Towards Automatic Machine Learning Pipeline Design

Mitar Milutinovic

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2019-123
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-123.html

August 16, 2019

Towards Automatic Machine Learning Pipeline Design

by

Mitar Milutinovic

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Dawn Song, Chair
Professor Trevor Darell
Professor Joseph Gonzalez
Professor James Holston

Summer 2019

Towards Automatic Machine Learning Pipeline Design

Abstract

Towards Automatic Machine Learning Pipeline Design

by

Mitar Milutinovic

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Dawn Song, Chair

The rapid increase in the amount of data collected is quickly shifting the bottleneck of making informed decisions from a lack of data to a lack of data scientists to help analyze the collected data. Moreover, the publishing rate of new potential solutions and approaches for data analysis has surpassed what a human data scientist can follow. At the same time, we observe that many tasks a data scientist performs during analysis could be automated. Automatic machine learning (AutoML) research and solutions attempt to automate portions or even the entire data analysis process.

We address two challenges in AutoML research: first, how to represent ML programs suitably for metalearning; and second, how to improve evaluations of AutoML systems to be able to compare approaches, not just predictions.

To this end, we have designed and implemented a framework for ML programs which provides all the components needed to describe ML programs in a standard way. The framework is extensible and framework's components are decoupled from each other, e.g., the framework can be used to describe ML programs which use neural networks. We provide reference tooling for execution of programs described in the framework. We have also designed and implemented a service, a metalearning database, that stores information about executed ML programs generated by different AutoML systems.

We evaluate our framework by measuring the computational overhead of using the framework as compared to executing ML programs which directly call underlying libraries. We observe that the framework's ML program execution time is an order of magnitude slower and its memory usage is twice that of ML programs which do not use this framework.

We demonstrate our framework's ability to evaluate AutoML systems by comparing 10 different AutoML systems that use our framework. The results show that the framework can be used both to describe a diverse set of ML programs and to determine unambiguously which AutoML system produced the best ML programs. In many cases, the produced ML programs outperformed ML programs made by human experts.

To Andrea

Contents

List of Figures

List of Tables

Acknowledgments

Foremost, I would like to thank my wife Andrea and my son Nemo for their utmost patience. All your hugs gave me all the energy I needed.

In no particular order, I would like to thank Ryans Zhuang, Kevin Zeng, Julia Cao, Roman Vainshtein, Rok Mihevc, Asi Messica, Charles Packer, and many other colleagues and students at UC Berkeley with whom I have worked on projects and research underpinning this work. Without you this work would not have been possible.

This work builds on many other projects and collaborations, primarily through the Darpa D3M program. I would like to thank everyone in the program, and especially those active in working groups through which discussed many topics present in this work. Just to name a few who have again and again stepped up to various challenges along the way: Diego Martinez, Brandon Schoenfeld, Sujen Shah, Mark Hoffmann, Alice Yepremyan, Shelly Stanley. No list would be complete without Wade Shen, who has had the vision and commitment to push the program through despite all the issues along the way. Moreover, Rob Zinkov, Atılım Güneş Baydin, and Frank Wood were pivotal in pushing me to see that what has been a simple initial straw man proposal can be much more, and that has ultimately led to this work.

Amazing Ray team, especially Robert Nishihara and Richard Liaw, thank you for guiding me when I got stuck. Moreover, thank you for addressing issues and feature requests quickly and efficiently, this makes your project really special.

Thank you to all who read through early drafts of this work and gave valuable feedback, especially Diego Martinez, Brandon Schoenfeld, Adrianne Zhong, Marten Lohstroh, and Andreas Mueller.

I was lucky to have not just one but three advisors: professors Dawn Song, Trevor Darell, and Joseph Gonzalez. Thank you for all the insights, suggestions, hard questions, and gentle pushes. Each of you contributed a fundamental piece of my experience at UC Berkeley.

Of course, without my parents and their support at every step along the way, nothing would have ever been possible. Thank you.

Chapter 1

Introduction

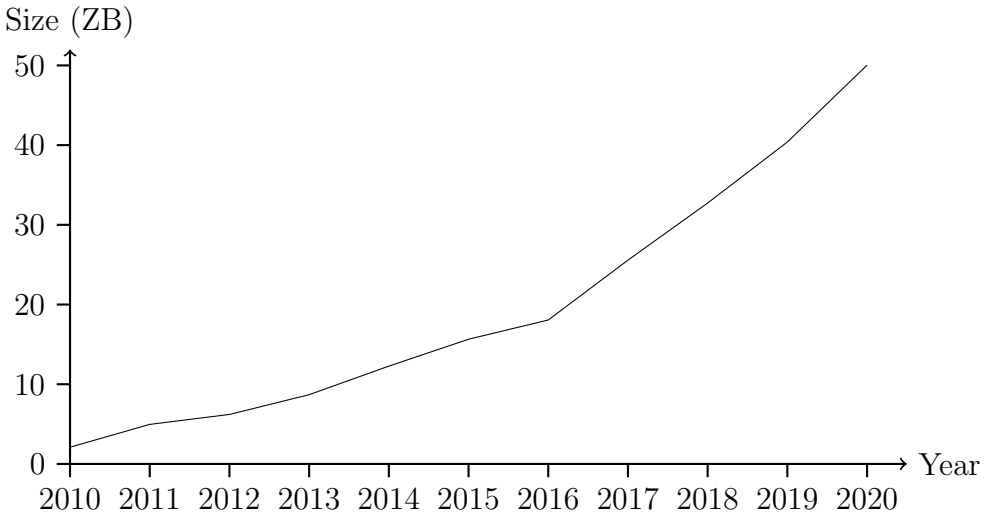


Figure 1.1: Annual size of the global datasphere. Source: IDC, November 2018, sponsored by Seagate [39].

Data available for potential use in ML programs is growing at a high rate as shown in Figure 1.1. IDC forecasts the global datasphere to grow to 50 ZB by 2020 [39]. At the same time there is a shortage of data scientists, Table 1.1. Looking at the number of AI/ML preprints on arXiv in cs.AI, stat.ML, and cs.NE categories published each year (Figure 1.2) we can see that it is growing dramatically and that just in 2018 there were more than 12,000 preprints published on arXiv alone. Those preprints can contain potential new solutions and approaches which could be used in ML programs. But it is not possible for any single individual to learn about them all, to learn for which problems and data types they are useful, to learn their effective combinations, nor how they could be used in ML programs.

One way to address this challenge is by using an Automated Machine Learning (AutoML) system to help analyze data and build ML programs which use data. Given data and a

Metro Area	July 2015	July 2018	3Y Delta
New York City, NY	+4,132	+34,032	+29,900
San Francisco Bay Area, CA	+10,995	+31,798	+20,803
Los Angeles, CA	+425	+12,251	+11,826
Boston, MA	+1,667	+11,276	+9,609
Seattle, WA	+1,182	+9,688	+8,506
Chicago, IL	−1,826	+5,925	+7,751
Washington, D.C.	+735	+7,686	+6,951
Dallas-Ft. Worth, TX	−2,496	+3,641	+6,137
Atlanta, GA	−2,301	+3,350	+5,651
Austin, TX	+26	+4,949	+4,923

Table 1.1: The intensification of local shortages for data science skills, July 2015 to July 2018. Table provides the shortage (+) or surplus (-) of people with data science skills in each metro area, and the associated delta over three years. Source: LinkedIn [26].

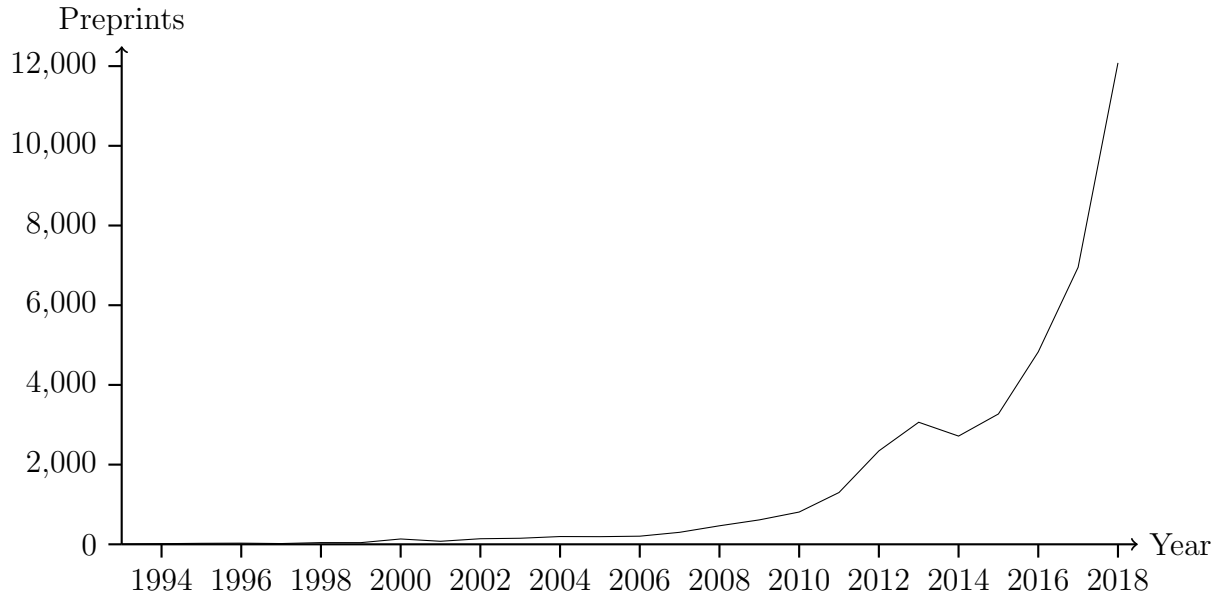


Figure 1.2: Number of AI/ML preprints on arXiv in cs.AI, stat.ML, and cs.NE categories published each year. Source: arXiv, May 2019 [8].

problem description, an AutoML system automatically creates an ML program to preprocess this data and to build a machine learning model solving the problem. Ideally, the automatic process of creation of an ML program should take into account all existing ML knowledge.

In the context of AutoML research, there are two challenges we tackle in this work: how to compare AutoML systems and how to better support AutoML systems which use metalearning.

Comparison of AutoML systems

There are many attempts at AutoML systems both in academia and industry (see Chapter 2), but there are many challenges to determine the quality of those AutoML systems.‘ Quality of an AutoML system can consist of many factors, e.g.:

- How much resources it needs to run?
- How quickly it creates an ML program?
- How far is this ML program from the best ML program?
- How clean or structured input data has to be?
- Which problem types does it support?
- How well it searches the space of possible ML programs?

Moreover, comparison of AutoML systems is hard because they use different sets of building blocks in their ML programs and use different datasets for their reported evaluation. If building blocks are different, maybe one AutoML system has simply a better building block available and this is why its ML program outperforms an ML program made by some other AutoML system. But if both systems used the same building blocks, it might be the case that the latter AutoML system creates that same (better) ML program as well and even faster.

Furthermore, it is hard to compare AutoML systems if the quality of ML programs themselves is not well defined or if different ML programs use different definitions of quality. Generally we care about ML program's quality of predictions as computed by some metric, but there are also other aspects of ML programs we can care about: complexity, interpretability, generalizability, resources and data requirements, etc.

Metalearning

One big family of approaches to AutoML is centered around metalearning. Metalearning treats AutoML itself as an ML problem. Because both ML programs as created by such an AutoML system and the AutoML system itself both operate on data and build an ML

sepal length	sepal width	petal length	petal width	species
5.1	3.5	1.4	0.2	Iris-setosa
4.9	3	1.4	0.2	Iris-setosa
7	3.2	4.7	1.4	Iris-versicolor
6.4	3.2	4.5	1.5	Iris-versicolor
6.3	3.3	6	2.5	Iris-virginica
5.8	2.7	5.1	1.9	Iris-virginica

Table 1.2: A sample of the Iris dataset [2, 15].

dataset	problem description	program ID	score
iris	classification	ca41e6a5	0.87
iris	classification	759e40f2	0.93
boston	regression	8727d30d	0.85
boston	regression	3ad4cc03	0.99
sunspots	forecasting	37181a32	0.91
sunspots	forecasting	e863382f	0.51

Table 1.3: An example metalearning dataset.

model, we use prefix *meta* when talking about the data and model of an AutoML system which uses metalearning: metalearning dataset or meta-dataset and meta-model.

For metalearning a dataset is needed which serves as input to a meta-model. If a regular tabular dataset looks like Table 1.2, a metalearning dataset looks like Table 1.3: instead of samples with attributes and targets, each sample consists, conceptually, of a dataset, a problem description, a program, and a score achieved by executing the program on the dataset and the problem description. A meta-model is trained that for a given new dataset and a problem description it constructs an ML program which achieves the best score. There are many challenges about metalearning, but in this work we focus on how to represent programs in such metalearning dataset. Representation of datasets and problem descriptions we leave to future work in Sections 5.2 and 5.3.

1.1 Contributions

We address challenges presented with the following contributions:

- We have designed and implemented a framework for ML programs which provides all components needed to describe ML programs in a standard way suitable for metalearning. The framework is extensible and framework's components are decoupled from each other.
- We provide reference tooling for execution of programs described in the framework.

- We present how this framework is used by 10 AutoML systems and how it addresses the challenge of comparison of AutoML systems.
- We have designed and implemented a service to serve as a metalearning dataset, storing information about executed ML programs by different AutoML systems.

Contributions empower each other: a standard way of describing ML programs enables both better comparison between AutoML systems and metalearning across ML programs created by different AutoML systems, allowing shared representation of information about executed ML programs and construction of a metalearning dataset.

Chapter 2

Related work

The AutoML research field is active and vibrant and has produced many academic and non-academic systems [10, 14, 18, 20, 22, 23, 27, 28, 31, 32, 38, 40, 41, 42, 43, 45, 47, 50, 52], including some focusing on neural networks only [3, 9, 21, 29, 36, 53]. Wse can observe [12] that there are many approaches they take and that they are implemented in various programming languages. Those differences lead to challenges in comparison of AutoML systems. Existing comparisons [17, 19] compare only predictions made by those systems. While for practical purposes it is important to compare what can systems achieve as they are, it does not provide any insight into how well the approaches they are taking fundamentally compare. Comparison is further complicated because different systems support different data types and task types. In this work we present a framework which enables comparison of approaches and not just predictions, across data types and problem types.

Many AutoML systems, to our knowledge at least [10, 14, 27, 40, 41, 43, 52], use some sort of metalearning. But they cannot learn from results across systems. Our framework addresses that through shared representation of ML programs and a shared metalearning service. [49] is a similar shared service to store information about ML programs and their performance on various datasets, but stored performance scores are self-reported and ML programs are not necessarily reproducible, limiting usefulness for cross-system metalearning.

Systems which focus on neural networks [3, 9, 21, 29, 36, 53] can be combined with other AutoML systems using our framework.

AutoML systems do not use one shared representation of ML programs. There are some popular pipeline languages which might be candidates for such a purpose. scikit-learn [33] pipeline allows combining multiple scikit-learn transforms and estimators. While powerful, it inherits some weaknesses from scikit-learn itself, primarily its support for only tabular and already structured data. This prevents it to be used when inputs are raw files. Moreover, its combination of linear and nested structure can become very verbose. Common Workflow Language [46] is a standard for describing data-analysis workflows with focus on reproducibility. But its focus is also on combining command line programs into workflows, which is generally not what ML programs made by AutoML systems consist of. Kubeflow [24] provides a pipeline language and at the same time makes their deployments on Kubernetes

simple. Similar to our framework it allows combining components using different libraries, but every component is a Docker image, and instead of directly passing memory objects between components, inputs and outputs have to be serialized.

There are existing tools to describe hyper-parameters configuration [5, 13]. Our framework aims to be compatible with them while extending a static configuration with optional custom sampling logic. This allows authors to define a new type of a hyper-parameter space and provide a custom sampling logic without AutoML systems having to support that type of a hyper-parameter space in advance.

Chapter 3

Framework for ML pipelines

We have designed and implemented a *framework* for ML programs for use in AutoML systems. We provide reference tooling for execution of programs described as framework's *pipelines*. We have designed and implemented a service to store and share information about executed pipelines as *pipeline run descriptions*. The collection of pipeline run descriptions can serve as a metalearning dataset.

In this chapter we present technical details of the framework and related tooling and service.

3.1 Design goals

In 2019, the most popular programming language for ML programs was Python [11]. An example of such an ML program in Python for the Thyroid disease dataset [48] is available in Figure 3.1. In the example program we first select a target column and attribute columns from input data. Then we further select numerical and categorical attributes. We encode categorical attributes and we impute missing values in numerical attributes. After that we combine categorical attributes and numerical attributes back into one data structure of all attributes. We then pass this data structure, together with targets, to a classifier to fit and predict. The program runs in two passes, in the first we fit on training data and in the second pass we only predict on testing data. This example program contains general steps found in ML programs: data loading, data selection and cleaning, and finally model building. It does not contain common steps like feature extraction, construction, and selection.

If we look at such ML programs as raw input of an AutoML system from which the system might want to learn from, we can observe:

- Language specific constructs which have nothing to do with the ML task at hand, e.g., `import` statements.
- Syntax of the programming language allows logically equivalent programs to be represented with different characters (changing a variable name does not change the logic

```
1 import numpy
2 import pandas
3 from sklearn.preprocessing import OrdinalEncoder
4 from sklearn.impute import SimpleImputer
5 from sklearn.ensemble import RandomForestClassifier
6
7 train_dataframe = pandas.read_csv('sick_train_split.csv')
8 test_dataframe = pandas.read_csv('sick_test_split.csv')
9
10 encoder = OrdinalEncoder()
11 imputer = SimpleImputer()
12 classifier = RandomForestClassifier(random_state=0)
13
14 def one_pass(dataframe, is_train):
15     target = dataframe.iloc[:, 30]
16     attributes = dataframe.iloc[:, 1:30]
17
18     numerical_attributes = attributes.select_dtypes(numpy.number)
19     categorical_attributes = attributes.select_dtypes(numpy.object)
20
21     categorical_attributes = categorical_attributes.fillna('')
22
23     if is_train:
24         encoder.fit(categorical_attributes)
25         imputer.fit(numerical_attributes)
26
27     categorical_attributes = encoder.transform(categorical_attributes)
28     numerical_attributes = imputer.transform(numerical_attributes)
29
30     attributes = numpy.concatenate([
31         categorical_attributes,
32         numerical_attributes,
33     ], axis=1)
34
35     if is_train:
36         classifier.fit(attributes, target)
37
38     return classifier.predict(attributes)
39
40 one_pass(train_dataframe, True)
41 predictions = one_pass(test_dataframe, False)
```

Figure 3.1: An example ML program in Python programming language.

```
import numpy
import pandas
from sklearn.compose import make_column_transformer
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import OrdinalEncoder
from sklearn.impute import SimpleImputer
from sklearn.ensemble import RandomForestClassifier

train_dataframe = pandas.read_csv('sick_train_split.csv')
test_dataframe = pandas.read_csv('sick_test_split.csv')

train_attributes = train_dataframe.iloc[:, 1:30]
train_target = train_dataframe.iloc[:, 30]
test_attributes = test_dataframe.iloc[:, 1:30]

def get_numerical_attributes(X):
    return X.dtypes.apply(lambda d: issubclass(d.type, numpy.number))

def get_categorical_attributes(X):
    return X.dtypes == 'object'

pipeline = make_pipeline(
    make_column_transformer(
        (
            make_pipeline(
                SimpleImputer(strategy='constant', fill_value=''),
                OrdinalEncoder(),
            ),
            get_categorical_attributes,
        ),
        (SimpleImputer(), get_numerical_attributes),
    ),
    RandomForestClassifier(random_state=0),
)

pipeline.fit(train_attributes, train_target)

predictions = pipeline.predict(test_attributes)
```

Figure 3.2: An example program from Figure 3.1 in a different programming style.

of the program, adding a comment or empty lines neither).

- Different programming styles might lead to very different programs and different ways of expressing the same underlying logic. The program in Figure 3.2 uses a different programming style, but is logically equivalent to the program in Figure 3.1.
- The programming language used is a general programming language which allows program to do more than just solve an ML task, e.g., display user interface, periodically save state, parallelize execution. That code is interleaved with code corresponding to the ML task.
- The programming language allows code with side effects and non-determinism. This can lead to a program not producing the same results when run multiple times on the same input data. Reproducibility of results can be achieved primarily through programming discipline.

Such properties of a programming language and programs in that language are generally reasonable and even seen as an advantage of the programming language when the programming language is used by humans. But in the context of AutoML systems which would consume such programs for learning and produce new programs as their outputs, all automatically, those properties can be seen as unnecessary complexity.

In this work we present a framework for ML pipelines that AutoML systems can directly consume and produce. The design goals of this framework are:

- The framework should allow most of ML and data processing programs to be described as its pipelines, if not all, but be as simple as possible to facilitate both automatic generation and automatic consumption of pipelines.
- Pipelines should allow description of complete end-to-end ML programs, starting with raw files and finishing with predictions or any other ML output from models embedded in pipelines.
- The focus of the framework is machine generation and consumption as opposed to human generation and consumption. It should enable automation as much as possible.
- The framework should be extensible and framework's components should be decoupled from each other, cf. in most programming languages a typing system and execution semantics are tightly coupled with the language itself.
- Control of side-effects and randomness in pipelines, and in general full reproducibility should be part of the framework and not an afterthought.

3.2 Syntax of pipelines

Pipelines do not have a human-friendly syntax and are primarily represented as in-memory data structures. Many of our framework's components, including pipelines, can be represented in JSON [6] or YAML [4] serialization formats. We provide validators using JSON Schema [16] to validate serialized data structures.

3.3 Pipeline structure

In our framework, ML programs are described as pipelines. Such pipelines consist of:

- Pipeline metadata.
- Specification of inputs and outputs of the pipeline.
- Definition of pipeline steps.

While pipeline is an in-memory structure, we call its standard representation a *pipeline description*. We support JSON and YAML serialization formats for pipeline descriptions and we provide a validator for pipeline descriptions using JSON Schema. The full list of standardized top-level fields of pipeline descriptions is available in Appendix B. Moreover, we can represent the main aspects of a pipeline structure visually. In this work we will use YAML and visual representations to present the pipeline structure.

Pipeline metadata contains mostly non-essential information about the pipeline: a human-friendly name and description, and when and how the pipeline was created. The only required metadata is a pipeline's universally unique identifier, UUID [25]. We standardize metadata as part of a pipeline description's JSON Schema.

Specification of inputs and outputs of the pipeline consist of defining the number of inputs and outputs the pipeline has, and optionally providing human-friendly names for them.

Pipeline steps define the logic of the pipeline. They are specified in order and each step defines its inputs and outputs and how step's inputs connect to any output of any previous step or pipeline's inputs. Connecting steps in this manner forms a DAG. There are currently three types of steps defined:

- Primitive step.
- Sub-pipeline step.
- Placeholder step.

Primitive step represents execution of a *primitive*. Primitives are described in Section 3.4. Sub-pipeline step represents execution of another pipeline as a step. This is similar to a function call in programming languages. Placeholder step can be used to define *pipeline templates* which can be used to represent partially defined pipelines.

Pipeline metadata can contain a digest over whole pipeline description. References to primitives and sub-pipelines can contain their expected digest as well. When a pipeline is loaded and references are de-referenced, it might happen that a different version of a primitive or a sub-pipeline is found. Those differences can be detected through mismatched digests and can help better understand why pipeline results might not be reproducible. We discuss reproducibility of pipelines in more detail in Section 4.3.

Note that pipeline structure is defined in general terms and can be extended with other step types. Moreover, the semantics of inputs, outputs, and the connections between them are not restricted by the pipeline structure.

3.4 Primitives

Primitives are basic building blocks of pipelines. They represent *learnable functions*, functions which do not have their logic necessarily defined in advance, but can learn it given example inputs and outputs. Concrete definition of semantics of such learnable functions depends on execution semantics used, which we will explore in Section 3.10. Moreover, primitives can be defined as regular functions as well, with pre-defined logic, what we see as a special case of learnable functions.

Primitives are written in Python by extending a suitable base class and optionally additional mixins. Their underlying logic can also be written in Python, but it can also be written in any other language with Python just serving as a wrapper to expose underlying logic in a standard way. Among primitives used by AutoML systems described in Section 4.6, we have seen primitives (partially) written in C/C++, Julia, R, and Java, besides those in just Python. Moreover, primitives can execute code on both CPUs and GPUs. From the perspective of the framework, those details are abstracted out.

In general, a primitive is defined by:

- Primitive metadata.
- Parameters (state) being learned.
- Hyper-parameters.
- Structural type of primitive's inputs.
- Structural type of primitive's outputs.
- Implementation of required methods, the *primitive interface*.

Not all primitives require all of those. E.g., primitives defining regular functions with pre-defined logic do not need parameters.

Primitive metadata

Primitive metadata describes the primitive in a standard way, standardized through a JSON Schema. Part of metadata is provided by a primitive's author and part of it is automatically generated by inferring it from a primitive's Python code (e.g., available/implemented methods, which base class is extended, with which additional mixins). A general rule is that if anything can be automatically inferred, it should be to assure that metadata stays in sync with primitive's implementation. One goal of primitive metadata is that it can serve as a readily available and standard source of potential metafeatures for metalearning. By extracting a part of metadata automatically, researchers developing AutoML systems do not have to do that themselves.

There are many standardized metadata fields and the full list is available in Appendix D. Here we describe the most important ones of those which are provided by primitive's author:

id Primitive's universally unique identifier, UUID. It does not change as the primitive changes to allow tracing the evolution of the primitive through time, which can potentially allow metamodels to adapt to new versions without having to retrain from scratch.

installation One of the design goals of the framework is to enable automation as much as possible. To this end primitive metadata contains instructions how can the primitive be installed in a completely automatic manner. Those instructions primarily contain information about the Python package which contains the primitive, but also list any other dependencies, including non-Python dependencies. Moreover, it is also standardized how Python packages containing primitives can be automatically discovered on Python's official package index, PyPi. In this way AutoML systems can automatically discover available primitives and install them.

python_path We provide a mechanism that once the Python package containing the primitive is installed, the primitive is automatically registered under a standardized Python path namespace. **python_path** metadata field specifies the full path under the namespace. This Python path is fully functional and can be used to import a primitive in Python, without having to know the name of the Python package it is installed with. Furthermore, this means that different Python packages can provide primitives while for a user it looks like they are all part of one large package. All this is primarily targeting debugging and easier reading and understanding of pipelines by humans.

To make primitive's Python path even more useful to humans its structure is standardized. A standard Python path consists of three segments following the `d3m.primitives`:

- Primitive family.
- Primitive name segment, from a semi-controlled list of potential primitive names, with the goal of grouping all existing implementations of more or less the same primitive under the same name.

- Kind segment, which allows differentiation between different implementations. E.g., that can be a library name (scikit-learn [33], Keras [7]), the author's name, some special feature (GPU), or a combination of those.

primitive_family, algorithm_types, keywords We provide controlled vocabularies to categorize primitives and open-ended keywords for any additional categories as deemed useful by the author. The primitive family describes the high-level purpose/nature of the primitive. Algorithm types describe the underlying implementation of the primitive. We bootstrapped the vocabulary of those two fields based on English Wikipedia pages and categories related to ML [51].

Parameters

Primitive's parameters (state) are internal parameters of the primitive which are being learned given example inputs and outputs. We require all parameters to be explicitly defined in advance, including their structural types. This helps programming discipline in otherwise dynamically typed programming language Python.

Hyper-parameters

Primitive also defines hyper-parameters. In our framework hyper-parameters are broader than what is usually understood in ML community and are general configuration parameters. Hyper-parameters generally do not change during the lifetime of a primitive. They can be:

- Tuning hyper-parameters. They do not change the logic of the pipeline, but they potentially influence the predictive performance of the primitive. Generally an AutoML system would search for the best configuration of tuning hyper-parameters given a pipeline. Examples: learning rate, depth of trees in random forest, an architecture of the neural network.
- Control hyper-parameters. Changing them generally changes the logic of the pipeline and are determined at the same time with the pipeline itself. Example: whether the primitive should pass through or discard the columns on which it did not operate.
- Hyper-parameters which control the use of resources by the primitive. Examples: number of cores to use, should GPUs be used or not.

A hyper-parameter can belong to more than one of these categories. Besides defining available hyper-parameters, their descriptions in natural language, their default values, and categories to which they belong, primitive's author should also attempt to describe the space of valid values a hyper-parameter can take. We describe our *hyper-parameters configuration* component in Section 3.6.

3.5 Primitive interfaces

Primitives extend a suitable base class and optional mixins. By deciding which base class and mixins to extend, the author of the primitive both communicates the nature of the primitive and assures that required methods have to be implemented, which is assured with use of abstract Python methods.

Base classes

We provide a primary base class `PrimitiveBase` and four main sub-classes from which primitive authors can choose from:

`SupervisedLearnerPrimitiveBase` A base class for primitives which have to learn from examples of both inputs and outputs before they can start producing outputs from inputs. For example, during learning inputs could be features/attributes of known examples and outputs could be known target values. Then at test time, the primitive would be given new features/attributes to produce predicted target values.

`UnsupervisedLearnerPrimitiveBase` A base class for primitives which have to learn from examples before they can start producing (useful) outputs from inputs, but they only learn from example inputs.

`GeneratorPrimitiveBase` A base class for primitives which have to learn from examples before they can start producing (useful) outputs, but they only learn from example outputs. Moreover, they do not accept any inputs to generate outputs, but are only provided how many outputs are requested, and which ones are requested from the potential set of outputs.

`TransformerPrimitiveBase` A base class for primitives which do not learn at all and can directly produce (useful) outputs from inputs. As such they also do not have any parameters (state).

These four main sub-classes cover all combinations of learning the logic of the primitive: from both the inputs and outputs, only inputs, only outputs, and no learning necessary. That is the only difference between them in regards to abstract Python methods: which arguments they take during learning. The rest of their interface is defined by the primary base class, `PrimitiveBase`.

The primary base class, `PrimitiveBase`, defines the following Python methods, some of them abstract:

`__init__(hyperparams, random_seed, volumes, temporary_directory)` Constructor. All primitives accept all their hyper-parameters through a constructor as one value.

Provided random seed should control all randomness used by this primitive. Primitive should behave exactly the same for the same random seed across multiple invocations.

Primitives can also use additional static files which can be added as a dependency to **installation** metadata. When done so, static files are provided to the primitive through `volumes` argument to the primitive's constructor with paths where downloaded and extracted files are available to the primitive. All provided files and directories are read-only. For example, this is used to provide pretrained weights to primitives.

Primitives can also use the provided temporary directory to store any files for the duration of the current pipeline run phase. Directory is automatically cleaned up after the current pipeline run phase finishes.

`set_training_data(inputs, outputs)` Sets current training data of this primitive. For example, inputs could be features/attributes of known examples and outputs could be known target values.

`fit(timeout, iterations)` Learns the parameters of the primitive from input and output examples using currently set training data.

Caller can provide timeout information to guide the length of the fitting process. Ideally, a primitive should adapt its fitting process to try to do the best fitting possible inside the time allocated. The purpose of the `timeout` argument is to give opportunity to a primitive to cleanly manage its state instead of interrupting execution from outside.

Some primitives have internal fitting iterations (e.g., epochs). For those, caller can provide how many of primitive's internal iterations should a primitive do before returning. Primitives should make iterations as small as reasonable. If `iterations` argument is not provided, then there is no limit on how many iterations the primitive should do and primitive should choose the best amount of iterations on its own (potentially controlled through hyper-parameters).

`timeout` and `iterations` arguments can serve to guide the learning process and optimize it for both the predictive performance and time consumption.

`produce(inputs, timeout, iterations) -> outputs` Produces primitive's best choice of the output for each of the inputs.

In many cases producing an output is a quick operation in comparison with `fit`, but not all cases are like that. For example, a primitive can start a potentially long optimization process to compute outputs. `timeout` and `iterations` arguments can serve as a way for a caller to guide this process.

`get_params() -> params` Returns parameters (state) of this primitive.

`set_params(params)` Sets parameters (state) of this primitive.

Primitives can have additional produce methods. They should have the same semantics as the main produce method. Moreover, they should not expose new logic of the primitive, but mostly serve as a way to return different representations of the same result. E.g., a clustering primitive could return a membership map for inputs samples from the main produce method, and a distance matrix between samples as an additional produce method.

All arguments to all methods are *primitive arguments*. Primitive arguments together with all hyper-parameters are seen as inputs to the primitive as a whole, *primitive inputs*. Primitive inputs are identified by their names and any input name must have the same type and semantics across all methods and hyper-parameters, effectively be one value.

`set_training_data` and produce methods can have less or additional arguments that the primary base class, as needed.

Sub-classes of this class (or its sub-classes) allow *functional compositionality*.

Mixins

Mixins are additional classes which can be extended in addition to the main base class and contribute additional methods to the primitive class. We provide two main groups of standard mixins: compositionality mixins and utility mixins.

Compositionality mixins expose methods which enable additional execution semantics of primitives composed into pipelines:

`SamplingCompositionalityMixin` Signals to a caller that the primitive is probabilistic but may be likelihood free. It adds `sample` method, which samples output for each input.

`ProbabilisticCompositionalityMixin` Provides additional abstract methods which primitives should implement to help callers with doing various end-to-end refinements using probabilistic compositionality. It adds `log_likelihoods` method, which returns log probability of outputs given inputs and parameters under this primitive: $\log(p(\text{output}_i|\text{input}_i, \text{params}))$, and `log_likelihood` method, which returns sum $\sum_i(\log(p(\text{output}_i|\text{input}_i, \text{params})))$.

`GradientCompositionalityMixin` Provides additional abstract methods which primitives should implement to help callers with doing various end-to-end refinements using gradient-based compositionality. It adds methods:

- `gradient_output` returns the gradient of loss $\sum_i(L(\text{output}_i, \text{produce\_one}(\text{input}_i)))$ with respect to outputs.
  When fit term temperature is set to non-zero, it returns the gradient with respect to

outputs of:

$$\sum_i (L(\text{output}_i, \text{produce\_one}(\text{input}_i)))$$

$$+$$

$$\text{temperature} \sum_i (L(\text{training\_output}_i, \text{produce\_one}(\text{training\_input}_i)))$$

When used in combination with the `ProbabilisticCompositionalityMixin`, it returns gradient of $\sum_i(\log(p(\text{output}_i|\text{input}_i, \text{params})))$ with respect to outputs.
When fit term temperature is set to non-zero, it returns the gradient with respect to outputs of:

$$\sum_i (\log(p(\text{output}_i|\text{input}_i, \text{params})))$$

$$+$$

$$\text{temperature} \sum_i (\log(p(\text{training\_output}_i|\text{training\_input}_i, \text{params})))$$

- `gradient_params` returns the gradient of loss $\sum_i(L(\text{output}_i, \text{produce\_one}(\text{input}_i)))$ with respect to parameters.
  When fit term temperature is set to non-zero, it returns the gradient with respect to parameters of:

$$\sum_i (L(\text{output}_i, \text{produce\_one}(\text{input}_i)))$$

$$+$$

$$\text{temperature} \sum_i (L(\text{training\_output}_i, \text{produce\_one}(\text{training\_input}_i)))$$

  When used in combination with the `ProbabilisticCompositionalityMixin`, it returns gradient of $\sum_i(\log(p(\text{output}_i|\text{input}_i, \text{params})))$ with respect to parameters.
  When fit term temperature is set to non-zero, it returns the gradient with respect to parameters of:

$$\sum_i (\log(p(\text{output}_i|\text{input}_i, \text{params})))$$

$$+$$

$$\text{temperature} \sum_i (\log(p(\text{training\_output}_i|\text{training\_input}_i, \text{params})))$$

- `forward` is similar to `produce` method but it is meant to be used for a forward pass during backpropagation-based end-to-end training. Primitive can implement it differently

than `produce`, e.g., forward pass during training can enable dropout layers, or `produce` might not compute gradients while `forward` does.

- `backward` returns the gradient with respect to inputs and with respect to parameters of a loss that is being backpropagated end-to-end in a pipeline. This is the standard backpropagation algorithm: backpropagation needs to be preceded by a forward propagation (`forward` method call).
- `set_fit_term_temperature` sets the temperature used in `gradient_output` and `gradient_params`.

Utility mixins expose additional methods which can help AutoML systems and other primitives additional ways of interacting with a primitive:

`ContinueFitMixin` Provides an abstract method `continue_fit` which is similar to `fit`, but the difference is what happens when currently set training data is different from what the primitive might have already been fitted on. `fit` refits the primitive from scratch, while `continue_fit` fits it further.

`LossFunctionMixin` Provides abstract methods for a caller to call to inspect which loss function a primitive is using internally, and to compute loss on given inputs and outputs.

`NeuralNetworkModuleMixin` Provides an abstract method `get_neural_network_module` for connecting neural network modules together. These modules can be either single layers, or they can be blocks of layers. The construction of these modules is done by mapping the neural network to the pipeline structure, where primitives (exposing modules through this abstract method) are passed to followup layers through hyper-parameters. The whole such structure is then passed for the final time as a hyper-parameter to a training primitive which then builds the internal representation of the neural network and trains it.

`NeuralNetworkObjectMixin` Provides an abstract method `get_neural_network_object` which returns auxiliary objects for use in representing neural networks as pipelines: loss functions, optimizers, etc.

We will discuss the use of `NeuralNetworkModuleMixin` and `NeuralNetworkObjectMixin` mixins in more detail in Section 4.4.

3.6 Hyper-parameters configuration

We provide a framework's component to describe the configuration of hyper-parameters a primitive has. The configuration is essentially a mapping between hyper-parameter names and *hyper-parameter definitions*. Each hyper-parameter definition is an instance of a Python

class and among other properties defines *hyper-parameter space*. A hyper-parameter space defines valid values a hyper-parameter can take. We also provide a standard representation of the hyper-parameters configuration in JSON serialization format.

```
class HyperparamsConfiguration(Hyperparams):
    tolerance = Bounded[float](
        default=0.0001,
        lower=0,
        upper=None,
        lower_inclusive=True,
        description="Tolerance for stopping criteria.",
        semantic_types=['https://metadata.datadrivendiscovery.org/types/TuningParameter'],
    )
```

Figure 3.3: An example hyper-parameters configuration.

Figure 3.3 shows an example hyper-parameters configuration. It contains one hyper-parameter, named `tolerance`, with hyper-parameter definition being an instance of class `Bounded`, which is a class corresponding to a hyper-parameter space which is an interval, bounded on at least one side, but without known distribution. In this example, structural type of values of the interval is `float`. Moreover, interval does not have the upper bound and the lower bound is inclusive. Hyper-parameter definition includes a description in natural language and is categorized as a tuning hyper-parameter.

We use a Python class to define a hyper-parameter and its space instead of a static structure to allow different space definitions to also provide default methods to navigate the space. This allows a primitive author to define a non-standard space definition for a hyper-parameter, while still being compatible with the framework and AutoML systems using the framework. For example, a primitive could define a custom hyper-parameter class defining a space of all neural networks it knows how to use and provide hyper-parameter methods to sample from this space.

Every primitive has a hyper-parameters configuration associated with it. There are four ways to provide values for those hyper-parameters:

- As a constant value in the pipeline. Useful for control hyper-parameters.
- As a value provided to the runtime during execution of the pipeline. Used to try different sets of values of tuning hyper-parameters for same pipeline.
- As a value computed in prior steps in the pipeline.
- As a primitive itself of a prior step in the pipeline. This is useful to support meta-primitives: primitives which take other primitives as a hyper-parameter and use them. E.g., a primitive can run another primitive over each cell in a column, mapping input cell values to output cell values.

The main methods of hyper-parameter definition class are:

`__init__(default, semantic_types, description, ...)` Constructor. Used to provide static parameters of the instance, including the default value, hyper-parameter categories, and a natural language description.

`validate(value)` Validates that a given value belongs to the space of the hyper-parameter.

`sample(random_state)` Samples a random value from the hyper-parameter search space.

`sample_multiple(min_samples, max_samples, random_state, with_replacement)` Samples multiple random values from the hyper-parameter search space. At least `min_samples` of them, and at most `max_samples`.

We list standard hyper-parameter definition base classes in Appendix H.

3.7 Basic data types

The framework defines and provides basic data types, reusing Python data types and data types provided by popular Python libraries. Basic data types are organized as follows:

- Container types: NumPy `ndarray`, Pandas `DataFrame`, `List`, `Dataset`
- Simple data types: `str`, `bytes`, `bool`, `float`, `int`, NumPy numerical values
- Data types: all container types, all simple data types, and `dict`

Container types are types of values which are passed between pipeline steps. Data types can be contained inside container types, or themselves, recursively. The main motivation behind limiting container types to only a limited set of data types is to make interoperability between primitives easier, without the need for potentially costly or lossy conversions between data types.

While simple data types and `dict` are regular Python and NumpPy data types, container types are designed and implemented specifically for the framework. `ndarray`, `DataFrame`, and `List` container types extend their respective standard data types with support for *metadata* (more about metadata in Section 3.9). Moreover, they have additional methods for manipulation of both data and metadata at the same time, and their constructors have been extended to make it easier to convert between all container types in a reasonable way, especially taking into consideration cases when container types are contained (nested) inside container types, recursively.

In addition, we provide a container type named `Dataset`.

Dataset container type

One of the design goals of the framework is to allow expression of complete end-to-end ML programs, starting with raw files. To achieve this we designed and implemented a specialized `Dataset` container type to serve as a starting point to a pipeline and which can represent a wide range of input data: tabular data, graph data, time-series data, tabular data referencing media files, raw files without known structure, etc.

`Dataset` container type is conceptually simple, but its combination with metadata (see Section 3.9) makes it powerful and expressive. It is implemented as a Python `dict`, mapping resource IDs to resources. Resources can be any other container type, but most often resources are `DataFrame`s. Metadata associated with the `Dataset` container type provides additional information useful to understand the data stored in resources: are there foreign keys between resources, is a resource representing a collection of raw files and where can those raw files be found stored, how many dimensions does a resource have, has tabular resource any special properties like time-series column, or does it represent a graph in edge-list structure, etc. Not all metadata is necessarily available at `Dataset` loading time, but primitives in a pipeline can try to infer missing metadata and augment it, for later primitives to use.

We also provide support for extensible loaders and savers which allow users to load `Dataset`s from different storage representations, and save `Dataset`s as well. All this makes `Dataset` container type a suitable standard data input to pipelines.

3.8 Data references

To represent the available data sources in a pipeline, we use *data references*. A data reference is a string with standardized structure to make it easier to debug:

- `inputs.X` represents pipeline inputs, with `X` corresponding to the index of the pipeline's input
- `steps.X.NAME` represents a step's outputs, with `X` corresponding to the step index and `NAME` corresponding to the name of the step's output (for a primitive step this is the name of the primitive's produce method)
- `outputs.X` represents pipeline outputs, with `X` corresponding to the index of the pipeline's output

In pipelines, data references are used to identify which data source should be connected to a step's input, forming a data-flow connection.

3.9 Metadata

Besides data, container types also contain metadata. Metadata is stored as an attribute on a container type value in a specialized metadata object we designed and implemented.

Primitives can use both input data and metadata in their logic and return both data and metadata as a result, as one container type value. Metadata object is designed to be independent from a particular container type implementation, but at the same time interoperable with all of them. This is achieved through *selectors*. A selector matches in a general way a subset of data, a scope, for which a *metadata entry* applies. Full metadata is then a series of (selector, metadata entry) pairs. Each metadata entry is a mapping between *metadata fields* and *metadata values*.

Primitive metadata is stored using the same metadata object, but it does not use or need selectors and it is stored as only one metadata entry, because primitives do not have data structure.

Selectors

Selectors declare to which part of the data a metadata entry applies by matching on the structure of the data itself. A selector consists of a series of *segments*, in the order of dimensions in the data structure, across contained (nested) data types. E.g., selector can match inside a `DataFrame` which is by itself a cell value of another `DataFrame`. The order of dimensions is defined for each data type:

- `ndarray`: following `ndarray` dimensions order itself
- `DataFrame`: first rows, then columns
- `List`: has only one dimension
- `Dataset`, `dict`: has only one dimension

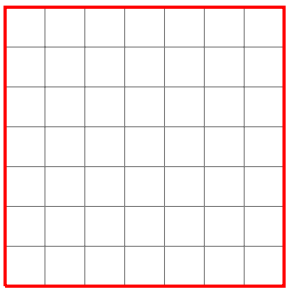
Each segment matches value or values of the corresponding dimension for which the metadata entry applies. Segments can be:

- A numerical index, used with `ndarray`, `DataFrame`, and `List` dimensions.
- A mapping key, used with `Dataset` and `dict` dimensions.
- A special wildcard value `ALL_ELEMENTS` which applies to all values of a dimension.

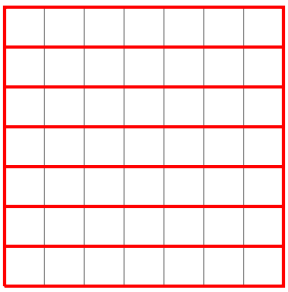
Currently there is only one special segment value defined, `ALL_ELEMENTS`. Selectors apply in a *cascading* manner: a priority scheme is defined to determine which metadata values apply if more than one selector matches a particular part of the data with overlapping metadata fields. For non-overlapping metadata fields, metadata values are formed as a union over all metadata entries applicable to a particular part of the data.

This cascading priority scheme is predictable. A general rule is that more specific selectors have precedence. Currently, this means that numerical index and mapping key segments have precedence over `ALL_ELEMENTS` segment.

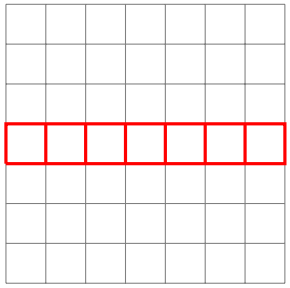
Metadata object provides low-level methods for updating and querying metadata and additional higher-level methods for common use cases, e.g., updating and querying metadata of tabular data. A metadata object is immutable.



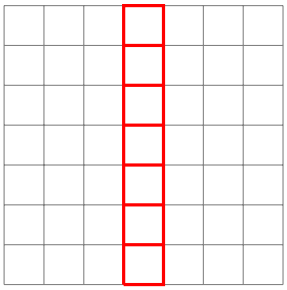
`()`
Select container itself.



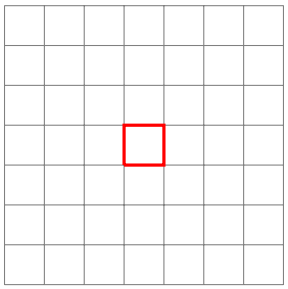
`(ALL_ELEMENTS)`
Select all rows.



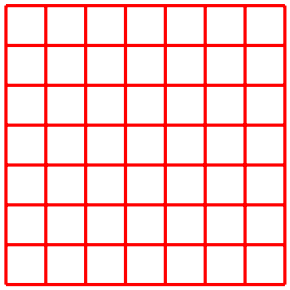
`(3, ALL_ELEMENTS)`
Select all cells in the fourth row.



`(ALL_ELEMENTS, 3)`
Select all cells in the fourth column.



`(3, 3)`
Select a single cell.



`(ALL_ELEMENTS, ALL_ELEMENTS)`
Select all cells.

Figure 3.4: Visual representation of example metadata selectors on tabular value.

Figure 3.4 visually shows example metadata selectors and how they match parts of tabular data.

Metadata entries

Each metadata entry is a mapping between *metadata fields* and *metadata values*, implemented as a Python `dict`. To facilitate interoperability between primitives, we standardized common metadata fields and values using JSON Schema, but metadata entries can also contain other custom fields and values. Appendix E lists top-level metadata fields at the container type level, and Appendix F lists top-level fields metadata for data inside the container type.

Among standard metadata fields are those which describe the structure of data (dimensions, number of rows, columns, etc.), other properties of data (e.g., sampling rate for audio

data), metadata targeting humans (column names and descriptions of data), and fields to record data provenance. Moreover, there is an extensive list of standardized metafeature fields to describe various properties of data potentially useful for metalearning. Standard metadata also provides typing information about data: *structural types* and *semantic types*, so one can understand from the metadata what the structure of the data is, how it is represented as Python types (structural types), and what the meaning or use of the data is (semantic types).

Semantic types

Semantic types are an important part of metadata and expose implicit information often part of variable names or comments in programming languages. For example, the program in Figure 3.1 has a variable named `categorical_attributes`. To a human reading the source code this variable name means something and makes it easier for the human to understand the program. But the program itself has no access or use of this information.

In the framework such information is explicit through semantic types. Every part of data (through metadata selectors) can have a set of relevant semantic types applied to it. E.g., `DataFrame` columns corresponding to `categorical_attributes` variable would have `https://metadata.datadrivendiscovery.org/types/Attribute` and `https://metadata.datadrivendiscovery.org/types/CategoricalData` semantic types set. Semantic types facilitate multiple use cases:

- We can encode human knowledge about input data in a machine readable way.
- Primitives can automatically decide on which parts of data to operate, without needing to slice or combine data first. We will expand on this in Section 4.2.
- Primitives can communicate to later primitives in the pipeline semantic information about data by setting or removing semantic types, guiding logic of later primitives. This enables decoupling data analysis from actions based on the analysis. E.g., one primitive can detect which columns are likely to be categorical and a later primitive can then one-hot encode them, or a different later primitive can instead encode columns into ordinal integers.

List of commonly used semantic types are available in Appendix G. Besides those, any other URI representing a semantic type can be used.

3.10 Execution semantics

Execution semantics of a pipeline is on purpose decoupled from the rest of the framework, including pipeline structure. Together with extensible nature of primitives and their interfaces (through standard and non-standard mixins) this allows experimentation in pipeline execution while reusing framework's components. Moreover, sub-pipelines provide a reasonable

abstraction layer and different execution semantics can be tied to individual sub-pipelines while retaining overall interoperability.

Standard execution semantics

We have designed a standard execution semantic named *fit-produce*. It executes the pipeline in a data-flow manner in two phases: fit and produce. The run of each phase of a pipeline execution proceeds in order of its steps. All values passed between steps are defined immutable. Initially, at the start of each phase, only pipeline inputs are available as inputs to steps. After each step is executed, its outputs are added to values available to later steps, and to values which can be used as pipeline outputs. It is required that steps are ordered in the pipeline and that all step outputs are defined in a way that when executing steps in order, all necessary step inputs are always available before they are needed, forming a DAG. Fit phase is usually run on training data and produce phase on testing data.

Fit phase, primitive step The following primitive methods are called in order:

1. `__init__`, passing:
   - an instance of primitive's hyper-parameters configuration class, which is a mapping between hyper-parameter names and their values,
   - deterministically computed random seed value for this primitive, based on the main random seed and the pipeline structure,
   - `volumes` and `temporary_directory` constructor arguments.
2. `set_training_data`, passing only those primitive arguments as method arguments that the method accepts.
3. `fit`, without any method arguments.
4. For every specified primitive output, produce method with corresponding name is called, passing only those primitive arguments as method arguments which the method accepts. The output of the produce method becomes the corresponding output of the primitive. Same input values passed to `set_training_data` are passed for same arguments once more to produce methods.

Fit phase, sub-pipeline step Fit phase of the sub-pipeline step is run, mapping step inputs to sub-pipeline inputs, and sub-pipeline outputs to step outputs.

Produce phase, primitive step For every specified primitive output, produce method with corresponding name is called, passing only those primitive arguments as method arguments that the method accepts. The output of the produce method becomes the corresponding output of the primitive.

Produce phase, sub-pipeline step Produce phase of the sub-pipeline step is run, mapping step inputs to sub-pipeline inputs, and sub-pipeline outputs to step outputs.

Placeholder steps are not allowed during execution. Each pipeline output have a corresponding step output which is at the end of a phase passed on as the output of the pipeline itself.

Parameters (state) of primitives are updated during fitting in the fit phase, but do not change anymore in the produce phase. `set_training_data` and `fit` are called even on primitives without parameters (state), but the methods do nothing. Note that both phases are run on the same pipeline structure, only input data is potentially different between phases. Moreover, input data structure generally stays the same for both phases (e.g., which columns exist in data), with differences only in amount of data and contents.

When step is a primitive step, step inputs are primitive inputs. Recall that primitive inputs are primitive arguments and all hyper-parameters, and furthermore that all primitive arguments are all arguments to all primitive methods combined. Passing primitive arguments to method arguments is straightforward and follows usual method calling semantics, because values can be only container types. We do have to take care we pass only those method arguments that the method really accepts, which we do through introspection at runtime. An exception here is support for variable number of values connected to the same primitive argument. In this case we make a list from all values, before passing the list to the primitive as one value for the primitive argument.

Passing hyper-parameters to primitive's constructor is more involved because we have to handle the case when a primitive instance is passed to another primitive as a hyper-parameter value. Primitive instance can come from a constant value or from a prior step in a pipeline. In both cases we have to make a clone of the primitive to assure that every primitive instance is independent, not sharing any state. Because primitive state is explicit (primitive parameters) we can achieve such cloning through `get_params` and `set_params` methods.

Another aspect of passing primitive instances as hyper-parameters is that primitive instances can be in a fitted or unfitted state. For constant values, the primitive instance is what it is. But when primitive instance comes from a prior step in a pipeline, how do we know if it should be first fitted or not, before being passed on as a hyper-parameter? To address this we have an exception: if a primitive in a pipeline has no primitive arguments connected, then during pipeline execution such primitive is not fitted nor produced, but just instantiated and then passed on as a hyper-parameter value to another primitive, as a unfitted primitive.

Alternative execution semantics

Execution semantics is decoupled from the rest of the framework, so pipelines or sub-pipelines can be executed using non-standard execution semantics. For this to be possible steps, especially primitives, might have to implement additional mixins.

For example, if all primitives in a pipeline extend `GradientCompositionalityMixin`, this makes a pipeline differentiable end-to-end across primitives implemented in different ML frameworks. A differentiable pipeline enables an alternative execution semantics where instead of doing a fit and produce phases, we could do multiple forward and backward passes, training primitives through backpropagation. Moreover, we could combine this with standard execution semantics and first do a standard fit phase on training data and then continue with forward and backward passes on another set of data, to refine a previously fitted pipeline [30].

Another example of alternative execution semantics is support for batching during pipeline execution. Standard execution semantics requires in-core execution, where values being passed between steps have to fit into memory. For large datasets this might be a prohibitive restriction. A solution could be batching: splitting input dataset into smaller batches of data and then instead of doing one fit phase and one produce phase, we can do both phases for each batch, incrementally training primitives. This could be done if all primitives in a pipeline extend `ContinueFitMixin` and their logic operates on each input sample independently.

Currently all pipeline steps are executed in the order in which they are specified, but an alternative execution semantics could determine which steps can be run in parallel, especially because generally pipeline steps do not have side-effects, and run them in parallel.

3.11 Example pipeline

Visual representation of an example pipeline is available in Figure 3.5. The example pipeline is logically equivalent to the ML program in Figure 3.1 and is available in YAML serialization format in Appendix J.

First we convert a `Dataset` object to a `DataFrame` object. We can do this in a straightforward manner because the `Dataset` contains only one resource, a tabular `DataFrame` object. After that we extract numerical and categorical attributes, and then the target column, too. When loading `Dataset` objects no automatic parsing of strings of any kind is done and are values are kept as strings. Because of that we have to parse numerical attributes using an explicit primitive, after which we can impute numerical attributes. We also encode categorical attributes and combine them with numerical attributes, which concludes all preprocessing of attributes. We then build a random forest model. The final step is necessary to restore the row index column which is a required part of standard predictions output of a *standard pipeline*. We will discuss standard pipelines in Section 4.1. Random forest primitive does not preserve the index column by default for compatibility with scikit-learn behavior, on which it is based.

In our example the `Dataset` comes with semantic information for each column and we use it when extracting columns so that we do not have to hard-code column indices into the pipeline. If such semantic information was missing, we could have used a primitive to infer it. Using such primitive to infer semantic types is an example how populating

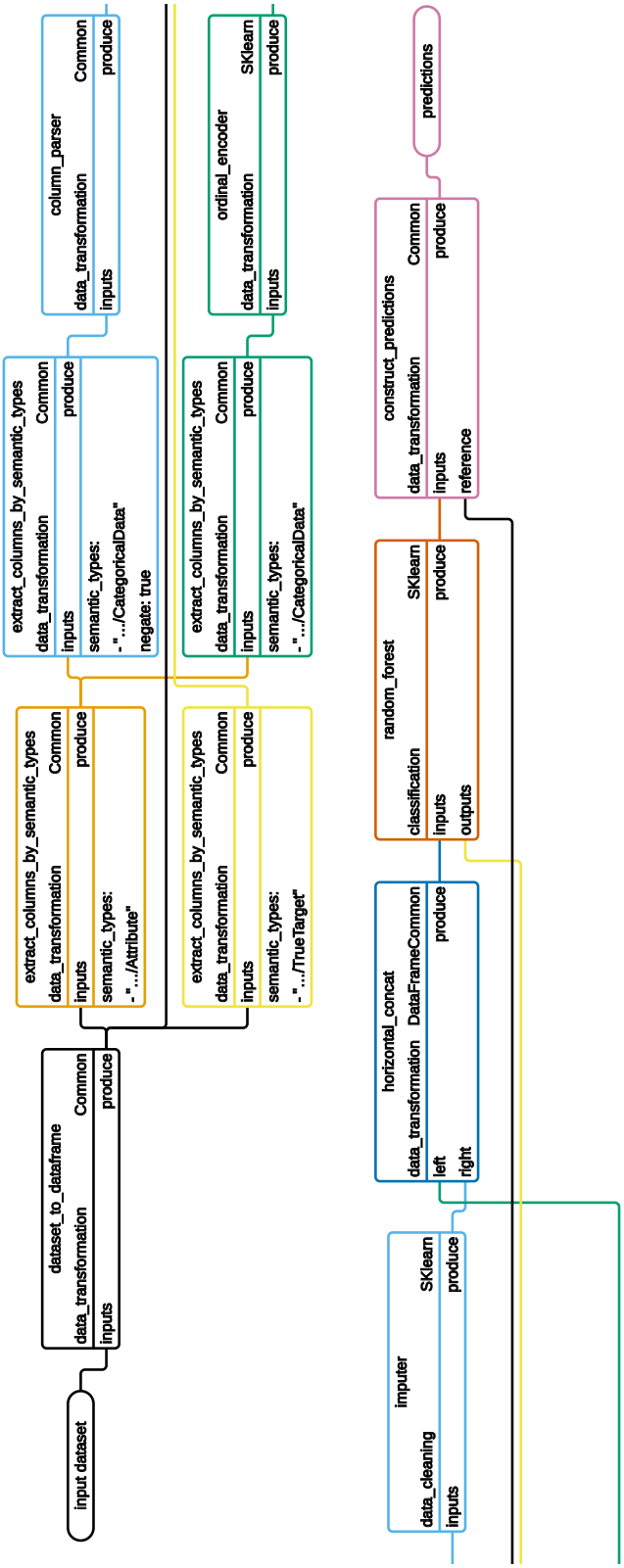


Figure 3.5: Visual representation of an example pipeline. It is available in YAML serialization format in Appendix J.

with semantic information can be decoupled from acting on this information (in our case extracting columns).

Alternatively, we could instead extract columns by their structural type or dtype, like the ML program in Figure 3.1 does, but this is error-prone and fragile in the AutoML context because it is hard to control or predict when and how will low-level data operations in primitives change them, sometimes just as a side-effect of an operation. Semantic types are independent from data operations and do not have these shortcomings.

Observe high branching in the pipeline. We will discuss implications of branching in Section 4.2.

In this example pipeline, many primitives are transformer primitives (extending `TransformerPrimitiveBase` base class) without any parameters to learn. For some primitives we set control hyper-parameters in the pipeline to guide their behavior, but there is no difference in their behavior between the fit phase and produce phase. Primitives for parsing, imputing, and encoding as unsupervised learner primitives (extending `UnsupervisedLearnerPrimitiveBase` base class), learning during the fit phase the properties of training data, updating their parameters. After fitting in fit phase, and in whole produce phase, those parameters are then used to guide the logic of primitives when they produce outputs given input data. A supervised learner primitive, random forest (extending `SupervisedLearnerPrimitiveBase`) learns from example attributes and targets during the fit phase and produces its own best predicted targets afterwards, during fit phase it produces predicted targets on training data and during produce phase it produces predicted targets on testing data.

The example pipeline in Figure 3.5 and Appendix J has only non-default hyper-parameter values set as part of the pipeline description. The full set of hyper-parameters defined by all primitives and their values in effect can be seen in Figure 3.6.

3.12 Problem description

Main use of a problem description in an AutoML system is to guide the system towards solving a meaningful problem given data. Our problem description is currently simple and contains:

- Task type and subtype categories with controlled vocabularies to categorize problems into a wide range of tasks, beyond just basic classification and regression.
- Which performance metrics the AutoML system should optimize for.
- Which columns in a dataset are target columns.
- A list of privileged data columns related to unavailable attributes during testing. These columns do not have data available in the test split of a dataset.
- Additional metadata like human-friendly name and description.

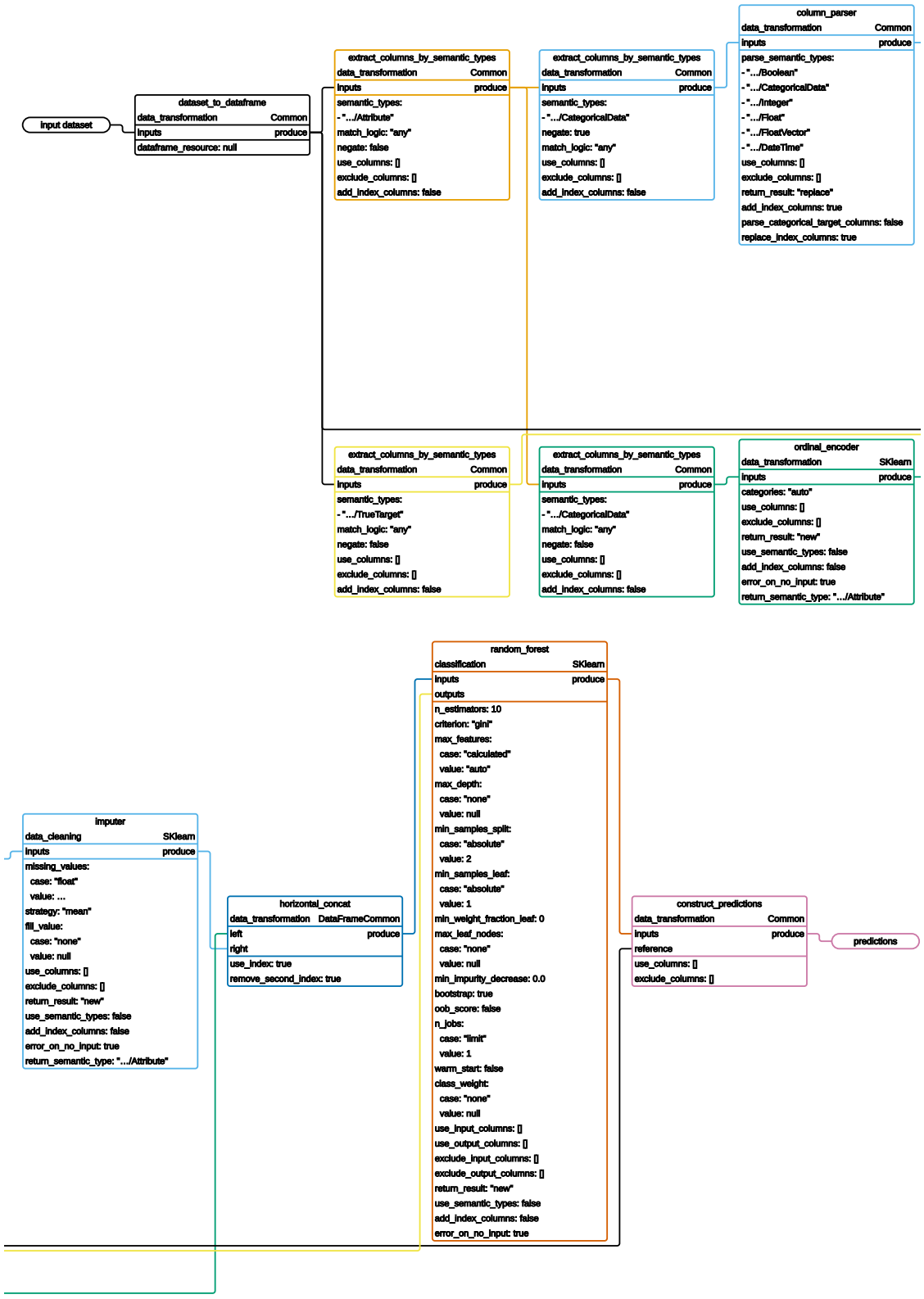


Figure 3.6: Visual representation of the example pipeline with all hyper-parameter values shown.

A problem description is fully available to an AutoML system, but to the pipeline is exposed through semantic types: target columns and privileged data columns are marked as such on input `Dataset` before it is passed into the pipeline during execution.

We provide a validator for problem descriptions using JSON Schema. The full list of standardized top-level fields of problem descriptions is available in Appendix C.

3.13 Reference runtime

We provide a reference runtime implementation. Furthermore, it is designed in an extensible manner allowing AutoML systems to directly integrate it and use it as their primary runtime for pipelines. We provide a command line interface (CLI) to use the reference runtime outside of an AutoML system.

Reference runtime is an interpreter for pipelines and executes pipelines according to the fit-produce execution semantics. Reference runtime ties together all steps necessary for pipeline execution: input data loading into `Dataset` objects, problem description loading and marking of target and privileged data columns, and executing the pipeline itself.

3.14 Evaluating pipelines

In addition to generating and running pipelines describing ML programs, AutoML systems have to be able to evaluate those pipelines and score them using relevant performance metrics. The design of the framework supports describing evaluation of pipelines using the framework itself, through two additional pipelines:

- A data preparation pipeline prepares input data for evaluation, primarily by splitting data into multiple parts, supporting various approaches to evaluation: train/test splits, k-fold cross validation, etc. It also redacts any target values in test splits to assure valid evaluation. Because data preparation pipeline is just a regular pipeline, it can contain any primitives and do any other data preparation steps. A data preparation pipeline is executed once for input data and produces potentially multiple splits or folds of input data.
- A scoring pipeline takes outputs of executing a pipeline on one split or fold of input data and computes scores using performance metrics listed in a problem description. We provide a standard scoring pipeline supporting all standard performance metrics, but a custom scoring pipeline can be used as well.

Reference runtime supports such evaluation of pipelines and provides all necessary logic to execute all three pipelines properly together. Because evaluation if done through the utilization of the framework we inherit all its useful properties, e.g., full reproducibility, decoupled design, compositionality, support for raw files, etc.

Having a standard framework for ML programs, a reference runtime, and a standard way to evaluate pipelines allows us to record all pipelines, their scores, and everything else about the execution of a pipeline in a standard way as well, enabling metalearning over pipelines and results across multiple AutoML systems.

3.15 Metalearning

One of the design goals of the framework is to allow automatic consumption of pipelines. The motivation behind this goal is that we want pipelines to serve for metalearning purposes. The framework defines a standard description for ML programs, pipelines, and a standard way to evaluate pipelines.

In addition, we have designed and implemented also a standard way of recording detailed information about the execution of a pipeline, a *pipeline run description*. A pipeline run description contains information which pipeline was executed for which problem description and input data, using which values for tuning hyper-parameters. It records information about any data preparation, evaluation, and scoring which was done. It contains information about execution environment and timing information about execution of every part of the pipeline, to the level of method calls. Moreover, it records metadata of all step outputs produced during pipeline execution. The full list of standardized top-level fields of pipeline run descriptions is available in Appendix I.

Because pipeline run descriptions record metadata of all step outputs, they record also all metafeatures primitives might compute about data passing through the pipeline, at various steps and not just at the pipeline input.

With all this information we have everything to build a metalearning dataset: we have a standard machine readable way to represent pipelines, problem descriptions, input data, data metafeatures, and scores.

To enable building such metalearning dataset, we have designed a metalearning database, where all this information can be stored and shared between various AutoML systems. A goal of this database is that anyone can record a pipeline run description of an execution of their pipeline, using any data preparation pipeline and any scoring pipeline, on any data. Furthermore, to facilitate easier comparison between pipeline run descriptions, we provide standard data preparation and scoring pipelines.

Once a pipeline run description for a given pipeline is submitted to the database, anyone else can re-run it and try to reproduce it. No matter the result, reproduction or not, the new pipeline run description can be submitted as well, expanding the understanding of the pipeline.

Chapter 4

Pipelines in practice

In this chapter we explore various aspects of the framework in practice.

4.1 Standard pipelines

The framework is by design very general, to allow most of ML and data processing programs to be described through pipelines.

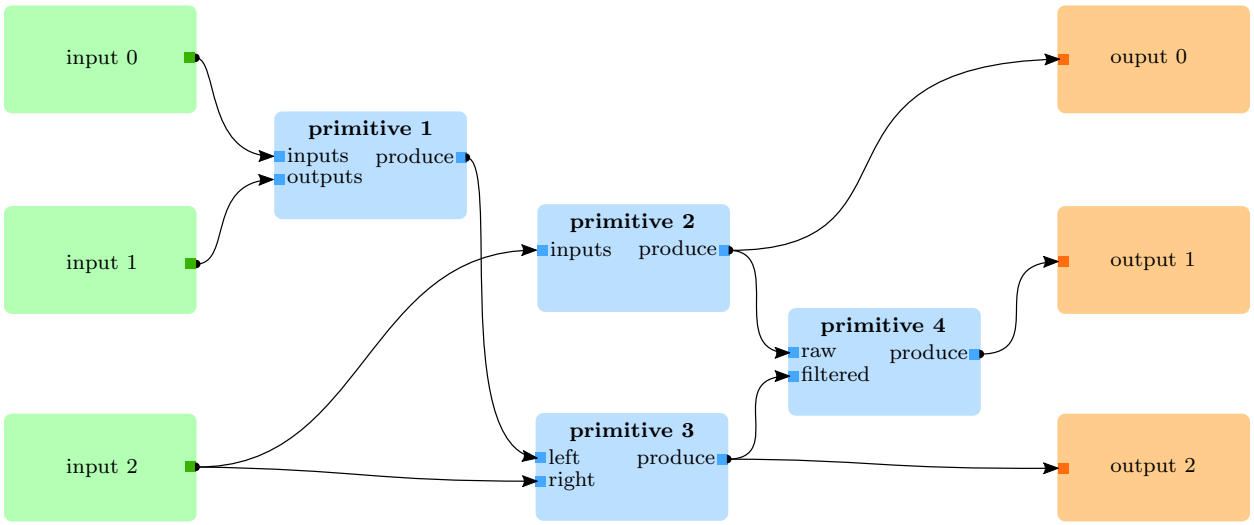


Figure 4.1: A conceptual representation of a general pipeline (DAG).

Figure 4.1 shows a conceptual representation of a general pipeline. With multiple pipeline inputs and outputs it connects primitives as a DAG. Inputs and outputs can be anything supported by the framework. While this is powerful, and is used for special pipelines like data preparation pipelines for evaluation, it is unnecessary for pipelines for many ML problems.

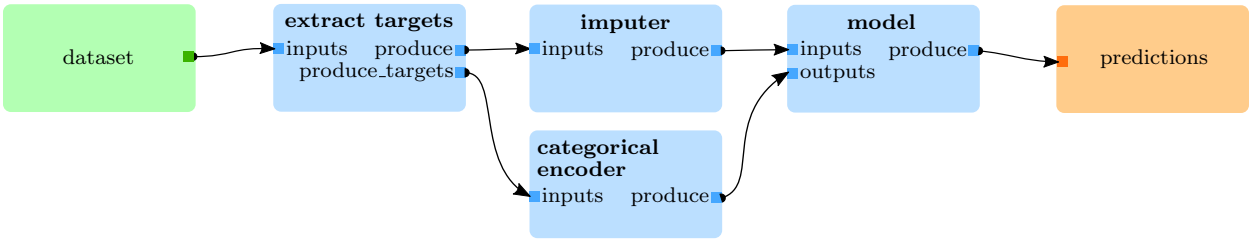


Figure 4.2: A conceptual representation of a standard pipeline.

Instead we standardized a standard pipeline, depicted in Figure 4.2. A standard pipeline has as the pipeline inputs one (or more, but often only one) `Dataset` objects and should return a `DataFrame` with predictions. Standardizing the pipeline in this manner also allows us to use standard data preparation and scoring pipelines, Moreover, it generally makes it easier to compare pipelines and integrate them with other systems. The metalearning database we described in Section 3.15 stores just standard pipelines and their pipeline run descriptions.

4.2 Linear pipelines

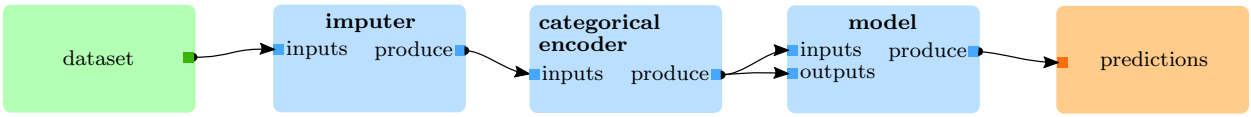


Figure 4.3: A conceptual representation of a linear pipeline.

The example pipeline visually represented in Figure 3.5 and available in YAML serialization format in Appendix J is logically equivalent to the ML program in Figure 3.1. It is a standard pipeline, but observe high branching in the pipeline which follows the flow of data in the example ML program. We observe that there are two main reasons for branching of flow of data in ML programs:

- Slicing data to effectively select the data for a function to operate on, and then combining results back with the rest of the data.
- To use multiple models and combine them for ensemble learning.

We will show how we can address both of those using the framework to form linear pipelines as depicted in Figure 4.3.

It is important to keep in mind that the framework supports highly branching pipelines and that there is nothing inherently wrong with them and that for some ways of generating pipelines (e.g., generating them as linear snippets and then combining them to form a branched final pipeline) this could be the most suitable structure. Moreover, in branching pipelines it might be easier to discover opportunities for parallelization. Here we want to demonstrate the power of the framework to express linear pipelines.

Slicing and combining data

Human developers use their background knowledge and knowledge about a particular function (often from its documentation) to slice the data correctly and then combine results back. This means that an AutoML system needs to have or obtain this knowledge of how to slice and combine data for every function, given data.

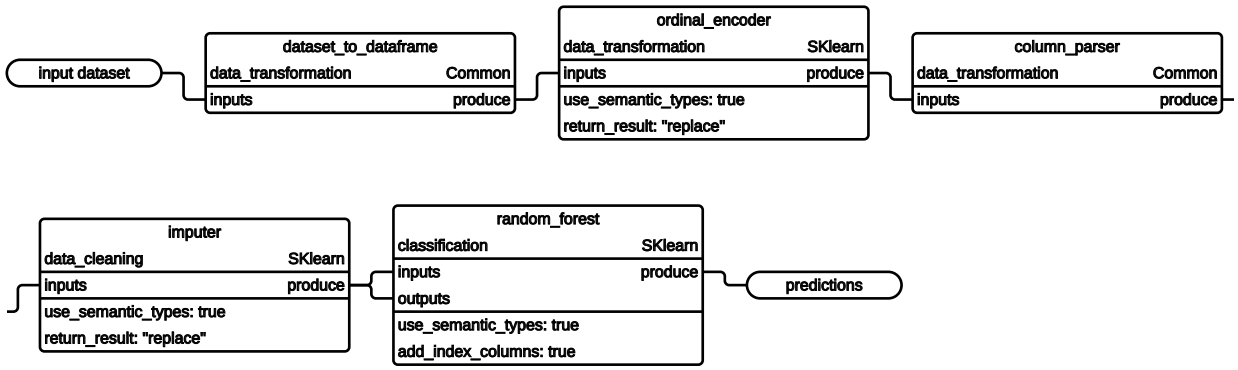


Figure 4.4: Visual representation of an example linear pipeline. It is available in YAML serialization format in Appendix K.

If instead primitives themselves contain logic for slicing and combining results, pipelines can be simplified to linear pipelines, as seen in an example linear pipeline in Figure 4.4, available in YAML serialization format in Appendix K. The example linear pipeline is logically equivalent to the pipeline in Figure 3.5.

Primitives can do this because the logic in most cases depends on the primitive itself. Moreover, primitive authors are the most suitable to encode this knowledge of slicing and combining into the primitive itself. Such change can be seen just as a different abstraction, especially if knowledge of slicing and combining was available in a machine readable description which would mean that we could in an algorithmic way add necessary primitives for slicing and combining around the primitive in question. But the change offers additional benefits:

- Combined with metadata associated with data in the pipeline, it allows prior primitives to influence slicing and combining logic of later primitives by modifying accordingly

metadata prior primitives produce. This allows decisions about slicing and combining to be done not just based on current data available to the primitive, but also from previous states of data, which might not even be available anymore. In a way, metadata allows prior primitives to pass standardized state to later primitives and later primitives can use it to make slicing and combining decisions.

- It can be seen as an abstraction on top of any machine readable descriptions describing rules for slicing and combining. In this way different primitives can use different ways of internally specifying how and what to slice and combine, some using static machine readable descriptions, but others also more sophisticated dynamic approaches. An AutoML system does not have to know about those to be able to use a primitive in a linear pipeline.

- It better aligns separation of concerns between an AutoML system and primitives, especially because prior primitives can influence behavior of later primitives through metadata. An AutoML system can focus on generating a sequence of primitives and influencing which columns to operate on can be done through inclusion or not of primitives setting metadata. This allows an AutoML system to have only one search space. For example, an AutoML system can decide to include or not, and where, a primitive which marks all newly engineered attributes from a series of primitives as the only attributes in effect, disabling the original attributes. Otherwise an AutoML system would have to track information about new attributes during pipeline construction and generate necessary primitives to slice data accordingly. Which would also mean that it would have to know which primitives are engineering new attributes.

- Having support for slicing and combining in primitives does not prevent AutoML systems to take control and still manage slicing and combining themselves, through primitives in a pipeline or through hyper-parameters for this purpose.

- Many linear pipelines with primitives supporting linear pipelines just work: maybe a primitive is unnecessary at a given location in the pipeline and it simply does not do anything, not finding anything suitable in the input data, but will generally not break the pipeline. This allows easier bootstrapping of pipeline generation, allowing as well that researchers focus on different AutoML questions and not necessary tackle head-on all of the challenges of generating sophisticated pipelines.

- Pipelines themselves become simpler and shorter. The pipeline in Figure 3.5 has 11 steps. The pipeline in Figure 4.4 has 5 steps.

Currently, the standardized approach to linear pipelines focuses on primitives which operate on tabular data and where slicing and combining is on columns. But the same principle could be generalized to other types of data. Moreover, in practice it turns out that in most cases use of semantic types and structural types is enough to decide on which

columns to operate, but any other metadata or even concrete data values could be used by primitives to make the decision.

The following standard hyper-parameters control this behavior:

use_columns A set of column indices to force primitive to operate on. If any specified column cannot be used, it is skipped.

exclude_columns A set of column indices to not operate on. Applicable only if **use_columns** is not provided.

return_result Should resulting columns be appended, should they replace original columns, or should only resulting columns be returned?

add_index_columns Also include primary index columns in the output data if input data has them? Applicable only if **return_result** is set to return only resulting columns.

error_on_no_columns Throw an exception if no columns are to be operated on. Otherwise issue only a warning, allowing pipeline not to fail if it has an unnecessary primitive.

Some primitives use an additional hyper-parameter **use_semantic_types** to enable or disable this behavior primitive-wise. Primitives with multiple primitive arguments can also have multiple pairs of **use_columns** and **exclude_columns** hyper-parameters, a pair for each primitive argument.

Primitive authors can use utility functions we provide to simplify implementation and assure its standard behavior. Utility functions decide on columns to operate on, slice input columns, and provide just sliced columns to the internal logic of the primitive. Afterwards, they combine internal resulting columns with input columns for final results. Hyper-parameters **use_columns** and **exclude_columns** allows an AutoML system to fine-control on which columns a primitive should operate. By default a primitive operates on all columns on which it can operate.

Ensemble learning

Ensemble learning combines multiple models to obtain final prediction. We will show how a linear pipeline can support stacking and cascading.

Primitives for supervised learning generally use `https://metadata.datadrivendiscovery.org/types/Attribute` semantic type to decide which input columns are attributes and `https://metadata.datadrivendiscovery.org/types/TrueTarget` semantic type to decide which column has known target values. They produce a column of predicted target values, with, by default, semantic type `https://metadata.datadrivendiscovery.org/types/PredictedTarget`. This can be controlled using standard hyper-parameter **return_semantic_type** to choose another semantic type.

Stacking can be implemented as a linear pipeline by having a series of supervised learning primitives one after the other in a pipeline, each appending their produced column to

the input data. Each of those primitives operates on the same set of attribute columns and generates a new column. After all supervised learning primitives, we add a primitive which modifies semantic types on columns: it removes semantic type `https://metadata.datadrivendiscovery.org/types/Attribute` from all columns, and replaces all cases of semantic type `https://metadata.datadrivendiscovery.org/types/PredictedTarget` with semantic type `https://metadata.datadrivendiscovery.org/types/Attribute`. After that the final supervised learning primitive is used to produce final predictions.

Cascading can be implemented in a similar way: a series of supervised learning primitives one after the other in a pipeline, each appending their produced column to the input data. The difference is that we use **return_semantic_type** hyper-parameter to configure primitives to set `https://metadata.datadrivendiscovery.org/types/Attribute` semantic type on every produced column. This means that every later supervised learning primitive uses predictions of all prior supervised learning primitives as its attributes. The final learning primitive is used to produce final predictions.

4.3 Reproducibility of pipelines

Control of side-effects and randomness, and full reproducibility is one of the design goals of the framework. Having a pipeline produce same predictions every time is important for successful metalearning. Otherwise unnecessary noise is added to the metalearning process.

Assuring complete reproducibility is hard, especially in a programming language like Python, which allows side-effects and where many standard modules and 3rd party libraries use global variables to influence their behavior.

Here we list some known issues and potential solutions to improve reproducibility.

- Python uses by default hash randomization which makes some built-in data structures (e.g., `set`) have randomized structure. This is a security measure to prevent DoS attacks but it can influence reproducibility because the order in which a data structure is iterated over might change between executions of a pipeline.

  In the implementation of the reference runtime and other framework's components we have ensured that the internal order of data structures never influences the behavior of the program. But primitives can still depend on the internal order of data structures. Python allows disabling hash randomization by running with `PYTHONHASHSEED` environment variable set.

- Moreover, computations on GPUs are inherently non-deterministic because of parallel execution and limited precision of floating-point numbers.

  It is possible to configure execution to use deterministic order of operations, but this generally introduces performance penalty. While resulting numbers do change, we have not observed drastic influences because of this: models still converge during training.

- There are multiple global sources of randomness primitives could use: `random`, `numpy.random`, `tensorflow.random`, etc. Seeding those global random generators once does not really help in the context of AutoML systems where many pipelines are generally executed in parallel and the results of one pipeline should not depend on which other pipelines are executed at the same time.

  We try to mitigate this issue by discouraging the use of global random generators and instead providing a safe alternative to primitive authors, together with guidelines. Every primitive is provided with a random seed value as a constructor argument which a primitive should use as a seed for all the randomness in the primitive. We suggest to primitive authors to use this random seed to create a local instance of a random generator which they can then use instead of a global random generator. The reference runtime provides those random seeds to primitives in a deterministic manner.

- Primitives should keep any state inside its defined parameters. But in practice, especially when interacting with 3rd party libraries, using files is sometimes needed. This can lead to reproducibility issues if care is not taken. At worst, different executions of pipelines could influence each other. For example, if a primitive is storing files at a fixed location, that location would become effectively a shared state between executions of pipelines which include the same primitive. In practice, many primitives would use files as a cache. They would download additional files the first time they are initialized. While caching, if implemented properly, generally does not influence differences to predictions produced by a pipeline when executed multiple times, it can influence timing information of a pipeline execution. The same primitive given same input data, hyper-parameters, random seed, on the same machine, could run longer the first time because it is downloading additional files. This can introduce noise to metamodels predicting how long a primitive will run.

  To address this problem:

  - We support defining static files as a primitive's dependency in **installation** metadata. Runtime then provides paths to those static files through `volumes` argument to the primitive's constructor. Moreover, those dependencies are integrity protected to assure that static files do not change between pipeline executions. Additional advantages of this approach are that there is no penalty when a primitive is run for the first time and that additional files do not have to be downloaded repeatedly, every time cached files gets deleted.
  - Runtime provides `temporary_directory` constructor argument where a primitive can store any files for the duration of the current pipeline run phase. Runtime assures that the directory is unique for the primitive and that the directory is automatically cleaned up after the current pipeline run phase finishes.

- Some primitives connect to the Internet or some other resources not controlled by the runtime. Such primitives are inherently non-deterministic and reproducibility cannot

be assured.

For these primitives, we provide **pure_primitive** metadata flag which they should set to `false` to annotate themselves as such.

- When a pipeline is loaded and references are de-referenced, it might happen that a different version of a primitive, sub-pipeline, dataset, or problem description is available on the system where a pipeline is being executed. Those differences can be detected through mismatched digests. When this happens runtime can abort, or proceed with pipeline execution. Mismatched digests can suggest an explanation if results of this pipeline execution do not match expected results from the pipeline run which is being reproduced.

Reproducibility is not an absolute notion and various levels of reproducibility have different costs, so costs should be compared with benefits we obtain and needs we have. Do we want all pipelines to be reproducible forever? Or are we satisfied with reproduction of a subset of pipelines inside one session on one machine? When a level of reproducibility is not possible or its cost is prohibitive, the framework at least tries to provide information where is the source of non-determinism. In addition to **pure_primitive** metadata flag, pipeline and pipeline run descriptions try to capture all relevant information for this to be deductible: versions and digests of primitives, sub-pipelines, datasets, and problem descriptions involved in pipeline execution, information about the environment in which the pipeline was executed, like the hardware properties of the worker machine, metadata of all step outputs produced during pipeline execution, etc.

4.4 Representation of neural networks

Primitives can internally use neural networks. But such implementation hides neural network's structure itself from an AutoML system, not allowing the AutoML system to influence the structure to adapt it to data or problem.

A primitive could expose the structure in some way through a custom hyper-parameter, e.g., as a list of number of neurons in hidden layers of the network. This has multiple shortcomings:

- It still allows only the structure expressible through the hyper-parameter. For example, a list of number of neurons cannot represent skip connections.
- A more complicated hyper-parameter might require a specialized language to describe the structure. Such hyper-parameter might be tricky for an AutoML system to understand and/or use.
- Even if a hyper-parameter could describe the structure, the contents of the structure would still be limited. For example, only a fixed set of layer types could be used.

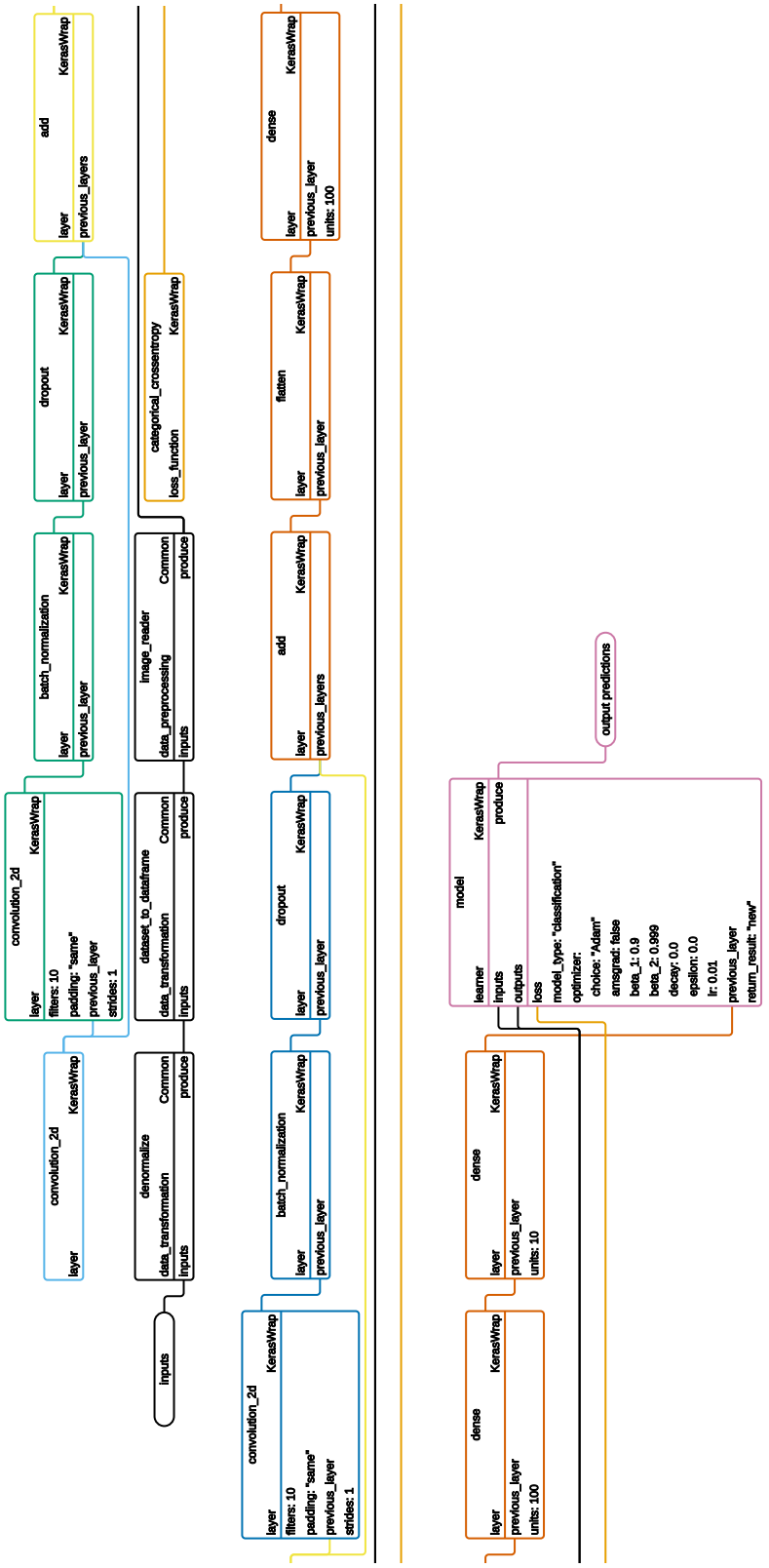


Figure 4.5: Visual representation of an example pipeline of a neural network. It is available in YAML serialization format in Appendix L.

`NeuralNetworkModuleMixin` primitive mixin allows defining primitives representing neural network modules. These modules can be either single layers, or they can be blocks of layers. The neural network can then be mapped to the pipeline structure, with neural network module primitives being passed to followup modules through hyper-parameters. The whole such neural network structure is then passed for the final time as a hyper-parameter to a training primitive which then builds the internal representation of the neural network and trains it.

Visual representation of an example pipeline of a neural network is available in Figure 4.5, and is available in YAML serialization format in Appendix L. Starting with the **convolution_2d** primitive we can see a branch of the pipeline with primitives representing a neural structure, together with skip connections. Some primitives have additional hyper-parameters which can configure various aspects of that module, e.g., the number of units in a dense layer. It is important to note that connections for this branch of the pipeline do not represent data connections, but connections of primitives themselves. As mentioned in Section 3.10, pipelines support connections where a primitive itself is passed to another primitive as a hyper-parameter value and this is used here to represent the neural network structure. The last primitive in the neural network is then passed to the **model** primitive which then through it accesses the whole neural network structure and builds an internal representation of the neural network it knows how to use. Another branch of the pipeline contains regular data preprocessing primitives to prepare inputs for fitting.

This approach allows an AutoML system to directly control a neural network structure, describing it as part of the pipeline itself. It allows defining skip connections and other special cases in neural networks. Moreover, it uses hyper-parameters to configure the properties of modules, allowing the AutoML system to tune them together with other hyper-parameters. Because the ecosystem of primitives is open, this approach makes also neural network modules be part of this ecosystem, enabling changes to the set of available modules through time.

A limitation of this approach is that all primitives in the neural network has to correspond to the same underlying library for neural networks. The library is then used in **model** primitive to build the internal and optimized representation of the neural network. If a more general approach is needed, which allows mixing modules from different libraries, the `GradientCompositionalityMixin` mixin can be used, at the expense of lower performance because of the overhead of communicating through Python API while backpropagating between primitives.

4.5 Overhead

The framework is a layer of abstraction on top of existing ML libraries. It enables powerful use of those diverse libraries for the purpose of AutoML, but as such it adds an overhead over using those libraries directly. Moreover, the reference runtime is an interpreter

for pipelines and does not compile or optimize their execution in any way. In this section we investigate this overhead.

	Python program	Python pipeline	Pipeline	Linear pipeline
Run time (s)	0.069 ± 0.009	0.053 ± 0.005	0.300 ± 0.056	0.442 ± 0.038
Memory usage (KB)	5688 ± 84	5729 ± 131	7753 ± 245	10027 ± 655

Table 4.1: Run time and memory usage of example programs and pipelines.

In Table 4.1 we show results of running program from Figure 3.1, program from Figure 3.2, pipeline from Appendix J, and pipeline from Appendix K. All programs and pipelines use the same Thyroid disease dataset [48]. We ran each 10 times and averaged measurements. We measured overall run time and memory usage during execution. We observe that framework's pipeline execution is an order of magnitude slower than execution of a Python program. Memory usage of framework's pipeline execution is twice that of a Python program.

Furthermore, we converted existing 231 ML programs in Python, done by ML experts, into our framework's pipelines. Those programs and pipelines use a wide range of datasets, primarily from OpenML [49], with different properties and sizes. Moreover, programs and pipelines themselves use a diverse set of ML libraries. As such we believe those programs and pipelines represent a good sample of potential ML programs one would use our framework for. We ran both original ML programs and pipelines 5 times and measured the average time it took for them to be fitted on training data, to produce predictions on testing data, and for predictions to be scored. The reason why we included scoring was because ML programs include it, so we measured pipelines with scoring as well. Additional obstacle in obtaining comparable measurements is that for pipelines we can get precise timing information from pipeline run descriptions, while for ML programs we can readily obtain only timing information of the whole Python process. To account for this, we measured average Python interpreter's startup time, including the time to import common ML libraries, and subtracted this time (1.232 s) from measured execution times of ML programs.

Figure 4.6 shows averaged execution times for all 231 pairs. Each pair is represented as two neighboring vertical lines, red for programs in Python and blue for pipelines. Average of (average) execution times of all ML programs is 2.0 s, average of (average) execution times of all pipelines is 33.9 s. We observe again that execution of pipelines is an order of magnitude slower than execution of Python programs.

We explore some ideas for pipeline execution optimization in Section 5.6.

4.6 Use in AutoML systems

Our framework has been used by 10 research groups in their AutoML systems as part of the Darpa D3M program [44]. They use different approaches to AutoML but use the same set of primitives and output pipelines. As part of the program's Winter 2019 Dry

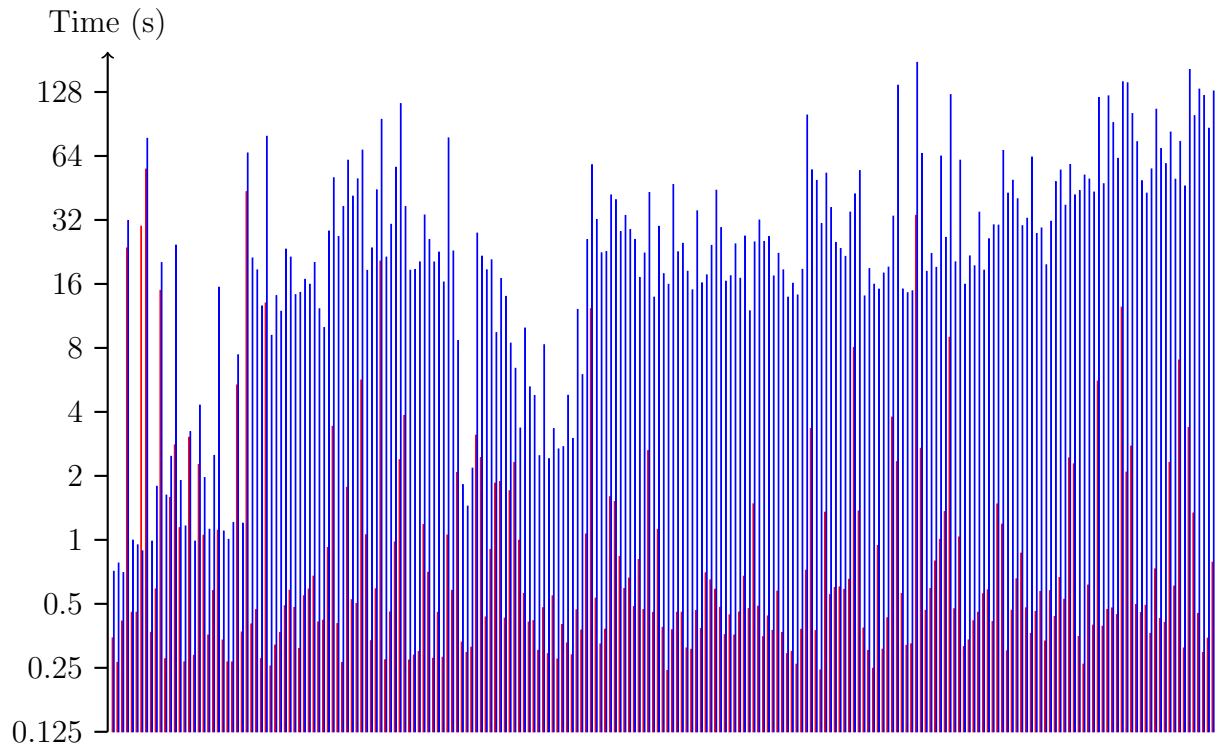


Figure 4.6: Averaged execution times of 231 ML programs in Python (red) and corresponding pipelines (blue).

Run they were run against the same set of 48 datasets. Datasets are diverse: tabular data, time-series data, graph data, images, audio, etc. Datasets have tasks associated with them: classification, regression, forecasting, graph matching, link prediction, etc. Each system was run for each dataset for one hour, producing zero or more pipelines per dataset. Those pipelines were scored and the best of them (per each system and dataset combination) was compared against a baseline: scores of expert-made ML programs for those same datasets. Figure 4.7 shows the results. We can see that systems have managed to beat the baseline for some datasets and for many datasets they have managed to produce working pipelines. Still for many datasets they have not been able to produce any working pipeline, but for almost all datasets at least one system managed to produce a working pipeline.

These results demonstrate that the framework can be used to describe both winning pipelines and a diverse set of pipelines. Using the framework allows comparison of different AutoML systems which can all use the same datasets, tasks, and primitives while focusing on their own approaches to AutoML. Scoring is done in a consistent way across systems and scores are reproducible. Moreover, all produced pipelines of all AutoML systems and pipeline run descriptions recording all the details of running them and scoring them are stored in a shared metalearning database, in a standard and machine consumable representation, enabling metalearning from results of all AutoML systems.

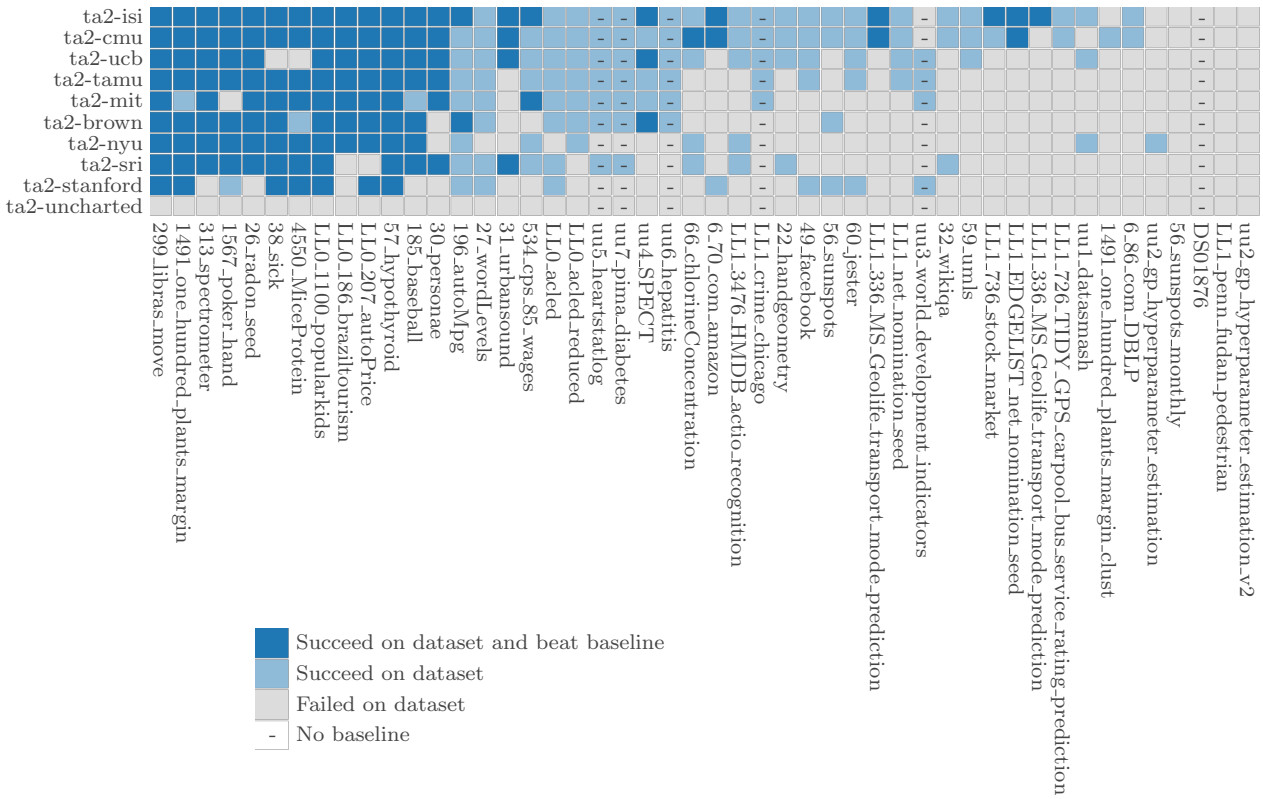


Figure 4.7: Results of running 10 AutoML systems on 48 datasets. Compared against expert-made ML programs as a baseline.

Chapter 5

Future work and conclusions

In this section we explore some of the broader issues related to AutoML which have we discovered during our work on the framework. We discuss potential solutions, but leave them to future work. At the end we provide conclusions of this work.

5.1 Evaluating pipelines on raw data

One of the promises of AutoML systems is that you should be able to give them any data and they do automatically the whole ML workflow. The issue is what to do when this data is raw data: provided as raw files without any additional (meta)information and not already structured in common ways for ML. Our framework allows loading such data and running it through a pipeline, but an AutoML system has to first know what the data even is, how it is organized, and how to preprocess it, to determine how to construct the rest of the pipeline. Assuming that an AutoML system is capable of constructing a pipeline for any data, including raw data, it is still a challenge how can an AutoML system evaluate such a pipeline during its search to determine the quality of the pipeline.

One way to evaluate the quality of the pipeline is for an AutoML system to split data into training data and testing data, fit the pipeline on training data and then score the pipeline using testing data. There are other ways to do it, but generally they all split data in some way. When working with raw data it is unclear how to apply such an approach: you have to preprocess the raw data to be able to split it, but you also want for preprocessing steps to be evaluated as part of the pipeline as well. Otherwise they could introduce bias or artifacts which could make the evaluation of the pipeline be invalid.

For example, imagine a preprocessing step which sets all values in a target column to the same constant value. Building a model for such target column is trivial, splitting into training and testing data after preprocessing and evaluating the pipeline (which does not include this step) would show a perfect score, but when predictions would be compared with real data, the results would be bad. If preprocessing steps are inside evaluation as well and are part of the pipeline itself, such problematic preprocessing step would be detected.

It is yet unclear how to address this chicken and egg problem.

5.2 Simplistic problem description

The problem description currently used in our framework and described in Section 3.12 can describe only a narrow set of ML problems: those which have one or more target columns. It follows that this requires the dataset to have at least one tabular resource with known metadata about columns. For some types of data this means that an artificial tabular resource has to be created to define such a target column, which means modifying original dataset. Moreover, a mapping of task types to such structure with a target column has to be defined and supported by AutoML systems using our framework. This might be cumbersome for some data types but it does not work at all for raw data, where such a target column does not readily exist.

A better and more general problem descriptions should be defined. Maybe instead of defining the problem in terms of standard ML tasks and target columns, we could describe the problem by describing how data available during training looks like, how data available during testing looks like, and how do predictions look like. Maybe the most general way to do that is to provide sample training, testing, and predictions data, and leave to the AutoML system to determine the rest. In a way this is already what we do at the primitive level, so we could just do it at the "meta" AutoML level as well.

5.3 Data metafeatures

When building meta-models for AutoML systems it is common to represent in the metalearning dataset input data as metafeatures [1, 14, 34, 35, 37], as described in Chapter 1. For this reason we have as part of standard metadata an extensive list of standardized metafeature fields to describe various properties of data potentially useful for metalearning. But they are all limited to tabular data with numeric, categorical, or textual attributes. In the future, metafeatures to support image, audio, video, graph, time-series and other types of data should be defined and added.

5.4 Pipeline metafeatures

Our framework provides a standard way to describe ML programs as pipelines. In Section 4.2 we presented how many pipelines can be made into linear pipelines. The motivation behind linear pipelines is that we believe they are simpler to integrate into meta-models, but we can see pipeline linearization as a special case of a broader question of how to represent any pipeline as a set of metafeatures to be used in meta-models. Especially when a pipeline contains many branches and sub-pipelines. More work is needed to answer this.

5.5 Pipeline validation

When searching a space of pipelines it is useful to know which ones are valid (would execute without failure) and which one are not. Knowing this we can prune the search space without having to execute a pipeline. In programming languages such validation is often done through type checking of the program. In early versions of the framework we defined an extensible validation interface for pipelines, modeled after type checking. Every primitive could define or extend a `can_accept` method which would be given input metadata object (instead of input data object) and if primitive could operate on data with such metadata, the method should return a new metadata object corresponding to primitive's output given input metadata. The idea was that one could simulate execution of a pipeline but instead of operating on data they would operate on (cheaper) metadata. In this idea the metadata could be seen as a generalization of typing information.

In practice this has not worked out. Implementing `can_accept` method correctly turned out to be hard. The behavior had to exactly match the behavior of the primitive when operating on data. Often implementing `can_accept` method meant implementing the primitive twice. Once for data and once for metadata. Having just input metadata available has shown to not be enough to construct a reasonable output metadata object, which lead to degradation of metadata quality with every step. If any primitive in a pipeline has not implemented the method well, the whole validation process failed.

Instead, future work could explore an approach where input data is sampled into quick to execute data sample and pipeline validation can be done by pipeline execution using this sample instead of full input data. In this way the same code path in primitives is used both for validation and full execution, addressing all the issues with `can_accept` method we observed.

5.6 Pipeline execution optimization

In Section 4.5 we showed how current implementation of the reference runtime is an order of magnitude slower and consumes twice as much memory as running an ML program directly. The reference runtime is currently an interpreter for pipelines and does not compile or optimize their execution in any way. It does not run segments of a pipeline in parallel, even if they are independent from each other. There are other known inefficiencies. Every primitive currently has to copy input values before modifying them and returning them, to simulate immutability of input values. This is costly and often unnecessary (when the same input value is not passed to any other primitive), but primitives have to do it just in case. Primitives have to maintain metadata and keep it up-to-date with data modifications. For large data and many modifications, metadata can also be large and require many changes. This is something ML programs do not do.

Pipeline execution could be optimized by running independent segments of pipelines in parallel (determining independent segments is trivial in our framework) and improve handling

of input values immutability. Metadata implementation itself could be further optimized for memory use and performance.

5.7 Conclusions

In this work, we addressed the challenge of representing ML programs for metalearning purposes with the end goal of improving AutoML systems that use metalearning. Moreover, this standard representation facilitates better comparison of different AutoML systems by allowing us to compare the approaches used by those systems, not just comparing their predictions. To this end we have designed and implemented this framework for ML pipelines.

Throughout our work, we explained how our framework satisfies the design goals laid out in Section 3.1. In particular:

- The framework allows existing ML programs to be described as pipelines while exposing just critical parts of the program's logic. We show this by converting existing 231 ML programs in Python into our framework's pipelines, as described in Section 4.5.
- We presented our dataset container type to support operating on raw files. This container type wraps raw files to serve as the input value to standard pipelines.
- We show how 10 AutoML systems are using our framework to generate comparable pipelines in Section 4.6.
- Throughout the description of the framework in Chapter 3, we demonstrated that the framework's components are decoupled and how framework itself is extensible.
- In Section 4.3 we explain how our framework addresses the issue of reproducibility.

In Chapter 4, we discussed various implications and extensions of the framework as designed and implemented. We presented standard and linear pipelines. In particular, we showed how to represent neural network programs as pipelines. We measured the overhead of using the framework and presented the results of comparing 10 AutoML systems, which use the framework, to each other. The results showed that the framework can be used both to describe a diverse set of ML programs and to determine unambiguously which AutoML system produced the best ML programs.

At the end, we noted the necessary future work on the framework itself, offering suggestions for improving current pipeline execution overhead, and potential research directions. While the framework supports operating on raw data, it is yet unclear how to evaluate AutoML systems and even just ML programs when input is raw data. Moreover, problem description currently depends on some structure to the data and cannot be used with raw data either. Similarly, existing work on metafeatures is limited to well structured tabular data and will need to be expanded to other problem and data types.

All the components of the framework, including its pipeline representation, are building blocks for existing and future AutoML research. They bring closer automatic generation and consumption of reproducible ML programs. This enables not just better comparison of existing AutoML systems but better future AutoML systems as well.

Appendix A

Terminology

In this work we use the following terminology:

framework The framework for ML pipelines that AutoML systems can directly consume and produce. See Chapter 3.

pipeline An ML program described using the framework.

pipeline run description A standard way of recording detailed information about the execution of a pipeline. See Section 3.15.

pipeline description Pipeline's standard representation. See Section 3.3.

primitive A basic building block of a pipeline. See Section 3.4.

pipeline template A partially defined pipeline. A pipeline with at least one placeholder step.

learnable function A primitive represents a learnable function. A function which does not have its logic necessarily defined in advance, but can learn it given example inputs and outputs. See Section 3.4.

primitive interface Implementation of required methods of a primitive. See Section 3.5.

primitive argument All arguments to all methods of a primitive are primitives arguments.

primitive input Primitive arguments together with all hyper-parameters are seen as inputs to the primitive as a whole, primitive inputs.

data reference A data reference is a string representing available data sources in a pipeline. See Section 3.8.

hyper-parameters configuration A mapping between hyper-parameter names and hyper-parameter definitions. See Section 3.6.

hyper-parameter definition An instance of a Python class and among other properties defines hyper-parameter space.

hyper-parameter space Valid values a hyper-parameter can take.

metadata A series of (selector, metadata entry) pairs representing all metadata associated with a value. See Section 3.9.

selector A selector matches in a general way a subset of data, a scope, for which a metadata entry applies.

segment A selector consists of a series of segments, in the order of dimensions in the data structure, across contained (nested) data types.

metadata entry A mapping between metadata fields and metadata values.

metadata field A name of the field of a metadata entry. Many field names are standardized.

metadata value An immutable value associated with the metadata field. Standardized field names have their values standardized as well.

structural type How the value is represented in memory. A Python type.

semantic type What is the meaning or use of the value. Represented as a URI.

fit-produce A standard execution semantics of pipelines. See Section 3.10.

standard pipeline A standard pipeline has as the pipeline inputs one (or more, but often only one) `Dataset` objects and should return a `DataFrame` with predictions. See Section 4.1.

Appendix B

Pipeline description

Full JSON Schema of pipeline description is available at its canonical location at: `https://metadata.datadrivendiscovery.org/schemas/v0/pipeline.json`.

The following is the list of standardized top-level fields and their descriptions:

created A timestamp.

description A natural language description in an unspecified language.

digest A SHA256 hexadecimal digest of value's content. It is a digest of the canonical JSON-serialization of the structure, without the **digest** field itself.

id A static id. It should never change for a given value, even if the value itself is changing. If possible, it should be a UUID generated in any way, but if there is an existing id available, it can be reused.

inputs Inputs to a pipeline. The order of inputs matter. Inputs are references by steps using a data reference.

name A human readable name in an unspecified language or format.

other_names Any other names associated with the value.

outputs Outputs from a pipeline. The order of outputs matter. Each output references an output of a step and in this way makes that step output a pipeline output as well.

schema A URI representing a metadata.datadrivendiscovery.org schema and version to which metadata conforms.

source Information about the source. Author and other information on how the value came to be.

steps Steps defining pipeline's logic.

users A list of users associated with the value.

Appendix C

Problem description

Full JSON Schema of problem description is available at its canonical location at: `https://metadata.datadrivendiscovery.org/schemas/v0/problem.json`.

The following is the list of standardized top-level fields and their descriptions:

data_augmentation Information about internal or external sources of data that can be used to address the challenge of data augmentation.

description A natural language description in an unspecified language.

digest A SHA256 hexadecimal digest of value's content. It is a digest of the canonical JSON-serialization of the structure, without the **digest** field itself.

id A static id. It should never change for a given value, even if the value itself is changing. If possible, it should be a UUID generated in any way, but if there is an existing id available, it can be reused.

inputs A list describing input datasets for the problem and associated targets. This list should match the list of inputs to a solution pipeline, in order.

location_uris A list of URIs where the value is stored.

name A human readable name in an unspecified language or format.

other_names Any other names associated with the value.

problem Metadata describing performance metrics to optimize for, and task type and subtype categories of the problem.

schema A URI representing a metadata.datadrivendiscovery.org schema and version to which metadata conforms.

version A string representing a version. Versions can be PEP 440 version strings or a SHA256 hexadecimal digest of value's content, if applicable. In the former case they are compared according to PEP 440 rules.

Appendix D

Primitive metadata

Full JSON Schema for primitive metadata is available at its canonical location at: `https://metadata.datadrivendiscovery.org/schemas/v0/primitive.json`.

The following is the list of standardized metadata top-level fields and their descriptions:

algorithm_types Algorithm type describes the underlying implementation of the primitive. It uses controlled, standardized, but open vocabulary which can be extended by request.

description A natural language description in an unspecified language.

digest A SHA256 hexadecimal digest of value's content. It is a digest of **id** and **installation** metadata.

effects A set of postconditions obtained by the data processed by this primitive. For example, a primitive may remove missing values.

hyperparams_to_tune A list containing the significant hyper-parameter names of a primitive that should be tuned (for prioritizing hyper-parameter tuning). For instance, if a primitive has 10 hyper-parameters, this metadata may be used to specify the two or three that affect the results the most.

id A static id. It should never change for a given value, even if the value itself is changing. For example, all versions of the same primitive should have the same id. If possible, it should be a UUID generated in any way, but if there is an existing id available, it can be reused.

installation Installation instructions for a primitive. Everything listed has to be installed, in order listed, for a primitive to work.

keywords A list of keywords. Strings in an unspecified language and vocabulary.

location_uris A list of URIs where the value is stored.

model_features A set of features supported by an underlying model of a primitive.

name A human readable name in an unspecified language or format.

original_python_path A fully-qualified Python path to primitive's class inside installable package and not one under the `d3m.primitives` namespace.

other_names Any other names associated with the value.

preconditions A set of requirements for the data given as an input to this primitive. For example, a primitive may not be able to handle data with missing values.

primitive_code Metadata describing the primitive's code.

primitive_family Primitive family describes the high-level purpose/nature of the primitive. Only one value per primitive is possible.

pure_primitive Does a primitive behave as a pure function. Are produced values always the same for same hyper-parameter values, arguments, random seed, and method calls made, including the order of method calls? Are there no side effects (mutations of state outside of primitive's internal state) when running the primitive? If primitive is connecting to the Internet or some other resources not controlled by the runtime, then primitive is not pure. If primitive caches files during execution, then primitive is pure despite modifying more than primitive's internal state, given that caching is implemented so that it does not leak information between different runs of a primitive.

python_path A fully-qualified Python path to primitive's class under the `d3m.primitives` namespace.

schema A URI representing a metadata.datadrivendiscovery.org schema and version to which metadata conforms.

source Information about the source. Author and other information on how the value came to be.

structural_type A Python type.

supported_media_types Which media types a primitive knows how to manipulate.

version A string representing a version. Versions can be PEP 440 version strings or a SHA256 hexadecimal digest of value's content, if applicable. In the former case they are compared according to PEP 440 rules.

Appendix E

Container metadata

Full JSON Schema for container metadata is available at its canonical location at: `https://metadata.datadrivendiscovery.org/schemas/v0/container.json`.

The following is the list of standardized metadata top-level fields and their descriptions:

approximate_stored_size Approximate size in bytes when or if stored to disk.

data_metafeatures Computed metafeatures over data. Some metafeatures can apply both at the container (dataset) or internal data levels (resource, table, column).

description A natural language description in an unspecified language.

digest A SHA256 hexadecimal digest of value's content. For datasets is digest over all files.

dimension Metadata for the dimension (e.g., rows and columns).

id A static id. It should never change for a given value, even if the value itself is changing. If possible, it should be a UUID generated in any way, but if there is an existing id available, it can be reused.

keywords A list of keywords. Strings in an unspecified language and vocabulary.

location_uris A list of URIs where the value is stored.

name A human readable name in an unspecified language or format.

other_names Any other names associated with the value.

schema A URI representing a metadata.datadrivendiscovery.org schema and version to which metadata conforms.

semantic_types A list of canonical URIs defining semantic types.

source Information about the source. Author and other information on how the value came to be.

stored_size Size in bytes when or if stored to disk.

structural_type A Python type.

version A string representing a version. Versions can be PEP 440 version strings or a SHA256 hexadecimal digest of value's content, if applicable. In the former case they are compared according to PEP 440 rules.

Appendix F

Data metadata

Full JSON Schema for data metadata is available at its canonical location at: `https://metadata.datadrivendiscovery.org/schemas/v0/data.json`.

The following is the list of standardized metadata top-level fields and their descriptions:

boundary_for A column in a table can be a boundary for another column in the same table or a table in another dataset resource.

data_metafeatures Computed metafeatures over data. Some metafeatures can apply both at the container (dataset) or internal data levels (resource, table, column).

description A natural language description in an unspecified language.

dimension Metadata for the dimension (e.g., rows and columns).

file_columns When the value is referencing a file with columns (e.g., a CSV file), columns metadata might be known in advance.

foreign_key Columns in a table in a dataset resource can reference other resources.

keywords A list of keywords. Strings in an unspecified language and vocabulary.

location_base_uris A list of URIs which can be used as a base to determine where the value is stored.

media_types Media type of the value in its extended form defining encoding, e.g., `text/plain; charset=utf-8`.

name A human readable name in an unspecified language or format.

other_names Any other names associated with the value.

sampling_rate Sampling rate (frequency) is the number of samples per second.

semantic_types A list of canonical URIs defining semantic types.

source Information about the source. Author and other information on how the value came to be.

stored_size Size in bytes when or if stored to disk.

structural_type A Python type.

Appendix G

Semantic types

Commonly used canonical URIs for semantic types are listed as possible values here, but any URI representing a semantic type can be used:

`http://schema.org/address` Value is an address, broadly defined.

`http://schema.org/addressCountry` Value is a country code.

`http://schema.org/AudioObject` Value is an audio clip.

`http://schema.org/Boolean` Value represents a boolean.

`http://schema.org/City` Value is a city, could be US or foreign.

`http://schema.org/Country` Value is a country.

`http://schema.org/DateTime` Value represents a timestamp.

`http://schema.org/email` Value is an email address.

`http://schema.org/Float` Value represents a float.

`http://schema.org/ImageObject` Value is an image.

`http://schema.org/Integer` Value represents an integer.

`http://schema.org/latitude` Value represents a latitude.

`http://schema.org/longitude` Value represents a longitude.

`http://schema.org/postalCode` Value is a US postal code.

`http://schema.org/State` Value is a state, could be US or foreign.

`http://schema.org/Text` Value is text/string.

`http://schema.org/URL` Value represents a URL.

`http://schema.org/VideoObject` Value is a video.

`https://metadata.datadrivendiscovery.org/types/AmericanPhoneNumber`
: Value can be recognized as an American style phone number, e.g., `(310)822-1511` and `1-310-822-1511`.

`https://metadata.datadrivendiscovery.org/types/Attribute`
: Value serves as an attribute (input feature) to fit on or be used for analysis.

`https://metadata.datadrivendiscovery.org/types/Boundary`
: Value represents a boundary.

`https://metadata.datadrivendiscovery.org/types/BoundingPolygon`
: Value represents a bounding polygon as a series of (X, Y) coordinate pairs of vertices in counter-clockwise order.

`https://metadata.datadrivendiscovery.org/types/CategoricalData`
: Value represents categorical data.

`https://metadata.datadrivendiscovery.org/types/ChoiceParameter`
: Hyper-parameter is selecting one choice among multiple hyper-parameters space choices.

`https://metadata.datadrivendiscovery.org/types/ColumnRole`
: A column can have a role in a table.

`https://metadata.datadrivendiscovery.org/types/Confidence`
: Value serves as a confidence of a predicted target variable. `confidence_for` metadata can be used to reference for which target column(s) this column is confidence for.

`https://metadata.datadrivendiscovery.org/types/ConstructedAttribute`
: Value serves as a constructed attribute (input feature). This is set by primitives when constructing attributes. It should not be used for fitting.

`https://metadata.datadrivendiscovery.org/types/ControlParameter`
: Hyper-parameter is a control parameter of the primitive.

`https://metadata.datadrivendiscovery.org/types/CPUResourcesUseParameter`
: Hyper-parameter is a parameter which controls the use of CPU resources (cores) by the primitive.

`https://metadata.datadrivendiscovery.org/types/DatasetEntryPoint`
: Resource is a dataset entry point.

`https://metadata.datadrivendiscovery.org/types/DatasetResource`
: Value is a dataset resource.

`https://metadata.datadrivendiscovery.org/types/DimensionType`
: Value represents a dimension.

`https://metadata.datadrivendiscovery.org/types/DirectedEdgeSource`
: Value serves as a source of a directed graph edge.

`https://metadata.datadrivendiscovery.org/types/DirectedEdgeTarget`
: Value serves as a target of a directed graph edge.

`https://metadata.datadrivendiscovery.org/types/EdgeList`
: Value is an edge list of a graph structure.

`https://metadata.datadrivendiscovery.org/types/EdgeSource`
: Value serves as a source of a graph edge.

`https://metadata.datadrivendiscovery.org/types/EdgeTarget`
: Value serves as a target of a graph edge.

`https://metadata.datadrivendiscovery.org/types/FileName`
: Value is a filename.

`https://metadata.datadrivendiscovery.org/types/FilesCollection`
: Resource is a files collection.

`https://metadata.datadrivendiscovery.org/types/FloatVector`
: Value represents a vector of floats.

`https://metadata.datadrivendiscovery.org/types/GeoJSON`
: Value represents a GeoJSON object.

`https://metadata.datadrivendiscovery.org/types/GPUResourcesUseParameter`
: Hyper-parameter is a parameter which controls the use of GPU resources by the primitive.

`https://metadata.datadrivendiscovery.org/types/Graph`
: Value is a graph structure or a node list of a graph structure.

`https://metadata.datadrivendiscovery.org/types/GroupingKey`
: Value serves as an active grouping key to group rows (samples) together. Used in time-series datasets containing multiple time-series to identify individual time-series. Each column with this semantic type should be used individually and if multiple columns with this semantic type exist, each column represents a different grouping.

`https://metadata.datadrivendiscovery.org/types/HyperParameter`
: Value is a hyper-parameter.

`https://metadata.datadrivendiscovery.org/types/InstanceWeight`
: Value serves as a weight for an instance.

`https://metadata.datadrivendiscovery.org/types/Interval`
: Value represents an interval as a pair of start and end.

`https://metadata.datadrivendiscovery.org/types/IntervalEnd`
: Value represents an end of an interval.

`https://metadata.datadrivendiscovery.org/types/IntervalStart`
: Value represents a start of an interval.

`https://metadata.datadrivendiscovery.org/types/InvalidData`
: Value is present, but is invalid.

`https://metadata.datadrivendiscovery.org/types/JSON`
: Value represents a JSON object.

`https://metadata.datadrivendiscovery.org/types/Location`
: Value represents a location.

`https://metadata.datadrivendiscovery.org/types/MetafeatureParameter`
: Hyper-parameter controls which meta-feature is computed by the primitive.

`https://metadata.datadrivendiscovery.org/types/MissingData`
: Value is missing.

`https://metadata.datadrivendiscovery.org/types/MultiEdgeSource`
: Value serves as a source of a multigraph edge.

`https://metadata.datadrivendiscovery.org/types/MultiEdgeTarget`
: Value serves as a target of a multigraph edge.

`https://metadata.datadrivendiscovery.org/types/OrdinalData`
: Value represents ordinal data.

`https://metadata.datadrivendiscovery.org/types/PredictedTarget`
: Value serves as a predict target variable for a problem. This is set by primitives when predicting targets.

`https://metadata.datadrivendiscovery.org/types/PrimaryKey`
: Value serves as a primary key.

`https://metadata.datadrivendiscovery.org/types/PrimaryMultiKey`
: Value serves as a primary key without uniqueness constraint to allow the same row to be repeated multiple times.

`https://metadata.datadrivendiscovery.org/types/PrivilegedData`
: Value serves as a privileged (available during fitting but not producing) attribute.

`https://metadata.datadrivendiscovery.org/types/RedactedPrivilegedData`
: Value is redacted, but would otherwise be a privileged attribute.

`https://metadata.datadrivendiscovery.org/types/RedactedTarget`
: Value is redacted, but would otherwise be a target variable for a problem. This is a property of input data.

`https://metadata.datadrivendiscovery.org/types/ResourcesUseParameter`
: Hyper-parameter is a parameter which controls the use of resources by the primitive.

`https://metadata.datadrivendiscovery.org/types/Score`
: Value is a prediction score computed by comparing predicted and true target.

`https://metadata.datadrivendiscovery.org/types/SimpleEdgeSource`
: Value serves as a source of a simple graph edge.

`https://metadata.datadrivendiscovery.org/types/SimpleEdgeTarget`
: Value serves as a target of a simple graph edge.

`https://metadata.datadrivendiscovery.org/types/Speech`
: Value is an audio clip of human speech.

`https://metadata.datadrivendiscovery.org/types/SuggestedGroupingKey`
: Value serves as a potential grouping key to group rows (samples) together. Used in time-series datasets containing multiple time-series to hint how to identify individual time-series. If there are multiple columns with this semantic type the relation between them is unspecified, they can be used individually or in combination.

`https://metadata.datadrivendiscovery.org/types/SuggestedPrivilegedData`
: Value serves as a potential privileged (available during fitting but not producing) attribute.

`https://metadata.datadrivendiscovery.org/types/SuggestedTarget`
: Value serves as a potential target variable for a problem. This is a property of input data.

`https://metadata.datadrivendiscovery.org/types/Table`
: Value is tabular data.

`https://metadata.datadrivendiscovery.org/types/TabularColumn`
: Value is a column in tabular data.

`https://metadata.datadrivendiscovery.org/types/TabularRow`
: Value is a row in tabular data.

`https://metadata.datadrivendiscovery.org/types/Target`
: Value serves as a target variable for a problem.

`https://metadata.datadrivendiscovery.org/types/Time`
: Value represents time.

`https://metadata.datadrivendiscovery.org/types/Timeseries`
: Value is time-series data.

`https://metadata.datadrivendiscovery.org/types/TrueTarget`
: Value serves as a true target variable for a problem. This is set by a runtime based on problem description.

`https://metadata.datadrivendiscovery.org/types/TuningParameter`
: Hyper-parameter is a tuning parameter of the primitive.

`https://metadata.datadrivendiscovery.org/types/UndirectedEdgeSource`
: Value serves as a source of an undirected graph edge.

`https://metadata.datadrivendiscovery.org/types/UndirectedEdgeTarget`
: Value serves as a target of an undirected graph edge.

`https://metadata.datadrivendiscovery.org/types/UniqueKey`
: Value serves as an unique key, i.e., it satisfies the uniqueness constraint among other values.

`https://metadata.datadrivendiscovery.org/types/UnknownType`
: It is not known what the value represents.

`https://metadata.datadrivendiscovery.org/types/UnspecifiedStructure`
: Value has unspecified structure.

Appendix H

Hyper-parameter base classes

We provide standard hyper-parameter definition base classes:

Hyperparameter(structural_type, default, semantic_types, description)
A base hyper-parameter class does not provide any information about the space of the hyper-parameter, besides a default value. It can be defined over any structural type.

Primitive(structural_type, default, primitive_families, algorithm_types, produce_methods, semantic_types, description)
A hyper-parameter describing a primitive or primitives. Matching primitives are determined based on their structural type (a matching primitive has to be an instance or a subclass of the structural type), their primitive's family (a matching primitive's family has to be among those listed in the hyper-parameter), their algorithm types (a matching primitive has to implement at least one of the listed in the hyper-parameter), and produce methods provided (a matching primitive has to provide all of the listed in the hyper-parameter).

Constant(structural_type, default, semantic_types, description)
A hyper-parameter that represents a constant default value.

Bounded(structural_type, default, lower, upper, lower_inclusive, upper_inclusive, semantic_types, description)
A hyper-parameter with lower and upper bounds, but no other information about the distribution of the space of the hyper-parameter, besides a default value.

Enumeration(structural_type, values, default, semantic_types, description)
A hyper-parameter with a value drawn uniformly from a list of values.

UniformBool(default, semantic_types, description)
A boolean hyper-parameter with a value drawn uniformly from **true** and **false** values.

UniformInt(default, lower, upper, lower_inclusive, upper_inclusive, semantic_types, description)

An integer hyper-parameter with a value drawn uniformly from the interval.

Uniform(default, lower, upper, lower_inclusive, upper_inclusive, q, semantic_types, description)

A floating number hyper-parameter with a value drawn uniformly from the interval. If **q** argument is provided, then the value is drawn according to $\text{round}(\text{uniform}(\text{lower}, \text{upper})/\text{q}) \cdot \text{q}$.

LogUniform(default, lower, upper, lower_inclusive, upper_inclusive, q, semantic_types, description)

A floating number hyper-parameter with a value drawn from the interval according to $\exp(\text{uniform}(\log(\text{lower}), \log(\text{upper})))$ If **q** argument is provided, then the value is drawn according to $\text{round}(\exp(\text{uniform}(\log(\text{lower}), \log(\text{upper})))/\text{q}) \cdot \text{q}$.

Normal(default, mu, sigma, q, semantic_types, description)

A floating number hyper-parameter with a value drawn normally distributed according to **mu** and **sigma** arguments. If **q** argument is provided, then the value is drawn according to $\text{round}(\text{normal}(\text{mu}, \text{sigma})/\text{q}) \cdot \text{q}$.

LogNormal(default, mu, sigma, q, semantic_types, description)

A floating number hyper-parameter with a value drawn according to $\exp(\text{normal}(\text{mu}, \text{sigma}))$ so that the logarithm of the value is normally distributed. If **q** argument is provided, then the value is drawn according to $\text{round}(\exp(\text{normal}(\text{mu}, \text{sigma}))/\text{q}) \cdot \text{q}$.

Union(structural_type, configuration, default, semantic_types, description)

A hyper-parameter which combines multiple other hyper-parameters and creates a union of their hyper-parameter spaces. This is useful when a hyper-parameter has multiple modalities and each modality can be described with a different hyper-parameter. No relation or probability distribution between modalities is prescribed.

Choice(choices, default, semantic_types, description)

A hyper-parameter which combines multiple hyper-parameter configurations into one hyper-parameter. This is useful when a combination of hyper-parameters should exist together. Then such combinations can be made each into one choice. No relation or probability distribution between choices is prescribed. This is similar to **Union** hyper-parameter that it combines hyper-parameters, but **Choice** combines configurations of multiple hyper-parameters, while **Union** combines individual hyper-parameters.

Set(structural_type, elements, min_size, max_size, default, semantic_types, description)

A hyper-parameter which samples without replacement multiple times another hyper-parameter or hyper-parameters configuration. This is useful when a primitive is interested in more than one value of a hyper-parameter or hyper-parameters configuration.

The order of elements does not matter (two different orders of same elements represent the same value), but order is meaningful and preserved to assure reproducibility.

SortedSet(structural_type, elements, min_size, max_size, ascending, default, semantic_types, description)
Similar to **Set** hyper-parameter, but elements of values are required to be sorted.

List(structural_type, elements, min_size, max_size, default, semantic_types, description)
A hyper-parameter which samples with replacement multiple times another hyper-parameter or hyper-parameters configuration. This is useful when a primitive is interested in more than one value of a hyper-parameter or hyper-parameters configuration. The order of elements matters and is preserved but is not prescribed.

SortedList(structural_type, elements, min_size, max_size, default, semantic_types, description)
Similar to **List** hyper-parameter, but elements of values are required to be sorted.

Appendix I

Pipeline run description

Full JSON Schema of pipeline run description is available at its canonical location at: `https://metadata.datadrivendiscovery.org/schemas/v0/pipeline_run.json`.

The following is the list of standardized top-level fields and their descriptions:

context Context in which a pipeline was run.

datasets A list of input datasets. The order matters because it is mapped to pipeline inputs.

end Absolute timestamp of the end of the run of the pipeline.

environment A description of the runtime environment, including engine versions, Docker images, compute resources, and benchmarks.

id An UUIDv5 id is computed by using UUID namespace `8614b2cc-89ef-498e-9254-833233b3959b` and JSON-serialized contents of the document without the **id** field for UUID name.

pipeline A pipeline associated with this pipeline run.

previous_pipeline_run References a pipeline run that occurred immediately before this pipeline run. Used for reproducibility, for example a test run would reference the train run. If it is not provided, it indicates the first pipeline run.

problem A problem description associated with this pipeline run.

random_seed The main random seed used to run the pipeline.

run How a pipeline was run and corresponding results and scores.

schema A URI representing a metadata.datadrivendiscovery.org schema and version to which metadata conforms.

start Absolute timestamp of the start of the run of the pipeline.

status Indicates whether a pipeline, or some portion of it, ran successfully. May include a message with more details about the status.

steps All of the steps invoked in the pipeline run. There is a one-to-one correspondence between this array and the steps in the pipeline.

users A list of users associated with the value.

Appendix J

Example pipeline

An example pipeline. It is logically equivalent to the ML program in Figure 3.1. It is in YAML serialization format which supports inline comments. A corresponding visual representation of the example pipeline is available in Figure 3.5. The example pipeline is described in more detail in Section 3.11.

```
id: 0a382f70-e4f3-4e03-b99b-e6e1bee86d66
schema: https://metadata.datadrivendiscovery.org/schemas/v0/pipeline.json
digest: 6ad90ccf5eafb899fa2da8f595163611cd06fce1cba5998d80e811709228bd2a
source:
  name: Mitar
created: '2019-06-18T00:13:00.992902Z'
name: Sick dataset pipeline
description: |-
  A simple pipeline which runs on Sick dataset.
inputs:
- name: input dataset
outputs:
- data: steps.10.produce
  name: predictions
steps:
# Step 0. Convert input Dataset to a DataFrame
# (there is only one tabular resource in Dataset).
- type: PRIMITIVE
  primitive:
    id: 4b42ce1e-9b98-4a25-b68e-fad13311eb65
    version: 0.3.0
    python_path: |-
      d3m.primitives.data_transformation.dataset_to_dataframe.Common
    name: Extract a DataFrame from a Dataset
    digest: 458a82145751686619d331a0d15cecb28d6c9f5eeb9450b9ebccf83a0ece49fd
  arguments:
    inputs:
      type: CONTAINER
      data: inputs.0
  outputs:
```

```
  - id: produce
# Step 1. Extract attributes.
- type: PRIMITIVE
  primitive:
    id: 4503a4c6-42f7-45a1-a1d4-ed69699cf5e1
    version: 0.3.0
    python_path: |-
      d3m.primitives.data_transformation.extract_columns_by_semantic_types.Common
    name: Extracts columns by semantic type
    digest: ead1ae24b4da1b6f547e99863347b061dcef80028ac3ff89e37d2bf9d1c5af2f
  arguments:
    inputs:
      type: CONTAINER
      data: steps.0.produce
  outputs:
  - id: produce
  hyperparams:
    semantic_types:
      type: VALUE
      data:
        - https://metadata.datadrivendiscovery.org/types/Attribute
# Step 2. Extract numerical attributes.
- type: PRIMITIVE
  primitive:
    id: 4503a4c6-42f7-45a1-a1d4-ed69699cf5e1
    version: 0.3.0
    python_path: |-
      d3m.primitives.data_transformation.extract_columns_by_semantic_types.Common
    name: Extracts columns by semantic type
    digest: ead1ae24b4da1b6f547e99863347b061dcef80028ac3ff89e37d2bf9d1c5af2f
  arguments:
    inputs:
      type: CONTAINER
      data: steps.1.produce
  outputs:
  - id: produce
  hyperparams:
    semantic_types:
      type: VALUE
      data:
      - https://metadata.datadrivendiscovery.org/types/CategoricalData
    negate:
      type: VALUE
      data: true
# Step 3. Extract categorical attributes.
- type: PRIMITIVE
  primitive:
    id: 4503a4c6-42f7-45a1-a1d4-ed69699cf5e1
    version: 0.3.0
    python_path: |-
```

```
      d3m.primitives.data_transformation.extract_columns_by_semantic_types.Common
    name: Extracts columns by semantic type
    digest: ead1ae24b4da1b6f547e99863347b061dcef80028ac3ff89e37d2bf9d1c5af2f
  arguments:
    inputs:
      type: CONTAINER
      data: steps.1.produce
  outputs:
  - id: produce
  hyperparams:
    semantic_types:
      type: VALUE
      data:
        - https://metadata.datadrivendiscovery.org/types/CategoricalData
# Step 4. Extract target.
- type: PRIMITIVE
  primitive:
    id: 4503a4c6-42f7-45a1-a1d4-ed69699cf5e1
    version: 0.3.0
    python_path: |-
      d3m.primitives.data_transformation.extract_columns_by_semantic_types.Common
    name: Extracts columns by semantic type
    digest: ead1ae24b4da1b6f547e99863347b061dcef80028ac3ff89e37d2bf9d1c5af2f
  arguments:
    inputs:
      type: CONTAINER
      data: steps.0.produce
  outputs:
  - id: produce
  hyperparams:
    semantic_types:
      type: VALUE
      data:
        - https://metadata.datadrivendiscovery.org/types/TrueTarget
# Step 5. Parse numerical attributes.
- type: PRIMITIVE
  primitive:
    id: d510cb7a-1782-4f51-b44c-58f0236e47c7
    version: 0.5.0
    python_path: |-
      d3m.primitives.data_transformation.column_parser.Common
    name: Parses strings into their types
    digest: 14723622d623237138acbbc8b1a8652afd4da72b370ccf11f872502139987f37
  arguments:
    inputs:
      type: CONTAINER
      data: steps.2.produce
  outputs:
  - id: produce
# Step 6. Impute numerical attributes.
```

```
- type: PRIMITIVE
  primitive:
    id: d016df89-de62-3c53-87ed-c06bb6a23cde
    version: 2019.6.7
    python_path: |-
      d3m.primitives.data_cleaning.imputer.SKlearn
    name: sklearn.impute.SimpleImputer
    digest: 4da1eb6ad85ae67702c565fc5f107eb3acf94acb7f109b031615131df6aa1328
  arguments:
    inputs:
      type: CONTAINER
      data: steps.5.produce
  outputs:
  - id: produce
# Step 7. Encode categorical attributes.
- type: PRIMITIVE
  primitive:
    id: a048aaa7-4475-3834-b739-de3105ec7217
    version: 2019.6.7
    python_path: |-
      d3m.primitives.data_transformation.ordinal_encoder.SKlearn
    name: sklearn.preprocessing._encoders.OrdinalEncoder
    digest: 0d96119f490e1b0997922c78de9bc1897c3f37a77efc5914580ad7ab27d0bd9e
  arguments:
    inputs:
      type: CONTAINER
      data: steps.3.produce
  outputs:
  - id: produce
# Step 8. Concatenate attributes.
- type: PRIMITIVE
  primitive:
    id: aff6a77a-faa0-41c5-9595-de2e7f7c4760
    version: 0.2.0
    python_path: |-
      d3m.primitives.data_transformation.horizontal_concat.DataFrameCommon
    name: Concatenate two dataframes
    digest: 9fe4316defd76f2699d85547223bd7bd069d30fe6720c668c0fd2b2eb7ef0223
  arguments:
    left:
      type: CONTAINER
      data: steps.7.produce
    right:
      type: CONTAINER
      data: steps.6.produce
  outputs:
  - id: produce
# Step 9. Random forest.
- type: PRIMITIVE
  primitive:
```

```
    id: 1dd82833-5692-39cb-84fb-2455683075f3
    version: 2019.6.7
    python_path: |-
      d3m.primitives.classification.random_forest.SKlearn
    name: sklearn.ensemble.forest.RandomForestClassifier
    digest: 827f301cbff659371471def6f86f200b43497298937ed868eb630984560709c3
  arguments:
    inputs:
      type: CONTAINER
      data: steps.8.produce
    outputs:
      type: CONTAINER
      data: steps.4.produce
  outputs:
  - id: produce
# Step 10. Restore row indices.
- type: PRIMITIVE
  primitive:
    id: 8d38b340-f83f-4877-baaa-162f8e551736
    version: 0.3.0
    python_path: |-
      d3m.primitives.data_transformation.construct_predictions.Common
    name: Construct pipeline predictions output
    digest: 9d1ecd3c3172cad6a49e1d89a9a7f1a531cb1e5bab40db5cca1b40160155e519
  arguments:
    inputs:
      type: CONTAINER
      data: steps.9.produce
    reference:
      type: CONTAINER
      data: steps.0.produce
  outputs:
  - id: produce
```

Appendix K

Example linear pipeline

An example linear pipeline. It is logically equivalent to the pipeline in Appendix J. A corresponding visual representation of the example linear pipeline is available in Figure 4.4. Linear pipelines are described in Section 4.2.

```
id: fe8a6192-51dc-42d4-8de2-3e6d530b73d7
schema: https://metadata.datadrivendiscovery.org/schemas/v0/pipeline.json
digest: a0daf1e95c9c1570ce16b9f369e49f3e56d3b1a271bf909bb29df6c6a0928442
source:
  name: Mitar
created: '2019-06-18T15:09:18.283219Z'
name: Sick dataset linear pipeline
description: |-
  A linear pipeline which runs on Sick dataset.
inputs:
- name: input dataset
outputs:
- data: steps.4.produce
  name: predictions
steps:
# Step 0. Convert input Dataset to a DataFrame
# (there is only one tabular resource in Dataset).
- type: PRIMITIVE
  primitive:
    id: 4b42ce1e-9b98-4a25-b68e-fad13311eb65
    version: 0.3.0
    python_path: |-
      d3m.primitives.data_transformation.dataset_to_dataframe.Common
    name: Extract a DataFrame from a Dataset
    digest: 458a82145751686619d331a0d15cecb28d6c9f5eeb9450b9ebccf83a0ece49fd
  arguments:
    inputs:
      type: CONTAINER
      data: inputs.0
  outputs:
  - id: produce
```

```
# Step 1. Encode categorical attributes.
- type: PRIMITIVE
  primitive:
    id: a048aaa7-4475-3834-b739-de3105ec7217
    version: 2019.6.7
    python_path: |-
      d3m.primitives.data_transformation.ordinal_encoder.SKlearn
    name: sklearn.preprocessing._encoders.OrdinalEncoder
    digest: 0d96119f490e1b0997922c78de9bc1897c3f37a77efc5914580ad7ab27d0bd9e
  arguments:
    inputs:
      type: CONTAINER
      data: steps.0.produce
  outputs:
  - id: produce
  hyperparams:
    use_semantic_types:
      type: VALUE
      data: true
    return_result:
      type: VALUE
      data: replace
# Step 2. Parse numerical columns.
- type: PRIMITIVE
  primitive:
    id: d510cb7a-1782-4f51-b44c-58f0236e47c7
    version: 0.5.0
    python_path: |-
      d3m.primitives.data_transformation.column_parser.Common
    name: Parses strings into their types
    digest: 14723622d623237138acbbc8b1a8652afd4da72b370ccf11f872502139987f37
  arguments:
    inputs:
      type: CONTAINER
      data: steps.1.produce
  outputs:
  - id: produce
# Step 3. Impute numerical attributes.
- type: PRIMITIVE
  primitive:
    id: d016df89-de62-3c53-87ed-c06bb6a23cde
    version: 2019.6.7
    python_path: |-
      d3m.primitives.data_cleaning.imputer.SKlearn
    name: sklearn.impute.SimpleImputer
    digest: 4da1eb6ad85ae67702c565fc5f107eb3acf94acb7f109b031615131df6aa1328
  arguments:
    inputs:
      type: CONTAINER
      data: steps.2.produce
```

```
  outputs:
  - id: produce
  hyperparams:
    use_semantic_types:
      type: VALUE
      data: true
    return_result:
      type: VALUE
      data: replace
# Step 4. Random forest.
- type: PRIMITIVE
  primitive:
    id: 1dd82833-5692-39cb-84fb-2455683075f3
    version: 2019.6.7
    python_path: |-
      d3m.primitives.classification.random_forest.SKlearn
    name: sklearn.ensemble.forest.RandomForestClassifier
    digest: 827f301cbff659371471def6f86f200b43497298937ed868eb630984560709c3
  arguments:
    inputs:
      type: CONTAINER
      data: steps.3.produce
    outputs:
      type: CONTAINER
      data: steps.3.produce
  outputs:
  - id: produce
  hyperparams:
    use_semantic_types:
      type: VALUE
      data: true
    add_index_columns:
      type: VALUE
      data: true
```

Appendix L

Example neural network pipeline

An example pipeline of a neural network. A corresponding visual representation of the example neural network pipeline is available in Figure 4.5. Representation of neural networks in a pipeline is described in Section 4.4.

```
id: c917f854-2630-4837-9de0-ccc2e6bda2af
schema: https://metadata.datadrivendiscovery.org/schemas/v0/pipeline.json
digest: 2e17eb20105838ed684f547516ed5b369c5e9fed746ab84e3db7f7161b6e2083
created: '2019-06-21T17:59:06.976929Z'
inputs:
- name: inputs
outputs:
- data: steps.17.produce
  name: output predictions
steps:
# Step 0. Multiple tabular resources are joined into one.
# This combines collection of files with the main tabular resource.
- type: PRIMITIVE
  primitive:
    id: f31f8c1f-d1c5-43e5-a4b2-2ae4a761ef2e
    version: 0.2.0
    python_path: |-
      d3m.primitives.data_transformation.denormalize.Common
    name: Denormalize datasets
    digest: 3555a5cfd37f4e9d08e7aaf48e4c9e87b7321bff3cf68d81838ee09e840fece8
  arguments:
    inputs:
      type: CONTAINER
      data: inputs.0
  outputs:
  - id: produce
# Step 1. Convert input Dataset to a DataFrame.
- type: PRIMITIVE
  primitive:
    id: 4b42ce1e-9b98-4a25-b68e-fad13311eb65
    version: 0.3.0
```

```
    python_path: |-
      d3m.primitives.data_transformation.dataset_to_dataframe.Common
    name: Extract a DataFrame from a Dataset
    digest: 458a82145751686619d331a0d15cecb28d6c9f5eeb9450b9ebccf83a0ece49fd
  arguments:
    inputs:
      type: CONTAINER
      data: steps.0.produce
  outputs:
  - id: produce
# Step 2. Read all images into cells in the column referencing files.
- type: PRIMITIVE
  primitive:
    id: 8f2e51e8-da59-456d-ae29-53912b2b9f3d
    version: 0.2.0
    python_path: |-
      d3m.primitives.data_preprocessing.image_reader.Common
    name: Columns image reader
    digest: 46b07b47d2b37c2f375c50654fd11aa877a9f97b112bb837844fdaa65466ef54
  arguments:
    inputs:
      type: CONTAINER
      data: steps.1.produce
  outputs:
  - id: produce
# Step 3. Loss function. Used as a hyper-parameter value.
- type: PRIMITIVE
  primitive:
    id: 01cee53a-88e3-4bf3-993a-fd64805e9b8e
    version: 0.1.0
    python_path: |-
      d3m.primitives.loss_function.categorical_crossentropy.KerasWrap
    name: categorical_crossentropy
    digest: 378c9e4ef29333dabe3dd1d1371d078b46c4180c26c56fcb1093f97520564f2f
# Step 4. Layer. Used as a hyper-parameter value.
- type: PRIMITIVE
  primitive:
    id: 6b1cceeb-a3be-4d7d-809a-052ae72b2b13
    version: 0.1.0
    python_path: |-
      d3m.primitives.layer.convolution_2d.KerasWrap
    name: convolution_2d
    digest: 2af0acc46d4c2925f550bc88f0927afe11c78008f28f767d740d8146cc2af52d
# Step 5. Layer. Used as a hyper-parameter value.
- type: PRIMITIVE
  primitive:
    id: 6b1cceeb-a3be-4d7d-809a-052ae72b2b13
    version: 0.1.0
    python_path: |-
      d3m.primitives.layer.convolution_2d.KerasWrap
```

```
    name: convolution_2d
    digest: 2af0acc46d4c2925f550bc88f0927afe11c78008f28f767d740d8146cc2af52d
  hyperparams:
    filters:
      type: VALUE
      data: 10
    padding:
      type: VALUE
      data: same
    # Here we connect this layer to another layer.
    previous_layer:
      type: PRIMITIVE
      data: 4
    strides:
      type: VALUE
      data: 1
# Step 6. Layer. Used as a hyper-parameter value.
- type: PRIMITIVE
  primitive:
    id: 288162cd-b71d-418d-9555-cc177c5f592e
    version: 0.1.0
    python_path: |-
      d3m.primitives.layer.batch_normalization.KerasWrap
    name: batch_normalization
    digest: 8c1d94282d1747a5fb639d3b3ec3934c252e706ff38396a61439ad56186a381d
  hyperparams:
    # Here we connect this layer to another layer.
    previous_layer:
      type: PRIMITIVE
      data: 5
# Step 7. Layer. Used as a hyper-parameter value.
- type: PRIMITIVE
  primitive:
    id: 7a76922c-bf6f-37ce-9e58-1d3315382506
    version: 0.1.0
    python_path: |-
      d3m.primitives.layer.dropout.KerasWrap
    name: dropout
    digest: 52b8acabf95b3781c60a26d613a6f876ec583c9fb0ea018bb4780f0390248633
  hyperparams:
    # Here we connect this layer to another layer.
    previous_layer:
      type: PRIMITIVE
      data: 6
# Step 8. Layer. Used as a hyper-parameter value.
- type: PRIMITIVE
  primitive:
    id: c97c5620-6274-4a5f-83b0-1c090c68ac7b
    version: 0.1.0
    python_path: |-
```

```
      d3m.primitives.layer.add.KerasWrap
    name: add
    digest: af4e6117e493e1903f8f4e1803fa8f599d33295f13b8cf36c4bbb853eeaed405
  hyperparams:
    # Here we connect this layer to two other layers.
    previous_layers:
      type: PRIMITIVE
      data:
      - 7
      - 4
# Step 9. Layer. Used as a hyper-parameter value.
- type: PRIMITIVE
  primitive:
    id: 6b1cceeb-a3be-4d7d-809a-052ae72b2b13
    version: 0.1.0
    python_path: |-
      d3m.primitives.layer.convolution_2d.KerasWrap
    name: convolution_2d
    digest: 2af0acc46d4c2925f550bc88f0927afe11c78008f28f767d740d8146cc2af52d
  hyperparams:
    filters:
      type: VALUE
      data: 10
    padding:
      type: VALUE
      data: same
    # Here we connect this layer to another layer.
    previous_layer:
      type: PRIMITIVE
      data: 8
    strides:
      type: VALUE
      data: 1
# Step 10. Layer. Used as a hyper-parameter value.
- type: PRIMITIVE
  primitive:
    id: 288162cd-b71d-418d-9555-cc177c5f592e
    version: 0.1.0
    python_path: |-
      d3m.primitives.layer.batch_normalization.KerasWrap
    name: batch_normalization
    digest: 8c1d94282d1747a5fb639d3b3ec3934c252e706ff38396a61439ad56186a381d
  hyperparams:
    # Here we connect this layer to another layer.
    previous_layer:
      type: PRIMITIVE
      data: 9
# Step 11. Layer. Used as a hyper-parameter value.
- type: PRIMITIVE
  primitive:
```

```
    id: 7a76922c-bf6f-37ce-9e58-1d3315382506
    version: 0.1.0
    python_path: |-
      d3m.primitives.layer.dropout.KerasWrap
    name: dropout
    digest: 52b8acabf95b3781c60a26d613a6f876ec583c9fb0ea018bb4780f0390248633
  hyperparams:
    # Here we connect this layer to another layer.
    previous_layer:
      type: PRIMITIVE
      data: 10
# Step 12. Layer. Used as a hyper-parameter value.
- type: PRIMITIVE
  primitive:
    id: c97c5620-6274-4a5f-83b0-1c090c68ac7b
    version: 0.1.0
    python_path: |-
      d3m.primitives.layer.add.KerasWrap
    name: add
    digest: af4e6117e493e1903f8f4e1803fa8f599d33295f13b8cf36c4bbb853eeaed405
  hyperparams:
    # Here we connect this layer to two other layers.
    previous_layers:
      type: PRIMITIVE
      data:
      - 11
      - 8
# Step 13. Layer. Used as a hyper-parameter value.
- type: PRIMITIVE
  primitive:
    id: e8acc97a-7868-427e-b022-e2aa51116d19
    version: 0.1.0
    python_path: |-
      d3m.primitives.layer.flatten.KerasWrap
    name: flatten
    digest: 642ce85496ad05ce664e72ad5d1a96d7fa6b819c6bc494d5b2e1897d7dcd930d
  hyperparams:
    # Here we connect this layer to another layer.
    previous_layer:
      type: PRIMITIVE
      data: 12
# Step 14. Layer. Used as a hyper-parameter value.
- type: PRIMITIVE
  primitive:
    id: eb6a13fd-c3e6-4407-a15f-280905e6243e
    version: 0.1.0
    python_path: |-
      d3m.primitives.layer.dense.KerasWrap
    name: dense
    digest: 6a41e848de2dc411d65837e7ca73a40a5a9c21993c6e194a5fea44a2ee924675
```

```
232   hyperparams:
233     # Here we connect this layer to another layer.
234     previous_layer:
235       type: PRIMITIVE
236       data: 13
237     units:
238       type: VALUE
239       data: 100
240 # Step 15. Layer. Used as a hyper-parameter value.
241 - type: PRIMITIVE
242   primitive:
243     id: eb6a13fd-c3e6-4407-a15f-280905e6243e
244     version: 0.1.0
245     python_path: |-
246       d3m.primitives.layer.dense.KerasWrap
247     name: dense
248     digest: 6a41e848de2dc411d65837e7ca73a40a5a9c21993c6e194a5fea44a2ee924675
249   hyperparams:
250     # Here we connect this layer to another layer.
251     previous_layer:
252       type: PRIMITIVE
253       data: 14
254     units:
255       type: VALUE
256       data: 100
257 # Step 16. Layer. Used as a hyper-parameter value.
258 - type: PRIMITIVE
259   primitive:
260     id: eb6a13fd-c3e6-4407-a15f-280905e6243e
261     version: 0.1.0
262     python_path: |-
263       d3m.primitives.layer.dense.KerasWrap
264     name: dense
265     digest: 6a41e848de2dc411d65837e7ca73a40a5a9c21993c6e194a5fea44a2ee924675
266   hyperparams:
267     # Here we connect this layer to another layer.
268     previous_layer:
269       type: PRIMITIVE
270       data: 15
271     units:
272       type: VALUE
273       data: 10
274 # Step 17. Primitive which trains the neural network as a classifier.
275 - type: PRIMITIVE
276   primitive:
277     id: f8b81d1a-3e22-4edf-aa99-15bcbe827954
278     version: 0.1.0
279     python_path: |-
280       d3m.primitives.learner.model.KerasWrap
281     name: model
```

```
    digest: aa0f3fafbbb7eb159b694c88a4c5576c4f423ea0d799e0fc00c6c81ac7d3f406
  arguments:
    inputs:
      type: CONTAINER
      data: steps.2.produce
    outputs:
      type: CONTAINER
      data: steps.2.produce
  outputs:
  - id: produce
  hyperparams:
    # Here we connect the loss to use.
    loss:
      type: PRIMITIVE
      data: 3
    model_type:
      type: VALUE
      data: classification
    optimizer:
      type: VALUE
      data:
        choice: Adam
        amsgrad: false
        beta_1: 0.9
        beta_2: 0.999
        decay: 0.0
        epsilon: 0.0
        lr: 0.01
    # Here we connect the neural network to train.
    previous_layer:
      type: PRIMITIVE
      data: 16
    return_result:
      type: VALUE
      data: new
```

Bibliography

[1] Shawkat Ali and Kate A. Smith. "On Learning Algorithm Selection for Classification". In: *Appl. Soft Comput.* 6.2 (Jan. 2006), pp. 119–138. ISSN: 1568-4946. DOI: 10.1016/j.asoc.2004.12.002. URL: http://dx.doi.org/10.1016/j.asoc.2004.12.002.

[2] Edgar Anderson. "The species problem in Iris". In: *Annals of the Missouri Botanical Garden* 23.3 (1936), pp. 457–509.

[3] Bowen Baker et al. "Designing neural network architectures using reinforcement learning". In: *arXiv preprint arXiv:1611.02167* (2016). URL: https://arxiv.org/abs/1611.02167.

[4] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. *YAML Ain't Markup Language*. Oct. 2009. URL: https://yaml.org/spec/ (visited on 05/14/2019).

[5] James Bergstra, Dan Yamins, and David D Cox. "Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms". In: *Proceedings of the 12th Python in science conference*. Citeseer. 2013, pp. 13–20.

[6] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. RFC Editor, Dec. 2017. URL: http://www.rfc-editor.org/rfc/rfc8259.txt.

[7] François Chollet et al. *Keras*. 2015. URL: https://keras.io (visited on 05/14/2019).

[8] Cornell University. *arXiv*. URL: https://arxiv.org/ (visited on 05/14/2019).

[9] Joe Davison et al. *DEvol: Deep Neural Network Evolution*. 2017. URL: https://github.com/joeddav/devol (visited on 05/14/2019).

[10] Iddo Drori et al. "AlphaD3M: Machine learning pipeline synthesis". In: *AutoML Workshop at ICML*. 2018.

[11] Thomas Elliott. *The State of the Octoverse: machine learning*. 2019. URL: https://github.blog/2019-01-24-the-state-of-the-octoverse-machine-learning/ (visited on 05/14/2019).

[12] Radwa Elshawi, Mohamed Maher, and Sherif Sakr. "Automated Machine Learning: State-of-The-Art and OpenChallenges". In: *arXiv preprint arXiv:1906.02287* (2019). URL: https://arxiv.org/abs/1906.02287.

[13] Matthias Feurer et al. *ConfigSpace*. URL: https://github.com/automl/ConfigSpace (visited on 05/14/2019).

[14] Matthias Feurer et al. "Auto-sklearn: Efficient and Robust Automated Machine Learning". In: *Automated Machine Learning*. Springer, 2019, pp. 113–134.

[15] Ronald A. Fisher. "The use of multiple measurements in taxonomic problems". In: *Annals of eugenics* 7.2 (1936), pp. 179–188.

[16] Francis Galiegue and Kris Zyp. *JSON Schema: interactive and non interactive validation*. Feb. 2013. URL: `https://tools.ietf.org/html/draft-fge-json-schema-validation-00` (visited on 05/14/2019).

[17] P. Gijsbers et al. "An Open Source AutoML Benchmark". In: *arXiv preprint arXiv:1907.00909* (2019). Accepted at AutoML Workshop at ICML 2019. URL: `https://arxiv.org/abs/1907.00909`.

[18] Google. *Cloud AutoML*. URL: `https://cloud.google.com/automl/` (visited on 05/14/2019).

[19] Isabelle Guyon et al. "Analysis of the AutoML Challenge Series 2015-2018". In: *Automatic Machine Learning: Methods, Systems, Challenges*. Ed. by Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. In press, available at http://automl.org/book. Springer, 2018. Chap. 10, pp. 191–236.

[20] *H2O's AutoML*. URL: `http://docs.h2o.ai/h2o/latest-stable/h2o-docs/automl.html` (visited on 05/14/2019).

[21] Haifeng Jin, Qingquan Song, and Xia Hu. *Auto-Keras: Efficient Neural Architecture Search with Network Morphism*. June 27, 2018. arXiv: `cs.LG/1806.10282 [cs.LG]`.

[22] Brent Komer, James Bergstra, and Chris Eliasmith. "Hyperopt-Sklearn". In: *Automatic Machine Learning: Methods, Systems, Challenges*. Ed. by Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. Springer, 2018. Chap. 5, pp. 105–121. URL: `http://automl.org/book`.

[23] Lars Kotthoff et al. "Auto-WEKA: Automatic model selection and hyperparameter optimization in WEKA". In: *Automatic Machine Learning: Methods, Systems, Challenges*. Ed. by Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. Springer, 2018. Chap. 4, pp. 89–103. URL: `http://automl.org/book`.

[24] *Kubeflow*. URL: `https://www.kubeflow.org/` (visited on 05/14/2019).

[25] Paul Leach, Michael Mealling, and Rich Salz. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122. RFC Editor, July 2005. URL: `http://www.rfc-editor.org/rfc/rfc4122.txt`.

[26] *LinkedIn Workforce Report, United States, August 2018*. Aug. 2018. URL: `https://economicgraph.linkedin.com/resources/linkedin-workforce-report-august-2018` (visited on 05/14/2019).

[27] Mohamed Maher and Sherif 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.

[28] Hector Mendoza et al. "Towards Automatically-Tuned Deep Neural Networks". In: *Automatic Machine Learning: Methods, Systems, Challenges*. Ed. by Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. Springer, 2018. Chap. 7, pp. 145–161. URL: http://automl.org/book.

[29] Microsoft. *Neural Network Intelligence*. 2018. URL: https://github.com/Microsoft/nni (visited on 05/14/2019).

[30] Mitar Milutinovic et al. "End-to-end Training of Differentiable Pipelines Across Machine Learning Frameworks". In: *NIPS Workshop on Autodiff*. 2017.

[31] Felix Mohr, Marcel Wever, and Eyke Hüllermeier. "ML-Plan: Automated machine learning via hierarchical planning". In: *Machine Learning* 107.8 (Sept. 2018), pp. 1495–1515. ISSN: 1573-0565. DOI: 10.1007/s10994-018-5735-z. URL: https://doi.org/10.1007/s10994-018-5735-z.

[32] Randal S. Olson and Jason H. Moore. "TPOT: A Tree-based Pipeline Optimization Tool for Automating Machine Learning". In: *Automatic Machine Learning: Methods, Systems, Challenges*. Ed. by Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. Springer, 2018. Chap. 8, pp. 163–173. URL: http://automl.org/book.

[33] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[34] Yonghong Peng et al. "Improved dataset characterisation for meta-learning". In: *International Conference on Discovery Science*. Springer. 2002, pp. 141–152.

[35] Bernhard Pfahringer, Hilan Bensusan, and Christophe Giraud-carrier. "Meta-learning by landmarking various learning algorithms". In: *in Proceedings of the 17th International Conference on Machine Learning, ICML'2000*. Morgan Kaufmann, 2000, pp. 743–750.

[36] Hieu Pham et al. "Efficient neural architecture search via parameter sharing". In: *arXiv preprint arXiv:1802.03268* (2018). URL: https://arxiv.org/abs/1802.03268.

[37] Fábio Pinto, Carlos Soares, and João Mendes-Moreira. "Towards automatic generation of metafeatures". In: *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer. 2016, pp. 215–226.

[38] Herilalaina Rakotoarison, Marc Schoenauer, and Michèle Sebag. "Automated Machine Learning with Monte-Carlo Tree Search". In: *arXiv preprint arXiv:1906.00170* (2019). URL: https://arxiv.org/abs/1906.00170.

[39] David Reinsel, John Gantz, and John Rydning. *The digitization of the world: from edge to core*. Whitepaper US44413318. International Data Corporation, Nov. 2018. URL: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf.

[40] Alex G. C. de Sá, Alex A. Freitas, and Gisele L. Pappa. "Automated Selection and Configuration of Multi-Label Classification Algorithms with Grammar-Based Genetic Programming". In: *Parallel Problem Solving from Nature – PPSN XV*. Ed. by Anne Auger et al. Cham: Springer International Publishing, 2018, pp. 308–320. ISBN: 978-3-319-99259-4.

[41] Alex G. C. de Sá et al. "RECIPE: A Grammar-Based Framework for Automatically Evolving Classification Pipelines". In: *Genetic Programming*. Ed. by James McDermott et al. Cham: Springer International Publishing, 2017, pp. 246–261. ISBN: 978-3-319-55696-3.

[42] Salesforce. *TransmogrifAI*. URL: `https://transmogrif.ai/` (visited on 05/14/2019).

[43] Zeyuan Shang et al. "Democratizing data science through interactive curation of ml pipelines". In: *Proceedings of the 2019 International Conference on Management of Data*. ACM. 2019, pp. 1171–1188.

[44] Wade Shen. "Data-driven discovery of models (d3m)". In: *Defense Advanced Research Projects Agency, Arlington, VA* (2016).

[45] Christian Steinrucken et al. "Automatic Machine Learning: Methods, Systems, Challenges". In: ed. by Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. In press, available at http://automl.org/book. Springer, 2018. Chap. 9, pp. 175–188.

[46] Peter Amstutz; Michael R. Crusoe; Nebojša Tijanić; Brad Chapman; John Chilton; Michael Heuer; Andrey Kartashov; John Kern; Dan Leehr; Hervé Ménager; Maya Nedeljkovich; Matt Scales; Stian Soiland-Reyes; Luka Stojanovic. *Common Workflow Language, v1.0*. 2016. DOI: `10.6084/m9.figshare.3115156.v2`. URL: `https://doi.org/10.6084/m9.figshare.3115156.v2`.

[47] T. Swearingen et al. "ATM: A distributed, collaborative, scalable system for automated machine learning". In: *2017 IEEE International Conference on Big Data (Big Data)*. Dec. 2017, pp. 151–162. DOI: `10.1109/BigData.2017.8257923`. URL: `https://github.com/HDI-Project/ATM`.

[48] *Thyroid disease records supplied by the Garavan Institute and J. Ross Quinlan, New South Wales Institute, Sydney, Australia*. 1987. URL: `https://www.openml.org/d/38` (visited on 05/14/2019).

[49] Joaquin Vanschoren et al. "OpenML: Networked Science in Machine Learning". In: *SIGKDD Explorations* 15.2 (2013), pp. 49–60. DOI: `10.1145/2641190.2641198`. URL: `http://doi.acm.org/10.1145/2641190.2641198`.

[50] Wei Wang et al. "Rafiki: machine learning as an analytics service system". In: *Proceedings of the VLDB Endowment* 12.2 (2018), pp. 128–140.

[51] Wikipedia contributors. *Category: Machine learning — Wikipedia, The Free Encyclopedia*. 2019. URL: `https://en.wikipedia.org/wiki/Category:Machine_learning` (visited on 05/14/2019).

[52] Chengrun Yang et al. "OBOE: Collaborative Filtering for AutoML Model Selection". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM. 2019, pp. 1173–1183.

[53] Barret Zoph and Quoc V Le. "Neural architecture search with reinforcement learning". In: *arXiv preprint arXiv:1611.01578* (2016). URL: `https://arxiv.org/abs/1611.01578`.