AUTOMATED MACHINE LEARNING SYSTEMS:

EVALUATION, EASE OF USE, DATA TRANSFORMATION

A Dissertation

by

DIEGO SERAFIN MARTINEZ GARCIA

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Xia Hu
Committee Members,	Frank Shipman
	Zhangyang Wang
	Yang Shen
Head of Department,	Scott Schaefer

May 2022

Major Subject: Computer Science

ABSTRACT

Machine Learning (ML) has been rapidly progressing through the years due to its versatility to solve different problems. As a result, many machine learning frameworks have been created that are a collection of different algorithms. However, data scientists often struggle to determine which one to use to develop their ML solutions due to many options. For this reason, many Automated Machine Learning (AutoML) systems have been created to help users create ML solutions easily by defining the problem they want to solve, providing the data, and setting a budget such as time search for a solution.

Through the years, many AutoML systems have been developed, and their applications range from solving simple tasks such as tabular classification to more complex such as object detection. Due to many AutoML systems, it becomes challenging for users to determine which one suits them the best because most of the systems focus on specific tasks and data, and sometimes they overlap on the tasks they can solve. Another issue that users need to be aware of is that although most of the search process is already automated, it is necessary for the user to get involved in data preprocessing in most systems. Such preprocessing can be trivial, from selecting images files to more challenging tasks such as merging multiple database tables.

Another aspect of AutoML systems is the research involved in the development. AutoML research can focus on the way of searching, to some more deep processes such as optimizing how models are run. This research leads to the creation of many AutoML systems every year. Creating an AutoML system is challenging since there are many things to consider, from the design to the implementation, and attempting to use an existing system to test a new hypothesis becomes challenging. The challenge of reusing an existing AutoML system is that most systems were designed towards proposing some research improvement rather than usability. Another problem is that these systems are not maintained, and if they are maintained, it is difficult to use them due to the lack of documentation.

Enabling the advance on AutoML is challenging. Firstly, it is necessary to standardize components, so there is a more efficient way to compare different frameworks. There are many AutoML systems every year with new search strategies. However, it becomes challenging to objectively compare them since they could be improving in other areas rather than the ones claimed. Secondly, creating an AutoML system should not be challenging since it can stop many researchers from contributing to the field. Creating a new AutoML system with the sole purpose of testing a new component should not be difficult. Thirdly: we identify that state-of-the-art AutoML systems only focus on model selection and hyperparameter tuning while leaving room for improvement on the data preprocessing. To tackle the challenges above, several contributions are made in the preliminary work, and future work is proposed to conclude the dissertation:

- The first contribution of this research dissertation is the development of standards for AutoML, which generalize components that have been used for a while and give them proper definitions.
- Second, we propose a better methodology for the evaluation of AutoML systems that provide a better understanding of the capabilities of different systems
- To alleviate the burden of human efforts to create single-use AutoML systems, we propose an extendible to enable AutoML. The proposed AutoML frameworks enable customizable AutoML solutions without designing and implementing every aspect of an AutoML system.
- Considering the potential benefit of data preprocessing search for AutoML in the last piece of work, we focus on the preprocessing search by creating an end-to-end framework that takes advantage on contextual feature similarities.

DEDICATION

To my family.

ACKNOWLEDGMENTS

Foremost, I would like to thank my family, my mother Celia for her emotional support and always being there, my father Serafin for his wisdom, my sister Lourdes for visiting me and keeping an eye on me all the time, and my sister Selina for her affection. Without my family, none of this work would be possible. Also, I was fortunate to have many friends across the journey that provided advice and support when I needed it. I would like to thank Jorge, Maria, Irving, Patricia. I would like to especially thank Jonathan Martinez and Tsung-Lin Yang, who helped me stay on track and never give up; I really appreciate them.

Most of this work has been done through collaborations through the DARPA D3M program. I want to thank everyone on the program, especially Brian Sandberg, Mitar Milutinovic, Shelly Stanley, Brandon Schoenfeld, Curtis Lisle, Daragh Hartnett, Dayne Freitag, Mark Hoffmann, Sonia Castelo, and Remco Chang. Thanks for the hard work through these years and the wonderful discussions and collaboration on such an exciting project. Also, I would like to thank the JPL team for their contributions to the project and for making me feel welcome while doing my internship; special thanks to Brian Wilson, Chris Mattmann, Alice Yepremyan, Sujen Shah, and Bhavin Shah.

I would like to express my gratitude to my advisor, Dr. Xia Hu, who provided his mentorship, support, and almost infinite patience. I would like to thank the rest of my dissertation committee, Dr. Frank Shipman, Dr. Zhangyang Wang, and Dr. Yang Shen, for their continuous advice and support during my Ph.D.

Finally, I would like to thank an extraordinary person that is no longer among us, that helped me during rough times by providing support and music and made me challenge myself to achieve greater things. Thank you, my friend, Luna. Rest in peace.

Special thanks to Texas A&M University and all the funding agencies, including The Mexican National Council for Science and Technology and Defense Advanced Research Projects Agency.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professor Xia Hu [advisor], Professor Frank Shipman, and Professor Zhangyang Wang of the Department of Computer Science and Engineering, and Professor Yang Shen from the Deparment of Electrical & Computer Engineering.

Funding Sources

This work is, in part, supported by The Mexican National Council for Science and Technology (CONACYT), and by DARPA (#FA8750-17-2-0116)The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government or the Mexico Government.

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

1.1 Motivation and Challenges

Machine Learning (ML) has been rapidly progressing through the years due to its versatility to solve different problems. As a result, many machine learning frameworks have been created that are a collection of different algorithms. However, data scientists often struggle to determine which one to use to develop their ML solutions due to many options. For this reason, many Automated Machine Learning (AutoML) systems have been created to help users create ML solutions easily by defining the problem they want to solve, providing the data, and setting a budget such as time search for a solution.

Through the years, many AutoML systems have been developed, and their applications range from solving simple tasks such as tabular classification to more complex such as object detection. Due to many AutoML systems, it becomes challenging for users to determine which one suits them the best because most of the systems focus on specific tasks and data, and sometimes they overlap on the tasks they can solve. Another issue that users need to be aware of is that although most of the search process is already automated, it is necessary for the user to get involved in data preprocessing in most systems. Such preprocessing can be trivial, from selecting images files to more challenging tasks such as merging multiple database tables.

Another aspect of AutoML systems is the research involved in the development. AutoML research can focus on the way of searching, to some more deep processes such as optimizing how models are run. This research leads to the creation of many AutoML systems every year. Creating an AutoML system is challenging since there are many things to consider, from the design to the implementation, and attempting to use an existing system to test a new hypothesis becomes challenging. The challenge of reusing an existing AutoML system is that most systems were designed towards proposing some research improvement rather than usability. Another problem is

that these systems are not maintained, and if they are maintained, it is difficult to use them due to the lack of documentation.

AutoML systems often follow simple patterns to generate machine learning pipelines; the embedded knowledge on the system bounds such patterns. In most cases, the ML pipelines generated follow a simple structure that contains data transformation and then the machine learner. Although this approach is effective, it restricts the search by focusing on model selection and hyperparameter tuning. Feature transformation for preprocessing can become complicated to automate since it requires having some knowledge of the data beforehand.

1.2 Related Work

Evaluation of AutoML systems. Several frameworks attempt to help practitioners to identify strengths and weaknesses of AutoML systems by taking the practical approach of evaluating systems as-is and comparing only the predictions generated by each system, such as the AutoML Challenge Series [1], and Benchmarking automatic machine learning frameworks [2]. These comparisons, as well as the benchmarking suite OpenML benchmarking suites and the OpenML100 [3] focus only on tabular data consisting primarily of classification and regression tasks with numerical and categorical data and some missing values, but the AutoML Challenge Series also includes various other problem types that are preprocessed to fit the tabular paradigm. Notably, the AutoML Benchmark evaluates systems given multiple time budgets, helping demonstrate how AutoML systems evolve with increasingly available resources. Other studies such as Survey on automated machine learning [4] used 137 OpenML datasets to evaluate multiple AutoML systems, such as, auto-sklearn [5], TPOT [6], hyperopt-sklearn [7], RoBO [8], BTB [9].

Standardization of AutoML components. Every year, numerous Machine Learning algorithms are created, in which most of them there almost to none existing documentation. To solve this, Machine Learning libraries have been created containing collections of different algorithms that are updated through time. One of the most outstanding examples of this is Scikit-Learn Library (Sklearn) [10] which is one of the most extensive collections of machine learning algorithms that keeps growing with contributions. Sklearn has defined a standard of rules to follow with its influence

to implement the most standard code interface such as the function fit and predict. Later on, Sklearn introduced automated pipelines that only use functions that fit and predict. The overall approach to standardize ML algorithms follows human interaction. The interfaces are designed to be human-friendly by providing enough documentation for humans to get a better sense of what everything does in natural language. Another important work that looks towards the standardization of AutoML is OpenML [11] which takes further the concept of pipelines defined by Sklearn. OpenML defines ML pipelines as flows that later on are serialized and stored on the OpenML database. OpenML also generates the concept of runs, which focuses on the information of a workflow that is run or evaluated with a dataset; these runs also store information regarding the metrics used to be evaluated. Workflow runs are then submitted to the OpenML database to be stored and later be used to either attempt to reproduce the experiments or to be used as data to design or train new AutoML systems.

Building an AutoML System. Every year several AutoML systems are created to test a new hypothesis. Each of the systems focuses on different strategies to generate better pipelines. The most common approach for such systems is to have templates mainly in charge of doing the data preprocessing and then applying some model selection algorithms. Other approaches such as genetic programming (TPOT), reinforcement learning (AlphaD3M [12]), probabilistic matrix factorization [13] or combining multiple approaches like Bayesian Optimization, and Metalearning (AutoSklearn) can yield a substantial improvement on performance. Unfortunately, the systems are designed not to be used as an AutoML system, and it can be challenging to use them as a base to create new systems or test different search strategies. AutoSklearn is the most widely known and can be extended to test new algorithms, but the learning curve is very steep due to all intricated parts of the program.

Preprocessing Pipeline Search for AutoML. Current AutoML solutions are capable of generating complex pipelines by creating ensembles of simple ones such as AutoSklearn [5] or by adding multiple data transformation primitives on the middle such as AlphaD3M [12]. A more complex approach can be found on DeepLine [14] in which multiple pipelines are created on a parallel that later is ensembled; this framework uses Deep reinforcement learning to learn across multiple datasets. All of the methods have a common approach: apply the same data transformation for all

of the same data. This approach leads to a good overall performance since it relies on embedded human knowledge about how to tread different feature types. So far, Preprocessing Pipeline search on AutoML systems focuses on the best cases to select one of the very few options of available preprocessing pipelines. These pipelines are humanly handcrafted or a simple combination of different data transformation primitives such as imputers, encoders, and scalers.

1.3 Dissertation Contributions

Enabling the advance on AutoML is challenging. Firstly, it is necessary to standardize components, so there is a more efficient way to compare different frameworks. There are many AutoML systems every year with new search strategies. However, it becomes challenging to objectively compare them since they could be improving in other areas rather than the ones claimed. Secondly, creating an AutoML system should not be challenging since it can stop many researchers from contributing to the field. Creating a new AutoML system with the sole purpose of testing a new component should not be difficult. Thirdly: we identify that state-of-the-art AutoML systems only focus on model selection and hyperparameter tuning while leaving room for improvement on the data preprocessing. To tackle the challenges above, several contributions are made in the preliminary work, and future work is proposed to conclude the dissertation:

- The first contribution of this research dissertation is the development of standards for AutoML, which generalize components that have been used for a while and give them proper definitions, as well as, generate APIs so AutoML can use them so they can be built by using the same building blocks.

- Second, we propose a better methodology for evaluating AutoML systems that provide a better understanding of the capabilities of different systems that take advantage of the use of the same building components on the systems.

- To alleviate the burden of human efforts to create single-use AutoML systems, we propose an extendible framework to enable AutoML. The proposed AutoML frameworks enable customizable AutoML solutions without designing and implementing every aspect of an

AutoML system. The main focus is to allow developers and researchers to prototype and test different methodologies within AutoML quickly.

- Considering the potential benefit of data preprocessing search for AutoML. We proposed a framework for the exploration of feature-wise preprocessing pipeline search. We focus on reducing the complexity of the preprocessing pipeline search space by clustering features based on an end-to-end framework that guides the search.

1.4 Dissertation Overview

The remainder of this dissertation is organized as follows:

- **Chapter 2: On Evaluation of AutoML Systems.** In this chapter, we develop a framework to enable Automated machine learning by standardizing the language for machine learning pipelines and providing a deterministic way to evaluate them.

- **Chapter 3: AutoML System: Axolotl.** In this chapter, we develop an Automated Machine learning system that enables non-expert to expert users to create, test, and evaluate their Automated Machine learning methods without the burden of creating a system from scratch.

- **Chapter 4: Feature-wise Transformation for AutoML.** In this chapter, we explore the benefit of searching for feature transformation methods to boost the performance of machine learning pipelines by proposing an end-to-end discovery framework to search for transformation for a group of features automatically.

- **Chapter 5: Conclusions and Future Research Opportunities.** We conclude the dissertation with a summary of contributions and discuss potential research directions to the results presented.

2. ON EVALUATION OF AUTOML SYSTEMS: FUNDAMENTAL AUTOML EVALUATION PRACTICES AND UNIFIED OPEN-SOURCE MACHINE LEARNING FRAMEWORK[1]

In this chapter, we tackle the standardization and evaluation for AutoML systems. Our developed framework can provide more information to AutoML systems about their components and attempts to standardize the components across multiple systems. As for the evaluation, the framework can provide better insight into how AutoML systems perform in different scenarios.

2.1 Introduction

The AutoML community has produced many academic and non-academic systems: AlphaD3M [12], Auto-sklearn [5], Google Cloud AutoML [15], H20 [16], Hyperopt [7], Auto-WEKA [17], SmartML [18], Auto-Net [19], auto_ml [20], ML-Plan [21], TPOT [6], Mosaic [22], Auto-Meka [23], RECIPE [24], TransmogrifAI [25], Alpine Meadow [26], The Automatic Statistician [27], ATM [28], Rafiki [29], Adaptive TPOT [30], Oboe [31], TPE [32], SMAC [33], FABOLAS [34], BOHB [35], APRL [36], Hyperband [37], and DarwinML [38]. Table 2.1 shows an overview of the different techniques used for these systems; most of them focus on Bayesian Optimization, Evolutionary Algorithms, and Reinforcement Learning, but also other systems use non-conventional techniques such as matrix factorization, among others. Some systems focus on neural networks only: MetaQNN [39], DEvol [40], Auto-Keras [41], Neural Network Intelligence [42], ENAS [43], Neural Architecture Search [44], NASNet [45], DARTS [46], Proxyless [47], NASBot [48], and NAONet [49]. We observe [50] that their approaches, even the programming languages, vary significantly, complicating system comparison. Table 2.2 shows an overview of the different techniques used for these Neural Architecture Search systems; most of them focus on Bayesian Optimization, Evolutionary Algorithms, Reinforcement Learning, and Gradient-based Approaches.

[1]This chapter is reprinted with permission from "On evaluation of automl systems" by Mitar Milutinovic, Brandon Schoenfeld, Diego Martinez-Garcia, Saswati Ray, Sujen Shah, and David Yan, 2020. In Proceedings of the ICML Workshop on Automatic Machine Learning. 2020. Copyright 2020 by M. Milutinovic, B. Schoenfeld, D. Martinez-Garcia, S. Ray, S. Shah, D. Yan.

Bayesian Optimization	Evolutionary Algorithm	Reinforcement Learning	Other techniques
Auto-sklearn [5]	TPOT [6]	AlphaD3M [12]	H20 [16]
Hyperopt [7]	auto_ml [20]	Rafiki [29]	ML-Plan [21]
Auto-WEKA [17]	Auto-Meka [23]	APRL [36]	Mosaic [22]
SmartML [18]	RECIPE [24]	Hyperband [37]	Alpine Meadow [26]
TransmogrifAI [25]	Adaptive TPOT [30]		Auto Statistician [27]
ATM [28]	DarwinML [38]		Oboe [31]
TPE [32]			
SMAC [33]			
FABOLAS [34]			
BOHB [35]			

Table 2.1: List of AutoML systems that focus on model selection and hyperparameter tuning categorized according to the search strategy used. Most of the frameworks fit into three main categories: Bayesian Optimization, Evolutionary Algorithms, and Reinforcement Learning. An extra category has been added as Other Techniques since these frameworks do not fit the previous ones.

Bayesian Optimization	Reinforcement Learning	Gradient-based Approaches	Evolutionary Algorithm
Auto-Keras [41]	MetaQNN [39]	DARTS [46]	DEvol [40]
NN Intelligence [42]	NAS [44]	Proxyless [47]	
NASBot [48]	ENAS [43]	NAONet [49]	
	NASNet [45]		

Table 2.2: List of AutoML systems that focus on Neural Architecture Search. The systems are categorized according to the search strategy used: Bayesian Optimization, Reinforcement Learning, Gradient-based Approaches, and Evolutionary Algorithm

Existing evaluations, including the AutoML Benchmark [51], the AutoML Challenge Series [1], and another benchmarking study [2], attempt to help practitioners identify strengths and weaknesses of systems by taking the practical approach of evaluating systems "as-is", comparing only the predictions of each system's chosen pipeline. These comparisons, as well as another benchmarking suite [3], focus on tabular data, consisting primarily of classification and regression tasks with numeric and categorical data and some missing values, but the AutoML Challenge Series also includes various other problem types that are preprocessed to fit the tabular paradigm. Notably, the AutoML Benchmark evaluates systems given multiple time budgets, helping demonstrate how AutoML systems evolve with increasingly available resources. Another study [4] used 137 OpenML datasets to evaluate multiple AutoML systems (auto-sklearn, TPOT, hyperopt-sklearn, RoBO [8], BTB [9]). TUPAQ [52] emphasizes large-scale distributed and scalable machine learning, focusing their evaluation on scalability and convergence rates.

AutoML Benchmarks evaluate different systems by providing fixed problems that consist of a single table with categorical and numerical columns and some missing values. Then the benchmarks provide different resource constraints in which only the search time changes most of the time. This approach for measuring the AutoML system performance can be drastically affected by external factors, such as the programming language used for the system, the available search space, and the determinism of the final solutions. AutoML systems' comparison by these means does not provide a fair comparison since they only consider the final output and not any other part of the process.

In order to make a more fair comparison among AutoML systems, it is necessary to consider other factors such as the use of the same framework. A fair comparison becomes a challenging task since different systems attempt to get performance improvements in different ways, such as focusing on model selection, hyperparameter tuning, or just by speeding up the evaluation of solutions so they can cover a more extensive search space.

To achieve the goal of creating a more fair comparison framework for AutoML systems, it is necessary to tackle three challenges: standard machine learning primitive representation, unified broad standard pipeline representation, standard pipeline execution. We briefly describe each of the challenges and their importance:

- Current machine learning primitives such the one on scikit-learn [10] library are made for a human to use, and their documentation provides a general idea of their usage. Unfortunately, machine learning systems cannot use almost any of the knowledge on the documentation to learn the proper use of the primitives, such as their interfaces. AutoML systems have to wrap primitives manually so that they can be used in an automated fashion. The drawback for this approach is that every system has a different method to wrap them, thus making it difficult to use them across frameworks since they only focus on some specific information for their systems.

- The pipeline generated by current AutoML systems can be from a sequence of ML primitives executed one after another or a more complex representation that is defined as the composition of the ML Primitives (scikit-learn). None of the current pipeline representations created by AutoML authors can capture a general pipeline representation containing all the different approaches, making it difficult to compare their solutions across different frameworks.

- The general output of AutoML systems are the predictions for a given problem. Although this is the final goal of AutoML, in practice, this makes the process of comparing them even more challenging since the system may or may not be deterministic, and their solutions can be implemented on artifacts that they can only use. For this reason, AutoML systems must express their solutions in the general pipeline language and be evaluated in a deterministic way that is not dependent on the system that generated them.

2.2 Preliminaries

The implementation of this work (D3M Core Package [2]) has been a collaborative effort among different institutions via the Data Driven Discovery of Models Project by DARPA.

2.2.1 Machine Learning Primitives

A primitive is a basic building block that is the implementation of a function or an algorithm. The primitives can be written by using any programming language, but they must have a python interface. The python interface contains metadata that defines information about the wrapped primitive such as the algorithm type and information about all the parameters and the accepted inputs and outputs. Primitives under our definition have two different kinds of hyperparameters: control hyperparameters, which are these parameters that control the logic of the primitive, and tuning hyperparameters that are the ones that directly affect the performance of the primitives.

Current implementations of machine learning primitives such as the Sklearn [10] lack machine-readable information. The information contained in such primitives' documentation mainly focuses on explaining in natural language what each parameter does and sometimes providing more information about how to tune the parameters, such as the range of a variable. This approach makes it even more difficult for AutoML systems to ingest or learn how to use the primitives. As previously mentioned, this problem can be solved by creating a standardization of primitive information in the machine-readable interface. This interface mainly focuses on three aspects, general information about the primitives, hyperparameters definitions, and standard class methods.

The first aspect of the general information about the primitive is metadata on JSON format that contains a universally unique identifier (UUID) to help keep track of the primitive through different versions. The metadata also contains automatic installation instructions that mainly focus on python packages and contain other relevant dependency information such as external code needed for the primitive to work, i.e., Java code. Other information such as the primitive family, algorithm types, and keyboards are provided to help AutoML systems to identify primitives in a more accessible

[2] `https://gitlab.com/datadrivendiscovery/d3m`

manner depending on specific characteristics.

The second aspect about hyperparameters definition, the primitive interface should be able to provide more information about the different hyperparameters that can change and their data types such as categorical, numerical, or even more complex structures such as other primitives. The hyperparameters also need to contain information about how to tune them, like the different possible values, the range of values, and, if possible, the distribution of them. These hyperparameters should be descriptive enough so the AutoML system can select and sample different values within their definition. An important thing to mention is that such definitions can only be validated during execution and not via semantics. Several primitives have some hyperparameter definitions, such as the clustering method K-Means where the number of clusters is at most as many input features.

The third aspect of standard class methods is that the primitive interface must have an expected set of functions so the AutoML system can always call them. For this work, the expected functions are *set_training_data*, *fit* and *produce*, This function semantics are general enough to cover most of the cases for already written ML primitives. Other popular and useful methods such as exposing *log_likelihoods* can be added by extending the bases classes and following the naming conventions. The functions must also contain metadata information about the parameters, inputs, and outputs. The function's parameters could be something extra that change the function's behavior, such as defining the number of iterations for the method to take or the maximum amount of time for the function to take. The inputs mostly apply to *set_training_data* and *produce* functions since they pass the data to the primitive, e.g., the data that a Random forest primitive uses for train or to produce predictions. The outputs are often associated with producing methods.Standard methods can be found on base classes provided on the D3M core package, such as supervised and unsupervised learning classes.

A visualization example of a Sklearn wrapped primitive with the interface can be found in Figure 2.1. In this case, the Random Forest primitive is being wrapped by using a supervised learning primitive base class since it uses inputs (features) and outputs (predictions). The wrapped primitive contains some relevant information such as the algorithm type, random forest, the primitive family

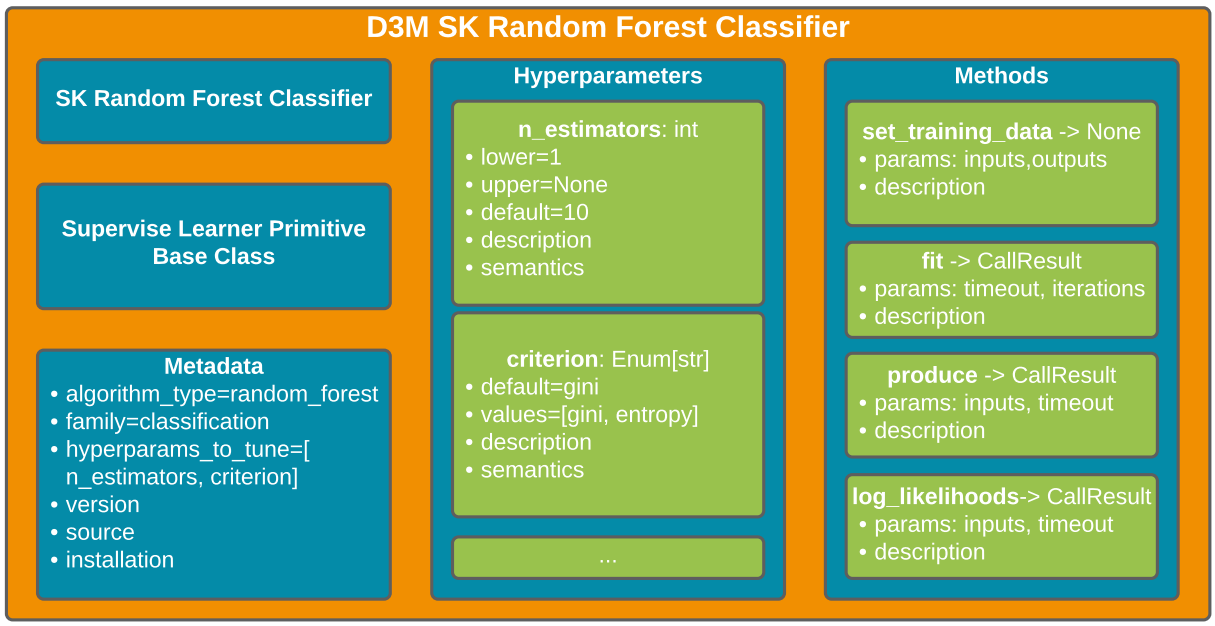


Figure 2.1: An illustrative example of the exposed interfaced of the Random Forest primitive by Sklearn wrapped by the D3M API. This wrapped primitive contains general information on the metadata, as well as standard methods and hyperparameters definitions.

classification, version, and other relevant information regarding the installation instructions. Then, the definition such as types and possible values of different hyperparameters such as the number of estimators (numerical) and the decision criteria (categorical) with multiple predefined values. Finally, the wrapped primitive contains standard methods, such as *set_training_data* that takes features and predictions that later on are used to train the ML learner by using *fit* and finally generate prediction by using *fit*.

2.2.2 Pipeline Language

A machine learning pipeline consists of a series of steps of different operations that are applied to the input data to achieve a goal, such as classification or regression. Currently, there are several approaches to represent this workflow, such as sklearn pipeline[3]) that consist of a sequence of steps applied one after another. The main disadvantage of this kind of representation is that it does not allow the generation of more creative solutions that are not necessarily linear; for example, applying

[3] `https://scikit-learn.org/stable/modules/generated/sklearn.pipeline`

a transformer to a specific set of columns or even having a set of ML learners to do ensembling learning.

In our proposed framework, we defined a pipeline as a Directed Acyclic Graphs representing end-to-end execution of a program containing inputs and outputs. For the graph pipeline representation, we define each node as a pipeline step and each edge as the relation of inputs and outputs among the steps. Pipeline steps can be divided into two categories primitive steps and sub-pipeline steps. A primitive step consists of an ML primitive with defined inputs, outputs, parameters, and hyperparameters, while sub-pipeline steps refer to another existing pipeline. The sub-pipeline approach enables the composition of pipelines and primitives that provides more flexibility. The approach of representing pipelines as DAGs is general enough to be able to represent the already existing pipelines and room for new ones. The proposed pipeline representation is language-independent and can be represented in JSON or YAML serialization.

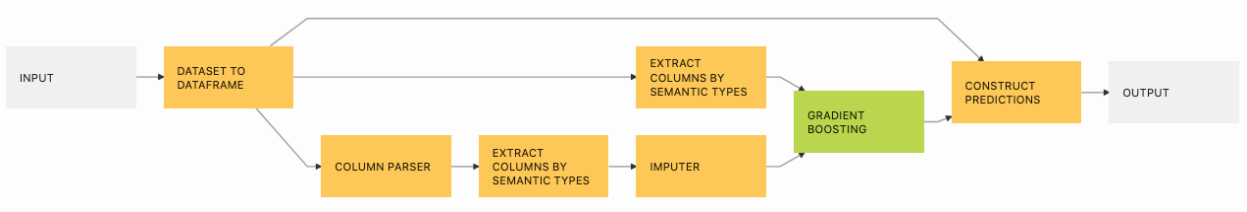


Figure 2.2: An example of DAG pipeline that can generate predictions for data with only numerical values. The pipeline focuses on three aspects, missing feature imputation, passing targets to the learning, and formatting the output predictions.

An example of a pipeline that can generate classification predictions for problems with only numerical values can be found in Figure 2.1 and the JSON representation can be found in Appendix A. This pipeline does not follow the conventional sequence as other frameworks but instead applies different operations for different input columns. The most upper branch allows keeping track of the

sample indexes because some other problem types can vary. The middle branch keeps track of the targets later passed to the ML learner. Finally, the bottom branch performs profiling to determine the data types of the features (since everything is considered as strings) to later apply an imputation method and finally use the transformed features to be used by the ML to generate predictions.

2.2.3 Reference Runtime

Machine learning pipelines represent the workflow used to solve some tasks. The same pipeline can be executed in different ways, leading to different results. For this reason, we propose a deterministic reference runtime that executes the pipeline with given input and produces predictions in the same way. The reference runtime focuses on two execution semantics, fit and produce. In Figure 2.3 we elaborate in more details such semantics. For the fit phase, the runtime uses as parameters a pipeline and a set of inputs, then iterates overall steps and calls the methods *FitPrimitive* primitive if the step is a primitive step or *fit*; after this calls, the runtime store the current state of the executed primitive or pipeline into the state. Later on, the produce method is called to generate more data stored in the environment that would be used later to propagate to the following states. The produce semantic follows the same flow as the fit by iteration over all the pipeline steps, calling the produce methods, and storing the values on the environment. It is important to notice that we can iterate over the pipeline steps in order since the pipeline language defined does not allow to put steps that do not have specified inputs before they are generated.

2.3 Towards Better Evaluation of AutoML Systems

AutoML systems should be evaluated thoroughly as machine learning algorithms. We identified critical AutoML evaluation best practices, including 1) separating datasets used for system development from those used for evaluation, 2) using the same set of ML building blocks (primitives) systems use, and 3) standardizing a common ML program description language (pipeline language).

Evaluating machine learning algorithms relies on reporting the score on the test data, which is different from the training set used to build the algorithm's internal state. When ML algorithms are evaluated, a common practice is to report its score on the test data, different from the training

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

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

Figure 2.3: Pseudo-code describing execution semantics of the reference runtime. The fit semantic in which the runtime iterates over all the pipeline steps and passes the data around to fit the sub-pipelines or primitives. The produce semantic in which the runtime passes the data through all the pipeline steps to produce new values forwarded to the next steps.

data used to build the algorithm's internal state. Similarly, AutoML system evaluators should split data, which for AutoML systems is a dataset of datasets. Any dataset used for system evaluation should not have already been used for system development and fine-tuning. This approach prevents AutoML systems, and also it can help demonstrate that systems can automatically generate solutions for unseen datasets. Ideally, evaluations would use k-fold cross-validation over datasets, but this is not always possible because it can be computationally too expensive to build AutoML systems multiple times, and not all AutoML systems can be trained. As an alternative, evaluators can use blind datasets not known to authors of AutoML systems in advance.

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

ML programs produced by systems should be described in a standardized language. The standardization of a language ensures that systems generate ML programs that contain only the ML building blocks permitted by the evaluators, without extra glue code that could leak into final ML programs. The use of a standard language promotes reproducibility. It allows evaluations where AutoML systems produce only ML programs, not predictions, so those programs can be trained and scored outside of systems, assuring equal access to ML building blocks and other resources.

In order to have a more objective evaluation of AutoML systems, we decided to split the valuation into three stages (Figure 2.4) Offline training, search, and scoring.

In the first stage, the offline training, a set of datasets is given to the performers. These datasets can be used to either design an AutoML system around them, use them for training and AutoML system, or else. The dataset set should contain at least an example of a task that will be asked on the search; for example, if an object detection dataset is not provided during training, it should not be used for searching. The second part of this component consists of the AutoML system. The system should use as inputs a dataset and a problem description and should output any number of ML pipelines. As previously mentioned, AutoML systems should be using the same building blocks, but for evaluation, it does not matter if they are using them internally, and the only part where they come in handy is for describing the ML pipelines. AutoML systems are allowed to use different frameworks, techniques, or optimizations to do the search. They can use other primitives that are not within the D3M ecosystem but have to generate pipelines that the ecosystem can run. For example, it is known that some D3M wrapped primitives from the Sklearn libraries are sometimes slower

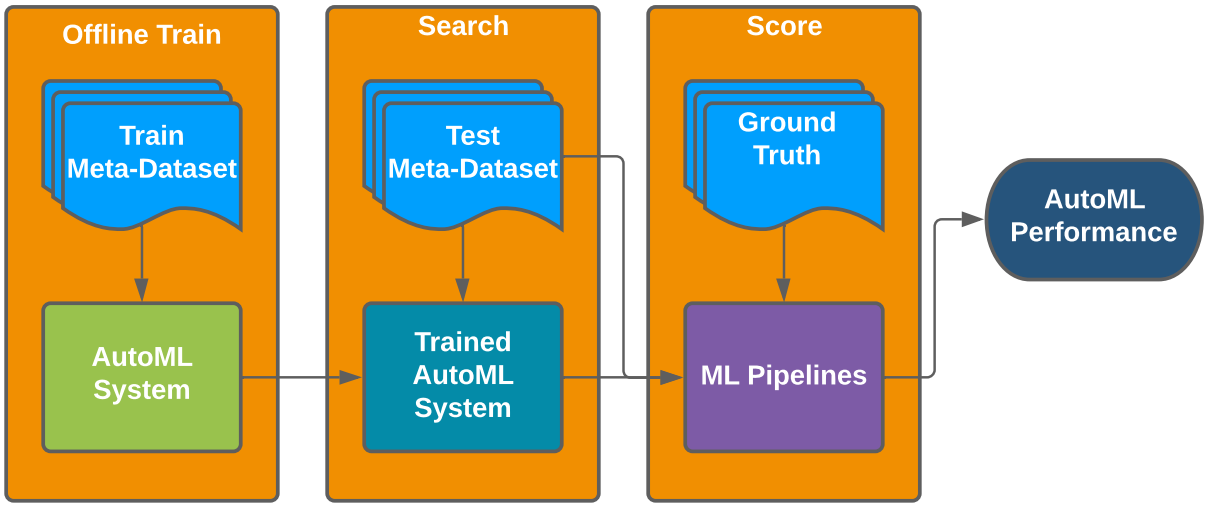


Figure 2.4: Evaluation Framework for AutoML Systems. The framework is divided into three stages, offline training, search and score. During the offline training, some data is available that can be used as a reference to train or create AutoML systems. On the search, the AutoML system must generate machine learning pipelines within some budget such as time, memory, etc. In the scoring face, the pipelines generated by the systems are independently trained and evaluated so AutoML systems can be compared based on their results.

than the pure Sklearn version due to the handling of metadata, so if a system is designed with this in mind, they could potentially get a speed up to internally evaluate candidate pipelines, and they should be able to represent them on the Pipeline language since there are mirror components.

The second stage, searching, is similar to the first stage in the sense that a set of datasets is given to generate ML pipelines with the already trained AutoML systems. The test dataset set could be a blind set from the training datasets or a completely unseen dataset, as long as the datasets contain the same datatypes used on the training set and the same tasks. Then the trained AutoML systems are going to be iterating over all test datasets to generate ML pipelines. For this searching process, the evaluator needs to be providing the AutoML systems with information regarding the amount of time to do the search, the computation resources such as the number of CPUs, GPUs, memory RAM, storage space allowed for each problem, the target metric to optimize, etc. The same set of parameters and machines needs to be used to provide a more fair evaluation. The AutoML systems, in this stage, are allowed to use whatever techniques and optimization, but they do not have access to

anything else just to the current dataset that is searching pipelines for; this is to reduce the different possible variations and advantages that a system could have. For example, a system could potentially have access to the internet to search for similar datasets to do data augmentation; However, this is a valid approach; this provides an overstanding advantage for such a system. Similar approaches are valid as long as the system is self-contained,i.e., the system has a local database where it can search for similar datasets.

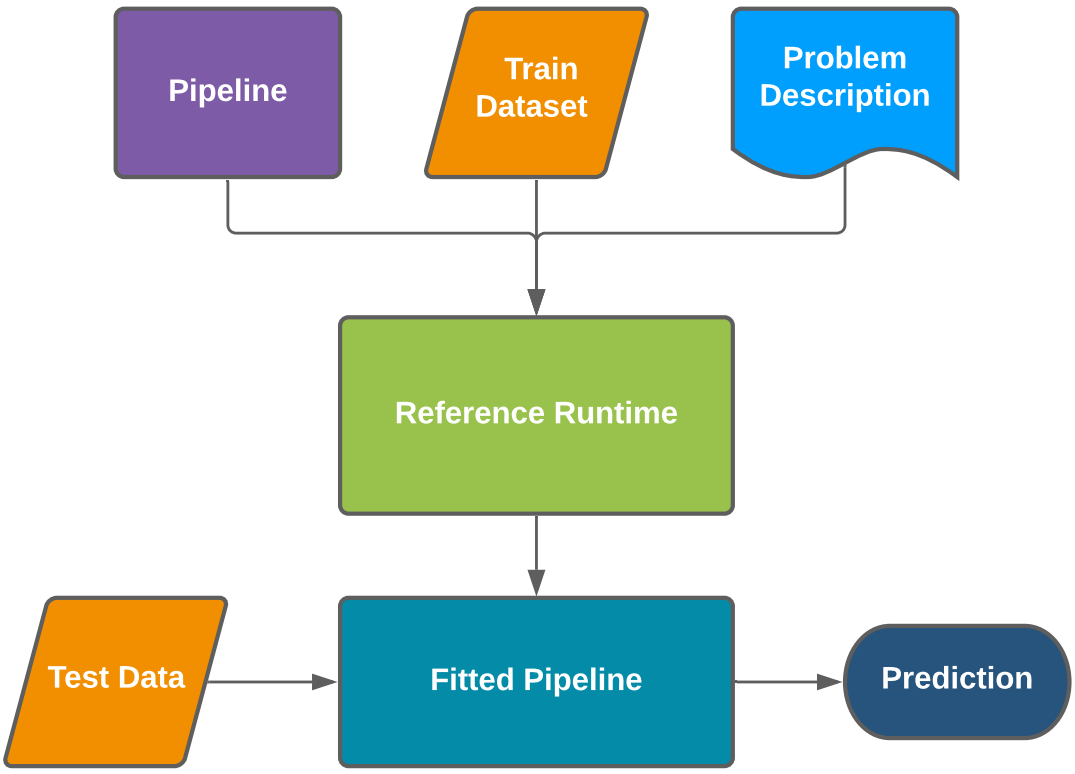


Figure 2.5: Reference Runtime as an evaluation tool. The reference runtime facilitates pipelines evaluation by fitting them and producing predictions. The reference runtime generates a fitted pipeline that can later be used with the test data and produce predictions to be later evaluated. To achieve this, the reference runtime requires information about the pipeline that will be used, a set of training data, and the problem description that contains information regarding the target column.

The final stage, scoring, focuses on gathering all the ML pipelines generated by the AutoML systems and evaluating them according to the metric of interest. In Figure 2.5 we show all the components needed to evaluate a pipeline. First, we need the ML Pipeline generated by the AutoML system; then, we use the training data provided during the search process and a problem description. The problem description, as its name states, is a collection of relevant information for the AutoML system that contains the task type, task subtype, metric of interest, and the target column identifier and instructions on how to evaluate the pipelines, such as K-fold cross-validation, train/test split, etc. Still, for our purpose, we focus on already split data. All of this information is passed to the reference runtime, and the ML pipeline is fitted with the input data, generating a fitted pipeline. Later on, the fitted pipeline is used to use the test data to generate predictions that are evaluated using the Groud Truth. After getting the scores of the pipelines, the evaluator could use different metrics to determine which AutoML system is better using different criteria, such as focusing only on single tasks, or which AutoML system generates better scoring pipelines, or just using the best pipeline as a reference point.

2.4 Experiments

The proposed framework was designed to support evaluations based on the best practices. As an example, eight AutoML systems have used our framework. An impartial third party organization (Data Machines) evaluated them using a suite of 106 available datasets and 62 blind datasets with AutoML systems with identical computing resources allocated. The datasets were prepared by Another 3rd party organization (MIT Lincoln Laboratory) and created their corresponding expert-made baseline solutions. The datasets represent 15 different task types. Overall, those datasets span a wide range of task types, including (i) classification, (ii) regression, (iii) semi-supervised classification, (iv) time-series forecasting, (v) graph matching, (vi) link prediction, (vii) collaborative filtering, (viii) community detection, (ix) object detection and (x) vertex classification. List of raw data types present include (i) tabular data, (ii) text, (iii) time-series, (iv) graphs, (v) images, (vi) audio, (vii) video. Table 2.3 shows a subset of known datasets.

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

Table 2.3: A subset of known datasets, their task type, and metric used. Achieved scores are not shown because scores on known datasets can be overfitted.

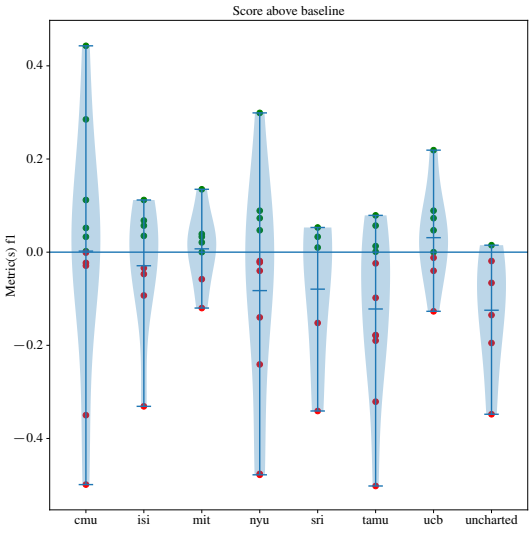
During evaluation or execution of ML pipelines via the reference runtime, the runtime can generate a pipeline run. A pipeline run consists of a series of data collection that tracks the execution of the pipeline and its steps. The information contains information that links the problem description, the dataset description, and the pipeline that is being executed. It also tracks information related to the execution time and memory usage of the methods that the primitives call and the metadata of the input data across the steps. All of this extra information is stored in JSON serializable format that can later be stored to be analyzed.

During the evaluation, all pipelines and pipeline run documents made during evaluation by all AutoML systems have been stored into the metalearning database as standardized documents for 168 datasets, 519 primitives, 160,999 pipelines, and 65,866 pipeline runs. Success rates are shown in Table 2.4. For this experiment, we consider a success, such as an AutoML system generating a runnable pipeline by the reference runtime capable of producing predictions. Non-tabular datasets, while supported by our framework, presented a more significant challenge to evaluated systems, leading to sparse results with only a few AutoML systems succeeding on a particular dataset.

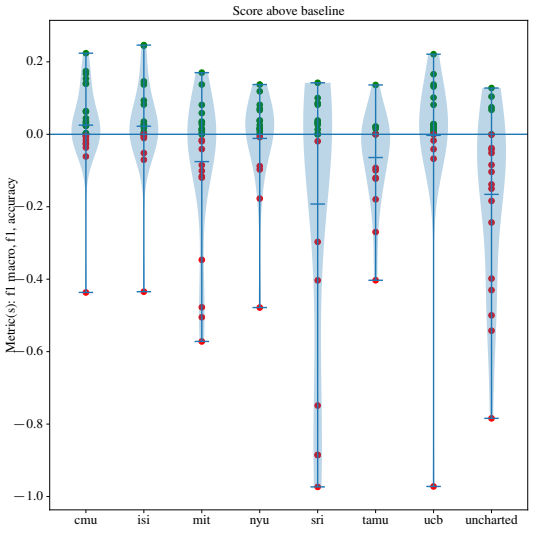
AutoML System	Known Datasets	Blind Datasets
CMU	93%	95%
ISI	92%	87%
MIT	92%	77%
NYU	91%	76%
SRI	76%	77%
TAMU	93%	74%
UCB	92%	87%
Uncharted	73%	55%

Table 2.4: Percentage of the 106 known datasets and 62 blind datasets for which each evaluated AutoML system successfully created a functional pipeline that can produce predictions using the reference runtime. Results show that the systems search method directly affects the performance since they all use the same building blocks.

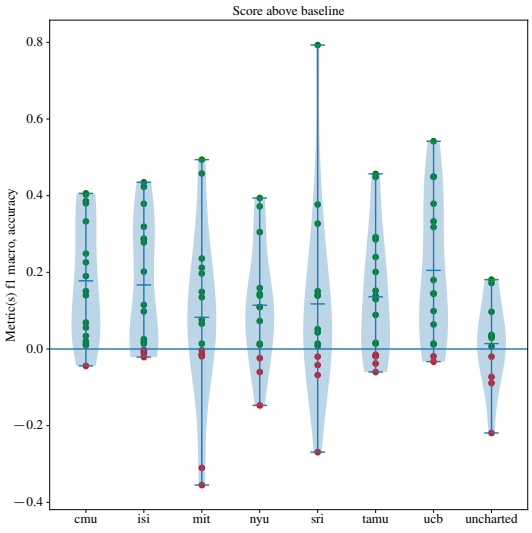
Figure 2.6 shows the results on tabular datasets and on some non-tabular datasets. Each dot on sub-figures of Figure 2.6 show how well system's best solution performed in comparison with the expert-made baseline solution. Because we are aggregating multiple datasets together, we show only relative distance from the baseline of each of those datasets. Higher is always better, for all metrics. Note as well that dots are shown only for datasets a system succeeded in producing pipelines for. We can observe the worse overall performance of AutoML systems on blind datasets. Not easily visible in Figure 2.6 AutoML systems have produced much less successful pipelines on blind datasets as well. Success rates are shown in Table 2.4. Non-tabular datasets, while supported by our framework, presented a greater challenge to evaluated systems, leading to sparse results with only few AutoML systems succeeding on a particular dataset.



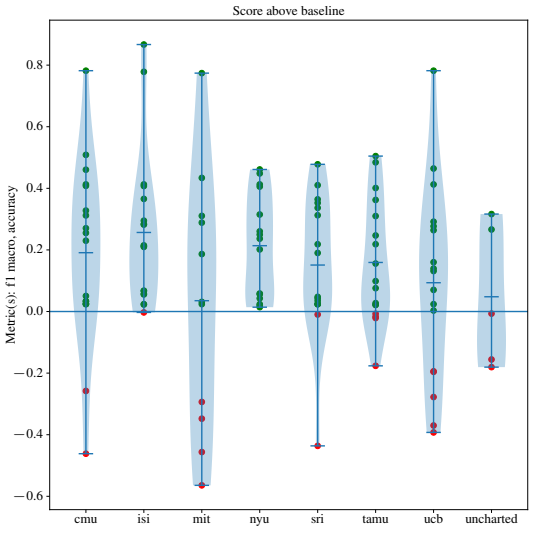
(a) Known tabular binary classification



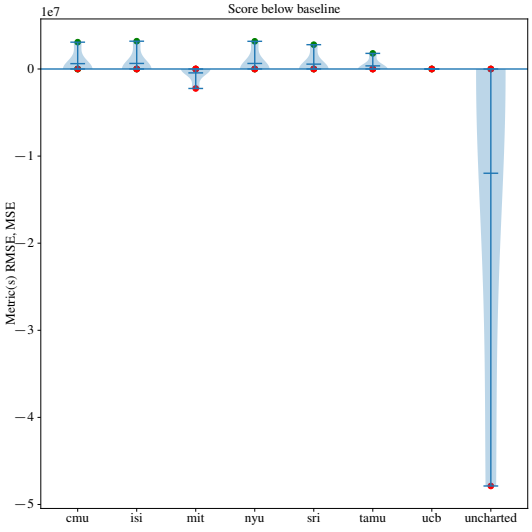
(b) Blind tabular binary classification



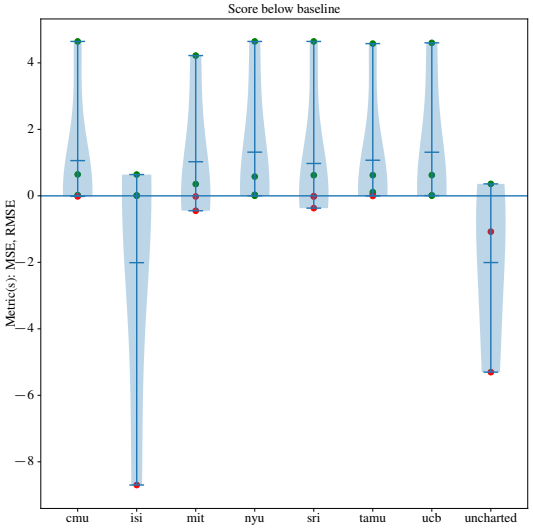
(c) Known tabular multi-class classification



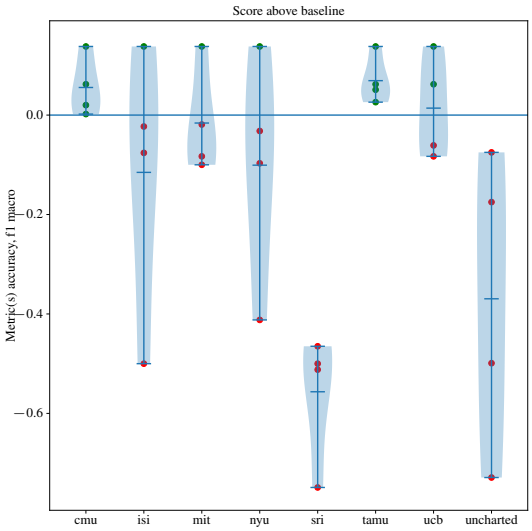
(d) Blind tabular multi-class classification



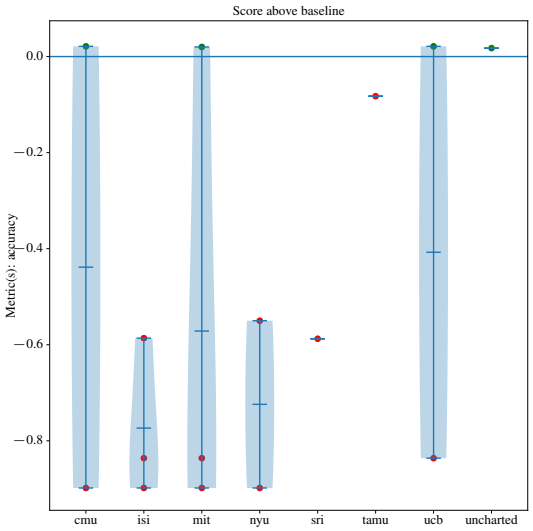
(e) Known tabular regression



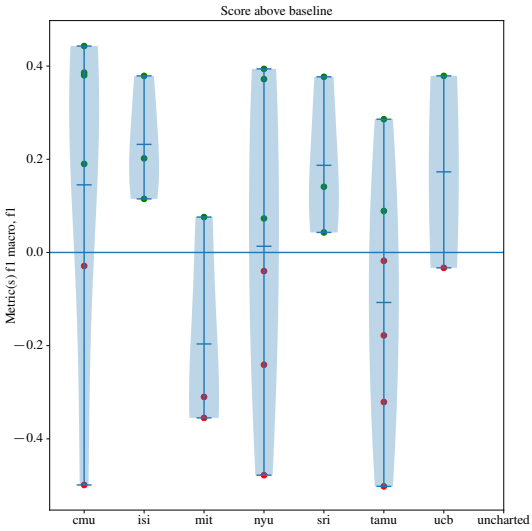
(f) Blind tabular regression



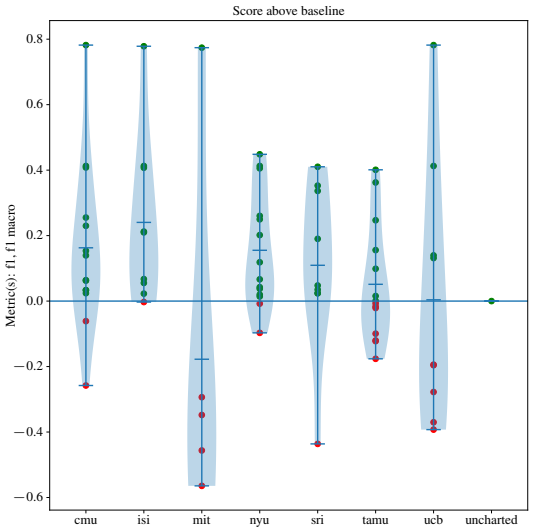
(g) Known graph vertex classification



(h) Blind graph vertex classification



(i) Known tabular semi-supervised



(j) Blind tabular semi-supervised

Figure 2.6: Evaluation results on known and blind datasets, aggregated for select task types. Note that some systems failed to produce any pipelines for some datasets, so individual plots may contain fewer data points. Higher is always better.

2.5 Conclusions

In this work, we focus on the standardization and evaluation aspect of AutoML systems and propose a general framework that enables a more fair comparison among systems. The proposed standard framework provides a set of tools that alleviate the need for AutoML systems of more information about the building blocks used. An advantage of using a standardized ML framework across ML systems is that various tools are available. A proposed tool, the Pipeline Profiler [53] visualizes and compares pipelines generated by different AutoML systems, which provides insight into the behavior of various components. Marvin [54] is an index of all current and historic primitives and provides a web interface to search for primitives by their metadata.TwoRavens wrapper [55] wraps AutoML systems to expose the same control interface to make the execution of AutoML systems uniform. The metalearning database service[56] is a publicly accessible python Flask-based app backed by an Elasticsearch database that enables users and systems to search and upload data. Currently, the metalearning database contains about 202 datasets, 519 primitives used in the most recent evaluation, 160999 pipelines, and 65866 pipeline run documents. By conducting experiments on various real-world recommendation tasks, we empirically validate the effectiveness and efficiency of our proposed framework.

3. Axolotl System: Modular Framework for development and testing AutoML solutions

In this chapter, we propose a framework to enable data scientists to prototype different AutoML systems rapidly without spending too much time learning the base system. To achieve this, we designed the framework based on the D3M ecosystem (Chapter 2) and created tools that simplify users' interaction with the complex ecosystem.

3.1 Introduction

AutoML solutions aim to solve a wide range of problems, from tabular classification to more complex problems such as graph matching or video recognition. Although AutoML systems provide new users simple interfaces for running out-of-the-box, the systems lack adaptability and extensibility for new problems or approaches. Most of the systems are very complex pieces of code that do not allow users to customize, such as creating different search methods easily.

For example, AutoSklearn [5] one of the most popular AutoML frameworks, it lacks extensibility. If a user would like only to use the framework to search for the pre-processing pipeline, make a model selection, and avoid hyperparameter tuning, they would need to re-wrap all the primitive search space since their code creates the search space considering all hyper-parameters since the beginning. The engineering effort to achieve such an easy task (search without hyperparameter tunning) depends on the user's familiarity with the whole framework. Another example of the over-engineering aspects of designing an AutoML system is the re-usability. AutoML solutions such as PMF-AutoML [13] and OBOE [31] share the same pipeline representation and search space, as well as the one defined on AutoSklearn. In essence, the framework they use is almost identical, and the search method is different, but they re-implemented all over again in their work.

Developing AutoML systems from scratch is challenging since the developers must consider many aspects, such as defining a pipeline representation, their execution, the search method, and other things. In chapter 2, we presented a work for standardize AutoML. Although all of the code is available for system developers to use, it has a steep learning curve. The framework provides

many benefits at a high engineering cost. In order to build a better AutoMl system, it is necessary to overcome the next challenges: (1) Use of D3M standards and reduce the engineering overhead; (2) Ease of applicability: competitive performance and ease to use for new users; (3) Extensibility: a modular system that allows easy customization of every system's components.

3.2 Framework

To cope with the challenges, we propose the AutoML framework Axolotl [1] which aims to provide tools for easily creating specific purpose AutoML solutions, as well as provide a competitive performance to other AutoML solutions. Axolotl framework (Figure 3.1) is divided into multiple components; the two main components are the search algorithms and the backend. The search algorithm is designed to implement their different search methods and interact with all the other system parts with minimum effort. Search algorithms, in general, try to generate pipelines and then 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 a time and a backend implementation that uses Ray [57] 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. In the following sections, we will discuss the aspects of each component's design.

[1] `https://gitlab.com/axolotl1/axolotl`

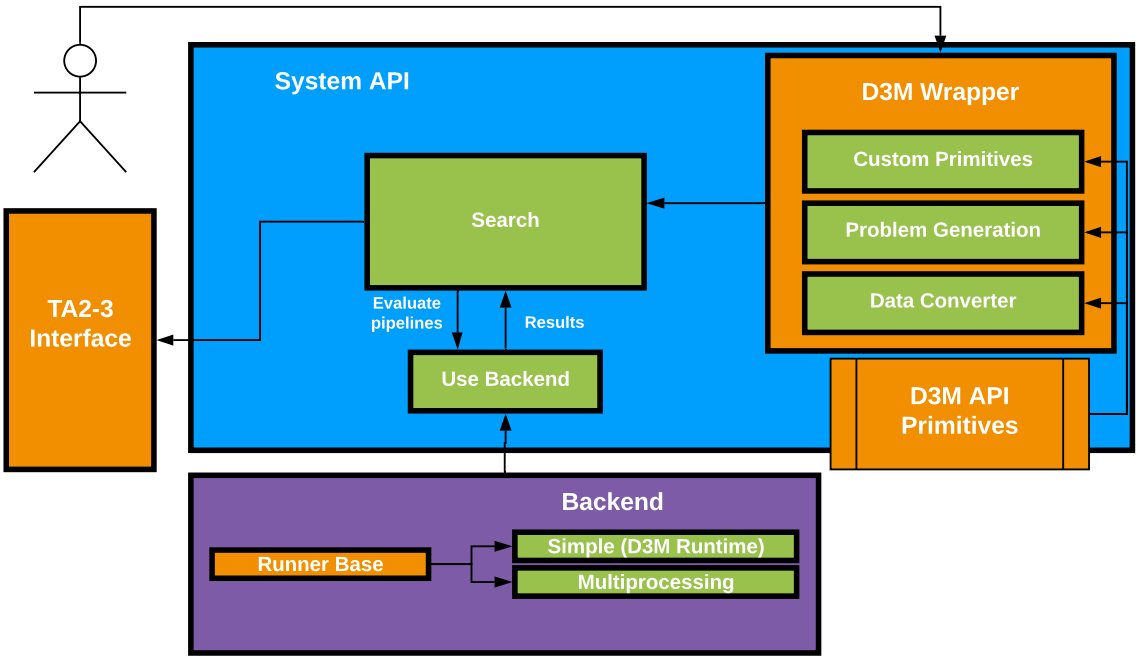


Figure 3.1: AutoML System Axolotl Architecture: System API: It is divided into two main components, the D3M wrapper, which aims to simplify user interactions with the AutoML framework, and the Search, which is in charge of generating pipelines to be produced by the system. Backend: Deals with running and evaluating pipelines generated by the Search.

3.2.1 Simplification of D3M Ecosystem

The D3M ecosystem (Chapter 2) provides the tools to create AutoML systems that can be compared among each other in a more efficient fashion. Unfortunately, the API D3M-Core Package [2]) and the Collection of D3M ML Primitives [3]) are very complicated pieces of software that has a very steep learning curve for on-coming users. Although the D3M ecosystem has a large collection of documentation and examples, it lacks user-friendliness since it was designed to facilitate machines using it. In this work, we focus on simplifying diverse aspects of D3M: unwrapping d3m primitive to a simplified interface, generation of D3M pipelines, and importing datasets and tasks.

[2] `https://gitlab.com/datadrivendiscovery/d3m`
[3] `https://gitlab.com/datadrivendiscovery/primitives`

Simplification of Primitives. Machine Learning primitives have been standardized in a human-readable form; for example, the classification primitive RandomForest by Sklearn contains an explanation for what each hyperparameter represents on the documentation, and to use it, a user need to follow the steps of instantiating the primitive by stating the desired hyperparameter on the constructor, then fit it and finally generate predictions by calling the method to predict (Figure 3.2).

```
from sklearn.ensemble import RandomForestClassifier

# Create a random forest instance.
primitive_instance = RandomForestClassifier(n_estimators=10)

# Train the ML primitive.
primitive_instance.fit(attributes, targets)

# Generate predictions.
predictions = primitive_instance.predict(attributes)
```

Figure 3.2: Example of a Sklearn Primitive usage. The primitive can directly be used from the import, then assign the hyperparameters on the constructor and use the interface fit to train the ML primitive and use it with the predict interface to generate predictions.

Later on, the D3M standard was written, which provided more machine-readable information that defined the hyperparameter space and added more standard functions definitions that allowed primitive designers to create new functions that AutoML systems can automatically use without further modifications. The hyperparameter space is defined by implementing the hyperparameter class that contains the possible and default values for each hyperparameter; this allows AutoML systems to query this information and make decisions to tune them during their search. This process facilitates the use of primitive automated systems. Still, it makes it difficult for humans to use; for example, in Figure 3.3 we have all the operations needed to have similar results as the

Sklearn interface. First, the primitive class must be retrieved through the D3M index that indexes all the d3m compatible primitives. The hyperparameter class needs to be obtained by accessing the metadata of the primitive, then getting all the default values of the hyperparameters and overriding the one of interest. After all this, the primitive class can be instantiated by passing the modified hyperparameter instance to the constructor. The training data needs to be set on the primitive instance via the *set_training_data* method to be used later to train the primitive by using the *fit* method and later generate predictions by calling the *produce* method. All of this process requires some understanding of the D3M ecosystem, in which, in most cases, it should not be necessary to expose it to the end-users. The end-users are more likely to fit and train an ML primitive to check how the data changes, and by using this API, things become more complicated since they need to know how primitives and metadata are built.

We designed a primitive simplification mechanism to tackle this challenge, allowing users to experience the primitives as simple as a higher API. This primitive simplification is achieved by exploiting the Python API that allows dynamic object creation. Users mostly create Python classes by statically describing their variables and methods and rarely adding more during execution. These classes follow common programming patterns that state that all methods and variables should be known before execution so everything can be tacked during debugging. One of the most common instances where dynamic classes can be seen on Python is when users manually attach a new variable to the instance; this leads to missing documentation. They can be problematic later since Python is not a strong typing language, and users or other methods can easily modify the variable a lose track of its content. By taking into consideration these issues, we decided to create dynamic classes (not regular classes with attached methods) that are built during runtime and are fully documented.

The approach of creating dynamic classes on the fly allows us to generate simple primitives for all primitives on the D3M ecosystem without having to insert methods or variables manually. To achieve this, we created a base class that invokes the automatic insertion of methods when called and returning a new class. First, the class generators create a new class using the *BaseEasyPrimitive* class that later on inserts all the methods of interest, such as *set_training_data fit* and the different

```python
from d3m.index import get_primitive

# Retrieve D3M primitive class.
RandomForestPrimitive = get_primitive(
    'd3m.primitives.classification.random_forest.Common'
)

# Hyperparameter handling.
hyperparams_class =
 ↪  RandomForestPrimitive.metadata.get_hyperparams()
default_hyperparameters = hyperparams_class.defaults()
new_hyperparameters =
 ↪  default_hyperparameters.replace({'n_estimators': 10})

# Create a random forest instance.
primitive_instance =
 ↪  RandomForestPrimitive(hyperparams=new_hyperparameters)

# set_training_data and fit are the equivalent to fit on sklearn.
primitive.set_training_data(inputs=attributes, outputs=targets)
primitive.fit()

# Generate prediciton.
predictions = primitive.produce(inputs=attributes).value
```

Figure 3.3: Example of a D3M Random Forest Primitive usage. The primitive needs to be retrieved from the D3M index, and then the Hyperparameter class needs to be obtained to get the default values. Later the primitive can be instantiated via the constructor by passing the hyperparameters. The primitive instance can later be trained by setting the input data via set_training_data and later fit. Finally, the primitive can generate predictions via the produce method.

produce. The base class's constructor is in charge of wrapping and handling all the interactions with the hyperparameters that allow users to specify the ones of interest and the function parameters. The methods to expose such as *fit* and *produce* are later introduced by creating special function handlers. These handlers are created via code in string format that is updated according to the different methods to expose and later converted to python executable code. This approach allows the creation of templates for the standard D3M methods and leaves room for users to add their custom methods inside the dynamic class. The metadata is inspected at every step of creating the new simple primitive. The information regarding the hyperparameters, parameters, and methods

is retrieved to dynamically create documentation that can be accessed via IDEs or python help commands. Automatically generating the documentation allows users to inspect the documentation quickly to understand the primitive better.

An example of using the simple primitive wrapper can be found in Figure 3.4. For this example, the user can directly generate the primitive by using the same python path used on D3M. After the primitive is materialized, it can be used as the Sklearn example by creating an instance, stating the hyperparameters on the constructor, training it, and later generating predictions. This approach shows that the D3M primitives interface can be simplified to alleviate the overhead of classes and provide a friendly interface to the end-user.

```
from axolotl.simple.d3m_primitive_unwrapper import
    ↪ SimplifyPrimitive

# Retrieve D3M primitive class and create a dynamic unwrapped
    ↪ class.
RandomForestPrimitive = SimplifyPrimitive.from_primitive_path(
    'd3m.primitives.classification.random_forest.Common'
)

# Create a random forest instance and pass new hyperparameters.
primitive_instance = RandomForestPrimitive(n_estimators=10)
primitive_instance.fit(inputs=attributes, outputs=targets)

# Generate prediciton.
primitive_instance.produce(inputs=attributes)
```

Figure 3.4: Example of simplified D3M Random Forest Primitive usage. A dynamic class is built during execution by stating the python path of the primitive to use as base. Then the dynamic class can be use as a regular ML Primitive by stating the parameters and hyperparameters directly on the methods.

Simplification of Pipelines Constructions The D3M Pipeline language allows flexibility with the Direct Acyclic Graph approach to cover many pipeline configurations. To achieve this, it is necessary to create the pipeline one step at a time, adding the nodes of the pipeline (primitive or sub-pipelines) and then defining the edges (inputs and outputs). Also, since the pipelines are descriptions of Machine Learning programs, it is crucial to define the parameters and hyperparameters for each step. Figure 3.6 shows an example of how to add a primitive step to a Pipeline using the D3M API. First, an empty pipeline needs to be created and add an input for the original data. Then, it is necessary to retrieve the primitive class that is going to be added and create a PrimitiveStep instance by using the primitive class and a resolver (use to resolve the primitive path if needed). For this example, the primitive *DatasetToDataframe* helps to transform a D3M dataset to a D3M Dataframe; the primitive only takes one input that is the original dataset. After creating the primitive step, the inputs of the primitives are added (edges) by specifying the parameter's name, the argument type, and the data reference; if the primitive takes more parameters, this step needs to be repeated as many times as possible parameters. Adding hyperparameters for the primitive step follows the same logic as adding parameters; on the example, a hyperparameter named *dataframe_resource* with the value *None* is added to the primitive step. A thing to notice is that *ArgumentType* has different values according to data type; for example, during adding arguments most of the time, it could be *CONTAINER* since it's referencing to a data container, while on the hyperparameters is more likely to refer to *VALUE*.

Regular primitives can only be used under a single semantic: apply the transformation or operation to the whole input data. D3M Primitives have different execution semantics that adds more complexity to the users. The most common execution semantics on D3M are apply to the whole data, use columns by index, or use semantic types.

- **Use all input data** semantic follows the same logic as regular ML primitives, where the operation defined by the primitive is applied to the whole input data.
- **Use columns by index** semantic allows a more granular control in how the operations are applied to specific columns that are defined on the control parameters.

```
from d3m.index import get_primitive
from d3m.metadata.base import ArgumentType
from d3m.metadata.pipeline import Pipeline, PrimitiveStep,
    ↪  Resolver

# Create a new Pipeline and add inputs
pipeline = Pipeline()
pipeline.add_input('input_data')
# Retrieve D3M primitive class.
DatasetToDataframe = get_primitive(
    'd3m.primitives.data_transformation' +\
    '.dataset_to_dataframe.Common'
)
# Create a primitive step.
primitive_step = PrimitiveStep(
    primitive=DatasetToDataframe, resolver=Resolver()
)
# Add inputs to the primitive step.
primitive_step.add_argument(
    name=inputs,
    argument_type=ArgumentType.CONTAINER,
    data_reference='inputs.0'
)
# Add hyperparameters one by one.
primitive_step.add_hyperparameter(
    name=dataframe_resource,
    argument_type=ArgumentType.VALUE,
    data=None
)
# Add method that is going to be use to generate the output.
primitive_step.add_output('produce')
# Add primitive step to the pipeline.
pipeline.add_step(primitive_step)
```

Figure 3.5: Example of adding a Primitive to a D3M Pipeline. The primitive needs to be retrieved from the index to be later used to instantiate a PrimitiveStep. After instantiating the primitive step, the arguments and hyperparameters are added one by one, and later an output method is added referencing the desired produce method. Finally, the primitive step is added to the pipeline.

- **Use semantic types** semantic similar to use columns by index, enables a higher level where the columns to apply can be queried by semantic types defined on D3M API, such as applying the operation to specific values or choosing to apply it to all the features.

```
from axolotl.simple.primitives import PrimitiveHandler

# Create a new Pipeline and add inputs
pipeline = Pipeline()
pipeline.add_input('input_data')

# Create a primitive handler instance
primitive_handler = PrimitiveHandler(
    primitive='d3m.primitives.data_transformation' +\
              '.dataset_to_dataframe.Common',
    hyperparams={'dataframe_resource': None},
    resolver=Resolver()
)

# Add primitive to pipeline
primitive_handler.add_produce(
    [], pipeline, arguments_references={'inputs': 'inputs.0'}
)
```

Figure 3.6: Example of adding a Primitive to a D3M Pipeline via Axolotl Primitive Handler simplification. A primitive handler is created by defining the desired primitive and hyperparameters to be later added to the pipeline by calling the add_produce method and stating the arguments references.

Although there are multiple semantic executions on D3M Primitives, it is not standardized across all of them, making it difficult for new users to determine whether or not the primitives can handle them. For this reason, we designed our primitive handler for pipelines, considering the flexibility the different semantics provide. We develop a primitive handle that allows users to add primitives by simplifying most unnecessary code calls and allowing them to specify any of the three semantic executions even without the primitives implementing them.

To achieve this, we mapped all the different semantic executions to use columns by index. We are allowed to do this since the use of semantic types operation relied on iterative query all the features' metadata and check if they share the semantic type of interest. Instead, we do all this process on the handler and split the input data into columns of interest and columns that should not change. Then, we apply the desired primitive to only columns of interest and concatenate the results with the columns that did not change. While doing this process, the necessary primitive steps to

split and concatenate the columns are automatically added to the pipeline so the user does not have to handle it.

Besides automatically supporting the missing semantic executions, we also simplified the interface to add primitives to pipelines. In Figure 3.6 an example of adding a primitive with its parameters and hyperparameters can be found. We simplify different aspects of the interface by removing redundant calls such as *add_argument* and *add_hyperparameter*, so it can support defining all of the information at once. We also removed the overhead of the user to determine the data type of the parameters. In general, with the simplification, the user follows the steps of creating the pipeline, then creates an instance of the Primitive handler that takes as parameters, the primitive that will be added, and the hyperparameters. Later, the method *add_produce* needs to be called by specifying the pipeline where the primitives will be added and the inputs references. The previous method also provides a way for the user to directly execute the primitive with extended semantics by adding the data instead of the references.

3.2.2 Pipelines Evaluation via Backend

The backend is in charge of running different queries related to D3M pipelines search, such as running pipelines, exposing their intermediate results, and evaluating them. It is one of the most important components since the heavy computational part relies on it. We decided to abstract the heaviest computational part as an independent component since different approaches could be implemented, such as caching all the results to avoid computing again later with new pipelines or even one that allows lazy evaluation of pipelines. A base class was created based on this abstraction since it is necessary to cover few specific requirements 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. However, it uses Ray to run pipelines in parallel, allowing scaling pipeline search algorithms running in clusters or small servers.

The backend is designed around the idea that the pipeline search component would generate a set of pipelines that are sent to the backend to be evaluated according to the criteria defined on the search. To generalize the approach, we designed a base class called *RunnerBase* that abstracts all the necessary functionality for backends to be compatible with other parts of the framework. Overall, the base class defines a few abstract methods to be implemented; these methods are defined as requests. Request methods are designed with an asynchronous paradigm, so different methods of parallelizations can be supported. We support vertical and horizontal parallelization among the different ways to speed up the evaluation of pipelines. Vertical parallelization in our context means that pipelines are offered all the available resources so every primitive (if supported) can be trained and produce predictions on parallel. In comparison, horizontal parallelization refers to splitting the computational resources so multiple pipelines can be evaluated simultaneously.

Reference Runtime Backend. The first implementation of the backend purely uses the D3M Reference Runtime. This implementation allows pipelines to be fitted, generate predictions, and evaluate them. We achieve this by extending the methods defined on the reference runtime and automatically generating the required extra information. The reference runtime backend allows the vertical parallelization of pipelines by automatically modifying the parameters of the primitive if they support it. In Figure 3.7 the flow diagram is presented. The search algorithm generates a set of pipelines to evaluate and provides the evaluation definition for the pipelines. The evaluation definition contains information regarding how to evaluate the pipelines, such as splitting the data, doing cross-fold validation, and the metrics to use to evaluate the performance. After the search component gathers the required information, the information is forwarded to the reference runtime to evaluate the pipelines via the interface defined on the base class. Then, access the dataset defined on how to evaluate the pipelines, and the runtime evaluates one pipeline at a time; if multiple pipelines are required to be evaluated, the reference runtime will iterate all over them. After this,

the results are returned to the search algorithm, and the process can be repeated.

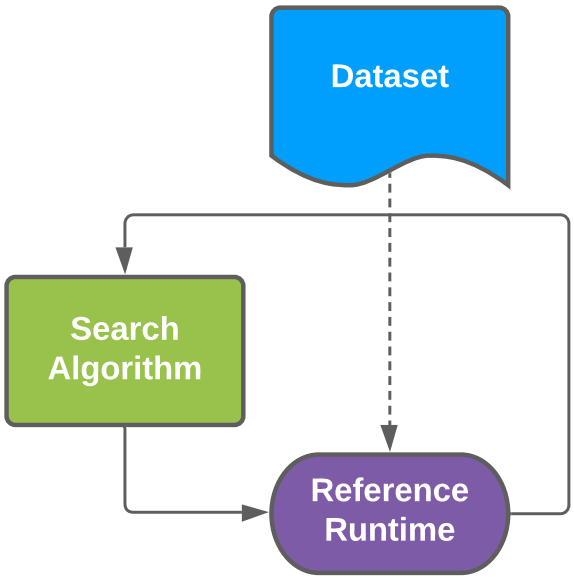


Figure 3.7: Reference Runtime Backend. In order to interact the search with the reference runtime, the search generates several pipeline candidates and defines the evaluation metrics and the dataset to use. Then the Reference Runtime evaluates one pipeline at a time and forwards the results back to the search.

Parallel Backend with Reference Runtime and Ray. The parallel implementation of the backend allows evaluating multiple pipelines simultaneously by dividing the computer resources among multiple pipeline runners. In Figure 3.8 the flow diagram is presented. This approach allocates all information needed for the runners on shared storage. The shared storage contains a queue of pipelines to be evaluated that the search algorithm populates. Then the data used to evaluate the pipelines is shared across the runtime workers by hashing it to be identified easily. After all the data is accessible to the workers, the runtime workers take a pipeline from the shared queue and evaluate the pipeline. Later, they return the results to the shared storage, and the search algorithm can read

the results so later can allocate more pipelines to be evaluated. There are several challenges when evaluating pipelines on parallel related to the data size; for this reason, we developed a data worker in charge of handling data across the runtime workers instead of workers directly reading it.

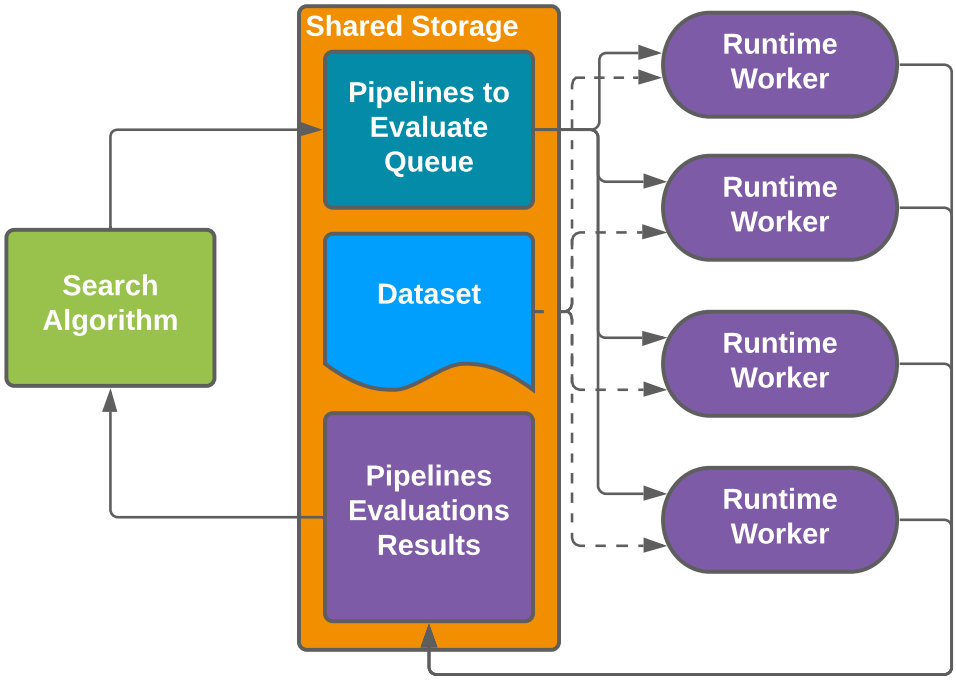


Figure 3.8: Parallel Backend with Reference Runtime and Ray. The parallel backend using Ray follows the same interface as the reference runtime with the difference that the parallelization is horizontal, allowing multiple pipelines to be evaluated simultaneously. Several components are added to the shared storage, such as the dataset, a queue containing the pipelines to evaluate, and a queue containing the evaluation results. Then, multiple runtime workers access the shared storage, evaluate a pipeline on the queue, and return the evaluation results. The process is repeated until no more pipelines are allocated to the evaluation queue.

3.2.3 Search Algorithms

Axolotl provides a simple API for developers to create and test custom search algorithms without learning or interacting with other parts of the system. We created the *PipelineSearchBase* class that abstracts all the interaction with the other components of the system. To create a new search algorithm, the user needs to inherit from the base class and implement a single abstract method called *_search*. The method only takes as a parameter the time left for the search. The main search method will iteratively call the *_search* method and provide the remaining time for the search.

The arguments for the search algorithms contain a problem description with information regarding the task to solve, the backend to be used, and possibly a ranking function to rank evaluated pipelines. Later, during the search, it is necessary to specify the dataset to use to evaluate the pipelines and the time constraints. The use case of specifying the dataset during the search and not during the construction is that users can potentially modify the data, such as reducing or increasing the number of observations. An example of a simple search algorithm that uses previously generated pipelines can be found in Appendix B.

The search component on the systems provides two base classes, the *PipelineSearchBase* that it was previously described and the *TunnerBase*. Also, Axolotl provides multiple search algorithms (Figure 3.9) that allow users to create new ones by reusing existent components. We designed four search methods: Random Search, Bayesian Search, Dummy Search, and Data-Driven Search. Random and Bayesian Search focuses on hyperparameter tuning, while Dummy and Data-Driven search focuses on pipeline generation.

Random Hyperparameter Search. This search method focuses on performing hyperparameter tuning for a given set of pipelines. The method takes a set of pipelines as input and later generates the hyperparameter space by querying the information from the metadata of every primitive on the pipeline. Then, it proceeds to perform random sampling on the hyperparameter space and generate new pipelines that copy the original ones but update the hyperparameters. These pipelines are later evaluated, and the sampled configurations are removed from the space to avoid repeating the same configurations.

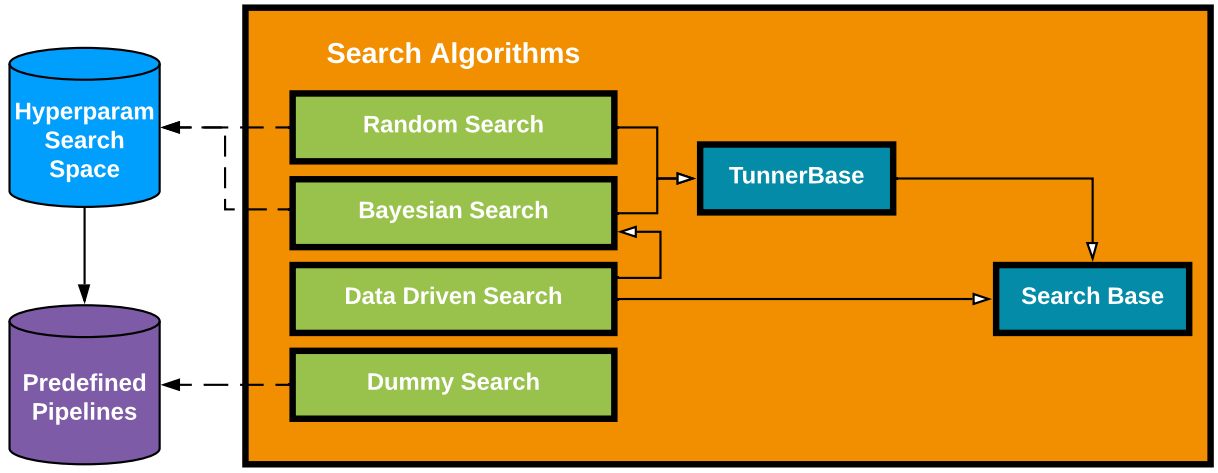


Figure 3.9: Axolotl's search: Search algorithms must implement the required methods from the Search Base class to be compatible with the other parts of the system. Tunner base class helps to define classes for hyperparameter tuning.

Bayesian Hyperparameter Search. This search method uses Bayesian optimization tuning the hyperparameters of a given set of pipelines. The method uses an Oracle that extends the KerasTuner[58] by mapping the D3M hyperparameter space. The oracle is in charge of transforming categorical hyperparameters to numerical by using One Hot Encoding. After the hyperparameter space is transformed into the continuous space, the oracle generates new hyperparameter configurations that are later evaluated. The results are used to train a Gaussian Process that is later used to sample new configurations. The method uses the input pipelines as part of the search space, so the search will return pipelines whose performance is better or equal to the original ones.

Dummy Search. This search provides valid pipelines for classification and regression problems for most tabular data with numerical and categorical values. The search uses a small collection of generic pipelines that were collected from previous AutoML systems evaluations that later are queried according to the task to solve and retrieve them. Then, the pipelines are evaluated, and those with higher performance are selected.

Data-Driven Search. We propose a Data-Driven approach for a 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 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 seen in Figure 3.10. The search strategy is divided into four steps: Data Prepossessing, Feature Selection, Model Selection, and Hyperparameter Tuning.

- **Data preprocessing.** The Data-Driven search strength comes from the data preprocessing search. Most AutoML systems have very limited available operations on the data before passing it to an ML learner. Such systems mainly focus on transforming the categorical data to numerical representation by performing some transformation like OneHotEncoding. Later, the systems add steps to scale the data, such as MinMax value, standard value, and maximum absolute value. The limitation is that it becomes challenging to automatically handle other data types such as images, audio, graph data, etc. Our approach is to extend the concept of *PrimitiveHandler* (Chapter 3.2.1) that allows users to add primitives to pipelines easily. We extended the *PrimitiveHandler* base class to several classes that are considered *SemanticHandlers*. SemanticHandlers, similar to PrimitiveHandlers tries to encapsulate and simplify the interactions with D3M API. Such handlers have been created to contain hand-made heuristics for data preprocessing to tackle different data types and can be directly be applied to the data or automatically added the needed steps to the pipeline to transform the data. The Semantic Handlers contains information related to what data type transforms, and when applied to data that does not contain such type, they are ignored. The main goal is to transform all the input data to a common numerical data frame representation.

  An example of the implementation of a SemanticHandler can be found in Appendix C. The handler is only applied to data with categorical semantic and is based on a heuristic that considers the cardinality of the data. Since handlers only affect the columns with specific semantics, there are some cases when the semantics of the data can change after applying it. For example, if the data contains categorical values and we apply the Categorical semantic handler, the output data will now contain numerical semantic.

Since semantics can change after handlers, we decided to create *DataSemanticHandlers* class. This class aims to provide multiple ways to execute semantic handlers, such as vertical and horizontal execution. The vertical execution refers to executing one semantic handler at a time so that the following ones will use the output. This approach allows a nested execution in the sense that if a handler transforms the data, the following will include that data. The horizontal execution splits the data into different semantics and feeds specific semantics to the handler so that the outputs can be concatenated later.

- **Feature Selection.** After the preprocessing pipeline is generated and the data has been transformed to a standard numerical representation, the search attempts to run every feature selection primitives on the library with a subset of the already transformed data. If it is successful, a new set of pipelines contains the successful primitives. By running the feature selection primitives on a data subset, we guarantee that we will be evaluating pipelines that are most likely to produce a score.

- **Model Selection.** In this step, the pipelines are completed by adding a Machine learner primitive. The machine learners' primitives are queried from the D3M library by considering the task and metric from the problem description. For example, the system will query primitives with terms, binary, and multi-class classification if a task is only specified as classification with accuracy metric. In some instances, problems are not well defined, such as semi-supervised classification, so the system will try to map to the closest task that will be classification. After a set of candidate machine learners primitives are retrieved, new pipelines are created by doing all the combinations of preprocessing pipelines and machine learner candidates. After all the pipelines to be evaluated are generated, the system will evaluate them and rank them according to their performance scores.

- **Hyperparameter Tuning.** This final step is only triggered if there is time left for the search. Hyperparameter optimization on this search is not essential and is only used to fine-tune the best pipelines. The intuition behind this is that the default values of hyperparameters provide a good reference of the overall performance of the pipeline, and the performance improvement should not be far [59]. If hyperparameter tuning is triggered (if there is time left to evaluate more pipelines), the system will select the *top-k* primitives based on performance and generate more pipelines candidates by tuning their hyperparameters by using the Bayesian Search. Pipelines with different hyperparameter configurations will be evaluated until there is no time left.

3.3 Conclusions

The work we discussed in this chapter focuses on the design of the Axolotl AutoML system that allows flexibility to different levels of users to do customization. The system tries to reduce the overhead caused by the Standard D3M API for AutoML by providing tools to developers that hide the complexity by using dynamic classes and handlers generated on Python. Axolotls AutoML system also provides search algorithms and techniques capable of solving an extensive collection of Machine Learning tasks such as univariate and multivariate regression, binary and multi-class classification, vertex classification, community detection, and collaborative filtering. The system can also handle a diverse collection of data types such as numerical, boolean, categorical, text, time series, image, audio, and graph. By conducting experiments on various real-world datasets, we empirically validate our proposed framework's effectiveness and validate the stability of our framework (see Chapter 2).

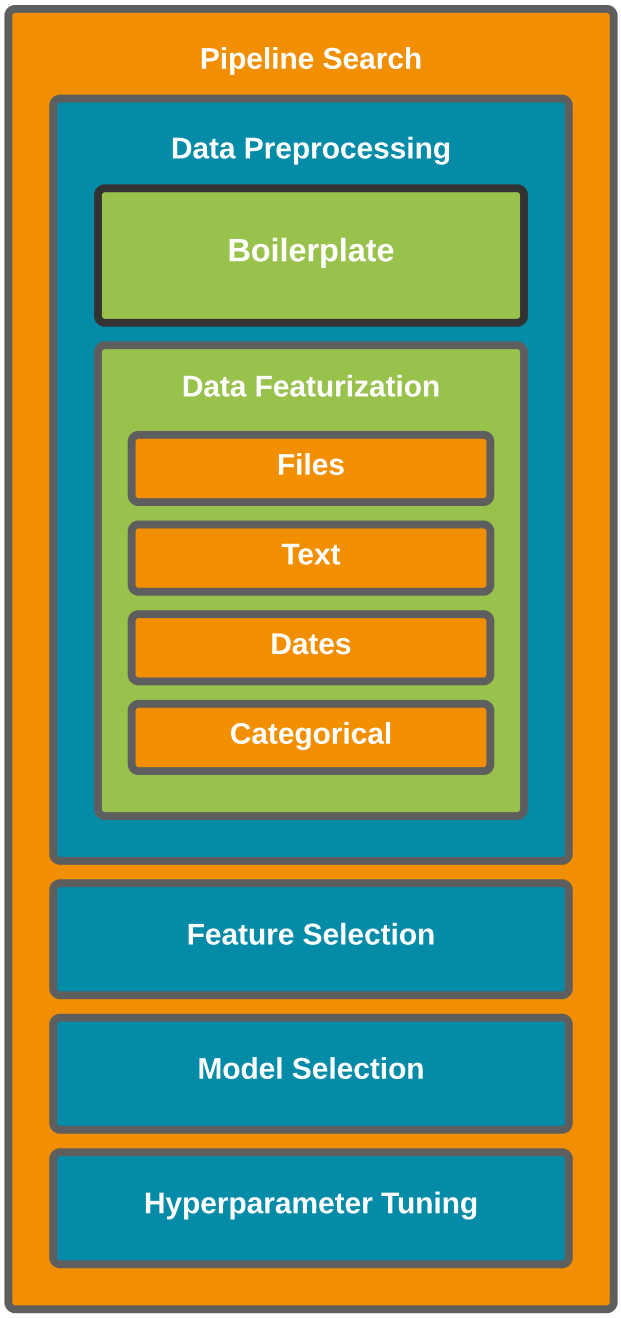


Figure 3.10: Axolotl's Data-Driven search algorithm consists of a mixture of different search strategies focusing on the pipeline's specific components. A heuristic that transforms the input data to a common numerical representation is used for the Data Preprocessing step, dealing with different data types such as files, text, dates, and categorical attributes. A brute force method is used for the feature and model selection. Finally, Bayesian optimization is used for the Hyperparameter tuning step.

4. Feature-wise Transformation Search for AutoML

AutoML systems methods mainly focus on model selection, ensemble models, and hyperparameter tuning leaving room for improvement on data preprocessing. In this chapter, we explore increasing the performance of machine learning pipelines by exploiting preprocessing pipeline search. We propose an algorithm that searches for feature-wise preprocessing pipelines instead of a single pipeline for all the data. The algorithm uses feature context similarity to search and propose pipelines to similar pipelines to similar features and reduce the search time. The algorithm's outcome is expected to be a set of preprocessing pipelines that boost the performance of a pipeline with a specific machine learning learner.

4.1 Introduction

Designing and implementing an AutoML system is challenging since it has many moving parts. The design focuses on the hypothesis to test, for example, a system that quickly evaluates more pipelines to cover a more extensive search space to a system that evaluates a few configurations thoroughly to get more accurate measurements to tune a prediction model. Despite differences in AutoML systems, most of them focus only on tasks related to tabular data that contains numerical and categorical values; this leads to systems following the same patterns when creating pipelines.

Most of the pipelines generated for AutoML systems follows the follow the following pattern preprocessing data and machine learning learner. On the preprocessing steps, systems attempt to transform the data to a numerical representation by applying data transformations. In most cases, these transformations are limited to applying some encoding to non-numerical values, imputing missing values, and normalizing them. Although the number of data transformation primitives is limited, the complexity of generating preprocessing from scratch could easily explode due to the number of possible combinations. To alleviate this problem, AutoML systems generally have a set of already defined preprocessing pipelines to use. This approach leads to a good overall performance since it relies on embedded human knowledge about how to tread different feature types.

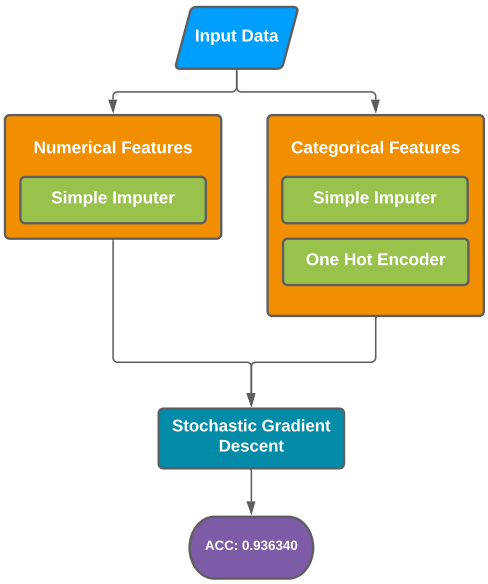
Motivated by the current limitations on the preprocessing pipelines, we expected to improve the performance of ML pipelines by generating more complex preprocessing pipelines structures. Transforming the data based on data types might not necessarily be optimal for all of the cases. For example, in Figure 4.1 we present a couple of pipelines that are solutions for Thyroid Disease dataset [60] [1]. The Thyroid Disease dataset contains integer, float, categorical and boolean features. A common solution from an AutoML system will be similar to the pipeline Figure 4.1a in which the system will apply One Hot Encoder to any non-numerical value, while a more elaborated solution (Figure 4.1b) will be applying specific transformation for different columns. Both of the pipelines were evaluated using ten cross-fold validation with the same splits. The most common pipeline achieved 93 percent accuracy, while the most complex one achieved 95 percent accuracy.

Current pipeline search is not a trivial task since the search space is extensive. We need to consider that AutoML systems alleviate this problem by creating templates or, in the case of preprocessing pipeline search, limit the available options. Given the ample search space, searching for preprocessing pipelines for all the input features could lead to an exponential grow of the search space. For example, for a dataset with seven features and seven different combinations of transformation, the search space would be over eighty-five million combinations, without even considering the hyperparameter space of each transformer. To tackle this challenge, we propose the use of a deep-clustering-driven search that searches for preprocessing pipelines for clusters of similar features instead of independently.

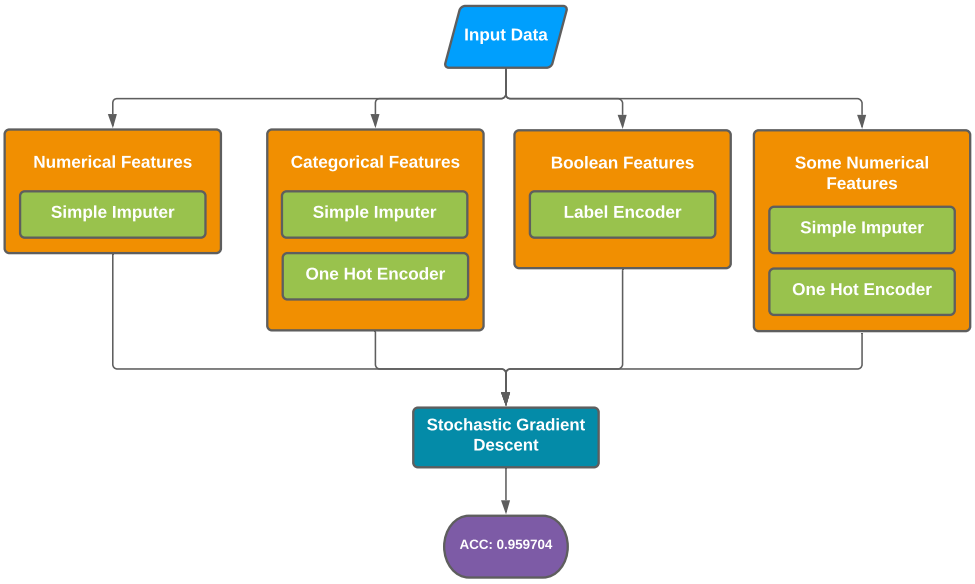
4.2 Preliminaries

AutoML algorithm's primary goal is to provide a machine learning pipeline that performs well given specific criteria and data. The pipelines are usually divided into two components: the preprocessing pipeline and the machine learning learner. Overall the pipeline search problem for AutoML could be defined as:

[1] `http://archive.ics.uci.edu/ml/datasets/thyroid+disease`



(a) Common Pipeline.



(b) Feature-wise Pipeline.

Figure 4.1: An illustration of different pipeline architectures and performance on the same dataset. Figure (a) is a standard pipeline that imputes missing values and applies One Hot Encoder transformation to non-numerical data. Figure (b) is a pipeline that handles numerical and categorical data similarly to the Common Pipeline but also applies One Hot Encoder transformation to some numerical data and uses Label Encoder transformation to some categorical data. The performance improvement is because some numerical data such as age has a limited range and can be seen as categories instead of numerical data, while some categorical values only contain a couple of options.

$$\arg\min_{\mathcal{M},\mathcal{P},\boldsymbol{\theta}_m,\boldsymbol{\theta}_p} \mathcal{L}\left(\mathcal{M}\left(\mathcal{P}\left(\mathbf{x};\boldsymbol{\theta}_p\right);\boldsymbol{\theta}_m\right),\mathbf{y}\right), \tag{4.1}$$

where $\mathcal{P}\left(\mathbf{x};\boldsymbol{\theta}_p\right)$ is the transformed data using the preprocessing pipeline $\mathcal{P}$ with hyperparameters $\boldsymbol{\theta}_p$ on the inputs $\mathbf{x}$. $\mathcal{M}$ is the machine learning model (i.e Random Forest, SVM) with hyperparameters $\boldsymbol{\theta}_m$ that generates predictions, $\mathbf{y}$ are the targets or labels and $\mathcal{L}$ is the loss function.

Our goal in this work is not focus on model or hyperparameter search but to find a collection of preprocessing pipelines $\mathcal{C}$ where $\mathcal{C}(x) = p_0(x_0) \cdot \ldots \cdot p_k(x_k)$, and p_k a preprocessing for the feature k where $0 \leq k \leq$ number of features to solve the optimization problem:

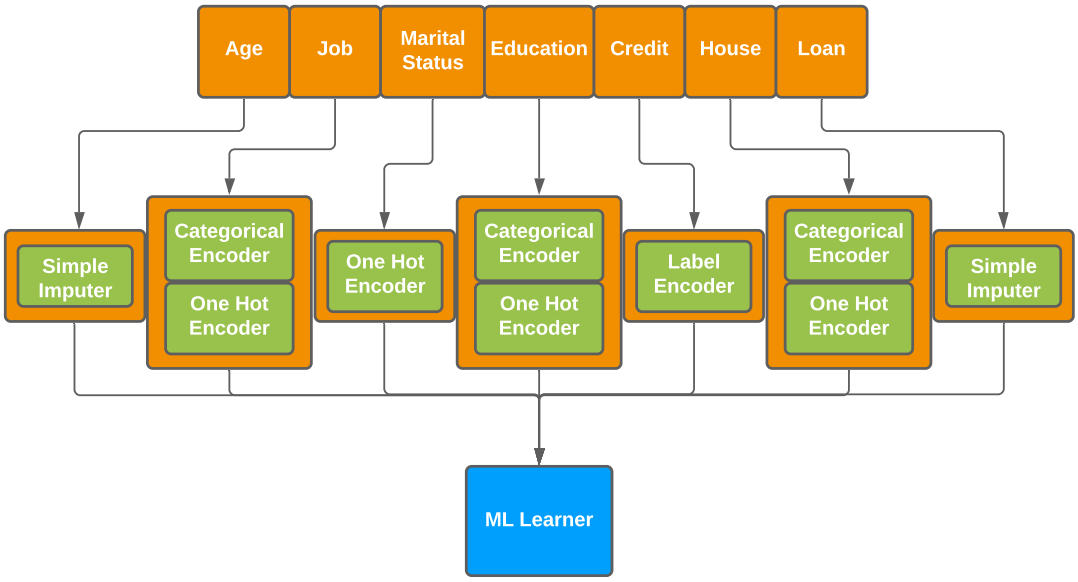
$$\arg\min_{\mathcal{C}} \mathcal{L}\left(\mathcal{M}(\mathcal{C}(\mathbf{x})),\mathbf{y}\right), \tag{4.2}$$

Note that although we are following the traditional setting, we want to solve the optimization problem by searching for the pipeline collection $\mathcal{C}$ and do not take into consideration the ML model $\mathcal{M}$ or the hyperparameters. As previously mentioned, since we focus on searching for the set preprocessing pipeplines $\mathcal{C}$, we decided to fix the loss function $\mathcal{L}$ to be the 10-cross fold valiation accuracy score and the ML model $\mathcal{M}$.

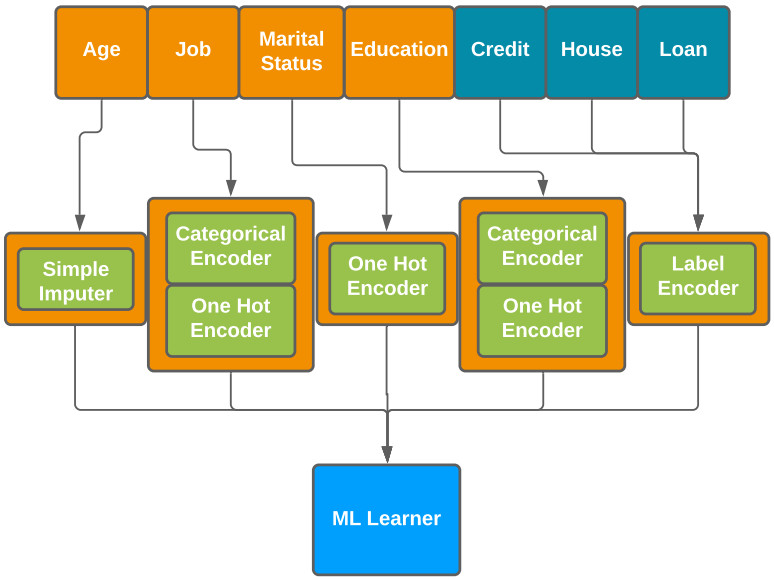
4.3 Framework

To tackle the challenge of high dimensional search space on feature-wise transformation search, we propose to instead search searching for transformations for similar features. The key idea is to reduce the number of preprocessing pipelines to search by clustering similar features. This approach will drastically reduce the search space. For example, for a feature subset from the dataset Credit Approval [2] dataset with seven features and eight different combinations of transformation, the search space would be over two million combinations. In figure 4.1a we show an example of how a clustered feature-wise pipeline could reduce the search space by clustering similar features that share the same values; by doing this, the search space is drastically reduced to around thirty-two thousand combinations.

[2] `https://archive.ics.uci.edu/ml/datasets/Credit+Approval`



(a) Feature-wise transformation pipeline.



(b) Clustered Feature-wise transformation pipeline.

Figure 4.2: Figure (a) shows an illustration of a preprocessing pipeline that uses Feature-wise pipelines for each feature. Figure (b) shows an example of how our method deals with multiple attributes by using a single pipeline. If we consider a search space with only eight pipelines to choose from for each feature, we would have over two million possible combinations for the Feature-wise case. However, our proposed method is capable of reducing this space to around thirty-two thousand.

We develop a framework that attempts to reduce the search space for preprocessing pipeline search by searching for transformation for clusters of features instead of each independent feature. The framework is three stages, feature embedding, feature clustering, and the preprocessing pipeline search. The idea behind of search strategy is that features can be clustered in so many different ways, so the search needs to learn how to cluster them efficiently. In Figure 4.3 the overall framework overlay is presented. The workflow goes as follows. First, the feature columns are embedded using an AutoEncoder, the embedding is used to cluster them. After the feature clusters are generated, they are forwarded to the random search in charge of searching for the preprocessing pipelines. After a valid set of preprocessing pipelines is found, the data is transformed to generate predictions with the Machine Learning Learner. The prediction is later used to tune the parameters of the clustering network. The process repeats until there are no more resources.

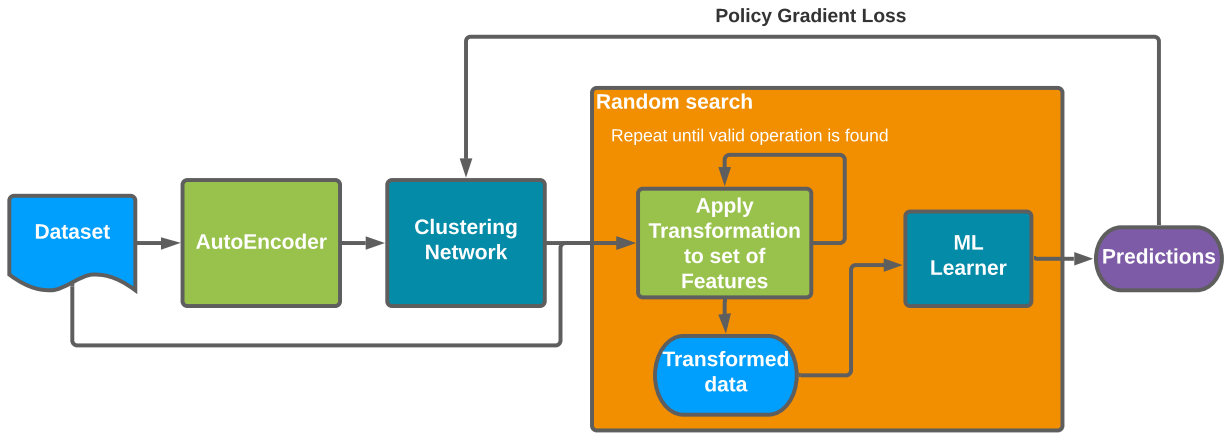


Figure 4.3: Clustered Feature-wise Pipeline Search Framework. An Autoencoder is used to generate the column feature embedded from the input dataset. Then a randomly initialized clustering network is used to generate the feature clusters. The cluster information is forwarded to the feature-wise random search that will eventually generate transformed data. Then the data is forwarded to the ML Learner to generate predictions used on the Policy Gradient Loss to generate a reward. Then, the reward is used to tune the Clustering Network. The process keeps repeating until no more resources are allocated.

4.3.1 Feature embedding

Feature embedding has been widely studied, and several methods have been proposed. Traditional methods such as PCA and SVD focus only on numerical data, and in order to support non-numerical values such as categorical, these methods rely on transforming the data to a numerical representation by using other transformers such as OneHotEncoder. Other methods provide more flexibility by creating vector representation of words and sentences, such as word2vec [61] and BERT [62]. Although these methods perform well on NLP tasks, it becomes challenging to use when a large corpus of unseen values is presented. Another drawback of these approaches is that they tend to require large amounts of data and time if they were to be trained from scratch.

Our approach to embedding the input features assumes that all the features are non-numerical values, then the features are transformed to numerical representation and computed the embedding by the use of AutoEncoders. As previously mentioned, most of the frameworks deal with non-numerical data by using OneHotEncoder, but in our case, this is not suitable since it can complicate the process of comparing the feature columns later on. First, to do the feature embedding, we compute the unique values across all features and create vector counts of every value for each column. The use of value count aims to capture the similarity of the columns in terms of repeated values, such as columns sharing the same numerical representation or the same categorical values; this approach also aims to capture missing values. After encoding the feature columns into vector counts, we use an AutoEncoder with Mean squared error (reconstruction error) to generate a more general embedding.

It is important to note that our embedding approach aims to generate feature embeddings such as columns that can be compared later on as a whole and not by observations. The feature embedding method aims to capture the relationships among different values across the features and generates an embedding on a lower dimension that can be easily compared.

4.3.2 Feature Clustering

Once the data is embedded, we could potentially use any out-of-the-shelve clustering method such as *kmeans*, mean-shift, or spectral clustering. The downside of these methods is that they take into consideration the initial state of the data (train), and when new data appears, it becomes complicated to adapt, and in most of the cases, the models need to be re-trained, and with this, the previous information gets lost. For most clustering methods, the number of clusters is a hyperparameter that gets fixed when the method is initialized, and they are not flexible enough to change. In our framework, we tackle these limitations from traditional cluster methods by the use of a clustering deep neural network. The framework uses a *Deep K-means* [63]. The neural network is treated as a classification network that takes as input a set of observations and a set of pseudo labels that is trained with the Cross-Entropy Loss. The pseudo labels are created by generating a set of cluster labels using K-means.

The clustering network in our framework aims to create clusters based on the features' similarities. Since the similarity can be measured according to different metrics [64], we decided instead of measuring similarity in terms of performance. The clustering network in our framework follows a similar logic as previously described. First, we generate pseudo-cluster labels by using k-means and the embedded data (note that these labels are not going to be used during training), and then we create a network that will be used to learn the cluster strategy. For the network, we define the size of the input layer according to the number of input features and the output layer according to a hyperparameter that represents the maximum number of clusters to generate. Then, the network is randomly initialized, and the embedded features are used in forward-pass generating clusters.

After generating the clusters, the information is passed to the preprocessing pipeline search, and then the predictions are returned. The predictions are later evaluated to compute the ground truth. Then, we use a Policy Gradient Loss [65] that takes as inputs the labels generated on k-means, the labels generated by the clustering network, and the performance. The loss function will generate a reward that penalizes the clusters generated by the network that do not score well during the search. Then, the reward is used for the clustering network to tune.

Since we are not directly training the clustering network with the labels from k-means, but instead, we are randomly initializing it, we expect to generate random clusters of features so with every iteration, the network will be following the reward that is provided by the performance of the search. By doing this, we are allowing the clustering network to generate different clusters through iterations that enable the exploration of other clusters configurations.

4.3.3 Preprocessing Pipeline Search

The preprocessing pipeline search component on the framework is in charge of searching for the preprocessing pipelines for each cluster of features. There are many strategies for this, such as Bayesian optimization, tree search, etc., but we decided to implement a random search variation. The intuition behind this is that the search would likely not search for the pipelines for the same set of clusters often since the clusters are changing through time by tuning the clustering network. Having a different set of clusters also leads to problems related to optimization techniques since they will require a fresh start for every new set of clusters. Also, it is very likely that if the search happens to get the same cluster configuration, the optimization would not have enough data samples to lead to a better solution.

We tackle these problems by using Random Search without replacements and a hashing mechanism to track the different cluster searches in our framework. The idea is that when a cluster configuration is forwarded to the search, the search will hash and check if the configuration already exists. If not, it will automatically create a new random search. If the cluster configuration exits, it will access the previous results and sample a new set of preprocessing pipelines. The preprocessing pipelines are generated by iterating on the different clusters and randomly sampling a transformation. If the transformation can successfully be applied to the data, then another transformation is selected, and the process is repeated. If the transformation fails, it is removed from the pool from that set of features to not be sampled again. The approach allows cutting evaluation time by hashing already seen configurations and avoiding the data transformations that are more likely to fail. The result is that we are sampling new preprocessing pipelines more efficiently over time.

4.4 Experiments

In this section, we empirically evaluated the proposed clustered feature-wise preprocessing pipeline search. To evaluate the performance of our method, we use eight datasets from the OpenML Benchmark Suit[66]. The datasets were chosen because they contain missing values and different data types, such as integer, float, boolean, and categorical values.

4.4.1 Search Space

The search space that we are using for our experiments is standard in AutoML System. In Table 3.9 we listed all the data transformations in the space. We limited the space to three main operations, imputers, encoders, and scalars. We follow the same order of operations as other frameworks where the order cannot be changed, but operations can be skipped—for example, a preprocessing pipeline with only encoders and scalars. This search space leads to the generation of forty-eight possible preprocessing pipelines.

Imputers	Encoders	Scalars
Median	Ordinal	MinMax
Most Common Value	One Hot	Standard
Mean	None	MaxAbs
None	-	None

Table 4.1: Preprocessing pipeline search space. The space follows a strict order of operations. It is divided into three main components: imputers, encoders, scalars. Each component has different available options that can be selected and includes the None operation.

4.4.2 Baselines

We utilize two main baselines in the final experiment. The first one is for the traditional AutoML preprocessing pipeline, and the second is a variation of the proposed method that randomly generates clusters to search. Their details are summarized as follows:

- **Predefined preprocessing pipeline.** Following the traditional AutoML setting, this method imputes missing values by using the most common value, then for the numerical values, apply a standard scaler. For the categorical values, OneHotEncoder is used. This method only distinguishes between two data types, numerical and non-numerical.

- **Random Clusters Search.** This method generates a set of preprocessing pipelines for a set of random clustered features. It is challenging for the search to yield valid results due to the large number of invalid clusters generated. We let the method run 10000 iterations to guarantee that at least one set of preprocessing pipelines can be scored for a fair comparison.

4.4.3 Method Settings

For our proposed framework, we set the number of iterations to 500; it will generate 500 clusters settings and pipelines so the clustering network can adjust. The reason for this is that if we were to run it using a time constraint, the number of iterations could vary due to the dataset size and complexity, which will cause us not properly to compare the results. Also, since the datasets we are using have a different number of features, we decided to fix the maximum number of features in our method to 5; this means that there are at most 5 clusters at any given time during the search.

4.4.4 Results

We empirically evaluate the frameworks by providing the full dataset [66] and evaluating the best pipeline they can generate as specified in previous subsections. We evaluate the pipelines using a ten cross-fold validation with the accuracy metric. Since our model focuses on the preprocessing pipeline search, we repeated the search multiple time with different standard machine learning

Dataset	Baseline	Random Clusters	CFWT
38-sick	0.9593	0.9729	0.9769
29-credit-a	0.8428	0.8255	0.8443
1049-pc4	0.8869	0.9006	0.9108
1480-ilpd	0.7077	0.7060	0.7111
23381-desses-sales	0.5804	0.5944	0.6219
3-kr-vs-kp	0.9477	0.9488	0.9518
40975-car	0.8642	0.8440	0.8670
50-tic-tac-toe	0.7752	0.8336	0.9098
Average	0.8205	0.8336	0.8492

Table 4.2: Comparison between the average accuracy of standard, random clustered and Clustered Feature-wise pipelines on OpenML-CC18 Benchmark datasets. CFWT shows a promising increase of performance for standard machine learners.

learners: Ada boost, Gradient Boosting, Random Fores, Support vector classification, a linear classifier with stochastic gradient descend learning, and a multi-layer perceptron.

In Table 4.2, the average accuracy is presented. Overall, our framework is capable of boosting the performance of machine learning learners by proposing other preprocessing pipelines with the same components. In Figure 4.4 the accuracy improvement on different learners is presented. The proposed method has more impact on the learners if the data contains different data types such as integer, boolean values. The performance improvement on machine learning learners that are tree-based is minimal since tree methods are already dealing with categorical data better.

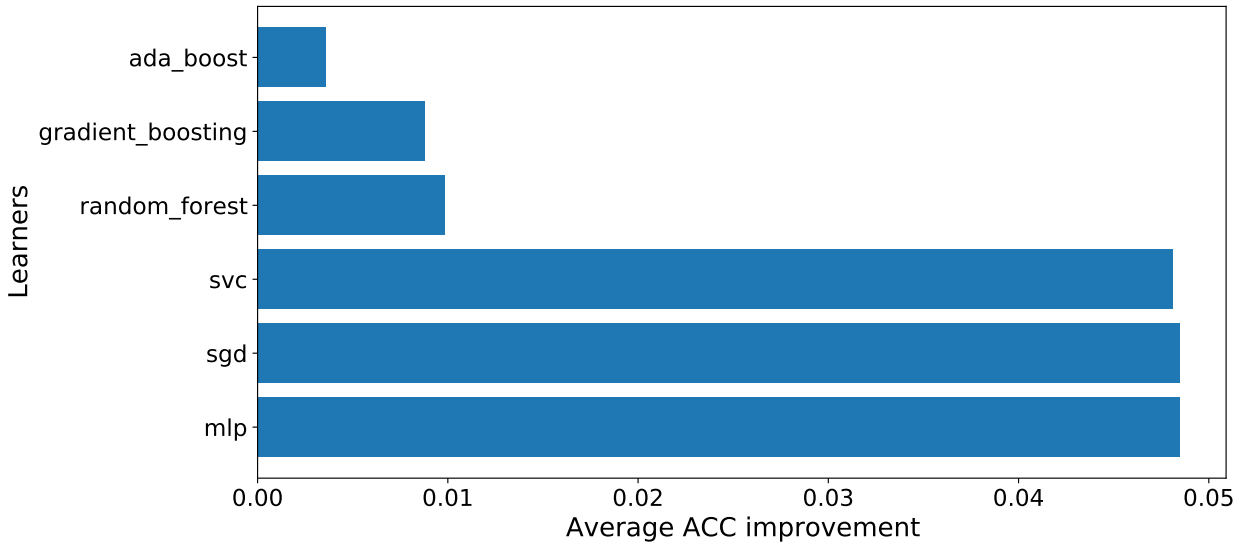


Figure 4.4: Accuracy improvement on different machine learning learners focuses on heterogeneous data from the OpenMl-CC18 Benchmark with a 500 iteration search. The proposed method has more impact on the learners if the data contains different data types such as integer, boolean values. The performance improvement on machine learning learners that are tree-based is minimal since tree methods are already dealing with categorical data better.

4.5 Conclusions

In this work, we proposed a framework that enables the search of more complex preprocessing pipelines by searching for pipelines for each feature instead of general ones that only consider at most two according to data types. A search algorithm is proposed that reduces the complexity of the search space generated by searching for individual pipelines by clustering similar features that lead to searching for fewer pipelines. The algorithm takes advantage of a Deep clustering network to guide the search that tunes itself with every iteration. Experimental results on eight benchmark datasets demonstrate the proposed search algorithm's effectiveness and its capability to improve the machine learning learners' performance.

5. CONCLUSION AND FUTURE RESEARCH OPPORTUNITIES

Automated Machine Learning provides tools for different levels of users to create ML solutions for their problems. However, most of the existing AutoMl systems only focus on the research aspect rather than usability, leading to the rise of the number of systems. Due to the creation of many systems, it becomes challenging to compare them since the components that each of them uses could lead to a broader impact rather than their proposed solution. In this dissertation, we made a series of contributions towards the standardization of AutoML components and evaluation of systems, how to design an AutoML framework to alleviate the increasing number of single-use AutoML solutions, and how to efficiently search for feature-wise data transformation for AutoML.

To tackle the challenges of standardization of AutoML components and evaluation of systems, we propose a standard framework that provides building blocks for AutoML systems. Such building blocks allows

Our research would promote the standardization of building blocks for AutoML systems, facilitate the development and evaluation of new systems, as well as enable the exploration of new approaches to preprocess the data on AutoML systems. Despite the efforts made in the thesis, there are still many challenges and open problems to be explored. With respect to future work, we are interested in investigating the following directions:

- **Create tools for automatic ingestion of machine learning primitives for AutoML.** With the number of machine learning primitives growing every year, it becomes almost impossible to keep up with manually creating code wrappers that provide the information for AutoML systems. This problem could be tackled from different perspectives; for example, the human documentation can be parsed to generate the information needed, or automatic testing tools could be used to test and assess the different hyperparameter values to create more dynamic documentation that, in most cases, humans are not able to describe the different scenarios, such as specific invalid configuration of hyperparameters.

- **Pipeline validation during syntax checking** AutoML systems, overall, spend a vast amount of time evaluating machine learning pipelines. In some cases, such as in the D3M ecosystem, the AutoML systems tend to waste so much time evaluating pipelines that they cannot be run because the inputs from one primitive to another are incorrect. Another reason why pipelines fail during evaluation is that the hyperparameter configuration is not valid because it relies on some aspect of the input data that is only available during runtime. The number of failed pipelines can be reduced if the errors are discovered before running them.

- **Enable interpretability in the D3M Ecosystem.** The D3M Ecosystem provides the building blocks to create AutoML systems such as primitives and pipelines definitions. Although this enables better comparison among systems, it becomes challenging to identify whether or not the improvement of performance is due to the search algorithm can produce better pipelines or because the machine learning primitives are powerful. The interpretation in AutoML could potentially inform the user why a system is selecting some primitives and creating an interactive system.

- **Enable discoverability of preprocessing pipelines for unseen data.** In this dissertation, we target the exploration on the preprocessing pipelines, but they are mainly restricted to known operations for know data such as categorical and numerical data. There are still many data types that require other primitives to transform to numerical representation, such as audio, video, image data. Although there are preprocessing pipelines for these data types, it is unknown whether or not they can be improved by combining with others.

REFERENCES

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

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

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

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

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

[6] R. S. Olson and J. H. Moore, "TPOT: A tree-based pipeline optimization tool for automating machine learning," in *Automatic Machine Learning: Methods, Systems, Challenges* (F. Hutter, L. Kotthoff, and J. Vanschoren, eds.), pp. 163–173, Springer, 2018.

[7] B. Komer, J. Bergstra, and C. Eliasmith, "Hyperopt-Sklearn," in *Automatic Machine Learning: Methods, Systems, Challenges* (F. Hutter, L. Kotthoff, and J. Vanschoren, eds.), pp. 105–121, Springer, 2018.

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

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

[10] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pp. 108–122, 2013.

[11] J. Vanschoren, J. N. van Rijn, B. Bischl, and L. Torgo, "OpenML: Networked science in machine learning," *SIGKDD Explorations*, vol. 15, no. 2, pp. 49–60, 2013.

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

[13] N. Fusi, R. Sheth, and M. Elibol, "Probabilistic matrix factorization for automated machine learning," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, (Red Hook, NY, USA), p. 3352–3361, Curran Associates Inc., 2018.

[14] Y. Heffetz, R. Vainshtein, G. Katz, and L. Rokach, "Deepline: Automl tool for pipelines generation using deep reinforcement learning and hierarchical actions filtering," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2103–2113, 2020.

[15] Google, "Cloud AutoML." `https://cloud.google.com/automl/`, 2019.

[16] H2O.ai, "H2o automl." `http://docs.h2o.ai/h2o/latest-stable/h2o-docs/automl.html`, June 2017. H2O version 3.30.0.1.

[17] L. Kotthoff, C. Thornton, H. H. Hoos, F. Hutter, and K. Leyton-Brown, "Auto-WEKA: Automatic model selection and hyperparameter optimization in weka," in *Automatic Machine Learning: Methods, Systems, Challenges* (F. Hutter, L. Kotthoff, and J. Vanschoren, eds.), pp. 89–103, Springer, 2018.

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

[19] H. Mendoza, A. Klein, M. Feurer, J. T. Springenberg, M. Urban, M. Burkart, M. Dippel, M. Lindauer, and F. Hutter, "Towards automatically-tuned deep neural networks," in *Automatic Machine Learning: Methods, Systems, Challenges* (F. Hutter, L. Kotthoff, and J. Vanschoren, eds.), pp. 145–161, Springer, 2018.

[20] P. Parry, "auto_ml: Automated machine learning for production and analytics," 2018.

[21] F. Mohr, M. Wever, and E. Hüllermeier, "ML-Plan: Automated machine learning via hierarchical planning," *Machine Learning*, vol. 107, pp. 1495–1515, 9 2018.

[22] H. Rakotoarison, M. Schoenauer, and M. Sebag, "Automated machine learning with monte-carlo tree search," *arXiv preprint arXiv:1906.00170*, 2019.

[23] A. G. C. de Sá, A. A. Freitas, and G. L. Pappa, "Automated selection and configuration of multi-label classification algorithms with grammar-based genetic programming," in *Parallel Problem Solving from Nature – PPSN XV* (A. Auger, C. M. Fonseca, N. Lourenço, P. Machado, L. Paquete, and D. Whitley, eds.), (Cham), pp. 308–320, Springer International Publishing, 2018.

[24] A. G. C. de Sá, W. J. G. S. Pinto, L. O. V. B. Oliveira, and G. L. Pappa, "RECIPE: A grammar-based framework for automatically evolving classification pipelines," in *Genetic Programming* (J. McDermott, M. Castelli, L. Sekanina, E. Haasdijk, and P. García-Sánchez, eds.), (Cham), pp. 246–261, Springer International Publishing, 2017.

[25] Salesforce, "TransmogrifAI." `https://transmogrif.ai/`, 2019.

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

[27] C. Steinrucken, E. Smith, D. Janz, J. Lloyd, and Z. Ghahramani, "The automatic statistician," in *Automatic Machine Learning: Methods, Systems, Challenges* (F. Hutter, L. Kotthoff, and J. Vanschoren, eds.), ch. 9, pp. 175–188, Springer, 2018. In press, available at `http://automl.org/book`.

[28] T. S. W. Drevo, B. Cyphers, A. Cuesta-Infante, A. Ross, and K. Veeramachaneni, "ATM: A distributed, collaborative, scalable system for automated machine learning," in *2017 IEEE International Conference on Big Data (Big Data)*, pp. 151–162, 12 2017.

[29] W. Wang, J. Gao, M. Zhang, S. Wang, G. Chen, T. K. Ng, B. C. Ooi, J. Shao, and M. Reyad, "Rafiki: machine learning as an analytics service system," *Proceedings of the VLDB Endowment*, vol. 12, no. 2, pp. 128–140, 2018.

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

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

[32] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," *Advances in neural information processing systems*, vol. 24, 2011.

[33] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *International conference on learning and intelligent optimization*, pp. 507–523, Springer, 2011.

[34] A. Klein, S. Falkner, S. Bartels, P. Hennig, and F. Hutter, "Fast bayesian optimization of machine learning hyperparameters on large datasets," in *Artificial Intelligence and Statistics*, pp. 528–536, PMLR, 2017.

[35] S. Falkner, A. Klein, and F. Hutter, "Bohb: Robust and efficient hyperparameter optimization at scale," in *International Conference on Machine Learning*, pp. 1437–1446, PMLR, 2018.

[36] U. Khurana and H. Samulowitz, "Automating predictive modeling process using reinforcement learning," *arXiv preprint arXiv:1903.00743*, 2019.

[37] L. Li, K. G. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: Bandit-based configuration evaluation for hyperparameter optimization," in *ICLR (Poster)*, 2017.

[38] F. Qi, Z. Xia, G. Tang, H. Yang, Y. Song, G. Qian, X. An, C. Lin, G. Shi, *et al.*, "A graph-based evolutionary algorithm for automated machine learning," *Software Engineering Review*, vol. 1, no. 2 (2020), pp. 10–37686, 2020.

[39] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," *arXiv preprint arXiv:1611.02167*, 2016.

[40] J. Davison *et al.*, "DEvol: Deep neural network evolution." `https://github.com/joeddav/devol`, 2017.

[41] H. Jin, Q. Song, and X. Hu, "Auto-Keras: Efficient neural architecture search with network morphism," *arXiv preprint arXiv:1806.10282*, 2018.

[42] Microsoft, "Neural network intelligence." `https://github.com/Microsoft/nni`, 2018.

[43] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, "Efficient neural architecture search via parameter sharing," *arXiv preprint arXiv:1802.03268*, 2018.

[44] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.

[45] "Learning transferable architectures for scalable image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8697–8710, 2018.

[46] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," *arXiv preprint arXiv:1806.09055*, 2018.

[47] H. Cai, L. Zhu, and S. Han, "Proxylessnas: Direct neural architecture search on target task and hardware," *arXiv preprint arXiv:1812.00332*, 2018.

[48] K. Kandasamy, W. Neiswanger, J. Schneider, B. Poczos, and E. Xing, "Neural architecture search with bayesian optimisation and optimal transport," *arXiv preprint arXiv:1802.07191*, 2018.

[49] R. Luo, F. Tian, T. Qin, E. Chen, and T.-Y. Liu, "Neural architecture optimization," *arXiv preprint arXiv:1808.07233*, 2018.

[50] R. Elshawi, M. Maher, and S. Sakr, "Automated machine learning: State-of-the-art and open challenges," *arXiv preprint arXiv:1906.02287*, 2019.

[51] P. Gijsbers, E. LeDell, J. Thomas, S. Poirier, B. Bischl, and J. Vanschoren, "An open source automl benchmark," *arXiv preprint arXiv:1907.00909*, 2019. Accepted at AutoML Workshop at ICML 2019.

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

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

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

[55] V. D'Orazio, J. Honaker, R. Prasady, and M. Shoemate, "Modeling and forecasting armed conflict: AutoML with human-guided machine learning," in *2019 IEEE International Conference on Big Data (Big Data)*, pp. 4714–4723, 2019.

[56] "Metalearning database service." `https://metalearning.datadrivendiscovery.org/`, 2019.

[57] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging AI applications," *CoRR*, vol. abs/1712.05889, 2017.

[58] T. O'Malley, E. Bursztein, J. Long, F. Chollet, H. Jin, L. Invernizzi, *et al.*, "Kerastuner." `https://github.com/keras-team/keras-tuner`, 2019.

[59] S. Sanders and C. Giraud-Carrier, "Informing the use of hyperparameter optimization through metalearning," in *2017 IEEE International Conference on Data Mining (ICDM)*, pp. 1051–1056, IEEE, 2017.

[60] D. Dua and C. Graff, "UCI machine learning repository," 2017.

[61] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013.

[62] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *ArXiv*, vol. abs/1810.04805, 2019.

[63] M. M. Fard, T. Thonet, and E. Gaussier, "Deep k-means: Jointly clustering with k-means and learning representations," *Pattern Recognition Letters*, vol. 138, pp. 185–192, 2020.

[64] J. Li, K. Cheng, S. Wang, F. Morstatter, R. P. Trevino, J. Tang, and H. Liu, "Feature selection: A data perspective," *ACM Computing Surveys (CSUR)*, vol. 50, no. 6, p. 94, 2018.

[65] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," 2016.

[66] B. Bischl, G. Casalicchio, M. Feurer, F. Hutter, M. Lang, R. G. Mantovani, J. N. van Rijn, and J. Vanschoren, "Openml benchmarking suites," 2019.

APPENDIX A

PIPELINE EXAMPLE IN JSON FORMAT

Example of a Pipeline on JSON format from Figure 2.2. The pipeline can generate predictions for data with only numerical values. First, a Dataset to Dataframe Primitive is used to transform the input dataset into a Dataframe common representation, and then it is parsed. Later, the attributes are extracted, and missing values are imputed. A gradient boosting classifier primitive is later used as a learner by using the features and targets. The predictions for the latter are forwarded to the Construct prediction primitive that also uses the original Dataframe as input to match the indexes and make sure that the pipeline output is in the correct format.

```
{
 "id": "67d78c06-691d-4b3d-8056-8bb67e3e75ff",
 "schema":
  ↪  "https://metadata.datadrivendiscovery.org/schemas/v0/pipeline.json",
 "inputs": [
  {
   "name": "inputs"
  }
 ],
 "outputs": [
  {
   "data": "steps.6.produce",
   "name": "Predictions from the input dataset"
  }
 ],
 "steps": [
  {
   "type": "PRIMITIVE",
   "primitive": {
```

```
      "python_path": "data_transformation.dataset_to_dataframe.Common",
      "name": "Extract a DataFrame from a Dataset",
      "id": "4b42ce1e-9b98-4a25-b68e-fad13311eb65",
      "version": "0.3.0"
     },
     "arguments": {
      "inputs": {
       "data": "inputs.0",
       "type": "CONTAINER"
      }
     },
     "outputs": [
      {
       "id": "produce"
      }
     ],
    },
    {
     "type": "PRIMITIVE",
     "primitive": {
      "python_path": "data_transformation.column_parser.Common",
      "name": "Parses strings into their types",
      "id": "d510cb7a-1782-4f51-b44c-58f0236e47c7",
      "version": "0.6.0"
     },
     "arguments": {
      "inputs": {
       "data": "steps.0.produce",
       "type": "CONTAINER"
      }
     },
     "outputs": [
      {
```

```
      "id": "produce"
     }
    ]
   },
   {
    "type": "PRIMITIVE",
    "primitive": {
     "python_path":
      ↪ "data_transformation.extract_columns_by_semantic_types.Common",
     "name": "Extracts columns by semantic type",
     "id": "4503a4c6-42f7-45a1-a1d4-ed69699cf5e1",
     "version": "0.3.0"
    },
    "arguments": {
     "inputs": {
      "data": "steps.1.produce",
      "type": "CONTAINER"
     }
    },
    "hyperparams": {
     "semantic_types": {
      "data": [
       "https://metadata.datadrivendiscovery.org/types/Attribute"
      ],
      "type": "VALUE"
     }
    },
    "outputs": [
     {
      "id": "produce"
     }
    ]
   },
```

```
    {
     "type": "PRIMITIVE",
     "primitive": {
      "python_path":
       ↪  "data_transformation.extract_columns_by_semantic_types.Common",
      "name": "Extracts columns by semantic type",
      "id": "4503a4c6-42f7-45a1-a1d4-ed69699cf5e1",
      "version": "0.3.0"
     },
     "arguments": {
      "inputs": {
       "data": "steps.0.produce",
       "type": "CONTAINER"
      }
     },
     "hyperparams": {
      "semantic_types": {
       "data": [
        "https://metadata.datadrivendiscovery.org/types/TrueTarget"
       ],
       "type": "VALUE"
      }
     },
     "outputs": [
      {
       "id": "produce"
      }
     ]
    },
    {
     "type": "PRIMITIVE",

     "primitive": {
```

```
      "python_path": "data_cleaning.imputer.SKlearn",
      "name": "sklearn.impute.SimpleImputer",
      "id": "d016df89-de62-3c53-87ed-c06bb6a23cde",
      "version": "2019.6.7"
     },
     "arguments": {
      "inputs": {
       "data": "steps.2.produce",
       "type": "CONTAINER"
      }
     },
     "hyperparams": {
      "return_result": {
       "data": "replace",
       "type": "VALUE"
      },
      "use_semantic_types": {
       "data": true,
       "type": "VALUE"
      },
      "error_on_no_input": {
       "data": false,
       "type": "VALUE"
      }
     },
     "outputs": [
      {
       "id": "produce"
      }
     ]
    },
    {
     "type": "PRIMITIVE",
```

```
    "primitive": {
     "python_path": "classification.gradient_boosting.SKlearn",
     "name":
      ↪  "sklearn.ensemble.gradient_boosting.GradientBoostingClassifier",
     "id": "01d2c086-91bf-3ca5-b023-5139cf239c77",
     "version": "2019.6.7"
    },
    "arguments": {
     "outputs": {
      "data": "steps.3.produce",
      "type": "CONTAINER"
     },
     "inputs": {
      "data": "steps.4.produce",
      "type": "CONTAINER"
     }
    },
    "outputs": [
     {
      "id": "produce"
     }
    ]
   },
   {
    "type": "PRIMITIVE",
    "primitive": {
     "python_path": "data_transformation.construct_predictions.Common",
     "name": "Construct pipeline predictions output",
     "id": "8d38b340-f83f-4877-baaa-162f8e551736",
     "version": "0.3.0"
    },
    "arguments": {
     "reference": {
```

```
        "data": "steps.0.produce",
        "type": "CONTAINER"
       },
       "inputs": {
        "data": "steps.5.produce",
        "type": "CONTAINER"
       }
      },
      "outputs": [
       {
        "id": "produce"
       }
      ]
     }
    ]
}
```

APPENDIX B

SEARCH ALGORITHM EXAMPLE

Example of creating a search algorithm using Axolotl AutoML. The search method called Predefined Search uses a small collection of already built pipelines and queries them based on the task. The collection only contains classification and regression pipelines that were previously collected. The search iterates over the pipelines and selects the ones with the highest rank, defined as the average normalized accuracy.

```
class PredefinedSearch(PipelineSearchBase):
  def __init__(self, problem_description, backend, *,
   ↪  primitives_blocklist=None, ranking_function=None):
    super().__init__(
      problem_description=problem_description, backend=backend,
      primitives_blocklist=primitives_blocklist,
       ↪  ranking_function=ranking_function
    )

    if self.ranking_function is None:
      self.ranking_function = random_rank
    self.task_description = schemas_utils.get_task_description(
      self.problem_description['problem']['task_keywords']
    )

    self.available_pipelines = self._return_pipelines(
      self.task_description['task_type'],
       ↪  self.task_description['task_subtype'],
       ↪  self.task_description['data_types'])
```

```
16
17      # Selection of a data preparation pipeline, we provide some
        ↪  predefine options such as train_test_split, k_fold, etc
18      # as well as the user can provide their own.
19      self.data_preparation_pipeline =
        ↪  schemas_utils.get_splitting_pipeline("K_FOLD")
20      # Get the metrics to evaluate the pipelines based on the
        ↪  problem description.
21      self.metrics =
        ↪  self.problem_description['problem']['performance_metrics']
22      # Pipeline to be use for scoring, we recommend using the one
        ↪  provided.
23      self.scoring_pipeline = schemas_utils.get_scoring_pipeline()
24      # Get the parameters for the datapreparation pipeline, such as
        ↪  number of folds, and so on.
25      self.data_preparation_params =
        ↪  schemas_utils.DATA_PREPARATION_PARAMS['k_fold_tabular']
26
27      self.offset = 10
28      self.current_pipeline_index = 0
29
30    def _search(self, time_left):
31      # Read all the pipelines to be evaluated
32      pipelines_to_eval =
        ↪  self.available_pipelines[self.current_pipeline_index:
        ↪  self.current_pipeline_index+self.offset]
33      self.current_pipeline_index += self.offset
34
35      # Evaluate the pipelines.
```

```
36      pipeline_results = self.backend.evaluate_pipelines(
37        problem_description=self.problem_description,
          ↪ pipelines=pipelines_to_eval, input_data=self.input_data,
38        metrics=self.metrics,
          ↪ data_preparation_pipeline=self.data_preparation_pipeline,
39        scoring_pipeline=self.scoring_pipeline,
          ↪ data_preparation_params=self.data_preparation_params)
40
41      return [self.ranking_function(pipeline_result) for
         ↪ pipeline_result in pipeline_results]
42
43    def _return_pipelines(self, task_type, task_subtype, data_type):
44      pipeline_candidates = []
45      for pipeline_dict in
         ↪ schemas_utils.get_pipelines_db()['CLASSIFICATION']:
46        pipeline = pipeline_utils.load_pipeline(pipeline_dict)
47        pipeline.id = str(uuid.uuid4())
48        pipeline.created = Pipeline().created
49        pipeline_candidates.append(pipeline)
50
51      return pipeline_candidates
```

APPENDIX C

SEMANTIC HANDLER EXAMPLE BUILT IN AXOLOTL

Example of a Semantic Handler in Axolotl that determines applies different transformation according to the cardinality of the categorical values. The handler extends from the PrimitiveHandler class. Semantic Handlers have definitions of what data semantics are supported. The transformations are applied to the columns containing the data semantics and ignored otherwise. This approach allows applying multiple handlers consecutively without affecting other data columns.

```
class CategoricalHandler(PrimitiveHandler):
  SUPPORTED_SEMANTICS = (schemas_utils.DATA_SEMANTICS.Categorical,)

  def __init__(self, resolver=Resolver()):
    super().__init__(primitive=None, hyperparams=[],
     ↪  resolver=resolver)

  def _add_produce(self, available_data, pipeline,
   ↪  arguments_references):
    data_ref = arguments_references['inputs']
    data = available_data[data_ref]

    # we get the indexes to work with
    categorical_indexes =
     ↪  data.metadata.list_columns_with_semantic_types(
      [schemas_utils.DATA_SEMANTICS.Categorical]
    )
    target_index = data.metadata.list_columns_with_semantic_types(
      [schemas_utils.DATA_SEMANTICS.Target]
```

```
    )[0]

    if target_index in categorical_indexes:
      categorical_indexes.remove(target_index)

    if categorical_indexes:
      arguments_references['inputs'] = add_produce_imputer(
        self, available_data, pipeline, arguments_references,
        ↪ PreparationPrimitives.Imputer,
        {
          'use_columns': categorical_indexes,
          'return_result': 'new',
          'strategy': 'most_frequent'
        }
      )
      return self._add_produce_categorical(available_data, pipeline,
      ↪ arguments_references)
    else:
      return data_ref

  def _add_produce_categorical(self, available_data, pipeline,
  ↪ arguments_references):
    data_ref = arguments_references['inputs']
    data = available_data[data_ref]
    total_n_values = len(data)

    index_to_drop = []
    index_to_onehot = []
    index_to_ordinal = []
```

```
43
44      for idx_col in range(len(data.columns)):
45        n_categories = len(data.iloc[:, idx_col].unique())
46        categories_ratio = n_categories / total_n_values
47
48        if categories_ratio > 0.8 and n_categories > 1000:
49          index_to_drop.append(idx_col)
50        elif n_categories <= 200:
51          index_to_onehot.append(idx_col)
52        else:
53          index_to_ordinal.append(idx_col)
54
55      new_data_refs = []
56      if index_to_onehot:
57        handler = PrimitiveHandler(
58          primitive=PreparationPrimitives.OneHotMaker,
59          hyperparams={
60            'use_columns': index_to_onehot,
61            'return_result': 'new',
62            'handle_unseen': 'column',
63            'handle_missing_value': 'column',
64            },
65          resolver=self.resolver
66        )
67        new_data_refs.append(handler.add_produce(available_data,
          ↪  pipeline, arguments_references={'inputs': data_ref}))
68
69      if index_to_ordinal:
70        handler = PrimitiveHandler(
```

```
71            primitive=PreparationPrimitives.OrdinalEncoder,
72            hyperparams={
73              'use_columns': index_to_ordinal,
74              'return_result': 'new',
75            },
76            resolver=self.resolver
77          )
78          new_data_refs.append(handler.add_produce(available_data,
              ↪  pipeline, arguments_references={'inputs': data_ref}))
79
80        if new_data_refs:
81          if len(new_data_refs) == 1:
82            return new_data_refs[0]
83          else:
84            handler = PrimitiveHandler(
85              primitive=PreparationPrimitives.HorizontalConcat,
                ↪  resolver=self.resolver
86            )
87            return handler.add_produce(
88              available_data, pipeline, arguments_references={'left':
                ↪  new_data_refs[0], 'right': new_data_refs[1]}
89            )
90        else:
91          return data_ref
```