juliacon

Designing Machine Learning Pipeline Toolkit for AutoML Surrogate Modeling Optimization

Paulito P. Palmes[1], Akihiro Kishimoto[2], Radu Marinescu[1], Parikshit Ram[3], and Elizabeth Daly[1]

[1]IBM Research Europe, Dublin Research Lab
[2]IBM Research Japan
[3]IBM Research USA

ABSTRACT

The pipeline optimization problem in machine learning requires simultaneous optimization of pipeline structures and parameter adaptation of their elements. Having an elegant way to express these structures can help lessen the complexity in the management and analysis of their performances together with the different choices of optimization strategies. With these issues in mind, we created the AutoMLPipeline (AMLP) toolkit which facilitates the creation and evaluation of complex machine learning pipeline structures using simple expressions. We use AMLP to find optimal pipeline signatures, datamine them, and use these datamined features to speed-up learning and prediction. We formulated a two-stage pipeline optimization with surrogate modeling in AMLP which outperforms other AutoML approaches with a 4-hour time budget in less than 5 minutes of AMLP computation time.

Keywords

Julia, Optimization, AutoMLPipeline, AutoML, ML Pipeline Search, Feature Engineering, Modeling, Classification, Prediction, Regression

1. Introduction

The typical machine learning (ML) workflow for classification and prediction requires some or a combination of the following preprocessing steps:

—*data cleaning*: Imputation, Interpolation, etc.

—*feature transformation*: Normalization, Scaling, One-hot encoding, etc.

—*feature selection*: Anova, Correlation, etc.

—*feature extraction*: PCA, ICA, FactorAnalysis, etc.

—*modeling*: RandomForest, XGBoost, SVM, AdaBoost, etc.

Each step has several choices of functions to use, together with their corresponding parameters to initialize. Optimizing the performance of the entire pipeline is a combinatorial search for the proper order and combination of preprocessing steps, optimization of their corresponding parameters, and a search for the optimal model and its hyper-parameters.

Because of close dependencies among these various steps to perform optimization, the entire process is commonly called combined algorithm selection and hyper-parameter optimization or CASH [9, 25]. Having an elegant way to express pipeline structures can help lessen the complexity in the management and analysis of the wide-array of choices of optimization routines. AMLP (Auto ML Pipeline) toolkit aims to address this issue by supporting the following features:

—Pipeline API that allows high-level description of modeling and preprocessing workflow to support explainability and easy dataminining of optimal pipeline signatures

—Symbolic pipeline parsing to facilitate easy expression of complex pipeline structures

—Common API wrappers for ML libs including *scikit-learn* [21], *caret* [17], etc.

—Easily extensible architecture by overloading two interfaces: `fit!` and `transform!`

—High-level implementation of meta-ensembles for the ensemble composition of ensembles (recursively if needed) for robust prediction routines.

—Categorical and numerical feature selectors for specialized preprocessing routines based on types.

—High-level code of parallelized cross-validation (CV) routines by leveraging on the multi-threading and distributed computing features in *Julia* [6].

AMLP's main feature is the use of relatively compact symbolic expression in pipeline composition. For instance, a pipeline expression to extract the numerical features (`numf`) for `pca` decomposition, concatenated with one-hot encoding (`ohe`) of categorical features (`catf`) of a given data for RandomForest (`rf`) modeling can be written as:

```
pipe = (numf |> pca) + (catf |> ohe) |> rf
```

AMLP (AutoMLPipeline) is written in *Julia* language [6] to leverage on the latter's support of modern features such as: multiple dispatch, just-in-time (*JIT*) compilation, multi-threading, parallel and distributed computing, dynamic types, coroutines, interactive shell, high-performance, code specialization, metaprogramming, and type inference. Pure *Julia* ML models in AMLP is easy to maintain and extend by relying on just one programming language because Julia's *JIT* avoids the need to implement some performance-critical tasks in C/C++.

1.1 Related Work

AMLP's main design was inspired by the *Orchestra* [14] package. Early ML packages for *Julia* are wrappers of existing ML toolkits from *scikit-learn* in *Python* [26] and *caret* in *R* [22]. By having common APIs across different ML implementations, *Orchestra* showed that one can mix and match ML preprocessing routines from *caret*, *scikit-learn*, and *Julia* seamlessly in a pipeline by leveraging on *PyCall* [15] and *RCall* [3] wrappers for *Python* and *R*, respectively. This is a convenient package because there are many cases where implementations of ML functions can be found exclusively in either *scikit-learn* or *caret*. Having both libraries available for perusal is a great productivity boost for machine learners in *Julia*. Unfortunately, *Orchestra* package has not been maintained for more than 6 years, but its legacy lives in the AMLP's design. Another popular toolkit in *Julia* ML ecosystem is the *MLJ* [8] package developed at the Alan Turing Institute. It serves as a meta-package of ML libraries both native to *Julia* as well as *scikit-learn*.

IBM [1] *Lale* [12] is a toolkit in *Python* with sophisticated support of pipeline optimization useful for AutoML [18] algorithm development. *Lale* builds on *scikit-learn* and extends it by supporting features such as: automation, correctness checks, and interoperability. *Lale* has consistent high-level interface for popular AutoML algorithms such as: *Hyperopt* [5, 4], *GridSearchCV* [21], and *SMAC* [13]. It makes it possible to have an easy comparison of the different AutoML algorithms for benchmarking, and research, as well as for applications. While AMLP is at the early stage of development, its pipeline architecture can be used as building blocks for the development of future AutoML algorithms similar to *Lale* in *Julia*'s ecosystem.

2. AMLP Architecture

The code in Listing 1 describes the abstract type hierarchy used in the AMLP toolkit. At the top of the hierarchy is the `Machine` abstraction with abstract functions `fit!` and `transform!` to be implemented by its subtypes. Calling both functions in sequence are handled by `fit_transform!`. A `Machine` has two subtypes: `Computer` and `Workflow`. A `Computer` can either be a `Transformer` or a `Learner` while a `Workflow` can be either a `Pipeline` or `ComboPipeline`.

All `Machine` subtypes are expected to define their own `fit!` and `transform!` before they can be part of the elements in a `Workflow`. Any `Workflow` instance uses these two functions during training, feature transformation, prediction, and cross-validation (CV). By convention in *Julia* [6], functions ending with exclamation mark (`!`) mutate the value(s) of their arguments which is done by `fit!` and `transform!` to mutate the states and parameters of the `Machine` instance in their arguments.

The AMLP workflow is based on the *Orchestra* [14] package which drew its inspiration from the *Unix* pipeline [16]. The main elements of a pipeline are a series of `Computer` instances with each instance performing a specific task. A typical pipeline for classification or prediction contains a series of transformers terminated by a learner. During `fit_transform!`, these series of transformers act as filters converting the input data into the same mathematical or statistical transform before feeding them to the learner for training and prediction. In a pipeline expression where the last element is not a learner, `fit_transform!` acts as a feature filter or transformer only.

```
1  abstract type Computer     <: Machine
2  abstract type Workflow     <: Machine
3  abstract type Learner      <: Computer
4  abstract type Transformer  <: Computer
5
6  function fit!(mc::Machine, input::DataFrame,
       output::Vector)
7      error(typeof(mc)," has no implementation.")
8  end
9
10 function transform!(mc::Machine, input::DataFrame)
11     error(typeof(mc)," has no implementation.")
12 end
13
14 function fit_transform!(mc::Machine, input::
       DataFrame, output::Vector)
15     fit!(mc,input,output)
16     transform!(mc,input)
17 end
```

Listing 1: Abstract Type Hierarchy

The `Pipeline` instance processes linearly the sequence of information among its elements. Its `fit!` implementation iteratively calls its elements' `fit_transform!` passing the output from one `Computer` instance to the next `Computer` instance in the sequence. Aside from sequential operations, two or more workflows can be combined using the `ComboPipeline` instance.

The `fit!` and `transform!` functions for a `Learner` are equivalent to training and prediction, respectively. A `Learner` instance implements the algorithm to acquire the mapping between its input and output during `fit!` and applies the learned model to perform prediction during `transform!`.

For a `Transformer`, `fit!` and `transform!` are preprocessing operations to convert the training data into the same mathematical or statistical transform. Depending on the function used, `fit!` can be a `noop` (no operation) like in `sqrt` or `log` transform. On the other hand, PCA or ICA uses `fit!` to compute and store the coefficient matrix derived from its training input and applies the same matrix in `transform!`.

2.1 AMLP workflow

The code in Listing 2 depicts the typical usage of the AMLP toolkit. Lines 1–2 loads the AMLP package and the pro football dataset [23, 27], respectively. The aim is to predict whether the game is held at *home* or *away* based on the following (C)ategorical or (N)umerical features: FavoritePoints (N), UnderdogPoints (N), Pointspread (N), FavoriteName (C), UnderdogName (C), Year (N), Week (N), Weekday (C), and Overtime (C).

Lines 5–10 of Listing 2 create instances of the preprocessing elements to be included in the pipeline. The toolkit uses the *scikit-learn* [21] wrappers, `SKPreprocessor` and `SKLearner`, to instantiate its transformers and learner: `PCA`, `MinMaxScaler`, and `RandomForestClassifier`. Other elements such as: `OneHotEncoder`, `CatFeatureSelector`, and `NumFeatureSelector` are implemented in pure *Julia*.

In line 13 of Listing 2, the expression:

[1] IBM and the IBM logo are trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on `http://ibm.com/trademark`.

```
using AMLP            # load package
(X,Y) = getprofb()    # load dataset

# Instantiate the pipeline elements
pca   = SKPreprocessor("PCA")
mx    = SKPreprocessor("MinMaxScaler")
ohe   = OneHotEncoder()
catf  = CatFeatureSelector()  # categorical columns
numf  = NumFeatureSelector()  # numerical columns
rf    = SKLearner("RandomForestClassifier")

# Setup the ML pipeline
pipe = ((catf |> ohe) + (numf |> mx |> pca)) |> rf

# train and predict
prediction = fit_transform!(pipe,X,Y)

# compute avg accuracy by 10-fold cv
performance = crossvalidate(pipe,X,Y)
```

Listing 2: AMLP toolkit sample usage

```
pipe = ((catf |> ohe) + (numf |> mx |> pca)) |> rf
```

describes the preprocessing workflow of the input data with the `RandomForest` classifier as the learner. The expression, `(x |> f)`, is equivalent to `f(x)` while the expression, `(x + y)`, signifies feature union: $\cup$`(x,y)`. The expression, `(catf |> ohe)`, selects columns with categorical features and transform them into one-hot representation. In `(numf |> mx |> pca))`, numerical columns are selected then scaled by `MinMaxScaler` and finally embedded in `PCA` subspace. Both features are then concatenated to become the input features of the `RandomForest` model (`rf`).

Line 13 of Listing 2 can also be written using function calls as:

```
Pipeline(ComboPipeline(Pipeline(catf, ohe),
         Pipeline(Pipeline(numf, mx), pca)), rf)
```

This latter expression looks less understandable compared to the former. The simplicity of AMLP pipeline expression becomes more significant in developing AutoML algorithms or in data mining optimal pipelines among different datasets for surrogate modeling [19]. More examples of AMLP usage including its extensive documentation and source code can be found in its open-source github resource [to be referenced if accepted].

3. Pipeline search strategy benchmark

The search for an optimal pipeline can be treated as a matching problem between a group of learners and a group of preprocessing pipelines. The objective is to find the optimal pair of preprocessing pipeline and learner such that their corresponding CV accuracy is the best among the rest of the pairs. We refer to any of these pairs as an ML pipeline (MLPL) in contrast to a preprocessing pipeline (PRPL) where the last element is not a learner. The time complexity to search for all possible combinations of MLPL is dependent on the number of learners as well as the size of elements in the PRPL: `n(learners) * n(PRPL)`.

3.1 Two-stage strategies

To avoid the brute-force approach of exhaustive search, we attack the problem by decomposing the search into two stages:

—*One-all* search strategy uses an arbitrarily chosen learner as the engine of CV in searching for the best preprocessing pipeline performance in the first stage. The second stage proceeds by using the best pipeline found in the first stage to search for the best learner.

—*All-one* search strategy uses an arbitrarily chosen pipeline as the base pipeline to search for the best learner during the first stage. The second stage uses the best learner found in the first stage to search for the best pipeline.

We call the first strategy *one-all* to indicate the utilization of a surrogate learner to evaluate all pipelines in the first stage. Similarly, we call the second strategy *all-one* to indicate the evaluation of all learners under a surrogate pipeline. To aid the comparison, we also implemented the *all-all* strategy which is an exhaustive search of the performance of all combinations of learners and PRPL to get the best MLPL.

In our implementation, we call the pipeline consisting of scaler (sc) and feature extractor (fx) a one-block PRPL, expressed as: `(sc |> fx)`. In the experiments, we use the maximum of two-block PRPL, `((sc1 |> fx1) + (sc2 |> fx2))`, where `sc1 != sc2` or `fx1 != fx2`. The pipeline can easily be extended by adding more PRPL blocks depending on the complexity of the dataset at the expense of longer computation time.

3.2 Experimental setup

Table 1 summarizes the statistical features of the 12 OpenML datasets [27] used in the experiment. Also included are the best results in a 4-hour time budget discussed in the review paper of [28]. The comparison involves 5 AutoML approaches, namely: TPOT [20], Auto-Sklearn [11, 10], ATM [24], Hyperopt-Sklearn [5, 4], and Random Search [2]. The best results among the 5 AutoML approaches from each dataset will be used as the baseline comparison in the results and discussion of the succeeding section of the paper. Among these 5 approaches, TPOT and ATM have dominated the rankings in terms of best performance based on the CV classification errors on the 12 datasets as shown in the last two columns.

The AMLP experiments use the following transformers and learners including `Noop` (no operation):

—6 scalers: Standard, MinMax, Robust, Normalizer, PowerTransfomer, Noop

—4 feature extractors: PCA, ICA, FactorAnalysis, Noop

—6 learners: RandomForest, AdaBoost, DecisionTree, GradientBoosting, LinearSvm, RbfSvm

For the one-block PRPL experiment, the size of all MLPL for the exhaustive search will be $6 \times 4 \times 6 = 144$. A relatively small size that can easily be evaluated to search for the optimal solution which is ideal for experimental comparison with the two-stage strategies. However, the size of all combinations of MLPL suddenly explodes to more than 3,000 pipelines by just adding another PRPL block. With 10-fold CV per MLPL, the exhaustive search requires 1440 CV operations for a one-block MLPL and more than 30,000 CV operations for a two-block MLPL.

Table 1. OpenML Datasets

Dataset	NAs	Size	Cat	Num	Rows	Cols	Class	Best Result	AutoML
analcatdata	0	22K	5	0	9873	5	6	75.10 ± 02.59	ATM
breast-w	16	18K	1	9	699	10	2	01.44 ± 02.22	ATM
cmc	0	30K	8	2	1473	10	3	43.33 ± 01.52	TPOT
credit-app	67	39K	10	6	690	16	2	11.11 ± 03.52	ATM
eucalyptus	448	74K	6	14	736	20	5	35.41 ± 05.02	AutoSk
first-order	0	2.6M	1	51	6118	52	6	38.48 ± 01.95	TPOT
GestureSeg	0	3.3M	1	32	9873	33	5	32.78 ± 03.59	TPOT
jm1	25	826K	1	21	10885	22	2	18.13 ± 01.55	TPOT
profb	1200	25K	5	5	672	10	2	32.43 ± 05.84	TPOT
plants-shape	0	892K	1	64	1600	65	100	36.54 ± 03.53	AutoSk
sick	6064	300K	23	7	3772	30	2	01.30 ± 00.87	TPOT
soybean	2337	171K	36	0	683	36	19	06.59 ± 02.79	ATM

The *one-all* search strategy requires $6 \times 4 = 24$ one-block PRPL matched to one learner for the first stage. The second stage requires matching the best pipeline against 6 learners. In this strategy, a total of 30 MLPL pipelines are needed to get the optimal or suboptimal solution. With 10-fold CV per MLPL, *one-all* requires 300 CV operations which is 4.8x smaller compared to the 1440 CV operations needed for the exhaustive search strategy.

For the *all-one* strategy, 6 learners are matched to a single pipeline in the first stage. The best learner found is then matched to $6 \times 4 = 24$ PRPL in the second stage. In total, the *all-one* uses 30 MLPL which translates to 300 CV operations similar to the *one-all*. The actual runtime performance for *one-all* and *all-one* in the same dataset depends mostly on the best learner performance in *all-one* and the surrogate learner performance in *one-all*.

All datasets undergo common cleaning workflow in all experiments. The entire process is summarized in the following:

```
bp = colnarm |> rownarm |> ((catf  |> ohe) + numf)
```

For each dataset, column-wise removal (`colnarm`) of missing values (*NA*) is carried out on those columns with *NA* count greater than 10% of the row size followed by row-wise removal (`rownarm`) of any remaining *NA*s. After *NA* removal, categorical features (`catf`) of the dataset are transformed to one-hot (`ohe`) representation and concatenated with its numerical features (`numf`).

3.3 Results and Discussion

Table 2 summarizes the performance of *all-all* strategy. The rank represents the relative performance of the proposed algo relative to the baseline performance of the best algorithm in Table 1. The sign of $=$, $<$, and $>$ indicate whether the rank performance is significantly different from the baseline at $\alpha = 0.05$ level of significance by t-test. For example, rank $1, =$ indicates that the proposed algo is as good as the baseline in Table 1. To differentiate the rank where the baseline or the proposed algorithm is significantly inferior or superior, the $<$ and $>$ are used, respectively. For example, among the datasets, the proposed algo in Table 2 is significantly superior to the baseline algos in *first-order-theo*, *profb*, and *plants-shape* but significantly inferior in *sick* and *soybean* datasets. The best algos in the baseline has similar performance with the *all-all* strategy in the rest of the datasets.

The metric of performance is based on the average classification error (*AvgErr*) using 10-fold CV. The ranking is based on its *AvgErr* performance in comparison to the 5 AutoML approaches [28] summarized in Table 1. The *all-all* median overall rank is 1 using one-block pipeline. Its median runtime to perform one-block exhaustive search is 0.23 hour or 13.8 minutes per dataset. This result is encouraging because the median runtime is significantly less compared to the 5 AutoML algorithms with a 4-hour budget. Closely examining the runtime of each dataset, *plants-shape* dataset requires 4.29 hours to finish while the rest require at most 1.87 hours to run. Take note, however, that the comparison for runtime is not a fair comparison, as we did not have the time to repeat the experiments done in [28] using the same machine with our two-stage strategies. Our experiments were conducted using 2017 Model of MacBook Pro with 2.8 GHz Quad-Core Intel Core i7 and 16 GB of RAM.

The solution can be improved further by searching all two-block pipelines, but the size of the search space becomes a limiting factor. Ideally, if either *one-all* or *all-one* has similar optimal results with that of *all-all*, the runtime required will be significantly less to attack the two-block MLPL. The succeeding discussions of the results of experiments aim to find out if there is a significant runtime saving in either one or both of the two-stage strategies based on their performance relative to the exhaustive *all-all* strategy.

Tables 3 and 4 show the performance of *one-all* and *all-one* strategies, respectively, for one-block MLPL. Their ranking is based on their performances against the 5 AutoML algorithms. Table 3 summarizes the performance of *one-all* using the `RandomForest` surrogate. All other learners tested as surrogate have similar 2.0 to 2.5 median rank. The *all-one* strategy in Table 4 uses the pipeline expression, `(((catf |> ohe) + numf)) |> robustsc`, as the surrogate pipeline to search for the best learner in the first stage. The expression indicates one-hot encoding (ohe) of the categorical features (`catf`) combined with the numerical features (`numf`) and transformed by robust scaling (`robustsc`).

Comparing Tables 2, 3 and 4, the *all-one* strategy median rank of 1 is similar to *all-all* and *one-all* but the *all-one* median validation error of 27.75 ± 3.69 is better than *all-all* and *one-all*. The biggest advantage of *all-one* is its speed which is significantly faster to both *all-all* and *one-all*. Its median time duration for all 12 datasets is just 0.05 hour (3 minutes) compared to the runtime median of 0.23 hour (13.8 minutes) for *all-all* and 0.11 hour (6 minutes) for *one-all*.

Table 2. *All-all* (one-block) search strategy

					Block		
Dataset	Rank	AvgErr	Std	Time	Sc	Fx	Learner
analcatdata-dmf	1, =	76.52	4.92	0.13	normalizer	noop	rbfsvc
breast-w	1, =	02.34	1.71	0.08	minmax	noop	rbfsvc
cmc	1, =	42.57	3.37	0.18	noop	noop	gb
credit-approval	1, =	11.42	2.60	0.09	normalizer	noop	rf
eucalyptus	1, =	36.02	6.61	0.17	noop	noop	gb
first-order-theo	1, >	36.61	1.44	1.27	noop	noop	rf
GestureSeg	1, =	31.96	1.28	1.87	minmax	noop	rf
jm1	1, =	17.97	0.61	1.84	stdsc	noop	rf
profb	1, >	25.59	5.15	1.84	stdsc	noop	lsvc
plants-shape	1, >	30.25	3.03	4.29	powertf	pca	rf
sick	4, <	06.40	0.70	0.41	noop	noop	gb
soybean	5, <	14.62	2.48	0.27	robustsc	noop	lsvc
Median	1	27.92	2.54	0.23			

Table 3. *One-all* (one-block)

					Block		
Dataset	Rank	AvgErr	Std	Time	Sc	Fx	Lr
analcatdata-dmf	1, =	78.42	5.77	0.35	minmax	factA	ada
breast-w	1, =	2.78	1.88	0.02	stdsc	pca	rbfsvc
cmc	4, <	45.76	2.94	0.02	norm	noop	gb
credit-approval	1, =	11.34	4.98	0.09	norm	noop	rf
eucalyptus	1, =	36.79	7.61	0.09	noop	noop	rf
first-order-theo	1, =	36.89	1.68	0.32	powertf	noop	rf
GestureSeg	1, =	32.04	1.16	0.15	noop	noop	rf
jm1	1, =	18.01	0.51	0.34	powertf	noop	rf
profb	1, =	28.87	4.86	0.09	noop	ica	rf
plants-shape	1, >	30.80	4.44	0.13	powertf	pca	rf
sick	4, <	6.60	0.66	0.06	robustsc	pca	rf
soybean	5, <	15.29	4.78	0.34	robustsc	noop	lsvc
Median	1	29.84	3.69	0.11			

Table 4. *All-one* (one-block)

					Block		
Dataset	Rank	AvgErr	Std	Time	Sc	Fx	Lr
analcatdata-dmf	1, =	77.92	3.42	0.01	noop	noop	rbfsvc
breast-w	1, =	2.34	2.08	0.01	robustsc	noop	rbfsvc
cmc	1, =	42.36	4.14	0.08	stdsc	noop	gb
credit-approval	1, =	11.26	3.72	0.03	norm	noop	rf
eucalyptus	1, =	36.44	4.60	0.04	robustsc	noop	rf
first-order-theo	1, >	36.48	2.08	0.17	powertf	noop	rf
GestureSeg	1, =	32.39	1.09	0.30	robustsc	noop	rf
jm1	1, =	18.08	0.96	0.45	stdsc	noop	rf
profb	1, >	25.14	6.42	0.02	robustsc	noop	lsvc
plants-shape	1, >	30.35	3.86	0.12	stdsc	pca	rf
sick	4, <	6.54	0.49	0.07	robustsc	pca	gb
soybean	5, <	14.66	3.96	0.01	stdsc	noop	rbfsvc
Median	1	27.75	3.57	0.05			

Table 5. *All-one* (two-block)

					Block 1		Block 2		
Dataset	Rank	AvgErr	Std	Time	Sc1	Fx1	Sc2	Fx2	Lr
analcatdata	1, =	75.54	4.91	0.56	minmax	noop	powertf	factA	ada
breast-w	1, =	2.11	1.65	0.41	norm	noop	minmax	noop	rbfsvc
cmc	1, =	42.57	3.45	2.28	stdsc	noop	robustsc	noop	gb
credit-app	1, =	11.03	4.59	0.50	stdsc	noop	powertf	factA	lsvc
eucalyptus	1, =	33.40	6.13	2.78	stdsc	noop	noop	factA	gb
first-order	1, >	36.37	2.49	5.51	noop	ica	noop	noop	rf
GestureSeg	1, =	30.46	1.03	9.69	norm	ica	stdsc	noop	rf
jm1	1, =	17.67	0.79	16.62	robustsc	noop	norm	pca	rf
profb	1, >	24.69	5.29	0.61	stdsc	ica	stdsc	noop	lsvc
plants-shape	1, >	26.05	2.77	3.97	stdsc	factA	norm	pca	rf
sick	4, <	6.26	1.01	2.65	robustsc	pca	noop	noop	gb
soybean	5, <	13.24	2.62	0.57	robustsc	noop	noop	noop	lsvc
Median/Mode	1	25.37	2.70	2.62	stdsc	noop	noop	noop	rf

Table 6. *All-one with Surrogates*

	Baseline: All-one				Surrogate: Baseline+PRP				Surrogates: Baseline+PRP+LR			
Dataset	Rank	AvgErr	Std	Time	Rank	AvgErr	Std	Time	Rank	AvgErr	Std	Time
analcatdata	1, =	75.54	4.91	0.56	1, =	76.67	4.24	0.07	1, =	76.17	0.92	0.00
breast-w	1, =	02.11	1.65	0.41	1, =	02.49	1.70	0.02	1, =	02.33	2.49	0.00
cmc	1, =	42.57	3.45	2.28	1, =	42.84	4.64	0.07	1, =	44.13	3.68	0.01
credit-app	1, =	11.03	4.59	0.50	1, =	11.60	4.83	0.03	1, =	12.08	3.80	0.04
eucalyptus	1, =	33.40	6.13	2.78	1, =	36.37	7.03	0.16	1, =	35.63	6.27	0.22
first-order	1, >	36.37	2.49	5.51	1, =	36.73	2.18	0.13	1, =	36.63	2.07	0.28
GestureSeg	1, =	30.46	1.03	9.69	1, =	31.58	1.45	0.44	5, <	45.19	0.96	0.76
jm1	1, =	17.67	0.79	16.62	1, =	18.01	0.86	0.23	1, =	17.98	1.47	0.40
profb	1, >	24.69	5.29	0.61	1, >	24.86	6.02	0.02	1, >	24.41	4.42	0.04
plants-shape	1, >	26.05	2.77	3.97	1, >	32.06	2.93	0.17	3, <	44.31	5.53	0.61
soybean	5, <	13.24	2.62	0.57	5, <	14.29	3.55	0.02	5, <	14.29	4.83	0.03
Median	1	26.05	2.77	2.28	1	31.58	3.55	0.07	1	35.63	3.68	0.04

Furthermore, the dataset with worst runtime in *all-one* is the *jm1* which requires 0.45 hour or 27 minutes. This worst runtime is still significantly less than the 4-hour budget allotted to the 5 AutoML algorithms. It is interesting to note that most solutions in *all-one* do not employ any feature extraction but only scaling. We can consider the solutions in *all-one* to be sparsed relative to the dominant presence of `noop` in the feature extraction part of the one-block MLPL.

Inspired by these promising results, the next experiment applies the *all-one* strategy to two-block MLPL with the results summarized in Table 5. The *all-one* strategy achieves a 1 median rank similar to one-block *all-one* but the two-block pipeline has much lower cross-validation error compared to one-block pipeline (25.37 vs 27.75). Similar to one-block *all-one*, the two-block is significantly superior to the baseline algos in *first-order-theo*, *profb*, and *plants-shape* but significantly inferior in *sick* and *soybean* datasets. The best algos in the baseline has similar performance with the *all-one* strategy in the rest of the datasets. While the runtime median of *all-one* is 2.62 hours, 3 datasets require more than 4 hours to finish. In the future, it will be interesting to incorporate the time budget into the *all-one* strategy and run the other AutoMLs in the same machine to have fair comparison.

Using the *all-one* strategy, there is a trade-off in its applications to one-block or two-block pipelines. The former has significantly quick runtime but may not be optimal, while the latter may require relatively longer runtime than the former for some datasets but with a higher likelihood of being optimal.

Another interesting observation in Tables 2, 4, and 5 regarding the composition of their best solutions is the high occurrence of `noop`. We can consider their solutions to be sparsed because in many cases they do not fully utilize the 2 out 5 non-noop scalers and 2 out of 3 non-noop feature extractors in their optimal pipelines. This insight can be valuable by using the datamined signatures of the optimal solutions to predict structure complexity of the pipeline for an efficient search. By mapping data metafeatures and the corresponding optimal pipeline signatures, we can train a metalearner to guide which subset of pipelines or elements of the pipelines can be used as a starting point in search.

Table 6 summarizes the results of implementing these insights using PRP and LR surrogate models. The PRP-surrogate is trained to learn the mapping between dataset metafeatures and its corresponding optimal pipeline signature complexity. Pipeline complexity has 4 categories based on the presence of *noop*. Category 1 has zero or one *noop*, category 2 has two *noops*, etc. More *noops* imply less pipeline complexity. On the other hand, the LR-surrogate is used to learn the mapping between the dataset metafeatures with its corresponding optimal learner type: Ensemble vs SVM. Both surrogate models are trained using the OpenML-CC18 [7] datasets by extracting their metafeatures using *pymfe* [1]. The extracted datasets, results, and julia-based pseudocode for surrogate modeling can be found in the supplementary material submission of the paper.

We use the *all-one* strategy as the baseline and extended it with PRP and LR surrogates to find out their effect in prediction error and computation time. While there is a 5% increase in error by using PRP-surrogate, the median computation time went down from 2.28 hours to just 4.2 minutes. In spite of the 5% drop in accuracy, PRP-surrogate median rank of 1 still indicates superior performance relative to other AutoML approaches.

The exponential reduction in computation time of PRP-surrogate is due to the use of smaller search space due to the removal of unnecessary preprocessing elements. Incorporating further the LR-surrogate increases the median prediction error by another 4% but reduces the median computation time to just 2.4 minutes. The main trade-off in relying on more surrogates is the reduction of computation time at the expense of less accurate prediction. Depending on the requirements, using one or more surrogates can be necessary to speed-up computation if the corresponding drop in prediction accuracy is acceptable which can be true in some application domains.

Among the datasets, the one- and two-stage strategies have consistent low performance in both *sick* and *soybean* problems. Both datasets are characterized by large number of missing values compared to the rest of the datasets. There are 6064 *NA*s in *sick* and 2337 *NA*s in *soybean*. The poor performance of the two-stage strategies can be attributed to the non-implementation of imputation or interpolation filter in the `basepipeline` data cleaning routine. Future experiments will examine which imputation or interpolation routines can be used to improve AMLP solutions.

4. Conclusion

Based on the set of experiments we conducted, the *all-one* strategy provides the optimal solution in a significantly shorter duration relative to other AutoML algorithms. The results indicate that using a simple pipeline in the first stage consisting of one-hot encoded categorical features together with their numerical features under robust scaling provides a good representation of the dataset difficulty for evaluating the best matched optimal learner from the group of learners. The winning learner picked by the first stage can then be used to improve the solution further by looking for a more optimal match with the rest of the pipelines. The *all-one* strategy computation speed can be exponentially reduced by utilizing PRP and LR surrogates in exchange of lower accuracy which remains competitive relative to other AutoML approaches.

The easy and straightforward experimental setup in the implementation of these series of experiments can be attributed to the usability of *Julia* and AMLP toolbox. AMLP's support for a high-level description of the pipelines makes it easy to track which of the pipeline elements are dominant among the optimal solutions. These insights can be used in the development of the runtime search strategy in future algorithms. Data mining optimal pipelines become much easier because one can directly use these high-level expressions to perform text-mining, frequency pattern mining, text associations, and NLP on the elements and structure of the optimal pipeline solutions.

The high-level and easy composability in AMLP can result in the creation of complex pipelines that can barely be understood. For similar reasons that regularization and information criterion are employed in mathematical and statistical modeling, any AutoML implementation relying on AMLP and similar toolkits must incorporate strategies that take into account a good balance between pipeline complexity, accuracy, and explainability for easy scrutiny, comprehensibility, and verification bias of its solutions.

Julia's great allure can be attributed to its high-performance, interactive, and dynamic type system together with its *JIT* compilation feature. It addresses the two-language problem which is often encountered as a critical issue in other interactive languages that require performance-critical tasks. One major benefit of using *Julia* which may have profound impact in the scientific community is the significant increase of code reuse and sharing brought about

by just having one language implementation. Without resorting to low-level implementations (C/C++) in creating high-performance libraries, program readability is significantly improved in *Julia* ecosystem without sacrificing performance. The line that differentiates between users and developers in *Julia* becomes blurred because *Julia* makes it effortless to transition between the two groups. The ease of transition from being users to becoming developers in *Julia* will be a boon for increasing productivity in research, experimentation, development, and sharing of applications to wider fields of study.

It is with this background that AMLP was developed. It started as a hobby to learn *Julia* by implementing ML algorithms. Due to it's high performance, *Julia* provides a great playground to test ideas and gain results easily, which then triggers the desire to test more ideas, all achievable in a short amount of time. The resulting explorations can easily be turned into a package which is highly readable but at the same time highly efficient and fast. The open-source AMLP is the product of such good usability in Julia. By open sourcing AMLP, we want to pass on the benefits we received from the machine learning, open-source, and *Julia* communities. We are hoping that AMLP can serve as a ML playground to those who want to study the application of ML workflow and easily extend and benchmark their creations. The future goal is to help users of AMLP and the *Julia* community in general to transition from being consumers to becoming producers of ML ideas and share their creations with future generations and hopefully feed them back to AMLP or other similar toolkits.

5. References

[1] Edesio Alcobaça, Felipe Siqueira, Adriano Rivolli, Luís P. F. Garcia, Jefferson T. Oliva, and André C. P. L. F. de Carvalho. Mfe: Towards reproducible meta-feature extraction. *Journal of Machine Learning Research*, 21(111):1–5, 2020.

[2] Richard Loree Anderson. Recent advances in finding best operating conditions. *Journal of the American Statistical Association*, 48(264):789–798, 1953.

[3] Douglas Bates, Randy Lai, and Simon Byrne. RCall: Call r from julia. `https://github.com/JuliaInterop/RCall.jl`, 2015.

[4] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(1):281–305, 2012.

[5] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kegl. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2546–2554, 2011.

[6] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017. doi:10.1137/141000671.

[7] Bernd Bischl, Giuseppe Casalicchio, Matthias Feurer, Frank Hutter, Michel Lang, Rafael G. Mantovani, Jan N. van Rijn, and Joaquin Vanschoren. Openml benchmarking suites, 2019. 1708.03731.

[8] Anthony Blaom, Franz Kiraly, Thibaut Lienart, and Sebastian Vollmer. alan-turing-institute/mlj.jl: v0.5.3, November 2019. doi:10.5281/zenodo.3541506.

[9] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.

[10] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, and F. Hutter. Auto-sklearn: Efficient and robust automated machine learning. In *NeurIPS*, pages 2962–2970, 2015.

[11] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, and F. Hutter. *Auto-sklearn: Efficient and robust automated machine learning*, volume 8, chapter Challenges in Machine Learning – Methods, Systems, Challenges, pages 113–134. Springer, 2019.

[12] Martin Hirzel, Kiran Kate, Avraham Shinnar, Subhrajit Roy, and Parikshit Ram. Type-driven automated learning with Lale. *CoRR*, abs/1906.03957, May 2019.

[13] F. Hutter, H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization (LION)*, pages 507–523, 2011.

[14] S. Jenkins. Orchestra: Heterogeneous ensemble learning for julia. `https://github.com/svs14/Orchestra.jl`, 2014.

[15] Steven G. Johnson. PyCall: Calling python functions from the julia language. `https://github.com/JuliaPy/PyCall.jl`, 2013.

[16] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice-Hall, 1984.

[17] Max Kuhn. Building predictive models in r using the caret package. *Journal of Statistical Software, Articles*, 28(5):1–26, 2008. doi:10.18637/jss.v028.i05.

[18] S. Liu, P. Ram, D. Vijaykeerthy, D. Bouneffouf, G. Bramble, H. Samulowitz, D. Wang, A. Conn, and A. Gray. An ADMM based framework for AutoML pipeline configuration. In *AAAI*, 2020. Their preprint available at `https://arxiv.org/pdf/1905.00424.pdf`.

[19] Tien-Dung Nguyen, Tomasz Maszczyk, Katarzyna Musial, Marc-Andre Zöller, and Bogdan Gabrys. Avatar - machine learning pipeline evaluation using surrogate model. In *Advances in Intelligent Data Analysis XVIII*, pages 352–365. Springer International Publishing, 2020.

[20] R. Olson, N. Bartley, R. Urbanowicz, and J. Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Genetic and Evolutionary Computation Conference (GECO)*, pages 485–492, 2016.

[21] F. Pedregosa, G. Varoquaux, and A. Gramfort. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(1):2825–2830, 2011.

[22] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013.

[23] H. Stern and R Lock. Pro football scores. `http://lib.stat.cmu.edu/datasets/profb`, 2014.

[24] Thomas Swearingen, Will Drevo, Bennett Cyphers, Alfredo Cuesta-Infante, Arun Ross, and Kalyan Veeramachaneni. Atm: A distributed, collaborative, scalable system for automated machine learning. *IEEE International Conference on Big Data*, pages 151–162, 2017.

[25] C. Thornton, F. Hutter, H. Hoos, and K. Leyton-Brown. Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In *KDD*, pages 847–855, 2013.

[26] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[27] J. Vanschoren, J.N. van Rijn, B. Bischl, and L. Torgo. Openml: Networked science in machine learning. *SIGKDD Expl.*, 15(2):49–60, 2013.

[28] M.A. Zoller and M. F. Huber. Survey on automated machine learning. `https://arxiv.org/pdf/1904.12054.pdf`, 4 2019.