

Optimizer Ensembles for Automated Machine Learning

Lukas Brandt

September 14, 2020



PADERBORN UNIVERSITY
The University for the Information Society

Department of Electrical Engineering,
Computer Science and Mathematics
Warburger Straße 100
33098 Paderborn



Intelligent Systems Group (ISG)

Master's Thesis

Optimizer Ensembles for Automated Machine Learning

Lukas Brandt

- | | |
|--------------------|---|
| <i>1. Reviewer</i> | Prof. Dr. Eyke Hüllermeier
Intelligent Systems and Machine Learning Group (ISG)
Paderborn University |
| <i>2. Reviewer</i> | Jun. Prof. Dr. Henning Wachsmuth
Computational Social Science Group (CSS)
Paderborn University |
| <i>Supervisor</i> | Marcel Meyer |

September 14, 2020

Lukas Brandt

Optimizer Ensembles for Automated Machine Learning

Master's Thesis, September 14, 2020

Reviewers: Prof. Dr. Eyke Hüllermeier and Jun. Prof. Dr. Henning Wachsmuth

Supervisor: Marcel Mever

Paderborn University

Intelligent Systems and Machine Learning Group (ISG)

Heinz Nixdorf Institute

Department of Electrical Engineering, Computer Science and Mathematics

Warburger Straße 100

33098 and Paderborn

Abstract

State-of-the-art approaches for Automated Machine Learning have combined their model selection and model configuration spaces into a joint optimization space, which is hence only examined with one optimizer. Recent publications have extended this joint space method by partitioning model selection and model configuration into two sub-spaces and approaching them alternating with two different optimization algorithms. However, for all possible outcomes of the model selection and all possible input datasets, the model configuration is always optimized with the same algorithm. According to the No-Free-Lunch Theorem for Optimization, one single optimization algorithm cannot be optimal for all problem classes in a constrained environment, for example a limited time budget. This thesis proposes an idea for tackling the model configuration with an ensemble of several optimizers, such that the most suitable optimizer for the current result of the model selection and the input dataset can be identified and utilized. An initial simple reference implementation of this idea showed partially competitive scores against current state-of-the-art AutoML tools in a comprehensive experimental evaluation.

Zusammenfassung

Ansätze für Automatisiertes Maschinelles Lernen aus dem aktuellen Stand der Forschung kombinieren gängigerweise das Auswählen des Modells and das darauffolgende Konfigurieren ebenjenes Modells in einem einzelnen Optimierungsraum, welcher deshalb auch nur von einem einzelnen Optimierer untersucht wird. Jüngste Publikationen haben diese gängige Methode jedoch erweitert und den vereinten Optimierungsraum in zwei Teil-Räume für die Modellauswahl und Modellkonfiguration aufgeteilt, die dann auch von zwei unterschiedlichen Optimierungsalgorithmen abwechselnd bearbeitet werden. Jedoch haben diese Ansätze immer noch für jedes mögliche Ergebnis der Modellauswahl und für jeden möglichen Eingabe-Datensatz ein und den selben Algorithmus zur Optimierung der Modellkonfiguration verwendet. Nach dem No-Free-Lunch Theorem für Optimierung ist es jedoch nicht möglich, dass ein einzelner Optimierungsalgorithmus für alle Problemklassen die optimale Lösung findet, solange er unter Beschränkungen arbeitet, wie zum Beispiel einem Zeitbudget. Diese Abschlussarbeit entwickelt einen Ansatz, in dem die Modellkonfig-

uration mit einem Ensemble aus verschiedenen Optimierern angegangen wird. Aus diesem Ensemble wird der Optimierungsalgorithmus herausgefunden und priorisiert verwendet, der am besten geeignet ist für den konkreten Eingabe-Datensatz, das aktuellste Ergebnis der Modellauswahl zu optimieren. Eine erste, einfache Referenzimplementierung dieses Ansatzes konnte zum Teil konkurrenzfähige Ergebnisse in einer umfassenden Experimentreihe gegen die derzeit im Stand der Forschung etablierten Ansätze erzielen.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Thesis Goal and Research Questions	3
1.3	Thesis Structure	4
2	Theoretical Foundations	5
2.1	Classical Machine Learning	5
2.2	Automated Machine Learning	7
2.2.1	General Workflow	7
2.2.2	Model Selection	9
2.2.3	Model Configuration	10
2.2.4	Formalization of AutoML as an Optimization Problem	11
2.3	Black Box Optimization	12
2.3.1	Differences to General Optimization Problems	13
2.3.2	Optimization by Searching	15
2.3.3	Genetic and Evolutionary Optimization	21
2.3.4	Bayesian Optimization	24
2.3.5	No-Free-Lunch Theorem	28
2.4	Related Work	29
3	AutoML via an Ensemble of Optimizers	35
3.1	Separation of Model Selection and Configuration	36
3.2	Model Selection with MCTS	36
3.2.1	Model Selection as a Graph Search Problem	37
3.2.2	Description of the Search Space Graph	39
3.2.3	Multiple Optimization Algorithms as a Multi-Armed Bandit Problem	43
3.2.4	Ensemble Interaction with MCTS	46
3.3	Model Configuration with Multiple Optimizers	49
3.3.1	Shared Parameter Domain for Selected Models	50
3.3.2	Warmstarted Versions of Optimization Algorithms	51
4	Implementation	55
4.1	Components of the Project and their Interaction	56

4.1.1	Model Selection Search Space Management	56
4.1.2	Pipeline Evaluation and Optimizers for Model Configuration .	58
4.1.3	MCTS and Optimizer Integration for AutoML	63
4.2	Utilized Python Libraries	65
4.3	Exemplaric File Format for defining the AutoML Optimization Space	66
5	Empirical Analysis and Evaluation	73
5.1	Research Questions in Detail	73
5.2	Experiment Setup	74
5.3	Results of the Experiments	78
5.4	Analysis of the Experiment Outcomes	80
5.4.1	Feasibility of AutoML via an Optimizer Ensembles	80
5.4.2	Optimizer Utilization Frequencies	82
6	Conclusion	89
6.1	Summary and Findings	89
6.2	Open Questions and Future Work	90
A	Appendix	93
A.1	Setup and Singularity Definition File of the Experiments	93
	Bibliography	99

Introduction

1.1 Motivation

Since machine learning is present in a fast-growing part of everyday life, economy, and science, the number of people who want to use machine learning and apply it to their use case is equally fast-growing. The problem is that using machine learning beyond simple examples and tutorials requires theoretical domain knowledge as well as solid programming skills.

Applying a machine learning approach includes the selection of a suitable learning algorithm or even multiple algorithms and other components combined as a complex pipeline. The cautious choice of machine learning algorithms and other data science components for each problem or use-case is crucial to have a suitable and well-performing pipeline.

Each selected pipeline component may expose hyperparameters. In turn, this hyperparameter configuration has to be tuned in terms of the specific problem instance. This is a second necessary step for the application of machine learning, because the optimal configuration values differ between problem instances.

Finally, the selected model or pipeline with the configured parameters must be implemented with a suitable programming language within a machine learning framework or toolbox, which are usually comparably big and complex to use.

Therefore, caused by these three necessary steps, there is a learning and knowledge barrier that prevents interested people with a non-technical background from using machine learning. Without having this deep machine learning domain knowledge, it is very hard to make suitable choices for the machine learning algorithm as well as corresponding hyperparameters.

But when the choices were not adequate, the performance measurements of the machine learning pipeline will only increase for a very high amount of training data or even not increase at all. However, the amount of data, which can be used to train a machine learning algorithm, is often limited for most use-case domains. Therewith, inadequate choices for components or configurations can not be compensated with a higher amount of data for these use-cases. Additionally, since this necessary deep knowledge in machine learning for making adequate choices is not broadly available and a lot of companies or research facilities have a high demand for machine learning

applications, it is not possible to apply machine learning in every use-case where it might be beneficial.

In the last years, different approaches were developed, which can empower nearly everyone to use machine learning for datasets of their domain or setting. These approaches allow this by automating this algorithm selection and configuration problem, consisting of the aforementioned three required sub-tasks: Selecting algorithms (respective components of pipelines), configuring all required hyperparameters, and constructing a usable implementation. All the end-user has to do is to provide their data in a suitable dataset format and usually specify a resource or time budget for the solver of the combined algorithm selection and configuration problem.

Within this budget, the best pipeline can be searched regarding a metric, as for example predictive accuracy. Hence, this problem of finding a machine learning pipeline for a dataset can be considered an optimization problem. Solving this optimization problem is the challenge of a research area called *Automated Machine Learning* or short *AutoML*.

AutoML received much attention in the scientific machine learning community and a broad spectrum of different methods were developed and published. The majority of these approaches and the plethora of different strategies they utilize have usually one thing in common. They see the model selection and the model configuration as a joint optimization problem and therewith solve these two problems within a single optimization space with a single optimization algorithm.

However, recently a few new approaches were published that partition this joint optimization space and approach with different optimization methods the two respective sub-spaces alternately. But this application of more than one optimization algorithm can be extended to three or more optimizers and in theory, this should show an improvement because of the *No-free-lunch Theorems for optimization*. It states that if an optimization algorithm performs under constraints superior for one problem or class of problems it has to pay for this by performing inferior for other problems. In the context of AutoML, this would imply that one optimization approach, can not yield the best solution across all datasets, timeouts, and pipeline component domains.

Combining multiple optimization methods into an ensemble of optimizers would allow this ensemble to optimize each possible pipeline with the most suitable optimization algorithm with respect to the parameter space of this pipeline and the properties of the dataset. Therewith, the most suitable optimization algorithm out of the ensemble could be identified and exploited to the given AutoML problem setting. For creating and utilizing these optimizer ensembles in the context of AutoML, a novel approach is required and is the topic of this thesis.

1.2 Thesis Goal and Research Questions

The recent related works approaches increased their number of applied optimizer up to two instead of the single optimizer in the established state-of-the-art approaches. However, implied by the No-free-lunch Theorems for optimization, this method would achieve the optimal results for only a slightly bigger set of problem instances compared to single optimizer approaches.

If the integrated selection of optimizers would be increased even further, this set of problem instances could theoretically be increased even further, under the assumption that more optimizers than just two are considered and the most suitable one is exploited. Additionally, during the evaluation of the different optimizers to identify the most suitable one for the problem instance, they could share gathered data between each other about their candidate evaluations up to that point.

This aspect of data sharing as well as identification and exploitation of the best suited optimizer introduces the novel approach of optimizer ensembles for AutoML problems. In the context of this thesis, a feasible concept and algorithm for such optimizer ensembles are devised and exemplarily realized as a reference implementation. Afterwards, the implemented optimizer ensemble algorithm is the test subject for a series of empirical evaluations.

Overall, the goal of this thesis is to answer two research questions regarding this approach for optimizer ensembles applied to the AutoML problem on the basis of the data gathered during these empirical evaluations:

1. Beyond the theoretical assumptions based on the No-Free-Lunch theorems, is the devised approach for optimizer ensembles indeed feasible in the context of AutoML? More precisely, is the result quality comparable or even better compared to other state-of-the-art approaches? And if it is a feasible approach, what influence have the particular components of the approach on the overall outcome?
2. Can knowledge about the optimization in the general AutoML context be extrapolated from this approach? From a theoretical point of view, the suitability of an optimization algorithm for a model configuration problem could depend on a variety of different factors like the dataset, the optimization budget, the components of the machine learning pipeline whose parametrizations are optimized, and several others. The pipeline components are influenced by the model selection and the timeout depends on the environment of the user. But the relationship between input datasets as well as more generalized properties of the datasets in relation to the optimizer utilization can be evaluated and could be valuable insights beyond the discussion approach.

When the frequencies of utilizing certain optimizations algorithms during the

execution of this approach are recorded, this may give indications regarding their capability for different AutoML problems, since this approach tries to find the best suited optimizer out of the ensemble for the input problem, i.e. the dataset with its properties, and exploit it. Based on these frequencies, is every optimizer called an equal number of times, or is one or more optimization method favored for the AutoML use case? Does it depend on certain dataset properties of the concrete AutoML problem, whether an optimization method is used relatively often?

1.3 Thesis Structure

Following the current introduction chapter, this thesis is structured into five chapters. The content and goals of each chapter are briefly explained here:

Chapter 2

At first, the foundations of the machine learning, the AutoML setting, and optimization in general are elucidated, to give a starting point for the related work part of this chapter and the approach of this thesis.

Chapter 3

The optimizer ensemble approach of this thesis is theoretically specified here and the underlying design choices are explained and justified.

Chapter 4

To apply the approach for actual AutoML problem settings as well as evaluating the approach empirically in an experiment series, an actual implementation is necessary and is explicated in this chapter.

Chapter 5

In order to answer the previously listed research questions, a series of empirical experiments is designed and the results analyzed in this chapter.

Chapter 6

Finally, the answers to the research questions are used to summarize the concepts of this approach and their validity. At last, an outlook is given with a selection of starting points for future work and follow-up research.

Theoretical Foundations

Before presenting the approach for tackling AutoML with an ensemble of optimizers, some theoretical foundations of both elements, AutoML and optimization, are given in the following. This theoretical background is structured in three parts:

1. Some basic concepts and intuitions of machine learning in general are outlined alongside with the challenges and problems that arise when applying machine learning.
2. The concepts and usual approaches of AutoML are introduced, which were developed to tackle the listed challenges of classical machine learning. In addition, the connection between the AutoML setting and typical optimization problems is illustrated.
3. As the foundation for building an ensemble of optimizers a selection of established optimization methods is given and explained.

The overview of optimization methods is concluded with the discussion of a theoretical drawback of using a single optimization method. Before explaining the ensemble approach in the next chapter, which could counter this drawback to a certain degree, this discussion is continued with a selection of related works publications, where other approaches that addressed this theoretical disadvantage of using a single optimization method are mentioned.

2.1 Classical Machine Learning

Machine learning in general has been a topic in computer science and mathematics for several decades even before attempts to automatize it in the form of AutoML were made. AutoML is of course not replacing this classical approach, but it is an extension of it. Therefore, a short overview from a more abstract and theoretical point of view of this classical machine learning without AutoML is given in this first theory section. Especially supervised learning is elucidated here and one possible definition is given since the approach of this thesis focusses on classification settings,

which are a sub-group of supervised learning settings. With this foundation, AutoML is explained in the following sections.

If a human is given a task where the correct solution or reaction is not evident, a human has always the possibility to react with a random solution or with an arbitrary reaction. But if the human has any prior first- or second-hand experience with the same or a similar task, the human can choose the reaction based on the memories of different outcomes for different reactions for the more or less similar prior task. With the high abstraction level and very symbolic nature of human thinking and memorization, it is comparably easy for humans to recognize even remote similarities between tasks.

This is very challenging for a computer in contrast because the models of tasks, experiences, and outcomes of reactions to tasks have to be readable and understandable for a machine, i.e. be in any kind of structured and consistent format. This setting was formalized by Mitchell [Mit97] as a combination of T , P and E , where T is a class of tasks, P a performance measurement for solutions of a specific instances of the tasks class T and E is either given or collected experience, i.e. performance measurements for certain solutions in the context of specific task instance $t \in T$.

Of course it is possible for a human programmer to manually specify the solution with the best performance measurement for any possible $t \in T$ but for a high $|T|$ this is rarely possible and viable. Here, machine learning has its use-case: "Machine learning enables us to tackle tasks that are too difficult to solve with fixed programs written and designed by human beings" [GBC16].

A high number of different types of task classes T is imaginable, but one of the most common ones in the machine learning field is *Supervised Learning*. In supervised learning, T includes a fix set of all possible solutions S . The concrete task for Supervised Learning is now to select for a given $t \in T$ a $s \subseteq S$ such that the performance measurement $P(t, s)$ is optimal. To enable a machine to learn supervised, the experience E has to be successively build in the form of $\{(t_1, s_1, P(t_1, s_1)), \dots, (t_n, s_n, P(t_n, s_n))\}$.

For a new task instance t_i , the computer will select an s_i based on a decision model build from E , which is supposed to be at least better than random guessing. Depending on the selected s_i , the computer will receive a performance feedback $P(t_i, s_i)$ and finally enrich E with $(t_i, s_i, P(t_i, s_i))$ as well as updating the decision model with the changed E . Therewith, a well working machine learning algorithm for supervised learning would now be able to achieve $P(t_j, s_j) \geq P(t_i, s_i)$ if t_j and t_i are similar instances, since it already has experience values which selection for $s_i \subseteq S$ had achieved a certain performance value for t_i and might therefore be a good or respectively bad choice for t_j such that s_j can be deduced to a certain degree.

This task of comparing and judging the similarity of different instances of T and to build a practicable decision model based on E to select a solution out of S has been tackled with a plethora of different algorithms or even combinations of multiple algorithms as a form of machine learning pipeline. Usually, these algorithms have to be configured with a set of hyperparameters depending on T and often T also has to be transformed before presenting concrete instances to the algorithm for learning, i.e. pre-processing each $t \in T$ with one or more transformation methods.

2.2 Automated Machine Learning

Automated Machine Learning, or short *AutoML*, tries to automate most tasks of the process of creating a machine learning application and also other associated tasks from the data science field. This serves two main purposes:

- By automating construction and configuration, it tackles the knowledge barrier, which prevents interested people from applying machine learning.
- By reducing the amount of manual work to a bare minimum, it offers a big speed-up for applying machine learning regardless of the level of knowledge of the user.

In this section, a description of the general workflow of an AutoML tool is given and the usual two main steps of such an AutoML workflow are illustrated. This is concluded with a formalization of the AutoML problem setting as an optimization problem.

2.2.1 General Workflow

AutoML applications are usually designed very homogeneously and therefore have very similar workflows among themselves. As an input, the application receives data in a machine-readable format and usually some form of constraint for its execution, such as hardware resource or time limitations.

The expected output is the best found machine learning pipeline for the given data measured by some task-dependent metric and additionally often the actual performance value of the found machine learning pipeline measured in said metric. Additionally, each AutoML tool can have further necessary configurations as an input.

The constraints for execution are required to prevent the AutoML tool from searching for the best machine learning pipeline for the given data virtually forever with probably continuously growing hardware consumption. Therefore, the execution

time and/or some hardware resources are constrained in the form of budgets. These inputs and outputs of a conceptual AutoML tool are illustrated in Fig. 2.1.

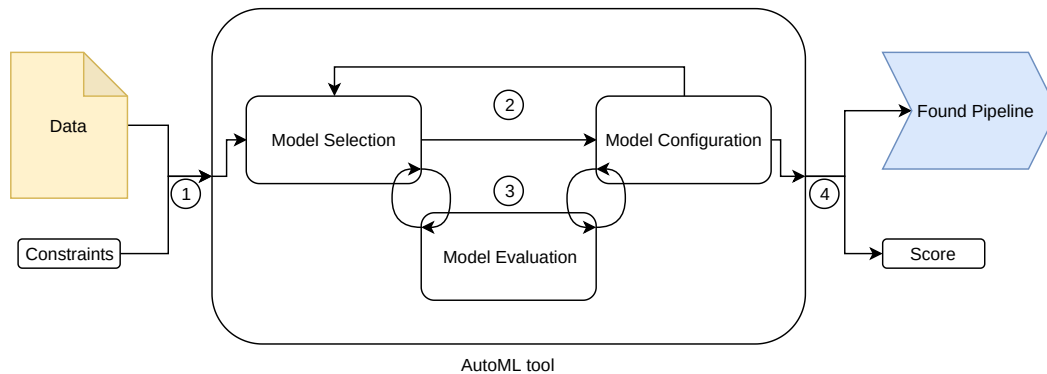


Fig. 2.1.: High-level view of a conceptual AutoML tool.

The general workflow of an AutoML tool, as numbered and illustrated in the figure, can be generalized with the following steps:

1. The AutoML tool gets some input data and execution constraints as an input.
2. Model selection and model configuration are repeated until execution budgets are spent.
3. Model selection as well as model configuration can perform a model evaluation to get a score for any pipeline candidate they produce.
4. When the user-specified constraints are reached, for instance a timeout is hit, the best evaluated pipeline as well as the score of this pipeline are returned as an output.

The model evaluation step is usually performed by training a pipeline on one split of the input data and testing it with another split of the data, sometimes called validation data, while observing some kind of performance measurement, for example the accuracy of predictions or some error metric. But other evaluation techniques, for example a cross-validation, are also common.

While the model evaluation step is only very loosely defined and usually not a complex procedure, the model selection and model configuration steps of the AutoML workflow are more sophisticated. They have to select a suitable candidate from one or more, usually large, spaces. Both spaces, for model selection as well as for model configuration, are individually explained in the following.

2.2.2 Model Selection

Model selection is the task of selecting 1 to n components that will be part of the machine learning pipeline and sequentially traversed when data is passed through the pipeline. For example, a simple, but valid pipeline would be to apply a Principal Component Analysis on the data as a first component and to use a Support Vector Machine for classification on the processed data as a second component afterwards. In this step of the workflow, the space one candidate has to be selected from, consists of all these valid pipeline component combinations, which usually consist of two or more components.

Usually, there are three types of components, although other classifications of the components are possible as well:

- Pre-Processing: Transform the input data before presenting it to the learning algorithm.
- Learning Algorithm: Perform the actual machine learning task on the data, for example classify a datapoint after training.
- Post-Processing: Transform the output of the learning algorithm before yielding it as a final result.

Although in theory, only a single learning algorithm is necessary and components for pre- and post-processing are optional, having at least a pre-processing component is very common in machine learning pipelines and included most of the time.

It is possible to use more than one component of each of the three types in a single pipeline, such that pipelines with arbitrary complexity and size can be created. More than one pre- or post-processor can be used in sequence or in parallel with some kind of aggregator component subsequently. Similarly, more than one learning algorithm can be combined to be used as a composite or ensemble model with a proper aggregation method as for example Bagging of models [Bre96]. An example of such a more complex pipeline is illustrated in Fig. 2.2.

Model selection, where the resulting pipelines can have this complexity, needs to add a structural relationship between the selected pipeline components. Therewith, a valid model selection result can be seen as a simple formalization consisting of two elements:

- With a given set of all possible components C a list of components $C' = \langle c_1, \dots, c_r \rangle$ is created with $\forall c_i, i \in [1, r] : c_i \in C$. The components of that list are the selected components that will be used to construct the pipeline.

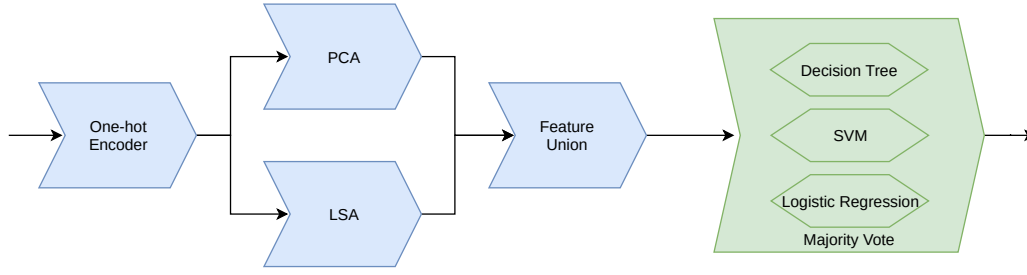


Fig. 2.2.: Example of a more complex machine learning pipeline with four pre-processing components and an ensemble learner. The pre-processors are shown in blue: A One-hot Encoder as well as a subsequent Feature Union of a PCA and an LSA. The learning algorithm in green is a Majority Voter ensemble as an aggregation of a Decision Tree, an SVM and a Logistic Regression classifier.

- A binary structure relation $\prec \subset C' \times C'$. It defines which components are a data input for another component to define an order of the components for the pipeline, for instance $(c_a, c_b) \in \prec$. This is also possible for aggregation components or other components with multiple expected inputs by adding all inputs individually. For example, with an aggregation component c_a and three input components c_i, c_j, c_k , this could be $\{(c_i, c_a), (c_j, c_a), (c_k, c_a)\} \subset \prec$.

Nevertheless, to find valid results for C' and \prec is not a trivial task. For instance, when an aggregator for multiple parallel pre-processing components is needed, a valid choice could be a Feature Union component but a Decision Tree component would be invalid for this pipeline position.

Therefore, if the pipelines shall be allowed to be more complex than for example a single pre-processing component and a single learning algorithm, there is depending on C' a probably big set of constraints for the binary structure relation \prec .

2.2.3 Model Configuration

With the component list $C' = \langle c_1, \dots, c_r \rangle$ and the associated binary structure relationship \prec as the two results of the model selection step, the pipeline is not ready to be used yet. Usually each component of $c_k \in C'$ has a set of hyperparameters and for each one a concrete value is needed, which is the task of the model configuration step.

The candidate space for the model configuration step is completely dependent on the resulting C' . Each component c_k has a set of hyperparameters $\Theta_k = \{\theta_{k_1}, \dots, \theta_{k_j}\}$ where each hyperparameter has a range of valid values for this hyperparameter from which values have to be chosen from. Such a range can for example be a numeric set like \mathbb{N} or a set with custom values as for example $\{\text{true}, \text{false}\}$.

When combining all ranges $\bigcup_{i=1}^r \Theta_i$, a parametrization space with dimension r is defined as the candidate space for the model configuration step, where configuration vectors can be drawn from. If there are no additional constraints attached to the possible configurations, all vectors of this space represent valid candidates and will result in a working pipeline instance created from C' and configured with this vector. When such a pipeline is constructed and configured with the vector values it can finally be evaluated as a candidate pipeline. An example for a concrete parametrization drawn from a three dimensional parameter space is shown in Fig. 2.3

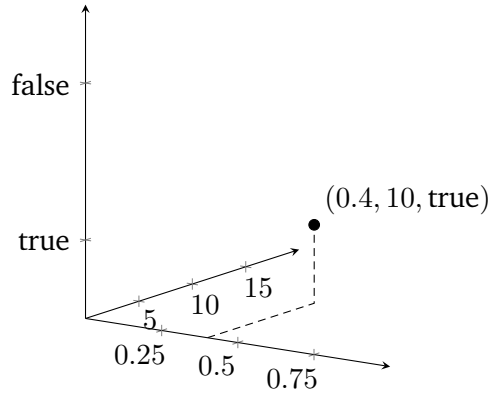


Fig. 2.3.: An exemplaric parameter configuration space. One concrete configuration with the values $(0.4, 10, \text{true})$ inside the parameter space was selected. The overall space has the dimensions $[0..20]$, $[0, 1)$ and $\{\text{true}, \text{false}\}$.

2.2.4 Formalization of AutoML as an Optimization Problem

The choices in the model selection and model configuration steps for one candidate out of the corresponding candidate spaces should not be arbitrary. Evaluations of a candidate pipeline yield a score for this candidate and intuitively a suitable AutoML approach should try to select candidates with scores as good as possible. Therefore, the AutoML problem is often considered as an instance of an optimization problem and can be formalized as one as well.

Thornton et al. [Tho+13] give a formalization for AutoML problems as a so called *Combined Algorithm Selection and Hyperparameter optimization* problem, or short *CASH* problem. Their definition is the following:

$$A^*, \lambda_* \in \operatorname{argmin}_{A^{(j)} \in \mathcal{A}, \lambda \in \Lambda^{(j)}} \frac{1}{K} \sum_{i=0}^K \mathcal{L}(A_{\lambda}^{(j)}, D_{\text{train}}^{(i)}, D_{\text{valid}}^{(i)})$$

This formula consists of the following parts:

- A^* : The best solution of the algorithm selection, i.e. for the AutoML setting the best pipeline construction found in the model selection step
- λ_* : The best hyperparameter configuration for A^* found in the the model configuration step
- \mathcal{A} : A set of possible algorithms to choose from and given in the form $\mathcal{A} = \{A^{(1)}, \dots, A^{(R)}\}$, which is the set of all valid pipeline component combinations in the case of AutoML
- $\Lambda^{(j)}$: Each algorithm $A^{(j)} \in \mathcal{A}$ has a parameter configuration domain $\Lambda^{(j)}$, where a concrete parametrization λ can be selected from
- $\frac{1}{K} \sum_{i=0}^K$: This is used to calculate the K-fold cross-validation of a pipeline, but other evaluation strategies are also possible if the formula is adjusted accordingly
- $\mathcal{L}(A_\lambda^{(j)}, D_{train}^{(i)}, D_{valid}^{(i)})$: This is the loss, calculated with the loss function \mathcal{L} , of the algorithm $A^{(j)}$ configured with the parametrization λ , trained on the i -th split of the training data $D_{train}^{(i)}$ and evaluated on the validation data $D_{valid}^{(i)}$

Instead of the integration of the K-fold cross-validation in the definition of Thornton et al. via $\frac{1}{K} \sum_{i=0}^K$, it is of course also possible to replace it with any evaluation method.

In summary, the goal is to select an algorithm together with a parameter configuration, such that the expected loss calculated via a cross-validation is minimal.

2.3 Black Box Optimization

With the formalization of AutoML as a CASH problem, the question arises whether it can be tackled similar to standard textbook optimization problems, where established optimization algorithms exist to approximate or to find an optimal solution. Unfortunately, the CASH problem is a special case of optimization problem called *Black Box* optimization problem.

In the following section, it is explained in what way black box problems are different from the standard textbook optimization problem and why established optimization algorithms cannot be used. Afterwards, a selection of existing algorithms is presented that can instead be used for black box optimization problems, as for instance the AutoML setting. Finally, with the aid of the *No-Free-Lunch Theorem*, it is reasoned

why these approaches cannot be optimal and why therefore an ensemble approach could be more promising.

2.3.1 Differences to General Optimization Problems

Boyd and Vandenberghe [BV14] define a standard single-objective optimization problem as the following:

$$\begin{array}{ll} \underset{x}{\text{minimize or maximize}} & f_0(x) \\ \text{subject to} & f_i(x) \leq 0, i = 1, \dots, m \\ & h_j(x) = 0, j = 1, \dots, p \end{array}$$

Here is the goal to find a value for the optimization variable x such that $f_0(x)$ has either its minimum or maximum value. Hereby, x does not have to be a single value but can also be a vector of values. f_0 is usually called an objective function, target function, or cost function.

The choices for x can be limited by a set of inequality constraint functions f_i and a set of equality constraint functions h_j . If $m = p = 0$, there are no constraint functions and the optimization problem is called unconstrained. For such cases, x could be any value from the domain of f_0 .

If f_0 has certain properties, for example if it is a convex function, there are specialized optimization methods that can solve the optimization problem without the derivative f'_0 . But for general optimization algorithms, where no properties of f_0 besides derivability are expected, f'_0 is required to solve the optimization problem analytically.

The problem is, for some optimization problems, the exact formula of f_0 is not known or not given and therefore no derivative can be calculated such that the general optimization algorithms are not applicable. Instead, it is only possible to obtain $f_0(x)$ for any requested x for example by performing an experimental evaluation, looking the result up in a table, or asking an expert. For such problems it is only possible to observe the inputs and corresponding outputs of f_0 but the inner workings are hidden. f_0 is metaphorically a black box, where something goes in and something comes out but it is not possible to look inside, and therefore such optimization problems are called black box optimization problems.

AutoML, formalized as a CASH problem, is a black box optimization problem as well. It is not possible, or at least not yet realizable, to determine a general formula for f_0 . The relationship between the properties of the dataset, the different pipeline components with their configuration parameters and other factors, as for example

randomness of some components, is way to complex and inscrutable to be formulated into a single cohesive function.

But it is possible to get an evaluation of any x , i.e. a concrete pipeline with a complete configuration, by training and testing the pipeline with given data and an evaluation metric as for example the prediction accuracy. The absence of a concrete formula for f_0 but the possibility to determine $f_0(x)$ renders AutoML a black box optimization problem.

Without the possibility to determine an optimal or near optimal candidate analytically, as in the case of a black box optimization problem, a series of candidates has to be selected and evaluated while trying to find or at least approximate an optimal solution. Now the question remains for the AutoML setting, how candidates from the two presumably large candidate spaces of model selection and model configuration should be selected for each iteration of the candidate selection and candidate evaluation loop of the optimization and how such an approximation can be attempted.

A solely random selection in every iteration is not very likely to find good results with such a high number of choices. Since the approach evaluates several candidates iteratively, a better approach would be to base the selection in the current iteration on an updated view of the optimization space landscape based on the knowledge gathered from previous iterations instead.

Several algorithms were developed to tackle such an iteratively approximation of the optimum of a black box optimization problem. Some of them have already shown promising results for CASH problems and therefore the current state-of-the-art AutoML research works primarily with the following three optimization strategies:

- (Heuristic) Search
- Genetic and Evolutionary Algorithms
- Bayesian Optimization

In general, every black box optimization strategy is defined by a selection criterion for a set of candidates for evaluation out of the candidate space and an order in which they are evaluated. Hereby, the selection as well as the order can be defined in many ways. For example, the two could be (partially) pre-defined or random or a mixture of both. It is also possible that the combination of both, definition of candidates as well as order of candidate evaluations, is defined implicitly by the prior selected candidates. In this case, selecting one more candidate for the set would depend on the already selected candidates in the set as well as knowledge about the

optimization landscape gathered from them, and with this stepwise extension the order of evaluations would be implicitly defined.

For each one of the aforementioned black box optimization strategies, this candidate selection and candidate order are explained in general and on the basis of some concrete approaches from the AutoML research in the following. These explanations of the strategies are supported by a simple running example.

The unconstrained target function $f_0\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = x^2 + y^2$, which can be seen in Fig. 2.4, shall be minimized. This function is defined in \mathbb{R}^2 and has one global minimum. Although the running example is a minimization task, the presented algorithms can also perform maximization tasks out of the box or with minor modifications.

This function with only two dimensions would probably not be the target function of any AutoML problem and since it is derivable and convex, there is actually no need to apply a black box optimization algorithm. However, this function can be illustrated clearly and has not a complex shape and applying optimization algorithms on it is therefore easier to understand than a more realistic function.

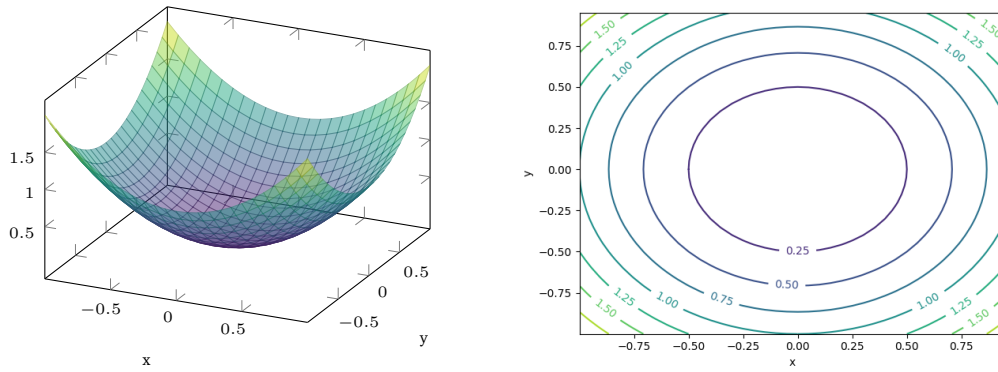


Fig. 2.4.: A two dimensional optimization target function. It has a global minimum at $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$. Displayed as a surface plot on the right and on the left as a contour plot.

2.3.2 Optimization by Searching

Optimizing with a search will evaluate candidates with the concept of *Neighborhood* between candidates. Beginning with a single starting candidate, the search will extend its set of evaluated candidates by selecting one candidate a out of this set and evaluating one candidate out of the set of neighboring candidates of a .

The concrete definition of this neighborhood concept is depending on the examined search space, which is constructed from the optimization candidate space, and the applied search algorithm. For example, a neighbor could be any candidate within

a certain distance of another candidate according to a given distance metric, or a graph could be created out of a set of candidates and neighbors of a candidate are the adjacent candidate nodes in the graph.

Which neighbor should be selected for evaluation next, and therefore defining the order of candidate evaluations implicitly, can vary with the different approaches as well. It could be completely random, pre-defined before the search starts, or based on a heuristic scoring of the candidates.

A wide range of search algorithms was developed in the corresponding research area and with a suitable neighborhood concept, most of them are applicable to optimization problems as well. Some state-of-the-art AutoML approaches use search algorithms as their optimization method. Three of these search algorithms are explained in more detail with the aid of the running example in the following.

Random Search

The random search algorithm comes in two variants. In the first one, the candidates are drawn completely at random out of the whole candidate space and evaluated. When the optimization budget is spent, the best candidate is returned as the result. Since this method does not use any knowledge gathered during the optimization, it depends on chance and the optimization budget if the optimal solution or at least a near optimal solution is found.

In the second one, a starting candidate is drawn at random out of the candidate space and evaluated. The next candidate will be randomly drawn out of a hypersphere with a pre-defined radius r around the starting candidate and evaluated, i.e. for all dimensions, if the value z is the current value for this dimension, it has to be from the range $[z - r, z + r]$. If the drawn candidate has a better evaluation score than the starting candidate, the next candidate will be drawn from a hypersphere around the second candidate and from the hypersphere of the starting candidate otherwise. The loop is continued until the optimization budget is spent. By choosing a large r the risk of jumping over an optimum increases, but with a low r the coverage of the search of the optimization space will be low as well.

However, this method is very prone to local minima/maxima. This can be circumvented to a certain degree by stopping the loop of sampling from the hypersphere around the current best candidate if there was no improvement during the latest n iterations. In this case, the currently searched area is disregarded and the loop is started over with a completely new random start point. Both variants are illustrated for the running example in Fig. 2.5.

One example of the random search algorithm applied in the context of AutoML is *Hyperband* [Li+16]. In the beginning, a big set of candidates is randomly cho-

sen to have a high chance for big coverage of a joint optimization space of model selection and model configuration. Therefore, this is basically multiple steps of the first random search variant at once. These chosen candidates are evaluated in parallel and the single evaluations are iteratively terminated, when it becomes evident that the corresponding candidate is not competitive with other currently evaluated candidates. Selecting candidate evaluations for termination is tackled with concepts from Multi-Armed Bandit problems. Each evaluation instance is treated as the arm of a bandit and single instances are terminated over time, if they do not yield competitive first results during the evaluation. The probability that this candidate will improve significantly later in the evaluation process is low and therefore there is a low regret when terminating and not completing the candidate evaluation. After freeing a part of the optimization budget that was allocated for a candidate evaluation by terminating it, the gained budget can be used for a new randomly chosen candidate. This new candidate is added to the pool of evaluation instances as a new bandit arm. By terminating some probably low performing evaluations, the optimization budget is not wasted on fully evaluating candidates from low performing areas and increasing the chance for a good optimization space coverage instead with more evaluations. Strategies like this one are often referred to as *Early Stopping*. In the concrete context of Hyperband, selecting the instances, which will not be stopped early, is done in a non-stochastic manner with the *Successive Halving* algorithm [JT15].

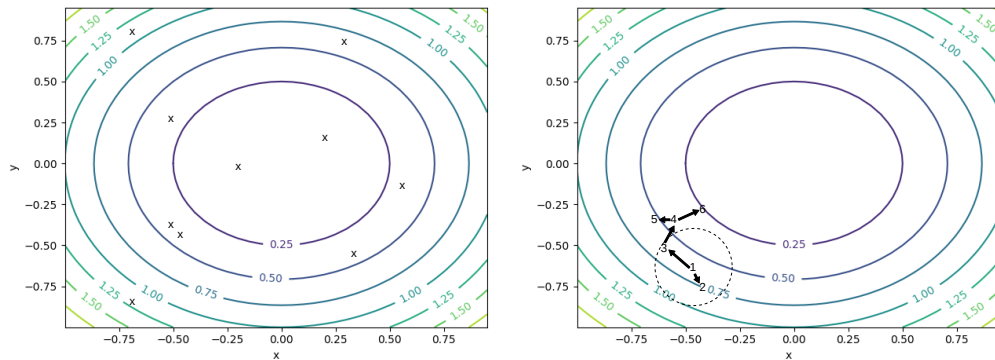


Fig. 2.5.: Visualization of different variants of random search strategies. On the left is the first variant with a few randomly selected candidates. On the right are six possible first iterations of the second variant. The hypersphere is only shown for the first candidate.

Grid Search

A grid search tries to cover with a naïve brute-force approach as much of the candidate space as possible. From each dimension of the candidate space, a few values are

selected either manually, or more commonly in periodic intervals from the different dimensions. For example, if the x-dimension of the running example should be evaluated in the range $[-1, 1]$, depending on the available optimization budget, potential value selections for this dimension could be $\{-1, 0, 1\}$ or $\{-0.75, -0.25, 0.25, 0.75\}$. With this method the candidate space is discretized and therewith becomes enumerable.

After selecting the value sets for each dimension, the Cartesian product of all sets is calculated. The elements of this Cartesian product, which are valid candidates because they have a value of each dimension, are the candidates that will be evaluated during the search and for the order of evaluations, any arbitrary sequence of the set elements can be selected. After either all candidates from the Cartesian product are evaluated or all of the optimization budget is spent, the best evaluated candidate is the overall optimization result. An illustration of a grid search over the candidate space of the running example can be seen in Fig. 2.6.

Therefore, as opposed to random searches where by chance only a small portion of the candidate space could be covered, a certain coverage of the space is assured. Especially in contrast to the second variant of the random search, the advantage is that there is no risk of getting stuck in a local optimum. But in comparison with the second random search variant, the main drawback of a grid search comes clear as well: A grid search has pre-defined candidates that are homogeneously distributed in the space, including the areas with a low performance, and a previously good evaluated candidate does not have any influence on the evaluation candidate set or evaluation order. This implies that even if the grid search would evaluate a candidate comparably close to a global optimum, it would completely disregard this chance and not continue the search close to this candidate to potentially find this global optimum.

Grid searches are as well as random searches, classical algorithm choices for hyperparameter optimization because they are easy to implement. In the context of AutoML, a 2-dimensional grid search was utilized in the Weka toolbox [Wit+16] for a simple AutoML approach. There, a joint candidate space was searched with a set of classifiers for model selection on one dimension and the other dimension is used to select the number of components of Partial Least Squares filter, which is used as a pre-processor, and can therefore be considered as a simple model configuration.

(Heuristic) Graph Search

While a random search does it leave to a certain degree to chances which candidates are evaluated, the candidate space will not be adequately covered with a high probability. A grid search will achieve a certain coverage guarantee by default, depending on the number of selected values per dimension. But on the other hand,

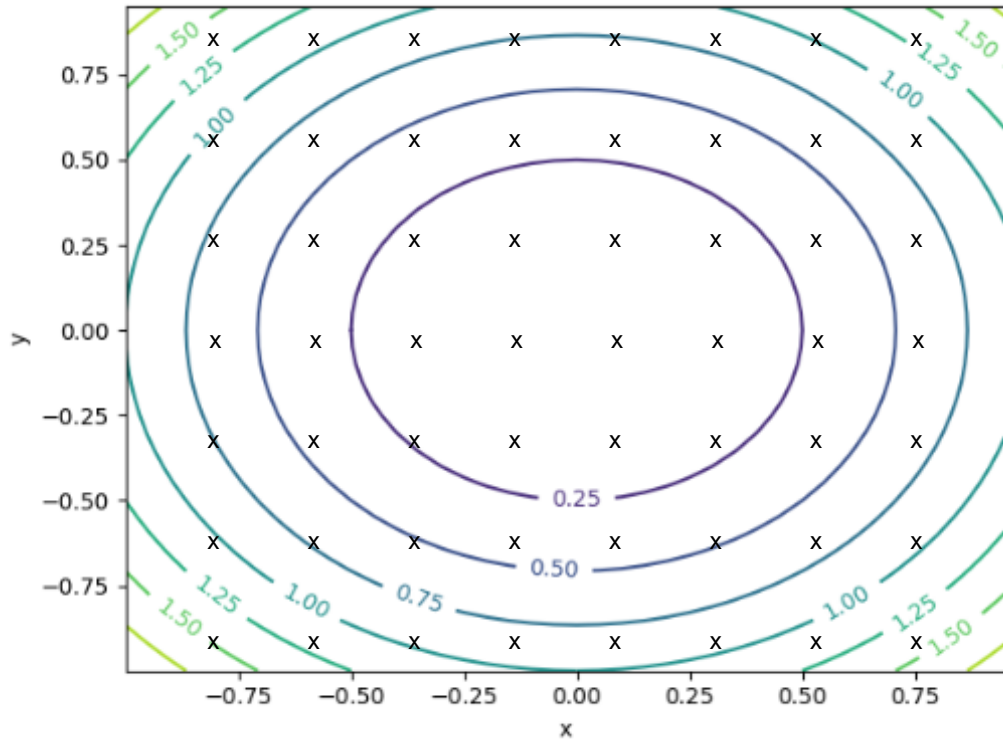


Fig. 2.6.: Illustration of a grid search covering the search space of the running example systematically.

a grid search will not utilize any gathered knowledge about the candidate space and searches it completely homogeneously while at least the second variant of a random search is capable of utilizing knowledge about one prior evaluation.

A graph search, usually in combination with a heuristic for search guidance, tries to achieve a trade-off between the advantages of both. Via the transformation to a graph, it has a discretized view of the optimization space just as a grid search and with such an underlying structure, the coverage of the space can be increased. But by utilizing the knowledge gathered during the search, depending on the search algorithm and the heuristic, often to a greater degree than a random search, it circumvents the homogeneous search of the complete space of grid searches.

As the name suggests, for a graph search the candidate space is depicted as a graph, but how the space is transformed into a representation consisting of nodes and edges is very domain and approach specific and several methods exist. One exemplaric method could be to use a tree as a search graph, where only the leaf nodes are concrete candidate nodes, which can be evaluated. Here, the inner nodes are used to progressively discretize the candidate space dimensions hierarchically by splitting the range into multiple intervals and creating a new child node for each one. This is continued until the intervals of each dimension either only consist of one element or are small enough such that one value out of it can be selected arbitrarily. At this point, a leaf node would be the child node because for each dimension a value is

selected such that the candidate's configuration is complete and could be evaluated. With a given start node and a method to expand this start node incrementally into a bigger graph, a large number of search algorithms exists to search this graph for the best candidate. They mostly differ in the selection method deciding which unexplored node should be visited next. The set of nodes, consisting of the unexplored neighbors of already visited and expanded nodes, is sometimes called a *Search Front* and it is exemplarily illustrated for a search tree in Fig. 2.7.

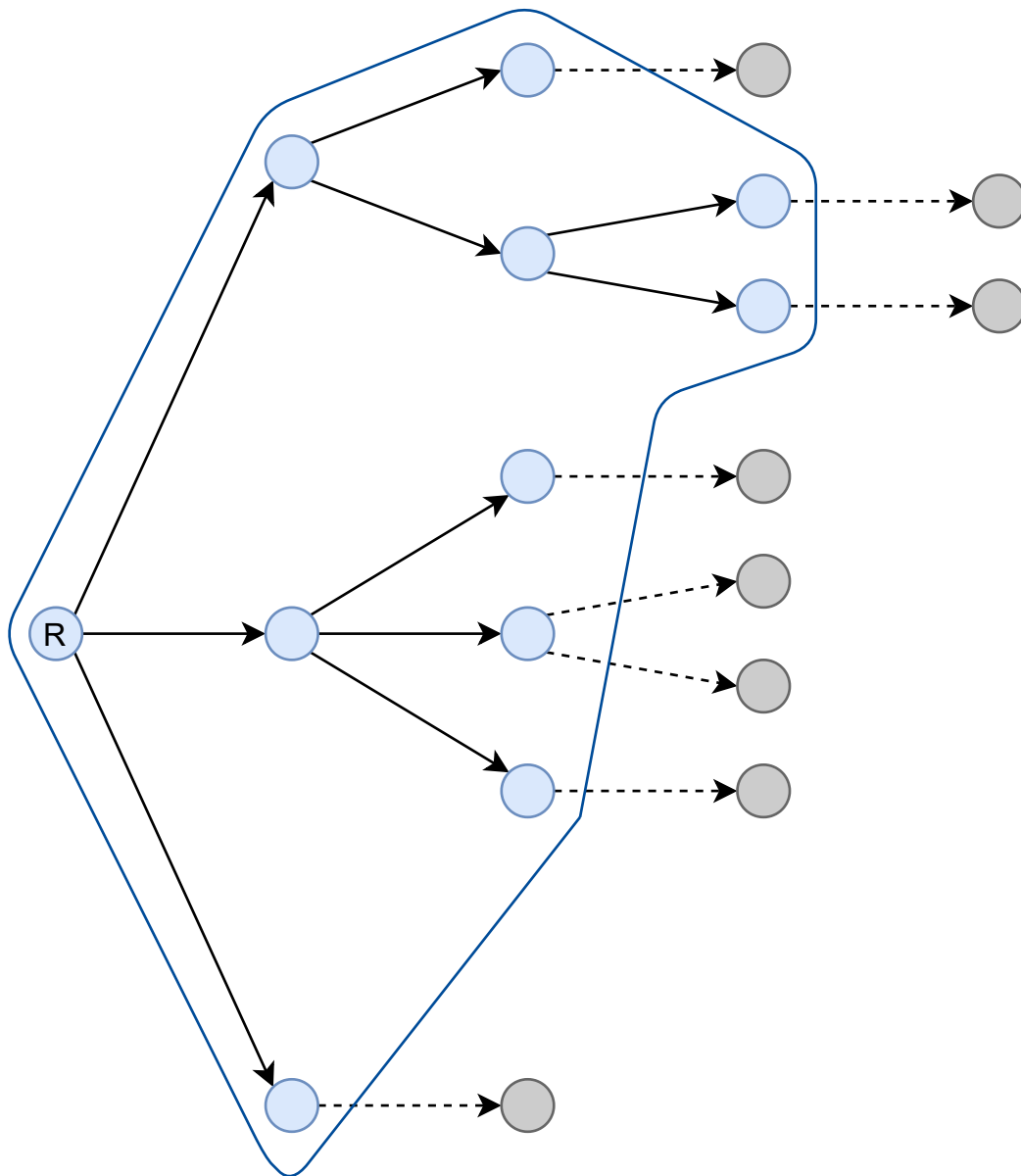


Fig. 2.7.: Search graph of a graph search in the middle of an ongoing search. After a few iterations of a search algorithm beginning at the root node (R), a few nodes are already explored and expanded, here illustrated in blue. The expanded child nodes of the searched area (blue) represent the search front (grey nodes) and one of them will be explored and expanded in the next search iteration.

Some search algorithms select nodes out of the search front based on the point in time, where they were added to the search front, as for example a Depth-First Search or a Breadth-First Search. But to achieve a purposeful search, the node selection should not depend on the insertion order of the search front. Instead, the next node out of the search front should be the node, where the best evaluation score or a path to the best evaluation score can be expected.

If each node would exactly represent one candidate, this is trivial because each node can be evaluated immediately. But for the aforementioned graph creation method, this is more challenging because only leaf nodes can be evaluated and not inner nodes. To assign a score to an inner node without searching the complete sub-graph below it, an estimation method is utilized, usually called a *heuristic*.

For example with some available domain knowledge, if certain discretization intervals or selected values are more promising than others, the heuristic can assign higher scores to the corresponding child nodes representing the interval or value. In the AutoML context, this could be done if for example a certain pipeline component performed outstandingly for previous experiments. But usually, this will not be the case because the relationships between pipeline components to each other and their parameter configuration is too complex and even minor changes can have a big influence on the evaluation score. Therefore, it is hard to score an incomplete pipeline even with pre-existing domain knowledge.

Without a way to deterministically score an inner node, a probabilistic approach is more promising. Here, it can be assumed if enough leaf nodes at the end of the sub-graph below the inner node, which shall be scored, are evaluated, it is possible to score the inner node based on these evaluations with sufficient certainty. Which path to follow to reach a leaf node in the sub-graph is decided randomly. To score something based on a series of random experiments is called a *Monte-Carlo* simulation/experiment and is a common approach to obtain a value, where the exact calculation of the value is either impossible or too complex. This combination of utilizing a heuristic graph search with Monte-Carlo simulations as a heuristic for optimization in the AutoML context was applied to a Best-First Search in the ML-Plan approach [MWH18].

2.3.3 Genetic and Evolutionary Optimization

Genetic and evolutionary optimization methods are inspired by the process of the evolution in nature, as described in Charles Darwins *Natural Selection* theory. Exactly like a biological evolution model for a species, the optimization process is modeled in consecutive generations and each generation consists of a high number of individuals. Ideally, from an optimization point of view, only the individuals that are most adapted to their surroundings should survive and produce offspring and therewith influence

the succeeding generation, which is based on these individuals with minor changes. This is often referred to as *survival of the fittest*.

Transferred to optimization, this would mean that each individual is a possible solution candidate and therefore a generation is a set of solution candidates. Starting with a base generation of usually randomly created individuals, a genetic algorithm follows this pattern:

1. Evaluate each individual of the current generation.
2. Select a subset of the current generation that will be used to create the next generation. Normally, this subset is created as a mix of the highest scoring individuals as well as some randomly selected individuals to prevent focussing on local optima. But other selection methods were also developed.
3. Generate the next generation based on the selected subset. A pre-defined ratio of individuals will be directly copied into the next generation and the remaining individuals will be modified with evolutionary operators to attempt an enhancement.

These steps are usually repeated until either a certain target score is achieved with one individual, the optimization budget is spent, or the scores did not improve over the last n generations.

The aforementioned evolutionary operators work on so called *genetic representations* of candidates, sometimes also called *genomes* or *chromosomes*. In theory, nearly every data structure is suitable for this representation but most often simple arrays or vectors are used, where each cell represents one property of the candidates or alternatively the value of one dimension of the candidate space. For the running example, this would mean an array with length 2, one for the x and one for the y dimension.

Just as in the general principle from biological evolution, these genomes can change by random mutations as well as by recombination, when two individuals produce a new individual. For a mutation, one individual is selected and one or more cells are changed slightly at random and the other values are kept the same. If the number of cells to change is equal to the length of the array or vector, this is the same idea of randomly selecting a candidate out of hypersphere around another candidate already mentioned for random searches. But especially in combination with the recombination operator, the positions of the individuals in the optimization space can be re-organized as new combinations to increase the chance of a high candidate space coverage.

A recombination based on two genomes g_1 and g_2 to produce a new genome g^* is usually performed as a cross-over. For each cell of g^* the value is taken from

either g_1 or g_2 . Depending on the type of cross-over, there are k cross-over points, with $k \in [1 \dots |g^*| - 1]$. The points themselves are usually selected randomly. Each cross-over point marks an index of g^* at which it will be switched from g_1 to g_2 or vice versa from which source genome the next value will be taken from. An example for a two-point cross-over is shown in Fig. 2.8.

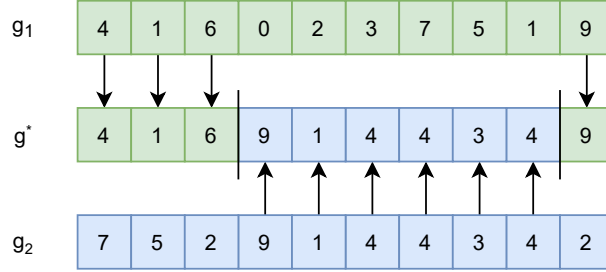


Fig. 2.8.: An example of a two-point cross-over to produce genome g^* from the genomes g_1 and g_2 .

With the random factors during the selection of the individuals for the creation of the next generation as well as during the application of the evolutionary operators, genetic and evolutionary optimization approaches are a stochastic optimization method just as a random search or some graph search algorithms with certain heuristics. Therefore, the chances of finding a high scoring solution, the coverage of the candidate space, as well as the risk of ending in a local optimum are highly dependent on the number of iterations the algorithm performs as well as other configuration parameters as for example the population size.

Additionally to the challenges of handling chances and probabilities emerging from any probabilistic approach, in the context of AutoML there is one more major challenge in applying a genetic algorithm for optimization. A more complicated data structure than arrays or vectors is needed to represent a pipeline configuration as a single genome because usually the pipelines do not have a maximal length and can be composed as an arbitrary graph or tree, which cannot be represented in an array with a maximal length. Even if a suitable data structure is found, the mutation and cross-over implementations have to be made very cautiously, because representations of pipelines can be very error-prone. Already small changes to one representation value can lead to an invalid pipeline construction that cannot be instantiated.

Two state-of-the-art approaches tackling these challenges and applying genetic algorithms as optimizers for AutoML problems are *TPOT* [OM16], using expression trees as a data structure for genetic programming, and *RECIPE* [Sá+17], using grammar-based genetic programming. Both utilize specialized genetic operators, that acknowledge the complex structure of individuals representing AutoML candidates, such that mutations and cross-overs produce fewer invalid pipelines.

2.3.4 Bayesian Optimization

Similar to some heuristic graph searches and genetic algorithm implementations, Bayesian optimization tries to utilize the knowledge from previous candidate evaluations as much as possible. The goal is to select the most promising candidate for the next evaluation and therefore minimize the overall number of evaluations until a near-optimal score is reached. Since each evaluation in the context of AutoML comes with assembling a candidate pipeline and usually performing a cross-validation of the pipeline on the dataset, each evaluation consumes an often not insignificant portion of the optimization budget. Therewith, Bayesian Optimization is very well suited for black box optimization in the context of AutoML.

The notation of a *most promising* candidate could be interpreted in different ways. For example, a candidate near the current optimum, which has a high probability of having a comparable or even better score, or a candidate in a completely unevaluated region of the candidate space, where therefore an optimum could lie unnoticed. But in the case of Bayesian Optimization, this notation of most promising is more sophisticated.

In general, the goal is to create a surrogate function model that replicates the unknown target function as closely as possible with the aid of Bayesian statistics. To model the Bayesian surrogate, a common choice is a regression via a Gaussian Process. Detailed information about Gaussian Processes can be found for example in [RW06].

This surrogate is based on the results of several evaluations as samples and will become more precise with each new sample, which is usually the next most promising candidate. For the actual selection of the next most promising candidate sample, a so called *Acquisition Function* is used and the next value will be at an optimum of this acquisition function. The value of such an acquisition function for a candidate is based on two factors:

1. The estimated score of the candidate, which is based on the modeled Bayesian surrogate
2. The size of the credible interval (basically the concept of confidence intervals in the context of Bayesian statistics) of the Bayesian surrogate model at the position of the candidate

The size of the credible interval is especially important because if at a point this interval is very broad, the chances are higher that the value of the target function can differ widely from the surrogate at this position. This would indicate a higher uncertainty of the model. Of course, the value differ can in both directions, but

this can be examined with an additional sample at this position. Therefore, if the estimated performance already indicates an optimum and additionally the credible interval is broad (i.e. an optimum of the acquisition function based on these two factors), the value of the target function could be even more optimal at this position and it is the most promising candidate to be drawn as the next sample.

For candidates that were already evaluated, this acquisition score is set to zero because the target score at this position is already known and the surrogate can be certain at this position. A graphical intuition for the concept of acquisition functions is given in Fig. 2.9, where Bayesian optimization on one dimension of the running example is illustrated.

An exemplaric algorithm for Bayesian Optimization is given by Frazier [Fra18] with the following pseudo-code in Algorithm 1.

Algorithm 1: Basic pseudo-code for Bayesian optimization

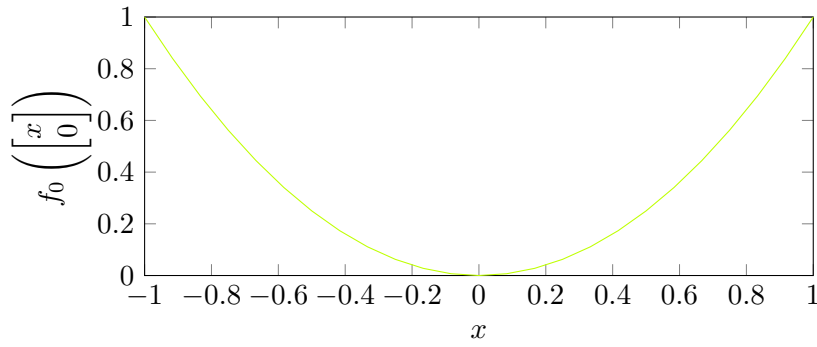
```

1 Place a Gaussian process prior on  $f$ 
2 Observe  $f$  at  $n_0$  points according to an initial space-filling experimental design
3 Set  $n \leftarrow n_0$ 
4 while  $n \leq N$  do
5   | Update posterior probability distribution on  $f$  using all data
6   | Let  $x_n$  be a maximizer of the acquisition function over  $x$ , where the
   |   acquisition function is computed using the current posterior distribution
7   | Observe  $y_n \leftarrow f(x_n)$ 
8   | Increment  $n$ 
9 end
10 Return a solution: either the point with the largest  $f(x)$ , or the point with the
    largest posterior mean

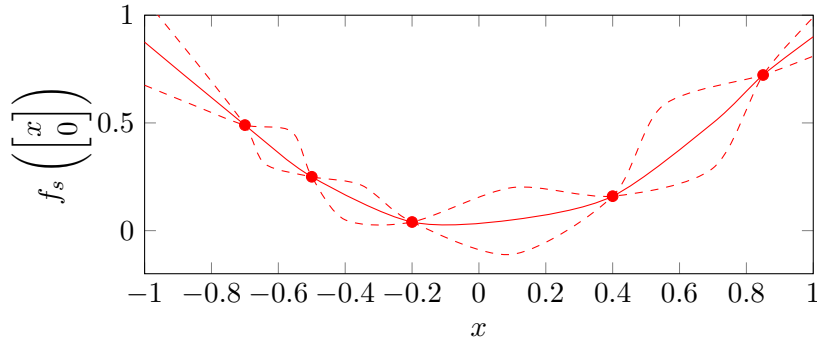
```

The functionality of this pseudo-code as well as some technical terms are explained in the following line by line:

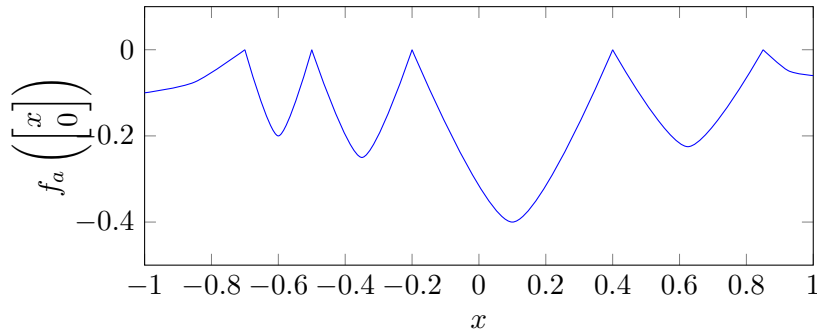
1. A probability distribution, for example a Normal distribution is selected as an initial prior distribution f_0 to further construct the surrogate function with.
2. Since this initial f_0 is not very meaningful regarding the target function, a few candidates will be evaluated before the actual optimization loop starts. These candidates are selected to be a space-filling design, which means to have the best possible coverage of the candidate space. Therewith, the credible intervals between the initially observed points can be kept as small as possible before the acquisition function, which uses these intervals, is used. An example of a simple space-filling experiment design is to choose candidates uniformly distributed in the candidate space.



(a) Slice of the target function f_0 at $y = 0$.



(b) Estimated surrogate function f_s modeled after 5 samples. The evaluated samples are shown as red dots, the surrogate function as a red line, and the credible intervals as the two red dashed lines. Important note: The values of the surrogate function and the credible intervals are not calculated but are only rough estimates to give an intuition of the method. The exact values can vary significantly based on the applied kernel of the Gaussian Process.



(c) Acquisition function f_a based on f_s and its credible intervals after the 5 drawn samples. As in (b), the values are only estimates and not explicitly calculated. The minimum of f_a at $x = 0.1$ would be the best choice for the sixth sample. There the expected value of the surrogate regression is already low and the credible interval is additionally broad.

Fig. 2.9.: Simple illustration of acquisition functions for Bayesian optimization. It is applied to the running example of minimizing $f_0 \left(\begin{bmatrix} x \\ y \end{bmatrix} \right) = x^2 + y^2$. For simplicity will the Bayesian optimization only be used to minimize on the x -dimension and y will be set to 0, but in theory can this approach be extended to any number of dimensions. (a) shows a slice of the target function in green. (b) shows in red the already evaluated samples, the estimated surrogate function model, and the credible intervals. (c) shows in blue the acquisition function based on the values in (b).

3. Since this implementation uses the number of observations of f for a given point, i.e. candidate evaluations, as an optimization budget, the variable n , which keeps track of the spent budget, is set to n_0 , because this is the number of initial evaluations.
4. Here starts the main optimization loop. As long as the optimization budget is not spent and therefore $n \leq N$, further candidates can be selected and evaluated.
5. Let d_n be all evaluated points from previous iterations or from n_0 , the surrogate function model of this iteration f_n will be updated based on d_n . Therefore, a new surrogate function posterior probability distribution is calculated by updating the chosen distribution with the previous posterior distribution f_{n-1} as a new prior, i.e. $f_n := f(x)|f_{n-1}$.
6. x_n is selected to be the candidate of this iteration for evaluation. It is chosen by searching the acquisition function, which was updated with the data from prior iterations, for an optimum.
7. Get an evaluation score y_n for the selected candidate of this iteration x_n by observing $f(x_n)$, thus having one more data point to base to posterior probability distribution on and to construct a more precise surrogate function f_{n+1} in the next iteration.
8. Increment n because another candidate evaluation was performed and therefore the remaining optimization budget decreased.
9. The optimization budget is spent and the main optimization loop will terminate.
10. After the main optimization loop finishes, the algorithm will terminate and return a candidate as the result. It will select the best candidate among the evaluated points so far and compare it to the point with the largest posterior mean, i.e. the point where the global optimum, based on the overall constructed surrogate function f_N , is assumed. If this point is assumed to be better than the best evaluated candidate it will be returned and the best evaluated candidate otherwise.

A state-of-the-art implementation of Bayesian optimization for algorithm configuration problems is *SMAC* [HHL11], an abbreviation for Sequential Model-Based Optimization for General Algorithm Configuration. It is utilized as an optimization approach for a joint model selection and model configuration AutoML opti-

mization space by several approaches, including *Auto-WEKA* [Tho+13] and *auto-sklearn* [Feu+15].

2.3.5 No-Free-Lunch Theorem

The state-of-the-art AutoML approaches, presented with their corresponding black box optimization method, have one thing in common: They all apply a single optimization algorithm to find the best candidate from an optimization space, which is a joint model selection and model configuration space. While this is a valid approach and will find an optimal or near optimal solution for many AutoML problem instances, it is theoretically impossible that the best solution can be found with a single optimization method for every AutoML problem instance, if it is a setting with constraints as for instance an optimization budget.

This can be deduced from the *No-free-lunch Theorems*, which were proven for optimization problems by Wolpert and Macready [WM97]. In these theorems, it is stated that when an optimization algorithm performs superior for one problem or class of problems, it has to pay for this by performing inferior for other problems. Of course, this is only the case for optimizing with an optimization budget or other constraints because the most optimization algorithms can find the best solution if they have an unlimited budget, even if it is just by evaluating each possible candidate in a brute force manner.

For the context of the AutoML setting, the notion of a problem class could for example be transferred to the data, which is given as an input. Therefore, in this setting with a limited budget of time or resources, one single optimization algorithm may find the best pipelines and configurations for some datasets but cannot achieve this for all datasets. Thus, if the so far presented optimization approaches would be applied to an AutoML problem, they would be inherently incapable of being the superior approach for all datasets. With a given dataset D_1 a genetic algorithm may find the best solution, while a Bayesian optimization method outperforms the same genetic algorithm for another dataset D_2 .

It is additionally possible that the quality of an optimizer does not only depend on the input dataset, i.e. the AutoML problem class instance, but additionally on the concrete optimization space it operates on and the optimization budget. These three components could be used to construct a specific problem instance for a dataset problem class.

If this optimization space is modified and thus a new problem instance p_2 is created from the original problem instance p_1 , this can have a huge influence on the optimizer qualities and the optimization method that was the most successful one for p_1 could now theoretically be the worst one for p_2 .

Such a modification of the optimization space can be created if, for instance, the model selection part is already finished and the remaining model configuration step induces a new optimization space, which is now the problem of optimizing the parameters of a concrete pipeline. Here, different optimization spaces can be created out of one initial space by pre-selecting a pipeline and having the pipeline hyperparameters define the remaining optimization space.

Now, different optimizers can differ in quality here as well solely based on the selected pipeline components. For example, with a given dataset, optimizer *A* could find good configurations for a pipeline consisting of a Principal Component Analysis followed by a Decision Tree, while optimizer *B* yields better results in the case of a single Support Vector Machine. The same transformations of the optimization space can of course be done by only considering the model selection instead and not the model configuration.

Since one optimization method will only be the best one for a limited set of problem classes, it can be beneficial to use more than one optimization method and utilize a different optimizer for different transformations of the optimization space. There-with, the the set of problem classes, where a combination of these optimizers can now be superior, is extended.

In the next section, a selection of approaches is presented, which use such transformations of the optimization space and more than one optimizer, to utilize this extension of problem classes in the AutoML context, where the theoretical superiority can be achieved by multiple optimizers.

2.4 Related Work

The first presented approach that uses different optimizers for different transformed AutoML optimization spaces is *ReinBo* [SLB19]. Here, the overall optimization space, including model selection and model configuration, is transformed into a solely model selection space at first. For practicability, each pipeline in this approach can only consist of up to three components: One for data pre-processing, one for feature engineering, and one as the machine learning algorithm. The model selection is performed in three stages, in each of which one of the three component types is selected successively. It is also possible to not select any component for a pre-processing or feature engineering, which could be represented by a symbol as for example "NA". A simple example for a small optimization space for such a model selection method would therefore consist of three sets:

- A set S_p with pre-processing algorithms: For example $S_p = \{\text{Min-Max Scaling, One-Hot Encoding, NA}\}$

- A set S_f with feature engineering methods: For example $S_f = \{ \text{PCA, Variant Threshold Filtering, NA} \}$
- A set S_m with machine learning algorithms: For example $S_m = \{ \text{SVM, Decision Tree} \}$

Such a model selection space as in ReinBo can be imagined as a tree, where the root is an empty pipeline that has none of the three component types selected. This root node has a child node for each $s_p \in S_p$, including the representation symbol "NA" for not selecting any pre-processing component. Therefore, the child node which represents one s_p means that the pipeline from this subtree uses s_p as a pre-processor. Accordingly, such a child node for an $s_p \in S_p$ has a child node itself for each $s_f \in S_f$ and ultimately each s_f node has a child node as well for each $s_m \in S_m$, which now is a leaf node since the pipeline components are completely constructed. An illustration of the previous examples optimization space can be seen in Fig. 2.10.

Because of the tree-shaped structure of this optimization space, a hierarchical reinforcement learning approach is trained during the optimization. With learning a policy for suitable pipeline component choices, it can act as an optimizer for model selection.

After the reinforcement learning algorithm has selected up to three pipeline components, the possible parametrization of each component is examined. From the possible parametrization of each selected component, a model configuration space can be induced. Therewith, based on a completed model selection, the overall optimization space is once more transformed, but now to a space representing possible model configurations for the constructed pipeline. Each leaf node of the tree structure has such a small model configuration space.

Once the reinforcement learning algorithm performed the action of selecting a component from the last hierarchical layer and it reaches such a leaf node, Bayesian optimization is conducted in the model configuration space of this leaf. The evaluation result of the best candidate of this optimization run is given as a reward to the reinforcement learning algorithm and it can update its policy. Afterwards, in the next iteration, the reinforcement learning algorithm starts once again at the root node and has to select pipeline components based on the policy and this is repeated until the optimization budget is spent.

Because Bayesian optimization is not very effective for high dimensional spaces, it is a suitable adjustment for an AutoML approach to not run a Bayesian optimization on the complete candidate space but on a smaller transformed space. Instead of Bayesian optimization, the model selection is performed by another optimizer and only a small portion of the model configuration space is the input for a Bayesian optimization run, which includes only the hyperparameters of the selected components. The dimensionality is reduced significantly and Bayesian optimization will be

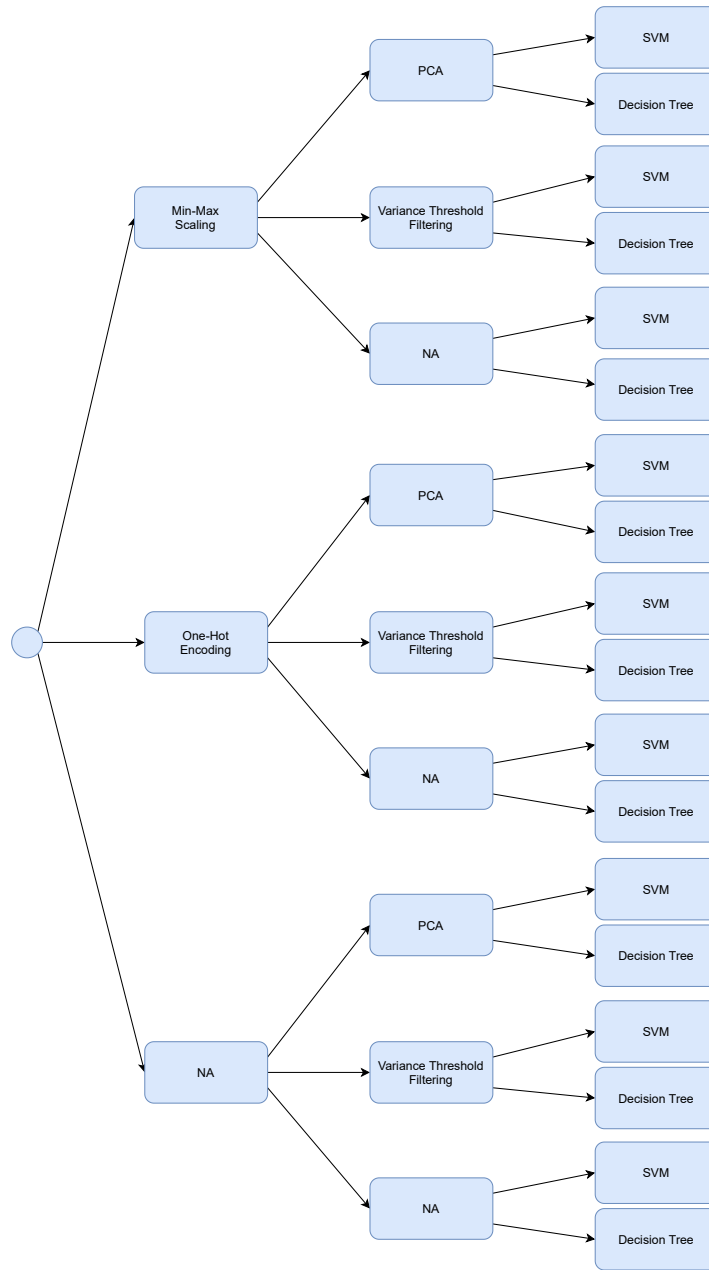


Fig. 2.10.: Simple visualization of the underlying concepts of the three layer ReinBo model selection candidate space.

more effective.

But there is still a big amount of problem classes where the Bayesian optimization method itself can not be the suitable optimization algorithm choice. Therewith is the set of problem classes, where this approach can be superior, not extended drastically by the improvements regarding the reduction of dimensions of the Bayesian optimization space and the inclusion of a reinforcement learning algorithm as an additional optimizer.

The advantages of performing the AutoML optimization in several phases on transformed optimization spaces was further examined by Quemy [Que19]. Besides the aforementioned opportunity to compensate one less suitable optimization approach with another possibly better suited optimizer, in this publication additionally was stated, that if the optimization space is transformed into two or more smaller sub-spaces, the overall optimization process can be sped up. There, a two-stage optimization transformation was applied, alike ReinBo separating model configuration and model selection. Although, this approach is not as limited to a mandatory pipeline topology as ReinBo, with the constraint of three components at most and only one of each type, and can construct pipelines in a more unrestricted fashion as a directed acyclic graph. To create an optimization space of all valid pipeline graphs, all pipeline components have specifications of their input and output datatypes and can only be connected to another component node, if the output types of one node match the input types of another node.

In the formal descriptions of this approach, no specific optimization algorithm is proposed. For this framework, any optimization algorithm could be applied for either phase or the same one for both phases as well. Instead, Quemy focusses on the allocation of the optimization budget, in their case a time budget T , between the two phases. The following allocation policies were defined:

- *Split policy*: T is split into pre-defined phase allocations T_1 and T_2 . T_1 is completely spent onto model selection and afterwards T_2 is spent as a whole as well.
- *Iterative policy*: A very short time period $t \ll T$ is set and it will be alternated iteratively between the two phases, where each phase has t as a budget in one iteration, until T is spent.
- *Adaptive policy*: This policy is an extension of the Iterative policy but here the t is not fix but is adaptive to the achievements of the optimization in each phase. Therefore there is a t_1 and a t_2 for the corresponding phase. If in one iteration the evaluation score increases during the optimization of the active phase i , t_i is doubled. In return, if the phase i did not increase the evaluation score during its last two active iterations, t_i is halved.
- *Joint policy*: Here, no separation into different optimization phases takes place and model selection and model configuration is performed in a joint search space. Hence, it can spend T completely.

Monte-Carlo simulations as a heuristic for a graph search, as explained in section 2.3.2, are the basis of a graph search algorithm named *Monte-Carlo tree search*. In this search, the task of selecting the next child node of one node for expansion is

viewed as selecting an arm of a Multi-Armed Bandit problem. With this approach, a balance between exploring more unevaluated areas of the tree and exploiting subtrees that have already shown promising scores can be achieved. This balance is achieved via a heuristic scoring of nodes via Monte-Carlo simulations and a node selection strategy based on these scores that is formalized as *UCT* (Upper Confidence bound applied to Trees) [KS06]. This combination of *UCT* and Monte-Carlo simulations is then often referred to as a Monte-Carlo tree search (MCTS). MCTS is exemplified in more detail in section 3.2.4.

One example of a heuristic graph search based AutoML approach is called *Mosaic* [RSS19] and it leverages MCTS employed in a multi-stage optimization methodology as well. In *Mosaic*, the tree that is the subject of the MCTS is modeled as a transformation of the optimization space into a model selection space. The first k layers of the search tree are representing k possible types of pipeline components, for example pre-processing methods or machine learning algorithms. When a component is selected in layer $i \in [1, k]$ of the tree, it will also be the component with position i in the resulting linear pipeline. All resulting pipelines are only linear and not generalized to any form of a directed acyclic graph as in the earlier discussed related work. In each of the first k layers, one component of each type is selected. Therewith, this approach has a comparable tree structure to ReinBo but is more flexible regarding the length of the pipeline.

After the model selection for a pipeline is concluded, the k components of the pipeline need a configuration. Thus, the continuous model configuration space that remains of the initial optimization space after the discrete model selection space has been covered with the search tree, is transformed into a sole model configuration space for the selected pipeline, which is embedded in the corresponding leaf node of the selected pipeline at the bottom of the k model selection layers. Similar to ReinBo, a surrogate model f_s will be constructed in each leaf node. It is supposed to predict the expected evaluation score for any point of the optimization space, if the pipeline that consists of the components represented by the current leaf is constructed with the parametrization of this point. The construction of f_s and the optimization is based on Bayesian optimization.

Mosaic and ReinBo have two remaining drawbacks:

- They can only create pipelines of a fixed length and are without further extensions not capable of constructing more sophisticated non-linear pipelines, restricting the final pipeline potential for some datasets.
- Hyperparameter optimization in the second phase after the model selection is only performed with a single optimization method, namely Bayesian optimization, and is therefore limited by the capabilities of this method. But as outlined before with the aid of the No-Free-Lunch theorems, it is possible to improve the

set of problem classes, for which an AutoML approach is superior, by utilizing more than one optimization algorithm, if it is possible to automatically detect and exploit the most suitable optimizer for a problem class.

The approach of this thesis is similar to the principle of Mosaic and ReinBo but tries to overcome these two drawbacks. In the following chapter, the functional principles of the approach of this thesis are explained in detail as well as the strategies for both improvements.

AutoML via an Ensemble of Optimizers

As outlined in the previous chapter, in theory, any optimization approach under a budget is limited to a set of problem classes, where it can find a solution that is better than any solution another optimization approach could find for the same problem class, i.e. where the approach is superior. For an actual use-case, as for instance AutoML, it would be necessary to either have significant domain knowledge or to have conducted a broad empirical evaluation to select the optimizer that has a high chance of performing superior for a newly encountered problem class. This can be avoided if during the optimization process a set of optimizers would be evaluated in the context of the presented optimization problem and the aptest one would be applied automatically.

Furthermore, it was argued, how a separation of the optimization space into model selection and model configuration spaces by optimization space transformations could be beneficial. The anticipated advantages are a speed-up at first because the dimensionality of the input space for an optimizer can be reduced. Secondly, the best suited optimizer for the optimization space of the model configuration of a constructed pipeline can be selected independently for each different result of the model selection steps.

Although this approach could be generalized to any form of algorithm selection and hyperparameter configuration problem, this thesis focusses on the AutoML use-case and therewith has to address well suited machine learning pipelines as an expected output for a wide variety of datasets as an input. Thus, the constructed pipelines should be able to be as sophisticated as necessary to suit the complexity of any possible input dataset. Therewith, the approach has the additional requirement of preventing overly stringent limitations of the pipeline topology such as a fixed length or solely linear compositions.

In the following chapter, such a method is presented and applied to the AutoML use-case. This approach is explained in the following steps in separate sections:

1. A short overview how model selection and configuration spaces can be separated for this approach

2. The structure of the model selection optimization space, with a focus on how the optimization in this space is performed to enable the selection of different optimizers for model configuration out of an embedded optimizer ensemble
3. The structure of the model configuration optimization space and which benefits can be utilized from using and re-using the different optimizers of the ensemble

3.1 Separation of Model Selection and Configuration

A usual AutoML optimization space is a joint space for model selection and model configuration consisting of a set of components, which can be used to construct a machine learning pipeline, and based on the selected components, a valid configuration has to be created. Since in the joint optimization space the selected components are not known beforehand, the dimensionality of this space has to be selected for a fixed parametrization size. Hence, the model configuration can only create parameter configurations with a pre-defined size as a constraint and the set of pipeline components for model selection can only contain components whose configuration does not exceed this size.

If the model selection and the model configuration steps are performed sequentially or alternatingly on partitioned optimization spaces, the dimensionality of the model configuration space can depend on the outcome of the model selection step and have the necessary dimensionality for the constructed pipeline's parametrization.

Similar to ReinBo and Mosaic, the model selection space of this approach is represented by a tree structure. After the optimization method reaches a leaf node and has therewith a complete and valid pipeline structure, the size of the required configuration becomes clear. Any leaf node is for this reason the connection point and foundation for a model configuration space and the configuration requirements can be deduced. Thus, dimensionality and structure of the optimization space for the model configuration can be determined from the configuration requirements of the pipeline represented by this leaf node.

3.2 Model Selection with MCTS

While the overall goal of the model selection step is to construct a pipeline out of a set of pipeline components, in this approach two additional sub-goals come as supplementary requirements:

1. The constructed pipeline should be able to be constructed more flexible and more sophisticated than in ReinBo and Mosaic, i.e. have an unrestricted length and no constraints regarding the topology.
2. The model selection step should be able to evaluate the different optimization methods and their performance regarding the input dataset or even regarding the model configuration for specific pipeline components in the context of the dataset. With such an approach, the most suitable optimizer can be detected and exploited, which will be the key concept in this approach for creating an optimizer ensemble.

In the following, both sub-goals are addressed. The next two sub-sections tackle the first sub-goal by modeling the model selection process as a graph search problem at first and describing this search graph in more detail afterward. Thereafter, in the third sub-section, a formalization of the ensemble concept for multiple optimizers is given in the form of a Multi-Armed Bandit problem. Finally, in the fourth sub-section, it is illustrated how both sub-goals can be achieved during the model selection by performing an MCTS.

3.2.1 Model Selection as a Graph Search Problem

A machine learning pipeline can vary significantly in structure and complexity. For example, using a decision tree as the only component is a valid pipeline as well as many components in a more sophisticated topology such as the pipeline in Fig. 3.1. Depending on the concrete input dataset, both levels of pipeline complexity could be the optimal output of the model selection step.

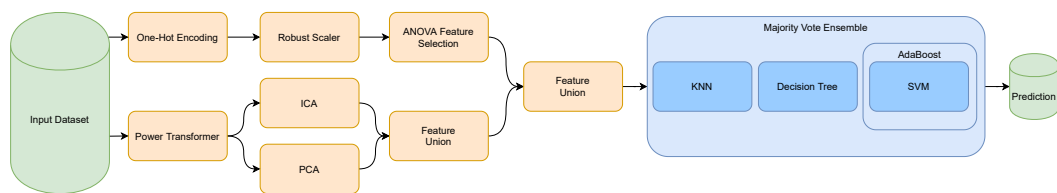


Fig. 3.1.: An example of a complex machine learning pipeline with a higher length and a non-linear topology. Input dataset and output prediction data are shown in green, pre-processing components in orange and the machine learning components constructing the machine learning algorithm in blue.

In their current implementation, ReinBo and Mosaic solve their model selection tasks by defining a fixed pipeline length and creating a model selection tree containing all combinatorially possible combinations with this length and the given component set. Although it would be possible to increase the maximal pipeline length of the two

approaches with minor modifications, the concept, where each layer represents one respective open component slot of the result pipeline, does not allow more complex pipeline topologies than linear.

Other approaches, as for example ML-Plan, TPOT, or RECIPE utilize respectively hierarchical task network planning (*HTN planning*) combined with a best-first search, or expression trees and formal grammars together with genetic programming, to achieve a more unconstrained model selection optimization space.

The optimizer ensembles approach of this thesis try to achieve two aspects. First, the optimizers of the ensemble should share the gathered knowledge with each other and utilize it. Second, during the ongoing AutoML workflow, the best optimizer for the given problem instance should be identified and exploited. This second aspect will be controlled by the model selection. For this control function, an MCTS is a suitable choice (this is reasoned in section 3.2.4), which is a heuristic graph search algorithm like the best-first search as in ML-Plan. The search of this ensemble approach will take this HTN planning approach as a foundation to construct the model selection space in the form of a tree.

As the name suggests, HTN planning is based around the notion of tasks and usually, one task is given as the planning problem input. The goal of the planning algorithms is now to construct a plan to solve this input task. In the case of HTN planning, tasks can either be primitive tasks or compound task.

Primitive tasks are simple enough to be directly achievable without the need for further planning or other calculations. Compound tasks, on the other hand, are not directly realizable and it is necessary to further decompose the task. Usually, the input task is a compound task, because otherwise there would be no need for any planing. This planning is done for HTN planning in the form of decompositions of compound tasks.

Each compound task can be decomposed into one or most commonly two or more sub-tasks. Such sub-tasks of a compound task can be primitive tasks, compound tasks, or a mixture of both. If a compound task can be decomposed into solely primitive tasks, this task is directly solvable via the actions of the primitive tasks. But if the decomposition of the compound task consists of at least one compound task, this child compound task needs to be decomposed recursively as well until it becomes solvable and this solution can be used to solve the original compound task. Depending on the planning domain, there are decomposition rules where a compound task has one or more possible decompositions.

For example in an AutoML setting, one compound task could be something like a formalized form of *"Construct a machine learning pipeline out of a set of components"*. In the same context, an example for a primitive task could be *"Use a decision tree as a learning algorithm for the pipeline"* as this task can be directly transformed into a pipeline construction command. The exemplaric AutoML compound task *"Construct*

a machine learning pipeline out of a set of components" could be decomposed for example into the to smaller compound tasks "Construct a pre-processing pipeline out of a set of pre-processing components" and "Construct a learning algorithm out of a set of machine learning components".

With this decomposition of compound tasks into different sub-tasks, the hierarchical aspect of HTN planning is added. A plan as the solution of the planning problem is therefore a task-tree, where each leaf node is a primitive task and each inner node is a compound task. To create a search tree for an HTN planning problem, solution states, i.e. solved or incomplete task-trees, as well as planning operations to solve an incomplete task-tree, i.e. selecting and applying decomposition rules, need to be represented with nodes and edges. Solved or incomplete tasks-trees are also called plans or in the incomplete case plan prefixes.

For the search tree creation is one common possibility to have each node represent a task-tree and each outgoing edge is a decomposition rule that applies to the task-tree of the node. These edges are therefore connected to nodes, where the represented task-tree is the result of applying this decomposition rule on the previous task-tree. In the case of a bigger HTN problem, an unsolved task-tree will contain a high number of undecomposed compound nodes and therefore an immense number of decomposition rules would be applicable. To reduce the maximum degree of the graph and therefore the memory and runtime complexity of most search algorithms, there should not be an outgoing edge for the applicable decomposition rules of each undecomposed compound task. A simplification is to create an ordering for the undecomposed compound task of the node's task-tree and only create outgoing edges for decomposition rules, which apply to the first compound task of this ordering. The representation of one simple incomplete task-tree and two applicable decomposition rules for the first compound task of an ordering is illustrated in Fig. 3.2.

3.2.2 Description of the Search Space Graph

The examples of AutoML HTN tasks in section 3.2.1 are given as instructions in plain English but this cannot be used in an algorithm and therefore a formalization is necessary. With a suitable formalization and corresponding data model, the HTN planning domain is adjustable and expandable for the exact use-case. Based on these adjustments and expansions, it is possible to modify valid pipeline topologies and the set of pipeline components for pipeline construction for the AutoML use-case.

While attempting pipeline constructions, it is not possible to create arbitrary pipeline graphs because components often expect one specific type of input that can only be provided from other components, which have this type as their output type. Additionally, sometimes the selection of some components implies the requirement of selecting certain other components. For example, if an ensemble method was

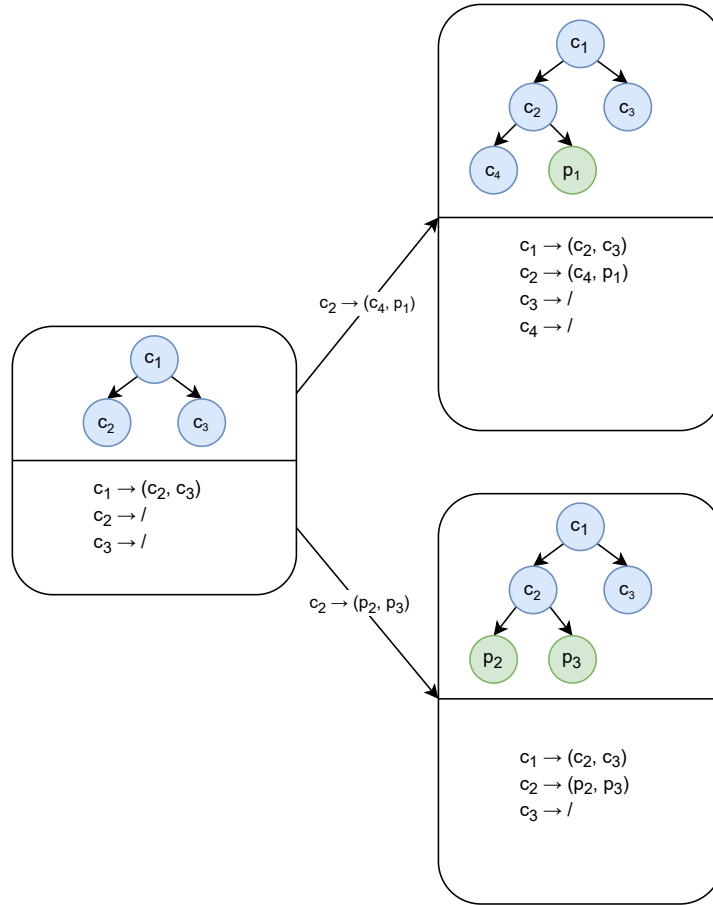


Fig. 3.2.: An example of a partial HTN planning search tree for a simple planning problem. Each node is illustrated with the task-tree for the partially solved plan up to this point in its upper half, where compound task nodes are blue and primitive task nodes are green. In the lower half are all involved compound tasks listed in the utilized ordering. If any decomposition rule was already applied for a compound task, it is listed here as well. The first node on the left has already utilized a decomposition for compound task c_1 into c_2 and c_3 . In this ordering the first undecomposed compound task is c_2 and since two decomposition rules $c_2 \rightarrow (c_4, p_1)$ and $c_2 \rightarrow (p_2, p_3)$ are applicable, the node of this task-tree has two outgoing edges to nodes, where the corresponding decomposition rule was applied to the represented task-tree. After applying the second decomposition rule, c_2 has solely primitive tasks as sub-tasks and can therefore be marked as solved. In the case of the first decomposition rule, c_4 needs to be decomposed and solved before c_2 would be solved.

selected as a learning algorithm, the components that are used to be the predictors of the ensemble must be learning algorithm components and cannot be pre-processing components.

To incorporate such constraints into tasks and decomposition rules, ML-Plan utilizes a simple type system in the form of required and provided interfaces as an additional

abstraction layer above the tasks. From this interface abstraction layer point of view, each task has one or more type definitions that represent certain properties of the solution to the associated task, i.e. they are provided by solving this task and are therefore called provided interfaces. For example, the primitive task of using a decision tree as well as the compound task of creating an ensemble learning algorithm out of several other learning algorithms would both offer a solution in the form of a machine learning algorithm. Both of them could have something like `Classifier` or `Learner` as their solution type and therewith as the types of interfaces they provide.

On the other hand, compound tasks are only solvable if they get solutions of certain types as the results of the sub-tasks they are decomposed into. For example, a Stacking classifier can only be based on other classifiers, i.e. solutions with types like `Classifier`. Thus, each compound task has one or more required interfaces, which represent the necessary solution types of the tasks the compound task could be decomposed into.

Decomposition rules are now simple matching rules, i.e. which required interfaces can be satisfied with which provided interface. Of course, as the easiest form of matching rule, it is possible that these interface types must be identical. A compound task with a single required interface of type a can only be decomposed into another task, which has a as one of its provided interface types. With such definitions of required and provided interface types for each task, it is straightforward to incorporate construction constraints into the planning problem without the need to write an extensive list of decomposition rules. These constraints can be pipeline topology specific constraints like splits into two sub-pipelines and uniting such sub-pipelines again, or machine learning specific constraints as for example creating ensembles out of classifiers.

The overall pipeline structure, for example, Pre-Processing + Learning Algorithm or Pre-Processing + Learning Algorithm + Post-Processing, can be encoded in the decomposition rules for input tasks. Every actual pipeline component, which shall be included in the pipeline construction of a solved task-tree, will be represented as primitive tasks.

Non-linear topologies like creating parallel pre-processing sub-pipelines can be for example achieved with a Feature Union pipeline component, which has one or more types of pre-processing components as required interfaces. Pipelines with a dynamic length can be achieved in a similar manner to generating sequences with an unlimited length in a regular grammar, where a pipeline can either be decomposed into just a component or a component and a pipeline.

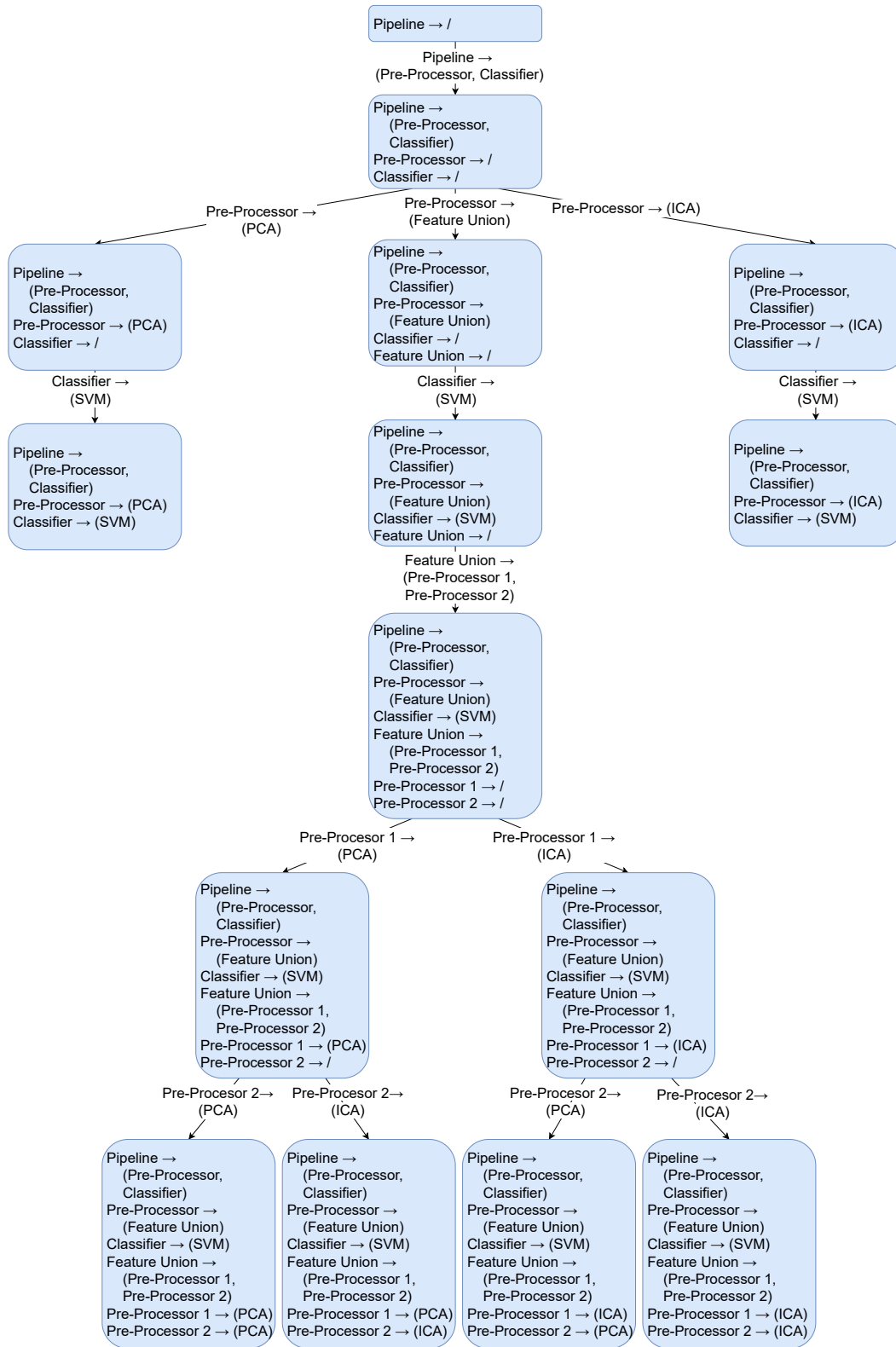


Fig. 3.3.: A complete search graph of an AutoML problem visualized in the HTN context. The primitive tasks of this problem are *PCA*, *ICA* and *SVM*. All available compound tasks are *Pipeline*, *Pre-Processor*, *Classifier*, *Feature Union*, *Pre-Processor 1* and *Pre-Processor 2*.

3.2.3 Multiple Optimization Algorithms as a Multi-Armed Bandit Problem

All nodes, where every compound task is solved, are therefore leaf nodes and all of them together will be grouped in N^* . Each $n^* \in N^*$ represents a completely defined pipeline $p_{n^*} \in P$ with the internal task-tree of n^* , where P is the set of all possible pipelines. For this p_{n^*} , the parameters each included component requires can be aggregated and therewith a model configuration optimization space of the parametrization of p_{n^*} can be deduced.

The approach of this thesis requires a set of optimization algorithms, or alternatively differently configured variants of one optimization algorithm, as an input. This forms the optimizer algorithms set $A = \{a_1, \dots, a_k\}$. To enable the actual model configuration with one of the available optimization algorithms, one additional node for each $a_i \in A$ will be attached as a child node to each model selection leaf node $n_j^* \in N^*$. The node that is attached to n_j^* as the child node for optimization algorithm a_i is written with the notation $n_{n_j^*}^{a_i}$. All of these additional nodes for optimization that are child nodes of the model selection leaf nodes N^* are in the node set $N^A = n_{n_1^*}^{a_1}, \dots, n_{n_l^*}^{a_k}$. A node with an optimization algorithm $n_{n_j^*}^{a_i} \in N^A$ represents the node that is attached as a child to the model selection leaf node n_j^* and uses the optimization algorithm a_i to optimize a configuration for pipeline $p_{n_j^*}$. Therefore, the search tree $G = (V, E)$ of the model selection HTN planning is extended for N^A in the form of:

- $V' = V \cup N^A$
- $E' = E \cup \left(\bigcup_{n_i^* \in N^*} \bigcup_{a_j \in A} n_{n_i^*}^{a_j} \right)$
- $G' = (V', E')$

Since the nodes in N^* are now not longer leaf nodes of G' they are now referred to as *Pipeline Nodes* and the newly added leaf nodes N^A are the *Optimizer Nodes*. A simplified illustration of a possible G' without the HTN planning aspects of this extended search tree can be seen in Fig. 3.4. The pipeline nodes and the optimizer nodes are also marked for a better intuition of the extended search tree G' .

Now, if a pipeline node is reached, the task arises to select one of the connected optimizer nodes. The set of optimizer nodes below a pipeline node n_j^* forms the set $N_{n_j^*}^A = \bigcup_{a_i \in A} n_{n_j^*}^{a_i}$. But without prior knowledge it is impossible to reliably predict, which optimizer is ideally suited to optimize the configuration for the pipeline $p_{n_j^*}$, i.e. to select one node out of $N_{n_j^*}^A$ during a graph search.

A naïve approach is to randomly try out different optimization algorithms for optimiz-

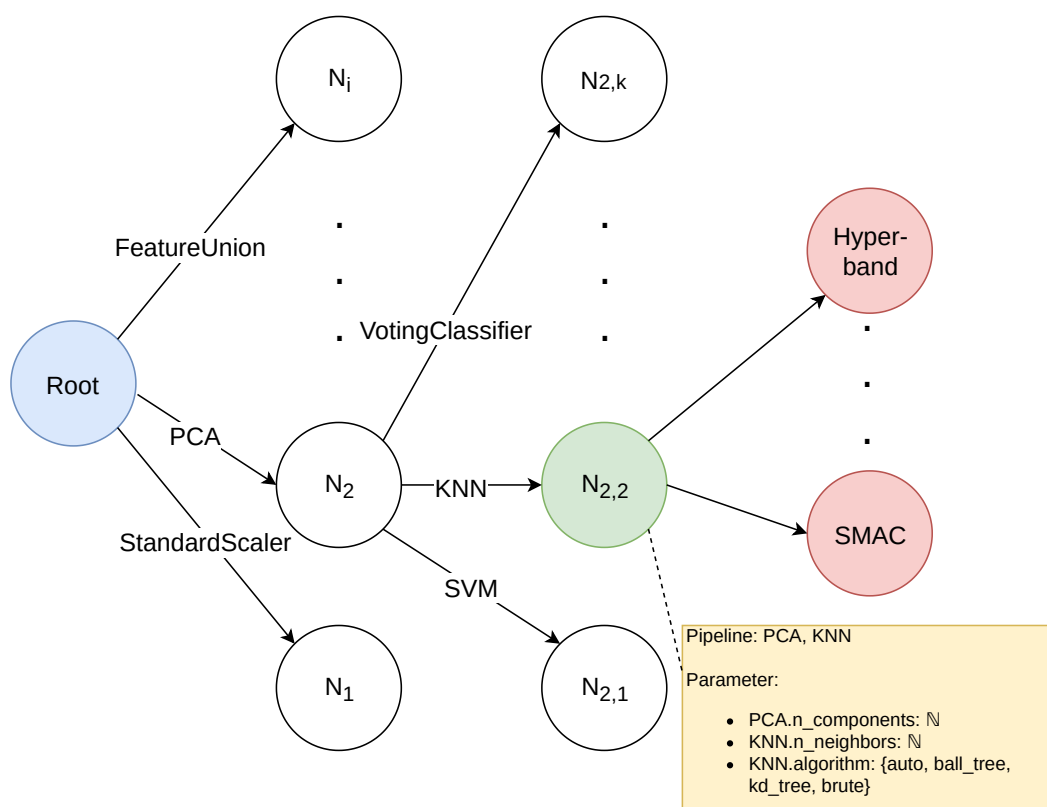


Fig. 3.4.: A simplified illustration of a possible extended search tree G' . The root is drawn in blue, the inner nodes with unsolved task-trees in white, pipelines nodes in green and optimizer nodes in red. The yellow box shows the configuration space of the pipeline that is represented by the task-tree in that pipeline node.

ing the parametrization and examine their optimization results to gather knowledge successively. This try-and-error approach is also referred to as *exploration*. During the collection of exploration results, it will become more evident, which optimizer yields which score quality.

Collecting a few sample scores for each optimizer is basically conducting an experiment since black box optimization is often a probabilistic process. Based on the number of experiment samples, the optimization capability of an optimizer can be estimated with a certain probability. After a sufficient number of samples is gathered, the most suitable optimizer can be selected and utilized with high certainty. Deliberately selecting and utilizing one specific choice with the expectation of a good result instead of more or less randomly exploring is also named *exploitation*.

The balance between both, exploration and exploitation, is a difficult challenge because of the limited optimization budget. With too much exploitation it can not be assured with a high probability that the exploited choice is the actual best one since there is a lack of optimization score samples. But with too much exploration it may

be possible to determine the probably best choice, but there is not enough budget left to actually utilize this choice to a higher extend. This balance between exploring and exploiting a set of choices, which have an unknown quality, is the fundamental problem of the so called *Multi-Armed Bandit* problems.

A Multi-Armed Bandit problem is usually considered in the context of time steps t_1, t_2, \dots, T , which are sometimes also called turns. T is in this notation the current turn of the ongoing series, i.e. the current point in time. For every turn up to T , one arm, i.e. one of the K choices $C = \langle c_1, \dots, c_K \rangle$, is selected by a selection strategy $s : t \rightarrow [1, K]$.

For a turn t , a random reward r is received after selecting the choice $c_{s(t)}$. Each arm c_i has an underlying probability distribution D_{c_i} for the reward values, i.e. $r_{c_{s(t)}} \sim D_{c_{s(t)}}$. In the AutoML context, where each arm is one optimizer, this reward could be the testing accuracy score of a pipeline, which is configured with the best found parametrization of the chosen optimizer after a certain optimization budget. Of course, each distribution D_{c_i} has a corresponding expected value μ_{c_i} . Therefore, if for example the selection strategy is $s(t) = 1$ the accumulated expected reward is $\sum_{t=1}^T \mu_{c_1}$.

An optimal strategy s^* would always pick the choice with the highest expected reward $\mu^* = \max_{i \in [1, K]} \mu_{c_i}$, such that $s^*(t) = \operatorname{argmax}_{i \in [1, K]} \mu_{c_i}$. However, s^* is unknown and has to be approximated by adjusting the constructed s over time.

For every turn where $s(t) \neq s^*(t)$, the strategy loses $\mu^* - \mu_{c_{s(t)}}$ in expected reward. The exploration vs. exploitation balance problem is here noticeable, since the strategy has to find out the arm with μ^* by trying out every arm, but also wants to keep $\mu^* - \mu_{c_{s(t)}}$ as minimal as possible via exploitation of an optimal arm. Otherwise, the difference to μ^* is expected to grow with every turn where a non-optimal arm is selected. This idea of keeping the expected reward loss minimal in the overall context of $\lim_{T \rightarrow \infty}$, such that the strategy has time to approximate s^* , is formulated in the form of the *total expected regret*: $R_T = T \cdot \mu^* - \sum_{t=1}^T \mu_{c_{s(t)}}$. Different algorithms for creating such selection strategies s over time can be compared by their respective expected total regret R_T , which should be as low as possible.

Selecting an optimizer node out of $N_{n_j}^A$ to perform the model configuration of the pipeline $p_{n_j}^*$ can be formalized as a Multi-Armed Bandit problem. With a given timeout for each optimization run, the optimizers are called repeatedly, which would be the turns of the bandit problem. The selection strategy for an optimal $p_{n_j}^*$ as well as the most suitable optimizer for a model configuration of this pipeline is unknown beforehand and has to be explored. But since the time budget is limited, the expected regret of not exploiting this most suitable optimizer for the optimal pipeline is supposed to be as low as possible.

With that assumption of formalizing this selection as a Multi-Armed Bandit problem,

there are several existing algorithms to construct selection strategies for optimization nodes until the optimization budget is spent, which balance exploration and exploitation intending of keeping the total expected regret minimal. Common choices are for example *Thompson Sampling* [Tho33] or the usage of *Upper confidence bounds* (UCB) as for instance in the *UCB1* algorithm [ACF02].

3.2.4 Ensemble Interaction with MCTS

Let n_j^* and therewith $p_{n_j^*}$ be already selected out of the model selection graph, now there is one arm for each optimization algorithm, i.e. $|A|$ arms. With this very limited set of arms, any of the referenced Multi-Armed Bandit selection strategy construction algorithms could be applied and with a large enough optimization budget, there is a high probability that the most suitable optimizer will be exploited enough to get a good parametrization for $p_{n_j^*}$.

But since n_j^* is not selected yet because no actual model selection is done until now, there is no pipeline $p_{n_j^*}$ selected and therefore the set of arms cannot be limited to $N_{n_j^*}^A$. Instead, the set of arms would be the set of optimizer nodes N^A , i.e. the set of combinations of all possible pipelines with all optimization algorithms. This set of combinations can be enormously big or even have an infinite amount of elements, depending on the specification of the model selection space. For such a high number of arms, any Multi-Armed Bandit algorithm would need a massive number of steps for exploring every arm sufficiently, until an educated guess for a good exploitation candidate could take place. As a more effective alternative for this use-case, the tree structure of G' should be utilized for a better selection mechanism.

The selection of an optimizer from A for optimizing the configuration of a pipeline $p_{n_j^*}$ is formalized as a Multi-Armed Bandit problem in the form of choosing one of the optimizer nodes $N_{n_j^*}^A$ each turn. An appropriate Multi-Armed Bandit algorithm will exploit with a high probability the most suitable one of this optimizers $a \in A$ for the specific configuration optimization space of this particular pipeline. This suitability of an optimizer for a space is dependent on the structure of the space, i.e. the number of dimensions as well as the value sets of the different dimensions, and the properties and landscape of the target function in this space.

Although a is the best optimizer for the configuration space of this specific pipeline $p_{n_j^*}$, this does not apply to all pipelines. Another optimizer a' could, for instance, be the best optimizer for any pipeline that includes one specific component, which is not part of $p_{n_j^*}$. In that case, all subtrees below nodes where this component was selected would ideally exploit a' instead of a as the optimization algorithm. Because such hierarchical dependencies influence the best optimizer choice, the exploration and exploitation of a Multi-Armed Bandit algorithm should utilize these hierarchical dependencies as well.

A *Monte-Carlo tree search* (MCTS) is a heuristic search algorithm, where the heuristic for scoring nodes is based on Monte-Carlo simulations. With the results of the simulations, an MCTS implementation now needs a selection strategy for the next node that will be expanded. The aforementioned upper confidence bounds were successively applied as such a selection strategy in the form of the *UCB applied to trees* algorithm (UCT) [KS06]. For UCT is the goal to utilize the possibly hierarchical dependencies inside the trees. Although MCTS does not necessarily has to use this selection strategy, the two terms UCT and MCTS are often used interchangeably. In the case of UCT but also some other MCTS selection strategies, each node has an attached score, which is the basis for these selection strategies. This value is for example for UCT calculated via $\operatorname{argmax}_{n_i} \left(w_{n_i} + f \cdot \sqrt{\frac{\ln(t)}{v_{n_i}}} \right)$, but this formula can be adjusted depending on the desired exploration/exploitation behavior. w_{n_i} is the overall sampled score so far of the subtree below node n_i as an average, f is a factor for the tendency towards exploration, v_{n_i} is the amount of times the subtree below n_i was sampled, and t is the amount of overall turns/iterations so far. MCTS is usually defined as an algorithm with an iteration of four steps, which are listed in the following and additionally illustrated in Fig. 3.5:

1. **Selection:** The next node for expansion is chosen. Commonly, each node has a value assigned, and starting at the root node, the next node is the child node with the highest value until a not expanded node is reached.
2. **Expansion:** The node is expanded, i.e. each possible follow-up state of the state in the unexpanded node is constructed and a child node is attached for each.
3. **Simulation:** The new child nodes need some initial scores. Since an inner node represents an incomplete state, they are hard to score directly and a heuristic scoring for the subtree below this inner node is necessary. A pre-defined number of Monte-Carlo Simulations, which are basically random walks down the search tree until a leaf node is reached, are performed. The scores of the states in the found leaf nodes are used to approximate a score for this subtree. Thus, the scoring gets more accurate with a higher number of simulations per child node, since more different leaf nodes are reached and therefore more possible solution states are covered. But with a higher number of simulations, a bigger portion of the budget is spent each iteration and fewer iterations can be performed overall. A careful trade-off is therefore necessary.
4. **Backpropagation:** For the child node n , the simulations returned some scores x_1, \dots, x_k . Based on these scores, n will receive an initial value. Afterwards, every node in the path from the root to n will be updated to incorporate the

newly gathered knowledge as well via backpropagation starting at n . At first, n is scored initially with the following formula $\frac{\sum_{i=1}^k x_i}{v_t} + f \cdot \sqrt{\frac{\ln(V_t)}{v_t}}$. The formula consists of the following parts: $\frac{\sum_{i=1}^k x_i}{k}$ is the average of the simulation results, f is a pre-defined factor for the focus on exploration, v_t is the amount of visits n had after t rounds of the MCTS loop via previous Monte-Carlo simulations before it was expanded in this round or alternatively 1, and V_t is the visit counter v_t of the parent of n . For the actual backpropagation, starting with the parent of n which is referred to as n' and continuing until the root is reached, each nodes score is updated similar to the initial scoring of n . At first, the v_t counter of n' is increased. Let w_{t-1} be the scoring of n' of the previous round, it will be updated during the backpropagation of this round with a cumulative moving average $w_t = w_{t-1} + \frac{w_c - w_{t-1}}{t}$, where w_c is the already updated node score of the child node that was the previous stop during the backpropagation. Now, the new score of n' is $\frac{w_t}{v_t} + f \cdot \sqrt{\frac{\ln(V_t)}{v_t}}$.

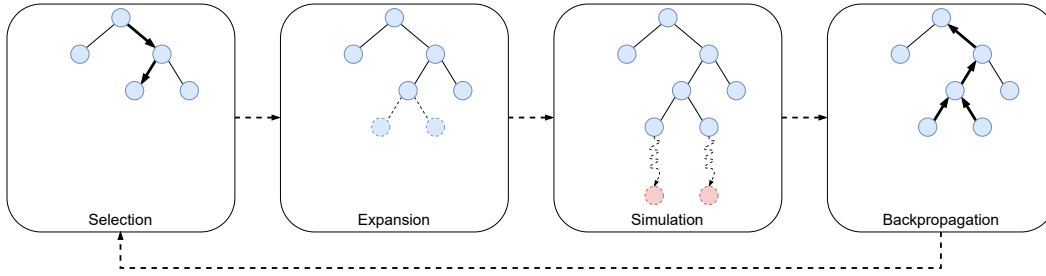


Fig. 3.5.: Illustration of the four MCTS steps. Inner graph nodes are shown in blue and the leaf nodes that are found during Monte-Carlo simulations are shown in red.

To apply MCTS for a model selection in a search tree for example constructed via HTN planning, minor adjustments are necessary. After incorporating the following changes, this modified MCTS is the algorithmic foundation of the approach of this thesis:

- Optimizer nodes are leaves and can not be further expanded. The Monte-Carlo simulations end their search once they find an optimizer node and start an optimization run of the optimizer of this node. If an optimizer node is chosen in the Selection phase, a model configuration run with a given optimization budget is started with the optimization algorithm represented by this node instead of a node expansion.
- The main MCTS loop does not run forever or for a fixed amount of steps. Instead, it keeps track of the spent optimization budget during the search itself and during the optimization runs and will stop after the current iteration, if

the budget limit is reached or the budget is nearly spent such that it is not possible to perform another complete optimization run.

As a user-specified configuration, an MCTS model selection implementation with embedded optimizers for model configuration would require some values:

- The overall optimization budget B .
- The budget b a single optimizer run can spent with $b \ll B$.
- The exploration factor f .
- The number of Monte-Carlo simulations starting at each new child node after expansion.

Additionally, the definition of the HTN planning space and the set of optimizers A is required to construct the search tree including the optimizer leaves to achieve an optimizer ensemble for model configuration. A simplified formulation in high-level pseudo-code of the modified MCTS for the approach of this thesis can be found in Algorithm 2. For this simplification the following parameters are expected: The overall AutoML timeout, the timeout for each single optimization run, a searchspace created beforehand from searchspace definition files, and the dataset in the format instances x and classes \hat{y} .

3.3 Model Configuration with Multiple Optimizers

With the modified MCTS, the model selection is conducted and additionally it is controlled when an optimizer is started for model configuration. But since the optimizers of the optimizer nodes $N_{n_j^*}^A$ below the pipeline node n_j^* , all try to optimize a configuration for the same pipeline $p_{n_j^*}$, they can utilize the fact they share the same optimization space. When this sharing and utilization of knowledge about the shared optimization space is achieved, this will be called an *optimizer ensemble* and is in combination with the integration into a search graph the core novelty of the approach of this thesis.

First, it is explained in the following, how previous candidate evaluations of the shared optimization space are stored and afterwards, how optimizers re-use this stored candidate scores for new optimization runs, which is often also referred to as warmstarting.

Algorithm 2: frankensteins-automl

Result: Best found pipeline and corresponding score.**Data:** Timeout, OptimizerTimeout, Searchspace, x , \hat{y}

```
1 bestPipeline  $\leftarrow$  null
2 bestScore  $\leftarrow$  0
3 graphGenerator  $\leftarrow$  new MCTSGraphGenerator(Searchspace,  $x$ ,  $\hat{y}$ )
4 rootNode  $\leftarrow$  graphGenerator.getRootNode()
5 while  $\neg$  Timeout is spent do
6   n  $\leftarrow$  rootNode
7   while  $n$  is expanded  $\wedge \neg$   $n$  is leaf node do
8     | n  $\leftarrow$   $\underset{c \in n.\text{getChildNodes}()}{\text{argmax}}$  c.getNodeValue()
9   end
10  startNodes  $\leftarrow$  {}
11  if  $n$  is leaf node then
12    | startNodes  $\leftarrow$  startNodes  $\cup$  n
13  else
14    | graphGenerator.expand(n)
15    | startNodes  $\leftarrow$  startNodes  $\cup$  n.getChildNodes()
16  end
17  (reachedLeafNodes, results)  $\leftarrow$  monteCarloSimulations(startNodes,
18    OptimizerTimeout)
19  backpropagate(reachedLeafNodes, results)
20  for (pipeline, score)  $\in$  results do
21    | if score > bestScore then
22      | bestScore  $\leftarrow$  score
23      | bestPipeline  $\leftarrow$  pipeline
24    | end
25  end
26 return (bestPipeline, bestScore)
```

3.3.1 Shared Parameter Domain for Selected Models

In every pipeline node $n^* \in N^*$, the optimization space for the model configuration of pipeline p_{n^*} can be deduced. Therefore, n^* can be the single point of contact for every subjacent optimizer node $n_{n^*}^a \in N_{n^*}^A$. With this centralized control instance it is possible to store and distribute knowledge from and to the optimizers.

At first, every $n_{n^*}^a \in N_{n^*}^A$ can get the concrete model configuration space from n^* , as well as other helpful information as for example the default parametrization of certain pipeline components. Sometimes the default parametrization may be a good starting point, i.e. first evaluation candidate, for some optimization algorithms. Additionally, each optimizer can store their evaluated candidates together with the

evaluation score at n^* . Although this requires additional memory storage, collecting and saving already evaluated candidates has the following advantages:

- If saved in a hashtable, candidates can be inserted with a generated key and retrieved with one as well. Therewith, it can be checked if a candidate was already evaluated before and if yes, return the existing score from n^* to the optimizer instead of training and evaluating a pipeline on the dataset again. The lookup complexity of a hashtable is constant, while training a machine learning algorithm is generally very time consuming in comparison. For example, the complexity of training an SVM with SMO [Pla98] is between linear and quadratic, depending on the dataset.
- If saved in a heap, retrieving the best candidates has a logarithmic complexity and thus, the best n candidates from previous runs of the same or another optimizer from $N_{n^*}^A$ can efficiently be requested from n^* as a starting point for a new optimization run.

Memory is in general a comparably cheap hardware component, which can be added to the computer nearly indefinitely such that a lot of evaluation results can be stored. In comparison, the capacity of the CPU, which affects the time for evaluations, has upper limits. When these two suitable datastructures are used, the speedup of sharing information about evaluations instead of evaluating once more outweighs therewith the disadvantage of higher memory demands.

3.3.2 Warmstarted Versions of Optimization Algorithms

If the optimization budget is high enough, there is a high probability that the MCTS will select or the Monte-Carlo simulations will reach a pipeline node more than once. Meaning that an optimization run is started in a model configuration space where already candidate evaluations have taken place. Additionally, all these previous candidate evaluations are stored in the pipeline nodes for the given model configuration optimization space. Hence, the optimization algorithms should utilize this advantage of prior collected data instead of performing the optimization anew from scratch every time.

Starting with a foundation of previously collected data instead of a completely unexplored optimization space is called *warmstarting* an optimizer. However, it is important to have a certain robustness against local optima, because without it an optimization algorithm could start in a local optimum, which was found in another optimization run, and might not get out of this local optimum. This robustness can

be achieved by the optimization algorithms themselves, but additionally a little bit by the selection of the data for the warmstart as well.

In the following, it is explained how the optimization algorithms that are used for the optimizer ensembles of this approach can be modified to be executed warmstarted. All these optimization algorithms are established black box optimization algorithms, which were presented in section 2.3.2 up to section 2.3.4. They will therefore utilize an ordered list of pairs of already evaluated candidates for warmstarting in the form of $l = \langle (c_1, s_1), \dots, (c_k, s_k) \rangle$, where c_i is an already evaluated pipeline configuration and s_i the corresponding score of this evaluation. The list l is created from a subset with the top $|l|$ candidates out of the set of all previous evaluations and $|l|$ can be decided by the corresponding optimization algorithm.

Random Search

The first variant of a random search, where each candidate was drawn independently and at random, has no advantages from warmstarting besides not having to re-evaluate candidates that were already evaluated.

But the second variant of a random search, where a candidate is sampled out of the hypersphere around the current best candidate up until either the optimization budget is spent or no improvement happened for i iterations, can profit from a modification to a warmstarted version. This version could use the top pair of l , or alternatively one randomly chosen pair out of the top n pairs of l for more variation, as a starting point.

Now, the random search will start sampling candidates in the hypersphere around this starting point and has the chance to find a better neighboring configuration, because it already is in a candidate with some good previous evaluations.

Of course the chosen starting point could be a local optimum, but this random search variant will use this hypersphere for at most i iterations if no improvement is achieved, which can be an indicator of an optimum. In this case, it will select a new random starting point. Thus, not more than i iterations are lost to a local optimum selected from l , which is depending on the optimization budget usually manageable, since random search is a comparably simplistic and therefore fast optimization algorithm.

Hyperband

Hyperband is basically a method of treating multiple, parallel running candidate evaluations as a Multi-Armed Bandit problem and stopping the less successful

instances to have more optimization budget for the better performing instances as well as new instances. The candidates for evaluation are chosen randomly and this is therewith similar to a random search to a certain degree. Hence, it can also be warmstarted similar to a random search by sometimes choosing candidates in the neighborhood from candidates with high evaluation results.

The typical Multi-Armed Bandit problem of balancing exploration and exploitation should be applied here as well and not all candidates should be from currently high performing neighborhoods. If Hyperband is configured to have k parallel running evaluation instances, $\lfloor \sigma \cdot k \rfloor$ instances with $\sigma \in (0, 1)$ should start in the direct neighborhood of one of the top candidates from l for exploitation, while $\lceil (1 - \sigma) \cdot k \rceil$ instances should start at new and randomly chosen configuration candidates for a better exploration coverage of each Hyperband run.

Genetic Optimization

A very important part of a genetic optimization algorithm is the selection scheme with which individuals from the last generation will be selected to construct the next generation. Such a selection can be a naïve approach, as for example a random selection or the best individuals from the current generation. But it can also be a more sophisticated scheme as for example a *Boltzmann Tournament Selection* [Gol90].

With such a selection scheme in place, the warmstarting can be implemented straightforward. Instead of selecting individuals with a selection scheme from a previous generation, the selection scheme of the genetic algorithm will be applied on l . Therefore $|l|$ is set to the value that is used as the population size.

Alternatively it is also possible to re-use the Hyperband warmstarting approach, such that $\lfloor \sigma \cdot k \rfloor$ candidate are selected via the selection scheme from l and $\lceil (1 - \sigma) \cdot k \rceil$ candidates are randomly created to achieve more variance and a certain robustness against local optima.

SMAC

For the Bayesian optimization implementation *SMAC*, a warmstarted variant was developed by Lindauer and Hutter [LH18]. During the optimization, *SMAC* trains a regression model, for instance a Random Forest, to predict the performance of a configuration before actually evaluating it. This regression model is then used to guide the selection of candidates that are actually evaluated to build the surrogate function model. Additionally to training this regression model progressively during the optimization, it can be trained beforehand with l or any other subset of all

previously evaluated candidates to have a more accurate regression model right from the start of the optimization run.

Discretization Search

This optimization algorithm is based on the optimization method from *ML-Plan*, where the inner nodes of a search tree are used to further discretize one dimension of the optimization space at a time with each node. This is achieved by refining the ranges of the dimensions, i.e. splitting them, into several value ranges. For each split, a child node is attached during a node expansion until, depending on the parameter type, either the interval only consists of one value or the splits interval size is below a given threshold. Afterwards the next dimension is refined until each dimension was completely discretized and a leaf node is reached.

Therefore, this can be viewed as converting a grid search approach into a tree structure, which will be searched more dynamically than a grid search, with a best-first search, where the nodes are scored via Monte-Carlo simulations.

In *ML-Plan* the optimization was done for model selection and model configuration in a joint graph. Here for this approach, instead of creating a search graph for the complete AutoML problem solution space, it is solely applied on the model configuration space, i.e. hyperparameter optimization.

Warmstarting does here not imply that a starting point of the optimization is selected out of l , because the selection is controlled by the best-first search depending on the node scores. However, if the same optimizer instance is started more than once, this optimization approach can be warmstarted in the way that a search graph is already created for this space and traversed to a certain degree. Hence, the search can simply be continued.

Implementation

The approach of this thesis, described in the previous chapter, is realized as a reference implementation in the *Python* programming language and can be found here: <https://github.com/Berberer/frankensteins-automl>.

For simplicity and practicability reasons, this implementation has some minor restrictions regarding the expected input, but this can be changed or generalized to more inputs with minor modifications. As for the current implementation state, the optimization budget input is limited to time as the budget type and the dataset input has to be either in the *ARFF* file format, popularized by *WEKA* [Wit+16], or directly as *NumPy* arrays.

This reference implementation has two contributions to this thesis:

- Help for a better understanding as an alternative and more practical representation of the approach, if the reader prefers code instead of a theoretical and formal description.
- For the evaluation of this approach and the comparison against other approaches in chapter 5, this approach must be provided as an executable program to get results for different experimental settings.

The implementation is explained in the following chapter, which is structured into three parts. At first, the architectural composition of the implementation and the interaction of the included components is explained. With this overall perspective of the implementation, the codebase will be easier to understand and to navigate. Since the implementation was not done from scratch, the utilized Python libraries are listed and their functionality is explained as a second part to acknowledge their creators as well as to elucidate the contribution of the library to the overall functionality of the reference implementation. Finally, the description schema of the AutoML optimization space definition is provided, which is a required input for the implementation, such that a user can understand the included optimization spaces of the implementation and modify them if needed.

4.1 Components of the Project and their Interaction

The components of the implementation and their architectural composition can be separated into three parts with regard to the subject matter and this separation will be used in the following to outline and explain the implementation. All explanations of the implementation will be accompanied by UML class diagrams of the corresponding sources to support the description of the functionality. These diagrams only depict the most relevant core mechanics of the respective classes, while getter and setter methods as well as other minor utility methods are omitted.

For the model selection as the first of the three components, the search space needs to be constructed from the input optimization space definition. Additionally, the management and creation of a search tree for HTN planning domains is part of this module as well.

In the second part, the components for evaluating machine learning pipelines and handling the different warmstarted optimizers as an optimizer ensemble are explained.

Finally, all components from the two previous parts are combined via an MCTS implementation into an overall AutoML tool and all components for this are explained in the third part.

4.1.1 Model Selection Search Space Management

Since the model selection is reduced to a graph search problem, the search space is managed as a graph datastructure as shown in Fig. 4.1. Because in other components of this implementation, basic graph modeling functionalities are needed as well, abstract base classes for graph nodes and a graph generator are created to provide this basic datastructure.

Each `GraphNode` is identified by a unique id and knows its predecessor nodes as well as the successor nodes. Depending on the use-case for the graph node, the abstract method `is_leaf_node` has to be implemented with the respective logic of the use-case.

The associated `GraphGenerator` class has abstract methods for `get_root_node` and `get_node_successors`, which have to be implemented domain-specific as well. Nevertheless, `get_node_successors` is not intended for general usage in other classes. Instead, the method `generate_successors` shall be used. It performs checks if the parameter node is either a leaf node or already has a reference to its successor nodes. Only if neither is the case, the abstract method `get_node_successors` is used to get new successors from scratch with the use-case specific logic.

For this specific case of having an HTN planning graph for model selection, the `SearchSpace`, `SearchSpaceComponent`, and `SearchSpaceRestProblem` classes are the underlying data structure for modeling the HTN planning problem.

Basically, `SearchSpaceComponent` is a wrapper class for components of the AutoML search space definition, which are given as input in a schema defined in section 4.3. Additionally, this class has some helper methods for handling parameters. It can create the default parametrization or alternatively draw a random parametrization of this component. For a given parametrization, it can check the validity, i.e. exists a value for each parameter and are all selected values allowed for their corresponding parameter. Finally, it can split a given parametrization into positional and keyword parameters for utilizing them for a concrete Python function call that is supposed to create this component, which is usually but not necessarily a constructor of the component's class.

Within the `SearchSpace` class, all these components defined in the user-provided input are managed and can be retrieved either by their name or by an interface they provide.

Each planning state in an HTN planning process has a rest problem, i.e. which tasks are solved and which are not, for example because they are not decomposed yet. The class `SearchSpaceRestProblem` is utilized to store this information. It saves all required interfaces from both solved and unsolved tasks, which are included in the current plan. Additionally, it has a mapping, which required interface is satisfied with which component that provides this interface.

For using problem representations in a graph search, it has the method `is_satisfied`, which returns true if all required interfaces have a suitable mapped provided interface, and `get_first_unsatisfied_required_interface` to pick the next interface of an unsolved task to progress the planning by satisfying it. Each graph node for the HTN planning graph represents, that one additional interface has been satisfied in comparison to the parent node. Therefore the class has an additional static helper method `from_previous_rest_problem` to create a rest problem from the rest problem of the parent node by satisfying one required interface.

When all required interfaces in the interface abstraction layer of the considered HTN task-tree are satisfied and a parametrization is generated afterward via the model selection, a concrete `SearchSpaceComponentInstance` can be created for a `SearchSpaceComponent` with this mapping from interface to component and the parametrization. With the method `construct_pipeline_element`, the actual pipeline element which is represented by the component instance can be instantiated and configured with the parameters and if necessary, the instantiated pipeline components represented by the required interfaces.

Finally, the `GraphNode` and `GraphGenerator` classes are inherited for the use-case of model selection HTN planning via a graph search with the `SearchSpaceGraphNode` and `SearchSpaceGraphGenerator` classes. The abstract methods of the base classes are implemented in the following manner:

- `GraphNode.is_leaf_node`: Each `SearchSpaceNode` represents one specific rest problem. This node can be considered a leaf node of the planning graph if `SearchSpaceRestProblem.is_satisfied` returns true.
- `GraphGenerator.get_root_node`: The `SearchSpaceGraphGenerator` has the name of the component that represents the input task of the planning problem. It can get this component from the `SearchSpace` instance, construct a `SearchSpaceRestProblem` for this initial component, and create the root node with it.
- `GraphGenerator.get_node_successors`: The rest problem of the node, which is provided as a parameter of this method, will return one specific unsatisfied interface with its `get_first_unsatisfied_required_interface` method. By calling `SearchSpace.get_components_providing_interface` with the name of this first unsatisfied interface, all suitable components are found and afterwards calling `SearchSpaceRestProblem.from_previous_rest_problem` for each of these components, will generate the rest problems for the successor nodes.

4.1.2 Pipeline Evaluation and Optimizers for Model Configuration

Once the model selection procedure has created a pipeline topology in the form of a satisfied *SearchSpaceRestProblem*, the model configuration has to optimize a parametrization for the components of this topology. This is the task of the optimizer ensembles, which are implemented as illustrated in Fig. 4.2.

For each created pipeline topology of the model selection, i.e. one for each Pipeline Node, a `OptimizationParameterDomain` object is created and attached to the pipeline node object. It can manage the components and the corresponding parameter space of this pipeline as well as all already evaluated candidates parametrizations.

All parametrizations are considered from two points of view: For creating actual pipeline objects for evaluation with a component mapping, where the parameters are stored in a map to have them associated to their component, and for optimizing these configurations, which is done in the more generalized form of NumPy

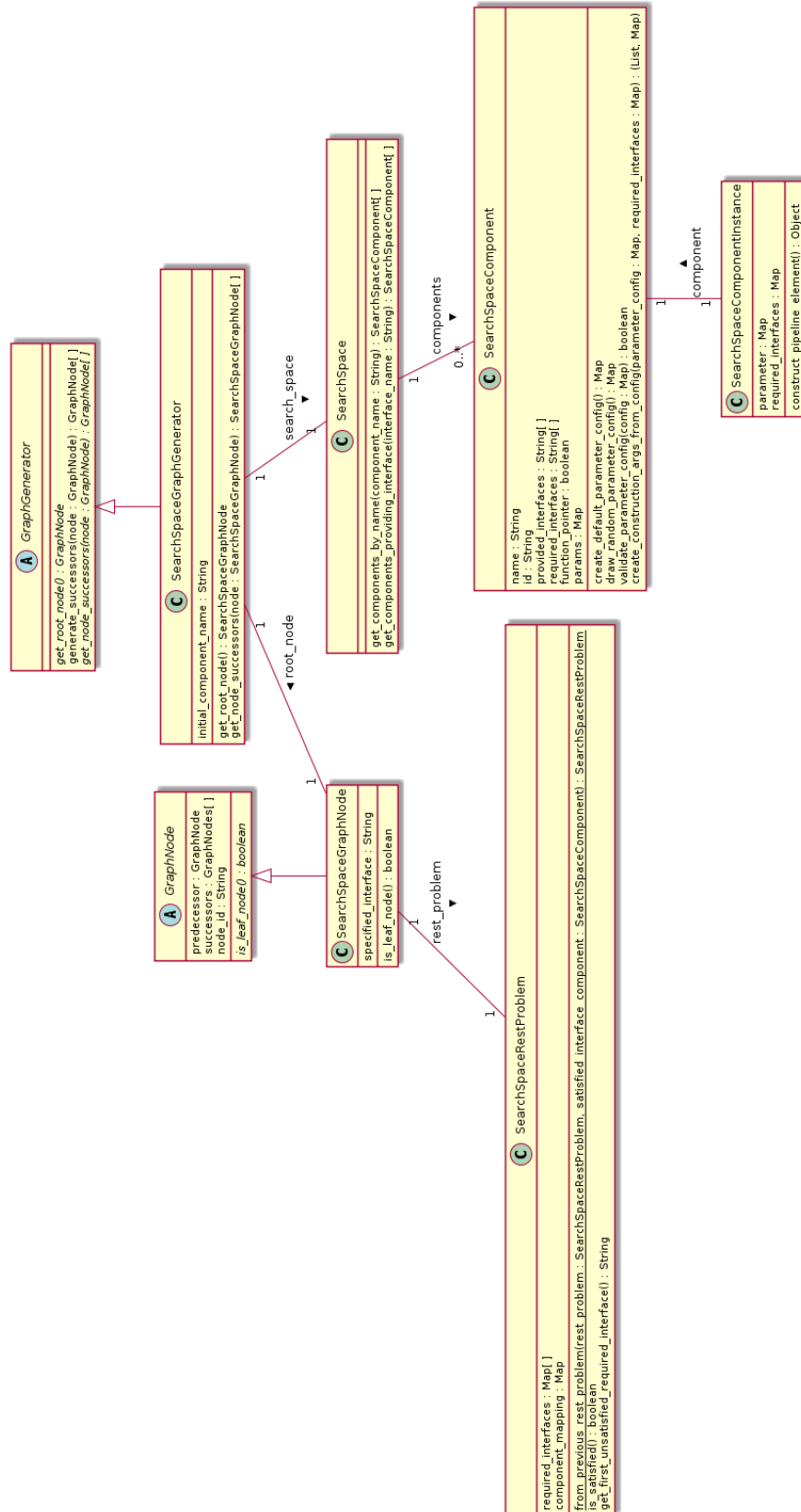


Fig. 4.1.: A simplified overview of the implementation components for managing the search space and enabling HTN planning in this space.

arrays, i.e. basically vectors. For these two perspectives, the optimization domain `OptimizationParameterDomain` can perform transformations between both formats.

As a starting point for optimization runs, the domain can create the default configuration and random configurations. Afterwards during the optimization, the domain keeps track of all results in both a heap as well as a mapping from a hashed parametrization id to the scores, i.e. a hashtable. This duplicate storing has the downside of a higher memory consumption, but the upside of faster access in two different use-cases:

1. Retrieving the top n results from all already evaluated results via the heap, if one optimization method needs this for a warmstart. Thus, it can be defined individually for each optimization algorithm how many datapoints it utilizes for warmstarting. When a high n is selected, the optimizer will also get information about regions of the optimization landscape with a low performance such that it can avoid these regions.
2. Checking if a parametrization was already evaluated and if yes to retrieve the score via the mapping in the hashtable to prevent a repeated evaluation.

The core functionalities that all optimization algorithms might need, are provided in the `AbstractOptimizer` base class. All of the following methods are implemented there:

- `_score_candidate`: Check in the domain, if the parametrization has already been evaluated and return this result if yes. Otherwise, utilize the attached `PipelineEvaluator` class to construct a pipeline from a satisfied rest problem, which is provided from the pipeline node. This constructed pipeline is evaluated regarding the accuracy of the pipeline via a cross-validation with two different stratified 70%/30% splits and afterwards this new accuracy score is added to the domain. The construction and cross-validation is executed in a detached process for a better parallelization and easier termination if the timeout for a pipeline evaluation is reached.
- `_random_transform_candidate`: Changes n random elements of the candidate parametrization vector by a small random value. This is used for the search step in the random search, therewith the initialization of Hyperband as well, and the mutations of the genetic algorithm.

Each optimization algorithm now has to realize their optimization logic by implementing the abstract `perform_optimization` method in the sub-class for this optimization algorithm. This method has the optimization time budget as an input

parameter, to perform the corresponding optimization within this budget. Compliance with this budget constraint is the task of the optimizer and is not controlled from any other module of the implementation externally.

In this current reference implementation, five optimization algorithms are implemented:

- **RandomSearch:** This random search implementation is based on the second random search variant from section 2.3.2, i.e. it warmstarts with the best candidate so far from the optimization domain of this pipeline topology, if one exists, as the current candidate and takes the default configuration as an alternative option if no parametrization was evaluated yet. Afterwards, a random point from a small hypersphere surrounding the current candidate is selected and evaluated. If this newly selected candidate shows an improvement, it is selected as the next current candidate for the next search iteration. Otherwise, the last current candidate is kept. This is repeated until the optimization budget is spent.
- **Hyperband:** This Hyperband implementation is based on an existing open-source implementation¹. The existing implementation is utilized in the wrapper class `HyperbandRunner` which is called by the `Hyperband` optimizer class as a part of the common integration concept via the `AbstractOptimizer` class. It is warmstarted by selecting candidates from the direct neighborhood of the top 50 configurations from the optimization domain, or alternatively all existing if there are less than 50, and fill the list of starting points with random configurations up until 100 starting points for the Hyperband. This is done to achieve a balance between exploration and exploitation right from the start, because Hyperband is internally managed as a Multi-Armed Bandit problem.
- **GeneticAlgorithm:** For this reference implementation, the genetic algorithm implementation was modeled after *TPOT*, since this approach has shown good results for tackling AutoML problems with genetic algorithms. Therefore, the following operations are applied in every iteration to produce the next generation of 100 individuals from the current population:
 1. Select the top 20 individuals.
 2. Create 5 copies of each individual to get back to a population size of 100 for the next generation.

¹<https://github.com/zygmuntz/hyperband>

3. Select 10 random offsprings out of this 100 offsprings and create 10 new offsprings out of them via one-point crossovers of random pairs of this 10.
4. The remaining 90 candidates are modified in the form of point mutations.

The warmstarted initial generation of 100 individuals for each genetic algorithm run is initiated with the best 20 candidate configurations from the optimization domain and 80 random configurations.

- **SMAC:** This is a wrapper class for the official *SMAC* implementation. As in the Hyperband integration, it is warmstarted with the top 50 candidates, if so many exist, and additional random configurations to get 100 starting point configurations. In theory, it would be beneficial for SMAC to have as much warmstarting data as possible, but SMAC requires a so called *RunHistory* object as warmstarting input that has to be created on-the-fly for each optimization run. Hence, the size of the input was limited to 100 for this reference implementation. As future work, this limit can probably be extended and thus increase the capabilities of the SMAC integration.

- **DiscretizationSearch:** For a discretization search, in the optimization domain, an order is set for the parameters of the pipeline topology and this order is also applied for creating new child nodes via discretizations.

For parameters of type boolean or categorical, a single refinement is sufficient where two child node represents true and false, or alternatively one node for each categorical value. In the case of integer or real numbers, the minimum and maximum can be really far apart and creating a lot of child nodes in a single refinement step to achieve a high coverage of the values in between will blow up the degree of the graph enormously. Instead, numeric parameters are split by half and one child node is attached for each half. This is continued until each half consists only of a single number in the case of integers, or the lower and upper bound of the halves are smaller than $\varepsilon \cdot (\max(p) - \min(p))$ of the range of parameter p , in which case this node represents the value $\frac{\max(p) - \min(p)}{2}$.

For creating these discretizations depending on the type, the *Discretization* class and especially its static methods are used. Once more, subclasses of *GraphNode* and *GraphGenerator* are created where each node represents one discretization down to the leaf nodes where the last parameter dimension is discretized as far as possible. Therewith, the actual values for all parameters are selected in this leaf, because every dimension is discretized down to a single value. Here, the discretization is atomic.

The graph that is created with this mechanism is searched via a best-first

search, where each node is scored via the best result of three Monte-Carlo simulations, similar to ML-Plan. If all child nodes of a node were visited during the search, this node is marked as covered and will be ignored from this point on. Hence, it can occur for small parameter spaces and large optimization budgets, that this approach will finish with every possible candidate evaluated before the optimization budget is spent because of the discretized view of a continuous space.

4.1.3 MCTS and Optimizer Integration for AutoML

So far, the model selection is implemented as an HTN planning problem with a suitable graph search space. Now, the model configuration has to be embedded, i.e. the graph generation has to be extended such that optimizer nodes are created for the optimizer ensembles. Afterwards, this model selection search space with embedded model configuration has to be actually searched for a functioning AutoML tool. Additionally, it needs a user-facing interface, i.e. a Python class the user can use directly in their program, such that anyone can start an AutoML process with their custom dataset. These functionalities are illustrated in Fig. 4.3 and outlined in the following.

A user should start with creating a `FrankensteinsAutoMLConfig`. They can customize all configuration values as wanted, but all have suitable default values such that this can be omitted as well. The only requirement is a dataset as the AutoML problem input, which can either be given as a `*.arff` file in combination with the index of the target class column or alternatively as two NumPy arrays. With such a configuration object, the actual AutoML runner `FrankensteinsAutoML` can be instantiated and started.

After the start, it will create an `MctsSearchConfig` according to the user-provided `FrankensteinsAutoMLConfig` and start an `MctsSearch` with it. This search object will create the HTN `SearchSpace` object with the search space definition files of the AutoML optimization space, which is the basis for the MCTS graph.

For the concrete use-case of an MCTS as well as embedding the optimizers, the `SearchSpaceGraphGenerator` and the `SearchSpaceGraphNode` are extended with additional functionality to support these two aspects on top of the the HTN planning search graph generation. The graph generation works for the most part identical to the graph generation of the HTN planning but will generate the extended search graph G' instead of the HTN planning graph G . It will attach an `OptimizationParameterDomain` instance to the pipeline nodes, i.e. the model selection leaf nodes $n_j^* \in N^*$, which is the parameter domain for this concrete model selection pipeline outcome $p_{n_j^*}$. The graph generator receives a list of `AbstractOptimizer` classes as an input, such that it can create the additional optimizer nodes $N_{n_j^*}^A$ as

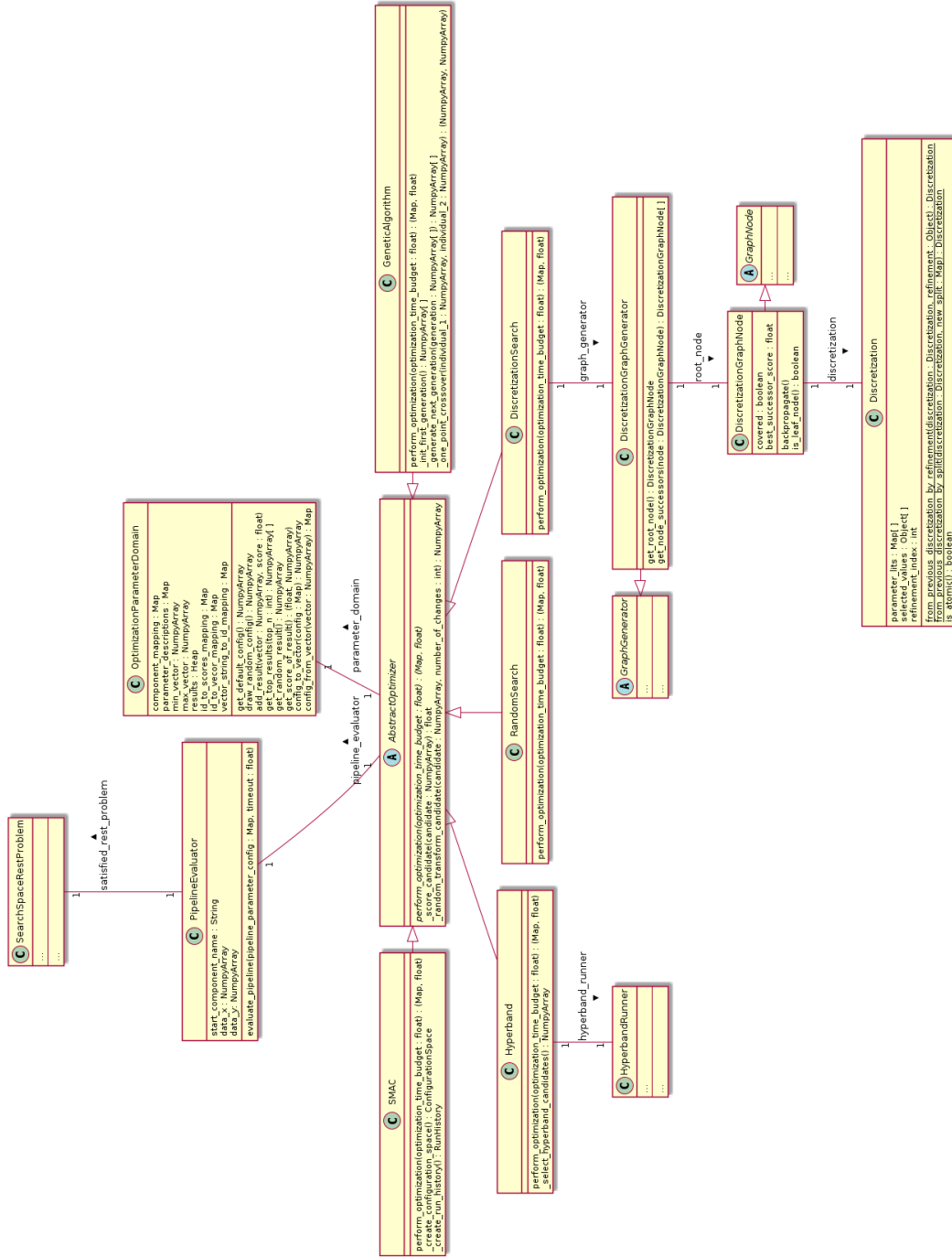


Fig. 4.2.: A simplified overview of the implementation components for storing the information about a model configuration optimization space and how the different optimizers approach this space.

child nodes to n_j^* for all given optimizer classes. Thus, there can be three possible types of an `MctsGraphNode`:

1. It is an inner node, where the HTN planning state is not satisfied yet. It has neither a parameter domain nor an optimizer attached.

2. It is a pipeline node, where the HTN planning state is satisfied and the pipeline model selection finalized. Therewith, a parameter domain can be created and is attached to the node.
3. It is an optimizer node, where one of the ensemble optimizers is attached. It will utilize the optimization domain that is attached to this node's parent node, i.e. the corresponding pipeline node.

The `MctsSearch` class follows the usual MCTS loop, where the UCT values are stored and updated on `MctsGraphNode` level which is the basis for selecting the search paths. Every time a node at the current end of such a path is expanded during this loop, i.e. the child nodes are generated, each new child node is scored with Monte-Carlo simulations.

For this scoring mechanism, a list of all new nodes is given to an instance of a `MonteCarloSimulationRunner` as well as the number of runs that will be started for each new node. The single simulations are random searches that extend the Python `Thread` class in the form of the class `RandomSearchRun` to parallelize the search for optimizer nodes. Once all random searches reached an optimizer node, the optimizers in the corresponding optimizer nodes are started with the given `timeout` parameter as an optimization budget and when all optimizers terminate with their best result of this optimization run after the budget is spent, the simulation runner will return a list of node and score tuples. With this tuple list, the scores can be backpropagated from the expanded child nodes and the next MCTS loop iteration can be started.

4.2 Utilized Python Libraries

This reference implementation relies on a few Open-Source Python libraries. In the following they are listed and their contribution to the functionality of the approach explained in combination with references to their repository and/or creators. Libraries for development purposes that have no direct influence on the functionality, as for example libraries for unit-testing or code formatting, are omitted from this list but can be seen here: <https://github.com/Berberer/frankensteins-automl/blob/master/requirements.txt>.

- *LIAC-ARFF*²: A parser for files in the ARFF format to access their data from a Python program.

²<https://github.com/renatopp/liac-arff>

- *NumPy*³: As a part of the *SciPy* project [Vir+20] for scientific computation, NumPy is an implementation for a wide range of mathematical functions, especially matrix calculation.
- *PyPubSub*⁴: Library for implementing a Publish-Subscribe-Pattern in Python for event handling. If for different events of the AutoML workflow event publishers are added, it is easy to add additional functionality, as for example logging or visualization, as subscriber plug-ins in the form of event listeners, without having to add the functionality of the plug-in at every location in the code, where a relevant event occurs.
- *scikit-learn*⁵: The solved task-trees from the model selection and parametrizations from the model configuration have to be instantiated as an executable machine learning pipeline for evaluation during the optimization and as the final result of the AutoML problem. Scikit-learn [Ped+11] is an established machine learning library with a variety of included components. Most of the desired components of this AutoML optimization space will be included in scikit-learn and can be referenced in the model selection space definition JSON files. Although it is possible to replace scikit-learn components in this space definition with components from another machine learning toolbox to construct pipelines automatically with this toolbox as a foundation, scikit-learn would still be a required dependency because it is also used for the evaluation of pipelines.
- *SMAC v3*⁶: Current version of the official SMAC implementation. It already has new additions to the original approach included, as for example warmstarting.

4.3 Exemplaric File Format for defining the AutoML Optimization Space

To generate an HTN planning space for the model selection as well as to deduce a model configuration space for a constructed pipeline, an AutoML optimization space specification is required. The reference implementation of this approach uses an optimization space definition schema in the form of JSON files in a certain format. Since it is centered around the HTN planning tasks, which can be translated to

³<https://github.com/numpy/numpy>

⁴<https://github.com/schollii/pypubsub>

⁵<https://github.com/scikit-learn/scikit-learn>

⁶<https://github.com/automl/SMAC3>

pipeline components or pipeline construction steps, this JSON format is an array of components. Files for the component definitions have the following structure:

```
{
  repository: String,
  components: [
    {
      name: String,
      providedInterface: [
        String
      ],
      requiredInterfaces: [
        {
          name: String,
          construction_key: String | Number
        }
      ],
      function_pointer ? : Boolean,
      parameter: [
        {
          name: String,
          type: String,
          values ? : [
            String | Number
          ]
          min ? : Number,
          max ? : Number,
          default: String | Boolean | Number,
          construction_key ? : String | Number
        }
      ]
    }
  ]
}
```

The elements of this definition structure have the following semantic:

- **components:** An array of the components defined in this file. They all have the following structure:

- **name:** This value is used to resolve the corresponding Python class or function when constructing this component and has therefore to include the Python module path as well as the actual name, as for example `"sklearn.naive_bayes.GaussianNB"`.
- **providedInterfaces:** This array of strings contains unique identifiers for each interface this component provides. With one of this provided interfaces and possibly more other provided interfaces from other components, a compound task can be decomposed.
- **requiredInterfaces:** If this component is represented as a compound task and needs to be decomposed and solved to be instantiable, this array has to contain the required interface definitions for possible decompositions. Each definition has the following structure:
 - **name:** A unique identifier of a required interface to match valid components against it for decomposition.
 - **construction_key:** Since the resolved Python class constructor will be called to instantiate the current component, which has this required interfaces, the parameter type signature of the constructor has to be met with the call. The constructor will require the constructed component of the provided interface that is matched for this required interface, as one of its parameters. Therefore, the resulting component of this required interface's solved task has to be placed at the correct position of the constructor call regarding the type signature of the constructor. The construction key can either be an integer to represent the position in the parameter list or a string in the case of a keyword parameter.
- **function_pointer:** If this property exists and is set to true, the resolved Python function or constructor will not be actually called and the result returned after resolving the path. Instead, a pointer to the resolved function itself will be returned, for the case that another component requires a function pointer as a parameter and not a component object.
- **parameter:** Most components will require a parametrization if they are added to a pipeline. To deduce the model configuration space for a pipeline, this array includes a definition for each parameter the component requires:

- **name:** An identifier for this parameter to be distinguishable from other parameters of this component. Hence, it has to be unique in this concrete parameter array.
- **type:** The type of the parameter as a string. Valid values are "int" for integer numbers, "double" for numbers with potential decimal places, "cat" for categorical values(i.e. one out of a pre-defined set of values) and "bool" for a boolean value.
- **values:** If the parameter has the type "cat" this is an array of strings or numbers with the allowed categorical values. If the parameter has another type, this field can be omitted.
- **min and max:** For numerical parameters, i.e. parameters with type "int" or "double" a parameter range for valid values is defined with a minimum and a maximum. For other types, this two fields can be omitted.
- **default:** If the implementation of the represented component has a suitable default value for this parameter, it is defined here as a string, number, or boolean, depending on the parameter type.
- **construction_key:** Same functionality as the construction key of a required interface. If the component is instantiated, here is defined at which position or with which keyword parameter the parameter value will be passed to the resolved class constructor of this component. However, contrary to required interfaces this value can be omitted for parameters. In this case, it is assumed that the parameter is passed as a keyword parameter and the `name` property will be used as the keyword.

The component definition of a scikit-learn feature selection component named `SelectPercentile`, which can be included in the pre-processing part of a pipeline, is given in the following as an example:

```
{
  "name": "sklearn.feature_selection.SelectPercentile",
  "providedInterface": [
    "sklearn.feature_selection.SelectPercentile",
    "FeatureSelection",
    "AbstractPreprocessor",
```

```

        "BasicPreprocessor"
    ],
    "requiredInterface": [
        {
            "name": "FeatureSelectionScoreFunction",
            "construction_key": "score_func"
        }
    ],
    "parameter": [
        {
            "name": "percentile",
            "type": "int",
            "default": 50,
            "min": 1,
            "max": 100,
            "construction_key": "percentile"
        }
    ]
}

```

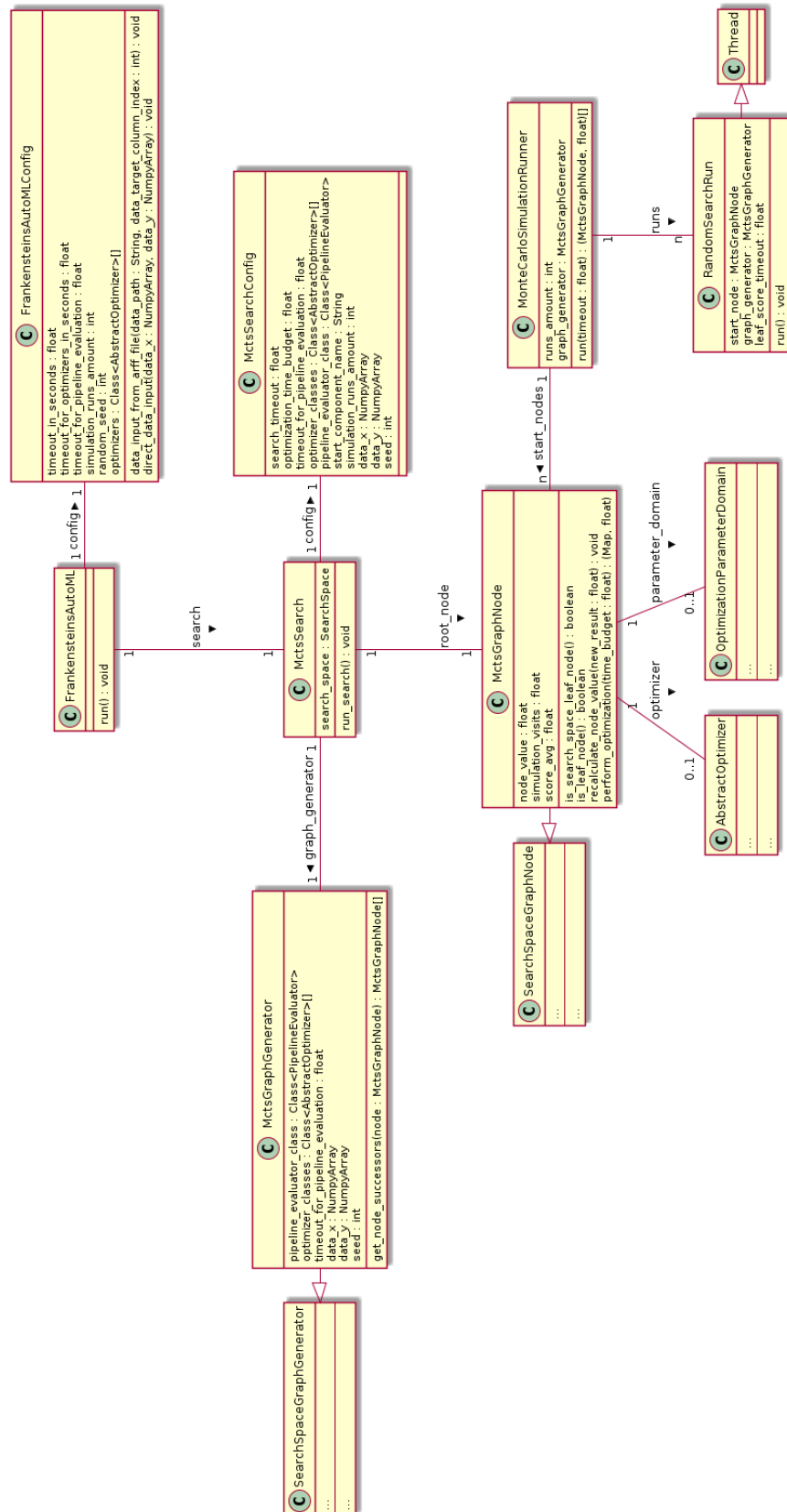


Fig. 4.3.: A simplified overview of the implementation components for searching HTN planning space with the integrated optimizer ensembles via an MCTS in an unified AutoML tool.

Empirical Analysis and Evaluation

After devising the approach for optimizer ensembles and implementing it as a reference implementation, it is essential to evaluate and assess this approach. This assessment is the goal of this chapter and the final step of this thesis.

The evaluated reference implementation is nicknamed *frankensteins-automl*, because this AutoML approach is metaphorically stitched together from different optimizers like Frankenstein's Monster. It will henceforth be referenced by this name while describing the experiment setup as well as listing and evaluating the experiment results.

To have a clear guided structure for this assessment of *frankensteins-automl*, it is based on the research questions presented in section 1.2, which are elucidated here in more detail at first. This more detailed elucidation of the research questions has a focus on which data is required to answer the corresponding question.

Afterwards, this chapter explains a series of Empirical evaluations in the form of experiments to gather this required data. Therefore, the setup and configuration of the experiments are given as part of these explanations.

Subsequently, the outcomes of the experiments are presented and analyzed to deduce the answers to the research questions from them.

5.1 Research Questions in Detail

The first research question is about the feasibility of the approach, i.e. is it a viable idea to optimize an AutoML problem solution with an optimizer ensemble. This will be answered in two steps.

At first, are the solution scores of this approach similar to or better than the scores of other state-of-the-art approaches for several different datasets? If they are not similar/better at all or not similar/better for a high number of datasets, *frankensteins-automl* is not a competitor to the state-of-the-art approaches. In this case, there would be no motivation to use this approach instead of other approaches, until this approach is further enhanced and the scores have improved.

Second, if there is an improvement in some scores, which of the two components of the approach, the MCTS model selection or the warmstarted optimizer ensemble model configuration, had which influence on the resulting scores. To assess these

influences, three versions of *frankensteins-automl* are evaluated, where two of them have slight modifications:

1. To evaluate the influence of a Model Selection via an MCTS, this modified version does not utilize an optimizer ensemble. Every optimizer node represents the same optimizer, which is a wamrstarted SMAC for this evaluation. In the remaining chapter it will be referenced as *frankenstein-mcts*.
2. To evaluate the influence of the ensembled optimizers, this modified version does not utilize an UCT based node selection. The node selection on each level down the paths to the different optimizer nodes, is random for each search iteration. Therefore, it is basically a repeated random search. Hence, it will be named *frankenstein-rs*.
3. The original and unmodified *frankensteins-automl* as described in the two previous chapters with an MCTS and optimizer ensembles with all five optimization algorithms.

By comparing the scores of the three variants for different datasets and different timeouts, data about the scores with and without one component can be gathered, which can be used to analyze the influence of the component.

As the second research question, the focus lies on the utilization of the different optimizers. During each run of *frankensteins-automl* it is counted how many times each optimizer was selected by the MCTS and therewith an optimization run with this optimizer started. Since the MCTS tries to find the best suited optimizer and to exploit it, this selection frequency could be an indicator of the suitability of this optimizer to the input dataset.

The frequencies are aggregated together with the properties of the AutoML input dataset. Thus, it is possible to check if one or more optimizers were preferably selected in general for all settings, i.e. for all datasets, or just in an AutoML problem setting with a certain input dataset that has certain properties. Alternatively, there could be no significant difference and each optimizer is selected comparably often across all observed AutoML settings, or one or more optimizer are used primarily unrelated to the dataset.

5.2 Experiment Setup

To answer the first step of the first research question, *frankensteins-automl* is benchmarked against different other competitor state-of-the-art approaches. The selection

of these approaches is made to represent the different established optimization strategies in the current state-of-the-art.

ML-Plan is selected for optimization via searching, *TPOT* as an approach with a genetic algorithm, *auto-sklearn* for Bayesian optimization, and finally *Mosaic* as a related work approach that utilizes more than one optimization method via an optimization space transformation.

Since *ML-Plan* is the only approach out of this selection, which is not implemented in Python but in Java, it can not be integrated seamlessly in the same experimental setup as the remaining ones. The experimental results for *ML-Plan* were kindly provided by the original authors of the approach Mohr et al. [MWH18]. All other approaches will be evaluated in a unified Python experiment environment that is explained in more detail in Appendix A.1.

The experimental setup has a certain set of constraints and rules, which are enforced equally for all approaches to achieve a fair and meaningful comparison. They are summarized in the following:

1. Each AutoML experiment setting (i.e. a combination of dataset + timeout + approach) will be conducted with 30 different random seeds to achieve a better statistical robustness against outliers in the evaluation scores.
2. The different approaches have 60 Gb RAM and 16 CPU cores, Intel Xeon E5-2670 with 2.6 Ghz, as a hardware limitation.
3. A pipeline candidate evaluation during the AutoML process, i.e. constructing, training and testing a candidate, must not take longer than five minutes and will be terminated otherwise.
4. All AutoML approaches will utilize the model selection and configuration space from *auto-sklearn*. This prevents the possibility that one approach appears superior because it utilizes other pipeline components that the other approaches have not included in their model selection space.
5. Before the AutoML approaches are started, the dataset is separated into two stratified splits. The AutoML approach gets the 70% split as an input and the resulting pipeline of the approach will be evaluated with the other 30% split to get a final score in the form of an accuracy measurement.

The approaches are configured to comply with this constraints, but all other configuration possibilities of the corresponding frameworks besides that are kept as their default configuration value for the experiments.

Restricting the optimization spaces to the auto-sklearn space implies three rules for a model selection and configuration space:

- The pipelines cannot have a higher complexity than the following topology data pre-processing + feature pre-processing + classification model/ensemble. But the topologies are allowed to be simpler by omitting one type or both types of pre-processor.
- A set of components for all three types is defined.
- A set of parameters as well as allowed parameter values is defined. All other parameters of a component are not allowed to be optimized and will be left to the default value.

Eventually, this may limit the capabilities of frankensteins-automl, as well as ML-Plan and TPOT, because they are capable of constructing more complex pipelines beyond this auto-sklearn space. However, these limitations are important for an informative and meaningful evaluation.

In this manner, the actual accomplishments of the approaches can be evaluated and compared to each other under fair conditions. It can be excluded that one approach outperforms another because his model selection space contains for example one component that the other approaches do not have for selection.

The adaptation of the auto-sklearn space in the JSON format of this approach¹, which is used by frankensteins-automl for the experiments, has a few limitations in comparison to the original auto-sklearn search space and is not a one-to-one adaptation. Optimization space modeling for auto-sklearn is done as Python code, which is as a turing-complete programming language more expressive than a data interchange language as JSON can ever be. With this modeling in Python, it is possible to create constraints and relationships between parameters of a component. For example something like if parameter p_1 has value a , p_2 cannot have value b , or p_4 will only be part of the model configuration if p_3 has value c .

Additionally, it is possible to include logical processing steps for the selected values after the model configuration in the Python code, for example to change a configured value depending on properties of the actual input dataset.

Improving the modeling capacities of this JSON format can be a starting point for further work. However, it will not be possible to make it as expressive as the model configuration space definitions in a programming language as Python.

¹https://github.com/Berberer/frankensteins-automl/tree/master/res/search_space

In a first step of answering the first research question, the 4 state-of-the-art approaches as well as frankensteins-automl have a timeout of one hour for selecting and configuring a pipeline. For the dataset inputs in the experiments, nine different datasets from the *OpenML* platform [Van+13] were selected, which are from a broad spectrum of different domains:

- CAR²
- CIFAR10SMALL³
- DEXTER⁴
- DOROTHEA⁵
- KRVSKP⁶
- SEMEION⁷
- WAVEFORM⁸
- WINEQUALITY⁹
- YEAST¹⁰

The second step, i.e. the comparison of the influence of the different components of this approach via the three modifications, has a similar setup. Again, the five experiment constraints are applied and the same nine datasets are given as inputs. But to evaluate if the different influences might change with a different timescale, this second experiment has one hour, six hours, and twelve hours as timeouts for the three variants frankensteins-automl, frankenstein-rs and frankenstein-mcts.

During the two experiment stages, frankensteins-automl (in the unmodified version for the second stage) additionally counts the frequencies of utilizing each type of optimizer. With this counting, there is aggregated data of the optimizer utilization for three different timeouts and nine different datasets. The properties of the datasets,

²<https://www.openml.org/d/40975>

³<https://www.openml.org/d/40926>

⁴<https://www.openml.org/d/4136>

⁵<https://www.openml.org/d/4137>

⁶<https://www.openml.org/d/3>

⁷<https://www.openml.org/d/1501>

⁸<https://www.openml.org/d/60>

⁹<https://www.openml.org/d/40498>

¹⁰<https://www.openml.org/d/181>

for example number of classes or datapoints, or balance of datapoints for some classes, are evaluated in the context of the optimizer utilization frequencies to try to extrapolate knowledge about the suitability of the different optimizers for different dataset types to answer the second research question.

For both experiment stages, frankensteins-automl is configured with the following values:

- When a node is expanded during the MCTS, each new child node is scored with the results of three Monte-Carlo simulations.
- Each run of one of the optimizers in their leaf nodes has an optimization time budget of three minutes. In the worst case, where the pipeline evaluation requires the complete five minutes, this would mean that the optimization run would only consist out of one evaluation. However, this is usually not the case for valid pipeline topologies and datasets with a reasonable size.

These values were chosen after some initial experiments to balance the hardware workload according to the experiment hardware and the solution quality. They may not be the best configuration values for every use-case and every input dataset. A more thorough survey of possible configurations for this approach can be a starting point for further research as future work.

For a better reproducibility of this experiments, they are conducted inside of a *Singularity* container [KSB17]. Hence, the results can be reproduced in every environment where Singularity is installed to verify the data or to conduct customized variants of these experiments. The Singularity image recipe as well as some additional information regarding the setup can be found in Appendix A.1.

5.3 Results of the Experiments

The results of these experiments are presented and assessed in the following before they are evaluated in more detail and used to answer the research questions in the following chapter.

At first, Tab. 5.1 shows the benchmark results of frankensteins-automl against the competitor state-of-the-art approaches. The experiment runs were only classified as successful and included in the results if the AutoML approach was able to return a valid pipeline in one hour plus an additional 15 minutes range of tolerance and without throwing an exception or error.

None of the approaches was able to achieve successful runs for all 30 seeds across all nine datasets. But especially Mosaic was very unstable and was not able to achieve successful runs for more than half of the seeds for six out of the nine datasets. Hence, the comparison of frankensteins-automl with Mosaic has not the same significance as the comparison with the other benchmark approaches.

Nevertheless, a check for significant improvement or deterioration of the competitor approaches in comparison to the scores of frankensteins-automl is executed via Welch's t -tests with $p = 0.05$, because equal variances among the samples could not be assumed. The absolute frequencies how often the approaches were significantly better, significantly worse or without a significant difference compared to frankensteins-automl out of the nine datasets are shown in Tab. 5.2.

Additionally, the approach of this thesis is evaluated by itself in more detail. The first evaluated aspect of frankensteins-automl is the usage of the different optimization algorithms of the ensemble.

Tab. 5.3 shows the relative frequency of the optimizers in the context of corresponding datasets as well as a visualization of these frequencies in terms of a heatmap. Here, the data is considered only as relative frequencies because the data is aggregated among all three different timeouts (1h, 6h, 12). Therefore, absolute values are less meaningful because with different timeouts the amount of MCTS iterations and thus the amount of optimization runs differs greatly but relative values are balanced by the higher amount of optimization runs of all optimizers.

After listing the relative optimizer utilization frequencies, this optimizer utilization is considered in the context of the different datasets and their properties in Tab. 5.4. The following general dataset properties are gathered from OpenML for each dataset:

- *Instances*: The number of instances of the dataset.
- *Features*: The number of features that each instance has.
- *Classes*: The number of available target classes the instances are classified into.
- *Dimensionality*: The relationship between features and instances that is calculated via *Features* divided by *Instances*.
- *Autocorrelation*: After assigning consecutive numbers to the classes, the average difference of the classes for successive instances.
- *Minority Class*: How much percent of *Instances* is classified by the minority class.

- *Majority Class*: How much percent of *Instances* is classified by the majority class.
- *Class Entropy*: Information theoretic entropy value of the class values.

These values are examined in terms of a correlation with the optimizer utilization of the five different optimization algorithms. For the sake of completeness, this is here not only done for the relative frequency but also for the absolute frequency and rankings of the five optimizer utilization frequencies as well.

Because linearity and an underlying normal distribution cannot be ensured, since the datasets were deliberately chosen to be very different from one another, the correlation coefficient has to be calculated with the non-parametric Spearman's rank correlation coefficient ρ .

The second evaluated aspect of frankensteins-automl's mechanics is the influence study of the two main components, i.e. model selection via MCTS and model configuration via the warmstarted optimizer ensembles.

This is evaluated with the three variants of frankensteins-automl for three different timeouts in the context of 30 random seeds and the same nine datasets. An analysis for significant improvement or deterioration within one timeout value is here done in the form of a Welch's t -test with $p = 0.05$ as well.

All accuracy score averages as well as the significance analysis results can be seen in Tab. 5.5.

5.4 Analysis of the Experiment Outcomes

This collected data is now analysed in more detail and used to answer the two main research questions of this thesis.

5.4.1 Feasibility of AutoML via an Optimizer Ensembles

The first research question is about the feasibility of this optimizer ensembles approach for AutoML problems.

As observed in Tab. 5.1 and Tab. 5.2, this approach is partially competitive to the compared state-of-the-art approaches. It appears for the evaluated datasets in large parts inferior to auto-sklearn mosaic, where it was only able to achieve competitive scores without a significant deterioration for two respectively three datasets but yields no significant improvements for any dataset. Compared to ML-Plan and TPOT, the benchmark appears more balanced. It shows a minor advantage compared to ML-Plan with significant improvements for five datasets and minor disadvantages

compared to TPOT with significant improvements for only one dataset.

Based on these results, the current reference implementation of the approach of this thesis shows no significant improvements against the established state-of-the-art approaches. However, it was able to yield partially competitive results and this could indicate the ability to close up to the current state-of-the-art with further developments and refinements.

As part of the feasibility question, it is also evaluated which influences the two core components of this approach, i.e. model selection and optimizer selection as a Multi-Armed Bandit problem tackled via MCTS as well as model configuration approached with warmstarted optimizer ensembles, have on the results. The results of this evaluation can be found in Tab. 5.5.

For this experiment, one of the components is replaced with an alternative default behavior. At first, a random search instead of MCTS but all optimizers included. Second, only Bayesian optimization but reached via an MCTS. Both were seen as competitors to the unmodified approach with an MCTS and all optimizers.

Unfortunately, across all three evaluated timeouts, there are only four occurrences of one of the modified variants having significantly different scores compared to the unmodified and complete variant. Additionally, these four significant differences do not suggest any kind of pattern and are most likely just outliers.

Solely based on the evaluated datasets and timeouts, this data can be interpreted in two ways. Either, the approach appears to be powered by a combination of both components and is not clearly influenced by one more than the other. Alternatively, it is also possible that one of the more sophisticated components of this approach is not required and could be replaced by one of the simplified alternatives, random search instead of MCTS or only Bayesian optimization instead of optimizer ensembles. A broader influence study is necessary to decide with interpretation is more accurate.

All in all, the current reference implementation does not indicate feasibility of the optimizer ensemble approach of this thesis under the constraints of the experiments. It has to be noted, that these constraints prevented the approach from its ability to construct pipeline topologies with arbitrary size and complexity. Additionally, the other approaches, especially auto-sklearn and TPOT, haven been in development for a significantly longer time and have therefore reached a much higher level of implementation maturity and code optimization.

But since the results were at least partially competitive, it is not completely ruled out, that this idea can become feasible when this reference implementation has reached a comparable degree of maturity and optimization as auto-sklearn and TPOT and the some of the future work ideas are incorporated.

The apparently similar influence of both components on the overall behavior does not give a clear suggestion, on which of these two components this future research should be focussed. More evaluations and experimentation with modifications and

additions in both components will be necessary. Furthermore, this approach should be evaluated in an experimental setting where the pipeline topology and length are not constrained especially against other approaches that are able to construct such pipelines as well, i.e. ML-Plan and TPOT.

5.4.2 Optimizer Utilization Frequencies

Since an MCTS as a method to tackle a Multi-Armed Bandit problem is used to select the optimizer, it will try to explore the most suitable optimizer for each problem class/instance and exploit it. Hence, if the timeout is long enough and MCTS was able to perform a sufficient exploration, there should be a visible trend towards one or more optimizers for the problem class, i.e. an input dataset.

This inherent mechanism of gathering indications about the suitability of an optimization algorithm towards the optimization of parametrizations of constructed pipelines for different datasets is the basis for the second research question.

It should be examined if there are recognizable trends or tendencies towards one or more optimization algorithms across all datasets or otherwise if it is correlated to properties of the corresponding dataset which optimizer was utilized with which frequency.

As presented in Tab. 5.3, SMAC, i.e. an implementation of Bayesian optimization, was utilized most often for each of the datasets. Although Hyperband and the Genetic Algorithm came more or less close regarding the relative utilization frequency, they appear to be selected less favorably by the MCTS. Both random search, as well as discretization search, trail significantly far behind the other three and have really low utilization frequency with a maximum of around 15%.

If a ranking is created along with the five overall relative utilizations it would look like the following:

1. SMAC
2. Genetic Algorithm
3. Hyperband
4. Discretization Search
5. Random Search

This ranking is interesting in two ways. First, since Hyperband is more or less a more sophisticated random search, the three least suitable optimization algorithms are all

based on search algorithms.

Second, auto-sklearn as well as Mosaic use internally Bayesian Optimization, TPOT a genetic algorithm, and ML-Plan a more complex form of Discretization Search. When this ranking is now compared to the results of these four AutoML approaches in Tab. 5.1, there is a similarity to be observed as auto-sklearn, Mosaic, and TPOT being at the top with comparable scores, but clearly superior to ML-Plan for the evaluated datasets in this experiment series. Theoretically, this would be another match of findings for the benchmark scores in comparison to the optimizer utilization ranking. But this is probably all in all solely a coincidence and not caused by the actual choice of optimization algorithm, because the three approaches perform more competitive in other benchmark evaluations with longer timeouts [MWH18] than in this evaluation.

Since there is a clear trend recognizable regarding the most favorable optimizer choices of the MCTS across all datasets of this experiment, the evaluation regarding correlations between dataset properties and optimizer utilization becomes less relevant but is done nevertheless.

As presented in Tab. 5.5, there are only weak correlation coefficients between all dataset properties and optimizer utilization across all three utilization metrics. The correlation coefficients are overall in the range $-0.1662 \leq \rho \leq 0.2286$, which does not suggest any significant correlation altogether.

These interpreted results suggest that SMAC is the most suitable choice of optimization algorithm across all regarded datasets independently of the eight evaluated dataset properties. If this is a trend beyond the scope of this evaluation, has to be controlled in an evaluation with a much higher amount of different datasets.

Tab. 5.1.: Results of the benchmark experiments for comparing frankensteins-automl with other state-of-the-art AutoML approaches. The individual cells have the following format: average \pm standard deviation (amount of successful experiment runs). Additionally, there is a \uparrow or \downarrow if the score of the compared AutoML approach is significantly better or significantly worse compared to frankensteins-automl. This statistical significance is tested via Welch's t -tests with $p = 0.05$. The highest average score for each dataset is printed in bold. If the AutoML approach had not a single successful experiments run for the corresponding dataset, the cell is filled with a $/$.

	frankensteins-automl	autosklearn	mlplan	mosaic	tpot
car	$0.9789 \pm 0.0426(30)$	$0.9874 \pm 0.0065(26)$	$0.4430 \pm 0.2452(19) \downarrow$	$0.9777 \pm 0.0145(09)$	$0.9900 \pm 0.0092(30)$
cifar	$0.3140 \pm 0.0621(28)$	$0.3997 \pm 0.0068(28) \uparrow$	$0.4226 \pm 0.0080(20) \uparrow$	$0.3663 \pm 0.0086(27) \uparrow$	$0.2729 \pm 0.0481(28) \downarrow$
dexter	$0.9004 \pm 0.0453(28)$	$0.9263 \pm 0.0189(30) \uparrow$	$0.9447 \pm 0.0170(20) \uparrow$	$0.9509 \pm 0.0165(22) \uparrow$	$0.9289 \pm 0.0231(30) \uparrow$
dorothea	$0.9245 \pm 0.0136(30)$	$0.9332 \pm 0.0085(30) \uparrow$	$/$	$0.9469 \pm 0.0124(28) \uparrow$	$0.9275 \pm 0.0117(26)$
krvskp	$0.9933 \pm 0.0023(27)$	$0.9925 \pm 0.0034(25)$	$0.5017 \pm 0.0171(19) \downarrow$	$0.9932 \pm 0.0022(03)$	$0.9934 \pm 0.0024(30)$
semeion	$0.9235 \pm 0.0588(30)$	$0.9468 \pm 0.0125(29) \uparrow$	$0.1538 \pm 0.0375(20) \downarrow$	$0.9423 \pm 0.0102(12)$	$0.9356 \pm 0.0129(30)$
waveform	$0.8412 \pm 0.0278(29)$	$0.8588 \pm 0.0074(29) \uparrow$	$0.8645 \pm 0.0079(17) \uparrow$	$0.8686 \pm 0.0079(05) \uparrow$	$0.8606 \pm 0.0075(30) \uparrow$
wine	$0.5292 \pm 0.1629(28)$	$0.6329 \pm 0.0096(29) \uparrow$	$0.6601 \pm 0.0123(20) \uparrow$	$0.6446 \pm 0.0116(17) \uparrow$	$0.6614 \pm 0.0127(30) \uparrow$
yeast	$0.5752 \pm 0.0592(30)$	$0.6009 \pm 0.0169(30) \uparrow$	$0.2987 \pm 0.0623(20) \downarrow$	$0.6126 \pm 0.0044(02) \uparrow$	$0.6048 \pm 0.0180(30) \uparrow$

Tab. 5.2.: Absolute frequencies of significant improvements or deterioration of the different benchmark approaches compared to frankensteins-automl for the different datasets. For the two cases, where the benchmark approach was not able to produce a single result, this is counted as *Significantly worse* compared to frankensteins-automl, because no valid result is basically a score of 0.

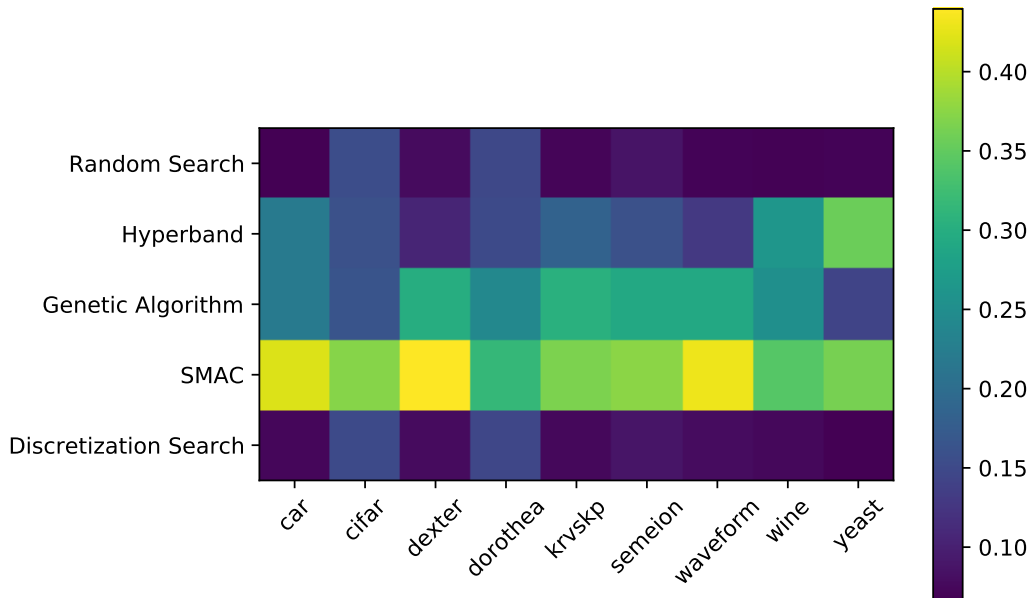
	<i>Significantly better</i>	<i>Significantly worse</i>	<i>No significant difference</i>
autosklearn	7	0	2
mlplan	4	5	0
mosaic	6	0	3
tpot	4	1	4

Tab. 5.3.: The relative frequency of optimizer calls for the different datasets.

(a) Numerical values with four decimal places. The highest frequency for each dataset is printed in bold. RS = Random Search, HB = Hyperband, GA = Genetic Algorithm, DS = Discretization Search

	<i>RS</i>	<i>HB</i>	<i>GA</i>	<i>SMAC</i>	<i>DS</i>
car	0.0679	0.2185	0.2194	0.4202	0.0740
cifar	0.1545	0.1591	0.1640	0.3721	0.1501
dexter	0.0780	0.1052	0.2991	0.4396	0.0781
dorothea	0.1470	0.1502	0.2415	0.3153	0.1460
krvskp	0.0717	0.1835	0.3036	0.3666	0.0746
semeion	0.0860	0.1596	0.2913	0.3748	0.0884
waveform	0.0708	0.1291	0.2918	0.4294	0.0789
wine	0.0692	0.2622	0.2524	0.3419	0.0742
yeast	0.0701	0.3560	0.1434	0.3636	0.0669

(b) Visualization of the frequencies as a heatmap.



Tab. 5.4.: Analysis of possible correlations between optimizer utilization and dataset properties via the Spearman's rank correlation coefficient ρ with four decimal places. RS = Random Search, HB = Hyperband, GA = Genetic Algorithm, DS = Discretization Search

(a) Relative utilization frequency.

	<i>RS</i>	<i>HB</i>	<i>GA</i>	<i>SMAC</i>	<i>DS</i>
<i>Instances</i>	0.0223	0.0051	0.0044	0.0080	0.0294
<i>Features</i>	0.2286	−0.0560	0.1109	0.0087	0.2091
<i>Classes</i>	−0.0275	0.0899	−0.0780	−0.0165	−0.0208
<i>Dimensionality</i>	0.2027	−0.0323	0.0903	0.0053	0.1799
<i>Autocorrelation</i>	−0.0906	−0.0621	0.0002	−0.0508	−0.0746
<i>Minority Class</i>	0.0643	−0.1615	0.1050	0.0496	0.0494
<i>Majority Class</i>	0.0039	−0.0224	0.0359	−0.0081	0.0060
<i>Class Entropy</i>	−0.0315	0.0476	−0.0587	−0.0053	−0.0250

(b) Absolute utilization frequency.

	<i>RS</i>	<i>HB</i>	<i>GA</i>	<i>SMAC</i>	<i>DS</i>
<i>Instances</i>	0.0188	0.0155	0.0038	0.0179	0.0284
<i>Features</i>	0.0724	−0.1501	−0.0533	−0.0782	0.0563
<i>Classes</i>	0.0349	0.1146	0.0026	0.0328	0.0482
<i>Dimensionality</i>	0.0660	−0.1232	−0.0531	−0.0714	0.0464
<i>Autocorrelation</i>	0.0084	−0.0103	0.0254	−0.0214	0.0075
<i>Minority Class</i>	0.0184	−0.1662	0.0001	−0.0006	0.0075
<i>Majority Class</i>	−0.0730	−0.0694	−0.0276	−0.0574	−0.0762
<i>Class Entropy</i>	0.0496	0.0938	0.0214	0.0499	0.0619

(c) Optimizer utilization ranking.

	<i>RS</i>	<i>HB</i>	<i>GA</i>	<i>SMAC</i>	<i>DS</i>
<i>Instances</i>	−0.0259	0.0030	0.0171	0.0202	−0.0319
<i>Features</i>	−0.0163	0.1286	0.0418	0.0279	−0.0750
<i>Classes</i>	−0.0043	−0.0894	0.0673	0.0168	−0.0065
<i>Dimensionality</i>	−0.1409	0.0989	0.0432	0.0190	−0.0520
<i>Autocorrelation</i>	0.0167	0.0548	−0.0415	−0.0074	−0.0219
<i>Minority Class</i>	−0.0745	0.1617	−0.0381	−0.0449	−0.0273
<i>Majority Class</i>	0.0485	0.0253	−0.0613	0.0047	0.0113
<i>Class Entropy</i>	0.0191	−0.0541	0.0598	0.0043	−0.0153

Tab. 5.5.: Results of the experiments for comparing the three frankensteins-automl variants over longer timeouts. The individual cells have the same layout as in Tab. 5.1.

(a) Timeout: 1h

	frankensteins-automl	frankenstein-rs	frankenstein-mcts
car	$0.9789 \pm 0.0426(30)$	$0.9871 \pm 0.0148(30)$	$0.9633 \pm 0.0674(30)$
cifar	$0.3140 \pm 0.0621(28)$	$0.2841 \pm 0.0632(28)$	$0.2664 \pm 0.0768(30) \downarrow$
dexter	$0.9004 \pm 0.0453(28)$	$0.9020 \pm 0.0552(28)$	$0.9090 \pm 0.0426(29)$
dorothea	$0.9245 \pm 0.0136(30)$	$0.9183 \pm 0.0235(29)$	$0.9197 \pm 0.0140(29)$
krvskp	$0.9933 \pm 0.0023(27)$	$0.9872 \pm 0.0301(30)$	$0.9931 \pm 0.0025(29)$
semeion	$0.9235 \pm 0.0588(30)$	$0.9317 \pm 0.0452(30)$	$0.9221 \pm 0.0533(30)$
waveform	$0.8412 \pm 0.0278(29)$	$0.8420 \pm 0.0313(30)$	$0.8124 \pm 0.1066(30)$
wine	$0.5292 \pm 0.1629(28)$	$0.5709 \pm 0.1327(27)$	$0.6218 \pm 0.0890(29) \uparrow$
yeast	$0.5752 \pm 0.0592(30)$	$0.5594 \pm 0.0822(28)$	$0.5323 \pm 0.0933(29) \downarrow$

(b) Timeout: 6h

	frankensteins-automl	frankenstein-rs	frankenstein-mcts
car	$0.9714 \pm 0.0577(30)$	$0.9926 \pm 0.0068(30)$	$0.9729 \pm 0.0602(30)$
cifar	$0.3048 \pm 0.0652(30)$	$0.3025 \pm 0.0682(29)$	$0.3210 \pm 0.0542(30)$
dexter	$0.8904 \pm 0.1030(29)$	$0.9044 \pm 0.0600(29)$	$0.9222 \pm 0.0255(29)$
dorothea	$0.9094 \pm 0.1078(29)$	$0.9086 \pm 0.1108(30)$	$0.9218 \pm 0.0173(30)$
krvskp	$0.9930 \pm 0.0022(29)$	$0.9920 \pm 0.0086(30)$	$0.9926 \pm 0.0022(30)$
semeion	$0.9070 \pm 0.1576(30)$	$0.9487 \pm 0.0206(30)$	$0.9093 \pm 0.0860(30)$
waveform	$0.8500 \pm 0.0241(30)$	$0.8480 \pm 0.0322(29)$	$0.8362 \pm 0.0423(30)$
wine	$0.6293 \pm 0.0985(29)$	$0.6510 \pm 0.0403(28)$	$0.6326 \pm 0.0612(30)$
yeast	$0.5759 \pm 0.0758(30)$	$0.5697 \pm 0.0892(30)$	$0.5850 \pm 0.0648(29)$

(c) Timeout: 12h

	frankensteins-automl	frankenstein-rs	frankenstein-mcts
car	$0.9838 \pm 0.0187(30)$	$0.9940 \pm 0.0057(30) \uparrow$	$0.9766 \pm 0.0556(30)$
cifar	$0.3316 \pm 0.0467(30)$	$0.3319 \pm 0.0497(30)$	$0.3069 \pm 0.0645(30)$
dexter	$0.9265 \pm 0.0154(30)$	$0.9296 \pm 0.0128(30)$	$0.9285 \pm 0.0127(29)$
dorothea	$0.9303 \pm 0.0136(30)$	$0.9003 \pm 0.1494(30)$	$0.9242 \pm 0.0205(30)$
krvskp	$0.9928 \pm 0.0042(30)$	$0.9933 \pm 0.0025(29)$	$0.9904 \pm 0.0113(30)$
semeion	$0.9501 \pm 0.0173(30)$	$0.9411 \pm 0.0346(30)$	$0.9498 \pm 0.0236(30)$
waveform	$0.8547 \pm 0.0106(30)$	$0.8575 \pm 0.0165(30)$	$0.8551 \pm 0.0095(30)$
wine	$0.6574 \pm 0.0404(30)$	$0.6508 \pm 0.0793(30)$	$0.6491 \pm 0.0764(30)$
yeast	$0.6028 \pm 0.0213(30)$	$0.5995 \pm 0.0209(29)$	$0.5981 \pm 0.0245(30)$

Conclusion

In this final chapter after previously answering the concrete research questions of the thesis, the prior descriptions, and findings regarding the optimizer ensemble AutoML approach are summarized and concluded with an outlook towards possible future works.

6.1 Summary and Findings

This thesis was based on the statements of the No-Free-Lunch theorems for optimization, which had proven that under given constraints no optimization algorithm can be superior for all problems. To approach this problem, the model selection of an AutoML setting combined with choosing an optimizer that should perform a model configuration for the selected pipeline model was seen as a hierarchical Multi-Armed Bandit problem. Hence, it was tackled with an MCTS, which searches a model selection graph that is realized as an HTN planning graph with embedded optimizers, such that the optimizers can be utilized independently from each other depending on the suitability for the model configuration space of the constructed pipeline. These optimizers are combined as an ensemble to utilize the gathered information from each other about evaluated candidates and perform successive warmstarting for each optimization run.

This novel AutoML idea was realized as a first simple reference implementation, which afterwards was used in a series of experiments to examine the feasibility of the approach and to gather additional knowledge about the functionality of the components.

In its current reference implementation, the approach of this thesis was not able to surpass the currently best state-of-the-art approaches for the examined datasets. However, it was able to show competitive results for some of the datasets and a few significant improvements against TPOT and ML-Plan. This could indicate the potential to close the current gap between the other tools or even surpass them.

6.2 Open Questions and Future Work

Besides the mentioned further development regarding general implementation maturity and optimization, some other factors, which could lead to better results, were not covered by this thesis and are good starting points for future work. To conclude this chapter, they are listed in the following, grouped into different categories:

- **Configuration:** The evaluation was solely done with one configuration of this approach. Although the utilized configuration showed the best results in some tests prior to the experiments, there could exist better configurations, for example regarding the number of Monte-Carlo simulations or the time budget of each single optimization run. A thorough study across a variety of possible configurations in the context of different AutoML timeouts and more datasets than the evaluated nine can be a first next step to identify better configurations.

Additionally, the configuration of this approach remained static during the entire MCTS search. Modifying the configuration values during the search could improve the balance between exploration and exploitation. For example, the number of Monte-Carlo simulations could start with a high value and be incrementally reduced during the run, such that a higher exploration coverage of the search space is reached in the beginning. Or alternatively, the optimization time budget could start with a low value and be improved later in the search, once the probability of exploiting more suitable optimizers is higher after a certain degree of exploration. The adjustment of time budgets could utilize one of the time allocation policies of Quemy [Que19], which were presented in section 2.4.

- **Model Selection Search:** During the evaluation of different configurations, it could also prove beneficial to evaluate different search algorithms instead of an MCTS. Since the variant of this approach that used a random search instead of MCTS was not significantly worse, other search algorithms should also be considered and tested.

Furthermore, this approach utilized UCT, but there are also other published MCTS policies for scoring nodes and selecting them accordingly. A survey of a broad selection of such policies was for example done by Browne et al. [Bro+12]. Besides completely different search algorithms, a study of different MCTS variations and policies could also be a valid starting point for future research.

- **Optimizer Ensemble Model Configuration:** In the case of the evaluated datasets, MCTS rarely exploited a random search or discretization search for

optimization in comparison to the other optimizers. If this is also the case during a broader study of more datasets, the subset of optimizers that are selected for the ensemble, could be reduced to the remaining three optimizers, or alternatively, the two optimizers could be replaced by other optimization algorithms, which were not integrated and evaluated yet.

Besides the selection of optimizers for the ensembles, the interaction within the ensembles is also a departure point for future research. It has to be evaluated if every optimizer really profits from warmstarting or if this leads for example to an over-searching of local optima.

Similar to adjusting the configuration dynamically during the run, it could also be a variation to enable or disable warmstarting for different time periods of the overall search or for different parts of the search graph. In the same manner, early stopping for all or for some optimizers could be added as a functionality and then dynamically be enabled and disabled during the run, if there is a chance for a better budget allocation towards more promising optimizers.

Additionally, the possibilities for creating the model configuration optimization spaces can be extended to a degree of the expressiveness of auto-sklearn's space modeling features. Constraints across multiple parameters as for example, *"If value a is selected for parameter p_1 , value b is no longer an allowed value for parameter p_2 "* could prevent this approach from evaluating unsuitable or even invalid pipeline parametrization candidates.

As part of this thesis it was only evaluated if the dataset properties had any correlations to the optimizer utilization, but the same could be done concerning the components and the topology of the pipeline. For example, if one optimization algorithm would be utilized more often for shorter pipelines or alternatively for example, if the learning algorithm is a decision tree, an optimizer selection policy could integrate such insights.

- **Problem Settings:** In theory, the approach of this thesis has no limitations regarding the desired functionality of the algorithm selection and hyperparameter optimization outcome. This thesis focusses on AutoML as one specific outcome, but with minor modifications regarding problem description input and internal candidate evaluation, it can construct models for other combined algorithm selection and configuration domains as well.

Once the above mentioned improvements were made and these minor modifications for allowing other domains as well, an interesting follow-up research question is to evaluate how the optimizer ensemble approach performs in other domains besides AutoML.

Appendix

A.1 Setup and Singularity Definition File of the Experiments

The evaluation utilizes a Python library for experiment execution. It requires a definition of all possible input parameters for one experiment run and the desired result values.

Experiment executions themselves, are orchestrated via one central database. The experiment runner will select one possible parametrization of the allowed input parameter values, check via the database if this parametrization was already evaluated and perform the evaluation of the parametrization otherwise. With this experiment management via a single database, the actual experiment execution can be parallelized arbitrarily. Further details about the libraries setup and functionality can be found in the corresponding repository¹.

The setup for the evaluations with this Python experiment environment is explained in the following. Every time a certain file name is assumed for the setup it is given as well, but this can of course be customized for a different setup.

The experiment runner has to be configured with the information regarding the database connection and the values for possible parametrization and expected results. For example in the case of the experiments with the timeout of one hour to compare the different approaches regarding their accuracy scores and additionally to count the optimizer runs of frankensteins-automl, the experiment configuration file (`experiment.properties`) could look like the following:

```
keyfields = timeout,seed,dataset,algorithm
resultfields = score,random_count,hyperband_count,genetic_count,
               smac_count,discretization_count

# Keyfield values(i.e. values for input parametrizations)
timeout=3600
seed=1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,
    26,27,28,29,30
algorithm=tpot,autosklearn,mosaic,frankenstein
dataset=car-6,cifar-3072,dexter-20000,dorothea-100000,krvskp-36,
```

¹<https://github.com/Berberer/python-experimenter>

```

semeion-256,waveform-40,wine-11,yeast-8

# Database connection setup values
db.host = 192.168.0.1
db.type = MYSQL
db.username = experimenter
db.password = password123
db.database = experiments
db.table = automl

```

Here, the datasets have a number attached to their name, to indicate the index of the column with the prediction target class.

This experiment configuration file is given as an input to the following Python script (experiment.py) that defines how the experiments for a single parametrization are actually conducted and then starts the execution of experiments:

```

import resource
from python_experimenter.experiment_runner import ExperimentRunner
from tpot import TPOTClassifier
from autosklearn.classification import AutoSklearnClassifier
from mosaic_ml.automl import AutoML
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from frankensteins_automl.frankensteins_automl import (
    FrankensteinsAutoMLConfig,
    FrankensteinsAutoML,
)
from frankensteins_automl.machine_learning.arff_reader import read_arff
from frankensteins_automl.optimizers.baysian.smac_optimizer import SMAC

def automl_experiment(keyfields):
    # Get experiment parametrization
    algorithm = keyfields["algorithm"]
    dataset = keyfields["dataset"]
    seed = int(keyfields["seed"])
    timeout = int(keyfields["timeout"])

    # Set results to dummy values
    results = {}
    results["random_count"] = -1
    results["hyperband_count"] = -1
    results["genetic_count"] = -1
    results["smac_count"] = -1
    results["discretization_count"] = -1
    results["score"] = -1

    # Read input dataset and perform stratified split
    parts = dataset.split("-")
    dataset_file = parts[0]
    class_index = int(parts[1])
    data_x, data_y, _, _ = read_arff(
        f"res/datasets/{dataset_file}.arff",
        class_index
    )
    x_train, x_test, y_train, y_test = train_test_split(

```

```

        data_x,
        data_y,
        test_size=0.3,
        random_state=seed,
        stratify=data_y,
    )
score = None

# Configure and execute approach of this experiments run
if algorithm == "tpot":
    tpot_automl = TPOTClassifier(
        generations=None,
        random_state=seed,
        max_time_mins=int(timeout / 60.0)
    )
    tpot_automl.fit(x_train, y_train)
    score = tpot_automl.score(x_test, y_test)
elif algorithm == "autosklearn":
    autosklearn_automl = AutoSklearnClassifier(
        time_left_for_this_task=timeout,
        seed=seed,
        ml_memory_limit=16384
    )
    autosklearn_automl.fit(x_train, y_train)
    predictions = autosklearn_automl.predict(x_test)
    score = accuracy_score(predictions, y_test)
elif algorithm == "mosaic":
    mosaic_automl = AutoML(
        time_budget=timeout,
        seed=seed,
        scoring_func="accuracy",
        memory_limit=59392,
    )
    _, score = mosaic_automl.fit(
        x_train,
        y_train,
        x_test,
        y_test,
        categorical_features=["numerical"]*len(x_train[0])
    )
else:
    config = FrankensteinsAutoMLConfig()
    config.direct_data_input(x_train, y_train)
    config.timeout_in_seconds = float(timeout)
    config.timeout_for_optimizers_in_seconds = 180.0
    config.timeout_for_pipeline_evaluation = 300.0
    config.simulation_runs_amount = 3
    config.random_seed = seed

# Select frankensteins-automl variant
if algorithm == "frankenstein":
    config.count_optimizer_calls = True
elif algorithm == "frankenstein-rs":
    config.random_node_selection = True
elif algorithm == "frankenstein-mcts":
    config.optimizers = [SMAC]

# Return accuracy and optimizer run counts if available
e = None

```

```

r = None
try:
    frankensteins_automl = FrankensteinsAutoML(config)
    r = frankensteins_automl.run()
except Exception as exception:
    e = exception
if r is not None:
    pipeline = r["pipeline_object"]
    if pipeline is not None:
        pipeline.fit(x_train, y_train)
        predictions = pipeline.predict(x_test)
        score = accuracy_score(predictions, y_test)
    if algorithm == "frankenstein":
        if "optimizer_calls" in r:
            oc = r["optimizer_calls"]
            if "RandomSearch" in oc:
                results["random_count"] = oc["RandomSearch"]
            if "Hyperband" in oc:
                results["hyperband_count"] = oc["Hyperband"]
            if "GeneticAlgorithm" in oc:
                results["genetic_count"] = oc["GeneticAlgorithm"]
            if "SMAC" in oc:
                results["smac_count"] = oc["SMAC"]
            if "DiscretizationSearch" in oc:
                results["discretization_count"] = (
                    oc["DiscretizationSearch"]
                )
        elif e is not None:
            raise e

if score is not None:
    results["score"] = score

return results

# Start the experiment execution with the here defined evaluation method
runner = ExperimentRunner(automl_experiment, "experiment.properties")
runner.run()

```

This experiment script can be used for the experiments of all research questions of this thesis with an appropriate configuration in the corresponding `experiment.properties` file as for example the one aforementioned.

At last, this script has to be executed in a reproducible Python environment. For instance, this can be achieved with Singularity containers [KSB17]. The Singularity recipe for the container of this experiments is the following:

```

BootStrap: library
From: ubuntu:20.04

%files
    # Approach of this thesis to be copied into the container
    frankensteins-automl /experiment/

%post

```



```

# Bring Ubuntu up to date
apt-get -y update
apt-get -y upgrade
# Install Python and different required libraries
apt-get -y install python3
apt-get -y install python3-dev
apt-get -y install python3-distutils
apt-get -y install wget
apt-get -y install git
apt-get -y install bison
apt-get -y install flex
apt-get -y install curl
apt-get -y install build-essential
apt-get -y install autotools-dev
apt-get -y install automake
apt-get -y install libpcrc3-dev
apt-get -y install gcc
apt-get -y install g++
# Install pip
wget https://bootstrap.pypa.io/get-pip.py
python3 get-pip.py
pip install --upgrade pip setuptools wheel Cython numpy scipy
# Build and install swig
git clone https://github.com/swig/swig.git --branch=v3.0.8
cd swig
./autogen.sh
./configure
make
make install
# Install approach of this thesis locally in the container
cd /experiment/frankensteins-automl
pip install -r requirements.txt
pip install .
cd ..
# Clone and install experiment runner locally in the container
git clone https://github.com/Berberer/python-experimenter.git
cd python-experimenter
pip install -r requirements.txt
pip install .
cd ..
# Install the benchmark approaches and their dependencies
pip install tpot
a="https://raw.githubusercontent.com/automl/auto-sklearn/master/requirements.txt"
curl ${a} | xargs -n 1 -L 1 pip install
pip install auto-sklearn
pip install git+https://github.com/herilalaina/mosaic@0.1
pip install git+https://github.com/herilalaina/mosaic_ml

```

This recipe assumes that the image is build in a directory where the reference implementation is located in a folder called `frankensteins-automl` to allow direct changes and customizations of the code. Alternatively, this can be easily changed in the recipe to directly clone it from GitHub.

After building the image (`experiment.sif`), the experiments can be started for example via singularity `exec experiment.sif python3 experiment.py`.

This execution assumes the following file setup in the same directory where the built container is located and used:

```
/
├── experiment.properties
├── experiment.py
├── experiment.sif
├── res
│   ├── config
│   │   └── logging.conf
│   ├── searchspace
│   │   ├── frankensteins_automl_classifiers.json
│   │   ├── frankensteins_automl_data_preprocessors.json
│   │   ├── frankensteins_automl_feature_preprocessors.json
│   │   └── frankensteins_automl_topologies.json
│   └── datasets
│       ├── dorothea.arff
│       ├── wine.arff
│       ├── car.arff
│       ├── krvskp.arff
│       ├── yeast.arff
│       ├── cifar.arff
│       ├── semeion.arff
│       ├── dexter.arff
│       └── waveform.arff
```

The `res/searchspace/*.json` files can be downloaded from https://github.com/Berberer/frankensteins-automl/tree/master/res/search_space. Additionally, `res/logging/logging.conf` is used to configure the logging of frankensteins-automl with the usual Python logging configuration and can be set to the desired values. Finally, `res/datasets/*.arff` are the dataset files listed in section 5.2.

Bibliography

- [ACF02] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. „Finite-time Analysis of the Multiarmed Bandit Problem“. In: *Mach. Learn.* 47.2-3 (2002), pp. 235–256 (cit. on p. 46).
- [Bre96] Leo Breiman. „Bagging Predictors“. In: *Machine Learning* 24.2 (1996), pp. 123–140 (cit. on p. 9).
- [Bro+12] C. B. Browne, E. Powley, D. Whitehouse, et al. „A Survey of Monte Carlo Tree Search Methods“. In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (2012), pp. 1–43 (cit. on p. 90).
- [BV14] Stephen P. Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2014 (cit. on p. 13).
- [Feu+15] Matthias Feurer, Aaron Klein, Katharina Eggenberger, et al. „Efficient and Robust Automated Machine Learning“. In: *Advances in Neural Information Processing Systems* 28. Ed. by C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett. Curran Associates, Inc., 2015, pp. 2962–2970 (cit. on p. 28).
- [Fra18] Peter I. Frazier. „A Tutorial on Bayesian Optimization“. In: *CoRR* abs/1807.02811 (2018). arXiv: 1807.02811 (cit. on p. 25).
- [GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. [http : //www.deeplearningbook.org](http://www.deeplearningbook.org). MIT Press, 2016 (cit. on p. 6).
- [Gol90] David E. Goldberg. „A Note on Boltzmann Tournament Selection for Genetic Algorithms and Population-Oriented Simulated Annealing“. In: *Complex Systems* 4.4 (1990) (cit. on p. 53).
- [HHL11] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. „Sequential Model-Based Optimization for General Algorithm Configuration“. In: *Learning and Intelligent Optimization - 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers*. Ed. by Carlos A. Coello Coello. Vol. 6683. Lecture Notes in Computer Science. Springer, 2011, pp. 507–523 (cit. on p. 27).
- [JT15] Kevin G. Jamieson and Ameet Talwalkar. „Non-stochastic Best Arm Identification and Hyperparameter Optimization“. In: *CoRR* abs/1502.07943 (2015). arXiv: 1502.07943 (cit. on p. 17).

- [KS06] Levente Kocsis and Csaba Szepesvári. „Bandit Based Monte-Carlo Planning“. In: *Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings*. Ed. by Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou. Vol. 4212. Lecture Notes in Computer Science. Springer, 2006, pp. 282–293 (cit. on pp. 33, 47).
- [KSB17] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. „Singularity: Scientific containers for mobility of compute“. In: *PLOS ONE* 12.5 (May 2017), pp. 1–20 (cit. on pp. 78, 96).
- [LH18] Marius Lindauer and Frank Hutter. „Warmstarting of Model-Based Algorithm Configuration“. In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*. 2018, pp. 1355–1362 (cit. on p. 53).
- [Li+16] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. „Efficient Hyperparameter Optimization and Infinitely Many Armed Bandits“. In: *CoRR* abs/1603.06560 (2016). arXiv: 1603.06560 (cit. on p. 16).
- [Mit97] Tom M. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997 (cit. on p. 6).
- [MWH18] Felix Mohr, Marcel Wever, and Eyke Hüllermeier. „ML-Plan: Automated machine learning via hierarchical planning“. In: *Machine Learning* 107.8-10 (2018), pp. 1495–1515 (cit. on pp. 21, 75, 83).
- [OM16] Randal S. Olson and Jason H. Moore. „TPOT: A Tree-based Pipeline Optimization Tool for Automating Machine Learning“. In: *Proceedings of the 2016 Workshop on Automatic Machine Learning, AutoML 2016, co-located with 33rd International Conference on Machine Learning (ICML 2016), New York City, NY, USA, June 24, 2016*. 2016, pp. 66–74 (cit. on p. 23).
- [Ped+11] F. Pedregosa, G. Varoquaux, A. Gramfort, et al. „Scikit-learn: Machine Learning in Python“. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on p. 66).
- [Pla98] John Platt. *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines*. Tech. rep. MSR-TR-98-14. Apr. 1998, p. 21 (cit. on p. 51).
- [Que19] Alexandre Quemy. „Two-stage Optimization for Machine Learning Workflow“. In: *CoRR* abs/1907.00678 (2019). arXiv: 1907.00678 (cit. on pp. 32, 90).
- [RSS19] Herilalaina Rakotoarison, Marc Schoenauer, and Michèle Sebag. „Automated Machine Learning with Monte-Carlo Tree Search“. In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, July 2019, pp. 3296–3303 (cit. on p. 33).
- [RW06] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian processes for machine learning*. Adaptive computation and machine learning. MIT Press, 2006 (cit. on p. 24).

- [Sá+17] Alex Guimarães Cardoso de Sá, Walter José G. S. Pinto, Luiz Otávio Vilas Boas Oliveira, and Gisele L. Pappa. „RECIPE: A Grammar-Based Framework for Automatically Evolving Classification Pipelines“. In: *Genetic Programming - 20th European Conference, EuroGP 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings*. 2017, pp. 246–261 (cit. on p. 23).
- [SLB19] Xudong Sun, Jiali Lin, and Bernd Bischl. „ReinBo: Machine Learning Pipeline Conditional Hierarchy Search and Configuration with Bayesian Optimization Embedded Reinforcement Learning“. In: *Machine Learning and Knowledge Discovery in Databases - International Workshops of ECML PKDD 2019, Würzburg, Germany, September 16-20, 2019, Proceedings, Part I*. Ed. by Peggy Cellier and Kurt Driessens. Vol. 1167. Communications in Computer and Information Science. Springer, 2019, pp. 68–84 (cit. on p. 29).
- [Tho+13] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. „AutoWEKA: combined selection and hyperparameter optimization of classification algorithms“. In: *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*. 2013, pp. 847–855 (cit. on pp. 11, 12, 28).
- [Tho33] William R. Thompson. „On the Likelihood that One Unknown Probability Exceeds Another in View of the Evidence of Two Samples“. In: 25.3/4 (Dec. 1933), pp. 285–294 (cit. on p. 46).
- [Van+13] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luís Torgo. „OpenML: networked science in machine learning“. In: *SIGKDD Explorations* 15.2 (2013), pp. 49–60 (cit. on p. 77).
- [Vir+20] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, et al. „SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python“. In: *Nature Methods* 17 (2020), pp. 261–272 (cit. on p. 66).
- [Wit+16] Ian H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. *Data Mining: Practical Machine Learning Tools and Techniques*. 4th ed. Burlington, MA: Morgan Kaufmann, 2016 (cit. on pp. 18, 55).
- [WM97] David H. Wolpert and William G. Macready. „No free lunch theorems for optimization“. In: *IEEE Trans. Evolutionary Computation* 1.1 (1997), pp. 67–82 (cit. on p. 28).

List of Figures

2.1	High-level view of a conceptual AutoML tool.	8
2.2	Example of a more complex machine learning pipeline.	10
2.3	An exemplaric parameter configuration space.	11
2.4	A two dimensional optimization target function.	15
2.5	Visualization of different variants of random search strategies.	17
2.6	Illustration of a grid search covering the search space of the running example systematically.	19
2.7	Search graph of a graph search in the middle of an ongoing search. . .	20
2.8	An example of a two-point cross-over to produce genome g^* from the genomes g_1 and g_2	23
2.9	Simple illustration of acquisition functions for Bayesian optimization. .	26
2.10	Simple visualization of the underlying concepts of the three layer ReinBo model selection candidate space.	31
3.1	An example of a complex machine learning pipeline with an increased length and a non-linear topology.	37
3.2	An example of a partial HTN planning search tree for a simple planning problem.	40
3.3	A complete search graph of an AutoML problem visualized in the HTN context.	42
3.4	A simplified illustration of a possible extended search tree G'	44
3.5	Illustration of the four MCTS steps.	48
4.1	A simplified overview of the implementation components for managing the search space and enabling HTN planning in this space.	59
4.2	A simplified overview of the implementation components for storing the information about a model configuration optimization space and how the different optimizers approach this space.	64
4.3	A simplified overview of the implementation components for searching HTN planning space with the integrated optimizer ensembles via an MCTS in an unified AutoML tool.	71

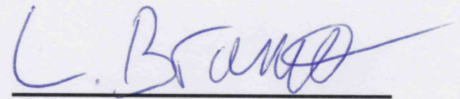
List of Tables

5.1	Results of the benchmark experiments for comparing frankensteins-automl with other state-of-the-art AutoML approaches.	84
5.2	Absolute frequencies of significant improvements or deterioration. . . .	85
5.3	The relative frequency of optimizer calls for the different datasets. . . .	86
5.4	Analysis of possible correlations between optimizer utilization and dataset properties.	87
5.5	Results of the experiments for comparing the three frankensteins-automl variants over longer timeouts.	88

Declaration

I, Lukas Brandt(Matrikel-Nr. 7011823), hereby declare in accordance to § 63 Abs. 5 HZG that I am the sole author of this master thesis and that I have not used any sources besides those referenced and listed in the bibliography. I further declare that I have not submitted this thesis for any other examination or any other institution in order to obtain a degree.

Paderborn, September 14, 2020



Lukas Brandt

