Automatic Machine Learning: Hierarchical Planning Versus Evolutionary Optimization

Marcel Wever[1], Felix Mohr[2], Eyke Hüllermeier[3]

Paderborn University
Warburger Str. 100, 33098 Paderborn

[1]E-Mail: marcel.wever@uni-paderborn.de
[2]E-Mail: fmohr@uni-paderborn.de
[3]E-Mail: eyke@upb.de

Abstract

These days, there is a growing need for machine learning applications, coming with the quest to automate parts of the process of engineering machine learning tools and algorithms. This development has triggered the emergence of automated machine learning (AutoML) as a new subfield of machine learning. In AutoML, the selection, composition and parametrization of machine learning algorithms is automated and tailored to a specific problem, resulting in a machine learning pipeline. Current approaches reduce the AutoML problem to optimization of hyperparameters. Based on recursive task networks, in this paper we present one approach from the field of automated planning and one evolutionary optimization approach. Instead of simply parametrizing a given pipeline, this also includes the structure optimization of machine learning pipelines. We evaluate the two approaches in an extensive evaluation, finding both of them to have their strengths in different areas. Moreover, the two approaches outperform the state-of-the-art tool Auto-WEKA in many settings.

1 Introduction

While the demand for machine learning functionality is growing quite rapidly these days, end users in application domains are normally not machine learning experts. Therefore, 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., could be automated to a large extent [7]. 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. Given a specific problem (data set), the goal of AutoML is to automatically set up a suitable machine learning pipeline, comprising the aforementioned steps of model induction.

In spite of quite impressive first results, state-of-the-art tools such as Auto-WEKA [13] and Auto-sklearn [4] are still limited in scope and restricted to rather simple learning problems such as classification, essentially because automated machine learning is reduced to the optimization of hyperparameters. In this paper, we address the additional problem of optimizing the structure of a machine learning pipeline, instead of simply parametrizing a given one. We consider structure optimization as an important prerequisite for the application of automated machine learning to learning problems more complex than standard (binary) classification or regression, such as multi-target prediction or structured-output prediction. More specifically, we compare two approaches for optimizing machine learning pipelines, which are based on two different principles for searching the space of configurations.

Our first approach is based on the idea to consider the machine learning problem as a planning task. This idea is arguably quite natural: what the machine learning expert has to do is to devise a plan determining which data processing steps are to be executed in which order, and how the different steps shall be configured or parametrized. More specifically, we make use of recursive task networks for hierarchical planning [9], which offers a versatile approach for the configuration of machine learning pipelines.

As an alternative to the systematic search strategy realized by hierarchical planning, we make use of an evolutionary approach that is based on recursive task networks as well. To this end, the evolutionary algorithm maintains a population of individuals which represent a single path in the network and thus a machine learning pipeline including hyperparameters being set. We apply multi-objective optimization to assess the fitness of ML pipelines considering the generalization behavior of a pipeline in different stages of the learning process, i.e. for different proportions of training-validation splits, in order to prevent the optimized machine learning pipelines from overfitting the provided data.

2 An HTN-Based Search Graph

2.1 Hierarchical Task Networks

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 concrete decision tree object 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 Figure 1 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 then a tuple $\langle s_0, T_0, O, M \rangle$, where O and M are finite sets of operators and methods, respectively.

The HTN problem definition induces a search graph (a tree) that can be searched with standard search algorithms such as depth first search,

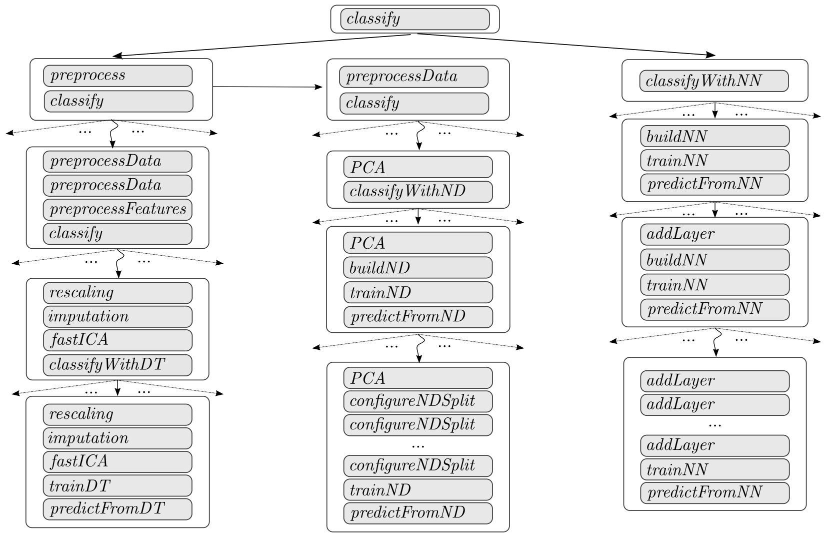

Figure 1: Task networks allow not only more flexible pipelines but even to configure its elements.

best first search, etc.. The graph in Figure 1 sketches (an exerpt of) such a search graph for the AutoML problem. Every node corresponds to a partially defined pipeline (complex tasks are partially defined aspects that still require refinement). 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-one correspondence between search space elements, e.g., the machine learning pipelines, and the goal nodes.

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 technique is called forward-decomposition. While it is also possible to derive graphs with other structures for the same HTN problem, we adopt forward-decomposition in this paper.

2.2 The AutoWEKA-Simulation Search Graph

The search graph is induced by the description of a task network problem. As described above, we adopt forward-decomposition, i.e., a node has successors iff it has at least one complex task, and there is one successor for each method or operator that can be used to refine the *first* task in the network. The complete problem description is very technical, so we focus on giving an intuition. The full formal specification is available with our implementation[2].

The root node is a task network consisting of three tasks `createRawPP`, `setupClassifier`, `refinePP`. The first task can be refined to the existing preprocessing algorithms *without* parametrizing them. There is one method for each classifier to refine the second task `setupClassifier`. Each of these methods refines the task to a network of the form `setupParam1`, ..., `setupParamN` for all of the N parameters of the respective classification algorithm, so the network enforces that a decision is made for each of the parameters. 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 same technique is then applied to refine `refinePP` and thereby to configure the initially chosen preprocessor (if any). We support the same preprocessors, classifiers, and parameters as used in AutoWEKA.

This modeling technique induces a tree with two regions, which are separated in a relatively low depth of the search tree, say d. This is because the whole construction process encodes the idea of (i) selecting the preprocessor, (ii) selecting the classifier, (iii) parametrizing the classifier, (iv) parametrizing the preprocessor. Partial solutions of nodes in the shallower region (up to a depth of d) define the algorithms that will be used but have not yet made any decision about parametrization, i.e., the steps (i) and (ii). All nodes in depth of at least d deal with the parametrization, i.e. correspond to decisions either in phase (iii) or (iv).

Numerical parameters are discretized either on a linear scale or a log scale. The discretization technique for a parameter is not a choice point but is fixed in advance.

2.3 Potential of HTN Planning for AutoML

Given one concrete example of how to create ML pipelines using HTN planning, we stress that we are not committed to one particular HTN problem definition. In fact, there are many different HTN problems that can cover exactly the same search space. So apart from any questions

[2] Attached as supplementary material during review phase.

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. Just to give an example, we could 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. Yet, we may also interleave these configurations and first choose the preprocessor and the classifier, and *then* configure both of them. While this looks like a trivial change that does not affect the set of constructible pipelines, it has important consequences on the structure of the search tree.

The dominating expressiveness of HTN techniques for AutoML is not only reflected in the ability to construct pipelines of arbitrary lengths but also in that they can *configure complex elements* within it. For example, the right branch of Figure 1 shows that we can configure a neural network. Note, once again, that even though the figure does not show any parameters, we cannot only control the number of layers but also their connections.

Another important advantage of HTN for AutoML is its ability to encode *reduction*. Reduction or decomposition techniques such as one-versus-rest [11], all-pairs [5], or error correcting output codes [3] are quite popular in machine learning. For example, instead of solving the multi-class classification problem directly on the set of k classes, we can first separate two *sets* of classes from each other. This induces two reduced classification problems, which can be solved either directly or again by recursing in the same way (unless there are only two classes left), and this is a very natural use case of HTN planning.

3 ML-Plan: AutoML through HTN Planning

ML-Plan directly solves the HTN problem defined above using an HTN planner. For a specific problem type, e.g., classification, the HTN problem that needs to be solved is usually fixed. The variety for different queries arises from the fact that a plan (composition) performs differently on different data sets. The data set is then used to *guide* the search.

We adopt a best first search algorithm in order to identify good pipelines. A best first search algorithm assigns a number to each node and always chooses the node with the currently lowest known value for expansion.

Since the prediction error as the solution quality does not decompose over the path (necessary for A*), we adopt a randomized depth first search similar to the one applied in Monte Carlo Tree Search 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 goal node is reached. We then compute the solution “qualities” of each of n such random completions and take the minimum as an estimate for the best possible solution that can be found under that node.

The qualities of the completed solutions are determined by computing a k-step Monte Carlo Cross Validation. That is, a fixed portion of the data initially provided to the search algorithm is allocated for node evaluation; in our implementation we used 70% of the data. To evaluate a single solution, this portion is then partitioned k-times into a stratified training and validation set; here, we also chose a split of 70% for training and 30% for validation and $k = 5$. For each of the k splits, the solution pipeline is trained with the respective training set and tested against the validation set. The mean 0/1-loss of this evaluation is the score of that solution.

Since the node evaluation function actually 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, we can always return solutions even if the main search routine did not already discover any goal node.

In order to make this strategy more reliable, in the upper region of the search graph, we use a biased breadth first search instead. This is to avoid that the randomized depth search averages over too many heavily distinct solutions. The bias within a layer is towards the nodes corresponding to frequently well-performing algorithms: KNN, random forests, voted perceptron, SVM, logistic regression (in this order). This arbitrary preference is hard-coded, but we plan to make it data-dependent in a follow-up version.

Since evaluating the solution candidates is very costly, we use a reduced version of the originally given dataset. For a number n of classes, we reduce the number of examples to at most $250 \times n$ (stratified removal) and the number of features to at most $5 \times n$ using principal component analysis. Of course, this reduction only needs to be computed once in a preprocessing step of the overall search process. In [10], we do not make this simplification.

4 EvoML-Plan: Genetic HTN Planning

As an alternative to the systematic search strategy realized by hierarchical planning, we make use of a so-called messy genetic algorithm [6]. In contrast to classical genetic approaches that take a fixed length of the genotype for granted, messy genetic algorithms allow for variable length strings of genes. The variable length of genestrings is crucial for solving HTN planning problems as the length of plans may vary substantially.

4.1 Genetic Representation

Each individual represents one possible plan derived from the hierarchical task network (HTN). In order to derive plans from the HTN, complex tasks need to be refined to primitive tasks until finally a concrete plan is obtained. As for a refinement different choices might be eligible, we can define our genetic representation to encode these choices. Since the length of concrete plans may have an arbitrary length and moreover may vary for different plans, the number of choices to be made are varying accordingly. Moreover, the genetic representation must not be fixed to a certain length since otherwise it could not represent each possible solution.

Therefore, we choose the genetic representation to be a list of non-negative integers. For each refinement, the possible refinement options are listed and the next gene is taken as the index in this list. If the value of this gene exceeds the number of possible options, genes will get skipped until we finally reach a gene in this range. Note that the genetic representation does not take any semantics into account. Therefore, the i-th gene may represent for instance the choice of a parameter value or a particular algorithm.

Due to the variable length of chromosomes, an individual might be over- or underspecified. While we can deal with over-specification by simply ignoring the remaining part of the chromosome, the other scenario requires additional genetic material to be added dynamically. To this end, we extend the chromosome by randomly drawing new genes until the plan is entirely specified.

Another problem using this genetic representation arises from the application of genetic operators such as mutation and crossover. As a small change in a single gene may lead to dramatic changes since all the choices made subsequently lose their previous semantics, we divide the overall genotype into so-called *codons*. A codon represents a cohesive sequence of genes that is inherited by offspring en bloc. More specifically, for each complex task, we use a distinct codon of variable length.

Considering our scenario of configuring ML pipelines, the chromosome is divided into codons describing the feature preprocessor, the classifier, the classifier's parametrization, and the parameters chosen for the feature preprocessing. An example is illustrated in Figure 2. Due to this segmentation, changing a gene for the choice of the feature preprocessor does no longer influence the decisions for the classifier and its parametrization at all. Still the codon for the parametrization of the feature preprocessor adopt its semantics according to the feature preprocessor.

The segmentation into codons allows a variable length of the chromosome while isolating semantic structures in order to facilitate the exchange of partial solutions without changing other components. In particular, with this technique it is possible to swap the feature preprocessors of two individuals only.

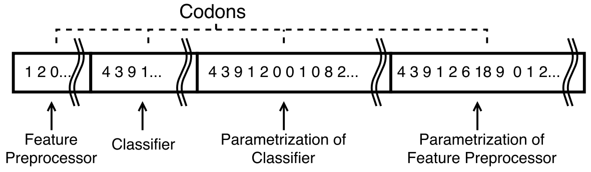

Figure 2: Genotype for the representation of ML pipelines

4.2 Genetic Operators

For the variation of individuals, on one hand, we use standard mutation for single genes. On the other hand, we use an instantiation of n-point crossover fitted to our genetic representation. Meaning, the crossover takes the semantics of the ML pipelines into account, exchanging only entire building blocks of the pipeline. An example for the genetic operator recombining two individuals is presented in Figure 3. In the example figure the crossover is a two-point crossover. Generally, the crossover operator is not fixed to two intersection points. Depending on the order of the codons and depending on how many building blocks a ML pipeline may involve, the crossover can be adapted to exchange the entire feature

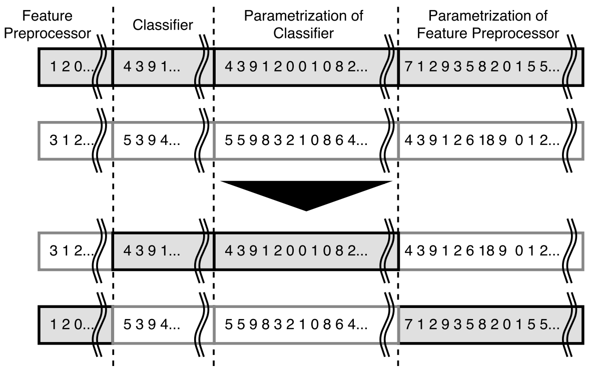

Figure 3: Crossover of two ML pipeline individuals

preprocessing or one to multiple codons. The only important criterion for the crossover is to exchange codons for the selection of an algorithm and its parametrization together.

4.3 Multi-Objective Optimization

Optimizing for the best or most accurate ML pipeline comes also with a high danger for overfitting the data. Therefore, we choose a multi-objective optimization approach in order to have a more differentiated look at the generalization behavior of each evaluated ML pipeline. To this end, we use repeated random sub-sampling validation, aka. Monte Carlo cross-validation, creating repeatedly stratified splits at random [1, 12]. Since the data contained in the training and validation set varies with each split and evaluation, we prevent the ML pipelines from getting to specialized for a certain partitioning.

More specifically, we evaluate each individual performing 5-fold Monte Carlo cross-validation for 50:50, 67:33, and 80:20 splits of training and validation data. Taking the average error rate for the different split proportions as an individual fitness function, we use these as objectives in the multi-objective evolutionary algorithm NSGA-II (see [2]) in general. However, the evaluation of an individual is rather costly, and an individual's evaluation might take several minutes. Due to this and since the main reason for multiple objectives is to prevent the individuals from overfitting, we only use the error rate for the 80:20 splits as a fitness function if a timeout of less the 5 minutes is given.

Selection is done via tournament selection involving 2 individuals, where these two are compared primarily whether the fitness values of one individual dominate the other one's fitness values. If this is not the case for one of the two individuals, a crowding distance comparator is taken into account. The crowding distance comparator calculates how close other individuals are to the considered individuals, and it prefers individuals that are more different from the remaining population.

In order to support diversification of the population, we reinitialize the population every 5 generations with random individuals, keeping only the elite of the current population.

4.4 Solution Selection

The result of NSGA-II is represented by a Pareto set of non-dominated individuals. Therefore, we still need to select a single individual as the final result of the genetic algorithm. As usual in multi-objective optimization, the selection of a particular candidate from a Pareto set is a non-trivial decision. We proposed three fitness functions evaluating an ML pipeline on different splits of the given dataset. Interpreting the three fitness values as a vector in the three-dimensional space, we choose the solution which is closest to the optimum, i.e. the solution with the smallest distance to the origin.

Moreover, this rather simple selection method allows us to always maintain the best solution seen so far. Hence, in the face of timeouts the algorithm is always able to immediately return a solution even if the algorithm is currently processing a generation. However, as a prerequisite for returning a solution, a valid individual, i.e., an ML pipeline that can be applied to the particular problem, must have been already evaluated.

5 Evaluation

We evaluate the two approaches on a selection of 21 datasets from the UCI repository. In our evaluation we compare the two proposed approaches EvoML-Plan and ML-Plan to each other and additionally to the state-of-the-art tool Auto-WEKA.

5.1 Experimental Setup

For the comparison of the three approaches, we carried out 25 runs for each of the approaches on the 21 datasets for timeouts of one minute and one hour, yielding a total number of 3,150 experiments. The timeout for the evaluation of a single individual in EvoML-Plan resp. the internal evaluation of a single solution in ML-Plan was set to 10s for a timeout of one minute. In the case of a timeout of one hour, the internal evaluation timeout was set to 5 minutes. For all the runs, 70% of a stratified split of the entire data were provided to the algorithms as training data and 30% were used for testing. The computations were executed on 200 Linux machines in parallel with 8 cores (Intel Xeon E5-2670, 2.6GHz) and 32GB memory each.

Runs exceeding the timeout limit or the resource limitations of the nodes were canceled and their results are disregarded in the following discussion. Nevertheless, the algorithms were admitted an extra amount of time so that they were killed after taking 110% of the set timeout. Moreover, the algorithms were killed when consuming more resources (CPU and memory) than allocated, which happens because in the implementation controlling the CPU and memory consumption of forked subprocesses is rather hard.

For significance testing, we use the Mann-Whitney-U Test [8], and we denote significant improvements respectively degradation if $p < 0.05$.

5.2 Results

In Tables 1 and 2 the results of the runs with a timeout of one minute respectively one hour are presented. In the tables, for each dataset and each approach, the number of returned solutions (n) and the average test set loss (0/1-loss) plus or minus the standard deviation is shown. Furthermore, for each row the best performing approach is highlighted with bold letters and non-significant degradations are highlighted underlining these values.

The results in Table 1 show that EvoML-Plan as well as ML-Plan clearly outperform Auto-WEKA for all the evaluated datasets. As Auto-WEKA does not even return a solution in the given timeout for some datasets, we can conclude that both proposed approaches find solutions at least faster than Auto-WEKA does. Furthermore, in settings where Auto-WEKA returned solutions, our approaches return solutions that are significantly

Table 1: Experimental results (mean 0/1-losses±std) for a timeout of 1 minute

Dataset	n	EvoML-Plan	n	ML-Plan	n	Auto-WEKA
abalone	23	73.94 ± 0.59	19	**73.33** ± 0.60	12	74.94 ± 1.25
amazon	14	**90.41** ± 5.28	0	?	0	?
car	24	5.05 ± 1.56	19	**3.40** ± 0.74	12	7.18 ± 1.00
cifar10	4	86.34 ± 6.33	16	**68.82** ± 6.15	0	?
cifar10small	16	84.81 ± 7.89	20	**69.98** ± 6.71	0	?
convex	12	47.65 ± 0.34	18	**27.71** ± 0.19	0	?
dexter	20	**24.88** ± 9.90	0	?	0	?
dorothea	17	**26.81** ± 10.32	0	?	0	?
germancredit	24	27.52 ± 1.46	20	**24.83** ± 0.83	12	28.64 ± 0.72
gisette	13	35.65 ± 15.93	17	**3.65** ± 0.15	0	?
kddcup09appe	6	**1.77** ± 0.00	3	2.47 ± 0.92	0	?
krvskp	25	**1.15** ± 0.44	20	5.09 ± 1.76	12	1.60 ± 1.64
madelon	25	**27.74** ± 2.47	18	39.83 ± 2.12	12	30.35 ± 2.49
mnist	6	35.88 ± 11.67	20	**6.50** ± 0.23	0	?
mnistrotatio	11	88.76 ± 0.00	9	**74.88** ± 2.93	0	?
secom	25	**6.42** ± 0.00	19	6.44 ± 0.06	12	6.57 ± 0.27
semeion	24	12.69 ± 2.53	20	**10.49** ± 0.86	12	14.0 ± 0.79
shuttle	24	0.11 ± 0.06	15	**0.02** ± 0.01	12	0.14 ± 0.01
waveform	25	13.92 ± 1.13	16	**13.71** ± 0.27	12	15.14 ± 1.54
winequality	25	38.86 ± 2.89	20	**32.43** ± 0.60	12	36.56 ± 0.30
yeast	22	40.29 ± 1.31	19	**40.14** ± 1.32	12	42.17 ± 1.77

better than Auto-WEKA's solutions in 7 out of 11 cases. Comparing EvoML-Plan and ML-Plan, we can observe that best performances alternate for the different datasets. While EvoML-Plan leads to significantly better performing results in 5 datasets and ML-Plan in 7 datasets, for the remaining the performances are competitive to each other.

After a timeout of one hour (see Table 2), the two proposed approaches still perform better than Auto-WEKA for many datasets. However, the clear dominance is diminishing but still Auto-WEKA yields better results only in 5 out of 21 cases, where a significant improvement over both EvoML-Plan and ML-Plan is achieved only once. Considering the performance of our two approaches, we notice alternating significant improvements. While ML-Plan yields superior results in 8 cases, EvoML-Plan returns significantly better solutions for 6 datasets. The remaining 7 datasets indicate competitiveness of the two approaches.

Table 2: Experimental results (mean 0/1-losses±std) for a timeout of 1 hour

Dataset	n	EvoML-Plan	n	ML-Plan	n	Auto-WEKA
abalone	22	**72.93** ± 0.61	10	73.12 ± 0.13	25	73.49 ± 0.76
amazon	23	51.31 ± 10.84	18	**31.15** ± 1.32	24	51.44 ± 1.61
car	23	2.59 ± 1.12	4	0.83 ± 0.38	24	**0.65** ± 0.23
cifar10	20	77.04 ± 7.03	2	**60.73** ± 1.80	0	?
cifar10small	21	72.19 ± 6.92	4	**60.11** ± 0.77	1	70.23 ± 0.00
convex	24	44.28 ± 5.10	12	**27.7** ± 0.22	25	46.86 ± 0.25
dexter	23	**9.81** ± 2.43	4	19.1 ± 2.32	11	11.24 ± 0.51
dorothea	24	**9.58** ± 2.54	0	?	0	?
germancredit	22	25.5 ± 1.09	9	**24.94** ± 0.42	25	26.81 ± 1.04
gisette	23	4.21 ± 1.49	5	5.00 ± 1.84	23	**3.93** ± 0.29
kddcup09appe	19	1.77 ± 0.00	3	1.78 ± 0.01	17	**1.77** ± 0.00
krvskp	25	**0.73** ± 0.24	10	1.73 ± 0.44	24	2.27 ± 2.28
madelon	25	26.65 ± 2.99	9	39.89 ± 0.44	24	**26.11** ± 2.31
mnist	17	12.59 ± 1.40	5	**7.21** ± 1.70	25	7.21 ± 0.12
mnistrotatio	22	73.27 ± 6.65	3	**62.95** ± 0.38	25	78.61 ± 0.27
secom	25	**6.42** ± 0.00	8	6.53 ± 0.11	25	6.52 ± 0.25
semeion	19	**7.44** ± 0.87	13	10.0 ± 0.48	18	13.01 ± 2.02
shuttle	24	0.06 ± 0.05	5	**0.05** ± 0.05	25	0.12 ± 0.04
waveform	24	**13.24** ± 0.57	11	14.51 ± 0.50	25	13.26 ± 0.40
winequality	22	35.66 ± 2.27	15	**32.89** ± 0.69	22	33.34 ± 1.11
yeast	21	40.26 ± 1.59	11	40.23 ± 0.68	23	**39.87** ± 1.36

To sum up, we notice that the two approaches already lead to promising results comparing to the state-of-the-art Auto-WEKA. Especially, in a timeout of one minute, it becomes clear that the two proposed approaches perform faster. Even for a timeout of one hour, EvoML-Plan] and ML-Plan continue to perform better than Auto-WEKA. For both timeouts and all the datasets, we observe competitiveness for the two approaches and find that both have their individual strengths, albeit we notice ML-Plan to have a slight edge over EvoML-Plan.

6 Conclusion and Future Work

We proposed two new approaches to the AutoML problem, both being based on hierarchical planning. While on one hand, we made use of a classical planning approach, as an alternative search strategy, we applied a multi-objective evolutionary algorithm to the same problem. In our evaluation, we found significant performance improvements over the state-of-the-art. Moreover, we have seen that both strategies have their advantages becoming evident in better performance compared to the other approach.

In our evaluation, we limited the configured machine learning pipelines to a length of two, i.e. incorporating a feature preprocessor and a classifier, in order to remain comparable to Auto-WEKA. However, by this limitation we did not even leverage the full potential of HTN, and thus, exploiting the latter is an important point for future work. Moreover, there were datasets where worse results have been obtained in the run with a timeout of one hour compared to the ones returned after one minute. This indicates that the solutions returned after one hour tend to overfit the data, requiring a mechanism to deal with this problem.

Acknowledgment

This work is part of the Collaborative Research Centre “On-the-Fly Computing” at Paderborn University, which is supported by the German Research Foundation (DFG).

References

[1] P. Burman. A comparative study of ordinary cross-validation, v-fold cross-validation and the repeated learning-testing methods. *Biometrika*, 76(3), 1989.

[2] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evolutionary Computation*, 6(2):182–197, 2002.

[3] T. Dietterich and G. Bakiri. Solving multiclass learning problems via error-correcting output codes. *Journal of Artificial Intelligence Research*, 2:263–286, 1995.

[4] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter. Efficient and robust automated machine learning. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2962–2970. Curran Associates, Inc., 2015.

[5] J. Fürnkranz. Round robin classification. *Journal of Machine Learning Research*, 2:721–747, 2002.

[6] D. E. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3(5), 1989.

[7] J. R. Lloyd, D. K. Duvenaud, R. B. Grosse, J. B. Tenenbaum, and Z. Ghahramani. Automatic construction and natural-language description of nonparametric regression models. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 1242–1250, 2014.

[8] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Statist.*, 18(1):50–60, 03 1947.

[9] F. Mohr, T. Lettmann, and E. Hüllermeier. ITN planning: Planning with independent task networks. In *Proceedings KI-2017, 40th German Conference on Artificial Intelligence, Dortmund, Germany*, 2017.

[10] F. Mohr, M. Wever, and E. Hüllermeier. ML-Plan: Automated machine learning via hierarchical planning. Submitted for publication, 2018.

[11] R. Rifkin and A. Klautau. In defense of one-vs-all classification. *Journal of Machine Learning Research*, 5:101–141, 2004.

[12] J. Shao. Linear model selection by cross-validation. *Journal of the American Statistical Association*, 88(422):486–494, 1993.

[13] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Auto-weka: Automated selection and hyper-parameter optimization of classification algorithms. *CoRR*, abs/1208.3719, 2012.