: neural network (NN), random forest (RF), and regression splines (RS). Models predict execution time for Spark analytic workloads. Each line represents the average from repeated experiments, while the shaded area indicates the standard deviation.

configurations) when training neural networks. This observation suggests we require large training sets to effectively understand a neural network's bias and identify more effective model structures. One notable exception arises when we train a neural network for Kmeans (Figure 6(d)). Here, the model growth strategy significantly reduces error early in the training procedure when data is relatively scarce, which may reflect poorly initialized model structure. Initially, we perform 10-fold cross-validation to identify the most promising network architectures from a set of 200 sampled candidates. However, the 200 sampled candidates may not contain a good architecture.

Model growth has a more significant impact on prediction errors when training random forests (Figures 6(f)–6(j)) and regression splines (Figures 6(k)–6(o)). This result suggests that initialized model structures offer significant room for further improvement. We cannot rely on initial model selection alone because cross-validation may not be able to identify the best of candidate models given only modest training datasets.

**Benefits of Phronesis.** Phronesis systematically outperforms other training strategies by combining active learning and dynamic model growth. Neural networks for PageRank, TeraSort, and Logistic Regression workloads benefit from active learning's more informative configurations early in the training procedure and from dynamic model growth's better model structure later. In contrast, neural networks for the Kmeans workload benefits more from dynamic model growth at the beginning and then continues to train efficiently with active learning.

For regression splines, the benefits of Phronesis mainly come from dynamic model growth. For random forests, Phronesis benefits more from dynamic model growth in PageRank and Logistic Regression and more from active learning in TeraSort and Kmeans. Note that Phronesis may initially perform slightly worse than active learning alone, possibly because the dynamic growth strategy identifies an overly complex model at the start.

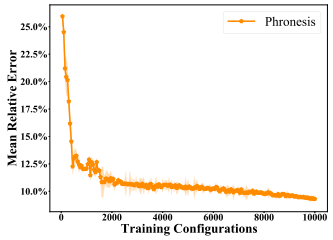



Fig. 7. Increasing the number of configurations in the dataset (up to 10,000) when training a neural network to predict the Logistic Regression workload's execution time.

Finally, note that neural networks and random forests perform better than regression splines for different applications at different numbers of training configurations. This suggests that neural networks and random forests effectively model the complex interactions among model parameters.

**Effect of Scaling Training Data.** We use up to 2,500 training configurations for each method in our comparison studies. We observe diminishing marginal benefits from scaling up the size of the training dataset for the varied methods. For example, Figure 7 shows the result of increasing the number of training configurations to 10,000 when using Phronesis to construct a neural network model for the Logistic Regression workload. We observe <1% reduction in prediction error as data size increases from 2,500 to 10,000. Therefore, it is sufficient to report Phroensis results for dataset sizes up to 2,500.

### 5.2 Efficient Active Learning

We also compare Phronesis's active learning strategy with two other representative sampling approaches used for modeling the performance of system and framework configurations. These sampling alternatives are **Latin Hypercube sampling (LHS)** [17, 19] and **Non-Invasive Modeling for Optimization (NIMO)** [59].

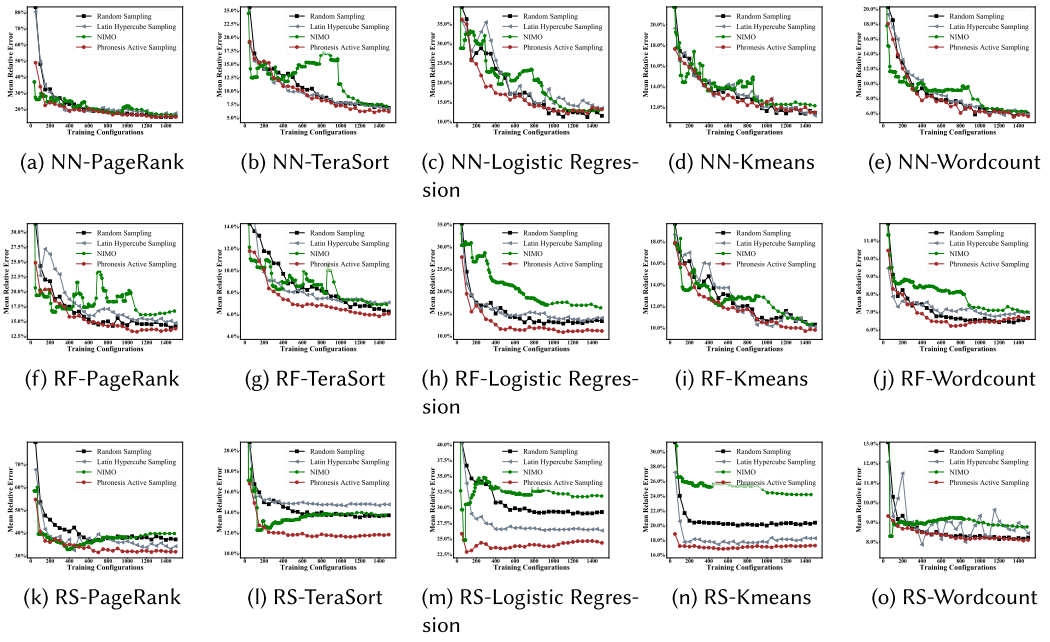
LHS is a statistical technique for generating quasi-random sequences in the high-dimensional space. LHS divides the high-dimensional space into small hypercubes. A hypercube is the generalization of squares in the two-dimensional space to high-dimensional spaces. LHS samples configuration such that each hypercube contains at most one sampled configuration. Prior work adopts LHS for performance modeling because it often provides better coverage of the parameter space than random sampling with the same number of sampled configurations.

NIMO is representative of approaches based on experiment design. It uses the Plackett-Burman method [70] to determine the importance of each parameter in determining application performance. Plackett-Burman generates $N + 1$ configurations for an $N$-dimensional parameter space to study the dependence of measured performance on each independent variable.

After determining each parameter's importance, NIMO then samples from a small configuration space that contains only the most important parameter. NIMO sequentially adds a new parameter to the configuration space, in order of importance, until there is no further reduction in prediction errors from more sampled configurations.

The original NIMO implementation discusses two approaches to sample efficiently from the subspace when a new parameter is added. The first uses multi-level, factorial design to explore only values of the newly added parameter. The second uses another Plackett-Burman design to explore combinations of different parameter values for the newly expanded space. Our implementation uses both approaches to generate new configurations.

Two other active learning approaches have been proposed for modeling the performance of data analytic applications. Ernest uses experiment design to optimize the number of configurations




(a) NN-PageRank (b) NN-TeraSort (c) NN-Logistic Regression (d) NN-Kmeans (e) NN-Wordcount

(f) RF-PageRank (g) RF-TeraSort (h) RF-Logistic Regression (i) RF-Kmeans (j) RF-Wordcount

(k) RS-PageRank (l) RS-TeraSort (m) RS-Logistic Regression (n) RS-Kmeans (o) RS-Wordcount

Fig. 8. Comparison of different sampling methods for three models: neural network (NN), random forest (RF), and regression spline (RS).

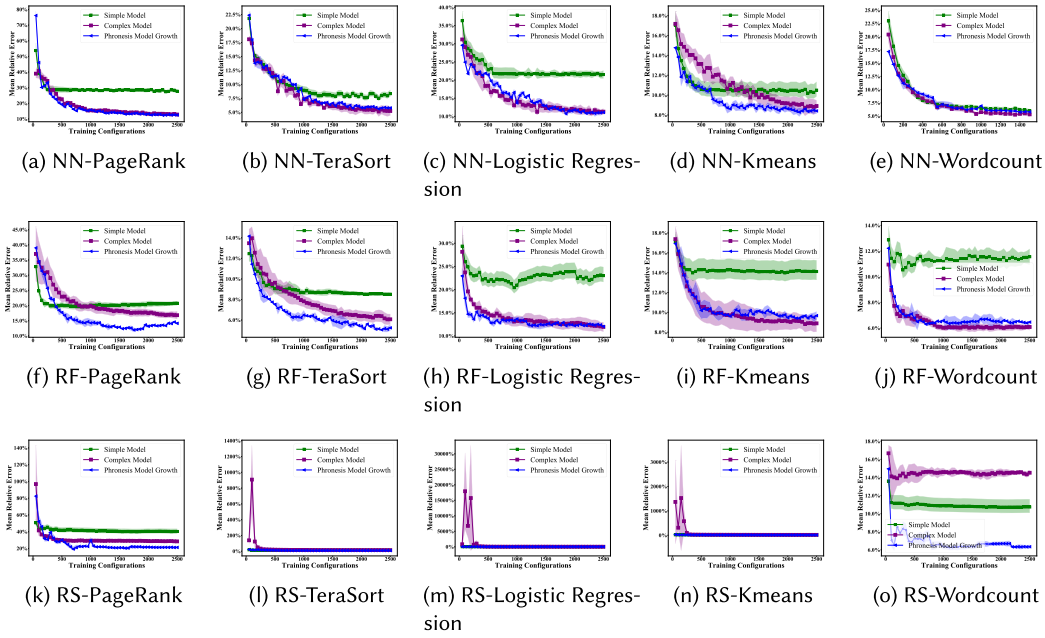
collected to train a linear regression model [63]. d-Simplexed uses Delaunay Triangulation to model the performance function of a Spark workload and adaptively samples to reduce the number of configurations collected for training [23]. These active learning strategies are designed for specific models, linear regression and Delaunay Triangulation. Because neither model can capture complex parameter interactions in high-dimensional configuration spaces effectively, we do not include them in our evaluation.

Figure 8 evaluates sampling methods for varied models and applications. We find the advantage of LHS over random sampling is model dependent. LHS is more efficient for regression splines (Figures 8(k)–8(o)) and is similarly efficient for neural networks and regression splines. This result suggests that maximizing the coverage of sampled configurations cannot consistently improve prediction accuracy across different models in the high-dimensional configuration space.

Second, we find that NIMO is often more accurate than any other method early during the training process. However, its advantage vanishes very quickly as the number of samples increases beyond 50 configurations. In most cases, the accuracy curves flatten out or even go reverse.

NIMO benefits from exploring multiple values of the most important parameters early in the training procedure. Focusing on the most important parameters quickly reduces prediction error. However, it suffers from an increasing number of biased samples generated later in training. As NIMO spends most of its time exploring sub-regions of the space, it generates samples that are biased toward a few parameters and cannot efficiently model the interactions between different parameters as training continues. Despite this disadvantage, NIMO remains valuable for data-efficient training. Phronesis could use NIMO's sampling strategy for an initial training set and then use its active learning strategy for subsequent sampling.

Finally, note that Phronesis's active sampling approach consistently outperforms alternatives for different combinations of models and applications. Phronesis's advantage from active sampling is




(a) NN-PageRank (b) NN-TeraSort (c) NN-Logistic Regression (d) NN-Kmeans (e) NN-Wordcount

(f) RF-PageRank (g) RF-TeraSort (h) RF-Logistic Regression (i) RF-Kmeans (j) RF-Wordcount

(k) RS-PageRank (l) RS-TeraSort (m) RS-Logistic Regression (n) RS-Kmeans (o) RS-Wordcount

Fig. 9. Comparison between dynamic model growth and pre-defined simple/complex models for training neural network (NN), random forest (RF), and regression spline (RS). Models predict execution time for Spark analytic workloads.

more significant when applied to random forest and linear regression, demonstrating that a wide class of models benefit from its active learning strategy.

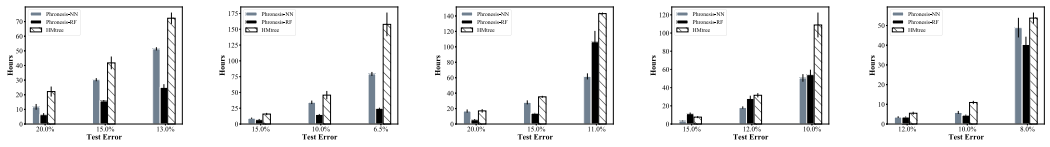
### 5.3 Model Growth Benefit

To better understand the benefits of model growth, we compare dynamic model growth against pre-defined model structures—either simple or complex model—for training. We manually specify simple and complex models for each model class. For the neural network, we specify a small neural network with two layers ((16,8)) and a large neural network with three layers ((512,512,64)). For the random forest, we specify a simple forest with two shallow trees (tree depth = 2) and a large forest with 200 deep trees (tree depth = 20). For regression splines, we specify a simple linear model and a complex polynomial with up to 20 degrees.

Figure 9 illustrates the benefits of model growth over pre-defined model structures. Simple models tend to reach limits at the early stages and their training curves flatten quickly. In contrast, complex models' predictive abilities gradually improve as we accumulate more data, but their accuracies improve slowly when compared against accuracies from dynamic model growth. In addition, a complex model may exhibit strong bias at the beginning of training, leading to extremely high prediction error as shown for regression splines; see Figures 9(l)–9(n).

### 5.4 Reducing Data Costs

Phronesis requires much less data and time to train effective models when compared against state-of-the-art methods. We compare Phronesis against HMTree [72], the state of the art in building performance models for optimizing high-dimensional Spark parameters. HMTree constructs a tiered ensemble of trees such that trees in higher tiers have greater influence on the final output, whereas




(a) PageRank (b) TeraSort (c) Logistic Regression (d) KMeans (e) Wordcount

Fig. 10. Time for collecting data required to achieve acceptable error levels. Bar plots report the average from five repeated measurements, and error bars indicate standard deviation.

our ensemble methods construct models with equal influence on the final output. HMTree collects training configuration randomly and does not apply any strategic data collection technique.

Moreover, HMtrees are affected by three critical hyper-parameters: the number of trees, tree depth, and learning rate. The authors of HMtree determine these parameters by manual tuning with a sufficient amount of training data. We implement HMTree and manually tune hyper-parameters with 3,000 training configurations, constructing 1,000 trees with the maximum tree depth of 10 and learning rate of 0.05.

Figure 10 reports the number of hours required to collect training data for varied methods to reach an error target on the test data. We compare HMTree with Phronesis for neural networks (Phronesis-NN) and random forests (Phronesis-RF) because regression splines cannot reach the reported error target in our training. Both Phronesis-NN and Phronesis-RF outperform HMtree at all error targets. Phronesis's advantages are greater when error targets are lower. Overall, Phronesis reduces time for evaluating Spark configurations for training data by up to 80 hours, a 57% reduction, for an error target, and by 25 hours, a 37% reduction, on average. Note that we consider varied error targets for each application because different applications incur varied costs for evaluating a Spark configuration.

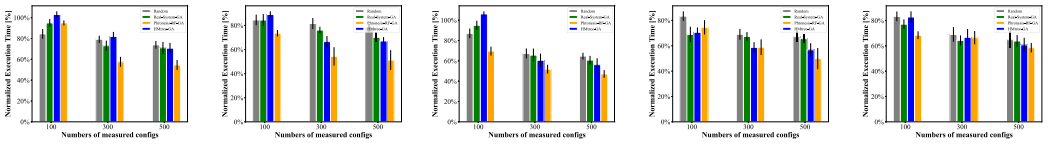
Phronesis outperforms HMtree for two reasons. First, Phronesis actively seeks configurations with high prediction variance, which efficiently reduces output variance of our ensemble models, whereas HMtree randomly selects configurations. When training data is scarce, active learning reduces the accuracy gap between our neural network ensemble and HMtrees.

Second, Phronesis dynamically adjusts model architectures tuning hyperparameters such as the number of neural network layers, tree numbers, and tree depth. With these adjustments, Phronesis finds a good architecture that can best exploit newly acquired data, whereas HMtree requires manual tuning to pre-determine the best hyperparameters. Because the original study does not provide insight into automatically tuning these hyperparameters and brute-force search for the best hyperparameters is slow, adjusting HMtree hyperparameters for new training data is impractical.

### 5.5 Improving Autotuning

Next, we evaluate whether Phronesis improves the efficiency of model-based autotuning methods. We construct a tuning pipeline that incorporates Phronesis and evaluates its efficiency. The pipeline first uses Phronesis to train a random forest that predicts a configuration's expected performance for a given application. Then it applies heuristics to optimize the surrogate model's output and find a good configuration. We evaluate a random forest because its predictions are most accurate for the five Spark applications. We evaluate a **genetic algorithm (GA)** because the heuristic can escape local optima and converge quickly a good configuration [45, 51].

We compare our approach with state-of-the-art, model-based tuning methods for optimizing high-dimensional Spark configurations [72]. This baseline constructs the HMtree model using random sampling and uses GA for optimization. In both model-based tuning apps, the evaluated GA




(a) PageRank (b) TeraSort (c) Logistic Regression (d) KMeans (e) Wordcount

Fig. 11. Optimal configurations identified by four tuning approaches with three different numbers of measured configurations in five repeated measurements. Bar plots report the average from five repeated measurements, and error bars indicate standard deviation.

has population size of 2,000, mutation rate of 0.01, and crossover rate of 0.5. In addition, we compare against two search-based tuning methods: Random sampling and Genetic Algorithm based on actual system evaluations (Real-System-GA). Random sampling randomly picks a set of configurations from the configuration space. Real-System-GA performs genetic algorithms based on the real system evaluation instead of model predictions. For Real-System-GA, we reduce the population size from 2,000 to 20 because it is extremely time-consuming to evaluate 2,000 configurations with real-system execution in one iteration. We use the same mutation rate and crossover rate as in the model-based tuning. These two search-based methods are representative of those in previous frameworks [15, 16, 65]. For example, opentuner [15], one of the most popular frameworks, implements random search and genetic algorithms [6, 15].

We report the best performance of configurations identified by these approaches when these approaches measure 100, 300, and 500 configurations. For each application, we report execution time of the identified configuration normalized to that of the default configuration. To account for randomness, we repeat each method five times and report both average and standard deviation in five experiments.

Figure 11 compares four tuning methods for five applications. Phronesis-RF-GA reduces execution time by an average of 15% and by up to 35% when compared against HMtree-GA and two search-based methods. Phronesis-RF-GA consistently outperforms HMtree-GA when tuning three applications—Pagerank, Terasort, and Logistic Regression—at all three evaluation budgets.

Phronesis-RF-GA benefits from different model training strategies. For Logistic Regression and Terasort, Phronesis-RF-GA's advantage mainly comes from active learning in the early stages of training (i.e., 100 sampled configurations). After profiling only 100 configurations for logistic regression, Phronesis produces a random forest with less than 20% prediction error using active learning. In comparison, HMtree's prediction error is larger than 30%. On the other hand, for Pagerank, Phronesis-RF-GA benefits mostly from dynamic model growth after 300 or 500 configurations have been profiled.

On the other hand, HMtree-GA occasionally achieves comparable or better performance for Kmeans and Wordcount. For Kmeans, HMtree is more accurate up to the first 200 smapled configurations, and this accuracy translates into better performance tuning. However, as the number of sampled configurations increases to 500, Phronesis-RF-GA discovers good random forest structure that is more accurate than HMtree's. For WordCount, Phronesis-RF and HMtree's prediction accuracies are similar at 300 and 500 configurations. The methods' accuracies differ by less than 1% and tuned performance is comparable.

We note that Real-System-GA generally under-performs when compared with model-based tuning methods when the number of measured configurations is greater than 300. At this point, model-based methods start to benefit from improved accuracy derived from more training data. Performance models provide an efficient way for search algorithms to explore the space, and they can have unlimited budgets to explore until they converge to an optimal configuration. In contrast,

$$
\begin{aligned}
y_{pagerank} = & -555.3 \cdot \texttt{default.parallelism}^3 + 524.1 \cdot \texttt{default.parallelism}^4 + 271.1 \cdot \texttt{default.parallelism}^2 \\
& - 184.6 \cdot \texttt{default.parallelism}^5 - 60.8 \cdot \texttt{default.parallelism} + 22 \cdot \texttt{executor.cores}^2 \\
& - 17.7 \cdot \texttt{executor.memory} \cdot \texttt{.executor.cores}^2 - 15.8 \cdot \texttt{executor.cores} \\
& + 13 \cdot \texttt{executor.memory} \cdot \texttt{executor.cores} - 11.2 \cdot \texttt{executor.cores}^2 \cdot \texttt{io.encryption.enabled}
\end{aligned}
$$

$$
\begin{aligned}
y_{sort} = & + 220 \cdot \texttt{io.compression.snappy.blockSize} \cdot \texttt{default.parallelism} \cdot \texttt{task.cpus}^2 \\
& + 42 \cdot \texttt{executor.memory} \cdot \texttt{executor.cores}^2 \cdot \texttt{task.cpus} \\
& - 26 \cdot \texttt{executor.memory} \cdot \texttt{executor.cores}^3 \cdot \texttt{task.cpus} + 22 \cdot \texttt{spark.executor.cores}^3 \cdot \texttt{task.cpus} \\
& - 18.2 \cdot \texttt{executor.memory} \cdot \texttt{executor.cores} \cdot \texttt{task.cpus} \\
& - 17.6 \cdot \texttt{executor.cores}^2 \cdot \texttt{task.cpus} - 14.6 \cdot \texttt{executor.memory}^2 \cdot \texttt{executor.cores} \cdot \texttt{task.cpus} \\
& + 10.5 \cdot \texttt{executor.memory}^2 \cdot \texttt{executor.cores}^2 \cdot \texttt{task.cpus} - 9.7 \cdot \texttt{.executor.cores}^4 \cdot \texttt{task.cpus} \\
& + 8.1 \cdot \texttt{task.cpus}^2
\end{aligned}
$$

Fig. 12. Regression spline models for predicting Pagerank and Terasort performance.

Real-System-GA is limited by the search budget and cannot converge to a final solution with up to 500 explored configurations.

### 5.6 Case Study: Understanding Parameter Interactions

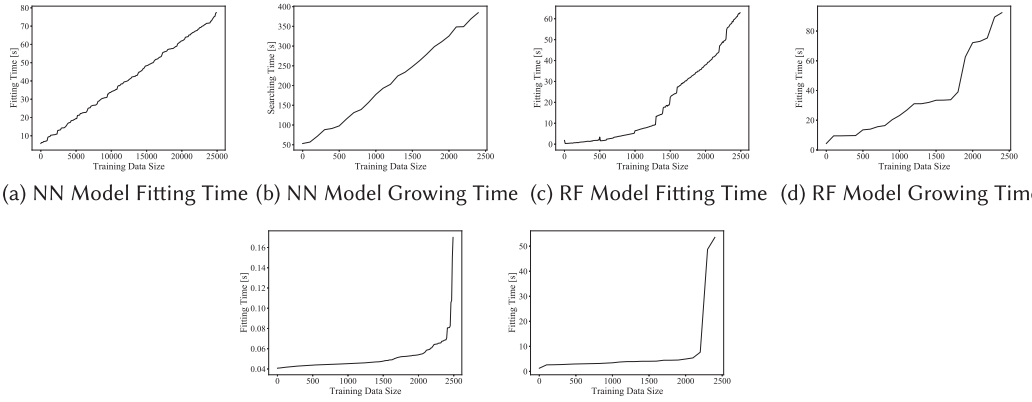
Phronesis supports three kinds of machine learning models. System architects can choose the model class based on their needs. Architects who want to understand parameter interactions could choose regression splines for performance modeling. We show how Phronesis's regression splines reveal non-linear interactions between input parameters.

Figure 12 shows Phronesis's regression splines for predicting Pagerank and TeraSort execution time. We examine individual terms in the regression to understand the impact of each parameter and interactions between parameters. For Pagerank, data parallelism (`default.parallelism`) has the greatest performance impact. This impact is highly non-linear as the spline specifies an exponent of up to 5 for variable `default.parallelism`. For TeraSort, performance is primarily affected by the interactions between I/O compression (`io.compression.snappy.blockSize`), data parallelism (`default.parallelism`), and the number of cores available for each task (`task.cpus`).

The importance of each parameter depends on the program's computational characteristics. Pagerank executes a large number of distributed `join` and `reducebykey` operations, whose performance is sensitive to the number of data partitions (i.e., `default.parallelism`). Terasort is an IO-intensive program that must perform intensive disk reads and writes at the beginning and end of execution. It also must perform large amounts of network IO when shuffling data. Thus, its performance is affected by I/O compression parameters and I/O parallelism, which is determined by `task.cpus`.

### 5.7 Measuring Overheads

**Computational Time Overhead.** We characterize computational time for two key Phronesis components—model fitting and model growth—and calculate the average across five applications and three model classes. We observe a monotonic relationship between the amount of training data and the time spent on these components for three models, as shown in Figure 13. Computational costs are at most a few minutes per training iteration, which is negligible compared to the hours saved in data collection. A third Phronesis component—active sampling—incurs computational costs on the order of 100 ms, which are negligible compared to the time spent fitting and growing models.




(a) NN Model Fitting Time (b) NN Model Growing Time (c) RF Model Fitting Time (d) RF Model Growing Time

(e) RS Model Fitting Time (f) RS Model Growing Time

Fig. 13. Phronesis's computational costs for three models: neural network (NN), random forest (RF), and regression spline (RS).

**Memory Overhead.** We use the `htop` command to monitor Phronesis's memory usage during execution. Profiles indicate Phronesis's peak memory usage is around 1.5GB when it must explore multiple network architectures in parallel. This memory usage comes from storing parameters of different neural network ensembles. On the other hand, the memory usage for training random forest and regression spline is relatively small—normally tens of megabytes.

## 6 RELATED WORK

**Performance Tuning for Data Analytics.** Researchers have sought to automatically tune big data platforms on distributed platforms such as Hadoop and Spark [22, 41, 49, 69, 74, 75]. Spark DAC and following studies auto-tune Spark based on predictive modeling [46, 52, 64, 72]. However, these studies sample randomly for data collection and define model structure by manually tuning model hyperparameters. None of these studies considers data efficiency when training. In this article, we improve data efficiency by co-adjusting data collection and model architecture.

**Active Learning and Efficient Data Acquisition in Systems Management.** Active learning has been used in computer systems research for modeling performance of hardware systems [37], scientific workloads [59], and data analytics [63]. These works build performance models for relatively small design spaces with fewer than 20,000 points [37] or five-dimensional parameter space [59, 63]. Furthermore, these studies neglect the problem of determining the best model with which to apply active learning. Phronesis considers model bias and dynamically adjusts model structure.

**Program Autotuning.** A substantial body of related work has proposed methods for autotuning different types of applications efficiently [13, 15, 40, 50, 53, 62]. Opentuner [15] is a popular framework for auto-tuning generic programs. It supports user-defined configuration space and tunes the program with an ensemble of search-based tuning strategies ranging from simple random sampling to advanced evolutionary algorithms. Furthermore, it utilizes a multi-armed bandit algorithm to determine which tuning strategy gets more budget to explore dynamically. Phronesis can be combined with the Opentuner framework to improve tuning efficiency. For example, we can create a model-based tuning strategy with the Phronesis framework and add it to the ensemble of strategies in Opentune.

Bliss [56], GPtuner [50], and CherryPick [14] apply variants of **Bayesian Optimization (BO)** to guide program tuning. BO is data efficient when optimizing program configurations, but it predicts system outcomes accurately only for near-optimal configurations and fails to provide a global view

of performance across the whole parameter space. On the other hand, Bliss [56] uses an ensemble of BO models with different hyper-parameters for exploring the high-dimensional space and picks the best configuration for exploration during progression. This is another way of constructing ensembles of models and could be considered a future direction for our works.

Recent works tune program performance in heterogeneous systems. HeteroMap [13] builds a performance model for predicting graph analytics performance using combinations of workload features and hardware configuration parameters. Rinnegan [53] tunes resource utilization with models that predict performance from heterogeneous processor configurations. Finally, model-based techniques tune compiler configurations of programs running on Tensor Processing Units [40], predicting performance from compiler configurations and application features. Phronesis complements techniques. Its active learning strategy can generate suitable application-hardware pairs that are likely to improve model accuracy. Furthermore, Phronesis can leverage previously proposed techniques for encoding application features [13, 40].

## 7 CONCLUSION

Phronesis is a modeling framework for predicting performance from system configuration parameters in distributed computing. Phronesis uses data acquisition and dynamic model growth techniques that reduce variance and bias efficiently in predictive modeling. We implement Phronesis and evaluate it for building predictive models for Spark configuration parameters. We show that Phronesis systematically outperforms other common training methods for three classes of machine learning models: neural network, random forest, and regression splines. We demonstrate that Phronesis can significantly reduce data collection time for training predictive Spark models compared to state-of-the-art methods. We also show that Phronesis can improve the tuning efficiency of model-based autotuning methods.

## REFERENCES

[1] Apache software foundation. [n. d.]. Apache Flink 1.11 Documentation: Configuration. Retrieved August 14, 2021, from https://ci.apache.org/projects/flink/flink-docs-stable/ops/config.html.

[2] Apache Spark. [n. d.]. Apache Spark- Unified Analytics Engine for Big Data. Retrieved May 8, 2021, from https://spark.apache.org/sql/.

[3] [n. d.]. Apache Spark Performance Tuning – Degree of Parallelism. Retrieved August 14, 2021, from https://dzone.com/articles/apache-spark-performance-tuning-degree-of-parallel.

[4] Eric Xing and Aarti Singh. [n. d.]. Bias-Variance Tradeoff and Model Selection. Retrieved September 1, 2021, from https://www.ics.uci.edu/~smyth/courses/cs274/readings/xing_singh$_$CMU$_$bias$_$variance.pdf.

[5] Apache Hadoop. [n. d.]. Hadoop Tutorial. Retrieved August 14, 2021, from https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.

[6] [n. d.]. Introduction to Opentuner. Retrieved May 14, 2021, from https://opentuner.org/slides/opentuner-cgo2015-ansel-opentuner-intro.pdf.

[7] Apache Hadoop. [n. d.]. Manhattan Distance. Retrieved August 14, 2021, from https://en.wiktionary.org/wiki/Manhattan_distance.

[8] [n. d.]. Py-earth: Multivariate Adaptive Regression Splines. Retrieved March 30, 2021, from https://contrib.scikit-learn.org/py-earth/.

[9] [n. d.]. scikit-Learn: Machine Learning in Python. Retrieved March 30, 2021, from https://scikit-learn.org/.

[10] Apache Storm. [n. d.]. Storm Configurations. Retrieved August 14, 2021, from https://storm.apache.org/releases/current/Configu-ration.html.

[11] Apache Spark. [n. d.]. Tuning Spark. Retrieved August 14, 2021, from http://spark.apache.org/docs/latest/tuning.html.

[12] Apache Spark. [n. d.]. Tuning Spark. Retrieved August 14, 2021, from https://spark.apache.org/docs/latest/tuning.html.

[13] Masab Ahmad, Halit Dogan, Christopher J. Michael, and Omer Khan. 2019. Heteromap: A runtime performance predictor for efficient processing of graph analytics on heterogeneous multi-accelerators. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'19)*. IEEE, 268–281.

[14] 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)*. 469–482.

[15] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. 303–316.

[16] Jason Ansel, Maciej Pacula, Saman Amarasinghe, and Una-May O'Reilly. 2011. An efficient evolutionary algorithm for solving bottom up problems. In *Annual Conference on Genetic and Evolutionary Computation*, Vol. 10.

[17] Liang Bao, Xin Liu, and Weizhao Chen. 2018. Learning-based automatic parameter tuning for big data analytics frameworks. In *2018 IEEE International Conference on Big Data (Big Data'18)*. IEEE, 181–190.

[18] Zhendong Bei, Zhibin Yu, Huiling Zhang, Wen Xiong, Chengzhong Xu, Lieven Eeckhout, and Shengzhong Feng. 2015. RFHOC: A random-Forest approach to auto-tuning Hadoop's configuration. *IEEE Transactions on Parallel and Distributed Systems* 27, 5 (2015), 1470–1483.

[19] Zhendong Bei, Zhibin Yu, Huiling Zhang, Wen Xiong, Chengzhong Xu, Lieven Eeckhout, and Shengzhong Feng. 2016. RFHOC: A random-forest approach to auto-tuning Hadoop's configuration. *IEEE Transactions on Parallel and Distributed Systems* 27, 5 (2016), 1470–1483.

[20] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[21] Maria Casimiro, Diego Didona, Paolo Romano, Luis Rodrigues, Willy Zwaenepoel, and David Garlan. 2020. Lynceus: Cost-efficient tuning and provisioning of data analytic jobs. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)* IEEE, 56–66. In *The 40th International Conference on Distributed Computing Systems*.

[22] Chi-Ou Chen, Ye-Qi Zhuo, Chao-Chun Yeh, Che-Min Lin, and Shih-Wei Liao. 2015. Machine learning-based configuration parameter tuning on Hadoop system. In *2015 IEEE International Congress on Big Data*. IEEE, 386–392.

[23] Yuxing Chen, Peter Goetsch, Mohammad A. Hoque, Jiaheng Lu, and Sasu Tarkoma. 2019. d-Simplexed: Adaptive Delaunay triangulation for performance modeling and prediction on big data analytics. *IEEE Transactions on Big Data* (2019), 1.

[24] Sasank Chilamkurthy. 2018. *Writing Custom Datasets, Dataloaders and Transforms*. Retrieved July 12, 2021, from https://pytorch.org/tutorials/beginner/data_loading_tutorial.html.

[25] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 153–167.

[26] Balázs Csanád Csáji. 2001. Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary* 24, 48 (2001), 7.

[27] Telmo da Silva Morais. 2015. Survey on frameworks for distributed computing: Hadoop, Spark and storm. In *Proceedings of the 10th Doctoral Symposium in Informatics Engineering (DSIE'15)*, Vol. 15.

[28] Imad Dabbura. 2018. *Coding Neural Network - Parameters' Initialization*. Retrieved July 12, 2021, from https://imaddabbura.github.io/post/coding-nn-params-init/.

[29] Mostafa Ead. 2013. *PStorM: Profile Storage and Matching for Feedback-based Tuning of MapReduce Jobs*. Master's thesis. University of Waterloo.

[30] Jerome H. Friedman. 1991. Multivariate adaptive regression splines. *Annals of Statistics* 19, 1 (1991), 1–67.

[31] Wanling Gao, Jianfeng Zhan, Lei Wang, Chunjie Luo, Daoyi Zheng, Xu Wen, Rui Ren, Chen Zheng, Xiwen He, Hainan Ye, et al. 2018. Bigdatabench: A scalable and unified big data and ai benchmark suite. *arXiv preprint arXiv:1802.08254* (2018).

[32] Yonatan Geifman and Ran El-Yaniv. 2019. Deep active learning with a neural architecture search. In *Advances in Neural Information Processing Systems*. 5976–5986.

[33] Gene H. Golub, Michael Heath, and Grace Wahba. 1979. Generalized cross-validation as a method for choosing a good ridge parameter. *Technometrics* 21, 2 (1979), 215–223.

[34] Anastasios Gounaris and Jordi Torres. 2018. A methodology for Spark parameter tuning. *Big Data Research* 11 (2018), 22–32.

[35] Jake Hoare. 2020. *How Is Splitting Decided for Decision Trees?* Retrieved July 12, 2021, from https://www.displayr.com/how-is-splitting-decided-for-decision-trees/.

[36] Chin-Jung Hsu, Vivek Nair, Vincent W. Freeh, and Tim 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'18)*. IEEE, 660–670.

[37] Engin Ipek, Sally A. McKee, Karan Singh, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. 2008. Efficient architectural design space exploration via predictive modeling. *ACM Transactions on Architecture and Code Optimization (TACO)* 4, 4 (2008), 1.

[38] Muhammad Hussain Iqbal and Tariq Rahim Soomro. 2015. Big data analysis: Apache storm perspective. *International Journal of Computer Trends and Technology* 19, 1 (2015), 9–14.
[39] Selvi Kadirvel and José A. B. Fortes. 2012. Grey-box approach for performance prediction in map-reduce based platforms. In *2012 21st International Conference on Computer Communications and Networks (ICCCN'12)*. IEEE, 1–9.
[40] Sam Kaufman, Phitchaya Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. 2021. A learned performance model for tensor processing units. *Proceedings of Machine Learning and Systems* 3 (2021), 387–400.
[41] Nodira Khoussainova, Magdalena Balazinska, and Dan Suciu. 2012. Perfxplain: Debugging Mapreduce job performance. *Proceedings of the VLDB Endowment* 5, 7 (2012), 598–609.
[42] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
[43] Ron Kohavi. 1995. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, Vol. 14. Montreal, Canada, 1137–1145.
[44] Anita Krishnakumar. 2007. Active learning literature survey. Technical Report. University of California, Santa Cruz.
[45] Manoj Kumar, Mohammad Husain, Naveen Upreti, and Deepti Gupta. 2010. Genetic algorithm: Review and application. *Available at SSRN 3529843* (2010).
[46] Mingyu Li, Zhiqiang Liu, Xuanhua Shi, and Hai Jin. 2020. ATCS: Auto-tuning configurations of big data frameworks based on generative adversarial nets. *IEEE Access* 8 (2020), 50485–50496.
[47] Teng Li, Jian Tang, and Jielong Xu. 2016. Performance modeling and predictive scheduling for distributed stream data processing. *IEEE Transactions on Big Data* 2, 4 (2016), 353–364.
[48] Chieh-Jan Mike Liang, Hui Xue, Mao Yang, Lidong Zhou, Lifei Zhu, Zhao Lucis Li, Zibo Wang, Qi Chen, Quanlu Zhang, Chuanjie Liu, et al. 2020. AutoSys: The design and operation of learning-augmented systems. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC}'20)*. 323–336.
[49] Guangdeng Liao, Kushal Datta, and Theodore L. Willke. 2013. Gunther: Search-based auto-tuning of Mapreduce. In *European Conference on Parallel Processing*. Springer, 406–419.
[50] Yang Liu, Wissam M. Sid-Lakhdar, Osni Marques, Xinran Zhu, Chang Meng, James W. Demmel, and Xiaoye S. Li. 2021. GPTune: Multitask learning for autotuning exascale applications. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 234–246.
[51] T. Meyarivan, Kalyanmoy Deb, Amrit Pratap, and Sameer Agarwal. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197.
[52] Nhan Nguyen, Mohammad Maifi Hasan Khan, and Kewen Wang. 2018. Towards automatic tuning of Apache Spark configuration. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD'18)*. IEEE, 417–425.
[53] Sankaralingam Panneerselvam and Michael Swift. 2016. Rinnegan: Efficient resource use in heterogeneous architectures. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. 373–386.
[54] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).
[55] Panagiotis Petridis, Anastasios Gounaris, and Jordi Torres. 2016. Spark parameter tuning via trial-and-error. In *INNS Conference on Big Data*. Springer, 226–237.
[56] Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, and Devesh Tiwari. 2021. Bliss: Auto-tuning complex applications using a pool of diverse lightweight learning models. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1280–1295.
[57] Burr Settles. 2009. *Active Learning Literature Survey*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences.
[58] Burr Settles, Mark Craven, and Soumya Ray. 2008. Multiple-instance active learning. In *Advances in Neural Information Processing Systems*. 1289–1296.
[59] Piyush Shivam, Shivnath Babu, and J. Chase. 2006. Active sampling for accelerated learning of performance models. In *1st Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SysML'06)*.
[60] Gagan Somashekar and Anshul Gandhi. 2021. Towards optimal configuration of microservices. In *Proceedings of the 1st Workshop on Machine Learning and Systems*. 7–14.
[61] J. Stamper, A. Niculescu-Mizil, S. Ritter, G. J. Gordon, and K. R. Koedinger. 2010. Challenge data set from KDD cup 2010 educational data mining challenge. (2010).
[62] Cristian Tapus, I.-Hsin Chung, and Jeffrey K. Hollingsworth. 2002. Active harmony: Towards automated performance tuning. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC'02)*. IEEE, 44–44.
[63] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI}'16)*. 363–378.

[64] Han Wang, Setareh Rafatirad, and Houman Homayoun. 2019. A+ tuning: Architecture + application auto-tuning for in-memory data-processing frameworks. In *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS'19)*. IEEE, 163–166.

[65] R. Clinton Whaley and Jack J. Dongarra. 1998. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (SC'98)*. IEEE, 38–38.

[66] Tom White. 2012. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc.

[67] Wikipedia. n.d. *Vapnik–Chervonenkis dimension*. Retrieved July 10, 2020, from https://en.wikipedia.org/wiki/Vapnik%E2%80%93Chervonenkis_dimension.

[68] Wikipedia. n.d. *One-hot Encode*. Retrieved July 10, 2020, from https://en.wikipedia.org/wiki/One-hot.

[69] Dili Wu and Aniruddha Gokhale. 2013. A self-tuning system based on application profiling and performance analysis for optimizing Hadoop Mapreduce cluster configuration. In *20th Annual International Conference on High Performance Computing*. IEEE, 89–98.

[70] Joshua J. Yi, David J. Lilja, and Douglas M. Hawkins. 2003. A statistically rigorous approach for improving simulation methodology. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture, 2003 (HPCA-9'03)*. IEEE, 281–291.

[71] Nezih Yigitbasi, Theodore L. Willke, Guangdeng Liao, and Dick Epema. 2013. Towards machine learning-based auto-tuning of Mapreduce. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, 11–20.

[72] Zhibin Yu, Zhendong Bei, and Xuehai Qian. 2018. Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*. 564–577.

[73] 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*. USENIX Association, 2–2.

[74] Zhuoyao Zhang, Ludmila Cherkasova, and Boon Thau Loo. 2013. Autotune: Optimizing execution concurrency and resource usage in Mapreduce workflows. In *Proceedings of the 10th International Conference on Autonomic Computing ({ICAC}'13)*. 175–181.

[75] Zhuoyao Zhang, Ludmila Cherkasova, and Boon Thau Loo. 2014. Optimizing cost and performance trade-offs for Mapreduce job processing in the cloud. In *2014 IEEE Network Operations and Management Symposium (NOMS'14)*. IEEE, 1–8.

[76] 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*. 338–350.

Received October 2021; revised June 2022; accepted June 2022