Noname manuscript No.
(will be inserted by the editor)

Data-Efficient Performance Learning for Configurable Systems

Jianmei Guo[†⋆] · **Dingyu Yang**[†⋆] · **Norbert Siegmund** · **Sven Apel** · **Atrisha Sarkar** · **Pavel Valov** · **Krzysztof Czarnecki** · **Andrzej Wasowski** · **Huiqun Yu**

Received: date / Accepted: date

Abstract Many software systems today are configurable, offering customization of functionality by feature selection. Understanding how performance varies in terms of feature selection is key for selecting appropriate configurations that meet a set of given requirements. Due to a huge configuration space and the possibly high cost of performance measurement, it is usually not feasible to explore the entire configuration space of a configurable system exhaustively. It is thus a major challenge to accurately predict performance based on a small sample of measured system variants. To address this challenge, we propose a *data-efficient* learning approach, called DECART, that combines several techniques of machine learning and statistics for performance prediction of configurable systems. DECART builds, validates, and determines

[†]These authors contributed equally to this work.
[⋆]Corresponding authors.

Jianmei Guo
Alibaba Group, China
E-mail: jianmei.gjm@alibaba-inc.com

Dingyu Yang
Shanghai Dianji University, China
E-mail: yangdy@sdju.edu.cn

Norbert Siegmund
Bauhaus-University Weimar, Germany

Sven Apel
University of Passau, Germany

Atrisha Sarkar · Pavel Valov · Krzysztof Czarnecki
University of Waterloo, Canada

Andrzej Wasowski
IT University of Copenhagen, Denmark

Huiqun Yu
East China University of Science and Technology, China

a prediction model based on an available sample of measured system variants. Empirical results on 10 real-world configurable systems demonstrate the effectiveness and practicality of DECART. In particular, DECART achieves a prediction accuracy of 90% or higher based on a small sample, whose size is linear in the number of features. In addition, we propose a sample quality metric and introduce a quantitative analysis of the quality of a sample for performance prediction.

Keywords Performance prediction · Configurable systems · Regression · Model selection · Parameter tuning

1 Introduction

Many software systems, such as databases, compilers, and Web servers, provide configuration options for stakeholders to tailor the systems' functional behavior and non-functional properties (e.g., performance, cost, and energy consumption). Configuration options relevant to stakeholders are often called *features* [10,2]. Each system variant derived from a configurable software system can be represented as a selection of features, called a *configuration*.

Performance (e.g., response time or throughput) is one of the most important non-functional properties, because it directly affects user perception and cost [45]. Finding an optimal configuration to meet a specific performance goal is often important for developers, system administrators, and users. Naively, one could measure the performance of all configurations of a system and then identify which is the fastest, or build a precise performance model linking feature selection and performance. Unfortunately, this is usually infeasible, as, due to a combinatorial explosion of possible combinations of features, even a small-scale configurable system can have a very large number of configurations. Moreover, measuring the performance of an individual configuration might be time-consuming in itself (e.g., executing a complex benchmark). Measuring many configurations, or measuring slow runs, incur an unacceptable measurement cost.

Typically, only a limited set of configurations can be measured in practice, either by simulation [45] or by monitoring in the field [42]. We call this subset of configurations (along with the corresponding performance measurements) a *sample*, and all configurations of a system (along with their performance values) the *whole population*. We want to *predict* the performance of configurations in the population based on a performance model built by measuring only the sample. Predicting the performance of a new configuration based on a given sample is likely less accurate than directly measuring its performance. A key challenge is to balance measurement effort and prediction accuracy, by using only a *small* sample (for example, a sample with a size that is linear in the number of features) to predict the performance of other configurations in the population with a *high* accuracy (say, above 90 %).

To address this challenge and to hit a sweet spot between measurement effort and prediction accuracy, we aim at a *data-efficient* learning approach, which is recently gaining momentum in the community of machine learning [11]. The key idea is to efficiently reuse available data and then make learning for many small-data problems. This is particularly useful if acquiring data is expensive, such as in personalized healthcare, robot reinforcement learning, sentiment analysis, and community detection [11]. We bring these ideas to performance prediction of configurable systems, proposing *Data-Efficient CART* (DECART), a performance prediction method that suffices with a small sample of measured configurations of a software system and that effectively determines a reasonably accurate prediction model therefrom. DECART works automatically and progressively with random samples, such that one can use it to produce predictions, starting with a small random sample, and subsequently extend it when further measurements are available. DECART combines a previous approach based on plain CART (Classification And Regression Trees) [15] with automated resampling and parameter tuning. Using resampling, DECART learns a prediction model and determines the model's accuracy based on a given sample of measured configurations. Using parameter tuning, DECART ensures that the prediction model has been learned using optimal parameter settings of CART based on the currently available sample.

In summary, we make the following contributions:

– We propose a data-efficient performance learning approach, called DECART, that combines CART with resampling and parameter tuning for performance prediction of configurable systems. DECART builds, validates, and determines a prediction model automatically based on a single given sample. In practice, if there is already a sample available, one can produce predictions directly based on the sample using DECART.
– We demonstrate the effectiveness and practicality of DECART by means of a set of experiments on 10 real-world configurable systems, spanning different domains, with different sizes, different configuration mechanisms, and different implementation languages. The evaluation is based on data from 30 independent experiments. It shows that DECART effectively determines an accurate prediction model based on a small sample and, more importantly, the prediction accuracy estimated based on the sample can represent the *generalized* prediction accuracy of the whole population.
– To evaluate the novel features and innovation of DECART, we compare DECART to previous work based on plain CART [15]. Also, we empirically compare three widely-used resampling techniques (hold-out, cross-validation and bootstrapping) and three parameter-tuning techniques (random search, grid search and Bayesian optimization). Our empirical results demonstrate that DECART produces more accurate predictions than the original CART-based method. Given that a systematic parameter tuning is involved, DECART still works very fast, and the entire process of selecting a prediction model takes, at most, seconds for all subject systems (excluding the time to obtain the sample measurements).
– To explore why DECART works with small random samples, we propose an analytical sample quality metric that quantifies a sample's *goodness of fit* to the whole population and thus provides a quantitative analysis of the

quality of a sample for performance prediction. Previously, this was only surveyed by an empirical analysis of performance distributions [15].
– We have implemented DECART and made the source code of our implementation of DECART publicly available at `http://github.com/jmguo/DECART/`.

This article is based on an earlier conference paper [15]. Compared to that paper, the text and the method have been significantly extended and improved. First, we propose a systematic approach that combines typical CART with resampling and parameter tuning to enable data-efficient performance learning, to reduce measurement effort for validating a performance-prediction model. Second, the previous approach tuned the parameters of CART using a set of empirically-determined rules, which may not work when new subject systems are considered, while DECART adopts systematic and automated resampling and parameter tuning, such that the best prediction model with optimal parameter settings can be selected. Third, previously we only explored why the learning approach works by a comparative analysis of performance distributions, which is empirical and subjective. By contrast, here we employ a quantitative analysis based on a new sample quality metric. Fourth, we extend the previous conference paper's set of subject systems to ten real-world configurable systems and collect results of 30 independent experiments to increase statistical confidence and external validity. Finally, related work and discussion are updated to include the latest progress in this area.

2 Problem Definition

We illustrate the problem of performance prediction of configurable systems using the example of the configurable video codec X264.[1] X264 is a command-line tool to encode video streams into the H.264/MPEG-4 AVC format. For the purpose of the example, we consider 16 encoder features of X264 (e.g., encoding with multiple reference frames or encoding with adaptive spatial transformation). Users can configure X264 by selecting different features to encode a video. After a feature is selected, the corresponding functionality will be activated during the encoding process. We use the *encoding time* to indicate the performance of X264 in different configurations. Even such a simple case with only 16 features gives rise to 1152 configurations in total (Note that not all feature combinations are valid). Table 1 lists a sample of 16 randomly-selected configurations of X264 and corresponding performance measurements. Given such a small sample of measured configurations, how can we determine the performance of other configurations?

Formally, we represent features of a configurable system as a set X of binary decision variables. If a feature is selected in a configuration, then the corresponding variable x is assigned the value 1, and 0 otherwise. The number of all features is denoted N, that is, $X = \{x_1, \ldots, x_N\}$. Let $\mathbf{C}$ be the set of

[1] `http://www.videolan.org/developers/x264.html`

Table 1 A sample of 16 randomly-selected configurations of x264 with corresponding performance measurements in seconds

Conf.	Features																Perf.[s]
$\mathbf{c}_i$	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	p_i
$\mathbf{c}_1$	1	1	0	1	1	1	1	0	1	0	0	1	1	0	0	1	651
$\mathbf{c}_2$	1	1	1	1	1	1	0	1	1	1	0	0	1	0	1	0	536
$\mathbf{c}_3$	1	1	1	1	0	0	0	0	1	1	0	0	1	0	0	1	581
$\mathbf{c}_4$	1	0	0	0	0	0	1	0	1	1	0	0	1	0	1	0	381
$\mathbf{c}_5$	1	1	0	1	0	0	0	1	1	1	0	0	1	0	1	0	424
$\mathbf{c}_6$	1	1	0	0	1	0	1	1	1	1	0	0	1	0	0	1	615
$\mathbf{c}_7$	1	0	1	0	1	1	1	0	1	1	0	0	1	0	1	0	477
$\mathbf{c}_8$	1	0	1	0	0	0	0	1	1	0	0	1	1	1	0	0	263
$\mathbf{c}_9$	1	0	0	0	0	0	1	1	1	0	0	1	1	1	0	0	272
$\mathbf{c}_{10}$	1	1	1	1	0	0	0	1	1	0	0	1	1	1	0	0	247
$\mathbf{c}_{11}$	1	0	0	0	0	0	0	0	1	0	1	0	1	0	0	1	612
$\mathbf{c}_{12}$	1	0	1	1	1	0	0	0	1	0	0	1	1	0	1	0	510
$\mathbf{c}_{13}$	1	1	1	1	0	1	1	0	1	0	1	0	1	0	0	1	555
$\mathbf{c}_{14}$	1	1	0	0	1	0	1	1	1	0	0	1	1	1	0	0	264
$\mathbf{c}_{15}$	1	0	1	0	0	1	1	1	1	0	0	1	1	0	0	1	576
$\mathbf{c}_{16}$	1	0	1	0	1	0	1	1	1	0	1	0	1	1	0	0	268

all valid configurations of the system in question. Each configuration $\mathbf{c}$ of a system is represented as an N-tuple, assigning 1 or 0 to each variable in X. For example, as we can see in Table 1, the x264 system has 16 features and each of its configurations is a 16-tuple; for instance, $\mathbf{c}_1 = (x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 1, \ldots, x_{16} = 1)$.

We assume that a performance benchmark is given, and that we can measure the performance of an arbitrary configuration $\mathbf{c}$ recording its actual performance value y. Note that performance values or measurements we use throughout this paper can be execution time, throughput, workload, access time, response time, and so on, as detailed in Section 5.1. We denote the set $\mathbf{C}$ of all valid configurations, along with their actual performance values Y, as the whole population W of the configurable system. Suppose that we acquire a set of configurations $\mathbf{C}_S \subsetneq \mathbf{C}$ and measure their actual performance Y_S—together forming the sample S (like in Table 1). The set $\mathbf{C}_S$ can be an existing sample already available or be acquired by random sampling. The task is to predict the performance of configurations in $\mathbf{C} \setminus \mathbf{C}_S$ based on the measured sample S.

3 DECART

We now present an overview of DECART, describe the basic principles of the CART-based performance prediction model for configurable systems, and walk through the steps of the DECART method. Finally, we introduce a metric to quantify the quality of a sample used for learning in DECART.

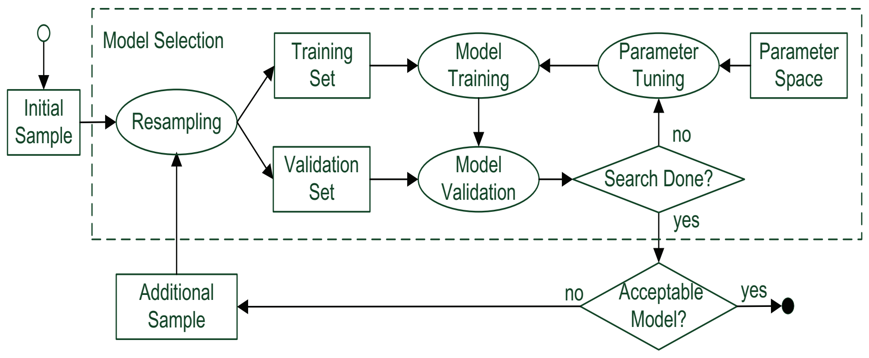

Figure 1 Overview of DECART

3.1 Overview

As shown in Figure 1, DECART begins with an initial sample of measured configurations. Based on this sample, an automated process of model selection is carried out, represented by the dashed box in the figure. When the resulting performance prediction model is acceptable for stakeholders, DECART stops (bottom right). Otherwise, we need to obtain an additional sample (i.e., measured configurations) and iterate again (bottom left) to produce an improved model.

DECART employs a systematic process of *model selection*, that is, an iterative process of model training and validation, mainly based on CART, resampling, and parameter tuning. Resampling partitions the input sample into two sets, one used for learning, the other for validation. CART is used for model training. Parameter tuning is used to explore the parameter space of CART systematically and automatically. Before the search is done, a set of parameter values are used to build candidate prediction models. After exploring the parameter space, the optimal prediction model with the highest prediction accuracy will be selected and presented to the stakeholders.

3.2 CART for Model Training

DECART uses CART as a fundamental learning approach to train a performance prediction model for a given configurable system. We introduce CART briefly by using the motivating example of Table 1. As formalized in Section 2, the prediction problem is to find a function f that predicts the performance value y for a configuration $\mathbf{c}$ based on an input sample S. The basic idea of CART is as follows [6]: Sample S is recursively partitioned into smaller segments until a simple *local* prediction model can be fit into each segment, and then all the local models are organized into a *global* prediction model, which is represented as a binary decision tree.

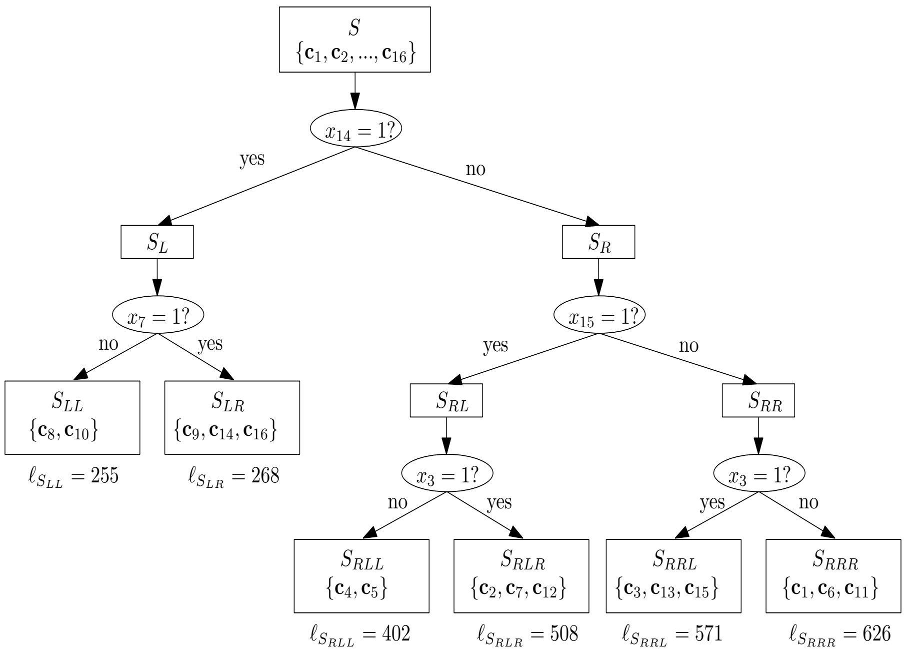
S
$\{\mathbf{c}_1, \mathbf{c}_2, ..., \mathbf{c}_{16}\}$
$x_{14} = 1?$
yes
no
S_L
S_R
$x_7 = 1?$
no
yes
S_{LL}
$\{\mathbf{c}_8, \mathbf{c}_{10}\}$
S_{LR}
$\{\mathbf{c}_9, \mathbf{c}_{14}, \mathbf{c}_{16}\}$
$\ell_{S_{LL}} = 255$
$\ell_{S_{LR}} = 268$
$x_{15} = 1?$
yes
no
S_{RL}
S_{RR}
$x_3 = 1?$
no
yes
$x_3 = 1?$
yes
no
S_{RLL}
$\{\mathbf{c}_4, \mathbf{c}_5\}$
S_{RLR}
$\{\mathbf{c}_2, \mathbf{c}_7, \mathbf{c}_{12}\}$
S_{RRL}
$\{\mathbf{c}_3, \mathbf{c}_{13}, \mathbf{c}_{15}\}$
S_{RRR}
$\{\mathbf{c}_1, \mathbf{c}_6, \mathbf{c}_{11}\}$
$\ell_{S_{RLL}} = 402$
$\ell_{S_{RLR}} = 508$
$\ell_{S_{RRL}} = 571$
$\ell_{S_{RRR}} = 626$

Figure 2 A sample prediction model of x264 learned by CART from the example of Table 1

Figure 2 shows a sample prediction model generated by CART from the example of Table 1. CART starts with the sample S that contains 16 configurations $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_{16}$ and their performance measurements $y_1, y_2, \ldots, y_{16}$. The sample S indicates an input sample, which is either an existing sample already available or is acquired by random sampling. Then, CART partitions the sample S into two segments S_L and S_R by performing a search over all feature-selection variables in X for the best split that minimizes the total prediction errors in its two resulting segments. For example, as shown in Figure 2, the first best split for the x264 sample S is the feature-selection variable x_{14}, because choosing x_{14} to partition S produces the minimal prediction errors in the two resulting segments S_L and S_R. Note that CART does not need any domain knowledge to determine the best split feature. After partitioning, all configurations with $x_{14} = 1$ go to the left segment S_L, and all configurations with $x_{14} = 0$ go to the right segment S_R. Each segment is partitioned recursively by further splits, such as the variables x_7, x_{15}, and x_3.

For each segment S_i, we use the *sample mean* of the actual performance measurements as the *local (prediction) model* of the segment to make prediction fast [5]:

$$\ell_{S_i} = \frac{1}{|S_i|} \sum_{y_j \in S_i} y_j \tag{1}$$

The elements of a local model are the configurations that are contained in the corresponding segment. The local model of each segment identifies the common feature selection (i.e., the selected and deselected features in the corresponding branch from the first split to the current split) and the average performance of the configurations contained in the segment. For example, the local model of the leftmost leaf in Figure 2 indicates the common feature selection ($x_{14} = 1, x_7 = 0$) and the average performance $\ell_{S_{LL}} = \frac{1}{2}(y_8 + y_{10}) = 255$ seconds for the two configurations $\mathbf{c}_8$ and $\mathbf{c}_{10}$.

To penalize the prediction errors in each segment S_i that uses the corresponding local model ℓ_{S_i}, we adopt the most common and convenient loss function, the *sum of squared error* [18]:

$$\sum_{y_j \in S_i} L(y_j, \ell_{S_i}) = \sum_{y_j \in S_i} (y_j - \ell_{S_i})^2 \tag{2}$$

Thus, the best split for each segment S_i is determined to partition S_i into two segments S_{iL} and S_{iR} such that:

$$\sum_{y_j \in S_{iL}} L(y_j, \ell_{S_{iL}}) \ + \sum_{y_j \in S_{iR}} L(y_j, \ell_{S_{iR}}) \ \textit{is minimal} \tag{3}$$

Suppose that there are q leaves in the tree structure of a prediction model. We organize all the local models of these leaves into a global model (i.e., the learned tree model) as follows:

$$f(\mathbf{c}) = \sum_{i=1}^{q} \ell_{S_i} I(\mathbf{c} \in S_i) \tag{4}$$

where $I(\mathbf{c} \in S_i)$ is an indicator function that denotes if configuration $\mathbf{c}$ belongs to a leaf S_i. For example, the final prediction model for Figure 2 is specified as follows:

$$\begin{aligned} f(\mathbf{c}) &= 255 \cdot I(x_{14} = 1, x_7 = 0) \\ &+ 268 \cdot I(x_{14} = 1, x_7 = 1) \\ &+ 402 \cdot I(x_{14} = 0, x_{15} = 1, x_3 = 0) \\ &+ 508 \cdot I(x_{14} = 0, x_{15} = 1, x_3 = 1) \\ &+ 571 \cdot I(x_{14} = 0, x_{15} = 0, x_3 = 1) \\ &+ 626 \cdot I(x_{14} = 0, x_{15} = 0, x_3 = 0) \end{aligned}$$

To determine to which leaf a configuration $\mathbf{c}$ belongs, we match the feature selections of a configuration with the corresponding branch in the tree, from the first split to a leaf. For example, in the tree shown in Figure 2, if a configuration $\mathbf{c}$ satisfies $x_{14} = 1$ and $x_7 = 0$, which is consistent with the feature selections of the leftmost branch, then this configuration falls into the leftmost leaf S_{LL}. Thus, the predicted performance of configuration $\mathbf{c}$ is 255 seconds.

3.3 Initial and Additional Samples

There are three methods to acquire an initial sample of measured configurations: (1) the *feature-size* heuristic randomly selects N configurations as the initial sample, where N equals to the number of features of a configurable system [15,43,48]; (2) the *feature-wise* heuristic specifically selects N_W configurations in which each feature is selected, at least, once [36,37]; and (3) the *feature-frequency* heuristic specifically selects N_F configurations in which each feature is selected and deselected, at least, once [33].

The feature-size heuristic is the easiest to use, since it only requires that a sample reaches a certain size. In contrast, both the feature-wise and feature-frequency heuristic must follow certain feature-coverage criteria by checking predefined constrains. In practice, the configurations that we can measure, or that we already have measured, are often limited and arbitrarily selected; they may not meet any feature-coverage criterion. Moreover, checking feature-coverage criteria can be time-consuming due to the constraints predefined among features [28,34]. To minimize the cost of collecting a sample, DECART adopts the feature-size heuristic to generate an initial sample simply satisfying the size requirement. Also, acquiring additional samples follows the feature-size heuristic, that is, randomly sampling N configurations with their performance values to produce an additional sample.

3.4 Resampling

Our previous approach [15] uses the entire input sample S to train a prediction model and evaluates the model by acquiring an additional set of measured configurations. To reduce measurement effort, DECART aims at training and validating a prediction model based on *only* the input sample. To this end, we use *resampling* to partition the input sample into two data sets: the *training* set is used to build a prediction model, and the *validation* set is used to evaluate the prediction model.

There are three well-established resampling methods to partition a sample S for training and validation [25,13]: *hold-out*, *cross-validation*, and *bootstrapping*. Hold-out partitions an input sample S into two disjoined sets T and V, one for training and the other for validation, that is, $S = T \cup V$ and $T \cap V = \emptyset$. Cross-validation partitions an input sample S into k disjoined subsets of the same size, that is, $S = S_1 \cup \ldots \cup S_k$ and $S_i \cap S_j = \emptyset$ $(i \neq j)$; each subset S_i is selected as the validation set, and all of the remaining $k - 1$ subsets form the training set, together producing k groups of training and validation sets. Bootstrapping relies on random sampling with replacement [13], and its basic workflow is as follows: Given an input sample S with m configurations, bootstrapping randomly selects a configuration b and copies it to another sample S', and then puts b back to S for the next selection; the above process is repeated m times, and thus sample S' contains m configurations. Subsequently, bootstrapping uses S' as the training set and $S \setminus S'$ as the validation set.

We empirically compare the above three methods in Section 5.3 and demonstrate the effectiveness of resampling in our setting.

3.5 Parameter Tuning

We consider three well-established methods of parameter tuning of the underlying learning method: *random search*, *grid search*, and *Bayesian optimization*. Random search is straightforward: Given a parameter space with all possible combinations of parameter values, random search selects a certain set of parameter-value combinations randomly for evaluation. Grid search is an exhaustive search through a manually-specified subset of the parameter space of a learning method [19]. It generates all possible combinations of parameter values, each of which forms a candidate prediction model for evaluation. Bayesian optimization uses a Gaussian Process to model the surrogate function that is used to approximate the true performance function, and it typically optimizes the expected improvement, which is the expected probability that new trials will improve on the current best observation [40].

In general, each parameter-tuning method tries a set of parameter values and produces a group of candidate prediction models for evaluation. As shown in Figure 1, before the search is done, all candidate prediction models will be evaluated and the optimal one will be selected. We empirically compare the three parameter-tuning techniques in Section 5.4 and demonstrate the effectiveness of parameter tuning in our setting.

3.6 Model Validation

A prediction model is usually evaluated in terms of the prediction error rate. Following prior work on performance prediction of configurable systems [36,37, 15,35,43,33,48], the prediction error rate of a prediction model is calculated by the *mean relative error (MRE)*:

$$MRE = \frac{1}{|V|}\sum_{\mathbf{c}\in V}\frac{|actual_{\mathbf{c}} - predicted_{\mathbf{c}}|}{actual_{\mathbf{c}}} \tag{5}$$

where V is the validation set, $actual_{\mathbf{c}}$ indicates the actual performance value of configuration $\mathbf{c}$ in V, and $predicted_{\mathbf{c}}$ is the performance value of configuration $\mathbf{c}$ predicted by the model built. Correspondingly, the *prediction accuracy* is $1 - MRE$.

3.7 Stopping Criteria for Sampling

The *stopping criteria* for sampling are the key to the tradeoff between measurement effort and prediction accuracy. As shown in Figure 1, after the model-selection process searches over the parameter space and produces the optimal prediction model with the lowest prediction error rate, stakeholders must determine whether the prediction error rate of the produced model is acceptable. If the model is still not satisfactory, an additional sample of measured configurations will be collected to build an improved model. According to the theory of learning curves [30], a learning method based on a larger sample usually produces a prediction model with a higher accuracy.

With the aid of resampling, a clear stopping criterion for sampling in DE-CART is the prediction error rate on the validation set, which we call *validation error*. Thus, stakeholders can stop sampling when the calculated validation error is acceptable (e.g., below 10%). Note that the validation error is calculated only based on the small input sample, which may not represent the whole population of all configurations of a system. Moreover, what stakeholders really care about is actually the generalized prediction error rate on new data (i.e., configurations not measured before), which we call *generalization error*. To evaluate the effectiveness of our stopping criteria, we calculate both validation error and generalization error and determine the correlation between them, which we will explore in Section 5.

3.8 Sample Quality Metric

An interesting research question is why a learning approach based on CART works with a small random sample at all. A general explanation from statistical learning theory is that a regression method works well when the problem it addresses or the data it evaluates does fit the regressive pattern it builds [5]. As described in Section 3.2, CART builds a tree-like prediction model that recursively partitions a sample and renders the total prediction errors in each partition minimal; this way, the prediction model always fits the sample well. If the sample can represent the whole population or reflect the important characteristics of the whole population, then the prediction model built on top of the sample also fits the whole population well and makes accurate predictions. To empirically confirm the above analysis, in previous work [15], we conducted a comparative analysis of several empirical performance distributions; we found that CART works well when the sample has a performance distribution similar to the whole population. However, our previous analysis on performance distributions depends on empirical and subjective observations.

Here, we introduce a *quantitative* analysis approach of the quality of a sample. In particular, we propose a sample quality metric to measure the *distance* between the input sample and the whole population. A smaller distance indicates a better sample that represents the whole population more closely.

A major challenge is to find a proper metric to combine heterogeneous variables that have different scales and could give rise to unbalanced domination. For example, feature selections are often encoded as Boolean variables with the domain $\{0, 1\}$ and performance values are numeric with domain $[0, +\infty]$. A straightforward combination of feature selections and performance values usually makes performance values dominate in the final results. However, if we perform a normalization that projects performance values to the domain $[0, 1]$, then feature selections will dominate.

Our sample quality metric aims at measuring a sample's distance or *goodness of fit* to the whole population. It relies on *Pearson's Chi-squared test*.

The key idea is to sum up the differences between *observed* and *expected* outcome *frequencies* in terms of both feature selections and performance values. By frequencies, the proposed metric mitigates the challenge of unbalanced domination when combining heterogeneous variables with different scales.

Given the whole population W and a sample S, the distance between the sample S and the whole population W must consider both feature selections and performance values of configurations. First, we calculate the distance $D_f(S,W)$ between the sample S and the whole population W in terms of feature selections. For each feature x_i ($i = 1 \dots N$), we count the observed (absolute) frequency of feature x_i in sample S and in the whole population W, respectively, denoted as $O^S_{x_i}$ and $O^W_{x_i}$. Then, the probability P_{x_i} of feature x_i appearing in the whole population is calculated by $\frac{O^W_{x_i}}{|W|}$, and the expected frequency $E^S_{x_i}$ of feature x_i in sample S equals to $P_{x_i} \cdot |S|$. Thus, following the idea of the Chi-squared test, the feature-selection distance between S and W is:

$$D_f(S,W) = \sum_{i=1}^{N} \frac{(O^S_{x_i} - E^S_{x_i})^2}{E^S_{x_i}} \tag{6}$$

Intuitively, this distance sums up the differences in the frequencies of feature selections between the sample and the whole population.

Second, we calculate the distance $D_p(S,W)$ between the sample S and the whole population W in terms of performance values. To collect reasonable results for performance values, we convert floating-point numbers to fit a certain precision (e.g., rounding a floating-point number to the nearest integer). Note that the fitting precision depends on the sensitivity of performance prediction, which tolerates noise data to some extent. Thus, we collect all M distinct performance values in the whole population W, that is, $\{y_1, \dots, y_M\}$. Much like as for the feature-selection distance, for each performance value y_j ($j = 1, \dots M$), we count the number of configurations holding performance value y_j in sample S and in the whole population W, respectively, denoted as $O^S_{y_j}$ and $O^W_{y_j}$. The probability of value y_j appearing in the whole population is $P_{y_j} = \frac{O^W_{y_j}}{|W|}$, and the expected frequency of value y_j in sample S is $E^S_{y_j} = P_{y_j} \cdot |S|$. Thus, the performance-value distance between S and W is calculated as follows:

$$D_p(S,W) = \sum_{j=1}^{M} \frac{(O^S_{y_j} - E^S_{y_j})^2}{E^S_{y_j}} \tag{7}$$

Intuitively, the distance sums up the differences in the frequencies of performance values between the sample and the whole population.

Finally, we assume that both feature selections and performance values contribute equally to the sample quality (which can be varied, if desired), so the final distance between the sample S and the whole population W is calculated by averaging the feature-selection distance and the performance-value distance:

$$D(S,W) = \frac{D_f(S,W) + D_p(S,W)}{2} \tag{8}$$

Note that, given two samples S_1 and S_2, we are able to calculate their distances $D(S_1, W)$ and $D(S_2, W)$ to the whole population and thus compare the quality of them.

4 Implementation

We implemented DECART using GNU R, Version 3.4.[2] GNU R is a language and environment for statistical computing and graphics. In particular, we used the R package RPART to implement the underlying CART and to train the performance models.[3] Moreover, we used the R package RBAYESIANOPTIMIZATION to implement Bayesian optimization.[4] There are a set of parameters provided by the RPART package. By fixing other parameters using default settings, we consider three key parameters:

- *minsplit*, an integer parameter, controls the minimum number of configurations that must exist in a tree node for further partitioning.
- *minbucket*, an integer parameter, specifies the minimum number of configurations that must be present in any leaf node.
- *complexity*, a real-value parameter, controls the process of pruning a decision tree, and it is used to control the size of the tree and to select an optimal tree size.

As we aim at learning performance from a small sample of size N, we set the upper bound of *minsplit* to the number of features N. Note that, in the ten subject systems considered in Section 5, N is at most 52. Moreover, we set the lower bound of *minsplit* to the minimum value 1. Thus, the domain of *minsplit*, which theoretically can be any integer, has been significantly reduced to $[1, 52]$. Since *minsplit* and *minbucket* have a strong dependency and usually $\mathit{minbucket} = \frac{1}{3}\mathit{minsplit}$ (as suggested by Williams [46]), we explore only parameter *minsplit* and set *minbucket* automatically in terms of the above equation. The default setting of *complexity* is 0.01. In previous work [15], we set *complexity* to 0.001 and found that a smaller *complexity* works better for a small sample. We delimit the domain of *complexity* to $[10^{-6}, 0.01]$.

Furthermore, testing all possible values of a parameter may not be necessary, since the difference in predictions might be trivial. Therefore, we consider all integer values of *minsplit* and the values of *complexity* with an incremental multiplier of 10. This way, the parameter space Γ is considerably reduced and has $52 \times 5 = 260$ combinations of parameter values in total. Every time the random search is executed, it selects a combination of parameter values randomly from the parameter space Γ. Grid search tries all combinations of parameter values once it is performed. Bayesian optimization works based on a Gaussian Process and performs a complex exploration of the parameter space in terms of the expected probability to find better results. By using the well-established

[2] `http://www.r-project.org/`

[3] `http://cran.r-project.org/web/packages/rpart/`

[4] `https://cran.r-project.org/web/packages/rBayesianOptimization/`

package rBayesianOptimization, we set only the bounds of two parameters *minsplit* and *complexity* and fix other parameters with default settings.

5 Evaluation

We conducted a series of experiments to evaluate DECART and to compare it to the original CART approach [15]. We aim at answering the following research questions:

RQ1: Which resampling technique is the best for DECART?
RQ2: Which parameter-tuning technique is the best for DECART?
RQ3: What is the accuracy and performance of DECART, compared to plain CART?
RQ4: Does the validation error calculated on a sample represent the generalization error on other configurations?
RQ5: Does our proposed quality metric capture the quality of a sample for performance prediction?

5.1 Subject Systems

We selected 10 subject systems to balance several criteria and increase external validity (see Table 2 for an overview). In particular, we aim at covering multiple domains (database systems, Web server, video encoder, compiler, etc.) with different sizes in terms of numbers of features and configurations. Furthermore, we selected systems that are practically relevant (e.g., Berkeley DB, SQLite, Apache Web server) and have a distinct performance characteristic. For example, HIPA^{cc} is an image processing acceleration framework, which runs on 3D hardware whereas SQLite is designed to run also on embedded systems or smart phones. This spread of applications allows us to draw conclusions about the practicality of our approach for real-world systems in heterogeneous domains. Next, we give a short overview of the systems including the benchmark we used to measure performance.

– AJStats is a code-analysis tool for AspectJ programs. It can be customized to collect different statistics, such as the number of aspects and pointcuts. As a benchmark, we analyzed the code basis Orbacus, a customizable CORBA implementation and measured analysis time.
– Apache is a prominent open-source Web server. We used the tools autobench and httperf to generate load on the Web server. We increased the load until the server could not handle any further requests and marked the maximum load as the performance value.
– Berkeley DB C Edition (BDB-C) is an embedded database system, which is one of the most deployed databases in the world, due to its low binary footprint and customizability. We used the benchmark provided by the vendor to measure response time.

Table 2 Overview of the subject systems; LOC: lines of code; N: number of all features; $|W|$: size of the whole population of all configurations

System	Domain	Language	LOC	N	$\|W\|$
AJStats	Code analyzer	C	14 782	19	30 256
Apache	Web server	C	230 277	9	192
BDB-C	Database system	C	219 811	18	2 560
BDB-J	Database system	Java	42 596	26	180
Clasp	Answer set solver	C++	30 871	19	700
HIPA^{cc}	Video processing library	C++	25 605	52	13 485
LLVM	Compiler infrastructure	C++	47 549	11	1 024
lrzip	Compression library	C++	9 132	19	432
SQLite	Database system	C	312 625	39	4 653
x264	Video encoder	C	45 743	16	1 152

- Berkeley DB Java Edition (BDB-J) is a complete re-development in Java with full SQL support. Again, we used a benchmark provided by the vendor measuring response time.
- Clasp is an answer set solver for normal logic programs. We measured solving time for the standard problems provided by this solver's benchmark.
- HIPA^{cc} is a framework accelerating image processing by generating efficient low-level code based on a high-level specification. Our benchmark consisted of a set of partial differential equations that were solved on an nVidia Tesla K20 card with 5GB RAM and 2496 cores.
- LLVM is a compiler infrastructure that supports various configuration options to tailor the compilation process. As benchmark, we measured the time to compile LLVM's test suite.
- LRZIP is a compression library. We used a generator specialized for benchmarking compression algorithms to generate a file of 652MB size and measured the time for compression using different features of the library.
- SQLite is an embedded database system deployed over several millions of devices. It supports several features in form of compiler flags. As benchmark, we used the benchmark provided by the vendor and measured the response time.
- x264 is a video encoder implemented in C. Features adjust the output quality of encoded video files and encoding time. As benchmark, we encoded the Sintel trailer (735 MB) from AVI to the H.264 codec and measured encoding time.

5.2 Experiment Setup

For each experiment, we randomly selected a certain number of configurations, together with corresponding performance measurements, from the whole population W of each subject system as the sample S for model training and validation. We used all the remaining configurations as the testing set G, for calculating the generalization error. That is, $G = W \setminus S$. As prescribed in Section 3.3, the sample size $|S|$ starts with N (i.e., the number of all features of a system) and progressively grows with an increment of N. Given a sample S, we generated a training set T and a validation set V using resampling. For hold-out, we set the partitioning ratio to $7:3$, that is, 70% of the sample S is used as the training set and 30% as the validation set. We adopted the widely-used 10-fold cross-validation, that is, the sample S will be partitioned into 10 subsets of the same size. Each subset is selected as the validation set and all of the remaining subsets form the training set.

As described in Section 3.6, we calculate the error rate of a prediction model in terms of the mean relative error, MRE, defined in equation (5). The validation error is the error rate calculated on a validation set, and the generalization error on a testing set. Correspondingly, the prediction accuracy is $1 - MRE$.

To evaluate the accuracy of DECART in the presence of random samples of different sizes, we conducted experiments using nine different sample sizes from N to $9N$, by an increment of N for each system. There are two exceptions: For BDB-J, the size of the whole population (=180) is less than $7N$ (=196), so we reduced the upper bound of the sample size to $6N$; for HIPA^{cc}, a sample of size $9N$ is still insufficient to achieve a validation error lower than 10%, so we increased the upper bound of the sample size to $20N$. Following equation (5), we calculated both the validation error (based on the validation set) and the generalization error (based on the testing set) for each subject system and each sample size. A lower error rate indicates a higher accuracy.

For comparison, we replicated our previous approach using plain CART with empirically-determined parameter settings [15]: Parameter *complexity* is fixed to 0.001; *minsplit* and *minbucket* are set in terms of a set of empirical rules: If $|S| \leq 100$, then $minbucket = \lfloor \frac{|S|}{10} + \frac{1}{2} \rfloor$ and $minsplit = 2 \cdot minbucket$; if $|S| > 100$, then $minsplit = \lfloor \frac{|S|}{10} + \frac{1}{2} \rfloor$ and $minbucket = \lfloor \frac{minsplit}{2} \rfloor$; the minimum of *minbucket* is 2; and the minimum of *minsplit* is 4. As said previously, the original CART approach uses the entire input sample S as the training set to build a prediction model, and uses the testing set G to calculate the prediction error rate.

In our experiments, the *independent variables* are the choice of the subject system and the size of the input sample. The validation error, generalization error, time cost of model selection, and sample quality metric are the *dependent variables.*

All experiments have been performed on the same Windows 8 machine with Intel Core i7-5600U CPU 2.60 GHZ and 8GB RAM. To reduce fluctuations in the values of dependent variables caused by randomness (e.g., the random generation of input samples and the randomness of measuring the running time), we evaluated each combination of the independent variables 30 times. That is, for each subject system and each sample size, we performed our approach and measured the values of all dependent variables 30 times.

By a large number of iterations of independent experiments, many test statistics are approximately normally distributed according to the central limit theorem. Hence, to increase statistical confidence on the experimental results, we performed a Z-test for each dependent variable at a confidence level of 95%. For brevity, we report only the means and the 95% confidence intervals of the experimental results. Here, the confidence interval is defined as $[\bar{x} - z^* \frac{\sigma}{\sqrt{n}}, \bar{x} + z^* \frac{\sigma}{\sqrt{n}}]$, where $\bar{x}$ and σ are the sample mean and the sample standard deviation of a random variable x, n is the number of tests and $n = 30$ in our experiments, and z^* is the Z-score and $z^* = 1.96$ at the confidence level of 95%.

5.3 Comparison of Three Resampling Techniques

We fixed the parameter tuning to grid search and empirically compared three resampling techniques: hold-out, 10-fold cross-validation, and bootstrapping. For conciseness, Table 3 provides only the subset of experimental results for five sample sizes from N to $5N$.

In terms of the validation error (columns E_V), as shown in Table 3, 10-fold cross-validation outperforms the other two techniques for 8 out of 10 subject systems. But it does not work for Apache when the sample size equals to N, which is 9, since the required fold number is 10.

For Apache, hold-out produces the lowest mean of validation error, but it suffers from a higher margin, which indicates that hold-out does not work stably. In contrast, cross-validation produces a lower mean of validation error than bootstrapping, and it tends to work stably with a small margin of validation error. For SQLite, it is hard to determine which method is definitely the best, since all validation errors produced are close.

In terms of the running time of resampling, hold-out is the fastest. 10-fold cross-validation tends to be faster than bootstrapping but slower than hold-out. All running time involving resampling usually takes seconds.

5.4 Comparison of Three Parameter-Tuning Techniques

We fixed the resampling to 10-fold cross-validation and empirically compared three techniques of parameter tuning, including random search, grid search and Bayesian optimization. Table 4 provides only the subset of experimental results for five sample sizes from N to $5N$.

In terms of the validation error (columns E_V), as shown in Table 4, grid search outperforms the other two techniques for 9 out of 10 systems. Only for SQLite, it is hard to determine which method is definitely the best, since all validation errors produced are close.

In terms of the running time of parameter tuning, random search is clearly the fastest because of its simplicity. Bayesian optimization is the slowest, since it involves a complex process of exploring the parameter space. Grid search is faster than Bayesian optimization but a little bit slower than random search.

Table 3 Experimental results of comparing three resampling techniques in terms of the validation error E_V (%) and the running time (seconds); numbers in bold indicate the lowest mean of validation error; Margin: margin of the 95% confidence interval; '–' indicates an unavailable case

System	$\|S\|$	Bootstrapping				10-fold cross-validation				Hold-out			
		E_V (%)		Time (s)		E_V (%)		Time (s)		E_V (%)		Time (s)	
		Mean	±Margin	Mean	±Margin	Mean	±Margin	Mean	±Margin	Mean	±Margin	Mean	±Margin
AJSTATS	N	2.96	0.58	0.69	0.02	**0.60**	0.22	0.63	0.02	2.16	0.33	0.50	0.03
	$2N$	1.97	0.20	1.32	0.03	**1.49**	0.26	1.20	0.02	1.88	0.24	0.88	0.02
	$3N$	1.92	0.16	1.82	0.03	**1.35**	0.14	1.79	0.03	1.77	0.14	1.33	0.03
	$4N$	1.84	0.07	1.94	0.05	**1.47**	0.16	1.84	0.03	1.87	0.09	1.78	0.03
	$5N$	1.82	0.10	2.09	0.05	**1.49**	0.15	1.88	0.03	1.82	0.10	1.85	0.03
APACHE	N	17.87	3.3	0.22	0.00	–	–	–	–	**6.51**	27.73	0.15	0.00
	$2N$	17.79	3.01	0.45	0.01	5.8	2.30	0.42	0.01	**3.56**	14.95	0.34	0.01
	$3N$	9.36	1.35	0.67	0.00	5.72	1.18	0.65	0.01	**0.91**	10.19	0.47	0.01
	$4N$	8.44	0.61	0.88	0.00	6.40	1.59	0.85	0.01	**0.86**	9.48	0.68	0.02
	$5N$	6.83	0.57	1.14	0.01	5.16	0.61	1.05	0.01	**0.49**	8.83	0.82	0.02
BDB-C	N	89.6	32.25	0.59	0.00	**17.67**	7.84	0.60	0.02	76.38	29.19	0.38	0.01
	$2N$	36.38	8.91	1.21	0.03	**11.00**	5.99	1.08	0.02	27.82	11.46	0.77	0.01
	$3N$	19.04	4.06	1.76	0.04	**6.53**	2.59	1.65	0.03	15.88	6.33	1.15	0.01
	$4N$	11.63	3.12	1.83	0.04	**5.98**	2.28	1.74	0.04	6.32	1.12	1.63	0.02
	$5N$	6.36	1.44	1.87	0.04	**2.89**	0.92	1.83	0.04	5.71	1.14	1.71	0.03
BDB-J	N	4.70	2.33	0.87	0.01	**1.13**	0.29	0.81	0.02	8.05	4.02	0.61	0.01
	$2N$	1.76	0.20	1.80	0.02	**1.26**	0.24	1.67	0.03	1.71	0.17	1.23	0.01
	$3N$	1.79	0.13	1.88	0.03	**1.31**	0.19	1.93	0.02	1.57	0.16	1.8	0.02
	$4N$	1.58	0.12	1.92	0.04	**1.36**	0.22	1.93	0.02	1.66	0.14	1.96	0.05
	$5N$	1.56	0.10	1.95	0.02	**1.07**	0.13	1.99	0.02	1.46	0.13	1.89	0.02
CLASP	N	16.92	3.20	0.66	0.02	**7.03**	4.39	0.69	0.03	18.20	2.94	0.46	0.01
	$2N$	10.38	2.05	1.32	0.02	**5.31**	2.13	1.20	0.01	7.47	1.39	0.89	0.01
	$3N$	5.88	0.90	1.82	0.01	**1.93**	0.51	1.97	0.05	5.35	0.90	1.39	0.02
	$4N$	4.88	0.98	1.91	0.02	**2.05**	0.54	1.96	0.04	4.14	0.74	1.94	0.03
	$5N$	3.40	0.37	2.04	0.07	**2.07**	0.59	1.91	0.01	2.90	0.48	1.94	0.01
HIPA^{cc}	N	17.57	1.28	3.11	0.02	**12.52**	2.37	2.85	0.02	16.08	0.97	2.25	0.04
	$2N$	16.16	1.03	3.46	0.02	**12.94**	1.23	3.49	0.03	15.70	0.98	3.49	0.06
	$3N$	15.73	0.74	3.93	0.03	**11.63**	1.29	4.07	0.19	14.89	0.82	3.67	0.02
	$4N$	14.83	0.67	4.45	0.03	**12.18**	0.85	4.32	0.03	14.69	0.64	4.14	0.10
	$5N$	13.97	0.48	4.98	0.03	**11.72**	0.70	4.79	0.03	13.4	0.54	4.52	0.08
LLVM	N	4.01	0.82	0.31	0.01	**1.81**	0.76	0.29	0.01	4.97	0.71	0.22	0.01
	$2N$	3.85	0.50	0.60	0.01	**2.04**	0.56	0.59	0.01	4.47	0.58	0.45	0.01
	$3N$	3.47	0.38	0.88	0.00	**2.36**	0.58	0.86	0.01	3.19	0.37	0.65	0.01
	$4N$	3.16	0.37	1.2	0.01	**1.74**	0.28	1.09	0.01	2.80	0.26	0.93	0.03
	$5N$	2.66	0.35	1.43	0.01	**1.53**	0.23	1.36	0.01	2.48	0.24	1.12	0.02
LRZIP	N	53.46	17.27	0.68	0.01	**9.65**	6.42	0.66	0.03	48.83	10.78	0.46	0.01
	$2N$	34.46	5.73	1.36	0.01	**12.81**	3.82	1.36	0.03	42.72	10.39	0.89	0.01
	$3N$	32.63	4.11	1.93	0.03	**12.23**	3.36	1.90	0.02	31.21	4.68	1.36	0.01
	$4N$	22.39	3.09	1.97	0.01	**11.57**	3.07	1.99	0.03	21.12	3.24	1.86	0.02
	$5N$	16.86	2.16	2.11	0.03	**6.67**	1.61	1.95	0.01	14.56	2.62	1.89	0.03
SQLITE	N	**4.43**	0.32	1.18	0.01	4.57	0.72	1.09	0.01	4.52	0.33	0.84	0.01
	$2N$	4.58	0.24	1.57	0.01	**4.50**	0.41	1.60	0.01	4.61	0.21	1.63	0.01
	$3N$	4.77	0.16	1.60	0.01	4.71	0.33	1.62	0.01	**4.41**	0.18	1.63	0.01
	$4N$	4.58	0.16	1.63	0.01	4.53	0.30	1.65	0.01	**4.49**	0.15	1.63	0.01
	$5N$	4.54	0.13	1.70	0.04	**4.28**	0.21	1.67	0.01	4.54	0.13	1.7	0.03
x264	N	23.17	6.11	0.51	0.01	**4.27**	2.54	0.45	0.00	16.69	6.63	0.34	0.01
	$2N$	4.97	1.57	1.13	0.03	**2.00**	1.06	0.94	0.02	4.53	3.15	0.67	0.01
	$3N$	2.88	0.88	1.68	0.04	**0.68**	0.39	1.39	0.02	1.96	0.47	1.02	0.01
	$4N$	1.35	0.34	1.75	0.04	**0.72**	0.35	1.72	0.04	1.13	0.30	1.36	0.01
	$5N$	1.21	0.28	1.85	0.03	**0.92**	0.40	1.75	0.03	0.90	0.19	1.64	0.01

Moreover, grid search takes only seconds to finish the search for the optimal parameter values.

5.5 Comparison to Plain CART

We adopted 10-fold cross-validation and grid search for DECART and further compared DECART to our previous approach based on plain CART. Table 5 provides only the subset of results for five sample sizes from N to $5N$. For com-

Table 4 Experimental results of comparing three parameter-tuning techniques in terms of the validation error E_V (%) and the running time (seconds); numbers in bold indicate the lowest mean of validation error; Margin: margin of the 95% confidence interval; '–' indicates an unavailable case

System	$\|S\|$	Bayesian optimization				Grid Search				Random Search			
		E_V (%)		Time (s)		E_V (%)		Time (s)		E_V (%)		Time (s)	
		Mean	±Margin	Mean	±Margin	Mean	±Margin	Mean	±Margin	Mean	±Margin	Mean	±Margin
AJSTATS	N	2.23	0.57	6.01	0.15	**0.60**	0.22	0.63	0.02	5.06	1.81	0.07	0.02
	$2N$	1.98	0.34	3.62	0.74	**1.49**	0.26	1.20	0.02	3.72	0.92	0.08	0.02
	$3N$	2.23	0.27	2.87	0.52	**1.35**	0.14	1.79	0.03	3.96	0.62	0.07	0.02
	$4N$	1.76	0.24	3.47	0.58	**1.47**	0.16	1.84	0.03	4.09	0.50	0.08	0.02
	$5N$	1.64	0.16	3.89	0.62	**1.49**	0.15	1.88	0.03	3.67	0.49	0.08	0.02
APACHE	N	–	–	–	–	–	–	–	–	–	–	–	–
	$2N$	9.43	3.44	6.1	0.49	**5.80**	2.30	0.42	0.01	29.62	4.94	0.02	0.00
	$3N$	6.65	1.41	7.62	0.61	**5.72**	1.18	0.65	0.01	20.12	4.21	0.01	0.00
	$4N$	7.67	0.97	2.80	0.48	**6.40**	1.59	0.85	0.01	29.48	2.85	0.01	0.00
	$5N$	6.76	1.06	5.57	0.19	**5.16**	0.61	1.05	0.01	32.37	2.56	0.01	0.00
BDB-C	N	34.05	10.17	7.09	0.21	**17.67**	7.84	0.60	0.02	483.09	210.08	0.02	0.00
	$2N$	29.39	11.67	2.78	0.46	**11.00**	5.99	1.08	0.02	624.47	148.14	0.02	0.00
	$3N$	28.82	5.23	4.92	0.94	**6.53**	2.59	1.65	0.03	35.37	7.75	0.02	0.00
	$4N$	21.57	3.50	3.94	0.81	**5.98**	2.28	1.74	0.04	73.08	35.56	0.02	0.00
	$5N$	20.30	2.57	4.56	0.80	**2.89**	0.92	1.83	0.04	35.81	3.12	0.02	0.00
BDB-J	N	1.65	0.58	7.60	0.92	**1.13**	0.29	0.81	0.02	26.32	15.74	0.01	0.00
	$2N$	2.40	0.36	2.76	1.04	**1.26**	0.24	1.67	0.03	41.06	14.02	0.02	0.00
	$3N$	1.95	0.35	5.48	0.17	**1.31**	0.19	1.93	0.02	30.17	7.26	0.02	0.00
	$4N$	2.01	0.26	5.24	2.23	**1.36**	0.22	1.93	0.02	11.97	8.51	0.02	0.00
	$5N$	1.96	0.21	5.71	1.61	**1.07**	0.13	1.99	0.02	3.00	0.17	0.02	0.00
CLASP	N	13.18	5.31	5.97	0.12	**7.03**	4.39	0.69	0.03	51.12	12.66	0.02	0.00
	$2N$	8.13	1.92	4.48	0.67	**5.31**	2.13	1.20	0.01	56.60	8.48	0.02	0.00
	$3N$	6.04	0.94	3.40	0.74	**1.93**	0.51	1.97	0.05	31.26	4.46	0.02	0.00
	$4N$	4.58	0.69	3.38	0.42	**2.05**	0.54	1.96	0.04	35.86	5.5	0.02	0.00
	$5N$	5.03	0.60	4.35	0.55	**2.07**	0.59	1.91	0.01	23.27	2.17	0.02	0.00
HIPA^{cc}	N	14.79	1.99	5.04	1.02	**12.52**	2.37	2.85	0.02	20.17	3.12	0.15	0.02
	$2N$	15.51	1.24	6.81	0.88	**12.94**	1.23	3.49	0.03	20.81	1.8	0.14	0.02
	$3N$	15.15	1.42	6.54	0.81	**11.63**	1.29	4.07	0.19	19.5	1.19	0.15	0.02
	$4N$	12.62	0.70	4.29	0.90	**12.18**	0.85	4.32	0.03	17.76	0.93	0.17	0.02
	$5N$	12.95	1.00	6.20	0.95	**11.72**	0.70	4.79	0.03	19.06	1.07	0.12	0.02
LLVM	N	1.94	0.78	5.47	0.95	**1.81**	0.76	0.29	0.01	6.23	1.75	0.01	0.00
	$2N$	3.34	0.88	8.15	0.29	**2.04**	0.56	0.59	0.01	7.18	1.07	0.01	0.00
	$3N$	2.86	0.36	5.41	0.96	**2.36**	0.58	0.86	0.01	4.53	0.65	0.01	0.00
	$4N$	2.08	0.30	3.32	0.74	**1.74**	0.28	1.09	0.01	5.38	0.63	0.01	0.00
	$5N$	2.18	0.35	3.83	0.69	**1.53**	0.23	1.36	0.01	2.61	0.39	0.01	0.00
LRZIP	N	47.69	24.32	5.31	0.12	**9.65**	6.42	0.66	0.03	584.29	175.91	0.01	0.00
	$2N$	31.59	8.35	4.99	0.57	**12.81**	3.82	1.36	0.03	563.61	97.48	0.01	0.00
	$3N$	24.18	4.85	2.82	0.58	**12.23**	3.36	1.90	0.02	154.03	53.97	0.01	0.00
	$4N$	23.23	3.63	3.20	0.32	**11.57**	3.07	1.99	0.03	369.57	83.52	0.01	0.00
	$5N$	26.30	2.50	2.45	0.42	**6.67**	1.61	1.95	0.01	113.9	40.00	0.01	0.00
SQLITE	N	**4.14**	0.98	0.47	0.00	4.57	0.72	1.09	0.01	5.36	1.02	0.02	0.00
	$2N$	**4.29**	0.77	0.47	0.00	4.50	0.41	1.60	0.01	4.65	0.55	0.02	0.00
	$3N$	4.43	0.40	0.47	0.00	4.71	0.33	1.62	0.01	**3.73**	0.34	0.02	0.00
	$4N$	4.58	0.40	0.47	0.00	4.53	0.30	1.65	0.01	**4.46**	0.64	0.02	0.00
	$5N$	4.51	0.24	0.47	0.01	**4.28**	0.21	1.67	0.01	4.33	0.47	0.02	0.00
x264	N	7.34	3.59	6.30	0.13	**4.27**	2.54	0.45	0.00	39.93	8.82	0.01	0.00
	$2N$	4.38	1.57	3.80	0.64	**2.00**	1.06	0.94	0.02	26.33	6.36	0.01	0.00
	$3N$	2.67	0.67	4.51	1.15	**0.68**	0.39	1.39	0.02	23.60	3.42	0.01	0.00
	$4N$	2.35	0.60	7.85	0.40	**0.72**	0.35	1.72	0.04	12.33	1.71	0.02	0.00
	$5N$	2.06	0.22	6.12	0.89	**0.92**	0.40	1.75	0.03	5.83	0.73	0.01	0.00

parison, column E_{CART} lists the prediction error rate of the previous approach using plain CART [15].

Trend. For each system, we observe that the validation error, the generalization error, and the sample quality metric, together with their margins of 95% confidence intervals, decrease progressively when the sample size increases, while the time cost increases slightly and steadily.

Accuracy. When the size of the input sample is N, the validation error and the generalization error, along with their 95% confidence intervals, are located below 10% for 4 subject systems (AJSTATS, BDB-J, LLVM, and SQLITE).

Table 5 Overview of validation errors E_V (%), generalization errors E_G (%), time costs (seconds) of model selection, and sample quality Q_S; E_{CART} (%) denotes the prediction error of using the original CART approach; numbers in bold indicate a lower E_G than E_{CART}; $|S|$: sample size; Margin: margin of the 95% confidence interval; '–' indicates an unavailable case

System	$\|S\|$	E_V (%)		E_G (%)		E_{CART} (%)		Time (s)		Q_S	
		Mean	±Margin	Mean	±Margin	Mean	±Margin	Mean	±Margin	Mean	±Margin
AJSTATS	N	0.60	0.22	3.41	0.49	2.94	0.50	0.63	0.02	1901.12	0.45
	$2N$	1.49	0.26	**2.77**	0.36	3.04	0.38	1.20	0.02	1892.72	0.45
	$3N$	1.35	0.14	**2.43**	0.31	2.72	0.41	1.79	0.03	1886.25	0.65
	$4N$	1.47	0.16	**2.13**	0.24	3.02	0.39	1.84	0.03	1877.07	0.67
	$5N$	1.49	0.15	**1.93**	0.06	2.73	0.34	1.88	0.03	1870.22	0.69
APACHE	N	–	–	–	–	–	–	–	–	–	–
	$2N$	5.80	2.30	14.68	2.32	11.32	1.25	0.42	0.01	20.66	0.38
	$3N$	5.72	1.18	**10.16**	1.04	10.23	1.04	0.65	0.01	18.29	0.44
	$4N$	6.40	1.59	**9.19**	1.06	9.51	0.97	0.85	0.01	16.13	0.43
	$5N$	5.16	0.61	8.99	0.94	8.64	0.74	1.05	0.01	14.57	0.49
BDB-C	N	17.67	7.84	147.65	56.25	123.13	37.55	0.60	0.02	1276.58	0.59
	$2N$	11.00	5.99	**56.84**	30.89	96.89	26.18	1.08	0.02	1268.23	0.59
	$3N$	6.53	2.59	**15.11**	3.54	77.31	19.12	1.65	0.03	1260.82	0.65
	$4N$	5.98	2.28	**8.06**	1.38	74.42	20.35	1.74	0.04	1252.70	0.65
	$5N$	2.89	0.92	**5.24**	0.91	59.92	12.40	1.83	0.04	1244.12	0.62
BDB-J	N	1.13	0.29	**3.22**	0.86	8.82	3.61	0.81	0.02	77.04	0.55
	$2N$	1.26	0.24	**2.07**	0.10	3.67	1.32	1.67	0.03	65.01	0.57
	$3N$	1.31	0.19	**1.85**	0.07	2.95	0.05	1.93	0.02	53.19	0.38
	$4N$	1.36	0.22	**1.68**	0.11	2.96	0.07	1.93	0.02	42.06	0.44
	$5N$	1.07	0.13	**1.67**	0.11	2.86	0.07	1.99	0.02	30.51	0.32
CLASP	N	7.03	4.39	15.33	6.91	22.82	2.36	0.69	0.03	279.00	0.80
	$2N$	5.31	2.13	**8.27**	1.28	17.61	1.00	1.20	0.01	270.32	0.83
	$3N$	1.93	0.51	**4.77**	0.89	18.62	1.17	1.97	0.05	261.11	0.58
	$4N$	2.05	0.54	**3.01**	0.40	18.65	1.13	1.96	0.04	253.37	0.81
	$5N$	2.07	0.59	**2.64**	0.29	17.53	0.93	1.91	0.01	245.34	0.69
HIPA^{cc}	N	12.52	2.37	**20.05**	0.70	21.39	0.72	2.85	0.02	361.50	1.85
	$2N$	12.94	1.23	**18.35**	0.62	20.14	0.39	3.49	0.03	342.35	2.19
	$3N$	11.63	1.29	**16.35**	0.58	19.94	0.38	4.07	0.19	327.14	2.00
	$4N$	12.18	0.85	**15.06**	0.74	19.75	0.33	4.32	0.03	313.87	1.82
	$5N$	11.72	0.70	**14.04**	0.63	19.88	0.25	4.79	0.03	303.98	1.61
LLVM	N	1.81	0.76	5.97	0.24	5.93	0.35	0.29	0.01	497.45	0.52
	$2N$	2.04	0.56	5.04	0.32	4.73	0.30	0.59	0.01	492.08	0.32
	$3N$	2.36	0.58	3.95	0.41	3.63	0.18	0.86	0.01	486.99	0.36
	$4N$	1.74	0.28	**3.18**	0.35	3.42	0.17	1.09	0.01	481.89	0.31
	$5N$	1.53	0.23	**2.59**	0.23	3.74	0.14	1.36	0.01	476.62	0.28
LRZIP	N	9.65	6.42	107.71	40.82	105.16	12.64	0.66	0.03	209.92	0.99
	$2N$	12.81	3.82	**61.14**	20.28	107.18	16.08	1.36	0.03	201.22	0.74
	$3N$	12.23	3.36	**51.83**	27.75	94.48	7.20	1.90	0.02	192.80	0.72
	$4N$	11.57	3.07	**18.87**	3.10	105.04	13.86	1.99	0.03	183.36	0.62
	$5N$	6.67	1.61	**11.72**	2.05	94.16	6.59	1.95	0.01	175.29	0.56
SQLITE	N	4.57	0.72	4.51	0.04	4.51	0.03	1.09	0.01	35.31	18.71
	$2N$	4.50	0.41	4.50	0.03	4.49	0.02	1.60	0.01	21.09	8.16
	$3N$	4.71	0.33	4.52	0.03	4.52	0.02	1.62	0.01	18.13	6.91
	$4N$	4.53	0.30	4.51	0.02	4.51	0.02	1.65	0.01	15.47	3.05
	$5N$	4.28	0.21	4.51	0.02	4.51	0.01	1.67	0.01	17.93	4.34
x264	N	4.27	2.54	**10.28**	2.95	12.30	3.20	0.45	0.00	58.87	0.60
	$2N$	2.00	1.06	**2.71**	0.63	5.87	0.75	0.94	0.02	51.63	0.50
	$3N$	0.68	0.39	**1.73**	0.57	8.26	3.27	1.39	0.02	44.58	0.42
	$4N$	0.72	0.35	**1.16**	0.19	5.31	0.67	1.72	0.04	38.94	0.42
	$5N$	0.92	0.40	**1.11**	0.24	4.31	0.32	1.75	0.03	33.20	0.49

When the sample size increases to $5N$, 8 out of 10 subject systems (except HIPA^{cc} and LRZIP) achieve around 10% or lower validation and generalization errors at a confidence level of 95%.

Furthermore, since stakeholders care more about the prediction capability for the configurations not measured before, we compare the generalization error of DECART to the prediction error rate of using plain CART [15]. Shown in Table 5, we highlight the results of DECART in bold that are better than the original CART approach. When the sample size reaches $5N$, DECART produces a lower error rate than CART for 8 out of 10 subject systems (except Apache and SQLite, in which the results are very close).

Time Cost. For each subject system and each sample size, the time cost of model selection measures the execution time of the entire process of training and validating all candidate prediction models, each of which corresponds to a particular combination of parameter values of the underlying CART. In our implementation described in Section 4, at most 260 prediction models were built and validated based on a given sample for each system. In the end, the best prediction model with the lowest validation error was selected and further used to calculate the generalization error. From Table 5, we observe that the entire process of model selection took less than 5 seconds. In all our experiments, the maximum time cost of model selection was required for HIPA^{cc}, when we increased the sample size to $10N$, and it was only 7.60 ± 0.12 seconds at a confidence level of 95%. Clearly, since DECART incorporates a systematic processes of resampling and parameter tuning, it must be slower than plain CART.

Sample Size. To evaluate the capability of DECART to learn from a small sample, we define an explicit stopping condition for sampling as follows: Sampling stops if the validation error reduces to 10% or below, at a confidence level of 95%. Table 6 lists the experimental results when the above stopping condition was satisfied. We observe that the minimum sample size required for meeting the stopping condition was $10N$ for HIPA^{cc}, $5N$ for lrzip, and, at most, $3N$ for all other 8 subject systems. Moreover, 6 out of all 10 systems require only a small sample of size N.

To better understand the size of the input sample, we calculated the *size ratio* $\frac{|S|}{|W|}$ of the input sample compared to the whole population. In particular, we consider the order of magnitude of the size ratio to make robust comparisons. For example, in the first row of Table 6, the size ratio of AJStats has the order of magnitude 10^{-5}, which indicates that the input sample is about 10^5 times smaller in size than the whole population. From Table 6, we can see that the order of magnitude of the size ratio is 10^{-5} for two systems, 10^{-2} for six systems, and 10^{-1} for two systems. We did not observe a direct correlation between the required sample size and the size ratio, though. For example, both HIPA^{cc} and LLVM have an order of magnitude of size ratio 10^{-2}, but the required sample sizes of them, $10N$ and N, are quite different.

Representativeness of Validation Error. From Table 6, we see that the generalization error is slightly higher than the validation error. Specifically, when the validation error is less than 10%, the generalization error also approximates to 10%. All the observed generalization errors are below 16%, and 6 out of all 10 systems achieve a generalization error around 10% or below.

To disclose the correlation between the validation error and the generalization error, we collected all our experimental results where the sample size

Table 6 Overview of the required sample size, size ratio, validation error E_V (%), and generalization error E_G (%), to achieve a validation error of less than 10%, at a confidence level of 95%

System	$\|S\|$	$\frac{\|S\|}{\|W\|}$	E_V (%)		E_G (%)	
			Mean	±Margin	Mean	±Margin
AJSTATS	N	2×10^{-5}	0.60	0.22	3.41	0.49
APACHE	$2N$	9×10^{-2}	5.80	2.30	14.68	2.32
BDB-C	$3N$	2×10^{-2}	6.53	2.59	15.11	3.54
BDB-J	N	2×10^{-1}	1.13	0.29	3.22	0.86
CLASP	N	3×10^{-2}	7.03	4.39	15.33	6.91
HIPA^{cc}	$10N$	4×10^{-2}	9.46	0.55	10.32	0.42
LLVM	N	1×10^{-2}	1.81	0.76	5.97	0.24
LRZIP	$5N$	2×10^{-1}	6.67	1.61	11.72	2.05
SQLITE	N	1×10^{-5}	4.57	0.72	4.51	0.04
x264	N	1×10^{-2}	4.27	2.54	10.28	2.95

ranges from N to, at most, $20N$ and performed a correlation analysis for each system. The correlation is calculated in terms of two classic *correlation coefficients*: Pearson's r and Spearman's *rho* [16]. Pearson's r is computed on the true values for each variable and evaluates the linear relationship between two variables, while Spearman's *rho* is computed on the ranked values for each variable and evaluates the monotonic relationship between two variables. Table 7 lists the correlation coefficients calculated for each system. According to Salkind's correlation scales [32], we observe that the correlations between the validation error and the generalization error for 7 out of 10 subject systems are strong (coefficients located between 0.6 and 0.8) or very strong (coefficients located between 0.8 and 1.0) in terms of both Pearson's r and Spearman's *rho*. The exceptions include three systems: AJSTATS, BDB-J and SQLITE. These systems tend to produce a very low validation error even using a very small sample of size N, and all the validation errors and generalization errors are very close, which produces significant noise data for correlation analysis.

Sample Quality. From Table 5, we can see that the sample quality metric has the same decreasing trend as the validation and generalization error, when the sample size increases. The sample quality metric measures a sample's distance or goodness of fit to the whole population. Generally, a larger sample implies a better fitness to the whole population. Also, a smaller sample quality metric indicates a better sample that probably produces a higher prediction accuracy. Thus, the sample quality metric decreases when the sample size increases. Moreover, the margin indicates the fluctuation of sample quality metric in 30 independent experiments. A higher margin (e.g., in SQLITE) indicates that the sample is yet insufficient to fit the whole population and the sample quality metric fluctuates with small changes in feature selections and performance values. With the increase of sample size, the sample quality metric decreases and converges to a relatively stable value, and thus the margin decreases as well.

Table 7 Overview of Pearson's r and Spearman's rho of the correlations $Cor(E_V, E_G)$ between validation and generalization errors as well of the correlations $Cor(Q_S, E_G)$ between sample quality and generalization error

System	$Cor(E_V, E_G)$		$Cor(Q_S, E_G)$	
	r	rho	r	rho
AJSTATS	-0.81	-0.47	0.91	1.00
APACHE	0.96	0.90	0.86	1.00
BDB-C	0.96	0.98	0.73	1.00
BDB-J	-0.42	-0.43	0.82	0.94
CLASP	0.92	0.93	0.74	0.98
HIPA^{cc}	0.96	0.98	1.00	1.00
LLVM	0.65	0.87	0.94	1.00
LRZIP	0.61	0.87	0.87	0.98
SQLITE	0.41	0.40	0.34	0.70
X264	0.97	0.85	0.71	0.90

To better understand whether our proposed metric quantifies the quality of the input sample, we analyzed the correlation between the sample quality metric and the generalization error, since the generalization error is the key to the evaluation of sample quality and prediction effects. As presented in Table 7, in terms of Pearson's r, we observed 9 strong or very strong correlations between the sample quality metric and the generalization error. In terms of Spearman's *rho*, the correlation between the sample quality metric and the generalization error is strong or very strong for all 10 subject systems.

5.6 Discussion

Next, we discuss our experimental results and answer the research questions, followed by a discussion of threats to validity and perspectives on our approach.

5.6.1 Research Questions

Regarding RQ1 (resampling), as shown in Table 3, even though 10-fold cross-validation does not work for a very small sample (e.g., the APACHE sample of size N), it outperforms hold-out and bootstrapping for 8 out of 10 systems in terms of the validation error. In terms of the running time, cross-validation is faster than bootstrapping but a little slower than hold-out. Usually, cross-validation takes only seconds for our subject systems. Therefore, according to our experimental results, we recommend 10-fold cross-validation for the resampling of DECART.

Regarding RQ2 (parameter tuning), theoretically, grid search is an exhaustive search that tests all possible combinations of parameter values. In general, grid search is able to find the optimal parameter values if the running time is acceptable. Since our learning method works on a small sample and the parameter space is not very large, grid search is supposed to be feasible and efficient. Our experimental results demonstrated that grid search outperforms random search and Bayesian optimization for 9 out of 10 subject systems on the validation error. Even though grid search is slower than random search, it usually takes only seconds to complete the search. Therefore, from both theoretical and empirical points of view, we recommend grid search for the parameter tuning of DECART.

Regarding RQ3 (prediction accuracy and speed), for all ten subject systems, DECART achieves a prediction accuracy of 90% or higher based on a small sample whose size is linear in the number of features, from N to, at most, $10N$, as shown in Table 6. For 8 out of 10 subject systems, the size of the sample is smaller than $3N$. In particular, for 6 subject systems, even a very small sample of size N is sufficient. Note that the order of magnitude of the size ratio of the sample compared to the whole population is 10^{-5} for 2 systems, 10^{-2} for 6 systems, and 10^{-1} for 2 systems. In a word, DECART reaches a sweet-spot between measurement effort and prediction accuracy.

Compared to our previous approach of using plain CART with empirically-determined parameter settings [15], DECART produces a higher prediction accuracy for 8 out of 10 subject systems and a comparable accuracy for the other two systems, as shown in Table 5.

Clearly, DECART must be slower than plain CART, since it incorporates systematic processes of resampling and parameter tuning. However, DECART still works very fast, and the entire process of model selection takes less than 8 seconds for all subject systems and for the sample size of, at most, $10N$. An important reason for this efficiency is that the input sample is small and thus the domains of some parameters (e.g., *minsplit* and *minbucket*) are significantly limited, which makes the parameter space not very large.

Regarding RQ4 (representativeness of validation error), first, as listed in Table 7, the correlation between the validation error and the generalization error is strong or very strong for 7 out of 10 subject systems. Second, as shown in Table 5, the generalization error is usually higher than the validation error, for each subject system and for each sample size, and their difference steadily reduces when the sample size increases. Third, as shown in Table 6, the validation error approximates to the generalization error, when it reduces to 10% or below. Therefore, we conclude that the validation error calculated on the input sample can represent the generalization error estimated on the whole population very well, especially when the validation error is relatively small (e.g., below 10%).

Regarding RQ5 (sample quality metric), our proposed metric provides an appropriate way to quantify the quality of a sample. First, the metric combines feature selections and performance values properly and mitigates the unbalanced dominance issue caused by the heterogeneous variables with different scales. Second, as shown in Table 7, the metric exhibits a strong or very strong correlation with the generalization error. Since the generalization error is key to evaluate the quality of a sample, we conclude that the metric is able to properly measure sample quality. Furthermore, our sample quality metric provides a quantitative way to compare two samples in terms of their quality or goodness of fit to the whole population. However, a limitation of our sample quality metric is that the whole population has to be acquired as a baseline for the calculation, which is usually infeasible in practice. Hence, currently, our sample quality metric can only be used in empirical studies to help explore why a learning approach works with small random samples.

5.6.2 Multiple Performance Metrics

As described in Section 5.1, each subject system is practically relevant (e.g., Berkeley DB, SQLite, Apache Web server) and has a distinct performance characteristic (e.g., response time or compilation time). In our experimental setting, so far we did not collect multiple performance indicators for each system. Rather, we consider only one performance metric at a time and, thus, build a prediction model for this metric only. When multiple metrics need to be considered, we run DECART multiple times.

Our approach is agnostic to an individual performance metric, because we learn on the pure interval or ratio scaled performance values and do not interpret them. Nevertheless, to demonstrate this ability, we use the recently-proposed tool, THOR, to generate a realistic performance model for a given variability model [38]. The key features of THOR are that (a) variability models are enriched by interactions among features, such that a non-linear and complex performance behavior can be simulated and (b) performance values are generated for features and interactions such that their concrete values are drawn from a value distribution that has been obtained from a real-world software system. This way, THOR generates a ground-truth performance model that can be used for testing.

In this additional experiment, we used THOR to generate for all of our subject systems two performance models for two different metrics: main memory (*Metric_1*) and response time / throughput (*Metric_2*). We also generate interactions based on the number and degree (**p**air-wise, **t**hree-wise, **f**our-wise) of interactions that we found for the corresponding system using an alternative approach based on performance-influence models [35]. Note that, for each subject system, we generated the values of a certain metric using the value distribution of the same metric that has been obtained from a different system. So, the performance values generated here are different from those actually measured for each subject and used in our previous experiments in Section 5. The number and degree of interactions involved for each subject system are listed in Table 8.

In total, we generated 20 ground-truth performance models from which we sample configurations and predict the performance of the remaining configurations based on our approach. We adopted 10-fold cross-validation and grid search for DECART. Moreover, we used exactly the same configurations for sampling when handling each performance metric, so that we can account for the coupling effects among the two different metrics.

As described in Section 5.2, we performed experiments using nine different sample sizes from N to $9N$, by an increment of N for each subject system.

Table 8 The number and degree of interactions involved when generating two different performance metrics for each subject; **p**: pair-wise, **t**: three-wise, **f**: four-wise

System	Interactions	System	Interactions
AJstats	0	Apache	15 (15**p**)
BDB-C	25 (3**p**,6**t**,16**f**)	BDB-J	2 (2**p**)
Clasp	9 (9**p**)	HIPA^{cc}	8 (4**p**, 4**t**)
LLVM	5 (5**p**)	lrzip	18 (18**p**)
SQLite	20 (10**p**, 10**t**)	x264	6 (6**p**)

If a sample of $9N$ is still insufficient to achieve a validation error lower than 10%, we increased the upper bound of the sample size to $20N$. We made the performance models generated by Thor and the results of this experiment publicly available at our project website.[5] Table 9 shows the subset of our experimental results for five sample sizes from N to $5N$.

As shown in Table 9, we observed some fluctuations in the validation errors (e.g., for the *Metric_1* of HIPA^{cc} from N to $2N$), which is normal due to the overtraining on the small validation set. In contrast, for each subject system and each metric, the generalization error decreases progressively when the sample size increases.

Our approach learns a particular performance model for each performance metric. For the two performance models generated by Thor, the first model of *Metric_1* tends to be easier to be learned than that of *Metric_2*. By using a sample of size from N to $5N$, the number of subjects that obtain all validation errors higher than 10% is only three for *Metric_1*, but six for *Metric_2*.

We learned the prediction models for different performance metrics based on the same sample. Our experimental results demonstrate that these models may have different prediction capabilities, perhaps due to the potential interdependencies between different performance metrics. Only for three subjects (BDB-J, Clasp, and SQLite), both the validation errors and the generalization errors of two performance metrics are around 10% or lower. In contrast, for the remaining subjects, the prediction models for the two performance metrics behave quite differently. Take subject AJstats for example, by using exactly the same sample of size N, it achieves a validation error of 1.62 and a generalization error of 4.94 for *Metric_1*, but a validation error of 13.52 and a generalization error of 34.83 for *Metric_2*.

5.6.3 Threats to Validity

To increase internal validity, we implemented our approach to be fully automatic using well-established techniques, including progressive sample generation, resampling, model training, parameter tuning, and model validation. Resampling adopts three well-established techniques, hold-out, 10-fold cross-validation and bootstrapping. By using three well-established methods, includ-

[5] `http://github.com/jmguo/DECART/`

Table 9 Experimental results of validation error E_V (%) and generalization error E_G (%) when predicting two different performance metrics

System	$\|S\|$	*Metric_1*				*Metric_2*			
		E_V (%)		E_G (%)		E_V (%)		E_G (%)	
		Mean	±Margin	Mean	±Margin	Mean	±Margin	Mean	±Margin
AJStats	N	1.62	0.58	4.94	0.34	13.52	6.21	34.83	1.45
	$2N$	2.66	0.40	4.41	0.29	15.37	3.81	30.68	1.58
	$3N$	2.89	0.30	4.02	0.05	13.51	1.45	27.78	1.91
	$4N$	2.76	0.20	3.97	0.06	16.81	2.67	22.88	1.29
	$5N$	2.76	0.22	3.82	0.07	13.78	1.46	21.51	1.36
Apache	N	53.55	15.68	60.90	3.77	66.66	23.82	76.40	5.05
	$2N$	6.77	1.95	37.92	6.10	28.75	16.36	58.77	6.34
	$3N$	10.51	4.44	25.40	5.43	29.05	7.72	51.58	5.30
	$4N$	8.11	1.68	14.89	3.62	21.12	4.44	48.35	6.19
	$5N$	8.44	1.92	10.91	1.12	19.86	2.56	37.04	3.83
BDB-C	N	153.18	120.38	107.34	19.07	14.69	4.90	62.13	3.09
	$2N$	20.38	11.26	57.17	17.04	31.06	7.24	54.66	2.91
	$3N$	13.72	4.63	24.27	8.92	32.84	6.81	51.86	3.77
	$4N$	6.46	1.33	12.17	1.67	27.64	3.37	45.34	2.64
	$5N$	6.39	1.22	7.85	0.98	27.39	3.59	43.20	1.69
BDB-J	N	2.04	0.46	3.81	0.75	8.44	2.11	22.62	5.23
	$2N$	1.44	0.16	2.07	0.14	9.20	1.97	16.17	1.10
	$3N$	1.12	0.18	1.90	0.47	7.78	1.05	12.75	0.79
	$4N$	1.20	0.08	1.40	0.06	7.63	1.17	10.42	0.84
	$5N$	1.06	0.07	1.38	0.09	6.47	0.72	7.91	0.77
Clasp	N	18.16	7.80	41.67	4.03	5.67	2.15	19.65	1.22
	$2N$	11.28	2.75	27.29	4.00	8.44	1.52	16.26	0.54
	$3N$	9.86	2.28	18.77	2.84	9.86	1.30	14.84	0.71
	$4N$	9.73	3.24	14.31	1.31	9.29	1.06	12.61	0.59
	$5N$	8.42	0.87	11.09	0.64	8.35	1.06	12.42	0.59
HIPA^{cc}	N	18.50	2.99	34.98	1.08	14.87	1.61	25.93	1.18
	$2N$	21.97	3.19	30.84	1.21	15.53	1.75	22.92	0.39
	$3N$	20.45	1.93	27.40	1.08	17.92	1.76	21.27	0.43
	$4N$	21.77	1.22	24.59	1.08	15.91	0.94	20.43	0.42
	$5N$	18.98	1.52	22.09	0.95	16.60	1.15	19.70	0.39
LLVM	N	16.66	10.04	47.75	3.37	16.99	7.13	31.84	2.92
	$2N$	13.23	3.51	36.21	2.89	8.19	2.96	21.63	2.57
	$3N$	12.70	2.79	32.39	3.02	9.25	1.54	19.72	1.46
	$4N$	11.59	2.34	20.80	1.62	11.74	2.21	17.96	1.86
	$5N$	12.20	2.70	20.34	2.28	9.46	0.94	14.79	0.51
lrzip	N	14.18	6.88	53.44	15.09	9.56	4.89	28.72	2.18
	$2N$	11.11	3.17	21.25	1.47	12.34	2.09	22.76	1.55
	$3N$	10.75	2.82	19.00	1.71	12.24	1.88	19.08	0.85
	$4N$	9.95	1.68	16.31	0.95	12.05	1.88	18.67	0.95
	$5N$	8.73	1.21	14.11	0.77	11.26	1.92	17.50	0.80
SQLite	N	15.73	3.53	23.13	1.92	11.47	3.14	43.50	4.87
	$2N$	11.32	1.33	17.73	0.94	12.44	2.42	24.25	3.55
	$3N$	11.52	1.21	15.70	0.56	13.74	3.56	19.90	1.66
	$4N$	10.74	0.98	12.96	0.60	10.83	3.09	16.46	1.51
	$5N$	9.37	0.57	11.80	0.40	9.87	1.93	16.28	1.49
x264	N	17.01	10.79	60.68	6.06	14.36	7.44	68.30	9.90
	$2N$	17.60	6.18	46.50	3.51	20.87	11.35	59.65	11.73
	$3N$	18.01	7.09	39.44	3.69	21.87	11.23	36.52	4.74
	$4N$	17.75	6.25	36.55	2.43	13.11	3.30	26.77	3.80
	$5N$	14.31	2.22	30.33	2.90	10.62	2.60	21.33	2.95

ing random search, grid search and Bayesian optimization, parameter tuning incorporates an automated exploration of the parameter space of the learning method. That is, all candidate prediction models, each of which corresponds to a combination of parameter values, have been trained and validated to select the best model. We published the source code of our implementation of DECART for replication.

To avoid misleading effects caused by random fluctuation in measurements, we randomly selected samples of different sizes (N to $20N$) respectively from the whole population of each subject system, and we repeated each random sampling 30 times, with freshly generated samples of the same size as the input of our experiments. By such a large number of iterations of independent experiments, we reported the means and the 95% confidence intervals of all measured dependent variables (validation error, generalization error, time cost, and sample quality) for analysis; thus, we are confident that we controlled the measurement bias sufficiently.

To increase external validity, we evaluated 10 systems spanning different domains, with different sizes, different configuration mechanisms, and different implementation languages. All systems have been deployed and used in real-world scenarios. We measured their performance using standard benchmarks from the respective application domain. However, we are aware that the results of our experiments are not automatically transferable to all other configurable systems, but we are confident that we controlled this threat sufficiently.

5.6.4 Perspectives

Data-efficient learning is gaining momentum in academia and industry, since collecting sufficient and meaningful data is non-trivial in many domains, such as personalized healthcare, robot reinforcement learning, sentiment analysis, and community detection [11]. For configurable systems, the major challenge arises from the huge configuration space and the possibly high cost of performance measurement. DECART provides a data-efficient approach to learn performance from a small sample of measured configurations, which is very useful for performance prediction of configurable systems when the available data are limited.

A distinguished advantage of DECART is that it calculates the validation error based only on a small sample, and the validation error can represent the generalization error calculated on the whole population very well, especially when the validation error is relatively small (e.g., below 10%). This is very important and useful to smartly avoid additional sampling and to determine an accurate prediction model. For example, for many configurable systems (e.g., AJSTATS and BDB-J in Table 5), if we collected a small sample of N measured configurations and already learned a prediction model with a validation error below 10%, then we could infer that the generalization error of the learned model is around 10%; unless we demand a higher accuracy, we could stop additional sampling and determine the prediction model.

DECART still keeps the problem formalization of our previous conference paper [15] that considers only binary features. But we are aware that many practical scenarios contain numeric features [35]. Discretizing numeric features to binary ones is a feasible way, but it might not always be good (e.g., it may introduce more features and make prediction harder). In fact, the underlying learning method CART supports not only binary but also numeric variables. In future, we plan to extend our approach to support numeric features.

DECART considers for each configuration a single value for a single performance measure, which may be execution time, throughput, workload, and response time. However, such a setting might not be enough to conduct a complete performance analysis. Usually, performance analysis should take into account that measures such as response times are non-linear functions of the workload intensity (represented by the arrival rate of requests to the system or the number of simultaneous users served by the system). However, workloads typically get harder on a continuous scale, and it is not clear that a binarization or quantization of workloads into a few levels or categories would allow the detailed enough modeling of performance to be practically useful. In this paper, we keep the workload fixed and learn a non-linear model over the features of the configurable system. To incorporate also the workload variation into the predictions, one could treat this as an orthogonal problem, learning multiple models for different workloads.

6 Related Work

6.1 Model-Based Prediction

Model-based approaches are common for performance prediction [3,1]. For example, linear and multiple regression explore relationships between input parameters and measurements. Furthermore, machine-learning approaches can be used to find the correlation between a configuration and a measurement. CART and its variants, such as Random Forests and Boosting, have been widely used in statistics and data mining, because CART's algorithm is fast and reliable, and its tree structure can provide insights into the relevant input variables for prediction [5,18].

Courtois and Woodside [9] proposed an approach based on regression splines for software performance analysis. Lee et al. [27] compared piecewise polynomial regression and artifical neural networks for the performance modeling of parallel applications. Bu et al. [7] proposed a reinforcement learning approach to the auto-configuration of online Web systems. Thereska et al. [42] proposed a practical performance model based on CART for interactive client applications, such as Microsoft Office and Mozilla. They focused on a range of deployment parameters from the users' application environment, such as CPU speed and memory size; instead, we consider the configuration options of a software system. Moreover, our approach targets all kinds of configurable software systems, as long as the valid configurations can be derived. Westermann et al. [45] presented an approach for the automated improvement of performance-prediction functions by three measurement-point-selection strategies based on the prediction accuracy. They constructed the prediction functions by statistical inference techniques, including CART. Their approach, however, assumes that all input variables of the prediction function are already relevant to performance, while our approach does not have such a restriction, but considers all features of a software system.

In prior work, we reduced the performance-prediction problem to a non-linear regression problem and used CART to address the problem [15,43, 33]. Moreover, we extended our previous method for transfer learning of software performance across different hardware platforms [44]. Unfortunately, this method suffers from several limitations. First, the prediction model is build on a given sample and then its accuracy is validated using an additional sample. This increases measurement effort and possibly requires more measurements than necessary, since the original sample may already be sufficient to build an accurate prediction model. Second, CART and other related learning approaches heavily depend on their own tuning parameters. Previously, we used a set of empirically-determined rules, which may not work when further subject systems are considered. Third, the question of why CART works with small random samples has been addressed only by a comparative analysis of performance distributions, which is empirical and subjective. DECART aims at improving our previous work based on plain CART [15] and overcoming its limitations regarding additional measurement effort, systematic parameter tuning, and quantitative analysis of the sample quality. To make DECART easy to use and easy to understand, we combined CART with only well-established techniques, such as bootstrapping, grid search, and progressive sampling.

Zhang et al. [48,49] proposed a novel performance-prediction approach based on Fourier learning. Their method is the first able to provide theoretical guarantees of accuracy and confidence level for performance prediction of configurable systems. However, it works efficiently only when the performance function of the system is Fourier sparse, which is usually non-trivial to test in practice. In contrast, DECART aims at an easy-to-use performance-prediction approach based on a given sample, without additional effort to understand any particular properties of the system.

Happe et al. [17] proposed a compositional reasoning approach based on component specifications with resource demands and predicted execution time. Their approach is restricted to component-based systems, whereas our approach is applicable to all configurable systems, once their configurable options are exposed as features. Tawhid and Petriu [41] presented a model-driven approach to deriving a performance model from an extended feature model with performance-analysis information. The approach requires detailed up-front knowledge from a domain-specific performance analysis, which makes tuning prediction for accuracy difficult. Our approach avoids these problems by directly working with performance measurements. Ramirez and Cheng [31] presented an approach that leverages goal-based models to facilitate the automatic derivation of utility functions at the requirements level; our approach works at the level of actual system variants.

In addition, there are a number of approaches that use profiling data to create performance models [26]. For example, Jovic et al. analyzed samplings of call stacks of deployed versions of a program to find performance bugs [24]. Grechanik et al. proposed to learn rules for the generation of workloads that reveal program paths with suboptimal performance [14]. However, these approaches concentrate on workload variability rather than software-system configurability. Huang et al. [20] proposed a new lightweight and white-box approach, performance risk analysis (PRA), to improve performance regression testing efficiency via testing target prioritization, while we focus on a prediction method in a black-box fashion.

6.2 Parameter Tuning

Parameter tuning is important for a learning approach to achieve the optimal performance. Experimental designs have been successfully used to tune input parameters in many domains [47,29,22,23]. These approaches, such as SMAC [21], often target a high-dimensional (e.g., hundreds or thousands of parameters) *hyperparameter* space and use different techniques, such as Bayesian optimization [40], deep neural networks [12], and random search [4], to find the best parameter settings efficiently. DECART targets a small input sample and the number of features is, at most, 52 for our subject systems, so the domains of key parameters (such as *minsplit* and *minbucket*) are relatively small and the parameter space is significantly delimited. In such a case, using grid search to carry out an exhaustive search is acceptable for DECART, to achieve reasonable efficiency. Still, modern techniques of hyperparameter optimization and experimental designs are important to explore a parameter space efficiently, and they can be combined with DECART when the target parameter space is considerably huge.

6.3 Measurement-Based Predication

Siegmund et al. [36,37,35] proposed a performance-prediction approach by detecting performance-relevant feature interactions. Following certain feature-coverage criteria, Siegmund's approach selects a specific sample of configurations and then measures their performance, which is then the input for predicting the performance of other configurations. Two fundamental feature-coverage criteria are *feature-wise* and *pair-wise*. The feature-wise criterion quantifies an individual feature's performance influence by calculating the performance delta of two minimal configurations, one with and one without the feature in question. The pair-wise criterion selects and measures additionally a specific set of configurations to detect all pair-wise feature interactions. Also, additional heuristics are provided for detecting higher-order feature interactions [36].

A distinguished advantage of feature-interaction detection is the ability to represent the performance influence of features and feature interactions explicitly, which facilitates program comprehension. Its limitation is the requirement of specific samples of measured configurations, meeting pre-defined coverage criteria, possibly more than needed for plain learning approaches [15]. In practice, the configurations that we can measure, or that we already have measured, are often limited and arbitrarily selected; they may not meet any feature-coverage criterion. Moreover, checking the feature-coverage criteria can be time-consuming due to the constraints predefined among features [28, 34].

Chen et al. [8] combined benchmarking and profiling to predict the performance of component-based applications. In contrast, our approach correlates performance measurements with configurations and can work with any set of configurations measured by simulation or by monitoring in the field. Sincero et al. [39] used existing configurations and measurements to predict a configuration's non-functional properties. They designed the approach called Feedback to find the correlation between feature selections and measurements and to provide qualitative information about how a feature influences a non-functional property during the configuration process. In contrast to our approach, their approach does not actually predict a performance value quantitatively.

7 Conclusion

In this article, we propose a data-efficient performance learning approach, called DECART, that combines CART with systematic resampling and parameter tuning. DECART automatically builds, validates, and determines a prediction model based only on a given small sample of measured configurations, without additional effort to measure more configurations for validation. Moreover, DECART employs systematic resampling and parameter tuning to ensure that the resulting prediction model holds optimal parameter settings based on the currently available sample.

We conducted experiments on 10 real-world configurable systems, spanning different domains, with different sizes, different configuration mechanisms, and different implementation languages. We empirically compared three well-established resampling techniques (hold-out, cross-validation and bootstrapping) and three parameter-tuning techniques (random search, grid search and Bayesian optimization). Our empirical results demonstrate the effectiveness and practicality of DECART. Specifically, DECART achieves a prediction accuracy of 90% or higher based on a small sample whose size is linear in the number of features, from N to, at most, $10N$. For 8 out of 10 subject systems, the size of the input sample is smaller than $3N$. In particular, for 6 subject systems, even a very small sample of size N is sufficient. In addition, DECART is very fast and its entire process of model selection takes seconds, even though a systematic parameter tuning is performed.

DECART avoids additional sampling when the validation error calculated on the input sample is acceptable. Our experiments demonstrate that the validation error represents the generalization error calculated on the whole population very well, especially when it is relatively small (e.g., below 10%). This makes DECART effectively learn an accurate prediction model with as little measurement effort as possible for a given system, such that a sweet spot between measurement effort and prediction accuracy is reached.

Finally, we propose a sample quality metric to measure a sample's goodness of fit to the whole population and introduce a quantitative analysis of the quality of a sample for performance prediction. This complements our previous comparison of performance distributions [15] for understanding why the learning approach works with small random samples. We confirm that the prediction model built on top of a sample makes accurate predictions if the sample fits the whole population well.

In the future, we expect that a combination of DECART and other related techniques, such as feature-interaction detection [36, 35] and experimental design [21, 23], is beneficial to further increase prediction accuracy and reduce prediction effort in wider application domains (e.g., by combining various feature-coverage heuristics to identify suitable samples for learning and by combining various experimental design techniques to quickly find an optimal model when encountering a huge parameter space). Moreover, we plan to extend our approach to support numeric features and multiple performance indicators.

Acknowledgements We would like to thank anonymous reviewers for their insightful comments. This research was partially supported by National Natural Science Foundation of China (No. 61772200, 61702320), Shanghai Pujiang Talent Program (No. 17PJ1401900), Shanghai Municipal Natural Science Foundation (No. 17ZR1406900), Shanghai Municipal Education Commission Funds of Young Teacher Training Program (No. ZZSDJ17021), Specialized Fund of Shanghai Municipal Commission of Economy and Informatization (No. 201602008), Specialized Research Fund for Doctoral Program of Higher Education (No. 20130074110015), the DFG grants (AP 206/4, AP 206/5, AP 206/7, SI 2171/2, and SI 2171/3), Natural Sciences and Engineering Research Council of Canada, and Pratt & Whitney Canada.

References

1. Abdelaziz, A.A., Kadir, W.M.W., Osman, A.: Comparative analysis of software performance prediction approaches in context of component-based system. International Journal of Computer Applications **23**(3), 15–22 (2011)
2. Apel, S., Kästner, C.: An Overview of Feature-Oriented Software Development. Journal of Object Technology **8**(5), 49–84 (2009)
3. Balsamo, S., Marco, A.D., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: A survey. IEEE Trans. Software Eng. **30**(5), 295–310 (2004)
4. Bergstra, J., Bengio, Y.: Random Search for Hyper-Parameter Optimization. Journal of Machine Learning Research **13**(1), 281–305 (2012)
5. Berk, R.: Statistical Learning from a Regression Perspective. Springer (2008)
6. Breiman, L., Friedman, J., Stone, C., Olshen, R.: Classication and Regression Trees. Wadsworth and Brooks (1984)
7. Bu, X., Rao, J., Xu, C.: A reinforcement learning approach to online web systems auto-configuration. In: Proceedings of 29th IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 2–11 (2009)
8. Chen, S., Liu, Y., Gorton, I., Liu, A.: Performance Prediction of Component-Based Applications. Journal of Systems and Software **74**(1), 35–43 (2005)
9. Courtois, M., Woodside, C.M.: Using regression splines for software performance analysis. In: Proceedings of Second International Workshop on Software and Performance, pp. 105–114 (2000)
10. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)

11. Deisenroth, M., Mohamed, S., Doshi-Velez, F., Krause, A., Welling, M.: ICML Workshop on data-efficient machine learning (2016). URL `https://sites.google.com/site/dataefficientml/`
12. Domhan, T., Springenberg, J.T., Hutter, F.: Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In: Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI), pp. 3460–3468 (2015)
13. Efron, B., Tibshirani, R.J.: An Introduction to the Bootstrap. Chapman & Hall (1993)
14. Grechanik, M., Fu, C., Xie, Q.: Automatically finding performance problems with feedback-directed learning software testing. In: Proceedings of International Conference on Software Engineering, pp. 156–166. IEEE (2012)
15. Guo, J., Czarnecki, K., Apel, S., Siegmund, N., Wąsowski, A.: Variability-Aware Performance Prediction: A Statistical Learning Approach. In: Proceedings of International Conference on Automated Software Engineering, pp. 301–311. IEEE (2013)
16. Hand, D.J., Mannila, H., Smyth, P.: Principles of Data Mining. The MIT Press (2001)
17. Happe, J., Koziolek, H., Reussner, R.: Facilitating Performance Predictions Using Software Components. IEEE Software **28**(3), 27–33 (2011)
18. Hastie, T., Tibshirani, R., Friedman, J.: The Elements of Statistical Learning: Data Mining, Inference, and Prediction, second edn. Springer (2009)
19. Hsu, C.W., Chang, C.C., Lin, C.J.: A Practical Guide to Support Vector Classification. Tech. rep., Department of Computer Science, National Taiwan University (2003)
20. Huang, P., Ma, X., Shen, D., Zhou, Y.: Performance regression testing target prioritization via performance risk analysis. In: Proceedings of International Conference on Software Engineering, pp. 60–71. ACM (2014)
21. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential Model-Based Optimization for General Algorithm Configuration. In: Proceedings of International Conference on Learning and Intelligent Optimization, pp. 507–523. Springer (2011)
22. Hutter, F., Xu, L., Hoos, H.H., Leyton-Brown, K.: Algorithm runtime prediction: Methods & evaluation. Artif. Intell. **206**, 79–111 (2014)
23. Jamshidi, P., Casale, G.: An uncertainty-aware approach to optimal configuration of stream processing systems. In: International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, pp. 39–48 (2016)
24. Jovic, M., Adamoli, A., Hauswirth, M.: Catch me if you can: Performance bug detection in the wild. In: Proceedings of International Conference on Object Oriented Programming Systems Languages and Applications, pp. 155–170. ACM (2011)
25. Kohavi, R.: A study of cross-validation and bootstrap for accuracy estimation and model selection. In: Proceedings of International Joint Conference on Artificial Intelligence, pp. 1137–1145. Morgan Kaufmann (1995)
26. Kwon, Y., Lee, S., Yi, H., Kwon, D., Yang, S., Chun, B.G., Huang, L., Maniatis, P., Naik, M., Paek, Y.: Mantis: Automatic performance prediction for smartphone applications. In: Proceedings of the 2013 USENIX Conference on Annual Technical Conference, pp. 297–308. USENIX Association (2013)
27. Lee, B.C., Brooks, D.M., de Supinski, B.R., Schulz, M., Singh, K., McKee, S.A.: Methods of inference and learning for performance modeling of parallel applications. In: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), pp. 249–258 (2007)
28. Nadi, S., Berger, T., Kästner, C., Czarnecki, K.: Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. IEEE Transactions on Software Engineering **41**(8), 820–841 (2015)
29. Osogami, T., Kato, S.: Optimizing system configurations quickly by guessing at the performance. In: Proceedings of International Conference on Measurement and Modeling of Computer Systems, pp. 145–156 (2007)
30. Provost, F.J., Jensen, D., Oates, T.: Efficient Progressive Sampling. In: Proceedings of International Conference on Knowledge Discovery and Data Mining, pp. 23–32. ACM (1999)
31. Ramirez, A., Cheng, B.: Automatic Derivation of Utility Functions for Monitoring Software Requirements. In: Proceedings of International Conference on Model Driven Engineering Languages and Systems. IEEE (2011)

32. Salkind, N.J.: Exploring Research. Prentice Hall PTR (2003)
33. Sarkar, A., Guo, J., Siegmund, N., Apel, S., Czarnecki, K.: Cost-Efficient Sampling for Performance Prediction of Configurable Systems. In: Proceedings of International Conference on Automated Software Engineering, pp. 342–352. IEEE (2015)
34. She, S., Ryssel, U., Andersen, N., Wasowski, A., Czarnecki, K.: Efficient Synthesis of Feature Models. Information & Software Technology **56**(9), 1122–1143 (2014)
35. Siegmund, N., Grebhahn, A., Apel, S., Kästner, C.: Performance-Influence Models for Highly Configurable Systems. In: Proceedings of International Symposium on the Foundations of Software Engineering, pp. 284–294 (2015)
36. Siegmund, N., Kolesnikov, S., Kästner, C., Apel, S., Batory, D., Rosenmüller, M., Saake, G.: Predicting Performance via Automated Feature-Interaction Detection. In: Proceedings of International Conference on Software Engineering. IEEE (2012)
37. Siegmund, N., Rosenmüller, M., Kuhlemann, M., Kästner, C., Apel, S., Saake, G.: SPL Conqueror: Toward Optimization of Non-Functional Properties in Software Product Lines. Software Quality Journal **20**(3-4), 487–517 (2012)
38. Siegmund, N., Sobernig, S., Apel, S.: Attributed Variability Models: Outside the Comfort Zone. In: Proceedings of International Symposium on the Foundations of Software Engineering, pp. 268–278 (2017)
39. Sincero, J., Schröder-Preikschat, W., Spinczyk, O.: Approaching Non-functional Properties of Software Product Lines: Learning from Products. In: Proceedings of Asia-Pacific Software Engineering Conference. IEEE (2010)
40. Snoek, J., Larochelle, H., Adams, R.P.: Practical bayesian optimization of machine learning algorithms. In: Proceedings of 26th Annual Conference on Neural Information Processing Systems (NIPS), pp. 2960–2968 (2012)
41. Tawhid, R., Petriu, D.: Automatic Derivation of a Product Performance Model from a Software Product Line Model. In: Proceedings of International Software Product Line Conference, pp. 80–89. IEEE (2011)
42. Thereska, E., Doebel, B., Zheng, A., Nobel, P.: Practical Performance Models for Complex, Popular Applications. In: Proc. SIGMETRICS, pp. 1–12. ACM (2010)
43. Valov, P., Guo, J., Czarnecki, K.: Empirical Comparison of Regression Methods for Variability-Aware Performance Prediction. In: Proceedings of International Software Product Line Conference, pp. 186–190. ACM (2015)
44. Valov, P., Petkovich, J., Guo, J., Fischmeister, S., Czarnecki, K.: Transferring performance prediction models across different hardware platforms. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE), pp. 39–50 (2017)
45. Westermann, D., Happe, J., Krebs, R., Farahbod, R.: Automated Inference of Goal-Oriented Performance Prediction Functions. In: Proceedings of International Conference on Automated Software Engineering. ACM (2012)
46. Williams, G.: Data Mining with Rattle and R: The Art of Excavating Data for Knowledge Discovery. Springer (2011)
47. Xi, B., Liu, Z., Raghavachari, M., Xia, C.H., Zhang, L.: A smart hill-climbing algorithm for application server configuration. In: Proceedings of International Conference on World Wide Web, pp. 287–296 (2004)
48. Zhang, Y., Guo, J., Blais, E., Czarnecki, K.: Performance Prediction of Configurable Software Systems by Fourier Learning. In: Proceedings of International Conference on Automated Software Engineering, pp. 365–373. IEEE (2015)
49. Zhang, Y., Guo, J., Blais, E., Czarnecki, K., Yu, H.: A mathematical model of performance-relevant feature interactions. In: Proceedings of the 20th International Systems and Software Product Line Conference (SPLC), pp. 25–34 (2016)