The Journal of Systems and Software 209 (2024) 111913

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

RSFIN: A Rule Search-based Fuzzy Inference Network for performance prediction of configurable software systems☆

Yufei Li [a], Liang Bao [a], Kaipeng Huang [a], Chase Wu [b,*], Xinwei Li [a]

[a] School of Computer Science and Technology, Xidian University, Xi'an, 710071, Shaanxi, China
[b] Department of Data Science, New Jersey Institute of Technology, Newark, 07102, NJ, USA

ARTICLE INFO

Keywords:
Configurable software performance prediction
Adaptive network-based fuzzy inference system
Neural architecture search
Entropy

ABSTRACT

Many modern software systems provide numerous configuration options to users and different configurations often lead to different performances. Due to the complex impact of a configuration on the system performance, users have to experimentally evaluate the performance for different configurations. However, it is practically infeasible to exhaust the almost infinite configuration space. To address this issue, various approaches have been proposed for performance prediction based on a limited number of configurations and corresponding performance measurements. Many of such efforts attempt to achieve a reasonable trade-off between experiment effort and prediction accuracy. In this paper, we propose a novel performance prediction model using a Rule Search-based Fuzzy Inference Network (RSFIN) based on ANFIS and NAS. One intuition is that, in systems, similar configurations produce similar performance. We experimentally validate this intuition based on data and introduce a configuration space under entropy. This view suggests the use of RSFIN to capture hidden distributions in configuration space. We implement and evaluate RSFIN using eleven real-world configurable software systems. Experimental results show that RSFIN achieves a better trade-off between measurement effort and prediction accuracy compared to other algorithms. In addition, the results also confirm that the evaluation of configuration space complexity based on data entropy is beneficial.

1. Introduction

Highly configurable software systems, such as databases and web servers, allow users to customize variants of the system by selecting configuration options to achieve desirable functional behaviors. On the other hand, the non-functional properties of different variants may vary, which is one of the key factors for users to select variants in different application scenarios (Pereira et al., 2019). Specifically, users may have completely different non-functional requirements when deploying software systems in different application scenarios, such as mobile devices or desktop computers. For example, a variant for real-time systems must provide a deterministic response time, and a variant for a mobile device requires minimized energy consumption. According to Siegmund et al., under the premise of measurability, non-functional properties are classified as: qualitative properties (security, reliability, etc.), feature-specific quantifiable properties (footprint, maintainability, etc.) and variant-specific quantifiable properties (performance, response time, etc.) (Siegmund et al., 2012b). Among these categories, the first two are considered to be determinable by measuring and inferring about a single configuration option. However, variant-specific properties cannot be determined until they are measured, as the influence of individual configuration options on them cannot be quantified accurately.

User requirements for non-functional properties become a challenge for system development. System development is differentiated between domain engineering and application engineering (Czarnecki et al., 2002), as shown in Fig. 1. Different from the clear common requirements in domain engineering, developers in application engineering need to deal with highly customized requirements from users, which cannot be completely determined in the system development stage. As a result, users or developers often rely on experiences to obtain variants of the requirements, which is inefficient and potentially risky. To address the above challenges, developers must measure the non-functional properties of all candidate variants, which leads to an expensive and time-consuming trial-and-error process, because even a system with only a few configuration options can have millions of possible variants. Therefore, instead of acquiring actual measurements, it is preferred to predict the non-functional properties of different variants using less computational resources.

☆ Editor: Hongyu Zhang.

* Corresponding author.
E-mail addresses: yufeili@stu.xidian.edu.cn (Y. Li), baoliang@mail.xidian.edu.cn (L. Bao), kphuang@stu.xidian.edu.cn (K. Huang), chase.wu@njit.edu (C. Wu), lixinwei@stu.xidian.edu.cn (X. Li).

https://doi.org/10.1016/j.jss.2023.111913
Received 30 May 2023; Received in revised form 9 October 2023; Accepted 20 November 2023
Available online 1 December 2023

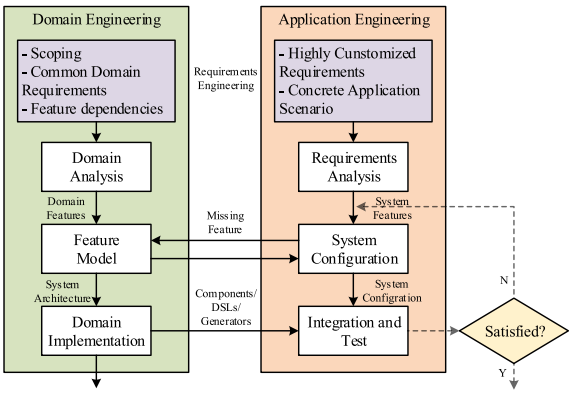


Fig. 1. Domain and application engineering phases in system development (Czarnecki et al., 2002).

For performance prediction, machine learning is an attractive technique due to its powerful feature representation and prediction capabilities (Kotsiantis et al., 2007). Based on the previous research, the work in Pereira et al. (2019) established a four-stage process to tackle this problem: (1) sampling; (2) measuring; (3) learning; and (4) validation, as shown in Fig. 2. The basic idea is to learn from a sample of configurations with performance measurements in the hope of generalizing to the whole configuration space.

Existing machine learning-based algorithms include performance-influence models (Siegmund et al., 2012b,a, 2015), CART-based algorithms (Guo et al., 2013, 2018) and neural network-based algorithms (Ha and Zhang, 2019). These algorithms try to generate an accurate performance prediction model and the key idea is to find a good trade-off between measurement effort and prediction accuracy. Although the previous studies succeeded in their respective contexts, it is still challenging to achieve ideal results on effort and accuracy simultaneously, namely, Pareto optimality (Pardalos et al., 2008). Specifically, performance-influence models (Siegmund et al., 2015) require extensive measurements to support learning, neural network-based algorithms (Ha and Zhang, 2019) achieve high accuracy but with high time cost, and CART-based algorithms (Guo et al., 2018) only tackle systems with binary options and achieve lower accuracy than the work in Ha and Zhang (2019) although with less time cost.

To overcome the shortcomings of the existing methods and optimize the trade-off between effort and accuracy, we propose a Rule Search-based Fuzzy Inference Network (RSFIN)-based performance prediction model for different configurable software systems. In fact, due to the internal constraints of a system configuration, valid configurations do not spread over the entire configuration space. Also, one intuition is that within a system, similar configurations are more likely to yield similar performance. We propose the entropy of the configuration space and validate the above intuition using actual data, as discussed in Section 4. Based on this view, we develop a new performance model that captures hidden distributions in the configuration space and exploits these distributions to optimize predictions. Specifically, we combine Adaptive Network-based Fuzzy Inference System (ANFIS) (Jang, 1993) and Neural Architecture Search (NAS) (Elsken et al., 2019) to meet our design goals.

The adoption of the above technical components is justified as follows. (1) ANFIS can model heterogeneous options and effectively apply prior knowledge (i.e., configuration distribution) to form a network architecture. (2) NAS is a powerful tool for automation architecture engineering that helps the model focus only on the subspace with configuration cluster since it can find the best architecture that describes configuration space. (3) The use of NAS implicitly mitigates the exponential complexity of ANFIS to significantly reduce the size of learning set (Jang and Sun, 1995).

In summary, our research makes the following contributions:

- We construct rigorous cost models, define a formal performance prediction problem for configurable software systems, and provide an innovative perspective for performance prediction based on the view and concept of configuration distribution.
- We introduce configuration space from the perspective of entropy, and then propose a method for evaluating the complexity of system configuration space based on data entropy.
- We design a novel algorithm, RSFIN, which combines ANFIS and NAS, and evaluate its performance using eleven real-world configurable software systems. The experimental results show that RSFIN achieves a better trade-off between measurement effort and prediction accuracy than state-of-the-art approaches.

2. Related work

According to the four-stage process identified in Pereira et al. (2019), performance prediction relies on sampling and learning, and there seems to be no combination that is always superior (Kaltenecker et al., 2020). Therefore, we divide the previous efforts into two categories: sampling strategies and learning algorithms, the latter of which is more relevant to our work because we focus on developing a learning method.

2.1. Sampling strategies

A sampling strategy is to identify valid configurations that conform to the constraints between options, and the goal of sampling is to ensure that the sample configuration is representative of the whole configuration space.

Random sampling is widely used with different notions of randomness (Heradio et al., 2022; Alipourfard et al., 2017; Oh et al., 2017), which aims to randomly choose configurations as part of the samples. Besides, several heuristics have been developed to better cover features and features' interactions as part of the sample. Specifically, the authors of SPL Conqueror (Siegmund et al., 2012b,a, 2015) developed several sampling heuristics to select configurations. Sarkar et al. (2015) used projective sampling based on a novel feature-frequency heuristic. Jamshidi et al. (2018) proposed a sampling strategy, L2S, which extracts transferable knowledge from the source environment to drive the selection of more informative samples in the target environment. Kaltenecker et al. (2019) proposed a distance-based sampling method to generate a sample set to cover the configuration space as uniformly as possible.

2.2. Learning algorithms

In recent years, a plethora of machine learning-based algorithms have been proposed to build a performance prediction model. These algorithms treat performance prediction of individual configurations as a supervised regression problem and can be classified into four categories as follows.

Performance-influence models. Siegmund et al. (2012b,a, 2015) proposed SPL Conqueror to derive a performance-influence model by combining machine learning and sampling heuristics. Its key idea is to quantify the influences of individual configuration options on performance and explore their interactions from the difference between performance measurements. Dorn et al. (2020) developed a Bayesian performance model to explicitly model the uncertainty for option influences, and consequently provide a confidence interval for each performance prediction.

CART-based regression. *Classification And Regression Trees* (CART) (De'ath and Fabricius, 2000) technique is suitable for modeling the correlation between option selections and performance. The seminal work in Guo et al. (2013) applied CART to establish a non-linear model for performance prediction. Guo et al. (2018) extended the previous approach (Guo et al., 2013) with automated resampling and parameter

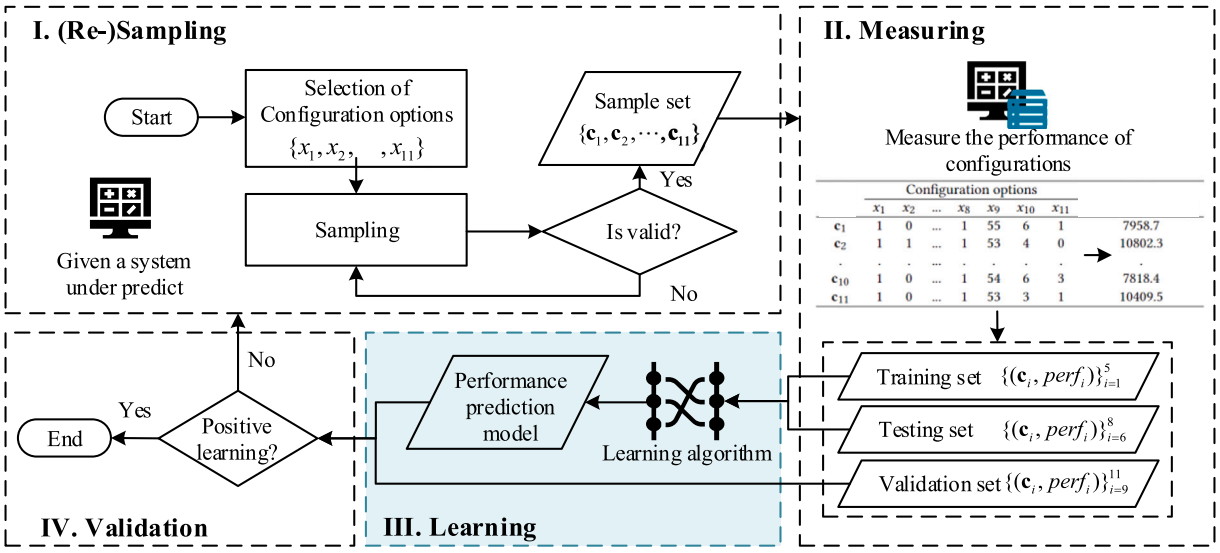


Fig. 2. A four-stage process to predict system performance using machine learning.

tuning techniques, referred to as a data-efficient learning algorithm (DECART) .

Neural network-based regression. More recently, Ha and Zhang (2019) proposed DeepPerf, a deep neural network combined with a sparsity regularization technique, which divides the whole training process into two steps: hyperparameter tuning and training.

Other regression algorithms. Thereska et al. (2010) developed a performance model named AppModel based on regression techniques. Another approach (Zhang et al., 2015) formulated performance prediction as a boolean function based on Fourier transform, and then learned the function via Fourier decomposition. In addition, Valov et al. (2015) proposed some other regression methods such as Bagging, Random Forest, and Support Vector Machine (SVM).

3. Problem statement

In this paper, we study performance prediction for highly configurable software systems. For a given system with limited performance measurements, our goal is to derive a performance model that can accurately predict the performance of unmeasured configurations with a limited cost. To formulate the problem, we first introduce the following concepts.

System Under Prediction. Many configurable software systems such as databases, compilers and web servers, provide a large number of options for users to configure. We refer to such a system as the system under prediction (SUP), denoted as S.

Configuration. We use $C = \{\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_N\}$ to denote the complete set of valid configurations of an SUP, where N is the total number of valid configurations, and $\mathbf{c}_i = (x_1, x_2, \ldots, x_n)$ represents a configuration with n options and x_i $(i = 1, \ldots, n)$ is the ith option, which takes either a binary value (0 or 1 to indicate whether the option is selected) or a numeric value (integers mostly).

Performance. Performance, as a critical non-functional property of an SUP, directly influences user perception and reflects running cost. We treat $Perf(\cdot)$ as a black-box function and use $Perf(S, C)$ to denote the performance set of a given SUP with configuration set C.

Performance Model. We denote a performance model as M generated by a specific algorithm with certain hyperparameters, and use $M(C|D)$ to denote a predicted performance value driven by dataset D.

Loss Function. Given S, C and M, the loss function is represented as $L(M(C), Perf(S, C))$, which implicitly indicates the prediction accuracy (i.e., a smaller value of the loss function means a higher prediction accuracy).

Effort Constraint. In practice, the effort of measurement collection (include all measurements while solving the problem) is often restricted due to the resource constraint and time cost. In this paper, we consider the sample size as this restricted measurement effort, and define it as *effort constraint*, denoted as EC.

Definition 1. For a given system S and measured configuration sets C and C_m, the software system performance prediction is defined as a data-driven model selection problem. Applying C_m to model training, the problem is defined as follow:

$$\begin{aligned} \arg\min_{M} \quad & L(M(C|D), Perf(S, C)) \\ under\ the\ constraint: \quad & C_m \cap C = \emptyset \\ & |C_m| \le EC \\ & D = \{C_m, Perf(S, C_m)\}. \end{aligned} \tag{1}$$

This definition indicates that the goal of this problem is to use limited samples to yield the most accurate model possible.

4. Rule search-based Fuzzy inference network

In this section, we introduce a rule search-based fuzzy inference network (RSFIN) to develop a novel performance model. RSFIN is based on the Adaptive Network-based Fuzzy Inference System (ANFIS) framework, whose key idea is to utilize the entropy of data to search the network architecture so that RSFIN can **capture hidden distributions in the configuration space**. We first present and analyze the configuration space from an entropy perspective, and then provide the design details of RSFIN.

4.1. Observing configuration space from entropy

In order to build a performance prediction model, we intend to start by learning the configuration space. Given configuration options and their optional value set: $x_i \in B_i, i = 1, \ldots, n$, the configuration space is $V = B_1 \times B_2 \times \cdots \times B_n$. However, due to the constraints between

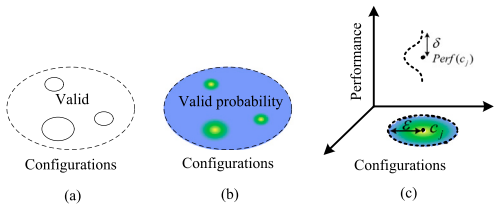


Fig. 3. (a) An abstraction of the configuration space, where the dashed line represent the configuration space and the solid lines represent valid configurations. (b) Configuration space from a probabilistic perspective, where the shade of color represents the probability that the configuration is valid. (c) A schematic illustration of configuration distribution and performance distribution. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

configuration options arising from the system design phase (SPLTBook-sNThesis, 2010), valid configurations do not spread over the entire configuration space, as illustrated in Fig. 3(a). To provide a consistent description of the configuration space, we propose to characterize it using probability distributions. Specifically, we replace the description of configuration validity with probabilities, as shown in Fig. 3(b).

In particular, when given configurations $\mathbf{c}_j, j = 1, \dots, N$ and a slight perturbation ε. It is straightforward to obtain the probability density function $f(\mathbf{c})$ of such configuration distribution, based on multivariate normal distribution (Gut, 2009):

$$f(\mathbf{c}) := \max_{\mathbf{c}_j} G(\mathbf{c}; \mathbf{c}_j, \varepsilon^2 \mathrm{E}), \tag{2}$$

where $G(\cdot|\mathbf{c}_j, \varepsilon^2 E)$ is a multivariate normal probability density function with $\mathbf{c}_j$ as the mean and ε^2E as the covariance matrix.

On the other hand, in order to perform predictions, we also need to understand the rules of configuration space to performance. An easy-to-understand view is that **similar configurations yield similar performances**, which can be expressed in probability as

$$\lim_{|\varepsilon| \to 0} P(|Perf(\mathbf{c}) - Perf(\mathbf{c} + \varepsilon)| > \delta) = 0, \tag{3}$$

where δ is an acceptable performance variation range. Similar views have been mentioned in other work (Bao et al., 2019; Zhu et al., 2017), but no detailed arguments are given. Therefore, in order to fully utilize it in performance prediction, we wish to validate this view based on data. The core of the validation is to measure fluctuations in the performance of similar configurations, which inspires us to consider entropy in information theory (Shannon, 1948), which measures the uncertainty of information. Specifically, the validation consists of three steps:

- **Step 1**: Normalize the configuration and modeled configuration distribution. Taking configuration $\mathbf{c}_j$ (normalized) as the mean and ε^2E as the covariance matrix, an n-dimensional Gaussian function is modeled as

$$g_j(\mathbf{c}) := \exp[-\frac{1}{2}(\mathbf{c} - \mathbf{c}_j)^{\mathrm{T}}(\varepsilon^2 \mathrm{E})^{-1}(\mathbf{c} - \mathbf{c}_j)] \tag{4}$$

  to describe the similarity probability between $\mathbf{c}$ and $\mathbf{c}_j$.
- **Step 2**: Normalize the performance and modeled performance distribution. Similarly, for performance, a Gaussian function $h_i(Perf(\mathbf{c}))$ is modeled as

$$h_j(\mathbf{c}) := \exp[-\frac{1}{2\delta^2}(Perf(\mathbf{c}) - Perf(\mathbf{c}_j))^2], \tag{5}$$

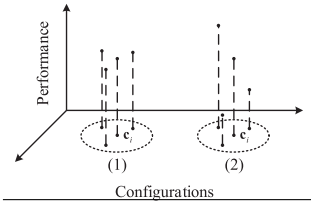
  which describes the similarity probability between the performance of different configurations, where δ is an acceptable performance variation range, as illustrated in Fig. 3(c).
- **Step 3**: Calculate the entropy to measure similarity. Based on Shannon information entropy, for a given measured configuration set $C = \{\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_N\}$, we propose the average entropy of configuration space as

$$H(C) := -\frac{1}{N}\sum_{j=1}^{N} P(\mathbf{c}_j) \log P(\mathbf{c}_j), \tag{6}$$

  where

$$P(\mathbf{c}_j) = P(|Perf(\mathbf{c}_j) - Perf(\mathbf{c}_j + \varepsilon)| < \delta) := \frac{\sum_{\mathbf{c} \in C} g_j(\mathbf{c}) h_j(\mathbf{c})}{\sum_{\mathbf{c} \in C} g_j(\mathbf{c})}.$$

Table 1
An example of configuration space from entropy. When calculating entropy, the parameters ε is 1 and δ is 0.1.



	Conf.	Perf.	$H(C)$
(1)	0 0	0.50	0.033
	0 1	0.52	
	1 0	0.49	
	1 1	0.51	
(2)	0 0	0.50	0.366
	0 1	0.10	
	1 0	0.90	
	1 1	0.70	

Table 2
Results of entropy calculation on 11 real SUPs.

S	N	δ	$H(C)$	$H(C) < H(C_b)$?
x264	1152	0.064	0.218	✓
SQLite	4553	0.015	0.297	✓
LLVM	1024	0.021	0.291	✓
Apache	192	0.060	0.288	✓
BDB-C	2560	0.100	0.172	✓
BDB-J	180	0.085	0.069	✓
HIPA^{cc}	13 485	0.043	0.293	✓
HSMGP	3456	0.062	0.141	✓
DUNE MGS	2304	0.021	0.301	✓
SaC	62 523	0.012	0.357	×
JavaGC	16 697	0.010	0.409	×

For illustration, we set two sets of example data in Table 1, where (1) shows the state of performance in our view instead of (2). The calculations show that (1) has less entropy than (2), which means that it is more stable.

Before computing the ground truth, we determine an entropy threshold $H(C_b)$ to judge if the data of a system satisfies the view. Constructing a set of data C_b where the performance difference between each pair of data is δ, then using Eq. (6), we have $H(C_b) = -h_i(Perf(\mathbf{c}_j) + \delta) \log h_i(Perf(\mathbf{c}_j) + \delta) = 0.303$. We use an open dataset[1] that contains several months of performance measurements of eleven widely used SUPs. The entropy results on these SUPs are provided in Table 2, where the parameter $\varepsilon = 1/3$, $\delta = std(Perf(S, C))/3$ and $std(\cdot)$ is the standard deviation.

Overall, our view is validated in most of these systems. However, for SaC and JavaGC, the standard deviation of their performance is small but the configuration space is large, hence resulting in a small entropy in the configuration and high uncertainty of the performance.

4.2. RSFIN algorithm

Based on the view mentioned in Section 4.1, we propose an approach to uncover and utilize the hidden distributions in configuration space, referred to as rule search-based fuzzy inference network (RSFIN).

Fig. 4 illustrates a three-phase training process of RSFIN. In the first phase, we initialize the architecture parameters and algorithm

[1] http://www.fosd.de/SPLConqueror/.

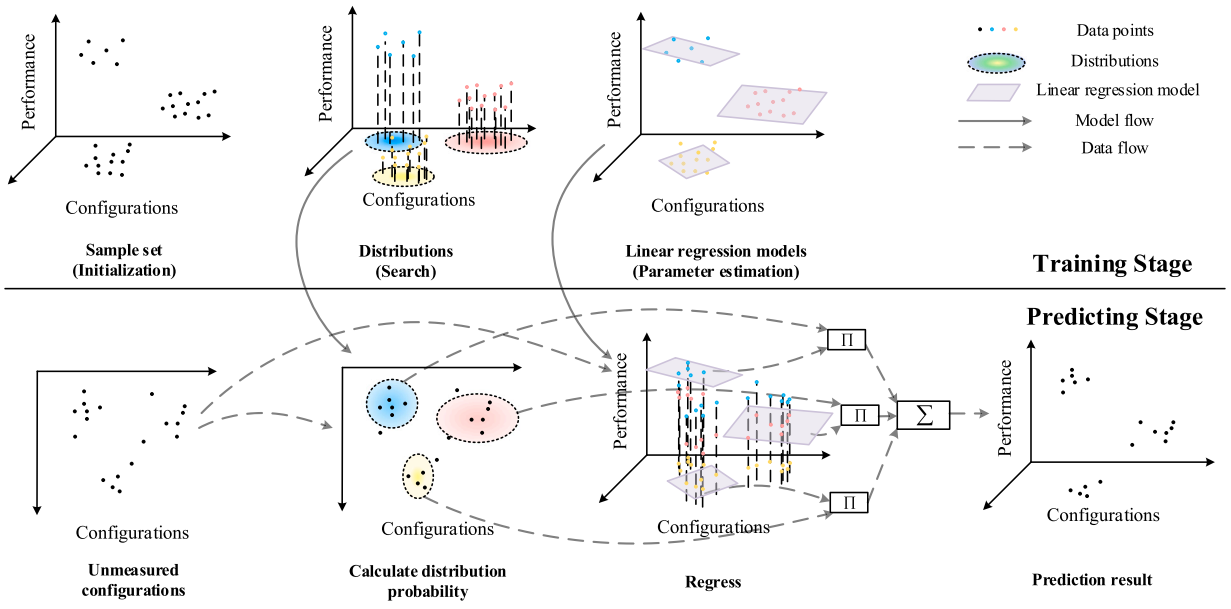


Fig. 4. An overview of RSFIN. The training stage is divided into three phases: (a) Initialization: initialize the model architecture and parameters using the sample set. (b) Search: iteratively update the model architecture to characterize the distributions of configuration aggregations (data points with different colors belong to their corresponding distributions). (c) Parameter Estimation: update algorithm parameters based on gradient descent. In the predicting stage, we apply the above models to infer the prediction results. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

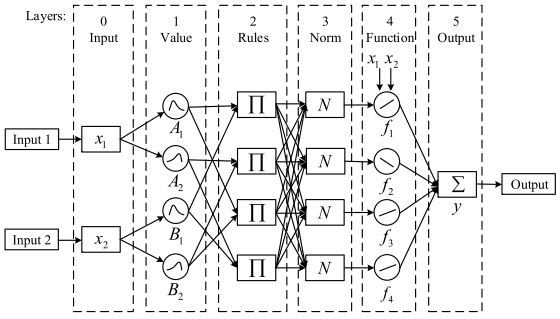


Fig. 5. The equivalent ANFIS architecture, where a circular node means that the layer contains hyperparameters.

parameters of RSFIN based on a given sample set. In the second phase, we search to update the architecture parameters based on the entropy of the dataset so that RSFIN can explicitly characterize the distributions of configuration aggregation (i.e., the hidden distributions of configuration space). In the final phase, we use Adam algorithm (Kingma and Ba, 2014) to update the algorithm parameters iteratively and produce the performance model. Following this process, we describe the RSFIN algorithm as shown in Algorithm 1.

Algorithm 1 RSFIN(C_s, P_s)

Input: (C_s, P_s): sample set $(C_s, P_s) \leftarrow \{(\mathbf{c}, Perf(S, \mathbf{c}))|\mathbf{c} \in C, \text{measurement effort} \leqslant EC\}$
Output: $M(\cdot|\mathcal{R}, w)$: performance model M, architecture parameter $\mathcal{R}$ and algorithm parameter w
Initialization:
Divide the sample set (C_s, P_s) into validation data (C_{valid}, P_{valid}) and training data (C_{train}, P_{train})
$w \leftarrow$ generate initial value based on C_s
$\mathcal{R} \leftarrow$ generate network architecture code based on entropy of C_s
Search:
$\delta_w \leftarrow$initialize the learning rate
for $t \leq$ number of search iterations **do**
  $\mathcal{R}_t \leftarrow$ update based on old $\mathcal{R}$ and entropy of C_s
  $w_t \leftarrow w - \delta_w \nabla_w L(M(C_{train}|\mathcal{R}_t, w), P_{train})$
  if $L(M(C_{valid}|\mathcal{R}_t, w_t), P_{valid}) < L(M(C_{valid}|\mathcal{R}, w), P_{valid})$ **then**
    $\mathcal{R} \leftarrow \mathcal{R}_t$
  end if
end for
Parameter estimation:
for $t \leq$ number of training iterations **do**
  $\delta_w \leftarrow$ update the learning rate based on t, δ_w and $\nabla_w L$
  $w \leftarrow w - \delta_w \nabla_w L(M(C_{train}|\mathcal{R}, w), P_{train})$
end for
return $M(\cdot|\mathcal{R}, w)$;

4.2.1. Adaptive network-based Fuzzy inference system

RSFIN is designed based on the Adaptive Network Fuzzy Inference System (ANFIS) framework. ANFIS can effectively solve complex and non-linear problems as it integrates FIS (Fuzzy Inference System (Mendel, 1995)) and ANN (Adaptive Neural Network (Popovic, 2000)). More importantly, ANFIS can easily express and utilize probabilistic models.

For an ANFIS with boolean input (x_1, x_2) and output y, we suppose that it has four fuzzy if–then rules of Takagi–Sugeno's type (Takagi and Sugeno, 1983; Ma et al., 1998), as follows:

$$\begin{aligned}
&Rule\ 1: \ If\ x_1\ is\ A_1\ and\ x_2\ is\ B_1\ Then\ f_1 = p_1x_1 + q_1x_2 + r_1,\\
&Rule\ 2: \ If\ x_1\ is\ A_2\ and\ x_2\ is\ B_2\ Then\ f_2 = p_2x_1 + q_2x_2 + r_2,\\
&Rule\ 3: \ If\ x_1\ is\ A_1\ and\ x_2\ is\ B_2\ Then\ f_3 = p_3x_1 + q_3x_2 + r_3,\\
&Rule\ 4: \ If\ x_1\ is\ A_2\ and\ x_2\ is\ B_1\ Then\ f_4 = p_4x_1 + q_4x_2 + r_4,
\end{aligned} \tag{7}$$

where $\{p_i, q_i, r_i | i = 1, \ldots, 4\}$ is the set of parameters. The corresponding ANFIS is shown in Fig. 5, where the node functions in the same layer are of the same function family. We provide a detailed description of this five-layer structure as follows.

Layer 1: Each node i in this layer uses the following function to calculate the membership value and output to layer 2:

$$O_i^1 = m_{\mathcal{P}_i}(x), \tag{8}$$

where $x \in \{x_1, x_2\}$, and $m_{\mathcal{P}_i}$ is the membership function to specify the degree, to which x satisfies the fuzzy partition $\mathcal{P}_i$. These connections between layer 1 and layer 2 correspond to the rules. Here, $\mathcal{P}_i = A_i$ if $i = 1, 2$, $\mathcal{P}_i = B_{i-2}$ *if* $i = 3, 4$. $m_{\mathcal{P}_i}(x)$ is usually a bell-shaped function whose curve varies with the change of hyperparameters, presented in the following form:

$$m_{\mathcal{P}_i}(x) = \exp[-(\frac{x - \mu_i}{\sigma_i})^2]. \tag{9}$$

We use w_p to represent the hyperparameter set $\{\sigma_i, \mu_i\}$, referred to as **premise parameters**.

Layer 2: The nodes in this layer multiply the input signals and output to layer 3, i.e.,

$$\begin{aligned} O_i^2 &= w_i = m_{A_i}(x_1) m_{B_i}(x_2) \ for\ i = 1, 2, \\ O_i^2 &= w_i = m_{A_{i-2}}(x_1) m_{B_{5-i}}(x_2) \ for\ i = 3, 4. \end{aligned} \tag{10}$$

The output of this layer represents the firing strength of each rule. In this paper, **these rules characterize the distributions of configuration**, and the firing strength reflects the probability of configuration $\mathbf{c}$ satisfying the corresponding distribution.

Layer 3: The nodes in this layer calculate the ratio of the firing strength of each rule to the sum of all rules' firing strengths, i.e.,

$$O_i^3 = \bar{w}_i = w_i / \sum_{i=1}^{4} w_i \ for\ i = 1, 2, 3, 4. \tag{11}$$

The output of this layer refers to the normalized firing strengths.

Layer 4: The nodes in this layer multiply the normalized firing strengths of each rule by the corresponding function of the Then-part and output to layer 5, i.e.,

$$O_i^4 = \bar{w}_i f_i = \bar{w}_i (p_i x_1 + q_i x_2 + r_i) \ for\ i = 1, 2, 3, 4. \tag{12}$$

We use w_c to denote the hyperparameter set $\{p_i, q_i, r_i\}$, referred to as **consequent parameters**.

Layer 5: The nodes of this layer add up all input signals as the final output, i.e.,

$$y = O_1^5 = \sum_{i=1}^{4} \bar{w}_i f_i. \tag{13}$$

In short, ANFIS is determined by a set of if–then rules and two parameters w_p and w_c. These will be determined in subsequent sections.

4.2.2. Initialization

RSFIN is a variant of ANFIS that can adaptively determine the rules layer. Its initialization is divided into three steps: sampling division, fuzzy partitions initialization, and rule-base initialization. The operations in this phase are based on the sample set, i.e.,

$$\begin{aligned} (C_s, P_s) = \{&(\mathbf{c}, Perf(S, \mathbf{c})) | \mathbf{c} \in C, \\ &\text{measurement effort } EC\}. \end{aligned} \tag{14}$$

The output from this phase includes the validation set $D_{valid} = (C_{valid}, P_{valid})$, the training set $D_{train} = (C_{train}, P_{train})$, the initial premise parameter w_p and the initial architecture parameter $\mathcal{R}$, referred to as the rule base.

Sampling division. The purpose of sampling division is to obtain the validation set D_{valid} and training set D_{train} that are respectively used in the search phase and parameter estimation phase. To make the distribution of D_{valid} approximate the distribution of the configuration space, we utilize random sampling to generate D_{valid} without bias from (C_s, P_s), and the remainder as D_{train} (line 1).

Fuzzy partition initialization. Automatic initialization fuzzy partition (i.e., layer 1) plays a role in handling heterogeneous configuration options and simplifying subsequent calculations. We apply Mean Shift (Comaniciu and Meer, 2002) to determine the cluster center of each option value, and derive the initial Gaussian membership function (9) from the clusters. Specifically, given C_s with n options, Mean Shift finds the dense areas of the option value, and determine the center of each cluster, denoted as:

$$\begin{aligned} K = \{&\mathbf{k}_j^{(i)} | j\text{th cluster center of } i\text{th option}, \\ &i = 1, 2, \ldots, n, j = 1, 2, \ldots\}. \end{aligned} \tag{15}$$

Mean Shift obviates the need to specify the number of clusters in advance, and its results can be used as the mean μ of Eq. (9). Another parameter of Eq. (9), the standard deviation σ, is conventional to be initialized with a small value (e.g., $\sigma_0 = 0.01$) (line 2), which guarantees that the membership functions of identical options can be separated. In sum, the initial fuzzy partitions of layer 1 are formulated as:

$$\begin{aligned} mf = \{&mf_j^{(i)} \mid mf_j^{(i)}(x) = \exp[-(\frac{x - \mathbf{k}_j^{(i)}}{\sigma_0})^2], \\ &i = 1, 2, \ldots, n, j = 1, 2, \ldots, \dim \mathbf{k}^{(i)}\}. \end{aligned} \tag{16}$$

Rule-base initialization. We consider the connections between layer 1 and layer 2 (i.e., rules) as the characterization of data distribution, and treat the firing strength as the distribution probability of the configuration. Therefore, we define a valid rule that conforms to the following: at least one configuration exists that makes the rule's firing strength greater than the predefined threshold.

For an original ANFIS, the number of rules is usually exponential in n. However, since the configuration space is not full of valid configurations, the number of valid rules is limited. To filter out the invalid rules, we treat the rules as a hyperparameter $R \in \mathbb{R}_{n \times m}$, and tune it during the training process.

To automatically initialize layer 2, we first randomly select M configurations as C_M in the configuration space V and calculate their entropy $H_i, i = 1, \ldots, M$ with C_s respectively in (6), where the performance of them are assumed to be the mean of P_s. Secondly, we select configurations with entropy lower than $H(C_b)$ as $C_R = \{\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_m\} \subseteq C_M$. Finally, we establish a mapping between C_R and the rules (line 3):

$$\begin{aligned} &\mathbf{c}_k \xrightarrow{K} \mathbf{r}^{(k)}, \\ &\mathbf{r}_i^{(k)} = \underset{1 \leqslant j \leqslant \dim \mathbf{k}^{(i)}}{\arg\max}\ mf_j^{(i)}(x_i), \end{aligned} \tag{17}$$

where $\mathbf{c}_k = (x_1, x_2, \ldots, x_n)^{\mathrm{T}}$. With this mapping, we can initialize the rule base $\mathcal{R}_0 = [\mathbf{r}^{(1)\mathrm{T}}, \mathbf{r}^{(2)\mathrm{T}}, \ldots, \mathbf{r}^{(m)\mathrm{T}}]$.

4.2.3. Search

Since the rule base $\mathcal{R}$ is difficult to obtain through conventional optimization algorithms, we propose a novel approach based on the "Search-Estimation" framework (Elsken et al., 2019) to iteratively optimize $\mathcal{R}$.

The training and the validation loss are denoted as $\mathcal{L}_{valid}$ and $\mathcal{L}_{train}$ respectively,

$$\mathcal{L}_T(\mathcal{R}, w) = L(M(C_T; \mathcal{R}, w | D_T), P_T),\ T = valid \text{ or } train. \tag{18}$$

Both of them are determined not only by the rule base $\mathcal{R}$, but also the premise parameter w_p and the consequent parameter w_c. This implies a bilevel programming (Dempe et al., 2015; Liu et al., 2018), with $\mathcal{R}$ as the upper-level variable and (w_p, w_c) as the lower-level variable:

$$\min_{\mathcal{R}} \quad \mathcal{L}_{valid}(\mathcal{R}, \{w_p^*, w_c^*\}) \tag{19}$$

$$s.t. \quad \{w_p^*, w_c^*\} = \arg\min_{w_p, w_c} \mathcal{L}_{train}(\mathcal{R}, \{w_p, w_c\}). \tag{20}$$

Table 3
Overview of the real-world subject systems. #Bin and #Num denote the numbers of binary and numeric configuration options, respectively. #C denotes the number of valid configurations.

System	Domain	Lang	#Bin	#Num	#C
×264 (Merritt and Vanam, 2006)	Video encoder	C	13	0	1152
SQLite (Bhosale et al., 2015)	Database	C	39	0	4653
LLVM (Lattner and Adve, 2004)	Compiler	C++	10	0	1024
Apache (Knaus et al., 1985)	Web server	C	8	0	192
BDB-C (Olson et al., 1999)	Database	C	16	0	2560
BDB-J	Database	Java	17	0	180
HIPA^{cc} (Membarth et al., 2015)	Image processing	C++	31	2	13 485
HSMGP (Grebhahn et al., 2014)	Stencil-Grid solver	C++	11	3	3456
DUNE MGS (MGS, 2021)	Multi-Grid solver	C++	8	3	2304
JavaGC	Runtime environment	Java	12	23	10^{31}
Sac (C, 2021)	Compiler	C	52	7	10^{23}

Approximate the gradient of $\mathcal{R}$. Evaluating the gradient of $\mathcal{R}$ is exactly prohibitive due to the expensive inner optimization. Therefore, we introduce a simple approximation scheme (Liu et al., 2018) for inner optimization:

$$\begin{aligned}&\nabla_{\mathcal{R}}\mathcal{L}_{valid}(\mathcal{R},\{w_p^*,w_c^*\})\\&\approx\nabla_{\mathcal{R}}\mathcal{L}_{valid}(\mathcal{R},\{w_p-\delta_w\nabla_{w_p}\mathcal{L}_{train}(\mathcal{R},\{w_p,w_c\}),w_c^*\}),\end{aligned}\tag{21}$$

where δ_w is the learning rate. The basic idea is to approximate w_p^* by updating w_p via only a single training step (Luketina et al., 2016; Finn et al., 2017; Metz et al., 2017) (line 7), rather than training until convergence to solve the inner optimization completely (20). In practice, this approximation scheme is feasible with an appropriate learning rate (Liu et al., 2018).

Search $\mathcal{R}$. Based on the discussion of configuration space in Section 4.1, we propose an entropy-based rule base update strategy to search for $\mathcal{R}$. The search process is similar to the initialization process of $\mathcal{R}$, but the existing rules will have negative effects as penalty terms to avoid repeated searches. Referring to the evolutionary-based architecture updating strategy (Real et al., 2019), we suggest a evolutionary-based (Angeline et al., 1994) rule search method. We use rule coding mutation to generate new rule r_{new} by changing arbitrary code of the randomly selected rule $r_i \in \mathcal{R}_0$. r_{new} is merged into $\mathcal{R}_0$ to update $\mathcal{R}$. Corresponding to the process of entropy (6), the penalty term can be expressed as:

$$\begin{aligned}P(c_i)&=\frac{\sum_{c\in C}p_i(\mathbf{c})h_i(\mathbf{c})}{\sum_{c\in C}p_i(\mathbf{c})},\\p_i(\mathbf{c})&=g_i(\mathbf{c})\prod_{\mathbf{c}_r\in C_R}(1-g(\mathbf{c}|\mathbf{c}_r)),\end{aligned}\tag{22}$$

where C_R is the set of configurations corresponding to the existing rules, and $g(\mathbf{c}|\mathbf{c}_r)$ represents the membership function of the configuration with c_r as the mean. By repeating the three steps of randomly selecting configuration sets, computing their entropy with C_s, and filtering them with $H(C_b)$, $\mathcal{R}$ will be obtained until the above process converges (line 6).

Premise parameter estimation. Inspired by the backpropagation algorithm (Rumelhart et al., 1986) in the feedforward neural network (FNN) (Bebis and Georgiopoulos, 1994), we apply the Adam algorithm (Kingma and Ba, 2014) combined with backpropagation to optimize w_p (line 13–14).

Consequent parameter estimation. Based on D_{train}, we can estimate parameters by solving the following:

$$w_c^*=\arg\min_{w_c}\sum_{i=1}^{N}(M(C_{train}^{(i)};\mathcal{R},\{w_p,w_c\})-P_{train}^{(i)})^2,\tag{23}$$

where N is the sample size and $Vec(\cdot)$ is a matrix vec operator. According to Chavent (1983), it has been proved that the solution of (23) can be formulated as :

$$Vec(w_c^*)=(A^\top A)^{-1}A^\top P_{train},\tag{24}$$

where

$$A=[Vec(X_1(O_3^{(1)})^\top),Vec(X_2(O_3^{(2)})^\top),\dots,Vec(X_N(O_3^{(N)})^\top)]^\top.$$

In practice, $(\cdot)^{-1}$ is the pseudo-inverse (Golub and Kahan, 1965), because the matrix $A^\top A$ is a singular matrix without the inverse when the input $\mathbf{c}$ is a sparse binary vector.

5. Performance evaluation

To evaluate RSFIN, we implement it in Python and test it on various datasets from real software systems developed in different domains.[2] This section presents the details of the experiments and discusses the experimental results.

5.1. Experiment setup

Subject systems. We use an open dataset that contains several months of performance measurements of eleven widely used SUPs. More details about these SUPs (e.g., version, how they were measured, etc.) are available online.[3] As shown in Table 3, these systems are adopted in various scenarios, compiled and implemented in distinct languages, and have different sizes of configuration space. The first six SUPs with only binary configuration options were used in Siegmund et al. (2012b,a) and Guo et al. (2013, 2018). The other five SUPs with both binary and numeric configuration options were also used in Ha and Zhang (2019) and Siegmund et al. (2015).

Experimental Setup. The essence of performance prediction is to establish the mapping relationship between configuration space and performance. Therefore, this experiment will focus on the prediction accuracy of the performance model generated by the algorithm. We use the mean relative error (MRE) to measure the prediction accuracy, calculated as:

$$\begin{aligned}&MRE(C_{test},P_{test})=\\&\frac{1}{|C_{test}|}\sum_{\mathbf{c}_0\in C_{test}}\frac{|M(\mathbf{c}_0)-Perf(S,c_0)|}{Perf(S,c_0)}\times 100\%,\end{aligned}\tag{25}$$

where $\mathbf{c}_0$ is a configuration of testing dataset C_{test}, and $M(\mathbf{c}_0)$ and $Perf(S,c_0)$ are the predicted and actual performance of $\mathbf{c}_0$, respectively. Correspondingly, the prediction accuracy is $1-MRE$. We also use $Margin$ as another valuation metric to denote the margin of the 95% confidence interval of the $MREs$ in experiments, which indicates the standard uncertainty of $MREs$ and reflects the consistency of the algorithm. $Margin$ calculated as $1.96\sigma/\sqrt{z}$, where σ is the standard deviation of $MREs$ and z is the number of experiments.

In our experiments, the prediction algorithms are evaluated through the following steps: 1. Under the constraint of EC, perform random sampling with a fixed random seed to obtain the training set, and meanwhile, use the remaining portion of the dataset for testing. 2.

[2] https://github.com/RSFIN/RSFIN.
[3] http://www.fosd.de/SPLConqueror/.

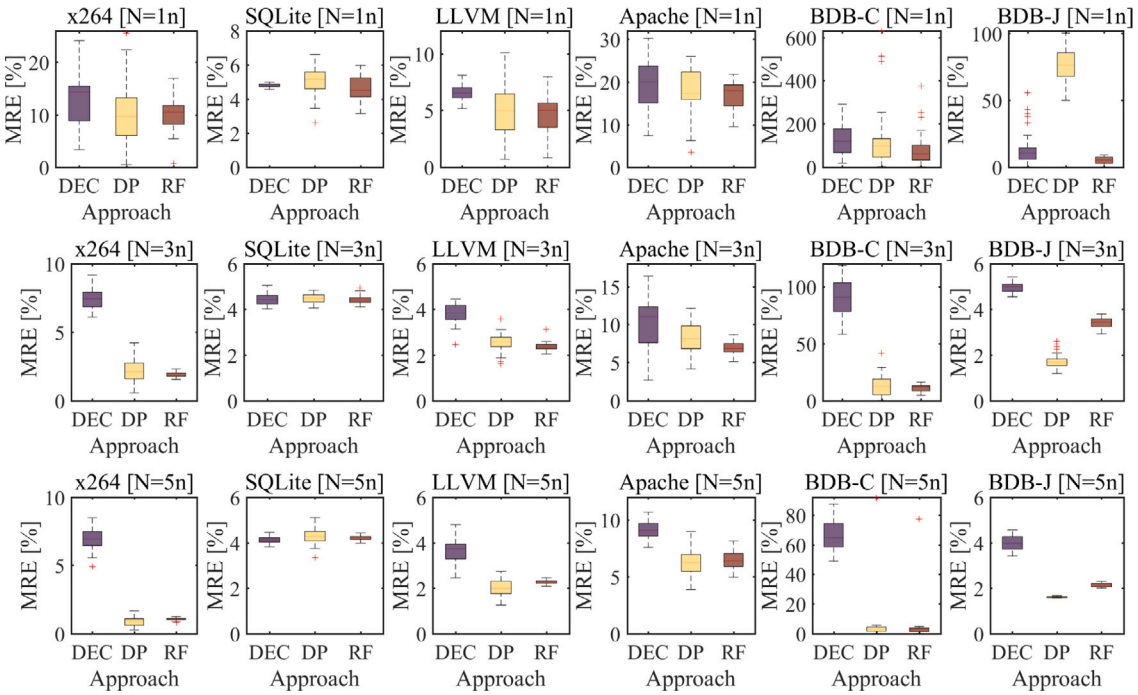


Fig. 6. Performance comparison of different approaches (DEC: DECART, DP: DeepPerf, RF: RSFIN) on six SUPs with only binary options.

Train a performance model based on the training dataset. 3. Evaluate the performance model in terms of MRE and $Magin$ based on the testing dataset; 4. Repeat step $1-3$ 30 times (i.e., $z = 30$) for statistical analysis (e.g., t-test Semenick, 1990), where Step 1 takes a different random seed for each repetition. In addition, we use the Vargha–Delaney statistic $\hat{A}_{12}$ (Vargha and Delaney, 2000) to measure the level of effect as a supplementary metric to help determine if the approach is worth exploration when the mean t-test is insignificant (i.e., $p > 0.05$).

Comparison algorithms DECART (Guo et al., 2018) and DeepPerf (Ha and Zhang, 2019) were recently proposed and both outperform SPL Conqueror in same dataset (Siegmund et al., 2012b,a, 2015). Hence, in our experiments, we only compare RSFIN with DECART and DeepPerf to evaluate the performance of RSFIN. Specifically, we compare RSFIN with DECART on SUPs with only binary options, and compare RSFIN with DeepPerf on all SUPs.

We utilize the code in their online repository to replicate the results of DECART[4] and DeepPerf.[5] We set the best hyperparameter tuning technique for DECART, as suggested in their research: 10-fold cross-validation and grid search. Other hyperparameters for DECART and DeepPerf are the same as described in their publications. Table 4 presents some hyperparameters of each algorithm (including RSFIN).

Effort constraint. Referring to the experimental designs in Guo et al. (2013, 2018) and Ha and Zhang (2019), we set the effort constraint (i.e., EC) to $\{n, 3n, 5n\}$ and $\{10n, 30n, 50n\}$, respectively, for SUPs with only binary options and SUPs with binary and numeric options, where n is the number of configuration options.

5.2. Experimental results

Given each SUP and EC pair, we conduct 30 experiments on RSFIN, DECART, and DeepPerf, respectively, and present the experimental results. Tables 5 and 6 tabulate the results including the mean and margin of MREs and the average execution time obtained in all the experiments. In addition, we visualize the experimental results with the box-plot (Williamson et al., 1989) in Figs. 6 and 7.

Table 4
Hyperparameters of each algorithm.

Algorithm	Hyperparameters
DECART	**Resampling techniques**: hold-out with random sampling; **Parameter tuning methods**: grid search; **max_depth**: [1, 25]; **min_samples_leaf**: [1, 15]
DeepPerf	**Sampling policy**: hold-out with random sampling; **Number of neurons/layers**: 128; **Epochs**: 2000;
RSFIN	**Sampling policy**: hold-out with random sampling; **Initial_learning_rate**: 0.001; **Epochs**: 100;

As shown in the table, compared with DECART, RSFIN statistically achieves a higher prediction accuracy and a smaller margin for x264, LLVM, Apache, BDB-C and BDB-J. Also, compared with DeepPerf, RSFIN statistically achieves a higher prediction accuracy and a smaller margin for Apache, SQLite, BDB-C and HSMGP. When $EC = 5n$, there is no statistically significant difference between the two algorithms when applied to Apache and BDBC (i.e., $p > 0.05$). It is worth noting that in all the experiments except for JavaGC and Sac, RSFIN exhibits superior or comparable performance as DeepPerf in the smallest EC (n/$10n$, for SUPs with binary and numeric options, respectively). Moreover, RSFIN takes much less execution time than DeepPerf for all SUPs during the experiment. The small $Margin$ running 30 times on all systems also suggests a very small variance that RSFIN produces on a given SUP, which also indicates that RSFIN has a narrower interquartile range as shown in the box-plot. Hence, we conclude that RSFIN achieves better stability and reliability.

On the other hand, the prediction errors across different SUPs may vary widely, even with the same EC, principally due to the different intrinsic structures of systems, which are reflected in the data as the difference in the average entropy of configuration space. Based on the entropy perspective mentioned in Section 4.1, the complexity of the internal interaction of the system will be reflected in the data entropy, which will affect the performance of the prediction model, as shown in Fig. 8. For example, under the constraint of $EC = 3n$, the average MRE

[4] https://github.com/jmguo/DECART.
[5] https://github.com/DeepPerf/DeepPerf.

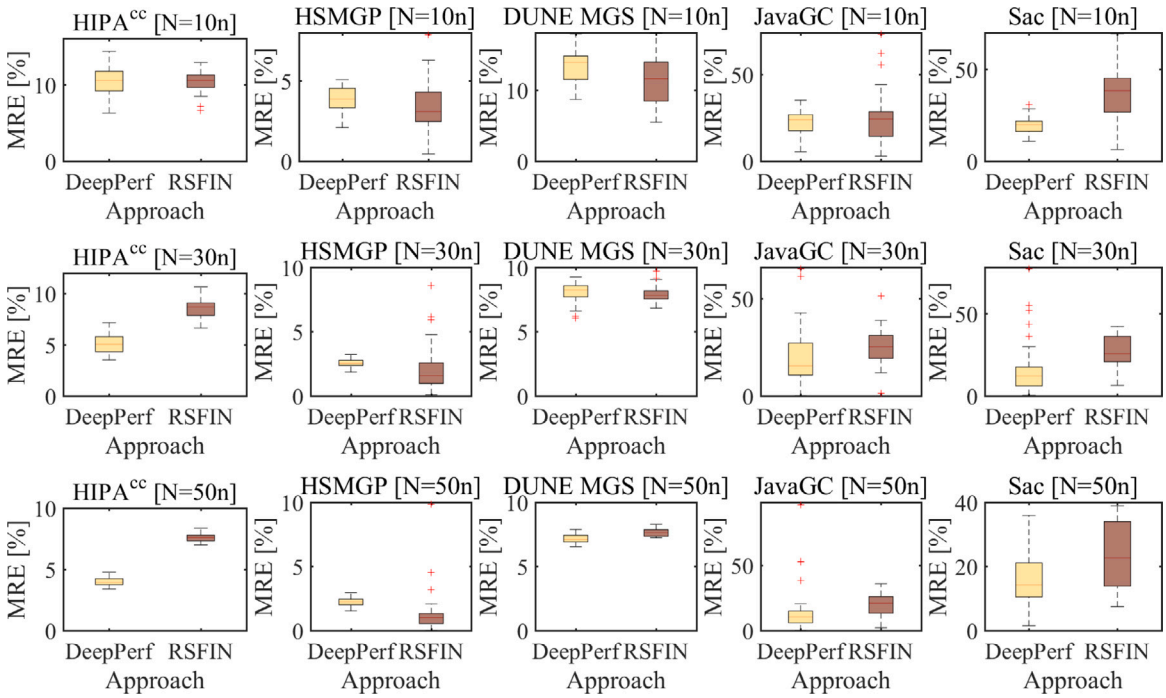


Fig. 7. Performance comparison with DeepPerf on five SUPs with numeric options.

Table 5
Statistics of the experimental results for SUPs with only binary options. n: The number of configuration options for the SUP. Mean: mean of the $MREs$ seen in 30 experiments. Margin: margin of the 95% confidence interval of the $MREs$ in 30 experiments. The $\hat{A}_{12}$ represents the level of effect of RSFIN in comparison with the other algorithm. The **bold** value represents a statistically significant performance improvement, which is chosen using t-test on 30 $MREs$ with a significant level of 0.05. Time: the average time cost of model training (in seconds).

SUP(n)	EC	DECART			DeepPerf				RSFIN			
		Mean	Margin	$\hat{A}_{12}$	Time	Mean	Margin	$\hat{A}_{12}$	Time	Mean	Margin	Time
×264 ($n = 13$)	n	12.83	1.86	**0.89**	1.26	10.43	2.28	0.59	217.54	10.31	1.25	17.54
	$3n$	7.44	0.27	**1.00**	1.31	2.13	0.31	**0.72**	223.97	1.88	0.07	24.51
	$5n$	6.91	0.28	**1.00**	1.28	0.87	0.11	0.02	221.77	1.05	0.04	33.50
SQLit ($n = 39$)	n	4.79	0.04	0.74	1.75	5.04	0.32	**0.89**	304.98	4.65	0.26	28.75
	$3n$	4.43	0.09	0.51	1.92	4.48	0.08	**0.79**	318.46	4.42	0.07	29.31
	$5n$	4.12	0.05	0.07	2.11	4.27	0.13	**0.73**	325.04	4.21	0.04	30.76
LLVM ($n = 10$)	n	6.52	0.23	**1.00**	1.71	5.09	0.80	**0.61**	205.44	4.65	0.53	1.91
	$3n$	3.81	0.15	**1.00**	1.70	2.54	0.15	**0.90**	207.17	2.37	0.07	2.07
	$5n$	3.63	0.21	**1.00**	1.75	1.99	0.15	0.11	209.91	2.27	0.03	2.14
Apache ($n = 8$)	n	19.60	2.13	**0.78**	1.31	17.87	1.85	**0.65**	192.67	16.93	1.25	6.96
	$3n$	10.46	1.17	**0.99**	1.30	8.25	0.75	**0.95**	194.91	6.92	0.30	7.12
	$5n$	9.08	0.26	**1.00**	1.32	6.29	0.31	0.50	196.35	6.43	0.29	7.26
BDB-C ($n = 16$)	n	128.83	26.72	**0.78**	1.76	133.60	54.33	**0.72**	227.34	85.74	30.12	30.12
	$3n$	90.71	5.85	**1.00**	1.87	13.10	3.39	**0.70**	229.54	10.89	1.21	32.64
	$5n$	65.98	3.74	**1.00**	1.92	5.82	1.33	0.67	236.59	4.92	1.44	36.95
BDB-J ($n = 17$)	n	14.84	4.96	**1.00**	1.69	7.25	4.21	**1.00**	325.42	5.31	1.01	12.92
	$3n$	4.95	0.09	**1.00**	1.75	1.73	0.12	0.00	327.51	3.41	0.08	13.15
	$5n$	3.98	0.12	**0.93**	1.87	1.61	0.01	0.00	333.94	2.14	0.03	13.47

of system BDB-C is 10.89 while the average MRE of system BDB-J is only 3.41. Correspondingly, recall that the entropy of the data for the two systems is 0.172 and 0.069 in Table 2, respectively.

Furthermore, although the average MRE of the other systems fell within 10 even when EC was set to 50n, RSFIN performed poorly when implemented on Sac and JavaGC. This is because the internal interactions of these two systems are so complicated that it is difficult to explore their impact on performance from the configuration level, which is reflected in their entropy greater than $H(C_b)$. As shown in Fig. 8, the point corresponding to Sac and JavaGC is the farthest from the origin, which means that the internal interaction of the above system is more complex and these approaches do not always yield satisfactory performance. However, RSFIN demonstrates a good generalization in other systems with entropy less than $H(C_b)$. Please note that the accuracy of DECART on the BDB-C system is not included in Fig. 8. This is because it is considered a severe outlier, deviating significantly from other data points. We have chosen to omit it from the scatter plot to ensure a more accurate representation of the relationships and trends among the majority of the data points, and to avoid any misleading interpretations. Additionally, during the fitting process, this outlier is also treated as an anomalous value and ignored.

Note that in all of these experiments, the average execution time of RSFIN is 30.47 s, while DeepPerff is 348.26 s. These experimental

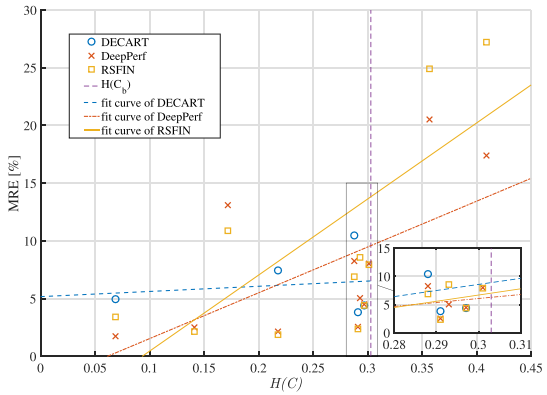


Fig. 8. The impact of average entropy on the prediction performance of different methods is studied for a total of 11 SUPs. The experiments are conducted with $EC = 3n$ (for SUPs with binary options) and $EC = 30n$ (for SUPs with numeric options). Each data point represents the MRE of an SUP, with its average entropy $H(C)$ matching the values in Table 2. The fitting curve represents the linear regression line of the algorithm's data points.

Table 6
Statistics of the experimental results for SUPs with numeric options.

SUP(n)	EC	DeepPerf				RSFIN		
		Mean	Margin	$\hat{A}_{12}$	Time	Mean	Margin	Time
HIPA^{cc} ($n = 33$)	$10n$	10.45	0.76	0.56	402.49	10.31	0.51	45.12
	$30n$	5.06	0.33	0.00	419.4	8.57	0.35	47.48
	$50n$	3.97	0.12	0.00	421.59	7.57	0.12	48.57
HSMGP ($n = 14$)	$10n$	3.87	0.27	**0.65**	151.84	3.41	0.56	16.94
	$30n$	2.53	0.12	**0.67**	176.78	2.14	0.71	17.01
	$50n$	2.21	0.11	**0.87**	188.43	1.39	0.66	17.16
DUNE MGS ($n = 11$)	$10n$	13.31	0.31	**0.91**	160.15	11.21	1.13	6.22
	$30n$	8.01	0.31	0.54	162.89	7.92	0.22	6.34
	$50n$	7.13	0.12	0.01	171.51	7.63	0.11	6.71
JavaGC ($n = 35$)	$10n$	22.51	2.73	0.24	491.52	26.53	5.87	48.36
	$30n$	20.54	5.84	0.24	503.16	24.93	3.42	49.37
	$50n$	15.98	7.13	0.33	527.31	20.11	3.17	51.17
Sac ($n = 59$)	$10n$	19.54	1.74	0.03	883.15	35.71	5.21	84.95
	$30n$	17.42	6.51	0.16	957.16	27.21	3.54	85.31
	$50n$	15.64	3.29	0.06	1015.34	23.45	3.67	87.48

results show that RSFIN achieves a competitive performance with state-of-the-art approaches, while requiring less execution time, which has great benefits when applied to prediction tasks.

5.3. Discussion

For configurable software systems, the same degree of change in different configuration options or combination of options has different impact on the measurement of system performance. This leads to the findings mentioned in Section 4.1 at the macro level that similar configurations yield a similar performance. On the other hand, in most cases, due to internal constraints between different options, valid configurations do not fill up the entire configuration space. The above factors cause the configurations to be distributed in clusters, and there is a small performance difference within the same cluster. Based on the above characteristics, we design RSFIN to find the configuration structure of different clusters, and build the corresponding performance distribution to achieve accurate performance prediction.

Performance. Given an effort constraint, RSFIN effectively and automatically captures the hidden structures in configuration space, and attains a reasonable prediction accuracy in our experiments. However, RSFIN may not achieve ideal prediction accuracy if the configuration space is too complicated and intensive to find hidden distributions, such as JavaGC and SaC. In sum, RSFIN can achieve acceptable performance in most application scenarios, which can be noticed in comparison with state-of-the-art approaches.

For most learning algorithms, it is inevitable to be nonconvergent or overfitting with less measurements (e.g., n or $3n$). Therefore, the measurements RSFIN requires are acceptable. For some algorithms such as DeepPerf, the accuracy increases as the measurements increase, which, however, incurs higher time cost consequently. On the contrary, RSFIN aims to capture hidden structures and build performance distributions corresponding to different structures to characterize the configuration space. Therefore, the measurements required by RSFIN focus on quality (i.e., maximize the coverage of different clusters), not on quantity, which helps reduce computing costs in practice.

Applicability. The applicability of RSFIN is determined by the quality of the data. When the internal interaction of the system is complex and the measured data is insufficient to characterize it (e.g., JavaGc and Sac), RSFIN faces challenges to achieve satisfactory performance. We propose to use the average entropy of configuration space (Eq. (6)) to evaluate the complexity of the internal interaction of the system. From the perspective of entropy, an increase in the entropy implies an increase in the entropy carried by data points. As the entropy increases, the performance distribution would appear more chaotic and divergent, resulting in a situation where a slight adjustment to the configuration may lead to a significant fluctuation in the performance. This phenomenon may be attributed to the omission of unknown configuration options that have a critical impact on the performance during the measurement process, or the existence of complex interactions between known configuration options. Therefore, when the average entropy of configuration space exceeds $H(C_b)$, it would be challenging to reconstruct the performance distribution based on known configurations without knowledge of their specific interactions. Consequently, it becomes difficult to predict performance through data-driven approaches.

As shown in Fig. 8, we try to characterize the impact of the average entropy on the prediction performance, and reach the conclusion of positive linear correlation. This conclusion has been confirmed by experiments and can be used as a measure of data quality to estimate the predictability of data before training the performance model. In addition, from Fig. 8, we also observe that this rule is applicable to not only RSFIN, but also other approaches.

Execution Time. The execution time of the learning algorithm is usually not an important issue in the performance prediction scenario, because the process of collecting software performance data is usually much more expensive in terms of time and energy. However, shorter training time often means smaller trial and error costs and wider application scenarios.

In the experiment, DECART can complete a training phase in 1.71 s on average, which is considered fast, but its performance is relatively poor. RSFIN requires a small amount of resources to search for hidden distribution, so its speed is relatively slower, and it takes an average of 30.47 s for training. DeepPerf based on adaptive structure search neural network takes more time to train, with an average of 348.26 s. In sum, RSFIN overcomes the performance limitations of DECART and DeepPerf simultaneously and achieves a balanced trade-off between accuracy and effort.

Interpretability. The goal of interpretability is to describe the internals of a system in a way that is understandable to humans (Gilpin et al., 2018). Researchers have proposed a large number of techniques, including black-box performance-influence models and white-box performance-influence models (Weber et al., 2021; Siegmund et al., 2012b; Guo et al., 2018; Dorn et al., 2020; Velez et al., 2021, 2020), which focus on mining and characterizing how options and their interactions affect the performance. However, for most users, understanding the effect completely and making suitable decisions on configuration can be time-consuming and challenging. It may be more intuitive and friendly to directly provide the configuration tutorial. Therefore, RSFIN attempts to determine the rules to describe the configuration

Table 7
Examples of rules for Apache and HSMGP. The If-part contains rules whose values are used to distinguish fuzzification partitions (Eq. (17)) with the corresponding normalized degree of impact. The Then-part depicts the distribution of performance by the mean of the distribution.

	Option	Apache			HSMGP					
		$r^{(1)}$	$r^{(2)}$	$r^{(3)}$	$r^{(1)}$	$r^{(2)}$	$r^{(3)}$	$r^{(4)}$	$r^{(5)}$	$r^{(6)}$
If-part	Option 1	0 (−0.15)	0 (−0.04)	0 (−0.20)	0 (−0.03)	0 (−0.32)	0 (−0.10)	0 (−0.06)	0 (0.03)	0 (−0.09)
	Option 2	1 (0.74)	0 (−0.22)	0 (0.31)	0 (−0.04)	0 (−0.27)	0 (−0.20)	1 (−0.08)	1 (<0.01)	0 (−0.08)
	Option 3	0 (0.08)	0 (−0.04)	1 (<0.01)	0 (−0.04)	1 (0.15)	0 (0.30)	0 (0.01)	0 (0.01)	0 (−0.04)
	Option 4	0 (0.02)	0 (−0.02)	1 (<0.01)	0 (−0.02)	0 (−0.15)	0 (−0.1)	0 (−0.03)	0 (0.03)	1 (−0.03)
	Option 5	0 (−0.24)	0 (0.05)	0 (−0.1)	1 (0.01)	0 (−0.27)	0 (0.17)	1 (−0.01)	0 (−0.04)	1 (<0.01)
	Option 6	1 (−0.10)	1 (−0.01)	0 (−0.06)	0 (0.19)	0 (0.20)	0 (0.49)	0 (0.19)	0 (−0.10)	0 (0.36)
	Option 7	1 (−0.33)	0 (0.32)	1 (0.14)	0 (<0.01)	1 (0.11)	0 (0.70)	0 (−0.07)	1 (−0.03)	0 (−0.05)
	Option 8	0 (−0.48)	1 (0.29)	0 (0.69)	0 (−0.01)	0 (−0.54)	1 (−0.56)	0 (0.07)	0 (0.08)	0 (0.06)
	Option 9				1 (−0.04)	0 (−0.31)	0 (−0.07)	0 (−0.05)	0 (0.04)	0 (−0.11)
	Option 10				1171 (>−0.01)	198 (<0.01)	198 (0.03)	198 (0.02)	172 (−0.01)	172 (0.02)
	Option 11				2 (0.08)	6 (0.36)	5 (0.57)	6 (0.15)	2 (0.10)	2 (0.03)
	Option 12				0 (0.12)	4 (−0.04)	4 (0.37)	0 (−0.03)	2 (0.05)	4 (0.07)
Then-part	PERF	2710.66	1427.59	1565.07	667.39	1123.18	3658.66	766.8	666.21	864.02

of different clusters and their corresponding system performance, and evaluates the degree of impact each option has on the performance when determining the membership of a cluster through the consequent parameters (Eq. (12)). Users are able to make a configuration plan that meets performance requirements through the above rules.

As described in Section 4, RSFIN updates the rule layer by searching for configurations with low information entropy, and builds the performance model of the corresponding cluster to make performance prediction. Table 7 provides examples of rules when deploying RSFIN based on the training sets of Apache and HSMGP.

Assuming that the user wants to maximize the performance, the user can select $r^{(1)}$(Apache)/$r^{(3)}$ (HSMGP) as the initial configuration, tune the configuration based on the impact degree and direction of different options, and apply RSFIN to further adjust the options. This way, users can intuitively understand system configuration rules without learning about specific options or the impact of option interactions on the performance.

The interpretability of existing performance models is mostly based on linear models (Siegmund et al., 2012b; Velez et al., 2021; Weber et al., 2021) or regression trees (Guo et al., 2018, 2013). Linear models can intuitively represent the degree of influence of each option through weights, but fail to reflect the impact of option interactions. Tree models focus on configuration interactions and can explain the reasoning process of the model. However, as the number of configurations increases, the number of leaf nodes in the tree model increases, making it difficult to understand. The complex internal interactions of many options in software systems, the large but sparse configuration space, and the unsmooth performance surface all contribute to the degraded performance of the above model. In contrast, RSFIN reaps the benefits of the above two models while decoupling the number of rules from the number of configuration options, and hence facilitates the interpretability of the model.

Configuring a system can be challenging for most users due to their lack of in-depth understanding of the software system. At the same time, engineering experts who can provide guidance are limited. Therefore, RSFIN is expected to act as an expert during the actual operation of the system. It determines the representative structure of the configuration cluster through rule search and establishes the corresponding performance model to achieve the characterization of the configuration space. Furthermore, the rules obtained by RSFIN can be provided to users in the form of reference configurations and corresponding performance levels, as shown in Table 7. This helps users understand the configuration space and provides guidance for users to configure software systems.

5.4. Threats to validity

We consider two types of validity, i.e., internal validity and external validity of our approach.

Internal validity: To validate the performance of our approach, we use mean relative error as a metric, which is widely adopted for evaluating prediction accuracy. In the process of experiments, we repeat 30 experiments with algorithms, and perform a significance test to mitigate the impact caused by the imbalance of random sampling. Finally, we record the hyperparameters of our proposed algorithm to ensure its reproducibility.

External validity: We select eleven real-world SUPs with public dataset to evaluate the algorithms. These subject systems with different numbers of options are from various application domains and have been extensively used in the literatures to evaluate the effectiveness of other algorithms, which increases the external validity of our approach. In addition, we also propose an algorithm validity estimation method based on the entropy of data, which can be used to evaluate the data quality to ensure the external validity of the algorithm in practical application.

6. Conclusion and future work

To address the shortcomings of existing techniques for performance prediction of software systems, we proposed RSFIN, which captures hidden distributions in the configuration space. We compared RSFIN with several state-of-the-art approaches based on real-life experiments, and the results show that RSFIN achieves a relatively better trade-off between measurement effort and prediction accuracy. In addition, we proposed and verified an evaluation index based on information entropy, referred to as the average entropy of the configuration space, which is used to measure the complexity of the internal interaction of the configuration space based on the measured data. Finally, the architecture parameters of RSFIN facilitate interpretability, which shows the effect of the configuration structure on performance in the form of rules to help users better configure the software system.

Software system configuration data has many special features, such as structured, clustered and sparsely distributed. When a general machine learning model is directly applied to analyze such data, it may incur unnecessary computing costs. Modeling and analysis of such data distribution is one of our future interests. For RSFIN, random sampling may lead to instability in the initialization process. It is also our plan to combine RSFIN with entropy-based heuristic sampling to further improve the performance.

CRediT authorship contribution statement

Yufei Li: Conceptualization, Methodology, Software, Writing – original draft. **Liang Bao:** Supervision, Writing – original draft. **Kaipeng Huang:** Software, Writing – original draft. **Chase Wu:** Writing – original draft. **Xinwei Li:** Software.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under Grant No. 62172316 and the Key R&D Program of Hebei under Grant No. 20310102D. This work is also supported by the Key R&D Program of Shaanxi under Grant No. 2019ZDLGY13-03-02, the Natural Science Foundation of Shaanxi Province under Grant No. 2019JM-368, and the Soft Science Research Plans of Chengdu [Grant No. 2021-RK00-00177-ZF].

References

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

Angeline, P.J., Saunders, G.M., Pollack, J.B., 1994. An evolutionary algorithm that constructs recurrent neural networks. IEEE Trans. Neural Netw. 5 (1), 54–65. http://dx.doi.org/10.1109/72.265960.

Bao, L., Liu, X., Wang, F., Fang, B., 2019. ACTGAN: Automatic configuration tuning for software systems with generative adversarial networks. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).

Bebis, G., Georgiopoulos, M., 1994. Feed-forward neural networks. IEEE Potentials 13 (4), 27–31.

Bhosale, S., Patil, T., Patil, P., 2015. Sqlite: Light database system. Int. J. Comput. Sci. Mob. Comput. 4 (4), 882–885.

C, S.-A., 2021. https://www.sac-home.org/.

Chavent, G., 1983. Leastsquare parameter estimation technique. Comput. Appl. Math. 2 (1), 3.

Comaniciu, D., Meer, P., 2002. Mean shift: A robust approach toward feature space analysis. IEEE Trans. Pattern Anal. Mach. Intell. 24 (5), 603–619. http://dx.doi.org/10.1109/34.1000236.

Czarnecki, K., Østerbye, K., Völter, M., 2002. Generative programming. In: European Conference on Object-Oriented Programming. Springer, pp. 15–29.

De'ath, G., Fabricius, K.E., 2000. Classification and regression trees: a powerful yet simple technique for ecological data analysis. Ecology 81 (11), 3178–3192.

Dempe, S., Kalashnikov, V., Pérez-Valdés, G.A., Kalashnykova, N., 2015. Bilevel programming problems: theory, algorithms and applications to energy networks. Springer.

Dorn, J., Apel, S., Siegmund, N., 2020. Mastering uncertainty in performance estimations of configurable software systems. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 684–696.

Elsken, T., Metzen, J.H., Hutter, F., 2019. Neural architecture search: A survey. J. Mach. Learn. Res. 20, 55:1–55:21, URL: http://jmlr.org/papers/v20/18-598.html.

Finn, C., Abbeel, P., Levine, S., 2017. Model-agnostic meta-learning for fast adaptation of deep networks. In: International Conference on Machine Learning. PMLR, pp. 1126–1135.

Gilpin, L.H., Bau, D., Yuan, B.Z., Bajwa, A., Specter, M., Kagal, L., 2018. Explaining explanations: An overview of interpretability of machine learning. In: 2018 IEEE 5th International Conference on Data Science and Advanced Analytics (DSAA). IEEE, pp. 80–89.

Golub, G., Kahan, W., 1965. Calculating the singular values and pseudo-inverse of a matrix. J. Soc. Ind. Appl. Math. Ser. B: Numer. Anal. 2 (2), 205–224.

Grebhahn, A., Siegmund, N., Apel, S., Kuckuk, S., Schmitt, C., Köstler, H., 2014. Optimizing performance of stencil code with SPL conqueror. In: Proceedings of the 1st International Workshop on High-Performance Stencil Computations (HiStencils). pp. 7–14.

Guo, J., Czarnecki, K., Apel, S., Siegmund, N., Wąsowski, A., 2013. Variability-aware performance prediction: A statistical learning approach. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 301–311.

Guo, J., Yang, D., Siegmund, N., Apel, S., Sarkar, A., Valov, P., Czarnecki, K., Wasowski, A., Yu, H., 2018. Data-efficient performance learning for configurable systems. Empir. Softw. Eng. 23 (3), 1826–1867.

Gut, A., 2009. The multivariate normal distribution. In: An Intermediate Course in Probability. Springer New York, New York, NY, pp. 117–145. http://dx.doi.org/10.1007/978-1-4419-0162-0_5.

Ha, H., Zhang, H., 2019. Deepperf: performance prediction for configurable software with deep sparse neural network. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, pp. 1095–1106.

Heradio, R., Fernandez-Amoros, D., Galindo, J.A., Benavides, D., Batory, D., 2022. Uniform and scalable sampling of highly configurable systems. Empir. Softw. Eng. 27 (2), 1–34.

Jamshidi, P., Velez, M., Kästner, C., Siegmund, N., 2018. Learning to sample: Exploiting similarities across environments to learn performance models for configurable systems. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 71–82.

Jang, J.R., 1993. ANFIS: adaptive-network-based fuzzy inference system. IEEE Trans. Syst. Man Cybern. 23 (3), 665–685. http://dx.doi.org/10.1109/21.256541.

Jang, J., Sun, C.T., 1995. Neuro-fuzzy modeling and control. Proc. IEEE 83 (3), 378–406.

Kaltenecker, C., Grebhahn, A., Siegmund, N., Apel, S., 2020. The interplay of sampling and machine learning for software performance prediction. IEEE Softw. 37 (4), 58–66.

Kaltenecker, C., Grebhahn, A., Siegmund, N., Guo, J., Apel, S., 2019. Distance-based sampling of software configuration spaces. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, pp. 1084–1094.

Kingma, D.P., Ba, J., 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Knaus, W.A., Draper, E.A., Wagner, D.P., Zimmerman, J.E., 1985. APACHE II: a severity of disease classification system. Crit. Care Med. 13 (10), 818–829.

Kotsiantis, S.B., Zaharakis, I., Pintelas, P., et al., 2007. Supervised machine learning: A review of classification techniques. Emerg. Artif. Intell. Appl. Comput. Eng. 160 (1), 3–24.

Lattner, C., Adve, V., 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004.. IEEE, pp. 75–86.

Liu, H., Simonyan, K., Yang, Y., 2018. DARTS: Differentiable architecture search. CoRR abs/1806.09055 arXiv:1806.09055 URL: http://arxiv.org/abs/1806.09055.

Luketina, J., Berglund, M., Greff, K., Raiko, T., 2016. Scalable gradient-based tuning of continuous regularization hyperparameters. In: International Conference on Machine Learning. PMLR, pp. 2952–2960.

Ma, X.-J., Sun, Z.-Q., He, Y.-Y., 1998. Analysis and design of fuzzy controller and fuzzy observer. IEEE Trans. Fuzzy Syst. 6 (1), 41–51.

Membarth, R., Reiche, O., Hannig, F., Teich, J., Körner, M., Eckert, W., 2015. Hipa cc: A domain-specific language and compiler for image processing. IEEE Trans. Parallel Distrib. Syst. 27 (1), 210–224.

Mendel, J.M., 1995. Fuzzy logic systems for engineering: a tutorial. Proc. IEEE 83 (3), 345–377.

Merritt, L., Vanam, R., 2006. X264: A high performance H. 264/AVC encoder. [online] http://neuron2.net/library/avc/overview_x264_v8_5.pdf.

Metz, L., Poole, B., Pfau, D., Sohl-Dickstein, J., 2017. Unrolled generative adversarial networks. In: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings.

MGS, D., 2021. https://www.synapse-audio.com/.

Oh, J., Batory, D., Myers, M., Siegmund, N., 2017. Finding near-optimal configurations in product lines by random sampling. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 61–71.

Olson, M.A., Bostic, K., Seltzer, M.I., 1999. Berkeley DB. In: USENIX Annual Technical Conference, FREENIX Track. pp. 183–191.

Pardalos, P.M., Migdalas, A., Pitsoulis, L., 2008. Pareto Optimality, Game Theory and Equilibria, Vol. 17.

Pereira, J.A., Martin, H., Acher, M., Jézéquel, J.-M., Botterweck, G., Ventresque, A., 2019. Learning software configuration spaces: A systematic literature review. arXiv preprint arXiv:1906.03018.

Popovic, D., 2000. CHAPTER 18 - Intelligent control with neural networks. In: Sinha, N.K., Gupta, M.M. (Eds.), Soft Computing and Intelligent Systems. In: Academic Press Series in Engineering, Academic Press, San Diego, pp. 419–467. http://dx.doi.org/10.1016/B978-012646490-0/50021-4, URL: https://www.sciencedirect.com/science/article/pii/B9780126464900500214.

Real, E., Aggarwal, A., Huang, Y., Le, Q.V., 2019. Regularized evolution for image classifier architecture search. In: Proceedings of the Aaai Conference on Artificial Intelligence, Vol. 33. pp. 4780–4789.

Rumelhart, D., Hinton, G., Williams, R., 1986. Leaning internal representations by back-propagating errors. Nature 323.

Sarkar, A., Guo, J., Siegmund, N., Apel, S., Czarnecki, K., 2015. Cost-efficient sampling for performance prediction of configurable systems (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 342–352.

Semenick, D., 1990. Tests and measurements: The T-test. Strength Cond. J. 12 (1), 36–37.

Shannon, C.E., 1948. A mathematical theory of communication. Bell Syst. Tech. J. 27 (3), 379–423.

Siegmund, N., Grebhahn, A., Apel, S., Kästner, C., 2015. Performance-influence models for highly configurable systems. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 284–294.

Siegmund, N., Kolesnikov, S.S., Kästner, C., Apel, S., Batory, D., Rosenmüller, M., Saake, G., 2012a. Predicting performance via automated feature-interaction detection. In: 2012 34th International Conference on Software Engineering (ICSE). IEEE, pp. 167–177.

Siegmund, N., Rosenmüller, M., Kuhlemann, M., Kästner, C., Apel, S., Saake, G., 2012b. SPL Conqueror: Toward optimization of non-functional properties in software product lines. Softw. Qual. J. 20 (3), 487–517.

SPLTBooksNThesis, 2010. Software product line engineering - foundations, principles, and techniques-2005-BOOK. Softw. Test. Prod. Line Reusability.

Takagi, T., Sugeno, M., 1983. Derivation of fuzzy control rules from human operator's control actions. IFAC Proc. Vol. 16 (13), 55–60. http://dx.doi.org/10.1016/S1474-6670(17)62005-6, URL: https://www.sciencedirect.com/science/article/pii/S1474667017620056, IFAC Symposium on Fuzzy Information, Knowledge Representation and Decision Analysis, Marseille, France, 19-21 July, 1983.

Thereska, E., Doebel, B., Zheng, A.X., Nobel, P., 2010. Practical performance models for complex, popular applications. ACM SIGMETRICS Perform. Eval. Rev. 38 (1), 1–12.

Valov, P., Guo, J., Czarnecki, K., 2015. Empirical comparison of regression methods for variability-aware performance prediction. In: Proceedings of the 19th International Conference on Software Product Line. pp. 186–190.

Vargha, A., Delaney, H.D., 2000. A critique and improvement of the CL common language effect size statistics of McGraw and wong. J. Educ. Behav. Stat. 25 (2), 101–132.

Velez, M., Jamshidi, P., Sattler, F., Siegmund, N., Apel, S., Kästner, C., 2020. Configcrusher: Towards white-box performance analysis for configurable systems. Autom. Softw. Eng. 27 (3), 265–300.

Velez, M., Jamshidi, P., Siegmund, N., Apel, S., Kästner, C., 2021. White-box analysis over machine learning: Modeling performance of configurable systems. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, pp. 1072–1084.

Weber, M., Apel, S., Siegmund, N., 2021. White-box performance-influence models: A profiling and learning approach. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, pp. 1059–1071.

Williamson, D.F., Parker, R.A., Kendrick, J.S., 1989. The box plot: a simple visual method to interpret data. Ann. Intern. Med. 110 (11), 916–921.

Zhang, Y., Guo, J., Blais, E., Czarnecki, K., 2015. Performance prediction of configurable software systems by fourier learning (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 365–373.

Zhu, Y., Liu, J., Guo, M., Bao, Y., Ma, W., Liu, Z., Song, K., Yang, Y., 2017. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In: Proceedings of the 2017 Symposium on Cloud Computing. pp. 338–350.

Yufei Li is currently a Ph.D. candidate in School of Computer Science and Technology at Xidian University, China under the supervision of Prof Liang Bao. He received his bachelor's degree from Xidian University in 2020. His research interests include data mining, machine learning and software system performance prediction.

Liang Bao received the Ph.D. degree in computer science from Xidian University, P.R. China, in 2010. He is currently a professor with the School of Computer Science and Technology, Xidian University. His research interests include software architecture, cloud computing and big data.

Kaipeng Huang is currently an algorithm engineer of Alibaba Group, Beijing, China. He received his bachelor's degree from Xidian University in 2020. His research interests include machine learning, software system configuration tuning, computer vision and graph computing.

Chase Wu received the Ph.D. degree in computer science from Louisiana State University in 2003. He was a research fellow at Oak Ridge National Laboratory during 2003–2006, and an associate professor at University of Memphis during 2006–2015. He is currently a professor with the Department of Data Science at New Jersey Institute of Technology. His research interests include big data, distributed computing, computer networks, scientific visualization, and cybersecurity.

Xinwei Li is currently a Master's candidate in School of Computer Science and Technology at Xidian University, China, under the supervision of Prof Liang Bao. He received his bachelor's degree from Xidian University in 2021. His research interests include data mining, deep learning, and reinforcement learning.