AMLB: an AutoML Benchmark

Pieter Gijsbers[1]   P.GIJSBERS@TUE.NL

Marcos L. P. Bueno[1,4]   MARCOS.DEPAULABUENO@DONDERS.RU.NL

Stefan Coors[2]   STEFAN.COORS@STAT.UNI-MUENCHEN.DE

Erin LeDell[3]   ERIN@H2O.AI

Sébastien Poirier[3]   SEBASTIEN@H2O.AI

Janek Thomas[2]   JANEK.THOMAS@STAT.UNI-MUENCHEN.DE

Bernd Bischl[2]   BERND.BISCHL@STAT.UNI-MUENCHEN.DE

Joaquin Vanschoren[1]   J.VANSCHOREN@TUE.NL

[1] EINDHOVEN UNIVERSITY OF TECHNOLOGY, EINDHOVEN, THE NETHERLANDS
[2] LUDWIG MAXIMILIAN UNIVERSITY OF MUNICH, MUNICH, GERMANY
[3] H2O.AI, MOUNTAIN VIEW, CA, UNITED STATES
[4] RADBOUD UNIVERSITY, NIJMEGEN, THE NETHERLANDS

Editor: Marc Schoenauer

Abstract

Comparing different AutoML frameworks is notoriously challenging and often done incorrectly. We introduce an open and extensible benchmark that follows best practices and avoids common mistakes when comparing AutoML frameworks. We conduct a thorough comparison of 9 well-known AutoML frameworks across 71 classification and 33 regression tasks. The differences between the AutoML frameworks are explored with a multi-faceted analysis, evaluating model accuracy, its trade-offs with inference time, and framework failures. We also use Bradley-Terry trees to discover subsets of tasks where the relative AutoML framework rankings differ. The benchmark comes with an open-source tool that integrates with many AutoML frameworks and automates the empirical evaluation process end-to-end: from framework installation and resource allocation to in-depth evaluation. The benchmark uses public data sets, can be easily extended with other AutoML frameworks and tasks, and has a website with up-to-date results.

Keywords: open source, benchmark, automated machine learning, automl

1. Introduction

To create useful machine learning (ML) models from data, data scientists must prepare the data for consumption by ML algorithms (for example, coercing it to a different format or by encoding categorical features), select an ML algorithm, and tune its hyperparameters. This requires extensive expertise, such as knowing which hyperparameters to tune and how (Probst et al., 2019; Weerts et al., 2020). Even with this knowledge, it is a time-

consuming task, since the best choices are unique to each data set and can be interdependent on each other (Van Rijn and Hutter, 2018).

The field of automated machine learning (AutoML) is focused on automating the design and optimization of ML pipelines in a data-driven way (Hutter et al., 2019). Neural Architecture Search (NAS) is an important part of AutoML that automates the design decisions of deep neural networks. AutoML aims to free up valuable time for experts to perform other tasks and allow novice users to train well-performing ML models.

Many different AutoML approaches have been proposed, including sequential model-based optimization (Hutter et al., 2011; Snoek et al., 2012), hierarchical task planning (Erol et al., 1994), and genetic programming (Koza, 1992). Novel systems are being developed in both academia and industry, and a recent survey by Van der Blom et al. (2021) showed that 69% of 307 practitioners (at least partially) adopt automated model selection and hyperparameter configuration.

1.1 The Need for Standardized Benchmarking

With considerable effort being spent on developing and improving AutoML frameworks as well as increased usage by practitioners, there is a need for systematic and in-depth comparisons of the the various approaches to track progress in the field. However, the comparison of AutoML frameworks is prone to several types of error. First, selection bias may be introduced, even accidentally, when authors decide which data sets to use in their evaluation. For example, too few data sets may be selected to accurately evaluate the framework's strengths and weaknesses, and the chosen data sets may no longer be challenging for current AutoML frameworks. Without a standard suite of data sets to use for evaluation, the selection of data sets is often not reasonably justified and motivated. Moreover, issues may arise from errors in the installation, configuration, or use of 'competitor' frameworks. Typical examples are misunderstanding memory management and/or using insufficient compute resources (Balaji and Allen, 2018), or failing to use comparable resource budgets (Ferreira et al., 2021).

Several suites of benchmark data sets have been used in the aforementioned papers, but none have become a standard in the AutoML community. The original selection of data sets by Thornton et al. (2013) was used in several earlier papers (Feurer et al., 2015a; Mohr et al., 2018), but fails to highlight differences between current AutoML frameworks. Most published AutoML papers use a self-selected suit of data sets on which their methods are evaluated. For example, Drori et al. (2018), Rakotoarison et al. (2019) and Gil et al. (2018) were all published at the same time but feature different suites on which they evaluate their contributions. This inconsistency makes it impossible to directly compare results across papers and track progress over time. It also potentially allows for presenting cherry-picked results.

1.2 Our Contributions

We introduce a novel AutoML benchmark following best practices to avoid these common pitfalls while stimulating progress towards more standardized benchmarking.[1] To ensure

---

1. A first look of this tool was presented at the ICML 2019 AutoML Workshop (Gijsbers et al., 2019). The current version is significantly more general, more systematic, and allows much more in-depth analysis.

reproducibility[2], we provide an open-source benchmarking tool[3] that allows easy integration with AutoML frameworks, and performs end-to-end evaluations thereof on carefully curated sets of open data sets. Our focus is on tabular data. Unstructured data are out of scope for this paper, since they are best tackled with NAS, and benchmarking NAS frameworks imposes different practical constraints, as discussed in Section 2.2. We also restrict our evaluations to open-source AutoML frameworks.

Our benchmarking tool, dubbed AMLB (for AutoML Benchmark), can be used to perform evaluations of AutoML frameworks in a fully automated way. The AutoML frameworks are integrated with the AMLB in direct agreement and jointly with the framework's developers to ensure correctness. We carry out a large-scale evaluation of 9 well-known AutoML frameworks, some of which with multiple configurations, across 71 classification and 33 regression tasks. To better understand how these systems perform across many tasks, we also introduce techniques for detailed comparison of AutoML frameworks, including final model accuracy, inference time trade-offs, and failure analysis. Finally, we provide an interactive visualization tool that may be used for further exploration of all our results or to reproduce the analyses performed in this paper.

In the remainder of this paper, we first discuss related work in Section 2 and cover several key open-source AutoML frameworks in Section 3. Next, we provide an overview of our proposed benchmarking tool in Section 4 and motivate our benchmark design choices in Section 5. The results obtained by running this benchmark are analyzed in Section 6. Finally, Section 7 summarizes our main conclusions and sketches directions for future work.

2. Related Literature

In this section, we motivate why we need benchmarks specifically designed for AutoML, review other work evaluating AutoML frameworks, and finally discuss the relevant ML benchmarking literature. Several benchmark suites have been developed in ML (Van Gestel et al., 2004; Olson et al., 2017; Wu et al., 2018; Bischl et al., 2021; Fischer et al., 2023). The data sets in these suites often do not include problematic data characteristics found in real world tasks (for example, many missing values), as many ML algorithms are not natively able to handle them. By contrast, in order to be applicable to a wide range of data, AutoML frameworks should be designed to handle these problematic data sets. As such, there is opportunity to allow more such problematic data sets in ML benchmarks and study how well AutoML frameworks handle these issues. Moreover, runtime budgets are crucial in an AutoML benchmark, as most AutoML frameworks are designed to run until a given time budget is exhausted. These runtime budgets are often not specified beforehand in traditional ML benchmarks, since algorithms must usually run to completion.[4] Consequently, new benchmarks that have been designed specifically for AutoML frameworks are needed.

---

2. Using the ACM definition: an independent group can obtain the same result using the author's own artifacts (https://www.acm.org/publications/policies/artifact-review-and-badging-current).
3. Code, results, and documentation at: https://openml.github.io/automlbenchmark/
4. One exception are performance studies, such as Kotthaus et al., 2015

2.1 Evaluation of Automated Machine Learning Frameworks

To establish new best practices for AutoML benchmarking, it is beneficial to study the shortcomings encountered in prior benchmarks as well as lessons learned. Balaji and Allen (2018) conducted one of the first benchmark studies on AutoML frameworks. They evaluated four open-source frameworks on both classification and regression tasks sourced from OpenML (Vanschoren et al., 2014), optimized for weighted F1 score and mean squared error, respectively. Unfortunately, they encountered technical issues with most AutoML frameworks that led to a questionable experimental evaluation. For example, H2O AUTOML (LeDell and Poirier, 2020) was configured to optimize to a different metric (log loss as opposed to weighted F1 score) and ran with a different setup (unlike the others, H2O AUTOML was not containerized), and AUTO_ML (Parry, 2018) had its hyperparameter optimization (HPO) disabled, making for incomparable results. This highlights the need for careful configuration of all AutoML frameworks involved.

A study by Ferreira et al. (2021) evaluated the effectiveness of AutoML frameworks for protein abundance prediction. This evaluation compared H2O AUTOML with a 6 hour limit to GAMA with a one hour limit, and TPOT with a limit defined by the hyperparameter configuration of its evolutionary optimization (1000 generations with a population size of 250). While the comparison was not the main contribution of the paper, the AutoML frameworks were directly compared against each other, and these very different budgets were not motivated.

A study on nearly 300 data sets across six different frameworks was conducted by Truong et al. (2019). Each experiment consisted of a single 80/20 holdout split on a 15-minute training time budget, which was chosen so that most tools returned a result on at least 70% of the data sets. We postulate it is reasonable to assume that the data sets for which no result is returned by a framework are most often those data sets for which optimization is hard. For example, a large data set might cause one framework to conduct only few evaluations while it completely halts another. Unfortunately, this makes the resulting comparisons uninterpretable, as a framework could seemingly demonstrate better performance simply because it failed to return models on data sets for which optimization was difficult. Hence, it is key that failures must be avoided as much as possible, and any remaining failures should be analysed and taken into account in subsequent analysis.

On the positive side, Truong et al. (2019) present their results across different subsets of the benchmark—for example, few versus many categorical features—which helps to highlight differences between different frameworks. The authors also conduct small-scale experiments to analyze performance over time by running the tools on multiple time budgets on a subset of data sets as well as the 'robustness', which denotes the variance in final performance given the same input data. Unfortunately, both experiments were conducted on only one data set per sub-category, which does not enable generalizing the results. Still, studying the impact of data characteristics and budget sizes should ideally be part of AutoML benchmark design.

Zöller and Huber (2021) present a survey on AutoML and combined algorithm selection and HPO frameworks (CASH, Thornton et al., 2013). Six CASH and five AutoML frameworks are compared across 137 classification tasks, with the former limited to 325 iterations and the latter constrained to a one hour time budget. Among the CASH frameworks, hyperopt (Bergstra et al., 2013) performed best, although absolute differences were small

between all optimizers. The AutoML frameworks are compared as they are, which might reflect common use. However, by not controlling their settings, it becomes impossible to draw conclusions about the effectiveness of individual parts of AutoML systems. A number of errors of AutoML frameworks are observed, including memory constraint violations, segmentation faults, and Java server crashes. The authors also find that most frameworks construct rather modest pipelines (with few preprocessing operators). As such, it is recommended that AutoML frameworks are controlled carefully, use a wide operator search space, and are evaluated on datasets that require non-trivial preprocessing.

Kaggle[5], a platform for data science competitions, is sometimes used to compare AutoML frameworks to human data scientists (Zöller and Huber, 2021; Erickson et al., 2020). Zöller and Huber (2021) found that the rankings of AutoML frameworks on benchmark datasets are very different from the rankings on competitions. Furthermore, humans still find better solutions than the examined AutoML frameworks. However, it is hard to interpret such results. Submissions on Kaggle leaderboards range from serious attempts to random test runs, and a significant portion of them do not outperform a simple baseline. Finally, most Kaggle results are several years old and possibly no longer represent state-of-the-art human-made models because of advances in the available hardware or algorithms.

A recent AutoML benchmark for multi-label classification (Wever et al., 2021) proposed a general tool with a configurable search space and optimizer, which allows for the inclusion of new methods and ablation studies. Unfortunately, this approach requires that existing AutoML frameworks be re-implemented within this tool, which is difficult in such a rapidly developing field. As such, we need a simpler way to include existing and new AutoML frameworks into benchmarking tools while still allowing control over their configuration.

In addition to AutoML benchmarks, a series of competitions for tabular AutoML was hosted (Guyon et al., 2019). The first two competitions focused on tabular AutoML, where data is assumed to be independent and identically distributed. In these competitions, participants submitted code that automatically builds a model on given data and produced predictions for a test set. During the development phase, competitors could make use of a public leaderboard and several validation data sets. After the development phase, the latest submissions of each participant would be evaluated on a set of new data sets to determine the final ranking. These data sets consisted of a mix of both new data and data taken from public repositories, although they were reformatted to conceal their identity. In their analysis Guyon et al. (2019) reveal that most methods failed to return results on at least some data sets due to practical issues, such as running out of memory.

2.2 Other (Benchmark) Literature

Instead of benchmarking whole (Auto-) ML frameworks, it is often useful to focus on various sub-parts and optimize them step-by-step. One part of such a system is HPO, and its available algorithms are as numerous and diverse as the learning algorithms themselves. Consequently, there exist various benchmarking suites to foster research by comparing those optimization algorithms.

Like non-black-box optimization, black box optimizers are typically evaluated on synthetic test functions or on real-world tasks. A well-established benchmarking suite for con-

---

5. https://www.kaggle.com/

tinuous optimization is COCO (Hansen et al., 2021), which includes a collection of various synthetic black-box benchmark functions. Nevergrad[6] is a popular platform for gradient-free optimization with benchmarking functionality. *kurobako* (Ohta and Yamazaki, 2022) provides various general black-box optimizers and benchmark problems, while *LassoBench* (Šehić et al., 2021) is suitable for benchmarking high-dimensional optimization problems. Especially for Bayesian Optimization, *Bayesmark* (Turner, 2022) combines several benchmarks on real-world tasks.

One of the first benchmarks for empirically evaluating HPO algorithms was *HPOlib* (Eggensperger et al., 2013), which allows accessing real-world HPO tasks, tabular and surrogate benchmarks, and synthetic test functions using a common API. This benchmark was used by Bergstra et al. (2014) in their empirical benchmark studies. *HPOBench* (Eggensperger et al., 2021) is a similar successor of *HPOlib* that concentrates on reproducible containerized benchmarks and multi-fidelity optimization problems. Based on OpenML (Vanschoren et al., 2014), Arango et al. (2021) recently introduced *HPO-B*, a large-scale reproducible benchmark for transfer-HPO methods. In contrast, *PROFET* (Klein et al., 2019) uses a generative meta-model to generate synthetic but realistic benchmark instances.

A related benchmark is *NAS-Bench-101* (Ying et al., 2019), which is a tabular data set that maps convolutional neural network architectures to their trained and evaluated performance on CIFAR-10. Its goal is to make NAS more accessible, despite the tremendous demand of computational resources. Additionally, there exists a series of NAS-Bench systems that includes *NAS-Bench-1shot1* (Zela et al., 2020), *NAS-Bench-301* (Siems et al., 2020), and others.

3. AutoML Frameworks

Automated ML pipeline design was first explored by Escalante et al. (2009), but the first prominent AutoML framework was AUTO-WEKA (Thornton et al., 2013). AUTO-WEKA used Bayesian optimization to select and tune the algorithms in an ML pipeline based on WEKA (Hall et al., 2009). Since then, a plethora of new AutoML frameworks have been developed, either by iteratively improving on old designs or using novel approaches. In this section, we will discuss the AutoML frameworks we considered for our benchmark.

Unfortunately, the cost of evaluating all frameworks is prohibitive, so we selected only 9 of them to evaluate in this work. Only open source frameworks were considered. From those, we made selections to cover a variety of different optimization approaches. We considered frameworks developed by industry as well as academia and included packages whose authors proactively integrated their AutoML framework, so that we are confident that they are integrated correctly.

As our experiments are run on machines without GPU, we focus on AutoML frameworks which are not significantly affected by the lack thereof. For example, AUTO-KERAS (Jin et al., 2019), and AUTOPYTORCH (Zimmer et al., 2021) gain large benefits from having a GPU available, thus reporting an evaluation with only CPU available might give a pessimistic performance estimation compared to typical use of those frameworks. However, even for some of our selected frameworks, such as H2O AUTOML and AUTOGLUON, specific base models support GPU acceleration when it is available.

---

6. https://facebookresearch.github.io/nevergrad/benchmarking.html

Framework	Optimization and Search Space	Reference
AUTOGLUON	Stacked ensembles of preset pipelines	Erickson et al. (2020)
AUTO-SKLEARN	BO of SCIKIT-LEARN pipelines	Feurer et al. (2015a)
AUTO-SKLEARN 2	BO of iterative algorithms	Feurer et al. (2020)
FLAML	CFO of iterative algorithms	Wang et al. (2021)
GAMA	EO of SCIKIT-LEARN pipelines	Gijsbers and Vanschoren (2021)
H2O AUTOML	Iterative mix of RS and ensembling	LeDell and Poirier (2020)
LIGHTAUTOML	BO of linear models and GBM	Vakhrushev et al. (2021)
MLJAR	Custom data science pipeline	Płońska and Płoński (2021)
NAIVEAUTOML	Custom data science pipeline	Mohr and Wever (2023)
TPOT	EO of SCIKIT-LEARN pipelines	Olson and Moore (2016)

Table 1: Used AutoML frameworks in the experiments. Different optimization techniques such as Bayesian optimization (BO), evolutionary optimization (EO), random search (RS) or cost frugal optimization (CFO) are used across different search spaces.

The most notable omission is AUTO-WEKA, which we decided to exclude based on the performance in our 2019 evaluation and its lack of updates since Gijsbers et al. (2019). Some frameworks integrated with AMLB are not evaluated in this paper. The authors of AUTOXGBOOST (Thomas et al., 2018) opted out of an evaluation because it is built on deprecated software with no plans for updates. ML-PLAN (Mohr et al., 2018), MLR3AUTOML[7], and MLNET[8] are excluded because we encountered significant technical problems evaluating the systems. We hope to publish results from these frameworks at a later date. There are still many AutoML frameworks not yet integrated with the AutoML benchmark, and we hope that we can work together with AutoML developers to add additional integrations in the future.

3.1 Integrated Frameworks

Table 1 offers an overview of the AutoML frameworks evaluated in this paper alongside a simplified description of their optimization and search space design, which are elaborated below. We refer the interested reader to the original publications and our website, which has links to the frameworks' source code and documentation. NAIVEAUTOML was also evaluated but excluded from the analysis in Section 6, as explained in Appendix E.

3.1.1 AutoGluon-Tabular

AUTOGLUON automates ML across a variety of tasks, including image, text, and tabular data. The subsystem that automates ML on tabular data is called AUTOGLUON-TABULAR (Erickson et al., 2020), but we will refer to it as AUTOGLUON. In contrast to other AutoML systems discussed here, AUTOGLUON does not perform a pipeline search or hyperparameter tuning.

---

7. Source and documentation of MLR3AUTOML at https://github.com/a-hanf/mlr3automl/.
8. Source and documentation of MLNET at https://github.com/dotnet/machinelearning/.

Instead, it has a predetermined set of models that are combined through multi-layer stacking and ensembling.

AUTOGLUON's ensemble consists of three layers. The first layer consists of models from a range of model families trained directly on the data. In the second layer, the same type of models are considered but rather as a stacking learner trained with both the input data and the predictions of the first layer. In the final layer, the predictions of the second-layer models are combined into an ensemble, using an ensemble method (Caruana et al., 2004) first used in AutoML by AUTO-SKLEARN (Feurer et al., 2015a).

To adhere to time constraints, AUTOGLUON may stop iterative algorithms prematurely or forgo training certain models altogether. Given more time, AUTOGLUON will train additional models using the same algorithms and hyperparameter configurations on different data splits, which further improves the generalization of the stacking layer.

AUTOGLUON offers many different presets that affect the trade-off between final model performance and inference time. In this paper, we evaluated three presets: best quality, high quality, and high quality with an inference time limit (denoted as B, HQ, and HQIL, respectively). These respective presets produce increasingly faster models at the cost of decreased model accuracy.

3.1.2 AUTO-SKLEARN

Based on the design of AUTO-WEKA, AUTO-SKLEARN (Feurer et al., 2015a) also uses Bayesian optimization but is instead implemented in Python and optimizes pipelines built with SCIKIT-LEARN (Pedregosa et al., 2011). Additionally, it warm-starts optimization through meta-learning, starting pipeline search with the best pipelines for the most similar data sets (Feurer et al., 2015b). After pipeline search has concluded, an ensemble is created from pipelines trained during search using the procedure described by Caruana et al. (2004, 2006). AUTO-SKLEARN has won two AutoML challenges (Guyon et al., 2019), although for both entries, AUTO-SKLEARN was customized for the competition, and not all changes are found in the public releases (Feurer et al., 2018).

Based on experience from the challenges, AUTO-SKLEARN 2 was subsequently developed (Feurer et al., 2020). The most notable changes include reducing the search space to only iterative learning algorithms and excluding most preprocessing, use of successive halving (Jamieson and Talwalkar, 2016), adaptive evaluation strategies, and replacing the data-specific warm-start strategy with a data-agnostic portfolio of pipelines. Because these changes make version 2.0 almost entirely different from 1.0, and 1.0 has been updated since our last evaluation, we evaluate both auto-sklearn versions in this paper. However, AUTO-SKLEARN 2 does not yet support regression, and its heavy use of meta-learning made it impossible for us to perform a 'clean' evaluation at this time (see Section 5.3.3).

3.1.3 FLAML

The Fast and Lightweight AutoML Library (FLAML, Wang et al., 2021) optimizes boosting frameworks (XGBOOST, Chen and Guestrin, 2016, CATBOOST, Prokhorenkova et al., 2018, and LIGHTGBM, Ke et al., 2017) and a small selection of SCIKIT-LEARN algorithms through a multi-fidelity randomized directed search called Cost-Frugal Optimization (Wu et al., 2021). This search is based on an expected cost for improvement, which tracks the expected compu-

tational cost of improving over the best model found so far for each learner. Only after choosing which learner to tune, hyperparameter optimization proceeds by a randomized directed search, sampling a new configuration from a unit sphere around the previous sample point. After evaluating its validation performance, the next sample point is moved to that direction (if better) or the opposite direction (if worse). FLAML positions itself as a fast AutoML framework that is designed to work for small time budgets (Wang et al., 2021).

3.1.4 GAMA

Designed as a modular AutoML framework for researchers, GAMA's search method and post processing are easily configurable and extensible (Gijsbers and Vanschoren, 2021). By default, GAMA uses genetic programming to optimize linear ML pipelines with an arbitrary amount of preprocessing algorithms. Similar to TPOT, GAMA's evolutionary algorithm uses NSGA-II to perform multi-objective optimization (Deb et al., 2002), maximizing performance while minimizing the number of components in the pipeline. By contrast, GAMA's evolutionary algorithm is asynchronous and does not work with distinct generations, which allows for higher resource utilization. The final model is constructed through ensemble selection (Caruana et al., 2004, 2006), similar to AUTO-SKLEARN.

3.1.5 H2O AutoML

Built on the H2O distributed machine learning platform (H2O.ai, 2013), H2O AUTOML (LeDell and Poirier, 2020) evaluates a portfolio of bespoke algorithm configurations and also performs random searches over the majority of the supervised learning algorithms offered in H2O. Early stopping is applied to the searches for efficiency. The amount of time the H2O AUTOML spends searching each algorithm is pre-specified by the authors, though this can be customized. This allocates the amount of optimization work done at each step of the algorithm, and some stronger algorithms (e.g. XGBoost) are favored, or given more time, over others (e.g. GLM). To further increase model performance, H2O AUTOML also trains two types of stacked ensemble models at various stages during the run: an ensemble using all available models at step s, and an ensemble with only the best models of each algorithm type at step s.

H2O AUTOML aims to cover a large search space quickly and relies on stacking to boost model performance. Additionally, H2O AUTOML uses a predefined strategy for imputation, normalization, and categorical encoding for each algorithm and does not currently optimize over preprocessing pipelines by default, but preprocessing can be turned on as an option. The H2O AUTOML algorithm is designed to generate models that are very fast at inference time, rather than strictly focusing on maximizing model accuracy, with the goal of balancing these two competing factors to produce practical models suited for production environments.

3.1.6 LightAutoML

LIGHTAUTOML is specifically designed with applications in the financial services industry in mind (Vakhrushev et al., 2021). In this framework, pipelines are designed for quick inference and interpretability. Only linear models and boosting frameworks (CATBOOST and LIGHTGBM) are considered. Expert rules are used to evaluate likely good hyperparameter configurations and design the search space. Tree-structured Parzen Estimators (Bergstra et al., 2011) are

used to optimize hyperparameters of the boosting frameworks, while warm-starting and early stopping are used to optimize linear models with grid search. Different models are combined in either a weighted voting ensemble (binary classification and regression) or with two levels of stacking (multi-class classification). In a special 'compete' mode for larger time budgets, the AutoML pipeline is run multiple times with different configurations and their resulting models are ensembled with weighted voting, which allows for a more robust model.

3.1.7 MLJAR

MLJAR (Płońska and Płoński, 2021) starts its search with a set of predetermined models and a limited random search—similar to H2O AUTOML. This is followed by a feature creation and selection step, after which a hill climbing algorithm is used to further tune the best pipelines. After search, the models can be stacked, used in a voting ensemble, or both. Learners from SCIKIT-LEARN are considered, as well as boosting packages XGBOOST, CATBOOST, and LIGHTGBM. MLJAR features multiple modes for different use cases, including exploratory data analysis or finding a fast model, a 'perform' mode aimed at finding a good model with good inference time, and a 'compete' mode which aims to find the best possible model. We evaluate both the 'compete' and 'perform' presets in this work.

3.1.8 Naive AutoML

Proposed as a baseline by Mohr and Wever (2023), it mimics a simple workflow a data scientist might execute. First, a base learner is chosen by evaluating their performance using their default hyperparameter configuration. Then several steps are performed in order, including feature scaling, feature selection, and hyperparameter tuning. Unfortunately, we encountered issues evaluating NAIVEAUTOML which prohibited us to include its results in Section 6. Appendix E motivates this choice, and shows the results we obtained for NAIVEAUTOML.

3.1.9 TPOT

Tree-based Pipeline Optimization Tool (Olson and Moore, 2016), or TPOT, optimizes pipelines using genetic programming (e.g., McKay et al. (2010)): ML pipelines can be expressed as trees where different branches represent distinct preprocessing pipelines. These pipelines are then optimized through evolutionary optimization. To reduce overfitting that may arise from the large search space, multi-objective optimization is used to minimize for pipeline complexity while optimizing for performance (Olson et al., 2016). It is also possible to reduce the search space by specifying a pipeline template (Le et al., 2018), which dictates the high-level steps in the pipeline (for example, a "Selector-Transformer-Classifier" template will result in pipelines with only those three steps, in that order). Development has focused around genomic studies, providing specific options for dealing with this type of high dimensional data for which prior knowledge may be present (Sohn et al., 2017). While TPOT supports neural networks in its search (Romano et al., 2021), the default search space uses SCIKIT-LEARN components and XGBOOST only.

3.2 Baselines

In addition to the integrated frameworks, AMLB allows for running several baselines. The `constant predictor` always predicts the class prior or mean target value, regardless of the values of the independent variables. The `Random Forest` baseline builds a forest 10 trees at a time, until one of two criteria is met: we expect to exceed 90% of the memory limit or time limit by building 10 more trees, or 2000 trees have been built.

The `Tuned Random Forest` baseline improves on the `Random Forest` baseline by using an optimized `max_features` value. The `max_features` hyperparameter defines how many random features are considered when determining each split and is found to be the most important hyperparameter (Van Rijn and Hutter, 2018).[9] The value is optimized by evaluating up to 11 unique values for the hyperparameter with 5-fold cross-validation before training a final model with the best found value. The `Tuned Random Forest` is our strongest baseline and aims to mimic a typical first approach for modeling by a human.

4. Software

We developed an open source benchmark tool that may be used for reproducible AutoML benchmarking, which we will refer to as AMLB.[10] This tool features robust automated experiment execution and supports multiple AutoML frameworks, many of which are evaluated in this paper. AMLB is implemented as a Python application consisting mainly of an *amlb* module and a *framework* folder hosting all the officially supported extensions, which have been developed together with AutoML framework developers. The main consideration for the design of AMLB is to produce correct and reproducible evaluations; the AutoML frameworks are used as intended by their authors with little to no room for user error, and the same evaluation conditions (including framework version, data set, and resampling splits) and controlled computational environments can easily be recreated by anyone. The *amlb* module provides the following features:

- a data loader to retrieve and prepare data from *OpenML* or local data sets.

- various benchmark runner implementations:

    - a *local* runner that runs the experiments directly on the local machine. This is also the runner to which each runner below delegates the final execution.

    - *container* runners (*docker* and *singularity* are currently supported), which allow preinstalling the *amlb* application together with a full setup of one framework and consistently run all benchmark tasks against the same setup. This implementation also makes it possible to run multiple container instances in parallel.

    - an *aws* runner that allows the user to safely run the benchmark on several EC2 instances in parallel. Each EC2 instance can itself use a pre-built docker image, as used for this paper, or can configure the target framework on the fly, which is useful for experiments in a development environment.

---

9. The hyperparameter `min_samples_leaf` is statistically equally important.
10. Source code and documentation under MIT license at `https://github.com/openml/automlbenchmark`.

- a job executor responsible for running and orchestrating all the tasks. When used with the *aws* runner, this allows distribution of the benchmark tasks across hundreds of EC2 instances in parallel, with each one being monitored remotely by the host.

- a post-processor responsible for collecting and formatting the predictions returned by the frameworks, handling errors, and computing the scoring metrics before writing the information needed for post-analysis to a file.

Figure 13 in Appendix C provides an architecture overview and description of the flow of the benchmarking tool, as used in the experiments for this paper.

4.1 Extensible Framework Structure

To ensure that AMLB is easily extensible to new AutoML frameworks, we integrate each tool through a minimal interface. Each of the current tools requires less than 250 lines of code across at most four files (most of which is boilerplate). The integration code handles installation of the AutoML framework as well as its software stack and provides the framework with data and recording predictions. The integration requirements are minimal, as both input data and predictions can be exchanged both in Python objects and common file formats, which makes integration across programming languages possible (currently integrated frameworks are written in $C^{\#}$, Java, Python and R). By keeping the integration requirements minimal, we hope that AutoML framework authors are encouraged to contribute integration scripts for their framework and, at the same time, avoid influencing the methods or software used to design and develop new AutoML frameworks, as opposed to providing a generic starter kit which may bias the developed AutoML frameworks (Guyon et al., 2019). Frameworks may also be integrated completely locally to allow for private benchmarking.[11]

4.2 Extensible Benchmarks

Benchmark suites define the data sets and one or more train/test splits, which should be used to evaluate the AutoML frameworks. AMLB can work directly with OpenML tasks and suites (Bischl et al., 2021), allowing for new evaluations without further changes to the tool or its configuration. This is the preferred way to use AMLB for scientific experiments, as it guarantees that the exact evaluation procedure can be reproduced easily by others. However, it is also possible to use data sets stored in local files with manually defined splits, e.g., to benchmark private use cases.[12]

4.3 Using the Software

To benchmark an AutoML framework, the user must first identify and define:

- the *framework* against which the benchmark is executed,

---

11. For information on how to add a framework, see `https://openml.github.io/automlbenchmark/docs/extending/framework/`.
12. For information on how to add a new benchmark task or suite, see `https://openml.github.io/automlbenchmark/docs/extending/benchmark/`.

- the *benchmark* suite listing the tasks to use in the evaluation, and

- the *constraints* that must be imposed on each task. This includes:

  - the maximum training time.
  - the number of CPU cores that can be used by the framework; not all frameworks respect this constraint, but when run in *aws* mode, this constraint translates to specific EC2 instances, therefore limiting the total number of CPUs available to the framework.
  - the amount of memory that can be used by the framework; not all frameworks respect this constraint, but when run in *aws* mode, this constraint translates to specific EC2 instances, therefore limiting the total amount of memory available to the framework.
  - the amount of disk volume that can be used by the framework (only respected in *aws* mode).

  Those constraints must then be declared explicitly in a *constraints.yaml* file (also in the *resources* folder or as an external extension).

4.3.1 Commands

Once the previous parameters have been defined, the user can run a benchmark on the command line using the basic syntax:

```
$ python runbenchmark.py framework_id benchmark_id constraint_id
```

For example, to evaluate the tuned random forest baseline on the classification suite for 1 hour on 8 cores, run:

```
$ python runbenchmark.py tunedrandomforest openml/s/271 1h8c
```

Additional options may be used to specify the *mode*, the *parallelization*, and other details of the experimental setup. For example, the following command may be used to evaluate the random forest baseline on the regression benchmark suite across 100 8-core *aws* instances in parallel with a time budget of one hour.

```
$ python runbenchmark.py randomforest openml/s/269 1h8c -m aws -p 100
```

5. Benchmark Design

In this section, we discuss both the design of the *benchmark suite* (that is, the chosen data sets and evaluation procedures, Bischl et al., 2021) and the experimental setup, as well as their limitations.

5.1 Benchmark Suites

To facilitate a reproducible experimental evaluation, we make use of OpenML Benchmark suites (Bischl et al., 2021). An OpenML benchmark suite is a collection of OpenML tasks,

which each reference a data set, an evaluation procedure (such as k-fold cross-validation) and its splits, the target feature, and the type of task (regression or classification). The benchmark suites are designed to reflect a wide range of realistic use cases for which the AutoML tools are designed. Resource constraints are not part of the task definition. Instead, we define them separately in a local file so that each task can be evaluated with multiple resource constraints. Both the OpenML benchmark suite (and tasks) and the resource constraints are machine-readable to ensure automated and reproducible experiments.

5.1.1 DATA SETS

We created two benchmarking suites, one with 71 classification tasks, and one with 33 regression tasks. The data sets used in these tasks are selected from previous AutoML papers (Thornton et al., 2013), competitions (Guyon et al., 2019), and ML benchmarks (Bischl et al., 2021) according to the following predefined list of criteria[13]:

- **Difficulty** of the data set must be sufficiently high. If a problem is easily solved by almost any algorithm, it will not be able to differentiate the various AutoML frameworks. This can mean that simple models (such as random forests, decision trees or logistic regression) achieve a generalization error of zero, or that the performance of these models and all evaluated AutoML tools is identical.

- **Representative of real-world** data science problems to be solved with the frameworks. In particular, we limit artificial problems. We included a small selection of such problems, either based on their widespread use (e.g., kr-vs-kp) or because they pose difficult problems, but we do not want them to constitute a large part of the benchmark. We also limit computer vision problems on raw pixel data because those problems are better solved with dedicated deep learning solutions. However, since they still make for real-world, interesting, and hard problems, we did not exclude them altogether.

- **No free form text features** that cannot reasonably be interpreted as a categorical feature. Most AutoML frameworks do not yet support feature engineering on text features and will process them as categorical features. For this reason, we exclude text features, even though we admit their prevalence in many interesting real-world problems. A first investigation and benchmark of multimodal AutoML with text features has been carried out by Shi et al. (2021).

- **Diversity** in the problem domains. We do not want the benchmark to skew towards any application domain in particular. There are various software quality problems in the OpenML-CC18 benchmark (jm1, kc1, kc2, pc1, pc3, pc4), but adopting them all would lead to a bias in the benchmark to this domain.

- **Independent and identically distributed** (i.i.d.) data are required for each task. If the data are of temporal nature or repeated measurements have been conducted, the task is discarded. Both types of data are generally very interesting but are currently

---

13. The airlines datasets violate these criteria, but are included for historical reasons.
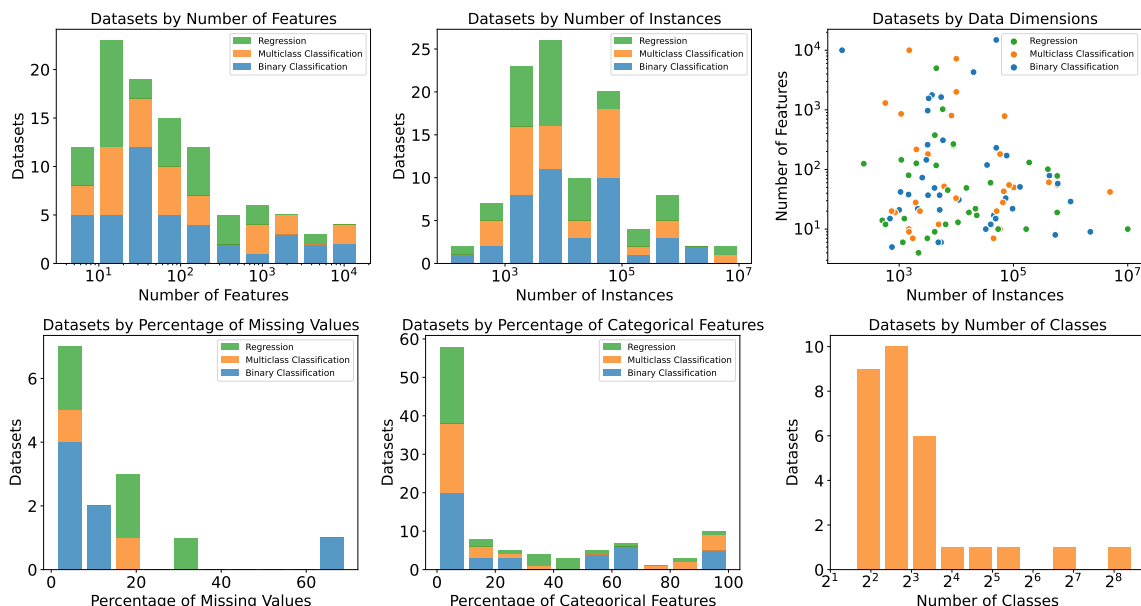
Figure 1: Properties of the tasks in both benchmarking suites.

not supported for most AutoML systems, and we plan to extend the benchmark in the future in this direction.

- **Freely available** and hosted on OpenML. Data sets that can only be used on specific platforms or are not shared freely for any reasons are not included in the benchmark.

Reasons to exclude a data set included label-leakage and near-duplicates of other tasks in independent variables (for example, different only in categorical encoding or imputation) or dependent variable (most commonly the binarization of a regression or multi-class task).

To study the differences between AutoML systems, the data sets vary in the number of samples and features by orders of magnitude and vary in the occurrence of numeric features, categorical features, and missing values. Figure 1 shows basic properties of the classification and regression tasks, including the distributions of the number of instances and features, the frequency of missing values and categorical features, and the number of target classes (for classification tasks). Properties of the tasks are shown in Appendix A and can be explored interactively on OpenML.[14] While the selection spans a wide range of data types and problem domains, we recognize that there is room for improvement. Restricting ourselves to open data sets without text features severely limits options, especially for big data sets. We hope to address this in future work.

All data sets are available in multiple formats for the AutoML frameworks, either as files (parquet, arff, or csv) or as Python objects (pandas dataframe, numpy array), and the used format depends on the AutoML framework. All frameworks have access to meta-data, such as the data type of the columns, either directly from the chosen data format or as separate

---

14. Visit `www.openml.org/s/269` for regression and `www.openml.org/s/271` for classification.

input, so that each AutoML framework has the same information available regardless of the chosen data format.

5.1.2 Performance Metrics

In our evaluation, we use area under the receiver operating characteristic curve (AUC) for binary classification, log loss for multi-class classification and root mean-squared error (rmse) for regression to evaluate model performance.[15] We chose to use these metrics because they are generally reasonable, commonly used in practice, and supported by most AutoML tools. The latter is especially important because it is imperative that AutoML systems optimize for the same metric on which they are evaluated. However, our tool is not limited to only these three metrics, and a wide range of performance metrics can be specified by the user.

Models are not calibrated, unless the AutoML framework does so by default. Whether or not calibrated models are important is highly dependent on the use case, and most AutoML frameworks do not support model calibration out of the box. This is especially important to keep in mind when we discuss results on our binary classification problems, as AUC is not sensitive to calibration.

5.1.3 Missing Values In Experimental Results

As will be discussed in more detail in Section 6.4, not all frameworks are equally well-behaved. There are situations where search time budgets are exceeded or the AutoML frameworks crash outright, which results in missing performance estimates. There are multiple strategies to consider on how to deal with these missing data.

One naive approach may be to ignore missing values and aggregate over the obtained results. However, we observe that failures do not occur at random. Failures correlate with data set properties, such as data set size and class imbalance, which may be correlated with "problem difficulty" and thus performance. Ignoring missing values thus means that AutoML frameworks may fail on harder tasks or folds and consequently obtain higher average performance estimates. Imputing missing values with performance obtained by the same AutoML framework on other folds is subject to the same drawback. Moreover, in case a framework fails to produce predictions on all folds of a task, neither method specifies how to deal with missing values.

Instead, we propose to impute the missing values with an interpretable and reliable baseline. An argument may be made for using the random forest baseline, since this may be a strong fallback that AutoML frameworks could realistically implement. However, we observe that training a random forest (of the size used in the baseline) requires a significant amount of time on larger data sets. Automatically providing this fallback by means of imputation would provide an unfair advantage to the AutoML frameworks that are not well-behaved. Moreover, many failures would not be remedied by having a random forest to fall back on, since the AutoML frameworks crash irrecoverably, for example, due to segmentation faults.

---

15. We use the implementations provided by `SCIKIT-LEARN` 1.2.2.

Therefore, we impute missing values with the constant predictor, or prior.[16] This baseline returns the empirical class distribution for classification and the empirical mean for regression. This is a very penalizing imputation strategy, as the constant predictor is often much worse than results obtained by the AutoML frameworks that produce predictions for the task or fold. However, we feel this penalization for ill-behaved systems is appropriate and fairer towards the well-behaved frameworks and hope that it encourages a standard of robust, well-behaved AutoML frameworks.

5.2 Experimental Setup

We execute the experiments on commodity-level hardware with AutoML frameworks generally in their default configurations.

5.2.1 Hardware

As discussed in Section 4, AMLB can be run on any machine. However, for comparable hardware and easy expandability, we opt to conduct the benchmark on standard `m5.2xlarge`[17] instances available on Amazon Web Services (AWS). These represent current commodity-level hardware with 32 GB memory, 8 vCPUs (Intel Xeon Platinum 8000 series Skylake-SP processor with a sustained all core Turbo CPU clock speed of up to 3.1 GHz). 100 GB of gp3-SSD storage is available for storage, which can be necessary for storing a larger number of evaluated pipelines. The use of AWS also enables others to fully reproduce and extend our results, as funding permits, since the results do not depend on private computing infrastructure.

5.2.2 Framework Configuration

AutoML frameworks are instantiated with their default configuration, except that we control the following settings:

- **'Mode'** to declare the user intent. For example, obtaining the best possible model versus finding an interpretable (less complex) model. The modes used to evaluate each AutoML framework is chosen by their developers.

- **Runtime** for the search. Additionally, there is one hour leeway for data loading, making predictions, and cleanup operations, but this is not communicated to the AutoML frameworks.

- **Resource constraints** that specify the number of CPU cores and amount of memory available.

- **Target metric** to use for optimization. This is the same metric that is used for evaluation in the benchmark.

The experiment design intentionally prohibits further customization of other AutoML system configuration parameters to reflect how these systems are usually applied in practice

---

16. We except one specific error for `AUTOGLUON`(HQIL) which is already fixed (c.f. Appendix D).
17. More information is available at `https://aws.amazon.com/ec2/instance-types/m5/`.

as closely as possible (note that AMLB does allow for this type of customization). An overview of used framework versions, their 'mode' configurations, and important integration details are provided in Appendix C.2.

5.3 Limitations

Both the design of the benchmark and the setup for the experiments described in this paper have some limitations with regard to the interpretation of their results. Limitations in the design stem from the desire to keep the use of the frameworks as close as possible to the original vision and usage intended by developers, whereas the limitations in the experiments are caused by resource constraints and may be alleviated by running additional experiments with the benchmark software. In this section, we highlight some important limitations and stress that this paper and the results within do *not* state which AutoML framework is ultimately the best.

5.3.1 Limitations of the Design

Perhaps the biggest limitation of the design is the inability to attribute the performance of an AutoML framework to any one aspect of its design, as is often done with ablation studies. The evaluated AutoML frameworks differ among multiple design choices, such as the underlying ML library, search space, preprocessing, and search algorithm. Concretely, a performance difference between, for instance, `AUTO-SKLEARN` and `TPOT` could be caused by `TPOT`'s built-in stacking, `AUTO-SKLEARN`'s ensembles, the difference in Bayesian optimization versus genetic programming, the difference in how multiprocessing is employed, or a combination of these or any other difference between them. Software that would allow for such conclusions essentially requires each AutoML framework to be reimplemented on a shared set of algorithms for building models, search, and evaluation. We acknowledge that this would be incredibly valuable for the research community. However, it would also no longer resemble the software as used in practice and thus would be different work altogether. Note that it *is* possible to perform ablation studies with AMLB for a specific AutoML framework, for example by comparing different framework configurations as done by Erickson et al. (2020).

Another limitation stems from only recording results produced by the final model. Anytime performance, where information about the performance during optimization is captured as if they were final models, can be very insightful. This method allows for the distinction between a framework that converges quickly from one that does not. This may be especially important for users who are interested to use the systems with a human-in-the-loop, such as when designing a search space or data features. Unfortunately, many frameworks do not support collection of anytime performance and— depending on how they are recorded—might interfere with resources used during search. We hope to be able to record anytime performance in the future, but in this work, we only approximate it by evaluating the tools under two different time constraints (1 and 4 hours).

Finally, the qualitative comparison of the frameworks is also limited. Certain "quality of life features" like analysis of the pipeline via interpretable ML methods, reports, usability, or support are not evaluated here but are important to many users. For a qualitative analysis of those characteristics, we refer the reader to one of the many existing overview papers

on AutoML (Zöller and Huber, 2021; Truong et al., 2019). We also provide an overview of various AutoML frameworks with links to their documentation on our website.[18]

5.3.2 Limitations of the Experiments

Most frameworks are highly configurable and allow the user to configure the search algorithm or its hyperparameters, among other aspects that affect the AutoML performance. Some frameworks even provide different configuration presets for different use cases, such as a performance-oriented competition mode and a mode that produces fast or interpretable models at the cost of some performance. However, comparing the effect on model performance of tuning AutoML hyper-hyperparameters quickly carries prohibitive costs. We evaluated only a few modes for each framework, if any, based on the developers' recommendations. It is likely that better results may be achieved by carefully meta-tuning the AutoML framework or evaluating additional modes. While it is cost prohibitive for us to evaluate many different scenarios, it is easy to run the benchmark with custom configurations for the various AutoML frameworks. This allows users to evaluate AutoML frameworks in a setting that reflects their interest.

5.3.3 Meta-learning

Many AutoML frameworks make use of meta-learning to better initialize and speed up the search (Yang et al., 2018; Feurer et al., 2015a, 2020). Since all data in the benchmark is publicly available and many of them are well known in the AutoML community, it is likely that there is a substantial overlap between data used by the developers for meta-learning and the data used in the benchmark. This is a very intricate problem, as we consider AutoML frameworks as black boxes. Removing the effect of the data set that is to be evaluated from the meta-learning procedure is not solvable in general.

In this paper specifically, both AUTO-SKLEARN and AUTO-SKLEARN 2 use meta-learning. AUTO-SKLEARN's meta-learning uses 208 data sets from OpenML, each associated with well-performing ML pipelines. The search is initialized with k-nearest datasets (Reif et al., 2012) using the 25-nearest datasets based on 38 meta-features and looking up the best known pipeline for those datasets. AUTO-SKLEARN can exclude data sets by name from the lookup, which we make use of in AMLB to ensure that there is no overlap to the 39 data sets from Gijsbers et al. (2019). Even so, it cannot be guaranteed that identical data sets with a different name might be used for meta learning out-of-the-box.

AUTO-SKLEARN 2's meta-learning model is more complicated and consists of: a) a static pipeline portfolio for warm-starting the search, which is computed across hundreds of data sets using a greedy forward selection, and b) a meta-model to predict the internal model selection strategy and budget allocation strategy. Single data sets cannot be excluded from these meta-learning procedures, and it is not feasible to retrain the meta-models and pipeline portfolio for each data set in our benchmark. This ultimately means that the result of AUTO-SKLEARN 2 must be considered very carefully and are likely optimistic, as their meta-model is built with information about many datasets in this benchmark. More thought and collaboration is required to address these issues and allow for the correct evaluation of AutoML systems that use meta-learning in common benchmarks.

---

18. `https://openml.github.io/automlbenchmark/frameworks.html`

5.4 Overfitting the Benchmark

One last issue that plagues any widely-adopted benchmark is the potential of algorithms overfitting on the data sets used in the benchmark. Since freely available, interesting, and usable (refer to Section 5.1.1 for our selection criteria) data sets are scarce, many AutoML developers use these data sets to benchmark and then improve their systems iteratively. While this is not as direct of an issue as with meta-learning, these data sets can in general not be assumed to be truly unseen. The only practical way to avoid this is to collect a novel set of data sets for the benchmark, which would entail a prohibitive effort. Moreover, after publishing such a benchmark, the new data sets are published, which again gives developers the possibility to use them to improve their systems. On the other hand, should the benchmarking data sets be kept private to avoid this issue, the benchmark is no longer entirely reproducible by independent researchers. We hope that the size of our benchmarking suites is large enough and their design general enough that overfitting is less of an issue, but this is difficult to guarantee, and a study as outlined above may be useful to evaluate this phenomenon in the future.

6. Results

In this section, we provide an overview and analysis of the results obtained. This section is accompanied by an interactive visualization tool[19], additional information in Appendix B, and all data artifacts generated from these experiments.[20] For a more comprehensive comparison than what we can provide here, we strongly encourage the reader to explore the data with the interactive visualization tool. This tool enables users to select any subset of frameworks, task types, performance measures, or data characteristics iteratively and interactively. The tool includes overview plots for different task types as well as detailed visualizations for individual data sets. Moreover, a statistical analysis of the results by critical difference plots and Bradley-Terry trees is implemented.

In addition to the limitations outlined in section 5.3, we also want to stress that these experimental results are obtained by running the frameworks as they were in June of 2023. Some of these frameworks are still under active development, and results from experiments run on later versions will almost certainly differ.

We used recent results as much as possible, but we also include some results of experiments ran of frameworks in September of 2021. In these cases, the benchmarked framework versions were not substantially different. We motivate this decision in Appendix C.2 and denote the older results with an asterisk (*) by the framework name. For example, TPOT denotes the June 2023 version and TPOT* denotes the September 2021 version. Some results are missing: AUTO-SKLEARN 2 does not support regression, and we could not complete our 4 hour classification evaluation of AUTOGLUON(HQIL) due to budget constraints.

We strongly encourage people to run additional experiments that match their use case with up-to-date frameworks, and use the results in this section as a reference on how to analyze the results. Results may differ substantially for different time constraints, amount of parallel processing available, availability of GPU's, and more. Table 16 in Appendix C.2

---

19. Results can be interactively explored at `http://openml.github.io/automlbenchmark/visualization`.
20. Experiment artifacts can be found at: `http://openml.github.io/automlbenchmark/data`.

shows an overview of the versions benchmarked for each framework as well as the most current version.

6.1 Predictive Performance

To report on the results for many AutoML frameworks across whole benchmarking suites, we propose using critical difference (CD) diagrams (Demšar, 2006) illustrated in Figure 2. In a CD diagram, the average rank of each framework as well as which ranks are statistically significantly different from each other are shown. To calculate the average rank per task, we first impute any missing values with the constant predictor and then average the performance over all folds. We may then test for the presence of statistically significant differences in the average rank distributions using a non-parametric Friedman test at $p < 0.05$ (here, $p \approx 0$ for every diagram) and use a Nemenyi post-hoc test to find which pairs differ. For each benchmarking suite and time budget, the CD diagrams are shown in Figure 2, which displays the rank of each framework (lower is better) averaged over all results from the given benchmarking suite and budget. AUTO-SKLEARN 2 is excluded in this comparison due to the meta-learning issues discussed in Section 5.3.3.

Overall, we observe that AUTOGLUON(B) and TPOT respectively achieve the best and worst rank among AutoML frameworks in almost every setting with respect to model accuracy. However, while AUTOGLUON(B) is generally not significantly better than the second best framework, it does rank statistically significantly better than the worst AutoML frameworks in any given scenario. Similarly, TPOT is never significantly worse than the second worst framework, but its rank is always statistically significantly worse than AUTOGLUON(B). In all cases, the baselines obtain worse ranks than any AutoML framework except TPOT, although the tuned random forest is a strong baseline. Only AUTOGLUON(B) and LIGHTAUTOML achieve significantly better ranks across all settings. All AutoML frameworks except AUTOGLUON(B) and TPOT are generally ranked close to each other, with small differences across the various suites and budgets.

To complement the CD diagrams—which obfuscate the relative performance differences—we show box plots of obtained results (after imputation) across all tasks in Figure 3. Because the performances are not commensurable across tasks, we first scale all results per task between the random forest performance (0) and the best observed performance (1) (that is, higher scores correspond to better performance). This also makes the scaled value interpretable. Furthermore, the value scales based on the improvement over the baseline that is observed to be achievable, and negative values are worse than the random forest baseline. While the boxplots are calculated over performance data on all tasks, the plots are cut off to allow a better visualization of the most relevant area. The number of outliers for each framework that are not shown in the plot are denoted on the x-axis.

Even if ranks are similar, the performance distribution might be noticeably different. For example, AUTOGLUON(HQIL) and MLJAR(P) achieve very similar average ranks on the one hour regression tasks. However, we observe from the boxplots that while AUTOGLUON(HQIL) achieves lower median normalized performance in this segment, it outperforms random forest with much greater consistency than MLJAR(P). Similarly, while TPOT's average rank is generally close to that of the Tuned Random Forest baseline, TPOT exhibits much higher variance in its prediction quality.

(a) Binary Classification, 1 hour

(b) Binary Classification, 4 hours

(c) Multi-class Classification, 1 hour

(d) Multi-class Classification, 4 hours

(e) Regression, 1 hour

(f) Regression, 4 hours

(g) All tasks, 1 hour

(h) All tasks, 4 hours
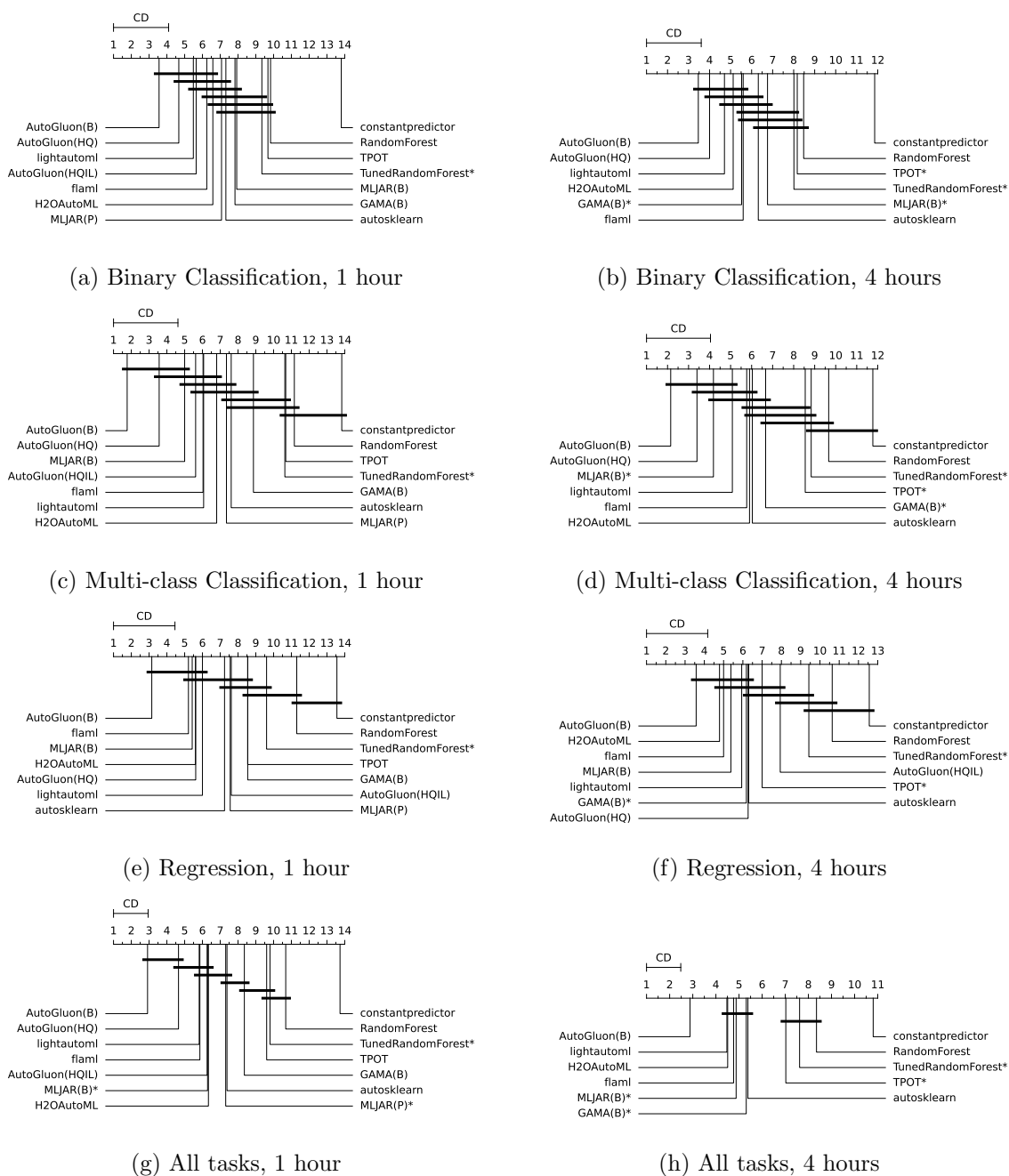
Figure 2: CD plots with Nimenyi post-hoc test after imputing missing values with the constant predictor baseline. An asterisk (*) next to a framework name means the results were obtained in 2021.
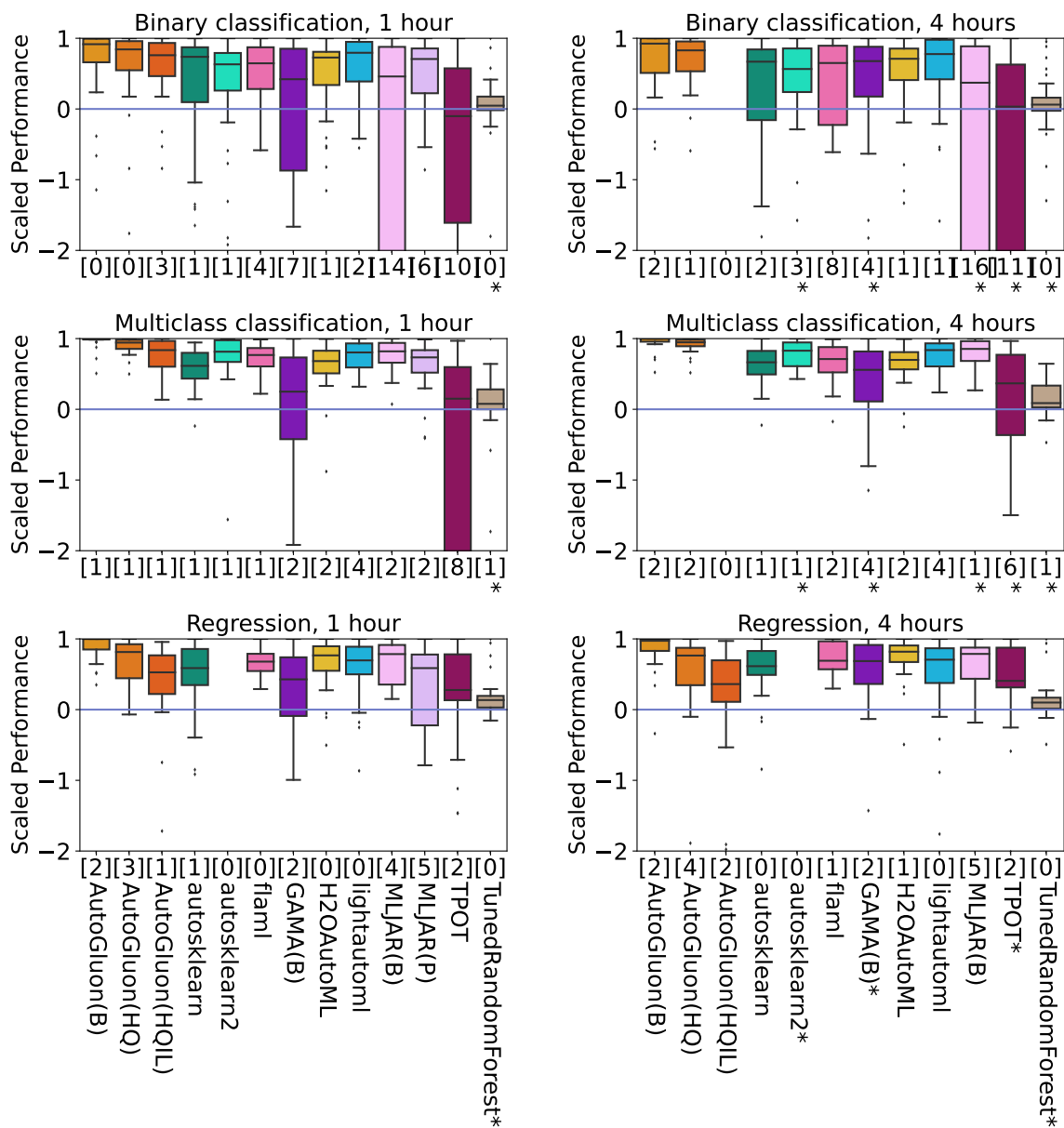
Figure 3: Boxplots of framework performance across tasks after scaling the performance values from random forest (0) to best observed (1). The number of outliers for each framework that are not shown in the plot are denoted on the x-axis. An asterisk (*) next to a framework name means the results were obtained in 2021.

Finally, we take a closer look at the difference in performance for different time constraints in Figure 4. Performance is scaled between 1 hour random forest performance (0) and best observed performance (1). We only include frameworks for which we evaluated all tasks on both constraints. We see that the performance is very similar overall, though

generally results improve slightly with a larger time budget. The performance of AUTOGLUON confirms that better performance is still possible for other frameworks, but we suspect that those frameworks are limited by their search space. For example, if a stacking ensemble is required to reach the best performance, then the other methods will not achieve the best performance regardless of time constraint.
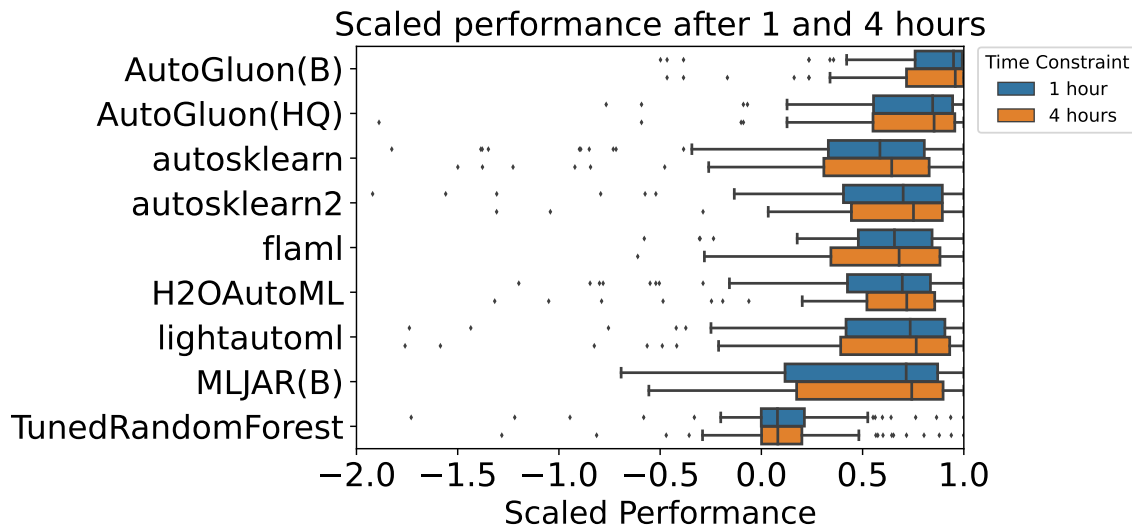
Figure 4: Scaled performance for each framework under different time constraints. Only frameworks which have evaluations on all tasks for both time constraints are shown. Performance generally does not improve much with more time.

6.2 Bradley-Terry Trees

Bradley-Terry (BT) trees (Strobl et al., 2011) can be used to statistically analyse benchmark experiments based on data set characteristics (Eugster et al., 2014). These trees use data set characteristics—such as the number of instances, the number of features, the ratio of missing values, and others—to split paired performance comparisons of the framework to find statistically significant differences in performance. Bradley-Terry models originate in psychology to analyze paired comparison experiments of subjects preferring one stimulus over another. For our benchmark, such a preference ranking can be easily derived by pairwise performance comparisons of all frameworks with regard to the data sets and cross-validation folds.

The underlying algorithm of model-based recursive partitioning of Bradley–Terry models works as follows: In each split of the BT tree, a BT model is fitted for the paired comparisons based on the underlying data set characteristics. Following Zeileis and Hornik (2007) and Eugster et al. (2014), the BT model performs a statistical test of parameter instability for the chosen data characteristics. If this test reveals a significant instability in the model parameters, the corresponding tree node splits the data according the the characteristic yielding the highest instability (lowest test p-value). The splitting cut-point is then

determined such that it has the highest improvement of the model fit. This procedure is repeated until either no significant instability is left, a set tree depth is reached, or further splits would exceed a set minimum number of observations in the leaves.

Numeric values in the tree leaves are *worth parameters* that can be interpreted as preferences for the different frameworks (Eugster et al., 2014). Since these values are in $[0, 1]$ and sum up to 1 within a leaf, they can be understood as the probability of a framework performing best, given the data characteristics in the corresponding leaf.

Figure 5 shows a Bradley-Terry tree for classification tasks for a runtime of one hour. For simplicity, in order to obtain an easily understood tree, only the number of instances (rows) and the number of features were chosen as data characteristics. The first split distinguishes between data sets with more than 8237 instances and those equal to or below that cutpoint. The second split then further split into leaves based on the number of features in the datasets. In the nodes, from left to right, we see that:

- For small datasets with a relatively low dimensionality (Node 3, 32 tasks) the performance differences among the best frameworks are not very pronounced. Even though
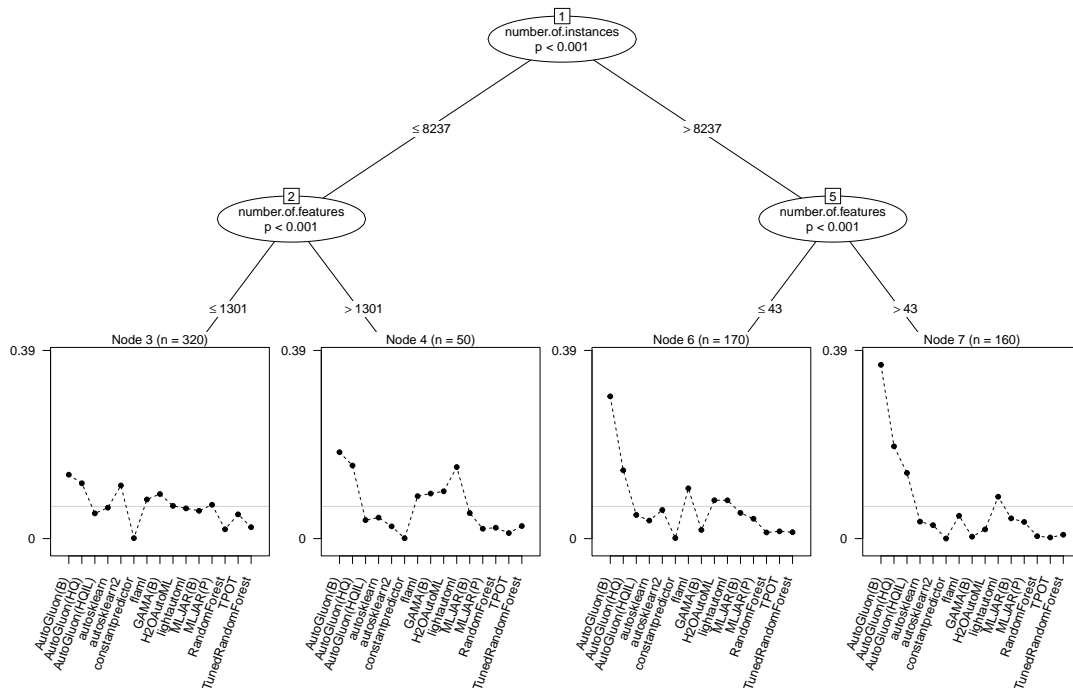

Figure 5: Bradley-Terry tree of depth three for classification tasks. Results from the one hour classification benchmark were used, and missing values were imputed by constant predictor performance. One observation within the BT tree equals the preference ranking of one fold on one data set. The n value denotes the number of observations in the leaf node.

25

AUTOGLUON(B) is preferred for those kinds of data sets, AUTOGLUON(HQ) and AUTO-SKLEARN 2 show similar performances.

- For very wide datasets, those with a similarly small number of instances but much higher dimensionality, more than 1301 features (Node 4, 5 tasks), performance differences increase overall with AUTOGLUON(B) still being preferred over all others, but now followed by AUTOGLUON(HQ) and LIGHTAUTOML with similar performances, and larger distance to H2O AUTOML, GAMA(B) and FLAML.

- For datasets with a larger number of instances, more than 8237 (Node 6 and 7), we see that AUTOGLUON(B) is preferred by a large margin over AUTOGLUON(HQ) as second. There are minor differences depending on the number of features, such as FLAML being a preferred third option when there are few features, at most 43, but AUTOGLUON(HQIL) being preferred if the dataset has a relatively higher dimensionality.

More Bradley-Terry trees for the subsets of binary and multiclass classification as well as regression can be found in the appendix. The findings from the BT trees are essentially the same as those presented here, as AUTOGLUON(B) is overall the preferred framework in most tree leaves. One exception is for small balanced binary classification tasks, shown in Figure 10, where various frameworks are preferred over AUTOGLUON.

In conclusion, it can be observed that the performance gap of AUTOGLUON(B) increases particularly as data sets become more complex. Moreover, the reader is strongly invited to explore the aforementioned interactive visualization tool, with which deeper BT trees based on several more data set characteristics can be constructed on various task types.

6.3 Model Accuracy vs. Inference Time Trade-offs

Model accuracy (here measured by AUC, log loss, or RMSE) plays a central role in evaluating performance of machine learning models. However, maximizing accuracy can come at the cost of added model complexity. One practical way to consider the complexity of the model is to measure the inference speed of the resulting model.

Some of the integrated frameworks offer a "compete" mode (AUTOGLUON, MLJAR, LIGHTAUTOML and GAMA) that maximizes accuracy, typically at the cost of increased model complexity, similar to competing in a Kaggle competition. This can lead to models being built that are highly accurate but are extremely slow at inference time and are therefore not practical in many real-life use-cases.

However, some frameworks provide multiple presets that allow the user to balance the trade-off between accuracy and inference time differently. Unless the use of a specific preset is mentioned (for AUTOGLUON and MLJAR), results in this section used presets which prioritize accuracy over inference time, or in the case of H2O AUTOML, a balance of the two. Thus, performing additional experiments with other presets is advised when inference time is important. We also recognize that there are applications for which inference time is not important.

In order to evaluate the limitations of the models produced by each framework we also measured their inference time. The inference time reported on in this section is measured by having the frameworks infer on samples of test data, either on a single row which is in memory, or on ten thousand rows loaded from disk. Each measurement is done ten times

and the median is reported. Measurements for TPOT are omitted from this section, as TPOT requires data which is already encoded and the encoding step may consume a considerable part of the total inference time, making direct comparisons invalid. This metric provides important insight into the trade-offs that tool authors make in their algorithm designs.

Figure 6 shows aggregated inference times across all models standardized to rows per second (fewer is slower). For H2O AUTOML we do not report in-memory inference speed as technical limitations of our benchmarking tool result in pessimistic inference time measurements for non-Python frameworks.[21] Here we see that the high accuracy of AUTOGLUON(B) comes at the cost of extremely slow inference times. This is explained by the large models produced by combining both stacking and ensembling to form ensembles of multiple layers. In general, we see that ensemble models are slower (AUTO-SKLEARN, GAMA, and MLJAR(B) also do ensembling), and GAMA and AUTO-SKLEARN with the same search space and ensemble methods have comparable inference speeds. H2O AUTOML and AUTOGLUON(HQIL) maintain fast inference speed despite making use of ensembles, by building on more optimized models or explicitly selecting for fast inference when building the ensemble. FLAML stand out as having very fast inference time, a potential explanation is that FLAML's cost-frugal optimization also indirectly takes inference time into account, as it is part of the time estimate which is used to consider which model to tune.
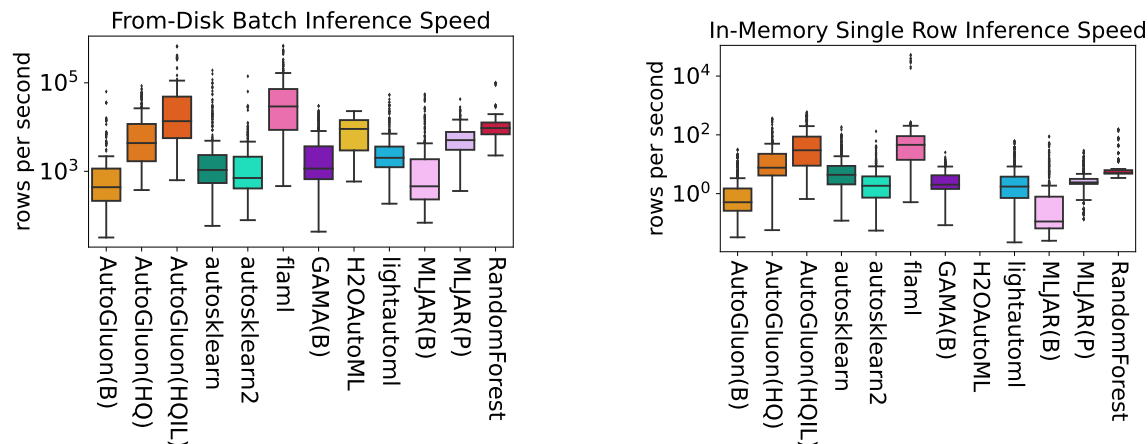
Figure 6: Inference speed on unseen data in rows per second, normalized from inference on 10 thousand rows from solid-state drive (left) or single rows in memory (right). Measured for models created with the one hour time constraint and all scenarios.

In Figure 7, we show the Pareto front for all three scenarios with a one-hour time constraint, demonstrating the average normalized model accuracy against corresponding median per-row prediction speeds. Here, it is more apparent that the frameworks that

---

21. Transferring data between our Python-based benchmark tool and H2O AUTOML's Java back-end requires serialization to disk. Serialization to disk occurs significant overhead, especially when comparing to in-memory inference. Faster inference speeds will be obtained by using H2O Frames directly in H2O AUTOML, or using H2O in it's production mode, which performs inference on exported models directly from disk instead of in memory.

achieve the highest accuracy do so at the cost of inference time performance. This demonstrates that when contextualizing any type of model accuracy results, it is important to consider any trade-offs that may have been made to achieve the extra performance and how that will affect the framework's usability in practice. In this case, measuring accuracy in isolation does not give the complete picture of the overall utility of a particular framework.

It should also be noted that with sufficient computing infrastructure and effort, scoring across rows or chunks of data could be parallelized in a production system, which would reduce the overall prediction time as compared to predicting a single test set on a single machine. However, our goal in this section was to compare the inference time of the high-accuracy models derived from different AutoML frameworks to each each other rather than evaluate different techniques for speeding up the inference of any individual system.
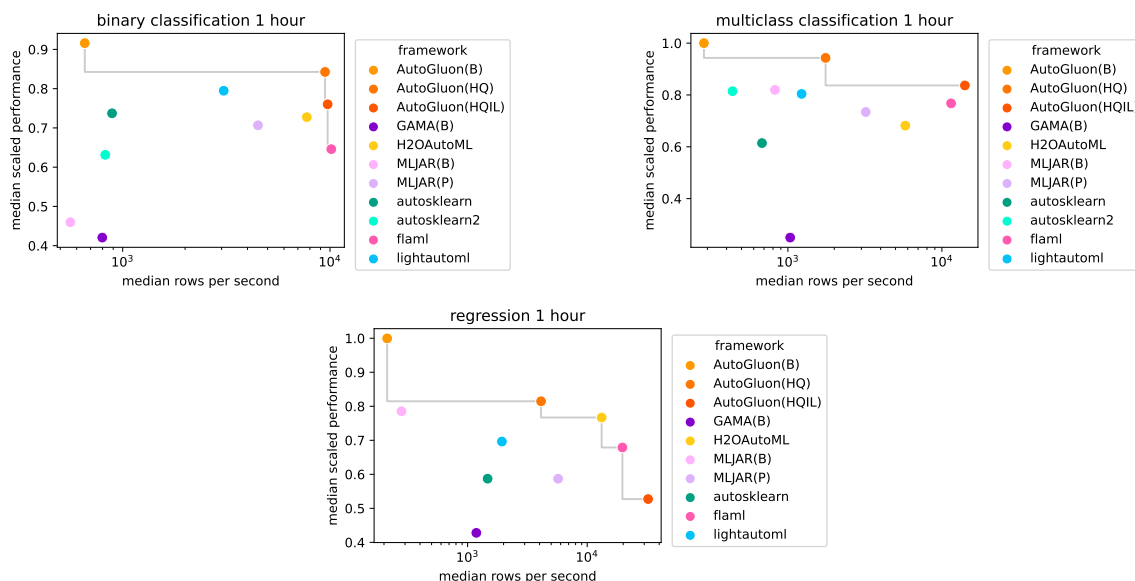
Figure 7: Pareto Frontiers of framework performance across tasks after scaling the performance values from the random forest (0) to best observed (1) for each task type on a one hour time budget.

6.4 Observed AutoML Failures

While most jobs completed successfully, we observed multiple framework errors during our experiments. In this section, we will discuss where AutoML frameworks fail, although we want to stress that development for these packages is ongoing. For that reason, it is likely that the same frameworks will not experience the same failures in the future (especially after gaining access to all experiment logs). We categorize the errors into the following categories:

Memory: The framework crashed due to exceeding available memory or encountering other memory-related errors, such as segmentation faults.

Time: The framework exceeded the time limit past the leniency period.

Data: Errors due to specific data characteristics (such as imbalanced data) occurred.

Implementation: Any errors caused by bugs in the AutoML framework code occurred.

These categories are a bit crude and ultimately subjective, since from a reductive viewpoint, all errors are implementation errors. However, they serve for a quick overview. We also introduce a 'fixed' category in Figure 8a to denote errors from a specific bug in AUTOGLUON(HQIL) which is already fixed in newer releases. Additional details on this, and other errors encountered, can be found in Appendix D.

Figure 8a shows the errors by type, and Figure 8b shows errors by dataset size and dimensionality on the right. Overall, memory and time constraints are the main cause for errors, with one major exception.[22] We observe that errors are far more common in the classification benchmark suite than the regression suite. This is largely accounted for by the difference in benchmarking suite size (33 and 71 tasks) and the fact that the largest data sets are mostly classification tasks, both in number of instances and features. Unique to classification, we do observe several frameworks failing to produce models or predictions on highly imbalanced data sets. This is also the case for the failures on the two small classification data sets ('yeast' and 'wine-quality-white'), where careless use of internal validation splits yields splits that no longer contain all classes. Interestingly, the distribution of the type of errors observed is different under different time constraints (as shown in Figure 14a of Appendix D). Both memory and time constraint violations happen more frequently, which may potentially be explained by frameworks saving increasingly more models or building increasingly larger pipelines. It is worth noting the stability of the different systems is varied, with some systems being far more stable than others. Some types of errors, such as memory errors, could potentially be easily resolved by running the AutoML system on a machine with more RAM. However, the implementation errors are more problematic because they represent an error from the code of the AutoML system. Those are not currently resolvable without changes to the AutoML system itself, which may or may not be fixed by code authors in the future.

Only when the framework exceeds the time budget by more than one hour do we record a time error. However, as we can see in Figure 9, not all AutoML frameworks adhere to the runtime constraints equally well, even if they finish within the leniency period. In the figure, the training duration for each individual job (task and fold combination) are aggregated, and timeout errors are shown above each framework, where missing values due to non-time errors are excluded. These plots reveal different design decisions around the specified runtime, with some frameworks never exceeding the limit by more than a few minutes, while others violate it by a larger margin with some regularity. Interestingly, we see that a number of frameworks consistently tend to stop far before the specified runtime limit.

---

22. MLJAR(B) has 284 'implementation errors' which are almost exclusively caused by only 3 distinct errors.
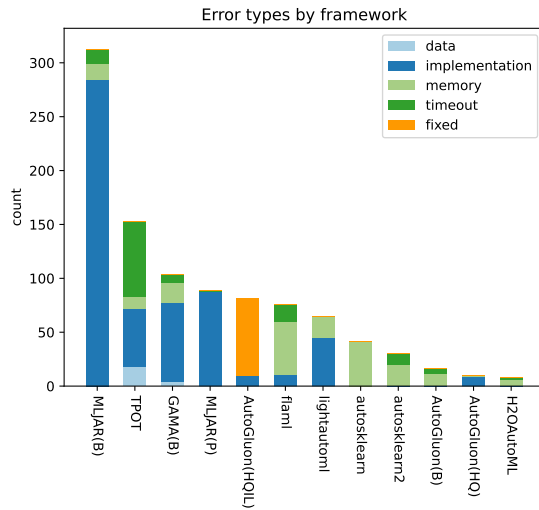
(a) Errors by type for each framework.
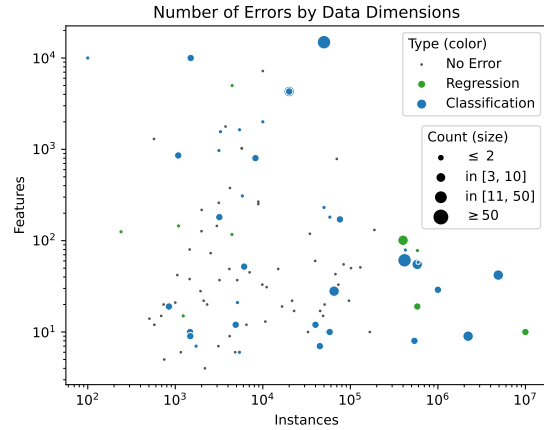
(b) Number of errors by size of the dataset.
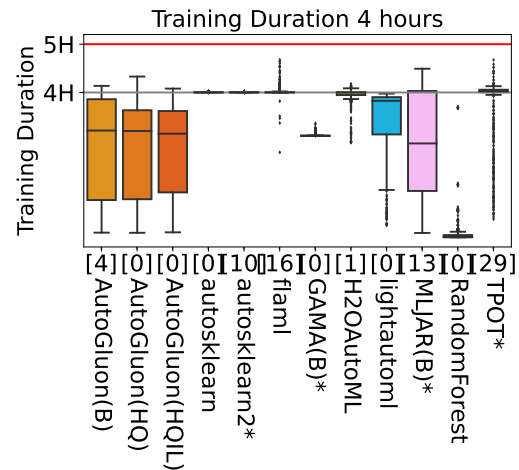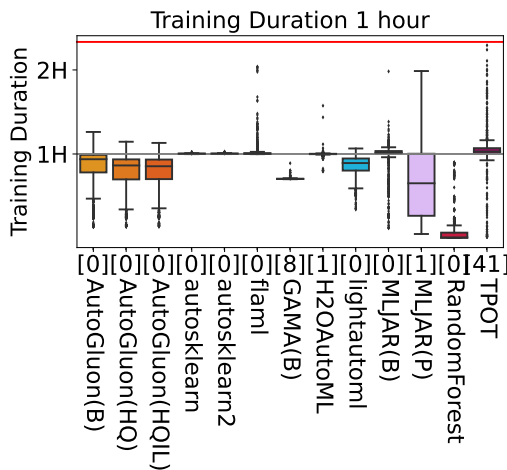
Figure 9: Time spent during search with a one hour budget (left) and four hour budget (right). The grey line indicates the specified time limit, and the red line denotes the end of the leniency period. The number of timeout errors for each framework are shown beside it.

7. Conclusion

We presented a novel benchmark for measuring and comparing AutoML frameworks. To ensure reproducibility, fair comparison, and detailed analysis of the results, our open-source benchmarking tool automates the empirical evaluation of any integrated framework on any supported task, including installation of the AutoML framework, provisioning the data for training and inference, resource allocation, and processing of the results. This greatly simplifies evaluating AutoML frameworks while enhancing reproducibility and reducing

errors. We worked jointly with the authors of 9 AutoML frameworks to evaluate their systems in a large-scale study on 71 classification and 33 regression tasks.

When analyzing the predictive performance of these AutoML systems, we find that the *average* ranks of the AutoML frameworks are generally very competitive with each other. Still, by using Bradley-Terry trees (Strobl et al., 2011), we find that their relative performance is affected by data characteristics—such as the data set size, dimensionality, and class balance—highlighting specific strengths and weaknesses. Overall, in terms of model performance, AUTOGLUON consistently has the highest average rank in our benchmark. Additionally, in most scenarios, the AutoML frameworks outperform even our strongest baseline.

Because inference time is an important factor in real-world applications, we also reviewed the inference time and accuracy trade-off and found large differences in inference time of the produced models, at times spanning multiple orders of magnitude. The most accurate frameworks achieve higher model accuracy at a large cost to performance in terms of inference speed. Broadly speaking, while the models with higher accuracy also have slower inference time, not all frameworks produce models that are Pareto optimal. Furthermore, specific AutoML frameworks allow users to choose presets for customizing the accuracy-inference time trade-off.

Finally, we analyzed scenarios in which AutoML frameworks fail to produce a model and found that the main cause for failure was data set size. In other words, not all methods scale well. To allow further analysis of our results, we provide an open-source interactive visualization tool, which includes graphical representations and statistical tests.

7.1 Limitations

These quantitative results are obtained by using AutoML frameworks with preset configurations only. The performance of frameworks under non-default settings can be very relevant when, for example, there is budget available to also optimize AutoML hyper-hyperparameters, or when a custom search space is used. Additionally, many frameworks are under active development, so the results presented here may not be representative of their performance in the future.

Next, all frameworks differ along multiple design axes, which prohibits attributing performance differences to any specific component of the AutoML framework (such as the search algorithm), without additional analysis.

Moreover, since we provide a purely quantitative comparison, it ignores qualitative aspects of AutoML frameworks that are very relevant in real-world settings, such as the produced model's interpretability or the level of support.

Lastly, the benchmarks were executed on 8-CPU machines, which is very modest by today's standards. Therefore, frameworks with better parallelism may show even greater advantages on higher powered machines (with, for example, 50 or 100 CPUs). On the other hand, even shorter time constraints may be useful for a human-in-the-loop or green AutoML setting.

7.2 Future Work

The benchmarking suites, or sets of tasks, proposed in this paper are meant as a starting point to be improved upon in collaboration with the AutoML community. We will seek to update these suites based on community discussion so that they continue to reflect modern challenges while also decreasing the risk of AutoML frameworks overfitting to the benchmark. In particular, we are interested in extending the benchmarking suites with problems that feature free-form text data. Real-world tabular data often contain instances of text data with different semantic meaning, such as addresses or URLs, from which meaningful features can be extracted. This kind of feature engineering is typically very important for model performance on such data sets, but the current selection of data sets does not reflect this, in part because not all AutoML frameworks support text features.

We would also like to extend the benchmarking tool to support new problems, such as multi-objective optimization tasks. While model accuracy is often an important metric, 'secondary' metrics—such as a model's inference time or fairness—are often crucial for real-world applications. Multi-objective optimization can be used to convey the importance of these metrics to AutoML frameworks, for example, by using a fairness metric as secondary objective, (Schmucker et al., 2021)), which can subsequently provide Pareto fronts of models that optimize this trade-off. In particular for fairness related tasks, additional support to convey sensitive attributes and protected groups to the AutoML frameworks must be added. Other interesting problem types include non-i.i.d. data, such as when temporal relationships are present in the data, or semi-supervised learning, where not all instances have an associated ground truth.

Finally, the evaluations in this paper were performed on the equivalent of commodity-level hardware using only CPU and a limited time budget. In some cases, it is more desirable to devote a large budget to building a single model, perhaps even days of compute, potentially with GPU access. In other cases, much smaller budgets may be desired, for example to reduce carbon footprints or improve the experience for workflows with a human in the loop. As different behavior (both in robustness and model performance) on one-hour and four-hour budgets are already observed, future work may reveal different behavior when evaluating AutoML frameworks at different scales.

7.3 Parting Words

The benchmark tool presented in this work makes producing rigorous reproducible research both easier and faster. We hope that the open and extensible nature of this benchmark motivates researchers to not only use the tool, but also to contribute their own data sets, framework integrations, or feedback and code contributions to the open source AutoML benchmark. We strongly encourage this participation so that the benchmark may remain useful to the community for a long time to come.

Acknowledgments

Appendix A. OpenML Benchmark Suites

Table 2 and Table 3 contain an overview of data sets used in the regression and classification benchmarking suites, respectively. We hope to continuously update the benchmarking suites with new data sets that represent current challenges.

Table 2: Tasks in the AutoML regression suite.

Task ID	name	n	p
359944	abalone	4177	9
359929	Airlines_DepDelay_10M	10000000	10
233212	Allstate_Claims_Severity	188318	131
359937	black_friday	166821	10
359950	boston	506	14
359938	Brazilian_houses	10692	13
233213	Buzzinsocialmedia_Twitter	583250	78
359942	colleges	7063	45
233211	diamonds	53940	10
359936	elevators	16599	19
359952	house_16H	22784	17
359951	house_prices_nominal	1460	80
359949	house_sales	21613	22
233215	Mercedes_Benz_Greener_Manufacturing	4209	377
360945	MIP-2016-regression	1090	145
167210	Moneyball	1232	15
359943	nyc-taxi-green-dec-2016	581835	19
359941	OnlineNewsPopularity	39644	60
359946	pol	15000	49
360933	QSAR-TID-10980	5766	1026
360932	QSAR-TID-11	5742	1026
359930	quake	2178	4
233214	Santander_transaction_value	4459	4992
359948	SAT11-HAND-runtime-regression	4440	117
359931	sensory	576	12
359932	socmob	1156	6
359933	space_ga	3107	7
359934	tecator	240	125
359939	topo_2_1	8885	267
359945	us_crime	1994	127
359935	wine_quality	6497	12
317614	Yolanda	400000	101
359940	yprop_4_1	8885	252

Table 3: Tasks in the AutoML classification suite.

Task ID	name	n	p	C	class ratio
190411	ada	4147	49	2	0.33
359983	adult	48842	15	2	0.31
189354	airlines	539383	8	2	0.80
189356	albert	425240	79	2	1.00
10090	amazon-commerce-reviews	1500	10001	50	1.00
359979	Amazon_employee_access	32769	10	2	0.06
168868	APSFailure	76000	171	2	0.02
190412	arcene	100	10001	2	0.79
146818	Australian	690	15	2	0.80
359982	bank-marketing	45211	17	2	0.13
359967	Bioresponse	3751	1777	2	0.84
359955	blood-transfusion-service-center	748	5	2	0.31
359960	car	1728	7	4	0.05
359973	christine	5418	1637	2	1.00
359968	churn	5000	21	2	0.16
359992	Click_prediction_small	39948	12	2	0.20
359959	cmc	1473	10	3	0.53
359957	cnae-9	1080	857	9	1.00
359977	connect-4	67557	43	3	0.15
7593	covertype	581012	55	7	0.01
168757	credit-g	1000	21	2	0.43
211986	Diabetes130US	101766	50	3	0.21
168909	dilbert	10000	2001	5	0.93
189355	dionis	416188	61	355	0.36
359964	dna	3186	181	3	0.46
359954	eucalyptus	736	20	5	0.49
168910	fabert	8237	801	7	0.26
359976	Fashion-MNIST	70000	785	10	1.00
359969	first-order-theorem-proving	6118	52	6	0.19
359970	GesturePhaseSegmentationProcessed	9873	33	5	0.34
189922	gina	3153	971	2	0.97
359988	guillermo	20000	4297	2	0.67
359984	helena	65196	28	100	0.03
360114	Higgs	1000000	29	2	0.89
359966	Internet-Advertisements	3279	1559	2	0.16
211979	jannis	83733	55	4	0.04
168911	jasmine	2984	145	2	1.00
359981	jungle_chess_2pcs_raw_endgame_complete	44819	7	3	0.19

Table 3: Tasks in the AutoML classification suite (continued).

Task ID	name	n	p	C	class ratio
359962	kc1	2109	22	2	0.18
360975	KDDCup09-Upselling	50000	14892	2	0.08
3945	KDDCup09_appetency	50000	231	2	0.02
360112	KDDCup99	4898431	42	23	0.00
359991	kick	72983	33	2	0.14
359965	kr-vs-kp	3196	37	2	0.91
190392	madeline	3140	260	2	0.99
359961	mfeat-factors	2000	217	10	1.00
359953	micro-mass	571	1301	20	0.18
359990	MiniBooNE	130064	51	2	0.39
359980	nomao	34465	119	2	0.40
167120	numerai28.6	96320	22	2	0.98
359993	okcupid-stem	50789	20	3	0.13
190137	ozone-level-8hr	2534	73	2	0.07
359958	pc4	1458	38	2	0.14
190410	philippine	5832	309	2	1.00
359971	PhishingWebsites	11055	31	2	0.80
168350	phoneme	5404	6	2	0.42
360113	porto-seguro	595212	58	2	0.04
359956	qsar-biodeg	1055	42	2	0.51
359989	riccardo	20000	4297	2	0.33
359986	robert	10000	7201	10	0.92
359975	Satellite	5100	37	2	0.01
359963	segment	2310	20	7	1.00
359994	sf-police-incidents	2215023	9	2	0.14
359987	shuttle	58000	10	7	0.00
168784	steel-plates-fault	1941	28	7	0.08
359972	sylvine	5124	21	2	1.00
190146	vehicle	846	19	4	0.91
359985	volkert	58310	181	10	0.11
146820	wilt	4839	6	2	0.06
359974	wine-quality-white	4898	12	7	0.00
2073	yeast	1484	9	10	0.01

Appendix B. Results

This appendix contains additional figures and tables with experimental results. Tables 4-15 report results per framework per task for different task types and time budgets. Each value denotes the mean score of completed folds, the standard deviation in those completed folds, and the number of folds for which the AutoML framework did not return a result, if applicable. A '-' denotes cases where the AutoML framework was unable to complete any fold of a task on a specific time budget. A '*' next to a framework name denotes experiments that were conducted on an older version of the framework in 2021.

task id	framework / task name	AutoGluon(B)	AutoGluon(HQ)	AutoGluon(HQIL)	autosklearn	autosklearn2	flaml	GAMA(B)	H2OAutoML
146818	Australi...	0.941(0.018)	0.942(0.017)	0.942(0.017)	0.931(0.022)	0.936(0.019)	0.938(0.023)	0.941(0.022)[1]	0.935(0.025)
146820	wilt	0.995(0.008)	0.994(0.010)	0.994(0.010)	0.994(0.009)	0.995(0.007)	0.991(0.011)	0.996(0.004)	0.992(0.010)
167120	numerai2...	0.531(0.004)	0.528(0.004)	0.529(0.004)	0.530(0.004)	0.531(0.004)	0.528(0.004)	0.530(0.006)	0.531(0.004)
168350	phoneme	0.968(0.009)	0.968(0.009)	0.968(0.009)	0.963(0.009)	0.970(0.009)	0.972(0.008)	0.971(0.010)[2]	0.968(0.009)
168757	credit-g	0.796(0.041)	0.788(0.044)	0.763(0.053)	0.778(0.030)	0.796(0.041)	0.788(0.039)	0.794(0.033)	0.779(0.048)
168868	APSFailu...	0.993(0.002)	0.993(0.002)	0.993(0.002)	0.993(0.002)	0.992(0.001)	0.992(0.002)	0.990(0.003)	0.993(0.002)
168911	jasmine	0.886(0.018)	0.885(0.019)	0.885(0.019)	0.885(0.016)	0.888(0.017)	0.887(0.016)	0.891(0.013)	0.887(0.018)
189354	airlines	0.732(0.002)	0.732(0.002)	0.732(0.002)	0.726(0.003)	0.727(0.002)	0.731(0.002)	0.717(0.007)	0.731(0.002)
189356	albert	0.782(0.002)	0.782(0.002)	0.782(0.003)	0.756(0.003)	0.737(0.003)	0.770(0.002)	0.726(0.006)	0.761(0.003)
189922	gina	0.991(0.005)	0.992(0.005)	0.992(0.005)	0.990(0.005)	0.988(0.007)	0.992(0.005)	0.991(0.006)	0.991(0.005)
190137	ozone-le...	0.933(0.016)	0.931(0.016)	0.931(0.016)	0.926(0.019)	0.927(0.026)	0.922(0.019)	0.925(0.030)	0.930(0.020)
190392	madeline	0.945(0.008)	0.944(0.008)	0.944(0.008)	0.965(0.007)	0.945(0.008)	0.954(0.007)	0.957(0.009)	0.943(0.010)
190410	philippi...	0.878(0.013)	0.877(0.011)	0.877(0.011)	0.915(0.011)	0.877(0.014)	0.891(0.014)	0.892(0.016)	0.882(0.011)
190411	ada	0.921(0.018)	0.920(0.018)	0.920(0.018)	0.918(0.018)	0.919(0.018)	0.924(0.018)	0.920(0.019)	0.921(0.019)
190412	arcene	0.873(0.155)	0.873(0.173)	0.852(0.169)	0.866(0.141)	0.845(0.196)	0.868(0.154)	0.878(0.154)[1]	0.839(0.228)
359955	blood-tr...	0.758(0.046)	0.759(0.043)	0.759(0.043)	0.747(0.050)	0.756(0.049)	0.730(0.042)	0.753(0.036)	0.764(0.041)
359956	qsar-bio...	0.942(0.032)	0.941(0.035)	0.939(0.034)	0.931(0.034)	0.936(0.027)	0.930(0.031)	0.936(0.032)	0.939(0.034)
359958	pc4	0.952(0.021)	0.949(0.022)	0.950(0.023)	0.942(0.019)	0.949(0.018)	0.950(0.019)	0.950(0.022)	0.948(0.022)
359962	kc1	0.840(0.034)	0.840(0.035)	0.840(0.035)	0.843(0.033)	0.838(0.037)	0.841(0.040)	0.852(0.030)	0.829(0.030)
359965	kr-vs-kp	1.000(0.000)	1.000(0.001)	1.000(0.001)	1.000(0.000)	1.000(0.000)	0.961(0.123)	1.000(0.000)	1.000(0.001)
359966	Internet...	0.985(0.011)	0.986(0.012)	0.978(0.016)	0.984(0.012)	0.984(0.013)	0.987(0.011)	0.983(0.012)	0.986(0.010)
359967	Biorespo...	0.886(0.017)	0.884(0.016)	0.878(0.016)	0.873(0.017)	0.870(0.018)	0.884(0.016)	0.884(0.017)	0.887(0.016)
359968	churn	0.922(0.021)	0.924(0.022)	0.923(0.023)	0.922(0.021)	0.920(0.024)	0.922(0.023)	0.920(0.023)	0.925(0.022)
359971	Phishing...	0.998(0.001)	0.998(0.001)	0.997(0.002)	0.997(0.001)	0.997(0.001)	0.998(0.001)	0.997(0.001)[1]	0.998(0.001)
359972	sylvine	0.990(0.003)	0.989(0.003)	0.989(0.003)	0.991(0.003)	0.990(0.002)	0.991(0.002)	0.993(0.002)[2]	0.990(0.003)
359973	christine	0.826(0.014)	0.827(0.013)	0.827(0.013)	0.828(0.015)	0.819(0.011)	0.824(0.013)	0.828(0.017)	0.825(0.013)
359975	Satellite	0.996(0.005)	0.996(0.007)	0.996(0.007)	0.994(0.007)	0.995(0.007)	0.978(0.041)	0.996(0.003)	0.979(0.033)
359979	Amazon_e...	0.902(0.012)	0.901(0.011)	0.901(0.011)	0.853(0.017)	0.876(0.014)	0.876(0.014)	0.867(0.012)[1]	0.877(0.012)
359980	nomao	0.997(0.001)	0.997(0.001)	0.997(0.001)	0.996(0.001)	0.996(0.001)	0.997(0.001)	0.995(0.001)	0.996(0.001)
359982	bank-mar...	0.941(0.006)	0.941(0.006)	0.940(0.006)	0.939(0.006)	0.938(0.007)	0.937(0.006)	0.936(0.006)	0.938(0.007)
359983	adult	0.932(0.004)	0.932(0.004)	0.932(0.004)	0.930(0.004)	0.931(0.004)	0.932(0.004)	0.929(0.004)	0.931(0.004)
359988	guillermo	0.914(0.008)	0.914(0.008)	0.914(0.009)	0.914(0.008)	0.906(0.009)	0.919(nan)[9]	0.865(0.031)	0.897(0.010)
359989	riccardo	1.000(0.000)	1.000(0.000)	1.000(0.000)	1.000(0.000)	1.000(0.000)	1.000(0.000)[3]	0.999(0.000)	1.000(0.000)
359990	MiniBooNE	0.989(0.001)	0.988(0.001)	0.988(0.001)	0.987(0.001)	0.987(0.001)	0.987(0.001)	0.982(0.002)	0.987(0.001)
359991	kick	0.791(0.007)	0.790(0.007)	0.790(0.007)	0.785(0.006)	0.787(0.007)	0.787(0.007)	0.791(0.016)	0.789(0.007)
359992	Click-pr...	0.710(0.012)	0.710(0.012)	0.710(0.012)	0.697(0.014)	0.699(0.014)	0.723(0.010)	0.655(0.019)	0.701(0.013)
359994	sf-polic...	0.697(0.001)	0.693(0.002)	0.677(0.002)	0.694(0.005)	0.693(0.022)	0.709(0.004)	0.623(0.011)	0.688(0.013)
360113	porto-se...	0.643(0.004)	0.643(0.005)	0.643(0.005)	0.639(0.004)	0.640(0.005)	0.642(0.004)	0.624(0.006)	0.642(0.004)
360114	Higgs	0.838(0.001)	0.837(0.001)	0.675(0.001)	0.838(0.001)	0.835(0.001)	0.839(0.001)	0.784(0.006)[1]	0.832(0.001)
360975	KDDCup09...	0.908(0.007)	0.906(0.007)	0.906(0.007)	-	-	-	0.558(0.085)[8]	0.903(0.005)
3945	KDDCup09...	0.849(0.013)	0.848(0.013)	0.848(0.013)	0.838(0.015)	0.838(0.014)	0.825(0.017)	0.818(0.020)	0.837(0.014)

Table 4: Results for binary classification (in AUC) on a one hour budget, denoted as mean(std)$^{\texttt{fails}}$.

task id	framework task name	lightautoml	MLJAR(B)	MLJAR(P)	TPOT	constantpredictor	RandomForest	TunedRandomForest*
146818	Australi...	0.946(0.020)	0.943(0.020)	0.944(0.017)	0.939(0.022)	0.500(0.000)	0.940(0.021)	0.938(0.022)
146820	wilt	0.994(0.007)	0.999(0.000)[8]	0.995(0.009)	0.996(0.004)	0.500(0.000)	0.989(0.012)	0.991(0.010)
167120	numerai2...	0.531(0.004)	0.530(0.004)	0.531(0.005)	0.528(0.007)	0.500(0.000)	0.519(0.006)	0.520(0.006)
168350	phoneme	0.966(0.008)	-	0.967(0.008)	0.969(0.008)	0.500(0.000)	0.965(0.009)	0.966(0.009)
168757	credit-g	0.796(0.037)	-	0.785(0.046)	0.791(0.052)	0.500(0.000)	0.801(0.038)	0.799(0.035)
168868	APSFailu...	0.993(0.002)	0.995(nan)[9]	0.992(0.002)	0.989(0.004)[1]	0.500(0.000)	0.991(0.002)	0.992(0.003)
168911	jasmine	0.880(0.018)	0.885(0.016)	0.886(0.019)	0.886(0.013)	0.500(0.000)	0.888(0.016)	0.888(0.017)
189354	airlines	0.727(0.002)	0.731(0.002)	0.730(0.002)	0.722(0.002)	0.500(0.000)	0.661(0.002)	0.661(0.002)
189356	albert	0.780(0.003)	0.783(0.003)	0.765(0.003)	0.718(0.014)[1]	0.500(0.000)	0.738(0.003)	0.738(0.002)
189922	gina	0.990(0.006)	0.991(0.005)	0.990(0.005)	0.988(0.007)[1]	0.500(0.000)	0.984(0.008)	0.984(0.008)
190137	ozone-le...	0.930(0.018)	-	0.929(0.019)	0.928(0.026)	0.500(0.000)	0.920(0.036)	0.921(0.033)
190392	madeline	0.935(0.010)	0.956(0.011)	0.950(0.012)	0.948(0.009)	0.500(0.000)	0.887(0.017)	0.932(0.013)
190410	philippi...	0.866(0.014)	0.882(0.013)	0.872(0.012)	0.879(0.012)[1]	0.500(0.000)	0.849(0.016)	0.858(0.015)
190411	ada	0.921(0.018)	0.921(0.018)	0.921(0.018)	0.917(0.019)	0.500(0.000)	0.903(0.014)	0.902(0.016)
190412	arcene	0.848(0.176)	0.870(0.183)	-	0.831(0.151)[6]	0.500(0.000)	0.857(0.175)	0.828(0.183)
359955	blood-tr...	0.753(0.058)	-	0.753(0.053)	0.724(0.102)	0.500(0.000)	0.686(0.064)	0.686(0.065)
359956	qsar-bio...	0.934(0.034)	0.932(nan)[9]	0.937(0.031)	0.926(0.045)	0.500(0.000)	0.932(0.033)	0.935(0.029)
359958	pc4	0.950(0.015)	0.951(0.016)	0.953(0.016)	0.944(0.021)	0.500(0.000)	0.947(0.019)	0.949(0.019)
359962	kc1	0.831(0.032)	0.828(0.030)	0.824(0.032)	0.844(0.038)	0.500(0.000)	0.837(0.037)	0.841(0.037)
359965	kr-vs-kp	1.000(0.001)	-	1.000(0.001)	0.999(0.001)	0.500(0.000)	0.999(0.001)	0.999(0.002)
359966	Internet...	0.986(0.010)	0.982(0.011)[4]	0.977(0.012)	0.981(0.012)[6]	0.500(0.000)	0.986(0.012)	0.988(0.009)
359967	Biorespo...	0.884(0.019)	0.882(0.017)	0.883(0.016)	0.880(0.020)[1]	0.500(0.000)	0.881(0.018)	0.881(0.018)
359968	churn	0.926(0.020)	0.928(0.021)	0.925(0.019)	0.919(0.025)	0.500(0.000)	0.916(0.023)	0.912(0.021)
359971	Phishing...	0.998(0.001)	-	0.998(0.001)	0.997(0.001)	0.500(0.000)	0.996(0.002)	0.996(0.002)
359972	sylvine	0.988(0.003)	0.992(0.003)	0.992(0.003)	0.992(0.002)	0.500(0.000)	0.983(0.004)	0.984(0.004)
359973	christine	0.831(0.013)	0.827(0.013)	0.823(0.012)	0.811(0.013)[2]	0.500(0.000)	0.805(0.016)	0.809(0.013)
359975	Satellite	0.986(0.025)	-	0.993(0.007)	0.984(0.038)	0.500(0.000)	0.985(0.029)	0.984(0.031)
359979	Amazon-e...	0.879(0.009)	0.905(0.013)	0.903(0.013)	0.864(0.014)	0.500(0.000)	0.863(0.015)	0.863(0.014)
359980	nomao	0.997(0.001)	0.997(0.000)[8]	0.997(0.001)	0.995(0.001)	0.500(0.000)	0.995(0.001)	0.995(0.001)
359982	bank-mar...	0.940(0.006)	-	0.940(0.006)	0.935(0.007)	0.500(0.000)	0.931(0.006)	0.931(0.007)
359983	adult	0.932(0.004)	-	0.931(0.004)	0.927(0.004)	0.500(0.000)	0.910(0.003)	0.910(0.003)
359988	guillermo	0.932(0.007)	0.915(0.009)	0.912(0.007)[3]	0.826(0.053)	0.500(0.000)	0.903(0.009)	0.903(0.009)
359989	riccardo	1.000(0.000)	1.000(0.000)	1.000(0.000)	0.998(0.001)	0.500(0.000)	0.999(0.000)	1.000(0.000)
359990	MiniBooNE	0.988(0.001)	0.987(0.001)	0.987(0.001)	0.992(0.001)	0.500(0.000)	0.982(0.001)	0.982(0.001)
359991	kick	0.785(0.008)	0.754(0.012)	0.751(0.009)	0.728(0.009)	0.500(0.000)	0.761(0.008)	0.766(0.009)
359992	Click-pr...	0.728(0.010)	0.709(0.014)	0.709(0.014)	0.715(0.008)	0.500(0.000)	0.687(0.009)	0.689(0.009)
359994	sf-polic...	0.685(0.002)	0.703(0.007)	0.694(0.012)[8]	0.615(0.058)[4]	0.500(0.000)	0.704(0.002)	0.703(0.002)
360113	porto-se...	0.641(0.004)	0.641(0.004)	0.643(0.004)	0.577(0.058)[1]	0.500(0.000)	0.610(0.005)	0.618(0.006)
360114	Higgs	0.838(0.001)	0.836(0.002)	-	0.737(0.043)[1]	0.500(0.000)	0.822(0.001)	0.819(0.001)
360975	KDDCup09...	0.910(0.007)	0.906(0.006)	-	-	0.500(0.000)	0.883(0.008)	0.883(0.007)
3945	KDDCup09...	0.851(0.012)	-	0.837(0.015)	0.831(0.017)[2]	0.500(0.000)	0.796(0.018)	0.795(0.021)

Table 5: Results for binary classification (in AUC) on a one hour budget, denoted as $\mathrm{mean(std)}^{\mathrm{fails}}$.

task id	framework task name	AutoGluon(B)	AutoGluon(HQ)	autosklearn	autosklearn2*	flaml	GAMA(B)*	H2OAutoML
146818	Australi...	0.941(0.018)	0.942(0.017)	0.931(0.023)	0.940(0.020)	0.941(0.025)	0.940(0.019)	0.935(0.021)
146820	wilt	0.995(0.008)	0.994(0.010)	0.994(0.009)	0.995(0.008)	0.991(0.010)	0.996(0.004)	0.994(0.008)
167120	numerai2...	0.531(0.004)	0.529(0.004)	0.530(0.005)	0.531(0.004)	0.529(0.004)	0.532(0.004)[1]	0.531(0.004)
168350	phoneme	0.968(0.009)	0.968(0.009)	0.964(0.009)	0.970(0.009)	0.972(0.009)	0.971(0.009)	0.967(0.010)
168757	credit-g	0.796(0.041)	0.788(0.044)	0.785(0.042)	0.795(0.038)	0.783(0.045)	0.791(0.030)	0.783(0.043)
168868	APSFailu...	0.993(0.002)	0.992(0.002)	0.993(0.002)	0.992(0.002)	0.992(0.002)[1]	0.992(0.002)	0.992(0.002)
168911	jasmine	0.886(0.018)	0.885(0.019)	0.883(0.016)	0.887(0.017)	0.887(0.016)	0.893(0.014)	0.887(0.020)
189354	airlines	0.733(0.002)	0.733(0.002)	0.727(0.002)	0.727(0.002)	0.732(0.002)	-	0.732(0.002)
189356	albert	0.786(0.002)	0.785(0.002)	0.762(0.003)	0.759(0.002)	0.771(0.002)[1]	0.747(0.009)	0.769(0.003)
189922	gina	0.991(0.005)	0.992(0.005)	0.991(0.006)	0.988(0.007)	0.992(0.005)	0.991(0.005)	0.990(0.006)
190137	ozone-le...	0.933(0.016)	0.931(0.016)	0.913(0.029)	0.933(0.022)	0.924(0.020)	0.926(0.032)	0.925(0.025)
190392	madeline	0.945(0.008)	0.944(0.008)	0.969(0.006)	0.945(0.008)	0.954(0.008)	0.959(0.008)	0.947(0.010)
190410	philippi...	0.878(0.013)	0.877(0.011)	0.917(0.012)	0.877(0.014)	0.893(0.013)	0.903(0.014)	0.880(0.015)
190411	ada	0.921(0.018)	0.920(0.018)	0.918(0.018)	0.920(0.018)	0.924(0.018)	0.921(0.018)	0.921(0.016)
190412	arcene	0.861(0.176)	0.869(0.173)	0.873(0.155)	0.832(0.156)	0.881(0.120)	0.856(0.162)	0.865(0.156)
359955	blood-tr...	0.758(0.046)	0.759(0.043)	0.748(0.052)	0.755(0.040)	0.730(0.042)	0.757(0.049)	0.756(0.047)
359956	qsar-bio...	0.942(0.032)	0.941(0.035)	0.932(0.032)	0.937(0.027)	0.934(0.028)	0.937(0.032)	0.939(0.033)
359958	pc4	0.952(0.021)	0.949(0.022)	0.942(0.020)	0.949(0.017)	0.946(0.021)	0.951(0.019)	0.951(0.019)
359962	kc1	0.840(0.034)	0.840(0.035)	0.843(0.027)	0.839(0.036)	0.841(0.039)	0.851(0.032)	0.821(0.040)
359965	kr-vs-kp	1.000(0.000)	1.000(0.000)	1.000(0.000)	1.000(0.000)	0.976(0.075)	1.000(0.000)	1.000(0.001)
359966	Internet...	0.985(0.011)	0.986(0.012)	0.986(0.011)	0.982(0.015)	0.986(0.012)	0.984(0.011)	0.987(0.009)
359967	Biorespo...	0.886(0.017)	0.885(0.016)	0.872(0.019)	0.873(0.018)	0.886(0.016)	0.885(0.017)	0.886(0.017)
359968	churn	0.922(0.021)	0.924(0.022)	0.917(0.020)	0.919(0.021)	0.926(0.027)	0.921(0.022)	0.929(0.015)
359971	Phishing...	0.998(0.001)	0.998(0.001)	0.997(0.001)	0.997(0.001)	0.998(0.001)	0.998(0.001)	0.998(0.001)
359972	sylvine	0.990(0.003)	0.989(0.003)	0.992(0.003)	0.990(0.002)	0.991(0.002)	0.993(0.002)	0.990(0.003)
359973	christine	0.827(0.013)	0.826(0.013)	0.829(0.014)	0.818(0.013)	0.827(0.012)	0.833(0.014)	0.824(0.011)
359975	Satellite	0.996(0.005)	0.996(0.007)	0.983(0.039)	0.995(0.005)	0.978(0.041)	0.996(0.002)	0.993(0.007)
359979	Amazon_e...	0.902(0.012)	0.901(0.011)	0.861(0.017)	0.878(0.010)	0.878(0.013)	0.862(0.013)	0.875(0.011)
359980	nomao	0.997(0.001)	0.997(0.001)	0.996(0.001)	0.997(0.001)	0.997(0.001)	0.996(0.001)	0.996(0.001)
359982	bank-mar...	0.941(0.006)	0.941(0.006)	0.938(0.006)	0.939(0.007)	0.938(0.006)	0.937(0.007)	0.939(0.007)
359983	adult	0.932(0.004)	0.932(0.004)	0.930(0.005)	0.931(0.004)	0.932(0.004)	0.930(0.004)	0.931(0.004)
359988	guillermo	0.919(0.008)	0.919(0.007)	0.912(0.010)	0.907(0.009)	-	0.908(0.010)	0.909(0.009)
359989	riccardo	1.000(0.000)	1.000(0.000)	1.000(0.000)	1.000(0.000)	1.000(0.000)[4]	1.000(0.000)	1.000(0.000)
359990	MiniBooNE	0.989(0.001)	0.988(0.001)	0.987(0.001)	0.988(0.001)	0.988(0.001)	0.985(0.001)	0.987(0.001)
359991	kick	0.792(0.007)	0.790(0.007)	0.786(0.008)	0.786(0.007)	0.788(0.007)	0.788(0.006)	0.789(0.007)
359992	Click-pr...	0.710(0.012)	0.709(0.012)	0.697(0.014)	0.703(0.012)	0.724(0.009)	0.660(0.015)	0.702(0.013)
359994	sf-polic...	0.711(0.004)[4]	0.717(0.007)	0.701(0.006)	0.706(0.002)	0.718(0.003)[2]	0.648(0.012)	0.707(0.002)[1]
360113	porto-se...	0.644(0.004)	0.644(0.005)	0.641(0.004)	0.640(0.004)	0.642(0.004)	0.633(0.005)	0.643(0.004)
360114	Higgs	0.851(0.001)	0.850(0.001)	0.842(0.003)	0.842(0.003)	0.841(0.001)[4]	0.806(0.008)[1]	0.834(0.001)
360975	KDDCup09...	0.910(0.007)	0.909(0.007)	-	-	-	-	0.901(0.008)
3945	KDDCup09...	0.850(0.012)	0.848(0.013)	0.836(0.015)	0.842(0.016)	0.839(0.015)	0.831(0.015)	0.837(0.014)

Table 6: Results for binary classification (in AUC) on a four hour budget, denoted as mean(std)$^{\text{fails}}$. Results obtained in 2021 are denoted with a '*' next to the framework name.

Table 7: Results for binary classification (in AUC) on a four hour budget, denoted as $\mathtt{mean(std)}^{\mathtt{fails}}$. Results obtained in 2021 are denoted with a '*' next to the framework name.

task id	framework task name	lightautoml	MLJAR(B)*	TPOT*	constantpredictor	RandomForest	TunedRandomForest*
146818	Australi...	0.946(0.020)	0.940(0.024)	0.936(0.024)	0.500(0.000)	0.940(0.021)	0.938(0.022)
146820	wilt	0.994(0.007)	0.994(0.003)[5]	0.985(0.025)	0.500(0.000)	0.989(0.012)	0.990(0.011)
167120	numerai2...	0.531(0.004)	0.530(0.004)	0.527(0.006)	0.500(0.000)	0.519(0.006)	0.520(0.006)
168350	phoneme	0.965(0.008)	-	0.971(0.009)	0.500(0.000)	0.965(0.009)	0.966(0.009)
168757	credit-g	0.797(0.043)	-	0.787(0.034)	0.500(0.000)	0.801(0.038)	0.802(0.034)
168868	APSFailu...	0.993(0.002)	0.993(0.002)[6]	0.989(0.003)[1]	0.500(0.000)	0.991(0.002)	0.990(0.003)
168911	jasmine	0.881(0.018)	0.891(0.016)	0.889(0.012)	0.500(0.000)	0.888(0.016)	0.887(0.017)
189354	airlines	0.730(0.002)	0.732(0.002)	0.724(0.002)	0.500(0.000)	0.661(0.002)	0.654(0.003)
189356	albert	0.780(0.002)	0.785(0.002)	0.734(0.009)	0.500(0.000)	0.738(0.003)	0.738(0.002)
189922	gina	0.990(0.006)	0.993(0.004)	0.991(0.005)[1]	0.500(0.000)	0.984(0.008)	0.984(0.008)
190137	ozone-le...	0.930(0.016)	0.911(0.019)[8]	0.916(0.026)	0.500(0.000)	0.920(0.036)	0.921(0.035)
190392	madeline	0.935(0.009)	0.963(0.008)	0.954(0.007)	0.500(0.000)	0.887(0.017)	0.933(0.013)
190410	philippi...	0.866(0.015)	0.905(0.010)	0.897(0.013)[1]	0.500(0.000)	0.849(0.016)	0.858(0.015)
190411	ada	0.921(0.018)	0.921(0.018)	0.917(0.018)	0.500(0.000)	0.903(0.014)	0.902(0.015)
190412	arcene	0.852(0.164)	0.864(0.156)	0.840(0.135)[4]	0.500(0.000)	0.857(0.175)	0.857(0.178)
359955	blood-tr...	0.751(0.060)	-	0.754(0.043)	0.500(0.000)	0.686(0.064)	0.687(0.062)
359956	qsar-bio...	0.934(0.033)	0.926(nan)[9]	0.933(0.031)	0.500(0.000)	0.932(0.033)	0.935(0.029)
359958	pc4	0.949(0.016)	0.951(0.017)	0.943(0.023)	0.500(0.000)	0.947(0.019)	0.949(0.019)
359962	kc1	0.830(0.033)	0.829(0.032)	0.844(0.036)	0.500(0.000)	0.837(0.037)	0.841(0.037)
359965	kr-vs-kp	1.000(0.000)	1.000(0.000)[7]	0.950(0.158)	0.500(0.000)	0.999(0.001)	1.000(0.001)
359966	Internet...	0.987(0.008)	0.991(nan)[9]	0.982(0.011)	0.500(0.000)	0.986(0.012)	0.987(0.011)
359967	Biorespo...	0.884(0.018)	0.885(0.018)	0.880(0.017)[1]	0.500(0.000)	0.881(0.018)	0.881(0.019)
359968	churn	0.926(0.022)	0.931(0.024)	0.919(0.022)	0.500(0.000)	0.916(0.023)	0.913(0.023)
359971	Phishing...	0.998(0.001)	-	0.849(0.240)	0.500(0.000)	0.996(0.002)	0.996(0.002)
359972	sylvine	0.988(0.003)	0.993(0.003)	0.995(0.001)	0.500(0.000)	0.983(0.004)	0.984(0.004)
359973	christine	0.831(0.013)	0.829(0.012)	0.816(0.013)[1]	0.500(0.000)	0.805(0.016)	0.809(0.015)
359975	Satellite	0.987(0.024)	0.989(0.015)[7]	0.990(0.023)	0.500(0.000)	0.985(0.029)	0.991(0.011)
359979	Amazon_e...	0.878(0.010)	0.904(0.012)	0.866(0.013)	0.500(0.000)	0.863(0.015)	0.862(0.014)
359980	nomao	0.997(0.001)	0.997(0.001)[7]	0.996(0.001)	0.500(0.000)	0.995(0.001)	0.995(0.001)
359982	bank-mar...	0.941(0.006)	0.943(0.005)[6]	0.935(0.007)	0.500(0.000)	0.931(0.006)	0.931(0.007)
359983	adult	0.932(0.004)	-	0.928(0.004)	0.500(0.000)	0.910(0.003)	0.910(0.004)
359988	guillermo	0.936(0.005)	0.917(0.007)	0.859(0.048)	0.500(0.000)	0.903(0.009)	0.906(0.009)
359989	riccardo	1.000(0.000)	1.000(0.000)	0.997(0.004)	0.500(0.000)	0.999(0.000)	1.000(0.000)
359990	MiniBooNE	0.988(0.001)	0.988(0.001)	0.983(0.001)	0.500(0.000)	0.982(0.001)	0.982(0.001)
359991	kick	0.784(0.007)	0.757(0.012)	0.742(0.006)	0.500(0.000)	0.761(0.008)	0.766(0.009)
359992	Click-pr...	0.728(0.010)	-	0.719(0.010)	0.500(0.000)	0.687(0.009)	0.688(0.009)
359994	sf-polic...	0.696(0.002)	0.708(0.003)	0.670(0.010)[1]	0.500(0.000)	0.704(0.002)	0.687(0.015)
360113	porto-se...	0.642(0.004)	0.643(0.004)	0.631(0.005)	0.500(0.000)	0.610(0.005)	0.617(0.006)
360114	Higgs	0.839(0.001)	0.838(0.002)	0.777(0.007)	0.500(0.000)	0.822(0.001)	0.822(0.001)
360975	KDDCup09...	0.912(0.008)	0.909(0.006)	-	0.500(0.000)	0.883(0.008)	0.860(0.031)
3945	KDDCup09...	0.851(0.012)	0.835(0.021)[5]	0.830(0.016)	0.500(0.000)	0.796(0.018)	0.794(0.019)

task id	framework task name	AutoGluon(B)	AutoGluon(HQ)	AutoGluon(HQIL)	autosklearn	autosklearn2	flaml	GAMA(B)	H2OAutoML
10090	amazon-c...	0.695(0.086)	0.747(0.104)	1.187(0.119)	1.139(0.132)	1.125(0.146)	1.115(0.155)	0.910(0.066)	1.077(0.092)
168784	steel-pl...	0.464(0.042)	0.465(0.050)	0.507(0.037)	0.516(0.032)	0.457(0.027)	0.482(0.037)	0.491(0.029)	0.490(0.036)
168909	dilbert	0.014(0.006)	0.013(0.007)	0.014(0.007)	0.033(0.012)[1]	0.037(0.007)	0.024(0.008)	0.176(0.033)	0.065(0.007)
168910	fabert	0.683(0.030)	0.684(0.030)	-	0.745(0.034)	0.725(0.029)	0.766(0.025)	0.763(0.032)	0.746(0.029)
189355	dionis	0.266(0.004)	0.302(0.009)	0.299(0.010)	0.731(0.169)	1.843(1.453)	0.495(0.287)	1.446(0.145)	3.331(0.098)[5]
190146	vehicle	0.312(0.050)	0.341(0.060)	-	0.364(0.036)	0.322(0.040)	0.439(0.041)	0.378(0.056)	0.351(0.077)
2073	yeast	1.003(0.091)	1.008(0.092)	1.019(0.097)	1.037(0.085)	1.011(0.083)	1.004(0.077)	1.042(0.094)[2]	1.061(0.120)
211979	jannis	0.650(0.006)	0.651(0.006)	0.651(0.006)	0.663(0.007)	0.671(0.004)	0.674(0.006)	0.732(0.014)	0.669(0.006)
211986	Diabetes...	0.831(0.006)	0.831(0.006)	0.831(0.006)	0.835(0.005)	0.832(0.006)	0.831(0.006)	0.847(0.008)	0.833(0.007)
359953	micro-ma...	0.211(0.079)	0.247(0.108)	0.247(0.109)	0.249(0.107)	0.212(0.079)	0.309(0.136)	0.226(0.063)	0.400(0.086)
359954	eucalypt...	0.654(0.065)	0.675(0.083)	0.691(0.091)	0.714(0.047)	0.688(0.048)	0.726(0.080)	0.701(0.061)[1]	0.666(0.075)
359957	cnae-9	0.126(0.056)	0.158(0.110)	0.200(0.118)[7]	0.159(0.051)	0.129(0.041)	0.164(0.052)	0.126(0.041)	0.200(0.075)
359959	cmc	0.927(0.064)	0.937(0.067)	-	0.890(0.034)	0.881(0.039)	0.904(0.049)	0.897(0.041)	0.902(0.045)
359960	car	0.002(0.003)	0.010(0.027)	0.011(0.012)[2]	0.013(0.026)	0.001(0.001)	0.002(0.002)	0.022(0.015)[1]	0.001(0.002)
359961	mfeat-fa...	0.071(0.026)	0.073(0.031)	0.074(0.030)	0.093(0.038)	0.073(0.033)	0.092(0.035)	0.077(0.036)	0.096(0.042)
359963	segment	0.052(0.024)	0.056(0.028)	0.075(0.049)	0.078(0.042)	0.059(0.023)	0.067(0.031)	0.067(0.025)	0.061(0.025)
359964	dna	0.106(0.030)	0.113(0.033)	0.097(0.029)[7]	0.116(0.031)	0.112(0.028)	0.108(0.029)	0.107(0.028)	0.108(0.030)
359969	first-or...	1.039(0.045)	1.036(0.040)	-	1.100(0.035)	1.032(0.031)	1.041(0.024)	1.057(0.033)	1.075(0.040)
359970	GestureP...	0.668(0.032)	0.671(0.031)	0.776(0.039)	0.803(0.024)	0.726(0.032)	0.769(0.031)	0.848(0.043)[1]	0.702(0.038)
359974	wine-qua...	0.698(0.027)	0.702(0.027)	0.765(0.034)	0.794(0.038)	0.722(0.028)	0.727(0.041)	0.766(0.032)[3]	0.782(0.056)
359976	Fashion-...	0.221(0.009)	0.225(0.008)	0.225(0.009)	0.253(0.008)	0.251(0.009)	0.253(0.014)	0.439(0.036)	0.283(0.017)
359977	connect-4	0.295(0.007)	0.318(0.016)	0.445(0.093)	0.366(0.013)	0.348(0.014)	0.340(0.007)	0.417(0.048)	0.311(0.009)
359981	jungle-c...	0.012(0.002)	0.017(0.004)	-	0.199(0.025)	0.200(0.021)	0.210(0.006)	0.243(0.012)	0.136(0.025)
359984	helena	2.467(0.016)	2.498(0.015)	2.560(0.014)	2.553(0.016)	2.537(0.030)	2.617(0.025)	2.802(0.021)[8]	2.791(0.022)
359985	volkert	0.672(0.010)	0.690(0.017)	0.907(0.140)	0.789(0.019)	0.778(0.010)	0.795(0.014)	1.102(0.018)	0.844(0.010)
359986	robert	1.304(0.021)	1.351(0.020)	1.363(0.027)	1.425(0.030)	1.382(0.026)	1.382(0.037)	1.710(0.056)	1.423(0.026)
359987	shuttle	0.000(0.000)	0.000(0.000)	0.000(0.000)[6]	0.000(0.001)	0.000(0.000)	0.000(0.000)	0.001(0.000)	0.000(0.001)
359993	okcupid-...	0.559(0.009)	0.559(0.009)	0.559(0.009)	0.567(0.008)	0.563(0.007)	0.562(0.007)	0.570(0.008)	0.571(0.016)
360112	KDDCup99	0.001(0.000)	0.001(nan)[9]	-	-	-	0.000(0.000)	0.099(0.136)[2]	0.000(0.000)
7593	covertype	0.057(0.001)	0.059(0.002)	0.067(0.012)	0.147(0.003)	0.117(0.011)	0.068(0.002)	0.526(0.026)	0.253(0.208)

Table 8: Results for multiclass classification (in logloss) on a one hour budget, denoted as mean(std)[fails].

task id	framework task name	lightautoml	MLJAR(B)	MLJAR(P)	TPOT	constantpredictor	RandomForest	TunedRandomForest*
10090	amazon-c...	0.843(0.088)	1.169(0.133)	1.252(0.178)	1.111(0.268)[5]	3.912(0.000)	2.057(0.068)	1.462(0.087)
168784	steel-pl...	0.478(0.032)	0.468(0.026)	0.484(0.030)	0.509(0.044)[1]	1.671(0.007)	0.592(0.061)	0.540(0.047)
168909	dilbert	0.033(0.006)	0.028(0.012)	0.030(0.009)	0.150(0.072)	1.609(0.000)	0.329(0.010)	0.302(0.009)
168910	fabert	0.768(0.035)	0.755(0.033)[5]	0.771(0.024)	0.886(0.069)[3]	1.875(0.001)	0.808(0.030)	0.804(0.024)
189355	dionis	-	1.234(0.103)[5]	1.135(0.045)	4.241(0.403)[8]	5.857(0.000)	0.882(0.019)	0.872(0.020)
190146	vehicle	0.389(0.074)[5]	0.321(0.037)	0.349(0.034)	0.417(0.062)	1.386(0.000)	0.498(0.035)	0.484(0.040)
2073	yeast	1.040(0.092)[5]	1.006(0.086)	1.013(0.089)	1.056(0.055)[5]	1.727(0.013)	1.089(0.158)	1.139(0.157)
211979	jannis	0.666(0.005)	0.662(0.006)	0.672(0.005)	0.734(0.009)	1.109(0.000)	0.729(0.004)	0.729(0.005)
211986	Diabetes...	0.810(0.007)	0.829(0.006)	0.831(0.006)	0.846(0.008)	0.945(0.000)	0.853(0.004)	0.852(0.005)
359953	micro-ma...	0.280(0.098)	0.472(0.182)	0.383(0.144)	0.345(0.182)	2.914(0.014)	0.627(0.059)	0.469(0.092)
359954	eucalypt...	0.684(0.069)	0.653(0.060)	0.677(0.056)	0.712(0.041)	1.568(0.004)	0.747(0.056)	0.720(0.047)
359957	cnae-9	0.152(0.056)	0.182(0.085)	0.323(0.195)	0.146(0.065)	2.197(0.000)	0.301(0.037)	0.295(0.065)
359959	cmc	0.886(0.043)	0.890(0.054)	0.883(0.042)	0.901(0.057)	1.067(0.001)	1.046(0.108)	1.052(0.104)
359960	car	0.001(0.001)	0.003(0.002)	0.010(0.006)	0.788(1.301)	0.836(0.006)	0.105(0.010)	0.042(0.010)
359961	mfeat-fa...	0.080(0.030)	0.100(0.044)	0.096(0.029)	0.135(0.071)	2.303(0.000)	0.234(0.021)	0.204(0.026)
359963	segment	0.061(0.019)	0.057(0.021)	0.059(0.020)	0.075(0.032)	1.946(0.000)	0.096(0.022)	0.172(0.028)
359964	dna	0.109(0.025)	0.108(0.025)	0.119(0.023)	0.112(0.026)	1.026(0.001)	0.284(0.012)	0.170(0.027)
359969	first-or...	1.046(0.026)	1.034(0.028)	1.046(0.024)	1.081(0.040)	1.594(0.002)	1.189(0.111)	1.188(0.075)
359970	GestureP...	0.761(0.038)	0.723(0.037)	0.788(0.030)	0.870(0.029)	1.520(0.001)	0.907(0.018)	0.888(0.022)
359974	wine-qua...	0.786(0.016)[5]	0.770(0.026)	0.806(0.030)	0.792(0.043)[5]	1.291(0.005)	0.775(0.034)	0.787(0.058)
359976	Fashion-...	0.248(0.008)	0.249(0.010)	0.259(0.010)	0.431(0.014)	2.303(0.000)	0.361(0.005)	0.361(0.005)
359977	connect-4	0.335(0.007)	0.322(0.006)	0.342(0.005)	0.392(0.027)	0.845(0.000)	0.494(0.004)	0.477(0.005)
359981	jungle-c...	0.145(0.013)	0.090(0.016)	0.198(0.012)	1.766(3.333)	0.935(0.000)	0.438(0.009)	0.402(0.009)
359984	helena	2.555(0.017)	2.608(0.036)	2.653(0.062)	2.951(0.037)[1]	4.143(0.001)	3.334(0.021)	12.887(2.590)
359985	volkert	0.815(0.011)	0.794(0.013)	0.808(0.013)	1.013(0.034)[1]	2.053(0.000)	0.980(0.008)	0.936(0.007)
359986	robert	1.283(0.022)	1.361(0.026)	1.417(0.028)	1.956(0.147)	2.302(0.000)	1.687(0.015)	1.689(0.016)
359987	shuttle	0.001(0.000)	0.000(0.000)	0.000(0.000)	0.001(0.000)	0.666(0.001)	0.001(0.000)	0.001(0.000)
359993	okcupid-...	0.560(0.009)	0.564(0.008)	0.565(0.009)	0.571(0.006)	0.779(0.000)	0.594(0.005)	0.593(0.005)
360112	KDDCup99	-	0.000(0.000)[1]	0.105(nan)[9]	-	1.031(0.000)	0.000(0.000)	0.000(0.000)
7593	covertype	0.082(0.002)	0.082(0.001)	-	0.696(0.111)[2]	1.205(0.000)	0.161(0.001)	0.101(0.001)

Table 9: Results for multiclass classification (in logloss) on a one hour budget, denoted as mean(std)$^{\mathtt{fails}}$.

task id	framework task name	AutoGluon(B)	AutoGluon(HQ)	autosklearn	autosklearn2*	flaml	GAMA(B)*	H2OAutoML
10090	amazon-c...	0.673(0.072)	0.707(0.079)	1.138(0.130)	0.837(0.122)	1.101(0.150)	0.907(0.094)	1.077(0.092)
168784	steel-pl...	0.462(0.043)	0.463(0.048)	0.515(0.035)	0.472(0.029)	0.500(0.044)	0.491(0.039)	0.495(0.033)
168909	dilbert	0.013(0.005)	0.013(0.006)	0.029(0.010)	0.029(0.008)	0.024(0.011)	0.115(0.044)	0.032(0.010)
168910	fabert	0.682(0.030)	0.683(0.030)	0.748(0.029)	0.733(0.026)	0.774(0.031)	0.737(0.029)	0.726(0.027)
189355	dionis	0.285(0.004)	0.323(0.004)	0.544(0.064)	0.523(0.144)	0.335(0.004)[8]	1.587(0.244)	1.568(0.019)[2]
190146	vehicle	0.312(0.050)	0.342(0.061)	0.366(0.041)	0.329(0.030)	0.438(0.042)	0.369(0.032)	0.336(0.062)
2073	yeast	1.003(0.091)	1.008(0.093)	1.042(0.086)	1.015(0.084)	1.004(0.077)	1.019(0.081)[5]	1.042(0.094)
211979	jannis	0.648(0.006)	0.654(0.006)	0.664(0.006)	0.672(0.005)	0.671(0.006)	0.698(0.009)	0.664(0.006)
211986	Diabetes...	0.830(0.006)	0.830(0.006)	0.835(0.006)	0.832(0.005)	0.831(0.006)	0.837(0.006)	0.835(0.008)
359953	micro-ma...	0.209(0.083)	0.240(0.107)	0.280(0.106)	0.189(0.073)	0.284(0.114)	0.223(0.092)	0.345(0.154)
359954	eucalypt...	0.654(0.065)	0.675(0.083)	0.707(0.044)	0.704(0.061)	0.729(0.088)	0.700(0.057)	0.665(0.072)
359957	cnae-9	0.138(0.095)	0.163(0.102)	0.163(0.059)	0.143(0.043)	0.152(0.060)	0.132(0.044)	0.156(0.059)
359959	cmc	0.927(0.063)	0.937(0.066)	0.889(0.040)	0.884(0.037)	0.904(0.052)	0.893(0.043)	0.900(0.043)
359960	car	0.001(0.002)	0.000(0.001)	0.000(0.001)	0.002(0.004)	0.002(0.002)	0.012(0.008)	0.001(0.001)
359961	mfeat-fa...	0.070(0.027)	0.073(0.031)	0.084(0.037)	0.074(0.030)	0.090(0.036)	0.082(0.028)	0.115(0.064)
359963	segment	0.053(0.024)	0.056(0.027)	0.082(0.040)	0.062(0.026)	0.073(0.034)	0.067(0.026)	0.064(0.024)
359964	dna	0.106(0.030)	0.116(0.040)	0.116(0.029)	0.111(0.025)	0.108(0.027)	0.106(0.028)	0.117(0.029)
359969	first-or...	1.043(0.043)	1.042(0.040)	1.098(0.038)	1.041(0.030)	1.037(0.027)	1.052(0.027)	1.199(0.056)
359970	GestureP...	0.668(0.032)	0.671(0.031)	0.805(0.022)	0.768(0.029)	0.765(0.037)	0.807(0.039)	0.806(0.079)
359974	wine-qua...	0.699(0.028)	0.703(0.028)	0.793(0.041)	0.715(0.028)	0.726(0.041)	0.772(0.018)[5]	0.794(0.032)
359976	Fashion-...	0.218(0.008)	0.220(0.008)	0.242(0.008)	0.247(0.012)	0.245(0.010)	0.356(0.009)	0.265(0.009)
359977	connect-4	0.294(0.007)	0.310(0.012)	0.351(0.008)	0.342(0.013)	0.336(0.008)	0.346(0.028)	0.309(0.011)
359981	jungle-c...	0.010(0.002)	0.017(0.004)	0.158(0.032)	0.203(0.021)	0.211(0.006)	0.217(0.022)	0.108(0.012)
359984	helena	2.447(0.015)	2.460(0.016)	2.534(0.014)	2.485(0.031)	2.621(0.032)[1]	2.731(nan)[9]	2.790(0.028)
359985	volkert	0.672(0.010)	0.685(0.014)	0.790(0.012)	0.833(0.038)	0.802(0.020)	0.951(0.016)	0.778(0.014)
359986	robert	1.265(0.021)	1.281(0.024)	1.337(0.038)	1.383(0.031)	1.318(0.032)	1.617(0.030)	1.406(0.034)
359987	shuttle	0.000(0.000)	0.002(0.003)	0.001(0.001)	0.000(0.000)	0.000(0.000)	0.000(0.000)	0.001(0.001)
359993	okcupid-...	0.558(0.009)	0.559(0.008)	0.567(0.007)	0.563(0.008)	0.561(0.007)	0.568(0.007)	0.569(0.012)
360112	KDDCup99	0.001(0.000)	0.003(0.001)[1]	-	0.000(0.000)	0.000(0.000)[3]	-	0.000(0.000)
7593	covertype	-	0.053(0.001)	0.089(0.015)	0.095(0.010)	0.066(0.003)[1]	0.255(0.048)[4]	0.086(0.002)

Table 10: Results for multiclass classification (in logloss) on a four hour budget, denoted as mean(std)$^{\texttt{fails}}$. Results obtained in 2021 are denoted with a '*' next to the framework name.

task id	framework task name	lightautoml	MLJAR(B)*	TPOT*	constantpredictor	RandomForest	TunedRandomForest*
10090	amazon-c...	0.817(0.077)	1.181(0.132)	0.852(0.159)[2]	3.912(0.000)	2.057(0.068)	1.462(0.115)
168784	steel-pl...	0.475(0.030)	0.467(0.032)	0.486(0.021)	1.671(0.007)	0.592(0.061)	0.538(0.047)
168909	dilbert	0.025(0.009)	0.024(0.009)	0.060(0.022)	1.609(0.000)	0.329(0.010)	0.300(0.009)
168910	fabert	0.778(0.033)	0.752(0.025)	0.795(0.049)	1.875(0.001)	0.808(0.030)	0.809(0.030)
189355	dionis	-	-	-	5.857(0.000)	0.891(0.031)	0.870(0.017)
190146	vehicle	0.401(0.069)	0.321(0.043)	0.339(0.065)	1.386(0.000)	0.498(0.035)	0.483(0.039)
2073	yeast	1.041(0.091)[5]	1.004(0.085)	1.029(0.083)[5]	1.727(0.013)	1.089(0.158)	1.129(0.186)
211979	jannis	0.666(0.004)	0.658(0.005)	0.715(0.011)	1.109(0.000)	0.729(0.004)	0.719(0.005)
211986	Diabetes...	0.809(0.007)	0.828(0.006)	0.843(0.005)	0.945(0.000)	0.853(0.004)	0.852(0.005)
359953	micro-ma...	0.268(0.098)	0.460(0.202)	0.289(0.153)	2.914(0.014)	0.627(0.059)	0.463(0.086)
359954	eucalypt...	0.684(0.070)	0.646(0.054)	0.752(0.130)	1.568(0.004)	0.747(0.056)	0.717(0.047)
359957	cnae-9	0.149(0.059)	0.176(0.057)	0.150(0.077)	2.197(0.000)	0.301(0.037)	0.297(0.065)
359959	cmc	0.887(0.043)	0.888(0.054)	0.908(0.060)	1.067(0.001)	1.046(0.108)	1.047(0.135)
359960	car	0.002(0.002)	0.002(0.003)	1.450(3.004)	0.836(0.006)	0.105(0.010)	0.042(0.010)
359961	mfeat-fa...	0.081(0.029)	0.102(0.029)	0.108(0.042)	2.303(0.000)	0.234(0.021)	0.204(0.026)
359963	segment	0.060(0.020)	0.058(0.021)	0.071(0.032)	1.946(0.000)	0.096(0.022)	0.081(0.025)
359964	dna	0.109(0.025)	0.109(0.025)	0.112(0.025)	1.026(0.001)	0.284(0.012)	0.169(0.027)
359969	first-or...	1.047(0.026)	1.032(0.029)	1.062(0.035)	1.594(0.002)	1.189(0.111)	1.169(0.083)
359970	GestureP...	0.763(0.039)[5]	0.720(0.035)	0.834(0.044)	1.520(0.001)	0.907(0.018)	0.888(0.022)
359974	wine-qua...	0.785(0.015)[5]	0.755(0.030)	0.787(0.021)[5]	1.291(0.005)	0.775(0.034)	0.787(0.058)
359976	Fashion-...	0.247(0.008)	0.245(0.008)	0.415(0.029)	2.303(0.000)	0.361(0.005)	0.350(0.008)
359977	connect-4	0.321(0.006)	0.316(0.007)	0.378(0.023)	0.845(0.000)	0.494(0.004)	0.478(0.005)
359981	jungle-c...	0.104(0.012)	0.073(0.006)	1.078(2.885)	0.935(0.000)	0.438(0.009)	0.401(0.009)
359984	helena	2.511(0.015)	2.575(0.021)[1]	2.922(0.039)	4.143(0.001)	3.309(0.031)	10.877(2.059)
359985	volkert	0.788(0.016)	0.794(0.013)	0.973(0.040)[1]	2.053(0.000)	0.980(0.008)	0.929(0.018)
359986	robert	1.278(0.022)	1.306(0.029)	1.809(0.114)	2.302(0.000)	1.687(0.015)	1.661(0.015)
359987	shuttle	0.001(0.000)	0.000(0.000)	0.000(0.000)	0.666(0.001)	0.001(0.000)	0.001(0.000)
359993	okcupid-...	0.561(0.009)	0.563(0.008)	0.569(0.009)	0.779(0.000)	0.594(0.005)	0.593(0.005)
360112	KDDCup99	-	0.000(0.000)	-	1.031(0.000)	0.000(0.000)	0.000(0.000)
7593	covertype	0.069(0.001)	0.083(0.006)	0.459(0.139)	1.205(0.000)	0.161(0.001)	0.099(0.001)

Table 11: Results for multiclass classification (in logloss) on a four hour budget, denoted as mean(std)$^{\texttt{fails}}$. Results obtained in 2021 are denoted with a '*' next to the framework name.

task id	framework task name	AutoGluon(B)	AutoGluon(HQ)	AutoGluon(HQIL)	autosklearn	flaml	GAMA(B)	H2OAutoML
167210	Moneyball	21(0.87)	21(0.87)	21(0.87)	21(0.75)	22(0.62)	2e+10(6.3e+10)	21(0.93)
233211	diamonds	5e+02(19)	5.3e+02(36)	5.2e+02(22)	5.2e+02(21)	5.2e+02(19)	5.3e+02(21)[1]	5.2e+02(20)
233212	Allstate...	1.9e+03(44)	1.9e+03(36)	1.9e+03(39)	1.9e+03(58)	1.9e+03(54)	2e+03(54)	1.9e+03(48)
233213	Buzzinso...	1.5e+02(49)	1.5e+02(49)	1.5e+02(49)	1.6e+02(52)	1.5e+02(47)	1.6e+02(45)	1.5e+02(46)
233214	Santande...	6.8e+06(4.4e+05)	6.8e+06(6.4e+05)	7.1e+06(4.5e+05)	6.8e+06(5.1e+05)	7e+06(4.5e+05)	7e+06(4.8e+05)	7e+06(4.9e+05)
233215	Mercedes...	8.6(1)	8.8(1)	8.8(1)	8.3(1.1)	8.3(1.1)	8.3(1.1)	8.3(1.1)
317614	Yolanda	8.4(0.035)	8.5(0.045)	8.8(0.047)	8.7(0.061)	8.3(0.045)	9.4(0.11)	8.8(0.048)
359929	Airlines...	29(0.27)	29(0.28)	29(0.25)	29(0.27)	29(0.26)	29(0.26)	29(0.23)
359930	quake	0.19(0.0089)	0.22(0.012)	0.22(0.015)	0.19(0.0084)	0.19(0.0098)	0.19(0.0077)[2]	0.19(0.01)
359931	sensory	0.67(0.061)	0.68(0.057)	0.68(0.057)	0.7(0.064)	0.68(0.049)	0.68(0.058)	0.69(0.052)
359932	socmob	12(7.8)	13(8.1)	13(9)	12(5.4)	14(8.1)	15(8.1)	15(13)
359933	space_ga	0.095(0.014)	0.1(0.016)	0.11(0.025)	0.097(0.013)	0.1(0.016)	0.097(0.019)	0.097(0.016)
359934	tecator	0.84(0.19)	0.86(0.2)	1(0.2)	0.58(0.19)	0.93(0.22)	0.89(0.39)	0.82(0.3)
359935	wine_qua...	0.57(0.023)	0.57(0.024)	0.6(0.023)	0.61(0.019)	0.57(0.02)	0.57(0.021)	0.57(0.022)
359936	elevators	0.0018(5.2e-05)	0.0019(6.7e-05)	0.002(0.00024)	0.002(6.1e-05)	0.002(6e-05)	0.002(0.00011)[1]	0.0019(5e-05)
359937	black_fr...	3.4e+03(27)	3.5e+03(28)	3.4e+03(47)	3.4e+03(31)	3.4e+03(30)	3.5e+03(35)	3.4e+03(27)
359938	Brazilia...	2.3e+04(4.5e+04)	1.7e+04(3e+04)	1.1e+04(2e+04)	1.5e+03(4.7e+03)	1.3e+03(2.8e+03)	4.3(4.9)	3.2e+02(3e+02)
359939	topo_2.1	0.028(0.0049)	0.028(0.0048)	0.028(0.0048)	0.028(0.0049)	0.028(0.0048)	0.028(0.0049)	0.028(0.0048)
359940	yprop_4.1	0.028(0.0049)	0.028(0.0048)	0.028(0.0049)	0.028(0.0049)	0.028(0.0048)	0.028(0.0052)[1]	0.028(0.0048)
359941	OnlineNe...	3.3e+06(9.2e+06)	1.4e+04(7.7e+03)	1.1e+04(3.5e+03)	1.1e+04(3.6e+03)	1.1e+04(3.7e+03)	1.1e+04(3.7e+03)[1]	1.2e+04(3.6e+03)
359942	colleges	0.14(0.0056)	0.14(0.0059)	0.14(0.0069)	0.14(0.0058)	0.14(0.0062)	0.15(0.0047)[1]	0.14(0.0054)
359943	nyc-taxi...	1.5(0.15)	1.5(0.16)	1.7(0.12)	1.8(0.2)	1.6(0.12)	1.8(0.18)	1.7(0.13)
359944	abalone	2.1(0.12)	2.1(0.12)	2.2(0.11)	2.1(0.11)	2.1(0.12)	2.1(0.11)	2.1(0.11)
359945	us_crime	0.13(0.0055)	0.13(0.0074)	0.13(0.013)	0.13(0.0071)	0.13(0.0057)	0.13(0.007)	0.13(0.0069)
359946	pol	2.6(0.3)	2.8(0.38)	3.9(1)	3.8(0.34)	3.9(0.37)	4.1(0.34)	2.9(0.39)
359948	SAT11-HA...	8.9e+02(59)	8.8e+02(66)	9.6e+02(78)	1.1e+03(72)	1e+03(57)	1.2e+03(45)[1]	9.3e+02(59)
359949	house_sa...	1.1e+05(1.1e+04)	1.1e+05(1.1e+04)	1.1e+05(1.5e+04)	1.2e+05(1.5e+04)	1.1e+05(1.3e+04)	1.2e+05(1.5e+04)	1.1e+05(1.2e+04)
359950	boston	2.9(0.82)	2.7(0.74)	2.8(0.73)	3(1.1)	2.9(0.71)	3(0.94)	2.9(0.88)
359951	house_pr...	2.5e+04(7.6e+03)	2.6e+04(8.3e+03)	2.7e+04(6.9e+03)	2.6e+04(8.8e-03)	2.6e+04(7.8e-03)	2.7e+04(6.6e+03)	2.7e+04(9.1e+03)
359952	house_16H	2.8e+04(2.3e+03)	2.8e+04(2.6e+03)	2.9e+04(2.4e+03)	3e+04(2.3e-03)	2.9e+04(2e-03)	3e+04(2.3e+03)	2.9e+04(1.8e+03)
360932	QSAR-TID...	0.71(0.073)	0.72(0.074)	0.72(0.071)	0.78(0.059)	0.73(0.072)	0.76(0.075)	0.73(0.069)
360933	QSAR-TID...	0.69(0.022)	0.69(0.021)	0.7(0.026)	0.73(0.031)	0.69(0.021)	0.72(0.029)	0.69(0.025)
360945	MIP-2016...	2.1e+04(1.6e+03)	2.1e+04(1.6e+03)	2.1e+04(1.9e+03)	2.2e+04(1.2e+03)	2.2e+04(1.7e+03)	2.2e+04(1.4e+03)	2.1e+04(2.2e+03)

Table 12: Results for regression (in RMSE) on a one hour budget, denoted as `mean(std)`$^{\texttt{fails}}$.

task id	framework task name	lightautoml	MLJAR(B)	MLJAR(P)	TPOT	constantpredictor	RandomForest	TunedRandomForest*
167210	Moneyball	21(0.72)	21(0.83)	21(0.66)	21(1.1)[1]	92(1.9)	24(1.2)	24(1.2)
233211	diamonds	5.2e+02(21)	5.1e+02(21)	5.1e+02(21)	5.3e+02(22)	4.3e+03(67)	5.4e+02(20)	5.4e+02(19)
233212	Allstate...	1.9e+03(49)	1.9e+03(58)	1.9e+03(58)	2.1e+03(1.3e+02)[6]	3e+03(63)	2e+03(57)	1.9e+03(53)
233213	Buzzinso...	1.6e+02(45)	1.5e+02(53)	1.7e+02(52)	1.7e+02(39)[2]	6.3e+02(43)	1.5e+02(50)	1.5e+02(49)
233214	Santande...	6.9e+06(4.4e+05)	6.9e+06(4.5e+05)	6.9e+06(4.2e+05)	7e+06(4.7e+05)[1]	9e+06(6.5e+05)	7.2e+06(3.8e+05)	6.8e+06(4.5e+05)
233215	Mercedes...	8.3(1.1)	8.3(1.1)	8.3(1.1)	8.3(1.1)	13(0.72)	8.9(0.93)	8.9(0.95)
317614	Yolanda	8.6(0.037)	8.6(0.068)	8.6(0.067)	9.6(0.052)[3]	12(0.069)	9.1(0.037)	9.2(0.039)
359929	Airlines...	29(0.25)	29(0.25)	-	29(0.12)[3]	31(0.24)	30(0.24)	30(0.24)
359930	quake	0.19(0.01)	0.19(0.0093)	0.19(0.0095)	0.19(0.0091)	0.2(0.013)	0.2(0.0095)	0.2(0.009)
359931	sensory	0.68(0.064)	0.68(0.044)	0.68(0.056)	0.68(0.058)	0.82(0.11)	0.71(0.045)	0.7(0.061)
359932	socmob	20(9.8)	41(94)	21(31)	18(9.6)	43(9.9)	19(6.7)	19(7.6)
359933	space_ga	0.1(0.017)	0.099(0.018)	0.1(0.015)	0.1(0.018)	0.2(0.019)	0.11(0.022)	0.11(0.022)
359934	tecator	0.8(0.27)	0.9(0.26)	1(0.23)	0.67(0.35)	15(2.5)	1.4(0.21)	1.3(0.16)
359935	wine_qua...	0.59(0.022)	0.57(0.026)	0.59(0.022)	0.58(0.015)	0.89(0.025)	0.59(0.023)	0.58(0.022)
359936	elevators	0.002(5.7e-05)	0.0019(6e-05)	0.002(6.1e-05)	0.002(8e-05)	0.007(0.00026)	0.0027(9.4e-05)	0.0026(8.8e-05)
359937	black.fr...	3.4e+03(28)	3.4e+03(31)	3.4e+03(29)	3.5e+03(29)	5.1e+03(44)	3.7e+03(30)	3.7e+03(29)
359938	Brazilia...	3.8(5)	2.6e+10(8.1e+10)	2.5e+13(7.9e+13)	3.9(5)	1.2e+04(1.2e+04)	4e+03(5e+03)	4.1e+03(5.2e+03)
359939	topo_2.1	0.028(0.0049)	0.028(0.0049)	0.028(0.0049)	0.028(0.0049)	0.03(0.0048)	0.028(0.0047)	0.028(0.0048)
359940	yprop_4.1	0.028(0.0049)	0.028(0.0049)	0.028(0.0049)	0.028(0.0049)	0.03(0.0048)	0.028(0.0048)	0.028(0.0048)
359941	OnlineNe...	1.1e+04(3.5e+03)	1.1e+04(3.7e+03)	1.5e+04(1.2e+04)	1.1e+04(3.7e+03)	1.1e+04(3.6e+03)	1.2e+04(3.5e+03)	1.1e+04(3.7e+03)
359942	colleges	0.14(0.0054)	0.14(0.0063)	0.14(0.0061)	0.14(0.0063)	0.23(0.004)	0.15(0.006)	0.14(0.0053)
359943	nyc-taxi...	1.7(0.17)	1.6(0.17)	1.7(0.15)	1.9(0.1)	2.7(0.19)	1.7(0.12)	1.7(0.13)
359944	abalone	2.1(0.12)	2.1(0.12)	2.1(0.13)	2.1(0.11)	3.3(0.16)	2.2(0.097)	2.1(0.11)
359945	us_crime	0.13(0.0056)	0.13(0.0058)	0.13(0.0066)	0.13(0.008)	0.25(0.025)	0.14(0.006)	0.14(0.0072)
359946	pol	3.9(0.34)	2.3(0.29)	2.4(0.26)	4(0.32)	51(1.2)	4.5(0.42)	4.3(0.36)
359948	SAT11-HA...	1.2e+03(88)	1e+03(91)	1.2e+03(96)	1.1e+03(77)	2.8e+03(98)	1.1e+03(50)	1.1e+03(49)
359949	house_sa...	1.1e+05(1.6e+04)	1.1e+05(1.2e+04)	1.1e+05(1.3e+04)	1.2e+05(2e+04)	3.8e+05(3.7e+04)	1.3e+05(1.5e+04)	1.3e+05(1.5e+04)
359950	boston	2.9(0.95)	3(0.95)	2.9(0.88)	3.1(0.91)	9.1(2)	3.1(0.66)	3.1(0.91)
359951	house_pr...	2.6e+04(7.2e+03)	2.4e+04(6.9e+03)	2.5e+04(6.8e+03)	2.7e+04(6.2e+03)	8.1e+04(6.3e+03)	2.8e+04(6e+03)	2.8e+04(6.6e+03)
359952	house_16H	2.9e+04(2.1e+03)	2.9e+04(2.1e+03)	2.9e+04(2e+03)	3.1e+04(1.8e+03)	5.5e+04(2.7e+03)	3.2e+04(2e+03)	3.1e+04(2e+03)
360932	QSAR-TID...	0.72(0.068)	-	-	0.75(0.075)	1.6(0.048)	0.76(0.066)	0.75(0.065)
360933	QSAR-TID...	0.69(0.021)	0.7(0.024)[5]	0.72(nan)[9]	0.72(0.03)	1.3(0.039)	0.72(0.028)	0.71(0.026)
360945	MIP-2016...	2.2e+04(1.1e+03)	2.2e+04(2e+03)	2.4e+04(2.4e+03)	2.2e+04(1.4e+03)[2]	3.2e+04(2.2e+03)	2.3e+04(2.2e+03)	2.3e+04(2.1e+03)

Table 13: Results for regression (in RMSE) on a one hour budget, denoted as mean(std). [fails]

47

task id	framework task name	AutoGluon(B)	AutoGluon(HQ)	AutoGluon(HQIL)	autosklearn	flaml	GAMA(B)*	H2OAutoML
167210	Moneyball	21(0.87)	21(0.87)	21(0.87)	21(0.73)	22(0.86)	21(0.77)	21(0.85)
233211	diamonds	5e+02(18)	5.3e+02(26)	5.2e+02(22)	5.2e+02(20)	5.2e+02(20)	5.2e+02(20)	5.1e+02(18)
233212	Allstate...	1.9e+03(40)	1.9e+03(39)	1.9e+03(1.3e+02)	1.9e+03(59)	1.9e+03(55)	8.6e+12(2.7e+13)	1.9e+03(59)
233213	Buzzinso...	1.5e+02(49)	2.3e+02(2.9e+02)	2.3e+02(2.9e+02)	1.5e+02(54)	1.5e+02(52)	1.6e+02(47)	1.5e+02(47)
233214	Santande...	6.8e+06(4.6e+05)	6.8e+06(4.7e+05)	7e+06(4.3e+05)	6.9e+06(5.6e+05)	6.9e+06(4.8e+05)	6.9e+06(4.3e+05)	6.9e+06(4.8e+05)
233215	Mercedes...	8.6(1)	8.8(1)	8.8(0.97)	8.3(1.1)	8.3(1.1)	8.3(1.1)	8.3(1.1)
317614	Yolanda	8.3(0.033)	8.4(0.033)	8.6(0.055)	8.7(0.063)	8.6(nan)[9]	9.2(0.1)	8.6(0.034)
359929	Airlines...	29(0.26)[2]	29(0.23)	29(0.23)	29(0.27)	28(0.23)	29(0.25)	29(0.24)
359930	quake	0.19(0.0089)	0.22(0.012)	0.22(0.015)	0.19(0.009)	0.19(0.0092)	0.19(0.0092)	0.19(0.0096)
359931	sensory	0.67(0.061)	0.68(0.057)	0.68(0.057)	0.69(0.062)	0.68(0.054)	0.68(0.055)	0.69(0.059)
359932	socmob	12(7.7)	13(8.1)	14(9.3)	12(6.4)[1]	15(8.4)	14(7)	14(13)
359933	space_ga	0.095(0.014)	0.1(0.016)	0.11(0.016)	0.1(0.018)	0.1(0.014)	0.096(0.019)	0.094(0.011)
359934	tecator	0.84(0.18)	0.85(0.19)	1.1(0.32)	0.61(0.2)	0.92(0.29)	0.82(0.33)	0.71(0.17)
359935	wine_qua...	0.57(0.022)	0.57(0.024)	0.6(0.021)	0.6(0.018)	0.57(0.02)	0.57(0.022)	0.57(0.022)
359936	elevators	0.0018(5.3e-05)	0.0019(0.00014)	0.002(0.00024)	0.0019(9.6e-05)	0.0019(5.6e-05)	0.0019(6.5e-05)	0.0019(5.7e-05)
359937	black_fr...	3.5e+03(28)	3.5e+03(31)	3.4e+03(28)	3.4e+03(31)	3.4e+03(28)	3.5e+03(29)[1]	3.4e+03(28)
359938	Brazilia...	8.6e+04(1.9e+05)	7.4e+04(1.7e+05)	1.5e+04(3.3e+04)	1.5e+03(4.7e-03)	1.8e+03(3e+03)	4.4(4.9)	3.4e+02(5e+02)
359939	topo_2.1	0.028(0.0048)	0.028(0.0048)	0.028(0.0048)	0.028(0.0049)	0.028(0.0048)	0.028(0.0048)	0.028(0.0049)
359940	yprop_4.1	0.028(0.0049)	0.028(0.0047)	0.028(0.0049)	0.028(0.0048)	0.028(0.0049)	0.028(0.0049)	0.028(0.0048)
359941	OnlineNe...	4.1e+06(9.1e+06)	1.3e+04(4.6e+03)	1.3e+04(4.6e+03)	1.1e+04(3.7e+03)	1.1e+04(3.7e+03)	1.1e+04(3.7e+03)	2.2e+04(3.3e+04)
359942	colleges	0.14(0.0059)	0.14(0.0061)	0.14(0.0061)	0.14(0.0058)	0.14(0.0058)	0.14(0.0048)	0.14(0.0062)
359943	nyc-taxi...	1.7(0.66)	3(4.2)	1.7(0.12)	1.6(0.15)	1.6(0.1)	1.7(0.09)[6]	1.6(0.15)
359944	abalone	2.1(0.12)	2.1(0.12)	2.2(0.11)	2.1(0.12)	2.1(0.11)	2.1(0.1)	2.1(0.11)
359945	us_crime	0.13(0.0055)	0.13(0.0068)	0.13(0.013)	0.13(0.0063)	0.13(0.0049)	0.13(0.0059)	0.13(0.0071)
359946	pol	2.6(0.31)	2.8(0.4)	3.9(1)	3.5(0.34)	3.6(0.34)	3.7(0.3)	2.9(0.31)
359948	SAT11-HA...	8.8e+02(66)	8.8e+02(65)	1.1e+03(62)	1.1e+03(72)	1e+03(66)	1.1e+03(63)	9e+02(58)
359949	house_sa...	1.1e+05(1.1e+04)	1.1e+05(1.2e+04)	1.1e+05(1.5e+04)	1.2e+05(1.7e+04)	1.1e+05(1.4e+04)	1.1e+05(1.6e+04)	1.1e+05(1.2e+04)
359950	boston	2.9(0.82)	2.7(0.74)	2.8(0.73)	2.9(1.2)	2.9(0.73)	3(0.99)	2.7(0.87)
359951	house_pr...	2.5e+04(7.8e+03)	2.6e+04(8e+03)	2.8e+04(1.1e+04)	2.6e+04(1e+04)	2.4e+04(5.9e+03)	2.4e+04(5.9e+03)	2.7e+04(1.1e+04)
359952	house_16H	2.8e+04(2.3e+03)	2.8e+04(2.5e+03)	3e+04(2.8e+03)	3e+04(2.1e+03)	2.9e+04(2e+03)	3e+04(2.2e+03)	2.9e+04(1.8e+03)
360932	QSAR-TID...	0.71(0.076)	0.72(0.078)	0.76(0.079)	0.77(0.064)	0.72(0.072)	0.75(0.067)	0.72(0.07)
360933	QSAR-TID...	0.69(0.022)	0.69(0.02)	0.72(0.023)	0.73(0.033)	0.68(0.025)	0.71(0.025)	0.69(0.027)
360945	MIP-2016...	2.1e+04(1.5e+03)	2.1e+04(1.7e+03)	2.1e+04(1.6e+03)	2.2e+04(1.5e+03)	2.1e+04(1.6e+03)	2.1e+04(1.4e+03)	2.1e+04(1.7e+03)

Table 14: Results for regression (in RMSE) on a four hour budget, denoted as mean(std)$^{\texttt{fails}}$. Results obtained in 2021 are denoted with a '*' next to the framework name.

task id	framework task name	lightautoml	MLJAR(B)	TPOT*	constantpredictor	RandomForest	TunedRandomForest*
167210	Moneyball	21(0.73)	21(0.84)	21(0.87)	92(1.9)	24(1.2)	24(1.3)
233211	diamonds	5.2e+02(21)	5.1e+02(22)	5.3e+02(23)	4.3e+03(67)	5.4e+02(20)	5.4e+02(20)
233212	Allstate...	1.9e+03(49)	1.9e+03(54)	1.9e+03(50)	3e+03(63)	2e+03(57)	2e+03(79)
233213	Buzzinso...	1.6e+02(46)	1.5e+02(47)	1.6e+02(46)	6.3e+02(43)	1.5e+02(49)	1.5e+02(50)
233214	Santande...	7.1e+06(2.7e+05)[5]	6.9e+06(4.3e+05)	7e+06(4.4e+05)	9e+06(6.5e+05)	7.2e+06(3.8e+05)	6.8e+06(4.6e+05)
233215	Mercedes...	8.3(1.1)	8.3(1.1)	8.3(1.1)	13(0.72)	8.9(0.93)	8.9(0.96)
317614	Yolanda	8.6(0.041)	8.5(0.045)	9.3(0.11)	12(0.069)	9.1(0.038)	9.1(0.04)
359929	Airlines...	29(0.24)	29(0.23)	29(0.28)[4]	31(0.24)	30(0.24)	30(0.23)
359930	quake	0.19(0.01)	0.19(0.0084)	0.19(0.0096)	0.2(0.013)	0.2(0.0095)	0.2(0.0088)
359931	sensory	0.68(0.057)	0.68(0.046)	0.68(0.054)	0.82(0.11)	0.71(0.045)	0.7(0.057)
359932	socmob	20(9.4)	40(91)	16(8)	43(9.9)	19(6.7)	19(7.3)
359933	space_ga	0.1(0.017)	0.098(0.017)	0.099(0.018)	0.2(0.019)	0.11(0.022)	0.11(0.022)
359934	tecator	0.79(0.26)	0.92(0.18)	0.56(0.094)[1]	15(2.5)	1.4(0.21)	1.3(0.17)
359935	wine_qua...	0.58(0.022)	0.57(0.023)	0.57(0.023)	0.89(0.025)	0.59(0.023)	0.58(0.022)
359936	elevators	0.002(5.7e-05)	0.0019(5.9e-05)	0.0019(6.4e-05)	0.007(0.00026)	0.0027(9.4e-05)	0.0026(9e-05)
359937	black_fr...	3.4e+03(27)	3.4e+03(28)	3.5e+03(30)	5.1e+03(44)	3.7e+03(30)	3.8e+03(37)
359938	Brazilia...	3.9(5)	1.1e+11(3.3e+11)	4.2(4.9)	1.2e+04(1.2e+04)	4e+03(5e+03)	4.1e+03(5.1e+03)
359939	topo_2.1	0.028(0.0049)	0.028(0.0048)	0.028(0.0048)	0.03(0.0048)	0.028(0.0047)	0.028(0.0047)
359940	yprop_4.1	0.028(0.0049)	0.028(0.0049)	0.028(0.0048)	0.03(0.0048)	0.028(0.0048)	0.028(0.0049)
359941	OnlineNe...	1.1e+04(3.7e+03)	4.9e+05(1.5e+06)	1.1e+04(3.7e+03)	1.1e+04(3.6e+03)	1.2e+04(3.5e+03)	1.1e+04(3.7e+03)
359942	colleges	0.14(0.0054)	0.14(0.0062)	0.14(0.005)	0.23(0.004)	0.15(0.006)	0.14(0.0054)
359943	nyc-taxi...	1.7(0.17)	1.6(0.13)	1.8(0.16)	2.7(0.19)	1.7(0.12)	1.7(0.16)
359944	abalone	2.1(0.12)	2.1(0.12)	2.1(0.11)	3.3(0.16)	2.2(0.097)	2.1(0.11)
359945	us_crime	0.13(0.006)	0.13(0.0057)	0.13(0.0058)	0.25(0.025)	0.14(0.006)	0.14(0.0064)
359946	pol	3.9(0.34)	2.3(0.26)	3.7(0.38)	51(1.2)	4.5(0.42)	4.3(0.36)
359948	SAT11-HA...	1.2e+03(95)	1.1e+03(1e+02)	1e+03(69)	2.8e+03(98)	1.1e+03(50)	1.1e+03(49)
359949	house_sa...	1.1e+05(1.6e+04)	1.1e+05(1.3e+04)	1.2e+05(1.2e+04)	3.8e+05(3.7e+04)	1.3e+05(1.5e+04)	1.3e+05(1.5e+04)
359950	boston	2.9(0.94)	3(0.9)	3.1(1)	9.1(2)	3.1(0.66)	3.1(0.92)
359951	house_pr...	2.5e+04(6.6e+03)	2.4e+04(7.1e+03)	2.7e+04(7.8e+03)	8.1e+04(6.3e+03)	2.8e+04(6e+03)	2.8e+04(6.5e+03)
359952	house_16H	2.9e+04(2.1e+03)	2.9e+04(2e+03)	3.1e+04(2.1e+03)	5.5e+04(2.7e+03)	3.2e+04(2e+03)	3.1e+04(2e+03)
360932	QSAR-TID...	0.72(0.068)	-	0.74(0.069)	1.6(0.048)	0.76(0.066)	0.75(0.066)
360933	QSAR-TID...	0.69(0.021)	0.7(0.02)[3]	0.71(0.027)	1.3(0.039)	0.72(0.028)	0.71(0.027)
360945	MIP-2016...	2.2e+04(1.2e+03)	2.2e+04(1.8e+03)	2.2e+04(2.2e+03)	3.2e+04(2.2e+03)	2.3e+04(2.2e+03)	2.3e+04(2e+03)

Table 15: Results for regression (in RMSE) on a four hour budget, denoted as mean(std)$^{\text{fails}}$. Results obtained in 2021 are denoted with a '*' next to the framework name.

49

B.1 BT-Trees

As described in Section 6.2, Bradley-Terry (BT) trees may be used to identify subsets of tasks for which the 'preferred' framework is significantly different. Figures 10-12 show BT trees for each task type and time budget, generated by splitting based on different data set characteristics. We observe that AUTOGLUON(B) is the preferred framework in many cases, especially for large number of observations, and more complex data sets in terms of class imbalance, number of classes or missing data. In contrast, differences in small datasets are generally smaller, and distinct preferred frameworks often emerge for various task types. The online visualization tool may be used to generate additional BT trees, from different subsets of tasks or allowing for different meta-features when calculating splits.
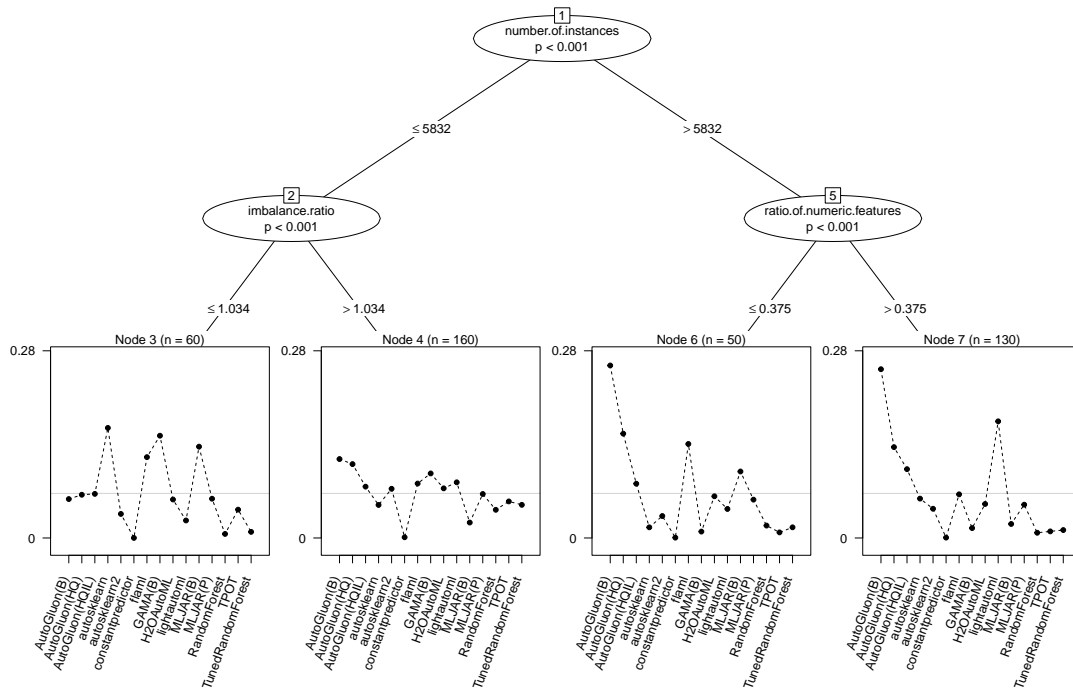
(a) Binary classification datasets, 1h.
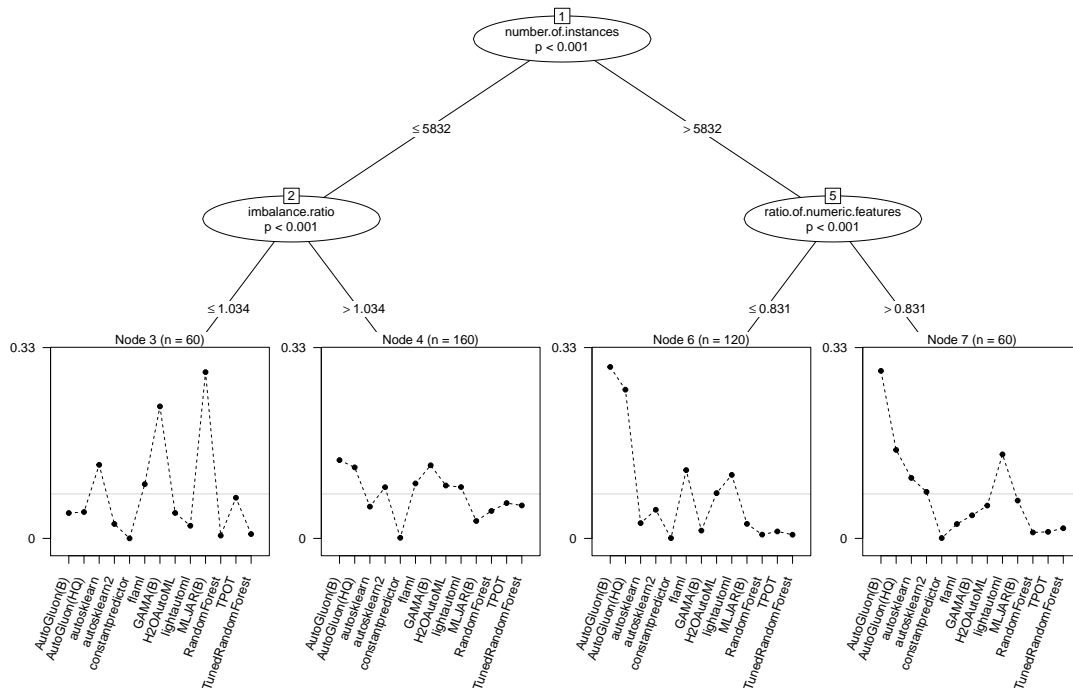
(b) Binary classification datasets, 4h.

Figure 10: Binary classification datasets, 1h and 4h.
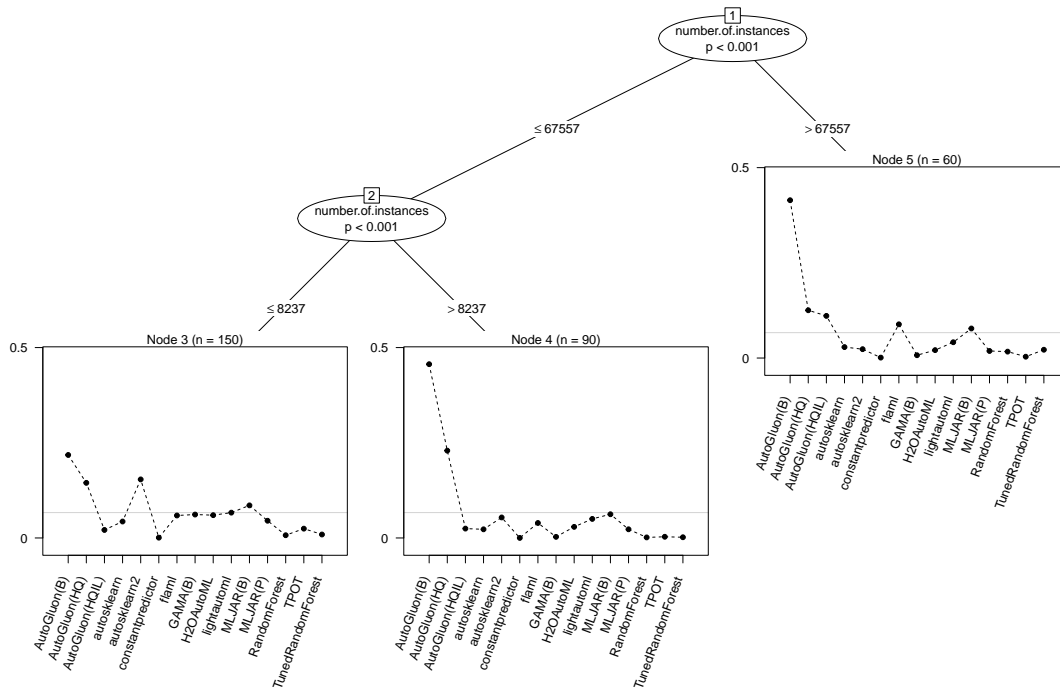
(a) Multiclass classification datasets, 1h.
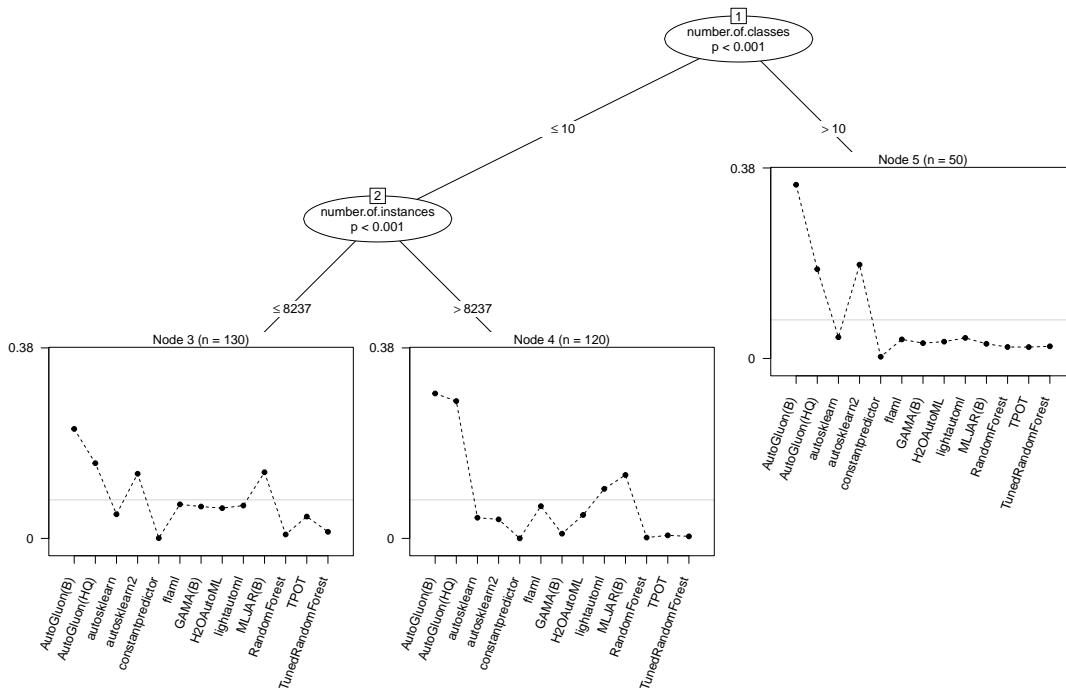
(b) Multiclass classification datasets, 4h.

Figure 11: Multiclass classification datasets, 1h and 4h.
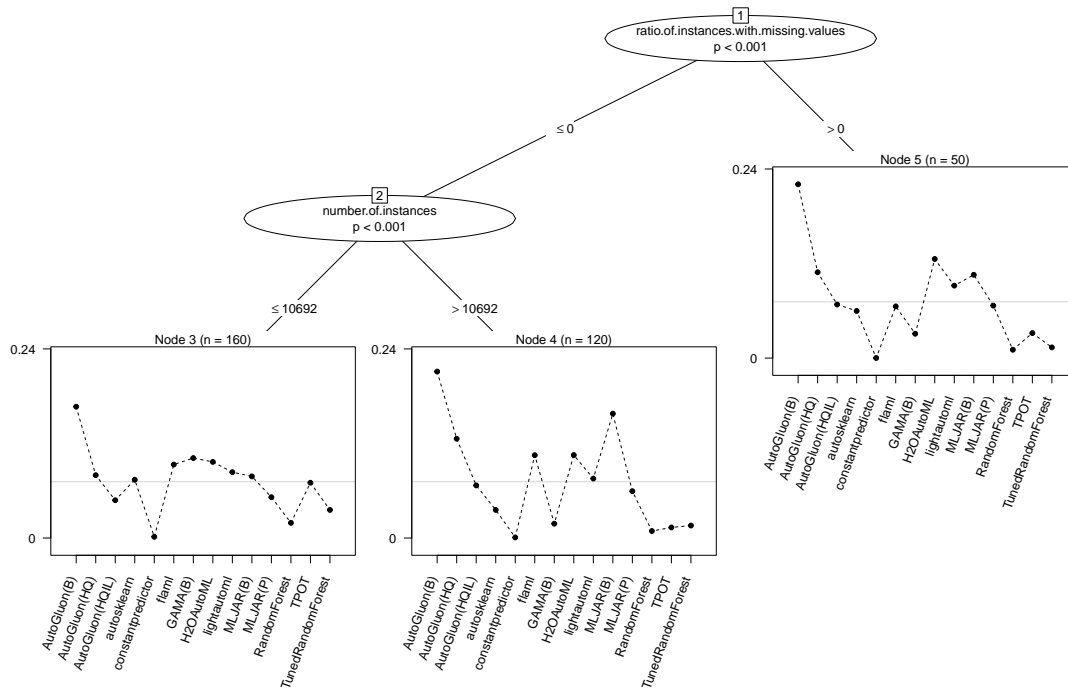
(a) Regression datasets, 1h.
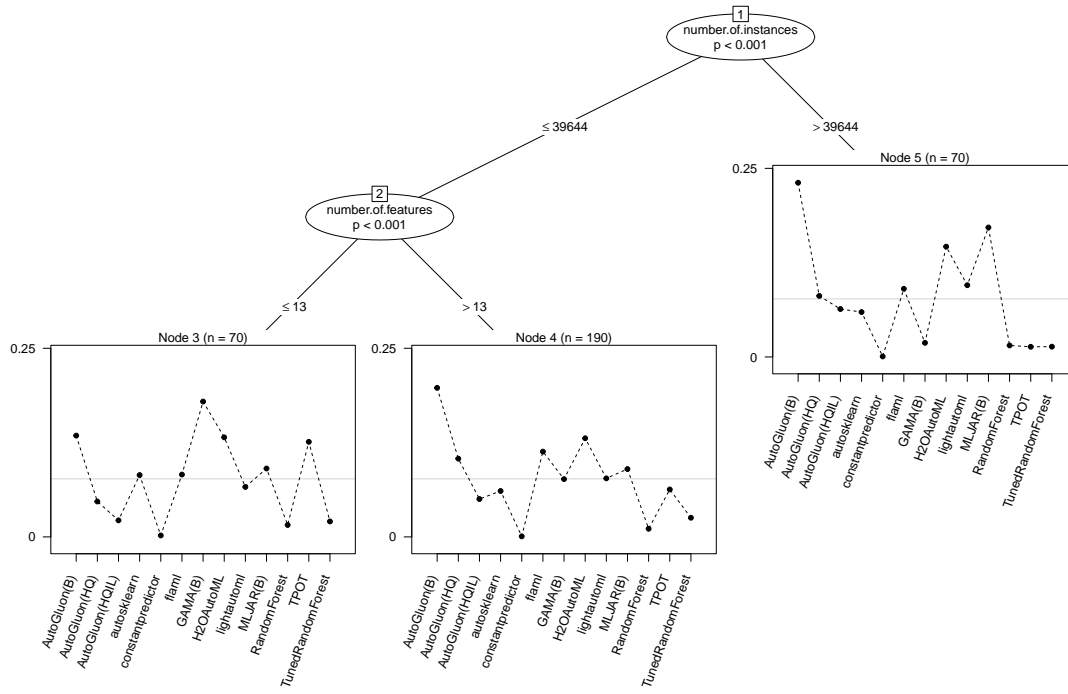
(b) Regression datasets, 4h.

Figure 12: Regression datasets, 1h and 4h.

Appendix C. Software

This appendix contains additional details about the developed and used software.

C.1 Architecture Overview

Figure 13 shows the architecture and information flow when running the benchmarking tool in the setup used in the experiments for this paper. The dotted lines indicate calls (communication) whereas the solid lines indicate transfer of files. In this paper we distribute our workload to AWS EC2 containers which run experiments within a docker environment.

The experiments are initiated from a 'local' machine, shown enclosed in a blue rectangle in the bottom. This local instantiation first reads information from a local configuration file and the command line, after which it uses boto3 to connect to AWS. It uploads local files required for the experiments, such as the configuration files, to an S3 bucket accessible by the EC2 instances, deploys the jobs to EC2 instances, and tracks the instance status through CloudWatch.

The EC2 instance then installs the specified version of the benchmarking tool (here of type m5.2xlarge), retrieves the user configuration from the S3 bucket, the specified docker image from the docker hub, and the dataset (task) from OpenML. The experiment is then run in a docker container (which in turn uses the benchmarking software in 'local' mode) which has the benefits of reproducibility of the software stack and requiring much less time than installing the framework from scratch.

After the experiment has completed, results are uploaded to an S3 bucket and the EC2 instance terminates. The local machine will observe this shutdown and subsequently fetch the downloaded results from the S3 bucket.

As you can see, with this setup there are almost no requirements to the local machine. Provided the docker images are built and published, the specified benchmark tool version is available online and OpenML is used, the only data transfer between the local machine and the cloud is the upload of configuration files and the download of results.

C.2 Framework Versions

Experimental results in this paper are mostly from the latest version of each AutoML framework as of early June 2023. Our original submission contained results from frameworks as of September 2021. Because of the delay between our 2021 experiments and submission as well as the delay between our submission and the reviews, and the additional presets we wanted to evaluate, we felt it best to redo our all of our experimental evaluations to reflect a more recent state of the AutoML frameworks. However, because of budget constraints we could not re-evaluate everything so we chose to re-use some of our old experimental results instead. In the text, these results are denoted by an asterisk (*). We only used old data for frameworks which did not have many updates between the two experimental evaluations.

Below are the exact versions of the frameworks used in both the 2021 and 2023 evaluations. Presets with a $'23$ are only evaluated in 2023. The specific version number of each framework can be seen in Table 16. For each framework, the latest available version as of the 2nd of April 2024 is also shown.

C.2.1 AutoGluon

With the 'best quality' preset, the models used in the final ensembles are the same ones created to produce out-of-bag predictions during the optimization step. This means that the same algorithm and configuration is represented multiple times, but trained on different subsets of the training data. For the 'high quality' preset, instead of using these multiple models a single model for the given algorithm and configuration is trained instead. This typically results in lower predictive performance, but faster inference times. When the 'limited inference' constraint is applied, first all models which have too slow inference time are discarded, but otherwise the procedure continues as normal.

AUTOGLUON's 'high quality' presets execute a post-processing step, i.e., refitting the models, after search regardless of elapsed time. Because this possible time constraint violation is by design, we agreed with the authors to we reduce the time constraint communicated to the framework for those presets by 10%. With this adjustment, we expected the final training time to observe our time constraints better. For NAIVEAUTOML we disable early-stopping as detailed in Appendix E.
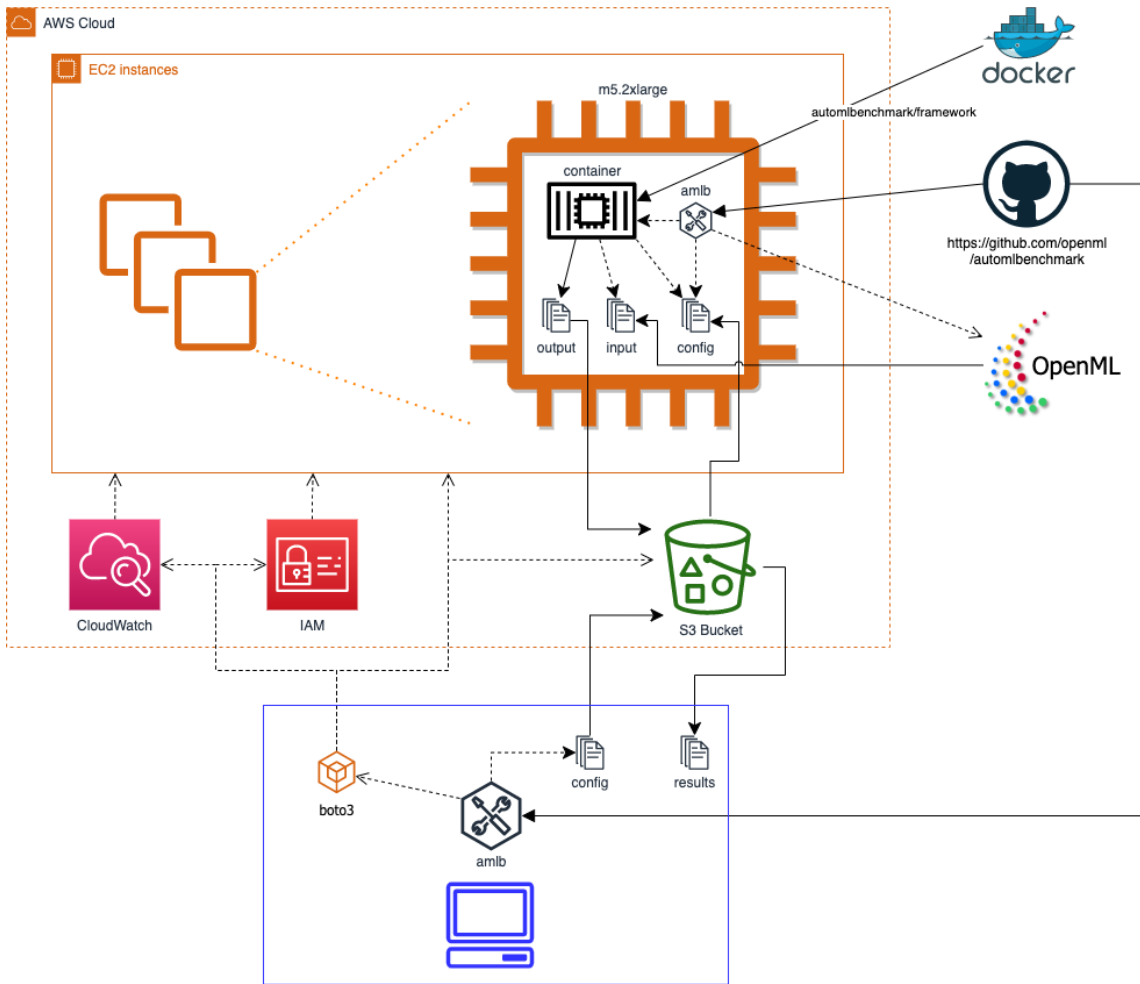
Figure 13: Architecture Overview of the AWS+docker mode, as used for this paper

C.2.2 TPOT

TPOT requires encoded data to function. The AutoML benchmark provides this encoded data, but the time to encode the data is not include in training time or inference time measurements. For this reason, TPOT's inference time is not reported on in this paper.

framework	2021	2023	latest	notes
AUTOGLUON	0.3.1	0.8.0	1.0.0	'best quality' (B), 'high quality'$'^{23}$ (HQ), and 'HQ with limited inference time'$'^{23}$ (HQIL)
AUTO-SKLEARN	0.14.0	0.15.0	0.15.0	
AUTO-SKLEARN 2	0.14.0	0.15.0	0.15.0	
FLAML	0.6.2	1.2.4	2.1.2	
GAMA	21.0.1	23.0.0	23.0.0	'performance' preset
H2O AUTOML	3.34.0.1	3.40.0.4	3.46.0.1	
LIGHTAUTOML	0.2.16	0.3.7.3	0.3.8.1	
MLJAR	0.11.0	0.11.5	1.1.6	'compete' and 'perform'$'^{23}$ preset
NAIVEAUTOML	-	0.0.27	0.0.27	
TPOT	0.11.7	0.12.0	0.12.2	

Table 16: Used AutoML framework versions in the experiments.
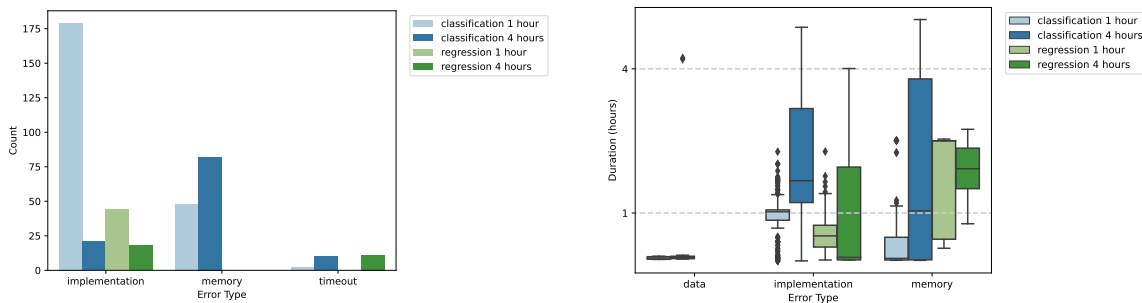
(a) Error count by category and time constraint.

(b) Distribution of when errors occurred.

Figure 14: Figure denoting for each benchmarking suite and time constraint how often errors occurred (on the left) and when (on the right).

Appendix D. AutoML Framework Errors

This appendix contains additional information on the AutoML framework errors encountered while running the experiments. In the results below, the 'fixed' category from Section 6.4 is simply included in the 'implementation' error category. More information on this particular error can be found later in section D.3 of this appendix.

Figure 14a shows the number of errors encountered across different time constraints and benchmarking suites. In this comparison, we only report on frameworks which had both a one hour and four hour evaluation in 2023. As mentioned in Section 6.4, the amount of memory an timeout errors increase with higher time budgets. However, the number of implementation errors decrease, lowering the total amount of errors encountered overall. We suspect a number of implementation errors on the one hour constraints are due to the framework aborting its pipeline optimization early to adhere to the time constraint while not being prepared to produce predictions yet. Figure 14b, which shows the time at which various errors occurred, supports this hypothesis for the one hour classification tasks where the majority of implementation errors occur. With a more generous time budget, the AutoML framework might be able to fall back on already optimized models, which could explain why we see fewer implementation errors under a larger time constraint.

D.1 Class Imbalance

The two classification tasks with a large amount of failures despite being small are 'yeast' and 'wine-quality-white', which feature a minority class with only 5 instances. This means that within the 10-fold cross-validation we perform in our experiments, either 4 or 5 of those instances are available in the training splits. We see that only in the case where one of those samples is in the test set failures occur. The exact error message differs per framework, though they indicate that evaluating pipelines fails. This is likely due to e.g., using 5-fold cross-validation out of the box. Failure on these specific datasets (and folds) is only observed for `GAMA`, `LightAutoML`, and `TPOT`.

D.2 MLJarSupervised

`MLJarSupervised`(B) the most 'implementation errors', more than twice the next framework. Of its 284 errors, 256 failures are caused by variations of the following three unique errors:

25 times `['Ensemble_prediction_0_for_neg_1_for_pos', ...,`
`'2_DecisionTree_prediction_0_for_neg_1_for_pos'] not in index"`

149 times `catboost/libs/data/model_dataset_compatibility.cpp:81:`
`At position 6 should be feature with name 60_NeuralNetwork_prediction_0_for_1_1_for_2`
`(found 60_NeuralNetwork_prediction).`

82 times `The feature names should match those that were passed during fit.  Feature names unseen`
`        at fit time:- 100_Xgboost_prediction-...`

This is specific to `MLJarSupervised`. While we can only guess, we assume it is related to the extensive AutoML pipeline `MLJarSupervised` has. It includes 10 different steps, including three steps for feature generation and selection and three steps for ensembling and stacking. These steps are not turned on by default[23]. Since these three errors only occur in `MLJARSupervised` in 'compete' mode and not in 'performance' mode, it likely stems from the ensembling step, which is only used in 'compete'.

D.3 AutoGluon

In the version of `AUTOGLUON`(HQIL) that we benchmarked (0.8.0), there was a bug in the newly added model calibration which made it crash on some tasks (and only under one hour constraints). However, this bug has since been fixed. We also confirmed that in the version with the fix, `AUTOGLUON`(HQIL) performs very similar to the `AUTOGLUON`(HQ) of version 0.8.0. Because the bug had been fixed and a very accurate imputation strategy was available, we found it more reasonable to impute missing values of `AUTOGLUON`(HQIL) which were specifically caused by this bug with values obtained by `AUTOGLUON`(HQ) instead.

---

23. `https://supervised.mljar.com/features/modes/`

Appendix E. Naive AutoML

NAIVEAUTOML is introduced as a baseline to compare AutoML frameworks to (Mohr and Wever, 2023). It designs the machine learning pipeline one step at a time in sequence, ignoring the fact that better performance may be obtained by performing joint algorithm selection and hyperparameter optimization over the whole machine learning pipeline at once. It was not designed to find the best possible model, but instead to provide an adequate model fast. To this end, it employs an aggressive early-stopping heuristic.

We developed the NAIVEAUTOML integration together with the framework authors. We changed a few hyperparameters from their default configuration in order to disable NAIVEAUTOML's early stopping. This allows for comparisons of models produced under similar time constraints. The integration was developed with version 0.0.16, which proved too unstable to benchmark. The experiments in this appendix ran on version 0.0.27. We changed the following hyperparameters from their default configuration:

- MAX_HPO_ITERATIONS defines after how many hyperparameter optimization iterations of no improvement NAIVEAUTOML should stop. The default is 100, but we effectively disable early stopping by setting it to 10^{10} iterations.

- EXECUTION_TIMEOUT defines how long a single evaluation at any step in the process may take (such as evaluating a random forest model). The default was set to 10 seconds in version 0.0.16, which is too short to evaluate models on larger datasets, so we set it to 5% of the total allowed time (that is, 3 minutes for a 1 hour timeout). A later version of NAIVEAUTOML changed the default to 5 minutes, but we were not aware of this change and the integration script was not changed to reflect that. As we will see below, the 5% limit was too strict.

We evaluated NAIVEAUTOML on a one hour budget for both the classification and regression benchmark suites. Figure 15 shows NAIVEAUTOML achieving poor performance compared to other AutoML frameworks. This low performance contrasts the evaluation by Mohr and Wever (2023), where it achieves performance similar to AUTO-SKLEARN.
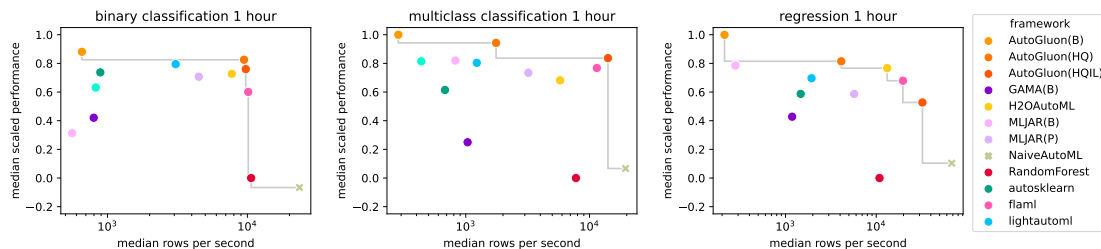

Figure 15: Performance and inference time trade-off as in Figure 7 of Section 6.3, but with NAIVEAUTOML results included. NAIVEAUTOML has bad performance but fast inference speeds due to a bad configuration.

This can be entirely explained due to the low EXECUTION_TIMEOUT. In many cases, NAIVEAUTOML fails to evaluate good baseline models such as random forests, and instead only manages to evaluate models such as Gaussian Naive Bayes which have fast train and inference times but relatively poor performance. The experiment logs and framework authors confirmed our findings, and explained that their original work also used a larger EXECUTION_TIMEOUT. If we had set our EXECUTION_TIMEOUT to a fixed 10 minutes (instead of 3), we might have observed a similar performance to that reported by Mohr and Wever (2023).

The framework authors state that they plan to integrate their work on learning curve cross-validation (LCCV, Mohr and van Rijn (2023)), which would provide automatic scaling of their evaluation resources and make setting EXECUTION_TIMEOUT obsolete. We hope to evaluate this future version of NAIVEAUTOML and publish the results online.

References

Sebastian Pineda Arango, Hadi S. Jomaa, Martin Wistuba, and Josif Grabocka. HPO-B: A Large-Scale Reproducible Benchmark for Black-Box HPO based on OpenML. *arXiv:2106.06257 [cs]*, 2021.

A. Balaji and A. Allen. Benchmarking automatic machine learning frameworks. *CoRR*, abs/1808.06492, 2018. URL http://arxiv.org/abs/1808.06492.

James Bergstra, Rémi Bardenet, B Kégl, and Y Bengio. Implementations of algorithms for hyper-parameter optimization. In *NIPS Workshop on Bayesian optimization*, page 29, 2011.

James Bergstra, Daniel Yamins, and David Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International conference on machine learning*, pages 115–123. PMLR, 2013.

James Bergstra, Brent Komer, Chris Eliasmith, and David Warde-Farley. Preliminary Evaluation of Hyperopt Algorithms on HPOLib. In *ICML Workshop on Automatic Machine Learning*, 2014.

Bernd Bischl, Giuseppe Casalicchio, Matthias Feurer, Pieter Gijsbers, Frank Hutter, Michel Lang, Rafael Gomes Mantovani, Jan N van Rijn, and Joaquin Vanschoren. Openml benchmarking suites. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

Rich Caruana, Alexandru Niculescu-Mizil, Geoff Crew, and Alex Ksikes. Ensemble selection from libraries of models. In *Proceedings of the twenty-first international conference on Machine learning*, page 18, 2004.

Rich Caruana, Art Munson, and Alexandru Niculescu-Mizil. Getting the most out of ensemble selection. In *Sixth International Conference on Data Mining (ICDM'06)*, pages 828–833. IEEE, 2006.

Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. *CoRR*, abs/1603.02754, 2016. URL http://arxiv.org/abs/1603.02754.

Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

J. Demšar. Statistical Comparisons of Classifiers over Multiple Data Sets. *The Journal of Machine Learning Research*, 7:1–30, 2006.

Iddo Drori, Yamuna Krishnamurthy, Remi Rampin, Raoni Lourenço, Jorge One, Kyunghyun Cho, Claudio Silva, and Juliana Freire. Alphad3m: Machine learning pipeline synthesis. In *5th ICML Workshop on Automated Machine Learning (AutoML)*, 2018.

K. Eggensperger, M. Feurer, F. Hutter, J. Bergstra, J. Snoek, H. Hoos, and K. Leyton-Brown. Towards an Empirical Foundation for Assessing Bayesian Optimization of Hyperparameters. In *NIPS Workshop on Bayesian Optimization in Theory and Practice*, 2013.

Katharina Eggensperger, Philipp Müller, Neeratyoy Mallik, Matthias Feurer, René Sass, Aaron Klein, Noor Awad, Marius Lindauer, and Frank Hutter. Hpobench: A collection of reproducible multi-fidelity benchmark problems for hpo, 2021.

Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. Autogluon-tabular: Robust and accurate automl for structured data, 2020.

Kutluhan Erol, James A Hendler, and Dana S Nau. Umcp: A sound and complete procedure for hierarchical task-network planning. In *Aips*, volume 94, pages 249–254, 1994.

Hugo Jair Escalante, Manuel Montes, and Luis Enrique Sucar. Particle swarm model selection. *Journal of Machine Learning Research*, 10(2), 2009.

Manuel J.A. Eugster, Friedrich Leisch, and Carolin Strobl. (psycho-)analysis of benchmark experiments. *Comput. Stat. Data Anal.*, 71(C):986–1000, mar 2014. ISSN 0167-9473.

Mauricio Ferreira, Rafaela Ventorim, Eduardo Almeida, Sabrina Silveira, and Wendel Silveira. Protein abundance prediction through machine learning methods. *Journal of molecular biology*, 433(22):167267, 2021.

M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015a.

Matthias Feurer, Jost Springenberg, and Frank Hutter. Initializing bayesian hyperparameter optimization via meta-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, 2015b.

Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. Practical automated machine learning for the automl challenge 2018. In *International Workshop on Automatic Machine Learning at ICML*, pages 1189–1232, 2018.

Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. Auto-sklearn 2.0: Hands-free automl via meta-learning, 2020. URL `https://arxiv.org/abs/2007.04074`.

Sebastian Felix Fischer, Matthias Feurer, and Bernd Bischl. OpenML-CTR23–a curated tabular regression benchmarking suite. In *AutoML Conference 2023 (Workshop)*, 2023.

P. Gijsbers, E. LeDell, S. Poirier, J. Thomas, B. Bischl, and J. Vanschoren. An open source automl benchmark. *arXiv preprint arXiv:1907.00909 [cs.LG]*, 2019. URL `https://arxiv.org/abs/1907.00909`. Accepted at AutoML Workshop at ICML 2019.

Pieter Gijsbers and Joaquin Vanschoren. GAMA: A General Automated Machine Learning Assistant. In Yuxiao Dong, Georgiana Ifrim, Dunja Mladenić, Craig Saunders, and Sofie Van Hoecke, editors, *Machine Learning and Knowledge Discovery in Databases. Applied Data Science and Demo Track*, pages 560–564, Cham, 2021. Springer International Publishing. ISBN 978-3-030-67670-4.

Yolanda Gil, Ke-Thia Yao, Varun Ratnakar, Daniel Garijo, Greg Ver Steeg, Pedro Szekely, Rob Brekelmans, Mayank Kejriwal, Fanghao Luo, and I-Hui Huang. P4ML: A phased performance-based pipeline planner for automated machine learning. In *5th ICML Workshop on Automated Machine Learning (AutoML)*, 2018.

Isabelle Guyon, Lisheng Sun-Hosoya, Marc Boullé, Hugo Jair Escalante, Sergio Escalera, Zhengying Liu, Damir Jajetic, Bisakha Ray, Mehreen Saeed, Michèle Sebag, et al. Analysis of the automl challenge series. *Automated Machine Learning*, page 177, 2019.

H2O.ai. *H2O: Scalable Machine Learning Platform*, 2013. URL `https://github.com/h2oai/h2o-3`. First version of H2O was released in 2013.

M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009. ISSN 1931-0145. doi: 10.1145/1656274.1656278. URL `http://doi.acm.org/10.1145/1656274.1656278`.

Nikolaus Hansen, Anne Auger, Raymond Ros, Olaf Mersmann, Tea Tušar, and Dimo Brockhoff. Coco: A platform for comparing continuous optimizers in a black-box setting. *Optimization Methods and Software*, 36(1):114–144, 2021.

Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011.

Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors. *Automated Machine Learning - Methods, Systems, Challenges.* Springer, 2019.

Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In Arthur Gretton and Christian C. Robert, editors, *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, volume 51 of *Proceedings of Machine Learning Research*, pages 240–248, Cadiz, Spain, 09–11 May 2016. PMLR. URL `http://proceedings.mlr.press/v51/jamieson16.html`.

Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1946–1956, 2019.

Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, 30:3146–3154, 2017.

Aaron Klein, Zhenwen Dai, Frank Hutter, Neil Lawrence, and Javier Gonzalez. Meta-surrogate benchmarking for hyperparameter optimization. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL `https://proceedings.neurips.cc/paper/2019/file/0668e20b3c9e9185b04b3d2a9dc8fa2d-Paper.pdf`.

Helena Kotthaus, Ingo Korb, Michel Lang, Bernd Bischl, Jörg Rahnenführer, and Peter Marwedel. Runtime and memory consumption analyses for machine learning r programs. *Journal of Statistical Computation and Simulation*, 85(1):14–29, 2015. doi: 10.1080/00949655.2014.925192.

John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

Trang T Le, Weixuan Fu, and Jason H Moore. Scaling tree-based automated machine learning to biomedical big data with a dataset selector. *BioRxiv*, page 502484, 2018.

Erin LeDell and Sebastien Poirier. H2O AutoML: Scalable automatic machine learning. In *7th ICML workshop on automated machine learning*, 2020. URL `http://docs.h2o.ai/h2o/latest-stable/h2o-docs/automl.html`.

Robert I McKay, Nguyen Xuan Hoai, Peter Alexander Whigham, Yin Shan, and Michael O'neill. Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines*, 11:365–396, 2010.

F. Mohr, M. Wever, and E. Hüllermeier. Ml-plan: Automated machine learning via hierarchical planning. *Machine Learning*, 107(8):1495–1515, Sep 2018. ISSN 1573-0565. doi: 10.1007/s10994-018-5735-z. URL `https://doi.org/10.1007/s10994-018-5735-z`.

Felix Mohr and Jan N van Rijn. Fast and informative model selection using learning curve cross-validation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2023.

Felix Mohr and Marcel Wever. Naive automated machine learning. *Machine Learning*, 112(4):1131–1170, 2023.

Takeru Ohta and Hiroyuki Vincent Yamazaki. Kurobako. `https://github.com/optuna/kurobako`, 2022.

Randal S. Olson and Jason H. Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. In Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors, *Proceedings of the Workshop on Automatic Machine Learning*, volume 64 of *Proceedings of Machine Learning Research*, pages 66–74, New York, New York, USA, 24 Jun 2016. PMLR. URL `http://proceedings.mlr.press/v64/olson_tpot_2016.html`.

Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. *CoRR*, abs/1603.06212, 2016. URL `http://arxiv.org/abs/1603.06212`.

Randal S Olson, William La Cava, Patryk Orzechowski, Ryan J Urbanowicz, and Jason H Moore. Pmlb: a large benchmark suite for machine learning evaluation and comparison. *BioData mining*, 10(1):1–13, 2017.

Preston Parry. auto_ml. `https://github.com/ClimbsRocks/auto_ml`, 2018.

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

Aleksandra Płońska and Piotr Płoński. Mljar: State-of-the-art automated machine learning framework for tabular data. version 0.10.3, 2021. URL `https://github.com/mljar/mljar-supervised`.

Philipp Probst, Anne-Laure Boulesteix, and Bernd Bischl. Tunability: Importance of Hyperparameters of Machine Learning Algorithms. *Journal of Machine Learning Research*, 20(53):1–32, 2019. ISSN 1533-7928. URL `http://jmlr.org/papers/v20/18-444.html`.

Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. Catboost: Unbiased boosting with categorical features. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 6639–6649, Red Hook, NY, USA, 2018. Curran Associates Inc.

Herilalaina Rakotoarison, Marc Schoenauer, and Michèle Sebag. Automated machine learning with monte-carlo tree search. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, pages 3296–3303, 2019.

Matthias Reif, Faisal Shafait, and Andreas Dengel. Meta-learning for evolutionary parameter optimization of classifiers. *Machine learning*, 87(3):357–380, 2012.

Joseph D Romano, Trang T Le, Weixuan Fu, and Jason H Moore. Tpot-nn: augmenting tree-based automated machine learning with neural network estimators. *Genetic Programming and Evolvable Machines*, 22(2):207–227, 2021.

Robin Schmucker, Michele Donini, Muhammad Bilal Zafar, David Salinas, and Cédric Archambeau. Multi-objective asynchronous successive halving, 2021.

Kenan Šehić, Alexandre Gramfort, Joseph Salmon, and Luigi Nardi. Lassobench: A high-dimensional hyperparameter optimization benchmark suite for lasso. *arXiv preprint arXiv:2111.02790*, 2021.

Xingjian Shi, Jonas Mueller, Nick Erickson, Mu Li, and Alexander J Smola. Benchmarking multimodal automl for tabular data with text fields. *arXiv preprint arXiv:2111.02705*, 2021.

Julien Siems, Lucas Zimmer, Arber Zela, Jovita Lukasik, Margret Keuper, and Frank Hutter. Nas-bench-301 and the case for surrogate benchmarks for neural architecture search. *CoRR*, abs/2008.09777, 2020. URL `https://arxiv.org/abs/2008.09777`.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.

Andrew Sohn, Randal S Olson, and Jason H Moore. Toward the automated analysis of complex diseases in genome-wide association studies using genetic programming. In *Proceedings of the genetic and evolutionary computation conference*, pages 489–496, 2017.

Carolin Strobl, Florian Wickelmaier, and Achim Zeileis. Accounting for individual differences in bradley-terry models by means of recursive partitioning. *Journal of Educational and Behavioral Statistics*, 36(2): 135–153, 2011. doi: 10.3102/1076998609359791. URL `https://doi.org/10.3102/1076998609359791`.

J. Thomas, S. Coors, and B. Bischl. Automatic gradient boosting. In *International Workshop on Automatic Machine Learning at ICML*, 2018.

C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Auto-WEKA: Combined selection and hyper-parameter optimization of classification algorithms. In *Proc. of KDD-2013*, pages 847–855, 2013.

Anh Truong, Austin Walters, Jeremy Goodsitt, Keegan E. Hines, C. Bayan Bruss, and Reza Farivar. Towards Automated Machine Learning: Evaluation and Comparison of AutoML Approaches and Tools. In *31st IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2019, Portland, OR, USA, November 4-6, 2019*, pages 1471–1479. IEEE, 2019. doi: 10.1109/ICTAI.2019.00209.

Ryan Turner. Uber. bayesopt benchmark. `https://github.com/uber/bayesmark`, 2022.

Anton Vakhrushev, Alexander Ryzhkov, Maxim Savchenko, Dmitry Simakov, Rinchin Damdinov, and Alexander Tuzhilin. Lightautoml: Automl solution for a large financial services ecosystem. *arXiv preprint arXiv:2109.01528*, 2021.

Koen Van der Blom, Alex Serban, Holger Hoos, and Joost Visser. Automl adoption in ml software. In *8th ICML Workshop on Automated Machine Learning (AutoML)*, 2021.

Tony Van Gestel, Johan AK Suykens, Bart Baesens, Stijn Viaene, Jan Vanthienen, Guido Dedene, Bart De Moor, and Joos Vandewalle. Benchmarking least squares support vector machine classifiers. *Machine learning*, 54(1):5–32, 2004.

Jan N Van Rijn and Frank Hutter. Hyperparameter importance across datasets. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2367–2376, 2018.

J. Vanschoren, J. van Rijn, B. Bischl, and L. Torgo. OpenML: Networked science in machine learning. *SIGKDD Explor. Newsl.*, 15(2):49–60, 2014.

Chi Wang, Qingyun Wu, Markus Weimer, and Erkang Zhu. Flaml: A fast and lightweight automl library. In *MLSys*, 2021.

Hilde JP Weerts, Andreas C Mueller, and Joaquin Vanschoren. Importance of tuning hyperparameters of machine learning algorithms. *arXiv preprint arXiv:2007.07588*, 2020.

M. Wever, A. Tornede, F. Mohr, and E. Hullermeier. AutoML for Multi-Label Classification: Overview and Empirical Evaluation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–19, 2021. doi: 10.1109/TPAMI.2021.3051276.

Qingyun Wu, Chi Wang, and Silu Huang. Frugal optimization for cost-related hyperparameters. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 10347–10354, 2021.

Zhenqin Wu, Bharath Ramsundar, Evan N Feinberg, Joseph Gomes, Caleb Geniesse, Aneesh S Pappu, Karl Leswing, and Vijay Pande. Moleculenet: a benchmark for molecular machine learning. *Chemical science*, 9(2):513–530, 2018.

C. Yang, Y. Akimoto, D.W. Kim, and M. Udell. OBOE: collaborative filtering for automl initialization. *CoRR*, abs/1808.03233, 2018. URL `http://arxiv.org/abs/1808.03233`.

Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. NAS-bench-101: Towards reproducible neural architecture search. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 7105–7114, Long Beach, California, USA, 09–15 Jun 2019. PMLR. URL `http://proceedings.mlr.press/v97/ying19a.html`.

Achim Zeileis and Kurt Hornik. Generalized m-fluctuation tests for parameter instability. *Statistica Neerlandica*, 61(4):488–508, 2007. doi: https://doi.org/10.1111/j.1467-9574.2007.00371.x. URL `https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-9574.2007.00371.x`.

Arber Zela, Julien Siems, and Frank Hutter. Nas-bench-1shot1: Benchmarking and dissecting one-shot neural architecture search. *CoRR*, abs/2001.10422, 2020. URL `https://arxiv.org/abs/2001.10422`.

Lucas Zimmer, Marius Lindauer, and Frank Hutter. Auto-pytorch tabular: Multi-fidelity metalearning for efficient and robust autodl. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 3079 – 3090, 2021. also available under https://arxiv.org/abs/2006.13799.

Marc-André Zöller and Marco F. Huber. Benchmark and survey of automated machine learning frameworks. *J. Artif. Intell. Res.*, 70:409–472, 2021. doi: 10.1613/jair.1.11854.