7th ICML Workshop on Automated Machine Learning (2020)

On Evaluation of AutoML Systems

Mitar Milutinovic mitar@cs.berkeley.edu
University of California, Berkeley, USA

Brandon Schoenfeld bjschoenfeld@gmail.com
Brigham Young University, USA

Diego Martinez-Garcia dmartinezg@protonmail.com
Texas A&M University, USA

Saswati Ray sray@cs.cmu.edu
Carnegie Mellon University, USA

Sujen Shah sujen.shah@jpl.nasa.gov
NASA Jet Propulsion Laboratory, California Institute of Technology, USA

David Yan david.yan@berkeley.edu
University of California, Berkeley, USA

Abstract

Maintaining continued progress in AutoML research requires that we enable and conduct thorough evaluations of AutoML systems. We outline several fundamental AutoML evaluation practices and present a unified, open-source machine learning framework, upon which AutoML systems can be built, to enable such evaluations. This framework formalizes ML programs as end-to-end pipelines with all the necessary components to support ML algorithms from a wide range of diverse libraries and reference tooling for reproducible pipeline execution. We demonstrate the usability of this framework by presenting brief evaluation results of eight AutoML systems that use it. Additionally, we have designed and implemented a service, a metalearning database, that stores information about executed ML programs generated by systems using this framework. Our approach enables the research community to test hypotheses about the internals of AutoML systems, e.g., how pipeline search algorithms used by different AutoML systems compare.

1. Introduction

Progress in machine learning (ML) is being published faster than humans can consume it, driving the community to rely more on automatic machine learning (AutoML) tools. This motivates us to analyze the AutoML approaches and systems themselves, not just the ML algorithms. The complexity of such systems, the diversity of modern problem types, and the variety of ML algorithms all complicate the comparison of AutoML systems. For example, how can we compare an AutoML system that searches over neural networks with another that searches over graphical models? These two AutoML systems may differ in the resources required to run, the speed at which they can build an ML model, the type of data they use, and the search algorithm used to search the space of models.

Currently, there exist many AutoML systems in both academia and industry (Section 2), but to advance the state of the art in AutoML, we need increasingly sophisticated evaluation methods. We first identify several important practices to consider when comparing different

AutoML systems (Section 3). We then present an ML framework (Section 4), implemented in Python, upon which AutoML systems can be built, that facilitates evaluations grounded in these practices. Our framework formalizes ML programs precisely as end-to-end *pipelines* with reference pipeline execution tooling and supports ML algorithms from a wide range of diverse libraries, potentially raw data, diverse data and problem types, a high degree of reproducibility and execution logging, and the creation of a cross-system metalearning database. We provide a public metalearning database populated with executed pipeline information, including the evaluation results presented here. Finally, we discuss (Section 5) the contributions of this work in the context of related work. The use of this framework in AutoML systems enables the community to test scientific hypotheses about system designs and promotes continuous progress in AutoML research. In Appendix A we present brief evaluation results of eight AutoML systems that use this framework.

2. Related Work

The AutoML community has produced many academic and non-academic systems: AlphaD3M (Drori et al., 2018), Auto-sklearn (Feurer et al., 2019), Google Cloud AutoML (Google), H20 (H2O.ai, 2017), Hyperopt (Komer et al., 2018), Auto-WEKA (Kotthoff et al., 2018), SmartML (Maher and Sakr, 2019), Auto-Net (Mendoza et al., 2018), auto_ml (Parry, 2018), ML-Plan (Mohr et al., 2018), TPOT (Olson and Moore, 2018), Mosaic (Rakotoarison et al., 2019), Auto-Meka (de Sá et al., 2018), RECIPE (de Sá et al., 2017), TransmogrifAI (Salesforce), Alpine Meadow (Shang et al., 2019), The Automatic Statistician (Steinrucken et al., 2018), ATM (Drevo et al., 2017), Rafiki (Wang et al., 2018), Adaptive TPOT (Evans et al., 2020), and Oboe (Yang et al., 2019). Some systems focus on neural networks only: MetaQNN (Baker et al., 2016), DEvol (Davison et al., 2017), Auto-Keras (Jin et al., 2018), Neural Network Intelligence (Microsoft, 2018), ENAS (Pham et al., 2018), and Neural Architecture Search (Zoph and Le, 2016). We observe (Elshawi et al., 2019) that their approaches, even the programming languages, vary significantly, complicating system comparison.

Existing evaluations, including the AutoML Benchmark (Gijsbers et al., 2019), the AutoML Challenge Series (Guyon et al., 2018), and another benchmarking study (Balaji and Allen, 2018), attempt to help practitioners identify strengths and weaknesses of systems by taking the practical approach of evaluating systems “as-is”, comparing only the predictions of each system’s chosen pipeline. These comparisons, as well as another benchmarking suite (Bischl et al., 2017), focus on tabular data, consisting primarily of classification and regression tasks with numeric and categorical data and some missing values, but the AutoML Challenge Series also includes various other problem types that are preprocessed to fit the tabular paradigm. Notably, the AutoML Benchmark evaluates systems given multiple time budgets, helping demonstrate how AutoML systems evolve with increasingly available resources. Another study (Zöller and Huber, 2019) used 137 OpenML datasets to evaluate multiple AutoML systems (auto-sklearn, TPOT, hyperopt-sklearn, RoBO (Klein et al., 2017), BTB (Gustafson, 2018)). TUPAQ (Sparks et al., 2015) emphasizes large-scale distributed and scalable machine learning, focusing their evaluation on scalability and convergence rates.

3. Towards Better Evaluation of AutoML Systems

AutoML systems should be evaluated with at least as much care as ML algorithms. We identify critical AutoML evaluation best practices, including: 1) the separation of datasets used for system development from those used for evaluation, 2) using the same set of ML building blocks systems use, and 3) standardizing a common ML program description language.

When ML algorithms themselves are evaluated, a common practice is to report its score on the test data, which is disjoint from the training data used to build the algorithm's internal state. Similarly, AutoML system evaluators should take care to split data, which for AutoML systems is a dataset of datasets. Any dataset used for system evaluation should not have already been used for system development and fine-tuning. This practice prevents overfitting and evaluators can demonstrate that systems are capable of automatically generating functional ML programs on novel datasets, not just on the datasets used during development. Ideally, evaluations would use k-fold cross-validation over dataset of datasets, but this is not always possible. It can be computationally too expensive to build AutoML systems multiple times. Moreover, not all AutoML systems can be automatically build from training datasets. As an alternative, evaluators can use blind datasets not known to authors of AutoML systems in advance.

Configuring systems to use the same collection of ML building blocks enables researchers to test scientific hypotheses about system designs, for example, comparing strategies for exploring the ML program search space. When two systems search for ML programs using different collections of ML building blocks, differences in system scores cannot be attributed to differences in search strategies alone. One system may have exclusive access to an algorithm that performs better on the evaluation datasets. If the other system were to have access to that same algorithm, it could potentially create that same (better) ML program, perhaps even sooner.

ML programs produced by systems should be described in a standardized language. This ensures that systems generate ML programs that contain only the ML building blocks permitted by the evaluators, without extra "glue code" that could leak into final ML programs. A common language also structures ML program execution semantics so the ML algorithms are used and composed in expected ways, e.g. one pass over the data, mini-batching for neural networks, etc. The execution semantic can be extensible to support various execution semantics, but fixed for any particular evaluation so that differences in ML program scores cannot be caused by differences in ML program interpretations. Use of a standard language promotes reproducibility and allows evaluations where AutoML systems produce only ML programs, not predictions, so that those programs can be trained and scored outside of systems, assuring equal access to ML building blocks and other resources.

4. Framework for Auto-generated ML Programs

To enable evaluation of AutoML systems as described in Section 3 and to enable a shared metalearning database between those AutoML systems, we designed and implemented the framework for auto-generated ML programs expressed as pipelines. We present the framework in more detail in (Milutinovic, 2019) and provide in this section a high-level overview.

4.1 Pipeline Language

We defined a pipeline as a Directed Acyclic Graph (DAG) of end-to-end steps to execute, where every step defines its inputs and outputs. There are two main step types: a primitive step and a sub-pipeline step. The former represents the execution of a basic building block, a *primitive*; the latter represents the execution of another pipeline. Pipelines also contain metadata. See the example pipeline in Appendix D.

4.2 Primitives

A primitive is an implementation of a function that could have parameters (like a classification algorithm) or hard-coded logic. In our framework, primitives can be written in any programming language, but they must expose Python interfaces that need to be extended from a set of base classes and additional mix-ins. In addition, a primitive defines metadata describing itself, defines its parameters (state), which are usually learned from sample input and output data, defines hyper-parameters, which are general configuration parameters that do not change during the lifetime of a primitive, and defines types of inputs and outputs.

There are two main types of hyper-parameters a primitive can use to define its hyper-parameters configuration. *Tuning hyper-parameters* potentially influence the predictive performance of the primitive, e.g., learning rate, depth of trees in a random forest, an architecture of the neural network. *Control hyper-parameters* control the behavior (logic) of primitives, e..g, whether or not a primitive is allowed to discard the unused columns.

4.3 Data and Metadata

The framework prescribes data types that can be passed between steps in a pipeline. Currently, a narrow set is allowed (Numpy ndarrays, Pandas DataFrames, Python lists, and Datasets). The Dataset data type serves as a starting point for a pipeline and can represent a wide range of input data, including raw data. Data types are extended to support the storage of additional metadata, for which the framework provides a standardized schema for many use cases. For example, *semantic types* are standardized descriptions of the meaning of the data, not just its representation in memory.

The dataset data type is a unified representation of the inputs that allow us to describe the relationships among multiple components and hints about how the data should be read. Some examples where the dataset representation is essential is when the dataset contains media (image, audio, video) distributed into multiples resources or datasets that are spread among multiple tables such as graph or relational data.

4.4 Standard Execution Semantics

We have defined a standard execution semantic named *fit-produce*. It executes the pipeline in a data-flow manner in two phases: fit and produce. The run of each phase of a pipeline execution proceeds in order of its steps, as in Appendix C. The framework provides a reference runtime to execute pipelines and log their execution into *pipeline run* documents.

4.5 Metalearning Database

By design, our framework allows for metalearning across AutoML systems because all pipelines share the same set of primitives and use the same language to describe the ML programs. All pipeline run documents can be contributed to a public metalearning database.

5. Discussion

Existing work (Section 2) has attempted to address some of the best practices identified in Section 3. The AutoML Challenge Series (ChaLearn) (Guyon et al., 2018) conducted evaluations by running system-submitted code on blind datasets. The AutoML Benchmark (Gijsbers et al., 2019) attempts to identify which evaluation datasets were also used for system building, for example, which datasets Auto-sklearn used to build its internal meta-model. The datasets used in ChaLearn (6 rounds with 5 datasets each) represented a variety of tasks, data distributions, and metrics, but were preprocessed into a tabular format, potentially causing generated ML programs to be atypical. A third benchmarking study (Balaji and Allen, 2018) comparing four popular systems noted that "[m]any open datasets require extensive preprocessing before use" and limited their study to clean OpenML (Vanschoren et al., 2013) datasets. All of these benchmark evaluations compared systems that each used their own set of ML building blocks instead of a shared set, for practical reasons. However, conclusions about pipeline search and optimization strategies (e.g., is Bayesian optimization better than genetic optimization?) are limited because differences in system performance is confounded by the differences in algorithm availability to each system.

Using the pipelines generated in the above studies for automatic warm-starting or metalearning between systems is not feasible because each system uses its own pipeline representation. There are some popular pipeline languages which might be candidates for such a purpose. scikit-learn's (Pedregosa et al., 2011) pipeline allows combining multiple scikit-learn transforms and estimators, supporting tabular and structured data, but not raw input files. Common Workflow Language (Amstutz et al., 2016) is a standard for describing data-analysis workflows with a focus on reproducibility. However, it also focuses on combining command line programs into workflows, a pattern not generally followed by AutoML-made ML programs. Kubeflow (kub) simultaneously provides a pipeline language and simple Kubernetes deployments. It supports the combining of components that use different libraries, but every component is a Docker image, thus requiring inputs and outputs to be serialized instead of directly passing memory objects between components.

Our proposed framework (Section 4) was designed to support evaluations based on the best practices outlined in Section 3. As an example, a third-party evaluator evaluated eight AutoML systems build upon the framework. Blind datasets were used to prevent all systems from overfitting to known datasets. Evaluation results (Appendix A) thus show how well systems generalize to new tasks in terms of both pipeline performance and the number of task types supported. All systems evaluated used the exact same set of ML building blocks, including same versions. Therefore, differences in scores can be attributed to system designs. Researchers can isolate design differences and improve system strategies, thus advancing AutoML research. Note that these advances are limited to the types of ML building blocks given to systems. E.g., the pipeline search strategies that produce the best pipelines with Scikit-learn (Pedregosa et al., 2011) algorithms might be different from those strategies that produce the best neural network pipelines. A similar caveat can be made for the types of tasks on which systems are evaluated. Still, evaluation of systems build using our framework allows us to explore these types of questions and test related hypotheses.

An advantage of using a standardized ML framework across ML systems is that there are already various tools available. The proposed tool by (Ono et al., 2020) visualizes and compares pipelines generated by different AutoML systems, which provides insight into the behavior of various components. Marvin (Mattmann et al., 2018) is an index of all current and historic primitives and provides a web interface to search for primitives by their metadata. TwoRavens wrapper (D'Orazio et al., 2019) wraps AutoML systems to expose the same control interface to make execution of AutoML systems uniform.

Future work could explore aspects of ML programs beyond predictions and scores, such as complexity, interpretability, generalizability, resource and data requirements, etc. Additional system strengths could be identified by varying time and resource budgets as well as evaluation metrics. The next steps could be to collect more pipelines and execution results into the metalearning database, analyze them, and iterate on AutoML system designs. There is potential even to automate AutoML design itself via metalearning over the metalearning database.

6. Conclusion

In this work we identified important best practices for evaluation of AutoML systems. The ML framework we presented allows systems that are built on it to be evaluated according to these practices. The use of this framework by eight systems and their evaluation results presented help demonstrate the framework's viability. We observed that the framework can describe a diverse set of ML programs solving many ML task types. More important than the actual results obtained from any one particular evaluation is the ability to identify strengths and weaknesses specific to AutoML system designs and make improvements.

Documentation and Source Code

Documentation for our ML framework is available at `https://docs.datadrivendiscovery.org/`, including detailed description of all components. Source code is open source and is available at `https://gitlab.com/datadrivendiscovery`, in particular at `https://gitlab.com/datadrivendiscovery/d3m`.

Acknowledgments

The authors gratefully acknowledge the Data Driven Discovery of Models (D3M) program of the Defense Advanced Research Projects Agency (DARPA) administered through AFRL contracts FA8750-17-2-0082, FA8750-17-2-0091, FA8750-17-2-0116, FA8750-17-2-0130. We thank especially Brian Sandberg for his leadership in D3M and Shelly Stanley for her tireless efforts in executing our third-party evaluations. We thank the anonymous reviewers whose feedback helped us improve this paper significantly. While this paper lists a few authors, a much broader group of people contributed to this work. We would like to thank them all for their participation, feedback, and willingness to endure.

References

Kubeflow. URL `https://www.kubeflow.org/`.

P. Amstutz, M. R. Crusoe, N. Tijanić, B. Chapman, J. Chilton, M. Heuer, A. Kartashov, J. Kern, D. Leehr, H. Ménager, M. Nedeljkovich, M. Scales, S. Soiland-Reyes, and L. Stojanovic. Common workflow language, v1.0, 2016. URL `https://doi.org/10.6084/m9.figshare.3115156.v2`.

B. Baker, O. Gupta, N. Naik, and R. Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016. URL `https://arxiv.org/abs/1611.02167`.

A. Balaji and A. Allen. Benchmarking automatic machine learning frameworks. *arXiv preprint arXiv:1808.06492*, 2018.

B. Bischl, G. Casalicchio, M. Feurer, F. Hutter, M. Lang, R. G. Mantovani, J. N. van Rijn, and J. Vanschoren. Openml benchmarking suites and the openml100. *arXiv preprint arXiv:1708.03731*, 2017.

J. Davison et al. DEvol: Deep neural network evolution, 2017. URL `https://github.com/joeddav/devol`.

A. G. C. de Sá, W. J. G. S. Pinto, L. O. V. B. Oliveira, and G. L. Pappa. RECIPE: A grammar-based framework for automatically evolving classification pipelines. In J. McDermott, M. Castelli, L. Sekanina, E. Haasdijk, and P. García-Sánchez, editors, *Genetic Programming*, pages 246–261, Cham, 2017. Springer International Publishing. ISBN 978-3-319-55696-3.

A. G. C. de Sá, A. A. Freitas, and G. L. Pappa. Automated selection and configuration of multi-label classification algorithms with grammar-based genetic programming. In A. Auger, C. M. Fonseca, N. Lourenço, P. Machado, L. Paquete, and D. Whitley, editors, *Parallel Problem Solving from Nature – PPSN XV*, pages 308–320, Cham, 2018. Springer International Publishing. ISBN 978-3-319-99259-4.

T. S. W. Drevo, B. Cyphers, A. Cuesta-Infante, A. Ross, and K. Veeramachaneni. ATM: A distributed, collaborative, scalable system for automated machine learning. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 151–162, 12 2017. doi: 10.1109/BigData.2017.8257923. URL `https://github.com/HDI-Project/ATM`.

I. Drori, Y. Krishnamurthy, R. Rampin, R. Lourenço, J. P. One, K. Cho, C. Silva, and J. Freire. AlphaD3M: Machine learning pipeline synthesis. In *AutoML Workshop at ICML*, 2018.

V. D'Orazio, J. Honaker, R. Prasady, and M. Shoemate. Modeling and forecasting armed conflict: AutoML with human-guided machine learning. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 4714–4723, 2019. URL `http://hona.kr/papers/files/AutoMLConflict.pdf`.

R. Elshawi, M. Maher, and S. Sakr. Automated machine learning: State-of-the-art and open challenges. *arXiv preprint arXiv:1906.02287*, 2019. URL `https://arxiv.org/abs/1906.02287`.

B. P. Evans, B. Xue, and M. Zhang. An adaptive and near parameter-free evolutionary computation approach towards true automation in automl. *arXiv preprint arXiv:2001.10178*, 2020.

M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, and F. Hutter. Auto-sklearn: Efficient and robust automated machine learning. In *Automated Machine Learning*, pages 113–134. Springer, 2019.

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

Google. Cloud AutoML. URL `https://cloud.google.com/automl/`.

L. Gustafson. *Bayesian tuning and bandits: an extensible, open source library for AutoML*. PhD thesis, Massachusetts Institute of Technology, 2018.

I. Guyon, L. Sun-Hosoya, M. Boullé, H. J. Escalante, S. Escalera, Z. Liu, D. Jajetic, B. Ray, M. Saeed, M. Sebag, A. Statnikov, W.-W. Tu, and E. Viegas. Analysis of the automl challenge series 2015-2018. In F. Hutter, L. Kotthoff, and J. Vanschoren, editors, *Automatic Machine Learning: Methods, Systems, Challenges*, pages 191–236. Springer, 2018. In press, available at `http://automl.org/book`.

H2O.ai. *H2O AutoML*, June 2017. URL `http://docs.h2o.ai/h2o/latest-stable/h2o-docs/automl.html`. H2O version 3.30.0.1.

H. Jin, Q. Song, and X. Hu. Auto-Keras: Efficient neural architecture search with network morphism, 2018.

A. Klein, S. Falkner, N. Mansur, and F. Hutter. Robo: A flexible and robust bayesian optimization framework in python. In *NIPS 2017 Bayesian Optimization Workshop*, 2017.

B. Komer, J. Bergstra, and C. Eliasmith. Hyperopt-Sklearn. In F. Hutter, L. Kotthoff, and J. Vanschoren, editors, *Automatic Machine Learning: Methods, Systems, Challenges*, pages 105–121. Springer, 2018. URL `http://automl.org/book`.

L. Kotthoff, C. Thornton, H. H. Hoos, F. Hutter, and K. Leyton-Brown. Auto-WEKA: Automatic model selection and hyperparameter optimization in weka. In F. Hutter, L. Kotthoff, and J. Vanschoren, editors, *Automatic Machine Learning: Methods, Systems, Challenges*, pages 89–103. Springer, 2018. URL `http://automl.org/book`.

M. Maher and S. Sakr. SmartML: A meta learning-based framework for automated selection and hyperparameter tuning for machine learning algorithms. In *EDBT: 22nd International Conference on Extending Database Technology*, 2019.

C. A. Mattmann, S. Shah, and B. Wilson. Marvin: An open machine learning corpus and environment for automated machine learning primitive annotation and execution, 2018.

H. Mendoza, A. Klein, M. Feurer, J. T. Springenberg, M. Urban, M. Burkart, M. Dippel, M. Lindauer, and F. Hutter. Towards automatically-tuned deep neural networks. In F. Hutter, L. Kotthoff, and J. Vanschoren, editors, *Automatic Machine Learning: Methods, Systems, Challenges*, pages 145–161. Springer, 2018. URL `http://automl.org/book`.

Microsoft. Neural network intelligence, 2018. URL `https://github.com/Microsoft/nni`.

M. Milutinovic. *Towards Automatic Machine Learning Pipeline Design*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2019. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-123.html`.

F. Mohr, M. Wever, and E. Hüllermeier. ML-Plan: Automated machine learning via hierarchical planning. *Machine Learning*, 107(8):1495–1515, 9 2018. ISSN 1573-0565. doi: 10.1007/s10994-018-5735-z. URL `https://doi.org/10.1007/s10994-018-5735-z`.

R. S. Olson and J. H. Moore. TPOT: A tree-based pipeline optimization tool for automating machine learning. In F. Hutter, L. Kotthoff, and J. Vanschoren, editors, *Automatic Machine Learning: Methods, Systems, Challenges*, pages 163–173. Springer, 2018. URL `http://automl.org/book`.

J. P. Ono, S. Castelo, R. Lopez, E. Bertini, J. Freire, and C. Silva. Pipelineprofiler: A visual analytics tool for the exploration of automl pipelines. *arXiv preprint arXiv:2005.00160*, 2020.

P. Parry. auto_ml: Automated machine learning for production and analytics, 2018. URL `https://github.com/ClimbsRocks/auto_ml`.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, A. Müller, J. Nothman, G. Louppe, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018. URL `https://arxiv.org/abs/1802.03268`.

H. Rakotoarison, M. Schoenauer, and M. Sebag. Automated machine learning with monte-carlo tree search. *arXiv preprint arXiv:1906.00170*, 2019. URL `https://arxiv.org/abs/1906.00170`.

Salesforce. TransmogrifAI. URL `https://transmogrif.ai/`.

Z. Shang, E. Zgraggen, B. Buratti, F. Kossmann, P. Eichmann, Y. Chung, C. Binnig, E. Upfal, and T. Kraska. Democratizing data science through interactive curation of ml pipelines. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1171–1188. ACM, 2019.

E. R. Sparks, A. Talwalkar, D. Haas, M. J. Franklin, M. I. Jordan, and T. Kraska. Automating model search for large scale machine learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 368–380, 2015.

C. Steinrucken, E. Smith, D. Janz, J. Lloyd, and Z. Ghahramani. The automatic statistician. In F. Hutter, L. Kotthoff, and J. Vanschoren, editors, *Automatic Machine Learning: Methods, Systems, Challenges*, chapter 9, pages 175–188. Springer, 2018. In press, available at `http://automl.org/book`.

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

W. Wang, J. Gao, M. Zhang, S. Wang, G. Chen, T. K. Ng, B. C. Ooi, J. Shao, and M. Reyad. Rafiki: machine learning as an analytics service system. *Proceedings of the VLDB Endowment*, 12(2):128–140, 2018.

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

M.-A. Zöller and M. F. Huber. Survey on automated machine learning. *arXiv preprint arXiv:1904.12054*, 9, 2019.

B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016. URL `https://arxiv.org/abs/1611.01578`.

Appendix A. Evaluation Results

AutoML System	Known Datasets	Blind Datasets
CMU	93%	95%
ISI	92%	87%
MIT	92%	77%
NYU	91%	76%
SRI	76%	77%
TAMU	93%	74%
UCB	92%	87%
Uncharted	73%	55%

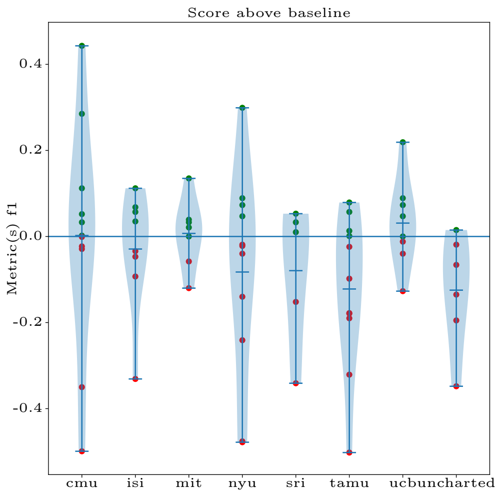
Table 1: Percentage of the 106 known datasets and 62 blind datasets for which each evaluated AutoML system successfully created a functional pipeline.

Our framework has been used by eight AutoML systems. An impartial 3rd party organization (Data Machines) evaluated them using a suite of 106 known datasets and 62 blind datasets, representing 15 different task types. Another 3rd party organization (MIT Lincoln Laboratory) prepared these datasets (Appendix B) and corresponding, expert-made baseline solutions. Systems had identical computing resources allocated.

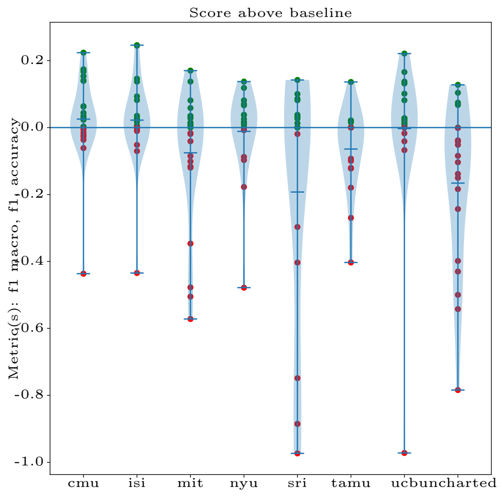
Figure 1 shows the results on tabular datasets and on some non-tabular datasets. Each dot on sub-figures of Figure 1 show how well system's best solution performed in comparison with the expert-made baseline solution. Because we are aggregating multiple datasets together, we show only relative distance from the baseline of each of those datasets. Higher is always better, for all metrics. Note as well that dots are shown only for datasets a system succeeded in producing pipelines for.

We can observe the worse overall performance of AutoML systems on blind datasets. Not easily visible in Figure 1 AutoML systems have produced much less successful pipelines on blind datasets as well. Success rates are shown in Table 1. Non-tabular datasets, while supported by our framework, presented a greater challenge to evaluated systems, leading to sparse results with only few AutoML systems succeeding on a particular dataset.

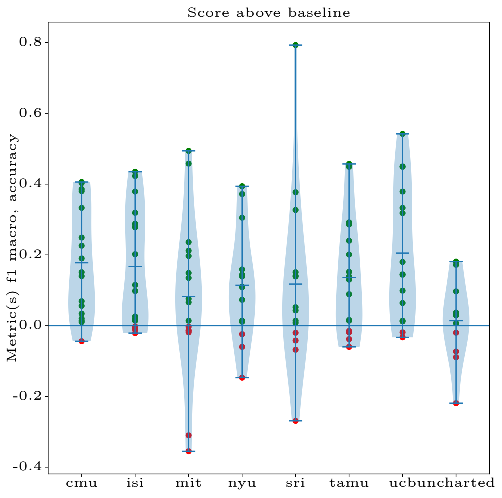
All pipelines and pipeline run documents made during evaluation by all AutoML systems have been stored into the metalearning database, as standardized documents for 168 datasets, 519 primitives, 160,999 pipelines, and 65,866 pipeline runs.



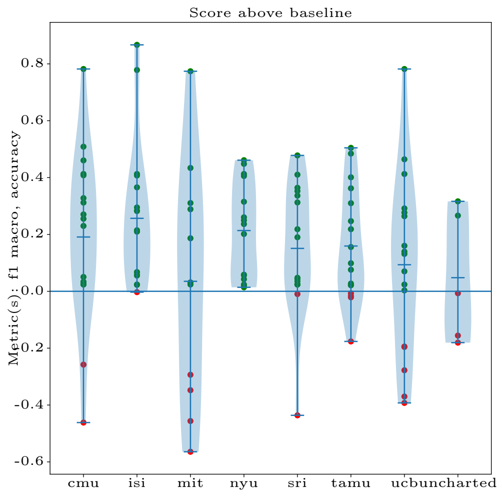
(a) Known tabular binary classification



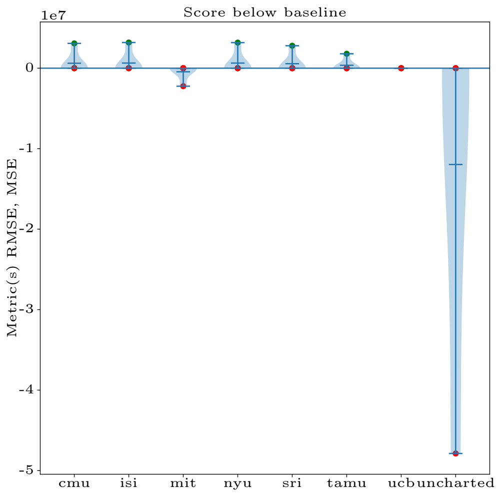
(b) Blind tabular binary classification



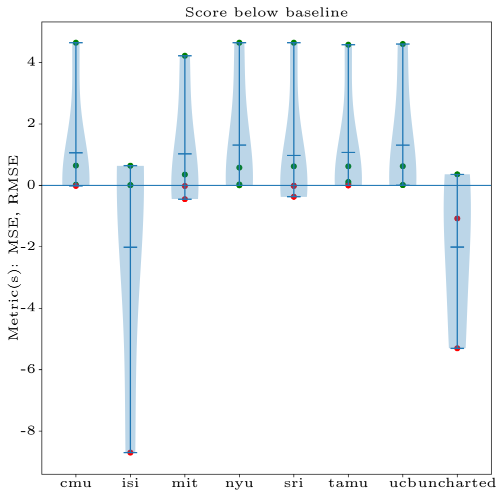
(c) Known tabular multi-class classification



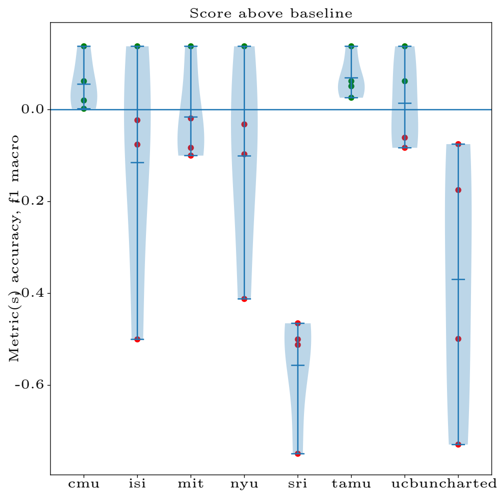
(d) Blind tabular multi-class classification



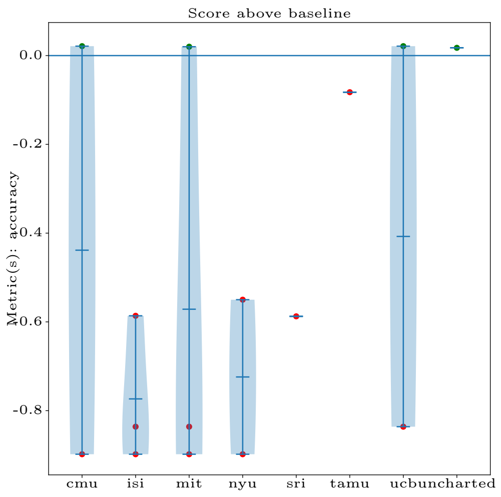
(e) Known tabular regression



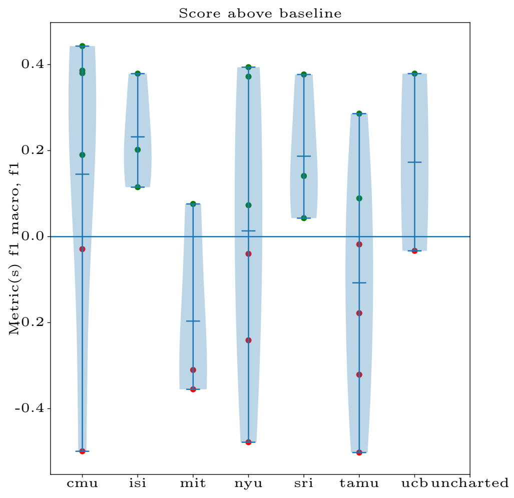
(f) Blind tabular regression



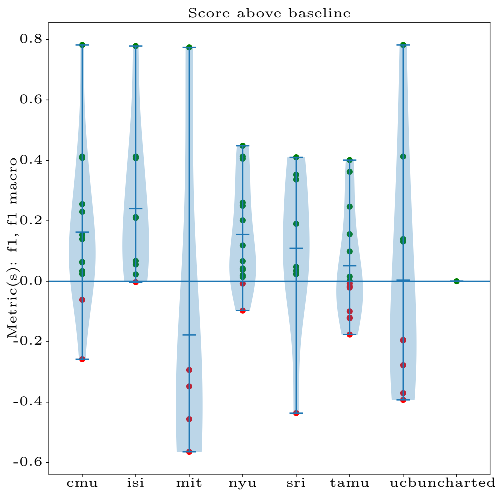
(g) Known graph vertex classification



(h) Blind graph vertex classification



(i) Known tabular semi-supervised



(j) Blind tabular semi-supervised

Figure 1: Evaluation results on known and blind datasets, aggregated for select task types. Note that some systems failed to produce any pipelines for some datasets, so individual plots may contain fewer data points. Higher is always better.

Dataset	Task type	Metric
CIFAR	Image classification4	Accuracy
Geolife	Classification	Accuracy
Mice protein	Classification	F1
Wikiqa	Text classification	F1
ElectricDevices	Time-series classification	F1
Arrowhead	Time-series classification	F1
Stock data	Time-series forecasting	MAE
Monthly sunspots	Time-series forecasting	MAE
TERRA canopy height	Time-series forecasting	MAE
Facebook	Graph matching	Accuracy
Retail sales	Regression	RMSE
Hand geometry	Image regression	MSE
Amazon	Community detection	NMI
Jester	Collaborative filtering	MAE
Net nomination	Vertex classification	Accuracy

Table 2: A subset of known datasets, their task type, and metric used. Achieved scores are not shown because scores on known datasets can be overfitted.

Appendix B. Datasets Used for Evaluation

There were 106 known datasets and 62 blind datasets used for evaluation, which are a subset of those available to the evaluated AuotML systems. Overall, those datasets span a wide range of task types, including (i) classification, (ii) regression, (iii) semi-supervised classification, (iv) time-series forecasting, (v) graph matching, (vi) link prediction, (vii) collaborative filtering, (viii) community detection, (ix) object detection and (x) vertex classification. List of raw data types present include (i) tabular data, (ii) text, (iii) time-series, (iv) graphs, (v) images, (vi) audio, (vii) video. Table 2 shows a subset of known datasets.

Appendix C. Reference Runtime Semantics

The reference runtime provided by the ML framework uses a fit-produce execution semantic as shown in Figure 2.

```
Function FitPipeline(pipeline, inputs)
    env ← inputs
    state ← ∅
    for step ∈ pipeline do
        if step ∈ PrimitivesStep then
            state ← state + FitPipeline(step, env)
            env ← env + ProducePipeline(state, step, env)
        else
            state ← state + FitPipeline(step, env)
            env ← env + ProducePipeline(state, step, env)
        end
    end
    return env, state
end

Function ProducePipeline(pipeline, state, inputs)
    env ← inputs
    for step ∈ pipeline do
        if step ∈ PrimitivesStep then
            env ← env + ProducePipeline(state, step, env)
        else
            env ← env + ProducePipeline(state, step, env)
        end
    end
    return env
end
```

Figure 2: Pseudo-code describing execution semantics of the reference runtime.

Appendix D. Example Pipeline

A visual representation of an example pipeline is presented in Figure 3. The example pipeline is described in more detail in (Milutinovic, 2019).

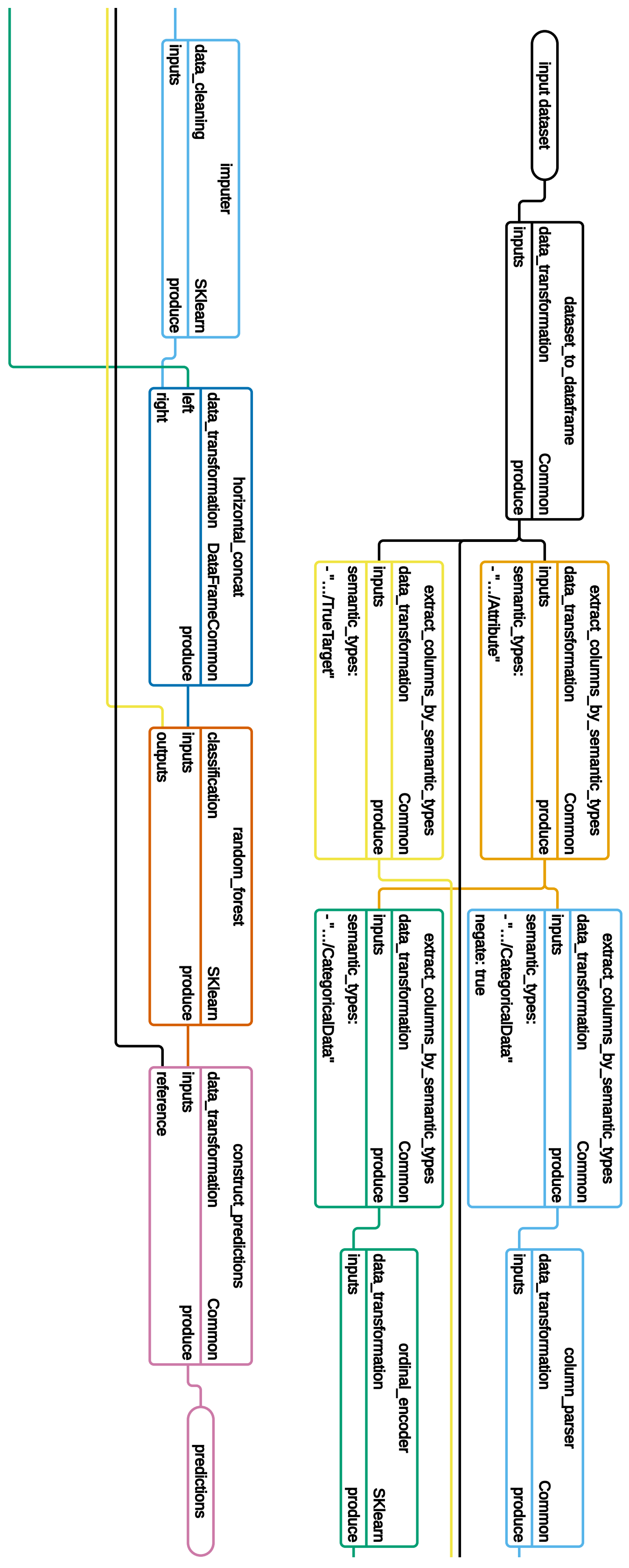


Figure 3: Visual representation of an example pipeline.