Machine Learning (2023) 112:1131–1170
https://doi.org/10.1007/s10994-022-06200-0

Naive automated machine learning

Felix Mohr[1] · **Marcel Wever**[2]

Received: 3 March 2021 / Revised: 12 May 2022 / Accepted: 26 May 2022 /
Published online: 29 September 2022

Abstract
An essential task of automated machine learning (AutoML) is the problem of automatically finding the pipeline with the best generalization performance on a given dataset. This problem has been addressed with sophisticated black-box optimization techniques such as Bayesian optimization, grammar-based genetic algorithms, and tree search algorithms. Most of the current approaches are motivated by the assumption that optimizing the components of a pipeline in isolation may yield sub-optimal results. We present Naive AutoML, an approach that precisely realizes such an in-isolation optimization of the different components of a pre-defined pipeline scheme. The returned pipeline is obtained by just taking the best algorithm of each slot. The isolated optimization leads to substantially reduced search spaces, and, surprisingly, this approach yields comparable and sometimes even better performance than current state-of-the-art optimizers.

Keywords Automated Machine Learning · Data Science · Black-Box Optimization

1 Introduction

An important task in Automated machine learning (AutoML) is the one of automatically finding the pre-processing and learning algorithms with the best generalization performance on a given dataset. The combination of such algorithms is typically called a (machine learning) *pipeline* (Feurer et al., 2015) because several algorithms for data manipulation and analysis are put into (partial) order. The choices to be made in pipeline optimization include the algorithms used for feature pre-processing and learning as well as the hyperparameters of the chosen algorithms.

Editors: Annalisa Appice, Grigorios Tsoumakas.

✉ Felix Mohr
felix.mohr@unisabana.edu.co

Marcel Wever
marcel.wever@uni-paderborn.de

[1] Universidad de La Sabana, Chia, Colombia

[2] Paderborn University, Paderborn, Germany

Maybe surprisingly, all common approaches to this problem try to optimize over all decision variables *simultaneously* (Thornton et al., 2013; Feurer et al., 2015; Olson and Moore , 2019; Mohr et al., 2018; Yang et al., 2019), and, to our knowledge, it has never been tried to optimize the different components in isolation. While one might intuitively expect significant interactions between the optimization decisions, one can argue that achieving a *global* optimum by *local* optimization of components could be at least considered a relevant baseline to compare against.

We present two approaches for pipeline optimization that do exactly this: They optimize a pipeline locally instead of globally. The most extreme approach, Naive AutoML, assumes that a locally optimal decision is also globally optimal, i.e., the optimality of a local decision is *independent* of how other components are chosen. In practice, this means that all components that are not subject to a local optimization process are left blank, except the learner slot, e.g., classifier or regressor, which is configured with some arbitrary default algorithm, e.g., kNN, in order to obtain a valid pipeline. Since Naive AutoML might sometimes be *too* naive, we consider a marginally less extreme optimizer, called Quasi-Naive AutoML. Quasi-Naive AutoML defines an *order* in which components are considered and optimizes each slot based on the previous decisions; it is only naive with respect to upcoming decisions.

On top of naivety, both Naive AutoML and Quasi-Naive AutoML assume that hyperparameter optimization is irrelevant for choosing the best algorithm for each slot. That is, they assume that the best algorithm under default parametrization is also the best among all tuned algorithms. Therefore, both Naive AutoML and Quasi-Naive AutoML optimize a slot by first selecting an algorithm and then optimize the hyperparameters of each chosen algorithm in a second phase.

Our experimental evaluation shows that these simple techniques are surprisingly strong compared to state-of-the-art optimizers. While Naive AutoML is outperformed in the long run (24 h), it is competitive with state-of-the art approaches in the short run (1 h runtime). On the contrary, Quasi-Naive AutoML even outperforms the state-of-the-art techniques in the short run and is often competitive in the long run (24h) by achieving a close-to-optimal performance in the majority of the cases.

While these results might suggest Quasi-Naive AutoML as a meaningful baseline over which one should be able to substantially improve, we see the actual role of Quasi-Naive AutoML as the door opener for sequential optimization of pipelines. The currently applied black-box optimizers come with a series of problems discussed in recent literature such as lack of flexibility (Drozdal et al., 2020; Crisan and Fiore-Gartland , 2021). The naive approaches follow a sequential optimization approach, optimizing one component after the other. While flexibility is not a topic in this paper, it can be arguably realized more easily in custom sequential optimization approaches than in black-box optimization approaches. The strong results of Quasi-Naive AutoML seem like a promise that extensions of Quasi-Naive AutoML such as Mohr and Wever (2021) could overcome the above problems of black-box optimizers *without sacrificing* global optimality. We discuss this in more depth in Sect. 5.3.

2 Problem definition

Even though the vision of AutoML is much broader, a core task of AutoML addressed by most AutoML contributions is to automatically *compose* and *parametrize* machine learning algorithms to optimize a given metric such as accuracy.

In this paper, we focus on AutoML for supervised learning. Formally, in the supervised learning context, we assume some *instance space* $\mathcal{X}$ and a *label space* $\mathcal{Y}$. A *dataset* $D \subset \{(x, y) \mid x \in \mathcal{X}, y \in \mathcal{Y}\}$ is a *finite* relation between the instance space and the label space, and we denote as $\mathcal{D}$ the set of all possible datasets. We consider two types of operations over instance and label spaces:

1. *Pre-processors.* A pre-processor is a function $t : \mathcal{D}_A \to (\mathcal{X}_A \to \mathcal{X}_B)$, where $\mathcal{D}_A$ is the space of datasets with instances from some space $\mathcal{X}_A$. The pre-processor t takes a dataset and maps it to a function, which in turn converts an arbitrary instance x of instance space $\mathcal{X}_A$ into an instance of another instance space $\mathcal{X}_B$.
2. *Predictor Builders.* A predictor builder (often called learner) is a function $p : \mathcal{D}_p \to (\mathcal{X}_p \to \mathcal{Y})$ that takes a dataset with instances from space $\mathcal{X}_p$ and creates a predictor, which assigns an instance of its instance space $\mathcal{X}_p$ to a label in the label space $\mathcal{Y}$. The constructed predictor is also often called a *hypothesis*.

In this paper, a *pipeline* $P = t_1 \circ .. \circ t_k \circ p$ is a sequential concatenation with the usual semantic: Training a pipeline on dataset $D \subseteq \mathcal{X} \times \mathcal{Y}$ means to (i) set $D_0 = D$, (ii) to sequentially induce $\tilde{t}_i = t_i(D_{i-1})$ and compute subsequent data $D_i = \tilde{t}_i(D_{i-1})$ for $1 \leq i \leq k$, and (iii) to eventually create a predictor $\tilde{p} = p(D_k)$. This leads to a *trained* pipeline $\tilde{t}_1 \circ .. \circ \tilde{t}_k \circ \tilde{p}$, which maps an instance $x \in \mathcal{X}$ to a label $y \in \mathcal{Y}$ by passing it through all the $\tilde{t}_i$ and finally the predictor. We denote as $\mathcal{P}$ the space of all such sequential pipelines. In general, the first part of a pipeline could be not only a sequence but also a *pre-processing tree* with several parallel pre-processors that are then merged (Olson and Moore , 2019), but we do not consider such structures in this paper since they are not necessary for our key argument. An extension to such tree-shaped pipelines is canonical future work.

In addition to the sequential structure, many AutoML approaches restrict the search space still a bit further (Thornton et al., 2013; Feurer et al., 2015; Mohr et al., 2018). First, often a particular best order in which different *types* of pre-processor should be applied is assumed. For example, we assume that feature selection should be conducted after feature scaling. So $\mathcal{P}$ will only contain pipelines compatible with this order. Second, the optimal pipeline uses at most one pre-processor of each type. These assumptions allow us to express *every* element of $\mathcal{P}$ as a concatenation of $k + 1$ functions, where k is the number of considered pre-processor *types*, e.g., feature scalers, feature selectors, etc. If a pipeline does not adopt an algorithm of one of those types, say the i-th type, then t_i will simply be the identity function.

The theoretical goal in supervised machine learning is to find a pipeline that optimizes a prediction performance metric (error rate, log-loss, ..) averaged over all instances from the same source as the given data. This performance cannot be computed in practice, so instead one optimizes some function $\phi : \mathcal{D} \times \mathcal{P} \to \mathbb{R}$ that *estimates* the performance of a candidate pipeline based on available *data*, e.g., using hold out or cross validation.

Consequently, a supervised AutoML problem *instance* is defined by a dataset $D \in \mathcal{D}$, a search space $\mathcal{P}$ of pipelines, and a performance estimation metric $\phi : \mathcal{D} \times \mathcal{P} \to \mathbb{R}$ for solutions. An AutoML *solver* $\mathcal{A} : \mathcal{D} \to \mathcal{P}$ is a function that creates a pipeline given some training set $D_{train} \subset D$. The performance of $\mathcal{A}$ is given by $\mathbb{E}\left[\phi\left(D_{test}, \mathcal{A}(D_{train})\right)\right]$, where the expectation is taken with respect to the possible (disjoint) splits of D into D_{train} and D_{test}. The goal of any AutoML solver is to optimize this metric, and we assume that $\mathcal{A}$ has access to ϕ (but not to D_{test}) in order to evaluate candidates with respect to the objective function.

3 Related work

Works on machine learning pipeline optimization can be roughly separated by the core topic they address. Some approaches propose methods to structure and explore a search space. We call these pipeline optimization approaches and discuss them in Sect. 3.1. A second group of approaches focuses on efficiency aspects of the optimization process. Since these ideas often make rather lose assumption about the used optimizer, they are relatively orthogonal and compatible with many optimizers. We discuss them in Sect. 3.2. While these two sections give a rather broad overview, Sect. 3.3 discusses other efforts in simplifying the optimization *problem* in itself.

3.1 Basic pipeline optimization approaches

The first work we are aware of that tries to algorithmically find an optimal pipeline is GEMS (Statnikov et al., 2005). GEMS selection of the best pipeline from a pre-defined portfolio of configurations is based on cross-validation. The problem is here addressed as an algorithm selection problem as hyperparameters are not explicitly subject to optimization. All the subsequent approaches discussed in this section address the combined algorithm selection and configuration (CASH) problem, which is the problem not only to choose an algorithm *or* to optimize hyperparameters but to address *both* aspects simultaneously.

There are mainly three approaches following the idea of tree-based optimization of data science workflows (Engels , 1996). The first approach we are aware of was designed for the configuration of RapidMiner modules based on hierarchical task network (HTN) planning (Kietz et al., 2009, , 2021) most notably MetaMiner (Nguyen et al., 2012, , 2014). With ML-Plan (Mohr et al., 2018), the idea of -based graph definitions was later combined with a best-first search using random roll-outs to obtain node quality estimates. Similarly, (Rakotoarison et al., 2019) introduced AutoML based on Monte-Carlo Tree Search, which is closely related to ML-Plan. However, the authors of Rakotoarison et al. (, 2019) do not discuss the layout of the search tree, which is a crucial detail, because it is the primary channel to inject knowledge into the search problem.

The CASH AutoML problem has also been addressed with evolutionary algorithms. One of the first approaches was PSMS (Escalante et al., 2009), which used swarm particles for optimization. More recent tools include TPOT (Olson and Moore , 2019), RECIPE de (Sá et al., 2017), and GAMA Gijsbers ‖ (Vanschoren , 2019). All these approaches are explicitly or implicitly based on grammars, which allow not just one pre-processing step but an arbitrary number of such techniques and hence go beyond the type of pipelines covered in this paper.

In this, they are similar to the tree search based approaches. The optimizing motor of the above tools is the genetic algorithm scheme NSGA-II (Deb et al., 2002). Focusing on stacking ensembles, another genetic approach was presented with AutoStacker (Chen et al., 2018).

Another line of research based on Bayesian Optimization (BO) was initialized with the advent of Auto-WEKA (Thornton et al., 2013; Kotthoff et al., 2017). Like Naive AutoML, Auto-WEKA assumes a fixed structure of the pipeline, admitting a feature selection step and a predictor. The decisions are encoded into a large vector that is then optimized using the BO tool SMAC (Hutter et al., 2011). Auto-WEKA optimizes pipelines with algorithms of the Java data analysis library WEKA (Hall et al., 2009). Note that experimental comparisons with Auto-WEKA must be handled with care since Auto-WEKA by default discards pipelines that are, in a cross-validation, not competitive after the first fold evaluation. For the Python framework scikit-learn (Pedregosa et al., 2011), the same technique was adopted by auto-sklearn (Feurer et al., 2015). In contrast to Naive AutoML, BO does not greedily commit to some parts of the search space but tries to cover it globally and only exclude parts of it that are unlikely to reveal a new best pipeline.

A recent line of research adopts a type of black-box optimization relying on the framework of multipliers (ADMM) (Boyd et al., 2011). The main idea here is to decompose the optimization problem into two sub-problems for different variable types, considering that algorithm selection variables are Boolean while most parameter variables are continuous. This approach was first presented in Liu et al. (, 2020).

3.2 Efficiency-enhancing technologies

Several techniques have been proposed to identify good pipelines faster. First, *warm-starting* employs not a random initial order of candidates but prioritizes based on beliefs about which pipeline is more suitable. As such, warm-starting is a meta-learning technique (Vanschoren , 2019). Approaches here include nearest neighbors as in auto-sklearn (Feurer et al.,, 2015), collaborative filtering like OBOE (Yang et al., 2019), probabilistic matrix factorization (Fusi et al., 2018), and recommendations based on average ranks (Cachada et al., 2017). Another approach to improve efficiency followed by Successive Halving (SH) and Hyperband (HB) to increase efficiency is multi-fidelity optimization, in which candidates are first evaluated on small budgets (training set sizes) and only considered for high budgets if competitive (Jamieson and Talwalkar , 2016; Li et al., 2017). These approaches are largely orthogonal to our contribution, and several of them could be used complementary inside Naive AutoML, e.g., those for warm-starting or for hyperparameter tuning.

3.3 Simplifying approaches

We are not the only ones to propose simplifications of the search space. An approach for AutoML based on beam search was proposed (Kishimoto et al., 2021) in parallel to our preliminary work (Mohr and Wever , 2021). The idea is very similar to Naive AutoML in that it proposes to quickly prune partial pipelines that look sub-optimal. It is however less extreme since it considers at least a couple of alternatives opposed to Naive AutoML. Second, AutoGluon (Erickson et al., 2020) suggests to not optimize at all but simply apply stacking (Wolpert , 1992) to a set of bagged (Breiman , 1996) learning algorithms, which have been defined a priori. Finally, a decompositional approach was presented in the Dragonfly framework (Kandasamy et al., 2020). This framework models the belief of the objective function as a Gaussian Process (GP), which is however not one huge GP but the sum over "smaller" GPs, one for

each partition of the search space. The motivation for this approach is to break the curse of dimensionality since GPs have been shown to only work well on low-dimensional problems.

4 Naive AutoML and Quasi-Naive AutoML

This section describes the Naive AutoML approach in detail. We first explain the assumptions underlying Naive AutoML in Sect. 4.1. The Naive AutoML algorithm itself is then formally introduced in Sects. 4.2, and 4.3 explains its modification towards Quasi-Naive AutoML.

4.1 Assumptions

Naive AutoML builds on top of two assumptions. First it assumes that pipeline slots can be optimized locally, which is formalized in Sect. 4.1.1. Second, it assumes that the best tuned algorithm for a slot is also the algorithm that performs best if being used with default parameters. This is discussed in Sect. 4.1.2.

4.1.1 Naivety assumption

Naive AutoML assumes that the optimal pipeline is the one that is locally best for each of its pre-processors and the final predictor. In other words, taking into account pipelines with (up to) k pre-processors and a predictor, we assume that for all datasets D and all $1 \leq i \leq k+1$

$$c_i^* \in \arg\min_{c_i} \phi(D, c_1 \circ .. \circ c_{k+1}) \tag{1}$$

is *invariant* to the choices of $c_1, ..c_{i-1}, c_{i+1}, .., c_{k+1}$. Note that, for simplicity of notation, we here use the letter c instead of t for pre-processors or p for the predictor.

We dub the approach Naive AutoML because of the assumption of independence of decisions. Consider $\mathcal{P}$ an urn of pipelines and denote as Y the event that an optimal pipeline is drawn. Then

$$\mathbb{P}(Y \mid c_1, .., c_{k+1}) \propto \mathbb{P}(c_1, .., c_{k+1} \mid Y)\mathbb{P}(Y) \overset{naive}{=} \mathbb{P}(c_i \mid Y) \prod_{j=1, j\neq i}^{k+1} \mathbb{P}(c_j \mid Y)\mathbb{P}(Y),$$

in which we consider c_j to be fixed components for $j \neq i$, and only c_i being subject to optimization. Applying Bayes' theorem again to $\mathbb{P}(c_i \mid Y)$ and observing that the remaining product is a constant regardless the choices of $c_{i\neq j}$, it follows that the optimal solution is the one that maximizes the probability of being locally optimal, and that this choice is *independent* of the choice of the other components. This is identical to the assumption that motivates the Naive Bayes classifier.

A direct consequence of the naivety assumption is that we can leave all components $c_{j\neq i}$ except the predictor component c_{k+1} even *blank* when optimizing c_i. This is because (i), by the naivety assumption, the choice for those components would not influence the best choice for c_i and (ii), by pipeline syntax, they are not required to construct a pipeline that

can be evaluated. The latter is however not true for the last component: We cannot assess the performance of a pipeline that only has a pre-processor but no predictor. So we cannot optimize pre-processing slots without using any predictor in the final slot. This being said, when optimizing a pre-processor c_i, we will have to commit to some predictor in the slot c_{k+1}; below we explain two strategies to set the predictor. On the other hand, it seems usually reasonable (under the naivety assumption) to leave the other pre-processing slots $\neq i$ blank. Pre-processing algorithms usually lead to a net increase of the training time of a pipeline in spite of potentially reduced dataset sizes on which subsequent algorithms in the pipeline work. Omitting them hence often substantially reduces the runtime of candidate evaluations.

It is clear that the naivety assumption does seldomly hold in practice. One way to see this is the fact that it would enable us even to use a guessing predictor to optimize the pre-processing steps. In fact, a reasonable default choice for the predictor would be the *fastest* learner, and the arguably fastest algorithm is one that just guesses an output (or maybe always predicts the most common label). It is unlikely that such a predictor is of much help when optimizing a pre-processor even if it is part of the candidates for c_{k+1}. On the practical side, we circumvent this problem by choosing a predictor that is known to frequently exhibit learning behavior on data such as kNN or a decision tree.

4.1.2 Separate algorithm selection and algorithm configuration

On top of the naivety assumption, Naive AutoML additionally assumes that even each component c_i can be optimized by local optimization techniques. More precisely, it is assumed that the algorithm that yields the best component when using the default parametrization is also the algorithm that yields the best component if all algorithms are run with the best parametrization possible.

Just like for the naivety assumption itself, we stress that this assumption is just an algorithmic *decision*, which does not necessarily hold in practice. In fact, the results on some datasets in the experiments clearly suggest that this assumption is not always correct. Our goal is precisely to study the *extent* by which state-of-the-art approaches can improve over the naive approach by *not* making this kind of simplifying assumptions.

However, our experiments indicate that the variance of the variable describing the *improvement* of a hyperparameter configuration over the performance with default configuration is relatively low for many learners – at least for the here considered datasets and learners. In other words, we are well aware that the simplifying assumptions are not always correct (specifically keeping some exceptions like neural networks or SVMs in mind), but we are interested in quantifying (on a moderate empirical basis) how often an optimal solution is indeed easily identified by a naive approach; this is surprisingly often the case. This should serve as a baseline for future work.

4.2 The Naive AutoML optimizer

The Naive AutoML optimizer is formally described in Alg. 1 and consists of three phases (i) algorithm selection, (ii) hyperparameter tuning, and (iii) definition and training of the final pipeline.

In the first phase (l. 1-9), it selects the best component for each slot of the pipeline based on default hyperparameter values. For each slot s and each component c_s that can be used for it, the algorithm builds one pipeline that only consists of c_s. This is done via the function getPipeline(s, c_s, θ_s), in which the third argument θ_s are the hyperparameters for c_s; these are omitted in the first phase (indicated by the $\perp$ symbol) so that the default hyperparameters are being used. If s is a pre-processor slot, getPipeline appends an additional standard prediction component, e.g., kNN. The score of that pipeline is computed with a customizable validation function Validate, e.g., k-fold cross-validation with some arbitrary metric. We here assume w.l.o.g. that the metric is to be minimized. Whenever a new (locally) best solution is found, it is memorized (v_s^*).

Algorithm 1 Naive AutoML - Optimization Routine

Require: Components $\mathcal{C} = (C_1, .., C_{k+1})$ for pipeline slots
Require: dataset D (only used for final training)
Require: validation function VALIDATE (has access to D)
1: **for all** slot $s \leftarrow 1, .., k+1$ **do**
2: $c_s^*, v_s^* \leftarrow \perp, \infty$ // dummy initialize vars for best choice/score for this slot
3: **for all** candidate $c_s \in C_s$ in random order **do**
4: $v_s \leftarrow$ VALIDATE(GETPIPELINE$(s, c_s, \perp)$) // get score of pipeline with c_s in slot s
5: **if** $v_s < v_s^*$ **then**
6: $c_s^*, v_s^* \leftarrow c_s, v_s$ // if this is a best observation for this slot, memorize the decision
7: **end if**
8: **end for**
9: **end for**
10: $\theta_s^* \leftarrow \perp \quad \forall s : 1 \leq s \leq k+1$ // set best seen hyperparameters to None
11: **while** timeout not reached **do**
12: **for all** slot $s \leftarrow 1, .., k+1$ **do**
13: $\theta_s \leftarrow$ choose configuration for c_s^*
14: $v_s \leftarrow$ VALIDATE(GETPIPELINE(s, c_s, θ_s))
15: **if** $v_s < v_s^*$ **then**
16: $\theta_s^*, v_s^* \leftarrow \theta_s, v_s$ // update hyperparams for algorithm of slot
17: **end if**
18: **end for**
19: **end while**
20: $i \leftarrow 1$ // last phase, train and, if necessary, trim pipeline
21: **while** training $((c_i^*, \theta_i^*), .., (c_{k+1}^*, \theta_{k+1}^*))$ on D fails **do**
22: $i \leftarrow i+1$
23: **end while**
24: **return** trained pipeline $((c_i^*, \theta_i^*), .., (c_{k+1}^*, \theta_{k+1}^*))$

In the second phase (l. 10-19), the algorithm runs in rounds in which it tries new hyperparameters θ_s for each component c_s^* (in isolation). If the performance of such a pipeline is better than the currently best, the hyperparameters θ_s^* for that slot's component are updated correspondingly. In our implementation, the non-deterministic choice in l. 13 is simply a uniform random sample from the space of hyperparameter values. Instead of optimizing slot after slot for some time, each main HPO step performs one optimization step for each slot. This procedure is repeated until the overall timeout is exhausted. Interleaving the hyperparameter tuning steps has little effect in our random search implementation but plays a role if the step in l. 13 employs a model-based optimizer, which can constraint the search space to several small spaces instead of a single

exponentially bigger one. Analyzing the implication of this when using, for example, Bayesian Optimization, is interesting future work.

In the final phase (l. 20-24), the pipeline defined by the local decisions is trained and returned. Let $p^* = ((c_1^*, \theta_1^*), .., (c_{k+1}^*, \theta_{k+1}^*))$ be that pipeline. It can happen (and in practice, it *does* happen occasionally) that p^* is not executable on specific data. For example, a pipeline p^* for scikit-learn (Pedregosa et al., 2011) may contain a StandardScaler, which produces negative attribute values for some instances, and a MultinomialNB predictor, which cannot work with negative values. Since the two components were never executed together *during* search, the optimizer did not detect any problem with the two outputs StandardScaler and MultinomialNB in isolation (and according to the naivety assumption no problem should occur). Several repair possibilities would be imaginable, e.g., to replace the pre-processors with earlier found candidates for that slot, or to simply try earlier candidates of p^*. To keep things simple, in this paper, we just removed pre-processors from left to the right until an executable pipeline $p^{*\prime}$ is created; in the extreme case just leading to a predictor without pre-processors. This case should however be rare. In our experiments, it occurred in less than 1% of the runs. The effect does not occur, by construction, in Quasi-Naive AutoML.

For simplicity of the code, the training of the final pipeline is not included in the overall timeout. Anticipating this runtime is a non-trivial problem, which can however be tried to be treated with local runtime models of the components (Mohr et al., 2021).

4.3 The Quasi-Naive AutoML optimizer

The Quasi-Naive AutoML Optimizer makes two minor changes in the above code of Naive AutoML. First, it defines a permutation σ on the set of slots $\{1, .., k+1\}$ in which they should be optimized. This order is used to traverse the loop in the first phase, i.e., $s \leftarrow \sigma(1), .., \sigma(k+1)$ in l. 1. Since every pipeline must contain a predictor, σ will order the predictor first, i.e., $\sigma(1) = k+1$, and then assume some order of decisions on the pre-processors. In our experiments, we used $\sigma(i) = i - 1$ for $i > 1$. Second, the getPipeline routine does not leave components of previous decision steps blank (or plugs in the default predictor) but puts in the component c_s^* chosen for the respective slot s in its default parametrization. More formally, if $\sigma(i) < \sigma(j)$ and the algorithm is building a pipeline with slot j as decision variable, then slot i is filled with $(c_i^*, \perp)$. Notably, it does *not* use the best hyperparameters found for c_i, not even in the second phase. In practice, one would of course use the best hyperparameters θ_i^* seen so far in phase 2 instead of $\perp$ since this has no extra cost over $\perp$. Here we abstain from this strategy since we are verifying the naivety assumption, which tells that θ_i^* should not affect the choice of θ_j.

Under this adjustment, the naivety assumption in Eq. (1) is relaxed as follows. Instead of assuming that *all* other components are irrelevant for the best choice of a component in the pipeline, one now only assumes that the *subsequently* chosen components are irrelevant for the optimal choice. In contrast, the *previously* made decisions *are* relevant for the current optimization question. Concerning the naivety assumption, they are relevant in the sense that the previously decided components cannot be chosen arbitrarily in the naivety property but are supposed to be fixed according to the choice that was made for that slot.

In practice, Quasi-Naive AutoML is usually preferable over strict Naive AutoML. The only advantage offered by strict Naive AutoML is that one can optimize the different slots in parallel. However, Quasi-Naive AutoML can also be parallelized in the

optimization process of a *single* slot (evaluate several candidates in a slot in parallel), so strict Naive AutoML is only of theoretical interest.

5 Evaluation

We compare Naive AutoML and Quasi-Naive AutoML with (even) simpler baselines as well as state-of-the-art optimizers used in the context of AutoML. We stress that we aim at comparing *optimizers* and not whole AutoML tools. That is, we explicitly abandon previous knowledge that can be used to warm-start an optimizer and also abandon post-processing techniques like ensembling (Feurer et al., 2015); Gijsbers || (Vanschoren , 2019) or validation-fold-based model selection (Mohr et al., 2018). Those techniques are (largely) orthogonal to the optimizer and hence irrelevant for its analysis. It is, of course, conceivable that some optimizers benefit more from certain additional techniques like warm-starting etc. than others, but this kind of analysis is beyond our scope.

When comparing the naive approaches with state-of-the-art optimizers, we should recognize that the naive approaches are indeed *very* weak optimizers. First, in contrast to global optimizers, the naive approaches do not necessarily converge to an optimal solution because large parts of the search space, possibly containing the optimal solution, are pruned early. In other words, the naive approaches cannot outperform the others in the long run. Second, the highly stochastic nature of the algorithms also does not give high hopes for great performance in the short run. Both Naive AutoML and Quasi-Naive AutoML are closely related to random search, which can be considered one of the most simple baselines. In fact, Naive AutoML *is* a random search in a decomposed search space: While the HPO phase is an explicit random search, the algorithm selection phase simply iterates over all possible algorithms, which is equivalent to a random search due to the small number of candidates (all of them are considered anyway).

To operationalize the terms “short run” and “long run”, we choose time windows of 1h and 1d, respectively. These time limits are, of course, arbitrary but are common practice (Thornton et al., 2013; Feurer et al., 2015; Mohr et al., 2018) and seem to represent a good compromise taking into account the ecological impact of such extensive experiments.

These observations then motivate three research questions, all of which are limited to the context of single-label classification, i.e., ignoring multi-label classification, regression, and other problems:

RQ 1: Do the naive approaches find better pipelines than state-of-the-art (SOTA) optimizers in the *short run*?
RQ 2: By *which margin* can SOTA optimizers outperform the naive approaches in the long run and *how long* do they need to achieve such a performance?
RQ 3: To which degree is the naivety assumption justified as far as algorithm selection is concerned?

Due to the common limitations in this type of research, we answer the above questions in a limited way based on a collection of datasets. In principle, the questions require to generalize over *all* possible datasets, which is not feasible in practice. Our evaluation and hence the possible conclusions are limited to a collection of 62 datasets on binary and multi-class classification described below.

In the whole evaluation, the default classifier used in Naive AutoML is a kNN algorithm with $k = 5$. This classifier is used whenever Naive AutoML configures a pre-processing algorithm. For Quasi-Naive AutoML, this is not necessary since the classifier is the first algorithm to be fixed, so whenever a pre-processing algorithm is optimized, the classifier has already been chosen before.

5.1 Experiment setup

5.1.1 Compared optimizers and search space definition

The evaluation is focused on the machine learning package scikit-learn (Pedregosa et al., 2011). As simple baselines we consider a random forest (Breiman , 2001) and a random search that uniformly draws (unparametrized) pipelines and then uniformly chooses the values for the hyperparameters. On the state-of-the-art side, we compare solutions with the competitive AutoML tools auto-sklearn (Feurer et al., 2015) and GAMA Gijsbers ‖ (Vanschoren , 2019). For the former, we use version 0.12.6, which underwent substantial changes and improvements compared to the original version (Feurer et al., 2015). We are not aware of other approaches that have shown to substantially outperform these tools at the optimizer level. Some works claim to outperform auto-sklearn Rakotoarison et al. (, 2019) and Liu et al. (, 2020), but the implementations are either not available Liu et al. (, 2020) or could not be adjusted to our setup (Rakotoarison et al. (, 2019). Since the claimed gaps are either not reported (Liu et al., 2020) or mostly small (Rakotoarison et al., 2019), we ignored those approaches in the evaluation. Code of the naive approaches and the experiments are available.[1]

Focusing only on the optimizers, the state-of-the-art baselines are Bayesian optimization (BO) and evolutionary algorithms (EA). The tools only serve to setup those optimizers for an AutoML task. auto-sklearn employs BO by means of SMAC (Hutter et al., 2011). In a nutshell, SMAC is a BO approach that uses Random forests (Breiman , 2001) to model the objective function and query next candidates for evaluation. GAMA employs the EA by means of the NSGA-II-based optimization (Deb et al., 2002). NSGA-II is a genetic algorithm capable of optimizing multiple objectives simultaneously and returning a set of non-dominated solutions.

To maximize the comparability and avoid confounding factors, (i) all components of the tools except the optimizer have been disabled, (ii) the search space has been unified, and (iii) a common pipeline evaluation technique has been applied. Aspect (i) refers to disabling warm-starting and ensemble building. Regarding (ii), we adopted the pipeline structure dictated by auto-sklearn since all tools except auto-sklearn can be configured relatively easily in their search space and pipeline structure. This pipeline consists of three steps, including so-called *data-pre-processors*, which are mainly feature scalers, *feature-pre-processors*, which are mainly feature selectors and decomposition techniques, and finally the estimator. Appendix B shows the concrete list of algorithms used for each category. We also used the hyperparameter space defined by auto-sklearn for each of the components. Unfortunately, the search spaces are not absolutely identical as auto-sklearn has proprietary

[1] https://github.com/fmohr/naiveautoml.

components (balancing, minority coalescer) that cannot be switched off or easily added to the other tools. The search spaces of the naive approaches and GAMA are identical except that GAMA requires explicitly described domains for the hyperparameters, which does not match the concept of numerical hyperparameters used in auto-sklearn and the naive approaches through the ConfigSpace library (Lindauer et al., 2019). We hence sampled 10000 values for each *hyperparameter* and used these as a discrete space; this sampling mechanism already included log-scale sampling where applicable.[2] To achieve (iii), the evaluation mechanism for a concrete pipeline candidate was fixed among all approaches to 5-fold cross-validation.

5.1.2 Benchmark datasets

The evaluation is based on the datasets in the "AutoML Benchmark All Classification" study[3] on the openml.org platform (Vanschoren et al., 2013). The covered datasets are a superset of those proposed in Gijsbers et al. (, 2019) and cover classification for both binary and multi-class classification with numerical and categorical attributes. Within this scope, the dataset selection is quite diverse in terms of numbers of instances, numbers of attributes, numbers of classes, and distributions of types of attributes. Appendix A lists the relevant properties of each of these datasets to confirm this diversity. Our assessment is hence limited to binary and multi-class classification.

Datasets with missing values or categorical attributes were pre-processed. Missing values were imputed by the median (numerical attributes) or mode (categorical attributes), and categorical attributes were replaced by a Bernoulli encoding. Thereby we avoid implicit search space differences, because auto-sklearn comes with some pre-processors specifically tailored for categorical attributes. Since these are partially hard-coded and not easily applicable with GAMA and the naive approaches, we simply eliminated this decision variable from the search space. This pre-processing should usually be done by the optimizer itself only on the training data, but we could not modify auto-sklearn and GAMA accordingly, so that we took this middle ground solution; we expect the side effects by this decision to be small. Even though the imputation is identical for all optimizers and hence probably without too much effect on the comparison, it is arguably arbitrary, so that we excluded datasets with more than 5% missing values from the experiments. The final evaluation was conducted on the remaining 62 datasets.

5.1.3 Validation mechanism and performance metrics

Results are reported summarizing, in different forms, log-losses computed in 10 repeated runs per dataset for each optimizer. For this, we chose a 90% train fold size and a 10% test fold size. Running each optimizer 10 times with different such random splits corresponds to a 10 iterations Monte Carlo cross-validation with 90% train fold size. Of course, splits were identical per seed among all optimizers. The minimized metric is the log-loss as suggested in the context of the AutoML benchmark (Gijsbers et al.,

[2] All these efforts were realized in collaboration with the authors of GAMA.

[3] The original study is found at https://www.openml.org/s/271. The selection here is based on a discussion around this study https://github.com/openml/automlbenchmark/issues/187#issuecomment-740716098.

2019) for multi-class classification. We also use it for binary classification to keep the overview simpler.

Note that our primary focus here is *not* on test performance but validation performance. This paper compares *optimizers*, so we should measure them in terms of what they optimize, namely validation performance. It can clearly happen that strong optimization of that metrics yields no better or even worse performance on the test data (over-fitting). Even though test performance is, in our view, not relevant for the research questions, we conduct the outer splits and hence provide test performance results in order to maximize insights for the 1d run.

5.1.4 Resources and used hardware

Timeouts were configured as follows. For the short (long) run, we applied a total runtime of 1h (24h), and the runtime for a single pipeline execution was configured to take up to 20 minutes (for both scenarios). The memory was set to 24GB and, despite the technical possibilities, we did *not* parallelize evaluations. That is, all the tools were configured to run with a single CPU core. The computations were executed in a computing center with Linux machines, each of them equipped with 2.6Ghz Intel Xeon E5-2670 processors and 32GB memory.

5.2 Results

5.2.1 RQ 1: Do the naive approaches find better pipelines than state-of-the-art (SOTA) optimizers in the *short run*?

To answer this question, consider Fig. 1, which summarizes the results for an overall timeout of 1h in terms of *validation performance*. That is, the performance of an optimizer up to some point of time t is the best (lowest) *average log-loss* observed in any 5-fold CV of pipeline evaluations up to t. Averaging these scores across the different runs of an optimizer on a dataset defines an anytime curve for each optimizer and dataset, and the left plot shows the *mean ranks* inferred from those curves over time. The right plot summarizes the absolute gaps in terms of log-loss to the best solution at some point of time. That is, on a specific random seed, there is at each point of time t some approach that has observed a best performance. The gap of an optimizer at time t is the difference between the performance of the best pipeline it has tried up to t and the best such performance among all

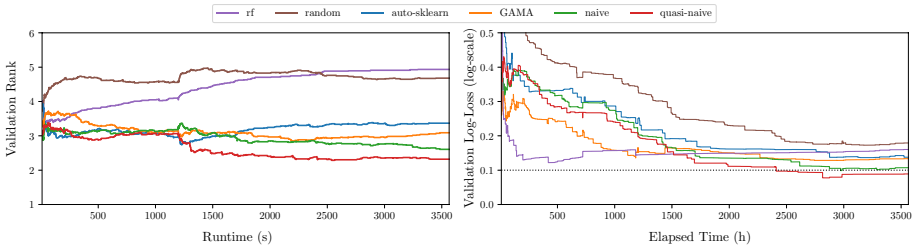


Fig. 1 Mean validation ranks (left) and gaps (right) after 1h. On the right, the black dotted line is a visual aid for a log-loss gap of 0.1, which is arguably negligible

optimizers. The mean gap is drawn as a solid line. The horizontal black dotted line is a visual aid for a log-loss of 0.1.

The plot shows that the naive approaches are competitive or even stronger than state-of-the-art tools in the short run. Naive AutoML is competitive with GAMA both of which outperform auto-sklearn 's SMAC in this time horizon. The right plot reveals that both naive approaches exhibit a gap of less than 0.1 after 30 minutes of runtime on average. It is even less than 0.05 in over 75% of the cases, and Quasi-Naive AutoML already achieves this after 20 minutes (not shown).

Clearly, the performance differences in general are rather small. Arguably, gaps in log-loss below 0.1 can be considered somewhat negligible. If the difference in log-loss between two models is below 0.1 this means that the *ratio* of probabilities assigned to the correct class is, on average, around 1.1. For a binary classification problem, this means that, even for situations of rather high uncertainty, if the better model assigns 55% probability to the correct class, the weaker model also still assigns at least 51% probability to the correct class and will hence choose it. Now, this degree of irrelevance increases with a higher certainty of the better model or with higher numbers of classes. In other words, in concrete situations where the two or three classes with the highest probability are at par, small differences in log-loss will not necessarily but often also indicate identical behaviors of the models.

The simple baselines also play an interesting role in this evaluation. First, a simple random forest is clearly the best solution for the first 15 minutes. In other words, on the examined datasets, if a timeout of less than 15 minutes is considered, it is more recommendable to not use an optimizer at all but simply take a random forest. Of course, this is due to the fact that the optimizers are cold-started. Using warm-starting techniques, random forests are typically among the first tried models. On the other hand, if the runtime is higher than 15 minutes, random forests get more and more sub-optimal. Second, the random search is consistently outperformed in the short run by all techniques. This means that searching blindly is a poor strategy, which is what one would expect.

Putting everything together, our assessment is that the naive approaches indeed compete with or even outperform the other approaches in the short run. In this we ignore runtimes of less than 15 minutes for which a simple random forest is preferable over optimizing at all (at least on the considered datasets). Neither auto-sklearn nor GAMA can substantially outperform even the strict Naive AutoML approach in terms of validation performance; rather the contrary is true. Overall, both Naive AutoML and Quasi-Naive AutoML are competitive with auto-sklearn and GAMA and even slightly outperform both of them in many cases. Among the two, Quasi-Naive AutoML has a small advantage over Naive AutoML and hence should be preferred since it has virtually no relevant disadvantage over the strictly naive approach.

5.2.2 RQ 2: By *which margin* can SOTA optimizers outperform the naive approaches in the long run and *how long* do they need to achieve such a performance?

To get a first idea about the behavior of the optimizers in the long run, we again consider the rank and gap plots, this time for the timeout of 24h in Fig. 2. The semantics in the figures are the same as above, but there are some additional visual elements. As expected, we can observe that, over time, the more sophisticated optimizers gain an advantage over the naive approaches. We added a first vertical dotted black line at the point of time where Naive AutoML is outperformed by GAMA (and quickly later by auto-sklearn) in terms of

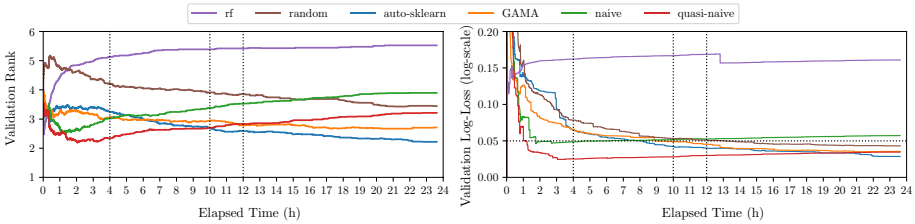


Fig. 2 Same semantics as for Fig. 1 but after 24h. The vertical lines show the points of time when, in terms of average rank, Naive AutoML is outperformed by GAMA, and when Quasi-Naive AutoML is outperformed by auto-sklearn and GAMA respectively. The visual aid on the right has been adjusted to an even tighter gap bound of 0.05

average rank. Next, we insert two such vertical lines at the respective points of time where auto-sklearn and GAMA start to rank better than Quasi-Naive AutoML.

With respect to the research question, we first observe that there is on average *no* advantage of neither auto-sklearn nor GAMA over the naive approaches within several hours. Indeed, both optimizers achieve to outperform strict Naive AutoML after approximately 4h to 5h in terms of average ranks. In terms of average gaps, this point is reached only after 10h. However, they need much more time to achieve the same effect against Quasi-Naive AutoML. In terms of ranks, this point sets in after approximately 10h of runtime for auto-sklearn and after 12h of runtime for GAMA. In terms of average gaps, both auto-sklearn and GAMA do *not* obtain better average gap within 24h. The latter however does not mean that auto-sklearn or GAMA would not occasionally outperform Quasi-Naive AutoML; the advantage is just so small that it vanishes in the average. What can be said, for the considered datasets, is that for scenarios of less than 10h of runtime, there is little reason to prefer either of those tools over Quasi-Naive AutoML.

Considering now the whole time horizon of 24h, the possible improvements of the state-of-the-art optimizers are surprisingly small. Being ranked at position 2 on average, the BO of auto-sklearn is the best optimizer in the long run, and the NSGA-II optimizer of GAMA ranks slightly better than rank 3 on average. However, some statistics not shown in the figures for readability reveal that the advantages of solutions found by those tools are really small in terms of actual gaps. The 0.75 quantile of gaps of Quasi-Naive AutoML is located at 0.025, from which we can conclude that Quasi-Naive AutoML achieves virtually optimal performance in at least 75% of the cases. Even the 0.95 quantile is located at 0.07. If we consider, as argued above, that gaps below 0.1 are rather negligible, then Quasi-Naive AutoML delivers on at most 3 of the 62 datasets a pipeline that is not close to optimal. In fact, even the strictly naive approach performs competitive on average in terms of gaps. However, the 0.95 quantile for gaps of the strict Naive AutoML approach is increased and indicates that there is a significant number of datasets on which strict Naive AutoML is indeed sub-optimal.

To complement these insights on validation performance with those on test performance, we also report the distributions of absolute gaps after 24h on the test folds. These results are summarized in Fig. 3. In general, auto-sklearn produces the best or second-best test performance in 50% of the cases whereas GAMA and Quasi-Naive AutoML have a slightly worse test rank performance (left plot). Among these two, both have the same median rank, but GAMA scores slightly better under the q1-quantile. Naive AutoML is outperformed in terms of ranks in this time horizon. However, the right plot again shows

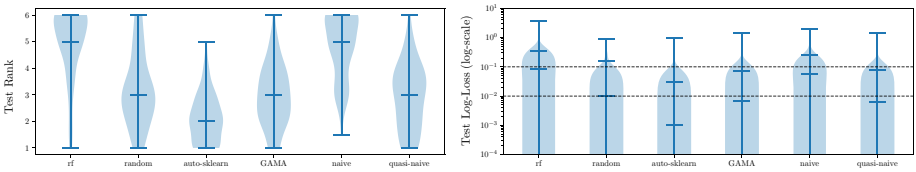


Fig. 3 Final ranks (left) and gaps (right) on the *test* performance on 24h timeouts. Whiskers show the medians, 90% quantiles, and maxima. Dotted lines are visual aids for 0.1 and 0.01 (colour figure online)

that differences are minimal. Quasi-Naive AutoML has a gap of less than 0.01 in 50% of the cases and less than 0.1 in 90% of the cases. Appendix C shows the concrete results per dataset.

This being said, we answer the research question as follows. auto-sklearn, as the algorithm that shows the best performance on *most* datasets after 24h, exhibits small to no relevant performance advantage over the naive approaches on 50% of the datasets. More precisely, the gap of Naive AutoML on over 50% of the datasets is smaller than 0.1, a fairly low value. For Quasi-Naive AutoML, the median gap is even close to 0.01, which can be considered de facto optimal in the huge number of cases in practice: for a single instance, a log-loss of 0.01 corresponds to a probability of above 0.99 assigned to the correct class. While auto-sklearn is able to significantly outperform Naive AutoML in the long run on *some* datasets, it rarely ever outperforms Quasi-Naive AutoML. The performance gap of Quasi-Naive AutoML is bigger than 0.1 only once and smaller than 0.05 in more than 80% of the cases. The same comparison holds for GAMA against Quasi-Naive AutoML; in fact the advantage of GAMA over Quasi-Naive AutoML is only minimal. To summarize, in the scope of the considered datasets, state-of-the-art tools can hardly find significantly better solutions than Quasi-Naive AutoML. On roughly 45% of the datasets, they can achieve small but measurable improvements if run for at least 18 hours, and only in one case they can substantially outperform Quasi-Naive AutoML.

5.2.3 RQ 3: To which degree is the naivety assumption justified as far as algorithm selection is concerned?

The strong results of Quasi-Naive AutoML motivate a dedicated analysis of the legitimacy of the naivety assumption since this would explain its success. We have examined this legitimacy in the scope of the used datasets. To this end, we computed for all of the datasets the performance of all pipelines that can be built with the considered algorithms (with default hyperparameter values). The number of such pipelines is 1200 in our case, and since the evaluation under experiment conditions almost always takes more than 24h (in fact 48h on average), we did not include this procedure into the set of baselines. For each such run, we identified the set of 0.03-optimal pipelines, i.e., the pipelines that have a (validation) log-loss of at most 0.03 more than the optimal one. For each pipeline slot, the set of choices that is accepted as a correct choice is precisely the union of the algorithms that occur in any of the 0.03-optimal pipelines for the respective slot.

The results are summarized in Fig. 4 (exact values in Appendix D). For each of the three pipeline slots and each of the datasets, we report how often (among the 10 seeds) an optimal algorithm was chosen by Quasi-Naive AutoML. Dark green/red means that

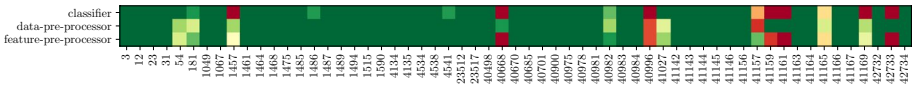


Fig. 4 Optimality of decisions of Quasi-Naive AutoML by pipeline slot per dataset aggregated over the ten seeds. Green/Red means: The choice was always optimal/sub-optimal. Yellow colors in-between indicate that the choice was optimal in some runs and sub-optimal in others

Quasi-Naive AutoML always chose an optimal/suboptimal algorithm. For some datasets, the situation is on the edge, which can be seen from the yellow or orange fields. As the figure shows, Quasi-Naive AutoML picks in 78% of the time the correct classifier. It picks the correct data-pre-processor in 85% of the cases and the correct feature-pre-processor in 80% of the cases. Clearly, if a wrong classifier is chosen, then this means that a classifier that is sub-optimal on its own can be combined with some pre-processor together with which it then outperforms the best stand-alone classifier. While this shows that the naivety assumption is not generally correct, we see that it works just fine in *many* cases. One question for future work is whether a slightly less naive algorithmic scheme such as (Kishimoto et al., 2021) can cover the remaining cases. While our analysis is limited to the 62 datasets under consideration, the study makes a strong case for Quasi-Naive AutoML.

5.3 Discussion

Putting the results together, the naive approach seems to make a maybe unexpectedly strong case against established optimizers for standard classification problems. Even the fully naive approach is competitive in the long run in 50% of the cases. When applying the quasi-naive assumption, we obtain an optimizer that is, on the analyzed datasets, hardly ever significantly outperformed neither by auto-sklearn nor by GAMA. Both AutoML and GAMA manage to gain measurable advantages over Quasi-Naive AutoML as runtime increases, but these are negligible most of the time. Whether or not Quasi-Naive AutoML is likewise competitive in practical applications still stands to be shown. Summarizing, the naive methods are somewhere inbetween yet simpler baselines and fully-fledged optimizers when no time limits are applied, and they perform better in the short term.

Our results suggest an entirely new way of thinking about the optimization process in AutoML. Until now, pipeline optimization has almost always been treated as a complete black-box. However, the strong performance of Quasi-Naive AutoML suggests that the optimization process can be realized *sequentially*. The ability of sequential optimization opens the door to optimization *flows*, which in turn give room for specialized components within the optimization process (Mohr and Wever , 2021). For example, based on the observations in the optimization of one slot, it would be possible to activate or deactivate certain optimization modules in the subsequent optimization workflow. Since this paper has shown even Quasi-Naive AutoML to be competitive, there is some reason to believe that such more sophisticated approaches might even be superior to black-box optimization.

6 Conclusion

In this paper, we have presented two *naive* approaches for the optimization of machine learning pipelines. Contrary to previous works, these approaches fully (Naive AutoML) or largely (Quasi-Naive AutoML) ignore the general assumption of dependencies between

the choices of algorithms within a pipeline. Furthermore, algorithm selection and hyperparameter optimization are decoupled by first selecting the algorithms of a pipeline only considering their default parametrizations. Only when the algorithms are fixed, their hyperparameters are optimized.

Results on 62 datasets suggest that naive approaches are much more competitive than one would maybe expect. For short timeouts (1h), both naive algorithms perform highly competitive to optimization algorithms of state-of-the-art AutoML tools and sometimes (Quasi-Naive AutoML in fact even consistently) superior. In the long run, 24h experiments show that Quasi-Naive AutoML is largely en par with auto-sklearn and GAMA in terms of gaps to the best solution.

We stress that our results do not imply that global approaches (Feurer et al., 2015; Olson and Moore , 2019; de Sá et al., 2017; Mohr et al., 2018) are obsolete. Not only is the number of datasets of our study too limited to draw such a strong conclusion with confidence, but also is it possible that significantly better solutions exist in the global search space that are simply not found by current global optimizers. Examining this question in more depth is a highly non-trivial research prospect, which, because of the search space size, calls for approaches that quantify the *probability* of having found an optimal pipeline based on certain smoothness assumptions. Besides, other techniques like ensembling and warm-starting (Feurer et al., 2015) can have different influence among the approaches, so the results only apply to *optimizers* but not whole *tools*.

However, our results clearly suggest the possibility of the existence of a generally competitive semi-greedy pipeline optimizer. This demands further research and also calls for more challenging benchmarks in which a simple greedy strategy does not perform so strong. In fact, our findings have made such benchmarks now necessary to better justify the usage of highly sophisticated methods.

Many aspects in this paper are deliberately kept simple to show the strength of this simple scheme, and these limitations offer several plans for future work. These include (i) imputation and treatment of categorical attributes as a part of the optimization, (ii) apply more sophisticated HPO techniques such as Bayesian Optimization, and (iii) coverage of tree-like pipelines instead of sequences only and a relaxation of a particular shape of the pipeline in general.

Besides, the sequential optimization flow of Naive AutoML naturally motivates a series of future work building upon this property. It seems imperative to further explore the potential of a less naive approach as suggested in Mohr and Wever (, 2021), which adopts a stage-based optimization scheme. Another interesting direction is to create a more interactive version of Naive AutoML in which the expert obtains visual summaries of what choices have been made and with the option for the expert to intervene, e.g., by revising some of the choices. This could lead to an approach considering different optimization *rounds* for different slots.

Appendix A: Datasets

All datasets are available via the openml.org platform (Vanschoren et al., 2013) (Table 1).

Table 1 Overview of datasets used in the evaluation. Meaning of the columns in this order: id at openml.org, name at openml.org, whether or not the dataset is part of the AutoML benchmark (Gijsbers et al., 2019), number of instances, number of features, number of numerical features, number of classes, percentage of smallest class, percentage of biggest class, % of missing entries, percentage of features in the [0, 1] interval, percentage of centered features, percentage of features with a standard deviation of 1. The last three only apply if there are numerical features

Id	in Gijsbers et al. (, 2019)	Name	Instances	Features	Numeric features	Classes	Min %	Maj %	% missing	% [0,1]	% $\mu = 0$	% $\sigma = 1$
3	1	kr-vs-kp	3196	36	0	2	47%	52%	0%	NaN	NaN	NaN
12	1	mfeat-factors	2000	216	216	10	10%	10%	0%	0%	0%	0%
23	0	cmc	1473	9	2	3	22%	42%	0%	0%	0%	0%
31	1	credit-g	1000	20	7	2	30%	70%	0%	0%	0%	0%
54	1	vehicle	846	18	18	4	23%	25%	0%	0%	0%	0%
181	0	yeast	1484	8	8	10	0%	31%	0%	25%	0%	0%
1049	0	pc4	1458	37	37	2	12%	87%	0%	3%	0%	0%
1067	1	kc1	2109	21	21	2	15%	84%	0%	0%	0%	0%
1457	0	amazon-commerce-revi	1500	10000	10000	50	2%	2%	0%	22%	0%	0%
1461	1	bank-marketing	45211	16	7	2	11%	88%	0%	0%	0%	0%
1464	1	blood-transfusion-se	748	4	4	2	23%	76%	0%	0%	0%	0%
1468	1	cnae-9	1080	856	856	9	11%	11%	0%	88%	0%	0%
1475	0	first-order-theorem-	6118	51	51	6	7%	41%	0%	0%	4%	2%
1485	0	madelon	2600	500	500	2	50%	50%	0%	0%	0%	0%
1486	1	nomao	34465	118	89	2	28%	71%	0%	83%	0%	0%
1487	0	ozone-level-8hr	2534	72	72	2	6%	93%	0%	0%	0%	0%
1489	1	phoneme	5404	5	5	2	29%	70%	0%	0%	100%	100%
1494	0	qsar-biodeg	1055	41	41	2	33%	66%	0%	7%	0%	0%
1515	0	micro-mass	571	1300	1300	20	1%	10%	0%	0%	17%	0%
1590	1	adult	48842	14	6	2	23%	76%	0%	0%	0%	0%
4134	0	Bioresponse	3751	1776	1776	2	45%	54%	0%	81%	1%	0%
4135	1	Amazon_employee_ac	32769	9	0	2	5%	94%	0%	NaN	NaN	NaN
4534	0	PhishingWebsites	11055	30	0	2	44%	55%	0%	NaN	NaN	NaN
4538	0	GesturePhaseSegmenta	9873	32	32	5	10%	29%	0%	0%	75%	0%

Table 1 (continued)

Id	in Gijsbers et al. (, 2019)	Name	Instances	Features	Numeric features	Classes	Min %	Maj %	% missing	% [0,1]	% $\mu = 0$	% $\sigma = 1$
4541	0	Diabetes130US	101766	49	13	3	11%	53%	0%	0%	0%	0%
23512	1	higgs	98050	28	28	2	47%	52%	0%	0%	7%	0%
23517	1	numerai28.6	96320	21	21	2	49%	50%	0%	100%	0%	0%
40498	0	wine-quality-white	4898	11	11	7	0%	44%	0%	0%	0%	0%
40668	1	connect-4	67557	42	0	3	9%	65%	0%	NaN	NaN	NaN
40670	0	dna	3186	180	0	3	24%	51%	0%	NaN	NaN	NaN
40685	1	shuttle	58000	9	9	7	0%	78%	0%	0%	0%	0%
40701	0	churn	5000	20	16	2	14%	85%	0%	0%	0%	0%
40900	0	Satellite	5100	36	36	2	1%	98%	0%	0%	0%	0%
40975	1	car	1728	6	0	4	3%	70%	0%	NaN	NaN	NaN
40978	0	Internet-Advertiseme	3279	1558	3	2	13%	86%	0%	0%	0%	0%
40981	1	Australian	690	14	6	2	44%	55%	0%	0%	0%	0%
40982	0	steel-plates-fault	1941	27	27	7	2%	34%	0%	11%	0%	0%
40983	0	wilt	4839	5	5	2	5%	94%	0%	0%	0%	0%
40984	1	segment	2310	19	19	7	14%	14%	0%	6%	0%	0%
40996	1	Fashion-MNIST	70000	784	784	10	10%	10%	0%	0%	0%	0%
41027	1	jungle_chess_2pcs	44819	6	6	3	9%	51%	0%	0%	0%	0%
41142	1	christine	5418	1636	1599	2	50%	50%	0%	0%	0%	0%
41143	1	jasmine	2984	144	8	2	50%	50%	0%	0%	0%	0%
41144	0	madeline	3140	259	259	2	49%	50%	0%	0%	0%	0%
41145	0	philippine	5832	308	308	2	50%	50%	0%	0%	3%	0%
41146	1	sylvine	5124	20	20	2	50%	50%	0%	0%	0%	0%
41150	1	MiniBooNE	130064	50	50	2	28%	71%	0%	0%	0%	0%
41156	0	ada	4147	48	48	2	24%	75%	0%	83%	8%	0%
41157	0	arcene	100	10000	10000	2	44%	56%	0%	0%	1%	0%

Table 1 (continued)

Id	in Gijsbers et al. (, 2019)	Name	Instances	Features	Numeric features	Classes	Min %	Maj %	% missing	% [0,1]	% $\mu = 0$	% $\sigma = 1$
41158	0	gina	3153	970	970	2	49%	50%	0%	0%	0%	0%
41159	1	guillermo	20000	4296	4296	2	40%	59%	0%	0%	0%	0%
41161	1	riccardo	20000	4296	4296	2	25%	75%	0%	0%	0%	0%
41163	1	dilbert	10000	2000	2000	5	19%	20%	0%	0%	0%	0%
41164	1	fabert	8237	800	800	7	6%	23%	0%	96%	4%	0%
41165	1	robert	10000	7200	7200	10	9%	10%	0%	0%	0%	0%
41166	1	volkert	58310	180	180	10	2%	21%	0%	15%	18%	0%
41167	1	dionis	416188	60	60	355	0%	0%	0%	0%	10%	0%
41168	1	jannis	83733	54	54	4	2%	46%	0%	4%	0%	0%
41169	1	helena	65196	27	27	100	0%	6%	0%	4%	0%	0%
42732	0	sf-police-incidents	2215023	9	3	2	12%	87%	0%	0%	0%	0%
42733	0	Click_prediction_s	39948	11	5	2	16%	83%	0%	0%	0%	0%
42734	0	okcupid-stem	50789	19	2	3	9%	71%	0%	0%	0%	0%

Appendix B: Considered algorithms

The following algorithms from the scikit-learn library were considered for the three pipeline slots (same setup for all optimizers). Please refer to https://github.com/fmohr/naive automl for the exact specification of the search space including the hyperparameter spaces.

Data-pre-processors

- Normalizer
- VarianceThreshold
- QuantileTransformer
- StandardScaler
- MinMaxScaler
- PowerTransformer
- RobustScaler

Feature-Pre-Processors

- FeatureAgglomeration
- PCA
- PolynomialFeatures
- Nystroem
- Selectquantile
- KernelPCA
- GenericUnivariateSelect
- RBFSampler
- FastICA

Classifiers

- SVC (once for each out of four kernels)
- KNeighborsClassifier
- QuadraticDiscriminantAnalysis
- RandomForestClassifier
- MultinomialNB
- LinearDiscriminantAnalysis
- ExtraTreesClassifier
- BernoulliNB
- MLPClassifier
- GradientBoostingClassifier
- GaussianNB
- DecisionTreeClassifier

Appendix C: Final result table

The following table shows the *mean* test score results of the approaches on the different datasets together with the standard deviation. Best performances are in bold, and entries that are not at least 0.1 worse (log-loss) than the best one or not statistically significantly different (according to a Wilcoxon signed rank test with $p = 0.05$) are underlined (Table 2).

Table 2 Avg. test Log-Loss after 1d

Id	rf	Random	Auto-sklearn	GAMA	Naive	Quasi-naive
3	0.05±0.0	**0.01±0.0**	**0.01±0.0**	**0.01±0.01**	0.02±0.03	**0.01±0.01**
12	0.26±0.06	0.11±0.05	0.11±0.04	**0.1±0.04**	2.75±4.11	0.48±0.72
23	1.45±0.24	0.9±0.06	**0.89±0.05**	0.9±0.03	1.0±0.06	0.92±0.06
31	0.56±0.12	**0.52±0.05**	0.53±0.04	0.54±0.03	0.57±0.04	0.57±0.08
54	0.49±0.06	4.19±6.41	**0.36±0.07**	0.42±0.09	0.99±0.37	0.67±0.46
181	1.5±0.22	**1.05±0.06**	1.06±0.06	1.14±0.16	1.22±0.24	1.13±0.12
1049	0.21±0.02	0.2±0.04	**0.19±0.04**	0.25±0.11	0.22±0.05	0.2±0.05
1067	0.56±0.17	0.39±0.08	**0.33±0.04**	**0.33±0.04**	0.36±0.06	0.35±0.04
1457	2.37±0.22	1.23±0.18	1.12±0.3	1.36±0.21	1.25±0.11	**0.87±0.22**
1461	0.62±0.06	0.3±0.01	**0.29±0.01**	0.3±0.01	0.31±0.02	0.3±0.01
1464	0.72±0.29	0.47±0.06	**0.46±0.05**	0.48±0.04	0.51±0.06	0.48±0.05
1468	0.32±0.05	0.15±0.08	0.16±0.1	0.17±0.09	0.18±0.07	**0.13±0.07**
1475	1.61±0.12	1.14±0.16	1.06±0.03	**1.05±0.03**	1.12±0.04	1.1±0.04
1485	0.61±0.01	0.51±0.89	**0.31±0.02**	0.36±0.11	0.71±0.32	0.4±0.03
1486	0.21±0.03	0.14±0.01	**0.13±0.01**	0.14±0.01	0.14±0.02	0.17±0.08
1487	0.17±0.05	0.16±0.04	**0.15±0.02**	0.18±0.07	0.21±0.06	**0.15±0.01**
1489	0.26±0.05	0.22±0.14	0.22±0.02	**0.21±0.02**	0.33±0.16	0.23±0.03
1494	**0.31±0.06**	0.33±0.13	0.32±0.14	0.35±0.09	0.58±0.44	0.34±0.13
1515	0.69±0.05	0.41±6.49	**0.38±0.1**	**0.38±0.15**	0.49±0.08	0.48±0.08
1590	0.55±0.04	**0.35±0.01**	**0.35±0.01**	**0.35±0.01**	0.39±0.04	**0.35±0.01**
4134	0.46±0.04	**0.43±0.07**	**0.43±0.02**	**0.43±0.07**	0.47±0.16	**0.43±0.02**
4135	0.34±0.04	**0.16±0.01**	**0.16±0.01**	0.17±0.01	**0.16±0.01**	**0.16±0.01**
4534	0.1±0.03	0.08±0.02	**0.07±0.01**	**0.07±0.01**	0.09±0.03	**0.07±0.02**
4538	0.92±0.02	0.98±0.36	0.84±0.03	**0.83±0.29**	0.89±0.05	0.88±0.03
4541	0.92±0.01	0.9±0.01	**0.89±0.01**	0.9±0.01	0.91±0.01	0.9±0.01
23512	0.57±0.01	0.62±0.08	**0.55±0.02**	**0.55±0.02**	0.59±0.05	**0.55±0.02**
23517	0.7±0.0	1.7±2.48	**0.69±0.0**	**0.69±0.0**	1.5±1.77	**0.69±0.0**
40498	0.94±0.11	1.21±0.32	0.76±0.06	1.09±0.55	0.89±0.31	**0.75±0.09**
40668	0.48±0.01	0.56±0.11	**0.39±0.05**	0.47±0.05	0.41±0.02	**0.39±0.01**
40670	0.25±0.02	**0.12±0.02**	**0.12±0.02**	0.19±0.04	0.26±0.09	0.13±0.02
40685	**0.0±0.0**	**0.0±0.01**	**0.0±0.0**	**0.0±0.0**	**0.0±0.0**	**0.0±0.0**
40701	0.24±0.09	**0.17±0.02**	**0.17±0.02**	**0.17±0.02**	0.24±0.03	**0.17±0.02**
40900	0.04±0.03	0.02±0.01	**0.01±0.0**	0.02±0.01	0.04±0.04	0.02±0.03
40975	0.16±0.03	**0.0±0.0**	0.01±0.01	**0.0±0.0**	0.22±0.19	0.11±0.16
40978	0.12±0.06	**0.09±0.03**	**0.09±0.02**	0.1±0.02	0.18±0.23	**0.09±0.03**
40981	0.35±0.06	0.4±0.19	**0.34±0.04**	0.35±0.07	0.39±0.06	**0.34±0.05**
40982	0.57±0.11	0.51±0.51	0.52±0.06	0.48±0.06	0.64±0.09	**0.47±0.07**
40983	0.06±0.04	0.34±0.39	**0.03±0.01**	0.04±0.01	0.41±0.65	0.07±0.06
40984	0.16±0.05	0.16±0.02	**0.15±0.03**	**0.15±0.02**	0.17±0.51	0.17±0.06
40996	0.38±0.01	0.44±0.04	**0.35±0.04**	0.41±0.07	0.45±0.07	0.38±0.01
41027	0.45±0.01	0.49±1.16	0.22±0.05	**0.21±0.06**	0.45±0.13	0.26±0.04
41142	0.55±0.01	0.58±0.15	**0.51±0.02**	0.52±0.02	0.66±0.07	0.52±0.02
41143	0.42±0.05	**0.41±0.02**	**0.41±0.02**	**0.41±0.02**	0.52±0.09	0.43±0.03
41144	0.54±0.01	0.34±0.12	**0.29±0.02**	0.34±0.02	0.62±0.1	0.37±0.03

Table 2 (continued)

Id	rf	Random	Auto-sklearn	GAMA	Naive	Quasi-naive
41145	0.51±0.02	0.44±0.03	**0.4±0.03**	0.41±0.02	0.57±0.02	0.46±0.02
41146	0.19±0.03	**0.14±0.01**	**0.14±0.01**	**0.14±0.02**	0.3±0.12	0.15±0.02
41150	0.18±0.01	0.28±0.21	**0.14±0.0**	0.25±0.2	0.16±0.01	**0.14±0.0**
41156	0.37±0.1	0.44±0.36	0.3±0.04	0.3±0.04	0.4±0.06	**0.29±0.04**
41157	0.5±0.07	0.48±0.35	0.46±0.98	0.37±0.12	0.48±0.86	**0.35±0.14**
41158	0.26±0.01	**0.15±0.05**	**0.15±0.03**	**0.15±0.04**	0.22±0.09	0.17±0.06
41159	0.43±0.01	0.52±0.12	**0.37±0.02**	0.51±0.11	0.42±0.1	0.38±0.01
41161	0.18±0.01	0.13±0.46	**0.02±0.03**	0.03±0.05	0.14±0.16	0.04±0.01
41163	0.33±0.01	**0.03±0.02**	0.05±0.03	0.08±0.49	0.05±0.01	0.05±0.01
41164	0.92±0.09	0.82±0.03	**0.79±0.06**	0.84±0.03	0.85±0.04	0.84±0.04
41165	1.73±0.02	**1.71±0.19**	1.75±0.08	1.8±0.07	1.73±0.11	1.73±0.04
41166	1.05±0.02	1.23±0.32	0.92±0.06	**0.91±0.07**	1.09±0.05	1.03±0.03
41167	**0.83±0.01**	1.66±0.56	1.83±0.3	5.6±10.21	2.8±0.13	2.32±0.03
41168	0.77±0.01	0.7±0.01	**0.68±0.02**	0.69±0.01	0.79±0.01	0.74±0.02
41169	6.14±0.12	2.99±0.45	**2.55±0.09**	2.76±0.07	2.83±0.14	2.64±0.02
42732	0.59±0.0	**0.36±0.0**	**0.36±0.0**	**0.36±0.0**	**0.36±0.0**	**0.36±0.0**
42733	2.53±0.14	0.44±0.01	**0.43±0.01**	0.44±0.01	0.48±0.11	0.45±0.01
42734	0.67±0.02	0.62±0.02	**0.6±0.01**	0.61±0.02	0.62±0.05	**0.6±0.01**

Best results in bold, comparable results are underlined

Appendix D: Slot analysis in detail

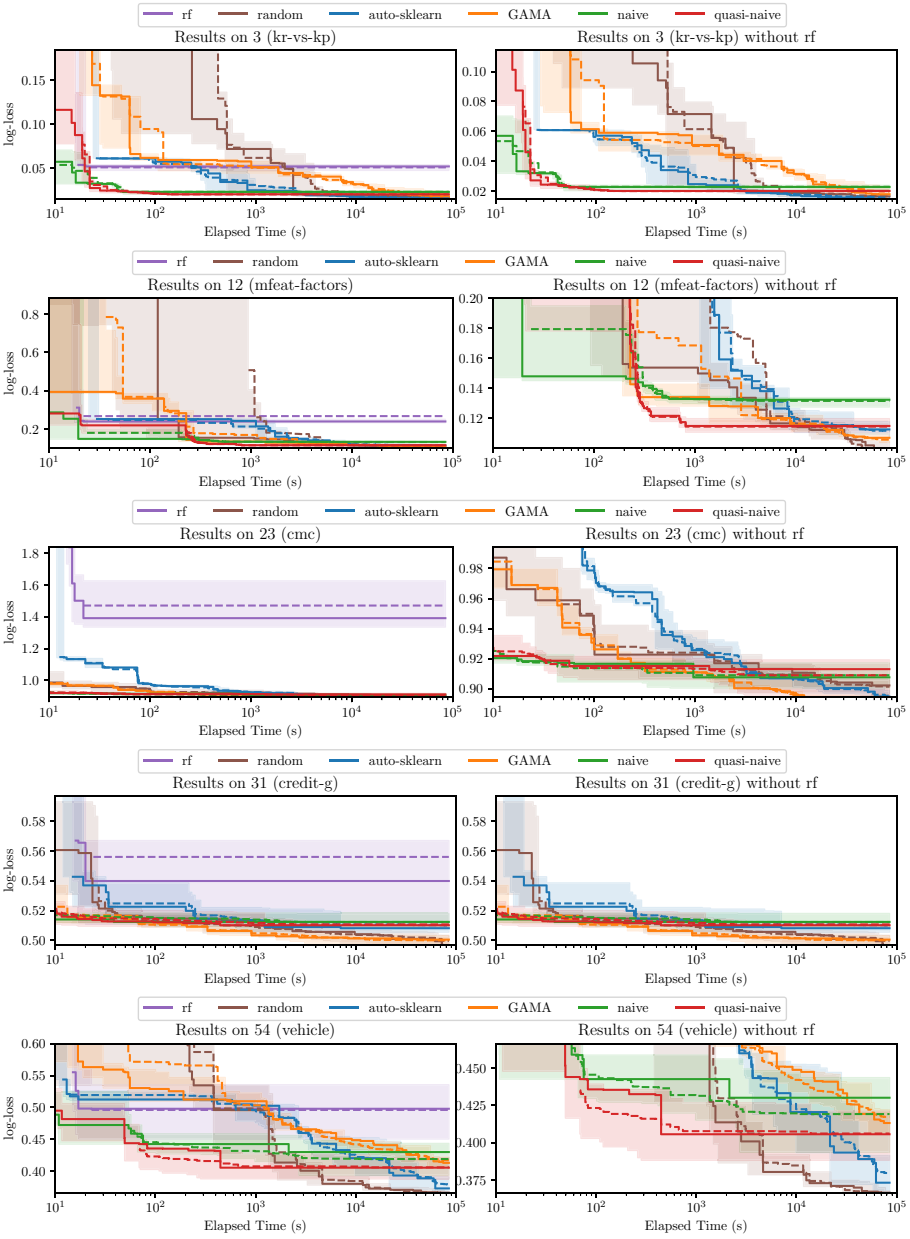
This table details Fig. 4 and shows in which fraction of the cases, Quasi-Naive AutoML chose a component for each of the slots that occurs in an 0.03-optimal pipeline.

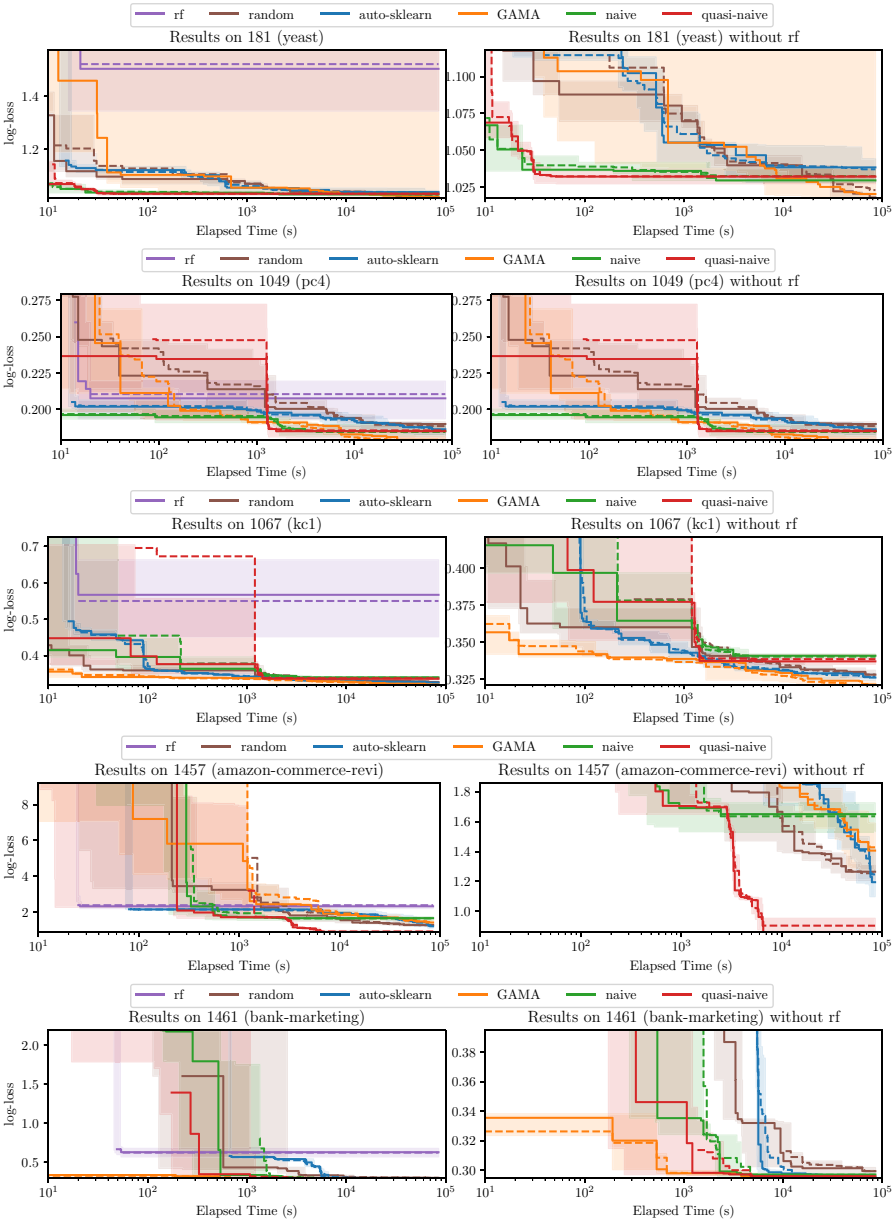
Openmlid	Classifier	Data-pre-processor	Feature-pre-pro-cessor
3	1.00	1.00	1.00
12	1.00	1.00	1.00
23	1.00	1.00	1.00
31	1.00	1.00	1.00
54	1.00	0.70	0.60
181	0.90	0.60	0.80
1049	1.00	1.00	1.00
1067	1.00	1.00	1.00
1457	0.00	0.70	0.50
1461	1.00	1.00	1.00
1464	1.00	1.00	1.00
1468	1.00	1.00	1.00
1475	1.00	1.00	1.00
1485	1.00	1.00	1.00
1486	0.89	1.00	1.00

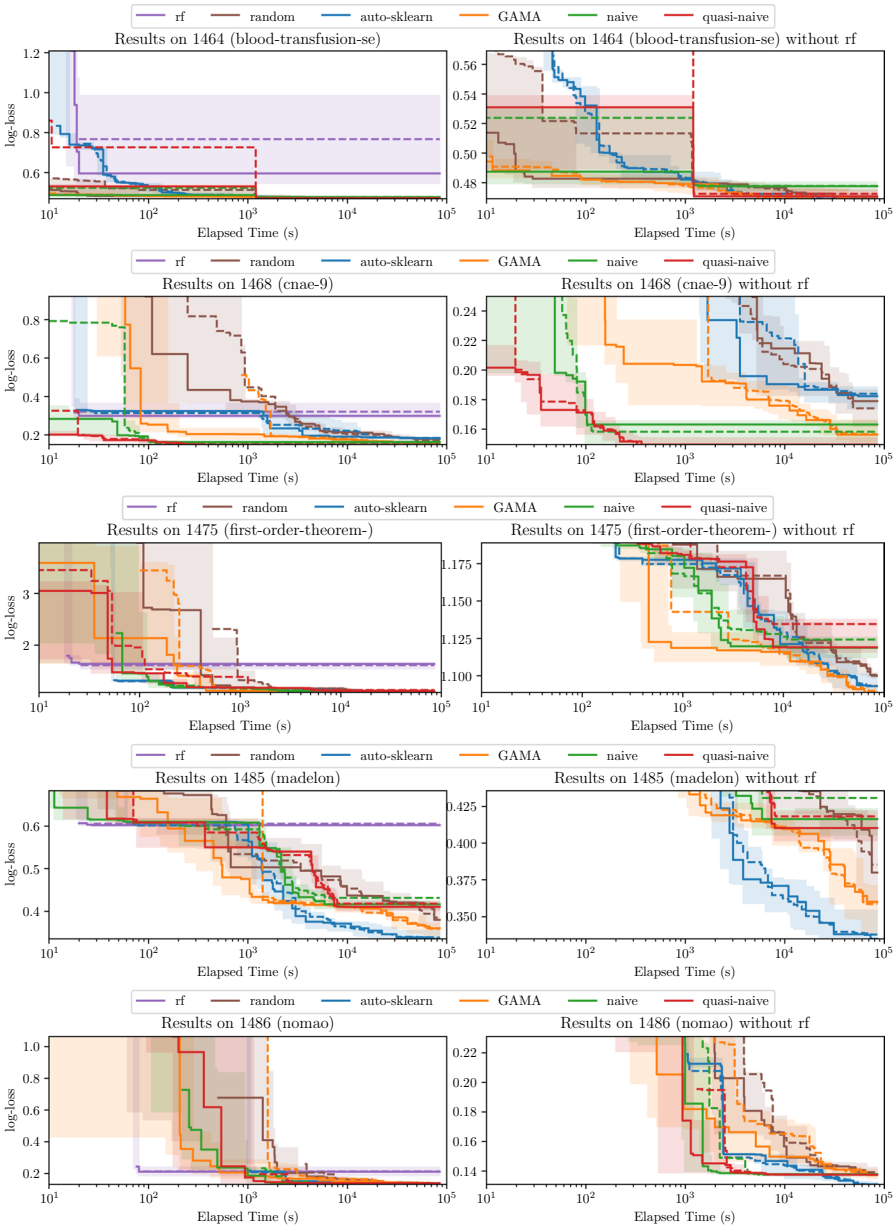
Openmlid	Classifier	Data-pre-processor	Feature-pre-pro-cessor
1487	1.00	1.00	1.00
1489	1.00	1.00	1.00
1494	1.00	1.00	1.00
1515	1.00	1.00	1.00
1590	1.00	1.00	1.00
4134	1.00	1.00	1.00
4135	1.00	1.00	1.00
4534	1.00	1.00	1.00
4538	1.00	1.00	1.00
4541	0.89	1.00	1.00
23512	1.00	1.00	1.00
23517	1.00	1.00	1.00
40498	1.00	1.00	1.00
40668	0.00	0.90	0.00
40670	1.00	1.00	1.00
40685	1.00	1.00	1.00
40701	1.00	1.00	1.00
40900	1.00	1.00	1.00
40975	1.00	1.00	1.00
40978	1.00	1.00	1.00
40981	1.00	1.00	1.00
40982	0.80	0.70	0.90
40983	1.00	1.00	1.00
40984	1.00	1.00	1.00
40996	0.00	0.14	0.14
41027	1.00	0.57	0.71
41142	1.00	1.00	1.00
41143	1.00	1.00	1.00
41144	1.00	1.00	1.00
41145	1.00	1.00	1.00
41146	1.00	1.00	1.00
41156	1.00	1.00	1.00
41157	0.30	0.10	0.80
41159	0.00	1.00	0.11
41161	0.00	1.00	0.00
41163	1.00	1.00	1.00
41164	1.00	1.00	1.00
41165	0.40	0.60	0.40
41166	1.00	1.00	1.00
41167	1.00	1.00	1.00
41169	0.00	0.67	0.56
42732	1.00	1.00	1.00
42733	0.00	1.00	0.00
42734	1.00	1.00	1.00

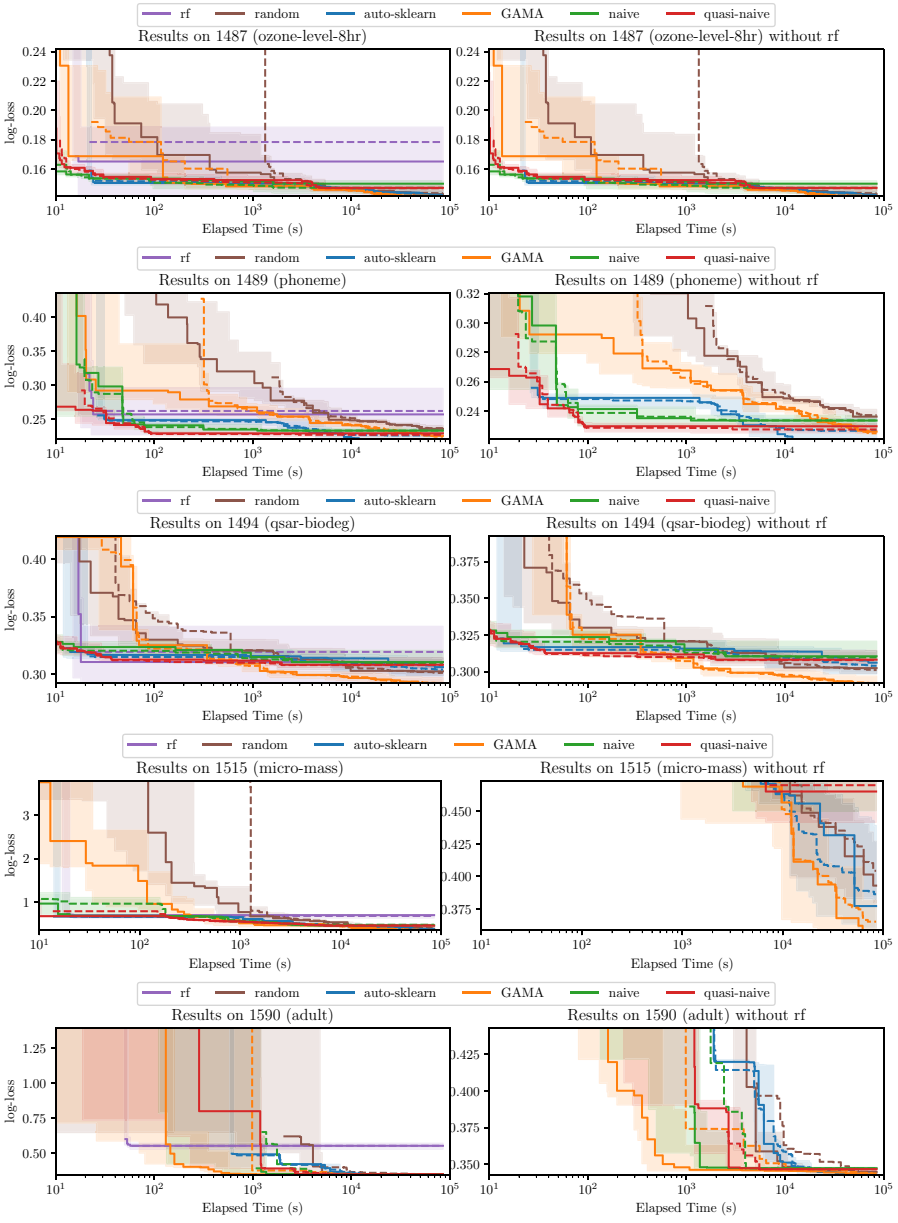
Appendix E: Performance plots over time

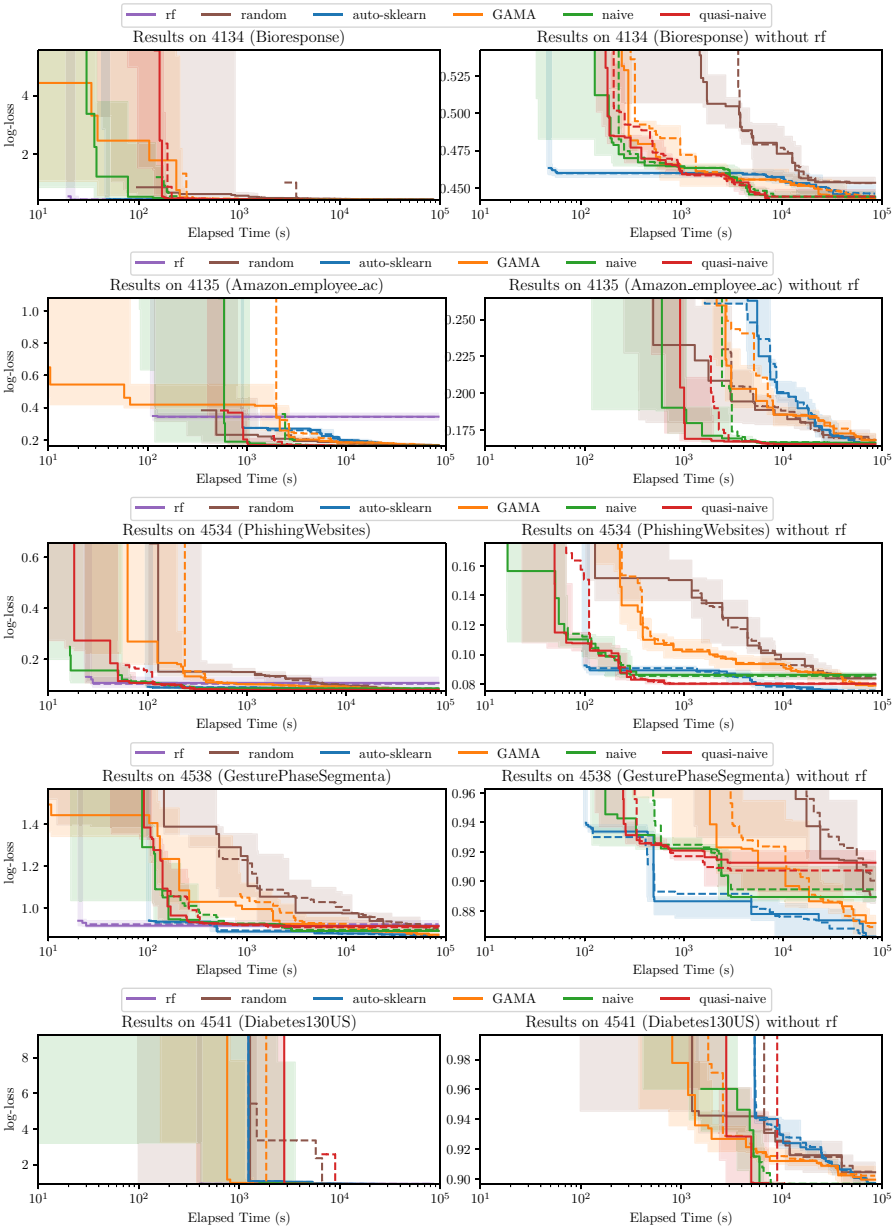
The figures in this section show the log-loss of the best pipeline an approach has identified at any point of time. Since the Random Forest is sometimes substantially outperformed or since substantial optimization occurs over times, it can be difficult to recognize details in late parts of the curve. Therefore, the right plots show results without Random Forests and with a scaling that only assures the visibility of all the observations occurring after 4h elapsed time.

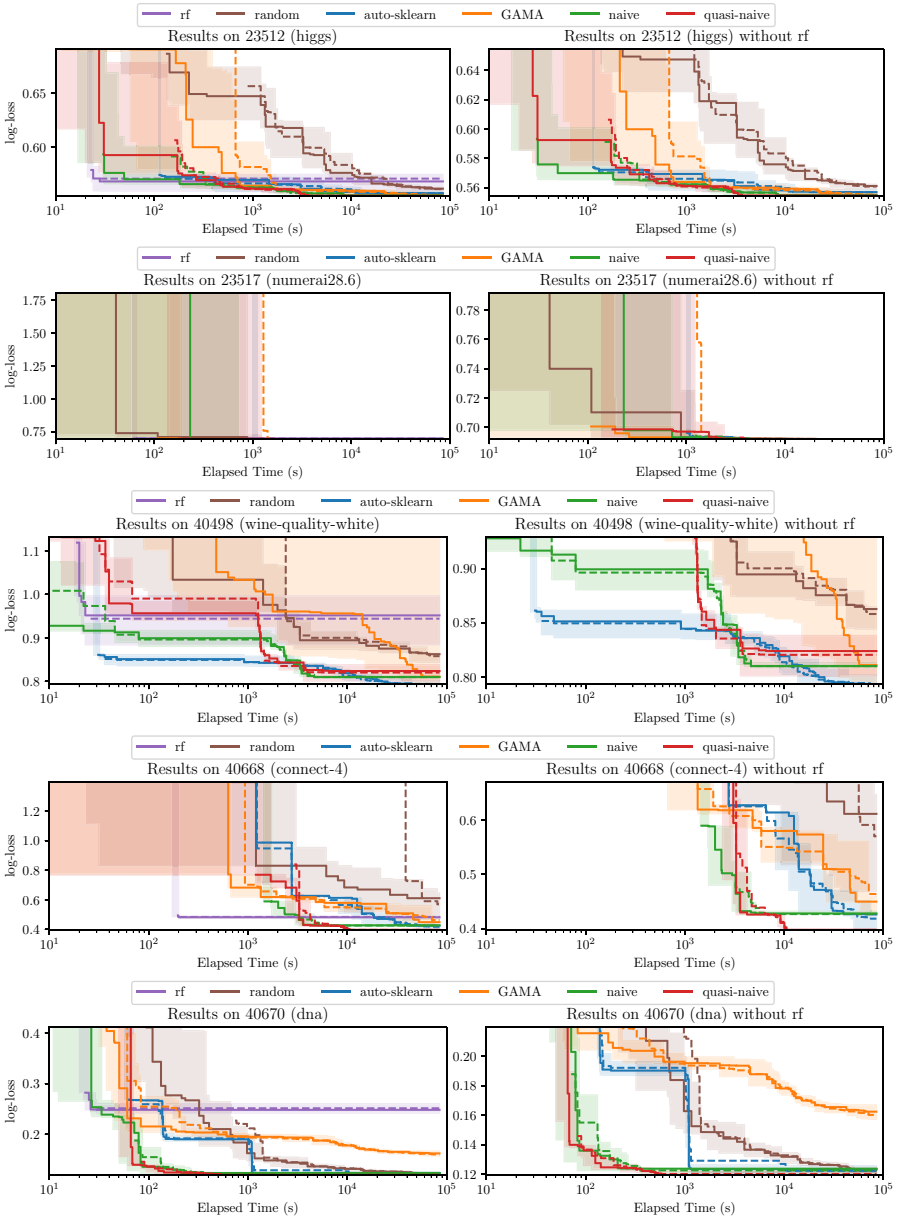


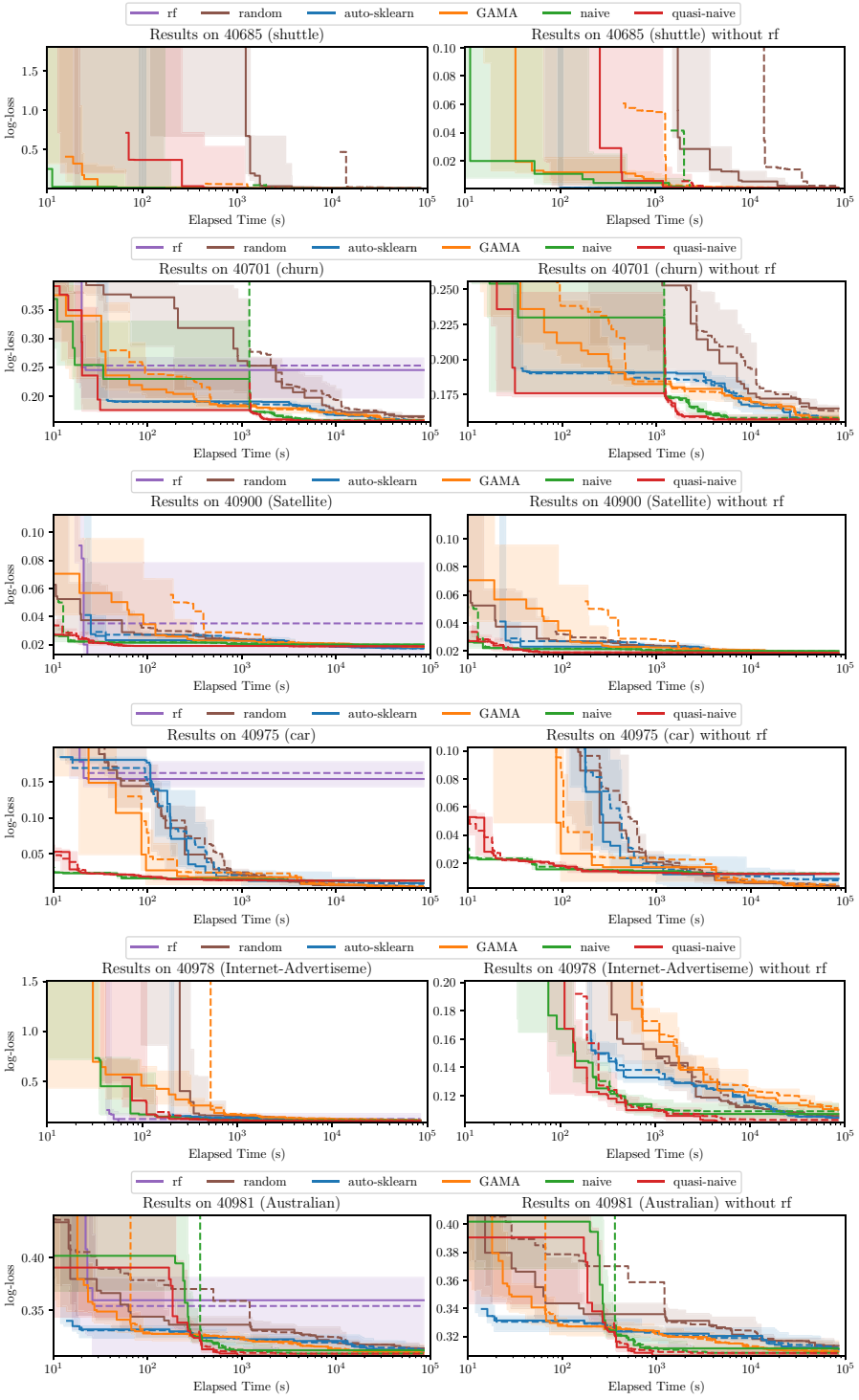

rf
random
auto-sklearn
GAMA
naive
quasi-naive
Results on 181 (yeast)
Results on 181 (yeast) without rf
Results on 1049 (pc4)
Results on 1049 (pc4) without rf
Results on 1067 (kc1)
Results on 1067 (kc1) without rf
Results on 1457 (amazon-commerce-revi)
Results on 1457 (amazon-commerce-revi) without rf
Results on 1461 (bank-marketing)
Results on 1461 (bank-marketing) without rf
log-loss
Elapsed Time (s)

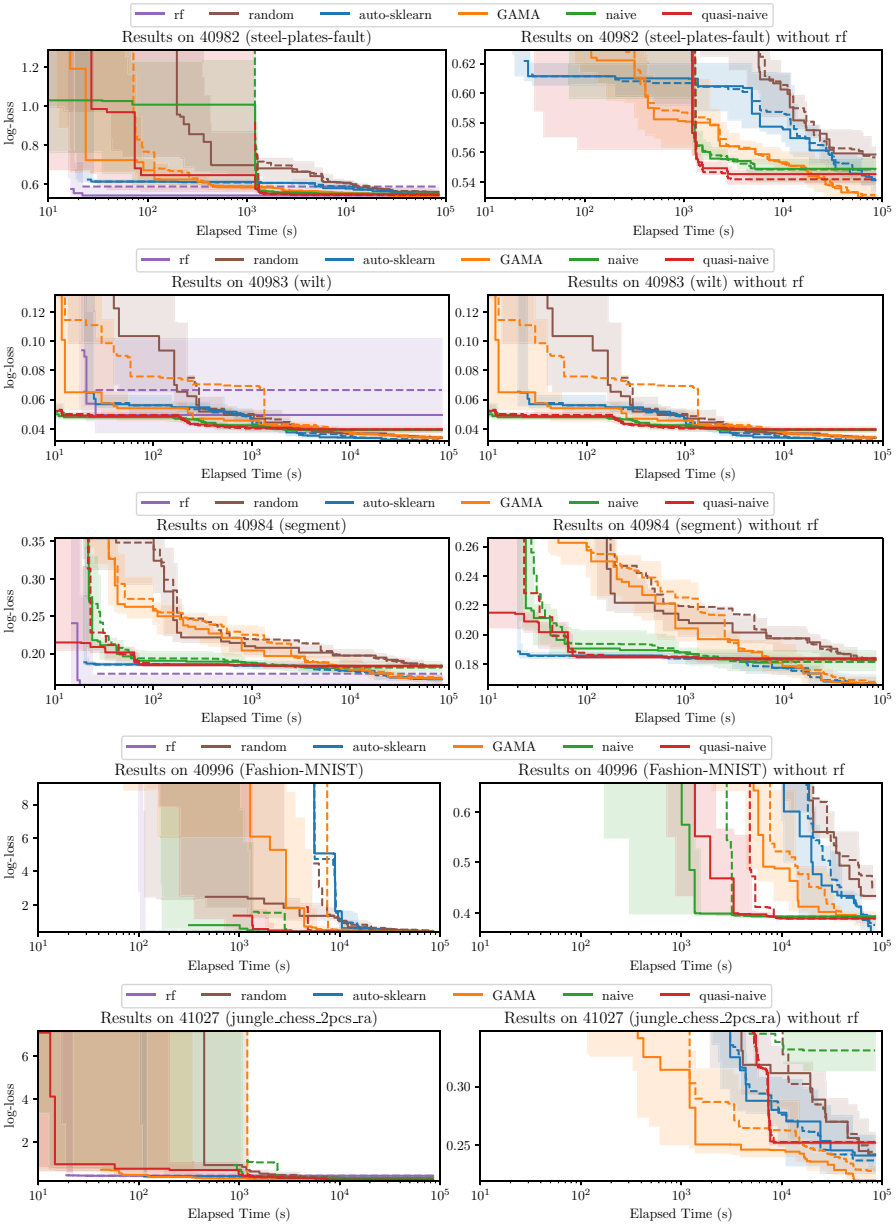

rf
random
auto-sklearn
GAMA
naive
quasi-naive
Results on 1464 (blood-transfusion-se)
Results on 1464 (blood-transfusion-se) without rf
Results on 1468 (cnae-9)
Results on 1468 (cnae-9) without rf
Results on 1475 (first-order-theorem-)
Results on 1475 (first-order-theorem-) without rf
Results on 1485 (madelon)
Results on 1485 (madelon) without rf
Results on 1486 (nomao)
Results on 1486 (nomao) without rf
log-loss
Elapsed Time (s)

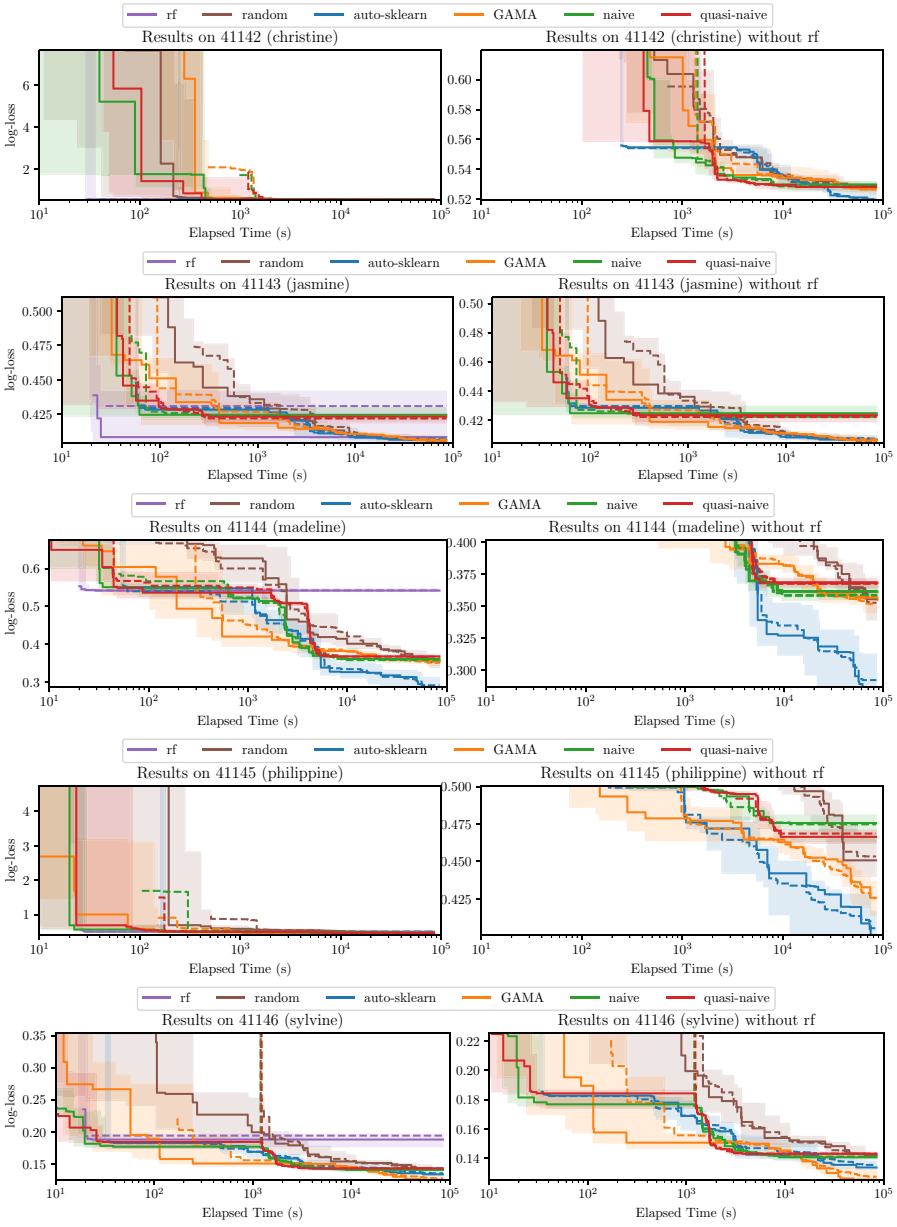

rf
random
auto-sklearn
GAMA
naive
quasi-naive
Results on 1487 (ozone-level-8hr)
Results on 1487 (ozone-level-8hr) without rf
Results on 1489 (phoneme)
Results on 1489 (phoneme) without rf
Results on 1494 (qsar-biodeg)
Results on 1494 (qsar-biodeg) without rf
Results on 1515 (micro-mass)
Results on 1515 (micro-mass) without rf
Results on 1590 (adult)
Results on 1590 (adult) without rf
log-loss
Elapsed Time (s)

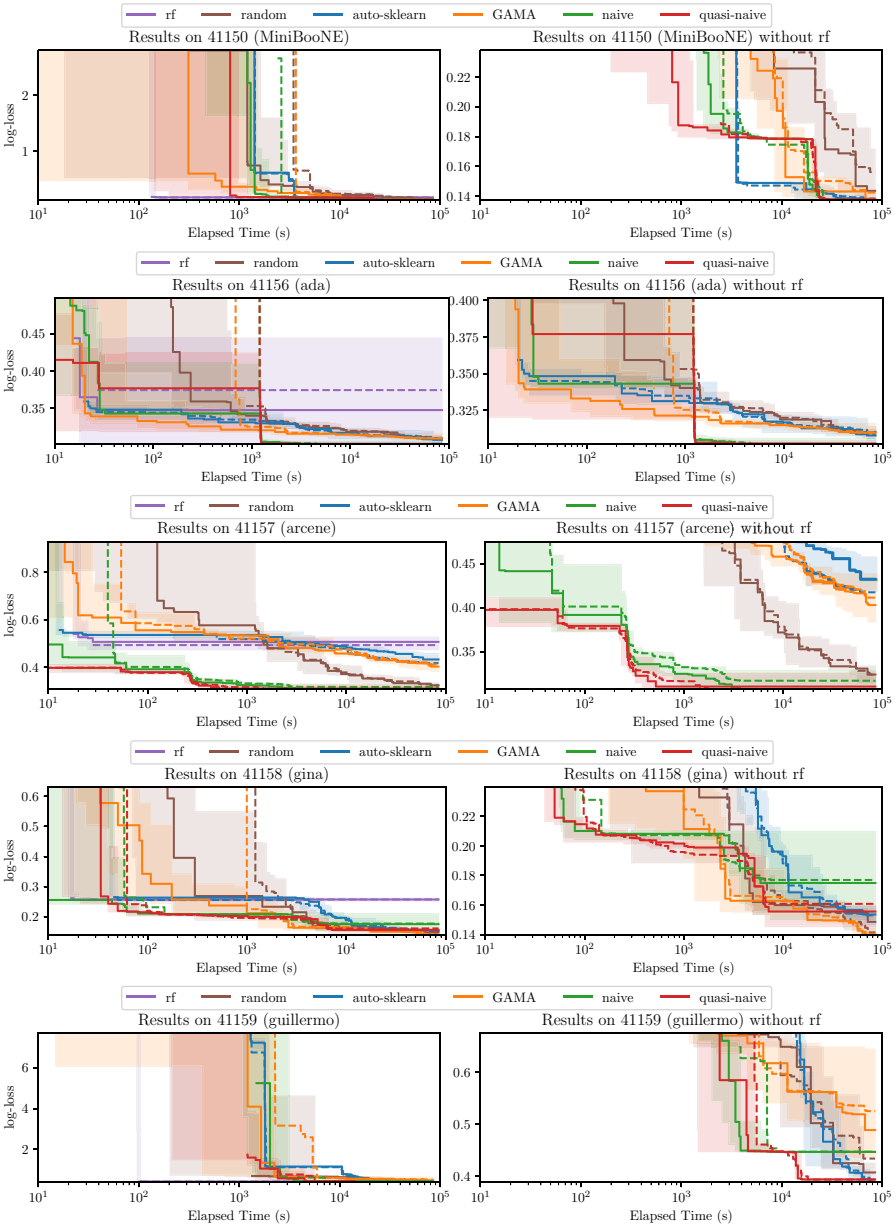

rf
random
auto-sklearn
GAMA
naive
quasi-naive
Results on 4134 (Bioresponse)
Results on 4134 (Bioresponse) without rf
Results on 4135 (Amazon_employee_ac)
Results on 4135 (Amazon_employee_ac) without rf
Results on 4534 (PhishingWebsites)
Results on 4534 (PhishingWebsites) without rf
Results on 4538 (GesturePhaseSegmenta)
Results on 4538 (GesturePhaseSegmenta) without rf
Results on 4541 (Diabetes130US)
Results on 4541 (Diabetes130US) without rf
log-loss
Elapsed Time (s)

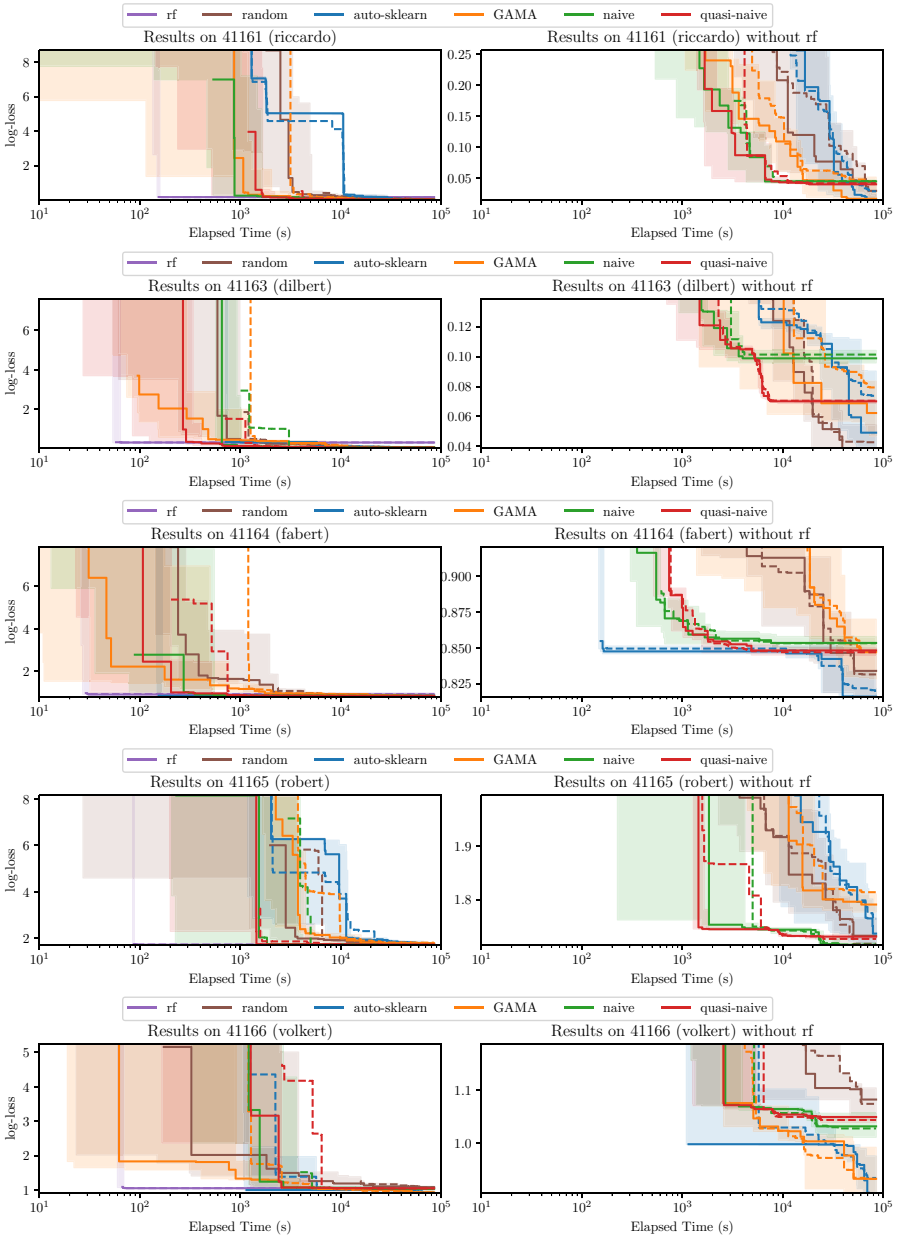

rf
random
auto-sklearn
GAMA
naive
quasi-naive
Results on 23512 (higgs)
Results on 23512 (higgs) without rf
Results on 23517 (numerai28.6)
Results on 23517 (numerai28.6) without rf
Results on 40498 (wine-quality-white)
Results on 40498 (wine-quality-white) without rf
Results on 40668 (connect-4)
Results on 40668 (connect-4) without rf
Results on 40670 (dna)
Results on 40670 (dna) without rf
log-loss
Elapsed Time (s)

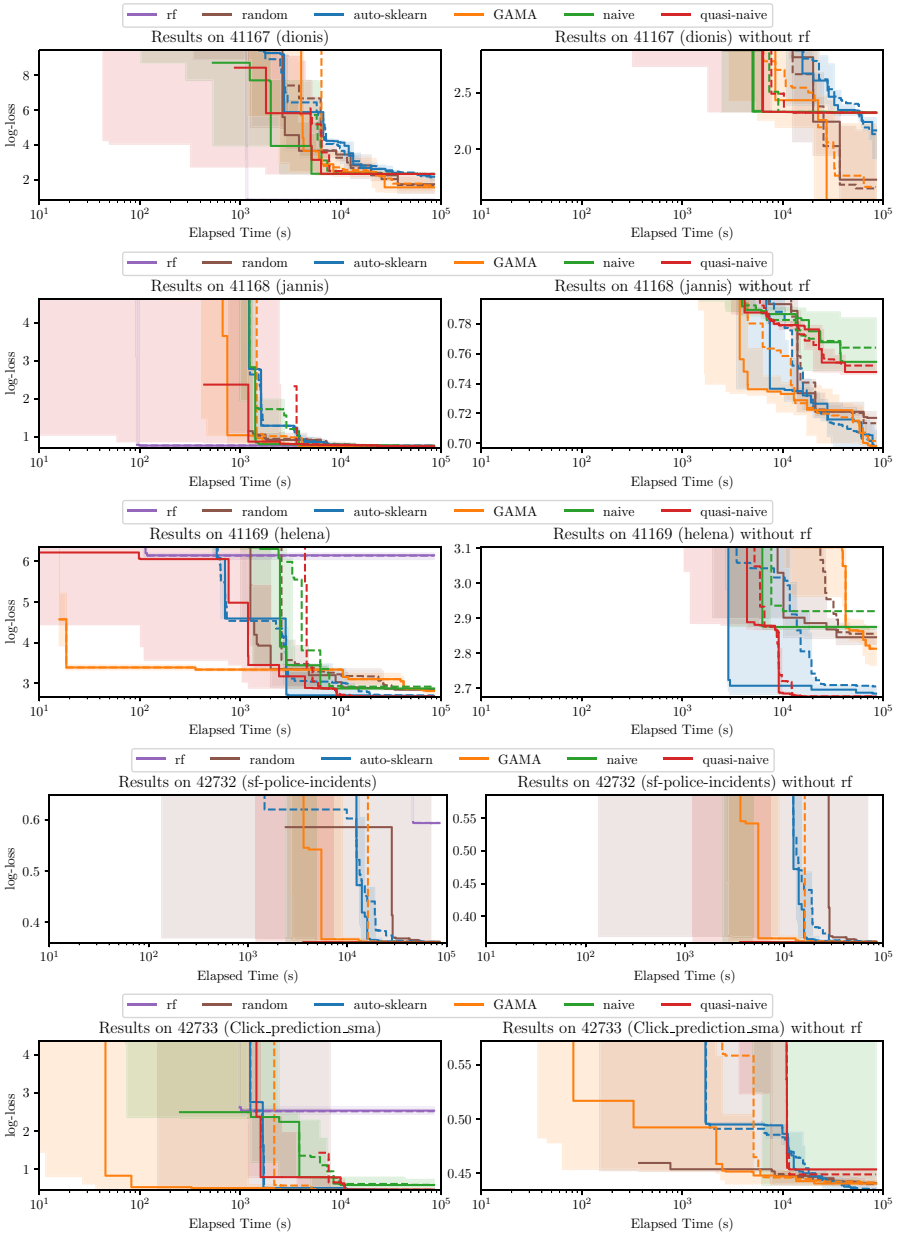

rf
random
auto-sklearn
GAMA
naive
quasi-naive
Results on 40685 (shuttle)
Results on 40685 (shuttle) without rf
Results on 40701 (churn)
Results on 40701 (churn) without rf
Results on 40900 (Satellite)
Results on 40900 (Satellite) without rf
Results on 40975 (car)
Results on 40975 (car) without rf
Results on 40978 (Internet-Advertiseme)
Results on 40978 (Internet-Advertiseme) without rf
Results on 40981 (Australian)
Results on 40981 (Australian) without rf
log-loss
Elapsed Time (s)

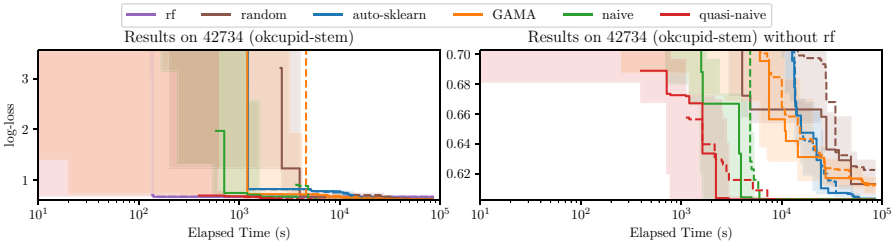

rf
random
auto-sklearn
GAMA
naive
quasi-naive
Results on 40982 (steel-plates-fault)
Results on 40982 (steel-plates-fault) without rf
Results on 40983 (wilt)
Results on 40983 (wilt) without rf
Results on 40984 (segment)
Results on 40984 (segment) without rf
Results on 40996 (Fashion-MNIST)
Results on 40996 (Fashion-MNIST) without rf
Results on 41027 (jungle_chess_2pcs_ra)
Results on 41027 (jungle_chess_2pcs_ra) without rf
log-loss
Elapsed Time (s)


rf
random
auto-sklearn
GAMA
naive
quasi-naive
Results on 41142 (christine)
Results on 41142 (christine) without rf
Results on 41143 (jasmine)
Results on 41143 (jasmine) without rf
Results on 41144 (madeline)
Results on 41144 (madeline) without rf
Results on 41145 (philippine)
Results on 41145 (philippine) without rf
Results on 41146 (sylvine)
Results on 41146 (sylvine) without rf
log-loss
Elapsed Time (s)


rf
random
auto-sklearn
GAMA
naive
quasi-naive
Results on 41150 (MiniBooNE)
Results on 41150 (MiniBooNE) without rf
Results on 41156 (ada)
Results on 41156 (ada) without rf
Results on 41157 (arcene)
Results on 41157 (arcene) without rf
Results on 41158 (gina)
Results on 41158 (gina) without rf
Results on 41159 (guillermo)
Results on 41159 (guillermo) without rf
log-loss
Elapsed Time (s)


rf
random
auto-sklearn
GAMA
naive
quasi-naive
Results on 41161 (riccardo)
Results on 41161 (riccardo) without rf
Results on 41163 (dilbert)
Results on 41163 (dilbert) without rf
Results on 41164 (fabert)
Results on 41164 (fabert) without rf
Results on 41165 (robert)
Results on 41165 (robert) without rf
Results on 41166 (volkert)
Results on 41166 (volkert) without rf
log-loss
Elapsed Time (s)


rf
random
auto-sklearn
GAMA
naive
quasi-naive
Results on 41167 (dionis)
Results on 41167 (dionis) without rf
Results on 41168 (jannis)
Results on 41168 (jannis) without rf
Results on 41169 (helena)
Results on 41169 (helena) without rf
Results on 42732 (sf-police-incidents)
Results on 42732 (sf-police-incidents) without rf
Results on 42733 (Click_prediction_sma)
Results on 42733 (Click_prediction_sma) without rf
log-loss
Elapsed Time (s)



Acknowledgements We thank Matthias Feurer and Pieter Gijsbers for their remarkable support in adjusting auto-sklearn and GAMA for our evaluations. We also thank the anonymous reviewers who considerably helped us improve this manuscript and its contribution. Finally, the authors gratefully acknowledge support of this project by the Paderborn Center for Parallel Computing (PC^2), which provided the computational resources and computing time to run our experiments. This work was supported by the CAPSAB Research Group at Universidad de La Sabana and the German Research Foundation (DFG) within the Collaborative Research Center "On-The-Fly Computing" (SFB 901).

Author contributions Felix Mohr is the main author of both paper and implementation. Marcel Wever contributed in the manuscript revision as well as the resolution of technical aspects of the evaluation.

Funding Open Access funding provided by Colombia Consortium. This work was supported by the CAPSAB Research Group at Universidad de La Sabana and the German Research Foundation (DFG) within the Collaborative Research Center "On-The-Fly Computing" (SFB 901)

Data availability https://github.com/fmohr/naiveautoml

Declarations

Conflict of interest Eyke Hüllermeier

Ethics approval Not applicable.

Consent to participate Not applicable

References

Boyd, S. P., Parikh, N., Chu, E., Peleato, B., & Eckstein, J. (2011). Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning, 3*(1), 1–122.

Breiman, L. (1996). Bagging predictors. *Machine Learning, 24*(2), 123–140.
Breiman, L. (2001). Random forests. *Machine Learning, 45*(1), 5–32.
Cachada, M., Abdulrahman, S.M., & Brazdil, P. (2017) Combining feature and algorithm hyperparameter selection using some metalearning methods. In *Proceedings of the international workshop on AutoML@PKDD/ECML 2017* (pp. 69–83)
Chen, B., Wu, H., Mo, W., Chattopadhyay, I., & Lipson, H. (2018). Autostacker: A compositional evolutionary learning system. In *Proceedings of the genetic and evolutionary computation conference* (pp. 402–409)
Crisan, A., & Fiore-Gartland, B. (2021). Fits and starts: Enterprise use of automl and the role of humans in the loop. *CoRR* abs/2101.04296.
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.
Deb, K., Pratap, A., Agarwal, S., & Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation, 6*(2), 182–197.
Drozdal, J., Weisz, J.D., Wang, D., Dass, G., Yao, B., Zhao, C., Muller, M.J., Ju, L., & Su, H. (2020). Trust in AutoML: exploring information needs for establishing trust in automated machine learning systems. In *IUI '20: 25th International conference on intelligent user interfaces* (pp. 297–307). ACM
Engels, R. (1996). Planning tasks for knowledge discovery in databases; performing task-oriented user-guidance. In *Proceedings of the second international conference on knowledge discovery and data mining (KDD-96)* (pp 170–175). AAAI Press.
Erickson, N., Mueller, J., Shirkov, A., Zhang, H., Larroy, P., Li, M., & Smola, A. J. (2020). AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data. *CoRR* abs/2003.06505.
Escalante, H. J., Montes-y-Gómez, M., & Sucar, L. E. (2009). Particle swarm model selection. *Journal of Machine Learning Research, 10*, 405–440.
Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., & Hutter, F. (2015). Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems* (pp. 2962–2970).
Fusi, N., Sheth, R., & Elibol, M. (2018). Probabilistic matrix factorization for automated machine learning. In: Advances in Neural Information Processing Systems (pp. 3352–3361).
Gijsbers, P., LeDell, E., Thomas, J., Poirier, S., Bischl, B., & Vanschoren, J. (2019). An open source automl benchmark. *CoRR* abs/1907.00909.
Gijsbers, P., & Vanschoren, J. (2019). GAMA: genetic automated machine learning assistant. *Journal of Open Source Software, 4*(33), 1132.
Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I.H. (2009). The WEKA data mining software: an update. ACM SIGKDD Explorations *11*
Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2011). *Sequential model-based optimization for general algorithm configuration, 6683*, 507–523.
Jamieson, K., & Talwalkar, A. (2016). Non-stochastic best arm identification and hyperparameter optimization. In *Artificial Intelligence and Statistics, AISTATS'16* (pp. 240–248).
Kandasamy, K., Vysyaraju, K. R., Neiswanger, W., Paria, B., Collins, C. R., Schneider, J., et al. (2020). Tuning hyperparameters without grad students: Scalable and robust Bayesian optimisation with dragonfly. *Journal of Machine Learning Research, 21,* 81:1-81:27.
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).
Kishimoto, A., Bouneffouf, D., Marinescu, R., Ram, P., Rawat, A., Wistuba, M., Palmes, P.P., & Botea, A. (2021). Bandit limited discrepancy search and application to machine learning pipeline optimization. In *8th ICML workshop on automated machine learning (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. *Journal of Machine Learning Research, 18*(1), 826–830.
Li, L., Jamieson, K. G., DeSalvo, G., Rostamizadeh, A., & Talwalkar, A. (2017). Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research, 18,* 185:1-185:52.

Lindauer, M., Eggensperger, K., Feurer, M., Biedenkapp, A., Marben, J., Müller, P., & Hutter, F. (2019). BOAH: A tool suite for multi-fidelity Bayesian optimization & analysis of hyperparameters. CoRR abs/1908.06756.

Liu, S., Ram, P., Vijaykeerthy, D., Bouneffouf, D., Bramble, G., Samulowitz, H., et al. (2020). An ADMM based framework for AutoML pipeline configuration. *Proceedings of the AAAI Conference on Artificial Intelligence, 34,* 4892–4899.

Mohr, F., & Wever, M. (2021). Replacing the ex-def Baseline in AutoML by Naive AutoML. In: *8th ICML workshop on automated machine learning (AutoML)*.

Mohr, F., Wever, M., & Hüllermeier, E. (2018). ML-Plan: Automated machine learning via hierarchical planning. *Machine Learning, 107*(8), 1495–1515.

Mohr, F., Wever, M., Tornede, A., & Hüllermeier, E. (2021). Predicting machine learning pipeline runtimes in the context of automated machine learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 43,* 1–1.

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. (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. (2019). TPOT: A tree-based pipeline optimization tool for automating machine learning. In *Automated machine learning: Methods, systems, challenges, The Springer series on challenges in machine learning* (pp. 151–160). Springer

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine learning in python. *Journal of Machine Learning Research, 12*, 2825–2830.

Rakotoarison, H., Schoenauer, M., & Sebag, M. (2019). Automated machine learning with monte-carlo tree search. In *Proceedings of the twenty-eighth international joint conference on artificial intelligence* (pp. 3296–3303). https://www.ijcai.org/.

Statnikov, A. R., Tsamardinos, I., Dosbayev, Y., & Aliferis, C. F. (2005). GEMS: A system for automated cancer diagnosis and biomarker discovery from microarray gene expression data. *International Journal of Medical Informatics, 74*(7–8), 491–503.

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* (pp. 847–855).

Vanschoren, J. (2019). Meta-learning. In *Automated machine learning - methods, systems, challenges, The Springer series on challenges in machine learning* (pp. 35–61). Springer.

Vanschoren, J., van Rijn, J. N., Bischl, B., & Torgo, L. (2013). OpenML: Networked science in machine learning. *SIGKDD Explorations, 15*(2), 49–60.

Wolpert, D. H. (1992). Stacked generalization. *Neural Networks, 5*, 241–259.

Yang, C., Akimoto, Y., Kim, D.W., & Udell, M. (2019). OBOE: Collaborative filtering for AutoML model selection. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (pp. 1173–1183).