AutoML for Multi-Label Classification: Overview and Empirical Evaluation

Marcel Wever, Alexander Tornede, Felix Mohr, Eyke Hüllermeier *Senior Member, IEEE*

Abstract—Automated machine learning (AutoML) supports the algorithmic construction and data-specific customization of machine learning pipelines, including the selection, combination, and parametrization of machine learning algorithms as main constituents. Generally speaking, AutoML approaches comprise two major components: a search space model and an optimizer for traversing the space. Recent approaches have shown impressive results in the realm of supervised learning, most notably (single-label) classification (SLC). Moreover, first attempts at extending these approaches towards multi-label classification (MLC) have been made. While the space of candidate pipelines is already huge in SLC, the complexity of the search space is raised to an even higher power in MLC. One may wonder, therefore, whether and to what extent optimizers established for SLC can scale to this increased complexity, and how they compare to each other. This paper makes the following contributions: First, we survey existing approaches to AutoML for MLC. Second, we augment these approaches with optimizers not previously tried for MLC. Third, we propose a benchmarking framework that supports a fair and systematic comparison. Fourth, we conduct an extensive experimental study, evaluating the methods on a suite of MLC problems. We find a grammar-based best-first search to compare favorably to other optimizers.

Index Terms—automated machine learning, multi-label classification, hierarchical planning, Bayesian optimization

1 Introduction

Automated machine learning (AutoML) is commonly understood as the task of automating the process of engineering a "machine learning pipeline" specifically tailored to a problem at hand, that is, to a dataset on which a (predictive) model ought to be induced. This includes the selection, combination, and parameterization of machine learning (ML) algorithms as basic constituents of the pipeline, which is the main output produced by an AutoML tool, and which can then be used to train a concrete model on the dataset. Thus, compared to "basic" ML algorithms such as neural networks or support-vector machines, which solve a learning problem, an AutoML tool can be seen as solving a "learning to learn" problem. For the standard problem classes of single-label (binary or multi-class) classification (SLC) and regression, several such tools have been proposed in the last couple of years, and their performance has been demonstrated quite impressively in several experimental studies.

For various reasons, however, the empirical comparison of AutoML tools is a difficult endeavor and prone to incorrect interpretations. In particular, since an AutoML tool is a complex system consisting of several components, most importantly a search space model and an optimization method for traversing this space, one typically faces a credit assignment problem: If a tool performs well, and perhaps even better than others, what component is actually responsible for the improvement? For example, different tools (e.g., [1] and [2]) are typically using different search spaces, i.e., the space of ML pipelines they consider is not the same. While optimizing the search space, in general, is indeed a reasonable approach to improve the performance of an AutoML tool, it impedes the interpretation of evaluation results when a new approach to tackle the search task is proposed simultaneously. In such cases, it is often unclear where the improved performance comes from, the modification of the search space or the newly proposed search algorithm.

Going beyond standard (single-target) prediction problems, first attempts at extending AutoML toward multi-target problems [3] have been made in the last couple of years, most notably for the popular problem of multi-label classification (MLC) [4], [5], [6], [7], [8]. While the space of candidate pipelines is already huge in SLC, the complexity of the search space is raised to an even higher power in the case of MLC. This is mainly caused by more complex learning algorithms employed for the problem of MLC, which often perform as meta-algorithms on top of multiple existing SLC learning algorithms (e.g., one per label). An example of a potential structure of a multi-label classifier is depicted in Fig. 1. In fact, as we detail in Section 4, the MLC search space subsumes the SLC search space (several times). Furthermore, the evaluation of solution candidates takes significantly longer for MLC than for SLC algorithms due to their increase in structural complexity.

In light of this, one may wonder whether existing optimization methods for searching candidate pipelines, which have mainly been developed for SLC, are able to scale to the increased complexity of MLC search spaces, and how they compare with each other. Addressing this question in a systematic way, this paper makes the following contributions:

- First, we survey the state of the art, compare different approaches on a methodological level with respect

- *Marcel Wever, Alexander Tornede and Eyke Hüllermeier are with the Heinz Nixdorf Institute and Department of Computer Science, Paderborn University, Germany.*
  E-mail: {marcel.wever, alexander.tornede, eyke}@upb.de
- *Felix Mohr is with Universidad de La Sabana, Chía, Cundinamarca, Colombia.*
  E-mail: felix.mohr@unisabana.edu.co

Manuscript received April 19, 2005; revised August 26, 2015.

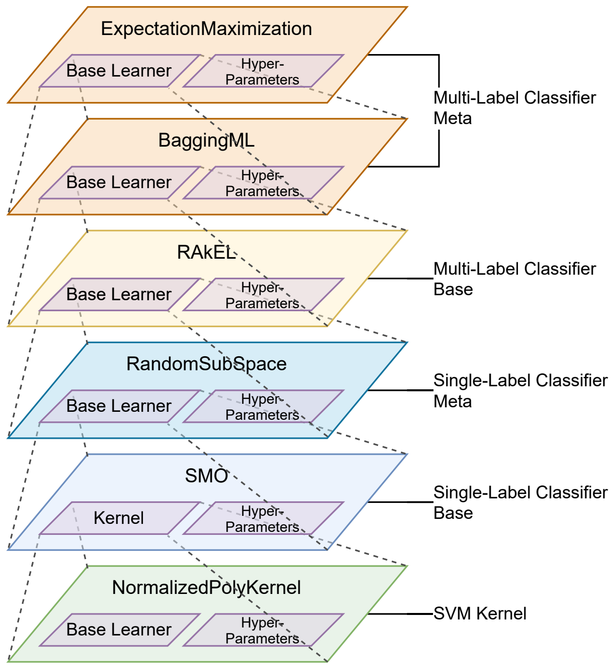

Fig. 1: Hierarchical representation of a multi-label classifier's structure being recursively configured with base learners and finally a kernel for the support vector machine (SMO, short for Sequential Minimal Optimization).

to their applicability to the MLC problem, and give an overview of existing approaches to AutoML for MLC, which are mainly characterized by the specification of the search space (Section 4).

- Second, we further augment these approaches by optimization methods that have not been tried for MLC so far, including Bayesian optimization, bandit algorithms, and hybrids thereof (see Section 5).
- Third, we propose a benchmarking framework that allows for a fair and systematic comparison (Section 6). Our framework ensures that all optimization methods adhere to the same runtime constraints, operate on equivalent search space models, and share the evaluation routine for solution candidates.
- Fourth, leveraging this framework, we conduct an extensive experimental study, in which we evaluate the methods on a suite of MLC problems (Section 7). In our experiments , we observe that all methods are visibly struggling with the tremendous size of the search space. However, a grammar-based best first search approach is found to perform best for the considered MLC search space, clearly outperforming the other optimizers.

Prior to elaborating on the main contributions of the paper as outlined above, we give a short introduction to AutoML (Section 2) and multi-label classification (Section 3).

2 Automated Machine Learning

Despite the short history of automated machine learning (AutoML), a diverse array of methods has been proposed to tackle the problem of so-called combined algorithm selection and hyper-parameter optimization (CASH), which was first stated in [9] and can formally be described as follows.

Let $\mathcal{A} := \{A^{(1)}, A^{(2)}, \dots, A^{(n)}\}$ denote a set of algorithms and $\Lambda^{(1)}, \Lambda^{(2)}, \dots, \Lambda^{(n)}$ the corresponding hyper-parameter spaces. Furthermore, let training (validation) and test data from a dataset space $\mathbb{D}$ be given by $\mathcal{D}_{\text{train}} = (X_{\text{train}}, Y_{\text{train}}) \in \mathbb{D}$ and $\mathcal{D}_{\text{test}} = (X_{\text{test}}, Y_{\text{test}}) \in \mathbb{D}$, as well as a target loss $\mathcal{L}$ to be minimized. The objective is now to find an algorithm $A^*_{\lambda^*}$ together with a suitable hyper-parameter configuration that generalizes well beyond the training data:

$$A^*_{\lambda^*} \in \underset{A^{(i)} \in \mathcal{A}, \lambda \in \Lambda^{(i)}}{\arg\min} \; \mathbb{E}\left[\mathcal{L}(Y_{\text{test}}, A^{(j)}_{\lambda}(X_{\text{test}}))\right]$$

In practice, however, the test loss is not accessible and thus approximated via the expected validation loss. To this end, the set of training data is again split into training data D'_{train} used for training and validation data $\mathcal{D}_{\text{val}} = (X_{\text{val}}, Y_{\text{val}})$ for validating the solution candidates' performance:

$$A^*_{\lambda^*} \in \underset{A^{(i)} \in \mathcal{A}, \lambda \in \Lambda^{(i)}}{\arg\min} \; \mathbb{E}\left[\mathcal{L}(Y_{\text{val}}, A^{(j)}_{\lambda}(X_{\text{val}}))\right]$$

The obtained estimate is then used for guiding the search for the best solution in the CASH problem.

Initial approaches reduced the CASH problem to a hyper-parameter optimization (HPO) problem by interpreting the choice of an algorithm as yet another hyper-parameter — a binary variable set to 1 if the respective algorithm is included in the pipeline — and concatenating those with the hyper-parameters of the respective algorithms to a single hyper-parameter vector. On the one side, such a reduction makes the original problem amenable to well-established tools for HPO such as SMAC [10] based on Bayesian optimization, Hyperband [11] based on a multi-armed bandit algorithm, or a combination of the two called BOHB [12]. In fact, by reducing AutoML to HPO and applying HPO tools, a variety of AutoML approaches have been proposed, including Auto-WEKA [9], auto-sklearn [1], hyperopt-sklearn [13], and Auto-Band [14].

On the other side, a reduction to HPO comes with the potential disadvantage of losing structural information due to "flattening" the search space. The structure of this space is naturally hierarchical, with a tree-like structure over the hyper-parameters. When using a flat, purely vectorial representation, parameter dependencies have to be captured in the form of additional constraints. For example, certain hyper-parameter configurations of a specific model might simply not be valid. Moreover, only those hyper-parameters belonging to selected algorithms are actually relevant or *active*, while all the others are irrelevant — information that is very important but not immediately accessible for the learner.

As an alternative to constraint-based vectorial representations, another branch of AutoML tools models the search space in a way that the hierarchical structure is maintained. Usually, these approaches rely on modeling solutions via a grammar that is used to derive valid candidates. This model can then be used for deriving (valid) individuals in (evolutionary) genetic programming [2], [15], [16]. Alternatively, such a grammar can also be used as a basis for deriving a search graph amenable to heuristic search algorithms, for example, a best-first search as in ML-Plan [17], [18] or a Monte Carlo Tree Search (MCTS) [19], [20].

Apart from the aforementioned tools, many other interesting techniques have emerged in the recent years, such as neural architecture search in general [21], tools with an emphasis on stacking [22], [23], leveraging reinforcement learning [24], or exploiting the potential of a random search for parallelization [25].

However, due to the rapid development, it is difficult to track the overall progress and understand the strengths and weaknesses of different optimizers and complete AutoML tools. In particular, newly proposed tools are often evaluated on different datasets and compared to a more or less randomly chosen subset of existing tools as baselines. This makes a global perception of the different AutoML tools and their performances very difficult. As another threat to comparability in empirical studies, new AutoML approaches are proposed as a combination of several components: optimization method, search space, and evaluation procedures (including timeouts, splitting for training, validation, and test data, performance measures) for assessing solution candidates. Due to this, performance gains or differences cannot be attributed to one particular change. Although there have been first steps in this direction [26], [27], an isolated large-scale comparison of the basic optimization strategies operating on an equivalent search space of a reasonable size is still an open issue. This is especially true for the problem domain of MLC.

3 Multi-Label Classification

Multi-label classification is a special type of multi-target prediction [3], where all the targets are binary variables encoding the "relevance" or the "irrelevance" of a specific aspect (identified by a label) for a data object (an instance). The main task in MLC is to learn a set-valued function that maps instances to subsets of (presumably) relevant class labels. As such, MLC can be seen as a generalization of standard multi-class classification, where an instance is assigned to exactly one class. As an example, consider the problem of image tagging: An image could be tagged with class labels `Sun` and `Beach` and `Sea` and `Yacht` at the same time. For a more comprehensive overview of multi-label classification, we refer to the survey articles [28] and [29].

3.1 Problem Setting

To formalize the MLC problem, let $\mathcal{X}$ denote an instance space and $\mathbb{L} = \{l_1, \ldots, l_m\}$ a finite set of m class labels. An instance $\boldsymbol{x} \in \mathcal{X}$ is (non-deterministically) associated with a subset of class labels $L \subseteq \mathbb{L}$. The subset L is often called the set of *relevant labels*. It is convenient to identify a set of relevant labels L with a binary vector $\boldsymbol{y} = (y_1, \ldots, y_m)$, where $y_i = 1$ if $l_i \in L$ and $y_i = 0$ otherwise.The set of all possible label combinations is denoted by $\mathcal{Y} = \{0,1\}^m$.

Formally, a multi-label classifier $\boldsymbol{h}$ is a mapping $\boldsymbol{h} : \mathcal{X} \longrightarrow \mathcal{Y}$. For a given instance $\boldsymbol{x} \in \mathcal{X}$ as an input, it outputs a prediction in the form of a vector

$$\boldsymbol{h}(\boldsymbol{x}) = \big(h_1(\boldsymbol{x}), h_2(\boldsymbol{x}), \ldots, h_m(\boldsymbol{x})\big) \enspace .$$

The task of inducing a multi-label classifier from data can be stated as follows: Given a finite set of observations

$$\mathcal{D}_{\text{train}} := (X_{\text{train}}, Y_{\text{train}}) = \big\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\big\}_{i=1}^{N} \subset \mathcal{X}^N \times \mathcal{Y}^N$$

as training data, the goal is to learn a classifier $\boldsymbol{h} : \mathcal{X} \longrightarrow \mathcal{Y}$ that generalizes well beyond these observations in the sense of minimizing the risk with respect to a specific loss function.

3.2 Loss Functions

A wide spectrum of loss functions has been proposed for multi-label classification, many of which are generalizations or adaptations of losses known for single-label classification. Generally speaking, these loss functions can be divided into three main categories: instance-wise, label-wise, and considering the label matrix as a whole (flattened to a single vector), which is also known as *micro averaging*. While instance-wise loss functions first compute a loss for every single test instance and then aggregate (average) over instances, label-wise loss functions compute a (binary classification) loss for each label and then aggregate the respective values across the labels. To be more specific, let $\mathcal{D}_{\text{test}} := (X_{\text{test}}, Y_{\text{test}}) \subset \mathcal{X}^S \times \mathcal{Y}^S$ be a test set of size S and $H = (\boldsymbol{h}(\boldsymbol{x}_1), \ldots, \boldsymbol{h}(\boldsymbol{x}_S)) \subset \mathcal{Y}^S$. Then, a loss function is a mapping $\mathcal{L} : \mathcal{Y}^S \times \mathcal{Y}^S \longrightarrow [0,1]$. In the following, we give three different ways of generalizing the F-measure to multi-label classification as instance-wise, macro averaging, and micro averaging loss functions that are commonly used in the literature.

Since the number of relevant labels is normally rather small (i.e., the label matrix is very sparse), the F-measure (which is actually not a loss function but a measure of accuracy, and thus to be maximized) has been adapted to the MLC setting in various ways. One possibility is to compute the F-measure for the predicted label vector of each instance in the test set first, and then aggregate across the instances; this is the instance-wise F-measure:

$$\mathrm{F}_I(Y_{\text{test}}, H) := \frac{1}{S} \sum_{i=1}^{S} \frac{2 \sum_{j=1}^{m} y_{i,j} h_j(\boldsymbol{x}_i)}{\sum_{j=1}^{m} (y_{i,j} + h_j(\boldsymbol{x}_i))} \tag{1}$$

Analogously, it can be defined in a label-wise manner:

$$\mathrm{F}_L(Y_{\text{test}}, H) := \frac{1}{m} \sum_{j=1}^{m} \frac{2 \sum_{i=1}^{S} y_{i,j} h_j(\boldsymbol{x}_i)}{\sum_{i=1}^{S} (y_{i,j} + h_j(\boldsymbol{x}_i))} \tag{2}$$

Finally, in a third variant, the F-measure can also be applied by so-called micro-averaging:

$$\mathrm{F}_\mu(Y_{\text{test}}, H) := \frac{1}{m \cdot S} \frac{2 \sum_{j=1}^{m} \sum_{i=1}^{S} y_{i,j} h_j(\boldsymbol{x}_i)}{\sum_{j=1}^{m} \sum_{i=1}^{S} (y_{i,j} + h_j(\boldsymbol{x}_i))} \tag{3}$$

Since the F-measure is the harmonic mean of precision and recall, good performance requires both a high true positive rate and a high true negative rate. In contrast to other commonly used MLC loss functions, such as the Hamming loss, the F-measure thereby addresses the problem of class imbalance and avoids an overly strong tendency toward negative predictions: too many negative predictions will yield a high precision but a low recall, and hence an overall low value for the F-measure. Nevertheless, depending on the variant used, the F-measure accounts for mistakes in the predictions in different ways, so that classifiers might be more appropriate for one and less for another version.

4 The Multi-Label Search Space

Taking standard (*aka* single-label) classification algorithms as a point of departure, multi-label classifiers have been developed in mainly two different ways: Either the multi-label problem is *transformed* into one or more single-label problems to which an existing algorithm can be applied, or an existing learning algorithm is *adapted* to the problem of MLC [30]. The latter essentially comes down to extending the algorithm so as to provide support for multiple labels in the algorithm structure. A simple example is the extension of decision tree learning from standard classification to multi-label classification [31].

4.1 Configuration of Multi-Label Classifiers

On one hand, the configuration of adapted learners such as neural networks with multiple output units, i.e., one per label, multi-target trees, or k-nearest neighbour learners works as in previous approaches and does not impose a particular challenge due to the multi-label classification setting. On the other hand, transformation techniques usually reduce the original MLC problem to a set of binary or multi-class classification problems, which can then be dealt with by known methods such as random forest, SVMs, logistic regression, etc. For example, binary relevance learning (BR) transforms it into a set of binary classification problems [32], one per label. These binary problems consist of predicting the relevance of the corresponding label independently of all other labels. While BR may look like a straightforward and efficient solution to the MLC problem, it is often criticized for ignoring interactions and statistical dependencies between class labels. Indeed, the idea of leveraging such dependencies to improve predictive performance is the main motivation of many multi-label learning algorithms. As an illustration, consider again the example, where the class label `Yacht` might be positively correlated with the class label `Sea`: If the former is positive, i.e., a yacht is on an image, then the latter is likely to be positive, too. Thus, while the predictions $(0,0)$, $(0,1)$, and $(1,1)$ appear completely plausible, a multi-label classifier should be more reluctant to predict $(1,0)$. As an example of a slightly more sophisticated (though still simple) transformation technique, let us mention *classifier chains* [33]. As suggested by the name, the classifier chain (CC) method trains predictive models in a sequential manner, sorting the labels along a chain. The basic idea is to condition the prediction of a label y_i, not only on the instance information $\boldsymbol{x}$, but also on the labels preceding y_i in the chain, which is specified by a permutation σ of $\{1,\dots,m\}$. Thus, starting with a model $\hat{y}_{\sigma(1)} = h_1(\boldsymbol{x})$, CC trains a second model $\hat{y}_{\sigma(2)} = h_2(\boldsymbol{x}, y_{\sigma(1)})$, a third model $\hat{y}_{\sigma(3)} = h_3(\boldsymbol{x}, y_{\sigma(1)}, y_{\sigma(2)})$, and so forth.

In the above example, for instance, CC may first predict the presence of `Yacht` based on properties of the image, and then additionally condition the prediction for `Sea` on the (predicted) presence or absence of a yacht on the image. In this way, label dependence could in principle be captured, at least to some extent. Yet, as a theoretical problem of CC, note that the label information used as additional features by the classifiers is only available for training but not at prediction time: Since the true label information $y_{\sigma(1)}$ cannot be used as an additional input, h_2 will actually deliver a prediction $\hat{y}_{\sigma(2)} = h_2(\boldsymbol{x}, \hat{y}_{\sigma(1)})$, replacing $y_{\sigma(1)}$ by the estimate $\hat{y}_{\sigma(1)}$ coming from h_1. Likewise, h_3 will predict $\hat{y}_{\sigma(3)} = h_3(\boldsymbol{x}, \hat{y}_{\sigma(1)}, \hat{y}_{\sigma(2)})$, etc. This creates a kind of attribute noise and possibly causes a problem error propagation along the chain [34].

Generally speaking, problem transformation methods can be seen as meta-learning methods, which need to be instantiated with a base learner, for example, a binary classifier in BR or CC. As already pointed out earlier, the structure of an MLC algorithm can thus become quite complex (cf. Fig. 1), requiring the user or ML engineer to make many decisions, e.g., choose up to 6 out of more than 70 algorithms, and configure up to 25 hyper-parameters simultaneously. Furthermore, empirical studies suggest that for optimizing the generalization performance of transformation methods, the choice of the base learner is indeed crucial [35], [36].

In addition to the selection and configuration of base learners, one may of course also think of parameterizing the meta-learner itself, thereby increasing the number of hyper-parameters even further. A simple example is the permutation σ in classifier chains, which is known to have a practical impact on performance [37].

Moreover, instead of choosing a single base learner to be used for each label, an individual base learner could be selected and tuned for each label separately. As shown in [36] for the case of BR, a label-wise configuration of that kind may indeed prove beneficial. Obviously, however, this will further increase the complexity of the configuration space by several orders of magnitude. Therefore, we stick to the simpler task of recursively selecting the base learners and tuning their hyper-parameters.

4.2 Search Space Description

The search space for multi-label classification considered here is shown in Fig. 2, comprising 5 different types of algorithms: meta and base algorithms for multi-label classification, meta and base algorithms for single-label classification, as well as kernels to be plugged into an SVM classifier (in the figure represented by the sequential minimal optimization algorithm; SMO). More precisely, the following algorithms are contained in the search space:

MEKA Meta MBR, SubsetMapper (SM), RandomSubspaceML (RSS), MLCBMaD (MLCBMD), BaggingML (BML), BaggingMLdup (BMLdup), EnsembleML (EML), EM, CM

MEKA Base BR, BRq, CC, CCq, BCC, PCC, MCC, PMCC, CT, CDN, CDT, FW, RT, LC, PS, PSt, RAkEL, RAkELd, BPNN, HASEL, MajorityLabelset (MLS), DBPNN

WEKA Meta AdaBoostM1 (ABM1), Vote (V), Stacking (S), LWL, RandomSubSpace (RSS), Bagging (B), RandomCommittee (RC), AttributeSelectedClassifier (ASC), AdditiveRegression (AR), ClassificationViaRegression (CVR), LogitBoost (LB), MultiClassClassifier (MCC)

WEKA Base J48, M5P, M5Rules (M5R), VotedPerceptron (VP), SimpleLinearRegression (SLR), SimpleLogistic (SL), NaiveBayesMultinomial (NBM), LMT, DecisionStump (DS),

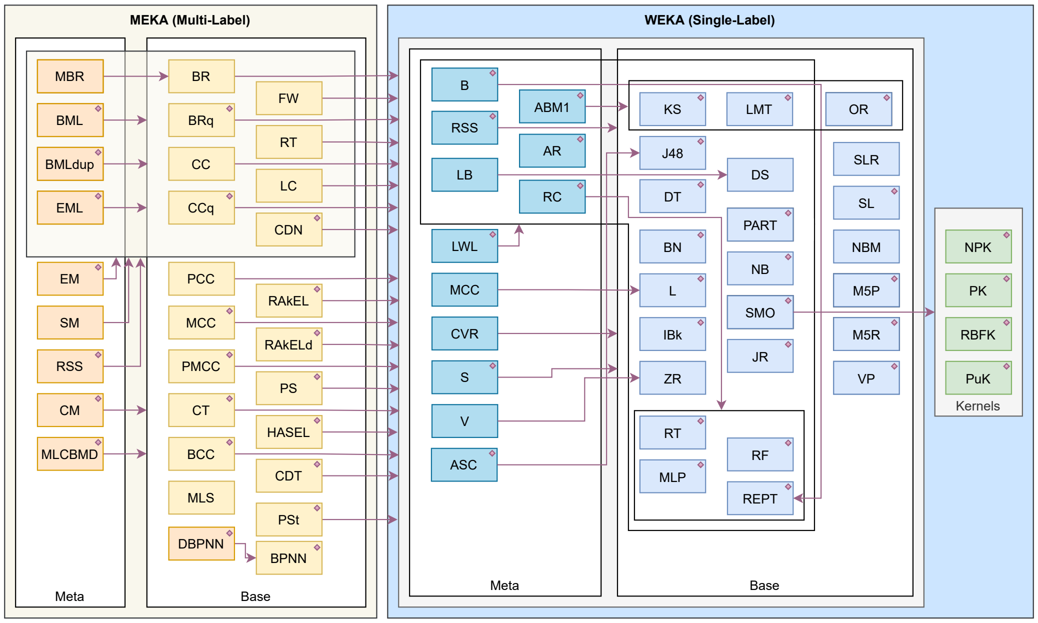

Fig. 2: Overview of the search space showing classification algorithms from MEKA for multi-label and WEKA for single-label classification. An arc pointing to a box frame means an arc to every classifier contained in this frame. Purple diamonds indicate whether the respective classifier exposes hyper-parameters to be tuned.

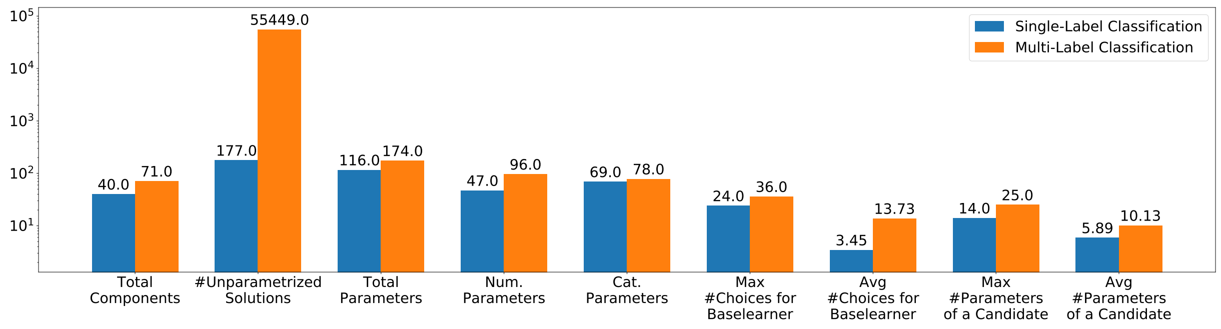

Fig. 3: Comparison of statistics regarding characteristics of the multi-label classification search space and the subsumed search space for single-label classification. Note that the there is a substantial increase in the number of unparameterized solution candidates, i.e., the number of distinct classifier configurations ignoring hyper-parameter configuration. Moreover, the maximum number of hyper-parameters that are optimized simultaneously for a single configuration is almost double the amount.

RandomForest (RF), RandomTree (RT), DecisionTable (DT), JRip (JR), OneR (OR), PART, ZeroR (ZR), IBk, KStar (KS), MultilayerPerceptron (MP), SMO, Logistic (L), NaiveBayes (NB), BayesNet (BN), REPTree (REPT)

Kernel NormalizedPolyKernel (NPK), PolyKernel (PK), RBFKernel (RBFK), Puk

From left to right, the algorithms typically require the configuration of a base algorithm, which can either be of the same type or the next type in the previously enumerated list. Within the figure, this requirement is indicated by an arc pointing either to a specific algorithm or a box containing several algorithms. The latter is a shortcut for drawing an arc from the respective algorithm to every algorithm contained in the box. Algorithms exposing hyper-parameters that need to be optimized are indicated by a purple diamond.

Fig. 2 provides a compact overview of the entire search space[1], such that the extension for AutoML from single-label to multi-label classification appears to only double the complexity, as only twice the number of algorithms is available. However, the real complexity lies in the need to configure base learners recursively, i.e., base learners of one method may require a base learner in turn to be configured. Therefore, the short cut arcs pointing from an algorithm to a box abstract most of the complexity.

A comparison of various statistics regarding the search spaces for single-label respectively multi-label classification is given in Fig. 3. While the number of algorithms (components) as well as the number of hyper-parameters defined in the search space increase only slightly, the size of the entire search space blows up from 177 unparameterized

1. A more detailed description including the hyper-parameters can be found in the GitHub repository: https://github.com/mwever/tpami-automlc

solution candidates to more than 55,000. However, not only the large number of distinct algorithm choices exacerbates the AutoML tasks, but also the maximum number of parameters a single solution candidate may expose. In the extreme case, a single solution candidate may expose up to 25 hyper-parameters, as compared to 14 in the case of single-label classification, but also the average number of hyper-parameters increases from 5.89 to 10.13.

In conclusion, compared to single-label classification, the multi-label classification search space itself contains considerably more solution candidates. Furthermore, due to more hyper-parameters that need to be optimized for a single candidate, the hyper-parameter optimization of the latter can be much more complex as well.

5 Optimization Methods

The literature on AutoML for standard classification and regression is rich of techniques that have been proposed for searching the huge space of solution candidates. However, for multi-label classification, only a few of these approaches have been considered so far. These include genetic algorithms [4], grammar-based genetic programming [5], hierarchical task network planning [6], [7], and a classifier specific approach based on neural architecture search [8]. Here, we focus on methods for classical AutoML dealing with the problem of combined algorithm selection and hyper-parameter optimization.

In the following, after a formal definition of the AutoML problem, we briefly outline various optimization approaches from the two branches of hyper-parameter optimization and grammar-based search. Moreover, we elaborate on how these methods can be applied to automating multi-label classification and whether this has already been done in the literature. For a more in-depth summary of the respective approaches, we refer the interested reader to survey papers on standard AutoML [38], [39], [40], [41]. In Fig. 4, an overview of the here considered optimization methods is given. Furthermore, we discuss to what extent these methods have already been considered in AutoML for single-label resp. multi-label classification. An overview of their use regarding standard AutoML and AutoML for multi-label classification is given in Table 1.

5.1 Reduction to Hyper-Parameter Optimization

A prominent way of tackling the AutoML problem is to reduce it to the problem of instance-specific hyper-parameter optimization. Here, one is given a hyper-parameter space Λ defined over multiple hyper-parameters, a dataset space $\mathbb{D}$ and a quality measure $u : \Lambda \times \mathbb{D} \longrightarrow \mathbb{R}$, stating how well a certain hyper-parameter configuration performs on a certain dataset. For a given dataset $\mathcal{D} \in \mathbb{D}$ the goal is to find the best hyper-parameter configuration $\lambda^*_{\mathcal{D}} \in \Lambda$ defined as

$$\lambda^*_{\mathcal{D}} = \arg\max_{\lambda \in \Lambda} u(\lambda, \mathcal{D}) \ . \quad (4)$$

In the context of AutoML, the quality measure u is usually a scoring or loss function such as the F-measure or the Hamming loss.

The reduction from the AutoML problem to hyper-parameter optimization is done by encoding the choice of each algorithm and its components via a categorical parameter for each choice. Each of these categorical parameters can take as many different values as there are choices for the respective algorithm or component. Hence, the result of the reduction is a single hyper-parameter vector consisting of these categorical hyper-parameters and the original hyper-parameters of each possible algorithm and component. Furthermore, many tools request a set of constraints, defining which hyper-parameters are connected to which algorithms and components. Thus, it becomes possible to leverage this information, e.g., by decomposing the vector into trees where only relevant hyper-parameters are considered.

5.1.1 Bayesian Optimization

Bayesian Optimization (BO) [52] is one of the most prominent techniques in the area of hyper-parameter optimization and the basis for the first approaches to AutoML [1], [9], [43]. On an abstract level, BO is an alternating process of building/updating a surrogate model $\widehat{u}$ inferred from observations of the (costly) measure u and leveraging the information contained in $\widehat{u}$ through a so-called acquisition function to choose the next candidate to be evaluated w.r.t u. This is repeated until a stopping criterion is met, e.g., wall-clock time or evaluations of u.

For AutoML tasks, typically Tree Parzen Estimators [53] or Random Forests [54] are employed as surrogate model. Although Gaussian Processes also represent a very natural choice for the surrogate model, they do not scale well with the high dimensional search space of the AutoML problem. In any case, the right choice depends on specifics of the optimization task, e.g., the structure and topology of the search space or the noisiness of u.

The surrogate model $\hat{u}$ is used in combination with an acquisition function to decide which hyper-parameter configuration to evaluate next with u. For the sake of efficiency,

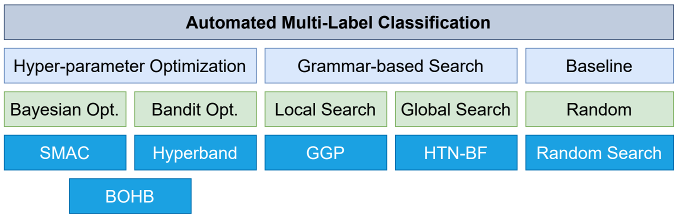

Fig. 4: Ontology showing the considered optimization techniques proposed for automating machine learning.

TABLE 1: Overview of optimization techniques considered in this paper for automating multi-label classification and an overview of whether and where these techniques have been employed for automating single-label resp. multi-label classification.

Method	AutoML SLC	AutoML MLC
Bayesian Optimization [42]	✓ [1], [9], [13], [43]	✗
Hyperband [44]	✓ [14], [45]	✗
Bayesian Optimization and Hyperband [12]	✓ [46]	✗
Genetic Algorithms [47]	✗	✓ [4]
Genetic Programming [48]	✓ [2], [15]	✓ [5]
HTN Planning [49]	✓ [17], [18], [50], [51]	✓ [6], [7]

this choice should reveal as much *useful* information about the search space as possible. Generally speaking, acquisition functions are a means to trade off exploration and exploitation so as to guide the search to promising candidates. To this end, not only the expected values (according to the surrogate model $\widehat{u}$) but also the uncertainty about these values are taken into account. While there are various functions of this kind, including entropy search [55], knowledge gradient [56], and expected improvement (EI) [57], [58], we focus on the latter, since it is mainly used in the field of AutoML.

The basic idea of EI is to sample the candidate that optimizes the improvement with respect to the best solution found so far. Formally, EI can be described with respect to a hyper-parameter configuration $\lambda_{\mathcal{D}}$ and the best hyper-parameter configuration $\lambda^*_{\mathcal{D}}$ found so far as

$$EI(\lambda_{\mathcal{D}}) = \mathbb{E}\left[\max(u(\lambda^*_{\mathcal{D}}, \mathcal{D}) - u(\lambda_{\mathcal{D}}, \mathcal{D}), 0)\right] . \quad (5)$$

Note that taking the expected value is required because $u(\lambda_{\mathcal{D}}, \mathcal{D})$ is a random variable with unknown outcome at the time of the computation of $EI(\lambda_{\mathcal{D}})$. Using this definition, the EI acquisition function chooses the configuration that maximizes EI.

BO has been employed as an optimization technique in several AutoML tools [1], [9], [13], [43] for tackling standard classification and regression tasks. However, to the best of our knowledge, it has not been used for tackling the AutoMLC problem before.

5.1.2 Hyperband

Another family of methods to tackle hyper-parameter optimization is based on formalizing the problem as a multi-armed bandit (MAB) problem, which is a sequential stochastic decision-making problem. The MAB agent (decision maker) selects one option at a time from a set of alternatives, also called "arms", and observes a numerical (and typically noisy) *reward* signal providing information on the quality of that option. The goal of the agent is to optimize an evaluation criterion such as the *cumulative regret*, i.e., the expected difference between the sum of rewards that could have been obtained by playing the best arm (defined as the one with the highest rewards on average) in each round and the sum of the rewards obtained while being challenged by the exploration-exploitation dilemma.

Hyper-parameter optimization can be cast as a MAB problem by considering each possible hyper-parameter configuration (or machine learning pipeline in the case of AutoML) as an arm. The rewards obtained when pulling an arm correspond to the evaluation of the corresponding configurations for a given budget, such as time, which is adapted over the course of the algorithm.

A classical naïve approach to finding a good arm (configuration) in such a setting is to allocate a total budget B equally to all K arms, i.e., pull each arm with a budget of $\lfloor {}^{B}/_{K} \rfloor$. While simple, this approach spends large amounts of the budget on non-optimal arms.

Successive halving [44], [59] mitigates this flaw by dividing the time steps into N brackets, allocating the budget equally across the brackets and halving the number of arms to be pulled at the end of each bracket. Based on the rewards obtained, the best half of the arms are kept and promoted

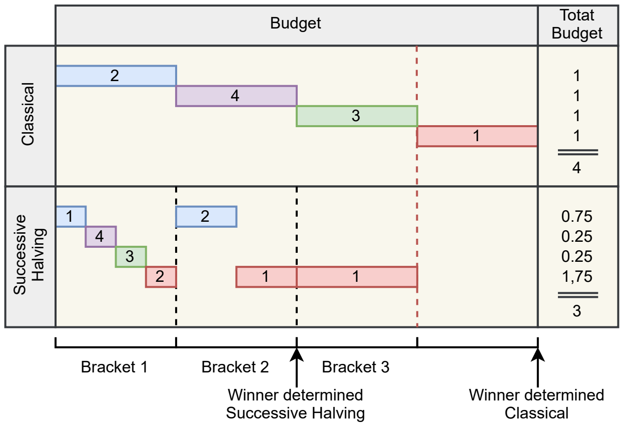

Fig. 5: Comparison of the classical approach (top) and successive halving (SH) (bottom) to identify the best performing configuration out of 4 candidates. Numbers within colored rectangles indicate the rank of a configuration. Within each bracket, the current set of configurations is evaluated on a portion of the totally assignable budget and after each bracket the worse half drops out. After bracket 2, SH already identified the winner configuration (red). The right column summarizes the total budget spent per configuration.

towards the next bracket resulting in a single final arm after $\lceil \log_2(K) \rceil - 1$ brackets. The success of this strategy in selecting the truly best arm heavily relies on the assumption that discarding arms based on low-budget evaluations does indeed correctly discard the bad configurations, but not those that may only show their potential when being evaluated on larger budgets. A visual comparison of the two approaches with $K = 4$ arms and $N = 3$ brackets in the case of SH is presented in Figure 5.

In the context of hyper-parameter optimization, the budgeted resource can vary, but common choices are the number of iterations for evaluating the configuration [44], the computation time for evaluating the configuration, the size of the subsampled dataset or the subsampled feature set on which the configuration is evaluated [11]. Here, we make use of the number of folds of a Monte Carlo cross-validation (MCCV) as budgeted resource, i.e., we present evaluation results based on one or more iterations to the optimization approach to allow for low fidelity optimization.

However, the set of hyper-parameter configurations, and hence the number of arms in the associated MAB, is typically extremely large or even infinite. The authors of [44] solve this problem by sampling a predefined number of configurations before SH is invoked, presenting thus only a finite set of arms to the algorithm while still covering the underlying space sufficiently well.

As shown in [11], the size of the set of configurations K presented to SH greatly influences the choice of the final arm. This is because picking too few configurations might lead to missing good ones but also offers the selected configurations more budget, whereas too many configurations may contain good ones but lead to less budget, which in turn might lead to wrong rejections (exploration-exploitation dilemma). Hyperband is a heuristic for choosing initial set sizes and repeatedly applying SH to finally return the best solution found in this process.

More precisely, Hyperband iteratively calls SH with different numbers of hyper-parameter configurations K and assigns a minimum budget to each of these configurations before any of them is discarded. The adaptation of K is based on a maximum budget to be allocated to a single configuration and the proportion of configurations to be discarded in each bracket of SH. Doing so, Hyperband gradually moves from exploration to exploitation by decreasing the amount of initial configurations while receiving a single final solution with each call of SH. Finally, the best configuration found during this process is returned.

Hyperband has been applied to AutoML for classification in [14]. Yet, to the best of our knowledge, it has not been used to tackle the AutoMLC problem so far, which will be done in this work for the first time.

5.1.3 *Bayesian Optimization and Hyperband (BOHB)*

An obvious weakness of Hyperband is its random sampling of configurations at the beginning of each iteration, which is addressed by an approach combining the idea of Hyperband with Bayesian Optimization, called BOHB [12]. More specifically, it replaces the random sampling procedure of Hyperband by BO-based sampling. TPE models are constructed for different budgets B based on observed configuration performances. In each iteration, the majority of configurations are iteratively sampled using these models, while the remaining configurations are sampled at random for reasons of convergence. As one is eventually interested in the performance of a configuration evaluated on the maximum budget, BOHB always queries the model associated with the largest budget available.

BOHB can be instantiated to solve AutoML problems in the same way as SMAC and Hyperband, namely by reducing the AutoML problem to a problem of hyper-parameter optimization. Once again, to the best of our knowledge, this work is the first one to apply BOHB for tackling the AutoMLC problem, although it has been used in the context of AutoML for classification before [46].

5.1.4 *Genetic Algorithms*

Genetic algorithms (GAs) are quite popular and frequently used as a tool for black-box optimization. The basic idea is to maintain a population of candidate solutions and to refine these candidates iteratively by applying randomized operators (e.g., mutation and cross-over inspired by biological evolution) with the aim of maximizing a given fitness function. Each of the candidate solutions is encoded by a fixed-size binary or real-valued vector of so-called *genes*, also referred to as a genetic representation.

Applying genetic algorithms to the problem of AutoML thus requires a proper genetic representation, which can be obtained by encoding every hyper-parameter by a single gene (using integers for categorical or integer hyper-parameters, and reals for any other numeric hyper-parameters). However, such an encoding is difficult to handle for standard GAs, because most of the genes are "inactive" in the sense of not belonging to the currently selected algorithm(s). This also hinders the exchange of parts of the current solution. Alternatively, messy GAs can be used but the mutual exchange of individuals remains difficult [60]. These issues may explain why standard GAs have not been considered very much in the AutoML literature.

To the best of our knowledge, only a simple GA called GA-Auto-MLC has been used for the problem of automating multi-label classification [4]. However, only a very small selection of algorithms has been considered in this work, which is mostly due to the chosen genetic representation. To compress the genetic representation, the genes for hyper-parameters were shared among different algorithms. More specifically, the number of genes for hyper-parameters was chosen according to the method exposing the highest number of hyper-parameters. The values encoded in the genes are then interpreted with respect to the selected method and the remaining information is ignored.

Later on, a detailed ablation study [5] revealed that a grammar-based genetic programming approach can outperform such a simple genetic algorithm for the same search space. These findings can be attributed to the more suitable genetic representation. Furthermore, the genetic programming approach is even more flexible and allows for a larger portfolio of algorithms. Because of these results, we exclude GA-Auto-MLC from our study.

5.2 Grammar-based Search

Grammar-based search approaches have emerged as another line of research for designing AutoML tools (cf. [2], [5], [16], [17]). In contrast to reduction techniques representing the optimization space by a (flat) vector of hyper-parameters combined with additional conditions, grammar-based formalisms allow for modeling the hierarchical structure of machine learning pipelines and classifiers more naturally. This hierarchical structure is particularly prominent in the case of multi-label classifiers, which usually employ single-label classifiers as a base learner. Yet, it is also inherent to single-label classifiers, as shown by examples like a bagged ensemble of support vector machines, which in turn require a kernel function to be specified. In the following, we describe two representatives of grammar-based approaches, first an evolutionary approach for evolving tree-shaped structures called grammar-based genetic programming (Section 5.2.1), and then a technique from the field of AI planning dubbed hierarchical task network (HTN) planning (Section 5.2.2).

5.2.1 *Grammar-Based Genetic Programming*

Just like genetic algorithms, grammar-based genetic programming (GGP) algorithms belong to the family of evolutionary algorithms. Yet, in contrast to standard GAs, GGPs make use of a grammar to describe the correct syntax of individuals. This syntax is used to generate an initial population of valid individuals, and also provided to genetic operators that are specifically crafted for GGP. Another difference to standard GAs is the genetic representation. Instead of representing individuals in terms of fixed-length vectors of genes, they are described in the form of trees describing derivations of the grammar, which makes the entire approach more flexible with respect to more complex structures and larger portfolios of algorithms. Furthermore, the size of such a tree does not necessarily need to be fixed or upper bounded. For a more comprehensive description of grammar-based genetic programming, we refer the interested reader to [48].

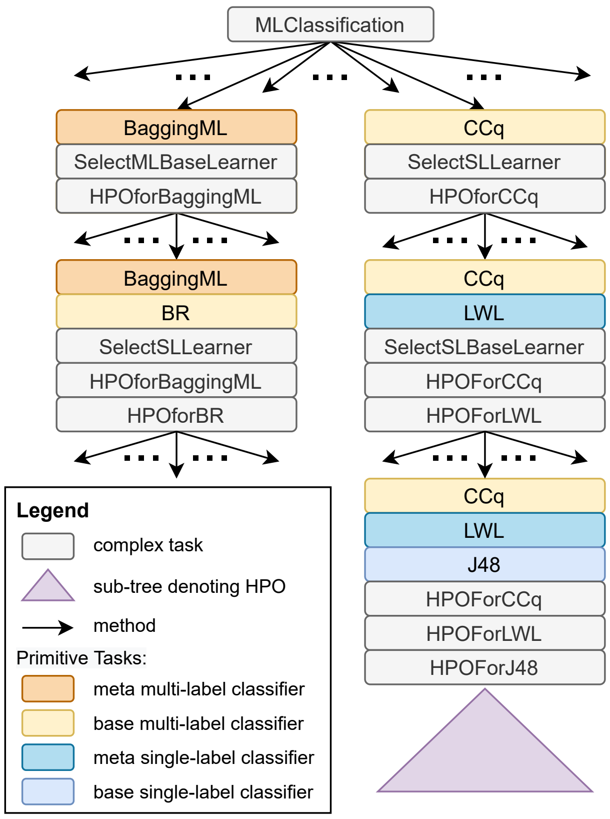

Fig. 6: Sketch of a search tree induced via HTN planning for automated multi-label classification. Primitive tasks are additionally distinguished by color according to their role within a multi-label classifier. Note that the indicated refinements are of exemplary character. Further options as well as sub-trees are only hinted at.

Due to their appealing properties, GGPs have been used to tackle the AutoML problem in various ways [2], [15], [16]. All these approaches have in common that the search space is described by a context-free grammar, structuring the space in a hierarchical way and having algorithm names and hyper-parameter values as terminals. Prominent examples of applying GGP to AutoML for single-label classification or regression are TPOT [2], RECIPE [16], and GAMA [15].

Even more interestingly, GGP provides the basis of an AutoML tool for multi-label classification called Auto-MEKA$_{\text{GGP}}$ [5]. However, from a methodological point of view, nothing has been implemented in Auto-MEKA$_{\text{GGP}}$ that could be considered as specific for MLC, except for the evaluation of multi-label classifiers. In particular, the search space is described in the same way (extended by descriptions for multi-label classifiers) as before.

5.2.2 HTN Planning and Best-First Search

The basic idea of Hierarchical Task Network (HTN) planning [49], a technique from the field of automated planning, is to hierarchically structure the space of possible solutions based on a logic language and specific operators. To this end, HTN planning describes the search space in terms of complex tasks, primitive tasks, and methods that specify how complex tasks are refined again into complex tasks or primitive tasks. While primitive tasks are considered atomic and usually represent something that can be "executed", complex tasks can be viewed as a composition of simpler tasks and thus need to be decomposed recursively. Intuitively, HTN planning mimics, e.g., the way a machine learning expert approaches a multi-label classification task, decomposing it into smaller and simpler tasks such as selecting classifiers, base learners, and eventually tuning the hyper-parameters [61]. A "ground" solution, also referred to as a *plan*, is obtained once all complex tasks are fully refined and only primitive tasks are left.

The idea is similar to derivations in context-free grammars, where complex tasks are non-terminal symbols and primitive tasks are terminals. In contrast to context-free grammars, primitive tasks do not only work in a generative manner, but can also modify a (logical) state, a concept featured in HTN.

HTN problems are typically solved by a reduction to a graph search problem that can be approached with standard algorithms, e.g., depth-first search. A typical translation of the HTN problem into a graph is to select the *first* complex task of a list and to define one successor for each applicable method that can be used to refine the task; this is called *forward-decomposition* [49]. As a consequence, the shape of the resulting search graph is a tree. While leaf nodes of the tree represent plans, an inner node represents a prefix of a plan. Hence, the root node is an empty plan.

HTN planning has been instantiated for automating data mining and machine learning by mapping primitive tasks to algorithm choices and the configuration of hyper-parameter values and building an abstract structure over these choices by means of complex tasks [17], [62]. The graph in Fig. 6 sketches an excerpt from such a search graph for the automated multi-label classification problem. In [17], a best-first search is applied to the resulting search graph. As a heuristic, the proposed best-first search assigns scores to inner nodes by randomly drawing several path completions to leaf nodes in order to obtain fully-specified pipelines that can be evaluated as usual, e.g., applying cross-validation. The score of the inner node is determined by the best completion to bound the true optimum that can be found in the respective sub-tree (assuming the objective function to be minimized). By configuring the number of random completions drawn for assessing the quality of an inner node in terms of an approximate score, we can trade-off the degree of exploitation and the degree of exploration of the search.

In analogy to AutoML for single-label classification, we can instantiate HTN planning combined with a best-first search for the MLC setting. Extending the search space, tuning the search and the evaluation strategy to the specifics of the MLC search space, extensions of [17] have been proposed in [6], [7].

6 AutoMLC Benchmark

In empirical AutoML studies, multiple components are often changed at a time without carrying out ablation studies. For example, different optimizers with different search spaces are compared, sometimes even with different candidate evaluation methods. One quite frequent example is to propose a new optimization technique together with a different search space, while not changing the search space for the baseline methods considered for comparison. In such cases, the results of the studies are difficult to interpret. Regardless of whether the newly proposed method is superior, competitive, or inferior to the baselines, it is not clear whether this finding should be attributed to the change of the search space or the optimization method.

The general issue has already been acknowledged in the literature [26], where AutoML tools are evaluated within consistent hardware and timeout environments as well as optimized for the same target loss function. However, the compared AutoML methods are considered a black box and the design of the search space is considered a part thereof. As a consequence, the latter differs from approach to approach. Therefore, it is unknown whether performance differences between AutoML methods can be attributed the optimization techniques or to the search space definition.

Note that the definition of the latter has a huge impact on the problem complexity. Even small changes may simplify the problem a lot or, on the contrary, make it much harder. Extending the search space by a single ensembling algorithm, comprising an arbitrary list of base learners, may increase the size of the search space from finite to infinite. Likewise, removing a single algorithm from the search space can lead to a significant simplification of the optimization task, but of course, also imply that the best algorithm for a particular task is no longer available. The question of which optimizer may perform best in which setting is thus still an open question.

In [39], the authors attempt to answer the question considering different optimizers for the same search space and even the same internal evaluation procedure. However, the approach taken in [39] is limited in several regards:

- It is restricted to optimizers available in Python, whereas the benchmark proposed here features cross-platform capabilities.
- The search space only considers a flat set of algorithms to be chosen, i.e. the optimizers are allowed to choose out of 13 different classifiers and activate hyper-parameters to be optimized according to this choice. Although there is a notion of parameters being configured in a hierarchical way in the case of SVMs, the search space definition has no concept for refining base learners, e.g., of ensembles.
- Furthermore, the runtime of the optimizers is indirectly limited via the number of evaluations, which in turn is bounded by a maximum of 10 minutes per evaluation. However, the limitation on the number of evaluations unnecessarily penalizes optimization strategies that prefer to extensively examine candidates with a very short runtime. While the number of evaluations is a proper means to ensure comparability in the realm of black-box function optimization, the solution candidates in AutoML are occasionally too diverse. In our experimental evaluation, we provide empirical evidence for the high variance of the evaluation times for different solution candidates.

Generally speaking, a common benchmark is desirable since AutoML studies are expensive in terms of time and computational resources. With each newly proposed method, the corresponding studies repeatedly execute multiple other methods and baselines. This is necessary, first because experimental setups, i.e., time constraints, assigned hardware resources, target functions, and datasets, are altered, and second, there is no common benchmark ensuring compatibility of experimental results. Moreover, common benchmarks are useful to streamline research, ensuring comparability of the evaluations of new methods to already existing ones and ideally enforce separation of concerns.

As the line of research on AutoML for multi-label classification is still in its infancy, we propose a unified framework for benchmarking methods and extensions for AutoML in the problem domain of MLC to ensure comparability across different optimizers (across different platforms) and to avoid unnecessary re-evaluations of already published methods in the future. Moreover, it forms a basis for future research on both refining the MLC search space and refining optimization techniques to cope with the more complex search space. An overview of the framework is sketched in Fig. 7. The key features of the framework are shared run constraints, a model-to-model transformation for search space descriptions, and a shared (cross-platform) performance evaluation procedure.

The framework is organized into two parts. First, the benchmarking setup (blue part of the figure) contains the technical specifications, i.e., the global run constraints, search space description, and the performance estimation procedure. Second, the interface of the optimizer (green part of the figure), which is responsible for translating the setup information into a format manageable by the specific optimizer and providing a stub that can be called to query the performance estimation procedure.

As an aside, except for its concrete instantiation, nothing of the framework is task-specific (regarding multi-label classification). Therefore, the benchmark framework could in principle be used to achieve comparability of different optimization techniques for any other AutoML task, too. For example, the benchmark could be used to investigate the capabilities of different optimizers to search for standard classification machine learning pipelines including (multiple) pre-processing algorithms. However, as we focus on automated machine learning for multi-label classification here, this kind of investigation is out of the scope of this paper and left for future work.

6.1 Benchmarking Setup

The benchmarking setup encapsulates all the parameters relevant to an AutoML benchmark, except for the optimizer that is used to explore the space of potential solution candidates. More precisely, the benchmarking setup defines the entire experimental setup, including constraints on the run defining the degree of parallelization and the timeouts. The framework allows for defining timeouts for both the entire AutoML process and the evaluation of a single candidate independently.

The core part of the benchmarking setup is the search space description, which specifies all potential solutions that may be tested by the algorithms. Our benchmarking environment comes with its own (JSON-based) language to describe a search space, which is easy to read and edit,

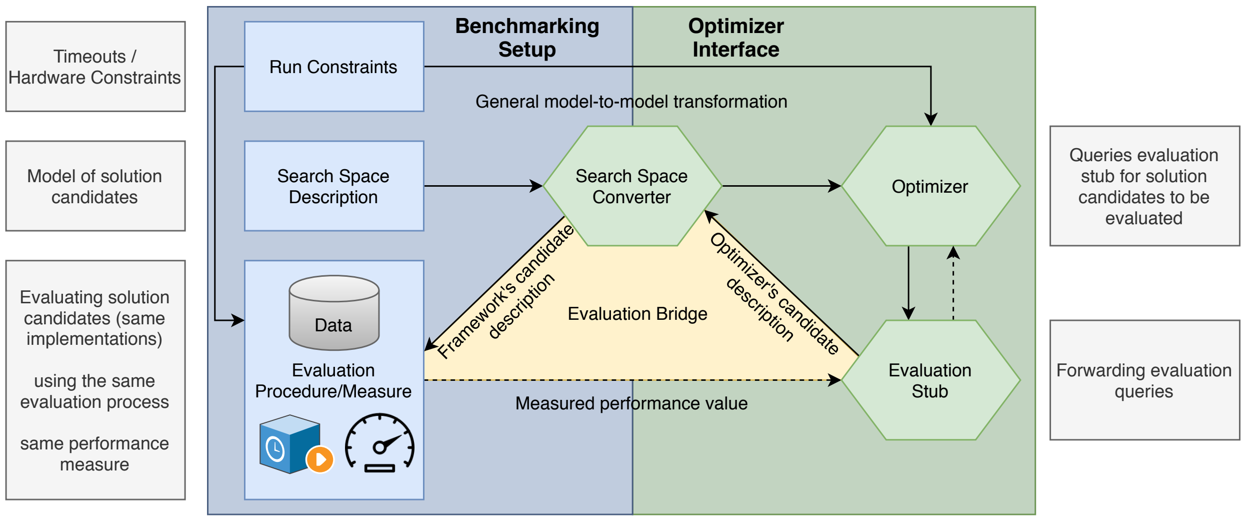
Timeouts / Hardware Constraints
Model of solution candidates
Evaluating solution candidates (same implementations)
using the same evaluation process
same performance measure
Benchmarking Setup
Optimizer Interface
Run Constraints
General model-to-model transformation
Search Space Description
Search Space Converter
Optimizer
Data
Evaluation Procedure/Measure
Framework's candidate description
Evaluation Bridge
Optimizer's candidate description
Evaluation Stub
Measured performance value
Queries evaluation stub for solution candidates to be evaluated
Forwarding evaluation queries

Fig. 7: Architecture of the benchmark for comparing different optimizers for the same run constraints, search space, and evaluation procedure. Blue parts are commonly used for all approaches, while green parts are specific to the respective optimizer, marshaling the description of candidate solutions for both the search space description and the description of candidates to be evaluated.

and which allows for modeling search spaces maintaining hierarchical structures. In this model, every algorithm is seen as a *software component* with provided and required *interfaces*. The interfaces are just names and have no functional specification. For example, a binary relevance learner provides an interface `MultiLabelClassifier` and requires an interface `BaseLearner`, which in turn can be provided, for example, by an SVM. For every component, one can define a set of parameters with their domains and dependencies among them, e.g., "if value of $x = 3$, then the *domain* of possible values for y becomes $[0, 1]$".

To make this search space description understandable to the different optimizers, a search space converter must be written for every optimizer to be considered in the benchmark. Clearly, every optimization tool accepts *some* form of search space description, but the concrete formats strongly vary among the different optimizers. For this paper, we implemented such converters for the considered optimizers to configure correct inputs for these optimizers.

Second, the run constraints comprise timeouts and computational resources. More precisely, one defines the overall timeout for the search process, the timeout for single evaluations, and constraints on memory and CPU usage. Needless to say, the concrete choice of timeouts can be more or less beneficial for an optimizer. However, since the same constraints apply to all optimizers, this impact should not be too large.

The third and last part of the benchmarking setup concerns the evaluation procedure, and thereby also the performance measure, which serves as the target loss to be optimized. Sharing this part of the benchmarking setup across the different optimizers ensures that there is no advantage in terms of evaluation speed, which might distort the overall performance. Usually, to ensure this kind of fairness, the number of allowed evaluations is limited. Our approach guarantees the same degree of fairness also for anytime settings.

In addition to ensuring fairness and comparability, an advantage of decoupling the benchmarking setup from the optimizer is to develop meta-learning approaches independent of a concrete optimizer. For example, a surrogate for assessing the performance of a solution candidate can be used by substituting the evaluation procedure. In this way, the surrogate can be tested in combination with any optimizer implemented within the framework. Furthermore, the framework allows for task-specific adaptations of the search space, e.g., by anticipating which algorithms will likely be too time-consuming for a chosen evaluation timeout and excluding these algorithms right from the start. Only the reduced search space is then provided to the optimizer.

6.2 Optimizer Interface

The optimizer interface is responsible for connecting an optimizer to the rest of the benchmarking framework. More specifically, this mainly concerns setting the hyperparameters of the optimizer and converting the search space description from the framework's format into the specific format of the optimizer.

In addition to the optimizer itself, the optimizer interface contains an evaluation stub bridging between the optimizer and the evaluation procedure that is part of the benchmarking setup. The evaluation stub takes evaluation requests from the optimizer and forwards them to the evaluation procedure. If the evaluation of the respective solution candidate is successful, the evaluation stub will feed the result value back to the optimizer. Of course, the optimizer and the evaluation stub are agnostic about the loss function used to calculate the return value. However, in the case of an unsuccessful evaluation, the evaluation procedure gives feedback regarding the cause and differentiates between crashed evaluations and those with a timeout.

The third component of the optimizer interface is a mapping from the framework's search space description format into the specific format of the optimizer. By automatically generating search space descriptions, only the model-to-model transformation needs to be correct, which simplifies maintenance and allows for considering different search spaces in a consistent way across multiple optimizers.

7 Experimental Evaluation

The experimental evaluation analyzes the performance of the optimization strategies for AutoML introduced above in the problem domain of multi-label classification. We investigate the scalability of the optimizers alone concerning the increased search space complexity, resulting from the deeper hierarchical structures of multi-label classifiers and the more costly candidate evaluations. To this end, we apply the benchmarking framework as proposed in Section 6, making sure that all optimizers are operating on the same search space and adhere to the same constraints in terms of hardware resources and timeouts.

7.1 Experimental Setup

In our experimental evaluation, we carry out all experiments in the proposed benchmarking framework considering the following optimization methods:

- Bayesian optimization (SMAC)
- Bandit optimization (Hyperband; HB)
- Bayesian Optimization & Hyperband (BOHB)
- Grammar-based genetic programming (GGP)
- HTN planning and best-first search (HTN-BF)

Additionally, as a primitive baseline, we run a random search that samples algorithm selections uniformly at random (including recursive dependencies on other algorithms) and subsequently chooses the hyper-parameters of the selected algorithms uniformly at random from the respective hyper-parameter domains.

All the runs were executed on nodes equipped with 8 CPU cores (Intel Xeon E5-2670) and 32GB of main memory with an overall timeout of 24h and a timeout for evaluating a single classifier of 30 minutes. For the performance estimation of a solution candidate, we used 5 randomly generated train/validation splits with 70% training and 30% validation data of the "training" data provided for the AutoML run. Moreover, we used three different performance measures as target function: instance-wise F-measure (F_I), label-wise F-measure (F_L) and micro-averaged F-measure (F_μ).

The best-first search was configured with the default configuration proposed in [17], i.e., it samples 3 random path completions for assessing the quality of a node, resulting in a relatively greedy search behavior. As for SMAC, we used its parallelized version, but otherwise the default parameterization. Furthermore, we allowed for multi-fidelity optimization by letting Hyperband and BOHB choose how many train and validation splits are used for estimating the performance of a solution candidate. To this end, they were configured to choose budgets b ranging from 1 to 5000 (to also allow for enough exploration as the budget limits also determine how many candidates are explored), which was translated to $\lceil b/1000 \rceil$ train and validation splits.

The grammar-based genetic programming approach was configured to operate on a population size of 15, as in the default configuration of Auto-MEKA$_{\text{GGP}}$. The probabilities for applying cross-over and mutation for recombination of individuals were set to 0.9 and 0.1, respectively. Each new generation keeps the best individual of the last generation. In contrast to Auto-MEKA$_{\text{GGP}}$, our implementation of grammar-based genetic programming does no reshuffling of train and validation splits but only uses the performance estimation procedure provided by the benchmarking framework as a fitness function. Moreover, the algorithm was used in an anytime setting, i.e., it can return a solution as soon as a first successful candidate evaluation was done, and continues the evolution as long as time is left.

Train and test splits are derived by 10-fold cross-validation, resulting in 10 train and test splits for each dataset. A list of the datasets used for benchmarking together with some descriptive statistics is given in Table 2. The descriptive statistics include the number of instances (#I), the number of labels (#L), the label to instance ratio (L2IR), the unique labeling combinations (ULC), and the average number of labels assigned to an instance (*aka* label cardinality).

TABLE 2: Benchmark datasets used in this study. The datasets are described by their name, number of instances (#I), number of labels (#L), the label-to-instance ratio (L2IR), the portion of unique label combinations (ULC), and the average label cardinality (card.).

Dataset	#I	#L	L2IR	ULC	card.
arts1	7484	26	0.0035	0.08	1.65
bibtex	7395	159	0.0215	0.39	2.40
birds	645	19	0.0295	0.21	1.01
bookmarks	87856	208	0.0024	0.21	2.03
business1	11214	30	0.0027	0.02	1.60
computers1	12444	33	0.0027	0.03	1.51
education1	12030	33	0.0027	0.04	1.46
emotions	593	6	0.0101	0.05	1.87
enron-f	1702	53	0.0311	0.44	3.38
entertainment1	12730	21	0.0016	0.03	1.41
flags	194	12	0.0619	0.53	4.12
genbase	662	27	0.0408	0.05	1.25
health1	9205	32	0.0035	0.04	1.64
llog-f	1460	75	0.0514	0.21	1.18
mediamill	43907	101	0.0023	0.15	4.38
medical	978	45	0.0460	0.10	1.25
recreation1	12828	22	0.0017	0.04	1.43
reference1	8027	33	0.0041	0.03	1.17
scene	2407	6	0.0025	0.01	1.07
science1	6428	40	0.0062	0.07	1.45
social1	12111	39	0.0032	0.03	1.28
society1	14512	27	0.0019	0.07	1.67
tmc2007	28596	22	0.0008	0.05	2.16
yeast	2417	14	0.0058	0.08	4.24

In total, we carried out 720 runs for each method, except for random search, which we executed only for 240 runs to reduce computation costs. As random search does not make any decisions based on candidate solutions seen so far, we only need one run for all the three target losses together. Each of the methods is executed with 8 parallel workers. Summing up to a total of 3,840 experiments à $24h$, the experimental evaluation contains data worth approximately 84 CPU years ($= 3,840 \times 24h \times 8$ cores $= 737,280$ CPUh).

To specify the search space, we considered the multi-label classifiers provided by MEKA [63], a multi-label classification extension of the well-known WEKA [64] machine learning library. Both libraries are implemented in Java, which is one reason why our benchmarking framework is implemented in Java, too. For the global model of the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPAMI.2021.3051276, IEEE Transactions on Pattern Analysis and Machine Intelligence

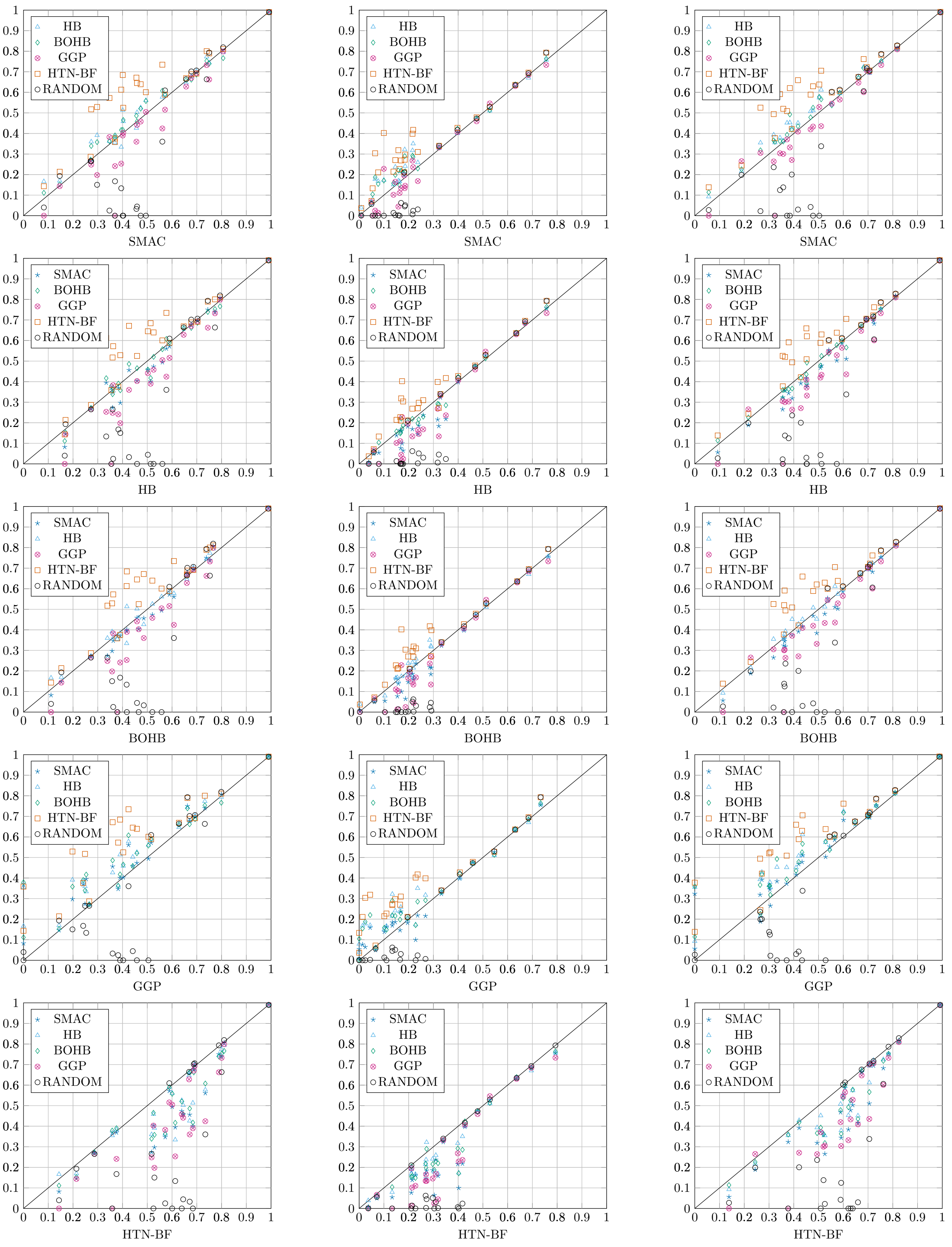

Fig. 8: Pair-wise comparison of one method (shown on the x-axis) against all other methods with respect to instance-wise F-Measure (left), label-wise F-Measure (center), and micro-averaged F-Measure (right).

TABLE 3: Test performances (mean ± std) of the considered approaches. Best performances are highlighted in bold, whereas results not significantly worse than the best performance are underlined. Average ranks across the datasets are given at the bottom of each part for the respective performance measure.

	Dataset	SMAC	HB	BOHB	GGP	HTN-BF	Random
Instance-Wise F-Measure (F_I)	arts1	.27±.05	.36±.10	.34±.09	.25±.01	**.52±.03**	.27±.09
	bibtex	.37±.07	.38±.06	**.39±.03**	.24±.02	.38±.02	.17±.06
	birds	.27±.04	.27±.04	.27±.04	.27±.04	**.29±.04**	.27±.04
	bookmarks	.08±.05	**.17±.08**	.11±.08	.00±.00	.14±.03	.04±.06
	business1	.74±.02	.77±.02	.75±.01	.73±.01	**.80±.02**	.67±.22
	computers1	.46±.05	.50±.06	.46±.02	.44±.01	**.65±.02**	.04±.13
	education1	.30±.07	.39±.11	.36±.10	.20±.10	**.53±.01**	.15±.15
	emotions	.68±.05	.68±.04	.66±.05	.67±.03	.69±.04	**.70±.04**
	enron-f	.57±.03	.59±.03	.59±.03	.52±.03	.59±.02	**.61±.01**
	entertainment1	.46±.11	.43±.10	.49±.14	.36±.09	**.67±.02**	.03±.10
	flags	.70±.03	.70±.04	.69±.04	.69±.03	.69±.02	**.71±.03**
	genbase	**.99±.01**	**.99±.01**	**.99±.01**	**.99±.01**	**.99±.01**	**.99±.01**
	health1	.56±.10	.58±.07	.61±.09	.42±.01	**.73±.02**	.36±.18
	llog-f	.15±.03	.17±.04	.15±.02	.14±.03	**.21±.04**	.19±.07
	mediamill	.50±.02	.56±.03	.56±.05	.50±.00	**.60±.03**	.00±.00
	medical	.81±.05	.79±.05	.77±.09	.80±.04	.81±.05	**.82±.05**
	recreation1	.40±.15	.34±.11	.42±.14	.25±.09	**.61±.05**	.13±.11
	reference1	.48±.07	.53±.04	.52±.04	.46±.06	**.64±.02**	.00±.00
	science1	.35±.13	.36±.08	.36±.12	.38±.20	**.57±.02**	.03±.08
	social1	.40±.02	.51±.09	.42±.14	.39±.02	**.68±.02**	.00±.00
	society1	.40±.01	.46±.04	.47±.04	.40±.01	**.52±.05**	.00±.00
	tmc2007	.37±.03	.36±.02	**.38±.02**	.00±.00	.36±.02	.00±.00
	yeast	.66±.01	.65±.02	.66±.02	.63±.02	**.67±.02**	**.67±.01**
	avg. rank	3.71	2.92	3.33	4.92	**1.67**	4.46
Label-Wise F-Measure (F_L)	arts1	.18±.09	.24±.05	.22±.05	.14±.01	**.30±.01**	.05±.08
	bibtex	.18±.08	**.32±.03**	.29±.06	.14±.01	.27±.03	.05±.02
	birds	.40±.05	.40±.06	.42±.06	.41±.06	**.43±.07**	.42±.06
	bookmarks	.01±.02	**.04±.06**	.00±.01	.00±.00	**.04±.04**	.00±.00
	business1	.17±.06	.22±.07	.22±.08	.13±.01	**.27±.06**	.06±.06
	computers1	.16±.08	.17±.05	.22±.07	.04±.07	**.32±.02**	.00±.01
	education1	.14±.08	.15±.05	.16±.08	.10±.04	**.22±.01**	.02±.02
	emotions	.68±.03	.67±.04	.68±.03	.68±.04	**.70±.04**	.69±.04
	enron-f	.18±.03	.20±.03	.20±.02	.20±.02	**.21±.03**	**.21±.03**
	entertainment1	.22±.15	.32±.08	.29±.13	.27±.09	**.40±.03**	.01±.02
	flags	.53±.07	.51±.05	.51±.06	**.55±.07**	.53±.09	.53±.08
	genbase	.63±.06	**.64±.06**	**.64±.06**	.63±.06	**.64±.07**	**.64±.06**
	health1	.24±.11	.26±.07	.23±.11	.17±.02	**.31±.02**	.03±.02
	llog-f	.05±.01	.06±.02	.06±.02	**.07±.01**	**.07±.01**	.06±.02
	mediamill	.10±.05	.17±.05	.17±.06	.23±.01	**.40±.05**	.00±.00
	medical	.32±.03	.33±.04	.33±.03	.33±.03	**.34±.04**	**.34±.03**
	recreation1	.22±.16	.35±.08	.29±.13	.24±.06	**.42±.14**	.02±.02
	reference1	.17±.07	.17±.05	.15±.08	.11±.03	**.23±.06**	.00±.00
	scene	.76±.02	.76±.02	.76±.02	.73±.03	**.79±.02**	**.79±.02**
	science1	.15±.11	.24±.05	.20±.07	.17±.07	**.27±.03**	.00±.01
	social1	.08±.06	.17±.08	.16±.09	.02±.01	**.21±.02**	.00±.00
	society1	.06±.08	.18±.06	.19±.08	.02±.00	**.30±.02**	.00±.00
	tmc2007	.05±.08	.08±.10	.10±.11	.00±.00	**.13±.11**	.00±.00
	yeast	.47±.01	.47±.01	.47±.01	.46±.01	**.48±.01**	.47±.01
	avg. rank	4.46	3.25	2.92	4.42	**1.33**	4.63
Micro-Averaged F-Measure (F_μ)	arts1	.32±.12	.39±.05	.37±.08	.27±.01	**.50±.01**	.24±.08
	bibtex	.39±.07	**.43±.02**	.42±.04	.27±.02	.42±.02	.20±.09
	birds	.55±.04	.54±.06	.54±.04	.55±.05	**.60±.05**	**.60±.06**
	bookmarks	.06±.02	.09±.06	.11±.06	.00±.00	**.14±.05**	.03±.04
	business1	.68±.03	.72±.03	.72±.04	.60±.20	**.76±.02**	.60±.20
	computers1	.47±.08	.51±.06	.48±.06	.42±.01	**.59±.04**	.04±.13
	education1	.36±.11	.37±.10	.36±.11	.30±.06	**.52±.01**	.14±.14
	emotions	**.71±.04**	.69±.04	.70±.05	.70±.03	**.71±.03**	.70±.04
	enron-f	.59±.02	.60±.02	.60±.02	.57±.02	**.61±.02**	**.61±.01**
	entertainment1	.42±.11	.45±.10	.43±.09	.41±.09	**.66±.01**	.03±.09
	flags	.70±.03	.72±.03	.71±.03	.71±.03	**.72±.03**	**.72±.03**
	genbase	**.99±.01**	**.99±.01**	**.99±.01**	**.99±.01**	**.99±.01**	**.99±.01**
	health1	.51±.12	.61±.04	.57±.07	.44±.02	**.71±.03**	.34±.17
	llog-f	.19±.05	.22±.04	.22±.03	**.26±.04**	.25±.05	.20±.08
	mediamill	.50±.02	.57±.06	.58±.04	.53±.00	**.64±.02**	.00±.00
	medical	.82±.04	.81±.04	.81±.04	.81±.04	.82±.04	**.83±.04**
	recreation1	.34±.16	.38±.11	.36±.14	.30±.02	**.59±.04**	.12±.10
	reference1	.48±.06	.51±.05	.53±.06	.43±.02	**.63±.02**	.00±.00
	scene	.75±.02	.75±.02	.75±.02	.73±.03	.78±.02	**.79±.02**
	science1	.27±.10	.35±.12	.32±.13	.31±.13	**.52±.03**	.02±.07
	social1	.39±.02	.45±.10	.49±.10	.33±.12	**.62±.08**	.00±.00
	society1	.37±.01	.45±.04	.39±.14	.37±.01	**.51±.02**	.00±.00
	tmc2007	.32±.07	.36±.06	.36±.06	.00±.00	**.38±.02**	.00±.00
	yeast	.66±.02	.67±.02	.67±.01	.65±.01	**.68±.02**	**.68±.01**
	avg. rank	4.00	3.04	3.13	4.96	**1.33**	4.54

search space, we used the AILibs[2] format of the project HASCO and the extensive description of MEKA and WEKA provided in [65]. The source code for the benchmarking framework and the experiments is publicly available via GitHub[3].

7.2 Analysis of Generalization Performance

The test performances for all the methods and datasets across 10 train and test splits and the three performance measures (instance-wise, label-wise, and micro-averaged F-Measure) are given in Table 3. At first glance, one can observe that HTN-BF performs best in most of the cases and tends to outperform all other methods on a wide range of datasets. To obtain a better and more profound overall impression, we have additionally visualized the results in the form of scatter plots in Fig. 8, where we compare the performance of one method against all others for each of the performance measures. A single point in this plot depicts the relative performance of the one method and another compared method for one of the datasets, where the performance of the one method is on the x-axis and that of the compared method on the y-axis. The generalization performance of the considered method improves from left to right, and the performance of the compared methods bottom up. A tie in the generalization performance is observed whenever a point is located on the diagonal. If a point lies below (above) the diagonal, it means the considered method performs better (worse).

These plots clearly show that HTN-BF mostly dominates the other methods and yields (most of the time just slightly) inferior results on a few datasets only. In fact, the few cases in which another algorithm exhibits better performance are not even statistically significant. While the advantage of HTN-BF is clearly visible for all performance measures, it is especially obvious for the case of label-wise F-Measure optimization. The measure seems to be rather hard to optimize by the AutoML approaches since the scores are in general rather low. Yet, HTN-BF manages to obtain scores that improve up to a factor of three compared to SMAC and Hyperband (let alone Random Search, which is completely off the mark). Furthermore, we can observe that SMAC is more in the midfield, whereas HB and BOHB perform usually superior to the other methods (except for HTN-BF). Apart from the random search, GGP typically performs inferior to the other considered methods, such that most of the points are located above the diagonal.

In Table 3, we can see that the advantage of HTN-BF is often statistically significant. For each dataset, we report the mean result of each algorithm together with its standard deviation. The algorithm with the best mean score is marked in bold, and we underline those results that are not significantly worse in a statistical sense (according to a Wilcoxon signed-rank test with a threshold for the p-value of 0.05) for the same dataset. As suggested by the rather low standard deviations and confirmed by the significance test, the results are not just by chance. Instead, the advantage of HTN-BF appears to be systematic. In spite of HTN-BF improving over other approaches by factors on some datasets, the statistical difference in summary is less pronounced for the label-wise F-measure. For the other two performance measures, the great majority of advantageous entries is also significant.

2. https://github.com/starlibs/AILibs

3. https://github.com/mwever/tpami-automlc

The random search baseline manages to return better solutions than the other optimizers on several datasets even after 24 hours of runtime. Furthermore, for two of the three measures, it is even able to obtain a better average rank than GGP, getting close to SMAC and GGP for the label-wise F-measure. Random search does not offer a practically useful alternative, however, as it also produces disastrous results on a considerable number of datasets. The strongly fluctuating performance can be explained by the fact that the random search first draws one element from the set of all possible unparameterized classifiers, which has, by definition, a bias towards more complex classifier structures (i.e. a higher tendency for including meta classifiers for multi-label classification as well as single-label base learners) since those represent a larger fraction of the set.

In the nested donut charts of Fig. 11, we present the relative frequency of an algorithm being selected by the respective optimizer across all runs. The layers of the nested donut charts represent the five different component types reading from outside to inside: meta multi-label, base multi-label, meta single-label, base single-label, and kernel algorithms. For a better readability, only algorithms with a portion of at least 0.05 are shown. Algorithms below this threshold are grouped together under the label "Others". If no algorithm has been selected for a particular layer, this is denoted by a "/". Note that meta methods do not necessarily need to be selected as opposed to base multi-label algorithms that are required to occur in any solution. This figure makes very clear that SMAC, HB, and BOHB select somewhat similar solutions which also explains their similar performance in various settings. However, SMAC's and HB's choices differ more from each other than each of them differs from BOHB. Another interesting observation is that the bias of the random search towards more complex classifier structures is obvious and clearly distinguishes it from any other method. On one hand, this bias enables random search to yield best performances on some of the datasets. On the other hand, classifier evaluations are more prone to timeouts, because more complex classifiers usually also need considerably more evaluation time, explaining the disastrous results previously mentioned. Lastly, GGP and HTN-BF favor simpler solutions barely incorporating meta algorithms at all. Still the methods selected by GGP and HTN-BF differ significantly, especially the set of chosen multi-label base algorithms is way more diverse in the case of HTN than for GGP.

Methods that are based on a reduction to hyper-parameter optimization are usually inferior to HTN-BF but still better on a few datasets. Overall, however, it is obvious that HB and BOHB compare favorably to SMAC, which we attribute to the feature of multifidelity optimization. Since HB and BOHB are allowed to evaluate single iterations of the Monte Carlo cross-validation (MCCV), they can use more time to explore a more diverse array of classifiers and then focus more and more on the promising candidates. In the anytime average rank plots in Fig. 10, we can observe that these methods usually perform superior in the beginning, but HTN-BF passes by after one hour (first vertical dashed line). While HB and BOHB race head-to-head, SMAC is more or less off the mark, especially for F_L and F_μ. Nevertheless, in the case of F_I, SMAC manages to perform competitively to BOHB. GGP and Random quickly drop to the back ranks, which is due to sampling the (first) incumbent uniformly at random leading to rather complex models that take longer to evaluate or might even timeout and any method is considered to have a score of 0 as long as no incumbent was found.

Grammar-based genetic programming (GGP) performs the worst on average. After 12h (third dashed vertical line) it significantly loses in terms of average ranks compared to all other methods, at this point performing even worse than random search for F_I and F_μ. However, the bad performance can be attributed to the parameterization of the evaluated GGP approach, which has been configured with a population containing only 15 individuals, as it was advised in [5]. While this seems to be a reasonable value for moderate runtimes of up to 6 hours (second dashed vertical line), it impedes a sufficient exploration of the search space as carried out by other methods. Additionally, we only use a straight-forward version of GGP which does not leverage more sophisticated features, as for example restarting.

Another interesting insight is with respect to the "stability" of the approaches. We can see that the standard deviation is smaller for HTN-BF than for all other algorithms, both on average and in the extremes. In other words, HTN-BF produces high-quality results on a quite constant level. As a consequence, HTN-BF can be expected to produce better results than SMAC or HB in almost all cases, not only on average. Furthermore, results obtained from other methods can also perform considerably worse than the mean performance, entailing a certain risk when being used in practice.

Finally, even if HTN-BF is playing quite a dominant role, it is worth mentioning that each of the other methods yields the best performance for at least one combination of dataset and performance measure.

7.3 Discussion of Results

The first conclusion one may want to draw from the results is that greedily pursuing candidate lines pays off in the multi-label scenario. Among all compared algorithms, HTN-BF along with GGP is clearly the most greedy algorithm; its only exploration is in the number of samples drawn for each node evaluation. But this number is small (here 3), and there is no further update of those values once the node evaluation has been completed. However, although GGP can also be considered quite greedy due to its local search behavior, it very much depends on its initialization and gets stuck in local optima quite easily. Given HTN-BF's great overall performance, we conclude that greediness is preferable over exploration for this setting, which is characterized by an extremely large search space.

This also seems to have an intuitive explanation in the long evaluation times in multi-label classification, which are shown in terms of boxplots in Fig. 9. There is simply not enough time for exhaustive evaluation, and being stuck in a local optimum, at least provided enough exploration in the beginning, is a substantially smaller risk here than not optimizing at all.

However, taking a closer look, it is not entirely clear whether the advantage of HTN-BF is due to the search behavior or due to the formal model for specifying the search space. In other words, maybe the advantage already comes from using a grammar-based approach for modeling the search space instead of flattening the space to a hyper-parameter optimization vector, whereas the (greedy) algorithm used to traverse that space has a less strong influence.

This suspicion seems to be confirmed by the fact that the random search, being the least greedy algorithm, does also sometimes perform well. In fact, among all cases in which HTN-BF is not best, the random search has the highest chance to be the winner. For these particular datasets, this is either attributed to the fact that the more sophisticated methods tend to focus on flatter classifiers and thus simpler classifier structures, or it does not seem to advocate any strategy that exploits the information encountered so far.

On the other hand, the results of the random search are often also quite disastrous, as it repeatedly runs into timeouts and cannot find any reasonable solution. For example, the score on mediamill, social1, society1, and tmc2007 is 0, compared to values between .3 and .6 for the other algorithms[4]. Hence, random search is certainly not a reasonable alternative. Note that in cases where the score is 0, classifiers are returned that are fast to evaluate but low in performance. Often these solutions employ a majority classifier as a base learner for the transformation methods, which due to the rare label activation always scores 0.

Overall, all methods seem to struggle with the tremendous size of the search space. While greediness still seems to be the best way to cope with this challenge, just like all other methods, it tends to ignore classifiers that are structurally more complex. As indicated by the results of the random search, which is more biased towards such methods, simply leaving out the more complex methods would come at the price of excluding the optimal solution for some tasks. Nevertheless, to improve the performance of the obtained solutions, either the methods need to be adapted further to work more effectively in the MLC search space, or the problem needs to be transformed so that the methods can better cope with it. For the latter, one option would be to implement meta-learning approaches to dynamically prune parts of the search space, i.e., in an instance-wise manner, which are anticipated not to be relevant for the final solution. For example this could be done employing approaches to extreme algorithm selection (XAS), which proved beneficial in settings with a large number of different algorithms [66].In this way the optimizers could focus on the more promising candidates as anticipated by the XAS model. Another option would be to incorporate safeguards for the evaluation of solution candidates to avoid timeouts, thus allowing one to waste time for regions that are omitted from the effective solution space anyways. Interestingly, the observation that either method needs to be adapted to better fit the MLC setting, or that the search space needs to be transformed in a way to better suit the methods we already have developed for SLC, perfectly matches the philosophy according to which classifiers for MLC have been developed in the literature so far.

4. One may wonder how any algorithm can have positive results if such results cannot be obtained even with maximum exploration. The explanation here is that the systematic searches have a *more systematic* exploration. For example, if the evaluation of a node in HTN-BF obtains a timeout, the corresponding sub-tree of this node is ignored, whereas random search may consider repeatedly instances of this algorithm which are also very likely to produce a timeout.

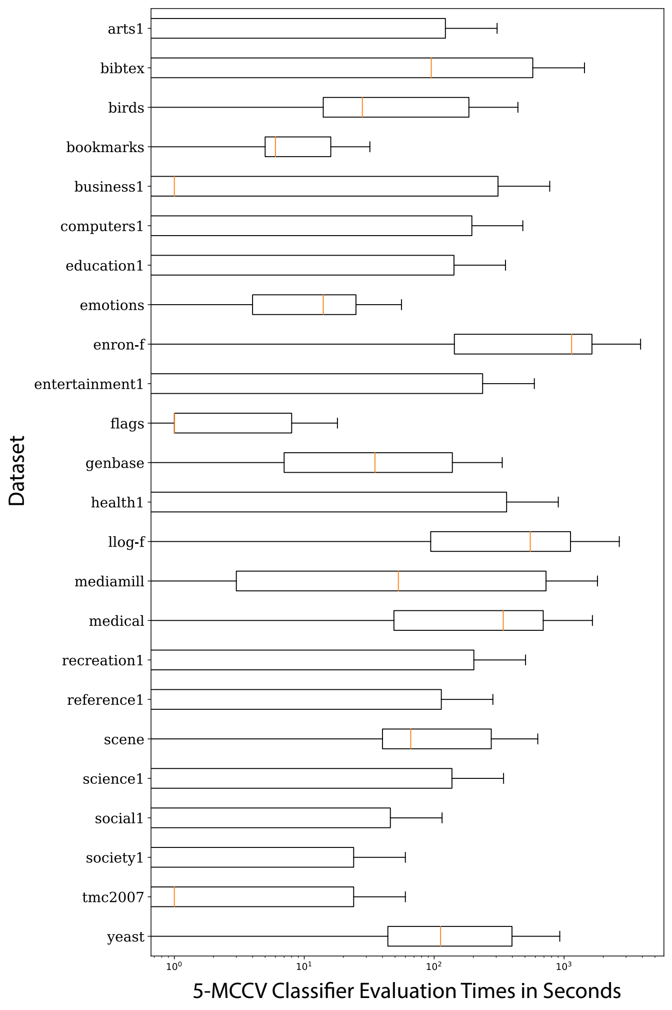

Fig. 9: Evaluation times of successful classifier evaluations.

8 Conclusion

In this work, we considered existing optimization approaches for automating multi-label classification and, moreover, transferred other AutoML approaches commonly used for single-label classification to the problem domain of MLC. Furthermore, we defined a benchmarking framework for multi-label classification, which allows for isolated optimizer comparisons ensuring that all of them run within the same computational and time constraints, and that they operate on the same search space, i.e., the same solution candidates can be found and the same performance estimation of solution candidates is used.

Our extensive study revealed that a reduction of the AutoML problem to hyper-parameter optimization does not scale well to the problem domain of MLC out of the box. Consequently, to apply those techniques properly, more work on dealing with the extremely large search space and the deep hierarchical configuration structures of multi-label classifiers is necessary.

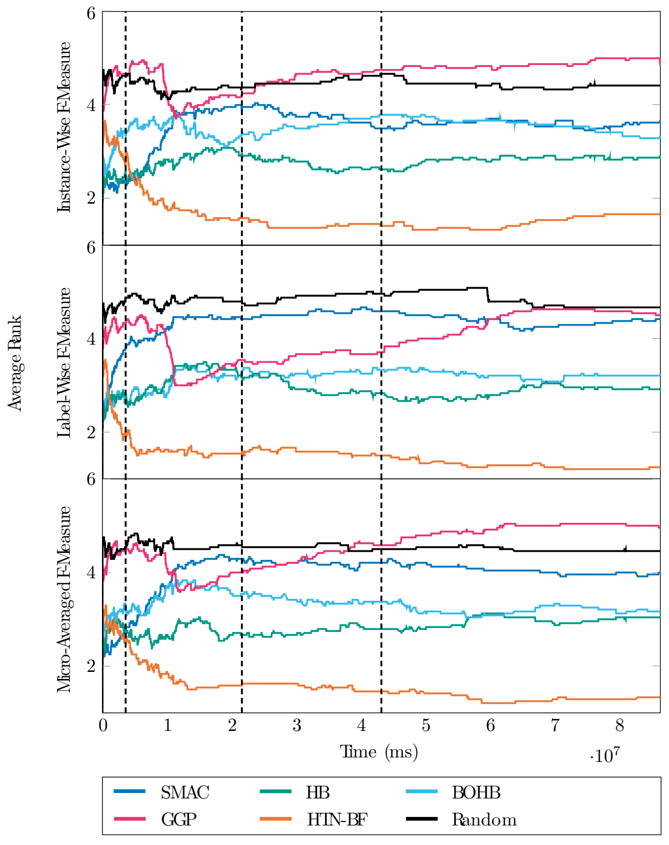

Fig. 10: Average ranks over time (in ms) for the three performance measures: instance-wise F-Measure (F_I), label-wise F-Measure (F_L), and micro-averaged F-Measure (F_μ).

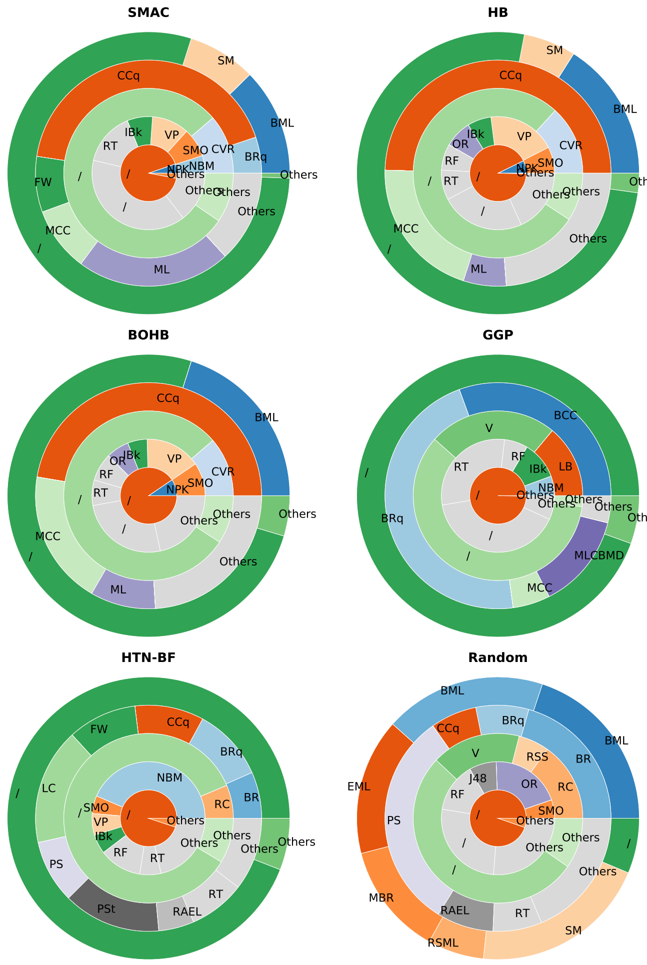

Fig. 11: Frequencies of chosen algorithms per optimizer and algorithm.

On the contrary, a greedy global search approach based on hierarchical task network planning yields promising results, showing the potential to properly deal with the hierarchical structures that are also reflected in the model of the search space. However, all of the considered AutoML approaches have in common that they focus on classifiers having a flatter structure than others. As a result, more complex classifiers with a better generalization performance are not yet sufficiently considered. To address this problem, we outlined two interesting research directions, which are in line with the two ways classifiers for MLC have been developed in the past: to either adapt the methods to the specifics of the MLC search space, or to transform the original AutoML problem for MLC into a problem that is more amenable to the already existing approaches.

Acknowledgments

This work was funded by the German Research Foundation (DFG) within the Collaborative Research Center “On-The-Fly Computing” (SFB 901/3 project no. 160364472). The authors gratefully acknowledge support of this project by the Paderborn Center for Parallel Computing (PC^2), which provided computational resources and computing time.

References

[1] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, “Efficient and robust automated machine learning,” in *Advances in neural information processing systems*, 2015.

[2] R. S. Olson, N. Bartley, R. J. Urbanowicz, and J. H. Moore, “Evaluation of a tree-based pipeline optimization tool for automating data science,” in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, 2016.

[3] W. Waegeman, K. Dembczyński, and E. Hüllermeier, “Multi-target prediction: a unifying view on problems and methods,” *Data Mining and Knowledge Discovery*, vol. 33, no. 2, 2019.

[4] A. G. de Sá, G. L. Pappa, and A. A. Freitas, “Towards a method for automatically selecting and configuring multi-label classification algorithms,” in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2017.

[5] A. G. de Sá, A. A. Freitas, and G. L. Pappa, “Automated selection and configuration of multi-label classification algorithms with grammar-based genetic programming,” in *International Conference on Parallel Problem Solving from Nature*. Springer, 2018.

[6] M. Wever, F. Mohr, and E. Hüllermeier, “Automated multi-label classification based on ml-plan,” *arXiv preprint arXiv:1811.04060*, 2018.

[7] M. Wever, F. Mohr, A. Tornede, and E. Hüllermeier, “Automating multi-label classification extending ml-plan,” in *Proceedings of the AutoML Workshop at ICML*, vol. 2020, 2019.

[8] A. Pakrashi and B. Mac Namee, “Cascademl: An automatic neural network architecture evolution and training algorithm for multi-label classification (best technical paper),” in *International Conference on Innovative Techniques and Applications of Artificial Intelligence*. Springer, 2019.

[9] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, “Auto-weka: Combined selection and hyperparameter optimization of classification algorithms,” in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2013.

[10] F. Hutter, H. H. Hoos, and K. Leyton-Brown, “Sequential model-

based optimization for general algorithm configuration," in *International conference on learning and intelligent optimization*. Springer, 2011.

[11] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *The Journal of Machine Learning Research*, vol. 18, no. 1, 2017.

[12] S. Falkner, A. Klein, and F. Hutter, "BOHB: Robust and efficient hyperparameter optimization at scale," pp. 1437–1446, 2018.

[13] B. Komer, J. Bergstra, and C. Eliasmith, "Hyperopt-sklearn," in *Automated Machine Learning*. Springer, 2019.

[14] S. C. N. das Dôres, C. Soares, and D. Ruiz, "Bandit-based automated machine learning," in *2018 7th Brazilian Conference on Intelligent Systems (BRACIS)*. IEEE, 2018.

[15] P. Gijsbers and J. Vanschoren, "Gama: genetic automated machine learning assistant," *Journal of Open Source Software*, vol. 4, no. 33, 2019.

[16] A. G. de Sá, W. J. G. Pinto, L. O. V. Oliveira, and G. L. Pappa, "Recipe: a grammar-based framework for automatically evolving classification pipelines," in *European Conference on Genetic Programming*. Springer, 2017, pp. 246–261.

[17] F. Mohr, M. Wever, and E. Hüllermeier, "Ml-plan: Automated machine learning via hierarchical planning," *Machine Learning*, vol. 107, no. 8-10, 2018.

[18] M. Wever, F. Mohr, and E. Hüllermeier, "Ml-plan for unlimited-length machine learning pipelines," in *ICML 2018 AutoML Workshop*, 2018.

[19] H. Rakotoarison, M. Schoenauer, and M. Sebag, "Automated machine learning with monte-carlo tree search," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, 2019.

[20] I. Drori, Y. Krishnamurthy, R. Rampin, R. Lourenço, J. One, K. Cho, C. Silva, and J. Freire, "Alphad3m: Machine learning pipeline synthesis," in *AutoML Workshop at ICML*, 2018.

[21] X. He, K. Zhao, and X. Chu, "Automl: A survey of the state-of-the-art," *arXiv preprint arXiv:1908.00709*, 2019.

[22] N. Erickson, J. Mueller, A. Shirkov, H. Zhang, P. Larroy, M. Li, and A. Smola, "Autogluon-tabular: Robust and accurate automl for structured data," *arXiv preprint arXiv:2003.06505*, 2020.

[23] B. Chen, H. Wu, W. Mo, I. Chattopadhyay, and H. Lipson, "Autostacker: A compositional evolutionary learning system," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2018, pp. 402–409.

[24] X. Sun, J. Lin, and B. Bischl, "Reinbo: Machine learning pipeline conditional hierarchy search and configuration with bayesian optimization embedded reinforcement learning," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2019, pp. 68–84.

[25] E. LeDell and S. Poirier, "H2o automl: Scalable automatic machine learning," in *Proceedings of the AutoML Workshop at ICML*, vol. 2020, 2020.

[26] A. Balaji and A. Allen, "Benchmarking automatic machine learning frameworks," *arXiv preprint arXiv:1808.06492*, 2018.

[27] P. Gijsbers, E. LeDell, S. Poirier, J. Thomas, B. Bischl, and J. Vanschoren, "An open source automl benchmark," in *6th ICML Workshop on Automated Machine Learning*, 2019.

[28] G. Tsoumakas, I. Katakis, and I. Vlahavas, "Mining multi-label data," in *Data mining and knowledge discovery handbook*. Springer, 2009.

[29] M.-L. Zhang and Z.-H. Zhou, "A review on multi-label learning algorithms," *IEEE transactions on knowledge and data engineering*, vol. 26, no. 8, 2013.

[30] G. Tsoumakas and I. Katakis, "Multi-label classification: An overview," *International Journal of Data Warehousing and Mining (IJDWM)*, vol. 3, no. 3, 2007.

[31] D. Kocev, C. Vens, J. Struyf, and S. Džeroski, "Ensembles of multi-objective decision trees," in *European conference on machine learning*. Springer, 2007.

[32] M.-L. Zhang, Y.-K. Li, X.-Y. Liu, and X. Geng, "Binary relevance for multi-label learning: an overview," *Frontiers of Computer Science*, vol. 12, no. 2, 2018.

[33] J. Read, B. Pfahringer, G. Holmes, and E. Frank, "Classifier chains for multi-label classification," *Machine learning*, vol. 85, no. 3, 2011.

[34] R. Senge, J. J. Del Coz, and E. Hüllermeier, "On the problem of error propagation in classifier chains for multi-label classification," in *Data Analysis, Machine Learning and Knowledge Discovery*. Springer, 2014.

[35] A. Rivolli, J. Read, C. Soares, B. Pfahringer, and A. C. de Carvalho, "An empirical analysis of binary transformation strategies and base algorithms for multi-label learning," *Machine Learning*, 2020.

[36] M. Wever, A. Tornede, F. Mohr, and E. Hüllermeier, "Libre: Label-wise selection of base learners in binary relevance for multi-label classification," in *International Symposium on Intelligent Data Analysis*. Springer, 2020.

[37] J. Nam, Y.-B. Kim, E. L. Mencia, S. Park, R. Sarikaya, and J. Fürnkranz, "Learning context-dependent label permutations for multi-label classification," in *International Conference on Machine Learning*, 2019.

[38] F. Hutter, L. Kotthoff, and J. Vanschoren, *Automated machine learning: methods, systems, challenges*. Springer Nature, 2019.

[39] M.-A. Zöller and M. F. Huber, "Benchmark and survey of automated machine learning frameworks," *arXiv preprint arXiv:1904.12054*, 2019.

[40] Q. Yao, M. Wang, Y. Chen, W. Dai, H. Yi-Qi, L. Yu-Feng, T. Wei-Wei, Y. Qiang, and Y. Yang, "Taking human out of learning applications: A survey on automated machine learning," *arXiv preprint arXiv:1810.13306*, 2018.

[41] R. Elshawi, M. Maher, and S. Sakr, "Automated machine learning: State-of-the-art and open challenges," *arXiv preprint arXiv:1906.02287*, 2019.

[42] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in neural information processing systems*, 2012.

[43] 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," *The Journal of Machine Learning Research*, vol. 18, no. 1, 2017.

[44] K. Jamieson and A. Talwalkar, "Non-stochastic best arm identification and hyperparameter optimization," in *Artificial Intelligence and Statistics*, 2016.

[45] M. Hoffman, B. Shahriari, and N. Freitas, "On correlation and budget constraints in model-based bandit optimization with application to automatic machine learning," in *Artificial Intelligence and Statistics*, 2014, pp. 365–374.

[46] M. Feurer, K. Eggensperger, S. Falkner, M. Lindauer, and F. Hutter, "Practical automated machine learning for the automl challenge 2018," in *International Workshop on Automatic Machine Learning at ICML*, 2018.

[47] M. Mitchell, *An introduction to genetic algorithms*. MIT press, 1998.

[48] R. I. Mckay, N. X. Hoai, P. A. Whigham, Y. Shan, and M. O'neill, "Grammar-based genetic programming: a survey," *Genetic Programming and Evolvable Machines*, vol. 11, no. 3-4, pp. 365–396, 2010.

[49] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: theory and practice*. Elsevier, 2004.

[50] F. Mohr, M. Wever, and E. Hüllermeier, "Automated machine learning service composition," *arXiv preprint arXiv:1809.00486*, 2018.

[51] F. Mohr, M. Wever, E. Hüllermeier, and A. Faez, "(wip) towards the automated composition of machine learning services," in *2018 IEEE International Conference on Services Computing (SCC)*. IEEE, 2018, pp. 241–244.

[52] P. I. Frazier, "A tutorial on bayesian optimization," *arXiv preprint arXiv:1807.02811*, 2018.

[53] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Advances in neural information processing systems*, 2011.

[54] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, 2001.

[55] P. Hennig and C. J. Schuler, "Entropy search for information-efficient global optimization," *The Journal of Machine Learning Research*, vol. 13, no. 1, 2012.

[56] P. I. Frazier, W. B. Powell, and S. Dayanik, "A knowledge-gradient policy for sequential information collection," *SIAM Journal on Control and Optimization*, vol. 47, no. 5, 2008.

[57] J. Močkus, "On bayesian methods for seeking the extremum," in *Optimization techniques IFIP technical conference*. Springer, 1975.

[58] 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, 1998.

[59] Z. Karnin, T. Koren, and O. Somekh, "Almost optimal exploration in multi-armed bandits," in *International Conference on Machine Learning*, 2013.

[60] M. Wever, F. Mohr, and E. Hüllermeier, “Automatic machine learning: Hierarchical planning versus evolutionary optimization,” in *Proceedings of the 27. Workshop Computational Intelligence*, 2017.
[61] F. Mohr, T. Lettmann, E. Hüllermeier, and M. Wever, “Programmatic task network planning,” in *Proceedings of the 1st ICAPS Workshop on Hierarchical Planning*, 2018, pp. 31–39.
[62] J.-U. Kietz, F. Serban, A. Bernstein, S. Fischer, J. Vanschoren, and P. Brazdil, “Designing kdd-workflows via htn-planning for intelligent discovery assistance,” 2012.
[63] J. Read, P. Reutemann, B. Pfahringer, and G. Holmes, “MEKA: A multi-label/multi-target extension to Weka,” *Journal of Machine Learning Research*, vol. 17, no. 21, 2016. [Online]. Available: http://jmlr.org/papers/v17/12-164.html
[64] F. Eibe, M. Hall, and I. Witten, “The weka workbench. online appendix for “data mining: Practical machine learning tools and techniques”, morgan kaufmann,” 2016.
[65] A. G. de Sá, A. A. Freitas, and G. L. Pappa, “Multi-label classification search space in the meka software,” *arXiv preprint arXiv:1811.11353*, 2018.
[66] A. Tornede, M. Wever, and E. Hüllermeier, “Extreme algorithm selection with dyadic feature representation,” in *Proceedings of the Discovery Science Conference*, 2020.

Marcel Wever received the B.Sc. and M.Sc. degrees from Paderborn University, Germany in 2015 respectively 2017. He is currently working as a research assistant in the intelligent systems and machine learning group at Paderborn University, studying towards a PhD degree with a main focus on automated machine learning and multi-label classification.

Alexander Tornede received the B.Sc. and M.Sc. degrees in Computer Science from Paderborn University, Germany in 2015 respectively 2018. He is currently working as a research assistant in the intelligent systems and machine learning group at Paderborn University, studying towards a PhD degree.

Felix Mohr is a professor in the Faculty of Engineering at Universidad de la Sabana in Colombia. His research focus lies in the areas of Stochastic Tree Search as well as Automated Software Configuration with a particular specialization on Automated Machine Learning. He received his PhD in 2016 from Paderborn University in Germany.

Eyke Hüllermeier is a professor in the Department of Computer Science at Paderborn University, where he heads the Intelligent Systems and Machine Learning Group, a member of the Heinz Nixdorf Institute, and a Director of the Software Innovation Campus Paderborn. He received his PhD in 1997 and a Habilitation degree in 2002. Prior to joining Paderborn University in 2014, he held professorships at the Universities of Dortmund, Magdeburg and Marburg.