Bauhaus-Universität Weimar
Faculty of Media
Degree Programme: Computer Science and Media

Generating Realistic Attributed Variability Models via Multi-Objective Optimization

Master's Thesis

Isabell Fidelak
Born July 4, 1992 in Arnstadt

Matriculation Number 117686

1. Referee: Prof. Dr.-Ing. Norbert Siegmund
2. Referee: Prof. Dr. rer. nat. habil. Andreas Jakoby

Submission date: December 5, 2018

Declaration

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

Weimar, December 5, 2018

. . . .
Isabell Fidelak

Abstract

Variability modeling is an evergrowing method for various industry and research applications, in importance as well as in usability. In this thesis the Thor, a tool for realistically attributing and generating interactions for variability models, is re-implemented in the Python 3 environment, which is easier to employ, and extended by a number of additional configuration options and functionalities. The most important addition to the new tool Loki is the functionality of modifying an existing attributed variability model in order to simulate changes in the system that is represented by the attributed variability model. This allows Loki to be used as a tool for evaluating transfer learning approaches, which aim to use information from the old model to quickly learn a fitting new model for the changed system. Lastly, Loki and its functionalities are evaluated with various test and realistic scenarios. The results show that Loki outperforms Thor in its qualitative results and that the functionality of evaluation transfer learning approaches is suitable.

Contents

List of Figures

List of Tables

List of Acronyms

AD	Anderson-Darling Test
AVM	Attributed Variability Model
EA	Evolutionary Algorithm
ED	Euclidean Distance
CNF	Conjunctive Normal Form
CSP	Constraint Satisfaction Problem
DM	Decision Maker
FODA	Feature-Oriented Domain Analysis
FPS	Fitness-Proportionate Selection
GA	Genetic Algorithm
KDE	Kernel Density Estimation
KS	Kolmogorov-Smirnov Test
MAPE	Median Absolute Percentage Error
MISE	Mean Integrated Squared Error
MOO	Multi-Objective Optimization
MOP	Multi-Objective Problem
NMSE	Normalised Mean Squared Error
NSGA	Non-Dominated Sorting Algorithm
PCC	Pearson's Correlation Coefficient

PVDD	Peformance Value Density Distribution
SAT	Satisfiability Problem
SPL	Software Product Line
SPLE	Software Product Line Engineering
SUS	Stochastic Universal Sampling
VM	Variability Model

Chapter 1

Introduction

1.1 Motivation and Problem Description

Variability modeling is an evergrowing method for various industry and research applications, in importance as well as in usability [Berger et al., 2013, Czarnecki et al., 2012].

If a company develops a new product, e.g. a smartphone, the combinations of available components alone - such as the type of battery, the CPU, sound chip, camera - result in a veritable number of possible configurations. If one factors in that not every combination is possible, since some components might exclude other ones, it quickly becomes more than a challenging task to manually analyze all possible combinations in order to determine the optimal setup for the new product. A screen covering the edges of the phone while having a slim case for example, might exclude a 3.5 mm headphone jack.

All the possible combinations of such a variable system including its constraints and features can be represented in a variability model (VM). In this model, the configurations of the system are formulated by variation points - e.g. the camera type in the case above - in terms of the system's features and constraints [Siegmund et al., 2017]. While the variability model describes which configurations are possible, it does not consider whether the configurations are advantageous regarding certain non-functional criteria specified by the user. In the case above, one combination might include a better camera than another, which in turn increases the configuration's, i.e. smartphone's, quality, but simultaneously causes a higher power requirement and cost, two potential criteria to consider. Factors like power consumption or camera quality are also called attributes and can be incorporated into classic variability models, which yields so-called attributed variability models (AVM) [Benavides et al., 2005]. These models allow their users to rate possible configurations according to specified attributes. One way of generating these attributes is by measuring different configurations of the system and estimating the influence of individual features and interactions, e.g. by employing machine learning techniques. In order for this approach to yield realistic values, a multitude of measurements are required. This is time- and resource-wise expensive, especially if you try to evaluate the system and find optimal configurations or if new features are added to the system later on.

A more favorable approach is the usage of a system which can generate values for AVMs. Proposals and implementations for such systems already exist [Segura et al., 2012], but they primarily rely on pre-defined distributions to generate the attributes' values which, in general, does not represent realistic scenarios well. Moreover, they are unable to take interactions between the features into consideration, which are proven to be common in real systems and can influence their performance [Apel et al., 2013].

Thor is a more recent system that is also used as the foundation of this thesis [Siegmund et al., 2017]. In addition to the aforementioned functionalities, this system is able to generate an AVM for one software system from its existing VM based on a previously generated AVM of another software system. This means the system analyzes the given attribute values, determines a density distribution and computes new values which are used to extend the VM. This results in a more realistic representation of the system. It also considers interactions between the features and offers the possibility of generating new ones. A disadvantage of this system is that the generation of attributes requires a lengthy computation time. In its current implementation, the system is configurable in the sampling method of the variants for creating diverse sampling sets (e.g. feature-wise or random sampling), the fitness metric for the employed genetic algorithm, the termination criterium and the weighted selection of results [Apel et al., 2018, Leutheusser, 2016]. Additionally, the installation of Thor requires various third-party dependencies, one of which is only free for academic use [Apel et al., 2018].

In the previously mentioned case, attribute values from an established system were used to attribute a new system in order to generate a new AVM in which a modeling or optimization technique can be applied to. This assumes two rather loosely related systems. In a different case two models could be so similar that they both describe the same system, except that in one model the software was updated. This slightly alters the system's properties, e.g. increasing the performance. Ongoing research focuses on the understanding of evolution of software systems [Cook et al., 2006, Neumann et al., 2014]. In order to assess the quality of new and existing approaches, a tool is needed with which an evaluation of the approaches becomes possible.

For this and other purposes, an additional useful functionality is the modification of already existing AVMs, e.g. by introducing noise or by applying a linear transformation to the observed data, which would simulate the slight performance changes between the two software systems. To our present knowledge such a functionality is not offered by any available program yet.

In order to close these gaps, we propose the system *Loki*, which offers the possibility of generating new AVMs either via transforming or rescaling their attribute-value profile while also considering and generating interactions between the model's features.

The purpose of this thesis is to:

- Re-implement the system Thor in Python 3.6 to reduce the amount of cross-language dependencies in order to increase the ease of use for the community.
- Extend the system with the functionality of modifying AVMs in order to enable the evaluation of transfer learning methods.
- Implement additional configurability of the system's settings and choices in algorithms to offer better customized approaches for problems. This includes but isn't limited to the genetic algorithm's selection method, the similarity measures used for the quality calculation or the used bandwidth for the kernel density estimation.
- Implement multi-threading to reduce the program's computation time.
- Evaluate the implementation.

1.2 Outline of this Thesis

Following the introduction, the theoretical background is addressed and illustrated in chapter 2. In chapter 3 related work to this field and their differences regarding this thesis is detailed. In chapter 4 Thor's re-implementation and it's extension are described. Chapter 5 covers the functionality of modifying existing AVM's. Both functionalities are evaluated in chapter 6. Finally, the results of the thesis are discussed and an outlook for future research is given in chapters 7 and 8 respectively.

Chapter 2

Theoretical Background

This chapter presents the theoretical foundation of this thesis. It offers a brief introduction to variability models in section 2.1. Section 2.2 explains the theory of constraint satisfaction problems, followed by a description of multi-objective optimization in section 2.3. Section 2.4 introduces Kernel density estimation, a method for estimating probability density functions. After that, genetic algorithms, their components and one popularly used algorithm - NSGA-II - are presented in section 2.5. The basic concept as well as different settings for transfer learning is addressed in section 2.6. Finally, section 2.7 focuses on five different similarity measures and how to use them.

2.1 Variability Models

Variability models define the common and variable characteristics, i.e. features, of products in a software product line (SPL) [Berger et al., 2010] and describe the variability of its application domain. Variability can hereby be defined as "the ability of a software system [...] to be efficiently extended, changed, customized or configured for use in a particular context" [Svahnberg et al., 2005]. They are the basis for many software product line engineering (SPLE) methodologies employed in research and industry [Berger et al., 2013]. The main goals of variability models are to facilitate the management of an SPL's growing and evolving set of features and the relationships among them, as well as support the variation and derivation of concrete products [Berger et al., 2010, Czarnecki et al., 2012]. If used correctly, variability models can reduce production costs, shorten the time-to-market and improve the overall productivity of an SPL [Acher et al., 2013].

Most variability models are either based on decision models or feature models. Decision models focus on organizing and managing decisions made in the configuration of products, while feature models focus on describing the feature space of a domain [Kang and Lee, 2013]. The latter will be explained in more detail in the next section.

2.1.1 Feature Models

[Czarnecki et al., 2002] describes feature modeling as "the activity of modeling the common and the variable properties of concepts and their interdependencies and organizing them into a coherent model referred to as a feature model." It is the most popular formalism for modeling commonality and variability [Acher et al., 2013] and was first introduced by Kang et al. in 1990 under the term Feature-Oriented Domain Analysis (FODA). It is a simple model which uses AND and OR dependencies to describe structural relationships -e.g. composition, generalization or specialization-, alternativeness and optionality between features [Kang and Lee, 2013, Kang et al., 1990]. Over the years, this model was extended by introducing new modeling primitives like the XOR dependency and feature cardinality.

A feature model is typically represented as a hierarchical, tree-like structure, which consists of nodes, directed edges and edge decorations [Acher et al., 2013, Czarnecki et al., 2002, Segura et al., 2014]. The features are hereby represented by the nodes, while the edges represent the relationships between them [Segura et al., 2014].Figure 2.1 depicts a simple example of a feature model.

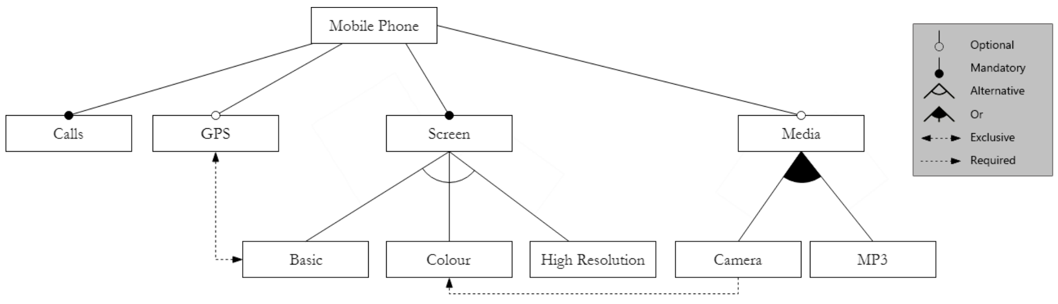


Figure 2.1: A simple example of a feature model. Adapted from Benavides et al. [2010].

[Kang et al., 1990] distinguishes between three different types of decompositional relationships:

- **mandatory**: *Iff* a feature's parent is included, the feature itself must be included as well. A mandatory feature node is denoted by a simple edge with a filled circle pointing to it. This relationship can be seen in Figure 2.1 between the root node *Mobile Phone* and the leaf note *Calls*.

- **alternative**: *Iff* the parent of a set of alternative features is included, exactly one of the features from this set must be included as well. A set of alternative feature nodes is denoted by edges that are connected by an arc. This relationship can be seen in Figure 2.1 between the node *Screen* and its leaf notes *Basic*, *Colour* and *High Resolution*.

- **optional**: *Iff* the parent of a set of optional features is included, every feature from this set may or may not be included as well. An optional feature node is denoted by a simple edge with an empty circle pointing to it. This relationship can be seen in Figure 2.1 between the root node *Mobile Phone* and the leaf note *GPS*.

Czarnecki et al. introduced an additional type

- **or**: *Iff* the parent of a set of or-features is included, at least one of the features from this set must be included as well. A set of or-feature nodes is denoted by edges that are connected by a filled arc. This relationship can be seen in Figure 2.1 between the node *Media* and its leaf notes *Camera* and *MP3*.

and Segura et al. mentioned the possibility of two different kinds of relationships between features that do not share the same parent, termed cross-tree relationships:

- **requires**: *Iff* feature A requires feature B and is included, then feature B must be included as well. A dotted arrow with one arrow head pointing away from the requirement denotes this relationship, which can be seen in Figure 2.1 between the leaf nodes *Camera* and *Colour*.

- **excludes**: *Iff* feature A excluded feature B and is included, then feature B must not be included. A dotted arrow with two arrow heads denotes this relationship, which can be seen in Figure 2.1 between the leaf nodes *GPS* and *Basic*.

2.1.2 Attributed Variability Models

Attributed variability models (AVM) are an extension of the conventional variability models, which provide additional information about its features [Karataş et al., 2013]. The need for such an extension was already expressed in 1990 by Kang et al., who noted the benefit of modeling relations between feature attributes in a quantitative manner to support the decision process of choosing features.

Benavides et al. proposed an explicit formulation for this idea in 2005. They introduced the term of extra-functional features, which describe the aforementioned relation between one or more attributes, i.e. measurable characteristic, of a feature [Benavides et al., 2005]. An attribute can be described as an influence on a model's quality measures, like the cost, runtime or required RAM of a software [Olaechea et al., 2012]. Each feature attribute has a domain which defines the space of possible values an attribute can take. This domain can be binary, discrete or continuous [Karataş et al., 2013].

The introduction of extra-functional features allows to express constraints and therefore apply filters to the model. One possible application could be the specification of a maximum cost. It is also possible to define criteria by which the products can be evaluated in order to find an optimal configuration, e.g. minimizing the program's runtime [Benavides et al., 2005].

2.1.3 Interactions

An interaction occurs when the combined inclusion of two or more features cause an unexpected influence on a model's quality measure [Siegmund et al., 2012]. This means that the sum of the feature's influences when activated separately is different from their influence when included together.

Interaction can be categorized by the number of involved features, which is referred to as their order. Thus, interactions of second order involve two features, interactions of third order involve three features etc.

2.2 Constraint Satisfaction Problems

A constraint satisfaction problem (CSP) is defined as a tuple (V, D, C), where V, D and C are finite sets of variables, domains and constraints respectively. Every variable $v_i \epsilon V$ has an assigned domain $d_i \epsilon D$, which describes the set of possible values the variable can assume. Every constraint $c_i \epsilon C$ is defined as a pair (R_i, S_i), where S_i defines a sequence of variables and where R_i resembles the subsets of the Cartesian products of the involved variables' domains [Petke, 2012, Rossi et al., 2006]. Constraints restrict the variables' range of allowed values and can be categorized by the number of variables that are involved. This categorization is referred to as scope. Thus unary constraints involve one variable, while binary constraints involve two variables. Constraints with a scope larger than two are termed n-ary [Mittal and Falkenhainer, 1990].

The core concept of CSPs is to assign a value to each variable from their respective domain, so that every constraint is simultaneously satisfied [Bayardo and Miranker, 1996]. Such an assignment is called solution to a CSP. If no such assignment can be determined, the CSP is called unsatisfiable [Apt, 2003, Rossi et al., 2006].

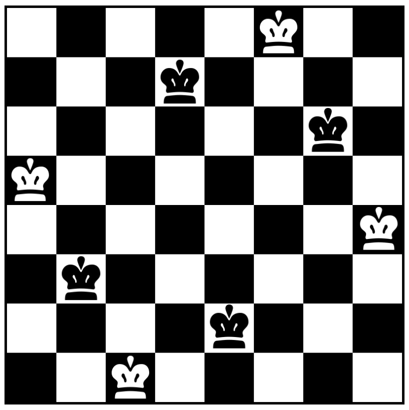


Figure 2.2: An example solution for the 8-Queens Puzzle.

A well-known CSP is the 8-Queens Puzzle, a problem where eight chess queens need to be placed on a regular-sized 8x8 chess board such that no two queens threaten each other. In this example, the queens are the CSP's variables, while the possible positions for each queen are their respective domains. The CSP's constraints are the conditions that no two queens share the same row, column or diagonal. The 8-Queens Puzzle is satisfiable, with a total of 92 solutions. One of them is shown in Figure 2.2.

A special case of CSPs are satisfiability problems (SAT), where each domain only consists of the two values $True$ or $False$ and where the constraints are given as a formula that must evaluate to $True$. If there exists no such value assignment, the SAT is called unsatisfiable. A simple example for an SAT is the formula (x AND y), which is satisfied if $x = True$ and $y = True$ [Hoos and Stützle, 2004].

Several methods can be used to solve CSPs. A naïve approach is to repeatedly generate value assignments for all variables, until one satisfies all constraints. More sophisticated approaches use algorithms that employ search, inference or a combination of both.

Backtracking is the fundamental algorithm for using search. In its basic form, values are assigned to variables successively and constraints are checked after every step. If a violated constraint is found the last decision is reversed and the algorithm tries another assignment. This is done in a systematic fashion to guarantee that all possible value assignments are tried until all constraints are satisfied again. Therefore, unlike with the naïve approach of brute-forcing solutions, constraints are found before a complete solution candidate is generated. Search trees are often used as a visual representation for a backtracking search, where each node below the root represents a decision of a variable's value and each branch represents a partial candidate solution [Rossi et al., 2006].

Inference, however, uses the specified constraints of a CSP to deduce and remove domain values that appear in no solution -referred to as redundant -, for a subset of the variables. It tightens the constraints itself as well. This supports the early detection of inconsistent partial solutions. Constraint propagation can be formalized with the two concepts of local consistency and rules iteration. Local consistency specifies conditions and characteristics of domain values to belong to a solution, while rules iteration defines a set of reduction rules based on the set of constraints, to rule out invalid values [Petke, 2012, Rossi et al., 2006, Tsang, 2014].

The oldest and best-known method of constraint propagation is arc consistency [Rossi et al., 2006]. Arcs are spanned by binary constraints between two variables (v_a, v_b). v_a is arc consistent *iff* for every value in the domain d_a there exists a value in d_b such that $(v_a, v_b)\epsilon c_a, b$ for every constraint v_a is part. Arc inconsistent values cannot be part of a valid solution and therefore can be removed. *Iff* all variables are arc consistent, the CSP is arc consistent as well [Larrosa and Schiex, 2004, Tsang, 2014].

2.3 Multi-Objective Optimization

Optimization can be defined as the process to find an optimal solution which minimizes or maximizes a given objective while satisfying mandatory constraints [Branke et al., 2008]. An objective can be defined in the form of a function [Luke, 2009]. In single-objective optimization, only one objective has to be optimized, which usually yields a single result. This solution can be called optimal solution [Branke et al., 2008].

Optimization problems which have to take more than one objective into account are called multi objective problems (MOP) [Talbi, 2009]. As opposed to handling single objective optimization, for MOPs, multiple objectives may conflict with each other [Branke et al., 2008, Bui and Alam, 2008, Luke, 2009]. Each objective itself has its own optimal solution, but considering all objectives at once, no single function can be optimal for all (conflicting) objectives. Owing to this, it is possible to find multiple solutions that usually will not have optimal results for a single objective, but result in an overall optimal solution with certain trade-offs between all objectives [Branke et al., 2008, Bui and Alam, 2008, Luke, 2009]. The process of finding an MOP's optimal solution is called multi-objective optimization (MOO) [Branke et al., 2008, Bui and Alam, 2008, Luke, 2009].

An example from the real world is the problem of finding a train connection with a short travel time and a cheap ticket price. One connection might cost 10 euros but takes 5 hours, while another one costs 20 euros and takes 2 hours, and yet another one costs 50 euros and takes 1 hour. With this problem, it is easy to see that a shorter travel time results in a higher ticket price, so one must find a satisfying trade-off between these two objectives.

One naïve approach to solve a MOO could be to bundle all objectives together into a single fitness function, using a linear function and treating the sum of the results as measurement [Luke, 2009]. It is also possible to use weights to differentiate more important objectives from lesser ones. This results in the problem of finding said weights, which might be unfeasible or even impossible if non-linear objectives are present [Luke, 2009].

The various optimal solutions which fulfill the objectives and constraints of a given problem can be grouped together in the so-called Pareto front. This front consists of candidate solutions which Pareto-dominate all solutions outside of the Pareto front. This is referred to as the solution being Pareto-optimal [Luke, 2009]. A solution Pareto dominates another solution if it has a better result for at least one of its objective while not being worse in the remaining objectives [Bui and Alam, 2008, Luke, 2009]. An alternative description of the solution's quality than being part of the Pareto front's solution set can be found in the Pareto strength [Luke, 2009]. The Pareto strength assigns each solution a strength and a weakness value. The former is defined by the amount of solutions the current individual dominates, while the latter describes the amount of solutions that dominate the current individual.

Another, more refined description is the so called wimpiness [Luke, 2009], which evaluates a solution by the sum of the total strength of every solution which dominates the individual, or formally defined:

$$Wimpiness_i = \sum_{\text{g}\epsilon \text{ that Pareto dominate i}} Strength_g \tag{2.1}$$

While a Pareto front may provide a multitude of viable optimal solutions to the given problem, a user normally requires only one optimal solution, which makes it necessary for the optimization process to choose a preferred solution. In [Branke et al., 2008, Bui and Alam, 2008] it is stated that MOO can therefore be defined as consisting of two equally important tasks: Finding the set of Pareto-optimal solutions, and choosing a single preferred solution. To achieve the latter, a so called decision maker (DM) is put to use [Branke et al., 2008, Bui and Alam, 2008]. The DM is a person who possesses knowledge of the problem and desired solution properties in the form of specific preferences for certain attributes of the solutions. With these, a DM ultimately makes the decision which solution in the Pareto front space to choose. The DM can be incorporated in a MOO process in various ways. In [Branke et al., 2008, Bui and Alam, 2008] the four most common types of methods are described, which are illustrated below:

- **A Priori**: The DM first articulates their preference information and its solution aspirations. With this information, the solution process solves for a Pareto-optimal solution, which satisfies the given preferences as well as possible. This method is advantageous for the DM in the way that they do bit need to invest much time into the optimization process - after the specification of his preferences, the process is fully automatic. However, the DM does not necessarily know if their specified preferences are feasible for the given problem and so it is possible that the solving process does not yield a good result.

- **A Posteriori**: In this method, the set of Pareto-optimal solutions is first generated. The DM then has to choose a solution from the set which best satisfies their preferences.

- **Interactive**: In the interactive method, the DM is involved in every iteration of the solution process: The iterative solution algorithm is defined and repeated. After every iteration the DM is given information about the current state of the possible solutions. They are then asked to specify their preferences, which will be taken into account during the next iteration. This, of course, is time consuming for the DM.

- **No preference**: The preferences of a DM are not taken into account, which results in the random selection of a solution from the Pareto front. This method is useful if there is no DM present or if there are no special preferences or expectations for the current problem in the first place, making a DM unnecessary.

2.4 Kernel Density Estimation

Being also known as Parzen's Window, Kernel density estimation (KDE) is one of the best-known methods for non-parametric density estimation. Unlike parametric density estimations, non-parametric approaches do not rely on the assumption that the data follows a specific probability distribution. This offers a greater flexibility in modeling the data [Botev et al., 2010, Chen, 2017].

Given a finite dataset F and $X_1, X_2, ..., X_n \epsilon \mathbb{R}$ as an independent, identically distributed random sample of size n drawn from F with an underlying density function f, the KDE is formally described by [Silverman, 2018] as:

$$\hat{f}(x) = \frac{1}{nh}\sum_{i=1}^{n} K(\frac{x - X_i}{h}) \tag{2.2}$$

where $K : \mathbb{R} \to \mathbb{R}$ is the kernel function and $h > 0$ is the bandwidth [Chen, 2017]. A commonly used kernel function is the Gaussian kernel [Sheather, 2004]:

$$K(x) = \frac{exp(-\frac{1}{2}x^2)}{\sqrt{2\pi}} \tag{2.3}$$

KDE takes the sample's data points and transforms each of them into a smooth peak; the sum of which yield the final density estimator [Chen, 2017]. This means that regions with a lot of data points have many peaks which result in a higher density value, while regions with only a few data points have fewer peaks and therefore a lower density value. This relation is illustrated in Figure 2.3.

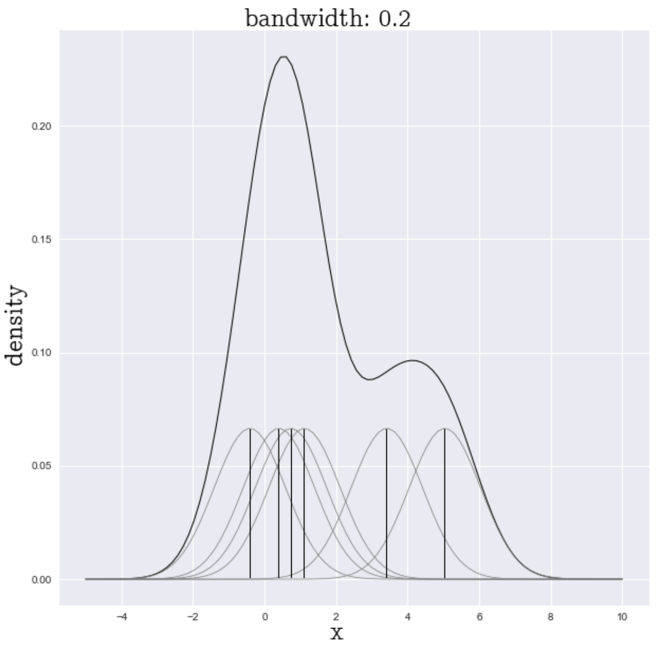


Figure 2.3: An illustration of how KDE is constructed. There are six observations (indicated by the black, vertical lines), which are smoothed into hills (grey curves). The sum of these bumps result in the final density estimate (black curve).

The overall shape of these data point peaks is determined by the Kernel function $K(x)$, and the peaks' width is dependent on the bandwidth. As illustrated in Figure 2.4, the choice of a suitable bandwidth is crucial for a good density estimation: The KDE will show a lot of unnecessary local minima and maxima if the bandwidth is chosen to small, a too large bandwidth on the other hand will smooth out important features [Chen, 2017, Sheather, 2004].

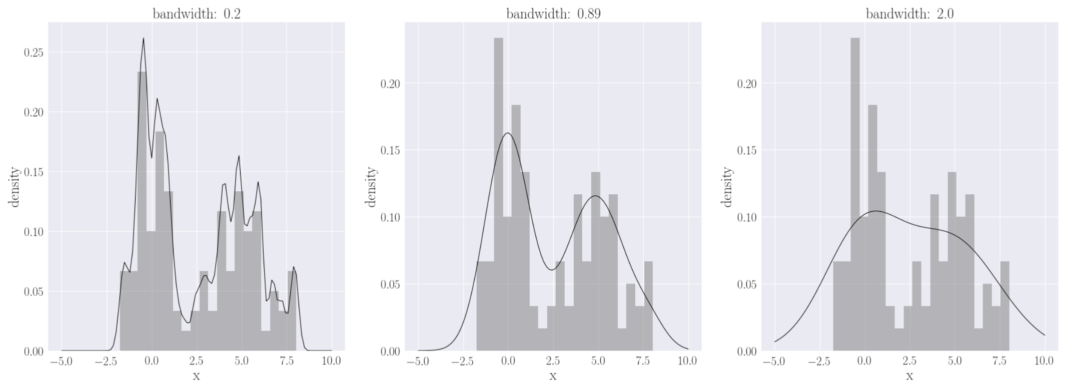


Figure 2.4: Three different bandwidths for KDE. The left image shows a bandwidth chosen too small (called undersmoothing), the right images shows a bandwidth chosen too large (called oversmoothing). The center images shows a suitable bandwidth, acquired by using cross-validation.

Finding the optimal bandwidth can be done by utilizing the mean integrated square error (MISE), also known as the L2 risk, which is formally defined by [Chen, 2017] as:

$$\int \mathbb{E}((\hat{f}(x) - f(x))^2)dx. \tag{2.4}$$

MISE is a function which measures the overall estimation error and gives an overall performance of the density estimator [Chen, 2017]. Smaller values of the MISE indicate a better bandwidth, which is why a lot of optimization methods try to minimize the MISE.

2.5 Genetic Algorithms

Genetic Algorithms (GA), which were first introduced by Holland in 1975 [Sivanandam and Deepa, 2010] belong to the class of Evolutionary Algorithms (EA) and are a stochastic search heuristic inspired by biology, genetics and Darwin's theory of evolution [Luke, 2009]. They derive their terminology from these fields. The ones used in this section are explained in Table 2.1. This class of algorithms mimics the process of natural selection, where well-adapted

Table 2.1: Terminology commonly used with EA, adapted from [Luke, 2009].

individual	a candidate solution
population	a set of candidate solutions
offspring	a new candidate solution obtained via breeding
parent	a candidate solution, which was used to generate offspring
fitness	quality
generation	one iteration of the algorithm; or the population produced in each iteration
gene	a particular string position on an individual
allele	a particular setting of a gene

individuals are able to survive long enough to reproduce and therefore pass on their genes to their offspring, which form the next generation.

Traditional GAs use fixed-length bit strings for representing individuals, where each position on this string is called a *gene* and portrays one particular feature. Each gene has a value assigned from a set of possible settings, which is referred to as an *allele* [Sivanandam and Deepa, 2010]. With a binary representation, these values are either 0 or 1, but individuals may also be represented by strings of integers or real numbers, where the set of possible values is much larger.

Each GA follows the same general sequence of steps, which is illustrated in Figure 2.5:

1. **Initialization:** The generation of an initial population of individuals. This is generally done in a random fashion, so that no accidental bias is introduced [Yu and Gen, 2010], which in consequence might prevent the algorithm from searching all regions of the solution space.

2. **Evaluation**: The computation of objective values for each individual in the population [Weise, 2009].

3. **Fitness Assignment**: The computation of a positive, real-numbered fitness value for each individual in the population using the objective values. The fitness indicates the quality of a solution and can be used for ranking the population [Weise, 2009].

4. **Selection**: Picking individuals for later breeding. Solutions with a higher fitness value generally have better chances of being picked in order to advance the optimization process [Weise, 2009].

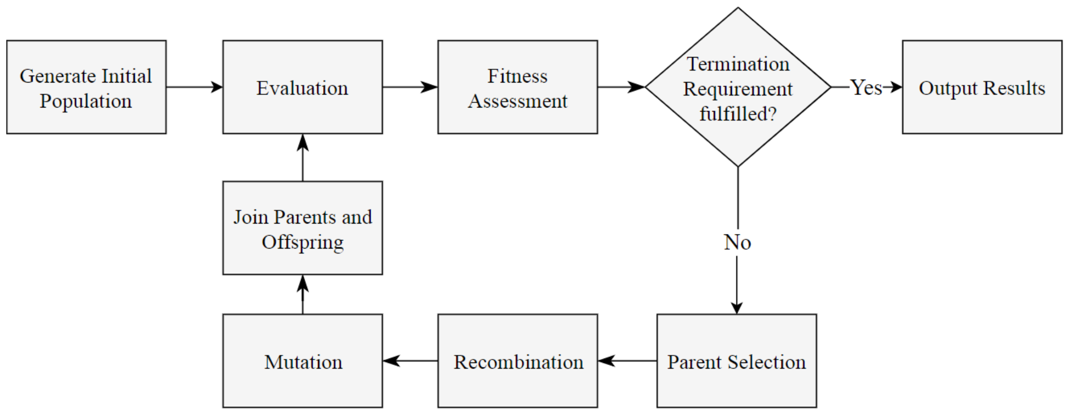

Generate Initial Population
Evaluation
Fitness Assessment
Termination Requirement fulfilled?
Yes
Output Results
No
Join Parents and Offspring
Mutation
Recombination
Parent Selection

Figure 2.5: Genetic Algorithm Procedure

5. **Recombination**: Using the selected parents and recombining their genes to generate new individuals [Luke, 2009]. This can be done by exchanging genes or by using the solutions' alleles in order to calculate new values. It allows the offspring to inherit alleles which yielded good solutions, and thereby encourage further exploration of that area of the solution space [Brabazon et al., 2015].

6. **Mutation:** The modification of the created offspring, usually via small, random changes [Luke, 2009]. It thereby allows the exploration of areas outside of the part of the search space that is spanned by the previous generation. This prevents the search process from converging prematurely [Brabazon et al., 2015].

7. **Joining:** The combination of parents and offspring. Usually, individuals with the highest fitness value become part of the next generation, while individuals with a bad fitness get discarded. It is also possible to simply replace all parents by their children.

The steps 2 - 7 are repeated until the algorithm terminates. This is generally bound to a condition, e.g. a defined number of generations that the algorithm is supposed to iterate through, an overall running time or a fitness improvement threshold from one iteration to the next.

The aforementioned components are explained in more detail in the following sections.

As opposed to algorithms which work on a single solution, GAs are population-based. This allows for parallel problem solving [Yu and Gen, 2010] and reduces the chance of converging to a local instead of the desired global optimum [Goldberg, 1989]. Another advantage is their robustness: There are no particular restrictions on problems to solve for, instead they can be adapted to a wide variety of tasks. Contrary to other search methods, GAs don't require derivative information about the problem at hand and work well with large solution spaces and large sets of parameters [Sivanandam and Deepa, 2010]. Like any other optimization technique, however, GAs too have their limitations: Due to their stochastic nature, there is no guarantee of finding optimal solutions, instead the algorithm might converge to a solution which it considers "good enough". GAs are also not well-suited for identifying local optima [Sivanandam and Deepa, 2010]. Finding a suitable encoding from features to genes might also be challenging depending on the problem.

2.5.1 Fitness Assessment

The fitness of an individual describes how well adjusted it is to its environment and therefore its odds of survival and reproduction [Fogel et al., 2000]. For EAs, this translates to the quality of a solution given a specific problem, i.e. how eligible a given individual is in solving the problem. It also offers information about the search space and about which regions may contain optimal or at least good solutions [Poli et al., 2008]. These two characteristics make it the driving force for the evolutionary process [Brabazon et al., 2015].

The fitness can be measured in many different ways, such as: The number of errors or differences between the real and desired output, the amount of time a program needs for its calculations, or the total cost of a process [Poli et al., 2008]. Additional information about the whole population, such as the overall diversity, could also be included [Weise, 2009]. This is achieved with the fitness function, which is defined as a transformation of the objective function. It uses the individual's objective values, compares them to the optimal or desired values and derives a non-negative, real fitness value [Weise, 2009]. In some cases, these two functions are identical. The fitness can be absolute or relative, with the former evaluating only the individuals quality and the latter evaluating this quality under consideration of the whole population's quality [Brabazon et al., 2015].

Assessing the fitness allows for the comparison of all individuals with each other and therefore a total ranking of the population. This ranking can be achieved with different methods such as the weighted sum ranking or the Pareto front assignment [Weise, 2009], which are explained in more detail in section 2.3. The fitness assessment is computationally expensive and consumes the vast majority of a program's runtime, since it must be performed once per generation and for each individual. Consequently, it is crucial to select a fitness function that is not only suitable for the problem but also time and resource efficient [Brabazon et al., 2015, Langdon, 1996].

2.5.2 Selection

The selection process picks a set amount of individuals as *parents*, which thereupon are used during the breeding process (see subsection 2.5.3 and subsection 2.5.4) to generate new individuals, in this context called *offspring*. This procedure can be carried out with or without replacement. In the first case, each parent might be used several times and therefore produce multiple offspring, whereas in the second case each parent is used only once. In the matter of GA, two parents are selected to generate two new offspring, which is done iteratively until the desired amount of new individuals is reached [Luke, 2009, Weise, 2009]. Selecting individuals is mostly done in regard to their fitness values, where fitter individuals have a higher probability of getting chosen as parents. This mimics Darwin's principles of natural selection and survival of the fittest. An alternative to this deterministic approach is a random selection of individuals, without considering their fitness [Weise, 2009, Yu and Gen, 2010]. The following subsections describe three different selection methods in more detail.

Tournament Selection

Tournament selection is the primarily used selection method for genetic algorithms. It randomly draws t individuals from the populations, compares them with each other and returns the one with the highest fitness value, which is then used for breeding. Here, t is a parameter called the *tournament size*, which is used for tuning the selection process. This implies that choosing a large value for t increases the probability of selecting the individual with the highest fitness, whereas choosing a small value for t increases the probability of individuals with smaller fitness values to be picked. Setting $t = 1$ results in selecting and returning a random individual. The most popular setting for t is 2 [Luke, 2009, Weise, 2009].

Fitness-Proportionate Selection

Fitness-proportionate selection (FPS), also known as *roulette selection*, picks individuals proportionate to their fitness compared to the accumulated fitness of the population [Weise, 2009]. An individual's fitness value therefore indicates its probability of getting selected. This probability can be calculated with:

$$p_i = \sum_{j=1}^{N} \frac{f_i}{f_j} \tag{2.5}$$

where N is the size of the population and f_i is the i^{th} individual's fitness. As illustrated in Figure 2.6, FPS can be interpreted as a mapping of the fitness values to contiguous segments of a line. Each segment's range is defined by its own fitness value, starting at the sum of its preceding individual's fitness values. Thus the line ranges from 0 to the accumulated fitness of all individuals, here called s. To select an individual a random number between 0 and s is drawn and the individual whose segment spans this number is picked.

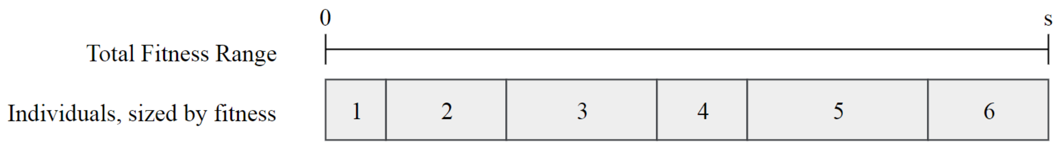


Figure 2.6: Fitness-Proportionate Selection. Adapted from [Luke, 2009].

Stochastic Universal Sampling

Stochastic universal sampling (SUS) works similar to FPS such that individuals are selected according to their fitness. Instead of using a single selection pointer however, SUS works with n uniformly distributed pointers, where n is the number of individuals to be selected. It uses the same mapping as FPS, but draws a single random number from the range 0 to s/n, increasing the number after each pick by s/n, and selecting the individuals whose segments span those numbers. This is illustrated in Figure 2.7. Individuals with a probability $p_i > s/n$ are guaranteed to be chosen at least once [Luke, 2009, Pencheva et al., 2009], which ensures zero deviation from the observed to the expected sampling frequency [Blickle and Thiele, 1995].

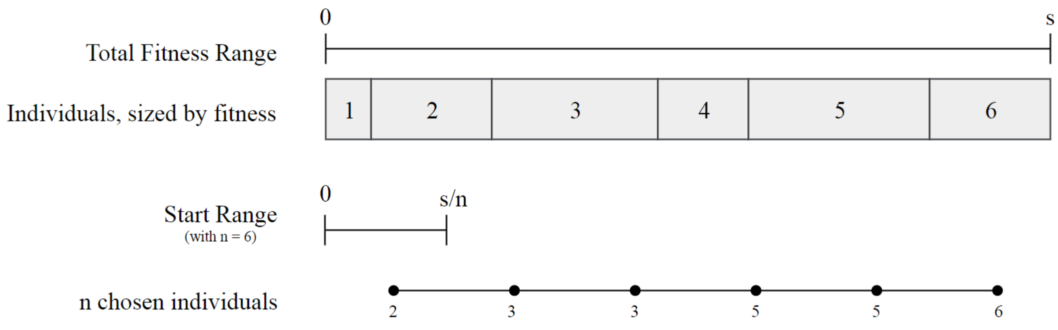


Figure 2.7: Stochastic Universal Sampling. Adapted from [Luke, 2009].

2.5.3 Recombination

Recombination is the first step of the breeding process and the distinguishing feature of GAs. It creates new individuals by exchanging or combining features of two or more existing parents, therefore it is referred to as sexual breeding [Luke, 2009]. While generating diversity, it also allows the offspring to inherit genes or building blocks from individuals that yielded good solutions, which in turn encourages a more exhaustive search around these individuals [Brabazon et al., 2015]. The following subsections describe five different recombination methods in more detail.

One-Point Crossover

A random value c between 1 and the individuals' length l is selected and every gene with an index smaller than c is swapped to produce new offspring. c will be assigned the value 1 with a probability of $\frac{1}{l}$ and no crossover takes place, which results in the parent individuals to also become the offspring individuals [Luke, 2009]. An example for the one-point crossover is depicted in Figure 2.8.

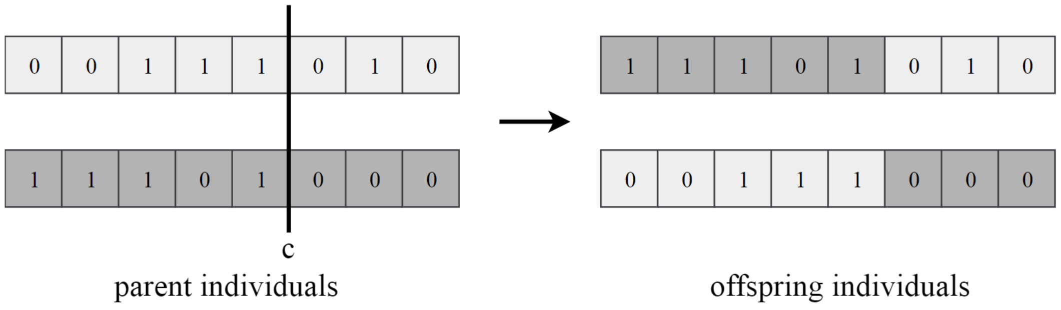


Figure 2.8: One-Point Crossover. Adapted from [Yu and Gen, 2010].

Two-point Crossover

Two random values c and d between 1 and the individuals' length l are selected and every gene with an index between c and d is swapped to produce new offspring. If c and d share the same assigned value no crossover takes place, which results in the parent individuals to also become the offspring individuals [Luke, 2009]. An example for the two-point crossover is depicted in Figure 2.9.

Universal Crossover

A probability value $p \leq 0.5$ is defined and for each gene a random value r between 0 and 1 is drawn independently. If $r < p$, the genes of both parent individuals are swapped [Luke, 2009]. For $p = 0.5$ both offspring individual are composed in equal shares of the parent individuals, whereas $p < 0.5$ causes both offspring to have a higher share of one of each parents respectively. An example for the universal crossover is depicted in Figure 2.10.

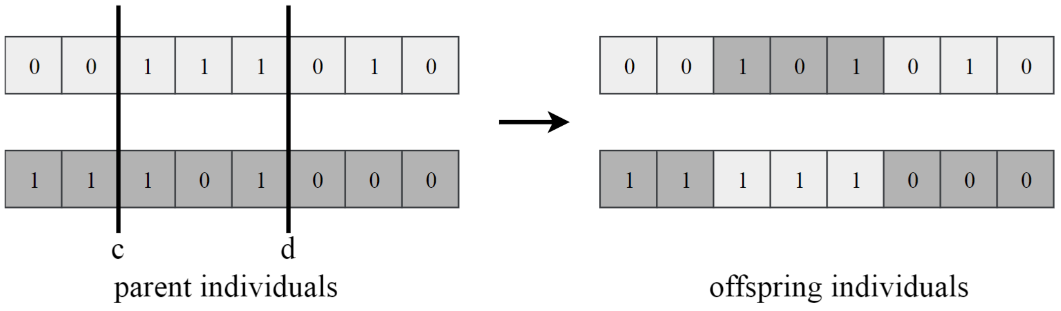


Figure 2.9: Two-Point Crossover. Adapted from [Yu and Gen, 2010].

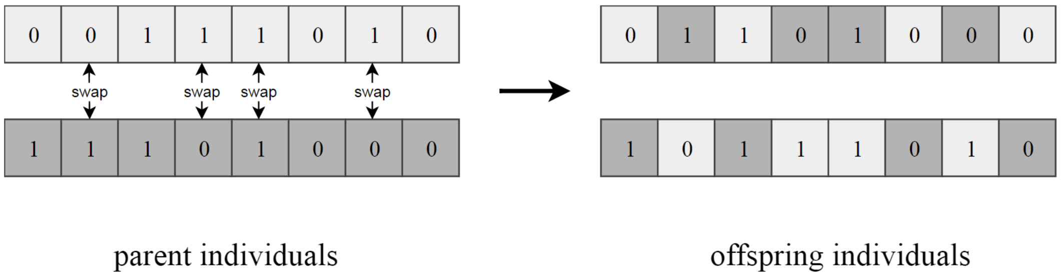


Figure 2.10: Universal Crossover. Adapted from [Yu and Gen, 2010].

Simulated Binary Crossover

Simulated binary crossover mimics the single point crossover's operating principle on binary strings for individuals with floating-point values. It uses a random value $u \epsilon [0, 1)$ and the parameter η, which is called the distribution index, to calculate the value β with the following formula [Deb and Beyer, 1999]:

$$\beta = \begin{cases} (2u)^{\frac{1}{\eta+1}}, \text{ if } u \leq 1 \\ (\frac{1}{2(1-u)})^{\frac{1}{\eta+1}}, \text{ otherwise.} \end{cases} \tag{2.6}$$

Choosing a high value for η causes the offspring individuals to be close to the parents with high probability, while a small value for η allows for the offspring to be more distinct from their parents. The most popular setting for η is 2 [Deb et al., 2007].

The obtained value β is then used in the following two equations to generate two offspring individuals:

$$c_1 = 0.5[(1+\beta)p_1 + (1-\beta)p_2] \tag{2.7}$$

$$c_2 = 0.5[(1-\beta)p_1 + (1+\beta)p_2] \tag{2.8}$$

where c_1 and c_2 are the two offspring and p_1 and p_2 are the two parent individuals [Deb and Beyer, 1999].

Line Recombination and Intermediate Recombination

Line recombination and intermediate recombination are used for individuals with floating-point instead of binary values. Here each individual can be interpreted as a point in hyperspace with their genes' values as coordinates and with a line drawn between them. A scaling factor $p \geq 0$ is defined and two random values α and β between $-p$ and $1+p$ are drawn. With line recombination, those values are drawn once per offspring individual whereas with intermediate recombination they are drawn newly for each gene. Those values are then used in the following formulas to generate the offspring individuals [Luke, 2009]:

$$t_i = \alpha v_i + (1-\alpha)w_i \tag{2.9}$$

$$s_i = \beta w_i + (1-\beta)v_i \tag{2.10}$$

where t_i and s_i are the offspring individuals' genes and v_i and w_i are the parent individual's genes at the i^{th} position. For $p = 0$ both offspring's positions will be on the line inside the hypercube spanned by the two parents, whereas for $p > 0$ the offspring individuals may also be located outside of the hypercube.

2.5.4 Mutation

Mutation is the second step of the breeding process and has the objectives of ensuring a continuous exploration of the search space and to prevent solutions from converging on local optima [Fogel et al., 2000]. It creates new individuals by modifying existing ones through a random change, hence it is referred to as asexual breeding [Luke, 2009]. The extent of this change is controlled by the *mutation rate*. Choosing an appropriate setting for this rate is an important task, since a value which is too large might turn the EA into a random search, while a value which is too small will not be able to introduce enough novelty [Brabazon et al., 2015]. The following subsections describe two different mutation methods in more detail.

Gaussian Convolution

Gaussian convolution is used for real valued individuals. For each of an individual's genes, some random noise drawn from a Gaussian distribution is added to their value with probability p. The distribution is characterized by its mean μ and its variance - in this context the mutation rate - σ^2, which influences the distribution's width [Luke, 2009]. This implies that small values for σ^2 induce a higher probability of random noise to be close to μ, whereas larger values for σ^2 increase the probability of random noise being spread further away from μ. This relation is illustrated in Figure 2.11. For the Gaussian convolution $\mu = 0$ and $\sigma^2 = 1$ are commonly used, since it leads to most random noise being close to 0, with only occasional occurrences of larger values.

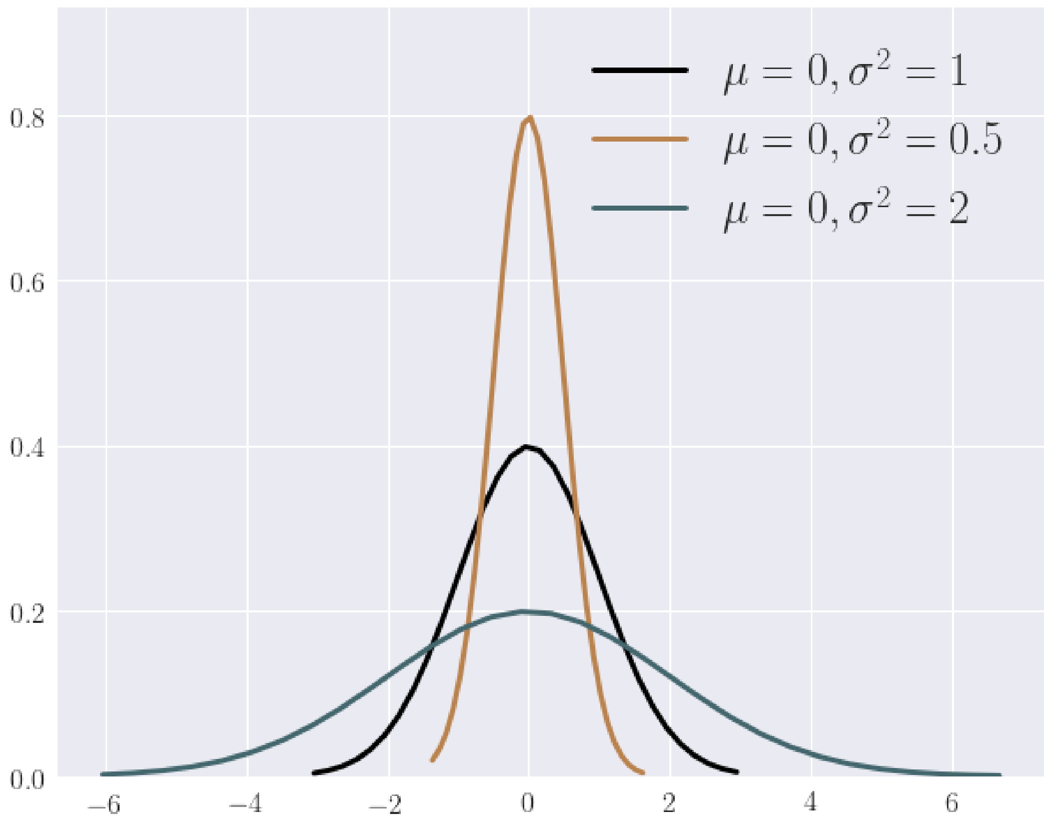


Figure 2.11: Different probability density functions. The black curve shows the Gaussian distribution.

Bit-Flip

Bit-flip mutation is used for individuals with boolean values. Every gene in an individual has the probability p of being inverted, i.e. a 0 is changed into a 1 and vice versa. This probability is often set to $1/l$, where l is the individual's length. This results in one inverted bit per individual on average [Luke, 2009, Yu and Gen, 2010].

2.5.5 NSGA-II

The Non-dominated Sorting Genetic Algorithm (NSGA) was proposed in 1994 by [Srinivas and Deb, 1994]. It is a multi-objective GA, that assigns ranks and correlating fitness values to individuals. This results in individuals of the same rank also having the same fitness [Yu and Gen, 2010]. The rank assignment is based on the Pareto optimality, where individuals of one rank Pareto-dominate all individuals of ranks with a higher order. Individuals of the last rank do not Pareto-dominate any other solution. A differentiating characteristic of NSGA is the use of a so-called *sharing-function*, which is used to modify a rank's fitness depending on the distance between its individuals. It rewards ranks with high distances with a fitness value increase, but leaves rank with small distances unchanged [Deb et al., 2002].

NSGA-II is the successor of NSGA and was introduced by Deb et.al in 2000. Like its predecessor, NSGA-II uses the assignment of ranks and a distance measure to guide the optimization process, but was improved by introducing the following three features:

- **Fast non-dominated sorting approach:** This approach ranks the individuals of a population according to their fitness value. It assigns two values to each solution: The domination count n_i, which is the number of individuals that dominate solution i; and the set of individuals S_i, which are dominated by the solution i. Individuals with a domination count of 0 are not dominated by any other solution and therefore assigned to the first rank, which is the Pareto front. For these individuals, each member of their respective set S_i has their domination count reduced by 1. Solutions whose domination count thus becomes 0 get assigned to the second rank. This process is repeated until all individuals are assigned a rank [Deb et al., 2002].

- **New crowding distance method, called sparsity:** This method ensures diversity among the individuals and thereby prevents the algorithm from converging prematurely. It sorts the population according to their objective value and assigns a distance to each individual. This distance is equal to the absolute, normalized difference between the objective values of the two solutions that are adjacent to the individual. Since the individuals with the highest and lowest objective values only have one adjacent solution they get assigned an infinite distance. This is done for every objective function. Adding up the individual distance values yields the overall crowding distance. After assigning this measure to each individual, they can be compared and sorted by their proximity to adjacent solutions. Individuals with high crowding distances are located in a sparse area and are preferred over individuals with low crowding distances [Deb et al., 2002].

- **Elitism mechanism:** To ensure that good solutions will not get lost between generations, a set of the best individuals in regards to their fitness and sparsity is defined and carried over to the next iteration of the algorithm. Generally this set constitutes 50 percent of the next generation's population [Deb et al., 2002].

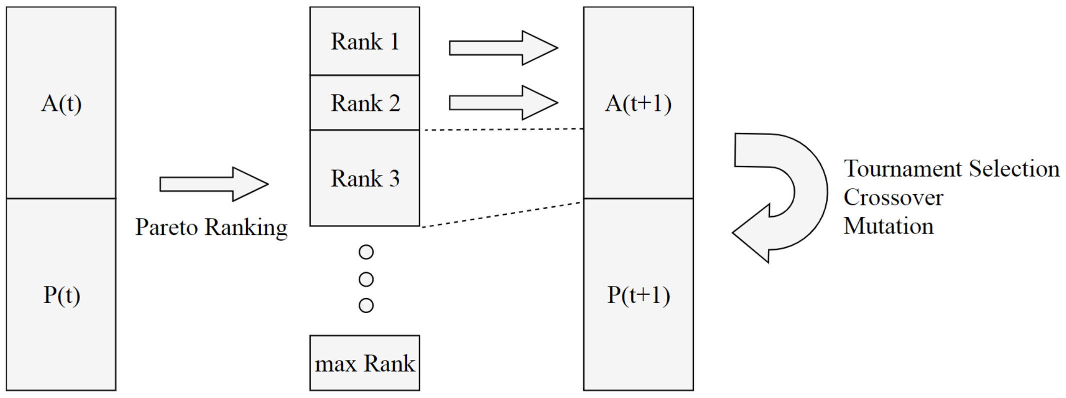


Figure 2.12: The optimization process in one generation of NSGA-II. Adapted from [Yu and Gen, 2010].

The main procedure of NSGA-II starts with initializing a random population P_0 of size n, calculating the fitness values of all individuals and ranking them according to the aforementioned sorting approach. The next step is the creation of an archive A_1 of maximum size m - typically $m = n$ -, which is filled with individuals from P_0. This is done rank-wise in ascending order, i.e. A_1 is populated by individuals of rank 1, followed by individuals of rank 2 etc. until A_1 has reached its size m or cannot accommodate the entire next rank. If A_1's size is less than m, the rank is sorted by its sparsity in descending order and the remaining slots of A_1 are filled by individuals which have the highest sparsity values. The archive's individuals are then used to create the next population P_1. This is achieved by selecting two individuals via tournament selection, recombining and mutating them to create two new offspring individuals before adding them to P_1. This process of selection and breeding is repeated until P_1 reaches size n, which concludes the first generation of NSGA-II. The next iteration starts by adding the archive A_1 to the new population P_1 and proceeds analogous to the first generation with the fitness assessment, ranking, archive creating and breeding. Figure 2.11 illustrates this process. The procedure is repeated until the algorithm either converges or reaches the upper bound of iterations, and returns the first rank of individuals as final solution.

2.6 Transfer Learning

A common application of machine learning methods is the learning of a classifier via labeled training data and evaluating its performance with test data. it is typically assumed that both originate from the same dataset and thus have the same input feature space and data distribution [Han et al., 2011, Pan et al., 2010, Weiss et al., 2016]. Differences in the training and test data distributions can negatively impact the resulting performance of the classifier. If the input data for the current problem changes, a classifier may have to be re-learned completely to offer satisfying results. Such a problem can arise, for example, when the used training data becomes outdated and must be replaced by more recent data. This is oftentimes too expensive or time-consuming, especially if the training data has to be acquired and labeled manually [Pan et al., 2010, Weiss et al., 2016].

A more efficient approach would be to train a classifier with more easily obtainable data from different domains, which still performs well on the actual dataset. This approach is the motivation for transfer learning and it attempts to transfer gained knowledge from other datasets, also called *source task*, and use it in the current problem, also called *target task* [Han et al., 2011, Pan et al., 2010, Weiss et al., 2016]. A basic visualization of transfer learning can be seen in Figure 2.13.

Transfer learning can be defined as a family of algorithms which loosen the assumption of training and test data being in the same input feature space, allowing the domains, tasks and input data distributions to be different [Han et al., 2011, Pan et al., 2010, Yao and Doretto, 2010]. This in turn allows the incorporation of already acquired knowledge from old data sources into the current learning task [Han et al., 2011, Torrey and Shavlik, 2010, Yao and Doretto, 2010]. This approach is analogous to a common human way of learning, where previously obtained knowledge can be used to solve a new problem quicker or better [Pan et al., 2010]. For example, learning to ride a motorcycle can be easier if you already know how to ride a bicycle, since both have common elements which can be incorporated in the learning process. Transfer learning can be used in the same problem categories as standard machine learning algorithms can be applied to, such as clustering, regression or classification [Pan et al., 2010].

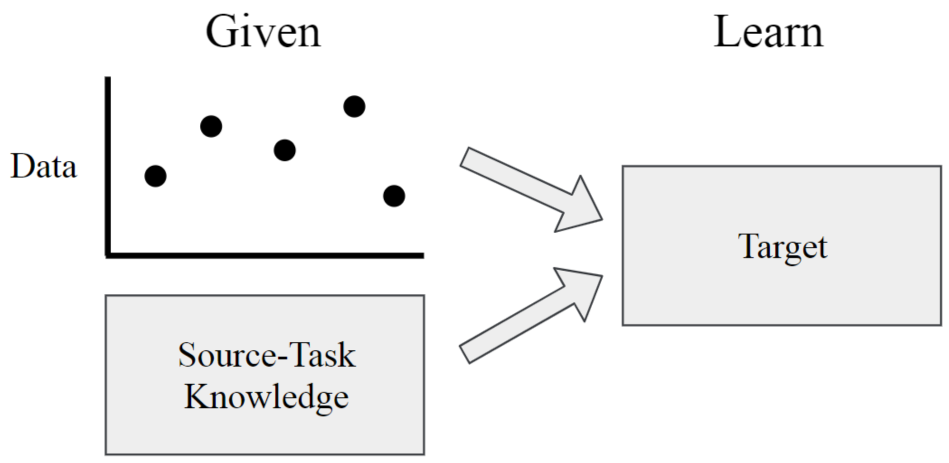


Figure 2.13: A visualization of the concept for transfer learning. Adapted from [Torrey and Shavlik, 2010].

Through the acquired knowledge, the construction of a new system potentially requires fewer training data and time. The initial performance of the classifier as well as its peak performance is also potentially better than without transfer learning [Han et al., 2011, Pan et al., 2010, Torrey and Shavlik, 2010, Yao and Doretto, 2010]. This is especially useful in cases where the amount of training data is small, or the current classifier's training data is outdated and has to be updated by new training samples [Han et al., 2011, Pan et al., 2010]. The degree of improvement in transfer learning is dependent on the relation between the datasets. With a stronger relation between the current and the old dataset, a higher amount of knowledge can be incorporated. A weak relation however might have barely any useful knowledge or even a negative influence, called *negative transfer* [Yao and Doretto, 2010]. Correctly identifying the strength of the relationship between the datasets and therefore determining the transferable information which yields a positive transfer is still a major challenge [Han et al., 2011, Weiss et al., 2016].

2.7 Similarity and Dissimilarity Measures

Similarity measures are functions which quantify the resemblance between two or more sample sets of data. Typically the values of these functions increase with the similarity of the datasets. The inverse of similarity is called dissimilarity, which in turn measures how distinct two data samples are. In this case the function's value typically increases with rising differences between the datasets [Luxburg, 2004]. The following subsections describe four specific tests and measures for (dis)similarity in more detail.

2.7.1 Anderson-Darling Test

The Anderson-Darling test (AD) is a non-parametric goodness-of-fit statistic based on the cumulative probability distribution and is used to calculate the similarity between two data distributions. It was developed in 1952 by T.W. Anderson and D.A. Darling, who wanted an alternative to already existing statistical tests for detecting differences between a sample's distribution and the normal distribution. The test works even with small (>7) datasets and analyzes not only the distributions' means but is also sensible to the differences in shape, scale and variability [Engmann and Cousineau, 2011].

The one-sample Anderson-Darling test analyzes the null-hypothesis that a sample X was drawn from a population which follows a specified, continuous cumulative distribution $F(x)$ [Anderson and Darling, 1954]. It can formally be defined as:

$$AD = -n - \frac{1}{n}\sum_{i=1}^{n}(2i-1)(ln(F(x_i)) + ln(1 - F(x_{n+1-i})) \tag{2.11}$$

where $(x_1, ...x_n)$ is the set of samples of size n in ascending order and $F(x)$ is the underlying distribution to which the sample is compared to [Anderson and Darling, 1954, Scholz and Stephens, 1987]. The null-hypothesis has to be rejected if AD is larger than the critical value AD_α at a given significance level α, i.e. the probability of rejecting the null-hypothesis despite it being true. A typically used value for α is 0.05. The critical values however depend on the distribution and sample size. For specific distributions these values have been published in tabular form [Stephens, 1977].

The two-sample Anderson-Darling test is a generalization of the one-sample case and was proposed by Darling in 1975 [Engmann and Cousineau, 2011]. It analyzes the null-hypothesis that two samples X and Y come from the same distribution, which remains unspecified [Scholz and Stephens, 1987]. It can be formally defined as:

$$AD = \frac{1}{mn} \sum_{i=1}^{n+m} (N_i Z_{n+m-ni})^2 \frac{1}{iZ_{n+m-i}} \tag{2.12}$$

where Z_{n+m} is the combination of the samples X_n and Y_m of size n and m respectively in ascending order, and N_i is the number of elements in X_n that are equal or smaller than the i^{th} observation in Z_{n+m} [Engmann and Cousineau, 2011]. As with the one-sample case the null-hypothesis has to be rejected if AD is larger than the critical value AD_α at a given significance level α.

2.7.2 Kolmogorov-Smirnov Test

The Kolmogorov-Smirnov test (KS) is a non-parametric goodness-of-fit statistic [Massey Jr, 1951], named after Andrey Kolmogorov and Nikolai Smirnov. It is based on cumulative probability distributions and the absolute differences between two functions. The test is applicable for datasets with small sample sizes and is not only sensitive to differences between the distributions' means or medians [Engmann and Cousineau, 2011], but can also detect deviations between their dispersions, skewnesses or shapes along the whole scale [Siegal, 1956].

The one-sample KS test analyzes the null-hypothesis that a sample X was drawn from a population which follows a specified, theoretical distribution $F_0(x)$ [Engmann and Cousineau, 2011]. This is done by computing the maximum distance between $F_0(x)$ and the sample's cumulative frequency distribution $S_n(x)$ under $F_0(x)$. In [Siegal, 1956], it is formally defined as:

$$D = maximum|F_0(x) - S_n(x)| \tag{2.13}$$

where $F_0(x)$ is the specified, theoretical distribution and $S_n(x)$ is the empirical cumulative distribution of sample X, which can be computed with:

$$S_n(x) = \frac{1}{n} \sum_{i=1}^{n} I_{x_i \leq x} \tag{2.14}$$

where $(x_1, ...x_n)$ are the elements of sample X, I is the indicator function, and $S_n(x)$ is the common cumulative distribution function, i.e. the probability for an element x_i of sample X to have a value smaller than x. The null-hypothesis has to be rejected if the probability p associated with D is equal to or smaller than the significance level α. Values for p have been available in tabular form since 1933 and can be found in publications such as [Siegal, 1956].

The two-sample KS-test analyzes the null-hypothesis that two independent samples X and Y were drawn from the same population. This is done by determining the samples' cumulative frequency distributions using equation 2.14 and computing the maximum distance between these two distributions. Formally this can be defined as:

$$D = maximum|S_n(x) - S_n(y)| \tag{2.15}$$

where $S_n(x)$ and $S_n(x)$ are the empirical cumulative distributions of samples X and Y respectively [Siegal, 1956]. As with the one-sample case the null-hypothesis has to be rejected if the probability p associated with D is equal to or smaller than the significance level α.

2.7.3 Pearson's Correlation Coefficient

Pearson's Correlation Coefficient (PCC), also referred to Pearson Product Moment Correlation or simply correlation coefficient, measures the linear relationship between two variables [Hall, 2015]. When applied to samples it can be formally defined as:

$$r = \frac{C_{xy}}{\sqrt{C_{xx}C_{yy}}} = \frac{C_{xy}}{\sigma_x \sigma_y} \tag{2.16}$$

with the covariance C_{xy} defined as:

$$C_{xy} = \frac{1}{N-1}\sum_i (x_i - \bar{x})(y_i - \bar{y}) \tag{2.17}$$

and the standard deviation of the two samples defined as:

$$C_{xx} = \sigma_x^2 = \frac{1}{N-1}\sum_i (x_i - \bar{x})^2 \tag{2.18}$$

$$C_{yy} = \sigma_y^2 = \frac{1}{N-1}\sum_i (y_i - \bar{y})^2 \tag{2.19}$$

with $\bar{x}$ and $\bar{y}$ being the means of the two samples:

$$\bar{x} = \frac{1}{N}\sum_i x_i \tag{2.20}$$

$$\bar{y} = \frac{1}{N}\sum_i y_i \tag{2.21}$$

where $(x_1, ...x_n)$ and $(y_1, ..., y_n)$ are two samples of size n [Hall, 2015].

r can range from -1 to 1, where values close to its boundaries indicate either a strong negative or strong positive correlation between the samples respectively. Values close to 0 however suggest no correlation between the datasets. This is also illustrated in Figure 2.14a - Figure 2.14c. The PCC only depends on the samples' values and their spread, but not on any hypothesized relationships between them [Hall, 2015]. It is unaffected by linear transformation, such as adding a constant to each element of a sample or converting a dataset from one unit to another. However, as illustrated in Figure 2.14d, it is not possible to detect non-linear relationships between datasets [James et al., 2013]. The significance of r depends on the sample size: Small datasets require a value close to -1 or 1 for the linear relationship to be considered significant; larger datasets on the other hand may have smaller values for r without losing their significance [Sedgwick, 2012].

2.7.4 Euclidean Distance

The Euclidean or L2 distance d_E (ED) measures the scalar distance between two points in Euclidean space and can be imagined as a straight line connecting them [Deza and Deza, 2009]. For two vectors, d_E can be described as the dissimilarity or difference between their elements [Gomaa and Fahmy, 2013]. It is formally defined as:

$$d_E = \sqrt{(x_1 - y_1)^2 + ... + (x_n - y_n)^2} \tag{2.22}$$

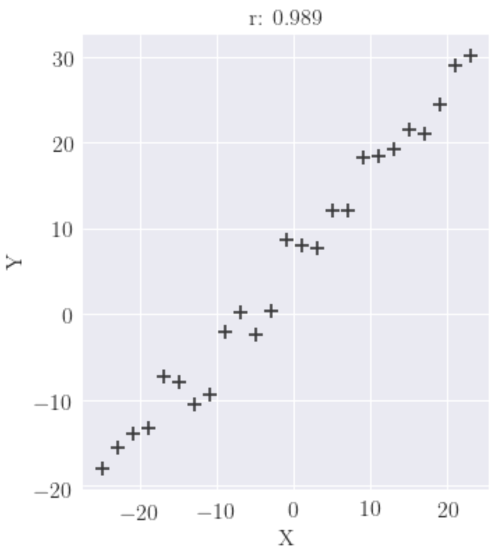
where $x = (x_1, ...x_n)$ and $y = (y_1, ...y_n)$ are two points of dimension n [Deza and Deza, 2009] or two vectors of size n.

2.7.5 Jaccard Index

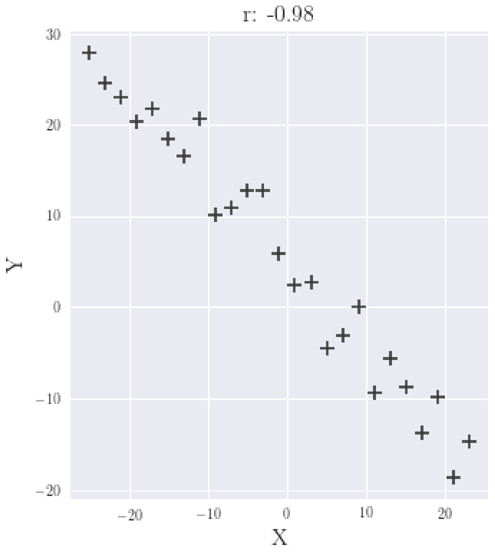
The Jaccard index, also called the Jaccard similarity coefficient, measures the similarity between two sets of data by calculating the intersection's size of both sets, and dividing it by the union's size of both sets. Its formal definition is:

$$J(X, Y) = \frac{|X \cap Y|}{|X \cup Y|} \tag{2.23}$$

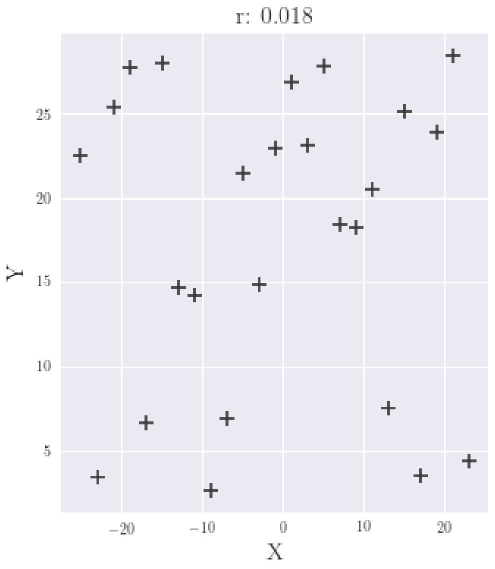
where X and Y are the two datasets and $J(X, Y)$ returns a value between 0 and 1, with 1 indicating that $X = Y$ [Gomaa and Fahmy, 2013]. It can easily be transformed into the Jaccard distance, a dissimilarity measure, by subtracting the Jaccard index from 1.



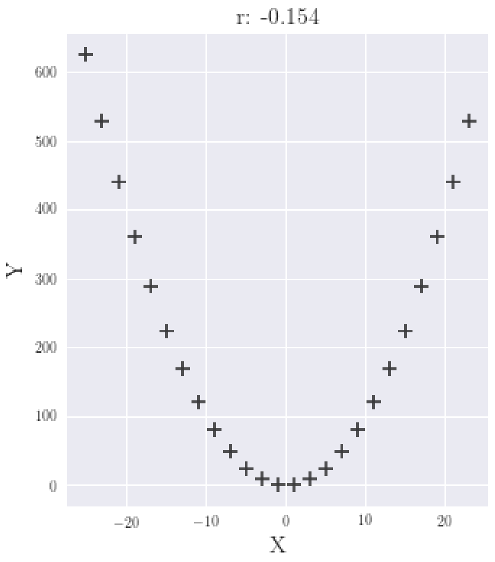
(a) Strong linear relationship between X and Y. $Y \approx X$



(b) Strong negative linear relationship between X and Y. $Y \approx -X$



(c) No relationship between X and Y.



(d) Strong non-linear relationship between X and Y. $Y = X^2$

Figure 2.14: Different correlation between two datasets. Figures (a) to (c) show a correctly detected correlation, while figure (d) shows a non-linear relationship, which stays undetected by the PCC.

Chapter 3

Related Work

There are a number of systems and approaches that cover parts of the functionality the proposed system Loki offers. These can broadly be categorized in two groups analogous to the two already specified functionalities of generating (A)VMs and modifying AVMs.

These systems shall be described in the following subsections.

3.1 Generation of (A)VMs

The system which is used as a basis for this thesis, and therefore naturally the most akin to it, is called Thor [Siegmund et al., 2017]. Its main feature is the rescaling of value distributions for a systems attribute values, such that the density distributions for features, interactions and variants stay consistent. This is achieved through the combined use of KDE and a genetic algorithm. Its advantage in attributing a model lies in the ability to use either one of its various predefined or a user-defined value distribution, which represents realistic software systems better than the commonly used normal distribution. It is also able to generate and incorporate feature interactions of varying degrees.

Another system that generates AVMs is called BeTTy (*Be*nchmarking and *T*es*t*ing on the Automated Anal*y*sis of Feature Models). It is described as a framework for generating and testing conventional and attributed feature models [Segura et al., 2012]. The system uses predefined distributions to generate the attributes' values, which generally does not represent realistic scenarios well. It is also not able to take interactions between the features into consideration, that are proven to be common in real systems and can influence their performance [Apel et al., 2013]. Further features of BeTTy include an automatic generation of test data, which can be used to evaluate VM analysis tools. It is also able to randomly generate VMs.

The tool SPLOT (Software Product Lines Online Tools) is a web-based reasoning and configuration system for SPLs and an example of a program, which can be used to manually generate an AVM [Mendonca et al., 2009]. It offers two major services: An interactive configuration, which means that each of the users decisions is propagated by the system to enforce consistency; and automated reasoning, which automatically updates various statistics of the system, such as depth of the feature tree of the number of features.

Other systems utilize the approach of generating VMs from product descriptions: In [Bécan et al., 2015] an algorithm is presented, which uses product descriptions in tabular form, such as spreadsheets or product comparison matrices, to generate AVMs. Furthermore, domain knowledge is needed to extract features, attributes and domains. Another automated technique which generates a VM based on product descriptions is presented in [Acher et al., 2012]. A functionality to attribute the model is not present. Furthermore, a generic meta model and a dedicated language called VariCell are introduced, which aim to normalize structured data and parametrize data transformation for easier conversion of product descriptions.

Another important part of this thesis is the consideration of feature interactions. Systems such as iGen and VarXplorer attempt to automatically detect feature interactions in configurable software systems [Nguyen et al., 2016, Soares et al., 2018]. Both programs analyze the target software in a test suite during runtime in combination with a set of test cases, iteratively inferring potential interactions from resulting performance values of each configuration. While both infer feature interactions in a software system, they do not offer the functionality of generating AVMs.

3.2 Modification of (A)VMs

For the second use-case, attribute value distributions of an AVM are modified. This can be used as a foundation for evaluating transfer learning approaches. To our knowledge, there are no familiar approaches or systems publicly available. The only related work in this field is the work by [Jamshidi et al., 2017], which is also the foundation for the second functionality of the proposed system. In [Jamshidi et al., 2017], an explorative empirical study was conducted, that aimed to discover how changes in a configurable software system influence the performance model. Four different software systems were exposed to several environmental and configuration changes with varying degrees of severity. The environmental changes include a change in workload, hardware and software version. More specifically, it was investigated how the performance models are related across those changes in order to define when and why approaches of transfer learning are applicable. The study shows that for small changes in the system the performance model is linearly transferable. More severe changes do not express a linear correlation, but exhibit overall similar performance distributions. Even if the performance distributions of the source and target model do not show any relations, it is still possible to infer information about regions in the search space, which yield invalid configurations. This shows that features, which have the most significant impact on the system's performance, remained invariant to changes in the environment regardless of severity.

Chapter 4

Implementation of Use-Case 1: Generating AVMs

This chapter describes the implementation for the functionality of generating AVMs with realistic attribute values and considered interactions. It starts with the problem description an analysis of the requirements. After that, the involved process steps are explained in detail.

An overview over the whole process is depicted in Figure 4.1.

4.1 Problem Description and Requirement Analysis

As described in section 3.1, Thor is a powerful tool that allows its users to create AVMs with feature interactions from an existing VM based on a previously generated AVM of another software system. While Thor provides good results for the stated problems, it suffers from a lack of configurability of the various steps used to create solutions. Additionally, Thor requires various third-party dependencies, one of which is only free for academic use [Apel et al., 2018]. This considerably reduces the accessability of the program to the scientific community, dampening its usefulness.

A better alternative would be an implementation of Thor that uses generally free libraries and is also easy to set up and use. Python offers a wide range of free libraries which cover the base algorithms needed for the process of generating an AVM from a VM. Additionally, libraries in Python are easy to set up and employ by using a package management system such as *pip*[1].

Henceforth, the goal of this task is to re-implement Thor in the Python 3 environment in order to eliminate the mentioned weaknesses of its current

[1]https://pypi.org/project/pip/l

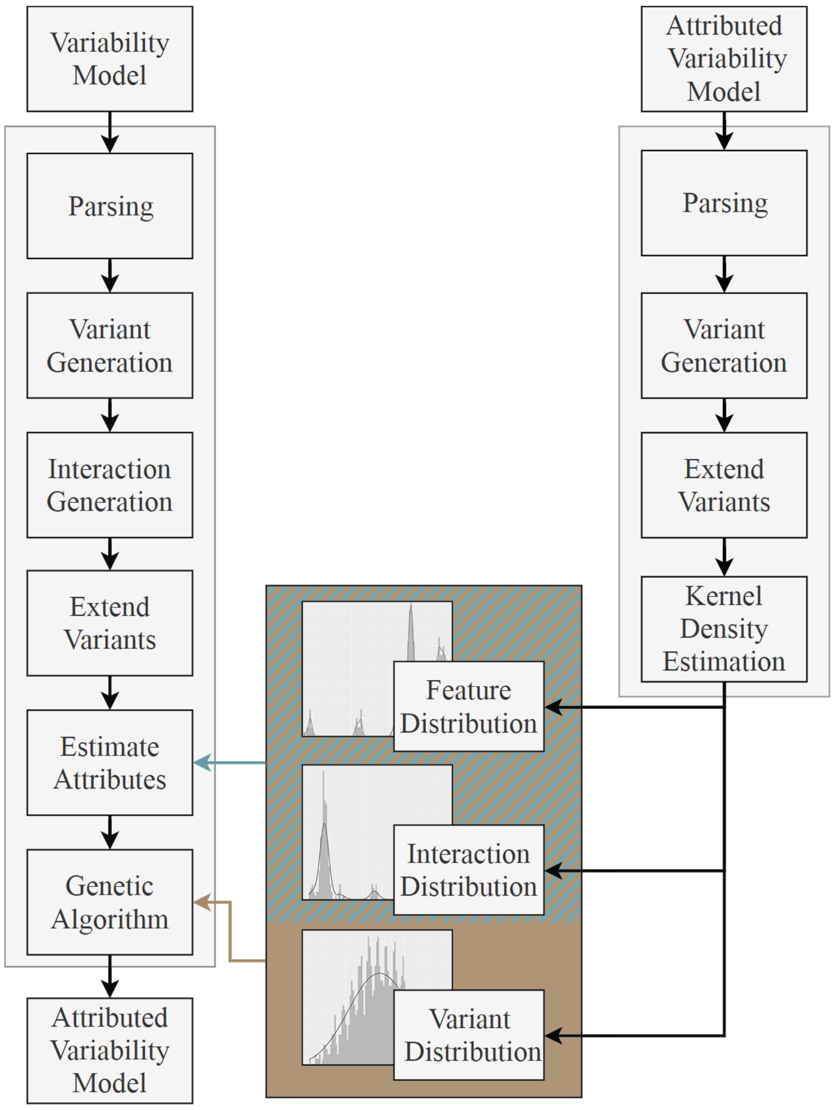


Figure 4.1: Overview over the process of generating AVMs.
It starts with parsing both models, which are each captured by one DIMACS and one or two text files. The DIMACS file and a SAT solver are used to generate valid variants for both models. As the next step, interactions for the VM are generated. This is not required for the AVM, since interactions are already part of the model. Using the generated and provided interactions, the valid variants are extended with information about if the interaction occurs or not. After this, the attribute values of the AVM and KDE is used to rescale the set of attribute values of the AVM's features and interactions, such that they fit the VM's set of features and interactions. Finally, a genetic algorithm is used to tweak the assigned attribute values of the VM in order to increase the similarity between the value distributions of both models. Once the genetic algorithm terminates, it outputs the set of attribute values, which yielded the highest similarity.

implementation. In addition to its current functionalities, the new implementation also shall incorporate new components that increase its configurability.

In detail, Loki must feature the following functional properties:

- Usability with or without constraints and with or without interactions of the models
- Generation of valid configurations for a model
- Generation of interactions for the non-attributed model and the estimation of their attribute values
- Estimation of feature values for a non-attributed VM based on a provided AVM
- Attribution of the VM, with identical (or at least similar) density distributions for the feature, interaction and performance values
- Visualization of the results
- Automatic saving of the results

As already mentioned in this section, Loki must also feature the following non-functional properties:

- Open source format
- Avoidance of cross-language dependencies and instead the incorporation of only open source software for ease of access
- High configurability to adapt algorithm to user requirements and to fulfill users goals
- Support for large models with large data files

4.2 User Settings

The system offers a high configurability to fulfill a wide variety of different user requirements. For ease of use, the user can specify their settings in a configuration file, which is provided with the system.

Two different kinds of configuration files - one for each use-case - are used by the system and can be differentiated by the value for the key *UseCase* under identically named section. In this use case, the configuration file with the key value *AVM-Generation* needs to be used.

In the file, the user can specify the following:

- The files' location for the attributed variability model, in the following sections also called *source model*.
- The files' location for the non-attributed variability model, in the following sections also called *target model*.
- Whether the system should consider the constraints and take the variants' performance value distribution into account
- Whether the system should consider and generate new interactions
- Specification about the new interactions
- The sampling method and the size of the sample
- The population size, maximum number of generations, selection and recombination method for the genetic algorithm
- The similarity measures, which are used in the target function to calculate fitness values
- Whether the system should utilize multi-threading and the number of threads
- The bandwidth for KDE
- Preferences for the ideal solution

4.3 Parsing the Models

The first step in the system's process is the parsing of the attributed and non-attributed variability models. They are each captured by up to three files: One DIMACS file and one text file each for features and for interactions.

It is mandatory to provide the system with the feature files of both models. The usage of the DIMACS and interaction file however is optional and could be omitted. These choices can be specified by the user in the configuration file. In order to present a complete description of Loki's functionality, it is assumed that the system is provided with all three aforementioned files.

The DIMACS file contains the relationships and constraints of a model in the form of a number of boolean expressions, which are written in the conjunctive normal form (CNF). Each clause in the file is represented as a list of either positive or negative integers and is terminated by a 0. The literals inside a clause are joined by OR dependencies, while all clauses are joint by AND dependencies. An example is shown in equation 4.1 below.

$$\begin{array}{r} -1\ 5\ {-7}\ 0 \\ 2\ 0 \\ -3\ 4\ 0 \end{array} \tag{4.1}$$

The text file for the features contains the name of each feature. If the model is an AVM, the file also contains the corresponding attribute values for the features in the form:

$$\begin{aligned} feature01 &: featurevalue01 \\ feature02 &: featurevalue02 \\ feature03 &: featurevalue03 \end{aligned} \tag{4.2}$$

The text file for the interactions contains the names of the features which are involved in an interaction along with its attribute value. This is written as:

$$\begin{aligned} feature01\#feature02 &: interactionvalue01 \\ feature02\#feature03\#feature04 &: interactionvalue02 \\ feature02\#feature04 &: interactionvalue03 \\ feature04\#feature06\#feature01\#feature02 &: interactionvalue04 \end{aligned} \tag{4.3}$$

The system first reads the files and converts each DIMACS file into a nested list, where each clause is defined as one separate list, encapsulated by one main list. This format was chosen since it allows for an easy generation of variants, which will be shown in section 4.4. Then the system parses the text files contents into dictionaries, where each interaction or feature name acts as a key and their corresponding attribute as value.

In Loki's current version, only this kind of model representation is supported.

4.4 Generating Valid Variants

The generation of valid variants has been realized with the Python library *pycosat*[2], which offers a suitable SAT solver for our task. This library was chosen since it is stable and can directly handle the CNF-format described in section 4.3, making a conversion of the parsed constraints unnecessary.

[2]https://pypi.org/project/pycosat/

Pycosat offers two functions: *solve()* and *intersolve()*. While the former is used to compute one single solution, the latter can be used to generate a specified amount or even all possible valid solutions of the model. Similarly to the CNF-form, the resulting solution is returned as a list of integers. In it each absolute number i represents the i^{th} feature, and the sign of the number states if it is selected (+) or unselected (-).

Ideally, every single variant would be computed and the value distribution of the model could be easily reconstructed. However, the computational time and resources required for this approach grows exponentially with the amount of features, making it quickly unfeasible. Because of this, various methods have been implemented that aim to sample the solution space in a manner that still allows for a reliable reconstruction of the value distribution while cutting down the computational time.

The implemented sampling methods are the following:

- **random**: The SAT solver is used to find the specified number of variant without any additional restrictions.
- **feature-wise**: The SAT solver is used to find one variant per feature where this feature is selected.
- **negative feature-wise**: The SAT solver is used to find one variant per feature where this feature is not selected.
- **pair-wise**: The SAT solver is used to find one variant per feature pair, i.e. two consecutive features, where both features are selected.
- **negative pair-wise**: The SAT solver is used to find one variant per feature pair, i.e. two consecutive features, where both features are not selected.

For the method **random** the desired number of variants can be specified. For the other methods, the number of variants depend on the model's number of features and constraints.

Additionally, three permutation methods have been implemented in this process. According to [Henard et al., 2015] more diverse solutions can be computed if parameters such as the constraint order, literal order and the order in which assignments to variables are instantiated, called phase selection, are manipulated. This observation can be explained by the SAT solver working on a tree-structure with backtracking. The mentioned techniques change the order of the branches or the order of the nodes within a branch. This subsequently leads to the SAT solver starting with a different root node and traversing different branches, resulting in a different solution.

Manipulating the phase selection is not supported by *Pycosat*, hence only the modification of the constraint and the literal order has been implemented. This allows for the following methods of permutation in Loki:

- **Without permutation:** *Pycosat* is used to compute one solution. Afterwards the inverse of the solution is added to constraint list, in order to avoid retrieving the same solution again. This process is repeated until the specified number of solutions is computed.
- **Change order of constraints:** The order of constraints is shuffled and *Pycosat* is used to compute one solution. Then the inverse of the solution is added to constraint list. This process is repeated until the specified number of solutions is computed.
- **Change order of constraints and literals:** First the order of constraints is shuffled, followed by shuffling the order of the literals in every constraint. *Pycosat* is used to compute one solution and the inverse of the solution is added to the constraint list. This process is repeated until the specified number of solutions is computed.

After generating the solution according to the specified sampling and permutation method, it is transformed into binary, where negative integers are represented by 0 and positive integers by 1. In this form they are appended to the list of solutions. The resulting set of solutions is again a nested list, where each solution is represented by one list and all solutions are encapsuled in a main list.

Once the specified amount of solutions has been computed, the set of solutions is transformed into a matrix where each solution is one row and each feature is one column. This allows for an easier calculation of the variants' performance values later in the system's process.

4.5 Generating Interactions

The process for generating new interactions starts with the random selection of two or more features. This set of features is formed as a constraint and temporarily added to the actual constraint list. With this temporary constraint, *pycosat* is used to check whether the interaction can occur in at least one valid variant. If no solution can be found, the set is regarded as an impossible interaction. Otherwise, it is checked if the interaction does not occur in every valid variant. This would make it a necessary constraint instead of an interaction and impossible to recognize in a real software system. In both cases the set will be discarded, and the process will be repeated with another set of random features, until one is found which satisfies at least one, but not all valid variants.

These valid sets are then sorted by name and connected analogous to the naming convention in the AVM's interaction file (see section 4.4). If an interaction with the same name is already present, it will instead be discarded to avoid duplicate interactions. As a final step, the interaction is added as a key to a dictionary without a corresponding value, which will later be estimated as described in section 4.6. This process is repeated until the specified amount of interactions for every degree is added.

The generation process of new interactions in the model can be customized in the provided configuration file. It is possible to specify:

- The total number of new interactions
- The ratio of interactions with a certain order, e.g. 25 % interactions with order 2 and 75% with order 3
- The number of threads to use for multiprocessing

4.6 Estimating Attribute Values for Features and Interactions

The estimation-function takes attribute values of the source model's set of features (interactions) and rescales them to the corresponding feature (interaction, respectively) set of the target model, while keeping the value distributions of both models as similar as possible.

In detail, the system utilizes KDE (see section 2.4) provided by the library *scikit-learn*[3] to estimate the probability density function of the source model [Pedregosa et al., 2011]. This function is represented in the system by two lists, which contain the equally spaced samples in the value range, and corresponding densities of each sample.

The computed set of densities is then used as a probability measure to compute the attribute values for features (interactions) of the target model. This means that values with a high density have a higher probability of being assigned to one or more features (interactions) than values with a small density. For each feature (interaction) in the non-attributed feature (interaction) dictionary, a corresponding value from the probability density distribution is drawn and assigned by setting it as the feature's (interaction's) value in the respective dictionary.

[3]https://scikit-learn.org/stable/

In order to achieve good results, the method for KDE uses cross-validation on the source model's datasets to determine an adequate bandwidth for each dataset. Alternatively it is also possible to customize that setting by specifying a preferred bandwidth in the configuration file.

4.7 Computing the Matrix of Extended Valid Variants

Up to this point the solution matrix obtained in section 4.4 only contains information about the selection status of every feature in each solution. In order to calculate the performance of every variant efficiently, the matrix is extended with information about the presence of existing interactions in each solution. An interaction is hereby defined as present if all of its involved features are selected.

To achieve this, each solution (row) in the matrix is iteratively checked for every interaction. If an interaction is present the row is extended by a 1, otherwise the row is extended by a 0.

The result is a matrix in which each solution is stored in a row with information about each selected feature and each occurring interaction. This allows for an efficient calculation of the performance of the solutions, which is used in section 4.9

4.8 Genetic Algorithm

The genetic algorithm forms the centerpiece of this system. Its main purpose is to find assignments of attribute values for features and interactions which keep the value distributions for features, interactions and the performance values of the source and target model as similar as possible.

The method for the genetic algorithm requires the following data:

- the dictionaries for the attributed features and interactions of the source model
- the dictionaries for the non-attributed features and interactions of the target model
- the list of valid variants of the source and target model

For Loki, the NSGA-II algorithm, originally introduced by [Deb et al., 2002], was implemented (see section 2.5.5) and modified to suit the goal of this thesis. In the original NSGA-II-algorithm the exclusively used selection method is the tournament selection. We further implemented two different selection methods, such that the user can choose the one which best suits their goals. The available selection methods are further detailed in section 2.5.2 and are:

- Tournament selection
- Fitness-proportionate selection
- Stochastic universal sampling

The original NSGA-II-algorithm uses a simulated binary crossover. Additionally to this method, Loki offers Line Recombination as an alternative crossover operation, which is further detailed in section 2.5.3. Both the selection and the crossover method can be chosen in the configuration file.

The algorithm begins with creating an initial population by generating individuals with random properties. Each individual is created separately by estimating attribute values for their features and interactions, using the method mentioned in section 4.6. This ensures that the individuals are distinct from each other, and aims to avoid redundant calculation processes. The generated population is stored in a $m * n$ matrix. Each of the m rows represents one individual and each of the n columns is one feature or interaction of the target model, where features are listed first, followed by the interactions. This means that each element represents the attribute value of a feature or interaction of the individual, which is represented by the current row. The number of individuals in the population can be specified in the configuration file.

Some methods employed in the fitness function require the amount of features and interactions of the source model to be equal to the amount of features and interactions of the target model respectively. In order to calculate the fitness score between the models, they are checked for a matching number of features and interactions. If they are not equal, the matrix of the source model will be extended column-wise by estimating the lacking amount of new features and interactions via KDE.

With the requirements for the genetic algorithm fulfilled, the main loop of the algorithm can be executed. The population and, if available, the archive are merged and the fitness values of each individual are determined as described in section 4.9. Using the fitness values as objectives, the individuals of the population are assigned ranks via non-dominated sorting. With this the Pareto front is determined and the population is transformed into a nested list, where each rank is encapsulated in one separate list.

The ranked population is then used to populate the archive. The archive has the same size as the population and acts as a storage for the best candidate solutions which occurred up to this point. It is populated rank-wise analogous to the ranking described above. It is filled until a specific size is reached or until it is not able to accommodate the next rank of individuals completely. In the latter case the rank is sorted according to their sparsity in descending order. This is done to promote diversity between solutions and to avoid a premature termination.

With the updated archive a new population is bred. The breeding process starts by selecting two parents via the specified selection method and recombining them using the specified crossover method. This yields two new offspring individuals, which are then mutated using Gaussian convolution and added to the new population. The breeding process is repeated until the new population has reached the target population size.

The new population along with the updated archive are carried over to the next iteration of the genetic algorithm and the main loop is executed again. This is repeated until the specified number of maximum generations is reached or if the computed Pareto front does not change over the span of three generations. Thereupon, the method returns the list of individuals which are part of the current Pareto front.

4.9 Fitness Calculation

The fitness of a solution is determined by calculating the similarity between the features' and interactions' attribute values as well as the variants' performance values of the solution and the respective values of the source model.

The solutions already contain the attribute values for their features and interactions. The performance values for the sample of variants are calculated via matrix multiplication, where the matrix of valid variants is multiplied with the matrix of the feature and interaction attribute values of the solutions. This results in a vector of performance values, where each row contains the performance of one variant.

Since the performance values depend on the number of features and interactions of the solution, the performance values are scaled using the following equation:

$$perf(x) = perf(x) * \frac{|x|}{|variant_{source}|} \tag{4.4}$$

where x is a variant, $|x|$ is the length of the variant and $|variant_{source}|$ is the length of variants in the source model. The performance values for the variants of the source model are therefore not rescaled.

A similarity is expressed as a normalized value between 0 and 1, where larger values indicate a stronger similarity. The actual similarity between the features, interactions and performances of the models is calculated by at least one of the following implemented similarity measures, which are provided by the library *SciPy*[4] and described in section 2.7:

- Anderson-Darling test
- Kolmogorov-Smirnov test
- Pearson's Correlation Coefficient
- Euclidean distance

The selection of similarity measures can be specified in the configuration file. If more than one similarity measure has been specified, the mean of the calculated results is used as the measure of fitness. The goal of the genetic algorithm is to find a solution that maximizes the fitness value for each of its objectives and therefore find a solution which has a maximized similarity to the source model.

4.10 Solution Evaluation

The method for the genetic algorithm returns the Pareto front of the set of candidate solutions, from which the final solution(s) must be selected.

The system offers three different ways of selecting an adequate solution:

- **Choose all solutions:** No best solution is chosen by the system. Instead the whole Pareto front is selected. This offers the possibility of choosing a solution manually later on, depending on certain criteria known to the user.
- **Choose the overall best solution:** The system compares all candidate solutions and chooses the overall best solution. This is determined by simply adding the fitness values of the different objectives together and then choosing the solution with the highest value. This might yield a solution which possibly does not have the best fitness in one (or more) of the objectives, but is overall the best on average.
- **Custom preference:** In this method the user can specify custom weights for each objective which indicates the relevance of this objectives fitness.

[4]https://docs.scipy.org/doc/scipy/reference/index.html

The sum of the weights has to be equal to 1. The quality of each solution is calculated by the following formula:

$$Quality(x) = \sum_{i=1}^{n} w_i * f_i \tag{4.5}$$

where x is a candidate solution, w_i is the weight for the i^{th} objective and f_i is the solution's fitness for the i^{th} objective. The solution with the highest quality is then selected.

Examples of the selection methods are depicted in Figure 4.2.

The chosen solutions are saved in the same format as the source model, i.e. one file for the attribute values of features and interactions each. Additionally, graphs of the distributions are generated and saved as a png and pdf file to visualize the similarity.

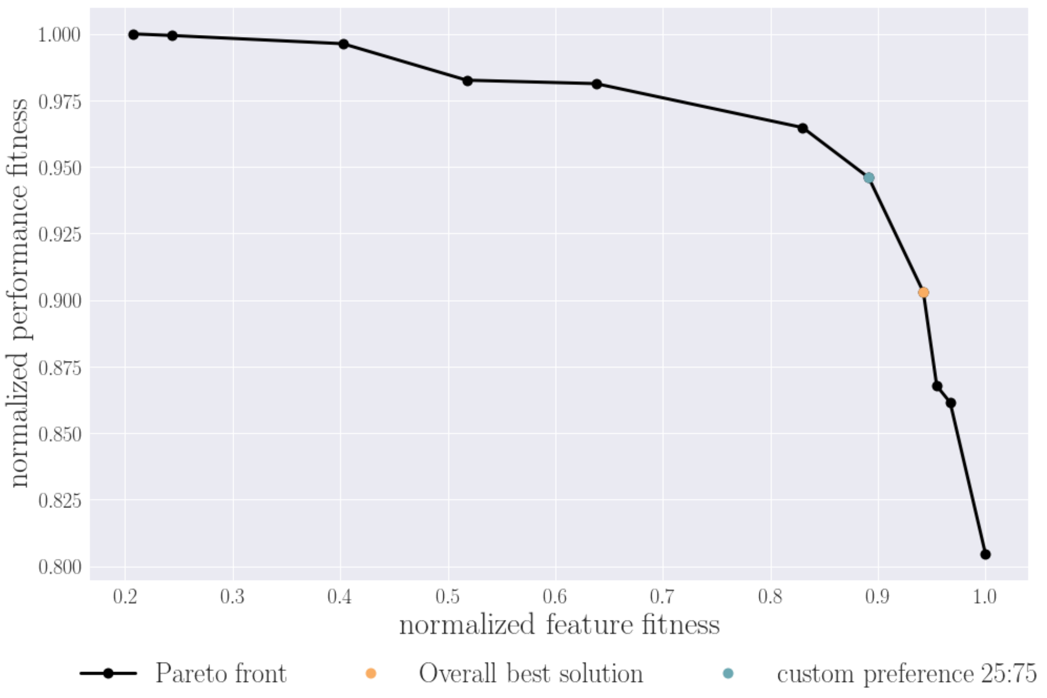


Figure 4.2: Visualization of the solution selection with a two-objective example problem. The colored points show ideal solutions under different user specifications.

Chapter 5

Implementation of Use-Case 2: Modifying AVMs

This chapter describes the implementation of the functionality of modifying AVMs in order to use them for the evaluation of knowledge transfer approaches. It starts with the problem description and an analysis of the requirements. After that, the involved process steps are explained in detail.

An overview over the whole process is depicted in Figure 5.1

5.1 Problem Description and Requirement Analysis

During the development process, software systems are subjected to many changes and updates, which alter the software's properties such as its performance. A model which represented the software before such an update might therefore not be suitable to also represent the software after an update, rendering it invalid. Although the software was not change completely, and thereby both systems share a substantial amount of information and properties, it would be necessary to build a new model, which is an expensive and time-consuming process.

A better alternative is the use of transfer learning approaches, that attempt to extract information from the old model in order to construct the new model more efficiently. For assessing the quality of these approaches, a tool is needed that simulates the changes between the models, which can be used for evaluation. In addition to the goals described in section 4.1, Loki should thus be further extended with the functionality of modifying the attribute values of an AVM, to offer such evaluation possibilities. Such modifications can be realized by different operations like adding noise, negating attribute values or

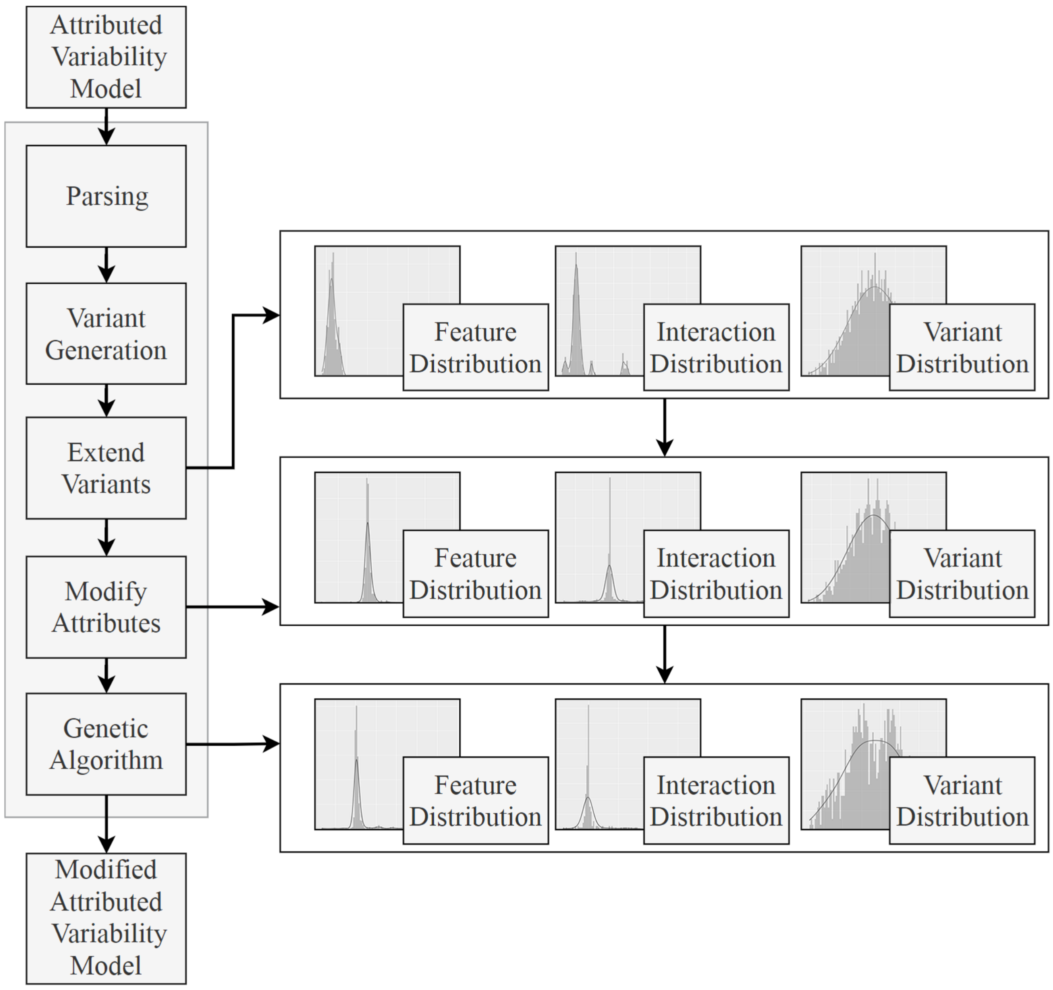


Figure 5.1: Overview over the process of modifying AVMs.
It starts with parsing the model, which is captured by one DIMACS and one or two text files. The DIMACS file and a SAT solver are used to generate valid variants. Using the provided interactions, the valid variants are then extended with information whether the interactions occur or not. After that, the attribute values of features and interactions are modified according to the user's specifications. Finally, a genetic algorithm is used to tweak the modification of the attribute values, such that they conform to the desired scope and severity of change. Once the genetic algorithm terminates, it outputs the set of modified attribute values, which are closest to the desired changes.

by performing linear transformations. The scope of these modifications refers to the amount of changed features and interactions as well as the amount of features and interactions which exhibit a certain modification. They should be specifiable by the user and according to the relation between the two models. Considering, that the different kinds of modifications influence each other, they can act as competing objectives. If the user, for example, wants to change 75% of all interaction values and 75% of all feature values, but only wants to modify 50% of all values by adding noise, the system must find a trade-off between these three objectives. For this reason, an optimization process is needed which finds value assignments for features and interactions that sufficiently satisfy the user's specifications.

Since there is also little to no information about the search space, it is not possible to determine a good starting point for the optimization. This makes the employment of a genetic algorithm favorable: It modifies attribute values in a probabilistic fashion resulting in various random starting points, which allows for the exploration of the search space and for an approximation of a solution that satisfies the user's specifications.

In detail, the extension to Loki must feature the following functional properties:

- Usability with or without constraints and with or without interactions of the models
- Generation of valid configurations for the model
- Transformation of data by adding noise, performing linear transformation and by negating the attribute value
- Determination of features which are (un)selected in every valid variants
- Visualization of the results
- Automatic saving of the results

5.2 User Settings

The system offers a high configurability to fulfill a wide variety of different user requirements. For ease of use, the user can specify their settings in a configuration file, which is provided with the system.

Two different kinds of configuration files - one for each use-case - are used by the system and can be differentiated by the value for the key *UseCase* under identically named section. In this use case, the configuration file with the key value *AVM-Modification* needs to be used.

In the file, the user can specify the following:

- The files' location for the attributed variability model
- Whether the system should consider the constraints and take the variants' value distribution into account
- Whether the system should consider interactions
- The desired scope of changed attribute values for features and interactions
- The desired change operations
- Whether the system should find dead and common features
- The sampling method and the size of the sample
- The population size, maximum number of generations, selection and recombination method for the genetic algorithm
- Whether the system should utilize multi-threading and the number of threads
- The bandwidth for KDE
- Preferences for the ideal solution

5.3 Parsing the Models

The first step in the system's process is the parsing of the AVM, which is captured by up to three files: One DIMACS file and one text file each for features and for interactions.

It is mandatory to provide the system with the feature file of the model. The usage of the DIMACS and interaction file, however, is optional and could be omitted. These choices can be specified by the user in the configuration file. In order to present a complete description of Loki's functionality, it is assumed that the system is provided with all three aforementioned files.

As already detailed in section 4.3, the DIMACS file contains the relationships and constraints of the model written in the conjunctive normal form (CNF) and the text files for the features and interactions each contain the name of the feature and interaction respectively as well as the corresponding attribute values. Examples for these files can also be found in the section 4.3.

The program first reads the files and converts the DIMACS file into a nested list, where each clause is defined as one separate list, encapsulated by one main list. This format was chosen, since it allows for an easy generation of variants, which will be shown in section 5.4. Then the program parses the text files contents into dictionaries, where each interaction or feature name acts as a key and their corresponding attribute as value.

In its current version, the system only supports this kind of model representation.

5.4 Generating Valid Variants

The generation of valid variants has been realized with the Python library *pycosat* the same way as described in section 4.4 .

In order to reliably reconstruct the value distribution of the model's performance in a feasible amount of computational time, the following sampling methods were implemented:

- **random**
- **feature-wise**
- **negative feature-wise**
- **pair-wise**:
- **negative pair-wise**

For the method **random** the desired number of variants can be specified. For the other methods, the number of variants depend on the model's number of features and constraints.

Additionally, three permutation methods have been implemented according to the paper by Henard et al., who demonstrated that more diverse solutions can be computed by manipulating parameters such as the constrain order, literal order and phase selection. The latter is not supported by *Pycosat*, hence only the modification of the constraint order and the literal has been implemented. This allows for the following methods of permutation in Loki:

- **without permutation**
- **Change order of constraints**
- **Change order of constraints and literals**

For a detailed explanation of the sampling and permutation methods see section 4.4

After generating the solution according to the specified sampling and permutation method it is transformed into binary, where negative integers are represented by 0 and positive integers by 1. In this form they are appended to the list of solutions. The resulting set of solutions is again a nested list, where each solution is represented by one list and all solutions are encapsuled in a main list.

Once the specified amount of solutions has been computed, the set of solutions is transformed into a matrix where each solution is one row and each feature is one column. This allows for a an easier calculation of the variants' performance values later in the system's process.

5.5 Computing the Matrix of Extended Valid Variants

Up to this point, the solution matrix obtained in section 5.4 only contains information about the selection status of every feature in each solution. In order to calculate the performance of every variant efficiently, the matrix is extended with information about the presence of existing interactions in each solution. An interaction is hereby defined as present if all of its involved features are selected.

To achieve this, each solution (row) in the matrix is iteratively checked for every interaction, extending the row by a 1 if the interaction is present, or a 0 if it is not.

The result is a matrix in which each solution is stored in a row with information about each selected feature and each occurring interaction.

5.6 Modifying the Model

The method for modifying the model takes the features' and interactions' attribute values and transforms them according to the user's specifications in the configuration file.

Three options for defining the features' and interactions' scope of change, respectively, are available:

- **none:** Allows no changes on the features (interactions)
- **all:** Allows changes on all features (interactions)

- **most-influential:** Allows changes on relevant features (interactions), i.e. features (interactions) with high attribute values. A feature (interaction) is considered relevant, if its attribute value is higher then a certain threshold, which can be specified by the user.

Depending on the user's choice, either none, all, or the subset of most relevant features (interactions) are subjected to the modification.

The modification operation(s) can also be specified by the user. The following modifications are available:

- **Noise:** A normal distribution is used to generate noise, which is added to the data. The user can specify the probability of adding the noise, as well as the mean and standard deviation for the distribution.
- **Linear Transformation:** A linear transformation is applied to the data. The user can specify the probability of applying the transformation, as well as the operation and operand. Four different operations are available: addition, subtraction, division and multiplication. For the operand the user can choose a value which is > 0. This value expresses the scope of change in percent, which means that the modification is dependent on the attribute value itself. For example: Using the operation addition and the operand 1 will change the attribute value 5 into a 10 and the attribute value 20 into a 40.
- **Negation:** A method to change the values sign of the attribute values, i.e. changing it from positive to negative and vice versa. The user can specify the probability of performing this change.

It is recommended to select only a single operation, but the system also supports the execution of multiple modification operations. If the user selects more than one operation, they are executed in the same order in which they were specified in the configuration file.

The modification method starts by selecting the data which the user specified to subject to the modification process. Following that, the selected modification operations are executed. Each of these operations iterate of the set of attribute values and modify each value with their respective probability.

After performing the modification, the method returns the dictionaries for both features and interactions, which contain the newly modified attribute values.

5.7 Genetic Algorithm

The genetic algorithm forms the centerpiece of this system. Its main purpose is to find assignments of attribute values for features and interactions that fulfill the modification-related specifications of the user.

The method for the genetic algorithm requires the following data:

- the dictionaries for the attributed features and interactions of the original model
- the list of valid variants for the original model

For Loki, the NSGA-II algorithm, originally introduced by [Deb et al., 2002], was implemented (see section 2.5.5) and modified to suit the goal of this thesis. In the original NSGA-II-algorithm the exclusively used selection method is the tournament selection. We further implemented two different selection methods, such that the user can choose the one, which best suit their goals. The available selection methods are further detailed in section 2.5.2 and are:

- Tournament selection
- Fitness-proportionate selection
- Stochastic universal sampling

The original NSGA-II-algorithm also used a simulated binary crossover, which was replaced by the following three methods:

- One-Point-Crossover
- Two-Point-Crossover
- Universal Crossover

All of them are explained in more detail in section 2.5.3. Both the selection and the crossover method can be chosen in the configuration file.

The algorithm begins by creating an initial population by generating modified versions of the AVM. Each individual is created separately by modifying the attribute values for features and interactions, using the modification method explained in section 5.6. This ensures that the individuals are distinct from each other, and aims to avoid redundant calculation processes. The generated population is stored in a $m * n$ matrix. Each of the m rows is one individual and each of the n columns is one feature or interaction of the AVM, where features are listed first, followed by the interactions. This means that each element represents the attribute value of a feature or interaction of the individual, which is represented by the current row. The number of individuals in the population can be specified in the configuration file.

With the initial population generated, the main loop of the genetic algorithm can be executed. The population and, if available, the archive are merged and the fitness values of each individual are determined as described in section 5.8. Using the fitness values as objectives, the individuals of the population are assigned ranks via non-dominated sorting. With this, the Pareto front is determined and the population is transformed into a nested list, where each rank is encapsulated in a separate list.

The ranked population is then used to populate the archive. The archive has the same size as the population and acts as a storage of the best candidate solutions which occurred up to this point. It is populated rank-wise, analogous to the ranking described above.It is filled until a specific size is reached or until it is not able to accommodate the next rank of individuals completely. In the latter case, the rank is sorted according to its sparsity in descending order. This is done as to promote diversity between solutions and to avoid a premature termination.

With the updated archive a new population is bred. The breeding process starts by selecting two parents via the specified selection algorithm and recombining them with the selected crossover method. This yields two new offspring individuals, which are added to the new population. The breeding process is repeated until the new population has reached half of the target population size. The other half is populated by newly generated individuals. This is done to introduce novelty, since the recombination of individuals alone might not allow to search in every region of the solution space.

The new population, along with the updated archive, are carried over to the next iteration of the genetic algorithm and the main loop is executed again. This is repeated until the specified number of maximum generations is reached or if the computed Pareto front does not change over the span of three generations. Subsequently the method returns the list of individuals, which are part of the current Pareto front.

5.8 Fitness Calculation

The fitness of a solution is determined by calculating its degree of modification relative to the original model and comparing it with the user's modification related specifications. These are:

- **The percentage of changed features.** This takes into account if the user specified to change all features in general or only the most relevant ones.

- **The percentage of changed interactions.** This takes into account if the user specified to change all interactions in general or only the most relevant ones.
- **The percentage of features and interactions, which exhibit a specific modification:** This depends on the modifications specified by the user. Each modification is analyzed separately.

Depending on the user's specifications, the fitness is calculated based on all feature (interaction) attribute values or on the set of most relevant ones. For example, if the user specified to only change the most relevant feature attribute values, the fitness function will only compare the most relevant feature attribute values of the original model and the solution.

The fitness value represents the satisfaction of the user's specifications and is expressed as a normalized value between 0 and 1, where larger values indicate a stronger satisfaction. Different methods are utilized to determine the satisfaction of the aforementioned specifications:

- **For changed features and interactions:** The method compares the set of attribute values of both the original model and the solution and determines the amount of changed elements. After that, it computes the ideal amount of changed elements according to the user's specifications and compares both amounts.
- **For specific modifications:** The method compares the set of attribute values of both the original model and the solution and determines the amount of changed elements, which can be accounted to this modification. After that, it computes the ideal amount of changed elements according to the user's specifications and compares both amounts.

As already mentioned, it is recommended to only select one kind of modification, since the selection of multiple ones can result in elements being modified more than once. The system, however, will not detect both changes but instead interprets it as the result of a single modification, which in turn will distort the fitness calculation.

The goal for NGSA-II is maximize the fitness values and therefore maximize the similarity between the actual and the ideal degree of modifications.

5.9 Finding common and dead features

As noted by [Jamshidi et al., 2017], information about invalid regions in the solution space can be transferred across environments, even after such severe modifications, that the source and target model do not show any relations. Hence a method was implemented, which examines the solution space in order to find features, which are either selected or unselected in each variant. Such features are referred to as *common features* and *dead features*, respectively.

The method takes the parsed list of constraints and checks for every feature whether there is at least one variant where it is selected and at least one variant where it is unselected, defining it as neither common nor dead. If such variants cannot be found for a feature, its index is saved in a list. The sign of the index indicates whether it is a dead (negative index) or a common (positive index) feature. Once the method has checked every feature, it returns the list of indices for the common and dead features.

5.10 Solution Evaluation

The method for the genetic algorithm returns the Pareto front of the set of candidate solutions, from which the final solution(s) must be selected.

The system offers three different ways of selecting an adequate solution:

- **Choose all solutions**
- **Choose the overall best solution**
- **Custom preference**

All selection methods are explained in more detail in section 4.10 and an example is depicted in Figure 4.2.

The chosen solutions are saved in the same format as the original model, i.e. one file for the attribute values of features and interactions each. If specified by the user, the common and dead features of the solution space are also saved in a text file. Additionally, graphs of the distributions are generated and saved as a png and pdf file to visualize the performed modifications.

Chapter 6

Evaluation

If not specified otherwise, all experiments were performed and evaluated on a computer with 8 GB of DDR3-1600MHz RAM, the Intel Core i7-4700MQ 2.4GHz processor, and the Windows 7 64bit operating system.

6.1 Performance Evaluation

In order to test how Loki's performance behaves with increasing amounts of valid variants, both functionalities have been evaluated in this section. The runtime of both functionalities was measured for a differing amount of variants, starting at 50 variants up to 10,000 variants with varying step sizes. Each test was repeated 10 times to rule out measurement bias and random effects.

6.1.1 AVM Generation

For the AMV generation the toybox variability model with 544 features and 109 interactions was used. The interactions were divided into 80% of degree 2, 10% of degree 3 and 10% of degree 4 in order to enable comparison to the results of [Siegmund et al., 2017]. The genetic algorithm was iterated 100 times and used a population size of 50. For the selection and crossover methods the tournament selection and simulated binary crossover were chosen, respectively. The fitness was calculated by employing PCC and ED. The variants were computed without permutation and the random sampling method. The resulting computation times are shown in Figure 6.1.

The results show that with an increasing number of variants an increasing percentage of the computation time required is caused by matrix multiplications, with 43% at 50 variants and a total of 87% at 10000 variants. Another major part of the computation time is caused by the fitness calculation, with 10.2% of the computation time at 50 variants and 11.8% at 10,000 variants.

Both these operations are repeated in every iteration, for every individual of the population and every variant. For an increased amount of iterations, these operations would therefore make up a larger portion of the computation time. The value estimation for the the first population in the algorithm caused 4.4% of the computation time for 50 variants and 1% of the computation time for 10,000 variants. Percentage-wise, the computation time for this step is proportional to the chosen population size and inversely proportional to the amount of maximum iterations, ranging from negligible to a considerable part of the computation time.

Operations which are performed only once, such as the interaction generation, saving of results, matrix creation and file parsing make up less than 1% of the computation time and are therefore small enough to be negligible.

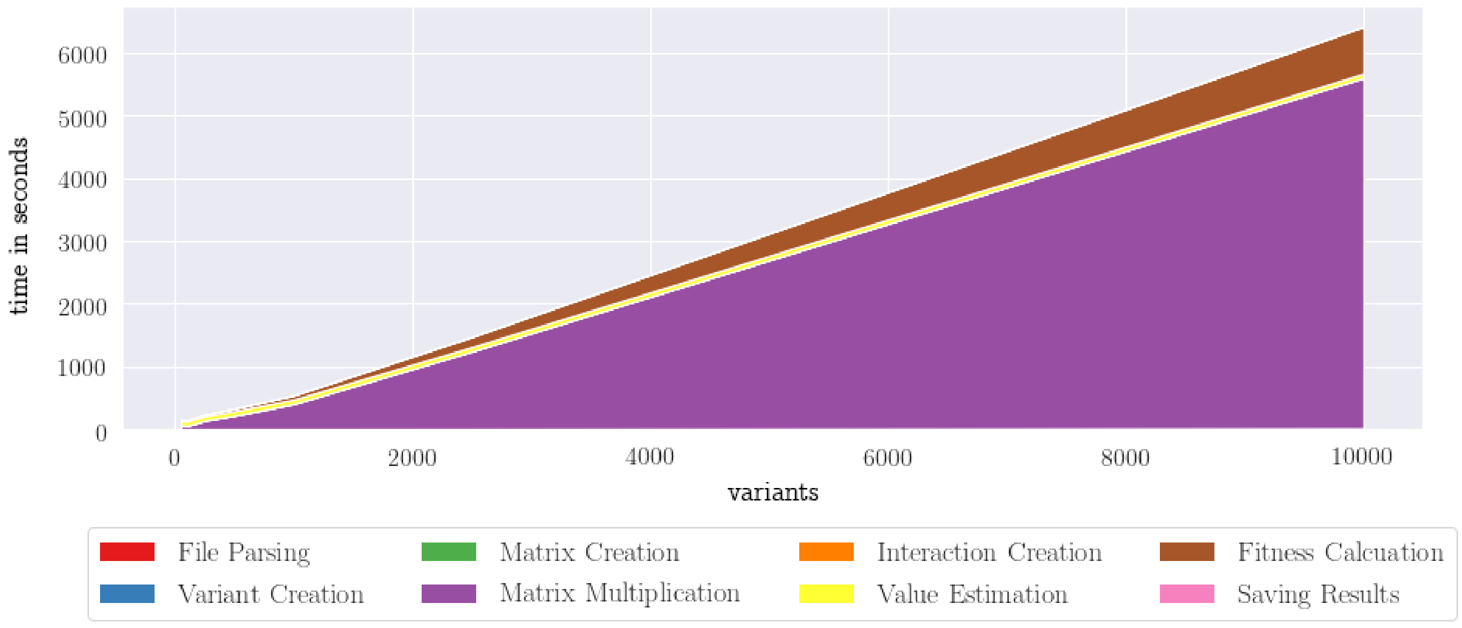


Figure 6.1: Visualization of the computation time fractions with an increasing variant set size

6.1.2 AVM Modification

For the AMV modification the toybox variability model with 544 features and 218 interactions was used. The interactions were divided into 70% of degree 2, 20% of degree 3 and 10% of degree 4. The genetic algorithm used a population size of 50. For the selection and crossover methods the tournament selection and universal crossover were chosen, respectively. The fitness was calculated as described in section 5.8. The variants were computed without permutation and the random sampling method. The resulting computation times are shown in figure Figure 6.2.

The results show that major parts of the computation time are caused by the model modification and the fitness calculation operations. The percentage of computation time caused by both can be considered independent of the amount of iterations and is growing linearly. For 100 iterations for example, the model modification takes 30% and the fitness calculation takes 62% of the computation time while for 5000 iteration the model modification takes 37% and the fitness calculation 63% of the computation time. The computation time needed to find common and dead features is invariant to the amount of iterations, it accounts for 4% of the total computation time for 100 iterations and less than 1% for 5000 iterations. Its computation time is only dependent on the amount of constraints and features in the model.

Operations which are performed only once, such as the variant generation, saving of results, matrix creation and file parsing make up less than 1% of the computation time and are therefore small enough to be negligible.

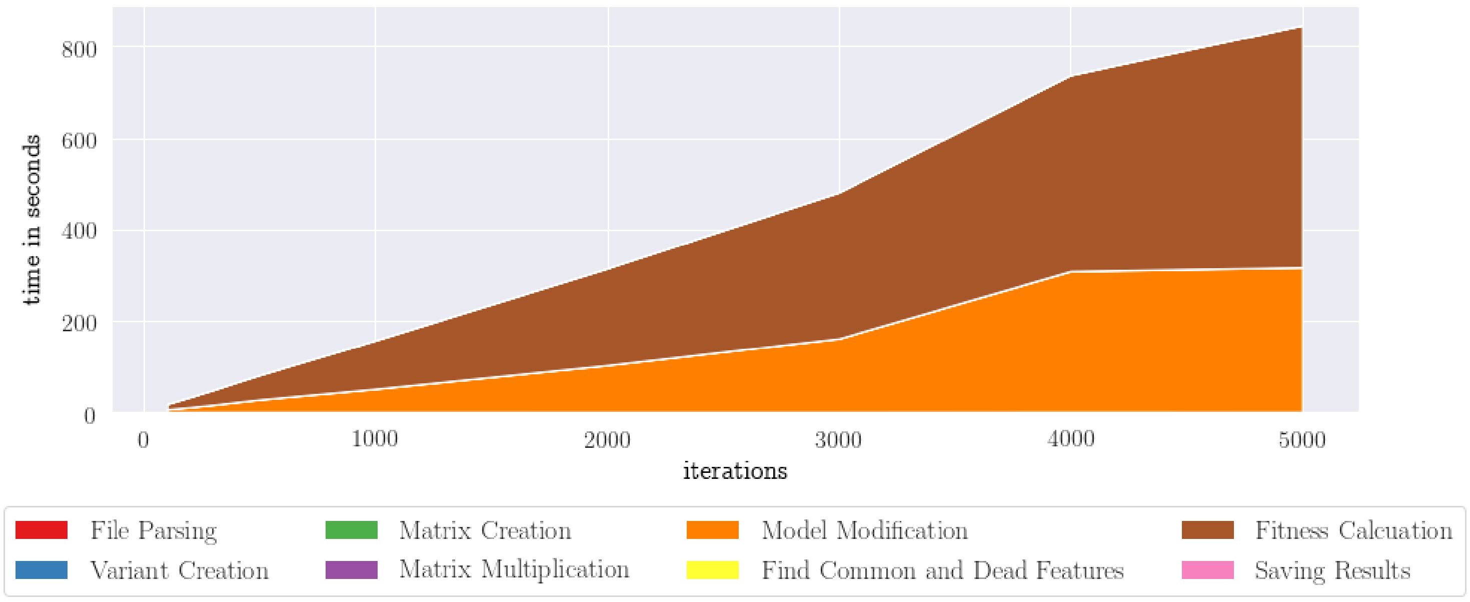


Figure 6.2: Visualization of the computation time fractions with an increasing amount of iterations for the genetic algorithm.

6.2 Variant Sampling Evaluation

Different sampling and permutation methods were implemented for generating valid variants, with the latter according to [Henard et al., 2015] as described in section 4.4. This was done to sample the solution space and offer a reliable estimation of the model's performance value distribution which is close to the true distribution.

Two experiments were conducted to investigate how the different sampling and permutation methods influence the diversity within the generated sets of valid variants and the similarity between the estimated and the true performance value distributions.

In both experiments an example model with 100 features and 403 clauses published by the University of British Columbia[1] was used.

6.2.1 Diversity Experiment

According to [Henard et al., 2015], generated variants with a SAT solver yield a higher diversity if the clauses and literals of a model's constraints are permutated after each iteration of variant generation. Owing to its promising results, this approach has been implemented into Loki.

In order to verify an improvement in diversity and to analyze the influence of the various incorporated sampling methods, the Jaccard distance has been measured as a difference between the selected and unselected features of two variants. This metric was calculated for all 15 combinations of the five sampling and three permutation methods with two different implementations: By comparing consecutive variants and by comparing all variants with each other. This can be expressed by the following two formulas:

$$J_{Cons}(V_i, V_j) = 1 - \frac{|V_i \cap V_j|}{|V_i \cup V_j|} \tag{6.1}$$

$$J_{Set} = \frac{1}{\binom{2}{n}} \sum_{i=1}^{n} \sum_{j=i+1}^{n} J_{Cons}(V_i, V_j) \tag{6.2}$$

where V_i andV_j are two variants of the same variant set. While J_{Set} captures the overall mean diversity of variant sets, J_{Cons} is used to capture the diversity between variants that were generated in direct succession by the SAT solver. The experiment was repeated ten times with different seeds for the random number generator in order to to account for measurement bias and random effects. Table 6.1 shows the resulting distances and Table 6.4 illustrates the corresponding computation times for all combinations.

[1]https://www.cs.ubc.ca/ hoos/SATLIB/benchm.html

Table 6.1: Diversity of variants under different sampling and permutation methods

		No Permutation		Permutated Clauses		Permutated Clauses and Literals	
Sampling Method	Number of Variants	Consecutive Variants	Set of Variants	Consecutive Variants	Set of Variants	Consecutive Variants	Set of Variants
random	1000	0.012502	0.057345	0.148536	0.160130	0.149156	0.158675
f-wise	94	0.259149	0.248051	0.264681	0.250906	0.262766	0.252246
nf-wise	96	0.234583	0.239789	0.246000	0.248710	0.249313	0.248974
p-wise	82	0.218292	0.303863	0.220048	0.299128	0.220195	0.297893
np-wise	84	0.204761	0.269153	0.196929	0.273468	0.195904	0.272501

f-wise: feature-wise; nf-wise: negative feature-wise; p-wise: pair-wise; np-wise: negative pair-wise

The data shows that permutation increases the diversity for random, feature-wise and negative feature-wise sampling compared to the variant sets generated without permutation. Only pair-wise and negative pair-wise sampling showed a decrease in diversity for J_{Set} and J_{Cons} respectively. The highest difference could be observed in the random sampling method, where the diversity increased by 1,091.34% for J_{Cons} and 176.7% for J_{Set} with permutation of clauses and literals. This is consistent with the results by [Henard et al., 2015], who reported a minimum increase of 2,768%.

In [Henard et al., 2015] only the difference in diversity between no permutation and permutation of phase selection, clauses and literals were analyzed. In Loki the influence of only permutating clauses and permutating clauses and literals was considered. The data shows that for J_{Cons} two of the five sampling methods had better values for only permutating clauses than permutating clauses and literals. For J_{Set} the result was consistent for three of five sampling methods. Only the negative feature-wise sampling showed an increase for both J_{Cons} and $Jset$. The highest value for J_{Cons} without permutation was achieved with feature-wise sampling, while the best diversity overall could be obtained by additionally permutating the clauses. The best value for J_{Set} was achieved with pair-wise sampling and permutation of clauses, which has also the best diversity for J_{Set} overall.

6.2.2 Similarity Experiment

The goal of the first experiment was to determine Loki's efficiency and accuracy of estimating a model's true performance value density distribution (PVDD). For this, the aforementioned example model was attributed with artificially generated attribute values and all 148,844 valid variants were computed in order to derive the true PVDD, which is illustrated in Figure 6.3.

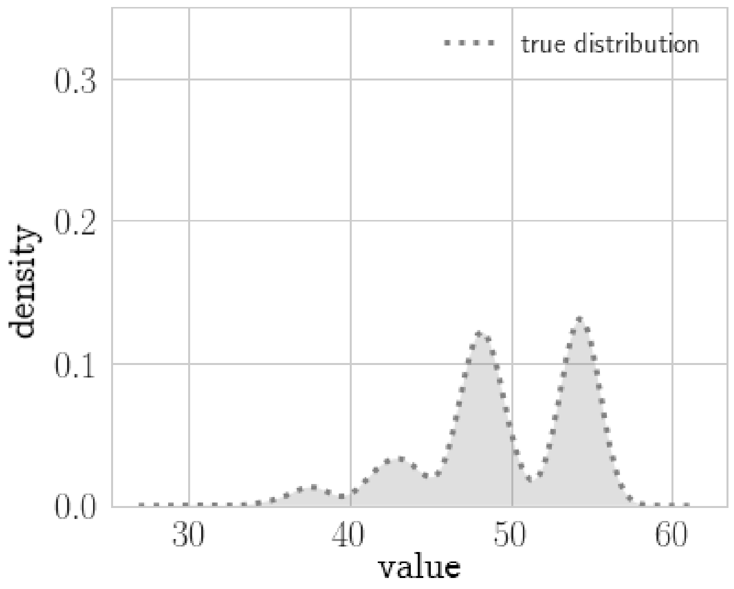
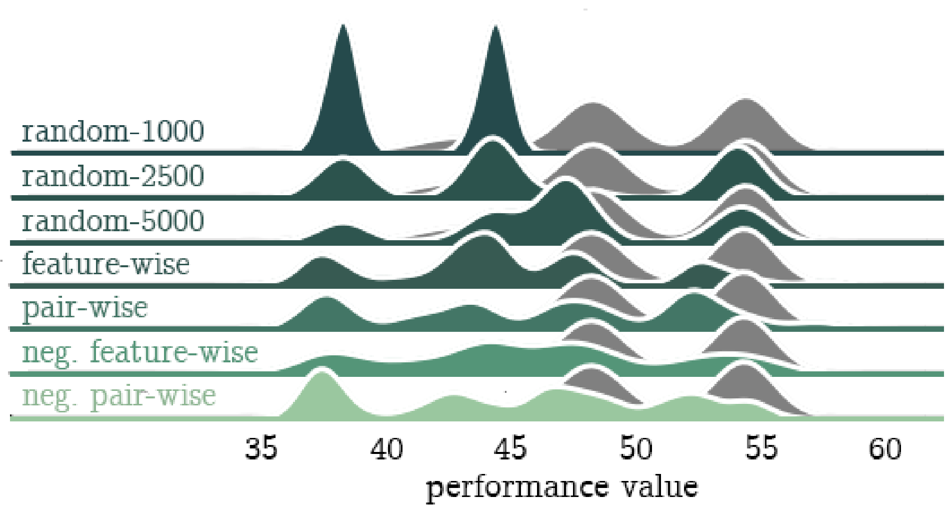


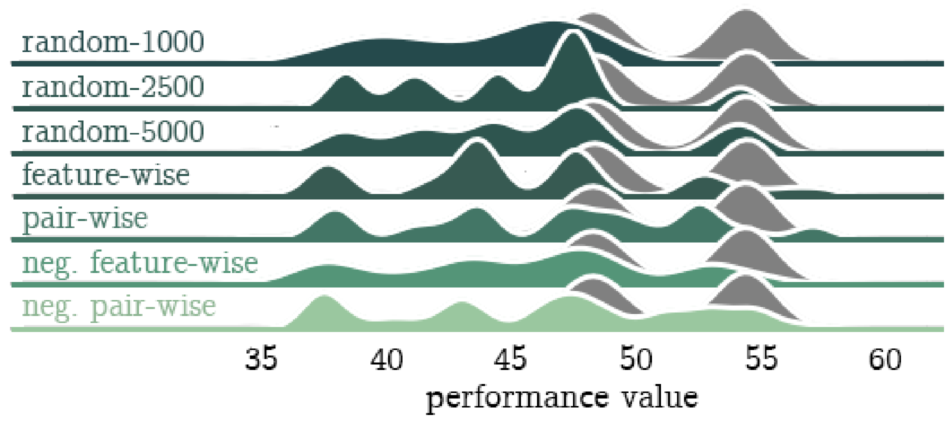
Figure 6.3: Visualization of the true performance value density distribution of the example model.

Loki was then used to determine the PVDD of the model. Three permutation methods and five sampling methods were available and used in every possible combination to generate sets of valid variants, resulting in a total of 15 combinations. Additionally, the random-method allows to define the amount of variants to be generated. In the experiment the random-method was used with three different amounts of variants, resulting in a total of 21 unique combinations. From the resulting variant sets performance values were calculated and the corresponding PVDD was determined via KDE. All of these distributions and the true distribution are shown in Figure 6.4.

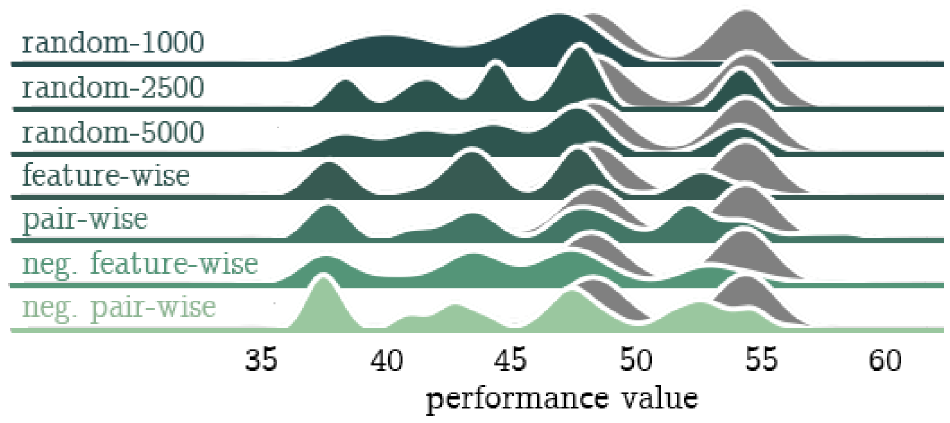
To estimate how close each approximated PVDD relates to the true PVDD, four similarity measures were used: ED, KS, PCC and AD, some of which require sample sets of the same size. Therefore each PVDD was evenly sampled in the range of 25 and 65 with 5000 samples. This range was chosen since no PVDD featured values outside of these bounds. While collecting the data it became clear that SciPy's implementation of the AD does not output a result if both datasets are too dissimilar. This resulted in only two collected measurements and therefore the Anderson-Darling test was omitted as a similarity measure.



(a) PVDDs without permutation



(b) PVDDs with permutation of clauses



(c) PVDDs with permutation of clauses and literals

Figure 6.4: Estimated PVDDs (green curves) compared to the true PVDD (grey curve) of the test model.

The experiment showed that the permutation of clauses yields the highest similarity for all similarity measures. Within the permutation of clauses, negative pair-wise sampling yielded the lowest ED (2.7468), random sampling with 5,000 variants yielded the highest PCC statistic (0.6678) while random sampling with 1,000 variants resulted in the lowest KS statistic (0.1086). Considering all three similarity measures simultaneously, the overall best similarity was obtained via permutation of clauses and negative feature-wise sampling. When considering the other two permutation methods separately, negative feature-wise sampling also outperforms all other sampling methods. The worst similarity according to all three similarity measures however was obtained via random sampling with 1000 variants and without permutation.

All results for this experiment are shown in Table 6.2

The experiment was repeated ten times with different seeds for the random number generator in order to avoid statistically irrelevant results. If no permutation was used, the variant sets were identical in every iteration, and consequently the PVDDs and similarity values were identical. Using either permutation of clauses or permutation of clauses and literals resulted in different variants, and slight deviations in PVDDs and similarity values. This is shown in Figure 6.5 and Figure 6.6.

The mean and median values for the similarity measures of all ten experiments show that the worst results were achieved with the random sampling with 1000 variants and no permutation. This is consistent with the result from the first run. The best results measured in ED were obtained with negative feature-wise sampling and without permutation, with the mean and median value at 2.803. Random sampling with 1000 variants and permutated clauses resulted in the best KS, with 0.1181 mean value and 0.1074 median value, while the best PCC was obtained with random sampling with 5000 variants without permutation. The overall best combination was negative feature-wise sampling without permutation.

Another important aspect to consider is the computational time required for computing the sets of variants. For this aspect each mentioned combination was again performed ten times. The results are shown in Table 6.4.

The table shows that random sampling with 5,000 variants with permutation of clauses and literals required the longest computational time, namely 2,086 seconds. The lowest runtime was negative pair-wise sampling without permutation at 0.17 seconds. Generally, all sampling methods apart from random sampling required less than two seconds for all permutation methods.

In theory using random sampling with more than 5,000 variants could yield better results considering the achieved similarity, but this would be accompanied by a tremendous increase in computational time, which would very quickly dampen its usability. Considering computer systems with lower performance,

Table 6.2: Similarity between estimated PVDDs and the true PVDD for one test case.

Permutation Method	Sampling Method	ED statistic	PCC statistic	PCC p-value	KS statistic	KS p-value
No Permutation	random-1000	8.1316	-0.0664	$2.95e^{-11}$	0.6446	0.0
	random-2500	4.0004	0.4391	0.0	0.2268	$1.38e^{-224}$
	random-5000	2.8623	0.6560	0.0	0.3939	0.0
	f-wise	3.8191	0.3898	0.0	0.1800	$1.27e^{-141}$
	nf-wise	2.8203	0.6421	0.0	0.1888	$9.17e^{-156}$
	p-wise	3.1430	0.5494	0.0	0.1554	$1.15e^{-105}$
	np-wise	3.3351	0.5099	0.0	0.2779	0.0
Permutated Clauses	random-1000	3.9227	0.3725	0.0	0.1086	$8.02e^{-52}$
	random-2500	3.7456	0.5294	0.0	0.2694	$1.05e^{-316}$
	random-5000	3.0336	0.6678	0.0	0.3017	0.0
	f-wise	3.6679	0.4568	0.0	0.1863	$1.12e^{-151}$
	nf-wise	2.7915	0.6508	0.0	0.1729	$1.06e^{-130}$
	p-wise	2.9912	0.5978	0.0	0.1743	$8.06e^{-133}$
	np-wise	2.7468	0.6665	0.0	0.2277	$2.27e^{-226}$
Permutated Clauses and Literals	random-1000	4.0936	0.3720	0.0	0.1452	$2.65e^{-92}$
	random-2500	3.7618	0.5264	0.0	0.2694	$1.05e^{-316}$
	random-5000	3.1681	0.6553	0.0	0.3354	0.0
	f-wise	3.4881	0.5087	0.0	0.1881	$1.29e^{-154}$
	nf-wise	3.1055	0.5632	0.0	0.1237	$4.15e^{-67}$
	p-wise	2.9832	0.6016	0.0	0.1926	$4.43e^{-162}$
	np-wise	3.2393	0.5622	0.0	0.3024	0.0

random-1000: random with 1,000 variants; random-2500: random with 2,500 variants; random-5000: random with 5,000 variants; f-wise: feature-wise; nf-wise: negative feature-wise; p-wise: pair-wise; np-wise: negative pair-wise

Table 6.3: Mean and median similarity between estimated PVDDs and the true PVDD for ten test cases.

Permutation Method	Sampling Method	Similarity Measure ED mean	ED median	PCC mean	PCC median	KS mean	KS median
No Permutation	random-1000	8.1316	8.1316	−0.0664	−0.0664	0.6446	0.6446
	random-2500	4.0005	4.0005	0.4391	0.4391	0.22680	0.2268
	random-5000	2.8623	2.8623	0.6560	0.6560	0.3939	0.3939
	f-wise	3.8191	3.81901	0.3898	0.3898	0.1800	0.1800
	nf-wise	2.8203	2.8203	0.6421	0.6421	0.1888	0.1888
	p-wise	3.1430	3.1430	0.5494	0.5494	0.1554	0.1554
	np-wise	3.3351	3.3351	0.5099	0.5099	0.2779	0.2779
Permutated Clauses	random-1000	3.8492	3.8939	0.3787	0.3777	0.1181	0.1074
	random-2500	3.7728	3.7690	0.5273	0.5294	0.2694	0.2694
	random-5000	3.1041	3.0740	0.6600	0.6600	0.3152	0.3017
	f-wise	3.6798	3.7019	0.4577	0.4560	0.1893	0.1916
	nf-wise	2.9091	2.8921	0.6244	0.6259	0.1703	0.172
	p-wise	3.1374	3.1377	0.5771	0.5873	0.2408	0.2389
	np-wise	3.2019	3.2367	0.5522	0.5610	0.2400	0.2248
Permutated Clauses and Literals	random-1000	3.9842	3.8698	0.3783	0.3860	0.1541	0.1357
	random-2500	3.7449	3.7473	0.5361	0.5373	0.2694	0.2694
	random-5000	3.1168	3.1519	0.6618	0.6589	0.3253	0.3354
	f-wise	3.4158	3.4952	0.5147	0.4965	0.1765	0.1881
	nf-wise	2.9873	3.0086	0.5975	0.5845	0.1539	0.1437
	p-wise	2.8613	2.8410	0.6352	0.6393	0.2000	0.2004
	np-wise	3.0790	3.1181	0.5757	0.5650	0.2271	0.2111

random-1000: random with 1,000 variants; random-2500: random with 2,500 variants; random-5000: random with 5,000 variants; f-wise: feature-wise; nf-wise: negative feature-wise; p-wise: pair-wise; np-wise: negative pair-wise

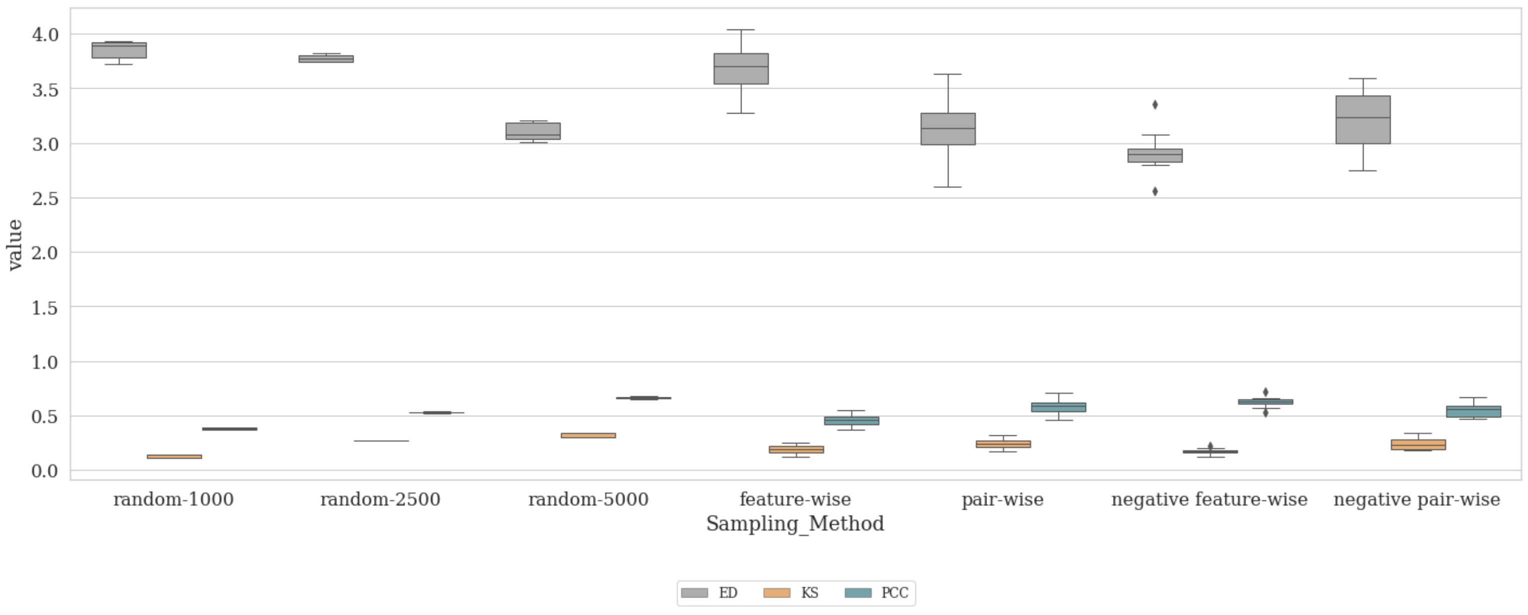


Figure 6.5

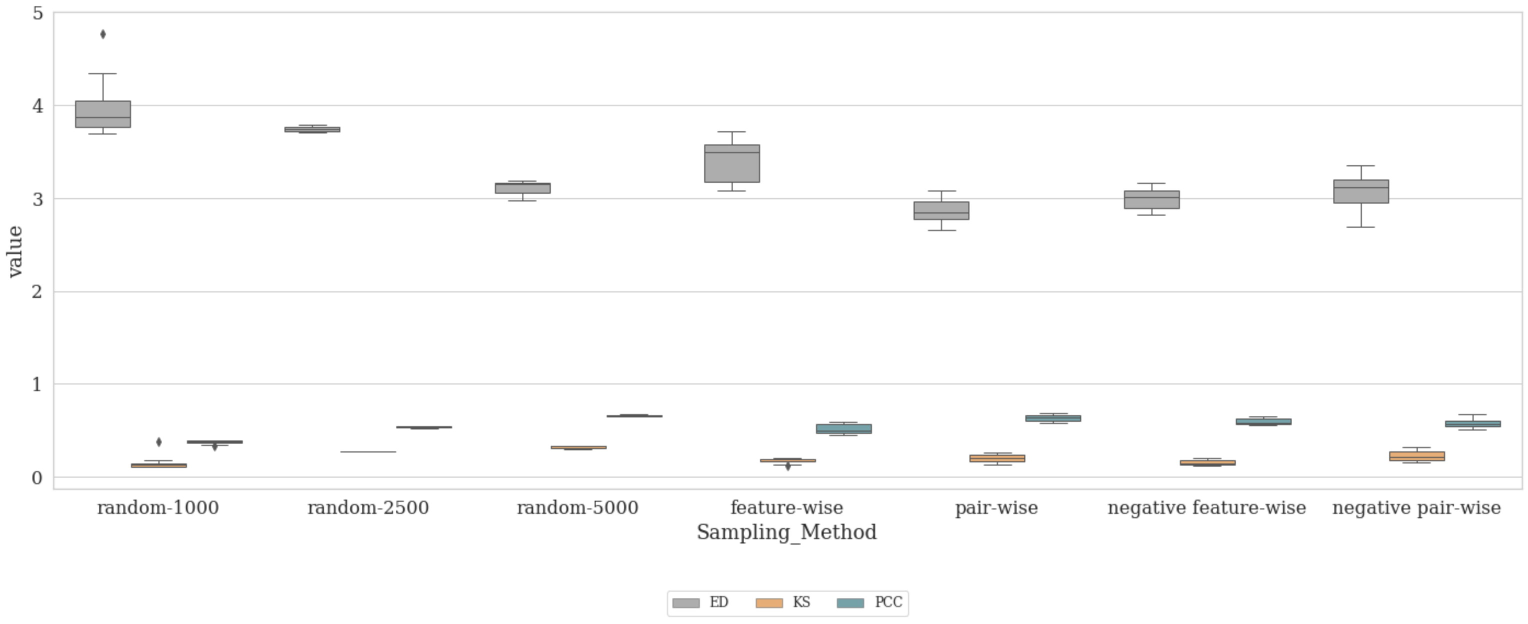


Figure 6.6

Table 6.4: Computation times in seconds for generating variant sets under different sampling and permutation methods

Sampling Method	Number of Variants	No Permutation	Permutated Clauses	Permutated Clauses and Literals
random	1000	0.189	8.985	79.326
random	2500	0.323	52.587	326.213
random	5000	0.760	482.168	2086.026
f-wise	94	0.170	0.220	1.218
nf-wise	96	0.174	0.219	0.706
p-wise	82	0.17971	0.211	0.985
np-wise	84	0.169	0.192	0.605

f-wise: feature-wise; nf-wise: negative feature-wise; p-wise: pair-wise; np-wise: negative pair-wise

negative feature-wise sampling without permutation is not only the best tested combination of sampling and permutation according to the achieved similarity but also the most time and cost efficient.

6.3 Transfer Learning Evaluation

In section 5 the functionality of AVM-modification was implemented. This functionality is intended to be a useful tool which can be used to evaluate the results of transfer learning approaches. In order to confirm that this tool can actually generate a modified AVM with which a rating can be realized, its results are compared to real results of an AVM generated by a transfer learning approach.

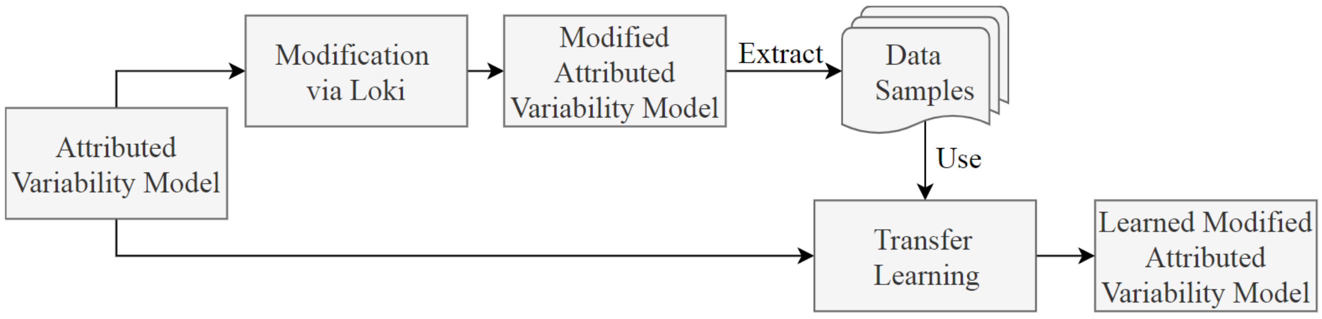

Attributed Variability Model
Modification via Loki
Modified Attributed Variability Model
Extract
Data Samples
Use
Transfer Learning
Learned Modified Attributed Variability Model

Figure 6.7: Overview over the process of modifying AVMs

For this experiment the toybox variability model with 544 features and 218 interactions was used once with regarding the interactions and once without. The model was modified six times by adding noise with varying severity, which resulted in modified AVM's with correlations shown in Table 6.5.

For each model samples were extracted. Those were used alongside the original model to approximate the modified model via a transfer learning approach (with courtesy to Pooyan Jamshidi, Ph. D, University of South Carolina). This process is illustrated in Figure 6.7.

Each training was repeated three times to account for basic measurement errors. The modified models obtained by Loki and by the transfer learning approach were then compared by computing the mean absolute percentage error (MAPE) and normalized mean squared error (NMSE). Both express the degree of deviation between the approximated and the true modified model, with the NMSE describing the overall deviations while the MAPE measures the error of prediction in terms of percentage.

The error measurements are shown numerically in Table 6.5 and graphically in Figure 6.8.

Table 6.5: Error measurements for the AVMs obtained via transfer learning.

	Scope of Modification	PCC	Sample Size	MAPE		NMSE	
				Mean	Median	Mean	Median
With Interaction	Small	0.993556	100	17.1549	17.1352	0.4160	0.3937
	Moderate	0.974252	100	19.5846	19.9347	0.5228	0.5031
	Large	0.584817	100	20.1130	19.1701	0.3220	0.3125
	Small	0.993556	200	4.7838	4.5281	0.0382	0.0344
	Moderate	0.974252	200	5.0474	5.1400	0.0421	0.0426
	Large	0.584817	200	6.6493	6.3181	0.0387	0.0352
Without Interaction	Small	0.998898	100	8.3720	8.1919	0.1652	0.1651
	Moderate	0.960956	100	11.0698	11.4650	0.2932	0.3155
	Large	0.735683	100	11.1880	12.4428	0.3070	0.3705
	Small	0.998898	200	5.1288	4.9327	0.0663	0.0601
	Moderate	0.960957	200	5.3023	5.3527	0.0723	0.0739
	Large	0.735683	200	4.1319	4.3188	0.0364	0.0400

The data shows that with an increasing number of samples the model is always learned more accurately. This can be seen in the smaller spread of the error measurements of bigger sample sizes as compared to the measurements of the small sample sizes. Small modifications are mostly learned with higher precision than medium or large modifications, which is a plausible result given that small changes result in a closer resemblance to the original AVM. The only exception is the model with large modifications without interactions and 200 used samples, which has smaller error measurements than the models with medium or small changes. Further testing is needed to explain this result.

For small sample sizes the data also shows that models containing no interactions are learned more accurately than models containing interactions. This is assumed to be due to the simpler nature of the model, since the lack of interactions reduces the model's dimensionality. However, with an increased sample size, this difference in accuracy decreases.

The results indicate that Loki's functionality of modifying AVMs is suitable as an evaluation tool for transfer learning approaches.

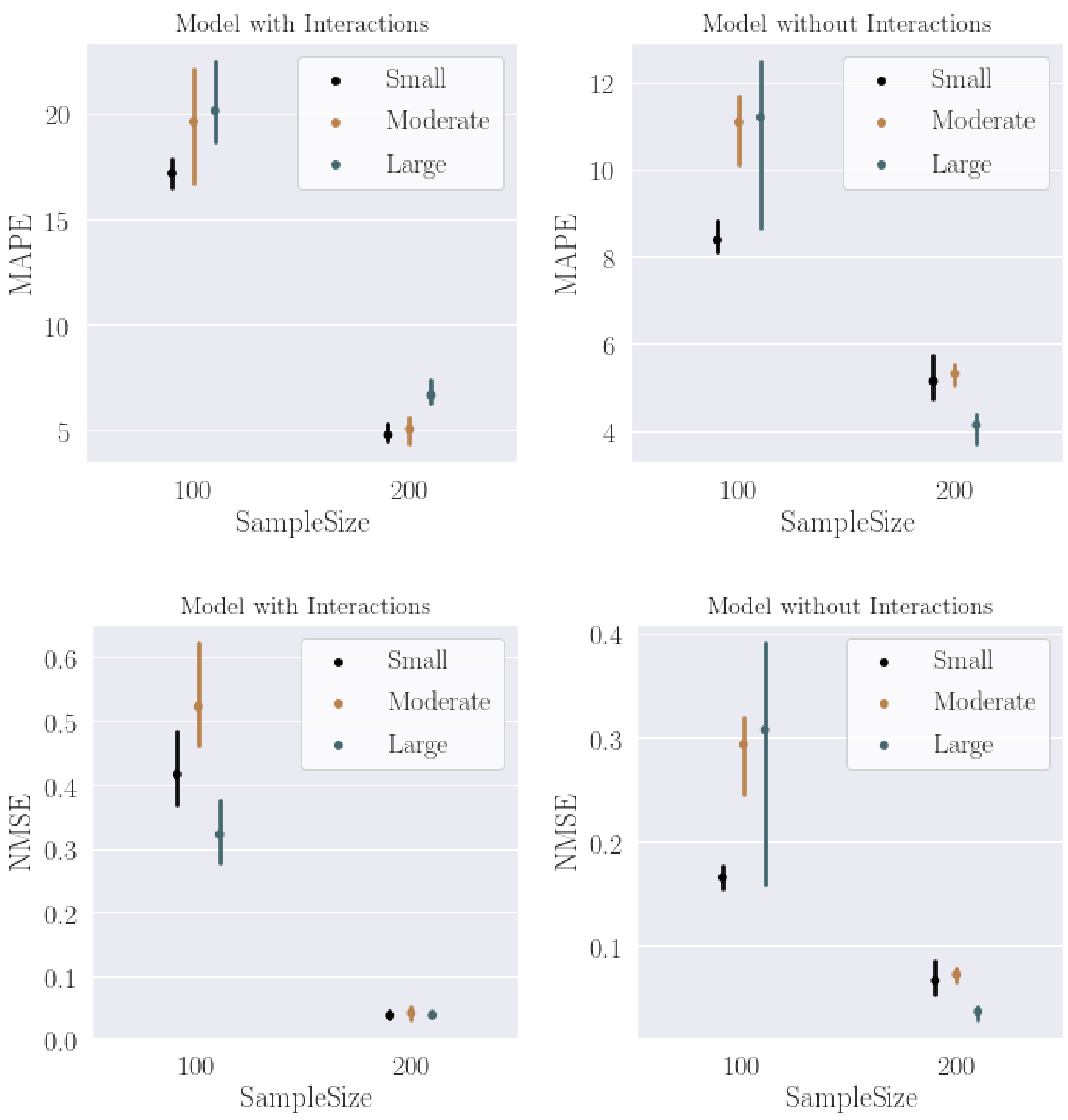


Figure 6.8: Value distribution of MAPE and NMSE depending on the sample size and the use of interactions

Chapter 7

Conclusion

In this thesis Thor - a system which extends a VM with realistic attribute values and interactions between features - has been re-implemented and extended with multiple functionalities in the Python 3 environment. This improved the accessibility and ease of usage for the userbase, since Thor suffers from a difficult installation and partially non-free third-party dependencies. For the new system, called Loki, the functionalities of Thor have been re-implemented, which is the generation of AVMs with interactions. Then, new possible options for configuring the generation process have been incorporated. This include sampling and permutation methods for generating variant sets, selection and recombination algorithms for the GA, and similarity measures for calculating the fitness value.

Tests showed that Loki is scalable and working reliably, even for models with large amounts of data. For example, Loki was tested with a model containing 526 features and 218 interactions. Another model, containing 100 features, was attributed by generating up to 10,000 variants, which showed that its computation time scaled linearly to the amount of variants. The added permutation methods are able to accurately approximate the PVDD of the model. Due to the Python 3 environment, Loki proves to be an easy to set up and accessible program while being highly configurable to fit the user's requirements.

The second added functionality for Loki is the modification of AVMs which allows for the evaluation of transfer learning approaches, specifically for related software systems. Loki offers the modification of features and/or interactions. The scope and type of modifications, such as linear transformation or added noise, can be specified by the user. It can also detect common and dead features automatically.

Tests showed that this functionality is also scalable and working reliably even for modifications with a high amount of iterations. For this test an AVM with 100 features was modified over the span of up to 5000 iterations, which showed a roughly linear increase in computation time. This functionality was tested with a transfer learning algorithm and models of varying modifications. The original and modified models were successfully used in evaluating the transfer learning system's results.

In summary, Loki is a promising program not only for attributing VMs and generating interactions, but also for providing an evaluation tool for transfer learning approaches in the research field of software system modeling by modifying AVMs.

Chapter 8

Outlook

Loki still has some unsolved problems that need to be dealt with in the future: One main concern is the possibility of the front rank having more members than the size of the archive can accommodate. In such a case - which is quite common if the population size is set to less than 100 - the individuals of the front rank are ranked and chosen according to their sparsity, which may result in a decrease of fitness and even in the loss of the current best solution. Futhermore, at this point only the DIMACS file format is supported. Extending Loki to support further file formats, such as xml or afm, would increase its usability and ease of access. Generating variants can take up a lot of computational time should the sample's size be large. If the same model is used more than once, it might be favorable to provide the system with a pre-generated set of valid variants, instead of computing them.

For the second functionality, only one transfer learning approach was available for testing. To further analyze the evaluation possibilities, modified AVMs by Loki should be used with multiple transfer learning approaches, to allow for a comparison between results.

Appendix A

Configuration Files

A.1 Configuration File for Generating Realistically Attributed Variability Models

```
[UseCase]
UseCase: AVM-Generation

[AttributedModel]
With_Variants:
DIMACS-file:
Feature-file:
With_Interactions:
Interactions-file:

[NonAttributedModel]
DIMACS-file:
Feature-file:
New_Interactions_Specs:

[Variants]
Sampling_Method:
NumberOfVariants:
Permutation_Method:

[NSGAII]
Population_Size:
Maximum_Generations:
Selection_Algorithm:
Recombination_Algorithm:
Similarity_Measures:
```

```
[Miscellaneous]
NumberOfThreads:
KDE_bandwidth:
NumberOfBins:
DirectoryToSaveResults:
ResultsToBeSaved:
ResultsCustomSpecs:
```

A.2 Configuration File for Modifying AVMs

```
[UseCase]
UseCase: AVM-Modification

[Model]
With_Variants:
DIMACS-file:
Feature-file:
With_Interactions:
Interactions-file:

[Variants]
Sampling_Method:
NumberOfVariants:
Permutation_Method:

[NSGAII]
Population_Size:
Maximum_Generations:
Selection_Algorithm:
Recombination_Algorithm:

[Scope_for_Changes]
Change_Feature:
Change_Feature_percentage:
Change_Interaction:
Change_Interaction_percentage:
Relevance_Treshhold:
Change_Operations:

[Noise_small]
Probability:
```

```
Mean:
Standard_deviation:

[Noise_big]
Probability:
Mean:
Standard_deviation:

[Linear_Transformation]
Probability:
Operation:
Operand:

[Negation]
Probability:

[Search_Space]
Find_common_and_dead_features:

[Miscellaneous]
NumberOfThreads:
KDE_bandwidth:
NumberOfBins:
DirectoryToSaveResults:
ResultsToBeSaved:
ResultsCustomSpecs:
```

Bibliography

Mathieu Acher, Anthony Cleve, Gilles Perrouin, Patrick Heymans, Charles Vanbeneden, Philippe Collet, and Philippe Lahire. On extracting feature models from product descriptions. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, pages 45–54. ACM, 2012. 3.1

Mathieu Acher, Benoit Combemale, Philippe Collet, Olivier Barais, Philippe Lahire, and Robert B France. Composing your compositions of variability models. In *International Conference on Model Driven Engineering Languages and Systems*, pages 352–369. Springer, 2013. 2.1, 2.1.1

Theodore W Anderson and Donald A Darling. A test of goodness of fit. *Journal of the American statistical association*, 49(268):765–769, 1954. 2.7.1, 2.7.1

Apel, Siegmund, Sobernig, and Leutheusser. Attributed variability models: Outside the comfort zone - supplementary website, 2018. URL `https://github.com/se-passau/thor-avm`. 1.1, 4.1

Sven Apel, Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. Exploring feature interactions in the wild: the new feature-interaction challenge. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*, pages 1–8. ACM, 2013. 1.1, 3.1

Krzysztof Apt. *Principles of constraint programming.* Cambridge university press, 2003. 2.2

Roberto J Bayardo and Daniel P Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *Proceedings of the National Conference on Artificial Intelligence*, pages 298–304. Citeseer, 1996. 2.2

Guillaume Bécan, Razieh Behjati, Arnaud Gotlieb, and Mathieu Acher. Synthesis of attributed feature models from product descriptions. In *Proceedings of the 19th International Conference on Software Product Line*, pages 1–10. ACM, 2015. 3.1

David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In *International Conference on Advanced Information Systems Engineering*, pages 491–503. Springer, 2005. 1.1, 2.1.2

David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010. 2.1

Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. Variability modeling in the real: a perspective from the operating systems domain. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 73–82. ACM, 2010. 2.1

Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A survey of variability modeling in industrial practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, page 7. ACM, 2013. 1.1, 2.1

Tobias Blickle and Lothar Thiele. A comparison of selection schemes used in genetic algorithms, 1995. 2.5.2

Zdravko I Botev, Joseph F Grotowski, Dirk P Kroese, et al. Kernel density estimation via diffusion. *The annals of Statistics*, 38(5):2916–2957, 2010. 2.4

Anthony Brabazon, Michael O'Neill, and Seán McGarraghy. *Natural computing algorithms*. Springer, 2015. 5, 6, 2.5.1, 2.5.3, 2.5.4

Jürgen Branke, Jurgen Branke, Kalyanmoy Deb, Kaisa Miettinen, and Roman Slowiński. *Multiobjective optimization: Interactive and evolutionary approaches*, volume 5252. Springer Science & Business Media, 2008. 2.3, 2.3

Lam Thu Bui and Sameer Alam. An introduction to multi-objective optimization. In *Multi-Objective Optimization in Computational Intelligence: Theory and Practice*, pages 1–19. IGI Global, 2008. 2.3, 2.3

Yen-Chi Chen. A tutorial on kernel density estimation and recent advances. *Biostatistics & Epidemiology*, 1(1):161–187, 2017. 2.4, 2.4, 2.4, 2.4, 2.4, 2.4

Stephen Cook, Rachel Harrison, Meir M Lehman, and Paul Wernick. Evolution in software systems: foundations of the spe classification scheme. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(1):1–35, 2006. 1.1

Krzysztof Czarnecki, Kasper Østerbye, and Markus Völter. Generative programming. In *European Conference on Object-Oriented Programming*, pages 15–29. Springer, 2002. 2.1.1, 2.1.1

Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. Cool features and tough decisions: a comparison of variability modeling approaches. In *Proceedings of the sixth international workshop on variability modeling of software-intensive systems*, pages 173–182. ACM, 2012. 1.1, 2.1

Kalyanmoy Deb and Hans-Georg Beyer. *Self-adaptive genetic algorithms with simulated binary crossover.* Secretary of the SFB 531, 1999. 2.5.3, 2.5.3

Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002. 2.5.5, 4.8, 5.7

Kalyanmoy Deb, Karthik Sindhya, and Tatsuya Okabe. Self-adaptive simulated binary crossover for real-parameter optimization. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1187–1194. ACM, 2007. 2.5.3

Michel Marie Deza and Elena Deza. Encyclopedia of distances. In *Encyclopedia of Distances*, pages 1–583. Springer, 2009. 2.7.4, 2.7.4

Sonja Engmann and Denis Cousineau. Comparing distributions: the two-sample anderson-darling test as an alternative to the kolmogorov-smirnoff test. *Journal of Applied Quantitative Methods*, 6(3):1–17, 2011. 2.7.1, 2.7.1, 2.7.1, 2.7.2

David B Fogel, Z Michalewicz, et al. Evolutionary computation 2: Advanced algorithms and operators. 2000. 2.5.1, 2.5.4

David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Professional, 1989. ISBN 0201157675. 2.5

Wael H Gomaa and Aly A Fahmy. A survey of text similarity approaches. *International Journal of Computer Applications*, 68(13):13–18, 2013. 2.7.4, 2.7.5

G Hall. Pearson's correlation coefficient. *other words*, 1(9), 2015. 2.7.3, 2.7.3

Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques.* Elsevier, 2011. 2.6, 2.6

Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. Combining multi-objective search and constraint solving for configuring large software product lines. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 517–528. IEEE Press, 2015. 4.4, 5.4, 6.2, 6.2.1, 6.2.1

Holger H Hoos and Thomas Stützle. *Stochastic local search: Foundations and applications*. Elsevier, 2004. 2.2

Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013. 2.7.3

Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal. Transfer learning for performance modeling of configurable systems: An exploratory analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 497–508. IEEE Press, 2017. 3.2, 5.9

Kyo C Kang and Hyesun Lee. Variability modeling. In *Systems and Software Variability Management*, pages 25–42. Springer, 2013. 2.1, 2.1.1

Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990. 2.1.1, 2.1.1, 2.1.2

Ahmet Serkan Karataş, Halit Oğuztüzün, and Ali Doğru. From extended feature models to constraint logic programming. *Science of Computer Programming*, 78(12):2295–2312, 2013. 2.1.2

William Benjamin Langdon. *Genetic programming and data structures*. University College London. Department of Computer Science, 1996. 2.5.1

Javier Larrosa and Thomas Schiex. Solving weighted csp by maintaining arc consistency. *Artificial Intelligence*, 159(1-2):1–26, 2004. 2.2

Thomas Leutheusser. Generating realistic attributed variability models. 2016. 1.1

Sean Luke. *Essentials of metaheuristics*, volume 113. Lulu Raleigh, 2009. 2.3, 2.5, 2.1, 5, 6, 2.5.2, 2.5.2, 2.6, 2.5.2, 2.7, 2.5.3, 2.5.3, 2.5.3, 2.5.3, 2.5.3, 2.5.4, 2.5.4, 2.5.4

Ulrike von Luxburg. Statistical learning with similarity and dissimilarity functions. 2004. 2.7

Frank J Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78, 1951. 2.7.2

Marcilio Mendonca, Moises Branco, and Donald Cowan. Splot: software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 761–762. ACM, 2009. 3.1

Sanjay Mittal and Brian Falkenhainer. Dynamic constraint satisfaction. In *Proceedings eighth national conference on artificial intelligence*, pages 25–32, 1990. 2.2

Gustaf Neumann, Stefan Sobernig, and Michael Aram. Evolutionary business information systems. *Business & Information Systems Engineering*, 6(1): 33–38, 2014. 1.1

ThanhVu Nguyen, Ugur Koc, Javran Cheng, Jeffrey S Foster, and Adam A Porter. igen: dynamic interaction inference for configurable software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 655–665. ACM, 2016. 3.1

Rafael Olaechea, Steven Stewart, Krzysztof Czarnecki, and Derek Rayside. Modelling and multi-objective optimization of quality attributes in variability-rich software. In *Proceedings of the Fourth International Workshop on Nonfunctional System Properties in Domain Specific Modeling Languages*, page 2. ACM, 2012. 2.1.2

Sinno Jialin Pan, Qiang Yang, et al. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010. 2.6, 2.6

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011. 4.6

Tania Pencheva, Krassimir Atanassov, and Anthony Shannon. Modelling of a stochastic universal sampling selection operator in genetic algorithms using generalized nets. In *Proceedings of the Tenth International Workshop on Generalized Nets, Sofia*, pages 1–7, 2009. 2.5.2

Justyna Petke. *On the bridge between constraint satisfaction and Boolean satisfiability.* PhD thesis, Citeseer, 2012. 2.2, 2.2

Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. *A field guide to genetic programming.* Lulu. com, 2008. 2.5.1

Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming.* Elsevier, 2006. 2.2, 2.2

Fritz W Scholz and Michael A Stephens. K-sample anderson–darling tests. *Journal of the American Statistical Association*, 82(399):918–924, 1987. 2.7.1

Philip Sedgwick. Pearson's correlation coefficient. *Bmj*, 345:e4483, 2012. 2.7.3

Sergio Segura, José A Galindo, David Benavides, José A Parejo, and Antonio Ruiz-Cortés. Betty: benchmarking and testing on the automated analysis of feature models. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, pages 63–71. ACM, 2012. 1.1, 3.1

Sergio Segura, José A Parejo, Robert M Hierons, David Benavides, and Antonio Ruiz-Cortés. Automated generation of computationally hard feature models using evolutionary algorithms. *Expert Systems with Applications*, 41 (8):3975–3992, 2014. 2.1.1, 2.1.1

Simon J Sheather. Density estimation. *Statistical science*, pages 588–597, 2004. 2.4, 2.4

Sidney Siegal. *Nonparametric statistics for the behavioral sciences.* McGraw-hill, 1956. 2.7.2, 2.7.2, 2.7.2

Norbert Siegmund, Sergiy S Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In *Proceedings of the 34th International Conference on Software Engineering*, pages 167–177. IEEE Press, 2012. 2.1.3

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*, pages 268–278. ACM, 2017. 1.1, 3.1, 6.1.1

Bernard W Silverman. *Density estimation for statistics and data analysis.* Routledge, 2018. 2.4

S.N. Sivanandam and S. N. Deepa. *Introduction to Genetic Algorithms.* Springer, 2010. ISBN 3642092241. 2.5, 2.5

Larissa Rocha Soares, Jens Meinicke, Sarah Nadi, Christian Kästner, and Eduardo Santana de Almeida. Varxplorer: Lightweight process for dynamic analysis of feature interactions. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*, pages 59–66. ACM, 2018. 3.1

Nidamarthi Srinivas and Kalyanmoy Deb. Muiltiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary computation*, 2 (3):221–248, 1994. 2.5.5

Michael A Stephens. Goodness of fit for the extreme value distribution. *Biometrika*, 64(3):583–588, 1977. 2.7.1

Mikael Svahnberg, Jilles Van Gurp, and Jan Bosch. A taxonomy of variability realization techniques. *Software: Practice and experience*, 35(8):705–754, 2005. 2.1

El-Ghazali Talbi. *Metaheuristics: from design to implementation*, volume 74. John Wiley & Sons, 2009. 2.3

Lisa Torrey and Jude Shavlik. Transfer learning. In *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*, pages 242–264. IGI Global, 2010. 2.6, 2.13, 2.6

Edward Tsang. *Foundations of constraint satisfaction: the classic text*. BoD–Books on Demand, 2014. 2.2

Thomas Weise. Global optimization algorithms-theory and application. *Self-published*, 2, 2009. 2, 3, 4, 2.5.1, 2.5.2, 2.5.2, 2.5.2

Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. A survey of transfer learning. *Journal of Big Data*, 3(1):9, 2016. 2.6, 2.6

Yi Yao and Gianfranco Doretto. Boosting for transfer learning with multiple sources. In *Computer vision and pattern recognition (CVPR), 2010 IEEE conference on*, pages 1855–1862. IEEE, 2010. 2.6, 2.6

Xinjie Yu and Mitsuo Gen. *Introduction to evolutionary algorithms*. Springer Science & Business Media, 2010. 1, 2.5, 2.5.2, 2.8, 2.9, 2.10, 2.5.4, 2.5.5, 2.12