A comparison of performance specialization learning for configurable systems

Hugo Martin, Mathieu Acher, Juliana Alves Pereira, Jean-Marc Jézéquel

A Comparison of Performance Specialization Learning for Configurable Systems

Hugo Martin
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
hugo.martin@irisa.fr

Mathieu Acher
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
mathieu.acher@irisa.fr

Juliana Alves Pereira
Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro, Brazil
jpereira@inf.puc-rio.br

Jean-Marc Jézéquel
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
jean-marc.jezequel@irisa.fr

ABSTRACT

The specialization of the configuration space of a software system has been considered for targeting specific configuration profiles, usages, deployment scenarios, or hardware settings. The challenge is to find constraints among options' values that only retain configurations meeting a performance objective. Since the exponential nature of configurable systems makes a manual specialization unpractical, several approaches have considered its automation using machine learning, *i.e.,* measuring a sample of configurations and then learning what options' values should be constrained. Even focusing on learning techniques based on decision trees for their built-in explainability, there is still a wide range of possible approaches that need to be evaluated, *i.e.,* how accurate is the specialization with regards to sampling size, performance thresholds, and kinds of configurable systems. In this paper, we compare six learning techniques: three variants of decision trees (including a novel algorithm) with and without the use of model-based feature selection. We first perform a study on 8 configurable systems considered in previous related works and show that the accuracy reaches more than 90% and that feature selection can improve the results in the majority of cases. We then perform a study on the Linux kernel and show that these techniques performs as well as on the other systems. Overall, our results show that there is no one-size-fits-all learning variant (though high accuracy can be achieved): we present guidelines and discuss tradeoffs.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; • **Computing methodologies → Classification and regression trees**.

1 INTRODUCTION

More and more software systems are configurable through command-line parameters, configuration files, or compile-time options. Users can set many configuration options' values to fit their functional requirements and performance objectives. For instance, users can change parameters' values of the x264 encoder to obtain a video in a fast way; users can configure Linux to obtain a kernel with a binary size below a certain threshold (*e.g.,* less than 20Mb).

The configuration of a configurable system is an error-prone and time-consuming task. There is a combinatorial explosion of possible configurations and the effects of options on performance is hard to document and formalize. Numerous works have shown that quantifying the performance influence of each individual option is not meaningful in most cases [3]. That is, the performance influence of n options, all jointly activated in a configuration, is not easily deducible from the performance influence of each individual option. Capturing the complex interactions among options and their effects on performance is mandatory.

An approach for supporting users in the configuration process is to *specialize* the configuration space of a software system. Given a performance objective, the specialization builds presets or profiles through constraints over options' values. Constraints can be on individual options: some specific values are already preset and users can focus on the remaining options. Constraints can also be among several options to only keep combinations of options' values (configurations) that have acceptable performance. For instance, the encoder x264 can be specialized to encode videos in a fast way: some options values are preset while the remaining options can still be configured for dealing with hardware constraints, output quality, or functional concerns.

Specializing the configuration space of a software system has a long tradition since the seminal paper of Czarnecki *et al.* [12]. Specialization has been considered for targeting specific profiles, usages, deployment scenarios, or hardware settings [2, 4, 11, 20, 41]. The idea is to retain only a subset of configurations that meet a performance threshold (*e.g.,* execution time below one second) and thus discard the rest. Specialization should not be confused with configuration optimization (tuning) where the goal is to find a unique and optimal configuration. Through specialization, users still have flexibility (variability) to configure their systems. The benefit is that some options' value are already preset for reaching a performance threshold. A unique challenge is to identify configurations that should be kept (or, equivalently, to discard non-acceptable configurations), which boils down to specify constraints among options' values.

On the one hand, measuring all configurations is infeasible in practice owing to the combinatorial explosion of possible configurations. On the other hand, the manual specification of constraints is an error-prone, time-consuming, and hardly repeatable task for any performance objective. It is however possible to automated the specialization process using machine learning, *i.e.,* measuring a sample of configurations and then learning what options' values

should be constrained. Even focusing on learning techniques based on decision trees for their built-in explainability, there is still a wide range of possible approaches that need to be evaluated, *i.e.,* how accurate is the specialization with regards to sampling size, performance thresholds, and kinds of configurable systems.

In this paper, we propose six learning techniques based on regression, classification, feature selection and a combination thereof. We first perform a study on 8 configurable systems with dozens of options considered in the related work. We then perform a study on a much larger configurable systems – the Linux kernel considering 9K+ options. In summary, our contributions are as follows:

- We propose a new way to use decision trees, called specialized regression and tailored toward performance specialization;
- We design and develop six performance specialization learning strategies with three variants of Decision Trees: regression, classification, and specialized regression;
- We perform an empirical study to compare the accuracy results of these techniques. Through our experimental results, we find that Decision Tree is a very accurate algorithm for performance specialization. However, they are sensitive to threshold variation. Feature selection is a reliable way to produce more accurate results; however training time was not significant.
- We perform a case study with the Linux kernel and we show very similar results to the 8 considered baseline configurable systems, except for training time which for Linux presented significant improvements with the use of feature selection.
- We gather stakeholders on how to use the different strategies for automated performance specialization. We discuss tradeoffs of training a model on the fly, and using default strategy and feature selection. We also point out future directions.
- We have made all the artifacts from our experiments publicly available at https://github.com/HugoJPMartin/SPLC2021.

2 PERFORMANCE SPECIALIZATION

In a configurable system, not all combinations of options' values are possible (*e.g.*, some options are mutually exclusive and some thresholds are required). Variability models are used to precisely define the space of valid (functional) configurations, typically through the specification of logical constraints among options. Assuming that all supposedly valid configurations of a variability model lead to acceptable products can be misleading since constraints may not be specified due to missing knowledge beforehand when the variability model is built. Moreover, nowadays systems are developed by integrating products, which belong to multiple systems, and communicate and interact with each other under various configurations [37]. In this scenario, configurations may fail to interact leading to unacceptable performances, *e.g.* dependencies on external libraries are not considered. The manual identification of constraints is a difficult task and it is easy to forget or wrongly specify a constraint leading to configurations that do not meet a particular requirement [42]. However, it is most of the time impractical to exhaustively test and measure system performance under all possible configurations. To overcome this issue, we can specialize a variability model to deliver the right functionality and performance assisting stakeholders in making informed decisions.

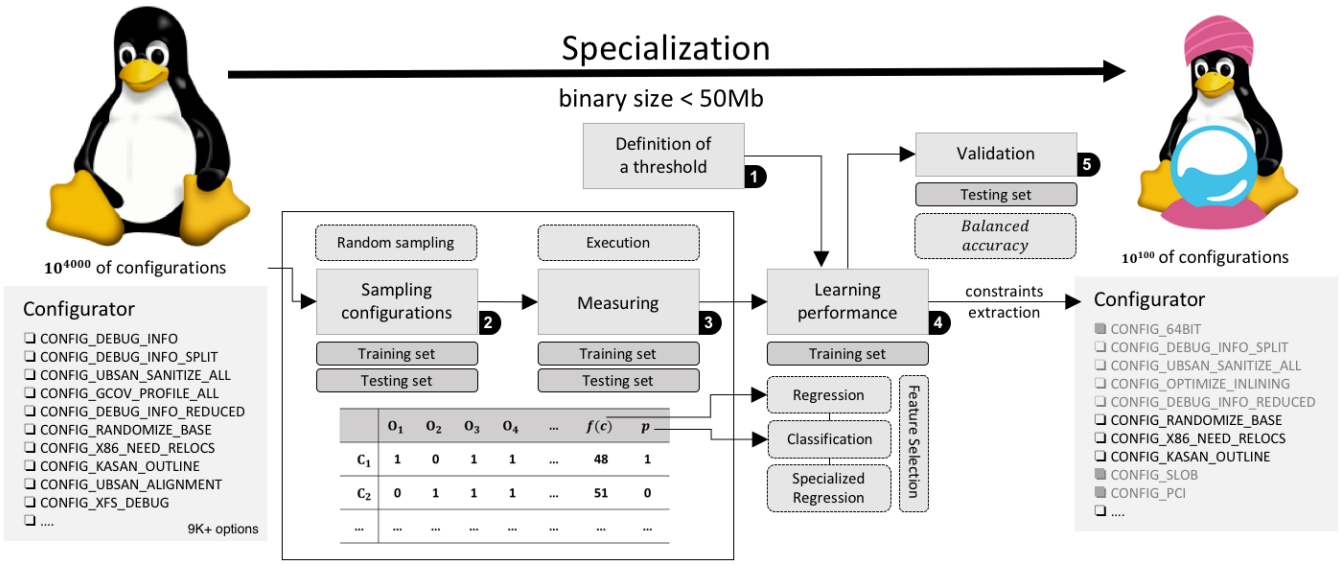
Specialization is the process of limiting the variability space to a subset of configurations that meet a specific threshold. Thresholds are logical decision rules over non-functional property values with regards to system limitations, such as `binary size` $<$ `50Mb`. They are constraints defined as equality (*i.e.*, $=$) or inequalities (*i.e.*, $\leq$; $<$; $>$; $\geq$) [30]. The specialization process is a transformation process that takes a variability space as input and yields another variability space as output, such that the set of configurations denoted by the latter space is a subset of the configurations denoted by the former space (see Figure 1) [11].

The specialization of a variability space involves the addition of a set of new constraints (rules) to options' values. Such rules describe how performance influences the system options' interactions, i.e. the variability space is restricted to only configurations satisfying the given performance threshold. The restricted subset of configurations represents the space of specialization. As an example, the Linux can be specialized to obtain a kernel binary size below `50Mb` (see Figure 1). To meet the threshold, a set of options values (in gray) are preset to true while others to false, and the remaining options (in black) can still be configured. This is helpful to avoid large kernels that take too much time to compile. Notice that we consider a threshold, thus contrary to optimization approaches that aim at satisfying a specific optimization objective (*e.g.* `minimize binary size`), we aim at specializing the configuration space to a subset of acceptable configurations. Although optimization is not our aim here, the specialization may highly assist the optimization process later when deriving a suited configuration, *e.g.* to support the variability at runtime.

How do we find a set of options that affect considerably performance? Options can be captured by running the system and measuring performance of each configuration. However, as the configuration space is typically very large and measuring each valid configuration is often infeasible, the use of learning-based techniques are promising for specialization. Learning techniques automatically obtains rules without exploring all possible configurations. They are used under the condition a sample of configurations' measurements is available. The idea is to learn out of a (small) sample of configurations' observations and hopefully generalize to the whole configuration space.

The learning process consists of four main stages: (1) definition of a performance threshold, (2) sampling, (3) measuring, and (4) learning (see Figure 1). The fifth stage (validation) is interesting for assessing the prediction models and their accuracy – a central research question of this paper. First, the process starts by defining a threshold and selecting a sample set of valid configurations. Second, building and measuring the sample of configurations. Third, these measurements and the performance threshold are used as input to accurately learn a prediction model. Finally, the validation step computes the accuracy of the prediction model.

Prediction models support stakeholders understanding the characteristics of the variability space, *i.e.* the effects of some options and how options interact. Interpretability for specialization is very important both for validating the insights of the learning and for



	O_1	O_2	O_3	O_4	...	$f(c)$	p
C_1	1	0	1	1	...	48	1
C_2	0	1	1	1	...	51	0
...	...	...	...	...	...	...	...

Figure 1: Configuration of a specialized Linux kernel. Given a performance threshold (binary size < 50Mb), the specialization process identifies constraints among options that preclude some (combinations of) values (in gray); users still have some flexibility to configure other options (in black).

encoding the knowledge into a variability model. As an example, consider learning techniques based on decision trees. Each branch of a tree represents a set of rules (decisions) that satisfy a threshold leaf. Thus, as a result, rules are mined by building the conjunction of a path to reach a threshold leaf. In the example of Figure 1, the algorithm learns that by having the options `CONFIG_64BIT`, `CONFIG_SLOB`, and `CONFIG_PCI` deselected; and `CONFIG_DEBUG_INFO_SPLIT`, `CONFIG_UBSAN_SANITIZE_ALL`, `CONFIG_OPTIMIZE_INLINING`, and `CONFIG_DEBUG_INFO_REDUCED` selected; dramatically increase performance (binary size). Thus, options are preset and no modification of these options' values are allowed. Without specialization, we are likely to set different options' values. An interesting example is "defconfig", a default configuration for Linux, in which the binary size is around `70Mb`. So, in case users start the configuration with "defconfig", they do not meet the specified threshold. Notice that learning is useful and necessary since manually discovering such rules is tedious, error-prone, and time-consuming. Moreover, it requires expert's knowledge of the system domain. The overall outcome is constraints among options that are retrofitted into a variability model and a configurator. Constraints can either force the values of individual options or logically involve several options. Such constraints are learned out of learning models.

Related Work. Performance specialization is applicable to several domains. Temple *et al.* [41, 42] explore the domain of video generator. To improve the learning process, Temple *et al.* [42] specifically target low confidence areas for sampling. The authors use an adversarial learning technique, called evasion attack, after a classifier is trained with a support vector machine. Beyond video generator, Temple *et al.* [40] explore also the domains of web server, compiler, database system, and image processing. Acher *et al.* [2] explore the domain of creating a specialized document (such as a paper or curriculum). Amand *et al.* [4] applied learning techniques in the 3D printing domain.

There are several works in the faulty specialization context [5, 14, 15, 22, 35, 37, 41, 47]. These works explore several domains, such as software-intensive systems, combinatorial model, and 3D printing. However, these works focus in mining rules for avoiding invalid configurations, without having performance in mind. As in the previous works they stick to machine learning classifiers.

There are also several works on performance prediction [3]. They use the same process of sampling, learning and validation. However they stick to predict the performance of configurations – not classifying configurations in terms of acceptable performance. To the best of our knowledge, such regression-based approaches have never been considered in the specialization context.

Sampling configurations is a first step for learning-based performance specialization. Several strategies have been proposed [6, 9, 18, 19, 21, 24, 27, 31, 34, 36, 45]. For instance, Oh *et al.* [31] explore t-wise coverage with uniform sampling, to ensure a good coverage of interactions between options. Munoz *et al.* [27] tackle the problem of uniform sampling through the handling of numerical options. Uniform random sampling is a strong baseline [18, 24, 34] but is challenging to compute for large feature models [19, 36].

Overall, studies have covered different aspects of specialization over the last years, such as the use of different systems, algorithms and sampling techniques. However, from an empirical perspective it is not evident to what extent interpretable learning approaches are effective for performance specialization. Current practices mainly focus on classification algorithms. Although in the optimization and performance prediction scenarios there are works using regression techniques [13, 16, 28, 29, 46], these techniques have not been yet

explored for specialization. Also, some additional steps are worth exploring (like feature selection prior to learning).

3 AUTOMATED PERFORMANCE SPECIALIZATION

The key to automated specialization is the identification of constraints that can preclude configurations not fitting a performance threshold. We first frame the specialization problem as a learning problem. We discuss the space of possible learning algorithms and show that decision trees represent a good fit between accuracy and the identification of constraints. We then present three techniques that can be combined with tree-based feature selection to realize performance specialization.

3.1 Specialization as a learning problem

We denote p the number of configuration options and define the configuration space $\mathbb{C} = \{0, 1\}^p$. Out of this space of configurations $\mathbb{C}$, we gather a subset of d configurations, denoted $\mathbb{C}_\mathbb{S} \subset \mathbb{C}^d$. We separate $\mathbb{C}_\mathbb{S}$ into a training set $\mathbb{C}_\mathbb{S}^{tr}$ and a test set $\mathbb{C}_\mathbb{S}^{te}$, so $\mathbb{C}_\mathbb{S} = \mathbb{C}_\mathbb{S}^{tr} \bigoplus \mathbb{C}_\mathbb{S}^{te}$. Let $\mathbb{B} = \{0, 1\}$ resp. for "non-acceptable" and "acceptable". Then, we denote:

- $f : \mathbb{C} \rightarrow \mathbb{R}^+$ the function affecting to any configuration $c \in \mathbb{C}$ its performance $f(c) \in \mathbb{R}^+$,
- $p : \mathbb{R} \rightarrow \mathbb{B}$ a predicate that determine whether a performance value is acceptable or non-acceptable,
- $s : \mathbb{C} \times p \rightarrow \mathbb{B}$ the specialization function affecting to any configuration $c \in \mathbb{C}$ its acceptability $p(f(c)) \in \mathbb{B}$.

With respect to these notations, the goal is to train a learning algorithm $\hat{s}$ estimating the function s for each measured configuration of the training set $c \in \mathbb{C}_\mathbb{S}^{tr}$. The training set $\mathbb{C}_\mathbb{S}^{tr}$ is used to obtain a learning model, while the testing set $\mathbb{C}_\mathbb{S}^{te}$ only tests the prediction accuracy of $\hat{s}$.

3.2 Learning algorithms for specialization

Numerous statistical learning algorithms can be used for performance specialization. These algorithms differ in terms of computational cost, expressiveness and interpretability. We now review the literature of configurable systems (based on the systematic survey [3]). We discuss what algorithms are suited or not for our specific problem. *Linear regressions* are considered as easy to interpret, but are unable to capture interactions between options or to handle non-linear effects [26]. *Neural networks* can reach high accuracy on large datasets. DeepPerf [17] has been developed for tackling configurable systems with dozens of options. Empirical results show the effectiveness of DeepPerf [17] on all systems of our study (except Linux that has not been considered). However, neural networks are black-box functions for which it is hard to extract rules (constraints) among options, a top requirement in the specialization problem.

Siegmund *et al.* [38] introduced a learning method called *performance-influence model*. Feature-forward selection and multiple linear regression are used in a stepwise manner to shrink or keep terms representing options or interactions. This method aims to handle interactions between options, limit the number of options to learn on, and provide a human-readable formula describing influence of options and combinations of options on performance. However, performance-influence model does not address the performance specialization problem. First, the model addressed a regression problem while we are interested in predicting the class of a configuration. Second, the technique does not retrofit constraints into a variability model. The synthesized information (a formula with coefficients) is not designed for extracting rules and constraints. Third, a performance-influence model aims to construct a global performance model: a rewrite is necessary to take the threshold into account. Fourth, the step-wise addition of interactions among options does not scale for Linux that exhibits 9K+ options (see more details hereafter).

*Decision trees (*e.g., *CART)* are the most used technique in the literature [3]. Decision trees have been used either for classification or regression, have reach competing accuracy, and are interpretable by construction [40, 43, 44]. The tree structure is ideal for capturing interactions between options. Rules can easily be extracted and retrofitted into a variability model.

Random forests are an ensemble learning method that constructs a multitude of decision trees at training time. As an ensemble method, the accuracy of random forests can be better than decision trees. However the question of extracting rules out of multiple trees is still an open issue.

Overall, decision trees represent a good fit for our specialization problem. We will consider this learning technique under different strategies in the following.

3.3 Learning strategies

Fundamentally, the learning problem presented in Section 3.1 is a supervised, binary *classification* problem. However, and which makes it remarkable, the learning has at its disposal continuous values (performance measurements). In this regard, the problem is closed to a *regression problem*, except that one does not want to eventually predict a quantity but a class.

We now describe three possible techniques for performance specialization learning: classification, regression, and specialized regression.

Classification. The straightest strategy is to use a classification tree. Decision tree models where the target variable can take a discrete set of values (acceptable, non-acceptable) are called classification trees. From the measured performance of configurations and the defined threshold, it is possible to label each configuration as acceptable or non-acceptable. Then a classification tree is trained to predict the performance acceptability on new configurations. One of the downside of this approach is that the actual performance value is lost from the dataset, as it is replaced by a simple boolean value. Not having the numerical information in case a configuration is either borderline from the performance threshold, or very far from that threshold, is likely to degrade the accuracy of the learning process.

Gap in the related work. Classification trees have been considered in prior works [2, 4, 40]. However, a systematic comparison with other techniques is missing and it is unclear how the approach works for large systems like Linux.

Regression. Another way to produce a decision tree is to tackle the regression problem. In this case, it would predict directly the performance of a configuration and then a post-process step can be applied to determine if that predicted performance is acceptable. It means that the regression tree has to be rewritten for predicting the outcome (a class). The promise is to better use the performance value, unlike the classification approach. However, the learning is focused on minimizing the error between the predicted performance value and the actual value independent of the threshold. As a result, a configuration could well have its performance value predicted with a very low error but just on the other side of the threshold, leading to a classification error unrecognized by the regression algorithm. It should be noted that a regression approach is agnostic to the threshold, meaning that the model can be used once and for all, for any threshold. In contrast, a classification tree should be trained anytime a new performance threshold is specified.

Gap in the related work. Although regression problems have been widely considered for performance prediction or optimization, we are unaware of existing works that investigate their effectiveness in the context of specialization. As stated, specialization is in-between a regression and classification problem, which questions the accuracy of such techniques in this context.

Specialized regression. Considering the weaknesses of the two previous approaches, we propose a novel and hybrid strategy. The training of a regression tree is performed over the dataset where all performance values higher than the performance threshold are increased by an important amount. Intuitively, we artificially create a "gap" in the performance distribution representing the threshold. The intent is to punish errors across the border, while still being able to take advantage of the insight given by the performance value. We call this techniques specialized regression since the regression is aware of the specialization *criterion* (performance threshold). In contrast, regression is only aware of the performance values of the training set and ignore the performance threshold.

3.4 Tree-based Feature selection

An hypothesis is that some configuration options of configurable systems have little effects on non-functional properties and can be removed without incurring much loss of information. Feature selection[1] techniques are worth considering when there are many features and comparatively few samples. Though we are not necessarily in extreme cases like the analysis of DNA microarray data [10, 39], the Linux case exhibits a very large number of options p. The goal is to operate over a reduced number of variables $p' \ll p$ when training. For smaller configurable systems, feature selection is also a promising approach as a way to shorten training times, simplify models, and focus on influential options.

Feature selection can be realized through *embedded methods* that keep and remove features as part of the model construction process [10]. Techniques like Lasso (for least absolute shrinkage and selection operator), performance-influence models [38] and decision trees enter in this category. In fact, most of the learning techniques use this method internally.

Another complementary approach is to use *model-based feature selection* prior to the actual learning. It is a two-step method. First, a learning process computes what we call a *feature ranking list* for ordering important features. Then, this list is fed to a learning algorithm (in our case: a decision tree) that operates over a subset of predictive features. Compared to traditional learning algorithms, the use of model-based feature selection enables to explicitly restrict the feature space and hopefully focus on the "right" features. Hence, the actual training (second step) can benefit from a more informative feature space. Decision trees are well-suited for the specialization problem w.r.t. constraints extraction, but may suffer from accuracy issues when the configuration space is large (*e.g.,* as is the case for Linux).

Knowing which configuration options are most predictive is crucial and corresponds to the first step of the method. We use random forest to learn the feature ranking list, owing to their predictive power and the ability to compute *feature importance*. Intuitively, feature importance is the increase in the prediction error of the model after we permuted the feature's values [26]. For random forest, we compute *feature permutation importance* through the observation of the effect on machine learning model accuracy of randomly shuffling each predictor variable [7, 26, 32].

Gap in the related work. Tree-based feature selection can be combined with classification, regression, and specialized regression. To the best of our knowledge, tree-based feature selection has not been considered for specializing configurable systems. In total, we have carefully selected and designed 6 specialization learning techniques for which we have no evidence of their (relative) effectiveness on real-world configurable systems – we aim to fill this gap in the rest of the paper.

4 STUDY DESIGN

To investigate the proposed approaches, we elaborate four research questions to conduct our experiment:

- **RQ1 : What is the accuracy and cost of learning strategies?** We aim to evaluate the prediction errors of specialization learning on nine real-world configurable systems, with varying cost in term of number of configurations' measurements.
- **RQ2 : What is the best learning strategy?** There are multiple ways to use decision trees (namely classification, regression and specialized regression) in order to synthesize the rules needed for the specialization. With regard to accuracy, is there a best strategy *e.g.,* whatever the subject system is?
- **RQ3 : What are the effects of tree-based feature selection on the accuracy of performance specialization?** We aim to investigate whether feature selection improves or degrades the accuracy of learning strategies and for which subject systems, performance thresholds, and training set size.
- **RQ4 : What is the time cost of the 6 learning strategies to predict performance of a configurable software system?** This question is to evaluate the practicality and feasibility of a proposed strategy. To answer this question, we show

[1] From a terminology point of view, feature selection is sometimes used in the software product line community to refer to the selection of features *i.e.,* to configuration. We use here as a machine learning terminology and refer to the selection of a subset of features considered as relevant for building models. We also consider that there is a mapping between configuration options and features.

System	Features	Measured Configurations
Apache	9	192
Berkeley C	19	2.560
Berkeley J	26	180
Dune	11	2.304
HMSGP	14	3.456
HIPAcc	33	13.485
LLVM	11	1.024
SQLite	39	4.553
Linux	9.467	92.562

Table 1: Datasets information, including number of features and number of measured configurations

the time consumed by the feature selection (if used), hyper-parameter searching and training process on various highly configurable software systems (including Linux). We also put in perspective the fact that some techniques are learned once and for all, whatever the performance thresholds.

4.1 Datasets

In order to perform our experimentation, we relied on datasets already used in the community (see details in Table 1).

4.1.1 Common SE datasets. Our first part of the experimentation is to perform train Decision Trees on reliable datasets commonly used on numerous occasions in the past [17, 38] to assert learning techniques efficiency.

4.1.2 Linux dataset. To investigate the scalability of the approaches, we rely on the dataset made available by Acher *et al.* [1] on Linux kernel size. It consists in the binary size measurement of 92.562 configurations, restrained to x86 architectures and 64 bits systems, and obtained with the use of randconfig, a widely used tool to generate random kernel configurations, and valid w.r.t constraints between options. It contains all boolean and tristate options, encoded as 0 and 1; the "module" options value encoded as a 0; as well as a feature counting the number of active options, for a total of 9.467 features.

4.2 Learning algorithms

Our three approaches rely on Decision Tree, and we used the CART implementation in scikit-learn[33], a widely used state-of-the-art Python library for machine learning. We explored a wide range of hyperparameters, such as the maximum depth of the Tree, or the loss function for regression approaches between MSE and Friedman MSE, in a grid-search fashion to optimize the efficiency of the Decision Trees. We also made vary the training set size from 10% to 70% and used the remaining (at least 30% to avoid overfitting) as test set to study its impact on the accuracy, the sampling being made at random among the datasets, as Pereira *et al.* [34] show that this sampling method is a strong baseline overall. We repeated each iteration from 5 to 20 times and report on the average value, to reduce the impact of randomness on the experimentation.

Specialized Regression. This novel technique relies entirely on the Regression Tree from scikit-learn. The difference is made on the dataset. The performance value, which is the learning target, is modified to suit a specific threshold. However, the amount of the modification can be any value, and we investigate multiple different values we will call *gap*. As using a fixed value might not make sense for some performance values, we choose gap values depending on the performance values found in each dataset: maximum, mean, ½ of the mean and ¼ of the mean.

4.3 Threshold influence

The threshold is the performance value for which a configuration becomes acceptable or not. To investigate a possible influence of this value on the accuracy of the models, we repeat the learning phase using different thresholds, hence specializing each model to a particular threshold. For the regression approach, we do not repeat the learning process, since a regression model is threshold-agnostic. Thus, we repeated only the accuracy measurement. We choose 5 threshold values based on the performance distribution for each dataset, namely 10%, 20%, 50%, 80% and 90% quantiles.

4.4 Metrics

To measure the accuracy of the Decision Trees classification, we use the balance accuracy[8], which is a combination of sensitivity and specificity:

$$Balanced\ accuracy = \frac{1}{2}\left(\frac{TP}{TP+FN} + \frac{TN}{TN+FP}\right)$$

with TP for True Positive, TN for True Negative, FN for False Negative and TP for False Positive. The balanced accuracy heavily takes in account the influence of imbalanced class distribution, which inevitably happens due to the threshold variation in the experiment. In the remaining of the paper, we will refer to balanced accuracy as "accuracy".

4.5 Tree-based Feature selection

We trained 20 Random Forests on 10% of the training set and extracted a Feature Ranking List, except for Linux kernel, where we used the List provided with the dataset [1]. We then repeated the learning as much as needed to find the optimal number of options.

We consider 6 different strategies:

(1) Classification
(2) Classification with Feature Selection
(3) Regression
(4) Regression with Feature Selection
(5) Specialized Regression
(6) Specialized Regression with Feature Selection

As mentioned, we evaluated all 6 strategies over 5 different thresholds, and with 4 different training sizes.

5 RESULTS

In this section, we discuss the answers to our research questions defined in Section 4.

5.1 Results RQ1 and RQ2—Accuracy

Table 2 is a summary of the accuracy over all thresholds for each system with 70% training set size. With the exception of the SQLite system, we report an accuracy of more than 90% overall, showing

System	All strategies	Classification	Classification FS	Regression	Regression FS	Spec. Regr.	Spec. Regr. FS
Apache	95.3% (±2.1)	94.3% (±2.3)	92.3% (±3.0)	93.1 (±2.7)	93.3% (±3.3)	92.8% (±2.7)	93.5% (±3.6)
BerkeleyC	99.9% (±0.3)	99.8% (±0.4)	99.9% (±0.3)	99.8 (±0.4)	99.8% (±0.3)	99.5% (±0.6)	99.6% (±0.5)
BerkeleyJ	93.3% (±9.4)	92.9% (±9.2)	92.4% (±9.8)	92.1 (±9.5)	92.4% (±10.1)	90.0% (±12.4)	91.2% (±10.8)
Dune	93.6% (±2.4)	92.8% (±1.7)	92.2% (±1.8)	91.2 (±3.2)	91.9% (±2.9)	92.7% (±3.2)	92.9% (±3.1)
HIPAcc	97.6% (±0.9)	96.6% (±1.1)	97.1% (±0.7)	95.5 (±2.3)	95.6% (±2.3)	94.9% (±4.2)	94.9% (±4.4)
HMSGP	97.3% (±2.1)	96.5% (±2.5)	96.9% (±2.4)	96.7 (±2.2)	97.2% (±2.0)	96.4% (±2.6)	97.0% (±2.3)
LLVM	94.6% (±4.4)	93.8% (±5.2)	94.0% (±5.1)	91.8 (±5.5)	93.2% (±4.7)	93.2% (±3.8)	93.7% (±4.0)
SQLite	74.9% (±10.4)	73.7% (±9.0)	73.4% (±9.0)	70.2 (±8.2)	70.6% (±8.4)	70.1% (±13.6)	70.6% (±13.4)
Linux	92.6% (±4.0)	91.1% (±2.4)	91.3% (±2.8)	91.2 (±2.9)	91.4% (±3.3)	91.8% (±4.2)	92.0% (±4.7)

Table 2: Average accuracy and standard deviation per system and learning technique over all thresholds and for 70% training set size. FS stands for *Feature Selection* while *All Strategies* mean that we systematically choose the best strategy among the 6.

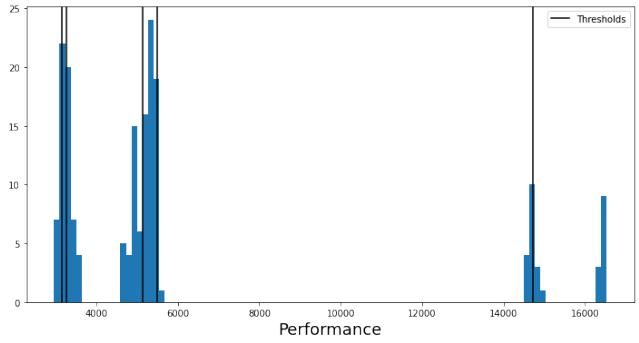


Figure 2: Performance Distribution for BerkeleyJ

that Decision Tree is an algorithm able to handle the complexity of configurable systems for the specialization task in most cases. We observe that the best results are held by the *Classification* strategy on 6 out of 9 systems. However, since the difference between strategies is very slight, we can consider the 6 strategies to be accurate. Furthermore, there is no learning strategy better than another on all thresholds. For some systems, a high standard deviation can be observed due to thresholds variations within a subject system. For instance, for BerkeleyJ[2] the accuracy for 70% training set size (T.S.) goes from 77.2% to 100%, respectively at 10% and 50% of A.C. (acceptable configurations), which is tied to the threshold variation. This can be explained by the very inconsistent performance distribution shown in Figure 2.

> (RQ1) **Decision Tree** is an accurate algorithm for performance specialization, with more than **90% accuracy** for 8 out of 9 systems, including Linux.

5.1.1 Strategies. In this section, we report mainly on the tables present in our supplementary material[2], where we detailed results over all strategies and systems for each considered threshold and training size. Here, due to space limitations, we add the detailed results only for Apache and Linux (see Tables 3 and 4). Notice that the results on Table 2 (for all systems) are referent to 70% training set size and an average over all thresholds.

Classification. As seen previously, classification is overall the best strategy, and its strength resides mainly in the restrictive threshold of 10% or 20% A.C. For Apache, at 10% A.C. and 10% and 20% T.S., the accuracy is more than 3 points higher on classification than the two other strategies. For HIPAcc, at 10% A.C. that difference is from 4 to 5 points, while regression shows 92.2% accuracy at 70% T.S., classification shows 97.3% accuracy.

Regression. The overall accuracy of regression is very similar to specialized regression. For 1 out of 8 systems (HMSGP) it is the best solution. However, regression is a threshold-agnostic approach. It tends to be very sensible to extreme threshold (10% and 90% A.C.), which can be seen as a limitation. HMSGP and Linux are the exceptions, where regression shows itself the best strategy on many cases. For Linux, regression is the best at 50% A.C. on all T.S., and at low T.S. and A.C. We note also that regression is often better than classification on higher thresholds, and better than specialized regression on lower thresholds.

Specialized Regression. This approach shines at higher thresholds, being the best strategy in almost all cases at 80% and 90% A.C. The improvement can be very important, for instance Apache at 90% A.C. and 10% T.S., specialized regression is better by 6 and 9 points over regression and classification. About the gap influence (Section 4.3), most of the time the minimum value (¼ of the mean) gives the best results, and in some cases (for Berkeley C and Berkeley J) it's the maximum value.

> (RQ2) The 6 strategies are particularly **sensitive to performance thresholds**. Classification is the most efficient on low thresholds, specialized regression on high thresholds, while regression proves itself a good middle ground.

5.2 Results RQ3—Feature selection

In Tables 3 and 4, the *influence of feature selection is represented by the value inside brackets*. In the vast majority of cases, the use of feature selection improves the results, though the improvement can be very low (less than 1 point). The most significant improvements

[2]https://raw.githubusercontent.com/HugoJPMartin/SPLC2021/master/tables.pdf

are for LLVM (see in our supplementary material), with cases showing up to 10.3 points more than their counterpart without feature selection.

On the other hand, some cases show that feature selection does not improve the accuracy (mostly for classification strategy). We report one case where the feature selection has a significantly negative impact with regression, while there are 21 of such cases (out of 180 possible cases per strategy) for classification.

Beside, the optimal number of options to select is very variable depending on threshold and training set size. We did not identify particular pattern to predict that number beforehand, so the number of selected options should be considered as a new hyperparameter for the Decision Tree.

> (RQ3) Feature selection is a reliable way to produce **more accurate results** (improvements up to 10%), although the increase is often insignificant. Note that classification-based learning sometimes does not take advantage of feature selection – using all features lead to better results.

5.3 Results RQ4—Training time

The training time of a Decision Tree is in the order of milliseconds (except for Linux). It is up to 22 ms for Specialized Regression on HIPAcc, which is the system with the largest configuration dataset (see Table 1). The training time reduction after feature selection is very slight. The most significant improvement is the reduction from 6.4 ms to 3.5 ms (for Regression Decision Tree on SQLite). Proportionally, it is a 45% time reduction. However, the reduction is imperceptible for a human.

For Linux, the training time of a Decision Tree is 54 seconds for Classification, 86 seconds for Regression and 93 seconds for Specialized Regression. With feature selection however, these training times are reduced to 0.5 second for Classification, 2.3 seconds for Regression and 1 second for Specialized Regression. These improvements are massive, cutting down the training time from 40 folds up to 100 folds, making them fit for a real-time usage.

> (RQ4) The training time of Decision Trees is in the order of milliseconds, which makes it very affordable as a learning technique, except for Linux which takes a minute or more. For Linux, the use of feature selection makes it possible to cut down that training time to the order of second(s).

6 DISCUSSION

Guidelines. Using the empirical knowledge we acquired during the experiments, we aim to propose some guidelines on how to use the different strategies for automated performance specialization.

Due to the cost of learning, or the specialization model being shipped as is, it is not always possible to train a new model each time it is needed ("on the fly"). This computational cost can be a barrier in terms of user experience. For instance, end-users may not be interested to wait almost a minute while maintainers may accept the necessary time to hopefully get accurate specialization. There are two scenarios:

- **You cannot train a model on the fly:**
  - The **default strategy** is *regression*, as being able to handle all thresholds at once and quite accurately.
  - The **profile strategy** is *classification and specialized regression*. Instead of having one generic model, it is possible to focus on one (or more) profile, i.e. threshold. Classification is better for highly constraining specialization, while specialized regression is better for slightly constraining specialization.
  - Note that is is possible to leverage multiple models depending on the needs, and having some particular profiles, backed up by a regression model for other cases.
- **You can train a model on the fly:** When the specialization model can be re-trained for a specific performance threshold, it is better not to use regression, but to use classification for highly constraining specialization and specialized regression for slightly constraining specialization.

Feature selection should be considered in almost every cases. When the training time is negligible, the cost of finding the optimal number of options is negligible too with high chances to have a more accurate model. When the training time is not negligible, such as for Linux, the reduced training time makes it worth to explore a few numbers of options.

Good regressor, bad classifier. "A good regressor should give out a good classifier" is an intuition that one could have when thinking about using a regressor to perform a classification task based on a continuous value, such as specialization. While this happens to be true in a lot of cases, it also happens to be very wrong in some cases. During our experiment, we observe a lot of different regressors and computed both their balanced accuracy and Mean Absolute Percentage Error (MAPE), a regression error metric, and we noticed some of them to go completely against the mentioned intuition. For instance, we have a quite good regressor, with 6% error rate, but with a quite bad balanced accuracy at just under 70%. On the other hand, on the same system, we have a very good classifier, presenting 100% balanced accuracy, but sitting at almost 45% error rate, one of the worst we found on that system.

Safety and flexibility. If we only used the balanced accuracy as the most fair and general metric, some other metric can reveal interesting aspects of the classification. *Precision* measures how much of the predicted acceptable configurations are actually acceptable, and indicates the safety of the evaluated classifier. *Recall* measures how much of all acceptable configurations are actually considered by the classifier, and indicates the flexibility. Except in the rare cases of finding a perfect classifier, there is always a trade-off between flexibility and safety. One interesting aspect of the Decision Tree is that for each rule, it can deliver a probability of *acceptability*, and this can be used to tweak the rules, either toward flexibility, or toward safety.

Active learning. In our experiment, we only considered a single step of learning to create a model. However, in future works, we can consider multiple steps of learning, and leveraging the interpretability of the Decision Trees to find the most error-prone rules in order to refine them by sampling around the misclassified configurations. It is also possible to use the Feature Ranking List to

Training set size	Acceptable configurations				
	10%	20%	50%	80%	90%
	Classification				
19 (10%)	**87.4** (-3.5)	**88.5** (+1.4)	83.2 (+1.4)	85.3 (+1.6)	77.1 (-1.3)
38 (20%)	**90.2** (-3.3)	**90.9** (+1.0)	**87.4** (+0.1)	88.5 (-0.6)	81.7 (+0.5)
96 (50%)	**91.7** (-0.9)	**91.9** (-0.5)	**90.5** (-0.8)	95.0 (-1.5)	85.1 (-0.3)
134 (70%)	**95.2** (-2.7)	**93.6** (-0.8)	**93.1** (-2.8)	97.7 (-0.9)	91.6 (-2.7)
	Regression				
19 (10%)	83.7 (+0.5)	87.6 (+3.8)	**84.5** (+0.9)	91.9 (+7.1)	80.5 (+2.8)
38 (20%)	86.7 (+0.9)	90.6 (+3.1)	86.8 (+1.0)	91.9 (+3.6)	84.0 (+3.8)
96 (50%)	91.3 (+0.6)	91.7 (+2.3)	88.1 (+0.2)	94.9 (+1.3)	89.2 (+0.1)
134 (70%)	93.3 (+0.3)	92.4 (+0.7)	89.9 (+0.6)	**98.5** (+1.2)	93.4 (+0.3)
	Specialized Regression				
19 (10%)	83.0 (+1.1)	85.9 (+1.6)	83.7 (+1.6)	**92.1** (+4.5)	**86.5** (+4.8)
38 (20%)	86.8 (+2.1)	86.7 (+1.2)	86.4 (+0.4)	**92.3** (+1.1)	**89.2** (+1.8)
96 (50%)	90.6 (+0.1)	88.9 (+0.3)	87.8 (+0.2)	**95.7** (+1.7)	**93.8** (+2.2)
134 (70%)	93.0 (+0.2)	91.2 (+0.8)	89.9 (+0.7)	98.1 (+1.8)	**95.9** (+1.5)

Table 3: Decision tree classification accuracy on performance specialization for Apache on three strategies. The difference of feature selection on accuracy is represented by the value inside brackets. Bold represents the best result among other strategies (including feature selection).

Training set size	Acceptable configurations				
	10%	20%	50%	80%	90%
	Classification				
9256 (10%)	84.4% (+1.7)	88.4% (+0.2)	90.3% (+0.2)	91.8% (+0.7)	91.6% (+2.8)
18512 (20%)	85.4% (+0.4)	89.0% (+0.4)	91.1% (-0.0)	92.5% (+1.0)	92.4% (+0.2)
46281 (50%)	**87.3%** (-0.4)	**90.1%** (-0.2)	92.1% (+0.2)	93.1% (+0.4)	93.5% (+1.2)
64793 (70%)	**87.4%** (-0.4)	**89.9%** (+0.0)	92.6% (-0.2)	93.4% (+0.4)	93.7% (+1.3)
	Regression				
9256 (10%)	**85.1%** (+1.8)	**88.5%** (+1.4)	**91.6%** (+1.3)	92.0% (+0.8)	92.0% (+2.4)
18512 (20%)	**86.1%** (+1.9)	**89.5%** (+1.2)	**92.0%** (+0.3)	92.9% (+1.0)	92.4% (+0.8)
46281 (50%)	86.8% (-1.0)	89.7% (+0.0)	**92.9%** (+0.7)	93.8% (+0.6)	94.2% (+1.2)
64793 (70%)	86.8% (-0.5)	89.9% (+0.2)	**92.9%** (-0.1)	93.9% (+0.2)	94.1% (+1.0)
	Specialized Regression				
9256 (10%)	85.0% (+2.0)	87.2% (+1.1)	91.5% (+1.0)	**94.8%** (+0.7)	**95.7%** (+0.5)
18512 (20%)	84.4% (+0.6)	87.6% (-0.0)	91.5% (+0.8)	**95.1%** (+0.7)	**96.3%** (+0.6)
46281 (50%)	84.7% (+0.9)	87.8% (-0.1)	92.3% (+0.4)	**95.6%** (+0.7)	**96.8%** (+0.7)
64793 (70%)	86.2% (-0.3)	89.1% (-0.4)	92.8% (+0.5)	**95.8%** (+0.6)	**97.0%** (+0.6)

Table 4: Decision tree classification accuracy on performance specialization for Linux kernel on three strategies. The difference of feature selection on accuracy is represented by the value inside brackets. Bold represents the best result among other strategies (including feature selection).

focus the sampling toward the most influential options and avoid the ones without influence.

Going further with constraints. The focus of this paper was accuracy since it is crucial in many specialization scenarios. The readability and comprehension of constraints might also be of interest for practitioners. It is an interesting research direction that requires *e.g.,* human studies and controlled experiments. The validation of the quality of the constraints with domain experts (*e.g.,* Linux developers) is in our agenda. The use of feature selection might also be of interest here, since rules and constraints will operate over a limited set of options. More aggressive pruning strategies [25] can also be envisioned with tradeoffs between accuracy and interpretability.

7 THREATS TO VALIDITY

In this section, we describe some *internal* and *external* threats to the validity of this study.

7.1 Internal validity

Hyperparameters for Decision Trees. Just like many learning algorithms, Decision Trees exhibit hyperparameters that can impact the performance of the algorithm (accuracy as well as training time). To mitigate this threat, we explored a wide range of them in a grid-search fashion *i.e.,* we test all combinations of multiple values for each of them. Complete results and scripts are available online.

Threshold variation. The performance threshold – the limit set by the user to define an acceptable configuration – can make vary a lot the accuracies of the learning techniques. We considered this aspect by using different thresholds, based on the performance distributions of each system and dataset. However, this may rise a problem: as we can see in Figure 2, the 10% and 20% thresholds are very close and also very constraining, which might explain the low accuracy of the specialization models. In practical terms, these thresholds do not correspond to any realistic expectation from a user. Despite this threat, we did observe the phenomenon for only one system while none of the 6 learning techniques have been favored. However, we warn that this is a threat that should be considered for any future work or transfer in a real use case.

Sampling. For this experiment, we focused only on random sampling to avoid spreading ourselves too much. Pereira *et al.* [34] shows that random sampling is a strong baseline, but also warns about the potentially strong influence of different sampling strategies, especially when comparing learning techniques. The exploration of other sampling techniques is definitely in the scope of future work.

7.2 External validity

Workload influence. It is well known that the input workload of a system process, for instance the input video for a video encoder, is another source of variability that should be taken into account. As we relied on datasets shared by the community, we did not focus on this aspect of variability at this time. Beside, all systems are not impacted by workload, such as the binary size of Linux kernel which is only dependent on the configuration. Thus, we focused on a single workload to be able at making robust and reliable statements about a specific strategy. We are confident that some method of transfer learning should be able to tackle other workloads. Demonstrating the validity of the approach for other workloads is part of our future work.

System generalization. As all empirical study, the conclusions are fully reliable on the studied systems. Thus, we acknowledge that the use of another system may lead to different results. However, once we are able to demonstrate evidences of good results to a set of real-world systems widely used in the literature, including Linux kernel, we are quite confident in the generalization of our approach. Still, conducting experiments with other systems is an important next step, which is part of our future work.

8 CONCLUSION

As software systems become more and more configurable, it becomes harder for users to correctly configure them or to reach a certain goal, being functional or in terms of performance. Specialization is a technique that adds constraints to a system in order to assist users in reaching a predefined goal while sacrificing the configurability as little as possible. However, a manual specialization process to create the constraints can be error-prone, time-consuming, and heavily dependent on a knowledge that is hard to formalize or simply not available. The recent rise of data and machine learning appears to be a good candidate to automate the specialization process and complement or replace the expert knowledge.

We explored the decision tree learning algorithm, for how suited it is to extract rules that can be retrofitted into variability models of configurable systems. We used and compared both classification and regression trees, as well as a novel variant of regression tree refined toward the specialization problem, that we called specialized regression. We also used tree-based feature selection to investigate how well it can be combined with specialization. We performed an empirical study to evaluate the ability of Decision Trees to accurately constrain the configuration space of the systems for the specialization problem and performance properties. We used dataset well known in the community, as well as taking up the challenge of the Linux kernel and its thousands of options.

Our results showed that the learning models are more than 90% accurate on 8 out 9 systems, including Linux. Every strategy is efficient, but each has its own strength regarding different thresholds, which makes them all worth considering. Feature selection proved itself a good and reliable way to improve accuracy, and also to reduce the training time. If for most systems, the training time is very low, in the order of the milliseconds, in the case of the Linux kernel it can take more than one minute and feature selection reduces that time to one or two seconds.

We have exposed a panorama of accurate learning techniques for the performance specialization problem. As future work, we plan to assess how the inferred configuration knowledge relates to domain experts' knowledge (*e.g.,* Linux developers). It will require to investigate how readable and comprehensible are constraints extracted from the specialization process. Another research direction is how this knowledge transfers across deep variability [23] (*e.g.,* versions and workloads of a system).

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments. This research was funded by the ANR-17-CE25-0010-01 VaryVary project; and PNPD/CAPES (88887.473590/2020-00).

REFERENCES

[1] Mathieu Acher, Hugo Martin, Juliana Pereira, Arnaud Blouin, Jean-Marc Jézéquel, Djamel Khelladi, Luc Lesoil, and Olivier Barais. 2019. Learning Very Large Configuration Spaces: What Matters for Linux Kernel Sizes. (2019).

[2] Mathieu Acher, Paul Temple, Jean-Marc Jezequel, José A Galindo, Jabier Martinez, and Tewfik Ziadi. 2018. VaryLaTeX: Learning Paper Variants That Meet Constraints. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems.* ACM, 83–88.

[3] Juliana Alves Pereira, Hugo Martin, Mathieu Acher, Jean-Marc Jézéquel, Goetz Botterweck, and Anthony Ventresque. 2019. *Learning Software Configuration Spaces: A Systematic Literature Review.* Research Report. Univ Rennes, Inria, CNRS, IRISA. https://doi.org/10.1145/nnnnnnn.nnnnnnn

[4] Benoit Amand, Maxime Cordy, Patrick Heymans, Mathieu Acher, Paul Temple, and Jean-Marc Jézéquel. 2019. Towards Learning-Aided Configuration in 3D Printing: Feasibility Study and Application to Defect Prediction. In *VaMoS 2019 - 13th International Workshop on Variability Modelling of Software-Intensive Systems.* Leuven, Belgium, 1–9. https://hal.inria.fr/hal-01990767

[5] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. 2016. Automatic Detection and Removal of Conformance Faults in Feature Models. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 102–112. https://doi.org/10.1109/ICST.2016.10

[6] Eduard Baranov, Axel Legay, and Kuldeep S. Meel. 2020. Baital: an adaptive weighted sampling approach for improved t-wise coverage. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1114–1126. https://doi.org/10.1145/3368089.3409744

[7] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[8] Kay Henning Brodersen, Cheng Soon Ong, Klaas Enno Stephan, and Joachim M. Buhmann. 2010. The Balanced Accuracy and Its Posterior Distribution. In *2010 20th International Conference on Pattern Recognition*. 3121–3124. https://doi.org/10.1109/ICPR.2010.764

[9] Sourav Chakraborty and Kuldeep S. Meel. 2019. On Testing of Uniform Samplers. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 7777–7784. https://doi.org/10.1609/aaai.v33i01.33017777

[10] Girish Chandrashekar and Ferat Sahin. 2014. A survey on feature selection methods. *Computers & Electrical Engineering* 40, 1 (2014), 16–28.

[11] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2005. Formalizing Cardinality-based Feature Models and their Specialization. In *Software Process Improvement and Practice*. 7–29.

[12] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2005. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice* 10, 2 (2005), 143–169.

[13] Abdellatif El Afia and Malek Sarhani. 2017. Performance prediction using support vector machine for the configuration of optimization algorithms. In *2017 3rd International Conference of Cloud Computing Technologies and Applications (CloudTech)*. IEEE, 1–7.

[14] Angelo Gargantini, Justyna Petke, and Marco Radavelli. 2017. Combinatorial interaction testing for automated constraint repair. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 239–248.

[15] Angelo Gargantini, Justyna Petke, and Marco Radavelli. 2017. Combinatorial Interaction Testing for Automated Constraint Repair. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 239–248. https://doi.org/10.1109/ICSTW.2017.44

[16] Alexander Grebhahn, Carmen Rodrigo, Norbert Siegmund, Francisco J Gaspar, and Sven Apel. 2017. Performance-influence models of multigrid methods: A case study on triangular grids. *Concurrency and Computation: Practice and Experience* 29, 17 (2017), e4057.

[17] Huong Ha and Hongyu Zhang. 2019. DeepPerf: performance prediction for configurable software with deep sparse neural network. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 1095–1106. https://dl.acm.org/citation.cfm?id=3339642

[18] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. 2019. Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack. *Empirical Software Engineering (ESE)* 24, 2 (July 2019), 674–717. https://doi.org/10.1007/s10664-018-9635-4 Empirical Software Engineering journal.

[19] Ruben Heradio, David Fernández-Amorós, José A. Galindo, and David Benavides. 2020. Uniform and scalable SAT-sampling for configurable systems. In *SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020, Volume A*, Roberto Erick Lopez-Herrejon (Ed.). ACM, 17:1–17:11. https://doi.org/10.1145/3382025.3414951

[20] Arnaud Hubaux, Patrick Heymans, Pierre-Yves Schobbens, Dirk Deridder, and Ebrahim Khalil Abbasi. 2011. Supporting Multiple Perspectives in Feature-based Configuration. *Software and Systems Modeling* (2011), 1–23.

[21] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. 2019. Distance-Based Sampling of Software Configuration Spaces. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[22] Thomas Krismayer, Rick Rabiser, and Paul Grünbacher. 2017. Mining Constraints for Event-Based Monitoring in Systems of Systems. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Press, 826–831.

[23] Luc Lesoil, Mathieu Acher, Arnaud Blouin, and Jean-Marc Jézéquel. 2021. Deep Software Variability: Towards Handling Cross-Layer Configuration. In *VaMoS 2021 - 15th International Working Conference on Variability Modelling of Software-Intensive Systems*. Krems / Virtual, Austria. https://hal.inria.fr/hal-03084276

[24] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 643–654.

[25] John Mingers. 1989. An empirical comparison of pruning methods for decision tree induction. *Machine learning* 4, 2 (1989), 227–243.

[26] Christoph Molnar. 2020. *Interpretable Machine Learning*. Lulu. com.

[27] Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory. 2019. Uniform Random Sampling Product Configurations of Feature Models That Have Numerical Features. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A* (Paris, France) *(SPLC '19)*. Association for Computing Machinery, New York, NY, USA, 289–301. https://doi.org/10.1145/3336294.3336297

[28] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2018. Faster discovery of faster system configurations with spectral learning. *ASE* (2018), 1–31.

[29] Vivek Nair, Zhe Yu, Tim Menzies, Norbert Siegmund, and Sven Apel. 2018. Finding Faster Configurations Using Flash. *IEEE Transact. on Software Engineering* (2018).

[30] Lina Ochoa, Oscar Gonzalez-Rojas, Alves Pereira Juliana, Harold Castro, and Gunter Saake. 2018. A systematic literature review on the semi-automatic configuration of extended product lines. *Journal of Systems and Software* 144 (2018), 511–532.

[31] Jeho Oh, Paul Gazzillo, and Don Batory. 2019. T-Wise Coverage by Uniform Sampling. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A* (Paris, France) *(SPLC '19)*. Association for Computing Machinery, New York, NY, USA, 84–87. https://doi.org/10.1145/3336294.3342359

[32] Terence Parr, Kerem Turgutlu, Christopher Csiszar, and Jeremy Howard. 2018. Beware Default Random Forest Importances. last access: july 2019.

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

[34] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, and Jean-Marc Jézéquel. 2020. Sampling Effect on Performance Prediction of Configurable Systems: A Case Study. In *ICPE '20: ACM/SPEC International Conference on Performance Engineering, Edmonton, AB, Canada, April 20-24, 2020*, José Nelson Amaral, Anne Koziolek, Catia Trubiani, and Alexandru Iosup (Eds.). ACM, 277–288. https://doi.org/10.1145/3358960.3379137

[35] Justyna Petke, Shin Yoo, Myra B. Cohen, and Mark Harman. 2013. Efficiency and Early Fault Detection with Lower and Higher Strength Combinatorial Interaction Testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. ACM, 26–36.

[36] Quentin Plazar, Mathieu Acher, Gilles Perrouin, Xavier Devroey, and Maxime Cordy. 2019. Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?. In *ICST 2019 - 12th International Conference on Software Testing, Verification, and Validation*. Xian, China, 1–12. https://hal.inria.fr/hal-01991857

[37] Safdar Aqeel Safdar, Hong Lu, Tao Yue, and Shaukat Ali. 2017. Mining cross product line rules with multi-objective search and machine learning. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 1319–1326.

[38] Norbert Siegmund, Alexander Grebhahn, Christian Kästner, and Sven Apel. 2015. Performance-Influence Models for Highly Configurable Systems. In *ESEC/FSE'15*.

[39] Terry Speed. 2003. *Statistical analysis of gene expression microarray data*. Chapman and Hall/CRC.

[40] Paul Temple, Mathieu Acher, Jean-Marc Jézéquel, and Olivier Barais. 2017. Learning Contextual-Variability Models. *IEEE Software* 34, 6 (2017), 64–70.

[41] Paul Temple, José Angel Galindo Duarte, Mathieu Acher, and Jean-Marc Jézéquel. 2016. Using Machine Learning to Infer Constraints for Product Lines. In *Software Product Line Conference (SPLC)*. Beijing, China.

[42] Paul Temple, Gilles Perrouin, Mathieu Acher, Battista Biggio, Jean-Marc Jézéquel, and Fabio Roli. 2021. Empirical assessment of generating adversarial configurations for software product lines. *Empir. Softw. Eng.* 26, 1 (2021), 6. https://doi.org/10.1007/s10664-020-09915-7

[43] Pavel Valov, Jianmei Guo, and Krzysztof Czarnecki. 2015. Empirical Comparison of Regression Methods for Variability-Aware Performance Prediction. In *SPLC'15*.

[44] Pavel Valov, Jean-Christophe Petkovich, Jianmei Guo, Sebastian Fischmeister, and Krzysztof Czarnecki. 2017. Transferring Performance Prediction Models Across Different Hardware Platforms. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ACM, 39–50.

[45] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. 2018. A classification of product sampling for software product lines. In *Proceeedings of the 22nd International Conference on Systems and Software Product Line-Volume 1*. ACM, 1–13.

[46] Dennis Westermann, Jens Happe, Rouven Krebs, and Roozbeh Farahbod. 2012. Automated Inference of Goal-Oriented Performance Prediction Functions. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 190–199.

[47] Cemal Yilmaz, Myra B Cohen, and Adam A Porter. 2006. Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces. *IEEE Transactions on Software Engineering* 32, 1 (2006), 20–34.