Machine learning for performance modelling on colossal software configuration spaces

Hugo Martin

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies
de l'Information et de la Communication
Spécialité : *« Informatique »*

Par

Hugo MARTIN

Machine Learning for Performance Modelling on Colossal Software Configuration Spaces

Thèse présentée et soutenue à Rennes, le 16 Décembre 2021
Unité de recherche : Equipe DiverSE, IRISA

Rapporteurs avant soutenance :

Laurence DUCHIEN	Professeur à l'Université de Lilles
Julia LAWALL	Directrice de Rercherche à INRIA - Paris

Composition du Jury :

Attention, en cas d'absence d'un des membres du Jury le jour de la soutenance, la composition du jury doit être revue pour s'assurer qu'elle est conforme et devra être répercutée sur la couverture de thèse

Président :	Isabelle PUAUT	Professeur à l'Université de Rennes 1
Examinateurs :	Laurence DUCHIEN	Professeur à l'Université de Lille
	Julia LAWALL	Directrice de Recherche à INRIA - Paris
	Isabelle PUAUT	Professeur à l'Université de Rennes 1
	Klaus Schmid	Professeur à l'Université de Hildesheim
Dir. de thèse :	Mathieu ACHER	Professeur à l'Université de Rennes 1
Co-dir. de thèse :	Jean-Marc JEZEQUEL	Professeur à l'Université de Rennes 1

Résumé en français

Presque tous les systèmes logiciels d'aujourd'hui sont configurables. A l'aide d'options, il est possible de modifier le comportement de ces systèmes, d'ajouter ou d'enlever certaines capacités pour améliorer leurs performances ou les adapter à différentes situations. Chacune de ces options est liée à certains parties de code, et s'assurer du bon fonctionnement de ces parties entre elles, ou de les empêcher d'être utilisées ensemble, est un des défis durant le développement et l'utilisation de ces logiciels, connus sous le nom de Lignes de Produits Logiciel (ou SPL pour Software Product Lines). Si cela peut sembler relativement simple avec quelques options, certains logiciels assemblent des milliers d'options réparties sur des millions de lignes de code, ce qui rend la tâche autrement plus complexe.

Durant la dernière décennie, les chercheurs ont commencé a utiliser des techniques d'apprentissage automatique afin de répondre aux problématiques des Lignes de Produits Logiciels. Un des problèmes clés est la prédiction des différentes propriétés de ces logiciels, comme la vitesse d'exécution d'une tâche, qui peut fortement varier selon la configuration du logiciel utilisé. Mesurer les propriétés pour chaque configuration peut être coûteux et complexe, voire impossible dans les cas les plus extrêmes. La création d'un modèle permttant de prédire les propriétés du logiciel, en s'aidant des mesures sur seulement d'une faible partie des configurations possible est une tâche dans laquelle l'apprentissage automatique excelle.

Différentes solutions ont été developées, mais elles n'ont été validées que dans des cas où le nombre d'options est assez faible. Or, une part importante des SPL sont dotés de plusieurs centaines voire plusieurs milliers d'options. Sans tester les solutions d'apprentissage automatique sur des logiciels avec autant d'options, il est impossible de savoir si ces solutions sont adaptées pour de tels cas.

La première contribution de cette thèse est l'application d'algorithmes d'apprentissage automatique sur une ligne de produits logiciels à une échelle jamais atteinte auparavant. Le noyau Linux, avec ses 15.000 options, dépasse complètement toutes les études de cas sur les Lignes de Produits Logiciels présentes dans la littérature, qui n'ont utilisé que des cas d'au mieux 60 options. En évaluant la précision de plusieurs algorithmes de l'état de l'art pour prédire la taille binaire du noyau Linux, nous avons pu déterminer les limites de

certaines techniques. Sans surprise, les algorithmes linéaires donnent de mauvais résultats en termes de précision, car ils ne peuvent pas appréhender la complexité causée par les interactions entre les options. Le modèle de performance-influence, bien que développé spécifiquement pour les SPL, ne peut pas gérer autant d'options malgré les mesures prises pour réduire l'explosion combinatoire. En fin de compte, nous avons constaté que les algorithmes basés sur les arbres, ainsi que les réseaux de neurones, étaient capables de fournir un modèle assez précis avec des ressources raisonnables en termes de temps et de mémoire.

La deuxième contribution est la Feature Ranking List, une liste des options classées par importance envers leur impact sur une propriété cible d'un logiciel, générée par une amélioration de la sélection des caractéristiques basée sur les arbres de décisions. Nous avons évalué ses effets sur les modèles de prédiction de la taille des binaires du noyau Linux dans les mêmes conditions que pour la première contribution. L'effet souhaité et le plus connu de la sélection de caractéristiques en général est une accélération majeure du temps d'apprentissage, le divisant par 5 pour les Gradient Boosting Trees et de 20 pour les Random Forests. De manière plus inattendue, nous constatons également une amélioration notable de la précision pour la plupart des algorithmes précédemment considérés. La Feature Ranking List est lisible par l'homme, et lorsqu'elle a été présentée à des experts, elle a été jugée assez juste. Cependant, il y a une grande divergence lorsqu'on la confronte à la documentation, principalement en raison du très faible nombre d'options explicitement documentées comme ayant un impact sur la taille du noyau Linux.

La troisième contribution est l'amélioration de la spécialisation automatisée des performances et son évaluation sur différents SPL dont Linux. La spécialisation des performances est un processus qui consiste à ajouter des contraintes sur un SPL afin de répondre à un certain seuil de performance défini par l'utilisateur, pour l'aider lors de la configuration du logiciel. Il est possible d'automatiser ce processus en utilisant l'apprentissage automatique, et plus précisément des arbres de décisions classifieurs qui prédisent si une configuration est acceptable ou non par rapport au seuil de performance, à partir desquels nous pouvons extraire des règles, ensuite appliquées comme contraintes sur la SPL cible. Les améliorations portent sur deux aspects. Le premier est l'application de la sélection des options basée sur la Feature Ranking List, qui a permis d'améliorer à la fois la précision et le temps d'apprentissage, même sur des SPL avec seulement quelques options. La seconde est la prise en compte des arbres de décisions régresseurs, qui produisent un modèle similaire pour l'extraction de règles, mais avec l'avantage d'être conscient de la performance, alors que cette dimension est réduite à un simple booléen dans un processus de classification. Nous avons développé une nouvelle technique, basée sur les arbres de décisions régresseurs, où nous simulons un écart dans la distribution de la performance pour rendre l'algorithme conscient du seuil en ce qui concerne la fonction de perte. En fin de compte, les trois techniques ont leurs propres forces et faiblesses, avec leurs propres cas d'utilisation, et doivent être considérées comme complémentaires.

La dernière contribution ouvre la porte à une utilisation plus durable de l'apprentissage automatique pour les lignes de produits logiciels. Dans ce domaine, peu voire aucune attention n'a été accordée à la précision des modèles sur différentes versions d'un SPL. Nous avons donc pris un modèle très précis issu des travaux de notre deuxième contribution, l'avons évalué sur des versions ultérieures et avons observé deux problèmes majeurs. Le premier est la différence d'options, car entre deux versions, certaines options apparaissent et d'autres disparaissent, créant un décalage dans l'espace de configuration. Le second est la baisse de précision lorsque le modèle est utilisé sur une autre version que celle sur laquelle il a été entraîné. Le coût pour entraîner le premier modèle était très important, en termes de mesures de configuration et d'efforts d'entraînement, et une telle dépense n'est pas raisonnable lorsque l'on considère une nouvelle version tous les deux ou trois mois. Pour résoudre ces deux problèmes, nous avons créé Evolution-Aware Model Shifting, une technique d'apprentissage par transfert adaptée à l'échelle et au rythme d'évolution du noyau Linux. Elle exploite le modèle original mais déprécié et, pour une fraction du coût initial, l'adapte aux nouvelles versions. Nos résultats montrent une très bonne précision sur trois ans d'évolution, et une meilleure précision que l'apprentissage à partir de zéro sur le même ensemble de données.

Abstract

Variability is the blessing and the curse of today software development. On one hand, it allows for fast and cheap development, while offering efficient customization to precisely meet the needs of a user. On the other hand, the increase in complexity of the systems due to the sheer amount of possible configurations makes it hard or even impossible for users to correctly utilize them, for developers to properly test them, or for experts to precisely grasp their functioning.

Machine Learning is a research domain that grew in accessibility and variety of usages over the last decades. It attracted interest from researchers from the Software Engineering domain for its ability to handle the complexity of Software Product Lines on problems they were tackling such as performance prediction or optimization. However, all studies presenting learning-based solutions in the SPL domain failed to explore the scalability of their techniques on systems with colossal configuration space (> 1000 options).

In this thesis, we focus on the Linux Kernel. With more than 15.000 options, it is very representative of the complexity of systems with colossal configuration spaces. We first apply various learning techniques to predict the kernel binary size, and report that most of the techniques fail to produce accurate results. In particular, performance-influence model, a learning technique tailored for SPL problem, does not even work on such large dataset. Among the tested techniques, only Tree-based algorithms and Neural Networks are able to produce an accurate model in an acceptable time.

To mitigate the problems created by colossal configuration spaces on learning techniques, we propose a feature selection technique leveraging Random Forest, enhanced toward better stability. We show that by using the feature selection, the training time can be greatly reduced, and the accuracy can be improved. This Tree-based feature selection technique is also completely automated and does not rely on prior knowledge on the system.

Performance specialization is a technique that constrains the configuration space of a software system to meet a given performance criterion. It is possible to automate the specialization process by leveraging Decision Trees. While only Decision Tree Classifier has been used for this task, we explore the usage of Decision Tree Regressor, as well as a

novel hybrid approach. We test and compare the different approaches on a wide range of systems, as well as on Linux to ensure the scalability on colossal configuration spaces. In most cases, including Linux, we report at least 90% accuracy, and each approach having their own particular strength compared to the others. At last, we also leverage the Tree-based feature selection, whose most notorious effect is the reduction of the training time of Decision Trees on Linux, downing from one minute to a second or less.

The last contribution explores the sustainability of a performance model across versions of a configurable system. We reused the model trained on the 4.13 version of Linux from our first contribution, and measured its accuracy on six later versions up to 5.8, spanning over three years. We show that a model is quickly outdated and unusable as is. To preserve the accuracy of the model over versions, we use transfer learning with the help of Tree-based algorithms to maintain it at a reduced cost. We tackle the problem of heterogeneity of the configuration space, that is evolving with each version. We show that the transfer approach allows for an acceptable accuracy at low cost, and vastly outperforms a learning from scratch approach using the same budget.

Overall, this thesis focuses on the problems of systems with colossal configuration spaces such as Linux, and show that Tree-based algorithms are a valid solution, versatile enough to answer a wide range of problem, and accurate enough to be considered.

ACKNOWLEDGEMENT

My first thank go to you, reader, as a big lesson I learned during this thesis, is that science is valuable only if it is passed to others, and reading this thesis is as much important as writing it.

I want to thank The University of Rennes 1 and all the professors for their teaching all along my 9 years of studies. In particular, Professor Rumen Andonov, you deserve a special place in my thoughts, as your decision to welcome that kind of lost student on that first day of L3 leads to this thesis. Thank you for the trust you placed in me that day.

Thanks to the jury members who accepted to read this thesis and review the works described in it. The feedback I received has been used, to my great pleasure, to improve the manuscript which is, in my humble opinion, far better now than when I send it to review.

Thanks to all master students who worked on the TuxML project, which have been fundamental for the works realized in this thesis.

Paul, you guided me on my first steps toward machine learning and always ensured I had all I needed, even when I did not ask while needing it, thank you.

Juliana, you scientific excellence impressed me, and I have been very please to be given the opportunity to work with you on many projects during your postdoc.

Luc, we shared a lot in the last few years at DiverSE, and your expertise in statistics helped a lot in going further in this thesis. Our discussions, about research or other topics, have always been a great pleasure and very enriching.

To all the members of the DiverSE team, too many to be all cited, you are all in my thoughts as you all marked me. I loved belonging to this group, in this constant sharing of knowledge, spanning of many domains, a true melting pot of reflection where curiosity makes science.

Mathieu, this internship you offered me opened a new path for that I thought unreachable, and this changed my life forever. Your enthusiasm is a true driving force for anyone working with you, never let that go!

Jean-Marc, you scientific skills have been recognized many times in many ways, but what impressed me the most was the degree you have been accessible to me during this thesis. We can feel your great experience of science through your concise and highly relevant critics, which were able to focus Mathieu's catching enthusiasm.

Thank you both for being an incredible duo of supervisors.

Jérôme, I follow you path a little by default, and if I forked to my own path, thank you for having been a model to follow.

Maman, you always bend over backward to offer your two big boys everything they needed, you can be proud of the result.

Pap, you offered us, Jérôme and me, the education you could never access, and you passed on us everything we needed to get the best out of it. Even if you will not be able to assist to the completion of this thesis, know that it is dedicated to you. Thank you for everything.

Remerciements

Mes premiers remerciements vont à vous, lecteur, car une grande leçon que j'ai tirée de cette thèse, est que la science n'a de valeur que si elle est transmise, et que lire cette thèse est aussi important que de l'écrire.

Je tiens à remercier l'Université de Rennes 1 et tous les professeurs pour leurs enseignements pendant mes 9 années d'études. En particulier, Professeur Rumen Andonov, vous méritez une place importante dans mes pensées, car de votre décision d'accueillir cet étudiant un peu perdu en ce jour de rentrée en L3 découle cette thèse. Merci de la confiance que vous m'avez accordée ce jour-là.

Merci aux membres du jury qui ont accepté de lire cette thèse et d'évaluer les travaux qui la composent. Les retours que j'ai eu ont servi à mon grand plaisir à améliorer le manuscrit qui est, à mon humble opinion, d'une bien meilleure qualité que quand il leur a été transmis.

Merci à tous les étudiants de master qui ont participé au projet TuxML, qui a été fondamental pour les études développées dans cette thèse.

Paul, tu m'as accompagné dans mes premiers pas vers le machine learning et t'es toujours assuré que j'avais ce dont j'avais besoin pour avancer, même quand je ne le demandais alors que j'en avais besoin, je t'en remercie.

Juliana, ton excellence scientifique m'a impressionné, et j'ai été très heureux de pouvoir travailler avec toi sur les nombreux projets au cours de ton postdoc.

Luc, on a beaucoup partagé au cours de ces dernières années chez DiverSE, et ton expertise en statistiques a largement participé à l'avancée de cette thèse. Les discussions que l'on a pu avoir, à propos de la recherche ou bien d'autres sujets, ont toujours été un grand plaisir et très enrichissantes.

A tous les membres de l'équipe DiverSE, trop nombreux pour être tous cités, mais je pense à chacun d'entre vous qui m'avez marqué. J'ai aimé faire partie de ce groupe, dans ce partage de connaissances permanent, couvrant de nombreux domaines, un véritable creuset de réflexion où la curiosité fait avancer la science.

Mathieu, ce stage que tu m'as offert a ouvert pour moi une voie que je pensais inaccessible, et cela a changé ma vie. Ton enthousiasme permanent est un véritable moteur pour quiconque travaille avec toi, ne perds jamais ça!

Jean-Marc, tes compétences scientifiques ont été maintes fois reconnues de bien des manières, mais ce qui m'a le plus impressionné est à quel point tu m'as été accessible au cours de cette thèse. Ta grande expérience de la science se ressent par tes critiques concises et pertinentes qui ont su canaliser l'enthousiasme communicatif de Mathieu.

Merci à vous deux pour avoir été un excellent duo de directeurs de thèse.

Jérôme, j'ai suivi ta voie un peu par défaut, et si aujourd'hui j'ai légèrement bifurqué pour aller sur ma propre voie, merci d'avoir été un modèle à suivre.

Maman, tu t'es toujours pliée en quatre pour fournir à tes deux grands garçons tout ce dont ils avaient besoin, tu peux être fière du résultat.

Papa, tu nous as offert, à Jérôme et à moi, l'éducation dont tu n'as jamais pu profiter, et tu nous as tout transmis pour que l'on puisse en tirer le meilleur. Même si tu ne pourras pas assister à l'achèvement de cette thèse, sache qu'elle t'est dédiée. Merci pour tout.

Table of Contents

Chapter 1

Introduction

Variability is a concept that affects everyone, everywhere, every day. It represents the set of all choices that we can make, or imposed by external factors. Moreover, each choice can influence others, and each combination of choices can lead to a different outcome. These choices are various, such as the different paths you will take to go to work. Each intersection is a choice and this succession of choices creates a unique path, with its own outcome. An outcome we could think of is the travel time, which is obviously dependant on the paths you take, but also on your speed, and is impacted by external factors (heavy traffic, roadworks, etc.) which could be avoided with a different choice of paths. However, it is often hard to predict how a choice can impact a specific outcome. Since most of the choices do not actually impact the outcome, or so little, they are mostly overlooked in order to avoid to be overburdened by decision making and the analysis work it requires. One could try every possible set of choices, but if we take our previous example, given the number of roads that exists, the set of all possible combinations, from the straightest to the most unexpected, is unfathomable. Such approach can only be considered in a context with very few choices.

In Software Engineering, variability has been introduced in order to create what we call Software Product Lines (SPL). The idea is to answer various client needs in a single system by offering them the possibility to configure that system with options to activate a part of the system that is needed. As many client can share the same needs, reusing a software system is a good way to avoid unnecessary costs. Variability brings out its own problems when used in Software Product Lines. Some options activated together might see their behavior change, as they can be interacting with each others. This implies that testing an option without considering others is not enough to ensure that the system works as expected. Beyond simply working well, options and combinations of options can affect performance or other non-functional properties. As Software Product Lines evolve, they become more complex, harder to understand and to profile.

Modelling the performance of a system has been the realm of experts for decades, but this time is over. With the constant increase in number of options, the rapid evolution of the systems, the complex interactions between options and the wide range of metrics to keep track of, the task became too overwhelming. In the last decade, the democratization of Machine Learning allowed it to set foot in SPL performance modelling. Numerous efforts have been made to derive performance models from a sample of performance measurements, sometimes able to produce more efficient and accurate models than experts can.

In these efforts, a certain routine appeared, particularly on the use of the same corpus of datasets for validation. Up to now, none of the subject systems used in the literature of SPL performance modeling reach more than a few dozens of options, 60 at best, while today's systems reach thousands of options such as the Linux kernel, or most of the web browsers on the market. Due to the combinatorial explosion of the interactions between options, the configuration space of such systems is colossal. With the slow but steady increase of the number of options in SPL in continuous development, persisting in elaborating new modelling techniques without challenging their ability to scale on the number of options is a mistake that should be corrected.

Beyond the warning for researchers on the scalability of the SPL performance modeling techniques, stemming from our observations of their limitations, this thesis offers solutions. We present the ability of tree-based learning techniques to handle efficiently the complexity of SPL for various tasks, from simple, yet scalable, performance modelling to transfer learning, by the way of feature selection and specialization.

Contributions

The first and foremost contribution of this thesis is the application of machine learning algorithms on a Software Product Line at a scale never reached before. The Linux kernel, with its 15,000 options, completely outscales all cases study on SPL of at best 60 options present in the literature. This contribution is focused on one research question: **How do state-of-the-art techniques perform on the Linux dataset?** The goal of this question is twofold, at first determine whether a technique *can* learn over such dataset, and then *how well* it can learn. By evaluating the accuracy of several state-of-the-art algorithms on predicting the binary size of the Linux kernel, we could determine the limitations of some techniques. Without surprise, linear-based algorithms give poor accuracy, as they cannot grasp the complexity caused by interactions between options.

Performance-influence model, while being developed specifically for SPL, cannot handle that many options despite the measures taken to reduce combinatorial explosion. In the end, we found out that Tree-based algorithms, as well as neural networks, were able to provide quite accurate models with reasonable resources in term of time and memory.

The second contribution is the Feature Ranking List, a list of features ranked by importance toward the prediction target, generated by an enhancement of Tree-based feature selection. We evaluated the relevance of this technique over multiple criterion spanning over multiple research question. The two first questions aims to compare the efficiency of learning techniques with and without tree-based feature selection: 1) **How accurate is the prediction model with and without tree-based feature selection?** and 2) **How much computational resources is saved with tree-based feature selection?** The desired and most known effect of feature selection in general is a major speed up of the training time, cutting it down by 5 for Gradient Boosting Tree and by 20 for Random Forests. More unexpected, we also report noticeable accuracy improvement over most previously considered algorithms. The next research question is 3) **How stable if the Feature Ranking List?** As the technique needs to be reliable to be convincing to use, we inquired about its stability, or its ability to repeatedly produce the same results. This allowed us to first discover a low stability problem, and then enhance the technique for better stability. At last the final question is 4) **How do feature ranking list, as computed by tree-based feature selection, relate to Linux knowledge?** By this question, we compared what is known about the Linux kernel, by experts or through documentation, with the insight synthesized by the Feature Ranking List. When presented to experts, the Feature Ranking List was deemed accurate. However, there is a high discrepancy when confronted to the documentation, mostly due to the very low number of options explicitly documented as impacting the Linux kernel size.

The third contribution is the enhancement of automated performance specialization and its evaluation over various SPL including Linux. Performance specialization is a process which consists in adding constraints over an SPLs in order to meet some user defined performance threshold, to assist him during the configuration of the software. It is possible to automate this using machine learning, and more specifically Decision Tree Classifiers predicting if a configuration is acceptable with regard to the performance threshold or not, from which we can extract rules that can be applied as constraints over the target SPL. Our enhancements are on two sides. The first is the application of feature selection based on Feature Ranking List, which improved both accuracy and training time, even on an SPL with only a few options. The second is the consideration of Decision Tree *Regressors*, producing a similar model when it comes to rule extraction, but with the advantage of being aware of the performance, while this dimension is reduced to a simple boolean in a classification process. However, both techniques lack an important information, the Classifier lacks information about the performance, while the Regressor lacks information about the performance threshold. We developed a novel technique, based on a Decision Tree Regressor, where we simulate a gap in the performance distribution to make the algorithm aware of the threshold with regard to the loss function. Our two first research questions are 1) **What is the accuracy and cost of learning strategies?** and 2) **What is the best learning strategy?** These inquire about the viability of the approaches and to compare them. In the end, all three techniques have their own strengths and weaknesses, with their own use cases, and should be considered as complementary. The next question is 3) **What are the effects of tree-based feature selection on the accuracy of performance specialization?** By applying tree-based feature selection, the specialization could achieve slightly more accurate results, but its most important effect was revealed by the last question: 4) **What is the time cost of the 6 learning strategies to predict performance of a configurable software system?**

The last contribution opens the door toward a more sustainable usage of machine learning for Software Product Lines. In this domain, little to no attention has been given to the accuracy of models on different versions of a SPL, so this lead to our first research question: **To what extent does Linux evolution degrade the accuracy of a binary size prediction model trained on a specific version?** We took a very accurate model from the works of our first contribution and evaluated it over later versions and observed two major problems. The first is the mismatching options, as between two versions, some options appear and some others disappear, creating a shift in the configuration space. The second is the drop in accuracy when the model is used on another version that the one it was trained on. The cost to train the first model was very important, in terms of configuration measurements and training effort, and such expense is not reasonable when you consider that there is a new version every two or three months. To tackle these two problems, we created Evolution-Aware Model Shifting, a transfer learning technique suitable for the Linux kernel scale and evolution pace. It leverages the original and deprecated model, and at a fraction of original cost, adapts it to the new versions. The second research question challenges this contribution: **What is the accuracy of our evolution-aware model shifting (tEAMS) compared to learning from scratch and other transfer**

learning techniques? Our results show very good accuracy over three years of evolution, and better accuracy than learning from scratch on the same dataset.

Outline

In Chapter 2, we present the state-of-the-art on Software Product Lines and Software Variability concerns, as well as machine learning and its uses in Software Engineering. We also present the Linux kernel and its particularities given how unique it is among other SPL.

In Chapter 3, we present our evaluation of the state-of-the-art machine learning techniques on the problem of predicting Linux kernel binary size. We show how we set up the evaluation, and present the dataset we used for the Linux kernel and then present the results.

In Chapter 4, we propose the Feature Ranking List, and how to use it to perform feature selection even at the scale of Linux. We develop the concept and present how we used Tree-based feature selection to obtain the Feature Ranking List and how we enhance it to increase its stability. By describing the limitations of the state-of-the-art, we show that this technique is the only reasonable way to deal with highly configurable systems at this time. We show the results of using this feature selection step with the algorithm previously explored, on both accuracy and training time. We also compare the Feature Ranking List with expert knowledge and documentation. At last, we investigate the stability of the Feature Ranking List, and explore its influence on accuracy for models using it.

In Chapter 5, we introduce new ways to perform automated performance specialization. We details the concept of performance specialization, and how it was automated. We propose two other ways to do it, and evaluate all approaches on Linux and other popular SPL from the state-of-the-art.

In Chapter 6, we explore the accuracy of a model over different versions of Linux kernel to measure its degradation. We present evolution-aware model shifting, a transfer learning technique compatible with software evolution concerns, and compare the accuracy of the resulting models to learning from scratch.

At the end, the conclusion will offer a wrap up of the contributions and provide the different perspectives the contributions opened in the field of machine learning applied to colossal configuration spaces.

Bibliographical notes

The research presented in this thesis reuses and extends publications of the author. The list of peer-reviewed papers is given below.

Journals

1. [91] Hugo Martin, Mathieu Acher, Juliana Alves Pereira, Luc Lesoil, Djamel Eddine Khelladi and Jean-Marc Jézéquel. (2021). Transfer Learning Across Variants and Versions: The Case of Linux Kernel Size, *accepted at IEEE Transactions on Software Engineering* `https://hal.inria.fr/hal-03358817`
2. [114] Juliana Alves Pereira, Hugo Martin, Mathieu Acher, Jean-Marc Jézéquel, Goetz Botterweck, Anthony Ventresque. Learning Software Configuration Spaces: A Systematic Literature Review. Journal of Systems and Software, Elsevier, 2021, ⟨10.1016/j.jss.2021.111044⟩ `https://hal.inria.fr/hal-02148791`

Conferences

3. [92] Hugo Martin, Mathieu Acher, Juliana Alves Pereira, and Jean-Marc Jézéquel. (2021). A comparison of performance specialization learning for configurable systems. *in:Proceedings of the 25th ACM International Systems and SoftwareProduct Line Conference - Volume A*, SPLC '21, Leicester, United Kingdom: Association for Computing Machinery, 2021, pp. 46–57, (Best paper Award) `https://hal.archives-ouvertes.fr/hal-03335263`
4. [8] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, and Jean-Marc Jézéquel. 2020. Sampling Effect on Performance Prediction of Configurable Systems: A Case Study. *In Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE '20).* Association for Computing Machinery, New York, NY, USA, 277–288. (Best paper Award) `https://hal.inria.fr/hal-02356290v2`

Reproducible Science: Software, Data, and Tools

As part of the PhD thesis, we contributed to build datasets, to analyse data for reporting quantitative and qualitative insights, and to develop tools. The following online resources reference tools, data, script, documentation, models, etc. developed as part of the conducted studies:

— The Linux dataset: `https://zenodo.org/record/4943884`

- A Docker image to train models and explore hyperparameters: `https://github.com/HugoJPMartin/rf-analysis`
- Artifacts for Automated Performance Specialization (ACM badge artifacts at SPLC): `https://github.com/HugoJPMartin/SPLC2021`
- Artifacts for Transfer Learning for Linux: `https://github.com/HugoJPMartin/transfer-linux`
- kpredict, a tool for predicting kernel binary size from .config files: `https://github.com/HugoJPMartin/kpredict`

Chapter 2

Background

In this chapter, we present the background knowledge needed to understand this thesis, and the state-of-the-art of the domains the thesis belongs to. At first, we detail the background in Software Product Lines, and the limitations that required the use of machine learning, which we introduce in the following section. Then, we develop the various efforts made to join Software Product Lines and Machine Learning in the literature. Next we describe Linux, a good example of a system with a colossal configuration space. At the end, we summarize the weaknesses in the state-of-the-art that this thesis aims to address.

2.1 Software Product Lines

Production line is a concept said to be created by Henry Ford in the early 1900's, and while this claim can be debatable, he brought massive improvements to assembly lines and standardization. This allowed mass production of cheap and easy to maintain cars, and paired with socioeconomic developments, it undeniably led to the democratization of cars. However, this achievement was at the cost of customization as Ford pointed it out in his famous quote: "Any customer can have a car painted any color that he wants so long as it is black".

This highlights divergence between expensive but mostly customized cars, and the cheap and massively produced cars but completely standardized ones. These two concepts exist in all production domains as of today, though in a more nuanced way, each one being an extremity of a spectrum. Mass production evolved to include the concept of mass customization, which allows answering specific customers' needs, yet still with the large-scale production requirements. This can be made through the use of *options* a customer can select, such as choosing a color other than black, electrical windows, or even a different engine. All in all, the same model of car, the core, can articulate itself in various end products through the combination of multiple available options.

In Software Engineering, the same concept of product lines has been applied to reduce cost and time of development. Clemens *et al.* [28] define a Software Product Line (SPL) as follow:

> A **software product line** is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

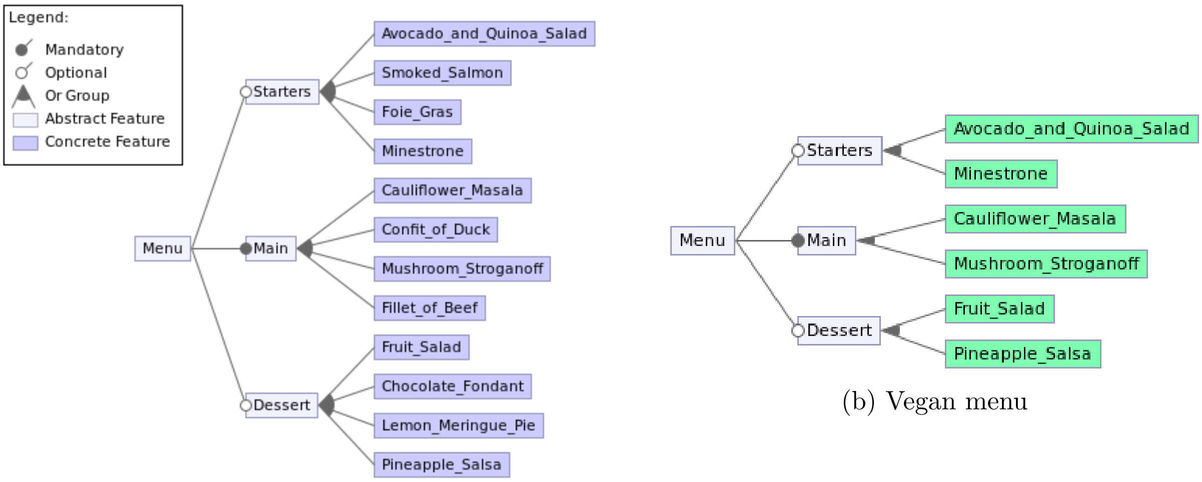
The most particular aspect of SPL that makes it stand out from other product line engineering is the ability to reuse software almost at will. Krueger [85] presents software reuse as "the process of creating a software systems from existing software rather than building software systems from scratch". Even more specifically, with the help of customization patterns like options, mentioned with cars, customization is also applied in Software Product Lines. It then became possible to create a software system meeting very specific needs without having to write a single line of code. Svahnberg *et al.* [140] define this as software variability:

> **Software variability** is the ability of a software system or artifact to be efficiently extended, changed, customized or configured for use in a particular context.

Nowadays, almost every software system is configurable in some way, and the number of options tends to grow bigger over time [31]. The good side of this phenomenon is that the number of possible configurations grows even faster due to combinatorial explosion, leading to an increased chance that some of them would meet customers' requirements. The bad side of this phenomenon is that the number of possible configurations grows even faster due to combinatorial explosion, leading to a massive increase in effort to ensure the proper functioning of every options with one another or to know the behavior of some options in a specific configuration.

Indeed, not all possible combinations of options are valid, due to constraints and dependencies. These represent the situations when some options do not work with one another, such as categorical options (SLOB, SLAB and SLUB are three different memory management options, only one should be picked at a time), or when an options requires another one to work. In order to model these, the concept of *feature models* has been developed [72]. Over the years, it has been improved, with support for cardinalities [32, 137], or with automated reasoning based on translation to propositional formulas and SAT solving [13].

Figure 2.1a gives an example of a feature model, applied on a well-known context: a



(a) Full menu

(b) Vegan menu

Figure 2.1 – Example of feature model representing restaurant menu.

restaurant menu. In this case, each dish is a feature, and without constraint, 12 dishes would mean 4095 different combinations of possible dishes. However, the feature model expresses different conditions to make a combination valid. First, all dishes are grouped into a category: Starters, Main and Dessert. Each of these categories is an "Or Group", meaning only one dish of each category can be taken. Then, we have optional categories, as in this menu, only the Main category is mandatory. With all these constraints, the number of valid menus drops down to 100, in other words, only 2% of the combinations are valid. This demonstrates the need of having a clearly defined map of the valid combinations to avoid mistakes during the selection of the features.

2.1.1 The Curse of Configurability

Configurability is a great opportunity for the developers, which comes at some cost in testing and maintenance, as expressed earlier. However, from a user point of view, being able to configure a system is more of a curse than a blessing. Xu *et al.* [163] report that most options are used by only a small fraction of the users, and that actually less than 10% of the options are used by more than 90% of the users. The growing number of options depicted in Figure 2.2 from the same study, with widely used systems having hundreds of options, only adds complexity at configuration time without effectively adding the desired flexibility as most options are not used.

Moreover, if unused options are set with a default value, it is shown that it is rarely

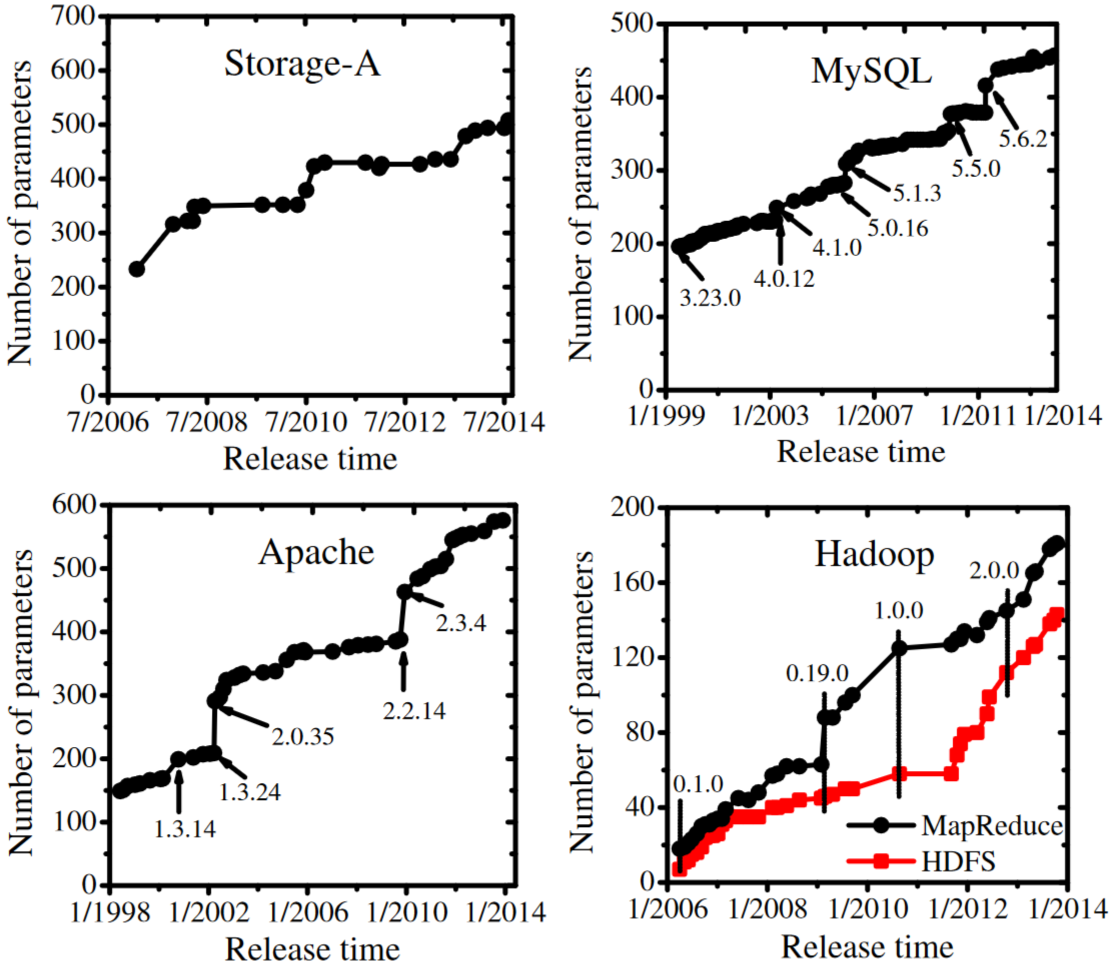


Figure 2.2 – The increasing number of configuration parameters with software evolution (credit to Xu *et al.* [163])

a value well-suited for the user's needs, meaning that every unused options would likely bloat the system and decrease its performance regarding user's needs.

One way to alleviate the variability burden on a user is specialization.

2.1.2 Specialization

In Software Engineering, we can find the concept of specialization in many shapes. Maybe the most known example is in the class diagrams of Unified Modeling Languages with the inheritance relationship, where specialization and generalization are the two directions to interpret this relationship.

The work of Consel *et al.* [29, 94, 126] on program specialization shows the concerns that generic programs can be less efficient than specialized ones, and the program specialization aims to strip down parts of the program that are useless for specific tasks but that might impact the performance.

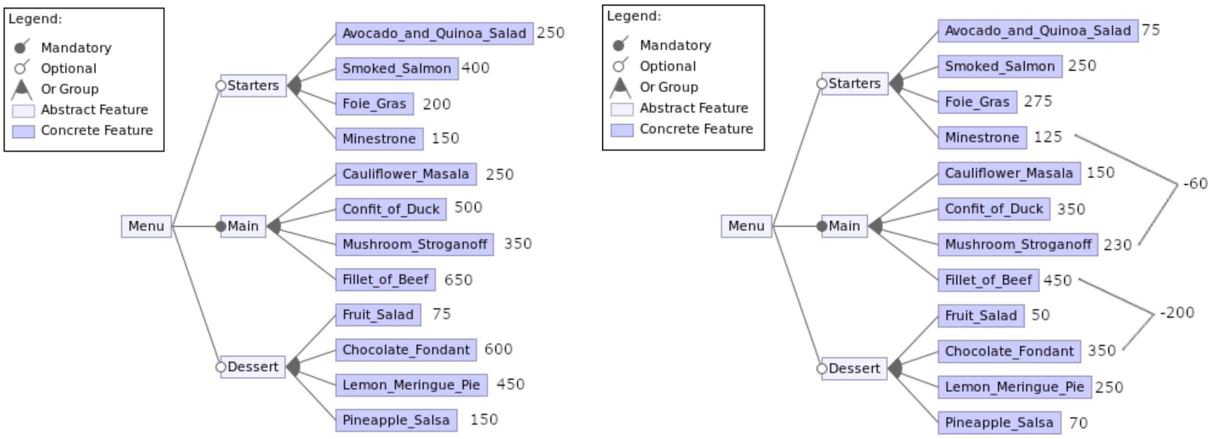
Czarnecki *et al.* [33, 34] enhance features models by porting specialization, or multi-stage configuration, into them. Usually, the configuration of a system happens in one go, all the desired features are selected, in accordance with the constraints. However, it is possible to consider other cases, such as when two or more parties are involved in the configuration of the system, each having their own needs in term of features. For instance, a software vendor could support only a subset of the features of the software system they sell in order to reduce the cost for testing, and provide the client with a specialized system, that is still configurable in the sense that there would be more than one configuration available, but always within the boundaries of the vendor support. Other constraints are possible in the limiting of the feature model expressiveness, such as forcing the activation of a feature when another is activated.

If we take the feature model in Figure 2.1a, it can be difficult to determine the vegan options in the menu. By using specialization, we can provide the feature model in Figure 2.1b, which only presents the vegan options. In this case, the client does not have to worry about choosing a wrong dish according to his needs.

Performance Specialization While the use cases of specialization proposed by Czarnecki *et al.* target directly features, it is possible to leverage the same principles of specialization for other, less direct and usually more complex targets, such as performance and properties (speed, energy consumption, etc.). In this case, the specialization works around a given performance threshold, and all configurations of the specialized systems should meet that threshold. To achieve that, further constraints have to be created in the feature model, based on expert knowledge or direct measurements.

Going further with our example of restaurant menu, we annotated each dish with its calories value in Figure 2.3a, which would allow to filter out menu variants with too many calories, let's say 1200 calories. The main difference with the vegan specialization is that the constraints on a dish will depend on other dishes, such as the constraint that the Fillet of Beef (650 calories) cannot be taken with the Chocolate Fondant (600) as this pair gives 1250 calories. In this case, a specialized menu would prune the available dishes along with the client selecting his own dishes to always respect the given threshold.

However, the calorie problem is an additive problem, meaning that you just have to



(a) Menu with calories

(b) Menu with Carbon Footprint

Figure 2.3 – Example of feature models representing restaurant menu with performance metrics.

add all dishes value from the menu to know the calories value of the menu. In that case, measuring the calorie for each dish separately gives enough information. The Figure 2.3b takes on a more complex problem and is annotated with the carbon footprint of each dish. In that example, the carbon footprint represent the greenhouse gas emission from cultivating or raising, transporting and transforming the elements of the dish. If we consider that the Fillet of Beef is cooked in the same oven as the Chocolate Fondant, actively reducing the energy to cook them independently, it explains why we denote a reduction of 200 of the carbon footprint when both are taken in the same menu. The difference with the previous example is that we consider interactions between the dishes and simply measuring the footprint of each dish separately as we do with calories would be wrong. For this problem, we need to know which dish interacts with each other and measure these combinations specifically.

In Software Product Lines, the same problems arise. Some options of the systems may interact with each other, and affect its performance or its properties. Identifying interacting options and the degree of impact on properties is crucial when it comes to predicting them.

However, measurements can be costly, and in case of large configuration spaces, the number of different combinations to measure can be very high, effectively multiplying the costs. Exhaustive approaches become more and more infeasible as configuration spaces grow. Expert knowledge is also a tool that does not scale with the increasing complexity of the systems. The more options there are in a system, the harder it is for an expert to identify the rules needed for specialization.

The growing complexity of Software Product Lines makes the solutions based on expert knowledge or exhaustive exploration less and less efficient with the regard to the accuracy or the cost. This has led the SPL community to investigate machine learning as an alternative approach.

2.2 Machine Learning for SPL

2.2.1 Machine Learning

Mitchell [96] defines learning as follow:

> A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E

The example given by Mitchell with checkers presents T (task) as the ability to play checkers, P (performance) as the win ratio, and E (experience) as playing a game of checkers. The field of Machine Learning aims to develop and enhance algorithms that enable computers to learn.

Machine learning algorithms come in all sorts of forms, all with their specific usage, features and complexity. James *et al.* [62] provide an extensive introduction to machine learning and first separate algorithms by their purpose. Unsupervised learning aims to find similarities between examples and create clusters; we do not use these techniques in this thesis. For *supervised learning*, the learned model is a function that links the features to a label, such as a list of symptoms to a specific illness, or a list of options to the price of a car. In the first case, it is specifically called *classification* as the label is a discrete value from a set of classes (the known illnesses), while the latter is called *regression*, as the label is a continuous value (the price).

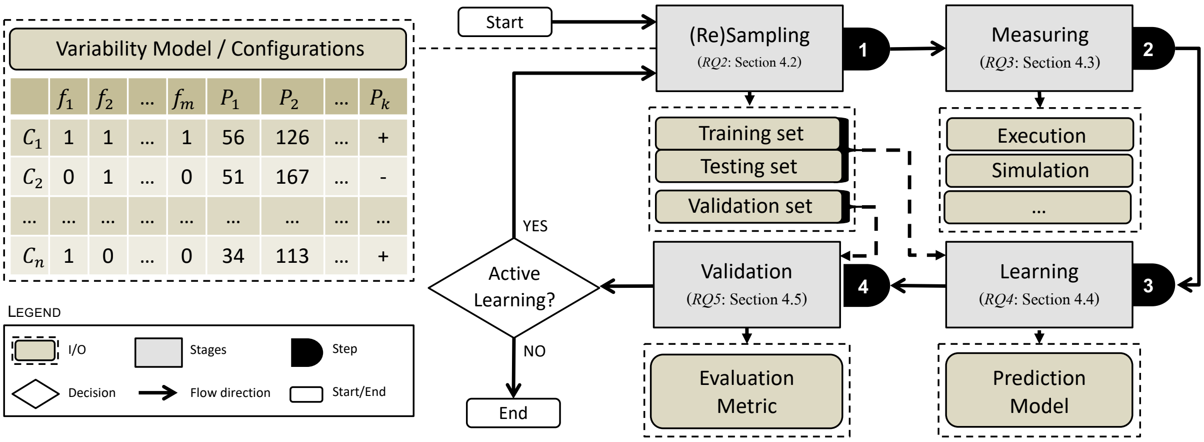
For supervised learning, we can mention different algorithms:

- **Linear Regression**: the most simple approach of machine learning, used for predicting quantitative value. The resulting model takes the form of $y = ax + b$, with x being a feature, a and b the values determined by the algorithm. Multiple Linear Regression combine these models to handle more features such as $y = a_1x_1 + a_2x_2 + b$
- **Logistic Regression**: the pendant of Linear Regression but for qualitative values,

hence a classification method, despite the name. The model is similar, with the difference that the values determined by the algorithm are probabilities.

- **Decision Trees (DT)**: By recursively splitting the dataset, this type of algorithm obtains a set of decision rules that segment the prediction space into a number of simple regions. This set of rules can be organized into a Decision Tree, hence the name of the algorithm. There are actually multiple algorithms that can create a Decision Tree, and while they can all handle regression and classification tasks, they can differ in their limitations regarding the number of branches at each node (only two branches means a Binary Decision Tree) or in whether they can handle categorical features instead of only numerical features. Decision Trees can be used in ensembles in different ways:
  - **Bagging**: By averaging the result of multiple Decision Trees trained on different subsets of the training set, it is possible to reduce the variance of the model.
  - **Random Forest (RF)**: Similar to bagging, RF also trains the Decision Trees on different subset of features, allowing for better use of moderately correlated features when there are strongly correlated features in the same set. It yields generally better results and lower variance.
  - **Boosting**: This technique creates Decision Trees sequentially, always with regard to the previous iteration's errors in order to fix them, hence resulting in better accuracy. One of the most known algorithm implementing this process is the Gradient Boosting Tree (GBT)
- **Support Vector Machines (SVM)**: These algorithms are based on the discovery of an hyperplane that separates different classes. As is, it is only able to handle linear separation, but with the help of the kernel trick to warp the problem space, non-linear problems can be solved. They can handle both regression and classification problems.
- **Neural Network**: Composed of interconnected neurons, individual nodes holding a function weighting inputs into an output, its flexibility allows to solve a broad range of problems, with the help of various architectures, the layout of neuron connections in the network. It is arguably the best type of algorithm developed in the past years, at great expense of computing power and number of examples needed.

This is only an excerpt of all existing learning algorithms, and they all come with variations for handling specific problems.



	f_1	f_2	...	f_m	P_1	P_2	...	P_k
C_1	1	1	...	1	56	126	...	+
C_2	0	1	...	0	51	167	...	-
...	...	...	...	...	...	...	...	...
C_n	1	0	...	0	34	113	...	+

Figure 2.4 – Machine Learning stages for exploration of SPL configuration spaces

Interpretability. One aspect of machine learning models is its human readability. A white-box model can be seamlessly understood by a human when it makes a prediction, such as by following the rules created by a Decision Tree, or calculating the result of a Linear Regression. On the other hand, a black-box model, such as SVM or Neural Networks, is abstruse and its predictions cannot be explained. In between, we can consider other models such as the ensembles of Decision Trees (Random Forest, Gradient Boosting Tree, etc.) which cannot be completely explained, but can still give insight on how they solve a problem, with the help of metrics like feature importance, which weights every feature according to how useful it is to predict an outcome.

2.2.2 Machine Learning for SPL

Pereira *et al.* [114] explore the different usages of machine learning in the context of SPL and identifies four big steps in the usual processes: Sampling, Measuring, Learning, and Validating (see Figure 2.4). While the state-of-the-art mostly follows the usual machine learning process, for SPL, the process of data acquisition is here split up between Sampling and Measuring. The main reason is that these two concepts have already been studied outside of the machine learning process and have their own existing state-of-the-art.

Sampling. In order to match the particularities of SPL and their constrained environment, several sampling techniques have been devised. The most basic solution is the *random sampling* that has been used extensively [50, 51, 67, 69, 101, 106, 131, 143], but has raised many concerns about its feasibility or its uniformity [117].

Other solutions rely on heuristics. By using expert knowledge or documentation, it is possible to find the features to focus on and the feature interactions to care of [133, 134].

Coverage sampling systematically samples all combinations of feature of n order. Pair-wise sampling (order $n = 2$) assumes feature interaction between every possible pair of features [131, 132, 133, 134]. Other works consider 3-wise sampling [71, 89, 124] or even higher order [165, 166].

Pereira *et al.* [8] compare 6 different sampling techniques while varying input workload on subject systems and with two different performance metrics. The goal is to determine the impact of workload and metrics on the accuracy of a learning method, when most of the studies comparing sampling techniques disregard these parameters. It appears that both workload and metrics greatly change the efficiency of all sampling techniques, and that random sampling is the most solid technique overall.

Property prediction. Numerous works have investigated the idea of learning non-functional properties or various performance indicators of configurable systems [48, 67, 69, 71, 81, 101, 102, 119, 124, 127, 131, 142, 143, 150, 151, 168], spanning different application domains, such as compression libraries, database systems, or video encoding. The goal is to create a model able to predict the behavior of a configuration without relying on an often costly exhaustive measurement step, but only on a sample of measurements that are generalized.

Guo *et al.* [51] explore the usage of CART (Classification and Regression Tree), an implementation of the Decision Tree concept, for performance prediction. They measured the accuracy of the model trained either on random sampling or on pair-wise sampling and obtain good results.

The work of Siegmund *et al.* [131, 132, 133, 134] revolves around the *performance-influence model.* This model takes the form of a Multiple Linear Regression, as seen in Section 2.2.1. During the construction of the model, the interactions between options are encoded as features, meaning that a system with two options a and b will have a *performance-influence model* of this form : $y = a \cdot w_1 + b \cdot w_2 + a \cdot b \cdot w_3$ with w_1, w_2 and w_3 the weight for each feature determined during the learning process. This format is very interpretable, as each option and combination of options is tied to a weight, a positive or negative number, allowing to easily understand the influence of each element in the formula. As is however, the model would be impractical due to combinatorial explosion.

The solution to that problem evolved over the different iterations of the algorithms to create the models. At first, they used expert knowledge to identify options that interacted with each other. Then they moved to a more data-driven approach with the help of heuristics: only the influential options are combined into features for the model. This is a process of feature selection which is described later in Section 2.2.4. Numerous works [39, 40, 63] then used and enhanced this model on deeper problems.

Optimization. The problem of optimizing performance of configurable systems (*i.e.,* finding the best configuration) has attracted some attention [82, 100, 101, 114]. Despite a very similar approach as performance prediction, the details on how to reach that goal can be almost opposed as shown by Nair*et al.* [102] : some models bad at predicting accurately performance can be used for optimization.

White-box approaches. By pairing learning techniques with static data-flow analysis or profiling to guide the performance analysis, white-box approaches [152, 153, 156] have been proposed to inspect the implementation of a configurable system. They are typically used to help understand options and their interactions in a fine-grained way and can provide valuable insights.

Automated Performance Specialization. For the same reasons that performance modelling is error-prone for large configurable systems if only based on expert knowledge, and almost impossible by systematic measurement, performance specialization is a complex task given how hard it is to find the right constraints.

Temple *et al.* [141, 143] leverage a Decision Tree Classifier and its decision rules to infer the right constraints to obtain a specialized feature model.

Performance specialization is applicable to several domains. Temple *et al.* [141, 143] explore the domain of a video generator. To improve the learning process, Temple *et al.* [141] specifically target low confidence areas for sampling. The authors use an adversarial learning technique, called evasion attack, after a classifier is trained with a support vector machine. Beyond a video generator, Temple *et al.* [142] explore also the domains of web server, compiler, database system, and image processing. Acher *et al.* [6] explore the domain of creating a specialized document (such as a research paper or a resume). Amand *et al.* [9] applied learning techniques in the 3D printing domain.

There are several works in the faulty specialization context [11, 45, 46, 84, 116, 122, 143, 165]. These works explore several domains, such as software-intensive systems, combinatorial models, and 3D printing. However, these works focus on mining rules for avoiding invalid configurations, without having performance in mind.

2.2.3 Transfer Learning

West [159] gives a definition of transfer learning (also called inductive transfer):

> **Inductive transfer** refers to any algorithmic process by which structure or knowledge derived from a learning problem is used to enhance learning on a related problem.

Transfer learning is subject to intensive research in many domains (*e.g.,* image processing, natural language processing) [110, 158, 167]. Different kinds of data, assumptions, and tasks are considered. Some of the most complex tasks handled by machine learning rely on models that can take months to train, and consequently a huge amount of resources. Transfer learning has been developed to heavily reuse models with slight modification to specialize to a user needs, effectively avoiding the costs of training a new model from scratch. Negative transfer is an important concern [167] in this context: divergence measures between source and target have been proposed as well as dedicated algorithms [35, 110, 158, 164].

Transfer Learning for SPL Transfer learning has attracted interest with the promise to save resources by reusing the knowledge of performance under different settings (*e.g.*, hardware settings) [24, 61, 64, 66, 68, 78, 83, 103, 149, 150, 155].

Defect prediction Defect prediction leverages transfer learning to identify defects in projects lacking data by reusing models created from other projects with more data [25, 26, 88, 104, 154]. Particularly, they handle the problem of heterogeneous context. The metrics used to predict the defects differ from one project to another. To use information from a project in another, this heterogeneity needs to be taken in account.

2.2.4 Feature Selection

Feature selection is a terminology used both in Product Line Engineering and Machine Learning with different significations. In Product Line Engineering, the feature selection involves setting a value to an option (also named feature), while in Machine Learning, it is the consideration of only a subset of the available features when learning a model.

In this part, as well as in the remainder of this thesis, feature selection uses only its Machine Learning signification. Chandrashekar *et al.* [23] provides an extensive review on feature selection. They describe it as helping "in understanding data, reducing computation requirement, reducing the effect of curse of dimensionality and improving the predictor performance". They distinguish three main categories of feature selection: filter, wrapper and embedded methods. A filter method only works as a preprocessing step by ranking features with regard to a relevance criterion and then selecting the highest ones. Wrapper methods rely on the prediction model, and by testing subsets of the features and comparing accuracy of the different subset, they can identify their best subset. This method is highly sensitive to the number of features as the number of potential subsets is exponential. In embedded methods, the feature selection is directly part of the model building phase, such as with decision trees where at each split, a feature is selected.

Sigmund *et al.* [131] rely on a forward backward feature selection algorithm. As a wrapper method, it compares subsets of features by measuring the accuracy of the model trained on each subset. The subsets are created by adding features one by one, in the forward step, so that only individual features with positive impact on the accuracy are kept. Then, the backward step consists in taking out features one by one, and putting them in again if there is an accuracy loss. This is to avoid bias due to the order of appearance of the features.

This usage of feature selection, backed up by heuristics, helps to reduce the combinatorial explosion coming from the increasing number of options in software systems. However, this does not ensure the scalability of the approach when dealing with thousands of options. In fact, there is no study applying machine learning on systems with more than a few dozens of options, and this lack of information can put all the state-of-the-art techniques at risk if they cannot deal with more complex systems.

The Linux Kernel is a perfect example of a system with a colossal configuration space with more than 15.000 options to date. We will thus use it as a subject to inquire about the scalability of state-of-the-art techniques.

2.3 The Linux Kernel

2.3.1 Linux, options, and configurations

The Linux Kernel is a free and open-source operating system, that is the most used operating system in the world. While it is almost absent in the desktop market, it is dominant in every other markets. More than 95% of the servers, the machines hosting Internet websites, run on Linux. 85% of all smartphones run on Linux. All of the Top 500 supercomputers run on Linux [44]. And if the Linux kernel is omnipresent, it is partly because it is a highly-configurable system which can adapt to a broad range of needs and environments.

Thousands of configuration options are available on different architectures (*e.g.,* x86, amd64, arm) and documented in several Kconfig files. For the x86 architecture and the version 4.13.3, developers can use 12,797 options to tailor (non-)functional needs of a particular use (*e.g.,* embedded system development). The majority of options has either boolean values (*'y'* or *'n'* for activating/deactivating an option) or tri-state values (*'y'*, *'n'*, and *'m'* for activating an option as a module). There are also numerical or string values. Options may have default values. Because of cross-cutting constraints, not all combinations of options will result in successful build.

```
config LOCK_STAT
        bool "Lock usage statistics"
        depends on STACKTRACE_SUPPORT && LOCKDEP_SUPPORT ...
        select LOCKDEP
        select DEBUG_SPINLOCK ...
        default n
        help
         This feature enables tracking lock contention points. For
         more details, see Documentation/locking/lockstat.txt This
         also enables lock events required by "perf lock", subcommand
         of perf. If you want to use "perf lock", you also need to
         turn on CONFIG_EVENT_TRACING. CONFIG_LOCK_STAT defines
         "contended" and "acquired" lock events. (CONFIG_LOCKDEP
         defines "acquire" and "release" events.)
```

Figure 2.5 – Option LOCK_STAT (excerpt)

For example, Figure 2.5 depicts the KConfig file that describes the *LOCK_STAT* option. This option has several direct dependencies (*e.g., LOCKDEP_SUPPORT*). When selected, this option activates several other options such as *LOCKDEP*. By default, this option is not selected (*'n'*). The documentation of this option (including the help part) does not give any indication about its impact on any property such as the kernel size. In this type of documentation, the impact of options on the different properties is often missing, and always expressed in a fuzzy way.

Linux kernel builders set values to options (*e.g.,* through a configurator [160]) and obtain a so-called .config file.

We consider that a *configuration* is an assignment of a value to each option, either by the user, or automatically with a default value. Based on a configuration, the build process of a Linux kernel can start and involves different layers, tools, and languages (C, CPP, gcc, GnuMake and Kconfig).

KConfig language

The KConfig files, which index all features and their dependencies, are used to express the feature model, and are written in the KConfig language [73] specifically developed for the Linux kernel. While the language suits the needs of the kernel developers, it appears that it is not very compatible with tools and methods developed by research in SPL and variability for many reasons. The semantics of the languages are reported not entirely figured out yet [14, 15, 75, 129]. Despite many initiatives and works, KConfig does not integrate a SAT solver [74], and the various works around it such as Undertaker [37, 147] tend to only translate the KConfig information towards another compatible format, at the price of certain loss of integrity and mistakes [128].

2.3.2 Kernel binary size

Use cases and scenarios

There are numerous use-cases for tuning options related to binary size of the kernel in particular [56, 107]:

— the kernel should run on very small systems (IoT) or old machines with limited resources;

— Linux can be used as the primary bootloader. The size requirements on the first-stage bootloader are more stringent than for a traditional running operating system;

— size reduction can improve flash lifetime, spare RAM and maximize performance;

— a supercomputing program may want to run at high performance entirely within the L2 cache of the processor. If the combination of kernel and program is small enough, it can avoid accessing main memory entirely;

— the kernel should boot faster and consume less energy: though there is no empirical evidence for how kernel size relates to other non-functional properties (*e.g.,* energy consumption), practitioners tend to follow the hypothesis that the higher the size, the higher the energy consumption.

— cloud providers can optimize instances of Linux kernels w.r.t. size;

— in terms of security, the attack surface can be reduced when optional parts are not really needed.

When configuring a kernel, binary size is usually neither the only concern nor the ultimate goal. The minimization of the kernel size has no interest if the kernel is unable to boot on a specific device. Size is rather part of a suitable tradeoff between hardware constraints, functional requirements, and other non-functional concerns (*e.g.,* security). The presence of logical constraints and subtle interactions between options further complicates the task.

Community attempts

There exists (informal) initiatives in the Linux community that are dealing with kernel sizes.

The Wiki `https://elinux.org/Kernel_Size_Tuning_Guide` provides guidelines to configure the kernel and points out numerous important options related to size. However the page is no longer actively maintained since 2011. Tim Bird (Sony) presented "Advanced size optimization of the Linux kernel" in 2013 [17].

Josh Triplett (Intel) introduced tinyconfig at Linux Plumbers Conference 2014 ("Linux Kernel Tinification" [146]) and described motivating use-cases.

The leitmotiv is to leave maximum configuration room for useful functionality while exploiting opportunities to make the kernel as small as possible. It led to the creation of the project `http://tiny.wiki.kernel.org`. The last modifications were made 5 years ago on Linux versions 3.X `https://git.kernel.org/pub/scm/linux/kernel/git/josh/linux.git/`. Pieter Smith (Philips) gave a talk about "Linux in a Lightbulb: How Far Are We on Tinification (2015)". Michael Opdenacker (Bootlin) described the state of Linux kernel size in 2018. According to these experts, techniques for size reduction are broad and related to link-time optimization, compilers, file systems, strippers, *etc.* In many cases, a key challenge is that configuration options are spread out over different files of the code base, possibly across subsystems [1, 2, 99, 16, 95, 112].

Tiny Kernel Configuration

The Linux community has introduced the command make tinyconfig to produce one of the smallest possible kernels. Though a user cannot use such a kernel to boot a Linux system, it can be used as a starting point for *e.g.,* embedded systems in efforts to reduce the kernel size. Technically, it starts from allnoconfig that generates a kernel configuration with as many options as possible set to 'n' values (*i.e.,* this option must not be used). allnoconfig works as follows: following the order of options in the KConfig files, a greedy algorithm iteratively sets 'n' values to options. Due to numerous constraints among options and throughout the process, other dependent options may be set to 'y' values.

In a second and final step, options' values of Figure 2.6 override the values originally set by allnoconfig. For example, the value of `CONFIG_CC_OPTIMIZE_FOR_SIZE` can be set to 'y' (overriding its original value 'n').

```
# CONFIG_CC_OPTIMIZE_FOR_PERFORMANCE is not set
CONFIG_CC_OPTIMIZE_FOR_SIZE=y
# CONFIG_KERNEL_GZIP is not set
# CONFIG_KERNEL_BZIP2 is not set
# CONFIG_KERNEL_LZMA is not set
CONFIG_KERNEL_XZ=y
# CONFIG_KERNEL_LZO is not set
# CONFIG_KERNEL_LZ4 is not set
CONFIG_OPTIMIZE_INLINING=y
# CONFIG_SLAB is not set
# CONFIG_SLUB is not set
CONFIG_SLOB=y
CONFIG_NOHIGHMEM=y
# CONFIG_HIGHMEM4G is not set
# CONFIG_HIGHMEM64G is not set
```

Figure 2.6 – Pre-set values of tinyconfig for the X86_64 architecture. As far as possible, other options are set to 'n' values.

Experts of the Linux project have specified options of Figure 2.6 based on their supposed influence on size. Specifically, `CONFIG_CC_OPTIMIZE_FOR_SIZE` calls the compiler with the $-Os$ flag instead of $-O2$ (as with `CONFIG_CC_OPTIMIZE_FOR_PERFORMANCE`).

The option `CONFIG_KERNEL_XZ` is the compression method of the Linux kernel: tinyconfig relies on XZ instead of GZIP (the default choice). `CONFIG_OPTIMIZE_INLINING` option determines whether the kernel forces gcc to inline functions. `CONFIG_SLOB` is one of three available memory allocators in the Linux kernel.

Finally, `CONFIG_NOHIGHMEM`, `CONFIG_HIGHMEM4G`, or `CONFIG_HIGHMEM64G` set the physical memory on x86 systems: the option chosen here assumes that the kernel will never run on a device with more than 1 Gigabyte of RAM.

2.3.3 Linux kernel rapid evolution

The Linux kernel is a system in constant development, involving thousands of developers and releasing a new important version every two months. Each release adds hundreds of thousands of lines of code, over hundred of thousands file changes in tens of thousands of commits. Each release also changes the configuration space, by adding, removing or modifying hundreds of options. These releases have an impact on performance and properties: for instance the tiny kernel configuration binary size went from 7MiB to 12MiB between version 4.13 and 4.15.

To better handle the evolution in such a large system, automation of tasks is more and more needed to avoid mistakes. Notably, Coccinelle [109] is a tool developed to automatically detect, document and update drivers impacted by an evolution in the kernel API.

2.3.4 Linux and Machine Learning

While using learning techniques to model properties of configurable systems is quite an active field, the efforts toward Linux are scarce, and always with drastic simplifications.

Sincero *et al.* [136] considered 352 options and 146 random configurations for the non-functional property scheduling performance. Using an *analysis of covariance*, they mapped a few options to their impact on the property.

Siegmund *et al.* [133] considered 25 options and 100 random configurations for binary size.

In both cases, only a tiny fraction of the options has been considered, and selected with the help of expert knowledge or documentation.

Up to our knowledge, we are not aware of approaches that try to predict or understand non-functional, quantitative properties (*e.g.,* binary size) over the entire set of options and

Algorithm	Handles interactions	Accuracy	Interpretability	Scales for 50 options	Scales for 5000+ options
Linear Regression	No	−	++	Yes	?
Performance-influence model	Yes	++	++	Yes	?
Decision Tree	Yes	++	++	Yes	?
Random Forest & GB Trees	Yes	+++	−	Yes	?
Neural Networks	Yes	+++	−−	Yes	?

Table 2.1 – Summary of known approaches.

thousands of Linux kernel configurations.

2.4 Conclusion

In Table 2.1 we summarize some concerns and algorithms of the state-of-the-art when it comes to performance prediction in Software Product Lines.

The main concern for this is the handling of interactions between options. Linear Regressions are not able to handle these, which is the reason of a poor accuracy for this task. Performance-influence models leverage Multiple Linear Regression with a specific encoding of interactions, which allows for a good accuracy, on par with Decision Trees, naturally handling interactions. Ensembles of Decision Trees, such as Random Forests or Gradient Boosting Trees, provide the best accuracy alongside Neural Networks, at the cost of interpretability.

While a model from Linear Regression and performance-influence model takes the form of a linear or polynomial function, and a Decision Tree is a propositional formula, which are directly readable – and understandable – by humans, the two most accurate models are not. For ensembles of Decision Trees, there are widely used solutions to extract the feature importance of the models, giving precious insights. For Neural Networks, there are also solutions to draw insight, but mainly by the use of a surrogate model, or approaches not targeted at our problem, such are saliency maps for computer vision or natural language processing.

In all cases, while the scalability on a few dozens options has been well demonstrated, no study has explored larger numbers of options. Although we know that some algorithms, notably Neural Networks, are well-suited for thousands or even millions of inputs, we still do not know how well they handle SPL performance prediction.

The first objective of this thesis is to explore this scalability over options, and we find out that most techniques do not scale. Tree-based solutions and neural networks being the only techniques that we determined to be scalable and accurate, we leverage Random Forests and its partial interpretability to automate feature selection, that allows other techniques to scale, or the already scalable technique to perform better and faster.

We then explore the different usages of Tree-based algorithms, for automated performance specialization, but also for transferring a model from one version to another at low cost.

Chapter 3

Binary Size prediction for Linux Kernel: The Challenge of Colossal Configuration Space

3.1 Introduction

With now more than 15,000 configuration options, including more than 9,000 just for the x86 architecture, the Linux kernel is one of the most complex configurable open-source system ever developed. If all these options were binary and independent, that would indeed yield 2^{15000} possible variants of the kernel. Of course not all options are independent (leading to fewer possible variants), but some of them have tri-states values: yes, no, or module instead of simply boolean values (leading to more possible variants). Users can thus choose values for activating options – either compiled as modules or directly integrated within the kernel – and deactivate options. The assignment of a specific value to each option forms a *configuration*, from which a kernel can hopefully be compiled, built, and booted. Linux kernels are used in a wide variety of systems, ranging from embedded devices and cloud services to powerful supercomputers [145]. Many of these systems have strong requirements on the kernel size due to constraints such as limited memory or instant boot [56, 107]. Obtaining a suitable trade off between kernel size, functionality, and other non-functional concerns (*e.g.,* security) is an important challenge. For instance, activating an individual option can increase the kernel binary size so much that it becomes impossible to deploy it. Hubaux *et al.* [58] report that many Linux users complained about the lack of guidance for making configuration decisions, and the low quality of the advice provided by the configurators. Beyond Linux and even for much smaller configurable systems, similar configuration issues have been reported [3, 60, 125, 161, 162, 163, 169].

A fundamental issue is that configuration options often have a significant influence on non-functional properties (here: binary size[1]) that are hard to estimate and model *a priori*. There are numerous possible options values, logical constraints between options, and potential interactions among configuration options [51, 71, 114, 123, 131] that can have an effect on quantitative properties such as size. As we will further elaborate in Section 4.2, the effort of the Linux community to document options related to kernel binary size is highly valuable, but mostly relies on human expertise, which makes the maintenance of this knowledge quite challenging on the long run. Furthermore, numerous works have showed that quantifying the performance influence of each individual option is not meaningful in most cases [51, 71, 114, 123, 131]. That is, the performance influence of n options, all jointly activated in a configuration, is not easily deducible from the performance influence of each individual option. As our empirical results will show, the Linux kernel binary size is not an exception: options such as `CONFIG_DEBUG_*` or `CC_OPTIMIZE_FOR_SIZE` have cross-cutting, non-linear effects and cannot be reduced to additive effects, hence basic linear regression models, which are unable to capture interactions among options, give poorly accurate results.

Numerous works have considered the problem of learning the effects of options and their interactions. The process of "sampling, measuring, learning" configurations has been applied on several configurable systems in different domains [114, 51, 123, 131, 71]. However, the number of options was limited to dozens of options. At the scale of Linux, there are more than 9,000 options for the x86 architecture and only a fraction of possible kernel configurations can be measured owing to the cost of compiling and building kernels (a build typically takes about 10 minutes on a modern computer). Linux thus questions whether learning methods used in the state of the art would scale (w.r.t. training time), provide accurate models, and provide interpretable information at such an unprecedented scale.

In this chapter, we consider a wide range of state-of-the-art learning algorithms to explore their behavior on a yet untested scale in term of number of options. The lack of empirical results on this matter left by previous state-of-the-art experimentations, limited to at most 60 options, makes it impossible to know if there is a problem or not.

The contributions of this chapter are as follows:

— The design and implementation of a large study about kernels' sizes. We compute at scale different measures and we engineer specific methods for feature encoding

[1] The terminology *footprint* is also used in the literature to refer to the *"binary size of a generated product"* [133, 135]

and feature construction [2];

— A comparison of a wide range of machine learning algorithms over prediction errors. We analyse the amount of configurations needed for training and replicate the experiments using different learning algorithms and measures of kernel size.

— An infrastructure, called TuxML, and a comprehensive dataset of 95,854 configurations with measurements of size as well as learning procedures for replication and reproducibility of the results [139].

3.2 Non-Functional Property prediction of Software Product Lines

In this section, we describe the different problems and solutions existing in the state-of-the-art when it comes to predicting the properties of specific configurations of software systems. Section 3.2.1 describes the process of building a dataset of configuration measurements. Section 3.2.2 formalizes our problem as a statistical learning one. Section 3.2.3 lists existing solutions in the state of the art and describes their limits.

3.2.1 Gathering configurations' data

Our approach, like other learning-based performance models, requires engineering effort to measure non-functional properties (here: the binary size of a Linux kernel) out of a sample of configurations. We have developed the tool TuxML to build the Linux kernel in the large *i.e.,* whatever options are combined. TuxML uses Docker to host the numerous packages and tools needed to compile, build, and measure the Linux kernel. Inside Docker, a collection of Python scripts automates the build process. Docker offers a *reproducible* and *portable* environment – clusters of heterogeneous machines can be used with the same libraries and tools (*e.g.,* compiler versions). In particular, we can use a grid computing or a cloud infrastructure to build a large set of configurations.

The two main steps followed by TuxML to measure kernel binary sizes are as follows:

1. *Sampling configurations.* For this step, we relied on randconfig to randomly generate Linux kernel configurations. randconfig has the merit of generating valid configurations respecting the numerous constraints between options. It is also a mature tool that the Linux community maintains and uses [95]. Though randconfig does not

[2] We consider options and features as two separate terminology as explained in 3.4.1

produce uniform, random samples (see Section 3.5), there is still a wide diversity within the values of options (being *'y'*, *'n'*, or *'m'*). We use this sample to create a .config file.

2. *Kernel size measurement.* Given a set of .config files, TuxML builds the corresponding kernels. Throughout the process, TuxML measures the size of vmlinux, a statically linked executable file containing the kernel. We saved the configurations and the resulting sizes in a database.

The outcome is a dataset of Linux configurations together with their binary size. Each line of the dataset is composed of a set of configuration option values and a quantitative value.

3.2.2 Statistical learning problem

At this step, we have built a dataset of configurations and binary sizes. Predicting the size of a configurable system now becomes a supervised regression problem, *i.e.,* we aim to predict a quantitative variable out of features, built from configuration options as explained above.

Let us formalize the problem. First, we denote p the number of configuration options and define the configuration space $\mathbb{C} = \{0,1\}^p$. Out of this space of configurations $\mathbb{C}$, we gather a subset of d configurations, denoted $\mathbb{C}_{\mathbb{S}} \subset \mathbb{C}^d$, as explained in Section 3.2.1.

We separate $\mathbb{C}_{\mathbb{S}}$ into a training set $\mathbb{C}_{\mathbb{S}}^{tr}$ and a test set $\mathbb{C}_{\mathbb{S}}^{te}$, so $\mathbb{C}_{\mathbb{S}} = \mathbb{C}_{\mathbb{S}}^{tr} \bigoplus \mathbb{C}_{\mathbb{S}}^{te}$.

Then, we denote:

— $F(\mathbb{C}, \mathbb{R}^+)$ the ensemble of functions of $\mathbb{C}$ in $\mathbb{R}^+$,

— $f : C \to R^+ \in F(\mathbb{C}, \mathbb{R}^+)$ the function affecting to any configuration $c \in \mathbb{C}$ its performance $f(c) \in \mathbb{R}^+$,

— $f(\mathbb{S})$ the distribution of performance of a set of configuration $\mathbb{S}$, verifying: $\forall\ \mathbb{S} \subset \mathbb{C}, f(\mathbb{S}) = \{f(c), c \in \mathbb{S}\}$.

With respect to these notations, the goal is to train a learning algorithm $\hat{f}$ estimating the function f for each measured configuration of the training set $c \in \mathbb{C}_{\mathbb{S}}^{tr}$, *i.e.,* $\hat{f} = \underset{h \in F(\mathbb{C},\mathbb{R}^+)}{\operatorname{argmin}} L(f(\mathbb{C}_{\mathbb{S}}^{tr}), h(\mathbb{C}_{\mathbb{S}}^{tr}))$, where L is a loss function.

The training set $\mathbb{C}_{\mathbb{S}}^{tr}$ is used to obtain a prediction model, while the testing set $\mathbb{C}_{\mathbb{S}}^{te}$ only tests the prediction performances of $\hat{f}$. The principle is to confront predicted values $\hat{f}(\mathbb{C}_{\mathbb{S}}^{te})$ to observed (*i.e.,* ground truth) values $f(\mathbb{C}_{\mathbb{S}}^{te})$.

From options to features. Due to discrepancy in vocabulary between the configurable systems domain and the machine learning domain, we clarify how both terms, "option" and "feature" will be used in this chapter and the remaining of this thesis. An *option* is a variable in a configuration of a system. In the Linux case, every available option has a value in the configuration file .config[3]. A *feature* is an independent variable given to a machine learning algorithm. Traditionally in Software Engineering, a feature corresponds to one and only one option and vice versa [49, 148]. Most of the time it is actually the case for Linux. However, we found opportunities to apply feature engineering techniques that can alter this observation. The most frequent change is an option not being represented at all by a feature, because it has the same value across all examples. Another example is when several options can be merged into a single feature, since they are logically implied or excluded. In any case, it is straightforward to trace back features to the corresponding options.

3.2.3 Limitations of state-of-the-art solutions

Numerous statistical learning algorithms, capable of handling a regression problem, are used in the literature of configurable systems. The most used are standard machine learning techniques, such as *linear methods for regression*, *decision trees (CART)*, or *random forests* [114]. These algorithms differ in terms of computational cost, expressiveness and interpretability. For instance, linear regressions are considered as easy to interpret, but are unable to capture interactions between options or to handle non-linear effects. On the opposite side of the spectrum, neural networks are blackbox models that can reach higher accuracy on larger datasets. In between, there are variants of algorithms (*e.g.,* Lasso [144]) or ensemble methods (*e.g.,* random forests [18]) together with hyperparameters that have different impacts on training time, accuracy, and interpretability. For instance, *max_depth*, a hyperparameter of random forest, controls the maximal depth of the trees (*i.e.,* the maximal number of times we can split the dataset) and may affect results.

Siegmund *et al.* [131] introduced a learning method called performance-influence model. Feature-forward selection and multiple linear regression are used in a stepwise manner to shrink or keep terms representing options or interactions. This method aims to handle interactions between options, limit the number of options to learn on, and provide a

[3] Technically, the Linux options that are not specified in a .config file have the value 'n'.

human-readable formula describing influence of options and combinations of options on performance. Performance-influence models can be seen as a *wrapper feature selection* method in which feature selection is performed by training models on different sets of features and comparing the resulting accuracy to determine the best set.

Though performance-influence models have been used in various settings [66, 69, 71, 80, 81, 131, 157], it is yet to be proven whether the method is suited for the scale of Linux. Specifically, the number of possible interactions in Linux is huge. As stated in [81], for p options, there are p possible main influences, $p \times (p-1)/2$ possible pairwise interactions, and an exponential number of higher-order interactions among more than two options. In the worst case, all 2-wise or 3-wise interactions among the 9K+ options are included in the model, which is computationally intractable. Even if a subset of options is kept, there is a combinatorial explosion of possible interactions. It may hinder the scaling of the method or dramatically increase the training time. Kolesnikov *et al.* [81] reported that it take hundreds of minutes for systems with fewer than 30 options, which is far from 9K+ options. Moreover, it is unclear whether linear regressions, used as part of the stepwise process to keep relevant options or interactions, are accurate enough in the context of Linux.

Lack of empirical evidence at the scale of Linux. In general, we do not know whether learning methods used in the state of the art would scale (w.r.t. training time), provide accurate models, and interpretable information at the scale of Linux. For instance, among linear regressions, decision tree, and random forest, what is the most accurate algorithm? How many configurations should be measured to reach a high accuracy? In fact, a gap in the literature is the lack of reference for large-scale configurable systems: there is no report of any experiment, or even any attempt, to work on systems with more than dozens of options. For instance, Valov *et al.* [148] and Guo *et al.* [49] compared various learning methods over datasets with 52 options at best. As is, there is no evidence that these techniques can be effective with large-scale systems, but there is no evidence of the opposite either. We aim to fill this lack of evidence through an empirical study.

3.3 Study design

The purpose of this chapter is to answer only one research question:

— **(RQ1, state-of-the-art) How do state-of-the-art techniques perform on the Linux dataset?** We use various state-of-the-art algorithms on our Linux dataset to assess how they can be used.

In this section, we describe our choices of implementation when addressing this question.

Section 3.3.1 details how we constitute our dataset of Linux configurations. Section 3.3.2 describes the statistical learning methods we have considered while Section 3.3.3 presents the metrics used to answer the different research questions of this chapter.

3.3.1 Dataset

For gathering data at scale, we used TuxML (see details in Section 3.2.1) and a computing grid called IGRIDA[4]. We only focus on the kernel version 4.13.3 (release date: 20 Sep 2017). It is a stable version of Linux (*i.e.,* not a release candidate). Furthermore, we specifically target the x86-64 architecture *i.e.,* technically, all configurations have values `CONFIG_X86=y` and `CONFIG_X86_64=y`. During several months, we used a cluster of machines to build and measure 95,854 random configurations. In total, we invested **15K hours of computation time**. A build took more than 9 minutes per configuration on average (standard deviation: 11 min). We removed configurations that do not build (*e.g.,* due to configuration bugs of Linux).

Data description

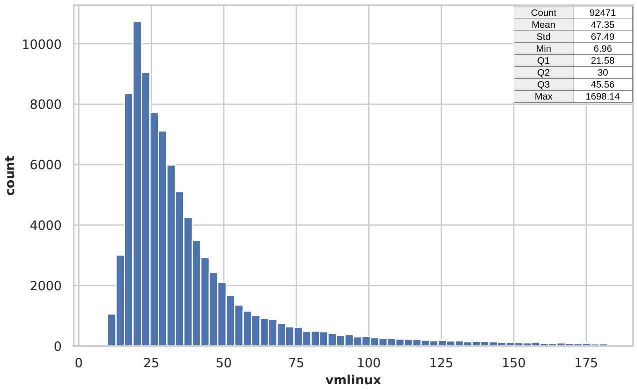
The number of possible options for the x86 architecture is 12,797 options, but the 64 bits constraint further restricts the possible values (some options are always set to *'y'* or *'n'* values). We observe that more than 3K options have a unique value mainly due to the presetting of `CONFIG_X86_64=y`. We use this opportunity to remove the options that have a unique value.

In total, 9,286 options have more than one value *i.e.,* there are nine thousand predictors that can potentially have an effect on binary size.

Each configuration is composed of 9,286 options together with one measurement (binary size of vmlinux).

Finally, after adding the number of activated options as a feature, and the elimination of feature collinearity, the dimension of the dataset (i.e. number of features x number of configurations) is 8,743 x 95,854.

[4] `http://igrida.gforge.inria.fr/`



Count	92471
Mean	47.35
Std	67.49
Min	6.96
Q1	21.58
Q2	30
Q3	45.56
Max	1698.14

Figure 3.1 – Distribution of size (in Mb) without outliers

Distribution of performances

Before discussing the statistical learning, we briefly analyze the distribution of kernel sizes of our dataset.

The minimum size of vmlinux is 7Mb and corresponds to the size of tinyconfig, a predefined configuration that tries to preset option values to 'n' as much as possible. The maximum size is 1,698Mb, the mean is 47Mb with a standard deviation of 67Mb. Figure 3.1 shows the distribution of vmlinux size (with 0.97 as quantile for avoiding extreme and infrequent values of sizes). Most values are around 25Mb, and there is an important variability in sizes.

3.3.2 Statistical learning implementation

Learning methods

For addressing the research question, we have considered a wide range of learning methods.

Algorithms. We considered *Linear Regressions*, *Performance-Influence Models*, *Decision Trees*, *Random Forests*, *Gradient Boosting trees* and *Neural Networks*. We chose to work with this set of algorithms for two reasons. *First,* they have already been successfully used in the literature of configurable systems [114]. (Polynomial) linear regressions, decision trees, and random forests are the most used learning techniques in the literature. *Second,* we aim to gather (strong) baselines and explore the tradeoffs w.r.t. accuracy and interpretability. In addition, we considered Lasso, Ridge, and Elastic Net that perform embedded feature selection and/or regularization in order to enhance the prediction accuracy and interpretability of standard linear regressions. We also included in our experiments so-called polynomial regression: it is linear regression that operates over polynomial combinations of the features with degree two. The goal was to investigate the effects of handling 2-wise interactions as part of the learning process.

Hyperparameters. Most of the selected algorithms are sensitive to *hyperparameters*, which may affect accuracy results. Selecting the right values for hyperparameters should not be neglected. Otherwise, the best algorithm could be sub-optimal after the hyperparameter optimization. We optimize their hyperparameters, and explore a wide range of values as part of our study using grid search and cross-validation [139]. Here are the choices of construction and hyperparameter optimization we made for the following algorithms:

- *Linear Regressions.* We rely on ordinary least squares (OLS) method and there is no specific hyperparameter here.
- *Decision Trees.* We find the best hyperparameters to be at max depth of 27 and minimum sample split of 45.
- *Random Forests.* We performed Random Forests with a maximum depth of 20 and a minimum sample split of 10, over 48 estimators (to match the 4 cores/8 threads capacity of the machine).
- *Gradient Boosting Trees.* For Gradient Boosting Trees, the maximum depth was 15 and the minimum sample split was 100, over 50 estimators.
- *Neural Networks.* Our implementation of neural networks is a multilayer feed forward network. Linux configurations go through three dense layers with ReLU activation functions. We rely on an Adam Optimizer [77], since in our case, it had better convergence properties compared to a standard stochastic gradient descent. Besides, we noticed that the architecture of the neural network should be revised for small training sets. Specifically, when the size of the training set is lower than one thousand configurations, we chose a monolayer (*i.e.,* a Linear Regression with a ReLU activation). When the size of the training set is between one and ten thousands of configurations, we chose a 2-layer architecture. We fed the network with batches of 50 configurations.

Training and test. For each learning method, we split our sample of 95,854 configurations and their associated size measures into multiple training and testing set of various sizes N (10%, 20%, 50%, 80%, 90% of the dataset). For instance, when $N = 10\%$ of the training set, we use 90% of the remaining configurations to test the predictions. In all cases, we reproduced the experiment 5 times to mitigate the randomness and report on the average as well as the standard deviation.

Loss function. We used the Mean Squared Error (MSE) as the loss function L (see Section 3.2.2) for all these algorithms. For instance, the loss function evaluated on the training set MSE_{tr} would be defined as follows:

$MSE_{tr} = \frac{1}{\#(\mathbb{C}_{\mathbb{S}}^{tr})} \sum_{c \in \mathbb{C}_{\mathbb{S}}^{tr}} (f(c) - \hat{f}(c))^2$

where $\#(\mathbb{C}_{\mathbb{S}}^{tr})$ is the number of configurations of the training set, w.r.t. the notations of Section 3.2.2.

Experimental infrastructure

Hardware When measuring computation time (RQ4), we only used one machine with an Intel Xeon E5-1630v3 4c/8t, 3,7GHz, with 64GB DDR4 ECC 2133 MHz memory. We also used this machine to report on prediction errors.

Reproducible environment. We created an infrastructure on top of a Docker image [139] The infrastructure takes a file as input that includes the specification of experiments (*e.g.,* training set size, feature selection method if any, learning algorithms) and serializes the results for further analysis (*e.g.,* MAPE and feature importance). This approach makes it easy to reproduce all our experiments.

Machine learning libraries. We rely on Python modules scikit-learn [20] and Tensorflow [93] to benefit from state of the art machine learning algorithm implementations. We also build an infrastructure around these libraries to automatically handle the feature engineering part, the control of hyperparameters, training set size, *etc.* For example, it is possible to specify a range of values in order to exhaustively search the best hyperparameters for a given algorithm (more details can be found online [139]). For Performance-influence model, we rely on SPLConqueror [5], the implementation made by the authors of the algorithm.

Bounds for training. We needed to set up bounds to what is deemed reasonable for the experimentation. The first is about memory consumption, as we performed all our experiments on the same machine with 64GB of memory, which is already higher than the current personal computer standard. Since the algorithms are meant to be used by developers or users of configurable systems on their regular machine, which usually are equipped from 8 to 32 GB of memory, and seldom with more than 64 GB, an algorithm needing more than 64 GB will be deemed out of scale. For CPU consumption, we put the limit at 10 days of computing.

[5] `https://github.com/se-passau/SPLConqueror`, git commit 89b68ce

3.3.3 Metrics and measurements

To assess the accuracy of each method, we rely on the Mean Absolute Percentage Error (MAPE), an interpretable and comparable metric. The MAPE of the testing set $MAPE_{te}$ would be defined as follows:

$MAPE_{te} = \frac{100}{\#(\mathbb{C}_{\mathbb{S}}^{te})} \sum_{c \in \mathbb{C}_{\mathbb{S}}^{te}} \frac{\left|f(c) - \hat{f}(c)\right|}{f(c)} \%$

where $\#(\mathbb{C}_{\mathbb{S}}^{te})$ is the number of configurations of the test set, w.r.t. the notations of Section 3.2.2.

We choose the MAPE to depict our results since:

1. it is frequently used when the quantity to predict is known to remain above zero (as in our case) [114];
2. it has the merit of being easy to understand and compare (it is a percentage);
3. it handles the wide distribution and outliers of vmlinux size (Figure 3.1);
4. it is commonly used in approaches about learning and configurable systems [114].

All reported accuracy values are made with this metric in order to have an easier comparison, and lower is the value the better.

3.4 Results

Most of studied techniques could perform their task in the time and memory limits we had set. We however failed to get results from two of the techniques we have tried: SPLConqueror and Polynomial regression:

— Importing the Linux dataset into SPLConqueror raises an error about insufficient memory, so SPLConqueror cannot perform anything on the dataset [6].

— Polynomial regression integrates interactions among features (in the same vein as performance-influence model) and does not scale for a degree 2.

Table 3.1 reports the highly variable accuracy in MAPE of all successfully tested techniques with various training set sizes. We can distinguish multiple groups of similar performance and techniques:

— Linear regression based techniques: OLS Regression, Ridge, ElasticNet and Lasso all present poor results, with Lasso being the only one with less than 50% MAPE, but still 34% at best. The results show that linear regressions are not suited for

[6] We used SPLConqueror from commit 9b68ce on Ubuntu 20.04 LTS and got the message "System.OutOfMemoryException: Insufficient memory to continue the execution of the program."

Algorithm	N=10	N=20	N=50	N=80	N=90
OLS Regression	74.54±2.3	68.76± 1.03	61.9 ± 1.14	50.37±0.57	49.42±0.08
Lasso	34.13±1.38	34.32±0.12	36.58±1.04	38.07±0.08	38.04±0.17
Ridge	139.63±1.13	91.43±1.07	62.42±0.08	55.75±0.2	51.78±0.14
ElasticNet	79.26±0.9	80.81±1.05	80.58±0.77	80.57±0.71	80.34±0.53
Decision Tree	15.18 ± 0.13	13.21 ± 0.12	11.32±0.07	10.61± 0.10	10.48± 0.15
Random Forest	12.5±0.19	10.75±0.07	9.27±0.07	8.6±0.07	8.4 ±0.07
GB Tree	11.13±0.23	9.43±0.07	7.70±0.04	7.02±0.05	6.83±0.10
N. Networks	-	13.92 ± 0.99	9.46 ± 0.15	8.29 ± 0.18	8.08 ± 0.12
Polynomial Regression	-	-	-	-	-

Table 3.1 – MAPE of different learning algorithms for the prediction of vmlinux size, with N being the percentage of the dataset used as training

Linux and that the problem of predicting size cannot be trivially resolved with a simple additive, linear model (as hypothesized earlier in the chapter);

- Tree-based techniques: Decision Tree, Random Forest and Gradient Boosting Tree all show MAPE at less than 20% even with "only" 10% of the dataset, even reaching 6.83% for GB Tree at 90% of the dataset. Decision trees are inferior to random forest and GB Trees;
- Neural Networks work quite well but require much more data to be efficient compared to Tree-based techniques.

How does state-of-the-art techniques perform on large configurable systems? At the exception of Performance-Influence model and Polynomial Regression, most of the techniques studied can handle the Linux dataset in reasonable time and memory limits. On the accuracy side, we can notice that Linear regression based techniques do not present very accurate results, and only Tree-based and neural network techniques are suited for Linux.

3.5 Threats to Validity

3.5.1 Internal Validity

The selection of the learning algorithms and their parameter settings may affect the accuracy and influence interpretability. To reduce this threat, we selected the most widely used learning algorithms that have shown promising results in this field [114] and for a fair comparison we searched for their best parameters. We deliberately used random sampling over the training set for all experiments to increase internal validity. For each sample size, we repeat the prediction process 5 times. For each process, we split the dataset into training and testing which are independent of each other. To assess the accuracy of the algorithms and thus its interpretability, we used MAPE since most of the state-of-the-art works use this metric for evaluating the effectiveness of performance prediction algorithm [114].

Another threat to validity concerns the (lack of) uniformity of randconfig. Indeed randconfig does *not* provide a perfect uniform distribution over valid configurations [95]. The strategy of randconfig is to randomly enable or disable options according to the order in the Kconfig files. It is thus biased towards features higher up in the tree. The problem of uniform sampling is fundamentally a satisfiability problem. To the best of our knowledge, there is no robust and scalable solution capable of comprehensively translating Kconfig files of Linux into a format usable by a SAT solver. Second, uniform sampling either does not scale yet or is not uniform [117, 21, 22, 42]. So we had to stick with randconfig. We see randconfig as a *baseline* widely used by the Linux community [121, 95].

3.5.2 External Validity

A threat to external validity is related to the target kernel version and architecture (x86) of Linux. Because we rely on the kernel version 4.13.3 and the non-functional property binary size, the results may be subject to this specific version and quantitative property. However, a generalization of the results for other non-functional properties (*e.g.,* boot time) would require additional experiments. It is actually a challenge *per se* to scale up such experiments. Binary size has the merit of being independent of the hardware: we do not need to control for the heterogeneity of machines or repeat the measurements. Binary size has also proved to be a non-trivial property to learn.

Overall, we focused on a single version and property to be able to make robust and reliable statements about whether learning approaches can be used in such settings. Our results suggest we can now envision to perform an analysis over other versions and properties to generalize our findings.

3.6 Conclusion

Is it possible to learn the essence of a gigantic space of software configurations with respect to a given property of interest? We addressed an instance of this problem through the prediction of Linux kernel sizes for the x86_64 processor architecture, dealing with over 9,000 options and thus a huge configuration space.

We invested 15K hours of computation to build and measure 95,854 Linux kernel configurations and found that:

- Linear regression based techniques give low accuracy results.
- Polynomial regression and performance-influence models run out of memory, possibly due to the way interactions are encoded.
- Tree-based algorithm and neural networks show quite a good accuracy, and Gradient Boosting Trees yield the best results.

The fact that performance-influence models approach, a well-established algorithm designed especially to deal with Software Product Lines problems and maybe the most advanced one currently in the field, does not scale with the number of options in Linux, is a worrying discovery for the relevance of current research based on that algorithm and yet claiming their scalability to all Software Product Lines.

We believe that the limitation comes from the way interactions are encoded into the algorithm, as each interaction is represented by a new features when presented to the algorithm building the model. This lead to a combinatorial explosion, and this despite the heuristics used to mitigate it. This type of encoding is used in other works around SPL, and thus severely limits its applicability for really large configurable systems.

Tree-based algorithm and neural networks are the best algorithms in our study and offer a positive answer to the question about the possibility to learning on such a configuration space. However, the costs in term of resources needed to gather the dataset, and to a lesser extent to train the models, can be a deterrent to engage in such initiative, especially in a context of constant evolution.

Chapter 4

Tree-based Feature Selection

4.1 Introduction

Feature Selection is a well-known process in machine learning, and after considering the previous chapter on property prediction of Linux Kernel, we aim to follow a novel research direction. Instead of considering the whole set of options when training a model, we first compute a subset of influential options with tree-based algorithms. Then learning methods can operate over a much reduced feature space.

Our hypothesis is that some configuration options of Linux have little effect on a given property of interest (e.g., the kernel binary size) and thus can be removed without incurring much loss of information. So-called feature selection techniques are worth considering when there are many features and comparatively few samples as in Linux (see *e.g.,* Chapter 18 of [53]). Though we are not necessarily in extreme cases like the analysis of DNA microarray data [138, 23] in which there are only a few samples available, the Linux case exhibits a very large number of options p.

The promise of feature selection is that the resulting models are easier to interpret and faster to train since only a few options are considered (*i.e.,* $p' << p$ with p' the new number of considered options). The danger, however, is that the accuracy might decrease if the subset of options is badly chosen. Hence several empirical questions arise: Can tree-based feature selection identify relevant options with effects on binary size? Is feature selection competitive w.r.t. accuracy? How many configurations should be measured and what machine learning algorithms should be used to reach a high accuracy?

Prior works considered only a few options and configurations. Sincero *et al.* [136] considered 352 options and 146 random configurations for the non-functional property scheduling performance. Siegmund *et al.* [133, 135] considered 25 options and 100 random configurations for binary size. In this study and in contrast to prior works, we make no assumption about the supposed influence of some options and our experiments consider the entire 9K+ options of the Linux kernel on the x86_64 architecture. We also considered a baseline of 95K+ different configurations. Beyond Linux, only dozens of options over a few configurations are usually considered in the literature of configurable systems [114]. The gap with the Linux case is substantial, up to the point that some learning algorithms, including the ones proposed in the literature, do not scale (see Section 3.2.3) or have poor accuracy (see Section 4.5). We also aim to automate the process of selecting a subset of relevant options prior to the training of a prediction model. Owing to the large number of options (9K+), feature selection is particularly suited for Linux. Yet, it is unclear whether feature selection is effective at this scale. Feature selection can be done manually: some experts chose a subset of options based on some knowledge to make the experiment less costly and thus affordable. We want to automate this process, thus avoiding to rely on a potentially incomplete and outdated knowledge. Feature selection can also be realized through *wrapper methods* that determine the best set of features by comparing models trained on different sets [23]. Techniques such as performance-influence models [131] enter into this category. Empirical results show that such state-of-the-art methods either do not scale or are poorly accurate at the scale of Linux. For addressing these limitations, we develop a two-step method that relies on *tree-based feature selection.* First, a learning process computes a so-called feature ranking list for ordering important features. Then, this list is fed to a learning algorithm that operates over a subset of predictive features. We use tree-based learning algorithms (*e.g.,* random forest) to learn the feature ranking list, owing to their predictive power and interpretability. As detailed in the rest of this chapter, empirical results show that (1) tree-based learning algorithms are the most accurate among al tested algorithms and (2) the use of tree-based feature selection outperforms learning approaches that do not resort to feature selection.

The contributions of this chapter are as follows:

- A measurement of the effect of tree-based feature selection on a wide range of machine learning algorithms over prediction errors, interpretability, and training time. We find that tree-based feature selection is able to identify a reduced set of options that both speeds up the training time and produces simpler and more accurate models;
- A qualitative analysis of identified options based on the cross-analysis of Linux documentation, default options' values and configurations, and experts' knowledge. We find that the subset of options found by tree-based feature selection is consistent w.r.t. Linux knowledge;

4.2 Kernel Sizes and Documentation

We first conducted a study that identifies the kernel options for which the documentation explicitly discusses an impact on the kernel size. We apply the following protocol.

Protocol. The objects of the study are the *KConfig* files that describe and document each kernel option. We use the *KConfigLib* tool to analyze the documentation of the Linux kernel 4.13.3 for the x86 architecture [54]. This architecture contains 12,797 options. For this analysis, we automatically gathered the options whose documentation contains specific terms related to size terminology: *big, bloat, compress, enlarge, grow, huge, increase, inflat, inlin, larger, little, minim, optim, overhead, reduc, shrink, size, small, space, trim*, percent and size values (*e.g.,* 3%, 23 MB). The list of terms was built in an iterative way: we intensively read the documentation and checked whether the terms cover relevant options, and to include new terms, until a fixed point is reached. We then manually scrutinized the KConfig documentation to state whether it indeed discusses a potential impact of the option on the kernel size.

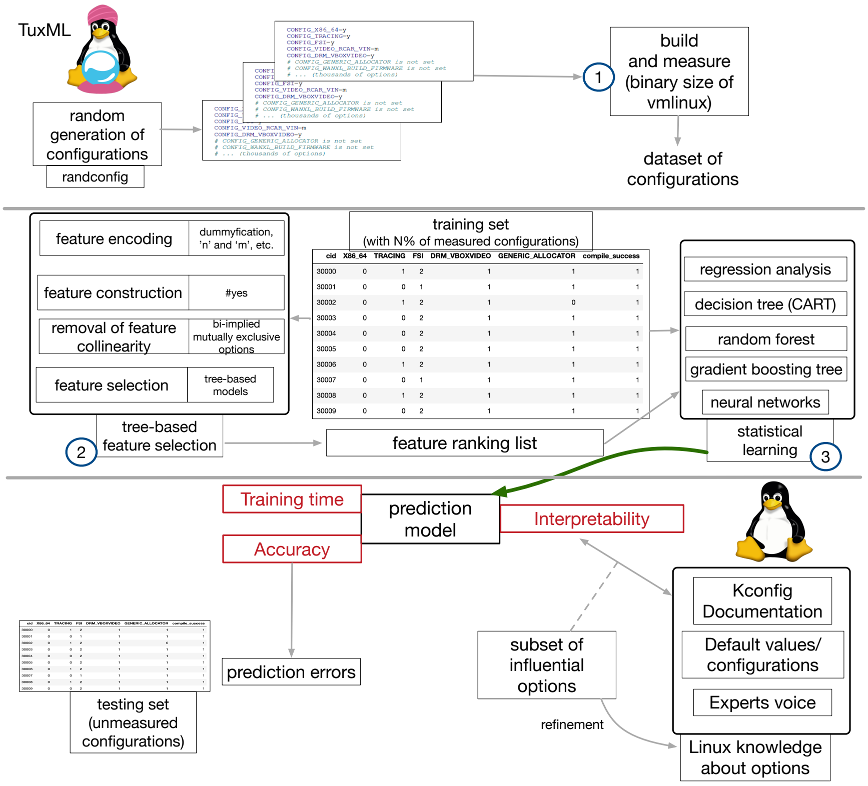
Results. On the 12,797 options, 2,233 options have no documentation (17.45%). We identified 147 (1.15%) options that **explicitly** discuss potential effects on the kernel size. Kernel developers use *quantitative* or *approximate* terms to describe the impact of options on the kernel size.

Quantitative examples include: "*will reduce the size of the driver object by approximately 100KB*"; "*increases the kernel size by around 50K*"; "*The kernel size is about 15% bigger*".

Regarding approximate terms, examples include: the "*kernel is significantly bigger*"; "*making the code size smaller*"; "*you can disable this option to save space*"; "*Disable it only if kernel size is more important than ease of debugging*".

4.3 Learning with Tree-based Feature Selection

This section describes our learning approach based on tree-based feature selection. The whole process is depicted in Figure 4.1 and each step is detailed in the remainder of this section. The first step of the process, the data acquisition part, is presented in the previous chapter, as we are using the exact same dataset. Section 4.3.1 details feature encoding, feature construction and tree-based feature selection that can be used as inputs of learning algorithms (see middle part of Figure 4.1). By selecting influential features in


TuxML
random generation of configurations
randconfig
1
build and measure (binary size of vmlinux)
dataset of configurations
feature encoding
dummyfication, 'n' and 'm', etc.
feature construction
#yes
removal of feature collinearity
bi-implied mutually exclusive options
feature selection
tree-based models
2
tree-based feature selection
training set (with N% of measured configurations)
feature ranking list
regression analysis
decision tree (CART)
random forest
gradient boosting tree
neural networks
statistical learning
3
Training time
Accuracy
prediction model
Interpretability
prediction errors
testing set (unmeasured configurations)
subset of influential options
refinement
Kconfig Documentation
Default values/ configurations
Experts voice
Linux knowledge about options

```
CONFIG_X86_64=y
CONFIG_TRACING=y
CONFIG_FSI=y
CONFIG_VIDEO_RCAR_VIN=m
CONFIG_DRM_VBOXVIDEO=y
# CONFIG_GENERIC_ALLOCATOR is not set
# CONFIG_WANXL_BUILD_FIRMWARE is not set
# ... (thousands of options)
```

cid	X86_64	TRACING	FSI	DRM_VBOXVIDEO	GENERIC_ALLOCATOR	compile_success
30000	0	1	2	1	1	1
30001	0	0	1	1	1	1
30002	0	1	2	1	0	1
30003	0	0	2	1	1	1
30004	0	0	2	1	1	1
30005	0	0	2	1	1	1
30006	0	1	2	1	1	1
30007	0	0	1	1	1	1
30008	0	1	2	1	1	1
30009	0	0	2	1	1	1

Figure 4.1 – From measuring to learning with tree-based feature selection

large configuration spaces, the promise of the solution is to improve accuracy, training time, and interpretability (see bottom part of Figure 4.1).

4.3.1 Our Approach: Tree-based feature selection

Feature engineering and selection for learning at scale. A major problem when predicting the non-functional property of software variants is to identify the influence of options together with their interactions. Trying to simply list all interactions would lead to a combinatorial explosion, especially at the scale of Linux. The approach taken by embedded feature selection methods such as Lasso or decision tree, or wrapper methods such as in performance influence model, is good in that sense: the goal is to reduce the number of options p by only taking the relevant ones in a step-wise manner. However, this reduction still operates over a very large number of variables *at training time*. In contrast, we propose to encode and identify a subset of options *before* applying machine learning algorithms. The idea of this pre-processing is that machine learning algorithms can then operate over a reduced number of variables at training time. We detail so-called *tree-based feature selection* in the next section. To the best of our knowledge, the use of feature selection prior to the learning has not received attention in the literature of configurable systems.

Intuition. A hypothesis is that some configuration options of Linux have little effects and can be removed without incurring much loss of information. Feature selection techniques are worth considering when there are many features and comparatively few samples. Though we are not necessarily in an extreme case like the analysis of DNA microarray data [23, 138] in which there are only a few available samples, the Linux case exhibits a very large number of options p. The goal is to operate over a reduced number of variables $p' \ll p$ when training, ideally over a relatively small training set (*e.g.,* less than 10K) to reduce the overall cost of learning.

The left middle part of Figure 4.1 lists each step of the tree-based feature selection method.

Feature encoding A first way to reduce p is to properly encode options as features. Regression analysis requires that we encode three possible values of options (*e.g.,* 'y', 'n', 'm') into numerical values. An encoding of 'n' as 0, 'y' as 1, and 'm' as 2 is a first possible solution. However, some learning algorithms (*e.g.,* linear regression) will assume that two nearby values are more similar than two distant values (here 'y' and 'm' would be more similar than 'm' and 'n'). This encoding will also be confusing when interpreting the negative or positive weights of a feature. There are many techniques to encode categorical variables (*e.g.,* dummy variables [41]). We observe that the 'm' value has no direct effect on the size since kernel modules are not compiled into the kernel and can be loaded as needed. Therefore, we consider that 'm' values have the same effect as 'n' values, and these values can be merged. As a result, the problem is simplified: an option can only take two values (*i.e., "yes"* or *"no"*, *1* or *0* as defined in Section 3.2.2). For non-tri-state options, which are only 319, we simply discarded them. With this encoding, the hypothesis is that the accuracy of the prediction model is not impacted whereas the problem is simpler to handle for learning algorithms and easier to interpret.

Feature construction. The number of 'y' options' values in a configuration, denoted #yes, can have an impact on the kernel size, and has been added as a new feature in the dataset. The rationale is that the more options are included, the larger is the kernel. At first glance, the inclusion of a new feature is counter-intuitive since our focus is rather to reduce the number of variables. However, this feature is informative and may have a strong predictive power: it makes explicit the domain knowledge about the relationship between binary size and number of options. By adding such a feature, the hypothesis is that the accuracy of the prediction model can be improved.

Feature selection Knowing which configuration options are most predictive of size is part of our goal. With that in mind, *feature importance* is a useful concept: it is the increase in the prediction error of the model after we permuted the feature's values [97]. For a decision tree, the importance of a feature is computed as the (normalized) total reduction of the splitting criterion (*e.g.,* Gini or entropy) brought by that feature. For random forest, *feature permutation importance* is computed through the observation of the effect on machine learning model accuracy of randomly shuffling each predictor variable [18, 97, 111].

Elimination of feature collinearity. In machine learning, feature collinearity happens when two (or more) features are completely correlated. Feature collinearity is in fact a well-known, general issue in machine learning, especially for model interpretability [38, 18, 111, 97, 43]. Feature collinearity also becomes an issue when quantifying how much a feature is important to predict the target values. Specifically, when multiple features share the same values, hence all having the same information value, there is no other choice to be made than random split between those features. It has two major consequences:

1. there is some *instability*: it is hard to reproduce a feature's importance without setting a random seed at all levels of the experiment, as a feature's importance depends on the previous splits in the tree.
2. the *importance* of the information held by a group of features will be *distributed between them* and considered far lower than it should be [97].

In order to mitigate feature collinearity, we grouped collinear options into a single feature. There are at least two relevant cases of collinear options: (1) some options are logically bi-implied and have always the same values in a configuration; (2) some options are mutually exclusive and have always opposite values (*e.g.,* `CC_OPTIMIZE_FOR_SIZE` and `CC_OPTIMIZE_FOR_PERFORMANCE`). In both cases, we create one "artificial" feature that acts as a representative group of all involved options. It does not bring any problem for interpretation as it is possible to retrieve which options compose one "artificial" feature.

The elimination of feature collinearity is an important step realized just before the actual feature selection. We choose to rely on a *tree-based feature selection.* The principle is to rely on decision trees or ensembles like random forest to produce a *feature ranking list.* Such algorithms are capable of handling non-linear effects and have strong prediction power. (Empirical results confirm that tree-based learning algorithms are the most accurate, see Section 4.5).

Once the feature importance of random forests is computed, the feature ranking list can be created. The feature ranking list orders features by importance. We use a threshold to select a subset of relevant options (*e.g.,* the top 500 of the list, by decreasing importance). This threshold allows one to control the reduced number of variables $p' \ll p$ to consider at training time. A learning algorithm can use the subset of features to eventually train a predictive model.

Stability. It should be noted that the algorithm of random forest relies on random selection: this can cause instability in the computation of feature importance. To mitigate this threat, we rely on two techniques. First, we simply repeat the process several times and compute the average of feature importance. Second, we eliminate collinear features before training the random forest (see previous explanations).

4.4 Study design

The main research question is *RQ: How effective is tree-based feature selection for predicting Linux kernel sizes?*, which can be decomposed into five main research questions:

— **(RQ1, accuracy) How accurate is the prediction model with and without tree-based feature selection?** Depending on *e.g.,* training set size, feature selection or hyperparameters of learning algorithms, the resulting model may produce more or fewer errors when predicting the size of unseen configurations.

— **(RQ2, stability) How stable is the Feature Ranking List?** Depending on the models used to create the Feature Ranking List, the set of selected features can vary.

— **(RQ3, training time) How much computational resources is saved with tree-based feature selection?** By selecting a subset of relevant features, learning algorithms operate on a smaller number of features at training time. Depending on *e.g.,* training set size, feature selection or hyperparameters of learning algorithms, the resulting model may need more or fewer computational resources to get trained.

— **(RQ4, interpretability) How do feature ranking lists, as computed by tree-based feature selection, relate to Linux knowledge?** We aim to compare the retrieved knowledge extracted from models with knowledge from the Linux community. Linux knowledge notably comes from the Kconfig documentation and experts' insights.

In this section, we describe our choices of implementation when addressing these questions. For most of the experiments, we use the same process as in the last chapter, with additional steps when it comes to feature selection.

Tree-based feature selection

Each learning algorithm previously described can be used with tree-based feature selection. Our automated feature selection process is based on the feature ranking list, created from insights extracted from a machine learning algorithm. In our case and for **RQ2**, we rely on 20 random forests to compute the list. Random forests appear to reach high accuracy, better than decision trees. For each random forest, the importance of each feature is computed, then all features are sorted by their importance, the most important feature being the top one in the ranking. We take the average ranking across all 20 random forests to get the final feature ranking list. As further elaborated in **RQ3**, we needed to rely on the results of multiple random forests due to the instability of the model. Even with the same dataset and hyperparameters, the feature importance of two random forest was fairly different. For example, if we take the 300 most important features of the first random forest, we could only retrieve half of them in the second random forest top 300.

An important factor that can influence the accuracy of a learning method with tree-based feature selection is k, the number of selected features within the feature ranking list. We make k vary from 50 (*i.e.,* the 50 top influential features) to 8,743 (the size of the entire list), in steps of 50. The goal is to identify how many features are ideally needed at training time for reaching high accuracy.

We apply all learning methods of Section 3.3.2 with and without tree-based feature selection, and report on the difference of accuracy in Section 4.5.1.

4.4.1 Metrics and measurements

Aside from the accuracy metric, Mean Absolute Percentage error which is described in Section 3.3.3, we also measure the stability of the Feature Ranking List, the training time and the quality of the List.

(RQ3) Stability

The Feature Ranking List is created using the feature importance given by a model. The Random Forest relies on random selection and this can cause inconsistency in the feature importance.

To measure the stability of Feature Ranking Lists, we count the number of common features in the top N features between two lists. The two lists have to be created under the same conditions (training set and hyperparameters) to be sure that only the randomness introduced by the machine learning algorithm causes the instability.

(RQ4) Training time

During experiments, we also collected the time needed to train each model. All reported times are given in seconds and have been measured on the same machine.

(RQ5) Interpretability

We quantitatively and qualitatively confront the feature ranking list with Linux knowledge coming from the Kconfig documentation, expert insights (two developers of the Linux kernel), and tinyconfig. We report on options considered as influential by both sides, but also options absent in the list or in the Linux documentation.

4.4.2 Feature Ranking List scenarios

Our approach consists in two distinct phases: the Feature Ranking List creation and the actual model training using the Feature Ranking List. In practice, we consider two usage scenarios. The first one is based on the creation of a feature ranking list over a large portion of the dataset. This usage typically occurs offline, once and for all. This would represent a case when organizations (*e.g.,* KernelCI[1] for the Linux kernel) with computational resources share a Feature Ranking List to the community. It aims to synthesize a knowledge that can be beneficial whenever a training (including hyper-parameterization) is performed. In our experimental settings, we leverage a Feature Ranking List trained on 90% of the dataset with an ensemble of random forests. We investigate the effect of this list on several training sizes and all algorithms. The goal is to understand the potential of the feature ranking list in terms of accuracy (RQ2), computation time (RQ3), and interpretability (RQ4 and RQ5).

A second scenario, with limited resources, is to consider both creating a feature ranking list and actual training phases on the same portion of the dataset. In order to investigate the effect of the Feature Ranking List trained on a lower number of examples, we set

[1] `https://kernelci.org/`

Algorithm	Without tree-based Feature Selection				
	N=10	N=20	N=50	N=80	N=90
OLS Regression	74.54±2.30	68.76± 1.03	61.9 ± 1.14	50.37±0.57	49.42±0.08
Lasso	34.13±1.38	34.32±0.12	36.58±1.04	38.07±0.08	38.04±0.17
Ridge	139.63±1.13	91.43±1.07	62.42±0.08	55.75±0.20	51.78±0.14
ElasticNet	79.26±0.90	80.81±1.05	80.58±0.77	80.57±0.71	80.34±0.53
Decision Tree	15.18 ± 0.13	13.21 ± 0.12	11.32±0.07	10.61± 0.10	10.48± 0.15
Random Forest	12.5±0.19	10.75±0.07	9.27±0.07	8.6±0.07	8.4 ±0.07
GB Tree	11.13±0.23	9.43±0.07	7.70±0.04	7.02±0.05	6.83±0.10
N. Networks	-	13.92 ± 0.99	9.46 ± 0.15	8.29 ± 0.18	8.08 ± 0.12
Polynomial Regression	-	-	-	-	-
Algorithm	With tree-based Feature Selection				
	N=10	N=20	N=50	N=80	N=90
OLS Regression	43.56±1.48	42.58±2.22	40.23±0.22	39.56±0.39	39.29±0.48
Lasso	35.18±0.45	36.53±0.60	39.28±1.06	38.28±0.04	38.61±0.81
Ridge	43.52±1.41	42.29±2.16	40.2±0.27	39.53±0.33	39.24±0.43
ElasticNet	79.66±2.11	81.74±0.65	81.0±0.24	80.84±0.60	81.45±0.20
Decision Tree	13.97±0.08	12.34±0.08	10.75±0.05	10.07±0.09	9.91±0.12
Random Forest	10.44±0.12	9.3±0.06	8.2±0.02	7.67±0.06	7.54±0.02
GB Tree	8.34±0.07	7.00±0.04	5.89±0.02	5.50±0.02	5.41±0.02
N. Networks	-	12.91±0.17	6.17 ± 0.09	5.77 ± 0.04	5.71 ± 0.05
Polynomial Regression	24.65 ± 1.23	22.58 ± 0.18	20.49 ± 0.24	21.53 ± 0.10	20.86 ± 0.04

Table 4.1 – MAPE of different learning algorithms for the prediction of vmlinux size, without and with tree-based feature selection, with N being the percentage of the dataset used as training

up another experiment. We take a percentage of the dataset (*e.g.,* N=10%), create the Feature Ranking List and train a model using that List – all on the same training set. We repeat the process 5 times and report the mean MAPE. Specifically, we report prediction errors on Random Forest and Gradient Boosting Tree – two learning methods that proved to be highly accurate (see results of RQ1) – and on varying training sizes.

4.5 Results

In this section, we present our results and answer the different research questions defined in Section 4.4.

4.5.1 (RQ1) Accuracy

What is the best algorithm to learn the Linux kernel's size?

In Table 4.1, we report the MAPE (and its standard deviation) of multiple statistical learning algorithms, on various training set sizes (N), with and without tree-based feature selection.

We observe that algorithms based on linear regression (Lasso, Ridge, Elasticnet) do not work well, having a MAPE of more than 30% whatever the training set size or the feature selection, showing that the size problem is not only an additive problem. Some do not even take advantage of a bigger training set, like Lasso which even increases its MAPE from 34% to 38% when the training set size goes from 10% to 90%. Tree-based algorithms (Decision Tree, Random Forest, Gradient Boosting Tree) tend to work far better (MAPE of 15% and lower) and effectively take advantage of more data to train on. As a base of comparison, we also report results from Neural Networks which are better than most of algorithms when fed with a big enough training set. 10% of the data in the training set was not enough to make a multiple layer network converge; it led to a MAPE greater than 90%. Besides, using a monolayer network would not be different from a linear regression.

How many configuration options do we need to learn the Linux kernel's size?

Figure 4.2 aims to show the influence on the accuracy of (1) the number k of selected features and (2) the training set percentage (N) of the full dataset. In particular, it depicts how Random Forest performs when varying k and N.

We observe that for a training set size of N=10% (9,500), the accuracy peaks (*i.e.,* lowest error rate) when k=100 with a MAPE of 9.57, and consistently increases with more features. For a training set size of N=50% (47,500), accuracy peaks when k=200 with a MAPE of 7.74. For N=90% (85,500), we reach a MAPE of 7.29 with k=250. Independently from the training set size, we consistently observe in Figure 4.2 that the MAPE is the lowest when k is in the range 100—250.

This optimal number of selected features depends on the algorithm used. Table 4.1 reports the accuracy results with feature selection of the optimal number for each learning method. For example, for Ridge or ElasticNet, it is in the range 250—300, for Lasso, 350—400, for Gradient boosting tree, 1400–1600, for Neural Networks, 1500 (see [139] for more details).

Given these results, we can say that out of the thousands of options of Linux kernel,

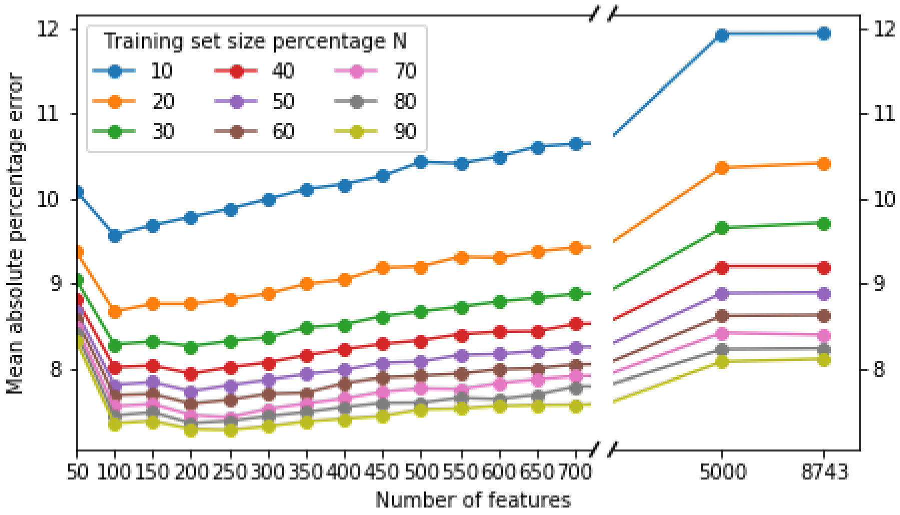


Figure 4.2 – Evolution of MAPE w.r.t. the number k of selected features and the training set percentage N (Random Forest).

Algorithm	With tree-based Feature Selection				
	N=10	N=20	N=50	N=80	N=90
Random Forest	11.70±0.02	9.73±0.02	7.97±0.01	7.53±0.01	7.49±0.01
GB Tree	8.64±0.10	7.20±0.07	6.16±0.29	5.21±0.12	5.13±0.04

Table 4.2 – MAPE of different learning algorithms for the prediction of vmlinux size, with a Feature Ranking List based on the same training set, with N being the percentage of the dataset used as training

only a few hundred actually influence its binary size. From the machine learning point of view, the other features do not bring any more information and even make the model worse by biasing it. The effect of the engineered feature #yes is also quite noticeable, by improving accuracy on tree-based solution by at least 2 points at every training set size. Overall, the approach of reducing the dimensionality of configuration data is validated: feature engineering and feature selection are beneficial to learning methods.

What is the effect of a lower training size on the Feature Ranking List?

Table 4.2 shows the MAPE of models trained on different training set sizes, with the help of a Feature Ranking List itself created with the same training set. The impact of having fewer measurements to create the Feature Ranking List can be noticeable but is non significant overall. The only noteworthy changes compared to Table 4.1 (bottom table, with tree-based feature selection) is that at 10% of training set size, the MAPE of Random Forest is 1.3 point higher (10.44% vs 11.70%). However, it is still better than without feature selection (12.5%). In this practical scenario, the use of tree-based feature selection gives better accuracy results than the traditional training.

Answering RQ1

(RQ1, accuracy) How accurate is the predictive model with and without tree-based feature selection? We find a sweet spot where only 200—300 features are sufficient to efficiently train a Random Forest and a Gradient Boosting Tree to obtain a prediction model that outperforms other baselines (6% prediction errors for 40K configurations). We observe similar feature selection benefits for any training set size and tree-based learning algorithms. Overall, the use of tree-based feature selection is to be recommended when building a predictive model for Linux.

4.5.2 (RQ2) Stability

First benefit of tree-based selection: a stable Feature Ranking List

We studied the stability of the Feature Ranking List, and revised our method to create it. At first, we simply created a list using only one Random Forest. However, when comparing two lists created using two different models yet trained in the same conditions, we could observe major differences already in the top 10 features, with only 9 features being in that top 10 in both. When comparing the top 300, we could only observe from 100 (with feature collinearity) up to 130 (after elimination of feature collinearity) common features. This instability would be a problem so we changed the way we create the Feature Ranking List by using an ensemble of Random Forests.

By using the average feature importance of 20 Random Forests, the resulting Feature Ranking List shows a stability of 10 common features in the top 10 and 286 common features in the top 300. Despite not being perfectly stable, relying on an ensemble of Random Forests proves itself far more stable then using a single model.

When comparing Feature Ranking Lists with and without eliminating feature collinearity, we can observe its influence. For instance, the group #153 is in the 30th position, so a quite important feature, but when each of the options composing the group are taken individually, namely `IOSCHED_NOOP`, `SBITMAP`, and `BLOCK`, they are positioned 136th, 109th, and 130th, which shows a significant decrease in their computed importance.

Answering RQ2

(RQ2, stability) How stable is the Feature Ranking List? Using an ensemble of Random Forest allows the creation of a far more stable list, with more than 95% features in common among the top 300 between multiple lists.

4.5.3 (RQ3) Training time

Our obtained results show that tree-based feature selection is promising w.r.t. accuracy and interpretability. However, this is not the only observed benefit. In fact, it also has a positive impact on computational resources.

Second benefit of tree-based feature selection: a shorter training time

During the experiments, we observed that training the model without feature selection took a lot of time (up to 24 hours for some settings of boosting trees). To further assess the effect of feature selection, we performed a controlled experiment. We measure the computation time for Random Forest and Gradient Boosting Tree, on multiple numbers of features and training set sizes, and report the results in Table 4.3. For Random Forest, we report a 4,632 seconds (77 minutes) computation over 85K rows of the dataset (N=90%), with all features. With 200 features selected, we report a computation time of 97 seconds, 147 seconds for 300 features and 254 seconds for 500 features. The time reduction was respectively 48, 31, 18 fold less than without feature selection.

With Gradient Boosting Tree, on all features and 85k rows of the dataset, we report a computation time of 22,090 seconds (more than 6 hours). In the same conditions, for different numbers of selected features, we observed a difference in time. With 500 features selected, we report a computation time of 1,252 seconds (21 minutes), 2,679 seconds (45 minutes) for 1,000 features, and 4,350 seconds (72 minutes) for 1,500 features. The time reduction was respectively 17, 8, 5 fold less than without feature selection.

The computation time difference between Random Forest and Gradient Boosting Tree is explained by the capability of Random Forest to use multiple threads while Gradient Boosting Tree is not multi-threaded. In the context of intensive search of hyperparameters,

Algorithm	Features	N=10%	N=20%	N=50%	N=80%	N=90%
Random Forest	100	13	14	26	43	49
	200	14	21	50	86	97
	300	17	29	75	129	147
	400	21	37	101	173	199
	500	25	47	126	220	254
	8743[1]	**383**	**814**	**2342**	**4030**	**4632**
GB Tree	250	51	108	317	555	604
	500	100	210	630	1089	1252
	750	148	328	993	1705	1952
	1000	202	449	1360	2341	2679
	1250	266	580	1759	3029	3562
	1500	328	719	2169	3755	4350
	8743[1]	**1693**	**3675**	**11234**	**19302**	**22090**

[1] **Without feature selection.**

Table 4.3 – Computing time (in seconds) for Random Forest and Gradient Boosting Trees with 5 folds, w.r.t. number of features and N (training set percentage). The baseline is when feature selection is not applied (8743 features are used).

it is perfectly possible to run multiple Gradient Boosting Trees in parallel, effectively using multiple threads.

For reference, the end-to-end process takes 13 hours per fold, from finding good hyperparameters for Random Forests to create the Feature Ranking List, to hyperparameters optimisation of Gradient Boosting Tree and training the final model.

These results show big savings in computation time, especially for Random Forest. It allows more intensive search of hyperparameters which greatly impacts the overall accuracy of the models. We are convinced that similar benefits of tree-based feature selection can be obtained for other learning algorithms.

Answering RQ3

(RQ3, training time) How much computational resources is saved with tree-based feature selection? We find that tree-based feature selection speeds the model training by at least 5 times and 0up to 48 times. This way, we show that it is possible to greatly reduce the time needed to produce a model without sacrificing accuracy.

4.5.4 (RQ) Interpretability

On the interpretability of the Linux documentation

Contrary to Neural networks, Gradient Boosting Tree or Random Forest are algorithms with built-in interpretability. There, we can extract a list of features that can be ordered by their importance with respect to the property of interest (here size). So the question is: How do feature ranking lists, as computed by tree-based feature selection, relate to Linux knowledge?

Confrontation with documentation. We confront the feature ranking list to the 147 options referring to size in the Kconfig documentation (see Section 4.2). First, we notice that 31 options have a unique value in our dataset: randconfig was unable to diversify the values of some options and therefore the learning phase cannot infer anything. We see it as an opportunity to further identify influential options. As a proof of concept, we sample thousands of new configurations with and without option `KASAN_INLINE` activated (in our original dataset, `KASAN_INLINE` was always set to 'n'). We did observe a size increase (20% on average), suggesting that the documentation could help identifying influential options we have missed due to randconfig. However, the documentation can put us on the wrong track: we have also tried for 5 other options and did not observe a significant effect on size [139].

Among the resulting 116 options (147 – 31), we found that:

- 7 are in the top 50, 6 are in the top 50—250, 6 are in the top 250—500, and 28 in the top 500—1,500
- 60% of options are beyond the rank 1,500, hence not considered after feature selection. We identified two possible explanations. First, the effect on size is simply negligible: It is explicitly stated as such in the documentation (*"This will increase the size of the kernelcapi module by 20 KB"* or *"Disabling this option saves about 300 bytes"*). Second, some option values are not distributed uniformly (*e.g.,* 98% *'y'* and 2% *'n'* value) in our dataset: the learning phase needs more diverse instances.

Finding of undocumented options. A consequence from the confrontation with Kconfig is that the *vast majority of influential options is either not documented or not referring to size*. In order to further understand this trend, we analyze the top 50 options in the feature ranking list (representing 95% of the feature importance in the Random Forest):

- only 7 options are documented as having a clear influence on size;

— our investigations and exchanges with domain experts [2] show that the 43 remaining options are either (1) necessary to activate other options; (2) the underlying memory used would be based on the size of the driver; (3) chip-selection configuration (you cannot run the kernel on the indicated system without activating this option); (4) related to compiler coverage instrumentation, which will affect lots (possible all) code paths; (5) debugging features; (6) related to DRM (for Direct Rendering Manager) and GPU drivers

Revisiting tinyconfig

In our dataset, we observe that tinyconfig is by far the smallest kernel (≈7Mb). The second smallest configuration is ≈11Mb. That is, despite 90K+ measurements with randconfig, we were unable to get closer to 7Mb. Can our prediction model explain this significant difference (≈4Mb)?

A first hypothesis is that the four pre-set options of tinyconfig have an important impact on the size (see Figure 2.6, page 41). We observe that our prediction model indeed ranks `CC_OPTIMIZE_FOR_SIZE`, one of the four pre-set options, in the top 200. The other three remaining options have no significant effects according to our model and cannot explain the ≈4Mb difference. In fact, there is a more simple explanation: the strategy of tinyconfig consists in minimizing #yes. According to our prediction model, #yes is highly influential feature. We notice that the number of 'y' options of tinyconfig is 224 while the second smallest configuration exhibits 646 options: the difference is significant. Furthermore, tinyconfig deactivates many top-influential options like `KASAN` or DEBUG_INFO that have a positive effect on the binary size when activated. *We can conclude that the insights of the predictive model are consistent with the heuristic of tinyconfig.*

Collinearity and interpretability.

The top 50 of our feature ranking list exhibits 4 groups with collinear features.

Thanks to removal of feature collinearity, we have automatically identified cases in which some options can be grouped together (*e.g.,* we can remove `KASAN_OUTLINE` and only keep `KASAN`). We found that taking in account feature collinearity is beneficial for two reasons (1) we do not miss influential features since the importance is not split among features; (2) experts only have to review a group of features instead of unrelated and

[2] The full data is available on the companion web site [139]

supposedly independent features.

Documentation-based feature selection

Instead of using the top X features from the Features Ranking List of a tree-based feature selection, the 147 options referring to size in the Kconfig documentation (see Section 4.2) can be used to form the feature list fed to the actual training. This scenario corresponds to feature selection realized with expert knowledge. We use all the 147 options in the Linux documentation and only them (considering that an option is encoded as a feature).

The resulting prediction model exhibits a MAPE of 23.6% for Gradient Boosting Tree (the best learning algorithm) and over 90% of the dataset for training. It is more than 4 times the error rate of using the Feature Ranking List used with tree-based feature selection. This show that documentation alone is not suited for feature selection. It is in line with previous observations: the Linux documentation is incomplete (*i.e.,* some important features are missing) and describes features that have little influence on size.

Answering RQ4

(RQ4, interpretability) How do feature ranking lists, as computed by tree-based feature selection, relate to domain knowledge? Thanks to our prediction model, we have effectively identified a list of important features that is consistent with the options and strategy of tinyconfig, and Linux knowledge (Kconfig documentation and expert insight). We also found options that can be used to refine or augment the incomplete documentation of the Linux kernel.

4.6 Threats to Validity

4.6.1 Internal Validity

Aside from the threats describe in Section 3.5.1 about the range of algorithms considered, and the sampling method used which both also affect the experiment from this chapter, we can raise another threat about the Feature Ranking List. The computation of feature importance is subject to some debate and some implementation over random forest, including the scikit-learn's one we rely on, may be biased [18, 43, 97, 111]. This issue may impact our experiments *e.g.,* we may have missed important options. To mitigate this threat, we have computed feature importance with repetitions (*i.e.,* 20 times). Our observation is that the top influential options then remain very similar [139] (see **RQ2**). As future work, we plan to compare different techniques to compute feature importance [18, 43, 97, 111].

4.7 Conclusion

Feature selection is a process that has proved itself efficient in the usage of machine learning in many domains. While Software Product Lines is not an exception, the only previous use was done in a way that does not scale to the number of options of Linux.

Tree-based feature selection is a technique that has been used a lot in biology when dealing with very large datasets, making it potentially suitable for SPL.

In this chapter, we explored this process, the concerns of stability and how to deal with it.

We compared multiple state-of-the-art solutions with and without feature selection and found that tree-based feature selection pays off:

- the accuracy is better with than without tree-based feature selection. It is possible to reach low prediction errors (5.5% for binary kernel size);
- the training time is decreased by at least an order of magnitude;
- the identification of influential options is consistent with, and can even improve, the expert knowledge about Linux kernel configuration. Models are easy to interpret: insights can be validated and communicated;

The decrease of prediction errors is perhaps the most surprising positive result of our empirical inquiry, since feature selection does not necessarily improve accuracy. Overall, we found a subset of options (≈500) that can achieve an interesting sweet-spot along the three dimensions (accuracy, training time, interpretability). Furthermore, feature selection has the merit of being agnostic and fully automated: we did not rely on expert knowledge to achieve this result.

Chapter 5

Automated Performance Specialization

5.1 Introduction

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

The configuration of a configurable system is an error-prone and time-consuming task. There is a combinatorial explosion of possible configurations and the effects of options on performance is hard to document and formalize. Numerous works have shown that quantifying the performance influence of each individual option is not meaningful in most cases [114]. That is, the performance influence of n options, all jointly activated in a configuration, is not easily deducible from the performance influence of each individual option. We thus need to try to capture the complex interactions among options and their effects on performance.

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

Specializing the configuration space of a software system has a long tradition since the seminal paper of Czarnecki *et al.* [33].

Specialization has been considered for targeting specific profiles, usages, deployment scenarios, or hardware settings [6, 9, 32, 59, 143]. The idea is to retain only a subset of configurations that meet a performance threshold (*e.g.,* execution time below one second) and thus discard the rest. Specialization should not be confused with configuration optimization (tuning) where the goal is to find a unique and optimal configuration. Through specialization, users still have flexibility (variability) to configure their systems. The benefit is that some options' value are already preset for reaching a performance threshold. A unique challenge is to identify configurations that should be kept (or, equivalently, to discard non-acceptable configurations), which boils down to specifying constraints among options' values.

On the one hand, measuring all configurations is infeasible in practice owing to the combinatorial explosion of possible configurations. On the other hand, the manual specification of constraints is an error-prone, time-consuming, and hardly repeatable task for any performance objective. It should however be possible to automate the specialization process using machine learning, *i.e.,* measuring a sample of configurations and then learning what options' values should be constrained. Even focusing on learning techniques based on decision trees for their built-in explainability, there is still a wide range of possible approaches that need to be evaluated, *i.e.,* how accurate is the specialization with regards to sampling size, performance thresholds, and kinds of configurable systems.

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

- We propose a new way to use decision trees, called specialized regression and tailored toward performance specialization;
- We design and develop six performance specialization learning strategies with three variants of Decision Trees: regression, classification, and specialized regression;
- We perform an empirical study to compare the accuracy results of these techniques. Through our experimental results, we find that Decision Tree is a very accurate algorithm for performance specialization. However, it is sensitive to threshold variation. Feature selection is a reliable way to produce more accurate results; however

the impact on training time was not significant.

— We perform a case study with the Linux kernel and we show very similar results as for the 8 previously considered configurable systems, except for training time which for Linux presented significant improvements with the use of feature selection.

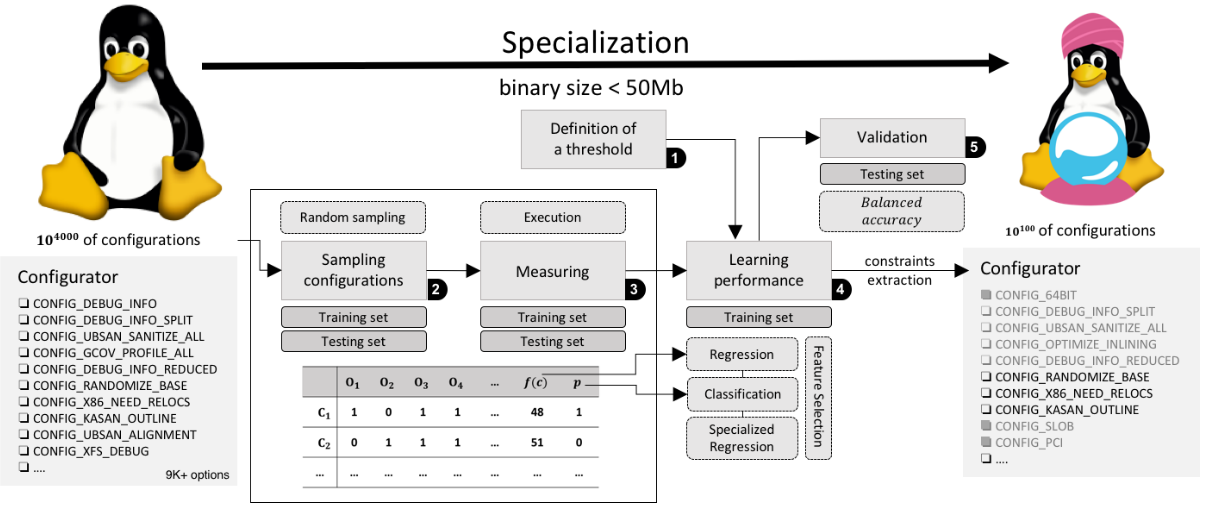
— We provide insights on how to use the different strategies for automated performance specialization. We discuss the tradeoffs of training a model on the fly, and using the default strategy and feature selection. We also point out future directions.

— We have made all the artifacts from our experiments publicly available at `https://github.com/HugoJPMartin/SPLC2021`.

5.2 Performance Specialization

In a configurable system, not all combinations of options' values are possible (*e.g.*, some options are mutually exclusive and some thresholds are required). Variability models are used to precisely define the space of valid (functional) configurations, typically through the specification of logical constraints among options. Assuming that all supposedly valid configurations of a variability model lead to acceptable products can be misleading since constraints may not be specified due to missing knowledge beforehand when the variability model is built. Moreover, nowadays systems are developed by integrating products, which belong to multiple systems, and communicate and interact with each other under various configurations [122]. In this scenario, configurations may fail to interact leading to unacceptable performance, *e.g.* dependencies on external libraries are not considered. The manual identification of constraints is a difficult task and it is easy to forget or wrongly specify a constraint leading to configurations that do not meet a particular requirement [141]. However, it is most of the time impractical to exhaustively test and measure system performance under all possible configurations. To overcome this issue, we can specialize a variability model to deliver the right functionality and performance, assisting stakeholders in making informed decisions.

Specialization is the process of limiting the variability space to a subset of configurations that meet a specific threshold. Thresholds are logical decision rules over non-functional property values with regards to system limitations, such as `binary size` $<$ `50Mb`. They are constraints defined as equality (*i.e.*, $=$) or inequalities (*i.e.*, $\leq$; $<$; $>$; $\geq$) [105]. The specialization process is a transformation process that takes a variability space as input and yields another variability space as output, such that the set of configurations denoted by the latter space is a subset of the configurations denoted by the former space (see Figure 5.1) [32].



	o_1	o_2	o_3	o_4	...	$f(c)$	p
c_1	1	0	1	1	...	48	1
c_2	0	1	1	1	...	51	0
...	...	...	...	...	...	...	...

Figure 5.1 – Configuration of a specialized Linux kernel. Given a performance threshold (binary size < 50Mb), the specialization process identifies constraints among options that preclude some (combinations of) values (in gray); users still have some flexibility to configure other options (in black).

The specialization of a variability space involves the addition of a set of new constraints (rules) to options' values. Such rules describe how performance influences the system options' interactions, i.e. the variability space is restricted to only configurations satisfying the given performance threshold. The restricted subset of configurations represents the space of specialization. As an example, the Linux can be specialized to obtain a kernel binary size below `50Mb` (see Figure 5.1). To meet the threshold, a set of options values (in gray) are preset to true while others to false, and the remaining options (in black) can still be configured. This is helpful to avoid large kernels that take too much time to compile. Notice that we consider a threshold, thus contrary to optimization approaches that aim at satisfying a specific optimization objective (*e.g.* `minimize binary size`), we aim at specializing the configuration space to a subset of acceptable configurations. Although optimization is not our aim here, the specialization may highly assist the optimization process later when deriving a suited configuration, *e.g.* to support the variability at runtime.

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

The learning process consists of four main stages: (1) definition of a performance threshold, (2) sampling, (3) measuring, and (4) learning (see Figure 5.1). The fifth stage (validation) is interesting for assessing the prediction models and their accuracy – a central research question of this chapter. First, the process starts by defining a threshold and selecting a sample set of valid configurations. Second, the sample of configuration is built, and the properties of interest are measured. Third, these measurements and the performance threshold are used as input to accurately learn a prediction model. Finally, the validation step computes the accuracy of the prediction model.

Prediction models support stakeholders understanding the characteristics of the variability space, *i.e.* the effects of some options and how options interact. Interpretability for specialization is very important both for validating the insights of the learning and for encoding the knowledge into a variability model. As an example, consider learning techniques based on decision trees. Each branch of a tree represents a set of rules (decisions) that satisfy a threshold leaf. Thus, as a result, rules are mined by building the conjunction of a path to reach a threshold leaf. In the example of Figure 5.1, the algorithm learns that by having the options `CONFIG_64BIT`, `CONFIG_SLOB`, and `CONFIG_PCI` deselected, and `CONFIG_DEBUG_INFO_SPLIT`, `CONFIG_UBSAN_SANITIZE_ALL`, `CONFIG_OPTIMIZE_INLINING`, and `CONFIG_DEBUG_INFO_REDUCED` selected, the binary size threshold is (almost) guaranteed to be respected. Thus, options are preset and no modification of these option values are allowed. Without specialization, we are likely to set different option values. An interesting example is "defconfig", a default configuration for Linux, in which the binary size is around `70Mb`. So, in case users start the configuration with "defconfig", they do not meet the specified threshold. Notice that learning is useful and necessary since manually discovering such rules is tedious, error-prone, and time-consuming. Moreover, it requires expert knowledge of the system domain. The overall outcome is constraints among options that are retrofitted into a variability model and a configurator. Constraints can either force the values of individual options or logically involve several options. Such constraints are learned out of learning models.

5.3 Automated Performance Specialization

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

5.3.1 Specialization as a learning problem

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

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

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

5.3.2 Learning algorithms for specialization

Numerous statistical learning algorithms can be used for performance specialization. These algorithms differ in terms of computational cost, expressiveness and interpretability. We now review the literature of configurable systems (based on the systematic survey [114]). We discuss what algorithms are suited or not for our specific problem. *Linear regressions* are considered as easy to interpret, but are unable to capture interactions between options or to handle non-linear effects [97]. *Neural networks* can reach high accuracy on large datasets. DeepPerf [52] has been developed for tackling configurable systems with dozens of options. Empirical results show the effectiveness of DeepPerf [52] on all systems of our study (except Linux that has not been considered). However, neural networks are black-box functions for which it is hard to extract rules (constraints) among options, a top requirement in the specialization problem.

Siegmund *et al.* [131] introduced a learning method called *performance-influence model.* Feature-forward selection and multiple linear regression are used in a stepwise manner to shrink or keep terms representing options or interactions. This method aims to handle interactions between options, limit the number of options to learn on, and provide a human-readable formula describing influence of options and combinations of options on performance. However, performance-influence model does not address the performance specialization problem. First, the model addressed a regression problem while we are interested in predicting the class of a configuration. Second, the technique does not retrofit constraints into a variability model. The synthesized information (a formula with coefficients) is not designed for extracting rules and constraints. Third, a performance-influence model aims to construct a global performance model: a rewrite is necessary to take the threshold into account. Fourth, the step-wise addition of interactions among options does not scale for Linux that exhibits 9K+ options (see more details hereafter).

*Decision trees (*e.g., *CART)* are the most used technique in the literature [114]. Decision trees have been used either for classification or regression, have reached competing accuracy with the state-of-the-art, and are interpretable by construction [142, 148, 150]. The tree structure is ideal for capturing interactions between options. Rules can easily be extracted and retrofitted into a variability model.

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

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

5.3.3 Learning strategies

Fundamentally, the learning problem presented in Section 5.3.1 is a supervised, binary *classification* problem. However, and which makes it noteworthy, the learning has at its disposal continuous values (performance measurements). In this regard, the problem is close to a *regression problem*, except that one does not want to eventually predict a quantity but a class.

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

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

Gap in the related work. Classification trees have been considered in prior works [6, 9, 142]. However, a systematic comparison with other techniques is missing and it is unclear how the approach works for large systems like Linux.

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

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

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

5.3.4 Tree-based Feature selection

As we demonstrate in Chapter 4, the feature selection can have positive impact on performance prediction models, both on accuracy and training time. The idea is to decrease the complexity of the problem, without removing too valuable information. This selection can be done in several ways, such as using expert knowledge, or in an automated fashion. We will reuse here the process based on extracting the feature importance of Random Forest, and ranking feature by this metric, as we showed its ability to scale on thousands of features.

As the learning task is a classification one and not a regression one, the effects of this feature selection on the automated specialization process are yet to be determined.

5.4 Study Design

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

— **RQ1: What is the accuracy and cost of learning strategies?** We aim to evaluate the prediction errors of specialization learning on nine real-world configurable systems, with varying cost in terms of the number of configuration measurements.

— **RQ2: What is the best learning strategy?** There are multiple ways to use decision trees (namely classification, regression and specialized regression) in order to synthesize the rules needed for the specialization. With regard to accuracy, is there a best strategy *e.g.,* whatever the subject system?

— **RQ3: What are the effects of tree-based feature selection on the accuracy**

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

Table 5.1 – Datasets information, including number of features and number of measured configurations

of performance specialization? We aim to investigate whether feature selection improves or degrades the accuracy of learning strategies and for which subject systems, performance thresholds, and training set size.

— **RQ4: What is the time cost of the 6 learning strategies to predict performance of a configurable software system?** This question is to evaluate the practicality and feasibility of a proposed strategy. To answer this question, we show the time consumed by the feature selection (if used), hyperparameter searching and training process on various highly configurable software systems (including Linux). We also put in perspective the fact that some techniques are learned once and for all, whatever the performance thresholds.

5.4.1 Datasets

In order to perform our experiments, we relied on datasets already used in the community (see details in Table 5.1).

Common SE datasets

Our first part of the experiments is to train Decision Trees on reliable datasets commonly used on numerous occasions in the past [52, 131] to observe the efficiency of learning techniques.

Linux dataset

To investigate the scalability of the approaches, we rely on the dataset made available by Acher *et al.* [5] on Linux kernel size. It consists of the binary size measurement of 92.562 configurations, limited to x86 architecture and 64 bits systems, and obtained with the use of randconfig, a widely used tool to generate random kernel configurations, and valid w.r.t constraints between options. It contains all boolean and tristate options, encoded as 0 and 1; the "module" options value encoded as a 0; as well as a feature counting the number of active options, for a total of 9,467 features.

5.4.2 Learning algorithms

Our three approaches rely on Decision Trees, and we used the CART implementation in scikit-learn [113], a widely used state-of-the-art Python library for machine learning. We explored a wide range of hyperparameters, such as the maximum depth of the Tree, or the loss function for regression approaches between MSE and Friedman MSE, in a grid-search fashion to optimize the efficiency of the Decision Trees. We also varied the training set size from 10% to 70% and used the remaining (at least 30% to avoid overfitting) as test set to study its impact on the accuracy, the sampling being made at random among the datasets, as Pereira *et al.* [115] show that this sampling method is a strong baseline overall. We repeated each iteration from 5 to 20 times and report on the average value, to reduce the impact of randomness on the experiments.

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

5.4.3 Threshold influence

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

5.4.4 Metrics

To measure the accuracy of the Decision Trees classification, we use the balance accuracy[19], which is a combination of sensitivity and specificity:

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

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

5.4.5 Tree-based Feature selection

We trained 20 Random Forests on 10% of the training set and extracted a Feature Ranking List, except for Linux kernel, where we used the List provided with the dataset [5]. We then repeated the learning as much as needed to find the optimal number of options.

We consider 6 different strategies:

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

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

5.5 Results

In this section, we discuss the answers to our research questions defined in Section 5.4.

Table 5.2 – Average accuracy and standard deviation per system and learning technique over all thresholds and for 70% training set size. FS stands for *Feature Selection* while *All Strategies* mean that we systematically choose the best strategy among the 6.

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

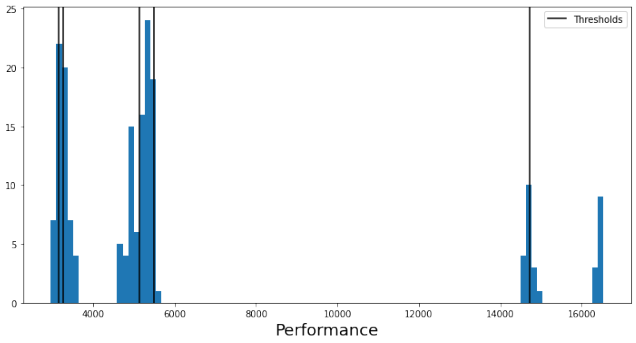


Figure 5.2 – Performance Distribution for BerkeleyJ

5.5.1 Results RQ1 and RQ2—Accuracy

Table 5.2 is a summary of the accuracy over all thresholds for each system with 70% training set size. With the exception of the SQLite system, we report an accuracy of more than 90% overall, showing that Decision Tree is able to handle the complexity of configurable systems for the specialization task in most cases. We observe that the best results are held by the *Classification* strategy on 6 out of 9 systems. However, since the difference between strategies is very slight, we can consider the 6 strategies to be accurate. Furthermore, there is no learning strategy better than another on all thresholds. For some systems, a high standard deviation can be observed due to thresholds variations within a subject system. For instance, for BerkeleyJ [1] the accuracy for 70% T.S. (training set size) goes from 77.2% to 100%, respectively at 10% and 50% of A.C. (acceptable configurations), which is tied to the threshold variation. This can be explained by the very inconsistent performance distribution shown in Figure 5.2.

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

Strategies

In this section, we report mainly on the tables present in our supplementary material[2], where we detailed results over all strategies and systems for each considered threshold and training size. Here, due to space limitations, we add the detailed results only for Apache and Linux (see Tables 5.3 and 5.4). Notice that the results in Table 5.2 (for all systems)

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

refer to a 70% training set size and an average over all thresholds.

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

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

Specialized Regression. This approach shines at higher thresholds, being the best strategy in almost all cases at 80% and 90% A.C. The improvement can be very important, for instance Apache at 90% A.C. and 10% T.S., specialized regression is better by 6 and 9 points over regression and classification. About the gap influence (Section 5.4.3), most of the time the minimum value (¼ of the mean) gives the best results, and in some cases (for Berkeley C and Berkeley J) it's the maximum value.

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

5.5.2 Results RQ3—Feature selection

In Tables 5.3 and 5.4, the *influence of feature selection is represented by the value inside brackets*. In the vast majority of cases, the use of feature selection improves the results, though the improvement can be very low (less than 1 point). The most significant improvements are for LLVM (see in our supplementary material), with cases showing up to 10.3 points more than their counterpart without feature selection.

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

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

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

5.5.3 Results RQ4—Training time

The training time of a Decision Tree is in the order of milliseconds (except for Linux). It is up to 22 ms for Specialized Regression on HIPAcc, which is the system with the largest configuration dataset (see Table 5.1). The training time reduction after feature selection is very slight. The most significant improvement is the reduction from 6.4 ms to 3.5 ms (for Regression Decision Tree on SQLite). Proportionally, it is a 45% time reduction. However, the reduction is imperceptible for a human.

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

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

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

Table 5.3 – Decision tree classification accuracy on performance specialization for Apache on three strategies. The difference of feature selection on accuracy is represented by the value inside brackets. Bold represents the best result among other strategies (including feature selection).

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

Table 5.4 – Decision tree classification accuracy on performance specialization for Linux kernel on three strategies. The difference of feature selection on accuracy is represented by the value inside brackets. Bold represents the best result among other strategies (including feature selection).

5.6 Discussion

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

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

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

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

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

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

5.7 Threats to validity

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

5.7.1 Internal validity

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

Threshold variation. The performance threshold – the limit set by the user to define an acceptable configuration – can make the accuracies of the learning techniques vary a lot. We considered this aspect by using different thresholds, based on the performance distributions of each system and dataset. However, this may rise a problem: as we can see in Figure 5.2, the 10% and 20% thresholds are very close and also very constraining, which might explain the low accuracy of the specialization models. In practical terms, these thresholds do not correspond to any realistic expectation from a user. Despite this threat, we did observe the phenomenon for only one system while none of the 6 learning techniques have been favored. However, we warn that this is a threat that should be considered for any future work or transfer in a real use case.

Sampling. For this experiment, we focused only on random sampling to avoid spreading ourselves too much. Pereira *et al.* [115] shows that random sampling is a strong baseline, but also warns about the potentially strong influence of different sampling strategies, especially when comparing learning techniques. The exploration of other sampling techniques is definitely in the scope of future work.

5.7.2 External validity

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

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

5.8 Conclusion

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

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

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

We have exposed a panorama of accurate learning techniques for the performance specialization problem. As future work, we plan to assess how the inferred configuration knowledge relates to domain experts' knowledge (*e.g.,* Linux developers). It will require to investigate how readable and comprehensible are constraints extracted from the specialization process. Another research direction is how this knowledge transfers across deep variability [86] (*e.g.,* versions and workloads of a system).

Chapter 6

Model Shifting for Performance prediction across Versions

6.1 Introduction

We detailed previously how complex and costly modelling the performance of a configurable system can be, especially in systems with thousands of options. In all previous work on performance modelling, a key aspect of software systems has been put aside to lower the complexity: evolution.

At each new release, the problem of obtaining an accurate performance model of a configurable system may arise again due to the variability in time. For configurable systems evolving at a rapid pace, **sampling again each new version is impractical**. Since it requires such a large amount of computational resources to measure the performance of each configuration to be used as input to a machine learning algorithm, this process could even take more time than the release period.

To overcome this issue, we propose to transfer the learning across versions. Since the feature space (i.e., the set of configuration options) can change across versions, our approach falls under the category known as *heterogeneous* transfer learning, opposite to *homogeneous* transfer learning, that assumes the feature space remains unchanged during evolution. Identifying how to efficiently apply transfer knowledge of the learned model as the systems evolve is challenging. This is indeed a well-known general problem in machine learning [110, 158], made even more difficult because of the heterogeneity of configuration spaces because it may cause bias on cross-version feature representation [36].

Existing works [24, 61, 64, 66, 67, 68, 78, 98, 103, 149, 150, 155] have attempted to investigate the transfer challenge. However, they investigated it in the context of homogeneous feature spaces and not for the case of system code evolving across different versions, i.e. heterogeneous feature spaces. Thus, to the best of our knowledge, no work has shown evidence that transfer learning can model variability in space and time of configurable systems accurately. Moreover, existing works do not consider very complex systems that have thousands of options and at the same time evolve frequently with large deltas between releases.

For this chapter, we purposely again consider the Linux kernel, because it is one of the most complex representative cases of this problem. It has thousands of options (*e.g.,* around 15,000 for version 5.8) and hundreds of releases over more than two decades. We selected seven releases spanning over three years from version 4.13 (2017) to 5.8 (2020). On this dataset we first answer the following research question: *RQ1. To what extent does Linux evolution degrade the accuracy of a performance prediction model trained on a specific version?*

To this end, we first perform an experiment to quantify the impact of Linux evolution (*i.e.,* the release of a new kernel version) on configuration performance (specifically: kernel binary size for a Linux configuration). To the best of our knowledge, no prior work has attempted to check whether a learnt configuration performance model can be used regardless of the applied system evolution over time. Our experiments show that evolution does indeed impact the learnt model by degrading the prediction (by a ratio of 4 to 6) down to the point where it cannot effectively be used on subsequent releases.

After that, we propose a new automated transfer *Evolution-aware Model Shifting* (tEAMS) approach that consists in applying a tree-based model shifting using gradient boosting trees, backed up with feature alignment to handle the problem of heterogeneity in configuration spaces. The goal of tEAMS, as any transfer learning technique, is to beat approaches that learn at every release ("from scratch"), as illustrated in Figure 6.2. We then answer the next research question: *RQ2. What is the accuracy of tEAMS compared to learning from scratch and other transfer learning techniques?*

Our results show that tEAMS outperforms state-of-the-art approaches by reaching a low and constant 5.6% to 7.1% error rate rather than 8.3 to 9.2% for learning from scratch method. Our contributions are summarized as follows:

1. We design and implement a large-scale study of the effectiveness of transfer learning to model kernels' variability in space and time.
2. We show empirical evidence for the degradation of binary size predictions from version 4.13 onward.
3. We propose a novel technique (tEAMS) for transfer learning across versions, with two variants: one shot transfer and incremental transfer.
4. We evaluate our results over seven kernel versions and a dataset of 243K+ con-

> figurations spanning over three years: tEAMS vastly outperforms the accuracy of state-of-the-art transfer learning strategies. Moreover, it is cost-effective since it works with the addition of a reduced set of training samples over future versions.

The chapter is organized as follows. Section 6.2 investigates how performance predictions degrade over evolution releases. Section 6.3 presents a new approach called tEAMS. Section 6.4 evaluates tEAMS on the history of Linux kernels from 4.13 to 5.8 and compares it to state-of-the-art. Sections 6.5 and 6.6 discuss various aspects of the proposed solution, and the threats to validity. Finally, Section 6.7 concludes this chapter.

We provide a replication package with all artifacts (including datasets and learning procedures): `https://zenodo.org/record/4960172`.

6.2 Impacts of Evolution on Configuration Performance

In chapter 3, we show that creating a performance prediction model, or in this case a binary size prediction model, is a costly task and out of reach of most learning techniques. Moreover, this has been done on a fixed version (4.13) and we now need to inquire about its durability.

In this section, we aim to quantify the impact of Linux evolution (*i.e.,* the release of a new kernel version) on configuration binary size and prediction models.

Mühlbauer *et al.* [98] investigated the history of software performance to isolate when a performance shift happens over time. If we know evolution can impact the performance of a configurable software, we do not actually know if and how much it can impact a performance prediction model.

A hypothesis is that the evolution has no significant impact and the Linux community can effectively reuse a binary size prediction model across all versions. The counter hypothesis is that the evolution changes the binary size distributions: in this case, a measurable and practical consequence would be that a binary size model becomes inaccurate for other versions. In other words, if the degradation of the accuracy of a prediction model is to be expected, it is necessary to know whether such degradation is sharp enough to be a problem for the Linux community. However, none of these hypotheses has been investigated in the literature. Therefore, quantifying the impacts of evolution is crucial and boils down to addressing the following research question:

(RQ1) To what extent does Linux evolution degrade the accuracy of a binary size prediction model trained on a specific version? To address it, we measure the accuracy of a performance prediction model, specifically a binary size prediction model, trained in one specific version (*i.e.,* 4.13), when applied to later versions (*e.g.,* up to 5.8).

6.2.1 Experimental Settings

We now present the datasets we gathered on different Linux configurations and versions; the learning algorithms we used to build the prediction models, as well as the accuracy metric.

Dataset

We compiled and measured Linux kernels on seven different versions. Table 6.1 further details each considered release version:

- 4.13: this release was the starting point of our work with huge investments (builds and measurements of 90K+ configurations);
- 4.15: the release was the first to deal with the serious chip security problems meltdown/spectre [118] that mainly apply to Intel-based processor (x86 architecture). A broad set of mitigations has been included in the kernel, which can have an effect on kernel sizes;
- 4.20: the last version before 5.0, with several x86/x86_64 optimizations. As part of the in-depth analysis on the evolution of core operation performance in Linux [120], Ren *et al.* identified several changes in latency for versions between 4.15 and 4.20;
- 5.0: an interesting version, as there has been some debate about the decrease of kernel performance on some macro-benchmarks (*e.g.,* see [57]);
- 5.4: it is a long term support release that will be maintained 6 years. This version also includes modifications for dealing with Linux performance [55, 57];
- 5.7: a recent version, more than a half-year after 5.4;
- 5.8: Linus Torvalds commented[1] *"IOW, 5.8 looks big. Really big."* and reported *"over 14k non-merge commits (over 15k counting merges), 800k new lines, and over 14 thousand files changed"*, suggesting an important and challenging evolution

[1] https://lore.kernel.org/lkml/CAHk-=whfuea587g8rh2DeLFFGYxiVuh-bzq22osJwz3q4SOfmA@mail.gmail.com/

to tackle.

As depicted in Table 6.1, the continuous evolution from 4.13 to 5.8 is significant in terms of numbers of added/deleted options, delta of the commits and the changes files. Note that those changes are computed for each release w.r.t. 4.13.

For all versions, we specifically targeted the x86-64 architecture, *i.e.,* technically, all configurations have values CONFIG_X86=y and CONFIG_X86_64=y. Overall, we span different periods during 3 years, with some modifications (security enhancements, new features) suggesting possible impacts on kernel non-functional properties (*e.g.,* size).

For each version, we build thousands of random configurations (see Table 6.1 column 5 - [Examples]). Owing to the computational cost, we balance the budget to measure at least and around 20K+ configurations per version. Such data is used to test the accuracy of a prediction model. We used TuxML[2], a tool to build the Linux kernel in the large *i.e.,* whatever options are combined.

TuxML relies on Docker to host the numerous packages needed to compile and measure the Linux kernel.

Docker offers a reproducible and portable environment – clusters of heterogeneous machines can be used with the same libraries and tools (*e.g.,* compilers' versions). Inside Docker, a collection of Python scripts automates the build process. We rely on randconfig to randomly generate Linux kernel configurations. randconfig has the merit of generating valid configurations that respect the numerous constraints between options. It is also a mature tool that the Linux community maintains and uses [95]. Though randconfig does not produce uniform, random samples (see Section 6.6), there is a diversity within the values of options (being *'y'*, *'n'*, or *'m'*).

Given .config files, TuxML builds the corresponding kernels. Throughout the process, TuxML can collect various kinds of information, including the build status and the size of the kernel. We concretely measure vmlinux, a statically linked executable file that contains the kernel in object file format.

The distribution of binary size in our dataset varies depending on the version. While the mean binary size on version 4.13 is 47 MiB, for other versions that mean value is between 89 MiB and 118 MiB. The minimum size for all version is around 10 MiB and the maximum around 2 GiB.

[2] https://github.com/TuxML/tuxml

Table 6.1 – Dataset properties for each version. The number of deleted/new features, delta commits, files changes are w.r.t. 4.13. † for versions 4.13 and 4.15, the build time (number of seconds to build one configuration) should be interpreted with caution since we used heterogeneous machines and did not seek to control their workload

Version	Release Date	LOC	Files	Examples	Seconds/config	Options	Features	Deleted features	New features	ΔCommits	Files changes
4.13	2017/09/03	16,616,534	60,530	92,562	271†	12,776	9,468	-	-	-	-
4.15	2018/01/28	17,073,368	62,249	39,391	263†	12,998	9,425	342	299	31,052	934,628
4.20	2018/12/23	17,526,171	62,423	23,489	225	13,533	10,189	468	1,189	104,691	1,972,020
5.0	2019/03/03	17,679,372	63,076	19,952	247	13,673	10,293	494	1,319	118,778	2,170,935
5.4	2019/10/24	19,358,903	67,915	25,847	285	14,159	10,813	663	2,008	181,308	3,827,025
5.7	2020/05/31	19,358,903	67,915	20,159	258	14,586	11,338	715	2,585	225,804	4,393,117
5.8	2020/08/02	19,729,197	69,303	21,923	289	14,817	11,530	730	2,792	242,381	4,681,313

Preprocessing

We distinguish between *options* and *features*, as an *option* is a variable in Linux kernel configuration, and a *feature* an independent variable from which the machine learning algorithm creates a model. The distinction is important as we made some manipulations over the dataset, in order to facilitate the learning. The first one is to put aside non-tristate options, which represent a very small subset (between 300 and 320 depending on the version). Most Linux kernel options values are *"yes"*, *"no"*, or *"module"*, hence the name "tristate" options. The second manipulation was to encode these option values into numbers to be processed by the algorithm. We observed that the values *"no"* and *"module"* had the same effect on the kernel size, so we encoded them as 0, and *"yes"* as 1. That is, we do not use "module" as an option value and we are not interested in module size. Then we put aside the options only having one value in the whole dataset, as it would bring no information from a statistical point of view and only make it longer for a algorithm to learn. Last but not least, we added a custom feature which is the sum of all activated options in the configuration, as it has proved important and helpful in a previous study of Acher *et al.* [5].

Performance Prediction Models

Extensive experiments on 4.13 (the oldest version of the dataset) showed what learning algorithms and hyper-parameters were effective and for which training set size [5]. Specifically, we chose gradient boosting trees (GBTs) that, according to [5], obtain the best results whatever the training set size. GBTs proved to be superior to linear regression models, decision trees, random forest, and neural networks for relatively small training set sizes (*e.g.,* 20K) but also for larger budgets (*e.g.,* 80K+).

We trained GBTs with 85.000 examples on version 4.13. We took care of finding the best hyperparameters, using grid-search, as it proved itself a quite important factor in the accuracy of the models. We relied on scikit-learn [20], a Python library that implements state-of-the-art machine learning algorithms. As a performance model only matches a specific set of features (here: the features of 4.13), we deleted features only contained in further, target versions (*e.g.,* 4.15).

Accuracy Measurement

Due to the wide distribution of Linux kernel binary sizes (from 7MB to around 2GB in our dataset), relying on some metrics such as Mean Absolute Error or Mean Squared Error can be biased, as an error of a few MB can be a big error for small kernels, and negligible for the biggest kernels. As numerous existing works (*e.g.,* [114, 150, 64, 63, 81]), we rely on a variant of Mean Absolute Error, but normalized in a percentage error, known as Mean Absolute Percentage Error (MAPE), computed as follows: $MAPE = \frac{100}{t}\sum_{i=1}^{t}\frac{|f(c_i)-\hat{f}(c_i)|}{f(c_i)}\%$, t being the number of predictions, $\hat{f}(c_i)$ the predicted values, and $f(c_i)$ the measured values (ground truth). Another advantage of this metric is that it is easily understandable and comparable, being a percentage.

To limit the impact of randomness in the experimentation, we performed the process of learning 5 times, leading to different training and test sets; we report the average error in our results.

Insights about evolution of options' importance

To better understand the evolution and possible degradation w.r.t. accuracy, we extract relevant information from the learning models created on the version 4.13 and the ones created on later versions. Specifically, we use the Feature Ranking List devised in chapter 4.

By comparing two feature ranking lists from two different versions, we can track which features have their importance evolving. We compare version 4.13 with (1) version 4.15 corresponding to an important drop in accuracy and subject to many changes due to meltdown/spectre[79, 90]; (2) version 5.8 the most recent version at our disposal.

6.2.2 Results

Figure 6.1 shows the degradation of models trained on the Linux Kernel version 4.13 by plotting their error rate (meaning lower is better) on later versions. The models get on average 5% MAPE on 4.13, and less than two versions after, on 4.15, the error rate is 4 times higher at 20%. It keeps this error rate for multiples version, at least up to 5.0, and goes even higher, at 32% for the version 5.7 and 5.8, i.e., an error rate 6 times higher. Note that the degradation do occur independently from the training set size, i.e., both with 20K and 85K. This is a crucial result that confirms the hypothesis of degradation

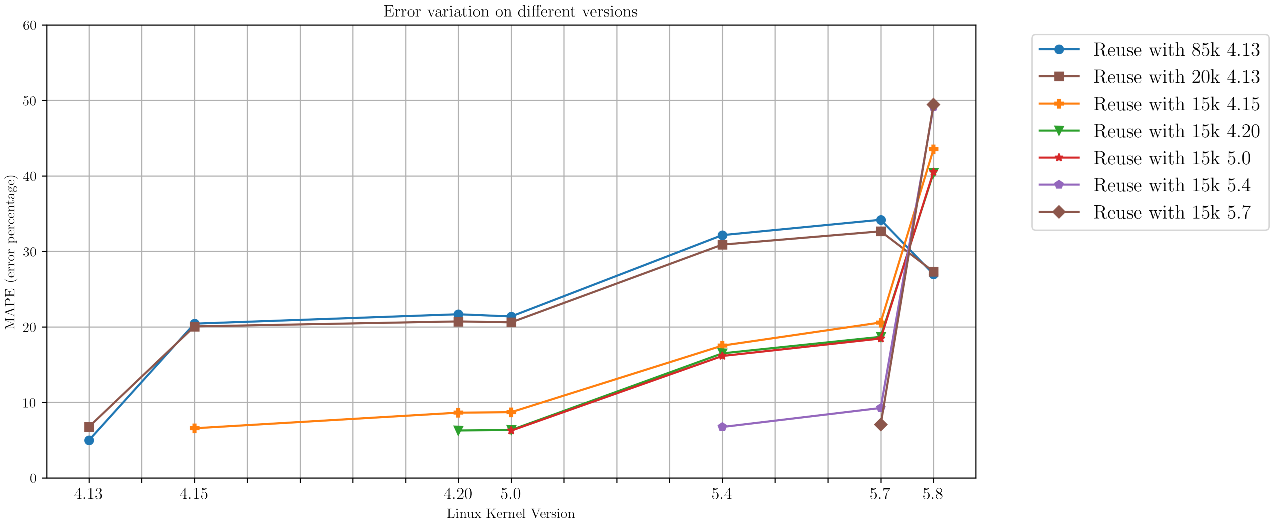


Figure 6.1 – Accuracy of prediction models, trained on 4.13, with training set size 20K and 85K, when applied on later versions.

over time, regardless of the training set size. Hence, we need a more sustainable solution like transfer learning [65, 82, 150].

It is worth noting though that the error rate stabilizes between 4.15 and 5.0. Hence, a hypothesis is that the evolution might be less costly between some versions. Unfortunately, a direct reuse of the prediction model is inaccurate for the early version 4.15 and subsequent ones (4.20, and 5.0). Moreover, the degradation slightly decreases between 5.7 and 5.8. A possible explanation is that the binary size distributions of 5.8 is closer to 4.13, at least for the way the basic transfer is performed. It also suggests an effect of the evolution between 5.7 and 5.8. Besides model reuse with 20K is more accurate than model reuse with much bigger training set size (85K) for all target versions, except 5.8. It is not what we would have expected for a learning model: a larger training set for the source model should lead to improved accuracy. This shows that despite the evolution changes both at the code level and options between the releases (see Table 6.1), the use of model reuse does not follow a logical or explainable reason from a machine learning point of view. Overall, simply transferring a prediction model is neither accurate nor reliable: the evolution of the configurations binary size is not captured.

We also measured the degradation of prediction models trained on other versions with 15K (see Figure 6.1). We can observe that the degradation is less immediate than with the version 4.13 but is still happening, especially on version 5.8 as the error percentage raises to 40% - 50%.

Insights about evolution and options. We first compare feature ranking lists from models trained on versions 4.13 and 4.15. We notice the following evolution patterns:

- **Features unchanged:** Numerous influential features do not change in importance from one version to another. Specifically, out of the top 50 features from both list (the top 50 representing 95% of the feature importance), 29 are the same. This is a key observation that allows one to envision the use of a model from one version to another, even partly;
- **Important new features:** 3 important features from the top 50 in version 4.15 did not exist in 4.13, such as CHASH (ranked #18), NOUVEAU_DEBUG_MMU (#21) or BLK_MQ_RDMA (#44);
- **Features losing importance:** 21 features from the top 50 that were important in 4.13 became unimportant in 4.15, meaning they do not impact kernel size anymore, such as RTL8723BE (from #48 to #8892), BT_BNEP_MC_FILTER (#38 to #4182) or AQUANTIA_PHY (#43 to #3348);
- **Features gaining importance:** 18 features from the top 50 that were unimportant in 4.13 became important in 4.15, such as HIPPI (from #6882 to #27), HAMACHI (from #7197 to #50) or NFC_MEI_PHY (from #4921 to #26).

We also compare feature ranking lists from models trained on versions 4.13 and 5.8 and find similar patterns:

- **Features unchanged:** Out of the top 50 features from both lists, 21 are the same. It is less than between versions 4.13 and 4.15, but there is still a substantial overlap;
- **Appearing important features:** 12 features from the top 50, such as DMA_COHERENT_POOL (ranked #12), DEBUG_INFO_COMPRESSED(#6) or AMD_MEM_ENCRYPT(#9);
- **Features losing importance:** 24 features from the top 50, such as ATH5K_TRACER (from #20 to #10423), DQL (#40 to #4153) or FDDI (#32 to #2532);
- **Features gaining importance:** 17 features from the top 50, such as LOCKDEP (#5639 to #22), PROVE_LOCKING (#2362 to #19) or SND_SOC_SI476X (#7075 to #31)

Our observations show that numerous features involved in different evolution patterns can cause the degradation of a prediction model. Prediction models trained on a specific version are challenged by 1) new features, which are unknown to old models, and 2) the changes in importance of the known features. Interestingly, we did not find important features that were removed between version 4.13 and 4.15. Another important insight is that a large subset of features remain important across versions: one can leverage this knowledge for building a prediction model. Overall, there is a potential to transfer a model from one version to another under the condition that new features together with the effects of important features are correctly handled. In fact, the insights drive the design of tEAMS: more details are provided in the next section.

Evolution does impact configuration prediction and degradation quickly occurs: from less than 5% to 20% (after only 4 months of evolution) to up to 32% for later versions. The reuse of a prediction model on different versions is not a satisfying solution, calling for other approaches.

6.3 Evolution-aware Model Shifting

In order to deal with the degradation issue highlighted in the previous section, we now present *evolution-aware model shifting (tEAMS)*, a method to transfer a prediction model for a given source version onto a target version.

6.3.1 Heterogeneous Transfer Learning Problem

Owing to the huge configuration space of the Linux kernel, it is impractical to re-learn at every release. To alleviate this issue, transfer learning was proposed as a new machine learning paradigm. It transfers the knowledge of a model built for a specific source domain (where sufficient measured/labeled data are available) to a related target domain (with little or no additional data) [36]. Figure 6.2 gives the general principle about how transfer learning works. Here, to make predictions over the Linux kernel version 5.8, we could directly reuse the performance model A built from the source version 4.13. Basically, the source model A is adapted to consider the aligned set of features from both source and target domains (i.e., model A'). Finally, the target model B is trained with only a few measured configurations in the target domain B plus the knowledge from the modified source model A'.

However, the evolution of Linux brings a specific scenario for transfer learning: configuration options for the source version (*e.g.,* 4.13) are not the same as further, target versions (*e.g.,* 5.8). It is worth noting in Table 6.1 that the number of options keeps increasing over versions. Between two versions, numerous options appear while some others

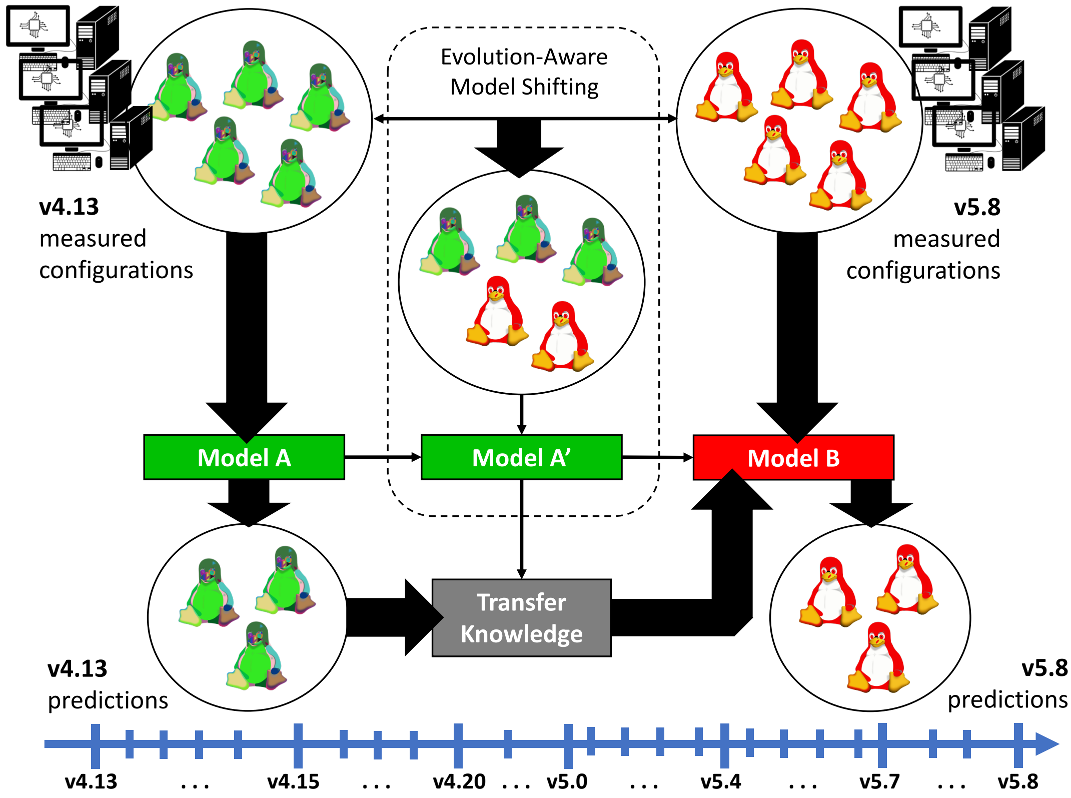


Figure 6.2 – An overview on how to predict the performance of Linux kernel configurations over versions 4.13 and 5.8.

disappear. If a model trained on a specific version cannot handle this new set of options, it will most likely have a negative effect on the accuracy of the size prediction. In terms of machine learning, since options are encoded as features (see Section 6.2), the feature spaces between the source and target version are non-equivalent.

This mismatch in feature space is the root of this instance of an *heterogeneous* transfer learning problem [36]. This case is potentially more challenging than *homogeneous* transfer learning in which the set of configuration options remains fixed over time. It can be a serious issue to not consider *e.g.,* new options that can have an impact on kernel size. Technically, prediction models assume that values (*e.g.,* 0 or 1) for a pre-defined set of features are given. If the set of features changes (as it is the case for the evolution of a configurable system), the predictions cannot be done. Hence numerous transfer techniques developed for configurable systems are simply not applicable.

The problem of heterogeneous transfer learning has recently caught attention in different domains (*e.g.,* image processing) with different assumptions and "gap" between the source and the target [36]. The intuition is that, for Linux, the shared set of features can be exploited to effectively transfer predictions.

6.3.2 Principles

The major challenge is to bridge the gap between the feature spaces. We rely on two steps: (1) feature alignment, which deals with the differences between features' sets among two versions; (2) the learning of a transfer function that map source features onto target size. For realizing *feature alignment*, we distinguish three cases:

— **commonality**: options that are common across versions (*i.e.,* options have the same names) are encoded as unique, shared features. There are two benefits: we can reuse a prediction model obtained over a source version *"as is"*, without having to retrain it with another feature scheme; we do not double the number of features, something that would increase the size of the learning model up to the point some learning algorithms might not scale. The anticipated risk is that some Linux options, though common across versions, may drastically differ at the implementation level, thus having different effects on sizes. We deal with this risk through the learning of a transfer function that aims to find the correspondences between the source and the target (see below);

— **deleted features**: options that are in the source version, but no longer in the target version: we add features in the target version with one possible value, *"0"* or *"1"*. Observations show that putting *"1"* as the default always gives slightly better accuracy.

— **new features**: options that are not in the source version, but only introduced in the target version: we ignore them when predicting the performance value since the source model cannot handle them, but we keep them in the target dataset.

Feature alignment alone is not sufficient; it is mainly a syntactical mapping at this step. There is a need to capture the *transfer function*, *i.e.,* the relationship between the source features, the source labels (kernel size of each configuration under source version), the target features, and the target labels. This transfer function should be learned. Owing to the complexity of the evolution, a "simple" linear function is unlikely to be accurate – our empirical results confirm the intuition, see next section. In contrast to existing works that rely on linear regression models for "shifting" the prediction models [82, 150, 65], we rely on more expressive learning models, capable of capturing interactions between source and target information.

Note that the feature alignment is a completely automated process and relies on the high similarity between the features spaces. In case of too disjoint features spaces, this solution would likely fail, and other solutions should be considered [36]

6.3.3 Algorithm

Figure 6.3 outlines the process for the TEAMS algorithm that gives the four key significant steps:

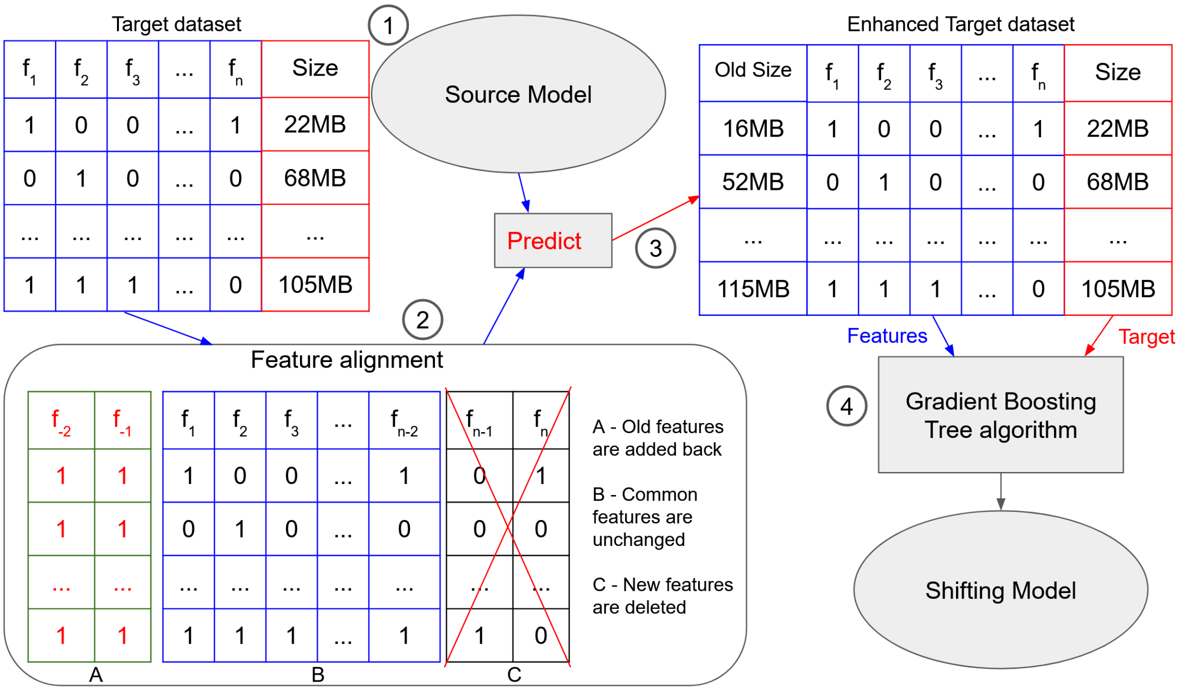
- ① Target dataset and Source model acquisition: Train or acquire a robust model on measurements from the source version and a dataset from the target version;
- ② Feature alignment: If the source and the target do not have the same set of options, an alignment of the feature spaces is applied (*e.g.,* as described in Section 6.3.2);
- ③ Target prediction: Using the source model, predict the value of the target data and add this prediction as a new feature in the target dataset;
- ④ Shifting model training: Using the enhanced target dataset, train a new model (*e.g.,* with a Gradient Boosting Tree algorithm capable of handling interactions).

Note that the source model is usually already trained beforehand, and its training step can be skipped in this case. Overall, our solution is fairly easy to implement and deploy. Moreover, once we train a source version the learning of the transfer function scales well (see next section).

6.3.4 Variant: Incremental Transfer

A possible variant of this technique is to use it in an incremental fashion, and to replace the source model by an already transferred model for a previous version. In the end, such a model consists of a source model, shifted multiple times in a row through multiple intermediate target versions until the final target version.

This variant could potentially give more accurate results, since the complexity of the transfer is spread over multiple models. The farther two versions are from each other, in terms of software evolution, the more performances-impacting changes can happen. A transfer model that handles two distant versions has to deal with all changes between these two versions at once, while in an incremental process, each model only has to deal with a fraction of the changes. On the other hand, we know that machine learning models are imperfect and error prone, even if the error is limited. Relying on a series of machine


Target dataset
1
Source Model
Enhanced Target dataset
Predict
3
2
Feature alignment
Features
Target
4
Gradient Boosting Tree algorithm
Shifting Model
A - Old features are added back
B - Common features are unchanged
C - New features are deleted
A
B
C

Target dataset

f_1	f_2	f_3	...	f_n	Size
1	0	0	...	1	22MB
0	1	0	...	0	68MB
...	...	...	...	...	...
1	1	1	...	0	105MB

Enhanced Target dataset

Old Size	f_1	f_2	f_3	...	f_n	Size
16MB	1	0	0	...	1	22MB
52MB	0	1	0	...	0	68MB
...	...	...	...	...	...	...
115MB	1	1	1	...	0	105MB

Feature alignment

f_{-2}	f_{-1}	f_1	f_2	f_3	...	f_{n-2}	f_{n-1}	f_n
1	1	1	0	0	...	1	0	1
1	1	0	1	0	...	0	0	0
...	...	...	...	...	...	...	...	...
1	1	1	1	1	...	1	1	0

Figure 6.3 – Model shifting process: ① Target dataset and Source model acquisition; ② Feature alignment for source model compatibility; ③ Enhance the dataset with predicted size; ④ Shifting model training

learning models can turn out to be risky, as these errors can be spread and amplified over the multiple models.

6.4 Effectiveness of Transfer Learning

Our goal is to evaluate the cost-effectiveness of our approach tEAMS in the context of Linux evolution. The effectiveness is the accuracy of the prediction model and its ability to minimize prediction errors as much as possible. If the source version and the target version have very little in common, configuration performance knowledge may not transfer well. In such situations, transfer learning can be unsuccessful and can lead to a "negative" transfer [36]. Specifically, we consider that the transfer is negative when learning from scratch directly onto the target version – without transfer and using only the limited measured target data available for transfer – leads to better accuracy than a transfer model with the same budget. Thus, we aim to answer the following research question:

(RQ2) What is the accuracy of our evolution-aware model shifting (tEAMS) compared to learning from scratch and other transfer learning techniques? The accuracy of tEAMS depends on the investment realized for creating or updating the prediction models. Specifically:

- the number of configuration measurements over the target model used to train the prediction model: non-transfer learning (*i.e.,* from scratch) uses the same training set and we can compare our results;
- the number of configuration measurements over the source version used to train the prediction model: from large training sets to relatively small ones;

Hence, we address RQ2 through different cost scenarios and we can identify for which investments tEAMS is effective.

6.4.1 Experimental settings

Dataset

For training and validating the prediction models, we use the same kernel versions and configuration measurements as in Section 6.2 and Table 6.1.

Training size for targeted versions

We vary the number of configurations' measurements amongst the following values $\{1K, 5K, 10K\}$. $5K$ corresponds to around 5% of the 95K configurations in the dataset of 4.13: it is representative of a scenario in which a relatively small fraction is used to update the model for a target version. As we have invested around 20K per version, we needed to take care of having a sufficiently large testing set for computing the accuracy. In particular, we cannot use the whole configuration measurements since otherwise we cannot simply compute the accuracy of the models. We stop at $10K$ since then the testing set can be set to around $10K$ too. Moreover, we repeat experiments 5 times with different training sets and report on standard deviations.

6.4.2 Baseline Methods and Parameters

Source prediction model for tEAMS

We use a prediction model trained with 4.13. It is the oldest version in our dataset and as such, we investigate an extreme scenario for the evolution and potentially the most problematic for transfer learning. As in Section 6.2, we rely on GBTs, the most accurate solution whatever the training set size is. We train GBTs over 4.13 with two different budgets: 85K configurations and 20K configurations. Hereafter, we call these prediction models 4.13_85K and 4.13_20K respectively.

Incremental tEAMS

We use the incremental method in the same way as without increment, only changing the base model for each increment by the model trained on the previous version. We also have two different series of increment, one based on the 4.13_85K model, the other on the 4.13_20K model. For instance, the first series starts with the transfer from model 4.13_85K to version 4.15 with a shifting model T4.15. This process creates a prediction model 4.15 composed of the two models: 4.15 = T4.15(4.13_85K). The next step is to transfer that model to version 4.20: 4.20 = T4.20(4.15). At the end of that series, we have a model looking like this:

5.8 = T5.8(T5.7(T5.4(T5.0(T4.20(T4.15(4.13_85K))))))

Learning from scratch

The most simple way to create a prediction model for a given version is to learn from scratch with an allocated budget. We use the GBTs algorithm to create prediction models from scratch, for each version of our dataset. As previously stated, the study in [5] shows that GBTs were a scalable and accurate solution compared to other state-of-the-art solutions. Furthermore, the superiority of GBTs is more apparent when small training sets are employed. This quality of GBTs is even more important when learning for the target version where the budget for updating the model is typically limited – we investigate budget with less than 20K measurements (see above *"Training size for targeted versions"*). We replicated the experiments of 4.13 on other versions: linear models, decision trees, random forests, and neural networks give inferior accuracy compared to GBTs, especially for small sampling size (*e.g.,* 10K). Thus, we do not report results of other learning algorithms and keep only GBTs, the strongest baselines for learning from scratch or for transfer learning techniques.

tEAMS with linear-based transfer function

In most state-of-the-art cases, model shifting processes use a simple linear learning algorithm to create a shifting model and they are performing quite well (e.g. [150, 61]). We rely on such a linear transfer function and also apply feature alignment as part of tEAMS.

6.4.3 Results

Figure 6.4 depicts the evolution of the MAPE for the reuse of the model 4.13_85K (*i.e.,* 4.13 with a 85K of training set), and the 4 studied techniques trained using 5K examples. We can quickly see that linear model shifting has more than 40% MAPE over all versions and is not accurate at all. It is surprisingly the worst by far, in particular, in comparison with the direct reuse of the prediction model. The standard deviation for Linear model shifting is between 1.5 and 3, while all other techniques are much more stable with a standard deviation always at 0.1 or less. Moreover, learning from scratch with 5K examples allows to create models having an MAPE between 8.2% and 9.2% quite consistently. On the other side, tEAMS with the same budget offers a lower MAPE from 5.6% on version 4.15 to 6.8% in version 5.8 with a peak at 7.1 in version 5.7. It is worth noting that tEAMS MAPE increases a little bit at each version. However, the increase

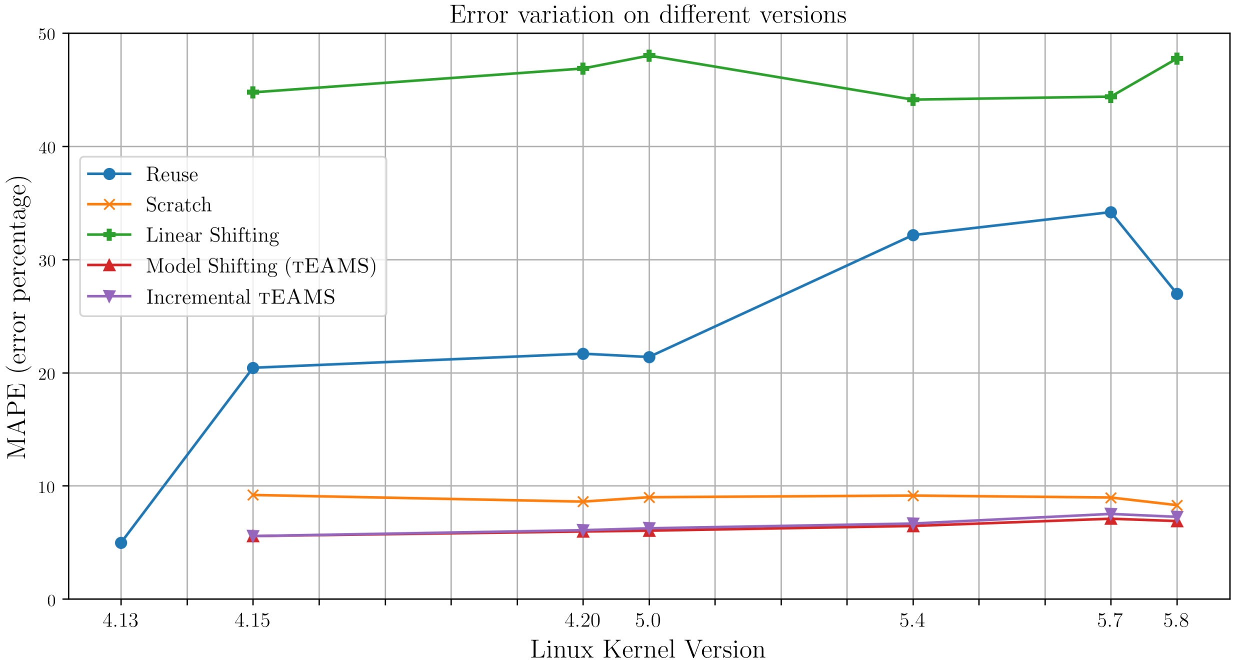


Figure 6.4 – Accuracy of tEAMS 4.13_85K compared to other techniques using 5K examples for the target

is not significant and remains low.

Comparing tEAMS with its incremental variant shows a very slight but constant advantage over the variant, which also show better results then learning from scratch.

Table 6.2 gives the results for MAPE with combinations of different models used (4.13_20K and 4.13_85K) and training set size (1K, 5K, 10K) for the scratch baseline,tEAMS and Incremental tEAMS. The other techniques already performed poorly, Table 6.2 hence focuses on the three competing solutions in Figure 6.4. We now report on these results.

Impact of training set size over target. As illustrated in Tables 6.2, when decreasing the training transfer set for the newer versions to 1K example (1% of the original set), the MAPE increases to 14.9%-16.7% depending on the version. Whereas, the MAPE for tEAMS only increases to 6.7%-10.6%, with the same trend consistently increasing MAPE over time (and versions). For Incremental tEAMS, the error rate increases faster up to 13.3 on version 5.8. On the other hand, if we increase the training set to 10K (10% of the original set), accuracy when learning from scratch gets better, with 7.0% to 7.7% MAPE. For tEAMS, the accuracy also gets better, from 5.2% MAPE on version 4.15 to 6.1% on version 5.7 and then slightly further improves to 6.1% on version 5.8. We observe the same trend for Incremental tEAMS, going up to 6.5% on 5.7 and then to 6.2 on 5.8.

Impact of tEAMS source model. We measured the same variations using model 4.13_20K as the source model, which was built from 20,000 examples instead of 85,000. This affects tEAMS by slightly increasing the MAPE. In particular, we observe that for tEAMS: 1) with 1K, the MAPE varies from 8.5% to 11.6%, 2) With 5K, it varies from 6.7% to 7.9%, and 3) with 10K, it varies from 6.2% to 6.7%. Whereas for learning from scratch, we observe that: 1) with 1K, the MAPE varies from 14.9% to 16.7%, 2) With 5K, it varies from 8.3% to 9.2%, and 3) with 10K, it varies from 7.04% to 7.67%. Therefore, our results show that tEAMS outperforms the two baselines, regardless of the size of training sets. In this situation, Incremental tEAMS also shows slightly better results than tEAMS in some cases. At 10k, Incremental tEAMS beats tEAMS on all versions except 5.7, and at 5k, only for versions 4.20 and 5.0. Given the fair increase in error rate at 1k, Incremental tEAMS seems to be very sensitive to higher error rate from previous versions.

Computational cost of training. We performed our experiments on a machine with an Intel Xeon 4c/8t, 3,7 GHz, 64GB memory. Training from scratch with 1K, 5K and 10K examples take respectively 21, 195 and 407 seconds. Learning with tEAMS takes a little more time with 60, 288 and 604 seconds for the same number of examples. The training time of tEAMS for updating a prediction model is thus affordable and negligible compared to the time taken to build and measure the kernel configurations. The overall cost of training is mainly due to the training of the source model (details can be found in [5]) which is done only once. As a final note, building kernels and gathering configurations data (see Table 6.1) is by far the most costly activity – the time needed to train the prediction model out of data through either transfer learning or from scratch (a few minutes) is negligible.

tEAMS and Incremental tEAMS are more accurate solutions than learning from scratch and linear model shifting to predict Linux Kernel size on different versions. Also, Incremental tEAMS shows results mostly worse than tEAMS and without significant improvement. Even with different source models and training set sizes, **tEAMS keeps better and acceptable accuracy with 6.9% MAPE on the latest 5.8 version leveraging model trained on 3 years old data.**

Table 6.2 – MAPE for Scratch and TEAMS, with varying source models and training set sizes for the target

Version	Scratch			TEAMS						Incremental TEAMS					
				4.13_20K			4.13_85K			4.13_20K			4.13_85K		
	1k	5k	10k	1k	5k	10k	1k	5k	10k	1k	5k	10k	1k	5k	10k
4.15	16.72	9.19	7.46	8.46	6.69	6.21	6.73	5.56	5.19	8.46	6.69	6.21	6.73	5.56	5.19
4.20	16.39	8.60	7.12	8.85	6.94	6.22	7.64	5.96	5.44	9.49	6.89	6.15	8.39	6.08	5.46
5.0	15.50	8.99	7.07	9.14	7.04	6.34	7.80	6.03	5.48	10.32	6.99	6.15	8.84	6.24	5.63
5.4	16.06	9.14	7.67	9.76	7.06	6.39	9.01	6.45	5.71	11.64	7.23	6.10	10.56	6.66	6.07
5.7	15.63	8.96	7.59	11.56	7.85	6.69	10.13	7.09	6.12	13.77	7.90	6.75	12.57	7.51	6.50
5.8	14.91	8.29	7.04	11.47	7.27	6.41	10.62	6.88	6.06	13.82	7.58	6.39	13.29	7.26	6.19

6.5 Discussions

Integration of tEAMS in the Linux project. Our results suggest that we can now provide accurate prediction tools that can be used on 7 versions spanning 3 years of period. In fact, it is an additional advantage of tEAMS compared to non-transfer learning methods that stick to a specific version. This ability is important, since older versions of the kernel are still widely used. Long term support (LTS) releases are particularly relevant since (1) they are maintained over a period of 6 years (2) they are considered by other related communities, like the Android one. The version 5.4 of our dataset is an LTS release and tEAMS can be used in this context.

Linux practitioners interested in a specific release, not present in our dataset, could well invest some resources to obtain a new model. For example, we have not considered version 4.19 (a LTS release). Non-transfer learning methods would be unable to reuse their models while tEAMS provide state-of-the-art cost-accuracy results.

We envision to integrate tEAMS as part of the ongoing continuous integration effort on the Linux kernel. We have released a tool, called kpredict[3], that predicts the size of the kernel binary size given only a .config file. kpredict is written in Python, available on pip, and supports all kernel versions mentioned in this article. A usage example is as follows:

```
> curl -s http://tuxml-data.irisa.fr/data/configuration/167950/config -o .config
> kpredict .config
> Predicted size : 68.1MiB
```

whereas the actual size of the configuration (see http://tuxml-data.irisa.fr/data/configuration/167950/ for more details) is 67.82 MiB.

Recently KernelCI [76], the major community-effort supported by several organizations (Google, Redhat, etc.), has added the ability to compute kernel sizes and this functionality is activated by default, for any build. Hence tEAMS will benefit from such data. Besides, the current focus of KernelCI and many CI effort is mostly driven by controlling that the kernels build (for different architectures and configurations). It is not incompatible with the prediction of kernel sizes since we did not employ a sampling strategy specifically devoted to this property. We rely on random configurations that are used to cover the kernel and find bugs (see *e.g.,* [4]). In passing tEAMS can benefit from kernel sizes' data while the CI effort continues to track bugs.

Besides, the development cost of integrating this process is fairly small and requires little maintenance. All steps in the process, including feature alignment and adaptation

[3] https://github.com/HugoJPMartin/kpredict/

of the learning model to the new version, are fully automated and do not require the intervention of kernel developers or maintainers. Our engineering experience with kpredict is that the overhead of implementing transfer learning steps is not much higher than learning from scratch.

Is the cost of tEAMS affordable for Linux? Under the same cost settings, tEAMS shows superior accuracy compared to non-transfer learning and other baselines. Still, one can wonder whether measuring thousands of configurations is practically possible, especially when there is a new release. Specifically, results show that with 5K measurements we can obtain almost stable accuracy. So, is the build of 5K affordable for Linux? To address this question, we investigated the number of builds realized by KernelCI and asked the leaders of the project. At the time of the publication, KernelCI is able to build 200 kernels in one hour. We then analyze the retrospective cost of measuring 5K as part of our experiments. On average, our investments sum around 260 seconds per machine whatever the version (see Table 6.1, page 108). That is, with 15 machines (16 cores) full time during 24 hours, we can already measure 5K configurations. As a side note, the numbers should be put in perspective: Linux exhibits thousands of options (see Table 6.1) and has a large community with many contributors and organizations involved. Overall, though the cost itself is affordable from a computational point of view, there is a tradeoff to find between (1) accuracy; (2) value for the community. This tradeoff should be considered for any configurable system. In any case tEAMS has demonstrated being the most cost-effective approach.

tEAMS versus Incremental tEAMS As we investigated the differences in accuracy between the two techniques, we observed that both show very similar results when using a significant amount of examples, while Incremental tEAMS drops in accuracy when having a reduced training set. This shows that it is very sensitive to low accuracy models in series. On the other hand, if the investment in measurements is sizeable, it would be wise to consider it.

When it comes to using and sharing a model, one should consider the cost of having an incremental solution as it comes at a cost. The first is the size of the model, as a solution composed of a multitude of models also means a larger size of the file to share. The other is the time taken to predict a value, which can be a critical point depending on the context. If using one or two models in series usually take less than a second, an incremental solution can take few seconds. If the model is used in a user interface, such an high response time can be considered a severe drawback.

tEAMS and transfer functions As part of tEAMS several transfer functions can be considered for shifting a prediction model. Results show that simple linear regression is clearly not effective. An alternative is to add interactions' terms as part of the linear regression. As stated in [81], for p options, there are p possible main influences, $p \times (p-1)/2$ possible pairwise interactions, and an exponential number of higher-order interactions among more than two options. In the worst case, all 2-wise or 3-wise interactions among the 9K+ options are included in the model, which is computationally intractable. Even if a subset of options is kept, there is a combinatorial explosion of possible interactions. Another possibility is to consider linear models with regularized regression. However, prior experiments [5] for training prediction models over Linux 4.13 showed that Lasso [144] or ElasticNet have severe limitations in terms of accuracy. We chose GBTs for their inherent ability to capture non-linear behavior or options' interactions and empirical results show high accuracy. However we do not claim that the use of GBTs is the best solution. In fact, as tEAMS is capable of handling other transfer functions, we plan to consider other learning techniques (*e.g.,* neural networks) and investigate tradeoffs between accuracy, computational cost, and interpretability.

When and how should we update tEAMS? Results show that our solution of transfer learning works well over a long period of three years from 2017 to 2020. Meaning that up to today, transferring is better than re-learning from scratch. However, eventually, in future evolution and releases, say in 2021 or 2022, transfer learning might degrade. Therefore, at some point, one must learn a new prediction model as the source on which to apply further transfer learning with our solution. Quantifying exactly when one must re-learn the performance model is out of the scope of this chapter and is left for future work. In practice, developers can set a limit of acceptable error rate under which transfer can be still applied and beyond which re-learning a prediction model is necessary.

Is tEAMS applicable to other configurable systems? We investigated in this chapter a case of heterogeneous transfer learning where our considered configurable system evolves in time and space, i.e., the set of options changes on every release over time. Therefore, other configurable systems that evolve while the set of options also changes can be ideal candidates to leverage our results and tEAMS. For example, `curl` is a widely used configurable system that has now more than 200 options. `curl` does evolves frequently[30], and has performance concerns. GCC[47] and Clang[27] (compilers), Apache Cassandra[10] (database), Amazon EC2 (Web cloud service), OpenCV [108] (image processing), Kafka[70] (distributed systems) are other examples of configurable systems that continuously add or remove options, maintain their code base at a fast pace, and with strong performance requirements. These systems have active user communities and have already been considered in the context of performance prediction of configurable systems (see [114, 67, 142, 130, 7, 12]), but without considering the evolution of their options' sets. In the future, we plan to replicate our study for such subject systems.

6.6 Threats to Validity

Threats to *internal validity* are related to the sampling used as training set for all experiments. We deliberately used random sampling to diversify our datasets of configurations. To mitigate any bias, for each sample size, we repeat the prediction process 5 times. For each process, we split the dataset into training and testing which are independent of each other. To assess the accuracy of the algorithms, we used MAPE that has the merit of being comparable among versions and learning approaches. Most of the state-of-the-art works use this metric for evaluating the effectiveness of performance prediction algorithms [114]. Another threat to internal validity concerns the (lack of) uniformity of randconfig. Indeed randconfig does *not* provide a perfect uniform distribution over valid configurations [95]. The strategy of randconfig is to randomly enable or disable options according to the order in the Kconfig files. It is thus biased towards features higher up in the tree. The problem of uniform sampling is fundamentally a satisfiability (SAT) problem. However, to the best of our knowledge, there is no robust and scalable solution capable of comprehensively translating Kconfig files of Linux into a SAT formula, especially for the new versions of Linux. Second, uniform sampling either does not scale yet or is not uniform at the scale of Linux [117, 21, 22, 42].

Looking at the decrease gap in accuracy between learning from scratch and TEAMS the more examples are given for training, we can expect that at some point, learning from scratch would be better than TEAMS. We did not investigate further as it would require an important effort to gather so much more examples on all the studied versions, but it would be interesting to know *if* such turning point exists and *where* is that turning point in term of number of examples. While we highlight how good transfer is with limited data, we did not investigate enough how good it is with a lot of data.

Regarding the analysis with the feature ranking list, some problems may arise that can threaten results of Section 6.2.2. First, the importance is based on a tree structure, and even if we average the ranking out of multiple models, the way the tree is built can create a bias. Second, the importance does not show if the feature has a positive or negative impact. Hence, there is a risk of having a feature evolving that gets the same importance value but with inverse impact from one version to another. Other interpretable techniques are needed to gain further insights.

Threats to *external validity* are related to the targeted kernel versions, targeted architecture (x86), targeted quantitative property (size), and used compilers (gcc 6.3). We have spanned different periods of the kernel from 2017 to 2020, considering stable versions and covering numerous evolutions (see Table 6.1). We could consider minor releases, but the practical interest for the Linux community would be less obvious. We are expecting similar results since the evolution in minor release is mostly based on patches and bug fixings. We have considered the most popular and active architecture (x86). Another architecture could lead to different options' effects on size. The idea of transfer learning could well be applied in this context and is left as future work. A generalization of the results for other non-functional properties (*e.g.,* compilation time, boot time) would require additional experiments with further resources and engineering efforts. In contrast to *e.g.,* compilation time, the size of a the kernel is not subject to hardware variations and we can scale the experiments with distributed machines. Hence, we focused on a single property to be able to make robust and reliable statements about whether learning approaches can be used in such settings. There is also a clear interest for the Linux community (see Section 6.5). Our results suggest we can now envision to perform an analysis over other properties to generalize our findings. Finally, as we used Linux kernel, we do not generalize to all configurable systems, especially to those with few options and small-medium code size. However, this is not the goal of this study since we purposely targeted Linux due to its complexity (in options and LOC). Further experimentation on other case studies remains necessary.

6.7 Conclusion

In this chapter, we showed that the evolution of Linux has a noticeable impact on kernel sizes of configurations with a large-scale study spanning 7 versions over 3 years and 240K+ configurations. As a result, a size prediction model learned from one specific version quickly becomes obsolete and inaccurate, despite a huge initial investment (15K hours of computation for building a training set of 90K configurations). We developed a heterogeneous transfer evolution-aware model shifting (tEAMS) learning technique.

It is capable of handling new options that appear in new versions while learning the function that maps the novel effects of shared options among versions. Our results showed that the transfer of a prediction model leads to accurate results (the prediction error (MAPE) remains low and constant) without the need of collecting a very large corpus of measurements' configurations. With only 5K configurations, we could transfer the model made in September 2017 for the 5.8 version released in August 2020.

Linux is an extreme case of a highly complex configurable system that rapidly evolves. Though not all systems have $\approx$ 14.500 options and such a frequency of commits, many software systems face the same problem of dealing with variability in space (variants) and time (versions). The increasing adoption of continuous integration in software engineering has exacerbated the problem of adding, removing, or maintaining configuration options (being realized as feature toggles, command line parameters, conditional compilation, etc.). The fact that transfer learning works for Linux is reassuring, which has a great potential impact on Linux developers and integrators. There is hope that the effort made for one version of a software system can be reused through transfer.

Chapter 7

Conclusion and Perspectives

7.1 Conclusion

As Software Product Lines become prevalent in many software development environments and grow bigger and bigger, various problems appear. One of them is to understand the systems well enough to predict their performances for all possible variants. Relying on expert knowledge is limited and prone to error, especially in a context of constant evolution. Over the last decade, researchers focused on leveraging machine learning techniques to create reliable performance models. However, none of the studies evaluated their new techniques on systems with more than a few dozens of options, leaving a grey area as to know how such techniques would perform on colossal configurations spaces from systems with hundreds or thousands of options.

The first contribution of this thesis was to enlighten this grey area, and we have shown that most learning techniques are either poorly accurate, or do not scale to the thousands of options of the Linux Kernel, in particular the learning techniques tailored to handle SPL specifics. On the contrary, we found out that Decision Tree and its ensemble techniques, as well as neural networks, are able to handle thousands of options in input and their interactions at the same time.

The second contribution leverages Random Forest to create a list of features ordered by their importance, the Feature Ranking List, then used in a feature selection process to filter out the less useful features. This feature selection can be used by any algorithm, with mixed results. However, for tree-based algorithms and neural networks, the feature selection increases the accuracy of the models, and greatly decreases the training time. We also investigated the quality of the Feature Ranking List, by submitting it to experts who approved it with respect to their knowledge. Finally we proposed to improve the Feature Ranking List stability through the use of an ensemble of Random Forest, in order to decrease the variance in the ranking.

The third contribution compares different strategies to perform automated performance specialization. The first iteration of the concept only used classification models to synthesize the rules needed for the specialization. We considered two new approaches. First, we used the other way to leverage Decision Trees, as a regression model. Then, we proposed an hybrid approach, tailored toward the specialization problem. We also used the feature selection process detailed in the second contribution and measured its influence on accuracy and training times. All approaches gave good results on most of the subject systems, without any one being better than the others overall. Training time are about a few milliseconds for systems with a few dozens of options, up to more than one minute for Linux. The feature selection gave mixed results in term of accuracy gain, though it mostly had a positive but limited impact. In term of training time reduction, it brought the training time on Linux to the order of the second, which is a massive improvement.

The final contribution explored the problem of maintaining the performance models over time, and more specifically across versions. First, we studied the model trained on the version 4.13 of the Linux Kernel and the variation in its accuracy on six later versions spanning 3 years from version 4.15 to 5.8, and reported that the model quickly looses accuracy. To mitigate the effect of evolution on the accuracy, we considered transfer learning with the model shifting approach to leverage the deprecated model knowledge to build a model for new versions at lower cost. We have shown that transfer learning offers a solution to maintain the prediction model at high accuracy even on versions years later with only a fraction of the cost of the original model, and a better accuracy than learning from scratch for the same cost. We also considered an incremental approach of the model shifting that is able to give slightly better accuracy when given enough data.

In fine, we can note that systems with colossal configurations spaces such as the Linux Kernel are particularly absent from the state-of-the-art in performance prediction of Software Product Lines, and it appears that most solutions, and particularly specialized ones, do not scale to such problem. We demonstrated that tree-based learning algorithms are (1) able to scale in colossal configurations spaces and (2) able to handle the interaction problem specific to SPL. They are also interpretable in ways that allow the creation of a Feature Ranking List to perform a feature selection by filtering, and to extract rules needed to automate performance specialization. We have shown that performance models suffer from software evolution and proposed a transfer learning by model shifting making possible to maintain acceptable accuracy at low cost over at least a few years.

7.2 Perspectives

While this thesis answered some questions about learning on colossal configuration spaces, it also rises many more and opens perspectives for future works. The first perspective we propose are general to the whole thesis, then we will be more specific chapter by chapter.

Beyond binary size. All along this thesis, we focused particularly on the prediction of binary size of the Linux kernel. While we defend the relevance of predicting the size of the kernel, this focus is mainly a technical limitation of TuxML, the tool for compiling and measuring the kernel, which is not mature enough *yet* to handle other metrics, such as boot time or energy consumption. However, binary size has one important advantage, the low impact of deep variability, as it is not influenced by hardware, workload or environment influence. This was raised recently as a very important problem calling previous works, with fixed or undisclosed hardware, workload or environment, into question. Even though, we know that other factors can impact the binary size such as the version of the compiler, a problem we hope to tackle with the help of transfer learning.

Beyond Linux. The case of the Linux Kernel has been a central theme of the thesis, both because of the opportunity given by the TuxML project and because of the excellent use case it is. However, Linux is not the only system with thousands of options, such as many systems from the automotive industry that has been particularly used as subject systems on feature modelling research. Investigating other similar cases are needed to ensure the generalization of the methods proposed.

7.2.1 Feature Selection

In Chapter 3, we saw that the wrapper methods used to select features in [131] lead to an explosion of computing time when used on thousands of options, especially given the particular way the interactions are encoded. The tree-based feature selection explored in Chapter 4 is only one of the many existing solutions [23], which should be compared in term of both accuracy gain and resources needed to perform the selection.

Stability. The stability problem of the tree-based feature selection has been a concern. While partially solved, it deserves a deeper study, with different stability metrics to better understand it, but also trying out different approaches than Random Forest. The impact of the training set size is to be determined as well.

Finding the sweet spot. While the Feature Ranking list is used for feature selection, we did not find an efficient way to determine the right number of features to include (the sweet spot) other than creating models with a wide range of number of features and comparing them. A first step to improve that would be to perform a less naive approach in gradient descent. Another possibility would be to determine it *a priori*, which could be a long shot given that we found different sweet spots depending on the algorithm used.

7.2.2 Automated Performance Specialization

In Chapter 5, we explored the process of Automated Performance Specialization and multiple learning strategies to synthesize the needed rules. However, we focused only on the accuracy of the model to validate the approach and this should be taken further on different aspects.

Safety and flexibility. The consideration of safety and flexibility can be very important to a user. If we know it is technically possible with Decision Trees to tweak the decision rules to tip the scale in favor of one over the other, we still need to make more experiments to compare cost and improvement with this process.

Going further with constraints. We only considered the extracted rules to be used by software and dismissed their actual human readability, which could also be of interest for practitioners. Another research direction requiring human studies and controlled experiments could focus on that aspect of learning-based rule synthesis. The use of feature selection might also be of interest here, since rules and constraints will operate over a limited set of options. More aggressive pruning strategies can also be envisioned with tradeoffs between accuracy and interpretability.

Active learning. In our experiment, we only considered a single step of learning to create a model. However, we can consider multiple steps of learning, and leveraging the interpretability of the Decision Trees to find the most error-prone rules in order to refine them by sampling around the misclassified configurations. It is also possible to use the Feature Ranking List to focus the sampling toward the most influential options and avoid the ones without influence.

7.2.3 Transfer Learning for Prediction across versions

While we successfully used transfer learning to create robust prediction models across different versions in Chapter 6, there are still some doubts to dispel and improvements to be made.

Minor version influence. In the study, we focused our problem space on major versions (4.15, 5.4, 5.8 ...), but we do not know the impact of minor versions (5.8.1, 5.8.2 ...) on the prediction models, and especially if transfer is needed even between minor versions, which could negatively impact the viability of our approach given the costs of measurements. However, given the relatively low loss of accuracy between two close versions outside of exceptional cases, odds are that this should not be necessary.

Targeted sampling. In [64], Jamshidi *et al.* use learned model to focus the sampling toward important features when transferring between different environments. While the solution is developed in a homogeneous context unlike the problem of transfer between versions, it could help in reducing the costs of measurement, and studying the feasibility and efficiency of such approach should be a good way to improve our method.

Commit mining. When developing our method for model shifting, we took a complete black-box approach regarding the target system, and this sometimes leads to arbitrary decisions to be made. For instance, during the feature alignment, we set all the old features value to 1 as a default value. Features can evolve in many ways, such as appearing or disappearing, but also they can be locked in a specific value, split among multiple new features, merged into one, or even just be renamed. Our approach can only handle the first two ways. By mining the commits, we hope to gather meaningful insight that can help to better handle these evolving features. Such insight can also be leveraged for targeted sampling as mentioned previously.

Other algorithms. In our study, we only used a Gradient Boosting Tree to build a source model. Since the proposed method for model shifting is completely algorithm-agnostic, in the sense that in theory any algorithm could be used to build both the source and the shifting models, we could investigate more algorithms. We could also investigate the interaction between the source and shifting models.

Feature selection. The interaction between the feature selection and the model shifting method is quite uncertain. First, our approach of feature alignment should be revised, as simply putting away the features not in the source model, which only has a few selected features, could seriously impact the accuracy. Then, we could consider multiple ways to transfer the Feature ranking List: creating a new list from the target dataset, at the risk of having a less reliable list due to stability concerns, or leverage a Feature Ranking List from the shifting model to update the list from the source model, which can be tricky due to the lack of expressiveness of the feature importance metric. We should also consider different cases, for handling cases when training the source model is possible with information about the target version features, and when it is not.

Multidimensional transfer. We studied the use of model shifting for the problem of evolution in time, while other works focused on different goal, such as transfer between environments [69]. When we explored the deep variability concerns [86], we figured that when we train a prediction model, we usually focus on a very specific context (version of the systems, version of the compiler, configuration of the compiler [87], environment ...) and the use of transfer learning becomes obvious. However, transferring along only one dimension at a time is not enough, and we should explore the use of multiple transfer strategies at once. Given the efficiency of the model shifting methods, a sequential combination of shifting could be a good initial research direction.

Bibliography

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski, « 42 variability bugs in the linux kernel: a qualitative analysis », *in*: *ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 421–432 (cit. on p. 41).

[2] Iago Abal et al., « Variability Bugs in Highly Configurable Systems: A Qualitative Analysis », *in*: *TSE Methodol.* 26.*3* (Jan. 2018), 10:1–10:34, ISSN: 1049-331X (cit. on p. 41).

[3] Ebrahim Khalil Abbasi et al., « The Anatomy of a Sales Configurator: An Empirical Study of 111 Cases », *in*: *Advanced Information Systems Engineering - 25th International Conference, CAiSE 2013, Valencia, Spain, June 17-21, 2013. Proceedings*, 2013, pp. 162–177 (cit. on p. 45).

[4] Mathieu Acher et al., *Learning From Thousands of Build Failures of Linux Kernel Configurations*, Technical Report, Inria ; IRISA, June 2019, pp. 1–12, URL: `https://hal.inria.fr/hal-02147012` (cit. on p. 124).

[5] Mathieu Acher et al., « Learning Very Large Configuration Spaces: What Matters for Linux Kernel Sizes », *in*: (2019) (cit. on pp. 89, 90, 109, 120, 122, 126).

[6] Mathieu Acher et al., « VaryLaTeX: Learning Paper Variants That Meet Constraints », *in*: *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*, ACM, 2018, pp. 83–88 (cit. on pp. 35, 80, 86).

[7] Omid Alipourfard et al., « Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics », *in*: *14th* {*USENIX*} {*NSDI*} *17*, 2017, pp. 469–482 (cit. on p. 127).

[8] Juliana Alves Pereira et al., « Sampling Effect on Performance Prediction of Configurable Systems: A Case Study », *in*: *International Conference on Performance Engineering (ICPE 2020)*, 2020, URL: `https://hal.inria.fr/hal-02356290` (cit. on pp. 22, 34).

[9] Benoit Amand et al., « Towards Learning-Aided Configuration in 3D Printing: Feasibility Study and Application to Defect Prediction », *in*: *VaMoS 2019 - 13th International Workshop on Variability Modelling of Software-Intensive Systems*, Leuven, Belgium, Feb. 2019, pp. 1–9, URL: https://hal.inria.fr/hal-01990767 (cit. on pp. 35, 80, 86).

[10] *Apache Cassandra*, URL: https://cassandra.apache.org/doc/latest/new/index.html (cit. on p. 126).

[11] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori, « Automatic Detection and Removal of Conformance Faults in Feature Models », *in*: *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2016, pp. 102–112, DOI: 10.1109/ICST.2016.10 (cit. on p. 35).

[12] Liang Bao et al., « AutoConfig: automatic configuration tuning for distributed message systems », *in*: *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ACM, 2018, pp. 29–40 (cit. on p. 127).

[13] David Benavides, Pablo Trinidad Martı************************************n-Arroyo, and Antonio Ruiz Cortés, « Automated reasoning on feature models. », *in*: *International Conference on Advanced Information Systems Engineering (CAiSE)*, vol. 5, 3520, Springer, 2005, pp. 491–503 (cit. on p. 26).

[14] Thorsten Berger et al., « A Study of Variability Models and Languages in the Systems Software Domain », *in*: *IEEE Transactions on Software Engineering* 99.*PrePrints* (2013), p. 1, ISSN: 0098-5589, DOI: http://doi.ieeecomputersociety.org/10.1109/TSE.2013.34 (cit. on p. 39).

[15] Thorsten Berger et al., « Variability Modeling in the Real: A Perspective from the Operating Systems Domain », *in*: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, Antwerp, Belgium: Association for Computing Machinery, 2010, pp. 73–82, ISBN: 9781450301169, DOI: 10.1145/1858996.1859010, URL: https://doi.org/10.1145/1858996.1859010 (cit. on p. 39).

[16] Cor-Paul Bezemer et al., « An empirical study of unspecified dependencies in make-based build systems », *in*: *EMSE* 22.*6* (2017), pp. 3117–3148, DOI: 10.1007/s10664-017-9510-8 (cit. on p. 41).

[17] Tim Bird, *Advanced Size Optimization of the Linux Kernel*, 2013, URL: `https://elinux.org/images/9/9e/Bird-Kernel-Size-Optimization-LCJ-2013.pdf` (cit. on p. 40).

[18] Leo Breiman, « Random forests », *in*: *Machine learning* 45.*1* (2001), pp. 5–32 (cit. on pp. 49, 64, 76, 77).

[19] Kay Henning Brodersen et al., « The Balanced Accuracy and Its Posterior Distribution », *in*: *2010 20th International Conference on Pattern Recognition*, 2010, pp. 3121–3124, DOI: `10.1109/ICPR.2010.764` (cit. on p. 90).

[20] Lars Buitinck et al., « API design for machine learning software: experiences from the scikit-learn project », *in*: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122 (cit. on pp. 54, 109).

[21] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi, « A scalable and nearly uniform generator of SAT witnesses », *in*: *International Conference on Computer Aided Verification*, Springer, 2013, pp. 608–623 (cit. on pp. 57, 127).

[22] Supratik Chakraborty et al., « On Parallel Scalable Uniform SAT Witness Generation », *in*: *Tools and Algorithms for the Construction and Analysis of Systems TACAS'15 2015, London, UK, April 11-18, 2015. Proceedings*, 2015, pp. 304–319 (cit. on pp. 57, 127).

[23] Girish Chandrashekar and Ferat Sahin, « A survey on feature selection methods », *in*: *Computers & Electrical Engineering* 40.*1* (2014), pp. 16–28 (cit. on pp. 37, 59, 60, 63, 133).

[24] Haifeng Chen, Wenxuan Zhang, and Guofei Jiang, *Experience transfer for the configuration tuning of large scale computing systems*, US Patent 8,315,960, Nov. 2012 (cit. on pp. 36, 103).

[25] Haowen Chen et al., « An Empirical Study on Heterogeneous Defect Prediction Approaches », *in*: *IEEE Transactions on Software Engineering* (2020) (cit. on p. 36).

[26] Jinyin Chen et al., « Multiview transfer learning for software defect prediction », *in*: *IEEE Access* 7 (2019), pp. 8901–8916 (cit. on p. 36).

[27] *Clang*, URL: `https://clang.llvm.org/` (cit. on p. 126).

[28] Paul Clements and Linda Northrop, *Software product lines: practices and patterns*, The SEI series in software engineering, Boston: Addison-Wesley, 2002, 563 pp., ISBN: 9780201703320 (cit. on p. 26).

[29] C. Consel et al., « Tempo: specializing systems applications and beyond », *in*: *ACM Computing Surveys* 30.*3* (Sept. 1, 1998), 19–es, ISSN: 0360-0300, DOI: `10.1145/289121.289140`, URL: `https://doi.org/10.1145/289121.289140` (visited on 07/22/2021) (cit. on p. 28).

[30] *cURL*, URL: `https://curl.haxx.se/docs/releases.html` (cit. on p. 126).

[31] *Curl Evolution*, 2021, URL: `https://daniel.haxx.se/blog/2020/03/20/curl-22-years-in-22-pictures-and-2222-words/` (cit. on p. 26).

[32] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker, « Formalizing Cardinality-based Feature Models and their Specialization », *in*: *Software Process Improvement and Practice*, 2005, pp. 7–29 (cit. on pp. 26, 80, 82).

[33] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker, « Staged configuration through specialization and multilevel configuration of feature models », *in*: *Software Process: Improvement and Practice* 10.*2* (2005), pp. 143–169 (cit. on pp. 29, 80).

[34] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker, « Staged Configuration Using Feature Models », *in*: *Software Product Lines*, 2004, pp. 266–283, DOI: `10.1007/b100081` (cit. on p. 29).

[35] Wenyuan Dai et al., « Boosting for Transfer Learning », *in*: *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, Corvalis, Oregon, USA: Association for Computing Machinery, 2007, pp. 193–200, ISBN: 9781595937933, DOI: `10.1145/1273496.1273521`, URL: `https://doi.org/10.1145/1273496.1273521` (cit. on p. 36).

[36] Oscar Day and Taghi M Khoshgoftaar, « A survey on heterogeneous transfer learning », *in*: *Journal of Big Data* 4.*1* (2017), p. 29 (cit. on pp. 103, 113–116, 118).

[37] Christian Dietrich et al., « A robust approach for variability extraction from the Linux build system », *in*: *SPLC'12*, 2012, pp. 21–30 (cit. on p. 39).

[38] C.F. Dormann et al., « Collinearity: a review of methods to deal with it and a simulation study evaluating their performance », English, *in*: *Ecography* 36.*1* (2013), pp. 27–46, ISSN: 0906-7590, DOI: `10.1111/j.1600-0587.2012.07348.x` (cit. on p. 64).

[39] Johannes Dorn, Sven Apel, and Norbert Siegmund, « Generating Attributed Variability Models for Transfer Learning », *in*: *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*, VAMOS '20, Magdeburg, Germany: Association for Computing Machinery, 2020, ISBN: 9781450375016, DOI: `10.1145/3377024.3377040`, URL: `https://doi.org/10.1145/3377024.3377040` (cit. on p. 35).

[40] Johannes Dorn, Sven Apel, and Norbert Siegmund, « Mastering Uncertainty in Performance Estimations of Configurable Software Systems », *in*: *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 684–696 (cit. on p. 35).

[41] Norman R Draper and Harry Smith, *Applied regression analysis*, vol. 326, John Wiley & Sons, 1998 (cit. on p. 63).

[42] Rafael Dutra et al., « Efficient sampling of SAT solutions for testing », *in*: *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 549–559, DOI: `10.1145/3180155.3180248`, URL: `http://doi.acm.org/10.1145/3180155.3180248` (cit. on pp. 57, 127).

[43] Aaron Fisher, Cynthia Rudin, and Francesca Dominici, *All Models are Wrong, but Many are Useful: Learning a Variable's Importance by Studying an Entire Class of Prediction Models Simultaneously*, 2018, eprint: `arXiv:1801.01489` (cit. on pp. 64, 76, 77).

[44] Nick Galov, *111+ Linux Statistics and Facts*, 2021, URL: `https://hostingtribunal.com/blog/linux-statistics` (cit. on p. 38).

[45] Angelo Gargantini, Justyna Petke, and Marco Radavelli, « Combinatorial Interaction Testing for Automated Constraint Repair », *in*: *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2017, pp. 239–248, DOI: `10.1109/ICSTW.2017.44` (cit. on p. 35).

[46] Angelo Gargantini, Justyna Petke, and Marco Radavelli, « Combinatorial interaction testing for automated constraint repair », *in*: *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, IEEE, 2017, pp. 239–248 (cit. on p. 35).

[47] *GCC*, URL: `https://gcc.gnu.org/releases.html` (cit. on p. 126).

[48] Jianmei Guo et al., « Data-efficient performance learning for configurable systems », *in*: *EMSE* (2017), pp. 1–42 (cit. on p. 34).

[49] Jianmei Guo et al., « Data-efficient performance learning for configurable systems », *in*: *Empirical Software Engineering* 23.*3* (2018), pp. 1826–1867, DOI: `10.1007/s10664-017-9573-6`, URL: `https://doi.org/10.1007/s10664-017-9573-6` (cit. on pp. 49, 50).

[50] Jianmei Guo et al., « SMTIBEA: a hybrid multi-objective optimization algorithm for configuring large constrained software product lines », *in*: *Software & Systems Modeling* (July 2017), ISSN: 1619-1374, DOI: `10.1007/s10270-017-0610-0`, URL: `https://doi.org/10.1007/s10270-017-0610-0` (cit. on p. 33).

[51] Jianmei Guo et al., « Variability-aware performance prediction: A statistical learning approach », *in*: *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2013, pp. 301–311 (cit. on pp. 33, 34, 46).

[52] Huong Ha and Hongyu Zhang, « DeepPerf: performance prediction for configurable software with deep sparse neural network », *in*: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, 2019, pp. 1095–1106, URL: `https://dl.acm.org/citation.cfm?id=3339642` (cit. on pp. 84, 85, 88).

[53] Trevor Hastie, Robert Tibshirani, and Jerome Friedman, *The elements of statistical learning: data mining, inference, and prediction*, Springer Science & Business Media, 2009 (cit. on p. 59).

[54] https://github.com/ulfalizer/Kconfiglib, *A flexible Python 2/3 Kconfig implementation and library*, Accessed = 2019-05-08 (cit. on p. 61).

[55] `https://kernelnewbies.org/Linux_5.4`, *TKernelNewbies: Linux_5.4*, Nov. 2019 (cit. on p. 106).

[56] https://tiny.wiki.kernel.org/, *Linux Kernel Tinification*, last access: july 2019 (cit. on pp. 39, 45).

[57] `https://www.phoronix.com/scan.php?page=article&item=linux-416-54&num=1`, *The Disappointing Direction Of Linux Performance From 4.16 To 5.4 Kernels*, Nov. 2019 (cit. on p. 106).

[58] Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki, « A user survey of configuration challenges in Linux and eCos », *in*: *Sixth International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, January 25-27, 2012. Proceedings*, 2012, pp. 149–155, DOI: `10.1145/2110147.2110164`, URL: `https://doi.org/10.1145/2110147.2110164` (cit. on p. 45).

[59] Arnaud Hubaux et al., « Supporting Multiple Perspectives in Feature-based Configuration », *in*: *Software and Systems Modeling* (2011), pp. 1–23 (cit. on p. 80).

[60] Arnaud Hubaux et al., « Supporting multiple perspectives in feature-based configuration », *in*: *Software and System Modeling* 12.*3* (2013), pp. 641–663, DOI: `10.1007/s10270-011-0220-1`, URL: `https://doi.org/10.1007/s10270-011-0220-1` (cit. on p. 45).

[61] Md Shahriar Iqbal, Lars Kotthoff, and Pooyan Jamshidi, « Transfer Learning for Performance Modeling of Deep Neural Network Systems », *in*: *2019 USENIX OpML*, Santa Clara, CA: USENIX Association, May 2019, pp. 43–46, ISBN: 978-1-939133-00-7 (cit. on pp. 36, 103, 120).

[62] Gareth James et al., *An Introduction to Statistical Learning: with Applications in R*, Springer, 2013, URL: `https://faculty.marshall.usc.edu/gareth-james/ISL/` (cit. on p. 31).

[63] Pooyan Jamshidi et al., « Learning to sample: Exploiting similarities across environments to learn performance models for configurable systems », *in*: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 71–82 (cit. on pp. 35, 110).

[64] Pooyan Jamshidi et al., « Learning to sample: exploiting similarities across environments to learn performance models for configurable systems », *in*: *Proceedings of the 2018 ACM ESEC/SIGSOFT*, ACM, 2018, pp. 71–82 (cit. on pp. 36, 103, 110, 135).

[65] Pooyan Jamshidi et al., « Learning to sample: exploiting similarities across environments to learn performance models for configurable systems », *in*: *Proceedings of the 2018 ACM ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, 2018, pp. 71–82, DOI: `10.1145/3236024.3236074` (cit. on pp. 111, 115).

[66] Pooyan Jamshidi et al., « Machine Learning Meets Quantitative Planning: Enabling Self-Adaptation in Autonomous Robots », *in*: *arXiv preprint arXiv:1903.03920* (2019) (cit. on pp. 36, 50, 103).

[67] Pooyan Jamshidi et al., « Transfer Learning for Improving Model Predictions in Highly Configurable Software », *in*: *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Buenos Aires: IEEE Computer Society, May 2017, pp. 31–41, DOI: `http://dx.doi.org/10.1109/SEAMS.2017.11` (cit. on pp. 33, 34, 103, 127).

[68] Pooyan Jamshidi et al., « Transfer learning for performance modeling of configurable systems: An exploratory analysis », *in*: *2017 32nd ASE*, IEEE, 2017, pp. 497–508 (cit. on pp. 36, 103).

[69] Pooyan Jamshidi et al., « Transfer learning for performance modeling of configurable systems: an exploratory analysis », *in*: *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE Press, 2017, pp. 497–508 (cit. on pp. 33, 34, 50, 136).

[70] *Kafka*, URL: `https://cwiki.apache.org/confluence/display/KAFKA/Index` (cit. on p. 126).

[71] Christian Kaltenecker et al., « Distance-Based Sampling of Software Configuration Spaces », *in*: *Proceedings of the International Conference on Software Engineering (ICSE)*, 2019 (cit. on pp. 34, 46, 50).

[72] K. Kang et al., *Feature-Oriented Domain Analysis (FODA)*, tech. rep. CMU/SEI-90-TR-21, SEI, Nov. 1990 (cit. on p. 26).

[73] *KConfig Language*, 2021, URL: `https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html` (cit. on p. 39).

[74] *KConfig Language SAT integration*, 2021, URL: `https://kernelnewbies.org/KernelProjects/kconfig-sat` (cit. on p. 39).

[75] *KConfig Language Semantics*, 2021, URL: `https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html#semantics-of-kconfig` (cit. on p. 39).

[76] *KernelCI*, URL: `https://kernelci.org/` (cit. on p. 124).

[77] Diederik P. Kingma and Jimmy Ba, *Adam: A Method for Stochastic Optimization*, 2017, arXiv: `1412.6980 [cs.LG]` (cit. on p. 53).

[78] Ekrem Kocaguneli, Tim Menzies, and Emilia Mendes, « Transfer learning in effort estimation », *in*: *EMSE* 20.*3* (2015), pp. 813–843 (cit. on pp. 36, 103).

[79] Paul Kocher et al., « Spectre Attacks: Exploiting Speculative Execution », *in*: *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019 (cit. on p. 110).

[80] Sergiy Kolesnikov et al., « On the relation of external and internal feature interactions: A case study », *in*: *arXiv preprint arXiv:1712.07440* (2017) (cit. on p. 50).

[81] Sergiy Kolesnikov et al., « Tradeoffs in modeling performance of highly configurable software systems », *in*: *SoSyM* 18.*3* (June 2019), pp. 2265–2283, ISSN: 1619-1374 (cit. on pp. 34, 50, 110, 126).

[82] R. Krishna et al., « Whence to Learn? Transferring Knowledge in Configurable Systems using BEETLE », *in*: *IEEE Transactions on Software Engineering* (2020), pp. 1–1 (cit. on pp. 35, 111, 115).

[83] Rahul Krishna, Tim Menzies, and Wei Fu, « Too much automation? The bellwether effect and its implications for transfer learning », *in*: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 122–131 (cit. on p. 36).

[84] Thomas Krismayer, Rick Rabiser, and Paul Grünbacher, « Mining Constraints for Event-Based Monitoring in Systems of Systems », *in*: *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE Press, 2017, pp. 826–831 (cit. on p. 35).

[85] Charles W. Krueger, « Software reuse », *in*: *ACM Computing Surveys* 24.*2* (June 1, 1992), pp. 131–183, ISSN: 0360-0300, DOI: `10.1145/130844.130856`, URL: `https://doi.org/10.1145/130844.130856` (visited on 09/15/2021) (cit. on p. 26).

[86] Luc Lesoil et al., « Deep Software Variability: Towards Handling Cross-Layer Configuration », *in*: *VaMoS 2021 - 15th International Working Conference on Variability Modelling of Software-Intensive Systems*, Krems / Virtual, Austria, Feb. 2021, URL: `https://hal.inria.fr/hal-03084276` (cit. on pp. 101, 136).

[87] Luc Lesoil et al., « The Interplay of Compile-Time and Run-Time Options for Performance Prediction », *in*: *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A*, SPLC '21, Leicester, United Kingdom: Association for Computing Machinery, 2021, pp. 100–111, ISBN:

9781450384698, DOI: 10.1145/3461001.3471149, URL: https://doi.org/10.1145/3461001.3471149 (cit. on p. 136).

[88] Zhiqiang Li et al., « Cost-sensitive transfer kernel canonical correlation analysis for heterogeneous defect prediction », *in*: *ASE* 25.*2* (2018), pp. 201–245 (cit. on p. 36).

[89] Max Lillack, Johannes Müller, and Ulrich W Eisenecker, « Improved prediction of non-functional properties in software product lines with domain context », *in*: *Software Engineering 2013* (2013) (cit. on p. 34).

[90] Moritz Lipp et al., « Meltdown: Reading Kernel Memory from User Space », *in*: *27th USENIX Security Symposium (USENIX Security 18)*, 2018 (cit. on p. 110).

[91] H. Martin et al., « Transfer Learning Across Variants and Versions : The Case of Linux Kernel Size », *in*: *IEEE Transactions on Software Engineering 01* (Sept. 5555), pp. 1–1, ISSN: 1939-3520, DOI: 10.1109/TSE.2021.3116768 (cit. on p. 22).

[92] Hugo Martin et al., « A Comparison of Performance Specialization Learning for Configurable Systems », *in*: *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A*, SPLC '21, Leicester, United Kingdom: Association for Computing Machinery, 2021, pp. 46–57, ISBN: 9781450384698, DOI: 10.1145/3461001.3471155, URL: https://doi.org/10.1145/3461001.3471155 (cit. on p. 22).

[93] Martin Abadi et al., *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, Software available from tensorflow.org, 2015, URL: https://www.tensorflow.org/ (cit. on p. 54).

[94] Dylan McNamee et al., « Specialization tools and techniques for systematic optimization of system software », *in*: *ACM Trans. Comput. Syst.* 19.*2* (2001), pp. 217–251, DOI: 10.1145/377769.377778, URL: http://doi.acm.org/10.1145/377769.377778 (cit. on p. 28).

[95] Jean Melo et al., « A quantitative analysis of variability warnings in linux », *in*: *Proceedings of the Tenth VaMoS*, ACM, 2016, pp. 3–8 (cit. on pp. 41, 47, 57, 107, 127).

[96] Tom M. Mitchell, *Machine Learning*, New York: McGraw-Hill, 1997, ISBN: 978-0-07-042807-2 (cit. on p. 31).

[97] Christoph Molnar, *Interpretable Machine Learning*, Lulu. com, 2020 (cit. on pp. 64, 76, 77, 84).

[98] S. Mühlbauer, S. Apel, and N. Siegmund, « Identifying Software Performance Changes Across Variants and Versions », *in*: *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 611–622 (cit. on pp. 103, 105).

[99] Sarah Nadi et al., « Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study », *in*: *IEEE Trans. Software Eng.* () (cit. on p. 41).

[100] Vivek Nair et al., « Finding Faster Configurations Using FLASH », *in*: *IEEE Transactions on Software Engineering* 46.*7* (2020), pp. 794–811, DOI: `10.1109/TSE.2018.2870895` (cit. on p. 35).

[101] Vivek Nair et al., « Finding Faster Configurations Using Flash », *in*: *IEEE Transact. on Software Engineering* (2018) (cit. on pp. 33–35).

[102] Vivek Nair et al., « Using bad learners to find good configurations », *in*: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017, pp. 257–267, DOI: `10.1145/3106237.3106238` (cit. on pp. 34, 35).

[103] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim, « Transfer defect learning », *in*: *2013 35th ICSE*, IEEE, 2013, pp. 382–391 (cit. on pp. 36, 103).

[104] Jaechang Nam et al., « Heterogeneous defect prediction », *in*: *IEEE Transactions on Software Engineering* 44.*9* (2017), pp. 874–896 (cit. on p. 36).

[105] Lina Ochoa et al., « A systematic literature review on the semi-automatic configuration of extended product lines », *in*: *Journal of Systems and Software* 144 (2018), pp. 511–532 (cit. on p. 81).

[106] Jeho Oh et al., « Finding near-optimal configurations in product lines by random sampling », *in*: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017, pp. 61–71, DOI: `10.1145/3106237.3106273`, URL: `http://doi.acm.org/10.1145/3106237.3106273` (cit. on p. 33).

[107] Michael Opdenacker, *BoF: Embedded Linux Size*, Embedded Linux Conference North-America, 2018 (cit. on pp. 39, 45).

[108] *OpenCV*, URL: https://github.com/opencv/opencv_contrib (cit. on p. 126).

[109] Yoann Padioleau et al., « Documenting and Automating Collateral Evolutions in Linux Device Drivers », *in*: *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, Glasgow, Scotland UK: Association for Computing Machinery, 2008, pp. 247–260, ISBN: 9781605580135, DOI: 10.1145/1352592.1352618, URL: https://doi.org/10.1145/1352592.1352618 (cit. on p. 42).

[110] Sinno Jialin Pan and Qiang Yang, « A survey on transfer learning », *in*: *TKDE* 22.*10* (2009), pp. 1345–1359 (cit. on pp. 36, 103).

[111] Terence Parr et al., *Beware Default Random Forest Importances*, last access: july 2019, 2018 (cit. on pp. 64, 76, 77).

[112] L. Passos et al., « A Study of Feature Scattering in the Linux Kernel », *in*: *IEEE Transactions on Software Engineering* (2018), pp. 1–1, ISSN: 0098-5589, DOI: 10.1109/TSE.2018.2884911 (cit. on p. 41).

[113] F. Pedregosa et al., « Scikit-learn: Machine Learning in Python », *in*: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on p. 89).

[114] Juliana Alves Pereira et al., « Learning software configuration spaces: A systematic literature review », *in*: *Journal of Systems and Software* 182 (2021), p. 111044, ISSN: 0164-1212, DOI: https://doi.org/10.1016/j.jss.2021.111044, URL: https://www.sciencedirect.com/science/article/pii/S0164121221001412 (cit. on pp. 22, 33, 35, 46, 49, 52, 55–57, 60, 79, 84, 85, 110, 127).

[115] Juliana Alves Pereira et al., « Sampling Effect on Performance Prediction of Configurable Systems: A Case Study », *in*: *ICPE '20: ACM/SPEC International Conference on Performance Engineering, Edmonton, AB, Canada, April 20-24, 2020*, ed. by José Nelson Amaral et al., ACM, 2020, pp. 277–288, DOI: 10.1145/3358960.3379137, URL: https://doi.org/10.1145/3358960.3379137 (cit. on pp. 89, 100).

[116] Justyna Petke et al., « Efficiency and Early Fault Detection with Lower and Higher Strength Combinatorial Interaction Testing », *in*: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, Saint Petersburg, Russia: ACM, 2013, pp. 26–36, ISBN: 978-1-4503-2237-9 (cit. on p. 35).

[117] Quentin Plazar et al., « Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet? », *in*: *ICST 2019 - 12th International Conference on Software Testing, Verification, and Validation*, Xian, China, Apr. 2019, pp. 1–12 (cit. on pp. 34, 57, 127).

[118] Andrew Prout et al., « Measuring the impact of Spectre and Meltdown », *in*: *2018 IEEE High Performance extreme Computing Conference (HPEC)*, IEEE, 2018, pp. 1–5 (cit. on p. 106).

[119] Clément Quinton et al., « Evolution in dynamic software product lines », *in*: *Journal of Software: Evolution and Process* (2020), e2293 (cit. on p. 34).

[120] Xiang Ren et al., « An analysis of performance evolution of Linux's core operations », *in*: *Proceedings of the 27th ACM SOSP*, 2019, pp. 554–569 (cit. on p. 106).

[121] Valentin Rothberg et al., « Towards scalable configuration testing in variable software », *in*: *ACM SIGPLAN Notices*, vol. 52, 3, ACM, 2016, pp. 156–167 (cit. on p. 57).

[122] Safdar Aqeel Safdar et al., « Mining cross product line rules with multi-objective search and machine learning », *in*: *Proceedings of the Genetic and Evolutionary Computation Conference*, ACM, 2017, pp. 1319–1326 (cit. on pp. 35, 81).

[123] A. Sarkar et al., « Cost-Efficient Sampling for Performance Prediction of Configurable Systems (T) », *in*: *ASE'15*, 2015 (cit. on p. 46).

[124] Atri Sarkar et al., « Cost-efficient sampling for performance prediction of configurable systems (t) », *in*: *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2015, pp. 342–352 (cit. on p. 34).

[125] Mohammed Sayagh et al., « Software Configuration Engineering in Practice: Interviews, Survey, and Systematic Literature Review », *in*: *IEEE Transactions on Software Engineering* (2018) (cit. on p. 45).

[126] Ulrik P. Schultz, Julia L. Lawall, and Charles Consel, « Automatic Program Specialization for Java », *in*: *ACM Trans. Program. Lang. Syst.* 25.*4* (July 2003), pp. 452–499, ISSN: 0164-0925, DOI: `10.1145/778559.778561`, URL: `http://doi.acm.org/10.1145/778559.778561` (cit. on p. 28).

[127] Amir Molzam Sharifloo et al., « Learning and evolution in dynamic software product lines », *in*: *2016 IEEE/ACM 11th Int. Symposium on Software Engineering for Adaptive and Self-Managing Systems*, IEEE, 2016, pp. 158–164 (cit. on p. 34).

[128] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid, « Analysing the Kconfig Semantics and Its Analysis Tools », *in*: *SIGPLAN Not.* 51.*3* (Oct. 2015), pp. 45–54, ISSN: 0362-1340, DOI: `10.1145/2936314.2814222`, URL: `https://doi.org/10.1145/2936314.2814222` (cit. on p. 39).

[129] S. She et al., « Reverse Engineering Feature Models », *in*: *ICSE'11*, 2011 (cit. on p. 39).

[130] Norbert Siegmund, Stefan Sobernig, and Sven Apel, « Attributed Variability Models: Outside the Comfort Zone », *in*: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, Paderborn, Germany: ACM, 2017, pp. 268–278, ISBN: 978-1-4503-5105-8 (cit. on p. 127).

[131] Norbert Siegmund et al., « Performance-Influence Models for Highly Configurable Systems », *in*: *ESEC/FSE'15*, 2015 (cit. on pp. 33, 34, 37, 46, 49, 50, 60, 85, 88, 133).

[132] Norbert Siegmund et al., « Predicting performance via automated feature-interaction detection », *in*: *ICSE*, 2012, pp. 167–177 (cit. on p. 34).

[133] Norbert Siegmund et al., « Scalable prediction of non-functional properties in software product lines », *in*: *Software Product Line Conference (SPLC), 2011 15th International*, 2011, pp. 160–169 (cit. on pp. 34, 42, 46, 59).

[134] Norbert Siegmund et al., « Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption », *in*: *Information and Software Technology* 55.*3* (2013), pp. 491–507 (cit. on p. 34).

[135] Norbert Siegmund et al., « Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption », *in*: *Inf. Softw. Technol.* 55.*3* (2013), pp. 491–507, DOI: `10.1016/j.infsof.2012.07.020`, URL: `https://doi.org/10.1016/j.infsof.2012.07.020` (cit. on pp. 46, 59).

[136] Julio Sincero, Wolfgang Schroder-Preikschat, and Olaf Spinczyk, « Approaching non-functional properties of software product lines: Learning from products », *in*: *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, 2010, pp. 147–155 (cit. on pp. 42, 59).

[137] Gustavo Sousa, Walter Rudametkin, and Laurence Duchien, « Extending Feature Models with Relative Cardinalities », *in*: *Proceedings of the 20th International Systems and Software Product Line Conference*, SPLC '16, Beijing, China: Association for Computing Machinery, 2016, pp. 79–88, ISBN: 9781450340502, DOI: `10.1145/2934466.2934475`, URL: `https://doi.org/10.1145/2934466.2934475` (cit. on p. 26).

[138] Terry Speed, *Statistical analysis of gene expression microarray data*, Chapman and Hall/CRC, 2003 (cit. on pp. 59, 63).

[139] Supplementary material (Web page), *https://github.com/TuxML/size-analysis/* (cit. on pp. 47, 53, 54, 69, 74, 75, 77).

[140] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch, « A taxonomy of variability realization techniques: Research Articles », *in*: *Softw. Pract. Exper.* 35.*8* (2005), pp. 705–754, ISSN: 0038-0644, DOI: `http://dx.doi.org/10.1002/spe.v35:8` (cit. on p. 26).

[141] Paul Temple et al., « Empirical assessment of generating adversarial configurations for software product lines », *in*: *Empir. Softw. Eng.* 26.*1* (2021), p. 6, DOI: `10.1007/s10664-020-09915-7`, URL: `https://doi.org/10.1007/s10664-020-09915-7` (cit. on pp. 35, 81).

[142] Paul Temple et al., « Learning Contextual-Variability Models », *in*: *IEEE Software* 34.*6* (2017), pp. 64–70 (cit. on pp. 34, 35, 85, 86, 127).

[143] Paul Temple et al., « Using Machine Learning to Infer Constraints for Product Lines », *in*: *Software Product Line Conference (SPLC)*, Beijing, China, Sept. 2016 (cit. on pp. 33–35, 80).

[144] Robert Tibshirani, « Regression shrinkage and selection via the lasso », *in*: *Journal of the Royal Statistical Society: Series B (Methodological)* 58.*1* (1996), pp. 267–288 (cit. on pp. 49, 126).

[145] Linus Torvalds, « The linux edge », *in*: *Communications of the ACM* 42.*4* (1999), pp. 38–38 (cit. on p. 45).

[146] Josh Triplett, *Linux Kernel Tinification*, 2014, URL: `https://events.static.linuxfound.org/sites/events/files/slides/tiny.pdf` (cit. on p. 40).

[147] *Undertaker*, 2021, URL: `http://vamos.informatik.uni-erlangen.de/trac/undertaker` (cit. on p. 39).

[148] Pavel Valov, Jianmei Guo, and Krzysztof Czarnecki, « Empirical comparison of regression methods for variability-aware performance prediction », *in*: *SPLC'15*, 2015 (cit. on pp. 49, 50, 85).

[149] Pavel Valov, Jianmei Guo, and Krzysztof Czarnecki, « Transferring Pareto Frontiers across Heterogeneous Hardware Environments », *in*: *Proceedings of the ACM/SPEC ICPE*, ICPE '20, Edmonton AB, Canada: Association for Computing Machinery, 2020, pp. 12–23, ISBN: 9781450369916 (cit. on pp. 36, 103).

[150] Pavel Valov et al., « Transferring Performance Prediction Models Across Different Hardware Platforms », *in*: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ACM, 2017, pp. 39–50 (cit. on pp. 34, 36, 85, 103, 110, 111, 115, 120).

[151] Dana Van Aken et al., « Automatic database management system tuning through large-scale machine learning », *in*: *Proceedings of the 2017 ACM International Conference on Management of Data*, ACM, 2017, pp. 1009–1024 (cit. on p. 34).

[152] Miguel Velez et al., « Configcrusher: Towards white-box performance analysis for configurable systems », *in*: *Automated Software Engineering* 27.*3* (2020), pp. 265–300 (cit. on p. 35).

[153] Miguel Velez et al., « White-box analysis over machine learning: Modeling performance of configurable systems », *in*: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, IEEE, 2021, pp. 1072–1084 (cit. on p. 35).

[154] Aili Wang et al., « Few-shot learning based balanced distribution adaptation for heterogeneous defect prediction », *in*: *IEEE Access* 8 (2020), pp. 32989–33001 (cit. on p. 36).

[155] Xuezhi Wang, Tzu-Kuo Huang, and Jeff Schneider, « Active transfer learning under model shift », *in*: *ICML*, 2014, pp. 1305–1313 (cit. on pp. 36, 103).

[156] Max Weber, Sven Apel, and Norbert Siegmund, « White-Box Performance-Influence models: A profiling and learning approach », *in*: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, IEEE, 2021, pp. 1059–1071 (cit. on p. 35).

[157] Markus Weckesser et al., « Optimal reconfiguration of dynamic software product lines based on performance-influence models », *in*: *Proceeedings of the 22nd International Conference on Software Product Line*, ACM, 2018, pp. 98–109 (cit. on p. 50).

[158] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang, « A survey of transfer learning », *in*: *Journal of Big data* 3.*1* (2016), p. 9 (cit. on pp. 36, 103).

[159] Jeremy West, *A Theoretical Foundation for Inductive Transfer*, 2007, URL: `https://web.archive.org/web/20070801120743/http://cpms.byu.edu/springresearch/abstract-entry?id=861` (visited on 08/05/2007) (cit. on p. 36).

[160] Yingfei Xiong et al., « Generating Range Fixes for Software Configuration », *in*: *34th International Conference on Software Engineering*, June 2012 (cit. on p. 39).

[161] Yingfei Xiong et al., « Generating range fixes for software configuration », *in*: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, 2012, pp. 58–68, DOI: `10.1109/ICSE.2012.6227206`, URL: `https://doi.org/10.1109/ICSE.2012.6227206` (cit. on p. 45).

[162] Yingfei Xiong et al., « Range Fixes: Interactive Error Resolution for Software Configuration », *in*: *IEEE Trans. Software Eng.* 41.*6* (2015), pp. 603–619, DOI: `10.1109/TSE.2014.2383381`, URL: `https://doi.org/10.1109/TSE.2014.2383381` (cit. on p. 45).

[163] Tianyin Xu et al., « Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software », *in*: *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, 2015, pp. 307–319 (cit. on pp. 27, 28, 45).

[164] Yi Yao and Gianfranco Doretto, « Boosting for transfer learning with multiple sources », *in*: *2010 IEEE computer society conference on computer vision and pattern recognition*, IEEE, 2010, pp. 1855–1862 (cit. on p. 36).

[165] Cemal Yilmaz, Myra B Cohen, and Adam A Porter, « Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces », *in*: *IEEE Transactions on Software Engineering* 32.*1* (2006), pp. 20–34 (cit. on pp. 34, 35).

[166] Cemal Yilmaz et al., « Reducing masking effects in combinatorialinteraction testing: A feedback drivenadaptive approach », *in*: *IEEE Transactions on Software Engineering* 40.*1* (2014), pp. 43–66 (cit. on p. 34).

[167] Wen Zhang et al., « Overcoming negative transfer: A survey », *in*: *arXiv preprint arXiv:2009.00909* (2020) (cit. on p. 36).

[168] Yi Zhang et al., « A mathematical model of performance-relevant feature interactions », *in*: *Proceedings of the 20th International SPLC*, ACM, 2016, pp. 25–34 (cit. on p. 34).

[169] Wei Zheng, Ricardo Bianchini, and Thu D Nguyen, « Automatic configuration of internet services », *in*: *ACM SIGOPS Operating Systems Review* 41.*3* (2007), pp. 219–229 (cit. on p. 45).