Learning input-aware performance models of configurable systems: An empirical evaluation

Luc Lesoil, Helge Spieker, Arnaud Gotlieb, Mathieu Acher, Paul Temple, Arnaud Blouin, Jean-Marc Jézéquel

Learning Input-aware Performance Models of Configurable Systems: An Empirical Evaluation

Luc Lesoil[a], Helge Spieker[b], Arnaud Gotlieb[b], Mathieu Acher[a,*], Paul Temple[a], Arnaud Blouin[a], Jean-Marc Jézéquel[a]

[a] *Univ Rennes, Inria, INSA Rennes, CNRS, IRISA, France*
[b] *Simula Research Laboratory, Oslo, Norway*

Abstract

Modern software-based systems are highly configurable and come with a number of configuration options that impact the performance of the systems. However, selecting inappropriate values for these options can cause long response time, high CPU load, downtime, RAM exhaustion, resulting in performance degradation and poor software reliability. Consequently, considerable effort has been carried out to predict key performance metrics (execution time, program size, energy consumption, etc.) from the user's choice of configuration options values. The selection of inputs (e.g., *JavaScript* scripts embedded in a web page interpreted by *Node.js* or input videos encoded with *x264* by a streaming platform) also impacts software performance, and there is a complex interplay between inputs and configurations. Unfortunately, owing to the huge variety of existing inputs, it is yet challenging to automate the prediction of software performance whatever their configuration and input. In this article, we empirically evaluate how supervised and transfer learning methods can be leveraged to efficiently learn performance models based on configuration options and input data. Our study over 1,941,075 data points empirically shows that measuring the performance of configurations on multiple inputs allows one to reuse this knowledge and train performance models robust to the change of input data. To the best of our knowledge, this is the first domain-agnostic empirical evaluation of machine learning methods addressing the input-aware performance prediction problem.

Keywords: Performance Prediction, Software Variability, Input Sensitivity, Configurable Systems, Learning Models

1. Introduction

Most modern software systems are widely configurable, featuring many configuration options that users can set or modify according to their needs, for

*Corresponding author
Email address: mathieu.acher@irisa.fr (Mathieu Acher)

instance, to maximise some performance metrics (*e.g.*, execution time, energy consumption). As a software system matures, it diversifies its user base and adds new features to satisfy new needs, increasing its overall number of options. However, manually quantifying the individual impact of each option and their interactions quickly becomes tedious, costly and time-consuming, which reinforces the need to automate how to study and combine these options together. Software reliability can be largely degraded if inappropriate configuration options are selected or if ageing-related bugs such as configuration-dependent memory leaks remain undetected [74].

To address these issues, researchers typically apply *machine learning (ML)* techniques [21, 63] to learn performance models from the selection of configuration options and/or software modules. Conversely, with an accurate performance predictive model, it becomes possible to predict the performance of any configuration, to find an optimal configuration, or to identify, debug, and reason about influential options of a system [52, 26, 66, 21, 44, 45, 60, 61, 30]. A recent survey [52] synthesized the large effort conducted in the software engineering, software variability, and software product line engineering communities.

However, the performance variability of a given software system also obviously depends on its input data [2, 72, 38, 77, 8], e.g., a video compressed by a video encoder [38] such as *x264*, a program analyzed by a compiler [8, 11] such as *gcc*, a database queried by a DBMS [77] such as *SQLite*. All these kinds of inputs might interact with the configuration space of the software [11, 3, 42, 34]. For instance, an input video with fixed and high-resolution images fed to *x264* could reach high compression ratios if configuration options like `mbtree` are activated. The same option will not be suited for a low-resolution video depicting an action scene [38] with lots of changes among the different pictures, leading to different performance distributions, as for input videos in Figure 1[1]. The interplay between input data and configurations has not caught a great attention in the literature [52]. It is a threat of practical interests, since performance models of configurable systems or software product lines, can be inaccurate and deprecated whenever a new input data is processed.

Recent works empirically show the significance of inputs (also called workloads) when predicting the performance of configurable systems and software product lines. Pereira *et al.* [3] showed that the performance distribution of 1152 configurations of x264 heavily depends on the inputs (19 videos) processed. Practically, a good configuration can be a bad one depending on the processed video; some configuration options have varying influence and importance depending on videos; a performance prediction model can be inaccurate if blindly reused whatever the video. Recent empirical results of Mühlbauer *et al.* [42] and Lesoil *et al.* [34] over different software systems, configurations and inputs demonstrate that inputs can induce substantial performance variations and interact with configuration options, often in non-monotonous ways. As a result,

[1]Videos 1 and 2 are extracted from our dataset (Animation_1080P-5083 and Animation_1080P-646f)

inputs should be considered when building performance prediction models to maintain and improve representativeness and reliability.

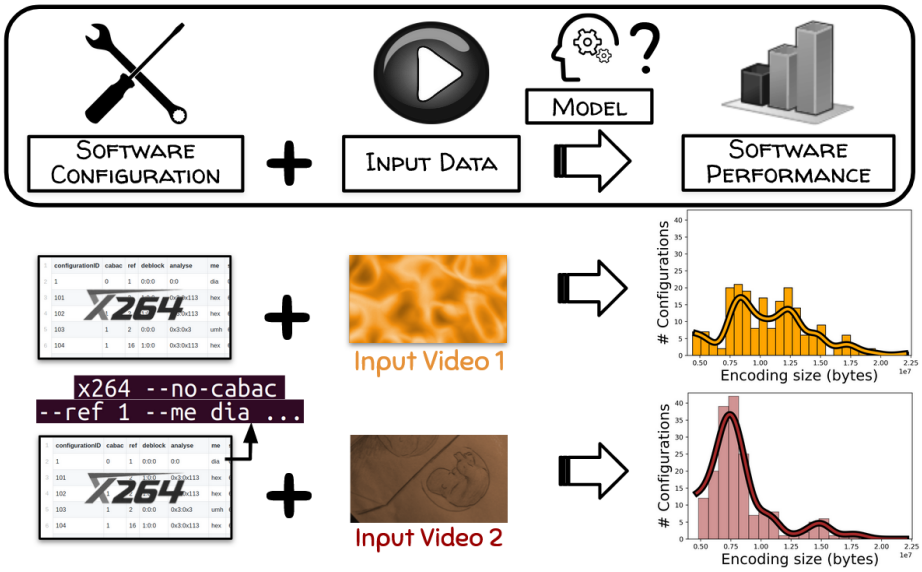

Figure 1: The *performance prediction problem*: how to predict software performance considering both configurations and inputs?

Measuring all configurations for all possible inputs of a configurable system is the most obvious path to resolve the issue. Given a potentially infinite input space, it is however either too costly and infeasible in practice or impossible. ML techniques are usually employed to measure only a sample of configurations and then use these configurations' measurements to build a performance model capable of predicting the performance of other configurations (i.e., configurations not measured before). However, these measurements are obtained on a specific input and are already costly to compute. Systematically repeating this process for many inputs would explode the budgets of end-users and organizations. The inputs add a new dimension to the problem of learning performance models. The combined problem space (configuration and input dimension) requires substantially more observations and measurements. It further increases the computational cost, since it requires running configurations over many input samples. The available budget end-users can dedicate to the measurements of configurations over inputs is limited by construction. *Hence, the challenge is to learn an accurate performance model, aware of configurations and inputs, with the lowest budget.*

Several input-aware approaches can be envisioned. The first is to learn from scratch a performance model, whenever an input is fed to a configurable system. Many works target the scenario where users build their own performance models for their own inputs and workloads [21, 60, 52, 3]. Unfortunately, users need to measure a sample of configurations for building a new prediction model each time a new input should be processed. The computational cost can be

prohibitive as it occurs in an *online* setting (at runtime). There is no reuse of past observations, knowledge, and performance models. Another second approach, at the opposite of the spectrum, is to pre-train in an *offline* setting a set of performance models for different inputs and configurations (as, *e.g.*,, proposed in [11]). The upfront cost can be high but can pay off since these models are systematically reused when a new input comes in. In the example of a configurable video encoder, performance models could be learned offline and reused each time a new video (input) is fed. The advantage is that users typically have only a small budget and cannot make additional measurements at run-time. The counter-part is that pre-trained models can have forgotten some inputs and be (much) less accurate than a performance model trained specifically on an input. At least, it is a hypothesis worth studying. In between, a third approach is to use both online and offline settings through transfer learning [52, 26, 4, 37, 68]. Certain transfer learning methods were originally intended to handle changes in computing environments, not actual inputs' changes, necessitating the development of new techniques that could effectively leverage specific input characteristics [25, 68, 26, 43, 67, 33, 24]. The principle is to adapt existing performance models (pre-trained in an offline setting over multiple inputs and configurations) thanks to additional measurements gathered over a specific input to process. The hope is to transfer the model with very few measurements at run time.

In this article, we study and compare the cost-effectiveness of these three input-aware approaches over a large dataset comprising 8 software systems, hundreds of configurations and inputs, and dozens of performance properties, spanning a total of 1,941,075 configurations' measurements. The distinction between offline and online learning has not received much attention yet (e.g., most works only apply either "offline" or "online" learning), certainly because of the lack of consideration of input data that can significantly alter the accuracy of predictive performance models of configurations, as is empirically shown in the rest of the article. We also consider the peculiarities of inputs (*i.e.*, their properties) to guide the transfer or reuse of pre-trained models. To the best of our knowledge, our work is the first domain-agnostic empirical evaluation of ML methods addressing the input-aware performance prediction problem in both online and offline settings.

Our contributions are as follows:

1. We perform an extended *comparative study* of performance model training approaches, including supervised learning as well as transfer learning. We also present their costs and error levels when addressing the performance prediction problem;
2. We provide guidelines to the user who faces a performance prediction problem and propose a learning-based solution based on her constraints and resources, *i.e.*, based on a trade-off between costs in offline and online settings.
3. We publish the code, the dataset, and the results of this paper online[2] as a

[2]See the companion repository at https://github.com/simula-vias/

basis for future work on predictive performance models.

2. Problem

As measuring the performance of each software configuration and input properties takes time and is computationally costly, we define three different scenarios based on user personas that have distinct levels of resources (*i.e.*, time and computation power). These scenarios lead to different learning strategies: a model pre-trained on multiple inputs used only as-is (*offline learning*); a model trained on a single input whenever an end-user processes an input (*supervised online learning*); a model pre-trained on multiple inputs but that will be adapted via *transfer learning* by an end-user. These three learning strategies have pros and cons and are further evaluated in the rest of the article (see Section 4).

2.1. User Persona (UP)

Performance prediction of software configurations has several interests for organizations, individual users and developers of configurable systems [21, 52, 25, 26, 68, 67]:

- prediction is interesting per se since users know the performance value they will get and this quantitative information is actionable. It allows users to determine whether it reaches a certain limit (or is within acceptable boundaries) and take informed decisions. For instance, users could know that the configuration *c1* of video encoding will be 56 s (less than 1 min, acceptable) while the configuration *c2* is 589 s (more than 10 min, unacceptable).

- prediction can help explore tradeoffs by (de)activating some options and see the concrete, quantitative effect on performance. Users are usually not optimizing w.r.t. one dimension (*e.g.*, execution time) but have also technical constraints related to output quality (*e.g.*, users do not want to alter much video quality and avoid using mbtree in x264) or other metrics in mind (*e.g.*, size of the output). Developers can also explore the configuration space to pinpoint inefficient performance and hopefully of some configurations;

- interpretable information can be extracted out of prediction models, like feature importance or interactions across features [40, 60, 12]. This information is useful for users and developers in charge of configuring, maintaining or debugging configurable software systems.

In these three cases, inputs can dramatically alter performance distributions and thus lead to inaccurate performance prediction if the specificities of input are not taken into account [41, 34]. Hence, UP can adopt different strategies to

`input-aware-performance-models`

adapt the performance model, with different computational costs and quality of performance predictions.

Let us consider UP $\mathcal{A}$ which is in a rush and has to quickly deploy configured software solutions to its customers. Luckily, some already trained models were made available online and can be used to make predictions, although they are not directly contextualized for UP $\mathcal{A}$'s application. Because fitting customer needs must be fast, UP $\mathcal{A}$ will directly reuse one of the prediction models as-is; *i.e.*, there will be no adaptation made whatsoever. Consequently, the computed performance predictions can be of poor quality, something that remains acceptable for UP $\mathcal{A}$. Typical examples of UP $\mathcal{A}$ include start-up company engineers or fast prototyping developers, and R&D software developers that can reuse pre-trained models and want to quickly reason over performance prediction of their configurable systems.

In contrast, UP $\mathcal{B}$ is not in a rush and has high expectations regarding customer satisfaction. $\mathcal{B}$ typically wants to retrieve the best predictions so that she/he can recommend configurations fitting the users' needs. To do so, $\mathcal{B}$ creates a prediction model on-demand specifically tailored to the provided input. Typically UP $\mathcal{B}$ corresponds to software engineers from large companies, which face high expectations in software capitalization and quality of service and can absorb the cost of building a performance model per input and workload.

Ultimately, UP $\mathcal{C}$ is committed to high standards, but $\mathcal{C}$ wants to quickly deliver high-quality software configurations. To do so, $\mathcal{C}$ wants to find a trade-off between the two previously-identified personas. $\mathcal{C}$ is likely to spend effort finding a pre-trained performance prediction model (or training it by using multiple inputs) so that the model can adapt to various cases. Note that in this case, even though most of the training cost is already high, $\mathcal{C}$ wants to tailor the model to customers' specific inputs to provide high-quality predictions. Examples of UP $\mathcal{C}$ include organizations or individuals capable of adapting pre-trained models based on the additional measurement of configurations over specific inputs.

2.2. Prediction Strategies

Based on these different UPs, we can distinguish several strategies for training and obtaining a performance prediction model.

Offline learning (rely on a pre-trained model.) UP $\mathcal{A}$ relies on a previously trained model, that is, a model which provides predictions before $\mathcal{A}$ comes up with an input sample. In this case, $\mathcal{A}$ is only using a pre-trained model and does neither fine-tune the model nor retrain it nor else apply any kind of transfer learning. The model is trained on a combination of multiple software configurations and multiple input samples. We follow an input-aware setup similar to the technique proposed in [11] for a compiler of the PetaBricks language. In practice (and it is the main advantage of the method), UP $\mathcal{A}$ does not need to compute new configurations' measurements. $\mathcal{A}$ simply passes an input sample to the trained model that provides a prediction by computing and leveraging input properties (see hereafter, in Section section 3.1.2). Eventually,

$\mathcal{A}$ sends the predictions to the customers. This "pre-trained model" setting advantageously reduces the computational load over UP $\mathcal{A}$ that does not need to measure configurations. To exploit the model, only the input's properties need to be computed and concatenated to the configurations' descriptions used for training. (Inputs' properties are specific to an application domain and many examples are given in the next section). The model can then be queried and all the retrieved predictions can be exploited directly. UP $\mathcal{A}$ does not control the training process and the quality of the training set (consisting of a selection of software configurations and inputs) meaning that the selections may be poorly distributed over the input space. Similarly, while querying the model, the provided input may be out of distribution, which can possibly result in weak predictions and choices.

Supervised online learning (train a model on demand.) Related to UP $\mathcal{B}$, the goal is to control as much as possible the quality of the prediction. In this setup, $\mathcal{B}$ does not rely on a pre-trained model (there is no cost in the offline setting) but rather builds a performance prediction model on-demand, in an online setting. For that, $\mathcal{B}$ has a pool of preselected software configurations ready to be executed with the desired inputs (*i.e.*, those coming from the customers). Another option is to have access to a software configuration sampling procedure. As soon as the input comes up, the first step consists in evaluating the performance of each selected software configuration using that input. Then, the prediction model is built and further predictions can be queried. Note that, unlike the previous strategy, only the desired input sample matters here, and thus there is no need to discriminate among inputs and compute input properties. The learned model is thus specific to the provided input, yet, predictions are expected to be more accurate than those from UP $\mathcal{A}$ (*i.e.*, as the input diversity dimension of the problem disappeared). Yet, it supposes that UP $\mathcal{B}$ has enough resources available to perform both the measurements and the training of the system before being able to provide predictions. Thus, in an online setting where predictions must come almost instantaneously, this strategy is probably inadequate. Also, the model is trained on-demand and for a specific input only. If multiple requests from different customers arrive at the same time, then the model will be re-trained again and again. This prevents sharing knowledge from different models and prevents capitalizing on previous training. Traditional statistical learning techniques (*e.g.*, [21, 60, 52, 3]) can be used. It boils down to address a regression problem each time a new workload or input is considered.

Transfer learning (adapt pre-trained model). Finally, UP $\mathcal{C}$ wants to answer best to his customers but cannot afford on-demand full training prediction models. A suitable strategy would be to leverage the bigger cost that can be left to organizations that can provide general prediction models, and adapt, on the fly, a model to the specific input that is provided. This way, the out-of-distribution and quality of the selected software configurations and inputs for training problems would be mitigated. The adaptation would then require the general model to be retrieved so that parameters can be modified. The idea is not to retrain completely though. Two different methods can be

Approach	Description of the approach	Offline cost (Offshore Organization)	Online measurement cost (User)	Input properties	User Persona
Supervised online learning	Train performance model on demand, from scratch, each time a new input is fed to the configurable system	None	High	No	$\mathcal{A}$
Offline learning	Use a pre-trained model over measurements of multiple configurations and inputs. Input properties are used to make the prediction (in an online setting).	High	None	Yes	$\mathcal{B}$
Transfer learning	Adapt a pre-trained model for a new targeted input. It requires to gather fresh measurements of some configurations over the input (in an online setting).	Medium	Medium	Yes	$\mathcal{C}$

Table 1: Comparison of the different approaches to learn input-aware performance models of configurable systems.

used to adapt models, the first one is fine-tuning, such that the model weights can be adjusted quickly to the incoming input; the second one is "transfer learning" [52, 26, 4, 37, 68] to find a transformation between the data distribution the model was trained for, and the data provided by $\mathcal{C}$'s customers. Transfer learning can be applied for several use cases without necessarily making changes to the original model, thereby providing additional flexibility.

In the context of this paper, we focus on transfer learning as it addresses the out-of-distribution problem. Some transfer learning techniques have not been designed to operate over actual inputs' changes, but rather changes of computing environments [25, 68, 26, 43, 67, 33, 24] (*e.g.*, hardware or versions). Hence, we had to design transfer learning techniques capable of leveraging the specifics of inputs. The idea is to compute a transformation allowing the transfer of the prediction capabilities from the inputs used for training to the one that comes from $\mathcal{C}$ and vice-versa. For that, $\mathcal{C}$ relies on a pre-trained model as does UP $\mathcal{A}$. Yet, as the input comes for the customer, the input's properties are sent to the model for prediction and in the meantime, $\mathcal{C}$ actually measures the performances of the configurations that are used for training on the new incoming input. Measured performances and predictions can then be compared to retrieve a mathematical transformation that can be applied to the pre-trained model to minimize prediction error. As said previously, the main advantage is to start from a rather general prediction model and simply adapt it. Yet it requires retrieving the model along with the software configurations that were used for training. In the end, the effort is split among online and offline settings, and between (1) the creation of the original, general model and (2) UP $\mathcal{C}$ that has to compute the adaptation. Table 1 sums up how efforts are split between online and offline settings, UPs, and model providers for the three different prediction strategies (offline learning, supervised online learning, and transfer learning).

2.3. Research Questions (RQs)

Accounting for the variety of UPs and prediction strategies, we spell out the following three research questions (RQs):

RQ$_1$. How do different machine learning algorithms compare for establishing a relevant performance prediction model? The production

of a relevant predictive performance model involves the selection of the most appropriate algorithm for that task. To address this question, we quantify the errors and the benefits of tuning hyperparameters of several relevant algorithms used in the literature. We also compare these results with a non-learning approach to the problem, used as a baseline for comparison.

RQ_2. How to select an appropriate set of inputs for training a performance prediction model? Prior to the prediction, we have to select a list of inputs whose measurements will form the training dataset. We call this process *input selection*. RQ_2 investigates what choice of input selection technique (e.g., all inputs, random selection of inputs, most diverse inputs) leads to the best results in terms of performance prediction. Depending on the offline budget, we propose and compare various input selection techniques.

RQ_3. How does the number of measured configurations affect the performance prediction models? Since inputs and configurations interact with each other to change software performance, input selection is sensitive to the sampling of configurations *i.e.*, in the way we select the configurations used to train the model. To answer RQ_3, we train performance models fed with different numbers of inputs and configurations.

3. Experimental protocol

To respond to these three research questions, we designed an experimental protocol based on the following data (Section 3.1) and prediction models (Section 3.2). This section also provides details about RQ1 (Section 3.3), RQ2 (Section 3.4) and RQ3 (Section 3.5).

3.1. Data

3.1.1. Dataset

We reused the measurements from 8 configurable systems and their inputs, as they are introduced in [34] and referenced in the companion repository[3]. Different elements are shown in Table 2. The total number of measurements taken for a software system is equal to the number of configurations multiplied by the number of inputs and the number of performance properties. For instance, 201 configurations of *x264* have been systematically measured along five performance properties and using 1397 videos coming from the YouTube **U**ser **G**eneral **C**ontent (YUGC) dataset [71], for a total of 1,403,985 measures for *x264*.

3.1.2. Using input properties to discriminate inputs

To differentiate the inputs directly in the learning process, we computed and added input properties [11] that describe specific characteristics of input data. The input properties are preprocessed into an input feature vector and

[3]Our data and measurement protocol are available and open

Table 2: An overview of the considered systems in their number of configurations (#Configs), configuration options (|Options|), inputs (#Input), and input properties (|Input|). It should be noted that some options have numerical values. The last column states the performance properties that were measured.

System	#Configs	|Options|	#Inputs	|Input|	Performance
gcc	80	5	30	7	size, ctime, exec
imagemagick	100	5	1000	5	size, time
lingeling	100	10	351	4	#conf, #reduc
nodeJS	50	6	1939	6	#operations/s
poppler	16	5	1480	6	size, time
SQLite	50	3	150	8	15 query times q1-q15
x264	201	23	1397	7	size, time, cpu, fps, kbs
xz	30	4	48	2	size, time

concatenated with the configuration features. Input and configuration options jointly form the feature vector that is passed as the input to the machine learning model.

The encoding into a single vector allows statistical learning techniques to operate over a unified encoding to predict performance. Though both are encoded as features, configuration options and input characteristics refer to and describe different entities (*i.e.*, inputs and software configurations respectively). Configuration options directly come from the software and the development activity. They are used to differentiate every single configuration that can be built. Options have been made explicit, traced back and implemented in the code by developers and/or domain experts. On the other hand, characteristics describing inputs are not necessarily made explicit and usually require domain experts to model features that should be both descriptive about the content, helpful for discriminating one input from the other, and also informative for predicting performance. They might not be as differentiating as the ones for configurations as different inputs may result in the same characteristics. In the end, paired with the configuration options, we expect that this feature vector allow us to observe very similar performances from the systems. Some learning approaches can leverage these input properties as predictors (or features) when building or adapting prediction models. Based on domain knowledge, we list hereafter what input properties have been computed for the different sorts of inputs (see also Table 2): for the *.c* scripts compiled by *gcc*, the size of the file, the number of imports, methods, literals, *for* and *if* loops and the number of lines of code (LOCs); for the images fed to *imagemagick*, the image size, width and height, category (describing the image content, e.g., ostrich, dragonfly, or koala), and its averaged (*r, g, b*) pixel value; for SAT formulae processed by *lingeling*, the size of the *.cnf* file, the number of variables, *or* operators, and *and* operators; for the test suite of *nodejs*, the size of the *.js* script, the LOCs, number of functions, variables, *if* conditions, and *for* loops; for the *.pdf* files processed by *poppler*, the page height and width, the image and pdf sizes, the

number of pages and images per input pdf; for databases queried by *SQLite*, the number of lines for eight different tables of the database; for input videos encoded by *x264*, the spatial, temporal, and chunk complexity, the resolution, the encoded frames per second, the CPU usage, the width and height of videos [4]; for the system files compressed by *xz*, the format and the size.

3.1.3. Separation Training-Test

We randomly split each set of configurations into a training set and a test set. We repeat this with varying proportions of configurations in the training set – we start with 10% of configurations dedicated to training and then 20%, 30%, ..., up to 90%. The training set is available for data selection and training of the models, whereas the test set is purely dedicated to evaluating the trained models. To avoid biasing the results with different samplings of configurations, we fix the random seeds so the different techniques work with the same training and test sets. Note that every training is done independently from one model to another and from scratch so that all training procedures do not share any information and start from the same point.

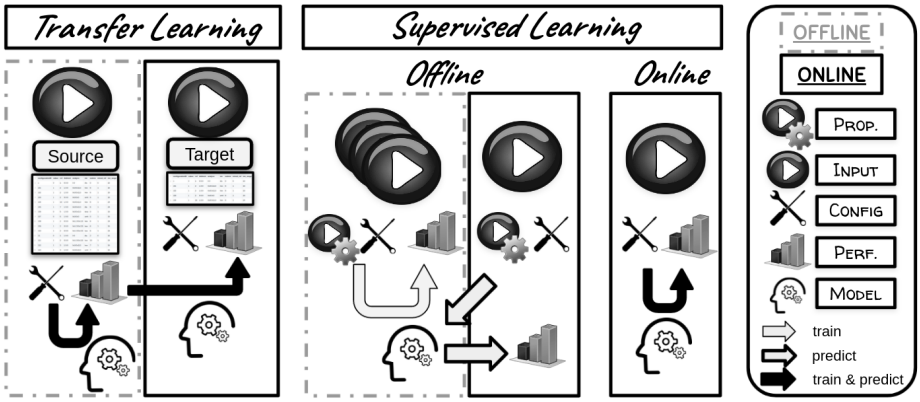

Figure 2: Learning approaches to address the performance prediction problem

3.2. Performance Prediction Models

In this section, we define the different predictive performance models used in the rest of the paper.

3.2.1. Strategies and baselines

In addition to the three learning strategies that we described in Section 2.2 (supervised online learning, offline learning, transfer learning), we consider a baseline that does not learn the specifics of the inputs. The approach, called

[4] For *x264*, input properties were already computed in [71]

Average, simply computes the average of configurations' measurements observed on the input. This is supposed to mimic the behaviour of an end-user that might consider first a configuration of a system (e.g., the default configuration [76]) and then slightly explore the configuration space to have an average of the performance values. This way, the end-user may measure various configurations and approximate the trend of performance values. Average is a better approximation than the systematic use of a single point (*e.g.*, default value) when it comes to predicting the performance of any configuration, especially when inputs change.

An advantage of this approach is that there is no cost for an offshore organization (*i.e.*, the ones that provide pre-trained models). Instead, the baseline applies on demand for every new input (in an online setting). Once the input is received, a set of randomly sampled configurations is executed over the input to retrieve performance measures. The average of these performances is returned, and these are the values that are communicated to the users in the sense that they can expect such performance on average. Obviously, such an averaged configuration does not account for possible variations in the performance of configurable systems. In particular, the average value remains fixed regardless of the configuration we wish to predict performance for, which could possibly lead to a high degree of inaccuracy.

3.2.2. Learning Algorithm

Until now, we described the learning strategies but did not mention which learning algorithms we were using. We chose 4 different algorithms, namely:

1. OLS Regression [57] from Scikit-learn [51], estimating the performance value with a weighted sum of variables. It is not designed for complex cases;
2. Decision Tree [55] from Scikit-learn, using decision rules to separate the configurations into sets and then predicting the performance separately for each set;
3. Random Forest [49] from Scikit-learn, an ensemble algorithm based on bagging, combining the knowledge of different decision trees to make its prediction;
4. Gradient Boosting Tree [18] from XGBoost [10], also derived from Decision Tree. Unlike Random Forest training different trees, gradient boosting aims at improving one tree by specialising its rules of decision at each step.

These are prevalent algorithms and are commonly used for tabular data. We do not include deep learning or neural networks since we do not have lots of measurements in an online setting, which is usually a requirement, and since these methods are still commonly outperformed by random forests or gradient boosting [58, 20]. We train each machine learning algorithm with its default parameters, if not stated otherwise. The optimal parameters for an algorithm are dependent on the dataset, its size, and the performance property. As a result, they are necessary to be adjusted on a per-case basis.

3.3. Selecting Algorithms (RQ_1)

First, we address RQ_1 - *How do different machine learning algorithms compare for establishing a relevant performance prediction model?*. We decompose it into three parts, that jointly answer the research question.

3.3.1. Why using machine learning?

Our first goal is to state whether machine learning is suited to address the performance prediction problem. To assess the benefit of using machine learning, we compare the *Average* baseline to different learning algorithms. We implement them in an online setting, *i.e.*, we use the supervised online approach; given the input of the user, we want to estimate its performance distribution. This is a prediction for one input at a time.

3.3.2. Which machine learning algorithm to use?

This evaluation is also the opportunity to compare these learning algorithms and search for the one that outperforms the others. We also study the evolution of their prediction errors with increasing training sizes. We consider those listed in Section 3.2.2. After training them on the training set, we predict the performance distribution of the test set and compute the prediction error. We repeat it for all combinations of system, inputs, and performance properties. As prediction error, we rely on the Mean Absolute Percentage Error [40]. In Figure 3, we display the average MAPE values for various training sizes.

3.3.3. What is the benefit of hyperparameter tuning?

Finally, we want to estimate how much accuracy we could expect to gain when we tune the hyperparameters of learning algorithms. To do so, we rely on a grid search [5] for hyperparameter tuning. We compute the training duration and prediction errors of these algorithms, with and without tuning their hyperparameters, and report the average difference for both.

3.4. Selecting Inputs (RQ_2)

RQ_1 studies the effectiveness of machine learning. However, different ways of selecting inputs during the data collection or a different number of inputs could alter the accuracy of the final performance model. Then, we address RQ_2 - *How to select an appropriate set of inputs for training a performance prediction model?* We separate this into three questions.

3.4.1. How many inputs do we need to learn an accurate predictive performance model?

From the perspective of a user in charge of the training, adding an input to measure incurs a computational cost and should be justified by an improvement of the model. So, what is the effect of adding new inputs on the accuracy of predictive performance models? How many inputs do we need to reach a decent level of accuracy? We aim at minimising the number of inputs used in the training while maximising the accuracy of the obtained model. To do so, in

this part of the evaluation, we train different performance models using various numbers of inputs and compare their prediction errors.

Then, once the number of inputs is fixed, we search if there is any benefit in precisely and methodologically selecting the inputs for data collection and model training. Does it bring any improvement over the random selection of inputs? Do the different inputs used in the training set change the final predictive performance model accuracy? Depending on the offline budget of the user, we have different objectives.

3.4.2. How to select the input data for an offline setting?

If the offline budget is restricted, a.k.a. the *offline setting*, the goal is to constitute a representative set of inputs to learn from, in order to build a performance model that will generalise as much as possible. We care about selecting diverse and representative inputs, in order to predict accurate results whatever the input data. For this setting, we compare the following input selections:

1. Random - Using a uniform distribution to decide which inputs should be included in the training;
2. K-means - Based on input properties, we apply K-means clustering to differentiate clusters of inputs with distinct characteristics. To increase the diversity in the selection, we pick the inputs closest to the center of the clusters;
3. HDBScan - Similar to the previous technique but with another clustering algorithm, namely the HDBScan [6], using density-based instead of means-based[5];
4. Submodular (Selection) - This technique computes a similarity matrix between the different inputs of a software system and optimises facility location functions [31] to choose a representative set of inputs[6].

For this *offline setting*, we implement the supervised offline approach with a Gradient Boosting (best in Section 4.1.2). Once the input selection technique chose the input, we include all related measurements in the training set. The test set is then composed of the measurements of all other inputs, not selected. To avoid biasing the machine learning model with different scales of performance distribution, we choose to standardise [13] all performance properties. But it has a drawback: since their values are close to zero, it artificially increases the MAPE values. To overcome this, we switch to the Mean Absolute Error (MAE) [40]. Since the performance property is standardised, we assume that models with MAE values inferior to 0.2 are good – way better than the expected average distance $\frac{2}{\pi} \simeq 1.13$ [64] between two points selected uniformly. We repeat the prediction 20 times and depict the average MAE (y-axis, left) in Figure 4 for different input selection techniques (lines) and number of inputs (x-axis) on a per-system basis. We added the number of training samples (y-axis, right). Except for *gcc* and *xz*, x-axis are in log scale.

[5] We rely on this implementation: https://hdbscan.readthedocs.io/ [39]
[6] We rely on this implementation: https://apricot-select.readthedocs.io/ [56]

3.4.3. How to select the input data for an online setting?

If the offline budget is low, a.k.a., in the *online setting*, then we must be efficient and focus on predicting the performance distribution for the current input of the user. For this setting, we implement a transfer learning approach: the input of the user becomes the target input, and the candidate input becomes the source input. In this online setting, the goal of input selection becomes to find one good source input that is as close as possible to the current input of the user - in terms of characteristics and performance. The choice of a good source should improve the performance prediction of the transfer learning approach [32]. We propose the following input selections:

1. Random - Same baseline as in the offline setting;
2. Closest (Input) Properties - Two inputs sharing common characteristics might also share common performance distributions. Following this reasoning, to improve the performance prediction, we have to select an input whose properties are similar to the current input's properties. To do so, we compute the MAE between the properties of the current input and the properties of all the candidate inputs. We pick the input obtaining the smallest MAE value;
3. Closest Performance - We use the few measurements already measured on the current input. For these, we compute the Spearman correlation [27] between the performance distribution of the current input and all the candidate inputs. Finally, we select the candidate with the highest correlation;
4. Input Clustering (& Random) - With the help of a K-Means algorithm, we form different clusters of inputs based on their properties. We randomly pick a candidate input in the cluster of the current input.

For this question, we use Gradient Boosting. We display the median MAPE results over 10 predictions for all software systems, performance properties and number of inputs in Table 3.

3.5. Selecting Configurations (RQ_3)

In this section, we vary the budgets of (1) inputs and (2) configurations used to constitute the training set fed to the model. RQ_3 - *How does the number of measured configurations affect the performance prediction models?* In this research question, we compare the accuracy of models according to the numbers of inputs and configurations used during their training, answering these questions:

3.5.1. What is the best tradeoff between selecting inputs and sampling configurations?

For a fixed number of configurations, we study the evolution of the accuracy with the number of considered inputs. Is it better to measure lots of configurations or numerous inputs? Since the evaluation is designed to improve the generalization of the model, it mostly relates to models trained in an offline setting. Therefore, we implement the offline approach, fix the algorithm to Gradient Boosting and use the random baseline as input selection technique. We repeat the experiment 20 times. In Figure 5, we depict the MAE (colour) for various numbers of inputs (x-axis) and configurations (y-axis).

We are then interested to look at the impact of the number of configurations on the accuracy of three input-aware approaches (transfer learning vs supervised online; offline learning vs supervised online).

3.5.2. Is it better to use the transfer learning or the supervised online approach?

Depending on the online budget of the user, it can be worth (or not) to transfer the knowledge from one input to another. If the online budget is high, we guess there is no need to transfer, *i.e.*, we do not use transfer learning. If this budget is low, we can benefit from the transfer learning approach. How much online budget justifies the decision for transfer learning? To answer this, we implement both approaches with different numbers of configurations on the target input. We use Gradient Boosting and predict the performance spanning all systems and performance properties. Due to outliers drastically increasing the average value, we compute the median MAPE instead of the average. In Figure 6, we display the MAPE value (y-axis) for different budgets of configurations (x-axis).

3.5.3. Should we train performance models offline or online?

To answer this question, we compare the supervised offline approach to the supervised online approach w.r.t. MAE. We predict performance implementing both approaches and compute the MAE value for different training size – varying from 10% to 90% of the configurations available per input. Figure 7 displays the results for both approaches, *i.e.*, the average MAE on all software systems and performance properties.

4. Evaluation

We report the results following the protocol of Section 3.

4.1. Selecting Algorithms (RQ_1)

4.1.1. Is using ML relevant in the context of predictive performance modelling?

Figure 3 shows the benefits of using ML as compared to the *Average* baseline. It appears that ML techniques clearly outperform the average performance value predicted by the baseline for all training sizes. The key indicator to study is the evolution of errors with increasing training proportions; while the baseline's accuracy does not progress with additional measurements, the learning algorithms improve their prediction, from 12% to 3% error.

4.1.2. Which ML algorithm to use?

While ML is generally beneficial, the prediction quality varies between the different algorithms. For instance, OLS Regression generally leads to bad results, *e.g.*, 22% of error with 10% of the configurations. Unlike the OLS regression, tree-based learning algorithms take advantage of the addition of new measurements. For a budget of 50% of the configurations, their median prediction goes under the 5% of error, which is encouraging. This result of 5% is

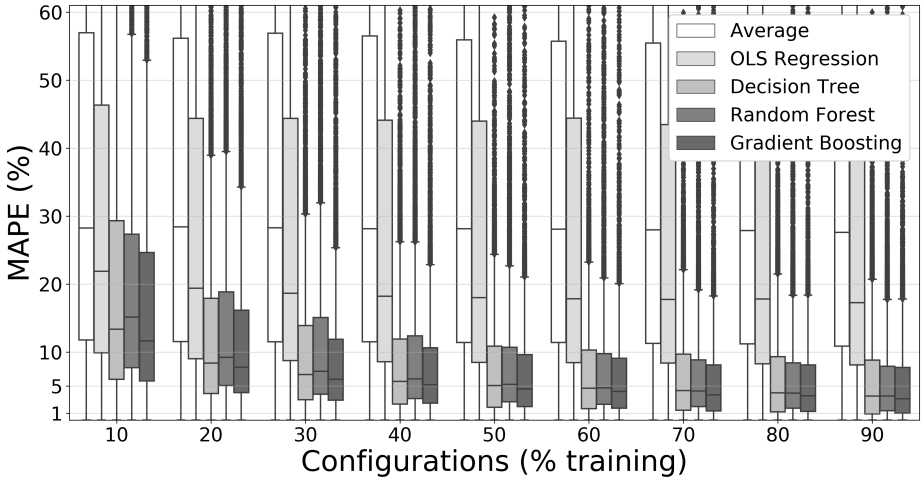

Figure 3: Which (learning) algorithm to use?

only valid in average; the prediction will be better for a few software systems, *e.g.*, *imagemagick* or *x264*, but does not hold for others, *e.g.*, *lingeling* or *poppler*. Though there is no big difference between these three learning algorithms, we observe slightly better predictions for Random Forest compared to Decision Trees, and for Gradient Boosting compared to Random Forest.

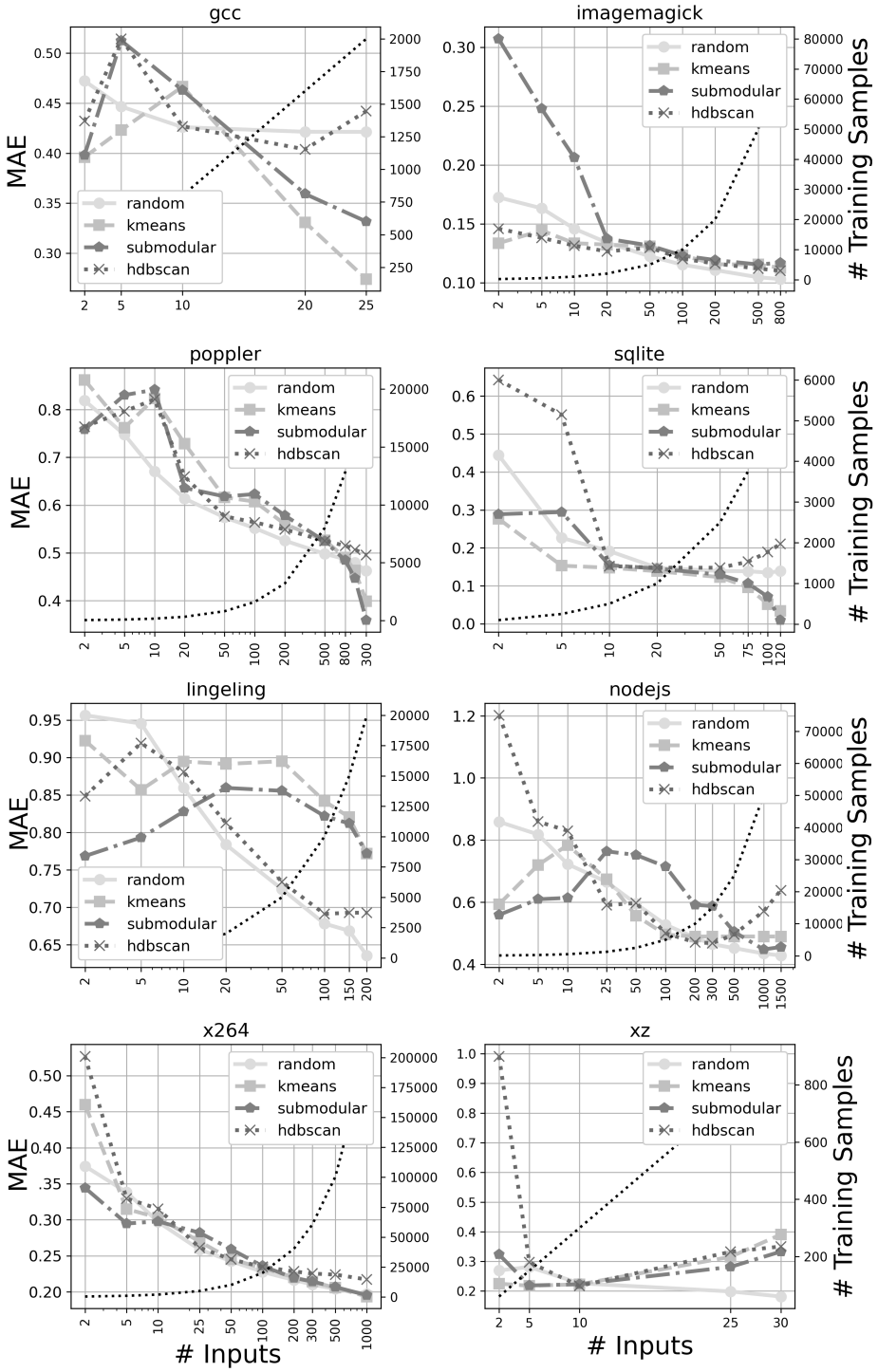

Figure 4: Offline setting with Gradient Boosting (best) and different input selection techniques: Influence of input selection and the number of inputs (lower MAPE = better).

4.1.3. What is the benefit of hyperparameter tuning?

Our results found that hyperparameter search improves the MAPE on average by $8 \pm 16\%$ within a range of $3 \rightarrow 37\%$ but also requires on average 120 times more training time when doing a grid search of estimator parameters. It should be noted that this is an improvement in percent, not percentage points. While the exact overhead of hyperparameter search is dependent on the number of configuration options of the model and the search method, we note that the overall benefit on the datasets is limited. This is especially confirmed by the observation that the best-found hyperparameters were changing depending on both the system and the size of the training dataset. For simplicity of the setup, the rest of the evaluation uses the default parameters of each model, if not otherwise noted.

> ***RQ*$_1$** Machine learning is well-suited to address the predictive performance modelling problem over configurations and inputs. It outperforms the *Average* baseline as it makes more accurate predictions. Our results also show that using tree-based learning should be favoured over OLS Regression. These tree-based algorithms reach decent levels of errors with reasonable online budgets, between 5% and 10% relative error for most of the cases, and potentially improved by 8% when tuning hyperparameters.

4.2. Selecting Inputs (RQ_2)

4.2.1. How many inputs are needed to learn an accurate predictive performance model?

Measuring inputs differs across software systems: measuring 5 inputs represents $5 * 201 = 1005$ configurations for *x264* but only $5 * 30 = 150$ for *xz*. Figure 4 shows that the performance model reaches its lowest error threshold when considering about 20 inputs. Results are thus easier to interpret on systems with more inputs, as compared to *gcc* and *xz*. There are however more difficult cases, *e.g.*, *Node.js* and *poppler*, in our experiment that might require more inputs to improve the accuracy of the prediction. To get a consistent prediction, we recommend the user to measure at least 25 inputs with a sufficient number of configurations per input.

4.2.2. How to select the input data for an offline setting?

In an offline setting, we seek to train a generalized model for all inputs; the selected inputs are supposed to be representative of the diverse set of inputs to be expected during deployment. Figure 4 presents our results. As expected, with an increased number of selected inputs, the influence of the input selection decreases. The input selection is especially important for small numbers of inputs. The evaluation shows that our techniques kmeans, submodular and hdbscan fail to beat the random baseline when selecting the inputs prior to

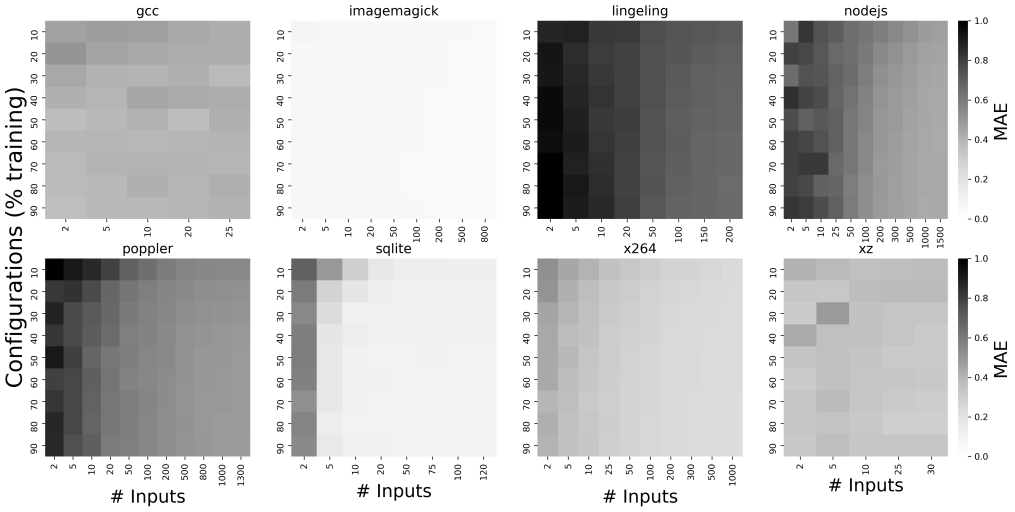

Figure 5: Offline setting with Gradient Boosting (best) and random baseline as input selection technique: Error of performance models (lower MAE = brighter color = better) according to different budgets (inputs & configurations). For imagemagick a consistent avg. MAE of 0.06 is observed.

the training of the model. There is no clear outperforming technique of input selection. These results could be explained by multiple factors: (1) the input properties processed by the input selections are not sufficient to differentiate the inputs (2) our baseline focuses on selecting different profiles of inputs, while it may be more efficient to select a set of average-like inputs. Out of this result, we advise keeping it simple and adopting the random baseline.

4.2.3. How to select the input data for an online setting?

In an online setting, we specifically build a model for the current input and select a similar input to transfer the knowledge from. Table 3 details the results for the different input selections, as the median MAPE results over 10 predictions for all software systems, performance properties, and the number of inputs. Unlike the offline setting, our input selection techniques were able to beat the random baseline, the best input selection technique being the Closest Performance with an average MAPE around 3.8, followed by the Closest Properties (4.2) and the Input Clustering (4.7). Wilcoxon signed-rank tests [59] (with significance levels at 0.05) confirm that predictions related to different input selections are significantly different from those using the random baseline: $p = 0.0$ for Closest Performance, $p = 1 * 10^{-184}$ for Closest Properties and $p = 1 * 10^{-27}$ for the Input Clustering. Therefore, and to continue to provide guidance for users, we advise using the Closest Performance to select the input in an online setting. But beyond the raw comparison of error values, beating the Random baseline with the Closest Property technique is a strong result. Empirically, it validates that these input properties are valuable to compute and should be

Table 3: Online Setting - Influence of input selection

Input Selection	MAPE (%)	Training Time (sec)
Random	5.22	0.02
Closest Properties	4.17	0.05
Closest Performance	3.82	0.07
Input Clustering	4.70	0.02

included in the models to improve the prediction. Besides, the evaluation shows that training times are negligible.

> ***RQ₂*** In an offline setting, the results show that diversification of inputs (rather than configurations) should be prioritized by using uniform distribution to select inputs, *i.e.*, the random baseline. In an online setting, the performance correlations technique gains about 1.4 point of error compared to a random selection of inputs. Our results empirically validate the important role of input properties when predicting software performance, *i.e.*, pretrained performance models (offline) can be reused under the condition input properties are computed and leveraged online.

4.3. Selecting Configurations (RQ_3)

4.3.1. What is the best trade-off between selecting inputs and sampling configurations?

According to Figure 5 results, diversifying the inputs is more effective than selecting different configurations to train an input-aware performance model. But for a fixed budget of inputs, there is a slight improvement in accuracy when increasing the number of configurations. As a result, both should be combined to obtain the best possible model. Overall, the ideal budget – both in terms of inputs and configurations – highly depends on the expected level of errors combined with the difficulty of learning a predictive performance model for the software under test. For instance, predicting the performance of *imagemagick* is relatively easy, with an average MAE value at 0.06. For this software system, picking 25 inputs is almost already a waste of resources, we do not need that much data - 5 inputs is already enough with 30% of the configurations. Other systems are harder to learn from *e.g.*, *lingeling* with an average MAE of 0.76. For these, we recommend increasing the number of inputs and configurations. The MAE obtained on the training set should be used as a proxy to estimate the difficulty of predicting the performance of the software under test. The greater its value, the greater the budget needed to learn an accurate model.

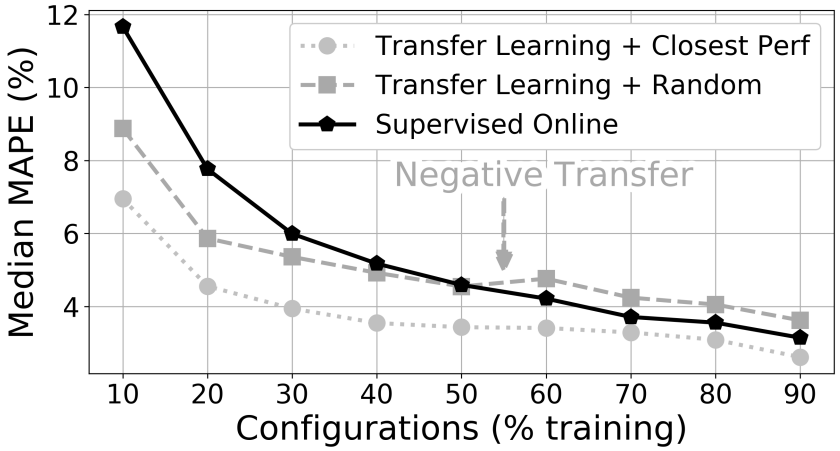

Figure 6: Transfer Learning VS Supervised Online Learning.

4.3.2. Is it better to use transfer learning or a supervised online approach?

With the input selection technique set to Closest Performance and whatever the percentage of configurations used in the training, the transfer approach always outperforms the supervised online approach, as shown in Figure 6. This strong result demonstrates the importance of capitalizing on the existing measurements, measured in an offline setting. Hence, if we are able to create representative sets of inputs for each software system (thanks to Closest Performance), then transfer learning becomes the best approach to use.

However, the strategy of picking the source input among all the inputs of our dataset is not always possible – it requires a high offline budget. We consider another scenario where we put ourselves in the situation of a user with a low offline budget *i.e.*, not able to select an ideal source input. We thus add the comparison of transfer learning with a random input selection baseline. Even in this case, we still outperform the supervised approach for less than 55% of the configurations. But this transfer with random selection has an expiration date; after a training proportion of 55% - represented by the arrow on the graph, it leads to negative transfer: the added measurements (of the source) become noisy data interfering with the training of the target model.

4.3.3. Should we train performance models offline or online?

Figure 7 shows the evolution of the two supervised approaches, offline and online, depending on how many configurations are used as part of the training. The first point we notice is the slow progression of the supervised offline approach, only from 0.37 to 0.30 between 10% and 90% of configurations.

A possible explanation is that it is so hard to generalize over the input dimension that it hides the benefit of adding configurations. While when considering only one input at a time, this difference in performance distributions does not bother the training of the machine learning model. Nevertheless, comparing the

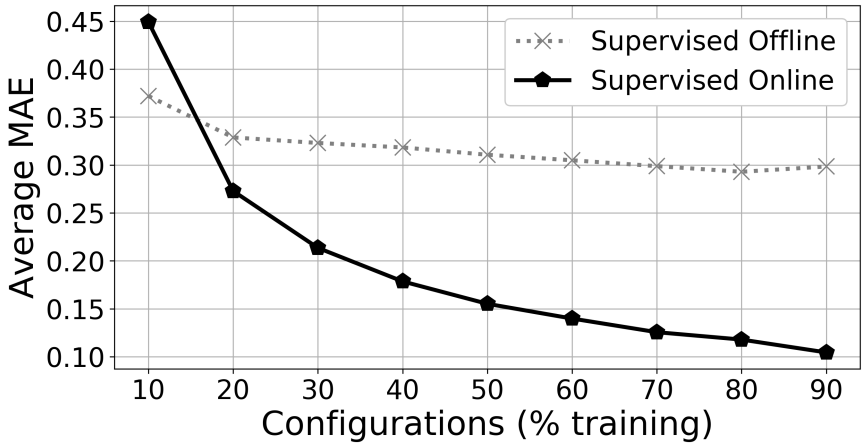

Figure 7: Offline Learning VS Supervised Online Learning.

raw numbers provides a straight answer to the initial question: unless the online budget is really low, if the choice between a supervised offline and a supervised online approach occurs, one should definitely prefer the online approach. But this finding has to be contextualized w.r.t. cost and effectiveness. From the point of view of the final user, the computation of measurements in an online setting to build a performance model will always last longer than offline prediction, using an already-trained model. Yet, when users have an online budget (even a small one), they should always prefer the online approach compared to the offline approach. Stated differently, the supervised offline approach should be adopted for lack of a better solution, as the last approach to implement when users cannot afford to measure configurations in an online setting.

> ***RQ*$_3$** Supervised online learning quickly outperforms the offline learning version. With more than 20% of configurations for training, online learning already shows a lower MAE on average than its offline counterpart. We come up with the following high-level recommendation: 1) if the online budget is low, transfer learning should be used; 2) with a substantial online budget (55% of the configurations in our experiments), it might be better to use the supervised online approach; 3) offline learning without transfer should be avoided, except for very small online configuration budgets.

5. Discussion

Each part of the evaluation provides a recommendation for the three user profiles defined in Section 2.1, depending on the trade-off between their offline

and their online budgets. This discussion summarizes our findings while answering RQ_1 to RQ_3 and turns these findings into recommendations and actionable rules, thus guiding the user to solve the performance prediction problem. **How to help users predict their software performance, whatever be the input data and their configuration?**

Depending on the available online budget, we distinguish the following cases:

- If the user has a **high online budget** (*e.g.*, user persona $\mathcal{C}$) we recommend using the supervised online approach (Sec. 4.3.3) with a Gradient Boosting Tree implementation (Sec. 4.1.2) and tuned hyperparameters (Sec. 4.1.3). In that case, users can expect low prediction errors. Assuming configurations' measurement has been collected, learning a performance model from scratch for each unique input is the ideal scenario, as is the case with a large online budget.
- If the user has a **low online budget**, we can also recommend the supervised online approach, but cannot promise outstanding performance estimations. Hence, if a representative set of inputs has already been measured *i.e.*, with a big offline budget (as for user persona $\mathcal{B}$) we rather recommend using the transfer learning approach (Sec. 4.3.2) with a Gradient Boosting algorithm and using the closest performance input selection technique (Sec. 4.2.3). Our experiments show that the performance predictions of user persona $\mathcal{B}$, who is counting on inputs in an offline setting, will outperform the online predictions of user persona $\mathcal{C}$ whatever be the budget of configurations for reasonable budgets of configurations;
- If the user has **no online budget** (*e.g.*, the user persona $\mathcal{A}$), then the available offline budget is key. If the offline budget is low, there is no silver bullet: since we cannot guarantee low errors with our models, it is probably better to avoid predicting than providing a poor estimation of software performance. If the user has access to a diverse set of configurations' measurements over different inputs (high offline budget), then we can advise using the supervised offline approach (Sec. 4.3.3) implementing a Gradient Boosting algorithm with a random selection of inputs (see Sec. 4.2.2).

Figure 8 summarizes these rules of thumb into a flow diagram. We also depict likely locations for user personas $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ at the end of the decision process, based on our previous recommendations, as well as the observed relative errors from our experiments. These observed errors serve as a rule-of-thumb, but are, of course, not directly transferable to other systems and setups.

As a limitation of our work, we highlight that it is difficult to learn the performance distribution for a few software systems *e.g.*, *lingeling*, *poppler*, and even *Node.js* For these systems, the prediction errors are above 20% when implementing a supervised offline approach with tight budgets of configurations or inputs. It is worth noticing that a computational effort in terms of measurements is requested *i.e.*, to measure more than the general and averaged threshold of 25 inputs on average. The requested amount of inputs on a per-system basis

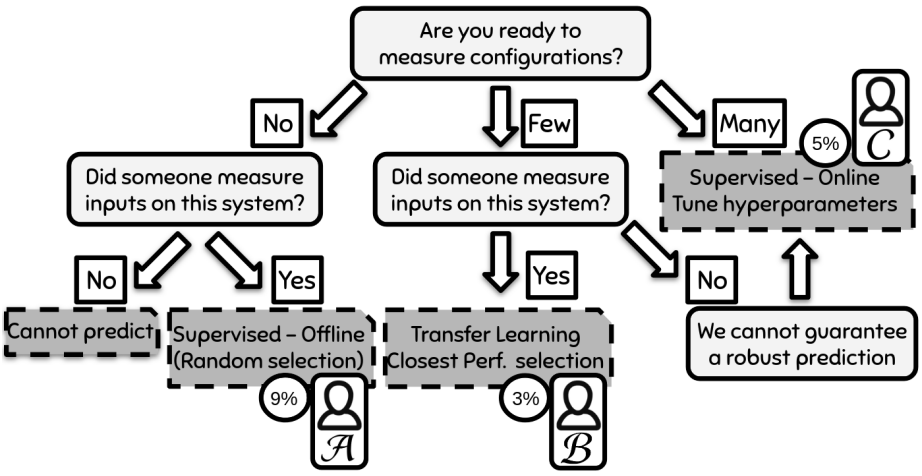

Figure 8: Lessons Learned - A flow diagram for users

is documented in the companion repository[7]. This is potentially related to the impact of individual features on the performance metric, i.e. the spread of the correlation, which was investigated in concurrent work for the same dataset as used in this paper [34]. For the difficult-to-learn software systems, there are multiple features that have a high impact on the performance metric [34, Table 4], leading to a more difficult-to-learn objective landscape for the regression task.

Our results relate further to the existing body of work and confirm some of the results previously found. For example, BEETLE [32] highlights the importance of selecting the right input-specific source(s) for transfer learning to maximize the accuracy and mitigate the risk of negative transfer, a problem similar to the input selection we consider in RQ_2. In [26], it is discussed in which scenarios transfer learning is more applicable compared to when it might lead to a negative transfer. The applicability is found to be hindered by a higher severity of the change between the original training environment and the target prediction environment to which the model is transferred. This problem of negative transfer in variability modelling was similarly confirmed for the Linux kernel in [37]. The negative transfer problem underlines the importance of establishing a training set that is as broad and diverse as possible, in order to best fit the data distribution that the model will apply during deployment. The closer the target environment, albeit new inputs, configurations, or other variability aspects, is to the environments from which the training data was collected, the smaller the risk of a negative transfer and the better the performance prediction quality.

[7] See https://github.com/simula-vias/input-aware-performance-models/blob/main/src/when_to_stop_measuring_inputs.ipynb

6. Threats to Validity

A first threat to validity is linked to the data we are using; since we rely on a dataset [34], we are exposed to the same threats to validity. In particular, an error in the measurement protocol could invalidate our results. Besides, we do not consider all the possible configuration options of the software systems. The fact that it includes multiple software systems (8 in total) and a consequent number of performance measurements (roughly two million when considering the different performance properties) is supposed to alleviate these threats. A second threat to validity relates to the input properties computed in Section 3.1.2. Since we are not domain experts of each of the eight software systems considered in this experiment, we cannot validate the construction of such properties, *i.e.*, it is likely that there is an opportunity to craft more expressive input properties. To the best of our knowledge, which input properties to use in order to improve the performance prediction remains an open question [11, 75]. Furthermore, we neglect their computational cost. As we mentioned, being able to report precisely the properties can be a tedious problem. Measures need to be precise, external factors need to be mitigated as much as possible to reduce potential interactions with the measurements, *etc.* Being meticulous about these aspects may drastically increase the cost to get such measurements and ultimately threaten the results of Section 4.2.2 or overestimate the benefit of the supervised offline approach. Another threat to validity is related to the randomness in machine learning methods, subject to modifications in their predictions. To reduce these stochastic effects, we (1) fix the random seed to feed the same training and test sets to all models and have comparable results and (2) repeat the experiments 20 times. Finally, we acknowledge that we relied on already existing libraries and implementations. These can be buggy or present some inaccuracies that may favour our results. We choose to use ML implementations coming from scikit-learn which is one of the most popular Python ML libraries at the moment. Its community is active and sensitive to these aspects, we can assume that if such a problem would exist, it would have been discovered and fixed quickly.

7. Related Work

Machine learning and configurable systems. Machine learning techniques have been widely considered in the literature to learn software configuration spaces [52, 54, 43, 25, 26, 68, 44, 45, 48, 19, 12, 16, 23]. Several works have proposed to predict the performance of configurations, with several use-cases in mind for developers and users of configurable systems [60, 12, 63, 62]: the maintenance and interpretability of configuration spaces, the exploration of tradeoffs in the configuration space, the automated specialization of configurable systems, or simply taking informed decisions when choosing a suited configuration. The selection of an optimal configuration [48, 19, 47] is also an extensive line of research. We do not target the problem of finding an optimal configuration in this article. Though prediction model can be leveraged, more targeted and

effective techniques have been proposed to find an optimal configuration [47]. Most of the studies support learning models restrictive to specific static settings, such that a new prediction model has to be learned from scratch once the environment changes. The variability of input data exacerbates the problem and questions the generalization of configuration knowledge, *e.g.*, a configuration is only optimal for a given input.

Input sensitivity of configurable systems. Input sensitivity has been partly considered in some specific research works. Let us take the video encoding [38] as an example: Pereira *et al.* [3] study the effect of sampling training data from the configuration space on *x264* configuration performance models for 19 input videos on two performance properties. Netflix conducts a large-scale study for comparing the compression performance of *x264*, *x265*, and *libvpx* [1]. 5000 12-second clips from the Netflix catalogue were used, covering a wide range of genres and signal characteristics. However, only two configurations were considered and the focus of the study was not on predicting performances. Our study covers much more inputs, systems, and performance properties. Valov *et al.* [67] proposed a method to transfer the Pareto frontiers (encoding time and size) of performances across heterogeneous hardware environments. Yet, the inputs (video) remain fixed, which is an immediate threat to validity. In fact, this threat is shared by numerous studies on configurable systems that consider configurations with the same input video (see [52] for the references). In response, we carefully assess numerous combinations of learning approaches, algorithms, and input selections to deal with input sensitivity. Input sensitivity is both the root cause hidden behind the need of input-aware performance models and the reason why these models may fail at predicting software performance whatever their input is. If there were no interaction at all between inputs and configurations, a simple performance model for all inputs would suffice. Based on this work combined with [34], our conjecture is as follows: the more input-sensitive a software system is, the more difficult (and costly) it is to train an efficient input-aware performance model.

The input sensitivity issue has also been identified — and sometimes dealt with — in some other domains: SAT solvers [75, 17], compilation [53, 11], data compression [29], database management [69, 14], cloud computing [15, 36, 12], *etc.* These works purposely leverage the specifics of their domain. However, it is unclear how proposed techniques could be adapted to any domain and all software systems [42]. Thus, We favour a generic, domain-agnostic approach (*e.g.*, transfer learning) as part of our study. Importantly, most of these works pursue the objective of optimizing the performance of a software system according to a given input (workload). In contrast, we consider the problem of predicting the performance of any configuration. Our key goal is to investigate how configuration knowledge can be generalized or transferred among inputs.

Transfer learning. Transfer learning has been considered for configurable software systems, with the idea of transferring knowledge across different computing environments *etc.* The promise is to reduce measurements' efforts and costs over configurations. Jamshidi *et al.* define Learning to Sample (L2S) [26] that combines an exploitation of the source and an exploration of the target to

sample a list of configurations. As many other transfer learning works [4], L2S is applied to transfer performance of executing environments (*e.g.*, hardware changes), not input changes. L2S could be adapted as part of transfer learning (see Figure 2). However, L2S is highly sensitive to the selection of a source (an input) for a given target (another input). Martin *et al.* develop TEAMs [37], a transfer learning approach predicting the performance distribution of the Linux kernel using the measurements of its previous releases. Valov *et al.* showed that linear models are effective to transfer knowledge across different hardware environments [68]. However, inputs can significantly alter performance distributions *e.g.*, Pearson correlations can be close to 0 for some pairs of inputs, systems, and performance properties. There is not necessarily a linear correlation and relationship, as for hardware changes. We assess model shifting as part of transfer learning. There are many studies in the literature of software engineering applying transfer learning for defect prediction [7, 35, 46, 9, 70]. They are used for handling a classification problem instead of a regression problem as in our case. Additionally, while researchers commonly utilize software quality metrics as predictive features for cross-software defects, our approach differs as we leverage configuration options to forecast performance. Beyond software systems, transfer learning is subject to intensive research in many domains (*e.g.*, image processing, natural language processing) [50, 73, 78]. Different kinds of data, assumptions, and tasks have been considered. The interplay between configuration options and inputs calls to tackle a regression problem over tabular data that differ from images or textual content. Some techniques are simply not applicable in our context. Another specificity of our problem is that there is this open question on how to select and adapt the source for a given target (here: a new input fed to a configurable system). Overall, we design transfer learning techniques that leverage characteristics of inputs and that can operate over tabular data.

In this paper, transfer learning techniques targeting the interplay between inputs and configurations work best if the source and the target inputs are close to each other (in terms of performance profiles [11]). To this matter and for this specific context, the most important part is neither the used ML algorithm nor the way to transfer the knowledge, but just to associate the right source input to the target input under prediction. Two insights regarding this finding; 1. to be optimal, transfer learning might require an additional offline effort (*i.e.*, measuring potential source inputs) to work best, even if the technique is supposed to be labelled as online, so we can pick the best source input among a sufficient set of inputs; 2. More than comparing TL techniques with each other, future efforts should be focusing the optimal association of inputs, how to find the best source input given the current target input. Our current proposition, deriving input properties as metrics between inputs to select the best source, can be seen as an extension of the bellwether effect (*e.g.*, used by BEETLE [32]), stating there exists a unique source input leading to superior transfer results whatever the target.

Selection problem. The automated algorithm selection problem is subject to intensive research [28, 22, 75, 65]: given a computational problem, a set

of algorithms, and a specific problem instance to be solved, the problem is to determine which of the algorithms can be selected to perform best on that instance. Techniques have substantially improved the State-of-the-Art in solving many prominent artificial intelligent problems, such as SAT, CSP, QBF, ASP, or scheduling problems [28]. For instance, SATZilla uses machine learning to select the most effective algorithm from a portfolio of SAT solvers for a given SAT formula [75]. There are several differences in our work. First, we target the problem of predicting the performance of any configuration as opposed to finding an optimal system. Second, in our case, the set comprises all (valid) configurations of a single, parameterized, configurable system. In the automated algorithm setting, the set of algorithms come from different individual software implementation and systems. As stated in [28] (Section 6), our problem differs and is still open because (1) the space of valid configurations to select from is typically very large; (2) learning the mapping from instance features (*i.e.*, inputs' properties) to configurations is challenging. We precisely address this problem in this article, considering a large dataset and multiple learning approaches.

8. Conclusion

Due to the interactions between inputs and configurations, predicting the performance property of a software configurable system whatever the input data is non-trivial and yet of practical importance. In particular, performance models trained on a single input can quickly become inaccurate and useless when used over other inputs. This lack of generalizability and practicality suggests to investigate solutions for learning input-aware performance models. In this article, we empirically evaluated the effectiveness of different learning strategies (offline learning, supervised online learning, transfer learning) and user personas when addressing this problem. We leveraged a large dataset comprising 8 software systems, hundreds of configurations and inputs, and dozens of performance properties, spanning a total of 1,941,075 configurations' measurements.

Our study empirically proves that measuring the performance of configurations on multiple inputs leads to 1) learning the complexity of predictive performance models ; 2) training models which are robust to the change of input data. Offline learning can build configuration knowledge that pays off and benefits to online learning when a new input needs to be processed. We emphasize the need to compute relevant input properties (*e.g.*, video characteristics) as part of the learning to discriminate the different inputs fed to the software system. As future work, we plan to consider input-aware optimization methods. The problem would differ: instead of transferring the whole performance distribution across inputs and configurations, optimization pursues the goal of finding a single optimal point, typically through the transfer of some configuration knowledge across inputs.

Acknowledgements.

This research was funded by the ANR-17-CE25-0010-01 VaryVary project and the associated Inria/Simula team Resilient Software Science (RESIST_EA) https://gemoc.org/resist/

References

[1] Jan De Cock, Aditya Mavlankar, Anush Moorthy, and Anne Aaron: A Large-Scale Comparison of x264, x265, and libvpx – a Sneak Peek. netflix-study (2016)

[2] Alourani, A., Bikas, M., Grechanik, M.: Input-sensitive profiling: A survey. In: Advances in Computers, pp. 31–52. Elsevier (2016)

[3] Alves Pereira, J., Acher, M., Martin, H., Jézéquel, J.M.: Sampling effect on performance prediction of configurable systems: A case study. In: Proc. of ICPE'20, p. 277–288 (2020)

[4] Ballesteros, J., Fuentes, L.: Transfer Learning for Multiobjective Optimization Algorithms Supporting Dynamic Software Product Lines, p. 51–59. Association for Computing Machinery, New York, NY, USA (2021). URL `https://doi.org/10.1145/3461002.3473944`

[5] Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. Journal of machine learning research **13**(2) (2012)

[6] Campello, R.J.G.B., Moulavi, D., Sander, J.: Density-Based Clustering Based on Hierarchical Density Estimates. In: J. Pei, V.S. Tseng, L. Cao, H. Motoda, G. Xu (eds.) Advances in Knowledge Discovery and Data Mining, Lecture Notes in Computer Science, pp. 160–172. Springer, Berlin, Heidelberg (2013). DOI 10.1007/978-3-642-37456-2_14

[7] Chen, H., Jing, X.Y., Li, Z., Wu, D., Peng, Y., Huang, Z.: An empirical study on heterogeneous defect prediction approaches. IEEE Transactions on Software Engineering (2020)

[8] Chen, J., Xu, N., Chen, P., Zhang, H.: Efficient compiler autotuning via bayesian optimization. In: Proc. of ICSE'21, pp. 1198–1209 (2021). DOI 10.1109/ICSE43902.2021.00110

[9] Chen, J., Yang, Y., Hu, K., Xuan, Q., Liu, Y., Yang, C.: Multiview transfer learning for software defect prediction. IEEE Access **7**, 8901–8916 (2019)

[10] Chen, T., Guestrin, C.: XGBoost: A scalable tree boosting system. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16, pp. 785–794. ACM, New York, NY, USA (2016). DOI 10.1145/2939672.2939785. URL `http://doi.acm.org/10.1145/2939672.2939785`

[11] Ding, Y., Ansel, J., Veeramachaneni, K., Shen, X., O'Reilly, U.M., Amarasinghe, S.: Autotuning algorithmic choice for input sensitivity. In: ACM SIGPLAN Notices, vol. 50, pp. 379–390. ACM (2015)

[12] Ding, Y., Pervaiz, A., Carbin, M., Hoffmann, H.: Generalizable and interpretable learning for configuration extrapolation. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021, p. 728–740. Association for Computing Machinery, New York, NY, USA (2021). DOI 10.1145/3468264.3468603. URL https://doi.org/10.1145/3468264.3468603

[13] Dowdy, S.M., Wearden, S.: Statistics for research. Wiley (1983)

[14] Duan, S., Thummala, V., Babu, S.: Tuning database configuration parameters with ituned. Proc. VLDB Endow. **2**(1), 1246–1257 (2009). DOI 10.14778/1687627.1687767. URL https://doi.org/10.14778/1687627.1687767

[15] Duarte, F., Gil, R., Romano, P., Lopes, A., Rodrigues, L.: Learning non-deterministic impact models for adaptation. In: Proc. of SEAMS'18, p. 196–205 (2018). DOI 10.1145/3194133.3194138. URL https://doi.org/10.1145/3194133.3194138

[16] Eggensperger, K., Lindauer, M., Hutter, F.: Neural networks for predicting algorithm runtime distributions. In: Proceedings of the 27th International Joint Conference on Artificial Intelligence, IJCAI'18, pp. 1442–1448. AAAI Press, Stockholm, Sweden (2018)

[17] Falkner, S., Lindauer, M., Hutter, F.: Spysmac: Automated configuration and performance analysis of sat solvers. In: Proc. of SAT'15, pp. 215–222 (2015)

[18] Friedman, J.H.: Stochastic gradient boosting. Computational Statistics & Data Analysis **38**(4), 367–378 (2002). DOI https://doi.org/10.1016/S0167-9473(01)00065-2. URL https://www.sciencedirect.com/science/article/pii/S0167947301000652. Nonlinear Methods and Data Mining

[19] Fu, W., Menzies, T.: Easy over hard: A case study on deep learning. In: Proc. of ESEC-FSE'17, p. 49–60 (2017). DOI 10.1145/3106237.3106256. URL https://doi.org/10.1145/3106237.3106256

[20] Gorishniy, Y., Rubachev, I., Khrulkov, V., Babenko, A.: Revisiting deep learning models for tabular data. In: A. Beygelzimer, Y. Dauphin, P. Liang, J.W. Vaughan (eds.) Advances in Neural Information Processing Systems (2021). URL https://openreview.net/forum?id=i_Q1yrOegLY

[21] Guo, J., Czarnecki, K., Apely, S., Siegmundy, N., Wasowski, A.: Variability-aware performance prediction: A statistical learning approach. In: Proc. of ASE'13, p. 301–311 (2013). DOI 10.1109/ASE.2013.6693089. URL https://doi.org/10.1109/ASE.2013.6693089

[22] Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Proc. of LION'05, p. 507–523 (2011). DOI 10.1007/978-3-642-25566-3_40. URL https://doi.org/10.1007/978-3-642-25566-3_40

[23] Hutter, F., Xu, L., Hoos, H.H., Leyton-Brown, K.: Algorithm runtime prediction: Methods & evaluation. Artificial Intelligence **206**, 79–111 (2014). DOI 10.1016/j.artint.2013.10.003

[24] Iorio, F., Hashemi, A.B., Tao, M., Amza, C.: Transfer learning for cross-model regression in performance modeling for the cloud. In: Proc. of CloudCom'19, pp. 9–18 (2019). DOI 10.1109/CloudCom.2019.00015

[25] Jamshidi, P., Siegmund, N., Velez, M., Kästner, C., Patel, A., Agarwal, Y.: Transfer learning for performance modeling of configurable systems: An exploratory analysis. In: Proc. of ASE'17, p. 497–508 (2017)

[26] Jamshidi, P., Velez, M., Kästner, C., Siegmund, N.: Learning to sample: Exploiting similarities across environments to learn performance models for configurable systems. In: Proc. of ESEC/FSE'18, p. 71–82 (2018). DOI 10.1145/3236024.3236074. URL https://doi.org/10.1145/3236024.3236074

[27] Kendall, M.G.: Rank correlation methods. Griffin (1948)

[28] Kerschke, P., Hoos, H.H., Neumann, F., Trautmann, H.: Automated algorithm selection: Survey and perspectives. Evolutionary Computation **27**(1), 3–45 (2019). DOI 10.1162/evco_a_00242. URL https://doi.org/10.1162/evco_a_00242

[29] Khavari Tavana, M., Sun, Y., Bohm Agostini, N., Kaeli, D.: Exploiting adaptive data compression to improve performance and energy-efficiency of compute workloads in multi-gpu systems. In: Proc. of IPDPS'19, pp. 664–674 (2019). DOI 10.1109/IPDPS.2019.00075

[30] Knüppel, A., Thüm, T., Pardylla, C.I., Schaefer, I.: Understanding parameters of deductive verification: An empirical investigation of key. In: J. Avigad, A. Mahboubi (eds.) Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings, *Lecture Notes in Computer Science*, vol. 10895, pp. 342–361. Springer (2018). DOI 10.1007/978-3-319-94821-8_20. URL https://doi.org/10.1007/978-3-319-94821-8_20

[31] Krause, A., Golovin, D.: Submodular function maximization. Tractability **3**, 71–104 (2014)

[32] Krishna, R., Nair, V., Jamshidi, P., Menzies, T.: Whence to learn? transferring knowledge in configurable systems using beetle. IEEE Transactions

on Software Engineering **47**(12), 2956–2972 (2021). DOI 10.1109/TSE.2020.2983927

[33] Larsson, H., Taghia, J., Moradi, F., Johnsson, A.: Source selection in transfer learning for improved service performance predictions. In: Proc. of Networking'21, pp. 1–9 (2021). DOI 10.23919/IFIPNetworking52078.2021.9472818

[34] Lesoil, L., Acher, M., Blouin, A., Jézéquel, J.M.: Input Sensitivity on the Performance of Configurable Systems: An Empirical Study. Journal of Systems and Software (2023). URL `https://hal.inria.fr/hal-03476464`

[35] Li, Z., Jing, X.Y., Wu, F., Zhu, X., Xu, B., Ying, S.: Cost-sensitive transfer kernel canonical correlation analysis for heterogeneous defect prediction. ASE **25**(2), 201–245 (2018)

[36] Liu, F., Miniskar, N.R., Chakraborty, D., Vetter, J.S.: Deffe: A data-efficient framework for performance characterization in domain-specific computing. In: Proc. of CF'20, p. 182–191 (2020). DOI 10.1145/3387902.3392633. URL `https://doi.org/10.1145/3387902.3392633`

[37] Martin, H., Acher, M., Lesoil, L., Jezequel, J.M., Khelladi, D.E., Pereira, J.A.: Transfer learning across variants and versions : The case of linux kernel size. IEEE Transactions on Software Engineering **1**, 1–1 (2021). DOI 10.1109/TSE.2021.3116768

[38] Maxiaguine, A., Yanhong Liu, Chakraborty, S., Wei Tsang Ooi: Identifying "representative" workloads in designing mpsoc platforms for media processing. In: 2nd Workshop onEmbedded Systems for Real-Time Multimedia, 2004. ESTImedia 2004., pp. 41–46 (2004). URL `https://ieeexplore.ieee.org/document/1359702`

[39] McInnes, L., Healy, J., Astels, S.: hdbscan: Hierarchical density based clustering. The Journal of Open Source Software **2**(11) (2017). DOI 10.21105/joss.00205. URL `https://doi.org/10.21105%2Fjoss.00205`

[40] Molnar, C.: Interpretable Machine Learning. Lulu.com, München, Bayern, Deutschland (2020)

[41] Mühlbauer, C.S., Sattler, F., Siegmund, N.: Analyzing the impact of workloads on modeling the performance of configurable software systems. In: Proceedings of the International Conference on Software Engineering (ICSE), IEEE (2023)

[42] Mühlbauer, S., Sattler, F., Kaltenecker, C., Dorn, J., Apel, S., Siegmund, N.: Analyzing the impact of workloads on modeling the performance of configurable software systems. In: Proceedings of the 45th International Conference on Software Engineering. IEEE (2023)

[43] Nair, V., Krishna, R., Menzies, T., Jamshidi, P.: Transfer learning with bellwethers to find good configurations. CoRR **abs/1803.03900** (2018). URL http://arxiv.org/abs/1803.03900

[44] Nair, V., Menzies, T., Siegmund, N., Apel, S.: Using bad learners to find good configurations. In: Proc. of ESEC/FSE'17, pp. 257–267 (2017)

[45] Nair, V., Yu, Z., Menzies, T., Siegmund, N., Apel, S.: Finding faster configurations using flash. IEEE Transactions on Software Engineering (2018)

[46] Nam, J., Fu, W., Kim, S., Menzies, T., Tan, L.: Heterogeneous defect prediction. IEEE Transactions on Software Engineering **44**(9), 874–896 (2017)

[47] OH, J., Batory, D., HERADIO, R.: Finding near-optimal configurations in colossal spaces with statistical guarantees. TOSEM (2023)

[48] Oh, J., Batory, D., Myers, M., Siegmund, N.: Finding near-optimal configurations in product lines by random sampling. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, p. 61–71. Association for Computing Machinery, New York, NY, USA (2017). DOI 10.1145/3106237.3106273. URL https://doi.org/10.1145/3106237.3106273

[49] Oshiro, T.M., Perez, P.S., Baranauskas, J.A.: How many trees in a random forest? In: P. Perner (ed.) Machine Learning and Data Mining in Pattern Recognition, pp. 154–168. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

[50] Pan, S.J., Yang, Q.: A survey on transfer learning. TKDE **22**(10), 1345–1359 (2009)

[51] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al.: Scikit-learn: Machine learning in python. the Journal of machine Learning research **12**, 2825–2830 (2011)

[52] Pereira, J.A., Acher, M., Martin, H., Jézéquel, J.M., Botterweck, G., Ventresque, A.: Learning software configuration spaces: A systematic literature review. JSS p. 111044 (2021). DOI https://doi.org/10.1016/j.jss.2021.111044. URL https://www.sciencedirect.com/science/article/pii/S0164121221001412

[53] Plotnikov, D., Melnik, D., Vardanyan, M., Buchatskiy, R., Zhuykov, R., Lee, J.H.: Automatic tuning of compiler optimizations and analysis of their impact. Procedia Computer Science **18**, 1312–1321 (2013). DOI 10.1016/j.procs.2013.05.298. URL https://doi.org/10.1016/j.procs.2013.05.298

[54] Quinton, C., Vierhauser, M., Rabiser, R., Baresi, L., Grünbacher, P., Schuhmayer, C.: Evolution in dynamic software product lines. Journal of Software: Evolution and Process p. e2293 (2020)

[55] Safavian, S.R., Landgrebe, D.: A survey of decision tree classifier methodology. IEEE transactions on systems, man, and cybernetics **21**(3), 660–674 (1991)

[56] Schreiber, J., Bilmes, J., Noble, W.S.: Apricot: Submodular selection for data summarization in Python. Journal of Machine Learning Research **21**(161), 1–6 (2020)

[57] Seber, G.A., Lee, A.J.: Linear regression analysis, vol. 329. John Wiley & Sons, North America (2012)

[58] Shwartz-Ziv, R., Armon, A.: Tabular data: Deep learning is not all you need. Information Fusion **81**, 84–90 (2022). DOI 10.1016/j.inffus.2021.11.011

[59] Siegel, S.: Nonparametric statistics for the behavioral sciences. The Journal of Nervous and Mental Disease **125**, 497 (1956)

[60] Siegmund, N., Grebhahn, A., Kästner, C., Apel, S.: Performance-influence models for highly configurable systems. In: ESEC/FSE'15 (2015). URL `https://doi.org/10.1145/2786805.2786845`

[61] Sinha, U., Cashman, M., Cohen, M.B.: Using a genetic algorithm to optimize configurations in a data-driven application. In: A. Aleti, A. Panichella (eds.) Search-Based Software Engineering - 12th International Symposium, SSBSE 2020, Bari, Italy, October 7-8, 2020, Proceedings, *Lecture Notes in Computer Science*, vol. 12420, pp. 137–152. Springer (2020). DOI 10.1007/978-3-030-59762-7_10. URL `https://doi.org/10.1007/978-3-030-59762-7_10`

[62] Temple, P., Acher, M., Jezequel, J.M., Barais, O.: Learning contextual-variability models. IEEE Software **34**(6), 64–70 (2017)

[63] Temple, P., Acher, M., Jézéquel, J.M., Noel-Baron, L., Galindo, J.A.: Learning-Based Performance Specialization of Configurable Systems. Research report, IRISA, Inria Rennes ; University of Rennes 1 (2017). URL `https://hal.archives-ouvertes.fr/hal-01467299`

[64] Thirey, B., Hickman, R.: Distribution of euclidean distances between randomly distributed gaussian points in n-space. arXiv preprint arXiv:1508.02238 (2015)

[65] Thornton, C., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In: Proc. of KDD'13, p. 847–855 (2013)

[66] Valov, P., Guo, J., Czarnecki, K.: Empirical comparison of regression methods for variability-aware performance prediction. In: Proceedings of the 19th International Conference on Software Product Line, SPLC '15, p. 186–190. Association for Computing Machinery, New York, NY, USA (2015). DOI 10.1145/2791060.2791069. URL https://doi.org/10.1145/2791060.2791069

[67] Valov, P., Guo, J., Czarnecki, K.: Transferring pareto frontiers across heterogeneous hardware environments. In: Proceedings of the ACM/SPEC International Conference on Performance Engineering, ICPE '20, p. 12–23. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3358960.3379127. URL https://doi.org/10.1145/3358960.3379127

[68] Valov, P., Petkovich, J.C., 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 '17, p. 39–50. Association for Computing Machinery, New York, NY, USA (2017). DOI 10.1145/3030207.3030216. URL https://doi.org/10.1145/3030207.3030216

[69] Van Aken, D., Pavlo, A., Gordon, G.J., Zhang, B.: Automatic database management system tuning through large-scale machine learning. In: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD '17, p. 1009–1024. Association for Computing Machinery, New York, NY, USA (2017). DOI 10.1145/3035918.3064029. URL https://doi.org/10.1145/3035918.3064029

[70] Wang, A., Zhang, Y., Wu, H., Jiang, K., Wang, M.: Few-shot learning based balanced distribution adaptation for heterogeneous defect prediction. IEEE Access **8**, 32989–33001 (2020)

[71] Wang, Y., Inguva, S., Adsumilli, B.: Youtube ugc dataset for video compression research. 2019 IEEE 21st International Workshop on Multimedia Signal Processing (MMSP) (2019). DOI 10.1109/mmsp.2019.8901772. URL http://dx.doi.org/10.1109/MMSP.2019.8901772

[72] Wei, H., Zhou, S., Yang, T., Zhang, R., Wang, Q.: Elastic resource management for heterogeneous applications on paas. In: Proceedings of the 5th Asia-Pacific Symposium on Internetware, Internetware '13. Association for Computing Machinery, New York, NY, USA (2013). DOI 10.1145/2532443.2532451. URL https://doi.org/10.1145/2532443.2532451

[73] Weiss, K., Khoshgoftaar, T.M., Wang, D.: A survey of transfer learning. Journal of Big data **3**(1), 9 (2016)

[74] Xu, B., Zhao, D., Jia, K., Zhou, J., Tian, J., Xiang, J.: Cross-project aging-related bug prediction based on joint distribution adaptation and improved

subclass discriminant analysis. In: 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE), pp. 325–334 (2020). DOI 10.1109/ISSRE5003.2020.00038

[75] Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Satzilla: portfolio-based algorithm selection for sat. Journal of artificial intelligence research **32**, 565–606 (2008)

[76] Xu, T., Jin, L., Fan, X., Zhou, Y., Pasupathy, S., Talwadker, R.: Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 307–319 (2015). DOI 10.1145/2786805.2786852

[77] Zhang, J., Liu, Y., Zhou, K., Li, G., Xiao, Z., Cheng, B., Xing, J., Wang, Y., Cheng, T., Liu, L., Ran, M., Li, Z.: An end-to-end automatic cloud database tuning system using deep reinforcement learning. In: Proceedings of the 2019 International Conference on Management of Data, SIGMOD '19, p. 415–432. Association for Computing Machinery, New York, NY, USA (2019). DOI 10.1145/3299869.3300085. URL `https://doi.org/10.1145/3299869.3300085`

[78] Zhang, W., Deng, L., Zhang, L., Wu, D.: Overcoming negative transfer: A survey. arXiv preprint arXiv:2009.00909 (2020)