Amazon SageMaker Autopilot: a white box AutoML solution at scale

Piali Das, Nikita Ivkin, Tanya Bansal, Laurence Rouesnel, Philip Gautier, Zohar Karnin, Leo Dirac, Lakshmi Ramakrishnan, Andre Perunicic, Iaroslav Shcherbatyi, Wilton Wu, Aida Zolic, Huibin Shen, Amr Ahmed, Fela Winkelmolen, Miroslav Miladinovic, Cedric Archembeau, Alex Tang, Bhaskar Dutt, Patricia Grao, Kumar Venkateswar
Amazon AI

ABSTRACT

We present Amazon SageMaker Autopilot: a fully managed system that provides an automatic machine learning solution. Given a tabular dataset and the target column name, Autopilot identifies the problem type, analyzes the data and produces a diverse set of complete ML pipelines, which are tuned to generate a leaderboard of candidate models that the customer can choose from. The diversity allows users to balance between different needs such as model accuracy vs. latency. By exposing not only the final models but the way they are trained, meaning the pipelines, we allow to customize the generated training pipeline, thus catering the need of users with different levels of expertise. This trait is crucial for users and is the main novelty of Autopilot; it provides a solution that on one hand is not fully black-box and can be further worked on, while on the other hand is not a do it yourself solution, requiring expertise in all aspects of machine learning. This paper describes the different components in the eco-system of Autopilot, emphasizing the infrastructure choices that allow scalability, high quality models, editable ML pipelines, consumption of artifacts of offline meta-learning, and a convenient integration with the entire SageMaker system allowing these trained models to be used in a production setting.

1 INTRODUCTION

Over the last decade and a half, Machine Learning (ML) has become an active area of research both in academia and industries. With the advent of cloud computing, compute resources became readily available, enabling scientists to explore techniques that were otherwise nearly impossible. Today, we have abundance of advanced algorithms and infrastructure that allow us to build various ML based smart solutions to benefit the business and social needs.

However, this abundance of machine learning algorithms and data preparation techniques also creates a difficulty: which one should you pick? How can you reliably figure out which model will perform the best for your specific business problem? In addition, machine learning algorithms usually have a long list of training parameters (also-known-as hyperparameters) that need to be set "just right" if you want to squeeze every bit of extra accuracy from your models. To make things worse, algorithms also require the data to be prepared and transformed in specific ways (i.e. feature engineering) for optimal learning. There are several other decision points that needs to be taken in the process; viz. picking the compute resources to ensure the model can be trained while still keeping the cost under control, how well the model generalizes, how efficiently, in terms of compute resources, can the model infer.

These decisions frequently require the expertise of both a data scientist and a software engineer. However, there is a lack of data science experts, and of machine-learning informed software engineers in the industry, but an over-abundance of data science problems. Even if the experts are available there is no escape from running plenty of trial and error experiments to find the optimal solution for the given data.

With a vision to reduce these repetitive development costs, the concept of automated machine learning (AutoML) has emerged in the recent years and has become a hot area of research.

Before proceeding further, it is important to describe an ML pipeline and the typical steps involved in building a good ML model. Any ML based solution has 2 phases — building a good ML model and using (a.k.a. deploying) that ML model. Figure 1 shows the typical stages of building a model.

(1) **Data Analysis:** Collecting statistics, such as missing entries, quantiles, skewness, correlation with the target and many others is useful for insights about the data and discovering bugs, faulty input or target leakage.
(2) **Problem Definition:** Looking at the target column in the raw dataset, determine which class of problem we are trying to solve? Is it a regression or classification problem?
(3) **Dataset schema detection:** For each column of the raw dataset identify the type of its content: numeric, categorical, natural language, timestamp, geolocation, etc.
(4) **Dataset meta-features extraction:** Extract datasets descriptors, such as number of rows and columns, sparsity, distribution of columns among feature types, landmark features [20] that identify performace of some ML model on the subsample, etc.

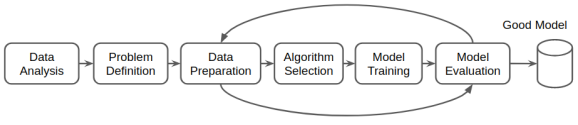

Figure 1: Stages of building a machine learning model

DEEM'20, June 14, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-8023-2/20/06.
https://doi.org/10.1145/3399579.3399870

(5) **Data Preprocessing:** How should data be prepared so that the model can predict as accurately as possible?
(6) **Algorithm Selection:** Which algorithm is best suited to solve the problem?
(7) **Model Training and Hyperparameter Tuning:** What is the optimal set of algorithm parameters that yields the best model?
   (a) Choose a set of Hyper Parameters.
   (b) Evaluate model against held out dataset.
   (c) Repeat steps (2) to (6) until a satisfactory model is obtained.

Note, that prediction quality is not the only target metric, real-life scenario imposes the series of constraints such as memory, latency, cost of each prediction and many others. After the model is build it need to be deployed to be used for inference. Handing over the model from science team to engineers is a complicated process due to limited expertise on each side. Thus, automating the deployment is crucial for AutoML solution.

Series of AutoML systems for tabular data were developed, both as academic packages [6, 9, 15, 18] and as industrial cloud-based solutions [4, 10, 11, 17]. Some of them target parts of the problem or require ML expertise, others work as a one-button back box. None of them meet all of the following criteria at the same time: transperency of generated artefacts, ease of build and deploy, scalability, feature preprocessing. For more details on the state of the field on automatic machine learning we refer reader to comprehensive surveys [25] and [12].

This paper describes Amazon SageMaker Autopilot, a scalable managed service that provides a reliable AutoML solution for tabular data. Following sections will describe the motivation behind the white-box approach, the description of the Autopilot system, its system architecture, the science behind the system and comparison of outcome with respect to other AutoML systems.

2 OUR APPROACH

The philosophy behind Autopilot is to offer a white box AutoML solution, with the vision to democratize machine learning. It aims to make the power for ML accessible to non-experts without requiring them to become an expert in this field first. The Autopilot aimed to be unique by satisfying the following requirements:

(1) **Educative, transparent and repeatable**
Autopilot demistifies the ML for the end-user with little to no expertise in the field and serves as a good starting point to apply the ML prediction power to the business problems. Autopilot helps to understand the real value of ML in specific scenarios avoiding more expensive and thus risky alternative of hiring a professional data scientist. In addition, its transparency serves an educational role, generated code of ML pipelines let users to educate themselves step-by-step, tranformer-by-transformer, and be guided through all the stages of ML development process. Ease of reproducibility and logging let users play with the system and see how the input characteristics such as adding more columns or records to the table, can change the quality of generated ML model.
(2) **Ability to allow an expert in loop**
Autopilot position itself as one-button white box solution. User can choose to trust the system entirely and rely on automatic decisions by only observing the intermediate artefacts, such as generated pipelines. However, if necessary, expert can be looped into the process, by overwriting some suggestions of the system as well as directly editing the code of the generated pipelines. For example, user can decide to introduce their own embedding algorithms as part of the pipeline. Whiteboxing all the generated code let customer keep improving the pipeline on-site or in more isolated environment.
(3) **Meant for users with different expertise**
Autopilot benefits ML savvy users by taking over the heavy lifting tasks of cleaning and prepping the dataset and engineering the features. Trained pipelines generated by the Autopilot are often used by experts as a baseline solution, which is further improved manually. Non-experts benefit from guided process with most decisions made automatically.
(4) **Production-level reliability and scalability**
While these are essential properties for any ML service nowadays, they are especially crucial for the non-expert user, which has little understanding what can potentially go wrong and which knobs might need adjustments.
(5) **Regression and classification problem type**
Autopilot covers two most common business driven machine learning problems on the tabular data: regression and classification. To accomodate non-expert users Autopilot can detect the problem type automatically and adjust quality metrics accordingly.

3 CORE FUNCTIONALITY

Amazon SageMaker Autopilot allows customers to quickly build classification and regression models without expert-level machine learning knowledge. To use Autopilot, customers issue a request that includes the following information:

- S3 path[1] to a CSV file
- the name of the target column to predict
- S3 location where output artifacts should be placed

Customers can optionally specify other parameters of the job, such as the problem type[2] and the computational budget.

Autopilot produces upto 250 consumable and ready-to-deploy models representing the entire ML pipelines. They can be sorted by various attribute including the prediction accuracy. Tangibly, Autopilot produces two kinds of artifacts: (1) artifacts to let customer inspect, modify and interactively re-generate the candidate models:

- a data preprocessor Python module using Scikit-Learn
- a SageMaker TransformJob model used to apply the associated data preprocessor module
- a trained and deployable SageMaker algorithm model paired with the data preprocessor
- train and validation folds of preprocessed dataset

(2) All intermediate and final artifacts that includes the following:

- train and validation folds of unprocessed dataset

[1] Amazon Simple Storage Service (Amazon S3) is an object storage service that offers industry-leading scalability, data availability, security, and performance.
[2] If the problem type is not specified, Autopilot detects it automatically.

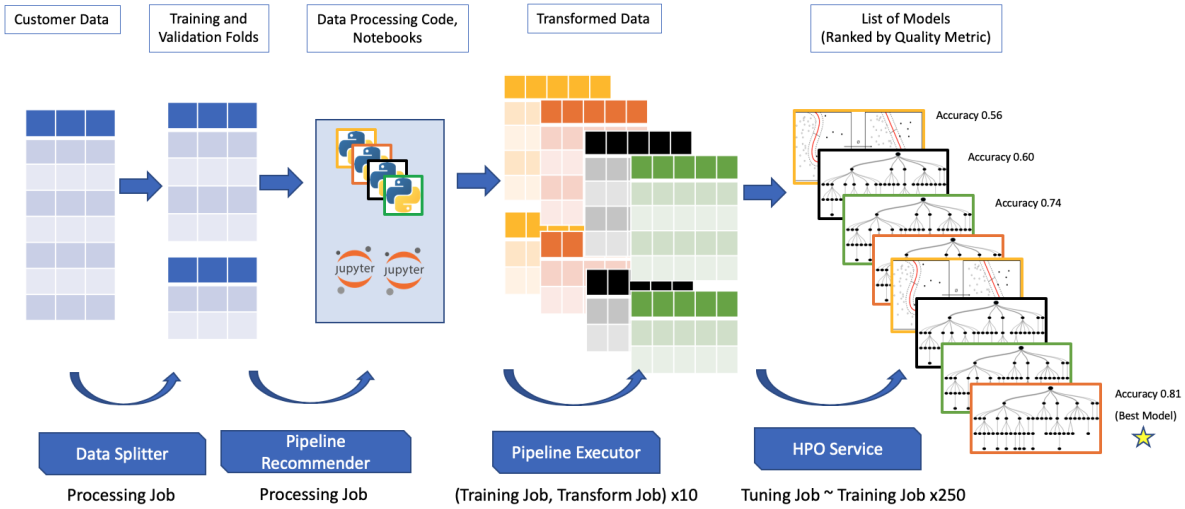

Figure 2: A high-level view of component jobs that Autopilot executes while processing a customer request.

- a Data Exploration Jupyter Notebook highlighting statistics about columns in the original dataset
- a Candidate Definition Jupyter Notebook that describes each candidate model, and allows the customer to easily customize and re-deploy any or all of them.

3.1 Internal workflow

Autopilot has two primary phases - candidate generation and candidate exploration. In the candidate generation phase, Autopilot does the following:

(1) splits the input data into training and validation;
(2) infers the problem type by analyzing the values in the target column (if not provided by user);
(3) generates a custom python module containing code to transform the input data in a upto 10 possible ways;
(4) generates 10 tunable ML pipeline definitions in form of a Jupyter notebook, referred to as Autopilot candidate definition notebook;
(5) generates a notebook with insights about the data.

In the candidate exploration phase, the generated pipelines are executed with sophisticated hyperparameter optimization techniques to find the set of ML pipelines that yields optimal prediction accuracy. At the end, Autopilot generates inference pipeline definitions for each of the candidates that can be deployed to production. Autopilot allows user to choose to execute only the candidate generation phase.

3.2 System Architecture

This section describes the underlying architecture of Autopilot and explain how each of the phases described in above section is accomplished.

Autopilot is built as an AWS managed service that manages the stateful entity named 'AutoMLJob' in SageMaker eco-system. It provides AWS API to manage life cycle of AutoMLJob. A separate "control plane" component handles user-driven control actions such as creating, describing, and stopping AutoML jobs. The operation of the system is handled by the "orchestrator" that executes several SageMaker compatible containers dedicated to perform specific actions in the Autopilot workflow. Figure 2 shows the internal flow and the system overview of Autopilot. Here we present brief descriptions of the components of Autopilot,

3.2.1 ***The control plane.*** The control plane is a high-availability, low-latency service that processes API requests from the AWS client. It exposes 4 main APIs that can be used by the customers with their programming language of choice. It lets customers to launch the job, query the status and stop the job via CreateAutoMLJob, DescribeAutoMLJob and StopAutoMLJob correspondingly. It provides validation of the request configuration and input data and performs authentication and authorization checks, based on the AWS Identity and Access Management (IAM) mechanism. Autopilot generates an abstract entity called "candidate", which represent an explored ML pipeline model. Customer can access the descriptions of all candidates via ListCandidatesForAutoMLJob API call.

3.2.2 ***The Orchestrator and first party containers.*** The orchestrator component is built using AWS Step Functions, which is modeled as a sequence of state machines. It orchestrates SageMaker Processing Job, Training Job, Transform Job and Hyperparameter Tuning Job to accomplish this workflow.

The first phase is candidate generation. It is accomplished by running two consecutive SageMaker Processing Jobs — Data Splitter and Candidate Generator, both implemented as a custom SageMaker compatible Docker Images, available in AWS Elastic Container registry. Data Splitter checks the data sanity, performs stratified shuffling and splits the data into training and validation. The Candidate Generator first streams through the data to compute statistics for the dataset. Then, uses these statistics to identify the problem type, and possible types of every column-predictor: numeric, categorical, natural language, etc.

Next, based on the characteristics of the data, the offline-trained 'recommendation' model predicts at most 10 different diverse unique pipelines strategies. Each pipeline strategy is a pair of data-processing strategy and algorithm along with their hyperparameters that can be tuned to generate a list of candidate models. The data-processing strategy gets translated to python scripts customized for the given dataset. Each such python script implements a Scikit-Learn Pipeline composed of native Scikit-Learn transformers [3, 16] and custom built transformers available in the open-source python library, sagemaker-scikit-learn-extension [1] developed by us. The pipeline strategy is then translated as an ordered listed of SageMaker job definition that are rendered as code in a Jupyter notebook referred to as Candidate Generation Notebook. This notebook can be executed to generate the candidate models. The SageMaker job definitions are complete and includes the resources configuration. This phase also generates the Data Analysis Notebook that provides insightful statistics about the dataset for the customer. This concludes the candidate generation phase. If the customer elects to generate the pipelines only, the state-machine stops here and mark the AutoMLJob as completed.

The second phase is candidate exploration. Note, that feature preprocessing part of each pipeline has all hyper parameters fixed, i.e. does not require tuning, thus feature preprocessing step can be done prior runing the hyper parameter optimization job. To accomplish it, each python script code for data-processing is executed inside a SageMaker framework container as a training job, followed by transform job. It outputs upto 10 variants of transformed data, therefore algorithms for each pipeline are set to use the respective transformed data. All algorithms are optimized using SageMaker Hyperparameter Optimization job. Up to 250 training jobs are selectively executed to find the best candidate model.

3.2.3 ***Generated Artifacts and customization.*** One of the advantages for Autopilots as mentioned in previous section is transparency. Hence, it is important to provide details of the artifacts generated, particularly the python scripts for data-processing and the Candidates Definition Notebook and how these can be inspected and customized to impart expert knowledge. As shown in Figure 2 all generated artifacts are available in the customer's S3 bucket.

The Candidate Definition Notebook, as briefly mentioned in the previous section, has the SageMaker job definitions that are executed. The notebook can be fetched from the AWS S3. As the first step, it downloads the python scripts from S3. It allows customers to inspect and customize each of the data-processing scripts locally. Figure 3 shows the definition of one of the pipelines. It provides an intuituve description to each of the pipeline and exposes its parameters and configuration. Thus it allows customers to select the pipelines to explore, modify the feature and label processing techniques used in the pipeline, specify the resource configuration for each stage of the pipeline. It allows customizing the algorithms' configurations including static hyperparamter values, choice of the tunable hyperparamters and their respective ranges and the metric to optimize for.

Thus we described how Autopilot provides solution for customers with varying degree of experise: a reliable 1-click solution for those who are happy with out-of-the-box solution and all the knobs to tweak the pipelines for experts to suffice their need.

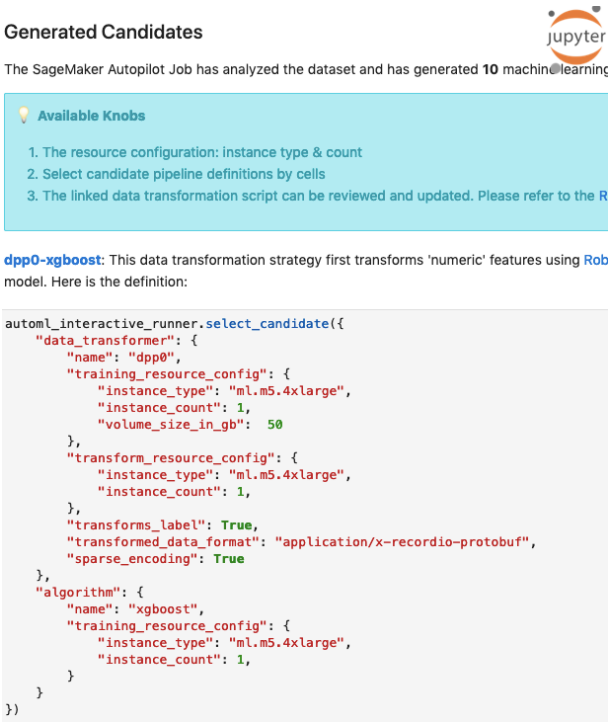

Figure 3: Snippet of Candidate Definition Notebook generated by Autopilot that describes the ML pipelines that can be modified and re-run.

4 PIPELINE SELECTION PROCESS

In Autopilot we differentiate between strategies and pipelines: in its nature strategy is more conditioned on the content and characteristics of the dataset, it includes if-else statements and symbolic hyper parameters [19]. For example consider two strategy transformers:

(a) if input column is numeric and there is more than X_1 entries beyond three standard deviations $\rightarrow$ perform quantile tranform with X_2 bins;
(b) if number of columns is larger than $X_3 \rightarrow$ apply PCA dimension reduction with $k = X_4 \cdot$ (number-of-columns);

where $X_{1,2,3,4}$ are the hyper parameters of the strategy. In contrary, pipeline transformers can be seen as a realization of the strategy transformer:

(a) apply quantile transform with X_2 bins to 13-th column;
(b) apply PCA dimension reduction with $k = 4$;

The high-level structure of the strategy can be seen as a sequence of single-column transformers followed by a sequence of mutli-column tranformers. Example strategy is depicted on Figure 5.

The choice of transformers from the pool of predesigned ones and its internal settings can be seen as hyper parameters (HP) of the strategy: quantile binning or log tranform for numeric columns, XGBoost or Linear Learner, number of components in PCA, maximum tree debth in XGBoost, and many others.

Hyper parameter optimization (HPO) traditionally considered as the most expensive part of the ML development cycle, as each HP configuration trial includes training of the algorithm on the

```python
def build_feature_transform():
    """ Returns the model definition representing feature processing."""

    # These features can be parsed as numeric.
    numeric = HEADER.as_feature_indices(
        [
            "Account Length", "Area Code", "VMail Message", "Day Mins",
            "Day Calls", "Day Charge", "Eve Mins", "Eve Calls", "Eve Charge",
            "Night Mins", "Night Calls", "Night Charge", "Intl Mins",
            "Intl Calls", "Intl Charge", "CustServ Calls"
        ]
    )

    # These features contain a relatively small number of unique items.
    categorical = HEADER.as_feature_indices(["Int'l Plan", "VMail Plan"])

    # These features can be parsed as natural language.
    text = HEADER.as_feature_indices(["State", "Phone"])

    numeric_processors = Pipeline(
        steps=[
            (
                "robustimputer",
                RobustImputer(strategy="constant", fill_values=nan)
            )
        ]
    )

    categorical_processors = Pipeline(
        steps=[
            ("thresholdonehotencoder", ThresholdOneHotEncoder(threshold=24))
        ]
    )

    text_processors = Pipeline(
        steps=[
            (
                "multicolumntfidfvectorizer",
                MultiColumnTfidfVectorizer(
                    max_df=0.9684,
                    min_df=0.0021002100210021,
                    analyzer="word",
                    max_features=10000
                )
            )
        ]
    )

    column_transformer = ColumnTransformer(
        transformers=[
            ("numeric_processing", numeric_processors, numeric
            ), ("categorical_processing", categorical_processors,
                categorical), ("text_processing", text_processors, text)
        ]
    )

    return Pipeline(
        steps=[
            ("column_transformer", column_transformer
            ), ("robuststandardscaler", RobustStandardScaler())
        ]
    )
```

Figure 4: One of the data-processing python scripts custom generated by Autopilot for the given dataset. It shows the SKLearn Pipeline representing the transformation for each feature in the input dataset.

entire dataset from scratch. Most widely adopted HPO engines (SMAC [13], ParamILS [14], BOHB [7]) are based on Bayesian Optimization (BO), where probabilistic surrogate model that maps HP configuration to loss is used to draw a next HP configuration trial. Initialization of the surrogate model takes portion of the optimization budget and is performed via Random Search HPO. BO is superior to Random HPO when HP search space is small and it was shown in practice by optimizing HP for the algorithms XGBoost or MLP. In our application, feature preprocessors introduce large number of HPs in addition to those from the algorithm, and many of those HPs are conditional.

In such settings, BO fails to initialize the surrogate model and performs on par with Random HPO. Both methods require high budgets to achieve reasonable performance. Applying either of

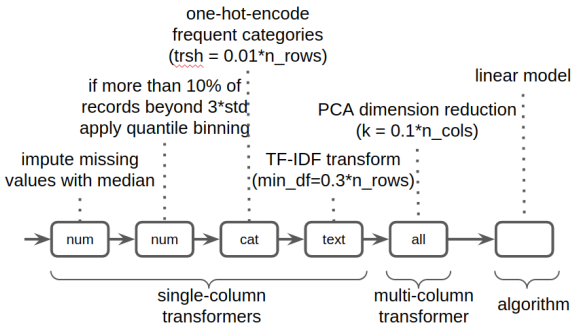

Figure 5: Example strategy includes three single column transformers, one multi-column transformer and the algorithm. Each single-column transformer preprocesses all columns of specified type (numeric, categorical of text).

them directly on arrival of customer dataset is prohibitively expensive. We address this problem by splitting the set of HPs into two parts: feature preprocessing and algorithm. We pre-optimize feature preprocessing HPs in offline manner and optimize algorithm HPs online using BO. This approach lets us leverage the power of transfer learning and thus keep the HPO cost low.

It was previously shown that default hyper parameters often outperform random search when number of HPO iterations is low. We adopt the zero-shot learning approach first introduced in [23, 24] and aim to find not one, but a set of $k = 10$ strategies, such that for a given customer dataset, at least one of them performs well.

To do so we perform an offline optimization:

- First, we build a performance table P by evaluating B randomly chosen HP configurations on D datasets, such that P_{ij} is the loss of i-th HP configuration on j-th dataset.
- Second, we find $k = 10$ HP configurations that minimize loss on all datasets as a group:

$$\min_{I} \left(\sum_{j \in D} \min_{i \in I} P_{ij} \right),$$

  where I is a subset of k strategies from B randomly chosen earlier.

The HPO of the algorithm hyper parameters stays online and performed for each end-user dataset. HPO is combined with choosing the one out of k strategies via colaboration of Bayesian Optimization and bandit approach. At first, 5 HP configurations drawn at random are evaluated for every pipeline. After this stage, a simple ϵ-greedy bandit algorithm [22] is used to select a among all pipelines, for which to generate candidate HP configuration of the estimator algorithm. Then, proposed HP configuration is evaluated, and the algorithm returns again to bandit based selection of HP configuration. This cycle repeats until budget of HP configurations to be evaluated is exhausted. Empirically, the behavior of this approach leads the tuning procedure first exploring different hyperparameter values and feature preprocessing / algorithm pairs, but once random exploration finishes, the tuning procedure focuses mostly on the tuning "the most promising" pipeline.

Parameter	Value
Regression datasets	50
Binary classification datasets	58
Multiclass classification datasets	67
Number of rows (range)	$[10^3, 5 \cdot 10^6]$
Number of columns (range)	[2, 7201]
Maximum number of classes	2428
Number of datasets with text columns	34
Size (range in Mb)	[0.02, 5000]

Table 1: Parameters of the datasets collection used for benchmarks.

5 RESULTS

We evaluate the perfromance of Autopilot on the collection of 176 datasets, obtained majorly from the open source repositories, such as UCI [5], Kaggle [2], and OpenML [21]. Collection inlcudes wide spectrum of datasets diverse in size, content, problem type etc. Each dataset represents a valid customer use-case. See some descriptive statistics of the datasets chosen in Table 1.

We benchmark Autopilot with several baseline solutions. Custom feature preprocessing (or feature engineering, FE) was used to accomodated baselines with limited preprocessing abilities, such as no handling for text or categorical input. Custom FE first detects the type of each column using simple set of heuristics: if all feature values can be converted to numerical values, then the feature is considered to be of numeric type; if it has less than 20 unique values, it is considered to be of categorical type, and if there are English words present in the values of features - it is of text type; in other cases, feature is ignored. For numerical columns, missing values are imputed with mean, and values are standardized to have variance of one. For categorical columns, one - hot encoding is used. TF-IDF features are extracted from the text.

Our benchmarks include the following algorithms:

- **Autopilot**
  Autopilot run with a budget of 250 candidates, and maximum 10 hours total runtime.
- **auto-sklearn [8], ensemble and defaults on**
  After training is complete, auto-sklearn automatically creates an ensemble of models that were found by Bayesian Optimization to further improve the performance. auto-sklearn has an option to warmstart Bayesian Optimization using configurations, which performed well on previously seen datasets, found by a meta-learning step, which is enabled in this configuration.
- **auto-sklearn [8], ensemble and defautls off**
  Same as above, except that ensemble construction and metalearning initialization (defaults) of the hyperparameter optimization procedure is disabled, to access the effect of such features on performance and stability.
- **XGBoost with custom Feature Engineering (FE)**
  Running XGBoost with single hyperparameter configuraiton, using equivalent of the latest version of feature preprocessing in Autopilot. One of the purposes of this step is to validate that it is possible for an AutoML procedure to succeed on all of the datasets used for our evaluation. Additionally, performance with such approach indicates if there is any value in running a hyperparameter procedure as used in Autopilot.

All of the metrics used for evaluation of the service were designed in such a way that they generalize across regression and classification. To surface different aspects of performance of Autopilot, the following metrics are calculated:

- **Relative Error Difference** Difference in normalized error metric. The error metric is error rate for classification and RMSE for regression. It is defined as (A-B)/max(A, B), where A is a score of method, and B is a score of a baseline. The values of relative error are averaged across all datasets. This metric is affected by the slight changes in absolute values of the evaluated and baseline methods.
- **Job Success Rate**
  How many of the jobs successfully completed end-to-end, meaning that an ML model was produced, and it was possible to score the model.
- **% matching baseline**
  How many datasets did the evaluated automl approach match or improve upon the baseline method.
- **Best models produced**
  How many times did the automl approach produce the best performing model. Note that the numbers need not necessarily sum up to total number of datasets, as some automl approaches might produce equivalent models, as well as for some datasets all the methods compared failed. Note that this metric depends less on slight changes in the error metrics. This metric depends less on slight changes in the error metrics.
- **Avg. inference endpoint latency**
  This metric corresponds to the time it takes to process a request to a server, where request contains a batch of 100 rows of test set data, and server performs inferences with the best found model. For the baselines compared to the Autopilot, a Flask application was used to serve the best found models; Autopilot model was deployed as a SageMaker endpoint. The latency of the SageMaker endpoint also includes overhead due to authentication. This metric allows to compare latency change with different model types produced by AutoML.

Comparison of Autopilot with other baselines can be found in Table 2. Note that there is a tremendous variety in contents of different datasets, and hence it is not trivial to ensure that an AutoML succeeds for all possible "edge cases" of data, hence we do not achieve perfect success rate with the current version of the Autopilot service, which is a subject of an ongoing work. Aside from this, we see that Autopilot outperforms the naive baseline of XGBoost with default hyperparameter values and performs on - par with baselines we have used, when no ensemble construction is used, which is currently not supported in Autopilot. The table shows that ensembles provide a boost to predictive performance, but at the expense of training time, model complexity and inference latency.

6 CONCLUSION

We discussed the importance of transparency and customization when building the AutoML solution and introduced new AutoML framework SageMaker Autopilot. It provides both ease of one-click end-to-end solution and flexibility to change the generated code

↑ – higher is better; ↓ – lower is better		Autopilot	Auto-sklearn ensemble and defaults off	Auto-sklearn ensemble and defaults on	XGBoost custom FE
Job Success Rate	↑	93.2%	74.4 %	82.4%	100.0%
% matching the baseline	↑	78.7%	74.0%	91.0%	100.0%
RED, relative error difference	↓	$-10.8(\pm 4.29)\%$	$-7.9(\pm 4.67)\%$	$-15.2(\pm 4.08)\%$	$+0.0(\pm 0.0)\%$
Best models produced	↑	78	24	80	23
Best models produced, no ensemble	↑	97	53	-	37
Average AutoML Job Runtime	↓	12119.0 s	36011.2 s	36123.9 s	19.9 s
Average Model Endpoint Latency	↓	97.8 ms	76.4 ms	939.2 ms	38.8 ms

Table 2: Performance of Autopilot compared to chosen baseline algorithms. Reference for the relative error as well as for the percentage of matching error is the performance of XGBoost.

and build on top. It guides novice user and let ML savvy customer to hand over the heavy lifting parts of ML pipeline development. We evaluated our solution with several baselines on a large collection of datasets.

7 ACKNOWLEDMENTS

Autopilot has contributions from several members of SageMaker team, notably Peter Liu, Furkan Bozkurt, Ugur Adiguzel, Per De Silva, Sylvia Wang, Pei Xu, Vikram Rajashekharan, Isabel Panapento.

REFERENCES

[1] 2019. sagemaker-scikit-learn-extension. https://github.com/aws/sagemaker-scikit-learn-extension/.
[2] 2020. Kaggle.com. http://kaggle.com.
[3] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. 2013. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 108–122.
[4] DataRobot. [n.d.]. *Automated Machine Learning*. https://www.datarobot.com/platform/automated-machine-learning/.
[5] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml
[6] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. 2020. AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data. *arXiv preprint arXiv:2003.06505* (2020).
[7] Stefan Falkner, Aaron Klein, and Frank Hutter. 2018. BOHB: Robust and efficient hyperparameter optimization at scale. *arXiv preprint arXiv:1807.01774* (2018).
[8] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and Robust Automated Machine Learning. In *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (Eds.). Curran Associates, Inc., 2962–2970. http://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning.pdf
[9] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. 2019. Auto-sklearn: efficient and robust automated machine learning. In *Automated Machine Learning*. Springer, 113–134.
[10] Google. 2019. *AutoML Tables*. https://cloud.google.com/automl-tables/.
[11] H2O.ai. 2017. *H2O AutoML*. http://docs.h2o.ai/h2o/latest-stable/h2o-docs/automl.html.
[12] Xin He, Kaiyong Zhao, and Xiaowen Chu. 2019. AutoML: A Survey of the State-of-the-Art. *arXiv preprint arXiv:1908.00709* (2019).
[13] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2010. Sequential model-based optimization for general algorithm configuration (extended version). *Technical Report TR-2010–10, University of British Columbia, Computer Science, Tech. Rep.* (2010).
[14] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. 2009. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research* 36 (2009), 267–306.
[15] Randal S Olson and Jason H Moore. 2019. TPOT: A tree-based pipeline optimization tool for automating machine learning. In *Automated Machine Learning*. Springer, 151–160.
[16] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
[17] Salesforce. 2018. *TransmogrifAI*. https://transmogrif.ai/.
[18] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2013. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. 847–855.
[19] Jan N van Rijn, Florian Pfisterer, Janek Thomas, Andreas Muller, Bernd Bischl, and Joaquin Vanschoren. 2018. Meta Learning for Defaults–Symbolic Defaults. In *Neural Information Processing Workshop on Meta-Learning*.
[20] Joaquin Vanschoren. 2018. Meta-learning: A survey. *arXiv preprint arXiv:1810.03548* (2018).
[21] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. 2013. OpenML: Networked Science in Machine Learning. *SIGKDD Explorations* 15, 2 (2013), 49–60. https://doi.org/10.1145/2641190.2641198
[22] John White. 2012. *Bandit algorithms for website optimization*. " O'Reilly Media, Inc.".
[23] Martin Wistuba, Nicolas Schilling, and Lars Schmidt-Thieme. 2015. Learning hyperparameter optimization initializations. In *2015 IEEE international conference on data science and advanced analytics (DSAA)*. IEEE, 1–10.
[24] Martin Wistuba, Nicolas Schilling, and Lars Schmidt-Thieme. 2015. Sequential model-free hyperparameter tuning. In *2015 IEEE international conference on data mining*. IEEE, 1033–1038.
[25] Marc-André Zöller and Marco F Huber. 2019. Survey on automated machine learning. *arXiv preprint arXiv:1904.12054* 9 (2019).