AFRL-RI-RS-TR-2021-137

AUTOMATIC COMPOSITION OF COMPLEX PIPELINES VIA END-TO-END LEARNING AND USER INTERACTION

TEXAS A & M ENGINEERING EXPERIMENT STATION

AUGUST 2021

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2021-137 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
PETER J. ROCCI
Work Unit Manager

/ S /
SCOTT D. PATRICK
Deputy Chief,
Intelligence Systems Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
AUGUST 2021	FINAL TECHNICAL REPORT	MAR 2017 – MAR 2021

4. TITLE AND SUBTITLE

AUTOMATIC COMPOSITION OF COMPLEX PIPELINES VIA END-TO-END LEARNING AND USER INTERACTION

5a. CONTRACT NUMBER
FA8750-17-2-0116

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
63706E

6. AUTHOR(S)

Xia Hu
Shuai Huang
Diego Martinez-Garcia

5d. PROJECT NUMBER
D3ME

5e. TASK NUMBER
00

5f. WORK UNIT NUMBER
03

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Texas A & M Engineering Experiment Station
400 Harvey Mitchell Pkwy South,
College Station TX 77845-3112

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RIED
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2021-137

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Automated machine learning (AutoML) has recently gained attention due to the ability to help data science experts rapidly create prototypes of machine learning solutions. The existing frameworks have shown an improvement of performance on common tasks such as tabular classification and regression. In the real world, it is challenging to decide which AutoML framework to use for a given task since there is no standard method to evaluate them. This report presents our work on providing utilities for the standardization of AutoML components such as evaluation tools, generalization of a pipeline language, standardization of pipeline execution, and general-purpose code creation. Also, we present different AutoML systems created during the different stages of the program and the creation of specialized frameworks for Neural architecture search and time-series outlier detection.

15. SUBJECT TERMS

Automated Machine Learning, Neural Architecture Search, Framework

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**PETER J. ROCCI**
U	U	U	UU	39	19b. TELEPHONE NUMBER (*Include area code*) **315-330-4654**

Table of Contents

List of Figures

List of Tables

1. SUMMARY

This report provides a summary of all of the achievements and contributions to the Data-Driven and Discovery of Models (D3M) program that includes contributions to the common infrastructure, the evolution of our Automated Machine Learning (AutoML) system, and research work that resulted from the project. During the program, we became one of the core contributors that maintained the D3M core package, created tools for evaluating AutoML systems, developed three AutoML systems, developed primitives widely used by multiple systems, designed an automated framework for testing primitives, and have a great impact on the AutoML community by the creation of AutoKeras.

2. INTRODUCTION

Automated machine learning has been rapidly progressing through the years due to multiple factors, such as the need for data scientists to create a prototype of machine learning pipelines to decide how to proceed with their solution and provide solutions for non-experts. Several AutoML frameworks have been created, but they are restricted by the problem types they can solve, the number of machine learning primitives they create, the pipeline representation language, and the tight data descriptions. Most of these restrictions are caused by considerable engineering effort. The D3M program aims to broaden the scope of AutoML by providing the tools needed to create AutoML systems capable of solving problem types beyond most of the framework and providing users with tools to enable machine learning tools with not much expertise. Also, the program works toward the standardization of AutoML components so different frameworks could be compared with fairness as well as helping the community to improve the field by sharing the infrastructure created during the program via open-source.

Our work on D3M focused on two main aspects: the creation of utilities for standardizing different works in the AutoML community within the D3M group and the creation of AutoML systems and frameworks with different purposes. In this report, we will be presenting our main contributions to the program and the evolution of our AutoML system.

3. METHODS, ASSUMPTIONS, AND PROCEDURES

By the end of the program we produced significant results and outcomes in diverse areas of AutoML. Firstly, we were core contributors on adding new features and maintain all the shared code by working with JPL and UBC. Also, we develop different AutoML systems through the program taking different approaches based on the different requirements. Besides that, our work also focuses on Neural architecture search in which we made some progress, although we managed to integrate it with D3M we could not get the best of it due to some engineering challenges related to the design of the common code.

3.1. D3M Infrastructure

Through the program, we actively engage and contribute to all working groups. We became one of the core contributors for multiple repositories such as the D3M core package [1], TA3-2 API [2], common primitives [3], dummy-ta3 [4], Keras wrapper [5], metalearning database. Also, we closely worked with different performers. We integrated our TA2s with TUFT, Two Ravens, NYU, Uncharted. We help to wrap primitives from multiple performers: JPL, BBN, CMU, UBC, RPI, JHU. And we integrated Royal Caliber pipeline acceleration with our system. Our main contributions to the program were the contributions to the creation of the Pipeline Schema, the creation of the reference runtime, integration of OpenML work into D3M, creation of primitives, the creation of tools to evaluate TA2s, tools to validate primitives, and the maintenance of core D3M libraries.

3.1.1 Pipeline Schema. At the beginning of the program each performer could use different tools to create their system, thus made it impossible to create a fair comparison among different AutoML systems, for this reason, the D3M group decided to create some common infrastructure for everyone to share. The main goal of TA2s performers was to generate machine learning pipelines that solve a specific problem, such as a tabular classification, regression, etc. One of the first steps to achieve fair comparison among systems was to create a common language for describing pipelines. For several months we discussed the different pros and cons of different pipeline representation, the most common one was the one proposed in AutoSklearn [6], in which pipelines are just a linear sequence of steps to follow. On D3M context was not enough, since performers wanted to be able to express in the pipeline language the ensemble of multiple pipelines among other things, so for that reason, we decided to represent pipeline as Direct Acyclic Graphs that later will be adopted by other ML Frameworks such as OpenML [7] and the Machine Learning Bazaar [8]. We were closely involved in the discussion about the pipeline representation by proactively proposing and discussing ideas.

3.1.2 D3M Runtime. After the working group formalized the pipeline representation and the first version of the TA1-2 API was released to unify the different work of TA1 performers, we decided to create de reference runtime. The reference runtime allows executing pipelines on given datasets and problems so that the output pipelines of the AutoML systems could be run by a third party to obtain replicable results. Running pipelines by a third party allows a fair comparison among systems since they are limited to expressing their solution using the same components.

The first version of the reference runtime we created was able to run most of the pipelines; the only limitation was that the pipeline does not contain any sub-pipelines. Also, we added some optimizations related to the order of the primitives' execution that allows running pipelines faster than the current implementation. Unfortunately, this optimization was removed by the main contributor since it made things difficult for new users to grasp the core package. Later on, through the runtime evolution, we made numerous bug fixes and added new features, such as exposing intermediate values on the pipelines so the users could debug or inspect them easier. Also, we contributed to the implementation of the pipeline runs into the runtime, which allows capturing information about how the pipeline ran, such as time for every step, metadata used, etc. We believed that the reference runtime had added value to the program since there was no similar framework in the AutoML community when it was created.

3.1.3 OpenML and D3M. During multiple meetings in the community outreach working group, it was brought to the attention that D3M wanted more exposure to the AutoML community. For this reason, we wanted to have some interoperability with the OpenML community. First, we added support for OpenML dataset in the D3M core package, but later on, we identified enough since the D3M environment requires a problem description. We worked with JPL to add full compatibility among these environments; we created the infrastructure necessary to map the OpenML task to the D3M problem description. As well we created a dataset crawler to help to prepare most of the OpenML datasets and problems. While doing it, we realized a lot of challenges that have not yet been fixed due to the community's lag of interest. One of the main problems is that OpenML datasets are split mainly by hand and have a fixed split, compared with D3M that uses pipeline language to represent them. Making a pipeline to split them is challenging since it requires many changes in how we run the splitting pipeline. Another important challenge is that OpenML tasks (D3M problems) use many more metrics than D3M, and some of them are irreproducible since they are custom-made.

For this reason, we added a fixed data split that allows easy compatibility. Also, we identified several challenges related to other factors, such as the D3M converter only worked on classification datasets because there are some challenges with the over-engineering of the D3M environment. Thanks to this work, Data Machines was able to run D3M AutoML systems with classification and regression problems and create a leaderboard, which is not a fair comparison to the OpenML leaderboard but is still moving in a good direction.

3.1.4 TA3-2 API. We actively engage in the discussion of the development of the TA3-2 API, which standardizes the communication between AutoML systems and graphical user interfaces. Such API was developed with gRemote Procedure Calls (grpc) since it can be used across multiple programming languages. We contributed by adding several calls on the protocol, which enable AutoML systems to save and load already trained models. Also, we worked with SRI to provide code utilities, so other AutoML systems do not have to implement most of the functionality from scratch.

3.1.5 Primitives. During the adoption of the TA1-2 API, we realized that there were no good examples of primitives so that other performers could use them as examples, and the machine learning models used came from few libraries. So, we decided to wrap several state-of-the-art machine learning primitives from the XGBoost [9] library and LightGBM [10] libraries. In total, we wrapped four primitives: LGBM classifier, XGBoost DART, XGboost GBTree, and XGBoost regressor. These primitives can be found in the common primitives' repository [3].

As TA2 performers, we noticed the lack of primitives related to ensemble models, so we decided to step in and create some primitives that could be helpful for multiple TA2s. Some of them are related are feature transformation, and others are feature selection. The primitives that we developed/wrapped are listed below and can be found our primitives repository [11] that later on were moved to the common primitives' repository for easy maintenance:

- Skfeature. A primitive that wraps a collection of well know feature selection algorithms from scikit-feature selection [12]
- DataFrameSampling. A primitive that is capable of doing sampling on dataframes and returns a list of dataframes generated by sampling the original one. This primitive is capable of sampling with different distributions (Uniform, Binomial, Geometric, Poisson, Normal).
- DataframeListExtractor. This primitive is meant to be used for extracting a single dataframe from a list of dataframes. The use case for this primitive is to generate a list with dataframes with different samples using DataFrameSampling and extract each of the elements to feed different machine learning models and later do bagging.
- HorizontalConcatList. A generalization of the primitive HorizontalConcat in common primitives. This version is capable of concatenate multiple dataframes at the same time, instead of only two as the standard version.

We also worked with JPL towards creating primitives that allow pipelines to output confidence scores, which were lacking for most of the program. To achieve this, we create two primitives. ComputeUniqueValues, as the name indicates, creates a list of unique values for categorical columns that are later use by the primitive ConstructConfidence, which uses a learner primitive to get the confidence score from all the labels.

3.1.6 Utils for evaluating TA2s. We created an automated TA3 that allows evaluating TA2 systems. This TA3 (Dummy TA3 [4]) uses the TA3-2 API that triggers the search of the TA2 systems to generate solutions. After this, the TA2's are asked to score the made solutions based on the problem description provided for the search. Then, dummy TA2 uses normalized these scores, and average them, and provides ranks for these solutions. Such a tool reduced the time spent by DataMachines for running TA2s systems since dummy-ta3 is responsible for waiting for TA2s as long as the length of the search, and it just gives the metrics and pipelines back to them without any extra work required.

3.1.7 Primitive testing.

```
################################################################################
#             d3m.primitives.classification.random_forest.Common               #
################################################################################
Init Done 0.018283605575561523
Basic Test Done 7.287029981613159
Scalability Test Done 48.90528225898743
Hyperparameter Importance Test Done 715.0794792175293
{
    "correct_output_metadata": true,
    "deterministic": true,
    "hyperparameter_importance": {
        "bootstrap": 0.0,
        "criterion": 0.0,
        "n_estimators": 0.414764350766392,
        "n_more_estimators": 0.0
    },
    "scalability": {
        "complexity": 2.210933054336364,
        "linearity": 1.4613539509335365,
        "memory_usage": 0.0007147019114939113
    },
    "semantic_types": {
        "has_use_semantic_types": false,
        "problems": [
            {
                "error": "fit_error No outputs columns.",
                "example": {
                    "categorial_targets": true,
                    "use_index": true,
                    "use_semantic_types": false
                }
            }
        ]
    }
}
Total_test_time:  771.2991998195648
```

Figure 1. Example of Statistical Analysis of Random Forest Primitive

Since the development of the D3M core package, we noticed that some performers were having trouble wrapping their primitives because of their inexpertise of Python or because their primitives were written with other languages such as C++ or Julia. Also, the overengineering aspect of the D3M code leads to some primitives not behaving as expected and not following the guidelines, such as having internal operators work with specific parts of the data. We decided to make automated testing tools [13] that allow testing certain aspects of all classification and regression primitives, an example of the analysis for the random forest primitive can be seen in Figure 1. Such aspects include the correctness of handling the metadata (setting the correct semantic types), checking how much a primitive is deterministic (it can replicate the result with the same conditions), and checking if the primitive has implemented the basic data operators.

The automated testing tool also measures other aspects of the primitives, such as hyperparameter importance and scalability. The hyperparameter importance measure the effect of the hyperparameters on the performance score (accuracy and root mean squared error); to achieve that, we create synthetic data and measure the change in performance of the algorithms by perturbing the hyperparameter setting. We measure the scalability in three different manners: Linearity, memory usage, and complexity. Linearity indicates how ell does a primitive scale with different data sizes; memory usage is the linearity of the input size to memory usage; the complexity is defined as the linearity of the number of features, the number of samples, and the execution time.

we managed to identify primitives that have issues with the hyperparameters "use_semantic_type" and discovered that the default value is not standardized, causing some trouble on the TA2 side. Also, we identified that primitives based on C/C++, such as "fastLVM" can break a TA2 system if the primitive crashes due to its nature. This tool allowed us to have a better understanding of the behavior of the primitives

3.2. AutoML on D3M

This section will be presenting the different Automl systems that we build in the different phases of the project to cover different requirements, as well as changes on the direction of how we perceived the challenges on AutoML. We describe the four general phases of our system in prototype, TAMUTA2, integration with AutoKeras and finally Axolotl.

3.2.1 Prototype. During the first year of the program, there were no strict requirements about elements that we should be using to create our AutoML system. The few requirements were related to solving only classification and regression problems. Other important requirements were related to creating standalone systems and creating pipelines that could train and later execute. For the first iteration, we created prototype [14], which aimed to solve classification and regression problems by using a handpicked selection of manually and locally wrapped primitives.

To have an AutoML system, we manually wrapped some handpicked primitives from the Scikit-learn library, which includes preprocessing primitives such as OneHotEncoder, label encoder, scalers, etc., as well as learners for classification and regression tasks.

The approach we used consisted of two phases: meta-learning and model selection. We use the same pipeline structure for both stages, which is defined with some preprocessing steps determined by the data and the learner who is specifically for classification and regression tasks.

For the meta-learning phase, we extract meta-features from datasets and then train a meta-model that recommends a learner who performs well on a task. The extraction of meta-features hinges on the information statistics from the dataset. The extracted features include: for the variables: whether or not the target variables are numerical values, number of possible values for the target, mean, median, variance, minimum and maximum (if the objective target is numeric); for the datasets: number of features, number of observations, number of continuous variables, number of missing elements. Besides, it also calculates PCA for explaining the variance. Then, we train the meta-feature classification model for primitive selection with datasets from the UCI machine learning repository; each training sample is a vector consisting of meta-features extracted from a dataset. We then generate the label for each training example, which is the algorithm's name, which achieves the best performance on the corresponding dataset. We divide the dataset into training and testing sets. Each algorithm in the pool runs on the dataset to measure its performance. Each training dataset is run on all the primitives we selected in that type of task and evaluated on the testing dataset. For evaluation, different metrics are used for various tasks; for classification tasks, we adopt accuracy as the metric, and for regression, we adopt the mean square error (MSE).

For the learner selection stage, the system will extract meta-features from the dataset. It will then load the meta-model trained from the disk and run the meta-feature classification on the extracted meta-features to select the primitive. Finally, the system applies the preprocessing primitive to process the dataset into numerical values. Eventually, the system trains the selected primitive on the dataset and outputs an executable file for testing.

Furthermore, the system also generates pipelines of ensemble models for regression and classification problems. The pipelines are generated according to the configuration that specifies which preprocessors to use, the size of the primitives' pool, and the number of primitives to use in the ensemble model. After the preprocessors are selected and trained, the system picks the top models suggested by the Meta-Model to form a primitive pool; then, it uses the top k primitives in the collection cross-validation results using the training data. Finally, the pipeline is produced using an ensemble learning model of the selected primitives with the preprocessors. Each selected primitive is combined with two strategies, either majority vote or summation of probability.

The AutoML prototype system helped us learn the required tools to make a robust system and enough programming experience and give us a clear direction in proceeding with the requirements. The system used to work, and we do not expect it to work still since many engineering things have changed through time, and we built it to be a prototype.

3.2.2 TAMUTA. After the first year, we decided to build another AutoML system due to multiple reasons, such as the maturity of the TA1-2 API (Core package), the need to do interoperability with TA3 systems and finally since we wanted to take advantage of the parallel computing resources on evaluation clusters. The system TAMUTA [15] (Figure 2) is divided into three main components, the execution manager, the search workers and the runtime workers.

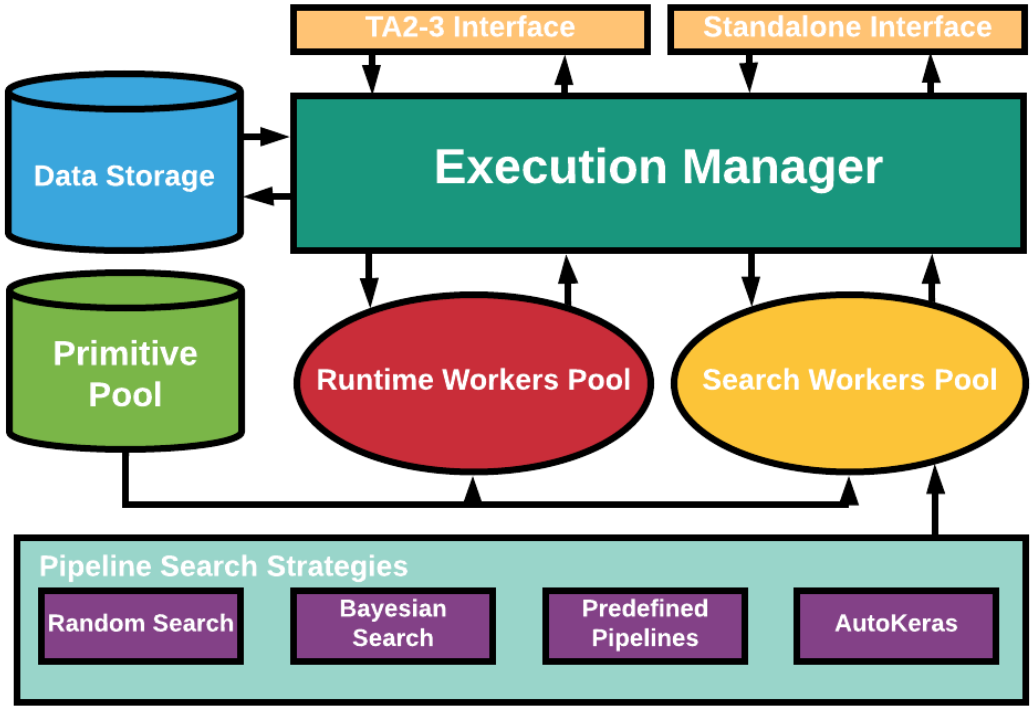

Figure 2. AutoML System TAMUTA Architecture

To generate valuable pipelines capable of solving multiple datasets, we created numerous search strategies that are run simultaneously for every search. Among the search strategies we develop, we have predefined pipelines, random search, and Bayesian optimization. The predefined pipeline strategy is a collection of multiple pipelines that we hand-picked due to numerous characteristics, such as performing well on multiple datasets or being the only pipeline capable of solving specific problems (like graph or image). These pipelines were the result of various leaderboards as well as example pipelines provided by performers.

For random search and Bayesian search, they are very similar in structure the pipelines: preprocessing and then learner. The preprocessing step is handled by only considering the problem description where it is specified the nature of the problem, such as image, audio classification, etc. After determining the data type of the problem, the system proceeds to identify the pipeline steps needed based on example pipelines that are problem specialized; then, the system will proceed to make a model selection, in

which the system maps every problem to either classification on regression depending on the nature of the problem, then it tries every task-specific model to find valid pipelines. For the final step, after generating and validating all possible combinations of preprocessing and learners, the system uses the top-k best-performing pipelines and does hyperparameter tuning. We created a wrapper for the SMAC framework, but later we noticed that it required a lot of engineering effort to keep it up to date and represented other challenges related to the compatibility of the hyperparameter space defined on D3M. Later on, we replaced our hyperparameter tuning strategy that uses the sampling methods implemented on the hyperparameters base classes on the core package. This hyperparameter tuner was designed following the interfaces presented by the Keras team for the Keras tuner and provides another way to add search algorithms in our system with interfaces that more people are familiar with.

Another important feature we integrated with the system is the use of dynamic caching of datasets using Plasma Arrow, as was recommended by the Ta3-2 working group. This mechanism allows the system to improve the number of pipelines that run in a specific time by avoiding loading the data repeatedly. Another benefit is that it improves the loading time and retrieving of results to TA3s that support it. Unfortunately, none of the other performers implement it, but the system still takes advantage of it since it runs pipelines in a parallel fashion across multiple nodes.

AutoML system TAMUTA provides the tools for experienced D3M users to create and test their search algorithms by reducing the overhead of creating an AutoML system from scratch. Also, it provides the interfaces to make their search algorithms compatible with the other components of the D3M ecosystems, such as the TA3-2 API interface and other methods that help them connect primitives on their pipelines without too much knowledge about the insight of the primitives.

3.2.3 Axolotl. Finally, we created a new open-source AutoML system Axolotl [16]. This new system aims to reduce the overhead of installing and using AutoML systems for new users and provides flexibility for more experienced users to develop and improve their tools.

The system was designed around five main concepts:

- Installability. New users can easily install the system and start using it right away; for this, we provide two installation methods, pip install and the use of the pre-build docker image.
- Easy to use. The system works out of the box and provides a set of tools to start looking for ML solutions. The design allows the user to export their data in Dataframe objects and interact with the system. Since most data scientists use Jupyter notebook, we decided to integrate it with it, provide some basic visualization tools, and provide documentation and examples on how to use it.
- Flexibility. We offer a different level of abstraction inside the system; this allows users to modify and improve various aspects of the system according to their requirements.
- Use of D3M. The system uses d3m infrastructure such as the pipeline language, primitives, problems, and dataset definitions.

- Easy deployment. After the data scientist is ready to deploy their search method, the system provides the tool to the user to export their algorithm into a server that uses the ta3-2 API.
-

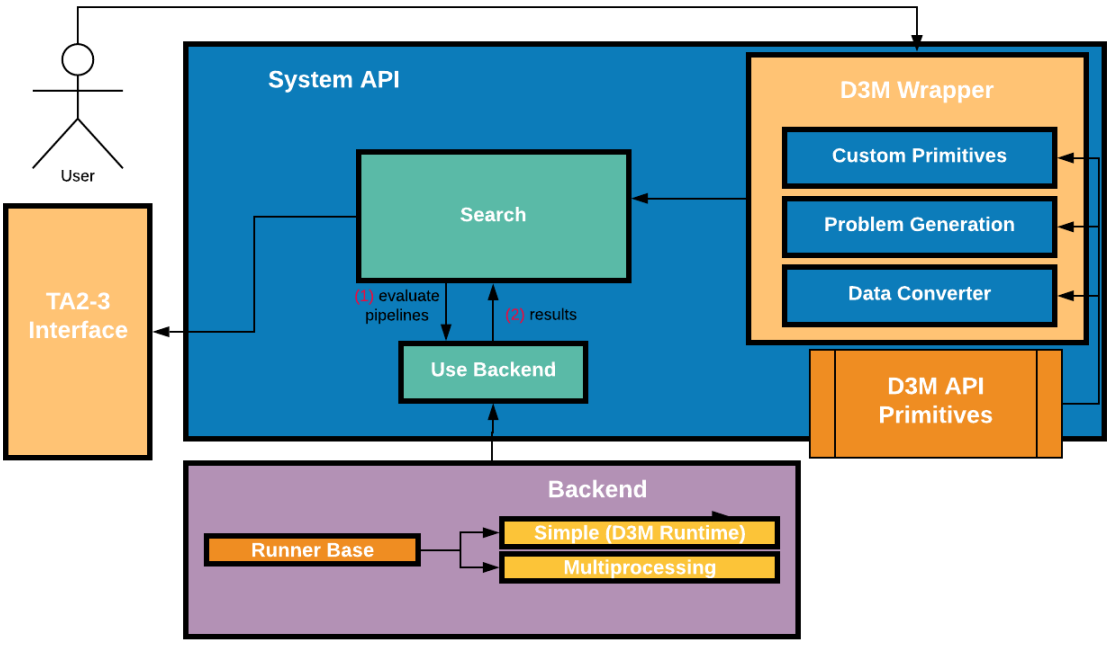

Figure 3. AutoML System Axolotl Architecture

The system (Figure 3) is divided into multiple components; the two main components are the search algorithms and the backend. The search algorithm is designed for the user to implement their different search methods and interact with all the other parts of the system by minimum effort. Search algorithms, in general, try to ensemble a pipeline and then proceed to evaluate their performance with a metric to use these results later and improve their output through iterations; this can be seen as evaluating more pipelines would increase the performance. For this reason, we decided to abstract the backend, which is in charge of executing and evaluating pipelines generated by the user or by search algorithms. We provide two different backends, a simple one that uses the reference runtime and runs one pipeline at the time, and a backend implementation that uses Ray [17] to run multiple pipelines in parallel. Base classes for both components can be found in the system; this allows users of various levels of knowledge to improve the system in different ways, such as creating better search algorithms tied to their specific needs or improving the pipeline executions.

The system API component aims to capture all the interactions with D3M components by creating simplified interfaces that allow users to call primitives or interact with TA3s without learning the insights about D3M. The D3M wrapper provides tools to transform tabular data to D3M datasets as well as helps the user create their problem specifications with simple information such as defining the target class and the metric that they want the system to use to evaluate the pipelines so far, this only works for

classification and regression problems. The wrapper also helps the user to create pipelines in a simplified manner by providing options to do certain operations with primitives that are not adequately implemented, such as allowing the user to operate into a specific column of the data or applying transformations to certain semantic types; This is achieved by automatically analyzing the primitive and check whether or not it has these operations implemented, otherwise, the system will create a sub-pipeline that consists on a set of primitives that are executed to achieve the operation. The system API wraps theD3M API Primitives component, which consists of the code required to ensemble pipelines and the primitives that are contained in the library.

As we previously mentioned, the backend is in charge of running different queries related to D3M pipelines search, such as running pipelines, exposing their intermediate steps, and evaluating them. It's one of the most important components since the heavy computational part relies on it. We know that there are different ways to implementing it, for example, a pipeline that caches all the results to avoid computing again later with new pipelines or even one that allows lazy evaluation of pipelines; for this reason, we decided to abstract it as an independent component of the system and created a base class that encapsulates the minimum requirements for a backend to be compatible with the other parts of the system. Such base class includes things defined on the reference runtime, such as training, producing and evaluating pipelines, or exposing intermediate results. An important abstraction detail is that every call to the backend should be implemented in two different manners, one that does the operations on single pipelines and the other to operate on many pipelines; the rationale behind this is that in many cases running multiple pipelines at the same time some optimization could be applied. The system provides two implementations of the backend. The first one is a wrapper around the reference runtime in which most of the functionality is exposed, and the second one is an extension of the first one, but it uses Ray to run pipelines in parallel, which allows scaling pipeline search algorithms running in clusters or small servers.

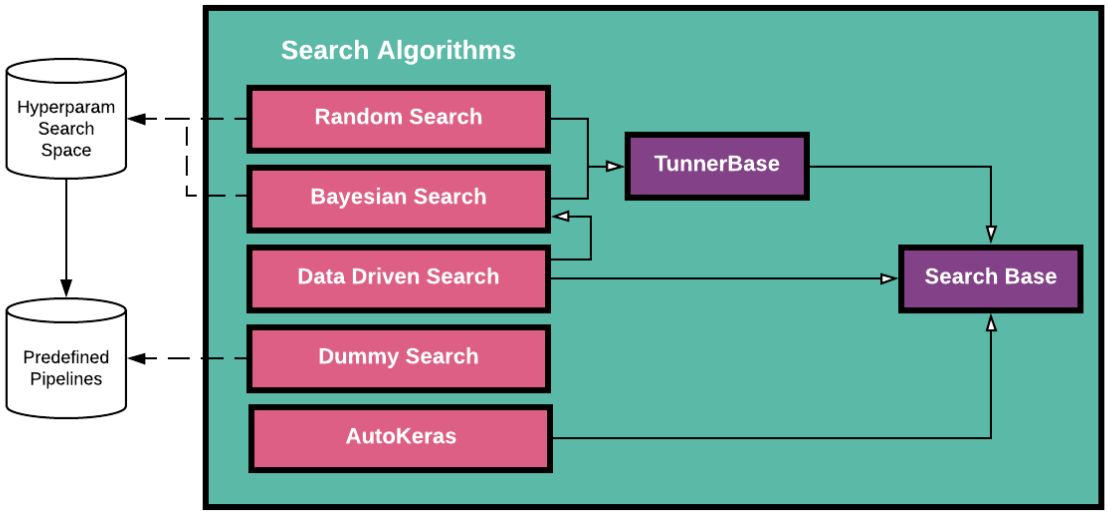

Figure 4. Relation of Different Component for the Creation of Search Algorithms

The search component is meant to interact with the D3M wrapper and the backend. The search is divided (Figure 4) into multiple searches that are derived from base classes: search base class and tuner base. Every search algorithm that aims to use other parts of the system needs to cover at least all the requirements specified on the search base class. The base class has implemented methods that allow the user to navigate through the history of the search as well as fit and produce the pipelines that were produced during the search with new data. When implementing new search methods, the user only needs to implement a single function to incorporate the search into the other parts of the system. Overall, the search flow goes as follows: the search generates pipeline candidates, then the candidates are evaluated via the backend, and finally, the candidates are ranked; this process will continue until there is no more time left. At the end of the search, the system will return the best-ranked pipeline as a result. The core of search algorithms is TunnerBase, the parent class of Random and Bayesian search, constructing the Hyperparameter Search Space from primitives and pipelines. The two derived search classes just sample candidate configurations from the space by their own search algorithms. TunnerBase will build pipelines to evaluate their performance based on candidates. The evaluation results will be fed back into the search algorithms (e.g., Bayesian optimization) to refine the underlying surrogate model.

We implemented multiple search algorithms into the system. The first one, called dummy search, uses a set of pipelines manually picked from the metalearning database based on their performance; this search is limited by the pipelines that have been digested and similar to Predefined search from the TAMUTA system. AutoKeras search will be cover in Section 3.3.1. Random and Bayesian search are classes created to tune hyperparameters from any given number of pipelines using different strategies (random or Bayesian optimization); this is achieved by using Keras tuners implemented with these characteristics.

Finally, we implemented the Data-Driven search algorithm that can be easily modified and extend by using our API. The data-driven search's general idea is to attempt to transform the input data into a common numerical representation (dataframe) and then do model selection, hyperparameter tuning, and pipeline selection. The search strategy is a combination of multiple strategies: heuristics, brute force, and Bayesian optimization. A visual representation of the search can be found in Figure 5. The search strategy is divided into four steps: Data Prepossessing, Feature Selection, Model Selection and Hyperparameter Tuning.

Data preprocessing. D3M datasets could have multiple resources (tables) that need to be transformed into a common representation, do the boilerplate is in charge of applying a set of predefined primitives steps into the pipelines that help to transform the D3M datasets into a single table. After generating a single table data representation, the system does data featurization. For this step, the system relies on the information about the data types and applies some manually defined transformations to specific columns.

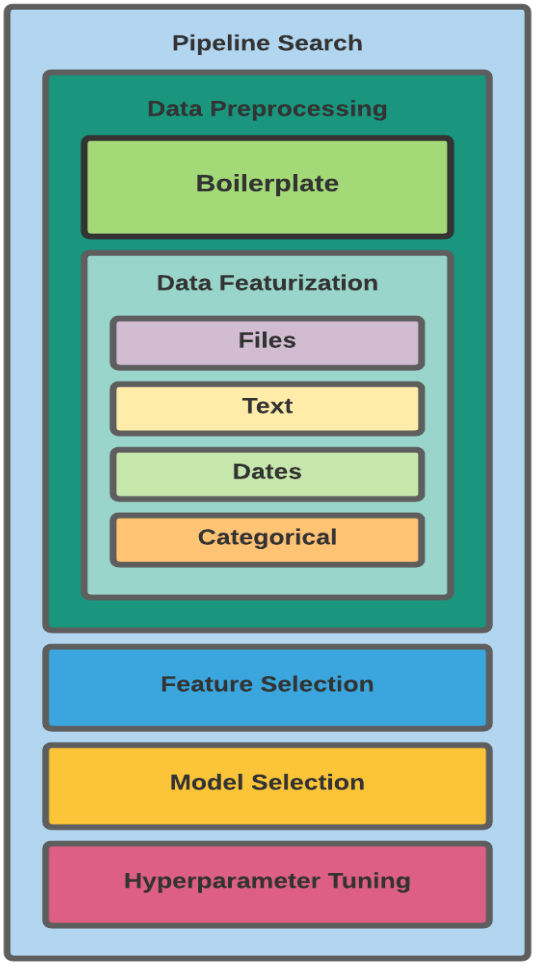

Figure 5. Axolotl Search Strategy

The primitives that we decided to use for the different cases of files are determined based on two factors, that the primitives can be consistently being used for the same data multiple times (since we notice that some primitives only work for a particular case) and the primitives have an example pipeline that is being shown multiple times. The main problem for algorithms to search preprocessing pipelines is that most of the primitives for such tasks can only be applied in a particular sequence and cannot be combined. For example, having heterogeneous data, where the table contains columns referencing CSV files and image files, most of the pipeline will not work since they do not know how to differentiate between the columns, and trying to apply them in a sequence will fail. For this reason, we made general operators that allow primitives to be executed in a single context and handling all the pipeline modification for the use; for example, a user could specify to which columns or columns with some semantic type to apply a certain sequence of steps without affecting the remaining of the data. This allows us to encapsulate

operations in a more friendly way and handling the misbehavior of primitives. Besides applying specific operations to file columns, the system can also apply specific operations for other column types like categorical, text, dates, etc. Each of the primitives applied uses some heuristic based on results observed on previous dry-runs.

Feature Selection. After transforming the input dataset to a normal numerical table, the system attempts to find a suitable feature selection primitive. For this task, we try every primitive that has information about whether or not it is a feature selection algorithm. \textbf{Model Selection.} Due to the limited number of specialized primitives for different problems, such as graph problem, we decided to treat every possible problem as a classification or regression; given this, the system queries the primitives that fits the task and then added them to the pipelines. The number of primitives that have been created for specific task type and data featurization is small for a given task; for this reason and because the time constraints for evaluation is one hour and most of the datasets are fairly small, we generate pipeline by doing the dot product of the preprocessing pipeline, all feature selection primitives, and all learners. During the pipeline evaluation in our system, we prioritize the most simple pipelines, which means the ones that do not contain feature selection primitives. We only contain certain models, such as random forest or XGboost.

Hyperparameter Tuning. Finally, after generating all of the possible pipelines based on our heuristics, the system tries to score them and assign them a rank. The system proceeds to take the top-k best-performing pipelines and do hyperparameter tuning using Bayesian Optimization. We decided to use the Keras tuner since the interfaces are aligned with ours and easy to update and understand. The hyperparameter tuning continues until there is no more time left for the search. In some cases, the system will never reach the hyperparameter tuning step due to dataset size and time constraints. For this reason, the system tries to run all the pipelines with default hyperparameters since they provide a good overall performance estimation. The design of our latest search strategy combines multiple algorithm types that allow extendability by providing extendable methods and simplifying operations such as applying a transformation to specific columns and flexibility and robustness for unseen problem types.

3.3. AutoKeras

Auto-Keras [18] proposed a novel framework enabling Bayesian optimization to guide the network morphism for efficient neural architecture search. The framework develops a neural network kernel and a tree-structured acquisition function optimization algorithm to efficiently explore the highly flexible search space.

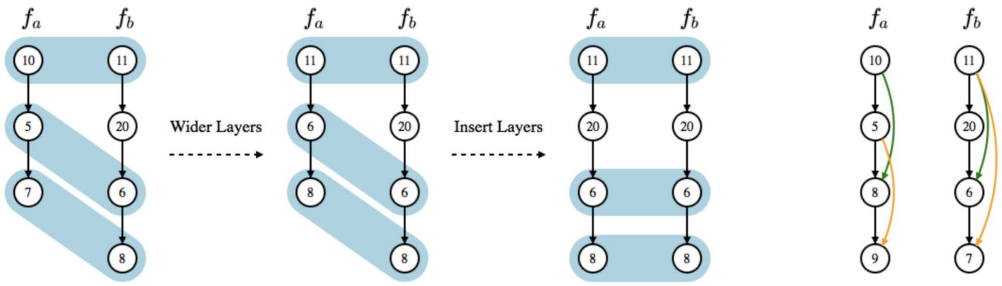

Figure 6. Edit-distance illustration between two networks

Specifically, we proposed an edit-distance kernel for neural networks. Edit-distance (Figure 6) means how many operations are needed to morph one neural network to another. The $f_a$and $f_b$are example architectures showing the number of steps we change kernel size, layer size and location of skip connection from one to the another. The concrete kernel function is defined as:$exp^{-\rho^2(edit\ distance)}$. The kernel function could define the distance of two architectures in Euclidean space, which is required to utilize Gaussian process (GP).

Second, we modified the upper-confidence bound (UCB) to a tree-structured acquisition function to generate the next architecture. In a tree-structured search space, each node represents a neural architecture and each edge is a morph operation. Inspired by various heuristic search algorithms, the new acquisition function combines A* search and simulated annealing. A* algorithm is widely used for tree-structure search. It maintains a priority queue of nodes and keeps expanding the best node in the queue. Since A* always exploits the best node, simulated annealing is introduced to balance the exploration and exploitation by not selecting the estimated best architecture with a probability.

Third, a network morphism operation on one layer may change the shapes of some intermediate output tensors, which no longer match input shape requirements of the layers taking them as input. We defined graph-level network morphism in the neural architectures based on layer-level network morphism. The effective area is a set of nodes in the computational graph, which can be recursively defined by the following rules: first, operation wide(G,u0) needs to change two set of layers; the previous layer set needs to output a wider tensor; and next layer set needs to input a wider tensor. Second, for operator add(G,u0,v0), additional pooling layers may be needed on the skip-connection. u0 and v0 have the same number of channels, but their shape may differ because of the pooling layers between them. Third, in concat(G,u0,v0), the concatenated tensor is wider than the original tensor v0. The concatenated tensor is input to a new layer Lc to reduce the width back to the same width as v0. Additional pooling layers are also needed for the concatenative connection.

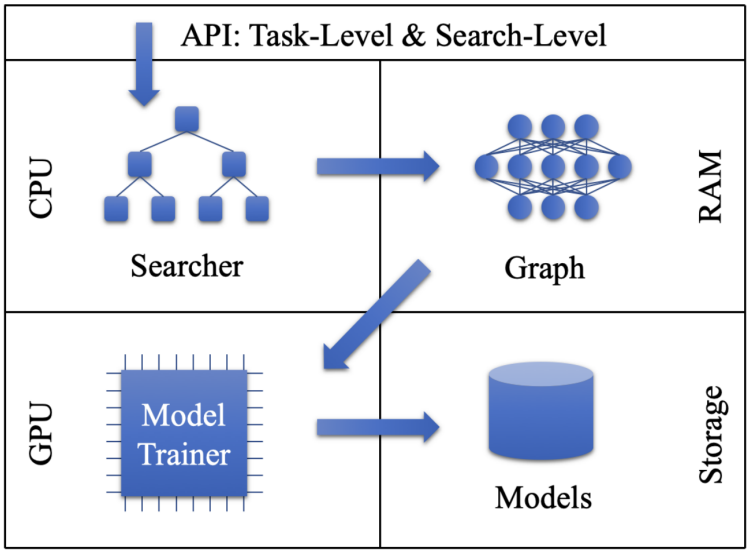

Figure 7. Auto-Keras System Overview

Based on the proposed neural architecture search method, we developed an open-source AutoML system, namely Auto-Keras. It is named after Keras. Figure 7 shows the Auto-Keras System Overview. (1) The user calls the API. (2) The Searcher generates neural architectures on CPU. (3) Graph builds real neural networks with parameters on RAM from the neural architectures. (4) The neural network is copied to GPU for training. (5) Trained neural networks are saved on storage devices. The Searcher is updated based on the training results. Step (2) to (5) will repeat until it reaches the time limit.

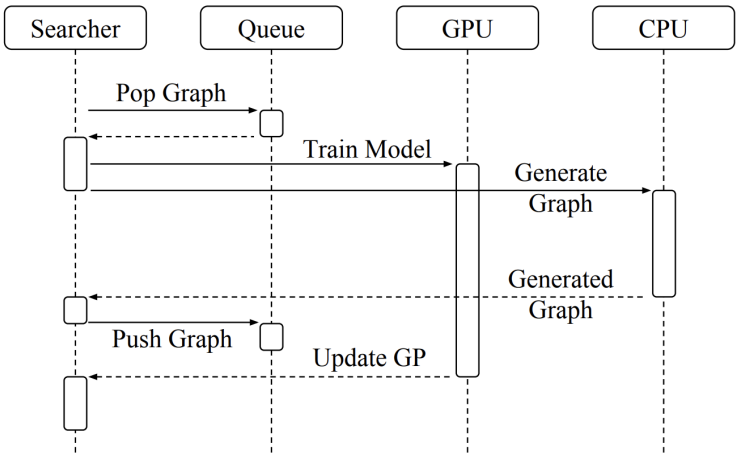

Figure 8. Sequence Diagram of CPU and GPU Parallelism in AutoKeras

To make full use of the limited local computation resources, the program can run in parallel on the GPU and the CPU at the same time. Figure 8 shows the Sequence diagram

of the parallelism between the CPU and the GPU. First, the Searcher requests the queue to pop out a new graph and pass it to GPU to start training. Second, while the GPU is busy, the searcher requests the CPU to generate a new graph. At this time period, the GPU and the CPU work in parallel. Third, the CPU returns the generated graph to the searcher, who pushes the graph into the queue. Finally, the Model Trainer finished training the graph on the GPU and returns it to the Searcher to update the Gaussian process. In this way, the idle time of GPU and CPU are dramatically reduced to improve the efficiency of the search process.

Table 1. Classification Error Rate on the three image datasets with AutoKeras

Methods	MNIST	CIFAR10	FASHION
RANDOM	1.79%	16.86%	11.36%
GRID	1.68%	17.17%	10.28%
SPMT	1.36%	14.68%	9.62%
SMAC	1.43%	15.04%	10.87%
SEAS	1.07%	12.43%	8.05%
NASBOT	NA	12.30%	NA
BFS	1.56%	13.84%	9.13%
BO	1.83%	12.90%	7.99%
AK	**0.55%**	**11.44%**	**7.42%**

We evaluated the effectiveness of the proposed method. The results are shown in Table 1. The proposed method AutoKeras (AK) achieves the lowest error rate on all the three datasets, which demonstrates that AK is able to find simple but effective architectures on small datasets (MNIST) and can explore more complicated structures on larger datasets (CIFAR10). Comparatively speaking, NASBOT also uses Bayesian optimization as our proposed method, but the low efficiency in training the neural architectures constrains its power in achieving comparable performance within a short time period. For the two variants of AK, BFS preferentially considers searching a vast number of neighbors surrounding the initial architecture, which constrains its power in reaching the better architectures away from the initialization. By comparison, BO can jump far from the initial architecture. But without network morphism, it needs to train each neural architecture in a much longer time.

3.3.1 AutoKeras in D3M. Due to the success of AutoKeras, we decided to integrate with the second iteration of the AutoML system that we previously described. For this, there were several challenges that we needed to solve, such as create a language to express Neural Networks within the D3M framework. For this, we actively participated in the discussion to develop such language. Unfortunately, our participation in the meetings did not have enough leverage compared to other performers. It resulted in an incomplete and overly complicated language to express multiple components of the architecture of the Neural Networks in the pipeline language. After JPL implemented such components, we created a wrapper [19] that maps the Neural Architectures discovered by Autokeras to the pipeline language.

There were several challenges that we took into consideration to design the compiler. The first one is that is related to that running neural network pipelines on D3M is very slow and does not provide any optimization to determine how good a pipeline is doing; this is a direct consequence of the design of the language. Second, it was almost impossible to run a neural network pipeline locally or with Data Machine cluster because the input dataset would never fit into memory even by providing vast amounts. Finally, due to the implementation of the neural network pipeline language, it was impossible to create an optimization that allows application transfer learning to shorten the training time of the neural networks. We decided to create a wrapper that compiles results from Autokeras to D3M pipeline language based on these challenges. We decided to do the neural architecture search directly with AutoKeras instead of doing it on the D3M pipeline language. As we previously mentioned, Autokeras already has some optimization that allows evaluating more architectures at the same time than the D3M language. The speedup is achieved using transfer-learning to reuse previous results and avoid training a neural network from scratch every time.

The AutoKeras compiler works by transforming the D3M dataset using predefined preprocessing steps that transform it into a common Dataframe representation used directly into Autokeras. Then, the system provides forty percent of the total time specified on the search to look for the best architecture; the reason for the shorter search time is because most of the spent time is by trying to evaluate the neural network inside the D3M ecosystem. From our experience, the results obtained from the same neural network being evaluated into the two frameworks changes due to how the network is trained.

We faced the primary challenge by introducing AutoKeras on D3M because things did not fit in memory. Autokeras deal with this problem by using batching; unfortunately, this was not implemented on D3M due to multiple factors such as Runtime implementation. For this problem, we added to our pipelines the primitive that we had created, "DataframSampling," that allowed us to chop the training data to a subset. Thus, it translates into training the pipeline with a fraction of the data and without running out of memory. We did not consider that our system scores the pipeline via cross-fold validation, so the training data is split, and we can score it. During the evaluation, the system could find a suitable pipeline, but the Data Machines pipeline runner crashed by trying to run the pipeline due to a lack of memory. We find out that the test set was still too large for being handle by the KerasWrapper. Later on, we decided to create a new primitive that enables batching capabilities inside the D3M Runtime. Our primitive "d3m.primitives.data_wrangling.batching.TAMU" enables batching inside pipelines for the cases where there are data loaders such as audio, image, or video loader; and

primitives that have implemented the `continue_fit.` The primitive is aligned with the design principle of the use of the hyperparameter `use_semantic_types' which defines how to handle the data generated by the primitive itself. As we mentioned, the primitive uses another two primitives as hyperparameters, the learner and the reader, and another two hyperparameters (batch_size and sampling_method) that define the behavior of the batches. The batch size defines how many instances we are using per batch, and the sampling method defines how we are grouping instances for the batch, currently random, stratified, and in order are supported. By using this primitive, we enabled the creation of pipelines using the Keras wrapper to solve problems such as MNIST and CIFAR10

As for integrating the AutoKeras with the AutoML systems TAMUTA and Axolotl, we added a Search method that uses it and works the same way as other search methods. Later on, we identified that the wrapper could not work under certain conditions related to using the legacy code inside the Keras wrapper. TAMUTA system using AutoKeras wrapper could only work if only if all the other primitives do not use Keras; for this reason, there is a version of the system that exclusively uses it.

3.4. AutoML System TODS: Time-series Outlier Detection System.

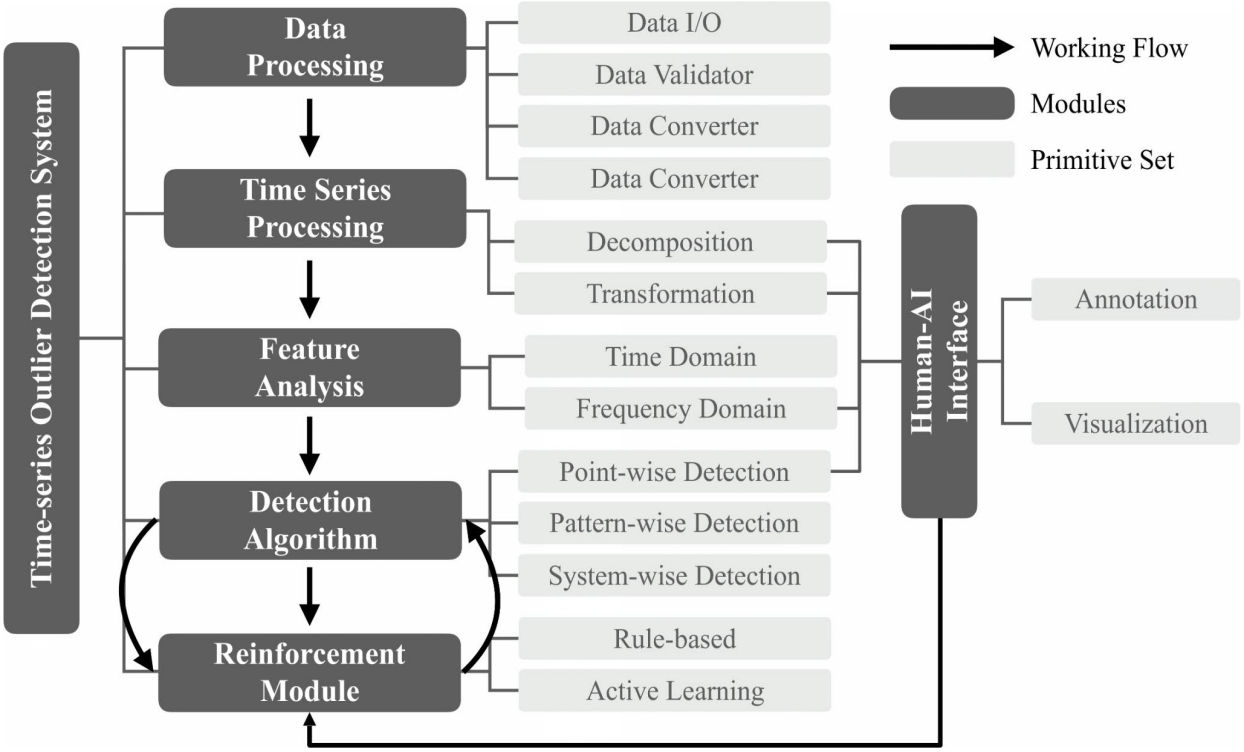

Figure 9. TODS System Structural Diagram

TODS [20] is an end-to-end **T**ime-series **O**utlier **D**etection **S**ystem. In TODS, there are six modules: Data preprocessor, time-series processor, feature analyzer, a detection

module, reinforcement module, and human-AI interface. Figure 9 illustrates the overall workflow and structural design of TODS. Specifically, each module consists of several primitive sets where each one is composed of various functions.

This package aims to provide an end-to-end machine learning system for outlier detection tasks on time series data, and the target audience of this package is general software engineers with limited machine learning/data mining expertise. We define three scenarios that can include all of the outlier detection scenarios in our daily life, such as point-wise detection, pattern-wise detection, and system-wise detection. The point-wise detection aims at detecting the anomalous time point within the data by defining the anomalies as time points. The pattern-wise detection aims to identify odd patterns in the data by specifying the anomalies on subsequences. The system-wise detection aims to find anomalous systems by defining a set of time series data as an anomaly. Next, we illustrate the functionality of each module that composes TODS.

3.4.1 Creation of specialized primitives.

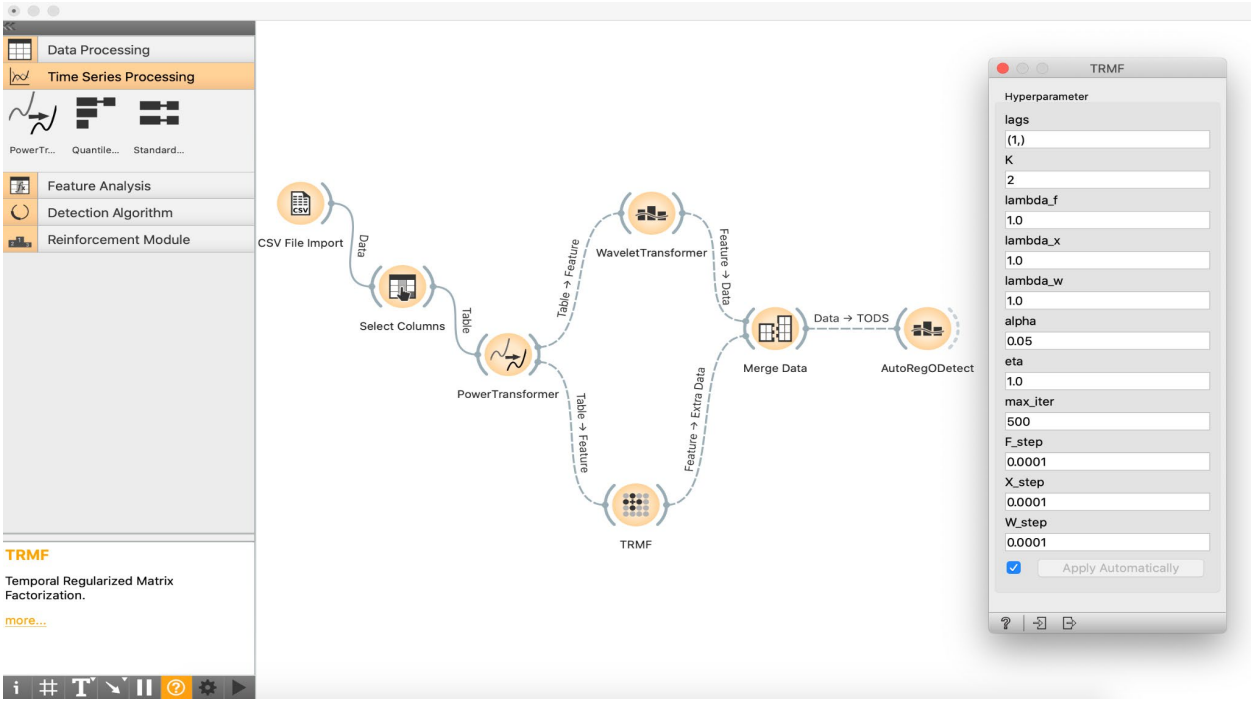

Figure 10. Snapshot of TODS-GUI

Our team designed the data processing module for preprocessing data in tabular manner, which includes 11 primitives such as loading data from various input sources, validating timestamps in the data, missing data imputation, and data conversion. We designed the time series processing module with 10 primitives to tackle the unique attributes in the time-series data, such as trend and seasonality, by various smoothing and transformation. The feature analysis module with 31 primitives aims on helping users to extract meaningful features from three aspects of time series data: temporal-domain (e.g., statistical features), frequency domain (e.g., spectral transformations), and latent feature

space (e.g., matrix factorization). The detection modules involve 24 popular and state-of-the-art machine learning algorithms for outlier detection. In this module, we involve classical point-wise approaches such as isolation-forest, local outlier factors and pattern-wise techniques such as matrix-profile and state-of-the-art deep learning models including LSTM autoencoder, generative adversarial neural networks as well as popular ensemble methods to address system-wise detection scenario.

Typically, in the early stage, the outliers' labels are usually inaccessible; therefore, domain expertise plays a critical role in the beginning. We designed the remaining two modules: human-AI interface and reinforcement module, to introduce human knowledge into the machine learning model. The human-AI interface aims to provide a graphic user interface for users to manipulate and construct the outlier detection pipeline in a drag-and-drop fashion (Figure 10), visualizing the result and annotating the labels. Each icon in Figure 10 indicates one primitive in the TODS. Users can easily construct a D3M pipeline description file with simple drag-and-drop interactions, and run the pipeline in one pipeline by a single click.

On the other hand, the reinforcement module aims to incorporate the domain expertise via active learning and a rule-based system. The active learning approaches aim to efficiently use annotated labels to improve the existing machine learning pipeline, where the rule-based method will allow users to construct their rule-based model to incorporate their domain knowledge with a machine learning pipeline with a set of predefined rules.

3.4.2 Searching Mechanism of TODS. TODS currently supports brute force search, other advanced searching algorithms as well as search space interface is under development. Specifically the search space is defined as a JSON file representing a multi-level dictionary. In the dictionary, the key of the first level is the module name of the TODS (such as data processing, time series processing, feature analysis or detection algorithms). In the first level, we define the general structure of the pipeline such as how many pre-processing, feature extraction, detection units do we want to be involved. Then, the content of the first level is the second-level, a list of dictionaries, where the key of the dictionaries are the function name, and the values are the third level dictionary for defining the candidate hyper-parameters for each of the primitives.

Figure 11 is an example description file defines a 4-step pipeline structure where each step has several candidate primitives. In each primitives, hyper-parameter names with the candidate values are defined as a dictionary, where key is the name of hyper-parameter and value is a list containing the search space of hyper-parameter. WIth the search space description file, TODS will perform hierarchical search. That is, TODS will first utilize the information in the first-level dictionary to search the combinations of primitives with default hyper-parameters and find a best candidate pipeline structure. Then, TODS will leverage the search space of hyper-parameters in the second-level dictionary for fine tuning the hyper-parameters.

```
{
      "data_processing": {
            "time_interval_transform":{
                  "time_interval": ["5T", "3T", "6T"]
            }
      },
      "timeseries_processing": {
            "seasonality_trend_decomposition":{
                  "period":[5, 10, 15],
                  "model": ["additive", "multiplicative"]
            },
            "moving_average_transform":{
                  "window_size": [30, 50, 70, 120]
            }
      },
      "feature_analysis": {
            "statistical_hmean":{
                  "window_size": [10, 20, 30, 40]
            },
            "fast_fourier_transform":{
                  "norm": ["None", "ortho"]
            },
            "hp_filter":{
                  "lamb": [1600, 2000, 3000, 5400]
            }
      },
      "detection_algorithms": [
            "isolation_forest":{
                  "num_tree": [10, 20, 30, 40],
                  "learning_rate": [0.01, 0.025, 0.3]
            },
            "ocsvm":{
                  "gamma": [0.3, 0.25, 0.1, 0.05],
                  "kernel": ["poly", "rbf", "sigmoid"]
            }
      ]
}
```

Figure 11. Example of Search Space Description File

3.4.3 Programming Interface of TODS. There are two kinds of interfaces to interact with TODS, scikit-learn interface (Figure 12) and Axolotl interface (Figure 13). Since TODS is an automated machine learning system built upon D3M, users are able to build and run the pipeline via D3M interface. In addition, to allow easier usage, TODS also adopts Axolotl (described in section 3.2) to develop pipeline running interface and AutoML searching interface (example inFigure 14). Moreover, to allow users to establish complex experiments and integrate with third party packages, TODS also provides a sklearn interface. In this way, users are able to access every individual primitive without building a pipeline in scikit-learn fashion.

```
import numpy as np
from tods.tods_skinterface.primitiveSKI.detection_algorithm.ABOD_skinterface import ABODSKI

X_train = np.array([[3., 4., 8., 16, 18, 13., 22., 36., 59., 128, 62, 67, 78, 100]])
X_test = np.array([[3., 4., 8.6, 13.4, 22.5, 17, 19.2, 36.1, 127, -23, 59.2]])

transformer = ABODSKI()
transformer.fit(X_train)
prediction_labels = transformer.predict(X_test)
prediction_score = transformer.predict_score(X_test)

print("Primitive: ", transformer.primitive)
print("Prediction Labels\n", prediction_labels)
print("Prediction Score\n", prediction_score)
```

Figure 12. Example of TODS Scikit-learn Interface

```
import sys
import argparse
import os
import pandas as pd

from tods import generate_dataset, load_pipeline, evaluate_pipeline

this_path = os.path.dirname(os.path.abspath(__file__))
default_data_path = os.path.join(this_path, '../../datasets/yahoo_network_intrusion.csv')

parser = argparse.ArgumentParser(description='Arguments for running predefined pipelin.')
parser.add_argument('--table_path', type=str, default=default_data_path,
                    help='Input the path of the input data table')
parser.add_argument('--target_index', type=int, default=6,
                    help='Index of the ground truth (for evaluation)')
parser.add_argument('--metric',type=str, default='F1_MACRO',
                    help='Evaluation Metric (F1, F1_MACRO)')
parser.add_argument('--pipeline_path',
                    default=os.path.join(this_path, './example_pipelines/autoencoder_pipeline.json'),
                    help='Input the path of the pre-built pipeline description')

args = parser.parse_args()

table_path = args.table_path
target_index = args.target_index # what column is the target
pipeline_path = args.pipeline_path
metric = args.metric # F1 on both label 0 and 1

# Read data and generate dataset
df = pd.read_csv(table_path)
dataset = generate_dataset(df, target_index)

# Load the default pipeline
pipeline = load_pipeline(pipeline_path)

# Run the pipeline
pipeline_result = evaluate_pipeline(dataset, pipeline, metric)
print(pipeline_result)
```

Figure 13. Example of TODS Pipeline Running with Axolotl

```
import pandas as pd

from axolotl.backend.simple import SimpleRunner

from tods import generate_dataset, generate_problem
from tods.searcher import BruteForceSearch

# dataset path
table_path = '../../datasets/yahoo_network_intrusion.csv'
target_index = 6 # what column is the target
time_limit = 30 # How many seconds you wanna search

metric = 'F1_MACRO' # F1 on both label 0 and 1

# Read data and generate dataset and problem
df = pd.read_csv(table_path)
dataset = generate_dataset(df, target_index=target_index)
problem_description = generate_problem(dataset, metric)

# Start backend
backend = SimpleRunner(random_seed=0)

# Start search algorithm
search = BruteForceSearch(problem_description=problem_description,
                          backend=backend)

# Find the best pipeline
best_runtime, best_pipeline_result = search.search_fit(input_data=[dataset], time_limit=time_limit)
best_pipeline = best_runtime.pipeline
best_output = best_pipeline_result.output

# Evaluate the best pipeline
best_scores = search.evaluate(best_pipeline).scores
```

Figure 14. Example code of using Axolotl interface to search optimal pipeline

3.5. Other work

3.5.1 Discriminative Graph Autoencoder. The paper [21] proposed a method for graph vectorization, namely discriminative graph autoencoder (DGA). Given a set of graphs, each of the graphs can be vectorized into a vector. Traditional methods usually require expensive computations on the graphs. The proposed method avoids any expensive computations by simply sampling some centroids in the graph, extract the neighborhood subgraphs from the centroids, and use the adjacency matrix to train an autoencoder neural network. The autoencoder learns the representation of a graph by encoding it to a vector and reconstructing it from its output. In the loss of the function of the autoencoder, in addition to the reconstructing error, there is also a term for the prediction error. If each of the graphs is associated with a label, it would try to predict the label using the encoded vector. It can learn a better vector representation for the downstream task with this term. A potential application of the DGA method is for neural architecture search (NAS). The NAS methods usually require a model to learn the relation between the neural architectures, which are graphs, and their performances. DGA can map the neural architectures to Euclidean space so that a wider range of methods can be applied to solve the NAS problem.

3.5.2 Coupled Variotional Recurrent Collaborative Filtering. Traditional work has been widely conducted in exploiting temporal dynamics to improve the recommendation performance under static settings or streaming settings. Though great advances have been made, the capacity of these models are usually restricted while capturing complex data structures and require delicate statistical assumptions comparing with structured deep frameworks. Despite the remarkable revolution achieved recently in deep recommender systems either in static setting or streaming, deep frameworks also have their own limitations. One of the most noticeable fact is that deep frameworks are usually deterministic approaches, which only output point estimations without taking the uncertainty into account. It significantly limits their power in modeling the randomness of the measurement noises~\cite{shi2017zhusuan} and providing predictions of the missing or unobserved interactions in recommender systems. As probabilistic approaches, especially Bayesian methods, provide solid mathematically tools for coping with the uncertainty, it motivates us to conduct streaming recommendations from the view of Deep Bayesian Learning (DBL) to conjoin the advantages of probabilistic models and deep learning models.

However, simply applying DBL on streaming recommendation is a non-trivial task due to the following challenges. First, coordinating the temporal dynamics is difficult given the continuous-time discrete-event recommendation process along with the protean patterns on both user and item modes. Second, the high velocity of streaming data requires an updatable model, which could expeditiously extract the prior knowledge from former time steps and effectively digest it for current predictions. Third, the DBL frameworks are usually expensive in terms of time and space complexities.

To tackle these aforementioned challenges, in this work [22], we propose to investigate how to conduct streaming recommendation by leveraging the advantages of both deep

models and probabilistic processes. Specifically, we study: (1) How to model the streaming recommender system with an updatable probabilistic process?
(2) How to incorporate deep architectures into the probabilistic framework?
(3) How to efficiently learn and update the joint framework with streaming Bayesian inference? Through answering the three questions, we propose a Coupled Variational Recurrent Collaborative Filtering (CVRCF) Framework. The core of CVRCF is a dynamic probabilistic factor-based model that consists of four components. The first two formulate the user-item interactions and temporal dynamics, respectively. Each of them incorporates a probabilistic skeleton induced by deep architectures. The third component is a sequential variational inference algorithm, which provides an efficient optimization scheme for streaming updates. The last component allows us to generate rating predictions based on the up-to-date model.

3.5.3 Deep Neural Network with Knowledge Instillation. In this paper [23], we propose a general knowledge instillation framework, named NeuKI, to instill human knowledge into feed-forward Deep Neural Network (DNN) for performance enhancement. In particular, besides the target-DNN for instillation, we build another separate knowledge-DNN. The knowledge-DNN is faithfully constructed to regularize the training process of target-DNN, so that instilled knowledge could affect the model prediction intentionally. Moreover, the proposed NeuKI framework is further demonstrated to be applicable to different forms of human knowledge, including both structured rules and declarative constraints. To be specific, rules can be encoded into the architecture of knowledge-DNN, while constraints can be injected through the loss function of knowledge-DNN. Furthermore, NeuKI does not have additional requirements on the type of target-DNN as long as it is feed-forward, which makes it convenient to incorporate knowledge into different types of DNN. The overall end-to-end training of NeuKI further facilitates the integration process of knowledge for neural networks. Experiments are conducted on several real-world datasets from different domains, and the results demonstrate the effectiveness of NeuKI in improving learning performance of target-DNN, as well as corresponding data efficiency and model interpretability.

3.5.4 On Robust of Neural Architecture Search under Label Noise. Neural architecture search (NAS) automatically design neural architectures given a specific task in supervised learning applications. In real-world situations, the class labels provided in the training data would be noisy due to many reasons, such as subjective judgments, inadequate information, and random human errors. We called the practical data corruption as label noise, polluting both training and validation labels. Existing work has demonstrated the adverse effects of label noise on the learning of weights of neural networks. For NAS, the problem is compounded because we need to search for architecture as well. Since different architectures are learned using training data and compared based on their validation performance, label noise in training and validation (hold-out) data may cause a wrong assessment of architecture during the search process. Thus label noise can result in undesirable architectures being preferred by the search algorithm, leading to the loss of performance.

In this paper [24], we systematically investigate the effect of label noise on NAS. We show that label noise in the training or validation data can lead to different degrees of

performance variation. To mitigate the effects of label noises on NAS, we replaced the categorical cross entropy (CCE) with robust log loss (RLL) function. RLL considers both correct and noisy labels into loss function and has been proved robust under symmetric label noise. We demonstrate through simulations that the use of a robust loss function (in place of CCE) in NAS can mitigate the effect of harsh label noise. We also provide a theoretical justification for this observed performance: for a class of loss functions that satisfies a robustness condition, we show that, under symmetric label noise, the relative risks of different classifiers are the same regardless of whether or not the data are corrupted with label noise. Through empirical experiments, using RLL can mitigate the performance degradation under symmetric label noise as well as under a simple model of class conditional label noise. Both empirical and theoretical results provide a strong argument in favor of employing the robust loss function in NAS under high-level noise.

The main contributions of the paper can be summarized as follows. We provide, for the first time, a systematic investigation of the effects of label noise on NAS. We provide the theoretical and empirical justification for using loss functions that satisfy a robustness criterion. We show that the use of robust loss functions is attractive because of the better performance under high-degree noise than that under the standard CCE loss.

3.5.5 Towards Automated Neural Interaction Discovering for Click-Through Rate Prediction. Predicting Click-Through Rate (CTR) is a crucial problem in many web applications such as real-time bidding, display advertising, and search engine optimization. Due to the large-scale dataset and high-cardinality feature property, extensive efforts have been devoted especially in deep learning to designing neural architectures for effectively learning combinatorial feature interactions towards condensed low-dimensional feature representations. Upon the prevalent adoption of deep learning techniques, Neural architecture search (NAS) has become a prevailing research field. It aims to discover optimal deep learning solutions automatically given a data-driven problem, thereby enabling practitioners to access the off-the-shelf deep learning techniques without extensive experience, and alleviating data scientists from the burden of manual network design. The rapid development of NAS research and systems has enabled the automation of state-of-the-art deep learning tools for various learning tasks in computer vision (CV) and natural language processing, which motivates us to explore its potential in the context of RSs in discovering complex neural interactions, specifically for CTR predictions.

Developing novel NAS approaches for neural interaction discovery and better CTR models is technically challenging. First, different from structure image data in CV tasks, CTR features are often heterogeneous, high-dimensional, and have both sparse and dense components, which are structureless and of diversified meanings in reality. Second, different from the dominant convolutional neural networks in CV tasks that consist of multiple structured convolutional operations, existing models for CTR prediction usually adopt multiple diverse and ad-hoc operations, leading to unstructured search space. Third, a practical model for CTR prediction is often trained on billions of data (e.g., Facebook has millions of daily active users and over 1 million active advertisers, yielding billions of instances), requiring the NAS process to be time and space efficient. Finally,

the performance of CTR models with different architectures are often quite close in practice, asking for the NAS approach to be sensitive and discriminating.

To cope with these challenges, we propose an automated neural interaction discovering framework for CTR prediction named AutoCTR [25]. It consists three main components following the classical design of NAS algorithm including search space, search techniques, and performance estimation strategy. Firstly, we design a two-level hierarchical search space by extracting and abstracting representative structures in existing human-crafted CTR prediction architectures into virtual blocks, and wire them together as a set of direct acyclic graphs (DAGs) with dimensionality alignment among features. The inner space is composed of the appendant hyperparameters of blocks, such as the number of units and layers in a multi-layer perceptron block, while the connections among blocks form the outer search space. Secondly, we propose a mixed searcher composed of an evolutionary algorithm and a learning-to-rank guider. We select the evolutionary algorithm in this pilot study due to its simplicity and effectiveness in balancing the exploitation and exploration. We adopt a multi-objective evolutionary searcher as the backbone and employ a tree-based learner to guide the mutation of a selected parent architecture in each iteration to facilitate the exploitation of superior offsprings. The search process could be described as a loop of three stages, i.e., parent selection, guided mutation, and survivor selection. The initial population is constructed via randomly selecting and evaluating a predefined number of architectures. Stratified selections could also be used to potentially enhance the performance, which we leave for future exploration. Thirdly, We consider two strategies of low-fidelity estimation and adopt a warm-start embedding trick leveraging the weight inheritance among architectures to mitigate the time and resource complexity. These strategies are all general and practical ways to speed up the manual tuning of recommender systems, which are itemized below. Several rank consistency tests are described in paper to provide the evidence and illustrate the feasibility of adopting these methods.

3.5.6 Techniques for Automated Machine Learning. AutoML brings about three advantages: (1) liberating the community from the time-consuming and trial-and-error tuning processes, (2) facilitating the development of machine learning in organizations or business, and (3) making machine learning universally accessible to all domain scientists. Echoing with the traditional pipeline, we classify AutoML into three categories (Figure 2): (1) automated feature engineering (AutoFE), (2) automated model and hyperparameter tuning (AutoMHT), and (3) automated deep learning (AutoDL). AutoFE discovers informative and distinct features for the learning model. AutoMHT tunes hyperparameters of a specific learning model or an entire machine learning pipeline automatically (AutoPipeline). AutoDL is a subfield of AutoMHT focusing on the automatic design of neural networks. Since deep learning has succeeded in image and language tasks to extract hierarchical features in an end-to-end manner without domain understanding, we explicitly separate AutoDL from AutoMHT.

Like machine learning, the essence of AutoML is an optimization problem, specifically bi-level optimization, with an expansive search space including features, hyperparameters, models, and network architectures. AutoML is an NP-complete problem as intractable as machine learning. As a result, several heuristic techniques are

proposed to solve the sophisticated optimization. On top of the categorization of AutoML, we will review three categories of AutoML in terms of techniques and frameworks. The techniques span optimization approaches, including reinforcement learning, evolutionary algorithm, Bayesian optimization, and gradient approaches. We generalize them by the iterative solver and explain how these techniques are deployed in AutoML. In the framework dimension, we list representative AutoML frameworks in commercial services and open source communities.

Different from existing surveys, we map AutoML to the optimization paradigm and elucidate AutoML techniques from the iterative solver. In this manner, the optimal solution is produced by the iterative procedure, where the explorer seeks out candidates and the evaluator inspects the effectiveness of the candidates. This illustration bridges the bi-level optimization and AutoML.

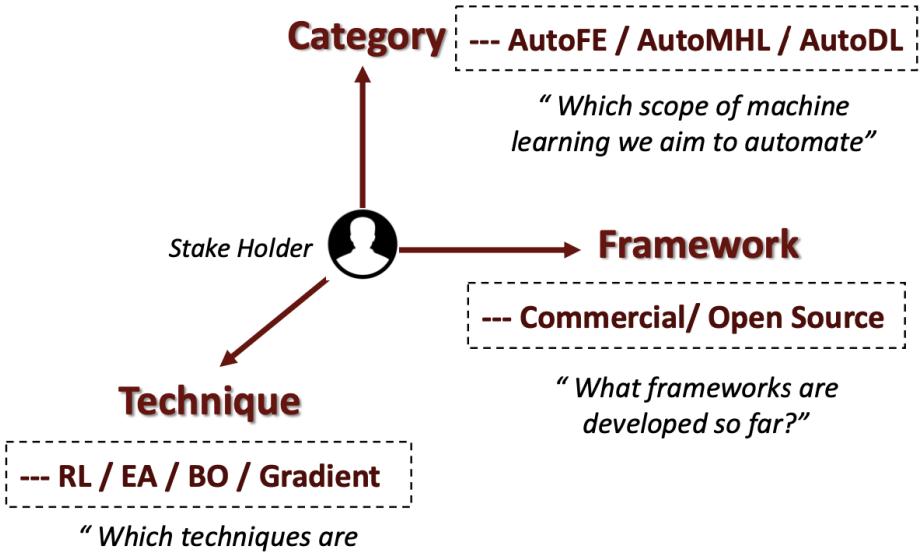

Figure 15. Content of the AutoML Survey: Categories, Techniques, and Frameworks

The contributions of this paper [26] are in the following. Figure 15 illustrate AutoML from the formal statement of machine learning, depict AutoML as a bi-level optimization problem, and display their optimization formats for the AutoFE, AutoMHT, and AutoDL. (2) We propose the iterative solver to generalize AutoML techniques and analyze these techniques commonly used in AutoML. Readers will learn how to apply them to three categories in AutoML. (3) We review the commercial and open-source frameworks in AutoML. Readers can get available AutoML tools for their research.

4. RESULTS AND DISCUSSION

We created three competitive AutoML systems that had consistent performance across the multiple evaluations and dry runs during the duration of the program. Also, the systems were capable of adapting to new requirements and primitives, such as data augmentation. The last AutoML system, Axolotl, is an easily extendable system that allows different users to add new search strategies or add new problem specifications to accommodate new problem types. Axolotl has demonstrated its versatility by being integrated into TODS to make D3M interactions easier and its robustness by performing well on the D3M leaderboard for Classification and Regression tasks. For future work, we aim to improve the performance by adding optimization techniques related to caching results to avoid recomputing things; also, we are looking towards adding state-of-the-art search methods to the system so users have more diversity.

TODS provide an end-to-end time series outlier detection system with more than 72 primitives. As far as we know, the project leader has been contacted by engineers in industry which shows that TODS is now being adopted into their experimental and production system. The toolkits is now raising more than 300 attentions from the GitHub open source community (without any advertisement). In the near future, we will advance the version of TODS with more advanced searching algorithms such as reinforcement learning searcher, tree-based searcher and Bayesian searcher. In addition, examples on Google Collab will be available soon, and we are now submitting articles to major social media to advertise this system to gain more users. In the long term future work, automated deep learning system such as AutoKeras will be integrated into TODS to provide more powerful performances.

Autokeras discovers promising neural architectures in image classification. The open-source package attracts more than 7.9k stars and 1.3k fork times in GitHub. Now, the Google Keras team adopts Autokeras as a part of Keras in the same ecosystem. Keras developers could benefit from the power of AutoML quickly. Moreover, Autokeras utilizes KerasTuner (e.g. Greedy search, Bayesian optimization, HyperBand) to tune the neural architecture and corresponding hyperparameters jointly. This integration demonstrates Autokeras could outperform AutoGluon (Amazon NAS framework) in terms of CIFAR-10, TITANIC, and House prediction dataset. Autokeras is developing to support more machine learning tasks, including time-series data classification and object detection. Furthermore, Autokeras will adopt common NAS acceleration, such as weight sharing supernet and data, and model parallelization to speed up the search efficiency. We hope that Autokeras evolves to the framework that makes machine learning accessible to everyone.

During the program, we made several attempts to convey what we proposed: Feed-forward pipelines. Unfortunately, due to several reasons, we were unable to follow it. One of the main reasons is due to engineering challenges. The Feedforward pipelines require that primitives implemented some interfaces related to the back-propagation of gradients which were not implemented due to their complexity. Besides the implementation complexity of primitives, also we realized that tuning end-to-end machine learning pipelines that use primitives with differentiable interfaces required a special implementation of the runtime. Such runtime represented a challenge since it could only work if all primitives on the pipelines implement the special characteristics; this runtime

would also not be available during evaluation due to the standardization of the procedures. We started to work in specialized primitives for this problem with UBC; unfortunately, they were not available for most of it, leaving all the work for us. The other main reason we did not achieve our proposed goal is due to constant changes of directions. The program was changing goals once or twice a year, making it difficult to focus on a single task.

Through the program, we became one of the most important contributors to the common infrastructure. Our works go from designing and discussing and implementing multiple components of the APIs, providing tools and help to Data Machines for running dry runs and evaluations. During our work for the common infrastructure, to be precise on the D3M core package, we noticed through time that it had been one of the bottlenecks of the program. The complexity of the D3M core package increased through time, making it difficult for other performers to contribute to it or even write primitives; this resulted in an overly engineering API that limited the growth of the primitive library, among other things. Also, our work on this area focused on adding OpenML compatibility, which in theory should increase the added value of the meta learning database.

5. CONCLUSIONS

We introduced multiple AutoMl systems that evolved with the program. Each of the systems focuses on different problems; Axolotl is a framework for developing AutoML search algorithms. TODS is a specialized system focused on time series problems, and AutoKeras focuses on Neural Architecture Search. We also made an impact in the AutoML community by having multiple works presented at different conferences.

References

[1] M. Milutinovic, B. Schoenfeld, D. Martinez-Garcia, S. Ray, S. Shah and D. Yan, "On Evaluation of AutoML Systems," in *7th ICML Workshop on Automated Machine Learning* , 2020.

[2] M. Milutinovic, J. Honaker, C. Bethune, R. Rampin and D. Martinez-Garcia, "Automl-rpc," 1 May 2021. [Online]. Available: https://gitlab.com/datadrivendiscovery/automl-rpc.

[3] M. Milutinovic, D. Martinez-Garcia, J. Gleason, R. Zinkov, C. Bethune and T. Yang, "Common Primitives Repository," 1 May 2021. [Online]. Available: https://gitlab.com/datadrivendiscovery/common-primitives.

[4] D. Martinez-Garcia and M. Milutinovic, "Dummy TA3 evaluation tool.," 1 May 2021. [Online]. Available: https://gitlab.com/datadrivendiscovery/dummy-ta3.

[5] M. Hoffmann, A. Yepremyan and D. Martinez-Garcia, "D3M Keras Wrapper," 1 May 2021. [Online]. Available: https://gitlab.com/datadrivendiscovery/jpl-primitives.

[6] M. Feurer, K. Eggensperger, S. Falkner, M. Lindauer and F. Hutter, "Auto-Sklearn 2.0," in *Advances in Neural Information Processing Systems 28*, 2020.

[7] J. Vanschoren, J. van Rijn, B. Bischl and T. Luis, "OpenML: networked science in machine learning.," in *SIGKDD Explorations 15(2)*, 2013.

[8] M. Smith, C. Sala, J. M. Kanter and K. Veeramachaneni, "The Machine Learning Bazaar: Harnessing the ML Ecosystem for Effective System Development," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020.

[9] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

[10] H. Zhang, S. Si and C.-J. Hsieh, "GPU Acceleration for Large-scale Tree Boosting.," in *Conference on Machine Learning and Systems*, 2018.

[11] D. Martinez-Garcia, "TAMU Primitives," 1 May 2021. [Online]. Available: https://gitlab.com/TAMU_D3M/d3m_primitives.

[12] J. Li, K. Cheng, S. Wang, F. Morstatter, R. Trevino, J. Tang and H. Liu, "Feature selection: A data perspective," in *ACM Computing Surveys (CSUR)*, 2018.

[13] D. Martinez-Garcia, "Statistical Analysis for Supervised Classification Machine Learning Models for Tabular Data," 1 May 2021. [Online]. Available: https://gitlab.com/dmartinez05/d3m_primitive_profiler/-/blob/master/report.pdf.

[14] J. Haifeng and D. Martinez-Garcia, "Texas A&M AutoML Prototype," 1 May 2021. [Online]. Available: https://gitlab.datadrivendiscovery.org/TA2/Texas-AnM-University_tamu_1.0.0/-/tree/master.

[15] D. Martinez-Garcia, T. Yang and Y.-W. Chen, "D3M AutoML System TAMUTA," 1 May 2021. [Online]. Available: https://gitlab.com/TAMU_D3M/Winter_2018_tamuta2.

[16] D. Martinez-Garcia and Y.-W. Chen, "D3M AutoML System Axolotl," 1 May 2021. [Online]. Available: https://gitlab.com/axolotl1/axolotl.

[17] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. Jordan and I. Stoica, "Ray: A Distributed Framework for Emerging AI Applications," in *13th USENIX Symposium on Operating Systems Design and Implementation*, 2018.

[18] J. Haifeng, S. Qingquan and X. Hu, "Auto-Keras: An Efficient Neural Architecture Search System," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019.

[19] Y.-W. Chen and D. Martinez-Garcia, "D3M AutoKeras Compiler," 1 May 2021. [Online]. Available: https://gitlab.com/TAMU_D3M/d3m_autokeras_wrap.

[20] K.-H. Lai, D. Zha, G. Wang, J. Xu, Y. Zhao, D. Kumar, Y. Chen, P. Zumkhawaka, M. Wan, D. Martinez-Garcia and X. Hu, "TODS: An Automated Time Series Outlier Detection System," in *arXiv*, 202.

[21] H. Jin, Q. Song and X. Hu, "Discriminative Graph Autoencoder," in *IEEE International Conference on Big Knowledge (ICBK)*, 2018.

[22] Q. Song, S. Chang and X. Hu, "Coupled Variotional Recurrent Collaborative Filtering," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019.

[23] F. Yang, N. Liu, M. Du, K. Zhou, S. Ji and X. Hu, "Deep neural networks with knowledge instillation," in *Proceedings of the 2020 SIAM International Conference on Data Mining*, 2020.

[24] Y.-W. Chen, Q. Song, X. Liu, P. Sastry and X. Hu, "On Robustness of Neural Architecture Search Under Label Noise," in *Frontiers in Big Data*, 2020.

[25] Q. Song, D. Cheng, H. Zhou, J. Yang, Y. Tian and X. Hu, "Towards Automated Neural Interaction Discovery for Click-Through Rate Prediction," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020.

[26] Y.-W. Chen, Q. Song and X. Hu, "Techniques for Automated Machine Learning," in *ACM SIGKDD Explorations Newsletter*, 2021.