Efficient tuning of automated machine learning pipelines

Nguyen, D.A.

Efficient Tuning of Automated Machine Learning Pipelines

Proefschrift

ter verkrijging van
de graad van doctor aan de Universiteit Leiden,
op gezag van rector magnificus prof.dr.ir. H. Bijl,
volgens besluit van het college voor promoties
te verdedigen op woensdag 9 oktober 2024
klokke 11.30 uur

door

Duc Anh Nguyen
geboren te Thai Binh, Vietnam
in 1987

Promotores:
Prof.dr. T.H.W. Bäck
Prof.dr. B. Sendhoff (Technical University Darmstadt, Germany)

Co-promotor:
Dr. A.V. Kononova

Promotiecommissie:
Prof.dr. M.M. Bonsangue
Prof.dr. A. Plaat
Dr. Jan N. van Rijn
Prof.dr. D. Zaharie (West University of Timisoara, Romania)
Prof.dr. G. Ochoa (University of Sterling, Scotland)
Prof.dr. J. Sun (Xi'an Jiaotong University, China)

This research has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 766186.

Contents

CHAPTER 1

Introduction

In recent years, Machine Learning (ML) has achieved success in many real-world applications, such as self-driving cars [1], assisting doctors in diagnosing diseases [2], playing video games [3], recognizing faces [4], translating languages [5], detecting automatic faults [6], and predictive maintenance [7]. In order to apply ML in real-world applications, practitioners must select a sequence of machine learning algorithms (e.g., data preprocessing, feature preprocessing, learning algorithm), together with their configurations well-suited to the target problem [8]. These tasks are always challenging due to the plethora of algorithms. Moreover, the No Free Lunch (NFL) theorem [9], [10] prescribes that there is no universally best algorithm for all problems. Therefore, the development of high-performance machine-learning applications requires highly skilled ML experts, data scientists and domain experts [11], [12]. Together, these experts attempt further experiments/trials, resulting in the best-performing ML pipeline for the given problem. In other words, applying ML to a real-world application is challenging, as it requires considerable human effort and has a strong dependency on data scientists.

The class of *Automated machine learning* (AutoML) approaches [8] are widely applied to automatically produce the best machine learning pipeline in order to minimize reliance on data scientists in the machine learning development cycle for real-world applications [8]. The domain experts can profit from using AutoML by automatically choosing a well-performing sequence of ML algorithms and their optimized hyperparameters, leading to a sensible ML model for their real-work problems without relying on ML experts.

Initially, AutoML was typically referred to as an optimization process for finding the optimal ML pipeline (with its tuned hyperparameters) for ML problems [13]. Hence, AutoML is known as a combination of Machine Learning and optimization for applying machine learning to real-world problems, that is, making ML easier to use for people without expert knowledge in ML. AutoML research mainly

focuses on optimization, whereas the ML part is inherited from the well-developed ML community. From the perspective of the optimization community, the ML part is treated as an objective configured by various ML algorithms with their hyperparameters and evaluated by an objective function for resulting in a real-valued performance, e.g., prediction accuracy or running time. A search space defines the possible choices among the algorithms and hyperparameters. Different hyperparameter settings led to different results [14]. Then, a practical optimization approach is used to find the best setting that maximizes the performance of the objective function. All the above steps are the core functions of a typical AutoML framework.

On the other hand, AutoML products are also known as a particular type of software in a ready-to-use state where the effectiveness in applying machine learning to real-world problems is the top goal [11]. Thus, modern AutoML systems incorporate many supportive functions, such as supporting the core part (e.g., applying meta-learning to find the better candidate earlier) and supporting humans to increase trust (e.g., explainable ML, visualization), and shortening the final product development cycle in production (e.g., integrating the low/no code techniques).

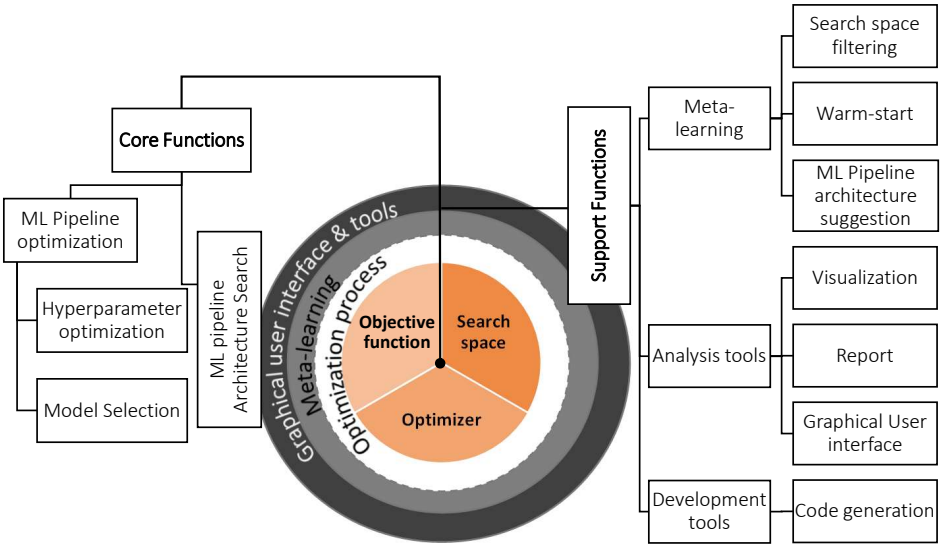


Figure 1.1: AutoML functions

Some of the core functions and support functions of a typical AutoML framework are summarized in Figure 1.1. All are discussed in this thesis, where the machine

learning pipeline optimization approaches are discussed in Chapter 3, and the remaining functions are discussed in Chapter 2, i.e., Meta-learning is discussed in Section 2.3, code generation and analysis tools for AutoML are presented in Section 2.4 and Section 2.2 presents ML pipeline architecture search. However, this thesis emphasizes the optimization of AutoML, called *AutoML optimization*, which is detailed in Section 1.1. Prior to the detailed discussions, we shall give a brief explanation of some important aspects of the AutoML optimization problem (the orange cycle in the middle of Figure 1.1).

Search space or domain space refers to the set of all possible combinations of algorithms and the algorithm's hyperparameters that the optimization algorithm can choose from. There are four common classes of hyperparameters:

1. *Ordinal hyperparameter*[1] belong to a subset of $\mathbb{Z}$, e.g., [1, 10];
2. *Continuous hyperparameter*[1] is a subset of $\mathbb{R}$, e.g., [0, 1];
3. *Nominal hyperparameter* is a set of categorical values, e.g., [Linear, RBF, Poly, Sigmoid];
4. *Hierarchical hyperparameter* or *conditional hyperparameter* is a particular type of hyperparameter, that is used to define the dependencies between hyperparameters, i.e., a hyperparameter depends on another hyperparameter;
5. In practice, the choice of algorithm is a particular type of nominal hyperparameter, which is referred to as the *algorithm choice* [15] and *one-of* nominal hyperparameter [16]. The dependent hyperparameters of an algorithm are mapped to an algorithm by a conditional hyperparameter. In this thesis, each set of algorithms for an operation function shall be called an "*operator*", e.g., a resampling operator– ([SMOTE [17], SMOTETomek [18], SVMSMOTE [19]]), learning operator–([k-nearest neighbors [20], Support Vector Machines [21]]).

Without loss of generality, let a machine learning pipeline structure be modeled as a directed acyclic graph (DAG) which is restricted by implicit constraints [22], [23], i.e., some operators are optional, which might have a *do not use* option, but the learning operator is mandatory and as the last operator. Note that we shall use the term *machine learning pipeline structure* and *sequence of algorithms over operators* interchangeably in this thesis. The complete machine learning pipeline includes a *sequence of algorithms over operators* and their dependent hyperparameters.

[1] We note that, ordinal and continuous hyperparameters are typically bounded for practical reasons.

In conclusion, the search space defines the range of hyperparameters that the optimizer can search among to find the best set of hyperparameters (i.e., the best ML pipeline).

Objective function in AutoML optimization problem is one or more performance metrics (e.g., recall, precision rate, accuracy rate), a.k.a., evaluation metrics. These metrics are used to measure and evaluate the performance of a ML pipeline (e.g., a sequence of data pre-processing, feature engineering, and a learning algorithm) on a particular task. The optimization problem is called a *single-objective problem* if it uses one performance metric. Otherwise, it is called a *multi-objective optimization problem*. Throughout this thesis, the objective function is limited to the single-objective optimization problem. In other words, an objective function is a real-valued function that measures the performance of the ML pipeline on a given dataset using a specific performance metric. The objective function can be constructed using cross-validation, ensemble methods, or other techniques.

Optimization algorithm applied to AutoML is an algorithm that aims to find the optimal machine learning pipeline on a given dataset using a performance metric, i.e., objective function. Optimizing the objective function involves a trade-off between computational complexity and accuracy. The choice of algorithm depends on the problem's complexity, the dataset's size, and the computational resources available. The literature has highlighted several possible optimization approaches that can be used for AutoML, such as:

- *Bayesian optimization* (BO), e.g., Hyperopt [24], SMAC [25], [26], SPO [27], Spearmint [28], BO4ML [15].
- *Grid search* [29] and *Random search* (RS) [30], [31].
- *Racing procedure* (RP) [32], [33].
- *Bandit learning* (BL), e.g., Successive Halving [34], HyperBand [35].
- *Hybrid approaches*, e.g., BOHB [36] and DACOpt [37] hybridize Bayesian optimization and bandit learning, MIP-EGO [38] hybridizes Bayesian optimization and evolutionary algorithms.

However, Bayesian Optimization is the most commonly used approach in this domain as it was used in Auto-sklearn [39], Auto-Weka [40], [41] and Hyperopt-sklearn [42]. Hence, this thesis will review all mentioned approaches but mainly focus on improving BO and using it to solve the AutoML optimization problems.

1.1 Problem definition

In this thesis, the discussion is focused on AutoML optimization (AO) problems, and the ML problem is limited to supervised machine learning fields, i.e., regression and classification problems. Given a classification problem with a dataset $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$, where $\mathbf{x} = \{x_1, \ldots, x_k\}$ is a vector representation of the k features x, and y represents a label in classification problem or a continuous value in a regression problem. Solving an AutoML optimization problem refers to finding the best ML model $\mathrm{P} : \mathbb{X} \to \mathbb{Y}$ that transforms a vector $\mathbf{x} \in \mathbb{X}$ into a target value $y \in \mathbb{Y}$. In AutoML optimization, the ML model refers to as a ML pipeline model, which can include data pre-processing, encoding, feature selection, resampling, and learning algorithm. Hence, we shall interchange the terms of *ML model* and *ML pipeline model* in this section.

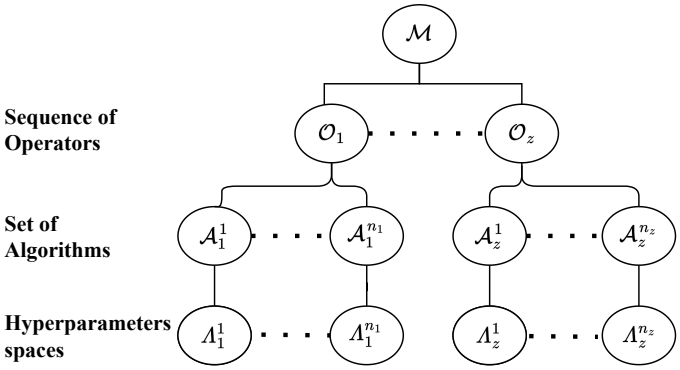


Figure 1.2: The structure of AutoML search space

Let $\hat{y}$ denote the predicted label calculated by model P for input features $\mathbf{x}$, i.e., $\hat{y} = \mathrm{P}(\mathbf{x})$, and $R(\hat{y}, y)$ denote a measure of classification accuracy between the predicted label $\hat{y}$ and the true label y. Let p be an ML pipeline setting trained on dataset $\mathcal{D}$ to produce the ML model P, i.e., $\mathrm{P} = (p, \mathcal{D})$. The performance of p is then calculated as:

$$\begin{aligned} f(p, \mathcal{D}) &= \frac{1}{m} \sum_{j=1}^{m} R(\mathrm{P}(\mathbf{x}_i), y_j) \\ &= \frac{1}{m} \sum_{j=1}^{m} R(\hat{y}_j, y_j) \end{aligned} \tag{1.1}$$

In practice, the predictive performance of ML pipeline p needs to be calculated based on its prediction on an unseen dataset, i.e., a test/validation set. The dataset $\mathcal{D}$ is then split into non-overlapping training and validation sets: $\mathcal{D}_{\text{train}} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ and $\mathcal{D}_{\text{valid}} = \{(\mathbf{x}_{n+1}, y_{n+1}), \ldots, (\mathbf{x}_m, y_m)\}$. Equation 1.1 allows calculating performance of the ML pipeline setting p when trained on $\mathcal{D}_{\text{train}}$ and evaluated on $\mathcal{D}_{\text{valid}}$ as:

$$f(p, \mathcal{D}_{\text{train}}, \mathcal{D}_{\text{valid}}) = \frac{1}{m-n} \sum_{j=1}^{m-n} R(\hat{y}_{n+j}, y_{n+j}) \tag{1.2}$$

However, the over-fitting problem is often encountered in applying optimization approaches [13]. To overcome this problem, the k-fold cross-validation can be applied to the Equation. 1.2, which becomes:

$$f(p, \mathcal{D}_{\text{train}}, \mathcal{D}_{\text{valid}}) = \frac{1}{k} \sum_{j=1}^{k} f(p, \mathcal{D}^j_{\text{train}}, \mathcal{D}^j_{\text{valid}}) \tag{1.3}$$

where $f(p, \mathcal{D}^j_{\text{train}}, \mathcal{D}^j_{\text{valid}})$ is performance of the ML pipeline p when trained and evaluated on the j^{th} data fold $\mathcal{D}^j_{\text{train}}$ and $\mathcal{D}^j_{\text{valid}}$, correspondingly.

Solving an AutoML optimization problem involves finding the optimal ML pipeline p that trains on a dataset $\mathcal{D}$ to produce the best-performing ML model P. Traditionally, the AutoML optimization problem is customarily treated and solved as a hyperparameter optimization problem (HPO), called the *HPO-based approach*, where the target ML pipeline p is constructed as a single algorithm. An alternate approach is to construct the target ML pipeline p structure as an ML pipeline of multiple ML algorithms, called *ML pipeline-based approach*. In this section, we will first look at solving the problem of AutoML optimization, also referred to as ML pipeline optimization. This topic is explained in detail in Section 1.1.1. Afterward, we will briefly discuss the traditional HPO-based approach, which treats the problem as a hyperparameter optimization problem. This discussion is covered in Section 1.1.2.

1.1.1 Machine Learning Pipeline Optimization

Let $\mathbb{O} = \mathcal{O}_1 \times \ldots \times \mathcal{O}_z$ denote the sequence of operators in the pipeline p, where each subsequent operator is applied to the output of the previous operator starting from input $\mathbf{x}$: $p(\mathbf{x}) = \mathcal{O}_z(\mathcal{O}_{z-1}(\ldots(\mathcal{O}_1(\mathbf{x}))\ldots))$. The functionality of each such operator can typically be delivered by one of the multiple available ML algorithms: herein, we assume $\mathcal{O}_{i\in\{1,\ldots,z-1\}} = \{\varnothing, \mathcal{A}^1_i, \ldots, \mathcal{A}^{n_i}_i\}$ for all operators except the

last and $\mathcal{O}_z = \{\mathcal{A}_z^1, \ldots, \mathcal{A}_z^{n_z}\}$ for the last operator which defines the learning algorithm – i.e., unlike the first $z-1$ operators, the last operator $\mathcal{O}_z$ has to be selected and cannot be $\varnothing$.

Typically, all algorithms have hyperparameters; we denote the domain of the i-th hyperparameter by $\boldsymbol{\lambda}_i$. Let $\Lambda^i = \boldsymbol{\lambda}_1 \times \ldots \times \boldsymbol{\lambda}_b$ be a hyperparameter space consisting of b hyperparameters of algorithm $\mathcal{A}^i$. Let $\Lambda_{\mathcal{O}_i} = \Lambda_i^1 \cup \ldots \cup \Lambda_i^{n_i}$ be the hyperparameter space of the operator $\mathcal{O}_i$ and $\Lambda = \Lambda_{\mathcal{O}_1} \cup \ldots \cup \Lambda_{\mathcal{O}_z}$ denotes the hyperparameter space of all considered algorithms for all considered operators. Let $\mathcal{M}$ denote the search space that includes the sequence of operators $\mathbb{O}$ and its corresponding hyperparameter spaces. The overall structure of the resulting AutoML search space is illustrated in Figure 1.2.

For readability, let $\mathcal{A}_{i,\lambda_j}$ represent algorithm $\mathcal{A}$ selected for an operator $\mathcal{O}_i$ and configured by a hyperparameter setting $\lambda_j \in \Lambda_{\mathcal{O}_i}$. Then, we denote a pipeline with algorithms selected and configured with their hyperparameters for all operators in the pipeline p as $p_{(\mathcal{A}_{1,\lambda_1},\ldots,\mathcal{A}_{z,\lambda_z})}$. We note that some approaches [43] consider in finding a suitable order of algorithms in the pipeline, which is called ML pipeline structure selection [22], [43]. Let the set of valid ML pipeline structures be denoted by $S = \{s_1, ..., s_h\}$, where $s \in S$ represents a valid ML pipeline structure that orders the position of algorithms $\mathcal{A}_1, \ldots, \mathcal{A}_z$ in the pipeline. The ML pipeline becomes $p_{(s,\mathcal{A}_{1,\lambda_1},\ldots,\mathcal{A}_{z,\lambda_z})}$. However, it is worth noting that most AutoML optimization approaches do not search for structure, such as Auto-sklearn [39], Auto-weka [40], Hyperopt-sklearn [42]. Instead, they use a fixed structure, i.e., $|S|= 1$. This fixed structure is based on a well-known linear sequence of operators recommended in literature or based on their experiences. When $|S|> 1$, it is referred to as the case of flexible ML pipeline structure search. This thesis limits our discussion to the fixed ML pipeline structure. Therefore, we will use the notation $p_{(\mathcal{A}_{1,\lambda_1},\ldots,\mathcal{A}_{z,\lambda_z})}$ to denote the ML pipeline, while the ML pipeline structure will not be further discussed. However, we will briefly discuss both approaches in Chapter 2, i.e., the discussion on the Fixed ML pipeline structure in Section 2.2.1 and the Flexible ML pipeline structure in Section 2.2.2.

To solve the AutoML problem (see Equation 1.4) and find the best choice of algorithms and their hyperparameters for the pipeline operators, every such choice needs to be evaluated. The $R(\hat{y}, y)$ denotes a metric that returns the accuracy of value $\hat{y}$ predicted by the pipeline compared to the real value y. Then, performance f of pipeline configuration $p_{(\mathcal{A}_{1,\lambda},\ldots,\mathcal{A}_{z,\lambda})}$ when trained on a training dataset $\mathcal{D}_{\text{train}} = \{(x_1, y_1), \ldots, (x_m, y_m)\}$ and evaluated on a validation dataset $\mathcal{D}_{\text{valid}} =$

$\{(x_{m+1}, y_{m+1}), \ldots, (x_{m+t}, y_{m+t})\}$ is calculated as: $f(p_{(\mathcal{A}_{1,\lambda},\ldots,\mathcal{A}_{z,\lambda})}, \mathcal{D}_{\text{train}}, \mathcal{D}_{\text{valid}}) = \frac{1}{t}\sum_{j=1}^{t} R(\hat{y}_{m+j}, y_{m+j})$. Hence, the AutoML optimization problem becomes the ML pipeline optimization maximizing problem:

$$(\mathcal{A}_{1,\lambda_1}, \ldots, \mathcal{A}_{z,\lambda_z})^* = \underset{\mathcal{A}_{1,\lambda_1},\ldots,\mathcal{A}_{z,\lambda_z}}{\operatorname{argmax}} f\Big(p_{(\mathcal{A}_{1,\lambda_1},\ldots,\mathcal{A}_{z,\lambda_z})}, \mathcal{D}_{\text{train}}, \mathcal{D}_{\text{valid}}\Big) \tag{1.4}$$

where $(\mathcal{A}_1, \ldots, \mathcal{A}_z) \in \bigtimes_{i=1}^{z} \mathcal{O}_i$ are all possible choices of algorithms for all pipeline operators, $\{\lambda_1, \ldots, \lambda_z | \lambda_1 \in \Lambda_{\mathcal{O}_1}, \ldots, \lambda_z \in \Lambda_{\mathcal{O}_z}\}$ are algorithms' hyperparameters and $f\Big(p_{(\mathcal{A}_{1,\lambda_1},\ldots,\mathcal{A}_{z,\lambda_z})}, \mathcal{D}_{\text{train}}, \mathcal{D}_{\text{valid}}\Big)$ is performance of the sequence operators and their corresponding hyperparameter choices when trained and evaluated on $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{valid}}$[2], correspondingly.

1.1.2 Combined Algorithm Selection and Hyperparameter Optimization

Traditionally, the AutoML optimization problem is commonly referred to as Combined Algorithm Selection and Hyperparameter Optimization (CASH) [39], [40] or Full Model Selection (FMS) [44] problem, in which the choice of algorithm is modeled as an additional categorical hyperparameter. Then, the AutoML optimization problem is treated as a HPO problem. As such, the choice of algorithms for each operator is modeled as an extra categorical hyperparameter $\boldsymbol{\lambda}^0$. The search space for the i^{th} operator is then defined as $\Lambda_{\mathbb{O}_i} = \boldsymbol{\lambda}_i^0 \cup \Lambda_i^1 \cup \ldots \cup \Lambda_i^{n_i}$, and the entire search space be $\Lambda = \Lambda_{\mathbb{O}_1} \cup \ldots \cup \Lambda_{\mathbb{O}_z}$ Hence, the AO problem becomes the HPO maximizing problem:

$$\lambda^* = \arg\max_{\lambda \in \Lambda} f(\lambda, \mathcal{D}_{\text{train}}, \mathcal{D}_{\text{valid}}) \,, \tag{1.5}$$

where $f(\lambda, \mathcal{D}_{\text{train}}, \mathcal{D}_{\text{valid}})$ is performance of the hyperparameter setting $\lambda \in \Lambda$ when trained and evaluated on $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{valid}}$, correspondingly.

In this setting, the categorical hyperparameters after the root of this hierarchical search space (see Figure 1.2) are known as the choice of algorithm for an operator. Consequently, algorithms and their local hyperparameters are treated at the same

[2]We note that AutoML tools are typically evaluated based on their performance on an unseen dataset during optimization, e.g., test set or ground truth set. These tools include several strategies to avoid overfitting together with other setups. These comparisons compare the performance of the whole AutoML system rather than the optimizer only. However, this thesis focuses on the optimization process to maximize performance on an unseen dataset for ML algorithms but known to the optimizer, referred to here as the validation set $\mathcal{D}_{\text{valid}}$. Therefore, we will not use the term test set $\mathcal{D}_{\text{test}}$ in this section. Instead, $\mathcal{D}_{\text{test}}$ will be used in Chapter 4 (Section 4.3), where we set up benchmarks for comparing different AutoML tools.

level. However, unlike the pure categorical hyperparameter, i.e., choose one in a set of nominal options, the choice of algorithms heavily affects other hyperparameters, i.e., once the algorithm is known, only its hyperparameters are relevant.

Another point worth mentioning is that HPO was originally developed to find the best hyperparameter setting from a single algorithm which is a much more straightforward problem compared to the AutoML optimization problem. In addition to HPO, AutoML optimization also searches for an optimized pipeline of algorithms. In AutoML, multiple algorithms must be considered, and these algorithms can belong to different phases in the ML pipeline, for example, pre-processing and learning models. This pipeline is restricted by some constraints, such as the learning task, i.e., classification and regression for supervised learning and clustering for unsupervised learning, which is the last step. For example,

- Auto-sklearn [39], [45] has up to six sequence operation steps: categorical encoder, numerical transformer, imputation transformer, rescaling, feature preprocessor, and learning operator.

- In comparison, Auto-Weka [40], [41] and Hyperopt-sklearn [42] have only two operators: preprocessor and learning operator.

Although, in general, AutoML can have different sizes in terms of operators and algorithms under operators, most operators are optional, and the learning operator is mandatory.

Furthermore, the algorithms and techniques used in an ML pipeline are tightly coupled because every operator step is directly affected by the previous step. For example, the data pre-processing step aims to produce a new dataset (balanced, reduced-dimensions, etc.), which can change the performance of the subsequent operator, such as the learning model. Consequently, the traditional approach for handling the choice of algorithm is a mismatch with the nature of the AutoML optimization problem.

1.2 Research Questions

In this thesis, we focus on AutoML techniques that aim at shorting the progress of producing ML applications. Some of the most critical questions that we will try to address are:

RQ1: How can we automatically construct high-quality ML pipelines for imbalanced data with HPO algorithms?

The classification algorithms commonly assume that the input data is equally distributed between classes. However, the distribution of classes in many real-world classification problems is ordinarily unequal, which reduces classification performance. To address this problem, we can apply a well-suited resampling technique to balance the imbalanced data before passing it to the classifier for training. There is no universal best algorithm for all problems. Hence, it becomes the model selection problem of finding the optimal combination of a resampler and classifier for the given problem, each selected from the sets of existing resampling techniques and classification algorithms. Besides, both resampling techniques and classification algorithms have local hyperparameters that need to be tuned to achieve better performance. Therefore, the *Model selection* (MS) and *Hyperparameter optimization* (HPO) tasks have to be considered. We note that HPO algorithms are initially designed for tuning hyperparameters of a single ML algorithm. The *Combined Algorithm Selection and Hyperparameter Optimization* (CASH) is a well-known approach for solving those two tasks simultaneously. CASH converts MS and HPO into a single HPO problem, which HPO algorithms can solve. Nevertheless, CASH has yet to be studied in detail for imbalanced class problems. Hence, we explore the potential of applying HPO algorithms to construct well-performing ML pipelines for imbalanced data automatically. Study on that problem gives us insights into how to design CASH experiments for class-imbalanced problems, such as determining applicable resampling and classification algorithms, constructing search space, and selecting performance metrics. The exploration and resolution of this research question are comprehensively tackled in Chapter 4 of this thesis.

RQ2: What is the most effective CASH optimization approach to achieve the optimal ML pipeline model for imbalance classification problems?

The Combined Algorithm Selection and Hyperparameter (CASH) problem can be addressed by a HPO algorithm. The two well-known HPO algorithms– Bayesian optimization and random search - are considered in our investigation to provide insights into choosing appropriate techniques for solving this CASH

for class imbalance problems. The comprehensive exploration and resolution of the research question can be found in Chapter 5 of this thesis.

RQ3: How to apply the CASH optimization to the steel surface defects classification problem where the distribution between classes is imbalanced and has unequal importance?

In collaboration with Tata Steel Europe - The Netherlands in the steel surface defects detection problem, we address the classification problem where the distribution between classes is imbalanced and has unequal importance, i.e., detecting severe defects that might heavily affect the quality of the final product is more priority than others with lighter affections. In CASH optimization problems, the proper performance metric is vital for evaluating ML models to correctly choose the optimal ML model. For the imbalance problem, several performance metrics (e.g., F1, geometric mean) can be used. However, those performance metrics treat all classes equally important. Thus, those metrics' overall accuracy cannot be used for this situation. In other words, the required performance metric for this real-world problem has yet to be. Therefore, to apply CASH to the steel surface defects detection problem, we need a new performance metric that considers both class imbalanced and unequal class importance problems. In Chapter 6, we delve into the heart of the research question, presenting a detailed analysis and solution.

RQ4: How does maximizing coverage of initial sampling improve BO performance to AutoML optimization problems? Bayesian optimization (BO) is a typical optimization approach that is structured by three fundamental components: initial sampling, surrogate model and acquisition function. The initial sampling step is typically restricted to a small budget since the effectiveness of BO becomes evident mainly in the later stages of optimization when it learns to produce better ML pipeline configurations. Another point worth mentioning is that the search space of AutoML is large, which includes many possible algorithms in the ML pipeline.

Many studies [46]–[48] noted that some algorithms have similar technical behaviours. To take advantage of this, we explore the potential of sampling on the group of similar algorithms for maximizing coverage of the AutoML search space already within a small budget of the initial sampling of BO. The exploration also provides insights into the effectiveness of optimized initial sampling to BO to characterize the response surface more accurately

and how we can adapt BO to solve the AutoML problems. For a detailed response to the research question, readers are directed to Chapter 7, where an exhaustive exploration and conclusion await.

RQ5: When should we stop tuning in an area of the search space?

As mentioned in **RQ4**, the AutoML search space is ample; trying every configuration is costly and typically impossible in practice due to the limited computation resources. The solution to **RQ4** can be adapted to separate the AutoML search space into multiple search areas (i.e., sub-spaces). Given limited computational resources for optimization, allocating more resources to the most promising areas while reducing resources to the unpromising areas to ensure achieving the best performance accuracy is necessary. Thus, the question is how to detect unpromising areas early and correctly. Chapter 8 provides an in-depth examination and resolution of the research question that guides this study.

RQ6: How can we efficiently allocate computational resources in the AutoML search space? The class of multi-fidelity approaches (Chapter 3. Section 3.2) aims to maximize the number of configurations to be evaluated within a limited budget. The central idea is to save computation resources for ineffective configurations and use them for other configurations. In this setting, the effective configuration will be tested on more data than the ineffective one. Racing procedures (Chapter 3. Section 3.2.1) and bandit learning (Chapter 3. Section 3.2.2) are the two well-known multi-fidelity approaches for HPO problems. Adopting their techniques to the AutoML optimization problem can provide insight into how we can handle resources efficiently in AutoML optimization. The comprehensive exploration and resolution of the research question can be found in Chapter 8 of this thesis.

1.3 Outline of the Dissertation

This thesis is organized as illustrated by a high-level overview in Figure 1.3. The motivation, research questions, and major contributions of each chapter are briefly introduced.

The relevant technical *background* for this thesis is split into two chapters where the AutoML optimization approaches are discussed in Chapter 3 and the relevant

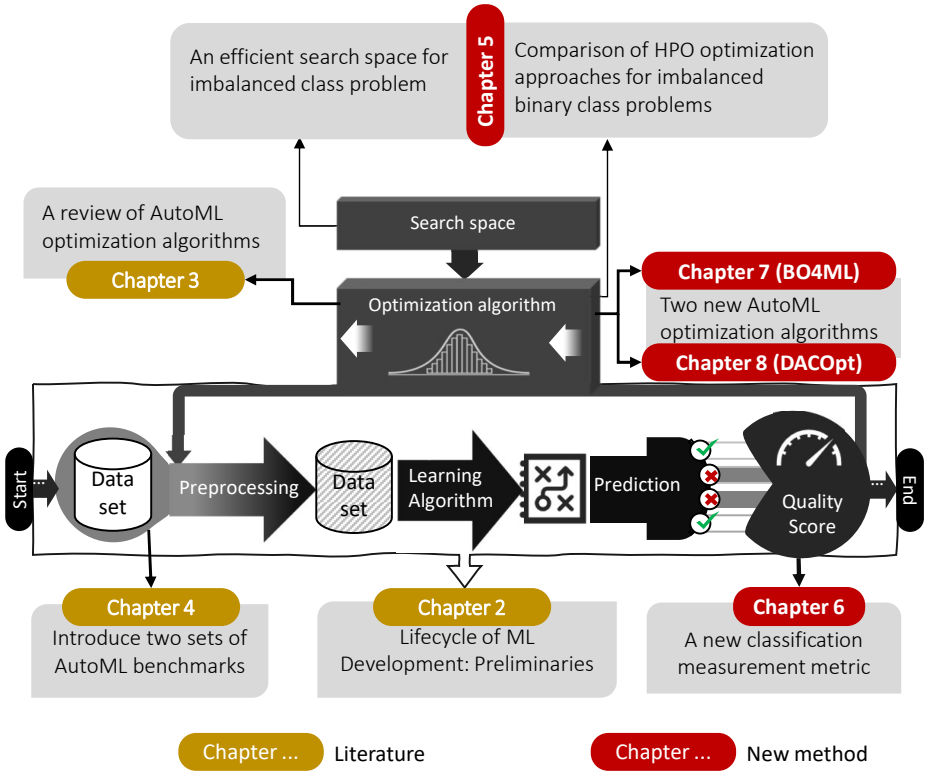


Figure 1.3: Mapping of chapters of this thesis to the process of development and life cycle of an AutoML pipeline.

machine learning development life-cycle knowledge provided by existing AutoML frameworks are discussed in Section 2.

Chapter 4 aims at introducing two sets of AutoML benchmarks to evaluate optimization approaches. Both scenarios provide a search space and a set of corresponding benchmark datasets– the first scenario includes two operators in the pipeline and 44 binary imbalanced benchmark datasets; the second scenario is a ML pipeline optimization of a dynamic search space with up to six operators and 73 AutoML benchmark datasets.

Chapter 5 focuses on the issue of how to apply optimization approaches to class imbalanced classification problems properly. Optimizer is a core component of an AutoML framework. In addition, various optimization approaches are compared in this chapter, where the most efficient approach is discovered.

Chapter 6 introduces a new classification measurement metric for the multi-class

classification problem where some classes are more important than others.

Chapter 7 and Chapter 8 discuss how to efficiently use BO for AutoML problems. Chapter 7 introduces a novel initial sampling strategy, *Combination-based sampling*, which is particularly designed for using BO for AutoML optimization problems. In addition, a novel BO approach, **BO4ML**, is proposed, where the proposed initial sampling approach is integrated. Chapter 8 introduces a novel contesting procedure, **D**ivide **A**nd **C**onquer **Opt**imization (**DACOpt**), which is an optimization approach specially designed for dealing with the large and complex search space of AutoML to help BO focus on promising search area earlier.

1.4 Publications and software packages

This thesis is based on the following peer-reviewed publications and software packages:

1. **Nguyen, D.A.**, Kong J., Wang H., Menzel S., Sendhoff B., Kononova A.V. & Bäck T.H.W. (2021), Improved automated CASH optimization with tree parzen estimators for class imbalance problems. In IEEE 8th international conference on data science and advanced analytics (DSAA), pp. 1-9, DOI: 10.1109/DSAA53316.2021.9564147.

    - Github: `https://github.com/ECOLE-ITN/NguyenDSAA2021`

2. **Nguyen, D.A.**, Kononova A.V., Menzel S., Sendhoff B. & Bäck T.H.W. (2021), Efficient AutoML via combinational sampling. In IEEE Symposium Series on Computational Intelligence (SSCI). pp. 01-10, DOI: 10.1109/SSCI50451.2021.9660073.

    - Github: `https://github.com/ECOLE-ITN/NguyenSSCI2021`
    - Pypi: `https://pypi.org/project/BO4ML`

3. **Nguyen, D.A.**, Kononova A.V., Menzel S., Sendhoff B. & Bäck T.H.W. (2022), An Efficient Contesting Procedure for AutoML Optimization, in IEEE Access, vol. 10, pp. 75754-75771, 2022, DOI: 10.1109/ACCESS.2022.3192036.

    - Github: `https://github.com/ECOLE-ITN/NguyenIEEEAccess2022`
    - Pypi: `https://pypi.org/project/DACOpt`

4. **Nguyen, D.A.**, Kononova A.V., Kong J., Jonker, K., Pipard, N., Mooi, J. & Bäck T.H.W. (2023), Automated Machine Learning Using Class Importance Weights For Imbalanced Multi-class Classification Of Steel Coil Defects, in review.

5. **Nguyen, D.A.**, Kononova A.V., Kong J., Jonker, K., Pipard, N., Mooi, J. & Bäck T.H.W. (2024), Efficient AutoML Optimization for Imbalanced Multiclass Data: A Case Study on Surface Defect Classification in Steel Manufacturing, in review.

    - Github: `https://github.com/anh05/AutoML-Multiclass-Imbalanced`

Other work by the author:

1. Kong J., Kowalczyk W.J., **Nguyen, D.A.**, Bäck T.H.W. & Menzel S., (2019), hyperparameter optimisation for improving classification under class imbalance. In IEEE Symposium Series on Computational Intelligence (SSCI), pp. 3072-3078, DOI: 10.1109/SSCI44817.2019.9002679.

2. Ullah S., **Nguyen, D.A.**, Wang H., Menzel S., Sendhoff B. & Bäck T.H.W. (2020), Exploring dimensionality reduction techniques for efficient surrogate-assisted optimization. In IEEE Symposium Series on Computational Intelligence (SSCI), pp. 2965-2974, DOI: 10.1109/SSCI4 7803.2020.9308465.

CHAPTER 2

Automated Machine Learning: An Overview

The term *machine learning* has become highly popular in today's technology and is expanding rapidly. Without realizing it, we use machine learning in our daily lives, such as in self-driving cars [1], medical diagnosis [2], automatic language translation [5], fraud detection [6], and defect detection [49].

Existing AutoML frameworks aim to automatically build the best ML pipeline for an arbitrary ML problem. However, applying AutoML to produce a useful ML product for a real-world problem eventually requires some knowledge of machine learning and software development. We first discuss some important aspects of the life cycle of machine learning development in Section 2.1. Next, this chapter will discuss all the functions of a typical AutoML tool, as shown in Figure 1.1, except for the optimization part, which we will discuss in Chapter 2 due to its significance in this thesis. Specifically, Section 2.2 provides an overview of commonly used approaches for determining the optimal ML pipeline architecture. Section 2.3 discusses various applications of meta-learning in AutoML. Section 2.4 presents the explainable and low-code techniques utilized in AutoML products.

2.1 Life Cycle of Machine Learning Development

The process of building up a machine learning system can be seen as a combination of the *Software Development workflow* [50] and the *Data science workflow* [51], which is shown in Figure 2.1. This workflow contains four main stages:

- *Data preparation* is the starting point for an ML project where the data is collected [52]. This will be discussed in Section 2.1.1.
- *ML pipeline optimization* is handled by the employed optimizer, which will be discussed in Chapter 3. Hence, Section 2.1.2 discusses the topic from the user's view rather than the view of an optimizer provider. In this section,

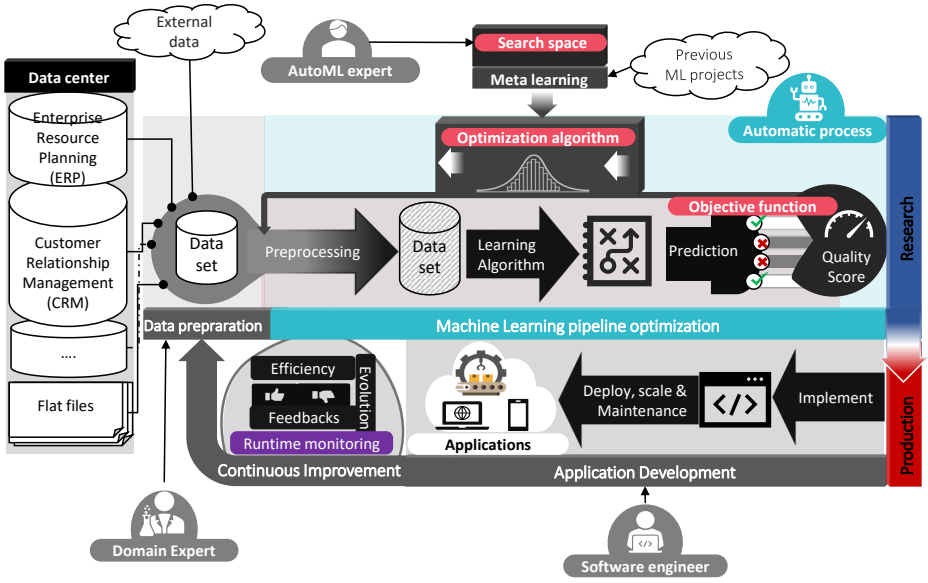


Figure 2.1: Machine learning development life cycle. This figure takes inspiration from the software development [50] and data science [51] lifecycles.

we clarify what parts of an ML pipeline can be applied for optimization for automatic purposes (Section 2.1.2.1), how we can evaluate an ML pipeline (Section 2.1.2.2), and the *over-fitting* problem [53] in Section 2.1.3, a common problem when applying optimization to AutoML [54].

- *Application Development* stage refers to the application development progress, where the final application is produced [50]. An engineering team typically does this stage with expertise in developing applications (e.g., software, web app, embedded application for Computer Numerical Control (CNC) machines).

- *Continuous Improvement* indicates the process of making small incremental changes to the developed application [55]. The ML application development is not a one-off process. In the ML applications development scenario, the constant improvement paradigm is more necessary since the data continues to be updated. The data distribution changes make the ML model's prediction performance less accurate. Thus, the deployed ML model must be retrained to adapt to the updated data. Besides, the selected ML pipeline may no longer be the best choice for the new data. Therefore, the ML pipeline has

to be re-tuned. We refer the interested reader to the Kaizen model [55] for this topic.

2.1.1 Data preparation

In practice, the data used for a particular machine-learning project can be collected from multiple sources, such as Enterprise Resource Planning (ERP), customer relationship management (CRM), network log files, Microsoft Excel, Microsoft Word, images, expert suggestions, or external sources. However, a real-life system, such as an ERP, can have thousands of data tables with complex relationships. Therefore, choosing the tables and data to use is a challenging question. Additionally, data from multiple sources (e.g., data crawled from websites, extracted from multiple internal systems, or gathered end-user working data files) might be incorrectly labelled or inconsistent in identity, used metric system units (e.g., millimetre (mm), centimetre (cm), meter (m)), and format (e.g., DateTime). Collecting data and joining different data sources are important and challenging. Hence, these tasks are typically performed manually by experts who deeply understand the data. In general, data preparation [52], [56]–[58] typically involves the following tasks:

1. **Data collection**: Useful data is obtained from operating systems, data warehouses, data lakes, and other information sources. During this step, it is crucial for data scientists, domain experts, members of the ML team, and end-users to verify that collected data is a good fit for the objectives of the anticipated ML applications [56].

2. **Data discovery and manual cleansing**: This step consists in investigating the gathered data to determine what it includes and what needs to be done to make it suitable for intended usage [52]. Next, the detected data flaws and mistakes are fixed to develop comprehensive and accurate data sets. For instance, faulty data is rectified or removed, missing values are filled in, and inconsistent entries are merged as part of the data sets clean up [59].

3. **Data organizing**: At this stage, the data must be modeled and organized to be suitable for ML [56]. For example, the data stored in comma-separated value (CSV) files and reorganizing image files.

It is worth noting that the target outcomes of all the above steps should include data and a set of prepared rules. The rule can be in the form of hard-coding, workflows, or generic formulas, which can be reused for future data in the production phase.

Apart from the mentioned *manual steps*, there are two classes of data-cleaning approaches:

1. *Semi-automatic* is a class of algorithms/tools that assist scientists in improving efficiency and reducing human effort during the data preparation phase, rather than being completely automatic. For instance, KARATA [60] is a tool that uses knowledge-based [61] and crowd-powered [62] approaches, detecting both correct and incorrect data to generate possible repairs for the identified incorrect data. Another class of methods, as exemplified by Krishnan et al. [63]–[65], involves suggesting cleaning for only a limited portion of the data while maintaining comparable outcomes to cleaning the entire dataset. However, these methods require humans to design data-cleaning operations applied to the dataset.

2. A small class of *automatic data cleaning* techniques aims to improve the data quality automatically. It can be applied to various datasets and used in AutoML frameworks (see Section 2.1.2.1 for an additional discussion). Nevertheless, they are usually hard-coded and limited to a few specific functions, such as handling data errors, missing values, redundant records, invalid values, and outliers [22], [66], [67].

2.1.2 Automated Machine Learning Pipeline

This section's topic is limited to supervised machine learning fields. This section focuses on processes that do not involve humans. We first start by listing the elements of a typical ML pipeline (Section 2.1.2.1). Second, we present the standard evaluation measurement metrics (Section 2.1.2.2). The last sub-section presents the common over-fitting problems when optimization approaches are applied (Section 2.1.3).

2.1.2.1 Machine learning pipeline

A generic ML pipeline $p : \mathbb{X} \rightarrow \mathbb{Y}$ designed for problem solving P is a sequence of operators that transforms a vector of features $\mathbf{x} \in \mathbb{X}$ into a target value $y \in \mathbb{Y}$ which can be, for example, a predicted value for a regression problem or a label for a classification problem. Examples of possible pipeline operators depend on problem P and can include data pre-processing, encoding, feature selection, and resampling.

In order to analyze this discussion, we need to use the notations introduced and explained in Chapter 1. These notations are crucial for our ongoing analysis, and we discussed them in detail in their original context in Chapter 1 to ensure a better understanding. Let $\mathbb{O} = \mathcal{O}_1 \times \ldots \times \mathcal{O}_z$ denote the sequence of operators in the pipeline p, where each subsequent operator is applied to the output of the previous operator starting from input $\mathbf{x}$: $p(\mathbf{x}) = \mathcal{O}_z(\mathcal{O}_{z-1}(\ldots(\mathcal{O}_1(\mathbf{x}))\ldots))$. Each operator's functionality can typically be delivered by one of the multiple available ML algorithms: here we assume $\mathcal{O}_{i\in\{1,\ldots,z-1\}} = \{\varnothing, \mathcal{A}_i^1, \ldots, \mathcal{A}_i^{n_i}\}$ for all operators except the last and $\mathcal{O}_z = \{\mathcal{A}_z^1, \ldots, \mathcal{A}_z^{n_z}\}$ for the last operator that defines the learning algorithm – i.e., unlike the first $z-1$ operators, the last operator $\mathcal{O}_z$ has to be selected and cannot be $\varnothing$.

ML pipeline structure Although several AutoML frameworks have been released to date, there are no best practices for ML pipeline structures for all ML problems in the literature. We only know that the learning algorithm must be at the end of the pipeline. Thus, an AutoML framework creates an ML pipeline using either a fixed structure based on the creator's expertise or a variable structure (the detailed discussion on *ML pipeline structure search* is given in Section 2.2). We note that the number of operators is flexible, as it highly depends on the ML problem (i.e., classification, regression), the underlying domain, and the input data itself (e.g., image, text, and tabulator input require different preprocessing algorithms).

The considered operators are usually grouped into two main phases:

1. **Preprocessing phase** includes several preprocessing tasks that can be seen as an augmentation step adding to the data preparation phase, but it is automatic. This step includes a sequence of optional steps. For example, a typical preprocessing sequence for a classification problem includes missing-value imputation, categorical encoding, data normalization, resampling, feature extraction, feature generation, and feature selection. Generally speaking, any data modification before utilizing a learning method is referred to as the *preprocessing step*. Note that the input data might be changed sequentially after processing via preprocessing operators.

2. **Learning phase** is the last operator in the ML pipeline. It aims to learn the relation within the dataset $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$ output from previous operator $\mathcal{O}_{z-1}$ and able to predict those of unseen dataset $\mathcal{D}_{\text{unseen}} =$

$\{(\mathbf{x}_{m+1}, y_{m+1}), \ldots, (\mathbf{x}_{m+t}, y_{m+t})\}$. More specifically, the learning algorithm aims at finding a learning function $f : \mathbb{X} \rightarrow \mathbb{Y}$ by generalizing beyond $\mathcal{D}$, that maps inputs $\mathbf{x} \in \mathbb{X}$ to outputs $y \in \mathbb{Y}$, i.e., $y = f(\mathbf{x})$. The learning function f can be formed by any option from the set of possible learning algorithms, e.g., linear models [68]–[70], tree-based models [71], [72], support vector machines [21], k-nearest neighbors [20], etc.

2.1.2.2 Evaluation measurement metrics

Finding the optimal ML model for the target problem is the main task in ML pipeline optimization. Hence, the vital questions are: How well does the model perform? Moreover, what is the accuracy of the model? To do so, we need a *performance metric* (or evaluation measure) to score the model's quality. The performance metric depends on the target problem, i.e., a classification problem or regression problem. For instance, the *accuracy rate* is usually used in classification problems [73]. However, when the classes are imbalanced, the *geometric mean* (GM) and *F-measure* are highly recommended [47], [74], [75] because of their ability to represent the minority class samples. In contrast, the *Mean Absolute Error* (MAE), and *Mean Squared Error* (MSE) are typically used to score a regression model [76]. Hence, choosing a suitable performance metric is the task given to experts. In this section, we will present the most commonly used performance metrics for classification and regression problems. For unsupervised machine learning, we refer the interested reader to other reviews of evaluation metrics for further discussions on that domain [77].

Measurement metrics for classification problems Classification algorithms stand for algorithms that predict label $y \in \mathbb{Z}$, i.e., discrete/categorical value such as Spam/Not Spam in Email Spam Filtering problem [78], based on a vector of features $\mathbf{x} \in \mathbb{X}$. Therefore, the accuracy of an individual prediction is either correct or incorrect by comparing the predicted value and the actual value. By comparing the predicted and actual values, the accuracy of an individual prediction is either correct or incorrect. The accuracy rate is most commonly used, as it simply computes the ratio of the number of samples correctly predicted to the total number of tested samples. However, the accuracy rate is not the sole metric to evaluate a classification model. In this section, we will summarize a total of seven performance metrics that are usually used for classification problems. This discussion is restricted to *binary-classification* problems. We refer the interested

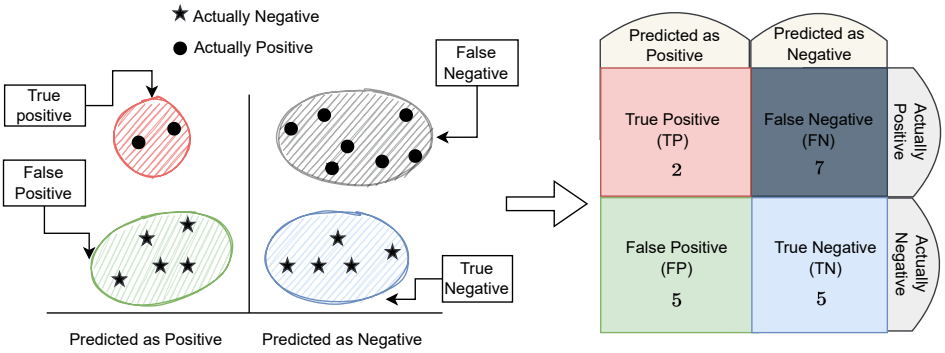
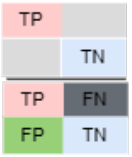
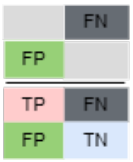
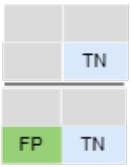
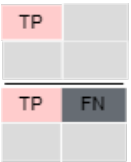
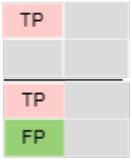


Figure 2.2: Illustration of confusion matrix. The illustration is inspired by the work of [79].

reader to Chapter 6 (Section 6.2.2) for multi-class classification problems and other reviews of evaluation metrics for other types of classification problems [73], [75], [80]. For binary classification problems, the performance metrics can be defined based on the confusion matrix. For consistency throughout this section, we use the confusion matrix example shown in Figure 2.2, as it can provide intuitive classification results. The figure on the left shows the distributions of the predicted and actual classes. The plot on the right shows a confusion matrix for this example. Using the confusion metric in Figure 2.2, we summarized several commonly used performance metrics in Table 2.1, including Accuracy rate, Error Rate, Specificity (or True Negative Rate), Sensitivity (or Recall or True Positive Rate), Precision, Balanced accuracy, Geometric mean[1], F_β-measure.

[1]The geometric mean mentioned here follows the work of [81], which is based on Sensitivity (accuracy on positive examples) and Specificity (accuracy on negative examples). Some other studies might be based on precision and recall.

Name	Formula	Illustration	Referring to Figure 2.2	Description
Accuracy rate	$\frac{TP+TN}{TP+FP+TN+FN}$		$\frac{2+5}{2+5+5+7} = 0.37$	The ratio of correct predictions on tested samples
Error Rate	$\frac{FP+FN}{TP+FP+TN+FN}$		$\frac{5+7}{2+5+5+7} = 0.63$	The ratio of incorrect predictions on tested samples
Specificity /True Negative Rate (TN_{rate})	$\frac{TN}{TN+FP}$		$\frac{5}{5+5} = 0.5$	The ratio of the number of correctly predicted negative samples overall actual negative samples
Sensitivity /Recall /True Positive Rate (TP_{rate})	$\frac{TP}{TP+FN}$		$\frac{2}{2+7} = 0.22$	The ratio of the number of correctly predicted positive samples overall actual positive samples
Precision	$\frac{TP}{TP+FP}$		$\frac{2}{2+5} = 0.29$	The ratio of actual positive samples among those predicted as positive

continued on the next page

Table 2.1: Performance Metrics for Classification Evaluations.

Name	Formula	Referring to Figure 2.2	Description
Balanced accuracy	$\frac{\text{Specificity}+\text{Sensitivity}}{2}$	$\frac{0.5+0.22}{2} = 0.36$	The arithmetic mean of class-wise sensitivity, i.e., Specificity and Sensitivity
Geometric mean	$\sqrt{\text{Specificity} \times \text{Sensitivity}}$	$\sqrt{0.5 \times 0.22} = 0.33$	The root of the product of class-wise sensitivity
F_β-measure	$F_\beta = (1+\beta^2)\frac{\text{precision}\times\text{recall}}{\beta^2\text{precision}+\text{recall}}$		The weighted harmonic mean of the precision and recall .
	$F_1 = (1+1^2)\frac{\text{precision}\times\text{recall}}{1^2\text{precision}+\text{recall}}$	$2 \times \frac{0.29\times0.22}{0.29+0.22} = 0.25$	$\beta = 1$ is typically used (i.e., F_β becomes F_1), also means the recall and the precision are equally important. Otherwise, recall is considered β times as important as precision .

Table 2.1: Performance Metrics for Classification Evaluations – continued from previous page

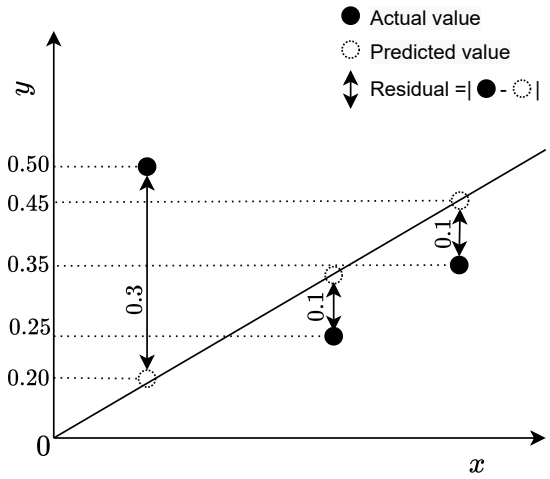


Figure 2.3: Example of regression prediction. The illustration is inspired by the work of [84].

Measurement metrics for regression problems Regression algorithms, as opposed to classification algorithms, predict a specific numeric value, i.e., $y \in \mathbb{R}$. For instance, weather forecasting [82] and housing price forecasting [83]. Hence, the accuracy of a prediction becomes the residual between the predicted value $\hat{y}$ and the actual value y, i.e., residual $= |y - \hat{y}|$ (see Figure 2.3 for illustration). The four common measurement metrics for regression problems are summarized in Table 2.2. For consistency reasons throughout this sub-section, we use the example shown in Figure 2.3, e.g., each method will be calculated base on that example. In addition, notations are shared throughout this section: t denotes the total number of tested samples.

Name	**Formula**	**Referring to Figure 2.3**	**Description**
Mean Absolute Error (MAE)	$\frac{1}{t}\sum_{i=1}^{t}\lvert y_i - \hat{y}_i\rvert$	$\frac{0.3+0.1+0.1}{3} = 0.17$	The average of the absolute differences between the actual values and the predicted values.
Mean Squared Error (MSE)	$\frac{1}{t}\sum_{i=1}^{t}(y_i - \hat{y}_i)^2$	$\frac{0.3^2+0.1^2+0.1^2}{3} = 0.04$	The average of the squared differences between the actual values and the predicted values.
Root Mean Squared Error (RMSE)	$\sqrt{\frac{1}{t}\sum_{i=1}^{t}(y_i - \hat{y}_i)^2}$	$\sqrt{0.04} = 0.2$	The square root of MSE
R^2 score	$1 - \frac{\sum_{i=1}^{t}(y_i-\hat{y}_i)^2}{\sum_{i=1}^{t}(y_i-\bar{y})^2}$	$\bar{y} = \frac{0.5+0.25+0.35}{3} = 0.37$	The coefficient of determination
	$\bar{y} = \frac{1}{t}\sum_{i=1}^{t} y_i$	$R^2 = 1 - \frac{0.11}{0.03} = -2.67$	

Table 2.2: Performance Metrics for Regression Evaluations.

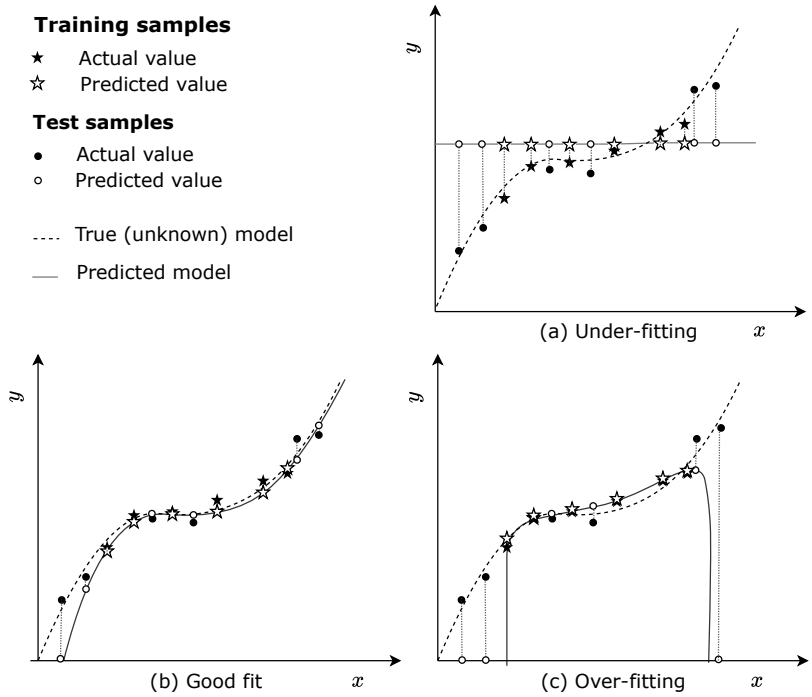


Figure 2.4: Illustration of Under-fitting (a), Good fit (b), and Over-fitting (c). The illustration is inspired by the work of [85].

2.1.3 Over-fitting and under-fitting

Generalization is a common problem in AutoML and hyperparameter optimization[53], [86]. During the optimization process, the highest-performing configuration is discovered through multiple trials on a training dataset. However, this optimal configuration may not generalize well to new, unseen data, leading to what is known as the over-fitting problem. Therefore, addressing over-fitting is an important concern in AutoML optimization. This section will provide an overview of the overfitting problem and highlight common practices to avoid overfitting in AutoML. In supervised machine learning, we have to find the best form of the function f that minimizes the difference between true value y_i and predicted value $\hat{y}_i$, i.e., $y_i \simeq f(\mathbf{x}_i), \forall \mathbf{x} \in \mathbb{X}$. Hence, the learning algorithm aims to produce a model that fits the data. However, if the model is *too fit* to the training data, it may result in the so-called *over-fitting problem*, in which the model performs well on the known training set but performs poorly on the unknown test set. In other words, the model does not have *generalization ability*. For clarification, Figure 2.4

illustrates under-fitting, good fit, and over-fitting problems. Assuming we have 12 samples, the 6 stars denote training samples, the 6 dots are test samples, the black colour indicates actual values, and the white colour indicates predicted values. The dashed curved line indicates the actual model, and the solid curved line indicates the predicted model. The plot in (a) shows an example of an underfitting. We can see that the predicted and actual models are completely different; the performance is poor for both the training and test samples. The predicted model in plot (b) was similar to the actual model. In this case, the predicted model may be a good fit. The last case is an example of over-fitting. The predicted model perfectly models the training samples but lacks generalization ability. Consequently, the model yielded poor predictions for the test samples. Under-fitting can easily be detected via performance metrics (see Section 2.1.2.2).

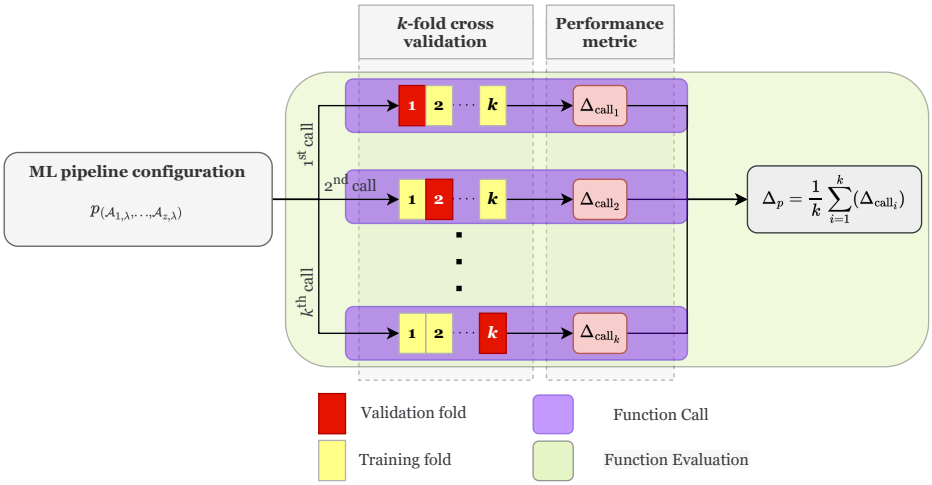


Figure 2.5: k-fold cross validation.

A common technique for avoiding over-fitting is *k−fold cross validation* (see Figure 2.5) in which the input data is randomly partitioned into k independent folds (also called *leave-one-out* technique). The average of the k *function calls*, computes the performance of one test case; a single fold is kept as a test set, while the remaining folds are used as the training set. The term *function evaluation* (green rounded-box) and *function call* (purple rounded-box) will be used in future discussions in this thesis, e.g., Chapter.3. Aside from $k-$fold cross-validation, there are several other techniques for preventing over-fitting, such as early stopping [68],

regularization [87], [88]. We refer the interested reader to those studies for further discussions on this topic.

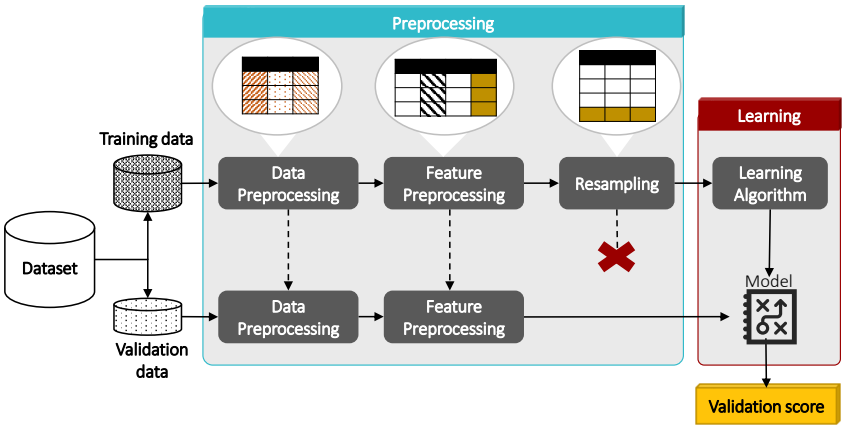


Figure 2.6: An example of calculating the validation performance of a typical Machine Learning pipeline.

Finally, when training and validating a model, we usually use exactly the same ML pipeline architecture for preprocessing and learning phases, for example, encoding categorical data and normalizing data in the same way. However, some operators in the preprocessing step must not be applied to the validation/test data. Otherwise, all techniques for overcoming over-fitting are useless. In general, the preprocessing can be grouped into two groups:

- Impact on the individual sample's quality, e.g., missing value imputation, categorical encoding, normalization data, feature extraction, feature generation, and feature selection.
- Changing the original distribution of the samples, for example, resampling (e.g., under-resampling, over-resampling, combine-resampling techniques) and removing outliers.

The first group has to stick to applying precisely the same (i.e., algorithms, hyperparameter settings, and their order) to both the training and test/validation sets. In contrast, the second group impacts the sample size to improve the training model's generalization and quality. Recall that a model's test/validation performance is based on the prediction values and actual values of the entire test/validation dataset samples — no more or fewer samples. Hence, the quality

measurement is incorrect when we base it on more or fewer samples' predictions than the actual test/validation set.

In other words, no approach that impacts the integrity of samples can be applied to the test/validation dataset. Figure 2.6 shows a workflow of a typical ML pipeline when applied to the training and validation data.

2.2 ML pipeline architecture search

The first step in solving any ML problem is to find a suitable machine learning pipeline structure. An ML pipeline includes several optional operators (e.g., imputation of missing data, encoding, scaling, feature extraction, and feature selection) and a mandatory learning operator. The generic ML pipeline is shown in Figure 2.7. The ML pipeline architecture search aims to answer the following research question: How many operators are required in the pipeline? Moreover, how are they ordered? The only known is the last operator, which is a learning algorithm, that is, classification, regression, or clustering. However, there is no fixed rule in the early steps, which confuses non-experts when creating their own ML pipelines.

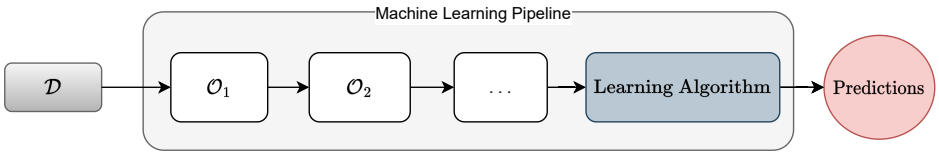


Figure 2.7: Prototypical ML pipeline architecture.

In addition, the performance of an ML pipeline is calculated based on its prediction quality on test data, that is, we only evaluate an ML pipeline when the input data are past all pipeline operators. Finally, every ML pipeline is unique [15], i.e., any changes to the ML pipeline lead to a different ML pipeline, which might change the final performance.

2.2.1 Fixed ML pipeline architecture

Many AutoML frameworks, such as Auto-Weka [40], [41], Hyperopt-sklearn [42], H2O [89], ATM [90], Auto-Gluon [91], [92] do not directly solve ML pipeline architecture search to reduce the complexity of determining ML pipeline structure.

Instead, they have predefined a fixed structure, which closely resembles a well-known linear sequence of operators recommended in literature or based on their experiences. Figure 2.8 shows an example of a commonly used fixed ML pipeline architecture for class-imbalanced problems, as used by [15], [37], [47], [74], [75]. A possible disadvantage of this approach is that it might result in inferior predictive performance for complex datasets requiring, e.g., multiple preprocessing steps.

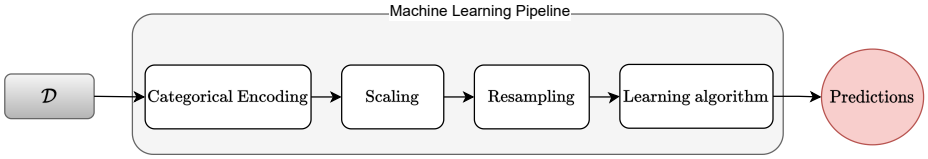


Figure 2.8: Fixed ML pipeline architecture used by most imbalanced-class classification studies.

2.2.2 Flexible ML pipeline architecture

In order to achieve the best performance for a given problem, human experts usually build highly specialized ML pipelines, i.e., the ML pipeline is adaptable to a specific task. An illustration on ML pipeline architecture search is shows in Figure 2.9.

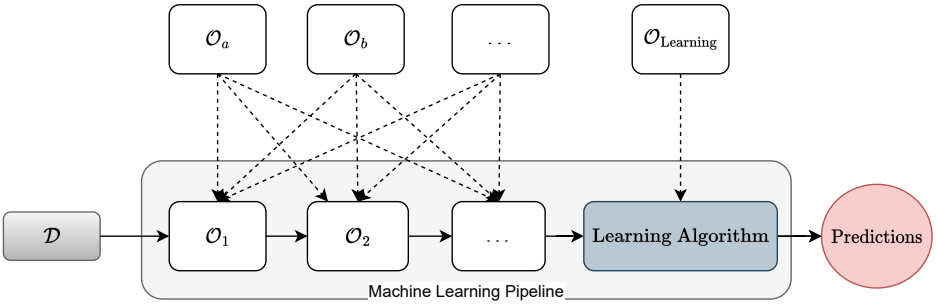


Figure 2.9: Flexible ML pipeline architecture, where the ML pipeline structure is also searched for.

This flexibility is missing from the fixed ML pipeline structure approach. To address this issue, one class of AutoML frameworks considers flexible ML pipeline architecture. Auto-Sklearn [39], [45] can be seen as a semi-flexible approach. Auto-Sklearn predefined a set of fixed structures for ML problems. For a given ML problem, it generates a pipeline structure using meta-features of the input

AutoML tool	Underlying technique domain
TPOT	Genetic programming [102], [103]
AlphaD3M	Reinforcement learning [104]
MLPlan	Hierarchical task networks [105]
Mosaic	MonteCarlo tree search [106], [107]
FEDOT	Evolutionary algorithms [108]–[110]

Table 2.3: AutoML frameworks support ML pipeline architecture search

data, e.g., the structure for binary and multi-class classification problems might be different. However, the same structure will be used for the same ML problem domain. Apart from that *if-else* style, there is a class of AutoML frameworks that support ML pipeline architecture search, including TPOT [43], AlphaD3M [93], [94], ML-Plan [95], [96], and P4ML [97], Mosaic [98], FEDOT [99]. The main idea of these approaches is to apply restrictions in the form of ad hoc configuration, primitive taxonomies, or context-free grammars [100], [101]. Table 2.3 provided the relevant techniques to these AutoML tools.

Lastly, a class of techniques based on meta-learning will be discussed in Section 2.3.

2.3 Meta Learning

A typical ML pipeline optimization problem consists of three fundamental components – a search space, an optimizer, and an objective function. The search space describes the feasible search domain. The optimizer is used to discover the best combination of algorithms over operators and their optimized hyperparameters, thereby maximizing the performance of the objective function. Finally, the objective function is a child program that evaluates the settings of the ML pipeline, resulting in a real-valued performance measures, such as accuracy, precision, and recall rate. In other words, the ML pipeline optimization process aims to find the most suitable solution from a predefined search space.

Usually, for optimizing a new (unknown) ML problem, the optimizer explores the search space from scratch. In contrast, ML experts take advantage of previous tasks (e.g., referring to literature and their experiences so far) to shorten the optimization process and avoid wasting time on unpromising search areas. Inspired by the behavior of human experts when dealing with a new ML task, meta-learning is quite similar to learning from the experiments of previous ML tasks to increase

the efficiency of the search. Meta-learning is the science of learning similar datasets, which can be characterized by a set of *meta-features* [111], which might include:

1. *Simple features* includes the following: number of samples, number of features, number of classes, number of missing values, number of instances with missing values, number of numeric features, number of categorical features, number of binary features.

2. *Statistical features*: mean, standard deviation, mean skewness, quarterlies.

3. *Information-theoretic features*: class entropy, mean mutual information.

4. *Land-marking* [112]: Performance evaluations of some simple classifiers on the entire data or sub-set of data [113], e.g., performances evaluated by k-nearest neighbor with 1 neighbor.

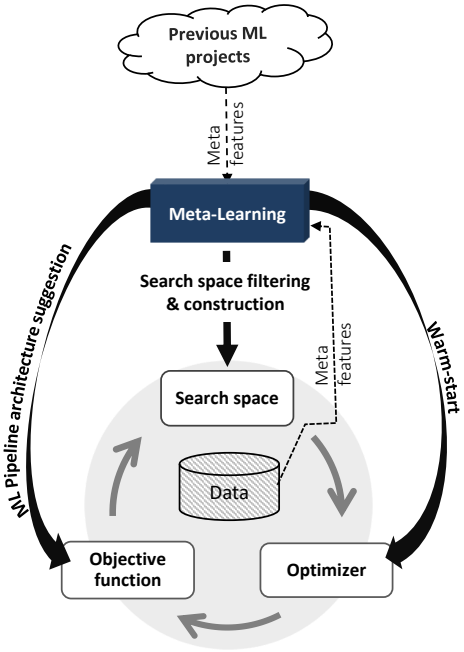


Figure 2.10: Applications of meta-learning. The illustration is inspired by the work of [114], [115].

Meta-learning can be applied in many stages of the ML pipeline optimization process (see Figure 2.10):

- *Search space filtering and reduction*: The search space is usually predefined by the AutoML owners, which is typically large with many possible algorithms that have been integrated. Furthermore, the hyperparameters of the chosen algorithms are set by a wide range of values. Nevertheless, the search space is arbitrarily defined without referencing the given ML problem. Therefore, meta-learning can be used to reduce the large search space, such as eliminating unimportant hyperparameters, irrelevant algorithms (e.g., [116]–[121]), and automatically construct a (minimal) search space (e.g., [101], [122]).

- *ML pipeline architecture suggestion*: Apart from the mentioned approaches for flexible ML pipeline architecture in Section 2.2. Meta-learning is a promising research domain for identifying ML pipeline synthesis. [101] proposed a data-centric approach, called DSWIZARD, that learns from related ML tasks to construct a suitable ML pipeline for the target problem. Similarly, predicting the pipeline's performance and favoring a good pipeline architecture to be constructed was studied in [94], [97], [98], [111], [123], [124].

- *Warm-start for optimizer*: Traditionally, the optimization process often starts from scratch. For example, the initialization step in Bayesian optimization (Section 3.1.3) randomly selects some ML pipeline configurations without any evaluation of the given dataset, that is, with the same random seed – the optimizer generates the same set of initialization configurations for any dataset. On the other hand, by learning from previous tasks, many studies proposed to start from a set of the best configurations of the related tasks [39], [120], [125]–[127]. In this manner, the underlying optimizer can characterize the search space by focusing on promising areas for the given data. Consequently, the optimizer maximizes the chance of finding the best solution early. This is the general idea of a warm-start in ML pipeline optimization.

Finally, we refer the interested reader to other reviews of meta-learning for further discussions on this topic [114], [115].

2.4 Explainable and low-code for AutoML

AutoML and optimization studies have been applied in many industrial domains [7], [128]–[130]. However, the current state of AutoML is only seen as a reference tool for human experts in real-life application development [131]. From a practitioner's

perspective, AutoML is also seen as a black-box, i.e., optimization process, that optimizes another black-box, i.e., objective function [11]. Also, different frameworks might lead to different results and suggestions. Many studies have been noted that users do not trust AutoML systems [12], [132]–[134]. Without understanding its optimization behaviors, users might not confidently decide to use its suggestion (i.e., the best-found ML pipeline). For instance, [12] has revealed that even though AutoML might deliver high-quality solutions, practitioners refuse to use them as they do not want to be held accountable for a model they do not understand. In addition, [11] concluded that the main reason for the limited trust of practitioners is the limited explanation and transparency of the outcome of AutoML.

Therefore, establishing trust in an AutoML system is an important motivation for explainability [135], that is, explaining in a way that humans can understand in a reasonable time [136].

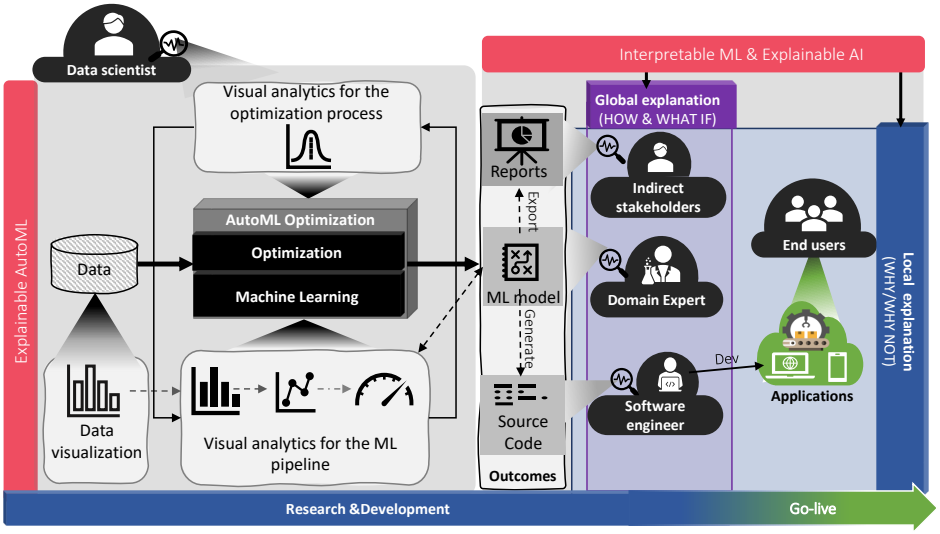


Figure 2.11: Requirements analysis of different user groups on their need for ML explainability. Explainable AutoML is a sub-class of explainable AI that targets ML high-skill user groups that can read/understand ML algorithms and visualization techniques.

2.4.1 Stakeholders of AutoML

Stakeholders involved in a ML project can be grouped into three groups:

- *Direct stakeholders*: Indicate actors who are directly involved in the whole process of developing their final ML application with the day-to-day activities, e.g., data scientists, domain experts, and software engineers.

- *Indirect stakeholders*: People are not affected by the project but may support the project in some terms but not focused on the process of finishing it, such as system infrastructure administrators who may prepare a suitable system infrastructure and system configuration, legal advisors might be involved in the term of making the end-users policies, managers who set the business goals for the project, build the project team, customers (for an internal project, and the customers can be the board of managers) might approve the project results.

- *End-users*: They can be internal or external actors who will use the output application for their daily work.

2.4.2 Components of an explainable AutoML

Each group of actors had different skill sets, knowledge, and demands. Thus, the explanation is that adaptability is based on its properties. Figure 2.11 summarizes the requirements analysis of different user groups on their need for ML explainability. The co-badged guidance [137] defines the explainability of intelligent systems as a combination of technical (information extraction) and non-technical (communication method) considerations. Furthermore, recall that AutoML is a combination of optimization and ML. Hence, the scope of explainable AutoML includes both aspects. According to the relevant discussions on optimization [138] and machine learning [51], we formulate the technical explainability requirements for AutoML as three complementary approaches that form to increase trust and transparency:

1. *Global explanations on the optimization level* aim to explain the decision-making process of the optimizer. In addition, this level of explainability provides helpful information on optimizer behavior to illustrate optimization convergence and how it constructs the pipeline.

2. *Global explanations on a particular model level* aim to explain the general model's decision-making process. This level of explainability is about understanding *how* the model makes decisions and the distribution of the target outcome based on the features.

3. *Local explanations for a single prediction*: Global explainability is more useful in the research and development phase as it helps project owners determine how their data distribute and how the ML pipeline transforms the input data to lead to the final result. The local explainability helps examine what the model predicts for a particular sample and explains *why/ why not*.

Those applications of explanation are used for many demands, such as debugging ML models [139], explaining medical decision-making [140], explaining predictions for classification problems [141], and explain autonomous agent behavior [142], [143].

2.4.3 Maturity Levels of Automation Tools

According to Figure 2.11, the optimization and programming phases necessitate a high level of technical knowledge. The optimization phase requires data science skills, and the programming phase aims to build the deliverable application, which requires coding skills. Both target a goal that can be used by the layperson (lay scientist/ developer). The levels of automation tools for AutoML are summarized in Figure 2.12.

2.4.3.1 Tools for data scientist

To be able to understand and be accountable for the outcome of AutoML in a real-world application, the practitioner usually investigates the optimization behaviors and its final suggestions. For example, the practitioner may plot several visualizations of how the data transformed through the pipeline or trace back the optimizer's decision-making process by plotting the necessary figures that they can explain to others (i.e., non-technical users). However, the process of making figures might be costly, as it depends on how familiar the data scientist is with the platforms used. Hence, the practitioners must develop practice skills in ML and coding [132]. That fact limits the usefulness of AutoML and misses the opportunity to lead toward helping humans apply ML to real-life applications with limited ML and statistics knowledge.

To overcome these limitations and save time for data scientists, some AutoML platforms provide an additional set of visualizations to explain the decision-making process. These can be classified into two main groups:

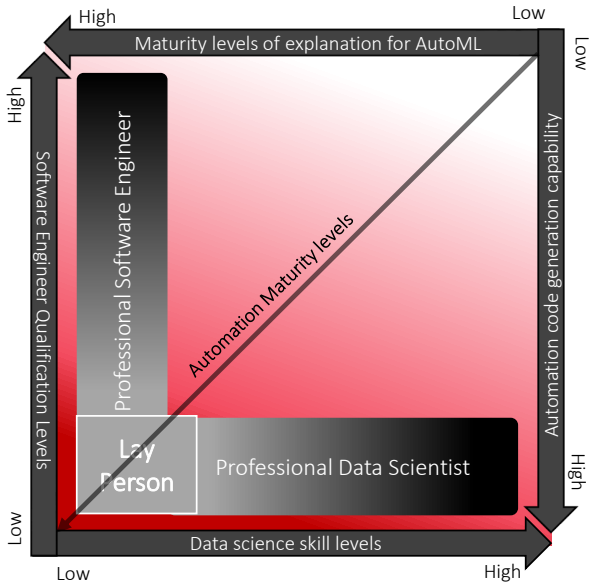


Figure 2.12: Automation Maturity levels for AutoML. The Top-bottom arrows show levels of explainability vs. the data science skill levels that can understand the corresponding explanation related to data scientists. The left-right arrows indicate the levels of automation in building an ML application vs. the needed skill.

1. *Command line based (CLB)*: This set of AutoML platforms provides visualizations via built-in functions. The user can plot figures by executing those functions. However, the user must have a ready-to-use development tool, such as Jupyter Notebook, Anaconda, or dot Net Visual Studio, as well as basic coding skills (e.g., Python, R, C#). They mainly focus on satisfying the *global explanations of the optimization level*. Their functions provide ways to visualize historical data over the tuning process regarding the performance (e.g., [45], [144]), the optimizer behavior for choosing hyperparameter values (e.g., [144]–[146]), and comparing ML pipeline structures (e.g., [11], [147]).

2. *Graphical user interface based (GUI)*: refers to a group of platforms that work as standalone software or web apps and is aimed at users who do not have coding skills. All steps, starting from importing the dataset to tuning and visualization, are integrated into a single interface. Well-known platforms are Google Vizier [145], HyperTuner [148], HyperTendril [149], IBM Watson

Studio[2], Microsoft Azure[3], Databricks AutoML[4], Rapid-Miner[5]. They significantly improved development speed. However, their main limitation is the limited range of applicability, manageability, and flexibility than the others.

Lastly, visualization is an important component, but it is only considered a support function in the value chain of AutoML. Therefore, some open-sourced AutoML platforms might not be interested in developing that function. Fortunately, several platforms add visualization functions on top of other platforms to create a new platform that supports explainability well. AutoViz [150], for example, is an extended version of AutoAI [131], ATMSeer [134] extends ATM [90], and XAutoML [11] based on five other platforms are – Auto-Sklearn [39], [45], DSWIZARD [101], Scikit-learn [151], FLAML [152] and Optuna [144]. While Hyperopt [153] was used in Google Vizier [145], HyperTuner [148] and HyperTendril [149].

2.4.3.2 Tools for software engineer

This section reviews the low-code technique used in AutoML products, mainly supporting the programmer in producing the final ML product rather than the scientist in the research phase. Low-code and no-code refer to software development methodologies that indicate the concept of practitioners creating their solutions with little (or no) technical skills. Technically, the no-code platforms consist of standard pre-built components and a visual development tool that allows practitioners to use a graphical interface to build their application in a *drag-and-drop* (e.g., Sway AI[6]) or a wizard-based interface (e.g., Akkio[7]) styles. However, those platforms are typically limited to some predefined problems, such as object detection in computer vision (MakeML[8]), image classification (Lobe[9]). That is to say, the automation ability (i.e., ready-to-use) is inversely proportional to the range of applicability. Thus, AutoML platforms that target solving unlimited problems usually stop at the basic level of low code as they only generate the relevant source code for the ML model, e.g., Amazon SageMaker Autopilot [154] export ready-to-use Jupyter notebooks for tested ML pipelines.

[2] https://www.ibm.com/cloud/watson-studio/autoai
[3] https://azure.microsoft.com
[4] https://databricks.com
[5] https://rapidminer.com/
[6] https://sway-ai.com
[7] https://www.akkio.com
[8] https://makeml.app/
[9] https://www.lobe.ai

CHAPTER 3

An In-Depth Review of AutoML Optimization Approaches

This chapter introduces commonly used optimization approaches for AutoML optimization problems. We note that our discussion in this chapter will rely heavily on the problem definition and accompanying notations discussed in Chapter 1 (Section 1.1). These notations are crucial for the ongoing analysis, and we discussed them in detail in their original context in Chapter 1 to ensure a better understanding.

All of the approaches presented in this study follow the same principle: finding the best machine learning pipeline configuration $p \in \mathcal{M}$ to maximize a measurement performance[1] for a given machine learning problem with the k-fold cross-validation technique. They can be divided into two groups:

1. The performance of a particular configuration will be evaluated on all k-folds.
2. They intend to save computational cost by evaluating it on a subset of data, e.g., on fewer folds, to infer performance on the entire data. Hence, we shall use the term *function call* to indicate one-time access to a configuration on one fold.

The term *function evaluation* indicates the average performance over k folds, i.e., a *function call* is k times cheaper than a *function evaluation* in terms of evaluating data input. The difference between the term *function call* and *function evaluation* in this thesis is shown in Figure 2.5.

3.1 Black-box optimization approaches

In general, both hyperparameter and AutoML optimization problems are typically treated as black-box optimization problems for various reasons. For instance, we

[1] See Section 2.1.2.2.

cannot access a gradient of the objective function concerning the hyperparameters, or it is not possible to directly optimize the generalization performance as the training datasets are of limited size [8]. Generally, every black-box optimization approach can solve these problems. This section introduces three common optimization approaches, grid search (Section 3.1.1), random search (Section 3.1.2), and Bayesian optimization (Section 3.1.3). The working principles of these three approaches are shown in Figure 3.1. As shown, grid search sequentially evaluates points individually on a user-defined grid. On the other hand, a random search evaluates points at random, as the name implies. Bayesian optimization (BO) is a more complex technique based on advanced probabilistic models that makes it intelligent for automatically finding suitable configurations in the search space. In this example, we can see that BO can find more configurations with stronger results than other approaches. That is, more samples in the purple-filled contours.

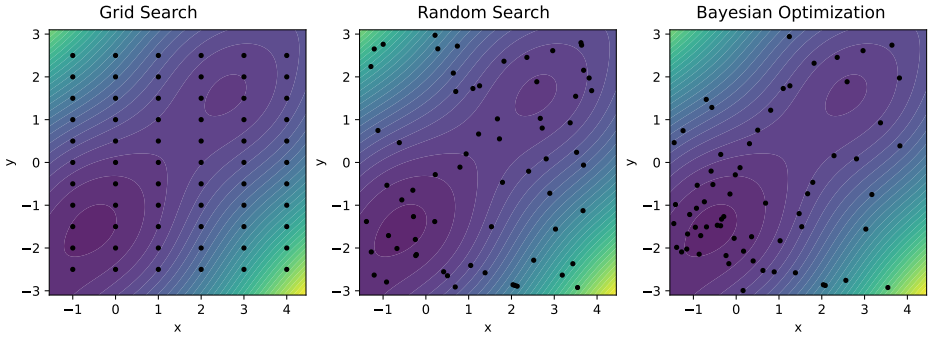


Figure 3.1: An illustration between grid search (left), random search (middle), and Bayesian optimization (right) for hyperparameter optimization on the McCormick's function $f(x, y) = -1.5x + 2.5y + (x - y)^2 + \sin(x + y) + 1$ with two continuous parameters: $x \in [-1.5, 4]$ and $y \in [-3, 3]$. All three approaches used a total of 66 functions evaluations. Purple-filled contours indicate regions with strong results, whereas yellow ones show regions with poor results.

3.1.1 Grid Search

Grid search is the most basic optimization algorithm. Given a set of hyperparameters, each of which has a (finite) set of values, for instance, continuous hyperparameter, e.g., $[0, 1] \in \mathbb{R}$, ordinal hyperparameter, e.g., $[1, 10] \in \mathbb{Z}$, boolean hyperparameter, e.g., [True, False], nominal hyperparameter, e.g.,[Linear, RBF, Poly, Sigmoid].

We enumerate all combinations of these sets and create a list of all candidates. Grid search evaluates each of these candidates and chooses the best configuration among them – the number of function evaluations is precisely the number of configurations. However, practitioners are usually restricted by a limited computational budget, i.e., the number of function evaluations, for hyperparameter optimization and AutoML optimization problems. Such a limited budget is typically much smaller than the number of possible evaluation configurations. Thus, a limited budget restricts the applicability of grid search.

3.1.2 Random Search

Unlike grid search, which assesses all configurations (for continuous hyperparameters based on a sufficiently coarse-grained discretization), random search [30] evaluates only a subset of available candidate configurations at random until the given budget runs out and returns the best of the sampled configurations. The random search for AutoML optimization is summarized in Algorithm 1, it consists of the following two steps:

- Generate a set of random configurations (line 3): here we adapted random sampling in unstructured HPO problem to AutoML optimization problem based on the search space (i.e., operators and hyperparameters space) and the number of needed configurations. The sampling algorithm is presented in Algorithm 2.
- Evaluating and selecting configuration: Each setting $p_i \in \{p_1, \ldots, p_B\}$ from the previous step will be evaluated on the objective function f (line 5). Next, the current best setting is updated (lines 6-9).

Lastly, when the optimization process is done, the best setting p^* is reported.

Random Sampling used in Algorithm 1 is presented in Algorithm 2:

- Random selection of a sequence of operators: All operators $\mathcal{O}_{1,\ldots,z} \in \mathbb{O}$ are randomly sampled to have a sequence of algorithms (line 5).
- Sampling hyperparameters: The corresponding hyperparameters are sampled randomly (lines 6-7), taking into consideration the selected algorithms in the previous step. The result is returned as a complete ML pipeline setting p, i.e., a sequence of ML algorithms and their hyperparameter settings.
- Lastly, the set of sampled configurations is returned.

Algorithm 1: Random Search

Input: $\mathbb{O}$: sequence of operators, Λ: hyperparameter spaces, f : objective function, B : number of iterations
Output: p^*: the best found configuration

$p^* \leftarrow \varnothing$
$\Delta^* \leftarrow 0$
$\Theta = \{p_1, \ldots, p_B\} \leftarrow$ RANDOM SAMPLING $(\mathbb{O}, \Lambda, B)$ `// see Algorithm 2`
foreach $p_i \in \{p_1, \ldots, p_B\}$ **do**
  $\Delta_i \leftarrow f(p_i)$ `// evaluate the configuration` p_i
  if $\Delta_i > \Delta^*$ **then**
    $p^* \leftarrow p_i$
    $\Delta^* \leftarrow \Delta_i$
  end
end
return p^*, Δ^* `// return the best found setting`

Recent studies [8], [13], [30], [47] have noted that random search can perform better than grid search, particularly when only a few hyperparameters impact the performance of the machine learning algorithm. Despite its simplicity, random search remains a crucial benchmark for evaluating the effectiveness of new optimization methods.

Algorithm 2: Random Sampling for AutoML optimization

Input: $\mathbb{O}$: sequence of operators, Λ: hyperparameter spaces, T: number of configuration
Output: $\Theta = \{p_1, \ldots, p_T\}$: set of T configurations

$t \leftarrow 1$
$\Theta = \varnothing$
while $t \leq T$ **do**
  $p \leftarrow \varnothing$
  foreach $\mathcal{O}_i \in \mathbb{O}$ **do**
    $\mathcal{A}_i^{n_i} \leftarrow \mathcal{U}(\mathcal{O}_i)$ `// randomly choose one algorithm for the` i^{th} `operator`
    $\lambda_i \leftarrow \mathcal{U}(\Lambda_i^{n_i})$ `// randomly select a hyperparameter setting for the selected algorithm` $\mathcal{A}_i^{n_i}$
    $p \leftarrow p \cup \{\mathcal{A}_i^{n_i}, \lambda_i\}$
  end
  $\Theta \leftarrow \Theta \cup p$ `// insert the new configuration` p `into` Θ
  $t \leftarrow t + 1$
end
return $\Theta = \{p_1, \ldots, p_T\}$ `// return a set of` T `configurations`

3.1.3 Bayesian Optimization

As the AutoML optimization task is typically time-consuming, it is preferable to devise/choose an optimizer that delivers a good ML pipeline setting with a relatively small computational budget. Building upon surrogate models and the expected improvement criterion, Bayesian Optimization (BO) [155] is designed for such a scenario. Generally, BO iteratively updates a surrogate model $\mathcal{P}(f|\mathcal{H})$which aims to learn the probability distribution of the response value conditioned on setting p, from the historical information, i.e., the so-far evaluated ML pipeline settings and the corresponding objective function $\mathcal{H} = \{(p_i, \Delta_i)_{i=1}^n\}$. The new candidate ML pipeline is chosen by optimizing the acquisition function [156], [157], which is defined over the surrogate model $\mathcal{P}$ and often balances the exploration and exploitation of the search. A detailed outline of the BO is presented in Algorithm 3 and Figure 3.2.

Many variants have been proposed for BO, including the Sequential Model-based Algorithm Configuration (SMAC) [25], Sequential Parameter Optimisation (SPO) [27], Mixed-Integer Parallel Efficient Global Optimization (MIPEGO) [38], and Tree-structured Parzen Estimator (Hyperopt) [24], [153], [158]. They differ mostly in the initial sampling method, the probabilistic model, and the acquisition function. Common choices for the probabilistic model are Random forests (RF) [72], Gaussian process regression (GP) [159], and TPE [24]. As for the acquisition function, the Expected Improvement (EI), the Probability of Improvement (PI) [157], and the Upper Confidence Bound (UCB) [160] are more frequently applied among many other alternatives.

3.1.3.1 Probabilistic Regression Models

The central idea of BO is to construct a surrogate model from the observed data points on real-valued objective function f. The surrogate model aims to predict the performance of untested ML pipeline configurations by modeling the relationship between the set of evaluated configurations Θ and their true response value Δ. In the following, we will briefly introduce three commonly used surrogate models: (1) Gaussian processes – the well-known traditional surrogate model, (2) Random Forest, and (3) Tree-structured Parzen Estimator – two popular surrogate models for AutoML optimization.

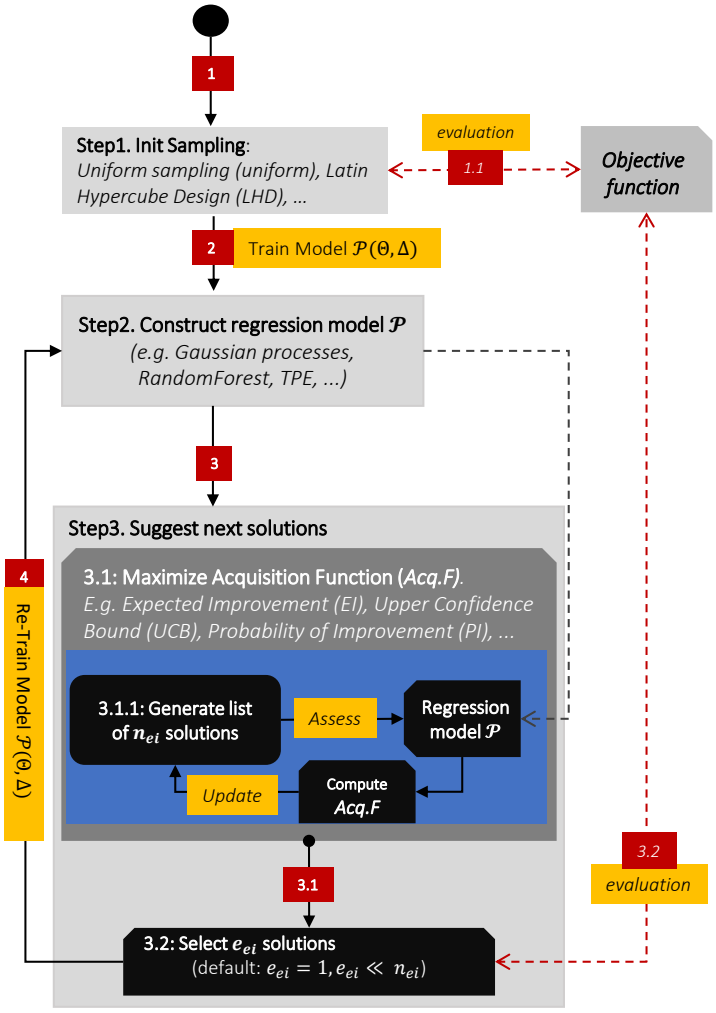


Figure 3.2: Bayesian Optimization

Gaussian processes

Gaussian processes (GP) [159] is a traditional surrogate model for Bayesian optimization. Generally speaking, GP is a generalization of Multivariate Gaussian distribution [161], where the mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$ are redefined by a mean function $\mu(p)$ and a covariance kernel function $K(p, p')$ of any two observations[2]. The objective function f (expensive) is modeled as a GP, and can

[2]Note that a function is an infinite vector.

Algorithm 3: Bayesian Optimization for AutoML optimization

Input: $\mathbb{O}$: sequence of operators, Λ: hyperparameter spaces f : objective function, n_{init}: initial sample size,n_{ei}: number of samples to be sampled in the maximize expected improvement (EI) step, e_{ei}: number of samples to be suggested at each BO' iteration,

Output: p^*: the best ML pipeline configuration

$\Theta \leftarrow$ INIT SAMPLING$(\mathbb{O}, \Lambda, n_{\text{init}})$ `// typically done via Random sampling in Algorithm 2`

`// Evaluate all configurations on the given objective function` f`, their performances are saved to` Δ

$\mathcal{P} \leftarrow$ TRAIN $\mathcal{P}(\Theta, \Delta)$

while *not terminate* **do**

  $\Theta_{\text{new}} \leftarrow \mathcal{P}$.MAXIMIZE EI$(n_{\text{ei}}, e_{\text{ei}})$ `// EI is Expected improvement`

  $\Delta_{\text{new}} \leftarrow \varnothing$

  foreach $p_i \in \Theta_{new}$ **do**

    $\Delta_i \leftarrow f(p_i)$

    $\Delta_{\text{new}} \leftarrow \Delta_{\text{new}} \cup \Delta_i$

  end

  $\Theta \leftarrow \Theta_{\text{new}}$

  $\Delta \leftarrow \Delta_{\text{new}}$

  $\mathcal{P} \leftarrow$ TRAIN $\mathcal{P}(\Theta, \Delta)$ `// Re-train model`

end

return $p^* \in \Theta$ `// return the best found configuration`

be defined as $f(p) \sim \mathcal{GP}(\mu(p), K(p, p'))$. In other words, for each sample p, a Gaussian process defines a mean μ and standard deviation σ within a Gaussian distribution. The squared exponential function is a commonly used kernel function: $K(p, p') = \sigma_f^2 \exp[\frac{-(p-p')^2}{2l^2}]$ where l denotes the length scale[3] and σ_f^2 denotes the output variance. Subsequently, the predictive distribution for an unseen configuration $\mathcal{P}(f_{\text{new}}|\Theta, \Delta, p_{\text{new}})$ follows a Gaussian distribution. Hence, the mean and variance can be computed as follows:

$$\mu(p_{\text{new}}) = \mathbf{k}_* \mathbf{K}^{-1} \Delta; \sigma^2(p_{\text{new}}) = k_{**} \mathbf{k}_* \mathbf{K}^{-1} \mathbf{k}_*^{\text{T}} \tag{3.1}$$

where the covariance matrices are calculated as $\mathbf{k}_* = [k(p_1, p^*), \ldots, k(p_t, p^*)]$, $\mathbf{K} = [k(p_i, p_j)]_{\forall i,j \in \{1,\ldots,t\}}$ and $k_{**} = k(p_{\text{new}}, p_{\text{new}})$.

[3]The length scale establishes a point's *area of influence.* inside the parameter space, where the effect of an observation diminishes as one moves away from it.

Random forests

Random forests (RF) [72], is an algorithm used in Machine Learning for regression and classification. It is employed in SMAC [25] as the mere surrogate model[4]. Fundamentally, RF can be considered as a collection of regression (decision) trees. The historical data (Θ, Δ) were randomly divided into multi decision trees with few features. The trees in the forest individually score the unseen configuration candidate p_{new} ($p_{\text{new}} \in \mathcal{M}$), and the final result is based on the majority of the votes. A major limitation of RF is that it does not provide an estimate of the variance in its predictions. When adopting it in the BO scenario, SMAC [25] uses the empirical variance in the predictions of trees in the ensemble. Hence, mean μ and variance σ^2 for the new candidate p_{new} are computed as the empirical mean and variance of each tree r in the forest of B trees:

$$\mu(p_{\text{new}}) = \frac{1}{|B|} \sum_{r \in B} r(p_{\text{new}}); \sigma^2(p_{\text{new}}) = \frac{1}{|B|-1} \sum_{r \in B} \left(r(p_{\text{new}}) - \mu(p_{\text{new}})\right)^2 \quad (3.2)$$

where B denotes a set of trees, r denotes a tree in the forest, i.e., $r \in B$. $r(p_{\text{new}})$ is the predicted value of the new (unseen) configuration p_{new} by a tree r.

Tree-structured Parzen Estimator

Tree-structured Parzen Estimator (TPE) [24] is another alternative to a GP, which is a tree-based model by using the Parzen-window density estimators [162], [163]. Instead of modeling the distribution of the true objective function f, TPE models the likelihood $\mathcal{P}(\mathcal{H}|f)$ by using the parzen window kernel density estimator. In this setting, the evaluated configurations are split into two density distributions of a well $l(p)$ and a badly $g(p)$ performing set depending on whether its performance is below or above a predefined threshold[5] α. We note that $l(p)$ and $g(p)$ probability models are usually represented by Gaussian Mixture Models (GMMs) or Kernel Density Estimation (KDE) independently. Hence, we have two means, that is, $\mu_{l(p)}$ for the mean of $l(p)$ and $\mu_{g(p)}$ for the mean of $g(p)$, and two variances, that is, $\sigma^2_{l(p)}$ for the variance of $l(p)$ and $\sigma^2_{g(p)}$ for the variance of $g(p)$, the computation of these values depend on the models used. For the detailed discussion and relevant formulas, we refer the interested reader to [24] for further discussion on the Hyperopt framework, a well-known implementation of TPE, [164] and [165] for further discussion on KDE and GMMs.

[4]Note that the property of regression trees is supported conditional variables domains, while GP family currently do not.

[5]By default, $\alpha = 25\%$.

3.1.3.2 Acquisition Function

Bayesian optimization is designed to find a global optimum for an optimization problem that may have many local optima. However, as noted in the previous section, the surrogate model only approximates the true objective function, and its estimations may be imperfect. This leads to the following question: Should we exploit the most known search area or explore other areas that are less known? The so-called Acquisition Function (AF, *or infill-criterion*) [155] is designed to answer this question, aiming to achieve a trade-off between exploration and exploitation. Generally, AF computes the expected improvement value from the mean and covariance (uncertainty) estimated by a regression model. Therefore, we can choose a new configuration candidate for evaluation by maximizing the expected improvement values.

Although many acquisition functions have been proposed [38], [157], [160], [166]–[172], the *Expected improvement* (EI) [156] is the most popular acquisition function for BO and remains the default AF in BO packages, such as SMAC [25], Spearmint [28], SPO [27] and TPE [24], [158]. EI balances the trade-off between exploration and exploitation via the expectation of the improvement function over the best-found value $\Delta^*_{(t)}$ at time step t as $I_t(p) = \max\{0, \hat{f}(p) - \Delta^*_{(t-1)}\}$, where $\Delta^*_{(t-1)} = \max(\Delta^0, \ldots, \Delta^{t-1})$ and $\hat{f}(p)$ denotes the predicted performance of the setting p via surrogate model $\mathcal{P}$. The EI is thus defined as:

$$\mathbb{E}[I_t(p)] = \int_0^\infty I_t(p)\, d\mathcal{P} \tag{3.3}$$

Let us denote by $z = z_{(t-1)}(p) = \frac{\mu_{(t-1)}(p) - \Delta^*_{(t-1)}}{\sigma_{(t-1)}(p)}$, we obtain the closed-form AF by taking the expectation via the improvement function $I_t(p)$ as:

$$\mathbb{E}[I_t(p)] = \sigma_{(t-1)}(p)\phi(z) + [\mu_{(t-1)}(p) - \Delta^*_{(t-1)}]\Phi(z) \tag{3.4}$$

where $\mu(p)$ and $\sigma(p)$ denoted the mean and standard deviations; $\phi(.)$ and $\Phi(.)$ are the probability density function (p.d.f) and the cumulative distribution function (c.d.f) of the standard normal distribution. Hence, the next setting is selected by maximizing the EI:

$$p_{\text{new}} = \underset{p \in \mathcal{M}}{\operatorname{argmax}}\, \mathbb{E}[I_t(p)] \tag{3.5}$$

where $\mathcal{M}$ denotes the AutoML search space (see Section 1.1).

3.2 Multi-fidelity approaches

As mentioned in Section 1.1, the k-fold cross-validation is typically used when evaluating a pipeline configuration to avoid the over-fitting problem (see Section 2.1.3). However, if the performance of a particular configuration is poor when evaluated on the first folds, it is likely to not perform well on the rest of the cross-validation fold [173]. Hence, we should not invest further computational resources in this configuration or redistribute the saved resources to the most promising configurations. A class of optimization methods named *multi-fidelity approaches* intends to save computational resources and speed up the optimizing process by evaluating configurations on a subset of the input data [174], [175], limiting iterations [13], or using a subset of features [8]. In this study, we use the term k-fold cross-validation (see Figure 2.5 for reference), then the multifidelity limits the use of a few folds of the cross-validation folds, that is, using i folds ($i \leq k$). The configuration candidates in this class of methods tend to be evaluated faster on fewer folds than the approaches in Section 3.1. Hence, we use the term *function call* to indicate a one-time access to a configuration on one fold.

This thesis reviews two commonly used classes of methods aiming at reducing computational effort, namely: (1) Racing procedure approaches and (2) Bandit-based approaches.

3.2.1 Racing procedure

Hoeffding Races [32], [173] were the first version of the racing procedure. It was initially designed to find the best machine learning model for a set of problem instances (here, we use the term k-folds instead) in the supervised machine learning domain. To reduce the computational cost of poor configurations, a (pairwise) statistical test (e.g., t-test, Friedman-test [176]) is used to determine poor configurations to be terminated as soon as enough statistical evidence arises against them, that is, the ones that are significantly worse than the best.

Although a number of racing procedure variants have been developed, such as F-Race [177], [178], Sampling F-Race [179], and Iterated racing (irace) [33], [180], [181] ; irace is the most the latest of this class. It is particularly well-suited for Hyperparameter Optimization [13] and AutoML optimization [182]. Hence, we present the irace algorithm in greater detail in the following section. For details on other methods, we refer the reader to Birattari (2009) and the book chapter by Hoos (2012).

3.2.1.1 Iterated racing (irace)

In contrast to Hoeffding Races, the later variants of the racing procedure add a rank-based Friedman test (i.e., Friedman two-way analysis of variance by ranks) to determine if there is any significant difference between configurations. If any differences were found, pairwise comparisons were performed with the best candidate. Irace also followed this procedure. The detailed outline of irace is shown in Algorithm 4. Irace first initializes parameters (lines 1-2). The first round uses the random sampling method in Algorithm 2, generates a set of $N_j (N_j = \lfloor \frac{B_j}{T^{\text{first}} + T^{\text{each}}} \rfloor)$ configurations (line 5). A racing procedure (line 6) is used to discard poorly performing configurations, based on their evaluated performance on T^{first} folds. This race relies on the Friedman test [176] and Conover post-hoc test [183] with a significance level α[6].

After the first race, a new race is initialized by taking the remaining budget $(B - B^{\text{used}})$ and the number of remaining races $(N^{\text{iter}} - j + 1)$ (lines 11-12). Next, EliteBasedSampling generates a set of N_j configurations by sampling the set of surviving configurations Θ^*, from the previous race. For every new sample, the sampling procedure repeats as follows:

1. One sequence of operators will be chosen as the parent sequence $(\mathcal{A}_1, \ldots, \mathcal{A}_z)^{\text{parent}}$ (see Figure 1.2 for the used notation) for this new race, with a probability ρ^{parent} that is based on its configuration p^{parent}, $(p^{\text{parent}} = (\mathcal{A}_{1,\lambda}, \ldots, \mathcal{A}_{z,\lambda})^{\text{parent}} \in \Theta^*)$ and its rank r^{parent} over the surviving set Θ^*. The probability ρ^{parent} is computed as follows:

$$\rho^{\text{parent}} = \frac{2 \cdot |\Theta^*| - r^{\text{parent}} + 1}{|\Theta^*| \cdot (|\Theta^*| + 1)} \tag{3.6}$$

2. The corresponding hyperparameters λ to $(\mathcal{A}_1, \ldots, \mathcal{A}_z)^{\text{parent}}$ are sampled by either a truncated normal distribution for numerical hyperparameters, or a discrete distribution for categorical hyperparameters[7].

3.2.2 Bandit-based approaches

The class of bandit-based approaches is similar to the racing procedure in that they terminate the worst pipeline configurations early. However, compared with the racing procedure, they differ in two ways:

[6] By default $\alpha = 0.05$.

[7] Ordinal hyperparameters are considered as numerical.

Algorithm 4: Iterated racing algorithm

Input: $\mathbb{O}$: sequence of operators, Λ: hyperparameter spaces, f : objective function, $\mathbf{I}$: set of problem instances (or *set of k folds*), T^{first}: the number of instances (*folds*) needed to do the first test, T^{each}: the number of instances (*folds*) to test on the later round (by default $T^{\text{each}} = 1$), B: total budget (*maximum number of function calls*)

Output: Θ^*: set of best configurations

$N^{\text{param}} \leftarrow |\mathbb{O}|+|\Lambda|$ // number of parameter spaces equal to the total number of operators and the total number of algorithms' hyperparameters

$N^{\text{iter}} \leftarrow \lfloor 2 + log_n N^{\text{param}} \rfloor$ // number of races to be executed

// THE FIRST RACE

$j \leftarrow 1$ // $j = 1, \ldots, N^{\text{iter}}$

$B_j \leftarrow \frac{B}{N^{\text{iter}}}$ // compute budget for the first round

$N_j \leftarrow \lfloor \frac{B_j}{T^{\text{first}}+T^{\text{each}}} \rfloor$ // number of configuration to be sampled at the first race

$\Theta^j \leftarrow$ RANDOM SAMPLING$(\mathbb{O}, \Lambda, N_j)$ // see Algorithm 2

$\Theta^* \leftarrow$ RACE$(\Theta^j, B_j, T^{\text{first}}, \mathbf{I})$ // determine the set of good configurations

$B^{\text{used}} \leftarrow B_j$ // used budgets

// LATER RACES

while *not terminate* **do**

  $j \leftarrow j + 1$

  $B_j \leftarrow \frac{B-B^{\text{used}}}{N^{\text{iter}}-j+1}$ // compute budgets for the current race

  $N_j \leftarrow \lfloor \frac{B_j}{T^{\text{first}}+T^{\text{each}} \times \min\{5,j\}} \rfloor - |\Theta^*|$ // number of configurations to be sampled for the current race

  $\Theta^j \leftarrow$ ELITEBASEDSAMPLING$(\mathbb{O}, \Lambda, N_j, \Theta^*)$

  $\Theta^j \leftarrow \Theta^j \cup \Theta^*$

  $\Theta^* \leftarrow$ RACE$(\Theta^j, B_j, T^{\text{each}}, \mathbf{I})$ // determine the set of good configurations

  $B^{\text{used}} \leftarrow B^{\text{used}} + B_j$ // update used budgets

end

return Θ^*

1. All configurations were compared directly based on their evaluated performance instead of using a statistical procedure.

2. The number of rounds and budget can be estimated based on the input budgets, that is, the budgets for each round are equally assigned, but later rounds have fewer candidates than the previous rounds.

In the following, we present *Successive Halving* [34] in more detail and outline its limitations. Next, we discuss two variants of Successive Halving to overcome these limitations [35], [36].

3.2.2.1 Successive Halving

Jamieson and Talwalkar (2016) introduced *Successive Halving* as a simple yet efficient algorithm for multi-fidelity optimization. The outline of *Successive Halving* is summarized in Algorithm 5. Here, we slightly adapt the algorithm for AutoML optimization. That is, we use the term AutoML search space instead of hyperparameter space, and use it for the k-fold cross-validation scenario, that is, at least one candidate will be assigned a sufficient budget to evaluate all k folds, and no configuration can have more than that budget. It requires a budget (finite value) B, i.e., the maximum number of function calls, the number of configurations n, the maximum number of folds that can be used for a single configuration R. The procedure pre-computes the number of rounds t to be executed (line 1). The value of t is then recomputed by using line $2-7$ to find an appropriate value of t for the provided budget B and n when using the discard ratio η. Next, a set of n configurations is generated randomly and saved to Θ_r (line 3). For each round, the budget for a configuration is computed in line 13, i.e., either a subset of data or $B_r \in \{1, \ldots, k\}$ folds ($B_r << k$). Herein, we slightly modify to adapt to the above mentioned scenario to ensure no configuration has more than R folds and less than 1 folds. Next, all configurations $p \in \Theta_r$ are assessed on B_r folds. At the end of the round, we only keep the top $\frac{1}{\eta}$ configurations[8] based on their performances to go to the next round (line 16). The successive procedure is then repeated until the last round is done. Lastly, the best-found configuration is returned (line 19).

3.2.2.2 Hyperband

Successive Halving requires the number of configurations n and budgets B, e.g., the number of function calls, as input parameters. Assume that we have a fixed budget B, e.g., total number of function calls, the proportion of $\frac{B}{n}$ leads to a consideration of whether we should consider (1) more configurations (large n) in the race with small average folds or (2) a small number of configurations (small n) with higher average folds. [35] pointed out that in practice, the problem itself might have some noise, i.e., the accuracy rate on folds might be significantly different. If the noise is

[8]We note that the default proportion discard of *half* was changed to *one third* with the recent studies [35]–[37].

Algorithm 5: Successive Halving algorithm

Input: $\mathbb{O}$: sequence of operators, Λ: hyperparameter spaces, f : objective function, $\mathbf{I}$: set of problem instances (or set of k folds), B: total budget (*maximum number of function calls*), n: number of configurations, η: Proportion discard ratio ($\eta = 2$ by default), R: maximum number of instances (folds) that can be allocated to a configuration ($R = |\mathbf{I}|$, by default).

Output: Θ_t

$t \leftarrow \log_\eta(\min\{R, n\})$ `// pre-compute the number of rounds based on` R, n `and` η
foreach $t \leq log_\eta(min\{R, n\})$ **do**
  if $nR(t+1)\eta^{-t} \leq B$ **then**
    Evaluate whether the given budget B are sufficient to accommodate the number of rounds t.
    return t `// number of rounds to be executed`
  else
    $t \leftarrow t - 1$
  end
end
$\Theta_r \leftarrow$ RANDOM SAMPLING$(\mathbb{O}, \Lambda, n)$ `// randomly create` n `configurations using Algorithm 2`
$R_{\text{remain}} \leftarrow R$ `// number of instances/folds remain unevaluated.`
$r \leftarrow 0$
while $r \leq t$ **do**
  $B_r \leftarrow \min(\max(\lfloor (R\eta^{r-t} \rfloor, 1), R_{\text{remain}})$ `// Budget for a surviving configuration in the current round.`
  foreach $p \in \Theta_r$ **do**
    Assess the configuration p on B_r folds, which have not been evaluated so far.
  end
  $\Theta_{r+1} \leftarrow$ SELECT TOP $\lfloor \frac{|\Theta_r|}{\eta} \rfloor$ in Θ_r `// keep` $1/\eta$ `good configurations in terms of their corresponding observed performances`
  $R_{\text{remain}}, r \leftarrow R_{\text{remain}} - B_r, r + 1$
end
return $p \in \Theta_r$ `// return the best found configuration`

low, we can quickly determine the quality of the configurations on fewer folds. We can select a large number of configurations to maximize the possibility of finding the optimal solution. Otherwise, we should consider fewer configurations, but we will evaluate them in more detail.

Exploiting this finding, [35] proposed to have an outer loop of *Successive Halving*,

which will consider a different proportion of $\frac{\text{total budgets}}{\text{number of configurations}}$, where the number of configurations on each outer loop is reduced. The complete algorithm is outlined in Algorithm 6. The idea behind Hyperband is that it divides resources into brackets, e.g., $\{N^{\text{iter}}, N^{\text{iter}}-1, \ldots, 0\}$, with different configurations and executes *Successive Halving* as a sub-program, that is, the first loop executes *Successive Halving* with many configurations, but most of them will validate fewer folds. In contrast, the last loop handles fewer configurations but will be validated on most folds. This outer loop is in lines $4-13$.

Algorithm 6: Hyperband algorithm

Input: $\mathbb{O}$: sequence of operators, Λ: hyperparameter spaces, f : objective function, $\mathbf{I}$: set of problem instances (or set of k folds), B: total budget (*maximum number of function calls*), η: Proportion discard ratio ($\eta = 3$ by default), R: maximum number of instances that can be allocated to a configuration ($R = |\mathbf{I}|$, by default).

1 $N^{\text{iter}} \leftarrow \lfloor log_{\eta}(R) \rfloor$
2 $\Theta \leftarrow \varnothing$ `// set of configurations`
3 $B_{\text{remain}} \leftarrow B$
4 **foreach** $r \in \{N^{iter}, N^{iter}-1, \ldots, 0\}$ **do**
5 $\quad n_r \leftarrow \lceil \frac{B}{R} \times \frac{\eta^r}{r+1} \rceil$ `// number configurations to be sampled`
6 $\quad$ **if** $r > 0$ **then**
7 $\quad\quad B_r \leftarrow \frac{B}{N^{\text{iter}}}$ `// total budget for the current round`
8 $\quad$ **else**
9 $\quad\quad B_r \leftarrow B_{\text{remain}}$ `// total budget for the last round`
10 $\quad$ **end**
11 $\quad \Theta \leftarrow \Theta \cup \text{SuccessiveHalving}(\mathbb{O}, \Lambda, f, \mathbf{I}, B_r, n_r, \eta, R)$
$\quad$ `// SuccessiveHalving (Algorithm 5) is used as a subroutine.`
12 $\quad B_{\text{remain}} \leftarrow B_{\text{remain}} - B_r$ `// remaining budget`
13 **end**
14 **return** $p^* \in \Theta$ `// return the best found configuration`

Finally, both *Hyperband* and *Successive Halving* are considered fast random search methods owing to the use of random sampling for generating configurations. Therefore, they also inherited the major limitation of random search for proposing new configurations but were not improved to take information accumulated over the search history into account, such as Evolutionary Strategies [108], Bayesian Optimization [159], [184], which can propose configurations based on the assessed points so far. Instead of randomly proposing new configurations, BOHB [36] and DACOpt [37] proposed to use combine a Bandit approach and Bayesian

Optimization to maximize expected improvement. BOHB replaces the random sampling step in Successive Halving by TPE at the step after the first round. In contrast, DACOpt uses Successive Halving as an outer loop that can suggest good search areas for Bayesian optimization.

CHAPTER 4

Setup of Benchmark Experiments

This chapter aims to evaluate the robustness and general applicability of optimization approaches empirically. It is worth noting that there are two common methods typically used to compare AutoML approaches. The first approach is designed to compare the underlying optimizers in a predefined scenario to identify the most effective optimization approach for the AutoML problem. This experimental setup aims to determine which optimizer can achieve the highest performance for a given dataset, using a similar experiment setting within a finite budget. AutoML tools are complex systems that incorporate meta-learning, pruning, early stopping, and evaluation strategies to prevent overfitting. Typically, these tools are evaluated based on their performance with unseen data. As a result, benchmark experiments include both of the above approaches. We introduce two sets of benchmark experiments to investigate the performance of AutoML optimization algorithms, which will be used in the later chapters. In order to increase comparability, we use agreed-on datasets, which are often different data sets, and standardized search spaces for benchmarking purposes [22], [185]. We conduct the experimental setup with a total of 117 benchmark datasets on two scenarios with optimization of 2 operators (Section 4.2) and 6 operators (Section 4.3).

The first scenario focuses on addressing the common problem in real-world applications known as class imbalance. It involves 44 well-known binary imbalanced datasets from the Keel collection [186]. These datasets represent real-world scenarios marked by imbalanced class distributions and are used in many class imbalanced studies [47], [74], [75], [187], [188]. In addition to the selected datasets, carefully crafted search spaces have been designed, including a collection of 21 options of resampling techniques thoughtfully combined with 5 commonly used classification algorithms customarily used to solve class imbalanced problems, each characterized by a carefully selected range of hyperparameters. The geometric mean is employed as the performance metric.

Simultaneously, the second dataset, drawn from the OpenML repository [189], comprises 73 well-known datasets in the AutoML community. These datasets come highly recommended by recent studies [22] and encompass a broad spectrum of problem domains and complexities, making them valuable for comparing the efficiency of AutoML optimization approaches. The second search space is directly inspired by the influential reference [22], aligning seamlessly with prior AutoML research. It includes the same trusted datasets, algorithmic selections, and their corresponding hyperparameter configurations, ensuring a consistent benchmarking framework. Furthermore, the performance metric follows the recommendations from the same reference, reinforcing the credibility and relevance of the benchmark experiments in contemporary AutoML investigations. These benchmark datasets and their associated search spaces offer a robust foundation for comparing AutoML optimization approaches.

4.1 Benchmarking methodology

Both benchmark experiments introduced in this chapter focus on AutoML optimization problems for classification problems. The problem of AutoML is precisely defined in Chapter 1 (Section 1.1). It is worth mentioning that we will use the established notations and equations in this chapter. As a recap, considering a classification problem with a dataset $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$, where $\mathbf{x} = \{x_1, \ldots, x_k\}$ is a vector of k features x and y represents a label. The general problem in this chapter is to find the best ML pipeline p that trains on dataset $\mathcal{D}$ to produces ML model P. This model is designed to transform a set of features $\mathbf{x} \in \mathbb{X}$ into a target value $y \in \mathbb{Y}$. All experiments were repeated 10 times with different random seeds to account for the nondeterministic effects of the involved algorithms. The performance of each ML pipeline configuration was determined at $i^{(th)}$ fold of the k-fold cross-validation, denoted as:

$$f(p, \mathcal{D}^{(i)}_{\text{train}}, \mathcal{D}^{(i)}_{\text{valid}}) = \frac{1}{|\mathcal{D}^{(i)}_{\text{valid}}|} \sum_{j=1}^{|\mathcal{D}^{(i)}_{\text{valid}}|} R(\hat{y}_j, y_j) \tag{4.1}$$

where $R(\hat{y}, y)$ denotes a metric that returns the accuracy of the value $\hat{y}$ predicted by the pipeline compared with the real value y. Then, f denotes the performance of pipeline configuration p when trained on training dataset $\mathcal{D}_{\text{train}}$ and evaluated on validation dataset $\mathcal{D}_{\text{valid}}$. The ML pipeline optimization problem is then used

to determine the best setting p^* that maximizes the objective function f with a given accuracy metric, for example, the geometric mean, accuracy rate.

$$p^* = \text{argmax} \frac{1}{k} \sum_{i=1}^{k} f\left(p, \mathcal{D}_{\text{train}}^{(i)}, \mathcal{D}_{\text{valid}}^{(i)}\right) \tag{4.2}$$

More precisely, the two scenarios were adapted from the experimental setups of [47] and [22]. Furthermore, we follow the parameter settings of those studies, including datasets, search space, k-fold cross-validation setup, train/test split, and performance metric – all parameter settings are summarized in Table 4.1:

1. The first experiment is limited to exactly 500 iterations[1]. Our initial experiments show no significant improvements before hitting this iteration limit. The performance of each configuration is determined using a 5-fold cross-validation technique, calculated as:

$$\Delta_p = \frac{1}{k} \sum_{i=1}^{k} \Delta_p^{(i)} \tag{4.3}$$

   where $\Delta_p^{(i)}$ denotes the performance of p for a *function call* (see Figure 2.5) on the i^{th} data fold, i.e., the performance of the ML model that uses configuration p trained and evaluated on the i^{th} data fold $\mathcal{D}_{\text{train}}^{(i)}$ and $\mathcal{D}_{\text{valid}}^{(i)}$, correspondingly. Therefore, when an optimization process is over, the performance of the underlying optimizer is the highest Δ_p^*, and the corresponding configuration is considered the best p^*. Because each optimizer had 10 independent runs, each optimizer had 10 configurations (they might be different) at the end of the experiment.

2. The second experiment is limited to 1 hour[1]. The dataset is split into a training dataset $\mathcal{D}_{\text{train}}^{(70\%)}$ for the optimization process and a test dataset $\mathcal{D}_{\text{test}}^{(30\%)}$ for calculating the performance of the optimizer when the optimization process is over. In other words, only $\mathcal{D}_{\text{train}}^{(70\%)}$ is involved during the optimization procedure. First, we do a similar procedure on $\mathcal{D}_{\text{train}}^{(70\%)}$ as the first experiment to determine the best configuration within the tuning budget of 1 hour, except k becomes 4 instead of 5, as we strictly follow the experiment procedure of [22] for a fair comparison with this study.

[1]This dual approach is chosen because optimization methods are commonly compared in terms of function evaluations, whereas AutoML tools are typically assessed based on their performance within a specified wall-time budget.

Table 4.1: Parameter settings

	1^{st} experiment	**2^{nd} experiment**
Optimizer parameters		
- Total budgets	500 (func. eval.)	1 (hour)
Experimental parameters		
- Search space	2 operators	6 operators
- Number of datasets	44	73
- Performance metric	Geometric mean (GM)	Accuracy rate (Acc)
- k-folds cross validation *(for optimization)*	5	4
- Train/test split	No	train: 70% test: 30%
** train set uses for the optimization process test set uses to compute final result once the optimization process is done*		
- Final results	Average GM over k-folds	Accuracy rate of the unseen test set

Once the best configuration p^* is found, we manually build an ML model configured by p^*. The performance of the underlying optimizer is then the performance of that ML model when trained on $\mathcal{D}_{\text{train}}^{(70\%)}$ and tested on $\mathcal{D}_{\text{test}}^{(30\%)}$. Consequently, at the end of the experiment, each optimizer has 10 configurations as 10 runs. We note that each run starts from the train/test split step with a different random seed.

4.2 First experiment: class-imbalanced classification problems with two operators

This problem is based on a machine learning pipeline optimization problem with two operators:

- A collection of 21 options of resampling techniques, i.e., 20 resampling algorithms belong to 3 groups– under-resampling, over-resampling, and combine-resampling, and a "no resampling" option.
- A set of 5 commonly used classification algorithms customarily used to solve class imbalanced problems, i.e., Support Vector Machines (SVM), Random

Forest (RF), k-Nearest Neighbors (KNN), Decision Tree (DT) and Logistic Regression (LR).

- A set of 44 binary class imbalanced datasets from Keel collection [186].

In this section, we briefly introduce the datasets (Section 4.2.1) and resampling techniques (Section 4.2.2) used in this work. We then specify the experimental procedure (Section 4.2.3). Finally, detailed information on the hyperparameters used is provided in Section A.1.1 of the Appendix.

4.2.1 Datasets

For this study, 44 binary class imbalanced datasets from the Keel repository [186] are used. Their *Imbalance Ratio* (IR), i.e., the ratio of the number of majority class instances to that of minority class instances, ranges here from 1.82 to 129.44. Figure 4.1 shows the 44 examined datasets presenting the imbalance ratio (#IR) on the x-axis and the number of samples (#samples) on the y-axis; where the color of the symbols denotes the number of attributes for each dataset. A full list of datasets is provided in Section (A.2) of the appendix.

4.2.2 Resampling Algorithms

The resampling algorithms were designed to handle the class imbalance scenario by producing balanced datasets. The resampling algorithms used in our experiments can be arranged into three groups:

1. *Over-resampling (7 algorithms)*: In the imbalanced learning domain, over-resampling technique balances the class distribution via producing synthetic minority samples. SMOTE is the most famous resampling technique and generates synthetic samples based on random interpolation between the chosen minority samples and their k-nearest neighbors. Various SMOTE-based extensions have been proposed to further improvement on the SMOTE basis. For example, ADASYN [190] focused on the harder-to-learn samples and BorderlineSMOTE [191] emphasized the borderline samples. Other over-resampling approaches considered in this experiment are KMeansSMOTE [192], SMOTENC [17], SVMSMOTE [19] and RandomOverSampler [193].

2. *Under-resampling (11 algorithms)*: In contrast, under-resampling approach balances the class distribution by removing majority samples. A Tomek link is defined as a pair of samples from different classes which are the nearest

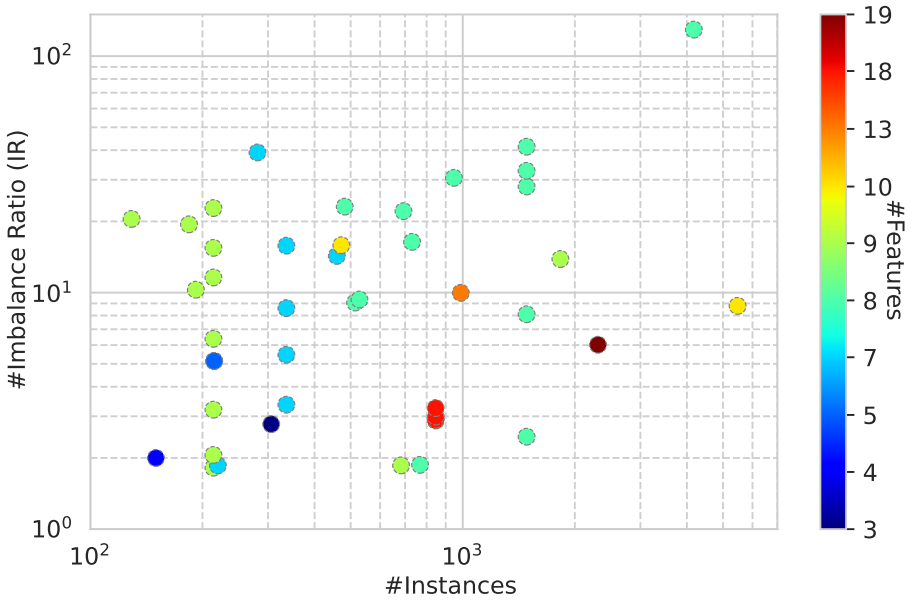


Figure 4.1: Overview of the characteristics of the datasets. The scatter plot shows the Imbalance Ratio (#IR) and the number of samples (#Instances) for all 44 datasets on a logarithmic scale. The color indicates the number of attributes (#Features).

neighbors for each other [194]. The undersampling method TomekLinks removes the Tomek links in the dataset in order to produce a clear decision boundary. OneSidedSelection [81] first removes noisy and borderline majority samples, then removes the safe majority samples which have limited contribution for building the decision boundary with the CondensedNearestNeighbour Rule [195]. Other under-resampling methods considered in this experiment are CondensedNearestNeighbour, EditedNearestNeighbours [196], RepeatedEditedNearestNeighbours [197], AllKNN [197], InstanceHardnessThreshold [198], NearMiss [199], NeighbourhoodCleaningRule [200], ClusterCentroids [201], and RandomUnderSampler [202].

3. *Combine-resampling (2 algorithms)*: In order to balance the class distribution, the combine-resampling integrates both over-resampling and under-resampling approaches, i.e., removing the majority samples and creating synthetic minority samples. For example, SMOTETomek first oversamples

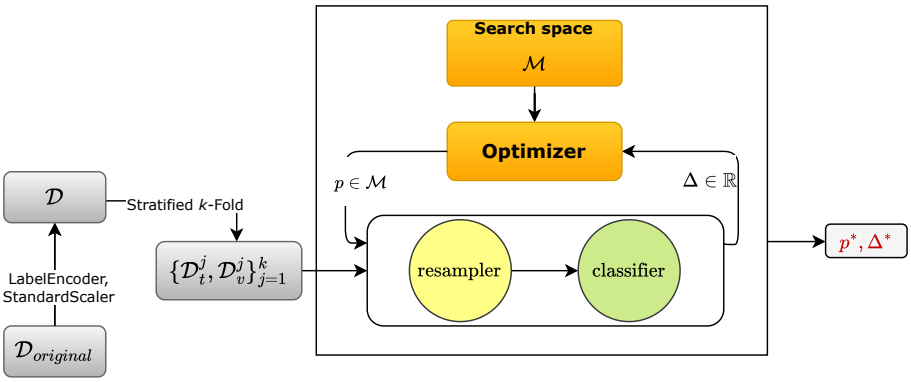


Figure 4.2: Flowchart of the experimental setup. The process begins with data pre-processing of the input dataset using LabelEncoder and StandardScaler. Next, we apply the 5-fold cross-validation to overcome the overfitting problem. The outcome is fed into the optimization phase, which has a budget of 500 function evaluations. The optimizer handles the process by generating a new configuration $p \in \mathcal{M}$ at each iteration. The objective function, in the rounded rectangle consisting of resampling and classification algorithms, is then parameterized by p and computes its performance, i.e., geometric mean, on $\{\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{valid}}\}_{j=1}^k$. Lastly, the performance $\Delta \in \mathbb{R}$ is returned to the optimizer as an extended input to generate a new configuration.

the minority class using SMOTE, after which the Tomek links for the after-sampled samples are removed. Similar to SMOTETL, SMOTEENN first oversamples the minority class with SMOTE. Thereafter, the Wilson's Edited Nearest Neighbors (ENN) was used to remove the sample that has a different class from at least two of its three nearest neighbors [203].

The setup also allows a "no resampling" option. The resampling algorithms are implemented in the Python package IMBALANCED-LEARN[2][48].

4.2.3 Implementation details

The overall structure of our implementation is summarized in Figure 4.2. The process begins with data pre-processing of the input dataset. A 5-fold cross-validation is then applied to overcome the overfitting problem. The outcome is fed into the second phase, which consists of the resampling and classification processes. The complete pseudo-code of this flowchart is elaborated in Algorithm 7.

Algorithm 7 consists of the following two steps:

[2] `https://github.com/scikit-learn-contrib/imbalanced-learn` (version 0.7.0)

Algorithm 7: Experimental setup

Input: $\mathbb{O} = (\mathcal{O}_{\text{resampler}}, \mathcal{O}_{\text{classifier}})$: sequence of operators, Λ: hyperparameter spaces, r: Random seed, k: Number of folds, B: Number of iterations
Output: p^*: the best configuration, Δ^*: GM achieved by p^*
Data: dataset $\mathbf{D}$

```
1 D ← DataPreProcess(D)
  // DataPreProcess includes LabelEncoder, StandardScaler
2 {D_train, D_valid}_{j=1}^k ← StratifiedK-fold(D, k, r)
3 Optimizer ← Optimizer.init(𝕆, Λ, f, r, {D_train, D_valid}_{j=1}^k) // initialize
    optimizer
4 p*, Δ* ← Optimizer.optimize()
```

- Preprocessing (line 1-2): We need to apply data preprocessing since machine learning models require input and output data to be numeric. Thus, we used the Label encoder[3] to encode any categorical data to a number for the input dataset. Then, we apply Standard Scaler[3] on the encoded dataset to have zero mean and a standard deviation of one (line 1). Next, stratified k-fold cross-validation[3] using $k = 5$, commonly used in the literature, is used.

- Hyperparameter optimization (line 3-4): All parameters of HPO are initialized (line 3), taking values from the provided input including sequence of operators $\mathbb{O}$, hyperparameter spaces Λ, random seed r, number of iterations B, objective function f and k folds of the examined dataset. The algorithm then optimizes the problem until the number of function evaluations reaches 500.

The computation of the objective function is presented in Algorithm 8. It elaborates further steps presented in the rounded rectangle in Figure 4.2. The input is a parameter setting generated by the optimizer consisting of a random seed r and ML pipeline configuration p. The configuration p consists of two parts: the choice of resampler represented by p_{re_0}, and classifier denoted by p_{cls_0}, together with their corresponding hyperparameter settings $\{p_{re_1}, \ldots, p_{re_q}\}$ and $\{p_{cls_1}, \ldots, p_{cls_p}\}$.

For a fold of the examined dataset, the computation of an evaluation has the following steps:

- Step 1 (line 2-3): Resampler and classifier are initialized, using values of the configuration p and random seed r.

[3] Label encoder, Standard scaler and Stratified k-fold cross-validation are implemented in the python library SCIKIT-LEARN (version 0.23.2).

Algorithm 8: Objective function

Input: Hyperparameter configuration p generated by the optimizer; r: Random seed

// p= ($\underbrace{p_{re_0}, p_{re_1}, \ldots, p_{re_q}}_{\text{RESAMPLER}}, \underbrace{p_{cls_0}, p_{cls_1}, \ldots, p_{cls_p}}_{\text{CLASSIFIER}}$)

Data: $\{\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{valid}}\}_{j=1}^{k}$

1 **foreach** $\{\mathcal{D}_{train}^{(j)}, \mathcal{D}_{valid}^{(j)}\} \in \{\mathcal{D}_{train}, \mathcal{D}_{valid}\}_{j=1}^{k}$ **do**

  `// Build resampler and classifier models`

2 RESAMPLER ← Parameterize RESAMPLER p_{re_0} with the hyperparameters $\{p_{re_1}, \ldots, p_{re_q}\}$ and random seed r

3 CLASSIFIER ← Parameterize CLASSIFIER p_{cls_0} with the hyperparameters $p_{cls_1}, \ldots, p_{cls_p}$ and random seed r

4 $\mathcal{D}_{\text{train}}^{(j)} \leftarrow$ RESAMPLER($\mathcal{D}_{\text{train}}^{(j)}$)

5 $\delta_j \leftarrow$ CLASSIFIER.LEARN($\mathcal{D}_{\text{train}}^{(j)}$).EVALUATE($\mathcal{D}_{\text{valid}}^{(j)}$)

6 **return** $\Delta \leftarrow \frac{1}{k} \sum_{j=1}^{k} \delta_j$

- Step 2 (line 4-5): The selected resampler is applied to the fold, followed by the classifier, which is applied to the balanced result from the resampler. The geometric mean δ_j for j^{th} validation fold is then calculated (line 5).

The final value of the objective function, denoted as Δ, is an average geometric mean of k folds (line 6).

4.3 Second experiment: AutoML benchmark with up to six operators

The second experiment is based on the search space used in the well-known AutoML software, i.e., Auto-Sklearn [39], with up to 6 operators for classification problems on 73 AutoML benchmark datasets. In this section, we briefly introduce the datasets (Section 4.3.1) and the experimental procedure (Section 4.3.2). Finally, detailed information on the hyperparameters used is provided in Section A.3.2 of the Appendix.

4.3.1 Datasets

This experiment is based on 73 datasets from OpenML [204] as described in Figure 4.3. More precisely, all datasets from the AutoML benchmark [189] suite and all datasets from the OpenML100 [205], OpenML-CC18 [206] suites that require data preprocessing steps, for example, containing missing values, were used. A full list of datasets is provided in Section A.3.1 in the Appendix. Finally, categorical features of the selected datasets are transformed by one-hot encoding implemented in Scikit-Learn [151], and datasets are shuffled to remove the potential impacts of ordered data.

4.3.2 Implementation details

The overall structure of our AutoML experiment is summarized in Figure 4.4:

1. The process begins by downloading the corresponding dataset from OpenML [204], [207] of the OpenML #Task ID (input by user).

2. The necessary metadata is extracted from the input dataset to generate a suitable search space χ by the Auto-Sklearn search space generator. It is worth noting that this search space generator is based on two aspects: the machine learning problem, that is, binary classification, multiclass classification, multilabel classification, regression, multioutput regression, and data representation, that is, either dense or sparse representation. In practice, the generated search space for a single ML problem is large and commonly has up to 153 hyperparameters and six operators, i.e., categorical encoder, numerical transformer, imputation transformer, re-scaling, feature pre-processor, and learning operator.

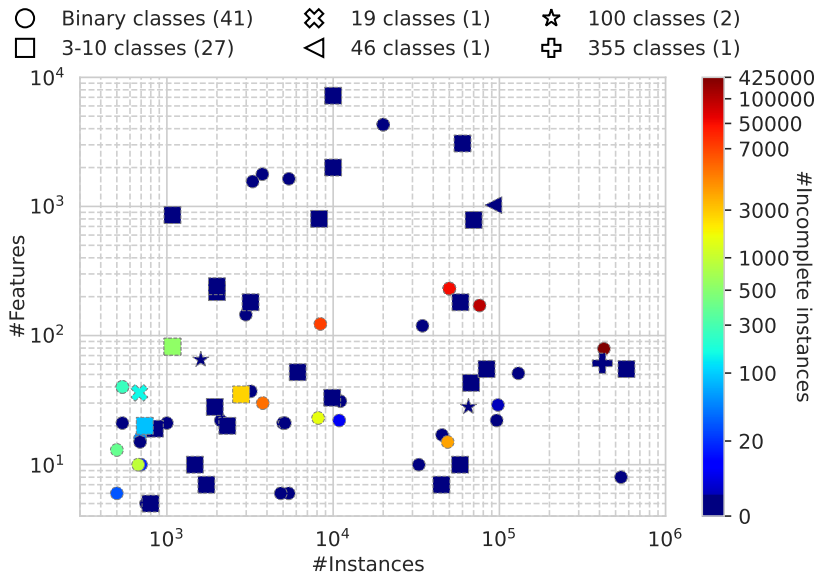


Figure 4.3: Overview of the characteristics of 73 AutoML benchmark datasets. The scatter plot shows the number of features (#Features) and instances (#Instances) on a logarithmic scale. The symbols indicate the number of classes and the color indicates the number of samples that contain missing values (#Incomplete instances).

3. The search space χ is converted to our search space $\mathcal{M}$ of the corresponding optimizer. Meanwhile, the input dataset is preprocessed and split into two independent sets $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{test}}$, with the original data preprocessing and train/test split techniques used in [22], i.e., 30% for testing and the remaining for training. Next, 4-fold cross-validation was applied to D_{train} to avoid overfitting. The later optimization phase takes k-folds and search space $\mathcal{M}$. For a fair comparison, the optimization time is only counted after this step.

4. The optimizer optimizes the given problem until the wall-time reaches 1 hour and returns the best-found pipeline setting p^*, consisting of a sequence of operators and their optimized hyperparameter settings.

5. Once the optimization process is done, the best-found pipeline setting p^* is used to initialize the corresponding machine learning model. Subsequently, it learns on $\mathcal{D}_{\text{train}}$ and predicts on $\mathcal{D}_{\text{test}}$. Lastly, the test performance measure on $\mathcal{D}_{\text{test}}$ was calculated.

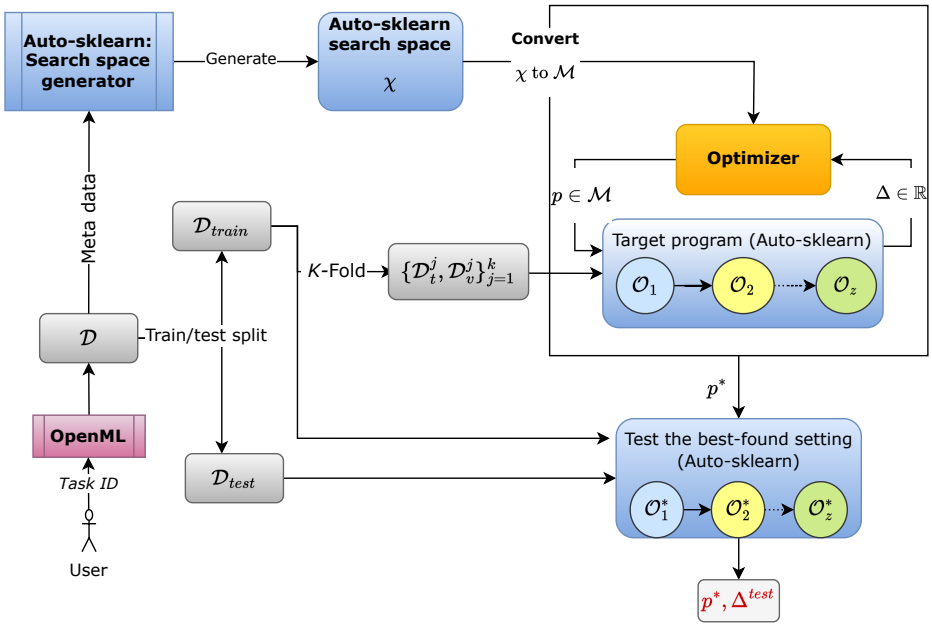


Figure 4.4: Flowchart of the second experimental setup.

4.3.3 Parameter setting

For a fair comparison, we used computational resources similar to those in [22]. For clarification, all experiments were conducted using our available computational clusters, namely *The Distributed ASCI Supercomputer 5* (DAS5) [208], where each computation node (32 cores) runs 4 experiments in parallel, that is, fixing 8 cores for one experiment. All experiments were repeated 10 times with different random seeds, limited by a soft limit of 1 hour[4] and a hard-limit of 1.25 hours[5]. The performance evaluation of a single configuration is limited to 10 min with 4-fold cross-validation on the training data, that is, the evaluation of a fold is allowed to take up to 150s. The evaluation of a configuration is aborted and returns zero if any fold has an error, for example, infeasible configuration and timeout.

[4]Soft-limit: the timeout' parameter set to the optimizer.
[5]Hard-limit: The optimization process will be manually aborted after 1.25 hours for any unexpected technical reasons. Thus, the configuration that achieves the highest performance is known as the best configuration for the run.

CHAPTER 5

An Empirical Investigation Comparing CASH Optimization Approaches for Class Imbalance Problems

This imbalanced classification problem is relevant to both academic and industrial applications. The task of finding the best machine-learning model to use for a specific imbalanced dataset is complicated because of the large number of existing algorithms, each with its own hyperparameters. In this chapter, we study ML pipeline optimization in detail in the class imbalance domain, where the best combination of resampling techniques and classification algorithms is searched for, together with their optimized hyperparameters. The Combined Algorithm Selection and Hyperparameter Optimization (CASH) has been introduced to solve ML pipeline optimization problem by converting the problem into a hyperparameter optimization problem. We experimented with the first experiment (see Section 4.2), i.e., a search space of 5 classification algorithms, 21 resampling approaches and 64 relevant hyperparameters. Moreover, we investigated the performance of two well-known optimization approaches: random search and the Tree Parzen Estimator approach, which is a type of Bayesian optimization. For comparison, we also performed a grid search for all combinations of resampling techniques and classification algorithms with their default hyperparameters. The remainder of this chapter is organized as follows. First, Section 5.1 shows the motivation and provides a brief introduction to our work. In Section 5.2, the relevant background knowledge on imbalance classification and hyperparameter optimization are provided, and Section 5.3 outlines the experimental setup. Experimental results are discussed in Section 5.4. Finally, the chapter is concluded, and further work is outlined in Section 5.5.

5.1 Introduction

The imbalanced classification problem has garnered increasing attention from both academic and industrial fields. Technically, any dataset with an unequal class distribution is imbalanced. However, only datasets with a significantly skewed distribution are traditionally regarded as imbalanced in the learning domain [209]. Academic researchers aim to propose novel algorithms to handle imbalanced classification problems in different scenarios, for example, resampling techniques and algorithm-level approaches, whereas industrial researchers focus on improving imbalanced classification performances for specific real-life applications, such as fault diagnosis or anomaly detection [210], [211].

The combination of resampling techniques and classification algorithms is the most commonly used approach for handling imbalanced data [74], [212]. This leads to a challenge for an imbalanced classification problem on how to choose the best model (i.e., a combination of a resampling method) and a classifier (the so-called model selection problem or algorithm selection problem [213]) and their optimized hyperparameters [74] to achieve the best performance. This is a case of ML pipeline optimization where two tasks have to be considered in this chapter: model selection (MS) and hyperparameter optimization (HPO). Typically, these tasks are addressed separately and sequentially [14], [46], where the practitioner can choose to handle either task first. Generally, practitioners proceed by tuning the hyperparameters for each modeling algorithm separately and then choosing the best model. However, this approach is considerably more expensive due to a high number of possible combination operations.

Alternatively, the practitioner can select a suitable model by training all models with their default hyperparameters or based on experience, and then tune the hyperparameters only for the best model. This approach might get stuck in a local optimum of the model that was initially chosen based on the default hyperparameter setting. On the other hand, instead of sequentially solving these problems, they can be combined into a single problem and solved simultaneously. This approach is commonly referred to as the Combined Algorithm Selection and Hyperparameter Optimization (CASH) [40] or Full Model Selection (FMS) [44] approach.

Approaches for tackling the CASH problem have been widely proposed in the machine learning domain, particularly in the context of automated machine learning (AutoML), such as, Auto-Weka [40], [41] and Auto-Sklearn [39], [41], TPOT [43], HyperOpt-Sklearn [42]. In addition, [14] demonstrated that

the CASH approach is competitive with the sequential approach and requires less computational effort. However, the CASH approach has not yet been studied in detail in the context of learning from unbalanced data.

Hence, in this study, we introduce CASH in the context of optimizing the machine learning pipeline of combined classification algorithms and resampling techniques for the class imbalance problem. We are particularly interested in studying which optimization approach for handling the CASH problem yields the best classification performance.

In the first experiment (see Section 4.2), we use two well-known optimization approaches – Random search and Bayesian optimization. Furthermore, we experiment with dropping the hyperparameter tuning and carrying out only the model selection (MS) part, as sometimes done by practitioners. Our results suggest the inferiority of such an approach and demonstrate that applying CASH optimization yields better performance, for all test cases considered. Moreover, we observe that the Bayesian optimization approach produces better results than Random search. Hence, we recommend using this approach for handling the CASH problem for the class-imbalanced classification problem.

5.2 Related Works

In this section, we first provide a brief introduction to imbalanced classification (Section 5.2.1) and the CASH problem (Section 5.2.2) studied in this chapter.

5.2.1 Imbalanced Classification

The main problem in imbalanced classification is that the number of samples of one class is much lower than that of other classes [209]. Herein, the one or more classes being underrepresented are called minority class(es) and the other class(es) are called majority classes.

It has been shown that both the data-level (resampling) approaches and algorithm-level approaches are efficient in handling class-imbalance problems [214]. The data-level approaches focus on producing balanced datasets based on the unbalanced original data, whereas the algorithmic-level approaches concentrate on adjusting classification algorithms to make them appropriate for the imbalanced datasets. In the imbalanced learning domain, resampling techniques can be further divided into three groups: under-resampling, over-resampling, and combine-resampling. Under-resampling balances the class distribution by removing majority

samples, for example, the TomekLinks [194], while over-resampling balances the class distribution via producing synthetic minority samples, e.g., SMOTE [17]. The combine-resampling integrates both removing the majority samples and creating synthetic minority samples in order to balance the class distribution, e.g., SMOTETomek [18].

Owing to recent developments in data storage and management, it has become possible for industry and engineering practitioners to collect a large amount of data in order to extract knowledge and acquire hidden insights. An application example may be illustrated in the field of computational design optimization [215], where product parameters are modified to generate digital prototypes and the performance is usually evaluated through numerical simulations which often require minutes to hours of computation time. Here, some parameter variations (minority number of designs) would result in effective and producible geometric shapes, but the given constraints are violated in the final step of optimization. In this case, applying proper imbalanced classification algorithms to the design parameters may save computation time.

The family of evolutionary under-resampling techniques (EUS) has proven to be powerful in handling instance reduction [216]. An EUS algorithm attempts to optimize the selected samples in the majority class by performing a binary search guided by an evolutionary algorithm [108], [110]. Results of the EUS and the most recent research studies in this family consist of EUS-Windowing (EUSW) [217], clustering-based surrogate model for EUS (EUSC) [218] and hybrid surrogate model for EUS (EUSHC) [187] are also compared with our approach in the followed section.

In the class imbalance domain, it is widely known that *accuracy* is a deceptive estimate of performance [74], [219]. Instead of *accuracy*, other metrics such as the area under the receiver operating characteristic (ROC) curve, F-measure, or geometric mean (GM) are commonly used to measure performance [220]. For comparison with previous studies [187], [218], we use GM as the performance evaluation metric (see Section 2.1.2.2).

5.2.2 The Combined Algorithm Selection and Hyperparameter Optimization (CASH) Approach

The Combined Algorithm Selection and Hyperparameter Optimization (CASH) [40] is a commonly used approach for solving the ML pipeline optimization problem by converting it into a hyperparameter optimization (HPO) problem. As we delve

into the ongoing discussion, it is essential to reference Chapter 1 (Section 1.1.2), where the CASH is extensively discussed. Throughout the ongoing discussion, we consistently employ the notations and problem definition introduced in that section for a comprehensive understanding. In the context of optimization, HPO is generally viewed as a black-box optimization problem, which aims at finding the global optimum λ^* of the hyperparameters, with respect to a real-valued loss function f. As a reminder, the CASH approach incorporates an additional hyperparameter $\boldsymbol{\lambda}^0$ to model the choice of algorithms for each operator.

As mentioned in Section 5.1, we use a combination of resampling and classification algorithms to handle the class-imbalanced problem. Hence, the search space includes a set of resampling techniques, a set of classification algorithms, and their hyperparameters. Let $\boldsymbol{\lambda}^0_{res} = \{\mathcal{A}^1_{res}, \ldots, \mathcal{A}^{n_r}_{res}, \varnothing\}$ and $\boldsymbol{\lambda}^0_{cls} = \{\mathcal{A}^1_{cls}, \ldots, \mathcal{A}^{n_c}_{cls}\}$ denote sets of possible choice of resampling and classification algorithms, correspondingly. In practice, the use of a resampling technique is optional, we hence add a choice of not using any resampler, i.e., represented by $\varnothing$. Let $\Lambda_{res} = \boldsymbol{\lambda}^0_{res} \cup \Lambda^1_{res} \cup \ldots \cup \Lambda^{n_r}_{res}$ and $\Lambda_{cls} = \boldsymbol{\lambda}^0_{cls} \cup \Lambda^1_{cls} \cup \ldots \cup \Lambda^{n_c}_{cls}$ represent the hyperparameter spaces of resampling and classification operators. Hence, the entire search space for this particular problem is denoted by Λ, which includes Λ_{res} and Λ_{cls}. The ML pipeline optimization problem becomes the HPO maximizing problem:

$$\lambda^* = \arg\max_{\lambda \in \Lambda} f(\lambda) \ , \tag{5.1}$$

Note that in practice, most HPO methods can handle the CASH problem by modeling the choice of algorithms as a categorical hyperparameter. Each algorithm is mapped to its locally dependent hyperparameters by the so-called conditional parameter (see hierarchical hyperparameter in Table 7.1).

The HPO algorithms chosen in this study include Grid Search (see Section 3.1.1), Random Search (see Section 3.1.2) and a Bayesian optimization variant, namely Tree Parzen Estimators approach (TPE) (see Section 3.1.3).

5.3 Experimental Setup

This study reports and discusses the first experiment that has been introduced in detail in Section 4.2 including the search space, datasets and ML algorithms. Therefore, we only provide some additional information.

Random search and Bayesian optimization algorithms implemented in the Python package HyperOpt[1] are used as HPO algorithms. Based on the initial experiments, we set the number of iterations of HPO to 500, after which the algorithms have shown no significant improvements.

Moreover, to study the effectiveness of the HPO algorithms, we evaluated all possible combinations of classification and resampling algorithms with their default hyperparameter settings, i.e., dropping hyperparameter tuning and carrying out only the model selection part, as sometimes done by practitioners. For each dataset, we reported the combination with the highest GM value. The considered search space includes 5 classification algorithms and 21 resampling techniques, resulting in $5 \times 21 = 105$ combinations. Evaluating these combinations individually is referred to as "Grid-Def" here (grid search HPO algorithm).

The experiment scripts for the reproducibility of the reported results are provided in a git-repository[2].

5.4 Results and discussion

In this section, we report the results and discuss our insights. The experimental results are summarized in Table 5.1 to illustrate the performance differences between the three integrated optimization approaches used, i.e., TPE, Random search (RS) and Grid-Def (Grid), and to compare them with the state-of-the-art Evolutionary under-resampling (EUS) methods [218]. In this table, our results are presented in the corresponding columns on the left side (not shaded) and the results from [218] are presented on the right side (grey shaded) for EUS, EUSW, EUSC and EUSHC. In both groups, the highest performance for each dataset is highlighted in bold font. In our experimental results, the methods that perform significantly worse than the best according to the Wilcoxon signed-rank test with $\alpha = 0.05$ are underlined. A value labeled with * indicates that our results outperform those of [218] for the corresponding dataset. Additionally, an extra column to the right summarizes the method that achieved the highest GM for the corresponding dataset.

The results allow the following insights:

- HPO approaches exhibit better performance compared to the Grid-Def approach which uses static default hyperparameters. Moreover, according

[1] `https://github.com/hyperopt/hyperopt` (version 0.2.5).
[2] `https://github.com/ECOLE-ITN/NguyenDSAA2021`

Table 5.1: Average geometric mean (rounded to 4 decimals) over 10 repetitions for the 44 datasets, ordered by increasing IR value.

Dataset	#IR	Our experimental results			Evolutionary algorithms [218]				Overall Winner
		TPE	RS	Grid	EUS	EUSW	EUSC	EUSHC	
glass1	1.82	*0.7989	0.7763	0.7793	0.7773	0.7010	0.7941	0.7367	TPE
ecoli-0_vs_1	1.86	*0.9864	*0.9864	*0.9864	0.9583	0.9312	0.9581	0.9615	TPE | RS | Grid
wisconsin	1.86	*0.9814	*0.9807	*0.9788	0.9690	0.9652	0.9600	0.9590	TPE
pima	1.87	*0.7711	*0.7651	*0.7599	0.6943	0.6749	0.6957	0.7145	TPE
iris0	2.00	1	1	1	1	1	1	1	-
glass0	2.06	*0.8749	*0.8588	*0.8719	0.8009	0.6176	0.8047	0.6595	TPE
yeast1	2.46	*0.7324	*0.7304	*0.7183	0.6533	0.6501	0.6600	0.6600	TPE
haberman	2.78	*0.7025	*0.6926	*0.6678	0.5475	0.5635	0.5521	0.5497	TPE
vehicle2	2.88	*0.9915	*0.9874	*0.9895	0.9259	0.9175	0.9265	0.9173	TPE
vehicle1	2.90	*0.8658	*0.8429	*0.8333	0.6729	0.6624	0.6512	0.6926	TPE
vehicle3	2.99	*0.8482	*0.8231	*0.8108	0.7280	0.7142	0.7165	0.7204	TPE
glass-0-1-2-3 _vs_4-5-6	3.20	0.9559	0.9505	0.9483	0.9525	0.9385	0.9647	0.9546	EUSC
vehicle0	3.25	*0.9863	*0.9809	*0.9766	0.9164	0.9027	0.9103	0.9016	TPE
ecoli1	3.36	*0.9047	*0.8966	*0.8999	0.8634	0.8306	0.8554	0.8424	TPE
new-thyroid1	5.14	*0.9969	*0.9966	*0.9944	0.9882	0.9809	0.9859	0.9653	TPE
new-thyroid2	5.14	*0.9978	*0.9966	*0.9910	0.9865	0.9773	0.9831	0.9746	TPE
ecoli2	5.46	*0.9361	*0.9337	*0.9361	0.9000	0.8663	0.9034	0.8772	TPE | Grid
segment0	6.02	*0.9993	*0.9990	*0.9965	0.9881	0.9870	0.9876	0.9858	TPE
glass6	6.38	*0.9524	*0.9516	*0.9381	0.8889	0.9071	0.9156	0.9054	TPE
yeast3	8.10	*0.9428	*0.9395	*0.9290	0.8728	0.8740	0.8752	0.8550	TPE
ecoli3	8.60	*0.9061	*0.9023	*0.9044	0.8348	0.8153	0.8500	0.8097	TPE
page-blocks0	8.79	*0.9456	*0.9422	*0.9401	0.9117	0.9038	0.9096	0.9085	TPE
yeast-2_vs_4	9.08	*0.9559	*0.9474	*0.9401	0.9042	0.8774	0.9156	0.8930	TPE
yeast-0-5-6 -7-9_vs_4	9.35	*0.8212	*0.8063	*0.7938	0.7685	0.7663	0.7901	0.7535	TPE
vowel0	9.98	0.9581	0.9483	0.9427	0.9897	0.9719	0.9877	0.9831	EUS
glass-0-1-6_vs_2	10.29	*0.8498	*0.8216	*0.7904	0.6383	0.6164	0.6651	0.5815	TPE
glass2	11.59	*0.8516	*0.8271	*0.7903	0.7194	0.6525	0.7262	0.6173	TPE
shuttle-c0-vs-c4	13.87	*1	*1	*1	0.9960	0.9968	0.9960	0.9960	TPE | RS | Grid
yeast-1_vs_7	14.30	*0.8028	*0.7926	*0.7979	0.7176	0.7079	0.7068	0.6669	TPE
glass4	15.46	*0.9323	*0.9244	*0.9318	0.8700	0.8513	0.8613	0.8531	TPE
ecoli4	15.80	*0.9727	0.9551	0.9415	0.8984	0.9362	0.8857	0.9645	TPE
page-blocks -1-3_vs_4	15.86	*0.9925	*0.9877	*0.9884	0.9674	0.9399	0.9471	0.9294	TPE
abalone9-18	16.40	*0.8889	*0.8752	*0.8536	0.7269	0.6772	0.7224	0.6559	TPE
glass-0-1-6_vs_5	19.44	*0.9567	*0.9530	0.9304	0.9214	0.9151	0.9160	0.9501	TPE
shuttle-c2-vs-c4	20.50	*1	*1	*1	0.9577	0.6449	0.9414	0.7365	TPE | RS | Grid
yeast-1-4 -5-8_vs_7	22.10	*0.7035	*0.6874	*0.6650	0.6569	0.6088	0.6604	0.6149	TPE
glass5	22.78	*0.9637	0.9555	0.9438	0.8105	0.9076	0.9600	0.9103	TPE
yeast-2_vs_8	23.10	*0.8231	*0.8031	*0.7945	0.7931	0.7496	0.7656	0.7668	TPE
yeast4	28.10	*0.8803	*0.8664	*0.8585	0.8050	0.7799	0.8288	0.7970	TPE
yeast-1-2 -8-9_vs_7	30.57	*0.7459	*0.7402	*0.7289	0.6721	0.6078	0.6704	0.6500	TPE
yeast5	32.73	*0.9803	*0.9790	*0.9788	0.9634	0.9494	0.9455	0.9653	TPE
ecoli-0-1 -3-7_vs_2-6	39.14	*0.9095	*0.8770	*0.9091	0.6700	0.7048	0.6625	0.6865	TPE
yeast6	41.40	*0.8972	*0.8905	*0.8840	0.8357	0.8080	0.8034	0.8031	TPE
abalone19	129.44	*0.7967	*0.7942	*0.7579	0.6258	0.6061	0.7214	0.6556	TPE

to the results of the Wilcoxon signed-rank test, TPE was always the best method: it significantly outperforms the Grid-Def in 32/44 datasets, whereas it significantly outperforms RS in 26/44 tested cases

- Overall, TPE shows the highest GM for most of the datasets, 41/44. Other compared methods win on different datasets, for example, EUSC and EUS achieve the highest GM on "glass-0-1-2-3_vs_4-5-6" and "vowel0", respectively. All approaches obtained the maximum GM on dataset "iris0".

- Furthermore, based on our experimental results, we conclude that TPE wins over the methods from [218] on 41/44 datasets, RS – on 38/44 datasets and Grid-Def – on 37/44 datasets. This is surprising because the number of function evaluations used in our experiment is much smaller than in [218]: 500 function evaluations for TPE and RS, 105 function evaluations for Grid-Def vs 10.000 function evaluations for each method in [218]. A possible explanation for this might be that [218] employed a simple KNN rule with $k = 1$ as the mere classifier, whereas more complicated classification algorithms were used in our study. More precisely, according to our experimental results, the tuned KNN wins only in 11% (TPE), 13% (RS), and 9% (Grid-Def) of all cases.

To investigate the tuning behavior of the methods, we plot single runs of TPE and RS on the dataset "abalone9-18" in Figure 5.1. The scatter plots on the left show the observed GM values over 500 function evaluations. The six stacked histogram plots to the right describe the distribution of the observed values, in which the first plot shows all observed values, and the five last plots indicate the distributions for each of the classification algorithms, such as SVM, RF, KNN, LR and DT. We conclude that:

- Configurations generated by TPE can avoid infeasible parameters better than RS[3]. In this run, the number of errors occurring in the TPE and RS runs are 14 and 22, respectively. Based on all datasets and repetitions, the number of infeasible configurations encountered by TPE and RS are 4.4% and 5.9%, respectively.

- Apart from zero values, the GM values of TPE are mostly in the range from 0.8 to 0.9, while the GM of RS are distributed in the range from 0.6 to 0.7.

[3]Evaluations with infeasible combinations of parameters are marked in the figure as black dashes with $GM = 0$.

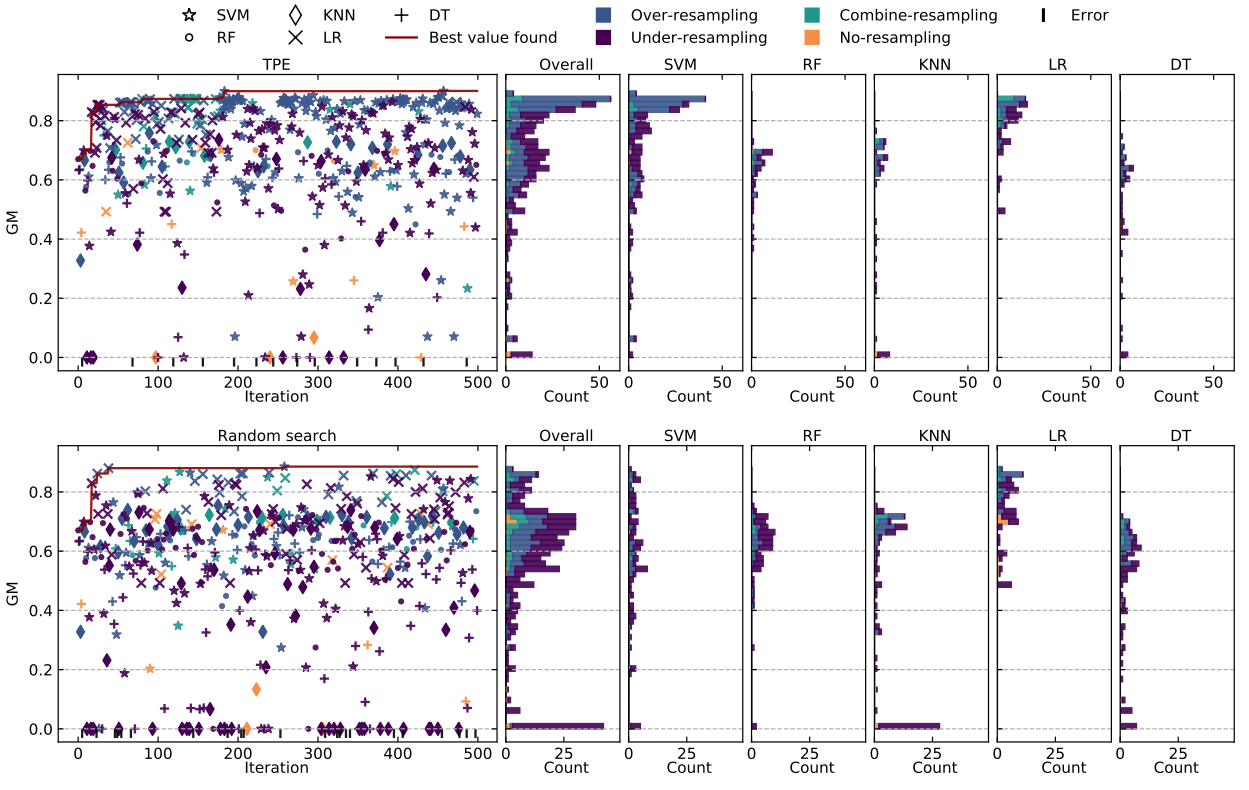


Figure 5.1: Illustration of the hyperparameter tuning process on dataset "abalone9-18" for TPE (top) and Random search (bottom). Scatter plots show the sequence of observed values of GM vs the number of function evaluations (iterations), the red line shows the current best value, and the black vertical bars on the horizontal axis indicate the infeasible configurations where GM returns to a zero if an invalid configuration is used. The stacked histogram plot next to the scatter plot shows the distribution of all observed values. The five last stacked histogram plots to the right indicate the distributions for each of the classification algorithms, namely SVM, RF, KNN, LR and DT.

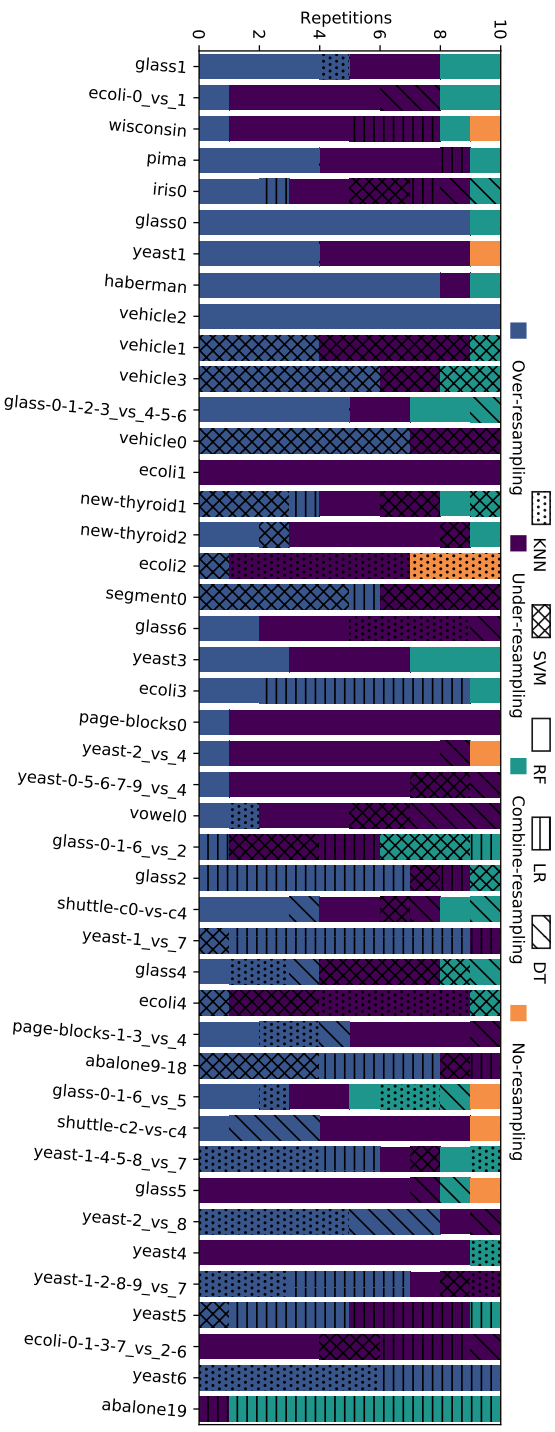


Figure 5.2: Resulting combination of classifier and resampler choices for an optimization process using TPE across 10 repetitions on 44 datasets.

Based on the highest results obtained by TPE, Figure 5.2 shows the final combination of choices of classification algorithms and resampling approaches once the optimization run is over. Clearly, no dominant algorithm exists over many datasets but different datasets benefit from different classification algorithms. For example, "glass0", "yeast1", "yeast3", "haberman", "vehicle2", "ecoli1" and "page-blocks0" achieve the best results with SVM, "vehicle0", "vehicle1", "vehicle3" with KNN, whereas "abalone19" always results in LR.

Besides, 98% of runs yield the best performance by using a resampling technique. Particularly, over-resampling, under-resampling and combine-resampling obtain 182, 199, 50 wins over $44 \times 10 = 440$ runs. Additionally, there is no classifier/resampler combination providing the best classification performance over all datasets. Specifically, RF and SVM obtain 206 and 84 wins, while other algorithms (LR, KNN, DT) find the best performance in 73, 48 and 29 runs.

5.5 Conclusions and Future Work

In this study, we applied a special type of Bayesian Optimization approach, the Tree Parzen Estimators to optimize the combined algorithm selection and hyperparameter optimization problem to improve the performance of classification algorithms for class imbalance problems. In other words, we propose an automated CASH optimization approach for imbalanced classification problems. Our approach automatically selects the best set of algorithms, i.e., the resampling technique and classification algorithm, together with their optimized hyperparameter settings for an arbitrary imbalanced dataset. The numeric results show significantly improved performance with respect to the state-of-the-art techniques in the imbalanced classification domain over 44 examined datasets.

Four main conclusions can be drawn from our experimental results:

1. Use of HPO clearly improves the classification performance compared to using static default parameters.

2. TPE outperforms the Random search on 91% of the tested datasets, while equal performance is found on the remaining cases.

3. Overall, the TPE approach produces the best results among other competitors in various scenarios. Hence, we recommend using TPE for handling CASH optimization in imbalanced classification problems.

4. Another finding was that 98% of runs yield the best performance with the help of resampling techniques. Thus we recommend to use resampling to handle class imbalanced problems.

There are several interesting research directions for extending this work. First, we intend to apply other Bayesian optimization variants such as SMAC, SPO, and MIPEGO, to study the performance of variants in this domain. Second, the scope of this study was limited in terms of classification problems; therefore, our future studies might extend the research for regression problems. Third, in addition to GM, other commonly used performance evaluation metrics in this domain will be investigated in our future work, including the Area Under the ROC Curve (AUC), F-measure, and recall. Fourth, the penalty-based methods, e.g., penalized-SVM, themselves can efficiently handle imbalanced datasets in several cases. Thus, we plan to study their effectiveness in the context of CASH optimization. Additionally, instead of applying hyperparameter tuning on the level of an individual dataset, we are interested in studying the behavior of HPO approaches when tuning for a set of datasets. Finally, besides Bayesian optimization, we shall extend our research with other state-of-the-art HPO approaches such as iRace [33] and Hyperband [35] for the class-imbalanced problem.

CHAPTER 6

On the use of AutoML optimization in real-world applications

Accurate classification of multiple classes is crucial in industrial applications, especially in identifying surface defects in the steel industry. Quality of surface of steel products is among the most significant contributors to their overall quality. Therefore, it is of vital importance to detect and classify various surface defects correctly. While established quality control measures implemented at various production stages successfully warrant against the high number of defects, they complicate further defect detection due to the high imbalance in the occurrence of defects vs defect-free cases. The situation is further complicated by a wide range of possible types of surface defects, with a heavily imbalanced distribution among them. In addition, setting appropriate hyperparameters of new classification methods to obtain a stable and accurate classification performance is far from straightforward given their strong interdependence. A hyperparameter optimizer is typically applied to identify the best Machine Learning (ML) model by evaluating its performance based on standard metrics such as accuracy rate, recall, precision, etc. However, some classes are more important in many real-world applications. Thus, to accommodate the latter, we propose an approach for penalizing existing classification performance metrics with a user-defined class importance matrix. We demonstrate the proposed approach on a highly imbalanced instance of multi-class classification of steel surface defects. We solve the Combined Algorithm Selection and hyperparameter optimization (CASH) problem to identify the best ML model. Such optimization is done by means of a competitive Bayesian optimization method in a search space of 21 resampling techniques and 5 classification algorithms (and their corresponding hyperparameter settings) for three commonly used multiple-class classification techniques (Multi-class direct classification, One vs. One and One vs. Rest). The results of our experiments show that the proposed approach

improves the performance significantly on the considered classification problem compared to the current classification system at TATA Steel (TATA).

The remainder of this chapter is organized as follows. The motivation, introduction and problem formulation are provided in Section 6.1. In Section 6.2, the relevant background knowledge on imbalance classification and hyperparameter optimization are provided, and Section 6.3 lays out the experimental setup. Additionally, experimental results are discussed in Section 6.3.3. Finally, the chapter is concluded in Section 6.4.

6.1 Introduction

The appearance of surface of a steel product is one of the significant quality aspects [49]. While established quality control measures already implemented at various production stages successfully warrant against the high number of defects in the resulting products, they complicate further defect detection due to the high imbalance in the occurrence of defects vs defect-free cases. The situation is further complicated by a wide range of different types of surface defects, with a heavily imbalanced distribution among these defect kinds. Additionally, setting appropriate hyperparameters of new classifiers to obtain a stable and accurate classification performance is far from straightforward given their strong interdependence. To maximize the classification performance, practitioners need to find a fine-tuned ML pipeline out of an extensive portfolio made up of a range of suitable algorithms with their complex hyperparameter settings. The practical surface defects classification problem faces two main challenges: (i) unequal/imbalanced distribution of defects across classes, (ii) unequal importance between classes (some imperfections are more severe than others).

The imbalanced classification problem can be solved by applying a well-performing combination of a resampling technique and a classification algorithm [74]. Finding such a well-performing combination of methods and the setting of their hyperparameters falls within the problem domain Combined Algorithm Selection and hyperparameter (CASH) optimization problems which can be solved effectively [47] via the Bayesian optimization [153].

Inside the optimization, the assessment method is critical in evaluating classification performance to choose the suitable classification model for the given problem. The performance metrics usually assume that all classes are equally important. However, users might want to stipulate preferences over classes. To

illustrate this, Figs. 6.1a, 6.1b show two classification outcomes that are indistinguishable from standard performance metrics. In practice, the practitioner will prefer the first outcome if `class 2` is more important than `class 1`. Therefore, the existing classification performance metrics are not able to evaluate and rank ML models for unequal class importance values. Consequently, the *automatic machine learning approaches* (such as Hyperparameter Optimization, CASH optimization, and AutoML optimization), which are mainly focused on selecting the ML model with the best predictive performance, *are not able to solve the problem efficiently in case of unequal class importance values.*

To solve that unequal importance classes problem, the assessment method has to be adjusted to reward correct predictions of important classes while penalizing incorrect predictions of those classes. Since almost all performance metrics are built on the confusion metric [221], we propose a novel approach that adjusts the confusion matrix by combining the confusion matrix with a user-defined penalty matrix (see Figure 6.1c), which contains different weights for predictions over classes. The general performance metric is then computed based on the penalized confusion matrix, potentially helping the optimizer solve the unequal importance classes efficiently.

Based on the formulation of the target objective function (Equation 1.2) given in Section 1.1, let $\alpha(\hat{y}_i, y_i)$ denote a penalty value of $\hat{y}_i$ and y_i, that can be extracted

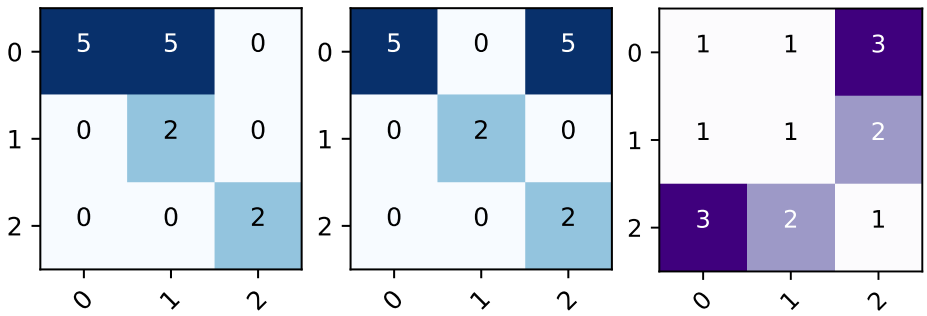


Figure 6.1: Examples of two confusion matrices with misclassifications in `class 0` (left and middle figures, where true and predicted labels are shown vertically and horizontally, respectively). A corresponding example of proposed penalty matrix (right figure) indicating that `class 2` is more important than `class 0` and `class 1`. Numbers in the penalty matrix indicate the misclassification severity weights per predicted-true label pair (e.g. a sample of `class 2` misclassified as `class 0` will be multiplied by 3).

from the pre-defined penalty matrix $\mathbf{P}_{\mathrm{M}\times\mathrm{M}}$ of M classes (see Figure 6.1c for illustration, for example, $\alpha(0,2)=3$). Adding $\alpha(\hat{y}_i, y_i)$ to the Equation 1.2, to punish wrong or reward correct prediction on class i. Then, the Equation 1.2 to compute the overall performance of the ML model p when trained on $\mathcal{D}_{\text{train}}$ of n samples and evaluated on $\mathcal{D}_{\text{valid}}$ of $(m-n)$ samples, can be adapted to:

$$f(p, \mathcal{D}_{\text{train}}, \mathcal{D}_{\text{valid}}, \mathbf{P}_{\mathrm{M}\times\mathrm{M}}) = \frac{1}{m-n}\sum_{j=1}^{m-n} R(\hat{y}_{n+j}, y_{n+j}, \alpha(\hat{y}_{n+j}, y_{n+j})) \tag{6.1}$$

where $\hat{y}$ is the predicted class, y is the true class and R indicates a measure metric.

Referring back to the discussion on CASH approach in Chapter 1 (Section 1.1.2) and search space for class imbalance problem in Chapter 5 (Section 5.2.2), the ML model p is structured as $p = \{(\mathcal{A}_{res}, \lambda_{res}, \mathcal{A}_{cls}, \lambda_{cls}) | \mathcal{A}_{res} \in \lambda^0_{res}, \mathcal{A}_{cls} \in \lambda^0_{cls}, \lambda_{res} \in \{\Lambda^1_{res}, \ldots, \Lambda^{n_r}_{res}\}, \lambda_{cls} \in \{\Lambda^1_{cls}, \ldots, \Lambda^{n_c}_{cls}\}\}$. Hence, the CASH optimization can be defined as:

$$p^* = \arg\max_{p\in\Lambda} f(p, \mathcal{D}_{\text{train}}, \mathcal{D}_{\text{valid}}, \mathbf{P}_{\mathrm{M}\times\mathrm{M}}) \,, \tag{6.2}$$

where $f(p, \mathcal{D}_{\text{train}}, \mathcal{D}_{\text{valid}}, \mathbf{P}_{\mathrm{M}\times\mathrm{M}})$ denotes the penalized performance accuracy of the ML pipeline p when trained on $\mathcal{D}_{\text{train}}$, evaluated on $\mathcal{D}_{\text{valid}}$, and penalized by the penalty matrix $\mathbf{P}_{\mathrm{M}\times\mathrm{M}}$.

6.2 Background

In this section, we review some background knowledge. We first provide a brief introduction of multi-class classification approaches (Section 6.2.1), the commonly used performance metric in the field of multi-classes imbalanced learning (Section 6.2.2) is presented.

6.2.1 Multi-Class Imbalance Learning

Most studies on the classification problem are devoted to the two-class classification scenario. However, a significant number of real-world applications contain more than two classes, for instance, image classification, protein classification, and medical diagnosis. The increasing number of classes poses new challenges for learning from multi-class imbalanced problems. First of all, more decision boundaries need to be defined during the multi-class classification process. Another challenging issue is that the imbalance among classes becomes more complicated as there will be multi-majority and multi-minority classes [222]. The data complexity, an

important cause of the degradation in binary case [220], is more sophisticated. Several solutions designed for imbalanced binary classification are extended to multi-class scenarios.

Class decomposition is an intuitive method to deal with multi-class classification problems [223]. After transforming the multi-class problem into multiple subsets, the existing approaches for handling the binary scenarios can be applied directly. Unlike the class decomposition approaches, *multi-class direct classification* [224] aims to solve the multi-class problem directly without reducing the problem to multiple binary classification tasks. This section first reviews two commonly used decomposition strategies: One vs. Rest (OvR) and One vs. One (OvO), see Sections 6.2.1.1and 6.2.1.2, respectively. Lastly, the multi-class direct classification method is given in Section 6.2.1.3.

6.2.1.1 One vs. Rest approach

Suppose there are M classes in the multi-class imbalanced problem. In the OvR decomposition, each of the M classes is trained against the remaining $(M-1)$ classes [225]. In other words, an M-class classification problem is decomposed into M binary classification problems. When predicting the final label for a test sample, each binary classifier provides a prediction with confidence, and the prediction with the *highest confidence* is usually determined as the final label for this test sample. While OvR provides the convenience of treating multi-class scenarios as binary scenarios, it also brings further imbalance into the binary subsets. In addition, all the individual classifiers are trained with the complete dataset, ensuring no information is dropped in the training procedure. However, this also preserves the overlapping regions, a factor leading to the degradation of the classification performance [220].

6.2.1.2 One vs. One approach

In the OvO decomposition, each of the M classes is trained against one of the remaining classes [226]. Thus, an M-class classification problem is decomposed into $M(M-1)/2$ binary problems, i.e. $M(M-1)/2$ classifiers will be built. The final predictions are usually determined via the *majority voting* among the $M(M-1)/2$ classifiers. Each binary classifier is only trained with pairs of classes; this makes the decision boundaries much simpler and properly addresses the overlapping issue. However, when pairing the classes, the number of binary classifiers increases quadratically in M [227]. The training time can be long if M is large.

6.2.1.3 Multi-class direct classification

The class decomposition methods are typically time-consuming as they transform the single multi-class problem into multiple binary problems, i.e., decomposing the input data into smaller parts or features. The *multi-class direct classification* (direct method) indicates the approach using a single classification algorithm to map input features to output (multi) classes directly, making it faster compared to the class decomposition approaches [228]. Hence, this approach only applies to the classification algorithms that can be modified, e.g., [229], [230] proposed to adjust the decision function in support vector machines, or are naturally designed to be applicable to multi-class problems. For examples, decision trees [71], [231], support vector machines [21], [232], k-nearest neighbors [233]–[235], logistic regression [236], and random forest [72], are suitable algorithms.

6.2.2 Performance Metrics

Table 6.1: Confusion matrix for a multi-class classification problem

		Predicted Class			
		A	B	...	M
True/Actual Class	A	$\mathbf{TP_A}$	$E_{A,B}$	...	$E_{A,M}$
	B	$E_{B,A}$	$\mathbf{TP_B}$	...	$E_{B,M}$
	...	...	...	**...**	...
	M	$E_{M,A}$	$E_{M,B}$	...	$\mathbf{TP_M}$

The assessment method is key in evaluating a classification performance to choose the right classification model for the given problem. In a classification problem, the confusion matrix is a common method to determine the performance of a classifier, as it can provide classification results. For instance, Table 6.1 shows the confusion matrix for a multi-class classification problem with M classes $(\mathrm{A}, \mathrm{B}, \ldots, \mathrm{M})$. As shown, $\mathrm{TP_A}$ is the number of True Positive (TP) samples in class A, and $\mathrm{E_{A,B}}$ is the number of samples from class A that were incorrectly predicted as class B. Hence, the number of False Negatives in class A ($\mathrm{FN_A}$) is the sum of $\mathrm{E_{A,B}}$ to $\mathrm{E_{A,M}}$, i.e., $\mathrm{FN_A} = \mathrm{E_{A,B}} + \ldots + \mathrm{E_{A,M}}$, which indicates the sum of all class A samples that were misclassified. Whereas the number of False Positives (FP) in class A is the sum of all samples that were misclassified as class A, i.e., $\mathrm{FP_A} = \mathrm{E_{B,A}} + \ldots + \mathrm{E_{M,A}}$.

Performance metrics for multi-class classification are usually decomposed into multiple single-class performance metrics by converting the confusion matrix in Table 6.1 into multiple 2×2 confusion matrices:
$\begin{bmatrix} \mathrm{TP_A} & \mathrm{FP_A} \\ \mathrm{FN_A} & \mathrm{TN_A} \end{bmatrix}, \ldots, \begin{bmatrix} \mathrm{TP_M} & \mathrm{FP_M} \\ \mathrm{FN_M} & \mathrm{TN_M} \end{bmatrix}$
The common per-class measurement metrics are presented in Table 2.1.

To compute an overall performance, the scores per class can be averaged to obtain a single score [227], [237]. There are three ways:

- **Macro** approach averages all per-class scores using the arithmetic mean of those values without considering the sample size difference between classes.
- **Weighted** approach is similar to the macro process but takes the sample size rate of classes, e.g., the sample size rate of class A is the number of samples of class A over the total number of samples.
- **Micro** approach computes the corresponding performance metrics by counting the sums of the True Positives (TP), False Negatives (FN), True Negatives (TN), and False Positives (FP).

In this chapter, we use the penalized geometric mean micro ($\mathrm{GM}^{\mathbf{P}}_{\mathrm{micro}}$) as the objective function to maximize, calculated as:

$$\begin{aligned} \mathrm{GM}^{\mathbf{P}}_{\mathrm{micro}} &= \sqrt{\mathrm{Specificity}^{\mathbf{P}}_{\mathrm{micro}} \times \mathrm{Sensitivity}^{\mathbf{P}}_{\mathrm{micro}}} \\ &= \sqrt{\frac{\sum_{i=1}^{\mathrm{M}} \mathrm{TN}^{\mathbf{P}}_i}{\sum_{i=1}^{\mathrm{M}} \mathrm{TN}^{\mathbf{P}}_i + \sum_{i=1}^{\mathrm{M}} \mathrm{FP}^{\mathbf{P}}_i} \times \frac{\sum_{i=1}^{\mathrm{M}} \mathrm{TP}^{\mathbf{P}}_i}{\sum_{i=1}^{\mathrm{M}} \mathrm{TP}^{\mathbf{P}}_i + \sum_{i=1}^{\mathrm{M}} \mathrm{FN}^{\mathbf{P}}_i}} \end{aligned} \tag{6.3}$$

where $\mathrm{TP}^{\mathbf{P}}_i, \mathrm{TN}^{\mathbf{P}}_i, \mathrm{FP}^{\mathbf{P}}_i, \mathrm{FN}^{\mathbf{P}}_i$ denote the number of penalized true positives, penalized true negatives, penalized false positives and penalized false negatives samples in class $i, i \in \mathrm{M}$ classes, respectively. Those values are based on the proposed penalized confusion matrix.

Based on the input (standard) confusion matrix $C_{\mathrm{M} \times \mathrm{M}}$ and a penalty matrix $\mathbf{P}_{\mathrm{M} \times \mathrm{M}}$ (defined by user). We take the Hadamard product, i.e., the pairwise product, of the two matrices, i.e., $(C'_{\mathrm{M} \times \mathrm{M}})_{ij} = (C_{\mathrm{M} \times \mathrm{M}})_{ij} \cdot (\mathbf{P}_{\mathrm{M} \times \mathrm{M}})_{ij}$, where C' denotes the penalized confusion matrix.

6.3 Experiments

In this section, we briefly introduce the dataset (Section 6.3.1) and the experimental procedure (Section 6.3.2).

6.3.1 Datasets

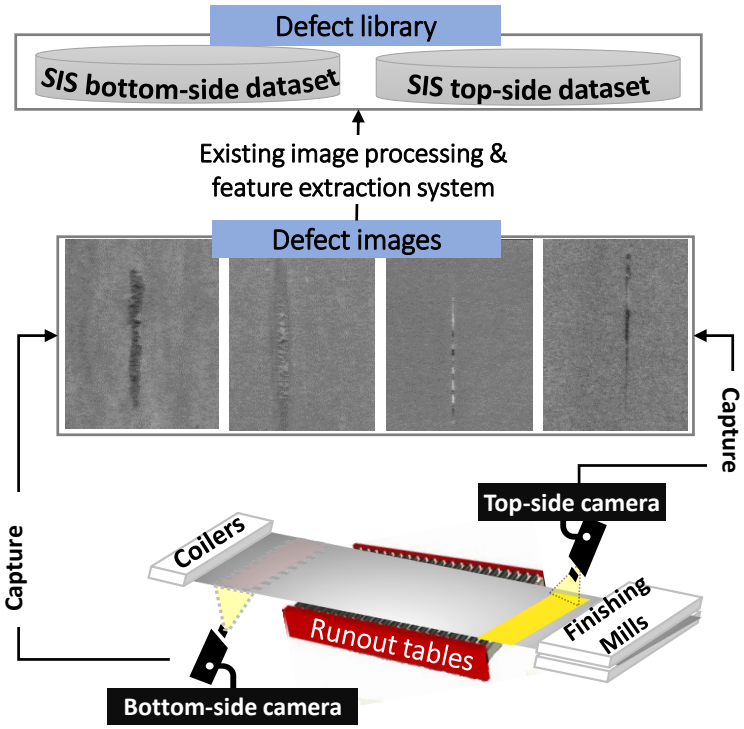


Figure 6.2: Schematic explanation of how the defect images are captured. Defect images are taken from TATA's official website[1], for illustration.

The appearance of the surface of a steel product is one of the major quality aspects. Therefore, surface defects should be avoided or at least known. A camera-based Surface Inspection System (SIS) is used in various process lines to identify those defects in the industry [238]. Grey value images taken from the surface by the SIS contains information on the defects. These images of various defects occurring in production are assessed and gathered in defined classes within a so-called *defect library*. Figure 6.2 shows a diagram, illustrating how the defect images are captured in the production process. The defect library is used to train and test classifiers, and these classifiers are finally used to identify the new surface defects from production. Thus, stable, accurate, and high classification

[1] https://automation.tatasteel.com/products/rolling-mills/squins-surface-quality-inspection-system/

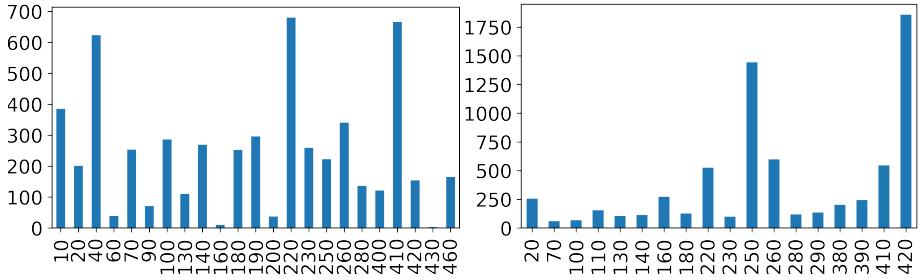


Figure 6.3: The distribution of samples over classes for the top-side (left) and bottom-side (right) camera datasets. The top-side dataset contains 23 classes and 5578 samples, while the bottom-side dataset contains 18 classes and 6908 samples (158 attributes in both cases).

performance is a must in the quality control procedure. However, the imbalance in the number of various defect types makes it challenging to obtain a stable and accurate classification performance.

The images captured by the SIS cameras are processed in the feature extraction module. Then, relevant defect features, e.g., geometrical, textural, and moment features, are extracted for classification. Both the images and information after extraction are stored in the defect library. The surface defects dataset used in this chapter is taken from a defect library after a specific selection was made (for confidentiality reasons). The library is split into two datasets with 158 features: the *top-side* camera and the *bottom-side* camera dataset. The top-side dataset contains 5578 samples distributed in 23 classes. The bottom-side dataset contains 6908 samples distributed in 18 classes. The distribution of the classes on surface defects data used for the experiments is given in Figure 6.3.

6.3.2 Experimental procedure

In this chapter, we experiment with two datasets (top- and bottom-side, see Section 6.3.1) with three multi-class classification strategies, i.e., One vs. Rest (OvR), One vs. One (OvO), and direct method (see Section 6.2.1). TPE as implemented in the `Python` package HyperOpt[2] (version 0.2.5) is used as the mere CASH optimization algorithm with a budget of 500 function evaluations. We reuse the search space identical to Section 4.2, with 5 classification algorithms (Support Vector Machines (SVM), Random Forest (RF), k-Nearest Neighbors

[2] `https://github.com/hyperopt/hyperopt`

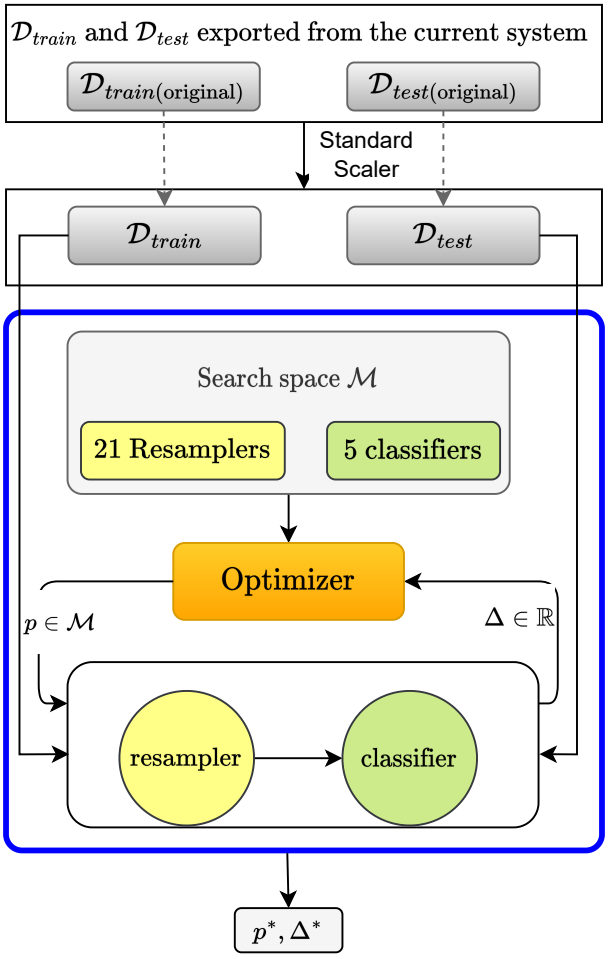


Figure 6.4: Flowchart of the experimental setup.

(KNN), Decision Tree (DT) and Logistic Regression (LR)) and 21 choices of resampling techniques.

In this study, we set up three independent experiments, each representing a different approach mentioned in Section 6.2.1, i.e., One vs. Rest, One vs. One and Direct method. Our experiments aim to compare the current classification system (*current system* in figures and tables below) used by TATA[3]. We use the same training and test datasets as the *current system* for a fair comparison. The

[3]For reasons of confidentiality, since proprietary software of a supplier is used by the industrial partner, no details about the algorithmic approach taken by the currently used system are available.

Table 6.2: Average penalized geometric mean (micro), rounded to 5 decimals over 10 repetitions for the 2 datasets. Boldface highlights the best-performing method per dataset and underline indicates results that are significantly different from the best method in that group according to a Wilcoxon signed-rank test ($p < 0.05$).

Dataset	Direct method	OvO	OvR	Current system
Top side	0.88293	0.88475	**0.88729**	0.81308
Bottom side	0.90990	**0.91275**	0.91245	0.79811

current system executes 10 times on each of the tested datasets. For each execution, the considered dataset is randomly split into training (80%) and test (20%) sets. The prediction performances are reported in Section 6.3.3, and the train/test sets are exported to use in our experiments. i.e., we have $2 \times 10 = 20$ different train/test sets in total. The overall structure of our implementation is summarized in Figure 6.4. The experimental process begins with a data normalization step by applying the so-called Standard Scaler[4] function to the input dataset, i.e., resulting in zero mean and a standard deviation of one. Then, the training and test datasets are fed into the optimization phase.

During the optimization process, the training dataset is used for the ML pipeline proposed by the optimizer. The ML pipeline is then measured by evaluating its prediction performance on the test dataset. We note that the performance is computed based on the penalized confusion matrix that is recomputed by multiplying values in the standard confusion matrix with the corresponding values in the penalty matrix, which is defined by TATA's domain experts (see Figure 6.5). The final evaluation value is calculated by computing the geometric mean (micro) on that penalized confusion matrix. Lastly, the reported result of each method for an individual dataset is averaged over 10 executions.

6.3.3 Results

In this section, we report the results and discuss our insights. The experimental results are summarized in Table 6.2 to illustrate the performance differences between the three classification strategies used: Direct method and two decomposition approaches, i.e., One vs. One (OvO) and One vs. Rest (OvR), and to compare them against the classification approach used in the *current system*. The highest performance for each dataset is highlighted in bold. The methods performing

[4]Standard scaler is implemented in the python library scikit-learn (version 0.23.2).

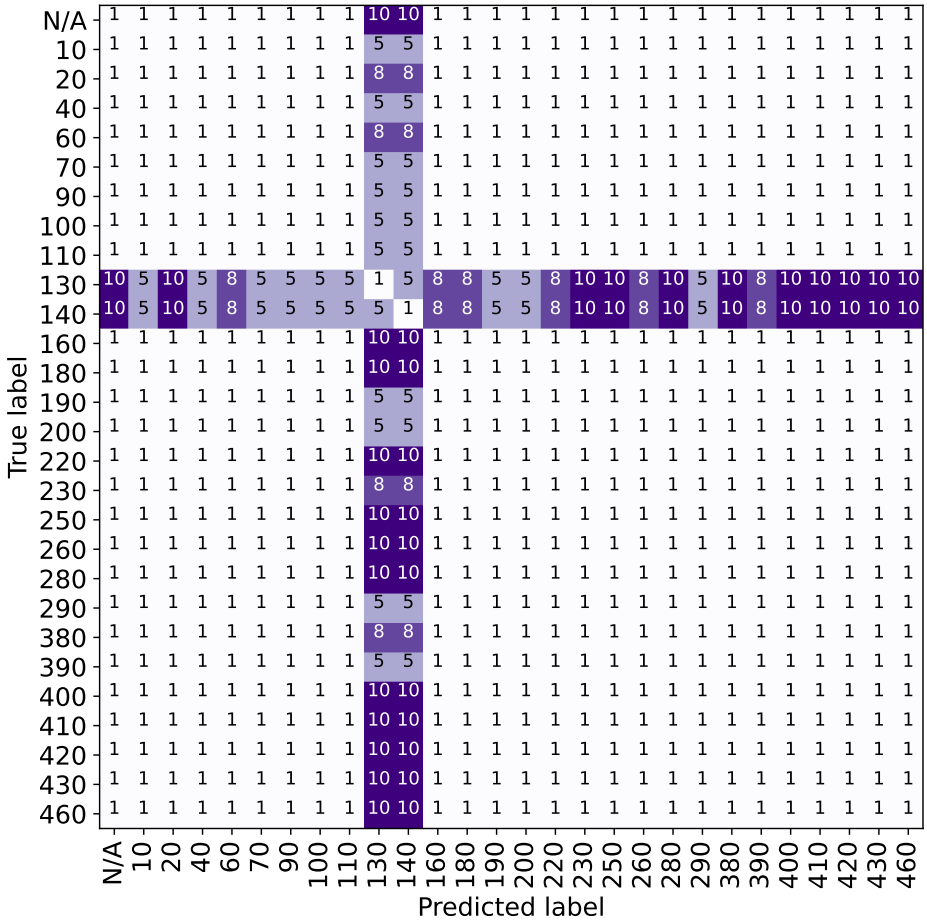


Figure 6.5: Penalty matrix used in our experiments

significantly worse than the best according to the Wilcoxon signed-rank test with $\alpha = 0.05$ are underlined.

Additionally, the distribution of the used performance metric, i.e., the penalized geometric mean (micro), over 10 repetitions for the two tested datasets, is visualized in Figure 6.6. Each box plot represents 10 repetitions. The horizon inner line shows the median. The whiskers show the lowest and the highest observed value[5]. The color dots show the observed values, and the dots outside the whisker represent the outliers. The box covers the first to the third quantiles.

The results allow the following insights:

[5]whisker scale is set as 1.5.

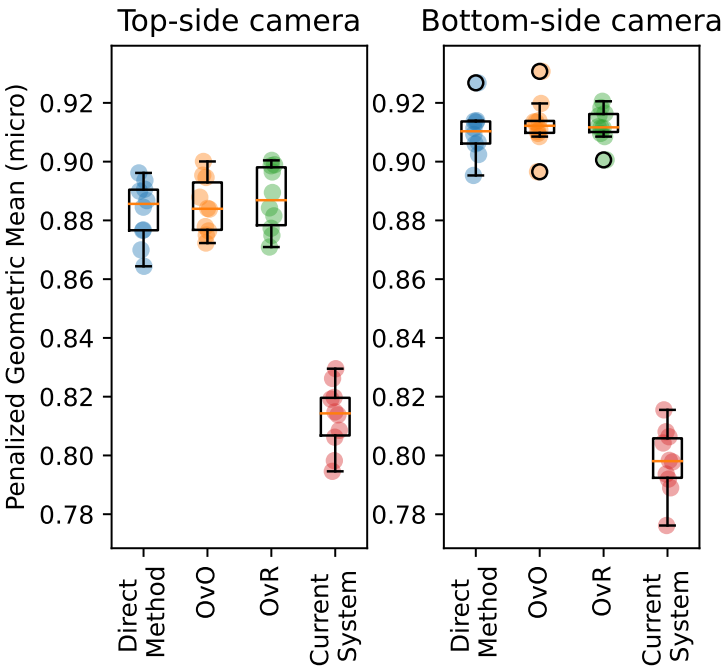


Figure 6.6: Box plots showing the distribution of classification results over 10 repetitions for two examined datasets.

- According to the results of the Wilcoxon signed-rank test, our experimental approaches significantly outperform the current approach used at TATA (*current system*). Additionally, from Figure 6.6, the median and whiskers of our three approaches are higher than those of the *current system*.

- Overall, the decomposition approaches produce the highest performance for both tested cases. More precisely, OvO shows the highest result on the "Bottom side camera" dataset, while OvR achieves the highest result on the "Top side camera" dataset.

- Additionally, according to our experimental results, *Direct method* does not outperform decomposition approaches but is not significantly worse than any decomposition approaches over all tested cases.

As mentioned in Section 6.2, the decomposition approaches are more expensive than the direct classification approach. To investigate this in more detail, we provide Figure 6.7 to show the running time of 10 executions for the 3 experimental approaches. The colour box shows the running time for 1 execution of 500 function

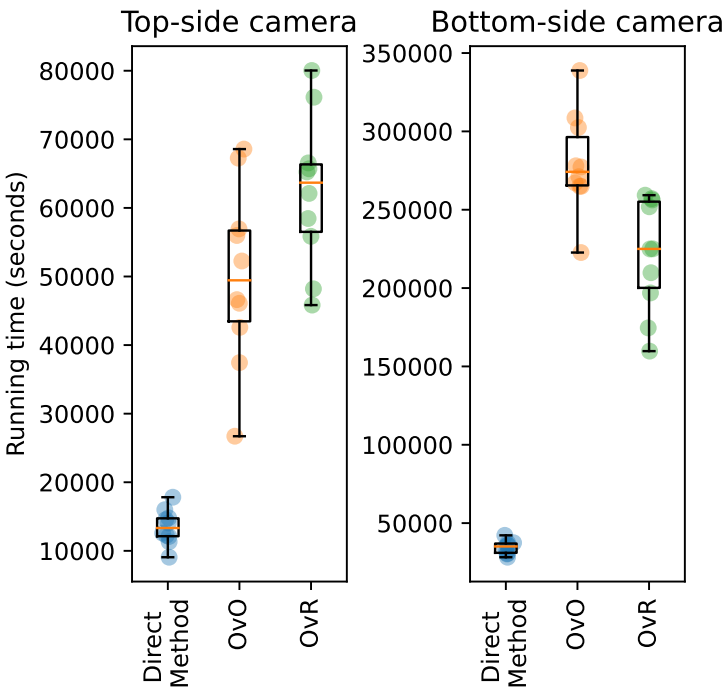
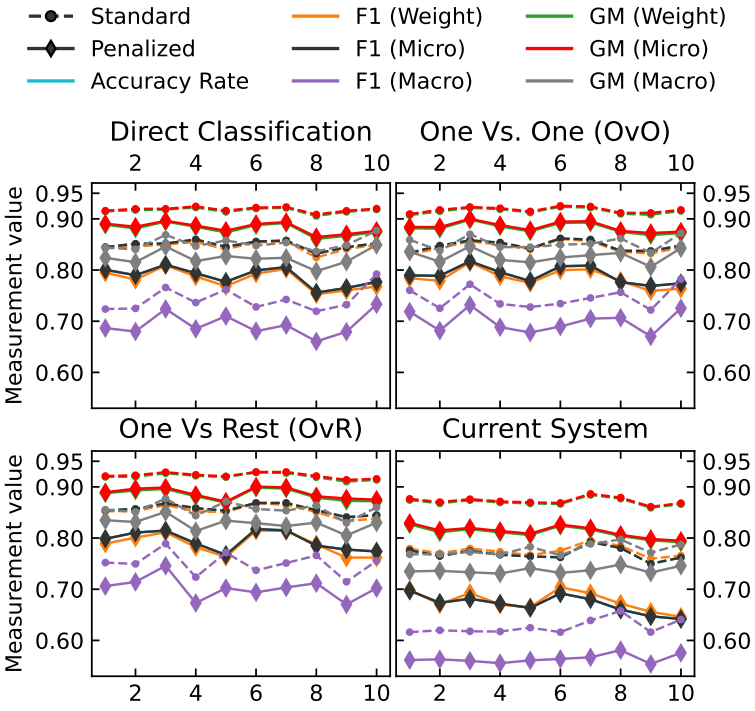


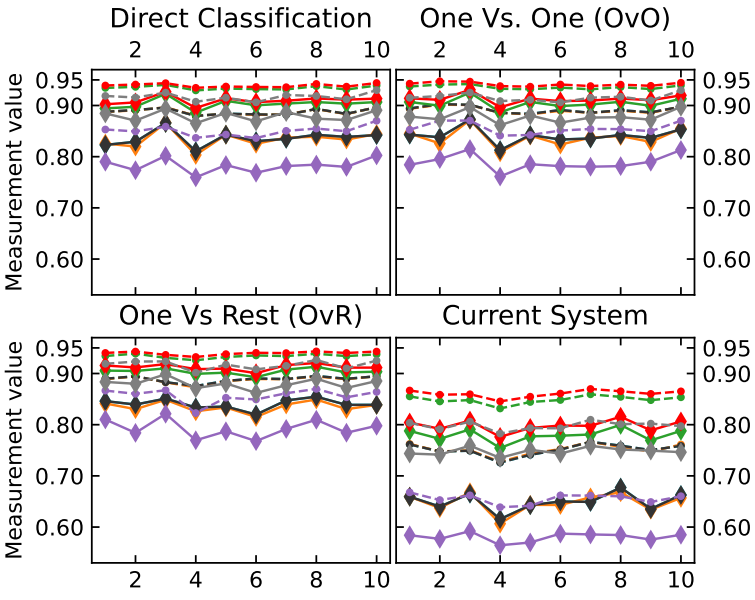
Figure 6.7: Running time over 500 function evaluations over 10 repetitions for two examined datasets.

evaluations. The box covers the first to the third quantiles. The horizon inner line shows the median. The whiskers show the fastest and the slowest execution. We can observe that the *direct method* is the fastest of the three experimental approaches. Notably, the average running time of the OvO and OvR on the "Top side camera" dataset is slower than the *direct classification* approach, approx 372% and 464%, respectively. In the same computation way on the "Bottom side camera", they are 811% and 643%. This is consistent with our presupposition, because the two decomposition approaches, i.e., OvO and OvR, convert the original multi-class dataset into multiple binary-class datasets, resulting in increased resource consumption for each iteration.

Figure 6.8 shows the results of 7 measurement metrics (i.e., F1 (weight), F1 (Micro), F1 (Macro), GM (Weight), GM (Macro), GM (Micro) and Accuracy rate) with and without penalization, by re-evaluating these metrics on the best-found ML pipelines once optimization processes are over. The results are shown for each of the ten runs performed. The dashed line with the dots marker shows the



(a) SIS top-side camera.



(b) SIS bottom-side camera

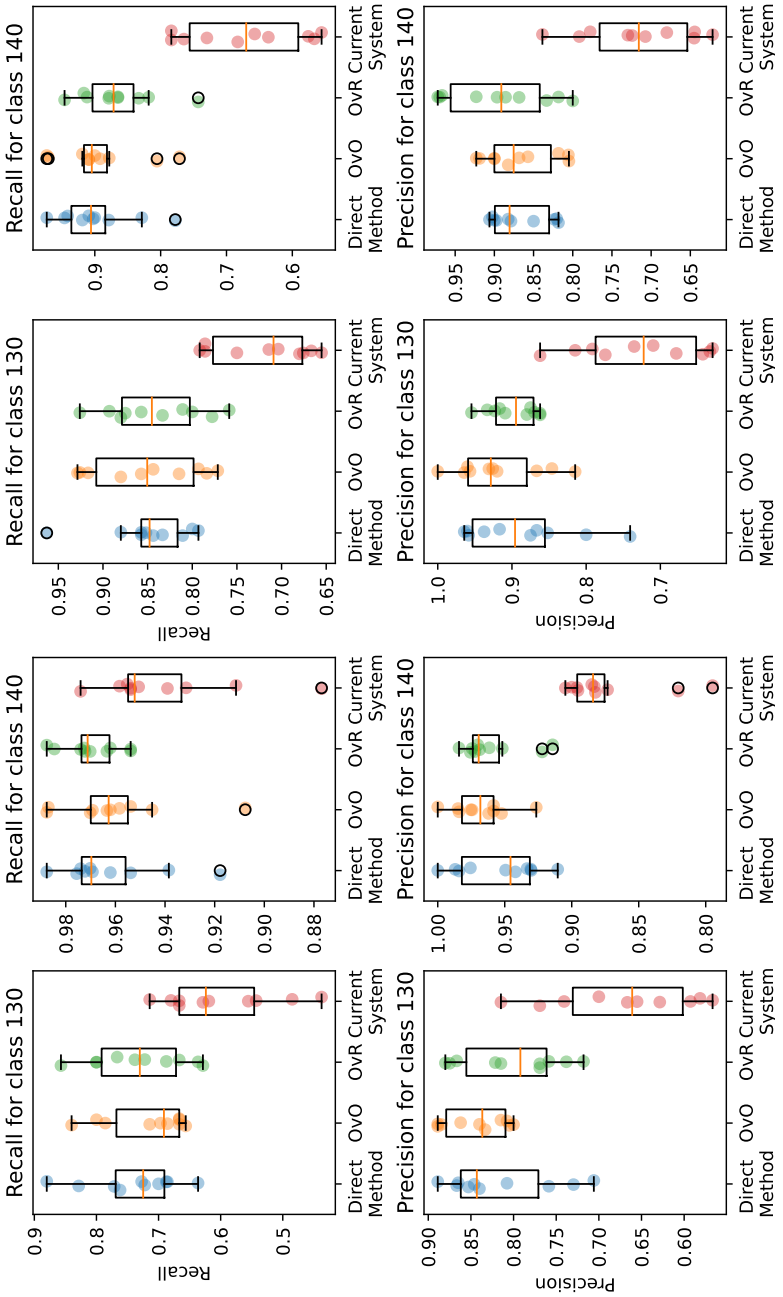
Figure 6.8: Comparisons between standard vs. penalized version of 7 measurement metrics on two datasets for three proposed vs. current approaches.

value of those standard metrics, i.e., without penalization. The solid line with the diamond marker shows those measurement metrics with our penalization approach. Penalized values are always lower than the corresponding standard values. A possible explanation is that the ML model misclassifies samples of the important classes. Lastly, we can observe that the penalized and the corresponding values of the used 7 measurement metrics are strongly correlated.

To investigate the predictive performance for the important classes (i.e., class 130 and 140), we provide Figure 6.9 to show recall and precision for those two important classes of 10 executions for the two tested datasets (i.e., Figure 6.9a for the top-side camera dataset and Figure 6.9b for the bottom-side camera dataset). Each box plot represents 10 repetitions. The box covers the first to the third quantiles. The horizontal inner line shows the median. The whiskers show the highest and the lowest values. The colour dots show the observed values, and the dots outside the whisker denotes the outliers. We can observe that the three CASH approaches produce better precision and recall scores for the two important classes on the two examined datasets.

6.4 Conclusion

In this chapter, we proposed an efficient approach to solving the steel surface defect classification, where the defect classes are (1) imbalanced and have (2) unequal importance. Firstly, we applied Bayesian Optimization (BO) to optimize the Combined Algorithm Selection and Hyperparameter Optimization (CASH) problem (i.e., the combination of resampling and classification algorithms, with their hyperparameter setting), to improve the classification performance for this class imbalance problem. Second, we propose a novel penalization approach to compute the classification performance metrics for unequal importance of classes. Based on our experimental results (Figure 6.6) and the running time analysis (Figure 6.7), we observed that the CASH approach clearly improves classification performance compared to the current classification system in use by TATA. Additionally, the direct classification method is much cheaper than the two experimented decomposition approaches (i.e., One vs. One and One vs. Rest) and not significantly worse than any of those in both test cases. Hence, we recommend to use the direct classification method to deal with similar problems. Lastly, the penalized performance metrics are strongly correlated to the standard metrics and



(a) SIS top-side camera.

(b) SIS bottom-side camera.

Figure 6.9: Recall and Precision for important classes over 10 repetitions per examined method.

efficient in measuring the important misclassified cases. Finally, future work will apply the proposed penalty approach to other industries.

CHAPTER 7

Efficient AutoML via Combinational Sampling

In the previous chapter, the CASH approach converted the ML pipeline optimization problem into a hyperparameter optimization (HPO) problem, where the choice of algorithms was modelled as an additional categorical hyperparameter. In this manner, algorithms and their local hyperparameters are referred to at the same level. Consequently, this approach renders the resulting initial sampling less robust. Unlike the CASH approach, in this study, we used a new hyperparameter class to model the choice of the algorithm under the operator. Additionally, we propose a novel initial sampling approach to maximize the coverage of the AutoML search space to help BO construct a robust surrogate model. We experimented with both experimental scenarios of AutoML with two operators and six operators over 117 benchmark datasets, as introduced in Section 4. The results of our experiments demonstrate that the performance of BO is significantly improved using our sampling approach.

The remainder of this chapter is organized as follows. First, the motivation and introduction are provided in Section 7.1. Next, our contributions are highlighted in Section 7.2, Section 7.3 lays out the experimental setup. The experimental results are discussed in Section 7.4. Finally, the chapter is concluded, and further work is outlined in Section 7.5.

7.1 Introduction

Recall that existing AutoML approaches (e.g., [39], [40]) can be considered as optimization processes for which the best ML pipeline is searched. Each pipeline includes an architecture and a set of hyperparameter settings.

Bayesian Optimization (BO) is a commonly used approach in AutoML as it has been successfully used in hyperparameter optimization (HPO) problems and plays a role of an optimizer in many AutoML frameworks, e.g., Auto-Sklearn [39],

Auto-Weka [40], and Hyperopt-sklearn (HPsklearn) [42]. BO is an efficient global optimization approach (in terms of the number of function evaluations) in which the trade-off between local exploitation and global exploration is well handled. Therefore, in this work, we focus on improving the BO by using it to solve the AutoML optimization problem. Traditionally, the AutoML optimization problem is treated as a HPO process, where the optimizer is inherited from the HPO domain. As HPO was originally developed to find the best hyperparameter setting from a single algorithm, it naturally does not consider the choice of algorithm. The choice of algorithm is then modelled as an extra categorical hyperparameter. Consequently, this HPO-based approach in handling the choice of algorithm mismatches the nature of the AutoML optimization problem.

The search space in the AutoML approach is largely owing to the many possible algorithm choices for pipeline operators. However, including many algorithms in the search space naturally leads BO to slow convergence or to get stuck in a local optimum [30], [31], [35]. One reason is that the initial sampling step in AutoML is typically restricted to a small budget, which is much smaller than the number of possible pipelines that can be constructed in the search space. The reason for this setting is that the effectiveness of BO becomes evident mainly in the later stages of optimization when it learns to produce better configurations. Many well-known sampling approaches, for example, the discrepancy-based quasi-random (quasi-random) sampling [30], the Latin Hypercube (LHD) sampling [239], have been employed for the initial sampling in this optimization context. However, they have shown themselves to be insufficiently robust [31], [240], [241] because they have been used in conjunction with the traditional approach of solving an AutoML optimization problem, which, as explained above, consists of converting it to a HPO, thus rendering obscure the differences between the choice of an algorithm and the choice of the algorithm's parameters.

Additionally, to construct a robust surrogate model, BO requires good coverage of the search space [31], but as the number of algorithms increases, the number of samples required to cover the search space increases exponentially. Previous studies [46], [47] pointed out that some algorithms can be grouped based on their technical behaviors.

To assess this theory, in this chapter, we propose a new two-fold approach to improve BO used in AutoML optimization:

- Group the similar operator algorithms when allocating initial sampling budget, e.g., the grouping of linear classifiers vs. the grouping of rule-based

Table 7.1: Hyperparameter types and functions used in our implementation

Hyperparameter	**Annotation**	**Description**
Continuous	FloatParam(min, max)	Choose a float value in range of $[min, max] \cap \mathbb{R}$
Ordinal	IntegerParam(min, max)	Choose a integer value in range of $[min, max] \cap \mathbb{Z}$
Nominal	CategoricalParam$(C_1, \ldots, C_n)$	Choose a value in set $\{C_1, \ldots, C_n\}$
*Algorithm	AlgorithmChoice$(A_1, \ldots, A_n)$	Choose a value in set $\{A_1, \ldots, A_n\}$
Hierarchical	ConditionalParam$(Parent, \{P_{value}\}, \{Child_1, \ldots, Child_n\})$	when a HyperParam has children
Infeasible	ForbiddenParam$((Param_1, \{P^1_{value(s)}\}), (Param_2, \{P^2_{value(s)}\}))$	when the combination of $Param_1$ and $Param_2$ is forbidden
* Grouping	HyperParam$(\underbrace{\{\text{value}^1_1, \ldots, \text{value}^1_{v_1}\}}_{\text{group}_1}, \ldots, \underbrace{\{\text{value}^n_1, \ldots, \text{value}^n_{v_n}\}}_{\text{group}_n})$	each group$_i$ can be of any type: $\{Continuous, Ordinal, Nominal, Algorithm\}$

classifiers [46]. Table 7.1 summarises different hyperparameter classes with their semantics in our work.

- Building on top of other sampling approaches, we propose a novel sampling method that aims to allocate reasonable budgets for each set of algorithms to maximize the coverage of sampling areas in terms of the grouping of algorithms to provide a robust surrogate model. In other words, our proposed approach is complementary to other sampling approaches, rather than competitive, with the aim of optimizing the performance of the search space of AutoML.

7.2 The Proposed Approaches for Automated Machine Learning

In this section, we first introduce our proposed combination-based sampling approach for increasing the efficiency and robustness of AutoML. Next, we introduce a new BO Python library for AutoML optimization and an AutoML framework that implements this paradigm.

7.2.1 Novel combination-based initial sampling for Bayesian optimization for AutoML optimization

The central idea of our approach is to provide optimized coverage of the algorithm-hyperparameter search space already during the initial sampling of BO in order to characterize the response surface more accurately.

To properly analyze this discussion, we need to utilize the notations that were introduced in Chapter 1 (Section 1.1.1). These notations are crucial for our ongoing analysis and were discussed in detail in their original context in Chapter 1 to ensure a better understanding. Given a search space denoted by $\mathcal{M}$ includes the sequence of z operators $\mathbb{O} = \mathcal{O}_1 \times \ldots \times \mathcal{O}_z$ and its corresponding hyperparameter spaces $\Lambda = \Lambda_{\mathcal{O}_1} \cup \ldots \cup \Lambda_{\mathcal{O}_z}$, as defined in Section 1.1.1 of Chapter 1.

A grouping of algorithms of operator $\mathcal{O}_i$ assumes that the set of all algorithms $\{\varnothing, \mathcal{A}_i^1, \ldots, \mathcal{A}_i^{n_i}\}$ available to be employed for operator[1] $\mathcal{O}_i$ can be partitioned into g_i non-empty and non-overlapping subsets, according to their inner workings[2]: $\{G_i^1, \ldots, G_i^{g_i}\}$, $g_i \leq n_i + 1$[3]. Such partitioning is called a *grouping of algorithms*.

[1] if $i < z$ or $\{\mathcal{A}_z^1, \ldots, \mathcal{A}_z^{n_z}\}$ if $i = z$.
[2] or any other user-defined logic.
[3] if $i < z$ and $g_z < n_z$ otherwise.

The operator can then be represented as $\mathcal{O}_i = \{G_i^1, \ldots, G_i^{g_i}\}$. According to our proposed combination-based initial sampling method (see Algorithm 9), the sequence of pipeline operators $\mathbb{O} = \mathcal{O}_1 \times \ldots \times \mathcal{O}_z$ should be sampled in BO from the domain space of sets $\{G_1^1, \ldots, G_1^{n_1}\} \times \ldots \times \{G_z^1, \ldots G_z^{n_z}\}$ and the total initial sampling budget should be split equally per group. The main idea behind such sampling budget reallocation is the potential exploitation of similarities between algorithms within the group: sampling fewer of the same algorithms frees up the budget to be distributed to other (different) algorithms, thus improving the coverage of algorithm-hyperparameter search space at an earlier stage of BO.

As an input parameter for our method, we require a number of data points B_{init} for the initial sampling and a maximum number of combinations K, $K \leq B_{init}$. If K exceeds the maximum number of possible combinations computed from the input operation steps $k = \prod_{i=1}^{z} |\mathcal{O}_i|$, then we use Algorithm 10 to randomly regroup algorithms in operators to ensure $k \leq K$. The proposed sampling algorithm, presented in Algorithm 9, consists of the three following steps:

1. Generate the list of combinations: List all k possible combinations of groups for all z operators; apply RandomRegrouping until k is small enough ($k \leq K$) (lines $2-4$).

2. Allocate budget to combinations: first allocate budget to all combinations based on the number of algorithms and hyperparameters behind (lines 5 - 10). Then, if there is any remaining budget B_{remain}, randomly allocate B_{remain} to the top $\frac{k}{\eta}$ combinations ordered by their size (i.e. the number of algorithms and hyperparameters in the combination). We take the size of the combinations into account to give larger combinations a higher chance of getting a larger budget.

3. Sampling configurations: for each combination s, an existing sampling approach (e.g., LHD, quasi-random, here we use quasi-random) is used to generate a trial sequence $s_j = (G_1, \ldots, G_z)$ (lines $12-16$); Lastly, the generated configurations must be verified by CheckForbidden[4].

Lastly, the generated configurations are shuffled to remove a potential impact of grouped configurations based on combinations. This is highly recommended

[4] An external function that verifies a combination of algorithms/a configuration with the forbidden rules defined by the user.

Algorithm 9: Combination-based sampling

Input: $\mathbb{O}$: sequence of operators, Λ: hyperparameter spaces, B_{init}: number of initial samples, K: maximum number of combinations of grouping of algorithms over operators, $\eta = 2$: proportion of combinations to be chosen to assign more budget if any remaining budgets are available.

Output: Θ: set of configurations

`// 1-Generating combinations`

$k = \prod_{i=1}^{z} |\mathcal{O}_i|$ `// maximum number of possible combinations`

if $k > K$ **then**

  $(\mathbb{O}, k) =$ RandomRegrouping$(\mathbb{O}, K)$ `// Algorithm 10`

`// Create a list` s `of all possible combinations from` $\mathbb{O}$

$s = \{s_1, \ldots, s_k\} = \{G_1^1, \ldots, G_1^{g_1}\} \times \ldots \times \{G_z^1, \ldots, G_z^{g_z}\}$

`// 2-Allocate budgets to` k `combinations`

$l_c = \frac{B_{init}}{k}$ `// number of inital samples per combination`

$m = \frac{1}{k} \sum_{i=1}^{k} (|\Lambda_{s_i}| + |s_i|)$ `//` $|s_i|$ `is the number of all unique algorithms and` $|\Lambda_{s_i}|$ `is the number of hyperparameters`

$\Theta = \varnothing$ `// set of initial configurations`

foreach $j \in \{1, \ldots, k\}$ **do**

  $l_j = \lfloor l_c \times \frac{|\Lambda_{s(j)}| + |s_j|}{m} \rfloor$ `//` l_j `is the number of samples for the combination` s_j

  $l_j = \begin{cases} 1, & \text{if } l_j = 0. \\ l_j, & \text{otherwise.} \end{cases}$

if $B_{remain} = B_{init} - \sum_{j=1}^{k} l_j > 0$ **then**

  `// Randomly allocate` B_{remain} `to the top` $\frac{k}{\eta}$ `combinations based on the number of algorithms and hyperparameters`

`// 3-Sampling Configurations`

foreach $j \in \{1, \ldots, k\}$ **do**

  $\Theta_j = \varnothing$ `// feasible configurations in the` j^{th} `combination`

  while $|\Theta_j| \leq l_j$ **do**

    $\Theta_j = \Theta_j \cup$ Sampling$(s_j, \Lambda_j, l_j - |\Theta_j|)$

    `//` Sampling `is done via an existing approach, here we choose quasi-random sampling with minor adjustments`

    foreach $\lambda \in \Theta_j$ **do**

      if CheckForbidden(λ) **then**

        $\Theta_j = \Theta_j \setminus \lambda$

  $\Theta = \Theta \cup \Theta_j$

Algorithm 10: Random Regrouping

Input: $\mathbb{O} = \left(\{G_1^1, \ldots, G_1^{g_1}\} \times \ldots \times \{G_z^1 \ldots G_z^{g_z}\}\right)$: sequence of operators, K: number of combinations

Output: $\mathbb{O}_{new}$: new sequence of operators, k: new number of combinations

$k = K$ `// number of all possible combinations`

$S = \varnothing$ `// split solutions`

$C_1 = \{1, \ldots, g_1\}, \ldots, C_z = \{1, \ldots, g_z\}$ `// set of possible groupings of` $\mathcal{O}_{i \in \{1,\ldots,z\}}$

`// List out all split solutions`

Create a list of all possible splits $H = \{h_i\}$ where $h_i = (c_1, \ldots, c_z) : c_j \in C_j \forall j$

`// Select split solutions which can produce` k `combinations,` $k \leq K$

while $S = \varnothing$ **do**

  $S = \{h = (c_1, \ldots, c_z) \in H : \left(\prod_j c_j\right) = k\}$

  if $S = \varnothing$ **then**

    $k = k - 1$

$s_{chosen} \sim \mathcal{U}(S)$`// randomly choose one solution`

$\mathbb{O}_{new} = (\varnothing_1, \ldots, \varnothing_z)$, $i = 1$

foreach $c_i \in s_{chosen}$ **do**

  `//` c_i `is the number of groups to be created`

  $n_i = |\mathcal{O}_i|$ `// number of groups in the` i^{th} `operator`

  if $c_i = n_i$ **then**

    $\mathcal{O}_i = \{\{G_i^1\}, \ldots, \{G_i^{n_i}\}\}$ `// when` $c_i = n_i$

  else if $c_i = 1$ **then**

    $\mathcal{O}_i = \{G_i^1, \ldots, G_i^{n_i}\}$ `// merging all predefined groups`

  else

    $G = \varnothing$, $\mathcal{O}_0 = \mathcal{O}_i$

    while $n_i > 0$ **do**

      $G_0 = \varnothing$, $n_{size} = \lceil \frac{n_i}{c_i} \rceil$

      if $n_i > n_{size}$ **then**

        $G_0 =$ {Random pick n_{size} items in $\mathcal{O}_0$}

      else

        $G_0 = \{\mathcal{O}_0\}$

      $G = G \cup G_0$, $\mathcal{O}_0 = \mathcal{O}_0 \setminus G_0$, $n_i = |\mathcal{O}_0|$

    $\mathcal{O}_i = \{G\}$

  $\mathbb{O}_{new}^{(i)} = \mathcal{O}_i$, $i = i + 1$

return $\mathbb{O}_{new}, k$

since, in some cases, the computational optimization budget, i.e., the run time limit, can run out before finishing this initialization step.

The RandomRegrouping method used in Algorithm 9 is presented in Algorithm 10. For a sequence of operators that consists of multiple groupings of algorithms, it produces, via a regrouping, k combinations of operators ($k \leq K$) using the following steps:

1. Step 1 (lines $3-9$): Based on the number of the grouping in operators, we list out all possible solutions of regrouping to have k combinations.

2. Step 2 (line 10): Randomly choose one solution $s_{chosen} = (c_1, \ldots, c_z)$ where c_i is the number of groupings to be created for the operator $\mathcal{O}_i$.

3. Step 3 (lines $11-27$): For each operator $\mathcal{O}_i$, we randomly group algorithms into c_i groups.

7.2.2 A New Optimization Library for AutoML Optimization

To take advantage of the new sampling approach introduced in Section 7.2.1, we introduce a BO library for AutoML optimization, named BO4ML[5], where the new sampling approach is implemented. In this work, we use the Tree-structured Parzen Estimator (TPE) implemented in Hyperopt [153] for the surrogate model and Expected improvement (EI) [156] for the acquisition function.

7.3 Experimental Setup

This study examines the two experiments introduced in Chapter 4 (Section 4.2 and Section 4.3). In both scenarios, we compare the performance of Bayesian optimization (see Chapter 3. Section 3.1.3) with and without our proposed initial sampling approach.

The first experiment uses similar parameter settings as in Chapter 5; we select two different values of the initial sample size 20 and 50. We use a budget of 500 function evaluations. The 5-fold cross-validation approach and the averaged geometric mean values over 10 repetitions are reported. The selected classification algorithms are not grouped together. The resampling techniques are grouped into

[5] The library is published at `https://github.com/ECOLE-ITN/NguyenSSCI2021`.

four groups "Over-resampling", "Under-resampling", "Combine-resampling", and "No-sampling", as suggested in [47], [48].

In the second experiment, we used a budget of 50 samples for the initial sampling. All the experiments performed 10 runs with different random seeds, with a time limit of 1 hour. The performance of a single configuration is limited to 10 minutes with 4-folds cross-validation on training data, i.e., the evaluation of a fold is allowed to take 150 seconds. The evaluation of a configuration is aborted and returns zero if any folds have an error, for example, due to an infeasible configuration or timeout. Then, the average accuracy values for the test data over 10 runs are reported. Finally, the selected algorithms are grouped, according to the suggestions in [46], [151].

The implementation of the proposed methods is published in a git-repository[5] and PyPi-repository[6]. The experiment scripts for the reproducibility of the reported results are provided in a git-repository[7].

7.4 Results and Discussion

In this section, we report and discuss the results obtained from using the above experimental setups. Our experiments has two objectives. First, we compare the performance of Bayesian optimization with the help of our proposed sampling approach with that without our contributions in terms of AutoML optimization for class-imbalance problems, with a search space of two operators. Second, we compared them against state-of-the-art AutoML frameworks with a search space of six operators.

7.4.1 First experimental results

The results of the first experiment are presented in Table 7.2 to illustrate the performance of BO with and without the help of our proposed approach for two different initial sample sizes, that is, 20 (left, not shaded) and 50 (right, gray shaded). In both scenarios, the best performance for the corresponding dataset is highlighted in bold. A method that performs significantly worse than the best method according to the Wilcoxon signed-rank test with $\alpha = 0.05$ is underlined. A value labeled * indicates the best result obtained for the corresponding dataset. The two extra rows at the end display the additional summaries. The first extra

[6] `https://pypi.org/project/BO4ML`.

[7] `https://github.com/ECOLE-ITN/NguyenSSCI2021/tree/assets/SSCI-Experiments`.

Table 7.2: Average geometric mean (rounded to 4 decimals) based on two different initial sampling settings, i.e., the Hyperopt approach and our approach (BO4ML), over 10 repetitions for the 44 examined datasets, ordered by increasing imbalance ratio (#IR) value. The two extra rows display summaries for each scenario, i.e., 20 and 50 initial samples: (1) *Highest performance* shows the number of times the optimizer achieved the highest value. (2) *Significantly better performance* shows the number of times the optimizer was significantly better than the competitor.

Dataset	**#IR**	**20 initial samples**		**50 initial samples**	
		Hyperopt	**BO4ML**	**Hyperopt**	**BO4ML**
glass1	1.82	0.7935	**0.7944**	*__0.7970__	0.7944
ecoli-0_vs_1	1.86	**0.9864**	**0.9864**	0.9864	*__0.9868__
wisconsin	1.86	0.9814	**0.9817**	0.9818	*__0.9819__
pima	1.87	*__0.7712__	0.7696	0.7703	**0.7707**
iris0	2	*__1__	*__1__	*__1__	*__1__
glass0	2.06	**0.8777**	0.8748	0.8740	*__0.8853__
yeast1	2.46	0.7319	**0.7332**	0.7321	*__0.7345__
haberman	2.78	*__0.7049__	0.7012	0.6991	**0.7040**
vehicle2	2.88	0.9908	*__0.9927__	0.9912	**0.9918**
vehicle1	2.9	**0.8690**	0.8684	0.8713	*__0.8735__
vehicle3	2.99	0.8463	**0.8486**	0.8416	*__0.8506__
glass-0-1-2-3_vs_4-5-6	3.2	*__0.9567__	0.9539	0.9534	**0.9553**
vehicle0	3.25	*__0.9876__	0.9867	**0.9867**	0.9867
ecoli1	3.36	0.9038	*__0.9053__	**0.9050**	0.9043
new-thyroid1	5.14	**0.9980**	0.9972	*__0.9983__	0.9966
new-thyroid2	5.14	*__0.9972__	0.9964	0.9952	**0.9966**
ecoli2	5.46	**0.9363**	0.9353	*__0.9365__	0.9360
segment0	6.02	*__0.9993__	0.9992	0.9992	**0.9992**
glass6	6.38	0.9488	**0.9514**	*__0.9518__	0.9511
yeast3	8.1	**0.9423**	0.9421	0.9427	*__0.9441__
ecoli3	8.6	0.9038	**0.9059**	0.9064	*__0.9072__
page-blocks0	8.79	*__0.9475__	0.9472	**0.9464**	0.9457
yeast-2_vs_4	9.08	**0.9549**	0.9542	*__0.9554__	0.9531
yeast-0-5-6-7-9_vs_4	9.35	**0.8245**	0.8177	*__0.8261__	0.8193
vowel0	9.98	0.9567	*__0.9593__	0.9525	**0.9561**
glass-0-1-6_vs_2	10.29	0.8404	**0.8421**	0.8334	*__0.8460__
glass2	11.59	*__0.8504__	0.8461	0.8462	**0.8471**
shuttle-c0-vs-c4	13.87	*__1__	*__1__	*__1__	*__1__
yeast-1_vs_7	14.3	0.7991	**0.8013**	*__0.8033__	0.8010
glass4	15.46	*__0.9390__	0.9230	0.9299	**0.9324**
ecoli4	15.8	**0.9712**	0.9694	0.9632	*__0.9737__
page-blocks-1-3_vs_4	15.86	**0.9931**	0.9874	0.9917	*__0.9944__
abalone9-18	16.4	*__0.8899__	0.8829	0.8856	**0.8859**
glass-0-1-6_vs_5	19.44	0.9494	*__0.9571__	0.9564	**0.9565**
shuttle-c2-vs-c4	20.5	*__1__	*__1__	*__1__	*__1__
yeast-1-4-5-8_vs_7	22.1	0.6989	**0.7024**	*__0.7052__	0.7045
glass5	22.78	**0.9589**	0.9558	0.9591	*__0.9595__
yeast-2_vs_8	23.1	0.8136	*__0.8348__	0.8136	**0.8150**
yeast4	28.1	0.8764	**0.8788**	0.8782	*__0.8788__
yeast-1-2-8-9_vs_7	30.57	**0.7500**	0.7489	0.7397	*__0.7538__
yeast5	32.73	*__0.9802__	0.9798	*__0.9802__	0.9800
ecoli-0-1-3-7_vs_2-6	39.14	*__0.9265__	0.9076	**0.9113**	0.8982
yeast6	41.4	**0.8953**	0.8918	0.8939	*__0.8955__
abalone19	129.44	0.7958	**0.7974**	0.7992	*__0.7998__
Highest performance		15	8	12	19
Significantly better performance		0	2	0	4

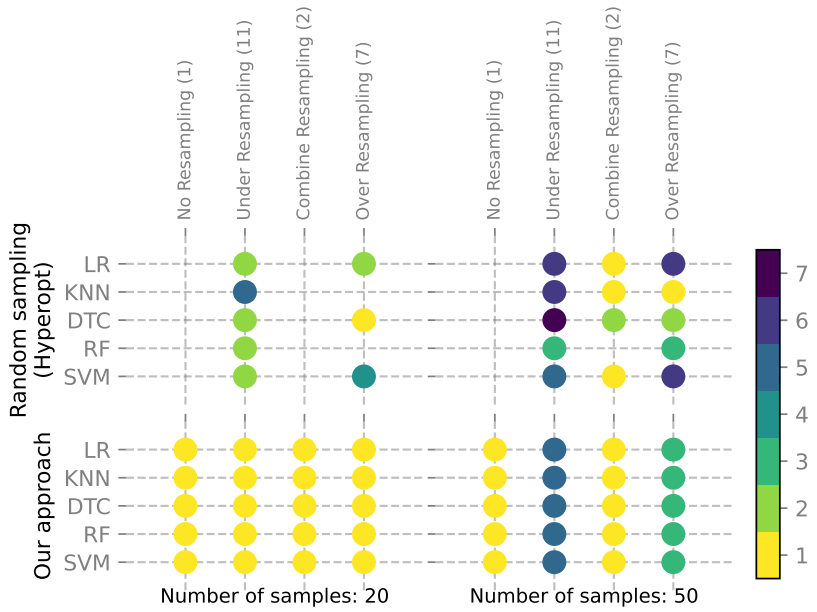


Figure 7.1: Illustration of the number of samples allocated to different combinations of methods in random sampling implemented in Hyperopt (top rows) vs. our proposed approach (bottom rows). The cases with 20 (left) and 50 (right) samples are shown.

row shows the number of times each scenario achieved the highest value over 44 datasets. The last extra row indicates the number of times each approach was significantly better than the others in the group. From the table, we can observe the following.

- In 20 initial samples scenario, Hyperopt achieved the best results on 28/44 cases, and our approach on 20/44 cases. However, our approach significantly wins on 2 tested cases, i.e., "ecoli3" and "yeast-2_vs_8" and is not significantly worse than Hyperopt in any tested cases.
- In the second scenario, our approach achieves the highest value on 31/44 cases and Hyperopt- on 16/44 cases. Similarly, our approach is not significantly worse than Hyperopt in any tested cases but significantly better on 4 examined datasets, i.e., "glass0", "yeast1", "ecoli4", "yeast-1-2-8-9_vs_7".

To investigate the sampling behavior of both approaches for the initial sample sizes, we provided two plots: Figure 7.1 shows the distributions of the samples and Figure 7.2 shows the distributions at the level of the individual algorithms. In both plots, the case with 20 samples is shown on the left and 50 on the right of the plot,

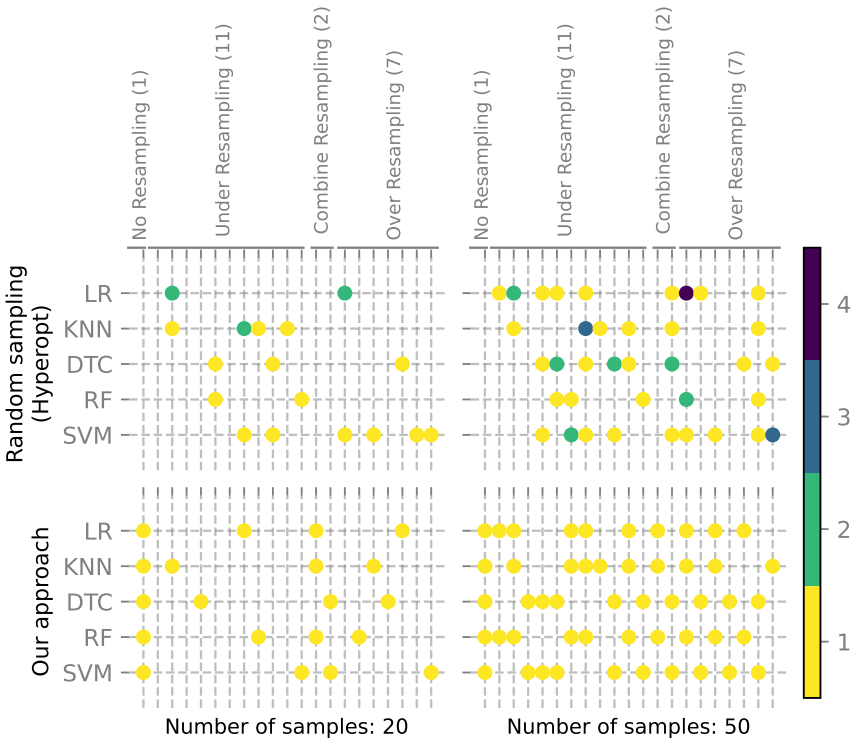


Figure 7.2: Illustration on the distribution of samples obtained via initial sampling methods on the level of individual methods, i.e., under resampling has 11 algorithms, combine resampling has 2 algorithms, over resampling has 7 algorithms and no resampling. The left part shows the case with 20 samples, while the case with 50 samples is shown on the right.

respectively. Looking at these figures, we observe that our approach samples all combinations of groupings over two operators for both sample sizes. By contrast, the sampling strategy used in the Hyperopt samples has less coverage in terms of these combinations. This is because we consider the choice of algorithms in operators to be different from categorical parameters, whereas Hyperopt does not. The plots clearly explain why BO performs better with the help of our approach.

7.4.2 Results of second experiment

The results of the second experiment are presented in Table 7.3. The third and fourth columns show our experimental results, i.e., TPE with and without our sampling approach. The remaining columns contain the results obtained using other AutoML frameworks according to [22]. This table reports the average accuracy over

Table 7.3: Average accuracy (rounded to 5 decimals) over 10 repetitions for the 73 OpenML datasets, ordered by #Task id.

OpenML IDs		Our Experiments		Existing AutoML frameworks [22]					
#TaskID	Dataset Name (#ID)	**TPE** *with our sampling* (**RobustAutoML**)	**TPE** *without our sampling* (**Auto-Hyperopt**)	**Auto-sklearn**		**HPsklearn**	**TPOT**	**ATM**	**H2O**
				SMAC	**Random**				
3	kr-vs-kp (3)	**0.99656**	0.99510	0.98986	0.99062	0.99051	0.99431	0.99326	0.99426
12	mfeat-factors (12)	**0.98417**	0.98117	0.97767	0.97633	0.94758	0.97333	0.98178	0.97433
15	breast-w (15)	0.97952	0.97048	0.96875	0.95873	0.96000	0.96571	**0.98474**	0.96286
23	cmc (23)	0.57285	0.55158	0.54638	0.53262	0.53047	0.55882	**0.58100**	0.53733
24	mushroom (24)	**1**	**1**	**1**	0.99993	**1**	**1**	**1**	0.99848
29	credit-approval (29)	0.88744	0.86522	0.87289	0.85507	0.85956	0.86377	**0.89133**	0.86184
31	credit-g (31)	**0.76600**	0.72733	0.73433	0.72400	0.70121	0.74400	0.76578	0.74867
41	sick (42)	**0.95171**	0.92878	0.91954	0.91911	0.92585	0.92732	0.94504	0.93122
53	soybean (54)	**0.86457**	0.83858	0.82008	0.81969	0.75787	0.81811	0.81522	0.82717
2079	vehicle (188)	**0.69502**	0.66018	0.63886	0.62670	0.64072	0.65566	0.64190	0.65570
3021	eucalyptus (38)	**0.99152**	0.98737	0.98288	0.98550	0.97438	0.98746	-	0.98419
3543	irish (451)	**1**	**1**	0.99019	0.99081	0.99404	0.99091	**1**	0.97967
3560	analcatdata_dmf (469)	0.23042	0.21125	0.20365	0.20382	0.19139	0.20833	**0.27028**	0.19542
3561	profb (470)	0.63119	0.64752	0.65687	0.64563	0.63762	0.66832	**0.71221**	0.71089
3904	jm1 (1053)	**0.82404**	0.81393	0.81344	0.81126	0.80998	0.81810	0.82100	0.74819
3917	kc1 (1067)	**0.87393**	0.85972	0.85118	0.85340	0.84044	0.86019	0.86856	0.80869
3945	KDDCup09_appete (1111)	**0.98323**	0.98197	0.98244	0.98228	0.98189	0.98182	-	0.96555
3946	KDDCup09_churn (1112)	**0.92901**	0.92624	0.92725	0.92586	0.92599	0.92624	-	0.78802
3948	KDDCup09_upsell (1114)	0.94345	0.94116	**0.95094**	0.95030	0.95068	0.95085	-	0.93415
7592	airlines (1590)	0.86251	0.85769	0.86938	0.87013	0.86727	**0.87089**	0.85448	0.86656
7593	bank-marketing (1596)	0.70278	0.80902	**0.96395**	0.89143	0.95227	0.94542	0.66390	0.92908
9910	blood-transfusi (4134)	0.80107	0.78073	0.78890	0.77762	0.77798	**0.80249**	0.77087	0.80044
9952	cnae-9 (1489)	**0.91319**	0.90826	0.89716	0.89205	0.89273	0.90450	0.89963	0.89205
9955	first-order-the (1492)	**0.67167**	0.65146	0.65172	0.62795	0.54667	0.61146	0.61097	0.56435
9977	nomao (1486)	0.96525	0.95924	0.96903	0.96656	0.96891	0.97026	0.96055	**0.97146**
9981	phoneme (1468)	0.95093	0.94228	0.94167	0.93117	0.94012	0.94784	**0.96049**	0.95216
9985	one-hundred-pla (1475)	0.61029	0.59853	0.59695	0.58601	0.58293	0.61291	0.60272	**0.61656**
10101	adult (1464)	0.80667	0.76578	0.76667	0.77778	0.78044	0.78711	**0.81956**	0.73378
14952	covertype (4534)	0.97094	0.96623	0.96590	0.96244	0.96964	0.96913	0.96464	**0.97160**
14954	Bioresponse (6332)	**0.83642**	0.81111	0.79012	0.76173	0.76667	0.81009	0.81701	0.78333
14965	Amazon_employee (1461)	0.90307	0.90007	0.90447	0.90398	0.90451	**0.90705**	0.89957	0.90060
14967	PhishingWebsite (23380)	**1**	**1**	0.98265	0.99841	0.97131	**1**	-	**1**
14968	GesturePhaseSeg (6332)	**0.83580**	0.80432	0.77353	0.77058	0.75823	0.81173	0.79155	0.80000
14969	MiceProtein (4538)	0.64001	0.61864	0.67733	0.65004	0.67272	0.67586	0.66217	**0.70165**
34538	cylinder-bands (4550)	**1**	0.99907	**1**	0.99907	0.99983	**1**	**1**	**1**
34539	cylinder-bands (4135)	0.94825	0.94557	0.94761	0.94444	0.94750	0.94891	0.94606	**0.95114**
125920	cjs (23381)	0.63133	0.56200	0.56667	0.55556	0.56844	0.56867	**0.66978**	0.58400

continued on the next page

Table 7.3: Average accuracy (rounded to 5 decimals) over 10 repetitions for the 73 OpenML datasets, ordered by #Task id. – continued from previous page

OpenML IDs		Our Experiments		Existing AutoML frameworks [22]					
#TaskID	Dataset Name (#ID)	TPE *with our sampling* (**RobustAutoML**)	TPE *without our sampling* (**Auto-Hyperopt**)	Auto-sklearn		HPsklearn	TPOT	ATM	H2O
				SMAC	Random				
146195	dresses-sales (40668)	0.77358	0.77321	0.82109	0.79628	0.82886	0.84123	0.77698	**0.86500**
146212	higgs (40685)	0.99965	0.99945	0.99978	0.99968	0.99253	0.99974	0.99955	**0.99987**
146606	numerai28.6 (23512)	0.70605	0.69761	**0.72296**	0.71930	0.70743	0.72031	0.67135	0.71281
146607	SpeedDating (40536)	0.86611	0.85871	0.86291	0.86225	**0.86661**	0.86392	0.86128	0.84968
146800	connect-4 (40966)	0.99969	0.99321	0.99043	0.99506	0.96380	0.99506	**1**	0.99551
146817	dna (40982)	**0.80497**	0.78216	0.78268	0.76364	0.75955	0.79091	0.76415	0.78062
146818	shuttle (40981)	0.88647	0.85845	0.87053	0.85556	0.86913	0.86184	**0.89050**	0.87633
146819	churn (40994)	0.95802	0.93951	0.94074	0.92407	0.92593	0.94547	**0.96975**	0.93642
146820	Devnagari-Scrip (40983)	0.97355	0.97906	0.98612	0.98581	0.95289	0.98540	**0.98657**	0.98574
146821	CIFAR_10 (40975)	**0.99441**	0.98748	0.97264	0.97958	0.98786	0.99422	0.96763	0.99191
146822	MiceProtein (40984)	**0.94473**	0.93189	0.93088	0.93333	0.90664	0.94055	0.92564	0.94185
146824	car (40979)	**0.98433**	0.98117	0.97783	0.97367	0.98121	0.96883	0.97750	0.97600
146825	Internet-Advert (40996)	0.84300	0.83891	**0.87844**	0.84450	0.85060	0.78089	0.82114	0.87341
167119	mfeat-pixel (41027)	0.84647	0.83956	0.86775	0.85378	0.88691	0.88735	0.87540	**0.90047**
167120	Australian (23517)	**0.52257**	0.52134	0.51926	0.51939	0.52033	0.52082	0.51941	0.50635
167121	steel-plates-fa (40923)	0.86652	0.74910	0.74009	0.02169	0.86438	-	**0.89470**	0.58220
167124	wilt (40927)	**0.39675**	0.37813	-	-	0.32093	0.29429	0.32001	0.36389
167125	segment (40978)	0.97713	0.97033	**0.97774**	0.97114	0.97358	0.97398	0.96900	-
167140	climate-model-s (40670)	0.96485	0.95397	0.95962	0.95889	0.96109	0.95931	0.95282	**0.96904**
167141	Fashion-MNIST (40701)	**0.96273**	0.95367	0.95620	0.95313	0.94533	0.96000	0.95007	0.95370
168329	jungle_chess_2p (41169)	0.31690	0.29294	0.30692	0.29566	0.28741	**0.33576**	0.32108	-
168330	APSFailure (41168)	0.68479	0.66670	**0.71814**	0.69273	0.68494	0.69642	0.63788	0.71786
168331	christine (41166)	0.60445	0.59520	0.66933	0.63762	0.65451	0.65075	**0.67940**	0.67841
168332	jasmine (41165)	0.42507	0.38497	**0.44843**	0.39922	0.34203	-	0.35252	-
168335	sylvine (41150)	0.92248	0.91035	0.94334	0.92891	0.87477	0.93850	0.90234	**0.94604**
168337	albert (41159)	0.78205	0.72707	0.64227	-	0.74347	0.72548	0.66063	**0.81928**
168338	MiniBooNE (41161)	0.98303	0.98035	0.74757	0.75042	0.82518	**0.98495**	0.90729	0.95625
168868	guillermo (41138)	0.98985	0.98900	0.99287	0.99137	0.99360	0.99339	0.97097	**0.99369**
168908	riccardo (41142)	0.73659	0.72565	**0.74754**	0.73081	0.71630	0.72645	0.72169	0.72811
168909	dilbert (41163)	0.95040	0.94437	**0.98357**	0.94793	0.97243	0.96254	0.95391	0.96988
168910	fabert (41164)	0.67565	0.66177	0.70255	0.67395	0.69104	0.68336	0.67357	**0.71752**
168911	robert (41143)	**0.83259**	0.80748	0.82009	0.80603	0.80078	0.82366	0.79911	0.80906
168912	volkert (41146)	**0.95709**	0.94655	0.93921	0.94753	0.94675	0.95533	0.93476	0.92510
189354	dionis (1169)	0.64626	0.63957	0.66665	0.59845	0.65080	**0.66895**	0.63671	0.61266
189355	jannis (41167)	0.73916	0.68112	-	-	**0.77971**	-	0.38666	-
189356	helena (41147)	0.64810	0.64737	0.68314	0.66709	0.66694	0.66110	**0.80064**	0.64798
Number of cases achieved the highest values		28	3	11	-	3	9	18	15
Significant wins over other approaches		23	-	9	-	1	6	13	13

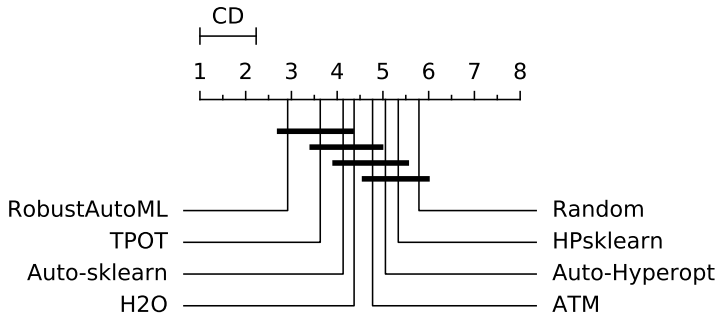


Figure 7.3: Comparison of all approaches against each other with the Nemenyi test with 5% significance level.

10 repetitions to illustrate the performance differences between the two implemented approaches in our AutoML framework[8], i.e., TPE with (RobustAutoML) and without (Auto-Hyperopt) our sampling approach, to compare them against other well-known AutoML frameworks, i.e., Auto-sklearn-SMAC (Auto-sklearn) and Auto-sklearn-Random search (Random), HPsklearn, TPOT, ATM, and H2O. Values in bold indicate the highest values in the corresponding dataset. Underline values indicate significantly different results from the best method according to a Wilcoxon signed-rank test with $p < 0.05$. The two extra rows at the end show the additional summaries. The first extra row shows the number of times each approach achieved the highest performance over 73 examined datasets. The last row presents the number of cases in which these methods significantly outperformed the other compared methods.

The results allow the following insights:

- Comparing the results of approaches using the search space of Auto-Sklearn includes our two approaches, Auto-Sklearn and Random Search. First, it is not surprising that all Bayesian optimization approaches perform better than random search in most tested cases. This has been demonstrated in other studies [40], [47]. Second, Auto-Sklearn won more tested cases than Auto-Hyperopt with the same search space. A possible explanation for this might be that Hyperopt lacks support for k-fold cross-validation yet, while SMAC, the BO variant used in Auto-Sklearn, uses racing algorithms to skip performing on unnecessary folds. Consequently, within the

[8]For readability, RobustAutoML stands for TPE with our sampling approach, and Auto-Hyperopt stands for the original version of TPE implemented by Hyperopt without our improvement.

same budget of time, Auto-Hyperopt evaluated a much smaller number of configurations than Auto-Sklearn. Lastly, the experimental results clearly indicate that the performance of TPE with the help of our sampling approach significantly improves.

- From the results of three approaches using TPE, we can observe that: Firstly, comparing the two approaches that do not use our sampling, i.e., HPsklearn vs. Auto-Hyperopt, we can conclude that the search space of Auto-Sklearn does not improve the final performance of TPE. Secondly, the results clearly demonstrate that significant improvement was achieved with the help of our sampling approach. Our approach outperforms others 23 times, significantly winning Auto-Hyperopt in 16 cases and HPsklearn in 20 cases. Furthermore, in all 3 cases where Auto-Hyperopt achieves the highest results, e.g., tasks 24, 3543, and 14967, both our approach and Auto-Hyperopt get maximum accuracy in those cases. On the other hand, HPsklearn got the highest results in 3 cases, e.g., tasks 24, 146607, 189355, but never performed significantly better than our approach in any of those.

- Overall, our proposed approach shows the best results in more cases than all other approaches compared, namely 28/73. Moreover, according to the results of the Wilcoxon signed-rank test, our approach also significantly outperforms other compared approaches in 23/73 test cases. However, Auto-Hyperopt, without our improvement, does not win for any of the datasets.

When all approaches are compared, Friedman's test reveals a significant difference in average accuracy with $p = 6.35 \cdot 10^{-11}$. Thus, we performed a post-hoc multiple comparison test with the Nemenyi test ($\alpha = 0.05$), shown in Figure 7.3. Approaches that have a distance higher than CD[9] are considered significantly different. According to this figure, we conclude that RobustAutoML is better than both TPE-based approaches and better than five other AutoML frameworks, such as, H20, ATM, Auto-Hyperopt, HPSklearn, and Random Search.

7.5 Conclusions and Future Work

In this chapter, we formulated AutoML as an optimization process for the machine learning pipeline. Then, we built on this paradigm, we proposed a new class for modeling the choice of algorithms and the concept of grouping algorithms. Second,

[9]Critical Difference, here CD=1.2288.

a robust sampling approach for Bayesian optimization for AutoML optimization problems was introduced; Third, a BO approach for AutoML optimization was presented, where our proposed sampling approach and new hyperparameter classes were implemented. Lastly, a robust AutoML framework was presented which takes advantage of the proposed BO approach mentioned above.

The experimental results demonstrate the effectiveness of our approaches in two independent experiments over 117 datasets. The results clearly show significant improvement achieved by using our approach.

There are several interesting research directions for extending this study. First, we intend to apply the proposed sampling approach to other AutoML frameworks. Additionally, we plan to apply some pruning approaches such as Hyperband [35] and racing algorithm to reduce the time for evaluating configurations that are not promising by evaluating fewer folds.

CHAPTER 8

An Efficient Contesting Procedure for AutoML Optimization

Classical AutoML-based Bayesian Optimization approaches often integrate all operator search spaces into a single search space. However, one drawback of this strategy is that it can be less robust when initialized randomly than optimizing each operator-algorithm combination individually. To overcome this issue, a novel contesting procedure, **D**ivide **A**nd **C**onquer **Opt**imization (DACOpt), is proposed in this chapter to make AutoML more robust. The DACOpt partitions the AutoML search space into a reasonable number of sub-spaces based on algorithm similarity and budget constraints. Furthermore, throughout the optimization process, DACOpt allocates resources to each sub-space to ensure that (1) all areas of the search space are covered and (2) more resources are assigned to the most promising sub-space. Two extensive sets of experiments on 117 benchmark datasets demonstrate that DACOpt is significantly better than its competitors. Furthermore, an experiment in surface defect classification in steel manufacturing indicated that the proposed contesting procedure significantly improved the performance of BO in real-world applications. The remainder of this chapter is organized as follows. The motivation and introduction are provided in Section 8.1. Section 8.2 presents the relevant background knowledge Divide and Conquer techniques and early-stop strategies. Our contributions are highlighted in Section 8.3, whereas Section 8.4 outlines the experimental setup. Experimental results are discussed in Section 8.5. Next, an investigation of the use of DACOpt in real-world applications is discussed in Section 8.6. Finally, the chapter is summed up and further work is outlined in Section 8.7.

8.1 Introduction

In this study, we evaluated BO-based approaches for solving the AutoML optimization problem. The AutoML optimization (AO) problem is typically considered as a single optimization problem in the BO-based method by merging the optimization space for all algorithms of all operators – this approach is typically refer to as *integrated approach* [14]. The Combined Algorithm Selection and Hyperparameter Optimization (CASH) approach [40] is a commonly used technique, where the AO problem is treated as a hyperparameter optimization (HPO) problem. However, HPO was initially developed to optimize hyperparameters of a single algorithm, where the considered search space is typically smaller, lower-dimensional, and less (even non)-structured than the AutoML search space. Hence, the HPO-based approach is not ideal for handling the AO problem. In order to alleviate the above limitation, we formulate the AO problem as a ML pipeline optimization problem, which is proposed by [15]. This can be seen as a generalization of the CASH approach, where the parameter classes for operator's algorithms, and hyperparameters in an algorithm were clearly identified.

As an alternative to the *integrated approach*, [241] proposed the so-called CASH-oriented Multi-Armed Bandits (MAB) approach to solve the model selection and hyperparameter optimization problem for the classification problem (i.e., selecting a classification algorithm and tuning the hyperparameters, simultaneously) by applying HPO to each classifier separately. However, this might not be applicable to AutoML scenarios, because the number of combinations of algorithms over operators can be up to thousands. Fortunately, [15], [46], [47], [74] have pointed out that operator algorithms can potentially be grouped and that different groups of algorithms perform better on different types of problems, for example, a group of linear classification algorithms performs best on linear classification tasks.

Hence, this study attempts to further improve BO performance for the AO problems by applying the Divide and Conquer (DAC) strategy: the AutoML search space is divided into multiple sub-spaces based on their similarity[1], and each sub-space is solved independently by a separate BO process (candidate). The budget is then allocated to each candidate using a novel competing mechanism, depending on its performance. Consequently, the most promising candidates have a larger tuning budget than the least promising candidates. Therefore, the worst candidates will be 'terminated' as soon as ample evidence against them has been

[1] see grouping approach proposed in [15]

gathered, saving computation time and resources for future assessments in those search areas.

Notably, as our approach handles BO[2] processes independently, it allows multiple optimization processes to be executed simultaneously without affecting the performance. In other words, our technique achieves the same numerical results in both parallel and sequential settings, with the exception of different execution times.

Our contributions: We summarize our main contributions, which are the following:

- We propose a novel contesting procedure, namely DACOpt, to solve the AutoML optimization problem efficiently, which is complementary to the existing BO approaches.
- DACOpt efficiently allocates resources to each sub-space to ensure that (1) all areas of the search space are covered and (2) more resources are assigned to the most promising sub-space. In addition, we provide a proof that our approach fixes the existing gap between serial and parallel BO execution (see Section 8.3.3).
- Two independent empirical studies on a range of AutoML optimization problems with 2 and 6 operators on a total of 117 benchmark datasets demonstrate the superiority of the proposed approaches.
- An empirical experiment on a real-world application of surface defect classification in steel manufacturing indicated that our proposed approach significantly improved BO's performance.

8.2 Background

In this chapter, we review the relevant techniques to the proposed contesting procedure (Section 8.2.1), and early-stop strategies (Section 8.2.2). Other related research to Bayesian Optimization and AutoML optimization can be found in Section 3.1.3.

[2]BO (a.k.a., Sequential model-based optimization) was originally intended as a sequential approach [25], [36].

8.2.1 Contesting procedure for AutoML optimization

AutoML optimization typically is a high-dimensional mixed-variables (continuous, discrete, nominal) optimization problem. In order to handle such a challenge by a BO approach, three facts are considered: (1) BO performs better for low-dimensional problems [242], (2) AO problems have low effective dimensionality [30], [31], and (3) the complexity of AO problem not only comes from its dimensionality, but also from the number of possible combinations of algorithms within the ML pipeline [15].

Divide and Conquer (DAC) [243] is a well-known strategy for handling large problems via decomposing the target problem into c small-scale and low-dimensional sub-problems. Consider an AO problem $p^* = \text{argmax}_{p \in \mathcal{M}} f(p)$. For applying DAC, the approach has to first decompose the AutoML search space into c sub-spaces, and then solve each sub-space by an optimizer. Assuming that we can split the AutoML search space $\mathcal{M}$ into c smaller spaces $\{\mathcal{M}_1, \ldots, \mathcal{M}_c\}$, the DAC approach can be formulated as:

$$p^* = (\underset{p \in \mathcal{M}_1}{\text{argmax}}\, f(p), \ldots, \underset{p \in \mathcal{M}_c}{\text{argmax}}\, f(p)) = (p_1^*, \ldots, p_c^*) \tag{8.1}$$

where p_i^* is the global optimum of sub-space $\mathcal{M}_i$ and p^* is the global optimum of the original search space $\mathcal{M}$.

The existing DAC studies typically treat elements of the input search space on the same level and decomposed by complementing [244]. That is, variables corresponding to sub-space $\mathcal{M}_i$ can change freely while the remaining $|\mathcal{M} \setminus \mathcal{M}_i|$ dimensions are set to some fixed values. However, such approaches cannot be used for the AutoML search space where dimensions are hierarchical and, thus, dependent. As a result, two challenges are faced when using DAC to solve AutoML problems: (1) how to divide the AutoML space $\mathcal{M}$ onto a set of c sub-spaces efficiently; (2) how to optimize resources during the 'conquer' phase, since some sub-spaces' performance might be significantly worse than others. To answer the above questions, we propose (1) a splitting approach based on the combination of groups of operator algorithms [15], (2) adopting efficient early-stop strategies based on the theoretical guarantees (see our discussion in Section 8.2.2) for optimizing resources for the 'conquer' phase.

In addition, since the number of algorithms (and therefore, the number of sets of their parameters) is smaller in the DAC-formulation of the AutoML problem in Equation 8.1 compared to the original formulation in Equation 1.4, the proposed

contesting procedure also concurs the assumption [30], [31] that the AO problem has low effective dimensionality.

8.2.2 Early-stop strategies

As we discussed in Section 1.1, the k-fold cross-validation is usually applied to the AutoML optimization problem to prevent the over-fitting problem as described in Equation 1.4. For readability, let p denote $p_{(\mathcal{A}_{1,\lambda},\ldots,\mathcal{A}_{z,\lambda})}$. The Equation 1.4 is then formulated as:

$$p^* = \underset{p \in \mathcal{M}}{\operatorname{argmax}} \frac{1}{k} \sum_{j=1}^{k} f\left(p, \mathcal{D}_t^j, \mathcal{D}_v^j\right) \tag{8.2}$$

where $f\left(p, \mathcal{D}_t^j, \mathcal{D}_v^j\right)$ is performance of the pipeline setting p when trained and evaluated on the j^{th} cross-validation data fold $\mathcal{D}_t^j$ and $\mathcal{D}_v^j$, correspondingly. As a consequence of using cross-validation, every function evaluation becomes k times more expensive. An early stop strategy, e.g., [32]–[36], [173] allows limiting this issue, since it avoids wasting time and resources on evaluating worse settings over all k folds.

An important concept is to stop investigating a setting as soon as sufficient information indicates that it is ineffective. A setting will only be examined in a few folds in this manner; an iterative elimination function will analyze its performance on the evaluated folds to compare it to other evaluated settings and determine how many folds should be utilized for the considered setting.

The elimination function in racing procedure approaches (see Section 3.2.1) is based on a statistical test procedure, i.e., Friedman test [176], whereas bandit-based approaches (see Section 3.2.2) compare the setting performance directly to the best-known setting. In a number of case studies, both strategies performed well [14], [33], [35], where the task of proposing new settings was commonly handled by a random search (see Section 3.1.2). Unfortunately, the inconsistencies in how settings are assessed may provide additional noise for BO, making it less reliable in suggesting subsequent settings. This means that such approaches should not be used directly and should be adopted only at the level of search sub-spaces, via the termination of unpromising sub-spaces (the detailed discussion on the termination functions is given in Section 8.3.1).

8.3 Proposed approach

We now discuss our proposed contesting procedure for AutoML optimization problems based on the **D**ivide **A**nd **C**onquer strategy, which we call DACOpt.

8.3.1 Algorithm description

We reformulate the AutoML optimization problem in Equations 1.4, 8.1 and 8.2 into the following:

$$p^* = \underset{p \in \mathcal{M}}{\operatorname{argmax}}(p_1^*, \ldots, p_c^*) \tag{8.3}$$

$$\text{s.t.} \quad p_i^* = \underset{p_i \in \mathcal{M}_i}{\operatorname{argmax}} \frac{1}{k} \sum_{j=1}^{k} f\left(p_i, \mathcal{D}_t^j, \mathcal{D}_v^j\right) \tag{8.4}$$

where $\mathcal{M}_i = \mathbb{O}_i \cup \Lambda_i$ denotes the i^{th} sub-space. $\mathbb{O}_i = \mathcal{O}_1^{(i)} \times \ldots \times \mathcal{O}_z^{(i)}$ denotes the possible sequence of operators in $\mathcal{M}_i$, herein $\forall i \in \{1, \ldots, c\}$, $\mathcal{O}_{l \in \{1,\ldots,z\}}^{(i)} = \{\mathcal{A}_l^1, \ldots, \mathcal{A}_l^{n_l}\}$ denotes a set of algorithms of the l^{th} operator $\forall |\mathcal{O}_l^{(i)}| \leq |\mathcal{O}_l|$, and a set of the corresponding hyperparameters of $\mathcal{O}_l^{(i)}$: $\Lambda_{l \in \{1,\ldots,z\}}^{(i)} = \Lambda_l^1 \cup \ldots \cup \Lambda_l^{n_l}$ and $f\left(p_i, \mathcal{D}_t^{(j)}, \mathcal{D}_v^{(j)}\right)$ denotes performance of the setting, similar to Equation 8.2.

The overall proposed structure of the contesting procedure is summarized in Figure 8.1. The process begins with a *Splitter* function to be applied on the input AutoML search space $\mathcal{M}$ to produce c possible sub-spaces. Here, we extend our work from the previous chapter with improvements (a detailed discussion on this function is given in Section 8.3.2). Then, c BO processes are initialized (in the following discussion, the BO processes shall be called candidates). The whole contest is controlled by the *Controller* function, which allocates budgets to each candidate per contest round based on the feedback from the *Elimination* function that decides which candidates will survive into the next round based on their performances so far. As mentioned in Section 8.2.2, we adopt two possible settings for the early-stop functionality. Therefore, two versions of DACOpt are provided, which differ mainly w.r.t. the elimination criteria: (1) based on highest performance (Section 8.3.1.1) and (2) based on a statistical procedure (Section 8.3.1.2).

8.3.1.1 Elimination criteria based on the highest performances

In BO (Section 3.1.3), the acquisition function maximizes the *best-found value* $\Delta_{(t)}^*$ up to time step t. Due to the fact that the goal of AutoML optimization

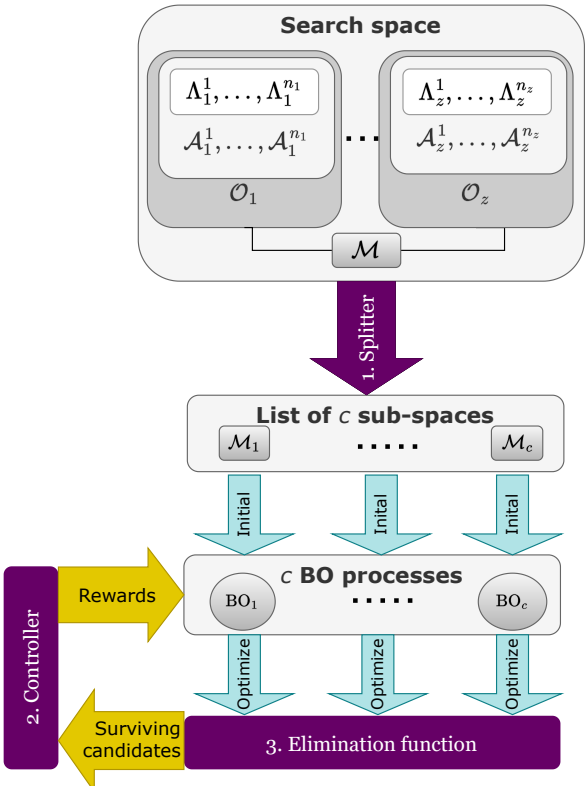


Figure 8.1: The workflow of the contesting procedure.

is to find the setting that achieves the highest performance on the target ML problem, we consider the highest performance as a suitable comparison criterion. Furthermore, the way of computing the budget, step size, and the number of rounds follow the Successive Halving (Chapter 3. Section 3.2.2.1) and Hyperband (Chapter 3. Section 3.2.2.2) approaches with minor adjustments: input parameters of our procedure include the maximum number of sub-spaces to be split c, total optimizing budget B, and the ratio of candidates discarded in each round[3] η. The number of rounds in the contest is then calculated as: $R_{max} = \lfloor \log_\eta(c) \rfloor$. Each round has the same budget $B_r = \frac{B}{R_{max}}$. That is, each of m surviving candidates at the round can have a budget of $b = \lfloor \frac{B_r}{m} \rfloor$. At the end of the round, the *Elimination* function keeps $\lceil \frac{m}{\eta} \rceil$ candidates for the following round. Therefore, the surviving candidate has η times the budget from the previous round.

Our approach is elaborated in Algorithm 11 which requires the maximum

[3] $\eta = 3$, can be changed by user.

number K of sub-spaces to be split and the ratio of candidates discarded in each round, η, as input parameters. This approach consists of the following steps:

- Initialize: Split the original search space into c ($c \leq K$) sub-spaces (line 1). Next, initialize c corresponding Bayesian optimization candidates (lines 2-3). Next, the number of contest rounds is calculated, $R_{max} = \lfloor \log_{\eta}(c) \rfloor$ (line 4).

- Parameter for each round: Based on the number of surviving candidates from the previous round, the number of candidates c_r for the current round r is computed in line 8 for the first round and in line 11 for subsequent rounds. The elimination function discards candidates labeled as badly performing and returns a set of c_r good candidates (line 11). Herein, we simply select the top c_r candidates based on their best-found values. A reasonable budget for each candidate is computed based on the remaining budget, remaining rounds, and the number of surviving candidates (lines 13-14). All the above steps (lines 10-14) are repeated every round, except the first round. In the *first round*, all candidates survive and are given a budget of $b = b_{init}$ (lines 7-9).

- Finally, using value b obtained in the previous step, all surviving candidates continue their optimization processes (lines 15-19).

8.3.1.2 Elimination criteria based on a statistical procedure

As mentioned in Section 8.2.2, our second option was the approach of racing procedures to determine well and badly-performing candidates. This approach also requires a maximum number of sub-spaces K and a level of significance α. Since the effectiveness of BO is mostly seen in the later phases of optimization when it learns to produce better settings, we only consider the best-found value of the initial sampling step for further statistical tests. Unlike the first elimination criteria method, this method does not compute the number of rounds or budget for each round since it completely depends on the statistical results; instead, we use a step size[4] γ to limit budget per round. At the end of the round, a Friedman test [176] is performed to verify whether there is a significant difference between the pair of candidates. If it is the case, a Holm post-hoc test [245][5] is applied to compare the highest-ranked candidate to others. Any candidate that fails the test

[4] $\gamma = 1$, can be changed by user.
[5] Following the recommendations by [246], [247].

Algorithm 11: DACopt based on the highest performance

Input: $\mathcal{M}$: Search space, K: number of sub-spaces to be split, f: objective function, B: maximal number of evaluations, b_{init}: number of evaluations for the initial step in each of c BO processes, η: ratio controlling the proportion of candidates discarded in each round

Output: p^*: the best pipeline setting, Δ^*: the best value

1 $(\{\mathcal{M}_1, \ldots, \mathcal{M}_c\}, c) \leftarrow$ `Splitter`$(\mathcal{M}, K)$ `// divide the input search space into` c `sub-spaces`

Initialization : INITIALIZATION PHASE

2 **for** $\mathcal{M}_i \in \{\mathcal{M}_1, \ldots, \mathcal{M}_c\}$ **do**

3 $\quad (\text{BO}_i, \mathcal{H}_i) \leftarrow$ `BayesianOptimizer`$(\mathcal{M}_i, f, b_{init})$ `// Initialize` BO_i `and its historical data` $\mathcal{H}_i = \{(p_n, \Delta_n)_{n=1}^{\texttt{evaluated}}\}$. (p, Δ) `represent configuration and performance.`

Initialization : CONTESTING PHASE

4 $R_{max} \leftarrow \lfloor \log_\eta(c) \rfloor$ `//` R_{max}`: number of rounds`

5 $r \leftarrow 0$ `//` r`: round number`

6 **while** $r \leq R_{max}$ **do**

7 $\quad$ **if** $r = 0$ **then**

8 $\quad\quad c_r \leftarrow c$; $(\text{BO}_1, \ldots, \text{BO}_{c_r}) \leftarrow$ `RandomPermute`$(\text{BO}_1, \ldots, \text{BO}_c)$ `// Note: at the first round` $c_r = c$`, but the order of candidates are shuffled.`

9 $\quad\quad b \leftarrow b_{\text{init}}$ `//all candidates have an equal budget` $b_{\texttt{init}}$

10 $\quad$ **else**

11 $\quad\quad c_r \leftarrow \lceil \frac{c_{\text{previous}}}{\eta} \rceil$ `// number candidates for` $r^{\texttt{th}}$ `round`

12 $\quad\quad (\text{BO}_1, \ldots, \text{BO}_{c_r}) \leftarrow$ `Eliminate`$((\text{BO}, \mathcal{H})_{i \in \{1,\ldots,c\}}, c_r)$ `// Select good` c_r `candidates, ordered by performance/rank`

13 $\quad\quad B_r \leftarrow \lfloor \frac{B}{R_{max} - r} \rfloor$ `//` B_r`:total budget for` $r^{\texttt{th}}$ `round`

14 $\quad\quad b \leftarrow \lfloor \frac{B_r}{c_r} \rfloor$ `//` B_r`:budget per candidate`

15 $\quad$ **for** $\text{BO}_i \in \{\text{BO}_1, \ldots, \text{BO}_{c_r}\}$ **do**

16 $\quad\quad \text{BO}_i$ .ADDBUDGET(b) `//Add budget` b `to` BO_i

17 $\quad\quad (\text{BO}_i, \mathcal{H}_i) \leftarrow \text{BO}_i$ .OPTIMIZE() `//Continues` BO_i `process`

18 $\quad\quad B \leftarrow B - b$`// Update the remaining budgets`

19 $\quad c_{\text{previous}} \leftarrow c_r$; $r \leftarrow r + 1$

20 **Return** $p^*, \Delta^* = \text{argmax}_{p,\Delta} \{\mathcal{H}\}_{i \in \{1,\ldots,c\}}$;

is removed from the list of surviving candidates. This loop is repeated until the best candidate is found. This process, summarized in Algorithm 12, consists of the following steps:

- Initialize: Using the same split function as Algorithm 11, to produce c ($c \leq K$) sub-spaces (line 1). All candidates are initialized with the minimum

required budget b_{init} (line 2-3).

- The main operates in the contesting phase: maintain a set of surviving candidates[6]. A statistical test is performed at each round to determine if there are any pairs of candidates that are significantly different (lines 7-10). If the null hypothesis is false, we first perform a rank test, e.g., the Wilcoxon signed rank test, to detect the highest-ranked candidate (line 12). Next, a post-hoc test is applied to the pair of every candidate and the highest-ranked candidate (line 13). Any candidate that fails the test is removed from the set of surviving candidates (line 14). Next, a budget γ is added to each candidate in the survived set (line 18) and continues the tuning process with the added budget (line 19). This procedure is repeated until the total budget runs out.

Lastly, both options naturally support parallel implementation. We require the number of maximum available threads $\tau, (\tau \geq 1)$, as an extra input parameter for the parallel mode. The parallel mode will be discontinued when the best sub-space is found. In both algorithms, parallel mode is applied to execute the BO processes.

[6]Note for the contesting phase: Since the effectiveness of BO is mainly determined during the initial sampling step as it learns to produce better settings. Therefore, we consider only the best-found value from the initial sampling step for further statistical tests, and we perform statistical tests only when the sample size exceeds 2.

Algorithm 12: DACOpt based on the statistical test

Input: $\mathcal{M}$: Search space, K: number of sub-search spaces, f: objective function, B: maximal number of evaluations, b_{init}: minimum evaluations per sub-search space, $\gamma = 1$: step size, $\alpha = 0.05$: level of significance

Output: p^*: the best pipeline setting, Δ^*: the best value

$\{\mathcal{M}_1, \ldots, \mathcal{M}_c\}, c \leftarrow$ SPLITTER$(\mathcal{M}, K)$ // divide the input search space into c sub-spaces, $c \leq K$

// BEGINNING OF INITIAL PHASE

for $\mathcal{M}_i \in \{\mathcal{M}_1, \ldots, \mathcal{M}_c\}$ **do**

  $(\text{BO}_i, \mathcal{H}_i) \leftarrow$ BayesianOptimizer$(\mathcal{M}_i, f, b_{init})$ // Initialize BO_i and its historical data $\mathcal{H}_i = \{(p_n, \Delta_n)_{n=1}^{\text{evaluated}}\}$. (p, Δ) represent configuration and performance.

// BEGINNING OF CONTESTING PHASE

$\{(\text{BO}_i, \mathcal{H}_i)\}_{i=1}^{survive} \leftarrow \{(\text{BO}_i, \mathcal{H}_i)\}_{i=1}^{c}$ // All candidates survive

$c_r \leftarrow c$ // c_r number of surviving candidates

while $B \geq 0$ **do**

  if $c_r < 3$ **then**

    STAC $\leftarrow$ WILCOXONTEST() // Init WILCOXONTEST if $c_r < 3$

  else

    STAC $\leftarrow$ FRIEDMANTEST() // Init FRIEDMANTEST if $c_r \geq 3$

  // Performs the chosen statistical test with α to detect if there is at least one pair of candidates that are significantly different

  if $(\neg STAC(\{\mathcal{H}_i\}_{i=1}^{survive}, \alpha))$ & $c_r > 1$ **then**

    $\mathcal{H}_{i^*} = \text{argmax}\, \text{RANKING}(\{\mathcal{H}_i\}_{i=1}^{survive})$ // detect the highest ranked $\mathcal{H}_{i^*}$ among the surviving candidates based on a ranking test, e.g., Wilcoxon signed rank test

    $\{(\text{BO}_i, \mathcal{H}_i)\}_{i=1}^{survive} \leftarrow$ HOLM_POST_HOC_TEST// detects candidates not significantly worse than $\mathcal{H}_{i^*}$

    $c_r \leftarrow$ number of surviving candidates

  else if $c_r = 1$ **then**

    $\gamma \leftarrow B$// If $c_r = 1$, allocate the remaining budget

  for $\text{BO}_i \in \{\text{BO}_i\}_{i=1}^{survive}$ **do**

    BO_i .ADDBUDGET(γ) // Add budget γ to the selected BO_i

    $(\text{BO}_i, \mathcal{H}_i) \leftarrow \text{BO}_i$ .OPTIMIZE() // Continues BO_i process.

    $B \leftarrow B - \gamma$ // Update the remaining budgets

Return $p^*, \Delta^* = \text{argmax}_{p,\Delta} \{\mathcal{H}\}_{i \in \{1,\ldots,c\}}$

8.3.2 The Splitting approach

In this section, we briefly describe of the splitting function (line 1 of the Algorithm 11 and Algorithm 12). The AutoML search space is complex owing to the number of operators and their choice of algorithms. In practice, the search space can lead to thousands of algorithm combinations over operators. Because the tuning budget is relatively small compared to a large number of possible pipelines over operators, we propose grouping them based on their similarities with the assumption that a good choice for one algorithm in the group can also serve as a good choice for other algorithms in the group. Consequently, the sampler can maximize the coverage of the search space by sampling at the group level instead of at the algorithm level.

The concept of grouping is similar to that done in Chapter 7. However, it mainly focused on initial sampling, where the budget was typically much smaller than the number of combinations of algorithm's groups. As a result, the group's level is limited to only 1, i.e., the group's item is a specific choice for the algorithm. In this study, we consider a scenario in which a set of algorithms under a group might be slightly different. For example, while the RandomOverSampler and SMOTE algorithms are both oversampling techniques (see the bottom plot in Figure 8.2), they differ significantly: RandomOverSampler randomly generates more data for minority classes, whereas SMOTE is based on interpolation. To account for the possible hierarchical groupings of the algorithms, we extended Algorithm 10 to allow any group at any level to contain child groups. Therefore, the required groups are produced by downing (or upping) the levels to minimize randomness. Consequently, the resulting subspaces are purer, that is, the difference between items in a group is minimized, representing their actual relationship.

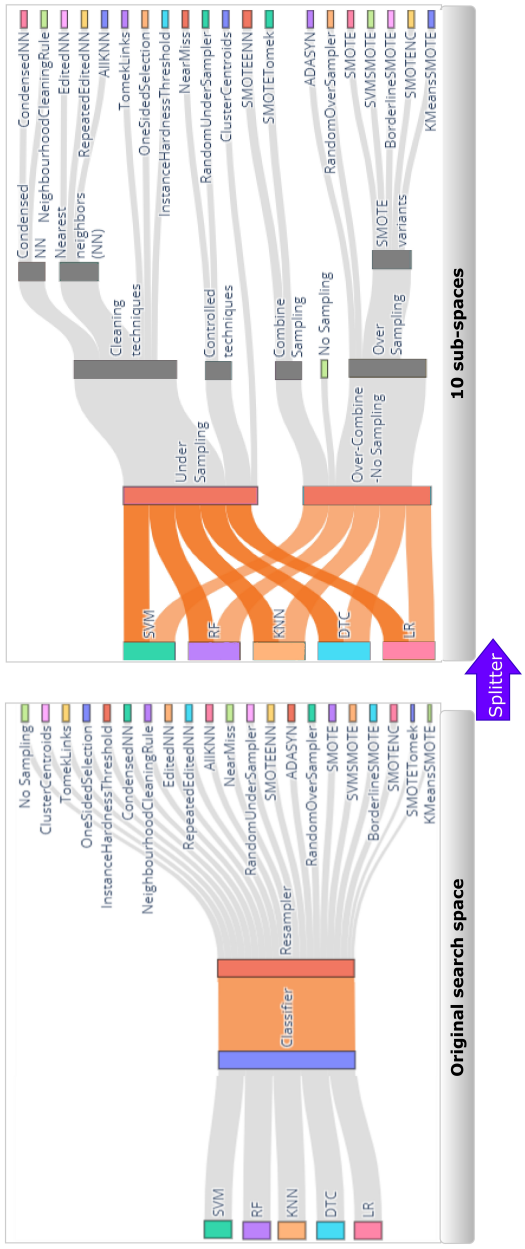


Figure 8.2: Illustration of the Splitting approach on a search space of two operators, i.e., Classifier and Resampler, used in our first experiment. The connection in orange indicates a search space/ sub-space.

8.3.3 Fixing the gap between serial and parallel BO

Bayesian optimization, called otherwise the Sequential model-based optimization (SMBO), is naturally sequential. However, most modern optimizer-based BO approaches include a parallelized version in addition to the original BO method. AutoML-based BO is typically parallelized by either assessing in parallel (1) cross-validation folds or (2) multiple settings, e.g., [26], [38], [248]. While the first approach focuses on parallelizing evaluations inside the objective function, it does not affect BO, however, it is efficient when k is less than the available resources. The second approach might lead to inefficient solutions proposed by BO, in terms of the number of function evaluations. Since the objective function is expensive, we have to choose a configuration that might perform best. In the following, we discuss how parallelized BO can lead to poorer results compared to serial approaches.

Let us consider a noiseless function $f : \mathcal{M} \subset \mathbb{R}^d \rightarrow \mathbb{R}$ and its real-valued surrogate model $\hat{f} = \{\mathcal{P}(p_i, \Delta_i)_{i=1}^t\}$ for time step t. At a new step $t+1$, a sampling approach (randomly) generates a set of solutions $\{\hat{p}_1, \ldots, \hat{p}_n\}$. Those later will be estimated by the surrogate model $\hat{f}$ and used to propose **one** setting $p_{t+1} \in \{\hat{p}_1, \ldots, \hat{p}_n\}$ by maximizing the acquisition function in Eq. 3.5. The set of m next settings from the time step t of the sequential approach is $\{p_{t+1}^{\curlywedge} = \text{argmax}_{p \in \mathcal{M}} \mathbb{E}[I(p_t)], \ldots, p_{t+m}^{\curlywedge} = \text{argmax}_{p \in \mathcal{M}} \mathbb{E}[I(p_{t+m})]\}$. In contrast, the parallel approach proposes a set of solutions $\{p_{t+1}^{\shortparallel_1}, \ldots, p_{t+1}^{\shortparallel_m}\} \in \text{argmax}_{p \in \mathcal{M}} \mathbb{E}[I(p_t)]$. Let $\overline{p} = |f(p) - \hat{f}(p)|$ denote the difference between the performance of the setting p on the true objective function f and its surrogate $\hat{f}$. Clearly, the quality of BO in suggesting new solution(s) is highly dependent on $\hat{f}$ and the statistical property of $\hat{f}$ (i.e., uncertainty) at time t, which significantly increases as more historical data is collected. Thus, $\sum_{j=1}^{m} \overline{p_{t+j}^{\curlywedge}} \geq \sum_{j=1}^{m} \overline{p_{t+j}^{\shortparallel_j}}$. Hence, the quality of m additional time steps in the sequential method may be more robust than those in the parallel technique. Thus, there is a discrepancy between the current serial and parallel BOs.

For the reasons above, we use sequential BO to solve each search sub-space. Fortunately, BO processes in our proposed procedure are independent (see Figure 8.1). Therefore, we introduce a partly-parallel approach instead of fully parallel. Instead of proposing a set of future solutions from a single search area like the fully parallel technique does, in order to ensure the best performance of BO at every iteration, DACOpt proposes a set of next setting solutions as sequential approach from multiple independent search areas: $p_{t+1}^{\curlywedge_i} = \text{argmax}_{p \in \mathcal{M}_i} \mathbb{E}[I(p_t)], \forall i \in \{1, \ldots, c\}$.

Table 8.1: Proposed DACOpt approaches compared in this study

Name	Contesting procedure		BO variant	
	Highest	Statistical	BO4ML	Hyperopt
DAC-HB	✓		✓	
DAC-HH	✓			✓
DAC-SB		✓	✓	
DAC-SH		✓		✓

Thus, $p_{t+1}^{r_i}$ in either serial or parallel situations are exactly the same. For parallel computing, a parallel pool of m available workers will be repeated $\lceil \frac{c}{m} \rceil$ times to finish c processes. The last iteration of that parallel pool is partly parallel if (c mod m) > 0 and fully parallel otherwise. As a result, our approach holds the same effectiveness in both cases.

The key benefits of our DACOpt approach are as follows:

- Based on the performance of the related BO process, the budget adaptively redistributes to the search area[7]. As a result, the budget is distributed effectively.
- As a partly-parallel BO variant, the proposed approach has parallel efficiency without harming BO performance.
- BO performance and robustness can be increased since each BO process optimizes a relatively small low-dimensional search space independently.

8.4 Experimental Setup

In order to evaluate the robustness and general applicability of our proposed approach, we compare it to other state-of-the-art AutoML optimization approaches. We reproduce the experimental setup with a total of 117 benchmark datasets on two scenarios with optimization of 2 operators (Section 4.2) and 6 operators (Section 4.3). In both scenarios, we compare the performance of BO-based variants with the TPE surrogate model BO4ML (Chapter 7) and Hyperopt [153], with the two proposed contesting procedures against those without such procedure (see Table 8.1). Our local parameter settings are summarized in Table 8.2.

Both experiments used similar parameter settings as Chapter 7. All approaches use an initial sample size of 50 function evaluations.

[7] In this thesis, we use the term *search area* to refer to an area (subset) of the search space.

Table 8.2: Parameter settings

	1st experiment	**2nd experiment**
Total budgets (B_{max})	500 (func. eval.)	1 (hour)
DACOpt parameters		
- Number of candidates (K)	10	10
- Initial sample size per candidate (B_{init})	5	5
- DAC variants used		
DAC-HB	✓	✓
DAC-HH	✓	✓
DAC-SB	✓	✓
DAC-SH	✓	✓
HyperOpt parameters		
- Initial sample size per	50	50
BO4ML parameters		
- Number of candidates (K)	10	10
- Initial sample size per candidate (B_{init})	5	5

The first experiment used a budget of 500 function evaluations. The 5-fold cross-validation approach and the averaged geometric mean values over 10 repetitions were reported. The selected classification algorithms were not grouped together. The resampling techniques were grouped by a hierarchical graph as shown on the right-hand side of Figure 8.2, following the suggestion in [48].

In the second experiment, all experiments are based on 10 runs with different random seeds, and a time limit of 1 hour. The performance evaluation of a single configuration is limited to 10 minutes with 4-folds cross-validation on training data, i.e., the evaluation of a fold is allowed to take 150 seconds. The evaluation of a configuration will be aborted and returned to zero if any of the folds have an error, for example, infeasible configuration or timeout. The average accuracy values for the test data over 10 runs were reported. Finally, the selected algorithms used a hierarchical tree of similarity of algorithms[8].

Reproducibility and Open Science: The implementation of the proposed methods is published in a git-repository[9] and PyPi-repository[10]. The experiment scripts

[8] based on the hierarchy used in [151] and discussed in [46].

[9] `https://github.com/ECOLE-ITN/NguyenIEEEAccess2022`

[10] `https://pypi.org/project/DACOpt`

for the reproducibility of the reported results are provided in a git-repository[11].

8.5 Results and Discussion

In this section, we report and discuss the results obtained from the two experimental setups introduced above. Generally speaking, we target three goals: (1) to compare the performance of our two contesting procedures in terms of *number of function evaluations* and *wall-time limit*; (2) to compare the performance of BO with and without the proposed contesting procedures; (3) to compare those against the current state-of-the-art AutoML frameworks.

The first experiment's results are provided in Table 8.3, and the second in Table 8.4. Both tables highlight the highest performance for the corresponding dataset/task in **bold**. According to the Wilcoxon signed-rank test, the method that performs significantly worse than the best with $\alpha = 0.05$ is underlined. Two extra rows at the end of the corresponding table display additional summaries. The first extra row shows the number of times each scenario got the highest value over tested datasets/tasks. The last extra row indicates the number of times each approach was significantly better than the other in a group.

For each tested case, the method that achieved the highest performance was counted as winning, provided that its performance was significantly better than that of all other methods. The method that performed significantly worse than the best was counted as a loss. They are considered equal if there is no significant difference in performance between the two methods. The method is counted as performing well if it either achieves the best performance or is not significantly worse than the best-found method in the corresponding case.

[11] `https://github.com/ECOLE-ITN/NguyenIEEEAccess2022/tree/main/Experiments`

Table 8.3: Average geometric mean (rounded to 4 decimals) based on six approaches, i.e., DAC-HB, DAC-HH, DAC-SB, DAC-SH, BO4ML and Hyperopt, over 10 repetitions for the 44 examined datasets, ordered by increasing imbalance ratio (#IR) value.

Dataset	#IR	DAC-HB	DAC-HH	DAC-SB	DAC-SH	BO4ML	Hyperopt
glass1	1.82	0.8015	0.8004	**0.804**	0.7945	0.7922	0.7968
ecoli-0_vs_1	1.86	0.9864	0.9864	0.9864	0.9864	**0.9868**	0.9864
wisconsin	1.86	0.9813	0.9816	0.9813	0.9814	0.9814	**0.9819**
pima	1.87	0.769	**0.7725**	0.768	0.7719	0.7694	0.7705
iris0	2.0	**1**	**1**	**1**	**1**	**1**	**1**
glass0	2.06	0.876	**0.8777**	0.8736	0.8757	0.8736	0.8745
yeast1	2.46	0.7332	0.7333	0.7322	0.7321	**0.7335**	0.7325
haberman	2.78	**0.7057**	0.701	0.7023	0.6974	0.6968	0.7012
vehicle2	2.88	0.9903	0.991	0.9902	0.9898	**0.9912**	0.991
vehicle1	2.9	**0.8709**	0.8707	0.8512	0.8445	0.862	0.8701
vehicle3	2.99	0.84	0.8476	0.8166	0.8202	**0.848**	0.8461
glass-0-1-2-3_vs_4-5-6	3.2	0.9571	0.9568	0.9545	**0.9572**	0.9562	0.9514
vehicle0	3.25	0.9865	**0.9868**	0.9837	0.9837	0.9864	0.9855
ecoli1	3.36	0.9034	0.9047	0.9036	0.9029	0.9031	**0.9047**
new-thyroid1	5.14	0.9975	**0.9986**	0.9966	0.9972	0.9972	0.9978
new-thyroid2	5.14	0.9975	**0.9978**	0.9972	0.9972	0.9975	0.9978
ecoli2	5.46	0.9375	**0.9375**	0.9365	0.9362	0.9361	0.9358
segment0	6.02	**0.9993**	0.9993	0.9992	0.9992	0.9991	0.9993
glass6	6.38	0.952	**0.9547**	0.9516	0.9503	0.9489	0.9524
yeast3	8.1	0.9436	**0.9437**	0.9422	0.942	0.9428	0.9425
ecoli3	8.6	0.9075	0.9079	0.907	0.9072	0.9054	**0.9091**
page-blocks0	8.79	0.9471	0.9467	0.9468	0.9463	**0.948**	0.9472
yeast-2_vs_4	9.08	0.9535	0.952	0.9533	0.951	**0.9538**	0.9533
yeast-0-5-6-7-9_vs_4	9.35	0.8145	**0.8258**	0.8146	0.8169	0.8238	0.8195
vowel0	9.98	**0.9628**	0.9569	0.9598	0.9554	0.9564	0.9555
glass-0-1-6_vs_2	10.29	**0.8515**	0.8424	0.845	0.8359	0.8436	0.8342
glass2	11.59	0.8593	0.8546	**0.8601**	0.8534	0.8578	0.856
shuttle-c0-vs-c4	13.87	**1**	**1**	**1**	**1**	**1**	**1**
yeast-1_vs_7	14.3	0.8017	**0.8043**	0.8003	0.8026	0.8017	0.8001
glass4	15.46	0.9355	**0.9372**	0.9291	0.935	0.9192	0.9334
ecoli4	15.8	**0.9743**	0.9709	0.9701	0.9637	0.9661	0.9698
page-blocks-1-3_vs_4	15.86	**0.9944**	0.9889	0.9929	0.9882	0.9901	0.99
abalone9-18	16.4	**0.8951**	0.8864	0.8834	0.8846	0.8873	0.8838
glass-0-1-6_vs_5	19.44	0.9655	0.9535	0.9588	0.9597	**0.9681**	0.9644
shuttle-c2-vs-c4	20.5	**1**	**1**	**1**	**1**	**1**	**1**
yeast-1-4-5-8_vs_7	22.1	**0.7166**	0.7037	0.7155	0.7011	0.6979	0.7011
glass5	22.78	**0.9716**	0.9699	0.9659	0.96	0.9625	0.96
yeast-2_vs_8	23.1	0.8242	0.8259	0.8135	0.8117	**0.828**	0.8254
yeast4	28.1	0.8794	**0.8812**	0.8694	0.8726	0.8708	0.8773
yeast-1-2-8-9_vs_7	30.57	0.7515	**0.7546**	0.7416	0.7459	0.7391	0.7429
yeast5	32.73	0.9801	**0.9806**	0.9795	0.9796	0.9801	0.9798
ecoli-0-1-3-7_vs_2-6	39.14	0.866	0.8799	0.8892	0.9035	0.9034	**0.9057**
yeast6	41.4	0.9007	**0.9018**	0.8994	0.9003	0.897	0.9004
abalone19	129.44	**0.8059**	0.8031	0.8022	0.8003	0.8049	0.8021
Cases achieved the highest values		14	18	5	4	11	7
Significant wins over other approaches		10	8	1	1	4	0

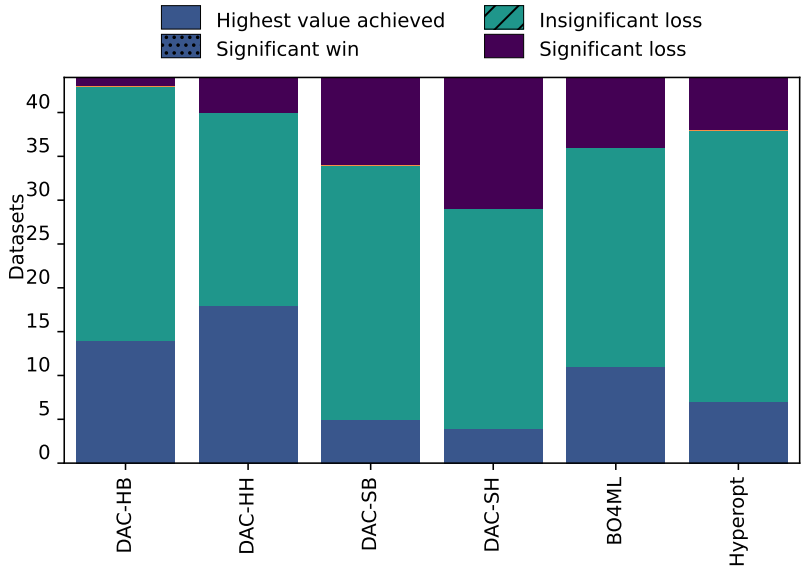


Figure 8.3: Overview of the results over 10 repetitions for the 44 binary imbalanced benchmark datasets.

8.5.1 First experiment results

The results of the first experiment are presented in Table 8.3 to illustrate the performance between 2 BO variants based on TPE surrogate model with and without proposed contesting procedures using 2 elimination criteria – highest performance and statistical test procedure, i.e., DAC-HB, DAC-SB, DAC-HH, DAC-SH compared to BO4ML and Hyperopt. Additionally, these results are summarized in Figure 8.3. This figure is based on the average geometric mean over a 5-fold cross-validation over 44 imbalanced binary benchmark datasets. We make the following observations:

- Comparing two methods that use the highest performance as the elimination criteria (highest value-based contest), DAC-HH achieved the highest performance more times than DAC-HB (18 vs. 14). However, DAC-HB significantly won on more tested cases than DAC-HH. Additionally, DAC-HB loses on fewer cases than DAC-HB (1 vs. 5).
- Compared to the contesting procedures that used statistical tests as the elimination criteria (statistical-based contest), two methods, i.e., DAC-SH and DAC-SB, achieved similar performance.

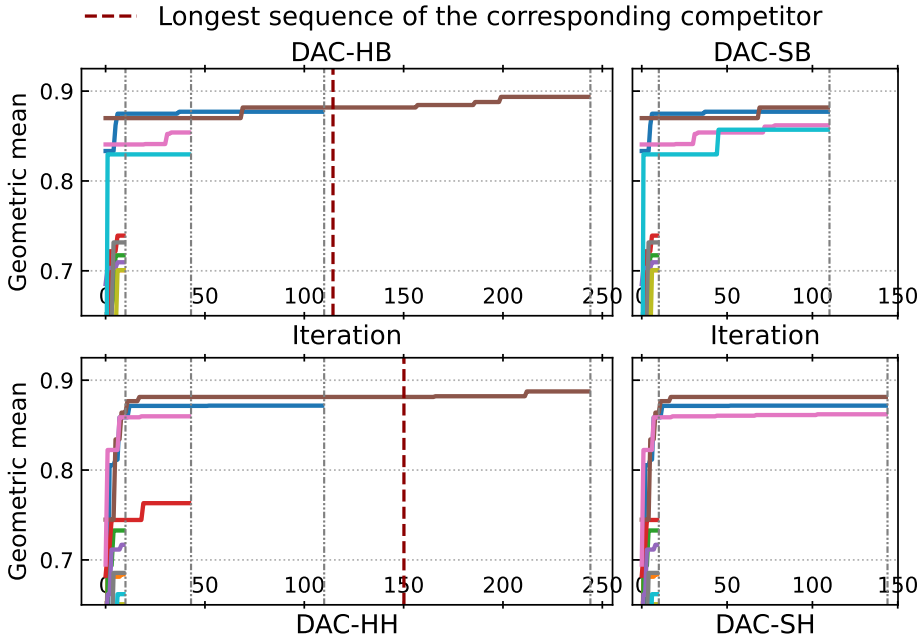


Figure 8.4: Illustration of the contesting process on dataset *abalone9-18*. This figure shows the optimization convergence plots of DAC-HB (top-left), DAC-SB (top-right), DAC-HH (bottom-left) and DAC-SH (bottom-right) approaches. All approaches are initialized with the same random seed. The colors represent BO processes on sub-spaces.

- Overall, DAC-HB performs well in most of the tested cases. More precisely, over 44 tested dataset, DAC-HB loses only on dataset *pima*, where DAC-HH is the winner.

Lastly, another point worth mentioning is that we expected the statistical-based approaches, i.e., DAC-SB and DAC-SH, to perform better than the highest-based approaches, i.e., DAC-HB and DAC-HH. However, the experimental results contradict our assumptions. To investigate their optimizing behavior, we plot a single run of these approaches on the dataset *abalone9-18* in Figure 8.4. The two plots on the left show the convergence behavior of the highest-based contests and the statistical-based contests are shown on the right. All approaches used a total budget of 500 function evaluations, and the search space was split into 10 sub-spaces. The colors represent BO processes on sub-spaces. The dashed-grey vertical lines indicate a contest round cutoff point, i.e., the end of the round where the elimination function is called. The extra dashed-red vertical line on the two left

plots shows the most extended sequence of the corresponding underlying optimizer. The statistical-based approach maintained more candidates throughout the contest than the highest-based approach. Consequently, the best candidate was found late with less budget than the best candidate in the competitor approach.

Table 8.4: Average accuracy (rounded to 5 decimals) over 10 repetitions for the 73 OpenML datasets, ordered by #Task ID. The first fourth columns after "Dataset" shows our experimental results, i.e., 4 variants of the contesting procedure. The remaining columns contain results obtained by other AutoML frameworks according to results from Chapter 7 and [22].

OpenML IDs	DACOpt contesting procedure				Chapter 7		[22]					
#TaskID	DAC-HB	DAC-HH	DAC-SB	DAC-SH	BO4ML	Hyperopt	Auto sklearn	Random search	HP sklearn	TPOT	ATM	H2O
3	**0.99802**	0.99666	0.99802	0.99666	0.99656	0.9951	0.98986	0.99062	0.99051	0.99431	0.99326	0.99426
12	**0.98617**	0.98383	0.98617	0.98383	0.98417	0.98117	0.97767	0.97633	0.94758	0.97333	0.98178	0.97433
15	0.98095	0.97524	0.98095	0.97524	0.97952	0.97048	0.96875	0.95873	0.96	0.96571	**0.98474**	0.96286
23	0.57376	0.57805	0.57376	0.57805	0.57285	0.55158	0.54638	0.53262	0.53047	0.55882	**0.581**	0.53733
24	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0.99993**	**1**	**1**	**1**	0.99848
29	0.88647	0.87778	0.88647	0.87778	0.88744	0.86522	0.87289	0.85507	0.85956	0.86377	**0.89133**	0.86184
31	**0.767**	0.74533	0.767	0.74533	0.766	0.72733	0.73433	0.724	0.70121	0.744	0.76578	0.74867
41	0.94927	0.93659	0.94927	0.93659	**0.95171**	0.92878	0.91954	0.91911	0.92585	0.92732	0.94504	0.93122
53	0.86299	0.8374	0.86299	0.8374	**0.86457**	0.83858	0.82008	0.81969	0.75787	0.81811	0.81522	0.82717
2079	**0.69864**	0.69548	0.69864	0.69548	0.69502	0.66018	0.63886	0.6267	0.64072	0.65566	0.6419	0.6557
3021	0.99134	0.9909	0.99134	0.9909	**0.99152**	0.98737	0.98288	0.9855	0.97438	0.98746		0.98419
3543	**1**	**1**	**1**	**1**	**1**	**1**	0.99019	0.99081	0.99404	0.99091	**1**	0.97967
3560	0.2375	0.22417	0.2375	0.22417	0.23042	0.21125	0.20365	0.20382	0.19139	0.20833	**0.27028**	0.19542
3561	0.69901	0.67228	0.69901	0.67228	0.63119	0.64752	0.65687	0.64563	0.63762	0.66832	**0.71221**	0.71089
3904	**0.82535**	0.82171	0.82535	0.82171	0.82404	0.81393	0.81344	0.81126	0.80998	0.8181	0.821	0.74819
3917	0.87172	0.86572	0.87172	0.86572	**0.87393**	0.85972	0.85118	0.8534	0.84044	0.86019	0.86856	0.80869
3945	**0.98325**	0.98285	0.98325	0.98285	0.98323	0.98197	0.98244	0.98228	0.98189	0.98182		0.96555
3946	0.92863	0.92809	0.92863	0.92809	**0.92901**	0.92624	0.92725	0.92586	0.92599	0.92624		0.78802
3948	0.9506	**0.95137**	0.9506	0.95137	0.94345	0.94116	0.95094	0.9503	0.95068	0.95085		0.93415
7592	0.86906	0.86527	0.86906	0.86527	0.86251	0.85769	0.86938	0.87013	0.86727	**0.87089**	0.85448	0.86656
7593	0.87738	0.93199	0.87738	0.93199	0.70278	0.80902	**0.96395**	0.89143	0.95227	0.94542	0.6639	0.92908
9910	**0.80595**	0.80062	0.80595	0.80062	0.80107	0.78073	0.7889	0.77762	0.77798	0.80249	0.77087	0.80044
9952	**0.91726**	0.91196	0.91726	0.91196	0.91319	0.90826	0.89716	0.89205	0.89273	0.9045	0.89963	0.89205
9955	**0.69562**	0.6775	0.69562	0.6775	0.67167	0.65146	0.65172	0.62795	0.54667	0.61146	0.61097	0.56435
9977	0.97132	0.97098	0.97132	0.97098	0.96525	0.95924	0.96903	0.96656	0.96891	0.97026	0.96055	**0.97146**
9981	**0.96235**	0.95432	0.96235	0.95432	0.95093	0.94228	0.94167	0.93117	0.94012	0.94784	0.96049	0.95216
9985	**0.61983**	0.61133	0.61983	0.61133	0.61029	0.59853	0.59695	0.58601	0.58293	0.61291	0.60272	0.61656
10101	0.80622	0.79644	0.80622	0.79644	0.80667	0.76578	0.76667	0.77778	0.78044	0.78711	**0.81956**	0.73378
14952	**0.97422**	0.97272	0.97422	0.97272	0.97094	0.96623	0.9659	0.96244	0.96964	0.96913	0.96464	0.9716
14954	0.83395	0.82037	0.83395	0.82037	**0.83642**	0.81111	0.79012	0.76173	0.76667	0.81009	0.81701	0.78333
14965	0.90695	0.90625	0.90695	0.90625	0.90307	0.90007	0.90447	0.90398	0.90451	**0.90705**	0.89957	0.9006
14967	**1**	**1**	**1**	**1**	**1**	**1**	0.98265	0.99841	0.97131	**1**		**1**
14968	**0.84259**	0.84074	0.84259	0.84074	0.8358	0.80432	0.77353	0.77058	0.75823	0.81173	0.79155	0.8
14969	0.66722	0.6812	0.66722	0.6812	0.64001	0.61864	0.67733	0.65004	0.67272	0.67586	0.66217	**0.70165**
34538	**1**	**1**	**1**	**1**	**1**	0.99907	**1**	0.99907	0.99983	**1**	**1**	**1**
34539	0.94915	0.94702	0.94915	0.94702	0.94825	0.94557	0.94761	0.94444	0.9475	0.94891	0.94606	**0.95114**
125920	0.61533	0.6	0.61533	0.6	0.63133	0.562	0.56667	0.55556	0.56844	0.56867	**0.66978**	0.584
146195	0.82042	0.82628	0.82042	0.82628	0.77358	0.77321	0.82109	0.79628	0.82886	0.84123	0.77698	**0.865**

continued on the next page

Table 8.4: Average accuracy (rounded to 5 decimals) over 10 repetitions for the 73 OpenML datasets, ordered by #Task ID. The first fourth columns after "Dataset" show the experimental results, i.e., 4 variants of the contesting procedure. The remaining columns contain results obtained by other AutoML frameworks according to results presented in Chapter 7 and [22]. – continued from previous page

OpenML IDs	DACOpt contesting procedure				Chapter 7		[22]					
#TaskID	DAC-HB	DAC-HH	DAC-SB	DAC-SH	BO4ML	Hyperopt	Auto sklearn	Random search	HP sklearn	TPOT	ATM	H2O
146212	**0.9999**	0.99989	0.9999	0.99989	0.99965	0.99945	0.99978	0.99968	0.99253	0.99974	0.99955	0.99987
146606	0.7115	0.71645	0.7115	0.71645	0.70605	0.69761	**0.72296**	0.7193	0.70743	0.72031	0.67135	0.71281
146607	**0.87383**	0.8708	0.87383	0.8708	0.86611	0.85871	0.86291	0.86225	0.86661	0.86392	0.86128	0.84968
146800	**1**	**1**	**1**	**1**	0.99969	0.99321	0.99043	0.99506	0.9638	0.99506	**1**	0.99551
146817	**0.80892**	0.79966	0.80892	0.79966	0.80497	0.78216	0.78268	0.76364	0.75955	0.79091	0.76415	0.78062
146818	0.88889	0.88261	0.88889	0.88261	0.88647	0.85845	0.87053	0.85556	0.86913	0.86184	**0.8905**	0.87633
146819	0.96728	0.96605	0.96728	0.96605	0.95802	0.93951	0.94074	0.92407	0.92593	0.94547	**0.96975**	0.93642
146820	**0.98864**	0.98747	0.98864	0.98747	0.97355	0.97906	0.98612	0.98581	0.95289	0.9854	0.98657	0.98574
146821	**0.99981**	0.99884	**0.99981**	0.99884	0.99441	0.98748	0.97264	0.97958	0.98786	0.99422	0.96763	0.99191
146822	**0.94603**	0.94055	0.94603	0.94055	0.94473	0.93189	0.93088	0.93333	0.90664	0.94055	0.92564	0.94185
146824	0.9835	0.98233	0.9835	0.98233	**0.98433**	0.98117	0.97783	0.97367	0.98121	0.96883	0.9775	0.976
146825	0.85193	0.86744	0.85193	0.86744	0.843	0.83891	**0.87844**	0.8445	0.8506	0.78089	0.82114	0.87341
167119	0.84892	0.86674	0.84892	0.86674	0.84647	0.83956	0.86775	0.85378	0.88691	0.88735	0.8754	**0.90047**
167120	**0.52457**	0.52385	0.52457	0.52385	0.52257	0.52134	0.51926	0.51939	0.52033	0.52082	0.51941	0.50635
167121	0.7812	0.89001	0.7812	0.89001	0.86652	0.7491	0.74009	0.02169	0.86438		**0.8947**	0.5822
167124	0.33284	**0.40956**	0.33284	0.40956	0.39675	0.37813			0.32093	0.29429	0.32001	0.36389
167125	0.97856	**0.97876**	0.97856	0.97876	0.97713	0.97033	0.97774	0.97114	0.97358	0.97398	0.969	
167140	0.96475	0.96287	0.96475	0.96287	0.96485	0.95397	0.95962	0.95889	0.96109	0.95931	0.95282	**0.96904**
167141	**0.9636**	0.96133	0.9636	0.96133	0.96273	0.95367	0.9562	0.95313	0.94533	0.96	0.95007	0.9537
168329	**0.33594**	0.32871	0.33594	0.32871	0.3169	0.29294	0.30692	0.29566	0.28741	0.33576	0.32108	
168330	0.68921	0.70161	0.68921	0.70161	0.68479	0.6667	**0.71814**	0.69273	0.68494	0.69642	0.63788	0.71786
168331	0.6319	0.65658	0.6319	0.65658	0.60445	0.5952	0.66933	0.63762	0.65451	0.65075	**0.6794**	0.67841
168332	0.39963	0.44613	0.39963	0.44613	0.42507	0.38497	**0.44843**	0.39922	0.34203		0.35252	
168335	0.93435	0.93889	0.93435	0.93889	0.92248	0.91035	0.94334	0.92891	0.87477	0.9385	0.90234	**0.94604**
168337	0.7521	0.81453	0.7521	0.81453	0.78205	0.72707	0.64227		0.74347	0.72548	0.66063	**0.81928**
168338	0.9701	**0.99552**	0.9701	0.99552	0.98303	0.98035	0.74757	0.75042	0.82518	0.98495	0.90729	0.95625
168868	0.99183	0.99318	0.99183	0.99318	0.98985	0.989	0.99287	0.99137	0.9936	0.99339	0.97097	**0.99369**
168908	**0.75609**	0.75166	0.75609	0.75166	0.73659	0.72565	0.74754	0.73081	0.7163	0.72645	0.72169	0.72811
168909	0.96513	0.983	0.96513	0.983	0.9504	0.94437	**0.98357**	0.94793	0.97243	0.96254	0.95391	0.96988
168910	0.69563	0.69308	0.69563	0.69308	0.67565	0.66177	0.70255	0.67395	0.69104	0.68336	0.67357	**0.71752**
168911	0.82578	0.81942	0.82578	0.81942	**0.83259**	0.80748	0.82009	0.80603	0.80078	0.82366	0.79911	0.80906
168912	0.95033	0.95273	0.95033	0.95273	**0.95709**	0.94655	0.93921	0.94753	0.94675	0.95533	0.93476	0.9251
189354	0.65866	0.6634	0.65866	0.6634	0.64626	0.63957	0.66665	0.59845	0.6508	**0.66895**	0.63671	0.61266
189355	0.81678	**0.82433**	0.81678	0.82433	0.73916	0.68112			0.77971		0.38666	
189356	0.65874	0.66576	0.65874	0.66576	0.6481	0.64737	0.68314	0.66709	0.66694	0.6611	**0.80064**	0.64798
Cases with the highest values achieved	28	10	6	5	13	3	8	1	1	6	16	12
Significant wins over other approaches	26	8	4	3	11	2	6	0	0	4	13	11

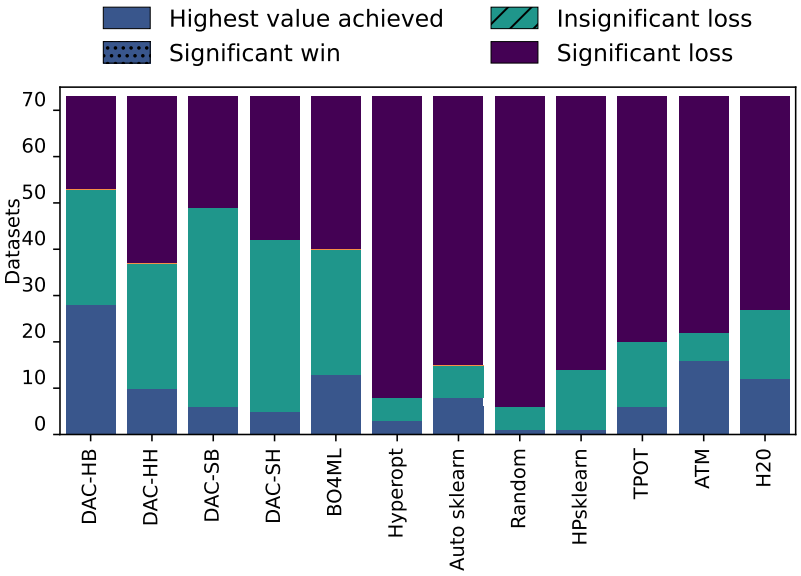


Figure 8.5: Overview of the results over 10 repetitions for the 73 AutoML benchmark datasets.

8.5.2 Second experimental results

In this experiment, we compare all approaches used in the first experiment to the current state-of-the-art AutoML frameworks, i.e., Auto-sklearn-SMAC (Auto-sklearn) and Auto-sklearn-Random search (Random), HPsklearn ([42], TPOT [43], ATM [90], H2O [89]), based on the results obtained by [22]. The detailed results of the second tested scenarios are presented in Table 8.4. We note that entries with missing values in the last 6 columns indicate arbitrary fails reported by [22]. Additionally, the results of the second experiment are summarized in Figure 8.5. This figure is based on the accuracy of the test dataset over 10 repetitions to show the performance differences between the two BO variant-based TPE surrogate models, namely BO4ML and Hyperopt, to compare both with and without the proposed contesting procedure, as well as with two elimination criteria, namely the highest value and a statistical procedure (see Table 8.1).

- First, when comparing the three approaches that use Hyperopt as the underlying optimizer, i.e., DAC-HH, DAC-SH, and Hyperopt, we observed that both proposed contesting procedures won on more tested cases than Hyperopt. More precisely, DAC-HH, DAC-SH, and Hyperopt significantly

outperformed others in $8, 3, and$ 2 cases, respectively. However, in these tested cases of Hyperopt, it is never significantly better than both DAC-SH and DAC-HH; DAC-SH and DAC-HH are not significantly different. In contrast, correspondingly, DAC-HH and DAC-SH significantly outperform Hyperopt in 5 and 1 cases. Therefore, we can conclude that (1) both contesting procedures significantly improve the performance of BO, (2) DAC-HH won against Hyperopt in more cases compared to DAC-SH.

- Secondly, we analyze the results of three approaches that use BO4ML as the underlying optimizer, i.e., DAC-HB, DAC-SB, and BO4ML. We observe that: (1) all three approaches performed well on $73\%, 67\%$, and 55% tested cases, respectively; (2) DAC-HB achieved the highest performances on most of the tested cases, followed by BO4ML and DAC-SB. In 11 cases where BO4ML significantly outperformed others, it was not significantly better than any of the competitors in this comparison. DAC-SB was significantly better than BO4ML on 1 tested case, i.e., task 146821, but it never won DAC-HB. In comparison, DAC-HB outperformed DAC-SH and BO4ML on 3 and 7 cases, correspondingly.

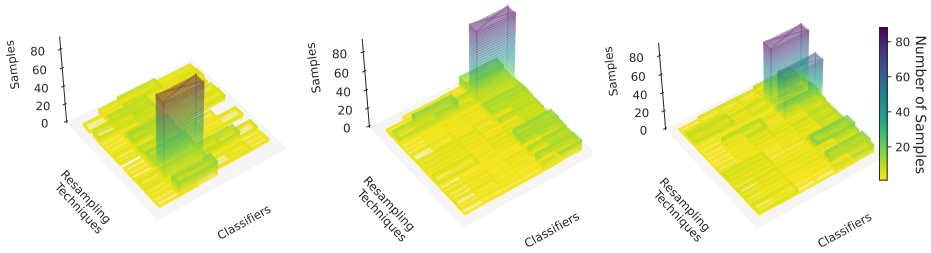
- Comparing the results of 8 approaches using the search space of Auto-sklearn, i.e., our four approaches, BO4ML, Hyperopt, Auto-sklearn, and Random search, we can observe that: First, all BO-based approaches performed better than random searches over all tested cases. Random search achieves the highest result in 1 case (#ID:24), in which all competitors perform equally (no win). Second, it can be seen that DAC-HB won in most tested cases, followed by BO4ML, DAC-HH, Auto-Sklearn, DAC-SH, DAC-SH, Hyperopt, and Random search. We conclude that the proposed approach clearly improves the efficiency of BO in solving AutoML optimization problems. This finding may be explained by the fact that the HPO-based approach does not consider the relationship between algorithms under operators; thus, it requires more resources to cover a large and complex search space in this experiment. In contrast, by grouping similar algorithms together and splitting the original search space into smaller independent subspaces, the proposed approach better utilizes the given budget. Consequently, the search space can be covered within a relatively small budget, and the most promising subspace can be identified early. As a result, resources are efficiently distributed. Additionally, BO is known to perform better for low-dimensional problems [31], [241], [242].

Our approach transfers the original high-dimensional problem of AutoML into multiple low-dimensional problems, thus improving the performance of BO.

- Additionally, when comparing all contesting variants, it can be seen that DAC-HB won on more tested cases than others. The contesting procedure based on the highest performance, i.e., DAC-HH, DAC-HB, won on more cases than those based on the statistical procedure, i.e., DAC-SH, DAC-SB. This finding may be explained by the fact that executing a statistical method adds to the overall computational cost of the procedure. As a result, the contesting technique that used statistical procedures examined fewer configurations in the same amount of time as the others.

- Finally, Figure 8.5 shows that the proposed contesting procedures performed well on up to 73% and at least 53% of all tested cases, when compared to Random Search -8% of all cases, Hyperopt - 11% of all cases, AutoSklearn - 21% of all cases, TPOT - 27% of all cases, ATM - 30% of all cases and H2O - 37% of all cases.

8.6 Application on Surface Defect Classification in Steel Manufacturing



(a) TPE (b) BO4ML (c) DACOpt

Figure 8.6: Illustration on the distribution of 500 samples across a search space of 5 classifiers and 21 resampling techniques of the three optimization algorithms, namely TPE, BO4ML, DACOpt. In this run, different approaches explore specific combinations of algorithms (cells in the figure) to find the combination that can achieve the best performance. BO4ML and DACOpt cover more combinations of algorithms (cells in the figure) than TPE. Specifically, TPE, BO4ML, and DACOpt have 35, 13, and 8 combinations with no samples (white cells color in the figure), respectively.

In this section, we present an application of DACOpt in a real-world application for surface classification in steel manufacturing. This multi-class imbalance problem was introduced in Chapter 6. We have two main objectives in this section:

- Firstly, we aim to enhance the performance of the current classification system used for surface defect detection at our industry partner, TATA, by applying AutoML optimization. This study will use a standard performance metric, i.e., geometric mean (micro), as the objective function. This is different from Chapter 6.
- Secondly, we apply our new method, DACOpt, to the real-world application for surface classification in steel manufacturing. As presented in Section 8.5.1 and Section 8.5.2, the experimental results show that DAC-HB won on more test cases compared to other variants of DACOpt. Thus, we use DAC-HB as the mere variant of DACOpt in this study. Additionally, we aim to investigate the efficiency of DACOpt as compared to BO4ML (Chapter 7) and TPE [158]. The difference between the three optimization approaches is illustrated in Figure 8.6. The illustration shows the sample distribution of 500 samples across the search space of 5 classification and 21 resampling algorithms. The TPE algorithm is shown in Fig. 8.6a, while the our two optimization algorithms are BO4ML (Fig. 8.6b) and DACOpt (Fig. 8.6c). The height and color of each bar represent the number of samples. The white cell shows unexplored algorithm combinations. The figure shows that TPE has more unexplored combinations than the other algorithms, indicating that some ML algorithm combinations were never explored.

In the remainder of this section, we present the experimental setup (Section 8.6.1) followed by the experimental results and discussion (Section 8.6.2).

8.6.1 Experimental setup

As mentioned, this study reuses the experimental setup introduced in Chapter 6[12], which includes the search space (Section 6.3), experimental procedure (Section 6.3.2), and datasets (Section 6.3.1). As a reminder, the search space includes five classification algorithms, namely Support Vector Machines (SVM), Random Forest (RF), k-nearest Neighbors (KNN), Decision Tree (DT), and Logistic Regression (LR), along with 3 commonly used multiple-class classification

[12]The experiment scripts for the reproducibility of the reported results are provided in a git-repository :`https://github.com/anh05/AutoML-Multiclass-Imbalanced`

techniques (Multi-class direct classification (Direct), One-vs-One (OvO), and One-vs-Rest (OvR)). We direct interested readers to Chapter 6 (Section 6.2) for a detailed discussion of the relevant background. Additionally, there are 21 options for resampling techniques, including the option of not using any resampler, leading to a total of 84 hyperparameters in the search space. We have improved the practicality of selected resampling techniques in tackling multi-class imbalanced problems by introducing a hyperparameter called `sampling strategy`. It offers a range of values including {`majority/minority`[13], `not minority`, `not majority`, `all`, `auto`}. We use the geometric mean micro ($\mathrm{GM}_{\mathrm{micro}}$) as the objective function to maximize. For M classes $(\mathrm{A}, \mathrm{B}, \ldots, \mathrm{M})$ in a multi-class classification problem, we calculate the $\mathrm{GM}_{\mathrm{micro}}$ as:

$$\begin{aligned} \mathrm{GM}_{\mathrm{micro}} &= \sqrt{\mathrm{Specificity}_{\mathrm{micro}} \times \mathrm{Sensitivity}_{\mathrm{micro}}} \\ &= \sqrt{\frac{\sum_{i=1}^{\mathrm{M}} \mathrm{TN}_i}{\sum_{i=1}^{\mathrm{M}} \mathrm{TN}_i + \sum_{i=1}^{\mathrm{M}} \mathrm{FP}_i} \times \frac{\sum_{i=1}^{\mathrm{M}} \mathrm{TP}_i}{\sum_{i=1}^{\mathrm{M}} \mathrm{TP}_i + \sum_{i=1}^{\mathrm{M}} \mathrm{FN}_i}} \end{aligned} \tag{8.5}$$

where $\mathrm{TP}_i, \mathrm{TN}_i, \mathrm{FP}_i, \mathrm{FN}_i$ denote the number of true positives, true negatives, false positives and false negatives samples in class $i \in \mathrm{M}$, respectively.

For this particular study, we conducted 9 optimization processes for a given dataset using different classification strategies and optimizers. To be specific, for each optimization approach, we set up three independent experiments, each representing a different approach– One vs. Rest (OvR), One vs. One (OvO), and Direct classification (Direct) strategies. Therefore, we had $3 \times 3 = 9$ optimization processes for a dataset. All 9 optimization processes have a budget of 500 for function evaluations. Additionally, our experiments aim to compare the *current classification system* (current system) used by our industry partner[14]. We use the same training and test datasets as the current system for a fair comparison. The current system executes 10 times on each of the tested datasets. For each execution, the considered dataset is randomly split into training (80%) and test (20%) sets. Those train/test sets are exported to use in our experiments, i.e., we have $2 \times 10 = 20$ different train/test sets in total.

[13] `majority` is an option for under resampling, `minority` is for over/combine resampling techniques

[14] For reasons of confidentiality, since proprietary software of a supplier is used by the industrial partner, no details about the algorithmic approach taken by the currently used system are available.

8.6.2 Experimental results and discussion

In this section, we present our findings and insights. We have summarized the experimental results in Table 8.5 to showcase the performance differences between three optimization algorithms, namely TPE, BO4ML, and DACOpt. For each optimizer, we have provided their optimization performance with the use of three classification strategies, namely multi-class direct classification (Direct), One vs. One (OvO), and One vs. Rest (OvR). This results in 9 experimental outcomes for each dataset. We have compared these results against the classification approach used in the current system. The highest performance for each dataset is highlighted in bold. The methods performing significantly worse than the best according to the Wilcoxon signed-rank test with $\alpha = 0.05$ are underlined. Additionally, the

Table 8.5: Average geometric mean (micro), rounded to 5 decimals over 10 repetitions for the 2 datasets. Boldface highlights the best-performing method per dataset and underline indicates results that are significantly different from the best method in that group according to a Wilcoxon signed-rank test ($p < 0.05$).

Dataset	Current	TPE			BO4ML			DACOpt		
	system	Direct	OvO	OvR	Direct	OvO	OvR	Direct	OvO	OvR
Top side	0.87269	0.92068	0.91957	0.92394	0.92076	0.91982	0.92474	0.92115	0.92069	**0.92554**
Bottom side	0.86064	0.94085	0.94175	0.94146	0.94096	0.9198	0.94165	0.94137	**0.94247**	0.94176

distribution of geometric mean micro over 10 repetitions for the two tested datasets, is visualized in Fig. 8.7. Each box plot represents 10 repetitions. The horizontal inner line shows the median. The whiskers show the lowest and the highest observed value[15]. The color dots show the observed values, and the dots outside the whisker represent the outliers. The box covers the first to the third quantiles. The results allow the following insights:

- According to the results of the Wilcoxon signed-rank test, our experimental approaches significantly outperform the current approach used at our industry partner (current system). Additionally, from Fig. 8.7, the median and whiskers of all optimization approaches on three classification scenarios are higher than those of the current system. In other words, our procedure has successfully enhanced the performance of the current classification system used by our industry partner.

[15]The whisker scale is set as 1.5.

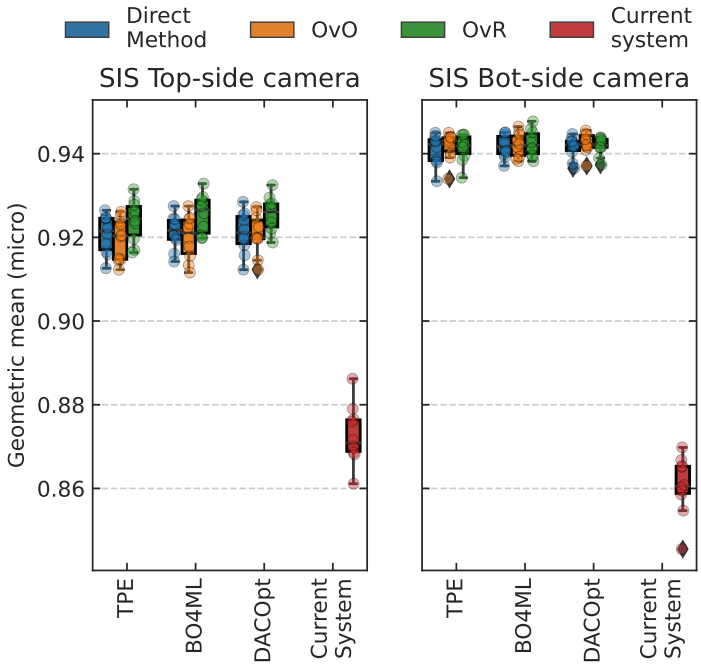


Figure 8.7: Box plots showing the distribution of classification results for two examined datasets.

- Overall, DACOpt achieved the best classification performance on both datasets that were examined. Specifically, for the SIS top-side dataset, the DACOpt approach using the OvR classification strategy outperformed TPE in all experiments. It also significantly outperformed BO4ML in two cases of direct classification and OvO strategies. For the SIS bottom-side dataset, the DACOpt method with the OvR classification strategy also achieved the highest result. It significantly outperformed TPE with the direct classification strategy.

- We conducted a Friedman's Test on all three strategies to determine the most effective classification strategy among direct classification, OvO, and OvR when used with AutoML optimization approaches. Surprisingly, the results showed no significant difference in the average GM (micro) with a p-value of 0.13169. It is surprising because the decomposition approach is the most commonly recommended for dealing with multi-class problems from literature, as it converts the multi-class problem into multiple binary-class

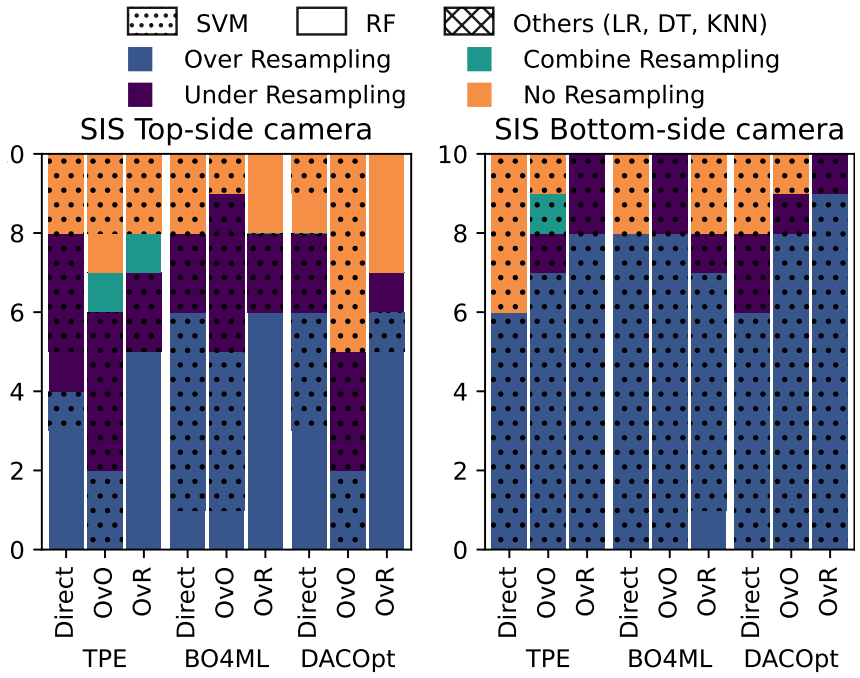


Figure 8.8: Illustration of the different combinations of resampling and classification algorithms generated by the best ML pipeline resulting from the optimization processes conducted by TPE, BO4ML, and DACOpt. Within each of the 9 optimization approaches (3 for each method), 3 distinct classification strategies, namely Direct classification (Direct), One-vs-One (OvO), and One-vs-Rest (OvR). The optimization process is repeated ten times on two examined datasets.

problems, which makes the classifier work more efficiently. However, our finding indicates that applying AutoML optimization performs similarly for three classification strategies.

We have presented Fig. 8.8 to investigate the final combination of choices for resampling and classification algorithms of all optimization methods. Our findings indicate that SVM and RF are the algorithms most frequently selected, while other classification algorithms have not been chosen in any of the test cases. SVM wins in 61% and 99% of cases for the SIS Top-side and SIS Bottom-side datasets, respectively. RF obtains 39% wins on the SIS Top-side dataset, but only one win in the SIS Bottom-side dataset by BO4ML with the OvR classification strategy. Regarding the usefulness of resampling techniques, 81% of the runs yield the highest results using some form of resampling technique. Over resampling, under

Figure 8.9: Comparison of all optimization algorithms compared to each other using Nemenyi test with a 5% significance level.

resampling, and combined resampling obtain 109 (61%), 34 (19%), and 3 (2%) wins over (2 datasets $\times$ 3 classification strategies $\times$ 3 optimizers) $\times$ 10 repetitions = 180 runs. At the same time, 34 (19%) runs yield the highest performance without using any resampling techniques.

Based on the results of Friedman's test in average GM (micro), we found significant differences among all optimization approaches, with a p-value of 9.8e-4. As a follow-up, we performed post hoc multiple comparison tests using the Nemenyi test at a significance level of 0.05, as shown in Fig. 8.9. Approaches with a distance greater than CD[16] are considered significantly different. Upon analyzing the figure, we can conclude that DACOpt outperforms all competitors, while BO4ML and TPE perform similarly.

Based on the results of our experiment, we have arrived at **four main conclusions**:

1. Our experiments demonstrated that AutoML optimization approaches significantly improved classification performance compared to the current system.

2. In addition, we found that our new approach, DACOpt, outperforms the two competitors, i.e., BO4ML and TPE. Therefore, we recommend the use of the DACOpt method for AutoML optimization.

3. Resampling techniques are recommended to deal with multi-class imbalanced problems, as 81% of runs yield the best performance using them.

4. Lastly, our findings indicate that applying AutoML optimization to direct classification yields similar performance compared to using it in OvO and OvR strategies. Therefore, we highly recommend using the direct classification approach to address similar problems, as it is much more cost-effective than OvO and OvR.

[16]Critical Difference, here CD=1.353

It is worth noting that our study focuses solely on enhancing the current classification component of the surface defect detection system used by our industry partner. Moving forward, we plan to expand our research to cover the entire surface defect detection system, including image processing and feature extraction. Additionally, we may look into utilizing deep learning and convolutional neural networks.

8.7 Conclusions and future work

In this study, we proposed a novel contesting procedure for the AutoML optimization problem, namely DACOpt, which is complementary to the existing BO approaches. DACOpt partitions the AutoML search space into multiple relatively small sub-spaces based on algorithm similarity and budget constraints. Next, BO approaches are employed to optimize these sub-spaces independently. The budget is then adaptively distributed to the search area based on the performance of the corresponding BO processes. The proposed contesting procedure has two different variants of elimination criteria – based on the highest performance and a statistical procedure. Additionally, we presented a partly parallel approach to using BO to address AutoML optimization problems with provably theoretical guarantees. Two extensive experiments on a total of 117 benchmark datasets demonstrated the superiority of our novel contesting procedures over the current state-of-the-art AutoML optimization approaches. Additionally, an experiment was conducted on surface defect classification in steel manufacturing. It was concluded that our proposed approach significantly improves BO's performance. In future studies, we intend to incorporate meta-learning approaches to identify search areas that may perform well in the early stages. Finally, the scope of this study was limited to the AutoML optimization problem; we plan to extend our research to Neural Architecture Search (NAS) problems in the future.

CHAPTER 9

Conclusions

Automated Machine Learning (AutoML) has emerged as an effective approach to streamline the machine learning development process for real-world applications. This thesis provides a comprehensive exploration of AutoML and its various important aspects. The research conducted in this thesis encompasses several significant contributions to the field.

To conclude the thesis, in this chapter, we provide a summary of the content of this thesis. We begin by highlighting the main contributions made in this research and addressing the research questions posed. This is presented in Section 9.1, where we summarize the key findings and achievements of this thesis.

Section 9.2 is dedicated to discussing potential avenues for future research. We outline these potential directions and offer insights into how they can contribute to advancing the state-of-the-art in AutoML.

9.1 Summary

Chapter 1 begins with a brief introduction and motivation to Automated Machine Learning (AutoML). It is attempted to give fundamental formulations of optimization approaches to address the AutoML optimization problem (i.e., *HPO-based* and *ML pipeline optimization* approach). Furthermore, the chapter outlines the significant contributions of the thesis within the AutoML domain, highlights the specific research questions and provides an overview of each chapter's main contributions and methodologies. Lastly, this chapter also provides a roadmap for the reader and establishes the overall organization and structure of the thesis.

Chapter 2 discusses the life cycle of machine learning (ML) application development and provides a comprehensive literature review on key technical aspects of ML. It explores the different stages involved in the ML application development process, from data preparation, data preprocessing, ML pipeline

optimization, model evaluation and application development. The chapter also discusses the challenges and considerations specific to each stage, highlighting the importance of an efficient and automated approach to streamline the ML development process. Building upon the ML life cycle, the discussion in this chapter expands to encompass various functions and techniques offered by existing AutoML products. The chapter also examines the underlying techniques and methodologies employed by the existing AutoML products, shedding light on the potential implications for ML application development.

Chapter 3 focuses on providing a thorough literature review of AutoML optimization approaches, with a specific emphasis on adapting hyperparameter optimization (HPO) approaches to address the AutoML optimization problem. The chapter begins by presenting a comprehensive review of common black-box optimization approaches employed in AutoML, including Grid search, Random search, and Bayesian optimization. These approaches serve as the foundation for understanding the evolution and advancements in AutoML optimization. Follow up with a literature review of optimization algorithms of the two well-known multi-fidelity approaches – racing procedure and bandit learning.

Throughout the literature review, the chapter provides insights into the theoretical foundations, algorithmic frameworks, and practical implementations of these optimization approaches in the context of AutoML. This chapter aims to establish the theoretical background and set the stage for the original contributions and research conducted in the subsequent chapters of the thesis.

Chapter 4 provides two benchmark experiments repeatedly used in this thesis. Detailed information about the two experiments is introduced in this chapter, where each includes a search space, examined datasets and a detailed experiment procedure. The first benchmark experiment is designed for class-imbalanced problems, where a set of 44 binary class-imbalanced benchmark datasets and a suitable search space are provided. The datasets are taken from the Keel collection [186]. The search space includes 21 resampling techniques and five classification algorithms, where algorithms and their hyperparameters are selected based on related work recommendations. Moreover, this experimental design is successfully used in Chapters [5-8] that demonstrate its usefulness for imbalanced classification problems. This study also answers **RQ1** on how to handle class imbalance problems.

The second set of benchmark experiments is identical to [22], which includes 73 classification benchmark datasets from OpenML [189], and the search space

is extracted from Auto-sklearn [45]. This set of experiments is later used in the works of chapter 7 and chapter 8.

Chapter 5 investigates the effectiveness of the commonly used HPO optimization approaches – Random search and Bayesian optimization (BO), to solve the CASH problem for the binary class imbalance classification problems. This investigation is to answer **RQ2** in which approach is most effective for optimizing the ML pipeline in addressing class imbalance problems. Besides, we are particularly interested in how CASH improved classification performance compared to using static default hyperparameters (i.e., try all combinations of resampler and classifier without hyperparameter tuning). The key findings from this indicate the following. We observed that CASH optimization significantly improves classification performance compared to using static default hyperparameters. Moreover, the experimental results indicate that BO is always the best method found. This study concludes that BO outperforms other approaches in answering **RQ2**. Additionally, 98% of runs yield the best performance by fine-tuned ML pipelines that contain a resampling and a classification algorithm, demonstrating the experimental design's usefulness as well as supporting our answer to **RQ1**.

Chapter 6 presents our new method to compute performance for classification problems where the distribution between classes is imbalanced and has unequal class importance. In ML, the assessment method is critical in evaluating an ML model's performance and choosing the suitable ML model that works well on the given problem. Many performance metrics, such as F1, geometric mean, recall, and precision, can be used in class imbalanced learning. However, these methods do not consider the unequal class problem.

Built on top of standard performance metrics, we propose a new performance metric incorporating unequal class importance into the standard performance metrics. More precisely, we propose to compute the classification performance based on the new penalized confusion matrix based on the actual confusion matrix and a user-defined penalty matrix. The domain experts define the penalty matrix, which contains the penalty values between actual and predicted classes. The penalized confusion matrix is then produced by multiplying every element of the actual confusion matrix with the corresponding element in the penalty matrix. The final classification performance is later calculated via the new penalized confusion matrix instead of the actual one, as usual. Our approach solves **RQ3** in handling the dual problem of class imbalanced and unequal importance between classes. Moreover, we also investigate the correlation of assessment values between the

new method and the standard one. Our finding indicates that the performance computed by the new approach and the standard metrics are strongly correlated. That is to say, our approach incorporates unequal class importance into the standard performance metrics and does not change their purposes (e.g., metrics for imbalanced problems).

Chapter 7 researches improving BO performance for AutoML optimization problems via maximizing coverage of search space already during the initial sampling of BO to characterize the response surface more accurately. The initial sampling step is the first step of BO to utilize the first response surface. It is typically restricted to a small budget since the effectiveness of BO becomes evident mainly in the later stages of optimization when it learns to produce better configuration. Considering how to improve coverage over AutoML search space within a limited budget, we propose the novel *combination-based sampling approach* for the initial sampling stage of BO. Our proposed initial sampling approach is as follows. We first attempt to group algorithms with similar technical behaviours as in the literature, where one can represent the rest of the group. Then, we reallocate the sampling budget to explore the potential of similarities between algorithms within the group: sampling fewer of the same algorithms frees up the budget to be distributed to other (different) algorithms, thus optimizing the coverage of algorithm-hyperparameter search space.

To investigate the potential of our proposed approach in AutoML optimization scenarios, we compare the performance of BO with and without using it on the two AutoML benchmark experiments (see Chapter 4) over 117 classification problems. The key findings from this indicate the following. In the first set of benchmark experiments, we evaluated them on two scenarios of initial sample sizes – 20 and 50 iterations. With our improvement, the performance of BO significantly improved in several cases and did not significantly worst in any tested cases. In the second experiment, we also compared the two experimented BO approaches against the other six well-known AutoML products (i.e., Auto-sklearn (BO and Random search), H2O, TPOT, ATM and Hyperopt-sklearn). The experimental results indicate that the BO using our initial sampling approach produces the best results and significantly outperforms others in more cases than all compared approaches. In contrast, the experimented BO without our improvement does not significantly win in any tested cases.

In conclusion, the experimental results answer **RQ4**: Our approach, which maximizes the coverage of algorithm-hyperparameter search space during the initial

sampling stage of BO, clearly improved BO performance in solving the AutoML optimization problems.

Chapter 8 introduces a novel contesting procedure algorithm, **D**ivide **A**nd **C**onquer **Opt**imization (DACOpt), to efficiently solve AutoML optimization. Motivated by the fact that BO performs better for low-dimensional problems [31], while AutoML is typically high-dimensional mixed variables. This causes BO to be less robust for solving AutoML. To limit this issue, we first partition the search space into a reasonable number of sub-spaces based on algorithm and budget constraints. Then, multiple BO performs on every sub-space independently (i.e., sub-process) to optimize BO performance. Due to their independence, this also allows them to be explored in parallel. Additionally, we adopt the ideas of the two well-known multi-fidelity approaches (i.e., bandit learning and racing procedure) into our procedure to eliminate ineffective sub-processes to free up the budget to be distributed to the better one, thus optimizing the budget usage.

Generally speaking, the contesting procedure is complementary to the existing BO approaches to handle their limit when optimizing the AutoML problems. The proposed approach is constructed of three main components:

- The *splitter function* to partition the search space into multiple sub-spaces. Then, an existing BO approach is employed to optimize those sub-spaces independently, leading to a corresponding number of optimization sub-processes.

- The *elimination function* decides to stop poorly performing sub-spaces (i.e., terminate the corresponding sub-processes). This function also addresses **RQ5**, which concerns when we should stop tuning in a particular area (sub-space) of the search space. To summarize, we can stop tuning in an area when it demonstrates significantly worse results compared to the most promising area.

- The *controller function* allocates budgets adaptively for tuning each sub-space based on the performance of the corresponding optimization processes. This function provides an answer to **RQ6** on how to allocate computational resources over the search space. In simple terms, we conduct multiple competitions over the available resources, which are calculated based on the input computation resources and the number of areas. After each race, several areas are eliminated, and the remaining areas share the saved resources of the race. This ensures that the most effective area stays longer and has the most tuning resources.

Lastly, we compare the performance of BO with and without our contesting procedures on the two AutoML benchmark experiments Chapter 7. We use Hyperopt [153] and BO4ML [15] as the underlying BO. Besides, the proposed procedure has variants of the elimination function (i.e., it adopts bandit learning and racing procedure). Thus, we have four variants in total. The key findings from this indicate the following.

- First, in both experimental scenarios, both BOs with our contesting procedure significantly perform better than one without using it. Overall, the contesting procedure that used BO4ML achieved more best results than any other competitors. This finding again demonstrated the effectiveness of BO4ML (i.e., also providing an answer to **RQ4**) for solving AutoML problems.
- Comparing the two adopted elimination functions, the one that adopted bandit learning performs better than the competitor. Thus, we recommend that researchers use the bandit learning variant for AutoML problems.
- Lastly, we also compared the proposed approaches against the six state-of-the-art AutoML products (i.e., Auto-sklearn (BO and Random search), H2O, TPOT, ATM and Hyperopt-sklearn), in the second scenario. The proposed contesting procedure produces the best results, performs well (i.e., either achieved the best performance or not significantly worse than the best-found method) and significantly outperforms others in more cases than all compared approaches (i.e., our best contest procedure produces 28 highest, 53 well-performing and 26 significantly outperforms values over 73 tested cases).

In conclusion, the experimental results demonstrated the effectiveness of our new contesting procedures in solving AutoML optimization problems. They notably enhanced BO's performance, and the AutoML, using our proposed contesting procedure as an optimizer, won over the current state-of-the-art AutoML tools, such as H2O, Auto-sklearn, ATM, Hyperopt-sklearn, and TPOT, in a wide range of benchmark tests.

9.2 Future work

This thesis has focused on conducting research in the field of Automated Machine Learning (AutoML) and has presented significant achievements and insights. However, numerous future works within AutoML still require further investigation

and development. In this section, we outline some potential directions for future research and provide insights into how they can contribute to advancing the state-of-the-art in AutoML. In the following discussion, we will explore potential future research directions to extend the work presented in this thesis.

9.2.1 Combination-based sampling

The combination-based sampling approach, introduced in Chapter 7, aims to maximize the coverage of algorithm-hyperparameter samples in the search space. This approach improves the accuracy of characterizing the response surface during the initial sampling stage of the historical-based optimization approach. Here are several potential future research directions for extending this work:

- The combination-based sampling is initially incorporated for Tree Parzen Estimators [153]). It would be interesting to exploit the potential of applying the proposed sampling approach to other BO variants (e.g., SMAC [25], MIPEGO [38]) and historical-based approaches, such as evolutionary strategies [108] and genetic algorithms [103]. By extending the evaluation of our sampling approach to different optimization techniques, we can assess its generalizability and potential for improving the performance of various optimization algorithms. This exploration may provide a comprehensive understanding of how the initial sampling effect to the later stage of historical-based optimization approaches.

- It would be interesting to explore how the combination-based sampling approach, introduced in our work, performs during the sequential sampling stage of BO. By maximizing the coverage of algorithm-hyperparameter samples in the search space, the candidate configurations proposed at each iteration may exhibit better exploration and exploitation properties. This could potentially lead to more efficient and effective sampling. This research direction can potentially enhance the optimization process, improve the quality of candidate configurations, and provide valuable guidelines for selecting suitable sampling strategies for historical-based optimization approaches to address AutoML problems.

9.2.2 Contesting procedures

The contesting procedures algorithm introduced in Chapter 8, serves as an efficient approach for addressing AutoML optimization problems. This algorithm offers

promising results, but several potential avenues for future research can further enhance and expand on this work. The following are some of the possible research directions discussed:

- As part of the proposed contest procedure, the algorithmic hierarchy attribute allows organizing *the choice of algorithms* in the search space in a hierarchical manner, with one algorithm in a branch potentially representing multiple other algorithms. Due to the large set of possible combinations of algorithms over operators (i.e., functional algorithms in ML pipeline structures), it is not possible to try every combination in practice. Hence, the algorithmic hierarchy is a realistic way to identify ineffective combinations. However, we acknowledge that the algorithmic hierarchy is currently constructed based on the experiences of practitioners in the field. As such, the resulting structure of the search space may not be optimized. It would be highly advantageous to explore methods to optimize the hierarchical structure using historical data from experiments. Advanced techniques such as clustering methods can be applied to automatically identify patterns and relationships among algorithms, enabling the creation of an optimized algorithmic hierarchy. This data-driven approach would enhance the efficiency and effectiveness of AutoML by incorporating empirical insights in a systematic and automated manner, reducing the reliance on manual construction.

- Moreover, it is worth considering the integration of meta-learning approaches to identify promising search areas at an early stage. By leveraging meta-learning techniques, we can leverage prior knowledge or learned patterns to guide the optimizer. This can help accelerate the optimization process by focusing on areas that have shown promising performance in previous similar tasks. We believe that the incorporation of meta-learning can enhance the efficiency and effectiveness of the optimizer, enabling them to make informed decisions and prioritize exploration in the search areas likely to yield favorable results.

9.2.3 Benchmarking methods and application domains

Indeed, an important concern in evaluating AutoML optimization algorithms is the inconvenience and high cost associated with using a wide range of **real datasets**. This process can be time-consuming, resource-intensive, and financially burdensome. While benchmarking with **synthetic test functions** is a common strategy in

optimization studies due to their closed-form representation and efficient evaluation, the existing synthetic test functions are not suitable for AutoML benchmarking [22]. This is primarily because they do not simulate categorical hyperparameters, pure categorical hyperparameters, or algorithm choice hyperparameters, nor do they account for structured search spaces. An interesting direction for future research is the development of *synthetic test functions specifically designed for AutoML benchmarking*. These test functions should accurately represent the complexities and characteristics of real-world AutoML problems, including the incorporation of categorical hyperparameters and structured search spaces. By defining such synthetic test functions, researchers and practitioners can evaluate and compare AutoML optimization algorithms in a more controlled, cost-effective, and efficient manner. This would enable the systematic analysis of algorithm performance and facilitate advancements in the field of AutoML.

Empirical Performance Models (EPMs), as an alternative to synthetic test functions, introduced by [249]–[252], provide a surrogate representation of the response surface of a specific performance metric. These models aim to capture the empirical performance characteristics of a real dataset, offering a means to theoretically evaluate algorithms in AutoML scenarios. It is important to note that existing empirical performance models (EPMs) are not tailored for AutoML scenarios, as mentioned in previous studies [22], [253]. This raises the need for further investigation and assessment of its suitability within the AutoML context. Exploring the applicability of EPMs in AutoML can lead to valuable advancements in the evaluation and benchmarking of AutoML algorithms.

These future works to AutoML benchmarking approaches would drive progress and advancements in the field, promoting the development of robust and efficient AutoML solutions for a wide range of practical applications.

Lastly, in this thesis, we have conducted comprehensive investigations into AutoML, focusing primarily on supervised machine learning problems. However, it would be beneficial to expand the discussion to include other domains of machine learning as well. Exploring studies and research in **unsupervised machine learning**, **reinforcement learning**, and **deep learning** can provide valuable insights and a broader understanding of the topic. This would allow for a more holistic analysis of AutoML's applications and effectiveness across various ML domains.

APPENDIX A

Appendix

A.1 Additional information for the first experiment

A.1.1 Parameter setting

In this section, we present detailed information of the hyperparameters used in the classification algorithms[1] and resampling algorithms[2] that we used in our experiment. The entire Python code project can be found at https://github.com/ECOLE-ITN/CASH4IMBALANCE.

The detailed information on hyperparameters is listed in Table A.1, Table A.2, and Table A.3.

A.2 Imbalance datasets

In this section, we present 44 examined datasets taken from the KEEL repository [186] in Table A.4. For each dataset, we include the *Imbalance Ratio* (IR), which is the ratio of the number of majority class instances to that of minority class instances.

[1] All classification algorithms are implemented in the Python package SCIKIT-LEARN(version 0.23.2) [151].

[2] All resampling algorithms are implemented in the Python package IMBALANCED-LEARN(version 0.7.0) [48]

Table A.1: Hyperparameters of Classification Algorithms

Algorithm	Hyperparameter	Range
Support Vector Machines	max_iter	10000
	cache_size	700 (Megabyte)
	probability	[True, False]
	C	[0.5^5,100]
	kernel	[linear, rbf, poly, sigmoid]
	gamma	[auto, value, scale]
	gamma_value	[3.1e-05,8]
	coef0	[-1.0, 1.0]
	degree	[2, 5]
	shrinking	[True, False]
	tol	[1e-05, 1e-01]
Random Forest	n_estimators	[1, 150]
	criterion	[gini, entropy]
	max_features	[1, sqrt, log2, None]
	min_samples_split	[2, 20]
	min_samples_leaf	[1, 20]
	class_weight	[balanced, balanced_subsample, None]
	bootstrap	[True, False]
K-Nearest Neighbors	n_neighbors	[1, 51]
	weights	[uniform, distance]
	algorithm	[auto, ball_tree, kd_tree, brute]
	p	[0, 20]
	metric	· p = 0 → metric = chebyshev · p = 1 → metric = manhattan · p = 2 → metric = euclidean · p > 2 → metric = minkowski
Decision Tree	criterion	[gini, entropy]
	max_depth	[2,20]
	max_features	[1, sqrt, log2, None]
	min_samples_split	[2, 20]
	min_samples_leaf	[1, 20]
Logistic Regression	C	[1, 150]
	criterion	[0.5^5, 100]
	tol	[1e-05, 1e-01]
	l1_ratio	[1e-09, 1]
	(penalty, solver)	[(l1, liblinear), (l1, saga), (l2, lbfgs), (l2, newton-cg), (l2, liblinear), (l2, sag), (l2, saga), (elasticnet, saga), (none, newton-cg), (none, lbfgs), (none, sag), (none, saga)]

Table A.2: Hyperparameters of Resampling techniques (part I)

Group.	Hyperparameter	Range
Under resampling	**CondensedNearestNeighbour**	
	sampling_strategy	*default*
	n_neighbors	[1, 50]
	n_seeds_S	[1, 50]
	EditedNearestNeighbours	
	sampling_strategy	*default*
	n_neighbors	[1, 20]
	kind_sel	[all, mode]
	RepeatedEditedNearestNeighbours	
	sampling_strategy	*default*
	n_neighbors	[1, 20]
	kind_sel	[all, mode]
	AllKNN	
	sampling_strategy	*default*
	n_neighbors	[1, 20]
	kind_sel	[all, mode]
	allow_minority	[True, False]
	InstanceHardnessThreshold	
	sampling_strategy	*default*
	estimator	none, decision-tree, adaboost knn, linear-svm, gradient-boosting
	cv	[2, 10]
	OneSidedSelection	
	sampling_strategy	*default*
	n_neighbors	[1, 20]
	n_seeds_S	[1, 20]
	RandomUnderSampler	
	sampling_strategy	*default*
	replacement	[True, False]
	TomekLinks	
	sampling_strategy	*default*
	NearMiss	
	sampling_strategy	*default*
	version	[1,3]
	n_neighbors	[1, 20]
	n_neighbors_ver3	[1, 20]
	NeighbourhoodCleaningRule	
	sampling_strategy	*default*
	n_neighbors	[1, 20]
	threshold_cleaning	[0.0, 1.0]
	ClusterCentroids	
	sampling_strategy	*default*
	estimator	[KMeans, MiniBatchKMeans]
	voting	[hard, soft]

Table A.3: Hyperparameters of Resampling techniques (part II)

Group.	Hyperparameter	Range
Combine resampling	**SMOTENN**	
	sampling_strategy	*default*
	SMOTETomek	
	sampling_strategy	*default*
Over resampling	**SMOTE**	
	k_neighbors	[1, 10]
	sampling_strategy	*default*
	BorderlineSMOTE	
	sampling_strategy	*default*
	k_neighbors	[1, 10]
	m_neighbors	[1, 10]
	kind	[borderline1, borderline2]
	SMOTENC	
	sampling_strategy	*default*
	categorical_features	True
	k_neighbors	[1, 10]
	SVMSMOTE	
	sampling_strategy	*default*
	k_neighbors	[1, 10]
	m_neighbors	[1, 10]
	out_step	[0.0, 1.0]
	KMeansSMOTE	
	sampling_strategy	*default*
	k_neighbors	[1, 10]
	cluster_balance_threshold	[1e-2, 1]
	ADASYN	
	sampling_strategy	*default*
	n_neighbors	[1, 10]
	RandomOverSampler	
	sampling_strategy	*default*

Table A.4: The number of positive, negative classes, attributes (#Att) and the imbalance ratio (IR) of the KEEL Datasets, ordered by increasing IR value.

Data Sets	# Negative	# Positive	#Att	**IR**
glass1	138	76	9	1.82
ecoli-0_vs_1	77	143	7	1.86
wisconsin	444	239	9	1.86
pima	500	268	8	1.87
iris0	100	50	4	2
glass0	144	70	9	2.06
yeast1	1055	429	8	2.46
haberman	225	81	3	2.78
vehicle2	628	218	18	2.88
vehicle1	629	217	18	2.9
vehicle3	634	212	18	2.99
glass-0-1-2-3_vs_4-5-6	163	51	9	3.2
vehicle0	647	199	18	3.25
ecoli1	259	77	7	3.36
new-thyroid1	180	35	5	5.14
new-thyroid2	180	35	5	5.14
ecoli2	284	52	7	5.46
segment0	1979	329	19	6.02
glass6	185	29	9	6.38
yeast3	1321	163	8	8.1
ecoli3	301	35	7	8.6
page-blocks0	4913	559	10	8.79
yeast-2_vs_4	463	51	8	9.08
yeast-0-5-6-7-9_vs_4	477	51	8	9.35
vowel0	898	90	13	9.98
glass-0-1-6_vs_2	175	17	9	10.29
glass2	197	17	9	11.59
shuttle-c0-vs-c4	1706	123	9	13.87
yeast-1_vs_7	429	30	7	14.3
glass4	201	13	9	15.46
ecoli4	316	20	7	15.8
page-blocks-1-3_vs_4	444	28	10	15.86
abalone9-18	689	42	8	16.4
glass-0-1-6_vs_5	175	9	9	19.44
shuttle-c2-vs-c4	123	6	9	20.5
yeast-1-4-5-8_vs_7	663	30	8	22.1
glass5	205	9	9	22.78
yeast-2_vs_8	462	20	8	23.1
yeast4	1433	51	8	28.1
yeast-1-2-8-9_vs_7	917	30	8	30.57
yeast5	1440	44	8	32.73
ecoli-0-1-3-7_vs_2-6	274	7	7	39.14
yeast6	1449	35	8	41.4
abalone19	4142	32	8	129.44

A.3 Additional information for the second experiment

A.3.1 Datasets used in the second experiment

In this section, we present 73 examined datasets taken from OpenML repository [204] in Table A.5. For each task, we include the *OpenML ID* (#task id) and the corresponding dataset (#ID, Name) together with the number of classes (#Class), the number of instances(#Sample), the number of features for one instance (Total features, number of numeric and categorical features), the number of missing values (#Missing values), the number of instances with missing value (#Incomplete sample).

A.3.2 Search space

Detailed information on the operators, algorithms and hyperparameters used in the second experiment is given in Table A.6. The search space was extracted from Auto-sklearn [45].

Table A.5: List of 73 datasets used in our second experiment, ordered by increasing OpenML ID.

Task ID	Data set		#Class	#Sample	Features			#Missing	#Incomplete
	#ID	Name			# Total	#Numeric	#Cate	Values	sample
3	3	kr-vs-kp	2	37	3196	0	37	0	0
12	12	mfeat-factors	10	217	2000	216	1	0	0
15	15	breast-w	2	10	699	9	1	16	16
23	23	cmc	3	10	1473	2	8	0	0
24	24	mushroom	2	23	8124	0	23	2480	2480
29	29	credit-approval	2	16	690	6	10	67	37
31	31	credit-g	2	21	1000	7	14	0	0
3021	38	sick	2	30	3772	7	23	6064	3772
41	42	soybean	19	36	683	0	36	2337	121
53	54	vehicle	4	19	846	18	1	0	0
2079	188	eucalyptus	5	20	736	14	6	448	95
3543	451	irish	2	6	500	2	4	32	32
3560	469	analcatdata_dmf	6	5	797	0	5	0	0
3561	470	profb	2	10	672	5	5	1200	666
3904	1053	jm1	2	22	10885	21	1	25	5
3917	1067	kc1	2	22	2109	21	1	0	0
3945	1111	KDDCup09_appete	2	231	50000	192	39	8024152	50000
3946	1112	KDDCup09_churn	2	231	50000	192	39	8024152	50000
3948	1114	KDDCup09_upsell	2	231	50000	192	39	8024152	50000
189354	1169	airlines	2	8	539383	3	5	0	0
14965	1461	bank-marketing	2	17	45211	7	10	0	0
10101	1464	blood-transfusi	2	5	748	4	1	0	0
9981	1468	cnae-9	9	857	1080	856	1	0	0
9985	1475	first-order-the	6	52	6118	51	1	0	0
9977	1486	nomao	2	119	34465	89	30	0	0
9952	1489	phoneme	2	6	5404	5	1	0	0
9955	1492	one-hundred-pla	100	65	1600	64	1	0	0
7592	1590	adult	2	15	48842	6	9	6465	3620
7593	1596	covertype	7	55	581012	10	45	0	0
9910	4134	Bioresponse	2	1777	3751	1776	1	0	0
34539	4135	Amazon_employee	2	10	32769	0	10	0	0
14952	4534	PhishingWebsite	2	31	11055	0	31	0	0
14969	4538	GesturePhaseSeg	5	33	9873	32	1	0	0
34538	4550	MiceProtein	8	82	1080	77	5	1396	528
14954	6332	cylinder-bands	2	40	540	18	22	999	263
14968	6332	cylinder-bands	2	40	540	18	22	999	263
14967	23380	cjs	6	35	2796	32	3	68100	2795
125920	23381	dresses-sales	2	13	500	1	12	835	401
146606	23512	higgs	2	29	98050	28	1	9	1

continued on the next page

Table A.5: List of 73 datasets used in our second experiment, ordered by increasing OpenML ID. – continued from the previous page

Task ID	Data set		#Class	#Sample	Features			#Missing	#Incomplete
	#ID	Name			# Total	#Numeric	#Cate	Values	sample
167120	23517	numerai28.6	2	22	96320	21	1	0	0
146607	40536	SpeedDating	2	123	8378	59	64	18372	7330
146195	40668	connect-4	3	43	67557	0	43	0	0
167140	40670	dna	3	181	3186	0	181	0	0
146212	40685	shuttle	7	10	58000	9	1	0	0
167141	40701	churn	2	21	5000	16	5	0	0
167121	40923	Devnagari-Scrip	46	1025	92000	1024	1	0	0
167124	40927	CIFAR_10	10	3073	60000	3072	1	0	0
146800	40966	MiceProtein	8	82	1080	77	5	1396	528
146821	40975	car	4	7	1728	0	7	0	0
167125	40978	Internet-Advert	2	1559	3279	3	1556	0	0
146824	40979	mfeat-pixel	10	241	2000	240	1	0	0
146818	40981	Australian	2	15	690	6	9	0	0
146817	40982	steel-plates-fa	7	28	1941	27	1	0	0
146820	40983	wilt	2	6	4839	5	1	0	0
146822	40984	segment	7	20	2310	19	1	0	0
146819	40994	climate-model-s	2	21	540	20	1	0	0
146825	40996	Fashion-MNIST	10	785	70000	784	1	0	0
167119	41027	jungle_chess_2p	3	7	44819	6	1	0	0
168868	41138	APSFailure	2	171	76000	170	1	1078695	75244
168908	41142	christine	2	1637	5418	1599	38	0	0
168911	41143	jasmine	2	145	2984	8	137	0	0
168912	41146	sylvine	2	21	5124	20	1	0	0
189356	41147	albert	2	79	425240	26	53	2734000	425159
168335	41150	MiniBooNE	2	51	130064	50	1	0	0
168337	41159	guillermo	2	4297	20000	4296	1	0	0
168338	41161	riccardo	2	4297	20000	4296	1	0	0
168909	41163	dilbert	5	2001	10000	2000	1	0	0
168910	41164	fabert	7	801	8237	800	1	0	0
168332	41165	robert	10	7201	10000	7200	1	0	0
168331	41166	volkert	10	181	58310	180	1	0	0
189355	41167	dionis	355	61	416188	60	1	0	0
168330	41168	jannis	4	55	83733	54	1	0	0
168329	41169	helena	100	28	65196	27	1	0	0

Table A.6: List of hyperparameters extracted from Auto-sklearn [39] for Classification problems

Operator	Hyperparameter	Type	Range
Classification	**AdaboostClassifier** (*sklearn.ensemble.AdaBoostClassifier*)		
	n_estimators	Integer	[50,500]
	learning_rate	Float	[0.01,2]
	algorithm	Categorical	[SAMME.R, SAMME]
	max_depth	Integer	[1,10]
	BernoulliNB (*sklearn.naive_bayes.BernoulliNB*)		
	alpha	Float	[0.01,100]
	fit_prior	Categorical	[True, False]
	DecisionTree (*sklearn.tree.DecisionTreeClassifier*)		
	criterion	Categorical	[gini, entropy]
	max_depth_factor	Float	[0,2]
	min_samples_split	Integer	[2,20]
	min_samples_leaf	Integer	[1,20]
	min_weight_fraction_le	Constant	[0]
	max_features	Constant	[1]
	max_leaf_nodes	Constant	[None]
	min_impurity_decrease	Constant	[0]
	ExtraTreesClassifier (*sklearn.ensemble.ExtraTreesClassifier*)		
	criterion	Categorical	[gini, entropy]
	max_features	Float	[0,1]
	max_depth_factor	Constant	[None]
	min_samples_split	Integer	[2,20]
	min_samples_leaf	Integer	[1,20]
	min_weight_fraction_le	Constant	[0]
	max_leaf_nodes	Constant	[None]
	min_impurity_decrease	Constant	[0]
	bootstrap	Categorical	[True, False]
	GaussianNB (*sklearn.naive_bayes.GaussianNB*)		
	Default hyperparameters		
	GradientBoostingClassifier (*sklearn.ensemble.HistGradientBoostingClassifier*)		
	loss	Constant	[auto]
	learning_rate	Float	[0.01,1]
	min_samples_leaf	Integer	[1,200]
	max_depth	Constant	[None]
	max_leaf_nodes	Integer	[3,2047]
	l2_regularization	Float	[0.0000000001,1]
	early_stop	Categorical	[off, train, valid]
	tol	Constant	[0.0000001]
	scoring	Constant	[loss]
	n_iter_no_change	Integer	[1,20]
	validation_fraction	Float	[0.01,0.4]
	KNearestNeighborsClassifier (*sklearn.neighbors.KNeighborsClassifier*)		
	n_neighbors	Integer	[1,100]
	weights	Categorical	[uniform, distance]
	p	Categorical	[1, 2]
	LDA (*sklearn.discriminant_analysis.LinearDiscriminantAnalysis*)		
	shrinkage	Categorical	[None, auto, manual]
	shrinkage_factor	Float	[0,1]
	n_components	Integer	[1,250]
	tol	Float	[0.00001,0.1]

continued on the next page

Table A.6: List of hyperparameters extracted from Auto-sklearn [39] for Classification problems – continued from the previous page

Operator	Hyperparameter	Type	Range
Classification	**LinearSVC** (*sklearn.svm.LinearSVC*)		
	penalty	Categorical	[l1, l2]
	loss	Categorical	[hinge, squared_hinge]
	dual	Constant	[False]
	tol	Float	[0.00001,0.1]
	multi_class	Constant	[ovr]
	fit_intercept	Constant	[True]
	LibSVM_SVC (*sklearn.svm.SVC*)		
	kernel	Categorical	[rbf, poly, sigmoid]
	degree	Integer	[2,5]
	gamma	Float	[0.000031,8]
	coef0	Float	[-1,1]
	shrinking	Categorical	[True, False]
	tol	Float	[0.00001,0.1]
	max_iter	Constant	[-1]
	MultinomialNB(*sklearn.naive_bayes.MultinomialNB*)		
	alpha	Float	[0.01,100]
	fit_prior	Categorical	[True, False]
	PassiveAggressive (*from sklearn.linear_model import PassiveAggressiveClassifier*)		
	loss	Categorical	[hinge, squared_hinge]
	tol	Float	[0.00001,0.1]
	average	Categorical	[True, False]
	QDA (*sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis*)		
	reg_param	Float	[0,1]
	RandomForest (*sklearn.ensemble.RandomForestClassifier*)		
	criterion	Categorical	[gini, entropy]
	max_features	Float	[0,1]
	max_depth	Constant	[None]
	min_samples_split	Integer	[2,20]
	min_samples_leaf	Integer	[1,20]
	min_weight_fraction_le	Constant	[0]
	max_leaf_nodes	Constant	[None]
	min_impurity_decrease	Constant	[0]
	bootstrap	Categorical	[True, False]
	SGD (*sklearn.linear_model.SGDClassifier*)		
	loss	Categorical	[hinge, log, modified_huber]
	penalty	Categorical	[l1, l2, elasticnet]
	alpha	Float	[0.0000001,0.1]
	l1_ratio	Float	[0.000000001,1]
	fit_intercept	Constant	[True]
	tol	Float	[0.00001,0.1]
	epsilon	Float	[0.00001,0.1]
	learning_rate	Categorical	[optimal, invscaling, constant,]
	eta0	Float	[0.0000001,0.1]
	power_t	Float	[0.00001,1]
	average	Categorical	[True, False]

continued on the next page

Table A.6: List of hyperparameters extracted from Auto-sklearn [39] for Classification problems – continued from the previous page

Operator	Hyperparameter	Type	Range
Feature Preprocessing	**ExtraTreesClassifier** (*sklearn.ensemble import ExtraTreesClassifier*)		
	n_estimators	Constant	[100]
	criterion	Categorical	[gini, entropy]
	max_features	Float	[0,1]
	max_depth	Constant	[None]
	min_samples_split	Integer	[2,20]
	min_samples_leaf	Integer	[1,20]
	min_weight_fraction_leaf	Constant	[0]
	max_leaf_nodes	Constant	[None]
	min_impurity_decrease	Constant	[0]
	bootstrap	Categorical	[True, False]
	FastICA (*sklearn.decomposition.FastICA*)		
	n_components	Categorical	[10,2000]
	algorithm	Categorical	[parallel, deflation]
	whiten	Categorical	[True, False]
	fun	Categorical	[logcosh, exp, cube]
	FeatureAgglomeration (*sklearn.cluster.FeatureAgglomeration*)		
	n_clusters	Integer	[2, 400]
	affinity	Categorical	[euclidean, manhattan, cosine]
	linkage	Categorical	[ward, complete, average]
	pooling_func	Categorical	[mean, median, max]
	KernelPCA (*sklearn.decomposition.KernelPCA*)		
	n_components	Integer	[10,2000]
	kernel	Categorical	[poly, rbf, sigmoid, cosine]
	gamma	Float	[3.05E-05,8]
	degree	Float	[-1,1]
	RandomKitchenSinks (*sklearn.kernel_approximation.RBFSampler*)		
	gamma	Float	[3.05E-05,8]
	n_components	Float	[50,10000]
	LibLinear (*sklearn.svm.LinearSVC*)		
	penalty	Constant	[l1]
	loss	Categorical	[hinge, squared_hinge]
	dual	Constant	[False]
	tol	Float	[1E-05 1E-01]
	C	Float	[0.03125, 32768]
	multi_class	Constant	[ovr]
	fit_intercept	Constant	[True]
	intercept_scaling	Constant	[1]
	No Preprocessing		

continued on the next page

Table A.6: List of hyperparameters extracted from Auto-sklearn [39] for Classification problems – continued from the previous page

Operator	Hyperparameter	Type	Range
Feature Preprocessing	**Nystroem**(*sklearn.kernel_approximation.Nystroem*)		
	kernel	Categorical	[poly, rbf, sigmoid, cosine]
	gamma	Float	[3.05E-05,8]
	n_components	Integer	[50, 10000]
	degree	Integer	[2,5]
	coef0	Float	[-1,1]
	PCA(*sklearn.decomposition.PCA*)		
	keep_variance	Float	[0.5, 0.9999]
	whiten	Categorical	[True, False]
	Polynomial(*sklearn.preprocessing.PolynomialFeatures*)		
	degree	Integer	[2,3]
	interaction_only	Categorical	[True, False]
	include_bias	Categorical	[True, False]
	RandomTreesEmbedding(*sklearn.ensemble.RandomTreesEmbedding*)		
	n_estimators	Integer	[10,100]
	max_depth	Constant	[2,10]
	min_samples_split	Integer	[2,20]
	min_samples_leaf	Integer	[1,20]
	min_weight_fraction_leaf	Constant	[0]
	max_leaf_nodes	Constant	[None]
	bootstrap	Categorical	[True, False]
	SelectPercentile(*sklearn.feature_selection.SelectPercentile*)		
	percentile	Float	[1,99]
	score_func	Categorical	[chi2, f_classif, mutual_info]
	score_func	Constant	[chi2]
	SelectRates(*sklearn.feature_selection.GenericUnivariateSelect*)		
	alpha	Float	[0.01,0.5]
	score_func	Constant	[chi2]
	score_func	Categorical	[chi2, f_classif]
	mode	Categorical	[for, fdr, fwe]
	TruncatedSVD(*sklearn.decomposition.TruncatedSVD(algorithm='randomized')*)		
	target_dim	Integer	[10,256]

continued on the next page

Table A.6: List of hyperparameters extracted from Auto-sklearn [39] for Classification problems – continued from the previous page

Operator	Hyperparameter	Type	Range
Balancing(*Additional hyperparameter to classification algorithms*)			
	strategy	Categorical	[none, weighting]
Categorical encoding			
	NoEncoding (*No hyperparameter*)		
	OneHotEncoder use *autosklearn.pipeline.implementations.SparseOneHotEncoder* for spare data and *sklearn.preprocessing.OneHotEncoder*for dense data		
Categorical Imputation (*sklearn.impute.SimpleImputer*)			
	strategy	Constant	[constant]
	fill_value	Constant	[2]
	copy	Constant	[False]
Numerical Imputation (*sklearn.impute.SimpleImputer*)			
	copy	Constant	[False]
	strategy	Categorical	[mean, median, most_frequent]
Minority Coalescense	**NoCoalescence** (*No hyperparameter*)		
	MinorityCoalescer (*autosklearn.pipeline.implementations.MinorityCoalescer*)		
	minimum_fraction	Float	[0.0001,0.5]
Rescaling	**MinMaxScaler** (*sklearn.preprocessing.MinMaxScaler*)		
	copy	Constant	[False]
	None (*No hyperparameter*)		
	QuantileTransformer (*sklearn.preprocessing.QuantileTransformer*)		
	n_quantiles	Integer	[10, 2000]
	output_distribution	Categorical	[uniform, normal]
	RobustScaler (*sklearn.preprocessing.RobustScaler*)		
	q_min	Float	[0.001, 0.3]
	q_max	Float	[0.7, 0.999]
	StandardScaler (*sklearn.preprocessing.StandardScaler*)		
	copy	Constant	[False]
Variance Threshold (*sklearn.feature_selection.VarianceThreshold*)			
	threshold	Constant	[0.0]

Bibliography

[1] S. Perla, N. N. K, and S. Potta, "Implementation of autonomous cars using machine learning," in *International Conference on Edge Computing and Applications (ICECAA)*, 2022, pp. 1444–1451.

[2] I. Kononenko, "Machine learning for medical diagnosis: History, state of the art and perspective," *Artificial Intelligence in Medicine*, vol. 23, no. 1, pp. 89–109, 2001.

[3] L. Galway, D. Charles, and M. Black, "Machine learning in digital games: A survey," *Artificial Intelligence Review*, vol. 29, pp. 123–161, 2008.

[4] P. Jain, S. Gupta, and K. Ramkumar, "Review on face recognition by machine learning and deep learning approaches," in *International Conference on Computing for Sustainable Global Development*, 2019, pp. 528–534.

[5] Y. A. Mohamed, A. Khanan, M. Bashir, A. H. H. M. Mohamed, M. A. E. Adiel, and M. A. Elsadig, "The impact of artificial intelligence on language translation: A review," *IEEE Access*, vol. 12, pp. 25 553–25 579, 2024.

[6] A. H. Sabry and U. A. B. Ungku Amirulddin, "A review on fault detection and diagnosis of industrial robots and multi-axis machines," *Results in Engineering*, vol. 23, p. 102 397, 2024.

[7] V. Duc Nguyen, E. Zwanenburg, S. Limmer, W. Luijben, T. Bäck, and M. Olhofer, "A combination of fourier transform and machine learning for fault detection and diagnosis of induction motors," in *International Conference on Dependable Systems and Their Applications (DSA)*, 2021, pp. 344–351.

[8] F. Hutter, L. Kotthoff, and J. Vanschoren, Eds., *Automatic machine learning: methods, systems, challenges* (Challenges in Machine Learning). Germany: Springer, 2019.

[9] D. Wolpert and W. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, 1997.

[10] C. Giraud-Carrier and F. Provost, "Toward a justification of meta-learning : Is the no free lunch theorem a showstopper ?" In *Proceedings of the ICML-2005 Workshop on Meta-learning*, 2005, pp. 12–19.

[11] M.-A. Zöller, W. Titov, T. Schlegel, and M. F. Huber, "XAutoML: A visual analytics tool for understanding and validating automated machine learning," *ACM Transactions on Interactive Intelligent Systems*, vol. 13, no. 4, pp. 1–39, 2023.

[12] J. Drozdal, J. Weisz, D. Wang, *et al.*, "Trust in automl: Exploring information needs for establishing trust in automated machine learning systems," in *Proceedings of the International Conference on Intelligent User Interfaces*, ser. IUI '20, Association for Computing Machinery, 2020, pp. 297–307.

[13] B. Bischl, M. Binder, M. Lang, *et al.*, *Hyperparameter optimization: Foundations, algorithms, best practices and open challenges*, 2021. arXiv: 2107.05847.

[14] D. Vermetten, H. Wang, C. Doerr, and T. Bäck, "Integrated vs. sequential approaches for selecting and tuning CMA-ES variants," in *The Genetic and Evolutionary Computation Conference*, 2020, pp. 903–912.

[15] D. A. Nguyen, A. V. Kononova, S. Menzel, B. Sendhoff, and T. Bäck, "Efficient AutoML via combinational sampling," in *IEEE Symposium Series on Computational Intelligence*, 2021, pp. 01–10.

[16] D. Peng, X. Dong, E. Real, *et al.*, "PyGlove: Symbolic programming for automated machine learning," in *Conference on Neural Information Processing Systems (NeurIPS)*, vol. 33, 2020, pp. 96–108.

[17] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[18] G. Batista, A. Bazzan, and M. C. Monard, "Balancing training data for automated annotation of keywords: A case study.," *II Brazilian Workshop on Bioinformatics*, pp. 10–18, 2003.

[19] H. M. Nguyen, E. W. Cooper, and K. Kamei, "Borderline over-sampling for imbalanced data classification," *International Journal of Knowledge Engineering and Soft Data Paradigms*, vol. 3, no. 1, pp. 4–21, 2011.

[20] W. Zuo, D. Zhang, and K. Wang, "On kernel difference-weighted k-nearest neighbor classification," *Pattern Analysis and Applications*, vol. 11, pp. 247–257, 2007.

[21] H. Drucker, C. J. C. Burges, L. Kaufman, A. Smola, and V. Vapnik, "Support vector regression machines," in *Conference on Neural Information Processing Systems (NIPS)*, vol. 9, MIT Press, 1996.

[22] M. Zöller and M. Huber, "Benchmark and survey of automated machine learning frameworks," *Journal of Artificial Intelligence Research*, vol. 70, pp. 409–472, 2021.

[23] J. Lévesque, A. Durand, C. Gagné, and R. Sabourin, "Bayesian optimization for conditional hyperparameter spaces," in *International Joint Conference on Neural Networks*, 2017, pp. 286–293.

[24] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Conference on Neural Information Processing Systems (NIPS)*, vol. 24, 2011, pp. 2546–2554.

[25] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Learning and Intelligent Optimization (LION)*, 2011, pp. 507–523.

[26] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Parallel algorithm configuration," in *Learning and Intelligent Optimization (LION)*, 2012, pp. 55–70.

[27] T. Bartz-Beielstein, C. Lasarczyk, and M. Preuss, "Sequential parameter optimization," *IEEE Congress on Evolutionary Computation*, vol. 1, pp. 773–780, 2005.

[28] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Conference on Neural Information Processing Systems (NIPS)*, vol. 25, Curran Associates, Inc., 2012.

[29] S. M. LaValle, M. S. Branicky, and S. R. Lindemann, "On the relationship between classical grid search and probabilistic roadmaps," *The International Journal of Robotics Research*, vol. 23, no. 7-8, pp. 673–692, 2004.

[30] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. 10, pp. 281–305, 2012.

[31] Z. Wang, M. Zoghi, F. Hutter, D. Matheson, and N. De Freitas, "Bayesian optimization in high dimensions via random embeddings," in *International Joint Conference on Artificial Intelligence*, 2013.

[32] O. Maron and A. W. Moore, "The racing algorithm: Model selection for lazy learners," *Artificial Intelligence Review*, vol. 11, pp. 193–225, 1997.

[33] M. López-Ibáñez, J. Dubois-Lacoste, T. Stützle, and M. Birattari, "The irace package: Iterated racing for automatic algorithm configuration," *Operations Research Perspectives*, vol. 3, pp. 43–58, 2016.

[34] K. Jamieson and A. Talwalkar, "Non-stochastic best arm identification and hyperparameter optimization," in *The International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2016, pp. 240–248.

[35] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *Journal of Machine Learning Research*, 2017.

[36] S. Falkner, A. Klein, and F. Hutter, "BOHB: Robust and efficient hyperparameter optimization at scale," in *International Conference on Machine Learning*, 2018, pp. 1437–1445.

[37] D. A. Nguyen, A. V. Kononova, S. Menzel, B. Sendhoff, and T. Bäck, "An efficient contesting procedure for automl optimization," in *IEEE Access*, vol. 10, 2022, pp. 75 754–75 771.

[38] B. van Stein, H. Wang, and T. Bäck, "Automatic configuration of deep neural networks with parallel efficient global optimization," in *International Joint Conference on Neural Networks*, 2019, pp. 1–7.

[39] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *Conference on Neural Information Processing Systems (NIPS)*, 2015.

[40] C. Thornton, F. Hutter, H. Hoos, and K. Leyton-Brown, "Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms," in *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, Association for Computing Machinery, 2012, pp. 847–855.

[41] L. Kotthoff, C. Thornton, H. H. Hoos, F. Hutter, and K. Leyton-Brown, "Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka," *Journal of Machine Learning Research*, vol. 18, no. 25, pp. 1–5, 2017.

[42] B. Komer, J. Bergstra, and C. Eliasmith, "Hyperopt-Sklearn: Automatic hyperparameter configuration for scikit-learn," in *Proceedings of the Python in Science Conferences (SciPy)*, 2014.

[43] R. S. Olson and J. H. Moore, "TPOT: A tree-based pipeline optimization tool for automating machine learning," in *Automated Machine Learning: Methods, Systems, Challenges*. Cham: Springer International Publishing, 2019, pp. 151–160.

[44] H. J. Escalante, M. Montes, and L. E. Sucar, "Particle swarm model selection," *Journal of Machine Learning Research*, vol. 10, no. 15, pp. 405–440, 2009.

[45] M. Feurer, K. Eggensperger, S. Falkner, M. L., and F. Hutter, *Auto-sklearn 2.0: The next generation*, 2020. arXiv: `2007.04074 [cs.LG]`.

[46] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim, "Do we need hundreds of classifiers to solve real world classification problems?" *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3133–3181, 2014.

[47] D. A. Nguyen, J. Kong, H. Wang, *et al.*, "Improved automated cash optimization with tree parzen estimators for class imbalance problems," in *IEEE International Conference on Data Science and Advanced Analytics*, 2021, pp. 1–9.

[48] G. Lemaître, F. Nogueira, and C. K. Aridas, "Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning," *Journal of Machine Learning Research*, vol. 18, no. 17, pp. 1–5, 2017.

[49] X. Wen, J. Shan, Y. He, and K. Song, "Steel surface defect recognition: A survey," *Coatings*, vol. 13, no. 1, p. 17, 2022.

[50] D. Jackson, "Towards a theory of conceptual design for software," in *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Association for Computing Machinery, 2015, pp. 282–296.

[51] C. Molnar, *Interprtable machine learning: A guide for making black box models explainable*, 2018.

[52] D. Conway and J. M. White, "Machine learning for hackers - case studies and algorithms to get you started," " O'Reilly Media, Inc.", 2012.

[53] L. Yang and A. Shami, "On hyperparameter optimization of machine learning algorithms: Theory and practice," *Neurocomputing*, vol. 415, pp. 295–316, 2020.

[54] I. V. Tetko, R. van Deursen, and G. Godin, *Be aware of overfitting by hyperparameter optimization!* 2024. arXiv: 2407.20786 [cs.LG].

[55] O. Falana, O. Durodola, and O. Oladipupo, "Implementing kaizen/continuous improvement in manufacturing industry," 2022.

[56] J. Brownlee, *Data preparation for machine learning: data cleaning, feature selection, and data transforms in Python*. Machine Learning Mastery, 2020.

[57] D. Pyle, *Data preparation for data mining*. Morgan Kaufmann Publishers Inc., 1999.

[58] R. Egele, J. C. S. J. Junior, J. N. van Rijn, *et al.*, *Ai competitions and benchmarks: Dataset development*, 2024. arXiv: 2404.09703 [cs.LG].

[59] R. Barga, V. Fontama, W. H. Tok, R. Barga, V. Fontama, and W. H. Tok, "Data preparation," *Predictive Analytics with Microsoft Azure Machine Learning*, pp. 45–79, 2015.

[60] X. Chu, J. Morcos, I. F. Ilyas, *et al.*, "KATARA: A data cleaning system powered by knowledge bases and crowdsourcing," in *Proceedings of the ACM SIGMOD international conference on management of data*, Association for Computing Machinery, 2015, pp. 1247–1261.

[61] O. Deshpande, D. S. Lamba, M. Tourn, *et al.*, "Building, maintaining, and using knowledge bases: A report from the trenches," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2013, pp. 1209–1220.

[62] S. Buchholz and J. Latorre, "Crowdsourcing preference tests, and how to detect cheating," in *Annual Conference of the International Speech Communication Association*, 2011.

[63] S. Krishnan, J. Wang, M. J. Franklin, *et al.*, "Sampleclean: Fast and reliable analytics on dirty data," *IEEE Data Engineering Bulletin*, vol. 38, pp. 59–75, 2015.

[64] S. Krishnan, M. J. Franklin, K. Goldberg, J. Wang, and E. Wu, "Activeclean: An interactive data cleaning framework for modern machine learning," in *Proceedings of the International Conference on Management of Data*, ser. SIGMOD '16, Association for Computing Machinery, 2016, pp. 2117–2120.

[65] D. Haas, S. Krishnan, J. Wang, M. J. Franklin, and E. Wu, "Wisteria: Nurturing scalable data cleaning infrastructure," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 2004–2007, 2015.

[66] X. He, K. Zhao, and X. Chu, "AutoML: A survey of the state-of-the-art," *Knowledge-Based Systems*, vol. 212, p. 106 622, 2021.

[67] R. E. Shawi, M. Maher, and S. Sakr, "Automated machine learning: State-of-the-art and open challenges," *ArXiv*, 2019. arXiv: 1906.02287.

[68] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, 2006.

[69] A. E. Hoerl and R. W. Kennard, "Ridge regression: Applications to nonorthogonal problems," *Technometrics*, vol. 12, no. 1, pp. 69–82, 1970.

[70] L. Melkumova and S. Shatskikh, "Comparing ridge and LASSO estimators for data analysis," *Procedia Engineering*, vol. 201, pp. 746–755, 2017.

[71] S. Safavian and D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 21, no. 3, pp. 660–674, 1991.

[72] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[73] M. Hossin and M. N. Sulaiman, "A review on evaluation metrics for data classification evaluations," *International journal of data mining & knowledge management process*, vol. 5, no. 2, p. 1, 2015.

[74] J. Kong, W. Kowalczyk, D. A. Nguyen, S. Menzel, and T. Bäck, "Hyperparameter optimisation for improving classification under class imbalance," in *IEEE Symposium Series on Computational Intelligence*, IEEE, 2019, pp. 3072–3078.

[75] A. Ali, S. M. Shamsuddin, and A. Ralescu, "Classification with class imbalance problem: A review," *Soft Computing Models in Industrial and Environmental Applications*, vol. 7, pp. 176–204, 2015.

[76] C. J. Willmott and K. Matsuura, "Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance," *Climate Research*, vol. 30, no. 1, pp. 79–82, 2005.

[77] J.-O. Palacio-Niño and F. Berzal, "Evaluation metrics for unsupervised learning algorithms," arXiv, 2019. arXiv: 1905.05667.

[78] E. G. Dada, J. S. Bassi, H. Chiroma, S. M. Abdulhamid, A. O. Adetunmbi, and O. E. Ajibuwa, "Machine learning for email spam filtering: Review, approaches and open research problems," *Heliyon*, vol. 5, no. 6, e01802, 2019.

[79] Y. Liu and S. Yang, "Application of decision tree-based classification algorithm on content marketing," *Journal of Mathematics*, vol. 2022, Mar. 2022.

[80] M. Grandini, E. Bagli, and G. Visani, "Metrics for multi-class classification: An overview," *ArXiv*, 2020. arXiv: 2008.05756.

[81] M. Kubat, "Addressing the curse of imbalanced training sets: One-sided selection," *International Conference on Machine Learning*, 2000.

[82] A. G. Salman, B. Kanigoro, and Y. Heryadi, "Weather forecasting using deep learning techniques," in *2015 International Conference on Advanced Computer Science and Information Systems (ICACSIS)*, 2015, pp. 281–285.

[83] T. D. Phan, "Housing price prediction using machine learning algorithms: The case of melbourne city, australia," in *International Conference on Machine Learning and Data Engineering (iCMLDE)*, 2018, pp. 35–42.

[84] D. C. Montgomery, E. A. Peck, and G. G. Vining, *Introduction to linear regression analysis*. John Wiley & Sons, 2021.

[85] O. A. Montesinos López, A. Montesinos López, and J. Crossa, "Overfitting, model tuning, and evaluation of prediction performance," in *Multivariate Statistical Machine Learning Methods for Genomic Prediction*. Cham: Springer International Publishing, 2022, pp. 109–139.

[86] G. C. Cawley and N. L. Talbot, "On over-fitting in model selection and subsequent selection bias in performance evaluation," *Journal of Machine Learning Research*, vol. 11, pp. 2079–2107, 2010.

[87] A. Krogh and J. Hertz, "A simple weight decay can improve generalization," in *Conference on Neural Information Processing Systems (NIPS)*, vol. 4, Morgan-Kaufmann, 1991.

[88] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014.

[89] E. LeDell and S. Poirier, "H2O AutoML: Scalable automatic machine learning," *ICML Workshop on Automated Machine Learning*, 2020.

[90] T. Swearingen, W. Drevo, B. Cyphers, A. Cuesta-Infante, A. Ross, and K. Veeramachaneni, "ATM: A distributed, collaborative, scalable system for automated machine learning," in *IEEE International Conference on Big Data (Big Data)*, 2017, pp. 151–162.

[91] R. Fakoor, J. W. Mueller, N. Erickson, P. Chaudhari, and A. J. Smola, "Fast, accurate, and simple models for tabular data via augmented distillation," *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[92] X. Shi, J. Mueller, N. Erickson, M. Li, and A. Smola, "Multimodal automl on structured tables with text fields," in *ICML Workshop on Automated Machine Learning (AutoML)*, 2021.

[93] R. Lopez, R. Lourenco, R. Rampin, *et al.*, "AlphaD3M: An open-source automl library for multiple ml tasks," in *Proceedings of the International Conference on Automated Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 224, PMLR, 2023, pp. 22/1–22.

[94] I. Drori, Y. Krishnamurthy, R. Lourenço, *et al.*, "Automatic machine learning by pipeline synthesis using model-based reinforcement learning and a grammar," *CoRR*, vol. abs/1905.10345, 2019. arXiv: 1905.10345.

[95] F. Mohr, M. Wever, and E. Hüllermeier, "ML-Plan: Automated machine learning via hierarchical planning," *Machine Learning*, vol. 107, pp. 1495–1515, 2018.

[96] M. Wever, F. Mohr, and E. Hüllermeier, "ML-Plan for unlimited-length machine learning pipelines," in *International Conference on Machine Learning*, 2018.

[97] Y. Gil, K.-T. Yao, V. Ratnakar, *et al.*, "P4ML: A phased performance-based pipeline planner for automated machine learning," in *Proceedings of Machine Learning Research, ICML 2018 AutoML Workshop*, 2018.

[98] H. Rakotoarison, M. Schoenauer, and M. Sebag, "Automated machine learning with monte-carlo tree search," in *International Joint Conference on Artificial Intelligence*, International Joint Conferences on Artificial Intelligence Organization, 2019, pp. 3296–3303.

[99] N. O. Nikitin, P. Vychuzhanin, M. Sarafanov, *et al.*, "Automated evolutionary approach for the design of composite machine learning pipelines," *Future Generation Computer Systems*, vol. 127, pp. 109–125, 2022.

[100] T.-D. Nguyen, K. Musial, and B. Gabrys, "Autoweka4mcps-avatar: Accelerating automated machine learning pipeline composition and optimisation," *Expert Systems with Applications*, vol. 185, p. 115 643, 2021.

[101] M. Zöller, T. Nguyen, and M. F. Huber, "Incremental search space construction for machine learning pipeline synthesis," *CoRR*, vol. abs/2101.10951, 2021. arXiv: 2101.10951.

[102] J. R. Koza and R. Poli, "Genetic programming," in *Search methodologies*, Springer, 2005, pp. 127–164.

[103] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[104] B. M. Lake, T. D. Ullman, J. B. Tenenbaum, and S. J. Gershman, "Building machines that learn and think like people," *Behavioral and Brain Sciences*, vol. 40, e253, 2017.

[105] M. Ghallab, D. Nau, and P. Traverso, *Automated planning. Theory & practice*. Morgan Kaufmann Publishers, 2004.

[106] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *European conference on machine learning*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 282–293.

[107] C. Browne, E. Powley, D. Whitehouse, *et al.*, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4:1, pp. 1–43, 2012.

[108] T. Bäck, *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[109] N. Hansen, "The CMA evolution strategy: A comparing review," *Towards a new evolutionary computation*, pp. 75–102, 2006.

[110] A. Słowik and H. Kwasnicka, "Evolutionary algorithms and their applications to engineering problems," *Neural Computing and Applications*, pp. 1–17, 2020.

[111] M. J. Post, P. van der Putten, and J. N. van Rijn, "Does feature selection improve classification? a large scale experiment in openml," in *International Symposium on Intelligent Data Analysis*, 2016.

[112] B. Pfahringer and C. Giraud-Carrier, "Meta-learning by landmarking various learning algorithms," in *International Conference on Machine Learning*, 2000, pp. 743–750.

[113] J. N. van Rijn, S. M. Abdulrahman, P. Brazdil, and J. Vanschoren, "Fast algorithm selection using learning curves," in *Advances in Intelligent Data Analysis XIV*, Cham: Springer International Publishing, 2015, pp. 298–309.

[114] J. Vanschoren, "Meta-learning," F. Hutter, L. Kotthoff, and J. Vanschoren, Eds., ser. Challenges in Machine Learning, Germany: Springer, 2019, ch. 2, pp. 39–68.

[115] K. Li and J. Malik, "Learning to optimize," in *International Conference on Learning Representations*, 2017.

[116] F. Hutter, H. Hoos, and K. Leyton-Brown, "An efficient approach for assessing hyperparameter importance," in *Proceedings of the International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 32, PMLR, 2014, pp. 754–762.

[117] M. Wistuba, N. Schilling, and L. Schmidt-Thieme, "Hyperparameter search space pruning – a new component for sequential model-based hyperparameter optimization," in *Machine Learning and Knowledge Discovery in Databases*, Cham: Springer International Publishing, 2015, pp. 104–119.

[118] J. N. van Rijn and F. Hutter, "Hyperparameter importance across datasets," in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 2367–2376.

[119] P. Probst, A.-L. Boulesteix, and B. Bischl, "Tunability: Importance of hyperparameters of machine learning algorithms," *Journal of Machine Learning Research (JMLR)*, vol. 20, no. 1, pp. 1934–1965, 2019.

[120] M. Wistuba, N. Schilling, and L. Schmidt-Thieme, "Learning hyperparameter optimization initializations," in *IEEE International Conference on Data Science and Advanced Analytics*, 2015, pp. 1–10.

[121] T.-D. Nguyen, D. J. Kedziora, K. Musial, and B. Gabrys, "Exploring opportunistic meta-knowledge to reduce search spaces for automated machine learning," in *2021 International Joint Conference on Neural Networks (IJCNN)*, 2021, pp. 1–8.

[122] V. Perrone, H. Shen, M. Seeger, C. Archambeau, and R. Jenatton, "Learning search spaces for bayesian optimization: Another view of hyperparameter transfer learning," in *Proceedings of the International Conference on Neural Information Processing Systems*. Curran Associates Inc., 2019.

[123] B. Bilalli, A. Abelló, and T. Aluja-Banet, "On the predictive power of meta-features in openml," *International Journal of Applied Mathematics and Computer Science*, vol. 27, no. 4, pp. 697–712, 2017.

[124] B. Schoenfeld, C. Giraud-Carrier, M. Poggemann, J. Christensen, and K. Seppi, *Preprocessor selection for machine learning pipelines*, 2018.

[125] M. Feurer, J. Springenberg, and F. Hutter, "Initializing bayesian hyperparameter optimization via meta-learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 29, no. 1, 2015.

[126] M. Lindauer and F. Hutter, "Warmstarting of model-based algorithm configuration," in *Proceedings of the AAAI Conference on Artificial Intelligence and Innovative Applications of Artificial Intelligence Conference and AAAI Symposium on Educational Advances in Artificial Intelligence*, ser. AAAI'18/IAAI'18/EAAI'18, AAAI Press, 2018.

[127] T. A. F. Gomes, R. B. C. Prudêncio, C. Soares, A. L. D. Rossi, and A. Carvalho, "Combining meta-learning and search techniques to svm parameter selection," in *2010 Eleventh Brazilian Symposium on Neural Networks*, 2010, pp. 79–84.

[128] S. Ullah, Z. Xu, H. Wang, S. Menzel, B. Sendhoff, and T. Bäck, "Exploring clinical time series forecasting with meta-features in variational recurrent models," in *International Joint Conference on Neural Networks*, 2020, pp. 1–9.

[129] C. Wang, T. Bäck, H. H. Hoos, M. Baratchi, S. Limmer, and M. Olhofer, "Automated machine learning for short-term electric load forecasting," in *IEEE Symposium Series on Computational Intelligence*, 2019, pp. 314–321.

[130] M. Kefalas, M. Baratchi, A. Apostolidis, D. van den Herik, and T. Bäck, "Automated machine learning for remaining useful life estimation of aircraft engines," in *International Conference on Prognostics and Health Management*, 2021, pp. 1–9.

[131] D. Wang, J. D. Weisz, M. Muller, *et al.*, "Human-ai collaboration in data science: Exploring data scientists' perceptions of automated ai," *Proceedings of the ACM on Human-Computer Interaction*, vol. 3, no. CSCW, 2019.

[132] A. Crisan and B. Fiore-Gartland, "Fits and starts: Enterprise use of automl and the role of humans in the loop," *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2021.

[133] D. Wang, J. Andres, J. D. Weisz, E. Oduor, and C. Dugan, "Autods: Towards human-centered automation of data science," *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2021.

[134] Q. Wang, Y. Ming, Z. Jin, *et al.*, "Atmseer: Increasing transparency and controllability in automated machine learning," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, ser. CHI '19, Association for Computing Machinery, 2019, pp. 1–12.

[135] N. Burkart and M. F. Huber, "A survey on the explainability of supervised machine learning," *Journal of Artificial Intelligence Research (JAIR)*, vol. 70, pp. 245–317, 2021.

[136] Z. C. Lipton, "The mythos of model interpretability: In machine learning, the concept of interpretability is both important and slippery.," *Queue*, vol. 16, no. 3, pp. 31–57, 2018.

[137] E. Kazim and A. Koshiyama, *Explaining decisions made with AI: a review of the co-badged guidance by the ICO and the Turing Institute*. 2020.

[138] I. van der Linden, H. Haned, and E. Kanoulas, "Global aggregations of local explanations for black box models," *ArXiv*, vol. abs/1907.03039, 2019.

[139] T. Kulesza, M. Burnett, W.-K. Wong, and S. Stumpf, "Principles of explanatory debugging to personalize interactive machine learning," in *Proceedings of the International Conference on Intelligent User Interfaces*, ser. IUI '15, Association for Computing Machinery, 2015, pp. 126–137.

[140] J. Fox, D. Glasspool, D. Grecu, S. Modgil, M. South, and V. Patkar, "Argumentation-based inference and decision making–a medical perspective," *IEEE Intelligent Systems*, vol. 22, no. 6, pp. 34–41, 2007.

[141] M. Ribeiro, S. Singh, and C. Guestrin, ""why should I trust you?": Explaining the predictions of any classifier," in *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*, San Diego, California: Association for Computational Linguistics, 2016, pp. 97–101.

[142] B. Hayes and J. A. Shah, "Improving robot controller transparency through autonomous policy explanation," in *ACM/IEEE International Conference on Human-Robot Interaction (HRI*, 2017, pp. 303–312.

[143] J. E. Mercado, M. A. Rupp, J. Y. C. Chen, M. J. Barnes, D. Barber, and K. Procci, "Intelligent agent transparency in human–agent teaming for multi-uxv management," *Human Factors*, vol. 58, no. 3, pp. 401–415, 2016, PMID: 26867556.

[144] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[145] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. E. Karro, and D. Sculley, "Google vizier: A service for black-box optimization," *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1487–1495, 2017.

[146] J. Liu, S. Tripathi, U. Kurup, and M. Shah, "Auptimizer - an extensible, open-source framework for hyperparameter tuning," *IEEE International Conference on Big Data (Big Data)*, pp. 339–348, 2019.

[147] J. Ono, S. Castelo, R. Lopez, E. Bertini, J. Freire, and C. Silva, "Pipelineprofiler: A visual analytics tool for the exploration of automl pipelines," *IEEE Transactions on Visualization and Computer Graphics*, vol. PP, pp. 1–1, 2020.

[148] T. Li and T. Zajonc, "HyperTuner: Visual analytics for hyperparameter tuning by professionals," in *IEEE Workshop on Machine Learning from User Interaction for Visualization and Analytics (MLUI)*, 2018, pp. 1–11.

[149] H. Park, Y. J. Nam, J.-H. Kim, and J. Choo, "HyperTendril: Visual analytics for user-driven hyperparameter optimization of deep neural networks," *IEEE Transactions on Visualization and Computer Graphics*, vol. 27, pp. 1407–1416, 2021.

[150] D. K. I. Weidele, J. D. Weisz, E. Oduor, *et al.*, "AutoAIViz: Opening the blackbox of automated artificial intelligence with conditional parallel coordinates," *Proceedings of the International Conference on Intelligent User Interfaces*, 2020.

[151] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[152] C. Wang, Q. Wu, M. Weimer, and E. Zhu, "FLAML: A fast and lightweight automl library," in *The Fourth Conference on Machine Learning and Systems (MLSys 2021)*, 2021.

[153] J. Bergstra, B. Komer, C. Eliasmith, D. Yamins, and D. Cox, "Hyperopt: A Python library for model selection and hyperparameter optimization," *Computational Science & Discovery*, vol. 8, no. 1, p. 014 008, 2015.

[154] P. Das, N. Ivkin, T. Bansal, *et al.*, "Amazon SageMaker autopilot: A white box automl solution at scale," in *SIGMOD/PODS 2020*, 2020.

[155] D. R. Jones, M. Schonlau, and W. J. Welch, "Efficient global optimization of expensive black-box functions," *Journal of Global Optimization*, vol. 13, no. 4, pp. 455–492, 1998.

[156] J. Močkus, "On bayesian methods for seeking the extremum," in *Optimization Techniques IFIP Technical Conference Novosibirsk*, Springer Berlin Heidelberg, 1975, pp. 400–404.

[157] D. Jones, "A taxonomy of global optimization methods based on response surfaces," *Journal of Global Optimization*, pp. 345–383, 2001.

[158] J. Bergstra, D. Yamins, and D. D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *International Conference on Machine Learning*, vol. 28, 2013, pp. 115–123.

[159] C. E. Rasmussen, "Gaussian processes in machine learning," in *Advanced Lectures on Machine Learning: ML Summer Schools*. Berlin, Heidelberg, 2004, pp. 63–71.

[160] P. Auer, "Using confidence bounds for exploitation-exploration trade-offs.," *Journal of Machine Learning Research*, vol. 3, pp. 397–422, 2002.

[161] S. Kotz, *Continuous multivariate distributions. volume 1, models and applications* (Wiley series in probability and statistics. Applied probability and statistics.), Array. Wiley, 2000, p. 722, "A Wiley-Interscience publication.".

[162] E. Parzen, "On estimation of a probability density function and mode," *The Annals of Mathematical Statistics*, vol. 33, no. 3, pp. 1065–1076, 1962.

[163] X. Wang, P. Tino, M. A. Fardal, S. Raychaudhury, and A. Babul, "Fast parzen window density estimator," in *2009 International Joint Conference on Neural Networks*, 2009, pp. 3267–3274.

[164] S. Węglarczyk, "Kernel density estimation and its application," in *ITM web of conferences*, EDP Sciences, vol. 23, 2018, p. 00 037.

[165] D. Reynolds, "Gaussian mixture models," in *Encyclopedia of Biometrics*. Boston, MA: Springer US, 2009, pp. 659–663.

[166] J. P. C. Kleijnen, W. C. M. V. Beers, and I. V. Nieuwenhuyse, "Expected improvement in efficient global optimization through bootstrapped kriging," *Journal of Global Optimization*, vol. 54, no. 1, pp. 59–73, 2012.

[167] A. Zilinskas, "A review of statistical models for global optimization," *Journal of Global Optimization*, vol. 2, no. 2, pp. 145–153, 1992.

[168] M. Schonlau, W. J. Welch, and D. R. Jones, "Global versus local search in constrained optimization of computer models," *Lecture Notes-Monograph Series*, vol. 34, pp. 11–25, 1998.

[169] M. Hoffman, B. Shahriari, and N. De Freitas, "On correlation and budget constraints in model-based bandit optimization with application to automatic machine learning," *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 365–374, 2014.

[170] V. Gabillon, M. Ghavamzadeh, and A. Lazaric, "Best arm identification: A unified approach to fixed budget and fixed confidence," in *Conference on Neural Information Processing Systems (NIPS)*, 2012, pp. 3221–3229.

[171] M. Hoffman, E. Brochu, and N. de Freitas, "Portfolio allocation for bayesian optimization," in *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, ser. UAI'11, AUAI Press, 2011, pp. 327–336.

[172] R. K. Ursem, "From expected improvement to investment portfolio improvement: Spreading the risk in kriging-based optimization," in *The International Conference on Parallel Problem Solving From Nature (PPSN)*, ser. Lecture Notes in Computer Science, vol. 8672, Springer, 2014, pp. 362–372.

[173] O. Maron and A. W. Moore, "Hoeffding Races: Accelerating model selection search for classification and function approximation," in *Conference on Neural Information Processing Systems (NIPS)*, 1993.

[174] B. Samanta, "Gear fault detection using artificial neural networks and support vector machines with genetic algorithms," *Mechanical Systems and Signal Processing*, vol. 18, no. 3, pp. 625–644, 2004.

[175] A. V. D. Bosch, "Wrapped progressive sampling search for optimizing learning algorithm parameters," in *Proceedings of the Belgian-Dutch Conference on Artificial Intelligence*, 2004, pp. 219–226.

[176] M. Friedman, "The use of ranks to avoid the assumption of normality implicit in the analysis of variance," *Journal of the American Statistical Association*, pp. 675–701, 1937.

[177] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp, "A racing algorithm for configuring metaheuristics," in *Proceedings of the Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO'02, Morgan Kaufmann Publishers Inc., 2002, pp. 11–18.

[178] M. Birattari, *Tuning Metaheuristics: A Machine Learning Perspective*, 1st ed. 2005. 2nd printing. Springer Publishing Company, Incorporated, 2009.

[179] H. H. Hoos, "Automated algorithm configuration and parameter tuning," in *Autonomous Search*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 37–71.

[180] P. Balaprakash, M. Birattari, and T. Stützle, "Improvement strategies for the F-Race algorithm: Sampling design and iterative refinement," in *Hybrid Metaheuristics*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 108–122.

[181] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle, "F-Race and Iterated F-Race: An overview," in *Experimental Methods for the Analysis of Optimization Algorithms*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 311–336.

[182] C. Vieira, A. de Araújo, J. E. Andrade, and L. C. T. Bezerra, "iSklearn: Automated machine learning with irace," in *IEEE Congress on Evolutionary Computation*, 2021, pp. 2354–2361.

[183] W. J. Conover, *Practical nonparametric statistics*. john wiley & sons, 1999, vol. 350.

[184] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas, "Taking the human out of the loop: A review of bayesian optimization," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, 2016.

[185] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *2012 IEEE conference on computer vision and pattern recognition*, IEEE, 2012, pp. 3354–3361.

[186] J. Alcalá-Fdez, L. Sánchez, S. Garcia, *et al.*, "KEEL: A software tool to assess evolutionary algorithms for data mining problems," *Soft Computing*, vol. 13, no. 3, pp. 307–318, 2009.

[187] H. L. Le, D. Landa-Silva, M. Galar, S. Garcia, and I. Triguero, "EUSC: A clustering-based surrogate model to accelerate evolutionary undersampling in imbalanced classification," *Applied Soft Computing*, vol. 101, p. 107 033, 2021.

[188] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Transactions on Knowledge & Data Engineering*, no. 9, pp. 1263–1284, 2008.

[189] P. Gijsbers, E. LeDell, S. Poirier, J. Thomas, B. Bischl, and J. Vanschoren, "An open source automl benchmark," *ICML Workshop on Automated Machine Learning*, 2019.

[190] H. He, Y. Bai, E. A. Garcia, and S. Li, "Adasyn: Adaptive synthetic sampling approach for imbalanced learning," in *IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, 2008, pp. 1322–1328.

[191] H. Han, W. Wang, and B. H. Mao, "Borderline-smote: A new over-sampling method in imbalanced data sets learning," vol. 3644, 2005, pp. 878–887.

[192] F. Last, G. Douzas, and F. Bacao, *Oversampling for imbalanced learning based on k-means and smote*, 2017. arXiv: `1711.00837 [cs.LG]`.

[193] Imbalanced-learn, *Randomoversampler*, https://imbalanced-learn.org /stable/generated/imblearn.over_sampling. RandomOverSampler.html, Accessed: 2020-08-30.

[194] I. Tomek, "Two modifications of CNN," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-6, no. 11, pp. 769–772, 1976.

[195] K. Gowda and G. Krishna, "The condensed nearest neighbor rule using the concept of mutual nearest neighborhood (corresp.)," *IEEE Transactions on Information Theory*, vol. 25, no. 4, pp. 488–490, 1979.

[196] D. L. Wilson, "Asymptotic properties of nearest neighbor rules using edited data," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-2, no. 3, pp. 408–421, 1972.

[197] I. Tomek, "An experiment with the edited nearest-neighbor rule," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-6, no. 6, pp. 448–452, 1976.

[198] M. R. Smith, T. Martinez, and C. Giraud-Carrier, "An instance level analysis of data complexity," *Machince Learning*, vol. 95, no. 2, pp. 225–256, 2014.

[199] J. Ziang, "KNN approach to unbalanced data distributions: A case study involving information extraction," *Proceedings of the ICML'2003 Workshop on Learning from Imbalanced Datasets*, 2003.

[200] J. Laurikkala, "Improving identification of difficult small classes by balancing class distribution," in *Artificial Intelligence in Medicine*, 2001, pp. 63–66.

[201] Imbalanced-learn, *Clustercentroids*, https://imbalanced-learn.org/ stable/ generated /imblearn.under_sampling.ClusterCentroids.html, Accessed: 2020-08-30.

[202] Imbalanced-learn, *Randomundersampler*, https://imbalanced-learn.org /stable/generated/imblearn.under_sampling.RandomUnderSampler.html, Accessed: 2020-08-30.

[203] G. E. Batista, R. C. Prati, and M. C. Monard, "A study of the behavior of several methods for balancing machine learning training data," *ACM SIGKDD explorations newsletter*, vol. 6, no. 1, pp. 20–29, 2004.

[204] J. Vanschoren, J. N. van Rijn, B. Bischl, and L. Torgo, "OpenML: Networked science in machine learning," *SIGKDD Explorations*, vol. 15, no. 2, pp. 49–60, 2013.

[205] B. Bischl, G. Casalicchio, M. Feurer, *et al.*, "Openml benchmarking suites and the OpenML100," 2017. arXiv: `1708.03731 [stat.ML]`.

[206] B. Bischl, G. Casalicchio, M. Feurer, *et al.*, "Openml benchmarking suites," *Proceedings of the NeurIPS Datasets and Benchmarks Track*, 2021.

[207] M. Feurer, J. van Rijn, A. Kadra, *et al.*, "OpenML-Python: An extensible python api for OpenML," *Journal of Machine Learning Research*, vol. 22, 2021.

[208] H. Bal, D. Epema, C. de Laat, *et al.*, "A medium-scale distributed system for computer science research: Infrastructure for the long term," *Computer*, vol. 49, no. 05, pp. 54–63, 2016.

[209] A. Fernández, S. García, M. Galar, R. C. Prati, B. Krawczyk, and F. Herrera, *Learning from imbalanced data sets*. Springer, 2018, vol. 10.

[210] S. Wang and X. Yao, "Using class imbalance learning for software defect prediction," *IEEE Transactions on Reliability*, vol. 62, no. 2, pp. 434–443, 2013.

[211] D. Rodriguez, I. Herraiz, R. Harrison, J. Dolado, and J. C. Riquelme, "Preliminary comparison of techniques for dealing with imbalance in software defect prediction," in *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering*, 2014, pp. 1–10.

[212] V. López, A. Fernández, J. G. Moreno-Torres, and F. Herrera, "Analysis of preprocessing vs. cost-sensitive learning for imbalanced classification. open problems on intrinsic data characteristics," *Expert Systems with Applications*, vol. 39, no. 7, pp. 6585–6608, 2012.

[213] P. Kerschke, H. H. Hoos, F. Neumann, and H. Trautmann, *Automated algorithm selection: Survey and perspectives*, 2018. arXiv: `1811.11597 [cs.LG]`.

[214] V. Ganganwar, "An overview of classification algorithms for imbalanced datasets," *International Journal of Emerging Technology and Advanced Engineering*, vol. 2, no. 4, pp. 42–47, 2012.

[215] J. Kong, T. Rios, W. Kowalczyk, S. Menzel, and T. Bäck, "On the performance of oversampling techniques for class imbalance problems," *Advances in Knowledge Discovery and Data Mining*, vol. 12085, p. 84, 2020.

[216] S. García and F. Herrera, "Evolutionary undersampling for classification with imbalanced datasets: Proposals and taxonomy," *Evolutionary Computation*, vol. 17, no. 3, pp. 275–306, 2009.

[217] J. Bacardit, D. Goldberg, M. Butz, X. Llora, and J. M. Garrell, "Speeding-up pittsburgh learning classifier systems: Modeling time and accuracy," *The International Conference on Parallel Problem Solving From Nature (PPSN)*, vol. 3242, pp. 1021–1031, 2004.

[218] H. L. Le, D. Landa-Silva, M. Galar, S. Garcia, and I. Triguero, "A hybrid surrogate model for evolutionary undersampling in imbalanced classification," in *IEEE Congress on Evolutionary Computation*, 2020, pp. 1–8.

[219] J. Huang and C. X. Ling, "Using auc and accuracy in evaluating learning algorithms," *IEEE Transactions on knowledge and Data Engineering*, vol. 17, no. 3, pp. 299–310, 2005.

[220] V. López, A. Fernández, S. García, V. Palade, and F. Herrera, "An insight into classification with imbalanced data: Empirical results and current trends on using data intrinsic characteristics," *Information sciences*, vol. 250, pp. 113–141, 2013.

[221] A. Tharwat, "Classification assessment methods: A detailed tutorial," *Applied Computing and Informatics*, vol. 17, no. 1, pp. 168–192, 2018.

[222] S. Wang, L. L. Minku, and X. Yao, "Dealing with multiple classes in online class imbalance learning.," in *International Joint Conference on Artificial Intelligence*, 2016, pp. 2118–2124.

[223] M. Galar, A. Fernández, E. Barrenechea, H. Bustince, and F. Herrera, "An overview of ensemble methods for binary classifiers in multi-class problems: Experimental study on one-vs-one and one-vs-all schemes," *Pattern Recognition*, vol. 44, no. 8, pp. 1761–1776, 2011.

[224] J. Xu, X. Liu, Z. Huo, C. Deng, F. Nie, and H. Huang, "Multi-class support vector machine via maximizing multi-class margins," in *International Joint Conference on Artificial Intelligence*, 2017.

[225] R. Rifkin and A. Klautau, "In defense of one-vs-all classification," *The Journal of Machine Learning Research*, vol. 5, pp. 101–141, 2004.

[226] J. Fürnkranz, "Round robin classification," *The Journal of Machine Learning Research*, vol. 2, pp. 721–747, 2002.

[227] A. C. Tan, D. Gilbert, and Y. Deville, "Multi-class protein fold classification using a new ensemble machine learning approach," *Genome Informatics*, vol. 14, pp. 206–217, 2003.

[228] D. Rathgamage Don and I. Iacob, "Dcsvm: Fast multi-class classification using support vector machines," *International Journal of Machine Learning and Cybernetics*, vol. 11, 2020.

[229] C.-W. Hsu and C.-J. Lin, "A comparison of methods for multiclass support vector machines," *IEEE transactions on Neural Networks*, vol. 13, no. 2, pp. 415–425, 2002.

[230] X. He, Z. Wang, C. Jin, Y. Zheng, and X. Xue, "A simplified multi-class support vector machine with reduced dual optimization," *Pattern Recognition Letters*, vol. 33, no. 1, pp. 71–82, 2012.

[231] B. Jijo and A. Mohsin Abdulazeez, "Classification based on decision tree algorithm for machine learning," *Journal of Applied Science and Technology Trends*, vol. 2, pp. 20–28, 2021.

[232] K. Crammer and Y. Singer, "On the algorithmic implementation of multiclass kernel-based vector machines," *Journal of machine learning research*, vol. 2, no. Dec, pp. 265–292, 2001.

[233] P. Piro, R. Nock, F. Nielsen, and M. Barlaud, "Multi-class leveraged k-nn for image classification," in *Proceedings of the Asian Conference on Computer Vision (ACCV)*, 2010.

[234] Z. Younes, F. Abdallah, and T. Denœux, "An evidence-theoretic k-nearest neighbor rule for multi-label classification," in *Scalable Uncertainty Management (SUM)*, Springer, 2009, pp. 297–308.

[235] A. Kontorovich and R. Weiss, "Maximum margin multiclass nearest neighbors," in *International conference on machine learning*, PMLR, 2014, pp. 892–900.

[236] J. D. Conklin, "Applied logistic regression," *Technometrics*, vol. 44, no. 1, pp. 81–82, 2002.

[237] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks," *Information processing & management*, vol. 45, no. 4, pp. 427–437, 2009.

[238] N. Neogi, D. K. Mohanta, and P. K. Dutta, "Review of vision-based steel surface inspection systems," *EURASIP Journal on Image and Video Processing*, vol. 2014, no. 1, pp. 1–19, 2014.

[239] M. D. McKay, R. J. Beckman, and W. J. Conover, "A comparison of three methods for selecting values of input variables in the analysis of output from a computer code," *Technometrics*, pp. 239–245, 1979.

[240] G. Muniraju, B. Kailkhura, J. J. Thiagarajan, P.-T. Bremer, C. Tepedelenlioglu, and A. Spanias, "Coverage-based designs improve sample mining and hyperparameter optimization," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 3, pp. 1241–1253, 2021.

[241] J. Li Y.and Jiang, J. Gao, Y. Shao, C. Zhang, and B. Cui, "Efficient automatic CASH via rising bandits," in *Association for the Advancement of Artificial Intelligence*, AAAI Press, 2020, pp. 4763–4771.

[242] K. Eggensperger, M. Feurer, F. Hutter, *et al.*, "Towards an empirical foundation for assessing bayesian optimization of hyperparameters," in *NIPS Workshop on Bayesian Optimization in Theory and Practice*, 2013.

[243] P. Yang, K. Tang, and X. Yao, "Turning high-dimensional optimization into computationally expensive optimization," *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 1, pp. 143–156, 2018.

[244] P. Yang, K. Tang, and X. Yao, "A parallel divide-and-conquer-based evolutionary algorithm for large-scale optimization," *IEEE Access*, vol. 7, pp. 163 105–163 118, 2019.

[245] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian Journal of Statistics*, vol. 6, no. 2, pp. 65–70, 1979.

[246] H. Ismail Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P. Muller, "Deep learning for time series classification: A review," *Data Mining and Knowledge Discovery*, vol. 33, pp. 917–963, 2019.

[247] I. Rodríguez-Fdez, A. Canosa, M. Mucientes, and A. Bugarín, "STAC: A web platform for the comparison of algorithms using statistical tests," in *IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, 2015, pp. 1–8.

[248] D. Ginsbourger, R. Le Riche, and L. Carraro, "Kriging is well-suited to parallelize optimization," in *Computational Intelligence in Expensive Optimization Problems*, ser. Springer series in Evolutionary Learning and Optimization, springer, 2010, pp. 131–162.

[249] K. Eggensperger, F. Hutter, H. Hoos, and K. Leyton-Brown, "Efficient benchmarking of hyperparameter optimizers via surrogates," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 29, 2015.

[250] K. Eggensperger, M. Lindauer, H. H. Hoos, F. Hutter, and K. Leyton-Brown, "Efficient benchmarking of algorithm configurators via model-based surrogates," *Machine Learning*, vol. 107, pp. 15–41, 2018.

[251] K. Leyton-Brown, E. Nudelman, and Y. Shoham, "Empirical hardness models: Methodology and a case study on combinatorial auctions," *Journal of the ACM (JACM)*, vol. 56, no. 4, pp. 1–52, 2009.

[252] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown, "Algorithm runtime prediction: Methods & evaluation," *Artificial Intelligence*, vol. 206, pp. 79–111, 2014.

[253] I. Guyon, L. Sun-Hosoya, M. Boullé, *et al.*, "Analysis of the automl challenge series 2015-2018," in *Automated Machine Learning*, ser. Springer series on Challenges in Machine Learning, 2019.

Index

English Summary

AutoML has attracted community attention due to its success in shortening the machine learning development cycle for real-world applications. Optimization plays a crucial role in AutoML frameworks by helping to identify a fine-tuned ML pipeline that suits a given practical problem. Several state-of-the-art optimization approaches, including Bayesian optimization, Bandit learning, and Racing procedures, have been proposed to enhance the performance of AutoML. However, the existing studies often formulate the AutoML optimization as a Hyperparameter Optimization (HPO) problem using the Combined Algorithm Selection and Hyperparameter Optimization (CASH) approach. This limited perspective can restrict their effectiveness in addressing the underlying problem effectively.

In this thesis, we comprehensively address this limitation by *formulating AutoML optimization*, covering the entire ML pipeline synthesis, as discussed in Chapter 1. Specifically, we introduce a novel class of hyperparameter, called "*algorithm choice*", which enables the modeling of algorithm selection. This class incorporates a unique attribute that allows for the hierarchical organization of algorithms based on their technical similarities. Notably, to our best knowledge, this is the first attempt to visualize the relationship between algorithms.

Next, a comprehensive investigation to provide a holistic understanding of AutoML and its optimization is provided in Chapters 2 and 3. Chapter 2 comprehensively investigates AutoML and its integral concepts and aspects. It delves into the foundations of AutoML, exploring the motivations behind its development and its significance in shortening the machine learning development cycle for real-world applications. Chapter 3 focuses specifically on AutoML optimization. This chapter delves into various optimization techniques and algorithms that are employed to improve the performance and efficiency of AutoML frameworks. It explores approaches such as Grid search, Random search, Bayesian optimization, Bandit learning, and Racing procedures, providing an in-depth analysis of their principles, strengths, and limitations in the context of AutoML optimization.

Another point of concern is benchmarking to evaluate the robustness and general applicability of optimization approaches empirically. Chapter 4 presents two sets of benchmark experiments. These benchmarks consist of 117 agreed-on datasets that may require processing through an ML pipeline of multiple operators such as encoding, normalization, resampling, and classification. The chosen datasets may contain categorical data, incomplete instances, or have an imbalanced distribution. Furthermore, the benchmarks comprise standardized search spaces along with an experimental methodology and setup for benchmarking purposes.

In Chapter 5, we conducted a detailed study on the effectiveness of AutoML optimization in handling class imbalance problems. Our findings suggest that BO is a highly efficient approach to tackle this problem. This finding enabled us to focus on improving BO to solve AutoML optimization problems confidently. Consequently, we were able to create two effective optimization algorithms based on BO in Chapter 7 and Chapter 8, as well as successfully implement the AutoML optimization approach to a real-world application in Chapter 6.

Another key contribution is the development of a performance metric specifically designed to address the dual challenges of class imbalance and unequal importance problems. This metric, discussed in Chapter 6, offers a comprehensive evaluation framework for assessing the performance of ML models in such scenarios.

In addition to the mentioned contributions, this thesis presents two novel AutoML optimization approaches. In Chapter 7 of this thesis, we presented a combinational sampling method that efficiently solves AutoML optimization problems through Bayesian optimization. Our approach is designed to maximize the coverage of ML algorithm samples in the search space, which results in a robust surrogate model for BO. We conducted experiments on 117 benchmark datasets and found that using our sampling approach significantly improves BO performance when compared to BO without our approach. In addition to the proposed sampling approach, we have also introduced an optimization software known as BO4ML in this chapter. This software is implemented based on the proposed sampling approach and can be used for optimization. Moving on to Chapter 8, we proposed an improved AutoML optimization technique known as DACOpt, which is an efficient contesting procedure. This innovative method aims to allocate tuning resources for optimizers over search areas more effectively. Specifically, DACOpt rewards resources to areas that show promise while terminating the optimizing process on areas that are less likely to yield desired results. Our experiments on

117 benchmark datasets indicate that our proposed approach offers a significant performance advantage over BO for AutoML optimization problems.

Nederlandse Samenvatting

Automatische machine learning technieken (AutoML) zijn sterk in opkomst vanwege het success van deze technieken in het reduceren van de ontwikkelingscyclus in praktische toepassingen. Optimalisatie speelt een cruciale rol in AutoML-frameworks door te helpen bij het identificeren van een nauwkeurig afgestelde ML-pijplijn die geschikt is voor een bepaald praktisch probleem. Er zijn verschillende geavanceerde optimalisatiebenaderingen voorgesteld, waaronder Bayesiaanse optimalisatie, Bandit-leren en raceprocedures, om de prestaties van AutoML te verbeteren.

De bestaande studies formuleren de AutoML-optimalisatie echter vaak als een Hyper Parameter Optimalisatie (HPO)-probleem met behulp van de Combined Algorithm Selection and Hyperparameter optimalisatie (CASH)-benadering. Dit beperkte perspectief kan hun effectiviteit bij het aanpakken van het onderliggende probleem beperken.

In dit proefschrift gaan we uitvoerig in op deze beperking door *AutoML-optimalisatie te formuleren*, die de volledige ML-pijplijnsynthese omvat, zoals besproken in Hoofdstuk 1. Specifiek introduceren we een nieuwe klasse hyperparameter – genaamd "*algorithm choice*", die het modelleren van algoritmeselectie mogelijk maakt. Deze klasse bevat een uniek attribuut dat de hiërarchische organisatie van algoritmen mogelijk maakt op basis van hun technische overeenkomsten. Voor zover wij weten, is dit de eerste poging om de relatie tussen algoritmen te visualiseren.

Vervolgens vindt u in de hoofdstukken 2 en 3 een uitgebreid onderzoek om een holistisch begrip van AutoML en de optimalisatie ervan te bieden. Hoofdstuk 2 onderzoekt AutoML en zijn integrale concepten en aspecten uitgebreid. Het diept in de fundamenten van AutoML, onderzoekt de motivaties achter de ontwikkeling ervan en draagt bij aan het verkorten van de ontwikkelingscyclus van machine learning voor toepassingen in de echte wereld. Hoofdstuk 3 richt zich specifiek op AutoML-optimalisatie. Dit hoofdstuk gaat dieper in op verschillende

optimalisatietechnieken en algoritmen die worden gebruikt om de prestaties en efficiëntie van AutoML-frameworks te verbeteren. Het onderzoekt benaderingen zoals rasterzoeken, willekeurig zoeken, Bayesiaanse optimalisatie, Bandit-leren en raceprocedures, en biedt een diepgaande analyse van hun principes, sterke punten en beperkingen in de context van AutoML-optimalisatie.

Een ander punt van zorg is benchmarking om de robuustheid en algemene toepasbaarheid van optimalisatiebenaderingen empirisch te evalueren. In Hoofdstuk 4 worden twee sets benchmarkexperimenten gepresenteerd. Deze benchmarks bestaan uit 117 overeengekomen datasets die verwerking vereisen via een ML-pijplijn van meerdere operatoren, zoals codering, normalisatie, hersteekproeven en classificatie. De gekozen datasets kunnen categorische gegevens bevatten, onvolledige instanties hebben, of een onevenwichtige distributie hebben. Verder omvatten de benchmarks gestandaardiseerde zoekruimtes, samen met een experimentele methodologie en opstelling voor benchmarkdoeleinden.

In Hoofdstuk 5 hebben we een gedetailleerde studie uitgevoerd naar de effectiviteit van AutoML-optimalisatie bij het omgaan met problemen van klassenonevenwichtigheid. Onze bevindingen suggereren dat Bayesiaanse Optimalisatie (BO) een zeer efficiënte aanpak is om dit probleem aan te pakken. Deze ontdekking stelde ons in staat om ons te concentreren op het verbeteren van BO om AutoML-optimalisatieproblemEn op te lossen. Als gevolg daarvan konden we twee effectieve optimalisatiealgoritmen creëren op basis van BO in Hoofdstuk 7 en Hoofdstuk 8, evenals succesvol de AutoML-optimalisatiebenadering implementeren in een praktische toepassing in Hoofdstuk 6.

Een andere belangrijke bijdrage is de ontwikkeling van een prestatie-metriek die specifiek is ontworpen om de dubbele uitdagingen van klassenonevenwichtigheid en ongelijke belangrijkheidsproblemen aan te pakken. Deze metriek, besproken in Hoofdstuk 6, biedt een uitgebreid evaluatiekader voor het beoordelen van de prestaties van ML-modellen in dergelijke scenario's.

Naast de genoemde bijdragen presenteert deze scriptie twee nieuwe AutoML-optimalisatiebenaderingen. In Hoofdstuk 7 hebben we een combinatie sampling methode gepresenteerd die AutoML-optimalisatieproblemen efficiënt oplost via Bayesiaanse optimalisatie. Onze aanpak is ontworpen om de dekking van ML-algoritme-monsters in de zoekruimte te maximaliseren, wat resulteert in een robuust surrogaatmodel voor BO. We hebben experimenten uitgevoerd op 117 benchmarkdatasets en ontdekt dat het gebruik van onze sampling benadering de prestaties van BO aanzienlijk verbetert in vergelijking met BO zonder onze aanpak. Naast

de voorgestelde sampling benadering hebben we ook een optimalisatiesoftware geïntroduceerd met de naam BO4ML in dit hoofdstuk. Deze software is geïmplementeerd op basis van de voorgestelde sampling benadering en kan worden gebruikt voor optimalisatie. Verder, in Hoofdstuk 8, hebben we een verbeterde AutoML-optimalisatietechniek voorgesteld, bekend als DACOpt, een efficiënte concurrerende procedure. Deze innovatieve methode beoogt het effectiever toewijzen van afstemmingsbronnen aan optimalisatoren over zoekgebieden. Specifiek beloont DACOpt middelen aan gebieden die belofte tonen, terwijl het optimalisatieproces wordt beëindigd in gebieden die minder waarschijnlijk gewenste resultaten opleveren. Onze experimenten op 117 benchmarkdatasets geven aan dat onze voorgestelde aanpak aanzienlijke prestatievoordelen biedt ten opzichte van BO voor AutoML-optimalisatieproblemen.

Acknowledgments

I express my deepest gratitude to my primary supervisor, Prof. Thomas Bäck, for his unwavering support and guidance throughout my academic journey. During the first years of my studies, when I found myself overwhelmed by challenges, Thomas provided me with reassurance and encouragement that became the driving force behind my perseverance. The simple yet profound words, "*Everything will be okay. We'll find a way*" were a beacon of hope during moments of doubt, and I am immensely grateful for their enduring impact. I would like to express my gratitude to my daily supervisor, Dr. Anna V. Kononova, for her outstanding work. She joined my PhD journey midway through and brought a fresh perspective and motivation that revitalized my research efforts. Her guidance not only helped me navigate the complexities of my work but also enabled me to develop ideas for publications and create helpful software for the community. Her encouragement to explore new avenues and her insightful feedback has been crucial in elevating the quality and impact of my work.

Special acknowledgments are also reserved for my industry supervisors, Prof. Bernhard Sendhoff and Dr. Stefan Menzel. I am grateful for their valuable guidance, insights, and contributions to my academic and research journey. Each supervisor has played a unique role in shaping my perspectives and enhancing my skills. Additionally, heartfelt gratitude is expressed to Dr. Hao Wang for his support and guidance in the early stage of my PhD. I am fortunate to have had the privilege of working under the guidance of Thomas, Anna, Bernhard, Hao, and Stefan. Their mentorship has not only contributed to the academic success of my PhD but has also left an indelible mark on my professional growth.

I want to express my sincere gratitude to my parents, Xuan Tai Nguyen and Thi Nghia Nguyen, for their unwavering support, constant presence, understanding, and love over the years. I would like to express my love for my wife, Hoang Yen Nguyen, and my son, Duc An Nguyen. I consider myself fortunate to have had

the privilege of having both of you by my side. Your constant encouragement and unwavering presence during challenging moments have been invaluable.

Last but not least, I want to express my sincere gratitude to my colleagues, Van Duc Nguyen, Sibghat Ullah, Jiawen Kong, and Diederick Vermetten. Their support, especially during moments when I felt discouraged, has been invaluable. Their words of encouragement and the camaraderie we share have been a constant source of motivation by my side. I feel fortunate to have such understanding and supportive colleagues, and I appreciate the positive impact they've had on our shared experiences.

In closing, I would like to extend my heartfelt thanks to all my supervisors, family, colleagues, and friends for their exceptional love, support, and encouragement. The impact of their contributions will be etched in my heart forever.

About the Author

Duc Anh Nguyen was born on May 20, 1987, in Thai Binh, Vietnam. He received his Master's Degree in Information Technology Management (MITM) from the International University - Vietnam National University Ho Chi Minh City in 2015. He started as a PhD candidate at the Leiden Institute of Advanced Computer Science (LIACS) - Leiden University, The Netherlands, in 2018. He worked there as an Early-Stage Researcher of the European research project ECOLE (Experience-based COmputation: Learning to optimisE), funded by the EU Horizon 2020 program. His research was performed in collaboration with Honda Research Institute Europe (HRI, D), NEC Labs Europe (D), and University of Birmingham (UK). His PhD study is under the supervision of Prof. Thomas Bäck, Prof. Bernhard Sendhoff (HRI), Dr. Anna V. Kononova and in cooperation with Dr. Hao Wang and Dr. Stefan Menzel (HRI). His research interests include Automated Machine Learning (AutoML) and optimization algorithms.