Automated Machine Learning for Multi-Label Classification

Marcel Wever

March 01, 2022

PADERBORN UNIVERSITY
The University for the Information Society
Department of Electrical Engineering,
Computer Science and Mathematics
Warburger Straße 100
33098 Paderborn

INTELLIGENT SYSTEMS
Intelligent Systems Group (ISG)

Dissertation

In partial fulfillment of the requirements for the academic degree of
Doctor rerum naturalium (Dr. rer. nat.)

Automated Machine Learning for Multi-Label Classification

Marcel Wever

1. Reviewer	Prof. Dr. Eyke Hüllermeier Künstliche Intelligenz und Maschinelles Lernen Ludwig-Maximilians-Universität München
2. Reviewer	Prof. Dr. Axel-Cyrille Ngonga Ngomo Data Science Paderborn University
3. Reviewer	Prof. Dr. Bernd Bischl Statistical Learning & Data Science Ludwig-Maximilians-Universität München
Supervisor	Prof. Dr. Eyke Hüllermeier

March 01, 2022

Marcel Wever
Automated Machine Learning for
Multi-Label Classification
Dissertation, March 01, 2022
Reviewers: Prof. Dr. Eyke Hüllermeier,
Prof. Dr. Axel-Cyrille Ngonga Ngomo,
Prof. Dr. Bernd Bischl
Supervisor: Prof. Dr. Eyke Hüllermeier

Intelligent Systems and Machine Learning
Department of Electrical Engineering,
Computer Science and Mathematics
Pohlweg 51
33098 Paderborn

Danksagung

Es gibt viele Menschen, die auf ihre ganz eigene Art und Weise zu dieser Arbeit beigetragen haben, und ich möchte an dieser Stelle eben diesen Menschen meinen herzlichen Dank aussprechen.

Zuallererst möchte ich meinem Doktorvater Eyke danken, für die Möglichkeit der Promotion, sowie das Vertrauen und die Förderung, für die Inspiration und die Mitarbeit an meinen Projekten und Papieren - auch außerhalb der gängigen Öffnungszeiten. Weiterer Dank gilt auch meiner Promotionskommission, bestehend aus Axel, Bernd, Henning und Matthias, für ihre Bereitschaft, sich mit meiner Arbeit kritisch auseinanderzusetzen und an meiner Disputation teilzunehmen. Eyke, Axel und Bernd danke ich außerdem für die Anfertigung ihrer Gutachten.

Der ehemaligen Fachgruppe Intelligente Systeme und Maschinelles Lernen an der Universität Paderborn sowie der neuen Fachgruppe Künstliche Intelligenz und Maschinelles Lernen an der LMU möchte ich dafür Danke sagen, mich in ihre Reihen aufgenommen, begleitet und unterstützt zu haben. Ohne dieses Umfeld von Kolleg:innen wäre vieles von dem, was ich in dieser Zeit erreicht habe, nicht möglich gewesen. Besonders danken möchte ich an dieser Stelle Elisabeth für ihre unermüdliche Unterstützung bei allen nicht-wissenschaftlichen Belangen, meinem Bürokollegen Alexander für viele wertvolle Diskussionen und nochmals Alexander sowie Viktor für das Korrekturlesen dieser Arbeit. Ein weiteres Dankeschön möchte ich den studentischen Hilfskräften aussprechen, die mich bei meinen Arbeiten über die Jahre unterstützt haben.

Außerdem möchte ich mich noch bei einem weiteren Kollegen und Freund bedanken. Felix hat mich von Tag Null an mit ins Boot geholt, mir gezeigt, unter welchem Kurs sich die Segel am besten mit Wind füllen und wie man mit den unterschiedlichsten Wetterlagen umgeht. Danke dafür, dass ich auch dann und wann mal ins kalte Wasser geschmissen wurde, damit ich meine eigenen Erfahrungen machen konnte.

Während meiner Zeit als Doktorand hatte ich zudem das Privileg im DFG-Projekt "InterGramm" sowie im DFG Sonderforschungsbereich (SFB) 901 "On-The-Fly Computing" mitzuarbeiten und im Rahmen des SFBs diese Arbeit schreiben zu dürfen. An dieser Stelle möchte ich auch nochmal dem Geschäftsführer des SFBs, Ulf, für sein Engagement und Unterstützung bei allen erdenklichen Fragen danken.

Zu guter Letzt möchte ich meinen Eltern für ihre steten Bemühungen, mich von Beginn an zu bestärken und in allem zu fördern, von ganzem Herzen danken. Zusätzlich danke ich auch meiner Familie und meinen Freunden für ihre Unterstützung auf Schritt und Tritt, für all die wertvollen Erinnerungen und dafür, dass sie immer für mich da sind. Ganz besonders möchte ich meiner Freundin Katharina danken, die vor allem auch dann für mich da war, wenn es mal etwas stressiger und chaotischer wurde.

Zusammenfassung

Das Ziel des automatisierten maschinellen Lernens (AutoML) ist es, zugeschnitten auf einen gegebenen Datensatz, Algorithmen für das maschinelle Lernen (ML) zu wählen, zu konfigurieren und in Form von ML-Pipelines zu kombinieren. Für überwachte Lernaufgaben, insbesondere binäre und multinomiale Klassifikation, auch als Single-Label-Klassifikation (engl.: single-label classification; SLC) bezeichnet, haben solche AutoML-Ansätze vielversprechende Ergebnisse geliefert. Die Aufgabe der Multi-Label-Klassifikation (engl.: multi-label classification; MLC), bei der Datenpunkte mit einer Menge von Klassenlabels anstelle eines einzelnen Klassenlabels assoziiert werden, hat bisher deutlich weniger Aufmerksamkeit erhalten. Im Kontext der Multi-Label-Klassifikation ist die datenspezifische Auswahl und Konfiguration von Multi-Label-Klassifikatoren selbst für Experten auf diesem Gebiet eine Herausforderung, da es sich um ein hochdimensionales Optimierungsproblem mit hierarchischen Abhängigkeiten über mehrere Ebenen hinweg handelt. Während der Raum von ML-Pipelines für SLC bereits äußerst viele Kandidaten umfasst, übertrifft die Größe des MLC-Suchraums die des SLC-Suchraums um mehrere Größenordnungen.

Im ersten Teil dieser Arbeit wird ein neuartiger AutoML-Ansatz für Single-Label-Klassifikationsaufgaben entwickelt, der ML-Pipelines optimiert, die aus maximal zwei Algorithmen bestehen. Dieser Ansatz wird dann erweitert, um zunächst Pipelines von unbegrenzter Länge und schließlich die komplexen hierarchischen Strukturen von Multi-Label-Klassifikatoren zu konfigurieren. Außerdem untersuchen wir, wie gut Ansätze, die den Stand der Technik im Bereich AutoML für Single-Label-Klassifikationsaufgaben bilden, mit der erhöhten Komplexität des AutoML Problems für Multi-Label-Klassifikation skalieren.

Im zweiten Teil wird untersucht, wie Methoden für SLC und MLC flexibler konfiguriert werden können, um die zur Verfügung stehenden Daten besser zu generalisieren, und wie die Effizienz von ausführungsbasierten AutoML-Systemen mit Hilfe von Laufzeitvorhersagen für ML-Pipelines gesteigert werden kann.

Abstract

Automated machine learning (AutoML) aims to select and configure machine learning algorithms and combine them into machine learning pipelines tailored to a dataset at hand. For supervised learning tasks, most notably binary and multinomial classification, aka single-label classification (SLC), such AutoML approaches have shown promising results. However, the task of multi-label classification (MLC), where data points are associated with a set of class labels instead of a single class label, has received much less attention so far. In the context of multi-label classification, the data-specific selection and configuration of multi-label classifiers are challenging even for experts in the field, as it is a high-dimensional optimization problem with multi-level hierarchical dependencies. While for SLC, the space of machine learning pipelines is already huge, the size of the MLC search space outnumbers the one of SLC by several orders.

In the first part of this thesis, we devise a novel AutoML approach for single-label classification tasks optimizing pipelines of machine learning algorithms, consisting of two algorithms at most. This approach is then extended first to optimize pipelines of unlimited length and eventually configure the complex hierarchical structures of multi-label classification methods. Furthermore, we investigate how well AutoML approaches that form the state of the art for single-label classification tasks scale with the increased problem complexity of AutoML for multi-label classification.

In the second part, we explore how methods for SLC and MLC could be configured more flexibly to achieve better generalization performance and how to increase the efficiency of execution-based AutoML systems.

Contents

1 Introduction

Within the past decade, the demand for artificial intelligence and, in particular, machine learning (ML) functionality has grown exceedingly fast and is still growing steadily. Certainly, this can be attributed to relevant and media-effective breakthroughs that have achieved superhuman performance: In 2011, for example, IBM's Watson already beat two champions in Jeopardy [Fer12; Fer+13], machine facial recognition managed to achieve an accuracy of more than 97% in 2014 [Tai+14], and in 2016 AlphaGo was the first computer program to beat professional human Go players [Sil+16; Sil+17]. Beyond that, machine learning applications can be found in more and more parts of society and the economy [RW14; JM15; ID20].

However, the engineering of such applications is a non-trivial endeavor and requires expertise in machine learning that end users typically do not have. More specifically, there exists a plethora of different machine learning algorithms which work differently well, depending on the tasks and data. Additionally, most of these algorithms expose parameters, so-called hyper-parameters[1], which need to be tuned to the data at hand to achieve the best possible performance [Wev+20; Riv+20]. To bridge the gap between supply and demand, the field of automated machine learning (AutoML) emerged to help non-experts access machine learning technology on the one hand and, on the other hand, to support machine learning experts in their work relieving them of some of the tedious tasks.

AutoML is the vision of automating as much of the data science process as possible, starting from raw data and providing a complete pipeline of machine learning algorithms. Such a pipeline may comprise algorithms for pre-processing the (raw) data, constructing or selecting features, and ultimately learning a model. The problem of AutoML was first formally stated in [Tho+13] as the combined algorithm selection and hyper-parameter optimization (CASH) problem, optimizing for the generalization performance of the respective algorithm choice and its configuration. Provided a CASH problem specification in terms of (i) a model of the search space, describing the space of available machine learning pipelines, and (ii) an evaluation function to assess the quality of solution candidates, the problem is usually treated as a sampling-

[1] Hyper-parameters are parameters to control the *learning process*, e.g., the number of neurons in a neural network. In turn, the learning process induces the parameters of a model from data, e.g., the weights of a neural network. For example, the learning rate is a hyper-parameter of the learning process for neural networks.

based black-box optimization problem. More specifically, an optimization algorithm is employed to traverse the search space in a trial-and-error fashion by executing solution candidates, i.e., machine learning pipelines, on the particular data. First approaches to AutoML are based on random search [Ber+11], Bayesian optimization [Tho+13; Kot+17; Feu+15], or genetic programming [Ols+16; Sá+17].

While these AutoML systems show promising performances for regression and binary or multi-class classification tasks, in the following referred to as single-label classification (SLC), where each data point is associated with a numeric value or a class label respectively, the task of multi-label classification (MLC) received far less attention. In MLC, instead of only a single class label, each data point is associated with a (sub-)set of class labels. MLC tasks can be found in various domains such as text categorization [SS00; Nam+14; MF08], image processing [Cab+11; Xue+11], video annotation [Qi+07], music classification [SZ11], bioinformatics [BST06], and medicine [Hei+13].

Generalizing the single-label classification setting, MLC methods often perform a problem transformation, reducing the original MLC task to a (set of) single-label classification problem(s) [TKV09; ZZ14]. Therefore, such problem transformation methods can be seen as a kind of meta-learner that can be configured with one or multiple SLC methods as a base learner. Moreover, since there are also meta-learners for MLC that require MLC methods in turn as base learners, this results in a nested configuration structure. The nesting causes the search space over the configuration options for MLC methods to be several orders of magnitude larger than search spaces for SLC or regression, thus, representing a particular challenge for AutoML systems [Wev+21]. Additionally, compared to SLC, solution candidates in MLC are often more expensive to execute. This requires AutoML systems that work in a trial-and-error fashion to be even more efficient.

The motivation for considering the AutoML problem in this thesis is rooted in the broader context of ”On-The-Fly Computing“ [Hap+13] which is also the name of the collaborative research center (CRC) 901. It deals with the automatic provision of IT services that are composed and configured on the fly out of base services which in turn are available on worldwide markets. Focusing on services that provide machine learning functionality, the role of this work is to provide a domain-specific configuration algorithm that is able to configure and tailor machine learning services to the data provided by a user.

In this thesis, we devise a novel approach to AutoML based on hierarchical task network planning [GNT04], a technique from the field of AI planning, which can naturally model hierarchical dependencies between algorithms as well as hyper-parameters. Hence, it is flexible enough to represent complex machine learning

pipelines and learner structures, as it is necessary for MLC. Employing a best-first search for a greedy traversal of the resulting search space model, our approach compares well with state-of-the-art AutoML systems in the SLC setting and especially scales with the increased search space complexity in the MLC case.

In Section 1.1, we give an overview of the structure and the contributions of this thesis. Section 1.2 presents a running example.

1.1 Thesis Structure

The remainder of the thesis is structured as follows. Chapter 2 is dedicated to the methodological and general background for this thesis introducing fundamental concepts of automated machine learning and multi-label classification.

This Ph.D. thesis consists of the following contributions, which are presented in Chapters 3 to 9, respectively:

1. Felix Mohr, Marcel Wever, and Eyke Hüllermeier. „ML-Plan: Automated machine learning via hierarchical planning“. In: *Mach. Learn.* 107.8-10 (2018), pp. 1495–1515. DOI: 10.1007/s10994-018-5735-z
2. Marcel Dominik Wever, Felix Mohr, and Eyke Hüllermeier. „ML-Plan for unlimited-length machine learning pipelines“. In: *ICML 2018 AutoML Workshop*. 2018. URL: https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxhdXRvbWwyMDE4aWNtbHxneDo3M2Q3MjUzYjViNDRhZTAx
3. Marcel Dominik Wever, Felix Mohr, Alexander Tornede, and Eyke Hüllermeier. „Automating multi-label classification extending ml-plan“. In: *ICML 2019 AutoML Workshop*. 2019. URL: https://www.automl.org/wp-content/uploads/2019/06/automlws2019_Paper46.pdf
4. Marcel Wever, Alexander Tornede, Felix Mohr, and Eyke Hüllermeier. „AutoML for Multi-Label Classification: Overview and Empirical Evaluation“. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43.9 (2021), pp. 3037–3054. DOI: 10.1109/TPAMI.2021.3051276
5. Marcel Wever, Alexander Tornede, Felix Mohr, and Eyke Hüllermeier. „LiBRe: Label-Wise Selection of Base Learners in Binary Relevance for Multi-label Classification“. In: *Advances in Intelligent Data Analysis XVIII - 18th International Symposium on Intelligent Data Analysis, IDA 2020, Konstanz, Germany, April 27-29, 2020, Proceedings*. Vol. 12080. Lecture Notes in Computer Science. **Frontier Prize**. Springer, 2020, pp. 561–573. DOI: 10.1007/978-3-030-44584-3_44

6. Marcel Wever, Felix Mohr, and Eyke Hüllermeier. „Ensembles of evolved nested dichotomies for classification“. In: *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2018, Kyoto, Japan, July 15-19, 2018*. ACM, 2018, pp. 561–568. DOI: 10.1145/3205455.3205562
7. Felix Mohr, Marcel Wever, Alexander Tornede, and Eyke Hüllermeier. „Predicting Machine Learning Pipeline Runtimes in the Context of Automated Machine Learning“. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43.9 (2021), pp. 3055–3066. DOI: 10.1109/TPAMI.2021.3056950

In principle, these contributions can be divided into two parts.

In the first part of this thesis (Chapter 3 to Chapter 6, contributions 1.-4.), we devise a novel approach to AutoML that leverages techniques from AI planning to model the space of machine learning pipelines in terms of a search tree amenable to standard graph search algorithms, such as a best-first search. In contrast to previous works, this method can naturally reflect hierarchical dependencies of machine learning pipelines, respectively a multi-label classifier, in the search space model, and additionally allows one to systematically search the space via a global search. After a first version, which is able to configure machine learning pipelines consisting of one learner and at most one preprocessor, which is competitive to the state of the art, we demonstrate the flexibility and scalability of the approach, configuring machine learning pipelines of unlimited length. Subsequently, we transfer this method to the MLC setting and show that our approach is flexible and efficient enough to scale with the more complex search space. Moreover, we compare our approach to other state-of-the-art AutoML approaches that we either adapt to the MLC setting or that have already been explicitly proposed for MLC. In an extensive empirical study, we find that our method is indeed very well suited for the MLC setting and compares favorably with the other considered methods.

The second part of this thesis is concerned with the configuration of learners to increase their effectiveness (Chapters 7 and 8, contributions 5 and 6) and how meta-learning can improve the efficiency of AutoML systems (Chapter 9, contribution 7). To this end, we first consider the internal structure of a classifier, so-called *nested dichotomies*, which recursively decomposes the original learning problem into smaller sub-problems, thereby forming a tree-like model structure. We demonstrate that optimizing this tree structure can lead to improved generalization performance and thus increase the effectiveness of this learner.

Second, we consider *binary relevance learning*, a multi-label classifier that transforms an MLC problem into a set of binary classification problems, one per label, and employs SLC methods for the problems obtained. While previous literature configures

the SLC method jointly for all labels, we show that the effectiveness of this learner can be increased by tailoring the choice of the SLC method to each label.

Third, we show how meta-learning for predicting the runtime of machine learning pipelines can be helpful to render AutoML systems more efficient. More specifically, we provide empirical evidence for significant savings in wasted computation time for AutoML systems that evaluate candidate pipelines on the given data and impose a limit on the time for this evaluation. The presented method succeeds in using runtime prediction to avoid timeouts in the evaluation of machine learning pipelines.

In Chapter 10, the thesis is concluded, and an outlook on future directions and open questions is given. Last but not least, since this work has been done in the context of the aforementioned CRC 901 "On-The-Fly Computing", Chapter 11 is dedicated to the vision of *On-The-Fly Machine Learning*, which refers to the on-demand provision of customized machine learning services.

1.2 Running Example

Throughout the preliminaries in the following chapter, a continuous example is used to illustrate definitions and concepts. Suppose we are provided a set of landscape pictures such as shown in Figure 1.1, and we want to classify them by what is shown. Suppose that we want to label each picture with the help of the class labels $\mathbb{L} = \{\texttt{MOUNTAIN}, \texttt{SEA}, \texttt{FOREST}, \texttt{BEACH}\}$, depending on what is visible on the picture.

When looking at the pictures in Figures 1.1a to 1.1d, each of them can clearly be associated with one of these class labels. More specifically, Figure 1.1a shows a `BEACH`, Figure 1.1b a `FOREST`, Figure 1.1c a `MOUNTAIN`, and Figure 1.1d the `SEA`.

However, considering pictures as shown in Figures 1.1e to 1.1f, it is not that clear anymore. These images can no longer be assigned to a unique class label since they can be associated with multiple class labels simultaneously. In Figure 1.1e, both `BEACH` *and* `SEA` is shown, whereas `MOUNTAIN` *and* `SEA` are visible in Figure 1.1f. Furthermore, Figure 1.1g pictures three of the four class labels, namely `FOREST`, `MOUNTAIN`, and `SEA`. Figure 1.1h can be associated with the full set of class labels.

(a) BEACH

(b) FOREST

(c) MOUNTAIN

(d) SEA

(e) BEACH SEA

(f) MOUNTAIN SEA

(g) FOREST MOUNTAIN SEA

(h) BEACH FOREST MOUNTAIN SEA

Figure 1.1: Each of the landscape pictures is associated with class labels BEACH, FOREST, MOUNTAIN, and SEA. While the first four pictures can be related to one label exclusively, more than one class label is relevant for the last four pictures. The corresponding sets of labels are detailed in the captions.

2 Preliminaries

In this chapter, we provide a general overview of the field of automated machine learning (Section 2.1) and the learning problem of multi-label classification (Section 2.2).

2.1 Introduction to Automated Machine Learning

Automated machine learning (AutoML) refers to the vision of automating the process of selecting and configuring machine learning algorithms, composing them into so-called *machine learning pipelines*, tailored to a given task, i.e., a dataset and a target loss function. While AutoML was primarily intended to meet the substantial increase in demand for machine learning functionality, it indeed benefits experts in the field as well. Especially in multi-label classification, the variety of options to configure a multi-label classifier is overwhelming (cf. Section 2.2.6). Hence, a systematic and targeted optimization appears to be hardly possible, even for experts.

The formal definition of the AutoML problem, also referred to as combined algorithm selection and hyper-parameter optimization (CASH) [Tho+13], as the name suggests, is composed of the respective individual optimization problems: algorithm selection (AS) and hyper-parameter optimization (HPO).

Given an instance space $\mathcal{X}$ and a target space $\mathcal{Y}$, let

- $\mathcal{A} = \{A^{(1)}, \ldots, A^{(n)}\}$ be a set of algorithms with corresponding hyper-parameter spaces $\Lambda^{(1)}, \ldots, \Lambda^{(n)}$,
- $\mathcal{D} = (X, Y) \subset (\mathcal{X}^N \times \mathcal{Y}^N) \subset \mathbb{D}$ a labeled data set from the data set space $\mathbb{D}$,
- and $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \mapsto \mathbb{R}$ a target loss function to be minimized.

In the context of machine learning, an algorithm $A : \Lambda \times \mathbb{D} \times \mathcal{X}^S \longrightarrow \mathcal{Y}^S$ refers to a learning algorithm that can be configured with a hyper-parameter configuration $\lambda \in \Lambda$ (we write A_λ in the following). Furthermore, A takes a data set $\mathcal{D} \in \mathbb{D}$ and a batch of instances $X_{\text{test}} \subset \mathcal{X}^S$ of size S as arguments. The data set $\mathcal{D}$ is used by A internally as training data to induce a hypothesis $\widehat{h}$ from some hypothesis space $\mathcal{H} \subseteq \{h \mid h : \mathcal{X} \longrightarrow \mathcal{Y}\}$. This hypothesis $\widehat{h}$ is then used to produce predictions on a batch of test instances $X_{\text{test}} \subset \mathcal{X}^S$ of size S by applying $\widehat{h}$ to each of the $\boldsymbol{x} \in X_{\text{test}}$ individually. Eventually, A returns the concatenation of all predictions $\widehat{Y} \in \mathcal{Y}^S$.

We further assume that every algorithm A has a default parameterization $\lambda_{\text{def}} \in \Lambda$. For convenience, if λ corresponds to the default parameterization λ_{def}, we write A instead of $A_{\lambda_{\text{def}}}$.

Then, in AS, the task is to select the most suitable algorithm A^* with respect to the target loss function $\mathcal{L}$, i.e.,

$$A^* \in \underset{A^{(i)} \in \mathcal{A}}{\arg\min} \int_{\mathcal{X} \times \mathcal{Y}} \mathcal{L}(\boldsymbol{y}, A^{(i)}(\mathcal{D}, \boldsymbol{x}))\, P(\boldsymbol{x}, \boldsymbol{y})\, d\boldsymbol{x} d\boldsymbol{y}\ , \tag{2.1}$$

where $P(\cdot, \cdot)$ is a joint probability distribution for $\boldsymbol{x}$ and $\boldsymbol{y}$. The selection is only made over the finite set of algorithms $\mathcal{A}$. Thus, in this sense, it does not consider different parameterizations of an algorithm. However, by considering multiple hyper-parameter configurations of an algorithm A again as a distinct algorithm [TWH20a], it is possible to include the optimization of hyper-parameters at least to a (very) limited extent.

In turn, the HPO problem fixes an algorithm A in advance and deals with the optimization of the respective hyper-parameters. Given the hyper-parameter space Λ of A, the optimization problem can be stated as follows.

$$\lambda^* \in \underset{\lambda \in \Lambda}{\arg\min} \int_{\mathcal{X} \times \mathcal{Y}} \mathcal{L}(\boldsymbol{y}, A_\lambda(\mathcal{D}, \boldsymbol{x}))\, P(\boldsymbol{x}, \boldsymbol{y})\, d\boldsymbol{x} d\boldsymbol{y} \tag{2.2}$$

However, the specific algorithm is fixed in this problem, and only its hyper-parameters are considered for optimization. Therefore, in CASH, as stated first in [Tho+13], the two optimization problems are combined into a joint one, where we seek to find an algorithm A^* together with its hyper-parameter configuration λ^*, minimizing the target loss $\mathcal{L}$:

$$A^*_{\lambda^*} \in \underset{A^{(i)} \in \mathcal{A},\, \lambda \in \Lambda^{(i)}}{\arg\min} \int_{\mathcal{X} \times \mathcal{Y}} \mathcal{L}\left(\boldsymbol{y}, A^{(i)}_\lambda(\mathcal{D}, \boldsymbol{x})\right)\, P(\boldsymbol{x}, \boldsymbol{y})\, d\boldsymbol{x} d\boldsymbol{y}\,. \tag{2.3}$$

In the subsequent Section 2.1.1, we deal with the concept and different shapes of machine learning pipelines, which are subject to optimization in AutoML. Furthermore, since the introduction of the CASH problem in 2013 [Tho+13], a variety of methods have been developed more or less following a general schema which is described in Section 2.1.2. These methods can be divided into three major groups: Reducing the CASH problem to the HPO problem, using grammar-based search approaches, and leveraging meta-learning techniques. We provide details on these three groups in Sections 2.1.3, 2.1.4, and 2.1.5 respectively. Going beyond CASH, we discuss a sub-field of AutoML called neural architecture search (NAS) in Section 2.1.6, which deals exclusively with the optimization of neural networks. For a more detailed and

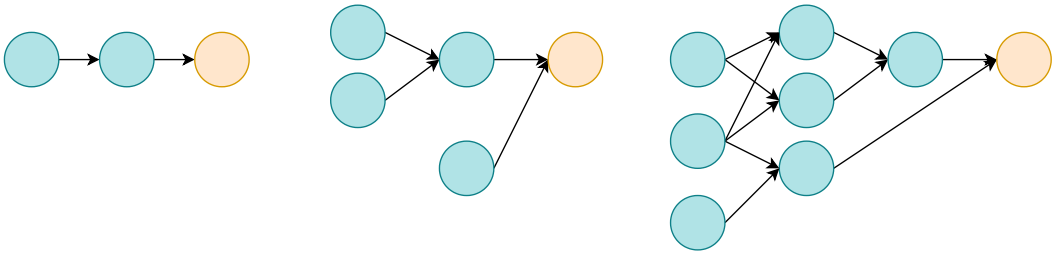


Figure 2.1: Visualization of different machine learning pipeline topologies. On the left-hand side, a sequential pipeline is shown. The center of the figure presents a tree-shaped pipeline topology, and on the right-hand side, the pipeline structure represents a directed acyclic graph.

in-depth overview of AutoML, we refer the interested reader to surveys shedding light on the field from diverse perspectives [Van18; Yao+18; ZH21; HZC21].

2.1.1 Machine Learning Pipelines

To achieve the best possible performance for a given data set, several machine learning algorithms are often combined into a so-called *machine learning pipeline*. In its simplest form, a machine learning pipeline consists solely of a learning algorithm, and the provided data set is directly used for training. However, depending on the shape of the data and properties of the data and the learning algorithm, some preprocessing of the data might become desirable.

For example, in the case of image tagging, as described in Section 1.2, the pictures may need to be preprocessed to make the data amenable to machine learning algorithms since most of these algorithms cannot deal with image data directly. In this case, we would need a machine learning pipeline including an algorithm transforming images into, for instance, a vector representation. Furthermore, it may prove beneficial to "manipulate" the images beforehand, e.g., by applying a grey-scale filter, Gaussian filter, or an edge detection algorithm. What kind of preprocessing is needed depends on the data, the task, and also the learning algorithm. For example, neural networks may be directly applied to the pictures, whereas a support vector machine will probably require a more sophisticated preprocessing to perform well. However, preprocessing is not only required for such unstructured data. If the learning algorithm cannot deal with nominal features, but the data contains such features, it is inevitable to preprocess the data making it amenable to this learning algorithm. In this case, a preprocessing algorithm could be applied to encode the nominal features in terms of numeric values, e.g., by mapping each attribute value to a specific number.

Modern machine learning libraries, such as WEKA [EHW16], sckit-learn [Ped+11], or mlR [Bis+16], comprise a variety of such preprocessing algorithms. In fact, even learning algorithms can be used as a preprocessor to augment the dataset by predictions of this learning algorithm as yet another feature. An application of preprocessing algorithms can be done both sequentially and in parallel.

Figure 2.1 shows different types of pipeline structures: A sequential, tree-shaped, and a DAG-shaped[1] pipeline are visualized. When executing a pipeline, as a first step, as many copies of the data are created as there are algorithms without a predecessor, i.e., nodes with no incoming edge. After applying the respective algorithm, the preprocessed data is forwarded to the algorithms that are referenced via an outgoing edge. If there are multiple incoming edges, the union of all incoming data is taken. Eventually, the preprocessed data arrives at the learning algorithm, which, in the figure, is highlighted in orange.

2.1.2 General Structure of AutoML Systems

The majority of AutoML systems consist of three main components: A specification of the search space that encodes what solution candidates are available, an optimization algorithm, and a function to evaluate solution candidates. Provided a task as an input, which is specified via a data set $\mathcal{D}$ and a target loss function $\mathcal{L}$, the optimization algorithm traverses the search space to find a machine learning pipeline that is most suitable for the given task. Eventually, the best-found machine learning pipeline is returned to the user. An illustration of this general structure is shown in Figure 2.2.

Generally speaking, the search space representation describes which algorithms are available for selection, which hyper-parameters they expose, and how they can be combined into a machine learning pipeline. Note that the search space representation is a *model* of the actual search space, i.e., some information or properties of the search space might not be reflected in the search space representation. Hence, the search space representation can also artificially add information or structures to make the optimization algorithm more effective or efficient, e.g., by grouping similar algorithms. Typically, the search space representation is designed by an expert and is an integral part of the respective AutoML system.

The optimization algorithm operates on the search space representation and seeks to identify the most suitable solution candidate. For strategically traversing the search space, the optimization algorithm requires some kind of feedback on the solution quality of candidate machine learning pipelines. As specified in the problem of Equation (2.3), we seek the machine learning pipeline that generalizes best, i.e.,

[1] DAG: directed acyclic graph.

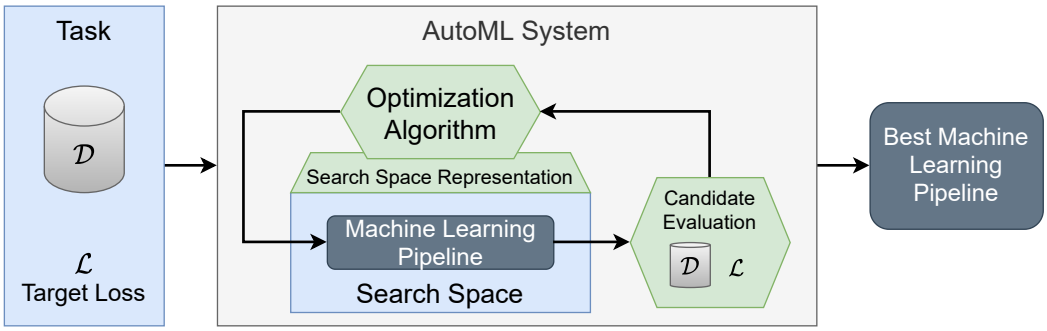


Figure 2.2: Generic illustration of the AutoML framework. Receiving a task as an input containing a training data set $\mathcal{D}$ and a target loss function $\mathcal{L}$, the AutoML system aims to identify a machine learning pipeline that generalizes well beyond the provided training data. To this end, AutoML systems usually comprise three major components: a search space representation, an optimization algorithm operating on this search space representation, and a candidate evaluation module to assess the solution quality of candidates. Typically, the candidate evaluation uses the provided dataset and the target loss to estimate a candidate's generalization performance.

minimizes the risk. However, since the out-of-sample error cannot be computed, we estimate a machine learning pipeline's generalization performance with the help of the provided data set $\mathcal{D}$. To this end, once commonly splits the data set (randomly) into training $\mathcal{D}_{\text{train}}$, which is used by the algorithm to induce a hypothesis $\widehat{h}$, and validation data $\mathcal{D}_{\text{val}} = (X_{\text{val}}, Y_{\text{val}})$ for estimating the quality of predictions. The observed loss on the validation data is returned as feedback to the optimization algorithm. In fact, we thus solve the optimization problem

$$A^*_{\lambda^*} \in \underset{A^{(i)} \in \mathcal{A},\, \lambda \in \Lambda^{(i)}}{\arg\min} \; \mathbb{E}_{\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{val}}} \left[\mathcal{L} \left(Y_{\text{val}}, A^{(i)}_{\lambda}(\mathcal{D}_{\text{train}}, X_{\text{val}}) \right) \right] \tag{2.4}$$

as a surrogate for the actual problem (2.3). Here, the expectation accounts for any randomness of algorithms $A^{(i)}_{\lambda}$, the data itself, and the precise way how the algorithms are validated, e.g., via k-fold cross-validation, a hold-out set, etc.

As already mentioned, to determine the loss of algorithm $A^{(i)}_{\lambda}$, it is typically executed on the given data. Specifically, this means that executable implementations of the respective algorithms are required to run the optimization process. Therefore, the search space descriptions are usually based on software libraries for machine learning such as scikit-learn [Ped+11; Var+15], WEKA [EHW16], or in the case of multi-label classification MEKA [Rea+16] and Mulan [TKV09].

2.1.3 Reduction to Hyper-Parameter Optimization

While the very first attempt to AutoML was made in [Ber+11] employing a random search, the first AutoML systems going beyond a simple random search perform a reduction from CASH to HPO [Tho+13; Kot+17; Feu+15], making it amenable to well researched and sophisticated HPO approaches, such as Bayesian optimization, multi-armed bandits, or genetic algorithms. To this end, the choice of algorithms is encoded as yet another (categorical) hyper-parameter and combined into a joint hyper-parameter vector together with all the hyper-parameters exposed by the respective algorithms.

In terms of the HPO problem (2.2), we consider an (artificial) meta-algorithm $\widehat{A}$ with hyper-parameter space $\widehat{\Lambda} = \{\lambda_A \mid \lambda_A \in \{1, \dots n\}\} \times \Lambda^{(1)} \times \dots \times \Lambda^{(n)}$. Depending on the concrete choice of the algorithm via the categorical hyper-parameter λ_A, only a small subset of hyper-parameters is relevant for optimization, namely the ones belonging to the selected algorithm $A^{(\lambda_A)}$. In the following, we refer to these hyper-parameters interchangeably as *active* hyper-parameters. Which hyper-parameters are considered active is usually modeled in an auxiliary conditional structure.

Random search Arguably, the simplest and most straightforward way of approaching the HPO problem is via random search [Ber+11; Gil+18; LP20]. In random search, solution candidates $\lambda \in \widehat{\Lambda}$ are sampled randomly and evaluated with respect to the given task. Despite its simplicity, a random search can yield reasonably good results [Gij+19]. Another advantage of random search is that it can be parallelized arbitrarily, making it particularly appealing for distributed systems and cloud environments.

Bayesian Optimization Bayesian optimization (BO) [Fra18] is an often-used approach for HPO, not least because of its efficiency and theoretical guarantees of convergence. Generally speaking, BO is a technique for optimizing black-box functions $f : U \longrightarrow \mathbb{R}$, where U denotes some input domain (e.g., $\mathbb{R}^N$) and the evaluation of f is assumed to be expensive. Furthermore, it does not require additional knowledge about f, such as gradients, convexity, linearity, etc.

The optimization procedure of BO, as depicted in Figure 2.3, comprises mainly two components: (i) a surrogate $\widehat{f}$ of f, which is cheap to evaluate, and (ii) an *acquisition function* forming the basis for decisions on which input to evaluate the actual function f next. The surrogate is represented by a probabilistic model estimated from the actual function evaluations of f that have been observed so far. The most commonly used models are Gaussian processes [SLA12], random forests [HHL11], and Tree-structured Parzen Estimator [BYC13; HHL11]. An essential

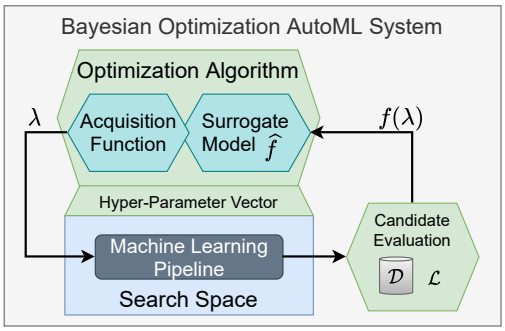


Figure 2.3: Illustration of an AutoML system employing Bayesian optimization. Solution candidates are represented in terms of a hyper-parameter vector. The surrogate model $\widehat{f}$ models the actual evaluation function f to be optimized. Furthermore, $\widehat{f}$ is used by the acquisition function to decide, which hyper-parameter configuration λ to sample next. Prior to evaluation, the chosen λ is translated into a machine learning pipeline. Then, $f(\lambda)$ augments the set of observations of f, which in turn updates the surrogate model $\widehat{f}$.

capability of these surrogate models is to provide information about the expected values – and thus possible optima of f – and quantify their uncertainty about these predictions. Based on this information, the acquisition function is tasked to trade-off exploration and exploitation to sample f efficiently. Here, exploration refers to "exploring" configurations for which the predictions of the surrogate $\widehat{f}$ has high uncertainty, whereas exploitation means to evaluate configurations for which the surrogate model predicts a high performance.

In the context of HPO, the black-box function f of interest is the (estimated) performance of a hyper-parameter configuration in terms of a loss function $\mathcal{L}$. Producing this information involves fitting and validating the algorithm with the respective hyper-parameter configuration, which can indeed be considered expensive as it can take up to minutes or even several hours. Consequently, the number of function evaluations that can be afforded is generally very limited.

An example of such an acquisition function is the expected improvement (EI) [Moc77; JSW98]. The basic idea of EI is to evaluate the hyper-parameter configuration next, which maximizes the improvement over the previous best solution candidate, as would be expected according to the surrogate model. Formally, the EI for a hyper-parameter configuration λ with respect to the best hyper-parameter configuration $\widehat{\lambda}^*$ observed so far can be computed via

$$EI(\lambda) = \mathbb{E}\left[\max\left(f(\widehat{\lambda}^*) - f(\lambda), 0\right)\right], \tag{2.5}$$

where $f(\hat{\lambda}^*)$ is the result of executing f for the hyper-parameter configuration $\hat{\lambda}^*$, i.e., the best outcome observed so far, $f(\lambda)$ is a random variable with an unknown outcome at the time of the computation of $EI(\lambda)$. Practically, in order to compute

$EI(\lambda)$, we make use of the surrogate model $\hat{f}$ to estimate the mean and the variance of $f(\lambda)$.

Based on BO, several approaches to AutoML have been proposed. Probably the better-known representatives of these AutoML systems are Auto-WEKA [Tho+13; Kot+17] and auto-sklearn [Feu+15]. However, various extensions and enhancements have been proposed [Zha+16; FH18; MS19; KBE19] but also specializations focusing on single algorithms [TCB18; JSH19].

Multi-Armed Bandits Formalizing the problem as a multi-armed bandit (MAB) problem [Sli19] is another way of tackling the CASH problem as a reduction to HPO. MAB denotes a sequential decision-making problem in which an agent must repeatedly select options from a given set of alternatives in an online setting. The metaphor of the eponymous slot machines in casinos associates the available options with the "arms" of the slot machine that are "pulled", i.e., the respective option is selected. By pulling an arm, the agent is provided a *reward* signal as feedback on the quality of his or her choice. Typically, only one arm is pulled at a time, and the goal is to optimize a certain evaluation criterion, e.g., maximizing the cumulative reward. To achieve such a goal, the agent must carefully balance exploration, to possibly find a better arm, but at the same time risking to draw arms yielding less reward, with exploitation, to take advantage of the arms already known to yield high rewards.

Instantiating the MAB problem for HPO, each arm represents a hyper-parameter configuration, and the aim of the agent is to identify the best arm. Moreover, the reward signal for pulling an arm is provided by evaluating the corresponding hyper-parameter configuration for a particular budget b, e.g., time or number of observations used for training. The agent may choose and adapt the budget b for a hyper-parameter configuration over time. Consequently, this requires that the candidate evaluation function f needs to be parameterizable in the budget b.

Based on the MAB setting, an approach to HPO is proposed in [JT16] applying the successive halving (SH) algorithm [KKS13]. Theoretically and empirically, the proposed approach is shown to yield good performance. Initially sampling a representative set of hyper-parameter configurations at random, as the name suggests, it successively disregards half of the worse performing configurations, based on the observed average performance after a certain amount of the available budget is reached. An illustration of this approach is provided in Figure 2.4.

Note that the set of all hyper-parameter configurations is usually huge, if not infinite, which is already the case as soon as real-valued parameters are considered. However, in [JT16], this problem is addressed by sampling a predefined number of configurations, thus, only presenting a finite set of configurations to SH. After

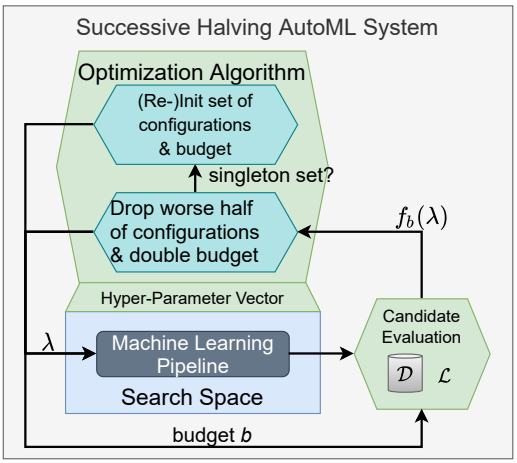


Figure 2.4: Illustration of an AutoML system, employing a successive halving algorithm for optimization with a budgeted candidate evaluation function $f_b(\cdot)$. After picking an initial set of configurations and initial budgets, the algorithm iteratively evaluates configurations for the current budget b, discards the worse half of configurations, and re-evaluates the remaining for an increased budget. This process is run multiple times, varying the initial budget and the size of the initial set of configurations.

each halving, the remaining configurations are evaluated for an increased budget. In this way, SH allows for wasting less budget on poorly performing configurations and for allocating more budget for the more promising ones, finding the best arm after $\lceil \log_2(N) \rceil$ iterations, where N is the number of initially sampled hyper-parameter configurations.

However, defining the value of N has a practical impact on the choice of the final arm [Li+17]. Mainly, this is due to (i) sampling too few configurations may lead to missing good ones, and (ii) sampling too many configurations decreases the budget initially assigned for evaluation such that configurations that perform better on larger budgets are disregarded early. Hence, the number of initially sampled configurations is a crucial hyper-parameter to the optimization algorithm. Depending on the maximum allocatable budget, in [Li+17], the authors propose a heuristic, which they dub Hyperband, to run SH for different numbers of initially sampled configurations repeatedly. As an optimization algorithm of an AutoML system, Hyperband performs competitively to systems based on Bayesian optimization [DSR18].

Hybrid Bayesian Optimization and Multi-Armed Bandits A significant disadvantage of Hyperband is that knowledge about already observed performances from previous iterations is not incorporated for sampling new sets of initial configurations. To overcome this limitation, the idea of Hyperband is integrated with the Bayesian optimization approach leading to a hybrid algorithm called Bayesian Optimization and Hyperband (BOHB) [FKH18]. In fact, the random sampling procedure of Hyperband is (mostly) substituted by sampling configurations according to the surrogate

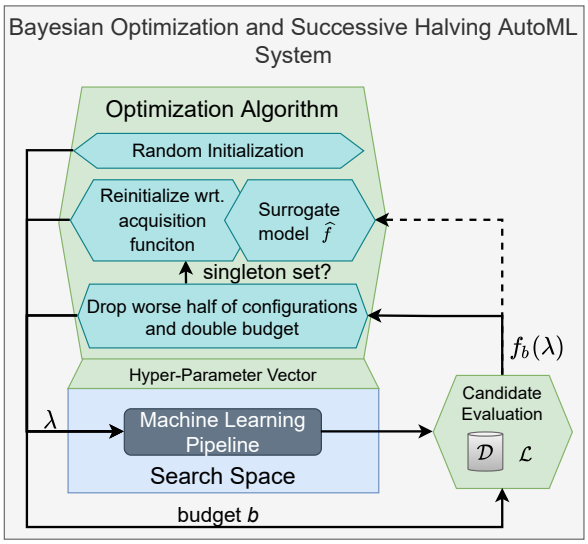


Figure 2.5: Illustration of an AutoML system combining Bayesian optimization and Hyperband (BOHB) into a hybrid optimization algorithm leveraging the best out of the two worlds: Successive halving as in Hyperband together with a model-based sampling of promising candidates when reinitializing the set of configurations.

$\widehat{f}$ and an acquisition function. Due to technical reasons, some of the configurations are still sampled at random to ensure convergence. The combination of BO and Hyperband is visualized in Figure 2.5. It has been applied to address the AutoML problem in [Swe+17] and [Feu+18].

Others Beyond the discussed methods, other approaches reducing the AutoML problem to HPO exist, for example, by using standard genetic algorithms [SPF17; WMH18b], or via grid search to successively build a stacking ensemble [Eri+20].

Another interesting approach is presented in [Liu+20], leveraging the alternating direction method of multipliers (ADMM) algorithm, where a bipartition of the numeric and categorical hyper-parameters is created. Thereby, the original optimization problem is decomposed into simpler sub-problems (concerning the number of variables), containing only variables of a single type. Furthermore, the ADMM framework allows for introducing additional (external) constraints on the optimization problem, such as fairness or robustness constraints.

2.1.4 Grammar-Based Search

The reduction of the CASH to the HPO problem has a considerable disadvantage with regard to more flexible machine learning pipelines. When working with a hyper-parameter vector as a search space representation, its size is fixed, and therefore also a maximum length or number of components within the pipeline is predetermined. Another branch of AutoML systems emerged to overcome this limitation that uses

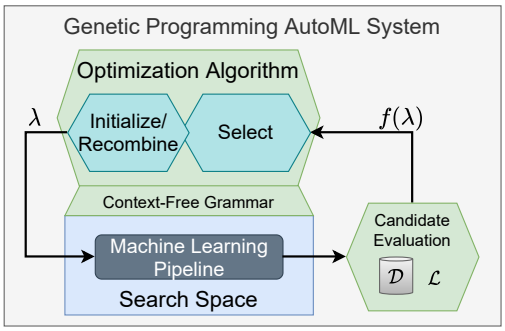


Figure 2.6: Illustration of an AutoML system employing genetic programming as an optimization algorithm. As common in evolutionary algorithms, genetic programming maintains a population of solution candidates, also referred to as individuals. The fitness values $f(\cdot)$ computed for individuals are used to select more promising ones and use them as input for recombination operators. The distinctive feature of genetic programming is that individuals are represented in the form of trees. In the AutoML domain, these trees are derivation trees of some context-free grammar describing the space of potential machine learning pipelines.

a grammar-based representation of the search space. This representation does not impose any limitations on the maximum length or the number of components contained in a machine learning pipeline. Furthermore, it allows for describing different shapes of machine learning pipelines – e.g., sequential, tree-shaped, and even in the shape of a directed acyclic graph – compared to a fixed structure as predefined in the reduction to HPO.

Another advantage of grammar-based approaches is that they can also naturally represent recursive dependencies. The configuration of learning algorithms, especially meta-learning algorithms, can become quite complex. This is because it requires the configuration of a base algorithm which in turn is a learning algorithm exposing hyper-parameters and needs to be configured by the AutoML system as well.

Grammar-Based Genetic Programming Grammar-based optimization approaches are pretty prevalent in the field of evolutionary algorithms, especially in the subfield of grammar-based genetic programming (GGP) [Koz95; McK+10]. In the original sense, genetic programming is about synthesizing software with the help of evolutionary algorithms [Cra85], assessing the fitness $f(\cdot)$ of individuals, for example, in terms of test coverage to verify that the program works as expected. Instead of representing individuals in terms of gene strings, GGPs employ a formal language or grammar to describe the space of solutions, and individuals are derived based on this grammar. More precisely, individuals in GGPs describe a derivation tree for this grammar.

Using GGPs for AutoML has the advantage that the grammar allows much more flexible structures for machine learning pipelines than a fixed-length hyper-parameter

vector as needed when applying BO. An illustration of a genetic programming AutoML system is given in Figure 2.6. AutoML systems based on genetic programming support tree-shaped pipelines [Ols+16; Sá+17] and even support representations for more complex structures such as pipelines involving feature extraction [Tor+21] or pipelines exhibiting a directed acyclic graph structure [Che+18]. Moreover, GGPs also natively allow for recursive structures as in the case of configuring multi-label classifiers [SFP18]. However, as is typical for evolutionary algorithms, GGPs maintain a population of individuals of a specific size. On the one hand, parallel processing of the individuals is trivial, but on the other hand, the size of the population is also a crucial hyper-parameter that needs to be chosen with care. While a smaller population allows for faster processing of generations, initializing a population that covers the search space in a sufficiently representative way is difficult. A larger population, in turn, may provide better coverage of the search space, but on the other hand, it also increases the computational costs for evaluating the offspring of each generation. Moreover, the next generation does not begin until the previous one is complete, so single expensive-to-evaluate individuals may stall the entire evolution. This is especially an issue when dealing with larger data sets where the evaluation of single candidates does not take seconds but rather minutes or hours.

To overcome these limitations and to improve the scalability of TPOT [Ols+16], it has been extended to leverage successive halving for fitness evaluations [Par+19] and to include a feature set selector as another algorithm that can be used in a candidate pipeline [LFM20]. Another work addresses the issues of waiting for a generation to finish before beginning a new one employing an asynchronous GGP variant [GV19]. More specifically, this GGP continuously recombines and evaluates individuals without the synchronization barriers of generations.

AI Planning and Graph Search Hierarchical task network (HTN) planning [GNT04] originates from the field of automated planning and scheduling, sometimes simply referred to as AI planning, and deals with the automated production of *plans*, i.e., sequences of actions, that are typically executed by an (intelligent) agent. In HTN planning, the idea is to structure the search space hierarchically based on a logic language and operators defined on that language.

A plan or a solution is derived in HTN planning by refining so-called tasks arranged in a partially ordered set of tasks, also referred to as task network. Tasks can either be *complex* or *primitive*. While the latter represent actionable items, i.e., actions that the agent can execute, the former needs to be refined until only primitive tasks are left. Complex tasks can be viewed as a composition of simpler tasks and thus need to be refined via so-called *methods* into those simpler tasks, which can be again complex or primitive. The way the search space is described is strongly reminiscent of context-free grammars, with complex tasks corresponding to non-terminals, primitive tasks

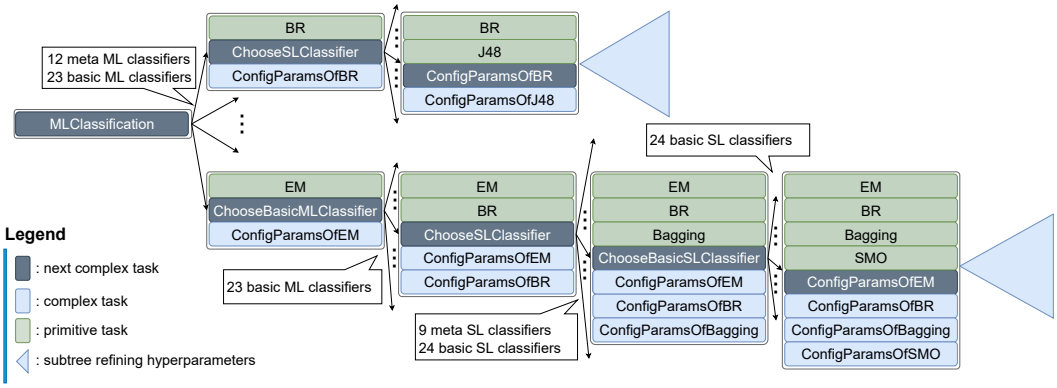


Figure 2.7: Creation of pipelines with hierarchical planning. **Top:** A binary relevance (BR) learning classifier is configured with a decision tree, which is called J48 in WEKA, as a base learner. **Bottom:** First, a meta multi-label classifier expectation maximization (EM) is selected and configured with a binary relevance learning classifier as a base learner, which in turn employs a bagged SMO classifier ensemble as a base learner for the individual labels.

to terminals, and methods to production rules. However, the difference here is that methods can impose additional constraints in the form of preconditions that must be satisfied for the method to be applicable to a particular state.

To solve HTN-planning problems, they are usually reduced to graph search problems, making them accessible to standard graph search algorithms, such as breadth-first or depth-first search. This reduction is often made by *forward-decomposition* [GNT04], which selects the *first* complex task in the network of a node to be refined. The node's successors are then obtained, considering each method that can be applied to refine the selected task. In this way, every inner node of the induced search graph corresponds to a plan prefix, i.e., the primitive tasks that have already been fixed, and to a task network containing the remaining complex tasks that still need to be refined. Hence, the start node represents an empty plan prefix and a task network containing the initial complex task. Leaf nodes, in turn, represent complete plans and empty rest networks. In Figure 2.7 a concrete example is depicted, where the initial complex task "MLClassification" is decomposed in plan prefixes. Already fixed primitive tasks of the plan prefixes are shown in green, whereas complex tasks are colored in blue. The next complex task to be refined is highlighted in dark blue.

In the context of automating data mining and machine learning, HTN planning, respectively an extension of HTN planning called programmatic task network (PTN) planning [Moh+18a], has been used for modeling algorithm choices and hyperparameter values in terms of primitive tasks and introducing complex tasks as abstraction layers for grouping different kinds of algorithms and defining structures such as the topology of a machine learning pipeline [Kie+12; WMH17; MWH18b; WMH18c; Kat+20]. In [MWH18b; WMH18c] (Chapters 3 and 4), we propose an

AutoML system named ML-Plan, applying a best-first search to the resulting search graph which requires each node to be assigned a heuristic score. Since an inner node of the search graph corresponds to an incomplete specification of a machine learning pipeline and thus, cannot be evaluated as a candidate, random completions are drawn to leaf nodes. The machine learning pipelines represented by the respective leaf nodes are then evaluated, and the observed performances are aggregated at the inner node, e.g., taking the minimum. The number of random completions affects the trade-off between exploration and exploitation. While a larger number promotes exploration, a smaller number reinforces the greediness of the search and consequently exploitation.

Beyond single-label classification, HTN planning was also used in extensions of ML-Plan to other types of tasks, such as remaining useful lifetime estimation in the problem domain of predictive maintenance [Tor+20b], and multi-label classification (Chapters 5 and 6) [WMH18a; Wev+19; Wev+21].

Representing the search space as a search graph or a search tree, other heuristic graph search methods can be successfully applied as optimization algorithms of AutoML systems. For example, in [RSS19] a Monte-Carlo tree search is applied along with a progressive widening strategy [Cha+08] to make the high branching factor manageable. In [MBH21], the authors suggest the use of a Plackett-Luce model [CML13] to improve Monte-Carlo tree search for single-player games, e.g., AutoML. An adaptation of AlphaZero [Sil+17], which also employs a Monte-Carlo tree search, is proposed in [Dro+18]. Another approach combines reinforcement learning for selecting algorithms with Bayesian optimization for configuring the hyper-parameters of machine learning pipelines [SLB19; Lin19]. During a hyper-parameter optimization phase, the best performance is fed back as a reward to the reinforcement learning algorithm.

2.1.5 Meta-Learning

A major criticism of AutoML systems is that they are too resource-intensive in terms of both computational power as well as time. Following the common design scheme, as shown in Figure 2.2, AutoML systems heavily rely on actually executing candidate machine learning pipelines in order to estimate their generalization performance for the given task. Besides the high computational costs, the execution of machine learning pipelines takes some time ranging from milliseconds to hours or even days – depending on the data and algorithms contained in the candidate pipelines. Hence, also the AutoML process can be pretty time-consuming.

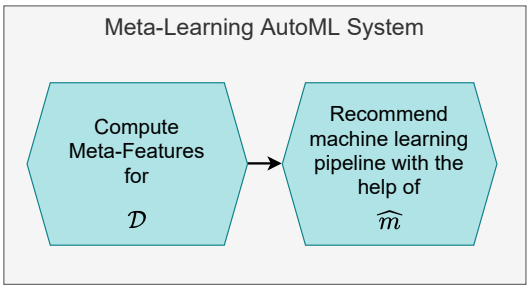


Figure 2.8: AutoML systems using meta-learning for recommending machine learning pipelines usually require a feature representation of datasets which needs to be computed for the given dataset $\mathcal{D}$, before the meta-model $\widehat{m}$ can be queried to obtain a recommendation. Note that this AutoML system does not involve a candidate evaluation, which would require the machine learning pipelines to be executed for the given data set $\mathcal{D}$.

One possible way to address these issues is through meta-learning. Inspired by the way humans learn, meta-learning (also known as "learning to learn") aims to accumulate experience (metadata) on already solved tasks to accomplish future tasks faster and/or more accurately. Transferred to the AutoML problem, this translates to observing performance data of machine learning pipelines on training data sets and, based on this, estimating a model to make recommendations for unseen data sets. On the one hand, meta-learning can be used to support and speed up existing AutoML systems, e.g., by first evaluating the best machine learning pipelines known for similar (already examined) datasets, as it is done in [Feu+15; MS19]. Moreover, in Chapter 9, we advocate the use of meta-learning for predicting runtimes of machine learning pipelines to decide whether a pipeline should be evaluated or not. Avoiding evaluations of machine learning pipelines that take too long significantly improves the efficiency of AutoML systems [Moh+21]. Alternatively, one may think of substituting the entire AutoML system with a predictor that recommends a machine learning pipeline for a new (unseen) data set.

Rather than treating each dataset independently, as suggested by the original problem statement in Equation (2.3), with the help of a trial-and-error-based AutoML system, we seek to learn a mapping $\widehat{\varphi}$ from the space of datasets to the space of algorithms and their parameterizations, as defined in the introduction of Section 2.1.

$$\widehat{\varphi} : \mathcal{D} \longrightarrow \{(A^{(i)}, \lambda) \mid A^{(i)} \in \mathcal{A} \text{ and } \lambda \in \Lambda^{(i)}\}$$

$\widehat{\varphi}$ represents a surrogate of the underlying ground truth function φ, assigning each dataset the best machine learning pipeline. Note that, in general, φ cannot be observed as the ground truth assignment of a machine learning pipeline to a data set can usually not be determined with absolute certainty since only *estimates* on the generalization performance are observable.

By simplifying the CASH problem to algorithm selection and covering different parametrizations in terms of (many) "different" algorithms, the problem can be solved, for example, by probabilistic matrix factorization [FSE18; Yan+19] or by leveraging learning algorithms, in turn, to predict performances of machine learning pipelines [TWH20a]. In the latter work, we use a dyadic feature representation, comprising a feature representation of data sets and a feature representation of algorithms, allowing the model to generalize across both dimensions data sets and algorithm parametrizations. While in the first place, this dyadic feature representation improves data efficiency of the meta-learner, i.e., only a few training observations are required to make highly accurate predictions, in principle, it technically allows for making predictions on unknown parametrizations of the already known algorithms. However, whether these meta-models generalize well to unknown parametrizations is still an open question.

A key role for good generalization performance, and thus, the successful application of such meta-models lies in the informativeness of the so-called meta-features, i.e., features describing properties of the data sets and – in the case of a dyadic feature representation – algorithms. Often, statistical features (such as the number of instances, features, classes, etc.) and performances of fast-to-evaluate learners, also known as landmarking features, are used to describe datasets. A more sophisticated way is proposed in [Dro+19] leveraging deep language models to transform natural language descriptions of data sets into a numeric feature representation. Although natural language embeddings seem promising [JSG21], coming up with a suitable feature representation is still an open research area.

2.1.6 Neural Architecture Search

Neural architecture search (NAS) denotes a particular branch of AutoML committed to and specialized in optimizing neural networks, especially on optimizing the topology of neural networks. Initial work in this sub-field of AutoML uses evolutionary algorithms for optimization and introduces an abstraction layer by defining building blocks also known as *cells* that can be composed to obtain the final architecture [Rea+17]. Although they are evolved from scratch, neural networks optimized via NAS manage to match the performance of highly sophisticated neural networks designed by human experts. Since the seminal work [Rea+17], a plethora of other approaches and improvements have been proposed to make the search for neural architectures more resource-efficient or to reduce the number of internal parameters while maintaining a competitive generalization performance [Wis18; Zhu+19; Xie+20]. Besides, NAS has also been used for optimizing neural networks for multi-label classification [PN19]. For a more comprehensive overview of NAS, we refer the interested reader to overview papers on NAS [EMH19; WRP19; Ren+20].

Although the increased interest in NAS – presumably due to the success of neural networks in image classification – seems rather recent, the optimization of neural network architectures has already been a topic of interest in the evolutionary computation community since the 80s and 90s [HSG89; Gru94]. For example, in the field of evolutionary robotics, controllers for robots are implemented via neural networks, which are optimized by evolutionary algorithms [NF00]. The optimization involves the internal parameters and the neural network architecture, i.e., individual neurons and connections between neurons. However, for image classification tasks, the neural network architectures are larger by several orders of magnitude, and a key component for making NAS approaches succeed is to work with the abstract entity of cells instead of the neurons and connecting edges directly.

2.2 Introduction to Multi-Label Classification

Multi-label classification (MLC) refers to a special form of a multi-target prediction problem [WKH19], where data points (instances) are associated with binary targets denoting the "relevance" or the "irrelevance" of a specific property of interest, which is represented by a class label. Moreover, it generalizes the more common classification tasks of binary and multi-class classification, where only a single such class label is associated with each instance. In MLC we aim to learn a predictor, mapping instances to *subsets of* (presumably) *relevant labels*. As an example, consider the image tagging example of Section 1.2 where a picture can be associated with multiple labels, e.g., `BEACH`, `FOREST`, `MOUNTAIN`, and `SEA`, at the same time. For a more comprehensive and in-depth overview of multi-label classification, we refer the interested reader to the survey articles [TKV10] and [ZZ14].

Subsequently, we give a formal definition of the MLC learning problem in Section 2.2.1. Furthermore, we elaborate on how single-label classification, i.e., binary and multi-class classification, can be framed as a special instance of MLC in Section 2.2.2. In Section 2.2.3, we discuss various loss functions that are typically used for assessing the quality of predictions. Then, we give an overview of different ways of approaching the MLC learning problem in Section 2.2.5. After elaborating on the concept of label dependence in more detail in Section 2.2.4, we conclude this introduction by elaborating on the challenge of configuring multi-label classifiers in Section 2.2.6.

2.2.1 Problem Definition

Let $\mathcal{X}$ denote the space of instances and $\mathbb{L} = \{l_1, \dots l_m\}$ a finite set of m labels. In MLC, we assume each instance $\boldsymbol{x} \in \mathcal{X}$ to be (non-deterministically) associated with

a subset of labels $L \subseteq \mathbb{L}$ via a joint probability distribution $P(\cdot, \cdot)$ for $\boldsymbol{x}$ and L. We call L the set of *relevant labels* and its complement $\mathbb{L} \setminus L$ the set of *irrelevant labels*.

The set of relevant labels L can be conveniently represented in terms of a binary vector $\boldsymbol{y} = (y_1, \ldots, y_m)$, where $y_i = 1$ if the label l_i is considered to be relevant, i.e., $l_i \in L$. Alternatively, if a label l_i belongs to the set of irrelevant labels, i.e., $l_i \in \mathbb{L} \setminus L$, we set $y_i = 0$. In terms of this representation, we can denote the set of all possible label combinations by $\mathcal{Y} := \{0, 1\}^m$.

Based on this, we can define a multi-label classifier $\boldsymbol{h}$ as a function $\boldsymbol{h} : \mathcal{X} \longrightarrow \mathcal{Y}$. This function takes an instance $\boldsymbol{x} \in \mathcal{X}$ as input and returns a binary vector (*prediction*)

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

indicating the relevance of each label, where $h_i(\boldsymbol{x})$ represents the i-th entry of the vector returned by applying $\boldsymbol{h}(\cdot)$ to $\boldsymbol{x}$. Furthermore, we denote as

$$\mathcal{H} \subseteq \{\boldsymbol{h} \mid \boldsymbol{h} : \mathcal{X} \longrightarrow \mathcal{Y}\}$$

the hypothesis space specifying the available multi-label classifiers $\boldsymbol{h}$.

Provided a finite set of N observations

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

as training data, we aim for inducing such a multi-label classifier $\boldsymbol{h} : \mathcal{X} \longrightarrow \mathcal{Y}$, generalizing well beyond this finite set of observations. More specifically, we seek to find a classifier $\boldsymbol{h}^* \in \mathcal{H}$ from a hypothesis space $\mathcal{H}$ that, after fitting the training data $\mathcal{D}_{\text{train}}$, minimizes the risk with respect to a target loss function $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \longrightarrow \mathbb{R}$ (cf. Section 2.2.3)

$$\boldsymbol{h}^* \in \operatorname*{arg\,min}_{\boldsymbol{h} \in \mathcal{H}} \int_{\mathcal{X} \times \mathcal{Y}} \mathcal{L}(\boldsymbol{y}, \boldsymbol{h}(\boldsymbol{x})) P(\boldsymbol{x}, \boldsymbol{y}) \, d\boldsymbol{x} d\boldsymbol{y} \,, \tag{2.6}$$

where $P(\cdot\,, \cdot)$ refers to a(n) (unknown) joint probability distribution for $\boldsymbol{x}$ and $\boldsymbol{y}$.

2.2.2 Single-Label Classification

Standard classification tasks, in the following referred to as *single-label classification* (SLC), such as binary or multi-class classification problems, can be understood as special cases of MLC. While limiting the number of labels m to 1 yields a binary classification task, we can derive the multi-class classification problem by imposing a constraint of mutual exclusiveness on the labels. More specifically, for any set of

relevant labels $L \subseteq \mathbb{L}$, we restrict the size of L to be $|L| = 1$. Following the notation of the binary vector $\boldsymbol{y}$, this constraint can be formulated by the sum of its entries y_i being equal to 1, i.e., $\sum_{i=1}^{m} y_i = 1$.

2.2.3 Loss Functions

Over time, a wide array of loss functions has been proposed to quantify the quality of predictions, many of which generalize or adapt losses that are well-known in the literature on single-label classification (SLC). This section presents a subset of these loss functions, most commonly used in the literature. For a more comprehensive overview, we refer the reader to [WZ17].

To estimate the generalization performance of a multi-label classifier, let

$$\mathcal{D}_{\text{test}} := (X_{\text{test}}, Y_{\text{test}}) = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{S} \subset \mathcal{X}^S \times \mathcal{Y}^S \tag{2.7}$$

be a test set of size S.

For convenience, we interpret Y_{test} in the following as a matrix

$$Y = \begin{pmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \\ \dots \\ \boldsymbol{y}_S \end{pmatrix} = \begin{pmatrix} y_{1,1} & \dots & y_{1,m} \\ y_{2,1} & \dots & y_{2,m} \\ \dots & \dots & \dots \\ y_{S,1} & \dots & y_{S,m} \end{pmatrix}$$

where $y_{i,j}$ denotes the ground truth relevance of a label j for observation i.

Similarly, we can write the predictions of a multi-label classifier $\boldsymbol{h}$ on X_{test} as a matrix $\widehat{Y}$:

$$\widehat{Y} = \begin{pmatrix} \boldsymbol{h}(\boldsymbol{x}_1) \\ \boldsymbol{h}(\boldsymbol{x}_2) \\ \dots \\ \boldsymbol{h}(\boldsymbol{x}_S) \end{pmatrix} = \begin{pmatrix} h_1(\boldsymbol{x}_1) & \dots & h_m(\boldsymbol{x}_1) \\ h_1(\boldsymbol{x}_2) & \dots & h_m(\boldsymbol{x}_2) \\ \dots & \dots & \dots \\ h_1(\boldsymbol{x}_S) & \dots & h_m(\boldsymbol{x}_S) \end{pmatrix} = \begin{pmatrix} \hat{y}_{1,1} & \dots & \hat{y}_{1,m} \\ \hat{y}_{2,1} & \dots & \hat{y}_{2,m} \\ \dots & \dots & \dots \\ \hat{y}_{S,1} & \dots & \hat{y}_{S,m} \end{pmatrix} = \begin{pmatrix} \hat{\boldsymbol{y}}_1 \\ \hat{\boldsymbol{y}}_2 \\ \dots \\ \hat{\boldsymbol{y}}_S \end{pmatrix}$$

Here, an entry $\hat{y}_{i,j}$ represents the prediction of $\boldsymbol{h}$ for label j of observation i. Hence, each row represents the predictions for an observation, and vice versa, each column for a specific label. Then, an MLC loss function can be defined as $\mathcal{L} : \mathcal{Y}^S \times \mathcal{Y}^S \longrightarrow [0, 1]$.[2]

[2] More generally, MLC loss functions may take a matrix $\mathcal{Y}^S$ for the ground truth labels and a matrix $\mathbb{R}^{S \times m}$ representing the predictions of $\boldsymbol{h}$ as arguments since various multi-label classifiers produce label relevance scores ranging in $[0, 1]$ instead of sharp assignments of 0 or 1. Again other classifiers may yield unbounded relevance scores. Via a threshold τ such scores can be transformed into 1 if

The generalization or adaptation of SLC losses to the MLC setting can be classified into three categories.

macro instance-wise In macro instance-wise multi-label loss functions, the loss function is first computed for each observation (row) individually and subsequently aggregated across all observations, e.g., by taking the mean. By computing the loss for each observation first, more emphasis is put on predicting all the labels of an instance correctly.

macro label-wise Here, the SLC loss function is first computed for each label, i.e., individually for each column. The losses obtained in this manner are then aggregated, e.g., via the mean. Thus, in contrast to the macro instance-wise losses that emphasize making correct predictions for instances, in macro label-wise loss functions, the focus is on predicting labels correctly.

micro Lastly, rather than reinforcing better predictions for either instances or labels, any prediction $\hat{y}_{i,j}$ can be considered equally important. To this end, micro loss functions arrange the entries of the matrices Y respectively $\widehat{Y}$ in terms of vectors before computing the loss.

For example, the *subset 0/1* loss is a macro instance-wise generalization of the well-known error rate. It considers the predicted and expected label subsets as a whole and considers the entire prediction for an instance to be wrong whenever the two sets do not coincide:

$$\mathcal{L}_{0/1}(Y_{\text{test}}, \widehat{Y}) := \frac{1}{S} \sum_{i=1}^{S} \left[\!\left[\boldsymbol{y}_i \neq \hat{\boldsymbol{y}}_i \right]\!\right] \ , \tag{2.8}$$

where $[\![\cdot]\!]$ is the indicator function. Although it is frequently used in the literature, one could criticize it for penalizing mistakes in the prediction overly stringent. In particular, there is no difference between almost correct and completely wrong predictions since the label sets need to match completely. Consequently, observed values for the subset 0/1 loss are usually relatively high and often close to 1.

The *Hamming* loss represents another extreme as a generalization of the error rate, counting the number of entries in $\widehat{Y}$ deviating from the expected values in Y and dividing by the total number of entries.

$$\mathcal{L}_H(Y_{\text{test}}, \widehat{Y}) := \frac{1}{S} \sum_{i=1}^{S} \frac{1}{m} \sum_{j=1}^{m} \left[\!\left[y_{i,j} \neq \hat{y}_{i,j} \right]\!\right] \ . \tag{2.9}$$

the score exceeds the threshold τ and 0 otherwise. When dealing with classifiers producing scores, for the sake of simplicity, we assume these scores to be thresholded in the following.

Note that the Hamming loss does not fall into one of the three categories exclusively but can be seen as a representative of all three categories. Nevertheless, we write it here in the form of an instance-wise loss function. As the subset 0/1 loss, the Hamming loss behaves rather extreme, and its usefulness for real-world applications might appear debatable. This is because, in practice, the number of irrelevant labels often outnumbers the number of relevant labels, i.e., the label matrix is usually very sparse, which in turn results in a very low loss. Hence, even a constant classifier always predicting all labels to be irrelevant has a Hamming loss close to 0.

To find a compromise between these two extremes, a generalized family of instance-wise loss functions, building on so-called non-additive measures, can be considered to interpolate between Hamming and subset 0/1 loss [Hül+22]. For this purpose, this interpolation considers all possible subsets of labels with a certain size and evaluates the average subset 0/1 loss for all the subsets. Hamming and subset 0/1 loss can be derived as special cases of this generalized family of loss functions if all subsets of size 1 or $|\mathbb{L}|$ are considered, respectively. Obviously, the latter considers the predicted and expected label sets as a whole. The former, in turn, considers $|\mathbb{L}|$ many subsets of size 1, thus, considering each label of an instance individually and counting the errors. In between, more emphasis is put on getting subsets of labels with a specific size correctly and thereby allows for interpolating between Hamming and the subset 0/1 loss.

As already mentioned above, in practice, one can frequently observe that the irrelevant labels outnumber the relevant ones by a large margin. To address this problem of class imbalance, the F1-measure can be considered as it is defined as the harmonic mean of precision and recall. Note that the F1-measure is not a loss function but rather a measure of accuracy and thus to be maximized[3]. It has been adapted to the MLC setting in the spirit of all three categories, i.e., as an instance-wise, label-wise, and micro loss function which are defined as follows:

$$\mathrm{F1}_I(Y_{\text{test}}, \widehat{Y}) := \frac{1}{S} \sum_{i=1}^{S} \frac{2 \sum_{j=1}^{m} y_{i,j}\, \hat{y}_{i,j}}{\sum_{j=1}^{m} (y_{i,j} + \hat{y}_{i,j})} \tag{2.10}$$

$$\mathrm{F1}_L(Y_{\text{test}}, \widehat{Y}) := \frac{1}{m} \sum_{j=1}^{m} \frac{2 \sum_{i=1}^{S} y_{i,j}\, \hat{y}_{i,j}}{\sum_{i=1}^{S} (y_{i,j} + \hat{y}_{i,j})} \tag{2.11}$$

$$\mathrm{F1}_\mu(Y_{\text{test}}, \widehat{Y}) := \frac{2 \sum_{j=1}^{m} \sum_{i=1}^{S} y_{i,j}\, \hat{y}_{i,j}}{\sum_{j=1}^{m} \sum_{i=1}^{S} (y_{i,j} + \hat{y}_{i,j})} \tag{2.12}$$

While the instance-wise F1-measure accounts for the imbalance between relevant and irrelevant labels for every observation, the label-wise F1-measure considers each label column individually and accounts for imbalance in terms of the respective

[3] Obviously, since F1 ranges between 0 and 1, it can be translated into a loss function by taking $1 - \mathrm{F1}$.

label being rarely relevant or irrelevant. F1_μ does not distinguish between labels or instances and more generally emphasizes predicting the relevant labels correctly. Nevertheless, good performance in terms of the F1-measure requires both a high true positive rate, i.e., predicting relevant labels correctly, and a high true negative rate, meaning to predict irrelevant labels correctly. Thus, as opposed to $\mathcal{L}_H$, the constant "always positive" or "always negative" predictor will be assigned the worst performance.

As can already be seen from the example of the constant predictor, the perception regarding the quality of predictions depends significantly on the considered performance measure. More specifically, a classifier performing best in terms of one loss function does not necessarily perform best for another loss function [Hül+22]. Consequently, this also means, in particular, that the choice and configuration of an MLC classifier need to be tailored to the loss function in question.

2.2.4 Label Dependence

One of the major themes and, arguably, the driving force in the multi-label classification literature deals with the development of methods that can exploit so-called *label dependence* to improve the generalization performance. Label dependence refers to a stochastic correlation between labels, such as labels being positively correlated and thus more likely to occur together.

To give an intuition of what is meant by label dependence, consider again the example of Section 1.2. Arguably, the class label `BEACH` is positively correlated with the class label `SEA` since beaches are usually located by the sea. The two class labels are positively correlated. To put it differently, if the label `BEACH` is positive, i.e., a beach is on a picture, then the label `SEA` is likely to be positive as well. Hence, in this case, a multi-label classifier should be able to acknowledge this correlation and be more reluctant to predict `BEACH` without `SEA`, whereas none of the two labels, only `SEA`, and both labels simultaneously seem to be reasonable predictions.

Formally, label dependence can be distinguished into two categories: *conditional* and *marginal (unconditional)* label dependence [Dem+12]. While the former refers to a dependence between labels conditioned on a particular instance $\boldsymbol{x}$, the latter is of a more general type and can be seen as a kind of *expected* label dependence across the entire instance space.

Following the definitions of conditional and marginal label dependence by Dembczyński et al. [Dem+12], let $\mathcal{Y} = \{0, 1\}^m$ be a label space with m labels. Furthermore, let $\boldsymbol{Z} = (Z_1, \ldots, Z_m)$ denote a random vector of labels for a probability

distribution $P(\cdot, \cdot)$ on $\mathcal{X} \times \mathcal{Y}$, as in Section 2.2.1. In this context, $\boldsymbol{y} \in \mathcal{Y}$ is a realization of this random vector $\boldsymbol{Z}$.

For a given instance $\boldsymbol{x} \in \mathcal{X}$, a random vector of labels $\boldsymbol{Z}$ is called conditionally independent, if

$$P(\boldsymbol{Z} \mid \boldsymbol{x}) = \prod_{i=1}^{m} P(Z_i \mid \boldsymbol{x}) \ .$$

With $P(\boldsymbol{Z} \mid \boldsymbol{x})$ we denote the conditional probability of observing a random vector $\boldsymbol{Z}$ given the instance $\boldsymbol{x}$. Hence, if the probability of the random vector $\boldsymbol{Z}$ given an instance $\boldsymbol{x}$ is equal to the product of the probabilities of the individual labels given instance $\boldsymbol{x}$, the labels are stochastically independent. Vice versa, this means that if this equality does not hold, we observe a conditional label dependence.

Marginal (unconditional) label dependence can be defined similarly. A random vector is called *marginally independent*, if

$$P(\boldsymbol{Z}) = \prod_{i=1}^{m} P(Z_i) \ .$$

Note that for marginal label dependence the probability is *not* conditioned on any instance $\boldsymbol{x}$. Likewise, we observe a marginal label dependence if the equation does not hold.

Furthermore, the two definitions are closely related to each other. As already briefly mentioned before, marginal label dependence can be seen as a kind of *expected* label dependence. This is because we obtain the definition of marginal label dependence by averaging conditional label dependence over the instance space $\mathcal{X}$:

$$P(\boldsymbol{Z}) = \int_{\mathcal{X}} P(\boldsymbol{Z} \mid \boldsymbol{x}) \, d\mu(\boldsymbol{x}) \ ,$$

where $\mu(\cdot)$ denotes the probability distribution on $\mathcal{X}$ according to the joint probability distribution $P(\cdot, \cdot)$. However, despite this relation, Dembczyński et al. [Dem+12] prove that neither dependence implies the other.

While multi-label classifiers can be more or less suitable for either type and may exploit this information, it is not necessarily important to do so for risk minimization. Depending on the target loss function, an optimal prediction might or might not require taking label dependence into account. For example, if the loss function is label-wise decomposable, i.e., the loss is first computed over all observations for each label individually and then aggregated, from a theoretical perspective, there is no need to consider label dependencies [Dem+12]. Instead, it suffices to know (learn) the marginal distributions for the corresponding labels in order to make a risk-minimizing prediction. Examples of such loss functions are the Hamming

loss and the label-wise F1-measure. Conversely, exploiting label dependence may indeed be advantageous in terms of generalization performance or even necessary to produce risk-minimizing predictions, especially when considering instance-wise loss functions such as the subset 0/1 loss.

To illustrate the impact of the target loss function on the optimality of a prediction, we consider a simple example, based on our running example of Section 1.2. Let $\mathbb{L} = \{\texttt{BEACH}, \texttt{FOREST}, \texttt{MOUNTAIN}, \texttt{SEA}\}$ denote the label space of the $m = 4$ labels and $\mathcal{Y} = \{0, 1\}^4$ correspondingly. Furthermore, given an observation $\boldsymbol{x}$, let the (conditional) ground-truth distribution on $\mathcal{Y}$ be as follows:

BEACH	FOREST	MOUNTAIN	SEA	$P(\boldsymbol{y} \mid \boldsymbol{x})$
0	0	0	0	$3/12$
0	1	1	1	$1/12$
1	0	1	1	$2/12$
1	1	0	1	$2/12$
1	1	1	0	$2/12$
1	1	1	1	$2/12$

Any other label combination is assigned a probability of 0 and the label dependence is captured via $P(\boldsymbol{y}|\boldsymbol{x})$. Note that label dependence is encoded here in terms of the probabilities of the respective label sets, i.e., given instance $\boldsymbol{x}$, the most probable label set is $(0, 0, 0, 0)$ with a ground truth probability of $3/12$. However, a 1 is observed with a ground truth probability of $8/12$ for BEACH and $7/12$ for all the remaining labels.

In this example, one can easily see that the Bayes-optimal prediction, which minimizes the loss in expectation, for subset 0/1 loss is $\boldsymbol{h}(\boldsymbol{x}) = (0, 0, 0, 0)$, i.e., the set of relevant labels is empty ($L_{\boldsymbol{x}} = \emptyset$). However, the Bayes-optimal prediction with respect to the Hamming loss is obtained through $\boldsymbol{h}(\boldsymbol{x}) = (1, 1, 1, 1)$, i.e., all class labels $L_{\boldsymbol{x}} = \{\texttt{BEACH}, \texttt{FOREST}, \texttt{MOUNTAIN}, \texttt{SEA}\}$ are relevant. Hence, for minimizing the risk in terms of the subset 0/1 loss, it is crucial to consider the joint mode of the distribution and thus the label set $(0, 0, 0, 0)$ as a whole. On the contrary, for an optimal prediction with respect to the Hamming loss, we examine each label individually and thereby ignore the relevance of other labels. In extreme cases, as in this example, the optimal predictions can even be orthogonal to each other.

As demonstrated in this small example, optimal predictions are not only dependent on the properties of the data, such as the presence of label dependence, but also on the loss function to be minimized. In particular, this also makes it clear that an MLC algorithm needs to be tailored to both the data – and therewith the type of label dependence – and the target loss function.

2.2.5 Multi-Label Classifiers

Based on the plethora of well-studied SLC algorithms, multi-label classification algorithms have been developed by either *transforming* the original MLC problem into a (set of) single-label classification problem(s) and applying SLC algorithms to the resulting problems or by adapting existing algorithms to the specifics of the MLC problem [TK07]. More precisely, the algorithms need to be adapted to predict label sets instead of single labels. In the following, we give a brief overview of both algorithm adaptation methods and problem transformation methods.

Problem transformation algorithms compile the given MLC problem into one or several SLC problems. Formally, they perform a reduction from the original problem to SLC such that the resulting problems can be dealt with by already known methods, e.g., decision trees, SVMs, or logistic regression. The most straightforward problem transformation algorithms arguably are the label power set (LP) classifier [Bou+04; Dip+05] and binary relevance (BR) learning [Zha+18].

LP casts the MLC problem as a single multi-class classification problem, treating each label combination as a distinct class. Although being conceptually simple, the approach comes with a few technical issues and limitations. First, the number of classes grows exponentially in the number of labels m, and thus, quickly reaches dimensions that render the learning problem unfeasible. Second, label combinations may be considered as classes that do not and will never actually occur in the data. Third, explicit information about structures over and dependencies between labels is lost since each label combination is represented by a simple class label. Hence, LP can learn about and account for label dependencies implicitly but not exploit such properties directly.

BR handles the problem transformation quite differently by performing a decomposition into a binary classification problem for each label. Each binary classification problem aims to predict whether the respective label is relevant or not. Again, due to the implicit independence assumption made about the labels, i.e., that the relevance of each label can be predicted independently of the other labels, interactions or dependencies between labels are ignored in BR as well. In fact, BR is often criticized for this independence assumption, but for label-wise performance measures, such as Hamming loss or the label-wise F1-measure, it can be theoretically shown that BR is indeed optimal [Dem+12; Lua+12]. Also, the flexibility resulting from the independence assumption can be leveraged to select and configure the SLC methods for tackling the induced binary classification tasks for each label individually. As shown in Chapter 7, this individualization can be done efficiently, and it benefits the generalization performance. Nonetheless, the idea of exploiting such label depen-

dencies in order to improve generalization performance is the primary motivation for various methods developed to tackle the MLC problem.

Taking BR as a point of departure, various more sophisticated methods have been developed over time [Zha+18; Rea+21]. A more sophisticated, very powerful, and probably the most prominent problem transformation technique is called *classifier chains* [Rea+11; Rea+21]. In classifier chains, a chain of binary classifiers is trained in a sequence such that each classifier predicts the relevance for a specific label but taking into account the relevance of the labels handled by previous classifiers. While the relevance of the previous labels is given in the data when training a classifier chain, this information is not available during prediction. Therefore, the values for the attributes are provided using the predictions of the previous classifiers. However, a preceding classifier's prediction about the relevance of the respective label might be wrong. In this case, the error propagates through the chain and affects subsequent predictions, resulting in a kind of attribute noise [SCH12]. Another issue is that the order of labels within a classifier chain plays an essential role in the generalization performance and has been the subject of optimization in various studies [Rea+21]. To overcome this issue, ensembles of classifier chains can be built to compensate for sub-optimal orders of labels [Rea+11].

Returning to the image tagging example (cf. Section 1.2), for instance, a classifier chain may first predict the presence of `BEACH` based on the properties of the picture. The prediction for `SEA` could then be conditioned on the properties of the picture *and* the (predicted) presence or absence of the class label `BEACH`. In this way, classifier chains may capture a potential dependence between class labels, at least to some extent. However, the ground truth label for `BEACH` is only available during training, and during prediction time, it is replaced by a prediction for that label. Hence, if the prediction for `BEACH` is wrong, the error propagates to the prediction of the class label `SEA`.

Algorithm adaptation is the other prominent way of developing MLC algorithms, taking SLC methods as a point of departure and adapting models and/or learning algorithms to the MLC setting. To give a brief impression of algorithm adaptation methods, we present a few such methods in the following. A more detailed overview can be found in [Bog+21]. For example, neural networks can be adapted to multi-label classification quite easily by adding an output neuron for each label and integrating the training algorithm with an MLC loss function [Zha09]. Moreover, decision trees have been transferred to the MLC setting by adapting the split criterion [CK01] or employing predictive clustering [Koc+07]. A more recent approach, called BOOMER [Rap+20], is a multi-label classification rule learner, leveraging boosting techniques to achieve state-of-the-art generalization performance while providing interpretable models.

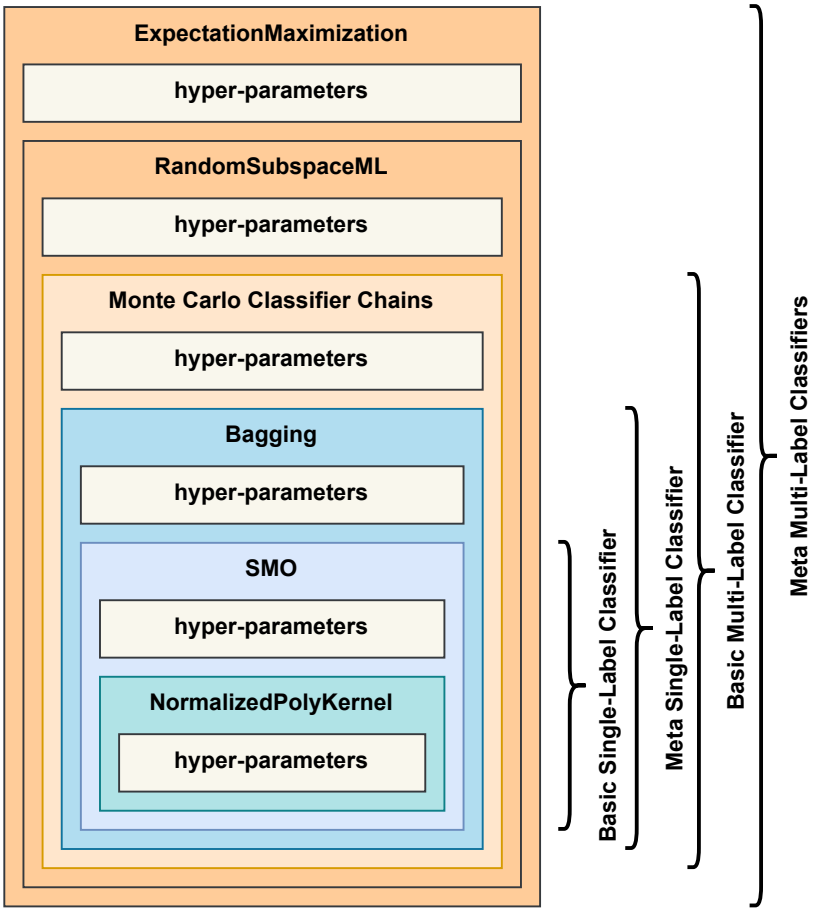


Figure 2.9: Exemplary illustration of a problem transformation multi-label classifier.

2.2.6 Configuration of Multi-Label Classifiers

To achieve the best possible generalization performance, the hyper-parameters of multi-label classifiers need to be tailored to the given data and the loss function in question, as we have already detailed above. Depending on whether the classifier was designed via algorithm adaptation or a problem transformation, optimizing the hyper-parameters is differently challenging. While in the former case, only the hyper-parameters directly exposed by the respective multi-label classifier are subject to optimization, in the case of problem transformation, the optimization of hyper-parameters is usually more complex. Compiling the original MLC problem into a (set of) SLC problem(s), problem transformation methods can be seen as meta-learning methods which are configured with a base learner, i.e., an SLC method, to tackle the resulting SLC problems. On the one hand, this degree of freedom allows a better fit of the algorithm to the given data and loss function. On the other hand, for configuring such multi-label classifiers, a decision needs to be made regarding the choice of the base learner and the hyper-parameters that the chosen base learner may additionally expose.

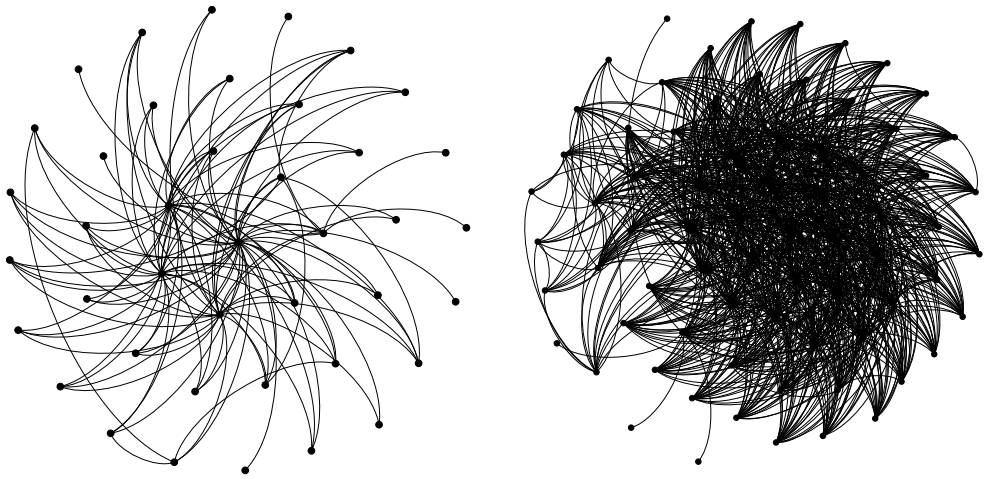


Figure 2.10: Illustration of the search spaces of ML-Plan for MLC [Wev+19] only considering the choice of algorithms for single-label classification (**left**) and multi-label classification (**right**) as directed acyclic graphs. The curvature of an edge in a clock-wise direction represents a directed edge from a parent to a child node. While each node represents an algorithm contained in the search space, the interpretation of an edge is that the algorithm represented by the parent node has a hyper-parameter that can be configured with the algorithm represented by the child node.

In Figure 2.9, the structure of an exemplary multi-label classification method is illustrated. The figure displays a multi-label classifier consisting of 5 layers of learning algorithms and a kernel algorithm for the support vector machine (SMO, short for Sequential Minimal Optimization). The outer two layers comprise meta-algorithms for multi-label classification named `ExpectationMaximization` and `RandomSubspaceML`. The actual multi-label classifier that is configured as the base learner for the latter meta multi-label classifier is set to be `Monte Carlo Classifier Chains`, which are in turn configured with a single-label classifier as a base learner. In this example, Bagging is chosen as a meta single-label classifier, which uses `SMO` as a base learner. Finally, the SMO algorithm is configured with `NormalizedPolyKernel`. While in this example, the decisions regarding the respective algorithms and base learners are already made, each of the algorithms exposes hyper-parameters that need to be tuned.

As can be seen from this single example, the configuration of multi-label classifiers is quite complex, and many decisions need to be made. In Chapter 6, we deal with the configuration of multi-label classifiers considering a pool of roughly 70 algorithms,

exposing a total of more than 170 hyper-parameters (without counting the choice of base learners), whereas a complete multi-label classifier may expose up to 25 hyper-parameters to be configured. To give an intuition about how much the search space, limited only to the discrete (recursive) choice of algorithms, grows for multi-label classification as compared to the search space for single-label classification, Figure 2.10 shows the respective search spaces as directed acyclic graphs (DAGs). Nodes of the DAG represent algorithms, and edges indicate a relation between two algorithms in the sense that the child node can be configured as the base algorithm of the parent node. The direction of an edge can be read from its curvature. When following an edge in a clockwise direction, one traverses the DAG from a parent node to a child node and vice versa.

While the graph on the left-hand side still shows a clear structure and seems quite manageable, the graph on the right side is much denser. Algorithms (represented by nodes) and relationships between those algorithms, represented by edges, are difficult to recognize and can only be glimpsed. More precisely, the search space not only contains more algorithms in total, i.e., 70 compared to 30, but also offers more possibilities to combine algorithms and configure them as base algorithms. Clearly, selecting and configuring a multi-label classifier is a demanding and cumbersome task, even for experts. This is all the more true for non-experts.

Going beyond existing hyper-parameters and the configuration of base learners, one may, of course, also think of adding more hyper-parameters to multi-label classifiers, thereby increasing the degrees of freedom and improving their generalization performance. For example, in [Nam+19] the permutation of base learners in classifier chains is found to have a practical impact on the performance. Improving the generalization performance by optimizing this permutation has been the subject of various studies [Rea+21]. Moreover, rather than committing to a single base algorithm used for all the labels, a base algorithm could be specifically selected for each label. For the case of BR, in Chapter 7, we demonstrate that a label-wise configuration of base algorithms for BR may indeed prove beneficial.

While this flexibility can positively affect performance, it obviously also increases the complexity of the configuration space by several orders of magnitude. Consequently, the configuration process becomes even more complex and increases the need for an automated solution to this problem.

3

ML-Plan: Automated Machine Learning via Hierarchical Planning

Declaration of the specific contributions of the author The implementation of ML-Plan was initially done by Felix Mohr, and later on, refined by Felix Mohr and the author. The experiments for the WEKA backend and the scikit-learn backend have been conducted by Felix Mohr and the author, respectively. Furthermore, the author described and summarized the results of the experiments in the paper, whereas Felix Mohr wrote the remaining sections of the publications. The entire paper was revised repeatedly by all the authors.

Machine Learning (2018) 107:1495–1515
https://doi.org/10.1007/s10994-018-5735-z

ML-Plan: Automated machine learning via hierarchical planning

Felix Mohr[1] · **Marcel Wever**[1] · **Eyke Hüllermeier**[1]

Received: 10 December 2017 / Accepted: 18 June 2018 / Published online: 3 July 2018

Abstract
Automated machine learning (AutoML) seeks to automatically select, compose, and parametrize machine learning algorithms, so as to achieve optimal performance on a given task (dataset). Although current approaches to AutoML have already produced impressive results, the field is still far from mature, and new techniques are still being developed. In this paper, we present ML-Plan, a new approach to AutoML based on hierarchical planning. To highlight the potential of this approach, we compare ML-Plan to the state-of-the-art frameworks Auto-WEKA, auto-sklearn, and TPOT. In an extensive series of experiments, we show that ML-Plan is highly competitive and often outperforms existing approaches.

Keywords Automated machine learning · Automated planning · Algorithm selection · Algorithm configuration · Heuristic search

1 Introduction

The demand for machine learning (ML) functionality is growing quite rapidly, and successful machine learning applications can be found in more and more sectors of science, technology, and society. Since end users in application domains are normally not machine learning experts, there is an urgent need for suitable support in terms of tools that are easy to use. Ideally, the induction of models from data, including the data preprocessing, the choice of a model class, the training and evaluation of a predictor, the representation and interpretation of results, etc., would be automated to a large extent (Lloyd et al. 2014). This has triggered the field of *automated machine learning* (AutoML).

Editors: Jesse Davis, Elisa Fromont, Derek Greene, and Bjorn Bringmaan..

Eyke Hüllermeier
eyke@uni-paderborn.de

Felix Mohr
felix.mohr@uni-paderborn.de

Marcel Wever
marcel.wever@uni-paderborn.de

[1] Paderborn University, Warburger Str. 100, 33098 Paderborn, Germany

State-of-the-art AutoML tools have shown impressive results (Thornton et al. 2013; Komer et al. 2014; Feurer et al. 2015) but still leave room for improvement. Most of those approaches squeeze the AutoML problem into the rigid corset of a (Bayesian) optimization problem with a fixed number of decision variables. Typically, there is one variable for the preprocessing algorithm, one variable for the learning algorithm, and one variable for each parameter of each algorithm. While this way of formalizing the AutoML problem leads to a solution space of fixed dimensionality, it comes with a significant loss of structural information for the search. TPOT (Olson and Moore 2016) and RECIPE (de Sá et al. 2017) allow for configuring ML pipelines in a more flexible manner, using evolutionary algorithms for optimizing structure and parameters, but suffer from scalability issues.

In this paper, we present ML-Plan, a new approach for AutoML based on hierarchical task networks (HTNs). HTN is an established AI planning technique (Erol et al. 1994; Nau et al. 2003) usually implemented as a heuristic best-first search over the graph induced by the planning problem. There have been earlier HTN-based approaches to configure data mining pipelines by Kietz et al. (2012) and Nguyen et al. (2014). However, the optimization potential of these techniques is rather limited. In fact, Kietz et al. (2012) ranks candidates based on usage frequencies of RapidMiner, and Nguyen et al. (2014) adopts a hill-climbing approach guided by a database of known problems. In contrast, ML-Plan guides the search by randomly completing partial pipelines like the ones in Auto-WEKA and auto-sklearn. This way, ML-Plan offers a middle-ground solution that combines ideas and concepts from different approaches, notably the evaluation of candidate pipelines at runtime, as done by Thornton et al. (2013), Feurer et al. (2015), and the idea of Nguyen et al. (2014) to use HTN for pipeline construction.

To the best of our knowledge, ML-Plan is the first AutoML approach that includes a dedicated mechanism to prevent over-fitting. While previous approaches did recognize the problem, too, specific remedies have not been offered. We propose a two-phase search model (search + select) and show the benefit of this technique in terms of reduced error rates.

Our experimental evaluation shows that ML-Plan is highly competitive and often outperforms the above state-of-the-art tools. Focusing on the search space of pipelines with fixed length as in Thornton et al. (2013), Feurer et al. (2015), we mainly compare our search technique against the techniques used by Auto-WEKA and auto-sklearn. ML-Plan can be run with algorithms from both libraries WEKA and scikit-learn. To minimize library-rooted confounding factors, we evaluate the WEKA version of ML-Plan against Auto-WEKA and its scikit-learn-version against auto-sklearn, where the available algorithms are respectively the same. TPOT was included as an additional baseline, although it has a different search space and allows for more complex pipelines.

2 Problem definition

AutoML seeks to automatically *compose* and *parametrize* machine learning algorithms to maximize a given metric such as predictive accuracy. The available algorithms are typically related either to preprocessing (feature selection, transformation, imputation, etc.) or to the core functionality (classification, regression, ranking, etc.). While there are successful approaches for the flexible composition of these algorithms such as TPOT (Olson and Moore 2016), most approaches arrange the algorithms sequentially and adopt a fixed template for these pipelines. For example, auto-sklearn optimizes the pipeline shown in Fig. 1 with exactly three elements, and the Auto-WEKA pipeline has only two such pipeline items. In this paper, we consider the problem of a 2-step pipeline consisting of a preprocessor and a classifier.

Fig. 1 A classical AutoML pipeline of fixed length

The planning technique we present in this paper is not limited to a particular type of learning problem. For simplicity and ease of exposition, we nevertheless focus on multi-class classification. Thus, we subsequently assume that the core machine learning algorithms are classification algorithms. Depending on the learning problem, the adaptation to other settings is either straightforward (e.g., for regression or ranking) or may require some changes in the routine (e.g., in structured output prediction, where not all preprocessing combinations may be allowed, or in unsupervised learning). However, even in those settings, no change of the actual search mechanism is required.

Formally, the goal is to find a machine learning pipeline that learns to associate output elements from a space Y (in our case classes) with input objects from an instance space X. Often, $X = \mathbb{R}^d$ for some integer d, i.e., instances are described in terms of d numerical attributes (features); however, features may also be binary or categorical. We denote by $\mathcal{X}$ the collections of all such input spaces X. A dataset is a (finite) subset $D = \{(x_i, y_i)\}_{i=1}^n \subset X \times Y$.

In this context, we may apply two types of algorithms. First, *preprocessors* are functions ϕ that map datasets D to datasets D', possibly changing the representation space from $X \times Y$ to $X' \times Y'$. Examples of such functions include methods for dimensionality reduction (such as principal component analysis), feature selection, imputation, discretization, normalization, etc. Second, *learners* are functions that map datasets D to a predictor function $\psi\colon X \to Y$.

A *pipeline* is the pair consisting of a parametrized preprocessor and a learner. Both types of algorithms can have continuous, discrete, ordinal, or nominal hyperparameters.[1] Let $\mathcal{A}_{pp}$ and $\mathcal{A}_{learn}$ be the space of *parametrized* preprocessing and learning algorithms, respectively. A pipeline is a pair $C \in \mathcal{A}_{pp} \times \mathcal{A}_{learn}$, where C itself is a learner.

Given a set of labeled data D, the task consists of combining the above algorithms into a pipeline C that, taking training data D as an input, produces an optimal predictor $\psi = C(D)$ as output. Here, optimality normally refers to the generalization performance, i.e., the expected loss caused by ψ when being used for predicting class labels on new data (not contained in the training data, but being produced by the same data-generating process). More formally, the goal is to find

$$C^* \in \operatorname*{argmin}_{C \in \mathcal{A}_{pp} \times \mathcal{A}_{learn}} \mathcal{R}\big(C(D)\big), \tag{1}$$

with the *risk* or expected loss of the predictor ψ given by

$$\mathcal{R}(\psi) = \int_{X \times Y} loss(y, \psi(x))\, dP(x, y). \tag{2}$$

Here, $loss(y, \psi(x)) \in \mathbb{R}$ is the penalty for predicting $\psi(x)$ for instance $x \in X$ when the true label is $y \in Y$, and P is a joint probability measure on $X \times Y$.

[1] The term "hyperparameter" is commonly used for parameters of the learning algorithm, to distinguish them from the parameters of the learned predictor ψ.

Since (2) cannot be evaluated (the data-generating process P is assumed to exist but is obviously not known to the learner), we replace the true generalization performance $\mathcal{R}(\psi)$ by an estimation $\hat{\mathcal{R}}(\psi)$. The latter is obtained by evaluating ψ on *validation data* D_{val} not used for training:

$$\hat{\mathcal{R}}(\psi) = \mathbb{E}\left[\frac{1}{|D_{val}|}\sum_{(x,y)\in D_{val}} loss\big(y, \psi(x)\big)\right],$$

where the expectation is taken with respect to the randomly chosen validation data D_{val} of predefined size $k = |D_{val}|$, i.e., with respect to all random splits of the data D into D_{train} and $D_{val} = D \backslash D_{train}$. Thus, we eventually solve the problem (1) with $\hat{\mathcal{R}}$ as a surrogate of $\mathcal{R}$.

Since computing the optimal solution is usually infeasible, we are interested in a solution that is as good as possible under given resource constraints. As usual, we consider limited runtime (1 h and 1 day) and hardware resources (CPU and memory).

3 Related work

Auto-WEKA (Thornton et al. 2013; Kotthoff et al. 2017) and auto-sklearn Feurer et al. (2015) are the main representatives for solving AutoML by so-called sequential parameter optimization. Both apply the general purpose algorithm configuration framework SMAC (Hutter et al. 2011) to find optimal machine learning pipelines. In order to fit the AutoML problem into the problem class of algorithm configuration and enable the application of SMAC, the ML algorithms that can be used in the pipeline are interpreted as parameters (of an imaginary pipeline algorithm) themselves. The parameters of the ML algorithms are considered by SMAC only in case the corresponding algorithms have been chosen (activated). As in ML-Plan, candidate solutions are *executed* and tested against a test set during search in order to estimate their quality.

Compared to Auto-WEKA, auto-sklearn introduces two main innovations. The first is a so-called "warm-start" technique that uses meta-features of the datasets to determine good candidates to be considered in the pipeline based on past experiences on similar datasets. Second, auto-sklearn can be configured to return an ensemble of classifiers instead of a single classifier.

The main difference between the above approaches and ML-Plan is that ML-Plan successively *creates* solutions in a global search instead of *changing* given solutions in a local search as done by Auto-WEKA and auto-sklearn. To organize this search space, ML-Plan uses hierarchical planning, a particular form of AI planning described in more detail in Sect. 4. ML-Plan can be configured with arbitrary machine learning algorithms written in Java or Python. In this paper, we consider a WEKA version and a scikit-learn version of ML-Plan that use the same algorithms as Auto-WEKA and auto-sklearn, respectively. The search space only deviates in the algorithm parameters, since ML-Plan adopts a discretization technique; this is discussed in Sect. 5.1.

Another interesting line of research is the application of evolutionary algorithms. One of these approaches is TPOT (Olson and Moore 2016). In contrast to the above approaches and ML-Plan, TPOT allows not just one pre-processing step but an arbitrary number of feature extraction techniques at the same time. While multiple pre-processors can be handled by HTN planning as well, the current implementation of ML-Plan does not exploit that opportunity. TPOT adopts a genetic algorithm to find good pipelines, and adopts the scikit framework to

evaluate candidates. Another approach is RECIPE (de Sá et al. 2017), which uses a grammar-based evolutionary approach to evolve pipeline construction. Like in other applications, evolutionary algorithms are not uncritical with regard to runtime. In fact, RECIPE has so far only been evaluated on rather small datasets, and our evaluation shows that TPOT is not able to return any solution for the more difficult problems even within 1 day. Of course, this neither excludes the usefulness of such approaches, especially since their results are often very good, nor the possibility to improve efficiency in one way or the other (e.g., using surrogate functions to speed up the evaluation of candidate solutions).

While AI planning has not yet been used in the core AutoML community, we are not the first to use AI planning for machine learning purposes. A first approach for the configuration of RapidMiner modules based on HTN planning was presented by Kietz et al. (2009, 2012). The search algorithm is guided by a ranking that is obtained from usage frequencies of human users of the RapidMiner tool. Nguyen et al. (2011, 2012, 2014) proposed the use of HTN planning for data mining in a tool called Meta-Miner. Similar to auto-sklearn, their focus is on learning the suitability of (partial) workflows for a dataset based on past experiences.

There are two main differences between ML-Plan and Meta-Miner. First, instead of evaluating candidates during search, they apply a hill climbing search strategy where the decisions are made based on past experiences. That is, the dataset of the active query is compared to others examined in the past, for which the performance of the candidate workflows is known, and based on this knowledge, the (partial) workflows are selected. This makes Meta-Miner very fast at the cost of not having any true estimate of the returned solution. Second, there is rather little emphasis on parameter tuning. In fact, Nguyen et al. experiment with single parameters, but in the form of different "versions" of an algorithm rather than considering the parameters as part of the HTN model. Due to the combinatorial explosion, of course, only a small subset of the parameters covered by other AutoML approaches (including ML-Plan) can be considered. In spite of these differences, their studies are of predominant importance for the further development of ML-Plan, in which we aim at a stronger incorporation of previous knowledge. In this sense, we consider the approaches as complementary.

4 Planning with hierarchical task networks (HTN)

The basis of HTN planning (Ghallab et al. 2004) is a logic language $\mathcal{L}$ and planning operators that are defined in terms of $\mathcal{L}$. The language $\mathcal{L}$ has function-free first-order logic capacities, i.e., it defines an infinite set of variable names, constant names, predicate names, and quantifiers and connectors to build formulas. An *operator* is a tuple $\langle name_o, pre_o, post_o \rangle$, where $name_o$ is a name and pre_o and $post_o$ are formulas from $\mathcal{L}$ that constitute preconditions and postconditions, respectively. For example, an operator *PCA* may conduct a principal component analysis on a given dataset; pre_o would specify the conditions under which the operator is applicable, and $post_o$ the effect it achieves.

A plan is a sequence of ground operations. As usual, we use the term *ground* to say that all variables have been replaced by terms that only consist of constants. That is, an operation is ground if all variables in the precondition and postcondition have been substituted by terms from $\mathcal{L}$ that only contain constants. Ground operators are also called *actions*; we write pre_a and $post_a$ for its precondition and postcondition, respectively.

The semantic of an action is that it modifies the state in which it is applied (e.g., turning numeric attributes into discrete ones). A *state* is a set of ground positive literals. Working under the closed world assumption, we assume that every ground literal not explicitly contained in

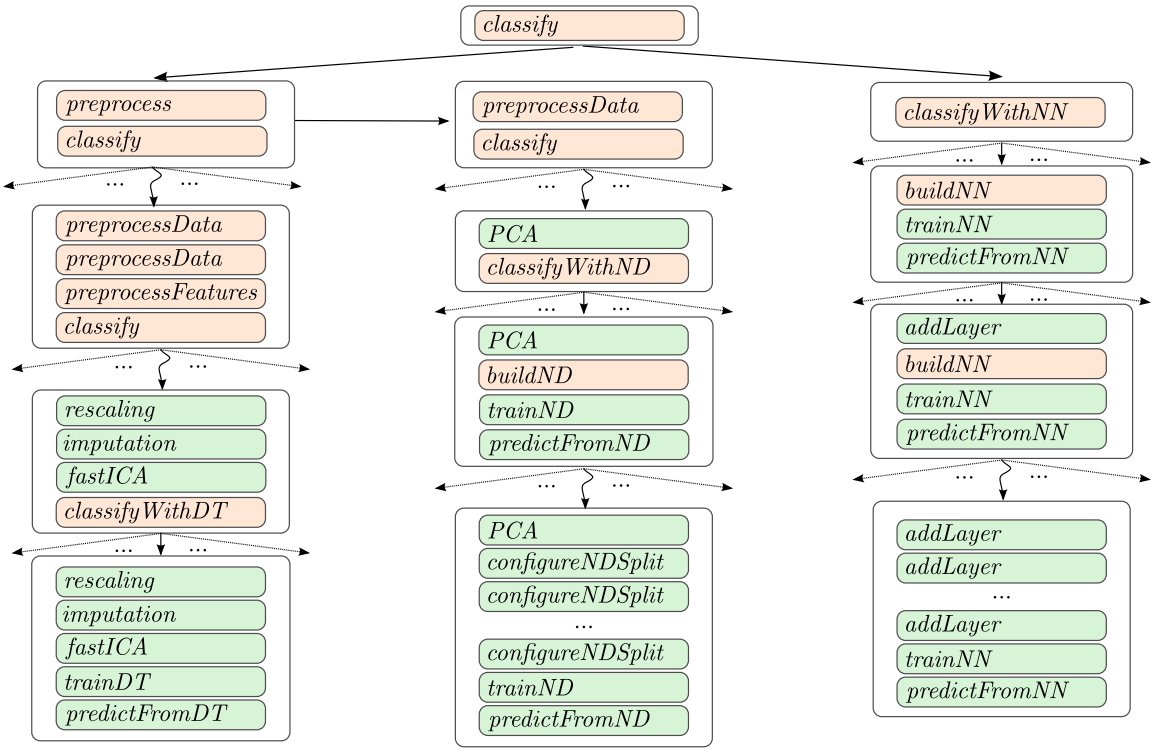


Fig. 2 Creation of pipelines with hierarchical planning. Left: a pipeline that uses a decision tree for prediction where data are preprocessed with rescaling and imputation and features are preprocessed using fast ICA. Middle: a pipeline that uses a (configured) nested dichotomy for prediction where data are preprocessed using PCA. Right: a pipeline that uses a (configured) neural network for prediction and without preprocessing

a state is false. An action a is *applicable* in state s iff $s \models_{cwa} pre_a$. The *successor state* s' induced by this application is s if a is not applicable in s and $(s \cup add) \backslash del$ otherwise; here, add and del contain all the positive and negative literals, respectively.

A hierarchical task network (HTN) is a partially ordered set T of tasks. A task $t(v_0, .., v_n)$ is a name with a list of parameters, which are variables or constants from $\mathcal{L}$. For example, $configureC45(c)$ could be the task of creating a set of options for a decision tree and assigning them to the decision tree c. A task named by an operator [e.g., $setC45Options(c, o)$] is called *primitive*, otherwise it is *complex*. A task whose parameters are constants is ground.

We are interested in deriving a plan from a task network. Intuitively, we can refine (and ground) complex tasks iteratively until we reach a task network that has only ground primitive tasks, i.e., a set of partially ordered actions. While primitive tasks can be realized canonically by a single operation, complex tasks need to be decomposed by *methods*. A method $m = \langle name_m, task_m, pre_m, T_m \rangle$ consists of its name, the (non-primitive) task $task_m$ it refines, a logic precondition $pre_m \in \mathcal{L}$, and a task network T_m that realizes the decomposition. Replacing complex tasks by the network of the methods we use to decompose them, we iteratively *derive* new task networks until we obtain one with ground primitive tasks (actions) only.

To get an intuition of this idea, consider the (totally ordered) task networks in the boxes of Fig. 2 as an example. The colored entries are the tasks of the respective networks. Orange tasks are complex (need refinement), and green ones are primitive. The tree shows an excerpt of the possible refinements for each task network. The idea is very similar to derivations in context-free grammars where primitive tasks are terminals and complex tasks are non-terminal symbols. The main difference is that HTN considers the concept of a state, which is modified by the primitive tasks and poses additional constraints on the possible refinements.

The definition of a simple task network planning problem is then straight-forward. Given an initial state s_0 and a task network T_0, the planning problem is to derive a plan from T_0 that is applicable in s_0. A simple task network planning problem is a quadruple $\langle s_0, T_0, O, M \rangle$, where O and M are finite sets of operators and methods, respectively.

HTN problems are typically solved by a reduction to a standard graph search problem that can be approached with algorithms such as depth-first search, best-first search, etc. A typical translation of the HTN problem into a graph is to select the *first* complex task in the network of a node and to define one successor for each ground method that can be used to resolve the task; this is called *forward-decomposition* (Ghallab et al. 2004). Every node in the resulting graph corresponds to a plan prefix (the part of the plan that has been fixed) together with remaining tasks. The root node has an empty plan with the initial task network, and the goal nodes have solution plans and empty rest networks. The graph in Fig. 2 sketches (an excerpt of) such a search graph for the AutoML problem. The root node corresponds to the pipeline with the initial complex task, and goal nodes are nodes that have fully defined pipelines. Usually, there is a one-to-one correspondence between search space elements, e.g., the machine learning pipelines, and the goal nodes.

5 ML-Plan

ML-Plan reduces AutoML to a graph search problem via HTN planning. More specifically, ML-Plan invokes a best-first search algorithm on the graph induced by a forward decomposition (see above) of "the" HTN planning problem. This is exactly what standard HTN solvers like SHOP2 (Nau et al. 2003) are doing, but those solvers require that the costs of a solution (plan) decompose over its actions, and that these costs are known in advance. ML-Plan overcomes this limitation and, hence, constitutes an HTN planner tailored for the needs of AutoML.

5.1 AutoML through HTN planning

ML-Plan encodes an HTN problem that divides the AutoML problem defined in Sect. 2 into an algorithm selection and an algorithm configuration phase. ML-Plan is initialized with a fixed set of preprocessing algorithms, classification algorithms, and the respective parameters and their domains. The first phase is to decide the feature preprocessing algorithm (if any) and then the classification algorithm. Inversely, the second phase first configures the classification algorithm and then the preprocessing algorithm (if any).

Note that these phases must not be understood as phases of the algorithm in the sense that ML-Plan first chooses the algorithms and then configures them, but as phases (regions) of the search graph. More precisely, our formulation of the HTN problem induces an upper and a lower part of the search graph—this is what we mean by phases. ML-Plan adopts a global best-first search within that graph and, hence, does not greedily pick algorithms and then configure them. Given sufficient time, ML-Plan will detect *all* solutions.

In the following, we describe the HTN problem encoded by ML-Plan. The complete problem description is very technical, so we focus on giving an intuition. In particular, we omit the variables of the tasks and methods to maintain readability. The full formal specification is available with our implementation.[2]

[2] Attached as supplementary material during review phase.

The initial task network consists of the tasks `choosePP`, `setupClassifier`, `configPP`. The first task can be refined to an empty task network to omit preprocessing or any of the available preprocessing algorithms. That is, for m preprocessing algorithms, there are $m + 1$ methods to resolve `choosePP`. The task networks associated with each of these methods consists of a single primitive task that adds the chosen algorithm to the state using a special predicate `chosenPP`. For example, it adds `chosenPP(PCA)` to indicate that the chosen preprocessor is the principal component analysis. Storing this decision is necessary to make sure that the correct preprocessor is refined when resolving the `configPP` task.

The second task `setupClassifier` is meant to choose and configure any of the available classifiers. Similar to Auto-WEKA, ML-Plan assumes that classifiers belong to predefined algorithm groups, e.g., basic learners, meta learners, ensembles, etc. To use this information for the organization of the search graph, ML-Plan generates one method for each of these algorithm groups, each of which has a task network with exactly one task that means to setup a classifier of that group. For example, for the group of meta learners, there is a method that refines `setupClassifier` to `setupMetaClassifier`. There is exactly one method for each classifier, and it can be used for the task of the respective algorithm group. Each of these methods refines the task to a network of the form `selectClassifier`,`setupParam1`, …, `setupParamN` for all of the N parameters of the respective classification algorithm, so the network enforces that a decision is made for all parameters. The `selectClassifier` task is primitive and only adds the choice of the classifier to the state. For each of the parameters, there are methods that induce primitive tasks either setting or not setting the respective parameter, i.e., leaving it at the default value. The base learners of ensemble classifiers such as voting are considered as parameters in this model.

The same technique is then applied to refine the third task `refinePP`. Refining this task means to configure the initially chosen preprocessor (if any) and completes the configuration.

Note that the reduction conducted by ML-Plan is not a canonical one. In fact, there are many different HTN problems that can cover exactly the same search space. So apart from any questions related to heuristics, node evaluation, etc., the mere way of how the HTN problem is *formulated* can have a tremendous impact on the search efficiency. For example, besides the above technique, we could also use a two-step network where we first choose and configure the preprocessor (or choose to not use any) and then choose and configure the classifier. While this looks like a trivial alteration that does not influence the set of constructible pipelines, it has important consequences on the structure of the search tree.

In the current implementation, ML-Plan chooses the parameter values from a small predefined set of possible values. To this end, numerical parameters are discretized either on a linear scale or a log scale. The interval and discretization technique for a parameter is not a choice point but is fixed in advance.

While there is certainly room for improvement in this technique, this simple discretization seems to be often sufficient. Indeed, discretization is less flexible than the native support for numeric variables offered, say, by Bayesian optimization as used in Auto-WEKA (Thornton et al. 2013) and auto-sklearn (Feurer et al. 2015), and the decision about how a parameter should be discretized may seem arbitrary and should be subject to optimization itself. On the other side, experience has shown that, for most learning algorithms, the performance is sufficiently robust toward small variations of the parameters. Thus, as long as the discretization contains a value that is not too far from the theoretical optimum, AutoML can be expected to find a good solution. Besides, it is interesting to note that restricting the set of possible parameter values (via discretization or in any other way) may also have a positive influence, as it comes with a regularization effect.

5.2 The node evaluation function

ML-Plan adopts a best-first search algorithm in order to identify good pipelines. A best-first search algorithm explores an implicitly given graph by assigning a number to each node and choosing in each iteration the node with the currently best (usually lowest) known value for expansion. Expansion means computing all successors of a node. The graph description consists of the root node, the successor computation function used for the expansion, and a predicate over the nodes that tells whether or not a node is a goal node. The task is to find a path from the root to a goal node with a minimum score, and a best-first algorithm tries to find such a path by expanding the intermediate nodes with minimum scores.

Since the prediction error as the solution quality does not decompose over the path (which is a necessary requirement for A*), we adopt a randomized depth-first search similar to the one applied in Monte Carlo tree search (MCTS) (Browne et al. 2012) to inform the search procedure. Given the node for which we need a score, we choose a random path to a goal node. This is achieved by randomly choosing a child node of the node itself, then randomly choosing a child node of the child node, etc. until a leaf node is reached; note that every leaf node is also a goal node in our search graph. We then compute the solution "qualities" of n such random completions and take the minimum as an estimate for the best possible solution that can be found beneath that node.

There are two main differences to standard MCTS when used with the UCT algorithm (Kocsis et al. 2006). First, UCT aims at optimizing the *average* score achieved below a node. While this is reasonable if one assumes that there is no full control about the eventual track that will be taken in the graph, such as in multi-player games, in our case, we want to optimize for the *minimum* score. Modifications of UCT for single-player games that take this issue into account have been presented (Schadd et al. 2008; Bjornsson and Finnsson 2009). Second, MCTS takes a rather asymptotic view where a large number of (cheap) play-outs makes sure that the algorithm converges to the optimal solution. However, play-outs in AutoML are quite expensive, so ML-Plan tries to reliably detect sub-optimal solutions early and effectively prunes them if all completions delivered bad results.

Since the node evaluation function computes solutions, we propagate these solutions to the search algorithm. More precisely, we propagate the best of the n solutions drawn for each node to the search routine. This way, ML-Plan is (as long as at least one node has been evaluated) always able to return solutions even if the main search routine did not already discover any goal node.

ML-Plan supports two procedures to determine the qualities of solutions:

1. *k-fold cross-validation (CV)* This is the standard cross-validation used by many approaches, which, in our case, is only applied to the portion of the data that is allocated for search. The dataset is split into k folds and, in k iterations, the validation procedure uses $k-1$ of them for training and the remaining one for validation. The performance is then the average performance over the k runs. Auto-WEKA adopts tenfold cross-validation. In ML-Plan, the number of folds can be defined by the user.
2. *Monte Carlo cross-validation (MCCV)* This technique is also called "hold out". The data (allocated for search) is partitioned k times into a stratified training and validation set. For each of the k splits, the solution pipeline is trained with the respective training set and tested on the validation set. The mean 0/1-loss of this evaluation is the score of that solution.

Even though the evaluation technique influences the overall algorithm performance, it is not meant to be optimized by the user but rather to give flexibility for scientific analysis.

In fact, ML-Plan is configured to use MCCV with 5 iterations, each of which with a split of 70% for training and 30% for validation. The ability to use other techniques has been included to eliminate confounding factors when comparing ML-Plan to other AutoML tools using different evaluation techniques. In fact, the choice of the validation technique can have a significant impact on the algorithm performance. On one hand, more exhaustive validations bring more reliable estimates of the quality of a single solutions. On the other hand, these validations can be very expensive in terms of time and memory; for some datasets, this can take several minutes, or even hours, and quickly exhaust the time resources of the entire algorithm.

Coming back to the discussion of the random completion strategy, we observed that the estimates acquired by the above strategy unfortunately give rather confusing estimates when used in the upper region of the search graph. We observed that sub-trees with very good solutions are sometimes effectively pruned just because all completions of the top-node of that sub-tree led to highly sub-optimal solutions. For example, on the MADELON dataset, we obtain around 18% error rate after 1 min for a pipeline consisting of a scaling preprocessor and a random forest classifier, but applying the random completion from the very beginning suggests that the best solution quality under a top-level node that contains this solution is about 49%. The node is effectively pruned, and the quality of the returned solution is around 45%. The problem is that the random completions adopt inappropriate classification algorithms with highly sub-optimal solutions. In other words, the top-layer nodes embrace many different types of solutions, so the estimates may deviate significantly from the truly best score obtainable in a corresponding sub-tree.

Based on several observations of this type, ML-Plan was designed to expand *all* nodes for the algorithm *selection* part of the search tree without computing any node evaluations and adopts the random completions only for nodes in the deeper layers corresponding to algorithm *configuration* decisions. More precisely, ML-Plan assigns a value of 0 (optimum) to all nodes in which the classifier has not yet been chosen. This effectively means to disable the informed search for the algorithm selection part, but since only a few hundred algorithm selections are usually possible, all these possibilities can be efficiently enumerated. The random completion technique then is only applied to nodes below the algorithm selection region. This strategy ensures that each combination of preprocessors and classifiers is at least considered once with random completions and also increases the reliability of these estimates.

In order to break ties among the different algorithm selections, ML-Plan defines a (preferential) pre-order on the classification algorithms. This order is used to sort the "leaf" nodes of the algorithm selection region and hence defines in which order the first random completion evaluations are conducted. More precisely, nodes whose partial plan contains the `selectClassifier` action but no parameters have been refined, are leaf nodes of the algorithm selection region and receive a score $\frac{k}{n}$, where k is their rank and n is the number of classifier algorithms; unkranked algorithms have $k = n$. The order is similar to the one used in Auto-WEKA: KNN, random forests, voted perceptron, SVM, logistic regression (in this order). These choices are based on results of Auto-WEKA reported in Thornton et al. (2013). In ongoing work, we develop a method for deciding this order in a more flexible, data-driven manner, specifically tailored for the learning problem at hand; this approach is not yet realized, however.

Since we use two different node evaluation functions within the same graph, these need to be made consistent to avoid strange behavior. To this end, the scores in the upper part are scaled by the factor 10^{-3}. This way, they are on a consistent scale with the accuracy estimates and are preferred (unless ML-Plan gets an estimate for a solution with error rate below 10^{-3}, which would indicate an almost perfect solution).

5.3 Mitigating oversearch: a two-phase model

Intuitively, an extensive, systematic search for good predictors should bear a strong risk of over-fitting, and previous approaches have confirmed this intuition (Thornton et al. 2013). By their ability to choose among all learners and even construct new and arbitrary large ones using ensemble methods, AutoML tools are on the right extreme of the bias-variance spectrum. If the data available for the search process is not sufficiently substantial and representative for "real" data, the danger of over-fitting in AutoML is higher than for basic learning algorithms.

We address this problem with a two-phase search mechanism. The first phase covers the actual search in the space described above, and produces a collection of solution candidates. The second phase takes these candidates and selects the one that minimizes the estimated generalization error. This estimation is achieved by splitting the data given to the AutoML tool into two sets D_{search} and D_{select}. Phase 1 only has access to D_{search}, which is used for the evaluation of nodes as described in the previous section. Phase 2 performs Monte Carlo cross-validation on $D_{search} \cup D_{select}$ for a fixed number of iterations (10 in our evaluation). For each iteration, we obtain a stratified split (70% train and 30% validation) that is used to train and evaluate a candidate solution s. We estimate the generalization error of s by taking the average of the "internal" evaluation of s as in the previous section (only on D_{search}), and the .75-percentile of the evaluations that include D_{select}. We use the .75-percentile to make the estimate for the generalization more conservative (and robust) but also robust toward outliers. Intuitively, a good solution should not only have a strong average performance on the internal data, but also perform well on most of the more general splits.

Since phase 1 may detect hundreds or even thousands of models, phase 2 only operates on a small subset of these solutions. The portfolio used in the second phase consists of two equally large subsets S_{best} and S_{random}. The size of these sets is fixed by a parameter k; we used a size of 25 in our evaluation. S_{best} and S_{random} contain, respectively, the k best and random solutions the internal evaluation of which deviates by at most ε from the optimal one. The random candidates are important to ensure a certain diversity in the selection set, but the expected quality should still be reasonably good. Since the domain for the prediction loss is fixed to [0, 1], ε is not a relative but an absolute deviation from the optimum; in our experiments, we set $\varepsilon = 0.03$.

Of course, this prevention strategy comes at a cost. First, less data is available for evaluating the nodes, which in particular implies that models with higher variance are more likely to be discarded even though they could be preferable choices. Second, the selection phase consumes valuable search budget. The search in phase 1 is accompanied by a timer that estimates the time required by phase 2; this is done by extrapolating from the times required to evaluate the models during search. When the expected time for phase 2 is close to the remaining overall budget, ML-Plan switches to phase 2.

6 Experimental evaluation

6.1 Experimental setup

The experimental evaluation of ML-Plan is twofold. First, we compare ML-Plan as introduced in the preceding sections to other state-of-the-art AutoML tools. Second, we carry out a more detailed analysis of individual components of the ML-Plan. To this end, we evaluate the influence of isolated concepts using Auto-WEKA as a baseline. More specifically, we assess

the impact of HTN, Monte Carlo cross-validation as evaluation technique, and adopting a second phase for selecting the returned solution on the performance of ML-Plan. Furthermore, we discuss the combinations of preprocessors and classifiers as chosen by ML-Plan.

In the first part, we compare ML-Plan to Auto-WEKA (version 2.3) (Thornton et al. 2013), auto-sklearn (both vanilla and warm-started with ensembles) (Feurer et al. 2015), and TPOT (Olson and Moore 2016), which represent the state-of-the-art in AutoML.

Due to missing features in RECIPE to set a timeout and other technical issues, we refrain from a comparison to RECIPE. In order to reduce the number of confounding factors as introduced by using different libraries, i.e., WEKA (Java) or scikit-learn (Python), we run ML-Plan once using WEKA and once using scikit-learn. Since Auto-WEKA was already shown to outperform other (more basic) baselines (Thornton et al. 2013), we do not consider these anymore.

To maximize the insights about the performances of individual changes brought in by ML-Plan, in the second part of our evaluation, we compare four different configurations of ML-Plan against Auto-WEKA. We chose Auto-WEKA over the other AutoML tools since (i) Auto-WEKA internally uses the same search strategy as auto-sklearn (SMAC), and (ii) ML-Plan itself is implemented in Java, so that confounding factors arising from the usage of different platforms can be excluded. Apart from the different search space model (HTN vs. SMAC), ML-Plan brings two new aspects into play, which could be confounding factors in a comparison with Auto-WEKA. The first is a different solution evaluation technique: Auto-WEKA uses tenfold cross-validation (10-CV) while ML-Plan, by default, uses fivefold Monte Carlo cross-validation (5-MCCV). Moreover, ML-Plan adopts a selection phase to prevent overfitting, whereas nothing comparable is used in Auto-WEKA.

To isolate the different effects, we consider the variants of ML-Plan for 10-CV/5-MCCV with the selection phase disabled (SD) and enabled (SE), respectively. If the selection phase is disabled, ML-Plan uses all the data during search. Thus, the version of ML-Plan that is closest to Auto-WEKA and only deviates in the search space exploration is 10-CV-SD.

Our evaluation is based on a selection of 20 datasets from the openml.org (Vanschoren et al. 2013) repository, all of which have previously been used to evaluate AutoML approaches (Thornton et al. 2013; Feurer et al. 2015). More precisely, we present results for the same datasets that were used in the original Auto-WEKA paper (Thornton et al. 2013). The implementation of ML-Plan, the evaluation code that produced the results shown in this section, and the used datasets are publicly available to assure reproducibility.[3]

Results were obtained by carrying out 20 runs on each dataset with timeouts of 1 h and 1 day, respectively. Depending on the overall timeout, the timeout for the internal evaluation of a single solution was set to 5m and 20m, respectively. In each run, we used 70% of a stratified split of the data for the respective AutoML framework and 30% for testing. Note that we used the *same* splits for *all* frameworks, i.e., for each split and each timeout, we ran once Auto-WEKA, auto-sklearn, TPOT, and ML-Plan. Likewise, the timeout to evaluate a single pipeline was set to the same values for all frameworks, i.e., we did not use the default values. The computations were executed on 100 Linux machines in parallel, each of them equipped with 8 cores (Intel Xeon E5-2670, 2.6 Ghz) and 32 GB memory. The accumulated time of all experiments was over 400k CPU hours (over 45 CPU years).

Runs that did not adhere to the time or resource limitations (plus a tolerance threshold) were canceled without considering their results. That is, we canceled the algorithms if they did not terminate within 110% of the predefined timeout. Likewise, the algorithms were killed if they consumed more resources (memory or CPU) than allowed, which happens as

[3] https://github.com/fmohr/ML-Plan.

both implementations fork new processes whose overall CPU and memory consumption is hard to control.

An exception for the timeout rule has been made for TPOT, as not even a single result has been returned in the long runs. Therefore, we configured TPOT to log intermediate solutions and considered the most recent one of these to compute the respective value in the results tables.

6.2 ML-Plan versus other AutoML tools

The results of the comparison with other AutoML tools is summarized in Table 1 (1 h timeout) and Table 2 (1 day timeout). The tables show the mean 0/1-loss over the 20 runs and the standard deviation. The best average results per library and dataset are in bold. A $\bullet$ indicates that ML-Plan is significantly better, and analogously a $\circ$ that it is significantly worse, than another approach; significance is determined by a t test with $p = 0.05$. At the bottom of the table, the numbers of wins and losses of each tool and the numbers of statistically significant improvements and degradations are summarized over the various datasets.

The key message resulting from Tables 1 and 2 is that ML-Plan is competitive with the other approaches in terms of the predictive accuracy of the returned solutions, and even shows some advantages. ML-Plan largely dominates Auto-WEKA in both time setups. It performs similar and sometimes superior to auto-sklearn (vanilla) and TPOT in the 1 h run and still with a slight advantage after 1 day. TPOT did not return any results for larger resp. more complex datasets, such as CIFAR10, DEXTER or MNIST. Thus, TPOT seems to scale worse than all the other AutoML approaches, which might be a configuration issue. We now discuss the results in some more detail.

In the setting of using WEKA as a library, we observe that ML-Plan clearly dominates Auto-WEKA and obtains worse performance on only a few datasets. For a number of datasets, ML-Plan achieved significantly better results even after 1 h compared to the result returned by Auto-WEKA within one *day*; e.g., on AMAZON, CONVEX, KRVSKP, and SEMEION. On some datasets, the gap is quite drastic, e.g., there are differences of 25% on AMAZON and CONVEX. But even when the difference is not so pronounced, the advantage of ML-Plan is often quite substantial, showing an improvement of at least 2% in 9 of 20 cases with a timeout of 1 day. In total, ML-Plan achieves the best result on 18 of 20 datasets for a timeout of 1 h, and on 17 of 20 for a timeout of 1 day. Out of these, ML-Plan is significantly better than Auto-WEKA 12 (1 h) and 14 (1 day) times, whereas a significant degradation can only be observed once for both the timeouts.

Coming to the comparison with AutoML tools based on scikit-learn, there is no such clear dominance, although significant improvements over auto-sklearn can still be observed. Irrespective of the timeout, ML-Plan performs best on 9 of 20 datasets while auto-sklearn yields the best result in 6 of 20 cases, and TPOT in 7 of 20 (1 h) resp. 6 of 20 (1 day) cases. Within the given timeouts, we note 7 (1 h) resp. 5 (1 day) significant improvements over auto-sklearn, whereas significant degradations occur in 1/20 resp. 3/20 cases.

Comparing ML-Plan to TPOT, there is no clear winner or loser. However, given the default parametrization (except for timeouts), TPOT often did not return any result within the specified timeout, so that the results shown in Table 2 had to be recovered from its log output. For larger datasets, TPOT did not even output a preliminary candidate solution within the specified timeout. This might be due to inappropriate parameters for the evolutionary algorithm, such as population size etc., which would have to be adapted to each specific dataset. Here, we only considered the default parameter setting and refrained from optimizing

Table 1 Mean 0/1-losses (in%) ± SD for 1 h timeout

Data set	WEKA		Scikit-learn			
	ML-Plan	Auto-WEKA	ML-Plan	Auto-sklearn-v	Auto-sklearn-we	TPOT
ABALONE	73.72 ± 1.23	**73.46 ± 1.08**	73.77 ± 1.11	82.92 ± 8.38 •	80.59 ± 8.32 •	**73.14 ± 1.02**
AMAZON	**25.55 ± 1.80**	51.72 ± 2.69 •	22.92 ± 3.04	27.83 ± 5.72 •	**19.72 ± 2.18** ○	–
CAR	1.27 ± 0.56	**0.66 ± 0.38** ○	**0.34 ± 0.51**	1.38 ± 0.67 •	1.26 ± 0.53 •	0.37 ± 0.33
CIFAR10	**68.90 ± 2.54**	–	**77.04 ± 8.71**	–	–	–
CIFAR10SMALL	**58.25 ± 0.62**	70.23 ± 0.00 •	58.09 ± 1.88	58.11 ± 0.65	**56.62 ± 1.50** ○	–
CONVEX	**15.60 ± 0.23**	46.83 ± 0.39 •	16.82 ± 2.22	16.34 ± 0.78	**13.56 ± 0.33** ○	–
CREDIT- G	**25.54 ± 1.28**	26.50 ± 2.32	24.56 ± 2.53	25.95 ± 1.89	25.39 ± 0.88	**23.91 ± 2.22**
DEXTER	**8.73 ± 2.67**	11.44 ± 2.68 •	**4.63 ± 1.29**	8.10 ± 2.13 •	6.01 ± 1.17 •	–
DOROTHEA	**6.49 ± 1.23**	–	8.69 ± 1.54	6.32 ± 1.16 ○	**6.02 ± 1.01** ○	–
GISETTE	**2.92 ± 0.27**	3.90 ± 0.40 •	2.76 ± 0.36	2.56 ± 0.36	**2.24 ± 0.33** ○	–
KRVSKP	**0.54 ± 0.20**	2.61 ± 2.68 •	0.70 ± 0.23	0.75 ± 0.32	0.77 ± 0.31	**0.58 ± 0.24**
MADELON	**19.28 ± 2.26**	25.52 ± 3.80 •	**14.75 ± 2.06**	15.99 ± 1.73	15.33 ± 1.89	15.3 ± 2.46
MNIST	**3.48 ± 0.11**	7.23 ± 0.20 •	3.87 ± 0.66	3.60 ± 0.11	**3.50 ± 0.11** ○	–
MNISTROTATIO	**55.88 ± 5.94**	78.56 ± 0.43 •	54.55 ± 13.19	**51.86 ± 1.26**	52.06 ± 0.37	–
SECOM	**6.47 ± 0.16**	6.55 ± 0.39	6.79 ± 0.00	6.69 ± 0.35	6.68 ± 0.44	**6.49 ± 0.19** ○
SEMEION	**6.78 ± 0.84**	12.59 ± 3.93 •	**4.66 ± 0.64**	6.79 ± 1.34 •	5.79 ± 1.02 •	6.22 ± 0.97 •
SHUTTLE	**0.01 ± 0.01**	0.12 ± 0.06 •	0.02 ± 0.01	0.02 ± 0.01	**0.02 ± 0.01**	0.02 ± 0.02
WAVEFORM	**13.24 ± 0.64**	13.35 ± 0.81	13.23 ± 0.76	13.6 ± 0.75	13.23 ± 0.57	**12.94 ± 0.62**
WINEQUALITY	**32.62 ± 0.91**	33.69 ± 1.90 •	**32.53 ± 1.31**	36.83 ± 1.27 •	35.87 ± 1.18 •	32.94 ± 1.09
YEAST	**39.37 ± 2.54**	39.72 ± 2.29	39.52 ± 2.66	40.51 ± 2.17	38.99 ± 2.28	**38.47 ± 2.36**
Wins/losses	18/2	2/18	6/14	1/19	7/13	7/13
t test imp/deg	–	12/1	–	6/1	5/6	1/1

Table 2 Mean 0/1-losses (in%) ± SD for 1 day timeout

Data set	WEKA		Scikit-learn			
	ML-Plan	Auto-WEKA	ML-Plan	Auto-sklearn-v	Auto-sklearn-we	TPOT
ABALONE	**72.83 ± 0.89**	73.46 ± 0.71 •	73.46 ± 1.07	–	74.8 ± 0.94 •	**73.35 ± 0.93**
AMAZON	**25.20 ± 2.31**	50.28 ± 3.51 •	**18.52 ± 1.88**	22.0 ± 1.00	19.94 ± 2.17	–
CAR	1.18 ± 0.53	**0.24 ± 0.26** ○	**0.35 ± 0.44**	1.64 ± 0.96 •	1.15 ± 0.30 •	0.40 ± 0.33
CIFAR10	**55.26 ± 0.51**	64.06 ± 1.37 •	**60.31 ± 4.12**	–	–	–
CIFAR10SMALL	**58.31 ± 0.58**	62.09 ± 3.19 •	56.46 ± 2.01	55.08 ± 2.59	**47.47 ± 1.90** ○	–
CONVEX	**14.80 ± 0.68**	45.70 ± 5.46 •	14.93 ± 1.72	12.16 ± 1.71 ○	**9.77 ± 1.06** ○	–
CREDIT- G	**25.17 ± 2.52**	26.44 ± 2.05	24.90 ± 2.09	–	–	**23.53 ± 1.52** ○
DEXTER	9.83 ± 2.71	**9.82 ± 2.43**	**5.06 ± 1.44**	7.30 ± 0.79 •	5.24 ± 1.99	–
DOROTHEA	**6.37 ± 0.93**	11.01 ± 1.73 •	6.59 ± 0.87	**6.04 ± 1.05**	6.58 ± 1.33	–
GISETTE	**2.88 ± 0.30**	4.18 ± 0.60 •	2.14 ± 0.27	2.22 ± 0.29	**1.97 ± 0.25**	–
KRVSKP	**0.53 ± 0.25**	4.02 ± 2.70 •	**0.66 ± 0.37**	0.74 ± 0.32	0.68 ± 0.33	1.08 ± 2.04
MADELON	**17.73 ± 3.07**	20.34 ± 2.53 •	14.37 ± 1.64	14.7 ± 1.61	**13.47 ± 1.14**	15.26 ± 0.73
MNIST	**3.44 ± 0.12**	5.39 ± 0.67 •	2.98 ± 0.36	2.90 ± 0.51	**1.62 ± 0.06** ○	–
MNISTROTATIO	**50.12 ± 1.30**	74.30 ± 4.79 •	47.33 ± 4.49	43.49 ± 2.19 ○	**31.51 ± 1.62** ○	–
SECOM	**6.48 ± 0.12**	6.60 ± 0.42	6.82 ± 0.09	6.57 ± 0.27	6.64 ± 0.23	**6.50 ± 0.20** ○
SEMEION	**4.73 ± 1.03**	8.42 ± 2.32 •	**4.79 ± 1.11**	6.29 ± 1.11 •	5.82 ± 1.37 •	6.06 ± 1.04 •
SHUTTLE	**0.01 ± 0.01**	0.13 ± 0.07 •	0.02 ± 0.01	0.01 ± 0.02	0.02 ± 0.01	**0.01 ± 0.02**
WAVEFORM	13.27 ± 0.64	**13.05 ± 0.68**	13.23 ± 0.83	13.6 ± 0.74	13.42 ± 0.69	**13.1 ± 0.66**
WINEQUALITY	**32.54 ± 0.99**	33.58 ± 1.23 •	**32.45 ± 0.98**	36.25 ± 1.53 •	36.03 ± 0.82 •	32.66 ± 0.77
YEAST	**38.30 ± 2.23**	39.80 ± 2.56	39.79 ± 2.38	39.27 ± 0.61	**37.73 ± 0.00** ○	38.75 ± 2.37
Wins/losses	17/3	3/17	9/11	6/14	7/13	6/14
t test imp/deg	–	14/1	–	4/2	4/5	2/2

the hyperparameters of TPOT; the only parameter we set was the timeout for evaluating a single pipeline. However, algorithm-specific configurations should not play an important role in AutoML since the goal is precisely to enable the functionality to non-experts. The number of significant improvements (1 for 1 h, 2 for 1 day) and degradations are evenly balanced for the datasets for which results could be obtained from TPOT. Due to this, we conclude ML-Plan to be at least competitive with TPOT.

Comparing ML-Plan to auto-sklearn, ML-Plan appears to be slightly superior. Yet, auto-sklearn with warm-start and ensembles outperforms ML-Plan in some cases. The latter comparison is not unproblematic, however, since additional features such as warm-starting and ensembling are not (yet) incorporated in ML-Plan. Indeed, our focus is on the comparison of search strategies, i.e., the algorithmic core, and less on complete AutoML systems/tools. In this sense, our primary comparison is between ML-Plan and auto-sklearn vanilla, while the performance of auto-sklearn with warm-start and ensembles is merely presented as an additional reference. Adopting this perspective, our interpretation is that ML-Plan does have an advantage over the core technique used in auto-sklearn (SMAC). Nevertheless, the improved performance of auto-sklearn under warm-start and ensembles provides an incentive to add these techniques to ML-Plan as well.

Unfortunately, for the datasets ABALONE and CREDIT- G, auto-sklearn did not return any results for the 1 day evaluations within the resource limitations, although there have been results already for the 1 h runs. According to the logs, we assume that this might be due to a bug in the auto-sklearn implementation.

The reader may have noticed significant differences between the results we report for Auto-WEKA and auto-sklearn in the 1 day run compared to the results reported in Thornton et al. (2013) and Feurer et al. (2015) for some of the datasets. For most of these (including, e.g., AMAZON), the authors of Auto-WEKA have confirmed the correctness of our results. For the others, such as CONVEX, there are two possible explanations. First, we only granted 24 h compared to 30 h as in previous studies. Second, the experiments in these studies were conducted on only a single train/test-split, which implies a high variance.

All in all, we notice that the more time is available for search, the closer the gap between the different AutoML tools. This comes at no surprise as, asymptotically, most of the algorithms return the same (best) solution—excepting TPOT, which is able to construct more complex pipelines (with multiple preprocessors). Furthermore, our results show that, for some datasets, scikit-learn-based approaches perform substantially better than the ones based on WEKA and vice versa. One possible reason for this might be a different portfolio of preprocessing and classification algorithms. Another reason might simply be the fact that the evaluation of candidate solutions in scikit-learn is much faster than in WEKA for most of the algorithms. For example, compared to WEKA, ML-Plan is able to do twice as many evaluations with scikit-learn.

6.3 Detailed analysis of ML-Plan

To better understand how HTN, Monte Carlo cross-validation, and the selection phase influence the performance of ML-Plan, we examined different configurations of ML-Plan. Since the techniques used by ML-Plan are essentially the same for both its WEKA and scikit-learn version, we conducted the experiments only for one of these versions; we chose the WEKA implementation, because the gap to Auto-WEKA is the largest one.

The results of these experiments are summarized in Table 3. The table shows the mean 0/1-loss and the standard deviation per configuration and dataset for a timeout of 1 h. The

Table 3 Mean 0/1-losses (in%) ± SD for 1 h timeout

Data set	Auto-WEKA	ML-Plan 10-CV SD	ML-Plan 10-CV SE	ML-Plan 5-MCCV SD	ML-Plan 5-MCCV SE
ABALONE	73.46 ± 1.08	73.26 ± 1.77	72.54 ± 0.46•	**72.34 ± 0.25**•	73.72 ± 1.23
AMAZON	51.72 ± 2.69	27.86 ± 1.68•	29.00 ± 0.00•	29.00 ± 0.00•	**25.55 ± 1.80**•
CAR	0.66 ± 0.38	0.97 ± 0.55○	0.58 ± 0.00	**0.48 ± 0.30**	1.27 ± 0.56○
CIFAR10SMALL	70.23 ± 0.00	70.51 ± 0.57○	**57.79 ± 0.06**•	**57.79 ± 0.06**•	58.25 ± 0.62•
CIFAR10	–	70.62 ± 0.30	70.40 ± 0.11	70.40 ± 0.11	**68.90 ± 2.54**
CONVEX	46.83 ± 0.39	27.48 ± 0.40•	27.45 ± 0.28•	27.76 ± 0.76•	**15.60 ± 0.23**•
CREDIT- G	26.50 ± 2.32	25.73 ± 1.61	**23.83 ± 0.00**•	25.67 ± 0.50	25.54 ± 1.28
DEXTER	11.44 ± 2.68	**8.34 ± 2.00**•	8.99 ± 2.81•	9.27 ± 0.28•	8.73 ± 2.67•
DOROTHEA	–	7.32 ± 1.56	6.88 ± 0.14	7.75 ± 0.73	**6.49 ± 1.23**
GISETTE	3.90 ± 0.40	3.30 ± 0.54•	2.31 ± 0.17•	**2.12 ± 0.36**•	2.92 ± 0.27•
KRVSKP	2.61 ± 2.68	0.58 ± 0.22•	0.78 ± 0.27•	0.62 ± 0.11•	**0.54 ± 0.20**•
MADELON	25.52 ± 3.80	19.32 ± 2.17•	23.01 ± 0.77•	21.02 ± 0.07•	**19.28 ± 2.26**•
MNISTROTATIO	78.56 ± 0.43	68.78 ± 0.56•	68.37 ± 0.42•	66.56 ± 2.22•	**55.88 ± 5.94**•
MNIST	7.23 ± 0.20	16.98 ± 0.68○	3.54 ± 0.01•	3.54 ± 0.01•	**3.48 ± 0.11**•
SECOM	6.55 ± 0.39	**6.41 ± 0.21**	6.86 ± 0.21○	7.07 ± 0.00○	6.47 ± 0.16
SEMEION	12.59 ± 3.93	6.98 ± 1.00•	7.13 ± 0.00•	**6.69 ± 0.21**•	6.78 ± 0.84•
SHUTTLE	0.12 ± 0.06	**0.01 ± 0.01**•	**0.01 ± 0.00**•	0.02 ± 0.00•	**0.01 ± 0.01**•
WAVEFORM	13.35 ± 0.81	13.04 ± 0.65	**12.65 ± 0.13**•	12.84 ± 0.06•	13.24 ± 0.64
WINEQUALITY	33.69 ± 1.90	32.80 ± 1.16	32.31 ± 0.00•	**31.69 ± 0.48**•	32.62 ± 0.91•
YEAST	39.72 ± 2.29	38.51 ± 2.46	**36.57 ± 0.00**•	41.20 ± 1.62○	39.37 ± 2.54
t test imp/deg versus WEKA	–	9/3	17/1	14/2	12/1
Best/non–sig worse/worse	0/2/18	3/4/13	5/4/11	6/2/12	9/3/8

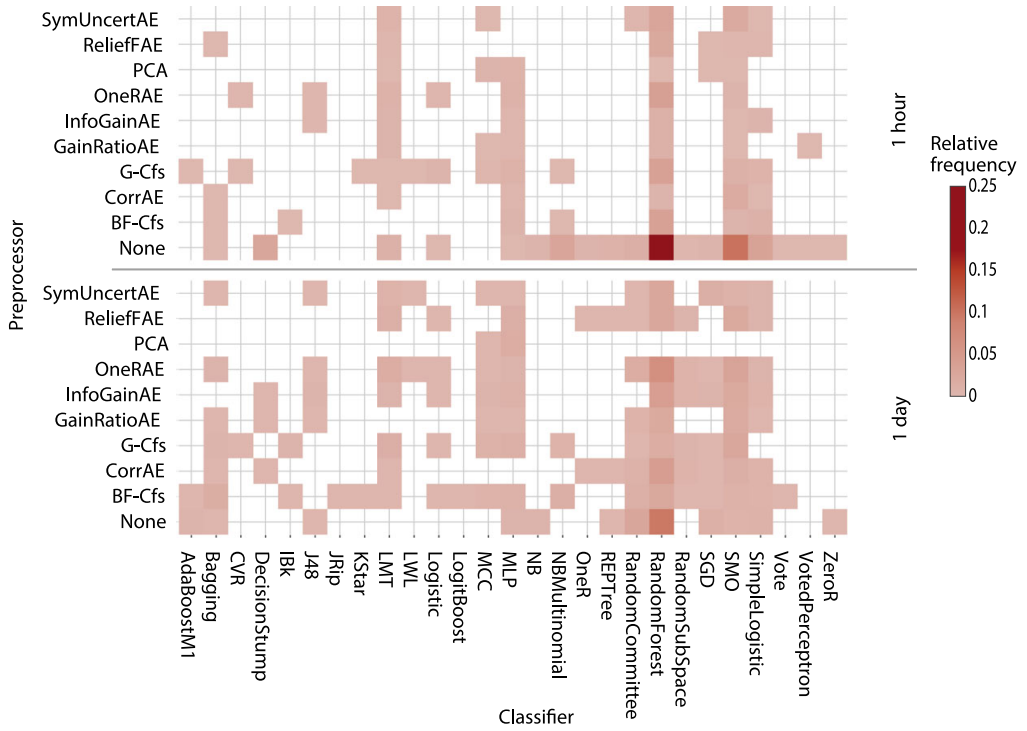


Fig. 3 Relative frequency of WEKA preprocessor-classifier combinations as returned by ML-Plan after 1 h resp. 1 day

best average results per dataset are highlighted in bold; additionally, those results that are not significantly ($p = 0.05$ using a t test) worse than the best result are underlined. Moreover, a • denotes a significant improvement of the respective ML-Plan configuration over Auto-WEKA, and ∘ a significant degradation. At the bottom of the table, we summarize how many times a configuration of ML-Plan yielded a statistically significant improvement or degradation with respect to the performance of Auto-WEKA. Furthermore, we count how many times each variant performs best, not statistically significantly worse, and significantly worse than the best.

Having eliminated the essential confounding factors in the 10-CV-SD variant of ML-Plan, we conclude from these experiments that solving the AutoML problem with HTN planning already gives significantly better results than using Auto-WEKA. Note that we do not claim that ML-Plan is generally superior to applying sequential parameter optimization in AutoML; instead, the results only apply to the implementation of parameter optimization in Auto-WEKA.

Comparing the test performances of the different configurations of ML-Plan, there is no single version that strictly dominates all others. Yet, all versions outperform Auto-WEKA, which indicates that much of the performance improvement can be traced back to the use of HTN planning.

However, counting the number of datasets where a configuration of ML-Plan achieves a significant improvement, it can be seen that enabling the selection phase yields more such improvements when 10-CV is used as the evaluation technique. Moreover, from this perspective, 10-CV together with the selection phase enabled performs the best compared to Auto-WEKA, yielding 17 significant improvements in 18 possible cases (Auto-WEKA did not return any result on 2 of the 20 datasets). Surprisingly, this observation does not hold for the case of 5-MCCV. In fact, switching on the selection phase while keeping the evaluation function to be 5-MCCV leads to less significant improvements.

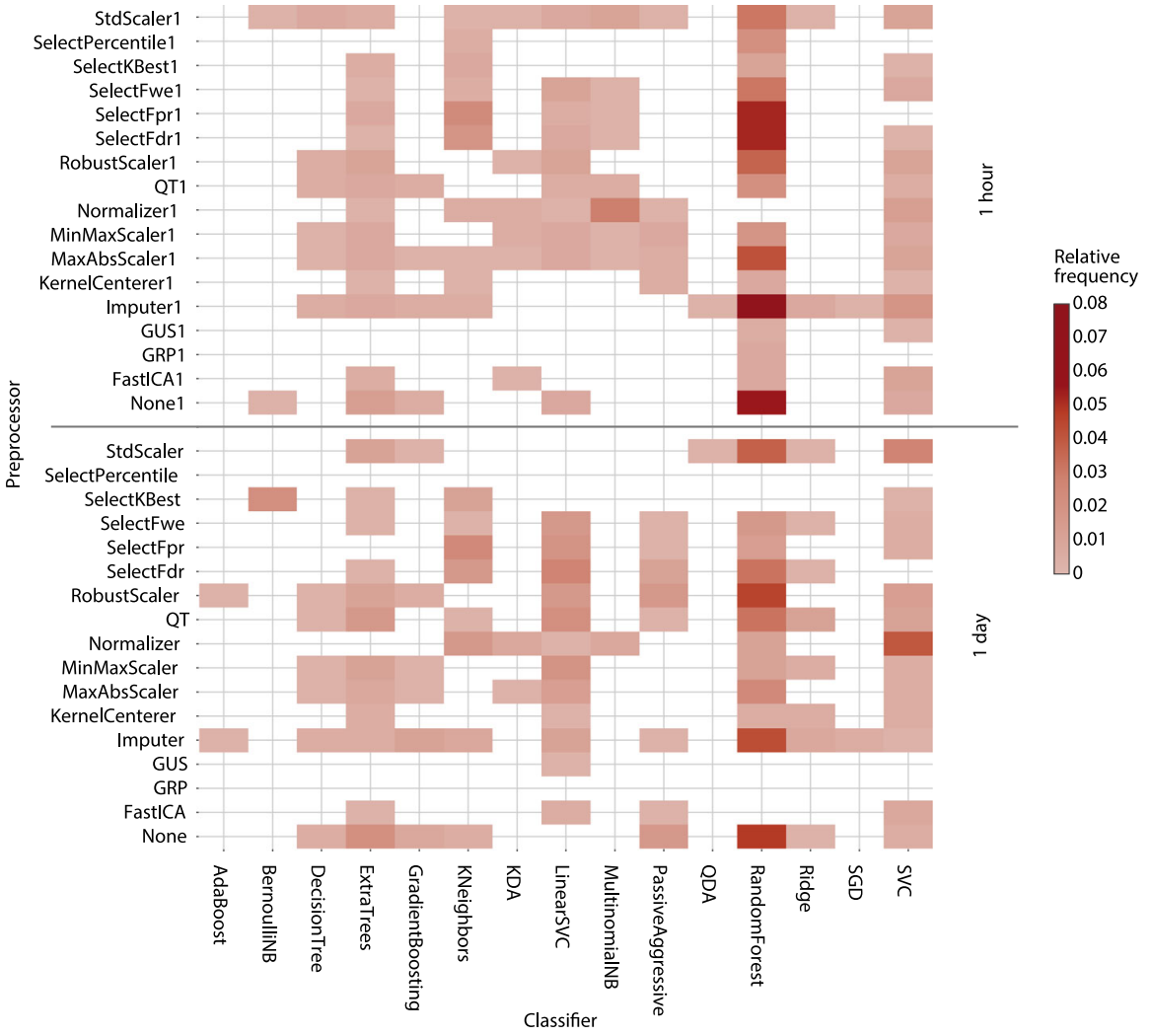


Fig. 4 Relative frequency of scikit-learn preprocessor-classifier combinations as returned by ML-Plan after 1 h resp. 1 day

Although the results are quite heterogeneous, we opt for using 5-MCCV-SE as a standard parametrization. While 10-CV-SE and 5-MCCV-SD yield 5 resp. 2 more significant improvements compared to Auto-WEKA, they do not ourperform the configuration with 5-MCCV and selection phase enabled. In fact, 5-MCCV-SE yields the best observed performance on nearly half of the evaluated datasets among the different configurations of ML-Plan. Counting the number of wins of those configurations, on one hand we see that the selection phase proves beneficial, and on the other hand 5-MCCV seems to be advantageous as compared to 10-CV.

Nevertheless, for some datasets it can also be seen that the test set performance worsens when enabling the selection phase. As one possible reason, recall that a portion of the training data is reserved to be used only in the selection phase, but in most of the cases, this effect does not arise on the same datasets for the different evaluation techniques. Further investigation of this observation poses interesting future work, which may help to automatically adapt the configuration of ML-Plan to the properties of the problem at hand.

6.3.1 Selected classifiers and preprocessors

The plots in Figs. 3 and 4 show the frequency with which a combination of preprocessor and classifier was selected by ML-Plan using WEKA and scikit-learn, respectively. They summarize the frequency over *all* datasets for each timeout, i.e., 1 h and 1 day.

While the purpose of the plots is to give an insight into the algorithm choices, they do in no way reflect the true distribution of optimal solutions. In particular, the dominant role of random forests does *not* support the idea that they are a dominant optimal choice (even though they often *are* good models of course). The node evaluation enforces a strong bias towards some models, including random forests, for a given timeout. In many of the datasets for which random forests were selected (e.g., CIFAR10SMALL, CONVEX, MNIST), ML-Plan observed only around 10–50 solutions in total (timeout 1 h).

In fact, only focusing on random forests can lead to high regrets. For example, the loss of random forests on AMAZON is over 70%, which is more than 40% points away from the best solution we report here.

This being said, our interpretation of the results resembles the one in Thornton et al. (2013). Looking at the variety of preprocessors and classifiers chosen for the different problems, the optimization effort is clearly justified. Even with a strong selection bias in favor of random forests, SVM, and KNN (which in some cases were the only models considered at all), other algorithms were better in more than 40% of the cases.

However, we also observe that, for some datasets, the current approach is simply not adequate in terms of search space coverage. As described above, since the evaluation of candidates is so costly for some problems, we only explored a very tiny part of the search space. This problem calls for more sophisticated node evaluation techniques in order to explore broader parts of the search space. One possibility is to reduce the amount of data considered, i.e., only work on a (random) sub-sample of instances.

7 Conclusion

We proposed ML-Plan, a new AutoML framework based on hierarchical task networks. Distinguishing features of ML-Plan include a conveniently structured solution space amenable to efficient search techniques, a reliable node evaluation based on random completions, and a strategy to avoid over-fitting. We have shown that our implementation of ML-Plan is highly competitive and often outperforms the state-of-the art tools Auto-WEKA, auto-sklearn, and TPOT.

In follow-up work, we plan to elaborate on the expressiveness of the HTN formalism, and to exploit its potential for creating more complex, variable-length pipelines. In particular, we are already working on optimizing over pipelines with algorithms from both libraries (WEKA and scikit-learn) simultaneously (Mohr et al. 2018). Moreover, the seed-strategy in the upper part of the search graph should be adaptive to the dataset instead of implementing a predefined preference on learning algorithms. Also, we expect the implementation of parameter refinement to yield better fine tuning. Last but not least, the current emphasis on exploitation can be balanced with more exploration, e.g., by occasionally choosing nodes for expansion at random.

Acknowledgements This work was supported by the German Research Foundation (DFG) within the Collaborative Research Center "On-The-Fly Computing" (SFB 901).

References

Bjornsson, Y., & Finnsson, H. (2009). Cadiaplayer: A simulation-based general game player. *IEEE Transactions on Computational Intelligence and AI in Games*, *1*(1), 4–15.

Browne, C., Powley, E. J., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., et al. (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, *4*(1), 1–43. https://doi.org/10.1109/TCIAIG.2012.2186810.

de Sá, A. G., Pinto, W. J. G., Oliveira, L. O. V., & Pappa, G. L. (2017). Recipe: A grammar-based framework for automatically evolving classification pipelines. In *European Conference on Genetic Programming* (pp. 246–261). Springer.

Erol, K., Hendler, J. A., & Nau, D. S. (1994). UMCP: A sound and complete procedure for hierarchical task-network planning. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems, University of Chicago, Chicago, Illinois, USA, June 13–15, 1994* (pp. 249–254). http://www.aaai.org/Library/AIPS/1994/aips94-042.php.

Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., & Hutter, F. (2015). Efficient and robust automated machine learning. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, & R. Garnett (Eds.), *Advances in neural information processing systems* (pp. 2962–2970). Curran Associates, Inc.

Ghallab, M., Nau, D. S., & Traverso, P. (2004). *Automated planning—Theory and practice*. New York City: Elsevier.

Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. *LION*, *5*, 507–523.

Kietz, J., Serban, F., Bernstein, A., & Fischer, S. (2009). Towards cooperative planning of data mining workflows. In *Proceedings of the Third Generation Data Mining Workshop at the 2009 European Conference on Machine Learning* (pp. 1–12). Citeseer.

Kietz, J. U., Serban, F., Bernstein, A., & Fischer, S. (2012). Designing KDD-workflows via HTN-planning for intelligent discovery assistance. In *5th planning to learn workshop WS28 at ECAI 2012* (p. 10).

Kocsis, L., Szepesvári, C., & Willemson, J. (2006). *Improved Monte-Carlo search*. Technical report 1, University of Tartu, Estonia.

Komer, B., Bergstra, J., & Eliasmith, C. (2014). Hyperopt-sklearn: Automatic hyperparameter configuration for scikit-learn. In *ICML workshop on AutoML*.

Kotthoff, L., Thornton, C., Hoos, H. H., Hutter, F., & Leyton-Brown, K. (2017). Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA. *The Journal of Machine Learning Research*, *18*(1), 826–830.

Lloyd, J. R., Duvenaud, D. K., Grosse, R. B., Tenenbaum, J. B., & Ghahramani, Z. (2014). Automatic construction and natural-language description of nonparametric regression models. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, Québec City, Québec, Canada* (pp. 1242–1250).

Mohr, F., Wever, M., Hüllermeier, E., & Faez, A. (2018). Towards the automated composition of machine learning services. In *Proceedings of the IEEE International Conference on Services Computing*. SCC.

Nau, D. S., Au, T., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., et al. (2003). SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research (JAIR)*, *20*, 379–404. https://doi.org/10.1613/jair.1141.

Nguyen, P., Hilario, M., & Kalousis, A. (2014). Using meta-mining to support data mining workflow planning and optimization. *Journal of Artificial Intelligence Research*, *51*, 605–644.

Nguyen, P., Kalousis, A., & Hilario, M. (2011). A meta-mining infrastructure to support KD workflow optimization. In *Proceedings of the PlanSoKD-11 Workshop at ECML/PKDD* (pp. 1–10).

Nguyen, P., Kalousis, A., & Hilario, M. (2012). Experimental evaluation of the e-lico meta-miner. In *5th planning to learn workshop WS28 at ECAI* (pp. 18–19).

Olson, R. S., & Moore, J. H. (2016). Tpot: A tree-based pipeline optimization tool for automating machine learning. In *Workshop on automatic machine learning* (pp. 66–74).

Schadd, M. P. D., Winands, M. H. M., van den Herik, H. J., Chaslot, G. M. J. B., & Uiterwijk, J. W. H. M. (2008). Single-player Monte-Carlo tree search. In H. J. van den Herik, X. Xu, Z. Ma, & M. H. M. Winands (Eds.), *Computers and games*. Berlin: Springer.

Thornton, C., Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2013). Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA* (pp. 847–855).

Vanschoren, J., van Rijn, J. N., Bischl, B., & Torgo, L. (2013). OpenML: Networked science in machine learning. *SIGKDD explorations*, *15*(2), 49–60. https://doi.org/10.1145/2641190.2641198.

4 ML-Plan for Unlimited-Length Machine Learning Pipelines

Declaration of the specific contributions of the author The publication *ML-Plan for Unlimited-Length Machine Learning Pipelines* is based on the idea of the author. For the realization, the extension of ML-Plan to build machine learning pipelines of unlimited length was implemented. The text was mainly written by the author, where the description of the original ML-Plan was contributed by Felix Mohr. All the authors revised the entire paper.

ML-Plan for Unlimited-Length Machine Learning Pipelines

Marcel Wever MARCEL.WEVER@UPB.DE
Felix Mohr FELIX.MOHR@UPB.DE
Eyke Hüllermeier EYKE@UPB.DE
Heinz Nixdorf Institute, Paderborn University, Paderborn, Germany

Abstract

In automated machine learning (AutoML), the process of engineering machine learning applications with respect to a specific problem is (partially) automated. Various AutoML tools have already been introduced to provide out-of-the-box machine learning functionality. More specifically, by selecting machine learning algorithms and optimizing their hyperparameters, these tools produce a machine learning pipeline tailored to the problem at hand. Except for TPOT, all of these tools restrict the maximum number of processing steps of such a pipeline. However, as TPOT follows an evolutionary approach, it suffers from performance issues when dealing with larger datasets. In this paper, we present an alternative approach leveraging a hierarchical planning to configure machine learning pipelines that are unlimited in length. We evaluate our approach and find its performance to be competitive with other AutoML tools, including TPOT.

Keywords: automated machine learning, complex pipelines, hierarchical planning

1. Introduction

The demand for machine learning functionality is increasing quite rapidly these days, not least because of recent impressive successes in practical applications. Since users in different application domains are normally not machine learning experts, a suitable support in terms of tools that are easy to use is required. Ideally, (nearly) the whole process including inducing models from data, data preprocessing, the choice of a model class, the training and evaluation of a prediction, etc. would be automated (Lloyd et al., 2014). This has triggered the field of *automated machine learning* (AutoML), which has developed into an important branch of machine learning research in the last couple of years.

Various state-of-the-art AutoML tools (Thornton et al., 2013; Komer et al., 2014; Feurer et al., 2015; Olson and Moore, 2016; de Sá et al., 2017) have shown impressive results in selecting machine learning algorithms and optimizing their hyperparameters to form a machine learning pipeline (ML pipeline). These approaches can be divided into two main categories. First, the AutoML problem is designed as an optimization problem with a fixed number of decision variables, which then is solved via standard (Bayesian) optimization tools such as SMAC (Hutter et al., 2011). Typically, these approaches, such as auto-sklearn and Auto-WEKA, have one variable for a pre-processing algorithm, one variable for the learning algorithm, and one variable for each parameter of each algorithm. However, a relaxation of the length restriction is not straight-forward. Approaches of the second category organize the AutoML search space in terms of a formal grammar. The advantage of this formalism is that it naturally allows for recursive structures, thereby supporting more flexible ML

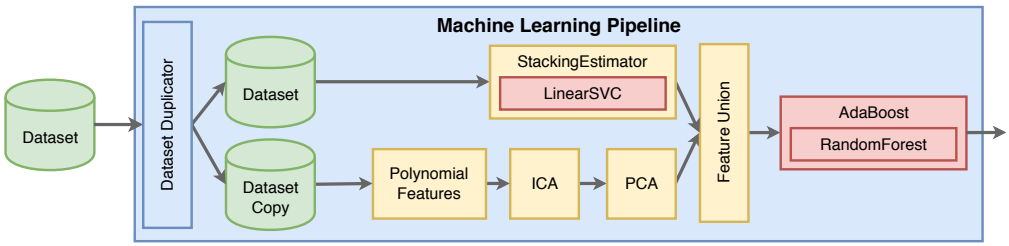


Figure 1: Possible workflow of an ML pipeline

pipelines. While de Sá et al. (2017) propose a grammar-based approach and allow for multiple preprocessing steps, the maximum length of an ML pipeline is still fixed. To the best of our knowledge, TPOT as originally introduced by Olson et al. (2016) is the only AutoML tool with no upper bound on the length of a pipeline. However, while the more flexible pipelines as composed by TPOT often perform particularly well, we also observed severe scalability issues of TPOT for more complex datasets (e.g., large number of features and/or instances).

In this paper, we present an alternative approach for composing ML pipelines that are unlimited in length, using a grammar-based formalism. More specifically, we show how ML-Plan (Mohr et al., 2018), an AutoML tool based on an AI planning technique called hierarchical task networks, can be extended for this purpose. In our evaluation, we find that our approach performs competitive to TPOT and furthermore improves on the scalability issues.

2. AutoML and Hierarchical Planning

AutoML seeks to automatically *compose* and *parametrize* machine learning algorithms into ML pipelines with the goal to optimize a given metric, e.g., predictive accuracy. Figure 1 shows an example of such a pipeline, which also illustrates that pipelines are by no means only sequences of atomic algorithms but can have parallel flows and nested structures as well. For example, StackingEstimator has a LinearSVC and AdaBoost uses RandomForest as a base learner. Especially, when it comes to meta methods, such recursive definitions of algorithms incorporating a base learner (and other components) constitute a very frequent pattern. In general, complete pipelines can be viewed as a hierarchical composition structure as in the example shown on the right-hand side of Figure 2. Furthermore, machine learning algorithms usually have hyperparameters that need to be chosen specifically for this algorithm. Thus, a hierarchical view of a machine learning pipeline represents its natural structure particularly well.

One interesting approach for creating such structure is hierarchical planning, a concept from the field of AI planning (Ghallab et al., 2004). In essence, it is about iteratively breaking down an initially given complex task into new sub-tasks, which may also be complex or simple (no further refinement required). Complex tasks are recursively decomposed until only simple tasks remain. This procedure is comparable, for example, to deriving a sentence from a context-free grammar. In that sense, complex tasks correspond to non-terminals and

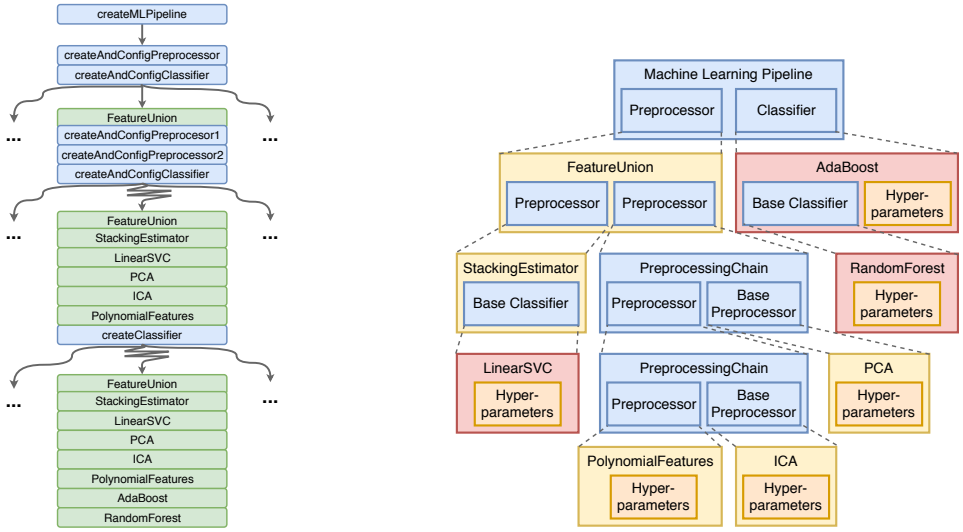


Figure 2: Excerpt of the search graph (left) listing the tasks of the planning problem at each node and its hierarchical representation in the form of an ML pipeline (right).

simple tasks match terminal symbols. An example is shown on the left-hand side in Figure 2 where complex tasks are displayed in blue and simple tasks in green.

We are aware of four approaches to AutoML using hierarchical planning or related techniques. The first approach is related to optimization within the RapidMiner framework based on hierarchical task networks (HTN) (Nguyen et al., 2014; Kietz et al., 2012). They conduct a beam search (hill-climbing in the most extreme case), where the beam is selected based on a *ranking* of alternative choices obtained from a meta-learning module, which compares the current dataset with previous ones and choices taken back then. The most recent representative of this line of research is Meta-Miner (Nguyen et al., 2014). While these approaches do not execute candidates during search to observe their performance, approaches of extensive evaluation is presented in RECIPE (de Sá et al., 2017) and TPOT (Olson and Moore, 2016). TPOT and RECIPE create pipelines using a grammar-based genetic programming algorithm; the pipeline candidates are evaluated in the course of computing their fitness. Last, ML-Plan (Mohr et al., 2018) recognizes the value of executing pipelines during search, but also observes that the extensive evaluation conducted in TPOT and RECIPE is infeasible for larger datasets. It reduces the number of evaluations by only considering candidates obtained from completions of currently best candidates. Like Meta-Learner, it is based on HTN planning.

Of course, other AutoML solutions such as Auto-WEKA or auto-sklearn can be extended to multiple pre-processing steps. It is clear that one can flatten any hierarchical structure into a vector as long as the allowed structures are bound in length. However, it is rather unclear how to represent ML pipelines that are unlimited in length.

In this paper, we extend ML-Plan to deal with unlimited-length ML pipelines, which is our approach for pipelines including a single preprocessor and a learner. In the following section, we give a brief overview of ML-Plan and explain how it is extended.

3. ML-Plan for Unlimited-Length ML Pipelines

3.1. ML-Plan

As briefly sketched above, ML-Plan is a hierarchical planner designed for AutoML problems (Mohr et al., 2018). Standard hierarchical planners such as SHOP2 (Au et al., 2011) lack some fundamental requirements of AutoML, e.g., to evaluate candidate solutions during search, which was a main motivation for developing ML-Plan.

The search technique adopted by ML-Plan is a best-first graph search. Like other planners, ML-Plan reduces the planning problem to the problem of finding a path to an optimal goal node in a graph. The graph is represented by a distinguished root node, a function for generating successors of a given node, and a function for testing whether a node is a goal node. In a nutshell, the best-first search algorithm assumes that every node in the explored part of the graph is associated with a score (in $\mathbb{R}$), and, in each iteration of the search, the leaf node with the lowest score is chosen for expansion. In contrast to A*, there is no assumption that the node score can be computed from edge costs; instead, there is just a function that returns the score without being related to the score of other nodes.

The node evaluation in ML-Plan is based on random path completion as also used in Monte Carlo Tree Search (Browne et al., 2012). To obtain the evaluation of a node, this strategy draws a fixed number of path completions, builds the corresponding pipelines and evaluates them against a validation set. The score assigned to the node is the *minimal* score that was observed over these validations in order to estimate the best solution that can be obtained when following paths under the node.

Intuitively, ML-Plan formalizes the HTN problem in a way that the resulting search graph is split into an algorithm selection region (upper region) and an algorithm configuration region (lower region). The main motivation for this strategy lies in the node evaluation we want to apply, which is based on random completions. Since algorithm selections usually constitute a much more significant change to the performance of a pipeline than parameter settings, we consider all solutions under a node that has all algorithms fixed as a kind of neighborhood, and random samples drawn in that sub-region yield more reliable estimates.

With the idea of a two-phased search graph in mind, the HTN definition of ML-Plan is as follows[1]. The initial task `createClassifier` can be broken down into a chain of the three tasks `createRawPP`, `setupClassifier`, `refinePP`. The first task is meant to choose the algorithms used for pre-processing *without* parametrizing them, the second task is meant to choose and configure the multi-label classifier, and the third step parametrizes the previously chosen pre-processors. The second task `setupClassifier` can, for each classifier, be decomposed into two sub-tasks. First, `<classifier>:create` is a simple task indicating the creation of a new classifier of the respective class, e.g. `RandomForestClassifier:create`. Second, `<classifier>:configure` is a complex task meant to configure the parameters of the classifier.

As an additional remark, ML-Plan comes with a built-in strategy to prevent over-fitting. This strategy apportions the assigned timeout for the whole search process among two phases. The first phase covers the actual search in the space. The second phase takes a collection of identified solutions and selects the one that minimizes the estimated general-

1. Since we have not formally introduced HTN planning, we describe the problem definition in a rather intuitive way. The formal definition can be found in the implementation published with this paper

ization error. Roughly speaking, the collection used for selection in phase 2 corresponds to the k best candidates and k random candidates that are not significantly worse than the best candidate. The time allocated at time step t for the second phase is flexible and corresponds to the accumulated time that was required in phase 1 to evaluate the classifiers that would be chosen at time step t for the selection process.

3.2. Extending ML-Plan for Unlimited-Length Pipelines

In order to compose ML pipelines with more complex pre-processing workflows, TPOT allows for chaining pre-processing algorithms and to fuse datasets taking the union of the respective features. Extending ML-Plan to operate on the same space of solution candidates, we add two complex and one simple tasks. First, we add a complex task `createFeatureUnion`, which may be resolved to the simple task `FeatureUnion` and two complex tasks to create the preceding pre-processing steps. Second, we add a complex task `createFeaturePreprocessorChainItem`, which may be resolved to one complex task for selecting a concrete pre-processing algorithm (e.g., PCA, Polynomial Features, etc.) and another complex task for creating any kind of pre-processing. In particular, the latter includes the building blocks for feature union and chaining pre-processing algorithms. We refer to this extension as ML-Plan-UL.

4. Experimental Evaluation

In our experimental evaluation, we focus on comparing ML-Plan-UL to TPOT, which both operate on the same solution space; besides, to the best of our knowledge, TPOT is the only AutoML tool supporting ML pipelines of unlimited length. As additional references, we also evaluate auto-sklearn and Auto-WEKA. All the tools are evaluated on a selection of 20 data sets from the `openml.org` (Vanschoren et al., 2013) repository, all of which were previously used to evaluate AutoML approaches (Thornton et al., 2013; Feurer et al., 2015).

The implementation of ML-Plan-UL, the evaluation code that produced the results shown in this section, and the used datasets are publicly available to assure reproducibility[2].

Results were obtained by carrying out 20 runs on each dataset with a timeout of one day per run. The timeout for the internal evaluation of a single solution was set to 20m for all the candidates. In each run, we used 70% of a stratified split of the data for the respective AutoML tool and 30% for testing. Note that we used the *same* splits for *all* tools. The computations were executed on 100 Linux machines in parallel, each of them equipped with 8 cores (Intel Xeon E5-2670, 2.6Ghz) and 32GB memory; every experiment used one machine at a time. The accumulated time of all experiments was over 300k CPU hours (over 34 CPU years).

Runs that did not adhere to the time or resource limitations (plus a tolerance threshold) were canceled without considering their results. That is, algorithms were canceled if they did not terminate within 110% of the predefined timeout or if they consumed more resources (memory or CPU) than allowed.

2. https://github.com/fmohr/ML-Plan

Dataset	abalone	amazon	car	cifar10small	cifar10	convex	credit-g	dexter	dorothea	gisette
ML-Plan-UL	**73.11**	**18.28**	0.78	57.06	-	14.17	24.13	**5.56**	**5.85**	2.23
TPOT	73.35	-	0.40	-	-	-	**23.53**	-	-	-
auto-sklearn	-	22.0 •	1.64 •	**55.08** ∘	-	**12.16** ∘	-	7.30 •	6.04	**2.22**
Auto-WEKA	73.46	50.28 •	**0.24** ∘	62.09 •	**64.06**	45.7 •	26.44 •	9.82 •	11.01 •	4.18 •

Dataset	krvskp	madelon	mnistrot	mnist	secom	semeion	shuttle	waveform	winequality	yeast
ML-Plan-UL	**0.65**	15.55	49.5	3.36	6.71	**5.51**	0.02	13.61	33.17	38.94
TPOT	1.08	15.26	-	-	**6.50**	6.06	**0.01**	13.1	**32.66**	**38.75**
auto-sklearn	0.74	**14.7**	**43.49** ∘	**2.90**	6.57	6.29	0.02	13.6	36.25 •	39.27
Auto-WEKA	4.02 •	20.34 •	74.3 •	5.39 •	6.60	8.42 •	0.13 •	**13.05**	33.58	39.8

Table 1: Mean 0/1-losses [in %] for a timeout of one day

As TPOT did not return even a single result for any of the datasets within the timeout, we configured TPOT to log intermediate solutions and considered the most recent one of these to compute the respective value in the results table.

The results of the experiments are summarized in Table 1, with best performances highlighted in bold. To determine significance for differences in performance, we applied a t-test with $p = 0.05$. A significant improvement of ML-Plan-UL over another tool is indicated by • and a significant degradation is highlighted by ∘.

The overall image is that ML-Plan-UL performs competitive to TPOT as best performances vary among the datasets and there are neither significant improvements nor degradations. While ML-Plan-UL does not return any result for *cifar10* only (due to exceeding memory usage), TPOT does not manage to return anything (not even an intermediate solution) for nearly half of the datasets. Thus, despite the substantially larger search space ML-Plan-UL manages to find feasible solutions even for larger datasets as compared to TPOT. Furthermore, ML-Plan-UL also performs particularly well compared to auto-sklearn and Auto-WEKA. In comparison to auto-sklearn, we observe 4 significant improvements and 3 significant degradations. For the reference Auto-WEKA, we observe 13 significant improvements and only a single degradation.

5. Conclusion

We have presented ML-Plan-UL as an extension of ML-Plan to deal with ML pipelines of unlimited length, i.e., allowing for more complex pre-processing worklflows. To this end, we slightly adapted the search space description by additional tasks that allow for sequential and tree-shaped pre-processing workflows. In our experimental evaluation, we have shown that ML-Plan-UL performs competitive to TPOT and, in contrast to the latter, even manages to return solutions for larger datasets.

Acknowledgement

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Center "On-The-Fly Computing" (SFB 901).

References

Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dana S. Nau, Dan Wu, and Fusun Yaman. SHOP2: an HTN planning system. *CoRR*, abs/1106.4869, 2011. URL http://arxiv.org/abs/1106.4869.

Cameron Browne, Edward Jack Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(1):1–43, 2012. doi: 10.1109/TCIAIG.2012.2186810. URL https://doi.org/10.1109/TCIAIG.2012.2186810.

Alex Guimarães Cardoso de Sá, Walter José G. S. Pinto, Luiz Otávio Vilas Boas Oliveira, and Gisele L. Pappa. RECIPE: A grammar-based framework for automatically evolving classification pipelines. In *Genetic Programming - 20th European Conference, EuroGP 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings*, pages 246–261, 2017. doi: 10.1007/978-3-319-55696-3_16. URL https://doi.org/10.1007/978-3-319-55696-3_16.

Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Proc. NIPS 2015, Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.

Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Elsevier, 2004.

Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. *LION*, 5:507–523, 2011.

Jörg-Uwe Kietz, Floarea Serban, Abraham Bernstein, and Simon Fischer. Designing KDD-workflows via HTN-planning. In *ECAI 2012, 20th European Conference on Artificial Intelligence*, pages 1011–1012, 2012. doi: 10.3233/978-1-61499-098-7-1011. URL https://doi.org/10.3233/978-1-61499-098-7-1011.

Brent Komer, James Bergstra, and Chris Eliasmith. Hyperopt-sklearn: Automatic hyperparameter configuration for scikit-learn. In *ICML Workshop on AutoML*, 2014.

James Robert Lloyd, David K. Duvenaud, Roger B. Grosse, Joshua B. Tenenbaum, and Zoubin Ghahramani. Automatic construction and natural-language description of nonparametric regression models. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, Québec City, Québec, Canada*, pages 1242–1250, 2014.

Felix Mohr, Marcel Wever, and Eyke Hüllermeier. ML-Plan: Automated machine learning via hierarchical planning. *Machine Learning*, Jul 2018.

P. Nguyen, Melanie Hilario, and Alexandros Kalousis. Using meta-mining to support data mining workflow planning and optimization. *J. Artif. Intell. Res.*, 51:605–644, 2014. doi: 10.1613/jair.4377. URL https://doi.org/10.1613/jair.4377.

Randal S. Olson and Jason H. Moore. TPOT: A tree-based pipeline optimization tool for automating machine learning. In *Workshop on Automatic Machine Learning*, pages 66–74, 2016.

Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the 2016 Genetic and Evolutionary Computation Conference, Denver, CO, USA, July 20 - 24, 2016*, pages 485–492, 2016. doi: 10.1145/2908812.2908918. URL http://doi.acm.org/10.1145/2908812.2908918.

Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA*, pages 847–855, 2013.

Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. OpenML: Networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60, 2013. doi: 10.1145/2641190.2641198. URL http://doi.acm.org/10.1145/2641190.2641198.

5 Automating Multi-Label Classification Extending ML-Plan

Declaration of the specific contributions of the author The publication *Automating Multi-Label Classification Extending ML-Plan* is inspired by the idea of Eyke Hüllermeier to extend ML-Plan to the multi-label classification domain. The realization and the writing of the paper were mainly done by the author. Subsequently, the paper was refined by all the authors.

6th ICML Workshop on Automated Machine Learning (2019)

Automating Multi-Label Classification Extending ML-Plan

Marcel Wever marcel.wever@upb.de
Felix Mohr felix.mohr@upb.de
Alexander Tornede alexander.tornede@upb.de
Eyke Hüllermeier eyke@upb.de
Heinz Nixdorf Institut, Paderborn University, Paderborn, Germany

Abstract

Existing tools for automated machine learning, such as Auto-WEKA, TPOT, auto-sklearn, and more recently ML-Plan, have shown impressive results for the tasks of single-label classification and regression. Yet, there is only little work on other types of machine learning problems so far. In particular, there is almost no work on automating the engineering of machine learning solutions for multi-label classification (MLC). We show how the scope of ML-Plan, an AutoML-tool for multi-class classification, can be extended towards MLC using MEKA, which is a multi-label extension of the well-known Java library WEKA. The resulting approach recursively refines MEKA's multi-label classifiers, nesting other multi-label classifiers for meta algorithms and single-label classifiers provided by WEKA as base learners. In our evaluation, we find that the proposed approach yields strong results and performs significantly better than a set of baselines we compare with.

1. Introduction

In recent years, the field of AutoML has made significant progress in developing techniques for automating the task of model selection and hyperparameter tuning. State-of-the-art AutoML tools (Thornton et al., 2013; Komer et al., 2014; Feurer et al., 2015; Mohr et al., 2018b) have shown impressive results for binary and multinomial classification problems. We refer to this type of problems as single-label classification (SLC) in the following.

However, other learning problems, including multi-label classification (MLC), have received much less attention so far. In MLC, instead of predicting only a single class label for an instance, an entire subset of "relevant" labels is predicted. Learning algorithms for MLC have been designed by either adapting the learning algorithm itself or by reducing the original MLC problem to (multiple instances of) the SLC setting. The latter can be considered as a meta-learning technique with a single-label classifier as a base learner.

From an AutoML perspective, automating the configuration of a multi-label classifier is especially challenging, as these reduction techniques introduce deeper hierarchical structures. More specifically, while the configuration of a multi-label classifier's base learner is equivalent to the previous AutoML task for SLC, the meta-strategies for the multi-label classifiers themselves create another level of the hierarchy. The effect on the complexity of the search space is especially strong, because the evaluations are even more expensive.

In this paper, we propose the AutoML tool ML^2-Plan (Multi-Label ML-Plan) to configure multi-label classifiers based on ML-Plan. The latter provides a suitable basis to start from, especially due to its ability to model hierarchical dependencies by means of techniques from hierarchical task network (HTN) planning (Georgievski and Aiello, 2015). Besides,

ML-Plan has already been applied to deeper recursive structures in previous work (Wever et al., 2018b). Apart from the work by de Sá et al. (2017, 2018), which uses evolutionary algorithms, we are not aware of previous work on automated multi-label classification.

We compare ML^2-Plan to a random search, a genetic algorithm (de Sá et al., 2017), and a grammar-based genetic programming approach (de Sá et al., 2018). Empirically, we show that our approach performs particularly well and significantly outperforms the baselines.

2. Multi-Label Classification

In contrast to conventional (single-label) classification, the setting of *multi-label classification* (MLC) allows an instance to belong to several classes simultaneously, i.e., to be assigned several labels at the same time. For example, a single image could be tagged simultaneously with labels `Sun` and `Beach` and `Sea`.

More formally, let $\mathcal{X}$ denote an instance space, and let $\mathcal{L} = \{\lambda_1, \ldots, \lambda_m\}$ be a finite set of class labels. We assume that an instance $\boldsymbol{x} \in \mathcal{X}$ is (non-deterministically) associated with a subset of labels $L \in 2^{\mathcal{L}}$; this subset is often called the set of relevant labels, while the complement $\mathcal{L} \setminus L$ is considered as irrelevant for $\boldsymbol{x}$. We identify a set L of relevant labels with a binary vector $\boldsymbol{y} = (y_1, \ldots, y_m)$, in which $y_i = 1$ iff $\lambda_i \in L$. By $\mathcal{Y} = \{0, 1\}^m$ we denote the set of possible labelings.

In general, a multi-label classifier $\boldsymbol{h}$ is a mapping $\mathcal{X} \to \mathcal{Y}$. For a given instance $\boldsymbol{x} \in \mathcal{X}$, it returns a prediction in the form of a vector $\boldsymbol{h}(\boldsymbol{x}) = \big(h_1(\boldsymbol{x}), h_2(\boldsymbol{x}), \ldots, h_m(\boldsymbol{x})\big)$. The problem of MLC can be stated as follows: Given training data in the form of a finite set of observations $\big\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\big\}_{i=1}^{N} \subset \mathcal{X} \times \mathcal{Y}$, the goal is to learn a classifier $\boldsymbol{h} : \mathcal{X} \to \mathcal{Y}$ that generalizes well beyond these observations in the sense of minimizing the risk with respect to a specific loss function. Various loss functions are commonly used in MLC. Let $\mathcal{D}_{\text{test}} = (\mathcal{X}_{\text{test}}, \mathcal{Y}_{\text{test}}) \subset \mathcal{X}^S \times \mathcal{Y}^S$ be a test set of size S, where the ith entry $\boldsymbol{y}_i = (y_{i1}, \ldots, y_{im}) \in \mathcal{Y}_{\text{test}}$ represents the labeling of the ith instance $\boldsymbol{x}_i \in \mathcal{X}_{\text{test}}$. Further, let $H \subset \mathcal{Y}^S$ with the ith entry given by $h(\boldsymbol{x}_i)$. Then, the subset 0/1 loss (exact match) is defined as[1]

$$L_{0/1}(\mathcal{Y}_{\text{test}}, H) = \frac{1}{S} \sum_{i=1}^{S} [\![\boldsymbol{y}_i \neq \boldsymbol{h}(\boldsymbol{x_i})]\!] \ ,$$

and the Hamming loss as

$$L_H(\mathcal{Y}_{\text{test}}, H) = \frac{1}{S} \sum_{i=1}^{S} \frac{1}{m} \sum_{j=1}^{m} [\![y_{ij} \neq h_j(\boldsymbol{x_i})]\!] \ .$$

In slightly different tasks, such as ranking and probability estimation, the prediction of a classifier is not restricted to binary vectors. Instead, a hypothesis $\boldsymbol{h}$ is a mapping $\mathcal{X} \to \mathbb{R}^m$, which assigns scores to labels. Corresponding predictions also require other loss functions. An example is the rank loss, which compares a ground-truth labeling with a predicted ranking of the labels and counts the number of incorrectly ordered label pairs:

$$L_R(\mathcal{Y}_{\text{test}}, H) = \frac{1}{S} \sum_{i=1}^{S} \sum_{(j,j'):y_{ij} > y_{ij'}} \left(\frac{[\![h_j(\boldsymbol{x_i}) < h_{j'}(\boldsymbol{x_i})]\!]}{|\{(j, j') \mid y_{ij} > y_{ij'}\}|} \right), 1 \leq j, j' \leq m$$

1. $[\![\cdot]\!]$ is the indicator function.

Complementary to *instance-wise* losses, which are defined for (and averaged over) instances, losses are sometimes considered *label-wise*. An example is the macro-F1 measure:

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

A linear combination of the four measures defined above is proposed by de Sá et al. (2017, 2018), who use it as an "objective function" in the respective AutoML tools:

$$\mathrm{L}_{\text{Fit}}(\mathcal{Y}_{\text{test}}, H) = \frac{1}{4}\Big(L_{0/1}(\mathcal{Y}_{\text{test}}, H) + L_H(\mathcal{Y}_{\text{test}}, H) + (1 - F(\mathcal{Y}_{\text{test}}, H)) + L_R(\mathcal{Y}_{\text{test}}, H)\Big).$$

As it combines different types of losses with different interpretations, this metric is debatable and difficult to interpret. Yet, in spite of our reservations, we will use it under the notion of "fitness loss" in our experimental study as well, mainly to reduce confounding factors and to ensure a fair comparison.

At first sight, MLC problems can be solved in a quite straightforward way, namely through decomposition into several binary classification problems: One binary classifier is trained for each label and used to predict whether, for a given query instance, this label is relevant or not. This approach is known as *binary relevance* (BR) learning. However, BR has been criticized for ignoring important information hidden in the label space, namely information about the interdependencies between the labels. Since the presence or absence of the different class labels has to be predicted *simultaneously*, it is arguably important to exploit any such dependencies. Correspondingly, a large repertoire of methods for MLC beyond BR has been proposed in the recent years. Most of these methods seek to improve predictive accuracy by exploiting label dependencies in one way or the other. We refer to Zhang and Zhou (2014) for an up-to-date survey on MLC algorithms.

3. A Multi-Label Extension of ML-Plan

As illustrated in Fig. 1 (left), multi-label classifiers may nest several classifiers in a recursive manner. Additionally, each of the classifiers has a set of parameters that need to be configured. While flattening these recursive structures to a single vector comprised of decision variables for the algorithm choices and a variable for each parameter that may occur for a specific layer, as done by Auto-WEKA and auto-sklearn, may work in principle, this approach would require many constraints to make sure that only relevant variables are considered. An arguably more natural way of representing these hierarchical dependencies is hierarchical task network (HTN) planning (Ghallab et al., 2004), or more specifically programmatic task network (PTN) planning (Mohr et al., 2018a) as incorporated in ML-Plan (Mohr et al., 2018b). Via HTN resp. PTN planning, the search space of possible algorithm choices and respective hyperparameters to be tuned is structured into *complex tasks* (blue), which are refined by *methods* to one or multiple complex tasks or *primitive tasks* (green). Intuitively, this formalism mimics a human expert who is tackling a (complex) problem by decomposing the original task into several sub-tasks until the resulting sub-tasks are (simple) primitive tasks.

Translated to AutoML for MLC, the initial task could be, for example, to do multi-label classification as shown on the right-hand side of Fig. 1. With the help of methods that are

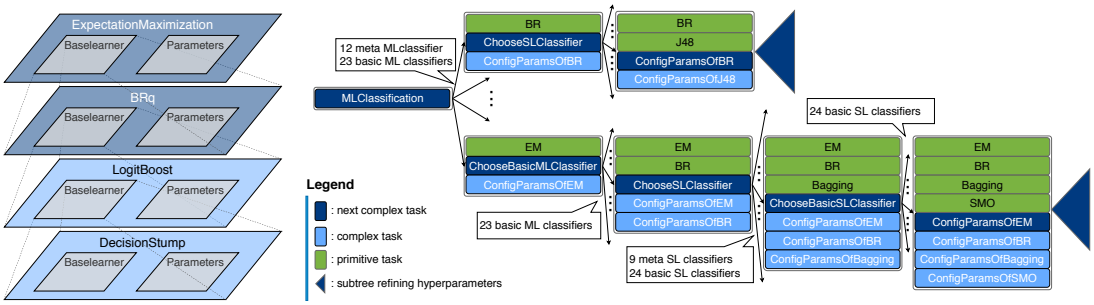


Figure 1: Visualization of the hierarchical structure of a machine learning pipeline (left) and an excerpt of the hierarchical planning search graph (right).

displayed in the form of arcs, this task can be refined by choosing a multi-label classifier. This can be either a meta multi-label classifier, e.g., Expectation Maximization (EM), or a basic one, e.g., Binary Relevance (BR). Depending on the decision, new decision-specific tasks arise, namely to choose base learners and to set the parameter values of the respective algorithm. Modeling the search space in this fashion yields a tree that can be used as a search graph for standard search algorithms. ML-Plan incorporates a Best-First search with random completions to complete partial specifications (decisions already made until a certain point) to fully specified classifiers that can be evaluated (using cross-validation or a holdout set). For more details on ML-Plan, we refer to (Mohr et al., 2018b) and (Wever et al., 2018a).

To derive ML^2-Plan from ML-Plan, we use the default configuration of ML-Plan and extend it in the following three ways:

- We extend the search space from SLC (WEKA) to MLC (MEKA+WEKA) but discarding preprocessors. This extension increases the size of the search space dramatically and yields in roughly 76,000 possible algorithm combinations (choice of main model and recursive base learner selections) to setup a multi-label classifier, compared to 234 as in the case of SLC. A breadth-first search to spawn all possible algorithm combinations, as it is done in ML-Plan, is thus unfeasible. Therefore, we adapt ML-Plan also to spawn only the first layer of algorithm choices, i.e., each multi-label classifier (basic and meta) is considered at least once as a main model. In contrast to (Wever et al., 2018a) we consider hyperparameter optimization as well. Hyperparameter optimization is done via a single decision for categorical and boolean parameters and by iteratively splitting the domain of numeric parameters into sub-intervals and refining those step-by-step until an interval size is reached that is considered atomic.

- Compared to evaluating single-label classifiers, the evaluations are much more expensive. Therefore, we introduced an early stopping criterion for the Monte Carlo cross-validation. After each iteration, we perform a significance test to check whether the currently considered candidate might be added to the pool of candidates for the selection phase, i.e., the difference of its performance and the best hitherto solution

Dataset	Auto-MEKA$_{GGP}$	GA-Auto-MLC	Random Search	ML^2-Plan
arts1	**0.5196±0.01** (1) ○	0.5509±0.01 (3)	0.5527±0.13 (4)	0.5508±0.04 (2)
bibtex	- (3)	- (3)	0.5822±0.02 (2) •	**0.4910±0.05** (1)
birds-fixed	0.3259±0.02 (3) •	0.3239±0.03 (2) •	0.3277±0.04 (4) •	**0.2962±0.03** (1)
business1	0.4711±0.00 (3) •	- (4)	0.4034±0.13 (2)	**0.3452±0.01** (1)
emotions	0.3643±0.02 (4) •	0.3356±0.02 (3) •	**0.2858±0.02** (1)	0.2944±0.02 (2)
enron-f	0.4750±0.00 (3) •	0.4793±0.01 (4) •	0.4540±0.03 (2) •	**0.3997±0.02** (1)
flags	0.4407±0.03 (3) •	0.4427±0.03 (4) •	0.3613±0.04 (2) •	**0.3303±0.02** (1)
genbase	0.0701±0.02 (2)	0.1044±0.02 (3) •	0.1511±0.15 (4) •	**0.0684±0.01** (1)
health1	0.5153±0.00 (4) •	**0.4499±0.01** (1)	0.5141±0.07 (3)	0.4564±0.03 (2)
llog-f	0.5054±0.01 (3) •	0.4981±0.01 (2) •	0.5183±0.02 (4) •	**0.4792±0.02** (1)
medical	0.2770±0.05 (3)	0.2648±0.01 (2) •	0.3167±0.10 (4) •	**0.2425±0.01** (1)
scene	0.1998±0.02 (2) •	0.2614±0.03 (4) •	0.2003±0.04 (3) •	**0.1746±0.01** (1)
science1	0.5495±0.00 (3)	**0.5318±0.00** (1) ○	0.6052±0.04 (4) •	0.5462±0.02 (2)
yeast	0.4495±0.01 (3) •	0.4773±0.01 (4) •	0.3935±0.05 (2)	**0.3632±0.01** (1)
average-rank	2.86	2.86	2.93	1.29
#best	1	2	1	10
sig (i/t/d)	12 / 1 / 0	9 / 2 / 1	13 / 0 / 1	- / - / -

Table 1: L_{Fit} (mean ± standard deviation) for the test data of 20 runs per dataset. The rank per dataset of the respective approach is enclosed in parentheses.

is not more than 3%. If case the hypothesis test fails, we abort the evaluation of the classifier and return the mean of the evaluated iterations so far.

- We adapt the internal evaluations part of ML-Plan to the MLC setting, incorporating L_{Fit} as the objective function instead of error rate for SLC, and creating train and test splits at random instead of class-stratified splits.

4. Experimental Evaluation

We evaluate ML^2-Plan as introduced in the previous section on various datasets and compare it to three baselines: a random search, GA-AutoMLC (de Sá et al., 2017), and Auto-MEKA$_{GGP}$ (de Sá et al., 2018). The random search evaluates candidates that are picked uniformly at random from the set of 76,000 possible nested classifiers and chooses values of the resulting parameters at random as well. Note that while ML^2-Plan, random search and Auto-MEKA$_{GGP}$ operate on the same search space, GA-Auto-MLC is based on a much simpler space (see (de Sá et al., 2017)). All approaches underly the same timeout and resource limitations and use the implementations of the basic loss functions provided by MEKA. The implementation of ML^2-Plan is publicly available[2].

Results were obtained by carrying out 20 runs on 14 datasets with a timeout of 1 hour for each run and a timeout of 10 minutes for evaluating a single candidate. The datasets stem from the MULAN project website[3]. In each run, we used 70% of a randomized split of the data for learning (search) and 30% for testing. We used the *same* splits for all candidates, i.e., for each split, we ran ML^2-Plan as well as each baseline exactly once. The significance

2. Implementation of ML2-Plan: `https://github.com/fmohr/AILIbs`,
Dataset splits, seed project, and ReadMe: `https://github.com/mwever/ML2PlanAtAutoML2019`
3. `http://mulan.sourceforge.net/datasets-mlc.html`

of an improvement (marked by •) resp. degradation (◦) per dataset was determined using a Wilcoxon signed-rank test (Wilcoxon, 1945) with a threshold for the p-value of 0.05.

The experiments were run on up to 220 Linux machines in parallel, each of which with a resource limitation of 8 cores (Intel Xeon E5-2670, 2.6Ghz) and 32GB RAM. Runs that did not adhere to the time or resource limitations (plus a tolerance threshold of 10%) were canceled without considering their results for the respective approach.

A summary of the results is given in Table 1. While the upper part of the table describes the observed values for L_{fit} on the test data, the bottom part gives a summary regarding the average rank and statistics about number of times an approach has been the best solver. On the last row of the table, it is counted how many times ML^2-Plan achieved significantly improved, significantly degraded or equally performing results compared to a baseline.

The general impression is that ML^2-Plan performs clearly superior to the baselines and for the majority of datasets manages to return significantly better solutions compared to the competitor tools and the random search. Nevertheless, ML^2-Plan does not win in every case and it must admit defeat on arts against Auto-MEKA$_{\text{GGP}}$, on `emotions` against the random search, and on `health1` and `science1` against GA-Auto-MLC.

Furthermore, it was surprising to see that, according to the average rank statistic, Auto-MEKA$_{\text{GGP}}$ and GA-Auto-MLC perform only slightly better than the random search baseline. This might be due to the relatively small timeout of 1 hour we gave each tool for a single run but note that on the contrary ML^2-Plan already manages to outperform the random search. Moreover, Auto-MEKA$_{\text{GGP}}$ did not return any result for the dataset `bibtex` and GA-Auto-MLCfor the datasets `bibtex` and `business1` which may also be due to the low time budget. Nevertheless, we will consider larger timeouts in future work to investigate the long-term behavior of the different approaches as well.

5. Conclusion

In this paper, we presented an AutoML approach to multi-label classification. ML^2-Plan builds on ML-Plan and combines hierarchical task network planning with a global best-first search as proposed by Mohr et al. (2018b). Compared with previous AutoML tools for single-label classification, ML^2-Plan has to deal with more deeply nested structures to automatically select and configure multi-label classifiers for a given dataset. In an experimental study, we showed that ML^2-Plan outperforms the baselines, including the only existing approaches to AutoML for multi-label classification (de Sá et al., 2018, 2017). Future work will be dedicated to improving scalability, e.g., by moving to a service-oriented architecture (Mohr et al., 2018d,c) or incorporating meta-learning techniques for warmstarting (Feurer et al., 2015), and to improving efficiency during search, e.g., by biasing the random completion towards in general more promising solutions. Finally, the wide spectrum of loss functions in MLC motivates a multi-objective optimization process that seeks for trade-offs between different (and potentially conflicting) performance metrics.

Acknowledgement

This work was supported by the German Research Foundation (DFG) within the Collaborative Research Center "On-The-Fly Computing" (SFB 901).

References

Alex Guimarães Cardoso de Sá, Gisele L. Pappa, and Alex Alves Freitas. Towards a method for automatically selecting and configuring multi-label classification algorithms. In *Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings*, pages 1125–1132, 2017.

Alex Guimarães Cardoso de Sá, Alex Alves Freitas, and Gisele L. Pappa. Automated selection and configuration of multi-label classification algorithms with grammar-based genetic programming. In *PPSN (2)*, volume 11102 of *Lecture Notes in Computer Science*, pages 308–320. Springer, 2018.

Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.

Ilche Georgievski and Marco Aiello. HTN planning: Overview, comparison, and beyond. *Artif. Intell.*, 222:124–156, 2015.

Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated planning - theory and practice*. Elsevier, 2004. ISBN 978-1-55860-856-6.

Brent Komer, James Bergstra, and Chris Eliasmith. Hyperopt-sklearn: automatic hyperparameter configuration for scikit-learn. In *ICML workshop on AutoML*, 2014.

Felix Mohr, Theodor Lettmann, Eyke Hüllermeier, and Marcel Wever. Programmatic task network planning. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling*. AAAI, 2018a.

Felix Mohr, Marcel Wever, and Eyke Hüllermeier. Ml-plan: Automated machine learning via hierarchical planning. *Machine Learning*, 107(8-10):1495–1515, 2018b.

Felix Mohr, Marcel Wever, and Eyke Hüllermeier. Automated machine learning service composition. *CoRR*, abs/1809.00486, 2018c.

Felix Mohr, Marcel Wever, Eyke Hüllermeier, and Amin Faez. (WIP) towards the automated composition of machine learning services. In *SCC*, pages 241–244. IEEE, 2018d.

Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA*, pages 847–855, 2013.

Marcel Wever, Felix Mohr, and Eyke Hüllermeier. Automated multi-label classification based on ml-plan. *CoRR*, abs/1811.04060, 2018a. URL `http://arxiv.org/abs/1811.04060`.

Marcel Wever, Felix Mohr, and Eyke Hüllermeier. *ML-Plan for Unlimited-Length Machine Learning Pipelines*. 2018b.

Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics bulletin*, 1(6): 80–83, 1945.

Min-Ling Zhang and Zhi-Hua Zhou. A review on multi-label learning algorithms. *IEEE Trans. Knowl. Data Eng.*, 26(8):1819–1837, 2014.

6

AutoML for Multi-Label Classification: Overview and Empirical Evaluation

Declaration of the specific contributions of the author The idea for this publication and the implementation can be attributed to the author. Eyke Hüllermeier helped to describe the configuration of multi-label classifiers. Alexander Tornede contributed the introductory paragraphs of the section about techniques reducing AutoML to hyper-parameter optimization and descriptions of Bayesian optimization, Hyperband, and BOHB. Felix Mohr helped with improving the descriptions of the benchmark. All authors repeatedly revised the paper.

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.

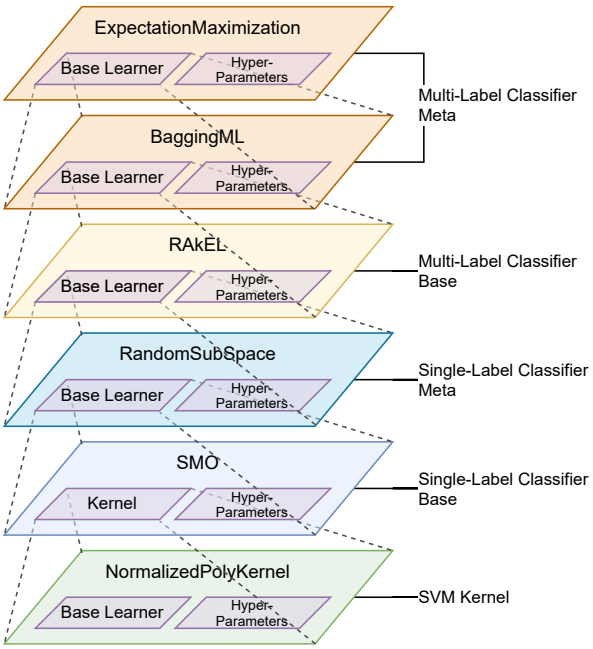


Fig. 1: Hierarchical representation of a multi-label classifier's structure being recursively configured with base learners and finally a kernel for the support vector machine (SMO, short for Sequential Minimal Optimization).

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 $\mathcal{A} := \{A^{(1)}, A^{(2)}, \ldots, A^{(n)}\}$ denote a set of algorithms and $\Lambda^{(1)}, \Lambda^{(2)}, \ldots, \Lambda^{(n)}$ 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

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

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 $\{1, \ldots, m\}$. 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),

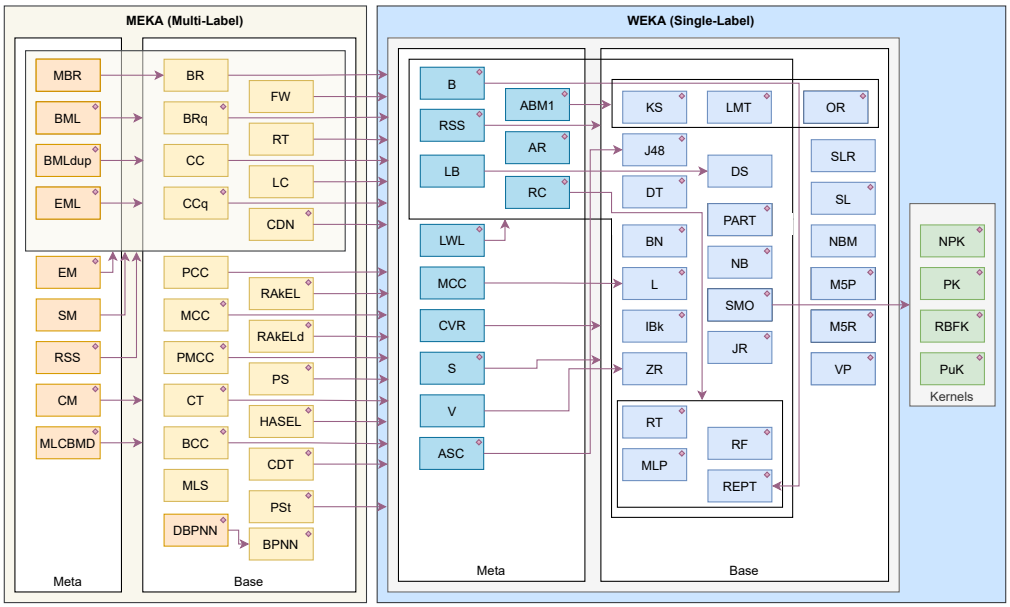


Fig. 2: Overview of the search space showing classification algorithms from MEKA for multi-label and WEKA for single-label classification. An arc pointing to a box frame means an arc to every classifier contained in this frame. Purple diamonds indicate whether the respective classifier exposes hyper-parameters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

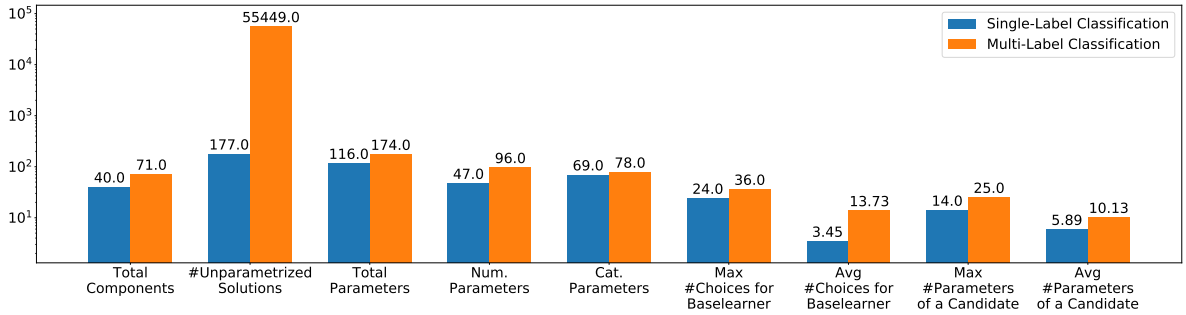


Fig. 3: Comparison of statistics regarding characteristics of the multi-label classification search space and the subsumed search space for single-label classification. Note that the there is a substantial increase in the number of unparameterized solution candidates, i.e., the number of distinct classifier 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

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

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.

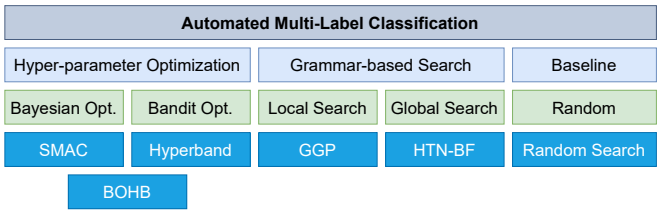


Fig. 4: Ontology showing the considered optimization techniques proposed for automating machine learning.

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]

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,

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 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.

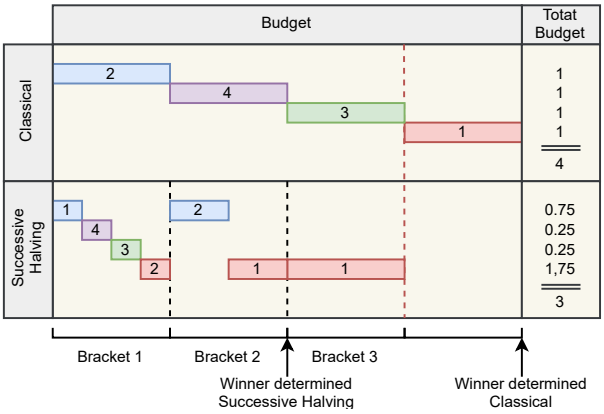


Fig. 5: Comparison of the classical approach (top) and successive halving (SH) (bottom) to identify the best performing configuration out of 4 candidates. Numbers within colored rectangles indicate the rank of a configuration. Within each bracket, the current set of configurations is evaluated on a portion 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.

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].

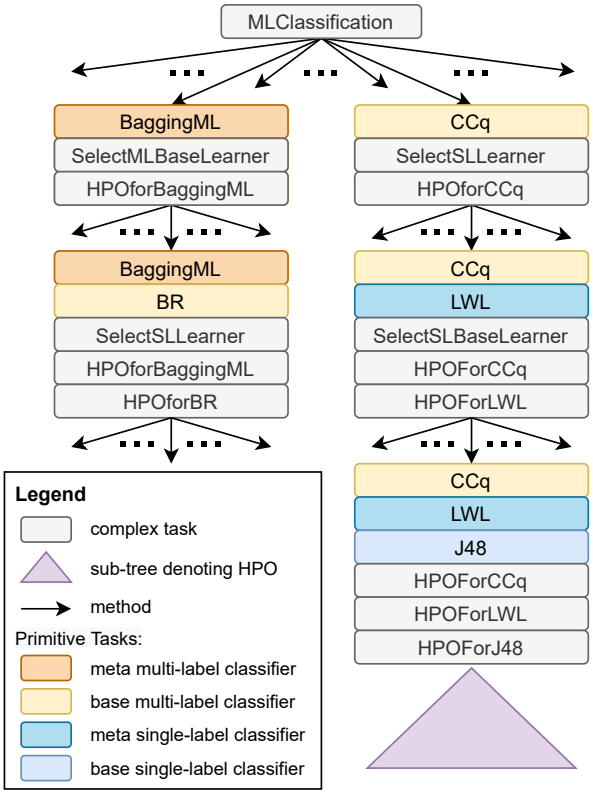


Fig. 6: Sketch of a search tree induced via HTN planning for automated multi-label classification. Primitive tasks are additionally distinguished by color according to their role within a multi-label classifier. Note that the indicated refinements are of exemplary character. Further options as well as 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$_{GGP}$ [5]. However, from a methodological point of view, nothing has been implemented in Auto-MEKA$_{GGP}$ 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,

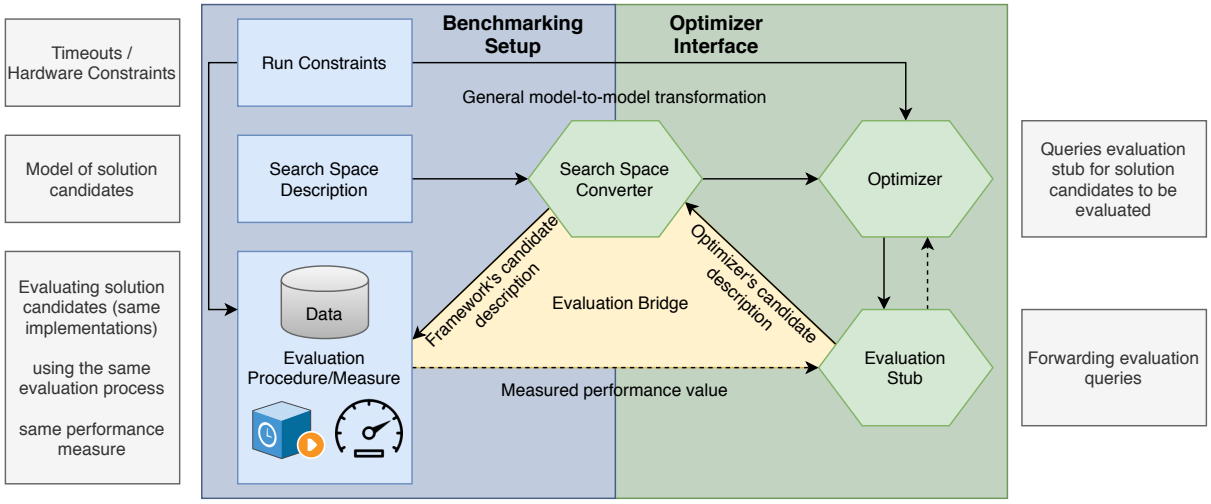


Fig. 7: Architecture of the benchmark for comparing different optimizers for the same run constraints, search space, and evaluation procedure. Blue parts are commonly used for all approaches, while green parts are specific to the respective optimizer, marshaling the description of candidate solutions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

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

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).

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

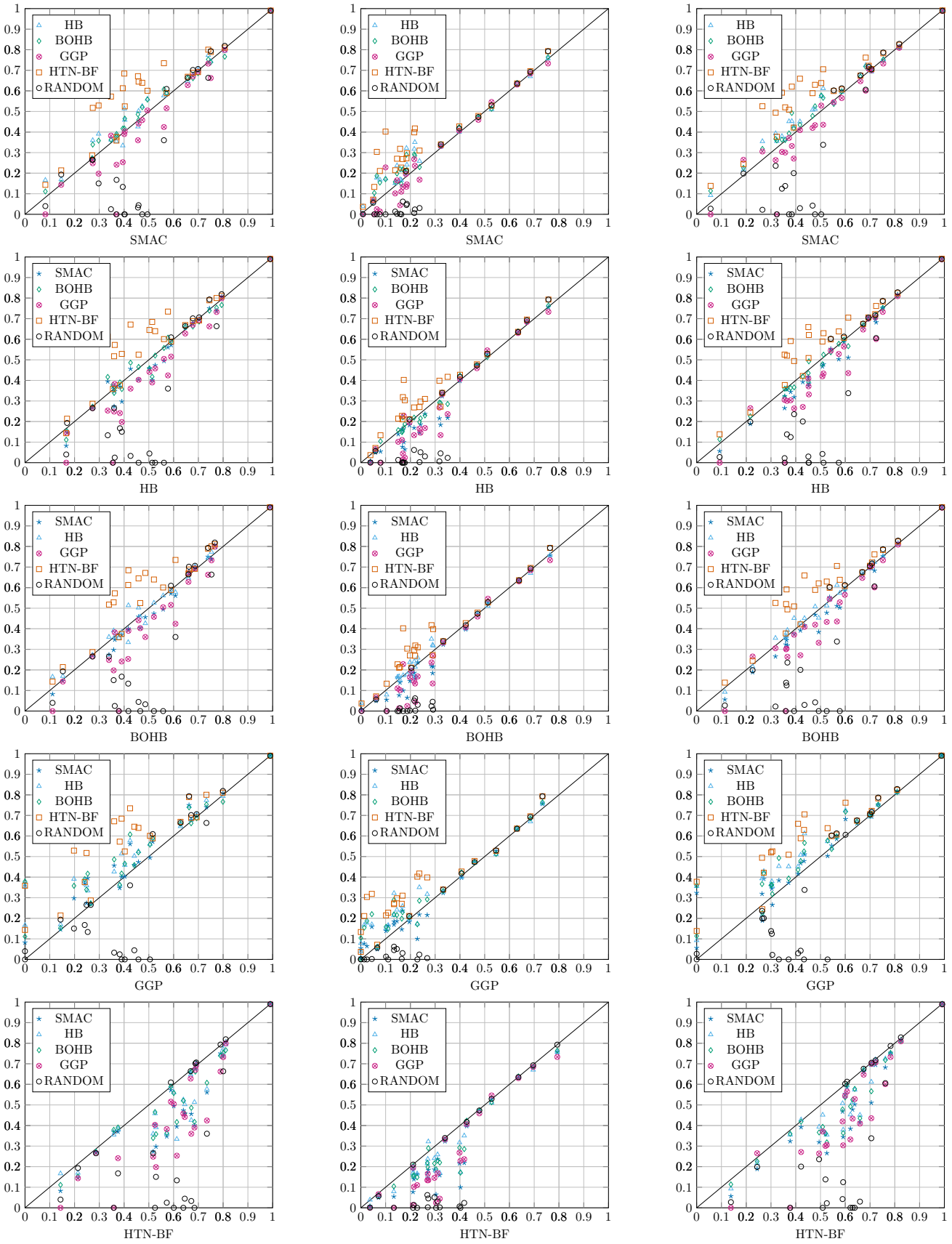


Fig. 8: Pair-wise comparison of one method (shown on the x-axis) against all other methods with respect to instance-wise F-Measure (left), label-wise F-Measure (center), and micro-averaged F-Measure (right).

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.

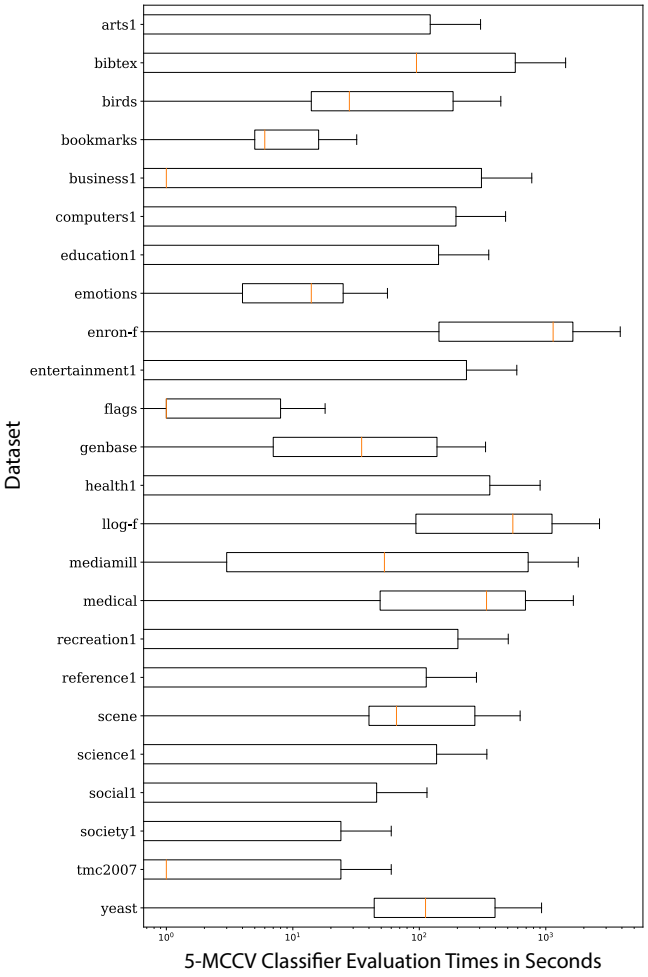


Fig. 9: Evaluation times of successful classifier evaluations.

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.

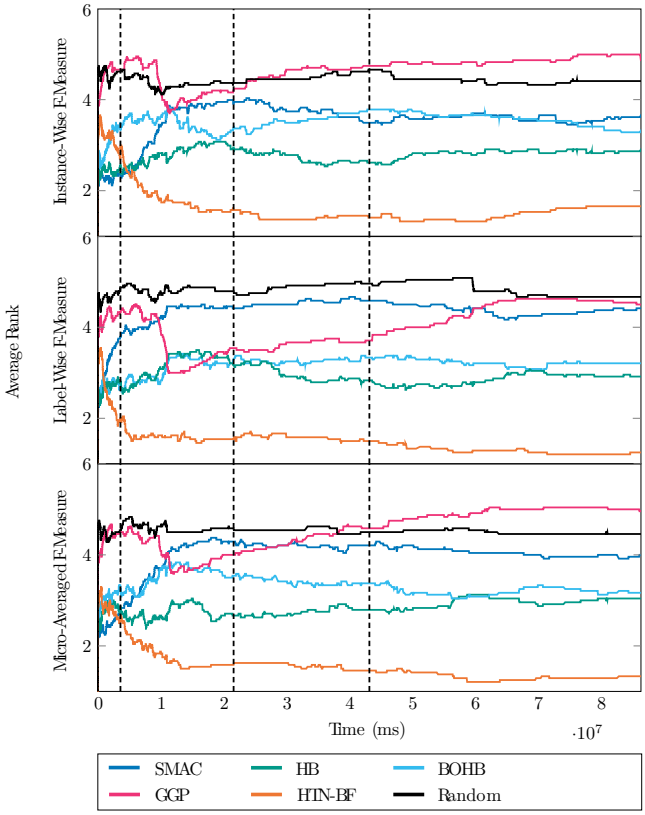


Fig. 10: Average ranks over time (in ms) for the three performance measures: instance-wise F-Measure (F_I), label-wise F-Measure (F_L), and micro-averaged F-Measure (F_μ).

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.

7

LiBRe: Label-Wise Selection of Base Learners in Binary Relevance for Multi-Label Classification

Declaration of the specific contributions of the author The general idea of this publication goes back to Felix Mohr, which was further refined, shaped, and targeted by the author. The implementation, as well as the conduction of the experiments, were done by the author. While the related work section was contributed by Felix Mohr, the remaining parts of the paper were initially written by the author and later revised by all authors.

LiBRe: Label-Wise Selection of Base Learners in Binary Relevance for Multi-label Classification

Marcel Wever[1( )], Alexander Tornede[1], Felix Mohr[2], and Eyke Hüllermeier[1]

[1] Heinz Nixdorf Institut, Paderborn University, Paderborn, Germany
{marcel.wever,alexander.tornede,eyke}@upb.de
[2] Universidad de La Sabana, Chia, Cundinamarca, Colombia
felix.mohr@unisabana.edu.co

Abstract. In multi-label classification (MLC), each instance is associated with a set of class labels, in contrast to standard classification, where an instance is assigned a single label. Binary relevance (BR) learning, which reduces a multi-label to a set of binary classification problems, one per label, is arguably the most straight-forward approach to MLC. In spite of its simplicity, BR proved to be competitive to more sophisticated MLC methods, and still achieves state-of-the-art performance for many loss functions. Somewhat surprisingly, the optimal choice of the base learner for tackling the binary classification problems has received very little attention so far. Taking advantage of the label independence assumption inherent to BR, we propose a label-wise base learner selection method optimizing label-wise macro averaged performance measures. In an extensive experimental evaluation, we find that or approach, called LiBRe, can significantly improve generalization performance.

Keywords: Multi-label classification · Algorithm selection · Binary relevance

1 Introduction

By relaxing the assumption of mutual exclusiveness of classes, the setting of *multi-label classification* (MLC) generalizes standard (binary or multinomial) classification—subsequently also referred to as single-label classification (SLC). MLC has received a lot of attention in the recent machine learning literature [23, 29]. The motivation for allowing an instance to be associated with several classes simultaneously originated in the field of text categorization [19], but nowadays multi-label methods are used in applications as diverse as image processing [4,26] and video annotation [14], music classification [18], and bioinformatics [2].

Common approaches to MLC either adapt existing algorithms (*algorithm adaptation*) to the MLC setting, e.g., the structure and the training procedure for neural networks, or reduce the original MLC problem to one or multiple SLC problems (*problem transformation*). The most intuitive and straight-forward

M. R. Berthold et al. (Eds.): IDA 2020, LNCS 12080, pp. 561–573, 2020.
https://doi.org/10.1007/978-3-030-44584-3_44

problem transformation is to decompose the original task into several binary classification tasks, one per label. More specifically, each task consists of training a classifier that predicts whether or not a specific label is relevant for a query instance. This approach is called *binary relevance* (BR) learning [3]. Beyond BR, many more sophisticated strategies have been developed, most of them trying to exploit correlations and interdependencies between labels [28]. In fact, BR is often criticized for ignoring such dependencies, implicitly assuming that the relevance of one label is (statistically) independent of the relevance of another label. In spite of this, or perhaps just because of this simplification, BR proved to achieve state-of-the-art performance, especially for so-called decomposable loss functions, for which its optimality can even be corroborated theoretically [7,9].

Techniques for reducing MLC to SLC problems involve the choice of a base learner for solving the latter. Somewhat surprisingly, this choice is often neglected, despite having an important influence on generalization performance [10–12,15]. Even in more extensive studies [10,12], a base learner is fixed a priori in a more or less arbitrary way. Broader studies considering multiple base learners, such as [6,22], are relatively rare and rather limited in terms of the number of base learners considered. Only recently, greater attention to the choice of the base learner has been paid in the field of automated machine learning (AutoML) [17,24,25], where the base learner is considered as an important "hyper-parameter" to tune. Indeed, while optimizing the selection of base learners is laborious and computationally expensive in general, which could be one reason for why it has been tackled with reservation, AutoML now offers new possibilities in this direction.

Motivated by these opportunities, and building on recent AutoML methodology, we investigate the idea of base learner selection for BR in a more systematic way. Instead of only choosing a single base learner to be used for all labels simultaneously, we even allow for selecting an individual learner for each label (i.e., each binary classification task) separately. In an extensive experimental study, we find that customizing BR in a label-wise manner can significantly improve generalization performance.

2 Multi-label Classification

The setting of *multi-label classification* (MLC) allows an instance to belong to several classes simultaneously. Consequently, several class labels can be assigned to an instance at the same time. For example, a single image could be tagged with labels `Sun` and `Beach` and `Sea` and `Yacht`.

2.1 Problem Setting

To formalize this learning problem, let $\mathcal{X}$ denote an instance space and $\mathcal{L} = \{\lambda_1, \ldots, \lambda_m\}$ a finite set of m class labels. An instance $\boldsymbol{x} \in \mathcal{X}$ is then (non-deterministically) associated with a subset of class labels $L \in 2^{\mathcal{L}}$. The subset L is often called the set of relevant labels, while its complement $\mathcal{L} \setminus L$ is considered

irrelevant for $\boldsymbol{x}$. Furthermore, a set L of relevant labels can be identified by a binary vector $\boldsymbol{y} = (y_1, \ldots, y_m)$ where $y_i = 1$ if $\lambda_i \in L$ and $y_i = 0$ otherwise (i.e., if $\lambda_i \in \mathcal{L} \setminus L$). The set of all label combinations is denoted by $\mathcal{Y} = \{0, 1\}^m$.

Generally speaking, a multi-label classifier $\boldsymbol{h}$ is a mapping $\boldsymbol{h} : \mathcal{X} \longrightarrow \mathcal{Y}$ returning, for a given instance $\boldsymbol{x} \in \mathcal{X}$, a prediction in the form of a vector

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

The MLC task can be stated as follows: Given a finite set of observations as training data $\mathcal{D}_{\text{train}} := (X_{\text{train}}, Y_{\text{train}}) = \big\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\big\}_{i=1}^{N} \subset \mathcal{X}^N \times \mathcal{Y}^N$, 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.

2.2 Loss Functions

A wide spectrum of loss functions has been proposed for MLC, many of which are generalizations or adaptations of losses for single-label classification. In general, these loss functions can be divided into two major categories: instance-wise and label-wise. While the latter first compute a loss for each label and then aggregate the values obtained across the labels, e.g., by taking the mean, instance-wise loss functions first compute a loss for each instance and subsequently aggregate the losses over all instances in the test data. As an obvious advantage of label-wise loss functions, note that they can be optimized by optimizing a standard SLC loss for each label separately. In other words, label-wise losses naturally harmonize with label-wise decomposition techniques such as BR. Since this allows for a simpler selection of the base learner per label, we focus on two such loss functions in the following. For additional details on MLC and loss functions, especially instance-wise losses, we refer to [23,29].

Let $\mathcal{D}_{\text{test}} := (X_{\text{test}}, Y_{\text{test}}) = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{S} \subset \mathcal{X}^S \times \mathcal{Y}^S$ be a test set of size S. Further, let $H = (\boldsymbol{h}(\boldsymbol{x}_1), \ldots, \boldsymbol{h}(\boldsymbol{x}_S)) \subset \mathcal{Y}^S$. Then, the Hamming loss, which can be seen as a generalized form of the error rate, is defined[1] as

$$\mathcal{L}_H(Y_{\text{test}}, H) := \frac{1}{m} \sum_{j=1}^{m} \frac{1}{S} \sum_{i=1}^{S} \llbracket y_{i,j} \neq h_j(\boldsymbol{x}_i) \rrbracket \quad . \tag{1}$$

Moreover, the label-wise macro-averaged F-measure (which is actually a measure of accuracy, not a loss function, and thus to be maximized) is given by

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

Obviously, to optimize the measures (1) and (2), it is sufficient to optimize each label individually, which corresponds to optimizing the inner term of the (first) sum.

[1] $\llbracket \cdot \rrbracket$ is the indicator function.

2.3 Binary Relevance

As already said, binary relevance learning decomposes the MLC task into several binary classification tasks, one for each label. For every such task, a single-label classifier, such as an SVM, random forest, or logistic regression, is trained. More specifically, a classifier for the j^{th} label is trained on the dataset $\{(\boldsymbol{x}_i, y_{i,j})\}_{i=1}^{N}$. Formally, BR induces a multi-label predictor

$$\mathbf{BR_b} : \mathcal{X} \longrightarrow \mathcal{Y}, \quad \boldsymbol{x} \mapsto \big(b_1(\boldsymbol{x}), b_2(\boldsymbol{x}), \ldots, b_m(\boldsymbol{x})\big) \ ,$$

where $b_j : \mathcal{X} \longrightarrow \{0, 1\}$ represents the prediction of the base learner for the j^{th} label.

3 Related Work

Binary relevance has been subject to modifications in various directions, an excellent overview of which is provided in a recent survey [28]. Extensions of BR mainly focus on its inability to exploit label correlations, due to treating all labels independently of each other. Three types of approaches have been proposed to overcome this problem. The first is to use *classifier chains* [15]. In this approach, one first defines a total order among the m labels and then trains binary classifiers in this order. The input of the classifier for the i^{th} label is the original data plus the predictions of *all classifiers* for labels preceding this label in the chain. Similarly, in addition to the binary classifiers for the m labels, *stacking* uses a second layer of m meta-classifiers, one for each label, which take as input the original data augmented by the predictions of *all* base learners [11,21]. A third approach seeks to capture the dependencies in a Bayesian network, and to learn such a network from the data [1,20]. One can then use probabilistic inference to compute the probability for each possible prediction.

Another line of research looks at how the problem of imbalanced classes can be addressed using BR. Class imbalance constitutes an important challenge in multi-label classification in general, since most labels are usually irrelevant for an instance, i.e., the overwhelming majority of labels in a binary task is negative. Using BR, the imbalance can be "repaired" in a label-wise manner, using techniques for standard binary classification, such as sampling [5] or thresholding the decision boundary [13]. An approach taking dependencies among labels into account (and hence applied prior to splitting the problem) is presented in [27].

To the best of our knowledge, this is the first approach in which the base learner used for the different labels is subject to optimization itself. In fact, except for AutoML tools, we are not even aware of an approach optimizing a single base learner applied to all labels. In all the above approaches, the choice of the base learners is an external decision and not part of the learning problem itself.

4 Label-Wise Selection of Base Learners

As already stated before, while various attempts at improving binary relevance learning by capturing label dependencies have been made, the choice of the base learner for tackling the underlying binary problems—as another potential source of improvement—has attracted much less attention in the literature so far. If considered at all, this choice has been restricted to the selection of a *single* learner, which is applied to all m binary problems simultaneously.

We proceed from a portfolio of base learners

$$\mathcal{A} := \left\{ a \mid a : (\mathcal{X}^n \times \{0,1\}^n) \longrightarrow (\mathcal{X} \longrightarrow \{0,1\}) \right\}.$$

Then, given training data $\mathcal{D}_{\text{train}} = (X_{\text{train}}, Y_{\text{train}})$, the objective is to find the base learner a for which BR performs presumably best on test data $\mathcal{D}_{\text{test}} = (X_{\text{test}}, Y_{\text{test}})$ with respect to some loss function $\mathcal{L}$:

$$\arg\min_{a \in \mathcal{A}} \ \mathcal{L}\big(Y_{\text{test}}, \mathbf{BR}_{\boldsymbol{b}}(X_{\text{test}})\big), \ \text{with } b_j := a\left(X_{\text{train}}, Y_{\text{train}}^{(j)}\right) \ , \tag{3}$$

where $Y_{\text{train}}^{(i)}$ denotes the j^{th} column of the label matrix Y_{train}.

Moreover, we propose to leverage the independence assumption underlying BR to select a different base learner for each of the labels, and refer to this variant as LiBRe. We are thus interested in solving the following problem:

$$\arg\min_{\boldsymbol{a} \in \mathcal{A}^m} \mathcal{L}\big(Y_{\text{test}}, \mathbf{BR}_{\boldsymbol{b}}(X_{\text{test}})\big), \ \text{with } b_j := a_j\left(X_{\text{train}}, Y_{\text{train}}^{(j)}\right) \ . \tag{4}$$

Compared to (3), we thus significantly increase flexibility. In fact, by taking advantage of the different behavior of the respective base learners, and the ability to model the relationship between features and a class label differently for each binary problem, one may expect to improve the overall performance of BR. On the other side, the BR learner as a whole is now equipped with many degrees of freedom, namely the choice of the base learners, which can be seen as "hyper-parameters" of LiBRe. Since this may easily lead to undesirable effects such as over-fitting of the training data, an improvement in terms of generalization performance (approximated by the performance on the test data) is by no means self-evident. From this point of view, the restriction to a single base learner in (3) can also be seen as a sort of regularization. Such kind of regulation can indeed be justified for various reasons. In most cases, for example, the binary problems are indeed not completely different but share important characteristics.

Computationally, (4) may appear more expensive than choosing a single base learner jointly for all the labels, at least at first sight. However, the complexity in terms of the number of base learners to be evaluated remains exactly the same. In fact, just like in (3), we need to fit a BR model for every base learner exactly once. The only difference is that, instead of picking one of the base learners for all labels in the end, LiBRe assembles the base learners performing best for the respective labels (recall that we head for label-wise decomposable performance measures).

5 Experimental Evaluation

This section presents an empirical evaluation of LiBRe, comparing it to the use of a single base learner as a baseline. We first describe the experimental setup (Sect. 5.1), specify the baseline with the single best base learner (Sect. 5.2), and define the oracle performance (Sect. 5.3) for an upper bound. Finally, the experimental results are presented in Sect. 5.4.

5.1 Experimental Setup

For the evaluation, we considered a total of 24 MLC datasets. These datasets stem from various domains, such as text, audio, image classification, and biology, and range from small datasets with only a few instances and labels to larger datasets with thousands of instances and hundreds of labels. A detailed overview is given in Table 1, where, in addition to the number of instances (#I) and number of labels (#L), statistics regarding the label-to-instance ratio (L2IR), the percentage of unique label combinations (ULC), and the average label cardinality (card.) are given.

The train and validation folds were derived by conducting a nested 2-fold cross validation, i.e., to assess the test performance we have an outer loop of 2-fold cross validation. To tune the thresholds and select the base learner, we again split the training fold of the outer loop into train and validation sets by 2-fold cross validation. The entire process is repeated 5 times with different random seeds for the cross validation. Throughout this study, we trained and evaluated a total of 14,400 instances of BR and 649,800 base learners accordingly.

Furthermore, we consider two performance measures, namely the Hamming loss $\mathcal{L}_H$ and the macro-averaged label-wise F-measure as defined in (1) and (2), respectively. A binary prediction is obtained by thresholding the prediction of an underlying scoring classifier, which produces values in the unit interval (the higher the value, the more likely a label is considered relevant). The thresholds $\boldsymbol{\tau} = (\tau_1, \tau_2, \ldots, \tau_m)$ are optimized by a grid search considering values for $\tau_i \in [0, 1]$ and a step size of 0.01. When optimizing the thresholds, we either allow for label-wise optimization or constrain the threshold to be the same for all labels (uniform τ), i.e., $\tau_i = \tau_j$ for all $i, j \in \{1, \ldots, m\}$.

In order to determine significance of results, we apply a Wilcoxon signed rank test with a threshold for the p-value of 0.05. Significant improvements of LiBRe are marked by $\bullet$ and significant degradations by $\circ$.

We executed the single BR evaluation runs, i.e., training and evaluating either on the validation or test split, on up to 300 nodes in parallel, each of them equipped with 8 CPU cores and 32 GB of RAM, and a timeout of 6 h. Due to the limitation of the memory and the runtime, some of the evaluations failed due to memory overflows or timeouts.

The implementation is based on the Java machine learning library WEKA [8] and an extension for multi-label classification called MEKA [16]. In our study, we consider a total of 20 base learners from WEKA: BayesNet (BN), DecisionStump (DS), IBk, J48, JRip (JR), KStar (KS), LMT, Logistic (L), MultilayerPerceptron

Table 1. The datasets used in this study. Furthermore, the number of instances (#I), the number of labels (#L), the label-to-instance ratio (L2IR), the percentage of unique label combinations (ULC), and the label cardinality (card.) are given.

Dataset	#I	#L	L2IR	ULC	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

(MlP), NaiveBayes (NB), NaiveBayesMultinomial (NBM), OneR (1R), PART (P), REPTree (REP), RandomForest (RF), RandomTree (RT), SMO, SimpleLogistic (SL), VotedPerceptron (VP), ZeroR (0R). All the data and source code is made available via GitHub (https://github.com/mwever/LiBRe).

5.2 Single Best Base Learner

To figure out how much we can benefit from selecting a base learner for each label individually, and whether this flexibility is beneficial at all, we define the single best base learner, subsequently referred to as SBB, as a baseline. In principle, SBB is nothing but a grid search over the portfolio of base learners (3).

When considering a base learner a, it is chosen to be employed as a base learner for every label. After training and validating the performance, we pick the base learner that performs best overall. This baseline thus gives an upper bound on the performance of what can be achieved when the base learner is not chosen for each label individually. As simple and straight-forward as it is, this baseline represents what is currently possible in implementations of MLC libraries, and already goes beyond what is most commonly done in the literature.

5.3 Optimistic Versus Validated Optimization

In addition to the results obtained by selecting the base learner(s) according to the validation performance (obtained in the inner loop of the nested cross validation), we consider optimistic performance estimates, which are obtained as follows: After having trained the base learners on the training data, we select the presumably best one, not on the basis of their performance on validation data, but based on their actual test performance (as observed in the outer loop

Fig. 1. The heat map shows the average share of each base learner being employed for a label with respect to the optimized performance measure: Hamming ($\mathcal{L}_H$) or the label-wise macro averaged F-measure (F).

of the nested cross-validation). Intuitively, this can be understood as a kind of "oracle" performance: Given a set of candidate predictors to choose from, the oracle anticipates which of them will perform best on the test data.

Although these performances should be treated with caution, and will certainly tend to overestimate the true generalization performance of a classifier, they can give some information about the potential of the optimization. More specifically, these optimistic performance estimates suggest an upper bound on what can be obtained by the nested optimization routine.

5.4 Results

In Fig. 1, the average share of a base learner per label is shown. From this heatmap, it becomes obvious that for the SBB baseline only a subset of base learners plays a role. However, one can also notice that the distribution of the shares varies when different performance measures are optimized. Furthermore, although random forest (RF) achieves significant shares of 0.8 for the Hamming loss and around 0.6 for the F-measure, it is not best on all the datasets. To put it differently, one still needs to optimize the base learner per dataset. This is especially true, when different performance measures are of interest.

In the case of LiBRe, it is clearly recognizable how the shares are distributed over the base learners, in contrast to SBB. For example, the shares of RF decrease to 0.29 for F-measure and to 0.25 for Hamming, respectively. Moreover, base learners that did not even play any role in SBB are now gaining in importance and are selected quite often. Although there are significant differences in the frequency of base learners being picked, there is not a single base learner in the portfolio that was never selected.

In Table 2, the results for optimizing Hamming loss are presented. The optimistic performance estimates already indicate that there is not much room for improvement. This comes at no surprise, since the datasets are already pretty much saturated, i.e., the loss is already close to 0 for most of the datasets. While LiBRe performs competitively to SBB for the setting with uniform τ, SBB compares favourably to LiBRe in the case where the thresholds can be tuned in a label-wise manner. Apparently, the additional degrees of freedom make LiBRe more prone to over-fitting, especially on smaller datasets.

In contrast to the previous results, for the optimization of the F-measure, the optimistic performance estimates already give a promising outlook on the

Table 2. Results obtained for minimizing $\mathcal{L}_H$ optimistically resp. with validation performances. Thresholds are optimized either jointly for all the labels (uniform τ) or label-wise. Best performances per setting and dataset are highlighted in bold. Significant improvements of LiBRe are marked by a • and degradations by ∘.

Dataset	Optimistic uniform τ		Validated uniform τ		Optimistic label-wise τ		Validated label-wise τ	
	LiBRe	SBB	LiBRe	SBB	LiBRe	SBB	LiBRe	SBB
arts1	**0.0515**	0.0536	**0.0531**	0.0538	**0.0504**	0.0513	0.0526	**0.0525**
bibtex	**0.0118**	0.0126	**0.0126**	0.0127	**0.0115**	0.0120	0.0151	**0.0139**
birds	**0.0357**	0.0397	0.0476	**0.0420** ∘	**0.0329**	0.0352	0.0470	**0.0422** ∘
bookmarks	**0.0085**	0.0087	**0.0086**	0.0087 •	**0.0085**	0.0086	**0.0105**	0.0114 •
business1	**0.0233**	0.0248	**0.0241**	0.0249 •	**0.0218**	0.0223	**0.0227**	0.0228
computers1	**0.0313**	0.0334	**0.0329**	0.0335	**0.0301**	0.0306	0.0323	**0.0312**
education1	**0.0352**	0.0365	**0.0359**	0.0369 •	**0.0340**	0.0344	0.0354	**0.0349** ∘
emotions	**0.1762**	0.1800	0.1926	**0.1856** ∘	**0.1684**	0.1712	0.1961	**0.1875** ∘
enron-f	**0.0447**	0.0474	0.0481	**0.0477**	**0.0437**	0.0445	0.0485	**0.0469** ∘
entertainment1	**0.0432**	0.0466	**0.0440**	0.0469 •	**0.0414**	0.0434	**0.0430**	0.0443 •
flags	**0.1732**	0.1979	0.2134	**0.2088**	**0.1635**	0.1799	**0.2105**	0.2158
genbase	**7.0E-4**	0.0014	0.0069	**0.0016** ∘	**6.0E-4**	7.0E-4	0.0070	**0.0023** ∘
health1	**0.0305**	0.0344	**0.0313**	0.0347 •	**0.0282**	0.0297	0.0303	**0.0302**
llog-f	**0.0149**	0.0153	0.0202	**0.0157** ∘	**0.0145**	0.0149	0.0230	**0.0178** ∘
mediamill	**0.0268**	0.0270	0.0271	**0.0270**	**0.0261**	0.0262	0.0265	**0.0265**
medical	**0.0084**	0.0103	0.0115	**0.0109**	**0.0078**	0.0093	0.0136	**0.0116**
recreation1	**0.0459**	0.0472	**0.0472**	0.0473	**0.0446**	0.0453	0.0468	**0.0462**
reference1	**0.0244**	0.0264	**0.0267**	0.0268	**0.0230**	0.0245	0.0255	**0.0251**
scene	**0.0781**	0.0788	0.0817	**0.0794** ∘	**0.0757**	0.0762	0.0816	**0.0800** ∘
science1	**0.0281**	0.0311	**0.0311**	0.0317	**0.0269**	0.0291	0.0304	**0.0302**
social1	**0.0197**	0.0208	0.0227	**0.0210**	**0.0188**	0.0196	0.0223	**0.0200**
society1	**0.0474**	0.0495	**0.0479**	0.0496 •	**0.0444**	0.0455	**0.0455**	0.0461 •
tmc2007	**0.0601**	0.0611	**0.0600**	0.0611 •	**0.0590**	0.0611	0.0613	**0.0611**
yeast	**0.1914**	0.1926	0.2002	**0.1930** ∘	**0.1886**	0.1890	0.1940	**0.1929** ∘

potential for improving the generalization performance through the label-wise selection of the base learners. More precisely, they indicate that performance gains of up to 11% points are possible. Independent of the threshold optimization variant, LiBRe outperforms the SBB baseline, yielding the best performance on two third of the considered datasets, 13 improvements of which are significant in the case of uniform τ, and 11 in the case of label-wise τ. Significant degradations of LiBRe compared to SBB can only be observed for 2 respectively 3 datasets. Hence, for the F-measure, LiBRe compares favorably to the SBB baseline.

In summary, we conclude that LiBRe does indeed yield performance improvements. However, increasing the flexibility of BR also makes it more prone to over-fitting. Furthermore, these results were obtained by conducting a nested 2-fold cross validation. While keeping the computational costs of this evaluation reasonable, this implies that, for the purpose of validation, the base learners were trained on only one fourth of the original dataset. Therefore, considering nested 5-fold or 10-fold cross validation could help to reduce the observed over-fitting.

Table 3. Results for maximizing the F-measure optimistically resp. with validation performances. Thresholds are optimized either jointly for all the labels (uniform τ) or label-wise. Best performances per setting and dataset are highlighted in bold. Significant improvements of LiBRe are marked by a • and degradations by ○.

Dataset	Optimistic uniform τ		Validated uniform τ		Optimistic label-wise τ		Validated label-wise τ	
	LiBRe	SBB	LiBRe	SBB	LiBRe	SBB	LiBRe	SBB
arts1	**0.3445**	0.2749	**0.3018**	0.2684 •	**0.3680**	0.3211	**0.3184**	0.3001 •
bibtex	**0.4020**	0.3027	**0.3391**	0.2998 •	**0.4194**	0.3516	**0.3378**	0.3041 •
birds	**0.5404**	0.4424	0.3707	**0.3961** ○	**0.5832**	0.5310	0.3843	**0.3981** ○
bookmarks	**0.2495**	0.2244	**0.2347**	0.2239 •	**0.2646**	0.2516	**0.2435**	0.2416
business1	**0.3692**	0.2854	**0.2970**	0.2659 •	**0.3874**	0.3197	**0.3006**	0.2790 •
computers1	**0.3646**	0.2861	**0.3099**	0.2810 •	**0.3833**	0.3486	**0.3224**	0.3190
education1	**0.3346**	0.2468	**0.2594**	0.2437 •	**0.3591**	0.3022	**0.2652**	0.2612
emotions	**0.7068**	0.6946	0.6670	**0.6779**	**0.7186**	0.7135	0.6761	**0.6859** ○
enron-f	**0.2870**	0.2192	0.2056	**0.2096**	**0.3138**	0.2773	**0.2077**	0.2069
entertainment1	**0.4470**	0.3673	**0.3929**	0.3500 •	**0.4639**	0.4049	**0.3950**	0.3774 •
flags	**0.6280**	0.5634	**0.5230**	0.5098	**0.6474**	0.5981	**0.5150**	0.5145
genbase	**0.8126**	0.7798	0.6039	**0.7421** ○	**0.8141**	0.8119	0.6201	**0.6390**
health1	**0.4203**	0.3259	**0.3486**	0.3208 •	**0.4312**	0.3582	**0.3464**	0.3225 •
llog-f	**0.1569**	0.0808	**0.0730**	0.0689	**0.1834**	0.1264	**0.0744**	0.0741
mediamill	**0.3766**	0.3499	0.3481	**0.3483**	**0.4010**	0.3898	0.3543	**0.3600** ○
medical	**0.4960**	0.3852	0.3560	**0.3639**	**0.5251**	0.4523	**0.3547**	0.3208 •
recreation1	**0.4964**	0.4224	**0.4669**	0.4160 •	**0.5093**	0.4675	**0.4670**	0.4494 •
reference1	**0.3185**	0.2254	**0.2477**	0.2021 •	**0.3393**	0.2860	**0.2587**	0.2418 •
scene	**0.7831**	0.7816	0.7734	**0.7776**	**0.7909**	0.7897	0.7759	**0.7812**
science1	**0.3824**	0.2724	**0.2928**	0.2637 •	**0.4033**	0.3240	**0.3036**	0.2662 •
social1	**0.3629**	0.3073	0.3046	**0.3060**	**0.3737**	0.3119	**0.3103**	0.2769 •
society1	**0.3437**	0.2807	**0.3180**	0.2688 •	**0.3597**	0.3382	0.3215	**0.3238**
tmc2007	**0.5659**	0.5342	**0.5467**	0.5342	**0.5782**	0.5525	**0.5656**	0.5484 •
yeast	**0.4970**	0.4750	**0.4800**	0.4731 •	**0.5145**	0.5084	0.4922	**0.4947**

6 Conclusion

In this paper, we have not only demonstrated the potential of binary relevance to optimize label-wise macro averaged measures, but also the importance of the base learner as a hyper-parameter for each label. Especially for the case of optimizing for F1 macro-averaged over the labels, we could achieve significant performance improvements by choosing a proper base learner in a label-wise manner. Compared to selecting the best single base learner, choosing the base learner for each label individually comes at no additional cost in terms of base learner evaluations. Moreover, the label-wise selection of base learners can be realized by a straight-forward grid search.

As the label-wise choice of a base learner has already led to considerable performance gains, we plan to examine to what extent the optimization of the hyper-parameters of those base learners can lead to further improvements. Furthermore, we want to increase the efficiency of the tuning by replacing the grid search with a heuristic approach.

Another direction of future work concerns the avoidance of over-fitting effects due to an overly excessive flexibility of LiBRe. As already explained, the restriction to a single base learner can be seen as a kind of regularization, which, however, appears to be too strong, at least according to our results. On the other side, the full flexibility of LiBRe does not always pay off either. An interesting compromise could be to restrict the number of different base learners used by LiBRe to a suitable value $k \in \{1, \ldots, m\}$. Technically, this comes down to finding the arg min in (4), not over $\boldsymbol{a} \in \mathcal{A}^m$, but over $\{\boldsymbol{a} \in \mathcal{A}^m \mid \#\{a_1, \ldots, a_m\} \leq k\}$.

Acknowledgement. This work was supported by the German Research Foundation (DFG) within the Collaborative Research Center "On-The-Fly Computing" (SFB 901/3 project no. 160364472). The authors also gratefully acknowledge support of this project through computing time provided by the Paderborn Center for Parallel Computing (PC^2).

References

1. Antonucci, A., Corani, G., Mauá, D.D., Gabaglio, S.: An ensemble of Bayesian networks for multilabel classification. In: IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, 3–9 August 2013, pp. 1220–1225 (2013)
2. Barutcuoglu, Z., Schapire, R.E., Troyanskaya, O.G.: Hierarchical multi-label prediction of gene function. Bioinformatics **22**(7), 830–836 (2006). https://doi.org/10.1093/bioinformatics/btk048
3. Boutell, M.R., Luo, J., Shen, X., Brown, C.M.: Learning multi-label scene classification. Pattern Recogn. **37**(9), 1757–1771 (2004). https://doi.org/10.1016/j.patcog.2004.03.009
4. Cabral, R.S., la Torre, F.D., Costeira, J.P., Bernardino, A.: Matrix completion for multi-label image classification. In: 25th Annual Conference on Neural Information Processing Systems 2011, Advances in Neural Information Processing Systems, Granada, Spain, vol. 24, pp. 190–198 (2011)
5. Charte, F., Rivera, A.J., del Jesús, M.J., Herrera, F.: Addressing imbalance in multilabel classification: measures and random resampling algorithms. Neurocomputing **163**, 3–16 (2015). https://doi.org/10.1016/j.neucom.2014.08.091
6. Cherman, E.A., Metz, J., Monard, M.C.: Incorporating label dependency into the binary relevance framework for multi-label classification. Exp. Syst. Appl. **39**(2), 1647–1655 (2012). https://doi.org/10.1016/j.eswa.2011.06.056
7. Dembczynski, K., Waegeman, W., Cheng, W., Hüllermeier, E.: On label dependence and loss minimization in multi-label classification. Mach. Learn. **88**(1–2), 5–45 (2012). https://doi.org/10.1007/s10994-012-5285-8
8. Frank, E., Hall, M.A., Witten, I.H.: The Weka workbench. Online appendix. In: Frank, E., Hall, M.A., Witten, I.H. (eds.) Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann, Cambridge (2016)
9. Luaces, O., Díez, J., Barranquero, J., del Coz, J.J., Bahamonde, A.: Binary relevance efficacy for multilabel classification. Prog. AI **1**(4), 303–313 (2012). https://doi.org/10.1007/s13748-012-0030-x
10. Madjarov, G., Kocev, D., Gjorgjevikj, D., Dzeroski, S.: An extensive experimental comparison of methods for multi-label learning. Pattern Recogn. **45**(9), 3084–3104 (2012). https://doi.org/10.1016/j.patcog.2012.03.004

11. Montañés, E., Senge, R., Barranquero, J., Quevedo, J.R., del Coz, J.J., Hüllermeier, E.: Dependent binary relevance models for multi-label classification. Pattern Recogn. **47**(3), 1494–1508 (2014). https://doi.org/10.1016/j.patcog.2013.09.029
12. Moyano, J.M., Galindo, E.L.G., Cios, K.J., Ventura, S.: Review of ensembles of multi-label classifiers: models, experimental study and prospects. Inf. Fusion **44**, 33–45 (2018). https://doi.org/10.1016/j.inffus.2017.12.001
13. Pillai, I., Fumera, G., Roli, F.: Threshold optimisation for multi-label classifiers. Pattern Recogn. **46**(7), 2055–2065 (2013). https://doi.org/10.1016/j.patcog.2013.01.012
14. Qi, G., Hua, X., Rui, Y., Tang, J., Mei, T., Zhang, H.: Correlative multi-label video annotation. In: Proceedings of the 15th International Conference on Multimedia 2007, Augsburg, Germany, 24–29 September 2007, pp. 17–26 (2007). https://doi.org/10.1145/1291233.1291245
15. Read, J., Pfahringer, B., Holmes, G., Frank, E.: Classifier chains for multi-label classification. Mach. Learn. **85**(3), 333–359 (2011). https://doi.org/10.1007/s10994-011-5256-5
16. Read, J., Reutemann, P., Pfahringer, B., Holmes, G.: MEKA: a multi-label/multi-target extension to Weka. J. Mach. Learn. Res. **17**(21), 667–671 (2016)
17. de Sá, A.G.C., Freitas, A.A., Pappa, G.L.: Automated selection and configuration of multi-label classification algorithms with grammar-based genetic programming. Parallel Prob. Solving Nat. - PPSN XV **2018**, 308–320 (2018). https://doi.org/10.1007/978-3-319-99259-4_25
18. Sanden, C., Zhang, J.Z.: Enhancing multi-label music genre classification through ensemble techniques. In: Proceeding of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval, Beijing, China, pp. 705–714 (2011). https://doi.org/10.1145/2009916.2010011
19. Schapire, R.E., Singer, Y.: BoosTexter: a boosting-based system for text categorization. Mach. Learn. **39**(2/3), 135–168 (2000). https://doi.org/10.1023/A:1007649029923
20. Sucar, L.E., Bielza, C., Morales, E.F., Hernandez-Leal, P., Zaragoza, J.H., Larrañaga, P.: Multi-label classification with bayesian network-based chain classifiers. Pattern Recogn. Lett. **41**, 14–22 (2014). https://doi.org/10.1016/j.patrec.2013.11.007
21. Tahir, M.A., Kittler, J., Bouridane, A.: Multi-label classification using stacked spectral kernel discriminant analysis. Neurocomputing **171**, 127–137 (2016). https://doi.org/10.1016/j.neucom.2015.06.023
22. Tsoumakas, G., Katakis, I.: Multi-label classification: an overview. IJDWM **3**(3), 1–13 (2007). https://doi.org/10.4018/jdwm.2007070101
23. Tsoumakas, G., Katakis, I., Vlahavas, I.P.: Mining multi-label data. In: Maimon, O., Rokach, L. (eds.) Data Mining and Knowledge Discovery Handbook, pp. 667–685. Springer, Boston (2010). https://doi.org/10.1007/978-0-387-09823-4_34
24. Wever, M., Mohr, F., Hüllermeier, E.: Automated multi-label classification based on ML-Plan. CoRR abs/1811.04060 (2018)
25. Wever, M.D., Mohr, F., Tornede, A., Hüllermeier, E.: Automating multi-label classification extending ML-Plan (2019)
26. Xue, X., Zhang, W., Zhang, J., Wu, B., Fan, J., Lu, Y.: Correlative multi-label multi-instance image annotation. In: IEEE International Conference on Computer Vision, pp. 651–658 (2011). https://doi.org/10.1109/ICCV.2011.6126300
27. Zhang, M., Li, Y., Liu, X.: Towards class-imbalance aware multi-label learning. In: Proceedings of the 24th International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, 2015, pp. 4041–4047 (2015)

28. Zhang, M.-L., Li, Y.-K., Liu, X.-Y., Geng, X.: Binary relevance for multi-label learning: an overview. Frontiers Comput. Sci. **12**(2), 191–202 (2018). https://doi.org/10.1007/s11704-017-7031-7
29. Zhang, M., Zhou, Z.: A review on multi-label learning algorithms. IEEE Trans. Knowl. Data Eng. **26**(8), 1819–1837 (2014). https://doi.org/10.1109/TKDE.2013.39

8 Ensembles of Evolved Nested Dichotomies for Classification

Declaration of the specific contributions of the author The idea for the approach and its implementation can be attributed to the author. The paper was mostly written by the author. Felix Mohr and Eyke Hüllermeier helped in increasing the overall quality of the paper with simplifications of the formalisms and revising the text.

Ensembles of Evolved Nested Dichotomies for Classification

Marcel Wever
Heinz Nixdorf Institute
Paderborn University, Germany
marcel.wever@upb.de

Felix Mohr
Heinz Nixdorf Institute
Paderborn University, Germany
felix.mohr@upb.de

Eyke Hüllermeier
Heinz Nixdorf Institute
Paderborn University, Germany
eyke@upb.de

ABSTRACT

In multinomial classification, reduction techniques are commonly used to decompose the original learning problem into several simpler problems. For example, by recursively bisecting the original set of classes, so-called nested dichotomies define a set of binary classification problems that are organized in the structure of a binary tree. In contrast to the existing one-shot heuristics for constructing nested dichotomies and motivated by recent work on algorithm configuration, we propose a genetic algorithm for optimizing the structure of such dichotomies. A key component of this approach is the proposed genetic representation that facilitates the application of standard genetic operators, while still supporting the exchange of partial solutions under recombination. We evaluate the approach in an extensive experimental study, showing that it yields classifiers with superior generalization performance.

CCS CONCEPTS

• **Computing methodologies → Classification and regression trees**; *Supervised learning by classification*; *Bagging*;

KEYWORDS

Classification, hierarchical decomposition, indirect encoding

ACM Reference Format:
Marcel Wever, Felix Mohr, and Eyke Hüllermeier. 2018. Ensembles of Evolved Nested Dichotomies for Classification. In *GECCO '18: Genetic and Evolutionary Computation Conference, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3205455.3205562

1 INTRODUCTION

The reduction of a multinomial (*aka* polychotomous, multi-class) classification problem to several binary problems is a common approach in supervised machine learning. There are several reasons for applying a reduction of this kind. In particular, it makes classifiers that are inherently binary (such as support vector machines) amenable to the problem. Moreover, even for classifiers that are not restricted in this sense, and which are in principle able to handle multi-class problems directly (such as decision trees), a reduction complying with the divide and conquer paradigm of algorithm design often leads to improvements in terms of predictive accuracy.

GECCO '18, July 15–19, 2018, Kyoto, Japan

ACM ISBN 978-1-4503-5618-3/18/07…$15.00
https://doi.org/10.1145/3205455.3205562

Different approaches for the decomposition of multi-class problems have been proposed in the literature; among the most common reduction schemes are one-vs-rest (OvR), one-vs-one (OvO), also known as all-pairs or pairwise classification [9], and nested dichotomies (ND) as introduced in [7].

The first two approaches, OvR and OvO, build ensembles of binary classifiers that are trained independently of each other. Since these classifiers may produce inconsistencies at prediction time, specific aggregation techniques (such as winner-takes-all or majority vote) are needed to combine their predictions into an overall prediction for the original problem. In contrast to that, NDs decompose the original multi-class problem in a hierarchical manner, recursively splitting a set of classes into two distinct subsets. This way, a hierarchical structure in the form of a binary tree is produced, which is inherently consistent and associates each original class with (a path from the root to) a leaf node. An example of a nested dichotomy for a 4-class problem is shown in Figure 1. The classifiers at inner nodes, referred to as *base classifiers* or *base learners*, are supposed to predict which of the two subsets the query instance belongs to, given it belongs to the set of classes associated with that node; typically, this prediction is given in terms of a probability, so that, according to the chain rule of probability, the overall probability of each class is obtained as the product of the probabilities along the path from the root to the corresponding leaf node.

Since the structure of a dichotomy determines the classifiers that need to be trained at the inner nodes, and hence the classification problems that need to be solved by the learner, it has a strong influence on performance and should be chosen with care. Indeed, recent work in the field of algorithm selection and configuration has shown that tuning learning algorithms for a given data set can lead to substantial performance improvements [11, 14]. Motivated by advances in that field, we consider the problem of finding optimal reductions from multi-class to binary classification with the help of a genetic algorithm.

To facilitate an efficient search of the space of NDs, we propose a representation that is suitable for being used by evolutionary optimization algorithms. This representation is robust to standard mutation and recombination techniques and still enables individuals to share partial solutions—in contrast to the tree structure itself, which is not suitable as a genetic representation. We also present an algorithm for the construction of an ND from its genotype. By performing multiple evolutionary runs with different randomization seeds, we form bagged ensembles of nested dichotomies.

In an extensive experimental study, we compare our approach for evolving nested dichotomies to several baselines, including uniformly sampled NDs [7] and a state-of-the-art heuristic called random-pair nested dichotomies (RPND) [12]. The results show that the proposed genetic algorithm does outperform the baselines, a fortiori in the case of weak base classifiers.

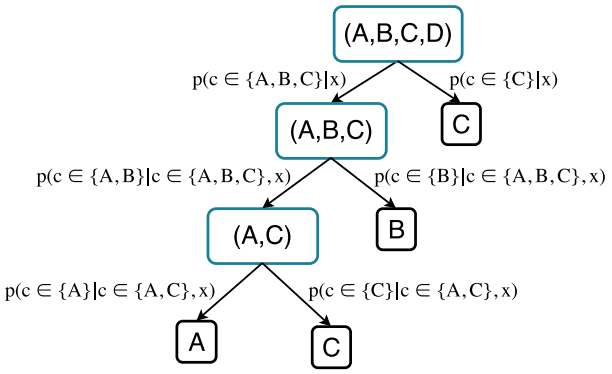


Figure 1: A nested dichotomy for a 4-class problem.

2 NESTED DICHOTOMIES

Given a classification problem with a set of n classes $C = \{c_1, \ldots, c_n\}$, a nested dichotomy (ND) is a recursive separation (C_l, C_r) of C into pairs of disjoint, nonempty subsets. Equivalently, an ND can be defined in terms of a full binary tree on n leaf nodes, which are (uniquely) labeled by the classes; moreover, the $n-1$ inner nodes are labeled by the set of classes in the subtree beneath that node (see Figure 1 for an example).

To turn a nested dichotomy into a multi-class classifier, a binary classifier is assigned to each inner node. The task of the classifier is to separate the set of classes C_l associated with its left successor node (the positive meta-class) from the set of classes C_r of its right successor (the negative meta-class). Given a set of training data

$$\mathcal{D} = \left\{(\boldsymbol{x}_i, c_i)\right\}_{i=1}^{N} \subset \mathcal{X} \times C\,,$$

where $\mathcal{X}$ is the underlying instance space, the binary classifiers are produced by applying a suitable *base learner* to the corresponding classification problems. We assume the classifier h_{C_l,C_r} associated with the dichotomy (C_l, C_r) to be a mapping of the form $\mathcal{X} \longrightarrow [0,1]$, where $h_{C_l,C_r}(\boldsymbol{x})$ is an estimation of the conditional probability $p(c \in C_l \mid c \in C_l \cup C_r, \boldsymbol{x})$, and hence $1 - h_{C_l,C_r}(\boldsymbol{x})$ an estimation of $p(c \in C_r \mid c \in C_l \cup C_r, \boldsymbol{x})$. Such a probabilistic classifier is produced by the base learner on the relevant subset of training data $\{(\boldsymbol{x}_i, c_i) \in \mathcal{D} \mid c_i \in C_l \cup C_r\}$.

Once the hierarchy of classifiers required by a nested dichotomy has been trained, a new instance $\boldsymbol{x}$ can be classified probabilistically as follows: For $c_j \in C$, let $C = C_0 \supset C_1 \supset \cdots \supset C_m = \{c_j\}$ be the chain of subsets associated with the nodes from the root to the leaf node labeled by c_j. Then, by virtue of the chain rule of probability,

$$p(c_j \mid \boldsymbol{x}) = \prod_{i=1}^{m-1} p\big(c_j \in C_{i+1} \mid c_j \in C_i, \boldsymbol{x}\big),$$

where $p(C_{i+1} \mid C_i, \boldsymbol{x})$ is given by $h_{C_{i+1},C_i \setminus C_{i+1}}(\boldsymbol{x})$ if C_{i+1} is the left successor of C_i, and $1 - h_{C_i \setminus C_{i+1},C_{i+1}}(\boldsymbol{x})$ if C_{i+1} is the right successor of C_i. In other words, the probability p_j of class c_j is obtained by multiplying the predicted probabilities along the path from the root of the ND tree to the leaf node associated with c_j. In case a definite decision in favor of a single class has to be made, the expected loss minimizer can be derived from the probability distribution $(p_1, \ldots, p_n)$; for 0/1 loss, this is simply the class c_j for which p_j is highest.

As already mentioned, the structure of a dichotomy has an important influence on performance and predictive accuracy. However, finding optimal structures is not an easy task, especially because the space of ND structures is extremely large. In fact, it is not difficult to see that this set is isomorphic to the set of unordered complete binary trees on n nodes, the size of which is known to be $1 \cdot 3 \cdot 5 \cdots (2n-3) = (2n-3)!!$.

3 RELATED WORK

Being rooted in statistics, nested dichotomies have been introduced to the field of machine learning by Frank and Kramer [7]. Assuming that each ND is equally likely to perform well, a binary tree is sampled uniformly at random from the set of all possible trees. Forming ensembles of NDs via repeated sampling of these binary trees, Frank and Kramer showed that ensembles of NDs perform competitive to other reduction techniques. At the inner nodes, they use logistic regression and decision trees as base classifiers.

Further investigations were mainly focused on heuristics for optimizing the recursive structure of NDs. To this end, Dong et al. [3] present two approaches to balance the binary splits, either in terms of the number of classes or the number of instances per subset of the training data. Despite having a positive impact on runtime, which was the main concern of the authors, these approaches have only little effect on the predictive accuracy of such balanced NDs (the second approach even tends to deteriorate performance).

Instead of repeatedly selecting NDs uniformly at random, Rodríguez et al. [16] investigate other approaches for the assembly of ensembles. In particular, the authors found that the accuracy of ensembles can be improved using bagging [1], AdaBoost [8], and MultiBoost [18] with random NDs as the base learner.

In [4], the authors propose a more sophisticated strategy for splitting classes. In a first step, the centroid of each class (considered as a cluster of instances) is determined. Based on this information, for each split, a pair of classes with maximal distance between their centroids is chosen. Subsequently, the base learner is trained to discriminate those two classes. Finally, the model thus produced is used to assign the remaining classes to one of the subsets, depending on how the majority of the instances within each class is assigned. The authors found that this approach performs superior compared to the previous random methods.

Leathart et al. [12] picked up the idea of splitting the set of classes by seeding each subset with a single class and assigning the remaining classes based on the predictions of the trained base learner. Instead of using distances between centroids as a criterion, they randomly select a pair of classes to be used as seeds. Experimentally, they show their heuristic to yield state-of-the-art performance in terms of predictive accuracy.

4 NESTED DICHOTOMIES EVOLUTIONARY ALGORITHM (NDEA)

In this section, we introduce an evolutionary algorithm for searching the space of nested dichotomies in a systematic and efficient way. Key to the efficiency of this algorithm is to decouple the genetic representation from the phenotype that is used to assess the

fitness of an individual. Therefore, our discussion will start with the description of these operators, followed by the algorithm itself.

At first sight, the tree structure of an ND itself may appear like a reasonable genotype. However, the structure of NDs has specific semantics. In particular, each class of a multi-class problem occurs exactly once as a leaf of the tree structure. Applying standard genetic operators such as crossover and mutation would frequently lead to invalid individuals. One way to overcome this problem is to design specific genetic operators complying with this constraint, but this would make these operators computationally expensive.

Therefore, we keep the tree structure as a phenotype and define another genotype reducing the problem of evolving NDs to the evolution of a string of real values.. Subsequently, we define a way to constructed the phenotype (an ND) from the genetic representation.

4.1 Genetic Representation

For the genetic representation of nested dichotomies, we aim for an indirect representation that is robust in the face of genetic operators but still supports the exchange of partial solutions under recombination. To this end, given an n-class problem, we define a function $\varphi : C \times C \rightarrow \mathbb{R}$ that assigns each combination (c_i, c_j) of classes a real value that we interpret as a "distance". We require this function to be symmetric, i.e., $\varphi(c_i, c_j) = \varphi(c_j, c_i)$ for all $c_i, c_j \in C$. Moreover, $\varphi(c_i, c_i) = 0$ for all $c_i \in C$. Fixing an ordering $c_1, c_2, \ldots, c_n$ of the classes in C, we can arrange the values of φ in a symmetric matrix with zeros on the main diagonal as follows:

$$\begin{array}{c} \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{array} \begin{array}{c} \begin{array}{cccc} c_1 & c_2 & \ldots & c_n \end{array} \\ \begin{pmatrix} 0 & \varphi(c_1, c_2) & \ldots & \varphi(c_1, c_n) \\ \varphi(c_2, c_1) & 0 & & \varphi(c_2, c_n) \\ \vdots & & \ddots & \vdots \\ \varphi(c_n, c_1) & \varphi(c_n, c_2) & \ldots & 0 \end{pmatrix} \end{array}$$

Concatenating the rows of the upper triangular matrix yields the sequence

$$\varphi(c_1, c_2), \varphi(c_1, c_3), \ldots \varphi(c_1, c_n), \varphi(c_2, c_3), \ldots \varphi(c_{n-1}, c_n).$$

By fixing the order of the elements of φ, we choose the sequence of φ's values to serve as the genotype of an individual. As there are $n(n-1)/2$ many pairs of elements for an n-class problem, the genotype of an individual is an element of $\mathbb{R}^{n(n-1)/2}$.

Compared to the structured object of a nested dichotomy, our genetic representation is rather simple. More importantly, it enables us to use standard recombination techniques such as crossovers and polynomial mutation. Another advantage is that partial solutions may be exchanged when doing crossovers, since the distance information for pairs of classes is exchanged. Furthermore, thanks to fixing the ordering of the φ-values, we can easily restore the tuples $(c_i, c_j, \varphi(c_i, c_j))$, which are required for the phenotype construction.

4.2 Phenotype Construction

The fitness of an individual is assessed by training the corresponding nested dichotomy and estimating its predictive accuracy on validation data. Thus, from the genotype of an individual, we need to construct its phenotype, i.e., the nested dichotomy, to subsequently determine the individual's fitness.

Listing 1: Phenotype construction procedure

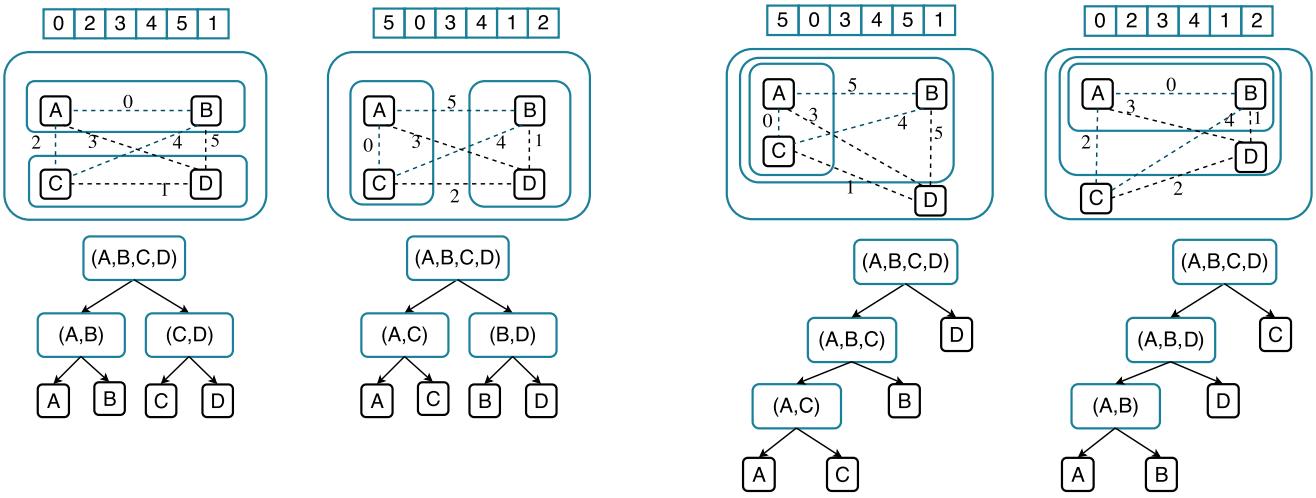
```
input: Queue Q, integer n
output Node root

Node array [1..n] nodeArray
for k=1 to n do
        nodeArray[k] = leaf(k)
od
while Q not empty do
    (c_i, c_j, φ(c_i, c_j)) ← dequeue(Q)
    leftNode ← nodeArray[i]
    rightNode ← nodeArray[j]
    if(leftNode == rightNode)
        continue

    subtreeRoot ← node(leftNode, rightNode)
    foreach k in newRoot.leaves do
        root[k] ← newRoot
    od
od
return nodeArray[0]
```

For the phenotype construction, we first need to restore the tuples $(c_i, c_j, \varphi(c_i, c_j))$ consisting of a pair of classes and a real value describing the distance between the two classes. Taking a sequence of genes $[\varphi(c_1, c_2), \varphi(c_1, c_3), \ldots \varphi(c_1, c_n), \varphi(c_2, c_3), \ldots \varphi(c_{n-1}, c_n)]$ as input, we define tuples of the form $(c_i, c_j, \varphi(c_i, c_j))$ for all $1 \leq i \leq n-1, i < j \leq n$. Subsequently, we sort the tuples with respect to the value of φ in ascending order and enqueue these tuples in a queue Q. Together with the number of classes $|C|$, Q is then provided as an input for the algorithm shown in Listing 1.

Initially, an array `nodeArray` is created that contains a leaf node for each class of the problem. The basic idea is now to create the binary tree bottom-up starting at the leaves. In the nodeArray, we maintain the root node of each subtree a class may already belong to. Iterating over the tuples contained in the queue, we retrieve the subtree roots of the respective classes from the array and combine these subtrees under a new root node if the classes are not already part of the same subtree. For all classes that are part of one of the two subtrees the entry in the nodeArray is updated accordingly. Finally, all entries of nodeArray contain the root node of the binary tree structure. Each inner node is assigned a base classifier completing the construction of the nested dichotomy.

In Figure 2, examples of genotypes and their corresponding nested dichotomies are shown for a 4-class problem with a set of classes {A,B,C,D}. In this example, we fixed the ordering on the classes to be A,B,C,D. Additionally, the genetic encoding of each individual is visualized in terms of a graph, illustrating the classes as nodes and edges between those nodes that are labeled with the values of φ. The processing done by the algorithm in Listing 1 is highlighted with blue borders surrounding the respective nodes. The corresponding inner nodes within the nested dichotomies are highlighted in blue as well.

The examples in Figure 2b illustrate offspring resulting from a crossover of the two individuals shown in Figure 2a. To obtain



(a) Genotypes representing balanced nested dichotomies

(b) Offspring resulting from applying crossover to the individuals in 2a

Figure 2: Examples of genotypes, visualization of pairwise distances as graphs, and corresponding nested dichotomies for a 4-class problem.

these individuals, the gene strings of the parents are cut into halves. Moreover, one can see that partial solutions, i.e., the subtree (A, C) respectively the subtree (A, B) have been transmitted from the parents, thus, partial solutions have been exchanged.

From the definition of genotypes and the construction of phenotypes, it is clear that every genotype representation corresponds to a valid ND and, moreover, each ND can be represented by a genotype. Since this representation is not unique, our mapping from the space of genotypes to the space of phenotypes is a surjection.

4.3 Fitness Function

For assessing the fitness of an individual, we only consider the performance of the phenotype. As we are mainly interested in the generalization performance of the NDs, i.e., the predictive accuracy, we use a variant of Monte Carlo cross-validation (MCCV). More concretely, we create random stratified splits with 90% training and 10% validation data on which the predictive accuracy of the ND is measured. We repeat this procedure 5 times.

Usually, the validation performance is then summarized taking the mean over the different splits. However, depending on the dataset, the performance for different splits of the data varies widely. Therefore, to increase robustness toward outliers, we take the trimmed instead of the standard mean, cutting off both the best and the worst measured value.

Another advantage of MCCV is that by drawing random splits each time to assess an individual's fitness, we prevents the latter from overfitting a fixed data split. Nevertheless, especially owing to the nature of optimizing the individuals, overfitting remains an omnipresent problem.

Note that the fitness function is actually not a proper function, since two independent evaluations of the same individual may lead to different fitness values. Instead, it should be seen as an approximation of the true fitness of an individual, namely the generalization performance of the ND it encodes.

4.4 Implementation

The actual evolutionary algorithm is an instantiation of NSGA-II [2], albeit with the fitness function described in the previous section as the only objective. For selection, we rely on the defaults of NSGA-II, performing a tournament with 2 individuals. In a tournament two individuals are compared according to their fitness value in the first place. If both individuals have the same fitness, the crowding distance comparator is used as a second criterion, as suggested in [2]. For recombination and mutation, we use standard genetic operators, i.e., single-point crossover with a probability of 0.9 and polynomial mutation with a probability of $2/(c(c-1))$, which leads to one mutated gene per individual in expectation. The source code of the implementation is publicly available[1].

4.5 Building Ensembles

To build bagged ensembles of size k, we do k independent evolutionary runs with different seeds. Obviously, these runs may be performed in parallel. The best individual of each run is then added to the ensemble. While each of the ensemble members is trained on the data independently, when making a prediction, the predictions of all the ensemble members are aggregated by averaging over the predicted probabilities for each class.

5 EVALUATION

In this section, we evaluate the nested dichotomies evolutionary algorithm (NDEA) as introduced in the previous section on 35 UCI datasets [13]. On one hand, we assess the feasibility of the proposed

[1] https://github.com/mwever/ndea.git

Dataset	#Classes	#Inst.	#Att.	Dataset	#Classes	#Inst.	#Att.	Dataset	#Classes	#Inst.	#Att.
anneal	6	898	39	arrhythmia	16	452	280	audiology	24	226	70
autos	7	205	26	balance.scale	3	625	5	car	4	1728	7
ecoli	8	336	8	glass	7	214	10	grub-damage	4	155	9
hypothyroid	4	3772	30	kropt	18	28056	7	led24	10	3200	25
letter	26	20000	17	lymph	4	148	19	mfeat-factors	10	2000	217
mfeat-fourier	10	2000	77	mfeat-karhunen	10	2000	65	mfeat-morphological	10	2000	7
nursery	5	12960	9	optdigits	10	5620	65	page-blocks	5	5473	11
pendigits	10	10992	17	primary.tumor	22	339	18	segment	7	2310	20
semeion	10	1593	257	shuttle	7	58000	10	soybean	19	683	36
splice	3	3190	62	vehicle	4	846	19	vowel	11	990	14
waveform5000	3	5000	41	wine	3	178	14	winequality	11	4898	12
yeast	10	1484	9	zoo	7	101	18				

Table 1: Description of UCI datasets used in the evaluation

genetic representation of nested dichotomies. On the other hand, we compare the predictive accuracy of the evolved nested dichotomies to several other approaches including the state-of-the-art random pairs nested dichotomies.

5.1 Configuration of NDEA

We instantiate NDEA with a population size to 16 individuals. The algorithm terminates as soon as at least one of the following three conditions holds: (1) The maximum of 200 generations is reached, (2) the best individual has not changed for 15 generations, or (3) a timeout for the evolutionary run occurs. For the experiments, we set the timeout to 7 minutes. Moreover, we reinitialize the population every 5 generations, keeping solely the best individual.

5.2 Experiment Setup

The evaluation of the NDEA is two-fold. First, the genetic representation is implicitly evaluated for its feasibility in general. Second, we compare the quality of solutions returned by the proposed method for nested dichotomies using three different base classifiers from the WEKA library [6, 10]. To achieve a broad coverage of the bias-variance spectrum, we use decision stumps, logistic regression, and decision trees (J48 in WEKA). While the former is an extremely simple classifier (high bias, low variance), the latter is able to create highly non-linear decision boundaries (low bias, high variance), and logistic regression is in-between these extremes.

We compare NDEA to the state-of-the-art random-pair nested dichotomies (RPND) [12], class balanced NDs (CBND), data-near-balanced NDs (DNBND) [3], furthest-centroid NDs (FCND) [4], and the original randomly sampled nested dichotomies (ND) [7]. The well-known reduction techniques one-vs-one (OvO) [9] and one-vs-rest (OvR) are considered as additional baselines.

Altogether, we evaluated a bagged ensemble of size 10 for each of the 8 methods for each dataset, performing 10 times a 10-fold cross validation. Thus, we conducted 2,800 experiments, and in each of these experiments, the base method for creating a single nested dichotomy has been invoked 100 times. The experiments were conducted on nodes with Intel Xeon E5-2670 CPUs having 8 cores, and the memory has been limited to 4GB.

To assess the statistical significance of the results, we use a rank-sum test, the so-called Mann-Whitney U Test [15]. We consider changes in the predictive accuracy as significant if $p < 0.05$.

5.3 Results

In Tables 2, 3, and 4, the results of the experiments for decision stumps, logistic regression, and decision trees are presented. The number of classes is given in parentheses next to the dataset names. Significant improvements of NDEA over other methods are highlighted with a •, significant degradations with a ◦. Best performances are highlighted in bold.

We start our evaluation with the results shown in Table 2, where decision stumps are used as a base classifier. For many datasets, it can be seen that NDEA is competitive to the other methods. For a few datasets, we observe that NDEA yields significantly degraded solutions, but most of these datasets have a rather small number of classes, i.e., the binary tree structure remains rather simple. For instance, in the case of 3 classes, ignoring the order of classes, there are only 3 different binary tree structures. Across all datasets, NDEA clearly dominates FCND and OvR, yielding significantly better solutions most of the time. Compared to CBND, DNBND, ND, and OvO, the number of significant improvements is smaller, but our approach is still preferable. Furthermore, NDEA yields significant improvements over the state-of-the-art approach RPND by Leathart et al. [12].

The results for logistic regression are presented in Table 3. In this setting, a clear dominance of NDEA can only be observed over FCND. Nevertheless, NDEA's performance is still preferable compared to CBND, DNBND, OvR, and even ND. OvO and RPND perform quite well and are rather competitive, but NDEA has still a slight edge over those two approaches.

In the experiments with decision trees as base learner, NDEA outperforms FCND as well as OvO and OvR. However, compared to CBND and DNBND, there is no clear winner, since the number of datasets with significant wins and losses is nearly balanced. Considering RPND and ND, although NDEA is performing clearly better than RPND, it performs significantly worse than ND on several datasets, most probably due to the effect of overfitting the respective data. This observation is somewhat surprising, as it

Dataset	CBND	DNBND	FCND	ND	OvO	OvR	RPND	NDEA
anneal (6)	87.13±0.94 •	85.82±0.38 •	85.38±0.35 •	88.0±0.73 •	89.08±0.14	89.15±0.09	88.75±0.40 •	**89.32±0.28**
arrhythmia (16)	64.91±0.66 •	60.09±1.05 •	69.45±1.43 •	67.37±1.31 •	**73.05±0.68**	70.15±1.02 •	72.83±0.82	72.52±0.46
audiology (24)	67.85±0.75	66.37±0.63	55.75±2.37	65.49±1.91	73.89±0.36	52.65±0.36	**75.07±0.21**	71.73±1.14
autos (7)	60.24±1.61	60.39±1.59	48.88±2.28 •	**60.59±1.34**	60.29±1.07	57.02±0.88 •	58.73±1.79	59.8±1.29
balance.scale (3)	**58.35±0.98**	58.11±0.74	58.32±1.03	58.21±1.11	57.58±0.62	57.86±0.75	57.74±0.68	57.58±0.62
car (4)	**70.02±0.00**	**70.02±0.00**	**70.02±0.00**	**70.02±0.00**	**70.02±0.00**	**70.02±0.00**	**70.02±0.00**	**70.02±0.00**
ecoli (8)	79.7±1.50	79.29±1.20	76.13±1.25 •	79.61±0.91	79.38±0.58	76.52±0.80 •	**80.09±0.73**	**80.09±1.58**
glass (7)	65.05±0.80 •	65.33±1.00 •	**68.41±1.26**	64.49±1.25 •	65.84±0.79 •	60.23±0.99 •	67.29±1.13	67.62±0.98
grub-damage (4)	**40.52±1.22**	**40.52±1.22**	34.65±1.85 •	39.61±2.04	39.81±2.33	39.74±2.40	40.13±2.43	40.45±2.04
hypothyroid (4)	97.66±0.10	**97.72±0.00**	96.68±0.02 •	97.69±0.05	97.58±0.05 •	97.71±0.02	**97.72±0.00**	**97.72±0.00**
kropt (18)	26.62±0.15	26.52±0.09 •	25.59±0.14 •	26.46±0.30 •	**27.12±0.04** ○	20.1±0.30 •	26.62±0.10	26.77±0.12
led24 (10)	68.53±0.53 •	67.47±1.15 •	58.33±0.78 •	67.63±0.81 •	67.33±0.13 •	46.76±0.11 •	69.24±0.48 •	**69.97±0.56**
letter (26)	56.36±0.49 ○	56.63±0.54 ○	35.84±0.34 •	51.55±0.66 •	**61.99±0.15** ○	30.31±0.13 •	57.57±0.29 ○	52.48±0.48
lymph (4)	77.03±0.80 •	77.3±0.33 •	77.36±1.49 •	77.7±1.00	**78.58±0.31**	76.96±0.47 •	**78.58±0.31**	78.45±0.20
mfeat-factors (10)	87.89±0.82 ○	**87.98±0.49** ○	72.7±0.35 •	87.84±0.39 ○	87.29±0.29 ○	77.89±0.39 •	87.23±0.56 ○	85.32±0.43
mfeat-fourier (10)	71.11±0.78	**71.69±0.83**	60.63±0.26 •	70.53±0.74	71.21±0.54	62.27±0.28 •	71.2±0.81	70.93±0.53
mfeat-karhunen (10)	**78.68±0.57** ○	78.58±0.81 ○	60.07±0.40 •	76.15±0.85 ○	76.88±0.43 ○	49.57±0.53 •	75.37±1.00	74.29±0.81
mfeat-morphological (10)	67.49±0.88 •	66.94±1.18 •	58.88±0.33 •	67.0±0.64 •	**70.19±0.24**	54.41±0.30 •	69.4±0.32	69.78±0.44
nursery (5)	76.01±0.51	76.25±0.28	**76.34±0.00**	**76.34±0.00**	**76.34±0.00**	66.25±0.00 •	**76.34±0.00**	**76.34±0.00**
optdigits (10)	**82.89±0.43** ○	82.85±0.55 ○	59.44±0.28 •	80.27±0.42	82.29±0.18 ○	54.04±0.36 •	79.38±0.52 •	80.38±0.67
page-blocks (5)	94.15±0.13 •	93.65±0.35 •	93.22±0.06 •	94.46±0.10 •	**95.55±0.06** ○	94.89±0.02 •	94.98±0.07 •	95.42±0.02
pendigits (10)	78.68±0.57 •	79.11±0.33 •	63.73±0.36 •	78.38±0.77 •	**81.62±0.11** ○	56.59±0.23 •	79.96±0.34	80.2±0.35
primary.tumor (22)	43.16±0.40	42.95±1.33	35.31±1.61 •	42.71±0.79 •	40.0±0.95 •	37.4±0.45 •	**45.96±1.24**	44.73±1.33
segment (7)	87.2±0.84	87.29±0.70	83.49±0.63 •	**88.57±0.64**	86.35±0.12 •	83.02±0.28 •	86.81±0.30 •	87.8±0.15
semeion (10)	**66.67±0.64** ○	66.35±0.76 ○	45.79±0.56 •	63.92±1.21	63.68±0.44	33.51±0.70 •	63.9±0.43	64.09±0.92
shuttle (7)	92.63±0.01 •	92.66±0.03	92.71±0.00 ○	92.66±0.03	**92.8±0.02** ○	92.63±0.00 •	92.74±0.02 ○	92.67±0.01
soybean (19)	78.9±1.04 •	78.01±1.04 •	60.24±0.68 •	81.6±0.82	**85.57±0.16** ○	74.54±0.39 •	82.84±0.80	82.05±0.66
splice (3)	79.99±1.82 ○	77.12±1.31 ○	56.11±0.25 •	80.0±1.38 ○	69.03±0.38 •	**81.41±0.00** ○	78.74±0.99 ○	73.98±0.00
vehicle (4)	48.77±1.66 •	48.77±1.66 •	50.65±0.29 •	**58.48±2.12** ○	53.62±0.81	50.34±0.74 •	58.07±1.01 ○	53.07±0.95
vowel (11)	50.6±1.12	50.99±0.80	43.19±0.89 •	49.26±0.91 •	**52.13±0.32**	40.19±0.46 •	51.16±0.67	51.02±1.02
waveform5000 (3)	69.68±0.69 ○	**69.88±0.42** ○	64.13±0.16 •	69.72±0.54 ○	68.08±0.26 ○	62.57±0.57 •	66.0±0.73	66.01±0.17
wine (3)	88.54±1.01	88.54±1.10	82.87±0.76 •	89.04±1.67	85.96±1.59 •	**90.39±1.02**	87.13±2.30	89.72±1.23
winequality (11)	50.29±0.48	49.98±0.47 •	44.95±0.59 •	50.51±0.45	47.8±0.34 •	**51.07±0.15** ○	47.61±0.41 •	50.81±0.18
yeast (10)	53.39±1.06 •	52.02±0.73 •	54.08±0.66 •	53.64±0.54 •	56.83±0.41	53.09±0.30 •	56.83±0.39	**57.08±0.63**
zoo (7)	88.42±1.83 •	90.3±1.81	74.75±2.04 •	91.88±1.07	90.89±0.59	**94.06±0.00** ○	91.88±1.07	90.89±0.86

Table 2: Accuracies (mean±std) of ensembles of 10 bagged nested dichotomies using decision stump as base classifier.

contradicts the results reported in [12], where no such significant degradations of RPND have been mentioned.

Altogether, we can conclude that the proposed genetic representation is suitable for optimizing the structure of nested dichotomies. Moreover, forming ensembles of nested dichotomies evolved with NDEA yields classifiers of superior predictive accuracy. Especially compared to OvO and OvR, such ensembles provide better reductions from multinomial to binary classification. NDEA even has a slight edge over the state-of-the-art approach RPND.

However, the improved predictive accuracy comes at the cost of runtime. As an ensemble of size 10 requires the same number of executions of NDEA, the runtime is multiplied by the ensemble size. Yet, as already mentioned, the evolutionary runs are independent of each other, and thus amenable to simple parallelization.

6 CONCLUSION

In this paper, we developed an evolutionary algorithm for learning ensembles of nested dichotomies, called NDEA. More specifically, by evolving the structure of NDs, we seek to find the most appropriate binary reductions of the original multi-class problem. To this end, we proposed a genetic representation based on pairwise distances between classes. This indirect representation allows for encoding individuals simply by a sequence of real values, the length of which is quadratic in the number of classes. Despite the simplicity of this representation, it facilitates the application of standard genetic operators. Furthermore, the encoding supports the exchange of partial solutions.

The main question we wanted to answer is whether an evolutionary approach to nested dichotomies is worthwhile, i.e., whether there is a potential to further improve on existing methods. The answer we can give to this question is affirmative. In fact, our experiments revealed that NDEA outperforms standard reduction techniques such as one-vs-one and one-vs-rest as well as other heuristics for optimizing the structure of dichotomies, for example based on clustering or balancing. Moreover, NDEA even has a slight edge over the state-of-the-art random pair approach, albeit at the cost of an increased runtime (as the evolutionary algorithm needs to train and test NDs repeatedly). In this regard, we observed that, the simpler the base learner, the more pronounced the improvements achieved by NDEA. This is plausible, because simple classifiers have

Dataset	CBND	DNBND	FCND	ND	OvO	OvR	RPND	NDEA
anneal (6)	**99.59±0.16**	99.4±0.24	99.15±0.26 •	99.58±0.16	99.45±0.18	99.51±0.14	**99.59±0.14**	99.58±0.17
arrhythmia (16)	58.63±1.37	56.75±1.21	47.9±2.28 •	57.41±1.00	**61.06±1.20** ○	53.14±1.23 •	56.22±1.65	57.64±2.43
audiology (24)	84.37±0.55	81.42±1.08 •	70.5±0.42 •	83.92±1.27	77.3±1.34 •	75.88±1.71 •	83.33±0.91	**84.51±0.76**
autos (7)	74.24±1.74	74.29±1.76	71.07±2.12 •	73.41±1.30	**75.32±1.58**	66.63±2.00 •	73.61±1.08	74.83±1.38
balance.scale (3)	86.69±0.61 •	86.37±0.43 •	**90.77±0.35**	86.54±0.48 •	90.74±0.32	85.95±0.49 •	90.75±0.33	90.58±0.52
car (4)	92.82±0.39 •	93.7±0.18 •	**94.07±0.15**	92.83±0.46 •	93.92±0.16	90.16±0.14 •	93.67±0.23 •	94.0±0.11
ecoli (8)	86.46±0.68	**86.82±0.42** ○	84.97±0.47 •	86.07±0.42	85.48±0.74	86.28±0.56	85.89±0.50	85.83±0.79
glass (7)	64.49±0.96	**64.81±1.22**	61.96±0.73	63.93±1.73	62.15±1.20	64.07±0.77	63.18±1.00	63.32±1.52
grub-damage (4)	**43.42±1.58**	**43.42±1.58**	42.65±1.79	42.71±2.41	41.81±1.15	41.94±2.40	43.16±2.57	41.87±2.33
hypothyroid (4)	96.55±0.36 •	96.82±0.20 •	97.61±0.07	96.72±0.27 •	**97.65±0.12**	95.12±0.15 •	97.54±0.11	**97.65±0.08**
kropt (18)	31.64±0.13 •	31.38±0.11 •	33.18±0.08 •	32.04±0.31 •	**35.18±0.09** ○	31.69±0.07 •	33.73±0.09 •	34.16±0.17
led24 (10)	72.6±0.16	72.54±0.24	72.28±0.18	72.55±0.17	71.69±0.30 •	**72.76±0.13** ○	72.57±0.14	72.47±0.22
letter (26)	73.71±0.45 •	73.72±0.29 •	72.25±0.13 •	74.41±0.21 •	**84.6±0.06** ○	72.39±0.03 •	79.22±0.19 ○	77.38±0.18
lymph (4)	78.99±2.55	78.11±2.22	78.18±1.25	79.19±2.13	**79.93±1.68**	77.84±2.70	79.59±2.17	78.92±2.13
mfeat-factors (10)	97.84±0.21 •	97.82±0.14 •	96.71±0.21 •	98.02±0.20	96.92±0.18 •	96.77±0.26 •	98.01±0.09	**98.25±0.23**
mfeat-fourier (10)	82.54±0.49 •	82.5±0.50 •	75.54±0.55 •	82.37±0.44 •	80.95±0.35 •	77.7±0.29 •	83.1±0.54	**83.3±0.40**
mfeat-karhunen (10)	95.03±0.32 •	95.13±0.27 •	90.94±0.51 •	94.92±0.29 •	93.84±0.23 •	89.22±0.35 •	95.03±0.31 •	**95.8±0.27**
mfeat-morphological (10)	72.67±0.61 •	72.59±0.76 •	70.41±0.43 •	73.47±0.38 •	73.71±0.55	73.77±0.31	73.51±0.49	**74.11±0.48**
nursery (5)	92.45±0.14	92.43±0.15	92.52±0.03	92.41±0.04 •	**92.59±0.05** ○	91.69±0.05 •	92.5±0.03	92.52±0.04
optdigits (10)	96.7±0.14 •	96.77±0.22 •	91.99±0.18 •	96.96±0.11 •	**97.31±0.15**	94.83±0.09 •	97.06±0.16	97.19±0.15
page-blocks (5)	95.85±0.08 •	95.61±0.07 •	95.72±0.08 •	96.07±0.10 •	**96.73±0.09** ○	95.76±0.07 •	96.33±0.09 •	96.51±0.07
pendigits (10)	94.56±0.16 •	94.57±0.18 •	87.97±0.08 •	95.09±0.17 •	**97.73±0.05** ○	93.56±0.06 •	95.41±0.21 •	96.69±0.09
primary.tumor (22)	45.81±1.29	**46.17±0.97**	39.47±1.46 •	45.6±1.62	40.0±0.89 •	42.6±1.33 •	44.31±1.58	45.06±0.88
segment (7)	94.07±0.38 •	93.88±0.25 •	88.87±0.12 •	94.21±0.34 •	95.96±0.14	92.34±0.15 •	95.48±0.29 •	**96.06±0.15**
semeion (10)	87.94±0.51	87.65±0.55	71.58±1.24 •	85.29±0.56 •	**90.57±0.36** ○	74.54±0.42 •	88.0±0.73	88.02±0.65
shuttle (7)	95.01±0.42 •	96.04±0.13 •	96.84±0.04 •	95.02±0.35 •	**97.42±0.06** ○	93.34±0.01 •	96.91±0.03	96.94±0.01
soybean (19)	94.05±0.50	94.05±0.50	91.62±0.52 •	**94.06±0.42**	92.71±0.46 •	92.01±0.65 •	93.95±0.47	93.64±0.52
splice (3)	**92.8±0.40** ○	91.62±0.35 ○	92.18±0.31 ○	92.32±0.42 ○	90.48±0.26 •	91.82±0.18 ○	91.74±0.35 ○	90.87±0.33
vehicle (4)	79.27±0.69	79.27±0.69	79.16±0.50	**80.01±0.58** ○	79.17±0.50	79.18±0.39	79.82±0.55	79.23±0.54
vowel (11)	79.74±1.69 •	79.82±1.35 •	80.53±0.67 •	79.79±1.52 •	**90.46±0.47** ○	65.85±0.49 •	89.39±0.65	89.28±0.49
waveform5000 (3)	86.46±0.22 ○	86.47±0.24 ○	84.38±0.10 •	86.28±0.27 ○	86.5±0.13 ○	**86.8±0.14** ○	85.92±0.28 ○	85.15±0.32
wine (3)	96.97±0.88	96.97±0.84	95.73±0.80 •	96.91±0.80	96.57±0.89	96.69±0.89	96.07±0.87 •	**97.58±0.44**
winequality (11)	**53.81±0.14** ○	53.7±0.18	53.13±0.15 •	53.71±0.18	53.55±0.13	53.5±0.06	53.29±0.11 •	53.52±0.14
yeast (10)	58.72±0.59 •	58.73±0.65 •	58.98±0.38 •	58.92±0.41 •	58.98±0.37 •	58.42±0.25 •	59.13±0.50 •	**59.84±0.47**
zoo (7)	**94.95±1.03**	94.65±0.79	87.72±1.48 •	94.75±1.09	93.27±0.59 •	89.8±1.77 •	94.85±0.59	94.75±1.09

Table 3: Accuracies (mean±std) of ensembles of 10 bagged nested dichotomies using logistic regression as base classifier.

a restricted ability to solve difficult binary problems—for them, a good binary reduction is therefore even more important than for flexible classifiers. Nevertheless, the interplay between ND structure optimization and the choice of the base learner ought to be investigated in more depth.

In this regard, and motivated by recent advances in algorithm selection [5, 17, 19], we also plan to extend our approach toward a more global hyperparameter optimization for NDs. While the ND structure can be seen as one such hyperparameter, the base learners at the inner nodes of an ND tree are parametrized, too. Since we only used default parameters in our experiments, there is obviously scope for further improvement. In this regard, one may even think of selecting different base learners for different nodes. Last but not least, there is of course also scope to improve our evolutionary algorithm itself. For example, instead of creating an ensemble by running several evolutionary processes independently of each other, it would be more efficient (and perhaps also more effective) to compose an ensemble from a single population. This, of course, requires an appropriate adaptation of strategies for fitness evaluation and selection of individuals.

ACKNOWLEDGMENTS

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Center "On-The-Fly Computing" (SFB 901).

REFERENCES

[1] Leo Breiman. 1996. Bagging Predictors. *Machine Learning* 24, 2 (1996), 123–140.
[2] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evolutionary Computation* 6, 2 (2002), 182–197.
[3] Lin Dong, Eibe Frank, and Stefan Kramer. 2005. Ensembles of Balanced Nested Dichotomies for Multi-class Problems. In *Knowledge Discovery in Databases: PKDD 2005, 9th European Conference on Principles and Practice of Knowledge Discovery in Databases, Porto, Portugal, October 3-7, 2005, Proceedings.* 84–95.
[4] Miriam Mónica Duarte-Villaseñor, Jesús Ariel Carrasco-Ochoa, José Francisco Martínez Trinidad, and Marisol Flores-Garrido. 2012. Nested Dichotomies Based on Clustering. In *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications - 17th Iberoamerican Congress, CIARP 2012, Buenos Aires, Argentina, September 3-6, 2012. Proceedings.* 162–169.
[5] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. [n. d.]. Efficient and Robust Automated Machine Learning. In *Advances in Neural Information Processing Systems 28.*
[6] Eibe Frank, Mark A. Hall, Geoffrey Holmes, Richard Kirkby, and Bernhard Pfahringer. 2005. WEKA - A Machine Learning Workbench for Data Mining. In *The Data Mining and Knowledge Discovery Handbook.* 1305–1314.

Dataset	CBND	DNBND	FCND	ND	OvO	OvR	RPND	NDEA
anneal (6)	98.67±0.20 •	98.7±0.17 •	98.56±0.20 •	98.76±0.25	98.57±0.18 •	**99.14±0.23**	98.44±0.21 •	99.0±0.25
arrhythmia (16)	72.7±1.07	71.84±0.99	66.19±1.39 •	**73.05±0.88**	67.81±1.03 •	64.71±1.52 •	71.13±1.01	71.86±1.01
audiology (24)	**80.27±1.37**	80.04±0.78	74.78±1.28 •	78.94±0.95	78.58±0.75	70.13±1.39 •	78.45±1.28	79.34±1.07
autos (7)	75.22±2.24 •	76.15±1.56 •	68.78±1.41 •	78.83±1.45	70.73±1.73 •	77.12±0.83 •	73.85±1.90 •	**79.66±1.66**
balance.scale (3)	79.63±0.73	79.57±0.91	**80.02±0.35**	79.65±0.72	79.23±0.96	78.32±0.66	79.81±0.89	79.39±0.97
car (4)	92.95±0.35	93.5±0.41	92.53±0.32 •	92.86±0.35	**94.2±0.27** ○	92.46±0.44 •	92.67±0.37	93.17±0.38
ecoli (8)	85.12±0.81	**85.42±1.12**	82.38±1.35 •	85.12±1.05	83.72±0.80	82.14±0.83 •	83.87±1.26	84.14±1.03
glass (7)	74.02±2.04	73.93±1.80	70.98±1.73 •	**75.0±1.88**	73.88±1.61	70.33±3.27	71.4±1.59	73.83±1.94
grub-damage (4)	**46.77±2.05**	**46.77±2.05**	38.65±1.84 •	45.1±2.32	41.35±2.09 •	44.58±1.72	44.0±2.79	45.61±1.85
hypothyroid (4)	99.54±0.04	**99.55±0.05**	99.5±0.03	99.52±0.07	99.44±0.05	99.48±0.05	99.53±0.03	99.51±0.04
kropt (18)	75.35±0.18 •	75.36±0.18 •	69.31±0.26 •	75.71±0.27 •	73.28±0.15 •	66.34±0.20 •	75.97±0.19	**76.4±0.17**
led24 (10)	72.42±0.26	72.5±0.27	71.77±0.19 •	**72.52±0.30**	72.21±0.22	69.64±0.39 •	72.47±0.28	72.38±0.31
letter (26)	**94.93±0.15** ○	**94.93±0.11** ○	86.67±0.16 •	94.73±0.06 ○	91.49±0.16 •	86.95±0.10 •	94.27±0.09	94.45±0.15
lymph (4)	77.77±2.76	76.96±2.35	77.5±2.58	77.7±1.45	78.04±2.81	76.96±1.90	**78.72±1.69**	76.82±2.47
mfeat-factors (10)	**96.08±0.28** ○	95.96±0.31 ○	88.6±0.60 •	95.68±0.28 ○	91.65±0.32 •	87.48±0.50 •	94.62±0.42	94.93±0.30
mfeat-fourier (10)	82.66±0.35	**82.67±0.59**	74.21±0.70 •	82.35±0.49	78.51±0.45 •	71.93±0.46 •	81.6±0.58	82.28±0.54
mfeat-karhunen (10)	**93.54±0.35** ○	93.44±0.29 ○	82.52±0.75 •	93.51±0.39 ○	88.53±0.45 •	81.07±0.55 •	91.9±0.48 •	92.7±0.37
mfeat-morphological (10)	73.11±0.19 •	73.17±0.31 •	72.28±0.27 •	73.73±0.24	72.32±0.35 •	72.95±0.55 •	73.46±0.43	**73.74±0.34**
nursery (5)	97.19±0.12	97.19±0.11	97.14±0.10	97.19±0.11	97.13±0.07	**97.21±0.10**	97.17±0.08	97.17±0.10
optdigits (10)	**97.31±0.11** ○	97.21±0.12 ○	90.77±0.19 •	97.12±0.20 ○	94.56±0.10 •	90.19±0.27 •	96.97±0.14 ○	96.74±0.12
page-blocks (5)	**97.24±0.10**	97.12±0.17	97.03±0.15	**97.24±0.09**	97.12±0.14	96.98±0.13 •	97.16±0.14	97.22±0.13
pendigits (10)	**98.78±0.07** ○	98.75±0.06 ○	95.8±0.12 •	98.68±0.06 ○	96.83±0.08 •	95.43±0.15 •	98.32±0.11 •	98.48±0.10
primary.tumor (22)	**47.08±1.02** ○	46.87±1.10 ○	40.27±1.15 •	45.81±1.53	43.19±0.82 •	39.82±0.78 •	43.83±0.96	44.87±0.81
segment (7)	97.72±0.20	97.82±0.25	96.6±0.30 •	97.8±0.16	96.72±0.21 •	95.45±0.29 •	97.41±0.15 •	**97.93±0.21**
semeion (10)	**90.85±0.47**	90.65±0.42	78.02±0.78 •	90.54±0.60	85.37±0.50 •	76.15±0.83 •	90.35±0.57	90.34±0.52
shuttle (7)	**99.98±0.00**	99.97±0.01 •	**99.98±0.00**	**99.98±0.00**	99.98±0.00	99.97±0.00	**99.98±0.00**	**99.98±0.00**
soybean (19)	**94.83±0.33**	94.42±0.49	92.14±0.42 •	94.04±0.45 •	93.82±0.37 •	90.98±0.39 •	94.25±0.42	94.77±0.54
splice (3)	94.46±0.43 •	93.02±0.22 •	91.03±0.50 •	94.71±0.36	94.71±0.12 •	94.72±0.19	**94.92±0.15**	**94.92±0.09**
vehicle (4)	**73.45±1.30**	**73.45±1.30**	70.12±0.65 •	72.96±0.98	71.52±0.45	70.22±1.08 •	72.04±1.05	72.62±1.45
vowel (11)	**91.27±0.73**	90.94±0.68	76.45±1.00 •	90.85±0.82	81.74±0.67 •	78.84±1.13 •	86.47±0.87 •	90.36±0.59
waveform5000 (3)	77.65±0.45 ○	**77.71±0.48** ○	75.37±0.47	77.53±0.28 ○	75.78±0.31	72.82±0.50 •	75.74±0.41	75.67±0.47
wine (3)	**93.31±1.47**	92.92±0.91	91.24±1.71	92.81±1.39	92.02±1.42	89.33±1.55 •	92.19±0.64	92.25±1.84
winequality (11)	63.93±0.29 ○	63.39±0.45 ○	57.76±0.58 •	**64.01±0.45** ○	58.8±0.47 •	59.45±0.92 •	62.05±0.43 •	62.76±0.36
yeast (10)	60.9±0.50	**61.06±0.73**	58.06±0.71 •	60.68±0.47	58.85±0.76 •	57.68±0.43 •	59.82±0.52 •	60.53±0.33
zoo (7)	92.97±0.53	93.07±0.89	87.62±1.42 •	**93.17±1.03**	91.29±0.40 •	92.18±1.12	92.38±0.63	**93.17±0.93**

Table 4: Accuracies (mean±std) of ensembles of 10 bagged nested dichotomies using decision trees as base classifier.

[7] Eibe Frank and Stefan Kramer. 2004. Ensembles of nested dichotomies for multi-class problems. In *Machine Learning, Proceedings of the Twenty-first International Conference (ICML 2004), Banff, Alberta, Canada, July 4-8, 2004.*

[8] Yoav Freund and Robert E. Schapire. 1996. Game Theory, On-Line Prediction and Boosting. In *Proceedings of the Ninth Annual Conference on Computational Learning Theory, COLT 1996, Desenzano del Garda, Italy, June 28-July 1, 1996.* 325–332.

[9] Johannes Fürnkranz. 2002. Round Robin Classification. *Journal of Machine Learning Research* 2 (2002), 721–747.

[10] Mark A. Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. 2009. The WEKA data mining software: an update. *SIGKDD Explorations* 11, 1 (2009), 10–18.

[11] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. 2009. ParamILS: An Automatic Algorithm Configuration Framework. *J. Artif. Intell. Res.* 36 (2009), 267–306.

[12] Tim Leathart, Bernhard Pfahringer, and Eibe Frank. 2016. Building Ensembles of Adaptive Nested Dichotomies with Random-Pair Selection. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2016, Riva del Garda, Italy, September 19-23, 2016, Proceedings, Part II.* 179–194.

[13] M. Lichman. 2013. UCI Machine Learning Repository. (2013). http://archive.ics.uci.edu/ml

[14] Yuri Malitsky. 2015. Instance-specific algorithm configuration. *Constraints* 20, 4 (2015), 474.

[15] H. B. Mann and D. R. Whitney. [n. d.]. ([n. d.]).

[16] Juan José Rodríguez, César Ignacio García-Osorio, and Jesús Maudes. 2010. Forests of nested dichotomies. *Pattern Recognition Letters* 31, 2 (2010), 125–132.

[17] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2013. Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013.* 847–855.

[18] Geoffrey I. Webb. 2000. MultiBoosting: A Technique for Combining Boosting and Wagging. *Machine Learning* 40, 2 (2000), 159–196.

[19] Marcel Wever, Felix Mohr, and Eyke Hüllermeier. 2017. Automatic Machine Learning: Hierarchical Planning Versus Evolutionary Optimization. In *Proceedings of the 27. Workshop Computational Intelligence.* Springer.

9 Predicting Machine Learning Pipeline Runtimes in the Context of Automated Machine Learning

Declaration of the specific contributions of the author The author contributed to the publication *Predicting Machine Learning Pipeline Runtimes in the Context of Automated Machine Learning* by discussing intermediate experiments and results thereof. Furthermore, he implemented a composed runtime predictor for integrating it into ML-Plan. He also contributed to the publication by conducting the experiments to evaluate the benefit of this integration for the AutoML task.

Predicting Machine Learning Pipeline Runtimes in the Context of Automated Machine Learning

Felix Mohr, Marcel Wever, Alexander Tornede, Eyke Hüllermeier

Abstract—Automated Machine Learning (AutoML) seeks to automatically find so-called machine learning pipelines that maximize the prediction performance when being used to train a model on a given dataset. One of the main and yet open challenges in AutoML is an effective use of computational resources: An AutoML process involves the evaluation of many candidate pipelines, which are costly but often ineffective because they are canceled due to a timeout. In this paper, we present an approach to predict the runtime of two-step machine learning pipelines with up to one pre-processor, which can be used to anticipate whether or not a pipeline will time out. Separate runtime models are trained offline for each algorithm that may be used in a pipeline, and an overall prediction is derived from these models. We empirically show that the approach increases successful evaluations made by an AutoML tool while preserving or even improving on the previously best solutions.

Index Terms—automated machine learning, runtime prediction for classifiers and pipelines, hierarchical runtime prediction

1 Introduction

Automated Machine Learning (AutoML) is concerned with the automatic construction of algorithmic solutions to machine learning tasks, so-called machine learning pipelines, which are specifically tailored to a given dataset. The field has gained increasing attention over the last years, and numerous approaches have been presented [1]–[8].

Regardless of the optimization technique used, the overwhelming majority of approaches involve the training and validation of candidate pipelines. Such executions deliver important estimates of the quality of pipelines, but can also be quite expensive in terms of execution time.

AutoML tools often *waste* a good part of their time budget in executions canceled due to a *timeout*. Timeouts are upper bounds on the CPU time a pipeline execution may consume to prevent the search process from getting stuck in the evaluation of an expensive candidate.Analyzing the evaluations in [5], we found that between 20% and 60% of the CPU time is completely lost in evaluations that time out before a result is returned. This corresponds to an absolute total loss of 34 minutes (of one hour, i.e., 56%) per dataset on average, a total loss of 400 CPU hours in that evaluation.

To our knowledge, no existing approach rejects executions that are believed to hit the timeout. An early approach has been to estimate runtimes of a whole grid search instead of single executions [9]. Some AutoML approaches incorporate runtime estimates when *choosing* the next evaluation candidates, trading runtime against expected quality. Some Bayesian optimizers use an implicit runtime model, which is then part of the objective function [10], [11]. We know of two approaches with explicit runtime models for learners [12], [13]. However, both approaches consider learners to be monolithic and also ignore their parameters. Hence, they can neither generalize over different parametrizations nor over composed learners, in particular pipelines.

Focusing on two-step classification pipelines, our contribution is a compositional approach to *predict* pipeline timeouts. In this paper, we not only admit algorithm parameters, but also allow classification algorithms to be *wrapped* into meta-learners and combined with a pre-processing algorithm. We solve the prediction task by training regression models *only* for pre-processors and base classifiers whose predictions are then requested and *aggregated* into a runtime prediction for meta-learners and complete pipelines. The focus on classification is a bit arbitrary; we expect the approach to also work for other data-processing pipelines such as regression or multi-label classification.

Our experimental results are diverse. First, we find that the basic models predict runtimes with quite a moderate error for almost all of the considered algorithms, yielding mostly correct decisions. Second, including observations collected earlier in the AutoML process improves this performance significantly. Next, we show that the compositional approach substantially outperforms a single non-decompositional model in the form of a random forest regressor. Finally, we analyze the effect of the model when being used in the AutoML tool ML-Plan [5]. For datasets with data matrices of above 1 million entries, the *number of* and *time spent on* successful evaluations is increased by over 400% and 200%, respectively, on average. While higher efficiency aims at saving resources and will not generally imply better models produced by the AutoML tool, in some cases such improvements can be observed as well.

For the sake of readability, we subsequently interleave conceptual parts and results. Following a formalization of the problem in Section 2, we give a *conceptual* overview of our solution in Section 3. Our approach is built on top of three independent building blocks to predict base algorithm runtimes, meta-learner runtimes, and the change of dataset meta-features, respectively. Each of them is described and evaluated separately in dedicated Sections 4-6. In Section 7, we merge these building blocks into a runtime predictor for complete pipelines. Finally, Section 8 describes integration aspects of the model into an AutoML tool and evaluates the overall performance improvement for the case of ML-Plan.

2 Problem Definition

All state-of-the-art AutoML approaches guide the search using performance estimates obtained by *evaluating* pipelines [1]–[8]. Here, by evaluation we mean that the pipeline is *trained* with a portion of the given data and *validated* with another portion of the data disjoint from the training data. Typically, this validation is repeated several times in a standard or hold-out cross-validation, and scores are averaged.

In this paper, we focus on search spaces covering two-step pipelines in which meta-learners, if used, must be homogeneous. Two-step pipelines consist of a learner that *may* be preceded by a pre-processing algorithm such as normalization or a principal component analysis. The learners can be base learners such as decision trees or neural networks, or homogeneous meta-learners like adaptive boosting, which rely on several *instances* of a single base learner to make their predictions. We do not admit *heterogeneous* meta-learners that use different base learners, say a decision tree and a support vector machine, at the same time. All algorithms, i.e. pre-processors as well as base-learners and meta-learners, may potentially be customized via parameters, and we consider pipelines to be composed of such *parametrized* algorithms. We denote as $\mathcal{P}$ the space of descriptions of such two-step pipelines, each element of which describes the used algorithms and their parameters.

To predict whether the evaluation of such a pipeline will time out, we predict its *runtime* via *regression*, and then pass this prediction together with the time-bound to some decision rule $\phi : \mathbb{R} \times \mathbb{R} \to \{0, 1\}$. The simplest rule is to multiply the predicted runtime with the number of executions required by the cross-validation, and check whether it exceeds the time bound. Although more sophisticated rules are clearly conceivable, we commit to that one in this paper.

Reducing the "timeout or no timeout" question to a regression task has two important advantages over learning a binary classification model. First, in a regression-based solution, the learned model does not depend on the time-bound the user configures in the AutoML tool. Using classification, each bound would give rise to a new learning problem. Second, a regression approach can be decompositional: Predicting runtimes of the individual components of a pipeline, and suitably aggregating them, prior knowledge about the structure of pipelines can be incorporated, and useful information can be shared among pipelines with common components. It is unclear how these advantages coming from a decompositional view could be reasonably exploited via classification.

Formally, we search for a fast-to-evaluate function

$$f : \mathcal{P} \times \mathcal{D} \times \mathcal{D} \times \mathcal{H} \to \mathbb{R}\,, \tag{1}$$

where $f(p, d_1, d_2, h)$ is the prediction of the time needed to train a pipeline p with data d_1 (training data) and make predictions for instances in d_2 (evaluation data), taking into account previous observations h. Here, $\mathcal{P}$ is a space of pipeline descriptions, $\mathcal{D}$ is the space of dataset descriptions, and $\mathcal{H}$ is the space of histories of runtime observations.

The objective is to maximize prediction quality with respect to the question of whether or not a pipeline will run into a timeout. That is, given training data of previous pipeline executions in different contexts, we aim to find

$$\arg\min_f \mathbb{E}\left[\frac{1}{|D_{val}|}\sum_{(x,y)\in D_{val}} [\![[\![y \leqslant t]\!] = \phi(f(x \oplus h), t)]\!]\right]$$

where ϕ encodes the fix decision rule introduced above, $[\![\alpha]\!] = 1$ iff condition α is true (else 0), $\oplus$ is the vector concatenation, and expectation is taken with respect to all random splits of the available data D into D_{train}, which is used to create f, and $D_{val} = D \backslash D_{train}$, time bounds t, and observation histories h. The data D associates vectors $x = (p, d_1, d_2)$ consisting of a description of a pipeline p and training and validation data d_1 and d_2 with the evaluation time $y \in \mathbb{R}$ of p on d_1 and d_2. In other words, we assess our *regression solution* f by its error rate in the original *timeout classification problem*.

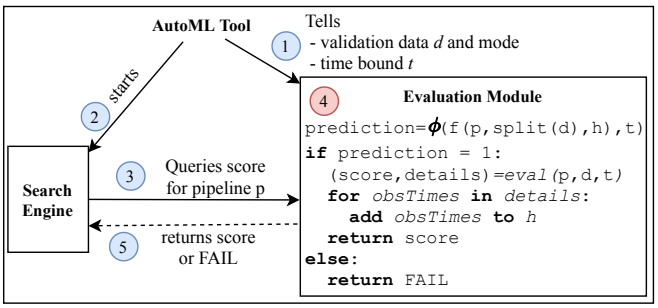


Fig. 1: The predictor resides inside the evaluation module and is hence isolated from the search engine.

Note that a solution to this problem applies to *every* AutoML tool relying on executions. Fig. 1 illustrates an integration of this predictor into the search process. With the guard function f in place, the evaluation module, instead of blindly executing the requested pipeline p, first solicits a prediction about the success of its execution and then either executes p and returns the results (memorizing the runtimes in the history) or returns with a failure. The functionality of the predictor is hence completely part of the evaluation module and independent of the underlying search engine.

3 A Composed Runtime Predictor

Solving the runtime prediction problem (1) directly for whole pipelines has several disadvantages. First, the pipelines in $\mathcal{P}$ are given in a structural and not in a vectorial representation, as required by classical regression algorithms. Squeezing a structural description into a vector [1], [2] yields a very high-dimensional regression problem — avoiding such constructions already motivated grammar-based approaches such as TPOT [3] and ML-Plan [5] for the AutoML task itself. Second, a direct approach would need to learn relationships that are not only implicitly contained in the data but actually given as prior knowledge. For example, if we know the runtime for a PCA and a decision tree, we can just *compute* the runtime of a pipeline connecting the two, knowing that the total runtime will be the sum. Ignoring this structural knowledge in the prediction imposes an unnecessary burden on the learner. Third, it is unclear how the previous knowledge from $\mathcal{H}$ could be included.

Instead, we propose a *compositional* prediction model. The idea, which is very much in line with the notion of "learning to aggregate" [14], is to create one prediction model for every *atomic algorithm*, i.e., pre-processors and base learners, and to derive all other runtimes, e.g., of meta-learners and the pipeline itself, using these common models.

The regression problem of a parametrized atomic algorithm is modeled as usual for runtime prediction problems [15]. An algorithm with n parameters is characterized by a vector $\theta = (\theta_1, \ldots, \theta_n) \in \times_{i=1}^{n} \Theta_i =: \Theta$, where $\Theta_1, \ldots, \Theta_n$ are *numeric* or *categorical* domains. The input data of the algorithm is characterized in terms of m *numeric features* $F = (F_1, \ldots, F_m) \in \times_{i=1}^{m} \mathcal{F}_i =: \mathcal{F}$, the so-called *dataset (meta-) features*. Typical dataset features include the number of instances, attributes, or classes, but many more can be used. A concrete *run* of the algorithm can be characterized by a joint vector $\theta \oplus F = (\theta_1, \ldots, \theta_k, F_1, \ldots, F_m)$ of features describing the parametrization and the input.

As machine learning pipelines have a very simple structure, it is straight forward to determine their overall runtime given the runtimes of their components. In general, they only allow for sequential or parallel executions of other algorithms but usually do not contain conditional branches let alone loops. Here, we only consider sequential pipelines, for which the overall runtime of the pipeline is simply the sum of the runtimes of the components.

Two main issues remain open on the conceptional level:

1) *How to obtain predictions for meta-learners?* Just as for the whole pipeline, models of atomic algorithms can be reused to predict meta-learner runtimes. But, in contrast to the pipeline, we do not know the flow structure of meta-learners, and it is not clear how the base learner models should be used.
2) *How to determine the dataset features $\mathcal{F}$ for models of algorithms that do not receive the original data as inputs?* If a feature selector is followed by a decision tree, we must use the data left by the feature selector to predict the runtime of the decision tree. But it is not clear how this data looks like until execution.

We defer the detailed answers to these questions to the following sections. On a high level, the idea is to learn separate models for these two cases. Regarding meta-learners, the idea is to learn a relationship between their configuration and the number of induced base learners and their inputs. With such models, one can estimate the overall runtime of a meta-learner given its configuration and an estimate of the runtime of the used base learner. Section 6 covers the detailed explanation of this part. Regarding the second point, the idea is to maintain a model for each dataset meta-feature and each pre-processing algorithm that predicts how that algorithm will *change* the respective meta-feature. The detailed procedure is described in Section 5, preceded by Section 4 elaborating on the main building block, i.e. the runtime estimation of the atomic algorithms.

The empirical evaluations in the following sections are based on algorithms from the WEKA library [16] and datasets from openml.org [17]. We list the relevant algorithms in the respective sections and provide a brief description of each of them in the supplement. We consider 170 datasets from openml.org as the *basis* of the datasets used in the evaluation. This is a strict superset of the datasets used in [2], which is a systematic selection of datasets with heterogeneous properties. The supplement lists details of these datasets. In total, we conducted 8.4 million experiments on computation units of 4 cores (Intel Xeon E5-2670, 2.6Ghz) with 16GB memory. In this setup, the experiments accumulated a total runtime of over 60 CPU years and results occupying a volume of over 600GB disk space.

4 Predicting Atomic Algorithm Runtime

The runtime prediction problems for *atomic* algorithms, by which we refer to pre-processors and base learners, can be phrased as regression problems with (up to) three types of variables on which the prediction can be based. Following the notation of Section 3, the independent variables may comprise (i) the algorithm parameters θ, (ii) any subset of dataset meta-features $\mathcal{F}$, and, in addition to those conventional features, (iii) previous observations $\mathcal{H}$. The two target variables are the total *train time* (in ms) and the time required to make $1,000$ *predictions*, the latter of which is motivated by the fact that the prediction time is always linear in the number of predictions to be made so that it is enough to estimate the slope of the runtime function. While we can simply use the parameters of the algorithms as-is for θ, the choices of dataset meta-features $\mathcal{F}$ and inclusion of previous observations $\mathcal{H}$ is less obvious.

This section gives insights into the runtime prediction of atomic algorithms in four steps. First, we discuss how well the runtime can be described only in terms of numbers of instances and attributes (a natural and simple choice), and which regressor would be good for this choice of $\mathcal{F}$. In the second step, we analyze the prediction performance for this $\mathcal{F}$ over a broad range of 170 datasets and 30 atomic algorithms. Third, we examine whether errors in the predictions lead to wrong pruning decisions. Finally, we analyze whether these models can be adjusted under particular conditions to improve performance.

The first question addresses an optimization problem on the meta-level. Clearly, we will want to include the meta-features and use the regressors that obtain the best runtime prediction performance. In order to later assess this prediction performance in an unbiased way, i.e., to avoid overfitting on a meta-level, this question is answered only using a representative selection of 10 of the 170 datasets. We refer to the supplement for details on the selection process.

The analysis of the remaining questions is then based on the results of the first one. In all models, we use the dataset meta-features and regressors identified in the first step. However, while the second question combines those meta-features with the features resulting from the algorithm parameters, the fourth question aims at assessing the regressor performance if we *only* use the dataset meta-features (and no algorithm description, assuming a default parametrization) or if we *add* another feature for the performance of the respective algorithm under default parametrization.

4.1 Choice of Dataset Meta-Features and Regressor

This section aims at answering the question of which regressor achieves high-quality runtime prediction performance

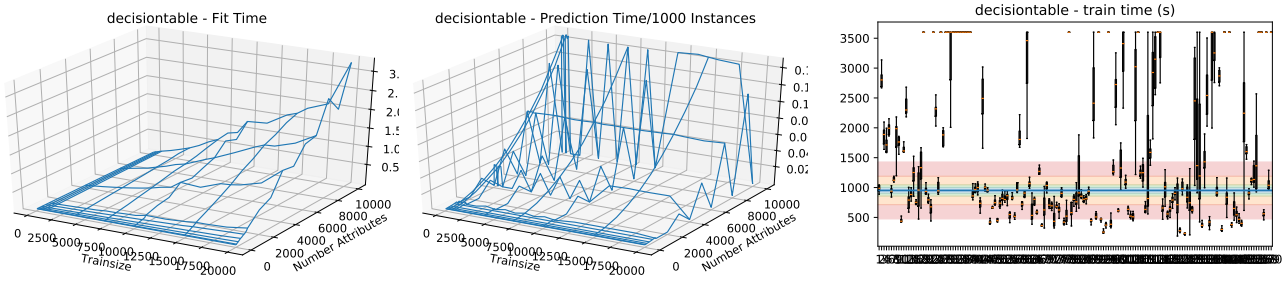


Fig. 2: Runtimes of DecisionTable.

and which dataset meta-features need to be used for this purpose. here, we implicitly assume that the optimality of these choices for the eventual performance does not depend on the other potential variables such as the algorithm parameters θ. It also assumes that there *is* a single best answer for all basic algorithms, which is probably not the case. While future research can try to investigate more fine granular approaches, we consider these simplifications acceptable at this step of research and in the attempt of getting a first grip on the problem.

To address this question, we gathered data in the following way. We consider $1,058$ combinations of numbers of instances (between 100 and 10^6) and numbers of attributes (between 5 and 10^5) such that no point induces a total data matrix size larger than $3 \cdot 10^8$, which is a kind of maximum that can be treated with 16GB of memory. We then chose 10 *reference datasets*, each of which was once transformed to each of the points of the grid, yielding a total of $10,580$ datasets with properties originating from the original 10 properties. The exact grid definition and exhaustive details on how the reference datasets were chosen and how the grid point datasets were derived from them are given in the supplement. We then evaluated each atomic algorithm on each of these $10,580$ datasets under default configuration, using the full number of instances for training, and $1,000$ extra instances for predictions. Note that we are not interested in a kind of cross-validation but only the *runtime* for making predictions. Exploiting the fact that the prediction runtime *always* behaves linearly in the number of instances for which predictions are made, we also measure the time to make $1,000$ predictions for each atomic algorithm.

In this paper, we evaluate pipelines consisting of algorithms of the WEKA library [16]. The atomic algorithms of this library are as follows (details in supplement):

Pre-processors: CfsSubsetEval + BestFirst (se-bf), CfsSubsetEval + GreedyStepwise (se-gs), Correlation (cr), GainRatio (gr), InfoGain (ig), OneR (or), PrincipalComponents (pca), ReliefF (re), and SymmetricalUncert (su).

Base learners: BayesNet (bn), DecisionStump (ds), DecisionTable (dt), IBk (ibk), J48 (j48), JRip (jrip), KStar (k*), LMT (lmt), Logistic Regression (lr), MultilayerPerceptron (ann), NaiveBayes (nb), NaiveBayesMultinomial (nbm), OneR (1-r), PART (part), RandomForest (rf), RandomTree (rt), REPTree (rep), SimpleLogistic (sl), SMO (smo), VotedPerceptron (vp), and ZeroR (0-r).

Similar to [13], we base our runtime predictions mainly on the number of instances and attributes of a dataset. Fig. 2 illustrates the relationship between the three variables for the decision table classifier; each point is the median of the 10 samples on the grid point. It is evident that there is a very systematic relationship of runtime that above all depends on the number of instances and the number of attributes used for training the algorithm, and this or a similar pattern can be observed for all atomic algorithms. In contrast to [13], we include the learner parameters into the model.

However, there are some caveats with this simple model. First, many atomic algorithms exhibit runtime *noise*, i.e. have different runtimes when invoked several times under identical conditions, and this noise proportionally increases with the runtime. Second, for some of the algorithms, other data properties substantially influence the *slope* of the general runtime shape and hence the overall runtime. Both issues become evident in the right plot of Fig. 2, which summarizes in boxplots, one for each of the 170 considered datasets, 10 observations for the training time at the $50,000 \times 1,000$ grid point. The noise issue is evidenced by the large ranges of some boxplots, and the distribution of boxplots in the range of runtimes evidences the second concern. While the second issue seems to be a problem of insufficient features, a preliminary investigation revealed that additional meta-features could *not* resolve this problem. It is unclear whether such meta-features exist at all, and if so, whether they are cheap to compute and easily predictable (as needed later, cf. Section 5). An exhaustive search for the existence of features that satisfy these necessities is an ambitious project on its own and beyond the scope of this paper.

To assess the suitability of different regressors, we aggregated learning curves for three candidates. We used $10,000$ of the above $10,580$ data points as training instances, each of which is described by four attributes for learning, which are the number of instances (ni) and attributes (na) respectively, their product $ni \cdot na$, and $ni \cdot na^2$ to cope for the super-linear increase observed in some situations. Based on a series of subsets of this $10,000 \times 4$ meta dataset, we created learning curves for a standard linear regression, a neural network with one hidden layer of 100 units, and a random forest with 100 trees. The prediction performance is based on 58 grid points, which are not used for training but only validation, so that between 10 and $10,000$ points are used for training and 580 used for validation. This performance is computed once for each atomic algorithm and regressor, and the performance is then aggregated and displayed per regressor in Fig. 3; the dashed lines depict the 0.1-trimmed mean, and the solid lines the medians. The shaded areas reflect the range between the 0.25 and the 0.9 quantile. Note

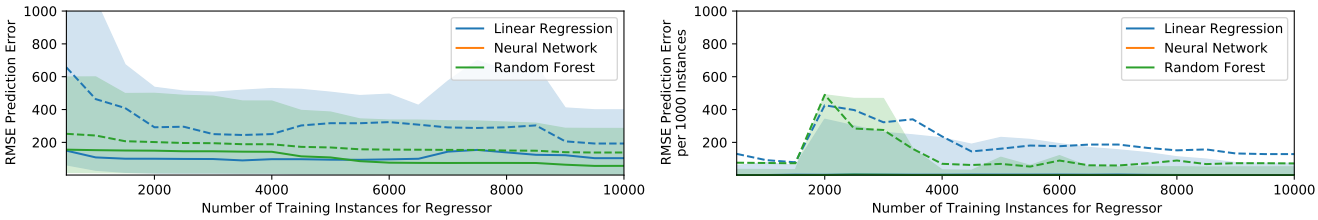


Fig. 3: Learning curves of the three considered regressors for train time (left) and prediction time (right).

that this entire validation is only based on data derived from the original 10 reference datasets.

The overall conclusion is that the number of instances and attributes are maybe not the only relevant but the most informative of the *examined* dataset meta-features, and that random forests seem best suited to predict runtimes on their basis. Surprisingly, the neural network regressor was not at all able to learn the runtime behavior. We suspect that this could be an issue of the used library, which was the NeuralNetworkRegressor of the Python package scikit-learn. Among the two others, one can see that linear regression already performs well. The random forest however seems better in terms of robustness. Besides, we learn here that in most cases we could also use only 1000 training examples and hence drastically reduce the effort to gather training data. As discussed above, the two key meta-features alone are not always sufficient to obtain accurate predictions. On the other hand, some of the problems are due to runtime noise, which cannot be tackled by any feature, and even the more systematic deviations cannot be easily assessed by common meta-features and require, if solvable at all, additional research.

4.2 Prediction Performance of Independent Models

While the last section gave an insight about general runtime behavior and an aggregated view on a kind of in-sample error for the three regressors, this section now answers the question of how well random forests can predict the runtime of each of the considered atomic algorithms using unseen datasets for validation. In contrast to the previous study, we now also consider the algorithm parameters as additional variables in the model.

4.2.1 Experimental Setup

How do we evaluate the capacity of the regressor to predict runtimes? The train and validation sets consist of runtime information made over variations of the 170 source datasets. A concrete observation is said to be a *d-instances* if the observed times stem from an application of the algorithm to the d-th (of the 170) dataset. Since the procedure to train and validate is identical for all atomic algorithms, we will for simplicity just refer to *the* (atomic) algorithm.

We propose to train the regressor with a dataset D_{train} consisting of a fixed number of observations for each grid point, exploiting the spectrum of parameter values of the algorithm and different data sources. In our case, we choose to produce 10 samples per grid point, i.e. $10,580$ samples in total, and for each sample, the algorithm parameters are drawn (uniformly) at random and the source dataset is selected in a round-robin fashion. That is, the source dataset (of 170) used to produce the i-th sample is $i \mod 170$.

The model performance is then assessed in a leave-one-out evaluation over the source datasets. For each dataset d of the 170 in the portfolio, the regressor is trained with the data described above, omitting from the $10,580$ points the d-instances. The regressor must then make predictions not on the d-instances of D_{train} but on 10 d-instances of large size ($50,000 \times 1,000$, see below) with random parameters. For each of the 10 instances, we measure the time to train the atomic algorithm and the time to make 1000 predictions for unseen instances. The regressor then tries to predict these 10 train and prediction (validation) times. This yields a total of $1,700$ predictions, which we can then interpret.

We use this kind of validation because we are interested in the prediction performance not in general but in rather *relevant* regions. A lot of runtimes in the grid are (very) low and hence irrelevant for the decision of pruning. Since we are particularly interested in difficult problems, in which the algorithms have rather high runtimes and are likely to hit the time-bound, each dataset is transformed into a version of size $50,000 \times 1,000$. For inputs of this size, all non-lazy algorithms have a training runtime of at least some seconds and often several minutes (or hours). Of course, the regressor should be able to predict runtimes for any input type, so the training does not specifically focus on this area of validation. In the supplement, we also add two additional validations for smaller datasets, in which it becomes apparent that the prediction accuracy for smaller sized input data is quite high, and where noise is negligible.

4.2.2 Atomic Algorithm Runtime Prediction Performance

Fig. 4 summarizes the results separately for each atomic algorithm. The plots summarize the differences between the true and the predicted runtime; negative values indicate runtime over-estimation. Even though the dataset size is always identical, the ground truth of the atomic learners can vary substantially. This is due to the effects discussed above but also the fact that we here have different algorithm parametrizations, which introduces substantial variance into the observations. Therefore, we group the ground truth observations into three bins (one in each row) and contextualize the prediction errors for this ground truth. This is because a prediction error of 5m is less severe for a total runtime of 40m than for a runtime of 10m. The colors indicate (our arbitrary) degrees of acceptability for the prediction error.

We can make several observations here. First, the runtime predictions for pre-processors are quite accurate in all cases except (se), which is somewhat anticipated; but

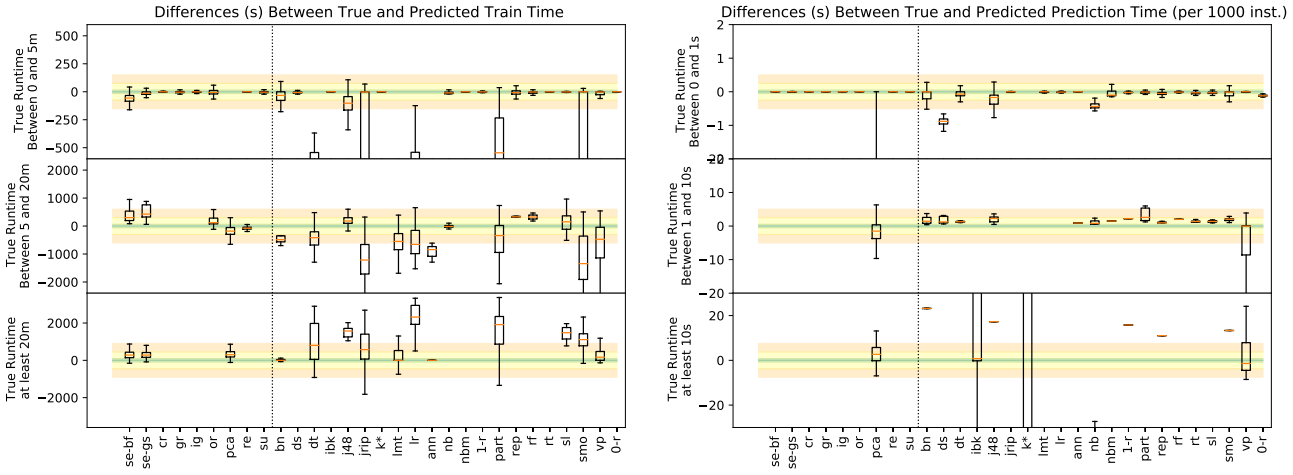


Fig. 4: Runtime prediction performances of a random forest with 100 trees on datasets of size 50000×1000. All 170 source datasets are considered, and a range of different parametrizations is considered for each base algorithm. Left and right column: Prediction results for training times and prediction times respectively. Rows group the ground truth runtimes into different bins in order to ease interpretation of the boxplots. Colors indicate degrees of acceptability for the prediction error.

the errors are still acceptable here. Second, the errors in predicting prediction times are almost negligible for most algorithms. Only for some lazy learners like nearest neighbors, the mistake can become quite large if the true prediction times are large; these are partially off the mark. Third, each atomic learner has a kind of *center* around which predictions seem to be oriented. Then, if the true runtime is substantially higher, the predictor tends to under-estimate the true runtime and it tends to over-estimate it if the true runtime is substantially lower. For example, the runtimes of the decision table are mostly between 5 and 20 minutes, and the error is small in those cases (middle row). For this learner, we can see in the upper row (lower true runtimes) a substantial over-estimation of runtimes and a substantial under-estimation in the third row.

The variance in the prediction accuracy between the learners is quite remarkable. In terms of prediction error on the training times, we can see that predictions are very precise even for high runtimes for bayesnet (bn), naivebayes (nb), or neural networks (ann). For other algorithms, the prediction accuracy is less pleasant, especially for the rule-based learners decisiontable (dt), jrip, and PART. Still, this is also partially due to a rather large input space, which we consider a stress-test here. The supplement contains identical plots for smaller input sizes, which show that the prediction errors in those cases substantially decrease.

4.3 Critical Over/Underestimation of Overall Runtimes

We now answer the question of how accurate the (regression-based) classifier ϕ (cf. Section 2) predicts timeouts for pipelines that only consist of an atomic algorithm. We check for each $t \in \{0.5, 1, 5, 10, 15, 20, 30, 60\}$ (minutes) and each atomic algorithm whether the rejection rule will correctly predict a timeout. For each timeout, we use a different reference size of datasets to check against (because the timeouts are typically adjusted to the dataset size as well). The dimensions are $\frac{t \cdot 10^6}{100} \times 100$ if $t < 10$ and $\frac{t \cdot 10^6}{1000} \times 1000$ else. Again, for each such size, the runtime of each algorithm is observed 10 times on the copy adjusted to this size for each of the 170 source datasets, and as in Section 4.2.1, the regressor used for prediction on dataset d is only trained with non-d-instances. We hence assess the correctness of $1,700$ decisions per algorithm and timeout.

The results are summarized in Fig. 5 in the form of true positives/negatives (green) and false positives/negatives (red) for each pair of atomic algorithms and considered timeouts. The top figure indicates the number of decisions for rejecting an execution in order to put the relative plots below into context. First of all, we can see that most preprocessors do never time out, and the predictor decides almost always correctly to allow their execution. A similar situation occurs with some classifiers, in particular decision stumps, Naive Bayes, or the random trees and forests. Among the cases in which the guard decides to allow execution, there are *very* few cases in which a timeout occurs. On the contrary, we can observe that the guard is sometimes quite restrictive, which results in a good deal of correct reject decisions, but also quite some wrong decisions. That is, the guard sometimes rejects executions that would *not* time out. This can be observed with particular frequency for logistic regression and PART.

The rather high false-positive rate for some of the learners indicates that too many solutions may be cut. This could indeed be the case when using the standard models M_θ. Hence, it is all the more important to look at whether we can improve these models using additional information gathered during the execution of an AutoML tool. The next section will show that we can substantially improve specifically on the algorithms for which false positives were high.

4.4 Prediction Performance of Enhanced Models

Finally, we want to answer the question of whether we can improve upon the above prediction performance in particular situations if we exploit knowledge about the candidate

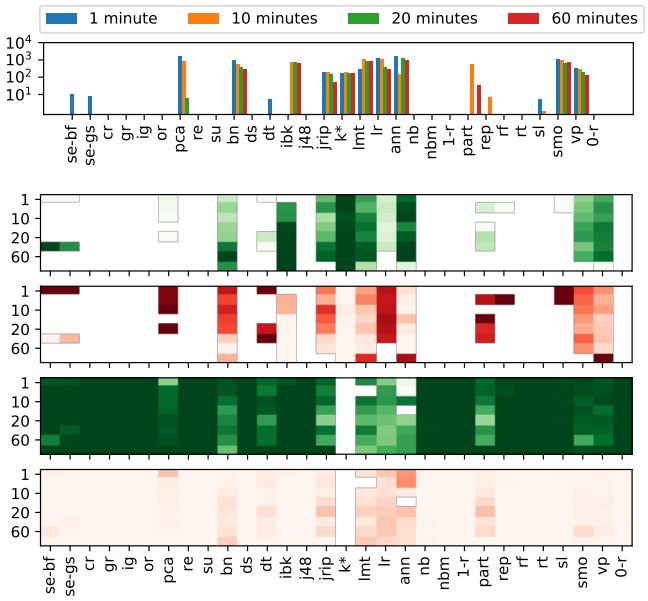


Fig. 5: Top figure: Number of rejected executions. Bottom figures: (1) + (2) Number of correctly/incorrectly rejected executions relative to the number of total rejects. (3) + (4) number of correctly/incorrectly permitted executions relative to the number of total accepts.

to be evaluated. Here, we exploit the knowledge that it is reasonable (and common practice) in AutoML to evaluate, for each pipeline candidate, first the variant in which all components have the parameters set to default. On one hand, this motivates the usage of a dedicated model only for the default parametrized version of each algorithm, because it reduces the number of variables and hence potentially simplifies the model. On the other hand, we can use the true runtime with default parametrization as a landmarking feature for runtime predictions of parametrized algorithms.

For comparability, the models are trained and validated like in Section 4.2. That is, for each atomic algorithm and each dataset d, we build, in addition to the model described in Section 4.2.1, which we label M_θ, a model $M_{-\theta}$, and $M_{\theta+d}$ *for* runtime prediction under default parametrization or *using* true runtimes under default parametrizations respectively; we call the latter the *posterior* model. More precisely, for each of these three, we have *two* models, one for predicting training times and one to predict prediction times. As in Section 4.2.1, each model is trained using those of the $10,580$ training instances, which are not d-instances, and then predict on 10 d-instances at point $50,000 \times 1,000$, which yields $1,700$ predictions per model we compare.

The results are summarized in Fig. 6. For each *configurable* atomic algorithm, we show 8 comparisons (rows in the plot), all based on the $1,700$ validation point observations. We use a hot-cool color map, which yields white for no change, dark blue for *improvements*, and dark red for *deteriorations* of at least 5m or 100% respectively. The two top plots show the improvement of a model for default parameters only (Def), and the lower two plots the improvement for the posterior model (Pos). For each of these two cases, we show the changes for the training time performance (first

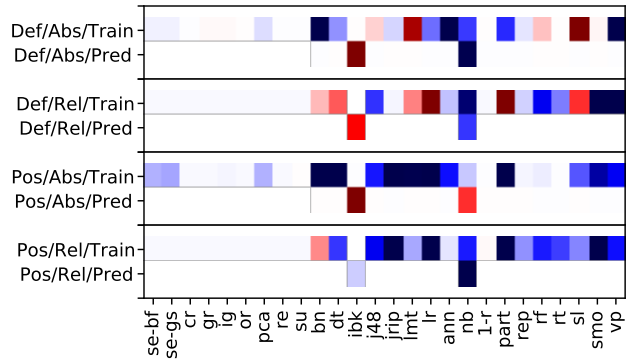


Fig. 6: Absolute (Abs) and relative (Rel) comparison of prediction performances between the standard model M_θ and the models $M_{-\theta}$ for default parametrization (Def) and models $M_{\theta+d}$ for posteriors (Pos). Blue indicates improvements, white is neutral, and red indicates deteriorations.

line) and prediction time performance (second line) once in terms of *absolute* (Abs) and once in terms of *relative* (Rel) improvements over the model of Section 4.2. Note that the algorithms *ds*, *kstar*, *nbm*, and *0r* are not configurable and hence do not appear in the comparison.

The figure conveys two messages. On one hand, the utility of default parametrization models is highly unclear, since there are a couple of cases in which the performance is even worse than for the parametrized model. It is hence rather not recommendable to use them except for specific learners. On the other hand, the posterior model *does* in fact add a substantial improvement to the prediction performance in almost all learners, especially for those with poor performance in the standard model. In many cases, the prediction error can be reduced by more than 5m on average. The supplement contains versions of Fig. 4 and 5 for the posterior model that also displays the substantial improvements for all algorithms; most observations then lie inside the colored ranges. Finally, we can observe that there is almost no impact on pre-processors, which is not surprising since the prediction performance is already quite well in the standard model.

5 Predicting Feature Transformations Realized by Pre-Processors

We now answer the question of how well we can predict the number of attributes the data have after a particular pre-processing step. As a quick recall, we need to predict the dataset meta-features of the output of pre-processors in order to feed the runtime model of the subsequent learner with estimates of meta-features of the data it will really work on. Since our runtime models rely only on the number of instances and the number of attributes, and since the number of instances is not touched by the (considered) pre-processors, we only need to estimate how the number of attributes will change.

Fortunately, we can predict the number of attributes quite well using only the number of attributes of the original data and the parameters of the pre-processor. This is not surprising since most pre-processors have an explicit

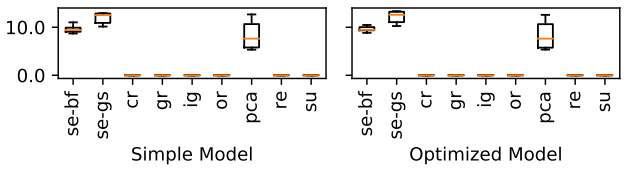


Fig. 7: Errors in the predicted number of attributes after pre-processor application with linear regression.

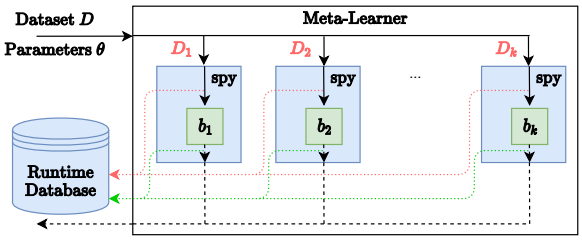


Fig. 8: Spy-wrapped base learners $b_1, .., b_k$ leak feature transformations of the meta-learner and base learner runtimes.

parameter to determine the maximum number of attributes that must be contained in the output. Taking the minimum of this value and the actual number of attributes of the dataset already achieves 100% accuracy in some cases.

The results of this simple strategy are summarized in Fig. 7 and compared against a more "optimized" feature set. To generate these results, all pre-processors were evaluated on all datasets in a leave-one-out fashion on two distinguished feature sets. The left plot shows the results when only using the derived feature above and the algorithm parameters θ. The right plot shows the feature set with the best performance found on all feature sets of length up to 3; these features were optimized in a preliminary step only on the 10 core datasets above. For each dataset, the information available for the other datasets is used to train a linear regressor, and the predictions obtained for the left-out dataset is used to compute an RMSE prediction error for the number of attributes afterward. The boxplots summarize the 170 points obtained over the datasets.

The evaluation shows that the simple feature set has almost no disadvantage compared to the optimized one. For the principal component analysis, the feature set does not yield a statistically significant difference. For the subset evaluation (with both best-first search and greedy step-wise search), the optimized feature set is slightly better, but also here, the advantage is almost negligible. To summarize, the best way to predict the changes of the data caused by a pre-processor is to use its parameters and the minimum of previous attributes and, if available, the attribute threshold configured in the pre-processor.

6 Predicting Meta-Learner Runtime

6.1 Runtime Prediction for Meta-Learners

Even if the implementation details of meta-learners are not known, we can make plausible assumptions about the general *behavior* of *every* (sensible) meta-learner. By definition, meta-learners rely on (several copies of) a base learner, which builds the actual model and makes predictions from it, while the meta-learner only organizes this resource. We can assume that a meta-learner, when being trained, will simply create several copies of its base learner, prepare their input data, and invoke a training (and maybe a prediction) procedure on them. The data fed to each base learner may deviate among each other and, more importantly, from the data given to the meta-learner itself. For example, some implementations of bagging [18] allow a flexible bag size (number of instances), and Random Subspace [19] reduces the number of features. Second, during the prediction phase, the meta-learner will consult and aggregate predictions from its base learners.

Under this premise, and given accurate estimates for the base learners and the parameters that tell us how many of them are created and how often they are called, there is no need to learn any model for the runtime of a meta-learner. Instead, one can directly compute its runtime based on its parameters and the base learner runtimes.

As sketched in the above example, meta-learners may also manipulate the data they receive as input, prior to passing it to their base learners. In order to obtain accurate predictions from the base learner models, we need to anticipate how these changes in the data look like, and call the base learner models with these anticipated values.

The claim we sustain in this section is that we can approximate the overall runtime of a meta-learner by

$$t_m(\theta, F) = k\left(tm^f(\theta_b, F') + (c^{p_f} + c^{p_p})tm^p(\theta_b, F')\right), \quad (2)$$

where k is the number of copies of the base learner maintained by the meta-learner, c^{p_f} and c^{p_p} are the number of invocations of the prediction routine of each base learner copy during the training and prediction phase of the meta-learner, respectively, θ_b and F' are the base learner parameters and meta-features of the data passed to each base learner copy, and tm^f and tm^p are the runtime models for training and prediction of the used base learner.

This model makes three important assumptions. First, it assumes that the meta-learner has negligible overhead, since the formula does not include any runtime contribution of the meta-learner itself. Second, it assumes that the parameters c^{p_f}, c^{p_p}, and F' are identical for all k copies of the base learner, whence the multiplication by k (the parameter θ_b is obviously identical for all by construction). Third, it assumes that the runtime does not depend on the *behavior* of the base learner (i.e., what it returns) but only on its runtime. We will discuss the legitimacy of these assumptions based on our empirical observations.

To predict an overall runtime for the meta-learner, we estimate the four parameters in (2) given F, θ, and θ_b as follows. For all considered meta-learners, the variable k is in fact an algorithm parameter and hence already part of θ. For the other three parameters c^{p_f}, c^{p_p}, and F', we create simple linear regression models that predict each of the respective variables from θ and F. To acquire ground truth target data to train and validate these models, we use an invasive approach illustrated in Fig. 8 by configuring the meta-learners with a spy base learner that records meta-features of the data received from the meta-learner for

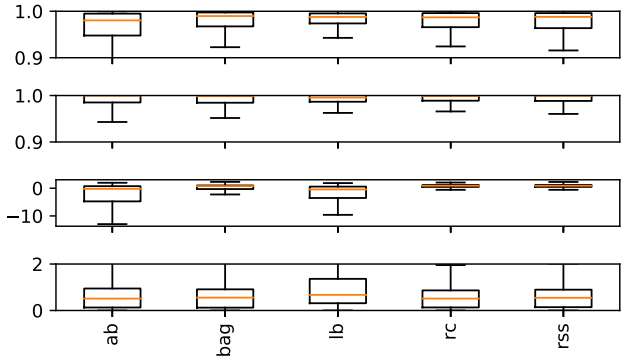


Fig. 9: Top 2 plots: Ratios of base learner train/prediction time and total meta-learner train/prediction time respectively. Bottom 2 plots: Abs/rel. difference of total true meta-learner runtime and the time predicted by (2) (in seconds).

training and prediction, and how often the different routines for training the base learner and making predictions were called. In order to get a ground truth for the overall runtimes of the meta-learners, the spy learners are not only dummies but also serve as wrappers for true base learners. Keeping track of the internal runtimes of base learners, we can also compute the net self-time of the meta-learner.

6.2 Evaluation

We consider the algorithms AdaBoost (ab), Bagging (bg), LogitBoost (lb), Random Subspace Classification (rss), and Random Committees (rc) from the WEKA library [16].

To verify that self-time is indeed negligible, consider the two top boxplots in Fig. 9. The plots show, as an aggregation over several hundred observations, the percentage of the base learner wall time compared to the total wall-time of the meta-learner for training and prediction, respectively. The top row compares the total wall time of the base learners during the meta-learner training phase to the total wall time of the meta-learner itself. The middle row computes the same ratios for the prediction phase of the meta-learner. We only show the ratios for cases where the overall runtime is at least 10s. In situations with runtimes under 10s, the overhead of the meta-learner is relatively substantial, but those quick runtime cases are practically irrelevant. We can see that the self-time of the meta-learner is under 5% in the great majority of cases, and even under 2% on average, which we consider a negligible fraction.

Concerning the equality of the parameters among the different copies of the base learner, our result is ambivalent. On the one hand, we could check that c^{p_f} and c^{p_p} are indeed constant across the different base learner instances for *all* meta-learners. On the other hand, this is not always true for the parameter F' but only for a couple of meta-learners. More precisely, AdaBoost (ab1) and LogitBoost (lb) may pass data of different dimensions to each copy of the base learner. However, those assignments are influenced by an unpredictable random component, so that no easy solution seems to be in sight.

The prediction quality obtained with (2) is summarized, for each meta-learner, in the bottom rows of Fig. 9. The boxplots show the absolute and relative difference between the true total wall time of a meta-learner and the value predicted by (2) in seconds using the *true average* base learner wall-time for tm^f and tm^p as to pretend a perfect estimate, which was determined using the spy wrapper discussed in Section 6. The other variables of the formula are derived from the meta-learner parameters.

The results clearly justify the usage of (2) to predict meta-learner runtimes. In particular, the good prediction results justify the assumption that the actual output of the algorithms does not affect the runtime of the meta-learner. For BAGGING, RANDOMCOMMITTEE, and RANDOMSUBSPACE, the prediction by the formula is almost perfect. For the other two algorithms, there are small but fairly acceptable deviations, which can be explained by the fact that the meta-features of the sub-datasets D_i are only averaged and hence do not provide perfect information.

7 Predicting Pipeline Runtime

7.1 Prediction Algorithm

The runtime of a pipeline (p, c_m, c_b) with pre-processor p, meta-learner c_m, and base learner c_b on folds D_f, D_e for fitting and evaluation, respectively, is predicted as described in Alg. 1. First, we compute the meta-features F from the training set D_f, which is just the shape of the data (cf. Section 4.1). We next check whether the pre-processor p is set (l. 2). If so, its runtime is estimated based on the corresponding runtime model tm_p using the dataset features F, and the features of the processed data are predicted, which yields an update of F. If this pre-processor has been executed before, we just obtain the information from the cache instead (l. 3–10). After this step, the meta-features of the data used by the learner (base or meta) are available (either as ground truth or as an estimate). Third, it is checked whether a meta-learner is used. If this is the case, the meta-features of the data shown to the base learner by the meta-learner is computed (l. 11). Fourth, the model corresponding to the used base learner is invoked with the meta-features determined before to obtain a prediction for the training and prediction runtime per instance. Alternatively, if we have executed the base learner with exactly the given configuration, we retrieve the information from the cache (l. 12–16). If a meta-learner is used, these values are plugged into (2) to obtain a runtime of the learner; otherwise, they are used as-is (l. 18 –22). Then, the runtimes of the pre-processor and the learner are added up and returned. The scaling factor of 1000 is of course arbitrary and just needs to coincide with the normalization used in the prediction time models.

This algorithm assumes that the following prediction models have already been trained offline. First, we have models for training (tm^f_a) and prediction time (tm^e_a) for every atomic algorithm a (cf. Section 4). Second, there is a meta-feature model fm_a for pre-processors and meta-learners a, which returns the meta-features of the processed data described with F and the algorithm parameters θ (cf. Sections 5 and 6). Finally, pm_{c_m} is a parameter model that allows one to infer the relevant runtime parameters of the meta-learner, given its parameters $\theta(c_m)$ (cf. Section 6).

Note the somewhat sloppy notation of $\theta(a)$ to specify the parameter description of an algorithm a, which in fact

Algorithm 1: GETPLRUNTIME$((p, c_m, c_b), D_f, D_e)$

$F \leftarrow shape(D_f)$;
if p *is defined* **then**
  if $(p, F) \in cacheT$ **then**
    $t_p \leftarrow cacheT(p, F, |D_e|)$;
    $F \leftarrow cacheF(p, F)$;
  else
    $t_p \leftarrow tm^f_p(F, \theta(p)) + \frac{|D_e|}{1000} tm^e_p(F, \theta(p))$;
    $F \leftarrow fm_p(F, \theta(p))$;
  end
end
if c_m *is defined* **then** $F \leftarrow fm_{c_m}(F, \theta(c_m))$;
if $(c_b, F, |D_e|) \in cacheT$ **then**
  $(t^f_{c_b}, t^e_{c_b}) \leftarrow cacheT(c_b, F, |D_e|)$;
else
  $t^f_{c_b} \leftarrow tm^f_{c_b}(F, \theta(c_b))$;
  $t^e_{c_b} \leftarrow tm^e_{c_b}(F, \theta(c_b))$;
end
if c_m *is defined* **then**
  $(k, p, q) \leftarrow pm_{c_m}(\theta(c_m))$;
  $t_l \leftarrow k\left(t^f_{c_b} + \frac{(p+q)}{1000} t^e_{c_b}\right)$;
else
  $t_l \leftarrow t^f_{c_b} + \frac{|D_e|}{1000} t^e_{c_b}$;
end
return $t_p + t_l$

is part of a itself. The models, e.g. tm_a, are of course not specific for θ but only for the algorithm associated with a.

There are two caches used in the approach. $cacheT$ is a cache for runtimes, and $cacheF$ is a cache for observed meta-features of a dataset after the application of a preprocessor. Both caches are only used in a reading fashion in this algorithm. They are updated with concrete observations whenever a pipeline is executed successfully.

7.2 Empirical Comparison to Naive Approach

To compare this approach against a non-decompositional runtime predictor, we gathered the runtime of 150k pipelines over the 170 datasets. The runtimes are between few milliseconds up to the time-bound of 1h, having roughly half of the observations with less than 1m runtime and one half above. As a baseline, we use a random forest with input space similar to the one used in Auto-WEKA [1] and auto-sklearn [2] in which every parameter of every algorithm becomes a feature in addition to some auxiliary features describing the occurring algorithms. The observation set is split into 170 folds of observations, one for each dataset, over which both predictors are validated. For each fold, the random forest is trained on the full set of observations not based on the respective dataset, and each component of the compositional model is trained exactly as in Section 4.2.1 using 10k instances of the runtime database not containing the dataset in question. For this comparison, we use the standard compositional model without posterior information of default configuration runtimes.

As a result of this comparison, we observe that the compositional model has a substantial advantage over the vector-based approach. More precisely, averaging over the 170 folds, we observe an average reduction in the RMSE of 768s, i.e. almost 13 minutes. In other words, the non-decompositional model is outperformed by a large margin and cannot be considered an adequate alternative.

8 Improvements in the AutoML Task

We now answer the question of what all these efforts are worth in the concrete context of an AutoML application. More precisely, we wonder how much more successful computations can be achieved by plugging Alg. 1 into function f in (1) of Section 2, and whether this also results in improved overall performance of the AutoML tool. To this end, we integrated Alg. 1 into the state-of-the-art AutoML tool ML-Plan and compared it with the vanilla version.

We compare evaluations of ML-Plan for the vanilla version against using the guard for 1h runs in a 5-fold hold-out cross-validation with 8 CPUs and 32GB memory. The timeout per candidate (cross-)evaluation here is 5 minutes, so a single execution must take not more than 1 minute. The results are based on 10 runs for each ML-Plan instance averaging over the error rates on stratified 70/30% splits.

In this evaluation, we are interested in the behavior of the AutoML tool on rather big datasets. On small datasets, the approach will rarely or never reject pipeline executions, so there is no point in applying it in such scenarios. In the following, we consider the datasets of our portfolio whose data matrix has at least 1 million entries.

We stress again that our primary goal is *not* to improve the performance of the AutoML tool but to improve *execution efficiency*. Since our approach does not improve any particular model, we cannot generally expect that the performance of the AutoML tool improves. It occasionally *may* improve results if avoided timeouts yield to the execution of models that otherwise would not have been executed. However, we consider this rather a desirable side effect. The main goal is to reduce wasted CPU time as much as possible, because large scale wasted CPU time is not only a severe ethical concern with the ambience (energy, CO2) but also leaves the AutoML user with the "what if?" question: Unless we exploit the resources as well as possible, we cannot know that there is no or little improvement through timeout avoidance. In other words, it is really desirable to avoid timeouts as much as possible, *regardless* the improvements in the overall AutoML process.

In this sense, Fig. 10 captures three metrics of success for the two versions of ML-Plan. Bars and lines are blue for ML-Plan without the guard and orange for ML-Plan with the guard. The first plot shows the *wasted* CPU time in hours (less is better). The numbers range between 0 and 8 since 8 CPUs are used for one hour. The middle and bottom plots capture the *number of* and the *time spent in* successful evaluations respectively. In both plot pairs, the first row shows the absolute metric value, and the second row shows the relative *increase* of the metric. Note that the absolute values for the *number* of executions are plotted on a log-scale, which hides a bit the high relative changes; these values are hence repeated in Table 1. In all plots, the horizontal *colored* dotted lines are the *mean* value of the respective approach. In addition, there are black dotted lines in the relative improvement plots for improvements of 100%, 200% and 500% respectively; these are just visual aids to ease interpretation of the bars inside the plots.

The plots on computational efficiency are complemented by Table 1, which shows average final error rates and numbers of successful evaluations per approach. Best achieve-

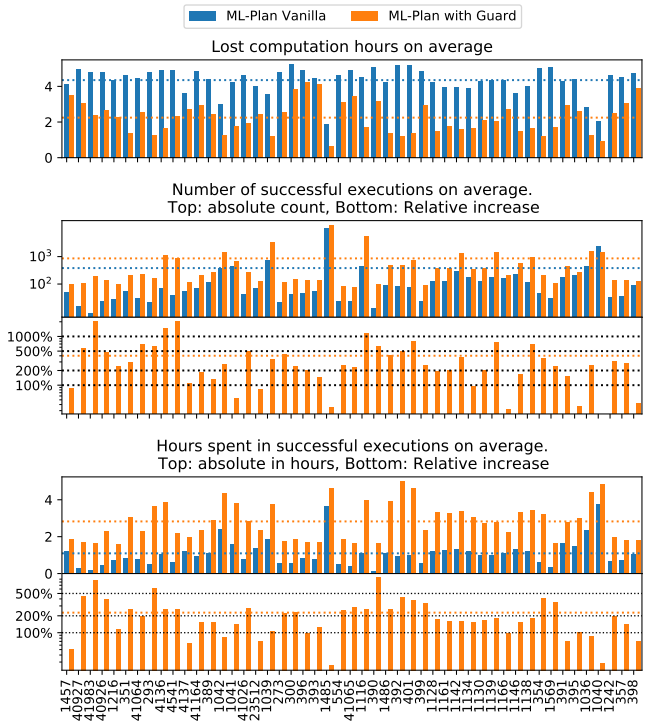


Fig. 10: Wasted computation times, numbers of successful evaluations, and hours spent in successful evaluations.

ments are bold and not significantly worse (according to a Wilcoxon signed-rank test with p-level 5%) are underlined.

The first observation is that the guard technique manages to *halve* the wasted CPU time. More precisely, the average reduction in wasted CPU time is 49% from a bit over 4h to almost "only" 2h. In fact, 2h wasted CPU time are even still a rather high number, and it would be desirable to achieve higher prediction accuracy to reduce this further. However, it is a substantial and important improvement. The stable results in Table 1 show that these savings do not usually come at the cost of worse results. That is, in very rare cases the eventually best pipeline is rejected by the guard.

Looking at efficiency from a more positive viewpoint and in terms of *successful* evaluations, we can see a substantial and partially dramatic increase in both the number of and the time spent in successful evaluations. On average, the number of successful executions is increased by over 400%, and the time spent in successful evaluations is increased by 227%. In some cases, improvements are even well above 1000%. For example, CIFAR-100 and DIABETES130US have improvements of approximately 2000%, i.e. enable 20 times more evaluations with the guard than without it.

These are huge improvements in the optimization process, and it is a bit unfortunate that these increases rarely yield better overall results. For most datasets, the observed score just remains unchanged, which is not very surprising since we know that for these datasets even in 24h with very few timeouts no better solution has been found. However, in those cases, the increase in successful evaluations is an additional argument that the best solution identifiable in the given timeout has been found.

Even if the decisions are not yet perfect, the alternative

Dataset (id on openml.org)	Vanilla		Guarded	
	error rate	evaluations	error rate	evaluations
AMAZON (1457)	**0.28± 0.03**	51.9	**0.28± 0.02**	**96.6**
CIFAR-100 (41983)	0.80± 0.00	8.89	**0.73± 0.08**	**187.5**
CIFAR-10 (40927)	0.64± 0.02	16.1	**0.58± 0.01**	**105.8**
CIFAR10SMALL (40926)	**0.58± 0.02**	24.4	0.59± 0.02	**141.9**
CLICKPREDSMALL (1216)	0.05± 0.01	27.6	**0.04± 0.00**	**95.9**
CODRNA (351)	**0.04± 0.00**	55.0	**0.04± 0.00**	**214.5**
CONVEX (41064)	**0.16± 0.01**	30.4	**0.16± 0.02**	**238.7**
COVERTYPE (293)	0.11± 0.01	22.7	**0.07± 0.01**	**165.4**
DEXTER (4136)	**0.07± 0.02**	73.2	**0.07± 0.02**	**1131.4**
DIABETES130US (4541)	**0.43± 0.01**	40.6	**0.43± 0.01**	**852.1**
DOROTHEA (4137)	0.08± 0.01	56.3	**0.07± 0.01**	**117.1**
FABERT (41164)	**0.32± 0.01**	71.5	**0.32± 0.01**	**204.0**
FBIS.WC (389)	0.20± 0.04	118.0	**0.19± 0.03**	**272.6**
GINAPRIOR2 (1041)	**0.07± 0.01**	458.6	**0.07± 0.01**	**701.7**
GINAPRIOR (1042)	**0.05± 0.01**	392.0	0.06± 0.01	**1449.6**
GISETTE (41026)	**0.04± 0.01**	43.6	**0.04± 0.01**	**262.7**
HIGGS (23512)	**0.29± 0.01**	72.3	0.30± 0.01	**131.2**
HIVAAGNOSTIC (1039)	**0.03± 0.00**	742.7	**0.03± 0.00**	**3238.6**
IMDB.DRAMA (273)	**0.36± 0.00**	21.89	**0.36± 0.00**	**114.8**
ISOLET (300)	**0.04± 0.01**	44.7	**0.04± 0.01**	**155.2**
LA1S.WC (396)	0.13± 0.02	45.6	**0.12± 0.01**	**136.8**
LA2S.WC (393)	**0.11± 0.01**	55.3	**0.11± 0.02**	**134.9**
MADELON (1485)	**0.26± 0.04**	10507.9	**0.26± 0.04**	**14161.6**
MNIST784 (554)	**0.03± 0.00**	23.7	0.05± 0.01	**82.7**
MNISTROTATION (41065)	0.58± 0.06	23.0	**0.53± 0.02**	**75.1**
MUSK (1116)	**0.00± 0.00**	431.4	**0.00± 0.00**	**5355.1**
NEW3S.WC (390)	**0.21± 0.00**	13.6	0.22± 0.00	**98.1**
NOMAO (1486)	**0.04± 0.01**	94.0	0.05± 0.01	**477.5**
OH0.WC (392)	**0.11± 0.02**	83.5	0.13± 0.02	**500.0**
OH10.WC (401)	**0.20± 0.03**	79.7	0.21± 0.02	**716.7**
OHSCAL.WC (399)	0.26± 0.01	24.6	**0.24± 0.02**	**87.8**
OVABREAST (1128)	**0.04± 0.01**	130.5	**0.04± 0.01**	**384.1**
OVACOLON (1161)	0.06± 0.04	122.7	**0.04± 0.01**	**373.8**
OVAENDOMETRIUM (1142)	**0.04± 0.00**	284.4	**0.04± 0.00**	**1322.2**
OVAKIDNEY (1134)	**0.02± 0.01**	182.11	**0.02± 0.01**	**350.4**
OVALUNG (1130)	**0.03± 0.03**	123.3	**0.03± 0.01**	**370.6**
OVAOMENTUM (1139)	**0.05± 0.00**	171.6	**0.05± 0.00**	**1467.3**
OVAOVARY (1166)	0.08± 0.02	162.7	**0.07± 0.01**	**214.1**
OVAPROSTATE (1146)	**0.01± 0.01**	221.2	**0.01± 0.00**	**591.8**
OVAUTERUS (1138)	0.08± 0.01	116.6	**0.07± 0.01**	**932.5**
POKER-HAND (1569)	0.38± 0.01	31.6	**0.37± 0.01**	**107.5**
POKER (354)	0.30± 0.03	46.0	**0.28± 0.03**	**207.4**
RE0.WC (391)	**0.19± 0.02**	175.7	**0.19± 0.02**	**446.4**
RE1.WC (395)	**0.16± 0.02**	203.7	0.18± 0.03	**278.7**
SYLVAAGNOSTIC (1036)	**0.01± 0.01**	441.0	**0.01± 0.00**	**1572.3**
SYLVAPRIOR (1040)	**0.01± 0.00**	**2356.7**	**0.01± 0.00**	1423.6
VEHICLENORM (1242)	**0.14± 0.00**	34.1	0.15± 0.01	**140.1**
VEHICLESENSIT (357)	**0.14± 0.01**	35.4	0.15± 0.01	**135.6**
WAP.WC (398)	0.19± 0.01	91.5	**0.18± 0.02**	**130.0**

TABLE 1: Comparison of overall error rates in ML-Plan.

with guard is certainly preferable over the one without guard. Only in three situations, the guarded version obtains a statistically significant worse result, and in these cases the different is at most 2 percentage points. On the other hand, we can see that CIFAR-10, CIFAR-100, COVERTYPE, and MNISTROTATION exhibit a substantially improved final error rate, and, to our knowledge, the score for CIFAR-10 is, by a large margin, the best that has been reported for any AutoML tool on the WEKA library with a timeout of 1h to date. The same holds for MNISTROTATION only that the margin to the best reported score on WEKA in 1h is not as large [5]. Overall, the guarded version performs mostly similar to the vanilla version with occasional small disadvantages or significant advantages.

To summarize, at least for ML-Plan, it seems preferable to use the guarded version from all considered points of view. The wasted CPU time can be halved, the number of successful evaluations increases substantially, the results hardly ever are substantially worse and sometimes substantially better. Of course, it would now be interesting to conduct this analysis for other AutoML tools as well. While there may be differences in the concrete outputs, we should expect for all common tools, such as auto-sklearn and TPOT, expect similar results.

9 Conclusion and Outlook

In this paper, we have proposed a regression-based approach to decide whether or not the execution of a machine learning pipeline in the context of AutoML will time out. In contrast to previous work on the runtime prediction of machine learning algorithms [12], [13], we admit parametrized algorithms, pre-processors, and meta-learners. We found that the general predictability of runtimes is satisfactory and that, especially on resource-intensive datasets, the approach can substantially increase both the number of and time spent in successful executions, without decreasing and sometimes substantially increasing its overall performance. Moreover, we believe that an execution guard is especially useful when performing AutoML on very resource-intensive problems such as predictive maintenance [20].

We envision four main directions for future work. The first one is to improve runtime predictions by more sophisticated learning methods, such as stacking or specialized regressors for different runtime regions. Second, building on the insights about runtime prediction for pipelines gained in this paper, and the solution concepts we developed for the case of two-step pipelines, a natural next step is to tackle runtime prediction for complex pipelines with fewer or without structural limitations as for example considered in [3], [21]. The ability to propagate dataset feature transformations, as proposed in this paper, will play a key role in this regard. The third direction is to extend runtime predictions by information about uncertainty, for example, by predicting confidence *intervals* instead of producing point estimates. Finally, more sophisticated decision rules are conceivable, for example based on the notion of expected utility maximization as shown in [22] for algorithm selection. Information about the expected runtime could be combined with information about the *expected quality* of a pipeline as in [10], [11], which we completely disregarded so far.

Acknowledgments

This work was partially 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 by the Paderborn Center for Parallel Computing (PC^2), which provided computational resources and computing time.

References

[1] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms," in *SIGKDD*, 2013.
[2] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *NeurIPS*, 2015.
[3] R. S. Olson and J. H. Moore, "Tpot: A tree-based pipeline optimization tool for automating machine learning," in *Workshop on Automated Machine Learning*, 2016.
[4] 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 *EuroGP*, 2017.
[5] F. Mohr, M. Wever, and E. Hüllermeier, "ML-Plan: Automated machine learning via hierarchical planning," *Mach. Learn.*, vol. 107, 2018.
[6] B. Chen, H. Wu, W. Mo, I. Chattopadhyay, and H. Lipson, "Autostacker: A compositional evolutionary learning system," in *GECCO*, 2018.
[7] I. Drori, Y. Krishnamurthy, R. Rampin, R. Lourenço, J. One, K. Cho, C. Silva, and J. Freire, "Alphad3m: Machine learning pipeline synthesis," in *AutoML@ICML Workshop*, 2018.
[8] H. Rakotoarison, M. Schoenauer, and M. Sebag, "Automated machine learning with monte-carlo tree search," in *IJCAI*, 2019.
[9] M. Reif, F. Shafait, and A. Dengel, "Prediction of classifier training time including parameter optimization," in *Annual Conference on Artificial Intelligence*, 2011.
[10] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *NeurIPS*, 2012.
[11] A. Klein, S. Falkner, S. Bartels, P. Hennig, F. Hutter *et al.*, "Fast bayesian hyperparameter optimization on large datasets," *Electronic Journal of Statistics*, vol. 11, 2017.
[12] T. Doan and J. Kalita, "Predicting run time of classification algorithms using meta-learning," *International Journal of Machine Learning and Cybernetics*, vol. 8, 2017.
[13] C. Yang, Y. Akimoto, D. W. Kim, and M. Udell, "Oboe: Collaborative filtering for automl model selection," in *SIGKDD*, 2019.
[14] V. Melnikov and E. Hüllermeier, "Learning to aggregate using uninorms," in *ECML/PKDD*, 2016.
[15] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown, "Algorithm runtime prediction: Methods & evaluation," *Art. Int.*, vol. 206, 2014.
[16] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD Explorations*, vol. 11, 2009.
[17] J. Vanschoren, J. N. van Rijn, B. Bischl, and L. Torgo, "OpenML: Networked science in machine learning," *SIGKDD Explorations*, vol. 15, 2013.
[18] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, 1996.
[19] T. K. Ho, "The random subspace method for constructing decision forests," *IEEE TPAMI*, vol. 20, 1998.
[20] T. Tornede, A. Tornede, M. Wever, F. Mohr, and E. Hüllermeier, "Automl for predictive maintenance: One tool to rul them all," in *IoTStream@ECMLPKDD 2020*, 2020.
[21] M. D. Wever, F. Mohr, and E. Hüllermeier, "Ml-plan for unlimited-length machine learning pipelines," in *AutoML@ICML Workshop*, 2018.
[22] A. Tornede, M. Wever, S. Werner, F. Mohr, and E. Hüllermeier, "Run2survive: A decision-theoretic approach to algorithm selection based on survival analysis," in *ACML*, 2020.

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.

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.

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.

Predicting Machine Learning Pipeline Runtimes in the Context of Automated Machine Learning - Supplement -

Felix Mohr, Marcel Wever, Alexander Tornede, Eyke Hüllermeier

1 Description of Considered Algorithms

1.1 General Overview

In this paper, we only consider pre-processing algorithms and classification algorithms of the WEKA library [1]. However, instead of using the original WEKA library, we use an *interruptible* modification [2] of WEKA since the WEKA algorithms cannot be stopped by default, an indispensable ability in the AutoML context. This modification is, in the used version, based on WEKA 3.9, which is therefore also the basis of all the algorithm descriptions above.

The parameter configurations considered for the different algorithms depend on the number and type of the parameters of each algorithm. For algorithms with only categorical attributes, we considered the whole parameter space. Numeric parameters were discretized, depending on the parameter semantic, on a linear or logarithmic scale. In situations where the cross product of all parameter values after discretization exceeded 1000, we considered only the 1-dimensional and 2-dimensional combinations of parameters (assuming independence of the others with respect to the runtime) to avoid the combinatorial explosion. Besides, we also consider each algorithm with its *default* parameter configuration.

In the following tables, we show every considered algorithm together with its parameters and the size of the considered parameter grid. Following the above explanation, the grid size is not always the product of the domain values; this is only the case if the resulting grid is not too big. Note that the grid size is *not* the number of experiments conducted for the algorithm but only a basis for it.

The upper part of Table 1 lists the pre-processors. Except for CfsSubsetEval, all pre-processors use the Ranker algorithm as a filter, which associates each attribute with a score using the evaluator and then simply sorts them by this score and eventually returns the best N, so the parameter N here defines the number of attributes kept. If N exceeds the number of attributes in the dataset, no attributes are discarded. The synthetic feature discussed in Sec. 7 of the paper is precisely $min\{N, n\}$, where N is the above parameter for the Ranker and n is the number of attributes of the dataset. For the PCA, this is only an upper bound, because it can be even more selective beforehand. The GreedyStepwise algorithm has a similar parameter also named N, but it is not guaranteed that the number of attributes afterwards is of this number. The BestFirst algorithm has no comparable parameter, so the number of attributes afterwards cannot be determined so easily.

The middle part of Table 1 shows the configurations of the base learners. For some parameters, we adopt the notation <name>: <expr of x>: <cond on x> where <name> is the name of the parameters, <expr of x> is some mathematical expression over a variable and <cond on x> explains the *integer* domain of which we consider values for x. There are four base learners that do not have parameters, which are DecisionStump, KStar, NaiveBayesMultinomial, and ZeroR; they are listed for completeness.

As written in the main paper, for each point on the main evaluation grid defined over numbers of instances and attributes, we run 10 experiments, which call for 10 different parametrizations. Of course, it is not feasible to test all configurations on each grid point, so we limit ourselves to an admittedly arbitrary number of 10 evaluations per point. These configurations are drawn in a kind of latin hypercube sampling, to possibly have "maximally distant" configurations on each grid point. The hope is that there is then a rather homogenous distribution of configurations over the whole evaluation grid.

Note that SMO plays a somewhat special role in this classifier portfolio in that it is the only classifier that has no native support for multi class classification. On multi class classification problems, SMO is run in an all-pairs mode, i.e. the algorithm is trained on each each pair of classes once and eventually the class that "wins" most pair-wise battles is chosen. It is natural to expect that predictions for SMO will not be too accurate in our model since we do not consider the number of labels in the set of features.

The last section of Table 1 describes the five used meta-learners. All of them have a parameter I to control the number of instances of the base learner, which corresponds to the parameter k in the paper. By default $I = 10$ for all meta-learners.

The name meta-learner is a bit arbitrary and, in fact, one could also call what we consider meta-learners in this paper as *homogeneous* ensembles. They are ensembles in that they maintain a set of base learners, which are trained and used to gather *prediction opinions*, which are then aggregated by the meta-learner into one final prediction. They are homoge-

Algorithm	Parameters and the considered subsets of their domains (0/1 for binary)	Grid Size
CfsSubsetEval + Best (se+bf)	M: 0/1, L: 0/1, Z: 0/1, D: (0, 1, 2), N: (1, 2, 4, 10, 100, 1000), S: (0, 1, 2, 3)	576
CfsSubsetEval + Greedy (se+gs)	M: 0/1, L: 0/1, Z: 0/1, C: 0/1, B: 0/1, N: (1, 2, 10, 100)	128
CorrelationAttributeEval (cr)	N: (1, 2, 4, 10, 100, 1000)	6
GainRatioAttributeEval (gr)	N: (1, 2, 4, 10, 100, 1000)	6
InfoGainAttributeEval (ig)	N: (1, 2, 4, 10, 100, 1000), M: 0/1, B: 0/1	24
OneRAttributeEval (or)	N: (1, 2, 10, 100), D: 0/1, F: (2, 4, 8, 10), B: (1, 2, 4, 6, 8, 16)	192
PrincipalComponents (pca)	N: (1, 2, 10, 100), A: (-1, 1, 2, 4, 8, 10, 100), C: 0/1, R: (.5, .7., .9, .95, .99), O: 0/1	560
ReliefFAttributeEval (re)	N: (1, 2, 4, 10, 100, 1000), K: (1, 2, 4, 10, 100), A: (1, 2, 3, 10), M: (1, 2, 10, 100, 1000)	198
SymmetricalUncert (su)	N: (1, 2, 4, 10, 100, 1000), M: 0/1	12
BayesNet (bn)	D: 0/1, Q: (K2, HillClimber, LAGDHillClimber, SimulatedAnnealing, TabuSearch, TAN)	12
DecisionStump (ds)	-	-
DecisionTable (dt)	I: 0/1, E: (acc, rmse, mae, auc), S: (BestFirst, GreedyStepwise), X: $1 \leqslant X \leqslant 10$	170
IBk (ibk)	K: (2, 4, 8, 16, 32, 64), X: 0/1, E: 0/1, I: 0/1, F: 0/1	16
J48 (j48)	O: 0/1, U: 0/1, B: 0/1, J: 0/1, S: 0/1, A: 0/1, C: $.1x : 1 \leqslant x \leqslant 10$, M: (1, 4, 8, 16, 32, 64)	22
JRip (jrip)	E: 0/1, P: 0/1, F: $1 \leqslant F \leqslant 5$, N: $1 \leqslant N \leqslant 5$, O:(1, 2, 4, 8, 16, 32, 64)	19
KStar (k*)	-	-
LMT (lmt)	B: 0/1, R: 0/1, C: 0/1, P: 0/1, A: 0/1, M: (1, 2, 4, 8, 16, 32, 64), W: (0, 0.5, 1, 1.5, 2, 4)	18
Logistic Regression (lr)	R: $10^x : -9 \leqslant x \leqslant 2$	13
MultilayerPerceptron (ann)	B: 0/1, R: 0/1, C: 0/1, D: 0/1, L&M: $0.1 \cdot x : 1 \leqslant x \leqslant 10$, H: (i, o, t)	25
NaiveBayes (nb)	K: 0/1, D: 0/1	4
NaiveBayesMultinomial (nbm)	-	-
OneR (1-r)	B: (1, 2, 4, 8, 16, 32, 64)	7
PART (part)	R: 0/1, B: 0/1, U: 0/1, J: 0/1, M: (1, 4, 8, 16, 32, 64), N: (1, 2, 4, 5, 6, 7, 8, 9, 10)	19
RandomForest (rf)	Options of RandomTree + I: $2^x : 0 \leqslant x \leqslant 7$	48
RandomTree (rt)	B: 0/1, K: $1 \leqslant K \leqslant 10$, M: $2^x : 1 \leqslant x \leqslant 7$, V: $10^x : -6 \leqslant x \leqslant 2$, D&N: $2^x : 0 \leqslant x \leqslant 6$	40
REPTree (rep)	P: 0/1, M: $2^x : 1 \leqslant x \leqslant 7$, V: $10^x : -6 \leqslant x \leqslant 2$, L&N: $2^x : 0 \leqslant x \leqslant 6$	31
SimpleLogistic (sl)	S: 0/1, A: 0/1, P: 0/1, W: (0, 0.5, 1, 1.5, 2), H&I&M: $2^x : 0 \leqslant x \leqslant 10$	41
SMO (smo)	C: $10^x : -6 \leqslant x \leqslant 5$, N: (1,2), L: $10^x : -6 \leqslant x \leqslant 2$, P: $10^x : -14 \leqslant x \leqslant -3$, V: $1 \leqslant V \leqslant 10$	42
VotedPerceptron (vp)	I: $2^x : 1 \leqslant x \leqslant 10$, E: (2, 3, 4, 5), M: (1, 10, 100,1000, 100000, 1000000)	20
ZeroR (0-r)	-	-
AdaBoost (ab)	Q: 0/1, P: (50, 60, 70, 80, 90, 95), I: (5, 20, 50)	36
Bagging (bg)	O: 0/1, P: (50, 60, 70, 80, 90, 95), I: (5, 20, 50)	36
LogitBoost (lb)	as AdaBoost + L: (0, 0.01, 0.1), H: (0.1, 0.5, 0.9), Z = (1, 2, 3, 5, 10)	2520
Random Subspace Classification (rss)	I: (5, 10, 20, 50)	4
Random Committees (rc)	P: (50, 60, 70, 80, 90, 95), I: (5, 10, 20, 50)	28

TABLE 1: Overview of all considered algorithms with their parameters

nous, because they require that all its base learners are of the same type. Of course, this restriction is not necessary, and we can, on a conceptual level, simply combine different classifier types in a voting ensemble or use a stacking approach, which also allows different base learner types to be combined.

In this paper, we disregard the latter type of ensembles for technical reasons. This is mainly due to the fact that the configurability of such ensembles in the current *implementation* of ML-Plan is still rather limited, and we would not have had the ability to make a detailed analysis of the runtime prediction of "very" heterogeneous ensembles. Hence, we decided to better omit this ensemble type at this point and leave it for future work.

1.2 Some Details on the Algorithms

1.2.1 Pre-Processors

CfsSubsetEval. This is an implementation of the Correlation based Feature Set (CFS) evaluation developed in Mark Hall's PhD thesis [3]. Roughly speaking, it evaluates a feature set by relating the average correlation between the features and the class attributes to the average correlation between the features themselves; higher scores are better. It is noteworthy that the term "correlation" here is not meant to capture linear correlation but rather generally a degree of dependency among variables. Attributes are first discretized internally and then, one out of different possible measures for inter-dependency can be applied. CfsSubsetEval can be used either inside a Best First Search (se-bf), which theoretically enables an exhaustive search stopped at some stalling point of time, or in a local search called GreedyStepwise (se-gs), in which only single (irrevocable) modifications to the feature set are allowed.

Correlation (cr). This ranks the attributes based on their (linear) correlation with the class attribute.

InfoGain (ig). This is an implementation of the traditional information gain criterion proposed by Quinlan [4]. It measures by how much the (Shannon) entropy reduces by conditioning (i.e. branching) on a particular attribute.

GainRatio (gr). The Gain Ratio criterion was proposed by Quinlan [4] in the context of decision tree inference. It evaluates an attribute by dividing its information gain by its entropy. It can hence be understood as a relative information gain that puts the conditional entropy into the context of

the unconditional one and hence, to a degree, measures the "surprisingness" of the conditional entropy observed. This can be thought of as not overweighing certain attributes that cause high information gain when effectively *all* attributes yield high information gain, just because of a high basic entropy.

OneR (or). This is an implementation of Holte's 1R algorithm [5], which builds on classifier for each attribute and tries to predict the class using only this attribute. This can be thought of as a one-level decision tree. Attributes with lowest error rates are retained.

PrincipalComponents (pca). The (linear) principal component analysis introduced by Pearson [6]. The coordinate system is rotated so that the axises of the new system correspond to the (orthogonal) eigenvectors of the covariance matrix of the data. The new attributes are hence non-pure mixtures of the original attributes, and they are ranked by the variance the data have in their direction, corresponding to the eigenvalue of the respective eigenvector.

ReliefF (re). This algorithm implements the RELIEFF algorithm introduced by Kononeko in [7]. It is a generalization of the original RELIEF algorithm introduced by Kira and Rendell in [8], which tries to estimate the importance of a feature by weighing it according to its relevance at the decision boundary. In a stochastic manner, it draws points and for each point its nearest neighbor with the same and with the other class. The weight of each attribute is initialized with 0 and updated with the squared mean distance (0/1 for nominal attributes) among each of the samples between the nearest miss and nearest hit instance, respectively. Hence, higher average distances to other classes increase the feature importance and, in a sense, point to better separability in this dimension.

SymmetricalUncert (su). This is a normalized version of information gain in which also the bias towards features with a high number of occurring values is reduced.

1.2.2 Base Learners

BayesNet (bn). This is an implementation that builds a Bayes net to derive posterior probabilities for classes based on a set of given attributes. In contrast to (nb), it does not make the assumption of independence among attributes but rather tries to assess the dependency structure in a network, which then allows for probabilistic inference.

DecisionStump (ds). Creates a *binary* (ternary if there are missing values) split over a single attribute and a value for it. One child fold contains the instances in which the attribute has a value "less or" equal to the split value, while the other folds contains all other instances with non-missing values for the attribute. This procedure is in contrast to (1r) in that a specific value is considered for splitting, instead of creating one fold for each possible value.

DecisionTable (dt). Decision tables were proposed by Kohavi in [9]. A decision table combines a pre-processor with a conditional majority class decision rule. The classifier is built by selecting the "most relevant" features using some preprocessing algorithm; continuous attributes are discretized. The algorithm memorizes all instances projected to the retained features, and to make a prediction for an unlabeled instance, it will search for all seen instances with *exact* matches in the retained attributes and return the majority class among them. If no match is found, the overall majority class is returned.

IBk (ibk). This is an implementation of a k-nearest neighbor classifier. When making a prediction, each class label is associated with a probability that depends on how many among the k nearest neighbors have that label and what their distance to the predicted point is. For a deterministic prediction, the class with the highest probability is predicted.

J48 (j48). This is an implementation of C4.5 decision trees as proposed by Quinlan [4], an extension of ID3 that facilitates pruning in order to avoid overfitting. The name is a bit misleading and probably refers to the fact that it is implemented in Java.

JRip (jrip). This is an implementation of the RIPPERk algorithm proposed by Cohen [10]. This is a rule generation algorithm that runs an open loop in which each iteration consists of building a new rule. This is done in two steps: First a rule is created by growing a rule premise with the same building blocks used in decision trees maximizing information gain, and in a second step the rule is pruned to remove possibly unnecessary or overfitting conditions. Examples satisfying the rule's premise are then removed from the considered dataset. The loop runs until the dataset is exhausted or a rule with an unacceptably high error rate is produced.

KStar (k*). Implementation of the K* algorithm, which is a nearest neighbors algorithm adopting the so-called K* distance introduced in [11]. The main idea is to base the distance between two points not in terms of their geometrical distance but on their information theoretic distance. This distance is defined in terms of *expected* numbers of modifications that need to be made on the first instance to obtain the second one. That is, instead of looking only at the shortest such modification, the algorithm sums over "all possible" transformations and weighs them with a probability.

LMT (lmt). Implementation of Logistic Model Trees [12]. This algorithm combines the idea of decision trees and logistic regression in the sense that it grows a tree that maintains in its leafs logistic regression models rather than constant classes. The algorithm adopts the LogitBoost algorithm [13] to iteratively fit logistic regression models already at each *inner* node of the tree. The model of the parent is extended by additional terms for the logistic model that are fit only based on the instances "still in play" at the respective child. The splitting criterion is the same as used for C4.5 trees, and the algorithm stops if less than 15 examples are available for a leaf.

Logistic Regression (lr). An implementation of the lr algorithm proposed in [14]. This approach combines the ideas of Ridge regression and "classical" lr. To cope with several classes, the algorithm is slightly modified as to derive a probability vector for an unseen instance and predicts the instance with the highest probability.

MultilayerPerceptron (ann). A standard artificial neural

network trained via back-propagation. In our evaluation, the networks have exactly one hidden layer (with number of hidden units depending on the parametrization).

NaiveBayes (nb). A classifier that computes posterior probabilities for each class based on an independency assumption between the features [15].

NaiveBayesMultinomial (nbm). [16]

OneR (1-r). A fast and very simple rule-based classifier, using a single attribute in the rule's head, with which the target attribute can most accurately be concluded [17]. Numeric attributes are discretized by splitting the ranges into a fixed number of intervals to mitigate overfitting.

PART (part). A rule-based classifier, building a decision list iteratively by fitting a partial C4.5 decision tree and transforming the path to the 'best' leaf into a rule [18].

RandomTree (rt). A tree classifier that considers a randomly sampled subset of k features at each node for splitting [19]. In contrast to other decision tree classifiers, it does no pruning. The main purpose of this classifier is to serve as a base learner in a RandomForest ensemble.

RandomForest (rf). An classifier, forming an ensemble of RandomTree classifiers via bagging [19].

REPTree (rep). A fast implementation of a decision tree that splits nodes by trading off information gain and variance and pruning the tree with backfitting afterwards.

SimpleLogistic (sl). A simple classifier consisting of linear logistic regression models that are fitted using LogitBoost [12], [20]. The optimal number of LogitBoost iterations is determined via cross-validation.

SMO (smo). A support vector machine that is induced via the *s*equential *m*inimal *o*ptimization algorithm [21], hence the name, for solving the underlying quadratic programming problem.

VotedPerceptron (vp). An implementation of the voted perceptron algorithm which builds a set of simple perceptrons and combines their predictions into a weighted sum [22]. The weight of each perceptron is determined by counting the number of instances that are correctly predicted during the training.

ZeroR (0-r). A classifier that agnostic of an instance simply predicts the mode of the training data sample, i.e., the majority class.

2 Datasets

We consider 170 datasets from the openml.org platform [23] as the *basis* of the datasets used in the evaluation. The considered set is a strict superset of those used in [24], which is a systematic selection of datasets with heterogeneous properties. Fig. 2 shows these datasets in terms of their numbers of instances and attributes respectively as orange points; the left is linear scaled with only an excerpt for readability, and the right one shows them on a log-scale.

2.1 Selection of 10 Representative Datasets

The idea is to adopt repetitions of a randomly initialized kMeans algorithms in the space of meta-features over the datasets. The algorithm is run with $k = 10$, i.e. identifies 10 centroids, and in each round, the point that is closest to each of the centroids gets a point. After 100 repetitions of this "game", the ten points this the highest scores are selected as representants.

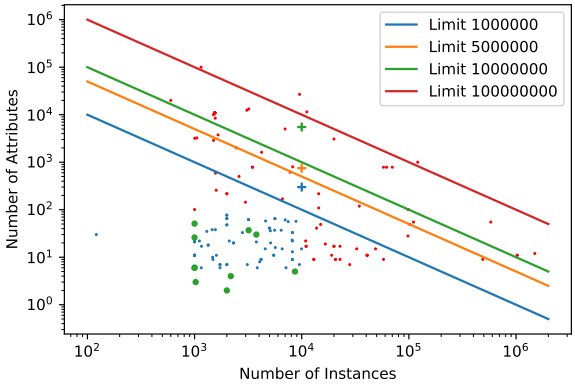


Fig. 1: Dataset Overview

From the pool of candidates, we exclude those with more than 10000 instances or more than 100 attributes. This only has the reason to be sure that none of the datasets that is part of the final evaluation will be selected here. Also, we are specifically interested in representants for the *other* attributes, since our procedure to produce new datasets will make the dataset size an arbitrary variable anyway.

Interestingly, this process is absolutely stable in that the 10 highest scored datasets are selected in almost all rounds as the best representants. These datasets are (together with their openml id): KR-VS-KP (3), HYPOTHYROID (57), RMFTSA-SLEEPDATA (741), FRI-C1-1000-5 (743), QUAKE (772), FRI-C3-1000-5 (813), FRI-C2-1000-25 (903), FRI-C0-1000-50 (904), BALLOON (914), and VISUALIZING-SOIL (923).

An overview of all the datasets in the number of instances and attributes is shown in Fig. 1. The red points are the datasets excluded from the clustering, the green ones are then 10 chosen datasets, and the orange ones the others. The colored lines are visual aids to recognize the resulting size of the input matrix.

2.2 Definition of the Grid and the Dataset Generator

Even taking into account several dozens of different datasets, it is necessary to derive further data points in the space of used datasets. On one hand, if we run a classifier on each of the datasets, we will obtain only 170 semantically distinct runtime records (even if several repetitions are made using different seeds). This is a rather small number of training examples for a regression algorithm, and it would be even harder to conduct a meaningful validation. On the other hand, as one also sees in the figures, those points only cover a couple of regions of the spectrum of the input space of the regressor, and we lack examples in many regions.

To overcome this limitation, we adopted a technique to cast a version of an arbitrarily shaped dataset to any point on the plain spanned by the numbers of instances and numbers of attributes respectively. Formally, this corresponds

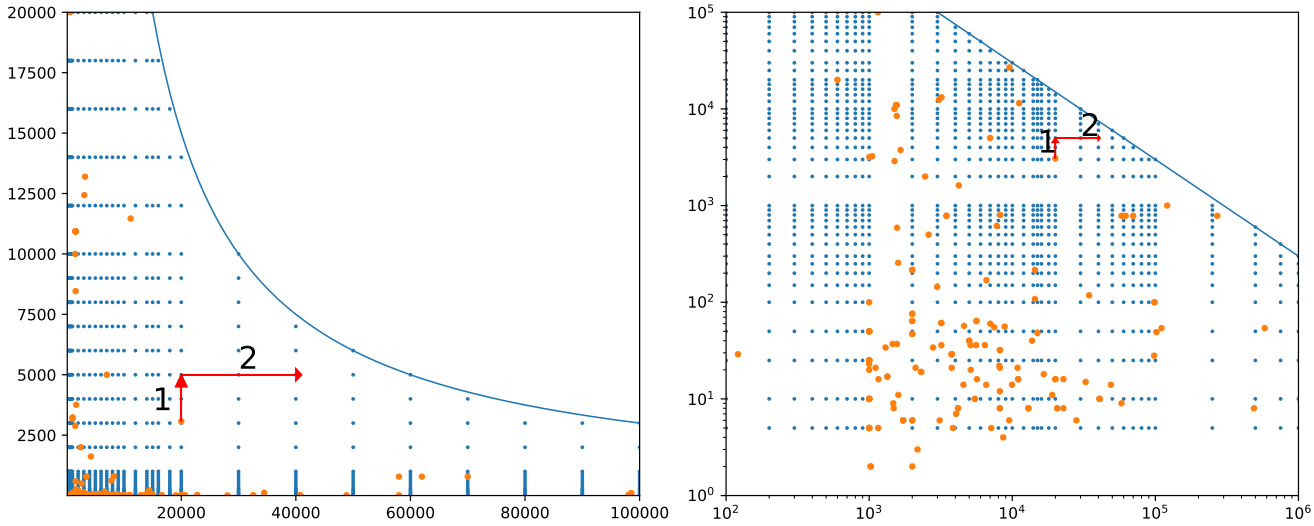


Fig. 2: Training Point Grid points (in blue) and sizes of the 170 original datasets (in orange). Left shows an excerpt of the linear scale plot and right shows log scales.

to a generator function $g : \mathbb{N} \times \mathbb{N} \times \mathcal{D} \to \mathcal{D}$ such that $g(n, d, D)$ is a $n \times d$ dataset derived from the original dataset D. The idea is to somewhat uniformly cover all regions of the input space regardless the sizes of the original data. To this end, we consider a sub-space of the grid over the following points whenever the product (input matrix size) has at most $3 \cdot 10^8$ entries, which fits into the memory for most datasets in a way that most learners can still be applied if 16GB memory are available in total:

- Number of instances: 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, 12000, 14000, 15000, 16000, 18000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000, 100000, 250000, 500000, 750000, 1000000
- Number of attributes: 5, 10, 25, 50, 100, 150, 200, 250, 300, 400, 500, 600, 700, 800, 900, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, 12000, 14000, 16000, 18000, 20000, 25000, 30000, 40000, 50000, 60000, 70000, 80000, 90000, 100000

There are 1058 points in total. The resulting grid together with the limit curve for a input size of $3 \cdot 10^8$ is shown in blue in Fig. 2. Given an original dataset D of the form $n_0 \times d_0$, we apply, in this order, four steps to compute $g(n, d, D)$:

1) if $d > d_0$, generate $d - d_0$ random numeric columns,
2) if $n > n_0$, generate $n - n_0$ new instances using SMOTE [25] (applying it once for each class and treating the respective class as if it was the minority class),
3) if $d_0 > d$, randomly eliminate $d_0 - d$ attributes of D,
4) if $n_0 > n$, randomly eliminate $n_0 - n$ instances of D.

An example transformation for the CIFAR10_small dataset (id 40926) with 20000 instances and 3072 attributes to point (40000, 5000) is shown with the red arrows.

Augmenting attributes by random columns is admittedly a bit arbitrary and could also be replaced by other, rather kernel-like, feature expansions. We chose this for simplicity but did not find any indications that this choice had any particular effects on the runtime. For example, we could observe for all algorithms cases in which a dataset with only original attributes had an identical runtime to one with a high number of generated features (but then the same overall number of features).

Using SMOTE for instance generation aims at preserving the structure of the original data, including class distribution, as well as possible.

2.3 Considered Meta-Features

Besides the number of instances and attributes, we considered also attributes in our analysis, and the overall set of considered candidates for dataset-metafeatures $\mathcal{F}$ was as follows: the number of instances (ni); number of attributes in total (na), numeric (nn), symbolic/categorical (ns), after binarization (nab); number of labels (nl), number of possible values for the nominal attributes (nc), total variance (tv), and the attributes required to cover 50%, 90%, 95%, and 90% of the variance (vx) respectively. Table 2 lists all the considered datasets with their respective meta-features. The meta-features are computed for each dataset as a whole but can of course be different for concrete splits considered during evaluation. Note that some dataset names appear twice, but the datasets then have modifications in the number of attributes or classes.

id	name	ni	na	nn	ns	nab	nl	nc	tv	v50	v90	v95	v99
3	kr-vs-kp	3.20E+03	3.70E+01	0.00E+00	3.60E+01	7.50E+01	2.00E+00	6.00E+00	9.49E+00	21	46	53	62
6	letter	2.00E+04	1.70E+01	1.60E+01	0.00E+00	1.70E+01	2.60E+01	0.00E+00	8.55E+01	6	14	15	16
12	mfeat-factors	2.00E+03	2.17E+02	2.16E+02	0.00E+00	2.17E+02	1.00E+01	0.00E+00	9.89E+05	34	85	95	107
14	mfeat-fourier	2.00E+03	7.70E+01	7.60E+01	0.00E+00	7.70E+01	1.00E+01	0.00E+00	4.20E-01	13	52	63	74
16	mfeat-karhunen	2.00E+03	6.50E+01	6.40E+01	0.00E+00	6.50E+01	1.00E+01	0.00E+00	4.15E+02	8	36	46	60
18	mfeat-morpho...	2.00E+03	7.00E+00	6.00E+00	0.00E+00	7.00E+00	1.00E+01	0.00E+00	1.41E+07	1	1	1	1
21	car	1.73E+03	7.00E+00	0.00E+00	6.00E+00	2.20E+01	4.00E+00	2.10E+01	4.25E+00	10	19	20	21
22	mfeat-zernike	2.00E+03	4.80E+01	4.70E+01	0.00E+00	4.80E+01	1.00E+01	0.00E+00	1.37E+05	6	19	22	27
23	cmc	1.47E+03	1.00E+01	2.00E+00	7.00E+00	2.50E+01	3.00E+00	1.60E+01	7.66E+01	1	2	2	15
24	mushroom	8.12E+03	2.30E+01	0.00E+00	2.20E+01	1.26E+02	2.00E+00	1.17E+02	1.12E+01	25	61	72	91
26	nursery	1.30E+04	9.00E+00	0.00E+00	8.00E+00	2.80E+01	5.00E+00	2.50E+01	5.47E+00	13	24	26	27
28	optdigits	5.62E+03	6.50E+01	6.40E+01	0.00E+00	6.50E+01	1.00E+01	0.00E+00	1.20E+03	17	36	41	45
30	page-blocks	5.47E+03	1.10E+01	1.00E+01	0.00E+00	1.10E+01	5.00E+00	0.00E+00	2.87E+07	1	2	3	3
31	credit-g	1.00E+03	2.10E+01	7.00E+00	1.30E+01	6.40E+01	2.00E+00	5.20E+01	7.96E+06	1	1	1	1
32	pendigits	1.10E+04	1.70E+01	1.60E+01	0.00E+00	1.70E+01	1.00E+01	0.00E+00	1.49E+04	6	13	14	16
36	segment	2.31E+03	2.00E+01	1.90E+01	0.00E+00	2.00E+01	7.00E+00	0.00E+00	2.25E+04	3	8	9	11
38	sick	3.77E+03	3.00E+01	7.00E+00	2.20E+01	5.30E+01	2.00E+00	5.00E+00	3.37E+03	2	4	4	4
44	spambase	4.60E+03	5.80E+01	5.70E+01	0.00E+00	5.70E+01	2.00E+00	0.00E+00	4.07E+05	1	1	2	2
46	splice	3.19E+03	6.20E+01	0.00E+00	6.10E+01	3.46E+03	3.00E+00	3.46E+03	4.54E+01	114	217	231	2016
57	hypothyroid	3.77E+03	3.00E+01	7.00E+00	2.20E+01	5.30E+01	4.00E+00	5.00E+00	3.37E+03	2	4	4	4
60	waveform-5000	5.00E+03	4.10E+01	4.00E+01	0.00E+00	4.00E+01	3.00E+00	0.00E+00	6.94E+01	11	33	37	40
179	adult	4.88E+04	1.50E+01	2.00E+00	1.20E+01	1.21E+02	2.00E+00	1.17E+02	1.12E+10	1	1	1	1
180	covertype	1.10E+05	5.50E+01	1.40E+01	4.00E+01	9.40E+01	7.00E+00	0.00E+00	4.32E+06	1	2	2	4
181	yeast	1.48E+03	9.00E+00	8.00E+00	0.00E+00	8.00E+00	1.00E+01	0.00E+00	8.00E-02	3	6	7	8
182	satimage	6.43E+03	3.70E+01	3.60E+01	0.00E+00	3.60E+01	6.00E+00	0.00E+00	3.60E+01	18	33	35	36
183	abalone	4.18E+03	9.00E+00	7.00E+00	1.00E+00	1.00E+01	2.80E+01	3.00E+00	1.01E+00	3	5	6	9
184	kropt	2.81E+04	7.00E+00	0.00E+00	6.00E+00	4.80E+01	1.80E+01	4.80E+01	4.83E+00	16	34	37	40
185	baseball	1.34E+03	1.80E+01	1.50E+01	2.00E+00	1.36E+03	3.00E+00	1.35E+03	5.56E+06	1	3	5	7
273	IMDB.drama	1.21E+05	1.00E+03	0.00E+00	0.00E+00	1.00E+03	2.00E+00	0.00E+00	1.86E+01	227	771	875	970
293	covertype	5.81E+05	5.50E+01	0.00E+00	0.00E+00	5.40E+01	2.00E+00	0.00E+00	9.95E+00	5	9	10	10
300	isolet	7.80E+03	6.18E+02	6.17E+02	0.00E+00	6.17E+02	2.60E+01	0.00E+00	1.15E+02	185	466	520	578
351	codrna	4.89E+05	9.00E+00	0.00E+00	0.00E+00	8.00E+00	2.00E+00	0.00E+00	7.99E+00	4	8	8	8
354	poker	1.03E+06	1.10E+01	0.00E+00	0.00E+00	1.00E+01	2.00E+00	0.00E+00	1.00E+01	5	9	10	10
357	vehicle_sensIT	9.85E+04	1.01E+02	0.00E+00	0.00E+00	1.00E+02	2.00E+00	0.00E+00	1.00E+02	51	91	96	100
389	fbis.wc	2.46E+03	2.00E+03	0.00E+00	0.00E+00	2.00E+03	1.70E+01	0.00E+00	5.47E+04	55	619	912	1467
390	new3s.wc	9.56E+03	2.68E+04	0.00E+00	0.00E+00	2.68E+04	4.40E+01	0.00E+00	1.37E+06	17	877	1980	6367
391	re0.wc	1.50E+03	2.89E+03	0.00E+00	0.00E+00	2.89E+03	1.30E+01	0.00E+00	2.16E+03	114	734	1063	1565
392	oh0.wc	1.00E+03	3.18E+03	0.00E+00	0.00E+00	3.18E+03	1.00E+01	0.00E+00	4.26E+03	254	1104	1437	2097
393	la2s.wc	3.08E+03	1.24E+04	0.00E+00	0.00E+00	1.24E+04	6.00E+00	0.00E+00	3.25E+04	342	2477	3729	6541
395	re1.wc	1.66E+03	3.76E+03	0.00E+00	0.00E+00	3.76E+03	2.50E+01	0.00E+00	3.73E+03	189	1101	1505	2230
396	la1s.wc	3.20E+03	1.32E+04	0.00E+00	0.00E+00	1.32E+04	6.00E+00	0.00E+00	3.97E+04	247	2362	3632	6681
398	wap.wc	1.56E+03	8.46E+03	0.00E+00	0.00E+00	8.46E+03	2.00E+01	0.00E+00	8.58E+03	308	2082	2951	4531
399	ohscal.wc	1.12E+04	1.15E+04	0.00E+00	0.00E+00	1.15E+04	1.00E+01	0.00E+00	2.29E+04	1023	4493	5781	8061
401	oh10.wc	1.05E+03	3.24E+03	0.00E+00	0.00E+00	3.24E+03	1.00E+01	0.00E+00	5.18E+03	269	1171	1527	2193
554	mnist_784	7.00E+04	7.85E+02	7.84E+02	0.00E+00	7.84E+02	1.00E+01	0.00E+00	3.43E+06	141	298	345	435
679	rmftsa_sleep...	1.02E+03	3.00E+00	2.00E+00	0.00E+00	2.00E+00	4.00E+00	0.00E+00	2.29E+02	1	1	1	1
715	fri_c3_1000_25	1.00E+03	2.60E+01	2.50E+01	0.00E+00	2.50E+01	2.00E+00	0.00E+00	2.50E+01	13	23	24	25
718	fri_c4_1000_100	1.00E+03	1.01E+02	1.00E+02	0.00E+00	1.00E+02	2.00E+00	0.00E+00	9.99E+01	50	90	95	99
720	abalone	4.18E+03	9.00E+00	7.00E+00	1.00E+00	1.00E+01	2.00E+00	3.00E+00	1.01E+00	3	5	6	9
722	pol	1.50E+04	4.90E+01	4.80E+01	0.00E+00	4.80E+01	2.00E+00	0.00E+00	5.31E+03	5	13	16	22
723	fri_c4_1000_25	1.00E+03	2.60E+01	2.50E+01	0.00E+00	2.50E+01	2.00E+00	0.00E+00	2.50E+01	13	23	24	25
727	2dplanes	4.08E+04	1.10E+01	1.00E+01	0.00E+00	1.00E+01	2.00E+00	0.00E+00	7.00E+00	5	9	10	10
728	analcatdata_...	4.05E+03	8.00E+00	7.00E+00	0.00E+00	7.00E+00	2.00E+00	0.00E+00	9.82E+01	1	1	1	2
734	ailerons	1.38E+04	4.10E+01	4.00E+01	0.00E+00	4.00E+01	2.00E+00	0.00E+00	6.85E+04	1	1	1	2
735	cpu_small	8.19E+03	1.30E+01	1.20E+01	0.00E+00	1.20E+01	2.00E+00	0.00E+00	2.55E+11	1	2	3	3
737	space_ga	3.11E+03	7.00E+00	6.00E+00	0.00E+00	6.00E+00	2.00E+00	0.00E+00	1.32E+14	1	1	1	1
740	fri_c3_1000_10	1.00E+03	1.10E+01	1.00E+01	0.00E+00	1.00E+01	2.00E+00	0.00E+00	9.99E+00	5	9	10	10
741	rmftsa_sleep...	1.02E+03	3.00E+00	1.00E+00	1.00E+00	5.00E+00	2.00E+00	4.00E+00	2.30E+02	1	1	1	1
743	fri_c1_1000_5	1.00E+03	6.00E+00	5.00E+00	0.00E+00	5.00E+00	2.00E+00	0.00E+00	5.00E+00	3	5	5	5
751	fri_c4_1000_10	1.00E+03	1.10E+01	1.00E+01	0.00E+00	1.00E+01	2.00E+00	0.00E+00	9.99E+00	5	9	10	10
752	puma32H	8.19E+03	3.30E+01	3.20E+01	0.00E+00	3.20E+01	2.00E+00	0.00E+00	9.42E+03	3	5	5	5
761	cpu_act	8.19E+03	2.20E+01	2.10E+01	0.00E+00	2.10E+01	2.00E+00	0.00E+00	2.55E+11	1	2	3	3
772	quake	2.18E+03	4.00E+00	3.00E+00	0.00E+00	3.00E+00	2.00E+00	0.00E+00	2.86E+04	2	2	2	3
797	fri_c4_1000_50	1.00E+03	5.10E+01	5.00E+01	0.00E+00	5.00E+01	2.00E+00	0.00E+00	5.00E+01	25	45	48	50
799	fri_c0_1000_5	1.00E+03	6.00E+00	5.00E+00	0.00E+00	5.00E+00	2.00E+00	0.00E+00	5.00E+00	3	5	5	5
803	delta_ailerons	7.13E+03	6.00E+00	5.00E+00	0.00E+00	5.00E+00	2.00E+00	0.00E+00	0.00E+00	1	3	3	4
806	fri_c3_1000_50	1.00E+03	5.10E+01	5.00E+01	0.00E+00	5.00E+01	2.00E+00	0.00E+00	5.00E+01	25	45	48	50
807	kin8nm	8.19E+03	9.00E+00	8.00E+00	0.00E+00	8.00E+00	2.00E+00	0.00E+00	6.56E+00	4	8	8	8
813	fri_c3_1000_5	1.00E+03	6.00E+00	5.00E+00	0.00E+00	5.00E+00	2.00E+00	0.00E+00	5.00E+00	3	5	5	5
816	puma8NH	8.19E+03	9.00E+00	8.00E+00	0.00E+00	8.00E+00	2.00E+00	0.00E+00	7.33E+00	4	6	6	8
819	delta_elevators	9.52E+03	7.00E+00	6.00E+00	0.00E+00	6.00E+00	2.00E+00	0.00E+00	6.74E+02	1	1	1	2
821	house_16H	2.28E+04	1.70E+01	1.60E+01	0.00E+00	1.60E+01	2.00E+00	0.00E+00	4.34E+09	1	1	1	1
822	cal_housing	2.06E+04	9.00E+00	8.00E+00	0.00E+00	8.00E+00	2.00E+00	0.00E+00	6.37E+06	1	2	3	4
823	houses	2.06E+04	9.00E+00	8.00E+00	0.00E+00	8.00E+00	2.00E+00	0.00E+00	1.33E+10	1	1	1	1
833	bank32nh	8.19E+03	3.30E+01	3.20E+01	0.00E+00	3.20E+01	2.00E+00	0.00E+00	5.01E+01	3	13	21	27
837	fri_c1_1000_50	1.00E+03	5.10E+01	5.00E+01	0.00E+00	5.00E+01	2.00E+00	0.00E+00	5.00E+01	25	45	48	50
843	house_8L	2.28E+04	9.00E+00	8.00E+00	0.00E+00	8.00E+00	2.00E+00	0.00E+00	6.22E+08	1	1	1	1
845	fri_c0_1000_10	1.00E+03	1.10E+01	1.00E+01	0.00E+00	1.00E+01	2.00E+00	0.00E+00	9.99E+00	5	9	10	10
846	elevators	1.66E+04	1.90E+01	1.80E+01	0.00E+00	1.80E+01	2.00E+00	0.00E+00	7.78E+04	1	1	1	1
847	wind	6.57E+03	1.50E+01	1.40E+01	0.00E+00	1.40E+01	2.00E+00	0.00E+00	3.70E+02	5	12	13	14
849	fri_c0_1000_25	1.00E+03	2.60E+01	2.50E+01	0.00E+00	2.50E+01	2.00E+00	0.00E+00	2.50E+01	13	23	24	25
866	fri_c2_1000_50	1.00E+03	5.10E+01	5.00E+01	0.00E+00	5.00E+01	2.00E+00	0.00E+00	5.00E+01	25	45	48	50
871	pollen	3.85E+03	6.00E+00	5.00E+00	0.00E+00	5.00E+00	2.00E+00	0.00E+00	2.41E+02	2	4	4	5
881	mv	4.08E+04	1.10E+01	7.00E+00	3.00E+00	1.40E+01	2.00E+00	3.00E+00	1.68E+04	1	2	2	2
897	colleges_aaup	1.16E+03	1.70E+01	1.30E+01	3.00E+00	1.21E+03	2.00E+00	1.20E+03	5.93E+06	1	1	1	5
901	fried	4.08E+04	1.10E+01	1.00E+01	0.00E+00	1.00E+01	2.00E+00	0.00E+00	8.30E-01	5	9	10	10

id	name	ni	na	nn	ns	nab	nl	nc	tv	v50	v90	v95	v99
903	fri_c2_1000_25	1.00E+03	2.60E+01	2.50E+01	0.00E+00	2.50E+01	2.00E+00	0.00E+00	2.50E+01	13	23	24	25
904	fri_c0_1000_50	1.00E+03	5.10E+01	5.00E+01	0.00E+00	5.00E+01	2.00E+00	0.00E+00	5.00E+01	25	45	48	50
910	fri_c1_1000_10	1.00E+03	1.10E+01	1.00E+01	0.00E+00	1.00E+01	2.00E+00	0.00E+00	9.99E+00	5	9	10	10
912	fri_c2_1000_5	1.00E+03	6.00E+00	5.00E+00	0.00E+00	5.00E+00	2.00E+00	0.00E+00	5.00E+00	3	5	5	5
913	fri_c2_1000_10	1.00E+03	1.10E+01	1.00E+01	0.00E+00	1.00E+01	2.00E+00	0.00E+00	9.99E+00	5	9	10	10
914	balloon	2.00E+03	2.00E+00	1.00E+00	0.00E+00	1.00E+00	2.00E+00	0.00E+00	3.00E-02	1	1	1	1
917	fri_c1_1000_25	1.00E+03	2.60E+01	2.50E+01	0.00E+00	2.50E+01	2.00E+00	0.00E+00	2.50E+01	13	23	24	25
923	visualizing_...	8.64E+03	5.00E+00	3.00E+00	1.00E+00	5.00E+00	2.00E+00	0.00E+00	8.32E+02	1	1	1	1
930	colleges_usnews	1.30E+03	3.50E+01	3.20E+01	2.00E+00	1.36E+03	2.00E+00	1.32E+03	1.29E+08	3	6	8	11
934	socmob	1.16E+03	6.00E+00	1.00E+00	4.00E+00	3.90E+01	2.00E+00	3.40E+01	1.97E+03	1	1	1	1
953	splice	3.19E+03	6.20E+01	0.00E+00	6.10E+01	3.46E+03	2.00E+00	3.46E+03	4.54E+01	114	217	231	2016
958	segment	2.31E+03	2.00E+01	1.90E+01	0.00E+00	1.90E+01	2.00E+00	0.00E+00	2.25E+04	3	8	9	11
959	nursery	1.30E+04	9.00E+00	0.00E+00	8.00E+00	2.70E+01	2.00E+00	2.50E+01	5.47E+00	13	24	26	27
962	mfeat-morpho...	2.00E+03	7.00E+00	6.00E+00	0.00E+00	6.00E+00	2.00E+00	0.00E+00	1.41E+07	1	1	1	1
966	analcatdata_...	1.34E+03	1.80E+01	1.50E+01	2.00E+00	1.36E+03	2.00E+00	1.35E+03	5.56E+06	1	3	5	7
971	mfeat-fourier	2.00E+03	7.70E+01	7.60E+01	0.00E+00	7.60E+01	2.00E+00	0.00E+00	4.20E-01	13	52	63	74
976	JapaneseVowels	9.96E+03	1.50E+01	1.40E+01	0.00E+00	1.40E+01	2.00E+00	0.00E+00	2.80E+02	1	1	2	2
977	letter	2.00E+04	1.70E+01	1.60E+01	0.00E+00	1.60E+01	2.00E+00	0.00E+00	8.55E+01	6	14	15	16
978	mfeat-factors	2.00E+03	2.17E+02	2.16E+02	0.00E+00	2.16E+02	2.00E+00	0.00E+00	9.89E+05	34	85	95	107
979	waveform-5000	5.00E+03	4.10E+01	4.00E+01	0.00E+00	4.00E+01	2.00E+00	0.00E+00	6.94E+01	11	33	37	40
980	optdigits	5.62E+03	6.50E+01	6.40E+01	0.00E+00	6.40E+01	2.00E+00	0.00E+00	1.20E+03	17	36	41	45
991	car	1.73E+03	7.00E+00	0.00E+00	6.00E+00	2.10E+01	2.00E+00	2.10E+01	4.25E+00	10	19	20	21
993	kdd_ipums_la...	7.02E+03	6.10E+01	3.30E+01	2.70E+01	6.77E+02	2.00E+00	6.27E+02	1.01E+12	3	5	5	6
995	mfeat-zernike	2.00E+03	4.80E+01	4.70E+01	0.00E+00	4.70E+01	2.00E+00	0.00E+00	1.37E+05	6	19	22	27
1000	hypothyroid	3.77E+03	3.00E+01	7.00E+00	2.20E+01	5.30E+01	2.00E+00	5.00E+00	3.37E+03	2	4	4	4
1002	ipums_la_98-...	7.48E+03	5.60E+01	1.60E+01	3.90E+01	2.46E+02	2.00E+00	2.13E+02	9.42E+11	3	5	5	6
1018	ipums_la_99-...	8.84E+03	5.70E+01	1.50E+01	4.10E+01	2.70E+02	2.00E+00	2.36E+02	5.34E+11	2	3	3	5
1019	pendigits	1.10E+04	1.70E+01	1.60E+01	0.00E+00	1.60E+01	2.00E+00	0.00E+00	1.49E+04	6	13	14	16
1020	mfeat-karhunen	2.00E+03	6.50E+01	6.40E+01	0.00E+00	6.40E+01	2.00E+00	0.00E+00	4.15E+02	8	36	46	60
1021	page-blocks	5.47E+03	1.10E+01	1.00E+01	0.00E+00	1.00E+01	2.00E+00	0.00E+00	2.87E+07	1	2	3	3
1036	sylva_agnostic	1.44E+04	2.17E+02	2.16E+02	0.00E+00	2.16E+02	2.00E+00	0.00E+00	1.12E+06	8	27	32	38
1037	ada_prior	4.56E+03	1.50E+01	6.00E+00	8.00E+00	1.02E+02	2.00E+00	9.40E+01	1.17E+10	1	1	1	1
1039	hiva_agnostic	4.23E+03	1.62E+03	1.62E+03	0.00E+00	1.62E+03	2.00E+00	0.00E+00	1.06E+02	287	1010	1221	1471
1040	sylva_prior	1.44E+04	1.09E+02	1.08E+02	0.00E+00	1.08E+02	2.00E+00	0.00E+00	5.57E+05	4	14	16	20
1041	gina_prior2	3.47E+03	7.85E+02	7.84E+02	0.00E+00	7.84E+02	1.00E+01	0.00E+00	3.44E+06	142	299	344	431
1042	gina_prior	3.47E+03	7.85E+02	7.84E+02	0.00E+00	7.84E+02	2.00E+00	0.00E+00	3.44E+06	142	299	344	431
1049	pc4	1.46E+03	3.80E+01	3.70E+01	0.00E+00	3.70E+01	2.00E+00	0.00E+00	3.93E+09	1	1	1	1
1050	pc3	1.56E+03	3.80E+01	3.70E+01	0.00E+00	3.70E+01	2.00E+00	0.00E+00	1.29E+11	1	1	1	1
1053	jm1	1.09E+04	2.20E+01	2.10E+01	0.00E+00	2.10E+01	2.00E+00	0.00E+00	1.89E+11	1	1	1	1
1059	ar1	1.21E+02	3.00E+01	2.90E+01	0.00E+00	2.90E+01	2.00E+00	0.00E+00	3.50E+07	1	1	1	1
1067	kc1	2.11E+03	2.20E+01	2.10E+01	0.00E+00	2.10E+01	2.00E+00	0.00E+00	3.05E+08	1	1	1	1
1068	pc1	1.11E+03	2.20E+01	2.10E+01	0.00E+00	2.10E+01	2.00E+00	0.00E+00	2.92E+10	1	1	1	1
1069	pc2	5.59E+03	3.70E+01	3.60E+01	0.00E+00	3.60E+01	2.00E+00	0.00E+00	7.82E+08	1	1	1	1
1116	musk	6.60E+03	1.69E+02	1.66E+02	2.00E+00	6.87E+03	2.00E+00	6.70E+03	1.30E+06	54	129	144	159
1119	adult-census	3.26E+04	1.50E+01	6.00E+00	8.00E+00	1.05E+02	2.00E+00	9.70E+01	1.12E+10	1	1	1	1
1120	MagicTelescope	1.90E+04	1.10E+01	1.00E+01	0.00E+00	1.00E+01	2.00E+00	0.00E+00	1.49E+04	2	4	6	7
1128	OVA_Breast	1.54E+03	1.09E+04	1.09E+04	0.00E+00	1.09E+04	2.00E+00	0.00E+00	2.41E+11	39	847	1834	5753
1130	OVA_Lung	1.54E+03	1.09E+04	1.09E+04	0.00E+00	1.09E+04	2.00E+00	0.00E+00	2.36E+11	42	867	1869	5806
1134	OVA_Kidney	1.54E+03	1.09E+04	1.09E+04	0.00E+00	1.09E+04	2.00E+00	0.00E+00	2.41E+11	39	847	1834	5753
1138	OVA_Uterus	1.54E+03	1.09E+04	1.09E+04	0.00E+00	1.09E+04	2.00E+00	0.00E+00	2.41E+11	39	847	1834	5753
1139	OVA_Omentum	1.54E+03	1.09E+04	1.09E+04	0.00E+00	1.09E+04	2.00E+00	0.00E+00	2.41E+11	39	847	1834	5753
1142	OVA_Endometrium	1.54E+03	1.09E+04	1.09E+04	0.00E+00	1.09E+04	2.00E+00	0.00E+00	2.41E+11	39	847	1834	5753
1146	OVA_Prostate	1.54E+03	1.09E+04	1.09E+04	0.00E+00	1.09E+04	2.00E+00	0.00E+00	2.41E+11	39	847	1834	5753
1161	OVA_Colon	1.54E+03	1.09E+04	1.09E+04	0.00E+00	1.09E+04	2.00E+00	0.00E+00	2.41E+11	39	847	1834	5753
1166	OVA_Ovary	1.54E+03	1.09E+04	1.09E+04	0.00E+00	1.09E+04	2.00E+00	0.00E+00	2.41E+11	39	847	1834	5753
1216	Click_predic...	1.50E+06	1.20E+01	1.10E+01	0.00E+00	1.10E+01	2.00E+00	0.00E+00	2.46E+37	1	1	1	1
1242	vehicleNorm	9.85E+04	1.01E+02	0.00E+00	0.00E+00	1.00E+02	2.00E+00	0.00E+00	1.00E+02	51	91	96	100
1457	amazon-comme...	1.50E+03	1.00E+04	1.00E+04	0.00E+00	1.00E+04	5.00E+01	0.00E+00	1.35E+04	8	1165	2213	4000
1485	madelon	2.60E+03	5.01E+02	5.00E+02	0.00E+00	5.00E+02	2.00E+00	0.00E+00	4.55E+05	76	254	306	389
1486	nomao	3.45E+04	1.19E+02	8.90E+01	2.90E+01	1.74E+02	2.00E+00	8.10E+01	1.38E+01	31	98	117	141
1501	semeion	1.59E+03	2.57E+02	2.56E+02	0.00E+00	2.56E+02	1.00E+01	0.00E+00	5.35E+01	113	219	235	250
1569	poker-hand	1.02E+06	1.10E+01	1.00E+01	0.00E+00	1.00E+01	9.00E+00	0.00E+00	7.63E+01	3	5	7	10
4136	Dexter	6.00E+02	2.00E+04	0.00E+00	0.00E+00	2.00E+04	2.00E+00	0.00E+00	1.46E+06	584	3921	5713	8595
4137	Dorothea	1.15E+03	1.00E+05	0.00E+00	0.00E+00	1.00E+05	2.00E+00	0.00E+00	8.88E+02	16008	52645	63830	81381
4541	Diabetes130US	1.02E+05	4.90E+01	1.30E+01	3.60E+01	2.47E+03	3.00E+00	2.44E+03	1.20E+16	1	2	2	2
4552	BachChoralHa...	5.66E+03	1.60E+01	2.00E+00	1.40E+01	1.04E+02	1.02E+02	7.80E+01	1.40E+03	1	1	1	1
23380	cjs	2.80E+03	3.30E+01	3.10E+01	2.00E+00	9.80E+01	6.00E+00	6.70E+01	6.30E+02	1	10	15	22
23512	higgs	9.80E+04	2.80E+01	2.80E+01	0.00E+00	2.80E+01	2.00E+00	0.00E+00	1.96E+01	9	18	22	26
40497	thyroid-ann	3.77E+03	2.10E+01	2.10E+01	0.00E+00	2.10E+01	3.00E+00	0.00E+00	6.60E-01	3	10	12	15
40685	shuttle	5.80E+04	9.00E+00	9.00E+00	0.00E+00	9.00E+00	7.00E+00	0.00E+00	5.67E+04	1	2	3	6
40691	wine-quality...	1.60E+03	1.10E+01	1.10E+01	0.00E+00	1.10E+01	6.00E+00	0.00E+00	1.20E+03	1	1	2	2
40900	Satellite	5.10E+03	3.60E+01	3.60E+01	0.00E+00	3.60E+01	2.00E+00	0.00E+00	7.40E+03	15	31	34	36
40926	CIFAR_10_small	2.00E+04	3.07E+03	3.07E+03	0.00E+00	3.07E+03	1.00E+01	0.00E+00	1.24E+07	1359	2711	2890	3035
40971	collins	1.00E+03	2.20E+01	2.00E+01	2.00E+00	3.70E+01	3.00E+01	1.50E+01	8.34E+04	1	1	1	1
40975	car	1.73E+03	6.00E+00	0.00E+00	6.00E+00	2.10E+01	4.00E+00	2.10E+01	4.25E+00	10	19	20	21
41026	gisette	7.00E+03	5.00E+03	0.00E+00	0.00E+00	5.00E+03	2.00E+00	0.00E+00	1.29E+03	981	2542	2998	3746
41064	convex	5.80E+04	7.84E+02	7.84E+02	0.00E+00	7.84E+02	2.00E+00	0.00E+00	1.81E+02	368	692	737	774
41065	mnist_rotation	6.20E+04	7.84E+02	7.84E+02	0.00E+00	7.84E+02	1.00E+01	0.00E+00	5.39E+01	356	695	739	775
41066	secom	1.57E+03	5.90E+02	5.90E+02	0.00E+00	5.90E+02	2.00E+00	0.00E+00	9.31E+07	2	6	8	28
41143	jasmine	2.98E+03	1.44E+02	8.00E+00	1.36E+02	2.80E+02	2.00E+00	0.00E+00	3.98E+01	53	163	194	239
41146	sylvine	5.12E+03	2.00E+01	2.00E+01	0.00E+00	2.00E+01	2.00E+00	0.00E+00	1.08E+07	3	5	5	7
41164	fabert	8.24E+03	8.00E+02	8.00E+02	0.00E+00	8.00E+02	7.00E+00	0.00E+00	2.79E+00	222	572	648	735
41946	Sick_numeric	3.77E+03	2.90E+01	2.90E+01	0.00E+00	2.90E+01	2.00E+00	0.00E+00	2.81E+04	2	3	4	6

TABLE 2: Overview of all datasets under consideration together with their meta-features.

3 Detailed Analysis of Atomic Algorithm Runtime Prediction

Here we shed more light on the prediction performances of the atomic algorithms under conditions different than in the main paper.

3.1 Basic Prediction Performance

The first analysis is concerned with a more detailed view on the prediction performance of the regression models on different input sizes than the large input size of the main paper. Fig. 3 shows the same results as in the main paper but for different grid sizes. In the main paper, we consider a rather large reference dataset of size $50,000 \times 1,000$. Here, we consider smaller datasets of sizes 5000×100 and $10,000 \times 100$. Naturally, for these datasets, the runtimes are generally much lower. At the same time, the predictions are more accurate as well.

Looking at those boxplots, we can see that most observations are in the upper plot rows, indicating short ground truth runtimes. At the same time, the inter-quartile ranges are extremely small, and many boxplots do not even have whiskers. This tells us that predictions are extremely precise in those cases. Looking at the concrete numbers, in fact most runtimes are in the range of under 5s, and the predictions are also in the same range.

Note that the predictions are generated with the *same* model as the predictions in the main paper and hence indicate a very good flexibility of the predictors. The predictors are not only able to predict the high runtimes as shown in the main paper but are also accurate in predicting very short runtime. In fact, they are even more precise for the small runtimes. It hardly ever occurs that a predictor believes that a short-running algorithm would run for a long time.

While this trend might lead one to believe that the learner is simply more focused on short runtimes, we think that this is not the case. Recall that the training examples are all over the map of different input sizes. There is no focus on a particular region, and the learners do not "know" in which region they will be evaluated. Having this rather homogeneous distribution of training examples in the input space in mind, one might actually rather expect the learners to specialize on high runtimes, because these are more likely to induce high errors.

The opposite is the case, and our explanation for this is the difficulty of the learning problem also discussed in Section 4.1 of the main paper: The higher the runtimes in general, the more weird and intense the (sometimes non-deterministic) side effects become. There is no surprise that the regressor has difficulties to perform perfectly if the execution of the same algorithm on the same data once takes 1 minute and once takes 2 minutes whereas there is little problem if it requires 1 second instead of 2 seconds. The fact that we observe, for identical input sizes, extraordinary runtime differences among different input datasets tells us that there are data-intrinsic properties we cannot (still) grasp and make the runtime prediction a tough problem.

We have not made the attempt yet to compare these observations to the results for the Oboe framework [26]. That paper suggests a prediction performance that is, compared with what we can achieve here, incredibly high. This raises the question why the predictability reported there is so much better than the one we report in our paper. We do not believe that there is a simple answer in sight currently. While we assume that the results in [26] have been prepared with utmost care, our research is likewise 100% reproducible, and we are as well sure that our observations are correct. At this point, our intuition for the different results is that this may be a programming language issue since Oboe was evaluated with sklearn algorithms, which *may* be more reproducible in terms of runtimes. We have made a lot of runtime observations that are *far* from what one would have expected based on its meta-features. Since no learner can discriminate these cases without additional features, we suppose that such effects did not occur in [26], which then has its roots in the implementation of the learning algorithms.

However, without challenging the results in [26], we believe that homogeneous runtimes are generally a rather unexpected phenomenon and consider our own observations as a realistic image of what happens in practice. Unless machines are totally idle, one would typically always expect deviating runtimes, even for iterative calls to the same routine. A lot of aspects argue in favor of this view, especially in environments like Java in which the runtime environment conducts a lot of optimizations at runtime, adopting sophisticated cache technologies etc. Maybe runtimes are well predictable if run on an idle machine with empty memory and with a fresh environment for each call. Probably Python is a bit more reproducible in this sense than Java. In any case, we find it somewhat more normal that runtime observations exhibit variance, and this will always impose a challenge to the learner.

This being said, we believe that this topic opens quite a range of interesting questions that could and maybe should studied independently. Especially given the fact that runtime predictions for machine learning algorithms is a so little studied field. First, it would be interesting to study the runtime variance of the learners depending on the load of the machine in general and whether repeated runs of the same algorithm (in the same environment) will have the same speed or whether there are side effects imposed by the runtime environment. Second, we already observed that some learners are more prone to these "hidden" properties in the data than others. For example, decision tables exhibit very heterogeneous runtimes for different input data of same size. Decision stumps on the other hand are very stable and have almost no variance. It would be interesting to analyze in more depth which learners are more robust than others in this sense and to try to find the features or properties of the data that explain these runtime differences of a single learner. The latter would then allow us to create a new and more informative feature.

All these observations also point into the direction of more individualized models. It is entirely clear that using the same meta-feature setup for all atomic algorithms cannot be globally optimal. Identifying optimal models for each atomic algorithm is a thrilling Meta-AutoML task itself. When addressing this problem, it would be important to define a clear methodology that explains how that AutoML approach should be used to find the respective attributes. Since AutoML tools are currently not good at *extracting* fea-

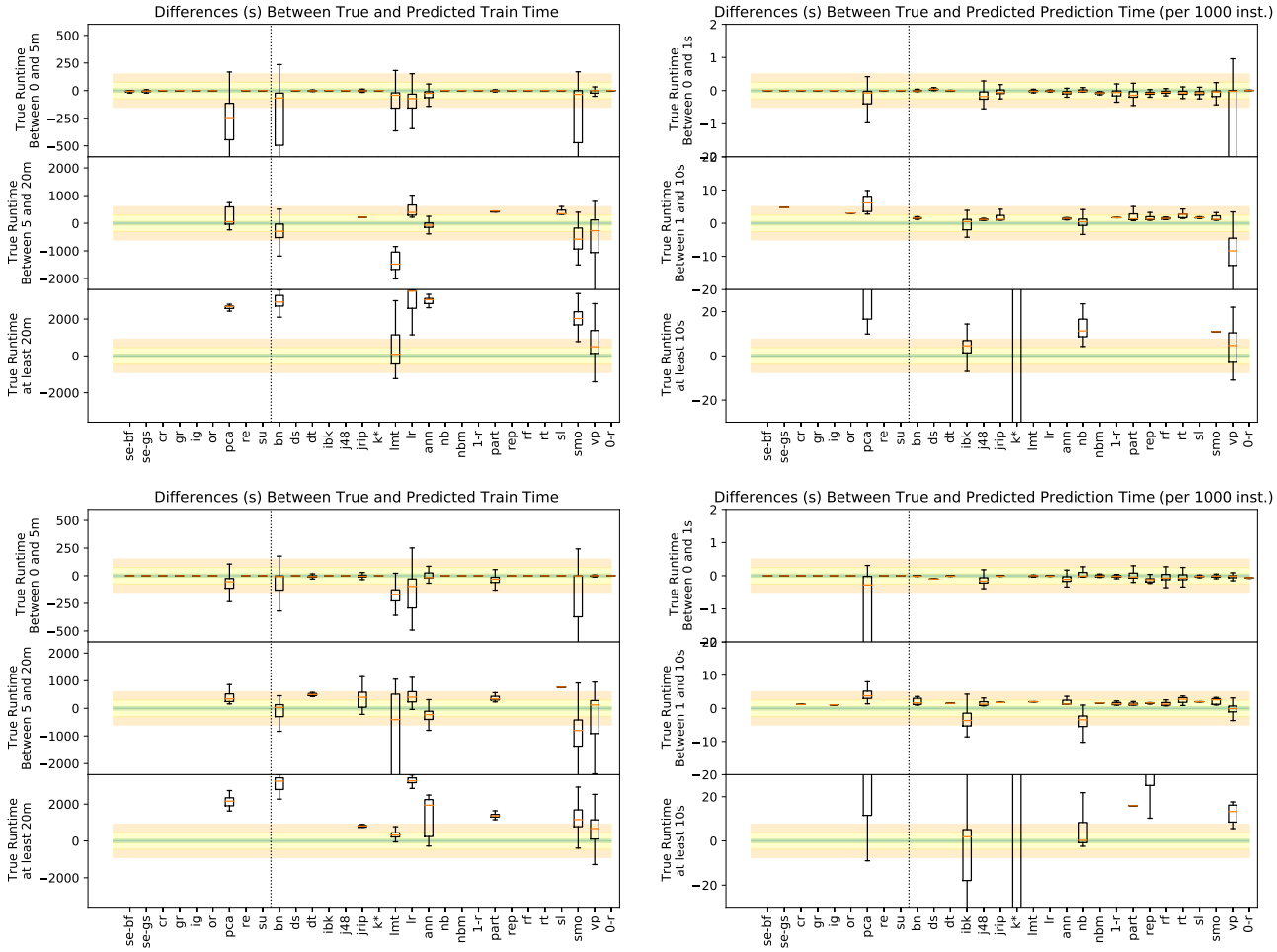


Fig. 3: Prediction results for smaller datasets than in the main paper. Above: 5000×100, below: 10000×100.

tures, it would be important to pre-define a quite extensive set of dataset meta-features that are the initially computed and passed to the system.

Of course, once we come up with features that do not depend any longer on the number of instances and attributes, the overall prediction solution becomes also more complex since we need to predict those features. Recall that we need to predict the value for each meta-feature for the *output* of a pre-processor. Clearly, this can become a challenging or even infeasible task. However, at this point, this is a highly speculative topic, and in a first step it would be a great advance to get better predictive models, regardless the challenges this would impose to pipeline runtime prediction.

3.2 Improvements By Posterior Models

We now provide some more details on the impact of the posterior model. The main paper shows that there is substantial improvement for most learners when using the runtime of the algorithm under default configuration as a landmark feature. While the main paper only compares the different models against each other in a very summarized fashion, we here complement those results by showing how the other plots would have looked like if drawn with the posterior model.

First, Fig. 4 shows the boxplots of the different prediction errors. It is clearly visible that the errors can be reduced extremely in the posterior model. In particular, the errors for models including logistic regression (such as lmt and logistic regression itself) and some rule based approaches such as jrip and part substantially benefit from the additional feature.

Second, Fig. 5 illustrates that these improvements have a direct impact onto the correctness of the rejection decision. The upper plot is the same as in the main paper and serves as a reference. The lower plot shows how our predictor would have decided with the posterior model. While there are still a couple of wrong decisions, these can be substantially reduced for the learners mentioned above. These observations suggest to use the posterior model whenever possible.

4 Functionality of Spy Wrappers

In order to gather the feedback of the spy components injected into the meta-learners, we use a separate database table in which every single member of the meta-learner registers itself when being used the first time and then sends observations during the training and prediction phases of the meta-learner. For each copy of the spy base learner, we

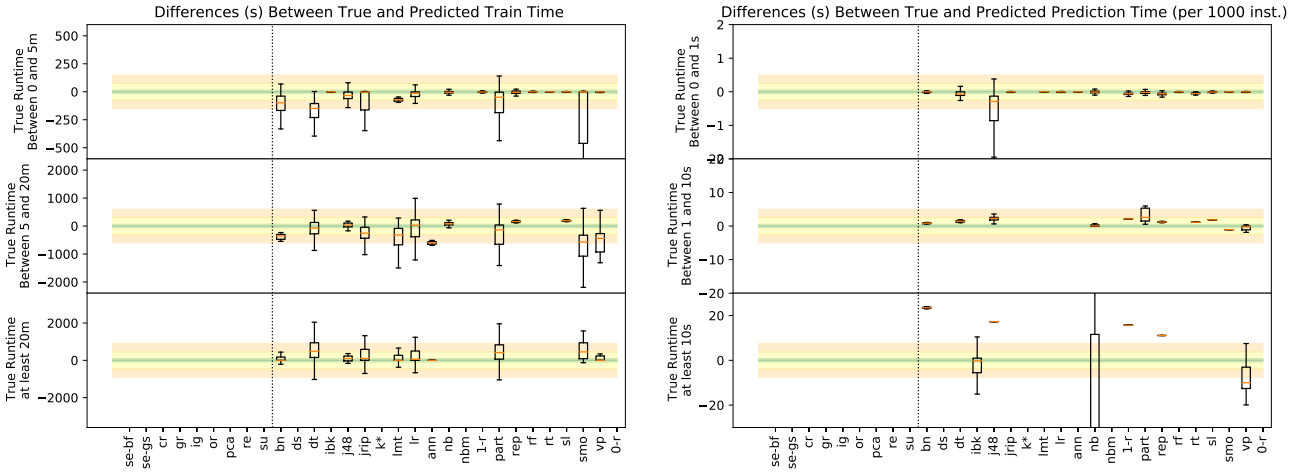


Fig. 4: Prediction performance under the posterior model. Semantics are identical to above and the main paper.

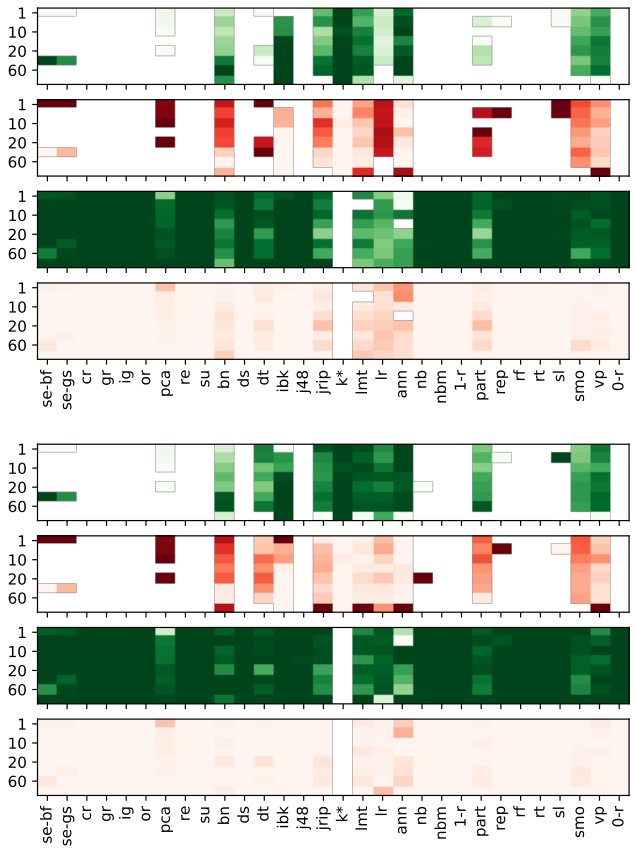


Fig. 5: Top plot: Standard model (same plot as in paper, just as reference). Bottom plot: Correct (green) and incorrect (red) decisions when using the *posterior* model.

track events of invocations for the start and stop times for training and prediction respectively. In order to understand in which of the phases of the meta-learner the events occur, the spy wrapper is informed from the outside once the meta-learner has returned control from the training invocation to the executing algorithm. This way, it can clearly assign each event to one of the two phases and hence allows to understand time contributions to the two different models.

With the above framework, we can assess three important metrics for *each base learner instance* in isolation:

1) the *number of invocations* of the training and prediction routines respectively,
2) the *overall runtime* of the base learner for training and prediction, and
3) the features of the data effectively passed to the base learner.

Knowing the original data used by the meta-learner and its parameters, we can use these statistics to study the behavior of the meta-learner in terms of the parameters used in Eq. (1) of the paper.

Note that while the relationship between algorithm parameters and meta-features afterwards is sometimes deterministic, it still makes sense to use a model for the prediction of those parameters in the eventual approach. One argument is simply with respect to the implementation: It is easier and cleaner to treat all predictors as models instead of hard-coding certain rules that are only valid under specific circumstances. A second more conceptual argument is that, using models, one assures that the rule one would otherwise code is really sustained by observed data.

5 Hierarchical Guard: Build and Usage

In Fig. 6 the processes for fitting (offline) and applying (online) the hierarchical guard. On the left hand side, the steps for fitting the hierarchical guard are displayed. The initial step is to run the calibration module in order to determine coefficients for linearly scaling the predictions to the current system. As described in the paper, to determine these coefficients we sample a few algorithms and parametrizations of those for which the constraint on the runtime in the training data is fulfilled (not being too fast but also not too slow). Afterward, we use the training data for the hierarchical guard to fit models to predict the runtime of the base learners ($M_\theta, M_{-\theta}, M_{\theta+d}$) and pre-processors.

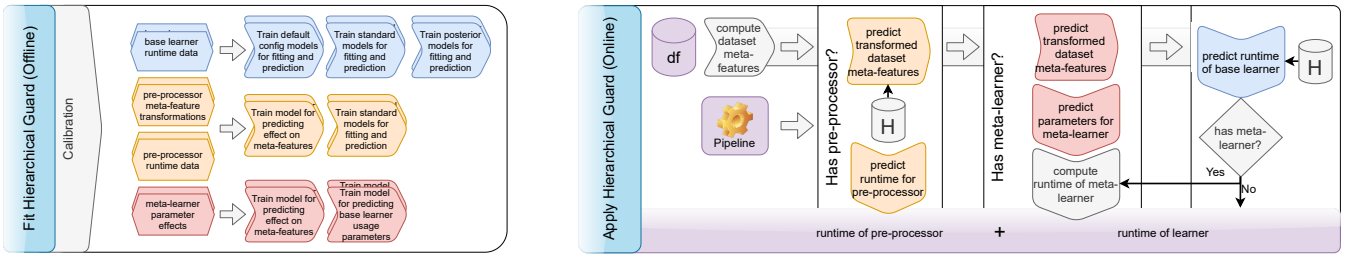


Fig. 6: Visualization of the steps that are executed for fitting a hierarchical guard offline (left) and how the runtime of a pipeline is predicted using the hierarchical safe guard.

Furthermore, we use a different set of meta-data to fit models to predict, how specific parametrizations of pre-processors and meta-learners transform the meta-features of data sets. More specifically these models predict whether the respective algorithms change the number of attributes.

Lastly, we fit models for predicting the coefficients of Eq. (2) which are needed to calculate the runtime of a meta-learner that is configured with a certain parametrization of a base learner. To this end, we simply reuse the models fitted for the base learners as described above.

On the right-hand side, it is presented how the runtime of a pipeline is predicted applying the models fitted on the left side. First of all we compute the dataset meta-features for the original data. If the given pipeline contains a pre-processor, we use the runtime prediction model for the corresponding pre-processor to estimate its fit time (in case of supervised pre-processors) and its runtime for transforming the data. Furthermore, we predict the transformed dataset meta-features in order to forward them to the next step.

The next action is to check whether the pipeline contains a meta-learner. If this evaluates to true, we first predict the transformed dataset meta-features and predict the coefficients for Eq. (2). To give an estimate on the runtime of the meta-learner we plug all the predicted coefficients as well as the runtime predicted for the base learner (based on the predicted transformed dataset meta-features) into the equation.

For predicting the runtime of a base learner, we first of all inspect the parametrization of the base learner and determine whether it is in default configuration or not. In the former case we use the default model $M_{-\theta}$, while in the latter case, we either apply the standard model M_{θ} or the posterior model $M_{\theta+d}$,[1] depending on the configuration of the guard. However, if the history H contains an entry for this specific parametrization and dataset meta-features, this value is simply returned instead of making a prediction.

Finally, to obtain the runtime of the entire pipeline, we add the runtimes predicted for the (optional) pre-processor and the learner.

References

[1] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD Explorations*, vol. 11, 2009.

[2] M. Wever, 2020. [Online]. Available: https://mvnrepository.com/artifact/ai.libs.thirdparty/interruptible-weka/0.1.5

1. In Sec. 8 we preferred the posterior model over the standard model due to the more accurate predictions.

[3] M. A. Hall, "Correlation-based feature subset selection for machine learning," Ph.D. dissertation, University of Waikato, Hamilton, New Zealand, 1998.

[4] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[5] R. C. Holte, "Very simple classification rules perform well on most commonly used datasets," *Machine learning*, vol. 11, no. 1, pp. 63–90, 1993.

[6] K. Pearson, "Liii. on lines and planes of closest fit to systems of points in space," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 2, no. 11, pp. 559–572, 1901.

[7] I. Kononenko, "Estimating attributes: analysis and extensions of relief," in *European conference on machine learning*. Springer, 1994, pp. 171–182.

[8] K. Kira, L. A. Rendell *et al.*, "The feature selection problem: Traditional methods and a new algorithm," in *Aaai*, vol. 2, 1992, pp. 129–134.

[9] R. Kohavi, "The power of decision tables," in *European conference on machine learning*. Springer, 1995, pp. 174–189.

[10] W. W. Cohen, "Fast effective rule induction," in *Machine learning proceedings 1995*. Elsevier, 1995, pp. 115–123.

[11] J. G. Cleary and L. E. Trigg, "K*: An instance-based learner using an entropic distance measure," in *Machine Learning Proceedings 1995*. Elsevier, 1995, pp. 108–114.

[12] N. Landwehr, M. Hall, and E. Frank, "Logistic model trees," *Machine learning*, vol. 59, no. 1-2, pp. 161–205, 2005.

[13] J. Friedman, T. Hastie, R. Tibshirani *et al.*, "Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors)," *The annals of statistics*, vol. 28, no. 2, pp. 337–407, 2000.

[14] S. le Cessie and J. van Houwelingen, "Ridge estimators in logistic regression," *Applied Statistics*, vol. 41, no. 1, pp. 191–201, 1992.

[15] G. H. John and P. Langley, "Estimating continuous distributions in bayesian classifiers," in *Eleventh Conference on Uncertainty in Artificial Intelligence*. San Mateo: Morgan Kaufmann, 1995, pp. 338–345.

[16] A. Mccallum and K. Nigam, "A comparison of event models for naive bayes text classification," in *AAAI-98 Workshop on 'Learning for Text Categorization'*, 1998.

[17] R. Holte, "Very simple classification rules perform well on most commonly used datasets," *Machine Learning*, vol. 11, pp. 63–91, 1993.

[18] E. Frank and I. H. Witten, "Generating accurate rule sets without global optimization," in *Fifteenth International Conference on Machine Learning*, J. Shavlik, Ed. Morgan Kaufmann, 1998, pp. 144–151.

[19] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[20] M. Sumner, E. Frank, and M. Hall, "Speeding up logistic model tree induction," in *9th European Conference on Principles and Practice of Knowledge Discovery in Databases*. Springer, 2005, pp. 675–683.

[21] J. Platt, "Fast training of support vector machines using sequential minimal optimization," in *Advances in Kernel Methods - Support Vector Learning*, B. Schoelkopf, C. Burges, and A. Smola, Eds. MIT Press, 1998. [Online]. Available: http://research.microsoft.com/~jplatt/smo.html

[22] Y. Freund and R. E. Schapire, "Large margin classification using the perceptron algorithm," in *11th Annual Conference on Computational Learning Theory*. New York, NY: ACM Press, 1998, pp. 209–217.

[23] J. Vanschoren, J. N. van Rijn, B. Bischl, and L. Torgo, "OpenML: Networked science in machine learning," *SIGKDD Explorations*, vol. 15, 2013.
[24] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *NeurIPS*, 2015.
[25] N. V. C. et. al., "Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002.
[26] C. Yang, Y. Akimoto, D. W. Kim, and M. Udell, "Oboe: Collaborative filtering for automl model selection," in *SIGKDD*, 2019.

10 Conclusion and Open Questions

According to the structure of the thesis, we draw conclusions and discuss open questions in two parts. First, we focus on AutoML itself and, in particular, AutoML for multi-label classification. Second, we address the topic of improving the efficiency and efficacy of AutoML systems.

In the first part of the thesis, i.e., in Chapters 3 to 6, we have devised a novel AutoML system based on hierarchical task network planning for a natural representation of hierarchical dependencies in the configuration space of machine learning pipelines (cf. Chapter 3). Furthermore, we demonstrated this search space representation to be flexible enough for dealing with machine learning pipelines that are unlimited in length (cf. Chapter 4) as well as for the configuration of multi-label classifiers (cf. Chapter 5). Especially in the multi-label classification scenario, where the configuration space is strongly characterized by hierarchical structures, this type of search space representation combined with a best-first search turns out to be very promising.

However, it has also become clear in Chapter 6 that the high dimensionality of the underlying AutoML problem for multi-label classification presents significant challenges to well-established optimization approaches such as Bayesian optimization and Hyperband. The high dimensionality of the CASH problem renders most optimization algorithms more or less ineffective. Although most of these methods still perform significantly better than a random search, a *greedy* best-first search algorithm proves to be the most beneficial in the experiments.

A common assumption made in the AutoML literature concerns the dependencies between decisions, i.e., algorithm choices and values for hyper-parameters, that require a joint consideration of the CASH problem. More specifically, it is assumed that, for example, fixing a pre-processing algorithm affects the optimal decision for learning algorithms and vice versa. The same assumption is made for the optimization of hyper-parameters. While these assumptions are certainly valid in theory, the question still is whether they are not unnecessarily obstructive in tackling the AutoML problem and whether a more pragmatic solution would work better in practice.

In particular, the research question arises whether the complexity that comes with the high dimensionality of the search space can be made more manageable. For example, via a divide and conquer strategy, the search space could be divided into smaller search spaces in which known optimization algorithms can still operate effectively [Tor+21]. Alternatively, one could also try to learn how to make the search space complexity more manageable by means of meta-learning [PS19], trading in theoretical optimality. Hence, an open research question is whether the search space can be safely pruned, i.e., without excluding the optimum, or in a way that the search space still contains a near-optimal solution. From a pragmatic perspective, the question remains whether an ”unsafe“ pruning still leads to better solutions or competitive solutions being found within a shorter time because of the reduced search space complexity.

Furthermore, it is questionable whether AutoML should really be considered as a black-box optimization problem or not since knowledge about what is being configured is completely ignored. Leveraging experience or expert knowledge from data scientists, about the provided training data, or about the algorithms which are combined into machine learning pipelines is difficult in a black-box optimization setting. Per definition of black-box optimization, it is assumed that nothing is known about the function itself. While this simplifies the AutoML problem on a conceptual level, it also deliberately ignores potentially valuable information. Making this knowledge accessible to black-box optimization algorithms is usually a non-trivial endeavor. In [MW21] we propose a very naive and easy-to-implement approach to AutoML, which, despite its simplicity, is highly competitive to state-of-the-art AutoML systems. The basic idea of this work is to consider the AutoML process as a modular step-by-step procedure. Each step focuses on a specific decision, e.g., choosing a basic learning algorithm or choosing a pre-processing algorithm. Thereby, incorporating expert knowledge becomes easier than integrating it into black-box optimization algorithms. The good performance of this approach suggests that opening the black box seems to be a promising direction.

The second part of this thesis (Chapters 7 to 9) dealt with how to increase the efficacy – through additional degrees of freedom in the configuration of learners – and the efficiency of AutoML systems. In Chapter 7 and 8 respectively, it turned out that by optimizing the structure of a nested dichotomy or by optimizing the base learners of binary relevance learning for each label individually, the generalization performance can be improved significantly. Even if this means a considerable additional effort for the configuration of such a learner, because of the extra degrees of freedom, this optimization can be automated and thus integrated with AutoML systems. Furthermore, we presented in Chapter 9 a meta-learning approach for predicting the runtimes of machine learning pipelines in order to prevent the execution of machine learning pipelines that would take too long.

Obviously, proposing to extend the search space contradicts the previous discussion concerning the problem of the high dimensional AutoML search space. Therefore, an open question is which decisions should be included in this search space. Related to this question is, first of all, the one about the importance of hyper-parameters of multi-label classification methods. While the choice of the base learner is acknowledged as an important hyper-parameter (cf. [Riv+20], Chapter 7), if not the most crucial hyper-parameter, it is still an open question how important other hyper-parameters of multi-label classifiers are.

Another open research question is to what extent considering runtime predictions of multi-label classifiers helps to avoid classifiers that require excessive time for evaluation. Avoiding such candidates would prevent the optimization process from stalling, and one would expect more solution candidates to be explored within a fixed time budget.

11 Epilog – On-The-Fly Computing for Machine Learning Services

As already mentioned at the beginning of the thesis, this work is motivated by our work in the collaborative research center 901 with the title “On-The-Fly Machine Learning”, we elaborate on the vision of on-the-fly computing for machine learning services in the following.

On-the-fly (OTF) computing refers to a computing paradigm dealing with the automatic, on-the-fly configuration and provision of customized IT services, which is investigated in the eponymous collaborative research center (CRC) 901[1] [Hap+13; Kar+20]. To this end, the IT services are composed of base services, which are available in worldwide markets, by so-called OTF providers and tailored to the specific needs of a customer. The research on methods for the configuration and provision of such services is accompanied by the investigation of methods for

- quality assurance,
- protection of market participants,
- target-oriented development of markets, and
- supporting interaction between participants

that account for the specific characteristics of such dynamically changing markets, also referred to as OTF markets.

In the following, we provide more details on the participants and the structure of OTF markets in Section 11.1 as well as the use case scenario of on-the-fly machine learning (Section 11.2), which can be seen as an extension of AutoML, as introduced in Section 2.1.

11.1 On-The-Fly Markets

The overall ecosystem of an OTF market comprises many entities ranging from underlying hardware in so-called OTF compute centers to managing entities for

[1] https://sfb901.upb.de (accessed 2021-04-21)

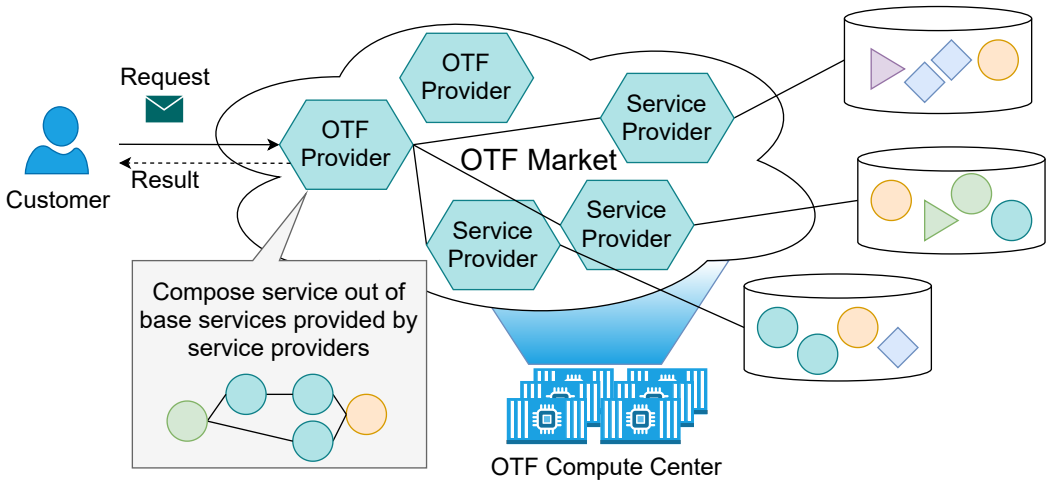


Figure 11.1: Simplified illustration of an OTF market, involving the roles customer, OTF provider, and service provider. In this illustration, a customer sends a request into the OTF market, which is received and processed by an OTF provider. The OTF provider answers the customer's request by composing a novel service out of base services provided by the service providers to meet the customer's requirements. The result, which might be the composed service itself or the output obtained by executing it, is eventually returned to the customer.

setting up markets to (human) participants of the market, which are grouped into roles [Jaz+17]. In the following, we focus on the roles of *customers*, *OTF providers* and *service providers* since those are the ones primarily involved in the process of on-the-fly provisioning software services.

Customer As is implied by the term "on-the-fly", in an OTF market, the requested services are provided on demand. More specifically, after a customer sends out a request for a service to the market, the request is processed, and the requested service is composed just then to fulfill the requirements stated in the request. A customer may either be interested in the result of a computation given a specific input, provided with the request, or in the composed service itself. In the latter case, the customer is provided access to the requested service, for example, via a graphical interface or an application programming interface (API).

OTF Provider The OTF provider is responsible for receiving, processing, and answering requests. While processing requests means "understanding" what is requested by the user, answering the requests involves the automatic composition of the requested service out of base services that are made available in the market by service providers.

Service Provider A service provider maintains a repository of base services and provides information about and access to these base services to OTF providers.

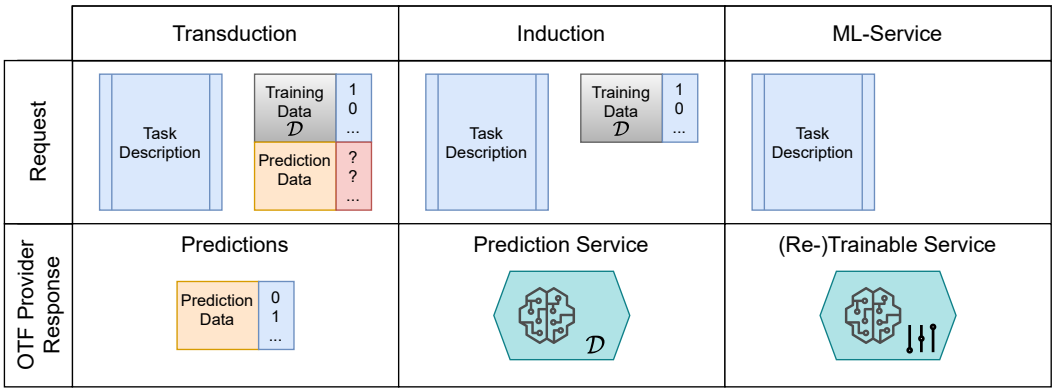


Figure 11.2: Comparison of different scenarios in on-the-fly machine learning. The scenarios differ in what is contained in the request and what needs to be provided by a customer as well as the desired output, which ranges from one time predictions to a service customized for a provided data set $\mathcal{D}$ that can be repeatedly used for making predictions on new data points to a trainable machine learning service. In the latter scenario, the customer does not provide any data to the OTF provider but trains the service his- or herself.

Obviously, the functionality that can be provided to the user is highly dependent on what base services are provided by service providers.

An illustration of these roles and their interaction is displayed in Figure 11.1.

11.2 On-The-Fly Machine Learning

While OTF computing in general deals with all kinds of IT services, a specific instance of the OTF paradigm solely dealing with machine learning services is referred to as on-the-fly machine learning (OTF-ML) [MWH18c; Moh+19]. In this particular scenario, the customer is provided services with machine learning functionality that needs to be tailored to the data in question, i.e., the customer's data for which he or she needs the machine learning functionality. To this end, the market provides machine learning algorithms as base services, and OTF providers compose these services into service-based machine learning pipelines, where the general concept of a machine learning pipeline remains the same but, instead of algorithms, the services representing the respective algorithms are composed into a machine learning service pipeline [Moh+18b; MWH18a].

Generally speaking, we distinguish three different types of on-the-fly machine learning services that can be requested by a customer [Moh+19]:

Transduction In the transduction scenario, along with a task description, the customer provides a training data set $\mathcal{D}$ together with a set of data points

for which the customer is interested in obtaining predictions, referred to as prediction data in Figure 11.2. The answer to this request consists of the predictions for the given prediction data.

Induction In the induction scenario, the customer requests a service that can be used (repeatedly) for labeling new data points. To this end, the customer sends a request containing a task description and training data for which the customer wants to obtain a corresponding machine learning service.

ML-Service In this last scenario, the customer only specifies the task the desired machine learning service is meant to accomplish. Without providing data, the task of the OTF provider is thus to provide a customized machine learning service that anyhow performs well for this specific task once data for training is provided.

Deploying AutoML services in an OTF environment offers many advantages. First, the cloud infrastructure offers more flexible and performant computational resources that can be used to achieve a high degree of parallelization for individual AutoML processes, allowing for a faster exploration of the search space. Furthermore, the abstraction from the platform through services allows for building cross-platform machine learning pipelines, i.e., machine learning pipelines comprising algorithms that are only available for certain platforms. More specifically, on a service level, it is possible to combine algorithms implemented in C, Python, and Java into a single pipeline.

Besides building machine learning service pipelines from scratch via AutoML, service providers may also offer pre-trained machine learning services specialized in specific tasks, e.g., object recognition for image data. While there might be multiple services offering the same functionality, they may differ in non-functional properties, say, financial costs and accuracy. For example, one service might be very cheap but sometimes very inaccurate, and another service is very accurate on average, but this also comes at a higher cost. Furthermore, other services might be available with non-functional properties ranging between those two extremes. In [CZZ20a; CZZ20b], it is shown that combining such services and deciding for each data point individually which service to use can reduce the overall costs while keeping the quality of predictions competitive to the most accurate single service. Moreover, combining the predictions of multiple services might even result in higher accuracy, as demonstrated in [CZZ21] within a multi-label classification setting.

Further References

[Ber+11] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. „Algorithms for Hyper-Parameter Optimization“. In: *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain*. 2011, pp. 2546–2554. URL: http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization (cit. on pp. 2, 12).

[Bis+16] Bernd Bischl, Michel Lang, Lars Kotthoff, et al. „mlr: Machine Learning in R“. In: *J. Mach. Learn. Res.* 17 (2016), 170:1–170:5. URL: http://jmlr.org/papers/v17/15-066.html (cit. on p. 10).

[Bog+21] Jasmin Bogatinovski, Ljupčo Todorovski, Sašo Džeroski, and Dragi Kocev. „Comprehensive Comparative Study of Multi-Label Classification Methods“. In: *arXiv preprint arXiv:2102.07113* (2021) (cit. on p. 32).

[Bou+04] Matthew R Boutell, Jiebo Luo, Xipeng Shen, and Christopher M Brown. „Learning multi-label scene classification“. In: *Pattern recognition* 37.9 (2004), pp. 1757–1771. DOI: 10.1016/j.patcog.2004.03.009. URL: https://www.sciencedirect.com/science/article/abs/pii/S0031320304001074 (cit. on p. 31).

[BST06] Zafer Barutcuoglu, Robert E Schapire, and Olga G Troyanskaya. „Hierarchical multi-label prediction of gene function“. In: *Bioinformatics* 22.7 (2006), pp. 830–836. DOI: 10.1093/bioinformatics/btk048 (cit. on p. 2).

[BYC13] James Bergstra, Dan Yamins, and David D Cox. „Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms“. In: *Proceedings of the 12th Python in Science Conference*. Citeseer. 2013, pp. 13–20. URL: https://pdfs.semanticscholar.org/d4f4/9717c9adb46137f49606ebbdf17e3598b5a5.pdf (cit. on p. 12).

[Cab+11] Ricardo S Cabral, Fernando Torre, Joao P Costeira, and Alexandre Bernardino. „Matrix completion for multi-label image classification“. In: *Advances in neural information processing systems*. Citeseer. 2011, pp. 190–198 (cit. on p. 2).

[Cha+08] Guillaume M JB Chaslot, Mark HM Winands, H JAAP VAN DEN HERIK, Jos WHM Uiterwijk, and Bruno Bouzy. „Progressive strategies for Monte-Carlo tree search“. In: *New Mathematics and Natural Computation* 4.03 (2008), pp. 343–357 (cit. on p. 20).

[Che+18] Boyuan Chen, Harvey Wu, Warren Mo, Ishanu Chattopadhyay, and Hod Lipson. „Autostacker: a compositional evolutionary learning system“. In: *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2018, Kyoto, Japan, July 15-19, 2018*. ACM, 2018, pp. 402–409. DOI: 10.1145/3205455.3205586 (cit. on p. 18).

[CK01] Amanda Clare and Ross D King. „Knowledge discovery in multi-label phenotype data“. In: *European conference on principles of data mining and knowledge discovery*. Springer. 2001, pp. 42–53 (cit. on p. 32).

[CML13] Josu Ceberio, Alexander Mendiburu, and José Antonio Lozano. „The Plackett-Luce ranking model on permutation-based optimization problems“. In: *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2013, Cancun, Mexico, June 20-23, 2013*. IEEE, 2013, pp. 494–501. DOI: 10.1109/CEC.2013.6557609 (cit. on p. 20).

[Cra85] Nichael Lynn Cramer. „A Representation for the Adaptive Generation of Simple Sequential Programs“. In: *Proceedings of the 1st International Conference on Genetic Algorithms, Pittsburgh, PA, USA, July 1985*. Lawrence Erlbaum Associates, 1985, pp. 183–187 (cit. on p. 17).

[CZZ20a] Lingjiao Chen, Matei Zaharia, and James Zou. „To Call or not to Call? Using ML Prediction APIs more Accurately and Economically“. In: *International workshop on Economics of Privacy and Data Labor at ICML*. 2020 (cit. on p. 162).

[CZZ20b] Lingjiao Chen, Matei Zaharia, and James Y. Zou. „FrugalML: How to use ML Prediction APIs more accurately and cheaply“. In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/789ba2ae4d335e8a2ad283a3f7effced-Abstract.html (cit. on p. 162).

[CZZ21] Lingjiao Chen, Matei Zaharia, and James Zou. „FrugalMCT: Efficient Online ML API Selection for Multi-Label Classification Tasks“. In: *CoRR* abs/2102.09127 (2021). arXiv: 2102.09127. URL: https://arxiv.org/abs/2102.09127 (cit. on p. 162).

[Dem+12] Krzysztof Dembczyński, Willem Waegeman, Weiwei Cheng, and Eyke Hüllermeier. „On label dependence and loss minimization in multi-label classification“. In: *Machine Learning* 88.1-2 (2012), pp. 5–45 (cit. on pp. 28, 29, 31).

[Dip+05] Sotiris Diplaris, Grigorios Tsoumakas, Pericles A Mitkas, and Ioannis Vlahavas. „Protein classification with multiple algorithms“. In: *Panhellenic Conference on Informatics*. Springer. 2005, pp. 448–456 (cit. on p. 31).

[Dro+18] Iddo Drori, Yamuna Krishnamurthy, Remi Rampin, et al. „AlphaD3M: Machine learning pipeline synthesis“. In: *International workshop on automatic machine learning at ICML*. 2018 (cit. on p. 20).

[Dro+19] Iddo Drori, Lu Liu, Yi Nian, et al. „AutoML using Metadata Language Embeddings“. In: *CoRR* abs/1910.03698 (2019). arXiv: 1910.03698. URL: http://arxiv.org/abs/1910.03698 (cit. on p. 22).

[DSR18] Silvia Cristina Nunes das Dores, Carlos Soares, and Duncan D. Ruiz. „Bandit-Based Automated Machine Learning“. In: *7th Brazilian Conference on Intelligent Systems, BRACIS 2018, São Paulo, Brazil, October 22-25, 2018*. IEEE Computer Society, 2018, pp. 121–126. DOI: 10.1109/BRACIS.2018.00029 (cit. on p. 15).

[EHW16] Frank Eibe, Mark A. Hall, and Ian Witten. *The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques", Morgan Kaufmann*. 2016 (cit. on pp. 10, 11).

[EMH19] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. „Neural Architecture Search: A Survey“. In: *J. Mach. Learn. Res.* 20 (2019), 55:1–55:21. URL: http://jmlr.org/papers/v20/18-598.html (cit. on p. 22).

[Eri+20] Nick Erickson, Jonas Mueller, Alexander Shirkov, et al. „AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data“. In: *CoRR* abs/2003.06505 (2020). arXiv: 2003.06505. URL: https://arxiv.org/abs/2003.06505 (cit. on p. 16).

[Fer+13] David Ferrucci, Anthony Levas, Sugato Bagchi, David Gondek, and Erik T Mueller. „Watson: beyond jeopardy!“ In: *Artificial Intelligence* 199 (2013), pp. 93–105 (cit. on p. 1).

[Fer12] David A Ferrucci. „Introduction to "this is watson"“. In: *IBM Journal of Research and Development* 56.3.4 (2012), pp. 1–1 (cit. on p. 1).

[Feu+15] Matthias Feurer, Aaron Klein, Katharina Eggensperger, et al. „Efficient and Robust Automated Machine Learning“. In: *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*. 2015, pp. 2962–2970. URL: http://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning (cit. on pp. 2, 12, 14, 21).

[Feu+18] Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. „Practical automated machine learning for the automl challenge 2018“. In: *ICML 2018 AutoML Workshop*. 2018. URL: https://ml.informatik.uni-freiburg.de/papers/18-AUTOML-AutoChallenge.pdf (cit. on p. 16).

[FH18] Matthias Feurer and Frank Hutter. „Towards further automation in automl“. In: *ICML 2018 AutoML Workshop*. 2018. URL: https://ml.informatik.uni-freiburg.de/papers/18-AUTOML-AutoAutoML.pdf (cit. on p. 14).

[FKH18] Stefan Falkner, Aaron Klein, and Frank Hutter. „BOHB: Robust and Efficient Hyperparameter Optimization at Scale“. In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Vol. 80. Proceedings of Machine Learning Research. PMLR, 2018, pp. 1436–1445. URL: http://proceedings.mlr.press/v80/falkner18a.html (cit. on p. 15).

[Fra18] Peter I. Frazier. „A Tutorial on Bayesian Optimization“. In: *CoRR* abs/1807.02811 (2018). arXiv: 1807.02811. URL: http://arxiv.org/abs/1807.02811 (cit. on p. 12).

[FSE18] Nicoló Fusi, Rishit Sheth, and Melih Elibol. „Probabilistic Matrix Factorization for Automated Machine Learning“. In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. 2018, pp. 3352–3361. URL: https://proceedings.neurips.cc/paper/2018/hash/b59a51a3c0bf9c5228fde841714f523a-Abstract.html (cit. on p. 22).

[Gij+19] Pieter Gijsbers, Erin LeDell, Janek Thomas, et al. „An Open Source AutoML Benchmark“. In: *CoRR* abs/1907.00909 (2019). arXiv: 1907.00909. URL: http://arxiv.org/abs/1907.00909 (cit. on p. 12).

[Gil+18] Yolanda Gil, Ke-Thia Yao, Varun Ratnakar, et al. „P4ML: A phased performance-based pipeline planner for automated machine learning“. In: *ICML 2018 AutoML Workshop*. 2018 (cit. on p. 12).

[GNT04] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated planning - theory and practice*. Elsevier, 2004 (cit. on pp. 2, 18, 19).

[Gru94] Frédéric Gruau. „Automatic Definition of Modular Neural Networks“. In: *Adapt. Behav.* 3.2 (1994), pp. 151–183. DOI: 10.1177/105971239400300202 (cit. on p. 23).

[GV19] Pieter Gijsbers and Joaquin Vanschoren. „GAMA: Genetic Automated Machine learning Assistant“. In: *J. Open Source Softw.* 4.33 (2019), p. 1132. DOI: 10.21105/joss.01132 (cit. on p. 18).

[Hap+13] Markus Happe, Friedhelm Meyer auf der Heide, Peter Kling, Marco Platzner, and Christian Plessl. „On-The-Fly Computing: A novel paradigm for individualized IT services“. In: *16th IEEE International Symposium on Object / Component / Service-Oriented Real-Time Distributed Computing, ISORC 2013, Paderborn, Germany, June 19-21, 2013*. IEEE Computer Society, 2013, pp. 1–10. DOI: 10.1109/ISORC.2013.6913232 (cit. on pp. 2, 159).

[Hei+13] Dominik Heider, Robin Senge, Weiwei Cheng, and Eyke Hüllermeier. „Multi-label classification for exploiting cross-resistance information in HIV-1 drug resistance prediction“. In: *Bioinformatics* 29.16 (2013), pp. 1946–1952 (cit. on p. 2).

[HHL11] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. „Sequential Model-Based Optimization for General Algorithm Configuration“. In: *Learning and Intelligent Optimization - 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers*. Vol. 6683. Lecture Notes in Computer Science. Springer, 2011, pp. 507–523. DOI: 10.1007/978-3-642-25566-3_40 (cit. on p. 12).

[HSG89] Steven A. Harp, Tariq Samad, and Aloke Guha. „Designing Application-Specific Neural Networks Using the Genetic Algorithm“. In: *Advances in Neural Information Processing Systems 2, [NIPS Conference, Denver, Colorado, USA, November 27-30, 1989]*. Morgan Kaufmann, 1989, pp. 447–454. URL: http://papers.nips.cc/paper/263-designing-application-specific-neural-networks-using-the-genetic-algorithm (cit. on p. 23).

[Hül+22] Eyke Hüllermeier, Marcel Wever, Eneldo Loza Mencıéa, Johannes Fürnkranz, and Michael Rapp. „A Flexible Class of Dependence-aware Multi-Label Loss Functions“. In: *Machine Learning* (2022), pp. 1–25. DOI: 10.1007/s10994-021-06107-2 (cit. on pp. 27, 28).

[HZC21] Xin He, Kaiyong Zhao, and Xiaowen Chu. „AutoML: A survey of the state-of-the-art“. In: *Knowl. Based Syst.* 212 (2021), p. 106622. DOI: 10.1016/j.knosys.2020.106622 (cit. on p. 9).

[ID20] Deloitte AI Institute and Media & Telecommunications the Deloitte Center for Technology. *Deloitte's State of AI in the Enterprise, 3rd Edition*. Retrieved from: https://www2.deloitte.com/content/dam/Deloitte/de/Documents/technology-media-telecommunications/TMT_State-of-AI-2020.pdf (Accessed 2021-05-11). 2020 (cit. on p. 1).

[Jaz+17] Bahar Jazayeri, Olaf Zimmermann, Gregor Engels, and Dennis Kundisch. „A Variability Model for Store-Oriented Software Ecosystems: An Enterprise Perspective“. In: *Service-Oriented Computing - 15th International Conference, ICSOC 2017, Malaga, Spain, November 13-16, 2017, Proceedings*. Vol. 10601. Lecture Notes in Computer Science. Springer, 2017, pp. 573–588. DOI: 10.1007/978-3-319-69035-3_42 (cit. on p. 160).

[JM15] Michael I Jordan and Tom M Mitchell. „Machine learning: Trends, perspectives, and prospects“. In: *Science* 349.6245 (2015), pp. 255–260. DOI: 10.1126/science.aaa8415 (cit. on p. 1).

[JSG21] Hadi S. Jomaa, Lars Schmidt-Thieme, and Josif Grabocka. „Dataset2Vec: learning dataset meta-features“. In: *Data Min. Knowl. Discov.* 35.3 (2021), pp. 964–985. DOI: 10.1007/s10618-021-00737-9 (cit. on p. 22).

[JSH19] Haifeng Jin, Qingquan Song, and Xia Hu. „Auto-Keras: An Efficient Neural Architecture Search System“. In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*. ACM, 2019, pp. 1946–1956. DOI: 10.1145/3292500.3330648 (cit. on p. 14).

[JSW98] Donald R. Jones, Matthias Schonlau, and William J. Welch. „Efficient Global Optimization of Expensive Black-Box Functions“. In: *J. Global Optimization* 13.4 (1998), pp. 455–492. DOI: 10.1023/A:1008306431147 (cit. on p. 13).

[JT16] Kevin G. Jamieson and Ameet Talwalkar. „Non-stochastic Best Arm Identification and Hyperparameter Optimization“. In: *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, AISTATS 2016, Cadiz, Spain, May 9-11, 2016*. 2016, pp. 240–248. URL: http://proceedings.mlr.press/v51/jamieson16.html (cit. on p. 14).

[Kar+20] Holger Karl, Dennis Kundisch, Friedhelm Meyer auf der Heide, and Heike Wehrheim. „A Case for a New IT Ecosystem: On-The-Fly Computing“. In: *Bus. Inf. Syst. Eng.* 62.6 (2020), pp. 467–481. DOI: 10.1007/s12599-019-00627-x (cit. on p. 159).

[Kat+20] Michael Katz, Parikshit Ram, Shirin Sohrabi, and Octavian Udrea. „Exploring Context-Free Languages via Planning: The Case for Automating Machine Learning“. In: *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling, Nancy, France, October 26-30, 2020*. AAAI Press, 2020, pp. 403–411. URL: https://aaai.org/ojs/index.php/ICAPS/article/view/6686 (cit. on p. 19).

[KBE19] Brent Komer, James Bergstra, and Chris Eliasmith. „Hyperopt-Sklearn“. In: *Automated Machine Learning - Methods, Systems, Challenges*. The Springer Series on Challenges in Machine Learning. Springer, 2019, pp. 97–111. DOI: 10.1007/978-3-030-05318-5_5 (cit. on p. 14).

[Kie+12] Jörg-Uwe Kietz, Floarea Serban, Abraham Bernstein, and Simon Fischer. „Designing KDD-Workflows via HTN-Planning“. In: *ECAI 2012 - 20th European Conference on Artificial Intelligence. Including Prestigious Applications of Artificial Intelligence (PAIS-2012) System Demonstrations Track, Montpellier, France, August 27-31 , 2012*. Vol. 242. Frontiers in Artificial Intelligence and Applications. IOS Press, 2012, pp. 1011–1012. DOI: 10.3233/978-1-61499-098-7-1011 (cit. on p. 19).

[KKS13] Zohar Shay Karnin, Tomer Koren, and Oren Somekh. „Almost Optimal Exploration in Multi-Armed Bandits“. In: *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*. 2013, pp. 1238–1246. URL: http://proceedings.mlr.press/v28/karnin13.html (cit. on p. 14).

[Koc+07] Dragi Kocev, Celine Vens, Jan Struyf, and Saso Dzeroski. „Ensembles of Multi-Objective Decision Trees“. In: *Machine Learning: ECML 2007, 18th European Conference on Machine Learning, Warsaw, Poland, September 17-21, 2007, Proceedings*. Vol. 4701. Lecture Notes in Computer Science. Springer, 2007, pp. 624–631. DOI: 10.1007/978-3-540-74958-5_61 (cit. on p. 32).

[Kot+17] Lars Kotthoff, Chris Thornton, Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. „Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA“. In: *J. Mach. Learn. Res.* 18 (2017), 25:1–25:5. URL: http://jmlr.org/papers/v18/16-261.html (cit. on pp. 2, 12, 14).

[Koz95] John R Koza. „Survey of genetic algorithms and genetic programming“. In: *Wescon conference record*. WESTERN PERIODICALS COMPANY. 1995, pp. 589–594 (cit. on p. 17).

[LFM20] Trang T. Le, Weixuan Fu, and Jason H. Moore. „Scaling tree-based automated machine learning to biomedical big data with a feature set selector“. In: *Bioinform.* 36.1 (2020), pp. 250–256. DOI: 10.1093/bioinformatics/btz470 (cit. on p. 18).

[Li+17] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. „Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization“. In: *J. Mach. Learn. Res.* 18 (2017), 185:1–185:52. URL: http://jmlr.org/papers/v18/16-558.html (cit. on p. 15).

[Lin19] Jiali Lin. „ML-ReinBo: Machine learning pipeline search with reinforcement learning and Bayesian optimization“. PhD thesis. 2019 (cit. on p. 20).

[Liu+20] Sijia Liu, Parikshit Ram, Deepak Vijaykeerthy, et al. „An ADMM Based Framework for AutoML Pipeline Configuration“. In: *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2020, pp. 4892–4899. URL: https://aaai.org/ojs/index.php/AAAI/article/view/5926 (cit. on p. 16).

[LP20] Erin LeDell and Sébastien Poirier. „H2o automl: Scalable automatic machine learning“. In: *ICML 2020 AutoML Workshop*. 2020. URL: https://www.automl.org/wp-content/uploads/2020/07/AutoML_2020_paper_61.pdf (cit. on p. 12).

[Lua+12] Oscar Luaces, Jorge Dıéez, José Barranquero, Juan José del Coz, and Antonio Bahamonde. „Binary relevance efficacy for multilabel classification“. In: *Progress in Artificial Intelligence* 1.4 (2012), pp. 303–313 (cit. on p. 31).

[MBH21] Felix Mohr, Viktor Bengs, and Eyke Hüllermeier. „Single Player Monte-Carlo Tree Search Based on the Plackett-Luce Model“. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 2021 (cit. on p. 20).

[McK+10] Robert I. McKay, Nguyen Xuan Hoai, Peter Alexander Whigham, Yin Shan, and Michael O'Neill. „Grammar-based Genetic Programming: a survey“. In: *Genetic Programming and Evolvable Machines* 11.3-4 (2010), pp. 365–396. DOI: 10.1007/s10710-010-9109-y (cit. on p. 17).

[MF08] Eneldo Loza Mencia and Johannes Fürnkranz. „Efficient pairwise multilabel classification for large-scale problems in the legal domain“. In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2008, pp. 50–65 (cit. on p. 2).

[Moc77] Jonas Mockus. „On Bayesian Methods for Seeking the Extremum and their Application“. In: *Information Processing, Proceedings of the 7th IFIP Congress 1977, Toronto, Canada, August 8-12, 1977*. 1977, pp. 195–200. URL: https://link.springer.com/content/pdf/10.1007/978-3-662-38527-2_55.pdf (cit. on p. 13).

[Moh+18a] Felix Mohr, Theodor Lettmann, Eyke Hüllermeier, and Marcel Wever. „Programmatic task network planning“. In: *Proceedings of the 1st ICAPS Workshop on Hierarchical Planning*. 2018, pp. 31–39 (cit. on p. 19).

[Moh+18b] Felix Mohr, Marcel Wever, Eyke Hüllermeier, and Amin Faez. „(WIP) Towards the Automated Composition of Machine Learning Services“. In: *2018 IEEE International Conference on Services Computing, SCC 2018, San Francisco, CA, USA, July 2-7, 2018*. IEEE, 2018, pp. 241–244. DOI: 10.1109/SCC.2018.00039 (cit. on p. 161).

[Moh+19] Felix Mohr, Marcel Wever, Alexander Tornede, and Eyke Hüllermeier. „From Automated to On-The-Fly Machine Learning“. In: LNI P-294 (2019), pp. 273–274. DOI: 10.18420/inf2019_40 (cit. on p. 161).

[Moh+21] Felix Mohr, Marcel Wever, Alexander Tornede, and Eyke Hüllermeier. „Predicting Machine Learning Pipeline Runtimes in the Context of Automated Machine Learning“. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43.9 (2021), pp. 3055–3066. DOI: 10.1109/TPAMI.2021.3056950 (cit. on pp. 4, 21).

[MS19] Mohamed Maher and Sherif Sakr. „SmartML: A Meta Learning-Based Framework for Automated Selection and Hyperparameter Tuning for Machine Learning Algorithms“. In: *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*. OpenProceedings.org, 2019, pp. 554–557. DOI: 10.5441/002/edbt.2019.54 (cit. on pp. 14, 21).

[MW21] Felix Mohr and Marcel Wever. *Naive Automated Machine Learning - A Late Baseline for AutoML*. 2021. arXiv: 2103.10496. URL: https://arxiv.org/abs/2103.10496 (cit. on p. 156).

[MWH18a] Felix Mohr, Marcel Wever, and Eyke Hüllermeier. „Automated Machine Learning Service Composition“. In: *CoRR* abs/1809.00486 (2018). arXiv: 1809.00486. URL: http://arxiv.org/abs/1809.00486 (cit. on p. 161).

[MWH18b] Felix Mohr, Marcel Wever, and Eyke Hüllermeier. „ML-Plan: Automated machine learning via hierarchical planning“. In: *Mach. Learn.* 107.8-10 (2018), pp. 1495–1515. DOI: 10.1007/s10994-018-5735-z (cit. on pp. 3, 19).

[MWH18c] Felix Mohr, Marcel Wever, and Eyke Hüllermeier. „On-the-Fly Service Construction with Prototypes“. In: *2018 IEEE International Conference on Services Computing, SCC 2018, San Francisco, CA, USA, July 2-7, 2018*. IEEE, 2018, pp. 225–232. DOI: 10.1109/SCC.2018.00036 (cit. on p. 161).

[Nam+14] Jinseok Nam, Jungi Kim, Eneldo Loza Mencıéa, Iryna Gurevych, and Johannes Fürnkranz. „Large-scale multi-label text classification—revisiting neural networks“. In: *Joint european conference on machine learning and knowledge discovery in databases*. Springer. 2014, pp. 437–452 (cit. on p. 2).

[Nam+19] Jinseok Nam, Young-Bum Kim, Eneldo Loza Mencia, et al. „Learning context-dependent label permutations for multi-label classification“. In: *International Conference on Machine Learning*. PMLR. 2019, pp. 4733–4742 (cit. on p. 35).

[NF00] Stefano Nolfi and Dario Floreano. *Evolutionary robotics: The biology, intelligence, and technology of self-organizing machines*. MIT press, 2000 (cit. on p. 23).

[Ols+16] Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. „Evaluation of a Tree-based Pipeline Optimization Tool for Automating Data Science“. In: *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference, Denver, CO, USA, July 20 - 24, 2016*. ACM, 2016, pp. 485–492. DOI: 10.1145/2908812.2908918 (cit. on pp. 2, 18).

[Par+19] Laurent Parmentier, Olivier Nicol, Laetitia Jourdan, and Marie-Eléonore Kessaci. „TPOT-SH: A Faster Optimization Algorithm to Solve the AutoML Problem on Large Datasets“. In: *31st IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2019, Portland, OR, USA, November 4-6, 2019*. IEEE, 2019, pp. 471–478. DOI: 10.1109/ICTAI.2019.00072 (cit. on p. 18).

[Ped+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, et al. „Scikit-learn: Machine Learning in Python“. In: *J. Mach. Learn. Res.* 12 (2011), pp. 2825–2830. URL: http://dl.acm.org/citation.cfm?id=2078195 (cit. on pp. 10, 11).

[PN19] Arjun Pakrashi and Brian Mac Namee. „CascadeML: An Automatic Neural Network Architecture Evolution and Training Algorithm for Multi-label Classification (Best Technical Paper)“. In: *Artificial Intelligence XXXVI - 39th SGAI International Conference on Artificial Intelligence, AI 2019, Cambridge, UK, December 17-19, 2019, Proceedings*. Vol. 11927. Lecture Notes in Computer Science. Springer, 2019, pp. 3–17. DOI: 10.1007/978-3-030-34885-4_1 (cit. on p. 22).

[PS19] Valerio Perrone and Huibin Shen. „Learning search spaces for Bayesian optimization: Another view of hyperparameter transfer learning“. In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. 2019, pp. 12751–12761. URL: https://proceedings.neurips.cc/paper/2019/hash/6ea3f1874b188558fafbab78e8c3a968-Abstract.html (cit. on p. 156).

[Qi+07] Guo-Jun Qi, Xian-Sheng Hua, Yong Rui, et al. „Correlative multi-label video annotation“. In: *Proceedings of the 15th ACM international conference on Multimedia*. 2007, pp. 17–26 (cit. on p. 2).

[Rap+20] Michael Rapp, Eneldo Loza Mencıéa, Johannes Fürnkranz, Vu-Linh Nguyen, and Eyke Hüllermeier. „Learning gradient boosted multi-label classification rules“. In: *arXiv preprint arXiv:2006.13346* (2020) (cit. on p. 32).

[Rea+11] Jesse Read, Bernhard Pfahringer, Geoff Holmes, and Eibe Frank. „Classifier chains for multi-label classification“. In: *Mach. Learn.* 85.3 (2011), pp. 333–359. DOI: 10.1007/s10994-011-5256-5 (cit. on p. 32).

[Rea+16] Jesse Read, Peter Reutemann, Bernhard Pfahringer, and Geoff Holmes. „MEKA: A Multi-label/Multi-target Extension to Weka“. In: *Journal of Machine Learning Research* 17.21 (2016), pp. 1–5. URL: http://jmlr.org/papers/v17/12-164.html (cit. on p. 11).

[Rea+17] Esteban Real, Sherry Moore, Andrew Selle, et al. „Large-Scale Evolution of Image Classifiers“. In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017, pp. 2902–2911. URL: http://proceedings.mlr.press/v70/real17a.html (cit. on p. 22).

[Rea+21] Jesse Read, Bernhard Pfahringer, Geoffrey Holmes, and Eibe Frank. „Classifier chains: a review and perspectives“. In: *Journal of Artificial Intelligence Research* 70 (2021), pp. 683–718 (cit. on pp. 32, 35).

[Ren+20] Pengzhen Ren, Yun Xiao, Xiaojun Chang, et al. „A Comprehensive Survey of Neural Architecture Search: Challenges and Solutions“. In: *CoRR* abs/2006.02903 (2020). arXiv: 2006.02903. URL: https://arxiv.org/abs/2006.02903 (cit. on p. 22).

[Riv+20] Adriano Rivolli, Jesse Read, Carlos Soares, Bernhard Pfahringer, and André CPLF de Carvalho. „An empirical analysis of binary transformation strategies and base algorithms for multi-label learning“. In: *Machine Learning* 109.8 (2020), pp. 1509–1563 (cit. on pp. 1, 157).

[RSS19] Herilalaina Rakotoarison, Marc Schoenauer, and Michèle Sebag. „Automated Machine Learning with Monte-Carlo Tree Search“. In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*. ijcai.org, 2019, pp. 3296–3303. DOI: 10.24963/ijcai.2019/457 (cit. on p. 20).

[RW14] Cynthia Rudin and Kiri L. Wagstaff. „Machine learning for science and society“. In: *Mach. Learn.* 95.1 (2014), pp. 1–9. DOI: 10.1007/s10994-013-5425-9 (cit. on p. 1).

[Sá+17] Alex Guimarães Cardoso de Sá, Walter José G. S. Pinto, Luiz Otávio Vilas Boas Oliveira, and Gisele L. Pappa. „RECIPE: A Grammar-Based Framework for Automatically Evolving Classification Pipelines“. In: *Genetic Programming - 20th European Conference, EuroGP 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings*. Vol. 10196. Lecture Notes in Computer Science. 2017, pp. 246–261. DOI: 10.1007/978-3-319-55696-3_16 (cit. on pp. 2, 18).

[SCH12] Robin Senge, Juan José del Coz, and Eyke Hüllermeier. „On the Problem of Error Propagation in Classifier Chains for Multi-label Classification“. In: *Data Analysis, Machine Learning and Knowledge Discovery - Proceedings of the 36th Annual Conference of the Gesellschaft für Klassifikation e. V., Hildesheim, Germany, August 2012*. Studies in Classification, Data Analysis, and Knowledge Organization. Springer, 2012, pp. 163–170. DOI: 10.1007/978-3-319-01595-8_18 (cit. on p. 32).

[SFP18] Alex Guimarães Cardoso de Sá, Alex Alves Freitas, and Gisele L. Pappa. „Automated Selection and Configuration of Multi-Label Classification Algorithms with Grammar-Based Genetic Programming“. In: *Parallel Problem Solving from Nature - PPSN XV - 15th International Conference, Coimbra, Portugal, September 8-12, 2018, Proceedings, Part II*. Vol. 11102. Lecture Notes in Computer Science. Springer, 2018, pp. 308–320. DOI: 10.1007/978-3-319-99259-4_25 (cit. on p. 18).

[Sil+16] David Silver, Aja Huang, Chris J Maddison, et al. „Mastering the game of Go with deep neural networks and tree search“. In: *nature* 529.7587 (2016), pp. 484–489 (cit. on p. 1).

[Sil+17] David Silver, Julian Schrittwieser, Karen Simonyan, et al. „Mastering the game of go without human knowledge“. In: *nature* 550.7676 (2017), pp. 354–359 (cit. on pp. 1, 20).

[SLA12] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. „Practical Bayesian Optimization of Machine Learning Algorithms“. In: *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*. 2012, pp. 2960–2968. URL: https://proceedings.neurips.cc/paper/2012/hash/05311655a15b75fab86956663e1819cd-Abstract.html (cit. on p. 12).

[SLB19] Xudong Sun, Jiali Lin, and Bernd Bischl. „ReinBo: Machine Learning Pipeline Conditional Hierarchy Search and Configuration with Bayesian Optimization Embedded Reinforcement Learning“. In: *Machine Learning and Knowledge Discovery in Databases - International Workshops of ECML PKDD 2019, Würzburg, Germany, September 16-20, 2019, Proceedings, Part I*. Vol. 1167. Communications in Computer and Information Science. Springer, 2019, pp. 68–84. DOI: 10.1007/978-3-030-43823-4_7 (cit. on p. 20).

[Sli19] Aleksandrs Slivkins. „Introduction to Multi-Armed Bandits“. In: *Found. Trends Mach. Learn.* 12.1-2 (2019), pp. 1–286. DOI: 10.1561/2200000068 (cit. on p. 14).

[SPF17] Alex Guimarães Cardoso de Sá, Gisele L. Pappa, and Alex Alves Freitas. „Towards a method for automatically selecting and configuring multi-label classification algorithms“. In: *Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings*. ACM, 2017, pp. 1125–1132. DOI: 10.1145/3067695.3082053 (cit. on p. 16).

[SS00] Robert E Schapire and Yoram Singer. „BoosTexter: A boosting-based system for text categorization“. In: *Machine learning* 39.2 (2000), pp. 135–168 (cit. on p. 2).

[Swe+17] Thomas Swearingen, Will Drevo, Bennett Cyphers, et al. „ATM: A distributed, collaborative, scalable system for automated machine learning“. In: *2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11-14, 2017*. IEEE Computer Society, 2017, pp. 151–162. DOI: 10.1109/BigData.2017.8257923 (cit. on p. 16).

[SZ11] Chris Sanden and John Z Zhang. „Enhancing multi-label music genre classification through ensemble techniques“. In: *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*. 2011, pp. 705–714 (cit. on p. 2).

[Tai+14] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. „Deepface: Closing the gap to human-level performance in face verification“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, pp. 1701–1708 (cit. on p. 1).

[TCB18] Janek Thomas, Stefan Coors, and Bernd Bischl. „Automatic Gradient Boosting“. In: *CoRR* abs/1807.03873 (2018). arXiv: 1807.03873. URL: http://arxiv.org/abs/1807.03873 (cit. on p. 14).

[Tho+13] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. „Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms“. In: *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*. ACM, 2013, pp. 847–855. DOI: 10.1145/2487575.2487629 (cit. on pp. 1, 2, 7, 8, 12, 14).

[TK07] Grigorios Tsoumakas and Ioannis Katakis. „Multi-Label Classification: An Overview“. In: *IJDWM* 3.3 (2007), pp. 1–13. DOI: 10.4018/jdwm.2007070101 (cit. on p. 31).

[TKV09] Grigorios Tsoumakas, Ioannis Katakis, and Ioannis Vlahavas. „Mining multi-label data“. In: *Data mining and knowledge discovery handbook*. Springer, 2009, pp. 667–685 (cit. on pp. 2, 11).

[TKV10] Grigorios Tsoumakas, Ioannis Katakis, and Ioannis P. Vlahavas. „Mining Multi-label Data“. In: *Data Mining and Knowledge Discovery Handbook, 2nd ed*. Springer, 2010, pp. 667–685. DOI: 10.1007/978-0-387-09823-4_34 (cit. on p. 23).

[Tor+20b] Tanja Tornede, Alexander Tornede, Marcel Wever, Felix Mohr, and Eyke Hüllermeier. „AutoML for Predictive Maintenance: One Tool to RUL Them All“. In: *IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning - Second International Workshop, IoT Streams 2020, and First International Workshop, ITEM 2020, Co-located with ECML/PKDD 2020, Ghent, Belgium, September 14-18, 2020, Revised Selected Papers*. Vol. 1325. Communications in Computer and Information Science. Springer, 2020, pp. 106–118. DOI: 10.1007/978-3-030-66770-2_8 (cit. on p. 20).

[Tor+21] Tanja Tornede, Alexander Tornede, Marcel Wever, and Eyke Hüllermeier. „Coevolution of Remaining Useful Lifetime Estimation Pipelines for Automated Predictive Maintenance“. In: *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2021, Lille, France, July 10-14, 2021*. ACM, 2021 (cit. on pp. 18, 156).

[TWH20a] Alexander Tornede, Marcel Wever, and Eyke Hüllermeier. „Extreme Algorithm Selection with Dyadic Feature Representation“. In: *Discovery Science - 23rd International Conference, DS 2020, Thessaloniki, Greece, October 19-21, 2020, Proceedings*. Vol. 12323. Lecture Notes in Computer Science. Springer, 2020, pp. 309–324. DOI: 10.1007/978-3-030-61527-7_21. URL: https://doi.org/10.1007/978-3-030-61527-7%5C_21 (cit. on pp. 8, 22).

[Van18] Joaquin Vanschoren. „Meta-Learning: A Survey“. In: *CoRR* abs/1810.03548 (2018). arXiv: 1810.03548. URL: http://arxiv.org/abs/1810.03548 (cit. on p. 9).

[Var+15] Gaël Varoquaux, Lars Buitinck, Gilles Louppe, et al. „Scikit-learn: Machine Learning Without Learning the Machinery“. In: *GetMobile Mob. Comput. Commun.* 19.1 (2015), pp. 29–33. DOI: 10.1145/2786984.2786995 (cit. on p. 11).

[Wev+19] Marcel Dominik Wever, Felix Mohr, Alexander Tornede, and Eyke Hüllermeier. „Automating multi-label classification extending ml-plan“. In: *ICML 2019 AutoML Workshop*. 2019. URL: https://www.automl.org/wp-content/uploads/2019/06/automlws2019_Paper46.pdf (cit. on pp. 3, 20, 34).

[Wev+20] Marcel Wever, Alexander Tornede, Felix Mohr, and Eyke Hüllermeier. „LiBRe: Label-Wise Selection of Base Learners in Binary Relevance for Multi-label Classification“. In: *Advances in Intelligent Data Analysis XVIII - 18th International Symposium on Intelligent Data Analysis, IDA 2020, Konstanz, Germany, April 27-29, 2020, Proceedings*. Vol. 12080. Lecture Notes in Computer Science. **Frontier Prize**. Springer, 2020, pp. 561–573. DOI: 10.1007/978-3-030-44584-3_44 (cit. on pp. 1, 3).

[Wev+21] Marcel Wever, Alexander Tornede, Felix Mohr, and Eyke Hüllermeier. „AutoML for Multi-Label Classification: Overview and Empirical Evaluation“. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43.9 (2021), pp. 3037–3054. DOI: 10.1109/TPAMI.2021.3051276 (cit. on pp. 2, 3, 20).

[Wis18] Martin Wistuba. „Deep Learning Architecture Search by Neuro-Cell-Based Evolution with Function-Preserving Mutations“. In: *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2018, Dublin, Ireland, September 10-14, 2018, Proceedings, Part II*. Vol. 11052. Lecture Notes in Computer Science. Springer, 2018, pp. 243–258. DOI: 10.1007/978-3-030-10928-8_15 (cit. on p. 22).

[WKH19] W. Waegeman, K. Dembczynski, and E. Hüllermeier. „Multi-target prediction: A unifying view on problems and methods“. In: *Data Mining and Knowledge Discovery* 33.2 (2019), pp. 293–324 (cit. on p. 23).

[WMH17] Marcel Wever, Felix Mohr, and Eyke Hüllermeier. „Automatic Machine Learning: Hierarchical Planning Versus Evolutionary Optimization“. In: *Proceedings of the 27. Workshop Computational Intelligence*. **Young Author Award (Runner Up)**. 2017, pp. 149–166 (cit. on p. 19).

[WMH18a] Marcel Wever, Felix Mohr, and Eyke Hüllermeier. „Automated Multi-Label Classification based on ML-Plan“. In: *CoRR* abs/1811.04060 (2018). arXiv: 1811.04060. URL: http://arxiv.org/abs/1811.04060 (cit. on p. 20).

[WMH18b] Marcel Wever, Felix Mohr, and Eyke Hüllermeier. „Ensembles of evolved nested dichotomies for classification“. In: *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2018, Kyoto, Japan, July 15-19, 2018*. ACM, 2018, pp. 561–568. DOI: 10.1145/3205455.3205562 (cit. on pp. 4, 16).

[WMH18c] Marcel Dominik Wever, Felix Mohr, and Eyke Hüllermeier. „ML-Plan for unlimited-length machine learning pipelines“. In: *ICML 2018 AutoML Workshop*. 2018. URL: https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxhdXRvbWwyMDE4aWNtbHxneDo3M2Q3MjUzYjViNDRhZTAx (cit. on pp. 3, 19).

[WRP19] Martin Wistuba, Ambrish Rawat, and Tejaswini Pedapati. „A Survey on Neural Architecture Search“. In: *CoRR* abs/1905.01392 (2019). arXiv: 1905.01392. URL: http://arxiv.org/abs/1905.01392 (cit. on p. 22).

[WZ17] Xi-Zhu Wu and Zhi-Hua Zhou. „A unified view of multi-label performance measures“. In: *International Conference on Machine Learning*. PMLR. 2017, pp. 3780–3788 (cit. on p. 25).

[Xie+20] Lingxi Xie, Xin Chen, Kaifeng Bi, et al. „Weight-Sharing Neural Architecture Search: A Battle to Shrink the Optimization Gap“. In: *CoRR* abs/2008.01475 (2020). arXiv: 2008.01475. URL: https://arxiv.org/abs/2008.01475 (cit. on p. 22).

[Xue+11] Xiangyang Xue, Wei Zhang, Jie Zhang, et al. „Correlative multi-label multi-instance image annotation“. In: *2011 International Conference on Computer Vision*. IEEE. 2011, pp. 651–658 (cit. on p. 2).

[Yan+19] Chengrun Yang, Yuji Akimoto, Dae Won Kim, and Madeleine Udell. „OBOE: Collaborative Filtering for AutoML Model Selection“. In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*. ACM, 2019, pp. 1173–1183. DOI: 10.1145/3292500.3330909 (cit. on p. 22).

[Yao+18] Quanming Yao, Mengshuo Wang, Hugo Jair Escalante, et al. „Taking Human out of Learning Applications: A Survey on Automated Machine Learning“. In: *CoRR* abs/1810.13306 (2018). arXiv: 1810.13306. URL: http://arxiv.org/abs/1810.13306 (cit. on p. 9).

[ZH21] Marc-André Zöller and Marco F. Huber. „Benchmark and Survey of Automated Machine Learning Frameworks“. In: *J. Artif. Intell. Res.* 70 (2021), pp. 409–472. DOI: 10.1613/jair.1.11854 (cit. on p. 9).

[Zha+16] Yuyu Zhang, Mohammad Taha Bahadori, Hang Su, and Jimeng Sun. „FLASH: Fast Bayesian Optimization for Data Analytic Pipelines“. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. ACM, 2016, pp. 2065–2074. DOI: 10.1145/2939672.2939829 (cit. on p. 14).

[Zha+18] Min-Ling Zhang, Yu-Kun Li, Xu-Ying Liu, and Xin Geng. „Binary relevance for multi-label learning: an overview“. In: *Frontiers Comput. Sci.* 12.2 (2018), pp. 191–202. DOI: 10.1007/s11704-017-7031-7 (cit. on pp. 31, 32).

[Zha09] Min-Ling Zhang. „ML-RBF: RBF Neural Networks for Multi-Label Learning“. In: *Neural Processing Letters* 29.2 (2009), pp. 61–74 (cit. on p. 32).

[Zhu+19] Hui Zhu, Zhulin An, Chuanguang Yang, et al. „EENA: Efficient Evolution of Neural Architecture“. In: *2019 IEEE/CVF International Conference on Computer Vision Workshops, ICCV Workshops 2019, Seoul, Korea (South), October 27-28, 2019*. IEEE, 2019, pp. 1891–1899. DOI: 10.1109/ICCVW.2019.00238 (cit. on p. 22).

[ZZ14] Min-Ling Zhang and Zhi-Hua Zhou. „A Review on Multi-Label Learning Algorithms“. In: *IEEE Trans. Knowl. Data Eng.* 26.8 (2014), pp. 1819–1837. DOI: 10.1109/TKDE.2013.39 (cit. on pp. 2, 23).

Own Publications

Journal Articles

[MWH18b] Felix Mohr, Marcel Wever, and Eyke Hüllermeier. „ML-Plan: Automated machine learning via hierarchical planning“. In: *Mach. Learn.* 107.8-10 (2018), pp. 1495–1515. DOI: 10.1007/s10994-018-5735-z (cit. on pp. 3, 19).

[MSW19] Marie-Luis Merten, Nina Seemann, and Marcel Wever. „Grammatikwandel digital-kulturwissen-schaftlich erforscht. Mittelniederdeutscher Sprachausbau im interdisziplinären Zugriff“. In: *Niederdeutsches Jahrbuch* 142 (2019), pp. 124–146.

[WRH20] Marcel Wever, Lorijn van Rooijen, and Heiko Hamann. „Multioracle Coevolutionary Learning of Requirements Specifications from Examples in On-The-Fly Markets“. In: *Evol. Comput.* 28.2 (2020), pp. 165–193. DOI: 10.1162/evco_a_00266.

[Moh+21] Felix Mohr, Marcel Wever, Alexander Tornede, and Eyke Hüllermeier. „Predicting Machine Learning Pipeline Runtimes in the Context of Automated Machine Learning“. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43.9 (2021), pp. 3055–3066. DOI: 10.1109/TPAMI.2021.3056950 (cit. on pp. 4, 21).

[Wev+21] Marcel Wever, Alexander Tornede, Felix Mohr, and Eyke Hüllermeier. „AutoML for Multi-Label Classification: Overview and Empirical Evaluation“. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43.9 (2021), pp. 3037–3054. DOI: 10.1109/TPAMI.2021.3051276 (cit. on pp. 2, 3, 20).

[Hül+22] Eyke Hüllermeier, Marcel Wever, Eneldo Loza Mencıéa, Johannes Fürnkranz, and Michael Rapp. „A Flexible Class of Dependence-aware Multi-Label Loss Functions“. In: *Machine Learning* (2022), pp. 1–25. DOI: 10.1007/s10994-021-06107-2 (cit. on pp. 27, 28).

Conference Articles

[WRH17] Marcel Wever, Lorijn van Rooijen, and Heiko Hamann. „Active coevolutionary learning of requirements specifications from examples“. In: *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2017, Berlin, Germany, July 15-19, 2017*. **Best Paper Award**. ACM, 2017, pp. 1327–1334. DOI: 10.1145/3071178.3071258.

[MWH18c] Felix Mohr, Marcel Wever, and Eyke Hüllermeier. „On-the-Fly Service Construction with Prototypes“. In: *2018 IEEE International Conference on Services Computing, SCC 2018, San Francisco, CA, USA, July 2-7, 2018*. IEEE, 2018, pp. 225–232. DOI: 10.1109/SCC.2018.00036 (cit. on p. 161).

[MWH18d] Felix Mohr, Marcel Wever, and Eyke Hüllermeier. „Reduction Stumps for Multi-class Classification“. In: *Advances in Intelligent Data Analysis XVII - 17th International Symposium, IDA 2018, 's-Hertogenbosch, The Netherlands, October 24-26, 2018, Proceedings*. Vol. 11191. Lecture Notes in Computer Science. Springer, 2018, pp. 225–237. DOI: 10.1007/978-3-030-01768-2_19. URL: https://doi.org/10.1007/978-3-030-01768-2%5C_19.

[Moh+18b] Felix Mohr, Marcel Wever, Eyke Hüllermeier, and Amin Faez. „(WIP) Towards the Automated Composition of Machine Learning Services“. In: *2018 IEEE International Conference on Services Computing, SCC 2018, San Francisco, CA, USA, July 2-7, 2018*. IEEE, 2018, pp. 241–244. DOI: 10.1109/SCC.2018.00039 (cit. on p. 161).

[WMH18b] Marcel Wever, Felix Mohr, and Eyke Hüllermeier. „Ensembles of evolved nested dichotomies for classification“. In: *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2018, Kyoto, Japan, July 15-19, 2018*. ACM, 2018, pp. 561–568. DOI: 10.1145/3205455.3205562 (cit. on pp. 4, 16).

[Moh+19] Felix Mohr, Marcel Wever, Alexander Tornede, and Eyke Hüllermeier. „From Automated to On-The-Fly Machine Learning“. In: LNI P-294 (2019), pp. 273–274. DOI: 10.18420/inf2019_40 (cit. on p. 161).

[Han+20] Jonas Hanselle, Alexander Tornede, Marcel Wever, and Eyke Hüllermeier. „Hybrid Ranking and Regression for Algorithm Selection“. In: *KI 2020: Advances in Artificial Intelligence - 43rd German Conference on AI, Bamberg, Germany, September 21-25, 2020, Proceedings*. Vol. 12325. Lecture Notes in Computer Science. Springer, 2020, pp. 59–72. DOI: 10.1007/978-3-030-58285-2_5. URL: https://doi.org/10.1007/978-3-030-58285-2%5C_5.

[TWH20a] Alexander Tornede, Marcel Wever, and Eyke Hüllermeier. „Extreme Algorithm Selection with Dyadic Feature Representation“. In: *Discovery Science - 23rd International Conference, DS 2020, Thessaloniki, Greece, October 19-21, 2020, Proceedings*. Vol. 12323. Lecture Notes in Computer Science. Springer, 2020, pp. 309–324. DOI: 10.1007/978-3-030-61527-7_21. URL: https://doi.org/10.1007/978-3-030-61527-7%5C_21 (cit. on pp. 8, 22).

[Tor+20a] Alexander Tornede, Marcel Wever, Stefan Werner, Felix Mohr, and Eyke Hüllermeier. „Run2Survive: A Decision-theoretic Approach to Algorithm Selection based on Survival Analysis“. In: *Proceedings of The 12th Asian Conference on Machine Learning, ACML 2020, 18-20 November 2020, Bangkok, Thailand*. Vol. 129. Proceedings of Machine Learning Research. PMLR, 2020, pp. 737–752. URL: http://proceedings.mlr.press/v129/tornede20a.html.

[Wev+20] Marcel Wever, Alexander Tornede, Felix Mohr, and Eyke Hüllermeier. „LiBRe: Label-Wise Selection of Base Learners in Binary Relevance for Multi-label Classification“. In: *Advances in Intelligent Data Analysis XVIII - 18th International Symposium on Intelligent Data Analysis, IDA 2020, Konstanz, Germany, April 27-29, 2020, Proceedings*. Vol. 12080. Lecture Notes in Computer Science. **Frontier Prize**. Springer, 2020, pp. 561–573. DOI: 10.1007/978-3-030-44584-3_44 (cit. on pp. 1, 3).

[Han+21] Jonas Hanselle, Alexander Tornede, Marcel Wever, and Eyke Hüllermeier. „Algorithm Selection as Superset Learning: Constructing Algorithm Selectors from Imprecise Performance Data“. In: *Advances in Knowledge Discovery and Data Mining - 25th Pacific-Asia Conference, PAKDD 2021, Virtual Event, May 11-14, 2021, Proceedings, Part I*. Vol. 12712. Lecture Notes in Computer Science. Springer, 2021, pp. 152–163. DOI: 10.1007/978-3-030-75762-5_13.

[Tor+21] Tanja Tornede, Alexander Tornede, Marcel Wever, and Eyke Hüllermeier. „Coevolution of Remaining Useful Lifetime Estimation Pipelines for Automated Predictive Maintenance“. In: *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2021, Lille, France, July 10-14, 2021*. ACM, 2021 (cit. on pp. 18, 156).

Workshop Papers

[WMH17] Marcel Wever, Felix Mohr, and Eyke Hüllermeier. „Automatic Machine Learning: Hierarchical Planning Versus Evolutionary Optimization“.

In: *Proceedings of the 27. Workshop Computational Intelligence*. **Young Author Award (Runner Up)**. 2017, pp. 149–166 (cit. on p. 19).

[Moh+18a] Felix Mohr, Theodor Lettmann, Eyke Hüllermeier, and Marcel Wever. „Programmatic task network planning“. In: *Proceedings of the 1st ICAPS Workshop on Hierarchical Planning*. 2018, pp. 31–39 (cit. on p. 19).

[WMH18c] Marcel Dominik Wever, Felix Mohr, and Eyke Hüllermeier. „ML-Plan for unlimited-length machine learning pipelines“. In: *ICML 2018 AutoML Workshop*. 2018. URL: https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxhdXRvbWwyMDE4aWNtbHxneDo3M2Q3MjUzYjViNDRhZTAx (cit. on pp. 3, 19).

[TWH19] Alexander Tornede, Marcel Wever, and Eyke Hüllermeier. „Algorithm Selection as Recommendation: From Collaborative Filtering to Dyad Ranking“. In: *CI Workshop, Dortmund*. **Young Author Award**. 2019.

[Wev+19] Marcel Dominik Wever, Felix Mohr, Alexander Tornede, and Eyke Hüllermeier. „Automating multi-label classification extending ml-plan“. In: *ICML 2019 AutoML Workshop*. 2019. URL: https://www.automl.org/wp-content/uploads/2019/06/automlws2019_Paper46.pdf (cit. on pp. 3, 20, 34).

[TWH20b] Alexander Tornede, Marcel Wever, and Eyke Hüllermeier. „Towards Meta-Algorithm Selection“. In: *NeurIPS 2020 Meta-Learning Workshop*. 2020. URL: https://meta-learn.github.io/2020/papers/37_paper.pdf.

[Tor+20b] Tanja Tornede, Alexander Tornede, Marcel Wever, Felix Mohr, and Eyke Hüllermeier. „AutoML for Predictive Maintenance: One Tool to RUL Them All“. In: *IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning - Second International Workshop, IoT Streams 2020, and First International Workshop, ITEM 2020, Co-located with ECML/PKDD 2020, Ghent, Belgium, September 14-18, 2020, Revised Selected Papers*. Vol. 1325. Communications in Computer and Information Science. Springer, 2020, pp. 106–118. DOI: 10.1007/978-3-030-66770-2_8 (cit. on p. 20).

Preprints

[MWH18a] Felix Mohr, Marcel Wever, and Eyke Hüllermeier. „Automated Machine Learning Service Composition“. In: *CoRR* abs/1809.00486 (2018). arXiv: 1809.00486. URL: http://arxiv.org/abs/1809.00486 (cit. on p. 161).

[WMH18a] Marcel Wever, Felix Mohr, and Eyke Hüllermeier. „Automated Multi-Label Classification based on ML-Plan“. In: *CoRR* abs/1811.04060 (2018). arXiv: 1811.04060. URL: http://arxiv.org/abs/1811.04060 (cit. on p. 20).

[HWH20] Stefan Heid, Marcel Wever, and Eyke Hüllermeier. „Reliable Part-of-Speech Tagging of Historical Corpora through Set-Valued Prediction“. In: *CoRR* abs/2008.01377 (2020). arXiv: 2008.01377. URL: https://arxiv.org/abs/2008.01377.

[Mer+21] Marie-Luis Merten, Marcel Wever, Michaela Geierhos, Doris Tophinke, and Eyke Hüllermeier. „Annotation Uncertainty in the Context of Grammatical Change“. In: *CoRR* abs/2105.07270 (2021). arXiv: 2105.07270. URL: https://arxiv.org/abs/2105.07270.

[MW21] Felix Mohr and Marcel Wever. *Naive Automated Machine Learning - A Late Baseline for AutoML*. 2021. arXiv: 2103.10496. URL: https://arxiv.org/abs/2103.10496 (cit. on p. 156).

List of Figures

Colophon

This thesis was typeset with LaTeX 2_ε. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.