7th ICML Workshop on Automated Machine Learning (2020)

AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data

Nick Erickson* NEERICK@AMAZON.COM
Jonas Mueller* JONASMUE@AMAZON.COM
Alexander Shirkov ASHYRKOU@AMAZON.COM
Hang Zhang HZAWS@AMAZON.COM
Pedro Larroy PLLARROY@AMAZON.DE
Mu Li MLI@AMAZON.COM
Alexander Smola SMOLA@AMAZON.COM
Amazon Web Services

Abstract

We introduce AutoGluon-Tabular, an open-source[1] AutoML framework that requires only one line of Python to train highly accurate machine learning models on a raw tabular dataset such as a CSV file. Unlike existing AutoML frameworks that focus on model/hyperparameter selection, AutoGluon-Tabular ensembles multiple models by stacking them in multiple layers. Experiments reveal that our multi-layer combination of many models offers better use of allocated training time than seeking out the best. Using a suite of 50 classification/regression tasks from Kaggle and the OpenML AutoML Benchmark, we compare AutoGluon with various AutoML platforms including TPOT, H2O, AutoWEKA, auto-sklearn, and GCP AutoML Tables, and find that AutoGluon is faster, more robust, and more accurate. AutoGluon often even outperforms the best-in-hindsight combination of all of its competitors. In two popular Kaggle competitions, AutoGluon beat 99% of the participating data scientists after merely 4h of training on the unprocessed data.

1 Introduction

The most common application of machine learning today involves classification/regression with data stored in a structured table of numeric/string values (as in a CSV file). Automated Machine Learning (AutoML) frameworks enable painless deployment of machine learning that is accurate, robust, and easy to use/maintain. Due to their immense potential, many AutoML frameworks have emerged to automate such tasks (He et al., 2019; Truong et al., 2019). Prior AutoML focused almost exclusively on the task of Combined Algorithm Selection and Hyperparameter optimization (CASH), offering strategies to find the best model and its hyperparameters from a sea of possibilities (Thornton et al., 2013). CASH algorithms rely on brute-force search that often expends significant compute evaluating poor model/hyperparameter configurations no reasonable data scientist would consider.

We introduce AutoGluon-Tabular, an easy to use and highly accurate Python library for AutoML with tabular data. Instead of focusing on CASH, AutoGluon relies on advanced data processing, deep learning, and multi-layer stack ensembling. It automatically recognizes the data type in each column for fully automated ML, including special handling of text fields. AutoGluon ensembles many models in a multi-layered fashion via stacking/bagging.

1. `github.com/awslabs/autogluon`
* Equal Contribution.

2 AutoGluon-Tabular

Consider a structured dataset of raw values in a CSV file, `train.csv`, with the label values stored in a column named `class`. Here's how to train and test a model with AutoGluon:

```
from autogluon import TabularPrediction as task
predictor = task.fit("train.csv", label="class")
predictions = predictor.predict("test.csv")
```

Within `fit()`, AutoGluon automatically: preprocesses the raw data, identifies what type of prediction problem this is (binary/multi-class classification or regression), infers feature types, partitions data into various folds for model-training vs. validation, individually fits various models, and creates an optimized model ensemble that outperforms the individual trained models. `fit()` provides many additional options that may be specified, including:

- `hyperparameter_tune = True` optimizes hyperparameters of the individual models.
- `auto_stack = True` applies bagging and (multi-layer) stacking.
- `time_limits` controls the runtime of `fit()`.
- `eval_metric` specifies the metric used to evaluate predictive performance.

When comparing AutoGluon with other frameworks, we utilize `auto_stack`, `eval_metric`, `time_limits` (the latter two are matched in all frameworks), but we do *not* utilize hyperparameter tuning to demonstrate CASH is not necessary in a successful AutoML system. By default, `auto_stack` is recommended to maximize AutoGluon's predictive accuracy (it is left optional for users who want to quickly train smaller ensembles at the expense of accuracy).

Data Processing. When left unspecified by the user, the type of prediction problem at hand (binary vs. multi-class classification vs. regression) is first inferred by AutoGluon based on the types of values present in the label column, and we also infer the variable-types of the other features. AutoGluon relies on both model-agnostic data preprocessing that transforms the inputs to all models, and model-specific preprocessing that is only applied to a copy of the data used to train a particular model. Model-agnostic preprocessing begins by categorizing each feature numeric, categorical, text, or date/time. Uncategorized columns are discarded from the data, comprised of non-numeric, non-repeating fields with presumably little predictive value (e.g. UserIDs). Date/times are transformed into ordered numbers and the values of each text column are transformed into numeric vectors of n-gram features (only retaining n-grams with the highest occurrence to save memory). Missing discrete variables are assigned an `Unknown` category rather than being imputed, which helps AutoGluon handle unknown categories at test-time.

Types of Models. AutoGluon trains a curated set of models: neural networks, LightGBM boosted trees (Ke et al., 2017), CatBoost boosted trees (Prokhorenkova et al., 2018), and scikit-learn implementations of: Random Forests, Extremely Randomized Trees, and k-Nearest Neighbors. The neural network used by AutoGluon is depicted in Figure 1A and is similar to the models of Howard and Gugger (2020); Cheng et al. (2016). Our network applies a separate embedding layer to each categorical feature (Guo and Berkhahn, 2016) which enables this model to separately learn about each categorical feature before its representation is blended with other variables. These categorical embeddings are concatenated with the (normalized) numerical features into a large vector which is both fed into a 3-layer feedforward network and directly connected to the output predictions via a linear skip-connection.

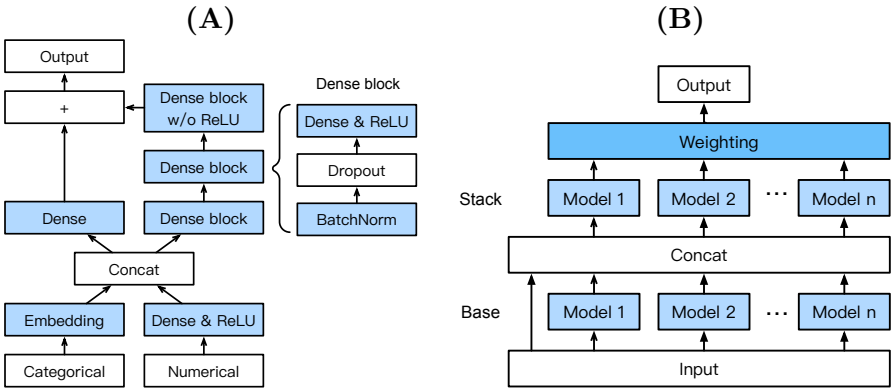


Figure 1: **(A)** Architecture of AutoGluon's neural network for numerical and categorical features. Blue layers contain learnable parameters. **(B)** AutoGluon's multi-layer stacking strategy, shown here using two stacking layers and n types of base learners.

Multi-Layer Stack Ensembling. Top AutoML frameworks all rely on model ensembles based on bagging, boosting, stacking, or weighted combinations (Dietterich, 2000). In particular, various AutoML frameworks utilize shallow stack ensembling (Wolpert, 1992). Here a collection of individual "base" models are individually trained in the usual fashion. Subsequently, a "stacker" model is trained using the aggregated predictions of the base models as its features. Multi-layer stacking feeds the predictions output by the stacker models as inputs to additional higher layer stacker models (Michailidis, 2018).

AutoGluon's multi-layer stacking strategy is depicted in Figure 1B. Here the first layer has multiple base models, whose outputs are concatenated and then fed into the next layer, which itself consists of multiple stacker models. These stackers then act as base models to an additional layer. AutoGluon simply reuses all of its base layer model types (with identical hyperparameters) as stackers. Like Chen et al. (2018), our stacker models take as input not only the predictions of the models at the previous layer, but also the original data features themselves (input vectors are data features concatenated with lower-layer model predictions). Our final stacking layer applies ensemble selection (Caruana et al., 2004) to aggregate the stacker models' predictions in a weighted manner.

Repeated k-fold Bagging. AutoGluon further improves its stacking performance by utilizing all of the available data for both training and validation, through k-fold ensemble bagging of all models at all layers of the stack. We randomly partition the data into k disjoint chunks (stratified based on labels), and subsequently train k copies of a model with a different data chunk held-out from each copy. AutoGluon bags all models and each model is asked to produce out-of-fold (OOF) predictions on the chunk it did not see during training. As every training example is OOF for one of the bagged model copies, this allows us to obtain OOF predictions from every model for every training example.

In stacking, it is critical that higher-layer models are only trained upon lower-layer OOF predictions. Our use of OOF predictions from bagged ensembles instead allows higher-layer stacker models to leverage the same amount of training data as those of the previous layer. While k-fold bagging efficiently reuses training data, it introduces a slight mismatch:

Algorithm 1 AutoGluon Training (with multi-layer stacking + n-repeated k-fold bagging)

Require: data (X, Y), family of models $\mathcal{M}$, # of layers L

1: Preprocess data to extract features
2: **for** $l = 1$ **to** L **do** {Stacking}
3: **for** $i = 1$ **to** n **do** {n-repeated}
4: Randomly split data into k chunks $\{X^j, Y^j\}_{j=1}^k$
5: **for** $j = 1$ **to** k **do** {k-fold bagging}
6: **for each** model type m in $\mathcal{M}$ **do**
7: Train a type-m model on X^{-j}, Y^{-j}, and ask it for predictions $\hat{Y}_{m,i}^j$ on OOF data X^j
8: Average OOF predictions over repeated bags $\hat{Y}_m = \{\frac{1}{n}\sum_i \hat{Y}_{m,i}^j\}_{j=1}^k$
9: $X \leftarrow$ concatenate$(X, \{\hat{Y}_m\}_{m\in\mathcal{M}})$

stacker models receive single models' OOF predictions as features during training, but during inference operate on lower-layer predictions averaged over the full bag. We propose a repeated bagging process to mitigate this, where if training time remains after k-fold bagging, we repeat the k-fold bagging process on n different random partitions of the training data, averaging all OOF predictions over the repeated bags (n chosen to fill specified time budget).

Overall Training Strategy. Our overall training strategy is summarized in Algorithm 1, where each stacking layer receives time budget T_{total}/L. By default (if `auto_stack` is specified), AutoGluon utilizes 2-layer stacking ($L = 2$) with repeated 10-fold bagging ($k = 10$), where the number of bagging repeats, n, is chosen to fill remaining time in the budget after the first round of k-fold bagging. In Step 7, AutoGluon first estimates the required training time and if this exceeds the remaining time for this layer, we skip to the next stacking layer (skipping any remaining models). After each new model is trained, it is immediately saved to disk for fault tolerance. This design makes the framework highly predictable in its behavior: both the time envelope and failure behavior are well-specified.

This approach guarantees that we can produce predictions as long as we can train at least one model on one fold within the allotted time. As we checkpoint intermediate iterations of sequentially trained models like neural networks and boosted/bagged trees, AutoGluon can still produce a model under meager time limits. We additionally anticipate that models may fail while training and skip to the next one in this event. Many AutoML frameworks train multiple models in parallel on the same instance. While this may sometimes save time, it leads to many out-of-memory errors on larger datasets without careful tuning. AutoGluon instead trains models sequentially and relies on their individual implementations to efficiently leverage multiple cores. This allows us to train where other frameworks may fail.

3 Experiments

We compare AutoGluon with popular AutoML frameworks: Auto-WEKA (Thornton et al., 2013), auto-sklearn (Feurer et al., 2015), TPOT (Olson et al., 2016), H2O AutoML (H2O.ai, 2017), GCP-Tables (Google, 2019). We run these tools on 50 curated datasets, spanning binary/multiclass classification and regression problems collected from two sources[2]. All

2. Code to reproduce our benchmarks is available at: `github.com/Innixma/autogluon-benchmarking`

Table 1: Comparing AutoML frameworks (with 4h training time) on: 39 datasets from OpenML AutoML Benchmark (top), 11 datasets from Kaggle (bottom). Listed are the number of datasets where each framework produced: better predictions than AutoGluon (Wins), worse predictions (Losses), a system failure/error (Failures), or better predictions than all of the other 5 frameworks (Champion). The latter 3 columns show the average: rank of the framework (among the 6 AutoML frameworks applied to each dataset), predictive performance (higher is better), and actual training time (in minutes). Averages are computed over the subset of datasets where all frameworks ran successfully. For Kaggle, predictive performance = percentile rank on each competition's leaderboard (proportion of teams beaten by AutoML). For OpenML, predictive performance = 1− rescaled loss, where we rescale the loss values for each dataset such that they span $[0, 1]$ among our AutoML frameworks.

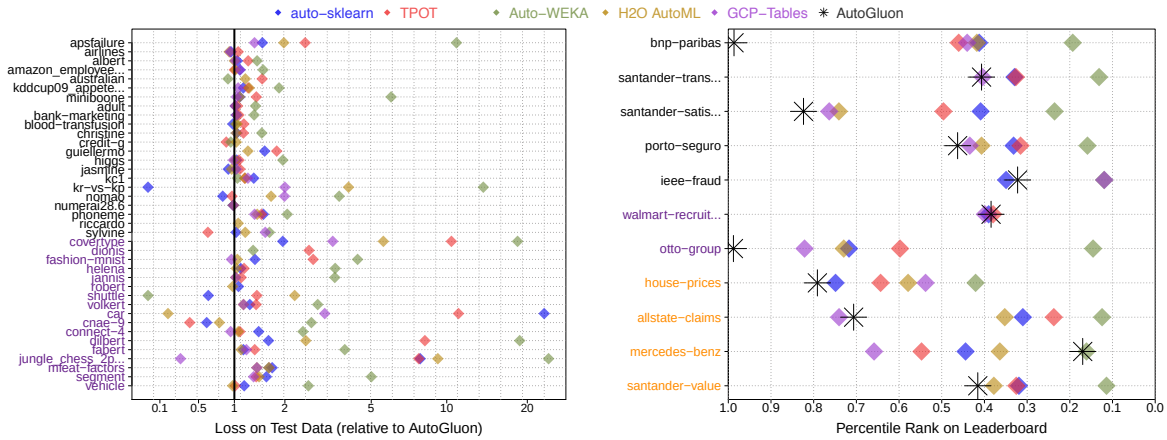
	Framework	Wins	Losses	Failures	Champion	Rank	Performance	Time
OpenML	AutoGluon	-	-	**1**	**23**	**1.8438**	**0.8615**	201
	H2O AutoML	4	26	8	2	3.1250	0.7553	220
	TPOT	6	27	5	5	3.3750	0.7966	235
	GCP-Tables	5	20	14	4	3.7500	0.6664	**195**
	auto-sklearn	6	27	6	3	3.8125	0.6803	240
	Auto-WEKA	4	28	6	1	5.0938	0.1999	244
Kaggle	AutoGluon	-	-	**0**	**7**	**1.7143**	**0.7041**	**202**
	GCP-Tables	3	7	1	3	2.2857	0.6281	222
	H2O AutoML	1	7	3	0	3.4286	0.5129	227
	TPOT	1	9	1	0	3.7143	0.4711	380
	auto-sklearn	3	8	**0**	1	3.8571	0.4819	240
	Auto-WEKA	0	10	1	0	6.0000	0.2056	221

AutoML frameworks were run with the same time limit on the same type of AWS EC2 cloud instance (except GCP-Tables which uses 92 GCP servers). As some frameworks only loosely respected specified time limits, we also report actual training times.

OpenML AutoML Benchmark. 39 binary and multi-class classification datasets curated by Gijsbers et al. (2019) to serve as a representative benchmark for AutoML frameworks. As in the original benchmark, we train for both 1h as well as 4h, and test set loss is calculated via $1 - \text{AUC}$ or log-loss for binary or multi-class classification tasks, respectively.

Kaggle Benchmark. 11 regression and binary/multiclass classification datasets chosen from recent Kaggle competitions to reflect real modern-day applications (see Table S1).

Results. Table 1 compares each framework against AutoGluon in these benchmarks, showing how *often* one framework is better than another. Figure 2 depicts the performance of each framework on individual datasets from our benchmarks, showing how *much* better each framework is than the others for particular applications. Overall, AutoGluon is much more accurate than all of the other AutoML frameworks, and it is the only framework with average rank $\leqslant 2$, indicating no other framework could beat AutoGluon consistently. On over half of the datasets in each benchmark, AutoGluon performed better than all of the



(A) AutoML Benchmark (1h) **(B)** Kaggle Benchmark (4h)

Figure 2: **(A)** Performance of AutoML frameworks relative to AutoGluon on the AutoML Benchmark (with 1h training time). **(B)** Proportion of teams in each Kaggle competition whose scores were beat by each AutoML framework (with 4h training time). The color of each dataset name indicates the task: binary classification (black), multi-class classification (purple), regression (orange).

Table 2: Ablation study of AutoGluon on the AutoML Benchmark (4h training time).

Framework	**Avg. Rank**	**Avg. Loss**
AutoGluon	**2.122**	**0.144**
NoRepeat	2.365	0.204
NoMultiStack	3.216	0.443
With-HPO	4.095	0.618
NoBag	4.446	0.658
NoNetwork	4.757	0.731

other frameworks. AutoGluon is additionally more robust (with less system failures) and better at adhering to the specified training time limits (Table 1, Figures S2-S3). After just 4 hours of training, AutoGluon placed 42 / 3505 and 39 / 2920 on the official leaderboards of the `otto` and `bnp-paribas` Kaggle competitions, respectively.

AutoGluon's performance continues to improve with additional training time, and does so more reliably than other frameworks which may start over-fitting (Table S3). We also study ablated AutoGluon variants with components sequentially removed: First, we omit repeated bagging, just using one round of k-fold bagging (*NoRepeat*). Second, we omit multi-layer stacking, so the resulting model ensemble only uses bagging and ensemble selection (*NoMultiStack*). Third, we omit bagging and rely on ensemble selection with only a single training/validation split of the data (*NoBag*). Fourth, we omit our neural network from the set of base models (*NoNetwork*). Table 2 shows how each component boosts performance. We also run *NoBag*-AutoGluon with hyperparameter tuning via random-search (*With-HPO*).

References

R. Caruana, A. Niculescu-Mizil, G. Crew, and A. Ksikes. Ensemble selection from libraries of models. In *ICML*, 2004.

B. Chen, H. Wu, W. Mo, I. Chattopadhyay, and H. Lipson. Autostacker: A compositional evolutionary learning system. In *GECCO*, 2018.

H.-T. Cheng et al. Wide & deep learning for recommender systems. In *Workshop on deep learning for recommender systems*, pages 7–10, 2016.

T. G. Dietterich. Ensemble methods in machine learning. In *International Workshop on Multiple Classifier Systems*, pages 1–15. Springer, 2000.

M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter. Efficient and robust automated machine learning. In *NIPS*, 2015.

P. Gijsbers, E. LeDell, J. Thomas, S. Poirier, B. Bischl, and J. Vanschoren. An open source AutoML benchmark. In *ICML AutoML Workshop*, 2019.

Google. Google Cloud AutoML Tables, 2019. URL `https://cloud.google.com/automl-tables`.

C. Guo and F. Berkhahn. Entity embeddings of categorical variables. *arXiv*, 2016.

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

X. He, K. Zhao, and X. Chu. AutoML: A survey of the state-of-the-art. *arXiv*, 2019.

J. Howard and S. Gugger. fastai: A layered api for deep learning. *arXiv*, 2020.

G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. LightGBM: A highly efficient gradient boosting decision tree. In *NIPS*, 2017.

M. Michailidis. StackNet, 2018. URL `https://github.com/kaz-Anova/StackNet`.

R. S. Olson, N. Bartley, R. J. Urbanowicz, and J. H. Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. In *GECCO*, 2016.

L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin. CatBoost: unbiased boosting with categorical features. In *NeurIPS*, 2018.

D. Rishi. Bringing Google AutoML to 3.5 million data scientists on Kaggle. *Google Cloud Blog*, 2019.

C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *KDD*, 2013.

A. Truong, A. Walters, J. Goodsitt, K. Hines, B. Bruss, and R. Farivar. Towards automated machine learning: Evaluation and comparison of AutoML approaches and tools. *arXiv*, 2019.

D. H. Wolpert. Stacked generalization. *Neural networks*, 5(2):241–259, 1992.

Appendix: AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data

Appendix A. AutoGluon Implementation Details

Default hyperparameter values for each model can be found in the AutoGluon source code: `github.com/awslabs/autogluon`, in directory: `autogluon/utils/tabular/ml/models/`. These default hyperparameter values were chosen a priori and not tuned based on the particular datasets used for benchmarking in this paper.

While boosted trees are often more accurate than neural networks on tabular datasets, properly trained neural networks can substantially improve accuracy when ensembled with other models, in particular providing useful diversity to the axis-aligned geometry of trees. Implemented using MXNet Gluon, the neural network in AutoGluon employs ReLU activations, dropout regularization, batch normalization, Adam with a weight decay penalty, and early stopping based on validation performance. While not tuned via hyperparameter optimization, the size of the hidden layers is nonetheless scaled adaptively based on properties of the training data (in a fixed manner). In particular, the feedforward branch of our model uses hidden layers of size 256 and 128 which are additionally scaled up based on the number of classes in multi-class settings. The width of the numeric embedding layer ranges between 32 - 2056 and is determined based on the number of numeric features and the proportion of numeric vs. categorical features. Inspired by Howard and Gugger (2020), a discrete feature with k unique categories observed in the training data is processed by an embedding layer of size: $1.6 \times k^{0.56}$ (up to threshold of size 100). AutoGluon using our proposed neural network architecture outperforms AutoGluon using an analogous standard feedforward network on 22 datasets of the AutoML Benchmark (and is outperformed on only 12 datasets).

To infer the type of prediction problem at hand (when unspecified), AutoGluon considers the label column's values. Non-numeric string values indicate a classification problem (with the number of classes equal to the number of unique values observed in this column), whereas numeric values with few repeats indicate a regression problem. This simple feature is just one example of the many AutoGluon optimizations that help users quickly translate raw data into accurate predictions. AutoGluon classifies each column's data type with the help of the `pandas` library, which it uses to load the raw data into Python. For string columns, how missing values are represented does not really matter since AutoGluon treats missing values as an extra `Unknown` category. For numerical columns, AutoGluon requires missing values be loaded by `pandas` as `NA`, and each model individually handles numerical missing values (for example, the Random Forest and Neural Network impute them, whereas CatBoost/LightGBM utilize special strategies listed in their documentation).

AutoGluon considers text features to be columns of mostly unique strings, which on average contain more than 3 non-adjacent whitespace characters. Columns with mainly repeated strings are treated as categorical. Columns with mostly unique strings and fewer whitespaces are dropped (as presumably ID-type columns with little predictive signal). Our n-gram text featurization ensures AG is lightweight and works well with any vocabulary (v.s. say contextual embeddings from bulky models like BERT).

Appendix B. Data used in Kaggle Benchmark

Table S1: 11 Kaggle competitions used in our benchmark, including: date of each competition, how many teams participated, and the number of rows/columns in training data. Metrics used to evaluate predictions in each competition include: root mean squared logarithmic error (RMSLE), coefficient of determination (R^2), mean absolute error (MAE), logarithmic loss (log-loss), area under the Receiver Operating Characteristic curve (AUC), and normalized Gini index (Gini).

Competition	**Task**	**Metric**	**Year**	**Teams**	**Rows**	**Colums**
house-prices-advanced-regression-techniques	regression	RMSLE	2020	5100	1460	80
mercedes-benz-greener-manufacturing	regression	R^2	2017	3800	4209	377
santander-value-prediction-challenge	regression	RMSLE	2019	4500	4459	4992
allstate-claims-severity	regression	MAE	2017	3000	1.8E+5	131
bnp-paribas-cardif-claims-management	binary	log-loss	2016	2900	1.1E+5	132
santander-customer-transaction-prediction	binary	AUC	2019	8800	2.2E+5	201
santander-customer-satisfaction	binary	AUC	2016	5100	7.6E+4	370
porto-seguro-safe-driver-prediction	binary	Gini	2018	5200	6.0E+5	58
ieee-fraud-detection	binary	AUC	2019	6400	5.9E+5	432
walmart-recruiting-trip-type-classification	multi-class	log-loss	2016	1000	6.5E+5	7
otto-group-product-classification-challenge	multi-class	log-loss	2015	3500	6.2E+4	94

Table S1 describes the datasets that comprised our Kaggle benchmark. Data for competition `x` can be obtained from: `kaggle.com/c/x/`. To select competitions for the benchmark, we first included those for which Google (2019); Rishi (2019) found GCP-Tables could produce strong results (indicating these are suitable candidates for AutoML). We omitted three datasets from Google (2019) in our benchmark because either the competition data/scores are unavailable, or the data require manual transformations to be formatted as a single table.

The remaining benchmark data were selected by optimizing for a mix of regression and binary/multiclass classification tasks with IID data (i.e. without temporal dependence), while favoring competitions that were either more recent (more timely applications) or had a large number of teams competing (more prominent applications). For each competition, every AutoML framework was trained on the provided training data, and its predictions on the provided (unlabeled) test data are submitted to Kaggle's server to evaluate their accuracy (using secret test labels). The performance reported in this paper is based on the private (rather than public) score from each Kaggle competition, which is used to decide the official leaderboard (except for `house-prices` which only offers public leaderboard).

Appendix C. Details Regarding Usage of AutoML frameworks

Our evaluations are based on running each AutoML system on every dataset in the exact same manner. Having to manually adjust tools to particular datasets would otherwise

undermine the purpose of automated machine learning. Before training, every frameworks is informed of the metric by which predictions will be evaluated.

Only H2O and GCP-Tables could robustly handle training with CSV files of raw data in our experiments (each utilizing their own automated inference of feature types). While Auto-WEKA aims to do the same, our experiments produced numerous errors when applying Auto-WEKA to raw data (e.g. when a new feature-category appeared in test data). To enhance its robustness, we provided Auto-WEKA with the same preprocessed data that we provided to TPOT and auto-sklearn. Lacking end-to-end AutoML capabilities, these packages do not support raw data input and require the data to be preprocessed. Thus, we provided Auto-WEKA, TPOT, and auto-sklearn with the same preprocessed version of each dataset, producing via the same steps AutoGluon uses to transform raw data into numerical features that are fed to certain models: Inferred categorical features are restricted to only their top 100 categories, then one-hot encoded into a vector representation with additional categories to represent rare categories, missing values, and new categories only encountered at inference-time. Inferred numerical features have their missing values imputed and then are rescaled to zero mean and unit variance. We find that given the AutoGluon-processed data, Auto-WEKA, TPOT, and auto-sklearn are able to match their performance in the original AutoML benchmark (Table S2), this time without requiring that the feature types have been manually specified for each package.

Where available, we used newer versions of each open-source AutoML framework than those Gijsbers et al. (2019) evaluated in the original AutoML Benchmark. In particular, we used TPOT version 0.11.1, Auto-WEKA 2.6, H2O 3.28.0.1, and auto-sklearn version[3] 0.5.2. For each of these AutoML libraries, we confirmed with the original package authors that any modifications we made to the default AutoML benchmark settings would be expected to maximize their predictive performance.

We followed the protocol of the original AutoML Benchmark and trained frameworks with 1h and 4h time limits. The Kaggle datasets tend to be larger than those of the AutoML Benchmark and posed memory issues for some of the baseline AutoML tools. To ensure no AutoML framework is resource-limited, we ran the Kaggle benchmark for longer than the AutoML datasets (4h and 8h time limits), and used more powerful AWS `m5.24xlarge` EC2 instances (384 GiB memory, 96 vCPU cores). For the AutoML Benchmark, we used the same machine as in the original benchmark, an AWS `m5.2xlarge` EC2 instance (32 GiB memory, 8 vCPU cores).

To ensure averaging over different datasets remains meaningful in the AutoML Benchmark, we report loss values over the test data that have been rescaled. We rescale the loss values for each dataset such that they span $[0, 1]$ among our AutoML frameworks. The *rescaled loss* for a dataset is set $= 0$ for the champion framework and $= 1$ for the worst-performing framework. The remaining frameworks are linearly scaled between these endpoints based on their relative loss. To ensure all head-to-head comparisons between frameworks remain fair, our reported averages/counts are taken *only* over those datasets where all frameworks trained without error.

3. While a newer 0.6.0 auto-sklearn version exists, it has `sckit-learn` dependency that is incompatible with AutoGluon preventing them from being installed together. There does not appear to be any updates to auto-sklearn's ML/modeling process between versions 0.5.2 and 0.6.0

Table S2: Comparing our usage of AutoML systems against the results from the original AutoML Benchmark (Gijsbers et al., 2019). Out of the 39 datasets, we count how often our implementation exceeded the original performance (>), or fell below the original performance (<), or was equally performant (=). Since there were no ties here, all missing counts are datasets where one framework failed. Rather than providing TPOT, auto-sklearn, and Auto-WEKA with information about the true feature types (as done in the original benchmark), we instead provided them with data automatically preprocessed by AutoGluon (without additional information about the ground-truth feature types). This shows AutoGluon's preprocessing is broadly useful, enabling these other AutoML methods to be applied in a more automated/robust manner to other datasets, without harming their performance.

System	**> Original**	**< Original**	**= Original**
H2O AutoML (1h)	18	16	0
auto-sklearn (1h)	16	14	0
TPOT (1h)	17	13	0
Auto-WEKA (1h)	18	12	0
H2O AutoML (4h)	15	15	0
auto-sklearn (4h)	15	17	0
TPOT (4h)	13	16	0
Auto-WEKA (4h)	17	12	0

Appendix D. Additional Results

Table S3: Performance of AutoML frameworks after 1h training vs. 4h training on each of the 39 AutoML Benchmark datasets. We count how many times the 1h variant performs better (>), worse (<), or comparably (=) to the 4h variant.

System	**> 4h**	**< 4h**	**= 4h**
AutoGluon (1h)	**5**	**30**	3
GCP-Tables (1h)	8	16	1
H2O AutoML (1h)	6	16	9
auto-sklearn (1h)	12	16	1
TPOT (1h)	6	24	2
Auto-WEKA (1h)	7	16	10

Table S4: Comparing each AutoML framework against AutoGluon on the 39 AutoML Benchmark datasets (with 1h training time). Listed are the number of datasets where each framework produced: better predictions than AutoGluon (Wins), worse predictions (Losses), a system failure during training (Failures), or better predictions than all of the other 5 frameworks (Champion). The latter 3 columns show the average: rank of the framework (among the 6 AutoML tools applied to each dataset), (rescaled) loss on the test data, and actual training time. Averages are computed over the subset of datasets where all methods ran successfully.

Framework	Wins	Losses	Failures	Champion	Rank	Rescaled Loss	Time (min)
AutoGluon	0	0	0	19	1.5455	0.0474	57
GCP-Tables	6	20	13	5	2.8182	0.2010	90
H2O AutoML	8	30	1	5	3.1818	0.1914	58
auto-sklearn	8	26	5	4	3.7273	0.2176	60
TPOT	5	30	4	4	4.0909	0.2900	67
Auto-WEKA	4	31	4	2	5.6364	0.9383	62

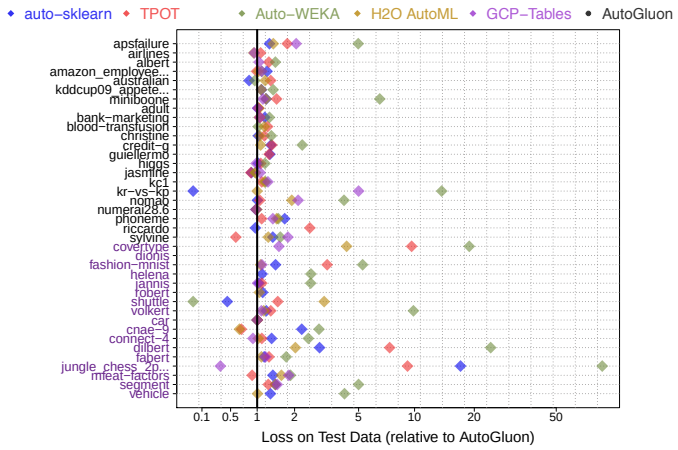
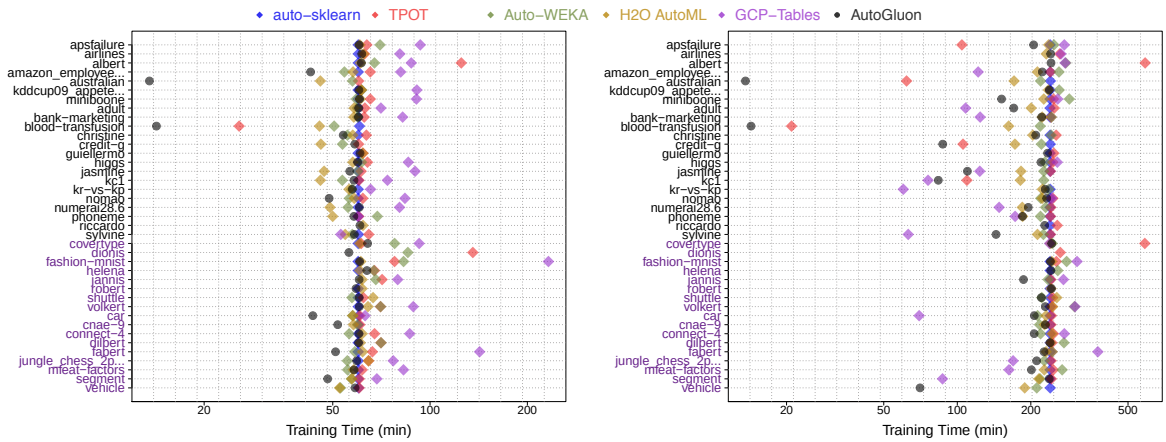


Figure S1: Performance of AutoML frameworks relative to AutoGluon on each dataset from the AutoML Benchmark (under 4h training time limit). Failed runs are not shown here (and we omit a massive loss of Auto-WEKA on `cars` as an outlier). Loss is measured via $1 - \text{AUC}$ for binary classification datasets (black text), or log-loss for multi-class classification datasets (purple text), and is divided by AutoGluon's loss here.



(A) 1h time limit specified (60 min)

(B) 4h time limit specified (240 min)

Figure S2: Actual training times of each framework in the AutoML Benchmark, which varied despite the fact that we instructed each framework to only run for the listed time limit. Unlike AutoGluon, some frameworks vastly exceeded their training time allowance (TPOT in particular). In these cases, the accuracy values presented in this paper presumably represent optimistic estimates of the performance that would be achieved if training were actually halted at the time limit. Datasets are colored based on whether they correspond to a binary (black) or multi-class (purple) classification problem.

Table S5: Ablation analysis of AutoGluon trained without various components on the AutoML Benchmark under 1h and 4h time limits. The ablated variants of AutoGluon are defined in the main text, columns are defined as in Table S4 (averaged columns are relative to this table and should not be compared across tables). Even after 4h, the *NoMultiStack* variant cannot outperform the full AutoGluon trained for only 1h.

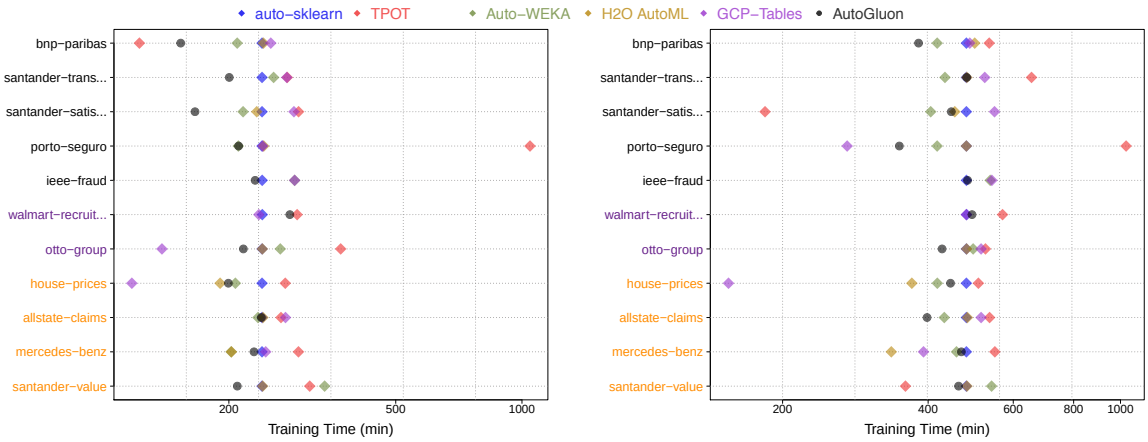
Framework	Wins	Losses	Champion	Rank	Rescaled Loss	Time (min)
AutoGluon (4h)	0	0	15	2.6757	0.1416	192
NoRepeat (4h)	17	20	11	3.3919	0.2106	114
AutoGluon (1h)	5	30	2	4.6351	0.3323	55
NoMultiStack (4h)	7	28	3	4.6622	0.4361	173
NoRepeat (1h)	5	32	2	4.9595	0.3600	43
NoMultiStack (1h)	6	31	2	6.1351	0.5868	53
NoBag (1h)	5	33	1	6.6351	0.6513	15
NoBag (4h)	5	33	1	7.0405	0.6605	27
NoNetwork (1h)	4	34	0	7.4189	0.7475	10
NoNetwork (4h)	5	33	0	7.4459	0.7431	16

Table S6: Comparing the open-source AutoML frameworks over all 10 different train/test splits of the AutoML Benchmark (with 4h training time limit). These results are based on the average performance across all 10 folds. A framework failure on any of the 10 folds is considered an overall failure for the dataset. Columns are defined as in Table S4, where averaged columns are relative to the particular table and should not be compared across tables. AutoGluon outperforms *all* other frameworks in 27 of the 38 datasets (`Dionis` dataset is excluded from this table because all frameworks failed on this massive dataset). Evaluating over 10 folds reduces variance and thus frameworks are less likely to get a strong/poor result by chance.

Framework	Wins	Losses	Failures	Champion	Rank	Rescaled Loss	Time (min)
AutoGluon	0	0	1	27	1.3684	0.0303	197
H2O AutoML	7	23	9	6	2.4737	0.0955	224
auto-sklearn	4	28	7	3	2.9474	0.1589	240
TPOT	3	27	9	2	3.3158	0.2093	236
Auto-WEKA	1	31	7	0	4.8947	0.9902	242

Table S7: Comparing open-source AutoML frameworks on all 10 folds of the AutoML Benchmark (with 4h training time limit). We include scores reported from the original AutoML Benchmark, indicated with (O). These results are based on the average performance across all 10 folds. A framework failure on any of the 10 folds is considered an overall failure for the dataset. Columns are defined as in Table S4, where the averaged columns are relative to the particular table and should not be compared across tables. AutoGluon outperforms *all* other frameworks in 24 of the 39 datasets. Even without access to the original feature type information which was provided in the original benchmark, AutoGluon is still able to outperform the other frameworks. Our runs of the other AutoML frameworks perform similarly to their original results, indicating feature type information can be inferred effectively in most cases. Note that the original runs failed fewer times than our runs. This is likely because the original AutoML Benchmark runs performed multiple retries of failed frameworks in an attempt to get a result, which we did not consider here.

Framework	Wins	Losses	Failures	Champion	Rank	Rescaled Loss	Time (min)
AutoGluon	0	0	1	24	1.8889	0.0391	195
H2O AutoML (O)	8	29	2	4	3.4444	0.0972	208
H2O AutoML	7	23	9	2	3.5000	0.0851	223
auto-sklearn (O)	6	31	1	2	4.6667	0.1385	246
auto-sklearn	4	28	7	2	4.7778	0.1427	240
TPOT (O)	7	29	3	5	4.7778	0.1519	247
TPOT	3	27	9	0	5.3889	0.1949	237
Auto-WEKA (O)	1	33	4	0	8.2222	0.8284	237
Auto-WEKA	1	31	7	0	8.3333	0.7194	242



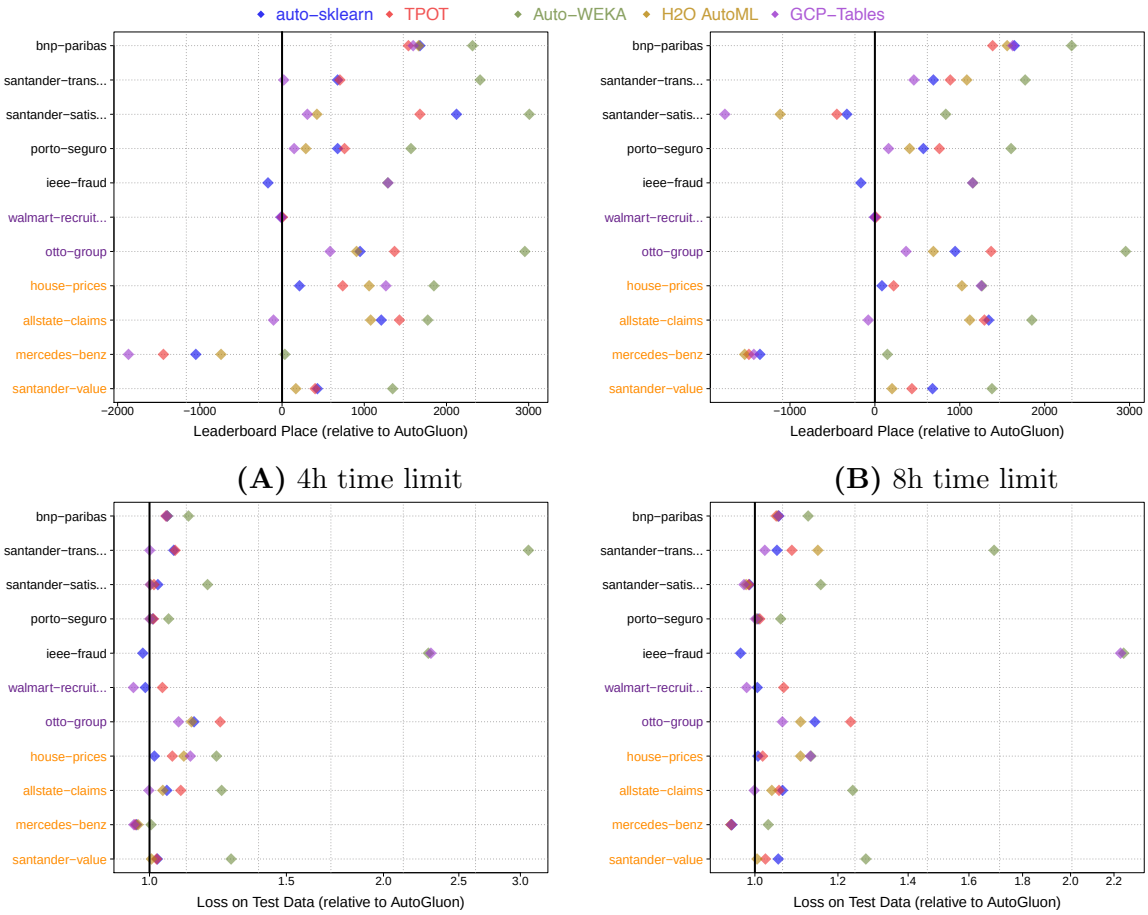
(A) 4h time limit specified (240 min)

(B) 8h time limit specified (480 min)

Figure S3: Actual training times of each framework in the Kaggle Benchmark, which varied despite the fact that we instructed each framework to only run for the listed time limit. Unlike AutoGluon, some frameworks vastly exceeded their training time allowance (TPOT in particular). In these cases, the accuracy values presented in this paper presumably represent optimistic estimates of the performance that would be achieved if training were actually halted at the time limit. The color of each dataset name indicates the corresponding task: binary classification (black), multi-class classification (purple), regression (orange).

Table S8: Comparing each AutoML framework against AutoGluon on the 11 Kaggle competitions (under 8h time limit). Listed are the number of datasets where each framework produced: better predictions than AutoGluon (Wins), worse predictions (Losses), a system failure during training (Failures), or more accurate predictions than all of the other 5 frameworks (Champion). The latter 3 columns show the average: rank of the framework (among the 6 AutoML tools applied to each dataset), percentile rank achieved in on the competition leaderboard (higher = better), and actual training time. Averages are computed over only the subset of 8 competitions where all methods ran successfully.

Framework	**Wins**	**Losses**	**Failures**	**Champion**	**Rank**	**Percentile**	**Time (min)**
AutoGluon	0	0	0	6	2.1250	0.6176	425
GCP-Tables	4	6	1	3	2.5000	0.5861	426
H2O AutoML	2	7	2	1	3.0000	0.5068	448
TPOT	2	8	1	0	3.5000	0.4793	565
auto-sklearn	3	8	0	1	3.8750	0.4851	480
Auto-WEKA	0	10	1	0	6.0000	0.2161	435



(A) 4h time limit

(B) 8h time limit

(C) 4h time limit

(D) 8h time limit

Figure S4: **(A)-(B)**: Difference in leaderboard ranks achieved by each AutoML framework vs. AutoGluon in the Kaggle competitions (under listed training time limit). These values quantify how much better one framework is vs. another, in terms of how many data scientists could beat one but not the other. **(C)-(D)**: Ratio of loss achieved by AutoML frameworks vs. AutoGluon loss on each Kaggle competition (under listed training time limit). Loss is a competition-specific metric (e.g. RMSLE, $1 - \text{Gini}$, etc.). In both plots, points > 0 indicate worse performance than AutoGluon and failed runs are not shown. The color of each dataset name indicates the task: binary classification (black), multi-class classification (purple), regression (orange).