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
© The Author(s) 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.

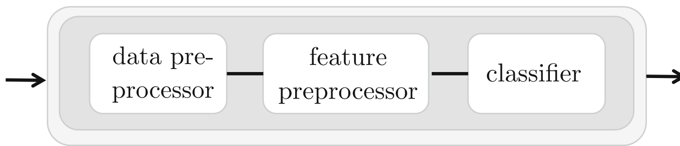

Fig. 1 A classical AutoML pipeline of fixed length

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 $D_{val} = D \setminus D_{train}$. 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

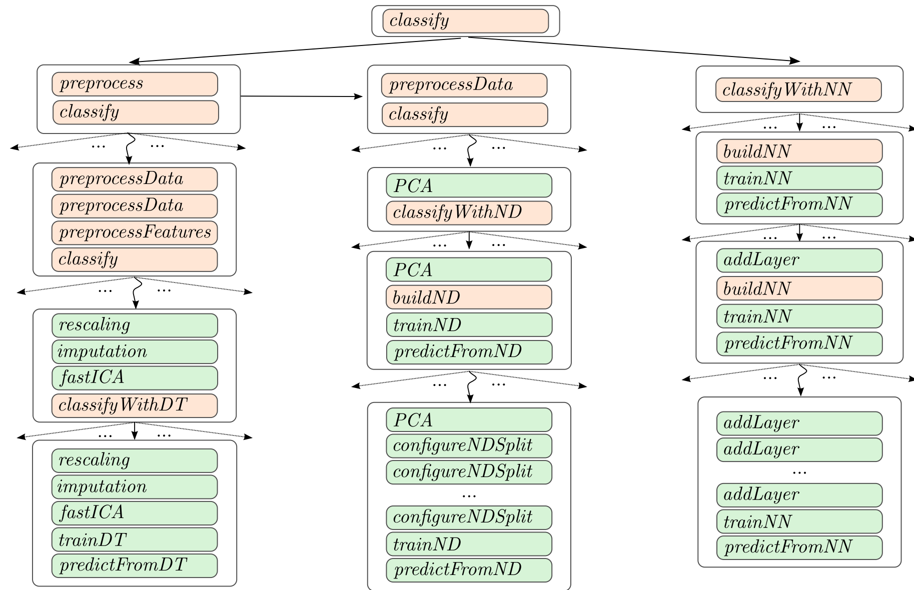

Fig. 2 Creation of pipelines with hierarchical planning. Left: a pipeline that uses a decision tree for prediction where data are preprocessed with rescaling and imputation and features are preprocessed using fast ICA. Middle: a pipeline that uses a (configured) nested dichotomy for prediction where 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 $(s \cup add) \setminus del$ 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 • indicates that ML-Plan is significantly better, and analogously a ◦ 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

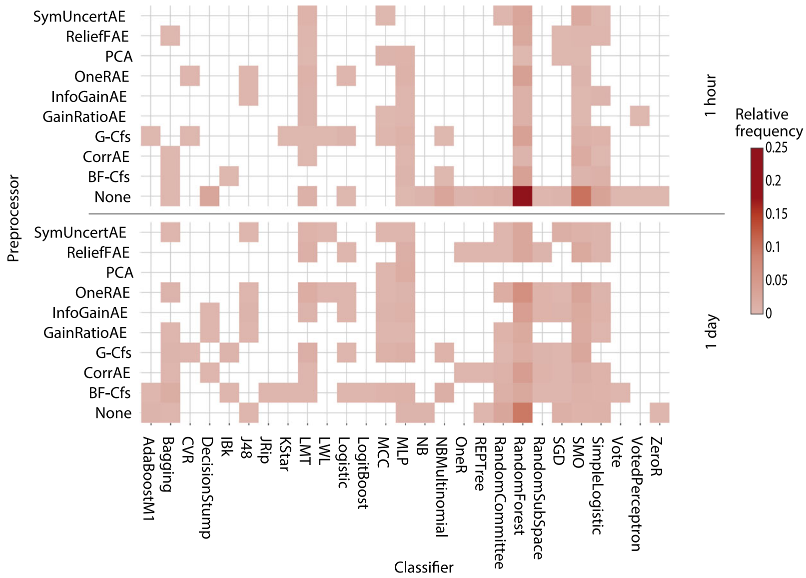

Fig. 3 Relative frequency of WEKA preprocessor-classifier combinations as returned by ML-Plan after 1 h resp. 1 day

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.

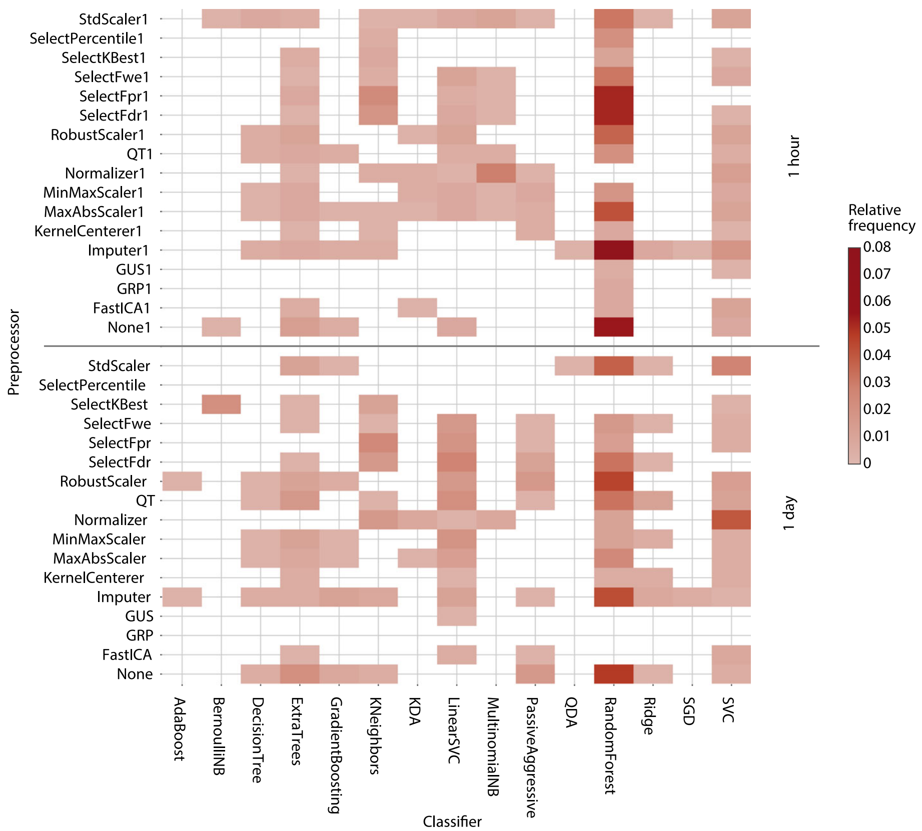

Fig. 4 Relative frequency of scikit-learn preprocessor-classifier combinations as returned by ML-Plan after 1 h resp. 1 day

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.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.