Learning-Based Performance Specialization of Configurable Systems

Paul Temple, Mathieu Acher, Jean-Marc Jézéquel, Léo Noel-Baron, José A Galindo

Learning-Based Performance Specialization of Configurable Systems

Paul Temple, Mathieu Acher, Jean-Marc Jézéquel, Léo Noel-Baron
Univ Rennes, IRISA
Rennes, France
Emails: firstname.lastname@irisa.fr
José A. Galindo
University of Sevilla
Sevilla, Spain
Email: jagalindo@us.es

Abstract—
A large scale configurable system typically offers thousands of options or parameters to let the engineers customize it for specific needs. Among the resulting many billions possible configurations, relating option and parameter values to desired performance is then a daunting task relying on a deep know how of the internals of the configurable system. In this paper, we propose a staged configuration process to narrow the space of possible configurations to a good approximation of those satisfying the wanted high level customer requirements. Based on an oracle (e.g. a runtime test) that tells us whether a given configuration meets the requirements (e.g. speed or memory footprint), we leverage machine learning to retrofit the acquired knowledge into a variability model of the system that can be used to automatically specialize the configurable system. We validate our approach on a set of well-known configurable software systems. Our results show that, for many different kinds of objectives and performance qualities, the approach has interesting accuracy, precision and recall after a learning stage based on a relative small number of random samples.

Index Terms—software product lines; machine learning; constraints and variability mining; software testing; variability modeling

1 Introduction

Most modern software systems are configurable and offer (numerous) configuration options to users. Web servers like Apache, operating systems like the Linux kernel, or a video encoder like x264: all can be highly configured at compile-time or at run-time for delivering the expected functionality, hopefully with an adequate performance (*e.g.,* execution time).

Even with know-how or expertise on the domain, configuring such systems is known to be a laborious activity with lots of trials and errors. Users typically set some Boolean or numerical values for their options, observe the behavior of the configured system, and iterate until obtaining a suitable configuration (*e.g.,* in terms of speed, footprint, size or any other performance related metric). Along the road, it may happen that the specified configuration leads to an error at compile-time or at runtime (*e.g.,* a crash during the execution). It may also happen that some configurations are functionally valid and yet lead to performance issues that are perceived as unacceptable. For example, the execution time is greater than 5 seconds or the quality of the output is less than an expected threshold.

Finding a good combination of options that meets both functional and performance requirements is in general a very challenging problem. Firstly, there may be billions of possible configurations: In practice it is too costly to manually or automatically execute, measure, observe, and test all of them so that only the right ones are retained. Secondly, empirical results show that quantifying the performance influence of each individual option is not sufficient in most cases, since there are numerous complex interactions and constraints between options [1]–[6]. As a result, users have hard time (1) to understand the effects of each configuration option on performance; (2) to express the performance models to reach or optimize. Thirdly, users can hardly find a suitable and complete configuration solely based on performance objectives. The use of an automatic procedure for picking a configuration may involve arbitrary choices that users may not actually want – the performance is OK and even optimized, but the functional requirements are not acceptable. Users usually need a (fine-grained) control and keep some *flexibility* during a multi-stage or multi-step configuration process.

In this article, we introduce the idea of automatically *specializing* a configurable system in such a way that admissible configurations satisfy a given performance objective. In a sense we aim to pre-configure a system so that any remaining choices have a high probability of satisfying the required performance. For instance, we can derive a specialized x264 with a low energy consumption, as defined by a value threshold. Software integrators can then embed x264 in small devices: They can still fine-tune their configurations w.r.t their functional constraints (*e.g.,* hardware characteristics of the targeted device). Importantly, our specialization method does not aim to select, tune, or optimize one config-

uration. We rather aim to offer a *set* of satisfying configurations. The specialized system remains (highly) configurable but constrained to satisfy the performance objective for any remaining configurations.

Our performance specialization method proceeds as follows. *First*, a performance objective is defined (*e.g.,* execution time is less than 10 seconds). Importantly the objective is not a performance model of the entire system, but simply a threshold value for which the system behavior is acceptable or not. Users of our method just have to specify a performance requirement (*e.g.,* "execution time less than 10 seconds"), the rest is automatic. An automated procedure (*e.g.,* a runtime test) operationalizes the checking of the required performance as measured on a set of relevant benchmarks. *Second*, we leverage machine learning techniques to classify configurations in two classes: configurations we want to keep because they meet the performance objective; and configurations that are not acceptable and that we want to discard. Thanks to learning, we expect to classify configurations that were never executed and measured before with a good precision and recall. Our empirical results confirm that only a small sample of configurations is needed to perform accurate predictions. *Third*, we synthesize constraints among configuration options to narrow down the space of possible configurations. That is, we directly retrofit the knowledge acquired with machine learning into a variability model. State-of-the-art solving or search-based techniques can then exploit such constraints to pilot a (multi-step) configuration process [7]–[16], [16]–[20]. As a result, the configuration experience should no longer be a semi-blinded endeavor based on numerous trials and errors. On the contrary, users of the specialized system should be guided at each configuration step (*e.g.,* propagation of options values) until finding a configuration satisfying the required performance and their own specific needs.

With our method, developers can pre-configure and automatically build specialized configurable systems that meet a certain objective (*e.g.,* energy consumption or execution time). Non-acceptable configurations are automatically discarded, leading to safe and still flexible configurable systems. Vendors or developers can deliver new bundles or product lines for targeting specific customers, profiles, use cases, hardware settings, or safety-critical contexts. Our method also has a practical interest when engineering adaptive systems or dynamic software product lines [21]–[23] that are highly configurable systems. At runtime, the decision-making engine will typically operate over a reduced and safe configuration space thanks to our specialization method.

In this article, we make the following contributions:

- We propose an automated method that produces a partially specialized configurable system so that users' configurations almost always meet a desired performance quality. The engineering effort only resides in the specification of an "objective" to indicate what quality is acceptable for the new configurable system.
- We implement the approach and demonstrate its practicality and generality by experiments on real-world configurable software systems. The results show that, for many different kinds of objectives and performance qualities, the approach has interesting accuracy, precision and recall after a learning stage based on a relative small number of random samples.
- We empirically explore the sensitivity of the method w.r.t objective specification and random sample size. A key finding is that it works well when the objective yields to a good separation of the performance distributions into two classes, somehow in between the two extreme cases when the objective is too permissive (allow all configurations, so no specialization is obtained) or too restrictive (no configuration does exist to satisfy the objective).
- We compare our solution with a sound (non learning-based) method. Our empirical results show that our learning-based specialized method produces safer configurable systems. This is especially the case for performance objectives that require to eliminate a large portion of non-acceptable configurations.

This article significantly extends our previous work published at SPLC'16 [24]. In [24], we used learning techniques to correct a configurable video generator. Compared to SPLC'16 paper, we generalize here our specialization method to support performance objectives. It broadens the applicability and the usefulness of our method, since many specialization scenarios are related to performance properties (speed, size, *etc.*). In terms of evaluation, we consider much more subjects, *i.e.,* in total 16 publicly available configurable systems and performance properties. We also study the effect of the sampling size used to train the machine learning technique. Furthermore we consider a third important independent variable: the threshold values of the performance objective. It simply does not exist in our previous work and challenges our learning-based specialization technique. A comprehensive evaluation framework with metrics (*e.g.,* specificity, negative prediction value) has been developed for assessing our method while key findings are precisely related to the effect of performance objectives.

The remainder of the article is structured as follows. Section 2 motivates our work and formalizes the problem. Section 3 describes our specialization method. Section 4 reports on our empirical results. Section 5 discusses threats to validity. Section 6 reviews related work. Section 7 concludes the article and outlines future work.

2 Motivation and Problem Statement

What configuration options or parameter value ranges am I still allowed to use when I want to satisfy some performance (or non-functional properties)? This question arises every time users configure a system, for example, when Web developers write a configuration file for a server like Apache, or when administrators explore the compilation options of a database engine like SQLite.

Motivating scenario. Let us consider the case of x264, a tool to encode videos into the H.264 compressed format. x264 offers configuration options such as output quality, encoder types, and encoding heuristics (*e.g.,* parallel encoding on multiple processors or encoding with multiple reference frames). Users configure x264 by setting options values,

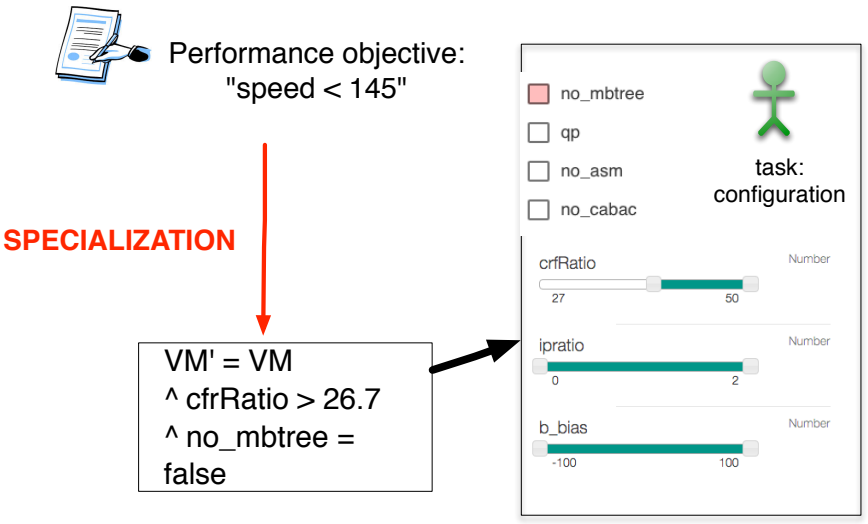


Fig. 1: Configuration of a specialized x264. Given a performance objective, the specialization method infers and fixes some options values (no_mbtree and crfRadtio); users still have some flexibility to configure other options.

through command-line parameters or through graphical user interfaces. Figure 1 shows an excerpt of options: no_mbtree, qp, no_asm, and no_cabac are Boolean options (true or false values) and crfRatio, ipratio, and b_bias are numerical values with different ranges.

Finding a configuration that performs as desired is a time-consuming and error-prone task, even with a good knowledge of the configurable system. Though x264 provides a comprehensive documentation of options [25], the description is in natural language and does not necessary capture all interactions between options. Furthermore it is practically impossible for maintainers or users of a configurable system to test the 10^{27} configurations of x264.

Users typically try some options, look at or measure the result, and iterate again and again until getting a satisfactory performance. A fundamental issue is that changing one individual option may dramatically change the behavior of the system. Empirical results on x264 show that each option and interaction between options can have severe consequences on the execution time, on the quality of the output, or on the energy consumption [1]–[6].

Approach. Our goal is to assist users in reaching configurations that meet both functional and performance requirements. On the one hand, we want to *restrict* the space of reachable configurations. In Figure 1, we have specialized x264 in such a way that each authorized configuration now has an execution time (speed) less than 145s. With regards to performance distribution[1] (see Figure 2), this threshold value means that x264 should be *fast*. On the other hand, we want to keep *open* the configurability (or variability) of the system. That is, we do not want to pick only one suitable configuration, but we rather want to still provide some flexibility to users when configuring the system.

Given a performance objective, our approach *specializes* a configurable system. Users can still configure their systems in different steps or stages (for setting the compression, input video format, blurriness of the result, *etc.*) but any remaining choice has a high probability of satisfying the required performance. For instance, thanks to the specialization of x264 (see Figure 1), no_mbtree is automatically deselected (and must never be activated) while the remaining possible values of crfRatio are now greater than 26.7. It is a real result of our method (see also Section 4 for other empirical results). It is also coherent with information found online [25] that mentions higher values of crfRatio tend to produce low quality videos and are thus faster.

Problem formalization. We assume a *variability model* VM documents the set of options of the configurable system under consideration. Options can take Boolean or numerical values (integer or float). The variability model may already contain some *constraints* that restrict the possible values of options. For example, the inclusion of some Boolean option implies the setting of a numerical value greater than 30. A *configuration* is an assignment of values for all options of a variability model. A configuration is *valid* if its values conform to constraints (*e.g.*, cross-tree constraints over numerical options). Semantically a configurable system characterizes a set of configurations $[\![VM]\!] = \{c \mid c \in [\![VM]\!]\}$.

A *specialized* configurable system is a new system in which all its valid configurations meet a certain *objective*.

An objective can be as simple as "it compiles" or "it passes the test suite". It can also be about some performance aspect of the system, for instance: "the execution time of the system should be less than one second". An objective, denoted OBJ, is a Boolean predicate over a configuration and a value of a non functional property (*e.g.*, execution time).

Formally, given a configurable system cs, we want to obtain a new configurable system: cs' such that

$$[\![VM']\!] = \{c' \mid c' \in [\![VM]\!] \wedge OBJ(c')\} \quad (1)$$

Overall the problem is to synthesize a new variability model VM' such that each valid configuration now yields to an acceptable performance (as defined by an objective).

1. The considered x264 has 10^{27} configurations in total. As it is practically impossible to compute and test all configurations, other researchers [1], [3], [6], [26], [27] have computed a large subset of 69k configurations (see also Section 4 for more details about the datasets).

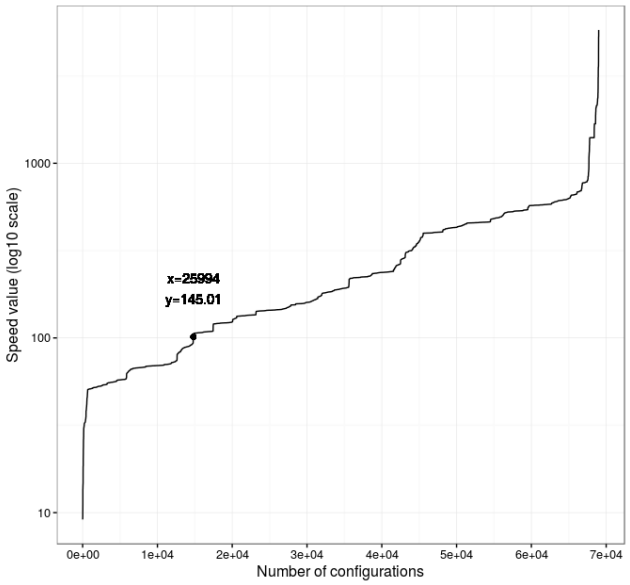


Fig. 2: Number of x264 configurations running under a certain time: X-axis represents a number of configurations; Y-axis represents the execution speed (in seconds) to encode a video benchmark; *e.g.,* about 25994 configurations can encode the video in less than 145.01 seconds.

Simply put, VM' is VM augmented with constraints. Since $[\![VM']\!]$ is a subset of $[\![VM]\!]$, VM' is a specialization [28] of VM. We use the term *specialization* in accordance with the terminology employed in the software product line literature (see *e.g.,* [16], [16]–[20], [28]).

Challenges. For numerous existing configurable systems, the major issue is that we cannot execute and check OBJ for all existing configurations (*e.g.,* 10^{27} for x264). In general, a solution based on a comprehensive enumeration is not possible because of the cost of execution and the cost of measurement. Therefore, in practice, we rather have at our disposal a (small) sample of configurations. Furthermore, performance qualities involve interactions between several options that are hard to quantify and model. Figure 2 shows the performance variations of x264 configurations w.r.t execution speed.

Boolean options like no_mbtree and crfRatio and their interactions influence the execution time. Overall, trying to find constraints explaining those performances is very difficult and leads to highly non-linear functions.

Our goal is to use machine learning techniques to constrain a configurable system with respect to a given performance objective, while still keeping open the configuration space.

A novel problem. Previous learning approaches addressed a complementary yet different problem [1]–[5]: They aimed to *predict* the performance of a given configuration using statistical regression techniques. For example, users may specify some parameters of x264 and a predictor gives the speed value – *without* executing and measuring x264. A prediction can be used for determining whether a configuration is acceptable w.r.t performance objective. If it is not, users have to iterate again until finding a good combination of values that meets their objectives. It remains a semi-blinded endeavor only slightly guided by the users's domain know-how, since non-acceptable configurations are not discarded. Such techniques do not retrofit constraints into the variability model and thus do not address our problem (see Equation 1). In the remainder of the paper, we will show that our specialization problem boils down to a *classification* problem (as opposed to a *regression* problem).

3 Automated Specialization

Users in charge of specializing a configurable system have to specify a performance objective. An automated procedure based on machine learning is then executed to synthesize a new variability model. Figure 3 depicts the different steps of the specialization process.

3.1 From user's objective to oracle

An objective states whether a given performance configuration is *acceptable* and is the only input of our method (see right part of Figure 3). Developers, maintainers, integrators, sellers of product lines, or even end-users can play the role of a specializer and express an objective in a declarative way. For instance, let say the objective is formulated as "the execution time of an x264 encoding on a given benchmark should be less than 145s". In Figure 2, there are about 25994 configurations out of 69000 that are able to encode a video benchmark within 0 to 145 seconds. It means that we want to only retain these 25994 configurations, representing about 37% of configurations of our dataset.

An objective is fed to an automated procedure, called an *oracle*, that takes as input a configuration and returns true or false. As a practical realization, an oracle follows these steps:

- execution or compilation of the configured system given an input configuration;
- measurement over the execution or the result of the compilation. Specializers can develop or simply reuse a procedure to measure a configuration;
- confrontation of the measure with the objective, giving true or false value.

3.2 Configuration sampling and oracle's labelling

Another step in the process is *configuration sampling*. Since deriving and measuring every possible configuration can have significant costs, we need to create a smaller set of configurations. Such configurations are valid w.r.t the original constraints and variability model VM (see left part of Figure 3). Numerous strategies can be considered such as the generation of T-wise configurations, random configurations, *etc.* [3], [29]–[35] In this article, we do not aim to develop or assess specific sampling techniques; we rather use a random strategy.

Now that a sample has been created and an oracle can be executed, we can determine which configurations in the sample fulfill the expected objective. We label configurations as acceptable (w.r.t an oracle) or non-acceptable (see right part of Figure 3). A configuration is characterized as a set of option values and a label, also called *class* hereafter. For example, in configuration c_1, ft_1 has value 1 (representing a

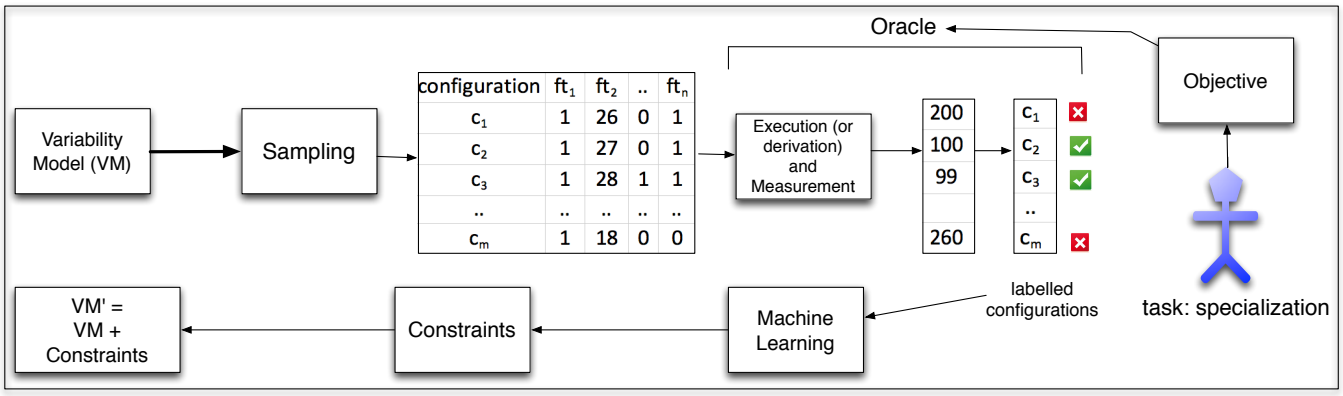


Fig. 3: Sampling, measuring, learning: given a performance objective, there is an automated method for specializing a configurable system

true Boolean value), ft_2 has value 26 (numerical value), *etc.* and ft_n has the true value. The class of c_1 is non-acceptable since its performance measure (200) is superior to 145 in our example. In configuration c_3, ft_1 has the true value, ft_2 has value 28, *etc.* and ft_n has the true value. The class of c_3 is acceptable.

3.3 Machine Learning (ML)

For specializing a configurable system, we need to classify configurations as acceptable or non-acceptable using a sample. Then constraints can then be synthesized for removing non-acceptable configurations. Our goal is to infer, out of a few configurations, what options' values lead to the class acceptable or non-acceptable. In essence, we try to find a function to relate the configuration options to a Boolean response. We need to resolve a *binary classification problem* [36].

Problem reduction. We represent all configurations options (being Boolean features or numerical attributes) of a configurable system as a set $F = \{ft_1, ft_2, \ldots, ft_n\}$ of variables. We consider all variables of F as *predictor* variables. A configuration is encoded as a n-dimensional vector. We have a training sample of m configurations on class variables Y that take values $\{0, 1\}$ (resp. acceptable and non-acceptable configuration). Given training data (c_i, y_i) for $i = 0 \ldots m$, with $c_i \in [\![VM]\!]$ and $y_i \in \{0, 1\}$, we want to learn a *classifier* able to predict the values Y from new F values.

Classification trees. Many approaches have been developed for addressing (binary) classification problems. Decision trees (DT) are a supervised ML technique which tries to classify data into classes [37]. We choose DTs since they are well suited to our specialization problem:

- DTs can handle both numerical and categorical data. In our case, we want to handle Boolean options (features) or numerical options (attributes) both present in the variability models of real-world configurable systems;
- We can extract rules or constraints expressed in propositional logics. We can seamlessly inject such constraints into VM;
- DTs are usually easy to understand and interpret. It can be useful for synthesizing human-readable constraints.

DTs where the class variable Y can take a finite set of values are called *classification trees*[2]. It is the case of our specialization problem since the set of values is either 0 (acceptable) or 1 (non-acceptable).

A classification tree consists of nodes and edges organized in a hierarchy. Nodes test for the values of a certain variable. Edges correspond to the outcome of a test and connect to the next node. In other words, at each level of the tree, a binary decision is made regarding the value of predictor variables (configuration options). Leaves of the tree are terminal nodes that predict the final outcome.

Figure 4 shows an example of a classification tree that has been built on the x264 configurable system with the objective of execution time $< 145s$. This tree has only two depth levels (decisions before reaching leaves). To know the class of a new configuration, one has to simply go through the tree until a leaf is reached. At each level, the path is decided by considering the configuration option indicated in the current node and by comparing its value with the test expressed on edges. In this tree, there are three leaves and two of them are labeled "1", meaning that every configuration falling into these two leaves are considered non-acceptable.

Building classification trees. Algorithms for building classification trees [36], [37] choose a node at each step that "best" splits the set of examples (*i.e.*, configurations in our case). In the ideal case, the distribution of examples in each node is homogeneous so that it mostly contains examples of a single class; the pure node can then be used to predict the class of a large portion of examples. Measures of node impurity based on the distribution of the observed Y values in the node are usually employed. The key challenge is to find an effective strategy for choosing nodes and splitting.

Algorithm and implementation. Different metrics for measuring the homogeneity or entropy of the class variable Y within the subsets have been proposed. ID3, C4.5, and C5.0 algorithms use *information gain* (difference in entropy) for

2. DTs where the target class variable can take continuous values (*e.g.*, real numbers) are called regression trees. Such DTs can be used to specifically address regression problems (*e.g.*, performance prediction problems). Furthermore, specific algorithms have been developed to build regression trees and differ from algorithms used to build classification trees.

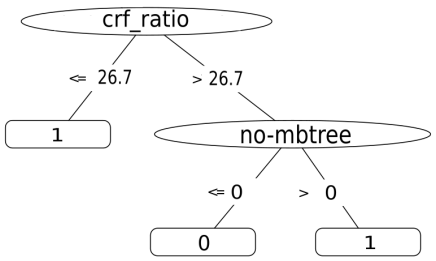


Fig. 4: An example of a classification tree based on x264 system (speed is the performance quality considered)

```
1 !(crf_ratio <= 26.7)
2 !( crfRatio > 26.7 & no_mbtree > 0 )
```

Fig. 5: Constraints extracted from Figure 4

its splitting function, whereas CART uses a generalization of the binomial variance called the *Gini index* [37]–[39]. We rely on scikit-learn [40], a widely used machine learning framework written in Python. Specifically, we use classification tree algorithms of scikit-learn[3] with information gain as splitting criterion. For other parameters of scikit-learn's algorithm, we use default values. The source code of our implementation can be found online [27].

3.4 From decision trees to constraints

It has been shown that rules (or constraints) can be extracted out of decision trees [41]. Since leaves are labeled, we can go through the entire tree and remember choices which lead to leaves that represent configurations that do not fulfill the desired behavior expressed by the oracle. Such paths (built by following those choices) can be considered as constraints to be added in the variability model in order to further exclude configurations that follow those choices. Considering again Figure 4, we can retrieve two paths leading to leaves labeled "1", thus, retrieving two constraints shown in Figure 5. Such constraints, denoted cst, are eventually injected into VM to get VM'. $VM' = VM \wedge cst$ is then exploited to guide users in a multi-stage configuration process.

In Figure 1, we can observe the effect of retrofitting the constraints of Figure 5. When configuring x264, no_mbtree is now automatically set to false. Users are also invited to choose crfRatio values greater than 26.7. In general, solving techniques based on SAT, BDD, CSP, or SMT [8]–[15] can operate over VM' for consistency checking, propagation of values, range fixes, *etc.* Ideally, constraints of VM' prevent all configurations that contradict a given performance objective but still allow users to select any of the remaining configurations. In the following, our primary assessment will be on the ability of our learning-based method to synthesize a semantically sound VM' and set of constraints[4].

3. http://scikit-learn.org/stable/modules/tree.html

4. Developers, maintainers, or experts can read constraints extracted out of decision trees for understanding or debugging configurable systems. The evaluation of the *syntactical* properties of constraints, however, is out of the scope of this article. We rather focus on the configuration *semantics* of $VM' = VM \wedge cst$

4 Evaluation

The main question we aim to address is whether our approach can accurately classify configurations as *acceptable* (resp. *non-acceptable*) w.r.t any performance objective and with a reasonably small training set.

On the one hand, our learning-based technique can introduce errors in the following cases:

- a configuration in VM' is valid (and thus classified as acceptable) despite being non-acceptable – the system is under-constrained and unsafe;
- a configuration in VM' is not valid (and thus classified as non-acceptable) despite being acceptable – the system is over-constrained and lacks flexibility.

On the other hand, we expect to observe such cases:

- a configuration in VM' is valid and correctly classified as acceptable;
- a configuration in VM' is non-valid and correctly classified as non-acceptable.

Errors or successes in the classification can indeed have severe consequences on the configurability of the resulting specialized system. Our specialization method can over-constrain the system – very few configurations are reachable – and the users' *flexibility* is undermined. There is also a risk to under-constrain the system (*e.g.,* very few constraints have been synthesized), thus compromising its *safety*.

We can quantify the ratio of correct classifications, *i.e.,* the *accuracy*. However it is well-known that accuracy alone can have limited interest since it hinders important phenomena (see hereafter for more details). Intuitively, in our specific problem, we also need to measure the *precision* and *recall* of our classification. It can give better insights on our ability to (1) discard non-acceptable configurations and (2) to keep acceptable configurations. Furthermore, we need to quantify how safe and flexible are the resulting specialized systems. Finally, we also compare our learning-based specialization method with an approach without learning and specialization.

To summarize, our research questions are as follows:

- **RQ1** *What is the accuracy of our specialization method for classifying configurations?*
- **RQ2** *What is the precision and recall of our specialization method for classifying configurations?*
- **RQ3** *How safe and flexible are specialized configurable systems when applying our method?*
- **RQ4** *How effective is our learning technique compared to a non-learning technique?*

For each research question, we study the influence of different performance objectives and sizes of training sets on a set of real-world variability-intensive systems.

4.1 Subject systems and configuration performances

We performed our experiments on a publicly-available dataset used in previous works [1], [3], [6], [26]. The dataset includes sets of configurations of real-world configurable systems that have been executed and measured over different performance metrics. As shown in Table 1, configurable systems come from various domains and are implemented

in different languages (C, C++ and Java). The number of options varies from as little as ten, up to about sixty; options can take Boolean and numerical values. The two last columns show, first, the total number of different possible configurations existing for the system and, last, the number of configurations that have been generated and measured in previous works [1], [3], [6], [26] ("All" meaning all different configurations have been measured).

It should be noted that most benchmarks measure a single performance value, except for x264 where benchmarks evaluate 7 performance metrics.

System	Domain	Lang.	Features	$\#[\![VM]\!]$	Meas.
Apache	Web Server	C	9/0	192	All
BerkeleyC	Database	C	18/0	2560	All
BerkeleyJ	Database	Java	26/0	400	181
LLVM	Compiler	C++	11/0	1024	All
SQLite	Database	C	39/0	10^6	4553
Dune	Solver	C++	8/3	2304	All
HIPA^{cc}	Image Proc.	C++	31/2	13485	All
HSMGP	Solver	n/a	11/3	3456	All
JavaGC	Runtime Env.	C++	12/23	10^{31}	166k
x264 (Energy)	Codec	C	8/12	10^{27}	69k
x264 (PSNR)	Codec	C	8/12	10^{27}	69k
x264 (SSIM)	Codec	C	8/12	10^{27}	69k
x264 (Speed)	Codec	C	8/12	10^{27}	69k
x264 (Size)	Codec	C	8/12	10^{27}	69k
x264 (Time)	Codec	C	8/12	10^{27}	69k
x264 (Watt)	Codec	C	8/12	10^{27}	69k

TABLE 1: *Features*: number of boolean features / number of numerical features; $\#[\![VM]\!]$: number of valid configurations; *Meas.*: number of configurations that have been measured.

4.2 Experimental setup

The independent variables are the subject systems[5], the size of the input sample, and the percentage of acceptable configurations (as determined by a performance objective). The classification metrics (accuracy, precision, recall, specificity, *etc.*) are measured as the dependent variables (see next section and Figure 6).

To evaluate our method, we need two sets of configurations: the training sample and the test sample. For each subject system, we have randomly selected a certain number of configurations from the ones available in the public dataset: our ML method operates over a *training sample* to learn a separating function. All remaining configurations form the *test sample*, *i.e.*, it is the whole configuration set minus the training set. They are used to verify whether the classification of our ML is correct. For instance, the public dataset contains 69,000 different configurations for x264. If we used 500 of them to learn constraints, then we evaluate the classification quality of ML on the remaining 68,500 configurations.

For each configurable system, we make vary the sample size one by one, from 1 to the total number of configurations. We make vary the objective value between the lower and upper bounds of the performance measured in each dataset.

We notice that threshold values have direct influence on the percentage of acceptable configurations. For instance, considering Figure 2, an execution time (speed) threshold less than 100s leads to only 21.4% of acceptable configurations. We thus vary the objective value from 0% to 100% of acceptable configuration with steps of 5%.

To reduce the fluctuations of the dependent variables caused by random generation, we performed ten repetitions for each combination of the independent variables. That is, for each subject system, we repeated ten times generating a random sample of a certain size and subsequently measured the dependent variables after applying our approach to the sample. We took only the average of these measurements for analysis.

4.3 Classification measurements

In our case, positive class (also called class-1) corresponds to non-acceptable configurations that we want to discard. The negative class (also called class-0) corresponds to acceptable configurations that we want to keep. There are four possible situations represented in a confusion matrix (see Figure 6):

- True Positives (TP) and True Negatives (TN) are on the main diagonal and represent the set of correctly classified configurations;
- False Positive (FP) and False Negatives (FN), on the contrary, represent classification errors

For obtaining TP, TN, FP, and FN, we confront the classifications made by ML in the training set with the oracle decisions available in the testing set. Based on a confusion matrix, we can compute several classification metrics:

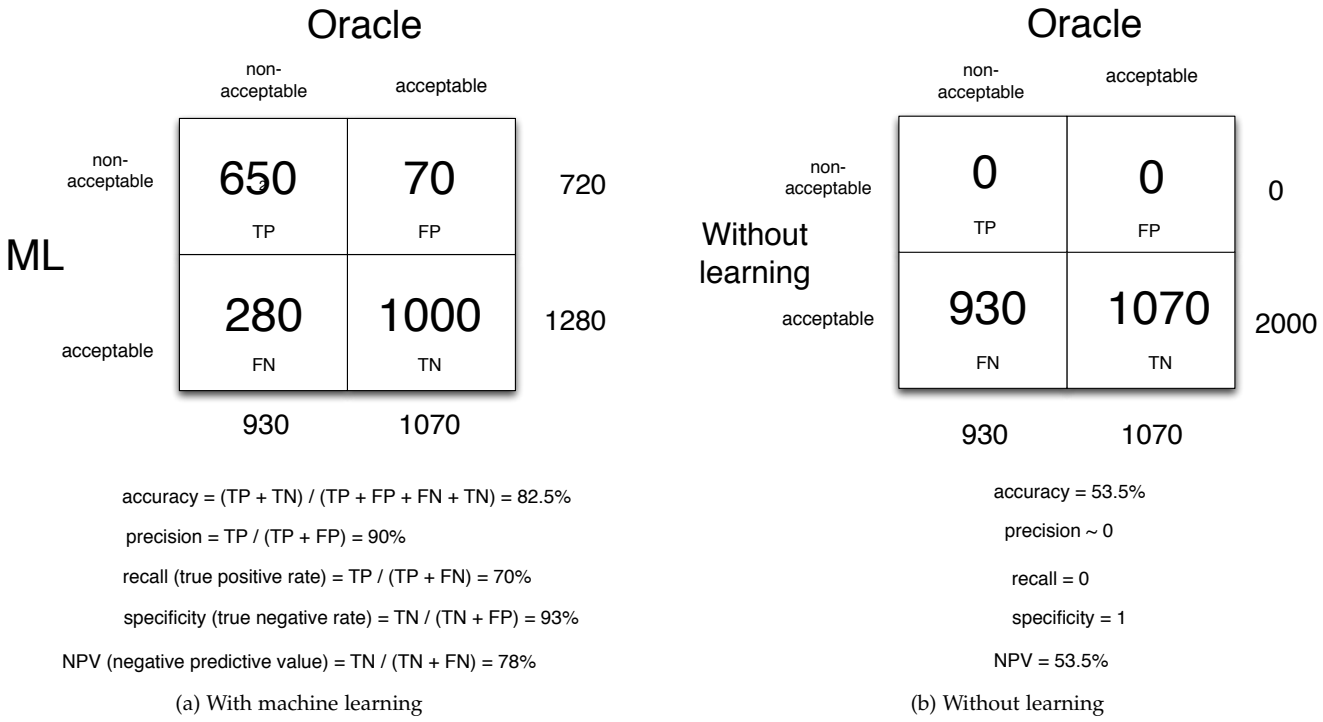
- **Accuracy** measures the portion of configurations accurately classified as acceptable and non-acceptable. It gives a global *classification efficiency* of the learning phase.
- **Precision** measures our ability to correctly classify non-acceptable configurations. A low precision may be problematic since numerous non-acceptable configurations are actually acceptable: Users may lose some *flexibility*.
- **Recall** (also called true positive rate) measures our ability to comprehensively identify non-acceptable configurations. A low recall may be problematic as well, since numerous non-acceptable configurations have not been classified as such: Users may still choose *unsafe* configurations.
- **Specificity** (also called true negative rate) quantifies how *flexible* is the resulting specialized system.
- **NPV** (also called negative predictive value) quantifies how *safe* is the resulting specialized system.

Each metric gives a complementary perspective to the qualities of our learning-based method.

Example. We consider the confusion matrix of Figure 6a to illustrate the metrics and their relations to safety and flexibility. In total, 2000 configurations are tested. Out of 2000, the machine learning succeeds to classify $650+1000 = 1650$ configurations. Hence the accuracy is 88.5%.

The first classification errors concern the false negatives (FP): 70 configurations are classified as non-acceptable despite being acceptable. The precision is high ($650/(650 + 70) = 90\%$). The other classification errors concern the false negatives (FN): 280 configurations are classified as

5. It should be noted that x264 has been evaluated 7 times since 7 performance measurements have been gathered



(a) With machine learning

(b) Without learning

Fig. 6: Confusion matrix and classification metrics: with machine learning *vs* without learning (example)

acceptable despite being non-acceptable. We can notice that the recall $(650/(650+280) = 70\%)$ is inferior to the accuracy (88.5%). It shows that the accuracy metric hinders some phenomenon and has some limitations (more details are given in the next research questions).

Again in the example of Figure 6a, the oracle states that 1070 configurations out of 2000 should have been kept. the machine learning correctly considers 1000 configurations (true negatives, TN) as acceptable; we miss to only retain 70 configurations. The specificity and so the flexibility are high $1000/1070 = 93\%$. Finally, the machine learning retains $280 + 1000 = 1280$ acceptable configurations in total. It is superior to the 1070 configurations of the ground truth. That is, there will be some unsafe configurations in VM' among the 1280. As we know that 1000 configurations are truly acceptable, we can conclude that $1000/1280 = 78\%$ of configurations will be safe in the resulting specialized system. It corresponds to *NPV*.

Comparison. An interesting perspective is to compare classification metrics of a learning approach with a non-learning approach. When there is no learning, all configurations are considered as acceptable; it corresponds to the situation in which no specialization of the configurable system is made. The number of false positive (FP) and true positive (TP) is 0, *i.e.,* opportunities to remove unsafe configurations are missed. The precision and recall always equal to 0. In the example of Figure 6b, the accuracy is 53.5%. It is less than 82.5% (the score of machine learning). The *NPV* also equals to 53.5%. The consequence is that the configurable system is quite unsafe. Only 53.5% of configurations (1070 out of 2000) will be truly consistent with the performance objective. In comparison, the machine learning achieves a 78% score and is much safer. Finally, the *specificity* of a non-learning approach is 100% by construction. In contrast, the machine learning score is 93% and is inferior in terms of flexibility.

Overall, in the example of Figure 6, the machine learning synthesizes much safer systems than a non-learning approach (78% vs 53.5%). We also consider it achieves a good tradeoff between safety and flexibility (specificity: 93% vs 100%).

The confusion matrix of Figure 6 is just an illustrative example. Based on our evaluation framework, we can now address the research questions using real configurable systems, sample trainings, and performance objectives.

4.4 Verifiability and presentation of results

For the sake of reproducibility of the experiments we provide:

- a website (http://learningconstraints.github.io/) with all heatmaps, plots, and figures.
- a git repository containing all data and the scripts used to: i) execute the procedures over different configurable systems; ii) generate all the different graphs within this paper as well as the rest shown in the website
- details on how we have collected and prepared data.

We recommend the reader to visualize heatmaps in color (*e.g.,* with a PDF viewer).

4.5 (RQ1) What is the accuracy of our specialization method for classifying configurations?

Accuracy (see Section 4.3 and Figure 6), a standard metric for classification problems, measures the fraction of

configurations accurately classified as acceptable and non-acceptable.

In Figure 7a, we depict a heatmap of x264 for showing its accuracy score w.r.t sampling size and objectives values (the darker, the higher accuracy is). For instance, with a percentage of acceptable configurations between 90 and 100 (Y-axis), the accuracy is above 0.9 whatever the sampling size is (X-axis). It means that for an objective threshold value that does not restrict too much the configuration space, the accuracy is quite high.

All other heatmaps are available in the Appendix A.1 and Figure 11. Based on their analysis, we can report on the following insights:

- the overall accuracy is quite high and is often above 90%. We can reach an accuracy of at least 80% for every system.
- there is no need to use a large training sample to reach such a high accuracy. In general, only dozens of configurations in the training set are needed for all systems.
- accuracy increases when the sampling size increases, following a logarithm curve. Hence the method can be used progressively.
- the performance objective for which accuracy is the highest (*e.g.,* greater than 0.9) is specific to systems. There are some sweet zones, but *e.g.,* an equal separation of acceptable and non-acceptable configurations does not equate to accuracy improvement.

Limits of accuracy. The high accuracy observed on all subjects suggests that our method is effective to classify configurations. However, a high accuracy does not necessarily mean that our specialization method is effective to identify non-acceptable configurations. For instance, in the example of Figure 6, accuracy is 82.5% (which is high) despite the fact the method is less effective to identify non-acceptable configurations we want to discard. Specifically, the recall is lower (70%) – 650 non-acceptable configurations have been identified out of 930.

Looking at the heatmaps of accuracy, precision, and recall of x264 (see Figure 7a, Figure 7b, and Figure 7c), we can observe that high accuracy does not imply high precision or high recall. When confronting the heatmaps of accuracy (see Appendix A.1, Figure 11) with the heatmaps of precision and recall (see Appendix A.2, Appendix A.3, Figure 12, and Figure 13) for all other subjects, we can make the same observation. It is a theoretical statement (see formulae) and *empirical results confirm there is no relationship between accuracy, precision and recall*.

Hence accuracy does not show an important phenomenon. Some methods may be very conservative (*flexible* but *unsafe*), thus under-constraining the configurable system, and eventually obtains a high accuracy. There is another related phenomenon that accuracy is unable to show: a method may be very agressive (*safe* but *unflexible*), over-constraining the configurable system, and eventually also obtains a high accuracy. It thus motivates the next research questions and the use of other classification measures.

To sum up, high accuracy can be obtained with a relative small number of configurations and for numerous performance objectives. However, accuracy cannot be used for characterizing the flexibility and safety aspects of a learning-based specialization method. We need to consider other classification metrics.

4.6 (RQ2) What is the precision and recall of our specialization method for classifying configurations?

Precision and recall (see Section 4.3 and Figure 6), two standard metrics for classification problems, measure our ability to correctly classify non-acceptable configurations. A low recall may be problematic, since numerous non-acceptable configurations have not been classified as such: Users might still choose unsafe configurations. A low precision may be problematic as well since numerous non-acceptable configurations are actually acceptable: Users might lose some flexibility.

In Figure 7b and Figure 7c, we depict two heatmaps of x264. We can observe that precision and recall values are quite high (> 0.8) for a vast majority of objectives, even with a low sampling size. There are some zones for which precision and recall are lower, but still acceptable.

Based on the analysis of other heatmaps [27] (see Appendix A.2, Appendix A.3, Figure 12, and Figure 13), we gather the following insights:

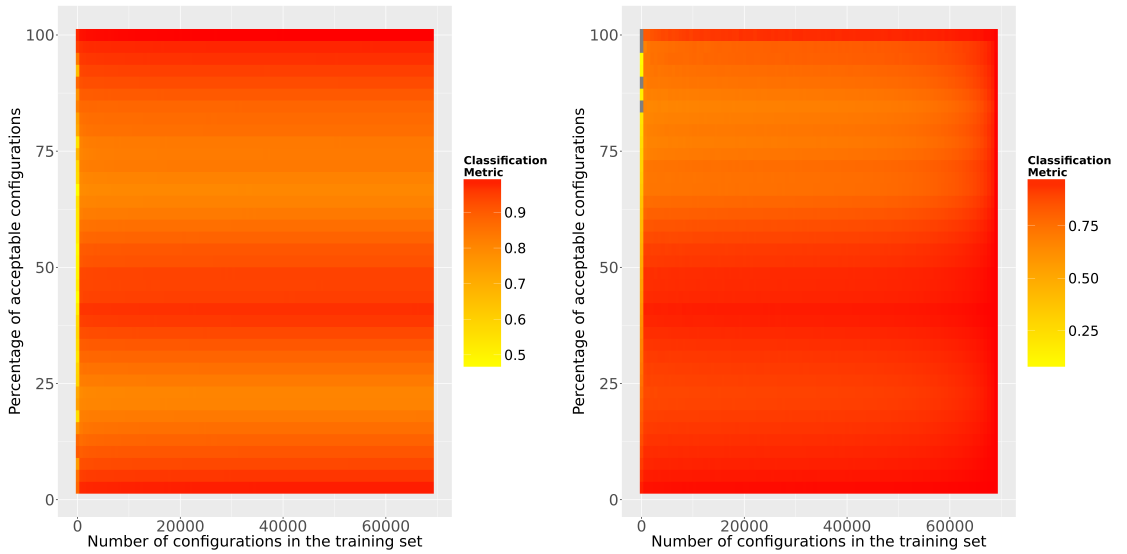
- precision *and* recall follow some of the trends we describe for accuracy: the overall precision and accuracy is high and often above 90%; only a dozen of configurations are needed for all systems for reaching 80%; precision and accuracy increase when the sampling size increases; the performance objective for which precision and recall are the highest is specific to systems.
- there is an interesting trend that we do not observe for accuracy. Precision (resp. recall) is the highest when the objective value leads to a predominance of non-acceptable configurations (*e.g.,* there is less than 25% acceptable configurations). It is the case in Figure 7b and for other subjects. On the contrary zones for which precision (resp. recall) are lower than the average precision (resp. recall) usually exhibit more than 75% acceptable configurations. Hence an hypothesis is that there is not enough non-acceptable configurations and the objective value is too "hard".

To sum up, high precision and recall can be obtained with a relative small number of configurations with the exception of some "hard" objective values for which the configurable system can be seen as too *permissive*.

4.7 (RQ3) How safe and flexible are specialized configurable systems when applying our method?

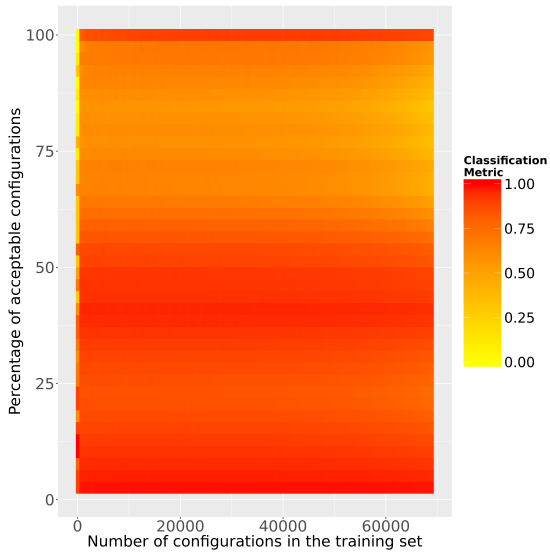
What we ignore at this step is how safe is our resulting specialized configurable system: how many valid configurations of VM' are truly acceptable? A low proportion and users will perform numerous configuration errors since the system is not safe enough. Conversely, how flexible is our system once specialized: how many configurations have been removed whereas they should still be reachable?

Accuracy, precision, and recall only measure the classification power of our method: They do not measure the portion of safe configurations and the flexibility users have

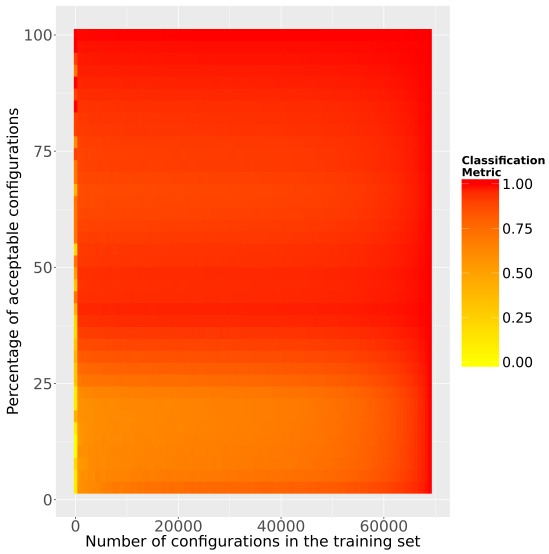


(a) Accuracy

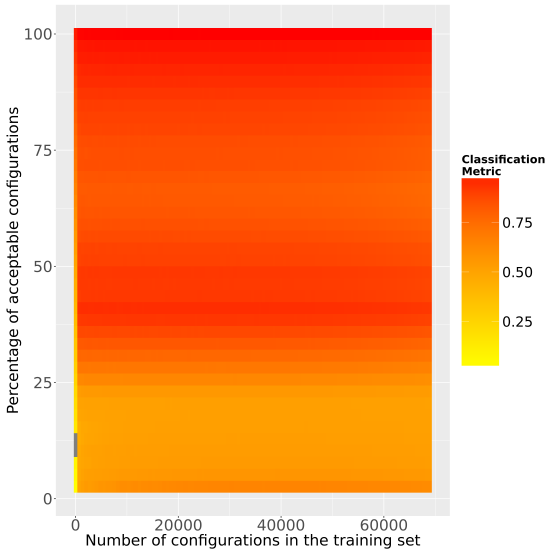
(b) Precision



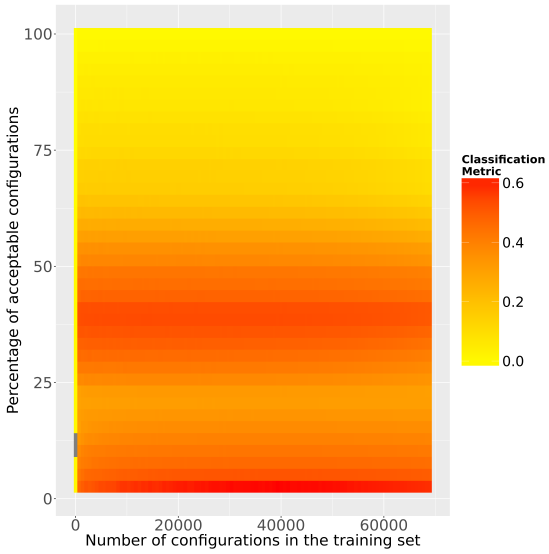
(c) Recall



(d) Specificity



(e) NPV



(f) negNPV (comparison with non-learning w.r.t. safety)

Fig. 7: Classification measures for x264 (execution speed) through heatmaps. The darker, the higher. X-axis: number of configurations in the training set. Y-axis: percentage of acceptable configurations (due to objective threshold value)

within the resulting system. Hence we rely on specificity and negative predictive value (NPV), also largely employed for classification problems. Specificity quantifies how flexible is the specialized system, while NPV quantifies how safe is the specialized system (see Section 4.3 and Figure 6).

In Figure 7d and Figure 7e, we can observe that specificity and NPV values of x264 are again quite high (> 0.8) for a vast majority of objectives, even with a low sampling size. some objective values (*i.e.,* leading to a percentage of acceptable configurations less than 25%) tend to decrease specificity and NPV values.

The analysis of all results (see Figure 8 and see Figure 9) shows that:

- the specificity and NPV values, despite some minor differences, follow the trends of RQ1 and RQ2. The overall value is high and often above 90%. Only a dozen of configurations are needed for all systems for reaching 80%. We considered it as reasonable since the ratio of measured configurations w.r.t the total number is low. Furthermore, specificity and NPV values progressively increase when the sampling size increases.
- specificity and NPV values are typically the lowest when the portion of acceptable configurations is low (*i.e.,* when there is less than 25% acceptable configurations). It is the opposite of precision and recall.
- as a consequence of the previous observation, our approach may be effective for identifying lots of non-acceptable configurations, but the specialized system still remains unsafe: There are simply too many configurations to discard.

To sum up, our approach can be effective to produce a safe and flexible system with a relative small number of configurations with the exception of some hard objective values for which the configurable system can be seen as *too restricted*. In such cases, the safety of the specialized system tends to decrease since there are simply lots of configurations to discard. Yet, even and especially for hard objectives, our specialization method significantly outperforms a non-learning technique (see next research question).

4.8 (RQ4) How effective is our technique compared to a non-learning technique?

Without learning and specialization, a configurable system exhibits non-acceptable configurations that violate the performance objective. If we consider the performance distribution of x264 (see Figure 1) and an objective: *"execution time < 145"*, the percentage of non-acceptable configurations is 63%. As a result, the *safety* of the configurable system can be (severely) undermined, with impacts on users or automated procedures in charge of choosing a configuration of x264 because in about 2/3 of the cases it would end up in an invalid configuration. With learning-based specialization, we can hope to produce safer configurable systems. On the other hand, a non-learning technique guarantees a maximum of *flexibility* since it retains all configurations. Comparatively, we aim to quantify how many configurations a learning-based technique wrongly discards. Overall, the question we want to address here is how safe and flexible are configurable systems (1) without specialization and (2) with specialization. We decompose RQ4 in two research questions:

- **(RQ4.1) How safe is our technique compared to a non-learning technique?**
- **(RQ4.2) How flexible is our technique compared to a non-learning technique?**

(RQ4.1) How safe is our technique compared to a non-learning technique?

In terms of metrics we have previously defined in our experimental settings, we can notice that a non-learning technique exhibits:

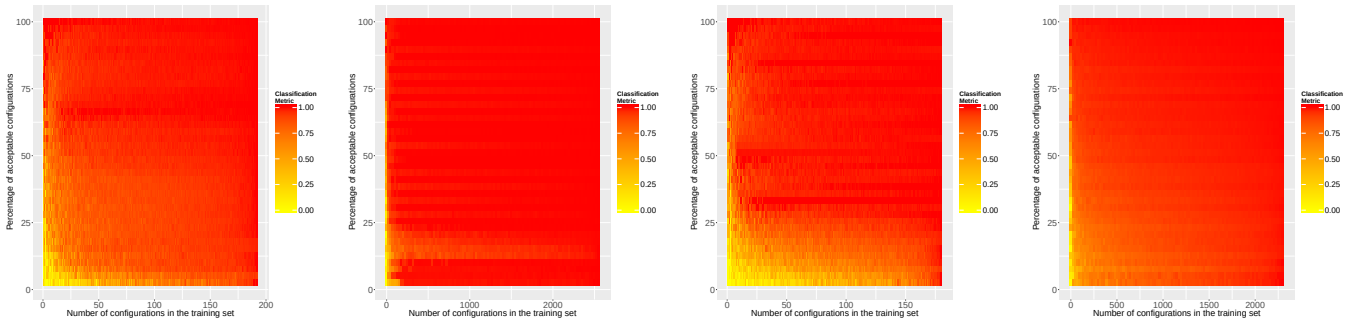
- a recall equals to 0 (the number of true positives equals to 0, due to the inability to classify configurations as non-acceptable);
- an NPV (for negative predictive value) equals to accuracy, suggesting that the NPV can be very low since a non-learning method does not classify configurations.

We remind that *NPV* measures how safe is the resulting configurable system. We denote $NPV_{non-learning}$ the negative predictive value of a non-learning technique. It is defined as: $NPV_{non-learning} = TN/(TN + FN)$. With a non-learning technique, TN equals to the number of acceptable configurations in the training set (since all configurations are classified as acceptable). Furthermore, $TN + FN$ is the total number of configurations in the training set. We can conclude that $NPV_{non-learning}$ corresponds to the *percentage of acceptable configurations* in the sample (the Y-axis in the heatmaps).

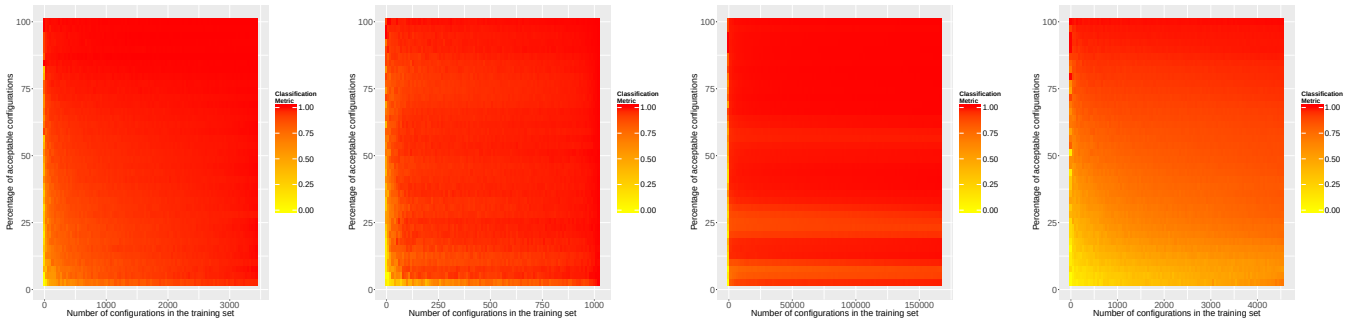
On the one hand, $NPV_{non-learning}$ (and so the safety of the system) is low (resp. high) when the percentage of acceptable configurations is low (resp. high). On the other hand, a low percentage of acceptable configurations challenges our learning technique. For further understanding, we compare the negative predictive values of our technique (denoted NPV) and of a non-learning technique (denoted $NPV_{non-learning}$). We compute a new metric, called $negNPV$, equals to $NPV - NPV_{non-learning}$. The higher $negNPV$, the safer is our specialized system comparatively to a non-specialized system.

Let us consider an example before generalizing to other configurable systems: Figure 7f gives the $negNPV$ heatmap of x264 (execution speed). We can observe that our learning-based specialization method is always superior. There is also a noticeable improvement (between 0.4 and 0.6) when the percentage of acceptable configurations is less than 50 since numerous non-acceptable configurations are successfully discarded. Another observation is that a high (resp. low) NPV (see Figure 7e) does not imply a high (resp.) $negNPV$. For example, when the percentage of acceptable configurations is less than 25, our specialization method succeeds to remove a lot of unsafe configurations despite relatively low NPV.

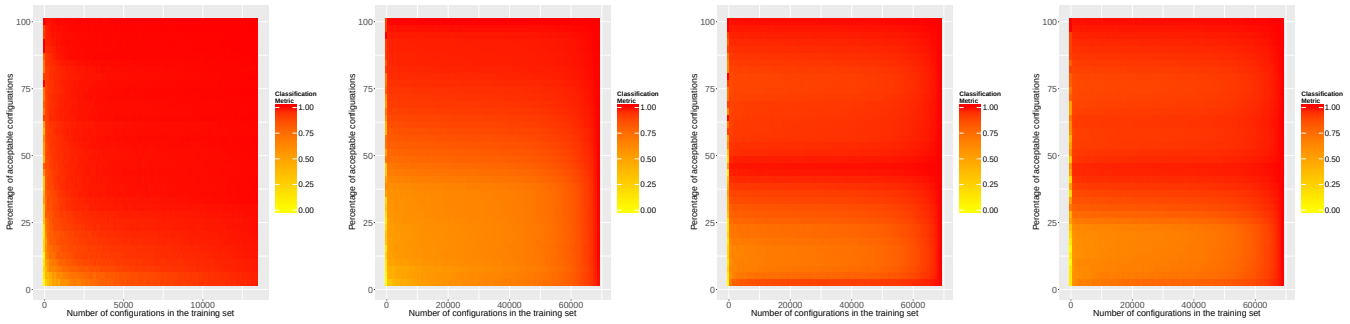
Figure 10 depicts the $negNPV$'s heatmaps of each configurable system. Empirical results confirm that our learning-based technique is safer in the vast majority of cases, even for a small training set. A first observation is



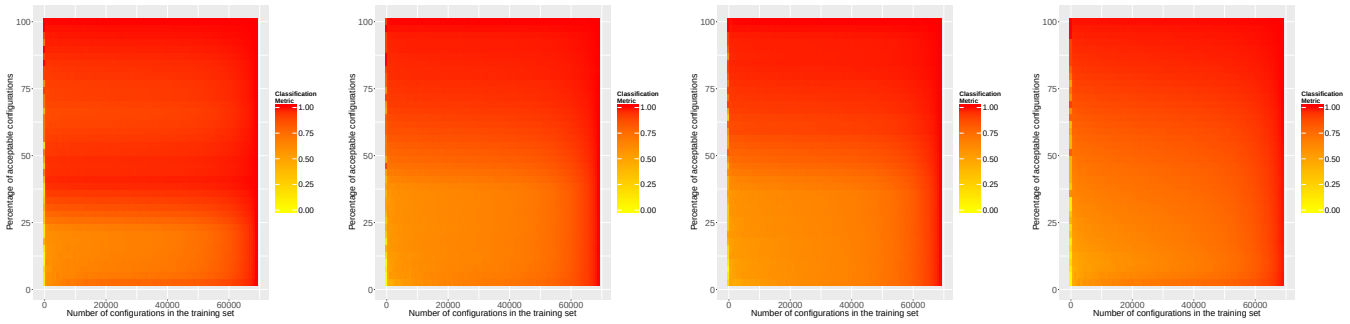
(a) specificity for (Apache) (b) specificity for (BerkeleyC) (c) specificity for (BerkeleyJ) (d) specificity for (Dune)



(e) specificity for (HSMGP) (f) specificity for (LLVM) (g) specificity for (JavaGC) (h) specificity for (SQLite)

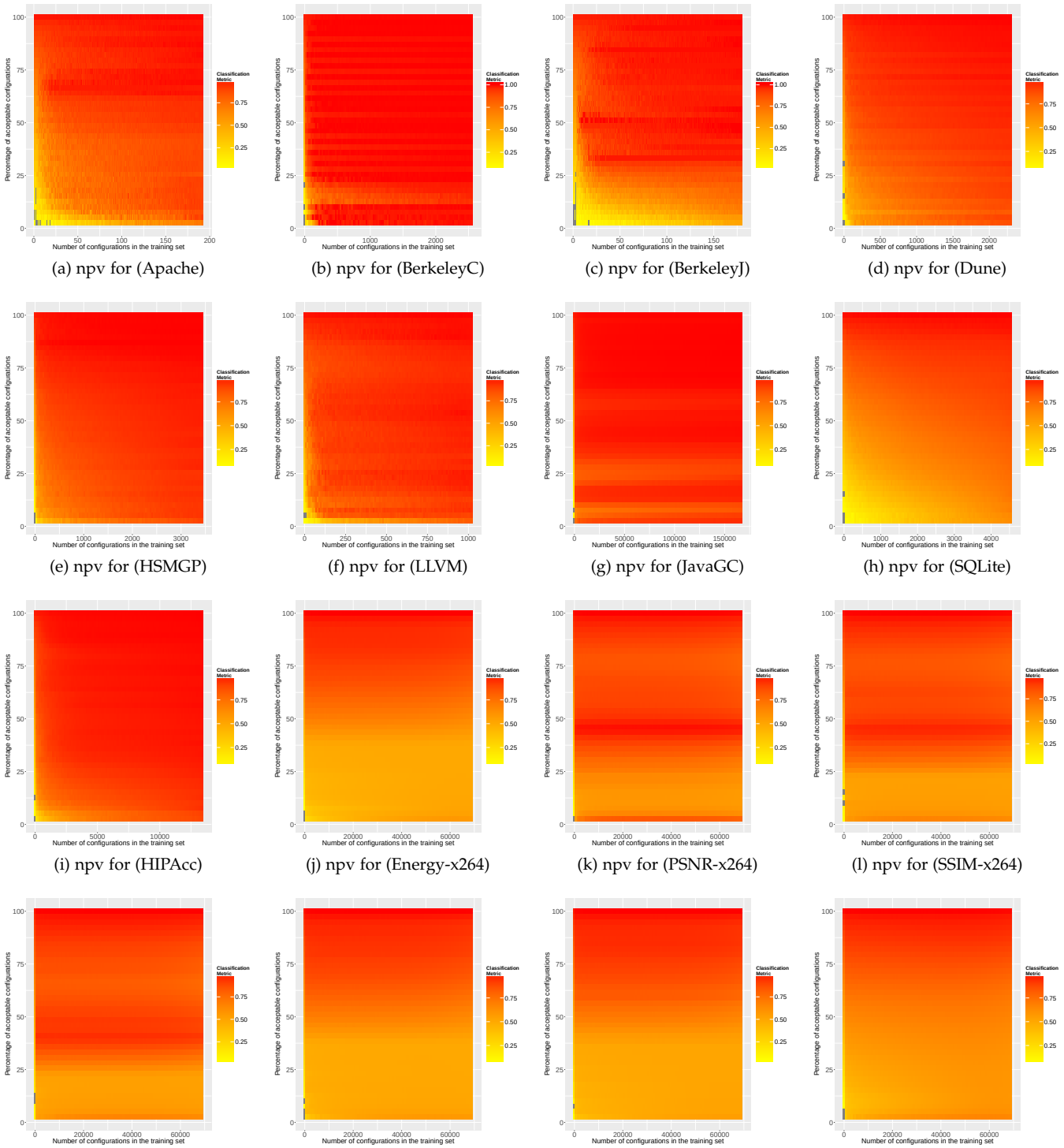


(i) specificity for (HIPAcc) (j) specificity for (Energy-x264) (k) specificity for (PSNR-x264) (l) specificity for (SSIM-x264)



(m) specificity for (Speed-x264) (n) specificity for (Size-x264) (o) specificity for (Time-x264) (p) specificity for (Watt-x264)

Fig. 8: specificity for all systems through heatmaps. The darker, the higher. X-axis: number of configurations in the training set. Y-axis: percentage of acceptable configurations (due to objective threshold value)



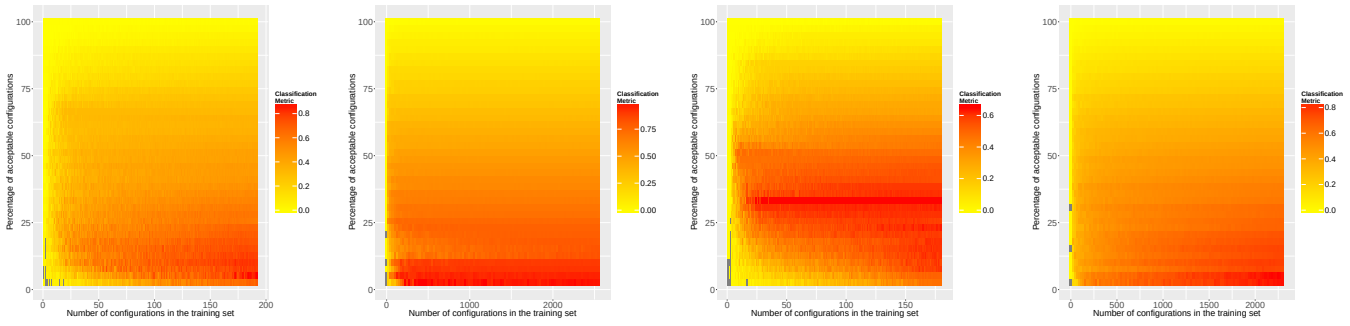
(a) npv for (Apache) (b) npv for (BerkeleyC) (c) npv for (BerkeleyJ) (d) npv for (Dune)

(e) npv for (HSMGP) (f) npv for (LLVM) (g) npv for (JavaGC) (h) npv for (SQLite)

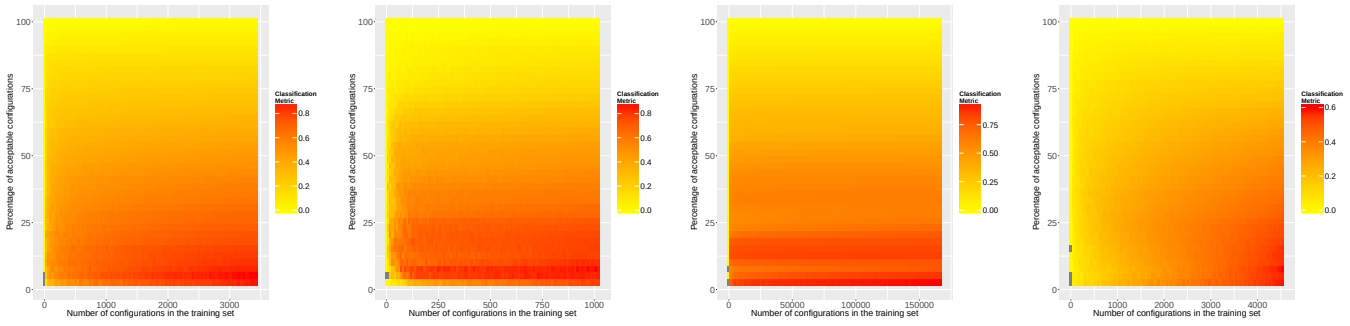
(i) npv for (HIPAcc) (j) npv for (Energy-x264) (k) npv for (PSNR-x264) (l) npv for (SSIM-x264)

(m) npv for (Speed-x264) (n) npv for (Size-x264) (o) npv for (Time-x264) (p) npv for (Watt-x264)

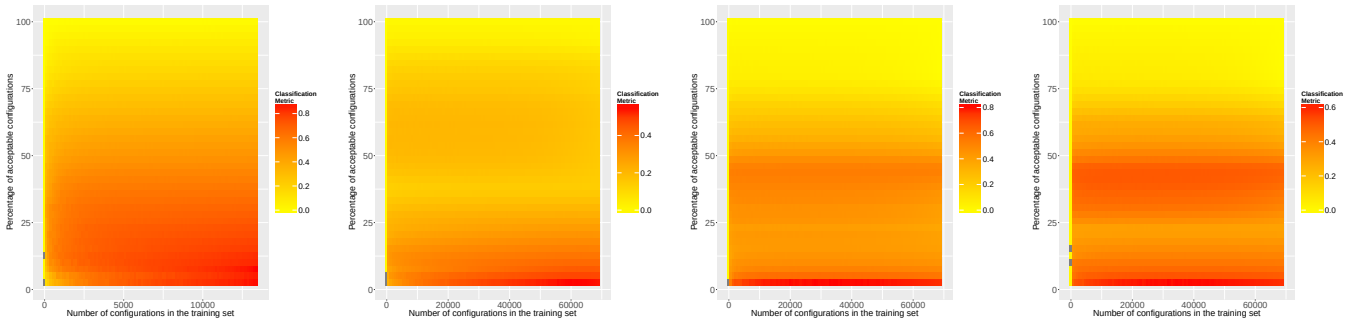
Fig. 9: npv for all systems through heatmaps. The darker, the higher. X-axis: number of configurations in the training set. Y-axis: percentage of acceptable configurations (due to objective threshold value)



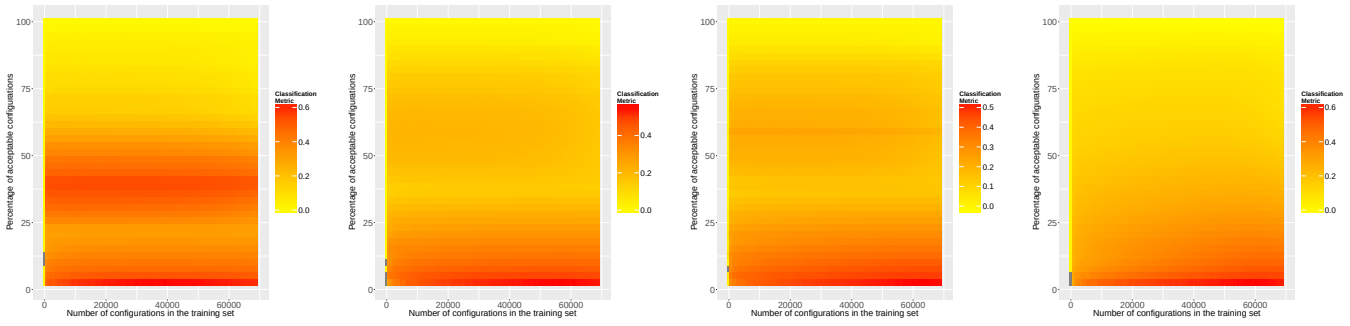
(a) negNPV for (Apache) (b) negNPV for (BerkeleyC) (c) negNPV for (BerkeleyJ) (d) negNPV for (Dune)



(e) negNPV for (HSMGP) (f) negNPV for (LLVM) (g) negNPV for (JavaGC) (h) negNPV for (SQLite)



(i) negNPV for (HIPAcc) (j) negNPV for (Energy-x264) (k) negNPV for (PSNR-x264) (l) negNPV for (SSIM-x264)



(m) negNPV for (Speed-x264) (n) negNPV for (Size-x264) (o) negNPV for (Time-x264) (p) negNPV for (Watt-x264)

Fig. 10: negNPV for all systems through heatmaps. The darker, the higher. X-axis: number of configurations in the training set. Y-axis: percentage of acceptable configurations (due to objective threshold value)

that a learning-based specialization always beats a non-learning technique ($negNPV$ varies from 0 to values higher than 0.8). A second observation is that the improvement is particularly paramount when the performance objective threshold value leads to a low percentage of acceptable configurations. In Figure 10, we can observe that when the percentage of acceptable configurations is less than 50, $negNPV$ tends to increase. In such cases, the number of configuration errors of a non-learning technique (as quantified by $NPV_{non-learning}$) is very high, since numerous configurations are actually non-acceptable. As a summary, our specialization method is particularly interesting for performance objective that calls to eliminate lots of non-acceptable configurations.

Another insight is that when $negNPV$ increases, NPV tends to decrease (see an example with Figure 7e and Figure 7f). Though a learning technique induces some classification errors (see NPV), it has the merit of eventually discarding many more configurations than a non-learning technique. On the one hand, it is challenging to properly classify configurations when the number of acceptable configurations is low. In such case, configurations classes are imbalanced and NPV is the lowest. On the other hand, a learning technique can capture lots of non-acceptable configurations since there are many of them in the configurable system. *For hard performance objectives and despite classification errors, a learning technique is much more effective to discard unsafe configurations than a non-learning technique.*

(RQ4.2) How flexible is our technique compared to a non-learning technique?

In terms of metrics we have previously defined in our experimental settings, we can notice that a non-learning technique has a $specificity$ always equals to 100%. The configurable system is highly flexible since all configurations are kept as acceptable. By construction, a learning-based approach cannot beat a non-learning approach w.r.t flexibility. In order to quantify this effect, we compute the difference between the specificity of our learning method (denoted $specificity$) and a non-learning method. The difference equals to $specificity - 1$.

All $specificity$ values of configurable systems are depicted in Figure 8. The differences remain on the same color scale and can be seamlessly visualized. Specifically, our learning method suffers from the comparison with a non-learning method when the $specificity$ is low. Hence the same observations made in Section 4.7 (see RQ3) apply. A major insight is that our learning method degrades flexibility when non-acceptable configurations are predominant (*i.e.,* less than 25%). *Considering the insights of **RQ4.1** and **RQ4.2**, we can conclude that our learning method increases safety to the expense of flexibility.*

(RQ4) To sum up, empirical results show that:

- our learning-based specialization method produces safer configurable systems for all subjects and performance objective thresholds;
- despite classification errors (*e.g.,* as quantified by NPV), a learning technique can be much more effective to discard unsafe configurations than a non-learning technique;
- safety differences between a learning and a non-learning method are more important when the percentage of non-acceptable configurations is high. In this case, our learning technique is particularly suited to increase safety and significantly outperforms a non-learning technique even with a small number of configurations in the sample;
- there exists some sweet zones for which we can achieve high flexibility and high safety. In general, our specialization method achieves a good trade-off between flexibility and safety.

5 Threats to validity

External validity. To study the generalization of our method, we have selected configurable systems in different domains (database, compiler, video compression, *etc.*), written in different languages, and that exhibit different configuration complexity. We have also considered different performance qualities. Furthermore we reused data published in papers for predicting and modeling performance, which is related to our topic. We are aware that our empirical results are not automatically transferable to all other configurable systems or performance qualities.

To increase the diversity of objectives, we used numerous values that cover the whole performance spectrum. We have considered the percentage of acceptable configurations as an independent variable of our experiment, since the effect of a performance objective is precisely to change the ratio of acceptable and non-acceptable configurations. Likewise we can vary performance threshold values (*e.g.,* $executiontime < 100$). We can also handle objectives in the following form: $50 < executiontime < 100$.

Internal validity. We used random generation to select our training set, while taking the rest of the population as testing samples. We argue that it could lead to a worst case scenario where little intelligence is put into the sampling phase. To mitigate the random selection, we executed ten times our experiments with the same settings (system, number of data in training set, and objective's value).

In order to only focus on independent variables, we did not try to optimize the parameters of the ML technique and we used the same ML settings in all experiments.

Our results are dependent on data that have been published before. Specially, we can notice that BerkeleyJ can lead to 400 possible configurations but only 181 were used. Similarly, for systems with more than 10^{20} possible configuration, less than 1% were used. The choice of other configurations could have led to different performance distributions that could be easier or harder to learn.

Construct validity. We adressed a binary classification problem with unbalanded classes: We carefully followed recommendations when choosing suitable evaluation metrics in such a context [42]. Specifically we reused several well-known classification measures and we discussed some of their limitations (*e.g.,* accuracy). Furthermore we showed how our classification metrics relate to the specificities of our specialization problem (*e.g.,* flexibility and safety of a configurable system).

6 Related Work

Learning and performance. The prediction of the performance of individual configurations is subject to intensive research [1]–[5]. These approaches aim at *predicting* the performance of a given configuration. For example, users specify some parameters of x264 and a predictor can give the speed value or energy consumption – *without* executing and measuring x264. Such works do not target our specialization problem: They do not eliminate non-acceptable configurations and do not retrofit constraints into the variability model. As a result, users can still spend significant effort and time in trying non-acceptable configurations. Similarly, automated procedures, usually based on solving techniques, cannot exploit constraints as part of their reasoning.

Besides distinct applicability scope, the problems of performance prediction and specialization are fundamentally different. We address a binary *classification* problem while predicting a performance value is a *regression* problem. In terms of solution, we rely on classification techniques and not on regression ones. Dedicated and optimized machine learning algorithms have been independently designed for either classification or regression, with very different mathematical formulations of the two problems (see Section 3). In terms of evaluation, we also need to consider the specificities of our classification problem and we have developed dedicated metrics.

Siegmund *et al.* [6] combined machine-learning and sampling heuristics to compute the individual influences of configuration options and their interactions. So-called performance-influence models are meant to ease understanding, debugging, and optimization of configurable systems. As previous works mentioned, performance-influence models do not address the performance specialization problem and do not retrofit constraints into a variability model. Incidentally, our specialization method has the potential to ease understanding and debugging of a configurable system. Extracted constraints can serve to verify whether options values or their interactions do make sense w.r.t a performance threshold. For instance, developers can identify potential weaknesses or validate some strengths of software variability *when* the execution is supposed to be very fast – perhaps some configuration options are deactivated whereas they should be activated in such specific cases. In the example of Section 2, maintainers can verify some parameter values of x264 for *fast* execution times. At this step of the research, however, it is unclear how the synthesized constraints are effective for understanding and debugging configurable systems. We plan to conduct comprehensive user studies in future work.

Mining constraints and configuration options. Numerous works address the problem of automatically building a configuration model (*e.g.,* Boolean feature model) [43]–[50]. Given a Boolean formula or a configuration matrix, algorithms have been notably proposed to synthesize constraints among options. Such techniques usually assume a comprehensive knowledge of what constitute valid and non-valid configurations. They cannot be reused in our context since we only have a *sample* of configurations out of which we need to learn and predict validity of the whole population.

In prior work, we presented a learning-based process for synthesizing constraints [24]. We showed that our method was effective to reinforce a video generator developed in the industry. The goal was mostly to avoid *faulty* configurations and correct the video generator; the oracle was simply a yes/no procedure. In this article, we have considered performance qualities. The oracle is parameterized with an objective value that users of our specialization method can control. Hence our proposal can be seen as a generalization of the idea originally proposed in [24]. Moreover our empirical results cover much more systems and classification metrics while taking training sizes and objective values into account. The effect of performance objective thresholds has been carefully considered and simply does not exist in our previous work.

Techniques have been proposed to mine configurations or constraints out of source code, build systems, or textual descriptions [51]–[59]. For instance, Yi *et al.* [54] applied support vector machine and genetic techniques to mine binary constraints (requires and excludes) from Wikipedia. Nadi *et al.* [60] combines different techniques to extract configuration constraints in the Linux kernel. In our case we aim to discover constraints that are *specific* to a given performance objective. A static extraction of such specialized constraints within artefacts is thus unlikely.

Perrouin *et al.* [61] proposed a search-based process to test and fix feature models. The idea is to validate or eliminate some configurations, based on some checking (*e.g.,* the configuration of the Linux kernel does compile). Our strategy for adding constraints significantly differs. Instead of only removing the configurations that have been tested, we rely on learning to also consider configurations that have not been tested. Furthermore we considered performance qualities (*e.g.,* execution time) as part of the testing process.

Testing and verification. Numerous techniques have been developed to verify product lines and configurable systems either based on testing, type checking, model checking, or theorem proving [62], [62]–[70].

SPLat is a dynamic analysis technique for pruning irrelevant configurations [70]. The goal is to reduce the combinatorial number of variants (*e.g.,* Java programs) to examine. SPLif aims to detect bugs of software product lines with incomplete feature models [66]. It helps to prioritize failing tests and configurations. Our method can enforce feature models with constraints and can benefit to SPLif.

Sampling. Several techniques can be used to sample the configurations to test [3], [29]–[35], [71]. Empirical studies show that sampling strategies can influence the number of detected faults or the precision of performance prediction [3], [35]. For instance, increasing the size of sample sets can have positive effects on the fault-detection capability, but it also has a cost [35]. As discussed, the problem of inferring constraints is a tradeoff between the costs of testing and the ability to learn constraints from an oracle and configurations. In our evaluation, we use a random strategy for picking values (see Section 3). More advanced strategies or techniques to sample or prioritize configurations can benefit to our method. A research direction is to conduct further empirical studies to measure their cost-effectiveness.

Random testing. Black-box or white-box software testing techniques have been developed [72]. The idea is to

generate random inputs and resulting outputs are observed for detecting faults in single programs. Our goal is to make safe a configurable system and is thus different. Whitebox fuzzing, such as SAGE, consists of executing the program and gathering constraints on inputs using search techniques and heuristics [73]. The role of constraints thus differs. In our case, constraints are reinjected into a variability model whereas SAGE exploits constraints to guide the test generation.

Configuration process. The idea of specializing a product line has a long tradition [16], [16]–[20]. At different steps or stages, users can partially configure a system. A manual activity is usually performed for elaborating constraints or (de-)activating options, both aiming to specialize the configuration space. Our work aims at *automating* such a specialization process with the definition of an objective and the use of automated procedures (oracles).

Numerous works aim to tune configuration parameters for improving the performance of a system (*e.g.,* an algorithm) or optimizing multiple quality criterion [7], [74]–[76]. Our specialization method does not aim to select, tune, or optimize *one* configuration. We rather aim to offer a *set* of satisfying configurations. We thus pursue a different goal: The specialized system remains (highly) configurable but constrained to satisfy a given performance objective.

Numerous techniques, based on solver or search-based heuristics, have been developed to assist users in configuring a system [8]–[15]. They operate over a variability model. A strength of our method is that such reasoning techniques can be reused once the specialization has been performed, since we compute a new variability model.

7 Conclusion

With billions of possible configurations to consider, software engineers, integrators, or simply end-users typically perform a semi-blinded endeavor based on numerous trials and errors. In response, we introduced the idea of automatically specializing a configurable system: all option values (and combinations thereof) presented to users should henceforth satisfy a performance objective (*e.g.,* an execution time below a certain threshold). A specialized configurable system can save significant time and effort since the options values are pre-configured while the configuration space is safer since non-acceptable configurations are discarded. Importantly, a specialized system remains (highly) configurable but constrained to satisfy the performance objective for any of the remaining configurations.

Since measuring the performance of a system in every possible configurations is not a viable solution, we relied on machine learning techniques to predict the suitability of unmeasured configurations. We then retrofitted constraints into a variability model to enforce the configuration space composed of Boolean and numerical options . We have developed a method based on sampling, testing, and learning to narrow down the configuration space and exclude configurations that do not satisfy a certain performance quality.

Empirical results on 16 publicly available subjects and performance qualities showed that our method can effectively classify configurations with a relative small training set and for numerous objective thresholds. The synthesis of constraints leads to safe and still flexible specialized configurable systems. Our learning-based specialization method produces safer configurable systems for all subjects and performance objectives than a non-learning based approach. It is particularly effective for hard performance objectives, *i.e.*, objectives for which less than 25% of configurations are still acceptable. Without our specialization method, users or automated procedures will most likely perform numerous configuration errors, with negative impacts on engineering effort/time, configuration usability, or simply software quality.

Future work

Our learning-based method has the potential to specialize numerous configurable systems, from software product lines to self-adaptive systems. Different research directions can be considered in the future.

Supporting the specializer. Our specialization method is fairly simple and takes as input a performance objective. We aim to further explore how to support a specializer in choosing a performance objective threshold. Considering the execution time, a specializer can choose as well "less than 5 seconds", "more than 10 seconds", or "less than 0.1 second". Depending on the configurable system, such threshold values have certainly a specific meaning: the system can be considered as "fast", "very slow" or "very fast". In general, it seems useful to help a specializer visualizing the performance distributions such that she can understand the different trends (extreme values, mean, *etc.*). The challenge is to produce a representative visualization by using a small configuration sample. Another related challenge is to provide to a specializer the means to validate and gain confidence in the specialization she has just applied. Our results indeed show that both sampling size and performance objectives can influence the safety or flexibility of the specialized system. A possible validation strategy is to test the training set and use the metrics of our evaluation. Testing (in the machine learning sense) typically requires to measure additional data (*i.e.*, configurations) and thus has a cost. Based on the test results, a specializer can change the objective threshold or increase the size of the training set. Overall, the extent to which safety and flexibility can be degraded is a socio-technical decision. We plan to conduct field studies in future work.

Readability of constraints. We used decision trees for classifying configurations. An interesting property of decision trees is their simplicity and expressiveness. It first allows one to inject constraints among Boolean and numerical options into a variability model. Another advantage worth investigating in future work is that humans can potentially read and understand such constraints. Experts can qualitatively validate the synthesized constraints. It can be useful to validate the specialization process itself (see previous point). It can also ease understanding and debugging of a configurable system, since *e.g.,* our specialization method identifies suspicious and unexpected constraints between options.

Machine learning. In this article, we used fairly simple machine learning techniques that already achieve good

results and act as a state-of-the-art baseline. Other classification algorithms (*e.g.*, bagging, random forest, boosted trees), sampling (*e.g.*, over-sampling, under-sampling, or T-wise sampling), or dimensionality reduction techniques can be considered in the future [35], [36], [77]. We plan to reuse our evaluation settings (subjects, metrics, *etc.*) to compare the classification power of other approaches. For the sake of reproducibility of the experiments, we provide data, source code and instructions:

http://learningconstraints.github.io/

Applicability. Given performance objectives, new product lines or pre-configured systems (x264, Apache, SQLite, *etc.*) can be built for targeting specific usages, customers, deployment scenarios, or hardware settings. For instance, we can specialize x264 for achieving a low energy consumption such that software integrators can embed x264 in small devices. We can consider other performance properties (execution time, footprint, *etc.*) as well and specialize x264 for achieving fast computations or delivering high-quality results. Users can can still fine-tune their configurations w.r.t their functional constraints. An immediate research and practical direction is to conduct further empirical studies on different configurable systems and performance qualities. An open challenge is, for example: can we specialize the Linux kernel (10000+ configuration options) such that all of the remaining configurations boot in less than 5 seconds?

REFERENCES

[1] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski, "Variability-aware performance prediction: A statistical learning approach," in *ASE*, 2013.

[2] N. Siegmund, M. RosenmüLler, C. KäStner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov, "Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption," *Inf. Softw. Technol.*, 2013.

[3] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki, "Cost-efficient sampling for performance prediction of configurable systems (t)," in *ASE'15*, 2015.

[4] Y. Zhang, J. Guo, E. Blais, and K. Czarnecki, "Performance prediction of configurable software systems by fourier learning (T)," in *ASE'15*, 2015.

[5] P. Valov, J. Guo, and K. Czarnecki, "Empirical comparison of regression methods for variability-aware performance prediction," in *SPLC'15*, 2015.

[6] N. Siegmund, A. Grebhahn, C. Kästner, and S. Apel, "Performance-influence models for highly configurable systems," in *ESEC/FSE'15*, 2015.

[7] A. Sayyad, T. Menzies, and H. Ammar, "On the value of user preferences in search-based software engineering: A case study in software product lines," in *Software Engineering (ICSE), 2013 35th International Conference on*, May 2013, pp. 492–501.

[8] C. Henard, M. Papadakis, M. Harman, and Y. L. Traon, "Combining multi-objective search and constraint solving for configuring large software product lines," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, 2015, pp. 517–528. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2015.69

[9] Y. Yu, J. C. S. do Prado Leite, A. Lapouchnian, and J. Mylopoulos, "Configuring features with stakeholder goals," in *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, 2008, pp. 645–649.

[10] M. Mendonca, A. Wasowski, and K. Czarnecki, "SAT-based analysis of feature models is easy," in *SPLC'09*. IEEE, 2009, pp. 231–240.

[11] M. Mendonca and D. Cowan, "Decision-making coordination and efficient reasoning techniques for feature-based configuration," *Science of Computer Programming*, vol. 75, no. 5, pp. 311 – 332, 2010.

[12] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki, "Generating range fixes for software configuration," in *34th International Conference on Software Engineering*, 06/2012 2012.

[13] R. Pohl, V. Stricker, and K. Pohl, "Measuring the structural complexity of feature models," in *ASE'13*, 2013.

[14] A. Nöhrer and A. Egyed, "C2O configurator: a tool for guided decision-making," *Autom. Softw. Eng.*, vol. 20, no. 2, pp. 265–296, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10515-012-0117-4

[15] ——, "Optimizing user guidance during decision-making," in *Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011*, 2011, pp. 25–34.

[16] K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged configuration through specialization and multilevel configuration of feature models," *Software Process: Improvement and Practice*, vol. 10, no. 2, pp. 143–169, 2005.

[17] ——, "Staged configuration using feature models," in *Software Product Lines*, 2004, pp. 266–283.

[18] K. Czarnecki, M. Antkiewicz, and C. H. P. Kim, "Multi-level customization in application engineering," *Commun. ACM*, vol. 49, no. 12, pp. 60–65, 2006.

[19] E. Abbasi, A. Hubaux, and P. Heymans, "A toolset for feature-based configuration workflows," in *SPLC'11*. IEEE, 2011, pp. 65 –69.

[20] A. Classen, A. Hubaux, and P. Heymans, "A formal semantics for multi-level staged configuration," in *VaMoS*, 2009, pp. 51–60.

[21] B. Morin, O. Barais, G. Nain, and J.-M. Jézéquel, "Taming dynamically adaptive systems using models and aspects," in *ICSE'09*. IEEE, 2009, pp. 122–132.

[22] B. Morin, O. Barais, J. Jézéquel, F. Fleurey, and A. Solberg, "Models@ run.time to support dynamic adaptation," *IEEE Computer*, vol. 42, no. 10, pp. 44–51, 2009. [Online]. Available: http://dx.doi.org/10.1109/MC.2009.327

[23] S. O. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, "Dynamic software product lines," *IEEE Computer*, vol. 41, no. 4, pp. 93–95, 2008. [Online]. Available: http://dx.doi.org/10.1109/MC.2008.123

[24] P. Temple, J. A. Galindo Duarte, M. Acher, and J.-M. Jézéquel, "Using Machine Learning to Infer Constraints for Product Lines," in *Software Product Line Conference (SPLC)*, Beijing, China, Sep. 2016. [Online]. Available: https://hal.inria.fr/hal-01323446

[25] x264 Settings, "http://www.chaneru.com/Roku/HLS/X264_Settings.htm."

[26] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. S. Batory, M. Rosenmüller, and G. Saake, "Predicting performance via automated feature-interaction detection," in *ICSE*, M. Glinz, G. C. Murphy, and M. Pezzè, Eds. IEEE, 2012, pp. 167–177.

[27] Companion Website for verificability and reproducibility with data, source code, raw results, heatmaps, graphs, and figures, "http://learningconstraints.github.io/."

[28] T. Thüm, "Reasoning about feature model edits," Bachelor's thesis, University of Magdeburg, Germany, Jun. 2008.

[29] M. F. Johansen, O. y. Haugen, and F. Fleurey, "An algorithm for generating t-wise covering arrays from large feature models," *SPLC'12*, 2012.

[30] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon, "Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines," *IEEE Trans. Software Eng.*, 2014.

[31] B. Pérez Lamancha and M. Polo Usaola, "Testing Product Generation in Software Product Lines Using Pairwise for Features Coverage," in *Testing Software and Systems*, ser. Lecture Notes in Computer Science, A. P. A. S. J. C. Maldonado, Ed. Springer, 2010, vol. 6435, pp. 111–125. [Online]. Available: https://hal.inria.fr/hal-01055240

[32] M. B. Cohen, M. B. Dwyer, and J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach," *IEEE Trans. Softw. Eng.*, vol. 34, no. 5, pp. 633–650, Sep. 2008. [Online]. Available: http://dx.doi.org/10.1109/TSE.2008.50

[33] M. Al-Hajjaji, T. Thüm, J. Meinicke, M. Lochau, and G. Saake, "Similarity-based prioritization in software product-line testing," in *SPLC'14*, 2014.

[34] J. A. Galindo, M. Alférez, M. Acher, B. Baudry, and D. Benavides, "A variability-based testing approach for synthesizing video sequences," in *International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. ACM, 2014, pp. 293–303. [Online]. Available: http://doi.acm.org/10.1145/2610384.2610411

[35] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in *ICSE'16*, 2016.

[36] E. Alpaydin, *Introduction to Machine Learning*, 2nd ed. The MIT Press, 2010.

[37] W.-Y. Loh, "Classification and regression trees," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 1, no. 1, pp. 14–23, 2011.

[38] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[39] L. Breiman *et al.*, *Classification and Regression Trees*. New York: Chapman & Hall, 1984, new edition of [39]? [Online]. Available: http://www.crcpress.com/catalog/C4841.htm

[40] 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, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[41] J. Fürnkranz, *Decision Lists and Decision Trees*. Boston, MA: Springer US, 2010, pp. 261–262. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-30164-8_200

[42] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks," *Information Processing and Management*, vol. 45, no. 4, pp. 427 – 437, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0306457309000259

[43] K. Czarnecki, S. She, and A. Wasowski, "Sample spaces and feature models: There and back again," in *SPLC*, 2008.

[44] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "Reverse engineering feature models," in *ICSE'11*, 2011.

[45] E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed, "Reverse engineering feature models from programs' feature sets," in *WCRE'11*. IEEE, 2011, pp. 308–312.

[46] ——, "On extracting feature models from sets of valid feature combinations," in *FASE*, 2013.

[47] N. Andersen, K. Czarnecki, S. She, and A. Wasowski, "Efficient synthesis of feature models," in *Proceedings of SPLC'12*, 2012.

[48] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cleland-Huang, and P. Heymans, "Feature model extraction from large collections of informal product descriptions," in *ESEC/FSE*, 2013.

[49] G. Bécan, R. Behjati, A. Gotlieb, and M. Acher, "Synthesis of attributed feature models from product descriptions," in *SPLC'15*, jul 2015.

[50] G. Bécan, M. Acher, B. Baudry, and S. Ben Nasr, "Breathing ontological knowledge into feature model synthesis: an empirical study," *Empirical Software Engineering*, 2015.

[51] V. Alves, C. Schwanninger, L. Barbosa, A. Rashid, P. Sawyer, P. Rayson, C. Pohl, and A. Rummler, "An exploratory study of information retrieval techniques in domain analysis," in *SPLC'08*. IEEE, 2008, pp. 67–76.

[52] K. Chen, W. Zhang, H. Zhao, and H. Mei, "An approach to constructing feature models based on requirements clustering," in *RE'05*, 2005, pp. 31–40.

[53] N. Niu and S. M. Easterbrook, "Concept analysis for product line requirements," in *AOSD'09*, K. J. Sullivan, A. Moreira, C. Schwanninger, and J. Gray, Eds. ACM, 2009, pp. 137–148.

[54] L. Yi, W. Zhang, H. Zhao, Z. Jin, and H. Mei, "Mining binary constraints in the construction of feature models," in *RE'12*, 2012, pp. 141–150.

[55] N. Weston, R. Chitchyan, and A. Rashid, "A framework for constructing semantically composable feature models from natural language requirements," in *SPLC'09*. ACM, 2009, pp. 211–220.

[56] E. Vacchi, B. Combemale, W. Cazzola, and M. Acher, "Automating Variability Model Inference for Component-Based Language Implementations," in *18th International Software Product Line Conference (SPLC'14)*, 2014.

[57] A. Rabkin and R. H. Katz, "Static extraction of program configuration options," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, 2011, pp. 131–140. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985812

[58] S. Zhang and M. D. Ernst, "Automated diagnosis of software configuration errors," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, 2013, pp. 312–321. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2013.6606577

[59] ——, "Which configuration option should I change?" in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, 2014, pp. 152–163. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568251

[60] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Where do configuration constraints stem from? an extraction approach and an empirical study," *IEEE Trans. Software Eng.*, 2015.

[61] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. Le Traon, "Towards automated testing and fixing of re-engineered feature models," in *ICSE '13 (NIER track)*, 2013.

[62] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A classification and survey of analysis strategies for software product lines," *ACM Computing Surveys*, 2014.

[63] C. Kim, S. Khurshid, and D. Batory, "Shared execution for efficiently testing product lines," in *Software Reliability Engineering (ISSRE)*, 2012.

[64] H. V. Nguyen, C. Kästner, and T. N. Nguyen, "Exploring variability-aware execution for testing plugin-based web applications," in *ICSE'14*, 2014.

[65] I. do Carmo Machado, J. D. McGregor, and E. Santana de Almeida, "Strategies for testing products in software product lines," *ACM SIGSOFT Software Engineering Notes*, 2012.

[66] S. Souto, D. Gopinath, M. d'Amorim, D. Marinov, S. Khurshid, and D. Batory, "Faster bug detection for software product lines with incomplete feature models," in *SPLC '15*, 2015.

[67] C. H. P. Kim, D. S. Batory, and S. Khurshid, "Reducing combinatorics in testing product lines," in *AOSD'11*, 2011.

[68] M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay, "Beyond boolean product-line model checking: dealing with feature attributes and multi-features," in *ICSE*, 2013.

[69] C. Ghezzi and A. M. Sharifloo, "Verifying non-functional properties of software product lines: Towards an efficient approach using parametric model checking," in *Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011*, 2011, pp. 170–174. [Online]. Available: http://dx.doi.org/10.1109/SPLC.2011.33

[70] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. D.Amorim, "Splat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems," in *ESEC/FSE 2013*, 2013.

[71] C. Yilmaz, A. Porter, A. S. Krishna, A. M. Memon, D. C. Schmidt, A. S. Gokhale, and B. Natarajan, "Reliable effects screening: A distributed continuous quality assurance process for monitoring performance degradation in evolving software systems," *IEEE Transactions on Software Engineering*, vol. 33, no. 2, pp. 124–141, Feb 2007.

[72] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," *SIGPLAN Not.*, 2005. [Online]. Available: http://doi.acm.org/10.1145/1064978.1065036

[73] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: Whitebox fuzzing for security testing," *Queue*, 2012. [Online]. Available: http://doi.acm.org/10.1145/2090147.2094081

[74] J. Guo, E. Zulkoski, R. Olaechea, D. Rayside, K. Czarnecki, S. Apel, and J. M. Atlee, "Scaling exact multi-objective combinatorial optimization by parallelization," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 409–420. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642971

[75] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "Paramils: An automatic algorithm configuration framework," *J. Artif. Int. Res.*, vol. 36, no. 1, pp. 267–306, Sep. 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1734953.1734959

[76] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown, "Algorithm runtime prediction: Methods and evaluation," *Artificial Intelligence*, vol. 206, pp. 79 – 111, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0004370213001082

[77] N. V. Chawla, *Data Mining for Imbalanced Datasets: An Overview*. Boston, MA: Springer US, 2005, pp. 853–867. [Online]. Available: http://dx.doi.org/10.1007/0-387-25465-X_40

APPENDIX A
ALL METRICS FOR ALL CONFIGURABLE SYSTEMS

We recommend the reader to visualize heatmaps in color (*e.g.,* with a PDF viewer).

A.1 Accuracy

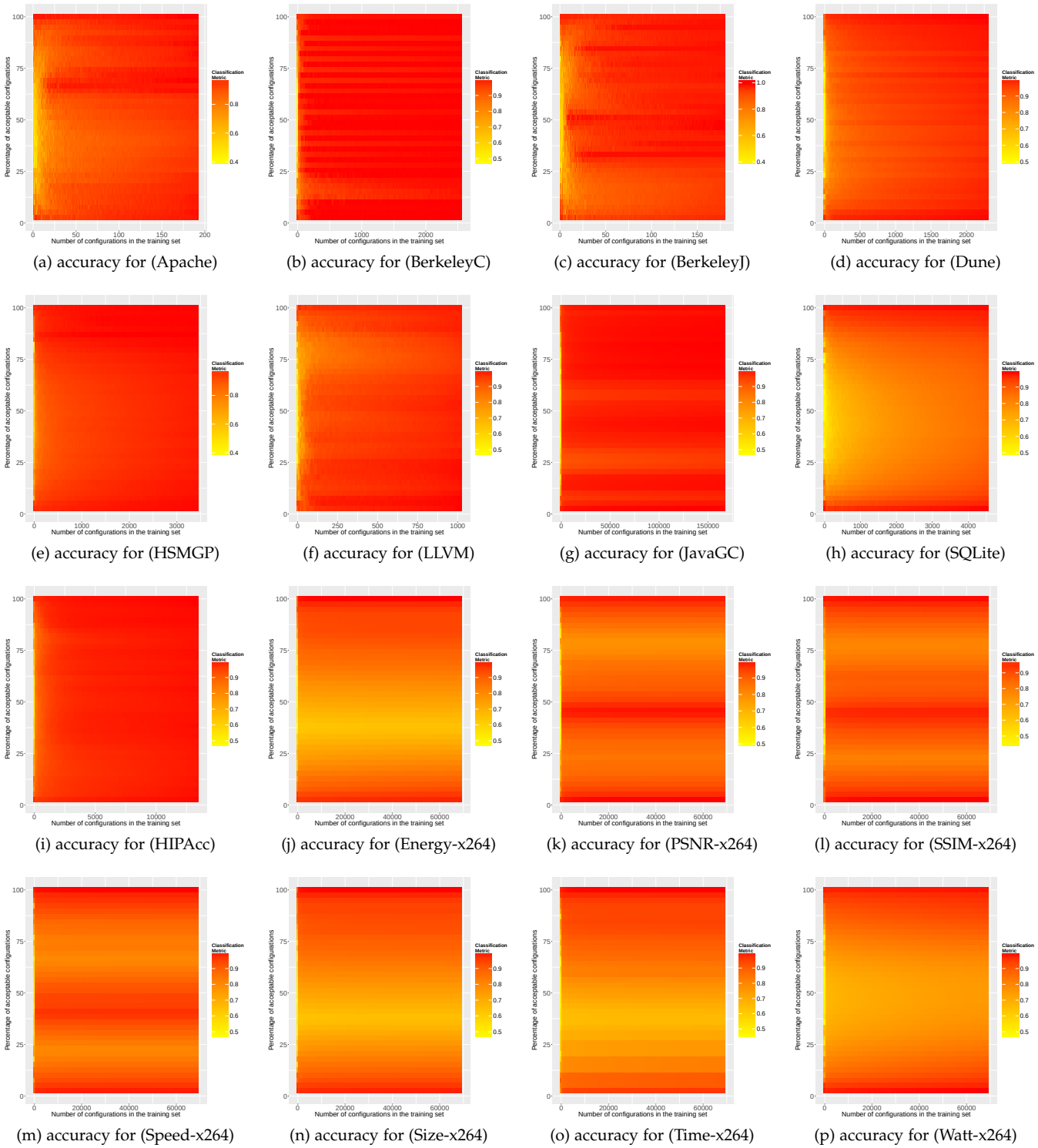
Figure 11 (see page 21) depicts all accuracy values of our learning-based technique.

A.2 Precision

Figure 12 (see page 22) depicts all precision values of our learning-based technique.

A.3 Recall

Figure 13 (see page 23) depicts all recall values of our learning-based technique.



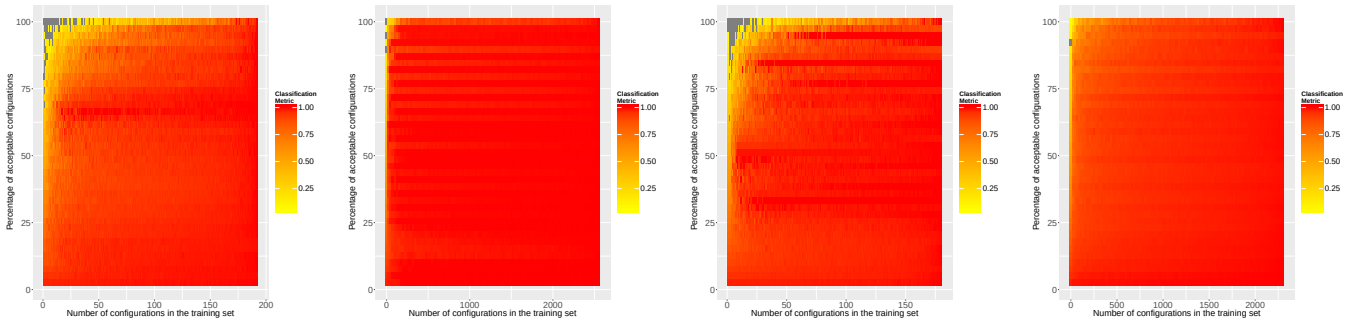
(a) accuracy for (Apache) (b) accuracy for (BerkeleyC) (c) accuracy for (BerkeleyJ) (d) accuracy for (Dune)

(e) accuracy for (HSMGP) (f) accuracy for (LLVM) (g) accuracy for (JavaGC) (h) accuracy for (SQLite)

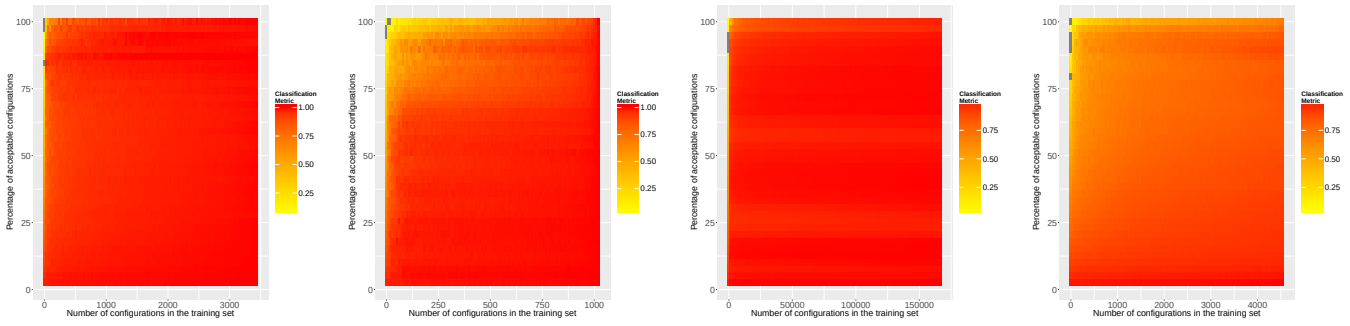
(i) accuracy for (HIPAcc) (j) accuracy for (Energy-x264) (k) accuracy for (PSNR-x264) (l) accuracy for (SSIM-x264)

(m) accuracy for (Speed-x264) (n) accuracy for (Size-x264) (o) accuracy for (Time-x264) (p) accuracy for (Watt-x264)

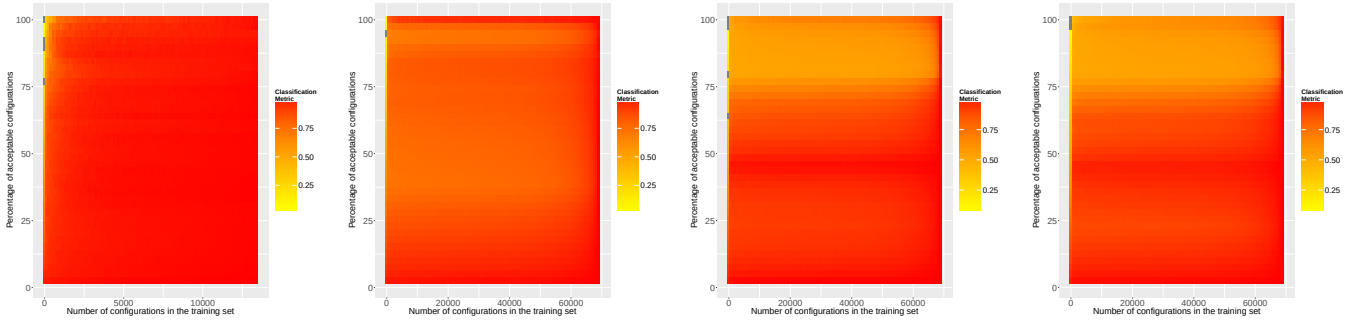
Fig. 11: accuracy for all systems through heatmaps. The darker, the higher. X-axis: number of configurations in the training set. Y-axis: percentage of acceptable configurations (due to objective threshold value)



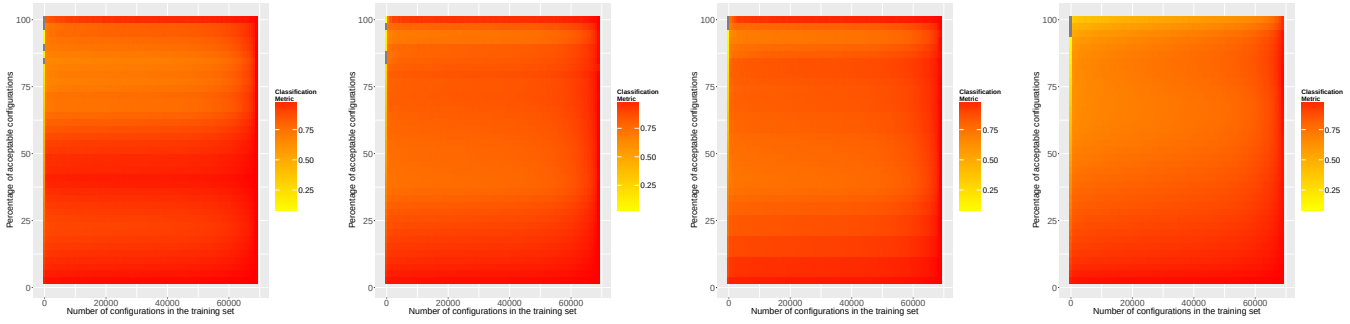
(a) precision for (Apache) (b) precision for (BerkeleyC) (c) precision for (BerkeleyJ) (d) precision for (Dune)



(e) precision for (HSMGP) (f) precision for (LLVM) (g) precision for (JavaGC) (h) precision for (SQLite)

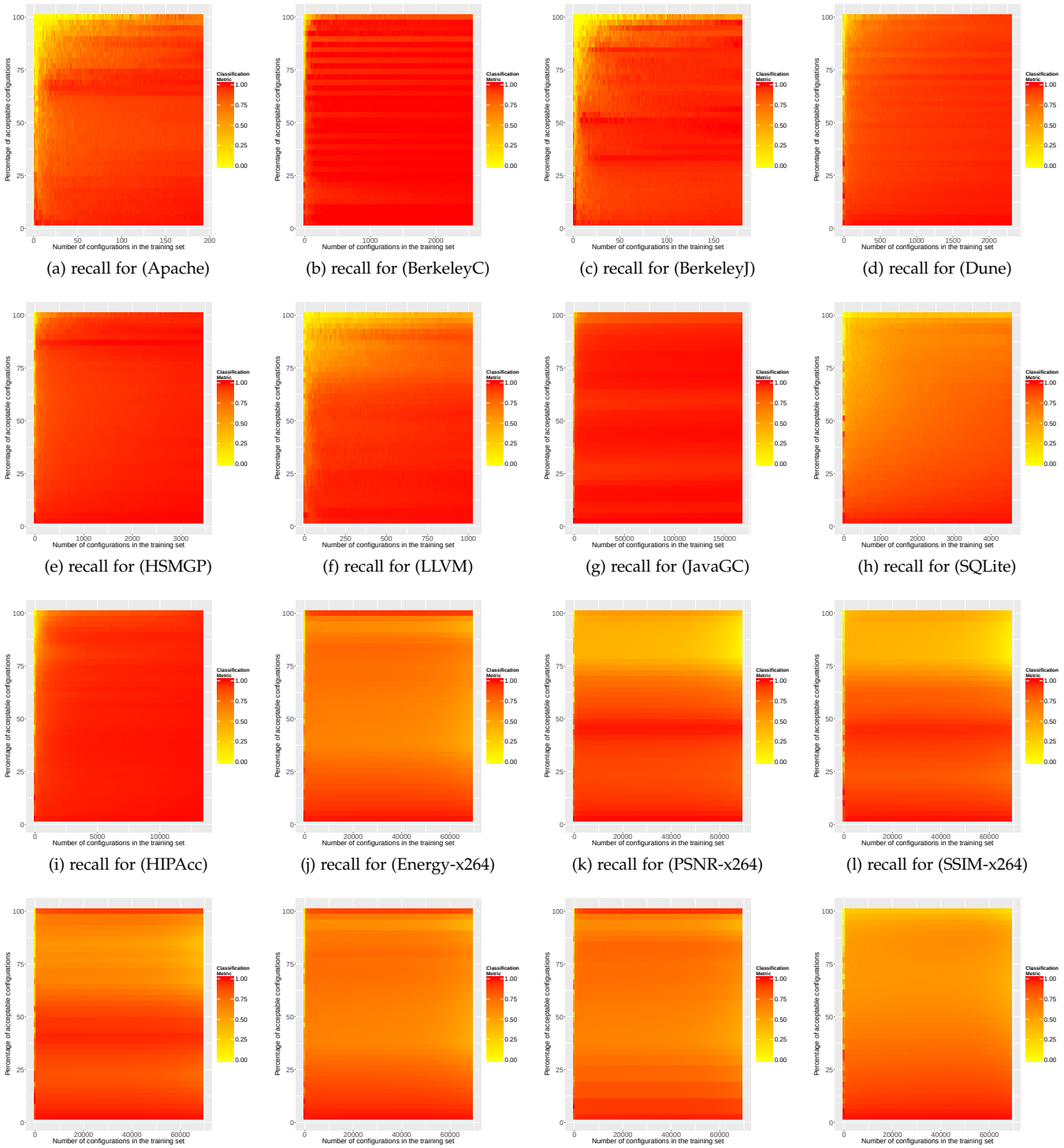


(i) precision for (HIPAcc) (j) precision for (Energy-x264) (k) precision for (PSNR-x264) (l) precision for (SSIM-x264)



(m) precision for (Speed-x264) (n) precision for (Size-x264) (o) precision for (Time-x264) (p) precision for (Watt-x264)

Fig. 12: precision for all systems through heatmaps. The darker, the higher. X-axis: number of configurations in the training set. Y-axis: percentage of acceptable configurations (due to objective threshold value)



(a) recall for (Apache) (b) recall for (BerkeleyC) (c) recall for (BerkeleyJ) (d) recall for (Dune)

(e) recall for (HSMGP) (f) recall for (LLVM) (g) recall for (JavaGC) (h) recall for (SQLite)

(i) recall for (HIPAcc) (j) recall for (Energy-x264) (k) recall for (PSNR-x264) (l) recall for (SSIM-x264)

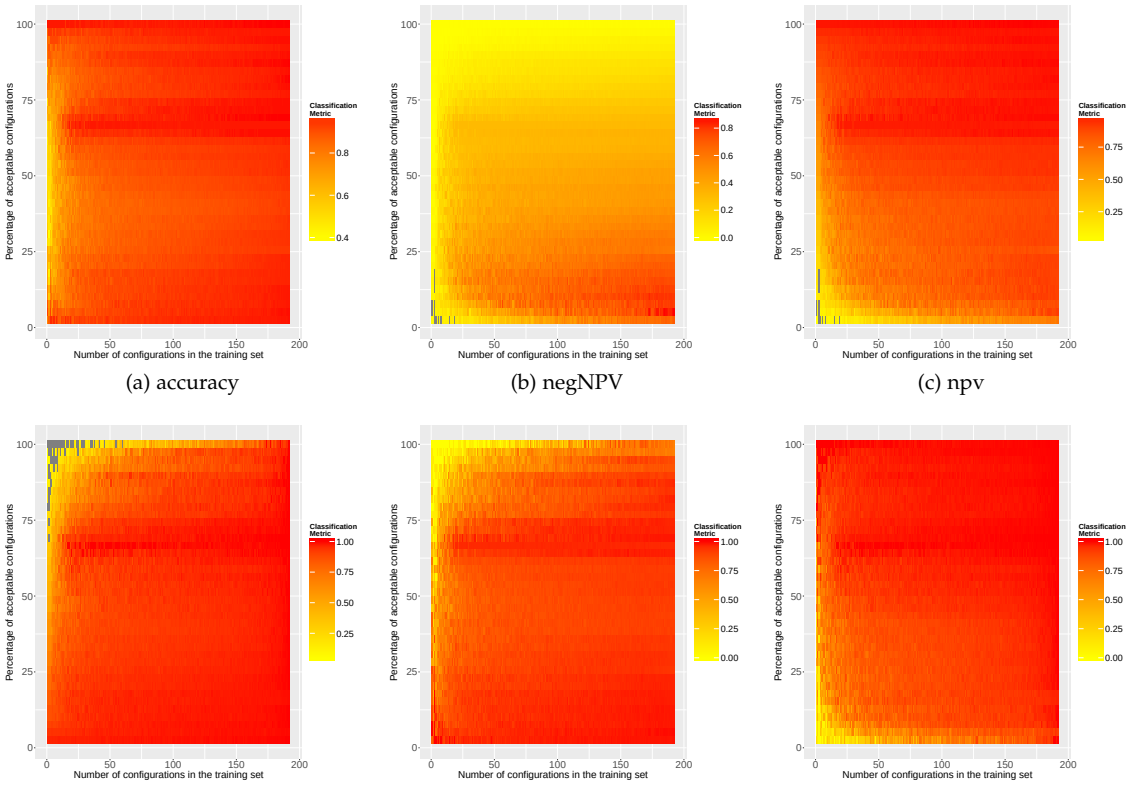
(m) recall for (Speed-x264) (n) recall for (Size-x264) (o) recall for (Time-x264) (p) recall for (Watt-x264)

Fig. 13: recall for all systems through heatmaps. The darker, the higher. X-axis: number of configurations in the training set. Y-axis: percentage of acceptable configurations (due to objective threshold value)

APPENDIX B
ALL HEATMAPS/METRICS PER CONFIGURABLE SYSTEM

Figure 14, Figure 15, Figure 16, Figure 17, Figure 18, Figure 19, Figure 20, Figure 21, Figure 22, Figure 23, Figure 24, Figure 25, Figure 26, Figure 27, Figure 28, Figure 29 depict all metrics ("accuracy", "precision", "recall", "specificity", "npv", "negNPV") per configurable system ("Apache", "BerkeleyC", "BerkeleyJ", "Dune", "HSMGP", "LLVM", "JavaGC", "SQLite", "HIPAcc", "Energy-x264", "PSNR-x264", "SSIM-x264", "Speed-x264", "Size-x264", "Time-x264", "Watt-x264").

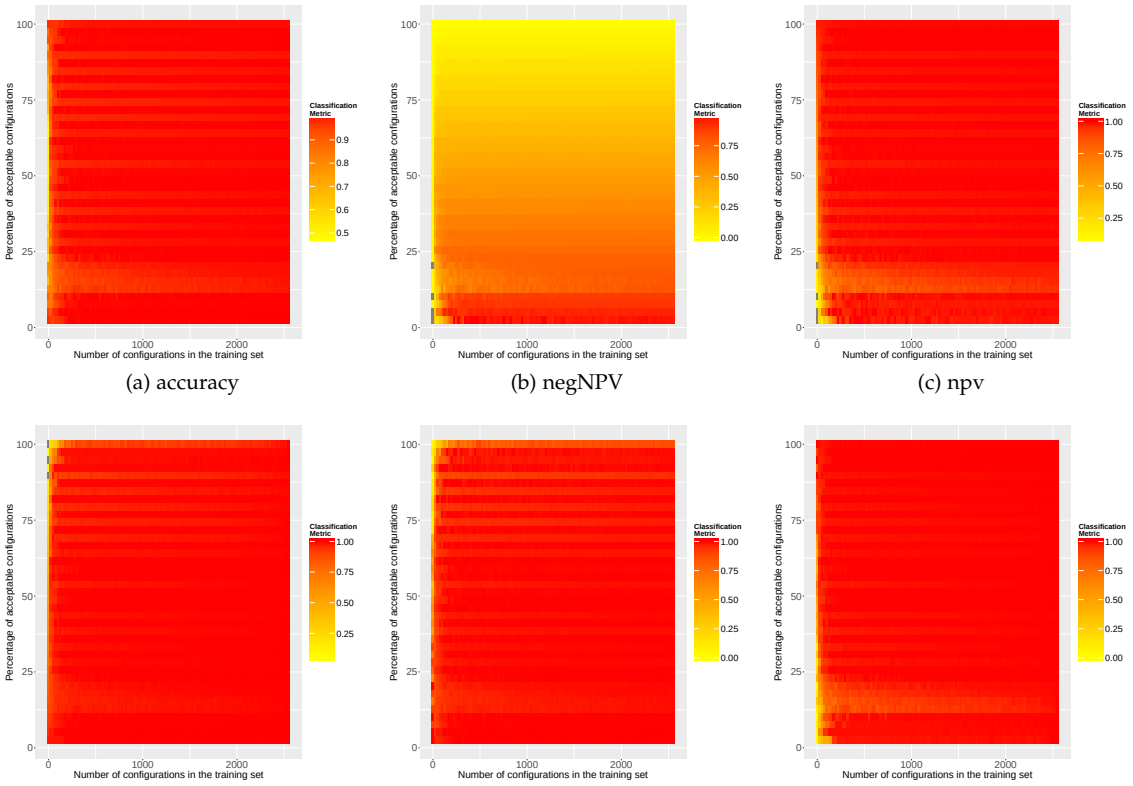
We recommend the reader to visualize heatmaps in color (*e.g.,* with a PDF viewer).



(a) accuracy (b) negNPV (c) npv

(d) precision (e) recall (f) specificity

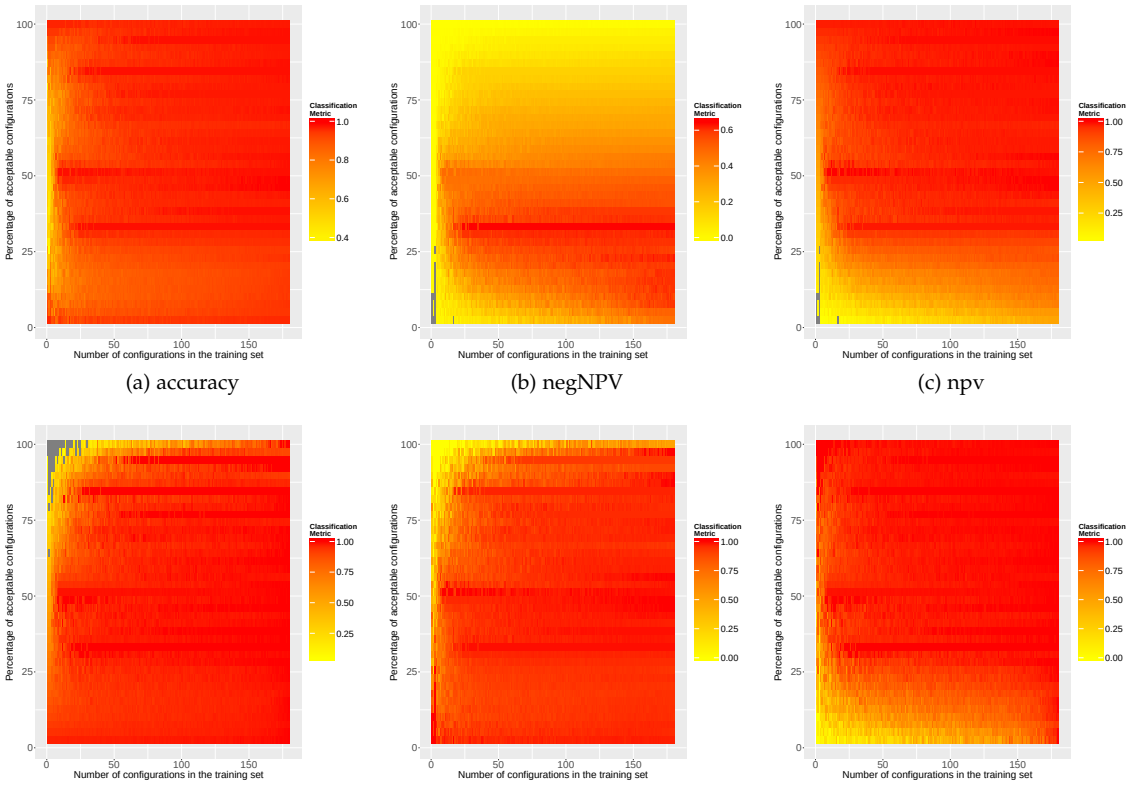
Fig. 14: Classification measures for Apache through heatmaps. The darker, the higher is the classification. X-axis: number of configurations in the training set. Y-axis: percentage of acceptable configurations (due to objective threshold value)



(a) accuracy (b) negNPV (c) npv

(d) precision (e) recall (f) specificity

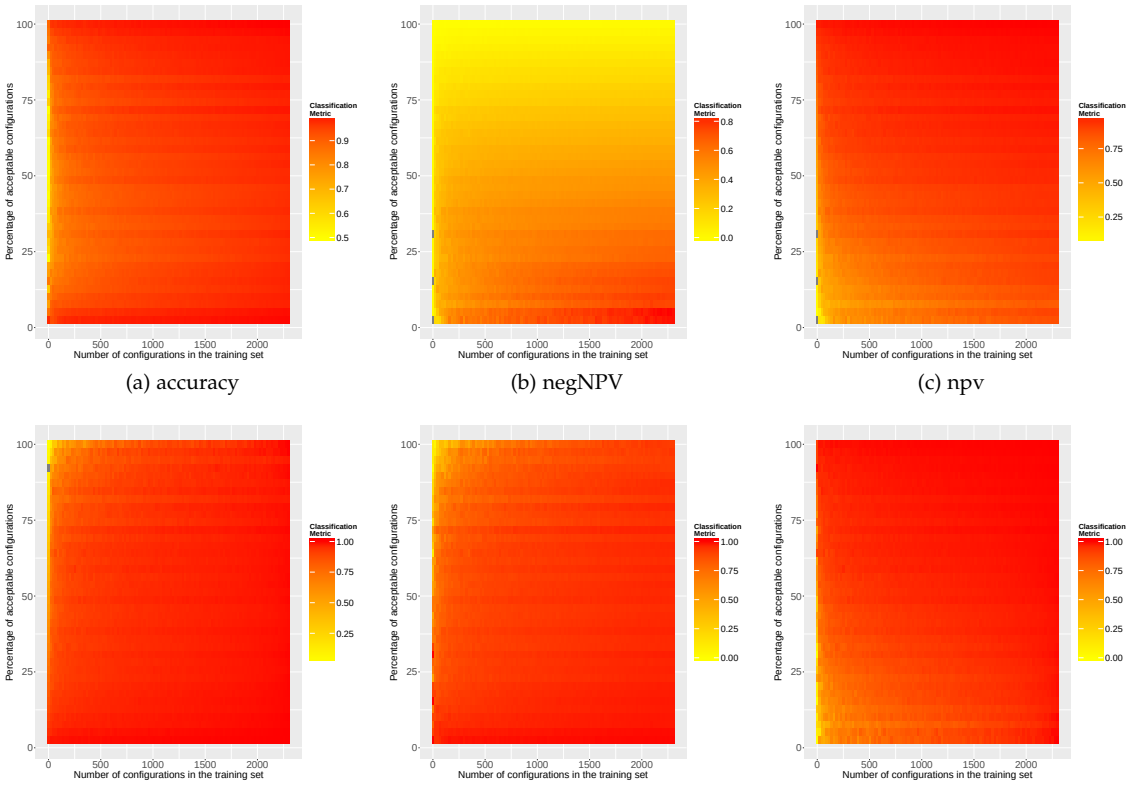
Fig. 15: Classification measures for BerkeleyC through heatmaps. The darker, the higher is the classification. X-axis: number of configurations in the training set. Y-axis: percentage of acceptable configurations (due to objective threshold value)



(a) accuracy (b) negNPV (c) npv

(d) precision (e) recall (f) specificity

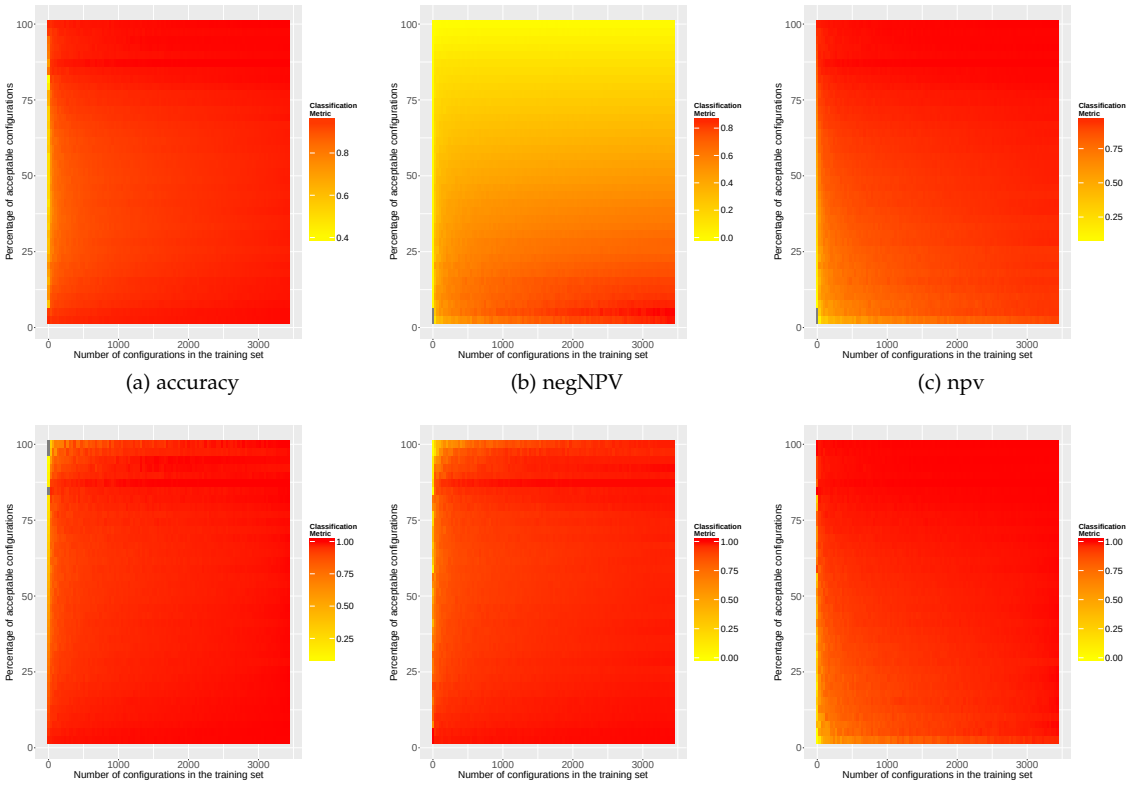
Fig. 16: Classification measures for BerkeleyJ through heatmaps. The darker, the higher is the classification. X-axis: number of configurations in the training set. Y-axis: percentage of acceptable configurations (due to objective threshold value)



(a) accuracy (b) negNPV (c) npv

(d) precision (e) recall (f) specificity

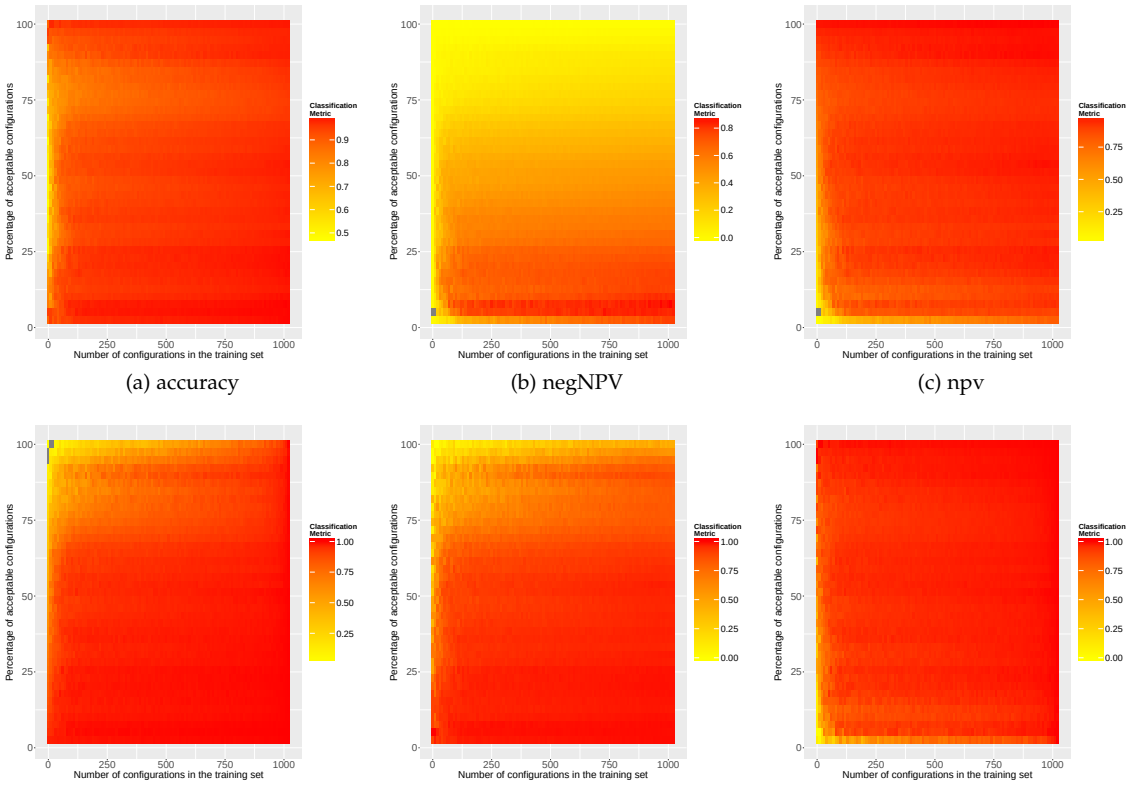
Fig. 17: Classification measures for Dune through heatmaps. The darker, the higher is the classification. X-axis: number of configurations in the training set. Y-axis: percentage of acceptable configurations (due to objective threshold value)



(a) accuracy (b) negNPV (c) npv

(d) precision (e) recall (f) specificity

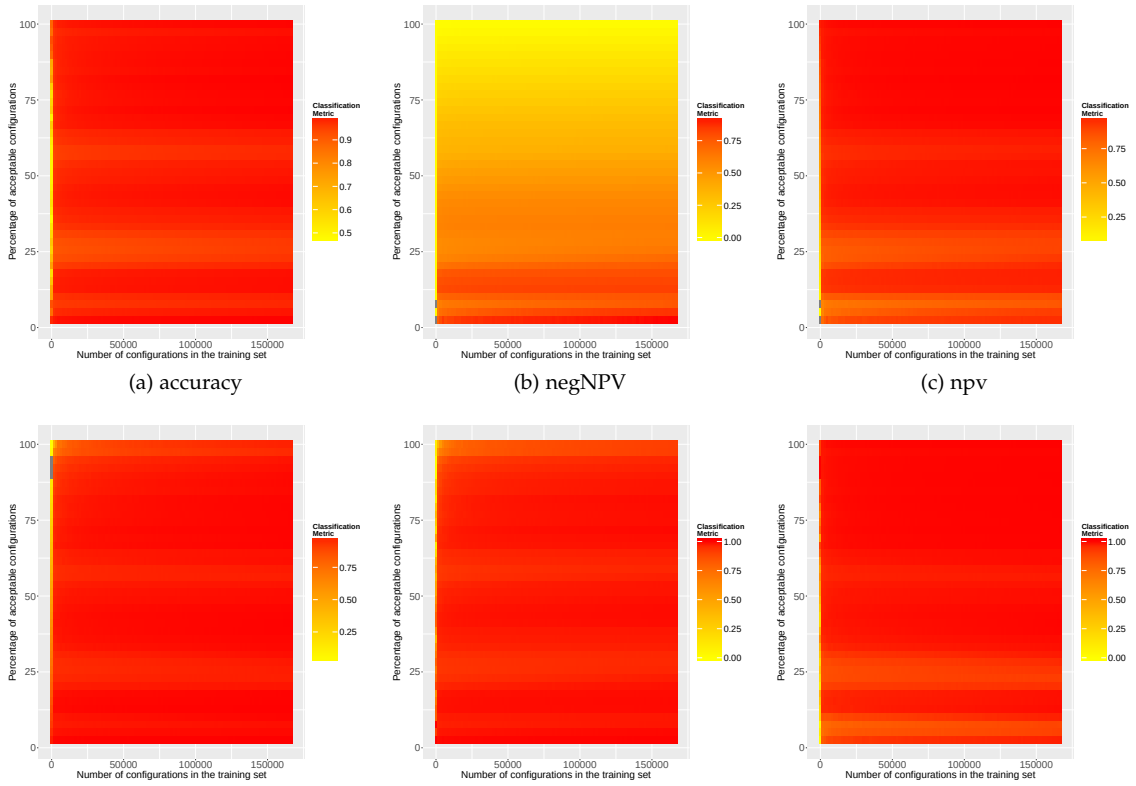
Fig. 18: Classification measures for HSMGP through heatmaps. The darker, the higher is the classification. X-axis: number of configurations in the training set. Y-axis: percentage of acceptable configurations (due to objective threshold value)



(a) accuracy (b) negNPV (c) npv

(d) precision (e) recall (f) specificity

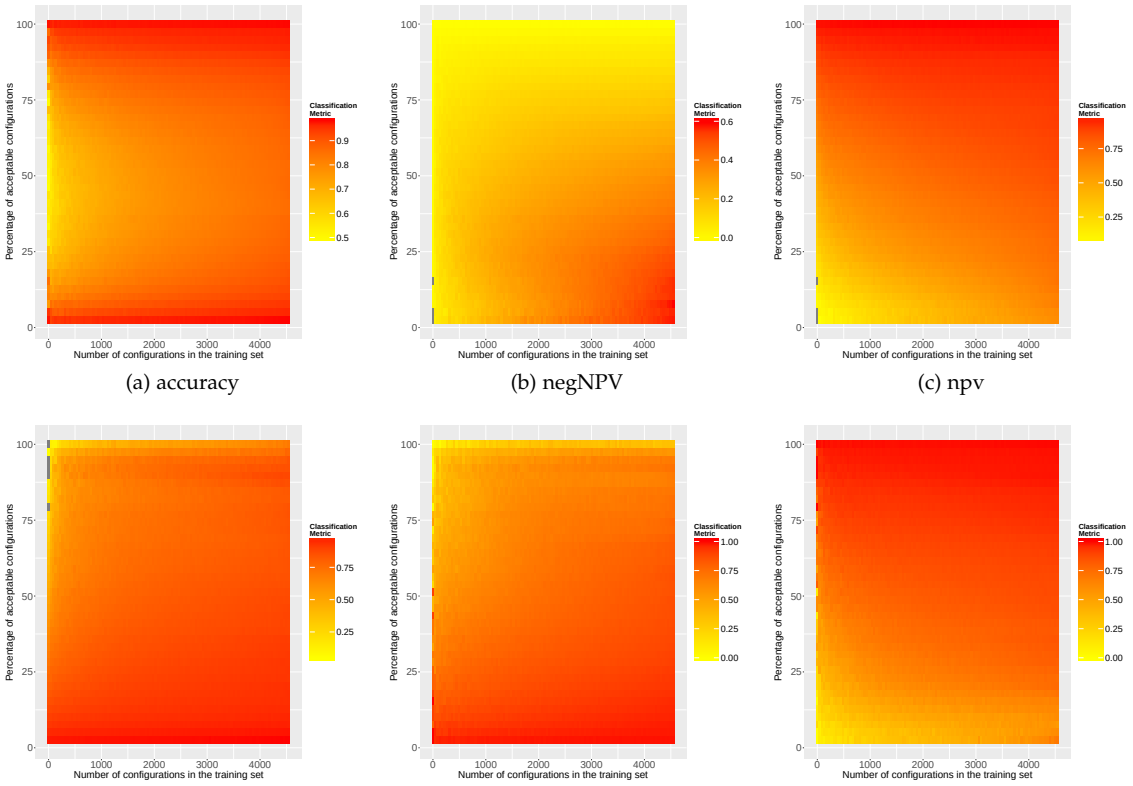
Fig. 19: Classification measures for LLVM through heatmaps. The darker, the higher is the classification. X-axis: number of configurations in the training set. Y-axis: percentage of acceptable configurations (due to objective threshold value)



(a) accuracy (b) negNPV (c) npv

(d) precision (e) recall (f) specificity

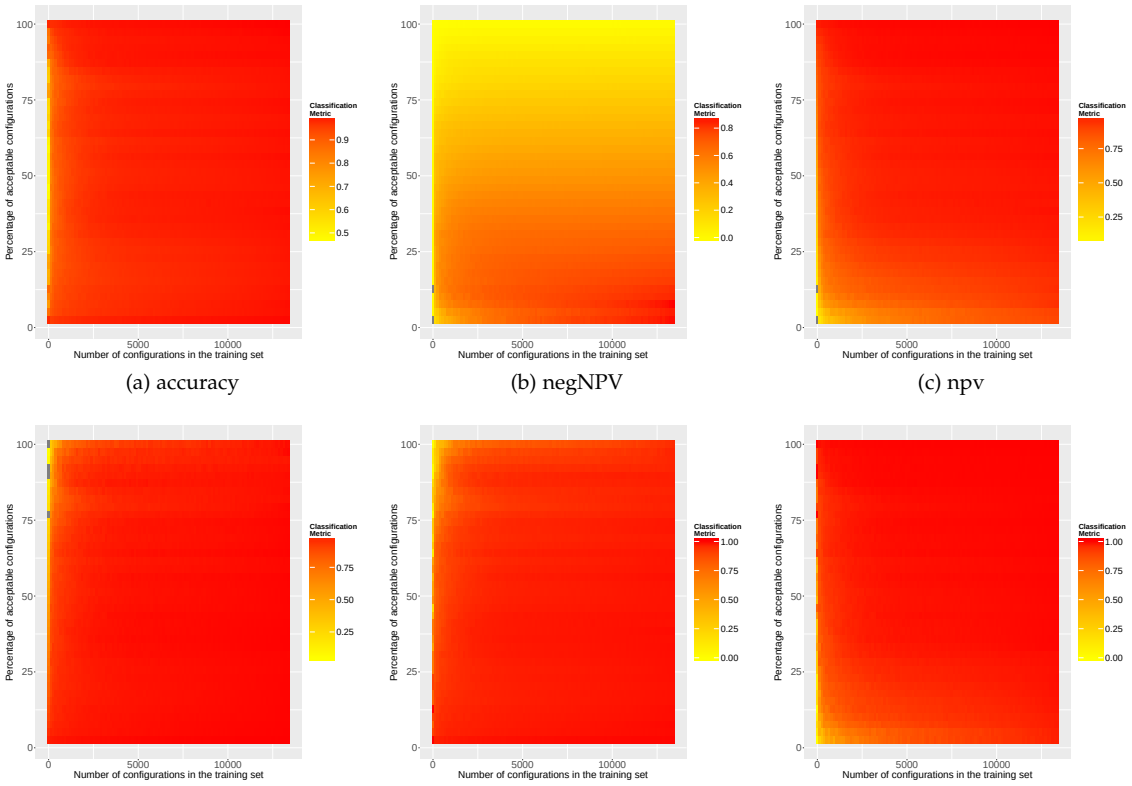
Fig. 20: Classification measures for JavaGC through heatmaps. The darker, the higher is the classification. X-axis: number of configurations in the training set. Y-axis: percentage of acceptable configurations (due to objective threshold value)



(a) accuracy (b) negNPV (c) npv

(d) precision (e) recall (f) specificity

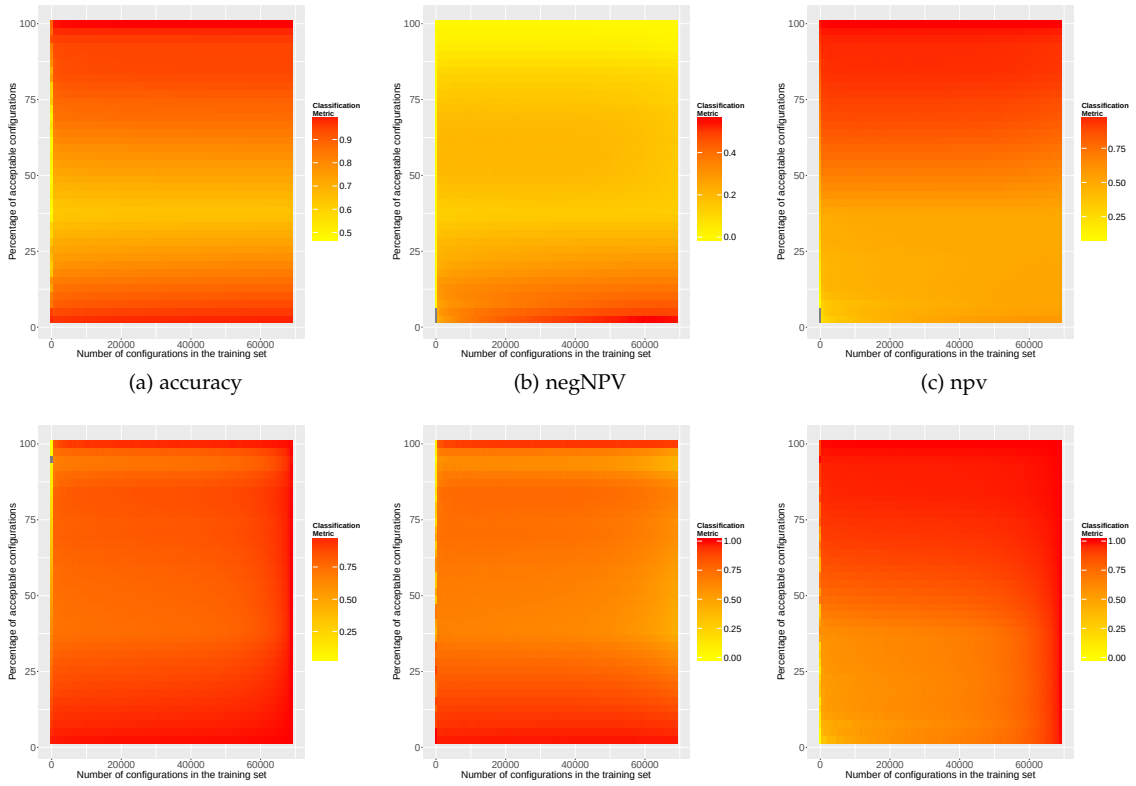
Fig. 21: Classification measures for SQLite through heatmaps. The darker, the higher is the classification. X-axis: number of configurations in the training set. Y-axis: percentage of acceptable configurations (due to objective threshold value)



(a) accuracy (b) negNPV (c) npv

(d) precision (e) recall (f) specificity

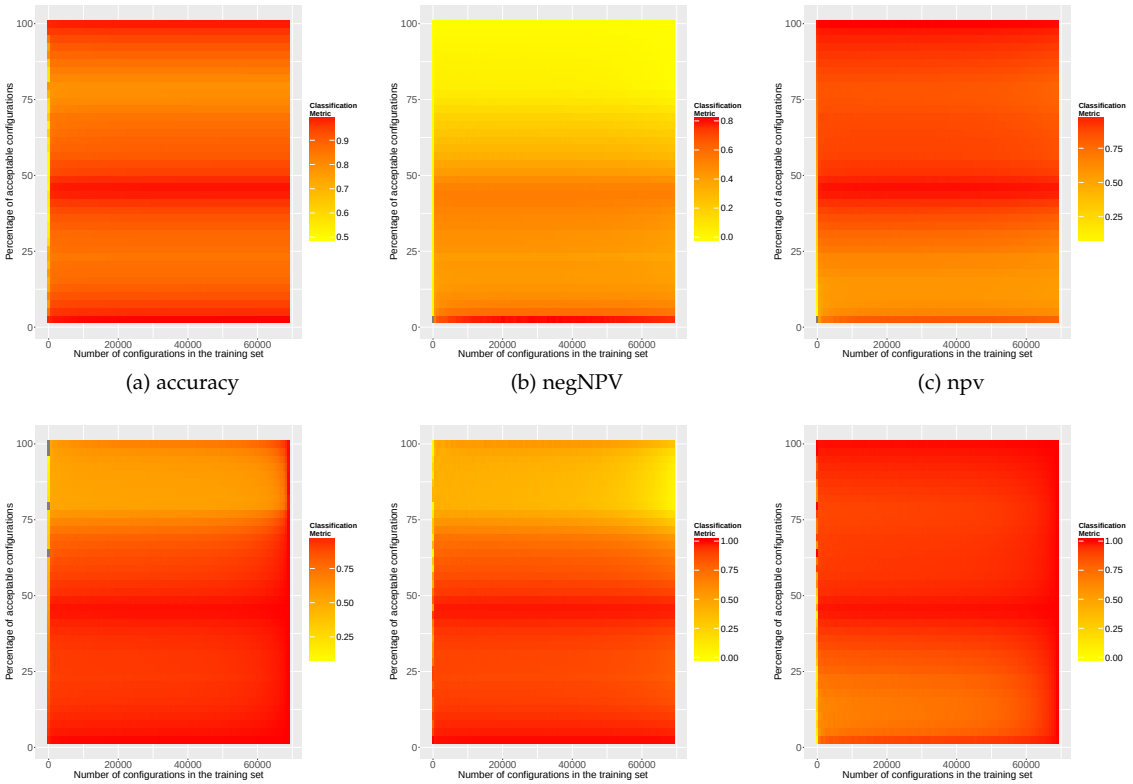
Fig. 22: Classification measures for HIPAcc through heatmaps. The darker, the higher is the classification. X-axis: number of configurations in the training set. Y-axis: percentage of acceptable configurations (due to objective threshold value)



(a) accuracy (b) negNPV (c) npv

(d) precision (e) recall (f) specificity

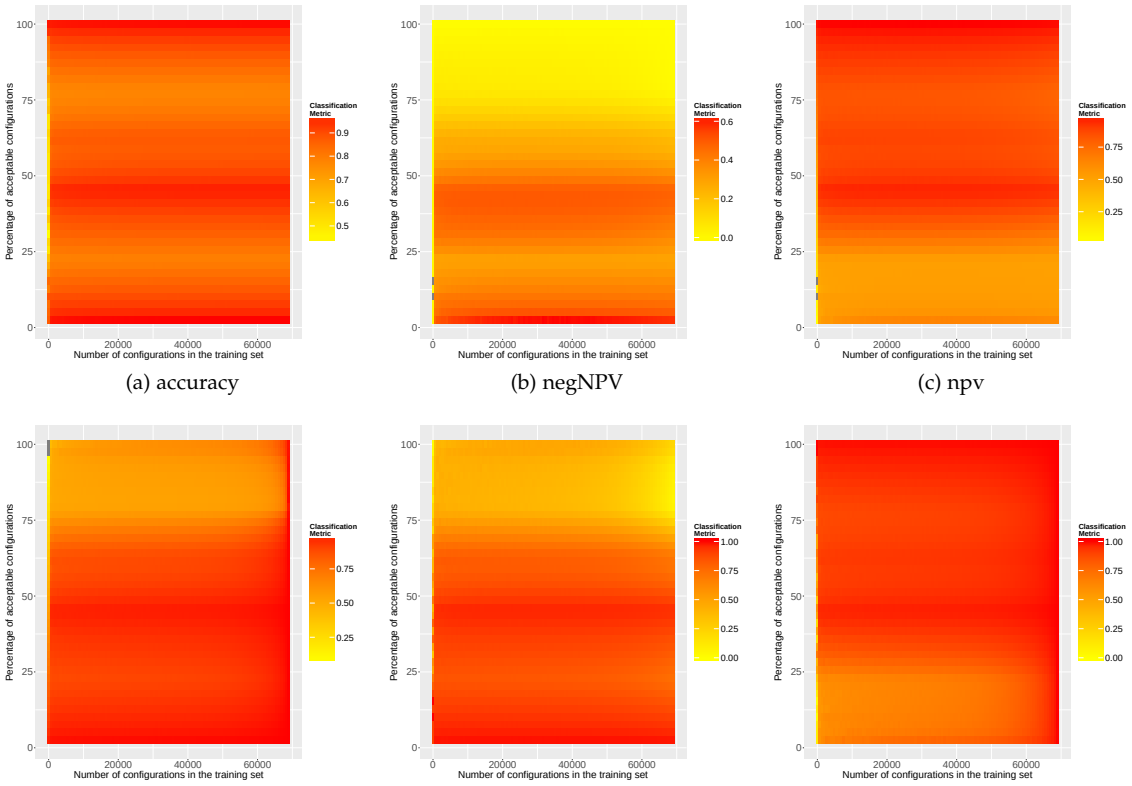
Fig. 23: Classification measures for Energy-x264 through heatmaps. The darker, the higher is the classification. X-axis: number of configurations in the training set. Y-axis: percentage of acceptable configurations (due to objective threshold value)



(a) accuracy (b) negNPV (c) npv

(d) precision (e) recall (f) specificity

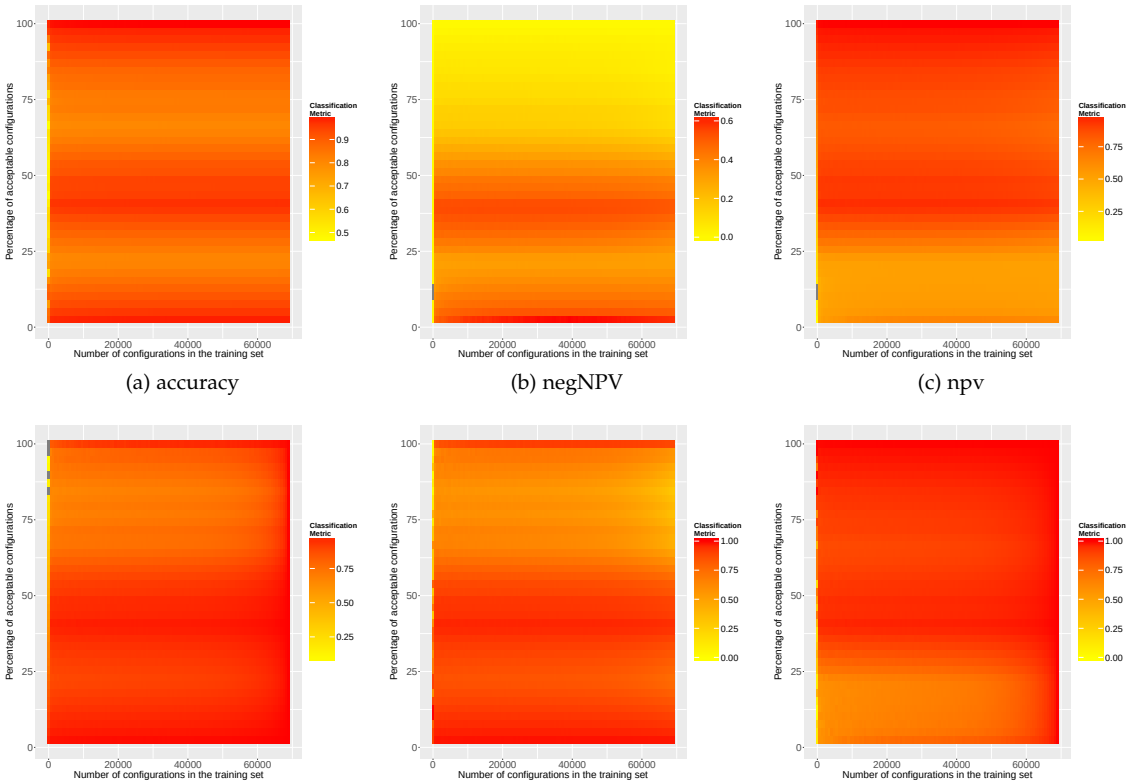
Fig. 24: Classification measures for PSNR-x264 through heatmaps. The darker, the higher is the classification. X-axis: number of configurations in the training set. Y-axis: percentage of acceptable configurations (due to objective threshold value)



(a) accuracy (b) negNPV (c) npv

(d) precision (e) recall (f) specificity

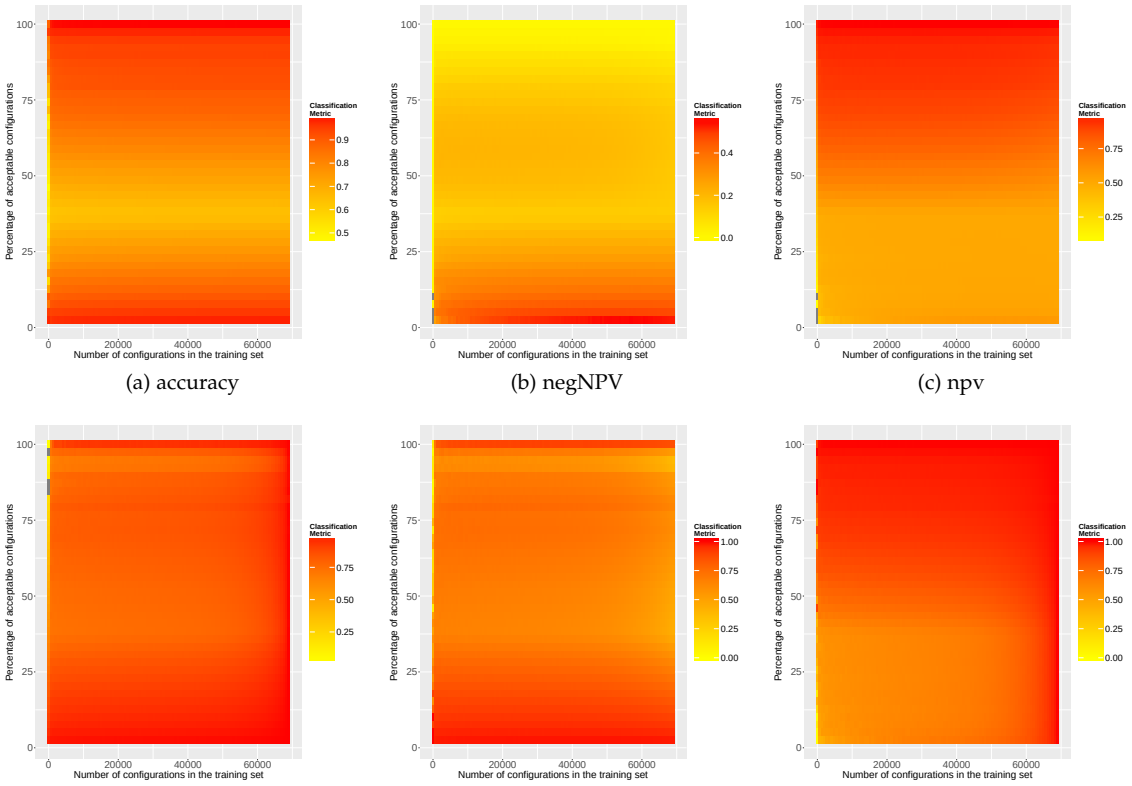
Fig. 25: Classification measures for SSIM-x264 through heatmaps. The darker, the higher is the classification. X-axis: number of configurations in the training set. Y-axis: percentage of acceptable configurations (due to objective threshold value)



(a) accuracy (b) negNPV (c) npv

(d) precision (e) recall (f) specificity

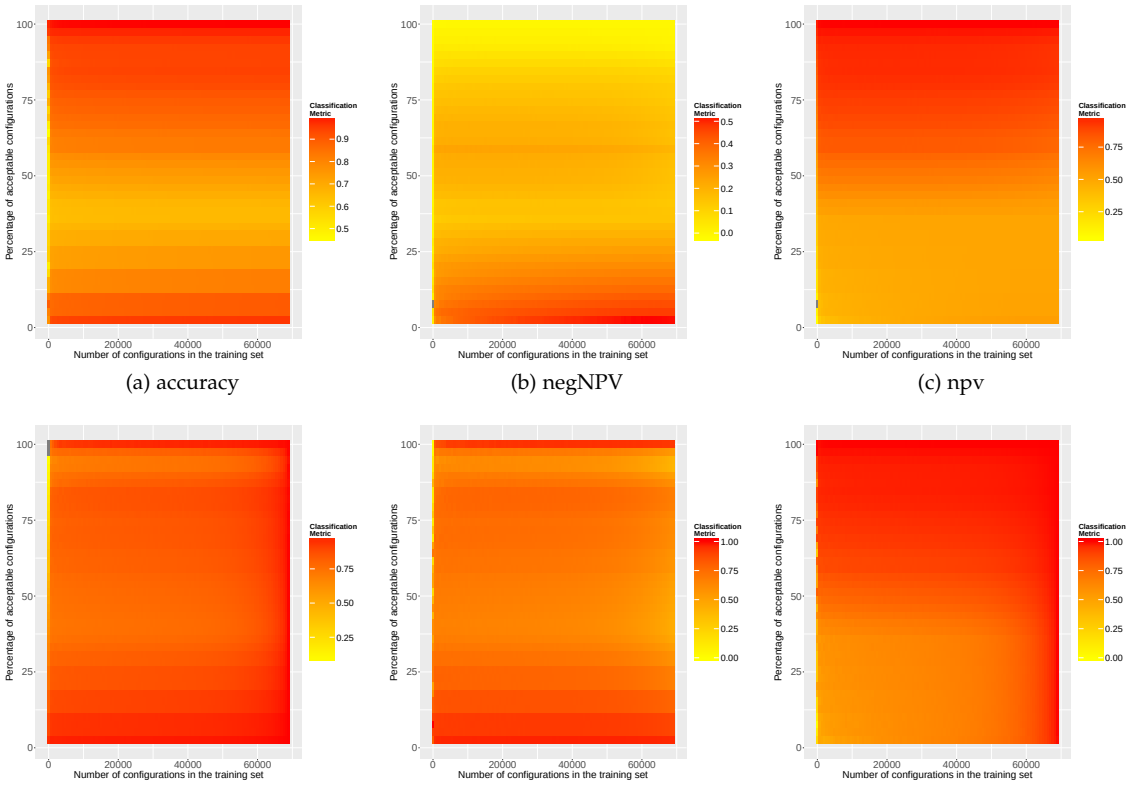
Fig. 26: Classification measures for Speed-x264 through heatmaps. The darker, the higher is the classification. X-axis: number of configurations in the training set. Y-axis: percentage of acceptable configurations (due to objective threshold value)



(a) accuracy (b) negNPV (c) npv

(d) precision (e) recall (f) specificity

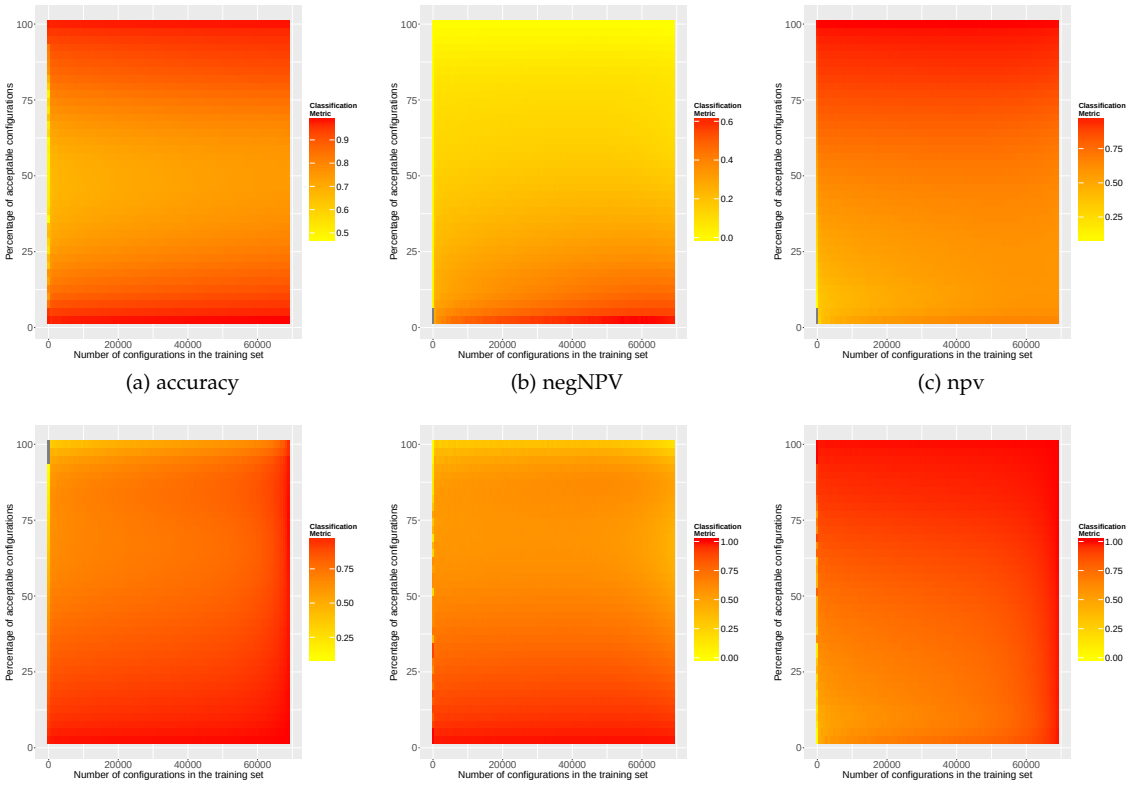
Fig. 27: Classification measures for Size-x264 through heatmaps. The darker, the higher is the classification. X-axis: number of configurations in the training set. Y-axis: percentage of acceptable configurations (due to objective threshold value)



(a) accuracy (b) negNPV (c) npv

(d) precision (e) recall (f) specificity

Fig. 28: Classification measures for Time-x264 through heatmaps. The darker, the higher is the classification. X-axis: number of configurations in the training set. Y-axis: percentage of acceptable configurations (due to objective threshold value)



(a) accuracy (b) negNPV (c) npv

(d) precision (e) recall (f) specificity

Fig. 29: Classification measures for Watt-x264 through heatmaps. The darker, the higher is the classification. X-axis: number of configurations in the training set. Y-axis: percentage of acceptable configurations (due to objective threshold value)