Universität Leipzig
Faculty of Mathematics and Computer Science

Group Sampling zum Lernen konfigurationsabhängiger Softwareperformance

---

Group Sampling for learning configuration-specific software performance

Bachelor's Thesis

Alexander Zwisler
Born Jan. 1, 1 in XXX

Matriculation Number XXX

1. Referee: XXX
2. Referee: XXX

Submission date: February 11, 2022

Declaration

Unless otherwise indicated in the text or references, this thesis is entirely the product of my own scholarly work.

Leipzig, February 11, 2022

. . . .
Alexander Zwisler

Abstract

Modern software systems can often offer many configuration options to adapt the system to the needs of the end-user. The influence of configuration options on non-functional properties such as performance or energy consumption are often poorly understood. Performance influence models help in understanding the influences of configuration options on non-functional properties and make it possible to pick optimal configurations or find performance hot spots.

This work introduces an approach to create a performance influence model using an experimental design called "group sampling". We apply the group sampling technique to an input space with constraints, which poses a challenge we tackle in this thesis. With repeated measurements of the effect different configurations have, we determine the influence single configuration options and interactions between them have on a non-functional property. We evaluate our performance influence model on real-world datasets and synthetically generated data. While the model created proved to be unreliable, the group sampling approach was able to identify influential features with a minimal amount of samples.

Data and code is available at:

- `https://git.informatik.uni-leipzig.de/az11kyho/ba-group-sampling`
- `https://github.com/zwisler-a/bachelor-thesis-group-sampling`

Contents

List of Figures

Chapter 1

Introduction

Nowadays, most software systems provide configuration options that allow end-users, developers, and administrators to adapt them to their specific needs. But this adaptability comes at a high cost.
The more options there are to configure a system, the harder it gets to find configurations optimal for the specific needs of the user. While the functional aspects of a configuration option are usually well known and documented, the non-functional properties are mostly unknown. This makes it difficult to configure a software system optimal for a non-functional property such as performance, power consumption or memory footprint.
The naive way to test all possible configurations in order to determine the optimal configuration for a given non-functional property is unfeasible. This is due to the combinatorial complexity of the configuration space and the constraints on the configuration options. Since there is no straightforward way to determine the influence of configuration options on non-functional properties, these properties are mostly ignored in the choice of a fitting configuration [1] or are only chosen by previous "tribal knowledge", possibly leaving already built-in optimizations in software systems untapped.

In literature, a commonly used method to help choosing the optimal configuration for a non-functional property is to create a model describing the influence of configuration options on the non-functional property [1]. This model can then be used to find optimal configurations and predict the non-functional property of previously unseen configurations without any further testing of the system. A developer, for example, might use this model to find a configuration that minimizes the power consumption of the software to reduce operational costs. To create this kind of model, machine learning techniques are a simple go-to. The creation of such models is a research field of its own and there are many well-understood machine-learning techniques, which can be used to create a model for the influence of software configurations on non-functional properties. Since it is infeasible to test all possible configurations, a subset of configurations must be selected. This process is called sampling. While sampling in traditional machine learning applications might be as easy as just randomly picking data points to train the model, it is not as straightforward for configurations options in the presence of constraints. This is mainly due to the constraints the configuration options have, which make the sampling of configurations into a task of finding a valid assignment of configuration options and thus NP-hard.

There are several techniques already tackling the problem of sampling on software configurations. These techniques allow a user to sample software configurations and select random or near-random configurations under specific criteria. With these sampling techniques, the creation of performance influence models with prediction errors of 1% are possible [1]. However, there is still motivation to further reduce the needed amount of samples to make predictions more performant and easier to use. This reduction of required data to model the influence of multiple parameters on a specific variable is not a problem specific to software configurations. Work by Andres [2] and Saltelli et al. [3] presents an experimental design for sensitivity analysis which reduces the number of needed simulations to less than the number of parameters and which is still able to give meaningful information of the influence of those parameters. This experimental design is called group sampling. While the core concept of analyzing the influence of multiple parameters on a single variable is quite similar to the problem of analyzing the influence of configuration options on non-functional properties, there are a few problems in adapting these techniques to software configurations. As already mentioned before, this is mainly due to the constraints and interactions of configuration options.

The constraints on configuration options make it hard to look at the influence of a single configuration option. This is because one option might not be able to be turned on independently of other configuration options. Therefore, measurements conducted on this pair of configuration options might not be able to tell the specific influence of the single configuration options in question. Further, certain configuration options might interact with each other, cancelling out or multiplying their influences. This is easily understood by looking at two common configuration options in compression programs. The encryption and the compression. If the compression is more aggressively compressing the data, the encryption part of the program has fewer data to encrypt, making the program more performant.

In this work, we analyze how the experimental design of group sampling can be adapted to software configurations, by implementing a group sampling algorithm for software configurations and answering the following questions:

RQ1 How can we mitigate the effect of constraints on the configuration options for a group sampling approach?

In literature, in group sampling, constraints on the parameters are non-existent, but with configuration options, these constraints make it impossible to create arbitrary groupings. We devise two approaches to create groups in the presence of constraints and test which results in a better performance influence model.

RQ2 How effective is a group sampling algorithm compared to random sampling in order to create performance influence models?

We answer this question with experiments on real-world and synthetic data sets. By comparing group sampling and random sampling with linear regression with varying sample and group sizes, we determine under which conditions, if any, group sampling is superior to random sampling.

RQ3 How scalable is group sampling?

Since group sampling is an experimental design for large amounts of parameters, the resulting model should be useful for large software systems where random sampling might not be of use because of the large sample sets required to create a performant model. This is done by creating synthetic data for a software system with a large configuration space.

RQ4 Can group sampling be used to effectively identify feature interactions and include them in the resulting model?

Since the grouping of configuration options already includes the influences of feature interactions, group sampling may be used to identify these interactions and effectively incorporate those in the resulting model. We answer this question with experiments on synthetic data, where the amount of feature interactions is at our discretion.

Further, we will have a look at optimization techniques, which are already described in literature to get better results when using a group sampling algorithm.

First, we give some background knowledge in chapter 2 and then explain our approach to group sampling in chapter 3. In chapter 4 and chapter 5 we evaluate and discuss our approach.

Chapter 2

Background

2.1 Configurable Software Systems

Almost all software systems in use today are configurable. This means, almost all software systems allow a user to customize the software system to their specific requirements. To allow for such customizability, software systems provide a number of configuration options, also called features, to specify the desired behaviour of the software system. A feature is an attribute of a system that directly affects end-users [4]. Each combination of configuration options describes a variant of the software system, making configurable software systems similar to Software Product Lines (SPL) [5]. For example, a web server like the Apache HTTP Server, allows the user to select features like compression, encryption or the use of specific transport protocols like HTTP/2.

Features usually satisfy functional requirements [5] of users. Functional requirement as described by Malan et al. "[...] capture the intended behaviour of the system - or what the system will do". Meaning, for the example of the webserver, a functional requirement would be the use of a specific compression algorithm like Brotli[1]. On the other hand, software systems also have non-functional requirements. Mylopoulos et al. describes non-functional requirements as "[...] requirements on its development or operational cost, performance, reliability, maintainability, portability, robustness, and the like."

In this work, non-functional requirements, like performance or memory usage, of a software system are analyzed in regard to the configurations used. Usually, the effect of configuration options on functional properties of the software are well documented and understood, but the effect on non-functional properties are unknown to the user and more often than not, to the developer.

[1]Brotli is a compression algorithm developed by Google primarily used by web servers to compress HTTP content.

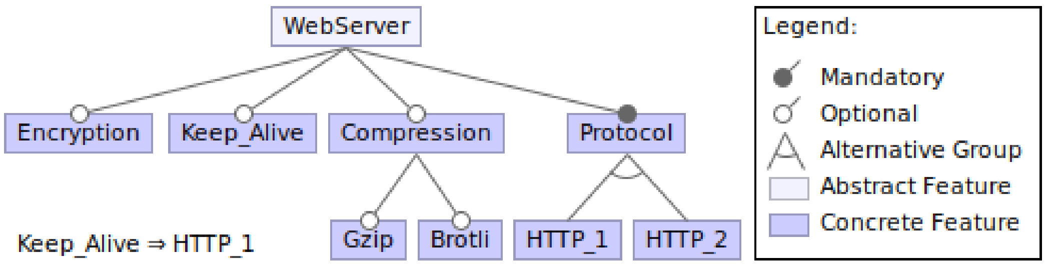

Figure 2.1: Feature diagram of an exemplary web server created with FeatureIDE

Feature models

Feature models, introduced by Kang et al. [4], serve as a mean to convey information about the features of a system. Which features can be chosen and how those features can be combined. For this, the structural relationship between features is described. Each feature can either be optional or mandatory and each feature can have child features that are either in an or group or an alternative group. In an alternative group, only one of the child features can be selected, in an or group, at least one of the child features must be selected. There are also composition rules, also known as cross-tree constraints, which define constraints not expressible in the hierarchical structure of the features.

Feature models can be represented visually by feature diagrams or textually by formats like SXFM [8]. Feature diagrams are an intuitive way for a user to work with feature models, they visualize the feature model in a tree diagram. An example of a feature diagram can be seen in Figure 2.1. There, the features of an exemplary web server are described. The web server has 4 features, of which 3 are optional and one is mandatory. It also includes two cross-tree constraints. The feature *Encryption* is an optional feature, without any constraints on it. This feature can be enabled or disabled however the user likes. In this work, this kind of feature without any constraints on it is also referenced as an independent feature. Even though the *Keep_alive* feature has no differences in the hierarchical structure to the *Encryption* feature, it has a cross-tree constrain. This constraint mandates, that if *Keep_alive* is selected, the *HTTP/1* feature is selected too, since the *Keep_alive* functionality is only available in the HTTP/1 protocol. *Compression* and *Protocol* are both features with child features. Since *HTTP/1* and *HTTP/2* are in an alternate group, only one of these two protocols can be chosen for a valid configuration. With *Compression*, one or both of the child features can be chosen, since they are in an or group.

Interactions

Features in a software system directly influence the way a system behaves with respect to its functional and non-functional properties. If we turned on the feature *Encryption* in our exemplary web server, the webserver would need to encrypt the sent data. This not only has impacts on the functional properties of the web server, but it also impacts the non-functional property. The web server now has to further process the data before sending it, which results in more computational power needed. The same goes for the feature *Compression*, or more specifically for the features *Brotli* and *Gzip*. If the compression is enabled in the webserver, the webserver needs to compress the data before sending it, which requires more computational power.

One could now assume, that if we enable both features at the same time, the computational power needed to process a request is the sum of the power needed if the encryption is enabled and if the compression is enabled. A more likely scenario would be, that the computational power needed would be less than that. If we first compress the data, the encryption part of the webserver has to encrypt fewer data. The two features *Encryption* and *Compression* interact.

2.2 Performance-Influence Models

The number of possible configurations of modern software systems and the complex constraints between them can be overwhelming. Making it difficult to find an optimal configuration, that performs as desired. Performance influence models are meant to ease understanding, debugging and optimization of configurable software systems [1].

A performance influence model consists of several terms that describe the performance of a configuration based on the values of configuration options [1]. In this context, performance can be measurable quality attributes such as execution time, memory size, or energy consumption. The model describes the influence of several independent variables X, our configuration, on a dependent variable y, our measurable quality attribute. While there are several approaches to predict performance, in general, they all work similarly. They sample a subset of configurations - this is done because it is infeasible to measure the performance of all configurations if the configuration space is too big - and learn a model with the sampled configurations.

Guo et al. [9] introduce a variability-aware approach to predict a configuration's performance based on random sampling. They use a Classification-And-Regression-Tree [10] to recursively partition the configuration space into smaller segments until they can fit a simple local prediction model into each segment.

Siegmund et al. [1] describe how to create human understandable models based on previous work [11]. They combined binary sampling strategies such as option-wise, negative option-wise, and pair-wise, with numerical sampling strategies, such as the Placket-Burman-Design. They then used stepwise linear regression to learn the influence model.

2.2.1 Sampling

The selection of a subset of configurations plays an integral role in almost all methods to predict the performance of a software system. If a configuration option is not present in the sampled subset of configuration a model can not learn the influence of this option. The random selection, random sampling, as used by most machine learning applications proves to be difficult with configurations. Mainly due to the constraints on the configuration options. Near uniform random sampling in the presence of constraints, although possible, is infeasible [12]. This resulted in developing dedicated sampling strategies for configuration spaces.

Distance based sampling

Kaltenecker et al. [13] describes a way to randomly sample a configuration space based on a distance metric and a probability distribution, called distance-based sampling. For this, they rely on a distance metric, like the Manhattan-Distance, to assign each configuration a distance value. By selecting a distance value through a discrete probability distribution and then picking a configuration with the corresponding distance value, they achieve a spread over the configurations resembling the given probability distribution. This allows for uniform random like sampling if the chosen probability distribution is the uniform distribution.

Binary decision diagram-sampling

Although Oh et al. [14] do not create a model to predict the performance of a system, they implement a way of random sampling configuration spaces through binary decision diagrams (BDD)[15]. They transform a given feature model into BDD, this makes it easy to count the number of valid configurations and thus easy to randomly sample from them. While a BDD allows for random sampling, a major drawback is the creation of it, which may exceed time or memory constraints for some use cases.

2.3 SAT-Solver

The satisfiability problem in propositional logic (SAT) is the problem of determining the existence of any solution, that satisfies a given boolean formula. A boolean formula is called satisfiable, if we can assign the variables in the formula true or false values in such a way, that the formula evaluates to true. As an example, the formula $A \lor B$ is satisfiable. We can prove this by assigning $A = true$ and $B = false$. The resulting formula would be $true \lor false$, which evaluates to true, meaning the formula is satisfiable. If a formula is not satisfiable, the formula is called unsatisfiable. An example of this would be the formula $A \land \neg A$. No possible assignment of A would result in the formula being evaluated to true. SAT-problems arise in many application domains and, most notably for this work, can be used in validating configurations for SPLs.

SAT-Solvers are programs, which aim to solve the SAT-problem. Even though the SAT-problem is NP-complete [16], modern SAT-Solvers can often handle problems with hundreds of thousands of variables and millions of constraints [17]. They often use algorithms such as DPLL [18] or variations of it at their core and their optimization is a large field of research on its own. Many free and open-source implementations of SAT-Solvers exist [19, 20, 21]. In this work, we use Z3 [22], which is an SMT-solver. Satisfiability modulo theories (SMT) generalize the SAT-problem to formulas involving real numbers and various data structures. The use of an SMT-solver, instead of an SAT-Solver, allows us to define cost functions to optimize the solution of the SMT-solver.

In this work, we make use of the SMT-solver Z3 to generate valid configurations for a configurable software system. Like Henard et al. [23] defined it, we define our feature model as a set of boolean features with a set of constraints over them. A configuration is a set of features, where all features in that set are selected and features that are not in the set are unselected. A configuration is valid if it satisfies all the constraints of the feature model, otherwise, it is called invalid. Batory [24] shows, that a feature model can be defined as a propositional formula. This not only allows the feature model to be stored in file formats like DIMACS, but it also allows the use of of-the-shelf SAT-Solvers for feature models.

2.4 Sensitivity analysis

Saltelli describes sensitivity analysis as "[...] the study of how the uncertainty in the output of a model (numerical or otherwise) can be apportioned to different sources of uncertainty in the model input" [25]. To understand this definition, we need to understand, what a model is. A model, in this context, is a mathematical description of a system and its rules. Such models could be weather models to predict future rainfall or financial models to predict the stock market. By defining the input of the model, we can get a prediction from the model. In this work, we view a configurable software system as a black-box model on which we can perform sensitivity analysis. Our input variables are the features the systems have, and our output variable is the measurable quality attribute. Such models, no matter what they predict or are, can be highly complex, and it can be hard to understand the relationship between the inputs and the outputs of the models. The degree to which an output variable is affected by the change in an input variable is called sensitivity. There are many approaches to performing sensitivity analysis. One common and simple approach to sensitivity analysis is the one-at-a-time (OAT) approach. In this experimental design, one input variable is changed, while all other values are kept at a baseline value. By observing the changes in the output variable, one can determine the influence an input variable has. Sensitivity can then be described, for example, by linear regression or partial derivatives.

Performing sensitivity analysis on a configurable software with the OAT-design would mean, enabling one feature at a time while all other features are disabled and observing the change in the quality attribute under analysis. We assume here, that the software system has no constraints among the configuration options. By fitting a regression model we can determine the sensitivity of the quality attribute to the features enabled.

Chapter 3

Group Sampling

3.1 Introduction

Group sampling is an experimental design introduced to handle large parameter sets in a sensitivity analysis [2]. This design allows an analyst to identify influential parameters and determine their influence. It even allows obtaining sensitivity analysis information from so-called supersaturated designs. These are designs, where the number of measurements is smaller than the number of parameters.

The assumption is made, that the influence of each parameter is negligible. This assumption is rejected if there is data providing strong evidence of this influence. With this approach, the problem can be viewed as one of statistical testing [3]. This makes it possible to test for influential parameters and later reuse this information to analyze the nature of these influences. At the core of group sampling is the idea, that information about parameters can be extracted if multiple parameters are put in a group and tested together.

How the amount of testing can be reduced by clever designs, can be explained by recent efforts to reduce the number of tests needed to test as many people as possible for the SARS-CoV-2-Virus. Mutesa et al. [26] describes several approaches for pooled testing.

Here, a slightly altered version will be explained to more easily show, how the number of tests can be reduced by pooled testing. We assume we want to test 9 individuals for SARS-CoV-2 and want to reduce the number of tests needed. We also assume that the chance of more than one person being infected is negligibly small. By grouping our 9 individuals into groups and only using one test for the whole group, we can reduce the needed tests without significantly impacting the chance of detection of an infected individual.

This can be done by grouping these 9 individuals in a specific way. First, we create a matrix with the same amount of elements as the individuals to test and the dimensions $m \times n$. In this case, a 3x3-matrix. Now we assign individuals to a corresponding place in the matrix. This is done by assigning the element x_{ij} to the individual $I_{(mi+j)}$. The groups for testing are then created by taking each column and each row of the matrix as a group, resulting in six groups total. For ease of reference, the groups created from the rows are g_i where i is the index of the row and the groups created from the columns are h_j where j is the index of the column. This form of grouping gives us a way to re-identify an infected person if there is one. Let's assume I_5 is infected with SARS-CoV-2 in this example. Subsequently, the groups containing this individual would test as positive. In our example, if I_5 is infected, the groupings g_2 and h_5 would test as positive. To re-identify the individual, the overlap between those two groupings needs to be identified. This can be done by looking at the indices of the groupings $f(g_i, h_i) = m * i + j$.

With this, the amount of needed tests to test 9 individuals is reduced from 9 to 6 tests. This design, in theory, can be scaled up but may suffer in reliability when increased in size. In the case of the SARS-CoV-2 example, the tests used on the group might not be able to identify if a group is infected if the amount of individuals in this group is higher than a certain threshold.

The same idea of extracting as much information as possible out of groupings is used in group sampling. Here, the parameters are grouped randomly in groups of the same size. For each group, a test is done, and the influence values are stored. By repeating this grouping multiple times with different groups, information about the influence of a single parameter can be extracted.

3.2 Group sampling for sensitivity analysis

Saltelli et al. [3] describes how group sampling can be applied to perform sensitivity analysis on a given model. Suppose we have a model with 1000 parameters X_i with $i \in [1 - 1000]$ where only a few parameters are influential. We group the parameters into groups of equal size. With 1000 parameters a sensible amount of groups could be M=10, giving us 10 groups containing 100 parameters. Group G_1 would be $G_1 = \{X_1, X_2, X_3, ..., X_{100}\}$, $G_2 = \{X_{101}, X_{102}, X_{103}, ..., X_{200}\}$ and Group $G_{10} = \{X_{901}, X_{902}, X_{903}, ..., X_{1000}\}$. This kind of grouping would be repeated N times, with randomly assigned parameters for each group, resulting in N * M groups $g_{n,m}$, where for each grouping $g_{n,M}$ all sets of parameters are distinct. Each group is now treated as a parameter of its own, assigning all parameters in a group the same value, giving us an abstraction of the model with only 10 parameters. Now, on these 10 groups, we can perform a set of simulations to determine the influence of the group as if it were a single parameter.

Table 3.1: Parameter groupings and their influences

M	G_1	G_2	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
1	**8.1**	1.8	**8.1**	1.8	1.8	**8.1**	1.8	**8.1**	1.8	1.8	**8.1**	**8.1**
2	**9.3**	5.1	**9.3**	5.1	5.1	**9.3**	5.1	**9.3**	**9.3**	5.1	**9.3**	5.1
3	**10**	6.5	6.5	6.5	6.5	**10**	6.5	**10**	**10**	6.5	**10**	**10**
4	**9.8**	5.4	5.4	**9.8**	5.4	**9.8**	**9.8**	**9.8**	**9.8**	5.4	5.4	5.4
5	**4.7**	0.9	**4.7**	0.9	**4.7**	**4.7**	**4.7**	0.9	0.9	**4.7**	0.9	0.9
			6.8	4.8	4.7	8.4	5.6	7.6	6.4	4.7	6.7	5.9

In Table 3.1 an example with 10 parameters, two groups M=2 and 5 rounds of random grouping can be found. All parameters with bold font are in the first group of the grouping M. For each group an influence value is determined and each parameter in the group is assigned this influence value for this grouping. If we look at the second grouping, the parameters X_1, X_4, X_6, X_7, X_9 are in this first group G_1 and have the influence value of 9.3. The parameters X_2, X_3, X_5, X_8, X_{10} are in the second group G_2 and have the influence value of 5.1. In this example, the parameter X_4 is an influential parameter. We can already see, with the second grouping alone, the influential parameter is most likely contained in group G_1 for the second grouping. If we take the average value of each parameter for each grouping, we can estimate the influence of each parameter. In this example, we can see the parameter X_4 has the highest influence and is most likely our influential parameter.

From this example, we can also see how non-influential parameters can look influential if they share the group with the influential parameter too often. If we look at X_6, the parameter looks influential, even though if the actual influential parameter is not in the same group, the group influence is small. This makes it harder to determine the actual influential parameter. Saltelli et al. [3] give us the probability of two parameters, X_i and X_j sharing a group t times with:

$$P_{ij}(t, N, S) = \binom{N}{t} \left(\frac{1}{S}\right)^t \left(\frac{S-1}{S}\right)^{N-t} \tag{3.1}$$

For our example with N = 5 groupings and a group size of $S = \frac{|X|}{M} = 5$, this gives us a probability of X_4 and X_6 sharing the same group t=3 times with $P_{4,6}(3, 5, 5) = 0.051$ and a total probability of any parameter sharing the group with the influential parameter X_4 t times with $1 - (1 - P_{ij}(3, 5, 5))^9 = 0.377$.

3.3 Group sampling with configuration options

In order to implement group sampling on configuration options, the method described by Saltelli et al. needs to be adapted. For simplicity, we only look at feature models with binary features. When grouping features, each group of features needs to be a valid configuration if all features in a group are enabled. Otherwise, measuring the influence of the group on its own is not possible. This forces us to adhere to the constraints on the feature model while creating groups. We can identify three different types of features, which affect the way we can assign features to groups.

Mutually exclusive features

In this work, we call a group of features, where we can not enable more than one at the same time, a group of mutually exclusive features and consequently a feature contained in one of those groups, a mutually exclusive feature. For example, an alternative group is always a group of mutually exclusive features, since only one feature can be enabled without violating the constraints. Mutually exclusive features are problematic when we group features. We can't have two features assigned to the same group if they are together in a mutually exclusive group. This would cause an invalid configuration once the features of the group are enabled.

Independent features

We call features, which do not have any constraints on them and are optional, independent features. If a set of features already are a valid configuration, these features can be added or removed completely independent of the already selected features. The resulting set of features would still be a valid configuration. These features allow for the easy creation of groups among them since they can be combined in any way possible without violating any constraints.

Implying feature

Implying features, or as Benavides et al. categorizes them, requires features [27], are features with a constraint, which forces us to select the implied feature if the implying feature is enabled. If an implying feature is assigned to a group, we need to assign the implied feature to the same group, otherwise, we would get an invalid configuration if the group with the implying feature is selected and the one with the implied feature is not.

In Saltelli et al. [3] group sampling, each group is a distinct set of parameters. This limits the number of groups possible on a set of features with constraints. While mandatory features would make it impossible to create any groups, we simply could ignore them, since they do not impact the performance of a given system. The most limiting factor would be a mandatory group of mutually exclusive features. The number of possible groups with a distinct set of features would equal the smallest mandatory group of mutually exclusive features. If a feature model contains multiple sets of mutually exclusive groups of different sizes, it is impossible to assign all mutually exclusive features to a group while still having a distinct set of features for each group.

To create a performance influence model we need to measure the influence of each group individually. By doing so, we can estimate the influence of each group of features. With repeated testing of different groupings, we can estimate the influence of each feature by averaging the influence of the feature across all groupings. While this does not give us a perfect estimate of the influence of each feature, it lets us test, which of the features is most likely influential and to what degree.

3.4 Creation of configuration groups

In this section, we tackle the problem of adapting the method described by Saltelli et al. [3] to configuration options. We especially try to handle the problem of constraints among the features during group creation. We devise two strategies to create groups on features and handle the complexity of constraints during group creation. In the previous section, we identified several problems the constraints cause when creating groups of features. Both methods described in the following sections aim to mitigate these obstacles in order to still create groups in the presence of constraints.

3.4.1 Maximizing Hamming distance

This method aims to generate groupings of features without any previous analysis of the feature model. The main idea is to create groups of features with minimal overlap between them. In the best-case scenario of a feature model with only independent features, this would create groups without any overlap between them and thus completely distinct sets of features. To measure the overlap between two groups of features we use the Hamming distance between the two groups.

$$D_H = \sum_{i=1}^{n} f(A_i, B_i)$$

$$f(A, B) = \begin{cases} 0 & \text{if } A = B \\ 1 & \text{else} \end{cases} \tag{3.2}$$

This means, the Hamming distance between two groups is equal to the amount of features which are not in both groups.

Since we do not have a hard constraint of no overlap between groups, mandatory features are allowed to be in all groups. This allows us to ignore them during group creation due to the fact, that their influence on performance does not contribute to performance variation. The complexity of mutually exclusive groups and features with implications is implicitly solved by the SAT-Solver, since it is responsible to create valid configurations.

We also need to define how many features should be contained in a group. Otherwise, we leave it open to the SAT-Solver to determine the number of features in a group, which would result in very uneven group sizes due to the nature of SAT-Solvers. Since we do not have any previous knowledge of the number of independent features, mutually exclusive groups or features with implications, the most straightforward answer to the number of features which should be contained in a group would be:

$$Features\ in\ group = \left\lfloor \frac{No.\ Features}{Group\ size} \right\rfloor \tag{3.3}$$

3.4.2 Grouping independent features

While grouping features using the Hamming distance between the groups simplifies the process of creating groups by pushing the complexity to the SAT-Solver. This method provides a more hands-on approach. We analyze the feature model and determine the independent features and the groups of mutually exclusive features. With the knowledge we have, we can more easily create valid groups of features.

Groups with mutually exclusive features

To create groups among mutually exclusive features, we want to pick one feature out of each group of mutually exclusive features, if possible. With this, we get the maximal amount of features we can group together across all mutually exclusive groups. If we assume, we have a feature model with two alternative groups, the maximal amount of features, we can group together is two, since only one feature out of each alternative group can be selected. To look for mutually exclusive features in a feature model, we can use the SAT-Solver. By enabling each combination of any two features and checking, if there is a valid configuration, we can find out which two features are mutually exclusive. The pairs of mutually exclusive features help us to determine the mutually exclusive groups. In a mutually exclusive group, only one feature of the group can be enabled. This means all features in the group are mutually exclusive with all other features in the group. If we translate the mutually exclusive relationship between features into an undirected graph, where the nodes represent the features and the edges the mutually exclusive relationship, we can reformulate the problem of finding the groups as the clique problem [28]. The cliques in the graph represent a mutually exclusive group because all nodes in the clique are fully connected. Figure 3.1 depicts such a graph.

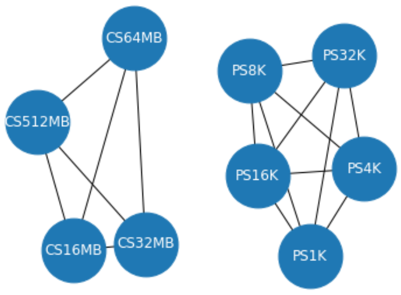

Figure 3.1: Graph of mutually exclusive features in the BerkeleyDB dataset.

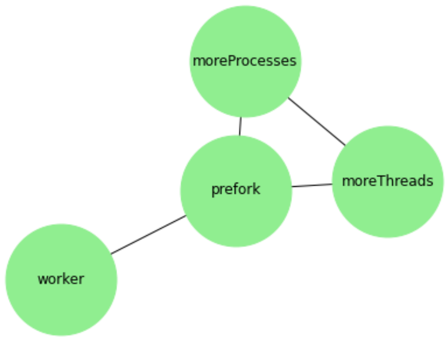

Figure 3.2: Partial graph of mutually exclusive features in the Apache dataset.

When we select features out of mutually exclusive groups we have to be aware, that mutually exclusive groups can overlap. This presents a problem when we select an overlapping feature. To illustrate the problem, a subgraph from the mutually exclusive relationships in the Apache dataset is presented in Figure 3.2. The mutually exclusive groups, the cliques, in this example would be *worker, prefork* and *prefork, moreProcesses, moreThreads*. Since *prefork* is contained in both groups, if we pick this feature, we can not pick any feature of the other mutually exclusive group. To circumvent this problem, instead of choosing one feature out of each mutually exclusive group, we choose one feature out of each component[1]. This leaves us with smaller groups since fewer features can be selected for each group, but a more equal distribution of selected features in connected mutually exclusive groups.

Groups with independent features

The independent features are all features, which can be disabled and are not in any of the mutually exclusive groups. Not all features in a feature model which are modelled as optional are really optional. Benavides et al. [27] describe these features, features which are included in all configurations, despite being modelled as optional, as false optional features. So to determine the independent features we need to determine features, which really can be enabled optionally. This can be done by iterating overall features and checking if a valid configuration exists where the feature is disabled [29]. The set of features which can be disabled without the features contained in a group of mutually exclusive features is the set of independent features. Since the independent features are not bound by any constraints, we can randomly assign a feature to a group without being concerned about violating any constraints.

We can now combine both approaches for grouping mutually exclusive features and independent features to create groups over the whole feature model. We are still limited by the fact, that we can not create more distinct groups than the amount of the smallest mutually exclusive component. This would mean, that we can't cover all features in one round of grouping. We can circumvent this problem by simply allowing for mutually exclusive features to be in multiple groups. This lets us create more groups, but makes it harder to determine the influence of a feature that is assigned to multiple groups.

To actually generate a group, we need to define how many features are in a group. The amount of features from the mutually exclusive features is simply

[1]A component of an undirected graph is connected subgraph has no edges to any other graph

the amount of components $|C|$ in the mutually exclusive graph. The amount of features in a group can be described with the following formula:

$$N = \left\lfloor \frac{No.\ independent\ features}{Group\ size} \right\rfloor \tag{3.4}$$

$$Features\ in\ group = N + |C| + mandatory\ features \tag{3.5}$$

We can see samples created from both variations in Figure 3.3. Each row represents a configuration and each column represents a feature. A feature is selected, if the block is black, otherwise it is not selected. A round of groupings is separated by a horizontal line. Looking at the figure, we can already identify a problem both varients have. The differences between the rounds of groupings are minimal, caused by the use of an SAT-Solver. We tackle this problem in subsection 3.6.2

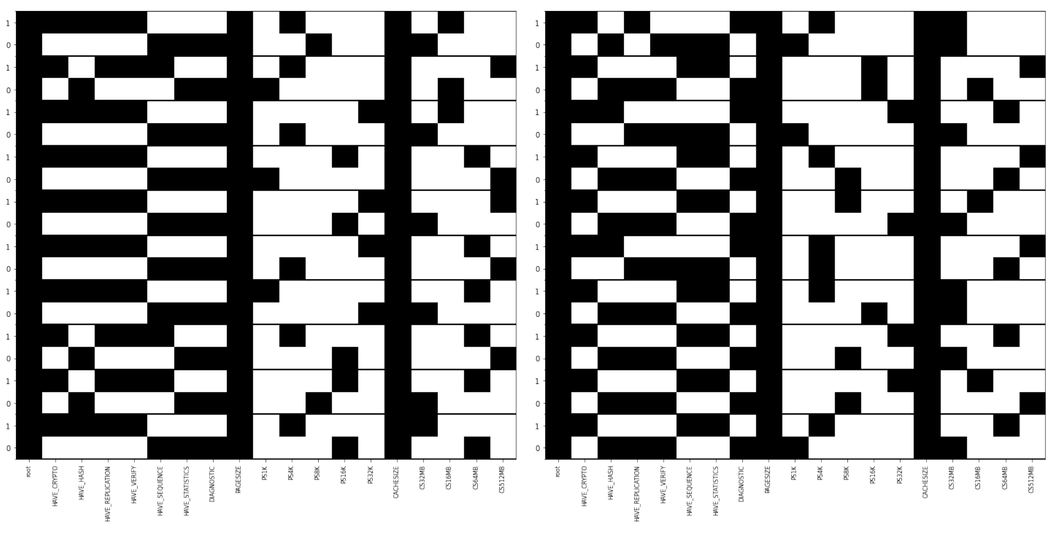

Figure 3.3: Samples generated on the BerkeleyDB Dataset by both variations to generate groups. **Left:** Group Sampling - Hemming Distance - Group Size 2 - Groupings 10 **Right:** Group Sampling - Independent features - Group Size 2 - Groupings 10

3.5 Influence model

With the groups created in the previous section, we can create a model, which predicts the performance of a given system. For this, we measure the non-functional property we want to predict for each group of features. Table 3.2 shows an example of such measurements. We only enable the features of one group at a time. The taken measurement then serves as the influence value of the group itself. In Table 3.2 each round of groupings is separated by a horizontal line. If we look at the first round of groupings, the influence value of G_1 is 3 and the features F_1 and F_2 are part of the group.

If in another measurement, the same configuration is chosen, we expect to see the influence value of the group as the measurement. Since measuring all possible groups (configurations) is not feasible, we want to determine the influence of single features to predict unseen configurations. By definition, only a few features are influential, this lets us assume that the influence measured in a grouping stems from only one or just a few features. In our example, this lets us assume that F_1 and F_2 have the influence of 3. With multiple different groupings, we can correct or confirm this assumption. The average influence value the feature has is our best approximation with the data available. If an influential feature is in a group, the group influence is most likely significantly higher. With enough groupings of different features, we can determine an influential parameter due to the fact, that the average influence of this feature is higher than that of the rest. In our example, we can see that the groups with F_6 assigned to them have, on average, a higher influence than the groups not containing F_6. With the average influence of the features, we can create a model of the system. We can use the formula for multiple linear regression as described by Kaya Uyanik and Güler [30].

$$y = \beta_0 + \beta_1 x_1 + ... + \beta_n x_n + \varepsilon \tag{3.6}$$

We adjust the formula so that our parameters β_n represents the influence of a feature. The feature influences we determined in Table 3.2 represent the measured non-functional property of the system and include its baseline performance. This means, the performance of the system if all features would be disabled. We need to compensate for this since it would otherwise make the prediction of the model unusable. We can do this by determining the baseline performance and subtracting it from the influence values. An approximation of the baseline performance would be the average of all measurements taken. The model constructed would be described by following forumlas:

$$f_0 = \frac{1}{n} \sum_{n=1}^{|I|} I_n \tag{3.7}$$

$$f_n = I_n - f_0 \tag{3.8}$$

$$y = f_0 + f_1x_1 + f_2x_2 + ... + f_nx_n + \varepsilon \tag{3.9}$$

Where I_n is the average of all measurements taken which included F_n and x_n is either one, if the feature is selected or 0, if the feature is not selected. We can use this formula to predict the behaviour of the system on unseen configurations, but the accuracy of the prediction is most likely not very good. We can see why if we look at F_3 and F_5 and their respective average measurements I_3 and I_4. They both have high average values, due to the fact, that they share a group with our influential feature F_6. The effect on their influence values due to sharing a group with an influential feature would average down if the number of groupings would be increased, but it is still a problem. We try to compensate for this problem with a stepwise analysis of the influence values in subsection 3.6.1.

Table 3.2: Feature groupings and their influences

G_1	G_2	G_3	R	F_1	F_2	F_3	F_4	F_5	F_6	I_1	I_2	I_3	I_4	I_5	I_6
1	0	0	3	1	1	0	0	0	0	3	3				
0	1	0	6	0	0	1	1	0	0			6	6		
0	0	1	28	0	0	0	0	1	1					28	28
1	0	0	2	1	0	0	1	0	0	2			2		
0	1	0	7	0	1	0	0	1	0		7			7	
0	0	1	25	0	0	1	0	0	1			25			25
										2,5	5	15,5	4	17,5	**26,5**

3.5.1 Multicollinearity

In regression analysis, having correlation among predictors is undesirable [31]. If two or more predictors in a multiple regression model have a linear relation, it is called multicollinearity. Multicollinearity increases the standard errors and makes the coefficient unreliable, decreasing their precision [32]. We can use the variance inflation factor (VIF) to find multicollinearity in our data. Alin [32] provides us with the formula:

$$VIF_i = \frac{1}{1 - R_i^2} \; for \; i = 1, 2, ..., k \tag{3.10}$$

Where R_i^2 is the coefficient of multiple determination of independent variable x_i on the remaining variables [32]. We can calculate the VIF by performing a regression on all independent variables except one, for each independent variable. Each created model gives us an R^2 value, which we can use in Equation 3.5.1 to determine the VIF for an independent variable. A VIF greater than 5 indicates a high correlation [31].

3.5.2 Feature interactions

With group sampling, we collect information about features grouped together. This includes the interactions between features. To make use of this information, we can add the interactions of features to our calculation of the influence of individual features. A way to do this is to treat interactions similar to features during the determination of feature influences. By assigning the group value to the interaction, if and only if all features involved in the interaction are in the group, we can estimate the influence of the interaction the same way as with features. In Table 3.3 we added each interaction $I_{i,j}$ between two features F_i and F_j to the table.

Table 3.3: Feature groupings and their influences

G_1	G_2	R	F_1	F_2	F_3	I_1	I_2	I_3	$I_{1,2}$	$I_{2,3}$	$I_{1,3}$
1	0	3	1	1	0	3	3		3		
0	1	6	0	0	1			6			
1	0	19	1	0	1	19		19			19
0	1	7	0	1	0		7				
						11	5	12,5	3		**19**

$I_{1,2}$, $I_{2,3}$ and $I_{1,3}$ are the interactions between the features and get assigned the group value if both features of the interaction are selected in the group. With a smaller group size, we capture more interactions during our measurements since more features are selected in one group. We can see in our example, that the interaction between F_1 and F_3 results in a higher measurement as if the features are grouped with another feature or are in a group alone. While we were able to capture some interactions, we can see that we did not capture the interaction between F_2 and F_3. To estimate the influence of all interactions we need a much larger sample size than to estimate the influence of features on their own.

3.6 Optimizations

3.6.1 Stepwise Influence

In section 3.5 we created a model of a system by taking the average value a feature has across groupings. In our example in Table 3.2 we were already able to see a problem Saltelli et al. [3] described with group sampling. Features that share a group with an influential feature look more influential than they truly are. Saltelli et al. describe a way to determine the influence more accurately by determining the influence of a single parameter at a time.

The idea is to remove the influence an influential feature has before determining the influence of the next influential feature. We can do this in two steps, for each feature we want to determine the influence for. First, we need to find the most influential feature we have in our data. Then we have to estimate its influence and remove it from our data.

If we look at Table 3.2 we can use the average measurements of a feature to identify the most influential. But, we have to be aware, that the average of the measurements still includes the baseline performance. By simply taking the highest measurement, we could end up picking the wrong feature, if, for example, the influential feature has a negative effect on the measurement. We want to pick the outlier value of the average feature measurements. One way to do this is to pick the feature where the average measurement is the furthest away from the average of all measurements. In Table 3.4 we can see, the average of all average measurements would be approximately 11.8, giving us F_6 as our outlier.

With the feature identified, we can remove it from our data. We estimate the influence of the feature the same way as previously, by taking the average of all its measurements. By subtracting the value from all measurements where the feature was part of the group, we can get a better estimate of the influence the rest of the features have. Then we remove the influential feature from our data and proceed with identifying the next influential feature. An example of this procedure can be found in Table 3.4. We identified F_6 as our most influential feature and removed it and its influences in other groups from the data. F_6 shared a group with F_5 and F_3 and after removing the estimated influence of F_6, both do not look as influential as in Table 3.2.

In each round, we identify the influence of one feature. With the influences of the features determined that way, we can build our model the same way as previously, but instead of using I_n in Equation 3.5 and Equation 3.5 we use the values determined by our stepwise analysis.

Table 3.4: Stepwise influence calculation

G_1	G_2	G_3	R	F_1	F_2	F_3	F_4	F_5	F_6	F_1	F_2	F_3	F_4	F_5	F_6
1	0	0	3	1	1	0	0	0	0	3	3				
0	1	0	6	0	0	1	1	0	0			6	6		
0	0	1	28	0	0	0	0	1	1					1,5	
1	0	0	2	1	0	0	1	0	0	2			2		
0	1	0	7	0	1	0	0	1	0		7			7	
0	0	1	25	0	0	1	0	0	1			-1,5			
										2,5	5	2,25	4	**4,25**	**26,5**

3.6.2 Feature coverage

In this work, we use an SAT-solver to create valid configurations. We can see the effects of it in Figure 3.3. Off-the-shelf SAT-solvers tend to find locally clustered solutions [13]. The repeating patterns in our samples are a result of it. The SAT-solver finds a solution to our constraints by changing as few variables as possible. Since at the beginning of each grouping, the constraints the solver has are similar, the SAT-solver gives a similar solution. This prevents us from making meaningful groupings since features regularly share the same group.

We adapt the distance-based sampling strategy introduced by Kaltenecker et al. [13] to help in creating more diverse groups. We use the Hamming distance as shown in Equation 3.2 as our distance metric and the uniform distribution as our probability distribution. Instead of measuring the distance from the origin, we measure the distance from the first group of the previously created grouping.

In detail, at the start of each grouping, we randomly pick a distance from the first group created in the previous grouping. We then set the distance as a constraint for our SAT-solver to avoid getting a similar first group. The following created groups are always dependent on the first group since they need to be distinct from each other. This way the groups created during each grouping do not follow a pattern of minimal change.

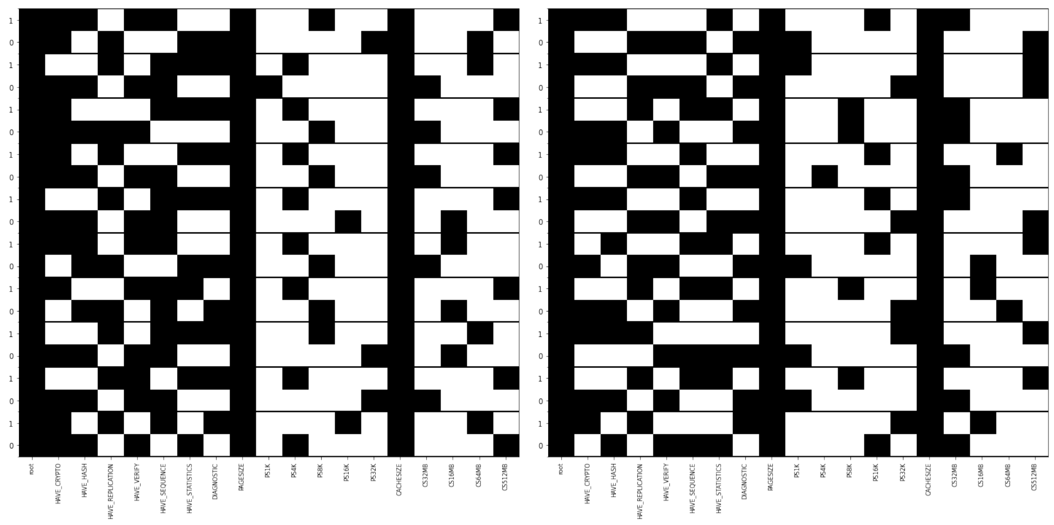

Figure 3.4: Samples generated on the BerkeleyDB Dataset by both variations with distance based group optimization. **Left:** Group Sampling - Hemming Distance - Group Size 2 - Groupings 10 **Right:** Group Sampling - Independent features - Group Size 2 - Groupings 10

Chapter 4

Evaluation

4.1 Datasets

The datasets used to evaluate the group sampling algorithm consist of real world software systems and synthetically generated data. We selected four software systems to be used to evaluate the group sampling algorithm on: Apache HTTP Server, BerkeleyDB, PostgreSQL and JavaGC. All possible configurations of all four real-world datasets are already measured and thus are not subject to any variation based on the testing environment. The real-world datasets are provided by the University of Leipzig. Since we only look at binary features in this work, numerical features are discretized into a number of mutually exclusive binary options [1]. This lets us use a unified approach to both numeric and binary features of a software system.

In the following section a short overview of our four real-world examples is presented and an explanation of how we generated our synthetic data.

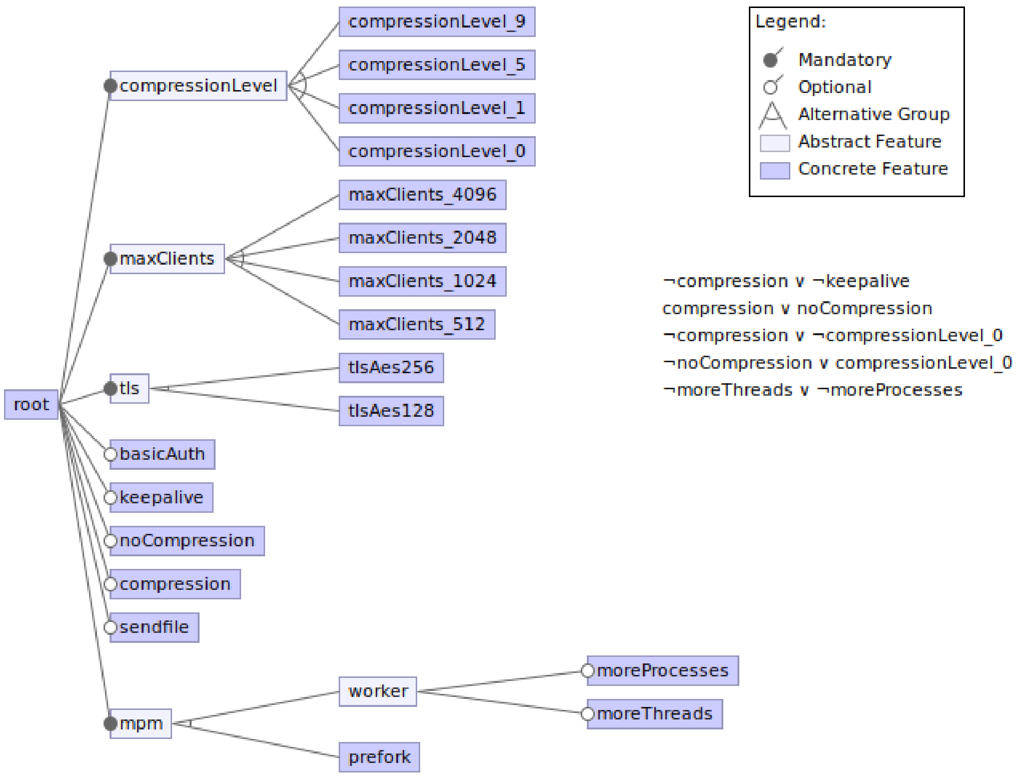

Figure 4.1: Apache HTTP Server feature diagram - created with FeatureIDE

Apache The Apache HTTP Server is an open-source web server developed by the Apache Software Foundation and is widely used today with a market share of 24.25% in 2021 [33]. The Apache dataset is the most restrictive of the real world datasets used with 5 cross-tree constraints and 5 optional features not contained in an alternative group.

Possible configurations:	13440
Measured:	Performance
Influential features:	*keepalive*
Independent features:	*sendfile*, *basicAuth*

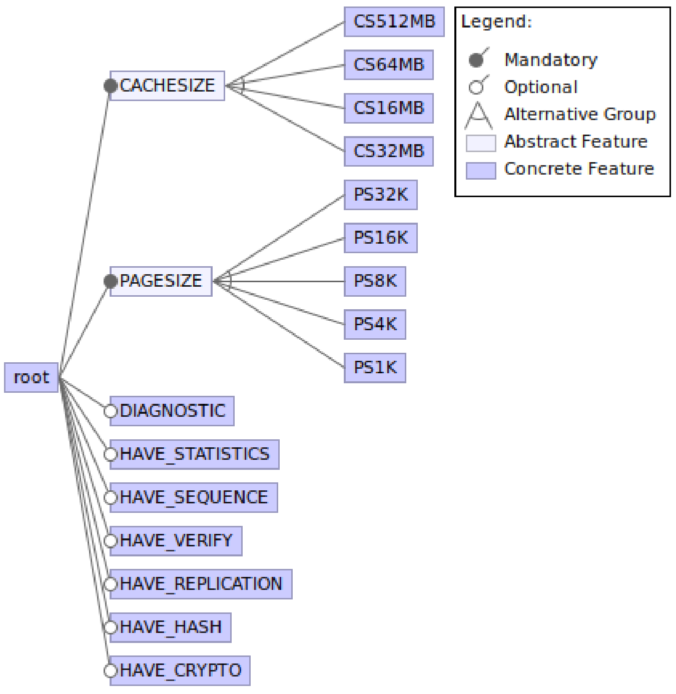

Figure 4.2: BerkeleyDB feature diagram - created with FeatureIDE

BerkeleyDB BerkeleyDB is an embedded database for key-value storage that originated at the University of California, Berkeley and is now owned by Oracle [34]. The BerkeleyDB dataset is the simplest dataset used for the real-world examples with two numerical values converted to an alternative group.

Possible configurations:	2560
Measured:	Memory
Influential features:	*PS32K*
Independent features:	*DIAGNOSTIC*, *HAVE_STATISTICS*, *HAVE_SEQUENCE*, *HAVE_VERIFY*, *HAVE_REPLICATION*, *HAVE_HASH*, *HAVE_CRYPTO*

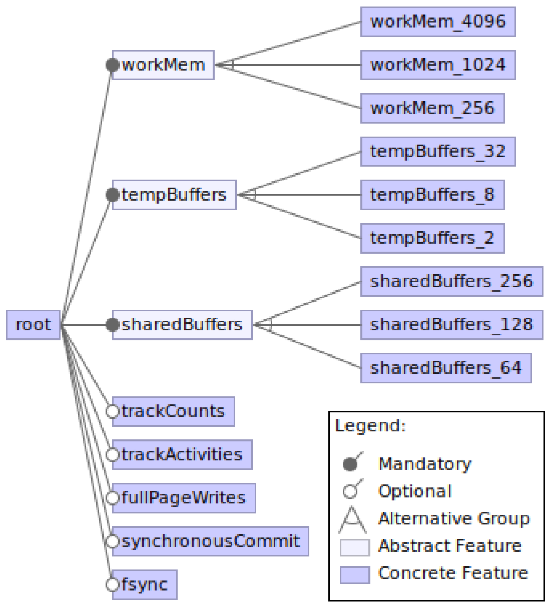

Figure 4.3: PostgreSQL feature diagram - created with FeatureIDE

PostgreSQL PostgreSQL is an open-source database management system supporting extensible data types, operators, aggregations and functions [35].

Possible configurations:	19008
Measured:	Performance
Influential features:	*fsync*
Independent features:	*trackCounts*, *trackActivities*, *full-PageWrites*, *synchronousCommit*, *fsync*

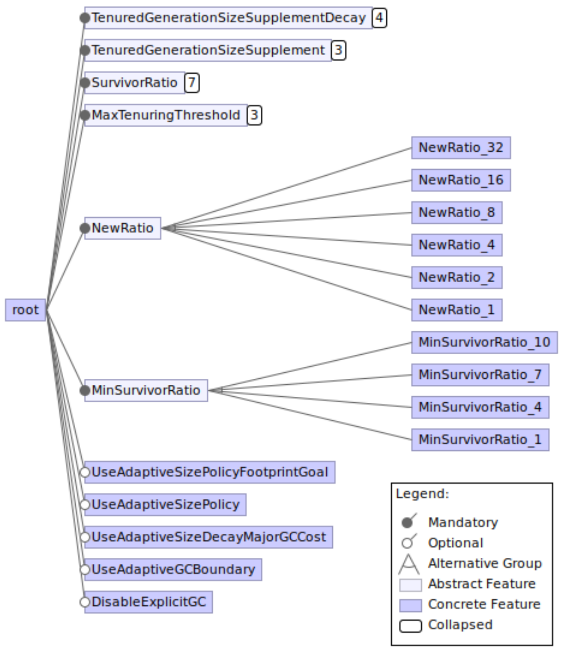

Figure 4.4: JavaGC feature diagram - created with FeatureIDE

JavaGC Java uses an automatic garbage collector (GC) to manage memory during the execution of a program. It frees the memory of object which are no longer referenced and therefore no longer used.

In the feature diagram in Figure 4.4 the features *TenuredGenerationSizeSupplementDecay*, *TenuredGenerationSizeSupplement*, *SurvivorRatio* and *MaxTenuringThreshold* are collapsed to make the diagram easier to read. All these values are discretized numerical features.

Possible configurations:	193536
Measured:	Performance
Influential features:	
Feature interactions:	*UseAdaptiveSizePolicy*, *SurvivorRatio*, *NewRatio*
Independent features:	*UseAdaptiveSizePolicy*, *UseAdaptiveSizePolicyFootprintGoal*, *UseAdaptiveSizeDecayMajorGCCost*, *UseAdaptiveGCBoundary*, *DisableExplicitGC*

4.1.1 Synthetic Data

Our real-world examples consist of relatively small feature models with around 20 features. Big software systems like the Linux kernel have more than 130,000 [36] features. To evaluate how the group sampling algorithm performs with a large dataset, we construct synthetic data to freely control the properties the feature model has. These let us use more easily determine the strength and weaknesses of our group sampling algorithm.

To generate the dataset, we specify the number of features, the number of constraints, the number of alternative groups and the number of influential features. The constraints are generated by randomly picking features and adding a random constraint between them. Either making them mutually exclusive or letting the first feature imply the second one. To add alternative groups to the feature model, we randomly pick a feature serving as the parent of the alternative group and add constraints among a random amount of the following features to make them mutually exclusive.

To make pseudo measurements on our feature model, each feature gets a weighting assigned to it. We then randomly pick the specified amount of influential features and assign each non-influential feature a random weighting from -5 to 5. The influential features are assigned a random weighting from 50 to 150. This lets us simulate the measurement of our generated feature model by calculating the sum of all weights of the selected features. We also add a base value of 200, to avoid getting negative simulated measurements.

We prepared three datasets for our evaluation. The first, which we will call *syn-100* contains 100 features, with four alternative groups, five random constraints and one influential feature. The second, *syn-500*, with 500 features, five alternative groups, ten constraints and one influential parameter. And the third one, *syn-1000*, with 1000 features, 10 alternative groups, 50 random constraints and two influential parameters.

We also prepared 4 datasets to test the group sampling on datasets with feature interactions. *SYN-10-Interactions*, *SYN-10-2-Interactions* with 10 feautres, one interaction and one influential feature. And, two bigger datasets *SYN-50-Interactions* and *SYN-50-2-Interactions* where *SYN-50-Interactions* has 50 features, one influential and one interaction and *SYN-50-2-Interactions* has 50 features, one influential and four interactions.

4.2 Setup

To evaluate the influence model created by the group sampling algorithm, the accuracy of the trained model and its predictions of not yet seen configurations need to be determined [37]. The distance between the predicted values and the actual values describes the accuracy of the model and can be used to compare different models on the same dataset. For this, we need data, which was not used during the training of the model. On our real-world examples, this can simply be done by randomly choosing samples from all known configurations. In all tests in this chapter, we use a fixed test sample size of 100 unless otherwise mentioned. We repeat the test 20 times and work with the average measurement of those tests unless otherwise mentioned.

To evaluate models like the one created by our group sampling algorithm, several well-known metrics exist. For simplicity, we use the MAPE metric to evaluate our model, Although it might not be perfect, it is easy to understand for humans because it gives the models average error as a percentage value.

Mean Absolute Percentage Error (MAPE) describes the distance in per cent between the actual value A_t and the predicted value P_t of all predictions n. MAPE can't be used if the values contain zero, since it would result in a division by zero.

$$MAPE = \frac{1}{n}\sum_{t=1}^{n}\left|\frac{A_t - P_t}{A_t}\right| \tag{4.1}$$

4.3 Preliminary tests

In order to answer our research question, we first need to determine an effective setup for the group sampling algorithm. For this, we try to answer two questions in this section.

Q_1 Which variant of group sampling is more effective?

In section 3.4, two variants to create groupings with the help of an SAT-Solver are described. While the creation of the influence model is the same, the actual result is dependent on the groupings created. To determine the superior strategy to create groupings, both variants are tested on our real-world examples.

Q_2 How can we determine the group size?

The results of the group sampling algorithm are dependent on the group size used during sampling. With bigger groups, more information for a single feature can be extracted, but it comes at the cost of more actual measurements required. We try to find a group size, which minimizes the measurements required while still giving us a decent model. As with Q_1, this is done by testing the group sampling on our real-world datasets with different group sizes.

To answer those question, we sample our four real-world examples with both variants described in section 3.4 and train an influence model as described in subsection 3.6.1. This is done with 2,3,4 and 5 groups and a sample size of one to 20. The models are evaluated on a test dataset of 100 samples and the MAPE value for each is calculated. The results can be found in Figure 4.5. Worth mentioning here is, that the sample size reflects the actual number of measurements needed, not the number of groupings done.

Q_1 Which variant of group sampling is more effective?

When comparing the models created by the first and second variants, "Hamming-Distance" and "Independent Feature", a fundamental flaw in the first variant is visible. If we look at the Apache dataset, for the group sizes of 3,4 and 5, the model has a constant MAPE value, indicating that the model does not learn the characteristics of the system. This is caused by the fact, that the number of features for each group, which the SAT-Solver tries to enable is smaller than the number of required features. If we look at Equation 3.3 the number of features per group is 8, 6, 4 for the group sizes of 3, 4, 5 respectively on the Apache dataset. With 10 required features on the Apache dataset, the SAT-Solver can not create a group with less than the required amount of features. This results in groups, where only the required features are enabled and thus completely leaving out all other features in the dataset.

Even if the inadequate group sizes are ignored, the grouping variant based on maximizing the Hamming distance is producing worse results than the grouping of independent features.

Q_2 **How can we determine the group size?**

The group sizes play a role in the determination of influential parameters. With too many features in a group, influential parameters are grouped together more often, making it harder to determine their influence. With too few features in a group, more measurements need to be made, which defeats the purpose of group sampling and starts to resemble an OAT-approach.

We have already seen that the grouping method based on the hamming distance does not work properly for certain group sizes. This method severely limits the selectable group sizes and makes it even harder to pick an appropriate group size. While selecting a meaningful group size with this method is possible if the amount of mutually exclusive groups and independent features is known, we will focus on the method of grouping independent features, since this method proves to be superior.

With the method of grouping independent features, the group size does not affect the amount of mutually exclusive features in a group. The group size only specifies how many independent features are in a group. This, for example, explains why in the Apache dataset the resulting accuracy of the model is not significantly changing. The influential feature in this dataset *keepalive* is part of a mutually exclusive group. We also can see, especially with only a few independent features, the group size plays close to no role in the accuracy of the model. Whereas the overall sample size plays a more significant role. On larger datasets with more independent features, the group size plays a more significant role. With more groups and fewer features in one group, the model gets more accurate. We can see this in Figure 4.6. The major drawback with more groups is the increased amount of measurements that have to be taken to determine the influence for each group. During our tests, we noticed diminishing returns with fewer features in a group until each group consists of a minimal amount of features. Group sizes of around one-tenth of the number of independent features but no less than 2 resulted, on average, in the most accurate model.

4.4 Results

In this section, all results to answer our research questions are visualised and discussed in section 4.5.

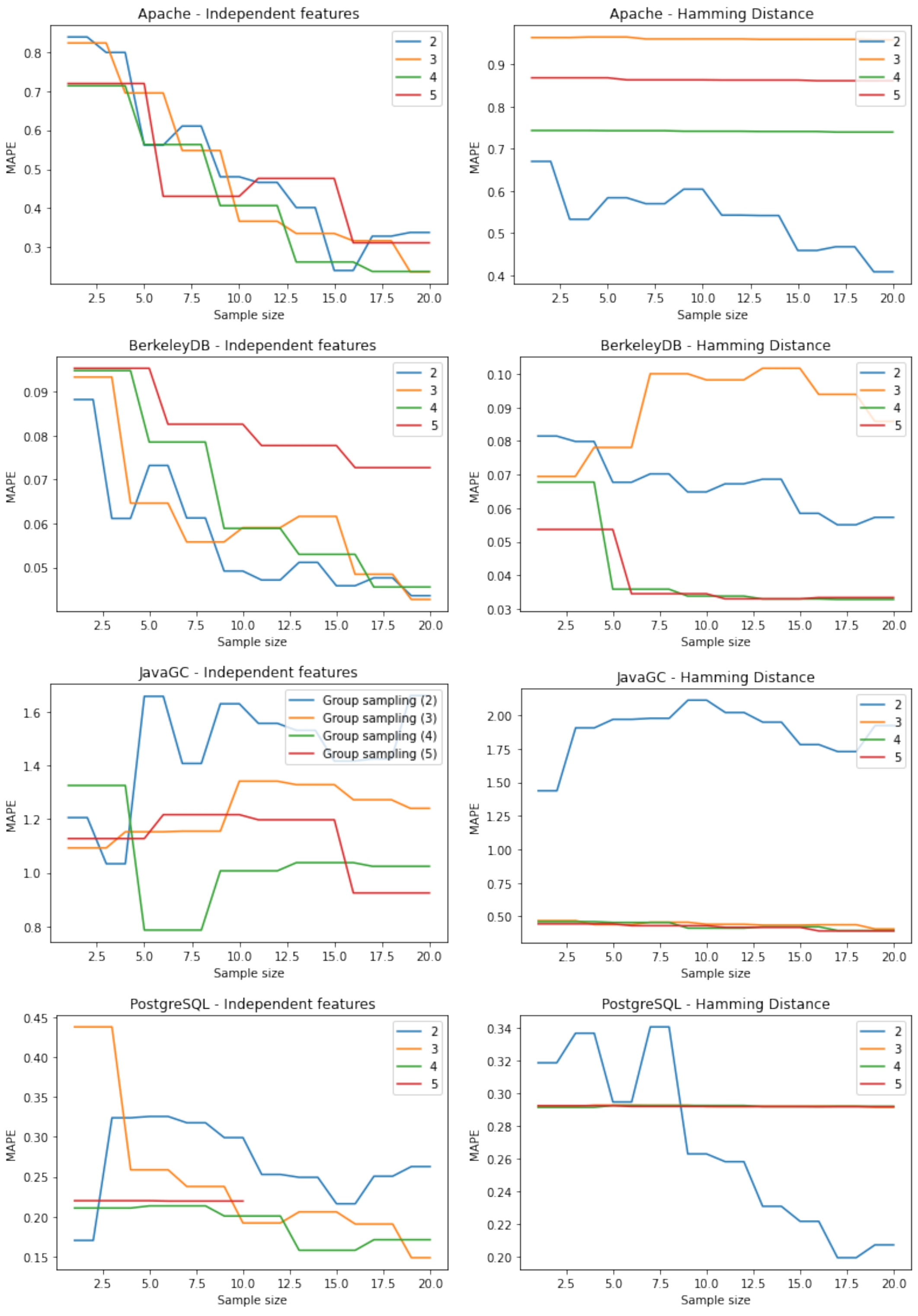

Figure 4.5: Different group sizes on the real world datasets.

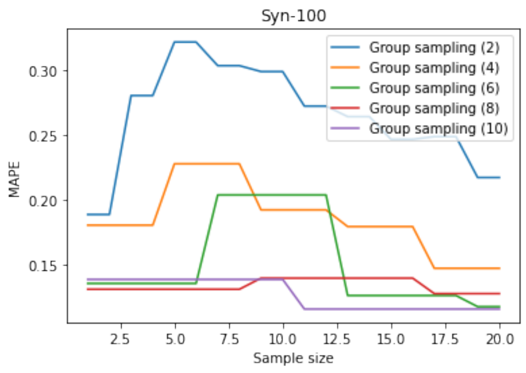

Figure 4.6: Different group sizes on the SYN-100 datasets.

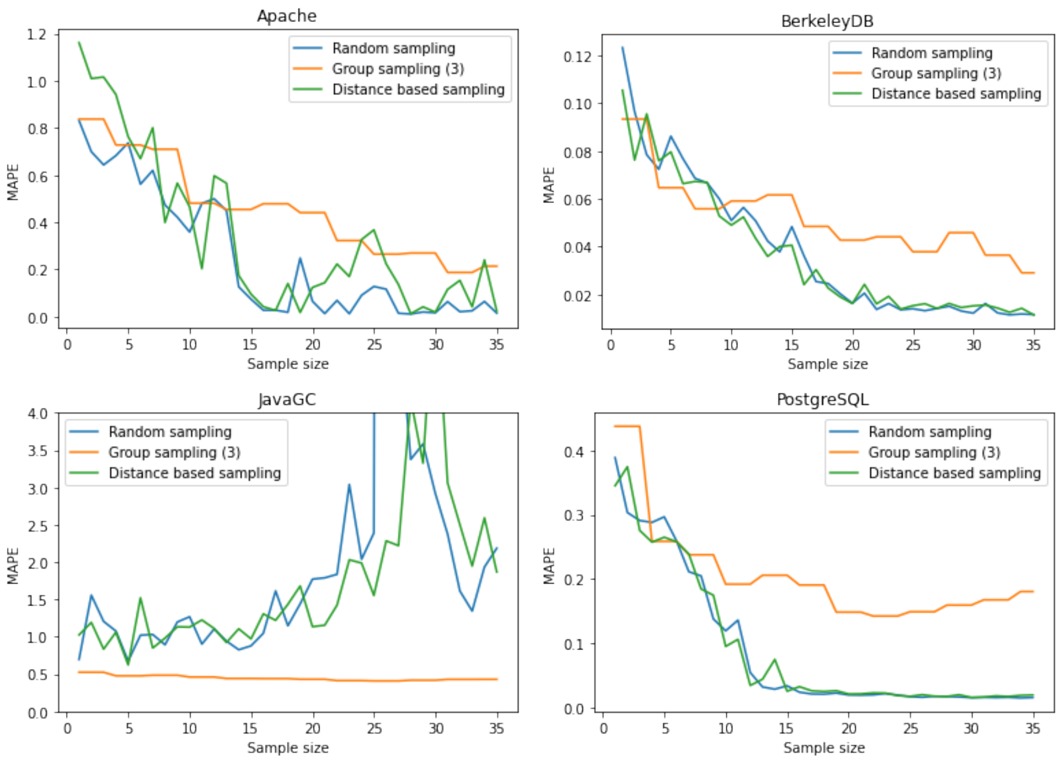

Figure 4.7: Group Sampling compared to random sampling with linear regression on real world examples

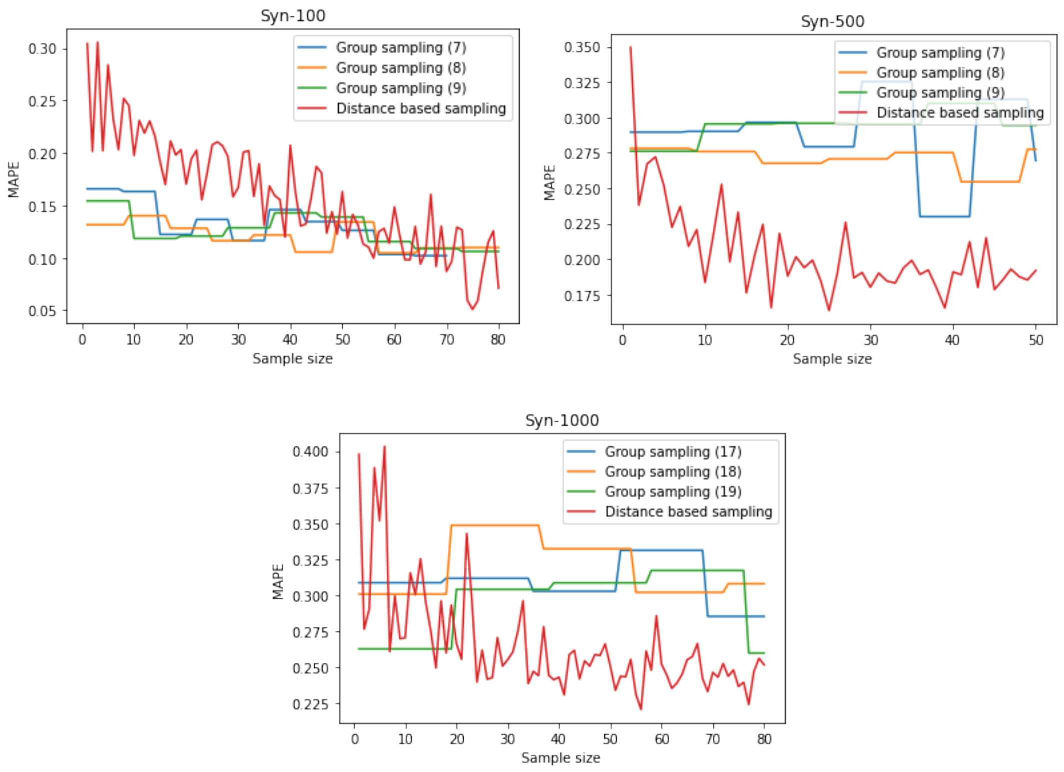

Figure 4.8: Group sampling compared to distance based sampling with linear regression on synthetic data.

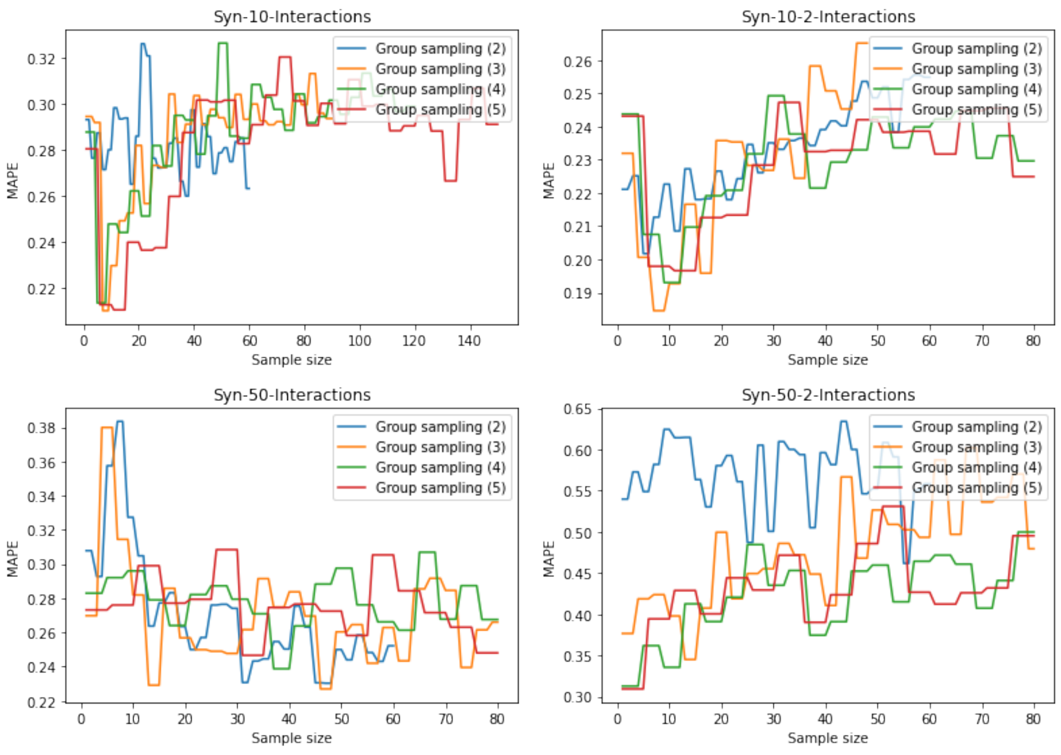

Figure 4.9: Group sampling with feature interactions

4.5 Interpretation

4.5.1 Mitigation of constraints

RQ1 How can we mitigate the effect of constraints on the configuration options for a group sampling approach?

In section 3.4 we described two methods to create groups of features to implement a group sampling approach. We identified groups of mutually exclusive features as the major problem in creating groups. The size of the smallest mutually exclusive group sets the maximum amount of groupings we can create, without having potentially influential features in multiple groups. Our solution is to weaken the constraint of having completely disjoint groups and allow mutually exclusive features to be in multiple groups in one round of groupings. While this allows us to create more groups, it limits our ability to make a statement about the influence of the feature. In section 5.2 we further discuss this solution and its influences. In our approach, we need to specify the number of features a group has, which leads to unusable groups in our first approach. Our tests show that more detailed knowledge about the feature model, especially mutually exclusive groups, drastically improves the quality of the created groups.

4.5.2 Effectiveness of group sampling

RQ2 How effective is a group sampling algorithm compared to random sampling?

To answer this question, we compared the group sampling algorithm, and its method to create a performance influence model, with a random sampling approach with linear regression. In our real-world examples in Figure 4.7 we can see, that the group sampling algorithm was mostly able to learn the influence of features. While the average prediction accuracy was good, it lacks behind compared to a more straightforward approach with random sampling and linear regression. Even though the group sampling algorithm seems to be ahead of the random sampling approach on the JavaGC dataset, it was not able to make any meaningful predictions about this dataset. The algorithm fails to identify any influential feature. This is mainly because the JavaGC dataset only has feature interactions that influence the performance. The group sampling algorithm, over multiple groupings, averages them down and simply predicts the average of all seen measurements. On the other hand, the random sampling algorithm with linear regression tries to fit the interactions once they appear in a big enough sample size, which makes the model unusable. In subsection 4.5.4 we look at feature interactions in more detail. In total, we can see that a group sampling approach to creating performance influence models results in worse models than a more straightforward approach like distance-based sampling [13] with linear regression. On our datasets with more features, the group sampling algorithm performs even worse.

4.5.3 Scalability

RQ3 How scalable is group sampling?

We tested the scalability of group sampling using multiple synthetically created datasets. In Figure 4.8 we can see the results. The group sampling algorithm struggles to create a meaningful model of the synthetic datasets. Even though, in the synthetic datasets more independent variables are available to create groups. The model does not improve with more samples, indicating an inability to learn more information about the features from the samples provided.

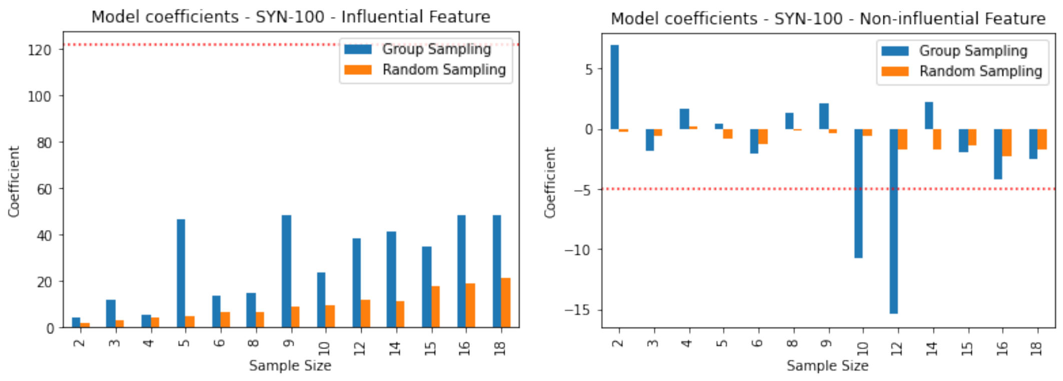

Figure 4.10: Resulting coefficients with different sample sizes.

If we look at the coefficients calculated with the larger datasets, we can see why. In Figure 4.10 the coefficients for an influential and a non-influential feature created by the group sampling algorithm and by random sampling with linear regression are depicted. Since it is a synthetically generated dataset we know the exact influence the feature should have. This influence is shown as the red dotted line. The coefficients show us, that the group sampling algorithm was able to identify the influential feature in the dataset with fewer samples even though it still wasn't able to estimate the influence correctly. With the correct identification of the influential feature, the model should be more accurate. Nevertheless, the group sampling algorithm is not able to correctly identify the influence of non-influential features. In the graph for the non-influential feature, we can see that it has a lot of variances when determining the influence of a non-influential feature. With a high average error among non-influential features, the group sampling algorithm produces worse predictions the more non-influential features are in the dataset. We further discuss the problem of inaccurate estimates on non-influential features insection 5.3.

4.5.4 Group sampling with interactions

RQ4 Can group sampling be used to effectively identify feature interactions and include them in the resulting model?

In order to answer if group sampling can identify feature interactions and include them in the model, we tested our approach described in subsection 3.5.2 on synthetically generated data. As with RQ3 the model's prediction accuracy did not follow an obvious trend and did not perform very well. In Figure 4.9 we can see how the group sampling algorithm performed with the inclusion of feature interactions as described in subsection 3.5.2. The initial gain in accuracy in respect to the sample size is caused by the group sampling algorithm identifying the influential feature in the dataset and only a few interactions appearing of which, most likely, none are influential. To explain the degrading performance with increasing sample size, it is, again, worth looking at the learned coefficients. Figure 4.11 shows the coefficients of the models trained with the highest sample count.

While the group sampling algorithm was able to identify the influential feature in all datasets, it only successfully identified the interaction in two out of four cases. A plausible explanation for this is that the number of groupings is not enough for an influential interaction to appear in the samples to train the model. We can also see a larger estimated influence in non-influential features and interactions. Since we treat interactions the same way as features, the average error on non-influential features and interactions make the model unreliable. We discuss a possible cause of the unreliable estimates on non-influential features in section 5.3.

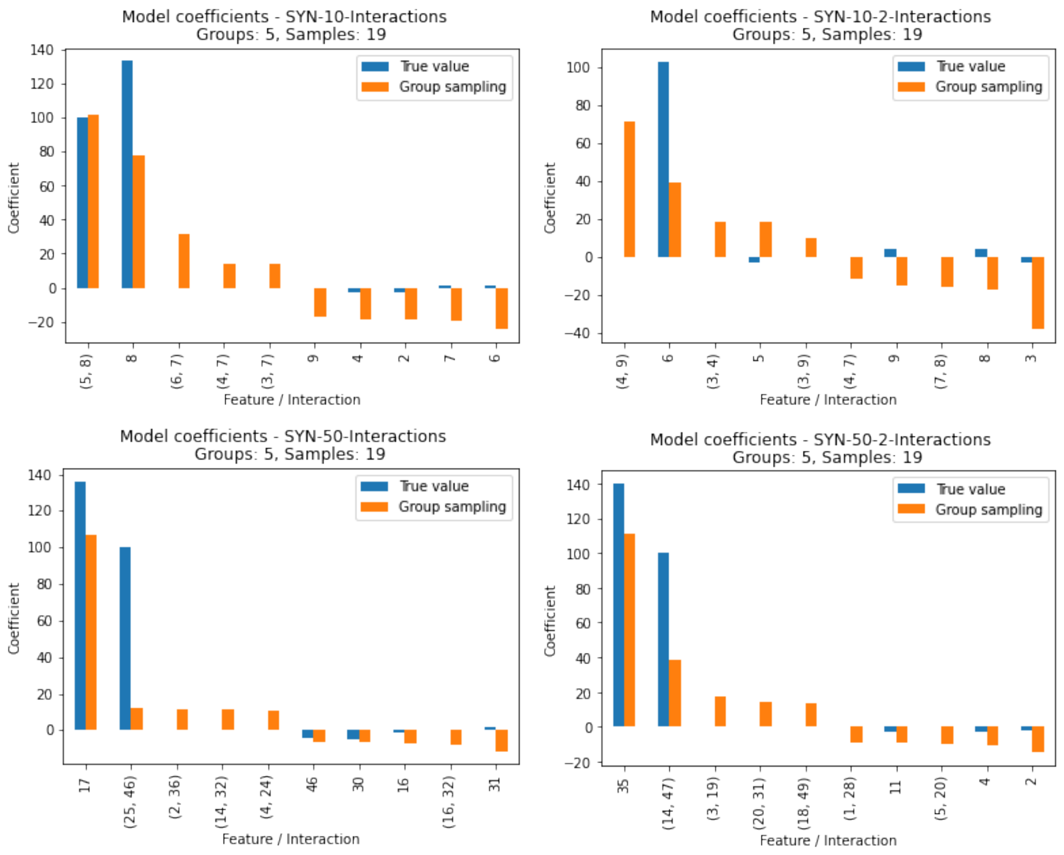

Figure 4.11: Feature and interaction coefficients - Top 5 positively influential and top 5 negatively influential

Chapter 5

Discussion

5.1 Features in multiple groups

To allow bigger group sizes than the smallest group of mutually exclusive features, we let mutually exclusive features to be in multiple groups in a round of grouping, which diverts from the setup of group sampling described by Saltelli et al.. Groups of two mutually exclusive features are not uncommon and would limit the number of groups to two groups. This would make learning the influence of features in bigger systems with group sampling more difficult. If, for example, a feature that is in more than one group is an influential feature, it would effectively render the groups the feature is in, as one group. All the groups would get assigned the influence of the influential feature. But even if the features, which are in multiple groups, are non-influential features, it causes some problems. Because it is enabled in more groups than other features, it is more likely to repeatedly share a group with an influential feature and make it look more influential than it actually is. Both problems, if the feature is influential and if it is not, could be avoided if the feature is not mandatory. It could simply be enabled just once per round of groupings. If the feature is mandatory, a way to mitigate the problem is by increasing the amount groupings made. More groupings and measurements made should make it easier to determine the influence of the feature in multiple groups.

5.2 Identifying influential features

We have seen the performance influence model created with group sampling does not perform as well as a more straightforward approach such as random or distance based sampling does. If the goal is to predict the performance of a system, group sampling showed to be less effective than, for instance, random sampling with linear regression. In subsection 4.5.3, we looked at the coefficients of the created model. While the group sampling algorithm was able to determine the influential features, the estimations of the created model were unreliable. Nonetheless, the information the group sampling algorithm gives us can be useful. With the identification of an influential feature, we can, for example, make assumptions about inefficient parts of the system. In Figure 5.1 and Figure 5.2 we can see that group sampling was able to identify the influential feature with fewer samples than distance based sampling with a linear regression on our synthetic dataset. This might be used to screen an existing feature model for further analysis.

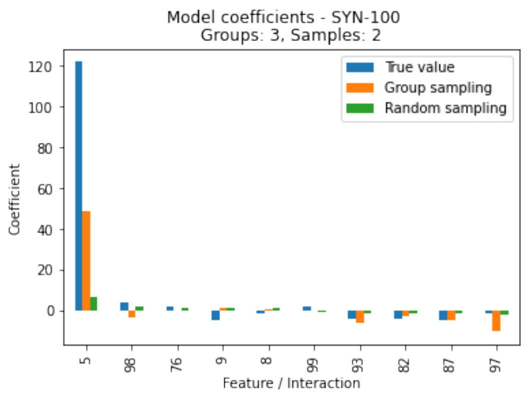

Figure 5.1: Model coefficients with 3 groups and 2 samples

5.3 Influence estimations of non-influential features

During our tests, we noticed that a major reason for the unreliability of our model is the inaccurate estimation of the influence of non-influential features. The coefficients for the model created by our approach to group sampling had a relatively high variance on non-influential features and often did not improve significantly with more samples. In Figure 5.1 we can see a lot of outliers in the estimated influence values. We can explain the positive outliers by features sharing a group with an influential feature too many times. Especially the outliers in mutually exclusive groups, since they are in more groups than independent features. The negative outliers can to some degree be explained by how we create the model. If we look at Equation 3.5 we determine f_0 by the average of all influence values. With a few groups, the average of all

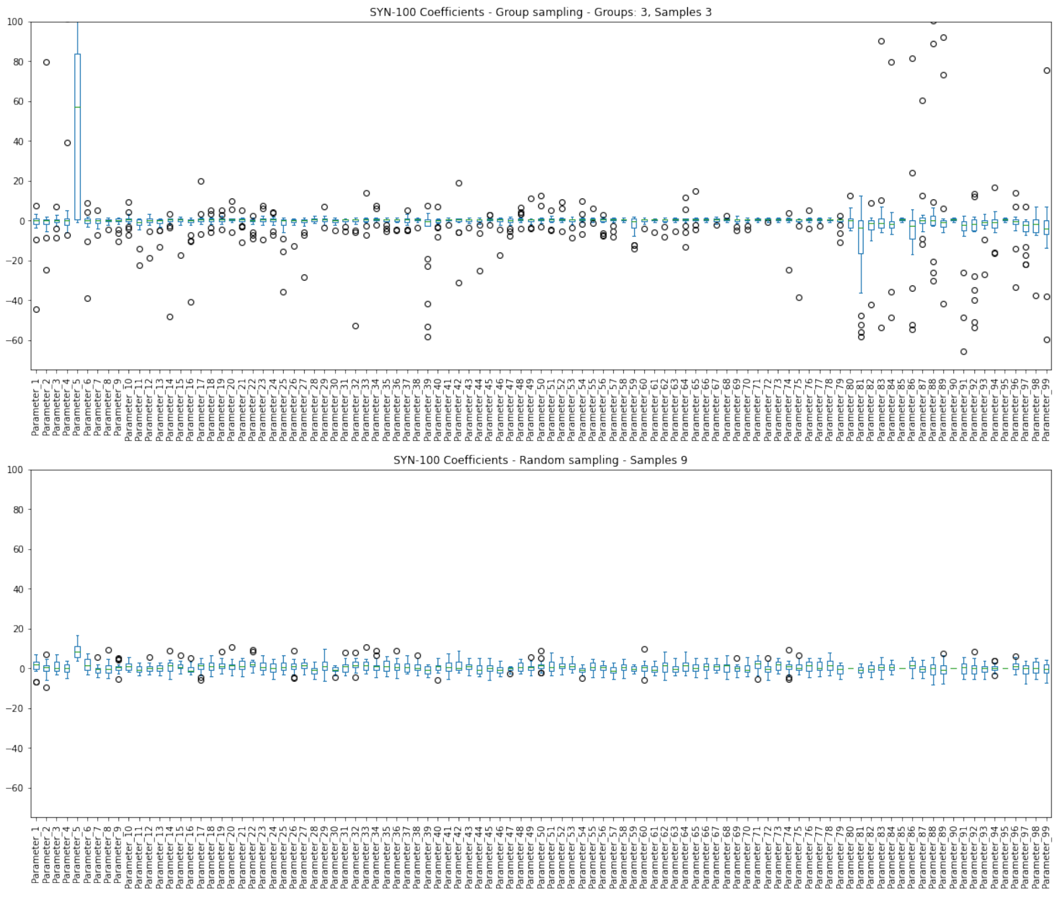

Figure 5.2: Model coefficients with 3 groups and 3 samples on the SYN-100 Dataset

influence values is significantly affected by the influential feature. If we assume we have a dataset with only independent features and apply a group sampling approach as we described with only two groups, in each round of grouping, half the features get assigned the value of the influential feature, resulting in an average of all influence values between the baseline performance and the influence of the influential feature. As Equation 3.5 shows, the value of f_n is dependent on the determined baseline performance f_0, making f_n inaccurate if f_0 is inaccurate. Especially features, which do not share a group with the influential feature are affected leading to a large error in the opposite direction of the influence of the influential feature. With more groups, this effect reduces, but it leads to significant problems with smaller group sizes.

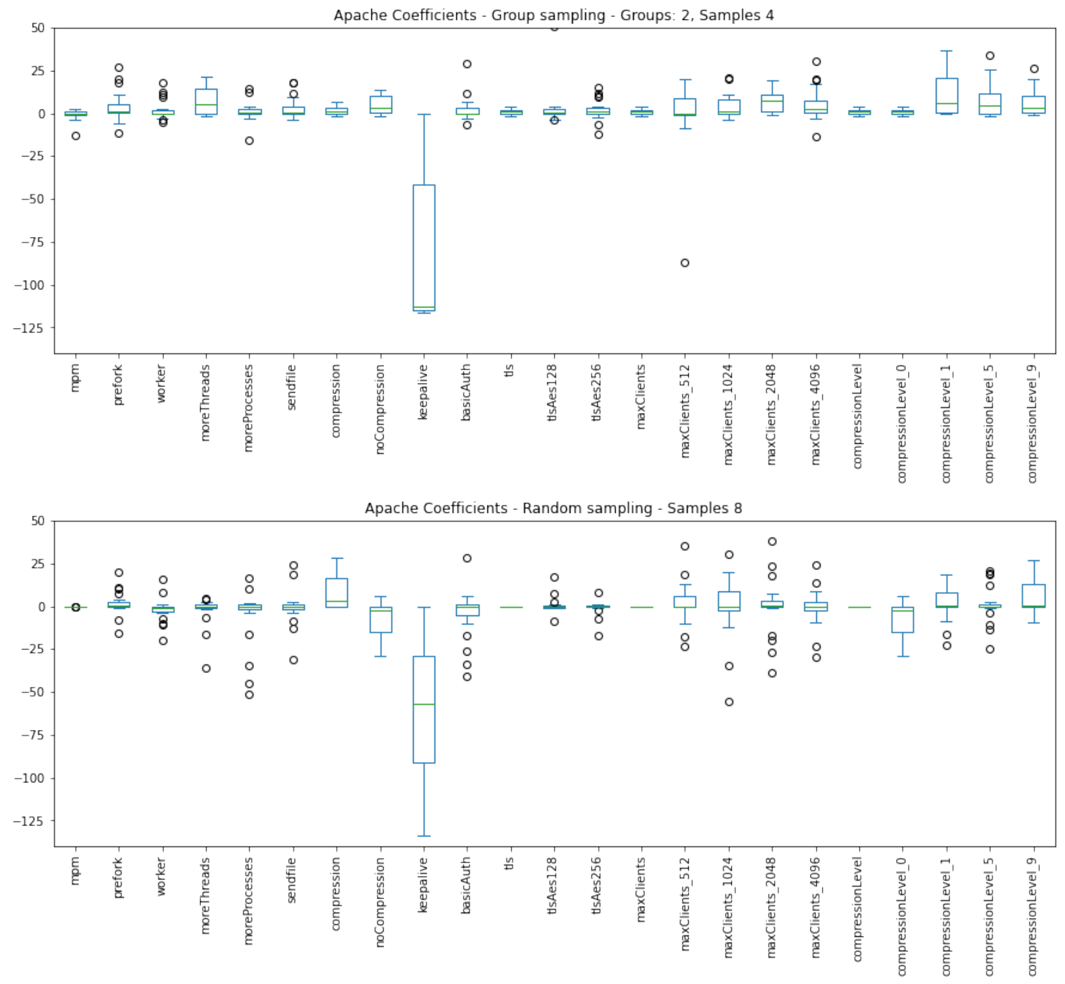

Figure 5.3: Model coefficients with 2 groups and 4 samples on the Apache dataset

Chapter 6

Conclusion and future work

6.1 Conclusions

In this work, we looked at how one can create a performance influence model using a group sampling approach. To do this, we devised two strategies to create groups among configuration options and their constraints with the help of an SAT-solver. We identified mutually exclusive features as the most limiting factor during group creation and allowed for mutually exclusive features to be in multiple groups to circumvent the issue. With the groups, we determined the influence of single features using the approach described by Saltelli et al. and created a performance influence model. This model was then evaluated on multiple datasets against a random sampling approach with linear regression. We also looked at how one could include feature interactions in the resulting performance influence model.

In our tests, we found that the group sampling approach while being able to identify influential features, did not result in a reliable model. We argue that this is due to the fact that the approach chosen to create the feature model is not able to accurately determine the influence of non-influential features. Leading to worse performance of the model on feature models with more features. With the inclusion of feature interactions, we were able to identify some interactions, if they appeared in the samples, but the model also proved to be unreliable due to the approach chosen. Since our approach treats interactions and features similar when determining the influences, the model also suffered from high average errors on non-influential features and interactions.

Even though group sampling was not able to make accurate predictions about a software system, it was able to identify influential features with only a few measurements.

6.2 Future work

Group sampling proves to be able to identify influential features with just a few measurements. With a round-based approach, this identification might even be possible with fewer measurements. By evaluating the influences after each round of groupings, one could create the following groups in a way to more easily identify an influential feature. Another interesting approach to perform sensitivity analysis on a feature model would be to apply the elementary effects method on groups of features maybe allowing for an efficient identification of influential features and interactions between features.

Bibliography

[1] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence models for highly configurable systems," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 284–294. 1, 2.2, 4.1

[2] T. Andres, "Sampling methods and sensitivity analysis for large parameter sets," *Journal of Statistical Computation and Simulation*, vol. 57, no. 1-4, pp. 77–110, 1997. 1, 3.1

[3] A. Saltelli, M. Ratto, T. Andres, F. Campolongo, J. Cariboni, D. Gatelli, M. Saisana, and S. Tarantola, *Global sensitivity analysis: the primer.* John Wiley & Sons, 2008. 1, 3.1, 3.2, 3.3, 3.3, 3.4, 3.6.1, 5.1, 6.1

[4] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, Tech. Rep., 1990. 2.1, 2.1

[5] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake, "Spl conqueror: Toward optimization of non-functional properties in software product lines," *Software Quality Journal*, vol. 20, no. 3, pp. 487–517, 2012. 2.1

[6] R. Malan, D. Bredemeyer *et al.*, "Functional requirements and use cases," *Bredemeyer Consulting*, 2001. 2.1

[7] J. Mylopoulos, L. Chung, and B. Nixon, "Representing and using nonfunctional requirements: A process-oriented approach," *IEEE Transactions on software engineering*, vol. 18, no. 6, pp. 483–497, 1992. 2.1

[8] M. Mendonca, M. Branco, and D. Cowan, "Splot: software product lines online tools," in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, 2009, pp. 761–762. 2.1

[9] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wąsowski, "Variability-aware performance prediction: A statistical learning approach," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 2013, pp. 301–311. 2.2

[10] W.-Y. Loh, "Classification and regression trees," *Wiley interdisciplinary reviews: data mining and knowledge discovery*, vol. 1, no. 1, pp. 14–23, 2011. 2.2

[11] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake, "Predicting performance via automated feature-interaction detection," in *2012 34th International Conference on Software Engineering (ICSE).* IEEE, 2012, pp. 167–177. 2.2

[12] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi, "Distribution-aware sampling and weighted model counting for SAT," *CoRR*, vol. abs/1404.2984, 2014. [Online]. Available: http://arxiv.org/abs/1404.2984 2.2.1

[13] C. Kaltenecker, A. Grebhahn, N. Siegmund, J. Guo, and S. Apel, "Distance-based sampling of software configuration spaces," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE).* IEEE, 2019, pp. 1084–1094. 2.2.1, 3.6.2, 4.5.2

[14] J. Oh, D. Batory, M. Myers, and N. Siegmund, "Finding near-optimal configurations in product lines by random sampling," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 61–71. 2.2.1

[15] Akers, "Binary decision diagrams," *IEEE Transactions on Computers*, vol. C-27, no. 6, pp. 509–516, 1978. 2.2.1

[16] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing*, 1971, pp. 151–158. 2.3

[17] O. Ohrimenko, P. J. Stuckey, and M. Codish, "Propagation via lazy clause generation," *Constraints*, vol. 14, no. 3, pp. 357–391, 2009. 2.3

[18] C. Sinz, "Visualizing sat instances and runs of the dpll algorithm," *Journal of Automated Reasoning*, vol. 39, no. 2, pp. 219–243, 2007. 2.3

[19] "MiniSat: A minimalistic, open-source SAT solver," [Accessed: 2022-01-23]. [Online]. Available: http://minisat.se/ 2.3

[20] "Batsat," [Accessed: 2022-01-23]. [Online]. Available: https://github.com/c-cube/batsat 2.3

[21] "Saturne," [Accessed: 2022-01-23]. [Online]. Available: https://github.com/jdrprod/SATurne 2.3

[22] "The z3 theorem prover," [Accessed: 2022-01-23]. [Online]. Available: https://github.com/Z3Prover/z3 2.3

[23] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon, "Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 650–670, 2014. 2.3

[24] D. Batory, "Feature models, grammars, and propositional formulas," in *International Conference on Software Product Lines.* Springer, 2005, pp. 7–20. 2.3

[25] A. Saltelli, "Global sensitivity analysis: an introduction," in *Proc. 4th International Conference on Sensitivity Analysis of Model Output (SAMO04)*, vol. 27. Citeseer, 2004, p. 43. 2.4

[26] L. Mutesa, P. Ndishimye, Y. Butera, J. Souopgui, A. Uwineza, R. Rutayisire, E. Musoni, N. Rujeni, T. Nyatanyi, E. Ntagwabira *et al.*, "A strategy for finding people infected with sars-cov-2: optimizing pooled testing at low prevalence," *arXiv preprint arXiv:2004.14934*, 2020. 3.1

[27] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Inf. Syst.*, vol. 35, no. 6, p. 615636, sep 2010. [Online]. Available: https://doi.org/10.1016/j.is.2010.01.001 3.3, 3.4.2

[28] C. Bron and J. Kerbosch, "Algorithm 457: finding all cliques of an undirected graph," *Communications of the ACM*, vol. 16, no. 9, pp. 575–577, 1973. 3.4.2

[29] R. Schröter, T. Thüm, N. Siegmund, and G. Saake, "Automated analysis of dependent feature models," in *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, 2013, pp. 1–5. 3.4.2

[30] G. Kaya Uyanik and N. Güler, "A study on multiple linear regression analysis," *Procedia - Social and Behavioral Sciences*, vol. 106, pp. 234–240, 12 2013. 3.5

[31] J. I. Daoud, “Multicollinearity and regression analysis,” in *Journal of Physics: Conference Series*, vol. 949, no. 1. IOP Publishing, 2017, p. 012009. 3.5.1, 3.5.1

[32] A. Alin, “Multicollinearity,” *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 2, no. 3, pp. 370–374, 2010. 3.5.1, 3.5.1

[33] “Web server survery,” [Accessed: 2022-01-13]. [Online]. Available: https://news.netcraft.com/archives/category/web-server-survey/ 4.1

[34] “Oracle berkeleydb,” [Accessed: 2022-01-13]. [Online]. Available: https://www.oracle.com/database/technologies/related/berkeleydb.html 4.1

[35] “Postgresql - feature matrix,” [Accessed: 2022-01-13]. [Online]. Available: https://www.postgresql.org/about/featurematrix/ 4.1

[36] C. Kaltenecker, A. Grebhahn, N. Siegmund, and S. Apel, “The interplay of sampling and machine learning for software performance prediction,” *IEEE Software*, vol. 37, no. 4, pp. 58–66, 2020. 4.1.1

[37] P. Domingos, “A few useful things to know about machine learning,” *Communications of the ACM*, vol. 55, no. 10, pp. 78–87, 2012. 4.2